Un vistazo a los Sistemas Operativos

Gabriel Gerónimo C.

BORRADOR VERSIÓN 2.0 ENERO 2026.

Universidad Tecnológica de la Mixteca
Instituto de Computación
gcgero@mixteco.utm.mx

Prólogo

Las presentes notas se basan en información recopilada de diversos libros de carácter introductorio, comúnmente recomendados a estudiantes de nivel licenciatura en la asignatura de Sistemas Operativos, así como en artículos y manuales de programación utilizados como material de apoyo para ampliar y complementar algunos de los temas abordados. Asimismo, se incluyen aportaciones y adaptaciones a códigos básicos desarrollados en la bibliografía fundamental, con la finalidad de facilitar la comprensión de ciertos conceptos

Para el estudio de los temas expuestos se recomienda la exploración del sistema operativo GNU/Linux. La distribución utilizada queda a elección del lector; no obstante, se hace notar que los programas en lenguaje C y los comandos presentados a lo largo de estas notas fueron probados y ejecutados en una distribución Ubuntu

 La presente versión corresponde a la 1.0; si bien aún existen aspectos por detallar y temas por actualizar, se espera que el material resulte de utilidad como apoyo para el estudio y la práctica.

Hay golpes en la vida, tan fuertes... Yo no sé!
Golpes como del odio de Dios; como si ante ellos,
la resaca de todo lo sufrido
se empozara en el alma... Yo no sé!
Los heraldos negros. César Vallejo.

Universidad Tecnológica de la Mixteca
Instituto de Computación
M.C. Gabriel Gerónimo C.
gcgero@mixteco.utm.mx

Contenido

Capítulo 1

Introducción a los Sistemas Operativos

Un sistema operativo es un conjunto de programas encargado de administrar los recursos de un dispositivo de cómputo. Proporciona, por un lado, una interfaz que permite la interacción del usuario con el sistema y, por otro, un núcleo (*kernel*) responsable de la comunicación directa con el hardware

1.1 Introducción

El sistema operativo (SO) es el software fundamental de todo dispositivo de cómputo, ya que constituye un elemento indispensable para que el hardware pueda ser utilizado de manera efectiva por el usuario. Ante la pregunta *¿qué es un sistema operativo?*, pueden plantearse, entre otras, las siguientes definiciones:

1. Un sistema operativo es el software que controla el hardware.
2. Un sistema operativo es un programa que controla la ejecución de otros programas y actúa como interfaz entre el usuario de una computadora y su hardware; sus funciones principales son la comodidad, la eficiencia y la capacidad de evolución (W.Stallings, pág.47).
3. Un sistema operativo es un administrador de los recursos del dispositivo, tales como procesadores, medios de almacenamiento, dispositivos de entrada/salida, medios de comunicación y datos; además, proporciona una interfaz para la interacción con el usuario.

Los sistemas operativos se encargan de la administración de procesos, memoria, seguridad, recursos y archivos. Los procesos representan las entidades que se encuentran en ejecución dentro del sistema, y su correcta gestión debe evitar conflictos entre ellos. Entre los problemas más comunes se encuentran la sincronización incorrecta, la falta de exclusión mutua y el interbloqueo.

En cuanto a la administración de memoria, el sistema operativo debe aislar los procesos, asignar y gestionar los espacios de memoria, así como crear, destruir y modificar dinámicamente el tamaño de los módulos. Adicionalmente, debe proteger y controlar el acceso a las distintas secciones de memoria.

Para la gestión de la seguridad, el sistema operativo puede implementar diversas políticas, tales como la no compartición, la compartición controlada de programas o archivos de datos, los subsistemas confinados, la diseminación controlada de la información, el control de acceso, el control del flujo de información y la certificación. Finalmente, en la administración de los recursos, el sistema debe considerar criterios como la equidad, la sensibilidad y la eficiencia.

1.2 Clasificación de los sistemas operativos

Con el paso del tiempo, los sistemas operativos han sido clasificados de acuerdo con su uso, su ámbito de aplicación y la arquitectura del hardware sobre el que operan. Entre las clasificaciones más comunes se encuentran los sistemas utilizados en computadoras de propósito general, ya sea con uno o varios procesadores, así como aquellos diseñados para dispositivos móviles. En las siguientes secciones se describen las principales características de cada uno de estos tipos de sistemas operativos.

1.2.1 Sistemas operativos por lotes

Los sistemas operativos por lotes procesan grandes cantidades de trabajos con poca o ninguna interacción entre los usuarios y los programas en ejecución. En este enfoque, los trabajos con características similares se agrupan y se ejecutan de manera conjunta, lo que permite reducir los tiempos muertos del procesador en comparación con el procesamiento estrictamente en serie. Este tipo de sistemas operativos se consideran entre los más tradicionales y antiguos, y fueron introducidos alrededor de 1956 con el objetivo de incrementar la capacidad de procesamiento de los sistemas de cómputo. Cuando están adecuadamente diseñados, pueden ofrecer un alto aprovechamiento del procesador, ya que la ejecución secuencial de los trabajos simplifica la planificación y minimiza la inactividad de la CPU. Entre los ejemplos más representativos de sistemas operativos por lotes se encuentran SCOPE, desarrollado para el DC6600 y orientado al procesamiento científico intensivo, así como EXEC II para el UNIVAC 1107, enfocado principalmente al procesamiento académico.

1.2.2 Sistemas operativos de tiempo real

Los sistemas operativos de tiempo real son aquellos en los que la prioridad no es la interacción con el usuario, sino la ejecución correcta y oportuna de los procesos, cumpliendo restricciones temporales estrictas. En este tipo de sistemas, el tiempo de respuesta es un factor crítico, por lo que se busca un comportamiento determinista. Por lo general, los recursos del sistema se utilizan de manera conservadora, con el objetivo de garantizar que los procesos puedan ser atendidos en el momento exacto en que lo requieran. Estos sistemas se emplean en entornos donde se procesan grandes cantidades de sucesos o eventos y donde un retraso puede tener consecuencias significativas. Muchos sistemas operativos de tiempo real están diseñados para aplicaciones específicas, tales como el control de tráfico aéreo, los sistemas bursátiles, el control de refinerías, los laminadores industriales, así como aplicaciones en el sector automotriz y de la electrónica de consumo. Otros campos de aplicación incluyen el control ferroviario, las telecomunicaciones, los sistemas de fabricación integrada, la producción y distribución de energía eléctrica, el control de edificios y los sistemas multimedia. Algunos ejemplos representativos de sistemas operativos de tiempo real son VxWorks, FreeRTOS, LynxOS, QNX y RTLinux.

1.2.3 Sistemas operativos de multitarea

Los sistemas operativos de multiprogramación o multitarea se caracterizan por su capacidad para soportar la ejecución concurrente de dos o más tareas activas. Esto permite que la Unidad Central de Procesamiento (CPU) tenga siempre algún trabajo que ejecutar, incrementando así su grado de utilización. El principio

fundamental de la multitarea consiste en mantener varias tareas en la memoria principal y alternar su ejecución mediante mecanismos de planificación y conmutación de contexto. Este enfoque no requiere necesariamente la existencia de múltiples procesadores, aunque puede combinarse con arquitecturas multiprocesador para mejorar el rendimiento. En la actualidad, prácticamente todos los sistemas operativos de propósito general soportan multitarea, entre ellos UNIX, macOS, GNU/Linux y Windows, por mencionar algunos de los más representativos.

La multitarea puede implementarse tanto a nivel de procesos como a nivel de hilos de ejecución (*threads*). Un proceso representa una unidad de ejecución con su propio espacio de direcciones, mientras que los hilos son flujos de ejecución más ligeros que comparten los recursos del proceso al que pertenecen. El uso de hilos permite una mayor concurrencia dentro de una misma aplicación, ya que múltiples hilos pueden ejecutarse de forma concurrente, compartiendo memoria y recursos. Los sistemas operativos actuales gestionan la planificación tanto de procesos como de hilos, lo que mejora la eficiencia y reduce el costo asociado a los cambios de ejecución.

1.2.4 Sistemas operativos distribuidos

Los sistemas operativos distribuidos permiten la ejecución y distribución de trabajos, tareas o procesos entre un conjunto de procesadores. Dichos procesadores pueden encontrarse en un solo equipo o estar físicamente distribuidos en múltiples sistemas interconectados; en ambos casos, esta distribución resulta transparente para el usuario, quien percibe el sistema como una única entidad de cómputo.
Existen dos esquemas básicos de organización:
a) Sistemas fuertemente acoplados, en los cuales los procesadores comparten la memoria principal y un reloj global, de modo que los tiempos de acceso a los recursos son similares para todos ellos.
b) Sistemas débilmente acoplados, en los que los procesadores no comparten ni memoria ni reloj, ya que cada uno dispone de su propia memoria local y se comunica con los demás a través de mecanismos de interconexión.

Existen dos esquemas básicos: a) un sistema fuertemente acoplado que comparte la memoria y un reloj global, cuyos tiempos de acceso son similares para todos los procesadores, b) un sistema débilmente acoplado en el cual los procesadores no comparten ni memoria ni reloj, ya que cada uno cuenta con su memoria local.
Una característica fundamental de los sistemas operativos distribuidos es su alto grado de confiabilidad. Estos sistemas deben ser capaces de tolerar fallos, de manera que, si uno de sus componentes deja de funcionar, otro pueda asumir sus funciones sin afectar significativamente el servicio ofrecido. Entre los sistemas operativos distribuidos más representativos se encuentran Sprite, Solaris-MC, Mach, Chorus, Spring, Amoeba y Taos.

1.2.5 Sistemas operativos de red

Los sistemas operativos de red son aquellos diseñados para mantener interconectadas dos o más computadoras a través de un medio de comunicación, ya sea alámbrico o inalámbrico, con el objetivo principal de permitir el intercambio de información y el uso compartido de recursos. A diferencia de los sistemas operativos distribuidos, en los sistemas operativos de red cada computadora conserva su propia autonomía, mientras que la red se utiliza como un mecanismo para compartir

recursos tales como archivos, impresoras, servicios y aplicaciones. Entre los ejemplos más representativos de sistemas operativos de red se encuentran *Windows Server,* ampliamente utilizado en entornos empresariales para la administración centralizada de usuarios y recursos; sistemas *UNIX y GNU/Linux* configurados como servidores de red mediante servicios como NFS, Samba y FTP; y *Novell NetWare,* uno de los sistemas de red más utilizados en entornos corporativos durante las décadas pasadas. Asimismo, plataformas modernas basadas en *GNU/Linux* son ampliamente empleadas como servidores web, de archivos y de bases de datos en infraestructuras de red.

1.2.6 Sistemas operativos paralelos

Los sistemas operativos paralelos están diseñados para explotar arquitecturas con múltiples procesadores o múltiples núcleos, permitiendo que dos o más procesos o hilos se ejecuten de manera simultánea. Su objetivo principal es incrementar el rendimiento y la capacidad de procesamiento del sistema, especialmente cuando varios procesos requieren acceso a los mismos recursos. En estos sistemas, el paralelismo real se logra mediante la ejecución simultánea de tareas en distintos procesadores, lo que exige mecanismos eficientes de sincronización y control de acceso a los recursos compartidos. Por otra parte, en sistemas como UNIX existe la posibilidad de ejecutar programas sin interacción directa con el usuario, por ejemplo, mediante la ejecución en segundo plano. Este mecanismo permite atender de forma concurrente varios procesos de un mismo usuario, dando la apariencia de paralelismo aun en sistemas con un solo procesador. En lugar de esperar a que un proceso termine su ejecución, el sistema devuelve el control al usuario inmediatamente después de crear el proceso, permitiendo la ejecución concurrente de múltiples tareas.

1.2.7 Sistemas operativos para dispositivos móviles

El gran auge de los dispositivos móviles ha impulsado una rápida evolución de los sistemas operativos diseñados para este tipo de plataformas. Estos sistemas se caracterizan por optimizar el uso de recursos limitados, ofrecer interfaces orientadas al usuario final y soportar una amplia variedad de aplicaciones y servicios. Entre los sistemas operativos móviles más conocidos se encuentran, o se encontraron en su momento —algunos de los cuales han desaparecido del mercado—, Android, iPhone OS (iOS), BlackBerry OS, Symbian, Palm OS, Windows Mobile y Ubuntu Touch. A continuación se describen las principales características de algunos de estos sistemas operativos.

Symbian

Symbian era un SO que estaba incorporado en los celulares Nokia, su core era común a todos los dispositivos con Symbian, contaba con: kernel, servidor de archivos, administración de memoria, y manejadores de drives. Contaba con una capa de sistema de comunicación como: TCP/IP, IMAP4, SMS, un administrador de base de datos, interfaz con el usuario y un conjunto de aplicaciones. En Symbian el kernel se ejecutaba en un espacio de dirección protegido al igual que cada proceso, por lo que permitia que varios procesos fueran ejecutados en forma concurrente. Symbian estaba escrito en C++, así que era natural el desarrollo de aplicaciones en este lenguaje, aunque también admitia el desarrollo en Java. El sistema era multitarea, y contaba con un hilo principal, el cual se encargaba de desprender nuevos hilos para las diferentes actividades a realizar.

Symbian fue un sistema operativo ampliamente utilizado en teléfonos móviles, principalmente en dispositivos de la marca Nokia. Su núcleo (*core*) era común a todos los dispositivos que lo implementaban y estaba compuesto por componentes fundamentales como el *kernel*, el servidor de archivos, los mecanismos de administración de memoria y los controladores de dispositivos. El sistema contaba además con una capa de comunicaciones que incluía protocolos y servicios como TCP/IP, IMAP4 y SMS, así como un administrador de bases de datos, una interfaz de usuario y un conjunto de aplicaciones integradas. En Symbian, el *kernel* se ejecutaba en un espacio de direcciones protegido, al igual que cada proceso, lo que permitía la ejecución concurrente de múltiples procesos de forma segura. Symbian estaba desarrollado principalmente en C++, lo que facilitaba el desarrollo de aplicaciones en este lenguaje; adicionalmente, admitía el desarrollo de aplicaciones en Java. El sistema operativo era multitarea y utilizaba un modelo basado en hilos, en el cual un hilo principal podía crear y gestionar nuevos hilos para atender las distintas actividades del sistema y de las aplicaciones.
A pesar de su amplia adopción y madurez técnica, Symbian comenzó a perder relevancia a finales de la década de 2000. La aparición de nuevas plataformas como iOS y Android introdujo modelos de desarrollo más sencillos, interfaces táctiles más avanzadas y ecosistemas de aplicaciones más atractivos para desarrolladores y usuarios. En contraste, Symbian presentaba una mayor complejidad en el desarrollo de aplicaciones y una evolución más lenta frente a las nuevas demandas del mercado. La decisión estratégica de Nokia de adoptar Windows Phone como su plataforma principal, junto con la disminución del soporte por parte de fabricantes y desarrolladores, aceleró el declive de Symbian. Finalmente, el sistema operativo fue descontinuado, marcando el fin de una plataforma que, durante varios años, dominó el mercado de los teléfonos inteligentes y sentó bases importantes en la evolución de los sistemas operativos móviles.

iPhone OS (iOS)

iPhone OS, actualmente conocido como iOS, fue desarrollado a partir de Mac OS X para funcionar como el sistema operativo nativo de los teléfonos inteligentes iPhone, los dispositivos táctiles iPod touch y las tabletas iPad. Este sistema operativo se caracteriza por una arquitectura en capas, diseñada para proporcionar seguridad, eficiencia y facilidad de desarrollo. La arquitectura del modelo de iOS está conformada por los siguientes elementos:

- *Capa de aplicaciones,* integrada por las aplicaciones nativas del sistema, como Calendario, Fotos y Cámara, así como por las aplicaciones instaladas por el usuario y desarrolladas por terceros.
- *Capa de middleware*, compuesta por tres subcapas principales:
    - *Cocoa Touch*, un conjunto de frameworks que proporciona la infraestructura fundamental para el desarrollo de aplicaciones, incluyendo el manejo de eventos táctiles, interfaces gráficas y servicios del sistema.
    - *Media*, que incluye frameworks gráficos y de audio y video, permitiendo la creación de aplicaciones multimedia.
    - *Core Services*, que ofrece servicios básicos utilizados directa o indirectamente por todas las aplicaciones, tales como Address Book, Core Data, la biblioteca SQLite y Core Location.

- *Kernel*, basado en el núcleo XNU, que integra componentes esenciales como los controladores de dispositivos, los mecanismos de seguridad y servicios de red como CFNetwork.

BlackBerry

El SO BlackBerry era una plataforma de software desarrollada por Research In Motion (RIM) para sus dispositivos smartphone liberados en 1999. El SO contaba con las siguientes características: multitareas, kernel basado en Java que utilizaba una arquitectura ARM, administrador de memoria dividida en memoria para aplicaciones, memoria del dispositivo y memoria de la tarjeta (opcional). iOS fue presentado por primera vez en 2007 junto con el lanzamiento del primer iPhone, marcando un cambio significativo en la industria de los dispositivos móviles. Desde sus primeras versiones, el sistema operativo introdujo un modelo de interacción basado en pantallas táctiles capacitivas y una interfaz orientada completamente al usuario final, lo que sentó nuevas bases para el diseño de sistemas móviles.

Con el paso del tiempo, iOS incorporó mejoras sustanciales en rendimiento, seguridad y capacidades de desarrollo. La introducción de la App Store en 2008 impulsó la creación de un ecosistema de aplicaciones centralizado, facilitando la distribución y actualización de software. Posteriormente, se añadieron características como la multitarea, el soporte para notificaciones, el uso de sensores avanzados y la optimización para diferentes tamaños de pantalla.
A lo largo de su evolución, iOS ha mantenido una fuerte integración entre hardware y software, lo que ha permitido a Apple ofrecer un alto grado de control sobre la plataforma. Esta estrategia ha contribuido a la estabilidad, seguridad y experiencia de usuario del sistema, consolidándolo como uno de los sistemas operativos móviles más influyentes y utilizados a nivel mundial.

Android

En 2005, Google incursionó en el mercado de los dispositivos móviles con la propuesta de Android, un sistema operativo basado en una versión modificada del kernel de GNU/Linux. Android fue concebido como una plataforma de código abierto, liberada principalmente bajo la licencia Apache, lo que facilitó su adopción por fabricantes y desarrolladores. El sistema operativo Android presenta, entre otras, las siguientes características:

- *Almacenamiento*, utiliza SQLite como sistema de base de datos embebido para el manejo de información local.
- *Conectividad*, soporta tecnologías como GSM/EDGE, iDEN, CDMA, UMTS, Bluetooth, Wi-Fi, LTE y WiMAX.
- *Navegación web*, basada en tecnologías web de código abierto.
- *Multimedia*, incluye soporte para formatos como H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF y BMP.
- *Soporte de hardware*, integra sensores como acelerómetro, cámara, brújula digital, sensor de proximidad y GPS
- *Interfaz multitáctil*, con soporte para pantallas *multitouch*.
- *Multitarea*, permite la ejecución concurrente de múltiples aplicaciones.
- *Tethering*, posibilita el uso del dispositivo como pasarela de red

En el entorno de ejecución (*runtime*), Android incluye bibliotecas centrales (*Core Libraries*) y una máquina virtual. Inicialmente se utilizó Dalvik, la cual fue posterior-

mente reemplazada por Android Runtime (ART), mejorando el rendimiento y la eficiencia en la ejecución de aplicaciones. El kernel de Android, derivado de GNU/Linux, incorpora componentes fundamentales como controladores de pantalla, cámara, memoria flash, el controlador Binder para la comunicación entre procesos (IPC), controladores de teclado, Wi-Fi y audio, así como un administrador de energía. Asimismo, Android cuenta con mecanismos de administración de memoria y procesos, y con sistemas eficientes de intercomunicación entre procesos.

Android fue presentado públicamente en 2007 como parte de la Open Handset Alliance, un consorcio liderado por Google e integrado por fabricantes de dispositivos, operadores de telecomunicaciones y desarrolladores de software. El primer dispositivo comercial con Android fue lanzado en 2008, marcando el inicio de una plataforma abierta que contrastaba con los modelos cerrados predominantes en ese momento. La adopción de Android por parte de fabricantes como Samsung, HTC, Motorola, LG y posteriormente otros actores del mercado se debió, en gran medida, a su flexibilidad, a la ausencia de costos de licenciamiento y a la posibilidad de personalizar el sistema según las necesidades de cada fabricante. Esta estrategia permitió una rápida expansión de Android en distintos segmentos del mercado, desde dispositivos de gama baja hasta teléfonos inteligentes de alto rendimiento.

Con el paso del tiempo, Android evolucionó incorporando mejoras en seguridad, rendimiento y experiencia de usuario, así como un ecosistema de aplicaciones consolidado a través de Google Play. Su compatibilidad con una amplia variedad de arquitecturas de hardware y su integración con servicios de Google contribuyeron a su posicionamiento como el sistema operativo móvil con mayor cuota de mercado a nivel mundial, convirtiéndose en un estándar de facto en la industria de los dispositivos móviles.

Capítulo 2

Procesos e Hilos

Un proceso es una entidad en ejecución a la que el sistema operativo asigna un identificador único, lo que permite su gestión y referencia dentro del sistema. Cada proceso es tratado como una entidad independiente y puede comunicarse con otros procesos mediante mecanismos específicos de intercomunicación.

Un hilo es una unidad de ejecución que existe dentro de un proceso. Los hilos que pertenecen a un mismo proceso comparten el mismo espacio de memoria y recursos, aunque mantienen estados de ejecución independientes, lo que permite una ejecución concurrente más eficiente.

En el presente capítulo se analizarán en detalle los conceptos de procesos e hilos, así como sus características y mecanismos de administración por parte del sistema operativo.

2.1 Introducción a procesos

Todos los sistemas de multiprogramación están construidos en torno al concepto de proceso. De manera simplificada, en un instante determinado un proceso puede encontrarse ejecutándose en el procesador o fuera de él, a la espera de ser ejecutado. Bajo esta visión básica, un proceso puede estar en uno de dos estados: **Ejecución** o **No ejecución**.

Para poder administrar los procesos, el sistema operativo debe identificar a cada uno de ellos y mantener información asociada, como su estado actual, su ubicación en memoria y otros datos de control. Los procesos que no se encuentran en ejecución deben almacenarse en alguna estructura, generalmente colas, donde esperan su turno para ser atendidos por el procesador.

Si todos los procesos que no están ejecutándose estuvieran siempre listos para hacerlo, una única cola de espera sería suficiente. Sin embargo, esta situación no refleja el comportamiento real de un sistema, ya que algunos procesos se encuentran listos para ejecutarse, mientras que otros están bloqueados esperando la finalización de una operación de entrada/salida u otro evento. Por esta razón, el estado de *No ejecución* se divide en dos estados diferenciados: *Listo* y *Bloqueado*. Al considerar esta división, junto con los estados de creación y finalización, se obtiene un modelo de **cinco estados**, como se muestra en la Figura 2-1. Estos estados son los siguientes:

- Nuevo. Proceso que acaba de ser creado, pero que aún no ha sido admitido por el sistema operativo en el conjunto de procesos ejecutables.
- *Listo*. Proceso que se encuentra preparado para ejecutar y que espera la asignación del procesador.
- Ejecución. Proceso que está siendo ejecutado actualmente por la CPU.

- Bloqueado Proceso que no puede continuar su ejecución hasta que ocurra un evento específico, como la finalización de una operación de entrada/salida. Un proceso puede pasar voluntariamente a este estado, por ejemplo, mediante una llamada a funciones como `sleep`.
- Terminado. Proceso que ha sido retirado del conjunto de procesos ejecutables, ya sea porque finalizó su ejecución de forma normal o porque fue abortado.

Este modelo permite al sistema operativo gestionar de manera eficiente la ejecución concurrente de múltiples procesos y optimizar el uso del procesador.

En los sistemas UNIX, el diagrama de estados de los procesos es más complejo que el modelo básico de cinco estados, debido a la distinción entre modos de ejecución y a la existencia de estados adicionales relacionados con la gestión de memoria y la finalización de los procesos. Un proceso en UNIX puede encontrarse en alguno de los siguientes estados o situaciones:

- Ejecutándose en modo usuario, cuando el proceso ejecuta código de aplicación.
- Ejecutándose en modo kernel, cuando el proceso se encuentra atendiendo una llamada al sistema o una interrupción.
- Listo para ejecutarse, pero no en ejecución, esperando a que el kernel le asigne el procesador.
- Dormido en memoria, cuando el proceso se encuentra bloqueado y cargado en memoria, a la espera de que ocurra un evento.
- Listo para ejecutarse, pero suspendido, esperando a que el swapper (proceso 0) lo cargue en memoria principal.
- En transición, estado en el que el proceso ha sido creado, pero aún no está completamente preparado para ejecutar.
- Finalizando, cuando el proceso ejecuta la llamada al sistema exit.
- Zombi, estado en el que el proceso ya ha terminado su ejecución y no existe como entidad ejecutable, pero conserva una entrada en la tabla de procesos para que su proceso padre pueda recuperar el código de salida y cierta información estadística, como los tiempos de ejecución.

El estado *zombi* representa el estado final de un proceso en UNIX y desaparece únicamente cuando el proceso padre recoge su estado de terminación.

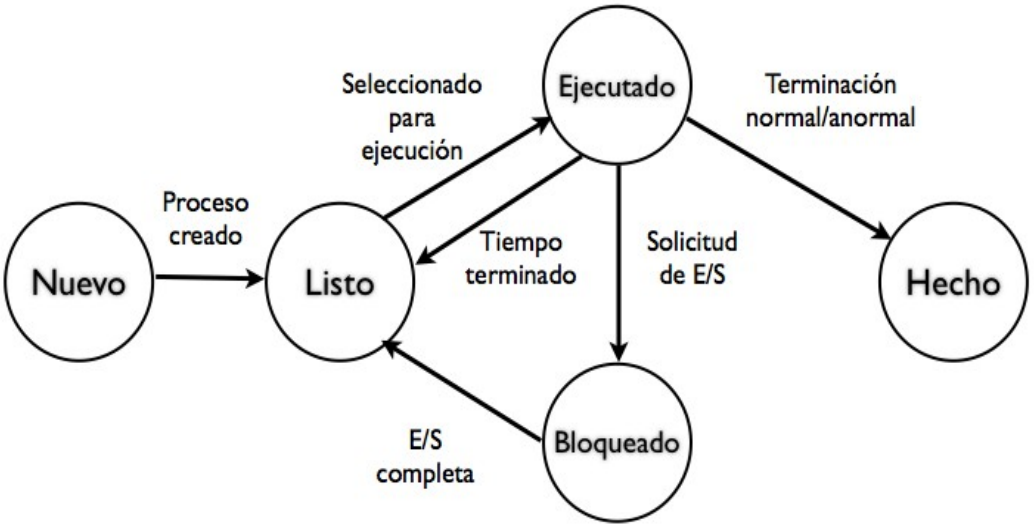


Figura 2-1. Diagrama del modelo de 5 estados.[1]

1 Figura tomada del libro: W. Stallings. Sistemas Opeativos. 2a. Ediciòn. 1997. p. 104

2.2 Control de procesos

El sistema operativo actúa como el controlador central de los sucesos que ocurren en el sistema. Es el encargado de planificar y despachar los procesos para su ejecución en el procesador, asignar y administrar los recursos del sistema, y responder a las solicitudes de servicios realizadas por los programas de usuario mediante llamadas al sistema

Para cumplir con estas funciones, un sistema operativo debe ser capaz de realizar diversas operaciones sobre los procesos, entre las que se incluyen: crear un proceso, destruirlo, suspenderlo, reanudarlo, cambiar su prioridad, bloquearlo, despertarlo, despacharlo y permitir la comunicación entre procesos.
La creación de un proceso implica varias acciones fundamentales, tales como:

- Asignarle un identificador único.
- Insertarlo en la lista de procesos administrados por el sistema.
- Determinar su prioridad inicial.
- Crear su bloque de control de proceso.
- Asignarle los recursos necesarios y un espacio de direcciones.

Cuando un nuevo proceso se añade a la lista de procesos que administra el sistema operativo, deben construirse las estructuras de datos necesarias para su gestión, así como asignarse el espacio de memoria que utilizará durante su ejecución.

En GNU/Linux, el kernel representa a cada proceso mediante un **descriptor de proceso**, definido por la estructura task_struct. Esta estructura contiene punteros a otras estructuras internas que, en conjunto, almacenan toda la información administrativa del proceso, como su estado, prioridad, identificador, contexto de ejecución y relaciones con otros procesos. El campo de estado del descriptor de proceso es un conjunto de indicadores que describe la situación actual del proceso. En la estructura *task_struct*, el campo p-> state puede tomar, entre otros, los siguientes valores principales:

- TASK_RUNNING: el proceso está ejecutándose o listo para ejecutarse.
- TASK_INTERRUPTIBLE: el proceso está dormido y puede ser despertado por una señal.
- TASK_UNINTERRUPTIBLE: el proceso está dormido y no puede ser despertado por señales.
- TASK_STOPPED: el proceso ha sido detenido.
- TASK_TRACED: el proceso está siendo rastreado (por ejemplo, por un depurador).

Cuando un proceso finaliza su ejecución, el kernel puede asignarle estados especiales como:

- EXIT_ZOMBIE: el proceso ha terminado, pero su proceso padre aún no ha recogido su estado de salida.
- EXIT_DEAD: el proceso ha sido completamente eliminado del sistema.

Cuando un proceso crea a otro, el proceso creador se denomina *proceso padre*, mientras que el proceso creado se conoce como *proceso hijo*. Generalmente, estos procesos relacionados necesitan comunicarse y cooperar entre sí para cumplir una tarea común. En las siguientes secciones se analizarán los mecanismos que permiten dicha comunicación y cooperación mediante programación en lenguaje C.

2.3 Sistema de llamado para crear procesos

Los procesos pueden ser creados por el sistema operativo desde el momento en que este se inicia, o conforme surge la necesidad de realizar distintas tareas internas. Por otra parte, el usuario también puede crear procesos, ya sea de forma directa mediante la ejecución de programas, o de forma indirecta cuando una aplicación en uso genera nuevos procesos durante su ejecución. En los sistemas GNU/Linux, la creación de procesos se realiza principalmente a través de la llamada al sistema *`fork()`*. Esta llamada permite que un proceso existente, denominado *proceso padre*, cree un nuevo proceso llamado *proceso hijo*. Tras la ejecución de `fork()`, el sistema operativo crea un nuevo descriptor de proceso y establece una relación de parentesco entre ambos procesos. Desde el punto de vista lógico, el proceso hijo recibe una copia del espacio de direcciones del proceso padre. Sin embargo, en implementaciones modernas de GNU/Linux esta copia se realiza utilizando la técnica de **copy-on-write**, lo que significa que las páginas de memoria no se duplican físicamente hasta que alguno de los procesos intenta modificarlas, optimizando así el uso de memoria y el rendimiento del sistema.

Una característica fundamental de fork() es que ambos procesos continúan su ejecución a partir de la instrucción siguiente a la llamada, pero con diferentes valores de retorno, lo que permite distinguirlos dentro del programa.
El prototipo de la llamada al sistema fork() es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
     #include <sys/types.h>
     #include <unistd.h>

     pid_t fork(void);
```

La creación de dos procesos que ejecuten exactamente el mismo código y sigan el mismo flujo de ejecución no resulta especialmente útil en la práctica. Por esta razón, el valor devuelto por la llamada al sistema fork() constituye el mecanismo fundamental que permite diferenciar el comportamiento del proceso padre y del proceso hijo, haciendo posible que cada uno ejecute código distinto a partir de ese punto.
La llamada fork() devuelve dos valores diferentes, dependiendo del contexto de ejecución:

- En el *proceso hijo,* el valor devuelto es **0**.
- En el *proceso padre*, el valor devuelto es un número entero **mayor que cero**, que corresponde al *identificador del proceso hijo*, conocido como PID (*Process Identifier*).
- En caso de error, fork() devuelve **-1**, y no se crea ningún proceso hijo.

El PID es un identificador único que el sistema operativo asigna a cada proceso para poder gestionarlo y referenciarlo internamente. En los sistemas UNIX y sus derivados, los PID se asignan de forma incremental hasta alcanzar un valor máximo predefinido. Una vez alcanzado dicho límite, el sistema operativo puede reciclar identificadores previamente utilizados por procesos que ya han finalizado su ejecución.
En los sistemas GNU/Linux, el valor máximo permitido para los PID puede consultarse y configurarse a través del archivo del sistema de archivos virtual /proc/sys/

kernel/pid_max. En arquitecturas de 64 bits, este valor suele ser 4,194,303, lo que permite la existencia simultánea de millones de procesos sin colisiones de identificadores.
Este mecanismo de diferenciación basado en el valor de retorno de fork() es esencial para la programación concurrente, ya que permite al desarrollador controlar el flujo de ejecución y definir claramente las responsabilidades del proceso padre y del proceso hijo.

Ejemplo. En el siguiente fragmento de código, tanto el proceso padre como el proceso hijo ejecutan la instrucción de asignación x = 1 después del regreso de la llamada al sistema fork():

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   int x = 0;
   fork();
   x = 1;
   return EXISTE_SUCCESS;
}
```

Tras la ejecución de `fork()`, existen dos procesos independientes: el proceso padre y el proceso hijo. Ambos continúan su ejecución a partir de la instrucción siguiente a la llamada, por lo que los dos ejecutan la asignación `x = 1`. Aunque el código es idéntico, cada proceso cuenta con su propio espacio de direcciones, por lo que la variable `x` en el padre es distinta de la variable `x` en el hijo. La llamada al sistema fork() crea un nuevo proceso mediante la creación de una copia lógica de la imagen del proceso padre en memoria. En los sistemas GNU/Linux modernos, esta copia se implementa mediante la técnica de copy-on-write, lo que implica que las páginas de memoria solo se duplican físicamente si alguno de los procesos intenta modificarlas.
El proceso hijo hereda la mayoría de los atributos del proceso padre, entre los que se incluyen:

- El entorno de ejecución.
- Los privilegios y credenciales.
- Los descriptores de archivos y dispositivos abiertos.
- La prioridad y los atributos de planificación.

Sin embargo, no todos los atributos del padre son heredados por el hijo. En particular:

- El hijo recibe un identificador de proceso (PID) distinto.
- Los tiempos de uso de CPU del hijo se inicializan en cero.
- El hijo no hereda los bloqueos mantenidos por el padre.
- Las alarmas establecidas por el padre no generan notificaciones en el hijo.
- El hijo comienza su ejecución sin señales pendientes, aun cuando el padre las tenga en el momento de ejecutar fork().

Aunque el proceso hijo hereda ciertos parámetros relacionados con la administración de procesos, debe competir por el tiempo de CPU con el resto de los proce-

sos del sistema, incluido su propio proceso padre, bajo la política de planificación del sistema operativo.

A continuación se presenta una extensión directa del ejemplo anterior que permite relacionar explícitamente fork() con el mecanismo de *copy-on-write (COW)*, utilizando una modificación condicional de la variable para forzar la duplicación de memoria.

Ejemplo: fork() y *copy-on-write*
En el siguiente fragmento de código, el valor de retorno de fork() se utiliza para distinguir entre el proceso padre y el proceso hijo. Cada uno modifica la variable x de manera distinta, lo que provoca que el sistema operativo realice la copia física de la página de memoria que contiene dicha variable.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   int x = 0;
   pid_t pid;

   pid = fork();

   if (pid == 0) {
      /* Código ejecutado por el proceso hijo */
      x = 5;
      printf("Hijo:  PID=%ld, x=%d\n", (long)getpid(), x);
   } else {
      /* Código ejecutado por el proceso padre */
      x = 10;
      printf("Padre: PID=%ld, x=%d\n", (long)getpid(), x);
   }
   return EXISTE_SUCCESS;
}
```

Este ejemplo ilustra cómo fork() es eficiente en términos de memoria, ya que evita copias innecesarias hasta que realmente se produce una escritura. El mecanismo de *copy-on-write* permite que la creación de procesos sea rápida y escalable, incluso cuando el espacio de direcciones del proceso padre es grande.

2.4 Sistema de llamado para identificar procesos

Todo proceso en un sistema operativo tipo UNIX tiene asociado un identificador único denominado PID (*Process Identifier*), el cual es un número entero positivo asignado por el kernel. Asimismo, cada proceso mantiene una referencia al proceso que lo creó, conocido como proceso padre, cuyo identificador es denominado PPID (*Parent Process Identifier*).

Para obtener estos valores, el sistema operativo proporciona las siguientes llamadas al sistema:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid(void);
```

El tipo de dato pid_t representa el identificador de un proceso. En la biblioteca GNU C, pid_t es un tipo entero con signo, cuyo tamaño depende de la arquitectura del sistema. La llamada getpid() devuelve el identificador del proceso que la invoca, mientras que getppid() retorna el identificador del proceso padre del proceso actual. Además de la relación padre–hijo, los procesos pueden organizarse en grupos de procesos, los cuales permiten al sistema operativo gestionar de forma conjunta conjuntos de procesos relacionados, por ejemplo, aquellos asociados a una misma sesión o terminal. Para identificar el grupo al que pertenece un proceso, se utiliza la llamada:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
```

Esta función devuelve el identificador del grupo de procesos (*Process Group ID*, PGID) del proceso actual. Para establecer un proceso como líder de un nuevo grupo de procesos, se puede utilizar la llamada:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <unistd.h>

pid_t setpgrp(void);
```

Internamente, esta función establece el PID del proceso como su PGID, convirtiéndolo en líder del grupo. En implementaciones modernas, esta operación suele estar respaldada por la llamada más general `setpgid()`. Debe considerarse que si un proceso padre termina su ejecución antes que sus procesos hijos, estos no quedan sin control. En tal caso, el kernel reasigna automáticamente a los procesos huérfanos a un proceso especial con PID 1, que históricamente fue init y que en sistemas GNU/Linux modernos suele corresponder a systemd. Este proceso se encarga de adoptar a los hijos y recoger su estado de terminación.

Ejemplo. En el siguiente ejemplo, tras la ejecución de la llamada al sistema fork(), tanto el proceso padre como el proceso hijo imprimen sus respectivos identificadores de proceso utilizando la salida estándar.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   pid_t hijo;

   hijo = fork();
   if (hijo == 0) {
      /* Código ejecutado por el proceso hijo */
      fprintf(stdout, "Soy el hijo, PID=%ld\n", (long)getpid());
   } else if (hijo > 0) {
      /* Código ejecutado por el proceso padre */
      fprintf(stdout, "Soy el padre, PID=%ld\n", (long)getpid());
   } else {
      /* Error al crear el proceso */
      perror("fork");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

El uso de stdout en el ejemplo anterior pone de manifiesto una convención fundamental en los sistemas UNIX y sus derivados, como GNU/Linux: a cada proceso se le asocia, por defecto, un conjunto de descriptores de archivo estándar, que permiten la comunicación básica con el entorno.
Estos descriptores son:

- Entrada estándar (stdin): descriptor 0, asociado típicamente al teclado.
- Salida estándar (stdout): descriptor 1, asociado normalmente a la pantalla.
- Error estándar (stderr): descriptor 2, asociado también a la pantalla, pero separado de la salida estándar.

Estos valores están estandarizados por la especificación POSIX.1, que define las constantes:

- STDIN_FILENO para el descriptor 0,
- STDOUT_FILENO para el descriptor 1,
- STDERR_FILENO para el descriptor 2.

Dichas constantes se encuentran definidas en la biblioteca <unistd.h>. Cuando se ejecuta fork(), el proceso hijo hereda los descriptores de archivo abiertos del padre, por lo que ambos procesos escriben inicialmente en el mismo destino de salida estándar. Esto explica por qué tanto el padre como el hijo pueden imprimir mensajes en la terminal sin necesidad de abrir explícitamente un archivo.

Ejemplo. Cadena de procesos.

En una **cadena de procesos**, cada proceso crea **un solo hijo**, y el padre deja de crear nuevos procesos. El resultado es una estructura lineal:
P0 → P1 → P2 → P3 → ... → Pn

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   pid_t hijo;
   int n = 5;

   for (int i = 0; i < n; i++) {
      hijo = fork();

      if (hijo > 0) {
         /* El padre deja de crear más procesos */
         break;
      }
      fprintf(stderr,
            "Proceso PID=%ld, PPID=%ld\n",
            (long)getpid(), (long)getppid());
   }
   return EXIT_SUCCESS;
}
```

Explicación

- En cada iteración, solo el proceso hijo continúa el ciclo.
- El padre sale del ciclo con break.
- Se genera una cadena lineal de procesos.
- Cada proceso tiene exactamente un hijo, excepto el último.

Ejemplo. **Abanico de procesos.**

En un *abanico de procesos,* un único proceso padre crea varios hijos, pero los hijos no crean más procesos:

```
                P0
          /     |     |     \
         P1  P2  P3 ... Pn
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   pid_t hijo;
   int n = 5;

   for (int i = 0; i < n; i++) {
      hijo = fork();

      if (hijo == 0) {
         /* El hijo no crea más procesos */
```

```
            break;
        }
    }
    fprintf(stderr, "Proceso PID=%ld, PPID=%ld\n", (long)getpid(), (long)getppid());
    return EXIT_SUCCESS;
}
```

Explicación

- Solo el proceso padre ejecuta el ciclo completo.
- Cada iteración crea un hijo nuevo.
- Cada hijo ejecuta el break y no genera más procesos.
- Se obtiene una estructura tipo estrella (abanico).

2.5 Sistema de llamada wait ()

¿Qué sucede con el proceso padre después de crear un proceso hijo? Tras la ejecución de la llamada al sistema `fork()`, tanto el proceso padre como el proceso hijo continúan su ejecución de manera concurrente a partir de la instrucción siguiente a dicha llamada. Como consecuencia, el proceso padre puede terminar antes que el hijo, o bien el hijo puede finalizar primero.

Si el proceso padre desea esperar a que uno de sus procesos hijos termine, debe invocar la llamada al sistema wait() o, de manera más general, waitpid(). Estas llamadas permiten al sistema operativo notificar al padre la finalización de sus hijos y recuperar su estado de terminación, evitando así la aparición de procesos zombi.
Los prototipos de estas llamadas son los siguientes:

```
PROTOTIPOS DE LAS FUNCIONES
    #include <sys/types.h>
    #include <sys/wait.h>

    pid_t  wait (int *stat_loc);
    pid_t waitpid(pid_t pid, int *wstatus, int options);
```

La llamada al sistema wait() suspende la ejecución del proceso que la invoca hasta que ocurre alguno de los siguientes eventos:

- Uno de sus procesos hijos termina su ejecución.
- Un proceso hijo se detiene.
- El proceso que invoca wait() recibe una señal.

La función wait() retorna inmediatamente si el proceso no tiene procesos hijos. Si un hijo termina o se detiene y aún no se ha solicitado su espera, wait() devuelve de inmediato.
Si wait() retorna debido a la terminación o detención de un hijo:

- El valor de retorno es positivo y corresponde al PID del proceso hijo.
- En caso de error, retorna -1 y se establece un valor apropiado en la variable global errno.

Algunos valores relevantes de errno son:

- ECHILD: el proceso no tiene hijos a los cuales esperar.
- EINTR: la llamada fue interrumpida por la recepción de una señal.

Análisis del estado de terminación

El parámetro stat_loc es un apuntador a una variable entera donde el kernel almacena información sobre el estado de terminación del proceso hijo. Este valor debe analizarse utilizando los macros definidos en <sys/wait.h>, entre los que se encuentran:

- WIFEXITED(*stat_loc). Evalúa a verdadero si el proceso hijo terminó de forma normal.
- WEXITSTATUS(*stat_loc). Si el hijo terminó normalmente, obtiene los 8 bits menos significativos del valor pasado a exit(), _exit() o retornado desde main().
- WIFSIGNALED(*stat_loc). Evalúa a verdadero si el proceso hijo terminó debido a una señal no capturada.
- WTERMSIG(*stat_loc). Si el proceso hijo terminó por una señal, obtiene el número de dicha señal.

Uso de `waitpid()`

Cuando se requiere esperar por un proceso hijo específico, o se necesita un control más fino del comportamiento de espera, se debe utilizar la función waitpid(). Esta llamada suspende la ejecución del proceso actual hasta que el proceso hijo especificado finaliza o hasta que ocurre un evento controlado por las opciones proporcionadas.
El parámetro pid puede tomar los siguientes valores:

- -1: espera por cualquier proceso hijo.
- > 0: espera por el hijo cuyo PID sea igual a pid.
- 0: espera por cualquier hijo cuyo grupo de procesos sea igual al del proceso llamador.
- < 0: espera por cualquier hijo cuyo PGID sea igual al valor absoluto de pid.

El parámetro **wstatus* cumple la misma función que stat_loc en wait() y se analiza con los mismos macros.
El parámetro *options* permite modificar el comportamiento de la llamada mediante una combinación de las siguientes banderas:

- WEXITED: espera por hijos que hayan terminado.
- WSTOPPED: espera por hijos que hayan sido detenidos por una señal.
- WNOHANG: retorna inmediatamente si ningún hijo ha terminado.
- WNOWAIT: no elimina al hijo de la tabla de procesos.
- WUNTRACED: retorna si un hijo se ha detenido (aunque no esté siendo trazado).
- WCONTINUED: retorna si un hijo ha reanudado su ejecución tras recibir SIGCONT.

Las banderas WUNTRACED y WCONTINUED solo tienen efecto si la opción SA_NOCLDSTOP no ha sido establecida para la señal SIGCHLD.
El uso correcto de wait() y waitpid() es esencial para: sincronizar procesos padre e hijo, recuperar códigos de terminación, evitar procesos zombi, implementar servidores concurrentes y gestores de tareas.

2.6 Sistema de llamada _exit () y exit ()

Como se ha visto, el proceso debe terminar de alguna manera, es decir, de forma normal o anormal. Lo que se desea es una terminación normal, en el mejor de los

casos, para esto el proceso debe hacer un llamado al sistema *_exit()*. El prototipo de la función *_exit()* es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <unistd.h>

        void  _exit (int status);
```

El argumento status define el estado de terminación del proceso, que está disponible para el padre de este proceso cuando invoca al llamado wait(). Aunque se define como un entero, solo los 8 bits finales del estado están disponibles para el padre. Por convención, un estado de terminación de 0 indica que un proceso se completó correctamente, y un valor de estado distinto de cero indica que el proceso terminó.
Otra forma de terminar el programa es invocando a la función *exit()* que realiza varias acciones antes del llamado a *_exit().* Esta función se encarga de retirar los recursos que está utilizando el proceso, así como dejarlo preparado para su eliminación, quitarlo del planificador e indicar su terminación a su padre por medio de la señal SIGCHLD. Para pasar de un estado a otro se define un estado transitorio que en el sistema se le llama zombie. En GNU/Linux si el proceso que termina no tuviera padre, ya que este acabó antes que él, se eliminaría directamente del planificador por la llamada *exit()*, y sería adoptado por el proceso 1 (init). El prototipo de la función *exit()* es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <stdlib.h>

        void  exit (int status);
```

El llamado a la función *exit* termina la ejecución del proceso y devuelve el valor de la variable *status* al proceso padre. Desde el sistema GNU/Linux se puede consultar lo que devuelve el último proceso que finaliza por medio de la variable de entorno “?”.

Ejemplo. Programa que muestra el empleo de wait.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
 pid_t hijo;
 int estado;
 if ( (hijo=fork()) == -1)
 {
  perror ("fallo el fork");
  exit (1);
 }
```

```
 else if (hijo == 0)
  fprintf (stderr, "soy el hijo con pid = %ld \n", (long) getpid());
 else if (wait(&estado) !=hijo)
  fprintf (stderr, "una señal debio interrumpir la espera \n");
 else
  fprintf (stderr, "soy el padre con pid = %ld e hijo con pid = %ld\n",
  (long)getpid(),(long)hijo);
 exit(0);
}
```

Ejemplo. Programa que muestra el empleo de waitpid.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
 #include <unistd.h>
int main(int argc,char*argv[])
{
 pid_t hijo[5];
 int estado,i,j;
 long factorial=1;

for (j=0;j<argc-1;j++)
 {
        if ( (hijo[j]=fork())== -1)
        {       perror ("fallo el fork"); exit (1);         }
        else
          if (hijo[j] == 0)
          {
                fprintf (stdout, "soy el hijo con pid = %ld \n", (long) getpid());
                for (i=atol(argv[j+1]);i>0;i--)
                        factorial=factorial*i;
                fprintf(stdout,"El factorial de 20 es:%ld\n",factorial);
                sleep(2);
                exit(0);
        }
        }//fin for
for (j=0;j<argc-1;j++)
{
        if (  (waitpid(-1,&estado,0)==-1)) fprintf (stderr, "una señal debio interrumpir
la espera \n");
        else fprintf(stdout,"el hijo:%d con pid %ld termino\n",j,(long)hijo[j]);
}
 exit(0);
}
```

2.7 Estado Zombi

*Un proceso zombi es un proceso que y*a terminó su ejecución, pero su proceso padre no ha recogido su estado de salida mediante wait() o waitpid(). El proceso zombi no consume CPU ni memoria de usuario, pero sí ocupa una entrada en la

tabla de procesos, conservando: PID, código de salida e Información estadística mínima.

A continuación se ilustra claramente el estado zombi, primero sin usar wait() y después usando wait(), conectando el comportamiento observable con el funcionamiento interno del sistema operativo.

Caso 1: Proceso zombi sin wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
pid_t pid;
pid = fork();
if (pid == 0) { /* Proceso hijo */
        printf("Hijo terminado. PID=%ld\n", (long)getpid());
        exit(0);
        } else { /* Proceso padre */
                printf("Padre en ejecución. PID=%ld\n",(long)getpid());
        sleep(30); /* El padre NO llama a wait() */
        }
return EXIT_SUCCESS;
}
```

Qué ocurre paso a paso

1. El proceso padre crea un hijo con fork().
2. El hijo finaliza inmediatamente con exit(0).
3. El padre sigue vivo, pero no ejecuta wait().
4. El kernel: Marca al hijo como EXIT_ZOMBIE y conserva su información para el padre.
5. El proceso hijo queda en estado zombi durante 30 segundos.

Cómo observar el zombi: desde otra terminal, mientras el padre duerme:

```
ps -el | grep Z
```

Caso 2: Proceso **sin zombi** usando wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
   pid_t pid;
   int status;

   pid = fork();

   if (pid == 0) {
      /* Proceso hijo */
      printf("Hijo terminado. PID=%ld\n", (long)getpid());
      exit(0);
```

```
    } else {
        /* Proceso padre */
        wait(&status);  /* Recolecta al hijo */
        printf("Padre: hijo recolectado\n");
    }
    return EXIT_SUCCESS;
}
```

Qué ocurre ahora

1. El hijo termina su ejecución.
2. El kernel marca al hijo como terminado.
3. El padre ejecuta wait().
4. El kernel: Entrega el estado de salida al padre y elimina completamente al hijo de la tabla de procesos.
5. No existe estado zombi.

Si se ejecuta ps, el proceso hijo no aparece.

Si el proceso padre termina antes que el hijo:

- El hijo se vuelve huérfano
- El kernel lo reasigna al proceso con PID 1 (init o systemd)
- PID 1 ejecuta wait() automáticamente
- **No queda zombi**

2.8 Sistema de llamada exec

La llamada al sistema fork() crea un nuevo proceso que es, inicialmente, una copia casi exacta del proceso padre. Sin embargo, en la práctica, muchas aplicaciones no desean que el hijo ejecute el mismo código que el padre, sino que ejecute un programa completamente distinto. Para este propósito existe la familia de llamadas exec, la idea fundamental es: exec no crea un nuevo proceso, sino que reemplaza la imagen del proceso actual por un nuevo programa.

Cuando un proceso invoca exec su código, datos, heap y stack son reemplazados, el PID permanece igual, los descriptores der archivos abiertos se conservan (salvo que tengan la bandera FD_CLOEXEC), y si exec tiene éxito, nunca regresa.

La manera tradicional de utilizar la combinación *fork()-exec()* es dejar que el hijo ejecute el *exec()* para el nuevo programa mientras el padre continúa con la ejecución del código original. Existen seis variantes de la llamada *exec()* al sistema, las cuales se distinguen por la forma en que son pasados los argumentos desde la línea de comando y desde el ambiente, por si es necesario proporcionar la ruta de acceso y el nombre del archivo ejecutable. Los dos grupos que albergan esas variaciones son *execl()* y *execv()*:

1. *execl* (*execl*, *execlp* y *execle*). Estos llamados pasan los argumentos de la línea de comando como una lista y son útiles si se conoce el número de argumentos de la línea de comando en el momento de la compilación.
2. *execv* (*execv*, *execvp* y *execve*). Estos llamados pasan los argumentos de la línea de comando en un arreglo de argumentos.

Los prototipos se muestra a continuación.

```
PROTOTIPO DE LA FUNCIÓN
#include <unistd.h>

int execl (const char *path, const char *arg0, ... , const char *argn,
                char * / *NULL*/);
int execle (const char *path, const char *arg0, ... , const char *argn,
                char * /*NULL*/, const char *envp[]);
int execlp (const char *file, const char *arg0, ... ,const char *argn,
                char */*NULL*/);
```

```
PROTOTIPO DE LA FUNCIÓN
#include <unistd.h>

int execv (const char *path, const char *argv[]);
int execvp (const char *file, const char *argv[]);
int execve (const char *'path, const char *argv[], const char *envp[]);
```

Ejemplo. Programa que crea un proceso para ejecutar ls –l.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main( )
{
  pid_t  hijo;
  int estado;

  if (( hijo = fork(  )) = = -1)
 {
   perror (“Error al ejecutar fork”);
  exit (1);
 }
 else if (hijo = = 0)
      {
        if  (execl (“/usr/bin/ls”, “ls”, “-l”, NULL) < 0)
        {
          perror (“Falla en la ejecución de ls”);
          exit (1);
        }
    }
   else if (hijo ! = wait (&estado))
           perror (“Se presentó una señal antes de la terminación del hijo”);
 exit (0);
}
```

La función *perror()* muestra un mensaje con el error estándar seguido por la última llamada al sistema o la biblioteca que lo produjo. Su prototipo es el siguiente.

PROTOTIPO DE LA FUNCIÓN

```
#include <stdio.h>

int perror ( const char *s);
```

2.8 Hilos

Los **hilos (threads)** representan un mecanismo de ejecución concurrente **dentro de un mismo proceso**, y constituyen una alternativa más ligera que la creación de procesos independientes. A diferencia de los procesos, los hilos **no son entidades completamente aisladas**, sino que **comparten el mismo espacio de direcciones**, los mismos archivos abiertos y otros recursos del proceso al que pertenecen. Aunque en ocasiones se les describe como una forma de trabajar con *procesos no emparentados*, conceptualmente los hilos deben entenderse como **unidades de ejecución que** coexisten dentro de un mismo proceso, cooperando de manera estrecha. El uso de hilos resulta especialmente adecuado cuando se requiere: ejecutar múltiples tareas de manera concurrente, compartir datos de forma eficiente sin mecanismos costosos de IPC, Reducir el overhead asociado a la creación y destrucción de procesos, mejorar la capacidad de respuesta de aplicaciones interactivas y aprovechar arquitecturas multinúcleo

Modelo de hilos en POSIX

Para trabajar con hilos en C se utiliza la librería pthreads, la cual forma parte del estándar POSIX (Portable Operating System Interface). En sistemas GNU/Linux, la interfaz de programación de hilos está especificada por el estándar: IEEE POSIX 1003.1-2008. Las implementaciones que cumplen este estándar son conocidas como POSIX Threads, o simplemente pthreads. Las características principales de es que cada hilo tiene su propio: contador de programa, stack, y registros. En la tabla 2-1 se pueden observar algunas funciones usadas para el manejo de hilos, estas se verán con mayor detalle en las secciones posteriores. Desde el punto de vista del kernel Linux moderno, los hilos se implementan como entidades planificables, de forma similar a los procesos, mediante el modelo 1:1 (un hilo de usuario corresponde a una entidad del kernel).

La diferencia entre un hilo y un proceso es que los procesos no comparten la misma memoria, mientras que, los hilos sí comparten totalmente la memoria. Para la creación de los hilos se usan las funciones de la librería *pthread* o de cualquier otra que soporte los llamados *threads* mientras que para crear procesos se usa la llamada al sistema *fork*. Teniendo en consideración que el proceso que crea a los hilos es el líder de ellos, y será el padre del conjunto. Para compilar programas que hagan uso de la librería *pthread*, usando GNU cc o GNU Compiler Collection gcc se ejecuta la orden:

```
cc hilos.c -o hilos -lpthread
```

o

```
gcc hilos.c -o hilos -lpthread
```

Tabla 2-1. Lista de llamados en POSIX 1.c.

Descripción	POSIX

administración de hilos	pthread_create pthread_exit pthread_kill pthread_join pthread_self
Exclusión mutua	pthread_mutex_init pthread_mutex_destroy pthread_mutex_lock pthread_mutex_trylock pthread_mutex_unlock
variables de condición	pthread_cond_init pthread_cond_destroy pthread_cond_wait pthread_cond_timedwait pthread_cond_signal pthread_cond_broadcast

2.8.1 Creación de hilos

La función *pthread_create()* es usada para crear un hilo con ciertos atributos, y esté ejecutará una determinada función o subrutina con los argumentos que se le indique. El prototipo de la función es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
    #include <pthread.h>

    int pthread_create (pthread_t *thread, const pthread_attr_t
    *attr, void* (*start_routine) (void *), void *arg);
```

Los atributos para un proceso son especificados dentro del parámetro *attr*, si *attr* es NULL, el atributo por omisión es usado. Si la función se realiza con éxito, se almacena el ID del hilo en la localidad referenciada por el apuntador *thread*. El hilo que está creado ejecuta la función o rutina *start_routine()* con *arg* como su argumento de entrada para la función. Si se necesita pasar o devolver más de un parámetro a la vez, se puede crear una estructura y colocar ahí los campos necesarios.

Cuando finaliza la función *start_routine()* se llama implícitamente a *pthread_exit()* o su equivalente.

El estado de las señales del nuevo hilo serán:

- La máscara de señales del hilo creador que le son heredadas, o
- Nulo para el conjunto de señales pendientes del nuevo hilo.

Un hilo termina si:

- Se llama a la función *pthread_exit()*, especificando un valor de estado final que es disponible para otros hilos en el mismo proceso llamando a la función *pthread_join()*.
- Realiza un return al final de la función start_routine, lo cual es equivalente al llamado de la función *pthread_exit()*.
- Es cancelado invocando a la función *pthread_cancel()*.
- El hilo principal ejecuta un return desde *main c*ausando así la terminación de todos los hilos.

La función *pthread_create()* retorna un 0 en caso de éxito o un número de error en otro caso, y el contenido del parámetro **thread* se indefine. Los errores que pueden retornarse en errno son:

- EAGAIN. Insuficiente recursos para crear un hilo, o el límite de los hilos del sistema fueron alcanzados (en GNU/Linux vea /proc/sys/kernel/threads-max).
- EINVAL. Inválido el atributo.
- EPERM. Política de planificación no permitida y los parámetros de *attr*.

La función *pthread_self()* devuelve el ID del hilo que realiza la llamada, el valor es el mismo que se devuelve en **thread* en la llamada *pthread_create* que creó este hilo. El prototipo de la función es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
    #include <pthread.h>

    pthread_t pthread_self (void );
```

2.8.2 Terminación de un hilo

La función *pthread_exit()* termina la ejecución del hilo que haga su invocación y hará disponible el valor asignado dentro del parámetro *value_ptr* para cualquier proceso que invoque a la función *pthread_join* con éxito con el hilo que hace la llamada. El prototipo de la función es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
    #include <pthread.h>

    int pthread_exit (void *value_ptr);
```

2.8.3 Atributos de un hilo

La función *pthread_attr_init()* se encarga de inicializar el objeto de atributos colocado dentro del parámetro *attr* del hilo con los valores por defecto utilizado en una implementación. Su prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <pthread.h>

int pthread_attr_init ( pthread_attr_t *attr);
```

2.8.4 Destrucción de los atributos de un hilo

La función *pthread_attr_destroy*() se encarga de destruir el objeto de atributos *attr* del hilo con valores no definidos en una implementación. Su prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <pthread.h>

int pthread_attr_destroy (pthread_attr_t *attr);
```

Un objeto de atributos destruido con la función podrá ser inicializado posteriormente con la función *pthread_attr_init*().

2.8.5 Esperar la terminación de un hilo creado

La función *pthread_join*() suspende la ejecución del hilo que hace la llamada hasta que el hilo destino termine. Esta función es un mecanismo que permite la sincronización de hilos (vea figura 2-2). El prototipo de la función es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <pthread.h>

int pthread_join ( pthread_t thread, void **value_ptr);
```

El valor que pase el hilo destino (parámetro *thread*) con la función *pthread_exit()* estará disponible en la localidad de memoria a la que hace referencia el apuntador *value_ptr*.

La función espera hasta que termine el hilo especificado como parámetro *thread*. La función retorna un 0 en caso de éxito o el número de error en otro caso. Los errores pueden ser:

- EDEADLK. Un interbloqueo fue detectado, es decir, dos hilos esperan uno del otro.
- EINVAL. El hilo especificado con el parámetro thread no está unido a otro hilo.
- ESRCH. El hilo especificado con el parámetro thread no fue encontrado.

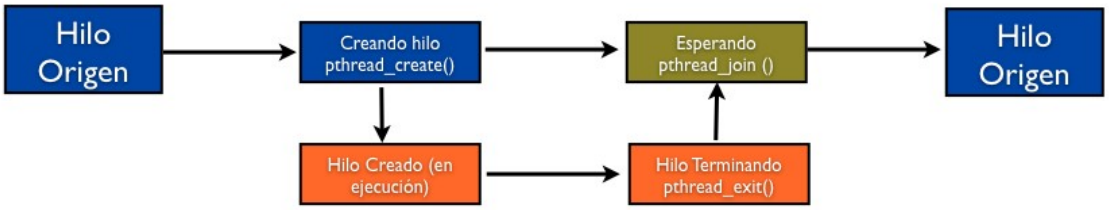


Figura 2-2. Sincronización de hilos.

Ejemplo. El siguiente código muestra la creación de un hilo que calcula el factorial de un número.

```
/* Crea un hilo para calcular el factorial de un número que se pasa como
argumento desde      línea de comandos. Modo de compilar:          gcc -Wall hilos.c
-lpthread -o hilos */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
long prod=1;
void *factorial (void *valor);
int main (int argc, char *argv[])
{
 pthread_t tid;
 pthread_attr_t attr;
 if (argc != 2)
 {
  fprintf(stderr, "Uso: ./hilos <entero>\n");
  return -1;
 }
 // Coloca atributo predeterminados o coloque NULL
 pthread_attr_init(&attr);
 // Crear hilo
 pthread_create(&tid, &attr, factorial, argv[1]);
 // Esperar finalizado de hijo --- semejante a wait
 pthread_join(tid, NULL);
 printf("Factorial: %ld\n", prod);
 return EXIT_SUCCESS;
 }

void *factorial (void *valor)
{
 int i=1;
 while (i <= atol(valor))
  prod *= (i++);
 pthread_exit(0);
 }
```

Ejemplo. El siguiente código muestra la creación de un conjunto de hilos.

```
/* Crea un conjunto de hilos para que cada  uno calcule el factorial de un número
que se pasa  como argumento desde línea de comandos.
```

```
Modo de compilar:     gcc -Wall chilos.c -lpthread -o chilos
Se ejecuta:           chilos <numero1> <numero2> ... <numero10>
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct dhilos {
        int id;
        long prod;
} DHILOS;
DHILOS pm_hilos[10];
void *factorial (void *valor);
int main (int argc, char *argv[])
{
 pthread_t tid[argc-1];
 pthread_attr_t attr;
 int i;
 if (argc<2)
 {
  perror ("Uso: ./chilos <entero1> <entero2> ... \n");
  exit(1);
 }
 // Crear hilos
   for (i=0;i<argc-1;i++)
   {
     pthread_attr_init(&attr); // Atributos predeterminados
     pm_hilos[i].id=i+1;
     pm_hilos[i].prod=atol(argv[i+1]);
     pthread_create(&tid[i], &attr, factorial, &pm_hilos[i]);
    }
   // espera el proceso principal a cada hilo creado
   for (i=0;i<argc-1;i++)
    pthread_join(tid[i], NULL); // Esperar finalización, semejante a wait-cuando usa
fork
  for (i=0;i<argc-1;i++)
    printf("Factorial de: %s = %ld\n", argv[i+1],pm_hilos[i].prod);
 return EXIT_SUCCESS;
 }
void *factorial (void *valor)
{
 int i=1, prod=1;
 DHILOS *datos;
 datos=(DHILOS*)valor;
 while (i <= datos->prod)
  prod *= (i++);
 datos->prod=prod;
 pthread_exit(&(datos->prod));
 }
```

EJERCICIOS PROPUESTOS

1. Investigue los procesos activos en su sistema.
   a) ¿Que instrucción le permite visualizarlos?
   b) ¿Qué parámetros son utilizados con dicha instrucción?

2. En el sistema GNU/Linux, investigue el uso del comando ps, y del comando top.
   a) ¿Cuál es la diferencia entre ellos?
   b) ¿Cuál es el proceso 1?
   c) ¿Quién es el padre del proceso 1?

Capítulo 3

Mecanismos de comunicación entre procesos-IPC

Todos los procesos, parientes o no, necesitan en ocasiones comunicarse entre sí, para esto, el sistema brinda formas básicas de comunicación como stream (pipe, fifo y sockets) y mensajes (colas de mensajes y sockets datagramas). Si los procesos son parientes, la comunicación se puede realizar por medio de una tubería (pipe), y si se necesita proteger el medio de comunicación se pueden utilizar mecanismos de sincronización, ya sea implementados como unidades que requieren una llave para poder accesar a los recursos compartidos, caso de los sistema derivados de System V, o sin utilizar llaves por medio de llamados a funciones POSIX.

3.1 Comunicación mediante tuberías

La comunicación entre procesos es fundamental para que intercambien datos, para llevar a cabo dicha comunicación, hay que tener en consideración, en primer lugar, si los procesos se van a comunicar en la misma máquina y si están emparentados, y segundo, si estos va a comunicar desde máquinas diferentes. A continuación se muestran las tuberías que son las formas básicas de comunicación, las tuberías son mecanismos clásicos de comunicación entre dos o más procesos emparentados y en la misma máquina. La teoría que se muestra, está basada en las facilidades de comunicación entre procesos (IPC) de los sistemas UNIX System V y derivados.

3.1.1 Tuberías sin nombre - pipe

Las tuberías sin nombre, también llamadas pipe, son viejas formas de IPC y son proporcionados por todos los sistemas UNIX y derivados. Su facilidad de uso, dá la ventaja de una sencilla implementación, pero, presentan dos limitaciones:

1. Los datos fluyen en una dirección.
2. Pueden usarse sólo entre procesos que tienen un ancestro en común.

Normalmente una tubería es creado por un proceso que invoca a la función *fork*, y es usada entre el proceso creador y su descendiente. Un tubería sin nombre se crea llamando a la función *pipe*, o *pipe2* (que es una función específica de GNU/Linux a partir d la versión 2.6.27 y con glibc está disponible a partir de la versión 2.9) cuyos prototipos son los siguientes:

PROTOTIPO DE LA FUNCIÓN

```
#include <unistd.h>

int pipe (int filedes[2]);
```

El valor retornado es un 0 si todo está correcto y un –1 si existe un error. Los dos descriptores son retornado a través del argumento *filedes,* donde el argumento *filedes*[0] se utiliza para abrir la tubería para lectura, y el argumento *filedes*[1] se utiliza para abrir la tubería para escritura. Normalmente, la salida de *filedes*[1] es la entrada para *filedes*[0], como se muestra en el esquema de las figuras 3-1. Cuando se quiere comunicar los dos procesos (padre e hijo) para que ambos lean y escriban, se puede programar un esquema como lo indica la figura 3-2.

PROTOTIPO DE LA FUNCIÓN

```
#include <unistd.h>

int pipe2 (int filedes[2], int flags);
```

En la función pipe2 si flags es 0 actua de forma semejante a la función pipe. Mediante la operación OR bit a bit sobre flags se obtiene información de la tubería:
O_CLOEXEC. Establece el indicador de cierre al ejecutar (FD_CLOEXEC) en los dos nuevos descriptores de archivo.
O_DIRECT (a partir de GNU/Linux 3.4). Crea una tubería que realiza E/S en modo "paquete". Cada escritura (write) en la tubería se procesa como un paquete independiente, y las lecturas (read) de la tubería leerán un paquete a la vez. Se debe tener enconsideraciónlos siguientes puntos:

- Las escrituras de más de PIPE_BUF bytes se dividirán en varios paquetes. La constante PIPE_BUF se define en <limits.h>.
- Si una lectura (read) especifica un tamaño de búfer menor que el siguiente paquete, se lee el número de bytes solicitado y se descartan los bytes sobrantes del paquete. Especificar un tamaño de búfer de PIPE_BUF será suficiente para leer los paquetes más grandes posibles.
- No se admiten paquetes de longitud cero. Una operación read que especifica un tamaño de búfer de cero no es operativa y devuelve 0. Los kernels antiguos que no admiten esta opción lo indicarán mediante un error EINVAL.

O_NONBLOCK. Establece el indicador del estado de archivo en los descriptores de archivos abiertos a las que hacen referencia los nuevos descriptores de archivo.

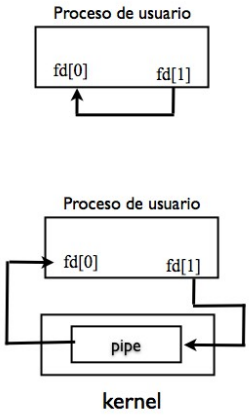


Figura 3-1. Dos forma de ver a una tubería en UNIX.

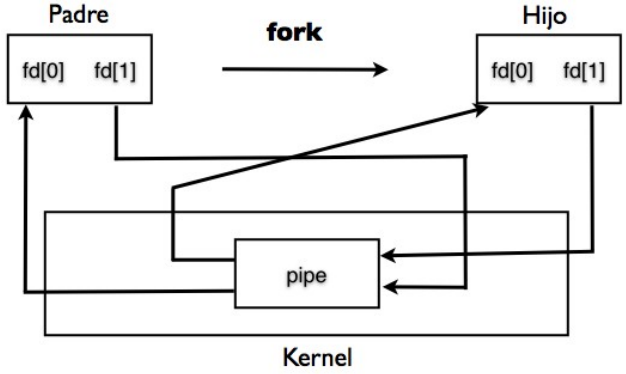


Figura 3-2. Una tubería half-duplex después de un fork.

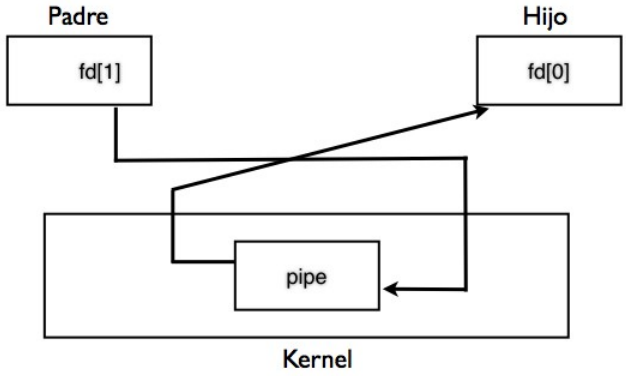


Figura 3-3. Tubería entre un padre y un hijo.

Ejemplo. Programa para crear una tubería desde el padre y heredada al hijo. Los procesos envían datos a través de ella. El esquema que se realiza es el mostrado en la Figura 3-2, y Figura 3-3.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```
#include <sys/wait.h>
#define MAXLINEA 80
int main ()
{
   int n, fd[2];
   pid_t hijo;
   char linea [MAXLINEA]

 if (pipe(fd) < 0) {
        fprintf (stderr,"error de pipe");
        exit(0);
       }
 if ( (hijo = fork() ) < 0) {
         fprintf (stderr, "error de fork");
         exit(0);
       }
 else if (hijo> 0)
         { /*  padre */
          close (fd[0]);
          write (fd[1], "hola mundo\n",12);
         }
       else {
              close (fd[1]);
              n = read (fd[0], linea, MAXLINEA);
             write (STDOUT_FILENO, linea, n);
            }
 return EXIT_SUCCESS;
}
```

Algunas veces el padre se ejecuta más rápido que el hijo, lo que se puede hacerse en el código es intercambiar el orden.

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#define MAXLINE 80

int main ()
{
 int n, fd[2];
 pid_t hijo;
 char linea[MAXLINE];
 int estado;
 if (pipe(fd) < 0) { fprintf (stderr,"error de tubería"); exit(0); }
 if ( (hijo=fork()) == 0)
 {
  close (fd[0]);
  write (fd[1], "hola mundo \n",12);
}
```

```
 else
 {
  close (fd[1]);
  n = read (fd[0], linea, MAXLINE);
  write (STDOUT_FILENO, linea, n);
  printf("numero de lineas %d \n",n);
}
return EXIT_SUCCESS;
}
```

3.1.2 Tuberías con nombre - fifo

El sistema de llamado *mkfifo* permite crear un archivo especial llamado FIFO, el cual es una tubería con un nombre asociado. Anteriormente se vio el uso del llamado a la función *pipe*, que crea un canal de comunicación para procesos, pero este es anónimo, es decir, no tiene nombre, ahora *mkfifo* también es usado para comunicar procesos, pero este si tiene nombre y ruta en el sistema de archivos del SO. El prototipo de la función es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

En el prototipo anterior, el parámetro *pathname* será el nombre y ruta donde se creará el archivo de tipo FIFO, y el parámetro *mode* es utilizado para especificar los permisos (atributos) de dicho archivo. La función *mkfifo()* retornará un 0 en caso de éxito y un -1 en caso de error, y en errno se colocará el tipo de error devuelto.

Una vez que se crea un archivo especial FIFO, cualquier proceso puede abrirlo para lectura o escritura, de la misma forma que a un archivo ordinario. Sin embargo, debe estar abierto en ambos extremos simultáneamente antes de que pueda realizar cualquier operación de entrada o salida en él. Abrir un FIFO para leer normalmente bloquea hasta que algún otro proceso abre el mismo FIFO para escribir, y viceversa.

Ejemplo. Programa que crea una tubería con nombre para enviar un mensaje del proceso hijo al proceso padre.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main (void)
{
 pid_t hijo;
```

```
int file;
char mensaje[20];
unlink("namepipe"); // borra el archivo del sistema de archivos
umask(~0666); // cambia la máscara de permisos por defecto
if(mkfifo("namepipe",0666)==-1)
{
   perror("error en mkfifo");
   exit(-1);
}
if ( (hijo=fork ( ))== 0)
{
 fprintf (stdout,"soy el hijo, ID=%ld\n",(long)getpid());
 if( (file=open("namepipe",O_WRONLY) ) ==-1)
 {
   perror("error en mkfifo");
   exit(-1);
 }
 write (file,"soy el hijo,ID...\n",20);
 close(file);
 getchar();
 exit(0);
}
  if (hijo > 0)
 {
   fprintf (stdout,"soy el padre, ID = %ld\n",(long)getpid());
   if((file=open("namepipe",O_RDONLY))==-1)
   {
   perror("error en open O_RDONLY");
   exit(-1);
   }
   read(file,mensaje,20);
   fprintf(stdout,"%s\n",mensaje );
   close(file);
  }
 return EXIT_SUCCESS;
}
```

3.2 Mecanismos IPC derivados de System V

El paquete de comunicación entre procesos de los sistema UNIX System V y derivados, como GNU/Linux, se componen de tres mecanismos:

1. Los semáforos, que permiten sincronizar procesos;
2. La memoria compartida, que va a permitir que los procesos compartan su espacio de direcciones virtuales; y
3. Las colas de mensajes, que posibilitan el intercambio de datos con un formato determinado.

Estos mecanismos (vea el resumen en la tabla 3-1) están implementados como una unidad y comparten características comunes, entre las que se encuentran:

- Una tabla cuyas entradas describen el uso que se hace del mecanismo.
- Cada entrada de la tabla tiene una llave numérica elegida por el usuario.

- Cada mecanismo dispone de una llamada “get” para crear una entrada nueva o recuperar alguna ya existente. Dentro de los parámetros de esta llamada se incluye una llave y una máscara de indicadores. El núcleo busca dentro de la tabla alguna entrada que se ajuste a la llave suministrada.
- Cada entrada de la tabla tiene un registro de permisos que incluye: ID del usuario y grupo del proceso que ha reservado la entrada.
- Cada entrada contiene información de estado, en la que se incluye el identificador del último proceso que ha utilizado la entrada.
- Una llamada de “control” que permite leer y modificar el estado de una entrada reservada, así como también permite liberarla.

Tabla 3-1. Resumen de las llamadas IPC.

	Semafóros	**Memoria compartida**	**Cola de mensajes**
Bibliotecas	<sys/types.h>, <sys/ipc.h>		
Biblioteca	<sys/sem.h>	<sys/shm.h>	<sys/msg.h>
Llamado para crear o abrir	semget	shmget	msgget
Llamado para operaciones de control	semctl	shmctl	msgctl
Llamado para operaciones	semop	shmat, shmdt	msgsnd, msgrcv

3.2.1 Llaves

Todas las formas de IPC, excepto las tuberías sin nombre, tienen asociado un espacio de nombres para llevar a cabo el intercambio de mensajes. Este conjunto de posibles nombres otorgados al IPC es el identificador que tiene los procesos para entablar un diálogo. En la tabla 3-2 se pueden observar los tipos de nombres de los espacios asociados con cada mecanismo IPC.

Tabla 3-2. Tipos de los nombres de espacios para IPC.

Mecanismo IPC	**Tipo de espacio de nombre**	**Identificación**
pipe	Sin nombre	Descriptor de archivo
FIFO	Nombre de ruta	Descriptor de archivo
Cola de mensajes	Llave ***key_t***	Identificador
Memoria compartida	Llave ***key_t***	Identificador

Mecanismo IPC	Tipo de espacio de nombre	Identificación
Semáforo	Llave ***key_t***	Identificador
Socket – dominio de UNIX	Nombre de ruta	Descriptor de archivo
Socket – otros dominios	Dependiendo del dominio	Descriptor de archivo

Una llave es una variable o constante del tipo *key_t* que se usa para acceder a los mecanismos IPC previamente reservados o para reservar nuevos, típicamente es un entero de 32 bits. En GNU C, se crean las llaves utilizando la función *ftok*, la cuál utiliza el nombre de una ruta y un identificador para crear un mecanismo de acceso. Se debe tener en consideración que los mecanismos que están siendo utilizados como parte de un mismo proyecto comparten la misma llave. El prototipo de la función *ftok* es el siguiente:

PROTOTIPO DE LA FUNCIÓN

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

key_t ftok(const char *pathname, int proj_id);
```

La constante *pathname* es el nombre de un archivo ordinario (se coloca la ruta completa donde se encuentra dicho archivo existente y accesible). El entero *proj_id* se usa junto con *pathname* para generar la llave, en caso de error la función retorna un -1.

La implementación de la función *ftok* es la siguiente: se combina los 8 bits menos significativos de *proj_id* con el número del i-nodo del archivo especificado en *pathname*, junto con el número menor del dispositivo del sistema de archivo en el cual reside el archivo en el disco, la combinación de estos tres valores producen una llave única de 32 bits. Después de generar dicha llave, esta puede ser usada por los llamados a *semget()*, *shmget()* y *msgget()*.

3.2.2 Semáforos en derivados de System V

Los semáforos son mecanismos utilizados por los sistemas para sincronizar los accesos a recursos compartidos. La dificultad que se presenta en la implementación de los semáforos es hacer que la operación de incremento o decremento sea atómica. Lo anterior es resuelto por los sistemas (por ejemplo los derivados del System V) implementandolos en el kernel, de está manera se garantiza que el grupo de operaciones de los semáforos se haga de forma atómica. En los sistemas V la implementación va en dos direcciones:

- Un semáforo no es un simple valor, pero si es un conjunto de valores enteros no negativos.

- Cada valor en el conjunto no es restringido a cero o a uno. Cada valor en el conjunto puede asumir un valor no negativo, el sistema tiene definido un número máximo que se puede otorgar.

Para cada conjunto de semáforos en el sistema, el kernel mantiene la siguiente estructura (definidas en <sys/types.h>, <sys/ipc.h>):

```
struct semid_ds {
  struct ipc_perm     sem_perm;   // estructura de permisos de operación
  struct sem          *sem_base;  // apuntador al primer semáforo del conjunto
  ushort              sem_nsems;  // número de semáforos que forman al conjunto
  time_t              sem_otime;  // tiempo de la última operación sobre el semáforo
  time_t              sem_ctime;  // tiempo del último cambio del semáforo
};
```

La estructura *sem* es la estructura de datos interna usada por el kernel para mantener el conjunto de valores asignados al semáforo, en donde cada miembro del conjunto de semáforos se describe dentro de los campos de la estructura:

```
struct sem {
        unsigned short      semval;     // valor, no negativo, del semáforo
        short               sempid;     // pid del proceso que realizó la última operación
        ushort              semncnt;    // contador del número de procesos que esperan que
                                        // el valor se incremente (semval > cval)
        ushort              semzct;     // contador del número de procesos que esperan que
                                        // el valor llegue a cero
};
```

En la siguiente figura 3-4, se muestra en forma gráfica la relación de las anteriores dos estructuras en el kernel.

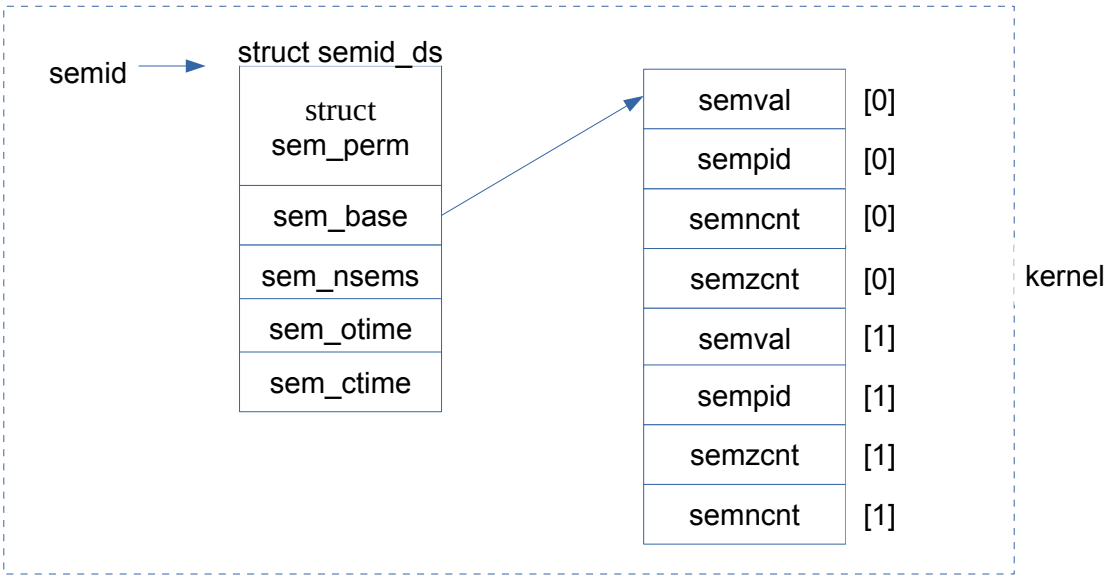


Figura 3-4. Estructuras de datos en el kernel para un conjunto de semáforos.[2]

A continuación se mostrarán los llamados al sistema para crear/accesar a un semáforo.

3.2.2.1 Llamados a funciones relacionadas con semáforo

El sistema de llamado *semget()* permite crear o accesar a un conjunto de semáforos unidos bajo una llave en común. El prototipo de la función es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/sem.h>

        int semget (key_t key, int nsems, int semflg);
```

Si la llamada se ejecuta correctamente retorna el identificador de tipo entero que servirá para accesar a los semáforos, en caso de error retornará un -1 y en *errno* el código del error producido. En este caso, la variable *errno* toma uno de los valores siguientes:

- EACCES. Existe un grupo de semáforos para la clave, pero el proceso no tiene todos los derechos necesarios para acceder al grupo.
- EEXIST. Existe un grupo de semáforos para la clave, pero están activadas las operaciones IPC_CREAT e IPC_EXCL.
- EIDRM. El grupo de semáforos ha sido borrado.
- ENOENT. El grupo de semáforos no existe y no está activada la opción IPC_CREAT.
- ENOMEN. El semáforo podría crearse pero el sistema no tiene más memoria para almacenar la estructura.
- ENOSPC. El semáforo podría crearse pero se sobrepasarían los límites para el número máximo de grupos (SEMMNI) de semáforos o el número de semáforos (SEMMNS).

La variable *key*, previamente retornada por la función *ftok*, sirve para indicar a qué grupo de semáforos se va a accesar. El núcleo busca dentro de la tabla alguna entrada que se ajuste a la llave suministrada. Si la llave suministrada toma el valor IPC_PRIVATE, el núcleo ocupa la primera entrada que se encuentra libre y ninguna otra llamada *semget()* podrá devolver esta entrada hasta que sea liberada.

La variable *nsems* sirve para indicar el total de semáforos que van a estar agrupados bajo el identificador devuelto.

La variable *semflg* es una máscara de bits para indicar el modo de adquisición del identificador, formada por dos campos separados por un el simbolo (|). Los campos son: IPC_CREAT, IPC_EXCL, para el primer campo, y un número de la forma 00X00 usuario, 000X0 grupo, 0000x otro, para el segundo campo.

2 Basada en la imagen de página 139 del libro Unix Network Programming de W. Richard Stevens. 1990.

Después de crear el semáforo o grupo de semáforos, se necesita realizar operación sobre ellos para bloquear o desbloquear la entrada al recurso compartido, para esto se utiliza el llamado a la función *semop()*. Dicha función es utilizada para realizar las operaciones atómicas (incremento, decremento o espera de nulidad) sobre los semáforos asociados bajo un identificador. El prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

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

int semop (int semid, struct sembuf  *sops, size_t nsops);
```

La variable *semid* es el identificador del semáforo (grupo sobre el cual las operaciones tendrán lugar). La variable *sops* es un apuntador al arreglo de operaciones (arreglo de estructuras sembuf que indica las operaciones a llevar a cabo sobre los semáforos). La variable *nsops* indica el número de operaciones a realizar en esta llamada (en realidad, tamaño de elementos que tiene el arreglo de operaciones).

La estructura involucrada en el llamado a semop es *sembuf*. Cada dato de este tipo especifica una operación a realizar sobre un semáforo particular dentro del conjunto. Este semáforo en particular es el especificado en *sem_num* (los números de semáforos van de 0 a n-1). En *sem_op* se especifica la operación a realizar sobre el semáforo. Este semáforo se puede incrementar, decrementar o esperar un valor nulo.

```
struct sembuf {
        unsigned short sem_num;   // número de semáforo
        short          sem_op;    // operación sobre el semáforo
        short          sem_flg;   // banderas de operación
        }
```

Si *sem_op* es negativo, el valor del semáforo se decrementará (operación P). Si *sem_op* es positivo, el valor del semáforo se incrementará (operación V), y tiene como efecto el despertar de todos los procesos que esperan a que este valor aumente a dicho valor. Si *sem_op* es 0, no hay alteración sobre el semáforo. Si es Nulo se bloquea el proceso. Las banderas reconocidas para el parámetro *sem_flg* son IPC_NOWAIT y SEM_UNDO.

Todas las operaciones deben efectuarse de manera atómica. En los casos en que esto no sea posible, o bien el proceso se suspende hasta que sea posible, o bien, si el identificador IPC_NOWAIT está activado (campo *sem_flg* de la estructura sembuf) por una de las operaciones, la llamada al sistema se interrumpe sin que se realice ninguna operación. Por cada operación efectuada, el sistema controla si el indicador SEM_UNDO está posicionado. Si es así, crea una estructura para

conservar el rastro de esa operación para poder anular y finalizar la ejecución del proceso.

Si la llamada al sistema *semop* se desarrolla correctamente el valor devuelto es un 0, si el proceso debe bloquearse devuelve un 1, en caso de error retorna un –1 y en *errno* se coloca uno de los siguientes valores:

- E2BIG. El argumento *nsops* es mayor que SEMOPM, y se sobrepasa el número máximo de operaciones autorizadas para una llamada.
- EACCES. El proceso que llama no tiene los derechos de acceso a uno de los semáforos especificados en una de las operaciones.
- EAGAIN. El indicador IPC_NOWAIT está activado y las operaciones no han podido realizarse inmediatamente.
- EFAULT. La dirección especificada por el campo *sops* no es válida.
- EFBIG. El número del semáforo (campo sem_num) es incorrecto para una de las operaciones (negativo o superior al número de semáforos en el grupo).
- EIDRM. El semáforo no existe.
- EINTR. El proceso ha recibido una señal cuando esperaba el acceso a un semáforo.
- EINVAL. El grupo del semáforo solicitado no existe (argumento semid), o bien el número de operaciones a realizar es negativo o nulo (argumento nsops).
- ENOMEN. El indicador SEM_UNDO está activado y el sistema no puede asignar memoria para almacenar la estructura de anulación.
- ERANGE. El valor añadido al contador del semáforo sobrepasa el valor máximo autorizado para el contador SEMVMX.

Por otra parte, se necesita administrar el semáforo, por lo que la función *semctl* permite inicializar, consultar, modificar y eliminar un grupo de semáforos. El prototipo de la función es:

PROTOTIPO DE LA FUNCIÓN

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

int semctl (int semid, int semnum, int cmd, union semun arg);
```

La variable *semid* especifica el identificador de semáforo válido. La variable semnum representa el número de semáforos, y la variable *cmd* indica la operación a realizar. Según los valores del parámetro *cmd* se puede realizarán alguno de los siguientes casos:

- IPC_STAT. Permite obtener las informaciones respecto a un grupo de semáforo, el campo semmun se ignora, arg es un apuntador a la zona que contiene las informaciones. El proceso debe tener derechos de lectura para realizar la operación.
- IPC_SET. Permite modificar ciertos valores de la estructura del grupo de semáforos. Los campos modificables son sem_perm.uid, sem_perm.gid y

sem_perm.mode. El campo sem_ctime se actualiza automáticamente. El proceso debe ser el creador o bien el propietario del grupo, o bien el superusuario.

- IPC_RMID. Indica al núcleo que debe borrar el conjunto de semáforos agrupados bajo el identificador semid. La operación de borrado no tendrá efecto mientras haya algún proceso que esté usando los semáforos.
- GETPID. Permite devolver el PID del último proceso que actuó sobre el semáforo. El proceso debe tener derechos de acceso en lectura sobre el grupo.
- GETNCNT. Permite leer el número de procesos que esperan que el contador del semáforo se incremente. El proceso debe tener el derecho de acceso en lectura sobre el grupo.
- GETZCNT. Permite leer el número de procesos que esperan a que el semáforo tome el valor de 0. Este número es calculado por la función *count_semzcnt*.
- GETVAL. Se utiliza para leer el valor de un semáforo. El valor se retorna a través de la función.
- GETALL. Permite leer el valor de todos los semáforos asociados al identificador semid. Estos valores se almacenan en arg.
- SETVAL. Permite inicializar un semáforo a un valor determinado que se especifica en arg.
- SETALL. Inicializa el valor de todos los semáforos asociados al identificador semid. Los valores de inicialización deben estar en arg.

Para el parámetro *arg* se utiliza la estructura semun, la cual es una unión, utilizada para almacenar o recuperar informaciones de los semáforos.

```
union semun {
        int             val;    // Valor para SETVAL
        struct semid_ds *buf;   // buffer para IPC_STAT e IPC_SET
        ushort          *array; // Tabla para GETALL y SETALL
        struct seminfo  *__buf; // buffer para IPC_INFO en Linux
        } arg;
```

La estructura *semid_ds*, es una estructura de control asociada a cada uno de los distintos conjuntos de semáforos existentes en el sistema. Contiene información del sistema, apuntadores a las operaciones a realizar sobre el grupo, y un apuntador hacia las estructuras *sem* que se almacenan en el kernel y contienen información de cada semáforo. Esta estructura está en desuso actualmente y ha sido sustituida por sem_array, sólo se utiliza para compatibilidades.

La estructura *seminfo* permite conocer los valores límite o actuales del sistema mediante una llamada a *semctl*. Estas llamadas no se realizan generalmente en modo directo, sino que están reservadas a las utilidades del sistema como para el comando *ipcs*.

Si la llamada a semctl se realiza con éxito, la función devuelve un número cuyo significado dependerá del valor de *cmd*. Si el llamado falla devuelve un -1 y errno tendrá el código de error:

- EACCES. El proceso que llama no tiene derechos de acceso necesarios para realizar la operación.
- EFAULT. La dirección especificada por arg.buf o arg.array no es válida.

- EIDRM. El grupo de semáforos ha sido borrado.
- EINVAL. El valor de semid o de cmd es incorrecto.
- EPERM. El proceso no tiene los derechos necesarios para la operación (mandato IPC_SET o IPC_RMID).
- ERANGE. El cmd tiene el valor SETALL o SETVAL y el valor del contador de uno de los semáforos a modificar es inferior o superior a SEMVMX.

Ejemplo. Sincronización de procesos.

```
/****  Programa para mostrar el uso de semáforos en la sincronización de procesos
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include <errno.h>

#define SEM_HIJO 0
#define SEM_PADRE 1
int main (int argc, char *argv[])
{
  int i=1000000, semid;
  pid_t pid;
  struct sembuf operacion;
  key_t llave;

  llave= ftok (argv[0],'a');
  if ( (semid = semget (llave,2,IPC_CREAT|0600))==-1)
  {
    perror ("semget");
    exit (-1);
  }
 semctl (semid, SEM_HIJO, SETVAL, 0);//cerrar semáforo del hijo
 semctl (semid, SEM_PADRE, SETVAL, 1);//abrir semáforo del padre
 if ( (pid = fork())==-1)
 {
    perror ("fork");
    exit (-1);
  }
  else if (pid==0)
  {
    while (i)
    {
     // Cerrar el semáforo del proceso hijo
      operacion.sem_num = SEM_HIJO;
      operacion.sem_op = -1;
```

```
      operacion.sem_flg = 0;
      semop (semid, &operacion, 1);
     printf ("Proceso hijo:%d\n", i--);
    // Abrir el semáforo del proceso padre
      operacion.sem_num = SEM_PADRE;
      operacion.sem_op = 1;
      semop (semid, &operacion, 1);
    }
     // Borrar el semáforo
     semctl (semid, 0, IPC_RMID,0);
  }
  else
  {
  while (i)
    {
     // Cerrar el semáforo del proceso padre
      operacion.sem_num = SEM_PADRE;
      operacion.sem_op = -1;
      operacion.sem_flg = 0;
      semop (semid, &operacion, 1);
     printf ("Proceso padre:%d\n", i--);
    // Abrir el semáforo del proceso hijo
      operacion.sem_num = SEM_HIJO;
      operacion.sem_op = 1;
      semop (semid, &operacion, 1);
    }
  // Borrar el semáforo
     semctl (semid, 0, IPC_RMID,0);
  }
  return EXIT_SUCCESS;
}
```

3.2.3 Semáforos en POSIX

El sistema de llamado *sem_init()* permite iniciar un semáforo sin nombre en una cierta dirección de memoria (para semáforos con nombre se usa la función *sem_open*). El prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

El parámetro *sem*, debe ser la dirección de la variable de tipo semáforo (sem_t) que se va a manipular. El argumento *value* especifica el valor inicial del semáforo. El argumento *pshared* indica si este semáforo se compartirá entre hilos de un proceso o entre procesos. Si *pshared* tiene el valor 0, entonces el semáforo se

comparte entre los hilos de un proceso y debe ubicarse en alguna dirección que sea visible para todos los hilos (por ejemplo, una variable global o una variable asignada dinámicamente la cabecera). Si *pshared* es distinto de cero, entonces el semáforo se comparte entre procesos y debe ubicarse en una región de memoria compartida (usando para su creación la función *shm_open*, o la función *shmget()*). Cualquier proceso que pueda acceder a la región de memoria compartida puede operar en el semáforo usando *sem_post()* y *sem_wait()*. Recuerde que si se crea un hijo con *fork()*, el hijo hereda las asignaciones de memoria de su padre, y también puede acceder al semáforo.

El sistema de llamado *sem_wait()* permite cerrar un semáforo. El prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

El parámetro *sem* es la dirección de la variable semáforo a manipular. La función se encarga de decrementar (bloquear) el valor apuntado por la variable semáforo. Si el valor es mayor que cero, entonces se decrementa y la función retorna de forma inmediata, si el valor es cero, entonces la función se bloquea hasta que no sea 0, y pueda decrementar o sea interrumpido por alguna señal.

El sistema de llamado *sem_post()* permite abrir un semáforo. El prototipo es:

PROTOTIPO DE LA FUNCIÓN

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

El parámetro *sem* es la dirección de la variable semáforo a manipular. La función se encarga de incrementar (desbloquear) el valor apuntado por la variable semáforo. Si el valor se vuelve mayor que cero entonces otro proceso bloqueado por la llamada a *sem_wait* se despierta y se desbloquea.

Ejemplo. En el programa se crean dos hilos que manipulan una variable global. Dicha variable es protegida por un semáforo para que cada hilo que la utilice no afecte al otro. Por lo que uno incrementa y otro decrementa, y el resultado final será 0 en la variable contador. Se recomienda probar el código sin el semáforo para que observe que el valor de la variable global (contador) cambia cada vez que ejecutamos el programa.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#define VALOR 1000
void *funcion1 (void *valor);
void *funcion2 (void *valor);
```

```
int contador=0;
sem_t semaforo;
int main ()
{
 pthread_t  hilo1,hilo2;
 pthread_attr_t attr;
 sem_init(&semaforo, 0, 1);//el 0 indica que el semaforo será compartida entre hilos de un proceso
 pthread_attr_init(&attr);
 pthread_create(&hilo1,&attr,funcion1,NULL);
 pthread_create(&hilo2,&attr,funcion2,NULL);
 pthread_join(hilo1,NULL);
 pthread_join(hilo2,NULL);
 printf("Valor de Contador=%d\n",contador);
 return EXIT_SUCCESS;
}

void *funcion1 (void *valor)
{
   for (int i=0;i<VALOR;i++)
  {
    sem_wait(&semaforo); // Decrementa variable del semáforo (bloquea)
     contador+=1;
    sem_post(&semaforo);// Incrementa variable del semáforo (desbloquea)
  }
 pthread_exit(0);
}
void *funcion2 (void *valor)
{
  for (int i=0;i<VALOR;i++)
  {
    sem_wait(&semaforo); // Decrementa variable del semáforo (bloquea)
      contador-=1;
    sem_post(&semaforo); // Incrementa variable del semáforo (desbloquea)
  }
 pthread_exit(0);
}
```

3.2.3.1 Sincronización de hilos usando mutex

Los **mutex** (***mut***ual ***ex***clusion) son mecanismos de sincronización a nivel hilos, y se comportan como semáforos binarios para proteger un recurso compartido entre hilos. Usar variables mutex en hilos es bastante sencillo, y los pasos a seguir son:
1. Cree e inicializa un mutex para cada recurso que se desee proteger, llamando a la función ***pthread_mutex_init***.
2. Cuando un hilo deba acceder al recurso compartido, se debe utilizar la función ***pthread_mutex_lock*** para bloquear y llevar a cabo la exclusión mutua del recurso. La biblioteca pthread se asegura de que sólo un hilo a la vez pueda bloquear el mutex; todas las demás llamadas a la función ***pthread_mutex_lock*** para el mismo mutex deben esperar hasta que el hilo que actualmente contiene el mutex lo libere.

3. Cuando el hilo haya terminado de utilizar el recurso compartido, se desbloquea el mutex llamando a la función ***pthread_mutex_unlock***.

Los prototipos de las funciones, anteriormente mencionadas, son los siguientes:

PROTOTIPO DE LA FUNCIÓN
```
#include <pthread.h>
pthread_mutex_t mutex;

int pthread_mutex_init (pthread_rautex_t *mutex,
ptheread_mutexattr_t *attr);
```

PROTOTIPO DE LA FUNCIÓN
```
#include <pthread.h>

int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

PROTOTIPO DE LA FUNCIÓN
```
#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);
```

PROTOTIPO DE LA FUNCIÓN
```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Ejemplo. En el siguiente código se crean un par de hilos para llevar a cabo exclusión mutua utilizando las funciones mutex.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
void *funcion1 (void *valor);
void *funcion2 (void *valor);
pthread_mutex_t EM;
int main (int argc, char *argv[])
{
 pthread_t hilo1,hilo2;
 pthread_attr_t attr;

 // Coloca atributo predeterminados o coloque NULL
 pthread_attr_init(&attr);
```

```
 // Crear hilos
 pthread_create(&hilo1, &attr, funcion1, NULL);
 pthread_create(&hilo2, &attr, funcion2, NULL);
 //Espera hilos
 pthread_join(hilo1, NULL);
 pthread_join(hilo2, NULL);
 return EXIT_SUCCESS;
 }

void *funcion1 (void *valor)
{
 // Enciende Semáforo (INICIO Exclusión) para indicar que entra a SC
  pthread_mutex_lock(&EM);
  // Inicio de Sección Crítica
  /*  COLOCAR CÓDIGO
  *****  Para el uso del
   *****  Recurso Compartido  */
  // Fin de Sección Crítica
  // Apaga Semáforo (FIN Exclusión) para indicar que sale de SC
 pthread_mutex_unlock(&EM);
 }
 pthread_exit(0);
}

void *funcion2 (void *valor)
{
 // Enciende Semáforo (INICIO Exclusión) para indicar que entra a SC
 pthread_mutex_lock(&EM);
// Inicio de Sección Crítica
  /*  COLOCAR CÓDIGO
  *****  Para el uso del
   *****  Recurso Compartido  */
  // Fin de Sección Crítica
  // Apaga Semáforo (FIN Exclusión) para indicar que sale de SC
  pthread_mutex_unlock(&EM);
 }
 pthread_exit(0);
}
```

3.3 Memoria compartida

La forma más rápida para comunicar dos procesos es hacer que compartan una zona de memoria. La función **shmget** crea una zona de memoria compartida o habilita el acceso a una ya creada. El prototipo de la función es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/types.h> // no requerida en Linux
        #include <sys/ipc.h>
        #include <sys/shm.h>

        int shmget ( key_t key, size_t size, int shmflg);
```

Los parámetros de la función son, *key,* que es una llave para crear el mecanismo IPC, *size,* que es el tamaño en bytes de la zona de memoria a crear, y *shmflg,* que es la máscara de bits para especificar los atributos de la zona de memoria. Si el valor de shmflg está compuesto por IPC_CREAT, indica que crea un nuevo segmento de memoria, si no es usada, entonces encuentra el segmento asociado con *key* y se verifica si el usuario tiene permiso para accesar al segmento. La bandera IPC_EXCL es utilizada junto con IPC_CREAT para garantizar que el segmento sea creado.

La llamada a *shmget* devuelve un identificador asociado a la zona de memoria (entero), en otro caso, retorna un -1, y en errno el código del error. Por ejemplo:

```
if ( ( shmid = shmget (IPC_PRIVATE, 4096, IPC_CREAT | 0600)) == -1)
```

La función *shmctl* realiza operaciones de control sobre una zona de memoria previamente creada. El prototipo de la función es:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/types.h> // no requerida en Linux
        #include <sys/ipc.h>
        #include <sys/shm.h>

        int shmctl (int shmid, int cmd, struct  shmid_ds  *buf);
```

El parámetro *shmid* debe ser el identificador válido devuelto por *shmget*. El parámetro *cmd* es el tipo de operación de control a realizar, sus posibles valores son:

- IPC_STAT. Lee el estado de la estructura de control de la memoria y lo devuelve a través de la zona de memoria apuntada por buf.
- IPC_SET. Inicializa algunos de los campos de la estructura de control de la memoria compartida. Los valores los toma de buf.
- IPC_RMID. Borra del sistema la zona de memoria compartida identificada por shmid. Está debe estar liberada por todos los procesos.
- SHM_LOCK. Bloquea en memoria el segmento identificado por shmid.
- SHM_UNLOCK. Desbloquea el segmento de memoria compartido.

La estructura *shmid_ds* se define como:

```
struct shmid_ds
{
  struct ipc_perm       shm_perm;    /* Datos del propietario y permisos del
segmento*/
  size_t          shm_segsz;    /* Tamaño del segmeto de memoria  */
```

```
  pid_t           shm_lpid;     /* PID del último proceso que hizo la última operación
con                                   este segmento de memoria */
 pid_t            shm_cpid;     /* PID del proceso creador del segmento */
 shmatt_t         shm_nattach;  /* Número de procesos unidos al seg. de memoria*/
 time_t           shm_atime;    /* Fecha de la última unión al seg. de memoria */
 time_t           shm_dtime;    /* Fecha de la última separación del seg. de
memoria*/
 time_t           shm_ctime;    /* Fecha del último cambio en el seg. de memoria*/
}
```

La estructura ipc_perm (permisos) es la siguiente:

```
        struct ipc_perm {
                keyt_t          __key; /* Llave */
                uid_t           uid;    /* ID del usuario propietario */
                gid_t           gid;    /* ID del grupo */
                uid_t           cuid;   /* ID del usuario creador */
                gid_t           cgid;   /* ID del grupo creador */
                unsigned short mode; /* Modos de acceso */
                unsigned short __seg; /* Número de secuencia */
                }
```

Para borrar del sistema la zona de memoria compartida se debe escribir:

shmctl (shmid, IPC_RMID, 0);

Para unirse/atarse a un espacio de direcciones virtuales (segmento de memoria compartida) del proceso se utiliza la función *shmat* y para desatarse se invoca a la función *shmdt*. Los prototipos de las funciones son:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <sys/ipc.h> //No requerida en Linux
#include <sys/shm.h>

char *shmat (int shmid, char *shmaddr, int shmflg);
int shmdt (char *shmaddr);
```

El llamado a la función *shmat* retorna la dirección inicial del segmento de memoria compartida. Las reglas para determinar estas direcciones son:

- Si el argumento *shmaddr* es cero, el sistema selecciona la dirección para el llamado.
- Si el argumento *shmaddr* no es cero, retorna la dirección dependiendo del valor de SHM_RND para el argumento *shmflg*:
  - Si el valor SHM_RND no es especificado, el segmento de memoria compartida es unida a la dirección especificada como argumento en shmaddr.
  - Si el valor de SHM_RND es especificado, el segmento de memoria compartida es unida a la dirección especificada como argumento en shmaddr, redondeando hacia abajo por la constante SHMLBA (Lower Boundary Address).

3.4 Cola de mensajes

El llamado a la función *msgget* permite retornar un identificador a la cola, el cual está asociado con el valor del argumento key. El prototipo de la función es:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/msg.h>

        int msgget ( key_t key, int msgflg);
```

Si se coloca IPC_PRIVATE en el parámetro key, se crea una nueva cola de mensajes, si no se coloca, y se usa la llave devuelta por *ftok*, se debe colocar en *msgflg* IPC_CREAT, junto con el modo de acceso.

El llamado a la función *msgctl* permite administrar la cola de mensajes, su prototipo es:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/types.h>
        #include <sys/ipc.h>
        #include <sys/msg.h>

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

El parámetro *cmd* permite especificar el tipo de operación a realizar sobre la cola de mensajes indicada en *msqid*. Los datos administrativos de la cola se encuentran en la estructura *msqid_ds*, cuyos campos son:

```
struct msqid_ds {
     struct ipc_perm   msg_perm;      /* Datos del propietario y permisos */
     time_t            msg_stime;     /* Último tiempo del mensaje escrito*/
     time_t            msg_rtime;     /* Último tiempo del mensaje leido */
     time_t            msg_ctime;     /* Último tiempo  de cambio*/
          unsigned long     __msg_cbytes; /* Número actual de bytes en la cola
(noestandard) */
     msgqnum_t         msg_qnum;      /* Número actual de mensajes en la cola */
     msglen_t          msg_qbytes;    /* Número máximo de bytes permitidos en la
cola */
     pid_t             msg_lspid;     /*Último PID del proceso que escribió mensaje
msgsnd*/
     pid_t             msg_lrpid;     /* Último PID del proceso que leyó mensaje
msgrcv*/
     };
```

La estructura *ipc_perm* está definida como sigue:

```
struct ipc_perm {
   key_t        __key;          /* Llave suministrada en msgget*/
   uid_t        uid;            /* UID del propietario */
   gid_t        gid;            /* GID del propietario */
   uid_t        cuid;           /* UID del creador */
   gid_t        cgid;           /* GID del creador */
   unsigned short mode;         /* Permisos */
   unsigned short __seq;        /* Número de secuencia */
};
```

Los valores de cmd pueden ser:

- IPC_STAT. Copia la información de la estructura de datos del kernel asociada con la cola indicada en msqid en la estructura msqid_ds apuntada por buf. El proceso que invoca debe tener permiso de lectura sobre la cola de mensajes.
- IPC_SET. Escribe los valores de algunos campos de la estructura msqid_ds apuntada por buf en la estructura de datos del kernel asociada con la cola de mensajes, actualizando también el campo msg_ctime. Así como también actualiza los campos: msg_qbytes, msg_perm.uid, msg_perm.gid, y msg_perm.mode (los 9 bits menos significativos). El UID efectivo del proceso de llamada debe coincidir con el propietario (msg_perm.uid) o el creador (msg_perm.cuid) de la cola de mensajes, o el proceso que invoca debe tener privilegios.
- IPC_RMID. Elimina inmediatamente la cola de mensajes, despertando todos los procesos de lectura y escritura en espera (con un retorno de error y errno establecido en EIDRM). El proceso que invoca debe tener los privilegios adecuados o su ID de usuario efectivo debe ser el del creador o propietario de la cola de mensajes. El tercer argumento de msgctl() se ignora en este caso. Por ejemplo para el borrado de la cola de mensajes se debe escribir: msgctl (msgid, IPC_RMID, NULL);

Los llamados a las funciones *msgsnd* y *msgrcv* perrmiten escribir (enviar) y leer (recibir) respectivamente mensajes en la cola. Sus prototipos son los siguientes:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>   // No requerido en Linux
#include <sys/ipc.h>     // No requerido en Linux
#include <sys/msg.h>

int msgsnd (int msqid, const void *msgp, size_t msgsz,
int msgflg);
```

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>   // No requerido en Linux
#include <sys/ipc.h>     // No requerido en Linux
#include <sys/msg.h>

ssize_t msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp,
    int msgflg);
```

Los procesos que invocan deben tener permisos de escritura de mensajes sobre la cola, si envían, y permisos de lectura, si reciben mensajes.

El parámetro *msgp* es un apuntador a una estructura de la forma siguiente:

```
struct msgbuf {
   long mtype;      /* Tipo de mensaje, debe ser  > 0 */
   char mtext[i];   /* Datos del mensaje de longitud i */
};
```

El campo *mtext* es un arreglo cuyo tamaño está especificado por msgsz, que es un entero no negativo. El campo *mtype* debe ser estrictamete un valor entero positivo. Este valor puede ser usado para que los procesos que reciben los mensajes puedan seleccionar cual van a leer.

El llamado a *msgsnd* añade una copia del mensaje apuntado por *msgp* a la cola de mensajes cuyo identificador es *msqid*. Si hay suficiente espacio en la cola, se realiza con éxito, si no hay suficiente espacio en la cola el llamado es bloqueado hasta que exista el espacio suficiente.

El llamado a *msgrcv* remueve un mensaje de la cola especificada por *msqid* y lo coloca en el buffer apuntado por *msgp*. El argumento *msgsz* especifica el tamaño máximo en bytes del campo *mtext* de la estructura apuntada por el parámetro *msgp*. Si el mensaje es más grande que msgsz, entonces el comportamiento depende si se especificó MSG_NOERROR en *msgflg*. Si fue especificado entonces el texto del mensaje es truncado, si no fue entonces el llamado retorna un -1 y en errno E2BIG.

El parámetro *msgtyp* se utiliza para indicar el tipo de mensaje solicitado:

- Si es 0, entonces se lee el primer mensaje que se encuentra en la cola.
- Si es mayor que 0, entonces se lee el primer mensaje de tipo *msgtyp* que se encuentra en la cola.
- Si es menor que 0, entonces se lee el primer mensaje con el tipo menor que o igual al valor absoluto de *msgtyp*.

El parámetro msgflg es una máscara de bit construida al combinar cero o más de los siguientes indicadores:

- IPC_NOWAIT. Regresa de forma inmediata si no hay mensaje del tipo solicitado dentro de la cola. Si la llamada falla retorna errno con ENOMSG.
- MSG_COPY (a partir de Linux 3.8) Obtiene de forma no destructiva una copia del mensaje en la posición ordinal en la cola especificada por *msgtyp* (los mensajes se consideran numerados a partir de 0). Esta bandera debe

especificarse junto con IPC_NOWAIT, con el resultado de que, si no hay un mensaje disponible en la posición dada, la llamada falla inmediatamente con el error ENOMSG.

- MSG_EXCEPT. Usado con *msgtyp* mayor que 0 para leer el primer mensaje en la cola con un tipo de mensaje que difiere de *msgtyp*.
- MSG_NOERROR. Trunca el texto del mensaje si es más largo que *msgsz* bytes.

Ejemplo. El siguiente código envía/escribe un mensaje con el tiempo del sistema en la cola, y se encarga de recibir/leer dicho mensaje de la cola. Se usa con el nombre del ejecutable y s o r, respectivamente. El código está basado en el manual de GNU/Linux (2017-09-15).

```
/*    ***************************************************************************
******* compilar: gcc -Wall mcola2.c -o mcola
*******  Ejecutar para enviar/escribir en la cola: ./mcola s
*******  Ejecutar para recibir/leer de la cola: ./mcola r
*************************************************************************** */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include <sys/msg.h>
void send_msg(int,int );
void get_msg(int, int);
struct msgbuf {
   long mtype;
   char mtext[80];
  };
int main(int argc, char *argv[])
{
 int qid;
 int modo;              // 1 = enviar/escribir, 2 = recibir/leer
 int msgtype = 1;
 int llave;

 llave=ftok (argv[0],'a');
  if (argc>1)
  {
    if (strcmp(argv[1],"s")==0) modo=1;
     else if (strcmp(argv[1],"r")==0) modo=2;
          else { printf ("Use: ./mcola s|r\n");
            exit(EXIT_FAILURE);
           }
    } else {
        printf ("Use: ./mcola s|r\n");
        exit(EXIT_FAILURE);
    }
  // Crea memoria compartida
  if ( (qid = msgget(llave, IPC_CREAT | 0666))==-1)
```

```
  {
   perror("msgget");
   exit(EXIT_FAILURE);
  }
  if (modo == 2) get_msg(qid, msgtype);
    else send_msg(qid, msgtype);
  return(EXIT_SUCCESS);
 }

void send_msg(int qid, int msgtype)
{
  struct msgbuf msg;
  time_t t;
   msg.mtype = msgtype;
   time(&t);
   snprintf(msg.mtext, sizeof(msg.mtext), "El mensaje salió el: %s", ctime(&t));
   // Enviar/Escribir en la cola
   if (msgsnd(qid, (void *) &msg, sizeof(msg.mtext), IPC_NOWAIT) == -1)
   {
     perror("ERROR en msgsnd");
     exit(EXIT_FAILURE);
    }
    printf("Mensaje enviado: %s\n", msg.mtext);
   }

void get_msg(int qid, int msgtype)
{
 struct msgbuf msg;
// Recibir/Leer de la cola
if      (msgrcv(qid,(void*)&msg,sizeof(msg.mtext),msgtype,MSG_NOERROR|IPC_NOWAIT)
== -1)
  {
   if (errno != ENOMSG)
   {
    perror("ERROR en msgrcv");
    exit(EXIT_FAILURE);
   }
   printf("No hay mensajes disponibles para leer/recibir con msgrcv()\n");
   } else printf("Mensaje recibido: %s\n", msg.mtext);
  }
```

Capítulo 4

Interbloqueo e Inanición

Un proceso pueden ocasionar un bloqueo a otros procesos si realiza una mala acción en la sección crítica, es decir, si retiene o se apropia de los recursos que son compartidos con otros procesos.

Los enfoques que se pueden adoptar en los sistemas operativos para tratar el problema de interbloqueo son prevención, detección y predicción. Pero antes de tocar cada uno de ellos, se presenta una definición de Interbloqueo, proporcionada por (Maekawa, 1987):

> Un interbloqueo se define como el bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o bien se comunican unos con otros.

4.1 Inanición, aplazamiento indefinido, bloqueo indefinido

La inanición, aplazamiento indefinido o bloqueo indefinido, se presenta cuando un proceso, aún cuando no esté bloqueado, está esperando un evento que quizá nunca ocurra debido a ciertas tendencias en las políticas de asignación de recursos del sistema. En cualquier sistema que mantenga los procesos en espera mientras se les asigna un recurso o se toman decisiones de planificación, la programación de un proceso puede postergarse indefinidamente mientras otro recibe la atención del sistema.

En el trabajo de Coffman[3], se muestran las condiciones para que se presente un interloqueo en un sistema. Tres de estas condiciones son:

1. *Exclusión mutua*. Sólo un proceso puede usar un recurso simultáneamente.
2. *Retención y espera*. Un proceso puede retener unos recursos asignados mientras espera que se le asignen otros.
3. *No apropiación*. Ningún proceso puede ser forzado a abandonar un recurso que retenga.

En función de las solicitudes y liberación de los recursos por parte de los procesos, se puede producir la siguiente condición:

4. *Espera circular*. Existe una cadena cerrada de procesos, cada uno de los cuales retiene, al menos, un recurso que necesita el siguiente proceso de la cadena. (Figura 4-1).

Las cuatro condiciones en conjunto constituyen una condición necesaria y suficiente para el interbloqueo.

3 E. G. Coffman, M. J. Elphick, and A. Shoshani. Systems Deadlocks. ACM Computing Surveys, 3(2):67–78, 1971

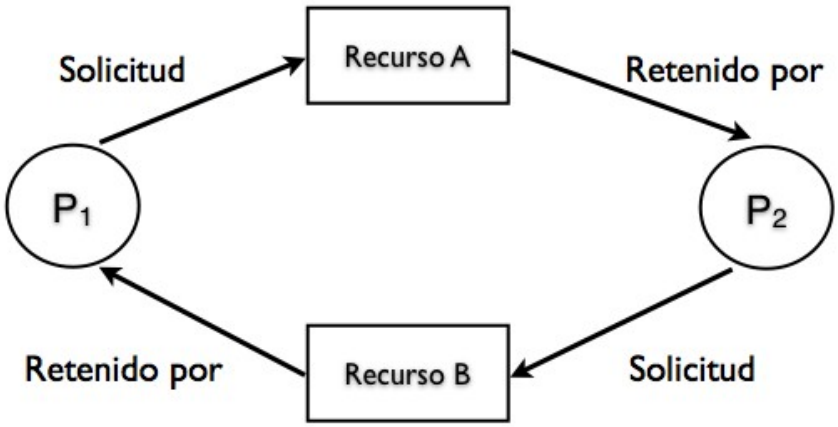


Figura 4-1. Espera circular.[4]

4.2 Prevención del bloqueo mutuo

Definitivamente la técnica empleada, con más frecuencia, por los diseñadores para tratar el problema del bloqueo mutuo es la prevención. Havender, J. W. llegó a la conclusión de que si falta una de las cuatro condiciones necesarias no puede haber bloqueo mutuo, él sugiere las siguientes estrategias para negar varias de estas condiciones:

- Cada proceso deberá pedir todos sus recursos al mismo tiempo y no podrá seguir la ejecución hasta haberlos recibido todos.
- Si a un proceso que tiene ciertos recursos se le niegan los demás, ese proceso deberá liberar sus recursos y, en caso necesario, pedirlos nuevamente junto con los recursos adicionales.
- Se impondrá un ordenamiento lineal de los tipos de recursos en todos los procesos, es decir, si a un proceso le han sido asignados recursos de un tipo específico, en lo sucesivo sólo podrá pedir aquellos recursos que siguen en el ordenamiento (Figura 4-2).

4 Figura tomada del libro:W.Stallings.Sistemas Opeativos. 2a. Ediciòn. 1997. p. 229

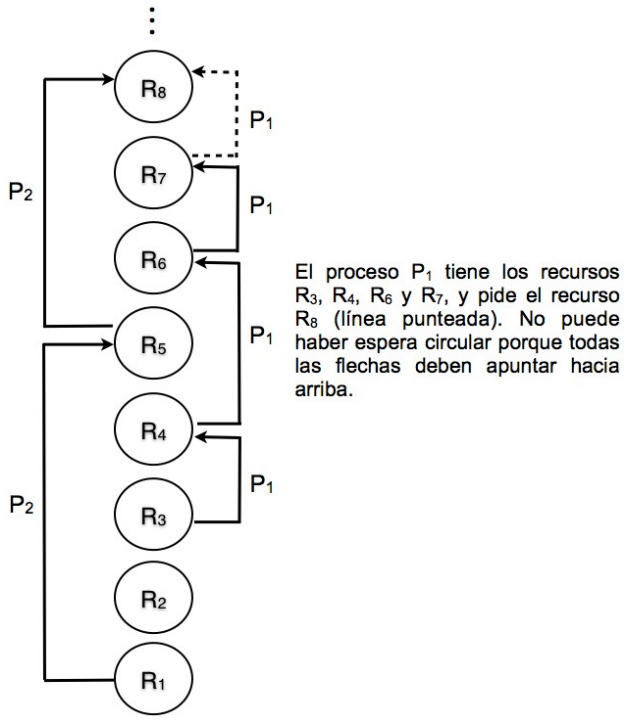


Figura 4.2. Ordenamiento lineal.[5]

4.3 Algoritmo del banquero

El algoritmo más famoso para evitar un bloqueo mutuo es el algoritmo del banquero planteado por Dijkstra en 1965, en el reporte *Cooperating Sequencial Processes*, cuyo interesante nombre se debe a que atañe de un banquero que otorga préstamos y recibe pagos a partir de una determinada fuente de capital.

Lo que planea Dijkstra es, un banco tiene un capital finito y acepta realizar préstamos a sus clientes bajo las siguientes condiciones:

1. Cada cliente realiza un préstamo para una transacción que será completada en un tiempo finito.
2. El cliente deberá especificar por adelantado sus necesidades máximas para sus transacciones.
3. Siempre que el préstamo no exceda la necesidad que indicó por adelantado, el cliente puede aumentar o disminuir su préstamo.
4. Un cliente no puede quejarse si solicita un incremento al préstamo actual, y el banco le dice que si se lo prestará, pero en un tiempo finito, debido a que no excede su necesidad declarada, pero por el momento no se lo prestará.
5. La garantía de que llegará el préstamo se basa en la cautela del banquero, y el hecho de que todos los clientes están sujetos a la misma condición: tan pronto como se le dé a un cliente el préstamo su transacción procederá en un periodo finito de tiempo, es decir, dentro de un tiempo finito pedirá un nuevo préstamo o devolverá el préstamo o finalizará la transacción, lo que implica la devolución completa del préstamo (uno por uno).

Dijkstra plantea las siguientes preguntas:

5 Figura tomada de: Deitel. Sistemas Operativos. 2a. Edición. 1993.

a) ¿Bajo qué condiciones puede el banco realizar un contrato con un nuevo cliente?
b) ¿Bajo qué condiciones puede el banquero prestar a un cliente solicitante sin correr el riesgo de un interbloqueo?

La respuesta de a) es simple: puede aceptar a cualquier cliente cuya declaración de necesidades no exceda el capital del banco.

Para responder a b), introduce la siguiente terminología.

- El banco tiene un capital fijo; cada cliente nuevo dice de antemano su necesidad máxima, y para el cliente se tendrá:
$$necesidad[i] \leq capital \qquad \forall i$$

- La situación actual de cada cliente está caracterizado por su préstamo. Cada préstamo es inicialmente=0 y se satisfará en cualquier instante:
$$0 \leq prestamo[i] \leq necesidad[i] \qquad \forall i$$

- Una cantidad útil se deriva de la "demanda" máxima dada por:
$$demanda[i] = necesidad[i] - prestamo[i] \qquad \forall i$$

- Se cuenta con una cantidad efectiva en el banco:
$$efectivo = capital - \sum_{i=1}^{N} prestamo[i]$$
$$0 \leq efectivo \leq capital$$

Para decidir si se puede prestar al cliente lo solicitado, el banco examina esencialmente la situación que se produciría si lo hubiese realizado. El algoritmo inicia a investigar si al menos un cliente tiene una demanda que no exceda el efectivo. Si es así, este cliente puede completar sus transacciones, y por lo tanto, el algoritmo investiga a los clientes restantes como si el primero hubiera terminado y devuelto su préstamo completo. La seguridad de la situación significa que todas las transacciones se pueden terminar, es decir, que el banquero ve una forma de recuperar todo su capital.

Ejemplo. Estado seguro.
Supóngase que un sistema tiene en capital 12 unidades de cinta y tres procesos que las comparten, como en el Estado I.

Estado I

proceso[i]	préstamo[i]	necesidad[i]	demanda[i]
P[1]	1	4	3
P[2]	4	6	2
P[3]	5	8	3
Efectivo		2	

Este es un estado “seguro” porque aún es posible que terminen los tres procesos. De esta forma, la clave para que un estado sea seguro es que exista al menos una forma de que terminen todos los procesos.

Ejemplo. Estado inseguro.
Supóngase que las 12 unidades de cinta de un sistema están asignadas como en el Estado II.

Estado II

proceso[i]	préstamo[i]	necesidad[i]	demanda[i]
P[1]	8	10	2
P[2]	2	5	3
P[3]	1	3	2
Efectivo		1	

Aquí, 11 de las 12 unidades de cinta del sistema se encuentran en préstamo y solamente 1 está disponible para asignación. En este momento, sin importar cuál de los procesos pida la unidad disponible, no se puede garantizar que terminen los tres procesos. Es importante señalar que un estado inseguro no implica la existencia, ni siquiera eventual, de un bloqueo mutuo. Lo que sí implica un estado inseguro es simplemente que alguna secuencia desafortunada de eventos podría llevar al bloqueo mutuo.

Ejemplo. Transición de estado seguro a estado inseguro.

Saber que un estado es seguro no implica que serán seguros todos los estados futuros. La política de asignación de recursos debe considerar cuidadosamente todas las peticiones antes de satisfacerlas. Por ejemplo, supóngase que el estado actual de un sistema como se muestra en el Estado III.

Estado III

proceso[i]	préstamo[i]	necesidad[i]	demanda[i]
P[1]	1	4	3
P[2]	4	6	2
P[3]	5	8	3
Efectivo		2	

Ahora supóngase que el P[3] pide un recurso más. Si el sistema satisface esta petición, el nuevo estado será el Estado IV.

Estado IV

proceso[i]	préstamo[i]	necesidad[i]	demanda[i]
P[1]	1	4	3
P[2]	4	6	2
P[3]	6	8	2

Efectivo	1

Ciertamente, el Estado IV no está necesariamente bloqueado, pero ha pasado de un estado seguro a uno inseguro. El estado IV caracteriza a un sistema en el cual no puede garantizarse la terminación de todos los procesos. Solamente hay un recurso disponible, pero deben estar disponibles al menos 2 recursos para asegurar que P[2] o P[3] puedan terminar, devolver sus recursos al sistema y permitir que los otros procesos terminen.

Dijkstra propone la siguiente subrutina para inspeccionar un estado seguro de los clientes numerados de 1 a N:

```
int dinero;
boolean  seguro, inseguro[1..N]

dinero=efectivo;
for(i=1;i<=N;i++)
        inseguro[i]=TRUE;
for(i=1;i<=N;i++)
        if  (inseguro[i]) && (demanda[i]<=dinero)
        {
          inseguro[i]=FALSE;
          dinero=dinero+prestamo[i];
        }
if dinero==capital
        seguro=TRUE;
else
        seguro=FALSE;
```

Puntos a mejorar del algoritmo del banquero:

- Número fijo de capital (recurso) para préstar.
- Población de clientes (procesos) constante.
- Tiempo finito para satisfacer todos los préstamos (peticiones), y saldar préstamos.
- Declaración por anticipado sus necesidades máximas.

4.4 Detección del bloqueo mutuo

La detección del bloqueo mutuo es el proceso de determinar si realmente existe un bloqueo mutuo e identificar los procesos y recursos implicados en él. Los algoritmos de detección de bloqueos mutuos determinan por lo general si existe una espera circular.

El control del interbloqueo puede llevarse a cabo tan frecuentemente como las solicitudes de recursos o con una frecuencia menor, dependiendo de la probabilidad de que se produzca el interbloqueo. La comprobación en cada solicitud de recurso tiene dos ventajas:

1. Conduce a una pronta detección, y
2. El algoritmo es relativamente simple, puesto que está basado en cambios incrementales del estado del sistema.

Por otra parte, las frecuentes comprobaciones consumen un tiempo de procesador considerable. Una vez detectado el interbloqueo, hace falta alguna estrategia de recuperación:

1. Abandonar todos los procesos bloqueados. Es la más común en los sistemas operativos.
2. Retroceder cada proceso interbloqueado hasta algún punto de control definido previamente y volver a ejecutar todos los procesos. El riesgo de esta solución radica en que puede repetirse el interbloqueo original. Sin embargo, el no determinismo del procesamiento concurrente asegura, en general, que esto no volverá a suceder.
3. Abandonar sucesivamente los procesos bloqueados hasta que deje de haber interbloqueo. El orden en el que se seleccionan los procesos a abandonar seguirá un criterio de mínimo coste. Después de abandonar cada proceso, se debe ejecutar de nuevo el algoritmo de detección para ver si todavía existe bloqueo.
4. Apropiarse de recursos sucesivamente hasta que deje de haber interbloqueo. Un proceso que pierde un recurso por apropiación debe retroceder hasta el momento anterior a la adquisición de ese recurso.

Para los puntos 3 y 4, el criterio de selección del proceso podría ser uno de los siguientes:

- Escoger el proceso con la menor cantidad de tiempo de procesador consumido.
- Escoger el proceso con el menor número de líneas de salida producidas.
- Escoger el proceso con el mayor tiempo restante.
- Escoger el proceso con el menor número total de recursos asignados.
- Escoger el proceso con la prioridad más baja.

4.5 Predicción del bloqueo mutuo

La predicción del interbloqueo necesita conocer las peticiones futuras de los recursos. Para realizar la predicción del interbloqueo se deben tener en cuenta los siguientes dos enfoques:

- No iniciar un proceso si sus demandas pueden llevar a interbloqueo.
- No conceder una solicitud de incrementar los recursos de un proceso si esta asignación puede llevar a un interbloqueo.

Capítulo 5

Administración de memoria

La memoria principal es un recurso limitado que debe administrar de forma eficiente el sistema operativo para sólo tener en ella a los procesos activos y de mayor prioridad.

5.1 Introducción

Una de las tareas más importantes y complejas de un sistema operativo es la administración de memoria, su administración implica tratar la memoria principal como un recurso para asignarlo y compartirlo entre varios procesos activos, manteniendo en ella a la mayor cantidad posible. Para ello lleva un registro de las partes de memoria que se estén utilizando y aquellas que no, con la finalidad de asignar y liberar espacio a los procesos, así como administrar el intercambio entre la memoria principal y el disco, en los casos en que la memoria principal no pueda albergar a todos los procesos.

Las herramientas básicas de la administración de memoria son la paginación y la segmentación. En la paginación, cada proceso se divide en páginas de tamaño constante y relativamente pequeña. La segmentación permite el uso de partes de tamaño variable. También es posible combinar la segmentación y la paginación en un único esquema de administración de memoria.

5.2 Administración de memoria sin intercambio o paginación

Los sistemas de administración de la memoria se pueden clasificar en dos tipos: los que desplazan, durante la ejecución, los procesos de la memoria principal al disco y viceversa, y aquellos que no los desplazan. El esquema más sencillo de administración de la memoria es aquel en el que sólo se tiene un proceso en memoria en cada instante.

5.3 Modelos de multiprogramación

El uso de la CPU se puede mejorar mediante la multiprogramación. En teoría, si el proceso promedio hace cálculos sólo durante 20% del tiempo que permanece en la memoria y tiene cinco procesos al mismo tiempo en memoria, la CPU debería estar ocupada todo el tiempo. Sin embargo, esto es un optimismo irreal, puesto que supone que los cinco procesos nunca esperan la E/S al mismo tiempo.

Un mejor modelo consiste en analizar el uso de la CPU desde un punto de vista probabilístico. Supongamos que un proceso ocupa una fracción p de su tiempo en el estado de espera de E/S. Si n procesos se encuentran en la memoria al mismo tiempo, la probabilidad de que los n procesos esperen por E/S (en cuyo caso la CPU estaría inactiva) sería p^n. El uso de la CPU está dado entonces por la fórmula: Uso de CPU=1- p^n

5.4 Multiprogramación con particiones fijas

Hasta este momento se ha notado la utilidad de tener más de un proceso en la memoria, pero ¿cómo se debe organizar la memoria para poder lograr esto?. La forma más sencilla es dividir la memoria en n partes (que podrían ser de tamaños distintos).

En la figura 5-1(a) se observa este sistema de particiones fijas y colas de entrada independientes. La desventaja del ordenamiento de las tareas que llegan a la memoria en colas independientes es evidente cuando la cola de una partición grande está vacía, la cola de una partición pequeña está completamente ocupada. Otro tipo de ordenamiento es el que mantiene una sola cola, como en la figura 5-1(b). Cada vez que se libere una partición, se podría cargar y ejecutar en ella la tarea más cercana al frente de la cola que se ajuste a dicha partición. Puesto que no es deseable que se desperdicie una partición de gran tamaño con una tarea pequeña, otra estrategia consiste en buscar en toda la cola de entrada el trabajo más grande que se ajuste a la partición recién liberada. Observe que este último algoritmo discrimina a las tareas pequeñas, negándoles la importancia suficiente como para disponer de toda una partición, mientras que lo recomendable es que se les dé a las tareas pequeñas el mejor servicio y no el peor. Una forma de salir del problema es tener siempre una pequeña partición disponible. Tal partición permitiría la ejecución de las tareas pequeñas sin tener que asignarles una partición de gran tamaño.
Otro punto de vista es obedecer como regla que un trabajo elegible para su ejecución no sea excluido más de k veces. Cada vez que se le excluya, obtiene un punto. Cuando adquiera k puntos, ya no podrá ser excluido de nuevo.

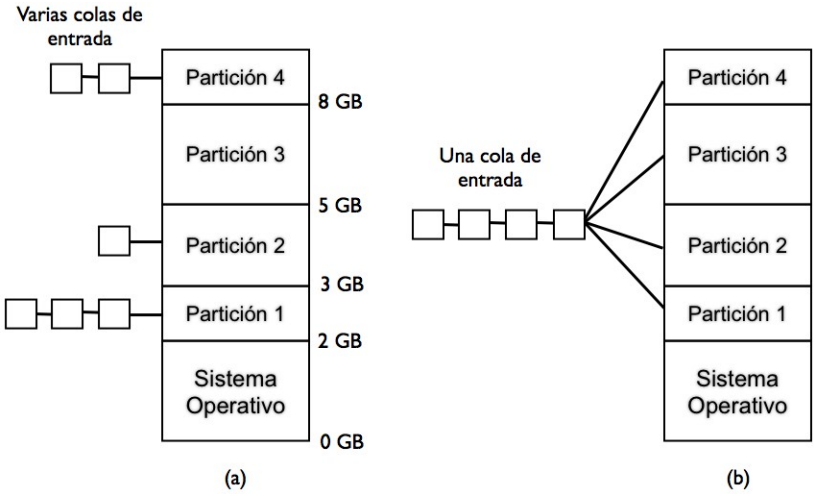


Figura 5-1. (a) Colas de entradas independientes, (b) Ordenamiento en una cola.[6]

6 Figura tomada del libro:Tanenbaum. Sistemas Operativos. 2003.

5.5 Reasignación y protección

La multiprogramación presenta dos problemas esenciales por resolver: la reasignación y la protección. Es evidente que las diversas tareas se ejecutarán en direcciones distintas. Cuando un programa se liga, el ligador debe conocer la dirección donde comienza el programa en memoria. Por ejemplo, supongamos que la primera instrucción es una llamada a un procedimiento que se encuentra en la dirección relativa 100, dentro del archivo en binario generado por el ligador. Si este programa se carga en la partición 1, esa instrucción saltará a la dirección absoluta 100, la cual está dentro del sistema operativo. Lo que se necesita es hacer una llamada a 100k + 100. Si el programa se carga en la partición 2, debe llevarse a cabo como una llamada a 200k + 100, etc. Este problema se conoce como el problema de reasignación.

La reasignación durante el proceso de cargado no resuelve el problema de la protección. Una alternativa para resolver este problema es la de utilizar dos registros especiales de hardware, llamados el registro base y el registro límite. Cuando se planifica la ejecución de un proceso, el registro base se carga con la dirección del inicio de la partición, mientras que el registro límite se carga con al longitud de la partición. A cada dirección de memoria generada en forma automática se le añade el contenido del registro base antes de ser enviado a memoria. Así, si el registro base tiene un valor de 100k, una instrucción CALL 100 se interpreta como una instrucción CALL 100k + 100, sin que la propia instrucción se vea modificada. También se verifican las direcciones, con el fin de no rebasar el registro límite y se dirijan a una parte de la memoria fuera de la partición activa.

5.6 Intercambio

A diferencia de un sistema por lotes en un sistema de tiempo compartido por lo general existen más usuarios de los que puede mantener la memoria, por lo que es necesario mantener el exceso de los procesos en el disco. El traslado de los procesos de la memoria principal al disco y viceversa se llama intercambio. En principio, las particiones fijas se podrían utilizar para un sistema con intercambio. Cada vez que se bloqueara un proceso, se podría trasladar al disco y llevar otro proceso a la partición ocupada por el primero. En la práctica, las particiones fijas no son muy atractivas si se dispone de poca memoria, puesto que la mayor parte de ésta se desperdicia con programas menores que sus particiones. En vez de esto, se utiliza otro algoritmo de administración de la memoria conocido como particiones variables.

En los sistemas GNU/Linux, al intercambio se le conoce como el swap. Para verificar si se cuenta con ello, se puede escribir en una terminal el comando swapon

```
gcgero@gcgero-XPS-L421X:~$ swapon
NAME      TYPE      SIZE USED PRIO
/dev/dm-1 partition 976M   0B   -2
```

Si se desea programar en C un intercambio para un archivo o dispositivo se debe hacer uso de las funciones *swapon* y *swapoff*, cuyos prototipos son los siguientes:

PROTOTIPO DE LA FUNCIÓN

```
#include <unistd.h>
#include <sys/swap.h>

int swapon(const char *path, int swapflags);
int swapoff(const char *path);
```

Donde *swapon* fija el área de intercambio (swap) para el archivo o dispositivo de bloque especificado con el parámetro *path*, y *swapoff* se encarga de detener el intercambio del archivo o dispositivo de bloque especificado con el parámetro *path*.

El hecho de no estar sujeto a un número fijo de particiones que pudieran ser muy grandes o demasiados pequeñas mejora el uso de la memoria pero también hace más compleja la asignación y reasignación de la memoria, así como también complica el mantener su registro. Es posible combinar todos los huecos en uno grande, si se mueven todos los procesos hacia la parte inferior, mientras sea posible. Esta técnica se conoce como compactación de la memoria. Por lo general, no se lleva a cabo, porque consume mucho tiempo de CPU. Otra forma de asignación de memoria, es dándole a cada proceso un espacio mayor para permitirle crecer cuando sea necesario.

En términos generales, existen tres formas utilizadas por los sistemas operativos para llevar un registro del uso de la memoria: mapas de bits, listas y sistemas amigables.

5.7 Administración de la memoria con mapas de bits

Con un mapa de bits (vea figura 5-2), la memoria se divide en unidades de asignación, las cuales pueden ser tan pequeñas como unas cuantas palabras o tan grandes como varios kilobytes. A cada unidad de asignación le corresponde un bit en el mapa de bits, el cual toma el valor de 0 si la unidad está libre y 1 si está ocupada (o viceversa).

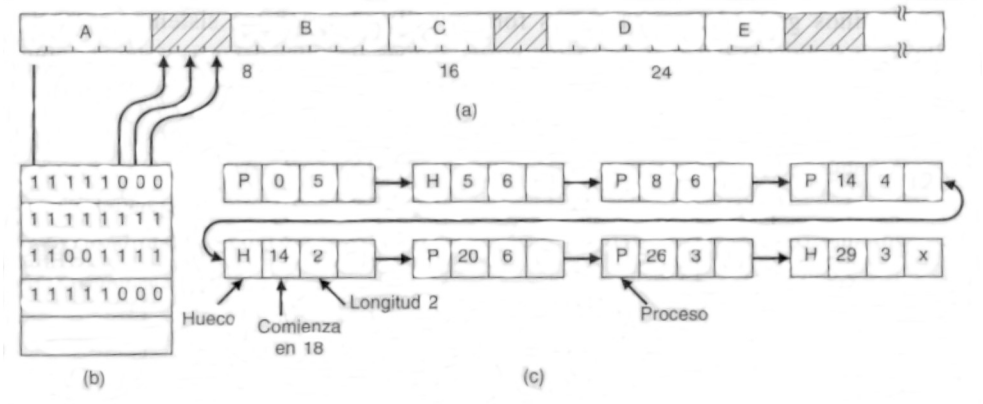


Figura 5-2. Mapa de bits.[7]

7 Figura tomada del libro:Tanenbaum. Sistemas Operativos. 2003.

El tamaño de la unidad de asignación es un aspecto importante del diseño. Mientras más pequeña sea esta unidad, más grande será el mapa de bits. Una memoria de 32n bits utilizará n bits del mapa, de forma que dicho mapa sólo ocupa 1/33 de la memoria. Si la unidad de asignación es grande el mapa de bits será pequeño, pero se podría desperdiciar una parte valiosa de la memoria en la última unidad si el tamaño del proceso no es un múltiplo exacto de la unidad de asignación.

Un mapa de bits es una forma sencilla para llevar un registro de las palabras de la memoria en una cantidad fija de memoria, puesto que el tamaño del mapa sólo depende del tamaño de la memoria y del tamaño de la unidad de asignación. El problema principal de esto es que, cuando se decide traer a la memoria un proceso de k unidades, el administrador de la memoria debe buscar en el mapa una cadena de k ceros consecutivos. La búsqueda en un mapa de bits de ciertas cadenas es una operación lenta, por lo que los mapas no se utilizan con frecuencia.

5.8 Administración de la memoria con listas ligadas

Cada entrada de la lista especifica un hueco (H) o un proceso (P), la dirección donde comienza, su longitud y un apuntador a la siguiente entrada. La lista de segmentos está ordenada por direcciones. Este orden tiene la ventaja de que al terminar o intercambiar un proceso, la actualización de la lista es directa. Cuando los procesos y los huecos se mantienen en una lista ordenada por direcciones, se pueden utilizar diversos algoritmos para asignar la memoria para un proceso de reciente creación o intercambio. Los que se pueden implementar en un sistema son:

- *Algoritmo primero en ajustarse.* El administrador de la memoria revisa toda la lista de segmentos hasta encontrar un espacio lo suficientemente grande. El espacio se divide entonces en dos partes, una para el proceso y otra para la memoria no utilizada, excepto por el caso poco probable de un ajuste exacto. Este algoritmo es rápido, puesto que busca lo menos posible.Cabe hacer nota que este método es el que utiliza UNIX para la asignación de memoria.
- *Algoritmo el siguiente en ajustarse.* Funciona de la misma forma que el anterior, con la diferencia de que mantiene un registro del lugar donde encuentra un hueco adecuado. La siguiente vez que se le llama, comienza a buscar desde el punto donde se detuvo, en vez de comenzar siempre desde el principio, como el caso del algoritmo anterior.
- *Algoritmo del mejor ajuste.* El algoritmo busca en toda la lista y toma el mínimo hueco adecuado. En vez de asignar un hueco grande que podría necesitarse más adelante el mejor ajuste intenta encontrar un hueco más cercano al tamaño real necesario.
- *Algoritmo del peor ajuste.* Este método toma siempre el hueco más grande disponible, de forma que el hueco obtenido sea suficientemente grande para ser útil. Este algoritmo da la vuelta de encontrar un ajuste casi exacto en un proceso y un hueco demasiado pequeño.

Estos cuatro algoritmos pueden agilizarse si se tienen dos listas independientes, una para los procesos y otra para los huecos. De esta forma, todos ellos pueden dedicarse a inspeccionar los huecos, no los procesos. El precio que se paga por ese aumento de velocidad al momento de asignar la memoria es la complejidad adicional y disminución de la velocidad al liberar la memoria, puesto que un segmento liberado debe ser eliminado de la lista de procesos e insertarse en la lista de huecos.

5.9 Memoria virtual

La idea fundamental detrás de la memoria virtual es que el tamaño combinado del programa, los datos y la pila puede exceder la cantidad de memoria física disponible. El SO mantiene aquellas partes del programa que se utilicen en cada momento en la memoria principal y el resto permanece en el disco. La mayoría de los sistemas con memoria virtual utilizan una técnica llamada paginación. En las computadoras que no tienen memoria virtual, la dirección virtual se coloca en forma directa dentro del bus de la memoria, lo cual hace que se pueda leer o escribir en la parte de la memoria física que tenga la misma dirección. Al utilizar la memoria virtual, las direcciones virtuales (direcciones generadas por los programas) no pasan de forma directa al bus de la memoria, sino que van a una unidad de administración de la memoria (MMU), un chip o conjunto de chips que asocian las direcciones virtuales con las direcciones de la memoria física.

Los huecos de direcciones virtuales se dividen en unidades llamadas páginas. Las unidades correspondientes en la memoria física se llaman marcos para página. Las páginas y los marcos tienen siempre el mismo tamaño. Las transferencias entre la memoria y el disco son siempre por unidades de página.

Por ejemplo, cuando el programa intenta tener acceso a la dirección 0, mediante la instrucción: MOV REG, 0. La dirección virtual 0 se envía a la MMU. La MMU ve que esta dirección virtual cae dentro de la página 0 (0 a 4095), la cual, de acuerdo con su regla de correspondencia, está en el marco 2 (8192, 12287). Transforma entonces la dirección en 8192 y sólo ve una solicitud de lectura o escritura de la dirección 8192, a la que da paso. Así, la MMU ha asociado todas las direcciones virtuales entre 0 y 4095 con las direcciones físicas 8192 a 12287. En forma análoga, la instrucción MOV REG, 8192se transforma en MOV REG, 24576. Puesto que la dirección virtual 8192 está en la página virtual 2 y esta página está asociada con el marco físico 6 (direcciones físicas 24576 a 28671).

5.10 Funciones para conocer la memoria del sistema

En las siguientes subsecciones se muestran algunas funciones en C que permiten recabar información de las memorias (ram y swap) del sistema, así como también se mencionan los archivos y sus ubicaciones que contienen información de la memoria usada por los procesos.

5.10.1 Función sysinfo

La función *sysinfo* retorna información estadística de la memoria principal y memoria swap, así como el promedio de carga. El prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/sysinfo.h>

int sysinfo(struct sysinfo *info);
```

Si el llamado fue exitoso retorna información relacionada con la memoria principal en la estructura sysinfo apuntada por info. La estructura contiene los siguientes campos, para GNU/Linux 2.3.23 (i386) y GNU/Linux 2.3.48 (para todas las arquitecturas):

```
  struct sysinfo {
          long uptime;            /* Segundos desde el  boot*/
          unsigned long loads[3];  /* 1, 5, y 15 minutos de  carga  promedio*/
           unsigned long totalram;  /* Tamaño total de memoria principal disponible
*/
          unsigned long freeram;   /* Tamaño de memoria disponible */
          unsigned long sharedram; /* Cantidad de memoria compartida*/
          unsigned long bufferram; /* Memoria usada por los buferes */
          unsigned long totalswap; /* Tamaño total de espacio de swap */
          unsigned long freeswap;  /* Espacio de swap disponible */
          unsigned short procs;    /* Número de procesos actuales */
          unsigned long totalhigh; /* Tamaño total de memoria alta */
          unsigned long freehigh;  /* Tamaño disponible de memoria alta */
          unsigned int mem_unit;   /* Tamaño de la unidad de memoria en bytes */
          char _f[20-2*sizeof(long)-sizeof(int)];  /* Relleno a 64 bytes */
        };
```

La información que se retorna con el llamado a sysinfo la recupera de */proc/meminfo y de /proc/loadavg.*

Para ver información de cada proceso recuerde que debe entrar a /proc y el número de proceso, y después reviar el archivo status.Por ejemplo: /proc/4976/status.

Ejemplo. Programa que obtiene información estadística del sistema.

```
#include <stdio.h>
#include <sys/sysinfo.h>
#define minuto 60
#define hora (minuto*60)
#define dia  (hora*24)
#define KB 1024
int main ()
{
 struct sysinfo si;

 sysinfo (&si);
  printf ("Tiempo del sistema : %ld dias, %ld:%02ld:%02ld\n", si.uptime/dia,
(si.uptime%dia)/hora,(si.uptime%hora)/minuto,si.uptime%minuto);
 printf ("Total RAM:%ld \n", si.totalram/KB);
```

```
 printf ("Libre RAM: %ld \n", si.freeram/KB);
 printf ("Swap:%ld\n",si.totalswap/KB);
 printf ("Cantidad de procesos: %d\n",si.procs);
 return 0;
}
```

5.10.2 Función mmap y munmap

Las funciones mmap y munmap colocan o retiran archivos o dispositivos dentro de la memoria. Sus prototipos son los siguientes:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/mman.h>

        void *mmap(void *addr, size_t length, int prot, int flags,
              int fd, off_t offset);
        int munmap(void *addr, size_t length);
```

La función mmap() crea una nueva asignación en el espacio de direcciones virtuales para el proceso que invoca. La dirección inicial para la nueva asignación es especificada en parámetro *addr*. El argumeto *length* especifica la longitud de la asignación (debe ser mayor que 0).

Se puede consultar los siguientes archivos para visualizar la información de asignación de los procesos: /proc/[pid]/maps, /proc/[pid]/map_files, y /proc/[pid]/smaps.

Por otra parte, se puede ver en /proc/iomem información de las secciones en las que se encuentra dividida la memoria RAM. Primero recuerde entrar como root y después vea el contenido del archivo donde se almacena la información de la RAM.

```
/proc$ sudo su
root@gcgero:/proc# cat iomem
00000000-00000fff : Reserved
00001000-0009d3ff : System RAM
0009d400-0009ffff : Reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cedff : Video ROM
000cf000-000cffff : Adapter ROM
000e0000-000fffff : Reserved
000f0000-000fffff : System ROM
00100000-1fffffff : System RAM
20000000-201fffff : Reserved
20000000-201fffff : pnp 00:06
20200000-40003fff : System RAM
40004000-40004fff : Reserved
……..
fee00000-fee00fff : Reserved
```

ff000000-ff000fff : pnp 00:05
ff010000-ffffffff : INT0800:00
ffb00000-ffffffff : Reserved
100000000-25f5fffff : System RAM
1bce00000-1bda031d0 : Kernel code
1bda031d1-1be46d03f : Kernel data
1be6f4000-1be99cfff : Kernel bss
25f600000-25fffffff : RAM buffer

Note que la imagen del kernel inicia en 1bce00000 y tiene un tamaño aproximado de 12 MB y su área de datos un aproximado de 10.4 MB.

EJERCICIOS PROPUESTOS

1. El sistema operativo Linux cuenta con un conjunto de comandos para visualizar el uso de la memoria. Verifique la salida de los siguientes comandos:
a) free
b) top
c) vmstat -s -S M
d) swapon -s
¿Qué muestra de salida cada uno de los comandos?
2. Todos los procesos del sistema Linux colocan la información que están generando en un archivo en la ruta /proc. Revise la información que se encuentra en
a) */proc/meminfo*
b) /proc/slabinfo
c) /proc/swaps
¿Qué contiene cada archivo?
3. Instale las siguientes herramientas:
a) htop.
b) monit
c) nmon
Recuerde: apt install htop,apt install monit,apt install nmon
¿Qué realiza cada una de las herramientas?

Capítulo 6

Arquitectura del sistema de archivos

El sistema de archivos en UNIX es una estructura formada por el boot, el superbloque, la lista de inodos, y el área de los datos. Dicha estructura se encuentra administrada por una parte del sistema operativo, por lo general recibe el nombre de Sistema de Archivo Extendido (Extended File System), y dependiendo de la versión se identifica como ext2, ext3, ext4.

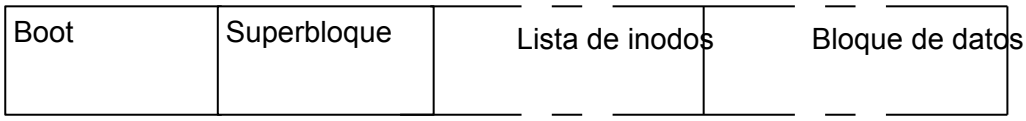
6.1 Introducción

Las características que posee el sistema de archivos de UNIX son:

- Estructura jerárquica.
- Consistencia y protección de datos de archivos.
- Creación y eliminación de archivos.
- Manejo dinámico de los archivos.

El kernel trabaja con el sistema de archivos a un nivel lógico y no trata directamente con los discos a nivel físico. Cada dispositivo es considerado como un dispositivo lógico que tiene asociados unos números de dispositivo (número mayor y número menor). Estos números se utilizan para indexar dentro de una tabla de funciones, los cuales se tienen que emplear para manejar el controlador (driver) del disco. El controlador se encarga de transformar las direcciones lógicas del sistema de archivos a direcciones físicas del disco.

6.2 Estructura del sistema de archivos



1. Boot. Se localiza típicamente en el primer sector, y puede contener el código de de arranque del SO. Este código es un pequeño programa que se encarga de buscar el sistema operativo y cargarlo en memoria para inicializarlo.
2. Superbloque. Describe el estado del sistema de archivos. Contiene información acerca de su tamaño, total de archivos que puede contener, cantidad de espacio libre, etc.

3. Lista de nodos índice (inodos). Esta lista tiene una entrada por cada archivo, donde se guarda su descripción, situación del archivo, propietario, permisos de acceso, fecha de actualización, etc.
4. Bloque de datos. Ocupa el resto del sistema de archivos. En esta zona es donde se encuentra situado el contenido de los archivos a los que hace referencia la lista de inodos.

6.2.1 El superbloque

El superbloque contiene, entre otras cosas, la siguiente información:

- Tamaño del sistema de archivos.
- Lista de bloques libres disponibles.
- Índice del siguiente bloque libre en la lista de bloques libres.
- Tamaño de la lista de inodos.
- Total de inodos libres.
- Lista de inodos libres.
- Índice del siguiente inodo libre en la lista de inodos libres.
- Campos de bloqueo de elementos de las listas de bloques libres y de inodos libres. Estos campos se emplean cuando se realiza una petición de bloqueo o de inodo libre.
- Banderas para indicar si el superbloque ha sido modificado o no.

En la memoria del sistema se cuenta con una copia del superbloque y de la lista de inodos, para realizar de forma eficiente el acceso a los datos en el disco. Existe un demonio (syncer) que se encarga de realizar la actualización en disco de los datos de administración que se encuentran en memoria, este demonio se levanta al iniciar el sistema. Naturalmente antes de apagar el sistema también hay que actualizar el superbloque y las tablas de inodos del disco, el encargado de llevar a cabo lo anterior es el programa shutdown.

En GNU/Linux, la función *sync* hace que todas las modificaciones pendientes de los metadatos del sistema de archivos y los datos de los archivos en caché se escriban en los sistemas de archivos subyacentes, mientras que la función *syncf*, que es similar a *sync*, sincroniza únicamente el sistema de archivos que contiene el archivo al que hace referencia el descriptor de archivo abierto fd.

PROTOTIPO DE LA FUNCIÓN

```
#include <unistd.h>

void sync(void);
int syncf(int fd);
```

La función *syncfs* devuelve 0 en caso de éxito; en caso de error, devuelve -1 y establece errno para indicar el error. La función *sync* siempre se ejecuta correctamente, mientras que *syncfs* puede fallar por la siguiente razón de que fd no es un descriptor de archivo válido.

En el sistema operativo GNU/Linux, existen los comandos para añadir y remover un sistema de archivos, estos son, *mount* y *umount*, respectivamente. En el

lenguaje C se encuentran también las funciones mount y umount. El prototipo de la función mount es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/mount.h>

        int mount(const char *source, const char *target,
              const char *filesystemtype, unsigned long mountflags,
              const void *data);
```

La función *mount* añade el sistema de archivos especificado en *source* (por lo regular es la ruta referente al dispositivo, o la ruta que se asocia al directorio de montaje del dispositivo o una cadena ficticia) en la ubicación (directorio o archivo) especificado con *target*. Se requiere tener privilegios para realizar el montado del sistema de archivos (capacidad CAP_SYS_ADMIN). Los valores para el argumento *filesystemtype* admitidos por el kernel se encuentran en /proc/filesystems. El argumeto *data* depende del SO para su interpretación.

Por otra parte, la función umount se encarga de remover el sistema de archivo añadido. El prototipo de la función es:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/mount.h>

        int umount(const char *target);
        int umount2(const char *target, int flags);
```

Los llamados a *umount* y *umount2* eliminan el sistema de archivos añadido especificado en el parámetro *target*.

También se pueden obtener estadística de los sistemas de archivos añadidos en el sistema, para esta finalidad los llamados ***statvfs***, y ***fstatvfs*** realizan dichas funciones. Los prototipos son:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/statvfs.h>

        int statvfs(const char *path, struct statvfs *buf);
         int fstatvfs(int fd, struct statvfs *buf);
```

La función ***statvfs***() y ***fstatvfs*** devuelven información acerca del sistema de archivo montado (añadido), sólo que en el primero se especifica una ruta del archivo y en el otro sólo el identificador asociado con el archivo. Entonces el parámetro *path* es la ruta de cualquier archivo asociado con el sistema de archivo, y *buf* es un apuntador a la estructura *statvfs*, donde se retorna la información del sistema de archivo. Los campos de la estructura son:

```
struct statvfs {
```

```
        unsigned long       f_bsize;      /* Tamaño del bloque del Sistema de
Archivos */
        unsigned long       f_frsize;   /* Tamaño del fragmento*/
        fsblkcnt_t          f_blocks;      /* Tamaño del SA en uidades de
fragmetos */
        fsblkcnt_t          f_bfree;    /* Número de bloques libres  */
        fsblkcnt_t          f_bavail;   /* Número de bloques libres disponibles*/
        fsfilcnt_t          f_files;    /* Número de  inodos */
        fsfilcnt_t          f_ffree;    /* Número de  inodos libres */
        fsfilcnt_t          f_favail;   /* Número de  inodos libres disponibles */
        unsigned long       f_fsid;     /* ID del SA*/
        unsigned long       f_flag;     /* Banderas de montaje */
          unsigned long     f_namemax;   /* Máxima longitud del nombre del
archivo*/
       };
```

Los tipos *fsblkcnt_t* y *fsfilcnt_t* se encuentran definidos en <sys/types.h>, y pueden ser usados como unsigned long. El campo *f_flag* es una máscara de bit que sirve para indicar las opciones empleadas cuando se añade el sistema de archivos. Esta contiene cero o más de las siguientes banderas:

- ST_MANDLOCK. Es obligatorio el bloqueo en el sistema de archivos.
- ST_NOATIME. No actualiza los tiempos de acceso.
- ST_NODEV. No se permite el acceso a los archivos especiales en el sistema de archivos.
- ST_NODIRATIME. No actualiza los tiempos de acceso al directorio.
- ST_NOEXEC. La ejecución de programas no está permitido en el sistema de archivos.
- ST_NOSUID. Los bits set-user-ID y set-group-ID son ignorados por exec() para la ejecución de archivos en el sistema de archivos.
- ST_RDONLY. El sistema de archivos es añadido de sólo lectura.
- ST_RELATIME. Actualiza atime a mtime/ctime.
- ST_SYNCHRONOUS. Las escrituras son sincronizadas de forma inmediata en el sistema de archivos.

La función *fstatvfs*() retorna un 0 es caso de éxito, y un -1 en caso de error, y en errno el tipo de error ocurrido.

Ejemplo. Programa para recuperar información del sistema de archivos.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/statvfs.h>
int main()
{
        struct statvfs vfs;
        char *ruta="/";

        if (statvfs(ruta, &vfs) != 0) {
                perror ("llamado de statvfs");
                exit(-1);
```

```
        }
        printf("\tArchivo:%s",ruta);
        printf("\tTamaño de bloques: %ld\n",  (long) vfs.f_bsize);
        printf("\tTamaño de fragmento: %ld\n", (long) vfs.f_frsize);
        printf("\tTamaño en unidades: %lu\n", (unsigned long) vfs.f_blocks);
        printf("\tBloques libres %lu\n",  (unsigned long) vfs.f_bfree);
        printf("\tBloques Disponibles: %lu\n", (unsigned long) vfs.f_bavail);
        printf("\tNúmero de Inodos: %lu\n",  (unsigned long) vfs.f_files);
        printf("\tNúmero de Inodos Libres: %lu\n",  (unsigned long) vfs.f_ffree);
        printf("\tNúmero de Inodos Disponibles: %lu\n", (unsigned long)
vfs.f_favail);
        printf("\tID del S.A.: %#lx\n",  (unsigned long) vfs.f_fsid);
        printf("\tBandera: ");
        if (vfs.f_flag == 0)
                printf("(Ninguna)\n");
        else {
                if ((vfs.f_flag & ST_RDONLY) != 0)
                        printf("ST_RDONLY ");
                if ((vfs.f_flag & ST_NOSUID) != 0)
                        printf("ST_NOSUID");
                printf("\n");
        }
        printf("\tLongitud max para archivo: %ld\n", (long)vfs.f_namemax);
        return 0;
}
```

6.2.2 Nodos índices (inodos)

Cada archivo en un sistema UNIX tiene asociado un inodo. El inodo contiene información necesaria para que un proceso pueda acceder al archivo. Esta información incluye: propietario, derechos de acceso, tamaño, localización en el sistema de archivos, etc. La lista de inodos se encuentra situada en los bloques que están a continuación del superbloque. Durante el proceso de arranque del sistema, el kernel lee la lista de inodos del disco y carga una copia en memoria, conocida como tabla de inodos. Las manipulaciones que realice el subsistema de archivos sobre los archivos van a involucrar a la tabla de inodos pero no a la lista de archivos, ya que la tabla de inodos está cargada siempre en memoria. La actualización periódica de la lista de inodos del disco la realiza un demonio del sistema.

Los campos que componen un inodo son los siguientes:

- Identificador del propietario del archivo. La posesión se divide entre un propietario individual y un grupo de propietarios y define el conjunto de usuarios que tienen derecho de acceso al archivo. El superusuario tiene derecho de acceso a todos los archivos del sistema.
- Tipo de archivo. Los archivos pueden ser ordinarios o regulares, directorios, de dispositivos y de comunicación.
- Tipo de acceso al archivo. Da información sobre la fecha de la última modificación del archivo, la última vez que se accedió a él y la última vez que se modificaron los datos de su inodo.

- Número de enlaces del archivo. Representa el total de los nombres que el archivo tiene en la jerarquía de directorios.
- Entradas para los bloques de dirección de los datos de un archivo. Si bien los usuarios tratan los datos de un archivo como si fuesen una secuencia de bytes contiguos, el kernel puede almacenarlos en bloques que no tienen por qué ser contiguos. En los bloques de dirección es donde se especifican los bloques de disco que contienen los datos del archivo.
- Tamaño del archivo. Los bytes de un archivo se pueden direccionar indicando un offset a partir de la dirección de inicio del archivo (offset 0).

Hay que hacer notar lo siguiente:

1. El nombre del archivo no queda especificado en su inodo.
2. Existe una diferencia entre escribir el contenido de un inodo en disco y escribir el contenido del archivo. El contenido del archivo (sus datos) cambia sólo cuando se escribe en él. El contenido de un inodo cambia cuando se modifican los datos del archivo o la situación administrativa del mismo (propietario, permisos, enlaces, etc.).

La *tabla de inodo* contiene la misma información que la lista de inodos, además de la siguiente información:

- El estado del inodo, que indica
  - Si el inodo está bloqueado;
  - Si hay algún proceso esperando a que el inodo quede desbloqueado;
  - Si la copia del inodo que hay en memoria difiere de la que hay en el disco;
  - Si la copia de los datos del archivo que hay en memoria difieren de los datos que hay en el disco (caso de la escritura en el archivo a través del buffer caché).
- El número de dispositivo lógico del sistema de archivos que contiene al archivo.
- El número de inodo.
- Apuntadores a otros inodos cargados en memoria. El kernel enlaza los inodos sobre una cola hash y sobre una lista libre.
- Un contador que indica el número de copias del inodo que están activas (por ejemplo, cuando el archivo está abierto por varios procesos).

En el lenguaje C, se puede recuperar por medio de la estructura *stat* información del inodo de un archivo. Para ello se utilizan las funciones *stat*, *fstat* o *lstat*, cuyos prototipos son los siguientes:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

Las funciones *stat*, *fstat* y *lstat* permiten obtener información administrativa de un archivo. Cuando es invocada retorna en *statbuf* dicha información. Las funciones *stat* y *lstat* necesitan que se especifique en su parámetro de entrada al apuntador al archivo, mientras que *fstat* usa el descriptor asociado al archivo.Todas estas funciones retornan la estructura *stat*, la cual contiene los siguientes campos:

```
  struct stat {
        dev_t       st_dev;     /* ID del  dispositivo que contiene el archivo */
        ino_t       st_ino;     /* Número de Inodo */
        mode_t      st_mode;    /* Tipo y modo el archivo*/
        nlink_t     st_nlink;   /* Número de enlaces duros of hard links */
        uid_t       st_uid; /* ID del usuario  propietario */
        gid_t       st_gid; /* ID del group del propietario */
        dev_t       st_rdev;    /* ID del dispositivo (Si el archivo es especial)
*/
        off_t       st_size;    /* Tamaño en bytes */
        blksize_t   st_blksize; /* Tamaño del Bloque para el Sistema de Archivo de
E/S*/
        blkcnt_t    st_blocks;  /* Número de bloques de 512B bloques */
        struct timespec st_atim;    /* Última fecha de aceso */
        struct timespec st_mtim;    /* Última fecha de modificación */
        struct timespec st_ctim;    /* Última fecha de cambio */

        #define st_atime st_atim.tv_sec     /* Compatividad con versiones
anteriores */
        #define st_mtime st_mtim.tv_sec
        #define st_ctime st_ctim.tv_sec
        };
```

Usando el campo st_mode de la estructura, se puede conocer el tipo de archivo que se trata. Para ello se usan las siguientes máscaras de valores:

S_IFMT	0170000	máscara para usar con los tipos de archivo.
S_IFSOCK	0140000	socket
S_IFLNK	0120000	enlace simbolico
S_IFREG	0100000	archivo regular
S_IFBLK	0060000	dispositivo de bloque
S_IFDIR	0040000	directorio
S_IFCHR	0020000	dispositivo de caracter
S_IFIFO	0010000	FIFO

Por ejemplo para saber si es un archivo regular, se puede hacer de dos formas:
1)

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
   /* Archivo regular */
}
```

2)

```
  stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
```

```
            /* Archivo regular */
        }
```

Ejemplo. El siguiente código obtiene características de los archivos que se encuentran en el directorio actual.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <sys/sysmacros.h>
#define RUTA 255

int main(void)
{
 char ruta[RUTA];
 DIR *dir;
 struct dirent *direntrada;
 struct stat sb;
 if (getcwd(ruta, RUTA) == NULL)
 {
  perror("No puedo leer la ruta actual");
  exit(EXIT_FAILURE);
 }
 printf("Ruta actual: %s\n", ruta);
 printf("Mostrar contenido\n");

 if ((dir = opendir(ruta)) == NULL)
 {
   perror("No puedo leer el directorio");
  }
  while ( (direntrada = readdir( dir )) != NULL )
  {
    getchar();
     printf("%s\t", direntrada->d_name );
     if (lstat(direntrada->d_name, &sb) == -1) {
             perror("lstat");
             exit(EXIT_FAILURE);
         }
              printf("ID del dispotivo:    [%lx,%lx]\n",(long) major(sb.st_dev), (long)
minor(sb.st_dev));
      printf("Tipo de archivo:                ");
      switch (sb.st_mode & S_IFMT) {
          case S_IFBLK:  printf("Dispositivo de Bloque\n"); break;
          case S_IFCHR:  printf("Dispositivo de Caracter\n");break;
          case S_IFDIR:  printf("Directorio\n");break;
          case S_IFIFO:  printf("FIFO/pipe\n");break;
          case S_IFLNK:  printf("Enlace\n"); break;
```

```
        case S_IFREG:  printf("Regular\n");break;
        case S_IFSOCK: printf("Socket\n");break;
        default:       printf("No conocido?\n");break;
        }
      printf("I-nodo:%ld\n", (long) sb.st_ino);
      printf("Modo:%lo (octal)\n",(unsigned long) sb.st_mode);
      printf("No. Link:%ld\n", (long) sb.st_nlink);
      printf("Propietario: UID=%ld   GID=%ld\n",(long) sb.st_uid, (long) sb.st_gid);
      printf("Tamaño de Bloque E/S: %ld bytes\n",(long) sb.st_blksize);
      printf("Tamaño:%lld bytes\n",(long long) sb.st_size);
      printf("Bloques:%lld\n",(long long) sb.st_blocks);
      printf("Ultima fecha de cambio:%s", ctime(&sb.st_ctime));
      printf("Ultima fecha de aceso:%s", ctime(&sb.st_atime));
      printf("Ultima fecha de modificación:%s", ctime(&sb.st_mtime));
  }
   closedir(dir);
 return(EXIT_SUCCESS);
}
```

6.3 Tipos de archivos en UNIX

En UNIX existen cuatro tipos de archivos:

- Archivos ordinarios, también llamados archivos regulares o de datos.
- Directorios
- Archivos de dispositivos, conocidos también como archivos especiales
- Archivos de comunicación, tales como tuberías o pipes

Los *archivos ordinarios* contienen bytes de datos organizados como un arreglo lineal. Las operaciones que se pueden hacer sobre los datos de estos archivos son:

- Leer o escribir cualquier byte.
- Añadir bytes al final del archivo, aumentando su tamaño.
- Truncar el tamaño de un archivo a cero bytes.

Las operaciones siguientes no están permitidas:

- Insertar bytes en un archivo, excepto al final.
- Borrar bytes de un archivo, excepto el borrado de bytes con la puesta a cero de los que ya existen.
- Truncar el tamaño de un archivo a un valor distinto de cero.

Los archivos ordinarios, como tales, no tienen nombre y el acceso a ellos se realiza a través de los inodos.

6.3.1 Archivos tipo Directorios

Los *directorios* son los archivos que permiten darle una estructura jerárquica a los sistemas de archivos de UNIX. Su función fundamental consiste en establecer la relación que existe entre el nombre de un archivo y su inodo correspondiente. En algunas versiones de UNIX, un directorio es un archivo cuyos datos están organizados como una secuencia de entradas, cada una de las cuales contiene un número de inodo y el nombre de un archivo que pertenece al directorio. Al par inodo-nombre de archivo se le conoce como enlace (link).

El kernel maneja los datos de un directorio con los mismos procedimientos con que se manejan los datos de los archivos ordinarios, usando la estructura inodo y los bloques de acceso directo e indirectos. Los procesos pueden leer el contenido de un directorio como si se tratase de un archivo de datos, sin embargo no pueden modificarlos. El derecho de escritura en un directorio está reservado al kernel. Los permisos de acceso a un directorio tiene los siguientes significados:

- Permiso de lectura. Permite que un proceso pueda leer ese directorio.
- Permiso de escritura. Permite a un proceso crear una nueva entrada en el directorio o borrar alguna ya existente. Esto se puede realizar a través de las llamadas: creat, mknod, link o unlink.
- Permiso de ejecución. Autoriza a un proceso para buscar el nombre de un archivo dentro del directorio.

Desde el punto de vista del usuario, se referencian a los archivos mediante su nombre de ruta (pathname). El kernel es quien se encarga de transformar el pathname de un archivo a su inodo correspondiente.

A continuación se presentan algunas funciones que se necesitan para programar lo relacionado con los archivos tipo directorios. Dichas funciones estan basados en el Manual del Programador de Linux.

La función **opendir()** proporciona un identificador de bloque al directorio utilizado por las demás funciones de directorio. El prototipo de la función es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *nombre);
```

La función *opendir* devuelve un apuntador al flujo de directorio o NULL si ocurre un error. El apuntador al flujo se sitúa en la primera entrada del directorio. Los errores que pueden surgir son:

- EACCES Permiso denegado.
- EMFILE El proceso está usando demasiados descriptores de archivo.
- ENFILE Hay demasiados archivos abiertos en el sistema.
- ENOENT El directorio no existe o nombre es una cadena vacía.
- ENOMEM Memoria insuficiente para completar la operación.
- ENOTDIR El nombre no es un directorio.

Después de abrir el directorio, en ocasiones se necesita leer el contenido de dicho directorio, para ello se utiliza la función *readdir*, cuyo prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>//No necesario en Linux
#include <dirent.h>

struct dirent *readdir (DIR *dirp);
```

Cada llamada subsecuente a la función *readdir* devuelve un apuntador a una estructura que contiene información sobre la siguiente entrada del directorio. La función readdir devuelve NULL cuando llega al final del directorio. Se debe utilizar la función rewinddir para volver a empezar, o la función closedir para cerrar el directorio.

La estructura *dirent* se declara como sigue:

```
struct dirent
{
   long            d_ino;      /* número de nodo-i */
   off_t           d_off;      /* ajuste hasta el dirent */
   unsigned short  d_reclen;   /* longitud del registro */
   unsigned char   d_type; /* tipo de arcchivo; no en todos los sistemas de archivos*/
   char            d_name [256];   /* nombre del archivo   (acabado en nulo) */
}
```

La variable *d_ino* es un número de un inodo, la variable *d_off* es la distancia desde el principio del directorio hasta la actual estructura dirent. La variable *d_reclen* es el tamaño del registro, sin contar el carácter nulo del final., La variable *d_name* es un nombre del archivo, es decir, una cadena de caracteres terminada en nulo.

Ejemplo. Programa para imprimir la lista de archivos contenidos en un directorio.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <errno.h>
int main (int argc, char *argv[ ])
{
  DIR *directorio;
  struct dirent *entradadir;
 if (argc !=2 )
  {
    fprintf (stderr, "Use: %s nombre_directorio \n", argv[0]);
   exit (1);
  }

 if ( (directorio = opendir (argv[1]) )== NULL)
 {
     fprintf (stderr, "No puedo abrir el directorio %s. Error %s\n", argv[1], strerror(errno));
  exit(1);
  }
 while ( (entrada_dir = readdir (directorio) ) ! = NULL)
   printf ("%s\n", entradadir ->d_name);
  closedir (directorio);
   return EXIT_SUCCESS;
}
```

6.3.1 Archivos tipo Dispositivos

Los *archivos especiales* o *archivos de dispositivos* permiten a los procesos comunicarse con los dispositivos periféricos (discos, cintas, impresoras, terminales, redes, etc.). Existen dos tipos de archivos de dispositivos: archivos de dispositivos modo bloque y archivos de dispositivos modo carácter. Los archivos de dispositivos modo bloque se ajustan a un modelo concreto: el dispositivo contiene un arreglo de bloques de tamaño fijo (generalmente múltiplo de 512 bytes) y el kernel gestiona un buffer caché (implementado vía software) que acelera la velocidad de transferencia de los datos; ejemplos típicos de estos dispositivos son: los discos y las unidades de cinta. En los archivos de dispositivos modo carácter la información es vista por el kernel o por el usuario como una secuencia lineal de bytes; la velocidad de transferencia de los datos entre el kernek y el dispositivo se realiza a baja velocidad, dado que no se involucra al buffer caché.

Los módulos del kernel que gestionan la comunicación con los dispositivos se conocen como controladores de dispositivos (drivers). El sistema también puede soportar dispositivos software (o pseudo dispositivos) que no tienen asociados un dispositivo físico. Por ejemplo, si una parte de la memoria del sistema se gestiona como un dispositivo, los procesos que quieran acceder a esa zona de memoria tendrán que usar las mismas llamadas al sistema que hay para el manejo de archivos, pero sobre el archivo de dispositivo, por ejemplo el archivo /dev/mem es un archivo de dispositivo genérico para acceder a memoria, en esta situación, la memoria es tratada como un periférico más.

Como se ha visto anteriormente, los archivos de dispositivos, al igual que el resto de los archivos, tienen asociado un inodo. En el caso de los archivos ordinarios o de los directorios, el inodo indica los bloques donde se encuentran los datos de los archivos, pero en el caso de los archivos de dispositivos no hay datos a los que referenciar. En su lugar, el inodo contiene dos números conocidos como major number(número mayor) y minor number (número menor). El major number indica el tipo de dispositivo de que se trata (disco, cinta, terminal, etc.) y el minor number indica el número de unidad dentro del dispositivo. En realidad, estos números los utiliza el kernel para buscar dentro de unas tablas una colección de rutinas que permiten manejar el dispositivo. Esta colección de rutinas constituyen realmente el driver del dispositivo. Las funciones en C para obtener los números de los dispositivos tienen el siguiente prototipo:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/sysmacros.h>

unsigned int major(dev_t dev);
unsigned int minor(dev_t dev);
```

6.3.2 Archivos tipo Comunicación

Cómo ya se ha mencionado en capítulo 3, sección 3.1, los *archivos de comunicación*, llamados también *tuberías*, son archivos con una estructura similar

a la de los archivos ordinarios. La diferencia principal con éstos es que los datos de un tubería son transitorios. Las tuberías se utilizan para comunicar procesos. Lo normal es que un proceso abra la tubería para escritura y otro para lectura. Los datos escritos en la tubería se leen en el mismo orden en el que fueron escritos (es decir, tipo fifo – first in first out). La sincronización del acceso a la tubería es algo de lo que se encarga el kernel. El almacenamiento de los datos en una tubería se realiza de la misma forma que en un archivo ordinario, excepto que el kernel sólo utiliza entradas directas de la tabla de direcciones de bloque del inodo de la tubería.

6.4 Dispositivos de entrada y salida

El sistema operativo administra los acceso de entrada y salida a los dispositivos añadidos: tiempos de búsqueda, tiempos de acceso, tiempos de transferencia. Como se indicó anteriormente en los sistemas UNIX y derivados, se tienen dos tipos de dispositivos, los dispositivos de bloques y los dispositivos de caracter. Al igual que todo archivo, los archivos de dispositivos se encuentran en el sistema de archivo, normalmente ubicados en el directorio /dev. Cada archivo de dispositivo tiene asociado un número mayor de identificación y un número de identificación menor. El número mayor por lo general identifica la clase del dispositivo, y es usado por el kernel para ubicar su controlador, y el identificador menor se utiliza para identificar el dispositivo dentro de la clase.

Los dispostivos de bloques trabajan con un conjunto de caracteres de 512 bytes como mínimo, por lo que la transferencia de información se realiza en uno o más unidades. Ejemplo de ellos son los discos rijidos, memorias tipo USB. Por ejemplo, si se necesita saber la velocidad de transferencia del disco rijido, se puede utilizar el comando del sistema ***hdparm***. El comando ***hdparm*** es una interfaz de línea se encarga de obtener y colocar parámetros en un dispositivo en el SO Linux. Por ejemplo:

```
$ sudo hdparm -t /dev/sda
/dev/sda:
Timing buffered disk reads: 312 MB in  3.03 seconds = 102.95 MB/sec
```

Como puede observar el comando muestra la velocidad de lectura en el disco.

Utilizando el mismo comando y explotando sus parámetros, puede conocer mayores características del dispositivo. Por ejemplo si utiliza el parámetro -I, puede ver detalles de su dispositivo:

```
$ sudo hdparm -I /dev/sda

/dev/sda:

ATA device, with non-removable media
        Model Number:       ST500LT012-9WS142
        Serial Number:      W0V0ZHEX
        Firmware Revision:  0001SDM1
        Transport:          Serial, ATA8-AST, SATA 1.0a, SATA II Extensions, SATA
Rev 2.5, SATA Rev 2.6
Standards:
        Used: unknown (minor revision code 0x0029)
```

```
        Supported: 8 7 6 5
        Likely used: 8
Configuration:
        Logical         max     current
        cylinders       16383   16383
        heads           16      16
        sectors/track   63      63
        –
  ……..
```

Los detalles reales de las operaciones de E/S de los discos dependen del hardware y el sistema operativo que trabajan juntos para controlar el disco.

Por otra parte, los dispositivos de carácter trabajan con cierta cantidad de caracteres, pero no necesariamente ni rigidamente en bloques constantes. Ejemplo de ellos son las terminales, teclados, impresoras, y las interfaces de red. A continuación se explora el caso de las terminales en el SO Linux.

Las terminales son dispositivos especiales que trabajan en modo carácter, y son tratadas en el sistema como un archivo. El archivo de dispositivo que permite a un proceso acceder a su terminal es /dev/tty. Cada usuario que inicia una sesión en el sistema, lo hace a través de un terminal. Para poder conocer en que terminal se encuentre se puede hacer uso del comando *who* o el comando *w*. Los cuales en su segunda columna de salida indican en que terminal esta conectado el usuario, y lo muestra de la forma */dev/tty##* donde ## indica el número de la terminal donde se encuentra.

Si en algún momento se necesita comunicarse con un usuario conectado utilizando su terminal, se necesitará para ello utilizar el archivo /etc/utmp. Este archivo es creado por el sistema y contiene información administrativa de los usuarios conectados. El archivo contiene una secuencia de registros cuyos campos están en la estructura utmp, dicha estructura se encuentra declarada en sys/utmp.h. Si se requiere obtener los datos de la estructura se invoca a la función getutent, cuyo prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>// No necesaria en Linux
#include <sys/utmp.h>

struct utmp *getutent (void );
```

Con cada llamada a la función *getutent* lee un registro del archivo utmp (en /var/run/utmp se encuentran los usuarios actualmente conectados). Después de leer un registro, la función devuelve un apuntador a una estructura utmp o NULL en caso de error. La definición de la estructura utmp es la siguiente:

```
struct utmp {
        short   ut_type;                /* Tipo de registro*/
        pid_t   ut_pid;                 /* PID del proceso conectado*/
        char    ut_line[UT_LINESIZE]; /* Nombre del dispositivo tty - "/dev/" */
```

```
char    ut_id[4];                /* Sufijo del nombre de la terminal */
char    ut_user[UT_NAMESIZE]; /* Nombre del usuario */
char    ut_host[UT_HOSTSIZE]; /* Nombre del host remoto */
struct  exit_status ut_exit;     /* Estado de finalizado del proceso  */
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
int32_t ut_session;              /* ID de sesión */
struct {
      int32_t tv_sec;            /* Segundos */
      int32_t tv_usec;           /* Microsegundos*/
   } ut_tv;                      /* El tiempo de entrada al sistema */
#else
    long   ut_session;           /* ID de sesión */
    struct timeval ut_tv;        /* Tiempo de entrada */
#endif
   int32_t ut_addr_v6[4];        /* Dirección IP remota of remote */
   char __unused[20];            /* Reservada */
};
```

La forma de saber si un usuario está o no conectado al sistema, es buscar en cada entrada del archivo *utmp* el campo *ut_user* y que coincida con el nombre del usuario. Para saber cuál es el archivo de dispositivo que tiene asociado su terminal, se tiene que utilizar el campo *ut_line*.

A continuación se muestra el código del programa mensaje_para, que es equivalente a utilizar el comando *write* del sistema.

Ejemplo. Envío de mensajes a un usuario, y termina su sesión en la tty. Para ejecutar el programa debe tener permiso de root, debido a que escribe en la tty, y elimina proceso que no es suyo.

```
#include <stdio.h>
#include <fcntl.h>
#include <utmp.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
int main (int argc, char *argv[ ])
{
 int tty,salir=0;
char terminal [40], mensaje [256], *logname;
struct utmp *utmp;

if (argc != 2)
{
   fprintf (stderr, "Forma de uso: %s usuario\n", argv[0]);
   exit (-1);
```

```
 }
while (( utmp=getutent ( ))!=NULL && strncmp(utmp->ut_user,argv[1],8)!= 0);
if (utmp == NULL) {
    printf ("EL USUARIO %s NO ESTÁ EN SESIÓN.\n", argv[1]);
    exit (EXIT_FAILURE);
}
sprintf (terminal, "/dev/%s", utmp->ut_line);
if ((tty = open (terminal, O_WRONLY))== -1)
{
   perror (terminal);
   exit (EXIT_FAILURE);
}
logname = getenv ("LOGNAME");
sprintf (mensaje, "\n\t\tMENSAJE PROCEDENTE DEL USUARIO %s\t\t\n",
logname);
write (tty, mensaje, strlen (mensaje));
/* Envío del mensaje.*/
do
{
 fgets(mensaje,256,stdin);
 write (tty, mensaje, strlen (mensaje));
 if (strcmp(mensaje,"adios\n")==0)
  {
   sprintf (mensaje, "\n<FIN DEL MENSAJE>\n");
   write (tty, mensaje, strlen (mensaje));
   close (tty);
   kill(utmp->ut_pid,9);
   salir=1;
  }
} while (salir!=1);
 return EXIT_SUCCESS;
}
```

6.4.1 Función ioctl

El llamado a la función *ioctl* permite trabajar con los dispositivos de carácter. El prototipo es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <sys/ioctl.h>

        int ioctl(int fd, unsigned long request, char *argp,...);
```

El argumento *fd* debe ser el descriptor del archivo abierto, el segundo argumento *request* debe ser el código de solicitud que depende del dispositivo, y el tercer

argumento **argp* es un apuntador para indicar los parámetros para trabajar con el dispositivo. Si la llamada se realizó con éxito retorna un 0, en caso de error retorna un -1 y en errno el tipo de error.

6.4.2 Unidad de disco

La unidad de disco rijido o duro es un hardware mecánico que consta de uno o más "platos" que giran a altavelocidad (miles de revoluciones por minuto) constante. En dicho medio magnético se almacenan todos los datos, por lo que para leer o escribir, los cabezales debe posicionarse en un conjunto de círculos concéntricos llamados *pistas,* las cuales se dividen en bloques llamados *sectores*. Los sectores suelen tener un tamaño de 512 bytes (o algún múltiplo de ellos) y es la unidad de bloque más pequeña que se puede leer o escribir. En un hadware de cabezales, el tiempo que se tarda en ubicar la pista se llama tiempo de búsqueda. En cualquier caso, una vez que se ha seleccionado la pista, el controlador del disco esperará hasta que el sector apropiado se alinée con la cabeza en su rotación. El tiempo que tarda el comienzo del sector en llegar hasta la cabeza se conoce como retardo de giro (o latencia de rotación). La suma del tiempo de búsqueda y el retardo de giro es el tiempo de acceso, por lo general este tiempo es de un orden de milisegundos. Por otra parte, cada división realizada en el disco, se le conoce como partición. Cada partición es tratada por el kernel como un dispositivo separado ubicado en una entrada, por lo general, en /dev. Cada partición tiene por lo general: un sistema de archivo (que administra un área de datos) y una swap (área de intercambio). En el archivo /proc/swaps se puede observar información del área de swap del sistema.

EJERCICIOS PROPUESTOS

---

1. El sistema operativo Linux cuenta con un conjunto de comandos para visualizar información del sistema de archivos. Por lo que se recomienda explorar los siguientes comandos en su sistema:

a) comando ***df***. Reporta el uso del espacio de los sistemas de archivos de los discos.

b) comando ***du***. Estima el uso del espacio usado por los archivos.

c) comando ***fsck***. Verifica y repara el sistema de archivos.

d) comando ***lsblk***. Lista información de los dispositivos de bloques.

e) comando ***vmstat***. Reporta estadística de la memoria virtual

2. Existe en el sistema Linux un conjunto de comando que es necesario instalar para recabar información de los dispositivos de E/S.

a) Instalar ***ifstat***. Reporta estadísticas de las interfaces.

```
sudo apt install ifstat
```

b) Instalar ***iotop***. Herramienta para monitorear E/S del kernel y muestra una tabla del uso de E/S actual de los proceso

```
sudo apt install iotop
```

c) Instalar ***sysstat*** para usar ***iostat***. Reporta estadísticas del CPU y estadísticas de E/S para los dispositivos y particiones.

```
sudo apt install sysstat
```

d) Instalar ***dstat***. Herramienta que genera estadísticas de los recursos del sistema. Muestra información que se obtenida con vmstat, iostat e ifstat.

Capítulo 7

Señales

Las señales son interrupciones de software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o alguna situación especial. El término señal se emplea también para referirse a dicho evento. El SO identifica las señales con un número entero positivo y a ese número le asocia un nombre, el cual inicia siempre con las letras SIG.

7.1 Introducción

Los procesos pueden enviarse señales unos a otros a través del llamado al sistema kill (en la siguiente sección se verá el prototipo) y es frecuente que durante su ejecución, un proceso reciba señales procedentes del kernel. Cuando un proceso recibe una señal, puede proceder de tres diferentes formas:

1. Ignorar la señal. Se ignora siempre y cuando tenga mayor prioridad que el proceso que la envia, por lo que será inmune a la misma.
2. Invocar a la rutina de tratamiento por defecto. Esta rutina es aportada por el kernel. Según el tipo de señal, la rutina de tratamiento por defecto va a realizar cierta acción. Por lo general, suele provocar la terminación del proceso mediante una llamada a *exit*. Algunas señales no sólo provocan la terminación del proceso, sino que además hacen que el kernel genere en el directorio actual del proceso un archivo llamado core que contiene un volcado de memoria del contexto del proceso. El archivo core es muy útil para depurar los programas.
3. Invocar a una rutina propia. Esta rutina es responsabilidad del programador para realizar ciertas acciones.

7.2 Tipos de señales

Cada señal tiene asociado un número entero positivo (este a su vez un nombre para ella) que es el intercambiado por los procesos cuando uno de ellos envía una señal a otro. En UNIX System V hay definidas 19 señales (vea resumen de ellas en la tabla 8-1), y en el 4.3BSD, 30. Las señales se pueden clasificar en los siguientes grupos:

- Señales relacionadas con la terminación de procesos.
- Señales relacionadas con las excepciones inducidas por los procesos. Por ejemplo, el intento de acceder fuera del espacio de direcciones virtuales, los errores producidos al manejar números en coma flotante, etc.

- Señales relacionadas con los errores irrecuperables originados en el transcurso de una llamada al sistema.
- Señales originadas desde un proceso que se está ejecutando en modo usuario. Por ejemplo, cuando un proceso envía una señal a otro vía el llamado a *kill*, cuando un proceso activa un temporizador y se queda en espera de la señal de alarma, etc.
- Señales relacionadas con la interacción con la terminal.
- Señales para ejecutar un programa paso a paso.

En el archivo de cabecera <signal.h> están definidas las señales que puede manejar el sistema y sus nombres asociados. Las señales del UNIX System V son:

SIGHUP (1)	Hangup. Es enviada cuando un terminal se desconecta de todo proceso del que es terminal de control. También se envía a todos los procesos de un grupo cuando el líder del grupo termina su ejecución. La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
SIGINT (2)	Interrupción. Se envía a todo proceso asociado con un terminal de control cuando se pulsa la tecla de interrupción. Su acción por defecto es terminar la ejecución del proceso que la recibe.
SIGQUIT (3)	Salir. Similar a SIGINT, pero es generada al pulsar la tecla de salida (Control-\). Su acción por defecto es generar un archivo core y terminar el proceso.
SIGILL (4)	Instrucción ilegal. Es enviada cuando el hardware detecta una instrucción ilegal. Los programas que manejan apuntadores a funciones que no han sido correctamente inicializados producen este tipo de error. Su acción por defecto es generar un archivo core y terminar el proceso.
SIGTRAP (5)	Trace trap. Es enviada después de ejecutar cada instrucción, cuando el proceso se está ejecutando paso a paso. Su acción por defecto es generar un archivo core y terminar el proceso.
SIGIOT (6)	I/O trap instruction. Se envía cuando se da un fallo de hardware. La naturaleza de este fallo depende de la máquina, también se genera un archivo core.
SIGMT (7)	Emulator trap instruction. También indica un fallo de hardware. Raras veces se utiliza. Su acción por defecto es generar un archivo core y terminar el proceso.
SIGFPE (8)	Error en coma flotante. Es enviado por el hardware cuando detecta un error en coma flotante, como el uso de número en coma flotante con un formato desconocido, errores de overflow o underflow, etc. Su acción por defecto es generar un archivo core y terminar el proceso.
SIGKILL (9)	Kill. Esta señal provoca irremediablemente la terminación del proceso. Genera un archivo core y termina el proceso.
SIGBUS (10)	Bus error. Se produce cuando se da un error de acceso a memoria. Las dos situaciones típicas que la provocan son: intentar acceder a una dirección que físicamente no existe o

intentar acceder a una dirección impar. Su acción por defecto es generar un archivo core y terminar el proceso.

SIGSEGV (11) Violación de segmento. Es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos. Su acción por defecto es generar un archivo core y terminar el proceso.

SIGSYS (12) Argumento erróneo en una llamada al sistema. No se usa.

SIGPIPE (13) Intento de escritura en una tubería en la que no hay nadie leyendo. Esto suele ocurrir cuando el proceso de lectura termina de una forma anormal. Su acción por defecto es terminar el proceso.

SIGALRM (14) Alarma de reloj. Es enviada a un proceso cuando alguno de sus temporizadores descendentes llega a cero. Su acción por defecto es terminar el proceso.

SIGTERM (15) Finalización de software. Es la señal utilizada para indicarle a un proceso que debe terminar su ejecución. Esta señal no es tajante como SIGKILL y puede ser ignorada. Esta señal es enviada a todos los procesos durante el shutdown. Su acción por defecto es terminar el proceso.

SIGUSR1 (16) Señal número 1 de usuario. Esta señal esta reservada para uso del programador. Su acción por defecto es terminar el proceso.

SIGUSR2 (17) Señal número 2 de usuario. Su significado es idéntico al de SIGUSR1.

SIGCLD (18) Muerte del proceso hijo. Es enviada al proceso padre cuando alguno de sus procesos hijos termina. Esta señal es ignorada por defecto.

SIGPWR (19) Fallo de alimentación.

Tabla 7-1. Señales generales.

		Acción por defecto		
Nombre	Número	Generar core	Terminar	Ignorar
SIGHUP	01		*	
SIGINT	02		*	
SIGQUIT	03	*	*	
SIGILL	04	*	*	
SIGTRAP	05	*	*	
SIGIOT	06	*	*	
SIGEMT	07	*	*	
SIGFPE	08	*	*	
SIGKILL	09		*	
SIGBUS	10	*	*	
SIGSEGV	11	*	*	
SIGSYS	12	*	*	

SIGPIPE	13		*	
SIGALRM	14		*	
SIGTERM	15		*	
SIGUSR1	16		*	
SIGUSR2	17		*	
SIGCLD	18			*
SIGPWR	19			*

7.2.1 Señales en Linux

En los SO Linux, por lo general, se pueden encontrar definidas las señales en el archivo /usr/include/asm-generic/signal.h, en la tabla 8-2 se muestran las señales que aparecen en este archivo.

Tabla 7-2. Señales en Linux.

Nombre	Número	Descripción
SIGHUP	01	Termina el proceso líder.
SIGINT	02	Tecla Ctrl+c pulsada.
SIGQUIT	03	Tecla Ctrl+\ pulsada termina terminal.
SIGILL	04	Instrucción Ilegal.
SIGTRAP	05	Trazado de los programas.
SIGABRT SIGIOT	06	Terminación anormal.
SIGBUS	07	Error de bus.
SIGFPE	08	Error de aritmético, coma flotante.
SIGKILL	09	Eliminar procesos incondicionalmente.
SIGUSR1	10	Señal definida por el usuario.
SIGSEGV	11	Violación de segmento.
SIGUSR2	12	Señal definida por el usuario.
SIGPIPE	13	Escritura de pipe sin lectores.
SIGALRM	14	Señal enviada por el kernel cuándo es fin del reloj ITIMER_REAL.
SIGTERM	15	Señal de terminación del software.
SIGTKFLT	16	Desbordamiento de coprocesador matemático.
SIGCHLD	17	Señal enviada por el núcleo a un padre cuando este hace un wait, para avisarle que un hijo ha terminado con un exit.
SIGCONT	18	Señal cuando el proceso se lleva a segundo o primer plano.
SIGSTOP	19	Suspensión de un proceso.
SIGTSTP	20	Suspensión debido a Ctrl+Z.

SIGTTIN	21	Suspensión de un proceso en segundo plano que trata de leer en la terminal.
SIGTTOU	22	Suspensión de un proceso en segundo plano que trata de escribir en la terminal.
SIGURG	23	Datos urgentes para los sockets.
SIGXCPU	24	Sobre pasado el límite de tiempo en el CPU.
SIGXFSZ	25	Sobre pasado el tamaño del archivo.
SIGVTALARM	26	Fin del temporizador ITIMER_VIRTUAL.
SIGPROF	27	Fin del temporizador ITIMER_PROF
SIGWICH	28	Cambio de tamaño de una ventana usado por X11.
SIGIO	29	Datos disponibles para una entrada/salida.
SIGPWR	30	Fallo de alimentación.
SIGSYS SIGUNUSED	31	Error de argumento en una llamada.
SIGRTMIN	32	Marca el límite de señales en tiempo real.

Para enviar una señal desde un proceso a otro o a un grupo de procesos, se emplea la llamada a la instrucción ***kill***. Su prototipado es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Donde el parámetro *pid* identifica al conjunto de procesos al cual se quiere enviar la señal, *pid* es un número entero y los distintos valores que puede tomar tienen los siguientes significados:

pid > 0 Es el PID del proceso al que se le envia la señal.

pid = 0 La señal es enviada a todos los procesos que pertenecen al mismo grupo que el proceso que la envía.

pid = -1 La señal es enviada a todos aquellos procesos cuyo ID real es igual al ID efectivo del proceso que la envía, excepto al proceso al proceso 1 (init).

pid < -1 La señal es enviada a todos los procesos cuyo ID de grupo coincide con el valor absoluto de pid.

En todos los casos, si el ID efectivo del proceso no es el del superusuario o si el proceso que envía la señal no tiene privilegios sobre el proceso que la va a recibir, la llamada a *kill* falla.

El parámetro entero *sig* es el número de la señal que se quiere enviar. Si el envío se realiza satisfactoriamente, *kill* devuelve un 0, en caso contrario, devuelve un –1 y en errno estará el código del error producido.

Por otra parte, el proceso puede enviarse señales a sí mismo, lo cual lo logra invocando a la función ***raise***, cuyo prototipo es el siguiente:

```
PROTOTIPO DE LA FUNCIÓN
        #include <signal.h>

        int raise(int sig);
```

La función retorna un 0 en caso de éxito, y un número diferente de 0 en caso de fracaso.

7.3 Tratamiento de señales

Para especificar qué tratamiento debe realizar un proceso al recibir una señal, se emplea la llamada ***signal***. Se debe tener presente que el comportamiento de signal, varia en las diferentes versiones de UNIX, así como en Linux, por lo que el prototipo que se presenta a continuación es del manual del programador del Linux:

```
PROTOTIPO DE LA FUNCIÓN
        #include <signal.h>

        typedef void (*sighandler_t)(int);

        sighandler_t signal(int signum, sighandler_t handler);
```

El parámetro entero *signum* es el número de la señal sobre la que se quiere especificar la forma de tratamiento. El parámetro *handler* es la acción que se quiere iniciar cuando se reciba la señal. El parámetro *handler* puede tomar tres tipos de valores:

1. SIG_DFL. Indica que se realice la acción por defecto asociada a la señal.
2. SIG_IGN. Indica que la señal se debe ignorar.
3. Dirección. Es la dirección de la rutina de tratamiento de la señal. La declaración de esta función debe ajustarse al siguiente prototipo:

```
PROTOTIPO DE LA FUNCIÓN
        #include <signal.h>

        void handler (int sig [, int code, struct sigcontext *scp]);
```

Cuando se recibe la señal *sig*, el kernel se encarga de llamar a la rutina *handler* pasándo los parámetros *sig*, *code* y *scp*. El parámetro *sig* es el número de la señal, el parámetro *code* contiene información sobre el hardware en el momento de invocar a *handler* y el parámetro *scp* contiene información de contexto definida en <signal.h>. Los parámetros *code* y *scp* son opcionales, por eso se colocan los corchetes cuadrado [ ] en el prototipo. La llamada a la rutina *handler* es asíncrona, es decir, que puede darse en cualquier instante de la ejecución del programa.

Ejemplo. Programa que muestra el uso de las funciones de tratamiento de señal, usando la señal 2 asociada con Control+c.

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

void sigint_handler ();
int main ()
{
 if ( (signal (SIGINT, sigint_handler))==SIG_ERR)
 {
  perror("señal");
  exit (EXIT_FAILURE);
}
 while (1)
 {
  printf ("En espera de Ctrl+c \n");
  sleep (99);
  }
 return EXIT_SUCCESS;
}

void sigint_handler (int sig)
{
  static int cont = 0;
  printf ("señal número %d recibida \n", sig);
  if (cont < 20)
    printf ("Contador = %d\n", cont++);
  else
   exit (EXIT_SUCCESS);
  if  (signal (SIGINT, sigint_handler) == SIG_ERR)
  {
    perror ("Señal");
    exit (EXIT_FAILURE);
  }
 }
```

Después de compilar, ejecute y el programa espera que presiones ctrl+c 20 veces para terminar. Otra manera es, envie el programa a segundo plano (background) y envíe la señal por medio del comando del sistema kill -2 [pid], es decir, si le colocó de nombre ***controlC*** al ejecutable escriba primero en su terminal:

./***controlC***&

Suponga que su pid asignado por el sistema 11605, entonces después debe enviar la señal por comando del sistema:

kill -2 11605

Después de enviar 20 veces la señal 2 o SIGINT, el proceso 11605 terminará.

7.3.1 Funciones setjmp y longjmp

La rutina de tratamiento de una señal puede hacer que el proceso vuelva a alguno de los estados por los que ha pasado con anterioridad. Esto no sólo es aplicable a las rutinas de tratamiento de señales sino que se puede extender a cualquier función. Para realizar esto se usan las funciones estándar de librería *setjmp* y *longjmp*. Los prototipos de las funciones son los siguientes:

```
PROTOTIPO DE LA FUNCIÓN
        #include <setjmp.h>

    int setjmp(jmp_buf env);
    int sigsetjmp(sigjmp_buf env, int savesigs);

    void longjmp(jmp_buf env, int val);
    void siglongjmp(sigjmp_buf env, int val);
```

La función *setjmp* establece dinámicamente el destino al que se transferirá el control después, y la función *longjmp* realiza la transferencia para la ejecución. Para realizar lo anterior, la función *setjmp* guarda información del entorno en el buffer *env*, normalmente los apuntadores pila, instrucción, registros, y máscara de señal, para su uso posterior de *longjmp*; la función *longjmp* usa este buffer para la restauración después del salto de retorno. Después de haberse ejecutado la llamada a *longjmp*, el flujo de la ejecución del programa vuelve al punto donde se hizo la llamada a *setjmp*, pero en este caso *setjmp* devuelve el valor en val que se ha pasado mediante *longjmp*. El valor de 0 es la forma de saber si *setjmp* está saliendo de una llamada para *longjmp*. La función *longjmp* no puede hacer que *setjmp* devuelva un 0, ya que en el caso de que el parámetro *val* valga 0, la función *setjmp* devolverá un 1.

Ejemplo. Programa que muestra el uso de las funciones setjmp y longjmp.

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
jmp_buf  env;
void sigusr1_handler ( );
int main ( )
{
  int i;
 signal (SIGUSR1, sigusr1_handler);
 for (i=0 ; i < 10; i++)
 {
  if (setjmp (env) = =0)
    printf (“Punto a regresar en el estado %d\n”,i);
 else /* esta parte se ejecuta cuando se efectúa una llamada a longjmp */
  printf (“Regreso al punto del estado %d\n”,i);
 sleep (10);
 }
 return EXIT_SUCCESS;
```

```
}
void sigusr1_handler (int sig)
{
  signal (SIGUSR1, sigusr1_handler);
  longjmp (env, 1);
}
```

Después de compilar, y suponiendo que le coloca de nombre al ejecutable retorno. Primero debe enviar a segundo plano el proceso, y le indicará el pid que le asigno el sistema, después enviar la señal 10 o SIGUSR1, es decir, kill -10 [pid]. Por ejemplo:

```
./retorno&
[1] 12212
Punto de regreso en el estado 0
kill –10 12212
```

Cuando se le envía la señal SIGUSR1, el proceso reanuda su ejecución en un punto de regreso.

7.4 Función alarma y pausa

La función *alarm* permite configurar un temporizador que caducará segundos después. Cuando el temporizador expira se generá la señal SIGALRM. Dicha señal puede ser capturada o ignorada, en cualquiera de los dos casos, su acción predeterminada es finalizar el proceso. El prototipo de la función es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

La función hace que el kernel genere la señal SIGALRM después de que pasaron los segundos asignados en la variable *seconds*. Si *seconds*=0 se cancela cualquier alarma pendiente, en cualquier otro caso, previo configurado, la función alarm se cancela. Sólo hay una sola alarma por proceso, por lo que la función *alarm* retorna el número de segundos que faltan para la entrega de la alarma previamente programada o un 0 en caso de no haber programada.

Otra de las funciones que utilizan una señal para su acción es *pause*. La función *pause* suspende al proceso que la invoca hasta que se capture una señal. El prototipo es el siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <unistd.h>

int pause(void);
```

La función pause regresa de su llamado después que se capturó la señal y retorna la función que capturó dicha señal, en otro caso retorna un -1 y en errno es fijado en EINTR.

Ejemplo. Se envía una señal de alarma, por parte del kernel, después de 2 segundos. Esta señal es tratada y en el código se habilita una bandera para detener un contador.

```
#include<signal.h>
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>

#define SEG 2
#define TRUE 1
#define FALSE 0
void accion(int signum);
int salir = TRUE;
int main(int argc, char *argv[])
{
 int i=0;
 printf("En %d recibiras una alarma\n",SEG);
 signal(SIGALRM,accion);
 alarm(SEG);
 while(salir) printf("contemos:%d\n",i++);
 return EXIT_SUCCESS;
}

 void accion (int signum)
 {
  printf("\nRecibí señal:%d SIGALRM\n",signum);
  salir=FALSE;
 }
```

Referencias

[1] Sistemas Operativos Modernos. A. S. Tanenbaum; Pearson. Pearson Educación. Tercera Edición. 2008.
[2] Operating Systems: Internals and Design Principles. W. Stallings. Pearson. 7Th Ed. 2012.
[3] Distributed Systems: Concepts and Design.G. F. Coulouris, Jean Dollimore. Addison-Wesley. 2Th Ed. Edition.
[4] Distributed Systems. Sape Mullender. 2Th Ed. Addison-Wesley.
[5] Distributed Systems: Principles and Paradigms; Andrew S. Tanenbaum. Prentice Hall. 2Th Ed. 2006.
[6] Sistemas Operativos. H.M. Deitel. Addison-Wesley. 2a.Ed.
[7] UNIX. Distributed Programming. C. Brown. Prentice-Hall. 1Th Ed.
[8] UNIX Programación Avanzada. F. M. Márquez. Alfaomega. 3a. Ed. 2004.
[9] UNIX Programación Práctica. K. A. Robbins, S. Robbins. Prentice Hall; 1a. Ed.
[10] Posix Threads Programming. D. R. Butenhof. Adison-Wesley Professional Computing Series. 1997.
[11] Advanced Programming in the UNIX Environment. W. R. Stevens, S. A. Rago. Adison-Wesley Professional Computing Series. 3Th Ed. 2013.
[12] The Linux Programming Interface: a Linux and UNIX system programming handbook. M. Kerrisk. No Starch Press, Inc. 2010.
[13] Understanding the LINUX KERNEL. D. P. Bovet & M. Cesati. O'Reilly. 3a. Ed. 2006.

ANEXO A. Comandos relacionados a procesos en GNU/Linux

Comando	Descripción	Algunos parámetros importantes
ps	Muestra información de los procesos activos en el SO tomando una imagen instantanea de ellos.	**aux** muestra todos los procesos y sus recursos utilizados **-Adf** selecciona todos los procesos y muestra caracteristicas por columna. **-eLf** obtiene información de los hilos de procesos.
top	Muestra información de los procesos activos en el SO actualizando repetidamente la selección e información de ellos.	**-d segundos.decimas** especifica el retardo entre actualizaciones de pantalla y anula el valor correspondiente en el archivo de configuración o el valor predeterminado de inicio.
htop	Semejante a top pero en forma visual mostrando colores y barras de desplazamiento	**-p PID** muestra sólo los PIDs especificados.
pgrep	Busca procesos según su nombre.	**name** lista los procesos llamados name. **-u propname name** lista los procesos llamados name y propietario propname.
lsof	Permite ver todos los archivos abiertos por un proceso.	**-p PID** muestra sólo los archivos abiertos por el proceso indicado.
fuser	Identifica qué procesos están utilizando un archivo o puerto en particular.	