Programación Concurrente

Conceptos básicos

Por José Mª Toribio Vicente
Universidad de Valladolid

edited by jsm
Barcelona, España. 7/11/2000

Comunicaciones entre procesos

En este capítulo vamos a intentar explicar los conceptos básicos necesarios para entender el significado de los hilos y la programación multihilo. Se explican, de forma genérica, los conceptos de sistema operativo, proceso, paralelismo, concurrencia y comunicación y sincronización de procesos. Para una explicación más detallada se puede consultar cualquier libro sobre conceptos de sistemas operativos como puede ser [TAN93].

Concepto de Sistema Operativo

La definición de sistema operativo es complicada, aunque todos los usuarios de computadoras tienen cierta experiencia con él. Su definición se basa en las dos funciones principales que realiza:

Sistema operativo como máquina extendida: La función del sistema operativo es presentar al usuario el equivalente de una máquina extendida o máquina virtual que sea más fácil de programar que el hardware subyacente, de forma que se le oculten características de bajo nivel relacionadas con el manejo de los discos, las interrupciones, cronómetros, control de memoria, etc.

Sistema operativo como controlador de recursos: La función del sistema operativo es proporcionar una asignación ordenada y controlada de los procesadores, memorias y dispositivos de E/S para los distintos programas que compiten por ellos. Es decir, el sistema operativo se encarga de llevar un registro de la utilización de los recursos, dar servicio a las solicitudes de recursos, contabilizar su uso y mediar entre las solicitudes en conflicto de los distintos programas y usuarios.
Los sistema operativos han ido evolucionando a lo largo de los años. La evolución de los sistemas operativos ha estado ligada con la arquitectura de las máquinas en las que se ejecutan :

La primera generación (1945-1955): Las primeras computadoras empleaban válvulas y su programación se realizaba directamente mediante conexiones o lenguaje máquina. No existían lenguajes de programación, ni sistemas operativos.

La segunda generación (1955-1965): Las computadoras se construían empleando transistores. Los programas se escribían en un lenguaje de programación (Ensamblador o Fortran) para después pasarlo a tarjetas perforadas. Se desarrollaron los primeros sistemas de procesamiento por lotes, que consistían en agrupar una serie de trabajos en un lote y pasárselos a la computadora para su resolución de forma secuencial, uno tras otro. Las grandes computadoras de esta época se dedicaban al cálculo científico y de ingeniería. Los sistemas operativos más habituales eran FMS (Fortran Monitor System) e IBSYS, el sistema operativo de IBM para la 7094 (la computadora más conocida de la época).
La tercera generación (1965-1978): La aparición de los circuitos integrados a pequeña escala favoreció la producción de máquinas más potentes, de dimensiones más reducidas, y de menor precio. Se desarrollaron familias de computadoras, como el sistema 360 de IBM, con el fin de lograr la compatibilidad entre las distintas máquinas de la familia, y poder emplear los mismos programas, incluso el sistema operativo, en toda la familia. Esto originó un sistema operativo monstruoso, el OS/360, que contenía miles de errores, y requería continuas revisiones que arreglaban una serie de errores, introduciendo muchos otros. Se generalizó la técnica de multiprogramación, que consistía en dividir la memoria en varias partes, con un trabajo distinto en cada una. Mientras un trabajo esperaba por E/S, otro podía emplear la CPU, lo cual mejoraba la utilización de la máquina. Para aislar cada una de las particiones se empleaba un hardware especial. Se inventó la técnica de spooling (Simultaneous Peripheral Operation On Line, Operación simultánea y en línea de periféricos), que se empleó para las entradas y salidas. Sin embargo, aunque los sistemas operativos de esta época eran adecuados para los cálculos científicos y el procesamiento de datos comerciales, seguían siendo esencialmente sistemas de procesamiento por lotes. A pesar de ello comenzaron los primeros desarrollos de sistemas de tiempo compartido, variante de la multiprogramación, como el sistema CTSS del MIT. Un diseño posterior de sistema de estas características fue MULTICS (MULTiplexed Information and Computing Service) cuyo objetivo era proporcionar una máquina para cientos de usuarios empleando tiempo compartido. MULTICS fue un sistema que fracasó y se abandonó. Durante esta época, se desarrollaron la familia de minicomputadoras DEC PDP-1 hasta PDP-11. Ken Thompson escribió una versión especial de MULTICS para un usuario en una PDP-7, que más tarde daría origen a UNIX.

La cuarta generación (1978-1991): Se desarrollaron los circuitos integrados del tipo LSI y VLSI (Very Large Scale Integración) que contenían miles y hasta millones de transistores en un centímetro cuadrado, lo que abarató los costes de producción. Se inventó de esta forma el microprocesador, que permitía a un usuario tener su propia computadora personal. Los sistemas operativos desarrollados han ido siendo cada vez más amigables, proporcionando entornos gráficos de trabajo. Los sistemas operativos más difundidos en estos años han sido MS-DOS de Microsoft funcionando en los chips de Intel, y UNIX funcionando en todo tipo de máquinas (especialmente con chips RISC). A mediados de los 80, aparecen las primeras arquitecturas con computadoras en paralelo que emplean memoria compartida o distribuida, y hardware vectorial, así como

los primeros sistemas operativos de multiproceso, lenguajes y compiladores especiales para estos sistemas.

La quinta generación (1990-): La tecnología del proceso paralelo comienza a madurar y se inicia la producción de máquinas comerciales. Los sistemas MPP (Massively Paralell Processing) son los sistemas paralelos en los que se vuelca la investigación. El desarrollo progresivo de las redes de computadoras, y la idea de la interconexión y compartición de recursos, ha promovido el desarrollo de sistemas operativos de red y sistemas operativos distribuidos. En los primeros, el usuario es consciente de la existencia de otras máquinas a las que puede conectarse de forma remota. Cada máquina ejecuta su propio sistema operativo local y tiene su propio grupo de usuarios. En cambio, en un sistema operativo distribuido, el usuario percibe el sistema como un todo, sin conocer el número de máquinas, ni su localización, de esta forma los usuarios no son conscientes del lugar donde se ejecuta su programa, ni donde se encuentran sus archivos físicamente; todo ello es manejado de forma automática por el sistema operativo. Los sistemas operativos de red no tienen grandes diferencias respecto a los sistemas operativos de un solo procesador, ya que sólo necesitan una serie de características adicionales que no modifican la estructura esencial del sistema operativo. En cambio, los sistemas operativos distribuidos necesitan una serie de modificaciones de mayor importancia, ya que esencialmente se diferencian de los sistemas centralizados en la posibilidad de cómputo en varios procesadores a la vez.
La organización de los sistemas operativos ha ido evolucionando también, al igual que lo ha ido haciendo el hardware. Se pueden distinguir una serie de organizaciones muy relacionadas con la época de cada sistema operativo:

1) **Estructura monolítica**: Los primeros sistemas operativos tenían una organización o estructura monolítica. No poseían estructura alguna. El sistema operativo constaba de una serie de procedimientos., cada uno de los cuales puede llamar al resto cuando sea necesario. Cada procedimiento del sistema tiene una interfaz bien definida. Los servicios que proporciona el sistema operativo se solicitan colocando los parámetros en lugares bien definidos (registros o pila), para después ejecutar una instrucción especial de trampa, una llamada al núcleo o una llamada al supervisor. Esta instrucción cambia la máquina del modo usuario al modo supervisor (o modo kernel), y transfiere el control al sistema operativo.

Así una estructura simple para un sistema monolítico sería la siguiente:

a) Un programa principal que realiza una llamada al procedimiento de servicio solicitado.

b) Un conjunto de procedimientos de servicio que realizan las llamadas al sistema.

c) Un conjunto de procedimientos de utilidad que ayudan al procedimiento de servicio.

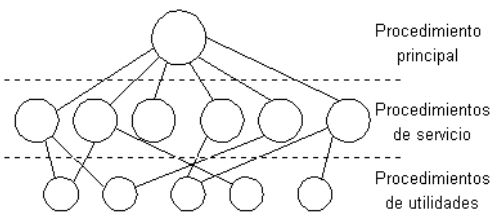


Figura 2-1 Estructura simple de un sistema operativo monolítico

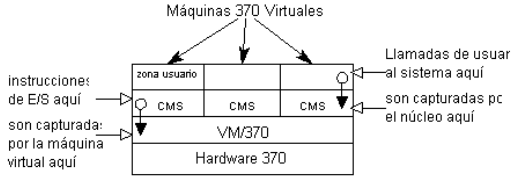
2) *Estructura en capas*: Es una generalización del modelo mostrado en la figura anterior. Se trata de organizar el sistema operativo como una jerarquía de capas, cada una construida sobre la anterior. Un ejemplo de esta estructura es el sistema operativo THE, desarrollado en Holanda por E.W. Dijkstra y un grupo de estudiantes en 1968.

5 El operador
4 Programas de usuario
3 Gestión de entrada/salida
2 Comunicaciones operador/proceso
1 Gestión de memoria y disco
0 Asignación del procesador y multiprogramación

Una generalización avanzada del concepto de capas se realizó en el sistema operativo MULTICS, que se organizó en anillos concéntricos. Los anillos interiores son los de mayores privilegios.

3) *Estructura de máquinas virtuales*: El origen de esta estructura se encuentra en el sistema operativo CP/CMS, que después se llamó VM/370, diseñado para el sistema 360/370 de IBM. Es un sistema de tiempo compartido que proporciona multiprogramación y una máquina extendida con una interfaz más agradable que el propio hardware, separando estas dos funciones de forma clara.
Figura 2-3 Estructura del sistema operativo VM/370 con CMS.

El sistema posee un **monitor de máquina virtual** que se ejecuta en el hardware y realiza la multiprogramación, proporcionando varias máquinas virtuales, no como

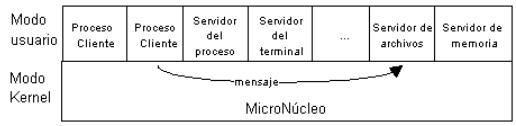


máquinas extendidas, sino como copias exactas del hardware simple, con su estructura de modo núcleo y modo usuario, sus interrupciones, etc. al igual que la máquina real. De esta forma cada máquina virtual, por ser idéntica a la real, puede ejecutar cualquier sistema operativo que funcione directamente sobre el hardware, pudiendo tener varios sistemas operativos funcionando al mismo tiempo. CMS (Conversational Monitor System) es un sistema interactivo monousuario que solía ejecutarse sobre cada máquina virtual.

Al emplear este tipo de organización se logra una mayor sencillez de diseño, flexibilidad y facilidad de mantenimiento.

4) *Estructura Cliente/Servidor*: La tendencia actual en el diseño de sistemas operativos es la de mover la mayor cantidad de código posible hacia capas superiores, con el objetivo de conseguir un núcleo reducido del sistema operativo, un micronúcleo o microkernel. Se trata de implementar la mayoría de las funciones del sistema en procesos de usuario, lo que descarga al núcleo y aumenta la seguridad del sistema. Se implementa un modelo cliente/servidor para la resolución de las llamadas al sistema. Así, si un proceso de usuario

(proceso cliente) desea leer un bloque de archivo, envía una solicitud a un proceso servidor que realiza la tarea y



devuelve la respuesta. El núcleo sólo se encarga de la comunicación entre cliente y servidor.
Figura 2-4 Estructura Cliente/Servidor de un sistema operativo.

El modelo cliente/servidor es también muy utilizado en el diseño de los nuevos sistemas operativos distribuidos, ya que los mensajes que se envían entre procesos pueden ser locales o remotos, pero la abstracción para el cliente es la misma, pues él no conoce el tipo de llamada, sólo sabe que envía una solicitud y recibe un respuesta.

Concepto de Proceso

El concepto central de todo sistema operativo es el proceso: abstracción de un programa en ejecución. Todo lo demás se organiza en relación a este concepto.

Los ordenadores modernos realizan varias tareas al mismo tiempo. Por ejemplo, mientras se ejecuta un programa de usuario, el ordenador puede estar leyendo del disco e imprimiendo un documento en la impresora. En un sistema de multiprogramación, la CPU alterna de un programa a otro, ejecutando cada uno durante milisegundos, y conmutando a otro inmediatamente, de tal forma que al usuario se le proporciona cierta sensación de ejecución paralela, como si el ordenador realizase varias tareas al mismo tiempo. Aunque, estrictamente, la CPU ejecuta en un determinado instante un solo programa, durante un segundo puede haber trabajado con varios de ellos, dando una apariencia de paralelismo. Es en estos casos cuando se tiende a hablar de seudoparalelismo, indicando la rápida conmutación entre los programas en la CPU, distinguiéndolo del paralelismo real de hardware, donde se realizan cálculos en la CPU a la vez que operan los dispositivos de E/S. Ya que es complicado controlar las distintas actividades paralelas, los diseñadores de sistemas emplean el modelo de procesos para facilitar la utilización del paralelismo.

Según este modelo, todo el software de la computadora se organiza en procesos secuenciales, o simplemente procesos. Un proceso es un programa en ejecución, incluyendo los valores del contador de programa (PC), registros y variables del programa. Conceptualmente, cada proceso tiene su propia CPU virtual. En realidad, lo que sucede es que la CPU física alterna entre los distintos procesos. Esta alternancia rápida es lo que se llama multiprogramación.

Si la CPU conmuta entre procesos, no es probable que un proceso que se vuelva ejecutar lo haga en las mismas condiciones en que lo hizo anteriormente. Por lo tanto, los procesos no se pueden programar con hipótesis implícitas a cerca del tiempo. El resultado de un proceso debe ser independiente de la velocidad de cálculo y del entorno en que se encuentre, de tal forma que sea reproducible.

La diferencia entre proceso y programa es sutil, pero crucial. El programa es el algoritmo expresado en una determinada notación, mientras que el proceso es la actividad, que tiene un programa, entrada, salida y estado. Una CPU puede compartirse entre varios procesos empleando un algoritmo de planificación, que determina cuando se debe detener un proceso en ejecución para dar servicio a otro distinto.

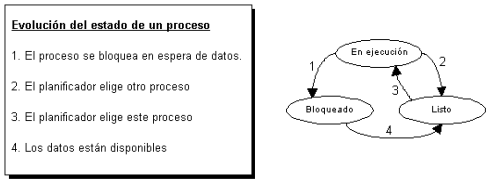
Normalmente los sistemas operativos modernos diseñados con una filosofía de kernel, y que emplean el concepto de proceso, ofrecen alguna forma de crear y destruir los procesos dinámicamente. En UNIX la creación de procesos se realiza mediante la llamada al sistema fork, la cual crea una copia idéntica del proceso que hace la llamada (proceso padre), tras la cual el proceso hijo sigue su ejecución en paralelo con el proceso padre. El padre puede crear nuevos hijos, al igual que el propio hijo, que puede crear nuevos hijos convirtiéndose en padre en relación a ellos. Se establece así una relación jerárquica de dependencia padre-hijo (árbol de procesos).

Cada proceso es una entidad independiente, con su contador de programa y su estado interno. Los 3 estados básicos en que puede encontrarse un proceso son:

1) En ejecución (RUNNING): El proceso está utilizando la CPU en el instante dado.

2) Listo (READY): El programa está en condición de ser ejecutado (pasar al estado En Ejecución) pero no lo está ya que se está ejecutando otro proceso temporalmente en la CPU.

3) Bloqueado (SUSPEND): El proceso no se puede ejecutar ya que se encuentra a la espera de un evento



externo, incluso aunque la CPU se encuentre libre.
Figura 2-5. Estados básicos de un proceso.

En el modelo de procesos conviven procesos de usuario con procesos del sistema. La parte del sistema operativo encarga de realizar la conmutación entre procesos, y de esta forma repartir la utilización de la CPU, es el planificador, el cual tiene una enorme importancia en el funcionamiento óptimo del sistema.

El modelo de procesos se implementa en el sistema operativo mediante una tabla de procesos, con una entrada por cada proceso. Cada entrada contiene información sobre el estado del proceso, el contador de programa (PC), el puntero de pila (SP), la asignación de memoria, el estado de los archivos abiertos, información contable, información sobre la planificación del proceso, y otros datos a conservar al producirse el cambio de contexto de proceso. En general son datos sobre la administración del proceso, la administración de memoria y la administración de archivos.

A continuación se presentan los campos más habituales que suelen encontrarse en la tabla de procesos en el núcleo de un sistema operativo ([BAC86] y [TAN93]):

La ilusión de varios procesos ejecutándose concurrentemente en una sola CPU con varios dispositivos de E/S se realiza de la siguiente forma:

Cada clase de dispositivo (discos duros, discos flexibles, reloj, terminal) tiene asociado una zona de memoria que se llama vector de interrupción, la cual contiene la dirección del procedimiento de servicio de la interrupción. Cuando se produce una interrupción, el contador de programa y la palabra de estado del proceso actual en ejecución son enviados a la pila por el hardware de la interrupción. Entonces, la máquina salta a la dirección indicada en el vector de interrupción correspondiente. Esto es lo que realiza el hardware.

El procedimiento de servicio de la interrupción guarda los registros en la entrada de la tabla de procesos correspondiente al proceso activo, se elimina la información depositada por el hardware en la pila y se llama a la rutina que procesa la interrupción.

- Después se debe encontrar el proceso que inició la solicitud y que estará a la espera de la interrupción. Normalmente dicho proceso estará en estado de bloqueado, por lo que debe ser despertado, para lo cual se pasa al estado listo, y a continuación se llama al planificador.

- El planificador se encargará de decidir qué proceso pasa a ejecutarse a continuación según el algoritmo de planificación implementado.

- Se inicia el proceso activo seleccionado por el planificador.

Estas acciones se realizan constantemente en el sistema, así pues, se debe intentar optimizar al máximo y reducir el tiempo empleado en esta tarea. Otras veces el cambio de proceso activo se debe a una interrupción temporal, ya que normalmente un proceso activo tiene un quantum de tiempo asignado, de tal forma que tras haber consumido ese tiempo de ejecución, el proceso es desalojado de forma similar a la explicada para el manejo de una interrupción. En realidad, lo que se produce es una interrupción del reloj (que es periódica).

Al proceso explicado se le suele llamar cambio de contexto o context-switch, ya que lo que en realidad ocurre es que se pasa de ejecutar un proceso a ejecutar otro, todo de una forma transparente y rápida.

Paralelismo y concurrencia

Parece claro que a pesar de los avances tecnológicos conseguidos en los últimos años, la tecnología del silicio está llegando a su límite. Si se quieren resolver problemas más complejos y de mayores dimensiones se deben buscar nuevas alternativas tecnológicas. Una de estas alternativas en desarrollo es el paralelismo. Mediante el paralelismo se pretende conseguir la distribución del trabajo entre las diversas CPU disponibles en el sistema de forma que realicen el trabajo simultáneamente, con el objetivo de aumentar considerablemente el rendimiento total.
Para que dos programas se puedan ejecutar en paralelo se deben verificar ciertas condiciones, que se presentan a continuación:

- Sea I_i el conjunto de todas las variables de entrada necesarias para ejecutar el proceso P_i.
- Sea O_i el conjunto de todas las variables de salida generadas por el proceso P_i.
Las condiciones de Bernstein para dos procesos P_1 y P_2 son las siguientes:

1. $I_1 \cap O_2 = \varnothing$
2. $I_2 \cap O_1 = \varnothing$
3. $O_1 \cap O_2 = \varnothing$

Si se cumplen las tres condiciones entonces se dice que P_1 y P_2 pueden ser ejecutados en paralelo y se denota como $P_1 \,||\, P_2$. Esta relación de paralelismo es conmutativa ($P_1 \,||\, P_2 \Longrightarrow P_2 \,||\, P_1$) pero no es transitiva ($P_1 \,||\, P_2$ y $P_2 \,||\, P_3 \not\Longrightarrow P_1 \,||\, P_3$).

Las condiciones de Bernstein aunque definidas para procesos son extrapolables al nivel de instrucciones.

Para conseguir un buen nivel de paralelismo es necesario que el hardware y el software se diseñen conjuntamente.

Existen dos visiones del paralelismo:

- *Paralelismo hardware*: Es el paralelismo definido por la arquitectura de la máquina.

- *Paralelismo software*: Es el paralelismo definido por la estructura del programa. Se manifiesta en las instrucciones que no tienen interdependencias.

El paralelismo se presenta, a su vez, en dos formas:

- *Paralelismo de control*: Se pueden realizar dos o más operaciones simultáneamente. Se presenta en los pipelines y las múltiples unidades funcionales. El programa no necesita preocuparse de este paralelismo, pues se realiza a nivel hardware.

- *Paralelismo de datos*: Una misma operación se puede realizar sobre varios elementos simultáneamente.

En relación al paralelismo hardware, Michael Flynn realizó la siguiente clasificación de arquitecturas de computadores:

SISD (Single Instruction stream over a Single Data stream): Es la arquitectura de las máquinas secuenciales convencionales de un sólo procesador.

SIMD (Single Instruction stream over a Multiple Data stream): Es la arquitectura de las computadoras con hardware para proceso vectorial.

MISD (Multiple Instruction stream over a Single Data stream): Es la arquitectura de las computadoras que poseen un conjunto de procesadores que ejecutan diferentes instrucciones sobre los mismos datos.

MIMD (Multiple Instruction stream over a Multiple Data stream): Es la arquitectura más genérica para los computadores paralelos, ya que es aplicable a cualquier tipo de problema, al contrario que las dos anteriores.

Existen ocasiones en las que se confunde el término concurrencia con el término paralelismo. La concurrencia se refiere a un paralelismo potencial, que puede o no darse. Existen dos formas de concurrencia:

- *Concurrencia implícita*: Es la concurrencia interna al programa, por ejemplo cuando un programa contiene instrucciones independientes que se pueden realizar en paralelo, o existen operaciones de E/S que se pueden realizar en paralelo con otros programas en ejecución. Está relacionada con el paralelismo hardware.

- *Concurrencia explícita*: Es la concurrencia que existe cuando el comportamiento concurrente es especificado por el diseñador del programa. Está relacionada con el paralelismo software.

Es habitual encontrar en la bibliografía el término de programa concurrente en el mismo contexto que el de programa paralelo o distribuido. Existen diferencias sutiles entre estos conceptos:

- *Programa concurrente*: Es aquél que define acciones que pueden realizarse simultáneamente.

- *Programa paralelo*: Es un programa concurrente diseñado para su ejecución en un hardware paralelo.

- *Programa distribuido*: Es un programa paralelo diseñado para su ejecución en una red de procesadores autónomos que no comparten la memoria.

El término concurrente es aplicable a cualquier programa que presente un comportamiento paralelo actual o potencial. En cambio el término paralelo o distribuido es aplicable a aquel programa diseñado para su ejecución en un entorno específico.

Cuando se emplea un solo procesador para la ejecución de programas concurrentes se habla de pseudoparalelismo.

Programación concurrente es el nombre dado a las notaciones y técnicas empleadas para expresar el paralelismo potencial y para resolver los problemas de comunicación y sincronización resultantes. La programación concurrente proporciona una abstracción sobre la que estudiar el paralelismo sin tener en cuenta los detalles de implementación. Esta abstracción ha demostrado ser muy útil en la escritura de programas claros y correctos empleando las facilidades de los lenguajes de programación modernos.

El problema básico en la escritura de un programa concurrente es identificar qué actividades pueden realizarse concurrentemente. Además la programación concurrente es mucho más difícil que la programación secuencial clásica por la dificultad de asegurar que el programa concurrente es correcto.

Características de la concurrencia

Los procesos concurrentes tienen las siguientes características [BEN82]:

Indeterminismo: Las acciones que se especifican en un programa secuencial tienen un orden total, pero en un programa concurrente el orden es parcial, ya que existe una incertidumbre sobre el orden exacto de ocurrencia de ciertos sucesos, esto es, existe un indeterminismo en la ejecución. De esta forma si se ejecuta un programa concurrente varias veces puede producir resultados diferentes partiendo de los mismos datos.

Interacción entre procesos: Los programas concurrentes implican interacción entre los distintos procesos que los componen:

- Los procesos que comparten recursos y compiten por el acceso a los mismos.

- Los procesos que se comunican entre sí para intercambiar datos.

En ambas situaciones se necesita que los procesos sincronicen su ejecución, para evitar conflictos o establecer contacto para el intercambio de datos. La interacción entre procesos se logra mediante variables compartidas o bien mediante el paso de mensajes. Además la interacción puede ser explícita, si aparece en la descripción del programa, o implícita, si aparece durante la ejecución del programa.

Gestión de recursos: Los recursos compartidos necesitan una gestión especial. Un proceso que desee utilizar un recurso compartido debe solicitar dicho recurso, esperar a adquirirlo, utilizarlo y después liberarlo. Si el proceso solicita el recurso pero no puede adquirirlo en ese momento, es suspendido hasta que el recurso está disponible. La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retraso indefinido (espera indefinidamente por un recurso) y de deadlock (bloqueo indefinido o abrazo mortal).

Comunicación: La comunicación entre procesos puede ser síncrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona, cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor los recoja, ya que los deja en un buffer de comunicación temporal.
Problemas de la concurrencia.

Los programas concurrentes a diferencia de los programas secuenciales tienen una serie de problemas muy particulares derivados de las características de la concurrencia:

Violación de la exclusión mutua: En ocasiones ciertas acciones que se realizan en un programa concurrente no proporcionan los resultados deseados. Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una región crítica, es decir, es una parte del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua.

Bloqueo mutuo o Deadlock: Un proceso se encuentra en estado de deadlock si está esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir deadlock:

Los procesos necesitan acceso exclusivo a los recursos.
Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
Los recursos no se pueden obtener de los procesos que están a la espera.
Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena.

Retraso indefinido o starvation: Un proceso se encuentra en starvation si es retrasado indefinidamente esperando un suceso que puede no ocurrir nunca. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso.

Injusticia o unfairness: Se pueden dar situaciones en las que exista cierta injusticia en relación con la evolución de un proceso. Se deben evitar estas situaciones de tal forma que se garantice que un proceso evoluciona y satisface sus necesidades sucesivas en algún momento.

Espera ocupada: En ocasiones cuando un proceso se encuentra a la espera por un suceso, una forma de comprobar si el suceso se ha producido es verificando continuamente si el mismo se ha realizado ya. Esta solución de espera ocupada es muy poco efectiva, porque desperdicia tiempo de procesamiento, y se debe evitar. La solución ideal es suspender el proceso y continuar cuando se haya cumplido la condición de espera.

Comunicación entre Procesos

En muchas ocasiones es necesario que dos procesos se comuniquen entre sí, o bien sincronicen su ejecución de tal forma que uno espere por el otro. La comunicación entre procesos se denota como IPC (InterProcess Communication).

A veces los procesos comparten un espacio de almacenamiento común (un fichero, una zona de memoria, etc.) en la que cada uno puede leer o escribir. Los problemas surgen cuando el acceso a dicha zona no está organizado, sino que es más o menos aleatorio o más bien dependiente de las condiciones de la máquina. A estas situaciones se las denomina condiciones de competencia. Se deben solucionar las condiciones de competencia, esto es, garantizar que sólo accede un proceso al mismo tiempo a la zona compartida. El objetivo es garantizar la exclusión mutua, una forma de garantizar que si un proceso utiliza una variable, archivo compartido o cualquier otro objeto compartido, los demás procesos no pueden utilizarlo.

Otra forma más abstracta de plantear el problema es pensar que normalmente un proceso realiza su tarea independientemente, pero a veces necesita acceder a una serie de recursos compartidos con otros procesos o bien debe llevar a cabo acciones críticas que pueden llevar a conflictos. A esta parte del programa se la llama sección crítica. Se debe evitar, pues, que dos procesos se encuentren en su sección crítica al mismo tiempo.

La condición de la exclusión mutua, no es suficiente para evitar todos los conflictos que se pueden producir entre procesos paralelos que cooperan y emplean datos compartidos. Existen cuatro condiciones que se deben cumplir para obtener una buena solución:

Garantizar la exclusión mutua: Dos procesos no deben encontrarse al mismo tiempo en su sección crítica.

Indeterminación: No se deben hacer hipótesis a cerca de la velocidad o el número de procesadores, durante la ejecución de los procesos.
Ninguno de los procesos que se encuentran fuera de su sección crítica puede bloquear a otros procesos.

Evitar el retraso indefinido: Ningún proceso debe esperar eternamente a entrar en su región crítica.
Existen varias soluciones para garantizar estos principios. Así para garantizar la exclusión mutua tenemos las siguientes opciones:

Desactivar las interrupciones: Consiste en desactivar todas las interrupciones del proceso antes de entrar a la región crítica, con lo que se evita su desalojo de la CPU, y volverlas activar a la salida de la sección crítica. Esta solución no es buena, pues la desactivación de las interrupciones deja a todo el sistema en manos de la voluntad del proceso de usuario, sin que exista garantía de reactivación de las interrupciones.

Emplear variables de cerradura: Consiste en poner una variable compartida, una cerradura, a 1 cuando se va a entrar en la región crítica, y devolverla al valor 0 a la salida. Esta solución en sí misma no es válida porque la propia cerradura es una variable crítica. La cerradura puede estar a 0 y ser comprobada por un proceso A, éste ser suspendido, mientras un proceso B chequea la cerradura, la pone a 1 y puede entrar a su región crítica; a continuación A la pone a 1 también, y tanto A como B se pueden encontrar en su sección crítica al mismo tiempo. Existen muchos intentos de solución a este problema:

- El algoritmo de Dekker

- El algoritmo de Peterson

- La instrucción hardware TSL : Test & Set Lock.

Las soluciones de Dekker, Peterson y TSL son correctas pero emplean espera ocupada. Básicamente lo que realizan es que cuando un proceso desea entrar en su sección crítica comprueba si está permitida la entrada o no. Si no está permitida, el proceso se queda en un bucle de espera hasta que se consigue el permiso de acceso. Esto produce un gran desperdicio de tiempo de CPU, pero pueden aparecer otros problema como la espera indefinida.

Una solución más adecuada es la de bloquear o dormir el proceso (SLEEP) cuando está a la espera de un determinado evento, y despertarlo (WAKEUP) cuando se produce dicho evento. Esta idea es la que emplean las siguientes soluciones:

Semáforos: Esta solución fue propuesta por Dijkstra [DIJ65]. Un semáforo es una variable contador que controla la entrada a la región crítica. Las operaciones P (o WAIT) y V (o SIGNAL) controlan, respectivamente, la entrada y salida de la región crítica. Cuando un proceso desea acceder a su sección crítica realiza un WAIT(var_semaf). Lo que hace esta llamada es, si var_semaf == 0 entonces el proceso se bloquea, sino var_semaf = var_semaf -1. Al finalizar su región crítica, libera el acceso con SIGNAL(var_semaf), que realiza var_semaf = var_semaf + 1. Las acciones se realizan de forma atómica, de tal forma que mientras se realiza la operación P o V ningún otro proceso puede acceder al semáforo. Son el mecanismo más empleado para resolver la exclusión mutua, pero son restrictivos y no totalmente seguros (depende de su implementación), aunque son empleados en ocasiones para implementar otros métodos de sincronización.

Regiones críticas condicionales: Esta solución fue propuesta por Hoare [HOA74] y Brinch Hansen [BRI75] como mejora de los semáforos. Consiste en definir las variables de una región crítica como recursos con un nombre, de esta forma la sección crítica se precede con el nombre del recurso que se necesita y opcionalmente una condición que se debe cumplir para acceder a la misma. Es un buen mecanismo, pero no suele ser soportado por la mayoría de los lenguajes de programación.

Monitores: Esta solución también fue propuesta por Brinch Hansen [BRI75] y Hoare [HOA74]. Un monitor es un conjunto de procedimientos, variables y estructuras de datos que se agrupan en un determinado módulo. Los procesos pueden llamar a los procedimientos del monitor cuando lo deseen para realizar las operaciones sobre los datos compartidos, pero no pueden acceder directamente a las estructuras de datos internas del monitor. Su principal propiedad para conseguir la exclusión mutua es que sólo un proceso puede estar activo en un monitor en cada momento.

Paso de mensajes o transferencia de mensajes: Es sin duda, el modelo más empleado para la comunicación entre procesos. Para la comunicación se emplean las primitivas SEND (para enviar un mensaje) y RECEIVE (para poner al proceso a la espera de un mensaje). Su ventaja es que se puede emplear para la sincronización de procesos en sistemas distribuidos. La comunicación puede ser síncrona o asíncrona. Empleando mensajes se pueden implementar semáforos o monitores, y viceversa.

En los sistemas operativos tradicionales al crear un nuevo proceso (llamada fork en UNIX), lo que en realidad suele hacer el sistema es realizar una copia exacta del padre en el hijo (operación de clonación). Tenemos, entonces, dos procesos iguales.

La ineficiencia aparece cuando el proceso padre es de un tamaño considerable, y el proceso hijo sólo se crea para realizar una pequeña tarea y después finalizar su ejecución, ya que es necesario volcar todo el contenido del proceso padre en el proceso hijo. Además, como padre e hijo son dos procesos totalmente independientes (salvo esa relación padre-hijo), su comunicación o compartición de datos es complicada y poco eficiente. Por último, cuando se produce el cambio de contexto entre padre e hijo, la mayoría de los datos no cambian, con lo que otro tipo de gestión podría optimizar este tipo concreto de cambio de contexto. Los hilos se emplean, en parte, para resolver y optimizar este tipo de situaciones.