Programación paralela

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

pperezm@tec.mx

06-2024

1. Introducción
   - Conceptos básicos
   - ¿Cómo paralelizar?
   - Retos de la programación paralela
   - Tipos de paralelismos
   - ¿Qué es un hilo?
   - Coherencia de caché
   - ¿Cómo medimos la eficiencia?

2 CUDA
Cómputo heterogéneo
Hardware
Ejecución de un programa
Asignación de tareas (kernels)

- Se dice que un sistema es **concurrente** si puede soportar **dos o más acciones en curso al mismo tiempo**. Se dice que un sistema es **paralelo** si puede soportar **dos o más acciones ejecutándose simultáneamente**.
- La **concurrencia** consiste en **tratar con muchas cosas a la vez**. El **paralelismo** consiste en **hacer muchas cosas a la vez**.

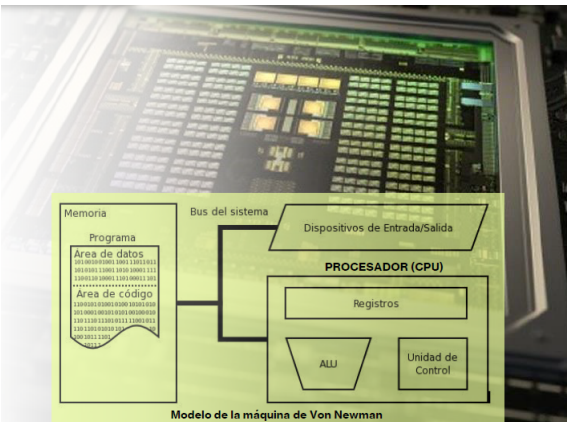
- Antes de la llegada de las arquitecturas SMP y multinúcleo, la mayoría de los sistemas informáticos tenían un solo procesador. Los programadores de CPU se diseñaron para proporcionar la ilusión de paralelismo al cambiar rápidamente entre procesos en el sistema (multiprogramación), lo que permite que cada proceso progrese. Este tipo de **sistemas son conocidos como sistemas concurrentes**.
- Por el contrario, un sistema es **paralelo**, al tener más de un núcleo de trabajo, si puede realizar más de una **tarea simultáneamente**.
- Por tanto, **es posible tener concurrencia sin paralelismo**.

¿Cómo paralelizar?

- Arquitectura del hardware.
- Sistema operativo.
- Lenguaje de programación.
- Diseño del algoritmo.

Aspectos que determinan la construcción de **hardware paralelo**:

- **Cantidad** de procesadores.
- **Localización** de los procesadores.
  - Núcleos, supercomputadoras, grids.
- **Red de interconexión** entre los procesadores.
  - Topologías.
- Forma de compartir la **memoria**.
  - Memoria compartida vs. distribuida.
- Mecanismos para el **control de flujos**.
  - Taxonomía de Flyn.

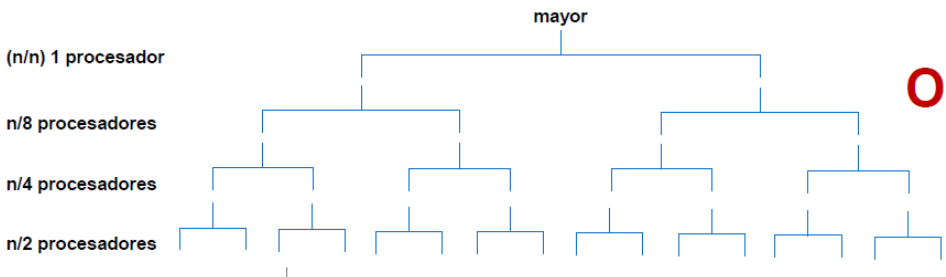


Encontrar el mayor de todos los elementos de un arreglo.

```
mayor = S[1];
for (i=2; i<=n; i++)
  if (S[i] > mayor)  mayor = S[i];
```

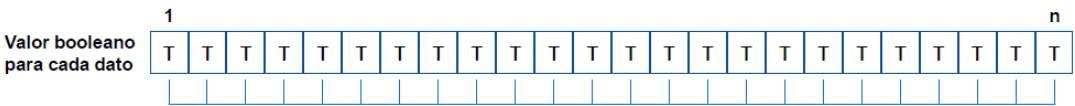
- ¿Cómo podemos **paralelizar** el proceso para hacerlo más eficiente?

Estrategia de torneo:
Cada procesador revisa 2 datos y selecciona el mayor



O(log n)

Estrategia de torneo simultáneo:
Cada procesador revisa 2 datos y selecciona el mayor



O(1)

- Encargado de **controlar el uso de los recursos** del hardware.
- **Interfase** con las aplicaciones de software.
- Administra los **procesos** que se ejecutan.
  - Evita los conflictos: interbloqueos (deadlock), exclusiones mutuas, etc.

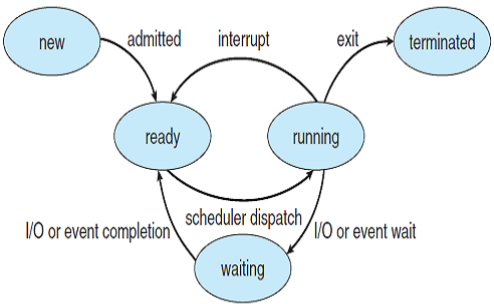
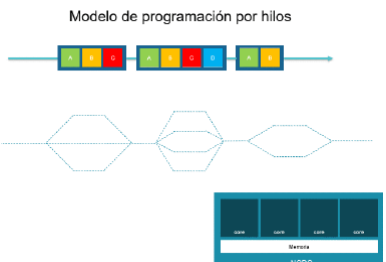


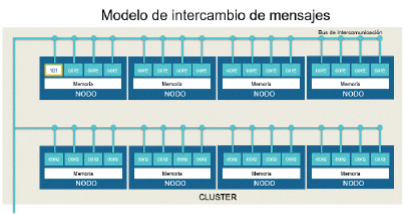
Figure 3.2 Diagram of process state.

- Enfoque sobre el **algoritmo de solución**.
  - El algoritmo se plantea considerando los procesos a paralelizar.
- Enfoque sobre el **lenguaje de programación**.
  - El lenguaje provee herramientas para paralelizar procesos.
    - Como extensión del lenguaje.
    - Como lenguaje dedicado.
- Enfoque sobre el **traductor**.
  - El proceso de traducción optimiza el código considerando la arquitectura paralela.

- Basada en **hilos** (threads) con estados compartidos.
  - Memoria compartida, que tiene que ser controlada para tener exclusión mutua (por medio de semáforos, monitores, etc.), cuidando problemas como interbloqueos (deadlocks).



- Basada en **paso de mensajes**.
  - Los procesos de comunican asincrónicamente y no hay memoria compartida.



La tendencia hacia la programación paralela 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.
- **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** 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.

- Un proceso es un programa que realiza un solo hilo de ejecución. Este único hilo de control permite que el proceso realice solo una tarea a la vez.
- Un hilo es una unidad básica de utilización del CPU. Comprende un identificador 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.

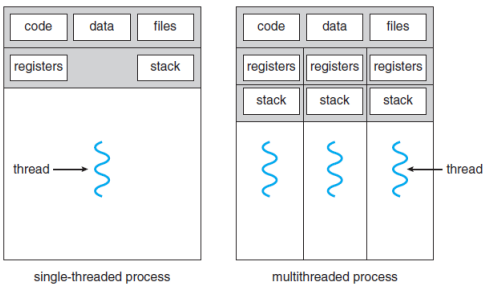


Figure 4.1 Single-threaded and multithreaded processes.

- 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.

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.

- 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).

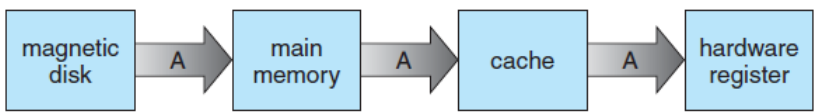


Figure 1.12 Migration of integer A from disk to register.

Introudcción a la Programando Multihilos en C/C++.

¿Cómo medimos la eficiencia?

- A nivel de **análisis asintótico**, el algoritmo paralelo tiene **la misma complejidad**.
- Para determinar la eficiencia de un algoritmo paralelo, empleamos la medida estándar conocida como **"Speed Up"**.
- El **Speed Up** de una aplicación que se ejecuta en una máquina paralela de **"p"** procesadores está definido de la siguiente manera:

$$S_p = \frac{T_s}{T_p}$$

  Dónde:
  T_s = Tiempo de ejecución de la implementación secuencial.
  T_p = Tiempo de ejecución de la implementación paralela con "p" procesadores.
- La eficiencia es un término relacionado con el **Speed Up** y se define de la siguiente manera:

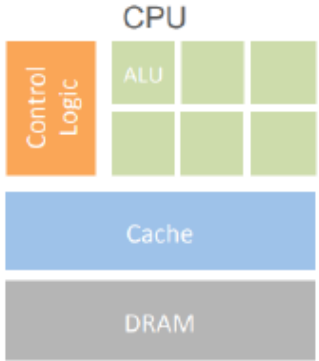
$$E = \frac{S_p}{p}$$

Revisar actividad en Canvas.

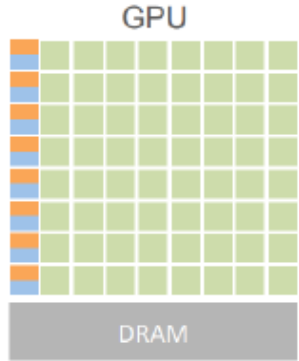
Paralelizando tareas con hilos en C/C++.

Revisar actividad en Canvas.

- CPU:
  - Optimizado para la ejecución rápida de un hijo.
  - Los núcleos del CPU está diseñados para ejecutar 1 o 2 hilos simultáneamente.
  - Grandes caches permiten ocultar los tiempos de acceso a la DRAM.
  - Núcleos optimizados para acceso de caché de baja latencia.
  - Lógica de control compleja con el fin de emplear la ejecución fuera de orden.
- GPU
  - Optimizado para tener alto rendimiento en multihilo.
  - Núcleos diseñados para ejecutar muchos hilos paralelos al mismo tiempo.
  - Núcleos optimizados para datos en paralelo.
  - Los chips utilizan multihilointensivo para tolerar los tiempos de acceso a la DRAM.



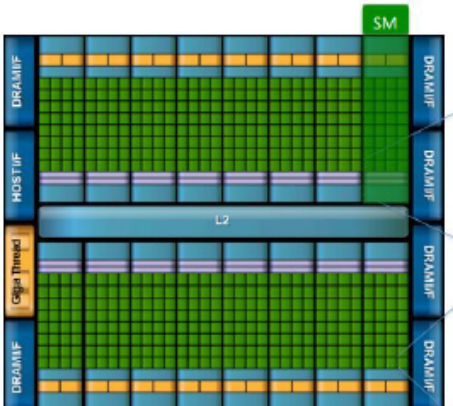
Less than 20 cores
1-2 threads per core
Latency is hidden by large cache



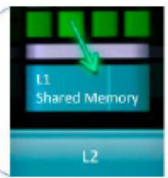
512 cores
10s to 100s of threads per core
Latency is hidden by fast context switching

GPUs don't run without CPUs

NVIDIA FERMI

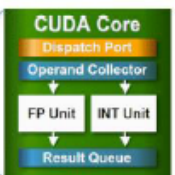


16 Stream Multiprocessors (SM)
512 CUDA cores (32/SM)
IEEE 754-2008 floating point (DP and SP)
6 GB GDDR5 DRAM (Global Memory)
ECC Memory support
Two DMA interface



Reconfigurable L1 Cache and Shared Memory
48 KB / 16 KB

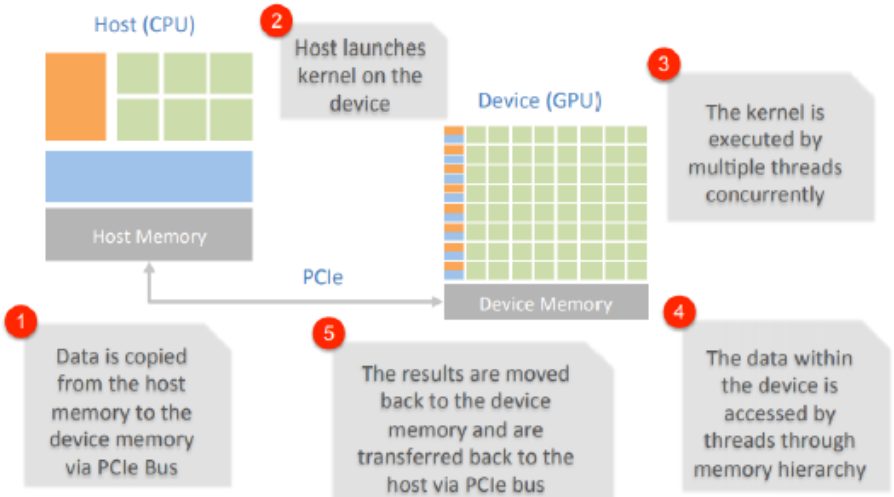
L2 Cache 768 KB



Load/Store address width 64 bits. Can calculate addresses of 16 threads per clock.

- La característica clave es que todos los núcleos en un SM son núcleos SIMT (Simple Instruction Multiple Threads):
  - Grupos de 32 núcloesejecutan las mismas instrucciones simultáneamente, pero con datos diferentes. Conocidos como warp.
  - Especializada en computación vectorial (tipo CRAY).
  - Especializadas en el procesamiento de gráficos y cómputo científico.
- Muchos hilos activos son la clave del alto rendimiento:
  - No hay cambio de contexto; cada hilo tiene sus propios registros, aunque esto limita el número de hilos activos.
  - La ejecución se alterne entre warpsactivos y warpstemporalmente inactivos (los que están esperando por datos).

Ejecución de un programa



- CUDA Kernels:
  - Se le llama así a la porción paralela de la aplicación.
  - Toda la GPU (o parte) utiliza el mismo kernel.
  - Los kernelsson capaces de crear, de manera eficiente, miles de hilos CUDA.

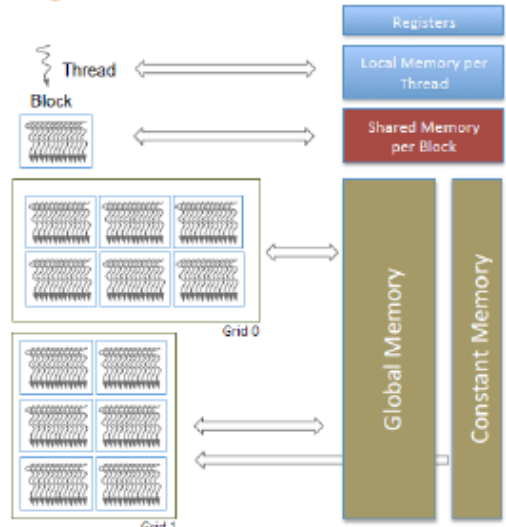
Memory Hierarchy

Private memory
Visible only to the thread

Shared memory
Visible to all the threads in a block

Global memory
Visible to all the threads
Visible to host
Accessible to multiple kernels
Data is stored in row major order

Constant memory (Read Only)
Visible to all the threads in a block



```
add<<< N, 1 >>>();
```

Cada invocación de **add()** se conoce como un block

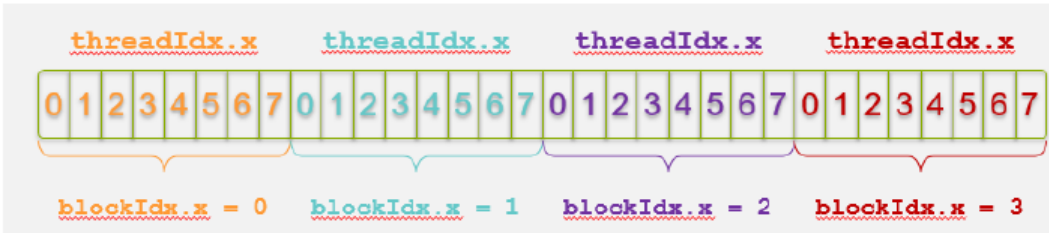
- El conjunto de bloques se conoce como grid
- Cada invocación puede referirse a su índice de bloque usando **blockIdx.x**

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

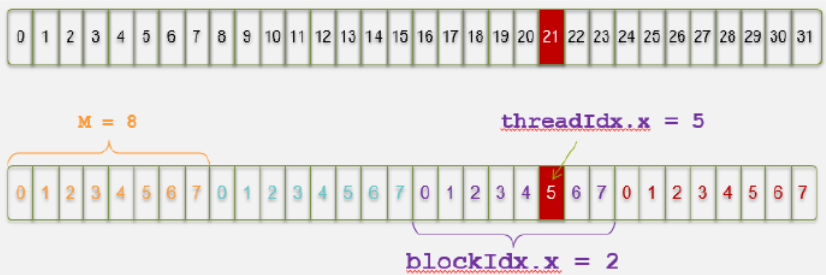
Al usar **blockIdx.x** para indexar la matriz, cada bloque maneja un índice diferente.

```
add<<< 1, N >>>();

__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```




```
int index = threadIdx.x + blockIdx.x * M;
```




```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2      * 8;
          = 21;
```

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

Programando en CUDA.

Revisar actividad en Canvas.

Un **hilo cooperativo** es aquel que **puede afectar o verse afectado por otros hilos** 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**.

Ver código de sincronización

Como pudieron observar, **llegamos a este estado incorrecto** porque permitimos que ambos hilos manipulen el contador **al mismo tiempo**. Una situación como esta, en la que **varios hilos 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 hilo a la vez** pueda manipular el contador. Para hacer tal garantía, requerimos que los hilos **estén sincronizados de alguna manera**.

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

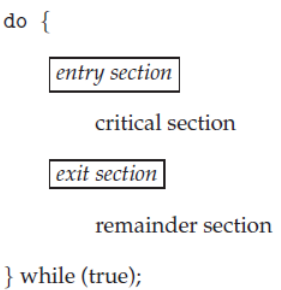


Figure 5.1 General structure of a typical process P_i.

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").

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

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

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

- El valor de un **semáforo binario** (**mutex**) 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**.

- La implementación de un semáforo con una cola de espera puede resultar en una situación en la que **dos o más hilos están esperando indefinidamente** un evento que **solo puede ser causado por uno de los hilos en espera** (**interbloqueo**, **deadlock**).
- Otro problema relacionado con los interbloqueos es la **muerte por inanicación** (**starvation**), una situación en la que los hilo esperan indefinidamente dentro del semáforo.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

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.

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.

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.

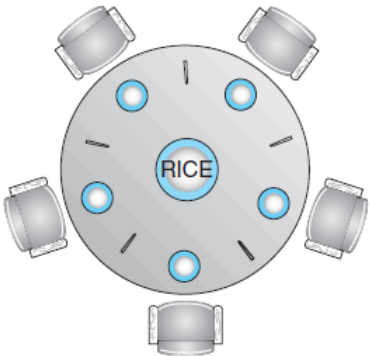


Figure 5.13 The situation of the dining philosopher

Implementando problemas de sincronización.

Revisar actividad en Canvas.