Sistemas Operativos

Curso 2013

Procesos

Agenda

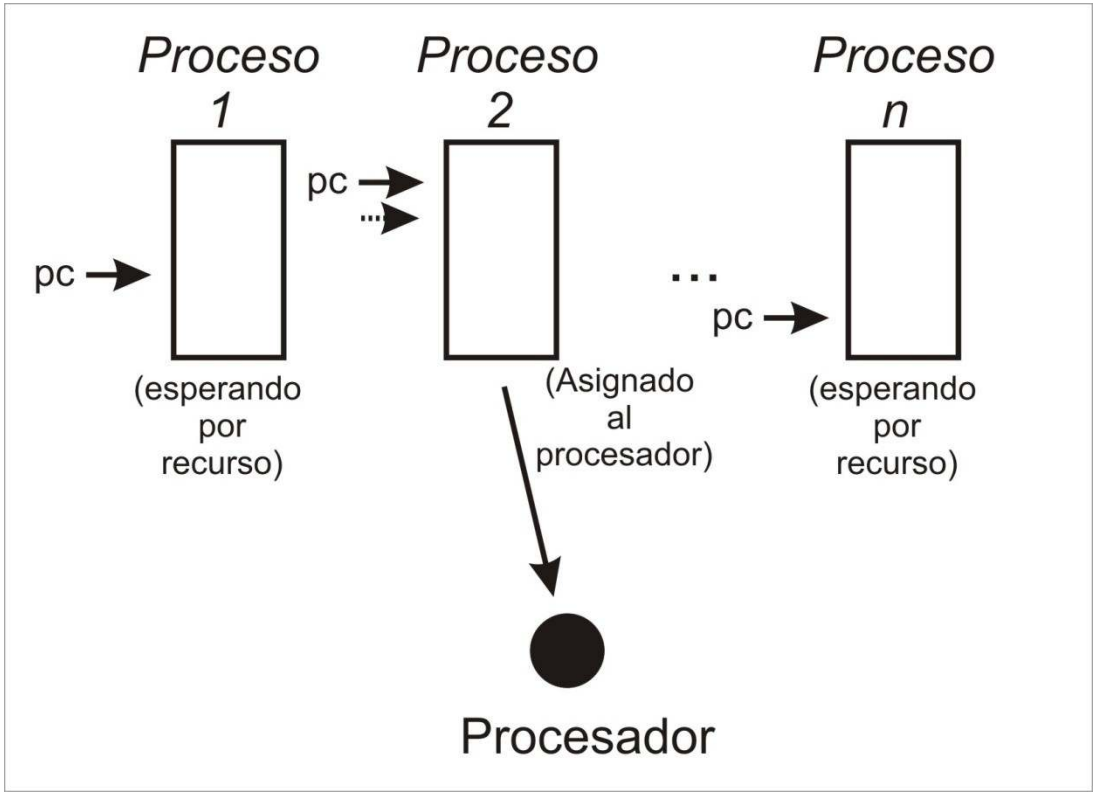
- Proceso.
  - Definición de proceso.
  - Contador de programa.
  - Memoria de los procesos.
- Estados de los procesos.
  - Transiciones entre los estados.
- Bloque descriptor de proceso (*PCB*).
- Creación de procesos.
- Listas y colas de procesos.
- Cambio de contexto (*context switch*).
- Hilos (*Threads*).
  - *Threads* a nivel de usuario.
  - *Threads* a nivel de núcleo del sistema.
  - Modelos de *threads*.

Definición de Proceso

- El principal concepto en cualquier sistema operativo es el de proceso.
- Un proceso es un programa en ejecución, incluyendo el valor del *program counter*, los registros y las variables.
- Conceptualmente, cada proceso tiene un hilo *(thread)* de ejecución que es visto como un CPU virtual.
- El recurso procesador es alternado entre los diferentes procesos que existan en el sistema, dando la idea de que ejecutan en paralelo (multiprogramación)

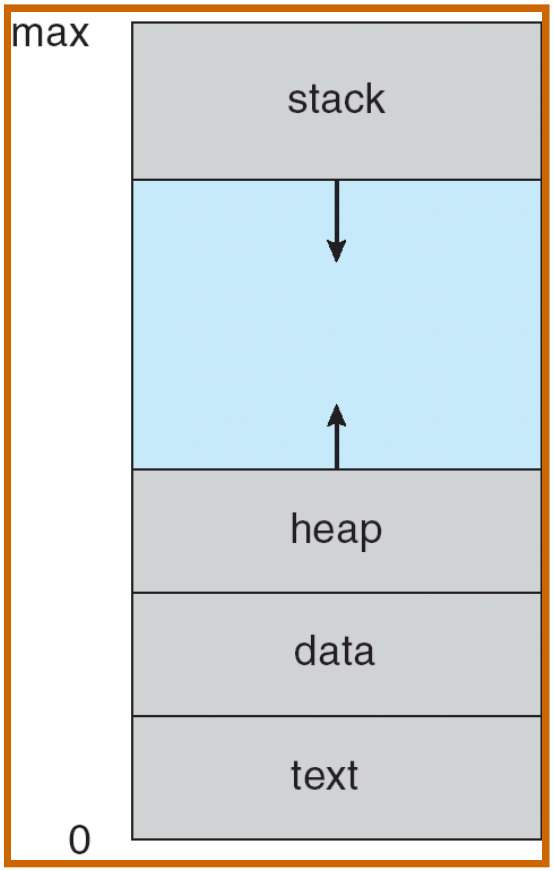
Contador de programa

Cada proceso tiene su *program counter,* y avanza cuando el proceso tiene asignado el recurso procesador. A su vez, a cada proceso se le asigna un número que lo identifica entre los demás: identificador de proceso (*process id*)



Memoria de los procesos

- Un proceso en memoria se constituye de varias secciones:
  - **Código** (*text*): Instrucciones del proceso.
  - **Datos** (*data*): Variables globales del proceso.
  - **Memoria dinánica** (*heap*): Memoria dinámica que genera el proceso.
  - **Pila** (*stack*): Utilizado para preservar el estado en la invocación anidada de procedimientos y funciones.

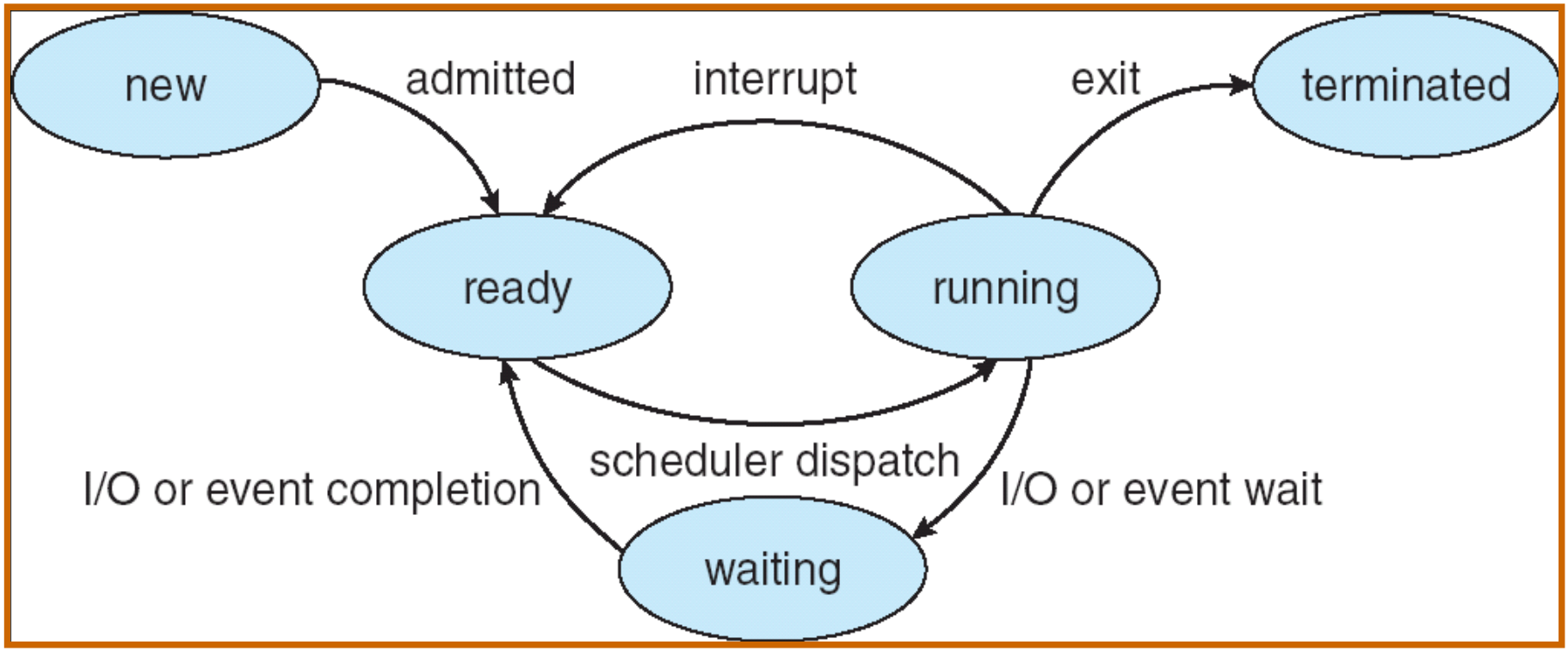


Estados de los procesos

- El estado de un proceso es definido por la actividad corriente en que se encuentra.
- Los estados de un proceso son:
  - **Nuevo** (*new*): Cuando el proceso es creado.
  - **Ejecutando** (*running*): El proceso tiene asignado un procesador y está ejecutando sus instrucciones.
  - **Bloqueado** (*waiting*): El proceso está esperando por un evento (que se complete un pedido de E/S o una señal).
  - **Listo** (*ready*): El proceso está listo para ejecutar, solo necesita del recurso procesador.
  - **Finalizado** (*terminated*): El proceso finalizó su ejecución.

Estados de los procesos

- Diagrama de estados y transiciones de los procesos.



Transiciones entre estados

- Nuevo $\Rightarrow$Listo
  - Al crearse un proceso pasa inmediatamente al estado listo.

- Listo $\Rightarrow$Ejecutando
  - En el estado de listo, el proceso solo espera para que se le asigne un procesador para ejecutar (tener en cuenta que puede existir más de un procesador en el sistema). Al liberarse un procesador el planificador (*scheduler*) selecciona el próximo proceso, según algún criterio definido, a ejecutar.

Transiciones entre estados

- Ejecutando $\Rightarrow$Listo
  - Ante una interrupción que se genere, el proceso puede perder el recurso procesador y pasar al estado de listo. El planificador será el encargado de seleccionar el próximo proceso a ejecutar.

- Ejecutando $\Rightarrow$Bloqueado
  - A medida que el proceso ejecuta instrucciones realiza pedidos en distintos componentes (ej.: genera un pedido de E/S). Teniendo en cuenta que el pedido puede demorar y, además, si está en un sistema multiprogramado, el proceso es puesto en una cola de espera hasta que se complete su pedido. De esta forma, se logra utilizar en forma más eficiente el procesador.

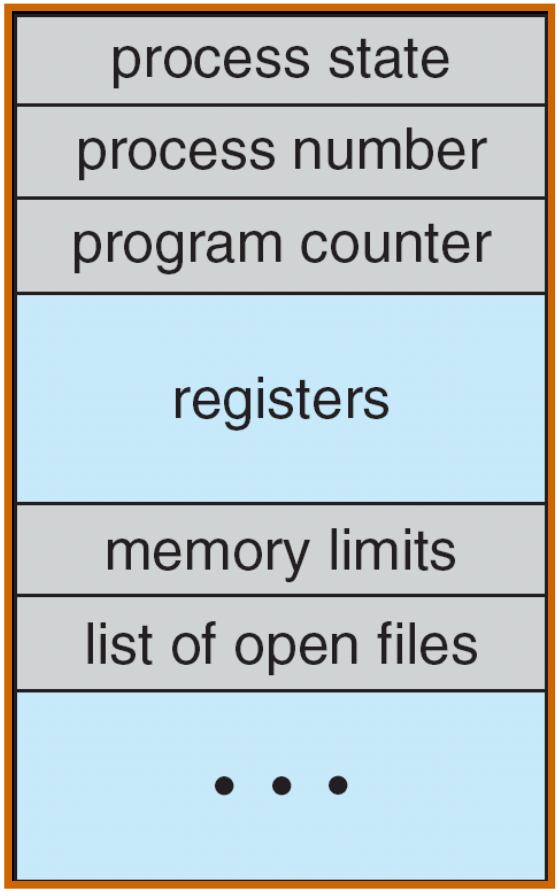
Transiciones entre estados

- Bloqueado $\Rightarrow$Listo
  - Una vez que ocurre el evento que el proceso estaba esperando en la cola de espera, el proceso es puesto nuevamente en la cola de procesos listos.

- Ejecutando $\Rightarrow$Terminado
  - Cuando el proceso ejecuta sus última instrucción pasa al estado terminado. El sistema libera las estructuras que representan al proceso.

Bloque descriptor de proceso

- El proceso es representado, a nivel del sistema operativo, a través del bloque descriptor de proceso (*Process Control Block*)



Bloque descriptor de proceso

- Todo proceso se describe mediante su estado, nombre, recursos asignados, información contable, etc.
- Para ello se utiliza una estructura de datos que será el operando de las operaciones sobre procesos, recursos y del planificador (*scheduler*).
- Los campos de esta estructura son:
  - **Estado CPU**: El contenido de esta estructura estará indefinido toda vez que el proceso está en estado *ejecutando* (puesto que estará almacenado en la CPU indicada por procesador). Registro de flags.
  - **Procesador**: [1..#CPU]: Contendrá el número de CPU que está ejecutando al proceso (si está en estado *ejecutando*), sino su valor es indefinido.
  - **Memoria**: Describe el espacio virtual y/o real de direccionamiento según la arquitectura del sistema. Contendrá las reglas de protección de memoria así como cual es compartida, etc..
  - **Estado del proceso**: *ejecutando*, *listo, bloqueado,* etc.

Bloque descriptor de proceso

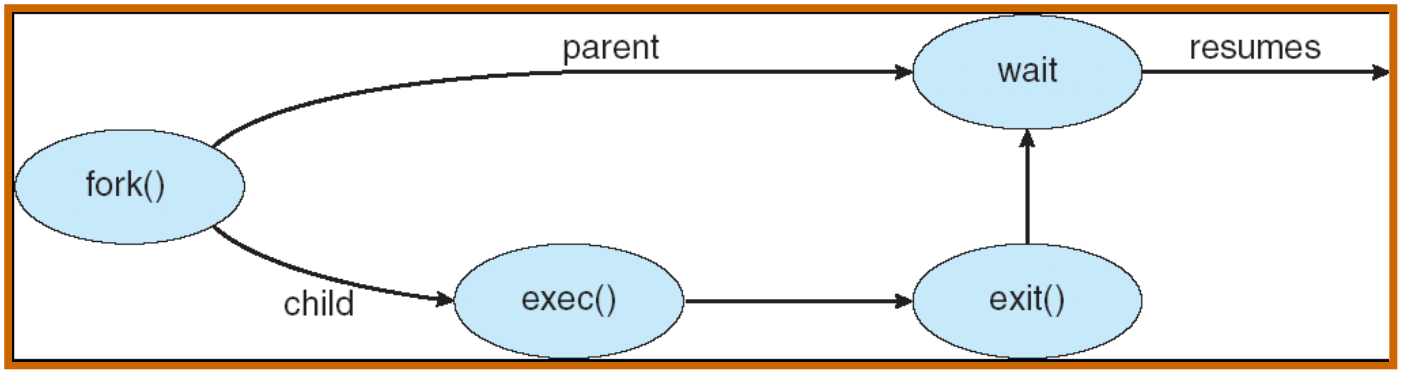
- **Recursos**: Recursos de *software* (archivos, semáforos, etc.) y *hardware* (dispositivos, etc.).
- **Planificación**: Tipo de planificador.
- **Prioridad**: Podrá incluir una prioridad externa de largo aliento, o en su defecto una prioridad interna dinámica de alcance reducido.
- **Contabilización**: Información contable como ser cantidad de E/S, fallos de páginas (*page faults*), consumo de procesador, memoria utilizada, etc.
- **Ancestro**: Indica quién creó este proceso.
- **Descendientes**: Lista de punteros a PCBs de los hijos de este proceso.

Creación de procesos

- Los procesos de un sistema son creados a partir de otro proceso.
- Al creador se le denomina padre y al nuevo proceso hijo. Esto genera una jerarquía de procesos en el sistema.
- En el diseño del sistema operativo se debe decidir, en el momento de creación de un nuevo proceso, cuales recursos compartirán el proceso padre e hijo. Las opciones son que compartan todo, algo o nada.
- También se debe determinar que sucede con los hijos cuando muere el padre. Pueden morir también o cambiar de padre.
- Una vez creado el nuevo proceso tendrán un hilo (*program counter*) de ejecución propio. El sistema genera un nuevo PCB para el proceso creado.

Creación de procesos

- Ej.: UNIX
  - UNIX provee el *system call* ***fork*** para la creación de un nuevo proceso.
  - La invocación a esta función le retorna al padre el número de *process id* del hijo recién creado y al hijo el valor 0. El hijo comienza su ejecución en el retorno del *fork*.
  - Además, se provee del *system call* ***exec*** que reemplaza el espacio de memoria del proceso por uno nuevo

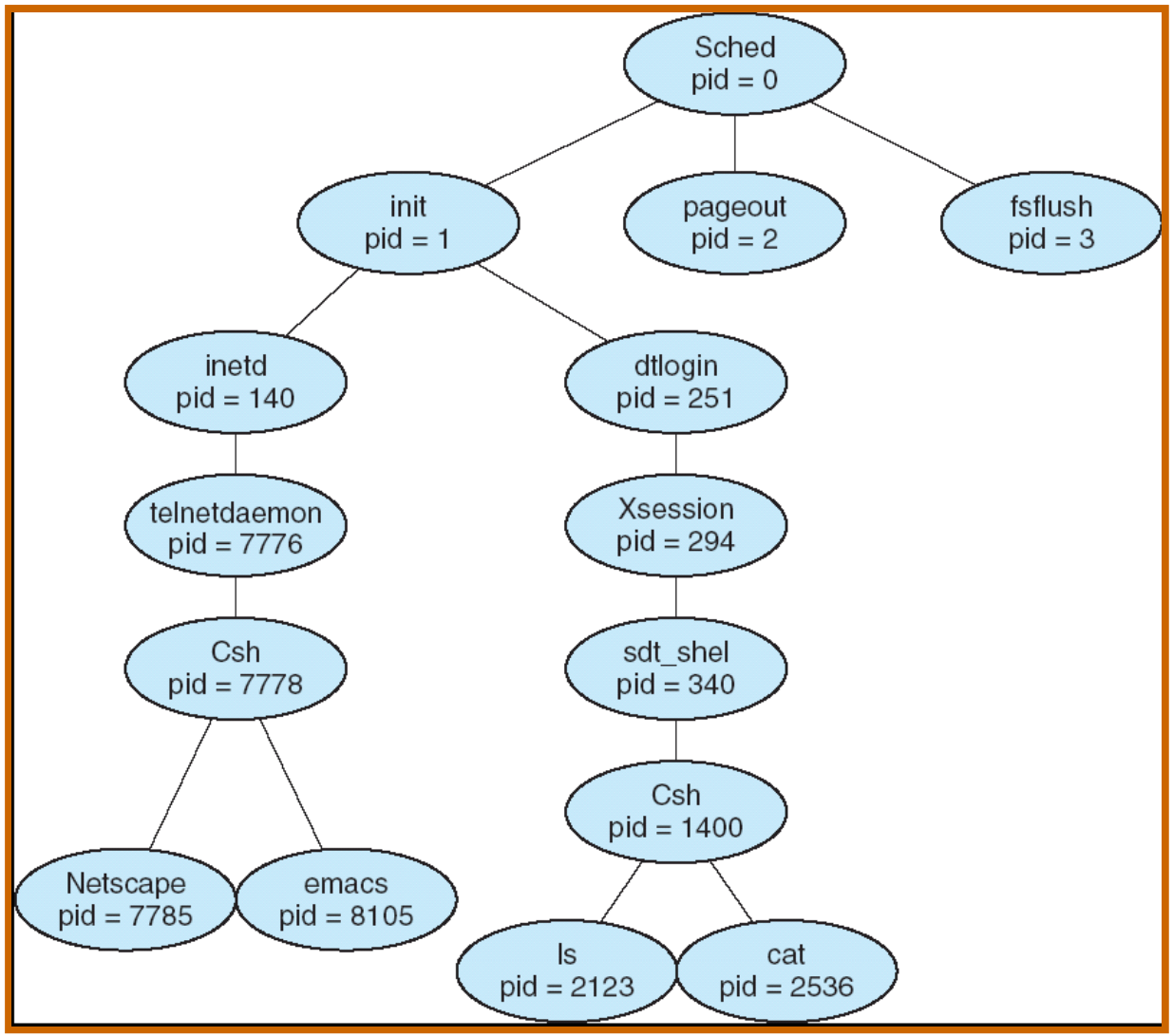


Creación de procesos

```
int main() {
    pid_t  pid;

    /* crea un nuevo proceso */
    pid = fork();
    if (pid < 0) { /* error */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    if (pid == 0) /* proceso hijo */
        execlp("/bin/ls", "ls", NULL);
    else { /* padre */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

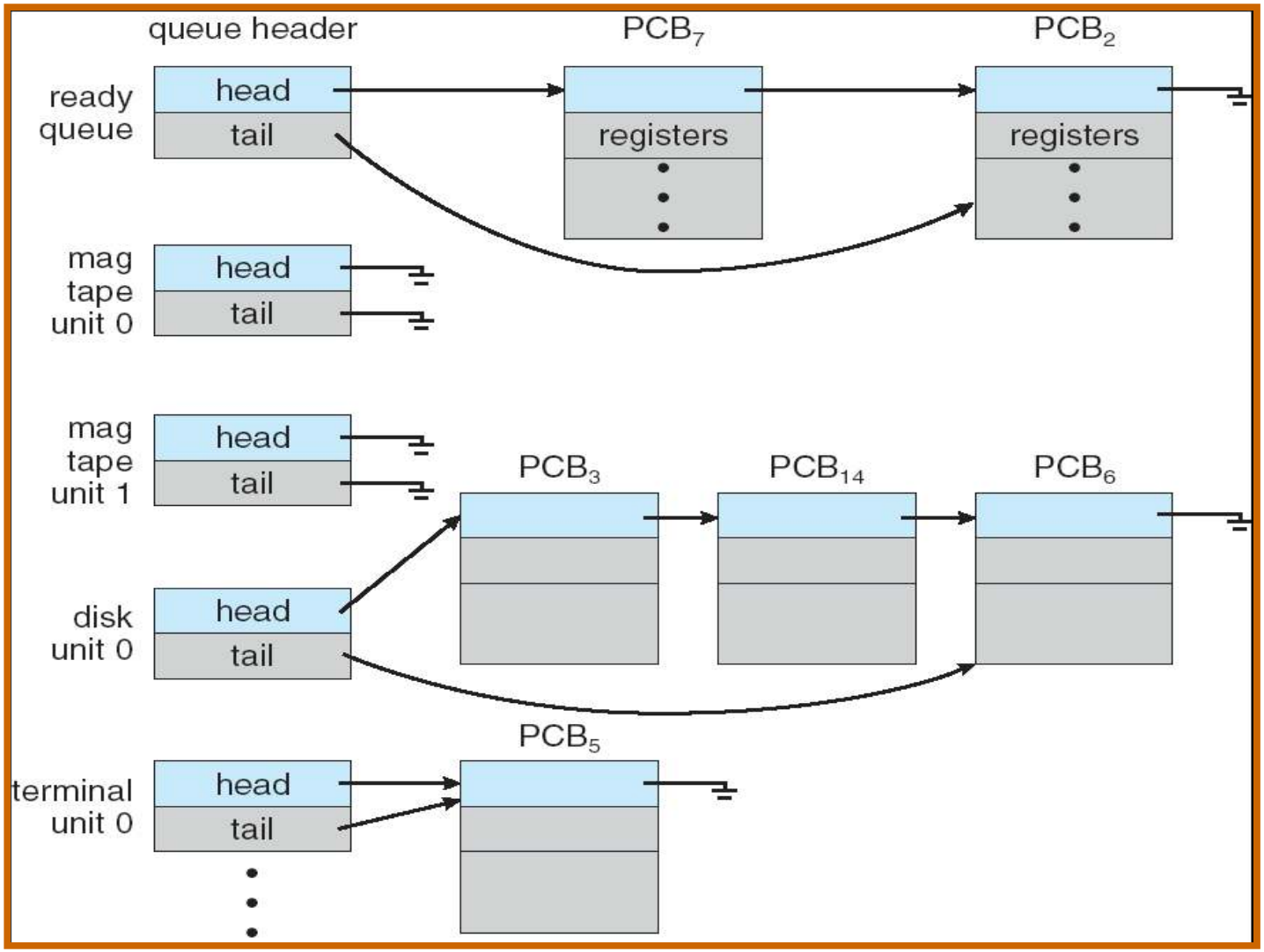
Creación de procesos



Listas y colas de procesos

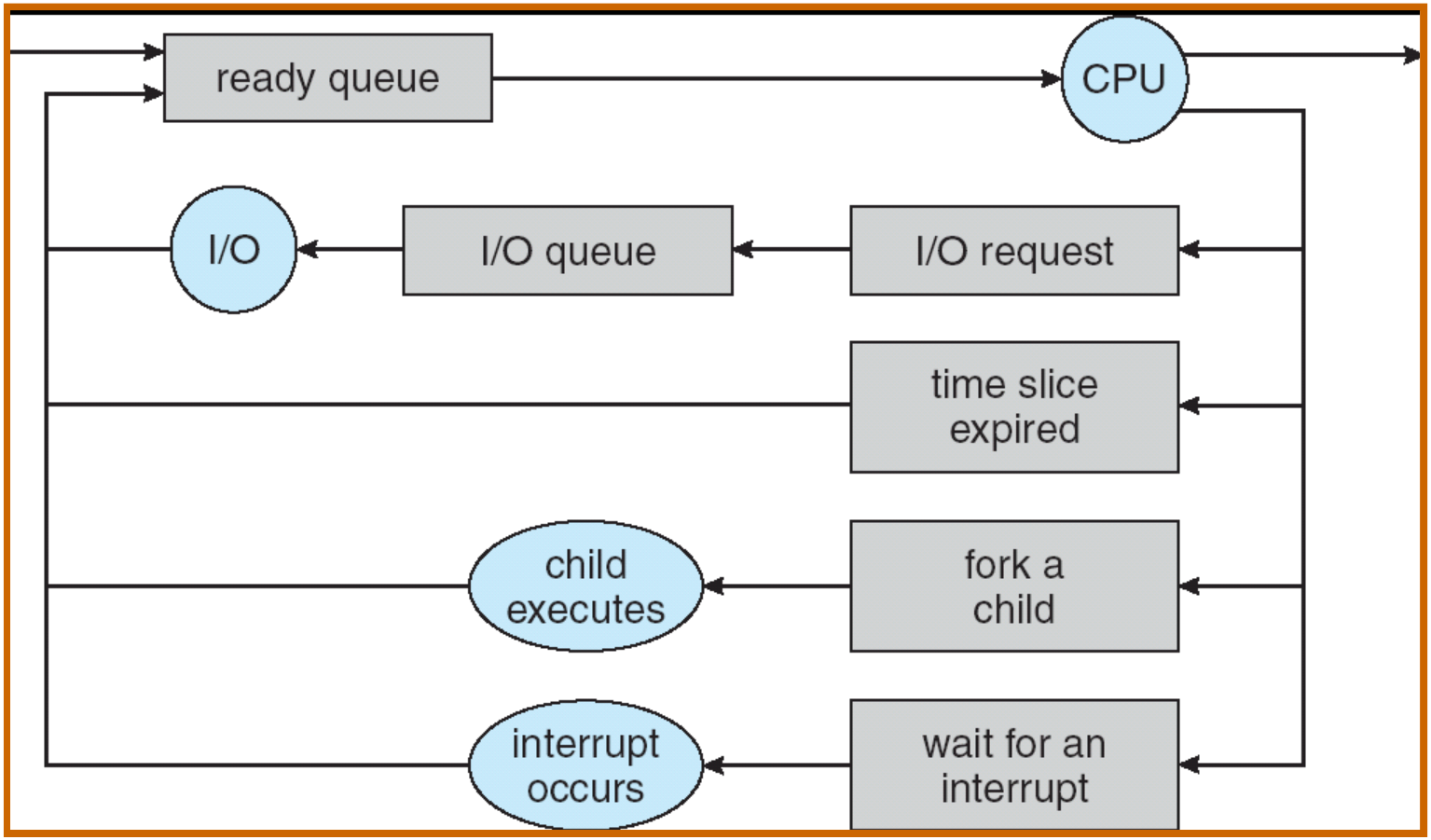
- Los procesos, en los distintos estados que tienen, son agrupados en listas o colas:
  - **Lista de procesos del sistema** (*job queue*): En esta lista están todos los procesos del sistema. Al crearse un nuevo proceso se agrega el PCB a esta lista. Cuando el proceso termina su ejecución es borrado.
  - **Cola de procesos listos** (*ready queue*): Esta cola se compondrá de los procesos que estén en estado listo. La estructura de esta cola dependerá de la estrategia de planificación utilizada.
  - **Cola de espera de dispositivos** (*device queue*): Los procesos que esperan por un dispositivo de E/S particular son agrupados en una lista específica al dispositivo. Cada dispositivo de E/S tendrá su cola de espera.

Listas y colas de procesos



Listas y colas de procesos

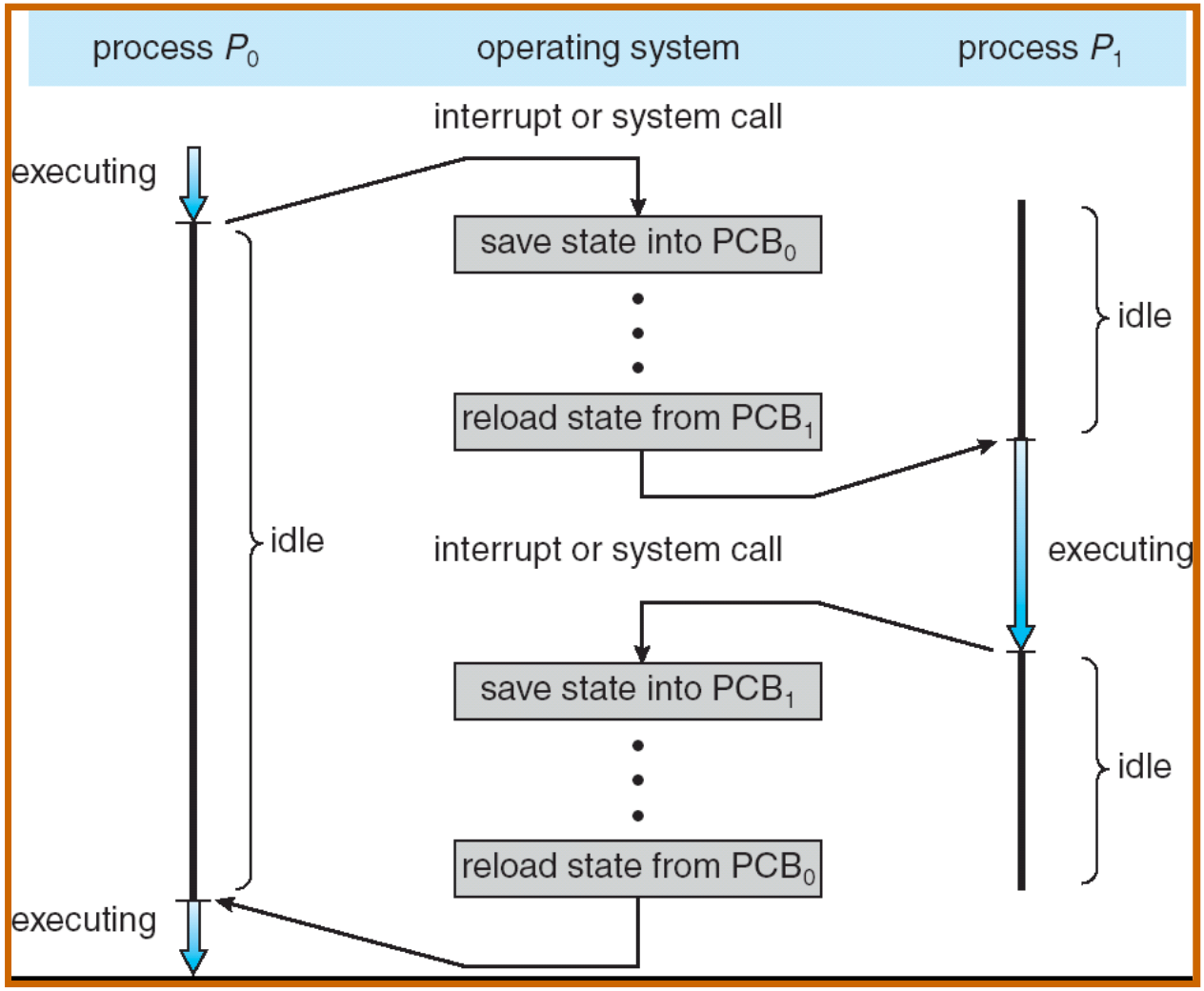
- Diagrama de transición de un proceso entre las colas del sistema.



Cambio de contexto

- A la tarea de cambiar un proceso por otro en el procesador se le denomina **cambio de contexto** (*context switch*).
- El cambio de contextos entre procesos implica las siguientes tareas:
  - Salvar el estado del proceso (registros, información de punteros de memoria) que está ejecutando en su PCB.
  - Cambiar el estado del proceso que estaba ejecutando al que corresponda.
  - Cargar el estado del proceso asignado a la CPU a partir de su PCB.
  - Cambiar el estado del proceso nuevo a *ejecutando*.

Cambio de contexto



Cooperación entre procesos

- Procesos concurrentes pueden ejecutar en un entorno aislado (se debe asegurar la ausencia de interferencias) o, eventualmente, podrán interactuar cooperando en pos de un objetivo común compartiendo objetos comunes.
- Es necesario que el sistema operativo brinde unas herramientas específicas para la comunicación y sincronización entre los procesos (*Inter Process Communication – IPC*).
- IPC es una herramienta que permite a los procesos comunicarse y sincronizarse sin compartir el espacio de direccionamiento en memoria.
- Hay dos enfoques fundamentales:
  - Memoria compartida
  - Pasaje de mensajes

Threads

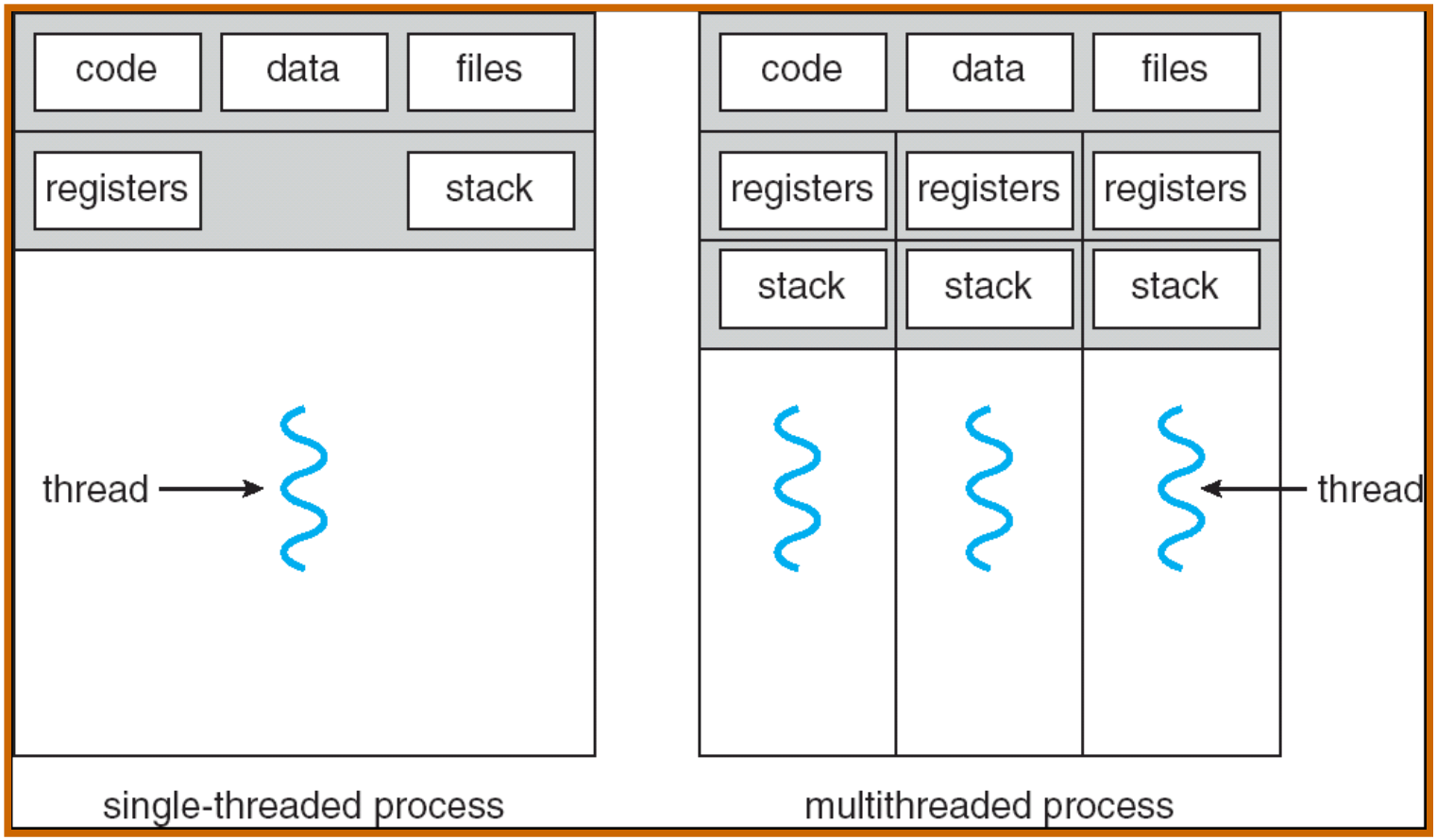
- Hay aplicaciones donde es necesario utilizar procesos que compartan recursos en forma concurrente.
- IPC brindan una alternativa a nivel de sistema operativo.
- Los sistemas operativos modernos están proporcionando servicios para crear más de un hilo (*thread*) de ejecución (control) en un proceso.
- Con las nuevas tecnologías multi-core esto se hace algo necesario para poder sacar mayor provecho al recurso de procesamiento.
- De esta forma, se tiene más de un hilo de ejecución en el mismo espacio de direccionamiento.

Threads

- Un *Thread* (Hilo) es una unidad básica de utilización de la CPU consistente en un juego de registros y un espacio de pila. Es también conocido como proceso ligero.
- Cada *thread* contendrá su propio *program counter*, un conjunto de registros, un espacio para el *stack* y su prioridad.
- Comparten el código, los datos y los recursos con sus hebras (*thread*) pares.
- Una tarea (o proceso pesado) está formado ahora por uno o varios *threads*.
- Un *thread* puede pertenecer a una sola tarea.

Threads

- Todos los recursos, sección de código y datos son compartidos por los distintos threads de un mismo proceso.



Ventajas del uso de *threads*

- **Repuesta**: Desarrollar una aplicación con varios hilos de control (*threads*) permite tener un mejor tiempo de respuesta.
- **Compartir recursos**: Los *threads* de un proceso comparten la memoria y los recursos que utilizan. A diferencia de IPC, no es necesario acceder al *kernel* para comunicar o sincronizar los hilos de ejecución.
- **Economía**: Es más fácil un cambio de contexto entre *threads* ya que no es necesario cambiar el espacio de direccionamiento. A su vez, es más "liviano" para el sistema operativo crear un *thread* que crear un proceso nuevo.
- **Utilización de arquitecturas con multiprocesadores**: Disponer de una arquitectura con más de un procesador permite que los *threads* de un mismo proceso ejecuten en forma paralela.

Desventaja del uso de *threads*

- **Dificulta la programación**: Al compartir todo el espacio de direccionamiento un *thread* mal programado puede romper el funcionamiento del resto de los *threads*.

Threads

- Los *threads* pueden ser implementados tanto a nivel de usuario como a nivel de sistemas operativo:
  - **Hilos a nivel de usuario** (*user threads*): Son implementados en alguna librería de usuario. La librería deberá proveer soporte para crear, planificar y administrar los *threads* sin soporte del sistema operativo. El sistema operativo solo reconoce un hilo de ejecución en el proceso.
  - **Hilos a nivel del núcleo** (*kernel threads*): El sistema es quien provee la creación, planificación y administración de los *threads*. El sistema reconoce tantos hilos de ejecución como threads se hayan creado

Threads

- Ventajas de *user threads* sobre *kernel threads*:
  - **Desarrollo de aplicaciones en sistemas sin soporte a hilo**: Se pueden aprovechar todos los beneficios de programar orientado utilizando *threads*. Además se puede portar la aplicación a un sistema operativo que carezca de la noción de varios hilos de ejecución.
  - **Cambio de contexto**: El cambio de contexto entre *threads* de usuario es más simple ya que no consume el *overhead* que tendría en el sistema operativo (*system call*).
  - **Planificación independiente**: Se puede crear una nueva estrategia de planificación diferente a la que tenga el sistema operativo.

Threads

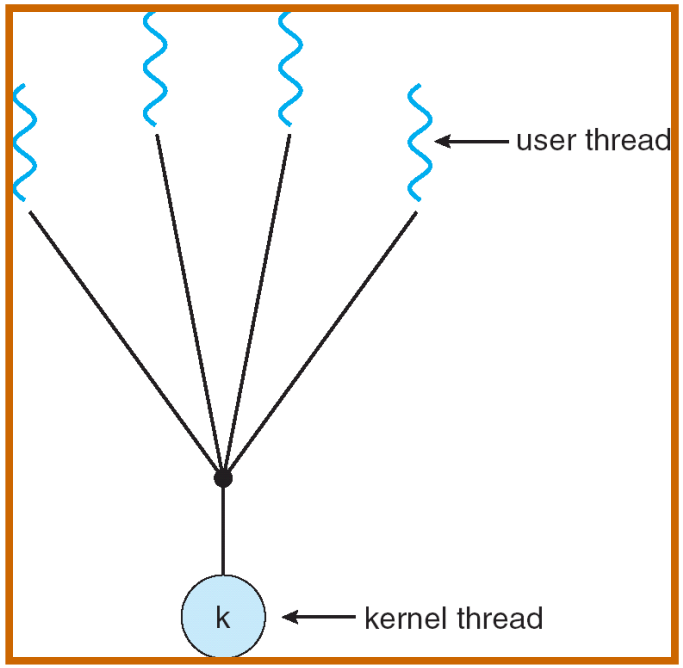
- Ventajas de *kernel threads* sobre *user threads*:
  - **Mejor aprovechamiento de un sistema multiprocesador**: el sistema operativo puede asignar *threads* del mismo proceso en distintos procesadores. De esta forma, un proceso puede estar consumiendo más de un recurso procesador a la vez.
  - **Ejecución independiente**: Al ser independientes los hilos de ejecución, si un *thread* se bloquea (debido a p.ej. una operación de E/S) los demás *threads* pueden seguir ejecutando.

Threads

- La mayoría de los sistemas proveen *threads* tanto a nivel de usuario como de sistema operativo.
- De esta forma surgen varios modelos:
  - **Mx1** (*Many-To-One*): Varios *threads* de a nivel de usuario a un único *thread* a nivel de sistema.
  - **1x1** (O*ne-to-One*): Cada *thread* de usuario se corresponde con un *thread* a nivel del núcleo (kernel thread).
  - **MxN** (*Many-To-Many*): Varios *threads* a nivel de usuario se corresponde con varios *threads* a nivel del núcleo.

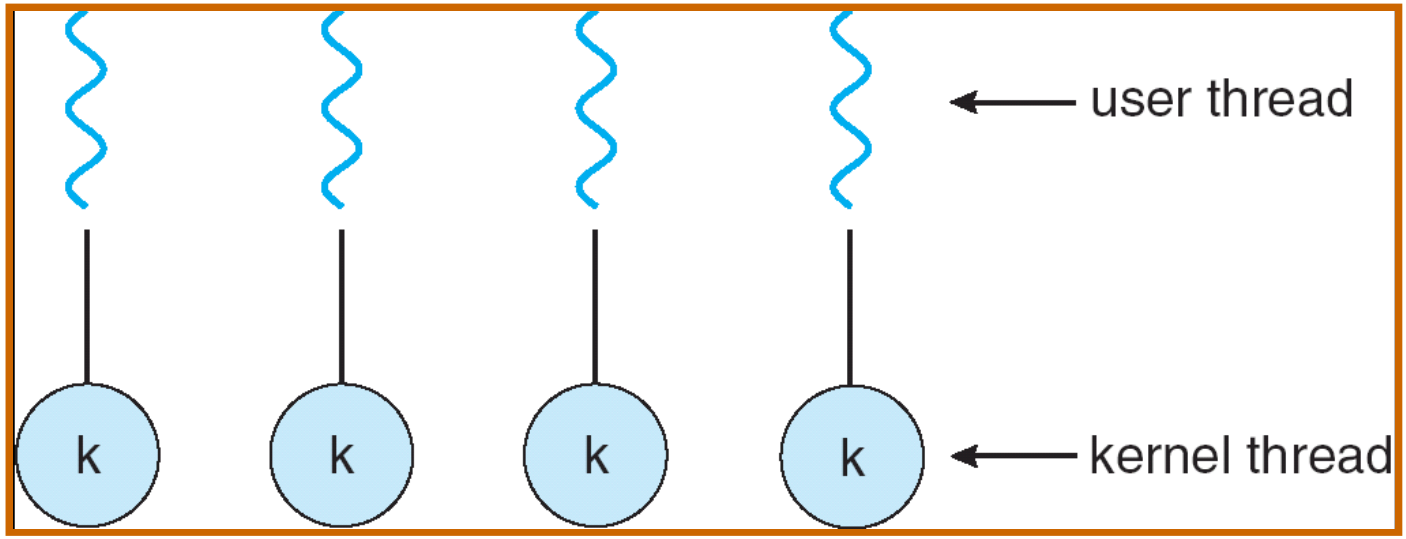
Mx1 *(Many-to-One)*

- Este caso se corresponde al de tener los *threads* implementados a nivel de usuario.
- El sistema solo reconoce un *thread* de control para el proceso.
- Los *threads* de usuario ejecutarán cuando estén asignados al *kernel thread* del proceso (tarea llevada a cabo por el planificador a nivel de usuario) y, además, a este le asigne la CPU el planificador del sistema operativo.



1x1 *(One-to-One)*

- Cada *thread* que es creado a nivel de usuario se genera un nuevo *thread* a nivel de sistema que estará asociado mientras exista.
- El sistema reconoce todos los *threads* a nivel de usuario y son planificados independientemente. En este caso no hay planificador a nivel de usuario.



MxN *(Many-to-Many)*

- Cada proceso tiene asignado un conjunto de *kernel threads* independiente de los *threads* a nivel de usuario que el proceso haya creado.
- El planificador a nivel de usuario asigna los *threads* en los *kernel threads*.
- El planificador de sistema solo reconoce los *kernel threads*.