Tema 2 – Procesos y Hebras

- Generalidades sobre Procesos, Hilos y Planificación
- Diseño e implementación de procesos e hilos en Linux
- Planificación de CPU en Linux

Sistemas Operativos
Alejandro J. León Salas, 2020
Lenguajes y Sistemas Informáticos

Objetivos

- Conocer y diferenciar los conceptos de **proceso y hebra.**
- Conocer el diagrama de estados de los procesos y las hebras.
- Saber en qué consiste un **cambio de contexto** y los costes que éste tiene para el sistema operativo.
- Contenido y utilidad de las **estructuras de datos** que el SO utiliza para la gestión de procesos y hebras.
- Conocer la utilidad de la **planificación de procesos** y las estrategias de planificación fundamentales.
- Comparar los distintos algoritmos de planificación de CPU.
- Conocer las operaciones básicas sobre procesos y hebras que pueden realizar los usuarios.
- Conocer la implementación en Linux de procesos y hebras, así como el funcionamiento básico del planificador de Linux.

Bibliografía

- [Sta2005] W. Stallings, Sistemas Operativos. Aspectos Internos y Principios de Diseño (5/e), Prentice Hall, 2005.
- [Car2007] J. Carretero et al., Sistemas Operativos (2ª Edición), McGraw-Hill, 2007.
- [Lov2010] R. Love, *Linux Kernel Development (3/e)*, Addison-Wesley Professional, 2010.
- [Mau2008] W. Mauerer, Professional Linux Kernel Architecture, Wiley, 2008.

Contenidos

- Conceptos fundamentales sobre procesos
- Operaciones sobre procesos
- *Threads* (Hebras o hilos)
- Conceptos fundamentales sobre planificación
- Políticas de planificación de la CPU
- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Concepto de proceso

- Un proceso es un programa en ejecución y la ejecución de este programa debe realizarse sin interferencias debidas a la ejecución de otros programas.
  - Programa = fichero estático ejecutable
  - Proceso = programa en ejecución = programa + estado de ejecución
- La ejecución del programa puede caracterizarse por su **traza de ejecución**.
- Distintos procesos pueden ejecutar el mismo programa.
- Para ejecutar un programa, un proceso requiere al menos: memoria para texto (código), datos y pila; y la CPU (conjunto de registros) para poder ejecutarse.

Concepto de proceso

- Un proceso requiere recursos que el SO se encarga de controlar: memoria, CPU,... (muchos más).
- Por ejemplo, el **SO multiplexa la CPU** para permitir la ejecución de varios procesos, por lo que es necesario almacenar el **contexto de ejecución en CPU** de un proceso para poder cargarlo posteriormente.
- Pero, el SO es un programa que también se ejecuta en la CPU, luego: ¿Cómo se gestiona la ejecución del programa SO y los distintos programas asociados a los procesos?

Conceptos sobre kernel y programa de usuario

- Con respecto a la ejecución en CPU se utilizan dos niveles de privilegio (**modos de ejecución**): **usuario** y **kernel** (ej. IA-32 ring 3 y ring 0 respectivamente)
- Con respecto a la memoria:
  - Espacio protegido para el kernel:
    - La CPU opera sobre este espacio solamente en modo kernel.
    - Para trabajar sobre este espacio se require una secuencia especial de instrucciones para cambiar de modo user a modo kernel.
    - El código de este espacio es **reentrante**.

Conceptos sobre kernel y programa de usuario

- Con respecto a la memoria (cont.):
  - Cada proceso tiene su propio espacio de direcciones protegido:
    - Es necesario tener información sobre el estado de la memoria asociada a cada proceso. Esta información se guarda en espacio de memoria del kernel.
    - Todos los procesos comparten el mismo espacio de kernel.
    - Existe una pila de kernel por cada proceso ya que el kernel es un código reentrante.

Modos de ejecución, espacio de direcciones y contexto

Modo / Contexto	User	Kernel
Proceso	Código de usuario	Llamadas al sistema y excepciones
Kernel	(No permitido)	Tratamiento de Interrupciones, Tareas del sistema.

Ejecución del SO

Núcleo fuera de todo proceso:

- Ejecuta el núcleo del sistema operativo fuera de cualquier proceso.
- El código del sistema operativo se ejecuta como una entidad separada que opera en modo privilegiado.

Ejecución dentro de los procesos de usuario:

- Software del sistema operativo en el contexto de un proceso de usuario.
- Un proceso se ejecuta en modo privilegiado (**modo kernel**) cuando se ejecuta el código del sistema operativo.

Nuestra idea de proceso

- Unidad de actividad caracterizada por la ejecución de una secuencia de instrucciones (traza de ejecución), un estado de computación actual (contexto de registros), y un conjunto de recursos del sistema operativo asociados.
- En el sistema hay muchos procesos simultáneamente. Cuando el SO decide que un proceso ejecutándose en CPU debe abandonarla, tiene que salvar los valores actuales de los registros de CPU (**contexto de registros**) en el **PCB** de dicho proceso.
- La acción de conmutar la CPU de un proceso a otro se denomina **cambio de contexto** (***context switch***). El tiempo que el SO emplea en cada cambio de contexto va en detrimento de la productividad del sistema.

Process Control Block (PCB)

- Estructura de datos que contiene la información relativa al concepto de proceso.
- El PCB es creado, gestionado y destruido por el kernel.
- Información básica que contiene:
  - *Process IDentifier* (PID).
  - *Process State* (*state diagram*).
  - Contexto de Registros.
  - Información de memoria.
  - Lista de recursos utilizados.

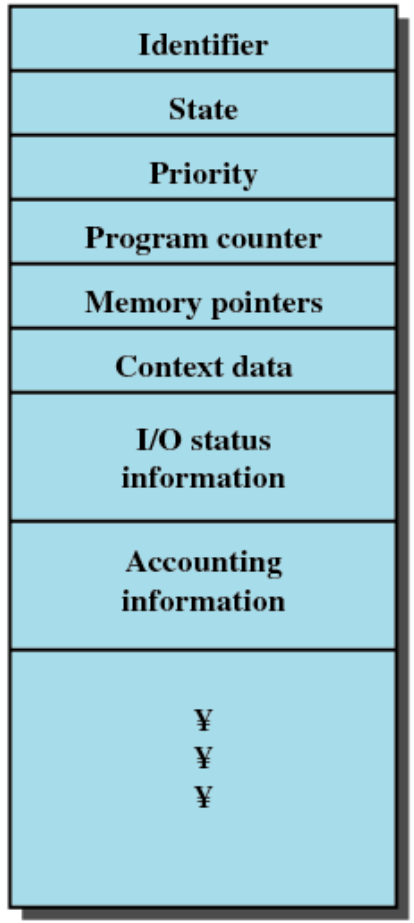


Figure 3.1 Simplified Process Control Block

PCB

- Un proceso tiene texto (código) y datos asociados (el programa a ejecutar).
- Un proceso tiene una pila asociada para poder realizar llamada a funciones.
- El SO almacena el estado de ejecución del programa en el PCB.
- Los “datos” (texto, datos y pila) residen en memoria, los metadatos son el PCB y también residen en memoria.

Un gazapo

- La figura muestra una imagen de un libro de introducción a los sistemas operativos en la que se describe el concepto de proceso…
- ¿Hay algo que “choca”?
- ¿Dónde reside el PCB en el espacio de direcciones del proceso?
- ¿Dónde residen las pilas de usuario y núcleo en el espacio de direcciones del proceso?

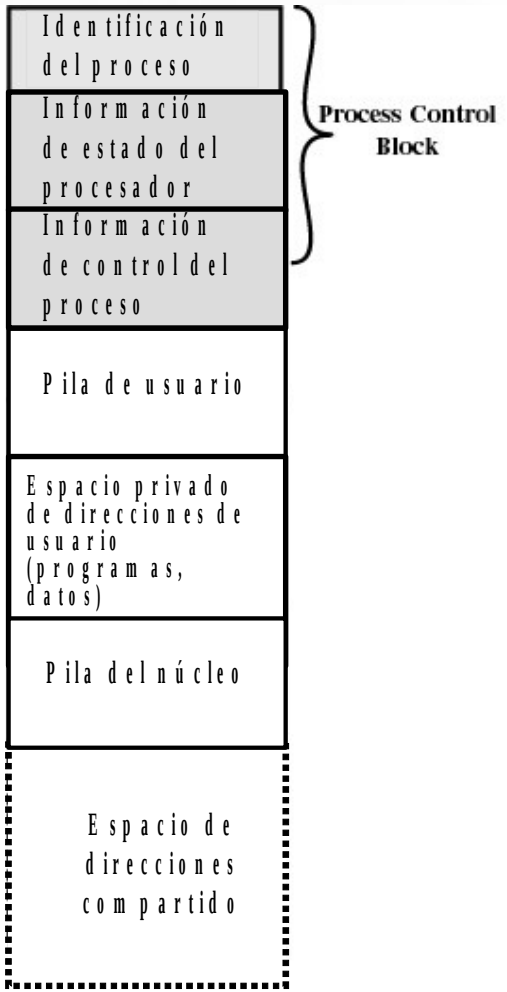


Figura 3.15. Imagen de un proceso: el sistema operativo se ejecuta dentro del proceso de usuario.

Tablas del kernel del SO

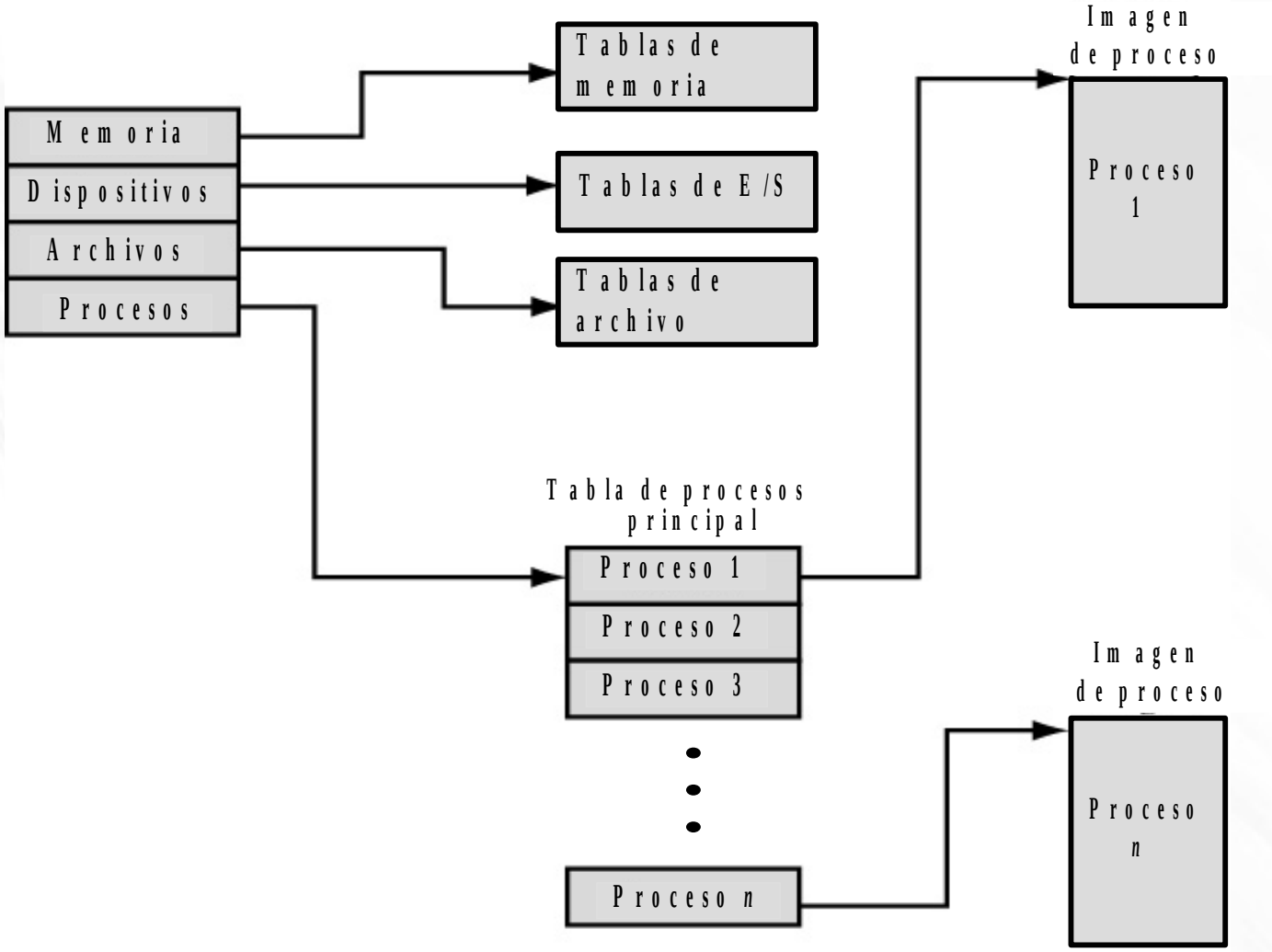


Figura 3.10. Estructura general de las tablas de control del sistema operativo.

Cambio de contexto

- Cuando un proceso esta ejecutándose, el programa asociado está utilizando los registros de la CPU: PC,SP,PSW... En otras palabras, los registros contienen el estado actual de la computación.
- Cuando el SO detiene un proceso que está ejecutándose, salva los valores actuales de estos registros (contexto de registros) en el PCB de dicho proceso.
- La acción de cambiar al proceso que está usando la CPU por otro proceso se denomina **cambio de contexto**.
- El cambio de contexto solamente puede llevarse a cabo en *kernel mode*... ¿Por qué?

Cambio de contexto

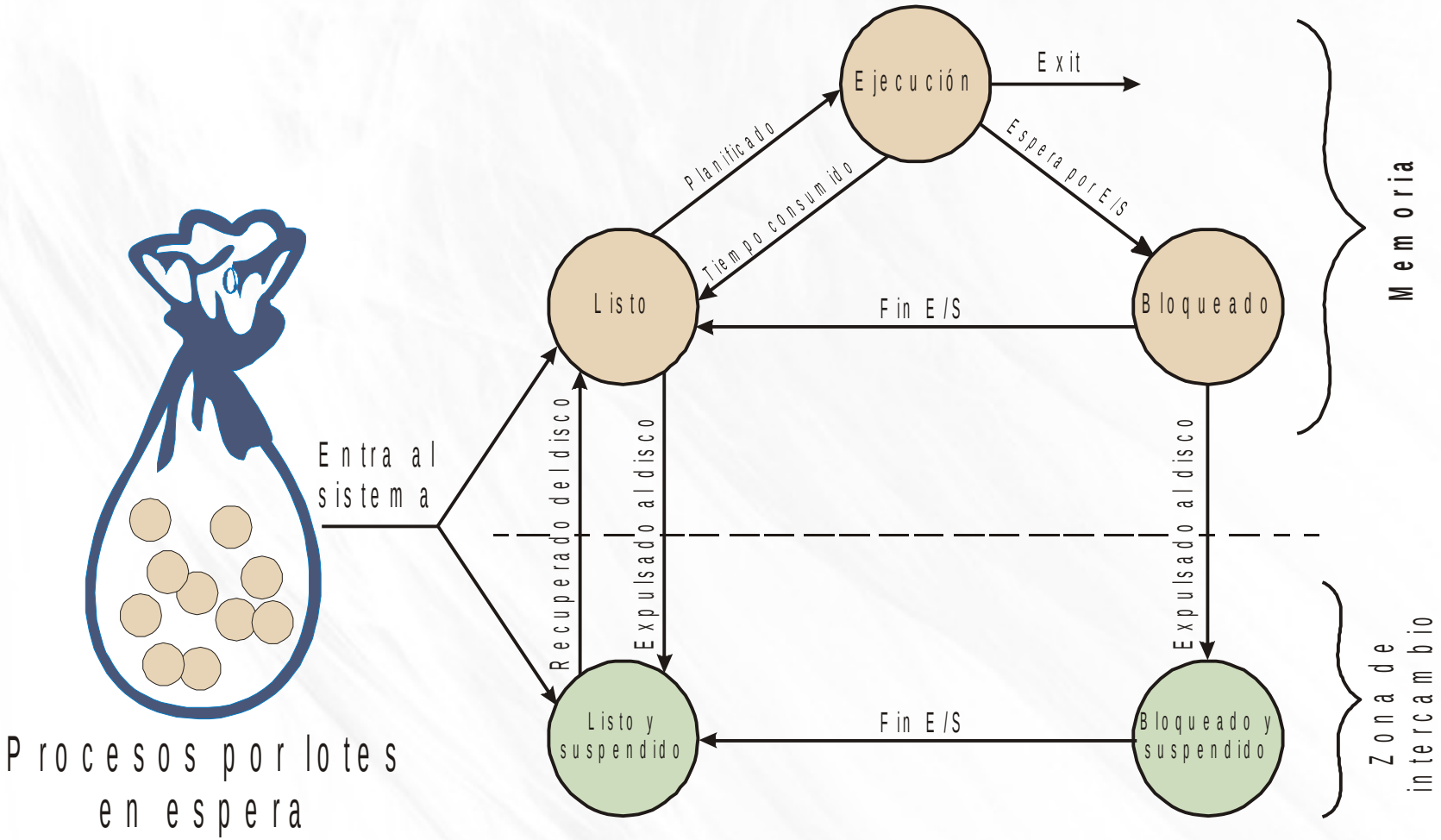
- La operación de cambio de contexto es fundamental en sistemas operativos multitarea (*multitasking* OS).
- ¿Cuándo se produce un cambio de contexto?
  - El proceso en ejecución ha agotado su *time slice*.
  - Como consecuencia de una llamada al sistema bloqueante.
  - ¡Como consecuencia de una interrupción!
  - ¡El proceso decide por si mismo abandonar la CPU!
- El cambio de contexto representa un alto coste para el sistema en términos de tiempo de CPU.
- Orden de magnitud de nanosegundos (10^{-9} segundos) para cada uno de los cientos (¡o miles o diezmiles!) de cambios de contexto por segundo.

Cambio de contexto de registros de CPU

Descripción general de la operación de cambio de contexto de registros de CPU:

1. Dejar en supenso la ”ejecución” de un proceso almacenando el contexto de registros en el PCB.
2. Restaurar el contexto de registros del proceso que va a ejecutarse en CPU, copiando el contexto de registros (previamente salvado) que se encuentra en su PCB.
3. Continuar el ciclo captación-ejecución de instrucciones utilizando el nuevo valor del registro PC.

Diagrama de estados modificado



Contenidos

- Conceptos fundamentales sobre procesos
- Operaciones sobre procesos
- *Threads* (Hebras o hilos)
- Conceptos fundamentales sobre planificación
- Políticas de planificación de la CPU
- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Operaciones sobre procesos
Creación de procesos

- ¿Qué significa crear un proceso?
  - Asignarle el espacio de direcciones que utilizará.
  - Crear las estructuras de datos para su administración.
- ¿Por qué motivos se puede crear un proceso?
  - En sistemas *batch*: en respuesta a la recepción y admisión de un trabajo.
  - "*logon*" interactivo. Cuando el usuario se autentifica desde un terminal (*logs on*), el SO crea un proceso que ejecuta el intérprete de órdenes asignado.
  - El SO puede crear un proceso para llevar a cabo un servicio solicitado por un proceso de usuario.
  - Un proceso puede crear otros procesos formando un árbol de procesos (relación padre-hijo).

Operaciones sobre procesos
Creación de procesos

En el caso de que un proceso cree un nuevo proceso (hijo), ¿cómo obtiene sus recursos el nuevo proceso (hijo)?

- Los obtiene directamente del SO: padre e hijo no comparten recursos.
- Comparte todos los recursos con el padre.
- Comparte un subconjunto de los recursos del padre.

Operaciones sobre procesos
Creación de procesos

- Posibilidades de ejecución del proceso padre e hijo:
  - Padre e hijo se ejecutan concurrentemente.
  - Padre espera a que el hijo termine.
- Posibilidades en cuanto a espacio de direcciones:
  - El espacio de direcciones del hijo es un duplicado del padre (UNIX-like OS).
  - El espacio del hijo contiene un programa nuevo distinto al del programa asociado al proceso padre (VMS).

Operaciones sobre procesos
Creación de procesos

¿Cómo funciona en UNIX-like OS?

- La llamada al sistema **`fork()`** crea un nuevo proceso (hijo).
- La llamada **`exec()`** reemplaza el espacio de direcciones con el programa pasado como argumento.
- Si se realiza una llamada **`fork()`** seguida de una llamada **`exec()`** se consigue un proceso hijo que ejecuta el programa pasado como argumento en **`exec()`**.
- Un ejemplo de esta técnica es la ejecución de órdenes desde un proceso que ejecuta un *shell* intérprete de órdenes.

Operaciones sobre procesos
Creación de procesos

- **`fork()`** crea un nuevo hijo que hereda:
  - Una copia idéntica de la memoria del padre.
  - Una copia idéntica de los registros de CPU del padre.
- Los procesos padre e hijo se ejecutan en el mismo punto de ejecución: tras el retorno de **`fork()`** y, por convenio:
  - **`fork()`** devuelve un 0 en el proceso hijo.
  - **`fork()`** devuelve el identificador de proceso del hijo en el proceso padre.

Operaciones sobre procesos
Creación de procesos

- Ejemplo de programa simple que usa **`fork()`**.

```
main(int argc, char* argv[])
{
  pid_t forkRetVal;    /* Stores the value returned by fork() syscall */

  /* fork() creates a new process named child process. The process which creates is named
the parent process. */
  forkRetVal= fork();

  /* fork() returns 0 in the child process and the child process' PID in the parent
process. */

  if (forkRetVal == 0) {  /* This code is ONLY executed by the child process */
    /* Things to do exclusively by child process */
    printf("Child -- Hello, I was born! forkRetVal = %d\n",forkRetVal);
  }
  else {  /* forkRetVal != 0, then this code is ONLY executed by the parent process */
    /* Things to do exclusively by the parent process. */
    printf("Parent -- Hello, I created a child! forkRetVal = %d\n",forkRetVal);
  }

  /* This code is executed both by the child and parent processes */
  printf("Bye, bye!\n\n");

  return EXIT_SUCCESS;
}
```

Operaciones sobre procesos
Creación de procesos

Pasos fundamentales a realizar por un kernel cuando se solicita la operación de creación de proceso:

- Identificar al proceso asignándole un PID único.
- Asignar espacio en memoria RAM y/o en memoria secundaria (SWAP) para el programa que se ejecutará.
- Crear el PCB e inicializar los campos de información.
- Insertar el PCB en la Tabla de Procesos y enlazarlo en la cola de planificación correspondiente, caso de residir el programa en RAM.

Operaciones sobre procesos
Terminación de procesos

¿Qué situaciones determinan la finalización de un proceso?

- Cuando un proceso ejecuta la última instrucción, solicita al SO su finalización mediante la llamada **`exit()`**, lo que implica:
  - Aviso de finalización al padre (SIGCHLD) y guardar estado de finalización.
  - Recursos asociados al proceso son liberados por el SO.
- El proceso padre puede finalizar la ejecución de sus procesos hijos mediante la llamada **`kill()`**.
- El proceso padre va a finalizar y el SO no permite a los procesos hijos continuar con la ejecución (Terminación en cascada).
- El SO puede terminar la ejecución de un proceso porque se hayan producido errores o condiciones de fallo.

Contenidos

- Conceptos fundamentales sobre procesos
- Operaciones sobre procesos
- *Threads* (Hebras o hilos)
- Conceptos fundamentales sobre planificación
- Políticas de planificación de la CPU
- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Threads (hebras o hilos)

- Una hebra es la unidad básica de utilización de la CPU y el kernel require la siguiente información para gestionarla:
  - Contexto de registros: CP,SP,PSW,...
  - Pila de ejecución.
  - Estado (diagrama de estados).
- Una hebra comparte con sus hebras pares una tarea (proceso) que mantiene el resto de información necesaria:
  - Sección de código (texto).
  - Sección de datos.
  - Recursos del SO (archivos abiertos, señales,...).

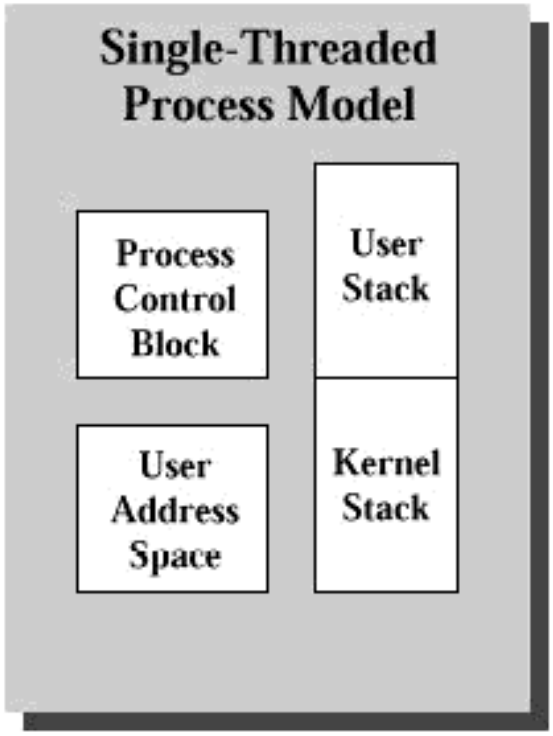
Threads (hebras o hilos)

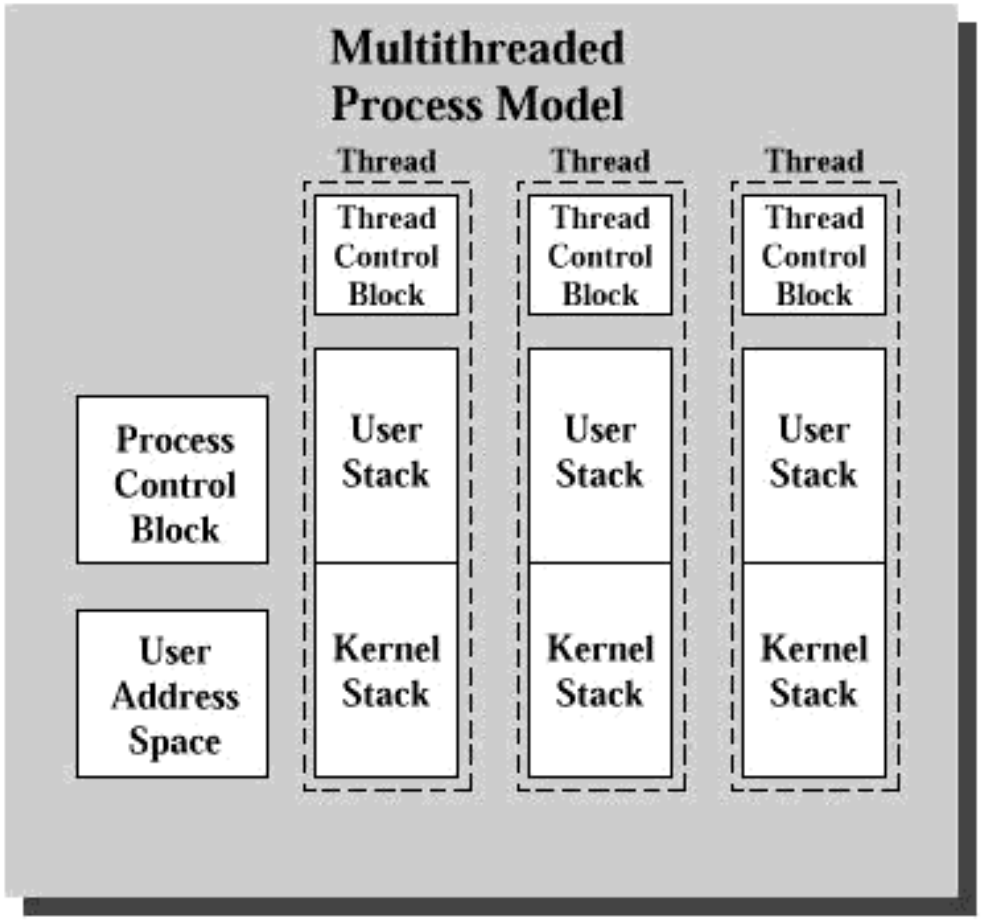
- Un proceso tradicional se puede ver como una tarea con una sola hebra.

```
$ ps -p 8175 -fL

UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
aleon     8175  7035  8175  0    1 10:59 pts/2    00:00:00 bash
```

```
 /* Create child process */
  forkRetVal= fork();

  switch (forkRetVal) {
  case -1: /**< fork failure */
    handle_error("Error en fork(): ");
    break;
  case 0:  /**< 0 is returned on the child process */
    printf("Child-- My PID is %d\n",getpid());
    printf("Child-- My TID is %ld\n", syscall(SYS_gettid));
    break;
  default: /**< PID of the child process is returned on the parent process */
    printf("Parent-- My PID is %d\n",getpid());
    printf("Parent-- My TID is %ld\n", syscall(SYS_gettid));
    printf("Parent-- My child's PID as returned by fork() is %d\n",forkRetVal);
    break;
  }
```

Threads (hebras o hilos)





Threads (hebras o hilos)

- Se entiende que un sistema operativo es ***single threading*** cuando el kernel no reconoce el concepto de hebra.
- Se entiende que un sistema operativo es ***multithreading*** cuando el kernel soporta múltiples hebras dentro de un proceso (tarea).
- UNIX con kernel tradicional soporta múltiples procesos de usuario pero solo una hebra por proceso.
- Sun Solaris fue el primero en soportar múltiples hebras por proceso.

Threads: Ventajas

- Se reduce el tiempo de cambio de contexto entre hebras, así como el tiempo de creación y terminación de hebras.
- Mientras una hebra de una tarea esta bloqueada, otra hebra de la misma tarea puede ejecutarse, siempre que el kernel sea *multithreading*. Por ejemplo, cualquier GUI.
- Usan los sistemas multiprocesador de manera eficiente y transparente, a mayor número de procesadores, mayor rendimiento (*performance*).
- La comunicación entre hebras de una misma tarea se realiza a través del espacio de direcciones asociado a la tarea por lo que no necesitan utilizar los mecanismos del núcleo.

Threads: Funcionalidad

- Al igual que los procesos la hebras poseen un estado (diagrama de estados) y pueden sincronizarse.
- Estados de las hebras: Nuevo, Ejecución, Listo, Bloqueado y Finalizado.
- Operaciones básicas relacionadas con el cambio de estado en hebras:
  - Creación.
  - Bloqueo.
  - Desbloqueo.
  - Terminación.
- Sincronización entre hebras.

Tipos de *threads*

- Teniendo en cuenta la distinción entre *single threading kernel* y *multithreading kernel*, podemos hablar de:
  - Hebras en biblioteca de usuario, *user-level threads* (ULT).
  - Hebras a nivel kernel, *kernel-level threads* (KLT).
  - Enfoques híbridos.

Tipos de *threads*: ULT

- Toda la gestión de hebras se realiza a nivel usuario mediante el uso de una biblioteca de hebras.
- La biblioteca se encarga de proporcionar operaciones para:
  - Crear/finalizar hebras.
  - Gestionar el modelo de estados de las hebras.
  - Planificar hebras y salvar/cargar el contexto de hebra.
  - Permitir la comunicación entre hebras.
- El kernel no es consciente de las actividades relacionadas con hebras ya que solo gestiona el proceso (tarea).
- La entidad de planificación para el kernel es el proceso.

Tipos de *threads*: KLT

- Toda la gestión de hebras se realiza a nivel kernel por lo que este tiene que mantener información para procesos (tareas) y hebras.
- El SO proporcional un conjunto de llamadas al sistema para la gestión de hebras.
- La entidad de planificación para el kernel es la hebra.
- Muchas de las funciones que realiza el kernel son gestionadas mediante **hebras kernel**.

ULT: Ventajas e inconvenientes

- Ventajas
  - El cambio de hebra no provoca un cambio de modo.
  - La planificación de hebras puede adaptarse a las necesidades de la aplicación.
  - Las aplicaciones se pueden ejecutar en cualquier SO.
- Inconvenientes
  - La mayoría de las llamadas al sistema son bloqueantes, por lo que si una hebra realiza una llamada, el resto de hebras se bloqueará.
  - El kernel solo asigna procesos a procesadores, por lo que no se puede asignar más de un procesador a más de una hebra de la misma tarea.

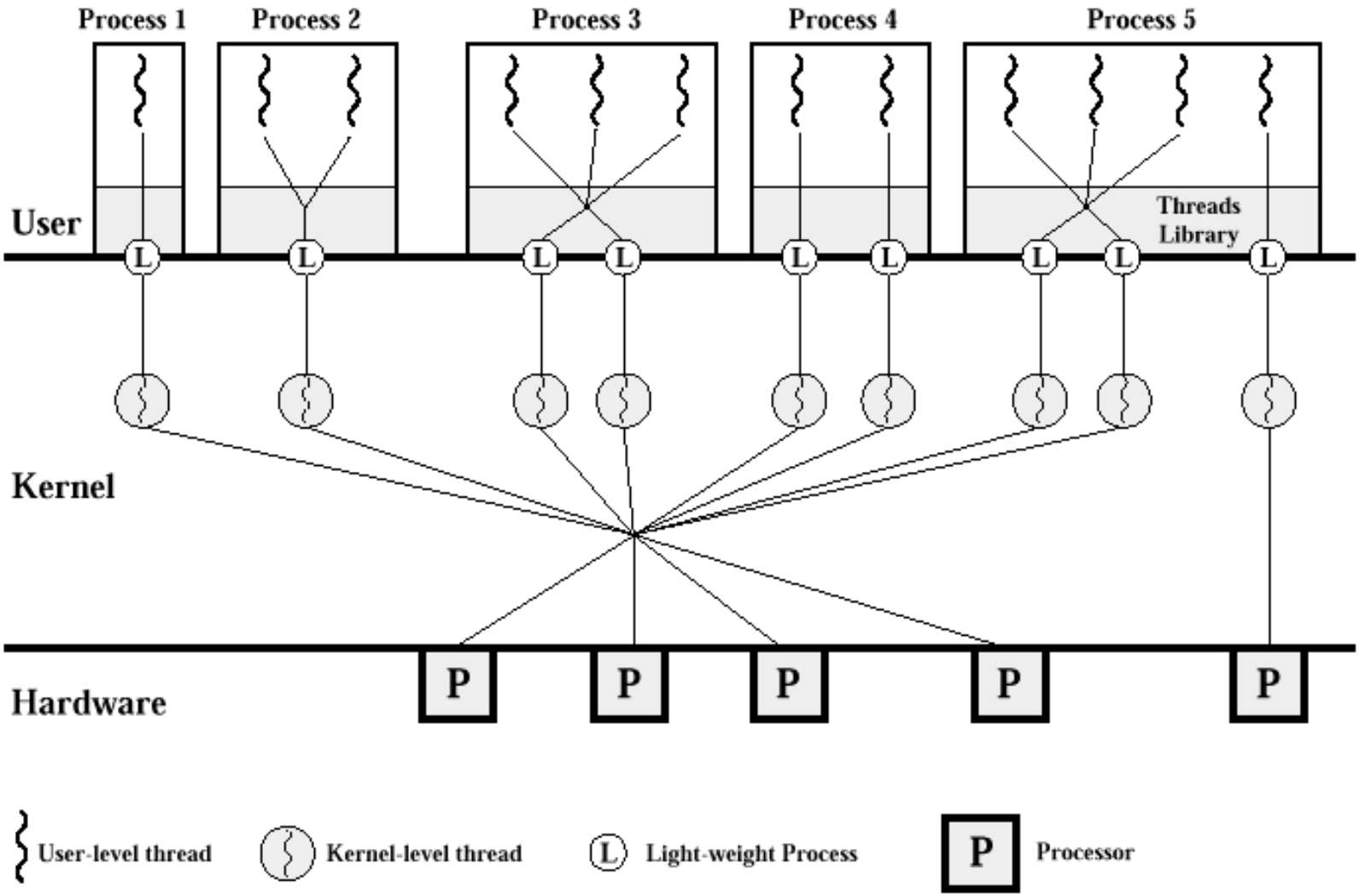
KLT: Ventajas e inconvenientes

- Ventajas
  - El kernel puede planificar distintas hebras de la misma tarea en distintos procesadores.
  - El bloqueo de una hebra de una tarea no provoca el bloqueo del resto de “hebras pares”.
  - Las rutinas del kernel pueden ser multihebra.
- Inconvenientes
  - El cambio de hebras dentro de la misma tarea se realiza en modo kernel, por lo que provoca un cambio de modo.

Enfoques híbridos: ULT y KLT

- Solaris OS es un ejemplo de combinación de ULT y KLT.
- Las ULT proporcionadas por la biblioteca de hebras son totalmente invisibles para el kernel (API de hebras).
- La creación de hebras, y la mayor parte de la planificación y sincronización de hebras se realiza en modo usuario.
- Las KLT son la unidad de planificación del kernel.
- El programador puede decidir el número de KLTs.
- Los “procesos ligeros” (Lightweight processes, **LWP**) soportan una o más ULT y se asocian con una KLT.
- El concepto LWP permite que las hebras de una tarea potencialmente bloqueantes no bloqueen al resto de las hebras de la tarea.

Enfoques híbridos: ULT y KLT



Contenidos

- Conceptos fundamentales sobre procesos
- Operaciones sobre procesos
- *Threads* (Hebras o hilos)
- Conceptos fundamentales sobre planificación
- Políticas de planificación de la CPU
- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Planificación

- Idea. Dispongo de ‘n’ clientes que desean acceder a un recurso y tengo que decidir a qué cliente le asigna el recurso.
- Definición del problema de Planificación de CPU:
  - El SO dispone de ‘n’ procesos/hebras (depende de la entidad de planificación del kernel) en estado ‘LISTO’.
  - El SO dispone de k >= 1 CPUs (o cores) para ejecutar procesos/hebras.
  - El SO debe decidir qué proceso/hebra asignar a qué CPU.

Planificación ¿qué vamos a ver?

- Primero extenderemos el diagrama de estados de los procesos con el modelo genérico de colas y los planificadores asociados.
- Luego se introduce la clasificación de los procesos en cortos (limitados por E/S) y largos (limitados por CPU) necesaria para entender la actuación de los planificadores y las políticas de planificación de CPU.
- Despachador (**`Dispatcher()`**)
- Medidas para prestaciones de planificadores.
- Políticas de planificación monoprocesador.
- Políticas de planificación multiprocesador y tº real.

Planificación: PCB's y Colas de Estados

Modelo genérico de colas

- El SO mantiene una colección de colas que representan el estado de todos los procesos en el sistema.
- Típicamente hay una cola por estado del proceso.
- Cada PCB esta encolado en una cola de estado acorde a su estado actual.
- Conforme un proceso cambia de estado, su PCB es retirado de una cola y encolado en otra.

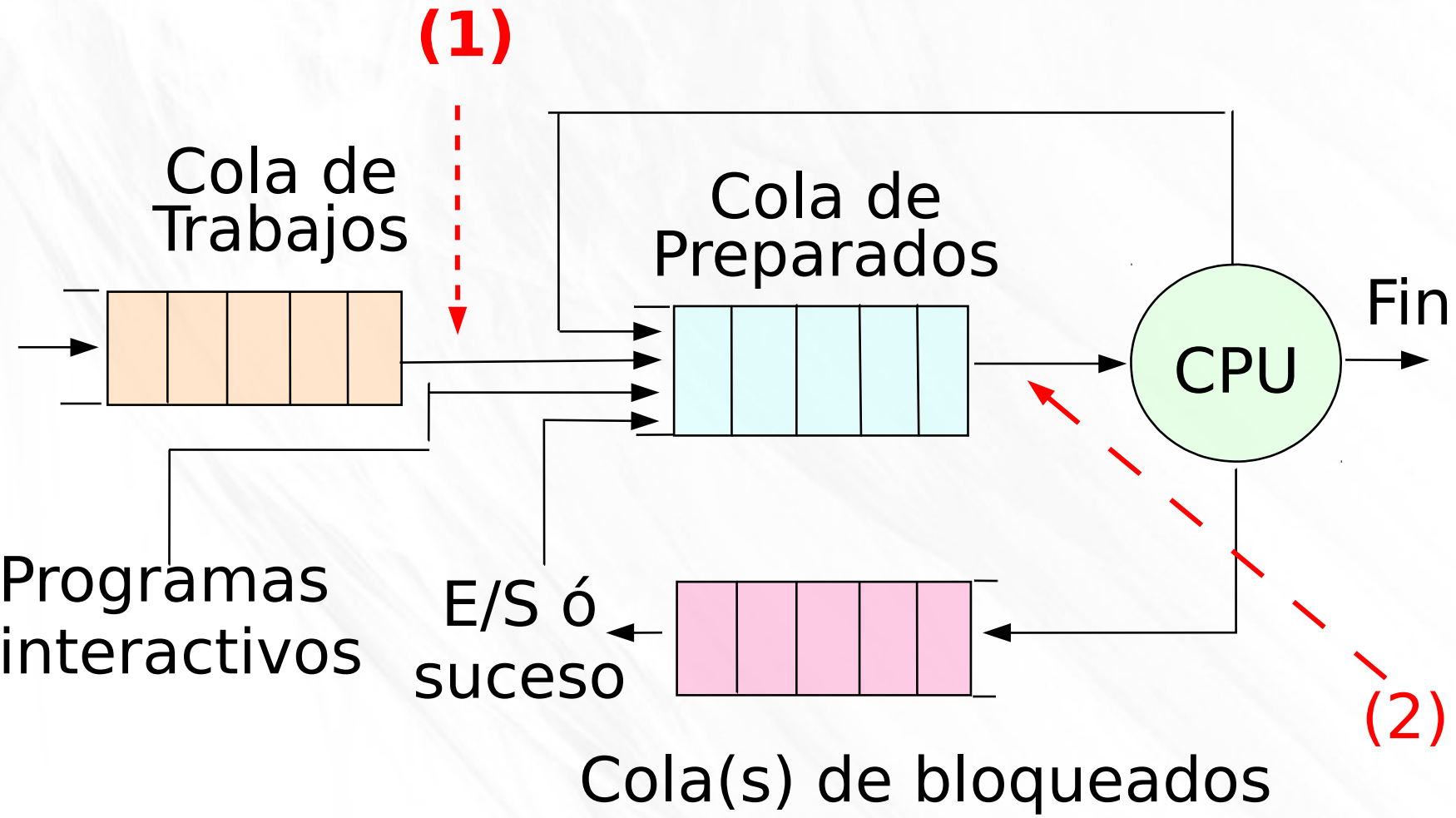
Colas de estados

- **Cola de trabajos**. Conjunto de los trabajos pendientes de ser admitidos en el sistema, e.d. Trabajos por lotes (*batch*).
- **Cola de LISTOS (Preparados para Ejecutar)**. Conjunto de todos los procesos, cuyos programas asociados residen en memoria principal, esperando para poder ejecutarse en la CPU.
- **Colas de BLOQUEADOS**. Conjunto de todos los procesos esperando por un dispositivo de E/S particular o por un evento que debe producirse para poder continuar su ejecución.

Tipos de planificadores

- **Planificador**. Parte del SO que controla la utilización de un recurso.
- Tipos de planificadores:
  - **Planificador a Largo plazo** (Planificador de trabajos): selecciona los trabajos que deben llevarse a la cola de preparados; (1) en la figura de la siguiente transparencia.
  - Planificador a corto plazo (**Planificador de CPU**): selecciona el proceso que debe ejecutarse a continuación, y le asigna la CPU; (2) en la figura.
  - **Planificador a Medio lazo.**

Migración entre colas



Características de los planificadores

Planificador de CPU:

- Trabaja con la Cola de LISTOS.
- Se invoca muy frecuentemente (milisegundos) por lo que debe ser rápido.

Planificador a Largo plazo:

- Permite controlar el ***grado de multiprogramación***.
- Se invoca poco frecuentemente (segundos o minutos), por lo que puede ser más lento.

Clasificación de procesos

- Procesos **limitados por E/S *o procesos cortos.***
  Dedican más tiempo a realizar E/S que computo; muchas ráfagas de CPU cortas y largos períodos de espera.

- Procesos **limitados por CPU *o procesos largos***.
  Dedican más tiempo en computación que en realizar E/S; pocas ráfagas de CPU pero largas.

Mezcla de trabajos

Es importante que el **Planificador a Largo plazo** seleccione una buena mezcla de trabajos, ya que:

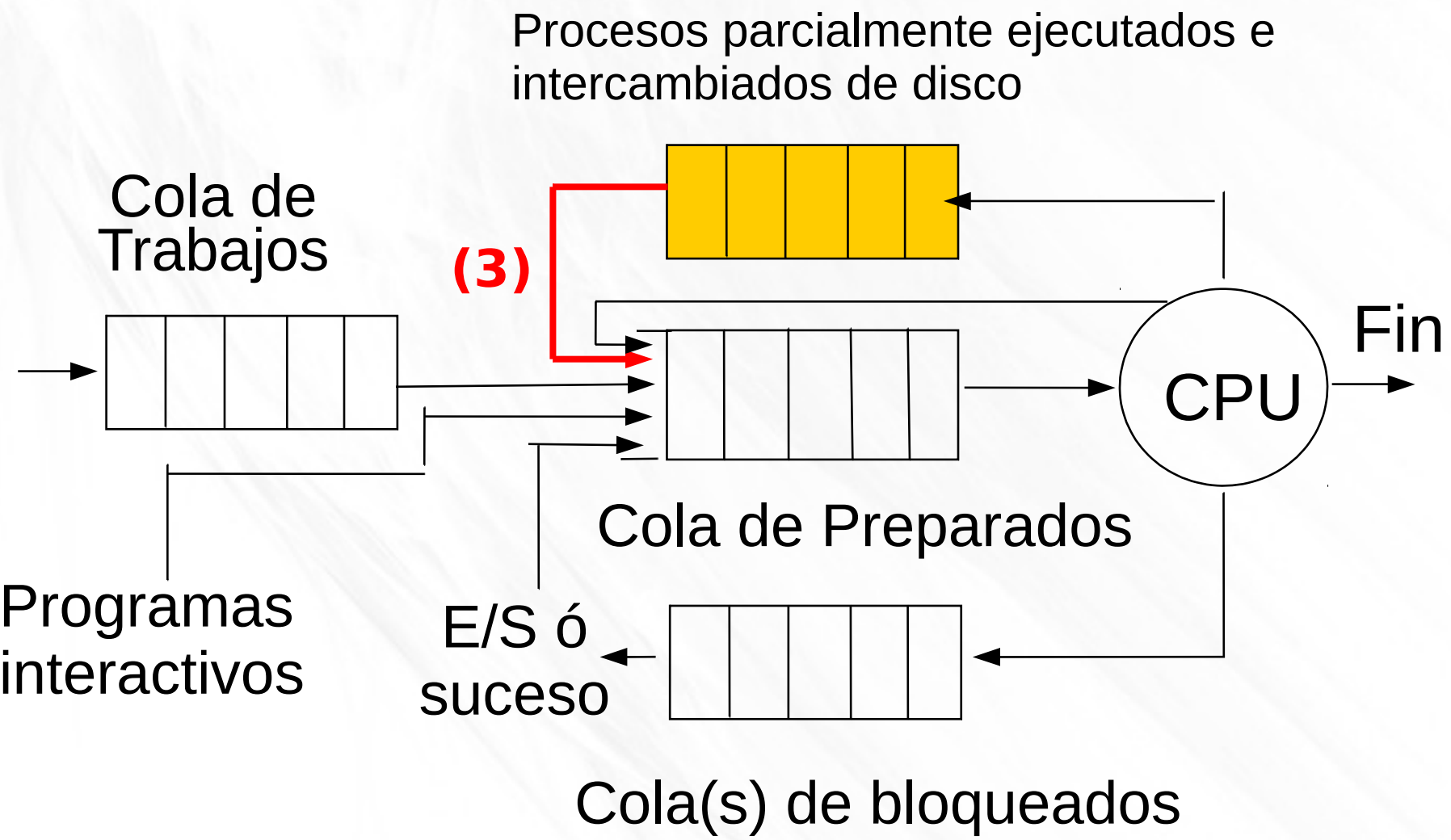
- Si todos los trabajos están limitados por E/S, la cola de preparados estará casi siempre vacía y el planificador a corto plazo tendrá poco que hacer.

- Si todos los procesos están limitados por CPU, la cola de E/S estará casi siempre vacía y el sistema estará de nuevo desequilibrado.

Planificador a Medio plazo

- En algunos SO’s, p. ej. SO de tiempo compartido, a veces es necesario sacar procesos de la memoria (reducir *el grado de multiprogramación*), bien para mejorar la mezcla de procesos, bien por cambio en los requisitos de memoria, y luego volverlos a introducir. Este prodecimiento se conoce como **Intercambio** (***swapping***).
- El **planificador a Medio plazo** se encarga de devolver los procesos a memoria. Transición (3) en la siguiente figura.

Planificador a medio plazo (y II)



Despachador (`dispatcher()`)

- El **Despachador** es la función del SO que da el control de la CPU al proceso seleccionado por el planificador a corto plazo. Esto involucra:
  - Cambio de contexto de registros desde el punto de vista de la CPU (se realiza en modo kernel).
  - Conmutación a modo usuario o a modo kernel dependiendo del nuevo contexto de registros.

- **Latencia de despacho**.Tiempo que emplea el despachador en detener un proceso y comenzar a ejecutar otro.

Activación del Despachador

El despachador puede actuar cuando:

- Un proceso no quiere (**finaliza**) o no puede continuar ejecutando instrucciones. En el segundo caso, la solicitud de recurso o comunicación por parte del proceso provoca que el SO lo pase a estado bloqueado.
- Un elemento del SO determina que el proceso no puede seguir ejecutándose (ej. E/S o retirada de memoria principal)
- El proceso agota el *quantum* de tiempo asignado (*time slice*).
- Un suceso cambia el estado de un proceso de BLOQUEADO a LISTO (más adelante lo vemos).

Políticas de planificación: medidas

- **Objetivos**:
  - Buen rendimiento (productividad)
  - Buen servicio
- Para saber si un proceso obtiene un buen servicio, definiremos un conjunto de medidas: dado un proceso **P**, que necesita un tiempo de servicio (**ráfaga**) ***r***
- ***Tiempo de respuesta (T)***-- tiempo transcurrido desde que se remite una solicitud (entra en la cola de listos) hasta que se produce la primera respuesta (no se considera el tiempo que tarda en dar la salida)
- ***Tiempo de espera (M)***-- tiempo que un proceso ha estado esperando en la cola de listos (preparados): T – r
- ***Penalización (P)*** -- T / r
- ***Indice de respuesta (R)*** -- r / T : fracción de tiempo que P está recibiendo servicio.

Políticas de planificación: medidas

Otras medidas interesantes son:

- **Tiempo del núcleo** – tiempo perdido por el SO tomando decisiones que afectan a la planificación de procesos y haciendo los cambios de contexto necesarios. En un sistema eficiente debe representar entre el 10% y el 30% del total del tiempo del procesador.
- **Tiempo de inactividad** – tiempo en el que la cola de ejecutables está vacía y no se realiza ningún trabajo productivo.
- **Tiempo de retorno (turnaround time)** – cantidad de tiempo necesario para ejecutar un proceso completo.

Políticas de planificación

- Las políticas de planificación se comportan de distinta manera dependiendo de la clase de procesos (cortos o largos).
- Ninguna política de planificación es completamente satisfactoria, cualquier mejora en una clase de procesos es a expensas de perder eficiencia en los procesos de otra clase.
- Podemos clasificarlas en:
  - **No apropiativas (no expulsivas)**: una vez que se le asigna el procesador a un proceso, no se le puede retirar hasta que éste voluntariamente lo deje (finalice o se bloquee)
  - **Apropiativas (expulsivas) (preemptive)**: al contrario, el SO puede apropiarse del procesador cuando lo decida.

Revisitando `context_switch()`

- **`context_switch()`** está compuesta de **`schedule()`**, planificador de CPU, y **`dispatch()`**, despachador.
- ¿Cuándo llama el SO a **`schedule()`**? O lo que es lo mismo... ¿Cuándo toma decisiones de planificación el SO?

1. E → B, pej. dentro de **`RutinaE/S()`** o **`wait()`** *syscall*.
2. E → F, **`sys_exit()`** *kernel algorithm*.
3. E → L, Dentro de la **`RSI_reloj()`**.
4. {N,B,SL} → L, implementando CPU *preemption*.

- Si el planificador es no apropiativo (*non-preemptive scheduling*) solo se tienen en cuenta 1 y 2.
- Si el planificador es apropiativo (*Preemptive scheduling*) se tienen en cuenta todas: 1-4.

Contenidos

- Conceptos fundamentales sobre procesos
- Operaciones sobre procesos
- *Threads* (Hebras o hilos)
- Conceptos fundamentales sobre planificación
- Políticas de planificación de la CPU
- Implementación de proceso/hebra en Linux: *task*
- Planificación de CPU en Linux

Políticas de planificación de la CPU

- *First Come First Served* (FCFS)
- El más corto primero (*Shortest Job First*, SJF):
  - No apropiativo
  - Apropiativo (*Shortest Remaining Time First*, SRTF)
- Planificación por prioridades
- Por turnos (*Round-Robin*)
- Colas múltiples
- Colas múltiples con realimentación.

FCFS (*First Come First Served)*

- Los procesos son servidos según el orden de llegada a la cola de ejecutables.
- Es ***no apropiativo***, cada proceso se ejecutará hasta que finalice o se bloquee.
- Fácil de implementar pero pobre en cuanto a prestaciones.
- Todos los procesos pierden la misma cantidad de tiempo esperando en la cola de ejecutables independientemente de sus necesidades.
- Procesos cortos muy penalizados.
- Procesos largos poco penalizados.

El más corto primero (*Shortest Job First*)

- Es ***no apropiativo***.
- Cuando el procesador queda libre, selecciona el proceso que requiera un **tiempo de servicio menor**.
- Si existen dos o más procesos en igualdad de condiciones, se sigue FCFS.
- Necesita conocer explícitamente el tiempo estimado de ejecución (tº servicio) ¿Cómo?.
- Disminuye el tiempo de respuesta para los procesos cortos y discrimina a los largos.
- Tiempo medio de espera bajo.

El más corto primero apropiativo. (*Shortest Remaining Time First*)

- Cada vez que entra un proceso a la cola de listos se comprueba si su tiempo de servicio es menor que el tiempo de servicio que le queda al proceso que está ejecutándose.
  - **SI es menor**: se realiza un cambio de contexto y el proceso con menor tiempo de servicio es el que se ejecuta.
  - **NO es menor**: continúa el proceso que estaba ejecutándose.
- El tiempo de respuesta es menor excepto para procesos muy largos.
- Se obtiene la menor penalización en promedio (mantiene la cola de ejecutables con la mínima ocupación posible).

Planificación por prioridades

- Asociamos a cada proceso un número de prioridad (entero).
- Se asigna la CPU al proceso con mayor prioridad (enteros menores = mayor prioridad)
- Se puede implementar con modalidades:
  - Apropiativa
  - No apropiativa
- **Problema**: **Inanición** -- los procesos de baja prioridad pueden no ejecutarse nunca.
- **Solución**: **Envejecimiento** -- con el paso del tiempo se incrementa la prioridad de los procesos.

Por Turnos (*Round-Robin*)

- La CPU se asigna a los procesos en intervalos de tiempo (***quatum***).
- Procedimiento:
  - » Si el proceso finaliza o se bloquea antes de agotar el quantum, libera la CPU. Se toma el siguiente proceso de la cola de ejecutables (la cola es FIFO) y se le asigna un quantum completo.
  - » Si el proceso no termina durante ese quantum, se apropia y se coloca al final de la cola de ejecutables.
- ***Es apropiativo***.
- ¿Cómo se implementa?

Por Turnos (*Round-Robin*)

- Valores típicos del quantum entre 10 y 100 ms.
- Penaliza a todos los procesos en la misma cantidad, sin importar si son cortos o largos.
- Las ráfagas muy cortas están más penalizadas de lo deseable.
- ¿Cómo elegir el **valor del quantum**?
  - Muy grande (excede del tº de servicio de todos los procesos) => se convierte en FCFS.
  - Muy pequeño => el sistema monopoliza la CPU haciendo cambios de contexto (tº del núcleo muy alto)

Colas múltiples (*Multilevel Queue*)

- La cola de listos se divide en varias colas y cada proceso es asignado permanentemente a una cola concreta P. ej. Dos colas: Procesos interactivos y procesos *batch*
- Cada cola puede tener su propio algoritmo de planificación P. ej. Interactivos con RR y procesos batch con FCFS
- ***Requiere una planificación entre colas:***
  - **Planificación con prioridades fijas**. P. ej. primero servimos a los interactivos luego a los batch
  - **Tiempo compartido entre colas**. Cada cola obtiene cierto tiempo de CPU que debe repartir entre sus procesos. P. ej. 80% interactivos en RR y 20% a los batch con FCFS

Colas múltiples con realimentación (*Multilevel Feedback Queue*)

- Un proceso se puede mover entre las distintas colas.
- Requiere definir los siguientes parámetros:
  - Número de colas
  - Algoritmo de planificación para cada cola
  - Método utilizado para determinar cuando trasladar a un proceso a otra cola
  - Método utilizado para determinar en qué cola se introducirá un proceso cuando necesite un servicio
  - Algoritmo de planificación entre colas
- Mide en tiempo de ejecución el comportamiento real de los procesos
- Planificación generalmente usada: Unix, Linux,Windows,...

Planificación en multiprocesadores

- Planificación de procesos
  - Igual que en monoprocesadores pero teniendo en cuenta:
    - Número de CPUs
    - Asignación/Liberación proceso-procesador
- Planificación de hilos
  - Permiten explotar el paralelismo real dentro de una aplicación con múltiples hebras.
- **Granularidad** en planificación es la frecuencia de sincronización entre entidades planificables:
  - Grano grueso. Procesos con baja sincronización.
  - Grano medio. Hebras con alta sincronización.

Planificación en multiprocesadores

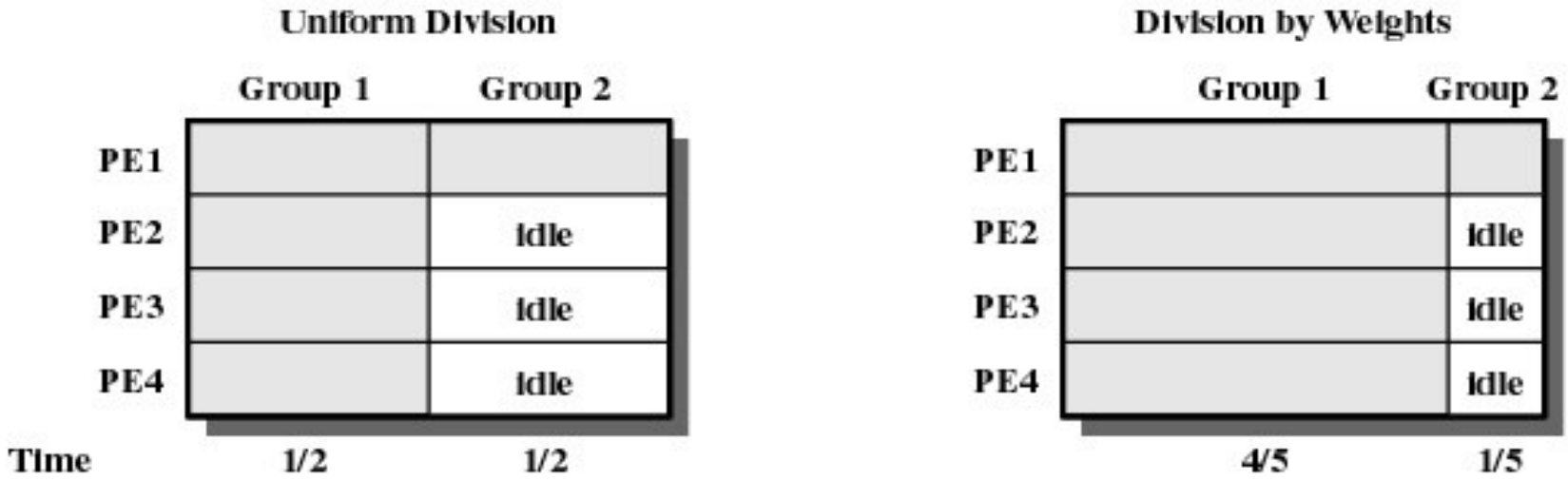
Tres aspectos de diseño del planificador interrelacionados:

- Asignación de procesos a procesadores
  - Cola dedicada para cada procesador
  - Cola global para todos los procesadores
- **Uso de multiprogramación en cada procesador individual**. En aplicaciones multihebra (grano medio) es más importante aumentar el tiempo de respuesta que el uso de CPU → todas las hebras tener asignado un procesador independiente.
- Activación del proceso: **`schedule()`**. Nuevos aspectos de diseño debido a aplicaciones multihebra.

Planificación de hilos en multiprocesadores

1) Compartición de carga
   - Cola global de hilos preparados (LISTOS).
   - Cuando un procesador está ocioso, se selecciona un hilo de la cola (método muy usado).
2) Planificación en pandilla
   - Se planifica un conjunto de hilos afines (de un mismo proceso) para ejecutarse sobre un conjunto de procesadores al mismo tiempo (relación 1 a 1)
   - Útil para aplicaciones cuyo rendimiento se degrada mucho cuando alguna parte no puede ejecutarse (los hilos necesitan sincronizarse)

Ejemplo de planificación en pandilla



37,5% Desperdiciado

15% Desperdiciado

Figure 10.2 Example of Scheduling Groups with Four and One Threads [FEIT90]

Planificación de hilos en multiprocesadores

3) Asignación de procesador dedicado

- Cuando se planifica una aplicación, se asigna un procesador a cada uno de sus hilos hasta que termine la aplicación.
- Algunos procesadores pueden estar ociosos → No hay multiprogramación de procesadores

4) Planificación dinámica

- La aplicación permite que varíe dinámicamente el número de hilos de un proceso
- El SO ajusta la carga para mejorar la utilización de los procesadores

Planificación en sistemas de tiempo real

- La exactitud del sistema no depende sólo del resultado lógico de un cálculo sino también del instante en que se produzca el resultado.
- Las tareas (o procesos) intentan controlar o reaccionar ante sucesos que se producen en “tiempo real” (eventos) y que tienen lugar en el mundo exterior.
- **Características** de las tareas de tiempo real:
  - tº real duro (*hard rt*). La tarea debe cumplir su plazo límite.
  - tº real suave (*soft rt*). La tarea tiene un tiempo límite pero no es obligatorio cumplirlo.
  - Periódicas. Se sabe cada cuánto tiempo se tiene que ejecutar.
  - Aperiódicas. Tiene un plazo en el que debe comenzar o acabar o restricciones respecto a esos tiempos pero son impredecibles

Planificación en sistemas de tiempo real

- Los distintos enfoques dependen de:
- Cuándo el sistema realiza un análisis de viabilidad de la planificación
  - Estudia si puede atender a todos los eventos periódicos dado el tiempo necesario para ejecutar la tarea y el periodo
- Si éste se realiza estática o dinámicamente.
- Si el resultado del análisis produce un plan de planificación o no.

Planificación en sistemas de tiempo real

- Enfoques estáticos dirigidos por tabla.
  - Análisis estático que genera una planificación que determina cuándo empezará cada tarea
- Enfoques estáticos expulsivos dirigidos por prioridad.
  - Análisis estático que no genera una planificación, sólo se usa para dar prioridad a las tareas. Usa planificación por prioridades.
- Enfoques dinámicos basados en un plan.
  - Se determina la viabilidad en tiempo de ejecución (dinámicamente): se acepta una nueva tarea si es posible satisfacer sus restricciones de tiempo.
- Enfoques dinámicos de menor esfuerzo (el más usado).
  - No se hace análisis de viabilidad. El sistema intenta cumplir todos los plazos y aborta ejecuciones si su plazo ha finalizado

Problema de Inversión de Prioridad

- Se da en un esquema de planificación de prioridad y cuando:
  - Una tarea de mayor prioridad espera por una tarea (proceso) de menor prioridad debido al bloqueo de un recurso de uso exclusivo (no compartible).
- Enfoques para evitarla:
  - **Herencia de prioridad**: la tarea menos prioritaria hereda la prioridad de la tarea más prioritaria.
  - **Techo de prioridad**: Se asocia una prioridad a cada recurso de uso exclusivo que es más alta que cualquier prioridad que pueda tener una tarea, y esa prioridad se le asigna a la tarea a la que se le da el recurso.
  - En ambos, la tarea menos prioritaria vuelve a tener el valor de prioridad que tenía cuando libere el recurso.