DISEÑO Y DESARROLLO DE GUÍAS DE LABORATORIO PARA EL CURSO DE SISTEMAS OPERATIVOS DE LA CUTB

NARLINDA ESPINOSA CANTILLO
YULISSA LEONOR MORA GONZÁLEZ

TECNOLÓGICA DE BOLÍVAR
INSTITUCIÓN UNIVERSITARIA

FACULTAD DE INGENIERÍA DE SISTEMAS
CARTAGENA DE INDIAS D.T. y C.
2003

DISEÑO Y DESARROLLO DE GUÍAS DE LABORATORIO PARA EL CURSO DE SISTEMAS OPERATIVOS DE LA CUTB

NARLINDA ESPINOSA CANTILLO
YULISSA LEONOR MORA GONZÁLEZ

TRABAJO DE GRADO

Director
GIOVANNY RAFAEL VÁSQUEZ MENDOZA
Ingeniero de Sistemas

TECNOLÓGICA DE BOLÍVAR
INSTITUCIÓN UNIVERSITARIA

FACULTAD DE INGENIERÍA DE SISTEMAS
CARTAGENA DE INDIAS D.T. y C.
2003

Cartagena, Mayo 23 de 2003

Señores:
COMITE DE EVALUACIÓN DE PROYECTOS
CORPORACION UNIVERSITARIA TECNOLÓGICA DE BOLÍVAR
L.C.

Respetados Señores:

A petición de los estudiantes **YULISSA LEONOR MORA GONZÁLEZ Y NARLINDA ESPINOSA CANTILLO,** he aceptado dirigir el proyecto **"DISEÑO Y DESARROLLO DE GUÍAS DE LABORATORIO PARA EL CURSO DE SISTEMAS OPERATIVOS DE LA CUTB",** como trabajo de grado para optar el título de Ingenieros de Sistemas.

Atentamente,

_____

Ing. GIOVANNY VÁSQUEZ MENDOZA

Cartagena, Mayo 23 de 2003

Señores:
COMITE DE EVALUACIÓN DE PROYECTOS
CORPORACION UNIVERSITARIA TECNOLÓGICA DE BOLÍVAR
L.C.

Respetados Señores:

Por medio de la presente nos permitimos presentar a ustedes, para que sea puesto a consideración, el estudio y aprobación del trabajo de grado que lleva por nombre "**DISEÑO Y DESARROLLO DE GUÍAS DE LABORATORIO PARA EL CURSO DE SISTEMAS OPERATIVOS DE LA CUTB",** como trabajo de grado para optar el título de Ingenieros de Sistemas.

Agradeciendo de antemano la atención prestada.

Atentamente,

_____	_____
Yulissa Leonor Mora González	Narlinda Espinosa Cantillo
Estudiante de Ingeniería de Sistemas	Estudiante de Ingeniería de Sistemas

AUTORIZACIÓN

Cartagena de Indias, D.T.C.H.,

Nosotras Yulissa Leonor Mora González y Narlinda Espinosa Cantillo, identificadas con número de cédula 45.523.840 de Cartagena y 45.521.638 de Cartagena respectivamente, autorizamos a la Corporación Universitaria Tecnológica de Bolívar para hacer uso de nuestro trabajo de grado y publicarlo en el catálogo online de la Biblioteca.

_____ _____

Yulissa Leonor Mora González
C.C. 45.523.840 de Cartagena

Narlinda Espinosa Cantillo
C.C. 45.521.638 de Cartagena

_____

_____

_____

_____

_____

_____

Firma del presidente del jurado

_____

Firma del jurado

_____

Firma del jurado

Cartagena, mayo 23 de 2003

A DIOS por mostrarme el camino
para alcanzar mis metas, a mis padres
y hermanos por apoyarme siempre,
sin ellos este sueño no sería realidad.

YULISSA MORA GONZÁLEZ

Especialmente a Dios por iluminarme y ayudarme siempre. A mi mamá, mi papá y mi hermano por estar a mi lado y apoyarme en todo momento. A mi novio, una personita muy especial, por acompañarme y darme ánimo cuando más lo necesité.

NARLINDA ESPINOSA CANTILLO.

CONTENIDO

pág.

FIGURAS

pág.

GLOSARIO

ADMINISTRACIÓN DE MEMORIA: módulo o parte del sistema operativo que determina qué parte de la memoria asignar a los procesos y qué parte de los mismos debe permanecer en disco o cuáles partes deben abandonar temporalmente la memoria para ceder su espacio a otros que lo requieran en un momento determinado.

ARCHIVO: conjunto de registros relacionados que se tratan como una unidad.

BLOQUE: división física del espacio de direcciones reales.

EXCEPCIONES: sucesos que se pueden dar durante la ejecución de un proceso y que se salen de lo normal.

INTERRUPCIÓN: suspensión de un proceso, tal como la ejecución de un programa de computadora, originada por un suceso externo a dicho proceso y llevada a cabo de forma que el proceso pueda reanudarse.

MEMORIA VIRTUAL: método que consiste en destinar parte del disco para almacenar aquellas partes de los programas que no se estén utilizando de manera que parezca que se excede la cantidad de memoria física disponible.

MENSAJE: bloque de información que puede intercambiarse entre los procesos como medio de comunicación.

MULTIPROGRAMACIÓN: consiste en tener varios procesos listos en memoria al tiempo, de forma que cada uno de ellos recibe un tiempo de procesamiento.

PÁGINA: división lógica de un proceso en tamaños iguales.

PROCESOS CONCURRENTES: procesos que tienen lugar en un intervalo común de tiempo los cuales pueden necesitar compartir recursos alternativamente.

PROCESOS COOPERATIVOS: procesos diseñados para consultar el estado del recurso que comparten y esperar que quede libre.

PROCESO SOLICITANTE: trabajo que requiere espacio en memoria para su ejecución.

SECCIÓN CRÍTICA: segmento de código de un proceso en el cual se pueden estar modificando variables comunes, actualizando tablas, escribiendo archivos y realizando operaciones de este tipo.

SEGMENTO: módulo independiente de un proceso. Es un componente lógico de tamaño variable de un proceso.

SEMÁFORO: un valor entero usado para la señalización entre procesos. Sólo se pueden utilizar tres operaciones sobre un semáforo, todas las cuales son atómicas: inicializar, decrementar e incrementar.

SEÑAL: mecanismo de software que informa a un proceso del acontecimiento de un suceso asíncrono.

SINCRONIZACIÓN: mecanismo mediante el cual dos o más procesos pueden utilizar un recurso compartido en forma alternada.

SISTEMA OPERATIVO: conjunto de programas y funciones que ocultan los detalles de hardware, ofreciendo al usuario una vía sencilla y flexible de acceso al mismo. A la vez que administra los recursos ofrecidos por el hardware para alcanzar un eficaz rendimiento de los mismos.

RESUMEN

El trabajo investigativo realizado sirve como complemento para la labor realizada por el docente en el momento de transmitir los conocimientos acerca de Sistemas Operativos ya que con estas Guías de Laboratorio el estudiante cuenta con unas bases para realizar prácticas a nivel de programación donde se da cuenta de la forma en que el sistema operativo opera desde el kernel o núcleo del sistema para poder realizar las diferentes tareas que el usuario final solicita. Además, el estudiante analiza y estudia los temas tomando como base dos sistemas operativos en particular Windows y Linux. En estos existen diferentes llamadas al sistema y comandos que le permiten al estudiante comprender la labor que el sistema operativo realiza como administrador de recursos.

Se elaboraron cinco guías de laboratorios que corresponden a los siguientes temas: instalación de los sistemas operativos Linux y Windows, control y administración de procesos, sincronización de procesos, comunicación de procesos y protección.

En cada guía se muestra al estudiante un marco teórico con información básica que necesita saber para poder realizar la práctica. El estudiante deberá complementar esta información investigando en diferentes fuentes, seguido a esto podrá analizar programas que han sido realizados con base a la teoría suministrada anteriormente y luego estará en capacidad de realizar prácticas correspondientes a cada uno de los temas utilizando los conceptos adquiridos. Además, finalmente podrá identificar la manera como cada uno de los sistemas operativos tratados lleva a cabo sus funciones y realizará conclusiones relevantes.

En cada guía el docente hará una retroalimentación para aclarar dudas y comprobar si los objetivos propuestos se cumplieron a cabalidad.

INTRODUCCIÓN

Actualmente en la Institución Universitaria Tecnológica de Bolívar, la materia Sistemas Operativos se imparte de forma teórica. Debido a esto, el estudiante no tiene la oportunidad de afianzar y verificar qué tanto ha comprendido los conocimientos adquiridos durante el desarrollo de la materia.

A pesar de que se realizan actividades tales como mapas conceptuales, exámenes, programas. Estos últimos no incluyen todos los conceptos experimentales de un tema en específico ya que tomaría mucho tiempo profundizar cada uno de los conceptos que abarcan los temas, es por esto que surgió la necesidad de realizar un trabajo donde el estudiante pueda contar con una guía básica que le ayude a complementar todos los conceptos y a la vez aplicarlos.

Hemos elegido los sistemas operativos Linux y Windows como tema de estudio para este trabajo investigativo ya que son los mas comúnmente usados por la mayoría de los usuarios y así poder conocer más a fondo su funcionamiento y determinar cual es el más adecuado para satisfacer nuestras necesidades.

Los temas tratados en el siguiente trabajo son: Instalación de los sistemas operativos Linux y Windows NT, Administración de Procesos, Sincronización de Procesos, Comunicación de Procesos y Protección.

OBJETIVOS

Objetivo General:

Diseñar y desarrollar guías de laboratorio para el curso de sistemas operativos, utilizando como plataforma Windows y Linux y como herramienta de programación C/C++, que faciliten el aprendizaje de los conceptos teóricos expuestos en clase.

Objetivos Específicos:

- ✓ Elaborar una guía de instalación para que el estudiante conozca los recursos necesarios que un sistema operativo requiere para su funcionamiento.
- ✓ Elaborar una guía de control y administración de procesos donde el estudiante practique las diferentes operaciones que puede realizar con los procesos.
- ✓ Elaborar una guía que muestre los diferentes procedimientos para la sincronización de procesos.
- ✓ Elaborar una guía para ilustrar cómo se realiza la comunicación entre procesos en una misma máquina.
- ✓ Elaborar una guía de protección para que el usuario del sistema operativo pueda proteger sus recursos de otros usuarios.

MARCO TEÓRICO

1. ¿QUÉ ES UN SISTEMA OPERATIVO?

Un Sistema Operativo es un programa que actúa como intermediario entre el usuario y el hardware del computador y su propósito es proporcionar el entorno en el cual el usuario pueda ejecutar programas.

1.1 Objetivos de los Sistemas Operativos

- ✓ Comodidad. Hace que el computador sea más fácil de utilizar.
- ✓ Eficiencia. Permite que los recursos del sistema se aprovechen eficientemente.
- ✓ Capacidad de evolución. Debe ser construido de tal forma que permita el desarrollo efectivo, la verificación y la introducción de nuevas funciones en el sistema y, a su vez, no interferir con los servicios que brinda.

1.2 Componentes del sistema operativo

Un sistema operativo está formado por tres capas: el núcleo, los servicios y el intérprete de mandatos o *shell.*

El núcleo es la parte del sistema operativo que interacciona directamente con el hardware de la máquina. Las funciones del núcleo se centran en la gestión de

recursos, como el procesador, tratamiento de interrupciones y las funciones básicas de manipulación de memoria.

Los servicios se suelen agrupar según su funcionalidad en varios componentes, cada uno de los cuales se ocupa de las siguientes funciones:

- ✓ Gestión de procesos.
- ✓ Gestión de memoria.
- ✓ Gestión de las E / S.
- ✓ Gestión de archivos y directorios.
- ✓ Comunicación y sincronización entre procesos.
- ✓ Seguridad y protección.

Todos estos componentes ofrecen una serie de servicios a través de una interfaz de llamadas al sistema.

La capa de intérprete de mandatos o *shell* suministra una interfaz a través de la cual el usuario puede dialogar de forma interactiva con la computadora.

1.3 LINUX

Linux es un poderoso Sistema Operativo creado en 1991 por Linus Torvalds y actualmente es sostenido por el trabajo de varios grupos de programadores distribuidos en varias partes del mundo.

GNU/Linux es un sistema operativo gratuito y de libre distribución bajo las condiciones que establece la licencia GPL (*GNU Public License*). Tiene todas las características que uno puede esperar de un sistema Unix moderno: multitarea

real, memoria virtual, bibliotecas compartidas, carga por demanda, soporte de redes TCP/IP, entre muchas otras funcionalidades.

1.3.1 Características

- ✓ Multitarea: varios procesos se ejecutan al mismo tiempo.
- ✓ Multiusuario: varios usuarios utilizan la misma maquina al mismo tiempo.
- ✓ Multiplataforma: corre sobre diversas arquitecturas de CPU.
- ✓ Multiprocesador: ofrece soporte para plataformas con varios procesadores.
- ✓ Incluye protección de memoria entre procesos, de forma que un único programa no puede "colgar" el sistema al completo.
- ✓ Mejor aprovechamiento de la memoria, mediante la lectura desde el disco de aquellas partes de un programa que se están ejecutando, así como a través de la compartición de la misma zona de memoria para varios procesos simultáneos.
- ✓ Control completo de tareas y procesos.
- ✓ Soporte multinacional.
- ✓ Interacción sencilla y transparente con otros sistemas, como MS-DOS, Windows 9X/Me/XP, Windows NT/2000, OS/2, Novell y, por supuesto, otros UNIX.
- ✓ Soporte de red: TCP/IP, Novell, NT, Appletalk.
- ✓ Multitud de software disponible, tanto comercial como gratuito.
- ✓ Compatibilidad con todo tipo de hardware.
- ✓ Actualizaciones continuas tanto del núcleo como de "drivers" de soporte de nuevas tecnologías.
- ✓ Es más estable que otros sistemas operativos. Está respaldado por 30 años de experiencia y estabilidad de Unix, del cual deriva su arquitectura.
- ✓ Puede operar tanto en 32 como 64 bits reales según la plataforma usada (Intel, Alpha, etc.).

- ✓ Es altamente efectivo en redes cliente-servidor (intranet y extranets) y como servidor web.
- ✓ Posee varios entornos gráficos de ventanas semejantes a Windows o Macintosh.
- ✓ Lee perfectamente otros sistemas de archivos como FAT16 y 32, NTFS, HPFS, ISO9660, Joliet, etc.

1.3.2 ¿Para que sirve?

- ✓ Estación de Trabajo Personal: Todas sus características la convierten en una maquina UNIX poderosa.
- ✓ Plataforma de Desarrollo Unix: Linux posee compiladores para la gran mayoría de lenguajes existentes como C, C++, F77, Java, ADA, Modula 2 y 3, Pascal, Tcl/Tk, Scheme, Smalltalk, etc, sin costo alguno. Además de librerías gráficas, de computación paralela y distribuida, y librerías científica para múltiples aplicaciones.
- ✓ Plataforma de Desarrollo Comercial: Bases de datos Clipper (dBase y Fox), Oracle e Informix.
- ✓ Servidor Internet: WWW, mail, ftp, news, gopher y todos los posibles servicios a través de la red. Combinado con conexiones remotas se convierte en un excelente proveedor remoto de Internet.
- ✓ Servidor de Terminales. FAX, MODEM, Líneas telefónicas, y equipos seriales, para proveer acceso directo o remoto a recursos locales.
- ✓ Servidor de comunicaciones: puede trabajar como gateway, firewall, proxy o servidor de módems, entre otros, ofreciendo además otros servicios populares como caché WWW o autentificación de usuarios.
- ✓ Servidor de Intranet: ficheros, web, correo electrónico, DNS, gestión de bases de datos, etc.

1.4 WINDOWS NT

Windows NT se trata de un sistema operativo de red de multitarea preferente, de 32 bits con alta seguridad y servicios de red.

1.4.1 Características

Fiabilidad

Una fiabilidad superior permite a Windows NT ser usado como base para aplicaciones críticas. Está especialmente indicado para estaciones de trabajo y servidores de red, los cuales necesitan el máximo rendimiento.

Rendimiento

Windows NT fue también diseñado para ser un sistema operativo de alto rendimiento. Características que contribuyen a esto son:

- ✓ **Diseño real de 32 bits**. Todo el código de Windows NT en 32 bits, lo que le proporciona mucha más velocidad que otros operativos escritos con tecnología de 16 bits.
- ✓ **Características de multitarea y multiproceso**. Windows NT proporciona multitarea preferente, lo que permite una ejecución de todos los procesos, y además soporta varias CPU lo que es rendimiento.
- ✓ **Soporta de CPU RISC**. Windows NT no sólo soporta CPU basadas en INTEL, sino en diferentes tipos de CPU como Poder PC, DEC Palpa y MAC.

Portabilidad

La portabilidad en Windows NT significa que este sistema operativo puede usarse con diferentes tipos de hardware sin necesidad de reescribir el código completamente de nuevo. Windows NT proporciona las siguientes características en portabilidad:

- ✓ **Arquitectura de micro-kernel modular**. Windows NT posee un diseño modular lo que le proporciona independencia del hardware.
- ✓ **Sistemas de archivos configurables**. Otra de las características de Windows NT que aumenta sus posibilidades de portabilidad es su capacidad de soportar diferentes sistemas de archivos. Actualmente soporta FAT usados en sistemas DOS, HPFS usados en sistemas OS/2, NTF sistema de archivos de NT, CDFS sistema de archivo de CD-ROM y sistemas de archivos Macintosh.

Compatibilidad

Es un elemento clave para la aceptación de un sistema operativo, es una capacidad de trabajar con las aplicaciones ya existentes. Microsoft ha diseñado Windows NT para que sea capaz de ejecutar una amplia variedad de diferentes aplicaciones e interactúe con diferentes sistemas operativos.

- ✓ **Diseño de aplicaciones como subsistemas**. Windows NT soporta aplicaciones MS-DOS, Windows 3-x (16 bits), Windows 95, POSIX y OS/2 1.x. Otra vez, el diseñado modular de Windows NT lo hace posible, simplemente añadiendo nuevos subsistemas.

- ✓ **Interfaz explorador de Windows 95**. Se ha mantenido el interfaz gráfico tan esperado en Windows 95. Es realmente un replica exacta a no ser por algunas carpetas que faltan y otras que se han añadido. De hecho cuando el usuario migra Windows 95 a Windows NT, puede optar por mantener el escritorio original.
- ✓ **Interoperatividad con UNIX**. Windows NT se comunica con sistemas UNIX a través del protocolo TCP/IP. También soporta impresión TCP/IP e incluye aplicaciones de conectividad básicas como por ejemplo FTP, Telnet y Ping.

Escalabilidad

Otro aspecto importante de Windows NT es que es un sistema operativo escalable. Esto quiere decir que puede ser usado por un amplio abanico de sistemas, desde un ordenador personal hasta grandes sistemas con múltiples procesadores. Estos sistemas pueden tener muy pocas cosas en común o casi ninguna. Una pequeña lista de las características de escalabilidad son estas:

- ✓ **Soporte multiplataforma**. Debido a que la arquitectura de micro-kernel está en capas de abstracción de hardware (HALL), Windows NT es capaz de soportar los más potentes procesadores desarrollados en el futuro.
- ✓ **Soporte multiprocesador**. Soporta múltiples CPU en sistemas, lo que le proporciona un funcionamiento más eficiente a medida que se aumentan los procesadores.

Seguridad

Otras de las características más importantes de Windows NT son sus características de seguridad.

- ✓ **Modelo de seguridad de dominio**. El modelo de seguridad de dominio es un sofisticado sistema de acceso a la red, de manera que se controla perfectamente por donde los recursos de red que un usuario puede utilizar. Unos servidores especiales llamados controladores de dominio son los encargados de realizar todo el trabajo de autentificación de usuarios. La información de seguridad se guarda en una base de datos llamada SAM (Security Account Manager).
- ✓ **Sistema de archivos NTFS**. Es un sistema de archivos propio de Windows NT, que complementa la seguridad del sistema. Permite a los administradores de la red el control de utilizar una variedad de acceso a la red para grupos o usuarios.
- ✓ **Características de tolerancia a fallos**. Windows NT incluye importantes características de tolerancia a fallos. La primera característica importante es el soporte RAID (Redundant Array Of Inexpensive Disk), la cual usa una tecnología parecida al disk mirroring. Si se produce un fallo en el disco, gracias al RAID la información se puede obtener de nuevo. Otra característica importante de tolerancia de fallos es el soporte de UPS, unidades de alimentación interrumpida. Windows NT detectaría una caída de tensión en la red y conmutaría inmediatamente a la UPS.

2. CONTROL Y ADMINISTRACIÓN DE PROCESOS

2.1 Procesos

Un proceso es un programa en ejecución el cu al necesita ciertos recursos para llevar a cabo su tarea. El sistema operativo mantiene por cada proceso una serie de estructuras de información que permite identificar las características de éste así como los recursos que tiene asignados. Una parte muy importante de esta estructura es el *bloque de control del proceso (BCP)* que incluye el estado de los registros de los procesos, cuando éste no se está ejecutando.

Dentro de las funciones realizadas por un sistema operativo para la gestión de los procesos p odemos listar las siguientes:

1. Crear y eliminar los procesos.
2. Controlar el avance de los procesos.
3. Actuar frente a las condiciones excepcionales presentadas durante la ejecución de un proceso, incluidas interrupciones y operaciones aritméticas.
4. Asignar los recursos hardware entre los diferentes procesos.
5. Proveer los medios para la comunicación entre los procesos.

2.1.1 Llamadas al sistema para la administración de procesos en Linux

Crear Procesos

fork( ): Cuando un proceso usa la llamada a sistema fork() se crea en memoria una copia exacta del mismo y las estructuras de tarea son esencialmente idénticas. El proceso creador se denomina proceso-padre y el proceso creado proceso-hijo, por lo tanto cada proceso tiene un único proceso p adre y cero o varios procesos hijos.

El proceso padre se diferencia del proceso hijo por los valores PID (*Process ID)* y PPID (*Parent Process ID*) y por el valor de retorno de la llamada fork(), que será -1 en caso de fallo, pero si tiene éxito al proceso hijo retornará el valor 0 y al proceso padre retornará el PID del proceso hijo.
vfork( ): Esta llamada permite crear procesos sin necesidad de copiar todo el espacio de direcciones del padre.

Proceso en espera

wait( ) : La llamada al sistema wait detiene al proceso que la invoca hasta que un hijo de éste termine. Si wait regresa debido a la terminación de un hijo, el valor devuelto es igual al PID del proceso que finaliza, de lo contrario devuelve -1.
waitpid( ): La llamada waitpid proporciona un método para esperar por un hijo en particular. Si pid es igual a -1, waitpid espera por cualquier proceso. Si pid es positivo entonces espera al hijo cuyo PID es pid.

Ejecutar código de un proceso

El servicio **exec( )** de POSIX tiene por objetivo cambiar el programa que está ejecutando un proceso. Existe una familia de funciones exec: ***excl., execv, execle, execve, execlp, execvp.***

Finalizar un proceso

El proceso ejecuta su última instrucción y le pide al sistema operativo que lo borre, o ejecuta la llamada al sistema **exit( ).** En ambos casos sistema operativo libera los recursos utilizados por el.

Identificación de procesos

getpid( ): Obtiene el identificador del proceso.
getppid( ): Obtiene el identificador del proceso padre.

2.1.2 Comandos para la administración de procesos en Linux

ps (process status): muestra los procesos que se están ejecutando y que fueron arrancados por el usuario actual**.** Los campos de información más importantes desplegados por ps para cada proceso son: Usuario (USER), identificadores de proceso (PID, PPID), Uso de recursos reciente y acumulado (%CPU, %MEM, TIME), Estado del proceso (STAT, S) y comando invocado (COMMAND).
ps –fea despliega todos los procesos activos.

ps -aux : Despliega todos los procesos del sistema, con nombre y tiempo de inicio.

pstree: en algunos sistemas está disponible el comando pstree, que lista los procesos y sus descendientes en forma de árbol. Esto permite visualizar rápidamente los procesos que están corriendo en el sistema.

top: esta herramienta monitorea varios recursos del sistema y tiene un carácter dinámico, muestra uso de CPU por proceso, cantidad de memoria, tiempo desde su inicio, etc.

nice: asigna la prioridad a los procesos, los valores nice oscilan desde -20 (mas prioridad) a 20 (menos prioridad), normalmente este valor se hereda del proceso padre.

kill: comando que se utiliza para eliminar un proceso.

kill -KILL <pid> : señala al proceso con numero <pid>,que termine de inmediato, el proceso es terminado abruptamente.

kill -TERM <pid> : señala al proceso con numero <pid>,que debe de terminar, a diferencia de -KILL , esta opción da la oportunidad al proceso de terminar.

kill -STOP <pid> : señala al proceso con numero <pid>, que pare momentáneamente.

kill -CONT <pid> : señala al proceso con número <pid>, que continué, este comando se utiliza para reanudar un proceso que le fue aplicado –STOP.

kill -INT <pid> : interrumpe al proceso con numero <pid>

bg: ordena a un proceso que está parado en segundo plano, que continúe ejecutándose en segundo plano.

fg: ordena a un proceso que está en segundo plano (parado o en funcionamiento) que vuelva al primer plano.

jobs: se usa para comprobar cuantos procesos se están ejecutando en segundo plano.

&: se usa en la línea de comando y con él le indicamos al sistema que la línea que se va a ejecutar deberá de ser puesta a trabajar en segundo plano. El "&" se pone al final de la línea a ejecutar.

Ctrl+Z: Envía al segundo plano el programa que se esté ejecutando y lo detiene.

vmstat: el comando vmstat reporta varias estadísticas que mantiene el kernel sobre los procesos, la memoria y otros recursos del sistema. Alguna de la información reportada por vmstat es la siguiente:

- ✓ Cantidad de procesos en diferentes estados (listo para correr, bloqueado, "swapeado" a disco.
- ✓ Valores totales de memoria asignada a procesos y libre.
- ✓ Estadísticas sobre paginado de memoria (page faults, paginas llevadas o traídas a disco, páginas liberadas)
- ✓ Operaciones de disco
- ✓ Uso de CPU reciente clasificado en inactivo, ejecutando en modo usuario y ejecutando en modo kernel.

2.1.3 Servicios POSIX para la planificación de procesos

Modificar los parámetros de planificación

sched_setparam( ): Modifica la prioridad de un proceso.

sched_scheduler( ): Modifica la prioridad y política de planificación de un proceso.

Obtener los parámetros de planificación

sched_getparam( ): Devuelve la prioridad del proceso.

sched_getscheduler( ): Devuelve la política de planificación del proceso

2.1.4 Llamadas al sistema para la administración de procesos en Windows

Creación de un Proceso

En Win32 los procesos se crean mediante la llamada CreateProcess. El prototipo de esta función es el siguiente:

```
BOOL CreateProcess (
                    LPCTSTR lpszImageName,
                    LPTSTR lpszCommandLine,
                    LPSECURITY_ATTRIBUTES lpsaProcess,
                    LPSECURITY_ATTRIBUTES lpsaThread,
                    BOOL fInheritHandles,
                    DWORD fdwCreate,
                    LPVOID lpvEnvironment,
                    LPCTSTR lpszCurdir,
                    LPSTARTUPINFO lpsiStartInfo,
                    LPPROCESS_INFORMATION lppiProcInfo);
```

Esta función crea un nuevo proceso y su proceso ligero principal. El nuevo proceso ejecuta el archivo ejecutable especificado en lpszImageName. Esta cadena puede especificar el nombre de un archivo con camino absoluto o relativo, pero la función no utilizará el camino de búsqueda. Si lpszImageName es NULL, se utilizará como nombre de archivo ejecutable la primera cadena delimitada por blancos del argumento lpszCommandLine.

El argumento lpszCommandLine especifica la línea de comandos a ejecutar, incluyendo el nombre del programa a ejecutar. Si su valor es NULL, la función utilizará la cadena apuntada por lpszImageName como línea de mandatos.

El argumento lpsaProcess determina si el manejador asociado al proceso creado y devuelto por la función puede ser heredado por otros procesos hijos. Si es NULL, el manejador no puede heredarse. Lo mismo se aplica al manejador lpsaThread, pero relativo al manejador del proceso ligero principal devuelto por la función.

El argumento fInheritHandles indica si el Nuevo proceso hereda los manejadores que mantiene el proceso que realiza la llamada. El argumento fdwCreate puede combinar varios valores que determinan la prioridad y la creación del nuevo proceso. Algunos de estos valores son:

- ✓ CREATE_SUSPEND: el proceso ligero principal del proceso se crea en estado suspendido y sólo se ejecutará cuando se llame a la función ResumeThread.
- ✓ DETACHED_PROCESS: para procesos con consola, indica que el nuevo proceso no tenga acceso a la consola del proceso padre.
- ✓ CREATE_NEW_CONSOLE: el nuevo proceso tendrá una nueva consola asociada y no heredará la del padre. Este valor no puede utilizarse con el anterior.
- ✓ NORMAL_PRIORITY_CLASS: indica un proceso sin necesidades especiales de planificación.
- ✓ HIGH_PRIORITY_CLASS: indica que el proceso se cree con una prioridad alta de planificación.

- ✓ IDLE_PRIORITY_CLASS: especifica que los procesos ligeros del proceso sólo ejecuten cuando no haya ningún otro proceso ejecutando en el sistema.
- ✓ REALTIME_PRIORITY_CLASS: indica un proceso con la mayor prioridad posible.

El parámetro lpvEnvironment apunta al bloque del entorno del nuevo proceso. Si el valor es NULL, el nuevo proceso obtiene el entorno del proceso que realiza la llamada.

El argumento lpszCurdir apunta a una cadena de caracteres que indica el directorio actual de trabajo para el nuevo proceso.

El parámetro lpStartupInfo apunta a una estructura de tipo STARTPINFO que especifica la apariencia de la ventana asociada al nuevo proceso.

Por último, el argumento lpProcessInformation, puntero a una estructura de tipo PROCESS_INFORMATION, se almacenará información sobre el nuevo proceso creado.

Terminación de un Proceso

Los servicios relacionados con la terminación de procesos se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso y servicios para esperar la terminación de un proceso. Estos servicios se describen a continuación:

1. Terminar la ejecución de un proceso: Un proceso puede finalizar su ejecución de forma voluntaria de tres formas:
   - ✓ Ejecutando dentro de la función *main* la sentencia *return.*
   - ✓ Ejecutando el servicio ExitProcess.
   - ✓ La función de la biblioteca de C *exit.* Esta función es similar a ExitProcess.

   El prototipo de la función ExitProcess es el siguiente:

   VOID ExitProcess (UINT nExitCode);

   La llamada cierra todos los manejadores abiertos por el proceso y especifica el código de salida del proceso. Este código lo puede obtener oro proceso mediante el siguiente servicio:

   BOOL GetExitCodeProcess (HANDLE hprocess, LPDWORD lpdwExitCode);

   Esta función devuelve el código de terminación del proceso con manejador hProcess. El proceso especificado por hProcess debe tener el acceso

PROCESS_QUERY_INFORMATION. Si el proceso todavía no ha terminado, la función devuelve en lpdwExitCode el valor STILL_ALIVE, en caso contrario almacenará en este valor el código de terminación.

Además, un proceso puede finalizar la ejecución de otro mediante el servicio:

BOOL TerminateProcess (HANDLE hProcess, UINT uExitCode);

Este servicio aborta la ejecución del proceso con manejador hProcess. El código de terminación para el proceso vendrá dado por el argumento uExitCode. La función devuelve TRUE si se ejecuta con éxito.

2. Esperar por la finalización de un proceso

En Win32 un proceso puede esperar la terminación de cualquier otro proceso siempre que tenga permisos para ello y disponga del manejador correspondiente. Para ello, se utilizan las funciones de espera de propósito general. Estas funciones son:

DWORD WairForSingleObject (HANDLE hObject, DWORD dwTimeOut);

DWORD WaitForMultipleObjects (DWORD cObjects, LPHANDLE lphObjects, BOOL fWaitAll, DWORD dwTimOt);

La primera función bloquea al proceso hasta que el proceso con manejador hObject finalice su ejecución. El argumento dwTimeOut específica el tiempo máximo de bloqueo expresado en milisegundos. Un valor de 0 hace que la función vuelva inmediatamente después de comprobar si el proceso finalizó la ejecución. Si el valor es INFINITE, la función bloquea el proceso hasta que el proceso acabe su ejecución.

La segunda función permite esperar la terminación de varios procesos. El argumento cObjects especifica el número de procesos (el tamaño del vector lphObjects) por los que se desea esperar.

El argumento lphObjects es un vector con los manejadores de los procesos sobre los que se quiere esperar. Si el parámetro fWaitAll es TRUE, entonces la función debe esperar por todos los procesos, en caso contrario la función vuelve tan pronto como un proceso haya acabado. El parámetro dwTimeOut tiene el significado descrito anteriormente.

Estas funciones, aplicadas a procesos, pueden devolver los siguientes valores:

- ✓ WAIT_OBJECT_0: Indica que el proceso terminó en el caso de la función WaitForSingleObject, o todos los procesos terminaron si en WaitForMultipleObjects el parámetro WaitAll es TRUE.
- ✓ WAIT_OBJECT_0+n, donde 0 <= n<= cObjects. Restando este valor de WAIT_OBJECT_0 se puede determinar el número de procesos que han acabado.

- ✓ WAIT_TIMEOUT: Indica que el tiempo de espera expiró antes de que algún proceso acabara.

Las funciones devuelven 0xFFFFFFFF en caso de error.

En Windows NT la unidad básica de ejecución es el proceso ligero y, por tanto, la planificación se realiza sobre este tipo de procesos. Windows NT implementa una planificación cíclica (Round Robin) con prioridades y con expulsión. Existen 32 niveles de prioridad, de 0 a 31, siendo 31 el nivel de prioridad máximo. Estos niveles se dividen en tres categorías:

- ✓ Dieciséis niveles con prioridades de tiempo real (niveles 16 a 31).
- ✓ Quince niveles con prioridades variables (niveles 1 al 15).
- ✓ Un nivel de sistema (0).

Todos los procesos ligeros en el mismo nivel se ejecutan según una política de planificación cíclica (Round Robin) con una determinada rodaja de tiempo. En la primera categoría, todos los procesos ligeros tienen una prioridad fija. En la segunda, los procesos comienzan su ejecución con una determinada prioridad y ésta va cambiando durante la vida del proceso, pero sin llegar al nivel 16. Esta prioridad se modifica según el comportamiento que tiene el proceso durante su ejecución. Así, un proceso (situado en el nivel de prioridades variable) ve decrementada su prioridad si acaba la rodaja de tiempo. En cambio, si el proceso se bloquea, por ejemplo, por una petición de E/S bloqueante, su prioridad aumentará. Con esto se persigue mejorar el tiempo de respuesta de los procesos interactivos que realizan E/S.

2.2 Hilos (threads)

Una hebra, o hilo, es un proceso ligero con un estado reducido. La reducción de estado se logra disponiendo que un grupo de hebras relacionadas compartan recursos como memoria y archivos.

Todo hilo pertenece a un solo proceso, y ningún hilo puede existir fuera de un proceso. Los procesos son estáticos y sólo los hilos pueden ser planificados para su ejecución. Los hilos son un mecanismo conveniente para explotar la concurrencia dentro de una aplicación. Los hilos pueden comunicarse eficientemente por medio de memoria compartida comúnmente accesible dentro del proceso que las engloba.

2.2.1 Llamadas al sistema para la administración de hilos en Linux

pthread_create(): Crea un nuevo hilo de ejecución, indicando los atributos del hilo.

pthread_equal(): Compara si dos identificadores de hilo son el mismo.

pthread_exit(): Finaliza el hilo que realiza la llamada.

pthread_join(): Espera la terminación de un hilo específico.

pthread_self(): Devuelve el identificador del hilo que realiza la llamada.

pthread_getschedparam(): Obtiene la política de planificación y los parámetros del hilo especificado.

pthread_setschedparam(): Establece la política de planificación y los parámetros del hilo especificado.

pthread_attr_init(): Permite iniciar un objeto atributo que se puede utilizar para crear nuevos hilos.

pthread_attr_destroy(): Destruye el objeto de tipo atributo.

2.2.2 Servicios POSIX para la planificación de hilos en Linux

Modificar los parámetros de planificación

pthread_setschedparam( ): Modifica la prioridad y política de planificación.

Obtener los parámetros de planificación

pthread_getschedparam( ): Devuelve la prioridad del proceso.

2.2.3 Llamadas al sistema para la administración de hilos en Windows

Creación de procesos ligeros

En Win32, los procesos ligeros se crean mediante la función CreateThread. El prototipo de esta función es:

```
BOOL CreateThread (
                    LPSECURITY_ATTRIBUTES lpsa,
                    DWORD cbstack,
                    LPTHREAD_START_ROUTINE lpStartAddr;
                    LPVOID lpvThreadParam,
                    DWORD fdwCreate,
                    LPDWORD lpIdThread) ;
```

Esta función crea un proceso ligero. El argumento lpsa contiene la estructura con los atributos de seguridad asociados al nuevo proceso ligero. El argumento cbStack específica el tamaño de la pila asociada al proceso ligero. Un valor de cero especifica el tamaño por defecto (1M). lpStartAddr apunta a la función a ser

ejecutada por el proceso ligero. El parámetro lpvThreadParam almacena el parámetro pasado al proceso ligero. Si fdwCreate es cero, el proceso ligero se ejecuta inmediatamente después de su creación. En lpIdThread se almacena el identificador del nuevo proceso creado. La función CreateThread devuelve el manejador para el nuevo proceso ligero creado o bien NULL en caso de error.

Terminación de Procesos Ligeros

Al igual que con los procesos en Win32, los servicios relacionados con la terminación de procesos ligeros se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso ligero y servicios para esperar la terminación de procesos (WaitForSingleObject y WaitForMultipleObjects).

Terminar la ejecución de un proceso: un proceso ligero puede finalizar su ejecución de forma voluntaria de dos formas:

- ✓ Ejecutando dentro de la función principal del proceso ligero la sentencia *return.*
- ✓ Ejecutando el servicio ExitThread.

El prototipo de la función Exithread es:

VOID ExitThread (DWORD dwExitCode);

Con esta función un proceso ligero finaliza su ejecución especificando su código de salida mediante el argumento dwExitCode. Este código puede consultarse con la función GetExitCodeThread similar a la función GetExitCodeProcess.

Un proceso ligero puede también abortar la ejecución de otro proceso ligero mediante el servicio:

BOOL TerminateThread (HANDLE hThread, DWORD dwExitCode);

Similar a la función TerminateProcess.

2.2.4 Servicios de planificación en win32

Win32 ofrece servicios para que los usuarios puedan modificar aspectos relacionados con la prioridad y planificación de los procesos y de los procesos ligeros. La prioridad de ejecución va asociada a los procesos ligeros, ya que éstos son las unidades básicas de ejecución. La clase de prioridad va asociada a los procesos.

La prioridad de cada proceso ligero se determina según los siguientes criterios:

- ✓ La clase de prioridad de su proceso.
- ✓ El nivel de prioridad del proceso ligero dentro de la clase de prioridad de su proceso.

En Windows NT existen seis clases de prioridad que se fijan inicialmente en la llamada CreateProcess. Estas seis clases son:

- ✓ IDLE_PRIORITY_CLASS, con prioridad base 4.
- ✓ BELOW_NORMAL_PRIORITY_CLASS, con prioridad base 6.
- ✓ NORMAL_PRIORITY_CLASS, con prioridad base 9.
- ✓ ABOVE_NORMAL_PRIORITY_CLASS, con prioridad base 10.

- ✓ HIGH_PRIORITY_CLASS, con prioridad base 13.
- ✓ REAL_TIME_PRIORITY_CLASS, con prioridad base 24.

La clase prioridad por defecto para un proceso es NORMAL_PRIORITY_CLASS.

Un proceso puede modificar o consultar su clase o la de otro proceso utilizando los siguientes servicios:

```
BOOL SetPriorityClass (HANDLE hProcess, DWORD
fdwPriorityClass);

DWORD GetPriorityClass (HANDLE hProcess) ;
```

Las funciones del API Win32 relacionadas con la planificación se describen a continuación:

- ✓ Suspend / Resume Thread : Suspende o reanuda un proceso detenido.
- ✓ Get / SetPriorityClass : Devuelve o fija la clase de prioridad de un proceso.
- ✓ Get / SetThreadPriority : Devuelve o fija la prioridad de un subproceso.
- ✓ Get / SetProcessAffinityMask : Devuelve o fija la máscara de afinidad del proceso.
- ✓ SetThreadAffinityMask : Fija la máscara de afinidad de un subproceso (debe ser un subconjunto de la máscara de afinidad del proceso) para un conjunto particular de procesadores, de forma que se restrinja la ejecución en esos procesadores.
- ✓ Get / SetThreadPriorityBoost : Devuelve o fija la capacidad de Windows NT para incrementar la prioridad de un subproceso temporalmente (sólo se aplica a subprocesos en el rango dinámico).

- ✓ SetThreadIdealProcesor : Establece el procesador preferido para un subproceso en particular, aunque no limita el subproceso a dicho procesador.
- ✓ Get/SetProcessPriorityBoost : Devuelve o fija el estado de control de incremento de prioridad predeterminado del proceso actual.
- ✓ SwitchToThread : Cede la ejecución del quantum a otro subproceso que está preparado para ejecutarse en el proceso actual.
- ✓ Sleep : Pone el subproceso actual en el estado de espera durante un intervalo de tiempo especificado. El valor cero hace que se pierda el resto del quantum del subproceso.
- ✓ SleepEx : Hace que el subproceso actual pase al estado de espera hasta que se termine una operación de E/S o se transcurra el intervalo de tiempo especificado.

3. COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Los procesos son entes independientes y aislados, puesto que, por razones de seguridad, no deben interferir unos con otros. Sin embargo, cuando se divide un trabajo complejo en varios procesos que cooperan entre sí para realizar ese trabajo, es necesario que se comuniquen para transmitirse datos y órdenes y se sincronicen en la ejecución de sus acciones.

En algunas situaciones, los procesos requieren la comunicación con otros procesos para su ejecución, los sistemas operativos deben proporcionar mecanismos para que esto se lleve a cabo. Si el uso de los recursos por parte de los procesos no se controla adecuadamente, los procesos pueden bloquearse, los sistemas operativos, por tanto, deben proporcionar mecanismos para el manejo de los bloqueos mutuos.

El acceso concurrente a datos compartidos puede dar pie a inconsistencia de los datos. Por lo tanto, es importante estudiar la sincronización de procesos que es un mecanismo que asegura la ejecución ordenada de procesos cooperativos que comparten un espacio de direcciones lógico, con el fin de mantener la consistencia de los datos.

Cada proceso tiene un segmento de código especial, llamado sección crítica, en el cual el proceso puede encontrarse manipulando recursos compartidos con los demás procesos. Cada proceso debe solicitar permiso para entrar en su sección crítica y si otro ya se está ejecutando en ésta, el permiso al proceso solicitante debe ser negado. Cualquier solución al problema de la sección crítica debe satisfacer los siguientes requisitos:

- ✓ Exclusión mutua. Sólo un proceso puede estar ejecutando la sección crítica propia.
- ✓ Progreso. Si ningún proceso se está ejecutando en su sección crítica y hay otros procesos que desean entrar en las suyas, entonces sólo aquellos procesos que no se están ejecutando en su sección restante pueden participar en la decisión de cuál será el siguiente en entrar en la sección crítica, y esta selección no puede postergarse indefinidamente.
- ✓ Espera limitada. Los procesos no deben esperar indefinidamente para el ingreso a su sección crítica.

Para solucionar el problema de la sección crítica, podemos encontrar *soluciones para dos procesos* y *soluciones para múltiples procesos*. Este tipo de soluciones no son fáciles de generalizar a problemas más complejos pero para superar esta dificultad, podemos usar una herramienta de sincronización denominada *Semáforo.* Un *Semáforo* S es una variable entera a la que, una vez que se le ha asignado un valor inicial, sólo puede accederse a través de dos operaciones atómicas estándar: espera (wait) y señal (signal). Es decir, podemos utilizar semáforos para resolver el problema de la sección crítica con *n* procesos y para resolver diversos problemas de sincronización.

3.1 Mecanismos de sincronización en Linux

3.1.1 Semáforos

Un semáforo es un objeto con un valor entero al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: wait y signal. Los semáforos son mecanismos de sincronización útiles

para coordinar el acceso a recursos. En POSIX, un semáforo se identifica mediante una variable del tipo sem_t. El estándar POSIX define dos tipos de semáforos:

- ✓ **Semáforos sin nombre**. Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso o a los procesos que lo heredan a través de la llamada fork.
- ✓ **Semáforos con nombre**. En este caso, el semáforo lleva asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada fork.

Creación de un semáforo sin nombre

Todos los semáforos en POSIX deben iniciarse antes de su uso. La función **sem_init** permite iniciar un semáforo sin nombre. El prototipo de este servicio es el siguiente:

int **sem_init**(sem_t *sem, int shared, int val);

Con este servicio se crea y se asigna un valor inicial a un semáforo sin nombre. El primer argumento identifica la variable de tipo semáforo que se quiere utilizar. El segundo argumento indica si el semáforo se puede utilizar para sincronizar procesos ligeros o cualquier otro tipo de proceso. Si shared es 0, el semáforo sólo puede utilizarse entre los procesos ligeros creados dentro del proceso que inicia el semáforo. Si shared es distinto de 0, entonces se puede utilizar para sincronizar procesos que lo hereden por medio de la llamada fork. El tercer argumento representa el valor que se asigna inicialmente al semáforo.

Destrucción de un semáforo sin nombre

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada sem_init. Su prototipo es el siguiente:

int **sem_destroy**(sem_t *sem)

Creación y apertura de un semáforo con nombre

El servicio **sem_open** permite crear o abrir un semáforo con nombre. La función que se utiliza para invocar este servicio admite dos modalidades, según se utilice para crear el semáforo o simplemente abrir uno existente. Estas modalidades son las siguientes:

sem_t ***sem_open**(char *name, int flag, mode_t mode, int val);
sem_t ***sem_open**(char *name, int flag);

Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un archivo. La función **sem_open** establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

El valor del segundo argumento determina si la función **sem_open** accede a un semáforo previamente creado o si crea un nuevo. Un valor 0 en flag indica que se quiere utilizar un semáforo que ya ha sido creado, en este caso no es necesario los dos últimos parámetros de la función **sem_open**. Si flag tiene un valor O_CREAT, requiere los dos últimos argumentos de la función. El tercer parámetro especifica los permisos del semáforo que se va a crear, de la misma forma que ocurre en la

llamada open para archivos. El cuarto parámetro especifica el valor inicial del semáforo.

Cierre de un semáforo con nombre

Cierra un semáforo con nombre rompiendo la asociación que tenía un proceso con un semáforo. El prototipo de la función es:

int **sem_close**(sem_t *sem);

Borrado de un semáforo con nombre

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función **sem_close.** El prototipo de este servicio es:

int **sem_unlink**(char *name);

Operación wait

La operación wait en POSIX se consigue con el siguiente servicio:

int **sem_wait**(sem_t *sem);

Operación signal

Este servicio se corresponde con la operación signal sobre un semáforo. El prototipo de este servicio es:

int **sem_post**(sem_t *sem);

Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito o –1 en caso de error.

3.1.2 Mútex

Son mecanismos de sincronización a nivel de hilos, se utilizan para garantizar la exclusión mutua en el acceso a un código. Cada mútex posee:

- ✓ Dos estados internos: abierto y cerrado
- ✓ Un hilo propietario, cuando el mútex está cerrado

Llamadas POSIX:

Creación del mútex: pthread_mutex_init

- ✓ int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- ✓ pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

donde podemos crear el mútex con los atributos **attr** o con los atributos por defecto (PTHREAD_MUTEX_INITIALIZER)

Destrucción del mútex: pthread_mutex_destroy

INT pthread_mutex_destroy(pthread_mutex_t * mutex);

Atributos de creación de mútex

- ✓ int pthread_mutexattr_init(pthread_mutexattr_t *attr);
- ✓ int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

Modificación de atributos de creación de mútex

- ✓ int pthread_mutexattr_getpshared (const pthread_mutexattr_t *attr, int *pshared);
- ✓ int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);

donde **pshared** indica si el mútex podrá ser utilizado por hilos del proceso que creó el mútex o por hilos de otros procesos:

PTHREAD_PROCESS_PRIVATE
PTHREAS_PROCESS_SHARED

- ✓ int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- ✓ int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol);

donde **protocol** indica el protocolo de sincronización del mútex:

- ⇒ PTHREAD_PRIO_NONE: Sin protocolo
- ⇒ PTHREAD_PRIO_INHERIT: Protocolo Básico de herencia de prioridad
- ⇒ PTHREAD_PRIO_PROTEC: Protocolo de techo de prioridad inmediato

- ✓ int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);
- ✓ int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int *prioceiling);

donde **prioceiling** indica el techo de prioridad del mútex. Solo tiene sentido en la política de sincronización PTHREAD_PRIO_PROTEC.

- ✓ int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
- ✓ int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int type);

donde **type** indica el tipo de mútex (respecto a interbloqueos):

- ⇒ PTHREAD_MUTEX_NORMAL: sin comprobación, interbloqueo posible
- ⇒ PTHREAD_MUTEX_ERRORCHECK: con comprobación de errores
- ⇒ PTHREAD_MUTEX_RECURSIVE: admite varios cierres seguidos (del mismo hilo) sin apertura previa.
- ⇒ PTHREAD_MUTEX_DEFAULT: sin comprobación (dependiente de la implementación)

Cierre y apertura de mútex

- ✓ int pthread_mutex_lock(pthread_mutex_t *mutex);
- ✓ int pthread_mutex_trylock(pthread_mutex_t *mutex);
- ✓ int pthread_mutex_unlock(pthread_mutex_t *mutex);

donde

lock/trylock: cierra(o intenta cerrar) el mútex: Si está abierto, se cierra y el hilo invocante pasa a ser el propietario. Si está cerrado:

⇒ lock: suspende el hilo invocante
⇒ trylock: devuelve un código de error

unlock: abre el mútex. Si hay hilos suspendidos, selecciona el más prioritario y permite que cierre el mútex.

3.1.3 Variables de condición

Las variables de condición están asociadas con un mútex y se emplean para sincronizar los hilos. Las tres operaciones básicas son:

- ✓ Espera
- ✓ Aviso simple
- ✓ Aviso múltiple

Llamadas POSIX

Creación de variables de condición: pthread_cond_init

- ✓ int pthread_cond_init (pthread_cond_t *cond, conspthread_condattr_t *attr);
- ✓ pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

donde podemos crear la variable **cond** con los atributos **attr** o con los atributos por defecto (PTHREAD_COND_INITIALIZER)

Destrucción de variables de condición: pthread_cond_destroy

int pthread_cond_destroy(pthread_cond_t *cond);

Atributos de creación de variables de condición

- ✓ int pthread_condattr_init(pthread_condattr_t *attr);
- ✓ int pthread_condattr_destroy(pthread_condattr_t *attr);

Modificación de atributos de creación de variables de condición

- ✓ int pthread_condattr_getpshared(const pthreadcondattr_t *attr, int *phared);
- ✓ int pthread_condattr_setpshared(pthread_condattr_t *attr, int phared);

donde **phared** indica si la variable podrá ser utilizado por hilos del proceso que creó el mútex o por hilos de otros procesos:

PTHREAD_PROCESS_PRIVATE
PTHREAS_PROCESS_SHARED

Espera sobre variables de condición

- ✓ int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- ✓ int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

donde
wait: ejecuta un pthread_mutex_unlock sobre mútex y atómicamente suspende al hilo invocante en la variable cond. Al despertarse, realizará atómicamente un pthread_mutex_lock sobre mútex.
timedwait: actúa como wait, salvo que el hilo se suspende hasta que se alcanza el tiempo abstime.

Aviso sobre variables de condición

- ✓ int pthread_cond_signal(pthread_cond_t *cond);
- ✓ int pthread_cond_broadcast(pthread_cond_t *cond);

donde

signal: selecciona el hilo más prioritario suspendido en cond y lo despierta.
broadcast: despierta todos los hilos que pueda haber suspendidos en cond.

Ambas no tienen efecto si no hay hilos suspendidos en la variable. Es recomendable que el hilo invocante sea el propietario del mútex asociado a cond en los hilos suspendidos.

3.2 Mecanismos de sincronización en Windows NT

Win32 dispone de dos mecanismos de comunicación que también se pueden utilizar para sincronizar. Estos mecanismos son las tuberías y los mailslots. Como mecanismos de sincronización puros, Win32 dispone de secciones críticas, semáforos, mútex y eventos.

3.2.1 Secciones críticas

Las secciones críticas son un mecanismo de sincronización especialmente concebido para resolver el acceso a secciones de código que deben ejecutarse en exclusión mutua.

Las secciones críticas son objetos que se crean y se borran pero no tienen manejadores asociados. Sólo se pueden utilizar por los procesos ligeros creados dentro de un proceso. Los servicios utilizados para tratar las secciones críticas son los siguientes:

```
VOID InitializeCriticalSection ( LPCCRITICAL_SECTION  lpcsCriticalsection) ;
```

VOID DeleteCriticalSection (LPCCRITICAL_SECTION lpcsCriticalSection) ;

VOID EnterCriticalSection (LPCCRITICAL_SECTION lpcsCriticalSection) ;

VOID LeaveCriticalSection (LPCCRITICAL_SECTION lpcsCriticalSection) ;

Las dos primeras secciones se utilizan para crear y borrar secciones críticas. Las dos siguientes sirven para entrar y salir de la sección crítica.

El acceso en exclusión mutua a una sección crítica se resuelve de forma muy sencilla utilizando este mecanismo de Win32. en primer lugar se deberá crear una sección crítica e inicializarla:

```
LPCCRITICAL_SECTION SC;
InitializeCriticalSection ( &SC );
```

Siempre que se desee acceder a una sección crítica deberá utilizarse el siguiente fragmento de código:

```
EnterCriticalSection ( &SC );
< Código de la sección crítica >
LeaveCriticalSection ( &SC );
```

Cuando deje de accederse a la sección crítica se deberá destruir utilizando:

```
DeleteCriticalSection ( &SC );
```

3.2.2 Semáforos

En Win32 los semáforos tienen asociado un nombre. Los servicios de win32 para trabajar con semáforos son los siguientes:

Crear un semáforo

Los semáforos en Win32 se manipulan, como el resto de los objetos, con manejadores. Para crear un semáforo se utiliza el siguiente servicio:

HANDLE CreateSemaphore (LPSECURITY_ATTRIBUTES lpsa, LONG cSemInitial, LONG cSemMax, LPCTSTR lpszSemName) ;

El primer parámetro especifica los parámetros de seguridad asociados al semáforo. El argumento cSemInitial indica el valor inicial del semáforo y cSemMax el valor máximo que puede tomar. El nombre del semáforo viene dado por el parámetro lpszSemName. La llamada devuelve un mensaje de semáforo válido o NULL en caso de error.

Abrir un semáforo

Una vez creado un semáforo, un proceso puede abrirlo mediante el servicio:

HANDLE OpenSemaphore (LONG dwDesiredAccess, LONG BinheritHandle, lpszName SemName) ;

El parámetro dwDesiredAccess puede tomar los siguientes valores:

- ✓ SEMAPHORE_ALL_ACCESS: total acceso al semáforo.
- ✓ SEMAPHORE_MODIFY_STATE: permite la ejecución de la función ReleaseSemaphore.
- ✓ SYNCHRONIZE: permite el uso del semáforo para sincronización, es decir, en las funciones de espera (wait).

El segundo argumento indica si se puede heredar el semáforo a los procesos hijos. Un valor de TRUE permite heredarlo. SenName indica el nombre del semáforo que se desea abrir. La función devuelve un manejador de semáforo válido en caso de éxito o NULL en caso de error.

Cerrar un semáforo

Para cerrar un semáforo se utiliza el siguiente servicio:

BOOL CloseHandle (HANDLE hObject);

Si la llamada termina correctamente, se cierra el semáforo. Si el contador del manejador es cero, se liberan los recursos ocupados por el semáforo. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Operación Wait

Se utiliza el siguiente servicio de Win32:

DWORD WaitForSingleObject (HANDLE hSem, DWORD dwTimeOut);

Ésta es la función general de sincronización que ofrece Win32. Cuando se aplica a un semáforo implementa la función wait. El parámetro dwTimeOut debe tomar en este caso valor INFINITE.

3.2.3 Mútex

Los mútex, al igual que los semáforos, son objetos con nombre. Los mútex sirven para implementar secciones críticas. La diferencia entre las secciones críticas y los mútex de Win32 radica en que las secciones críticas no tienen nombre y sólo se pueden utilizar entre procesos ligeros de un mismo proceso. Un mútex se manipula usando manejadores.

Los servicios utilizados en Win32 para trabajar con mútex son los siguientes:

Crear un mútex

Para crear un mútex se utiliza el siguiente servicio:

HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpsa, BOOL fInitialOwner,
LPCTSTR lpszMutexName);

Esta función crea un mútex con atributos de seguridad lpsa. Si fInitialOwner es TRUE, el propietario del mútex será el proceso ligero que lo crea. El nombre del

mútex viene dado por el tercer argumento. En caso de éxito la llamada devuelve un manejador de mútex válido y en caso de error NULL.

Abrir un mútex

Para abrir un mútex se utiliza:

HANDLE OpenMutex (LONG dwDesiredAccess, LONG BineheritHandle,
LpszName SemName);

Los parámetros y su comportamiento son similares a los de la llamada OpenSemaphore.

Cerrar un mútex

Para cerrar un semáforo se utiliza el siguiente servicio:

BOOL CloseHandle (HANDLE hObject);

Si la llamada termina correctamente, se cierra el mútex. Si el contador del manejador es cero, se liberan los recursos ocupados por el mútex. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Operación lock

Se utiliza el siguiente servicio de win32:

```
DWORD WaitForSingleObject ( HANDLE hMutex, DWORD dwTimeOut );
```

Ésta es la función general de sincronización que ofrece Win32. Cuando se aplica a un mútex implementa la función lock. El parámetro dwTimeOut debe tomar en este caso el valor INFINITE.

Operación unlock

El prototipo de este servicio es:

```
BOOL ReleaseMutex ( HANDLE hMutex );
```

Los eventos en Win32 son comparables a las variables condicionales, es decir, se utilizan para notificar que alguna operación se ha completado o que ha ocurrido algún proceso. Sin embargo, las variables condicionales se encuentran asociadas a un mútex y los eventos no.

Los eventos en Win32 se clasifican en manuales y automáticos. Los primeros se pueden utilizar pata desbloquear a varios threads bloqueados en un evento. En este caso, el evento permanece en estado de notificación y debe eliminarse este estado de forma manual. Los automáticos se utilizan para desbloquear a un único thread, es decir, el evento notifica a un único thread y a continuación deja de estar en este estado de forma automática.

El sistema operativo UNIX permite que procesos diferentes intercambien información entre ellos. Para procesos que se están ejecutando bajo el control de una misma máquina permite la comunicación mediante el uso de:

- ✓ Tuberías
- ✓ Memoria compartida
- ✓ Semáforos
- ✓ Colas de mensajes.

3.3 Mecanismos IPC (Inter process Communication - Comunicación entre Procesos) en Linux

Los medios IPC de Linux proporcionan un método para que múltiples procesos se comuniquen unos con otros. Hay varios métodos de IPC disponibles:

3.3.1 Tuberías (Pipes)

Una tubería *(pipe)* se puede considerar como un canal de comunicación entre dos procesos. Este mecanismo de comunicación consiste en la introducción de información en una tubería por parte de un proceso, posteriormente otro proceso extrae la información de la tubería de forma que los primeros datos que se introdujeron en ella son los primeros en salir.

Clasificación de tuberías

Se pueden distinguir dos tipos de tuberías dependiendo de las características de los procesos que pueden tener acceso a ellas:

- ✓ **Sin nombre**. Solamente pueden ser utilizadas por los procesos que las crean y por los descendientes de éstos. Un *pipe* en POSIX sólo puede ser utilizado entre los procesos que lo hereden a través de la llamada fork(). A continuación se describen los servicios que permiten crear y acceder a los datos de un *pipe*:

Creación:

Las pipes se crean con la llamada a sistema **pipe**:

int **pipe**(int fd[2])

Abre dos canales de acceso a una pipe.

- ✓ fd[0] canal de sólo lectura,
- ✓ fd[1], canal de sólo escritura

Devuelve 0, si se ha completado correctamente; -1 en caso de error.

Cierre

El cierre de cada uno de los descriptores que devuelve la llamada *pipe* se consigue mediante el servicio close, que también se emplea para cerrar cualquier archivo. Su prototipo es:

int **close**(int fd);

El argumento de close indica el descriptor de archivo que se desea cerrar. La llamada devuelve 0 si se ejecutó con éxito. En caso de error, devuelve –1.

Escritura

El servicio para escribir datos en un *pipe* en POSIX es el siguiente:

int **write**(int fd, char *buffer, int n);

El primer argumento representa el descriptor de archivo que se emplea para escribir en un *pipe*. El segundo argumento especifica el *buffer* de usuario donde se encuentran los datos que se van a escribir al *pipe*. El último argumento indica el número de bytes a escribir. Los datos se escriben en el *pipe* en orden FIFO. La semántica de esta llamada es la siguiente:

⇒ Si la tubería se encuentra llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.

⇒ Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve el correspondiente error. Este error se genera mediante el envío al proceso que intenta escribir de la señal SIGPIPE.

⇒ Una operación de escritura sobre una tubería se realiza de forma **atómica**, es decir, si dos procesos intentan escribir de forma simultánea en una tubería sólo uno de ellos lo hará, el otro se bloqueará hasta que finalice la primera escritura.

Lectura

Para leer datos de un *pipe* se utiliza el siguiente servicio, también empleado para leer datos de un archivo.

int **read**(int fd, char *buffer, int n);

El primer argumento indica el descriptor de lectura del *pipe*. El segundo argumento especifica el *buffer* de usuario donde se van a situar los datos leídos del *pipe*. El último argumento indica el número de bytes que se desean leer del *pipe*. La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve –1. Las operaciones de lectura siguen la siguiente semántica:

⇒ Si la tubería está vacía, la llamada bloquea al proceso en la operación de lectura hasta que algún proceso escriba datos en la misma.

⇒ Si no hay escritores y la tubería está vacía, la operación devuelve fin de archivo (la llamada read devuelve cero). En este caso, la operación no bloquea al proceso.

⇒ Al igual que las escrituras, las operaciones de lectura sobre una tubería son atómicas.

✓ **FIFO (pipe con nombre)**. Se utilizan para comunicar procesos entre los que no existe ningún tipo de parentesco. El funcionamiento de una *fifo* es similar al de

las tuberías sin nombre, pero se accede a ellas de igual manera que a los archivos de disco, es decir con la llamada al sistema *open*. Cualquier proceso que desee introducir información en ella ha de abrirla en modo de escritura, mientras que aquél que desee recibir su contenido debe abrirla en modo de lectura. La llamada al sistema para crear una *fifo* es *mkfifo*:

```
int mkfifo (char *ruta, mode_t modo);
```

El parámetro *ruta* es una cadena de caracteres que define la ruta y el nombre de la *fifo,* mientras que *modo* determina los permisos que se conceden a los diferentes usuarios utilizando valores numéricos pero expresados con cuatro cifras. La función devuelve el valor –1 en caso de error, en cuyo caso queda en *errno* el correspondiente código de error.

Llamadas al sistema

⇒ Open (char *name, int flag);

- Abre un FIFO (para lectura, escritura o ambas)
- Bloquea hasta que haya algún proceso en el otro extremo

⇒ Lectura y escritura mediante *read()* y *write()*
Igual semántica que los pipes

⇒ Cierre de un FIFO mediante *close()*

⇒ Borrado de un FIFO mediante *unlink()*

3.3.2 System V IPC

Se conocen como System V IPCs a tres técnicas de comunicación entre procesos que provee el UNIX System V:

⇒ *Memoria compartida:* Provee comunicación entre procesos permitiendo que éstos compartan zonas de memoria. La memoria compartida se puede describir mejor como el plano (mapping) de un área (segmento) de memoria que se combinará y compartirá por más de un de proceso. Esta es la forma más rápida de IPC (Inter process Communication - Comunicación entre Procesos), porque no hay intermediación (es decir, un tubo, una cola de mensaje, etc). En su lugar, la información se combina directamente en un segmento de memoria, y en el espacio de direcciones del proceso llamante. Un segmento puede ser creado por un proceso, y consecutivamente escrito y leído por cualquier número de procesos.

Llamadas al sistema:

- ✓ int shm_open(char *name, int oflag, mode_t mode): Crea un objeto de memoria a compartir entre procesos.
- ✓ int shm_open (char *name, int oflag): Sólo aperture.
- ✓ int close(int fd): Cierre. El objeto persiste hasta que es cerrado por el último proceso.
- ✓ int shm_unlink(const char *name): Borra una zona de memoria compartida.
- ✓ void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off): Establece una proyección entre el espacio de direcciones de un proceso y un descriptor de fichero u objeto de memoria compartida.

- ✓ void munmap(void *addr, size_t len): Desproyecta parte del espacio de direcciones de un proceso comenzando en la dirección ***addr***. El tamaño de la región a desproyectar es ***len***.

⇒ *Colas de mensajes:* Permiten tanto compartir información como sincronizar procesos. Un proceso envía un mensaje y otro lo recibe. El kernel se encarga de sincronizar la transmisión/recepción. Las colas de mensajes se pueden describir mejor como una lista enlazada interior, dentro del espacio de direccionamiento del núcleo. Los mensajes se pueden enviar a la cola en orden y recuperarlos de la cola en varias maneras diferentes. Cada cola de mensaje esta identificada de forma única por un identificador IPC.

Llamadas al sistema:

Msgget()

Para crear a una nueva cola de mensajes o acceder a una ya existente, usaremos la llamada al sistema msgget( ). Esta función nos devuelve el identificador de la cola de mensajes ya existente y, si la cola no existe, permite crearla.

int msgget (key_t llave,int msgflag);

key_t es el valor clave.
msgflag es un mapa de bits, que representa:

- ✓ Para crear una cola: IPC_CREAT | OXYZ.
- ✓ Para saber el identificador de una cola ya existente: OXYZ.

Msgsnd() y Msgrcv()

Estas llamadas al sistema se utilizan para escribir y leer de una cola:

int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

- ✓ msqid es el identificador de la cola.
- ✓ msgsz establece el tamaño del mensaje que se va a leer o escribir.
- ✓ El puntero msgp señala la zona de memoria donde se encuentran los datos a enviar o donde se almacenarán aquellos que van a ser leídos.
- ✓ El parámetro msgtyp sólo aparece en la llamada que permite la lectura de la cola y permite especificar qué mensaje de los que se encuentran en la cola queremos leer

  msgtyp =0 se accederá al primer mensaje que se encuentre en la cola independientemente de su tipo.

  msgtyp >0 se accederá al primer mensaje del tipo msgtyp que se encuentre almacenado en la cola.

  msgtyp <0 se accederá al primer mensaje cuyo tipo sea menor o igual al valor absoluto de msgtyp y a la vez sea el menor de todos.

  El campo msgflg permite indicar si el proceso desea suspenderse en espera de que la llamada puede ejecutarse o no

Msgctl()

Esta función nos permite modificar la información de control que el Kernel le asocia a cada cola (identificada esta por su idmsg).

```
int msgctl (int msqid, int cmd, struct msqid_ds * buf);
```

El primer parámetro msqid que le pasamos a la función indica la cola que pretendemos controlar.

El segundo parámetro cmd permite seleccionar el tipo de operación que se va a realizar sobre la cola, puede tomar los siguientes valores:

- ✓ IPC_STAT: devuelve el estado de la estructura de control asociada a la cola en la zona de memoria apuntada por buf .
- ✓ IPC_SET: inicializa los campos de la estructura de control de la cola según el contenido de la zona de memoria apuntada por buf .
- ✓ IPC_RMID: elimina la cola de mensajes identificada por msqid. La cola no se borra mientras haya algún proceso que la esté utilizando.

El tercer parámetro buf será un puntero que apunta a una zona de memoria.

Es conveniente recordar que estos objetos IPC no se van a menos que se quiten apropiadamente, o el sistema se reinicie. Y el uso de la función msgctl( ) es la forma adecuada de deshacernos de una cola de mensajes que ya cumplió su cometido.

3.3.3 Señales

Una señal es una interrupción software enviada a un proceso. El sistema operativo emplea las señales para informar acerca de situaciones excepcionales a un proceso.

Cualquier proceso puede enviar a otro proceso una señal, siempre que tenga permiso, cuando un proceso se prepara para la recepción de una señal, puede realizar las siguientes acciones:

- ✓ Ignorar la señal.
- ✓ Realizar la acción asociada por defecto a la señal.
- ✓ Ejecutar una rutina del usuario asociada a dicha señal.

La recepción de una señal en particular por parte de un proceso provoca que se ejecute una subrutina encargada de atenderla. A esa rutina se le llama el "manejador" de la señal (signal handler). Un proceso puede definir un manejador diferente para sus señales o dejar que el kernel tome las acciones predeterminadas para cada señal. Cuando un proceso define un manejador para cierta señal se dice que "captura" (catch) esa señal.

Un proceso tiene ciertas libertades para configurar como reacciona frente a una señal (capturando, bloqueando o ignorando la señal), el kernel se reserva ciertos derechos sobre algunas señales. Así, las señales llamadas KILL y STOP no pueden ser capturadas, ni bloqueadas, ni ignoradas y la señal CONT no puede ser bloqueada.

Una señal puede enviarse desde un programa utilizando llamadas al sistema operativo, o desde la línea de comandos de un shell utilizando el comando kill. Al

comando kill se le pasa como parámetro el número o nombre de la señal y el PID del proceso.

Llamada a sistema:

- ✓ *signal():* Asocia una acción determinada con una señal.
- ✓ *kill( )*: Envía una señal a un proceso
- ✓ *alarm( )*: Envía una señal de alarma en un período de tiempo especificado.
- ✓ *sigemptyset( )*: Crea un conjunto de señales vacío.
- ✓ *sigfillset( )*: Crea un conjunto de señales completo.
- ✓ *sigaddset( )*: Añade una señal al conjunto.
- ✓ *sigdelset( )*: Elimina una señal del conjunto.
- ✓ *sigprocmask( )*: Examina o modifica un conjunto de señales de proceso.
- ✓ *sigaction( )*: Especifica la acción a realizar cuando un proceso recibe una señal.
- ✓ *sigsuspend( )*: Suspende un proceso hasta que reciba una señal del conjunto.
- ✓ *alarm(sec):* El kernel le enviará al proceso llamante una señal SIGALRM dentro de sec segundos.
- ✓ *pause( ):* El proceso queda bloqueado hasta que le llegue una señal.
- ✓ *time ( ):* el kernel consulta el reloj y devuelve su valor como retorno de la llamada.

Algunas señales

- ✓ SIGHUP Colgar. Generada al desconectar el terminal.
- ✓ SIGINT Interrupción. Generada por teclado.
- ✓ SIGILL Instrucción ilegal. No se puede capturar.

- ✓ SIGFPE Excepción aritmética, de coma flotante o división por cero.
- ✓ SIGKILL Matar proceso, no puede capturarse, ni ignorarse.
- ✓ SIGBUS Error en el bus.
- ✓ SIGSEGV Violación de segmentación.
- ✓ SIGPIPE Escritura en una pipe para la que no hay lectores.
- ✓ SIGALRM Alarma de reloj.
- ✓ SIGTERM Terminación del programa.

3.4 Comunicación entre procesos en Windows NT

3.4.1 Tuberías

Win32 ofrece dos tipos de tuberías: sin nombre y con nombre. Las primeras sólo permiten transferir datos entre procesos que hereden el pipe. Las tuberías con nombre tienen las siguientes propiedades:

- ✓ Las tuberías con nombre están orientadas a mensajes, de tal manera que un proceso puede leer mensajes de longitud variable, escritos por otro proceso.
- ✓ Son bidireccionales, así que dos procesos pueden intercambiar mensajes utilizando una misma tubería.
- ✓ Puede haber múltiples instancias independientes de la misma tubería. Todas las instancias comparten el mismo nombre, pero cada una de ellas tiene sus propios buffers y manejadores. El uso de instancias permite que múltiples clientes puedan utilizar la misma tubería con nombre de forma simultánea.
- ✓ Se pueden utilizar en sistemas conectados a una red, por lo que los hace especialmente útiles en aplicaciones cliente-servidor. En este tipo de

aplicaciones un proceso (el servidor) crea una tubería con nombre y los procesos clientes se conectan a una instancia de esa tubería.

Las tuberías sin nombre tienen asociadas dos manejadores: uno para lectura y otro para escritura. Las tuberías con nombre únicamente tienen asociado un manejador que se puede utilizar para operaciones de lectura, escritura o ambas.

Los principales servicios que ofrece Win32 para trabajar con tuberías sin nombre son los siguientes:

Crear una tubería sin nombre

El prototipo es el siguiente:

```
BOOL CreatePipe ( PHANDLE phRead, PHANDLE phWrite,
LPSECURITY_ATTRIBUTES lpsa, DWORD cbPipe );
```

Los dos primeros argumentos representan dos manejadores de lectura y escritura respectivamente. El argumento lpsa indica los atributos de seguridad asociados a la tubería. El parámetro cbPipe indica el tamaño de la tubería. Si su valor es cero, se toma el valor por defecto. La llamada devuelve TRUE en caso de éxito y FALSE en caso contrario.

El atributo de seguridad viene dado por la estructura (LPSECURITY_ATTRIBUTES) con tres campos:

- ✓ nLength: especifica el tamaño de la estructura en bytes.

- ✓ lpSecurityDescriptor: puntero a un descriptor de seguridad que controla el acceso al objeto. Si es NULL, se utiliza el descriptor de seguridad asociado al proceso que realza la llamada.
- ✓ bInheritHandle: indica si el objeto puede ser heredado por los procesos que se creen. Si es TRUE, los procesos creados heredan el manejador.

Crear una tubería con nombre

La especificación de la función que permite crear una tubería con nombre en Win32 es la siguiente:

HANDLE CreateNamedPipe (LPCTSTR lpszPipeName, DWORD fdwOpenMode, DWORD fdwPipeMode, DWORD nMaxInstances, DWORD cbOutBuf, DWORD cbInBuf, DWORD dwTimeOut, LPSECURITY_ATTRIBUTES lpsa);

El parámetro lpszPipeName indica el nombre de la tubería. El nombre de la tubería debe tomar la siguiente forma:

\ \ . \ pipe \ [camino] nombre_tubería

El argumento fwOpenMode puede especificar alguno de los siguientes valores:

- ✓ PIPE_ACCESS_DUPLEX: permite que el manejador asociado a la tubería con nombre se pueda utilizar en operaciones de lectura y escritura.
- ✓ PIPE_ACCESS_INBOUND: sólo permite realizar operaciones de lectura de la tubería.

- ✓ PIPE_ACCESS_OUTBOUND: sólo permite realizar operaciones de escritura sobre la tubería.

El argumento fdwPipeMode tiene tres pares de valores mutuamente exclusivos. Estos valores indican si la escritura está orientada a mensajes o a bytes, si la lectura se realiza en mensajes o en bloques, y si las operaciones de lectura bloquean. Estos valores son:

- ✓ PIPE_TYPE_BYTE y PIPE_TYPE_MESSAGE: indican si los datos se escribe como un flujo de bytes o como mensajes.
- ✓ PIPE_READMODE_BYTE y PIPE_READMODE_MESSAGE: indican si los datos se leen como un flujo de bytes o como mensajes. PIPE_READMODE_MESSAGE requiere PIPE_TYPE_MESSAGE.
- ✓ PIPE_WAIT y PIPE_NOWAIT: indican si las operaciones de lectura sobre la tubería bloquearán al proceso lector en caso de que se encuentre vacía o no.

El parámetro nMaxInstances determina el número máximo de instancias de la tubería, es decir, el número máximo de clientes simultáneos. cbOutBuf y cbInBuf indican los tamaños en bytes de los buffers utilizados para la tubería. Un valor de cero representa los valores por defecto. dwTimeOut indica el tiempo de bloqueo, en milisegundos, de la función WaitNamedPipe. El último argumento representa los atributos de seguridad asociados a la tubería. La función devuelve el manejador asociado a la tubería.

Abrir una tubería con nombre

La apertura de una tubería con nombre es una operación que típicamente realizan los clientes en aplicaciones de tipo cliente / servidor. Para abrir una tubería con nombre se recurre al servicio CreateFile, que se utiliza en Win32 para crear archivos. Su prototipo es el siguiente:

```
HANDLE CreateFile ( LPCSTR lpFileName, DWORD dwDesiredAccess, WORD
dwShareMode, LPVOID lpSecurityAttributes,   DWORD CreationDisposition,
DWORD dwFlagsAndAttributes,   HANDLE hTemplateFile );
```

Su efecto es la creación o apertura de un archivo con nombre lpFileName. Este servicio se utiliza para abrir una tubería existente de la siguiente forma:

```
HandlePipe = CreateFile ( PipeName, GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
ILE_ATRIBUTE_NORMAL, NULL );
```

Con la llamada anterior se abrirá, para lectura y escritura, una tubería de nombre PipenName. La llamada fallará si la tubería con nombre todavía no se ha creado o todas sus instancias están ocupadas.

Relacionado con la apertura de la tubería con nombre se encuentra también el servicio WaitNamedPipe. Esta llamada bloquea a un proceso si todas las instancias de la tubería están ocupadas (normalmente por otros clientes). Su prototipo es el siguiente:

```
BOOL WaitNamedPipe ( LPCSTRPipeName, DWORD dwTimeOut );
```

Si dwTimeOut es NULL, se utilizará el tiempo de expiración especificado en CreateNamedPipe. Los posibles valores para este parámetro son:

- ✓ NMPWAIT_NOWAIT: devuelve inmediatamente si el servidor no está listo para recibir mensajes.
- ✓ NMPWAIT_WAIT_FOREVER: bloquea el proceso cliente hasta que el servidor esté listo para recibir mensajes.
- ✓ NMPWAIT_USE_DEFAULT_WAIT: utiliza el tiempo por defecto especificado en CreateNamedPipe.

Cerrar una tubería con nombre

Este servicio cierra el manejador asociado a la tubería con o sin nombre. Un prototipo es:

```
BOOL CloseHandle ( HANDLE hfile );
```

La llamada decrementa el contador de instancias asociado al objeto. Si el contador se hace cero, la tubería se destruye. Devuelve TRUE en caso de éxito y FALSE en caso de error.

Leer de una tubería

Para leer de una tubería se utiliza el servicio que ofrece WIN32 para leer de archivos. El prototipo de este servicio es:

BOOL ReadFile (HANDLE hFile, LPVOID lpBuffer, DWORD nBytes, LPDWORD lpnBytes, LPOVERLAPPED lpOverlapped);

El parámetro hFile es el manejador asociado a la tubería que se utiliza para leer. lpBuffer especifica el buffer de usuario donde se van a situar los datos leídos de la tubería. El argumento nBytes indica el número de bytes que se desean leer, en lpnBytes se almacena el número de datos realmente leídos. El último argumento es la estructura que se utiliza para indicar al posición de la cual se quiere leer, en el caso de tubería se utiliza NULL La función devuelve TRUE si se ejecutó con éxito o FALSE en caso contrario.

Escribir en una tubería

Al igual que con las lecturas, Win32 utiliza el mismo servicio que el empleado para escribir en archivos. El prototipo de esta función es:

BOOL WriteFile (HANDLE hFile, LPVOID lpBuffer, DWORD nBytes, LPDWORD lpnBytes, LPOVERLAPPED lpOverlapped);

El parámetro hFile es el manejador asociado a la tubería que se utiliza para escribir. lpBuffer especifica el buffer de usuario donde se encuentran los datos que se quieren escribir en la tubería. El argumento nBytes indica el número de bytes que se desean escribir. En lpnBytes se almacena el número de datos realmente escritos. El último argumento es una estructura que se utiliza para indicar la posición de la cual se quiere escribir, en el caso de tuberías se utiliza NULL. La función devuelve TRUE si se ejecutó con éxito o FALSE en caso contrario.

4. SEGURIDAD Y PROTECCIÓN

4.1 Protección

La protección se refiere a un mecanismo para controlar el acceso de programas, procesos o usuarios a los recursos definidos por un sistema de computación. Hay varias razones para proporcionar la protección, la más obvia es la necesidad de evitar una violación malintencionada de una restricción de acceso. Sin embargo, tiene una importancia más general la necesidad de asegurar que cada componente de un programa activo utilice los recursos sólo de manera consistente con las políticas establecidas para su uso. Este es un requisito absoluto para un sistema seguro.

4.2 Mecanismos y políticas

Las políticas que gobiernan la utilización de los recursos en un sistema de computación pueden establecerse de varias maneras: algunas están determinadas en el diseño del sistema, otras están formuladas por los administradores de un sistema, y algunas más son definidas por los usuarios para proteger sus propios archivos y programas.

Los mecanismos determinan cómo se hará algo, mientras que las políticas deciden qué se hará. Esta separación es importante respecto a la flexibilidad, ya que es probable que las políticas cambien de un lugar a otro o con el transcurso del tiempo.

4.3 Seguridad

La seguridad no sólo requiere un adecuado sistema de protección, sino también considera el entorno externo donde opera el sistema. La protección interna no es útil si la consola del operador está expuesta a personal no autorizado o si los archivos pueden extraerse del sistema de cómputo y transportarse a otro donde no hay protección. Estos problemas de seguridad son esencialmente problemas administrativos, no del sistema operativo.

El principal problema de seguridad para los sistemas operativos es el de la *validació*n. El sistema de protección depende de una habilidad para identificar los programas y procesos en ejecución, habilidad que, a su vez, finalmente depende del poder para identificar a cada usuario del sistema. Por lo general, la validación se basa en una combinación de tres conjuntos de elementos: la posesión del usuario (una llave o tarjeta), el conocimiento del usuario (un identificador de usuario y una contraseña) y los atributos del usuario (huellas digitales, patrón de rutina o firma). La estrategia más común para validar una identidad de usuario es por medio de *contraseña*s. Cuando el usuario se identifica, se le pide una contraseña. Si la contraseña que proporciona concuerda con la que está almacenada en el sistema, éste supone que el usuario es legítimo.

Las contraseñas presentan varios problemas, pero están muy extendidas por su facilidad de comprensión y uso. Los problemas de las contraseñas están relacionados con la dificultad para mantenerlas en secreto. Las contraseñas pueden ser generadas por el sistema o escogidas por los usuarios. Puede ser difícil recordar las contraseñas generadas por el sistema y, por tanto, es habitual que se anoten;

por otra parte, las contraseñas que eligen los usuarios muchas veces son fáciles de adivinar.

Se puede usar dos técnicas administrativas para mejorar la seguridad de un sistema. Una es la *búsqueda de amenaza*s, donde el sistema busca patrones de actividad sospechosos para tratar de detectar una violación a la seguridad. Otra técnica común es el *diario de auditorí*a, el cual sencillamente registra la hora, usuario y tipo de todos los accesos a un objeto. Después de una violación a la seguridad, puede emplearse el diario de auditoría para determinar cómo y cuándo ocurrió el problema y quizá la cantidad de daños.

4.4 Protección en Linux

El sistema UNIX es un sistema multiusuario, el cual tiene la capacidad de que todos los usuarios conectados a la máquina puedan leer y usar archivos de otros usuarios, debido a esto, los archivos deben ser protegidos contra cualquier uso indebido.

4.4.1 Protección de cuentas

En Unix cada usuario tiene asignada una cuenta individual. Cuando un usuario se desea conectar al sistema debe usar el nombre de conexión (login) y la autenticación del nombre de conexión (password). La protección de cuentas de usuarios es la p rimera línea de defensa contra los intrusos debido a que la forma más fácil por estos para ganar acceso es la consecución de un par login y password.

La información de la conexión de usuarios es almacenada en Unix en el archivo **/etc/passwd**, el cual contiene una línea por cuenta con la siguiente información:

<usuario>:<password>:<UID>:<GID>:<comentario>:<directorio home>:<Shell>

- ✓ Usuario: Es el nombre de la cuenta o login del usuario.
- ✓ Password: Puede contener una ***x*** o el password encriptado.
- ✓ UID: User ID (Identificador de usuario) único que tiene cada usuario para mostrar los dueños de los archivos.
- ✓ GID: Group ID (Identificador de grupo). Cada usuario tiene al menos un grupo de usuarios con similares privilegios. Este campo indica el Group ID primario.
- ✓ Comentario: Generalmente se usa para colocar la descripción o nombre completo del usuario.
- ✓ Directorio home: El directorio en el que es colocado el usuario cuando se conecta.
- ✓ Shell: El tipo de comando shell que usará el usuario cuando se conecte (***BourneShell, KornShell, Cshell***).

4.4.2 Shadow password

Es un método para mejorar la seguridad del sistema trasladando los passwords encriptados (encontrados normalmente en /etc/passwd) a **/etc/shadow** el cual puede ser leído únicamente por usuarios privilegiados, en cambio que /etc/passwd puede ser leído por todos los usuarios del sistema. Este archivo tiene la siguiente estructura:

usuario:password:last:may:must:warn:expire:disable:reserved

- ✓ *Usuario*: Es el nombre de la cuenta o *login.*
- ✓ *Password:* Indica el *password* que ha sido encriptado.
- ✓ *Last:* Número de días desde de enero 1 de 1970 que la contraseña fue cambiada por última vez.
- ✓ *May:* Número de días faltantes para que la contraseña pueda ser cambiada.
- ✓ *Warn:* Número de días faltantes para la expiración de la contraseña.
- ✓ *Expire:* Número de días desde que expiró la contraseña y que la cuenta ha sido deshabilitada.
- ✓ *Disable:* Número de días desde de enero 1 de 1970 que la cuenta de usuario ha sido deshabilitada.
- ✓ *Reserved:* Campo reservado.

Cabe anotar que el acceso a este archivo es restringido, ya que es peligroso que cualquier usuario vea o copie los ***passwords*** encriptados.

4.4.3 Cambio de password

Todos los usuarios en UNIX tienen asignado un *password* o contraseña. El *password* de cada usuario se cambia con el comando ***passwd.*** Este comando pregunta por el nuevo *password* (dos veces, para asegurarse que tecleó el *password* correcto). El usuario teclea el nuevo *password*, pero no se muestra en pantalla por razones de seguridad. El nuevo *password* surte efecto desde el momento en que se cambia. Si el usuario entra al sistema de nuevo, se le preguntaría por este nuevo *password* para permitir el acceso.

4.4.4 Protección de archivos

El sistema operativo UNIX provee un mecanismo llamado permisos de archivos, que permite a los archivos ser propiedad de un determinado usuario.

Tipos de archivos

En UNIX la estructura de directorios se encuentra clasificada de acuerdo a los tipos de archivos que en el sistema se encuentren, existe un carácter que define el tipo de archivo.

Socket: **s.**
Enlace simbólico: **l.**
Directorio: **d.**
Dispositivo de caracteres: **c.**
Dispositivo de bloque: **b.**
Archivo corriente: **-.**

4.4.5 Permisos otorgados al dueño, grupo y cualquier otro usuario del sistema.

Cada archivo en UNIX tiene asociado una serie de permisos que le permiten ser accesados o no por los usuarios. Los permisos de archivos caen dentro de tres grupos:

- ✓ Lectura (r): Permite a un usuario leer el contenido de un archivo, o si es un directorio, listar su contenido con ls.
- ✓ Escritura (w): Permite escribir y modificar un archivo. Para los directorios, este tipo de permiso permite crear nuevos archivos o borrar archivos dentro de ese directorio.
- ✓ Ejecución (x): Permite al usuario ejecutar un archivo. Para directorios, este permiso permite que el usuario pueda ubicarse en el directorio en cuestión usando el comando cd.

Existen tres clases de usuarios para los cuales se les asignan cada uno de estos permisos:

- ✓ El propietario del archivo.
- ✓ El grupo al cual pertenece el archivo
- ✓ Todos los usuarios, independientemente del grupo.

El comando ls con la opción -l muestra, entre otra información, los permisos de la siguiente manera:

-rwx r-x r--

- ✓ El primer carácter de la cadena de permisos ("-") representa el tipo de archivo.
- ✓ Las tres letras siguientes ("rwx"), representan los permisos otorgados al dueño del archivo. La "r" representa lectura, la "w" representa escritura, y la “x” representa ejecución.
- ✓ Los siguientes tres caracteres (“r-x”), representan los permisos del grupo en este archivo. Como solo aparece una r y una x, cualquier usuario que

pertenezca al grupo en cuestión sólo podrá leer el archivo o ejecutarlo, pero no modificarlo o borrarlo.

- ✓ Los últimos tres caracteres (“r--“), representan los permisos que tienen cualquier otro usuario en el sistema diferente al dueño del archivo o aquellos que pertenezcan al grupo. En este caso, como solo aparece una r, otros usuarios podrán leer el archivo, pero no escribir en él o modificarlo.

Un aspecto fundamental a tener en cuenta es que los permisos de un archivo también dependen de los permisos del directorio en el que se encuentra el archivo. Es decir, que para poder acceder a un archivo, primero se debe tener acceso de ejecución a todos los directorios que se encuentren en el path del archivo, y por supuesto, se debe tener el permiso respectivo en ese archivo.

4.4.6 Configuración de los permisos de archivos

Cuando se crea un archivo en UNIX, estos se crean sin tener en cuenta los derechos que se les da. Como primera medida de seguridad, es importante dar los permisos adecuados a cada archivo. Estos permisos pueden ser modificados sólo por el propietario del archivo o el superusuario (root). UNIX provee la utilidad **chmod** que nos permite alterar los permisos de un archivo. El comando **chmod** se usa de la siguiente forma:

- ✓ **chmod u+rwx programa.c** asigna los siguientes permisos al archivo:

  - rwx --- ---
- ✓ **chmod g+rw-x programa.c** asigna los siguientes permisos al archivo:

  - — rw- —
- ✓ **chmod o+r-wx programa.c** asigna los siguientes permisos al archivo:

- — — r--

- ✓ **chmod a+rw-x programa.c** asigna los siguientes permisos al archivo:

- rw- rw- rw-

Cualquier permiso puede ser añadido o suprimido mediante los símbolos “+” (signo más para añadir permisos) ó “-”(signo menos para suprimir permisos).

Los símbolos **u,g,o,a** permiten especificar la categoría del usuario:

- ✓ “u” para especificar la categoría de dueño o propietario.
- ✓ “g” para especificar la categoría del grupo al que pertenece.
- ✓ “o” para especificar a todos los usuarios.
- ✓ “a” para colocar los permisos en las tres categorías

Este tipo de modificaciones también se pueden hacer con un número en octal que varía entre 000 y 777.

Por ejemplo:

chmod 743 nombrearchivo

Asigna los siguientes permisos:

- rwx r-- -wx

El propietario y el grupo de un archivo son otras de las propiedades de un archivo o directorio que Linux permite que sean modificadas. Mediante la orden ***chown*** un propietario de un archivo puede transferir los privilegios que tiene sobre éste a otro usuario del sistema.

chown nombrenuevousuario nombredelarchivo

También es posible cambiar el grupo de un archivo empleando la orden ***chgrp.***

chgrp nombrenuevogrupo nombrearchivo

4.4.7 Permisos especiales

Existen dos tipos de permisos adicionales llamados permisos especiales que están disponibles para ejecutar archivos y directorios públicos. Cuando esos permisos son configurados, cualquier usuario puede ejecutar archivos, asumiendo los permisos del propietario para ejecutarlo.

Permiso setuid (set-user identification)

Cuando el **setuid** es configurado sobre un archivo ejecutable, cambia el **user id** de la persona que ejecuta el archivo por los del dueño del archivo. Esto permite a cualquier usuario ejecutar archivos y accesar a directorios que solo son accesados por el propietario, es decir, este permiso permite a un usuario normal ejecutar comandos a los cuales no tiene privilegios.

El permiso **setuid** se asigna a los archivos con el siguiente comando:

Chmod +s nombrearchivo

Permiso setgid (set-group identification)

El permiso **setgid** es similar a **setuid,** a excepción que el proceso identificación de grupo (GID), es cambiado al propietario del grupo del archivo, y un usuario tiene acceso basado sobre los permisos obtenidos para tal grupo. Cuando **setgid** es aplicado a un directorio, los archivos creados en ese directorio pertenecen al grupo que el directorio pertenezca, y no al grupo que pertenezca el proceso que lo creó. Cualquier usuario que tiene permiso de escritura en el directorio puede crear archivos ahí, sin embargo el archivo no pertenecerá al grupo del usuario, pero si pertenecerá al grupo del directorio.

El permiso **setgid** se asigna a los archivos con el siguiente comando:

Chmod +g nombrearchivo

Variable umask

La instrucción umask permite a un usuario definir una máscara de protección por defecto, acepta como parámetro un valor numérico (máscara) que indica los permisos que se negaran a todos los archivos creados a partir de ese instante.

umask [permisos_numéricos]

Cuando se utiliza sin parámetros, devuelve el valor actual de la máscara.
El valor asignado por umask es restado del que viene por defecto, esto tiene el efecto de negar permisos en la misma forma en que **chmod** los otorga.
Por ejemplo:

umask 022

Todos los archivos creados a partir de ese momento tendrán los siguientes permisos

rwx r-x r-x

Ya que

7	7	7	-	0	2	2	=	7	5	5
rwx	rwx	rwx		---	-w-	-w-		rwx	r-x	r-x

4.5 Protección en Windows NT

4.5.1 Servicios de Win32

Windows NT tiene un nivel de seguridad C2 según la clasificación de seguridad del Orange Book del DoD, lo que significa la existencia de control de acceso discrecional, con la posibilidad de permitir o denegar derechos de acceso para cualquier objeto partiendo de la identidad del usuario que intenta acceder al objeto. Para implementar el modelo de seguridad, Windows NT usa un descriptor de seguridad y listas de control de acceso (ACL), que a su vez incluyen dos tipos de entradas de control de acceso (ACE): las de permisos y las de negaciones de accesos.

Las llamadas al sistema de Win32 permiten manipular la descripción de los usuarios, los descriptores de seguridad y las ACL. A continuación se describen las llamadas de Win32 más frecuentemente usadas.

4.5.2 Dar valores iniciales a un descriptor de seguridad

El servicio InitializeSecurityDescriptor inicia el descriptor de seguridad especificado en psd con los valores de protección por defecto.

BOOL InitializeSecurityDescriptor (PSECURITY_DESCRIPTOR psd,
DWORD dwRevision);

dwRevision tiene el valor de la constante *SECURITY_DESCRIPTOR_REVISIÓN*. Esta llamada devuelve *TRUE* si psd es un descriptor de seguridad correcto y *FALSE* en otro caso.

4.5.3 Obtención del identificador de usuario

La llamada *GetUserName* permite obtener el identificador de un usuario que ha accedido al sistema.

BOOL GetUserName (LPTSTR NombreUsuario, LPDWORD LongitudNombre);

En caso de éxito, devuelve el identificador solicitado en *NombreUsuario* y la longitud del nombre en *LongitudNombre*. Esta función siempre debe resolverse con éxito, por lo que no hay previsto caso de error.

4.5.4 Obtención de la información de seguridad de un archivo

El servicio *GetFileSecurity* extrae el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.

BOOL GetFileSecurity (LPCTSTR NombreArchivo, SECURIY_INFORMATION secinfo, PSECURITY_DESCRIPTOR psd, DWORD cbsd, LPDWORD lpcbLongitud);

El nombre del archivo se especifica en *NombreArchivo*. *secinfo* es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. El argumento *psd* es el descriptor de seguridad en el que se devuelve la información de seguridad.

Esta llamada devuelve *TRUE* si todo es correcto y *FALSE* en otro caso.

4.5.5 Cambio de la información de seguridad de un archivo

El servicio *SetFileSecurity* fija el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.

BOOL SetFileSecurity (LPCTSTR NombreArchivo, SECURIY_INFORMATION secinfo, PSECURITY_DESCRIPTOR psd);

el nombre del archivo se especifica en *NombreArchivo*. El argumento *secinfo* es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. El argumento *psd* es el descriptor de seguridad en el que se pasa la información de seguridad. Esta llamada devuelve *TRUE* si todo es correcto y *FALSE* en otro caso.

4.5.6 Obtención de los identificadores de propietario y de su grupo para un archivo

Las llamadas *GetSecurityDescriptorOwner* y *GetSecurityDescriptorGroup* permiten extraer la identificación del usuario de un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo. Estas llamadas son sólo de consulta y no modifican nada.

BOOL GetSecurityDescriptorOwner (PSECURITY_DESCRIPTOR psd, PSID psidOwner);

BOOL GetSecurityDescriptorGroup (PSECURITY_DESCRIPTOR psd, PSID psidGroup);

El descriptor de seguridad se especifica en *psd*. El argumento *psidOwner* es un parámetro de salida donde se obtiene el identificador del usuario y *psidGroup* es un parámetro de salida donde se obtiene el grupo del usuario. En caso de éxito devuelve *TRUE* y si hay algún error *FALSE*.

4.5.7 Cambio de los identificadores del propietario y de su grupo para un archivo

Las llamadas SetSecurityDescriptorOwner y SetSecurityDescriptorGroup permiten modificar la identificación del usuario en un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo.

BOOL SetSecurityDescriptorOwner (PSECURITY_DESCRIPTOR psd, PSID psidOwner, BOOL fOwnerDefaulted);

BOOL SetSecurityDescriptorGroup (PSECURITY_DESCRIPTOR psd, PSID psidGroup, BOOL fGroupDefaulted);

El descriptor de seguridad se especifica en *psd*. El argumento *psidOwner* es un parámetro de entrada donde se indica el identificador del usuario y *psidGroup* es un parámetro de entrada donde se indica el grupo del usuario. El último parámetro de cada llamada especifica que se debe usar la información de

protección por defecto para rellenar ambos campos. En caso de éxito devuelve *TRUE* y si hay algún error *FALSE*.

4.5.8 Gestión de ACLs y ACEs

Las llamadas *InitializeACL*, *AddAccessAllowedAce* y *AddAccessDeniedAce* permiten inicializar una ACL y añadir entradas de concesión y denegación de accesos.

BOOL InitializeACL (PACL pACL, DWORD cbAcl, DWORD dwAclRevision);

pAcl es la dirección de una estructura de usuario de longitud *cbAcl*. El último parámetro debe ser *ACL_REVISIÓN*.

La ACL se debe asociar a un descriptor de seguridad, lo que puede hacerse usando la llamada *SetSecurityDescriptorDacl*.

BOOL SetSecurityDescriptorDacl (PSECURITY_DESCRIPTOR psd, BOOL fDaclPresent, PACL pACL, BOOL fDaclDefaulted);

psd incluye el descriptor de seguridad. El argumento *fDaclPresent* a *TRUE* indica que hay una ACL válida en *pACL*. *fdaclDefaulted* a *TRUE* indica que se debe iniciar la ACL con valores por defecto.

AddAccessAllowedAce y *AddAccessDeniedAce* permiten añadir entradas con concesión y denegación de accesos a una ACL.

BOOL AddAccessAllowedAce (PACL pAcl, DWORD dwAclRevision, DWORD dwAccessMask, PSID pSid);

BOOL AddAccessDeniedAce (PACL pAcl, DWORD dwAclRevision, DWORD dwAccessMask, PSID pSid);

pAcl es la dirección de una estructura de tipo ACL, que debe estar iniciada.
El argumento *dwAclRevision* debe ser *ACL_REVISIÓN*. El argumento *pSid* apunta a un identificador válido de usuario. El parámetro *dwAccessMask* determina los derechos que se conceden o deniegan al usuario o a su grupo. Los valores por defecto varían según el tipo de objeto.

PROCEDIMIENTO PARA REALIZAR LAS GUÍAS DE LABORATORIO

Las guías de laboratorio se desarrollan de la siguiente manera:

1. El docente da a conocer a los alumnos la teoría básica sobre el tema a tratar en el laboratorio. Los estudiantes además de recibir esta introducción, deben realizar investigaciones que conlleven a ampliar los conocimientos adquiridos. De esta manera, al momento de realizar la práctica, tendrán una fundamentación teórica que les permitirá agilizar el proceso de desarrollo de la misma.

2. A cada alumno se le hace entrega de la guía del estudiante, la cual contiene la siguiente estructura:

   - ✓ Titulo: especifica el tema de la guía de laboratorio.

   - ✓ Objetivos: expresan los resultados que se desean alcanzar con el desarrollo de la práctica. Permiten que el estudiante visualice hacia dónde se quiere llegar con el desarrollo de la práctica.

   - ✓ Marco teórico: es la base teórica con la cual el alumno se documenta para adquirir conocimientos que le permitan desarrollar las prácticas.

- ✓ Procedimiento: son pasos a seguir para poner en práctica la teoría suministrada en el marco teórico.

- ✓ Código fuente de las aplicaciones: ejemplos de programas que el estudiante podrá analizar para complementar la fundamentación teórica.

- ✓ Cuestionario: son actividades que el alumno deberá desarrollar después de haber realizado el laboratorio para evaluar el grado de comprensión de cada uno de los temas tratados en las guías.

- ✓ Bibliografía: son las fuentes que el estudiante puede consultar para ampliar y profundizar la información acerca de la API de Windows y las llamadas POSIX correspondientes a cada guía.

Es importante destacar que en las guías del estudiante no se describen por completo las llamadas al sistema y comandos que utilizan los sistemas operativos Linux y Windows para la manipulación de procesos. Esto se hace con el fin de que los alumnos a partir de la información que se les brinda, realicen investigaciones para profundizar los temas y así fomentar el autoaprendizaje.

El marco teórico de las guías del docente está más documentado y además contiene la solución de las actividades propuestas en las guías del estudiante, ya que el docente debe tener un amplio conocimiento de los temas para poder orientar a los estudiantes durante todo el proceso del desarrollo de la materia.

3. Por último, el alumno debe presentar al docente un informe final donde se registre el procedimiento realizado, los resultados, comentarios y conclusiones relevantes.

CONCLUSIONES

Después de implementar guías de laboratorio, los estudiantes lograron poner en práctica los conocimientos teóricos adquiridos durante el desarrollo de la materia Sistemas Operativos. Además, identificaron las principales diferencias de programación que existen entre los sistemas operativos Linux y Windows concluyendo lo siguiente:

- ✓ Linux al ser software de libre distribución, le brinda al usuario herramientas tales como comandos y llamadas al sistema que son más fáciles de implementar que las ofrecidas por el sistema operativo Windows, permitiendo así mayor comodidad en el momento de programar.

- ✓ Al trabajar con el sistema operativo Windows, se deben tener buenos conocimientos acerca de la plataforma porque de lo contrario, el sistema podría ser muy inestable al no saber manejarlo. Con Linux, podemos comenzar a programar con solo tener un poco de conocimiento acerca de la plataforma.

- ✓ Ambos sistemas operativos poseen sus propias características que pueden representar una ventaja o desventaja al momento de trabajar. El usuario es quien decide qué sistema utilizar dependiendo de las necesidades que quiera satisfacer. Por lo tanto, en ningún momento se puede afirmar que X o Y sistema es mejor que el otro.

- ✓ Durante el transcurso del semestre pusimos en práctica las guías de laboratorio y notamos que a los estudiantes que no tenían muchas habilidades para programar y los que nunca habían trabajado con el sistema operativo Linux, tuvieron inconvenientes en el momento de desarrollar las actividades propuestas en cada una de las guías. Por lo tanto, recomendamos que los estudiantes antes de realizar las actividades propuestas en las guías, deben saber cómo programar en los lenguajes de programación C/C++. Además, deben documentarse muy bien acerca de las diferentes llamadas al sistema que se utilizan para programar tanto en POSIX como en la API de Win32.

BIBLIOGRAFÍA

http://www.mandrakelinux.com/es/

http://www.monografias.com/trabajos6/linux/linux.shtml

http://enete.us.es/docu_enete/nt4/indice.asp

http://www.iespana.es/cpys/WinNT/3.pdf

http://support.ap.dell.com/docs/storage/73vexfms/sp/inswinnt.htm

http://temu.tco.plaza.cl/mg_tec/documentos/winnt/instalacion_nt.html

http://www.latiendadelfuturo.com/ANTAD/Html/Versatil/ManPOS/1.1InstalacionWindowsNT.pdf

http://www.atc.unican.es/~jagm/cii/Transparencias/procUnix.pdf

http://www.lab.dit.upm.es/~lprs/presentaciones/concurrencia_t8_1p.pdf

http://bernia.disca.upv.es/~eso/06-planificacion/06-planificacion.pdf

http://www.infor.uva.es/~arturo/Asig/InfASO/UCap6.html

http://www.lsi.us.es/docencia/cursos/seminario-1.html

http://aula.linux.org.ar/docs/tecnicos/index/cap2.html

http://bari.ufps.edu.co/personal/150802A/planificacion.htm

http://www.monografias.com/trabajos7/arso/arso.shtml#dise

http://www.cs.rpi.edu/courses/fall01/os/CreateProcess.html

http://www.adapower.com/os/win32-createprocess.html

http://www.global-shared.com/api/exec/crproces.html

http://winapi.conclase.net/

http://winapi.conclase.net/curso/index.php?000

http://aula.linux.org.ar/docs/tecnicos/index/cap3.html

http://redes.ens.uabc.mx/docencia/computacion/cursos/SO/prog_c/6_6_POSIX_THREADS.htm

http://redes.ens.uabc.mx/docencia/computacion/cursos/SO/prog_c/6_7_OTROS_PAQUETES.htm

http://studies.ac.upc.es/FIB/ISO/ISO-Comunicacion.pdf

http://bari.ufps.edu.co/personal/150802A/paso_mensajes.htm

http://www.kennedy.edu.ar/computacion/Apuntes%20y%20Utilidades/aptes-utilidades.htm

http://www.monografias.com/trabajos7/arso/arso.shtml#dise

http://linux.dsi.internet2.edu/docs/LuCaS/Manuales-LuCAS/DENTRO-NUCLEO-LINUX/dentro-nucleo-linux-html/dentro-nucleo-linux-5.html

http://www.dc.uba.ar/people/materias/so/datos/cap24.pdf

http://aula.linux.org.ar/docs/tecnicos/index/cap2.html

http://labsopa.dis.ulpgc.es/prog_c/IPC.HTM

http://docencia.ac.upc.es/FIB/ISO/ISO-lab-P6.pdf

http://www.tlm.unavarra.es/~daniel/docencia/arq/arq1998-1999/llamadas.pdf

http://bari.ufps.edu.co/personal/150802A/comunicacion.htm

http://shannon.unileon.es/~dielpa/Aero/Aero_Pract_3_sol.PDF

http://www.sis.ucm.es/SIS/UNIX/indice.htm

http://www2.ing.puc.cl/~iic10622/clases/unix.htm

http://bernia.disca.upv.es/~iripoll/seguridad/03-admin/03-admin.pdf

http://www.iti.upv.es/~pgaldam/sso/transparencias/pdf/tema2.pdf

SISTEMAS OPERATIVOS, Una visión aplicada, Jesús Carretero Pérez, Féliz García Carballeira, Pedro De Miguel Anasagasti, Fernando Pérez Costoya, Editorial Mc Graw Hill.

PROGRAMACIÓN EN LENGUAJE C, Schildt, Herbert, Editorial McGraw Hill, México.

PROGRAMACIÓN EN C++. ALGORITMOS, ESTRUCTURAS DE DATOS Y OBJETOS, Joyanes Aguilar Luis, Editorial McGraw Hill, 2000

PROGRAMACIÓN EN LINUX, Wall Kurt, Editorial Prentice may, 2000

ANEXOS A

GUÍA DEL ESTUDIANTE

INSTALACIÓN DE LINUX MANDRAKE Y WINDOWS NT
GUÍA DEL ESTUDIANTE

INSTALACIÓN DE LINUX MANDRAKE Y WINDOWS NT
GUÍA DEL ESTUDIANTE

OBJETIVOS

- ✓ Instalar y configurar el sistema operativo Linux Mandrake y Windows NT para que los estudiantes puedan conocer las ventajas que estos brindan con respecto a los demás sistemas operativos.

- ✓ Promover la introducción del Sistema Operativo Linux mediante un conocimiento básico de sus posibilidades y alcance.

- ✓ Conocer los requerimientos tanto de hardware como de software necesarios para un funcionamiento adecuado del sistema operativo.

MARCO TEÓRICO

LINUX

Linux es un poderoso Sistema Operativo creado en 1991 por Linus Torvalds y actualmente es sostenido por el trabajo de varios grupos de programadores distribuidos en varias partes del mundo.

GNU/Linux es un sistema operativo gratuito y de libre distribución bajo las condiciones que establece la licencia GPL (GNU Public License). Tiene todas las características que uno puede esperar de un sistema Unix moderno: multitarea real, memoria virtual, bibliotecas compartidas, carga por demanda, soporte de redes TCP/IP, entre muchas otras funcionalidades. Cuenta con las siguientes características: multitarea, multiusuario, multiplataforma, multiprocesador, mejor aprovechamiento de la memoria, control completo de tareas y procesos, soporte multinacional, compatibilidad con todo tipo de hardware y posee mayor estabilidad que otros sistemas operativos.

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

WINDOWS NT

Windows NT se trata de un sistema operativo de red de multitarea preferente, de 32 bits con alta seguridad y servicios de red, cuenta con las siguientes características: fiabilidad, rendimiento, portabilidad, compatibilidad, escalabilidad y seguridad.

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

INSTALACIÓN DE LINUX MANDRAKE

Antes de instalar

La máquina debe arrancar leyendo la unidad CD, si no es así se debe configurar la BIOS de la siguiente manera:

Lo primero es acceder a la BIOS, para ello durante el arranque de la misma muestra un mensaje que nos indica que pulsemos la tecla "Supr" para acceder a la BIOS. Una vez dentro de la BIOS iremos a la sección "BIOS FEATURES SETUP" y en ella modificaremos el valor de "Boot Sequence" a CDROM.

Comenzando la instalación

Para comenzar la instalación se introduce el CD #1 en la unidad lectora y se reinicia el ordenador. DrakX es el programa de instalación de Mandrake Linux. Tiene una interfaz de usuario gráfica la cual permite volver a las opciones de configuración previas en cualquier momento. Aparece una pantalla de bienvenida y luego aparece la pantalla de instalación donde en la parte izquierda se puede visualizar las diferentes fases de instalación y en el marco inferior una descripción detallada de cada una de ellas.

Eligiendo el idioma

El primer paso es elegir el idioma de instalación y uso del sistema. Al hacer clic sobre el botón Avanzado podrá seleccionar otros idiomas para instalar en su estación de trabajo.

Términos de licencia de la distribución

Antes de continuar, debería leer cuidadosamente los términos de la licencia. Los mismos cubren a toda la distribución Mandrake Linux, y si Usted no está de acuerdo con todos los términos en la misma, haga clic sobre el botón Rechazar, esto terminará la instalación inmediatamente. Para continuar con la instalación, haga clic sobre el botón Aceptar.

Clase de instalación

DrakX necesita saber si desea realizar una instalación por defecto (Recomendada) o si desea tener un control mayor (Experto).

Recomendada: elija esta si nunca ha instalado un sistema operativo GNU/Linux. La instalación será fácil y sólo se le formularán unas pocas preguntas.

Experto: si tiene un conocimiento bueno de GNU/Linux, puede elegir esta clase de instalación. La instalación experta le permitirá realizar una instalación altamente personalizada.

Configuración del ratón

DrakX generalmente detecta la cantidad de botones que tiene su ratón. Si no, asume que Usted tiene un ratón de dos botones y lo configurará para que emule el tercer botón. DrakX sabrá automáticamente si es un ratón PS/2, serie o USB.

Si desea especificar un tipo de ratón diferente, seleccione el tipo apropiado de la lista que se proporciona. Si elige un ratón distinto al predeterminado, se le presentará una pantalla de prueba. Use los botones y la rueda para verificar que la configuración es correcta. Si el ratón no está funcionando correctamente, presione la barra espaciadora o Intro para Cancelar y vuelva a elegir.

Configuración del teclado

Normalmente, DrakX selecciona el teclado adecuado para Usted (dependiendo del idioma que eligió). Sin embargo, podría no tener un teclado que se corresponde exactamente con su idioma. Haga clic sobre el botón Más para que se le presente la lista completa de los teclados soportados.

Selección de los puntos de montaje

Ahora necesita elegir el lugar de su disco rígido donde se instalará su sistema operativo Mandrake Linux.
Presione sobre la partición de Linux y luego en Crear y aparece una ventana que solicita el tamaño en MB, el tipo de sistema de archivos y el punto de montaje.

Ahora usted puede crear todas las particiones que necesite y después formatearlas para luego usarlas.

Elección de los paquetes a instalar

Ahora debe especificar los programas que desea instalar en su sistema. Los paquetes se ordenan en grupos que corresponden a un uso particular de su máquina. Los grupos en sí mismos están clasificados en cuatro secciones:

1. Estación de trabajo: si planea utilizar su máquina como una estación de trabajo, seleccione uno o más grupos correspondientes.
2. Desarrollo: si la máquina se utilizará para programación, elija el(los) grupo(s) deseado(s).
3. Servidor: finalmente, si se pretende usar la máquina como un servidor aquí puede seleccionar los servicios más comunes que desea que se instalen en la misma.
4. Entorno gráfico: seleccione aquí su entorno gráfico preferido.

Si mueve el cursor del ratón sobre el nombre de un grupo se mostrará un pequeño texto explicativo acerca de ese grupo.

Finalmente, dependiendo de si Usted elige seleccionar los paquetes individuales o no, se le presentará un árbol que contiene todos los paquetes clasificados por grupos y sub-grupos. Mientras navega por el árbol, puede seleccionar grupos enteros, sub-grupos, o simplemente paquetes.

Tan pronto como selecciona un paquete en el árbol, aparece una descripción del mismo sobre la derecha. Cuando ha finalizado con su selección, haga clic sobre el botón Instalar que lanzará el proceso de instalación.

Dependiendo de la velocidad de su hardware y de la cantidad de paquetes que se deben instalar, el proceso puede tardar un rato en completarse. En la pantalla se muestra una estimación del tiempo necesario para completar la instalación.

La instalación Mandrake Linux se divide en varios CD-ROMs. DrakX sabe si un paquete seleccionado se encuentra en otro CD y expulsará el CD corriente y le pedirá que inserte uno diferente cuando sea necesario.

Contraseña de Root

Este es el punto de decisión más crucial para la seguridad de su sistema GNU/Linux, tendrá que ingresar la contraseña de root. Root es el administrador del sistema y es el único autorizado a hacer actualizaciones, agregar usuarios, cambiar la configuración general del sistema, etc. Deberá elegir una contraseña que sea difícil de adivinar, DrakX le dirá si la que eligió es demasiado fácil. La contraseña debería ser una mezcla de caracteres alfanuméricos y tener al menos una longitud de 8 caracteres, recuerde que linux es sensible a las mayúsculas y a las minúsculas. Sin embargo, por favor no haga la contraseña muy larga o complicada debido a que Usted debe poder recordarla sin realizar mucho esfuerzo.

Agregar un usuario

GNU/Linux es un sistema multiusuario, y esto significa que cada usuario puede tener sus preferencias propias, sus archivos propios, y así sucesivamente. Pero, a diferencia de root, que es el administrador, los usuarios que agregue aquí no podrán cambiar nada excepto su configuración y sus archivos propios. Tendrá que crear al menos un usuario no privilegiado para Usted mismo. Primero tendrá que ingresar el nombre de usuario, el cual usará para ingresar al sistema. Luego tendrá que ingresar una contraseña. Si hace clic sobre Aceptar usuario, entonces puede agregar tantos como desee. Cuando haya terminado de agregar todos los usuarios que desee, seleccione Hecho.

Instalación del LILO

Lilo (Linux Loader) es un excelente gestor de arranque que nos permite seleccionar el sistema operativo con el que iniciar cada vez la máquina. El lugar habitual para instalarlo es el MBR (Master Boot Record). Este es necesario aún cuando Linux sea el único sistema en el computador. A continuación la instalación preguntará donde se desea poner el LILO, para ponerlo en el MBR seleccionar /dev/hda. No seleccione /dev/hda1 porque seguramente destruirá el sistema de archivos de Windows/DOS.

La instalación está completa y su sistema GNU/Linux está listo para ser utilizado. Simplemente haga clic sobre Aceptar para volver a arrancar el sistema. Puede iniciar GNU/Linux o Windows, cualquiera que prefiera (si está usando el arranque dual) tan pronto como su máquina haya vuelto a arrancar.

INSTALACIÓN DE WINDOWS NT

Windows NT se suministra con un entorno de instalación amigable y fácil de usar, detectando e instalado casi todo lo que se refiere a hardware él sólo.

Cuando se empieza una instalación lo primero que hay que hacer es calcular la plataforma que nos hace falta para instalar el sistema operativo para que su rendimiento sea eficaz y rápido.

Es importante tener en cuenta las siguientes recomendaciones:

Conviene que el sistema operativo y todos sus programas asociados estén en el mismo disco duro y si puede ser en la partición de arranque.

Que en esta partición se tenga todo el software de administración y soporte de hardware del sistema.

Cuando termine la instalación se debe hacer una copia de seguridad de este sistema de archivos en una unidad de backup.

Inicio de la instalación de Windows NT

La primera parte de la instalación de NT basada en modo texto, es idéntica para el servidor y la workstation.

El sistema operativo se suministra en formato CD, por lo tanto la plataforma tiene que tener instalada un lector de CD-ROM compatible con NT o estar conectado a una red que tenga uno compartido. Además, usted debe tener tres discos de inicio, si no es así debe generarlos. Los discos de inicio pueden generarse desde cualquier PC que tenga CD-ROM, basta con introducir el CD de NT, ir al directorio I386 y ejecutar la instrucción WINNT /OX, le pedirá tres diskettes formateados y vacíos.

Introduzca el CD-ROM en su lector, el disco 1 en la disquetera y después encienda el sistema.

Si está instalando NT en un equipo que pueda arrancar desde el lector de CDROM puede cambiar en la BIOS la secuencia de arranque de manera que empiece por el CD, aunque el disco duro esté sin formatear y consecuentemente sin ningún tipo de sistema operativo instalado, el programa de instalación se inicia sólo y permite hacer la instalación sin tener nada en el disco duro y sin tener que generar los diskettes de instalación.

Reconocimiento del sistema

Lo primero que hace es reconocer el hardware indispensable para empezar a trabajar y comprobar que no exista una versión de NT, en este caso se detendrá la instalación y tendrá que realizarla desde ese sistema NT ya instalado (usando WINNT32) o eliminar la partición donde estuviera ubicado. A continuación comenzará la carga de los archivos necesarios para la instalación y le pedirá que introduzca el disco 2 o en el caso de estar haciendo una instalación sin discos pasará a un menú donde pregunta:

Si queremos ayuda sobre la instalación (F1)

Si queremos instalar NT (ENTRAR)

Si queremos reparar (R), este apartado lo veremos en un próximo documento.

Si queremos salir de la instalación (F3)

Debe pulsar “ENTRAR”

Configuración de unidades de almacenamiento

Pasará a la fase de detección de los controladores ESDI/IDE, SCSI y unidades de CDROM conectadas, preguntándole si quiere detectar controladoras SCSI (ENTRAR) o no detectarlas “I”; éste sería el caso si no tuviera ningún dispositivo SCSI. Debe pulsar “ENTRAR”. Le pedirá el disco 3, aparece una pantalla con el resultado de la detección. Si no hubiera sido detectado alguno de sus discos duros o lectores de CDROM, tendría que instalar el driver del fabricante presionando “S”. Si los hubiera detectado todos pulse “ENTRAR”. Aparece en pantalla la licencia del producto la cual debe leer atentamente dando al avance página hasta que le permita dar “F8” para continuar, siempre que esté de acuerdo con las condiciones de la licencia. Seguidamente le dará un listado de componentes instalados en el sistema, los cuales podrá cambiar en caso necesario. Ahora pasa al gestor de particiones de disco y de ubicación de la instalación el cual pregunta: ¿Dónde quiere instalar NT? .Para ello debe moverse con el cursor hasta la partición donde quiera instalarlo y luego presione “ENTRAR”.

Si tiene espacio sin asignar muévase con el cursor a ese espacio no particionado y pulsando la tecla “C” cree una nueva partición. Lo más importante es tener un espacio de aproximadamente 300Mb para la instalación de NT.

Si quiere borrar una partición mueva el cursor a la partición existente y pulse “E”.

En su caso tendrá una partición FAT con el tamaño necesario para la instalación del NT, por lo que debe mover el cursor hasta situarlo encima de dicha partición y pulsar “ENTRAR”. Pasará a preguntarle si quiere convertir la partición a NTFS o dejarlo como está, con el cursor se moverá a la opción que desee. La instalación es más rápida sobre FAT pero recuerde que cuando termine la instalación tendrá que ejecutar CONVERT C: /FS:NTFS para convertir a NTFS, siempre que quiera convertir el sistema de archivo a este tipo.

Nota: NTFS le permite configurar permisos de seguridad sobre archivos y directorios; FAT es más rápido pero no tiene opciones de seguridad.

También le preguntará el directorio donde quiere ubicar el bloque de programas del NT, por defecto "\WINNT" y pasará a examinar los discos para comprobar su integridad, para ello pulse “ENTRAR”. Si considera que los discos están en perfecto estado pulse “ESC”.
Llegado a este punto el sistema se pondrá a copiar los archivos necesarios para la instalación del sistema NT, cuando acabe este proceso retire el disco de la disquetera y del CD-ROM y presione “ENTRAR”

Una vez pasada la primera parte de la instalación, se reinicia el ordenador y comienza la instalación basada en entorno gráfico.

Le saldrá una pantalla donde se indican los pasos que va a seguir la instalación, donde debe pulsar “SIGUIENTE”, y pasará a otra donde indicará el tipo de instalación que va a realizar:

Típica: Recomendada para la mayoría de los equipos
Portátil: Se instalará con opciones útiles para equipos portátiles
Compacta: Para ahorrar espacio en disco, no se instalará ninguno de los componentes opcionales
Personalizada: Para usuarios más avanzados. Puede personalizar todas las opciones de instalación disponibles

Seleccione la personalizada y pulse “SIGUIENTE”.

En el paso siguiente coloque el nombre y la organización a la que va a pertenecer la licencia, “SIGUIENTE”, e introduzca la clave del CD de NT, la cual viene en la carátula del CD, “SIGUIENTE”, pasará a poner el nombre que va ha tener el equipo para su reconocimiento en red, “SIGUIENTE”, y le preguntará la contraseña del administrador, “SIGUIENTE”, le pregunta si queremos un disco de rescate. El disco de rescate es importante por si existe un bloqueo o un fallo en el arranque de NT, este disco se tendrá que acuatizar cada cierto tiempo, y siempre antes de hacer un cambio en el equipo, sobre todo si es un cambio de hardware. En la pantalla siguiente debe seleccionar los componentes que desea instalar y pulsar “SIGUIENTE”.

Configurando el acceso a red

Si el equipo está conectado a una red a través de RDSI (ISDN) o un adaptador de red debe pulsar como activo en el cuadro a tal efecto. Si a su vez va ha tener control de acceso telefónico a redes también debe marcar el cuadro a tal efecto.

Si el equipo no va ha tener nada de lo anterior pulse el botón redondo que le indica tal opción (No conectar este equipo a una red en este momento).

Ahora pasará a la instalación de los protocolos con los que van a trabajar nuestro sistema, los cuales pueden ser TCP/IP, IPS/SPX, NetBEUI, pudiéndose seleccionar otros desde una lista o instalarlos desde un disco del fabricante.

Pantalla de los servicios de red

Sale un listado con los servicios mínimos de red que no se pueden tocar desde la instalación, en el caso que quiera quitar algunos tendrá que esperar a que se acabe la instalación. Ya habrá acabado la instalación de red pulse “SIGUIENTE”, si tiene alguna duda pulse “ATRÁS”. Ahora seguirá con la introducción de los datos del TCP/IP de su equipo, si tiene una dirección fija de red la puede colocar una vez activada la casilla a para tal efecto, con la mascara de red adecuada, si no tiene ningún ROUTER o GATEWAY para la solución de encaminamiento lo puede dejar en blanco, en caso de que existiera pondría la dirección de este. Ahora llega el momento de decirle si vamos a formar parte de un dominio NT o en un grupo de trabajo en el caso de trabajar en un dominio necesitamos la asistencia del administrador para que de alta la máquina. Pulse “SIGUIENTE”. Copiará el resto de los archivos, guardará la configuración y le pedirá que inserte un disco que etiquetará como “Disco de reparación” y pulse “ACEPTAR”, borrará los archivos temporales y le pedirá que reinicie.

REFERENCIAS

http://www.mandrakelinux.com/es/

http://www.monografias.com/trabajos6/linux/linux.shtml

http://www.mandrakelinux.com/es/

http://www.monografias.com/trabajos6/linux/linux.shtml

http://enete.us.es/docu_enete/nt4/indice.asp

http://www.iespana.es/cpys/WinNT/3.pdf

http://support.ap.dell.com/docs/storage/73vexfms/sp/inswinnt.htm

http://temu.tco.plaza.cl/mg_tec/documentos/winnt/instalacion_nt.html

http://www.latiendadelfuturo.com/ANTAD/Html/Versatil/ManPOS/1.1InstalacionWindowsNT.pdf

CONTROL Y ADMINISTRACIÓN DE PROCESOS

GUÍA DEL ESTUDIANTE

CONTROL Y ADMINISTRACIÓN DE PROCESOS

GUÍA DEL ESTUDIANTE

OBJETIVOS

- ✓ Conocer las llamadas a sistema de los sistemas operativos Unix y Windows para utilizarlas en la generación de aplicaciones.

- ✓ Identificar los parámetros que se deben tener en cuenta para el control y administración de procesos.

MARCO TEÓRICO

Un proceso es un programa en ejecución que tiene asociado código, datos, stack, registros e identificador único y requiere recursos (memoria, CPU, dispositivos de E/S, stack) para funcionar. A continuación describiremos las funciones principales para el control y administración de procesos en los sistemas operativos Linux y Windows NT.

LINUX

En Linux a cada proceso se le asigna un número de identificación o PID (Process ID) que cambia en cada ejecución del mismo, pero que permanece invariable mientras el programa se esté ejecutando; este número es asignado automáticamente por el sistema en tiempo de ejecución.

Llamadas al sistema para la administración de procesos

Crear Procesos

fork( ): el proceso padre se diferencia del proceso hijo por los valores PID (*Process ID)* y PPID (*Parent Process ID*) y por el valor de retorno de la llamada fork().
vfork( ): esta llamada permite crear procesos sin necesidad de copiar todo el espacio de direcciones del padre.

Proceso en espera

wait( ) : la llamada al sistema wait detiene al proceso que la invoca hasta que un hijo de éste termine.
waitpid( ): la llamada waitpid proporciona un método para esperar por un hijo en particular.

Ejecutar código de un proceso

El servicio **exec( )** de POSIX tiene por objetivo cambiar el programa que está ejecutando un proceso. Existe una familia de funciones exec: **excl., execv, execle, execve, execlp, execvp.**

Finalizar un proceso

Un proceso ejecuta su última instrucción y le pide al sistema operativo que lo borre, o ejecuta la llamada al sistema **exit( ).**

Identificación de procesos

getpid(): obtiene el identificador del proceso.
getppid(): obtiene el identificador del proceso padre.

Comandos para la administración de procesos

ps (process status): muestra los procesos que se están ejecutando y que fueron arrancados por el usuario actual**.**
pstree: en algunos sistemas está disponible el comando pstree, que lista los procesos y sus descendientes en forma de árbol.
top: esta herramienta monitorea varios recursos del sistema y tiene un carácter dinámico, muestra uso de CPU por proceso, cantidad de memoria, tiempo desde su inicio, etc.

nice: asigna la prioridad a los procesos, los valores nice oscilan desde -20 (mas prioridad) a 20 (menos prioridad), normalmente este valor se hereda del proceso padre.

kill: comando que se utiliza para eliminar un proceso.

bg: ordena a un proceso que está parado en segundo plano, que continúe ejecutándose en segundo plano.

fg: ordena a un proceso que está en segundo plano (parado o en funcionamiento) que vuelva al primer plano.

jobs: se usa para comprobar cuantos procesos se están ejecutando en segundo plano.

&: se usa en la línea de comando y con él le indicamos al sistema que la línea que se va a ejecutar deberá de ser puesta a trabajar en segundo plano. El "&" se pone al final de la línea a ejecutar.

Ctrl+Z: envía al segundo plano el programa que se esté ejecutando y lo detiene.

vmstat: el comando vmstat reporta varias estadísticas que mantiene el kernel sobre los procesos, la memoria y otros recursos del sistema.

Servicios POSIX para la planificación de procesos

Modificar los parámetros de planificación

sched_setparam( ): modifica la prioridad de un proceso.

sched_scheduler( ): modifica la prioridad y política de planificación de un proceso.

Obtener los parámetros de planificación

sched_getparam( ): devuelve la prioridad del proceso.

sched_getscheduler( ): devuelve la política de planificación del proceso.

Hilos (Threads)

Un thread, también llamado proceso liviano (lightweight process LWP) es una unidad básica de utilización de CPU, puede crearse a nivel de usuario y a nivel de kernel. Los threads a nivel de kernel se bloquean y despiertan en forma independiente de los demás, mientras que si un thread a nivel de usuario es bloqueado, causará que todo el proceso se bloquee.

Llamadas al sistema para la administración de hilos

pthread_create(): crea un nuevo hilo de ejecución, indicando los atributos del hilo.
pthread_equal(): compara si dos identificadores de hilo son el mismo.
pthread_exit(): finaliza el hilo que realiza la llamada.
pthread_join(): espera la terminación de un hilo específico.
pthread_self(): devuelve el identificador del hilo que realiza la llamada.
pthread_getschedparam(): obtiene la política de planificación y los parámetros del hilo especificado.
pthread_setschedparam(): establece la política de planificación y los parámetros del hilo especificado.
pthread_attr_init(): permite iniciar un objeto atributo que se puede utilizar para crear nuevos hilos.
pthread_attr_destroy(): destruye el objeto de tipo atributo.

Servicios POSIX para la planificación de hilos

Modificar los parámetros de planificación
pthread_setschedparam( ): Modifica la prioridad y política de planificación.

Obtener los parámetros de planificación

pthread_getschedparam( ): Devuelve la prioridad del proceso.

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

WINDOWS NT

En Windows NT se utilizan las siguientes llamadas al sistema para la administración de procesos:

Creación de un Proceso

En Win32 los procesos se crean mediante la llamada CreateProcess.

Terminación de un Proceso

Los servicios relacionados con la terminación de procesos se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso y servicios para esperar la terminación de un proceso:

1. Terminar la ejecución de un proceso: un proceso puede finalizar su ejecución de forma voluntaria de tres formas:

- ✓ Ejecutando dentro de la función *main* la sentencia *return.*
- ✓ Ejecutando el servicio ExitProcess.
- ✓ La función de la biblioteca de C *exit.* Esta función es similar a ExitProcess.

2. Esperar por la finalización de un proceso: En Win32 un proceso puede esperar la terminación de cualquier otro proceso siempre que tenga permisos para ello y disponga del manejador correspondiente. Para ello, se utilizan las siguientes funciones: WairForSingleObject y WaitForMultipleObjects.

Llamadas al sistema para la administración de hilos

Creación de procesos ligeros

En Win32, los procesos ligeros se crean mediante la función CreateThread.

Terminación de Procesos Ligeros

Al igual que con los procesos en Win32, los servicios relacionados con la terminación de procesos ligeros se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso ligero y servicios para esperar la terminación de procesos(WaitForSingleObject y WaitForMultipleObjects).

Las funciones del API Win32 relacionadas con la planificación se describen a continuación:

- ✓ Suspend / Resume Thread: suspende o reanuda un proceso detenido.
- ✓ Get / SetPriorityClass: devuelve o fija la clase de prioridad de un proceso.
- ✓ Get / SetThreadPriority: devuelve o fija la prioridad de un subproceso.
- ✓ Get / SetProcessAffinityMask: devuelve o fija la máscara de afinidad del proceso.

- ✓ SetThreadAffinityMask: fija la máscara de afinidad de un subproceso (debe ser un subconjunto de la máscara de afinidad del proceso) para un conjunto particular de procesadores, de forma que se restrinja la ejecución en esos procesadores.
- ✓ Get / SetThreadPriorityBoost: devuelve o fija la capacidad de Windows NT para incrementar la prioridad de un subproceso temporalmente (sólo se aplica a subprocesos en el rango dinámico).
- ✓ SetThreadIdealProcesor: establece el procesador preferido para un subproceso en particular, aunque no limita el subproceso a dicho procesador.
- ✓ Get / SetProcessPriorityBoost: devuelve o fija el estado de control de incremento de prioridad predeterminado del proceso actual.
- ✓ SwitchToThread: cede la ejecución del quantum a otro subproceso que está preparado para ejecutarse en el proceso actual.
- ✓ Sleep: pone el subproceso actual en el estado de espera durante un intervalo de tiempo especificado. El valor cero hace que se pierda el resto del quantum del subproceso.
- ✓ SleepEx: hace que el subproceso actual pase al estado de espera hasta que se termine una operación de E / S o se transcurra el intervalo de tiempo especificado.

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

DESARROLLO DE LA PRÁCTICA

PROCESOS

Linux

Procedimiento

1. El siguiente programa muestra la creación de dos procesos mediante la llamada al sistema fork( ), en donde el proceso hijo ejecuta las llamadas al sistema getpid( ) y getppid( ) para obtener el identificador de procesos propio y el de su padre, además ejecuta el comando ls -la mediante la llamada al sistema execlp( ). El proceso padre de forma similar obtiene su PID y el de su hijo, y ejecuta el comando ps -u utilizando la llamada al sistema execv( ). El código fuente es el siguiente:

```
#include <stdlib.h>
int main( int argc, char *argv[], char *env[] )
{
    pid_t id, id_padre, id_fork;  / / Declaracion de
                                  / / variables de tipo pid_t
                                  / / que guardan los
                                  / / identificadores de
```

```
                                        / / proceso
char *arg[3];
int estado;
switch(id_fork = fork())    / / Llamada al sistema para la
                                        / / creacion de p rocesos
{
   case -1:
                perror("Error en la llamada fork");
                exit(0);
                break;
   case 0:                  / / Codigo del proceso hijo
        id = getpid();    / / Comando para obtener el
                                / / ID del proceso
        id_padre = getppid(); / / Comando para obtener
                                / / el ID del proceso padre
        printf("\n\nHijo: Mi pid es: %d\n", id);
        printf("Hijo: Mi padre es: %d\n", id_padre);
        printf("Hijo: Ejecuto el comando ls -la: \n\n");
        execlp ("ls", "ls","-la", "./", (char *) 0);
        / / Llamada al sistema para ejecutar comandos
        perror ("Error en la ejecucion de ls -la");
        exit(0); / / Llamada al sistema para terminar el
                  / / proceso
                break;
   default:
        id = getpid();
        printf("\nPadre: Mi pid es: %d\n", id);
```

```
                printf("Padre: Mi hijo es: %d \n", id_fork);
                //La llamada al sistema fork() devuelve el PID del
                // proceso hijo al padre
                printf("Padre: Ejecuto ps -u \n\n");
                arg[0] = "ps";
                arg[1] = "-u";
                arg[2] = (char *)0;
                execv("/bin/ps", arg);      //Llamada al sistema para
                                            //ejecutar comandos
                perror("Error en la ejecucion de ls -l");
                wait(&estado);      //El proceso padre espera que el
                                    //proceso hijo
            }                       //termine de ejecutar su codigo.
        exit(0);  //Llamada al sistema para terminar el proceso
        }
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi, y guárdelo con el nombre procesos.c o busque el archivo procesos.c en el directorio ./Guia2/Linux.

3. Compile el archivo procesos.c de la siguiente manera:

```
gcc –o procesos procesos.c
```

4. Luego ejecútelo:

```
./procesos
```

Windows

Procedimiento

1. En este programa mediante la librería <windows.h> y las estructuras STARTUPINFO y PROCESS_INFORMATION de la función CreateProcess se crea un proceso que ejecuta el bloc de notas. El código fuente es el siguiente:

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
int main()
{
char CadenaProceso[50];
char WinDir[50];
STARTUPINFO si;
PROCESS_INFORMATION pi;
si.cb = sizeof(si); / /Tamaño de la estructura  STARTUPINFO
si.dwFlags=STARTF_USESHOWWINDOW; / /Para que se considere el
                                  / /miembro wShowWindow de
                                  / /dwFlags
si.wShowWindow = SW_SHOWNORMAL; / /La ventana del nuevo
                                 / /proceso es una ventana
                                 / /normal

GetWindowsDirectory(WinDir,sizeof(WinDir));
      strcpy(CadenaProceso,strcat(WinDir,"\\Notepad.exe"));
```

```
printf ("Directorio del nuevo proceso que se ejecuta:
%s\n",CadenaProceso);
if (!CreateProcess(CadenaProceso, // string aplicación
                   NULL,        // línea de comando
                   NULL,        // seguridad proceso
                   NULL,        // seguridad hilo
                   FALSE,       // no hay herencia de
                                                  //handles
                   0,           // sin banderas
                   NULL,        // hereda bloque de entorno
                   NULL,        // hereda directorio actual
                   &si,         // puntero a STARTUPINFO
                   &pi))        // puntero a PROCESS_INFORMATION
{
   if (GetLastError() == ERROR_FILE_NOT_FOUND)
   {
     printf ("Imposible encontrar el archivo. \n");
     return (-1);
   }
}
else{
  printf ("Se creo un proceso \n");
}
return 0;
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre CrearProceso.cpp o busque el archivo CrearProceso.cpp en el directorio C:\Guia2\Windows.

3. Compile el archivo CrearProceso.cpp

4. Ejecute el archivo CrearProceso.exe

HILOS

Linux

Procedimiento

1. El siguiente programa crea dos hilos que ejecutan dos funciones diferentes. El primer hilo muestra su ID mediante el uso de la llamada pthread_self() y el otro hilo ejecuta el comando ls –la. El código fuente es el siguiente:

```
#include <pthread.h>  //Libreria para utilizar hilos
#include <stdio.h>

void *ID()  //Funcion que ejecuta el hilo1
        {
            printf("\nHilo1: ID %d\n",pthread_self());
        }
        void *ejecutar()   //Funcion que ejecuta el hilo2
        {
            printf("\n\nHilo2: Ejecuto ls -la\n\n");
            execlp ("ls", "ls","-la", "./", (char *) 0);
            perror ("Error en la ejecucion de ls -la");
        }
        main()
        {
            pthread_t hilo1,hilo2;  //Declaración de los hilos
            pthread_create(&hilo1,NULL,ID,NULL);    //Creacion de los
        hilos
```

```
    pthread_create(&hilo2,NULL,ejecutar,NULL);
    pthread_join(hilo1,NULL);  //Espera que cada hilo //termine
    pthread_join(hilo2,NULL);
    exit(0);
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre hilos.c o busque el archivo hilos.c en el directorio ./Guia2/Linux.

3. Compile el archivo hilos.c de la siguiente manera:

```
gcc –o hilos hilos.c –lpthread
```

4. Luego ejecútelo:

```
./hilos
```

Windows

Procedimiento

1. Este programa mediante el uso de la función CreateThread, crea un hilo que muestra una figura geométrica.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
int a;
void Hilo()//Función que es ejecutada por el hilo
```

```
{
    int x,y;
    for (x=0; x<10; x++, printf("\n"))
        for (y=0; y<10; y++)
            printf("X");
    }
    void main(void)
    {
    HANDLE HandleHilo;
    DWORD IDHilo;
    HandleHilo=CreateThread(0, 0, (LPTHREAD_START_ROUTINE)
    Hilo,0, 0, &IDHilo);//Se crea el hilo
    if (HandleHilo==0)
        printf("No puedo crear el hilo. Error numero %x\n",
    GetLastError());
    a=getch();
    }
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre CrearProceso.cpp o busque el archivo Hilos.cpp en el directorio C:\Guia2\Windows.

3. Compile el archivo: Hilos.cpp

4. Ejecute el archivo: Hilos.exe

ENUNCIADO DE LA PRÁCTICA

Linux

1. Edite un programa en lenguaje C, que sea capaz de crear dos procesos en donde el proceso hilo se encargue de ejecutar comandos del shell tomados como datos de entrada y el proceso padre muestre su ID. Utilice las llamadas al sistema para el control de procesos.

2. Edite un programa en lenguaje C que cree un determinado número de hilos que modifiquen el valor de dos variables.

Windows

1. Elaborar un programa que mediante la librería <windows.h> y la función CreateProcess cree un proceso que ejecute la calculadora. Además, debe mostrar los siguientes datos: El handle del proceso nuevo, el handle del hilo hijo, el ID del proceso y el ID del hilo. Finalmente, debe cerrar el handle del proceso, el handle del hilo y explicar qué ocurre.

2. Elaborar un programa que cree un hilo que ejecute una función de incremento de una variable.

ANEXOS

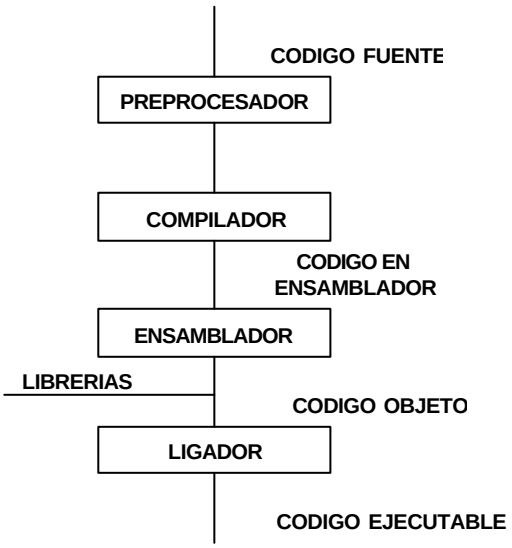
Compilación de programas en Windows

Para compilar los programas en windows se debe disponer del software Borland C++ o cualquier otro compilador de C que incluya la librería <windows.h>

Compilación de programas en Linux

Etapas de Compilación

El proceso de compilación involucra cuatro etapas sucesivas: preprocesamiento, compilación, ensamblado y enlazado.



- ✓ *Preprocesado:* En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con #define son sustituidas en el código por su valor en todos los lugares donde aparece su nombre.

- ✓ *Compilación:* La compilación transforma el código C en el lenguaje ensamblador propio del procesador de la máquina.

- ✓ *Ensamblado:* El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.

- ✓ *Enlazado:* Las funciones de C/C++ incluidas en el código, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones al ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.

Compilación en GCC (GNU Compiler Collection)

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr.

Sintaxis

gcc [opción | archivo]

Opciones

-c Realiza preprocesamiento y compilación, obteniendo el archivo en código objeto; no realiza el enlazado.

-E Realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.

-o *archivo* Indica el nombre del archivo de salida, cualesquiera sean las etapas cumplidas.

-I*ruta* Especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la I y la ruta, por ejemplo: -I/usr/include.

-L*ruta* Especifica la ruta hacia el directorio donde se encuentran los archivos de biblioteca con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la L y la ruta, por ejemplo: -L/usr/lib.

-Wall Muestra todos los mensajes de error y advertencia del compilador.

-g Incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB (GNU Debugger).

-v Muestra los comandos ejecutados en cada etapa de compilación y la versión del compilador. Es un informe muy detallado.

REFERENCIAS

http://www.atc.unican.es/~jagm/cii/Transparencias/procUnix.pdf

http://www.lab.dit.upm.es/~lprs/presentaciones/concurrencia_t8_1p.pdf

http://bernia.disca.upv.es/~eso/06-planificacion/06-planificacion.pdf

http://www.infor.uva.es/~arturo/Asig/InfASO/UCap6.html

http://www.lsi.us.es/docencia/cursos/seminario-1.html

http://aula.linux.org.ar/docs/tecnicos/index/cap2.html

http://bari.ufps.edu.co/personal/150802A/planificacion.htm

http://www.monografias.com/trabajos7/arso/arso.shtml#dise

http://www.cs.rpi.edu/courses/fall01/os/CreateProcess.html

http://www.adapower.com/os/win32-createprocess.html

http://www.global-shared.com/api/exec/crproces.html

http://winapi.conclase.net/

http://winapi.conclase.net/curso/index.php?000

SINCRONIZACIÓN DE PROCESOS
GUIA DEL ESTUDIANTE

SINCRONIZACIÓN DE PROCESOS
GUIA DEL ESTUDIANTE

OBJETIVOS

- ✓ Identificar, analizar y mostrar cómo se utilizan los principales mecanismos que ofrecen los sistemas operativos Linux y Windows para la sincronización de procesos.

- ✓ Trabajar con el sistema de hilos y las herramientas de concurrencia que ofrece el sistema operativo Unix (POSIX) en la implementación de regiones críticas mediante semáforos, mútex y variables de condición.

MARCO TEÓRICO

Normalmente, las aplicaciones constan de varios procesos ejecutándose de forma concurrente. Por lo tanto surgen necesidades:

- ✓ Compartir información entre los procesos.
- ✓ Intercambio de información entre los procesos.
- ✓ Sincronización entre los procesos.

La sincronización bajo memoria compartida plantea dos necesidades:

- ✓ Exclusión mutua
- ✓ Espera de una condición

LINUX

Los mecanismos de sincronización que se utilizan con más frecuencia en Linux son:

Semáforos

Un semáforo es un objeto con un valor entero al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: wait y signal. El estándar POSIX define dos tipos de semáforos:

- ✓ **Semáforos sin nombre**
- ✓ **Semáforos con nombre**

Creación de un semáforo sin nombre

Todos los semáforos en POSIX deben iniciarse antes de su uso. La función **sem_init** permite iniciar un semáforo sin nombre. El prototipo de este servicio es el siguiente:

int **sem_init**(sem_t *sem, int shared, int val);

Destrucción de un semáforo sin nombre

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada sem_init. Su prototipo es el siguiente:

int **sem_destroy**(sem_t *sem)

Creación y apertura de un semáforo con nombre

El servicio **sem_open** permite crear o abrir un semáforo con nombre.

sem_t ***sem_open**(char *name, int flag, mode_t mode, int val);
sem_t ***sem_open**(char *name, int flag);

Cierre de un semáforo con nombre

Cierra un semáforo con nombre rompiendo la asociación que tenía un proceso con un semáforo. El prototipo de la función es:

int **sem_close**(sem_t *sem);

Borrado de un semáforo con nombre

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función **sem_close.** El prototipo de este servicio es:

int **sem_unlink**(char *name);

Operación wait

La operación wait en POSIX se consigue con el siguiente servicio:

int **sem_wait**(sem_t *sem);

Operación signal

Este servicio se corresponde con la operación signal sobre un semáforo. El prototipo de este servicio es:

int **sem_post**(sem_t *sem);

Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito o –1 en caso de error.

Mútex

Son mecanismos de sincronización a nivel de hilos, se utilizan para garantizar la exclusión mutua en el acceso a un código.

Cada mútex posee:

- ✓ Dos estados internos: abierto y cerrado
- ✓ Un hilo propietario, cuando el mútex está cerrado

Llamadas POSIX:

Creación del mútex: pthread_mutex_init

- ✓ int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- ✓ pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

Destrucción del mútex: pthread_mutex_destroy

INT pthread_mutex_destroy(pthread_mutex_t * mutex);

Atributos de creación de mútex

- ✓ int pthread_mutexattr_init(pthread_mutexattr_t *attr);

- ✓ int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

Modificación de atributos de creación de mútex

- ✓ int pthread_mutexattr_getpshared (const pthread_mutexattr_t *attr, int *pshared);
- ✓ int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);
- ✓ int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);

- ✓ int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol);
- ✓ int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);
- ✓ int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int *prioceiling);
- ✓ int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
- ✓ int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int type);

Cierre y apertura de mútex

- ✓ int pthread_mutex_lock(pthread_mutex_t *mutex);
- ✓ int pthread_mutex_trylock(pthread_mutex_t *mutex);
- ✓ int pthread_mutex_unlock(pthread_mutex_t *mutex);

Variables de condición

Las variables de condición están asociadas con un mútex y se emplean para sincronizar los hilos. Las tres operaciones básicas son:

- ✓ Espera
- ✓ Aviso simple
- ✓ Aviso múltiple

Llamadas POSIX

Creación de variables de condición: pthread_cond_init

- ✓ int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);
- ✓ pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Destrucción de variables de condición: pthread_cond_destroy

int pthread_cond_destroy(pthread_cond_t *cond);

Atributos de creación de variables de condición

- ✓ int pthread_condattr_init(pthread_condattr_t *attr);
- ✓ int pthread_condattr_destroy(pthread_condattr_t *attr);

Modificación de atributos de creación de variables de condición

- ✓ int pthread_condattr_getpshared(const pthreadcondattr_t *attr, int *pshared);
- ✓ int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);

Espera sobre variables de condición

- ✓ int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- ✓ int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

Aviso sobre variables de condición

- ✓ int pthread_cond_signal(pthread_cond_t *cond);
- ✓ int pthread_cond_broadcast(pthread_cond_t *cond);

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

Windows NT

Como mecanismos de sincronización puros, Win32 dispone de secciones críticas, semáforos, mútex y eventos.

Secciones críticas

Los servicios utilizados para tratar las secciones críticas son los siguientes:

- ✓ InitializeCriticalSection
- ✓ DeleteCriticalSection
- ✓ EnterCriticalSection
- ✓ LeaveCriticalSection

Las dos primeras secciones se utilizan para crear y borrar secciones críticas. Las dos siguientes sirven para entrar y salir de la sección crítica.

Semáforos

En Win32 los semáforos tienen asociado un nombre. Los servicios de win32 para trabajar con semáforos son los siguientes:

- ✓ CreateSemaphore: crea un semáforo
- ✓ OpenSemaphore: abre un semáforo
- ✓ CloseHandle: cierra un semáforo

Mútex

Los servicios utilizados en Win32 para trabajar con mútex son los siguientes:

- ✓ CreateMutex: crea un mutex
- ✓ OpenMutex: abre un mutex
- ✓ CloseHandle: cierra un mutex

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

DESARROLLO DE LA PRÁCTICA

LINUX

Procedimiento

A continuación se muestra el problema de productores y consumidores, donde uno o más procesos productores generan una serie de datos que se depositan en un área de memoria compartida (buffer), de donde son extraídos por uno o más consumidores para procesarlos. Teniendo en cuenta que:

a. Un productor no puede depositar un dato cuando el buffer está lleno.
b. Un consumidor no puede extraer un dato cuando el buffer está vacío.

Semáforos

1. El código fuente del problema de productores/consumidores utilizando semáforos es el siguiente:

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#define TAMANO_BUFFER 1024 //Tamaño del Buffer
```

```
void * Consumidor(void);
void * Productor(void);
char Buffer[TAMANO_BUFFER]; / /Buffer Comun
char Mensaje[TAMANO_BUFFER];
int Leer = 0, Escribir = 0;
int Longitud;
sem_t Lleno, Vacio, Mutex; / /Declaración de semáforos
main()
{
    pthread_t Hilo_Productor,Hilo_Consumidor; / /Declaración
                                              / / de los hilos
    strcpy(Mensaje,"HOLA");
    Longitud = strlen(Mensaje);
    sem_init(&Lleno, 0, TAMANO_BUFFER);  / /Inicializa el
                                         / /semáforo LLeno
    sem_init(&Vacio, 0, 0);              / /Inicializa el
                                         / /semáforo Vacio
    sem_init(&Mutex, 0, 1);              / /Inicializa el
                                         / /semáforo Mutex
    system("clear");
    printf("Productor    Consumidor \n");
        pthread_create(&Hilo_Consumidor, NULL, (void
        *)Consumidor, NULL);  / /Creación del hilo consumidor
        pthread_create(&Hilo_Productor, NULL, (void
        *)Productor, NULL);   / /Creación del hilo productor
        pthread_join(Hilo_Consumidor, NULL);
         / /Espera que el hilo consumidor termine
```

```
        pthread_join(Hilo_Productor, NULL);
          //Espera que el hilo productor termine
}
void * Productor(void)
{    int n = 0;
     strcpy(Buffer, "");
     while (n < 20)
     {
               sem_wait(&Lleno); //Espera que el buffer no esté
                                 //lleno para producir
               sem_wait(&Mutex); //Bloquea el mutex para poder
                                 //escribir en el buffer
               Buffer[Escribir] = Mensaje[Escribir%Longitud];
               printf("    %c\n",Buffer[Escribir]);
               Escribir = (Escribir + 1) % TAMANO_BUFFER;
               fflush(stdout);
               sem_post(&Mutex);  //Desbloquea el mutex
               sem_post(&Vacio);
               if (rand() % 4 < 2)
                        sleep(1);
               n ++;
     }
}

void * Consumidor(void)
{
     int n = 0;/*Numero de elementos en el buffer*/
```

```
        while (n < 20)
        {
                sem_wait(&Vacio);//  Espera que el buffer tenga
                                        //algún elemento
                sem_wait(&Mutex);//Bloquea el mutex para poder
                                        //leer en el buffer
                printf("                %c\n",Buffer[Leer]);
                Leer = (Leer + 1) % TAMANO_BUFFER;
                fflush(stdout);
                sem_post(&Mutex); //Desbloquea el mutex
                sem_post(&Lleno);
                if (rand() % 4 < 2)
                        sleep(1);
                n ++;
        }
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre Semaforo.c o busque el archivo Semaforo.c en el directorio ./Guia3/Linux.

3. Compile el archivo Semaforo.c de la siguiente manera:

```
gcc –o Semaforo Semaforo.c –lpthread
```

4. Luego ejecútelo:

```
./semáforo
```

Variables Compartidas y mútex

1. El código fuente del problema de productores/consumidores utilizando variables compartidas y mutex es el siguiente:

```
#include <pthread.h>
#include <stdio.h>
#define BUFFER 1024
#define DATOS  20
void * Productor(void);
void * Consumidor(void);
pthread_mutex_t Mutex; //Declaración del mutex
pthread_cond_t Lleno;  //Declaración de las variables de
condición
pthread_cond_t Vacio;
int n=0;
int Buffer[BUFFER];
main(int arg, char *arg1[])
{
    pthread_t Hilo_Productor,Hilo_Consumidor;
    pthread_mutex_init(&Mutex, NULL); //Inicialización del
```

```
                                        / / mutex
    pthread_cond_init(&Lleno, NULL);  / / Inicialización de
                                      / / las variables de condición
    pthread_cond_init(&Vacio, NULL);
    system("clear");
    printf("Productor    Consumidor \ n");
    pthread_create(&Hilo_Productor, NULL, (void *) Productor,
NULL);  / / Creación del hilo productor
    pthread_create(&Hilo_Consumidor, NULL, (void *)
Consumidor, NULL); / / Creación del hilo consumidor
    pthread_join(Hilo_Productor, NULL);  / / Espera que el
                                         / / hilo productor termine
    pthread_join(Hilo_Consumidor, NULL);  / / Espera que el
                                         / / hilo consumidor termine
    pthread_mutex_destroy(&Mutex);  / / Destruye el mutex
    pthread_cond_destroy(&Lleno);   / / Destruyen las
                                         / / variables de condición
    pthread_cond_destroy(&Vacio);
    exit(0);
}
```

```
void * Productor(void)
{
    int dato, i, pos=0;
    for (i=0; i<DATOS; i++)
    {
            dato=i+1;
            pthread_mutex_lock(&Mutex);
          / / Bloquea el mutex para poder escribir en el buffer
            while (n==BUFFER)
                    pthread_cond_wait(&Lleno, &Mutex);
            Buffer[pos]=i;
            pos = (pos + 1) % BUFFER;
            printf("     %d \ n",pos);
            n = n + 1;
            if (n==1)
                    pthread_cond_signal(&Vacio);
            pthread_mutex_unlock(&Mutex);   / / Desbloquea el
                                                / / mutex
            if (rand() % 4 < 2)
                    sleep(1);
    }
```

```
    pthread_exit(0);
}

void * Consumidor(void)
{
    int dato, i, pos=0;
    for (i=0; i<DATOS; i++)
    {
        pthread_mutex_lock(&Mutex); / /Bloquea el mutex
                                    / /para poder leer en el buffer
        while (n==0)
            pthread_cond_wait(&Vacio, &Mutex);
        dato=Buffer[pos];
        pos = (pos + 1) % BUFFER;
        printf("          %d \ n",pos);
        n = n - 1;
        if (n==BUFFER-1)
            pthread_cond_signal(&Lleno);
        pthread_mutex_unlock(&Mutex); / /Desbloquea el
                                      / /mutex
```

```
            if (rand() % 4 < 2)
                sleep(1);
        }
        pthread_exit(0);
    }
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre MutexVble.c o busque el archivo MutexVble.c en el directorio ./Guia3/Linux.

3. Compile el archivo MutexVble.c de la siguiente manera:

```
gcc –o MutexVble MutexVble.c -lpthread
```

4. Luego ejecútelo:

```
./MutexVble
```

Windows

Sección Crítica

1. El siguiente programa crea seis hilos, cada uno de los cuales incrementa 10 veces el valor de una variable. La variable es compartida, por lo tanto se debe sincronizar el acceso de los hilos a ésta para que su valor no difiera entre

ejecuciones del programa. El mecanismo de sincronización que se utiliza es *Sección Crítica.*

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define numeroHilos 6
volatile INT c;
int a;
CRITICAL_SECTION SecCritica;
void HiloContador(INT iteraciones)
{
INT i;
INT x;
for (i=0; i<iteraciones; i++)
{
EnterCriticalSection(&SecCritica);
x=c;
x++;
c=x;
LeaveCriticalSection(&SecCritica);
}
}
void main(void)
{
HANDLE handles[numeroHilos];
DWORD hiloID;
```

```
INT i;
InitializeCriticalSection(&SecCritica);
for (i=0; i<numeroHilos; i++)
{
handles[i]=CreateThread(0, 0,
(LPTHREAD_START_ROUTINE) HiloContador,
(VOID *) 10, 0, &hiloID);
}
WaitForMultipleObjects(numeroHilos, handles, TRUE,
INFINITE);/ /Espera a que todas las hebras finalicen su
        / / ejecución
DeleteCriticalSection(&SecCritica);
printf("El contador vale %d \ n", c);
a=getch();
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre SecCritica.cpp o busque el archivo SecCritica.cpp en el directorio C:\Guia3\Windows.

3. Compile el archivo:

SecCritica.cpp

4. Ejecute el archivo:

SecCritica.exe

Semáforo

1. El siguiente programa muestra la solución al problema *productor – consumidor* con buffer acotado utilizando Semáforo. El búfer es una variable, y los hilos productor y consumidor se alternan en su ejecución.

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <conio.h>
int a;
int BuferCompartido;
HANDLE SEMAFORO;
void Productor() / /Función del productor
{
int i;
for (i=20; i>=0; i–) {
if (WaitForSingleObject(SEMAFORO,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: En productor\ n");
ExitThread(0);
}
printf("Produce: %d \ n", i);
BuferCompartido = i;
ReleaseSemaphore(SEMAFORO, 1, NULL); / /Fin de la sección
                                        / /critica
}
}
```

```
void Consumidor() / /Función del consumidor
{
int resultado;
while (1) {
if (WaitForSingleObject(SEMAFORO,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: consumidor()\n");
ExitThread(0);
}
if (BuferComparti do == 0) {
printf("Consumido %d: ultimo dato\n",BuferCompartido);
ReleaseSemaphore(SEMAFORO, 1, NULL); / /Fin de sección critica
ExitThread(0);
}
if (BuferCompartido > 0){
resultado = BuferCompartido;
printf("Consumido: %d \n", resultado);
ReleaseSemaphore(SEMAFORO, 1, NULL); / /Fin de sección critica
}
}
}
void main()
{
HANDLE hVectorHilos[2];
DWORD HebraID;
BuferCompartido = -1;
SEMAFORO = CreateSemaphore(0, 1, 2, "SemaforoContador");
hVectorHilos[0] = CreateThread (NULL, 0,
```

```
(LPTHREAD_START_ROUTINE)Productor,
NULL, 0, (LPDWORD)&HebraID);
hVectorHilos[1] = CreateThread (NULL, 0,
(LPTHREAD_START_ROUTINE)Consumidor,
NULL, 0, (LPDWORD)&HebraID);
WaitForMultipleObjects(2,hVectorHilos,TRUE,INFINITE);
a=getch();
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre productorconsumidor_semaforo.cpp o busque el archivo productorconsumidor_semaforo.cpp en el directorio C:\Guia3\Windows.

3. Compile el archivo:

productorconsumidor_semaforo.cpp

4. Ejecute el archivo:

productorconsumidor_semaforo.exe

Mútex

1. El siguiente programa muestra la solución al problema *productor – consumidor* con buffer acotado utilizando Mutex. El búfer es una variable, y los hilos productor y consumidor se alternan en su ejecución.

```
#include <stdio.h>
```

```
#include <windows.h>
#include <stdlib.h>
#include <conio.h>
int a;
int BuferCompartido;
HANDLE hMutex;
void Productor() / /Función del productor
{
int i;
for (i=10; i>=0; i–) {
if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: En productor\ n");
ExitThread(0);
}
printf("Produce: %d \ n", i);
Sleep(100);
BuferCompartido = i;
ReleaseMutex(hMutex); / / fin de la sección critica
}
}
void Consumidor() / / Función del consumidor
{
int resultado;
while (1) {
if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: consumidor()\ n");
ExitThread(0);
```

```
}
if (BuferCompartido == 0) {
printf("Consumido %d: ultimo dato \ n",BuferCompartido);
ReleaseMutex(hMutex); / / Fin de sección critica
ExitThread(0);
}
if (BuferCompartido > 0){
resultado = BuferCompartido;
printf("Consumido: %d \ n", resultado);
ReleaseMutex(hMutex); / / fin de sección critica
}
}
}
void main()
{
HANDLE hVectorHilos[2];
DWORD HebraID;
BuferCompartido = -1;
hMutex = CreateMutex(NULL,FALSE,NULL);
hVectorHilos[0] = CreateThread (NULL,
        0,(LPTHREAD_START_ROUTINE)Productor,
NULL, 0, (LPDWORD)&HebraID);
hVectorHilos[1] = CreateThread (NULL,
        0,(LPTHREAD_START_ROUTINE)Consumidor,
NULL, 0, (LPDWORD)&HebraID);
WaitForMultipleObjects(2,hVectorHilos,TRUE,INFINITE);
a=getch();
```

}

2. Copie el código fuente en un editor de texto y guárdelo con el nombre productorconsumidor_mutex.cpp o busque el archivo productorconsumidor_mutex.cpp en el directorio C:\Guia3\Windows.

3. Compile el archivo:

productorconsumidor_mutex.cpp

4. Ejecute el archivo:

productorconsumidor_ mutex.exe

ENUNCIADO DE LA PRÁCTICA

1. Elaborar cuatro programas en donde mediante la creación de hilos incremente n veces el valor de una variable. La variable es compartida, por lo tanto se debe sincronizar el acceso de los hilos a ésta para que su valor no difiera entre ejecuciones del programa. Los mecanismos de sincronización que debe utilizar son:

Linux	**Windows**
Semáforos	Semáforos
Mútex con Variables Condicionales	Mútex

2. Elaborar un programa que utilizando el mecanismo de *Sección Crítica,* sincronice el acceso a un determinado archivo ya sea en modo lectura o escritura. Recuerde incluir la librería <windows.h>

REFERENCIAS

http://aula.linux.org.ar/docs/tecnicos/index/cap3.html

http://redes.ens.uabc.mx/docencia/computacion/cursos/SO/prog_c/6_6_POSIX_THREADS.htm

http://redes.ens.uabc.mx/docencia/computacion/cursos/SO/prog_c/6_7_OTROS_PAQUETES.htm

http://studies.ac.upc.es/FIB/ISO/ISO-Comunicacion.pdf

http://bari.ufps.edu.co/personal/150802A/paso_mensajes.htm

http://www.kennedy.edu.ar/computacion/Apuntes%20y%20Utilidades/apte s-utilidades.htm

http://www.monografias.com/trabajos7/arso/arso.shtml#dise

COMUNICACIÓN DE PROCESOS
GUÍA DEL ESTUDIANTE

COMUNICACIÓN DE PROCESOS
GUÍA DEL ESTUDIANTE

OBJETIVOS

- ✓ Identificar, analizar y mostrar cómo se utilizan los principales mecanismos que ofrecen los sistemas operativos Linux y Windows para la comunicación de procesos.

- ✓ Intercambiar información entre procesos diferentes empleando los mecanismos de tuberías, con y sin nombre, que proporciona el sistema operativo UNIX y WINDOWS.

MARCO TEÓRICO

El sistema operativo provee mecanismos para que los procesos se puedan intercomunicar. La intercomunicación se puede realizar utilizando almacenamiento compartido (se comparte algún medio de almacenamiento entre ambos procesos) o utilizando el sistema de intercambio de mensajes (se utilizan llamadas al sistema para enviar y recibir mensajes).

LINUX

El sistema operativo UNIX permite que procesos diferentes intercambien información entre ellos. Para procesos que se están ejecutando bajo el control de una misma máquina permite la comunicación mediante el uso de:

- ✓ Tuberías
- ✓ Memoria compartida
- ✓ Semáforos
- ✓ Colas de mensajes.

Mecanismos IPC (Inter process Communication - Comunicación entre Procesos)

Los medios IPC de Linux proporcionan un método para que múltiples procesos se comuniquen unos con otros. Hay varios métodos de IPC disponibles:

Tuberías (Pipes)

Una tubería (*pipe)* se puede considerar como un canal de comunicación entre dos procesos.

Clasificación de tuberías

Se pueden distinguir dos tipos de tuberías dependiendo de las características de los procesos que pueden tener acceso a ellas:

✓ **Sin nombre**. Solamente pueden ser utilizadas por los procesos que las crean y por los descendientes de éstos.

Creación:

int **pipe**(int fd[2])

Cierre

int **close**(int fd);

Escritura

int **write**(int fd, char *buffer, int n);

Lectura

int **read**(int fd, char *buffer, int n);

✓ **FIFO (pipe con nombre)**. Se utilizan para comunicar procesos entre los que no existe ningún tipo de parentesco. La llamada al sistema para crear una *fifo* es *mkfifo*:

int **mkfifo** (char *ruta, mode_t modo);

Llamadas al sistema

⇒ Open (char *name, int flag);

- Abre un FIFO (para lectura, escritura o ambas)
- Bloquea hasta que haya algún proceso en el otro extremo

⇒ Lectura y escritura mediante *read()* y *write()*

Igual semántica que los pipes

⇒ Cierre de un FIFO mediante *close()*

⇒ Borrado de un FIFO mediante *unlink()*

System V IPC

Se conocen como System V IPCs a tres técnicas de comunicación entre procesos que provee el UNIX System V:

⇒ *Memoria compartida:* Provee comunicación entre procesos permitiendo que éstos compartan zonas de memoria.

Llamadas al sistema:

- ✓ int shm_open(char *name, int oflag, mode_t mode): Crea un objeto de memoria a compartir entre procesos
- ✓ int shm_open (char *name, int oflag): Sólo aperture
- ✓ int close(int fd): Cierre. El objeto persiste hasta que es cerrado por el último proceso.
- ✓ int shm_unlink(const char *name): Borra una zona de memoria compartida.
- ✓ void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off): Establece una proyección entre el espacio de direcciones de un proceso y un descriptor de fichero u objeto de memoria compartida.
- ✓ void munmap(void *addr, size_t len): Desproyecta parte del espacio de direcciones de un proceso comenzando en la dirección ***addr***. El tamaño de la región a desproyectar es ***len***.

⇒ *Colas de mensajes:* Permiten tanto compartir información como sincronizar procesos. Un proceso envía un mensaje y otro lo recibe. El kernel se encarga de sincronizar la transmisión / recepción.

Llamadas al sistema:

Msgget()

Para crear a una nueva cola de mensajes o acceder a una ya existente, usaremos la llamada al sistema msgget( ).

int msgget (key_t llave,int msgflag);

Msgsnd() y Msgrcv()

Estas llamadas al sistema se utilizan para escribir y leer de una cola:

int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

Msgctl()

Esta función nos permite modificar la información de control que el Kernel le asocia a cada cola (identificada esta por su idmsg).

int msgctl (int msqid, int cmd, struct msqid_ds * buf);

Es conveniente recordar que estos objetos IPC no se van a menos que se quiten apropiadamente, o el sistema se reinicie. Y el uso de la función msgctl( ) es la forma adecuada de deshacernos de una cola de mensajes que ya cumplió su cometido.

Señales

Una señal es una interrupción software enviada a un proceso. El sistema operativo emplea las señales para informar acerca de situaciones excepcionales a un proceso. Cualquier proceso puede enviar a otro proceso una señal, siempre que tenga permiso, cuando un proceso se prepara para la recepción de una señal, puede realizar las siguientes acciones:

- ✓ Ignorar la señal
- ✓ Realizar la acción asociada por defecto a la señal
- ✓ Ejecutar una rutina del usuario asociada a dicha señal.

Llamadas al sistema:

- ✓ ***signal():*** Asocia una acción determinada con una señal.
- ✓ ***kill( )*:** Envía una señal a un proceso
- ✓ ***alarm( )*:** Envía una señal de alarma en un período de tiempo especificado.
- ✓ ***sigemptyset( )*:** Crea un conjunto de señales vacío.
- ✓ ***sigfillset( )*:** Crea un conjunto de señales completo.
- ✓ ***sigaddset( )*:** Añade una señal al conjunto.
- ✓ ***sigdelset( )*:** Elimina una señal del conjunto.
- ✓ ***sigprocmask( )*:** Examina o modifica un conjunto de señales de proceso.
- ✓ ***sigaction( )*:** Especifica la acción a realizar cuando un proceso recibe una señal.
- ✓ ***sigsuspend( )*:** Suspende un proceso hasta que reciba una señal del conjunto.
- ✓ ***alarm(sec):*** El kernel le enviará al proceso llamante una señal SIGALRM dentro de sec segundos.

- ***pause( ):*** El proceso queda bloqueado hasta que le llegue una señal.
- ***time ( ):*** el kernel consulta el reloj y devuelve su valor como retorno de la llamada.

Algunas señales

- SIGHUP Colgar. Generada al desconectar el terminal.
- SIGINT Interrupción. Generada por teclado.
- SIGILL Instrucción ilegal. No se puede capturar.
- SIGFPE Excepción aritmética, de coma flotante o división por cero.
- SIGKILL Matar proceso, no puede capturarse, ni ignorarse.
- SIGBUS Error en el bus.
- SIGSEGV Violación de segmentación.
- SIGPIPE Escritura en una pipe para la que no hay lectores.
- SIGALRM Alarma de reloj.
- SIGTERM Terminación del programa.

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

WINDOWS NT

Tuberías

Win32 ofrece dos tipos de tuberías: sin nombre y con nombre.

Los principales servicios que ofrece Win32 para trabajar con tuberías sin nombre son los siguientes:

CreatePipe: crea una tubería sin nombre

CreateNamedPipe: crea una tubería con nombre

CreateFile: abre una tubería

CloseHandle: cierra una tubería

ReadFile: leer de una tubería

WriteFile: escribir en una tubería

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

DESARROLLO DE LA PRÁCTICA

Linux

Pipe (Tubería)

Procedimiento

1. El siguiente programa muestra como dos procesos padre e hijo pueden comunicarse a través de una tubería.

```
#include <unistd.h>
#include <sys/types.h>
#define TAMANOPIPE 4096

   int main(void)
   {
        int tubo[2]; /*Descriptor de la tubería*/
        int n;
        pid_t id;
        char cadena[TAMANOPIPE];
        char line[TAMANOPIPE];
        if (pipe(tubo)<0) /* Se crea la tubería sin nombre */
        {
        perror("Error al crear la tuberia \n");
        exit(-1);
  }
  id=fork(); /* Se crea un proceso hijo*/
  if (id<0)
  {
```

```
            perror("Error al crear el proceso hijo");
            exit(-2);
        }
        if (id>0) /*Proceso PADRE*/
        {
            printf("\nPADRE: Introduzca una cadena de caracteres:\n");
            n=read(STDIN_FILENO,cadena,TAMANOPIPE);
            close (tubo[0]); /*Cierra el extremo de lectura de la tubería */
            write(tubo[1],cadena,n); /*Escribe en la tubería */
            close (tubo[1]); /*Cierra el extremo de escritura de la tubería*/
            wait((int *)NULL); /*Espera a que acabe un proceso hijo*/
            exit(0);
        }
        else /*Proceso HIJO*/
        {
            close (tubo[1]); /*Cierra el extremo de escritura de la tubería */
            read(tubo[0],line,TAMANOPIPE); /* Lee de la tubería */
            printf("\nHIJO: He recibido la cadena: \n");
            sleep (1);
            printf("%s",line);
            close (tubo[0]); /*Cierra el extremo de lectura de la tubería*/
            printf("\n");
            exit(0);
        }
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre pipe.c o busque el archivo pipe.c en el directorio ./Guia2/Linux.

3. Compile el archivo pipe.c de la siguiente manera:

gcc –o pipe pipe.c

4. Luego ejecútelo:

./pipe

Colas

Procedimiento

1. El siguiente programa muestra la comunicación entre procesos mediante el uso de colas de mensajes.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define     BUFFER      1024
#define     PERMISOS    0666  //Permisos de lectura y ejecución para el dueño y otros

#define KEY ((key_t) 7777)
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
main()
{
        int i, msqid;
        char mensaje[BUFFER];
struct
    {
        long  m_type;
        char  m_text[BUFFER];
     } msgbuffs, msgbuffr;
if ( (msqid = msgget(KEY, PERMISOS | IPC_CREAT)) < 0)
/ /Llamada la sistema para crear una cola
        perror("msgget error");
msgbuffs.m_type = 1L;
printf("\nEscriba el mensaje a enviar... \n");
scanf("%s",&mensaje);
strcpy(msgbuffs.m_text,mensaje);
if (msgsnd(msqid, &msgbuffs, BUFFER, 0) < 0)  / /Llamada al
sistema para escribir en la cola
                perror("msgsnd error");
printf("\nEl mensaje enviado es: %s \n", msgbuffs.m_text);
sleep(1);
if (msgrcv(msqid, &msgbuffr, BUFFER, 0L, 0) != BUFFER)
/ /Llamada al sistema para leer en la cola
                perror("msgrcv error");
printf("\nEl mensaje recibido es: %s \n\n", msgbuffr.m_text);
if (msgctl(msqid, IPC_RMID, (struct msqid_ds *) 0) < 0)
/ /Llamada al sistema para cerrar la cola
        perror("IPC_RMID error");
exit(0);
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre colas.c o busque el archivo colas.c en el directorio ./Guia2/Linux.

3. Compile el archivo colas.c de la siguiente manera:

gcc –o colas colas.c

4. Luego ejecútelo:

./colas

Windows

Procedimiento

1. El siguiente programa ejecuta el mandato dir | sort **.**

```
#include <windows.h>
#include <stdarg.h>
#include <stdio.h>
#include <conio.h>
int main (void)
{
        HANDLE hRead, hWrite;
        SECURITY_ATTRIBUTES Apipe = {
      sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
        STARTUPINFO info;
        PROCESS_INFORMATION pid1,pid2;

         //inicialización de la estructura STARTUPINFO
        memset(&info,0,sizeof(info));
        info.cb = sizeof(info);

if (!CreatePipe(&hRead,&hWrite,&Apipe,0)) //se crea el pipe
{
        perror("Failed to create pipe");
        return -1;
}

//se prepara la redirección para el primer proceso
info.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
```

```
    info.hStdOutput = hWrite;
    info.hStdError = GetStdHandle(STD_ERROR_HANDLE);
    info.dwFlags =
  STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;

  if (!CreateProcess(NULL,"cmd /c
  dir",NULL,NULL,TRUE,0,NULL,NULL,&info,&pid1))
    {
        perror("Failed to create process for \"dir\"");
        return -1;
    }

    CloseHandle(pid1.hThread);
    CloseHandle(hWrite);//cierra el pipe para escritura

    //redirige la salida estándar para el segundo proceso
    info.hStdInput = hRead;
    info.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    info.hStdError = GetStdHandle(STD_ERROR_HANDLE);
    info.dwFlags =
STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;

    if
(!CreateProcess(NULL,"sort",NULL,NULL,TRUE,0,NULL,NULL,&info,&
pid2))
    {
        perror("Failed to create process for \"sort\"");
        return -1;
    }
```

```
        CloseHandle(pid2.hThread);
        CloseHandle(hRead);

        se espera la terminación de los dos procesos
        WaitForSingleObject(pid1.hProcess,INFINITE);
        CloseHandle(pid1.hProcess);
        WaitForSingleObject(pid2.hProcess,INFINITE);
        CloseHandle(pid2.hProcess);

        getch();
        return(0);
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre Tuberia_1.cpp o busque el archivo Tuberia_1.cpp en el directorio C:\Guia4\Windows.

3. Compile el archivo: Tuberia_1.cpp

4. Ejecute el archivo: Tuberia_1.exe

ENUNCIADO DE LA PRÁCTICA

Linux

Realizar un programa utilizando pipes que utilice la salida de un comando del shell como la entrada de otro comando. Por ejemplo ps | wc.

Windows

Realizar un programa utilizando tuberías sin nombre en el cual ponga en práctica la lectura y escritura en una tubería. En el programa principal puede mandar a escribir los datos en la tubería y crear un hilo que se encargue de leerlos.

REFERENCIAS

http://linux.dsi.internet2.edu/docs/LuCaS/Manuales-LuCAS/DENTRO-NUCLEO-LINUX/dentro-nucleo-linux-html/dentro-nucleo-linux-5.html

http://www.dc.uba.ar/people/materias/so/datos/cap24.pdf

http://aula.linux.org.ar/docs/tecnicos/index/cap2.html

http://labsopa.dis.ulpgc.es/prog_c/IPC.HTM

http://docencia.ac.upc.es/FIB/ISO/ISO-lab-P6.pdf

http://www.tlm.unavarra.es/~daniel/docencia/arq/arq1998-1999/llamadas.pdf

http://bari.ufps.edu.co/personal/150802A/comunicacion.htm

http://winapi.conclase.net/

http://winapi.conclase.net/curso/index.php?000

SISTEMAS OPERATIVOS, Una visión aplicada, Jesús Carretero Pérez, Féliz García Carballeira, Pedro De Miguel Anasagasti, Fernando Pérez Costoya, Editorial Mc Graw Hill

PROTECCIÓN
GUIA DEL ESTUDIANTE

PROTECCIÓN
GUIA DEL ESTUDIANTE

OBJETIVOS

- ✓ Identificar cuáles son los mecanismos que permiten implementar la política de protección adecuada en los sistemas operativos Linux y Windows.

- ✓ Establecer y modificar los permisos de acceso al sistema de archivos UNIX.

- ✓ Optimizar el acceso al sistema de archivos UNIX mediante el uso de caracteres especiales.

MARCO TEÓRICO

LINUX

El sistema UNIX es un sistema multiusuario, el cual tiene la capacidad de que todos los usuarios conectados a la máquina puedan leer y usar archivos de otros usuarios, debido a esto, los archivos deben ser protegidos contra cualquier uso indebido.

Protección de cuentas

En Unix cada usuario tiene asignada una cuenta individual. Cuando un usuario se desea conectar al sistema debe usar el nombre de conexión (login) y la autenticación del nombre de conexión (password). La información de la conexión de usuarios es almacenada en Unix en el archivo **/etc/passwd**, el cual contiene una línea por cuenta con la siguiente información:

<usuario>:<password>:<UID>:<GID>:<comentario>:<directorio home>:<Shell>

Shadow password

Es un método para mejorar la seguridad del sistema trasladando los passwords encriptados (encontrados normalmente en /etc/passwd) a **/etc/shadow** el cual puede ser leído únicamente por usuarios privilegiados, en cambio que /etc/passwd puede ser leído por todos los usuarios del sistema. Este archivo tiene la siguiente estructura:

usuario:password:last:may:must:warn:expire:disable:reserved

Cambio de password

Todos los usuarios en UNIX tienen asignado un *password* o contraseña. El *password* de cada usuario se cambia con el comando ***passwd*.** Este comando pregunta por el nuevo *password* (dos veces, para asegurarse que tecleó el *password* correcto). El nuevo *password* surte efecto desde el momento en que se cambia. Si el usuario entra

al sistema de nuevo, se le preguntaría por este nuevo *password* para permitir el acceso.

Protección de archivos

El sistema operativo UNIX provee un mecanismo llamado permisos de archivos, que permite a los archivos ser propiedad de un determinado usuario.

Tipos de archivos

En UNIX la estructura de directorios se encuentra clasificada de acuerdo a los tipos de archivos que en el sistema se encuentren, existe un carácter que define el tipo de archivo.
Socket: **s.**
Enlace simbólico: **l.**
Directorio: **d.**
Dispositivo de caracteres: **c.**
Dispositivo de bloque: **b.**
Archivo corriente: **-.**

Permisos otorgados al dueño, grupo y cualquier otro usuario del sistema.

Cada archivo en UNIX tiene asociado una serie de permisos que le permiten ser accesados o no por los usuarios. Los permisos de archivos caen dentro de tres grupos:

- ✓ Lectura (r)
- ✓ Escritura (w)
- ✓ Ejecución (x)

Existen tres clases de usuarios para los cuales se les asignan cada uno de estos permisos:

- ✓ El propietario del archivo.
- ✓ El grupo al cual pertenece el archivo.
- ✓ Todos los usuarios, independientemente del grupo.

El comando ls con la opción -l muestra, entre otra información, los permisos de la siguiente manera:

-rwx r-x r--

- ✓ El primer carácter de la cadena de permisos ("-") representa el tipo de archivo.
- ✓ Las tres letras siguientes ("rwx"), representan los permisos otorgados al dueño del archivo.
- ✓ Los siguientes tres caracteres (“r-x”), representan los permisos del grupo en este archivo.
- ✓ Los últimos tres caracteres (“r--“), representan los permisos que tienen cualquier otro usuario en el sistema diferente al dueño del archivo o aquellos que pertenezcan al grupo.

Configuración de los permisos de archivos

Cuando se crea un archivo en UNIX, estos se crean sin tener en cuenta los derechos que se les da. Como primera medida de seguridad, es importante dar los permisos adecuados a cada archivo. Estos permisos pueden ser modificados sólo por el propietario del archivo o el superusuario (root). UNIX provee la utilidad **chmod** que nos permite alterar los permisos de un archivo. El comando **chmod** se usa de la siguiente forma:

- ✓ **chmod u+rwx programa.c** asigna los siguientes permisos al archivo:

  - rwx --- ---

- ✓ **chmod g+rw-x programa.c** asigna los siguientes permisos al archivo:

  - — rw- —

- ✓ **chmod o+r-wx programa.c** asigna los siguientes permisos al archivo:

  - --- — r--

- ✓ **chmod a+rw-x programa.c** asigna los siguientes permisos al archivo:

  - rw- rw- rw-

Cualquier permiso puede ser añadido o suprimido mediante los símbolos “+” (signo más para añadir permisos) ó “-”(signo menos para suprimir permisos).

Los simbolos **u,g,o,a** permiten especificar la categoría del usuario:

- ✓ “u” para especificar la categoría de dueño o propietario.
- ✓ “g” para especificar la categoría del grupo al que pertenece.

- ✓ “o” para especificar a todos los usuarios.
- ✓ “a” para colocar los permisos en las tres categorías

Este tipo de modificaciones también se pueden hacer con un número en octal que varía entre 000 y 777.

Por ejemplo:

chmod 743 nombrearchivo

Asigna los siguientes permisos:

- rwx r-- -wx

El propietario y el grupo de un archivo son otras de las propiedades de un archivo o directorio que Linux permite que sean modificadas. Mediante la orden ***chown*** un propietario de un archivo puede transferir los privilegios que tiene sobre éste a otro usuario del sistema.

chown nombrenuevousuario nombredelarchivo

También es posible cambiar el grupo de un archivo empleando la orden ***chgrp.***

chgrp nombrenuevogrupo nombrearchivo

Permisos especiales.

Existen dos tipos de permisos adicionales llamados permisos especiales que están disponibles para ejecutar archivos y directorios públicos. Cuando esos permisos son configurados, cualquier usuario puede ejecutar archivos, asumiendo los permisos del propietario para ejecutarlo.

Permiso setuid (set-user identification)

Cuando el **setuid** es configurado sobre un archivo ejecutable, cambia el **user id** de la persona que ejecuta el archivo por los del dueño del archivo. Esto permite a cualquier usuario ejecutar archivos y accesar a directorios que solo son accesados por el propietario, es decir, este permiso permite a un usuario normal ejecutar comandos a los cuales no tiene privilegios.

El permiso **setuid** se asigna a los archivos con el siguiente comando:

Chmod +s nombrearchivo

Permiso setgid (set-group identification)

El permiso **setgid** es similar a **setuid,** a excepción que el proceso identificación de grupo (GID), es cambiado al propietario del grupo del archivo, y un usuario tiene acceso basado sobre los permisos obtenidos para tal grupo. Cuando **setgid** es aplicado a un directorio, los archivos creados en ese directorio pertenecen al grupo que el directorio pertenezca, y no al grupo que pertenezca el proceso que lo creó. Cualquier usuario que tiene

permiso de escritura en el directorio puede crear archivos ahí, sin embargo el archivo no pertenecerá al grupo del usuario, pero si pertenecerá al grupo del directorio.

El permiso **setgid** se asigna a los archivos con el siguiente comando:

Chmod +g nombrearchivo

Variable umask

La instrucción umask permite a un usuario definir una máscara de protección por defecto, acepta como parámetro un valor numérico (máscara) que indica los permisos que se negaran a todos los archivos creados a partir de ese instante.

umask [permisos_numéricos]

Cuando se utiliza sin parámetros, devuelve el valor actual de la máscara.

El valor asignado por umask es restado del que viene por defecto, esto tiene el efecto de negar permisos en la misma forma en que **chmod** los otorga.

Por ejemplo:

umask 022

Todos los archivos creados a partir de ese momento tendrán los siguientes permisos

rwx r-x r-x

Ya que

7	7	7	-	0	2	2	=	7	5	5
rwx	rwx	rwx		---	-w-	-w-		rwx	r-x	r-x

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

WINDOWS NT

Servicios de Win32

Las llamadas al sistema de Win32 permiten manipular la descripción de los usuarios, los descriptores de seguridad y las ACL. A continuación se describen las llamadas de Win32 más frecuentemente usadas.

InitializeSecurityDescriptor: da valores iniciales a un descriptor de seguridad.
GetUserName: permite obtener el identificador de un usuario que ha accedido al sistema.
GetFileSecurity: extrae el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.
El servicio *SetFileSecurity* fija el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.
GetSecurityDescriptorOwner y *GetSecurityDescriptorGroup* permiten extraer la identificación del usuario de un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo. Estas llamadas son sólo de consulta y no modifican nada.
SetSecurityDescriptorOwner y SetSecurityDescriptorGroup permiten modificar la identificación del usuario en un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo.

Llamadas al sistema para la gestión de ACLs y ACEs

InitializeACL, AddAccessAllowedAce y *AddAccessDeniedAce:* permiten inicializar una ACL y añadir entradas de concesión y denegación de accesos.

Para mayor información consulte el trabajo de grado “Diseño y desarrollo de guías de laboratorio para el curso de sistemas operativos de la CUTB”.

DESARROLLO DE LA PRÁCTICA

El siguiente programa lee los atributos de seguridad de un archivo

```
#include <windows.h>
#include <stdio.h>

PSECURITY_DESCRIPTOR LeerPermisosDeUnArchivo (LPCTSTR
NombreArchivo)
    /*Devuelve los permisos de un archivo*/
{
    PSECURITY_DESCRIPTOR pSD = NULL;
    DWORD Longitud;
    HANDLE ProcHeap = GetProcessHeap();
    /*Obtiene el tamaño del descriptor de seguridad*/
    GetFileSecurity (NombreArchivo,
OWNER_SECURITY_INFORMATION |
            GROUP_SECURITY_INFORMATION |
DACL_SECURITY_INFORMATION,
            NULL, 0, &Longitud);
    /*Obtiene el descriptor de seguridad del archivo*/
    pSD = HeapAlloc (ProcHeap, HEAP_GENERATE_EXCEPTIONS,
Longitud);
    if (!GetFileSecurity (NombreArchivo,
OWNER_SECURITY_INFORMATION |
```

```
            GROUP_SECURITY_INFORMATION |
DACL_SECURITY_INFORMATION,
            pSD, Longitud, &Longitud))
      {
            printf ("Error GetFileSecurity \n");
            return NULL;
      }
      return pSD;
}

int main(int argc, char* argv[])
{
      for (int i = 1; i < argc; i++)
      {
            PSECURITY_DESCRIPTOR pSD =
            LeerPermisosDeUnArchivo(argv[i]);
            PSID pOwnerSID;
            BOOL bDefaulted;
            if (GetSecurityDescriptorOwner(pSD, &pOwnerSID,
            &bDefaulted))
            {
                  if (pOwnerSID)
                  {
                        SID_NAME_USE eUse;
                        TCHAR szName[100];
                        DWORD lNameLen = 100;
                        TCHAR szDomainName[100];
                        DWORD lDomainNameLen = 100;
```

```
                        if (LookupAccountSid(NULL, pOwnerSID,
                        szName, &lNameLen,
                                szDomainName, &lDomainNameLen,
        &eUse))
                        {
                                printf("%s:\t%s\\%s\n", argv[i],
                                szDomainName, szName);
                        }
                }
        }
        return 0;
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre proteccion.cpp o busque el archivo proteccion.cpp en el directorio ./Guia5/Windows.

3. Compile el archivo proteccion.cpp

4. Desde MS-DOS, escriba el nombre del programa y pásele como parámetro el nombre del archivo al cual le quiere averiguar los atributos de seguridad. Por ejemplo:

proteccion proteccion.cpp

ENUNCIADO DE LA PRÁCTICA

Linux

- ✓ Active el bit SETUID para que todos los usuarios puedan ver cualquier archivo del sistema utilizando el editor vi.
- ✓ ¿Qué tipos de accesos permite el fichero /etc/passwd? ¿Y /etc/shadow? ¿Quién es el propietario de estos archivos?
- ✓ Copie el archivo /etc/passwd a su directorio y asigne a su copia los siguientes derechos de acceso:

  El propietario puede leer, escribir y ejecutar el archivo.

  Todos los usuarios del grupo pueden leerlo y ejecutarlo, pero no escribir en él.

  El resto de usuarios solamente pueden ejecutarlo.

  Compruebe que el archivo tiene los permisos de acceso deseados.
- ✓ Utilice umask para que ningún otro usuario distinto del propietario pueda acceder a los archivos que se creen a partir de este momento. Compruebe que la máscara ha quedado actualizada.

Windows

Elabore un programa en el que utilizando las llamadas al sistema de Win32 que permiten manipular los descriptores de seguridad y las ACL,

cree un archivo el cual solo pueda ser accedido en modo lectura por un usuario que usted determine.

REFERENCIAS

http://shannon.unileon.es/~dielpa/Aero/Aero_Pract_3_sol.PDF

http://winapi.conclase.net/curso/index.php?000

http://www.sis.ucm.es/SIS/UNIX/indice.htm

http://www2.ing.puc.cl/~iic10622/clases/unix.htm

http://bernia.disca.upv.es/~iripoll/seguridad/03-admin/03-admin.pdf

http://www.iti.upv.es/~pgaldam/sso/transparencias/pdf/tema2.pdf

SISTEMAS OPERATIVOS, Una visión aplicada, Jesús Carretero Pérez, Féliz García Carballeira, Pedro De Miguel Anasagasti, Fernando Pérez Costoya, Editorial Mc Graw Hill.

ANEXOS B

GUÍA DEL DOCENTE

INSTALACIÓN DE LINUX MANDRAKE Y WINDOWS NT
GUÍA DEL DOCENTE

INSTALACIÓN DE LINUX MANDRAKE Y WINDOWS NT
GUÍA DEL DOCENTE

OBJETIVOS

- ✓ Instalar y configurar el sistema operativo Linux Mandrake y Windows NT para que los estudiantes puedan conocer las ventajas que estos brindan con respecto a los demás sistemas operativos.
- ✓ Promover la introducción del Sistema Operativo Linux mediante un conocimiento básico de sus posibilidades y alcance.
- ✓ Conocer los requerimientos tanto de hardware como de software necesarios para un funcionamiento adecuado del sistema operativo.

MARCO TEÓRICO

LINUX

Linux es un poderoso Sistema Operativo creado en 1991 por Linus Torvalds y actualmente es sostenido por el trabajo de varios grupos de programadores distribuidos en varias partes del mundo.

GNU/Linux es un sistema operativo gratuito y de libre distribución bajo las condiciones que establece la licencia GPL (GNU Public License). Tiene todas las características que uno puede esperar de un sistema Unix moderno: multitarea

real, memoria virtual, bibliotecas compartidas, carga por demanda, soporte de redes TCP/IP, entre muchas otras funcionalidades.

Características

- ✓ Multitarea: varios procesos se ejecutan al mismo tiempo.
- ✓ Multiusuario: varios usuarios utilizan la misma maquina al mismo tiempo.
- ✓ Multiplataforma: corre sobre diversas arquitecturas de CPU.
- ✓ Multiprocesador: ofrece soporte para plataformas con varios procesadores.
- ✓ Incluye protección de memoria entre procesos, de forma que un único programa no puede "colgar" el sistema al completo.
- ✓ Mejor aprovechamiento de la memoria, mediante la lectura desde el disco de aquellas partes de un programa que se están ejecutando, así como a través de la compartición de la misma zona de memoria para varios procesos simultáneos.
- ✓ Control completo de tareas y procesos.
- ✓ Soporte multinacional.
- ✓ Interacción sencilla y transparente con otros sistemas, como MS-DOS, Windows 9X/Me/XP, Windows NT/2000, OS/2, Novell y, por supuesto, otros UNIX.
- ✓ Soporte de red: TCP/IP, Novell, NT, Appletalk.
- ✓ Multitud de software disponible, tanto comercial como gratuito.
- ✓ Compatibilidad con todo tipo de hardware.
- ✓ Actualizaciones continuas tanto del núcleo como de "drivers" de soporte de nuevas tecnologías.
- ✓ Es más estable que otros sistemas operativos. Está respaldado por 30 años de experiencia y estabilidad de Unix, del cual deriva su arquitectura.

- ✓ Puede operar tanto en 32 como 64 bits reales según la plataforma usada (Intel, Alpha, etc.).
- ✓ Es altamente efectivo en redes cliente-servidor (intranet y extranets) y como servidor web.
- ✓ Posee varios entornos gráficos de ventanas semejantes a Windows o Macintosh.
- ✓ Lee perfectamente otros sistemas de archivos como FAT16 y 32, NTFS, HPFS, ISO9660, Joliet, etc.

¿Para que sirve?

- ✓ Estación de Trabajo Personal: Todas sus características la convierten en una maquina UNIX poderosa.
- ✓ Plataforma de Desarrollo Unix: Linux posee compiladores para la gran mayoría de lenguajes existentes como C, C++, F77, Java, ADA, Modula 2 y 3, Pascal, Tcl/Tk, Scheme, Smalltalk, etc, sin costo alguno. Además de librerías gráficas, de computación paralela y distribuida, y librerías científica para múltiples aplicaciones.
- ✓ Plataforma de Desarrollo Comercial: Bases de datos Clipper (dBase y Fox), Oracle e Informix.
- ✓ Servidor Internet: WWW, mail, ftp, news, gopher y todos los posibles servicios a través de la red. Combinado con conexiones remotas se convierte en un excelente proveedor remoto de Internet.
- ✓ Servidor de Terminales. FAX, MODEM, Líneas telefónicas, y equipos seriales, para proveer acceso directo o remoto a recursos locales.

- ✓ Servidor de comunicaciones: puede trabajar como gateway, firewall, proxy o servidor de módems, entre otros, ofreciendo además otros servicios populares como caché WWW o autentificación de usuarios.
- ✓ Servidor de Intranet: ficheros, web, correo electrónico, DNS, gestión de bases de datos, etc.

WINDOWS NT

Windows NT se trata de un sistema operativo de red de multitarea preferente, de 32 bits con alta seguridad y servicios de red.

Características

Fiabilidad

Una fiabilidad superior permite a Windows NT ser usado como base para aplicaciones críticas. Está especialmente indicado para estaciones de trabajo y servidores de red, los cuales necesitan el máximo rendimiento.

Rendimiento

Windows NT fue también diseñado para ser un sistema operativo de alto rendimiento. Características que contribuyen a esto son:

- ✓ Diseño real de 32 bits. Todo el código de Windows NT en 32 bits, lo que le proporciona mucha más velocidad que otros operativos escritos con tecnología de 16 bits.
- ✓ Características de multitarea y multiproceso. Windows NT proporciona multitarea preferente, lo que permite una ejecución de todos los procesos, y además soporta varias CPU lo que es rendimiento.

- ✓ Soporta de CPU RISC. Windows NT no sólo soporta CPU basadas en INTEL, sino en diferentes tipos de CPU como Poder PC, DEC Palpa y MAC.

Portabilidad

La portabilidad en Windows NT significa que este sistema operativo puede usarse con diferentes tipos de hardware sin necesidad de reescribir el código completamente de nuevo. Windows NT proporciona las siguientes características en portabilidad:

- ✓ Arquitectura de micro-kernel modular. Windows NT posee un diseño modular lo que le proporciona independencia del hardware.
- ✓ Sistemas de archivos configurables. Otra de las características de Windows NT que aumenta sus posibilidades de portabilidad es su capacidad de soportar diferentes sistemas de archivos. Actualmente soporta FAT usados en sistemas DOS, HPFS usados en sistemas OS/2, NTF sistema de archivos de NT, CDFS sistema de archivo de CD-ROM y sistemas de archivos Macintosh.

Compatibilidad

Es un elemento clave para la aceptación de un sistema operativo, es una capacidad de trabajar con las aplicaciones ya existentes. Microsoft ha diseñado Windows NT para que sea capaz de ejecutar una amplia variedad de diferentes aplicaciones e interactúe con diferentes sistemas operativos.

- ✓ **Diseño de aplicaciones como subsistemas.** Windows NT soporta aplicaciones MS-DOS, Windows 3.x (16 bits), Windows 95, POSIX y OS/2 1.x. Otra vez, el

diseñado modular de Windows NT lo hace posible, simplemente añadiendo nuevos subsistemas.

- ✓ **Interfaz explorador de Windows 95.** Se ha mantenido el interfaz gráfico tan esperado en Windows 95. Es realmente un replica exacta a no ser por algunas carpetas que faltan y otras que se han añadido. De hecho cuando el usuario migra Windows 95 a Windows NT, puede optar por mantener el escritorio original.
- ✓ **Interoperatividad con UNIX.** Windows NT se comunica con sistemas UNIX a través del protocolo TCP/IP. También soporta impresión TCP/IP e incluye aplicaciones de conectividad básicas como por ejemplo FTP, Telnet y Ping.

Escalabilidad

Otro aspecto importante de Windows NT es que es un sistema operativo escalable. Esto quiere decir que puede ser usado por un amplio abanico de sistemas, desde un ordenador personal hasta grandes sistemas con múltiples procesadores. Estos sistemas pueden tener muy pocas cosas en común o casi ninguna. Una pequeña lista de las características de escalabilidad son estas:

- ✓ **Soporte multiplataforma**. Debido a que la arquitectura de micro-kernel está en capas de abstracción de hardware (HALL), Windows NT es capaz de soportar los más potentes procesadores desarrollados en el futuro.
- ✓ **Soporte multiprocesador.** Soporta múltiples CPU en sistemas, lo que le proporciona un funcionamiento más eficiente a medida que se aumentan los procesadores.

Seguridad

Otras de las características más importantes de Windows NT son sus características de seguridad.

- ✓ **Modelo de seguridad de dominio.** El modelo de seguridad de dominio es un sofisticado sistema de acceso a la red, de manera que se controla perfectamente por donde los recursos de red que un usuario puede utilizar. Unos servidores especiales llamados controladores de dominio son los encargados de realizar todo el trabajo de autentificación de usuarios. La información de seguridad se guarda en una base de datos llamada SAM (Security Account Manager).
- ✓ **Sistema de archivos NTFS.** Es un sistema de archivos propio de Windows NT, que complementa la seguridad del sistema. Permite a los administradores de la red el control de utilizar una variedad de acceso a la red para grupos o usuarios.
- ✓ **Características de tolerancia a fallos.** Windows NT incluye importantes características de tolerancia a fallos. La primera característica importante es el soporte RAID (Redundant Array Of Inexpensive Disk), la cual usa una tecnología parecida al disk mirroring. Si se produce un fallo en el disco, gracias al RAID la información se puede obtener de nuevo. Otra característica importante de tolerancia de fallos es el soporte de UPS, unidades de alimentación interrumpida. Windows NT detectaría una caída de tensión en la red y conmutaría inmediatamente a la UPS.

INSTALACIÓN DE LINUX MANDRAKE

Antes de instalar

La máquina debe arrancar leyendo la unidad CD, si no es así se debe configurar la BIOS de la siguiente manera:

Lo primero es acceder a la BIOS, para ello durante el arranque de la misma muestra un mensaje que nos indica que pulsemos la tecla "Supr" para acceder a la BIOS. Una vez dentro de la BIOS iremos a la sección "BIOS FEATURES SETUP" y en ella modificaremos el valor de "Boot Sequence" a CDROM.

Comenzando la instalación

Para comenzar la instalación se introduce el CD #1 en la unidad lectora y se reinicia el ordenador. DrakX es el programa de instalación de Mandrake Linux. Tiene una interfaz de usuario gráfica la cual permite volver a las opciones de configuración previas en cualquier momento. Aparece una pantalla de bienvenida y luego aparece la pantalla de instalación donde en la parte izquierda se puede visualizar las diferentes fases de instalación y en el marco inferior una descripción detallada de cada una de ellas.

Eligiendo el idioma

El primer paso es elegir el idioma de instalación y uso del sistema. Al hacer clic sobre el botón Avanzado podrá seleccionar otros idiomas para instalar en su estación de trabajo.

Términos de licencia de la distribución

Antes de continuar, debería leer cuidadosamente los términos de la licencia. Los mismos cubren a toda la distribución Mandrake Linux, y si Usted no está de acuerdo con todos los términos en la misma, haga clic sobre el botón Rechazar, esto terminará la instalación inmediatamente.

Para continuar con la instalación, haga clic sobre el botón Aceptar.

Clase de instalación

DrakX necesita saber si desea realizar una instalación por defecto (Recomendada) o si desea tener un control mayor (Experto).

Recomendada: elija esta si nunca ha instalado un sistema operativo GNU/Linux. La instalación será fácil y sólo se le formularán unas pocas preguntas.

Experto: si tiene un conocimiento bueno de GNU/Linux, puede elegir esta clase de instalación. La instalación experta le permitirá realizar una instalación altamente personalizada.

Configuración del ratón

DrakX generalmente detecta la cantidad de botones que tiene su ratón. Si no, asume que Usted tiene un ratón de dos botones y lo configurará para que emule el tercer botón. DrakX sabrá automáticamente si es un ratón PS/2, serie o USB.

Si desea especificar un tipo de ratón diferente, seleccione el tipo apropiado de la lista que se proporciona. Si elige un ratón distinto al predeterminado, se le presentará una pantalla de prueba. Use los botones y la rueda para verificar que la configuración es correcta. Si el ratón no está funcionando correctamente, presione la barra espaciadora o Intro para Cancelar y vuelva a elegir.

Configuración del teclado

Normalmente, DrakX selecciona el teclado adecuado para Usted (dependiendo del idioma que eligió). Sin embargo, podría no tener un teclado que se corresponde exactamente con su idioma. Haga clic sobre el botón Más para que se le presente la lista completa de los teclados soportados.

Selección de los puntos de montaje

Ahora necesita elegir el lugar de su disco rígido donde se instalará su sistema operativo Mandrake Linux.

Presione sobre la partición de Linux y luego en Crear y aparece una ventana que solicita el tamaño en MB, el tipo de sistema de archivos y el punto de montaje.

Ahora usted puede crear todas las particiones que necesite y después formatearlas para luego usarlas.

Elección de los paquetes a instalar

Ahora debe especificar los programas que desea instalar en su sistema.

Los paquetes se ordenan en grupos que corresponden a un uso particular de su máquina. Los grupos en sí mismos están clasificados en cuatro secciones:

1. Estación de trabajo: si planea utilizar su máquina como una estación de trabajo, seleccione uno o más grupos correspondientes.
2. Desarrollo: si la máquina se utilizará para programación, elija el(los) grupo(s) deseado(s).
3. Servidor: finalmente, si se pretende usar la máquina como un servidor aquí puede seleccionar los servicios más comunes que desea que se instalen en la misma.
4. Entorno gráfico: seleccione aquí su entorno gráfico preferido.

Si mueve el cursor del ratón sobre el nombre de un grupo se mostrará un pequeño texto explicativo acerca de ese grupo.

Finalmente, dependiendo de si Usted elige seleccionar los paquetes individuales o no, se le presentará un árbol que contiene todos los paquetes clasificados por grupos y sub-grupos. Mientras navega por el árbol, puede seleccionar grupos enteros, sub-grupos, o simplemente paquetes.

Tan pronto como selecciona un paquete en el árbol, aparece una descripción del mismo sobre la derecha. Cuando ha finalizado con su selección, haga clic sobre el botón Instalar que lanzará el proceso de instalación.

Dependiendo de la velocidad de su hardware y de la cantidad de paquetes que se deben instalar, el proceso puede tardar un rato en completarse. En la pantalla se muestra una estimación del tiempo necesario para completar la instalación.

La instalación Mandrake Linux se divide en varios CD-ROMs. DrakX sabe si un paquete seleccionado se encuentra en otro CD y expulsará el CD corriente y le pedirá que inserte uno diferente cuando sea necesario.

Contraseña de Root

Este es el punto de decisión más crucial para la seguridad de su sistema GNU/Linux, tendrá que ingresar la contraseña de root. Root es el administrador del sistema y es el único autorizado a hacer actualizaciones, agregar usuarios, cambiar la configuración general del sistema, etc. Deberá elegir una contraseña que sea difícil de adivinar – DrakX le dirá si la que eligió es demasiado fácil. La contraseña debería ser una mezcla de caracteres alfanuméricos y tener al menos una longitud de 8 caracteres, recuerde que linux es sensible a las mayúsculas y a las minúsculas. Sin embargo, por favor no haga la contraseña muy larga o complicada debido a que Usted debe poder recordarla sin realizar mucho esfuerzo.

Agregar un usuario

GNU/Linux es un sistema multiusuario, y esto significa que cada usuario puede tener sus preferencias propias, sus archivos propios, y así sucesivamente. Pero, a diferencia de root, que es el administrador, los usuarios que agregue aquí no podrán cambiar nada excepto su configuración y sus archivos propios. Tendrá que crear al menos un usuario no privilegiado para Usted mismo. Primero tendrá que ingresar el nombre de usuario, el cual usará para ingresar al sistema. Luego tendrá que ingresar una contraseña. Si hace clic sobre Aceptar usuario, entonces puede agregar tantos como desee. Cuando haya terminado de agregar todos los usuarios que desee, seleccione Hecho.

Instalación del LILO

Lilo (Linux Loader) es un excelente gestor de arranque que nos permite seleccionar el sistema operativo con el que iniciar cada vez la máquina. El lugar habitual para instalarlo es el MBR (Master Boot Record). Este es necesario aún cuando Linux sea el único sistema en el computador. A continuación la instalación preguntará donde se desea poner el LILO, para ponerlo en el MBR seleccionar /dev/hda. No seleccione /dev/hda1 porque seguramente destruirá el sistema de archivos de Windows/DOS.

La instalación está completa y su sistema GNU/Linux está listo para ser utilizado. Simplemente haga clic sobre Aceptar para volver a arrancar el sistema. Puede iniciar GNU/Linux o Windows, cualquiera que prefiera (si está usando el arranque dual) tan pronto como su máquina haya vuelto a arrancar.

INSTALACIÓN DE WINDOWS NT

Windows NT se suministra con un entorno de instalación amigable y fácil de usar, detectando e instalado casi todo lo que se refiere a hardware él sólo.

Cuando se empieza una instalación lo primero que hay que hacer es calcular la plataforma que nos hace falta para instalar el sistema operativo para que su rendimiento sea eficaz y rápido.

Es importante tener en cuenta las siguientes recomendaciones:

Conviene que el sistema operativo y todos sus programas asociados estén en el mismo disco duro y si puede ser en la partición de arranque.

Que en esta partición se tenga todo el software de administración y soporte de hardware del sistema.

Cuando termine la instalación se debe hacer una copia de seguridad de este sistema de archivos en una unidad de backup.

Inicio de la instalación de Windows NT

La primera parte de la instalación de NT basada en modo texto, es idéntica para el servidor y la workstation.

El sistema operativo se suministra en formato CD, por lo tanto la plataforma tiene que tener instalada un lector de CD-ROM compatible con NT o estar conectado a una red que tenga uno compartido. Además, usted debe tener tres discos de inicio, si no es así debe generarlos. Los discos de inicio pueden generarse desde cualquier PC que tenga CD-ROM, basta con introducir el CD de NT, ir al directorio I386 y ejecutar la instrucción WINNT /OX, le pedirá tres diskettes formateados y vacíos.

Introduzca el CD-ROM en su lector, el disco 1 en la disquetera y después encienda el sistema.

Si está instalando NT en un equipo que pueda arrancar desde el lector de CDROM, puede cambiar en la BIOS la secuencia de arranque de manera que empiece por el CD, aunque el disco duro esté sin formatear y consecuentemente sin ningún tipo de sistema operativo instalado, el programa de instalación se inicia sólo y permite hacer la instalación sin tener nada en el disco duro y sin tener que generar los diskettes de instalación.

Reconocimiento del sistema

Lo primero que hace es reconocer el hardware indispensable para empezar a trabajar y comprobar que no exista una versión de NT, en este caso se detendrá la instalación y tendrá que realizarla desde ese sistema NT ya instalado (usando WINNT32) o eliminar la partición donde estuviera ubicado. A continuación comenzará la carga de los archivos necesarios para la instalación y le pedirá que introduzca el disco 2 o en el caso de estar haciendo una instalación sin discos pasará a un menú donde pregunta:

Si queremos ayuda sobre la instalación (F1)

Si queremos instalar NT (ENTRAR)

Si queremos reparar (R), este apartado lo veremos en un próximo documento.

Si queremos salir de la instalación (F3)

Debe pulsar “ENTRAR”

Configuración de unidades de almacenamiento

Pasará a la fase de detección de los controladores ESDI/IDE, SCSI y unidades de CDROM conectadas, preguntándole si quiere detectar controladoras SCSI (ENTRAR) o no detectarlas “I” ; éste sería el caso si no tuviera ningún dispositivo SCSI. Debe pulsar “ENTRAR”. Le pedirá el disco 3, aparece una pantalla con el resultado de la detección. Si no hubiera sido detectado alguno de sus discos duros o lectores de CDROM, tendría que instalar el driver del fabricante presionando “S”. Si los hubiera detectado todos pulse “ENTRAR”. Aparece en pantalla la licencia del producto la cual debe leer atentamente dando al avance página hasta que le permita dar “F8” para continuar, siempre que esté de acuerdo con las condiciones de la licencia. Seguidamente le dará un listado de componentes instalados en el sistema, los cuales podrá cambiar en caso necesario. Ahora pasa al gestor de particiones de disco y de ubicación de la instalación el cual pregunta: ¿Dónde quiere instalar NT? .Para ello debe moverse con el cursor hasta la partición donde quiera instalarlo y luego presione “ENTRAR”.

Si tiene espacio sin asignar muévase con el cursor a ese espacio no particionado y pulsando la tecla “C” cree una nueva partición. Lo más importante es tener un espacio de aproximadamente 300 Mb para la instalación de NT.

Si quiere borrar una partición mueva el cursor a la partición existente y pulse “E”.

En su caso tendrá una partición FAT con el tamaño necesario para la instalación del NT, por lo que debe mover el cursor hasta situarlo encima de dicha partición y pulsar “ENTRAR”. Pasará a preguntarle si quiere convertir la partición a NTFS o dejarlo como está, con el cursor se moverá a la opción que desee. La instalación es más rápida sobre FAT pero recuerde que cuando termine la instalación tendrá que ejecutar CONVERT C: /FS:NTFS para convertir a NTFS, siempre que quiera convertir el sistema de archivo a este tipo.

Nota: NTFS le permite configurar permisos de seguridad sobre archivos y directorios; FAT es más rápido pero no tiene opciones de seguridad.

También le preguntará el directorio donde quiere ubicar el bloque de programas del NT, por defecto "\WINNT" y pasará a examinar los discos para comprobar su integridad, para ello pulse “ENTRAR”. Si considera que los discos están en perfecto estado pulse “ESC”.

Llegado a este punto el sistema se pondrá a copiar los archivos necesarios para la instalación del sistema NT, cuando acabe este proceso retire el disco de la disquetera y del CD-ROM y presione “ENTRAR”

Una vez pasada la primera parte de la instalación, se reinicia el ordenador y comienza la instalación basada en entorno gráfico.

Le saldrá una pantalla donde se indican los pasos que va a seguir la instalación, donde debe pulsar “SIGUIENTE”, y pasará a otra donde indicará el tipo de instalación que va a realizar:

Típica: Recomendada para la mayoría de los equipos
Portátil: Se instalará con opciones útiles para equipos portátiles
Compacta: Para ahorrar espacio en disco, no se instalará ninguno de los componentes opcionales
Personalizada: Para usuarios más avanzados. Puede personalizar todas las opciones de instalación disponibles

Seleccione la personalizada y pulse “SIGUIENTE”.

En el paso siguiente coloque el nombre y la organización a la que va a pertenecer la licencia, “SIGUIENTE”, e introduzca la clave del CD de NT, la cual viene en la carátula del CD, “SIGUIENTE”, pasará a poner el nombre que va ha tener el equipo para su reconocimiento en red, “SIGUIENTE”, y le preguntará la contraseña del administrador, “SIGUIENTE”, le pregunta si queremos un disco de rescate. El disco de rescate es importante por si existe un bloqueo o un fallo en el arranque de NT, este disco se tendrá que acuatizar cada cierto tiempo, y siempre antes de hacer un cambio en el equipo, sobre todo si es un cambio de hardware. En la pantalla siguiente debe seleccionar los componentes que desea instalar y pulsar “SIGUIENTE”.

Configurando el acceso a red

Si el equipo está conectado a una red a través de RDSI (ISDN) o un adaptador de red debe pulsar como activo en el cuadro a tal efecto. Si a su vez va ha tener control de acceso telefónico a redes también debe marcar el cuadro a tal efecto.

Si el equipo no va ha tener nada de lo anterior pulse el botón redondo que le indica tal opción (No conectar este equipo a una red en este momento).

Ahora pasará a la instalación de los protocolos con los que van a trabajar nuestro sistema, los cuales pueden ser TCP/IP, IPS/SPX, NetBEUI, pudiéndose seleccionar otros desde una lista o instalarlos desde un disco del fabricante.

Pantalla de los servicios de red

Sale un listado con los servicios mínimos de red que no se pueden tocar desde la instalación, en el caso que quiera quitar algunos tendrá que esperar a que se acabe la instalación. Ya habrá acabado la instalación de red pulse “SIGUIENTE”, si tiene alguna duda pulse “ATRÁS”. Ahora seguirá con la introducción de los datos del TCP/IP de su equipo, si tiene una dirección fija de red la puede colocar una vez activada la casilla a para tal efecto, con la mascara de red adecuada, si no tiene ningún ROUTER o GATEWAY para la solución de encaminamiento lo puede dejar en blanco, en caso de que existiera pondría la dirección de este. Ahora llega el momento de decirle si vamos a formar parte de un dominio NT o en un grupo de trabajo en el caso de trabajar en un dominio necesitamos la asistencia del administrador para que de alta la máquina. Pulse “SIGUIENTE”. Copiará el resto

de los archivos, guardará la configuración y le pedirá que inserte un disco que etiquetará como “Disco de reparación” y pulse “ACEPTAR”, borrará los archivos temporales y le pedirá que reinicie.

REFERENCIAS

http://www.mandrakelinux.com/es/

http://www.monografias.com/trabajos6/linux/linux.shtml

http://enete.us.es/docu_enete/nt4/indice.asp

http://www.iespana.es/cpys/WinNT/3.pdf

http://support.ap.dell.com/docs/storage/73vexfms/sp/inswinnt.htm

http://temu.tco.plaza.cl/mg_tec/documentos/winnt/instalacion_nt.html

http://www.latiendadelfuturo.com/ANTAD/Html/Versatil/ManPOS/1.1InstalacionWindowsNT.pdf

CONTROL Y ADMINISTRACIÓN DE PROCESOS

GUIA DEL DOCENTE

CONTROL Y ADMINISTRACIÓN DE PROCESOS

GUIA DEL DOCENTE

OBJETIVOS

- ✓ Conocer las llamadas a sistema de los sistemas operativos Unix y Windows para utilizarlas en la generación de aplicaciones.

- ✓ Identificar los parámetros que se deben tener en cuenta para el control y administración de procesos.

MARCO TEÓRICO

Un proceso es un programa en ejecución que tiene asociado código, datos, stack, registros e identificador único y requiere recursos (memoria, CPU, dispositivos de E/S, stack) para funcionar. A continuación describiremos las funciones principales para el control y administración de procesos en los sistemas operativos Linux y Windows NT.

LINUX

En Linux a cada proceso se le asigna un número de identificación o PID (Process ID) que cambia en cada ejecución del mismo, pero que permanece invariable mientras el programa se esté ejecutando; este número es asignado automáticamente por el sistema en tiempo de ejecución.

Estados de un proceso

Los estados básicos en los que puede estar un proceso son los siguientes:

- ✓ Creado (created)
- ✓ Listo (ready)
- ✓ En ejecución (running)
- ✓ Suspendido (waiting)
- ✓ Destruido (terminated)

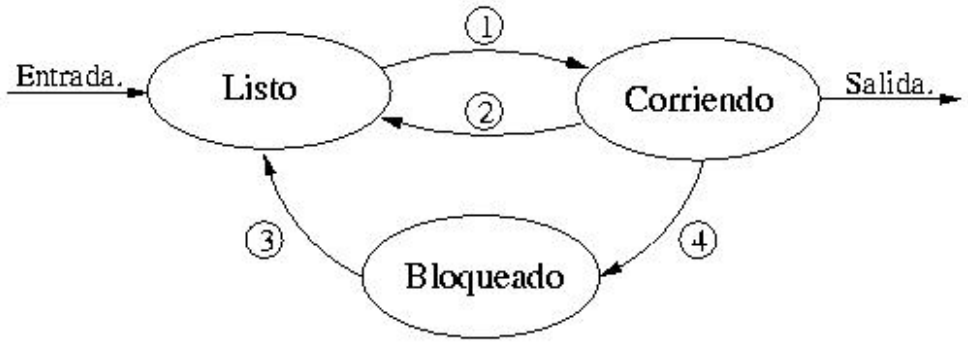


Figura 1. Diagrama de estados de un proceso

Cada proceso bajo Linux es dinámicamente asignado a una estructura struct task_struct. El número máximo de procesos que pueden ser creados bajo Linux está solamente limitado por la cantidad de memoria física presente.

Estados para los procesos en Linux:

- ✓ TASK_RUNNING: el proceso se encuentra estado activo, es decir se está ejecutando o cuando está listo para ejecutarse.
- ✓ TASK_INTERRUPTIBLE: Proceso "dormido" que puede despertar por alguna señal o despertador.
- ✓ TASK_UNINTERRUPTIBLE: El proceso está esperando por algún recurso hardware.
- ✓ TASK_ZOMBIE: Proceso hijo terminado pero que no ha sido liberado por su proceso padre.
- ✓ TASK_STOPPED: El proceso ha sido detenido momentáneamente.
- ✓ TASK_EXCLUSIVE: Estado complementario de UN/INTERRUPTIBLE, pero que requiere ser despertado sólo y no en grupo.

Llamadas al sistema para la administración de procesos

Crear Procesos

fork( ): Cuando un proceso usa la llamada a sistema fork() se crea en memoria una copia exacta del mismo y las estructuras de tarea son esencialmente idénticas. El proceso creador se denomina proceso-padre y el proceso creado proceso-hijo, por lo tanto cada proceso tiene un único proceso padre y cero o varios procesos hijos.

El proceso padre se diferencia del proceso hijo por los valores PID (*Process ID)* y PPID (*Parent Process ID*) y por el valor de retorno de la llamada fork(), que será -1 en caso de fallo, pero si tiene éxito al proceso hijo retornará el valor 0 y al proceso padre retornará el PID del proceso hijo.

vfork( ): Esta llamada permite crear procesos sin necesidad de copiar todo el espacio de direcciones del padre.

Proceso en espera

wait( ) : La llamada al sistema wait detiene al proceso que la invoca hasta que un hijo de éste termine. Si wait regresa debido a la terminación de un hijo, el valor devuelto es igual al PID del proceso que finaliza, de lo contrario devuelve -1.

waitpid( ): La llamada waitpid proporciona un método para esperar por un hijo en particular. Si pid es igual a -1, waitpid espera por cualquier proceso. Si pid es positivo entonces espera al hijo cuyo PID es pid.

Ejecutar código de un proceso

El servicio **exec( )** de POSIX tiene por objetivo cambiar el programa que está ejecutando un proceso. Existe una familia de funciones exec: ***excl., execv, execle, execve, execlp, execvp.***

Finalizar un proceso

El proceso ejecuta su última instrucción y le pide al sistema operativo que lo borre, o ejecuta la llamada al sistema **exit( ).** En ambos casos sistema operativo libera los recursos utilizados por el.

Identificación de procesos

getpid( ): Obtiene el identificador del proceso.
getppid( ): Obtiene el identificador del proceso padre.

Comandos para la administración de procesos

ps (process status): muestra los procesos que se están ejecutando y que fueron arrancados por el usuario actual**.** Los campos de información más importantes desplegados por ps para cada proceso son: Usuario (USER), identificadores de proceso (PID, PPID), Uso de recursos reciente y acumulado (%CPU, %MEM, TIME), Estado del proceso (STAT, S) y comando invocado (COMMAND).
ps –fea despliega todos los procesos activos.
ps -aux : Despliega todos los procesos del sistema, con nombre y tiempo de inicio.
pstree: en algunos sistemas está disponible el comando pstree, que lista los procesos y sus descendientes en forma de árbol. Esto permite visualizar rápidamente los procesos que están corriendo en el sistema.
top: esta herramienta monitorea varios recursos del sistema y tiene un carácter dinámico, muestra uso de CPU por proceso, cantidad de memoria, tiempo desde su inicio, etc.

nice: asigna la prioridad a los procesos, los valores nice oscilan desde -20 (mas prioridad) a 20 (menos prioridad), normalmente este valor se hereda del proceso padre.

kill: comando que se utiliza para eliminar un proceso.

kill -KILL <pid> : señala al proceso con numero <pid>,que termine de inmediato, el proceso es terminado abruptamente.

kill -TERM <pid> : señala al proceso con numero <pid>,que debe de terminar, a diferencia de -KILL , esta opción da la oportunidad al proceso de terminar.

kill -STOP <pid> : señala al proceso con numero <pid>, que pare momentáneamente.

kill -CONT <pid> : señala al proceso con número <pid>, que continué, este comando se utiliza para reanudar un proceso que le fue aplicado –STOP.

kill -INT <pid> : interrumpe al proceso con numero <pid>

bg: ordena a un proceso que está parado en segundo plano, que continúe ejecutándose en segundo plano.

fg: ordena a un proceso que está en segundo plano (parado o en funcionamiento) que vuelva al primer plano.

jobs: se usa para comprobar cuantos procesos se están ejecutando en segundo plano.

&: se usa en la línea de comando y con él le indicamos al sistema que la línea que se va a ejecutar deberá de ser puesta a trabajar en segundo plano. El "&" se pone al final de la línea a ejecutar.

Ctrl+Z: Envía al segundo plano el programa que se esté ejecutando y lo detiene.

vmstat: el comando vmstat reporta varias estadísticas que mantiene el kernel sobre los procesos, la memoria y otros recursos del sistema. Alguna de la información reportada por vmstat es la siguiente:

- ✓ Cantidad de procesos en diferentes estados (listo para correr, bloqueado, "swapeado" a disco.
- ✓ Valores totales de memoria asignada a procesos y libre.
- ✓ Estadísticas sobre paginado de memoria (page faults, paginas llevadas o traídas a disco, páginas liberadas)
- ✓ Operaciones de disco
- ✓ Uso de CPU reciente clasificado en inactivo, ejecutando en modo usuario y ejecutando en modo kernel

Servicios POSIX para la planificación de procesos

Modificar los parámetros de planificación

sched_setparam( ): Modifica la prioridad de un proceso.

sched_scheduler( ): Modifica la prioridad y política de planificación de un proceso.

Obtener los parámetros de planificación

sched_getparam( ): Devuelve la prioridad del proceso.

sched_getscheduler( ): Devuelve la política de planificación del proceso.

Hilos (Threads)

Un thread, también llamado proceso liviano (lightweight process LWP) es una unidad básica de utilización de CPU y consiste de una secuencia de instrucciones ejecutadas en un programa, un pc, un conjunto de registros y un espacio de stack. Todos los threads corren independientemente uno de otro en el mismo espacio de direcciones y no son visibles fuera del proceso. Los threads pueden crearse a nivel

de usuario y a nivel de kernel. Los threads a nivel de kernel son administrados independientemente por el sistema de operación en cuanto a la asignación de CPU (scheduling), mientras el despacho a CPU entre threads a nivel de usuario no requieren llamadas al sistema de operación (no producen una interrupción) para realizar el cambio de contexto, por lo tanto es muy rápido. Los threads a nivel de kernel se bloquean y despiertan en forma independiente de los demás. Mientras que si un thread a nivel de usuario es bloqueado, causará que todo el proceso se bloquee.

Llamadas al sistema para la administración de hilos

pthread_create(): Crea un nuevo hilo de ejecución, indicando los atributos del hilo.

pthread_equal(): Compara si dos identificadores de hilo son el mismo.

pthread_exit(): Finaliza el hilo que realiza la llamada.

pthread_join(): Espera la terminación de un hilo específico.

pthread_self(): Devuelve el identificador del hilo que realiza la llamada.

pthread_getschedparam(): Obtiene la política de planificación y los parámetros del hilo especificado.

pthread_setschedparam(): Establece la política de planificación y los parámetros del hilo especificado.

pthread_attr_init(): Permite iniciar un objeto atributo que se puede utilizar para crear nuevos hilos.

pthread_attr_destroy(): Destruye el objeto de tipo atributo.

Servicios POSIX para la planificación de hilos

Modificar los parámetros de planificación

pthread_setschedparam( ): Modifica la prioridad y política de planificación.

Obtener los parámetros de planificación

pthread_getschedparam( ): Devuelve la prioridad del proceso.

WINDOWS NT

Llamadas al sistema para la administración de procesos

Creación de un Proceso

En Win32 los procesos se crean mediante la llamada CreateProcess. El prototipo de esta función es el siguiente:

```
BOOL CreateProcess (
                    LPCTSTR lpszImageName,
                    LPTSTR lpszCommandLine,
                    LPSECURITY_ATTRIBUTES lpsaProcess,
                    LPSECURITY_ATTRIBUTES lpsaThread,
                    BOOL fInheritHandles,
                    DWORD fdwCreate,
                    LPVOID lpvEnvironment,
                    LPCTSTR lpszCurdir,
                    LPSTARTUPINFO lpsiStartInfo,
                    LPPROCESS_INFORMATION lppiProcInfo);
```

Esta función crea un nuevo proceso y su proceso ligero principal. El nuevo proceso ejecuta el archivo ejecutable especificado en lpszImageName. Esta cadena puede especificar el nombre de un archivo con camino absoluto o relativo, pero la función no utilizará el camino de búsqueda. Si lpszImageName es NULL, se utilizará como nombre de archivo ejecutable la primera cadena delimitada por blancos del argumento lpszCommandLine.

El argumento lpszCommandLine especifica la línea de comandos a ejecutar, incluyendo el nombre del programa a ejecutar. Si su valor es NULL, la función utilizará la cadena apuntada por lpszImageName como línea de mandatos.

El argumento lpsaProcess determina si el manejador asociado al proceso creado y devuelto por la función puede ser heredado por otros procesos hijos. Si es NULL, el manejador no puede heredarse. Lo mismo se aplica al manejador lpsaThread, pero relativo al manejador del proceso ligero principal devuelto por la función.

El argumento fInheritHandles indica si el Nuevo proceso hereda los manejadores que mantiene el proceso que realiza la llamada. El argumento fdwCreate puede combinar varios valores que determinan la prioridad y la creación del nuevo proceso. Algunos de estos valores son:

- ✓ CREATE_SUSPEND: el proceso ligero principal del proceso se crea en estado suspendido y sólo se ejecutará cuando se llame a la función ResumeThread.
- ✓ DETACHED_PROCESS: para procesos con consola, indica que el nuevo proceso no tenga acceso a la consola del proceso padre.
- ✓ CREATE_NEW_CONSOLE: el nuevo proceso tendrá una nueva consola asociada y no heredará la del padre. Este valor no puede utilizarse con el anterior.
- ✓ NORMAL_PRIORITY_CLASS: indica un proceso sin necesidades especiales de planificación.
- ✓ HIGH_PRIORITY_CLASS: indica que el proceso se cree con una prioridad alta de planificación.

- ✓ IDLE_PRIORITY_CLASS: especifica que los procesos ligeros del proceso sólo ejecuten cuando no haya ningún otro proceso ejecutando en el sistema.
- ✓ REALTIME_PRIORITY_CLASS: indica un proceso con la mayor prioridad posible.

El parámetro lpvEnvironment apunta al bloque del entorno del nuevo proceso. Si el valor es NULL, el nuevo proceso obtiene el entorno del proceso que realiza la llamada.

El argumento lpszCurdir apunta a una cadena de caracteres que indica el directorio actual de trabajo para el nuevo proceso.

El parámetro lpStartupInfo apunta a una estructura de tipo STARTPINFO que especifica la apariencia de la ventana asociada al nuevo proceso.

Por último, el argumento lpProcessInformation, puntero a una estructura de tipo PROCESS_INFORMATION, se almacenará información sobre el nuevo proceso creado.

Terminación de un Proceso

Los servicios relacionados con la terminación de procesos se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso y servicios para esperar la terminación de un proceso. Estos servicios se describen a continuación:

3. Terminar la ejecución de un proceso: Un proceso puede finalizar su ejecución de forma voluntaria de tres formas:
   - ✓ Ejecutando dentro de la función *main* la sentencia *return*.
   - ✓ Ejecutando el servicio ExitProcess.
   - ✓ La función de la biblioteca de C *exit*. Esta función es similar a ExitProcess.

   El prototipo de la función ExitProcess es el siguiente:

   VOID ExitProcess (UINT nExitCode);

   La llamada cierra todos los manejadores abiertos por el proceso y especifica el código de salida del proceso. Este código lo puede obtener oro proceso mediante el siguiente servicio:

   BOOL GetExitCodeProcess (HANDLE hprocess, LPDWORD lpdwExitCode);

   Esta función devuelve el código de terminación del proceso con manejador hProcess. El proceso especificado por hProcess debe tener el acceso PROCESS_QUERY_INFORMATION. Si el proceso todavía no ha terminado, la función devuelve en lpdwExitCode el valor STILL_ALIVE, en caso contrario almacenará en este valor el código de terminación.

   Además, un proceso puede finalizar la ejecución de otro mediante el servicio:

BOOL TerminateProcess (HANDLE hProcess, UINT uExitCode);

Este servicio aborta la ejecución del proceso con manejador hProcess. El código de terminación para el proceso vendrá dado por el argumento uExitCode. La función devuelve TRUE si se ejecuta con éxito.

4. Esperar por la finalización de un proceso

En Win32 un proceso puede esperar la terminación de cualquier otro proceso siempre que tenga permisos para ello y disponga del manejador correspondiente. Para ello, se utilizan las funciones de espera de propósito general. Estas funciones son:

DWORD WairForSingleObject (HANDLE hObject, DWORD dwTimeOut);

DWORD WaitForMultipleObjects (DWORD cObjects, LPHANDLE lphObjects, BOOL fWaitAll, DWORD dwTimOt);

La primera función bloquea al proceso hasta que el proceso con manejador hObject finalice su ejecución. El argumento dwTimeOut específica el tiempo máximo de bloqueo expresado en milisegundos. Un valor de 0 hace que la función vuelva inmediatamente después de comprobar si el proceso finalizó la ejecución. Si el valor es INFINITE, la función bloquea el proceso hasta que el proceso acabe su ejecución.

La segunda función permite esperar la terminación de varios procesos. El argumento cObjects especifica el número de procesos (el tamaño del vector lphObjects) por los que se desea esperar.

El argumento lphObjects es un vector con los manejadores de los procesos sobre los que se quiere esperar. Si el parámetro fWaitAll es TRUE, entonces la función debe esperar por todos los procesos, en caso contrario la función vuelve tan pronto como un proceso haya acabado. El parámetro dwTimeOut tiene el significado descrito anteriormente.

Estas funciones, aplicadas a procesos, pueden devolver los siguientes valores:

- ✓ WAIT_OBJECT_0: Indica que el proceso terminó en el caso de la función WaitForSingleObject, o todos los procesos terminaron si en WaitForMultipleObjects el parámetro WaitAll es TRUE.
- ✓ WAIT_OBJECT_0+n, donde 0 <= n<= cObjects. Restando este valor de WAIT_OBJECT_0 se puede determinar el número de procesos que han acabado.
- ✓ WAIT_TIMEOUT: Indica que el tiempo de espera expiró antes de que algún proceso acabara.

Las funciones devuelven 0xFFFFFFFF en caso de error.

En Windows NT la unidad básica de ejecución es el proceso ligero y, por tanto, la planificación se realiza sobre este tipo de procesos. Windows NT implementa una planificación cíclica (Round Robin) con prioridades y con expulsión. Existen 32 niveles de prioridad, de 0 a 31, siendo 31 el nivel de prioridad máximo. Estos niveles se dividen en tres categorías:

- ✓ Dieciséis niveles con prioridades de tiempo real (niveles 16 a 31).
- ✓ Quince niveles con prioridades variables (niveles 1 al 15).
- ✓ Un nivel de sistema (0).

Todos los procesos ligeros en el mismo nivel se ejecutan según una política de planificación cíclica (Round Robin) con una determinada rodaja de tiempo. En la primera categoría, todos los procesos ligeros tienen una prioridad fija. En la segunda, los procesos comienzan su ejecución con una determinada prioridad y ésta va cambiando durante la vida del proceso, pero sin llegar al nivel 16. Esta prioridad se modifica según el comportamiento que tiene el proceso durante su ejecución. Así, un proceso (situado en el nivel de prioridades variable) ve decrementada su prioridad si acaba la rodaja de tiempo. En cambio, si el proceso se bloquea, por ejemplo, por una petición de E/S bloqueante, su prioridad aumentará. Con esto se persigue mejorar el tiempo de respuesta de los procesos interactivos que realizan E/S.

Llamadas al sistema para la administración de hilos

Creación de procesos ligeros

En Win32, los procesos ligeros se crean mediante la función CreateThread. El prototipo de esta función es:

```
BOOL CreateThre ad (
                    LPSECURITY_ATTRIBUTES lpsa,
                    DWORD cbstack,
                    LPTHREAD_START_ROUTINE lpStartAddr;
                    LPVOID lpvThreadParam,
                    DWORD fdwCreate,
                    LPDWORD lpIdThread) ;
```

Esta función crea un proceso ligero. El argumento lpsa contiene la estructura con los atributos de seguridad asociados al nuevo proceso ligero. El argumento cbStack específica el tamaño de la pila asociada al proceso ligero. Un valor de cero especifica el tamaño por defecto (1M). lpStartAddr apunta a la función a ser ejecutada por el proceso ligero. El parámetro lpvThreadParam almacena el parámetro pasado al proceso ligero. Si fdwCreate es cero, el proceso ligero se ejecuta inmediatamente después de su creación. En lpIdThread se almacena el identificador del nuevo proceso creado. La función CreateThread devuelve el manejador para el nuevo proceso ligero creado o bien NULL en caso de error.

Terminación de Procesos Ligeros

Al igual que con los procesos en Win32, los servicios relacionados con la terminación de procesos ligeros se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso ligero y servicios para esperar la terminación de procesos(WaitForSingleObject y WaitForMultipleObjects).

Terminar la ejecución de un proceso: un proceso ligero puede finalizar su ejecución de forma voluntaria de dos formas:

- ✓ Ejecutando dentro de la función principal del proceso ligero la sentencia *return.*
- ✓ Ejecutando el servicio ExitThread.

El prototipo de la función Exithread es:

VOID ExitThread (DWORD dwExitCode);

Con esta función un proceso ligero finaliza su ejecución especificando su código de salida mediante el argumento dwExitCode. Este código puede consultarse con la función GetExitCodeThread similar a la función GetExitCodeProcess.

Un proceso ligero puede también abortar la ejecución de otro proceso ligero mediante el servicio:

BOOL TerminateThread (HANDLE hThread, DWORD dwExitCode);

Similar a la función TerminateProcess.

Servicios de planificación en win32

Win32 ofrece servicios para que los usuarios puedan modificar aspectos relacionados con la prioridad y planificación de los procesos y de los procesos ligeros. La prioridad de ejecución va asociada a los procesos ligeros, ya que éstos son las unidades básicas de ejecución. La clase de prioridad va asociada a los procesos.

La prioridad de cada proceso ligero se determina según los siguientes criterios:

- ✓ La clase de prioridad de su proceso.
- ✓ El nivel de prioridad del proceso ligero dentro de la clase de prioridad de su proceso.

En Windows NT existen seis clases de prioridad que se fijan inicialmente en la llamada CreateProcess. Estas seis clases son:

- ✓ IDLE_PRIORITY_CLASS, con prioridad base 4.
- ✓ BELOW_NORMAL_PRIORITY_CLASS, con prioridad base 6.
- ✓ NORMAL_PRIORITY_CLASS, con prioridad base 9.
- ✓ ABOVE_NORMAL_PRIORITY_CLASS, con prioridad base 10.
- ✓ HIGH_PRIORITY_CLASS, con prioridad base 13.
- ✓ REAL_TIME_PRIORITY_CLASS, con prioridad base 24.

La clase prioridad por defecto para un proceso es NORMAL_PRIORITY_CLASS.

Un proceso puede modificar o consultar su clase o la de otro proceso utilizando los siguientes servicios:

BOOL SetPriorityClass (HANDLE hProcess, DWORD fdwPriorityClass);

DWORD GetPriorityClass (HANDLE hProcess) ;

Las funciones del API Win32 relacionadas con la planificación se describen a continuación:

- ✓ Suspend / Resume Thread : Suspende o reanuda un proceso detenido.
- ✓ Get / SetPriorityClass : Devuelve o fija la clase de prioridad de un proceso.
- ✓ Get / SetThreadPriority : Devuelve o fija la prioridad de un subproceso.
- ✓ Get / SetProcessAffinityMask : Devuelve o fija la máscara de afinidad del proceso.
- ✓ SetThreadAffinityMask : Fija la máscara de afinidad de un subproceso (debe ser un subconjunto de la máscara de afinidad del proceso) para un conjunto particular de procesadores, de forma que se restrinja la ejecución en esos procesadores.
- ✓ Get / SetThreadPriorityBoost : Devuelve o fija la capacidad de Windows NT para incrementar la prioridad de un subproceso temporalmente (sólo se aplica a subprocesos en el rango dinámico).
- ✓ SetThreadIdealProcesor : Establece el procesador preferido para un subproceso en particular, aunque no limita el subproceso a dicho procesador.
- ✓ Get / SetProcessPriorityBoost : Devuelve o fija el estado de control de incremento de prioridad predeterminado del proceso actual.
- ✓ SwitchToThread : Cede la ejecución del quantum a otro subproceso que está preparado para ejecutarse en el proceso actual.

- ✓ Sleep : Pone el subproceso actual en el estado de espera durante un intervalo de tiempo especificado. El valor cero hace que se pierda el resto del quantum del subproceso.
- ✓ SleepEx : Hace que el subproceso actual pase al estado de espera hasta que se termine una operación de E / S o se transcurra el intervalo de tiempo especificado.

Administrador de tareas

El Administrador de tareas proporciona información acerca de los programas y procesos que se están ejecutando en el equipo. También muestra las medidas de rendimiento utilizadas normalmente para los procesos. Se utiliza para supervisar los indicadores principales del rendimiento del equipo. Puede ver rápidamente el estado de los programas que se están ejecutando y terminar programas que han dejado de responder. También puede evaluar la actividad de los procesos en marcha utilizando hasta 15 parámetros y ver gráficos y datos acerca de la utilización de la CPU y de la memoria.

La ficha *Aplicaciones* muestra el estado de los programas que se están ejecutando en el equipo. En esta ficha puede finalizar, cambiar a o iniciar un programa.

La ficha *Procesos* muestra información acerca de los procesos que se están ejecutando en el equipo. Por ejemplo, puede mostrar información acerca de la utilización de la CPU y de la memoria, errores de paginación, recuento de identificadores y otros parámetros.

Para terminar un programa con el Administrador de tareas, en la ficha *Aplicaciones*, haga clic en la tarea que desea terminar. Luego, haga clic en *Finalizar tarea.*

Si un programa deja de responder, presione CTRL+ALT+SUPR para iniciar el Administrador de tareas, haga clic en la ficha *Aplicaciones*, vaya al programa que no responde y haga clic en *Finalizar tarea.* Se perderán los datos introducidos o los cambios realizados en ese programa y que no se hayan guardado.

✓ Para terminar un proceso con el Administrador de tareas

Haga clic en la ficha *Procesos* y luego seleccione el proceso que desea terminar. Haga clic en *Terminar proceso.* Se debe tener cuidado al terminar un proceso. Si termina una aplicación, perderá los datos que no haya guardado. Si termina un servicio del sistema, alguna parte del sistema puede que no funcione correctamente.

Puede terminar un proceso más todos los procesos creados, directa o indirectamente, por el mismo. En la ficha *Procesos*, haga clic con el botón secundario del *mouse* en el proceso que desea terminar y, después, seleccione *Finalizar el árbol de procesos*.

✓ Para cambiar a otro programa

En la ficha *Aplicaciones*, haga clic en el programa al que desea cambiar. Haga clic en *Pasar a.* Si desea iniciar un programa nuevo, en la ficha *Aplicaciones*, haga clic en

Nueva tarea. En *Abrir*, escriba la ubicación y el nombre del programa que desee y, a continuación, haga clic en *Aceptar*.

- ✓ Para supervisar el rendimiento del equipo con el Administrador de tareas

Haga clic en la ficha *Rendimiento*.

- ✓ Para mostrar otros contadores de proceso

En la ficha *Procesos*, haga clic en *Ver* y en *Seleccionar columnas*. Luego, haga clic en los elementos que desee que aparezcan como encabezados de columna y, después, en *Aceptar*.

- ✓ Para ordenar la lista de procesos

En la ficha *Procesos*, haga clic en el encabezado de columna por el que desea ordenar. Para invertir el orden, haga clic en el encabezado de la columna una segunda vez.

La opción *Mostrar cronología del núcleo* agrega líneas rojas a los gráficos *Uso de CPU* e *Historial de uso de CPU*. Las líneas rojas indican la cantidad de recursos de CPU que utilizan las operaciones del núcleo.

Para cambiar la frecuencia con que se actualizan los datos automáticamente, en el menú *Ver*, seleccione *Velocidad de actualización* y, después, haga clic en la opción que desee. Para liberar temporalmente los datos mostrados en el Administrador de tareas, en el menú *Ver*, haga clic en *Velocidad de actualización* y, después, seleccione *Pausar*.

- ✓ Para cambiar la prioridad de un programa en ejecución

En la ficha *Procesos*, haga clic con el botón secundario del mouse en el programa que desea cambiar. Seleccione *Establecer prioridad* y, después, haga clic en la opción que desee.

- ✓ Para ver la prioridad de los programas que se están ejecutando.

Presione la ficha *Procesos*, seleccione *Ver*, *Seleccionar columnas*, *Prioridad base* y, a continuación, haga clic en *Aceptar*. Cambiar la prioridad de un proceso puede hacer que se ejecute más rápido o más lento (en función de si aumenta o disminuye la prioridad) y también puede afectar adversamente al rendimiento de otros procesos.

DESARROLLO DE LA PRÁCTICA

PROCESOS

Linux

Procedimiento

El siguiente programa muestra la creación de dos procesos mediante la llamada al sistema fork( ), en donde el proceso hijo ejecuta las llamadas al sistema getpid( ) y getppid( ) para obtener el identificador de procesos propio y el de su padre, además ejecuta el comando ls -la mediante la llamada al sistema execlp( ). El proceso padre de forma similar obtiene su PID y el de su hijo, y ejecuta el comando ps -u utilizando la llamada al sistema execv( ).

El código fuente es el siguiente:

```
include <stdlib.h>
int main( int argc, char *argv[], char *env[] )
{
    pid_t id, id_padre, id_fork;  //Declaracion de variables de //tipo
    pid_t que guardan los identificadores de proceso
    char *arg[3];
    int estado;
```

```
switch(id_fork = fork())    / / Llamada al sistema para la
                                / / creacion de procesos
{
   case -1:
         perror("Error en la llamada fork");
         exit(0);
         break;
   case 0:                    / / Codigo del proceso hijo
         id = getpid();       / / Comando para obtener el ID
                            / / del proceso
         id_padre = getppid(); / / Comando para obtener
                                / / el ID del proceso padre
printf("\n\nHijo: Mi pid es: %d \n", id);
printf("Hijo: Mi padre es: %d \n", id_padre);
printf("Hijo: Ejecuto el comando ls -la: \n\n");
         execlp ("ls", "ls","-la", "./", (char *) 0);  / / Llamada / / al sistema
         para ejecutar comandos
         perror ("Error en la ejecucion de ls -la");
         exit(0);                / / Llamada al sistema para
                               / / terminar el proceso
         break;
      default:
         id = getpid();
         printf("\nPadre: Mi pid es: %d \n", id);
         printf("Padre: Mi hijo es: %d \n", id_fork);
         / / La llamada al sistema fork() devuelve el PID
         / / del proceso hijo al padre
```

```
        printf("Padre: Ejecuto ps -u \n\n");
        arg[0] = "ps";
        arg[1] = "-u";
        arg[2] = (char *)0;
        execv("/bin/ps", arg);
        //Llamada al sistema para ejecutar comandos
        perror("Error en la ejecucion de ls -l");
        wait(&estado);
        //El proceso padre espera que el proceso hijo
        }               //termine de ejecutar su codigo.
    exit(0);   //Llamada al sistema para terminar el proceso
    }
```

1. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi, y guárdelo con el nombre procesos.c o busque el archivo procesos.c en el directorio ./Guia2/Linux.

2. Compile el archivo procesos.c de la siguiente manera:

gcc –o procesos procesos.c

3.Luego ejecútelo

./procesos

Windows

Procedimiento

1. En este programa mediante la librería <windows.h> y las estructuras STARTUPINFO y PROCESS_INFORMATION de la función CreateProcess se crea un proceso que ejecuta el bloc de notas. El código fuente es el siguiente:

```
#include <windows.h>
#include <stdio.h>
#include <string.h>

int main()
{
    char CadenaProceso[50];
    char WinDir[50];
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    si.cb = sizeof(si); //Tamaño de la estructura STARTUPINFO
    si.dwFlags=STARTF_USESHOWWINDOW; //Para que se considere
                                     //el miembro
                                   //wShowWindow de dwFlags
    si.wShowWindow = SW_SHOWNORMAL; //La ventana del nuevo
                                    //proceso es una //ventana
                                    normal

    GetWindowsDirectory(WinDir,sizeof(WinDir));
    strcpy(CadenaProceso,strcat(WinDir,"\\Notepad.exe"));
```

```
    printf ("Directorio del nuevo proceso que se ejecuta:
    %s\n",CadenaProceso);
    if (!CreateProcess(CadenaProceso, // string aplicación
                NULL,        // línea de comando
                    NULL,        // seguridad proceso
                        NULL,        // seguridad hilo
                        FALSE,      // no hay herencia de
                                                  //handles
                        0,           // sin banderas
                        NULL,     // hereda bloque de entorno
                        NULL,      // hereda directorio actual
                    &si,         // puntero a STARTUPINFO
                        &pi)) // puntero a PROCESS_INFORMATION
{
   if (GetLastError() == ERROR_FILE_NOT_FOUND)
   {
      printf ("Imposible encontrar el archivo. \n");
     return (-1);
   }
    }
    else{
          printf ("Se creo un proceso \n");
        }
        return 0;
        }
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre CrearProceso.cpp o busque el archivo CrearProceso.cpp en el directorio C:\Guia2\Windows.

3. Compile el archivo CrearProceso.cpp

4. Ejecute el archivo CrearProceso.exe

HILOS

Linux

Procedimiento

El siguiente programa crea dos hilos que ejecutan dos funciones diferentes. El primer hilo muestra su ID mediante el uso de la llamada pthread_self() y el otro hilo ejecuta el comando ls –la. El código fuente es el siguiente:

```
#include <pthread.h>  //Libreria para utilizar hilos
#include <stdio.h>
void *ID()  //Funcion que ejecuta el hilo1
{
      printf("\nHilo1: ID %d \n",pthread_self());
}
void *ejecutar()   //Funcion que ejecuta el hilo2
{
      printf("\n\nHilo2: Ejecuto ls -la\n\n");
      execlp ("ls", "ls","-la", "./", (char *) 0);
```

```
        perror ("Error en la ejecucion de ls -la");
}
main()
{
        pthread_t hilo1,hilo2;  //Declaración de los hilos
        pthread_create(&hilo1,NULL,ID,NULL);  //Creacion de los hilos
        pthread_create(&hilo2,NULL,ejecutar,NULL);
        pthread_join(hilo1,NULL);  //Espera que cada hilo termine
        pthread_join(hilo2,NULL);
        exit(0);
}
```

1.Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre hilos.c o busque el archivo hilos.c en el directorio ./Guia2/Linux.

2.Compile el archivo hilos.c de la siguiente manera:

```
gcc –o hilos hilos.c –lpthread
```

3.Luego ejecútelo:

```
./hilos
```

Windows

Procedimiento

1. Este programa mediante el uso de la función CreateThread, crea un hilo que muestra una figura geométrica.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
int a;
void Hilo()//Función que es ejecutada por el hilo
{
	int x,y;
	for (x=0; x<10; x++, printf("\n"))
		for (y=0; y<10; y++)
			printf("X");
}
void main(void)
{
HANDLE HandleHilo;
DWORD IDHilo;
HandleHilo=CreateThread(0, 0, (LPTHREAD_START_ROUTINE) Hilo,0,
0, &IDHilo);//Se crea el hilo
if (HandleHilo==0)
	printf("No puedo crear el hilo. Error numero %x\n", GetLastError());
a=getch();
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre CrearProceso.cpp o busque el archivo Hilos.cpp en el directorio C:\Guia2\Windows.

3. Compile el archivo: Hilos.cpp

4. Ejecute el archivo: Hilos.exe

ENUNCIADO DE LA PRÁCTICA

Linux

1. Edite un programa en lenguaje C, que sea capaz de crear dos procesos en donde el proceso hilo se encargue de ejecutar comandos del shell tomados como datos de entrada y el proceso padre muestre su ID. Utilice las llamadas al sistema para el control de procesos.

2. Edite un programa en lenguaje C que cree un determinado número de hilos que modifiquen el valor de dos variables.

Windows

1. Elaborar un programa que mediante la librería <windows.h> y la función CreateProcess cree un proceso que ejecute la calculadora. Además, debe mostrar los siguientes datos: El handle del proceso nuevo, el handle del hilo hijo, el ID del proceso y el ID del hilo. Finalmente, debe cerrar el handle del proceso, el handle del hilo y explicar qué ocurre.

2. Elaborar un programa que cree un hilo que ejecute una función de incremento de una variable.

DESARROLLO

Linux

1. Edite un programa en lenguaje C, que sea capaz de crear dos procesos en donde el proceso hilo se encargue de ejecutar comandos del shell tomados como datos de entrada y el proceso padre muestre su ID. Utilice las llamadas al sistema para el control de procesos.

```
#include <sys/types.h>
#include <stdio.h>
main(int argc, char **argv)
{
        pid_t pid;
        pid = fork( );  //Llamada al sistema para crear procesos
        switch(pid)
        {
                case -1: /* error del fork( ) */
                        exit(-1);  //Llamada al sistema para salir
                                   //en caso de error
                case 0: /* proceso hijo */
                        if (argv[1]==NULL)
                        {
                                printf("Indique el comando a
                                ejecutar \n");
                                exit(0);
```

```
                    }
                    else
                        //llamada al sistema para ejecutar
                        //comandos
                      if (execvp(argv[1], &argv[1]) < 0)
                          perror("Error al ejecutar el comando");
            default: /* padre */
                    //Obtiene el identificador del
                    //proceso padre
                    printf("\nPadre  ID=%d \n",getppid());
                    //Obtiene el identificador del proceso hijo
                    printf("Hijo   ID=%d \n\n",getpid());
                    exit(0);
     }
    }
```

2. Edite un programa en lenguaje C que cree un determinado número de hilos que modifiquen el valor de dos variables.

```
#include <pthread.h>      //Librería para utilizar hilos
#include <stdio.h>
#define HILOS 7           //Numero de hilos a crear
int a,b;                          //Variables acceden los hilos
void *funcion(void *NoHilo)//Función que ejecuta cada hilo
{
        sleep(3);         //Cada hilo duerme 3seg
        printf("Hilo %d",NoHilo+1);
```

```
    a= a + 5;
    b= b - 5;
    printf(": a= %d",a);
    printf(" y b= %d \ n",b);
    pthread_exit(0);       / / Llamada al sistema para finalizar el hilo
}

int main(int argc, char *argv[])
{
    pthread_t hilo[HILOS];   / / Declaración de los hilos
    int error, i;
    a=0;
    b=40;
    printf("\ n Antes de crear los hilos los valores de las variables
    son: \ n");
    printf("a= %d",a);
    printf(" y b= %d \ n \ n",b);
    printf("Se estan creando 7 hilos... \ n");
    for(i=0;i<HILOS;i++)
    {
            / / Llamada al sistema para crear los hilos
            error = pthread_create(&hilo[i], NULL, funcion,
            (void *)i);
        if (error)
        {
            printf("Error al crear hilos \ n");
            exit(-1);
```

```
                    }
                }
                pthread_exit(NULL);
            }
```

Windows

1. Elaborar un programa que mediante la librería <windows.h> y la función CreateProcess cree un proceso que ejecute la calculadora. Además, debe mostrar los siguientes datos: El handle del proceso nuevo, el handle del hilo hijo, el ID del proceso y el ID del hilo. Finalmente, debe cerrar el handle del proceso, el handle del hilo y explicar qué ocurre.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>

int a;
int main()
{
    char CadenaProceso[50];
    char WinDir[50];
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    si.cb = sizeof(si); / /Tamaño de la estructura STARTUPINFO
```

```
si.dwFlags=STARTF_USESHOWWINDOW; // Para que se considere el
                                 // miembro wShowWindow
si.wShowWindow = SW_SHOWNORMAL; // La ventana del nuevo
                                // proceso será una ventana normal
GetWindowsDirectory(WinDir,sizeof(WinDir));
strcpy(CadenaProceso,strcat(WinDir,"\\Calc.exe"));
printf ("Directorio del nuevo proceso que se ejecuta: %s\n",CadenaProceso);
if (!CreateProcess(CadenaProceso,   // string aplicación
            NULL,        // línea de comando
            NULL,        // seguridad proceso
            NULL,        // seguridad hilo
            FALSE,     // no hay herencia de handles
            0,           // sin banderas
            NULL,        // hereda bloque de entorno
            NULL,        // hereda directorio actual
            &si,         // puntero a STARTUPINFO
            &pi))    // puntero a PROCESS_INFORMATION

{
   if (GetLastError() == ERROR_FILE_NOT_FOUND)
   {
      printf ("Imposible encontrar el archivo. \n");
      return (-1);
   }
```

```
}else{
    printf ("Se creo un proceso. \n");
    printf ("El handle del proceso nuevo es %d. \n",pi.hProcess);
    printf ("El handle del hilo hijo es %d. \n",pi.hThread);
    printf ("El Process Id es %x. \n",pi.dwProcessId);
    printf ("El Thread Id es %x. \n \n",pi.dwThreadId);
    printf ("Cerramos el handle del objeto proceso. \n");
    CloseHandle(pi.hProcess);
    printf ("Cerramos el handle del objeto hilo. \n");
    CloseHandle(pi.hThread);
}
a=getch();
return 0;
}
```

La función CloseHandle no se termina o elimina el objeto, sino que el proceso al llamar a CloseHandle lo que hace es decirle al sistema operativo que ya no quiere acceder más a ese objeto. El sistema operativo acepta esa solicitud y desliga al proceso del objeto. Si el sistema operativo ve que nadie más accede al objeto, es decir, que ningún otro proceso tiene un handle abierto para ese objeto, elimina al objeto del sistema.

2. Elaborar un programa que cree un hilo que ejecute una función de incremento de una variable.

```
#include <windows.h>
#include <stdio.h>
```

```
volatile UINT contador;
void HiloContador()
{
        while(1)
        {
                contador++;
                Sleep(1000);
        }
}
void main(void)
{
        HANDLE HandleHilo;
        DWORD IDHilo;
        contador=0;
                HandleHilo=CreateThread(0,                              0,
                (LPTHREAD_START_ROUTINE) HiloContador,
                0, 0, &IDHilo);
                if (HandleHilo==0)
                printf("No puedo crear el hilo. Error numero %x\n",
                GetLastError());
                while(1)
                {
                        printf("Pulsar <ENTER> para ver el valor del
                        contador.\n", getchar());
                        printf("El valor del contador es: %d", contador);
                }
      }
```

ANEXOS

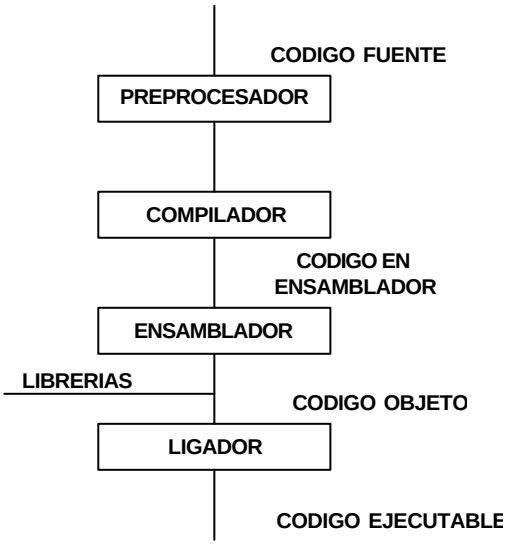
Compilación de programas en Windows

Para compilar los programas en windows se debe disponer del software Borland C++ o cualquier otro compilador de C que incluya la librería <windows.h>

Compilación de programas en Linux

Etapas de Compilación

El proceso de compilación involucra cuatro etapas sucesivas: preprocesamiento, compilación, ensamblado y enlazado.



- ✓ *Preprocesado:* En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con #define son sustituidas en el código por su valor en todos los lugares donde aparece su nombre.

- ✓ *Compilación:* La compilación transforma el código C en el lenguaje ensamblador propio del procesador de la máquina.

- ✓ *Ensamblado:* El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.

- ✓ *Enlazado:* Las funciones de C/C++ incluidas en el código, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones al ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.

Compilación en GCC (GNU Compiler Collection)

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr.

Sintaxis

gcc [opción | archivo]

Opciones

-c	Realiza preprocesamiento y compilación, obteniendo el archivo en código objeto; no realiza el enlazado.
-E	Realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.
-o *archivo*	Indica el nombre del archivo de salida, cualesquiera sean las etapas cumplidas.
-I*ruta*	Especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la I y la ruta, por ejemplo: -I/usr/include.
-L*ruta*	Especifica la ruta hacia el directorio donde se encuentran los archivos de biblioteca con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la L y la ruta, por ejemplo: -L/usr/lib
-Wall	Muestra todos los mensajes de error y advertencia del compilador.
-g	Incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB (GNU Debugger).
-v	Muestra los comandos ejecutados en cada etapa de compilación y la versión del compilador. Es un informe muy detallado.

REFERENCIAS

http://www.atc.unican.es/~jagm/cii/Transparencias/procUnix.pdf

http://www.lab.dit.upm.es/~lprs/presentaciones/concurrencia_t8_1p.pdf

http://bernia.disca.upv.es/~eso/06-planificacion/06-planificacion.pdf

http://www.infor.uva.es/~arturo/Asig/InfASO/UCap6.html

http://www.lsi.us.es/docencia/cursos/seminario-1.html

http://aula.linux.org.ar/docs/tecnicos/index/cap2.html

http://bari.ufps.edu.co/personal/150802A/planificacion.htm

http://www.monografias.com/trabajos7/arso/arso.shtml#dise

http://www.cs.rpi.edu/courses/fall01/os/CreateProcess.html

http://www.adapower.com/os/win32-createprocess.html

http://www.global-shared.com/api/exec/crproces.html

http://winapi.conclase.net/

http://winapi.conclase.net/curso/index.php?000

SINCRONIZACIÓN DE PROCESOS

GUIA DEL DOCENTE

SINCRONIZACIÓN DE PROCESOS
GUIA DEL DOCENTE

OBJETIVOS

- ✓ Identificar, analizar y mostrar cómo se utilizan los principales mecanismos que ofrecen los sistemas operativos Linux y Windows para la sincronización de procesos.

- ✓ Trabajar con el sistema de hilos y las herramientas de concurrencia que ofrece el sistema operativo Unix (POSIX) en la implementación de regiones críticas mediante semáforos, mútex y variables de condición.

MARCO TEÓRICO

Normalmente, las aplicaciones constan de varios procesos ejecutándose de forma concurrente. Por lo tanto surgen necesidades:

- ✓ Compartir información entre los procesos.
- ✓ Intercambio de información entre los procesos.
- ✓ Sincronización entre los procesos.

La sincronización bajo memoria compartida plantea dos necesidades:

- ✓ Exclusión mutua: Protección de una zona de código que manipula recursos comunes.
- ✓ Espera de una condición: Las tareas se suspenden a la espera de que se produzca una condición, esta puede estar relacionada por los recursos que las tareas comparten.

LINUX

Los mecanismos de sincronización que se utilizan con más frecuencia en Linux son:

Semáforos

Un semáforo es un objeto con un valor entero al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: wait y signal. Los semáforos son mecanismos de sincronización útiles para coordinar el acceso a recursos. En POSIX, un semáforo se identifica mediante una variable del tipo sem_t. El estándar POSIX define dos tipos de semáforos:

- ✓ **Semáforos sin nombre**. Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso o a los procesos que lo heredan a través de la llamada fork.
- ✓ **Semáforos con nombre**. En este caso, el semáforo lleva asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este

tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada fork.

Creación de un semáforo sin nombre

Todos los semáforos en POSIX deben iniciarse antes de su uso. La función **sem_init** permite iniciar un semáforo sin nombre. El prototipo de este servicio es el siguiente:

int **sem_init**(sem_t *sem, int shared, int val);

Con este servicio se crea y se asigna un valor inicial a un semáforo sin nombre. El primer argumento identifica la variable de tipo semáforo que se quiere utilizar. El segundo argumento indica si el semáforo se puede utilizar para sincronizar procesos ligeros o cualquier otro tipo de proceso. Si shared es 0, el semáforo sólo puede utilizarse entre los procesos ligeros creados dentro del proceso que inicia el semáforo. Si shared es distinto de 0, entonces se puede utilizar para sincronizar procesos que lo hereden por medio de la llamada fork. El tercer argumento representa el valor que se asigna inicialmente al semáforo.

Destrucción de un semáforo sin nombre

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada sem_init. Su prototipo es el siguiente:

int **sem_destroy**(sem_t *sem)

Creación y apertura de un semáforo con nombre

El servicio **sem_open** permite crear o abrir un semáforo con nombre. La función que se utiliza para invocar este servicio admite dos modalidades, según se utilice para crear el semáforo o simplemente abrir uno existente. Estas modalidades son las siguientes:

sem_t ***sem_open**(char *name, int flag, mode_t mode, int val);

sem_t ***sem_open**(char *name, int flag);

Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un archivo. La función **sem_open** establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

El valor del segundo argumento determina si la función **sem_open** accede a un semáforo previamente creado o si crea un nuevo. Un valor 0 en flag indica que se quiere utilizar un semáforo que ya ha sido creado, en este caso no es necesario los dos últimos parámetros de la función **sem_open**. Si flag tiene un valor O_CREAT, requiere los dos últimos argumentos de la función. El tercer parámetro especifica los permisos del semáforo que se va a crear, de la misma forma que ocurre en la llamada open para archivos. El cuarto parámetro especifica el valor inicial del semáforo.

Cierre de un semáforo con nombre

Cierra un semáforo con nombre rompiendo la asociación que tenía un proceso con un semáforo. El prototipo de la función es:

int **sem_close**(sem_t *sem);

Borrado de un semáforo con nombre

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función **sem_close.** El prototipo de este servicio es:

int **sem_unlink**(char *name);

Operación wait

La operación wait en POSIX se consigue con el siguiente servicio:

int **sem_wait**(sem_t *sem);

Operación signal

Este servicio se corresponde con la operación signal sobre un semáforo. El prototipo de este servicio es:

int **sem_post**(sem_t *sem);

Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito o –1 en caso de error.

Mútex

Son mecanismos de sincronización a nivel de hilos, se utilizan para garantizar la exclusión mutua en el acceso a un código. Cada mútex posee:

- ✓ Dos estados internos: abierto y cerrado
- ✓ Un hilo propietario, cuando el mútex está cerrado

Llamadas POSIX:

Creación del mútex: pthread_mutex_init

- ✓ int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- ✓ pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

donde podemos crear el mútex con los atributos **attr** o con los atributos por defecto (PTHREAD_MUTEX_INITIALIZER)

Destrucción del mútex: pthread_mutex_destroy

INT pthread_mutex_destroy(pthread_mutex_t * mutex);

Atributos de creación de mútex

- ✓ int pthread_mutexattr_init(pthread_mutexattr_t *attr);

- ✓ int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

Modificación de atributos de creación de mútex

- ✓ int pthread_mutexattr_getpshared (const pthread_mutexattr_t *attr, int *pshared);
- ✓ int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);

donde **pshared** indica si el mútex podrá ser utilizado por hilos del proceso que creó el mútex o por hilos de otros procesos:

PTHREAD_PROCESS_PRIVATE
PTHREAS_PROCESS_SHARED

- ✓ int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
- ✓ int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol);

donde **protocol** indica el protocolo de sincronización del mútex:

- ⇒ PTHREAD_PRIO_NONE: Sin protocolo
- ⇒ PTHREAD_PRIO_INHERIT: Protocolo Básico de herencia de prioridad
- ⇒ PTHREAD_PRIO_PROTEC: Protocolo de techo de prioridad inmediato

- ✓ int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);

- ✓ int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int *prioceiling);

donde **prioceiling** indica el techo de prioridad del mútex. Solo tiene sentido en la política de sincronización PTHREAD_PRIO_PROTEC.

- ✓ int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
- ✓ int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int type);

donde **type** indica el tipo de mútex (respecto a interbloqueos):

⇒ PTHREAD_MUTEX_NORMAL: sin comprobación, interbloqueo posible
⇒ PTHREAD_MUTEX_ERRORCHECK: con comprobación de errores
⇒ PTHREAD_MUTEX_RECURSIVE: admite varios cierres seguidos (del mismo hilo) sin apertura previa.
⇒ PTHREAD_MUTEX_DEFAULT: sin comprobación (dependiente de la implementación)

Cierre y apertura de mútex

- ✓ int pthread_mutex_lock(pthread_mutex_t *mutex);
- ✓ int pthread_mutex_trylock(pthread_mutex_t *mutex);
- ✓ int pthread_mutex_unlock(pthread_mutex_t *mutex);

donde

lock/trylock: cierra(o intenta cerrar) el mútex: Si está abierto, se cierra y el hilo invocante pasa a ser el propietario. Si está cerrado:

⇒ lock: suspende el hilo invocante
⇒ trylock: devuelve un código de error

unlock: abre el mútex. Si hay hilos suspendidos, selecciona el más prioritario y permite que cierre el mútex.

Variables de condición

Las variables de condición están asociadas con un mútex y se emplean para sincronizar los hilos. Las tres operaciones básicas son:

- ✓ Espera
- ✓ Aviso simple
- ✓ Aviso múltiple

Llamadas POSIX

Creación de variables de condición: pthread_cond_init

- ✓ int pthread_cond_init (pthread_cond_t *cond, conspthread_condattr_t *attr);
- ✓ pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

donde podemos crear la variable **cond** con los atributos **attr** o con los atributos por defecto (PTHREAD_COND_INITIALIZER)

Destrucción de variables de condición: pthread_cond_destroy

int pthread_cond_destroy(pthread_cond_t *cond);

Atributos de creación de variables de condición

- ✓ int pthread_condattr_init(pthread_condattr_t *attr);
- ✓ int pthread_condattr_destroy(pthread_condattr_t *attr);

Modificación de atributos de creación de variables de condición

- ✓ int pthread_condattr_getpshared(const pthreadcondattr_t *attr, int *phared);
- ✓ int pthread_condattr_setpshared(pthread_condattr_t *attr, int phared);

donde **phared** indica si la variable podrá ser utilizado por hilos del proceso que creó el mútex o por hilos de otros procesos:

PTHREAD_PROCESS_PRIVATE
PTHREAS_PROCESS_SHARED

Espera sobre variables de condición

- ✓ int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- ✓ int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

donde

wait: ejecuta un pthread_mutex_unlock sobre mútex y atómicamente suspende al hilo invocante en la variable cond. Al despertarse, realizará atómicamente un pthread_mutex_lock sobre mútex.

timedwait: actúa como wait, salvo que el hilo se suspende hasta que se alcanza el tiempo abstime.

Aviso sobre variables de condición

- ✓ int pthread_cond_signal(pthread_cond_t *cond);
- ✓ int pthread_cond_broadcast(pthread_cond_t *cond);

donde

signal: selecciona el hilo más prioritario suspendido en cond y lo despierta.

broadcast: despierta todos los hilos que pueda haber suspendidos en cond.

Ambas no tienen efecto si no hay hilos suspendidos en la variable. Es recomendable que el hilo invocante sea el propietario del mútex asociado a cond en los hilos suspendidos.

Windows NT

Win32 dispone de dos mecanismos de comunicación que también se pueden utilizar para sincronizar. Estos mecanismos son las tuberías y los mailslots. Como mecanismos de sincronización puros, Win32 dispone de secciones críticas, semáforos, mútex y eventos.

Secciones críticas

Las secciones críticas son un mecanismo de sincronización especialmente concebido para resolver el acceso a secciones de código que deben ejecutarse en exclusión mutua.

Las secciones críticas son objetos que se crean y se borran pero no tienen manejadores asociados. Sólo se pueden utilizar por los proceso ligeros creados dentro de un proceso. Los servicios utilizados para tratar las secciones críticas son los siguientes:

```
VOID InitializeCriticalSection ( LPCCRITICAL_SECTION  lpcsCriticalsection) ;
```

VOID DeleteCriticalSection (LPCCRITICAL_SECTION lpcsCriticalSection) ;

VOID EnterCriticalSection (LPCCRITICAL_SECTION lpcsCriticalSection) ;

VOID LeaveCriticalSection (LPCCRITICAL_SECTION lpcsCriticalSection) ;

Las dos primeras secciones se utilizan para crear y borrar secciones críticas. Las dos siguientes sirven para entrar y salir de la sección crítica.

El acceso en exclusión mutua a una sección crítica se resuelve de forma muy sencilla utilizando este mecanismo de Win32. en primer lugar se deberá crear una sección crítica e inicializarla:

```
LPCCRITICAL_SECTION SC;
InitializeCriticalSection ( &SC );
```

Siempre que se desee acceder a una sección crítica deberá utilizarse el siguiente fragmento de código:

```
EnterCriticalSection ( &SC );
< Código de la sección crítica >
```

LeaveCriticalSection (&SC);

Cuando deje de accederse a la sección crítica se deberá destruir utilizando:

DeleteCriticalSection (&SC);

Semáforos

En Win32 los semáforos tienen asociado un nombre. Los servicios de win32 para trabajar con semáforos son los siguientes:

Crear un semáforo

Los semáforos en Win32 se manipulan, como el resto de los objetos, con manejadores. Para crear un semáforo se utiliza el siguiente servicio:

HANDLE CreateSemaphore (LPSECURITY_ATTRIBUTES lpsa, LONG cSemInitial, LONG cSemMax, LPCTSTR lpszSemName) ;

El primer parámetro especifica los parámetros de seguridad asociados al semáforo. El argumento cSemInitial indica el valor inicial del semáforo y cSemMax el valor máximo que puede tomar. El nombre del semáforo viene dado por el parámetro lpszSemName. La llamada devuelve un mensaje de semáforo válido o NULL en caso de error.

Abrir un semáforo

Una vez creado un semáforo, un proceso puede abrirlo mediante el servicio:

```
HANDLE OpenSemaphore ( LONG dwDesiredAccess, LONG BinheritHandle,
                       lpszName SemName ) ;
```

El parámetro dwDesiredAccess puede tomar los siguientes valores:

- ✓ SEMAPHORE_ALL_ACCESS: total acceso al semáforo.
- ✓ SEMAPHORE_MODIFY_STATE: permite la ejecución de la función ReleaseSemaphore.
- ✓ SYNCHRONIZE: permite el uso del semáforo para sincronización, es decir, en las funciones de espera (wait).

El segundo argumento indica si se puede heredar el semáforo a los procesos hijos. Un valor de TRUE permite heredarlo. SenName indica el nombre del semáforo que se desea abrir. La función devuelve un manejador de semáforo válido en caso de éxito o NULL en caso de error.

Cerrar un semáforo

Para cerrar un semáforo se utiliza el siguiente servicio:

```
BOOL CloseHandle ( HANDLE hObject );
```

Si la llamada termina correctamente, se cierra el semáforo. Si el contador del manejador es cero, se liberan los recursos ocupados por el semáforo. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Operación Wait

Se utiliza el siguiente servicio de Win32:

```
DWORD WaitForSingleObject ( HANDLE hSem, DWORD dwTimeOut );
```

Ésta es la función general de sincronización que ofrece Win32. Cuando se aplica a un semáforo implementa la función wait. El parámetro dwTimeOut debe tomar en este caso valor INFINITE.

Mútex

Los mútex, al igual que los semáforos, son objetos con nombre. Los mútex sirven para implementar secciones críticas. La diferencia entre las secciones críticas y los mútex de Win32 radica en que las secciones críticas no tienen nombre y sólo se pueden utilizar entre procesos ligeros de un mismo proceso. Un mútex se manipula usando manejadores.

Los servicios utilizados en Win32 para trabajar con mútex son los siguientes:

Crear un mútex

Para crear un mútex se utiliza el siguiente servicio:

HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpsa, BOOL fInitialOwner,

LPCTSTR lpszMutexName);

Esta función crea un mútex con atributos de seguridad lpsa. Si fInitialOwner es TRUE, el propietario del mútex será el proceso ligero que lo crea. El nombre del mútex viene dado por el tercer argumento. En caso de éxito la llamada devuelve un manejador de mútex válido y en caso de error NULL.

Abrir un mútex

Para abrir un mútex se utiliza:

HANDLE OpenMutex (LONG dwDesiredAccess, LONG BineheritHandle,

LpszName SemName);

Los parámetros y su comportamiento son similares a los de la llamada OpenSemaphore.

Cerrar un mútex

Para cerrar un semáforo se utiliza el siguiente servicio:

```
BOOL CloseHandle ( HANDLE hObject );
```

Si la llamada termina correctamente, se cierra el mútex. Si el contador del manejador es cero, se liberan los recursos ocupados por el mútex. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Operación lock

Se utiliza el siguiente servicio de win32:

```
DWORD WaitForSingleObject ( HANDLE hMutex, DWORD dwTimeOut );
```

Ésta es la función general de sincronización que ofrece Win32. Cuando se aplica a un mútex implementa la función lock. El parámetro dwTimeOut debe tomar en este caso el valor INFINITE.

Operación unlock

El prototipo de este servicio es:

```
BOOL ReleaseMutex ( HANDLE hMutex );
```

Los eventos en Win32 son comparables a las variables condicionales, es decir, se utilizan para notificar que alguna operación se ha completado o que ha ocurrido algún proceso. Sin embargo, las variables condicionales se encuentran asociadas a un mútex y los eventos no.

Los eventos en Win32 se clasifican en manuales y automáticos. Los primeros se pueden utilizar pata desbloquear a varios threads bloqueados en un evento. En este caso, el evento permanece en estado de notificación y debe eliminarse este estado de forma manual. Los automáticos se utilizan para desbloquear a un único thread, es decir, el evento notifica a un único thread y a continuación deja de estar en este estado de forma automática.

DESARROLLO DE LA PRÁCTICA

LINUX

Procedimiento

A continuación se muestra el problema de productores y consumidores, donde uno o más procesos productores generan una serie de datos que se depositan en un área de memoria compartida (buffer), de donde son extraídos por uno o más consumidores para procesarlos. Teniendo en cuenta que:

a. Un productor no puede depositar un dato cuando el buffer está lleno.
b. Un consumidor no puede extraer un dato cuando el buffer está vacío.

Semáforos

1. El código fuente del problema de productores/consumidores utilizando semáforos es el siguiente:

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#define TAMANO_BUFFER 1024 //Tamaño del Buffer
void * Consumidor(void);
```

```
void * Productor(void);
char Buffer[TAMANO_BUFFER]; / /Buffer Comun
char Mensaje[TAMANO_BUFFER];
int Leer = 0, Escribir = 0;
int Longitud;
sem_t Lleno, Vacio, mútex; / /Declaración de semáforos
main()
{
    pthread_t Hilo_Productor,Hilo_Consumidor; / /Declaración
                                              / / de los hilos
    strcpy(Mensaje,"HOLA");
    Longitud = strlen(Mensaje);
    sem_init(&Lleno, 0, TAMANO_BUFFER);  / /Inicializa el
                                         / /semáforo LLeno
    sem_init(&Vacio, 0, 0);              / /Inicializa el
                                         / /semáforo Vacio
    sem_init(&mútex, 0, 1);              / /Inicializa el
                                         / /semáforo mútex
    system("clear");
    printf("Productor    Consumidor \n");
        pthread_create(&Hilo_Consumidor, NULL, (void
        *)Consumidor, NULL);  / /Creación del hilo consumidor
        pthread_create(&Hilo_Productor, NULL, (void
        *)Productor, NULL);   / /Creación del hilo productor
        pthread_join(Hilo_Consumidor, NULL);
         / /Espera que el hilo consumidor termine
       pthread_join(Hilo_Productor, NULL);
```

```
        //Espera que el hilo productor termine
}
void * Productor(void)
{   int n = 0;
    strcpy(Buffer, "");
    while (n < 20)
    {
            sem_wait(&Lleno); //Espera que el buffer no esté
                              //lleno para producir
          sem_wait(&mútex); //Bloquea el mútex para poder
                              //escribir en el buffer
            Buffer[Escribir] = Mensaje[Escribir%Longitud];
            printf("   %c\n",Buffer[Escribir]);
            Escribir = (Escribir + 1) % TAMANO_BUFFER;
            fflush(stdout);
            sem_post(&mútex);  //Desbloquea el mútex
            sem_post(&Vacio);
            if (rand() % 4 < 2)
                    sleep(1);
            n ++;
    }
}

void * Consumidor(void)
{
    int n = 0;/*Numero de elementos en el buffer*/
    while (n < 20)
```

```
        {
                sem_wait(&Vacio);//  Espera que el buffer tenga
                                //  algún elemento
                sem_wait(&mútex);//Bloquea el mútex para poder
                                //leer en el buffer
                printf("              %c\n",Buffer[Leer]);
                Leer = (Leer + 1) % TAMANO_BUFFER;
                fflush(stdout);
                sem_post(&mútex);  //Desbloquea el mútex
                sem_post(&Lleno);
                if (rand() % 4 < 2)
                        sleep(1);
                n ++;
        }
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre Semaforo.c o busque el archivo Semaforo.c en el directorio ./Guia3/Linux.

3.Compile el archivo Semaforo.c de la siguiente manera:

```
gcc –o Semaforo Semaforo.c –lpthread
```

4.Luego ejecútelo:

```
./semáforo
```

Variables Compartidas y mútex

1. El código fuente del problema de productores/consumidores utilizando variables compartidas y mútex es el siguiente:

```
#include <pthread.h>
#include <stdio.h>
#define BUFFER 1024
#define DATOS  20
void * Productor(void);
void * Consumidor(void);
pthread_mutex_t mútex; //Declaración del mútex
pthread_cond_t Lleno;  //Declaración de las variables de condición
pthread_cond_t Vacio;
int n=0;
int Buffer[BUFFER];
main(int arg, char *arg1[])
{
pthread_t Hilo_Productor,Hilo_Consumidor;
pthread_mutex_init(&mútex, NULL); //Inicialización del mútex
pthread_cond_init(&Lleno, NULL);  //Inicialización de las
```

```
                                        / / variables de condición
pthread_cond_init(&Vacio, NULL);
system("clear");
printf("Productor     Consumidor \n");
pthread_create(&Hilo_Productor, NULL, (void *) Productor, NULL);
/ / Creación del hilo productor
pthread_create(&Hilo_Consumidor, NULL, (void *) Consumidor,
NULL); / / Creación del hilo consumidor
pthread_join(Hilo_Productor, NULL);  / / Espera que el hilo
                                              / / productor termine
pthread_join(Hilo_Consumidor, NULL);  / / Espera que el hilo
                                              / / consumidor termine
pthread_mutex_destroy(&mútex);  / / Destruye el mútex
pthread_cond_destroy(&Lleno);   / / Destruyen las variables de
                                        / / condición
pthread_cond_destroy(&Vacio);
exit(0);
}

void * Productor(void)
{
```

```
int dato, i, pos=0;
for (i=0; i<DATOS; i++)
{
        dato=i+1;
        pthread_mutex_lock(&Mutex);
         / / Bloquea el mutex para poder escribir en el buffer
        while (n==BUFFER)
                pthread_cond_wait(&Lleno, &Mutex);
        Buffer[pos]=i;
        pos = (pos + 1) % BUFFER;
        printf("     %d \n",pos);
        n = n + 1;
        if (n==1)
                pthread_cond_signal(&Vacio);
        pthread_mutex_unlock(&Mutex);    / / Desbloquea el mutex
        if (rand() % 4 < 2)
                sleep(1);
}
pthread_exit(0);
}
void * Consumidor(void)
```

```
{
int dato, i, pos=0;
for (i=0; i<DATOS; i++)
{
        pthread_mutex_lock(&Mutex); / /Bloquea el mutex para
                                    / /poder leer en el buffer
        while (n==0)
                pthread_cond_wait(&Vacio, &Mutex);
        dato=Buffer[pos];
        pos = (pos + 1) % BUFFER;
        printf("               %d \ n",pos);
        n = n - 1;
        if (n==BUFFER-1)
                pthread_cond_signal(&Lleno);
        pthread_mutex_unlock(&Mutex); / /Desbloquea el mutex

        if (rand() % 4 < 2)
                sleep(1);
}
pthread_exit(0);
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre MutexVble.c o busque el archivo MutexVble.c en el directorio ./Guia3/Linux.

3. Compile el archivo MutexVble.c de la siguiente manera:

```
gcc –o MutexVble MutexVble.c –lpthread
```

4. Luego ejecútelo:

```
./MutexVble
```

Windows

Sección Crítica

1. El siguiente programa crea seis hilos, cada uno de los cuales incrementa 10 veces el valor de una variable. La variable es compartida, por lo tanto se debe sincronizar el acceso de los hilos a ésta para que su valor no difiera entre ejecuciones del programa. El mecanismo de sincronización que se utiliza es *Sección Crítica.*

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define numeroHilos 6
```

```
volatile INT c;
int a;
CRITICAL_SECTION SecCritica;
void HiloContador(INT iteraciones)
{
INT i;
INT x;
for (i=0; i<iteraciones; i++)
{
EnterCriticalSection(&SecCritica);
x=c;
x++;
c=x;
LeaveCriticalSection(&SecCritica);
}
}
void main(void)
{
HANDLE handles[numeroHilos];
DWORD hiloID;
INT i;
InitializeCriticalSection(&SecCritica);
for (i=0; i<numeroHilos; i++)
{
handles[i]=CreateThread(0, 0,
(LPTHREAD_START_ROUTINE) HiloContador,
(VOID *) 10, 0, &hiloID);
```

```
}
WaitForMultipleObjects(numeroHilos, handles, TRUE, INFINITE);
//Espera a que todas las hebras finalicen su ejecución
DeleteCriticalSection(&SecCritica);
printf("El contador vale %d \n", c);
a=getch();
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre SecCritica.cpp o busque el archivo SecCritica.cpp en el directorio C:\Guia3\Windows.

3. Compile el archivo:

SecCritica.cpp

4. Ejecute el archivo:

SecCritica.exe

Semáforo

1. El siguiente programa muestra la solución al problema *productor – consumidor* con buffer acotado utilizando Semáforo. El búfer es una variable, y los hilos productor y consumidor se alternan en su ejecución.

```
#include <stdio.h>
#include <windows.h>
```

```
#include <stdlib.h>
#include <conio.h>
int a;
int BuferCompartido;
HANDLE SEMAFORO;
void Productor() / /Función del productor
{
int i;
for (i=20; i>=0; i–) {
if (WaitForSingleObject(SEMAFORO,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: En productor\n");
ExitThread(0);
}
printf("Produce: %d \n", i);
BuferCompartido = i;
ReleaseSemaphore(SEMAFORO, 1, NULL); / /Fin de la sección
                                        / / critica
}
}
void Consumidor() / /Función del consumidor
{
int resultado;
while (1) {
if (WaitForSingleObject(SEMAFORO,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: consumidor()\n");
ExitThread(0);
}
```

```
if (BuferCompartido == 0) {
printf("Consumido %d: ultimo dato \ n",BuferCompartido);
ReleaseSemaphore(SEMAFORO, 1, NULL); / / Fin de sección critica
ExitThread(0);
}
if (BuferCompartido > 0){
resultado = BuferCompartido;
printf("Consumido: %d \ n", resultado);
ReleaseSemaphore(SEMAFORO, 1, NULL); / / Fin de sección critica
}
}
}
void main()
{
HANDLE hVectorHilos[2];
DWORD HebraID;
BuferCompartido = -1;
SEMAFORO = CreateSemaphore(0, 1, 2, "SemaforoContador");
hVectorHilos[0] = CreateThread (NULL, 0,
(LPTHREAD_START_ROUTINE)Productor,
NULL, 0, (LPDWORD)&HebraID);
hVectorHilos[1] = CreateThread (NULL, 0,
(LPTHREAD_START_ROUTINE)Consumidor,
NULL, 0, (LPDWORD)&HebraID);
WaitForMultipleObjects(2,hVectorHilos,TRUE,INFINITE);
a=getch();
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre productorconsumidor_semaforo.cpp o busque el archivo productorconsumidor_semaforo.cpp en el directorio C:\Guia3\Windows.

3. Compile el archivo:

productorconsumidor_semaforo.cpp

4. Ejecute el archivo:

productorconsumidor_semaforo.exe

Mútex

1. El siguiente programa muestra la solución al problema *productor – consumidor* con buffer acotado utilizando Mutex. El búfer es una variable, y los hilos productor y consumidor se alternan en su ejecución.

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <conio.h>
int a;
int BuferCompartido;
```

```
HANDLE hMutex;
void Productor() / /Función del productor
{
int i;
for (i=10; i>=0; i–) {
if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: En productor\ n");
ExitThread(0);
}
printf("Produce: %d \ n", i);
Sleep(100);
BuferCompartido = i;
ReleaseMutex(hMutex); / / fin de la sección critica
}
}
void Consumidor() / / Función del consumidor
{
int resultado;
while (1) {
if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
fprintf(stderr,"ERROR: consumidor() \ n");
ExitThread(0);
}
if (BuferCompartido == 0) {
printf("Consumido %d: ultimo dato \ n",BuferCompartido);
ReleaseMutex(hMutex); / /Fin de sección critica
ExitThread(0);
```

```
}
if (BuferCompartido > 0){
resultado = BuferCompartido;
printf("Consumido: %d \ n", resultado);
ReleaseMutex(hMutex); / / fin de sección critica
}
}
}
void main()
{
HANDLE hVectorHilos[2];
DWORD HebraID;
BuferCompartido = -1;
hMutex = CreateMutex(NULL,FALSE,NULL);
hVectorHilos[0] = CreateThread (NULL,
        0,(LPTHREAD_START_ROUTINE)Productor,
NULL, 0, (LPDWORD)&HebraID);
hVectorHilos[1] = CreateThread (NULL,
        0,(LPTHREAD_START_ROUTINE)Consumidor,
NULL, 0, (LPDWORD)&HebraID);
WaitForMultipleObjects(2,hVectorHilos,TRUE,INFINITE);
a=getch();
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre productorconsumidor_mutex.cpp o busque el archivo productorconsumidor_mutex.cpp en el directorio C:\Guia3\Windows.

3. Compile el archivo:

productorconsumidor_mutex.cpp

4. Ejecute el archivo:

productorconsumidor_ mutex.exe

ENUNCIADO DE LA PRÁCTICA

2. Elaborar cuatro programas en donde mediante la creación de hilos incremente n veces el valor de una variable. La variable es compartida, por lo tanto se debe sincronizar el acceso de los hilos a ésta para que su valor no difiera entre ejecuciones del programa. Los mecanismos de sincronización que debe utilizar son:

Linux	**Windows**
Semáforos	Semáforos
Mútex con Variables Condicionales	Mútex

DESARROLLO

LINUX

Semáforo

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
void * funcion1(void *);
void * funcion2(void *);
pthread_t hilo1,hilo2;
sem_t semaforo1, semaforo2;
int contador;
void * funcion1(void * parm)
{
  int i;
  for (i=0; i < 3; i++)
  {
  sleep(1);

printf("**************************************************************
*****\n");
printf("\nID Hilo 1: %d",pthread_self());
printf("\t\t\tBloqueo la variable\n");
```

```
sem_post(&semaforo2);
printf("\t\t\t\tContador: %d \n",contador);
contador=contador+2;
printf("\t\t\t\tEl nuevo valor de contador es:
%d \n",contador);
printf("\t\t\t\tDesbloqueo la variable \n\n");
sem_wait(&semaforo1);
}
}

void * funcion2(void * parm)
{
  int i;
  for (i=0; i < 3; i++)
  {
  sleep(1);
printf("**************************************************************
*****\n");
printf("\nID Hilo 2: %d",pthread_self());
printf("\t\t\tBloqueo la variable \n");
sem_post(&semaforo1);
printf("\t\t\t\tContador: %d \n",contador);
contador=contador+1;
printf("\t\t\t\tEl nuevo valor de contador es:
%d \n",contador);
printf("\t\t\t\tDesbloqueo la variable \n\n");
sem_wait(&semaforo2);
```

```
}
}
main( int argc, char *argv[] )
{
 contador=1;
 sem_init(&semaforo1, 0, 0);
 sem_init(&semaforo2, 0, 0);
  system("clear");
 pthread_create(&hilo1, NULL, funcion1, NULL);
 pthread_create(&hilo2, NULL, funcion2, NULL);
 pthread_join(hilo1, NULL);
 pthread_join(hilo2, NULL);
}
```

Mútex y Variables Condicionales

```
#include <pthread.h>
#include <stdio.h>
int     contador = 1;
pthread_mutex_t contador_mutex;
pthread_cond_t contador_vble;
void *funcion()
{
 int i;
 for (i=0; i < 2; i++) {
   pthread_mutex_lock(&contador_mutex);
```

```
   if (rand()% 4 >= 2)
    {
          pthread_cond_signal(&contador_vble);
          printf("\nID Hilo: %d",pthread_self());
          printf("\t\t Bloqueo la variable contador\n\n");
          printf("\t\t\t Contador: %d \n",contador);
          contador= contador+2;
          printf("\t\t\t El nuevo valor de contador es:
%d \n\n",contador);
          printf("\t\t\t Desbloqueo la variable\n\n");

          printf("**********************************************************
          ***************************** \n");
sleep(2);
     }
   contador++;
   printf("\nID Hilo: %d",pthread_self());
   printf("\t\t Incremento la variable contador");
   printf("\t\t Contador: %d\n\n",contador);
   printf("**********************************************************
   ************************** \n");
   pthread_mutex_unlock(&contador_mutex);
   }
  pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
```

```
  pthread_t hilo1,hilo2;
  pthread_attr_t attr;
  pthread_mutex_init(&contador_mutex, NULL);
  pthread_cond_init (&contador_vble, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
  system("clear");

printf("**************************************************************
********************** \ n");
  pthread_create(&hilo1, &attr, funcion, (void *)&hilo1);
  pthread_create(&hilo2, &attr, funcion, (void *)&hilo2);
  pthread_join(hilo1, NULL);
  pthread_join(hilo2, NULL);
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&contador_mutex);
  pthread_cond_destroy(&contador_vble);
  pthread_exit (NULL);
}
```

WINDOWS

Semáforo

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define numeroHebras 6
volatile INT contador;
int a;
void HebraContador(INT iteraciones)
{
INT i;
INT x;
HANDLE SEMAFORO;
SEMAFORO = CreateSemaphore(0, 1, 2, "SemaforoContador");
for (i=0; i<iteraciones; i++)
{
WaitForSingleObject(SEMAFORO, INFINITE);
x=contador;
x++;
contador=x;
ReleaseSemaphore(SEMAFORO, 1, NULL);
}
CloseHandle(SEMAFORO);
}
void main(void)
```

```
{
HANDLE handles[numeroHebras];
DWORD hebraID;
INT i;
for (i=0; i<numeroHebras; i++)
{
/* crea una hebra y pasa puntero a la estructura params */
handles[i]=CreateThread(0, 0,
(LPTHREAD_START_ROUTINE) HebraContador,
(VOID *) 10, 0, &hebraID);
}
/* Espera a que todas las hebras finalizen su ejecución */
WaitForMultipleObjects(numeroHebras, handles, TRUE,
INFINITE);
printf("El valor de contador es %d \n", contador);
a=getch();
}
```

Mútex

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define numeroHebras 8

volatile INT contador;
int a;
```

```
void HebraContador(INT iteraciones)
{
INT i;
INT x;
HANDLE MUTEX;
MUTEX = CreateMutex(0, FALSE, "MutexContador");
for (i=0; i<iteraciones; i++)
{
WaitForSingleObject(MUTEX, INFINITE);
x=contador;
x++;
contador=x;
ReleaseMutex(MUTEX);
}
CloseHandle(MUTEX);
}

void main(void)
{
HANDLE handles[numeroHebras];
DWORD hebraID;
INT i;
for (i=0; i<numeroHebras; i++)
{
/ / crea una hebra y pasa puntero a la estructura params
handles[i]=CreateThread(0, 0,
(LPTHREAD_START_ROUTINE) HebraContador,
```

```
(VOID *) 30000, 0, &hebraID);
}
/ / Espera a que todas las hebras finalizen su ejecución
WaitForMultipleObjects(numeroHebras, handles, TRUE, INFINITE);
printf("El valor de contador es %d \ n", contador);
a=getch();
}
```

2. Elaborar un programa que utilizando el mecanismo de *Sección Crítica*, sincronice el acceso a un determinado archivo ya sea en modo lectura o escritura. Recuerde incluir la librería <windows.h>

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define numeroHilos 10

FILE *fp;
char filename[40]= "6.txt";
int a, q, p;
CRITICAL_SECTION SecCritica;
void HiloLector(int iteraciones)
{
int i;
for (i=0; i<iteraciones; i++)
{
```

```
EnterCriticalSection(&SecCritica);
if ((fp = fopen(filename,"r")) != NULL)
     {
     q++;
  }
LeaveCriticalSection(&SecCritica);
}
}

void HiloEscritor(int iteraciones)
{
int i;
for (i=0; i<iteraciones; i++)
{
EnterCriticalSection(&SecCritica);
if ((fp = fopen(filename,"w")) != NULL)
     {
     p++;
  }
LeaveCriticalSection(&SecCritica);
}
}

void main(void)
{
HANDLE handles[numeroHilos];
DWORD hiloID;
```

```
DWORD hiloID1;
INT i;
INT x;
InitializeCriticalSection(&SecCritica);
for (i=0; i<numeroHilos; i=i+2)
    {
    handles[i]=CreateThread(0, 0,
    (LPTHREAD_START_ROUTINE) HiloLector,
            (VOID *)1, 0, &hiloID);
    }
for (x=1; x<numeroHilos; x=x+2)
    {
            handles[x]=CreateThread(0, 0,
            (LPTHREAD_START_ROUTINE) HiloEscritor,
            (VOID *)1, 0, &hiloID1);
    }
WaitForMultipleObjects(numeroHilos, handles, TRUE,
INFINITE);
DeleteCriticalSection(&SecCritica);
printf("\nEl numero de veces que se tuvo acceso al archivo %s en
modo lectura fue: %d \n", filename, q);
printf("\nEl numero de veces que se tuvo acceso al archivo %s
modo escritura fue: %d \n", filename, p);
a=getch();
}
```

REFERENCIAS

http://aula.linux.org.ar/docs/tecnicos/index/cap3.html

http://redes.ens.uabc.mx/docencia/computacion/cursos/SO/prog_c/6_6_POSIX_THREADS.htm

http://redes.ens.uabc.mx/docencia/computacion/cursos/SO/prog_c/6_7_OTROS_PAQUETES.htm

http://studies.ac.upc.es/FIB/ISO/ISO-Comunicacion.pdf

http://bari.ufps.edu.co/personal/150802A/paso_mensajes.htm

http://www.kennedy.edu.ar/computacion/Apuntes%20y%20Utilidades/apte s-utilidades.htm

http://www.monografias.com/trabajos7/arso/arso.shtml#dise

COMUNICACIÓN DE PROCESOS
GUÍA DEL DOCENTE

COMUNICACIÓN DE PROCESOS
GUÍA DEL DOCENTE

OBJETIVOS

- ✓ Identificar, analizar y mostrar cómo se utilizan los principales mecanismos que ofrecen los sistemas operativos Linux y Windows para la comunicación de procesos.

- ✓ Intercambiar información entre procesos diferentes empleando los mecanismos de tuberías, con y sin nombre, que proporciona el sistema operativo UNIX y WINDOWS.

MARCO TEÓRICO

El sistema operativo provee mecanismos para que los procesos se puedan intercomunicar. La intercomunicación se puede realizar utilizando almacenamiento compartido (se comparte algún medio de almacenamiento entre ambos procesos) o utilizando el sistema de intercambio de mensajes (se utilizan llamadas al sistema para enviar y recibir mensajes).

LINUX

El sistema operativo UNIX permite que procesos diferentes intercambien información entre ellos. Para procesos que se están ejecutando bajo el control de una misma máquina permite la comunicación mediante el uso de:

- ✓ Tuberías
- ✓ Memoria compartida
- ✓ Semáforos
- ✓ Colas de mensajes.

Mecanismos IPC (Inter process Communication - Comunicación entre Procesos)

Los medios IPC de Linux proporcionan un método para que múltiples procesos se comuniquen unos con otros. Hay varios métodos de IPC disponibles:

Tuberías (Pipes)

Una tubería *(pipe)* se puede considerar como un canal de comunicación entre dos procesos. Este mecanismo de comunicación consiste en la introducción de información en una tubería por parte de un proceso, posteriormente otro proceso extrae la información de la tubería de forma que los primeros datos que se introdujeron en ella son los primeros en salir.

Clasificación de tuberías

Se pueden distinguir dos tipos de tuberías dependiendo de las características de los procesos que pueden tener acceso a ellas:

- ✓ **Sin nombre**. Solamente pueden ser utilizadas por los procesos que las crean y por los descendientes de éstos. Un *pipe* en POSIX sólo puede ser utilizado entre los procesos que lo hereden a través de la llamada fork(). A

continuación se describen los servicios que permiten crear y acceder a los datos de un *pipe*:

Creación:

Las pipes se crean con la llamada a sistema **pipe**:

int **pipe**(int fd[2])

Abre dos canales de acceso a una pipe.

- ✓ fd[0] canal de sólo lectura,
- ✓ fd[1], canal de sólo escritura

Devuelve 0, si se ha completado correctamente; -1 en caso de error.

Cierre

El cierre de cada uno de los descriptores que devuelve la llamada *pipe* se consigue mediante el servicio close, que también se emplea para cerrar cualquier archivo. Su prototipo es:

int **close**(int fd);

El argumento de close indica el descriptor de archivo que se desea cerrar. La llamada devuelve 0 si se ejecutó con éxito. En caso de error, devuelve –1.

Escritura

El servicio para escribir datos en un *pipe* en POSIX es el siguiente:

int **write**(int fd, char *buffer, int n);

El primer argumento representa el descriptor de archivo que se emplea para escribir en un *pipe*. El segundo argumento especifica el *buffer* de usuario donde se encuentran los datos que se van a escribir al *pipe*. El último argumento indica el número de bytes a escribir. Los datos se escriben en el *pipe* en orden FIFO. La semántica de esta llamada es la siguiente:

⇒ Si la tubería se encuentra llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.

⇒ Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve el correspondiente error. Este error se genera mediante el envío al proceso que intenta escribir de la señal SIGPIPE.

⇒ Una operación de escritura sobre una tubería se realiza de forma **atómica**, es decir, si dos procesos intentan escribir de forma simultánea en una tubería sólo uno de ellos lo hará, el otro se bloqueará hasta que finalice la primera escritura.

Lectura

Para leer datos de un *pipe* se utiliza el siguiente servicio, también empleado para leer datos de un archivo.

int **read**(int fd, char *buffer, int n);

El primer argumento indica el descriptor de lectura del *pipe*. El segundo argumento especifica el *buffer* de usuario donde se van a situar los datos leídos del *pipe*. El último argumento indica el número de bytes que se desean leer del *pipe*. La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve –1. Las operaciones de lectura siguen la siguiente semántica:

⇒ Si la tubería está vacía, la llamada bloquea al proceso en la operación de lectura hasta que algún proceso escriba datos en la misma.

⇒ Si no hay escritores y la tubería está vacía, la operación devuelve fin de archivo (la llamada read devuelve cero). En este caso, la operación no bloquea al proceso.

⇒ Al igual que las escrituras, las operaciones de lectura sobre una tubería son atómicas.

✓ **FIFO (pipe con nombre)**. Se utilizan para comunicar procesos entre los que no existe ningún tipo de parentesco. El funcionamiento de una *fifo* es similar al de las tuberías sin nombre, pero se accede a ellas de igual manera que a los archivos de disco, es decir con la llamada al sistema *open*. Cualquier proceso que desee introducir información en ella ha de abrirla en modo de escritura, mientras que aquél que desee recibir su contenido debe abrirla en modo de lectura. La llamada al sistema para crear una *fifo* es *mkfifo*:

```
int mkfifo (char *ruta, mode_t modo);
```

El parámetro *ruta* es una cadena de caracteres que define la ruta y el nombre de la *fifo*, mientras que *modo* determina los permisos que se conceden a los diferentes usuarios utilizando valores numéricos pero expresados con cuatro cifras. La función devuelve el valor –1 en caso de error, en cuyo caso queda en *errno* el correspondiente código de error.

Llamadas al sistema

⇒ Open (char *name, int flag);

- Abre un FIFO (para lectura, escritura o ambas)
- Bloquea hasta que haya algún proceso en el otro extremo

⇒ Lectura y escritura mediante *read()* y *write()*
Igual semántica que los pipes

⇒ Cierre de un FIFO mediante *close()*

⇒ Borrado de un FIFO mediante *unlink()*

System V IPC

Se conocen como System V IPCs a tres técnicas de comunicación entre procesos que provee el UNIX System V:

⇒ *Memoria compartida:* Provee comunicación entre procesos permitiendo que éstos compartan zonas de memoria. La memoria compartida se puede describir mejor como el plano (mapping) de un área (segmento) de memoria que se combinará y compartirá por más de un de proceso. Esta es la forma más rápida de IPC (Inter process Communication - Comunicación entre Procesos), porque no hay intermediación (es decir, un tubo, una cola de mensaje, etc). En su lugar, la información se combina directamente en un segmento de memoria, y en el espacio de direcciones del proceso llamante. Un segmento puede ser creado por un proceso, y consecutivamente escrito y leído por cualquier número de procesos.

Llamadas al sistema:

- ✓ int shm_open(char *name, int oflag, mode_t mode): Crea un objeto de memoria a compartir entre procesos.
- ✓ int shm_open (char *name, int oflag): Sólo aperture.
- ✓ int close(int fd): Cierre. El objeto persiste hasta que es cerrado por el último proceso.
- ✓ int shm_unlink(const char *name): Borra una zona de memoria compartida.
- ✓ void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off): Establece una proyección entre el espacio de direcciones de un proceso y un descriptor de fichero u objeto de memoria compartida.
- ✓ void munmap(void *addr, size_t len): Desproyecta parte del espacio de direcciones de un proceso comenzando en la dirección ***addr***. El tamaño de la región a desproyectar es ***len***.

⇒ *Colas de mensajes:* Permiten tanto compartir información como sincronizar procesos. Un proceso envía un mensaje y otro lo recibe. El kernel se encarga de sincronizar la transmisión / recepción. Las colas de mensajes se pueden describir mejor como una lista enlazada interior, dentro del espacio de direccionamiento del núcleo. Los mensajes se pueden enviar a la cola en orden y recuperarlos de la cola en varias maneras diferentes. Cada cola de mensaje esta identificada de forma única por un identificador IPC.

Llamadas al sistema:

Msgget()

Para crear a una nueva cola de mensajes o acceder a una ya existente, usaremos la llamada al sistema msgget( ). Esta función nos devuelve el

identificador de la cola de mensajes ya existente y, si la cola no existe, permite crearla.

int msgget (key_t llave,int msgflag);

key_t es el valor clave.
msgflag es un mapa de bits, que representa:

- ✓ Para crear una cola: IPC_CREAT | OXYZ.
- ✓ Para saber el identificador de una cola ya existente: OXYZ.

Msgsnd() y Msgrcv()

Estas llamadas al sistema se utilizan para escribir y leer de una cola:

int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);

- ✓ msqid es el identificador de la cola.
- ✓ msgsz establece el tamaño del mensaje que se va a leer o escribir.
- ✓ El puntero msgp señala la zona de memoria donde se encuentran los datos a enviar o donde se almacenarán aquellos que van a ser leídos.
- ✓ El parámetro msgtyp sólo aparece en la llamada que permite la lectura de la cola y permite especificar qué mensaje de los que se encuentran en la cola queremos leer

  msgtyp =0 se accederá al primer mensaje que se encuentre en la cola independientemente de su tipo.

msgtyp >0 se accederá al primer mensaje del tipo msgtyp que se encuentre almacenado en la cola.
msgtyp <0 se accederá al primer mensaje cuyo tipo sea menor o igual al valor absoluto de msgtyp y a la vez sea el menor de todos.

El campo msgflg permite indicar si el proceso desea suspenderse en espera de que la llamada puede ejecutarse o no

Msgctl()

Esta función nos permite modificar la información de control que el Kernel le asocia a cada cola (identificada esta por su idmsg).

```
int msgctl (int msqid, int cmd, struct msqid_ds * buf);
```

El primer parámetro msqid que le pasamos a la función indica la cola que pretendemos controlar.

El segundo parámetro cmd permite seleccionar el tipo de operación que se va a realizar sobre la cola, puede tomar los siguientes valores:

- ✓ IPC_STAT: devuelve el estado de la estructura de control asociada a la cola en la zona de memoria apuntada por buf .
- ✓ IPC_SET: inicializa los campos de la estructura de control de la cola según el contenido de la zona de memoria apuntada por buf .
- ✓ IPC_RMID: elimina la cola de mensajes identificada por msqid. La cola no se borra mientras haya algún proceso que la esté utilizando.

El tercer parámetro buf será un puntero que apunta a una zona de memoria.

Es conveniente recordar que estos objetos IPC no se van a menos que se quiten apropiadamente, o el sistema se reinicie. Y el uso de la función msgctl( ) es la forma adecuada de deshacernos de una cola de mensajes que ya cumplió su cometido.

Señales

Una señal es una interrupción software enviada a un proceso. El sistema operativo emplea las señales para informar acerca de situaciones excepcionales a un proceso.

Cualquier proceso puede enviar a otro proceso una señal, siempre que tenga permiso, cuando un proceso se prepara para la recepción de una señal, puede realizar las siguientes acciones:

- ✓ Ignorar la señal.
- ✓ Realizar la acción asociada por defecto a la señal.
- ✓ Ejecutar una rutina del usuario asociada a dicha señal.

La recepción de una señal en particular por parte de un proceso provoca que se ejecute una subrutina encargada de atenderla. A esa rutina se le llama el "manejador" de la señal (signal handler). Un proceso puede definir un manejador diferente para sus señales o dejar que el kernel tome las acciones predeterminadas para cada señal. Cuando un proceso define un manejador para cierta señal se dice que "captura" (catch) esa señal.

Un proceso tiene ciertas libertades para configurar como reacciona frente a una señal (capturando, bloqueando o ignorando la señal), el kernel se reserva ciertos derechos sobre algunas señales. Así, las señales llamadas KILL y STOP no pueden ser capturadas, ni bloqueadas, ni ignoradas y la señal CONT no puede ser bloqueada.

Una señal puede enviarse desde un programa utilizando llamadas al sistema operativo, o desde la línea de comandos de un shell utilizando el comando kill. Al comando kill se le pasa como parámetro el número o nombre de la señal y el PID del proceso.

Llamada a sistema:

- ✓ *signal():* Asocia una acción determinada con una señal.
- ✓ *kill( )*: Envía una señal a un proceso
- ✓ *alarm( )*: Envía una señal de alarma en un período de tiempo especificado.
- ✓ *sigemptyset( )*: Crea un conjunto de señales vacío.
- ✓ *sigfillset( )*: Crea un conjunto de señales completo.
- ✓ *sigaddset( )*: Añade una señal al conjunto.
- ✓ *sigdelset( )*: Elimina una señal del conjunto.
- ✓ *sigprocmask( )*: Examina o modifica un conjunto de señales de proceso.
- ✓ *sigaction( )* Especifica la acción a realizar cuando un proceso recibe una señal.
- ✓ *sigsuspend( )*: Suspende un proceso hasta que reciba una señal del conjunto.
- ✓ *alarm(sec):* El kernel le enviará al proceso llamante una señal SIGALRM dentro de sec segundos.
- ✓ *pause( ):* El proceso queda bloqueado hasta que le llegue una señal.
- ✓ *time ( ):* el kernel consulta el reloj y devuelve su valor como retorno de la llamada.

Algunas señales

- ✓ SIGHUP Colgar. Generada al desconectar el terminal.
- ✓ SIGINT Interrupción. Generada por teclado.
- ✓ SIGILL Instrucción ilegal. No se puede capturar.
- ✓ SIGFPE Excepción aritmética, de coma flotante o división por cero.
- ✓ SIGKILL Matar proceso, no puede capturarse, ni ignorarse.
- ✓ SIGBUS Error en el bus.
- ✓ SIGSEGV Violación de segmentación.
- ✓ SIGPIPE Escritura en una pipe para la que no hay lectores.
- ✓ SIGALRM Alarma de reloj.
- ✓ SIGTERM Terminación del programa.

WINDOWS NT

Tuberías

Win32 ofrece dos tipos de tuberías: sin nombre y con nombre. Las primeras sólo permiten transferir datos entre procesos que hereden el pipe. Las tuberías con nombre tienen las siguientes propiedades:

- ✓ Las tuberías con nombre están orientadas a mensajes, de tal manera que un proceso puede leer mensajes de longitud variable, escritos por otro proceso.
- ✓ Son bidireccionales, así que dos procesos pueden intercambiar mensajes utilizando una misma tubería.
- ✓ Puede haber múltiples instancias independientes de la misma tubería. Todas las instancias comparten el mismo nombre, pero cada una de ellas tiene sus propios buffers y manejadores. El uso de instancias permite que múltiples clientes puedan utilizar la misma tubería con nombre de forma simultánea.
- ✓ Se pueden utilizar en sistemas conectados a una red, por lo que los hace especialmente útiles en aplicaciones cliente-servidor. En este tipo de aplicaciones un proceso (el servidor) crea una tubería con nombre y los procesos clientes se conectan a una instancia de esa tubería.

Las tuberías sin nombre tienen asociadas dos manejadores: uno para lectura y otro para escritura. Las tuberías con nombre únicamente tienen asociado un manejador que se puede utilizar para operaciones de lectura, escritura o ambas.

Los principales servicios que ofrece Win32 para trabajar con tuberías sin nombre son los siguientes:

Crear una tubería sin nombre

El prototipo es el siguiente:

```
BOOL CreatePipe ( PHANDLE phRead, PHANDLE phWrite,
LPSECURITY_ATTRIBUTES lpsa, DWORD cbPipe );
```

Los dos primeros argumentos representan dos manejadores de lectura y escritura respectivamente. El argumento lpsa indica los atributos de seguridad asociados a la tubería. El parámetro cbPipe indica el tamaño de la tubería. Si su valor es cero, se toma el valor por defecto. La llamada devuelve TRUE en caso de éxito y FALSE en caso contrario.

El atributo de seguridad viene dado por la estructura (LPSECURITY_ATTRIBUTES) con tres campos:

- ✓ nLength: especifica el tamaño de la estructura en bytes.

- ✓ lpSecurityDescriptor: puntero a un descriptor de seguridad que controla el acceso al objeto. Si es NULL, se utiliza el descriptor de seguridad asociado al proceso que realza la llamada.
- ✓ bInheritHandle: indica si el objeto puede ser heredado por los procesos que se creen. Si es TRUE, los procesos creados heredan el manejador.

Crear una tubería con nombre

La especificación de la función que permite crear una tubería con nombre en Win32 es la siguiente:

HANDLE CreateNamedPipe (LPCTSTR lpszPipeName, DWORD fdwOpenMode, DWORD fdwPipeMode, DWORD nMaxInstances, DWORD cbOutBuf, DWORD cbInBuf, DWORD dwTimeOut, LPSECURITY_ATTRIBUTES lpsa);

El parámetro lpszPipeName indica el nombre de la tubería. El nombre de la tubería debe tomar la siguiente forma:

\ \ . \ pipe \ [camino] nombre_tubería

El argumento fwOpenMode puede especificar alguno de los siguientes valores:

- ✓ PIPE_ACCESS_DUPLEX: permite que el manejador asociado a la tubería con nombre se pueda utilizar en operaciones de lectura y escritura.
- ✓ PIPE_ACCESS_INBOUND: sólo permite realizar operaciones de lectura de la tubería.
- ✓ PIPE_ACCESS_OUTBOUND: sólo permite realizar operaciones de escritura sobre la tubería.

El argumento fdwPipeMode tiene tres pares de valores mutuamente exclusivos. Estos valores indican si la escritura está orientada a mensajes o a bytes, si la lectura se realiza en mensajes o en bloques, y si las operaciones de lectura bloquean. Estos valores son:

- ✓ PIPE_TYPE_BYTE y PIPE_TYPE_MESSAGE: indican si los datos se escribe como un flujo de bytes o como mensajes.
- ✓ PIPE_READMODE_BYTE y PIPE_READMODE_MESSAGE: indican si los datos se leen como un flujo de bytes o como mensajes. PIPE_READMODE_MESSAGE requiere PIPE_TYPE_MESSAGE.
- ✓ PIPE_WAIT y PIPE_NOWAIT: indican si las operaciones de lectura sobre la tubería bloquearán al proceso lector en caso de que se encuentre vacía o no.

El parámetro nMaxInstances determina el número máximo de instancias de la tubería, es decir, el número máximo de clientes simultáneos. cbOutBuf y cbInBuf indican los tamaños en bytes de los buffers utilizados para la tubería. Un valor de cero representa los valores por defecto. dwTimeOut indica el tiempo de bloqueo, en milisegundos, de la función WaitNamedPipe. El último

argumento representa los atributos de seguridad asociados a la tubería. La función devuelve el manejador asociado a la tubería.

Abrir una tubería con nombre

La apertura de una tubería con nombre es una operación que típicamente realizan los clientes en aplicaciones de tipo cliente / servidor. Para abrir una tubería con nombre se recurre al servicio CreateFile, que se utiliza en Win32 para crear archivos. Su prototipo es el siguiente:

```
HANDLE CreateFile ( LPCSTR lpFileName, DWORD dwDesiredAccess,
WORD dwShareMode, LPVOID lpSecurityAttributes,  DWORD
CreationDisposition, DWORD dwFlagsAndAttributes,  HANDLE
hTemplateFile );
```

Su efecto es la creación o apertura de un archivo con nombre lpFileName. Este servicio se utiliza para abrir una tubería existente de la siguiente forma:

```
HandlePipe = CreateFile ( PipeName, GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
ILE_ATRIBUTE_NORMAL, NULL );
```

Con la llamada anterior se abrirá, para lectura y escritura, una tubería de nombre PipenName. La llamada fallará si la tubería con nombre todavía no se ha creado o todas sus instancias están ocupadas.

Relacionado con la apertura de la tubería con nombre se encuentra también el servicio WaitNamedPipe. Esta llamada bloquea a un proceso si todas las instancias de la tubería están ocupadas (normalmente por otros clientes). Su prototipo es el siguiente:

```
BOOL WaitNamedPipe ( LPCSTRPipeName, DWORD
dwTimeOut );
```

Si dwTimeOut es NULL, se utilizará el tiempo de expiración especificado en CreateNamedPipe. Los posibles valores para este parámetro son:

- ✓ NMPWAIT_NOWAIT: devuelve inmediatamente si el servidor no está listo para recibir mensajes.
- ✓ NMPWAIT_WAIT_FOREVER: bloquea el proceso cliente hasta que el servidor esté listo para recibir mensajes.
- ✓ NMPWAIT_USE_DEFAULT_WAIT: utiliza el tiempo por defecto especificado en CreateNamedPipe.

Cerrar una tubería con nombre

Este servicio cierra el manejador asociado a la tubería con o sin nombre. Un prototipo es:

```
BOOL CloseHandle ( HANDLE hfile );
```

La llamada decrementa el contador de instancias asociado al objeto. Si el contador se hace cero, la tubería se destruye. Devuelve TRUE en caso de éxito y FALSE en caso de error.

Leer de una tubería

Para leer de una tubería se utiliza el servicio que ofrece WIN32 para leer de archivos. El prototipo de este servicio es:

```
BOOL ReadFile ( HANDLE hFile, LPVOID lpBuffer, DWORD nBytes,
LPDWORD lpnBytes, LPOVERLAPPED lpOverlapped);
```

El parámetro hFile es el manejador asociado a la tubería que se utiliza para leer. lpBuffer especifica el buffer de usuario donde se van a situar los datos leídos de

la tubería. El argumento nBytes indica el número de bytes que se desean leer, en lpnBytes se almacena el número de datos realmente leídos. El último argumento es la estructura que se utiliza para indicar al posición de la cual se quiere leer, en el caso de tubería se utiliza NULL La función devuelve TRUE si se ejecutó con éxito o FALSE en caso contrario.

Escribir en una tubería

Al igual que con las lecturas, Win32 utiliza el mismo servicio que el empleado para escribir en archivos. El prototipo de esta función es:

```
BOOL WriteFile ( HANDLE hFile, LPVOID lpBuffer, DWORD nBytes,
LPDWORD lpnBytes, LPOVERLAPPED lpOverlapped );
```

El parámetro hFile es el manejador asociado a la tubería que se utiliza para escribir. lpBuffer especifica el buffer de usuario donde se encuentran los datos que se quieren escribir en la tubería. El argumento nBytes indica el número de bytes que se desean escribir. En lpnBytes se almacena el número de datos realmente escritos. El último argumento es una estructura que se utiliza para indicar la posición de la cual se quiere escribir, en el caso de tuberías se utiliza

NULL. La función devuelve TRUE si se ejecutó con éxito o FALSE en caso contrario.

DESARROLLO DE LA PRÁCTICA

Linux

Pipe (Tubería)

Procedimiento

1. El siguiente programa muestra como dos procesos padre e hijo pueden comunicarse a través de una tubería.

```
#include <unistd.h>
#include <sys/types.h>
#define TAMANOPIPE 4096

   int main(void)
   {
        int tubo[2]; /*Descriptor de la tubería*/
        int n;
        pid_t id;
        char cadena[TAMANOPIPE];
        char line[TAMANOPIPE];
        if (pipe(tubo)<0) /* Se crea la tubería sin nombre */
        {
             perror("Error al crear la tuberia \n");
             exit(-1);
}
id=fork(); /* Se crea un proceso hijo*/
if (id<0)
{
        perror("Error al crear el proceso hijo");
```

```
            exit(-2);
        }
        if (id>0) /*Proceso PADRE*/
        {
            printf("\nPADRE: Introduzca una cadena de caracteres:\n");
            n=read(STDIN_FILENO,cadena,TAMANOPIPE);
            close (tubo[0]); /*Cierra el extremo de lectura de la tubería */
            write(tubo[1],cadena,n); /*Escribe en la tubería */
            close (tubo[1]); /*Cierra el extremo de escritura de la tubería*/
            wait((int *)NULL); /*Espera a que acabe un proceso hijo*/
            exit(0);
        }
        else /*Proceso HIJO*/
        {
            close (tubo[1]); /*Cierra el extremo de escritura de la tubería */
            read(tubo[0],line,TAMANOPIPE); /* Lee de la tubería */
            printf("\nHIJO: He recibido la cadena: \n");
            sleep (1);
            printf("%s",line);
            close (tubo[0]); /*Cierra el extremo de lectura de la tubería*/
            printf("\n");
            exit(0);
        }
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre pipe.c o busque el archivo pipe.c en el directorio ./Guia2/Linux.

3. Compile el archivo pipe.c de la siguiente manera:

gcc –o pipe pipe.c

4. Luego ejecútelo:

./pipe

Colas

Procedimiento

1. El siguiente programa muestra la comunicación entre procesos mediante el uso de colas de mensajes.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define     BUFFER          1024
#define     PERMISOS   0666  //Permisos de lectura y ejecución para el dueño y otros

#define KEY ((key_t) 7777)
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
  main()
  {
        int i, msqid;
        char mensaje[BUFFER];
        struct
             {
          long  m_type;
          char  m_text[BUFFER];
       } msgbuffs, msgbuffr;
if ( (msqid = msgget(KEY, PERMISOS | IPC_CREAT)) < 0)  / /Llamada
la sistema para crear una cola
        perror("msgget error");
msgbuffs.m_type = 1L;
printf("\nEscriba el mensaje a enviar... \n");
scanf("%s",&mensaje);
strcpy(msgbuffs.m_text,mensaje);
if (msgsnd(msqid, &msgbuffs, BUFFER, 0) < 0)  / /Llamada al sistema
para escribir en la cola
                perror("msgsnd error");
printf("\nEl mensaje enviado es: %s \n", msgbuffs.m_text);
sleep(1);
if (msgrcv(msqid, &msgbuffr, BUFFER, 0L, 0) != BUFFER)  / /Llamada al
sistema para leer en la cola
                perror("msgrcv error");
printf("\nEl mensaje recibido es: %s \n\n", msgbuffr.m_text);
if (msgctl(msqid, IPC_RMID, (struct msqid_ds *) 0) < 0)    / /Llamada al
sistema para cerrar la cola
        perror("IPC_RMID error");
exit(0);
}
```

2. Copie el código fuente en un editor de texto, como por ejemplo gedit, kate, vi y guárdelo con el nombre colas.c o busque el archivo colas.c en el directorio ./Guia2/Linux.

3. Compile el archivo colas.c de la siguiente manera:

gcc –o colas colas.c

4. Luego ejecútelo:

./colas

Windows

Procedimiento

1. El siguiente programa ejecuta el mandato dir | sort .

```
#include <windows.h>
#include <stdarg.h>
#include <stdio.h>
#include <conio.h>
int main (void)
{
    HANDLE hRead, hWrite;
SECURITY_ATTRIBUTES Apipe = { sizeof(SECURITY_ATTRIBUTES),
NULL, TRUE };
    STARTUPINFO info;
    PROCESS_INFORMATION pid1,pid2;

     //inicialización de la estructura STARTUPINFO
    memset(&info,0,sizeof(info));
    info.cb = sizeof(info);

    if (!CreatePipe(&hRead,&hWrite,&Apipe,0)) //se crea el pipe
    {
        perror("Failed to create pipe");
        return -1;
    }

    //se prepara la redirección para el primer proceso
    info.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
```

```
    info.hStdOutput = hWrite;
    info.hStdError = GetStdHandle(STD_ERROR_HANDLE);
    info.dwFlags =
  STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;

  if (!CreateProcess(NULL,"cmd /c
  dir",NULL,NULL,TRUE,0,NULL,NULL,&info,&pid1))
    {
        perror("Failed to create process for \"dir\"");
        return -1;
    }

    CloseHandle(pid1.hThread);
    CloseHandle(hWrite);//cierra el pipe para escritura

    //redirige la salida estándar para el segundo proceso
    info.hStdInput = hRead;
    info.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    info.hStdError = GetStdHandle(STD_ERROR_HANDLE);
    info.dwFlags =
STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;

    if
(!CreateProcess(NULL,"sort",NULL,NULL,TRUE,0,NULL,NULL,&info,&
pid2))
    {
        perror("Failed to create process for \"sort\"");
        return -1;
    }
```

```
        CloseHandle(pid2.hThread);
        CloseHandle(hRead);

        se espera la terminación de los dos procesos
        WaitForSingleObject(pid1.hProcess,INFINITE);
        CloseHandle(pid1.hProcess);
        WaitForSingleObject(pid2.hProcess,INFINITE);
        CloseHandle(pid2.hProcess);

        getch();
        return(0);
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre Tuberia_1.cpp o busque el archivo Tuberia_1.cpp en el directorio C:\Guia4\Windows.

3. Compile el archivo: Tuberia_1.cpp

5. Ejecute el archivo: Tuberia_1.exe

ENUNCIADO

Linux

Realizar un programa utilizando pipes que utilice la salida de un comando del shell como la entrada de otro comando. Por ejemplo ps | wc.

```
#include <stdlib.h>
#include <unistd.h>
#define LEER                0
#define ESCRIBIR   1

int main (int parametro, char *args[])
  {
  int descr[2];         /* Descriptores de E y S de la tubería */
  if (parametro != 3)
    {
    printf ("Formato: %s comando_entrada comando_salida.\n", args[0]);
    exit (1);
    }
  pipe (descr);
  if (fork () == 0)
{
    close (descr[LEER]);
    dup2 (descr[ESCRIBIR], 1);
    close (descr[ESCRIBIR]);
    execlp (args[1], args[1], NULL);
    perror (args[0]);
    }
```

```
else
  {
  close (descr[ESCRIBIR]);
  dup2 (descr[LEER], 0);
  close (descr[LEER]);
  execlp (args[2], args[2], NULL);
  perror (args[0]);
  }
}
```

Windows

Realizar un programa utilizando tuberías sin nombre en el cual ponga en práctica la lectura y escritura en una tubería. En el programa principal puede mandar a escribir los datos en la tubería y crear un hilo que se encargue de leerlos.

```
include <windows.h>
#include <stdio.h>
#include <assert.h>
#include <conio.h>

int a;

void Hilo(HANDLE hRead)/ /hilo que lee los datos de la pipe
  {
  while(1)
    {
      char buf[100];
      DWORD dwRead;
```

```
        BOOL bRet;

        bRet=ReadFile(hRead,buf,100,&dwRead,NULL);
if(bRet==FALSE)
       {
              break;/ /  se cierra la tubería
        }
buf[dwRead]=0;
        printf("El hilo ha leido: %s \ n",buf);
        }
CloseHandle(hRead);
        ExitThread(0);
        }


        int main(void)
  {
        HANDLE hRead,hWrite;
        BOOL bRet;
        HANDLE hThread;
        DWORD dwThreadID;
        int i;

        bRet=CreatePipe(&hRead,&hWrite,NULL,40);/ / se crea la pipe
        assert(bRet==TRUE);

        hThread=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE
        )Hilo,(LPVOID)hRead,0,&dwThreadID);
        assert(hThread!=NULL);
```

```
for(i=1;i<=5;++i)    / / envía 5 lineas
        {
        char buf[100];
        int len;
        DWORD dwWrite;
        len=wsprintf(buf,"Esta es la linea %i \ n",i);
        Sleep(1000);
        bRet=WriteFile(hWrite,buf,len,&dwWrite,NULL);/ / se escribe en
la pipe
        assert(bRet!=NULL);
        }

        CloseHandle(hWrite);
        WaitForSingleObject(hThread,INFINITE);
        a=getch();
        return 0;
        }
```

REFERENCIAS

http://linux.dsi.internet2.edu/docs/LuCaS/Manuales-LuCAS/DENTRO-NUCLEO-LINUX/dentro-nucleo-linux-html/dentro-nucleo-linux-5.html

http://www.dc.uba.ar/people/materias/so/datos/cap24.pdf

http://aula.linux.org.ar/docs/tecnicos/index/cap2.html

http://labsopa.dis.ulpgc.es/prog_c/IPC.HTM

http://docencia.ac.upc.es/FIB/ISO/ISO-lab-P6.pdf

http://www.tlm.unavarra.es/~daniel/docencia/arq/arq1998-1999/llamadas.pdf

http://bari.ufps.edu.co/personal/150802A/comunicacion.htm

http://winapi.conclase.net/

http://winapi.conclase.net/curso/index.php?000

SISTEMAS OPERATIVOS, Una visión aplicada, Jesús Carretero Pérez, Féliz García Carballeira, Pedro De Miguel Anasagasti, Fernando Pérez Costoya, Editorial Mc Graw Hill

PROTECCIÓN

GUIA DEL DOCENTE

PROTECCIÓN
GUIA DEL DOCENTE

OBJETIVOS

- ✓ Identificar cuáles son los mecanismos que permiten implementar la política de protección adecuada en los sistemas operativos Linux y Windows.

- ✓ Establecer y modificar los permisos de acceso al sistema de archivos UNIX.

- ✓ Optimizar el acceso al sistema de archivos UNIX mediante el uso de caracteres especiales.

MARCO TEÓRICO

LINUX

El sistema UNIX es un sistema multiusuario, el cual tiene la capacidad de que todos los usuarios conectados a la máquina puedan leer y usar archivos de otros usuarios, debido a esto, los archivos deben ser protegidos contra cualquier uso indebido.

Protección de cuentas

En Unix cada usuario tiene asignada una cuenta individual. Cuando un usuario se desea conectar al sistema debe usar el nombre de conexión (login) y la autenticación del nombre de conexión (password). La protección de cuentas de usuarios es la primera línea de defensa contra los intrusos debido a que la forma más fácil por estos para ganar acceso es la consecución de un par login y password.

La información de la conexión de usuarios es almacenada en Unix en el archivo **/etc/passwd**, el cual contiene una línea por cuenta con la siguiente información:

<usuario>:<password>:<UID>:<GID>:<comentario>:<directorio home>:<Shell>

- ✓ Usuario: Es el nombre de la cuenta o login del usuario.
- ✓ Password: Puede contener una ***x*** o el password encriptado.
- ✓ UID: User ID (Identificador de usuario) único que tiene cada usuario para mostrar los dueños de los archivos.
- ✓ GID: Group ID (Identificador de grupo). Cada usuario tiene al menos un grupo de usuarios con similares privilegios. Este campo indica el Group ID primario.
- ✓ Comentario: Generalmente se usa para colocar la descripción o nombre completo del usuario.
- ✓ Directorio home: El directorio en el que es colocado el usuario cuando se conecta.
- ✓ Shell: El tipo de comando shell que usará el usuario cuando se conecte (***BourneShell, KornShell, Cshell***).

Shadow password

Es un método para mejorar la seguridad del sistema trasladando los passwords encriptados (encontrados normalmente en /etc/passwd) a **/etc/shadow** el cual puede ser leído únicamente por usuarios privilegiados, en cambio que /etc/passwd puede ser leído por todos los usuarios del sistema. Este archivo tiene la siguiente estructura:

usuario:password:last:may:must:warn:expire:disable:reserved

- ✓ *Usuario*: Es el nombre de la cuenta o *login.*
- ✓ *Password:* Indica el *password* que ha sido encriptado.
- ✓ *Last:* Número de días desde de enero 1 de 1970 que la contraseña fue cambiada por última vez.
- ✓ *May:* Número de días faltantes para que la contraseña pueda ser cambiada.
- ✓ *Warn:* Número de días faltantes para la expiración de la contraseña.
- ✓ *Expire:* Número de días desde que expiró la contraseña y que la cuenta ha sido deshabilitada.
- ✓ *Disable:* Número de días desde de enero 1 de 1970 que la cuenta de usuario ha sido deshabilitada.
- ✓ *Reserved:* Campo reservado.

Cabe anotar que el acceso a este archivo es restringido, ya que es peligroso que cualquier usuario vea o copie los ***passwords*** encriptados.

Cambio de password

Todos los usuarios en UNIX tienen asignado un *password* o contraseña. El *password* de cada usuario se cambia con el comando ***passwd*.** Este comando pregunta por el nuevo *password* (dos veces, para asegurarse que tecleó el *password* correcto). El usuario teclea el nuevo *password*, pero no se muestra en pantalla por razones de seguridad. El nuevo *password* surte efecto desde el momento en que se cambia. Si el usuario entra al sistema de nuevo, se le preguntaría por este nuevo *password* para permitir el acceso.

Protección de archivos

El sistema operativo UNIX provee un mecanismo llamado permisos de archivos, que permite a los archivos ser propiedad de un determinado usuario.

Tipos de archivos

En UNIX la estructura de directorios se encuentra clasificada de acuerdo a los tipos de archivos que en el sistema se encuentren, existe un carácter que define el tipo de archivo.

Socket: **s.**
Enlace simbólico: **l.**
Directorio: **d.**
Dispositivo de caracteres: **c.**
Dispositivo de bloque: **b.**

Archivo corriente: **-.**

Permisos otorgados al dueño, grupo y cualquier otro usuario del sistema.

Cada archivo en UNIX tiene asociado una serie de permisos que le permiten ser accesados o no por los usuarios. Los permisos de archivos caen dentro de tres grupos:

- ✓ Lectura (r): Permite a un usuario leer el contenido de un archivo, o si es un directorio, listar su contenido con ls.
- ✓ Escritura (w): Permite escribir y modificar un archivo. Para los directorios, este tipo de permiso permite crear nuevos archivos o borrar archivos dentro de ese directorio.
- ✓ Ejecución (x): Permite al usuario ejecutar un archivo. Para directorios, este permiso permite que el usuario pueda ubicarse en el directorio en cuestión usando el comando cd.

Existen tres clases de usuarios para los cuales se les asignan cada uno de estos permisos:

- ✓ El propietario del archivo.
- ✓ El grupo al cual pertenece el archivo
- ✓ Todos los usuarios, independientemente del grupo.

El comando ls con la opción -l muestra, entre otra información, los permisos de la siguiente manera:

-rwx r-x r--

- ✓ El primer carácter de la cadena de permisos ("-") representa el tipo de archivo.
- ✓ Las tres letras siguientes ("rwx"), representan los permisos otorgados al dueño del archivo. La "r" representa lectura, la "w" representa escritura, y la “x” representa ejecución.
- ✓ Los siguientes tres caracteres (“r-x”), representan los permisos del grupo en este archivo. Como solo aparece una r y una x, cualquier usuario que pertenezca al grupo en cuestión sólo podrá leer el archivo o ejecutarlo, pero no modificarlo o borrarlo.
- ✓ Los últimos tres caracteres (“r--“), representan los permisos que tienen cualquier otro usuario en el sistema diferente al dueño del archivo o aquellos que pertenezcan al grupo. En este caso, como solo aparece una r, otros usuarios podrán leer el archivo, pero no escribir en él o modificarlo.

Un aspecto fundamental a tener en cuenta es que los permisos de un archivo también dependen de los permisos del directorio en el que se encuentra el archivo. Es decir, que para poder acceder a un archivo, primero se debe tener acceso de ejecución a todos los directorios que se encuentren en el path del archivo, y por supuesto, se debe tener el permiso respectivo en ese archivo.

Configuración de los permisos de archivos

Cuando se crea un archivo en UNIX, estos se crean sin tener en cuenta los derechos que se les da. Como primera medida de seguridad, es importante dar los permisos adecuados a cada archivo. Estos permisos pueden ser modificados sólo por el propietario del archivo o el superusuario (root). UNIX provee la

utilidad **chmod** que nos permite alterar los permisos de un archivo. El comando **chmod** se usa de la siguiente forma:

- ✓ **chmod u+rwx programa.c** asigna los siguientes permisos al archivo:

  - rwx --- ---

- ✓ **chmod g+rw-x programa.c** asigna los siguientes permisos al archivo:

  - — rw- —

- ✓ **chmod o+r-wx programa.c** asigna los siguientes permisos al archivo:

  - — — r--

- ✓ **chmod a+rw-x programa.c** asigna los siguientes permisos al archivo:

  - rw- rw- rw-

Cualquier permiso puede ser añadido o suprimido mediante los símbolos "+" (signo más para añadir permisos) ó "-"(signo menos para suprimir permisos).

Los símbolos **u,g,o,a** permiten especificar la categoría del usuario:

- ✓ "u" para especificar la categoría de dueño o propietario.
- ✓ "g" para especificar la categoría del grupo al que pertenece.
- ✓ "o" para especificar a todos los usuarios.
- ✓ "a" para colocar los permisos en las tres categorías

Este tipo de modificaciones también se pueden hacer con un número en octal que varía entre 000 y 777.

Por ejemplo:

chmod 743 nombrearchivo

Asigna los siguientes permisos:

- rwx r-- -wx

El propietario y el grupo de un archivo son otras de las propiedades de un archivo o directorio que Linux permite que sean modificadas. Mediante la orden ***chown*** un propietario de un archivo puede transferir los privilegios que tiene sobre éste a otro usuario del sistema.

chown nombrenuevousuario nombredelarchivo

También es posible cambiar el grupo de un archivo empleando la orden ***chgrp.***

chgrp nombrenuevogrupo nombrearchivo

Permisos especiales

Existen dos tipos de permisos adicionales llamados permisos especiales que están disponibles para ejecutar archivos y directorios públicos. Cuando esos permisos son configurados, cualquier usuario puede ejecutar archivos, asumiendo los permisos del propietario para ejecutarlo.

Permiso setuid (set-user identification)

Cuando el **setuid** es configurado sobre un archivo ejecutable, cambia el **user id** de la persona que ejecuta el archivo por los del dueño del archivo. Esto permite a cualquier usuario ejecutar archivos y accesar a directorios que solo son accesados por el propietario, es decir, este permiso permite a un usuario normal ejecutar comandos a los cuales no tiene privilegios.

El permiso **setuid** se asigna a los archivos con el siguiente comando:

Chmod +s nombrearchivo

Permiso setgid (set-group identification)

El permiso **setgid** es similar a **setuid,** a excepción que el proceso identificación de grupo (GID), es cambiado al propietario del grupo del archivo, y un usuario tiene acceso basado sobre los permisos obtenidos para tal grupo. Cuando **setgid** es aplicado a un directorio, los archivos creados en ese directorio pertenecen al grupo que el directorio pertenezca, y no al grupo que pertenezca el proceso que lo creó. Cualquier usuario que tiene permiso de escritura en el directorio puede crear archivos ahí, sin embargo el archivo no pertenecerá al grupo del usuario, pero si pertenecerá al grupo del directorio.

El permiso **setgid** se asigna a los archivos con el siguiente comando:

Chmod +g nombrearchivo

Variable umask

La instrucción umask permite a un usuario definir una máscara de protección por defecto, acepta como parámetro un valor numérico (máscara) que indica los permisos que se negaran a todos los archivos creados a partir de ese instante.

umask [permisos_numéricos]

Cuando se utiliza sin parámetros, devuelve el valor actual de la máscara.
El valor asignado por umask es restado del que viene por defecto, esto tiene el efecto de negar permisos en la misma forma en que **chmod** los otorga.

Por ejemplo:

umask 022

Todos los archivos creados a partir de ese momento tendrán los siguientes permisos

rwx r-x r-x

Ya que

7	7	7	-	0	2	2	=	7	5	5
rwx	rwx	rwx		---	-w-	-w-		rwx	r-x	r-x

WINDOWS NT

Servicios de Win32

Windows NT tiene un nivel de seguridad C2 según la clasificación de seguridad del Orange Book del DoD, lo que significa la existencia de control de acceso discrecional, con la posibilidad de permitir o denegar derechos de acceso para cualquier objeto partiendo de la identidad del usuario que intenta acceder al objeto. Para implementar el modelo de seguridad, Windows NT usa un descriptor de seguridad y listas de control de acceso (ACL), que a su vez incluyen dos tipos de entradas de control de acceso (ACE): las de permisos y las de negaciones de accesos.

Las llamadas al sistema de Win32 permiten manipular la descripción de los usuarios, los descriptores de seguridad y las ACL. A continuación se describen las llamadas de Win32 más frecuentemente usadas.

Dar valores iniciales a un descriptor de seguridad

El servicio InitializeSecurityDescriptor inicia el descriptor de seguridad especificado en psd con los valores de protección por defecto.

BOOL InitializeSecurityDescriptor (PSECURITY_DESCRIPTOR psd,
DWORD dwRevision);

dwRevision tiene el valor de la constante *SECURITY_DESCRIPTOR_REVISIÓN*. Esta llamada devuelve *TRUE* si psd es un descriptor de seguridad correcto y *FALSE* en otro caso.

Obtención del identificador de usuario

La llamada *GetUserName* permite obtener el identificador de un usuario que ha accedido al sistema.

BOOL GetUserName (LPTSTR NombreUsuario, LPDWORD LongitudNombre);

En caso de éxito, devuelve el identificador solicitado en *NombreUsuario* y la longitud del nombre en *LongitudNombre*. Esta función siempre debe resolverse con éxito, por lo que no hay previsto caso de error.

Obtención de la información de seguridad de un archivo

El servicio *GetFileSecurity* extrae el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.

BOOL GetFileSecurity (LPCTSTR NombreArchivo, SECURIY_INFORMATION secinfo, PSECURITY_DESCRIPTOR psd, DWORD cbsd, LPDWORD lpcbLongitud);

El nombre del archivo se especifica en *NombreArchivo*. *secinfo* es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. El argumento *psd* es el descriptor de seguridad en el que se devuelve la información de seguridad.

Esta llamada devuelve *TRUE* si todo es correcto y *FALSE* en otro caso.

Cambio de la información de seguridad de un archivo

El servicio *SetFileSecurity* fija el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.

BOOL SetFileSecurity (LPCTSTR NombreArchivo, SECURIY_INFORMATION secinfo, PSECURITY_DESCRIPTOR psd);

el nombre del archivo se especifica en *NombreArchivo*. El argumento *secinfo* es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. El argumento *psd* es el descriptor de seguridad en el que se pasa la información de seguridad. Esta llamada devuelve *TRUE* si todo es correcto y *FALSE* en otro caso.

Obtención de los identificadores de propietario y de su grupo para un archivo

Las llamadas *GetSecurityDescriptorOwner* y *GetSecurityDescriptorGroup* permiten extraer la identificación del usuario de un descriptor de seguridad y del grupo al

que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo. Estas llamadas son sólo de consulta y no modifican nada.

BOOL GetSecurityDescriptorOwner (PSECURITY_DESCRIPTOR psd, PSID psidOwner);

BOOL GetSecurityDescriptorGroup (PSECURITY_DESCRIPTOR psd, PSID psidGroup);

El descriptor de seguridad se especifica en *psd*. El argumento *psidOwner* es un parámetro de salida donde se obtiene el identificador del usuario y *psidGroup* es un parámetro de salida donde se obtiene el grupo del usuario. En caso de éxito devuelve *TRUE* y si hay algún error *FALSE*.

Cambio de los identificadores del propietario y de su grupo para un archivo

Las llamadas SetSecurityDescriptorOwner y SetSecurityDescriptorGroup permiten modificar la identificación del usuario en un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo.

BOOL SetSecurityDescriptorOwner (PSECURITY_DESCRIPTOR psd, PSID psidOwner, BOOL fOwnerDefaulted);

BOOL SetSecurityDescriptorGroup (PSECURITY_DESCRIPTOR psd, PSID psidGroup, BOOL fGroupDefaulted);

El descriptor de seguridad se especifica en *psd*. El argumento *psidOwner* es un parámetro de entrada donde se indica el identificador del usuario y *psidGroup* es un parámetro de entrada donde se indica el grupo del usuario. El último parámetro de cada llamada especifica que se debe usar la información de protección por defecto para rellenar ambos campos. En caso de éxito devuelve *TRUE* y si hay algún error *FALSE*.

Gestión de ACLs y ACEs

Las llamadas *InitializeACL*, *AddAccessAllowedAce* y *AddAccessDeniedAce* permiten inicializar una ACL y añadir entradas de concesión y denegación de accesos.

BOOL InitializeACL (PACL pACL, DWORD cbAcl, DWORD dwAclRevision);

pAcl es la dirección de una estructura de usuario de longitud *cbAcl*. El último parámetro debe ser *ACL_REVISIÓN*.

La ACL se debe asociar a un descriptor de seguridad, lo que puede hacerse usando la llamada *SetSecurityDescriptorDacl*.

BOOL SetSecurityDescriptorDacl (PSECURITY_DESCRIPTOR psd, BOOL fDaclPresent, PACL pACL, BOOL fDaclDefaulted);

psd incluye el descriptor de seguridad. El argumento *fDaclPresent* a *TRUE* indica que hay una ACL válida en *pACL*. *fdaclDefaulted* a *TRUE* indica que se debe iniciar la ACL con valores por defecto.

AddAccessAllowedAce y *AddAccessDeniedAce* permiten añadir entradas con concesión y denegación de accesos a una ACL.

BOOL AddAccessAllowedAce (PACL pAcl, DWORD dwAclRevision, DWORD dwAccessMask, PSID pSid);

BOOL AddAccessDeniedAce (PACL pAcl, DWORD dwAclRevision, DWORD dwAccessMask, PSID pSid);

pAcl es la dirección de una estructura de tipo ACL, que debe estar iniciada.

El argumento *dwAclRevision* debe ser *ACL_REVISIÓN*. El argumento *pSid* apunta a un identificador válido de usuario. El parámetro *dwAccessMask* determina los derechos que se conceden o deniegan al usuario o a su grupo. Los valores por defecto varían según el tipo de objeto.

DESARROLLO DE LA PRÁCTICA

El siguiente programa lee los atributos de seguridad de un archivo

```
#include <windows.h>
#include <stdio.h>

PSECURITY_DESCRIPTOR LeerPermisosDeUnArchivo (LPCTSTR
NombreArchivo)
    /*Devuelve los permisos de un archivo*/
{
    PSECURITY_DESCRIPTOR pSD = NULL;
    DWORD Longitud;
    HANDLE ProcHeap = GetProcessHeap();
    /*Obtiene el tamaño del descriptor de seguridad*/
    GetFileSecurity (NombreArchivo,
OWNER_SECURITY_INFORMATION |
            GROUP_SECURITY_INFORMATION |
DACL_SECURITY_INFORMATION,
            NULL, 0, &Longitud);
    /*Obtiene el descriptor de seguridad del archivo*/
    pSD = HeapAlloc (ProcHeap, HEAP_GENERATE_EXCEPTIONS,
Longitud);
    if (!GetFileSecurity (NombreArchivo,
OWNER_SECURITY_INFORMATION |
```

```
GROUP_SECURITY_INFORMATION |
DACL_SECURITY_INFORMATION,
            pSD, Longitud, &Longitud))
    {
            printf ("Error GetFileSecurity \n");
            return NULL;
    }
    return pSD;
}

int main(int argc, char* argv[])
{
    for (int i = 1; i < argc; i++)
    {
            PSECURITY_DESCRIPTOR pSD =
LeerPermisosDeUnArchivo(argv[i]);
            PSID pOwnerSID;
            BOOL bDefaulted;
            if (GetSecurityDescriptorOwner(pSD, &pOwnerSID,
&bDefaulted))
            {
                    if (pOwnerSID)
                    {
                            SID_NAME_USE eUse;
                            TCHAR szName[100];
                            DWORD lNameLen = 100;
                            TCHAR szDomainName[100];
                            DWORD lDomainNameLen = 100;
```

```
                        if (LookupAccountSid(NULL, pOwnerSID,
                        szName, &lNameLen,
                                szDomainName, &lDomainNameLen,
        &eUse))
                        {
                                printf("%s:\t%s\\%s\n", argv[i],
                                szDomainName, szName);
                        }
                }
        }
        return 0;
}
```

2. Copie el código fuente en un editor de texto y guárdelo con el nombre proteccion.cpp o busque el archivo proteccion.cpp en el directorio ./Guia5/Windows.

3. Compile el archivo proteccion.cpp

4. Desde MS-DOS, escriba el nombre del programa y pásele como parámetro el nombre del archivo al cual le quiere averiguar los atributos de seguridad. Por ejemplo:

proteccion proteccion.cpp

ENUNCIADO DE LA PRÁCTICA

Linux

- ✓ Active el bit SETUID para que todos los usuarios puedan ver cualquier archivo del sistema utilizando el editor vi.
- ✓ ¿Qué tipos de accesos permite el fichero /etc/passwd? ¿Y /etc/shadow? ¿Quién es el propietario de estos archivos?
- ✓ Copie el archivo /etc/passwd a su directorio y asigne a su copia los siguientes derechos de acceso:

  El propietario puede leer, escribir y ejecutar el archivo.

  Todos los usuarios del grupo pueden leerlo y ejecutarlo, pero no escribir en él.

  El resto de usuarios solamente pueden ejecutarlo.

  Compruebe que el archivo tiene los permisos de acceso deseados.
- ✓ Utilice umask para que ningún otro usuario distinto del propietario pueda acceder a los archivos que se creen a partir de este momento. Compruebe que la máscara ha quedado actualizada.

Windows

Elabore un programa en el que utilizando las llamadas al sistema de Win32 que permiten manipular los descriptores de seguridad y las ACL, cree un archivo el cual solo pueda ser accedido en modo lectura por un usuario que usted determine.

DESARROLLO

Linux

- ✓ Active el bit SETUID para que todos los usuarios puedan ver cualquier archivo del sistema utilizando el editor vi.

  # chmod +s /bin/vi

- ✓ ¿Qué tipos de accesos permite el fichero /etc/passwd? ¿Y /etc/shadow? ¿Quién es el propietario de estos archivos?

  Passwd: solamente permite lectura para todos los tipos de usuario.

  Shadow: Solamente permite acceso de lectura al propietario.

  El propietario de ambos archivos es el usuario root (administrador).

- ✓ Copie el archivo /etc/passwd a su directorio y asigne a su copia los siguientes derechos de acceso:

  El propietario puede leer, escribir y ejecutar el archivo.

  Todos los usuarios del grupo pueden leerlo y ejecutarlo, pero no escribir en él.

  El resto de usuarios solamente pueden ejecutarlo.

  Compruebe que el archivo tiene los permisos de acceso deseados.

  ```
  cp /etc/passwd /$HOME
  chmod 751 /$HOME/passwd
  ls –l passwd
  ```

- ✓ Utilice umask para que ningún otro usuario distinto del propietario pueda acceder a los archivos que se creen a partir de este momento. Compruebe que la máscara ha quedado actualizada.

  umask 077

Windows

Elabore un programa en el que utilizando las llamadas al sistema de Win32 que permiten manipular los descriptores de seguridad y las ACL, cree un archivo el cual solo pueda ser accedido en modo lectura por un usuario que usted determine.

```
#include <windows.h>
#include <iostream.h>

int a;
SECURITY_ATTRIBUTES sa;
SECURITY_DESCRIPTOR sd;
BYTE aclBuffer[1024];
PACL pacl=(PACL)&aclBuffer;
BYTE sidBuffer[100];
PSID psid=(PSID) &sidBuffer;
DWORD sidBufferSize = 100;
char domainBuffer[80];
```

```
DWORD domainBufferSize = 80;
SID_NAME_USE snu;
HANDLE archivo;

void main(void)
{
  /* Inicializa el descriptor de seguridad */
  InitializeSecurityDescriptor(&sd,
  SECURITY_DESCRIPTOR_REVISION);

  /* Inicializa la ACL */
  InitializeAcl(pacl, 1024, ACL_REVISION);

  LookupAccountName(0, "usuario", psid,
  &sidBufferSize, domainBuffer,
  &domainBufferSize, &snu);

  /* Crea la ACE(Entradas de Control de Acceso) con el SID (Identificador
de Seguridad) */
  AddAccessAllowedAce(pacl, ACL_REVISION,
  GENERIC_READ, psid);

  /* Se incluye la ACL en el descriptor de seguridad */
  SetSecurityDescriptorDacl(&sd, TRUE, pacl,
  FALSE);

  sa.nLength= sizeof(SECURITY_ATTRIBUTES);
  sa.bInheritHandle = FALSE;
```

```
  sa.lpSecurityDescriptor = &sd;

  archivo = CreateFile("c:\\protección.txt",
  GENERIC_READ | GENERIC_WRITE,
  0, &sa, CREATE_NEW,
  FILE_ATTRIBUTE_NORMAL, 0);
  CloseHandle(archivo);
}
```

REFERENCIAS

http://shannon.unileon.es/~dielpa/Aero/Aero_Pract_3_sol.PDF

http://winapi.conclase.net/curso/index.php?000

http://www.sis.ucm.es/SIS/UNIX/indice.htm

http://www2.ing.puc.cl/~iic10622/clases/unix.htm

http://bernia.disca.upv.es/~iripoll/seguridad/03-admin/03-admin.pdf

http://www.iti.upv.es/~pgaldam/sso/transparencias/pdf/tema2.pdf

SISTEMAS OPERATIVOS, Una visión aplicada, Jesús Carretero Pérez, Féliz García Carballeira, Pedro De Miguel Anasagasti, Fernando Pérez Costoya, Editorial Mc Graw Hill.