# Programación Concurrente y Paralela

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

*pperezm@tec.mx*

07-2022

# ¿Qué es un sistema operativo?

- Un sistema operativo es un programa que administra el hardware de una computadora. También proporciona una base para los programas de aplicación y actúa como intermediario entre el usuario de la computadora y el hardware de la computadora.
- Un aspecto sorprendente de los sistemas operativos es la forma en que varían para realizar estas tareas:
  - Mainframes: Optimizar la utilización del hardware.
  - PC. Ejercutar juegos complejos, aplicaciones comerciales, etc.
  - Móviles: Proporcionar un entorno en el que usuario pueda interactuar fácilmente con la computadora para ejecutar programas.
- Por lo tanto, algunos sistemas operativos están diseñados para ser convenientes, otros para ser eficientes y otros para ser una combinación de los dos.

# ¿Qué hace un sistema operativo?

El sistema operativo controla el hardware y coordina su uso entre los diversos programas de aplicación para los distintos usuarios. El sistema operativo proporciona los medios para el uso adecuado de estos recursos en el funcionamiento del sistema computacional.

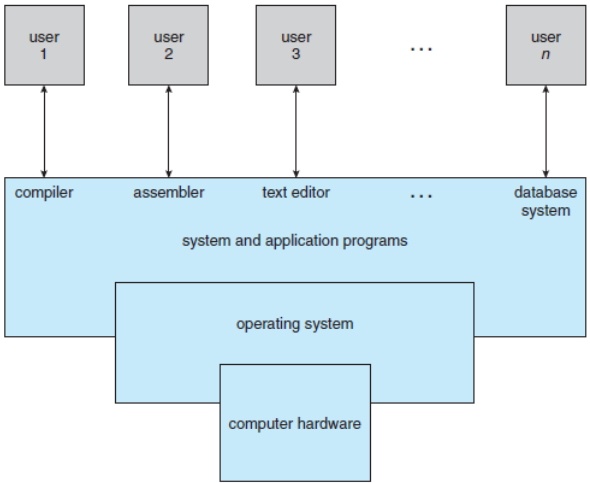



**Figure 1.1** Abstract view of the components of a computer system.

# Operación de un sistema computacional

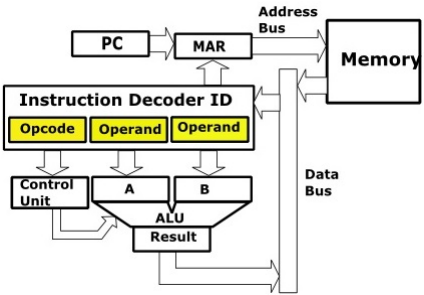
- Cuando una computadora se enciende, inmediatamente se ejecuta un programa inicial. Este programa inicial (bootstrap) tiende a ser simple y, por lo general, se almacenada dentro de hardware es una memoria de solo lectura (ROM, EEPROM), conocida por el termino general de firmware.
  - Primero, se encarga de inicializar todos los aspectos de la computadora.
  - Segundo, debe ubicar el núcleo del sistema operativo (kernel) y cargarlo en memoria.

- Una vez que el kernel se carga y se ejecuta, puede comenzar a brindar servicios al sistema y a sus usuarios. Algunos servicios se proporcionan fuera del kernel, mediante programas del sistema que se cargan en la memoria en el momento del arranque para convertirse en procesos del sistema, o demonios del sistema que se ejecutan todo el tiempo que se ejecuta el kernel.
- La ocurrencia de un evento generalmente se indica mediante una interrupción del hardware o del software. El hardware puede desencadenar una interrupción en cualquier momento enviando una señal al CPU, generalmente a través del bus del sistema. El software puede desencadenar una interrupción al ejecutar una operación especial llamada llamada al sistema.

- Cuando se interrumpe al CPU, detiene lo que está haciendo y transfiere inmediatamente la ejecución a una ubicación fija. La ubicación fija generalmente contiene la dirección de inicio donde se encuentra la rutina de servicio para la interrupción. Se ejecuta la rutina del servicio de interrupción; al finalizar, el CPU reanuda el cálculo interrumpido.

# Ciclo FETH-DECODE-EXECUTE

Un ciclo típico de ejecución de instrucciones, como se ejecuta en un sistema con una arquitectura de von Neumann, primero obtiene una instrucción de la memoria y almacena esa instrucción en el registro de instrucciones. Luego, la instrucción se decodifica y puede hacer que los operandos se obtengan de la memoria y se almacenen en algún registro interno. Una vez que se ha ejecutado la instrucción sobre los operandos, el resultado puede volver a almacenarse en la memoria. Observe que la unidad de memoria solo ve un flujo de direcciones de memoria.




Fuente: `http://bucarotechelp.com/computers/architecture/86080001.asp`

- Hasta hace poco, la mayoría de los sistemas informáticos utilizaban un solo procesador. En un sistema de un solo procesador, hay un CPU principal capaz de ejecutar un conjunto de instrucciones de propósito general, incluidas las instrucciones de los procesos del usuario.
- En los últimos años, los sistemas multiprocesador (también conocidos como sistemas paralelos o sistemas multinúcleo) han comenzado a dominar el panorama de la informática. Dichos sistemas tienen dos o más procesadores en comunicación cercana, compartiendo el bus de la computadora y, a veces, el reloj, la memoria y los dispositivos periféricos.

Los sistemas multiprocesador tienen tres ventajas principales:

1. Mayor rendimiento. Al aumentar la cantidad de procesadores, esperamos hacer más trabajo en menos tiempo.
2. Economía de escala. Los sistemas multiprocesador pueden costar menos que los sistemas equivalentes de un solo procesador, ya que pueden compartir periféricos, almacenamiento masivo y fuentes de alimentación
3. Mayor confiabilidad. Si las funciones se pueden distribuir correctamente entre varios procesadores, entonces la falla de un procesador no detendrá el sistema, solo lo ralentizará.

Los sistemas de procesadores múltiples que se utilizan hoy en día son de dos tipos:

1. Algunos sistemas utilizan multiprocesamiento asimétrico, en el que a cada procesador se le asigna una tarea específica. El procesador principal controla el sistema; los otros procesadores miran al jefe en busca de instrucciones o tienen tareas predefinidas.
2. Los sistemas más comunes utilizan multiprocesamiento simétrico (SMP), en el que cada procesador realiza todas las tareas dentro del sistema operativo. SMP significa que todos los procesadores son iguales; no existe una relación jefe-trabajador entre los procesadores.

Otro tipo de sistema multiprocesador es un sistema en clúster, que reúne varios CPU. Los sistemas agrupados se diferencian de los sistemas multiprocesador en que están compuestos por dos o más sistemas individuales, o nodos, unidos. Estos sistemas se consideran débilmente acoplados. Cada nodo puede ser un sistema de un solo procesador o un sistema multinúcleo.

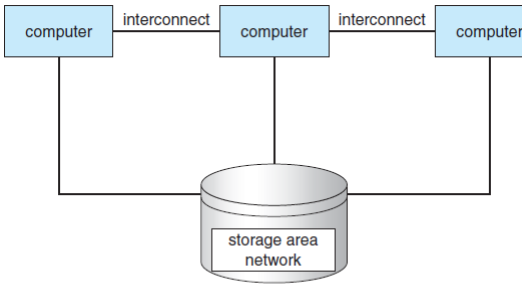



**Figure 1.8** General structure of a clustered system.

- Uno de los aspectos más importantes de los sistemas operativos es la capacidad de multiprogramación. En general, un solo programa no puede mantener ocupados ni al CPU ni a los dispositivos de E/S en todo momento. Los usuarios individuales suelen tener varios programas en ejecución. La multiprogramación aumenta la utilización de la CPU al organizar los trabajos (código y datos) para que la CPU siempre tenga uno para ejecutar.

La idea es la siguiente:

- El sistema operativo mantiene varios trabajos en la memoria simultáneamente. Dado que, en general, la memoria principal es demasiado pequeña para acomodar todos los trabajos, los trabajos se guardan inicialmente en el disco el "pool" de trabajos.

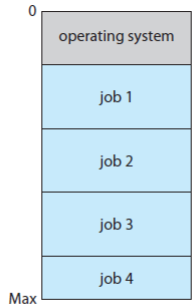



**Figure 1.9** Memory layout for a multiprogramming system.

- El conjunto de trabajos en la memoria puede ser un subconjunto de los trabajos que se mantienen en el "pool" de trabajos. El sistema operativo selecciona y comienza a ejecutar uno de los trabajos en la memoria. Eventualmente, es posible que el trabajo tenga que esperar a que se complete alguna tarea, como una operación de E/S. En un sistema no multiprogramado, el CPU permanecería inactiva. En un sistema multiprogramado, el sistema operativo simplemente cambia y ejecuta otro trabajo. Cuando ese trabajo necesita esperar, la CPU cambia a otro trabajo y así sucesivamente. Finalmente, el primer trabajo termina de esperar y recupera la CPU. Mientras sea necesario ejecutar al menos un trabajo, la CPU nunca estará inactiva.

- Un programa no hace nada a menos que sus instrucciones sean ejecutadas por una CPU. Un programa en ejecución es un proceso.
- El proceso necesita ciertos recursos, incluido el tiempo de CPU, la memoria, los archivos y los dispositivos de E/S, para realizar su tarea. Estos recursos se le dan al proceso cuando se crea o se le asignan mientras se está ejecutando.
- Un proceso es la unidad de trabajo en un sistema. Un sistema consta de una colección de procesos, algunos de los cuales son procesos del sistema operativo (los que ejecutan el código del sistema) y el resto son procesos del usuario (los que ejecutan el código del usuario).

El almacenamiento en caché es un principio importante de los sistemas de cómputo. Así es como funciona. La información normalmente se guarda en algún sistema de almacenamiento (como la memoria principal). A medida que se utiliza, se copia a un sistema de almacenamiento más rápido, el caché, de forma temporal. Cuando necesitamos una información en particular, primero verificamos si está en la caché. Si es así, usamos la información directamente del caché. Si no es así, usamos la información de la fuente, poniendo una copia en la caché bajo el supuesto de que la necesitaremos nuevamente pronto.

Debido a que las memorias caché tienen un tamaño limitado, la administración de la memoria caché es un problema de diseño importante. La selección cuidadosa del tamaño de la caché y de una política de reemplazo puede resultar en un rendimiento mucho mayor.

| Level | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Name | registers | cache | main memory | solid state disk | magnetic disk |
| Typical size | < 1 KB | < 16MB | < 64GB | < 1 TB | < 10 TB |
| Implementation technology | custom memory with multiple ports CMOS | on-chip or off-chip CMOS SRAM | CMOS SRAM | flash memory | magnetic disk |
| Access time (ns) | 0.25 - 0.5 | 0.5 - 25 | 80 - 250 | 25,000 - 50,000 | 5,000,000 |
| Bandwidth (MB/sec) | 20,000 - 100,000 | 5,000 - 10,000 | 1,000 - 5,000 | 500 | 20 - 150 |
| Managed by | compiler | hardware | operating system | operating system | operating system |
| Backed by | cache | main memory | disk | disk | disk or tape |

**Figure 1.11** Performance of various levels of storage.

En una estructura de almacenamiento jerárquica, los mismos datos pueden aparecer en diferentes niveles del sistema de almacenamiento. Por ejemplo, suponga que un entero A que debe incrementarse en 1 se encuentra en el archivo B y el archivo B reside en un disco magnético. La operación de incremento procede emitiendo primero una operación de E/S para copiar el bloque de disco en el que A reside en la memoria principal. A esta operación le sigue la copia de A en la caché y en un registro interno. Una vez que se produce el incremento en el registro interno, el valor de A difiere en los distintos sistemas de almacenamiento. El valor de A se vuelve el mismo solo después de que el nuevo valor de A se escribe desde el registro interno de regreso al disco magnético.

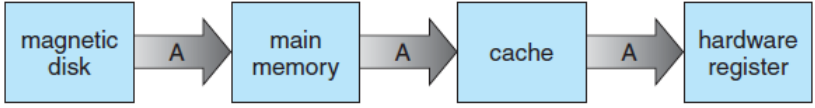



**Figure 1.12** Migration of integer A from disk to register.

- En un entorno donde solo se ejecuta un proceso a la vez, esta disposición no plantea dificultades, ya que un acceso al entero A siempre será a la copia en el nivel más alto de la jerarquía.
- Sin embargo, en un entorno multitarea, donde el CPU se alterna entre varios procesos, se debe tener mucho cuidado para garantizar que, si varios procesos desean acceder a A, cada uno de estos procesos obtendrá el valor actualizado más recientemente de A .

La situación se complica en un entorno multiprocesador donde, además de mantener registros internos, cada una de los CPU también contiene una caché local. En tal entorno, una copia de A puede existir simultáneamente en varias cachés. Dado que los distintos CPU pueden ejecutarse todas en paralelo, debemos asegurarnos de que una actualización del valor de A en una caché se refleje inmediatamente en todas las demás cachés donde A reside. Esta situación se denomina **coherencia de caché** y suele ser un problema de hardware (que se maneja por debajo del nivel del sistema operativo).

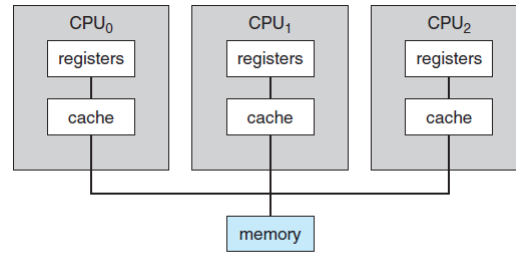



**Figure 1.6** Symmetric multiprocessing architecture.

# Actividad colaborativa

En grupos de 2 integrantes,

- Investiguen cuál el sistema operativo más utilizado en servidores.
- Seleccionen uno, investiguen sus características y qué sistemas se utilizan.
- Generen un reporte con los resultados, suban el reporte a Canvas.

- Las primeras computadoras permitían que solo se ejecutara un programa a la vez. Este programa tenía el control completo del sistema y tenía acceso a todos los recursos del sistema. Por el contrario, los sistemas informáticos contemporáneos permiten que se carguen múltiples programas en la memoria y se ejecuten al mismo tiempo. Esta evolución requirió un control más firme y una mayor compartimentación de los distintos programas; y estas necesidades dieron lugar a la noción de **proceso**, que es un programa en ejecución. Un proceso es la unidad de trabajo en un sistema moderno de tiempo compartido.

- Un sistema informático actual consta de una colección de procesos: procesos del sistema operativo que ejecutan el código del sistema y procesos del usuario que ejecutan el código del usuario. Potencialmente, todos estos procesos pueden ejecutarse al mismo tiempo, con el CPU (o CPUs) multiplexados entre ellos. Al cambiar el CPU entre procesos, el sistema operativo puede hacer que la computadora sea más productiva.

Mencionamos antes que un proceso es un programa en ejecución. Un proceso es más que el código del programa, que a veces se conoce como **sección de texto**. También incluye la actividad actual, representada por el valor del **contador del programa** y el contenido de los registros del procesador. Un proceso generalmente también incluye la **pila de procesos**, que contiene datos temporales (como parámetros de función, direcciones de retorno y variables locales), y **una sección de datos**, que contiene variables globales. Un proceso también puede incluir un **heap**, que es memoria que es asignados dinámicamente durante el tiempo de ejecución del proceso.

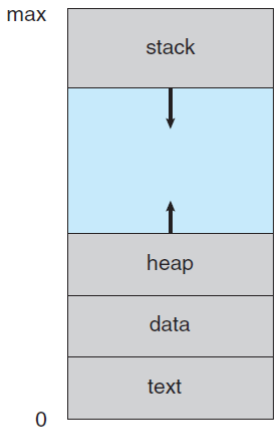



**Figure 3.1** Process in memory.

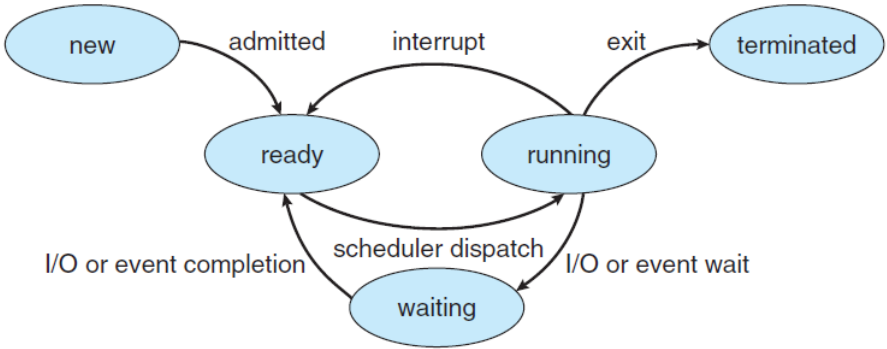



**Figure 3.2** Diagram of process state.

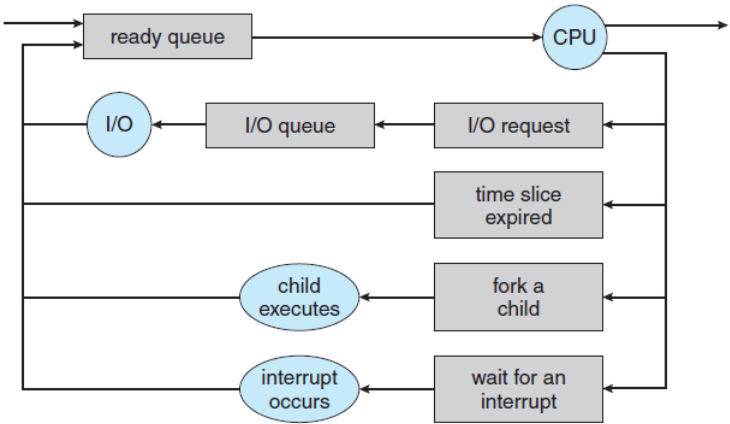



**Figure 3.6** Queueing-diagram representation of process scheduling.

- Un proceso migra entre las distintas colas de programación a lo largo de su vida. El sistema operativo debe seleccionar, para fines de programación, procesos de estas colas de alguna manera. El proceso de selección lo lleva a cabo el planificador correspondiente. El programador a largo plazo, o programador de trabajos, selecciona procesos de este grupo y los carga en la memoria para su ejecución. El programador a corto plazo, o programador del CPU, selecciona entre los procesos que están listos para ejecutarse y asigna el CPU a uno de ellos.

- La principal distinción entre estos dos programadores radica en la frecuencia de ejecución. El planificador a corto plazo debe seleccionar un nuevo proceso para el CPU con frecuencia. Un proceso puede ejecutarse durante unos pocos milisegundos antes de esperar una solicitud de E/S. A menudo, el programador a corto plazo se ejecuta al menos una vez cada 100 milisegundos. Debido al poco tiempo entre ejecuciones, el planificador a corto plazo debe ser rápido. Si toma 10 milisegundos decidir ejecutar un proceso durante 100 milisegundos, entonces 10 / (100 + 10) = 9 por ciento de la CPU se está usando (desperdiciado) simplemente para programar el trabajo.

# Planificador a largo plazo

- El planificador a largo plazo se ejecuta con mucha menos frecuencia; minutos pueden separar la creación de un nuevo proceso y el siguiente. El planificador a largo plazo controla el grado de multiprogramación (el número de procesos en la memoria). Si el grado de multiprogramación es estable, entonces la tasa promedio de creación de procesos debe ser igual a la tasa de salida promedio de los procesos que abandonan el sistema.

# Planificador a mediano plazo

- Algunos sistemas operativos, como los sistemas de tiempo compartido, pueden introducir un nivel intermedio adicional de programación. La idea clave detrás de un programador a mediano plazo es que a veces puede ser ventajoso eliminar un proceso de la memoria (y de la contención activa para el CPU) y así reducir el grado de multiprogramación. Posteriormente, el proceso puede reintroducirse en la memoria y su ejecución puede continuar donde se detuvo. Este esquema se llama intercambio. El proceso es reemplazado y luego reemplazado por el programador de mediano plazo. El intercambio puede ser necesario para mejorar la combinación de procesos o porque un cambio en los requisitos de memoria ha comprometido en exceso la memoria disponible, lo que requiere que se libere memoria.

- Un proceso es un programa que realiza un solo hilo de ejecución. Por ejemplo, cuando un proceso está ejecutando un programa de procesador de texto, se está ejecutando un solo hilo de instrucciones. Este único hilo de control permite que el proceso realice solo una tarea a la vez. El usuario no puede escribir caracteres simultáneamente y ejecutar el corrector ortográfico dentro del mismo proceso, por ejemplo. La mayoría de los sistemas operativos modernos han ampliado el concepto de proceso para permitir que un proceso tenga múltiples subprocesos de ejecución y, por lo tanto, realice más de una tarea a la vez. Esta característica es especialmente beneficiosa en sistemas multinúcleo, donde varios subprocesos pueden ejecutarse en paralelo. En un sistema que admite subprocesos, la PCB se amplía para incluir información para cada subproceso. También se necesitan otros cambios en todo el sistema para admitir subprocesos.

- Un hilo es una unidad básica de utilización del CPU. Comprende un ID de hilo, un contador de programa, un conjunto de registros y una pila. Comparte con otros hilos que pertenecen al mismo proceso su sección de código, sección de datos y otros recursos del sistema operativo, como archivos abiertos y señales. El proceso tradicional (o pesado) tiene un solo hilo de control. Si un proceso tiene varios hilos de control, puede realizar más de una tarea a la vez.

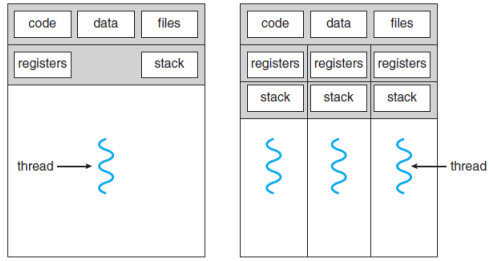



**Figure 4.1** Single-threaded and multithreaded processes.

- La mayoría de las aplicaciones de software que se ejecutan en las computadoras modernas son multihilos. Por lo general, una aplicación se implementa como un proceso separado con varios hilos de control. Un navegador web puede tener un hilo que muestra imágenes o texto mientras que otro hilo recupera datos de la red, por ejemplo. Un procesador de texto puede tener un hilo para mostrar gráficos, otro hilo para responder a las pulsaciones de teclas del usuario y un tercer hilo para realizar la revisión ortográfica y gramatical en segundo plano. Las aplicaciones también se pueden diseñar para aprovechar las capacidades de procesamiento en sistemas multinúcleo. Estas aplicaciones pueden realizar varias tareas de uso intensivo del CPU en paralelo en los múltiples núcleos informáticos.

# Beneficios

Los beneficios de la programación multihilo se pueden dividir en cuatro categorías principales:

- **Respuesta.** Una aplicación multihilos interactiva puede permitir que un programa continúe ejecutándose incluso si parte de él está bloqueado o está realizando una operación prolongada, aumentando así la capacidad de respuesta del usuario. Esta cualidad es especialmente útil en el diseño de interfaces de usuario.
- **Compartir recursos.** Los procesos solo pueden compartir recursos a través de técnicas como la memoria compartida y el paso de mensajes. Estas técnicas deben ser organizadas explícitamente por el programador. Sin embargo, los hilos comparten la memoria y los recursos del proceso al que pertenecen de forma predeterminada. El beneficio de compartir código y datos es que permite que una aplicación tenga varios hilos de actividad diferentes dentro del mismo espacio de direcciones.

- **Economía.** Asignar memoria y recursos para la creación de procesos es costoso. Debido a que los hilos comparten los recursos del proceso al que pertenecen, es más económico crear hilos y cambiar de contexto.
- **Escalabilidad.** Los beneficios del subproceso múltiple pueden ser aún mayores en una arquitectura de multiprocesador, donde los subprocesos pueden ejecutarse en paralelo en diferentes núcleos de procesamiento. Un proceso de un solo subproceso puede ejecutarse en un solo procesador, independientemente de cuántos estén disponibles.

# ¿Cómo programamos hilos en C/C++? (Ver código)

- La programación multihilo proporciona un mecanismo para un uso más eficiente de estos múltiples núcleos informáticos y una mejor concurrencia. Considere una aplicación con cuatro hilos. En un sistema con un solo núcleo informático, la concurrencia simplemente significa que la ejecución de los hilos se intercalará con el tiempo (Figura 4.3), porque el núcleo de procesamiento es capaz de ejecutar solo un hilo a la vez. Sin embargo, en un sistema con múltiples núcleos, la concurrencia significa que los hilos pueden ejecutarse en paralelo, porque el sistema puede asignar un hilo separado a cada núcleo (Figura 4.4).

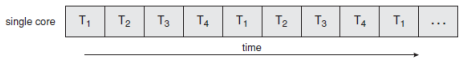



**Figure 4.3** Concurrent execution on a single-core system.

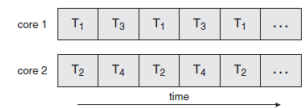



**Figure 4.4** Parallel execution on a multicore system.

- En este punto, es importante realizar la distinción entre paralelismo y concurrencia. Un sistema es paralelo si puede realizar más de una tarea simultáneamente. Por el contrario, un sistema concurrente admite más de una tarea al permitir que todas las tareas progresen. Por tanto, es posible tener concurrencia sin paralelismo. Antes de la llegada de las arquitecturas SMP y multinúcleo, la mayoría de los sistemas informáticos tenían un solo procesador. Los programadores de CPU se diseñaron para proporcionar la ilusión de paralelismo al cambiar rápidamente entre procesos en el sistema, lo que permite que cada proceso progrese. Dichos procesos se estaban ejecutando al mismo tiempo, pero no en paralelo.

La tendencia hacia los sistemas multinúcleo sigue ejerciendo presión sobre los diseñadores de sistemas y los programadores de aplicaciones para hacer un mejor uso de los múltiples núcleos informáticos. En general, cinco áreas presentan desafíos en la programación de sistemas multinúcleo:

- **Identificación de tareas.** Esto implica examinar aplicaciones para encontrar áreas que se puedan dividir en tareas simultáneas separadas. Idealmente, las tareas son independientes entre sí y, por lo tanto, pueden ejecutarse en paralelo en núcleos individuales.

- **Equilibrio.** Al identificar las tareas que pueden ejecutarse en paralelo, los programadores también deben asegurarse de que las tareas realicen un trabajo igual de igual valor. En algunos casos, una determinada tarea puede no aportar tanto valor al proceso general como otras tareas. Es posible que el uso de un núcleo de ejecución independiente para ejecutar esa tarea no valga la pena.
- **División de datos.** Así como las aplicaciones se dividen en tareas separadas, los datos a los que acceden y manipulan las tareas deben dividirse para que se ejecuten en núcleos separados.
- **Dependencia de datos.** Los datos a los que acceden las tareas deben examinarse en busca de dependencias entre dos o más tareas. Cuando una tarea depende de los datos de otra, los programadores deben asegurarse de que la ejecución de las tareas esté sincronizada para adaptarse a la dependencia de los datos.

- **Dependencia de datos.** Los datos a los que acceden las tareas deben examinarse en busca de dependencias entre dos o más tareas. Cuando una tarea depende de los datos de otra, los programadores deben asegurarse de que la ejecución de las tareas esté sincronizada para adaptarse a la dependencia de los datos.
- **Prueba y depuración.** Cuando un programa se ejecuta en paralelo en varios núcleos, son posibles muchas rutas de ejecución diferentes. Probar y depurar dichos programas simultáneos es intrínsecamente más difícil que probar y depurar aplicaciones de un solo hilo.

- El **paralelismo de datos** se centra en distribuir subconjuntos de los mismos datos en varios núcleos informáticos y realizar la misma operación en cada núcleo. Considere, por ejemplo, sumar el contenido de un arreglo de tamaño N. En un sistema de un solo núcleo, un hilo simplemente sumaría los elementos [0]. . . [N - 1]. En un sistema de doble núcleo, sin embargo, el hilo A, que se ejecuta en el núcleo 0, podría sumar los elementos [0]. . . [N / 2 - 1] mientras que el hilo B, que se ejecuta en el núcleo 1, podría sumar los elementos [N / 2]. . . [N - 1]. Los dos hilos se ejecutarían en paralelo en núcleos informáticos separados.

- El **paralelismo de tareas** implica distribuir no datos sino tareas (hilos) a través de múltiples núcleos informáticos. Cada hilo está realizando una operación única. Diferentes hilos pueden estar operando con los mismos datos o pueden estar operando con diferentes datos. Considere nuevamente nuestro ejemplo anterior. En contraste con esa situación, un ejemplo de paralelismo de tareas podría involucrar dos hilos, cada uno de los cuales realiza una operación estadística única en el arreglo de elementos. Los hilos nuevamente operan en paralelo en núcleos de computación separados, pero cada uno está realizando una operación única.

Entonces, fundamentalmente, el paralelismo de datos implica la distribución de datos en múltiples núcleos y el paralelismo de tareas en la distribución de tareas en múltiples núcleos. En la práctica, sin embargo, pocas aplicaciones siguen estrictamente el paralelismo de datos o tareas. En la mayoría de los casos, las aplicaciones utilizan un híbrido de estas dos estrategias.

- Una biblioteca de hilos proporciona al programador una API para crear y administrar hilos. Hay dos formas principales de implementar una biblioteca de hilos. El primer enfoque es proporcionar una biblioteca completamente en el espacio del usuario sin soporte del kernel. Todo el código y las estructuras de datos de la biblioteca existen en el espacio del usuario. Esto significa que la invocación de una función en la biblioteca da como resultado una llamada de función local en el espacio de usuario y no una llamada al sistema.

- El segundo enfoque es implementar una biblioteca de nivel de kernel compatible directamente con el sistema operativo. En este caso, el código y las estructuras de datos de la biblioteca existen en el espacio del kernel. Invocar una función en la API para la biblioteca normalmente da como resultado una llamada del sistema al kernel. Actualmente se utilizan tres bibliotecas de hilos principales: POSIX Pthreads, Windows y Java.
  - Pthreads, la extensión de hilos del estándar POSIX, se puede proporcionar como una biblioteca a nivel de usuario o a nivel de kernel.
  - La biblioteca de hilos de Windows es una biblioteca de nivel de kernel disponible en los sistemas Windows.
  - La API de hilos de Java permite que se creen y administren subprocesos directamente en programas de Java.

Un proceso cooperativo es aquel que puede afectar o verse afectado por otros procesos que se ejecutan en el sistema. Los procesos que cooperan pueden compartir directamente un espacio de direcciones lógicas (es decir, tanto código como datos) o se les permite compartir datos solo a través de archivos o mensajes. Sin embargo, el acceso simultáneo a los datos compartidos puede dar como resultado una inconsistencia en los datos.

Ya hemos visto que los procesos se pueden ejecutar al mismo tiempo o en paralelo. Vimos el papel de la programación de procesos y se describió cómo el programador del CPU cambia rápidamente entre procesos para proporcionar una ejecución concurrente. Esto significa que un proceso solo puede completar la ejecución parcialmente antes de que se programe otro proceso. De hecho, un proceso puede interrumpirse en cualquier punto de su flujo de instrucciones y el núcleo de procesamiento puede asignarse para ejecutar instrucciones de otro proceso.

Ver código de sección crítica.

Como pudieron observar, llegamos a este estado incorrecto porque permitimos que ambos hilos manipulen el contador de variables al mismo tiempo. Una situación como esta, en la que varios procesos acceden y manipulan los mismos datos al mismo tiempo y el resultado de la ejecución depende del orden particular en el que tiene lugar el acceso, se denomina **condición de carrera**. Para protegernos contra la condición de carrera anterior, debemos asegurarnos de que solo un proceso a la vez pueda manipular el contador de variables. Para hacer tal garantía, requerimos que los procesos estén sincronizados de alguna manera.

Si bien, situaciones como la que acabamos de describir ocurren con frecuencia en los sistemas operativos cuando diferentes partes del sistema manipulan los recursos. Ahora, la creciente importancia de los sistemas multinúcleo ha traído un mayor énfasis en el desarrollo de aplicaciones multiproceso. En tales aplicaciones, varios subprocesos, que posiblemente comparten datos, se ejecutan en paralelo en diferentes núcleos de procesamiento. Claramente, queremos que los cambios que resulten de tales actividades no interfieran entre sí. Debido a la importancia de este tema, en este tema hablaremos de la sincronización y coordinación de procesos.

# Definición del problema

Considera un sistema que consta de $N$ procesos $P_0$, $P_1$, ..., $P_{n-1}$. Cada proceso tiene un segmento de código, llamada sección crítica, en que el proceso puede cambiar variables comunes, actualizar una tabla, escribir un archivo, etc. La característica importante del sistema es que, cuando un proceso está ejecutando la sección crítica, ningún otro proceso puede ejecutarla. El problema de la sección crítica es diseñar un protocolo que los procesos puedan utilizar para cooperar.

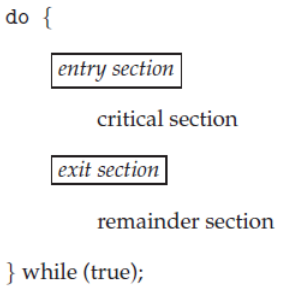



**Figure 5.1** General structure of a typical process $P_i$.

Una solución al problema de la sección crítica debe satisfacer los siguientes tres requisitos:

1. **Exclusión mutua.** Si el proceso $P_i$ está ejecutando su sección crítica, entonces ningún otro proceso puede ejecutar la suya.
2. **Progreso.** Si ningún proceso está ejecutando su sección crítica y algunos procesos desean ingresar a sus secciones críticas pueden participar para decidir cuál entrará a continuación en su sección crítica, y esta selección no se puede posponer indefinidamente.
3. **Espera limitada.** Existe un límite en la cantidad de veces que se permite que otros procesos ingresen a sus secciones críticas después de que un proceso haya realizado una solicitud para ingresar a su sección crítica y antes de que se otorgue dicha solicitud.

Un semáforo $S$ es una variable entera a la que, además de la inicialización, se accede solo a través de dos operaciones atómicas estándar: *wait()* y *signal()*. La operación *wait()* se denominó originalmente $P$ (del holandés proberen, "probar"); *signal()* originalmente se llamaba **V** (de verhogen, ïncrementar").

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}

     signal(S) {
        S++;
     }
```

Los sistemas operativos a menudo distinguen entre contadores y semáforos binarios.

- El valor de un semáforo binario sólo puede oscilar entre 0 y 1. Se utilizan para proporcionar exclusión mutua.
- Los semáforos contadores se pueden utilizar para controlar el acceso a un recurso determinado que consta de un número finito de instancias. El semáforo se inicializa con la cantidad de recursos disponibles. Cada proceso que desea utilizar un recurso realiza una operación de *wait()* en el semáforo (disminuyendo así el contador). Cuando un proceso libera un recurso, realiza una operación *signal()* (incrementando el contador). Cuando el recuento del semáforo llega a 0, se están utilizando todos los recursos. Después de eso, los procesos que deseen utilizar un recurso se bloquearán hasta que el recuento sea superior a 0.

La implementación de un semáforo con una cola de espera puede resultar en una situación en la que dos o más procesos están esperando indefinidamente un evento que solo puede ser causado por uno de los procesos en espera. El evento en cuestión es la ejecución de una operación *signal()*. Cuando se alcanza tal estado, se dice que estos procesos están bloqueados.

| $P_0$ | $P_1$ |
|---|---|
| `wait(S);` | `wait(Q);` |
| `wait(Q);` | `wait(S);` |
| . | . |
| . | . |
| . | . |
| `signal(S);` | `signal(Q);` |
| `signal(Q);` | `signal(S);` |

Otro problema relacionado con los interbloqueos es el bloqueo indefinido o la inanición, una situación en la que los procesos esperan indefinidamente dentro del semáforo. El bloqueo indefinido puede ocurrir si eliminamos procesos de la lista asociada con un semáforo en orden LIFO (último en entrar, primero en salir).

# Problema Productor-Consumidor

En computación, el problema del productor-consumidor es un ejemplo clásico de problema de sincronización de multiprocesos. El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

# Ver código de Productor-Consumidor

Supón que una base de datos se va a compartir entre varios procesos concurrentes. Algunos de estos procesos pueden querer solo leer la base de datos, mientras que otros pueden querer actualizar (es decir, leer y escribir) la base de datos. Distinguimos entre estos dos tipos de procesos refiriéndonos a los primeros como lectores y a los segundos como escritores. Obviamente, si dos lectores acceden a los datos compartidos simultáneamente, no se producirán efectos adversos. Sin embargo, si un escritor y algún otro proceso (ya sea un lector o un escritor) acceden a la base de datos simultáneamente, puede producirse el caos.

# Ver código de Lectores y Escritores

# Problema de los filósofos comedores

Considera a cinco filósofos que se pasan la vida pensando y comiendo. Los filósofos comparten una mesa circular rodeada de cinco sillas, cada una de las cuales pertenece a un filósofo. En el centro de la mesa hay un cuenco de arroz y la mesa se coloca con cinco palillos individuales. Cuando un filósofo piensa, no interactúa con sus colegas. De vez en cuando, un filósofo tiene hambre y trata de recoger los dos palillos que están más cerca de él (los palillos que están entre él y sus vecinos izquierdo y derecho). Un filósofo puede tomar solo un palillo a la vez. Evidentemente, no puede coger un palillo que ya está en la mano de un vecino. Cuando un filósofo hambriento tiene sus dos palillos al mismo tiempo, come sin soltar los palillos. Cuando termina de comer, deja ambos palillos y comienza a pensar de nuevo.

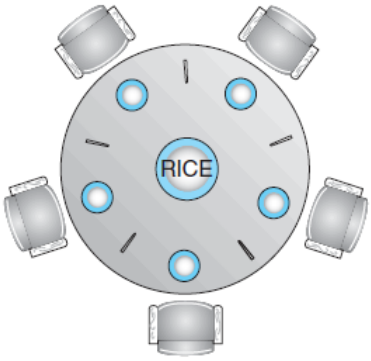



**Figure 5.13** The situation of the dining philosopher

Ver código de Filósofos comedores.

# Actividad colaborativa

En grupos de 2 integrantes,

- Implementa, usando threads, el problema de los lectores-escritores.
- Sube tu código a Canvas.