ARQUITECTURA Y SISTEMAS OPERATIVOS

CLASE 5

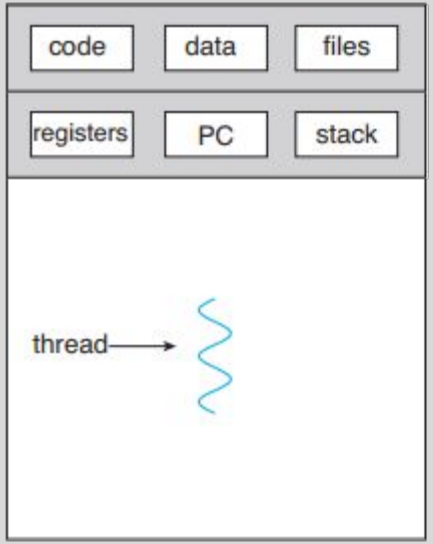
PLANIFICACIÓN DE LA CPU

REPASO

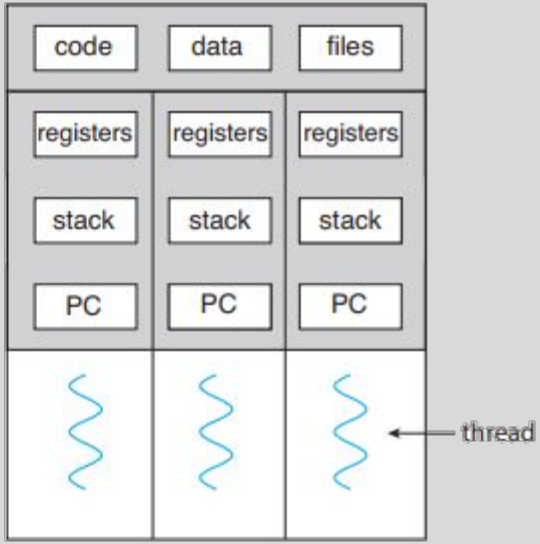
THREADS (HILOS)

Un *thread* (hilo) es la unidad de ejecución más pequeña dentro de un proceso. Mientras que un proceso es un programa independiente en ejecución con su propio espacio de memoria, un hilo representa **una única secuencia de instrucciones** dentro de ese proceso.

Permiten a las aplicaciones realizar múltiples actividades a la vez, incluso algunas de ellas pueden bloquearse. Son más livianos que los procesos, y por lo tanto, son más fáciles de crear y destruir. Y tener hilos permite que estas actividades se superpongan, acelerando así la aplicación.



single-threaded process



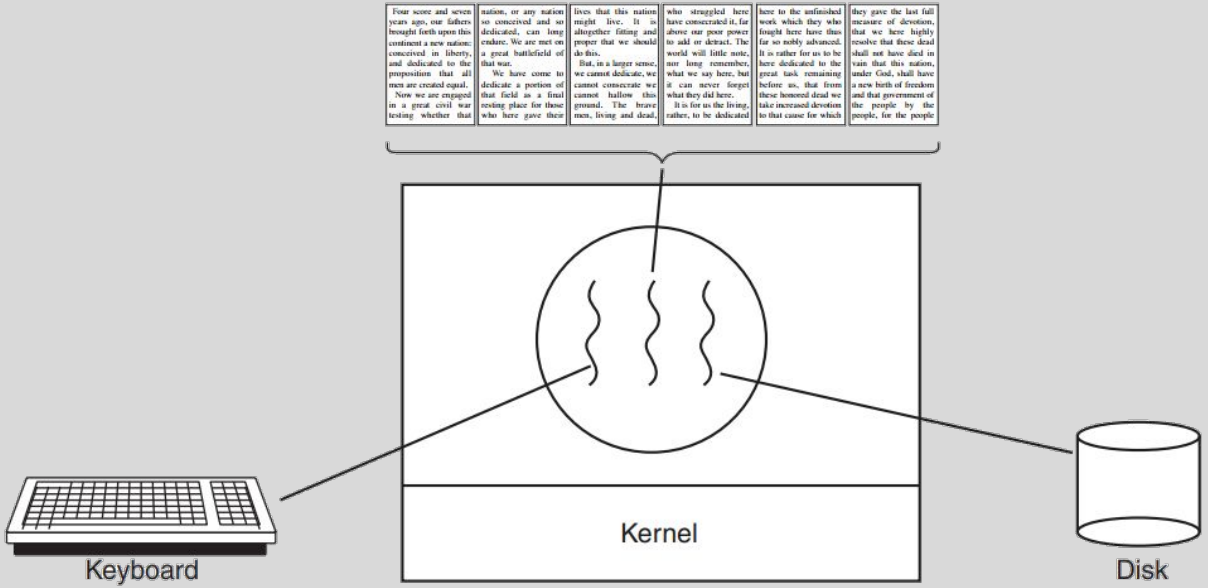
multithreaded process

HILOS - EJEMPLO

Consideren un procesador de texto simple que realiza 3 tareas en simultáneo:

1. Atender la E/S que ingresa el usuario por teclado.
2. Ejecutar la función de auto-corrector.
3. Realizar un *backup* del documento periódicamente.

Debe quedar claro que tener tres procesos separados no funcionaría, ya que las tres tareas deben operar en el mismo documento. Al tener tres hilos en lugar de tres procesos, estos comparten memoria y, por lo tanto, todos tienen acceso al documento que se está editando. Con tres procesos, esto sería imposible.

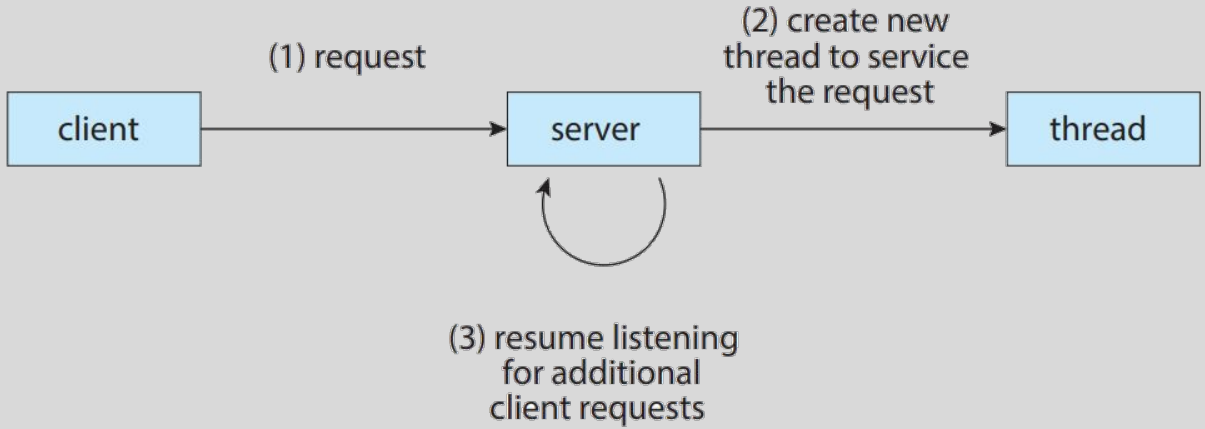


HILOS - EJEMPLO

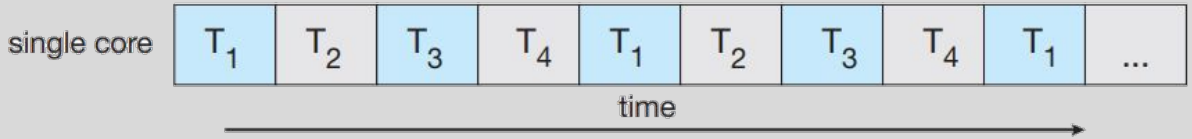
Si un servidor web se ejecuta como un único proceso que acepta *requests*, cuando recibe una solicitud, crea un nuevo proceso para atenderla.

Pero como vimos, la creación de procesos es costosa. En cambio, si el nuevo proceso va a realizar las mismas tareas que el padre, generalmente es más eficiente usar un proceso que cree un hilo para atender la solicitud.

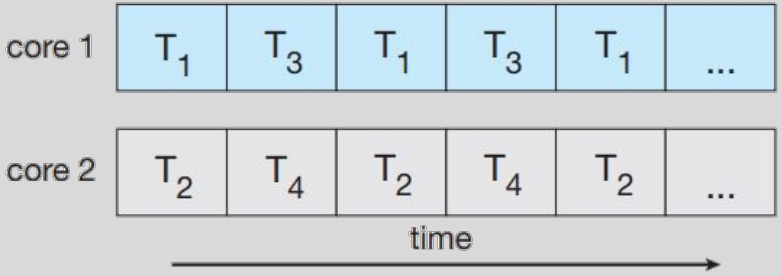
Si el proceso del servidor web es *multithread*, el servidor creará un nuevo hilo que escucha las solicitudes del cliente, así, cuando llega una una solicitud, en lugar de crear otro proceso, se crea un nuevo hilo para atenderla, y rápidamente reanuda la escucha de solicitudes siguientes.



MULTICORE



Concurrent execution on a single-core system.



Parallel execution on a multicore system.

En un sistema *multicore*, la concurrencia significa que algunos hilos pueden ejecutarse en **paralelo**, ya que el sistema puede asignar un hilo independiente a cada núcleo.

Importante: notar la distinción entre concurrencia y paralelismo.

- Un sistema **concurrente** admite más de una tarea permitiendo que todas las tareas avancen.
- Un sistema **paralelo** puede realizar más de una tarea simultáneamente.

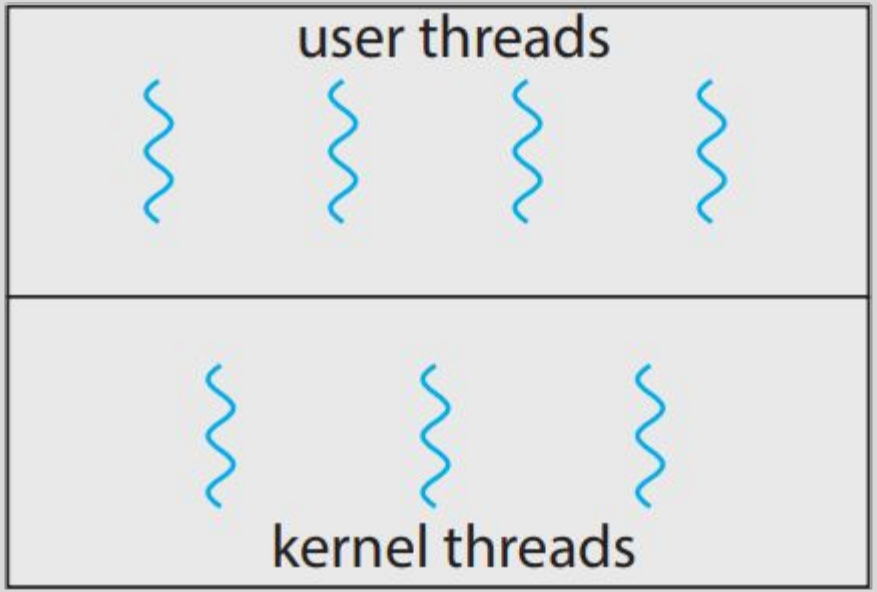
Es posible tener concurrencia sin paralelismo.

MODELOS MULTIHILOS

El soporte para hilos puede brindarse a nivel de usuario o por el kernel. Los hilos de usuario se gestionan sin soporte del kernel, mientras que los hilos de kernel son soportados y gestionados directamente por el sistema operativo.

En definitiva, debe existir una relación entre los hilos de usuario y los hilos de kernel. Comunmente, hay tres formas de establecer esta relación:

- muchos a uno
- uno a uno
- muchos a muchos



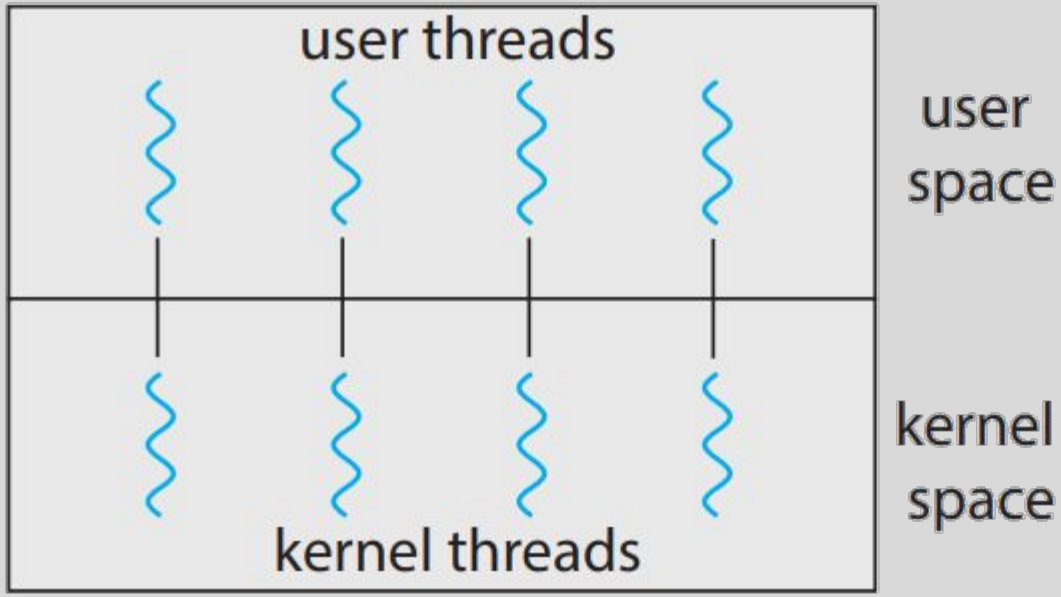
User and kernel threads.

UNO A UNO

El modelo uno a uno asigna cada hilo de usuario a un hilo del *kernel*. Tiene mayor concurrencia que el modelo muchos a uno, porque permite que otro hilo se ejecute cuando un hilo realiza una *syscall* bloqueante. Y permite que varios hilos se ejecuten en paralelo en sistemas multi núcleos.

La desventaja de este modelo es que la creación de un hilo de usuario, requiere la creación del hilo del *kernel* correspondiente, y un gran número de hilos del *kernel* puede afectar negativamente al rendimiento.

Linux, y Windows, implementan este modelo.



One-to-one model.

PTHREADS

src > 03-threads > unix > pthread_example_sum_of_n.c > ...

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid;       /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */
  /* set the default attributes of the thread */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid, NULL);
  printf("sum = %d\n", sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
  {
    sum += i;
    // usleep(1000); // Sleep for 1 millisecond
  }
  pthread_exit(0);
}
// Compile with: gcc -o pthread_example_sum_of_n pthread_example_sum_of_n.c -lpthread
// Run with: ./pthread_example_sum_of_n 1000000
```

WINDOWS THREADS

src > 03-threads > windows > windows_sum_of_n.c > ...

```c
#include <windows.h>
#include <stdio.h>

DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
  DWORD Upper = *(DWORD *)Param;
  for (DWORD i = 1; i <= Upper; i++)
    Sum += i;
  return 0;
}

int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;
  Param = atoi(argv[1]);
  /* create the thread */
  ThreadHandle = CreateThread(
      NULL,       /* default security attributes */
      0,          /* default stack size */
      Summation,  /* thread function */
      &Param,     /* parameter to thread function */
      0,          /* default creation flags */
      &ThreadId); /* returns the thread identifier */
  /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle, INFINITE);
  /* close the thread handle */
  CloseHandle(ThreadHandle);
  printf("sum = %d\n", Sum);
}
```

JAVA

src > 03-threads > java > Driver.java

```java
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
  private int upper;
  public Summation(int upper) {
    this.upper = upper;
  }

  /* The thread will execute in this method */
  public Integer call()
  {
    int sum = 0;
    for (int i = 1; i <= upper; i++)
      sum += i;
    return sum;
  }
}

public class Driver
{
  public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);
    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));
    try {
      System.out.println("sum = " + result.get());
    }
    catch (InterruptedException | ExecutionException ie) {
      System.out.println("Exception: " + ie.getMessage());
    }
    finally {
      pool.shutdown();
    }
  }
}
// The above code is a simple Java program that uses the Executor framework
```

BIBLIOGRAFIA

BIBLIOGRAFIA

- Operating System Concepts. By Abraham, Silberschatz.
  - Capítulo V

TEMAS DE LA CLASE

- Conceptos básicos
  - Ráfagas de CPU-E/S
  - Planificador del CPU
  - Planificación Apropiativa y No Apropiativa
  - *Dispatcher*
- Criterios de Planificación
- Algoritmos de Planificación
  - First-Come, First-Served (FCFS)
  - Shortest-Job-First (SJF)
  - Round-Robin (RR)
  - Priority
  - Multilevel Queue
  - Multilevel Feedback Queue

TEMAS DE LA CLASE

- Planificación de Hilos
- Planificación de Múltiples Procesadores
  - Enfoques
  - Balance de carga

CONCEPTOS BÁSICOS

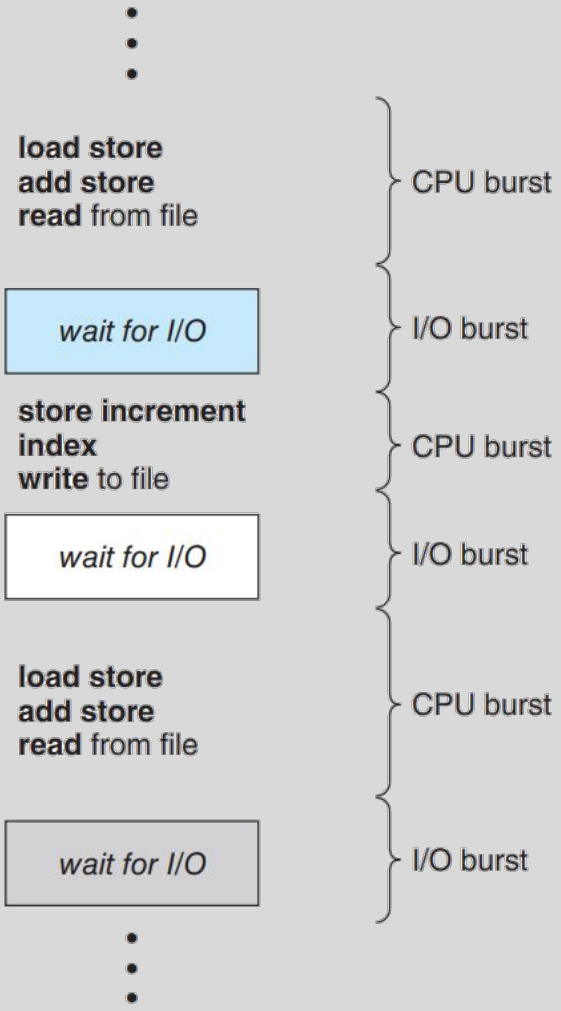
CONCEPTOS BÁSICOS

En un sistema simple, un proceso se ejecuta hasta que debe esperar, normalmente a que se complete una solicitud de E/S, en estos sistemas mientras se espera, la CPU permanece inactiva. Todo este tiempo de espera se desperdicia; no se realiza ningún trabajo útil. Con la multiprogramación se intenta utilizar ese tiempo de forma productiva. Varios procesos se mantienen en memoria simultáneamente, cuando un proceso tiene que esperar, el sistema operativo le quita la CPU y se la cede a otro. Así, cuando un proceso espera, otro puede asumir el uso de la CPU y generar progreso. En un sistema multinúcleo, este concepto de mantener la CPU ocupada se extiende a todos los núcleos de procesamiento del sistema.

Este tipo de planificación es una función fundamental del sistema operativo. Casi todos los recursos informáticos se planifican antes de su uso, la CPU es uno de los principales recursos de la computadora, y por lo tanto, su planificación es fundamental para el diseño del sistema operativo.

RAFAGAS DE CPU-E/S

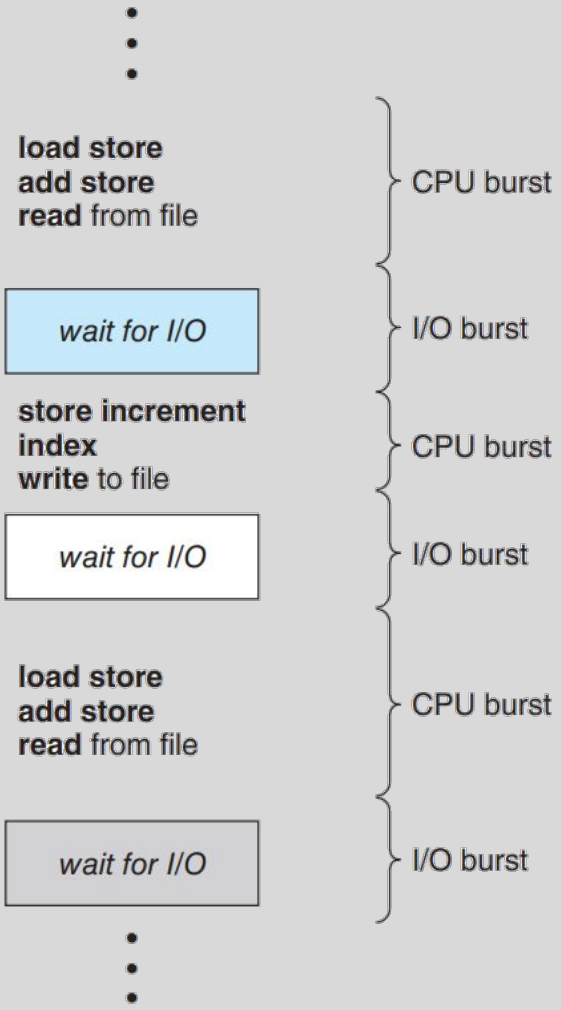
La planificación de la CPU depende de una propiedad observable en los procesos: su ejecución consiste en un ciclo de CPU y espera de E/S. La ejecución del proceso comienza con una ráfaga de CPU, le sigue una ráfaga de E/S, a la que le sigue otra ráfaga de CPU, y luego otra ráfaga de E/S, y así sucesivamente. Finalmente, una ráfaga de CPU termina con una solicitud del sistema para finalizar la ejecución.



RAFAGAS DE CPU-E/S

Un programa limitado por E/S suele tener muchas ráfagas cortas de CPU. Un programa limitado por CPU puede tener pocas ráfagas largas de CPU.

Esta distribución es importante al implementar un algoritmo de planificación de la CPU.



PLANIFICADOR DE LA CPU

Cada vez que la CPU queda inactiva, el sistema operativo debe seleccionar uno de los procesos de la cola de listos para ejecutarse. El proceso de selección lo realiza el **planificador de la CPU**, que selecciona un proceso de los procesos listos en memoria y le asigna la CPU.

La cola de espera de los procesos listos, puede implementarse como una cola FIFO (*first-in, first-out*), una cola de prioridad, un árbol o simplemente una lista enlazada desordenada. Más allá de su implementación, conceptualmente, todos los procesos de la cola de listos están esperando para ejecutarse en la CPU. Recuerden que los elementos de las colas de espera son generalmente bloques de control de procesos (PCB).

PLANIFICACIÓN APROPIATIVA Y NO APROPIATIVA

Las decisiones de planificación de la CPU pueden tomarse en cuatro momentos diferentes:

1. Cuando un proceso pasa del estado de ejecución al estado de espera (E/S).
2. Cuando un proceso pasa del estado de ejecución al estado de listo (Interrupción).
3. Cuando un proceso pasa del estado de espera al estado de listo (finaliza E/S).
4. Cuando un proceso termina.

En las situaciones 1 y 4, no hay opción de planificación. Solo se debe seleccionar un nuevo proceso para su ejecución. Para las situaciones 2 y 3 sí existe opción de planificación.

Para las circunstancias 1 y 4, se dice que el esquema de planificación es **no apropiativo**. Bajo este esquema, una vez asignada la CPU a un proceso, la conserva hasta que la libera, ya sea por finalizar o por pasar a un estado de espera. Un *kernel* no apropiativo esperará a que se complete una llamada al sistema o a que un proceso se bloquee mientras espera a que se complete la E/S, antes de realizar un cambio de contexto.

PLANIFICACIÓN APROPIATIVA Y NO APROPIATIVA

Desafortunadamente, la planificación apropiativa puede generar condiciones de carrera cuando se comparten datos entre varios procesos. Por ejemplo, cuando dos procesos comparten datos, puede darse que, mientras uno actualiza los datos, el SO puede apropiarse la CPU del primer proceso para que el segundo pueda ejecutarse. Así, cuando el segundo proceso intenta leer los datos, los encontrará en un estado inconsistente. Más allá de esto, prácticamente todos los sistemas operativos modernos (Windows, macOS, Linux), utilizan algoritmos de planificación apropiativa dado que suelen ser más eficientes.

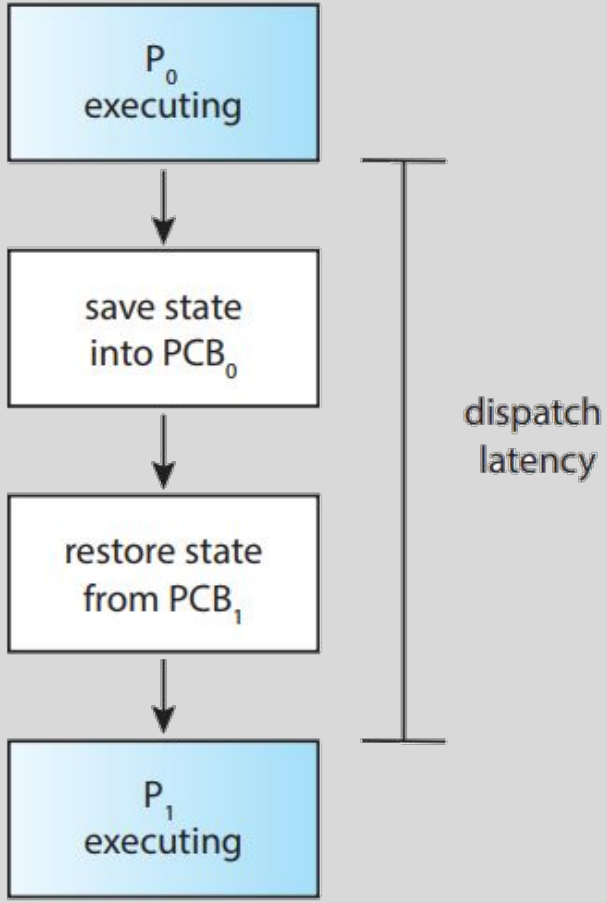
Un kernel apropiativo requiere mecanismos como bloqueos *mutex* para evitar condiciones de carrera al acceder a las estructuras de datos compartidas del *kernel*. Por ejemplo, el código de *kernel* que maneja las interrupciones. Como las interrupciones por definición pueden ocurrir en cualquier momento, las secciones de código afectadas por ellas deben protegerse del uso simultáneo. Para evitar que varios procesos accedan a estas secciones de código en el mismo momento, se desactivan las interrupciones al entrar y se reactivan al salir. Es importante destacar que las secciones de código que desactivan las interrupciones no son muy frecuentes y suelen contener pocas instrucciones.

DISPATCHER

El *dispatcher* (despachador) es el módulo que otorga el control del núcleo de la CPU al proceso seleccionado por el planificador. Su función implica:

- Cambiar el contexto de un proceso a otro
- Cambiar al modo de usuario
- Ir a la ubicación correcta en el programa de usuario para reanudarlo.

Debido a que se invoca durante cada cambio de contexto, el *dispatcher* debe ser lo más rápido posible. El tiempo que tarda el *dispatcher* en detener un proceso e iniciar otro se conoce como latencia de despacho.



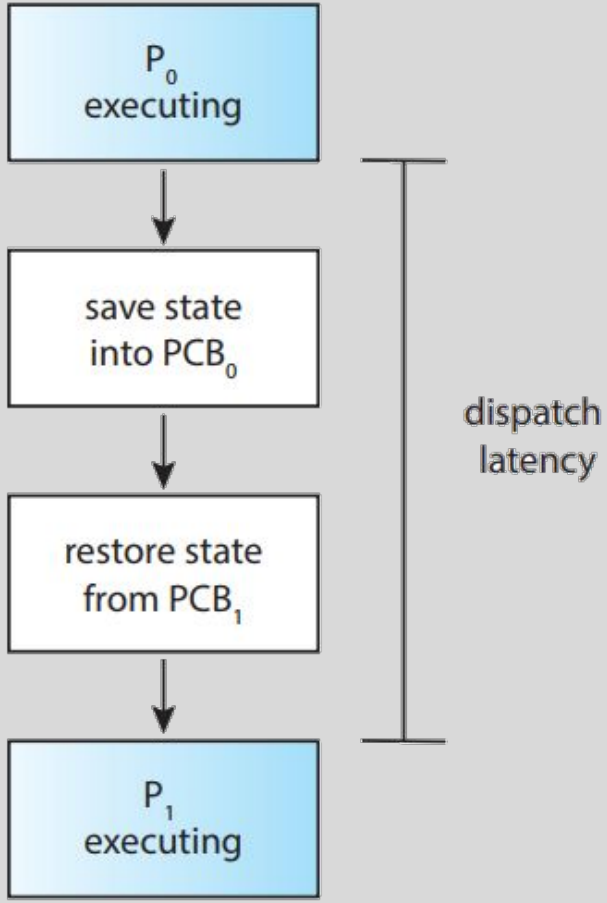
The role of the dispatcher

DISPATCHER

Una pregunta interesante puede ser: ¿con qué frecuencia ocurren cambios de contexto en un sistema?

- `vmstat 1 3`
- `cat /proc/<PID>/status`

Notar la distinción entre cambios de contexto voluntarios e involuntarios. Un cambio de contexto voluntario ocurre cuando un proceso cede el control de la CPU porque requiere un recurso que no está disponible (como el bloqueo de E/S). Un cambio de contexto involuntario ocurre cuando se retira la CPU de un proceso, como cuando su franja de tiempo ha expirado o ha sido apropiada por un proceso de mayor prioridad.



The role of the dispatcher

CRITERIOS DE PLANIFICACIÓN

CRITERIOS DE PLANIFICACIÓN

Se han sugerido muchos criterios para comparar algoritmos de planificación, los cuales incluyen:

- **Utilización de la CPU**. Mantener la CPU lo más ocupada posible.
- ***Throughput***. Una medida del trabajo realizado es el número de procesos que se completan por unidad de tiempo, se denomina *throughput*.
- **Tiempo de retorno**. Es el intervalo de tiempo desde el momento en que se despacha un proceso, hasta su finalización. Es la suma de los tiempos de espera en la cola de listos, la ejecución en la CPU y las operaciones de E/S.
- **Tiempo de espera**. Es importante destacar que el algoritmo de planificación de la CPU solo afecta el tiempo que un proceso pasa esperando en la cola de listos. El tiempo de espera es **la suma de los períodos de espera** en la cola de listos.
- **Tiempo de respuesta**. Es el tiempo que tarda un proceso en comenzar a responder, no el tiempo que tarda en generar la respuesta.

Lo deseable: maximizar la utilización y el rendimiento de la CPU, y minimizar los tiempos de retorno, de respuesta, y de espera. En la mayoría de los casos, se busca optimizar la medida promedio.

ALGORITMOS DE PLANIFICACIÓN

ALGORITMOS DE PLANIFICACIÓN

La planificación se ocupa del problema de decidir a qué proceso de la cola de listos se le asignará un núcleo de la CPU. Si bien la mayoría de las arquitecturas de CPU modernas tienen múltiples núcleos de procesamiento, por simplificación, describimos los siguiente algoritmos de planificación en el contexto de un solo núcleo de procesamiento disponible. Es decir, el sistema solo puede ejecutar un proceso a la vez.

Aclaración, los ejemplos a continuación sólo consideran una ráfaga de CPU (en milisegundos) por proceso. Y la medida de comparación es el **tiempo de espera promedio**.

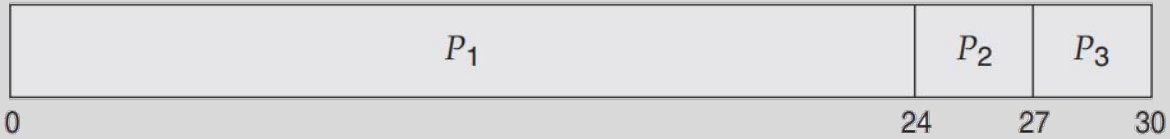
FIRST-COME, FIRST-SERVED

Es el algoritmo de planificación más simple, por orden de llegada (FCFS). En este esquema, el proceso que primero solicita la CPU, la recibe. La implementación de FCFS se logra mediante una cola de espera FIFO. Cuando un proceso entra en la cola de listos, su PCB se vincula al final, y cuando la CPU está libre, se asigna al primer proceso la cola. El tiempo de espera promedio bajo la política FCFS suele ser bastante largo.

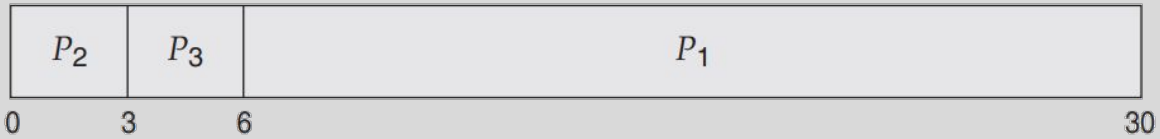
Por ejemplo, el siguiente conjunto de procesos llegan en el tiempo 0, con la duración de la ráfaga de CPU expresada en milisegundos.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Diagrama de Gantt



El tiempo de espera para P1 es de 0 ms, para P2, 24 ms y para P3, 27 ms.
Así, el **tiempo de espera promedio** es (0 + 24 + 27)/3 = **17 ms**.
En cambio, si los procesos llegan en orden P2, P3, P1:



El **tiempo de espera promedio** ahora es (6 + 0 + 3)/3 = **3 ms**.

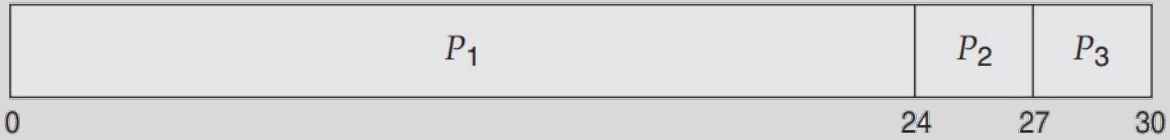
FIRST-COME, FIRST-SERVED

El algoritmo de planificación FCFS es no apropiativo. Una vez asignada la CPU a un proceso, éste la conserva hasta que la libera, ya sea finalizando o solicitando E/S.

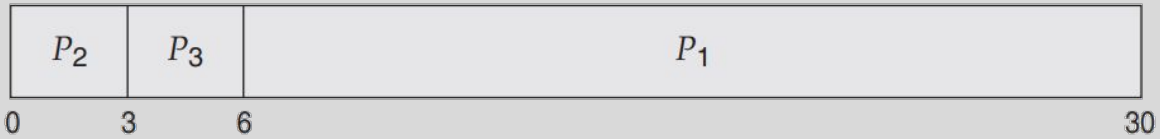
Por esto, el algoritmo FCFS no es bueno para sistemas interactivos, donde es importante que cada proceso obtenga una parte de la CPU a intervalos regulares.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Diagrama de Gantt



El tiempo de espera para P1 es de 0 ms, para P2, 24 ms y para P3, 27 ms.
Así, el **tiempo de espera promedio** es (0 + 24 + 27)/3 = **17 ms**.
En cambio, si los procesos llegan en orden P2, P3, P1:



El **tiempo de espera promedio** ahora es (6 + 0 + 3)/3 = **3 ms**.

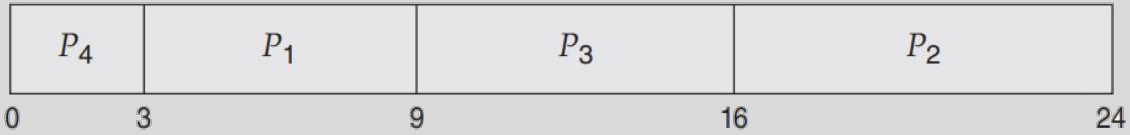
SHORTEST-JOB-FIRST

Este algoritmo de planificación "primero el trabajo más corto" (SJF) asocia a cada proceso la duración de su próxima ráfaga de CPU. Cuando la CPU está disponible, se asigna al proceso con la próxima ráfaga de CPU más corta. Si hay coincidencia, se utiliza la planificación FCFS.

Ejemplo:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Diagrama de Gantt



Para P1, el tiempo de espera es de 3 ms, para P2, de 16 ms, para P3, 9 ms y para el proceso P4, 0 ms. El tiempo de espera promedio es:
(3 + 16 + 9 + 0)/4 = **7 milisegundos**.
En comparación, si usáramos el esquema de planificación **FCFS**, el tiempo de espera promedio sería de **10,25 ms**.

SHORTEST-JOB-FIRST

Es posible demostrar que el algoritmo de planificación SJF es óptimo, es decir, proporciona el tiempo de espera promedio mínimo para un conjunto dado de procesos. Mover un proceso corto antes de uno largo disminuye el tiempo de espera del proceso corto más de lo que aumenta el tiempo de espera del proceso largo. En consecuencia, el tiempo de espera promedio disminuye.

Aunque el algoritmo SJF es óptimo, no puede implementarse a nivel de planificación de CPU, porque no hay forma de conocer, a priori, la duración de la próxima ráfaga de CPU del proceso. Una posible solución es intentar aproximar la planificación SJF. Si bien no es posible conocer la duración de la próxima ráfaga de CPU, se puede intentar predecir su valor. Asumiendo que la siguiente ráfaga de CPU tenga una duración similar a las anteriores. Así, al calcular una aproximación de la duración de la siguiente ráfaga de CPU, podemos seleccionar el proceso con la ráfaga de CPU prevista más corta.

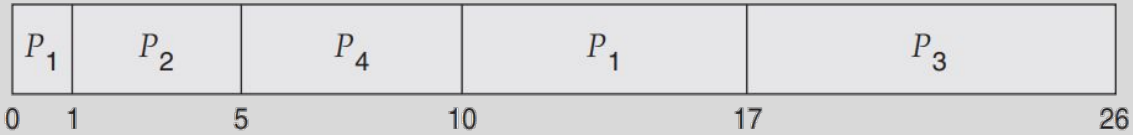
SHORTEST-JOB-FIRST

El algoritmo SJF puede ser apropiativo o no apropiativo. La elección surge cuando un nuevo proceso llega a la cola de listos mientras un proceso anterior aún se está ejecutando. Como la próxima ráfaga de CPU del proceso recién llegado puede ser más corta que el tiempo restante del proceso en ejecución, un algoritmo SJF apropiativo, le apropia la CPU al proceso en ejecución. Mientras que un algoritmo SJF no apropiativo permite que el proceso en ejecución finalice su ráfaga de CPU.

Por ejemplo, considerar los siguientes 4 procesos:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Diagrama de Gantt



P1 inicia en t0. P2 llega en t1. En ese instante, el tiempo restante de P1 (7 ms) es mayor que el tiempo requerido por P2 (4 ms), entonces se interrumpe el proceso P1 y se planifica el proceso P2. El tiempo de espera promedio para este ejemplo es [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = **6,5 ms**.
La planificación SJF no apropiativa resultaría en un tiempo de espera promedio de **7,75 ms**.

ROUND-ROBIN

El algoritmo de planificación round-robin (RR) es similar a FCFS, la diferencia es que se agrega la apropiación para permitir que el sistema alterne entre procesos. Lo que se hace es definir una pequeña unidad de tiempo, denominada *quantum*, que suele tener una duración de entre 10 y 100 ms. Además, la cola de listos se implementa como una cola circular, es decir, el siguiente elemento después del último, es el primero. Y el planificador de la CPU le asigna la CPU a cada proceso durante un intervalo de hasta 1 *quantum* de tiempo. Los nuevos procesos se añaden al final de la cola, y el planificador selecciona siempre el primer proceso de la cola de listos, configura un *timer* para interrumpirlo después de 1 quantum de tiempo y lo despacha.

En ese caso, ocurrirá una de dos cosas:

1. El proceso puede tener una ráfaga de CPU inferior a 1 *quantum* de tiempo. En este caso, el propio proceso liberará la CPU voluntariamente.
2. Si la ráfaga de CPU del proceso en ejecución supera 1 *quantum* de tiempo, el *timer* se detendrá y provocará una interrupción. Se ejecutará un cambio de contexto y el proceso se colocará al final de la cola de listos. El planificador de la CPU seleccionará el siguiente proceso en la cola de listos.

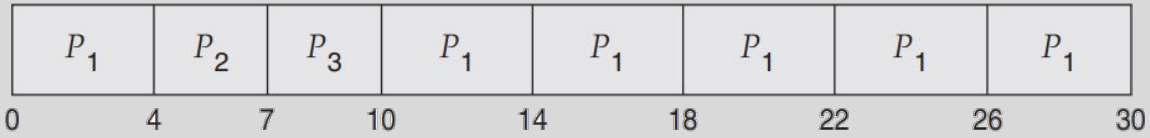
ROUND-ROBIN

Por ejemplo, con un *quantum* de 4 ms, el proceso P1 obtiene los primeros 4 ms, como requiere de otros 20 ms más, se interrumpe después del primer *quantum* y la CPU se asigna al siguiente proceso en la cola de listos. El proceso P2 no necesita 4 ms, por lo que finaliza antes de que expire su *quantum*. La CPU se asigna entonces al siguiente proceso, P3. Una vez que cada proceso recibe 1 *quantum* de tiempo, la CPU se devuelve al proceso P1 para un *quantum* de tiempo adicional.

Claramente, este algoritmo de planificación es apropiativo.

Process	Burst Time
P_1	24
P_2	3
P_3	3

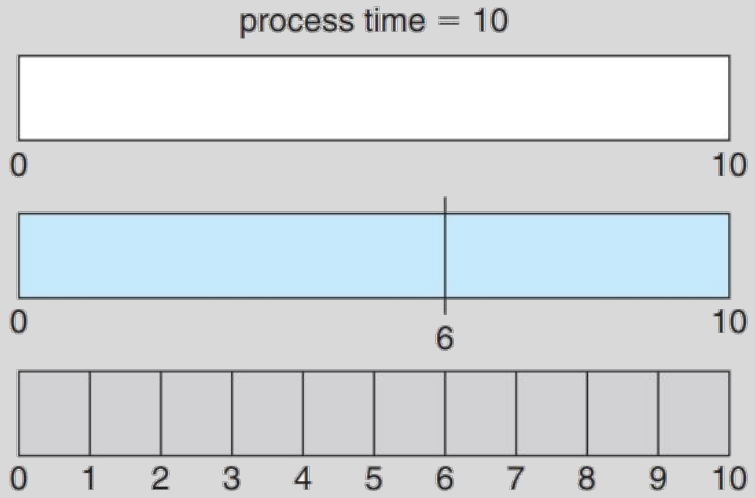
Diagrama de Gantt



El tiempo de espera promedio para esta planificación es: P1 espera 6 ms (10 − 4), P2 espera 4 ms y P3 espera 7 ms. Así, el tiempo de espera promedio es 17/3 = **5,66 ms**.

ROUND-ROBIN

El rendimiento del algoritmo RR depende mucho del *quantum* definido. En un extremo, si el *quantum* es extremadamente grande, la política de RR es la misma que la de FCFS. Por el contrario, si el *quantum* es extremadamente pequeño (1 ms), el algoritmo RR puede resultar en un gran número de cambios de contexto.



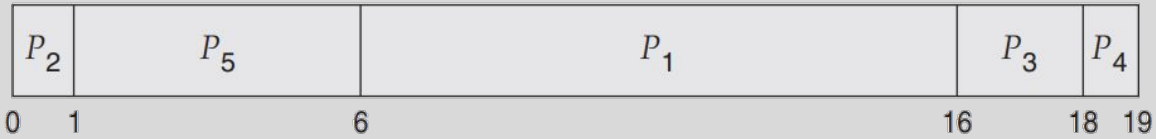
How a smaller time quantum increases context switches.

PRIORIDAD

El algoritmo SJF es un caso especial del algoritmo general de planificación por prioridad, donde cada proceso tiene una prioridad asociada y la CPU se asigna al proceso con la prioridad más alta. Los procesos con igual prioridad se programan en orden FCFS. Un algoritmo SJF es simplemente un algoritmo de prioridad donde la prioridad (p) es la inversa de la siguiente ráfaga de CPU. Cuanto mayor sea la ráfaga de CPU, menor será la prioridad, y viceversa. Se asume que los números bajos representan una prioridad alta.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Diagrama de Gantt



El tiempo de espera promedio es de **8,2 ms**.
Las prioridades pueden definirse interna o externamente.
Las prioridades internas utilizan una o varias cantidades medibles para calcular la prioridad de un proceso. Por ejemplo, los límites de tiempo, los requisitos de memoria, el número de archivos abiertos y la relación entre la ráfaga promedio de E/S y la ráfaga promedio de CPU.
Las prioridades externas se establecen mediante criterios externos al sistema operativo, como la importancia del proceso, el tipo y la cantidad de fondos que se destinan al uso de la computadora, el departamento que patrocina el trabajo y otros factores, a menudo políticos.
La planificación de prioridades puede ser apropiativa o no apropiativa.

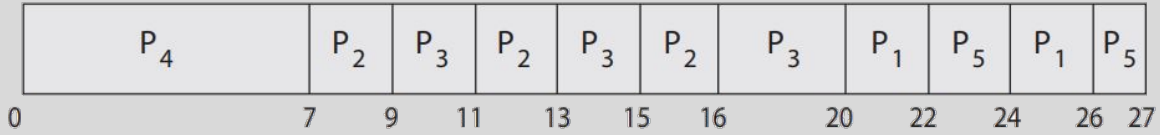
PRIORIDAD + ROUND-ROBIN

Otra opción es combinar la planificación RR y agregarle prioridad, de esta forma el sistema ejecuta el proceso de mayor prioridad y los procesos con la misma prioridad mediante la planificación RR.

Utilizando un *quantum* de 2 ms:

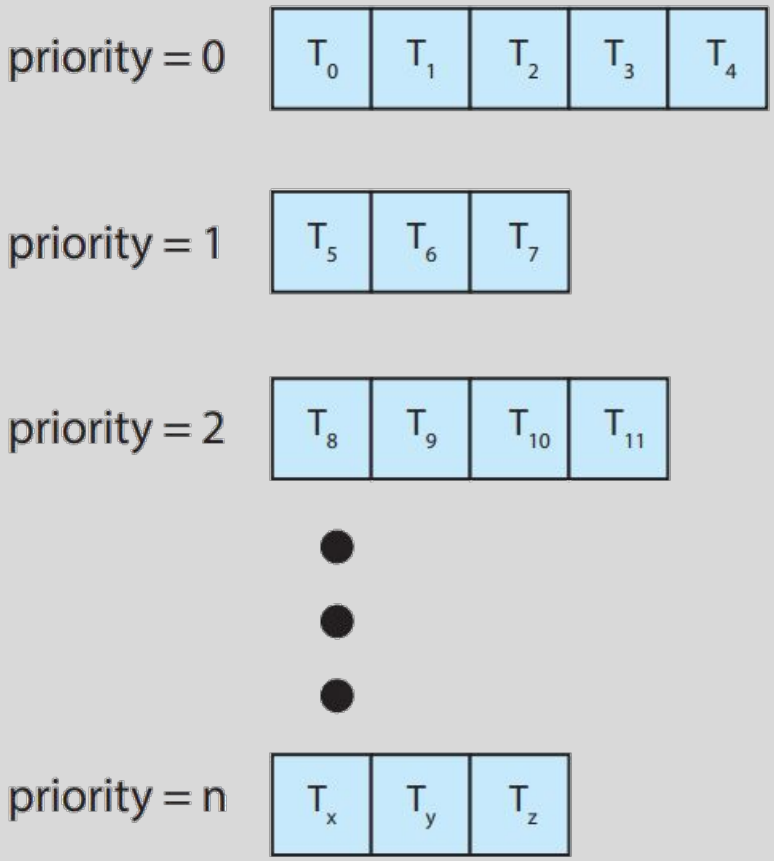
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Diagrama de Gantt



COLAS MULTINIVEL

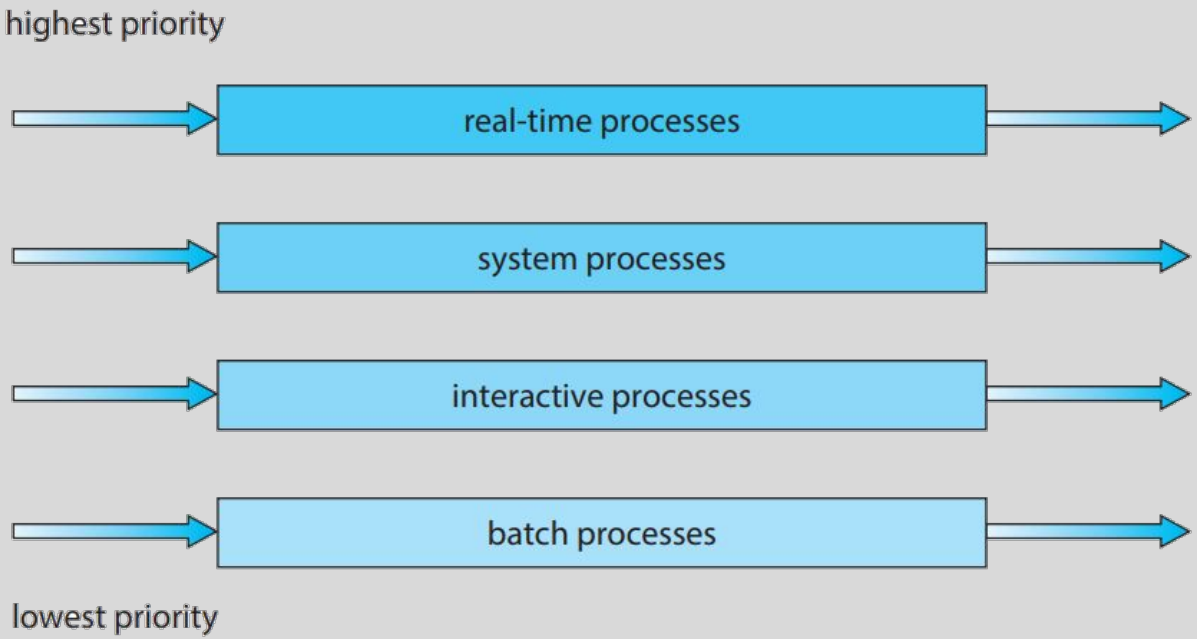
En la práctica, se implementan colas separadas para cada prioridad, y la planificación con prioridad simplemente planifica el proceso en la cola de mayor prioridad. Con este enfoque, si hay varios procesos en la cola de mayor prioridad, se ejecutan en orden RR. En su forma más generalizada, se asigna una prioridad estáticamente a cada proceso, y este permanece en la misma cola durante su tiempo de ejecución.



Separate queues for each priority.

COLAS MULTINIVEL

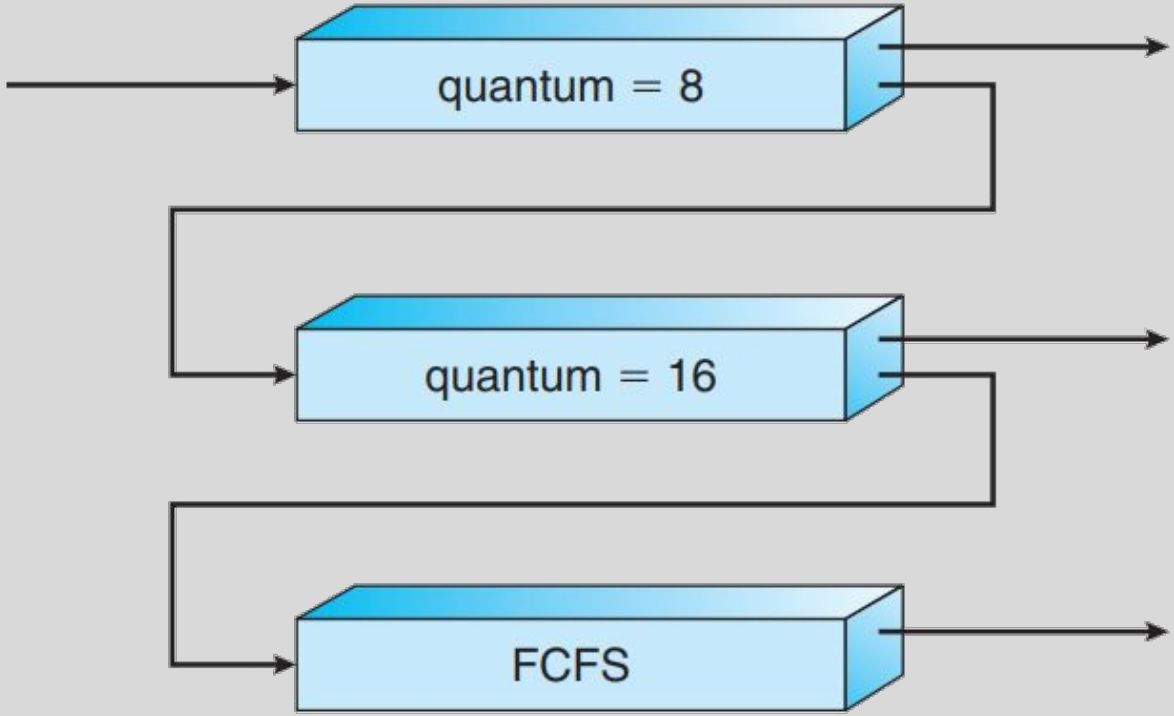
Este algoritmo, también puede implementarse basándose en los diferentes tipos de procesos.



Multilevel queue scheduling.

COLAS MULTINIVEL CON RETROALIMENTACION

En el algoritmo de planificación de colas multinivel, los procesos se asignan a una cola al entrar en el sistema de manera permanente. En cambio, el algoritmo de planificación de colas multinivel con retroalimentación, se permite que un proceso se mueva entre colas. La idea es separar los procesos según las características de sus ráfagas de CPU, si un proceso consume demasiado tiempo de CPU, se moverá a una cola de menor prioridad.



Multilevel feedback queues.

PLANIFICACIÓN DE HILOS

PLANIFICACIÓN DE HILOS

En la mayoría de los sistemas operativos actuales, son los hilos a nivel de *kernel*, no los procesos, los que planifica el sistema operativo. Los hilos a nivel de usuario son gestionados por una librería de hilos, y el *kernel* no tiene conocimiento de ellos. Para ejecutarse en una CPU, los hilos a nivel de usuario deben asignarse a un hilo a nivel de *kernel* asociado, generalmente de manera indirecta al utilizar un proceso ligero (LWP).

Una distinción entre los hilos a nivel de usuario y los de *kernel* radica en su planificación. En sistemas que implementan los modelos muchos a uno y muchos a muchos, la librería de hilos planifica los hilos a nivel de usuario para que se ejecuten en un LWP disponible. Este esquema se conoce como alcance del contenido del proceso (*Process content scope* (PCS)), ya que la competencia por la CPU se produce entre los hilos que pertenecen al mismo proceso.

En cambio, para decidir qué hilo a nivel de *kernel* se planifica en una CPU, el *kernel* utiliza el alcance del contenido del sistema (*System content scope* (SCS)). La competencia por la CPU con la planificación SCS se produce entre todos los hilos del sistema. Los sistemas que utilizan el modelo uno a uno, como Windows y Linux, planifican hilos utilizando SCS.

PLANIFICACIÓN MULTIPROCESADOR

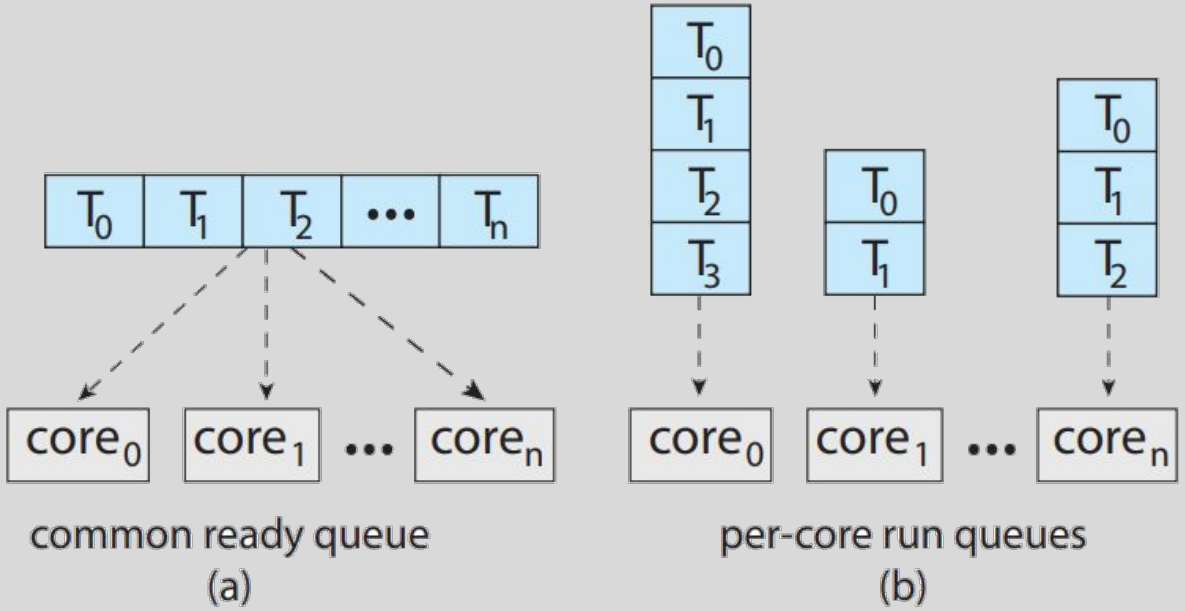
ENFOQUE - *MASTER-SLAVE*

Un enfoque consiste en que todas las decisiones de planificación, el procesamiento de E/S y otras actividades del sistema sean gestionadas por un solo procesador, denominado procesador maestro. Los demás procesadores ejecutan únicamente código de usuario. Este multiprocesamiento asimétrico es simple porque solo un núcleo accede a las estructuras de datos del sistema, lo que reduce la necesidad de compartir datos. La desventaja de este enfoque es que el procesador maestro se convierte en un posible cuello de botella que puede reducir el rendimiento general del sistema.

ENFOQUE ESTANDAR

El enfoque estándar para la compatibilidad con multiprocesadores es el multiprocesamiento simétrico (SMP), donde cada procesador se auto planifica. Cabe destacar que esto ofrece dos posibles estrategias para organizar los hilos elegibles para la planificación:

1. Todos los hilos pueden estar en una cola de hilos listos común.
2. Cada procesador puede tener su propia cola privada de hilos.



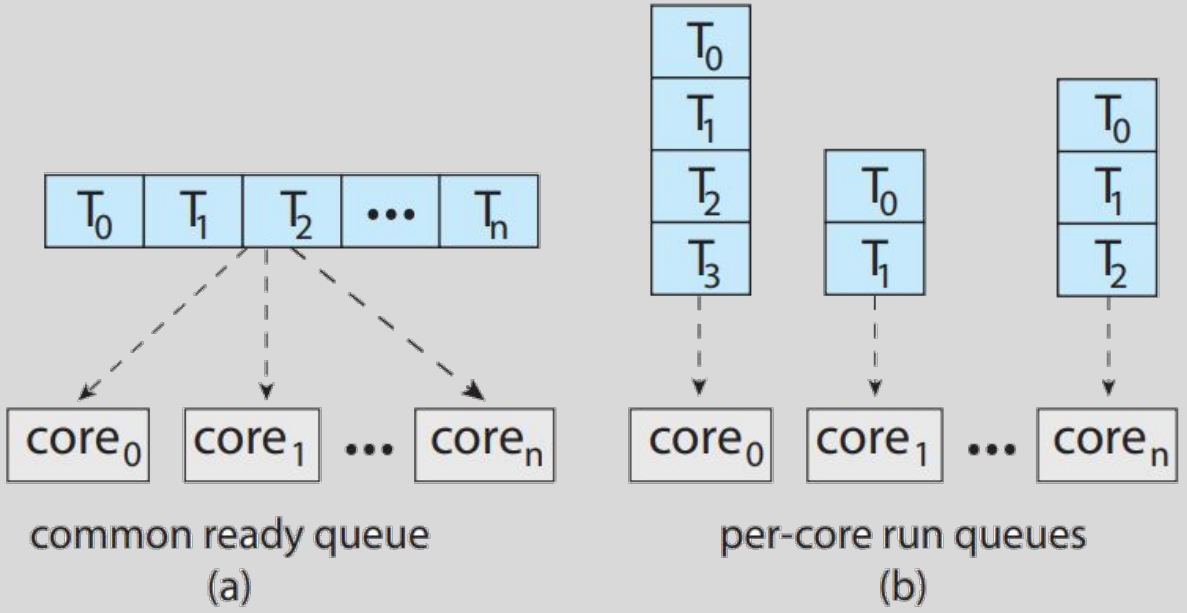
Organization of ready queues.

ENFOQUE ESTANDAR

En la primera opción, existe una posible condición de carrera en la cola de listos compartida, por lo tanto, debe asegurarse que dos procesadores independientes no planifiquen el mismo hilo y de que no se pierdan hilos de la cola de espera.

La segunda opción permite que cada procesador planifique hilos, y así evita los posibles problemas de rendimiento asociados a una cola compartida. Por lo tanto, es el enfoque más común en sistemas compatibles con SMP.

Prácticamente todos los sistemas operativos actuales son compatibles con SMP.

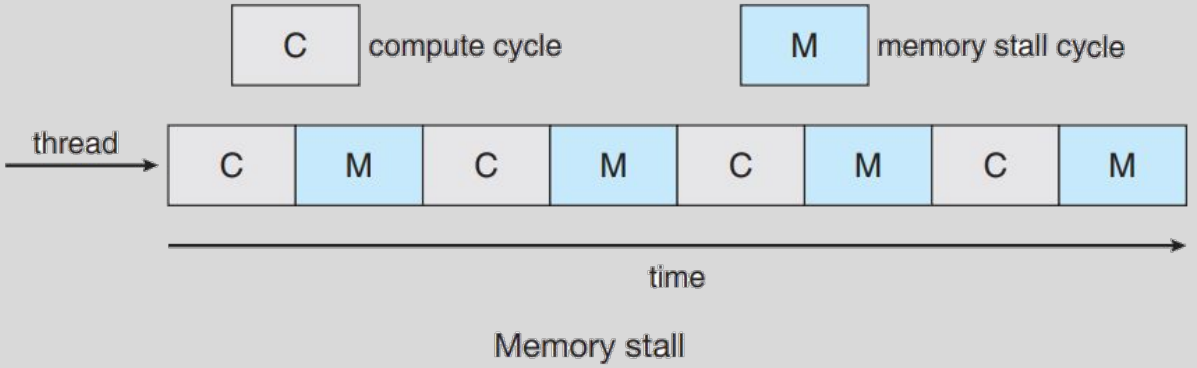


Organization of ready queues.

PROCESADORES MULTICORE

MEMORY STALL

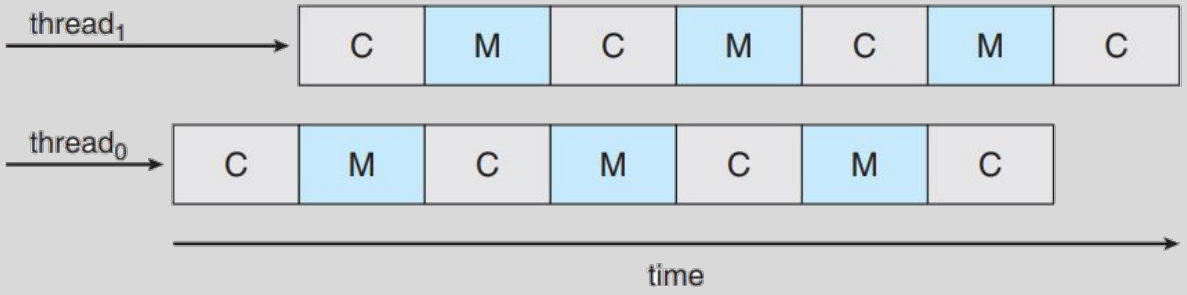
Cuando un procesador accede a memoria, pasa un tiempo considerable esperando a que los datos estén disponibles. Esta situación se conoce como *memory stall* y se produce principalmente porque los procesadores actuales operan a velocidades mucho mayores que la memoria. En este escenario, el procesador puede pasar hasta el 50 % de su tiempo esperando a que los datos estén disponibles en la memoria.



Memory stall

MULTITHREADED PROCESSING

Para solucionar esto, muchos diseños de hardware han implementado núcleos de procesamiento multihilo, donde se asignan dos (o más) hilos de hardware a cada núcleo. Así, si un hilo de hardware se bloquea mientras espera memoria, el núcleo puede cambiar a otro hilo.

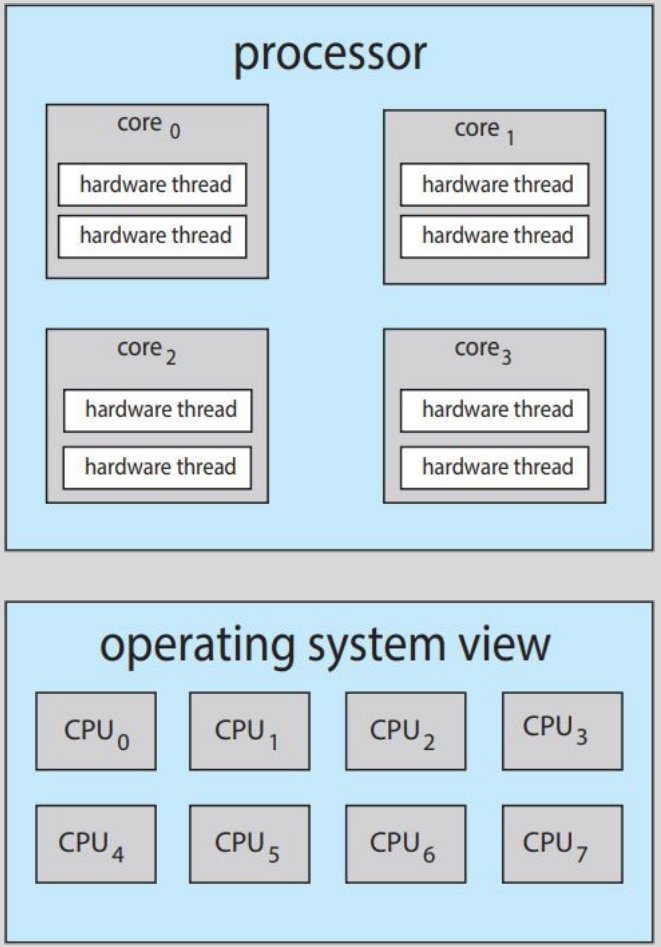


Multithreaded multicore system

MULTITHREADED PROCESSING

Desde la perspectiva del sistema operativo, cada hilo de hardware se presenta como una CPU lógica disponible para ejecutar un hilo de software. Por ejemplo, si un procesador contiene cuatro núcleos de procesamiento, cada uno con dos hilos de hardware, desde el punto de vista del sistema operativo, hay ocho CPU lógicas disponibles.

Los procesadores Intel utilizan el término *hyper-threading* para describir la asignación de múltiples hilos de hardware a un único núcleo de procesamiento.

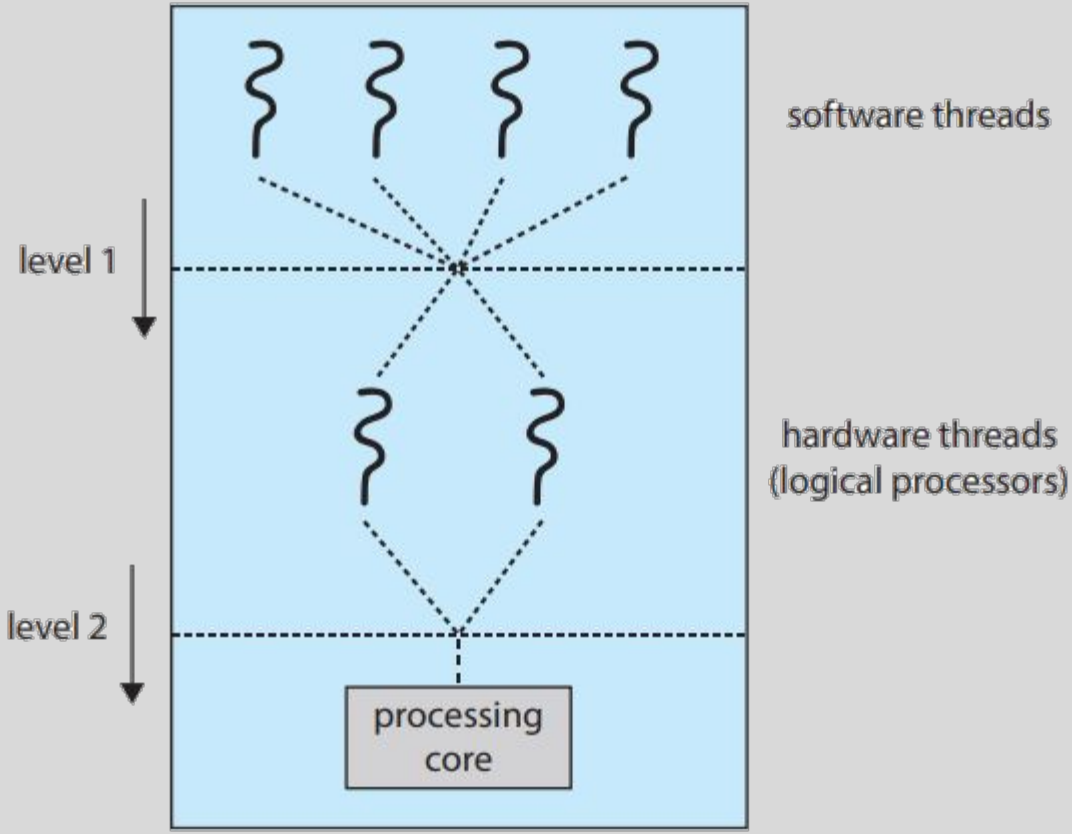


Chip multithreading

MULTITHREADED PROCESSING

Es importante tener en cuenta que los recursos del núcleo físico (cachés, pipelines) deben compartirse entre sus hilos de hardware, así, un núcleo de procesamiento solo puede ejecutar un hilo de hardware a la vez. Por esto, un procesador multihilo y multinúcleo requiere dos niveles de planificación.

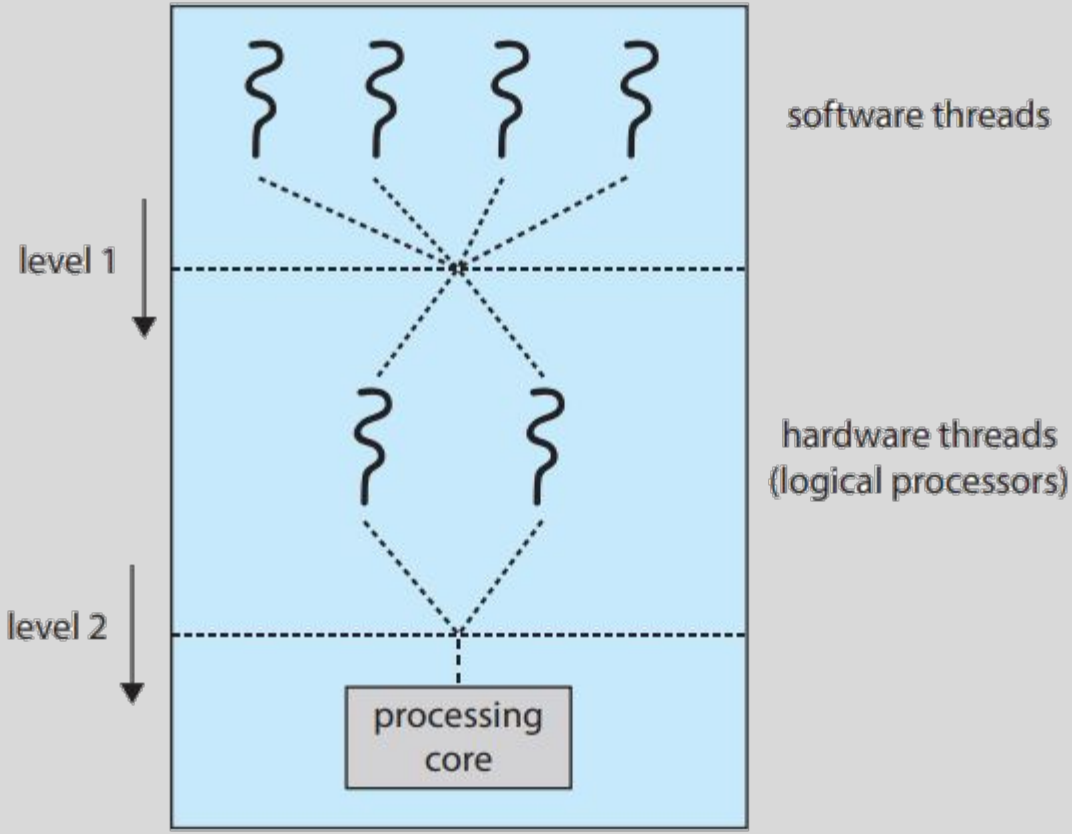
En un primer nivel se encuentran las decisiones de planificación que debe tomar el sistema operativo al elegir qué hilo de software ejecutar en cada hilo de hardware (CPU lógica). Donde el sistema operativo puede elegir cualquier algoritmo de planificación mencionados.



Two levels of scheduling

MULTITHREADED PROCESSING

Un segundo nivel de planificación especifica cómo cada núcleo decide qué hilo de hardware ejecutar. Existen varias estrategias para esta situación. Un enfoque tradicional consiste en utilizar un algoritmo simple de round-robin para planificar un hilo de hardware en el *core* de procesamiento.



Two levels of scheduling

Muchas Gracias

Jeremías Fassi

Javier E. Kinter