Módulo 3 - Arquitectura de Memorias Compartidas

Pedro O. Pérez M., PhD.

08-2025

Diseño de Sistemas Embebidos Avanzados
Tecnológico de Monterrey
pperezm@tec.mx

Contenido i

Contenido ii

- Sistemas Multiprocesador Simétrico (SMP)

Contenido iii

Contenido iv

Contenido vi

Contenido vii

Contenido viii

- Ejemplos de Programación en CUDA

Contenido ix

Conceptos Básicos

¿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.

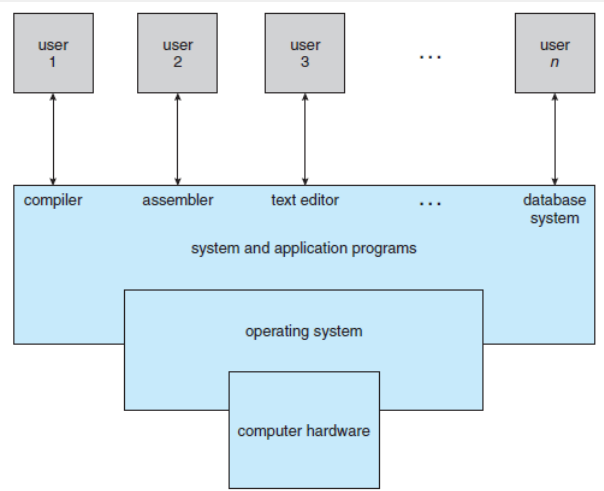


Figure 1.1 Abstract view of the components of a computer system.

Proceso

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.

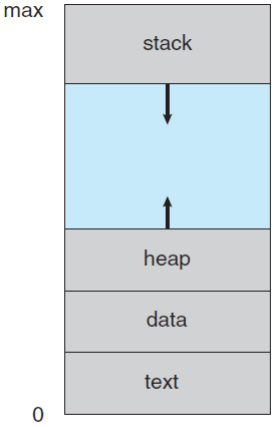


Figure 3.1 Process in memory.

Administración de procesos

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.

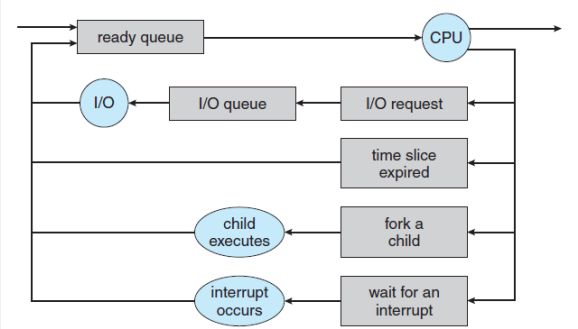
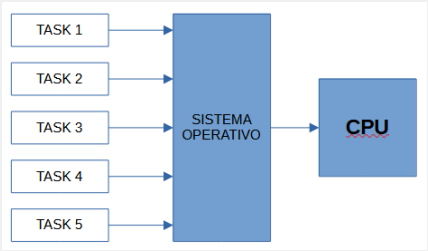


Figure 3.6 Queueing-diagram representation of process scheduling.

Multitarea

- La multitarea es la característica de los sistemas operativos modernos que permite que varios procesos (tareas o aplicaciones) se ejecuten aparentemente al mismo tiempo, compartiendo uno o más procesadores.
- Los sistemas operativos más comunes emplean la multitarea apropiativa o multitarea preventiva. En este tipo de multitarea, el sistema operativo es el encargado de administrar el/los procesador(es) repartiendo el tiempo de uso entre los procesos que estén esperando para utilizarlo. Cada proceso utiliza el procesador durante lapsos cortos, pero el resultado final es virtualmente igual a ejecutarse todo al mismo tiempo.



Concurrencia vs. Paralelismo

- 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 (multitarea), 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**.

Arquitectura de Memoria Compartida

- 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.

Existen dos tipos de sistemas multiprocesador:

- Arquitecturas de Procesamiento Distribuido: Los procesadores no comparten memoria física, lo que significa que cada procesador tiene su propia memoria local y no puede acceder directamente a la memoria de otros procesadores. En cambio, la comunicación y el intercambio de datos entre los procesadores se realizan a través de la red de comunicación.
- Arquitecturas de Memoria Compartida: En este tipo de arquitectura los procesadores tienen acceso directo a un espacio de memoria común. Esto permite a los procesadores compartir información y comunicarse entre sí mediante el uso de variables compartidas.

Arquitecturas de Memoria Compartida

- En un modelo de memoria compartida, los procesadores pueden leer y escribir en las mismas direcciones de memoria. Esto facilita la comunicación y la sincronización entre los procesadores, ya que pueden compartir datos y actualizarlos de manera simultánea.
- Existen diferentes formas de implementar un modelo de memoria compartida, como el uso de buses de datos compartidos, sistemas multiprocesador simétricos (SMP) o sistemas de memoria distribuida con acceso uniforme (NUMA).

Sistemas de Memoria Distribuida con Acceso Uniforme (NUMA)

- En un sistema NUMA, cada procesador tiene su propio conjunto de memoria local, pero también puede acceder a la memoria de otros procesadores. Sin embargo, el acceso a la memoria local es más rápido que el acceso a la memoria remota de otros procesadores. Imaginemos un servidor con 4 procesadores en el que cada procesador tiene una memoria local. Cada procesador puede acceder rápidamente a su propia memoria local, pero si necesita acceder a la memoria de otro procesador, tendrá que realizar una operación de acceso remoto que es más lenta.
- Este tipo de sistema es beneficioso en aplicaciones donde los datos se distribuyen de manera natural entre los procesadores. Por ejemplo, en una simulación numérica en la que cada procesador se encarga de un subconjunto de datos independiente, tener acceso rápido a la memoria local reduce la latencia y mejora el rendimiento.

Sistemas de Memoria Distribuida con Acceso Uniforme (NUMA)

- Sin embargo, si los procesadores necesitan acceder frecuentemente a datos que están alojados en la memoria local de otros procesadores, puede generar cuellos de botella y reducir el rendimiento.
- En resumen, un sistema NUMA ofrece una arquitectura de memoria distribuida en la que cada procesador tiene su propia memoria local y puede acceder a la memoria de otros procesadores, pero con diferentes tiempos de acceso dependiendo de si la memoria es local o remota.

Sistemas Multiprocesador Simétrico (SMP)

- En un sistema SMP, se utilizan múltiples procesadores idénticos que comparten una única memoria principal. Imaginemos un servidor con dos procesadores idénticos, cada uno con sus propias unidades de ejecución y cachés. Ambos procesadores pueden acceder a la memoria principal de manera uniforme y a la misma velocidad. Esto significa que no hay distinción en términos de latencia o tiempo de acceso a la memoria entre los procesadores.
- En un sistema SMP, los procesadores pueden ejecutar tareas de manera paralela y compartir la carga de trabajo de manera equitativa. Esto permite un mejor rendimiento y una mayor capacidad de procesamiento en comparación con un sistema de un solo procesador.

- El sistema operativo que se ejecuta en un sistema SMP debe tener la capacidad de administrar y distribuir las tareas de manera equitativa entre los procesadores, brindando a cada uno de ellos un tiempo justo de ejecución y acceso a los recursos compartidos.
- En resumen, los sistemas multiprocesador simétricos (SMP) son comunes en servidores y computadoras de alto rendimiento, donde múltiples procesadores idénticos comparten una memoria principal y trabajan en paralelo para brindar un mayor rendimiento y capacidad de procesamiento.

Modelo de computación

Modelo de computación

Todos los sistemas informáticos trabajan con un conjunto (o flujo) de instrucciones y un conjunto de datos de entrada. Según el flujo de instrucciones y el flujo de datos, las computadoras se pueden clasificar en cuatro categorías:

- Flujo de instrucciones único, flujo de datos únicos (Single Instruction stream, Single Data stream - SISD).
- Flujo de instrucciones únicos, flujo de datos múltiples (Single Instruction stream, Multiple Data stream - SIMD).
- Flujo de instrucciones múltiples, flujo de datos único (Multiple Instruction stream, Single Data stream - MISD).
- Flujo de instrucciones múltiples, flujo de datos múltiples (Multiple Instruction stream, Multiple Data stream - MIMD).

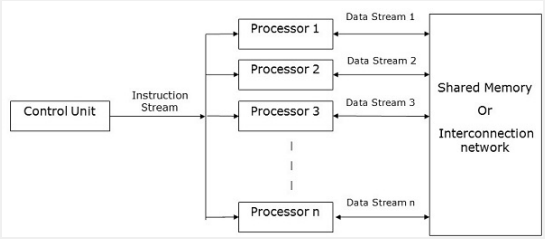
Single Instruction stream, Single Data stream - SISD

En este tipo de computadoras, el procesador recibe un único flujo de instrucciones y opera con un único flujo de datos procedente de la memoria.

Fuente: `https://www.tutorialspoint.com/parallel_algorithm/images/sisd_computers.jpg`

Single Instruction stream, Multiple Data stream - SIMD

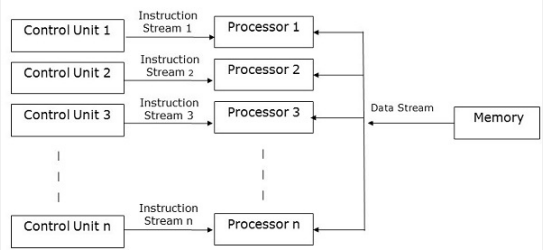
- Una sola unidad de control envía instrucciones a todas las unidades de procesamiento.
- Cada una de las unidades de procesamiento tienen su propia unidad de memoria para almacenar tanto los datos como las instrucciones.
- Algunos procesadores se encuentra ejecutando su conjunto de instrucciones, mientras otros esperan su siguiente conjunto.



Fuente:
`https://www.tutorialspoint.com/parallel_algorithm/images/simd_computers.jpg`

Multiple Instruction stream, Single Data stream - MISD

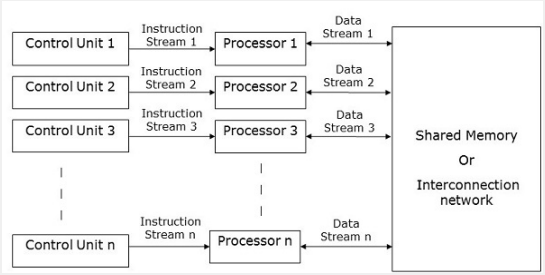
Cada procesador tiene su propia unidad de control y comparten una unidad de memoria común. Todos los procesadores reciben instrucciones individualmente de su propia unidad de control y operan en un solo flujo de datos según las instrucciones que han recibido de sus respectivas unidades de control.



Fuente:
https://www.tutorialspoint.com/parallel_algorithm/images/misd_computers.jpg

Multiple Instruction stream, Multiple Data stream - MIMD

- Cada procesador tiene su propia unidad de control, unidad de memoria local y unidad aritmética y lógica. Reciben diferentes conjuntos de instrucciones de sus respectivas unidades de control y operan con diferentes conjuntos de datos.
- Una computadora MIMD que comparte una memoria común se conoce como multiprocesadores, mientras que las que utilizan una red de interconexión se conocen como multicomputadoras.



Fuente:
`https://www.tutorialspoint.com/parallel_algorithm/images/mimd_computers.jpg`

Hilos

¿Qué es un hilo?

- 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 tarea (hilo de trabajo) que asigna a un hilo de ejecución (core).
- El S.O. asigna a cada hilo de trabajo 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.

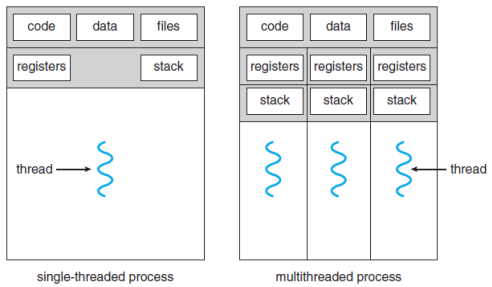


Figure 4.1 Single-threaded and multithreaded processes.

- 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 trabajo. 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.

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 de la programación multihilos pueden ser aún mayores en una arquitectura de multiprocesador, donde los hilos de trabajo pueden ejecutarse en paralelo en diferentes núcleos de procesamiento (hilos de ejecución).

Revisar código en Github (01-intro)

Programación Paralela

Retos de la programación paralela

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.

Tipos de Paralelismo

- Paralelismo de tareas.
- Paralelismo de datos.

Paralelismo de tareas

Definición

- 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.

El Problema de la sección crítica

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**.

Definición del problema

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**.

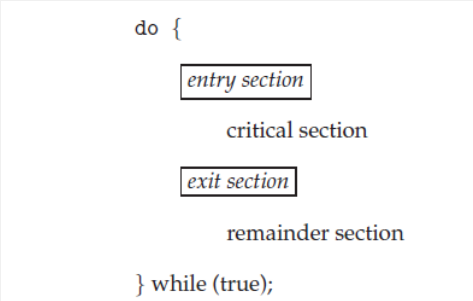


Figure 5.1 General structure of a typical process P_i.

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.

Semáforos

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**.

Deadlocks

- 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);`

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.

Problema Lectores-Escritores

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.

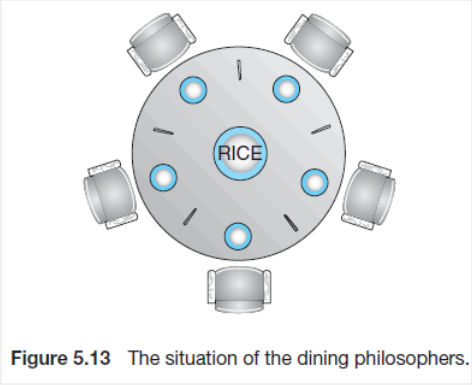


Figure 5.13 The situation of the dining philosophers.

Problema Lectores-Escritores

Hay cuatro hilos involucrados: un agente y tres fumadores. Los fumadores se repiten una y otra vez, primero esperando los ingredientes, luego fabricando y fumando cigarrillos. Los ingredientes son tabaco, papel y cerillas. Suponemos que el agente tiene un suministro infinito de los tres ingredientes y que cada fumador tiene un suministro infinito de uno de los ingredientes; es decir, un fumador tiene cerillas, otro tiene papel y el tercero tiene tabaco. El agente elige repetidamente dos ingredientes diferentes al azar y los pone a disposición de los fumadores. Dependiendo de los ingredientes elegidos, el fumador con el ingrediente complementario debe recoger ambos recursos y continuar.

Revisar actividad en Canvas.

Programación Multihilos

- 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.

¿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:

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

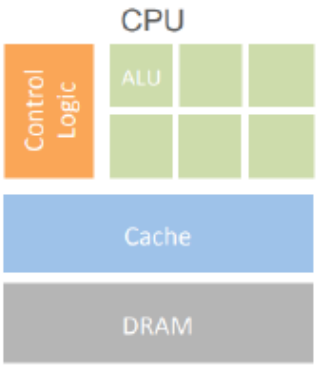
Ver ejemplos en Github (00-base, 03-threads)

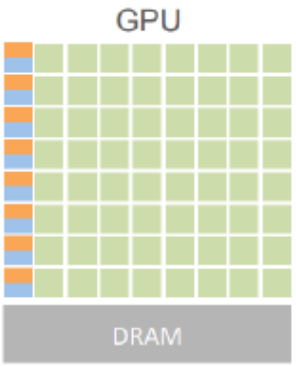
Revisar actividad en Canvas.

CUDA

Cómputo heterogéneo

- 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.





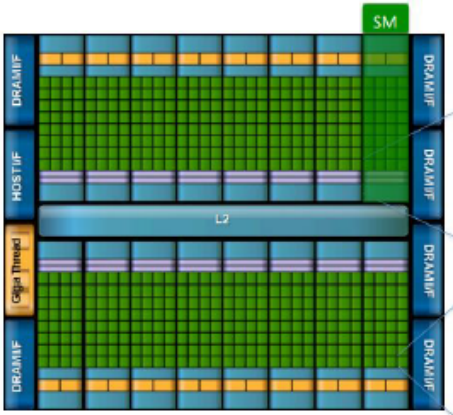
Less than 20 cores
1-2 threads per core
Latency is hidden by large cache

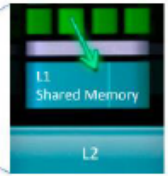
512 cores
10s to 100s of threads per core
Latency is hidden by fast context switching

GPUs don't run without CPUs

NVIDIA FERMI

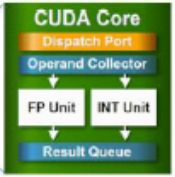
16 Stream Multiprocessors (SM)
512 CUDA cores (32/SM)
IEEE 754-2008 floating point (DP and SP)
6 GB GDDR5 DRAM (Global Memory)
ECC Memory support
Two DMA interface





Reconfigurable L1 Cache and Shared Memory
48 KB / 16 KB

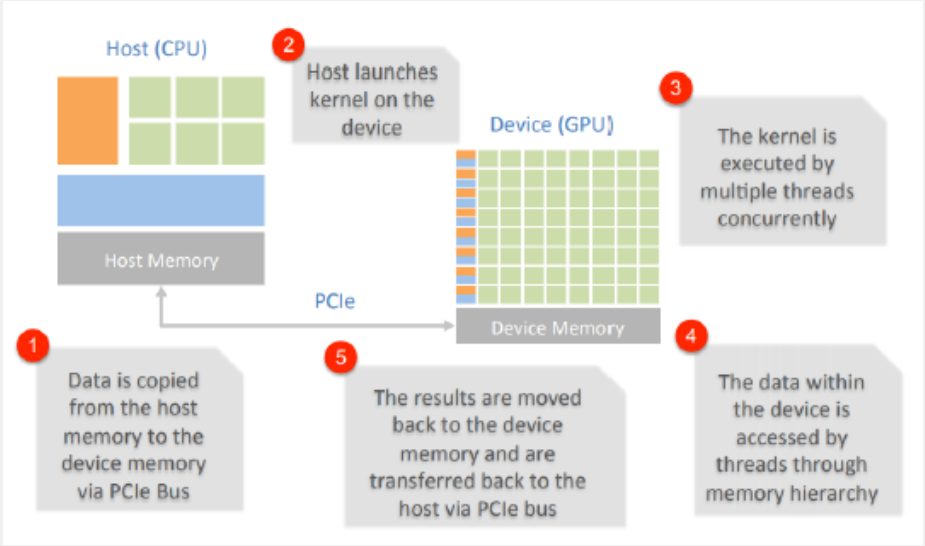
L2 Cache 768 KB



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 kernels son capaces de crear, de manera eficiente, miles de hilos CUDA.

Asignación de tareas (kernels)

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

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

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

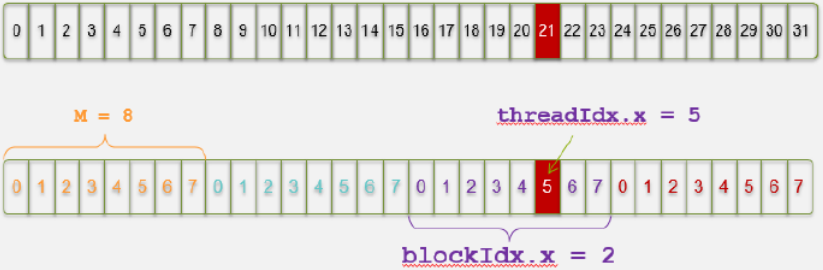
Cada invocación de **add()** se conoce como un block

- 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.


```
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];
}
```

Paralelizando tareas con CUDA.

Revisar actividad en Canvas.

Thread Pool

Descripción

- Uno pudiera pensar que si utilizando dos hilos se logra una reducción de más del 50 % del tiempo de ejecución (cómo vimos antes), crear muchos más hilos se puede lograr una mayor reducción.
- Sin embargo, esto no es cierto. Crear múltiples hilos es una tarea costosa en cuanto términos computacionales.

- Imaginemos que tenemos la necesidad ejecutar 1,000 tareas de forma concurrente. En primera instancia uno pudiera pensar que lo más sencillo será crear y ejecutar 1,000 hilos, uno por cada tarea y listo, problema resuelto.
- Sí hacemos esto no solo estaremos perjudicando el desempeño de nuestra aplicación, obteniendo un programa que incluso puede llegar a ser más lento que si trabajamos con un solo hilo.

Solución

- Un grupo de hilos (thread pool) reutiliza subprocesos creados previamente para ejecutar tareas y ofrece una solución al problema de la sobrecarga de creación de hilos y el uso indebido de recursos.
- Dado que el hila ya existe cuando llega la solicitud, se elimina el retraso introducido por la creación del hilo, lo que hace que la aplicación responda mejor.

- La implementación varía según el entorno, pero en términos simplificados, necesitamos lo siguiente:
  - Una forma de crear hilos y mantenerlos en un estado inactivo. Esto se puede lograr haciendo que cada hilo espere en una barrera hasta que el grupo le entregue trabajo.
  - Un contenedor para almacenar los hilos creados, como una cola o cualquier otra estructura que tenga una forma de agregar un hilo al grupo y extraer uno.
  - Una interfaz estándar o una clase abstracta para que los hilos la utilicen para realizar el trabajo. Esta podría ser una clase abstracta llamada Task con un método de ejecución () que realiza el trabajo y luego regresa.

Ver implementación de un grupo de hilos en Github.

Actividad Final

Revisar actividad en Canvas.