



UF1288: Desarrollo de componentes software para servicios de comunicaciones

Certificado de Profesionalidad
IFCT0609 - Programación de sistemas informáticos



IFCT0609 > MF0964_3 > UF1288

ic editorial

Federico Huércano Ruiz
José Villar Cueli

**Desarrollo de
componentes software
para servicios de
comunicaciones
IFCT0609**

Federico Huércano Ruiz

José Villar Cueli

ic editorial

Desarrollo de componentes software para servicios de comunicaciones.

IFCT0609

© Federico Huércano Ruiz

© José Villar Cueli

1^a Edición

© IC Editorial, 2024

Editado por: IC Editorial

c/ Cueva de Viera, 2, Local 3

Centro Negocios CADI

29200 Antequera (Málaga)

Teléfono: 952 70 60 04

Fax: 952 84 55 03

Correo electrónico: iceditorial@iceditorial.com

Internet: www.iceditorial.com

IC Editorial ha puesto el máximo esfuerzo en ofrecer una información completa y precisa. Sin embargo, no asume ninguna responsabilidad derivada de su uso, ni tampoco la violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Mediante esta publicación se pretende proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para **IC Editorial** ninguna forma de asistencia legal, administrativa ni de ningún otro tipo.

Reservados todos los derechos de publicación en cualquier idioma.

Según el Código Penal vigente ninguna parte de este o cualquier otro libro puede ser reproducida, grabada en alguno de los sistemas de almacenamiento existentes o transmitida por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de IC EDITORIAL; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

ISBN: 978-84-1184-305-8

Presentación del manual

El **Certificado de Profesionalidad** es el instrumento de acreditación, en el ámbito de la Administración laboral, de las cualificaciones profesionales del Catálogo Nacional de Cualificaciones Profesionales adquiridas a través de procesos formativos o del proceso de reconocimiento de la experiencia laboral y de vías no formales de formación.

El elemento mínimo acreditable es la **Unidad de Competencia**. La suma de las acreditaciones de las unidades de competencia conforma la acreditación de la competencia general.

Una **Unidad de Competencia** se define como una agrupación de tareas productivas específica que realiza el profesional. Las diferentes unidades de competencia de un certificado de profesionalidad conforman la **Competencia General**, definiendo el conjunto de conocimientos y capacidades que permiten el ejercicio de una actividad profesional determinada.

Cada **Unidad de Competencia** lleva asociado un **Módulo Formativo**, donde se describe la formación necesaria para adquirir esa **Unidad de Competencia**, pudiendo dividirse en **Unidades Formativas**.

El presente manual desarrolla la Unidad Formativa **UF1288: Desarrollo de componentes software para servicios de comunicaciones**,

perteneciente al Módulo Formativo **MF0964_3: Desarrollo de elementos software para gestión de sistemas**,

asociado a la unidad de competencia **UC0964_3: Crear elementos software para la gestión del sistema y sus recursos**,

del Certificado de Profesionalidad **Programación de sistemas informáticos**.

Índice

Portada

Título

Copyright

Presentación del manual

Índice

Capítulo 1

Programación concurrente

- 1. Introducción**
 - 2. Programación de procesos e hilos de ejecución**
 - 3. Programación de eventos asíncronos**
 - 4. Mecanismos de comunicación entre procesos**
 - 5. Sincronización**
 - 6. Acceso a dispositivos**
 - 7. Resumen**
- Ejercicios de repaso y autoevaluación**

Capítulo 2

Fundamentos de comunicaciones

- 1. Introducción**
 - 2. Modelos de programación en red**
 - 3. El nivel físico**
 - 4. El nivel de enlace**
 - 5. El nivel de transporte**
 - 6. Resumen**
- Ejercicios de repaso y autoevaluación**

Capítulo 3

Programación de servicios de comunicaciones

- 1. Introducción**
- 2. Aplicaciones y utilidades de comunicaciones. Estándares de comunicaciones**
- 3. Librerías de comunicaciones de uso común**
- 4. Programación de componentes de comunicaciones**
- 5. Técnicas de depuración de servicios de comunicaciones**
- 6. Rendimiento en las comunicaciones**

7. Resumen
Ejercicios de repaso y autoevaluación

Capítulo 4
Seguridad en las comunicaciones

- 1. Introducción**
 - 2. Principios de seguridad en las comunicaciones**
 - 3. Herramientas para la gestión de la seguridad en red. Scanners**
 - 4. Seguridad IP**
 - 5. Seguridad en el nivel de aplicación. El protocolo SSL**
 - 6. Seguridad en redes inalámbricas**
 - 7. Resumen**
- Ejercicios de repaso y autoevaluación**

Bibliografía

Capítulo 1

Programación concurrente

Contenido

1. Introducción
2. Programación de procesos e hilos de ejecución
3. Programación de eventos asíncronos
4. Mecanismos de comunicación entre procesos
5. Sincronización
6. Acceso a dispositivos
7. Resumen

1. Introducción

La programación concurrente trata de las notaciones y técnicas que se usan para expresar el paralelismo potencial entre tareas y para resolver los problemas de comunicación y sincronización entre procesos.

La programación secuencial tradicional presenta una línea simple de control de flujo, donde las instrucciones (sentencias) se ejecutan consecutivamente. En cambio, con la introducción de la concurrencia es posible que el programa se divida en procesos y se ejecuten de manera simultánea, esperando mensajes y respondiendo adecuadamente.

La principal razón para la investigación de la programación concurrente es que ofrece una manera diferente de afrontar la solución de un problema. La segunda razón es la de aprovechar el paralelismo del *hardware* existente para lograr una mayor rapidez.

2. Programación de procesos e hilos de ejecución

Un primer aspecto a tener en cuenta para entender mejor el paradigma de la concurrencia es identificar los diferentes escenarios en los que se puede ejecutar el programa a nivel de *hardware*.

La concurrencia tiene que ser funcionalmente independiente de la máquina en la que se utilice. En algunos casos, la concurrencia será más efectiva porque el *hardware* que la acompaña es el óptimo para aplicar el paradigma de ejecutar varias tareas al mismo tiempo.

Entornos *hardware* de programas concurrentes son:

- **Entorno monoprocesador con multiprogramación.** En este caso la concurrencia es virtual. Hay un solo procesador y por lo tanto la concurrencia no existe como tal, sino que las sentencias se van ejecutando de manera entrelazada.
- **Entorno multicamputador con memoria compartida.** Aquí la concurrencia sí es plena, pues existen varios procesadores que se van a encargar de ejecutar los procesos. La comunicación entre procesadores es a través de un bus compartido, memoria, etc.
- **Entorno distribuido.** En este caso está compuesto por un conjunto de computadores distribuidos geográficamente (monoprocesador o multiprocesador) y conforman los nudos de una red. No comparten memoria, y la comunicación y sincronización es a través de mensajería.



Actividades

1. Señale cuál cree que es la ventaja de la concurrencia en los sistemas monoprocesador.
 2. Investigue cuál es la diferencia entre programación concurrente y paralela.
-

En la programación concurrente el elemento principal sobre el que gira todo es el proceso y dentro del proceso, los hilos de ejecución.

Proceso

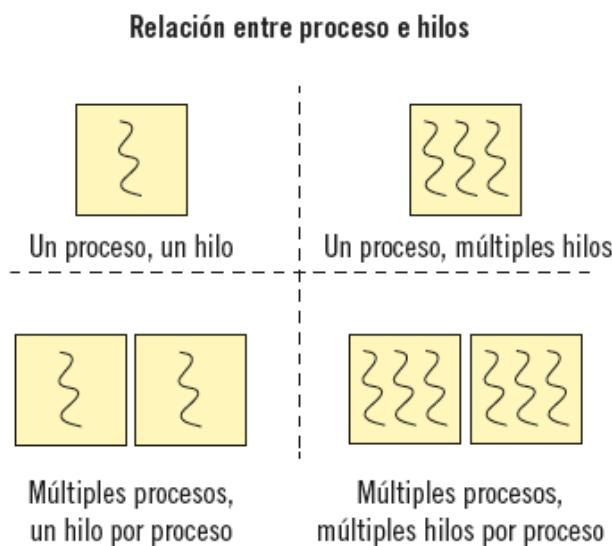
Un programa, bien sea de usuario o formando parte del sistema operativo, se puede dividir en módulos independientes para que sea posible que su ejecución se realice de manera concurrente. Con esto se consigue velocidad y un mejor aprovechamiento del procesador. El aspecto fundamental de los procesos es que se ejecuten de manera independiente.

Hilos de ejecución

A los hilos de ejecución (*threads*) se les suele llamar procesos ligeros, y forman parte de la ejecución del proceso. Pueden estar formados por un solo hilo (monohilo) o de varios hilos (multihilo).

Existe una relación muy fuerte entre hilos, pues comparten memoria, registros, procesador, etc.; y es fundamental la sincronización para que no se produzcan bloqueos ni esperas innecesarias.

En el siguiente gráfico se muestra la relación entre procesos e hilos de ejecución:

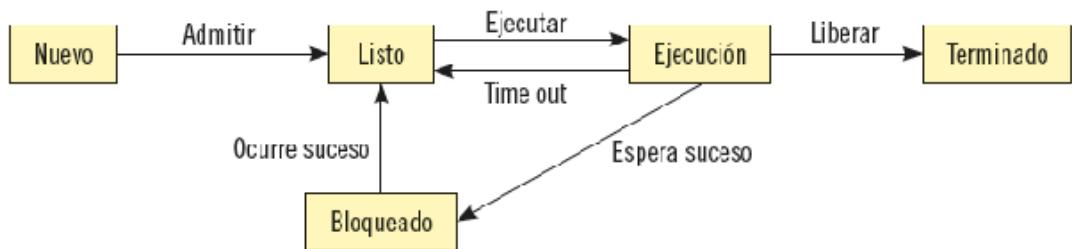


2.1. Gestión de procesos

Los procesos tienen su propio estado independiente del estado de otro proceso que se esté ejecutando en ese momento en el sistema. Van acompañados de recursos como archivos, memoria, etc.

Debido a que los procesos tienen que coexistir en el sistema, y muchos se tienen que ejecutar en un sistema monoprocesador donde la concurrencia se tiene que simular, es necesaria una planificación de los procesos, que se detalla en el siguiente diagrama de estado.

Diagrama de estados de un proceso



Los tres estados básicos son:

- **En ejecución:** el proceso está haciendo uso de la CPU, y las instrucciones que componen su tarea se están ejecutando.
- **Listo:** está en espera de que la CPU quede liberada para pasar a ejecutarse.
- **Bloqueado:** este estado de bloqueo se produce cuando el proceso está esperando a que se produzca un evento externo.

A partir de los estados enumerados anteriormente, se tienen las siguientes transiciones:

- **Ejecución-Bloqueado:** se pasa de Ejecución a Bloqueado, cuando ese proceso no puede continuar porque depende de que ocurra algún evento externo, por ejemplo, una E/S.
- **Ejecución-Listo:** esta transición se produce cuando el tiempo asignado al proceso en la CPU se acaba y aunque podría seguir ejecutándose debe pasar el testigo a otro proceso.
- **Listo-Ejecución:** cuando le llega el turno al proceso, deja de esperar en el estado de Listo y se ejecuta porque se le otorga tiempo de CPU.
- **Bloqueado-Listo:** el proceso está bloqueado porque necesitaba que se produjera un evento externo. Cuando se produce dicho evento, cambia al estado de Listo, dispuesto a ejecutarse en cuanto el planificador le de paso.

Por lo tanto, el proceso pasa por diferentes estados hasta que finalmente se ejecuta en la CPU. Pero en este momento surge una pregunta: ¿quién es el encargado de decidir qué procesos y en qué momento se van a ejecutar? Esta función recae sobre el RTSS (**Sistema de Soporte de Tiempo Real**), que es un componente que forma parte del sistema operativo. La política que se sigue para elegir el proceso a ejecutar se denomina **planificación de procesos (scheduling)**.

El RTSS debe disponer de toda la información del proceso. Esta información se almacena en una estructura de datos que recibe el nombre de **bloque de control de proceso (PCB)**. En esa estructura hay información para identificar de manera única al proceso, así como información de su estado, registros donde se identifica la instrucción

en la que se quedó ejecutándose, etc. De esta manera, al conservar dicha información, cuando el proceso pase de nuevo a ejecutarse en la CPU seguirá funcionando como si en ningún momento lo hubiera dejado.

La información que debe estar incluida en el PCB de un proceso es al menos:

- **Identificador del proceso (ID):** estructura de datos que identifica de forma única el proceso.
- **Status del proceso:** describe el estado en que se encuentra el proceso.
- **Entorno del proceso:** información que requiere el proceso para continuar su ejecución.
- **Enlaces:** permite la asociación entre procesos y su adscripción a las colas.

Existe un principio fundamental de la programación concurrente: “**El comportamiento funcional de un programa concurrente no debe depender de la política de planificación de los procesos que impone el RTSS**”.



Nota

Las aplicaciones móviles (apps) suelen hacer uso de la programación concurrente.

Este principio lo que indica es que el resultado del programa tiene que ser el mismo independientemente de la planificación que se haga de los procesos que lo componen.

Dentro de la política de planificación (*scheduling*) se pueden diferenciar dos aspectos.

Un primer aspecto es si se tiene la capacidad de echar o no un proceso de ejecución de la CPU. Desde este punto de vista caben dos posibilidades:

- **Política desalojante (*pre-emptive*):** Al proceso se le puede retirar de la CPU si hay otro proceso con mayor prioridad.
- **Política no desalojante (*no pre-emptive*):** esta política es más limitada, pues el proceso que se encuentra en ejecución no puede ser desalojado de la CPU hasta que termine, se duerme o se suspende en espera de un evento.



Nota

El RTSS es el encargado de decidir qué proceso es el que puede tomar el control de la CPU.

Un segundo aspecto de la política de planificación es el criterio a seguir para identificar el proceso que debe ejecutarse. Las políticas utilizadas son:

- **Orden FIFO (*Fist in First Out*):** el proceso que primero entró en la espera es el primero que se elige para ser ejecutado en la CPU.
- **Orden LIFO (*Last in First Out*):** el último que llegó a la espera es el primero que se elige para su ejecución.
- **Round Robin:** con esta política, el RTSS asigna a cada proceso un intervalo de tiempo (*quantum*) de la CPU, siguiendo el algoritmo FIFO para la elección del siguiente proceso. Cuando un proceso es desalojado por concluir el tiempo asignado, pasa a ocupar el último puesto de la cola.
- **Prioridad estática:** a cada proceso se le asigna una prioridad absoluta. El RTSS selecciona en cada punto de planificación aquel proceso con mayor prioridad de la lista de espera. Esto ocurre cuando se tiene muy claro el orden en el que se tienen que ejecutar los procesos por políticas superiores, que obligan o que recomiendan que sea en ese orden.
- **Prioridad dinámica:** en este caso la prioridad va cambiando respecto al tiempo de espera del proceso. La prioridad va aumentando para dar salida a los procesos que llevan más tiempo esperando.



Aplicación práctica

Se deben gestionar los procesos que se ejecutan en un servidor de manera que la optimización sea máxima en cuanto a que se ejecuten el mayor número de procesos en el menor tiempo posible. Teniendo en cuenta que los procesos a ejecutar son cortos, ¿qué criterio o combinación de criterios de los vistos anteriormente serían más convenientes de utilizar?

SOLUCIÓN

Si los procesos son cortos el criterio más acertado, sería el de *Round Robin*, asignándoles un intervalo de posesión de la CPU (*quantum*) lo suficientemente grande para que un porcentaje de procesos que rondase el 80 % pudiera terminar su ejecución sin ser interrumpido.

Otra opción podría ser crear varias colas de espera donde se identifiquen el tiempo necesario por cada proceso e incluirlo en la cola donde el quantum sea el que mejor se adapte a dicho tamaño, para llegar al porcentaje indicado anteriormente. En este caso debería haber una jerarquía de colas, que podría

seguir el siguiente criterio: "el siguiente proceso que se ejecutará se elegirá de la cola que en ese momento tenga mayor número de procesos en espera".

De esta manera, se favorece que las colas se equilibren en cuanto a número de procesos en espera.

2.2. Hilos y sincronización

Como ya se ha indicado, los hilos de ejecución, también llamados hebras, procesos ligeros, flujos, subprocesos o *thread*, son programas en ejecución que comparten memoria y otros recursos del proceso con otros hilos. Desde un punto de vista de programación, son como funciones que se pueden lanzar en paralelo con otras.



Importante

A los hilos también se les suele llamar procesos ligeros, subprocesos o también *thread*.

Todos los hilos que conforman un proceso comparten un entorno igual de ejecución (variables, direcciones, memoria, ficheros abiertos, etc.); sin embargo, cada hilo sí tiene un juego de registros de CPU, pila y variables locales, que permitirán que cuando vuelva a tomar el control del procesador por su ejecución pueda volver a proseguir por la línea en la que se quedó.

Con respecto a los hilos, es muy importante un buen diseño de su funcionalidad pues no existe protección entre ellos. Como se comparten recursos tales como la pila, un hilo podría estropear la pila del otro. Por este motivo, son necesarios mecanismos de sincronización.

La sincronización es fundamental en los hilos que conforman un proceso. Deben existir mecanismos que permitan la comunicación entre hilos para que no se produzca un interbloqueo, y que una tarea anule a la otra.

De igual forma, como se acceden a recursos compartidos, se debe asegurar la integridad de los valores a los que se accede. A modo de ejemplo, ¿qué ocurriría si dos hilos acceden al mismo tiempo a una misma variable que un tercer hilo también está modificando?

Más adelante, se detallarán las funciones de sincronización entre hilos.

El concepto de sincronización se entenderá mejor con el siguiente ejemplo, propuesto por Edsger Dijkstra en 1965:



Ejemplo

Cinco filósofos se disponen a comer un plato de fideos, y se sitúan como se muestra en la imagen. Delante de todos ellos hay un plato y tienen a su derecha y a su izquierda un tenedor. Para poder comer necesitan los dos tenedores, y solo pueden coger los que tienen a su lado, y de uno en uno.

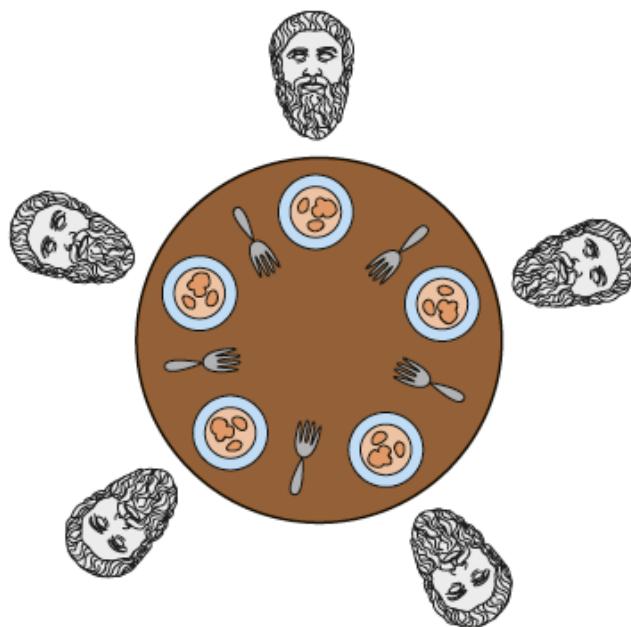
Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.

Si dos filósofos que se encuentran adyacentes intentan coger un tenedor al mismo tiempo, se produce una carrera crítica, en el que uno de ellos se quedará sin comer.

Si todos los filósofos intentan coger el tenedor de su derecha al mismo tiempo se produce una situación de interbloqueo, pues se están bloqueando mutuamente al no conseguir ninguno los recursos necesarios (dos tenedores) para poder comer.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

Sincronización de hilos



3. Programación de eventos asíncronos

Como ya se ha visto, el elemento principal dentro de la programación concurrente es el proceso y la comunicación entre ellos (*interprocess communication* o **IPC**). Los procesos siguen un protocolo (conjunto de reglas a observar) de comunicación entre ellos, donde en algunos casos se actúa como emisor y en otros como receptor.

En la comunicación entre procesos se tienen las siguientes operaciones:

- **Enviar:** operación invocada por el emisor, con el fin de enviar datos. Se identifica el proceso que lo recibe y los datos transmitidos.
- **Recibir:** operación invocada por el receptor, con el fin de aceptar los datos. Debe identificar quién le manda los datos y reservar memoria para almacenar la información del mensaje enviada.
- **Conectar:** operación que permite la conexión a través **solicitar_conexión y aceptar_conexión**, según se trate del emisor o del receptor.
- **Desconectar:** permitir que una conexión lógica establecida anteriormente pueda ser liberada.

El proceso que quiere comunicarse con otro tiene que utilizar estas operaciones en un orden determinado. Cada vez que se invoca una de estas operaciones se produce un **evento**.



Ejemplo

El navegador web utiliza el protocolo http, que tiene una serie de reglas (operaciones básicas) que se deben cumplir en un orden determinado. De esta forma el navegador web no puede enviar ninguna información hasta que no se haya completado la operación conectar, que tiene que ser previa.

La forma más sencilla para que haya sincronización de eventos, y no se ejecute un proceso hasta que no haya acabado el anterior, es por medio de peticiones bloqueantes (**síncronas**), que es la supresión de la ejecución del proceso hasta que la operación que se solicitó finalice. Sin embargo, este tipo de comunicaciones en algunas situaciones no son las más eficaces, pues si se produce un error en un proceso provocará que se detenga, dando lugar a lo que se conoce como un bloqueo.

Como alternativa a este tipo de comunicaciones están las peticiones no bloqueantes (**asíncronas**). En este caso no causa bloqueo, porque el proceso que invoca la operación sigue con su tarea. Posteriormente, se le informará al proceso si lo que solicitó se ha completado.

Se puede dar el caso que entre procesos se envíen peticiones síncronas, y se reciban asíncronas y viceversa.

3.1. Señales

Las señales permiten a los procesos comunicarse para conseguir la sincronización asíncrona necesaria para el uso óptimo del procesador y evitar la aparición de interbloqueos. Este concepto es análogo a las interrupciones que se producen en el procesador.

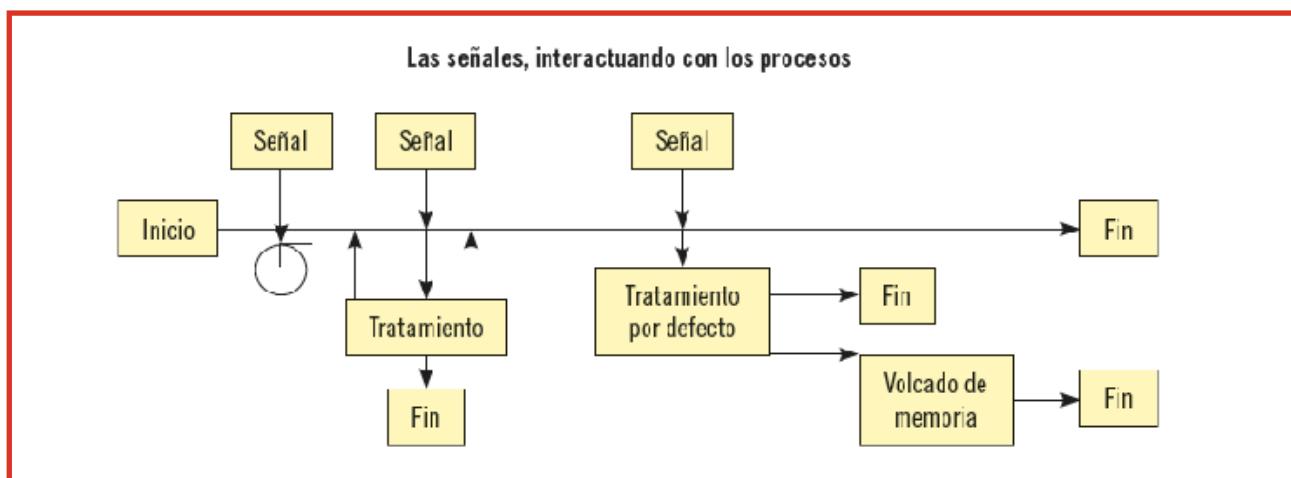
Un proceso puede estar ejecutando una tarea y en cualquier momento recibir una señal, bien de otro proceso o del sistema operativo. Cuando se recibe dicha señal, es el proceso quien decide el tratamiento que le va a dar.

El proceso cuando recibe la señal puede actuar de diferentes formas:

1. **Ignorar la señal**, con lo cual no tiene efecto. El proceso puede considerar según unos criterios que lo que está realizando es más prioritario que atender la petición externa, y no actúa.

2. **Invocar a la rutina de tratamiento**, correspondiente al número de señal. En este caso no es el programador el que codifica la rutina sino el kernel, y normalmente se finaliza el proceso que recibe la señal. El kernel es el encargado de recopilar toda la información de estado necesaria, para que cuando se retome la ejecución del proceso se haga a partir del punto en el que se quedó.
3. **Invocar a una rutina que se encarga de tratar la señal**, y que ha sido creada por el programador. Se produce una comunicación entre procesos o modifica el curso normal del programa. En estos casos, el proceso no va a terminar a menos que la rutina de tratamiento indique lo contrario.

En el siguiente esquema se pretende reflejar las tres situaciones indicadas anteriormente. En la primera señal, el proceso ignora la señal y no se hace nada. En la segunda señal, el proceso sí la contempla y recibe un tratamiento. Después de ese tratamiento se puede optar por volver a ejecutar el proceso por donde iba o finalizarlo. Por último, también puede llegar la señal que obliga a realizar un tratamiento por defecto. Por ejemplo, en este último caso, que se produzca un error de div/0, debe realizar un volcado de memoria y finalizar el proceso en ese punto.



Conjunto de señales básicas

Las señales se identifican con un código (número entero positivo), que es el mecanismo de intercambio entre procesos. Hay grupos de señales identificadas por su funcionalidad; así se tienen señales usadas para la terminación de procesos, relacionadas con la interacción con el usuario o con errores del sistema.

A modo de ejemplo, se describen algunas de las señales más usadas. Para un listado completo, se remite al lector a las guías de referencia del sistema operativo.

Señal	Descripción
SIGHUP (1) Hangup	La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
SIGINT (2) Interrupción	Es enviada cuando en medio de un proceso se pulsan las teclas de interrupción [Ctrl] + [C]. Por defecto se termina la ejecución del proceso que recibe la señal.
SIGQUIT (3) Salir	Similar a SIGINT, pero es generada al pulsar la tecla de salida [Ctrl] + []. Su acción por defecto es generar un fichero core y terminar el proceso.
SIGILL (4) Instrucción ilegal	Es enviada cuando el hardware detecta una instrucción ilegal.
SIGTRAP (5) Trace trap	Es enviada después de ejecutar cada instrucción cuando el proceso se está ejecutando paso a paso.
SIGIOT (6) I/O trap instruction	Es enviada a los procesos cuando se detecta un fallo hardware.
SIGEMT (7) Emulator trap instruction	Advierte de errores detectados por el hardware.
SIGFPE (8) Error en coma flotante	Es enviada cuando el hardware detecta un error en coma flotante, como el uso de número en coma flotante con un formato desconocido, errores de overflow o underflow, etc.
SIGKILL (9) Kill	Esta señal provoca irremediablemente la terminación del proceso. No puede ser ignorada ni tampoco se puede modificar la rutina por defecto.
SIGBUS (10) Bus error	Se produce cuando se intenta acceder de forma errónea a una zona de memoria o a una dirección inexistente. Su acción es terminar el proceso que la recibe.
SIGSEGV(11) Violación de segmento	Es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos
SIGSYS (12) Argumento erróneo en una llamada al sistema	Si uno de los argumentos de una llamada al sistema es erróneo se envía esta señal.
SIGPIPE (13) Intento de escritura en una tubería de la que no hay nadie leyendo	Esto suele ocurrir cuando el proceso de lectura termina de una forma anormal. De esta forma se evita perder datos. Su acción es terminar el proceso.

SIGALRM(14) Alarm clock	Cada proceso tiene asignados un conjunto de temporizadores. Si se ha activado alguno de ellos y este llega a cero, se envía esta señal al proceso.
SIGTERM(15) Finalización 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.
SIGUSR1(16) Señal número 1 de usuario	Esta señal está reservada para el usuario. Su interpretación dependerá del código desarrollado por el programador.
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	Es enviada cuando se detecta un fallo de alimentación.



Aplicación práctica

En el desarrollo de un programa concurrente se tiene que enviar una señal de terminación de proceso cuando se produzca un determinado evento. Sin embargo, se pretende que el proceso que se quiere finalizar pueda hacerlo de manera estable. A la vista de las señales estudiadas, ¿qué señal es la más adecuada en este caso?

SOLUCIÓN

Con la señal SIGTERM la finalización del proceso sería menos lesiva puesto que al recibirla, entre otras cosas, puede hasta ignorarla. Sin embargo, si el proceso la contempla como necesaria, en la rutina de tratamiento se podrían cerrar conexiones, ficheros, etc., para permitir recuperar el estado, una vez que tome de nuevo el control de la CPU.

Si por el contrario se utiliza la señal SIGKILL, el proceso se apagaría forzosamente, no dando pie a ninguna rutina de tratamiento, para preservar el estado del proceso. En este caso se asegura la finalización del proceso, pero a costa de perder datos.

Por lo tanto, sería mejor en la situación planteada hacer uso de la señal SIGTERM.

Tomando como base el lenguaje de programación C, se van a detallar las acciones asociadas a la comunicación entre procesos a través de señales. En esta comunicación se tiene:

- Enviar una señal.
- Recibir una señal.
- Esperar una señal.

Enviar una señal

Para el envío de una señal de un proceso a otro se emplea la llamada:

- **kill (pid,sig)**, donde **pid** identifica el proceso al que se le envía la señal, y **sig** representa la señal que se quiere enviar.

Si el envío se realiza satisfactoriamente, **kill (pid, sig)** devuelve 0; en caso contrario devuelve -1.

Recibir una señal

En el caso de que se quiera hacer un tratamiento determinado cuando llegue una señal a un proceso, diferente al que haría el kernel, se tiene que crear una rutina para tratar dicha señal. Ese código se situaría por encima del tratamiento por defecto.

Para ello se utiliza la llamada:

- **signal (sig, action)**. El parámetro **sig** identifica la señal a tratar; **action** es la acción que se desea iniciar cuando se reciba la señal.
- Este parámetro tomará uno de los siguientes tres valores:
 - **SIG_IGN**. Indica que la señal se debe ignorar. No todas las señales pueden ignorarse. Este es el caso de **SIGKILL**.
 - **SIG_DFL**. Indica que la acción a realizar cuando se reciba la señal es la acción por defecto asociada a dicha señal.
 - **dirección**. Es la dirección de inicio de la rutina de tratamiento de la señal.

La llamada a la rutina es asíncrona, lo cual quiere decir que puede darse en cualquier instante de la ejecución del programa.

La llamada a **signal (sig, action)** devuelve el valor que tenía **action**; valor que puede servir para restaurarlo en cualquier instante posterior. Si se produce algún error, **signal (sig, action)** devuelve **SIG_ERR**.



Actividades

3. Investigue qué función necesitaría utilizar para enviar una señal al mismo proceso que la envía, y piense en qué situación puede ser útil hacer esto.

Esperar una señal

Una señal es una llamada asíncrona, es decir, se puede producir en cualquier momento. Sin embargo, en algunas situaciones puede que interese parar un proceso hasta que se produzca una señal determinada.

Para ello se usa la llamada **pause()**, que bloquea el proceso, hasta que se recibe la señal que se está esperando.



Ejemplo

Se bloquea el proceso hasta que se recibe la señal SIGUSR1, presentando por pantalla un número aleatorio.

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

void sigusr1(int sig);
void sigterm();

/* Función principal */
main()
{
    signal(SIGTERM, termina);
    signal(SIGUSR1, tratamiento);
    while (1) pause ();
}

/* Función sigterm */
void termina()
{
    printf("Terminación del proceso %d a petición del usuario. \n",
getpid());
    exit (-1);
}

/* Función sigusr1 */
void tratamiento(int sig)
{
    signal(sig, SIG_IGN);
    printf("%d\n", rand());
    signal(sig, sigusr1);
}
```

Para ejecutar este programa y que muestre los números aleatorios, se debe enviar la señal 16 con la orden **kill**, y para terminar la ejecución del proceso se envía la señal número 15.

3.2. Temporizadores

Los temporizadores permiten crear períodos temporales que una vez finalizados pueden generar señales que se envíen a otros procesos o al sistema operativo, informando sobre la finalización del temporizador. De igual forma también se pueden utilizar como contadores de cuenta atrás. El temporizador está condicionado por un reloj programado.

En la programación concurrente es un elemento muy importante, pues permite sincronizar ciertos procesos que por su naturaleza tienen una duración definida o deben ejecutarse en un instante concreto.

Existen diferentes funciones para el manejo de temporizadores:

- **timer_create:** crea un temporizador asociado a un reloj.
- **timer_settime:** se arma o desarma un temporizador.
- **timer_gettime:** se lee el estado del temporizador.

La comunicación entre procesos es asíncrona y es inaceptable que un proceso se quede suspendido de forma indefinida, pues llevaría al proceso y al programa que lo contiene al bloqueo.



Definición

Interbloqueo

Bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o bien se comunican unos con otros.

El temporizador permite fijar el tiempo máximo de bloqueo, es decir, se le indica al proceso cuánto tiempo se está dispuesto a esperar a que una tarea se realice.

Los temporizadores los proporciona el propio mecanismo de comunicación y pueden ser fijados desde el programa por medio de las operaciones indicadas anteriormente.

Por ejemplo, se puede fijar un temporizador de 30 segundos para la operación de conectar. Si la petición no se completa en aproximadamente 30 segundos, el mecanismo de comunicación la abortará. En dicho instante el proceso que la solicitó se desbloqueará, pudiendo así continuar con su ejecución.

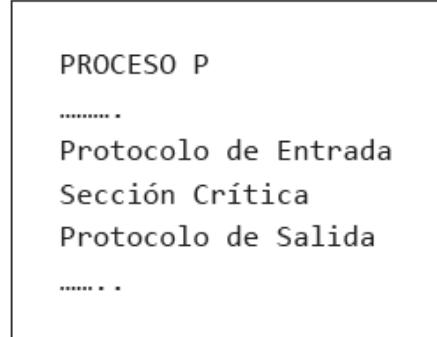
Los temporizadores también se pueden utilizar para generar señales en ciertos momentos o para ejecutar acciones periódicas. Por ejemplo, un proceso podría cada "x" segundos realizar una determinada tarea, generando una señal, que se mandaría a un proceso que la trataría.

Otro uso de los temporizadores es generar tiempos de espera concretos y absolutos. Por ejemplo, un proceso padre podría crear un proceso hijo, que tenga un temporizador de "x" segundos, y cuando finalice se lo indique a su proceso padre.

4. Mecanismos de comunicación entre procesos

En la comunicación entre procesos dos son los problemas fundamentales que aparecen: la exclusión mutua y la condición de sincronización.

La **exclusión mutua** lo que persigue es que un solo proceso pueda excluir temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema. A la parte del programa donde se usa el recurso compartido se le denomina **sección crítica**.



La **condición de sincronización** entre procesos consiste en que cuando entre procesos se usa un recurso compartido, este debe estar en un determinado estado para que el proceso pueda hacer uso de él.

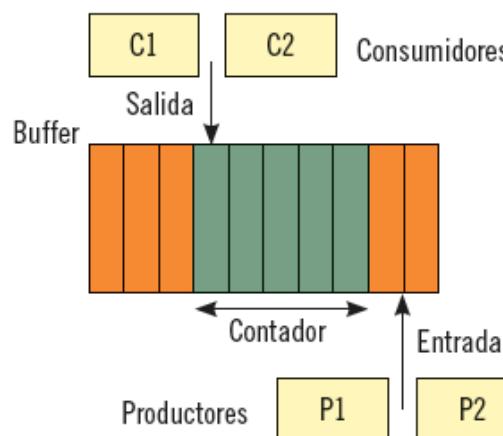
Para solucionar estos dos problemas existen diferentes mecanismos que son los que se van a tratar en este apartado.

4.1. Tuberías (PIPES)

Este mecanismo, como su propio nombre indica, simula una tubería, donde los procesos están **conectados de tal forma que la salida de uno es la entrada al otro**. Esto permite la comunicación y sincronización entre los procesos, usando para ello el uso de un buffer de datos.

La comunicación por medio de tuberías se basa en la interacción **productor/consumidor**.

Sincronización entre productor y consumidor



shorter, orden en el que el proceso mas corto es el elegido

Como se observa en el gráfico, los procesos productores (generadores de datos) se comunican con el consumidor (los que reciben los datos), a través de un buffer que sigue el protocolo **FIFO** (*First in First Out*). El primero que entró será el primero que recibirá el proceso consumidor. Cuando ya se ha leído un determinado elemento es eliminado de la tubería.

Las tuberías son un elemento importante en la programación concurrente, pues se liberan a los procesos de tener que estar esperando a que otro proceso esté preparado para recoger el dato generado. De esta manera el planificador de corto plazo va a dar el uso de la CPU a cada proceso a medida que pueda ejecutarse, minimizando los tiempos muertos.



Importante

El modelo indicado productor-consumidor no es igual al modelo cliente-servidor. En el primer caso, la comunicación es unidireccional.

Para un mejor funcionamiento de las tuberías la mayoría de los sistemas las combinan con buffers que permiten liberar aún más la generación de datos, para que el proceso consumidor pueda atenderlos inmediatamente.

Se pueden distinguir dos tipos de tuberías:

- **Tubería sin nombre (pipe):** las tuberías sin nombre tienen asociado un fichero en memoria principal, por lo tanto son temporales y se eliminan cuando no están siendo usadas ni por productores ni por consumidores. Son mucho más rápidas pero tienen la característica de la temporalidad.
- **Tubería con nombre:** la diferencia con respecto a las tuberías sin nombre es que en este caso la comunicación sí es a través del sistema de archivos, con lo que desaparece el carácter temporal. El acceso es exactamente igual que cualquier otro archivo (*open, close, read and write*).



Actividades

4. Aparte del relativo inconveniente de la temporalidad en las tuberías pipe, señale si se le ocurre algún otro inconveniente del uso de este tipo de tuberías con respecto a las que sí tienen nombre.
5. Si se ha cerrado el extremo de lectura al intentar escribir el proceso fallará. Comente cuál sería la señal que se generaría.

A continuación, se describen los servicios que permiten crear y acceder a los datos de un pipe.

Creación de un pipe

El servicio que permite crear un pipe es el siguiente:

- **int pipe(int fildes[2]);**
- **fildes[0]**, descriptor de archivo para leer del pipe.
 - **fildes[1]**, descriptor de archivo para escribir en el pipe.

La llamada pipe devuelve 0 si fue bien y -1 en caso de error.

Cierre de un pipe

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

- **int close(int fd);**

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

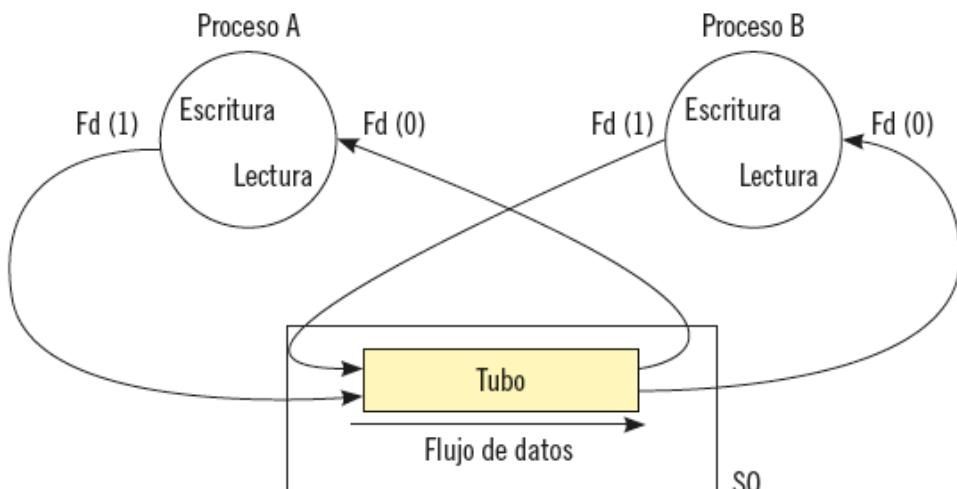
Escritura en un pipe

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

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

- **fd**: descriptor del archivo donde se escribe.
- **buffer**: identifica el buffer donde se encuentran los datos a escribir en el pipe.
- **n**: número de bytes a escribir.

Tubería entre dos procesos



El funcionamiento de la escritura sería el siguiente:

- Si la tubería está llena, la operación bloquea al proceso escritor hasta que se pueda llevar a cabo.

- Si no hay ningún proceso dispuesto para la lectura, la operación devuelve el correspondiente error. Se envía a través de la señal **SIGPIPE**.

El proceso de escritura se realiza en exclusión mutua, es decir, solo un proceso podrá escribir en la tubería. Si dos procesos lo intentan al mismo tiempo, uno de ellos se bloqueará hasta que termine el otro.

Lectura de un pipe

Para leer datos de un pipe se utiliza el siguiente servicio, que es el mismo utilizado para el tratamiento con archivos.

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

- **fd:** descriptor del archivo donde se lee.
- **buffer:** identifica el buffer donde se encuentran los datos a leer del pipe.
- **n:** número de bytes a leer.

La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve -1.

Al igual que las escrituras, las operaciones de lectura también son en exclusión mútua.

4.2. Semáforos

Los semáforos son componentes desarrollados a bajo nivel y sirven para asegurar el acceso correcto a un recurso compartido, es decir, **soluciona el problema de la exclusión mutua.**

Un semáforo tiene la siguiente estructura:

- Conjunto de valores que puede tomar.
- Conjunto de operaciones que admite.
- Lista de procesos que esperan para acceder al semáforo.

Con los semáforos también se consigue **resolver los problemas de sincronización entre procesos.**

Los valores que puede tomar un semáforo son enteros no negativos (o vale 0 o es **un valor entero positivo**). Cuando tiene asignado el valor 0, indica que el semáforo está cerrado y por lo tanto no se podrá entrar en la sección crítica; y cuando es >0 está abierto, y con el paso abierto.

Según los valores que puede tomar se clasifican los semáforos en:

- **Semáforos binarios:** son aquellos que solo pueden tomar los valores 0 y 1.
- **Semáforos generales:** son aquellos que pueden tomar cualquier valor no negativo.

La elección del tipo de semáforo depende del problema concurrente que tenga que solucionar.

Los semáforos admiten dos operaciones:

- **Operación Wait (P):** si el valor del semáforo no es nulo, esta operación decremente en uno el valor del semáforo. En el caso de que su valor sea nulo, la operación suspende el proceso que lo ejecuta y lo ubica en la lista del semáforo a la espera de que deje de ser nulo el valor.
- **Operación Signal (V):** se incrementa el valor del semáforo. Uno de los procesos bloqueados en la lista de espera del semáforo podrá finalizar su operación **wait** y continuar su ejecución.



Nota

La operación de chequear el valor del semáforo, y su actualización (incrementar o decrementar) es segura respecto a otros accesos concurrentes. Nunca dos procesos van a poder modificar su valor al mismo tiempo.

Operación wait (P)

La operación **wait** realiza las siguientes operaciones:

- **if Semáforo > 0**
- **then Semaforo:= Semaforo -1**
- **else Suspende al proceso en la lista de procesos del semáforo;**

Esta operación como indica su nombre es una potencial causa de retraso. Sin embargo, no siempre es así. Si en el momento de ejecutar la operación no hay ningún proceso en el uso de dicho semáforo, el proceso que lo llamó puede seguir con su ejecución.

Operación signal (V)

La operación signal realiza las siguientes operaciones:

- **if**: hay algún proceso en la lista del semáforo
- **then**: activa uno de ellos
- **else p:= p+1;**

Esta operación nunca provocará la suspensión del hilo. La elección del proceso que se activará de la lista de procesos seguirá la política utilizada por el programador (FIFO, LIFO, Prioridad, etc.).

A continuación, se muestra un ejemplo de exclusión mutua con semáforos:



Ejemplo

```
program Exclusion_Mutua;
    var semaforo: binsemaphore;
    process type Proceso;
    begin
        repeat
            wait(semaforo);
            (* Código de la sección crítica *)
            signal(semaforo);
        forever;
    end;
    var p, q: Proceso;
    begin
        initial(semaforo,1);
        cobegin p; q; coend;
    end;
```

En el código anterior se muestra cómo utilizando semáforos se consigue que la sección crítica se ejecute en exclusión mútua. En el programa principal, se lanzan los procesos **p**, y **q**, de manera concurrente (eso se indica con la sección de código **cobegin --- coend**), después de haber inicializado el semáforo a 1.

En este caso se ha usado un semáforo binario. Los procesos comenzarán al mismo tiempo, pero solo uno de ellos podrá entrar en la sección crítica. El resto irá esperando, a medida de que el proceso que haya conseguido obtener el semáforo abierto termine.



Aplicación práctica

En un programa concurrente de acceso a un dispositivo se tienen dos tareas:

- | Tarea X: acceder al estado del dispositivo.
- | Tarea Y: reiniciar el dispositivo, si se cumple una condición.

Sin embargo, no se puede hacer la Tarea Y sin antes haber realizado la Tarea X.

¿Cómo se podría aplicar la funcionalidad de los semáforos para conseguir esa sincronización?

SOLUCIÓN

Proceso A, quiere ejecutar la Tarea X.

Proceso B, quiere ejecutar la Tarea Y.

Mux, es la variable común semáforo, inicializada en 0.

En proceso A:

- | X;
- | Signal(Mux);

En proceso B:

- | Wait(Mux);
- | Y;

Debido a que el semáforo está inicializado a 0, hasta que no se ejecute X, y se incremente Mux, la tarea Y no podrá ejecutarse.

Los semáforos presentan las siguientes **ventajas**:

- Resuelven todos los problemas que presenta la concurrencia.
- Estructuras muy simples.
- Fácil de entender su funcionamiento.
- Tienen implementación muy eficiente.

En cuanto a los **peligros** que presentan los semáforos, se distinguen los siguientes:

- Son de muy bajo nivel y fácilmente pueden generar bloqueos con una mala programación.
- La depuración de los errores en su gestión es muy difícil.



Actividades

6. A lo largo del capítulo se ha estudiado un mecanismo que ayudaría a paliar en cierta medida el problema del bloqueo indefinido del ejemplo de los filósofos. Comente si sabría identificarlo y cómo aplicarlo.

4.3. Compartición de memoria

Este mecanismo de comunicación entre procesos se basa en compartir variables entre las tareas concurrentes. Se implementa a través de regiones críticas.

Las regiones críticas son bloques de código que son declarados como secciones críticas respecto de una variable. El compilador es el encargado de que esas instrucciones se ejecuten en exclusión mutua, respecto a otras regiones críticas declaradas con la misma variable compartida.

Una variable compartida únicamente puede ser modificada dentro de las regiones críticas.

Es el compilador el encargado de incluir los mecanismos necesarios para garantizar que la variable compartida sea modificada por un único proceso al mismo tiempo. Es un nivel de abstracción superior al de los semáforos.



Nota

Regiones críticas relativas a variables compartidas diferentes se pueden ejecutar concurrentemente.

Según el concepto propuesto por Brinch Hansen (1972), las regiones críticas se construyen con dos componentes:

- Una forma de declaración de variables compartidas (shared),

```
var variable : shared Tipo_variable;  
donde el Tipo_variable puede ser de cualquier  
tipo escalar o estructurado.
```

- Una sentencia estructurada que permite el acceso a una variable compartida, y que contiene un bloque de programa que se ejecuta en exclusión mútua, una vez que se ha obtenido el control de la variable compartida. El formato que se utiliza es:

```
region Variable do Bloque_de_programa;
```

Los procesos para entrar en la región crítica primero tienen que ganar el acceso a la variable compartida. Si lo ganan, ejecutan su bloque de programa; pero si lo pierden, pasan a la lista de procesos en espera de conseguir el acceso a la variable.

Cuando el proceso finaliza su ejecución libera la variable, enviando una señal que habilita de nuevo el acceso a la misma.

Si se aplican a las regiones críticas para solucionar el problema de la cena de filósofos que se veía en el ejemplo, se tendría lo siguiente:

```
Var cubiertos: shared Tipo_cubiertos;  
region cubiertos do  
begin  
    Comer ();  
end
```

Donde cubiertos sería la variable compartida. En este caso, los cubiertos identifican los tenedores de la izquierda y de la derecha.



Actividades

7. Indique una tarea concurrente donde sea más efectivo utilizar regiones críticas frente a semáforos.

Las **regiones críticas** tal como se han definido son útiles y simples, pero no proporcionan solución a muchos requerimientos de la interacción entre procesos, sobre todo en mecanismos de sincronización. Para solucionar esto se extiende el concepto a **regiones críticas condicionales (CCR)**.

Su definición sería la siguiente:

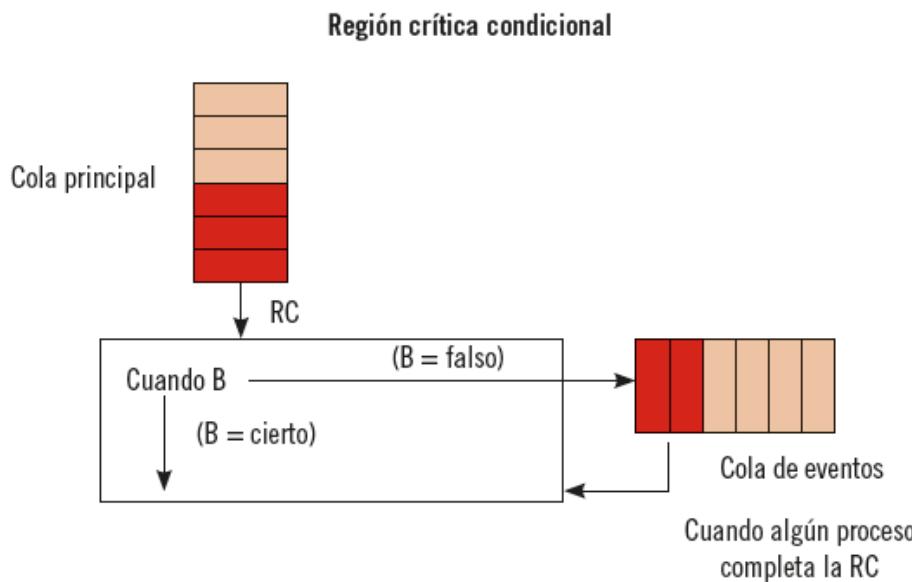
- La variable compartida se define igual que en la región crítica.
- Al bloque de programa se le asigna una condición para que pueda ser ejecutado.
De tal forma que se ejecutará esa región crítica solo si la condición se cumple.

El formato nuevo quedaría así:

```
region Variable_compartida when Expresión_booleana do
    Bloque_de_sentencias;
```

La **Expresión_booleana** de guarda se refiere habitualmente a la variable compartida, y se evalúa dentro del régimen de exclusión mutua.

Graficamente sería así:



Como se ha comentado anteriormente, las regiones críticas suben un nivel de abstracción sobre los semáforos, sin embargo presentan los siguientes inconvenientes:

region Variable compartida when Expresión_booleana do Bloque de sentencias;
--

- Sentencias dispersas por todo el código del programa.
- La integridad de la variable compartida puede ser fácilmente dañada.
- Estructura más compleja. Deben existir dos listas: una para llevar el control de los procesos que esperan conseguir la variable compartida y otra lista donde se encuentran los procesos que tras acceder a la variable no cumplen la condición impuesta.

Como aplicación de las regiones críticas condicionales, se va a solucionar el problema del productor y consumidor. En este problema, como se vio en el apartado de tuberías, un proceso genera datos y el otro proceso es el que los va recogiendo.



Ejemplo

```

proceso productor;
begin
    loop
        region buf when buffer.size < N do
            operaciones de producción;
        end region;
    end loop;
end;
proceso consumidor;
begin
    loop
        region buf when buffer.size > 0 do
            operaciones de consumición;
        end region;
    end loop;
end;
cobegin // programa principal
    productor();
    consumidor();
coend

```

4.4. Mensajes

En este caso, la sincronización y comunicación entre procesos es a base de envío y recepción de mensajes. Con este sistema se pretende evitar compartir variables como se vio anteriormente, y principalmente se utilizan en entornos distribuidos, es decir, en máquinas distribuidas en redes.

Para establecer la comunicación básicamente se utilizan dos operadores:

- **Send** (destino, mensaje): envío del mensaje.
- **Receive** (origen, mensaje): recepción del mensaje.

El mensaje puede tener una longitud fija o variable, y cuando dos procesos llegan a comunicarse se dice que han establecido un enlace.

Hay tres aspectos de interés, que hay que estudiar en el envío de mensajes, que son:

- La sincronización.
- La identificación del proceso emisor y receptor.
- La estructura de los mensajes.

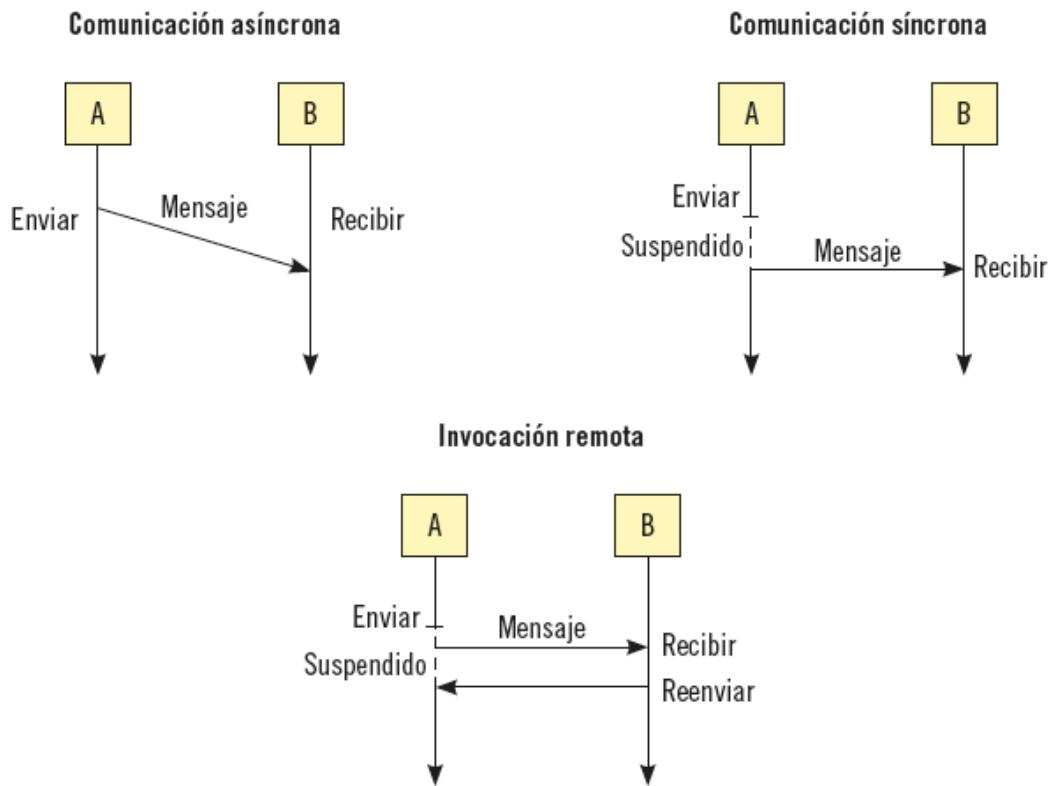
Sincronización

En la comunicación entre procesos a través de mensajes se identifica el proceso emisor, que emite el mensaje, y un proceso receptor, que lo recibe. El proceso receptor siempre espera si el mensaje no ha llegado todavía. Por lo tanto, este proceso tiene que tener implementado un mecanismo de escucha permanente.

Para el proceso emisor hay tres modelos básicos:

- Comunicación asíncrona: el emisor continúa su ejecución, y no tiene que esperar a que el receptor le envíe una señal de que le llegó.
- Comunicación síncrona (cita): el emisor espera a que el receptor reciba el mensaje.
- Invocación remota (cita extendida): el emisor espera a que el receptor reciba el mensaje, y la respuesta de este.

Haciendo analogías con situaciones de la vida cotidiana, en una comunicación asíncrona se puede echar una carta, y la persona que la echa sigue con su vida cotidiana, sin esperar a que se reciba dicha carta. En cuanto a una comunicación síncrona, se tiene la comunicación por teléfono. Aquí, el emisor espera que se realice el contacto y se verifique la identidad del receptor antes de enviar el mensaje.



Identificación del proceso emisor y receptor

Según se identifican los procesos en la comunicación se tiene la siguiente clasificación:

• Identificación directa o indirecta:

■ Directa. El emisor identifica explícitamente el receptor:

```
send mensaje to proceso
```

■ Indirecta. Se utiliza un intermediario (buzón, canal, tubería, etc.):

```
send mensaje to buzón
```

• **Simetría:**

■ **Comunicación simétrica.** El emisor identifica el receptor, y viceversa:

```
send mensaje to proceso (buzón)  
receive mensaje from proceso (buzón)
```

■ **Comunicación asimétrica.** El receptor acepta mensajes de cualquier emisor o buzón. Esta comunicación es típica de los servidores.

Estructura de los mensajes

Idealmente un lenguaje debería permitir que cualquier estructura de datos sea transmitida en un mensaje. Sin embargo, esto es complejo, pues se puede encontrar una representación diferente en el emisor y en el receptor, y más difícil aún si se incluyen punteros (Herlihy y Liskov, 1982).

Por estas dificultades, algunos lenguajes han restringido el contenido de los mensajes a objetos fijos no estructurados de tipo definido por el sistema. A continuación, se muestra el problema de productores consumidores utilizando el envío de mensajes:



Ejemplo

```
Productor
REPETIR
    Producir elemento;
    RECEIVE (Consumidor, mensaje);
    Construir mensaje;
    SEND (Consumidor, mensaje);

Consumidor
REPETIR
    RECEIVE (Productor, mensaje);
    Extraer mensaje;
    Consumir elemento;
    SEND (Productor, mensaje);
```

La comunicación entre procesos usando la mensajería es la base de la programación en red, permitiendo crear diferentes modelos de desarrollo.

5. Sincronización

La sincronización es uno de los problemas a solucionar en la programación concurrente, junto con la exclusión mutua. Esta sincronización es fundamental para que el proceso se ejecute de una manera óptima, intercalando la ejecución de las diferentes tareas si se trata de un entorno monoprocesador, o en cada uno de los procesadores si el programa se está ejecutando en una arquitectura multiprocesador.

Para dicha sincronización existen funciones y métodos que permiten la comunicación entre hilos.

5.1. Funciones de sincronización entre hilos

Las aplicaciones de múltiples hilos permiten al programador dividir un ejecutable en varias subtareas más pequeñas que se ejecutarán independientemente. En este apartado se van a describir algunas funciones relacionadas con la creación y sincronización de hilos, tanto en *Windows* como en *Unix*. Para un listado completo se remite al manual del sistema operativo.

Función API Windows	Resultado
CreateThread()	Crea un hilo.
SetThreadPriority()	Define la prioridad de un hilo.
SuspendThread()	Aumenta el contador de espera del hilo y detiene la ejecución.
ResumeThread()	Reduce el contador de espera y si hay hilos que están esperando los libera.
Sleep()	Detiene la ejecución del hilo durante un tiempo determinado. Corresponde con el mecanismo temporizador visto en apartados anteriores.
SleepEx()	Versión ampliada de Sleep, que se utiliza para en el funcionamiento simultáneo con operaciones I/O simultáneas.
ExitThread()	Finaliza el hilo actual.
TerminateThread()	Finaliza desde fuera un determinado hilo del proceso actual.
GetCurrentThread()	Proporciona un manejador del hilo actual.
GetCurrentThreadId()	Proporciona el id del hilo actual.
GetExitCodeThread()	Proporciona el código de salida de un hilo determinado.
GetThreadPriority()	Proporciona la prioridad de un hilo.
AttachThreadInput()	Desvía los mensajes de un hilo a otro.
Función librerías UNIX	Resultado
pthread_create	Crea un hilo.
pthread_detach	Elimina un hilo.
pthread_mutex_t refLock	Para declarar un sincronizador.
pthread_mutex_init	Para iniciar un sincronizador.
pthread_mutex_destroy	Para destruir un sincronizador.
pthread_mutex_lock	Para bloquear un segmento de código.
pthread_mutex_unlock	Para desbloquear un segmento de código.
usleep	Para generar una pausa en el thread.

Todo proceso comienza con la creación de un hilo utilizando la función **CreateThread()** en *Windows* o **pthread_create** en *Unix*, la cual proporciona un manejador del nuevo hilo creado con éxito. A través de este manejador, ya se puede acceder al hilo, y con las funciones indicadas anteriormente se puede ir obteniendo información de su estado e interactuar con él. Si una determinada función no se ejecuta con éxito se obtiene el valor **NULL**, y a través de funciones se puede lograr una descripción más detallada del error que se ha producido.



Nota

En ningún caso se pueden utilizar las funciones de finalización de hilos para terminar un hilo de otro proceso.

Cuando un proceso se inicia, a todos sus hilos se les asignan una prioridad normal. A medida que el proceso se va ejecutando las prioridades puede ir cambiando, según necesidades operativas. Para ello se usa la función **SetThreadPriority()**.

Tomando como referencia la **prioridad normal**, se puede tener el siguiente listado de prioridades:

- Priority_Lowest: -2.
- Priority_below_normal: -1.
- Priority_normal: 0.
- Priority_above_normal:+1.
- Priority_highest:+2.

Con ayuda de la función **SuspendThread()** se puede poner fuera de servicio al propio hilo que lo llama o a otro si se dispone de su manejador. El hilo queda parado, y no retoma la ejecución hasta que no se llama a la función **ResumeThread()**, con el manejador en cuestión.

La función **SuspendThread()** deja sin servicio durante un periodo indeterminado al hilo; sin embargo, si se quiere dejar suspendido un hilo durante un periodo fijo se usará la función **Sleep()**. Esta función solo se puede suspender a sí misma y no a otro hilo del proceso.

Un caso especial es la función **SleepEx()**. En este caso, el hilo se suspende por el tiempo determinado en el parámetro de entrada; sin embargo, si se produce algún evento de entrada o salida se "despierta" el hilo, retomando su ejecución.



Actividades

8. Señale cuánto tiempo cree que estará detenido un hilo si se define el tiempo de espera a 0.
9. Detalle un ejemplo en el que sea interesante utilizar la función AttachThreadInput().

5.2. Problemas de sincronización. Bloqueos (Deadlocks)

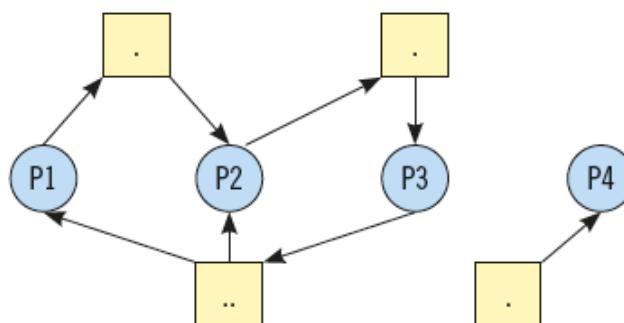
Si se busca la definición de bloqueo, se puede encontrar lo siguiente:

En sistemas operativos, el bloqueo mutuo (también conocido como interbloqueo, traba mortal, deadlock o abrazo mortal) es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente, que compiten por recursos del sistema o bien se comunican entre ellos. A diferencia de otros problemas de concurrencia de procesos, no existe una solución general para los interbloqueos.

Como muy bien se indica, es un problema crítico en la programación concurrente. Para ello, se deben utilizar algoritmos que minimicen al máximo, si no en su totalidad, la posibilidad de que aparezcan esos bloqueos.

El bloqueo surge fácilmente en el momento en el que dos procesos hagan uso de recursos que se necesitan al mismo tiempo. Por ejemplo, en el caso de los filósofos, necesitan tener los dos tenedores que se encuentran a su lado para poder comer.

Los procesos 1, 2 y 3 están en interbloqueo



Un sistema informático está compuesto por un número limitado de recursos, que son solicitados por los procesos que están en competición en un programa concurrente. El espacio de memoria, los ciclos de CPU, los archivos y los dispositivos de E/S constituyen ejemplos de tipos de recursos.

Cuando un proceso quiere emplear un recurso debe seguir esta secuencia:

1. Solicitud.
2. Uso.
3. Liberación.

Los pasos 1 y 3, o sea, la solicitud y liberación de los recursos, son llamadas al sistema. El problema crítico se da cuando dos procesos están en la secuencia de uso y necesitan utilizar otro recurso que al mismo tiempo está siendo utilizado.

El interbloqueo se puede abordar de las tres formas siguientes:

- Se puede emplear un protocolo para impedir o evitar los interbloqueos, asegurando que el sistema nunca entre en estado de interbloqueo.
- Se puede permitir que el sistema entre en estado interbloqueo, detectarlo y realizar una recuperación.
- Se puede ignorar el problema y actuar como si nunca se produjeran interbloqueos en el sistema.

La tercera solución es la que utilizan la mayoría de los sistemas operativos, y es conocida como el algoritmo del aveSTRUZ. Se fundamenta en que raramente se producen bloqueos, y el coste de montar un mecanismo de evitación es muy elevado para las veces que ocurre.

Prevención de interbloqueos

Para impedir los interbloqueos se tiene que conseguir que cualquiera de las siguientes cuatro condiciones no se cumpla. Si se consigue que una de ellas no ocurra, ya no se producirá el interbloqueo:

- **Exclusión mutua.** Asignar un recurso de manera exclusiva.
- **Retención y espera.** Un proceso tiene un recurso y está en espera de otro para llevar a cabo su tarea.
- **Sin desalojo.** No se desalojan los recursos de un proceso por parte del sistema. Solo voluntariamente desaloja dicho recurso.
- **Espera circular.** El primer proceso espera de manera circular a una sucesión de procesos, y el último espera al primero.

Evasión de interbloqueos

Los algoritmos de prevención de interbloqueos lo que buscan es que al menos una de las condiciones no se cumpla.

Sin embargo, en muchos casos los algoritmos no están optimizados para hacer el mejor uso de los recursos y evitar esperas innecesarias, con lo que se desaprovecha toda la potencia de la concurrencia. Un posible método para evitar interbloqueos sería recabar más información acerca de cómo se prevé que va a ser la solicitud de los recursos, y de esta manera optimizar el algoritmo.

Detección de interbloqueos

Si un sistema no emplea ni algoritmos de prevención ni de evasión de interbloqueos, entonces puede producirse una situación de interbloqueo en el sistema. En este caso el sistema debe proporcionar:

- Un algoritmo que examine el estado del sistema para determinar si se ha producido un interbloqueo.
- Un algoritmo para recuperarse del interbloqueo.

Recuperación de interbloqueos

Cuando se detecta un interbloqueo existen varias alternativas para que el sistema se recupere. Una posibilidad es informar al usuario para que lo trate de manera manual, y otra que el sistema de manera automática intente rehacerse. En este caso existen dos opciones para romper el interbloqueo:

- Una de ellas consiste en interrumpir uno o más procesos para romper la cadena en espera circular. De esta forma el proceso libera los recursos y el resto puede concluir.
- La otra opción consiste en quitar recursos de uno o más de los procesos bloqueados.



Aplicación práctica

Se ha desarrollado un programa concurrente, pero en ningún momento se ha estudiado la posibilidad de que ocurriera un interbloqueo. A modo de ejemplo, se podría pensar en la comida de los filósofos planteada a principio del capítulo. ¿Cómo actuaría para solucionar el interbloqueo y que el programa pudiera seguir ejecutándose?

SOLUCIÓN

En principio, lo ideal sería haber diseñado un algoritmo que evitara que se produjera el interbloqueo. Por ejemplo, obligar a tener los dos tenedores para comer y si no se tienen después de un tiempo intentando obtener el segundo tenedor, soltarlo para que otro filósofo comiera.

Pero en el caso que ocupa, ya se ha producido el interbloqueo y ahora se tiene que diseñar un algoritmo de recuperación.

Una opción sería obligar al filósofo que haya comido hace menos tiempo que devuelva el tenedor a la mesa, para que otro filósofo pudiera comer.

6. Acceso a dispositivos

En un ordenador el elemento principal de computación es la CPU, que es la encargada de procesar las instrucciones y los datos de un programa. En comunicación con ella están los dispositivos de E/S.

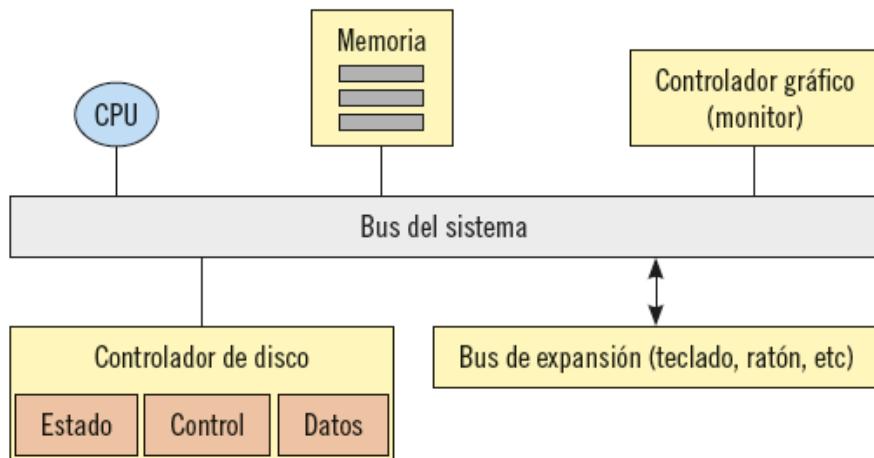
Dentro de los dispositivos se puede hacer la siguiente clasificación:

- **Periféricos:** permiten la comunicación entre los usuarios y la computadora. Por ejemplo:
 - Entrada: teclado, ratón...
 - Salida: impresora, pantalla...
- **Dispositivos de almacenamiento:** proporcionan almacenamiento no volátil de datos y memoria. Entre estos destacan:
 - Almacenamiento secundario: discos y disquetes.
 - Almacenamiento terciario: cintas y sistemas de archivo.
- **Dispositivos de comunicaciones:** conectan el ordenador con otros ordenadores a través de una red. Por ejemplo: tarjetas de red, módems, etc.

El acceso por parte de la CPU a estos dispositivos se realiza a través de **controladores**, que crean una capa de abstracción *software* que se coloca entre el procesador y el dispositivo.

La comunicación es a través de los registros del controlador. Existen registros de datos, de estado y de control. En los de datos se van recibiendo o enviando los datos de entrada o salida; en los de estado se puede ir conociendo el buen funcionamiento o no del dispositivo; y en los de control están las instrucciones a realizar.

Comunicación entre la CPU y dispositivos



El problema asociado a todos los dispositivos es que son muy lentos en comparación con la capacidad de procesamiento de la CPU, y por ello se hace necesario mecanismos que permitan sincronizar ambos elementos. Por ejemplo, la diferencia de velocidad del procesador con respecto a la impresión de un documento.

Los controladores de dispositivos se pueden clasificar según criterios de E/S, transferencia o interacción:

- Dirección E/S:
 - Dispositivos conectados por puertos.
 - Dispositivos proyectados en memoria.
- Unidad de transferencia:
 - Dispositivos de bloques: discos o cintas.
 - Dispositivos de caracteres: terminales o impresoras.
- Interacción computadora-controlador:
 - Entrada/salida programada.
 - Entrada/salida por interrupciones.
 - Acceso directo a memoria (DMA).



Nota

El sistema de E/S programada es idóneo en sistemas de tiempo real en los que la velocidad de E/S es rápida

En los dispositivos conectados por puertos, el controlador tiene asignado un puerto de E/S, que se identifica con posiciones de memorias diferenciadas, independientes y fuera del rango de posiciones de la memoria principal. Tienen operaciones específicas para interactuar con el dispositivo.

En el caso de controladores proyectados en memoria, se usa el espacio de direcciones de la memoria principal, compartiéndola con ella.

Con respecto a las unidades de transferencia, los dispositivos de bloques se comunican a base de grupos de caracteres, y en el de caracteres a nivel de byte.

Entrada/salida programada

En este caso la CPU es la encargada de ir consultando continuamente si el dispositivo está listo para interactuar con el sistema. Mientras que el controlador no está listo el proceso queda en un bucle de espera activa. Cuando está listo, es cuando se hace la lectura o escritura del dispositivo.

La desventaja es que se producen grandes tiempos de espera.

Entrada/salida por interrupciones

El controlador cuando se quiere comunicar con el sistema operativo activa una interrupción, que permite la comunicación asíncrona. El sistema operativo podía estar haciendo otras tareas, y en el momento en el que recibe la interrupción y según la prioridad asignada interrumpe su proceso para atender al dispositivo. Esta es la base que permite implementar un sistema operativo multiprogramado.

Acceso directo a memoria (DMA)

Hasta ahora se ha visto que la CPU es la encargada de realizar la transferencia de datos entre el dispositivo y la memoria. Sin embargo, para los dispositivos de alta velocidad este mecanismo no es viable por su lentitud.

Para este tipo de dispositivos se usa el acceso directo a memoria (DMA), donde el dispositivo opera directamente sobre la memoria, siempre que la CPU le haya concedido el permiso para hacerlo.

En este caso se avisa a la CPU solo al comienzo y al final de una operación sobre memoria. De esta forma se consiguen tasas de transferencia de datos superiores a los otros métodos.

Además, esto permite la concurrencia de operaciones de E/S con operaciones de la CPU, aumentando el rendimiento global del sistema.



Importante

La CPU y el dispositivo no pueden utilizar el bus de direcciones y de datos al mismo tiempo. La concurrencia solo se dará cuando la CPU opere con datos de una memoria caché.

6.1. Funciones de lectura y escritura

Las funciones de lectura y escritura permiten comunicar con los dispositivos, leyendo datos en el caso de la lectura (por ejemplo, el teclado) y escribiendo datos en la escritura (por ejemplo, la impresora).

Las funciones se utilizan en la capa de aplicación y hay que diferenciar si el dispositivo está proyectado en memoria o conectado por puertos.

En el primer caso, las funciones son iguales a las que se utilizarían en un acceso a memoria normal.

Las funciones principales son: **b** de byte, **w** de word, y **l** (ele) de long

- **readb, readw y readl:** lee 1, 2 o 4 bytes, respectivamente, de una posición de memoria compartida E/S.
- **writeb, writew y writel:** escribe 1, 2 o 4 bytes, respectivamente, de una posición de memoria compartida E/S
- **memcpy_fromio y memcpy_toio:** se encargan de copiar bloques de memoria compartida a memoria dinámica y viceversa.
- **memset_io:** fija un valor para una determinada zona de memoria compartida.

Se consigue una independencia entre la máquina y el software manejador.

En el caso de dispositivos conectados por puertos, las funciones de lectura y escritura son especiales, pues acceden a direcciones de memoria independientes de la memoria principal.

Existen cuatro instrucciones, **in**, **ins**, **out** y **outs**, para acceder a los puertos. Las siguientes macros del kernel simplifican dichos accesos. Están declaradas en **include/asm-*/io.h**.

Las instrucciones de lectura son:

- **inb()**, **inw()** e **inl()**: sirven para leer 1, 2 o 4 bytes consecutivos de un puerto de E/S.
- **inb_p()**, **inw_p()** e **inl_p()**: igual que las anteriores, pero con el añadido de que tras la lectura realizan una pausa.
- **insb()**, **insw()** e **insl()**: se usan para leer secuencias de 1, 2 o 4 bytes contiguos, teniendo como parámetro la duración de la secuencia.

Las instrucciones de escritura son:

- **outb()**, **outw()** y **outl()**: sirven para escribir 1, 2 o 4 bytes consecutivos de un puerto de E/S.
- **outb_p()**, **outw_p()** y **outl_p()**: igual que las anteriores, pero con el añadido de que tras la escritura se hace una pausa.
- **outsb()**, **outsw()** y **outsl()**: se usan para escritura de secuencias de 1, 2 o 4 bytes contiguos, teniendo como parámetro la duración de la secuencia.



Aplicación práctica

Está diseñando un programa para enviar datos a una impresora. Sin embargo, se encuentra ante la situación de que la velocidad de la CPU es muy superior a la capacidad de recepción de datos de la impresora. ¿Qué mecanismo que se ha visto anteriormente se podría aplicar para paliar esa diferencia de velocidad?

SOLUCIÓN

El mecanismo a utilizar sería el de un almacenamiento intermedio o buffering a la entrada de la impresora.

De esta forma se consigue que la CPU envíe toda la información a un área de memoria o del disco, y la impresora vaya cogiendo los datos a su velocidad. Con esto también se logra que la CPU se libere y se pueda dedicar a otros procesos.

Ese buffer también podría ir absorbiendo datos de distintos procesos, de un entorno monoprocesador o multiprocesador.

6.2. Puertos de entrada y salida

Los puertos lógicos de entrada y salida son zonas o localizaciones de la memoria de un ordenador que se asocian con un puerto físico o con un canal de comunicación, y que proporcionan un espacio para el almacenamiento temporal de la información que se va a transferir entre la localización de memoria y el canal de comunicación.

Los puertos se identifican por números desde 1 hasta 65000, pudiendo llegar a más, siendo conocidos los puertos de 1 a 1024 como:

- **HTTP:** puerto 80 transferencia de hipertexto por Internet.
- **FTP:** puerto 20 transferencia de data (mp3, documentos, etc.).

Los puertos lógicos son, al igual que los puertos físicos (usb, hdmi o lpt), necesarios para que las aplicaciones se puedan comunicar con dispositivos, o con el exterior en el caso de conexiones de red. La diferencia es que se enlazan virtualmente en la conexión TCP con los programas, para tener una referencia, y que los otros programas puedan conectarse y traspasar información. Por ejemplo, se puede decir que el servidor web suele estar enlazado (escuchando) en el puerto 80, o que el navegador sale por el puerto 4000 para conectarse a este servidor.

Los puertos están divididos según sus funciones, así pueden distinguirse:

- **Puertos reservados:** puertos de 1 a 1024. Tienen funciones específicas que mandan los estándares. La organización que se encarga de establecer los estándares es la IANA (*Internet Assigned Numbers Authority*), que se puede encontrar en <<https://www.iana.org/>>. Por ejemplo, el 22 es para SSH (*Secure Shell*) y del 135 al 139, para la NetBIOS.
- **Puertos no estándar:** puertos de 1025 a 49151. La IANA se encarga de asignarlos a distintas aplicaciones que lo necesitan.
- **Puertos efímeros:** el resto de puertos hasta el 65536. Son los clientes (el navegador, el cliente de correo y el cliente de FTP) los que lo eligen aleatoriamente para establecer desde ellos la conexión a los puertos servidores, y si la conexión cae se liberan y pueden ser usados por cualquier otra aplicación o protocolo más tarde.



Actividades

10. Aplique un escáner de puertos a su propio PC e identifique los puertos que están a la escucha.
11. Identifique los puertos usados para el servicio de correo electrónico.

12. Señale si cree que el protocolo http y el https se comunican a través del mismo número de puerto, y razoné la respuesta.

Seguidamente se mencionan algunos puertos importantes:

- **21-FTP Data:** utilizado por servidores FTP (*File Transfer Protocol*) para la transmisión de datos en modo pasivo.
- **22-SSH:** puerto utilizado por *Secure Shell* (SSH), el cual es un protocolo y un programa que lo utiliza para acceder a máquinas remotas a través de una red. Además, funciona como una herramienta de transmisión de datos, desde ficheros sueltos hasta una sesión de FTP cifrado.
- **23-Telnet:** Telnet es una herramienta que proporciona una ventana de comandos, los cuales permiten controlar un PC de forma remota.
- **53-DNS:** este puerto lo utiliza el DNS (*Domain Name System*). Esta base de datos distribuida o jerárquica se encarga de traducir nombres de dominios a IP's.
- **80-http:** puerto que transmite el protocolo HTTP (*Hypertext Transfer Protocol*), que es el utilizado en cada transacción web (WWW).
- **135-RPC:** *Remote Procedure Call*. Este servicio es el encargado de administrar la comunicación con otro PC cuando un programa solicita ejecutar código en ese otro PC. De esta forma, el programador no tiene que preocuparse por esta conexión.
- **139-NetBIOS:** por este puerto funciona el servicio NetBIOS, que es el encargado de compartir ficheros del PC por la red interna.
- **5000-UpnP:** El *Universal Plug and Play* define protocolos y procedimientos comunes para garantizar la interoperatividad sobre PC's permitidos por red, aplicaciones y dispositivos inalámbricos.
- **8080-WebProxy:** este puerto lo pueden utilizar terceros para ocultar su verdadero IP a los servidores web.

7. Resumen

Se han detallado los conceptos fundamentales de la programación concurrente. El proceso es el elemento principal dentro de la concurrencia, y dentro de él los hilos y su sincronización.

La concurrencia es simulada en entornos monoprocesadores, pero real en un sistema multiprocesador, consigiéndose una programación paralela.

La comunicación y sincronización entre procesos es fundamental para, además de cumplir las especificaciones funcionales del programa, evitar situaciones de interbloqueo. En la programación concurrente se utilizan mecanismos para evitar los dos

grandes problemas que subyacen en este tipo de programaciones: la exclusión mútua y la sincronización. Diferentes mecanismos como semáforos, regiones críticas o envío de mensajes solucionan esos dos problemas.

Para las situaciones de interbloqueo se han visto técnicas de evitación, detección y recuperación. Estudiar el uso de recursos de los procesos ayuda a crear un algoritmo óptimo que evite bloqueos. Si por el contrario no compensa crear un sistema de evitación por lo esporádico de dicha situación, la detección y recuperación permitirán estabilizar el sistema.

También se ha estudiado el acceso a los dispositivos y cómo aplican el mecanismo de concurrencia en su comunicación, para evitar esperas innecesarias y optimizar la transmisión de datos entre CPU y dispositivos externos.

Por último, se ha mostrado una introducción a los puertos de entrada y salida y cómo se realiza la comunicación entre el ordenador y los dispositivos a través de diferentes mecanismos de sincronización.