Procesos

Planificación

3

Sistemas Operativos y Distribuidos

Prof. Javier Echaiz

D.C.I.C. – U.N.S.

http://cs.uns.edu.ar/~jechaiz

je@cs.uns.edu.ar

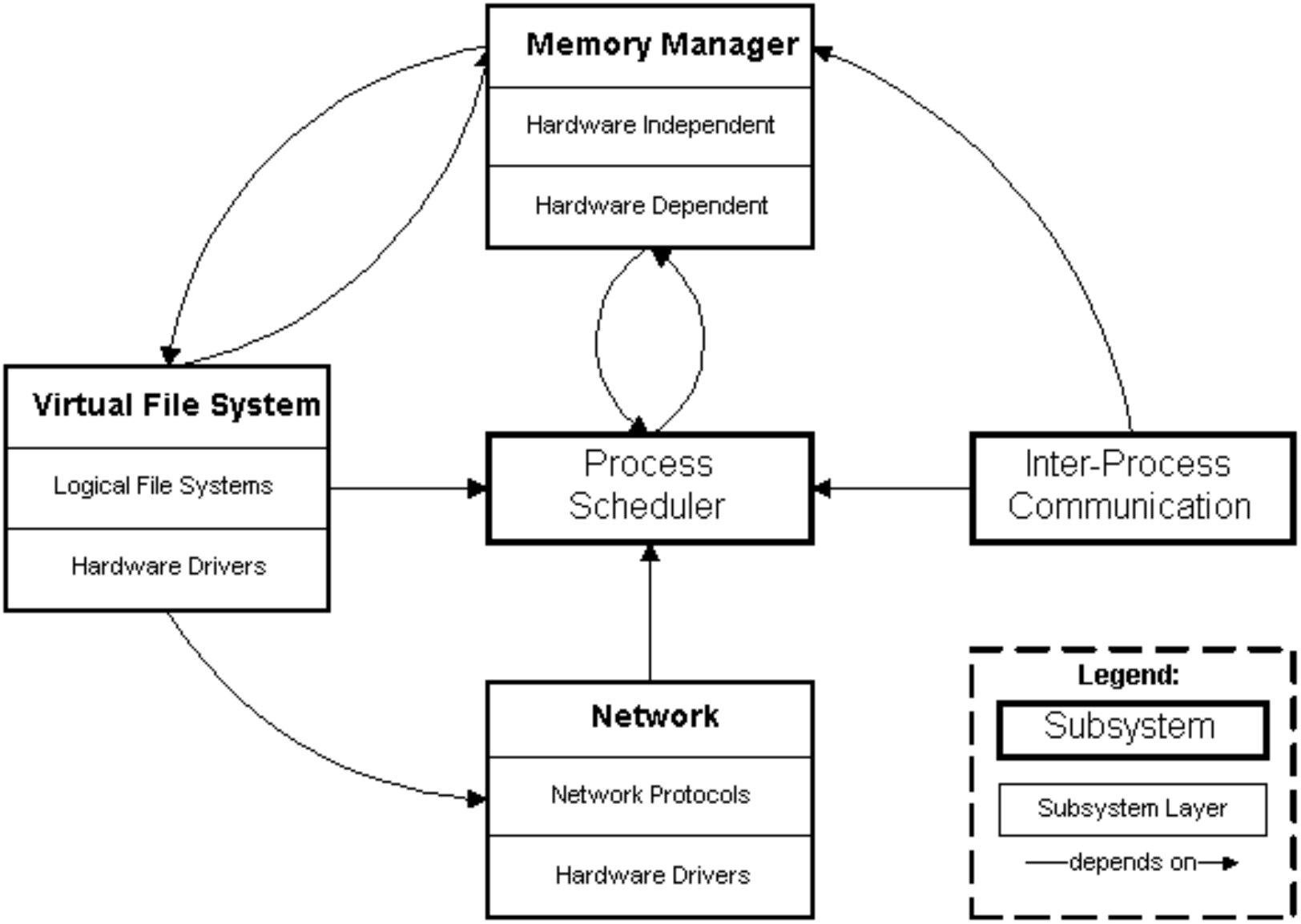
JE

Estructura del Kernel

El kernel está conformado por 5 grandes subsistemas.

- El planificador de procesos (**sched**).
- El administrador de memoria (**mm**).
- El sistema del archivo virtual (**vfs**).
- La interfaz de red (**net**).
- La comunicación entre procesos (**ipc**).

Descomposición Conceptual



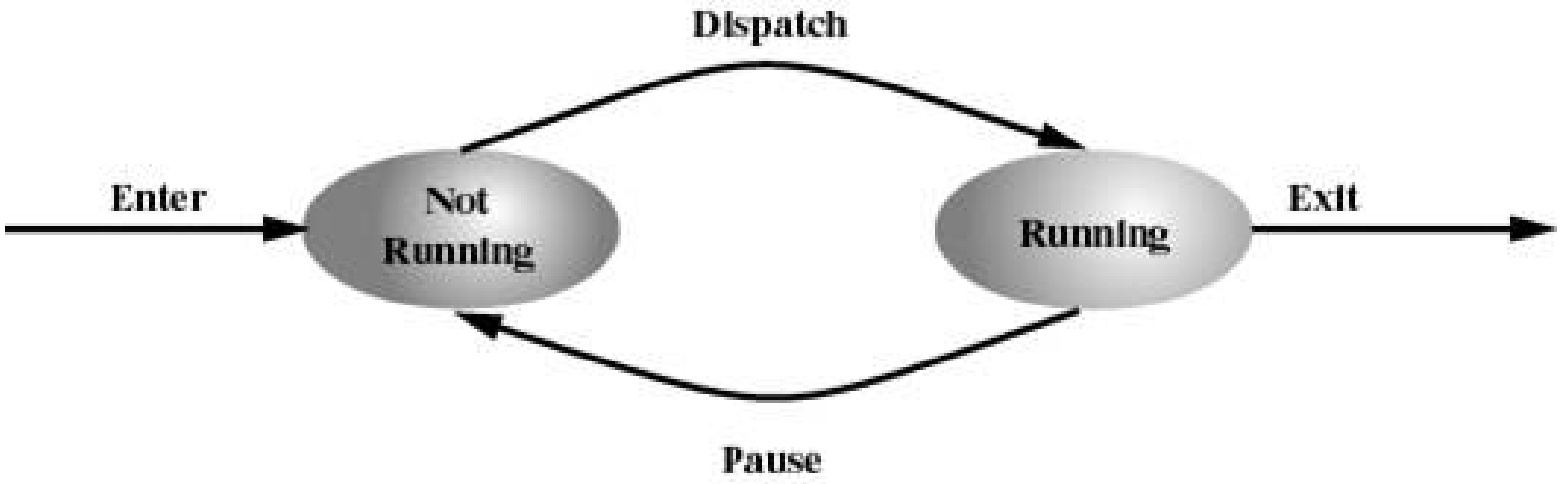
Concepto de Proceso

Un SO ejecuta una variedad de programas:

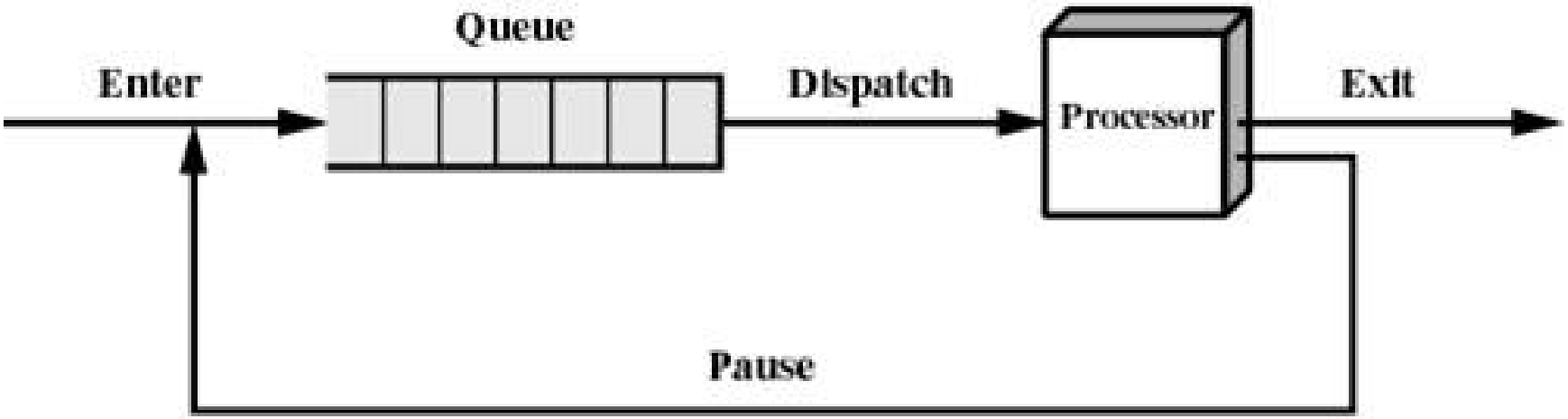
- Sistema Batch – jobs
- Sistemas de Tiempo Compartido – programas de usuario o tareas

- Los términos *job* y *proceso* se usan con similar sentido.
- **Proceso** – un programa en ejecución; la ejecución de los procesos debe progresar en forma secuencial.
- Un proceso incluye:
  - contador de programa
  - stack
  - sección de datos

Modelo de Procesos con Dos Estados

- El SO controla la ejecución de los procesos (se necesita saber en qué estado se encuentra cada uno).
- Cada proceso puede tener dos estados:
  - En ejecución.
- 

Cola de Procesos (dos estados)



Estado de un Proceso

- Mientras un proceso ejecuta, cambia de *estado*.
  - nuevo: el proceso es creado.
  - corriendo: las instrucciones están siendo ejecutadas.
  - espera/bloqueado: el proceso está esperando que ocurra algún evento.
  - listo: el proceso está esperando ser asignado a la CPU.
  - terminado: el proceso ha finalizado su ejecución.

Diagrama de Estados de un Proceso (1)

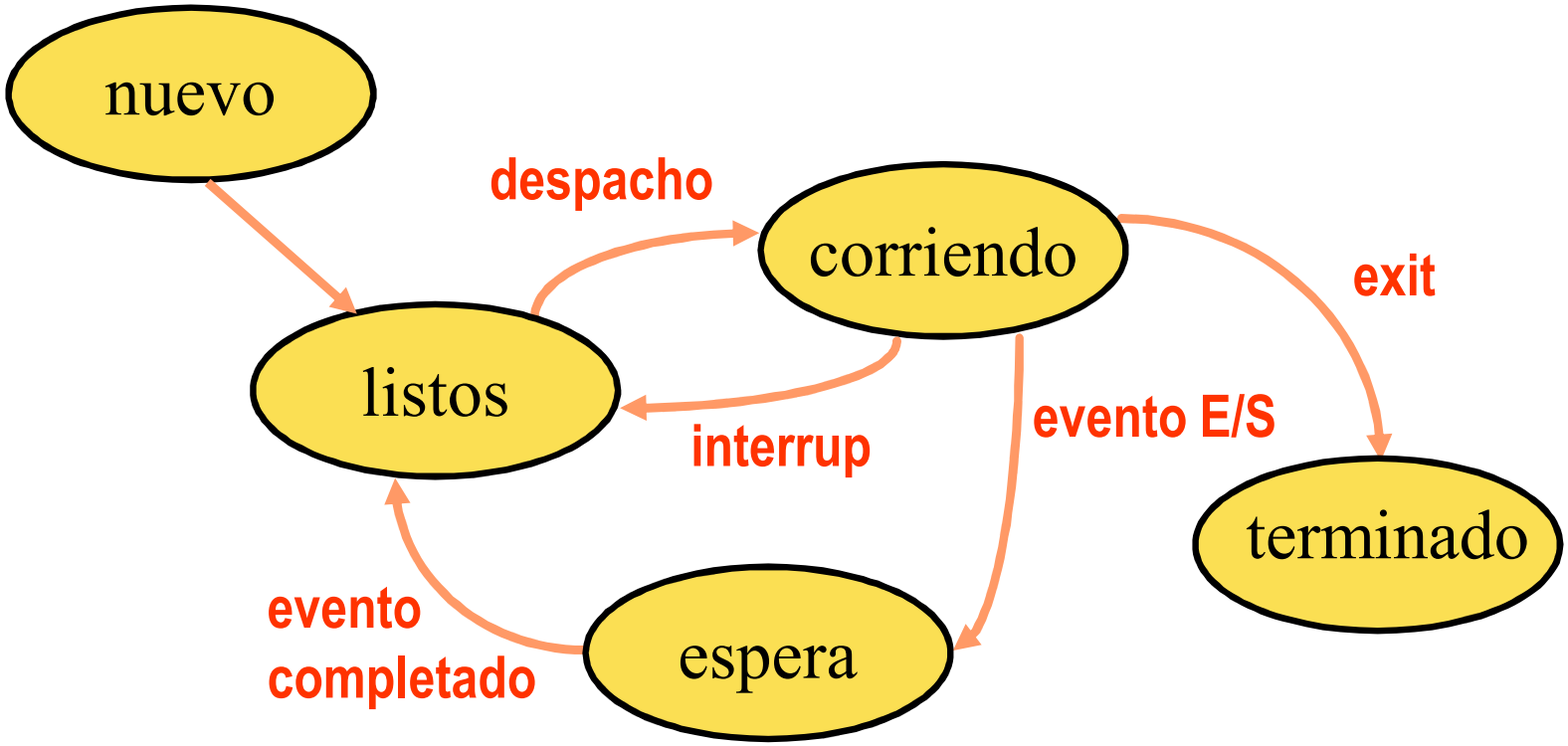
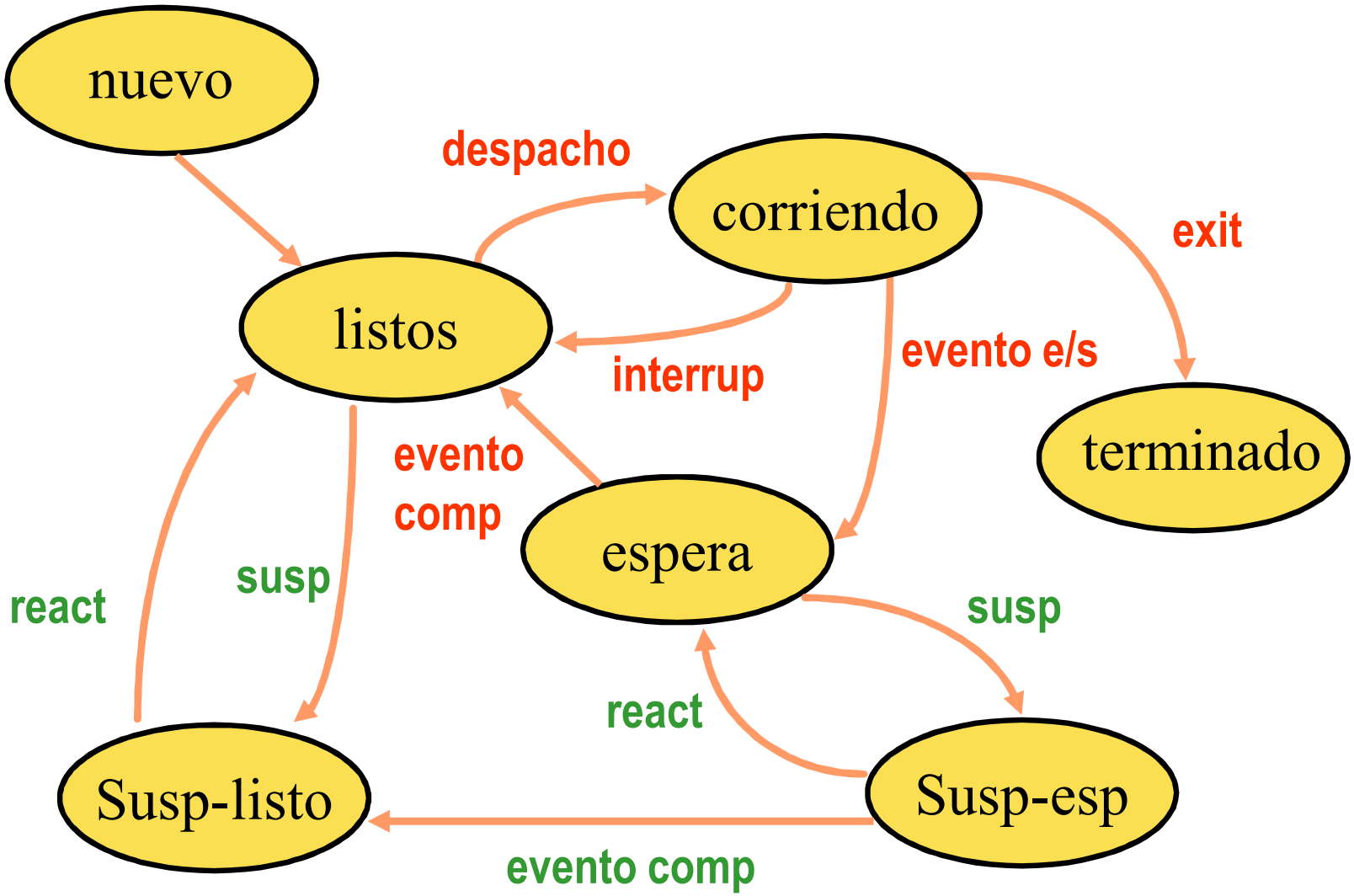


Diagrama de Estados de un Proceso (2)

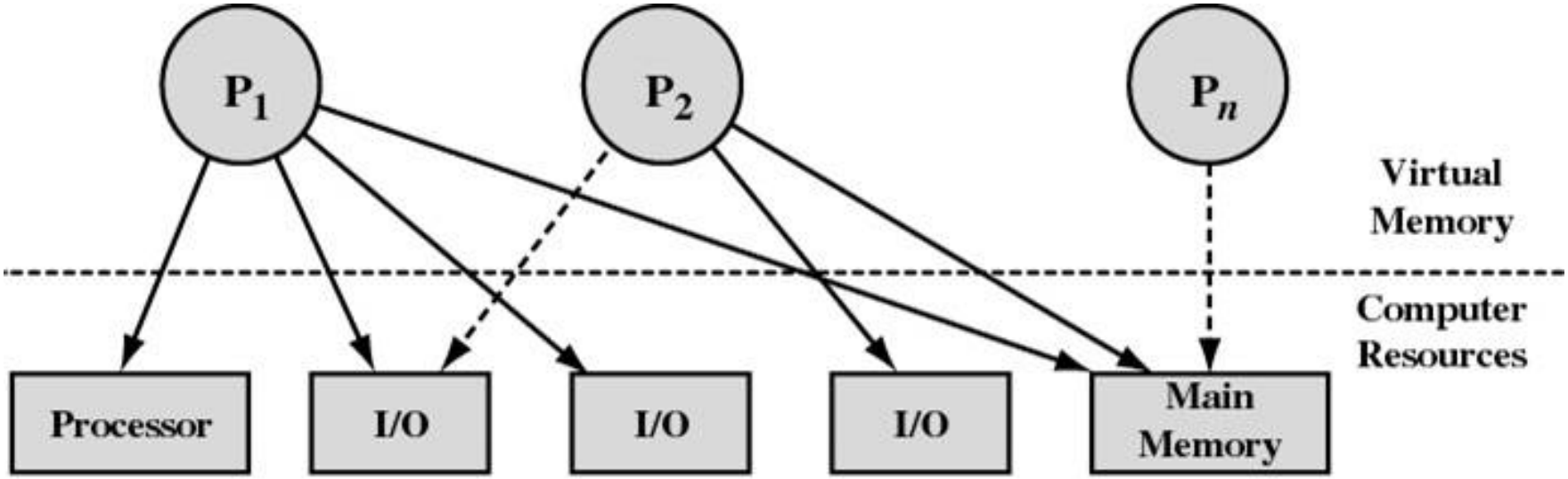


Razones para suspender procesos

- Intercambio (swapping).
  - El SO necesita liberar suficiente memoria RAM para cargar un nuevo proceso.
- Otra razón del SO.
  - El SO puede suspender un proceso que se sospecha causa un problema.
- Solicitud del usuario.
- Por tiempo.
  - Se ejecuta con cierta frecuencia, entonces mientras no se usa se suspende.
- Solicitud del proceso padre.
  - El padre desea suspenderlo para examinar o modificar el proceso o para coordinar con otros procesos.

Descripción de Proceso

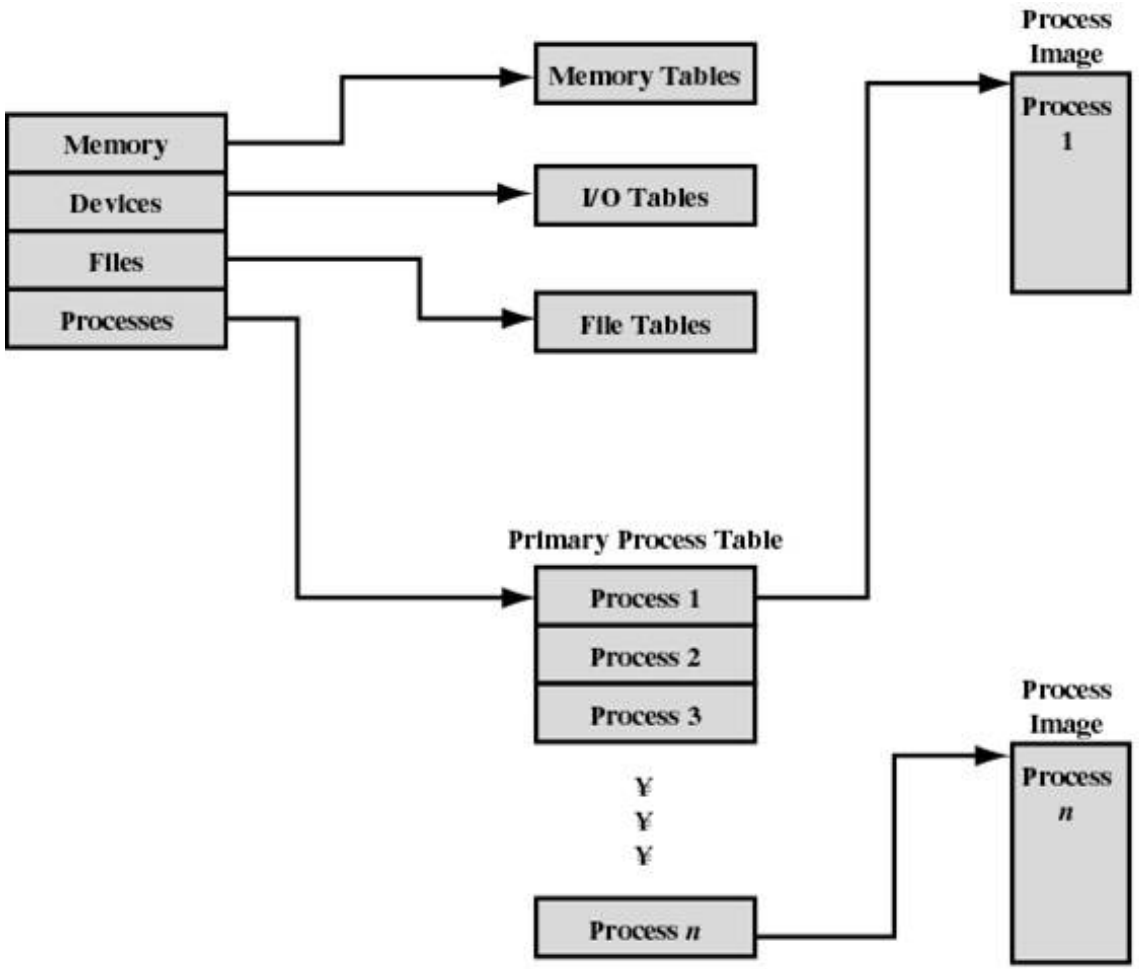
- En un entorno multiprogramado muchos procesos requieren y están haciendo uso de recursos.



- ¿Qué información necesita el SO para controlar los procesos y administrar los recursos?

Estructuras de Control del SO

- Para administrar todo lo que pasa en el sistema, el SO construye y mantiene tablas de información de cada entidad que esté administrando.



Tablas de Memoria

- Se utiliza para administrar la memoria virtual y la memoria real.
  - Asignación de memoria principal a los procesos.
  - Asignación de memoria secundaria a los procesos.
  - Atributos de protección para acceso a regiones de memoria compartida.
  - Información necesaria para administrar la memoria virtual.

Tablas de E/S

- Se utiliza para administrar los dispositivos y canales DES:
  - Estado del DES: disponible o asignado.
  - Estado de una operación con el DES.
  - Ubicación en la memoria principal que ha sido usada como fuente o destino de una operación de E/S.

Tabla de Archivos

- Existencia de archivos.
  - Ubicación en la memoria secundaria.
  - Estado actual.
  - Atributos.
  - A veces esta información es mantenida por el sistema de administración de archivos (*file-management system*).

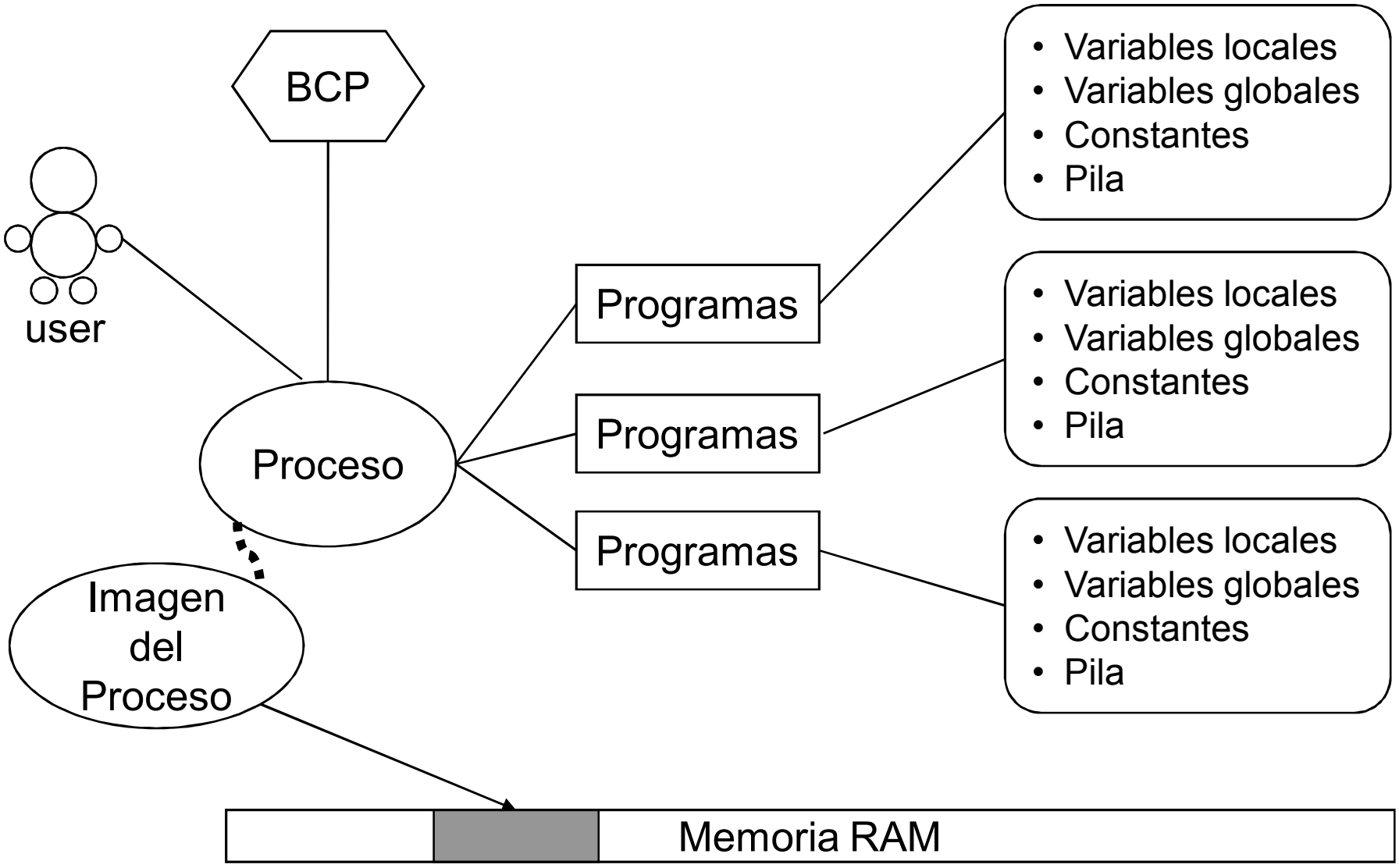
Tabla de Procesos

- Permite administrar la información de cada proceso
  - Donde está ubicado - proceso en memoria
    - imagen del proceso
  - Atributos necesarios para este administrador.
    - Process ID.
    - Estado del proceso.
    - Ubicación en memoria.

Estructuras de Control

- Para que el SO administre los proceso debe de conocer:
  - **Ubicación de proceso.**
  - **Atributos.**

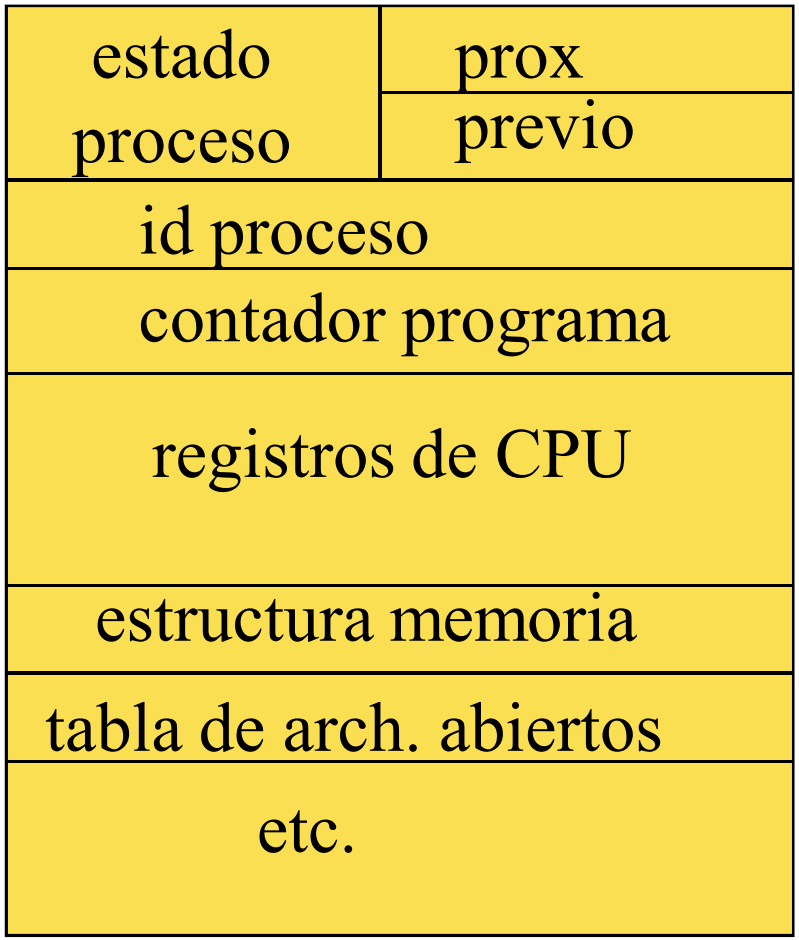
Ubicación de los Procesos



Atributos

- Bloque de Control de Proceso (BCP, PCB en ing.):
  - Identificación del proceso.
    - Identificadores.
  - Información del estado del procesador.
    - Registros visibles para el usuario.
    - Registro de control y de estado.
    - Punteros de pila.
  - Información de control del proceso.
    - Información de planificación y de estado.
    - Estructuración de datos.
    - Comunicación entre procesos.
    - Privilegios de los procesos.
    - Gestión de memoria.
    - Propiedad de los recursos y utilización.

Bloque de Control de Proceso (PCB)



Identificación del proceso

- Identificadores
  - Identificador del proceso.
  - Identificador del proceso que creó a este proceso (padre).
  - Identificador del usuario.

Información del estado del procesador

- Registros visibles al usuario
  - Un registro visible para el usuario es aquel que puede hacerse referencia por medio de un lenguaje de máquina que ejecuta el procesador.
  - Normalmente, existen entre 8 a 32 de estos registros, aunque algunas implementaciones RISC tienen más de 100.
- Registros de Control y de Estado.

  Son registros del procesador para controlar su funcionamiento:
  - Contador de programa. Siguiente instrucción.
  - Códigos de condición. Resultado de la operación aritmética más reciente (signo, cero, carry, overflow, paridad, …)
  - Información de estado. Habilitación e inhabilitación de interrupciones y el modo de ejecución.
  - PSW. Palabra de estado de programa. Códigos de condición.

Palabra de estado de programa (PSW)

- La PSW almacena información pertinente sobre el programa que esté ejecutándose.
  - Códigos de condición.
  - Indicadores de habilitación de traps.
  - Nivel de prioridad de interrupciones.
  - Modo previo.
  - Modo actual.
  - Pila de interrupciones.
  - Primera parte hecha (donde se quedó).
  - Traza pendiente (debug).

Información del estado del procesador

- Punteros de Pila.
  - Cada proceso tiene una o más pilas FIFO del sistema asociadas.
  - Las pilas se utilizan para almacenar los parámetros y las direcciones de retorno de los procedimientos y de las llamadas al sistema.
  - El puntero de pila (SP) siempre apunta al tope de la pila.
  - También se suele contar con FP.

Información de control del procesador (1)

- Información de Planificación y de Estado
  - Esta es la información que necesita el SO para llevar a cabo sus funciones de planificador:
  - **Estado del proceso**. Define la disposición del proceso para ser planificado para ejecutar (en ejecución, listo, esperando, suspendido).
  - **Prioridad**. Se puede usar con uno o más campos para describir la prioridad de planificación de los procesos. (pueden ser omisión, actual, la más alta permitida).
  - **Información de planificación**. Depende del algoritmo de planificación utilizado (tiempo de espera, tiempo de ejecución).
  - **Eventos**. Identidad del evento que el proceso esta esperando antes de reanudarse.

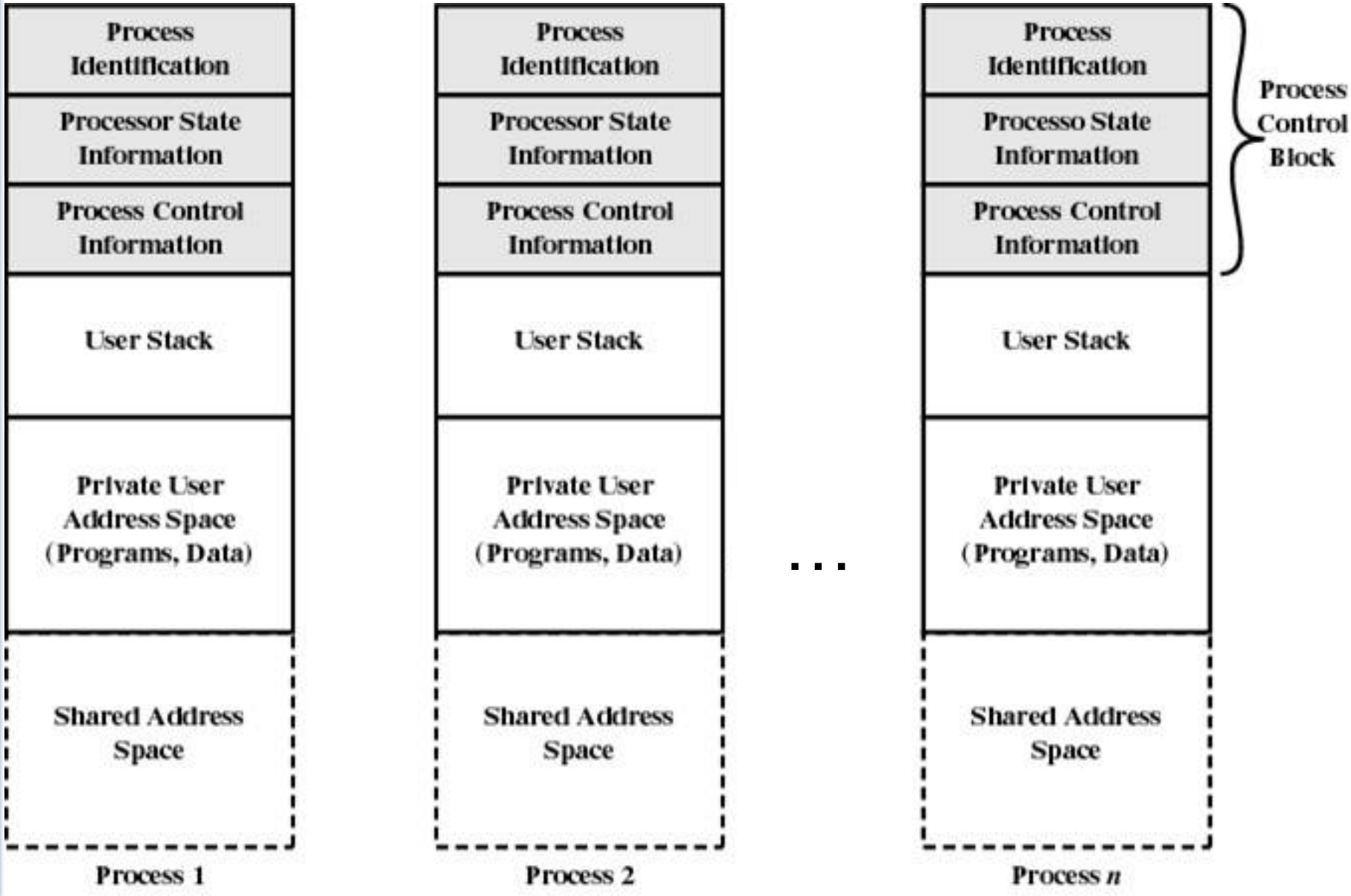
Información de control del procesador (2)

- Estructuras de datos.
  - Un proceso puede estar enlazado con otros procesos en una cola, un anillo o alguna otra estructura.
  - Por ejemplo todos los procesos que están en estado de espera de un nivel determinado de prioridad pueden estar enlazados en una cola.
  - Cada proceso mantiene una relación padre-hijo (creador-creado) con otro proceso. PPID de INIT?
  - El BCP puede contener punteros a otros procesos para dar soporte a estas estructuras.
- Comunicación entre procesos.
  - Puede haber varios indicadores, señales y mensajes asociados con la comunicación entre dos procesos independientes.
  - Una parte de esta información (o su totalidad) se puede almacenar en el BCP.

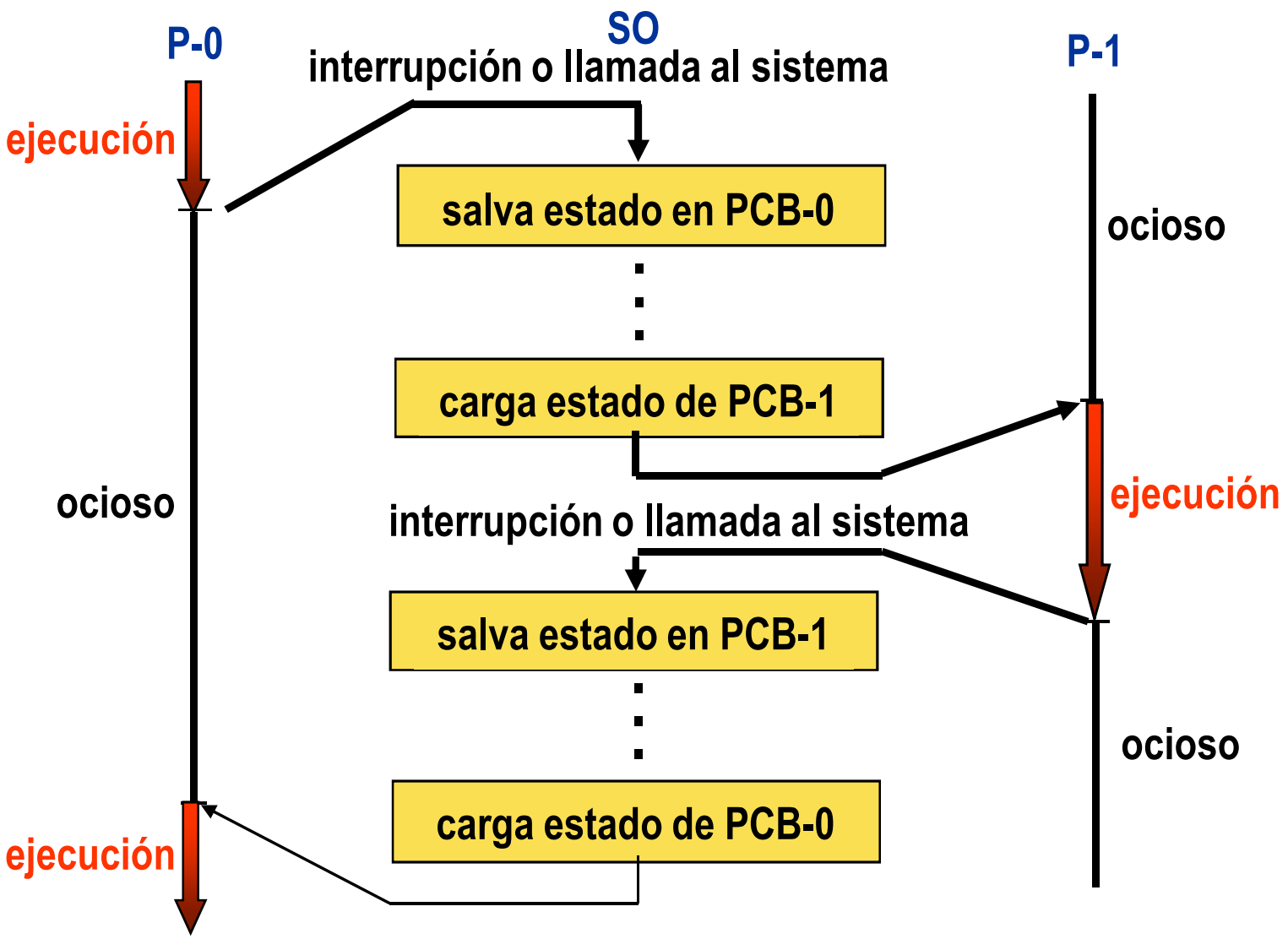
Información de control del procesador (3)

- Privilegios de los procesos.
  - A los procesos se les otorgan privilegios en términos de memoria a la que pueden acceder y el tipo de instrucciones que pueden ejecutar. Además, también se pueden aplicar privilegios al uso de los servicios y utilidades del sistema.
- Gestión de memoria.
  - Esta sección puede incluir punteros a las tablas de páginas y/o segmentos que especifican la memoria virtual asignada.
- Propiedad de los recursos y utilización.
  - Se pueden incluir los recursos controlados por el proceso, tales como los archivos abiertos.
  - Puede ser el histórico de la utilización del procesador o de otros recursos.
  - Información necesaria para el planificador.

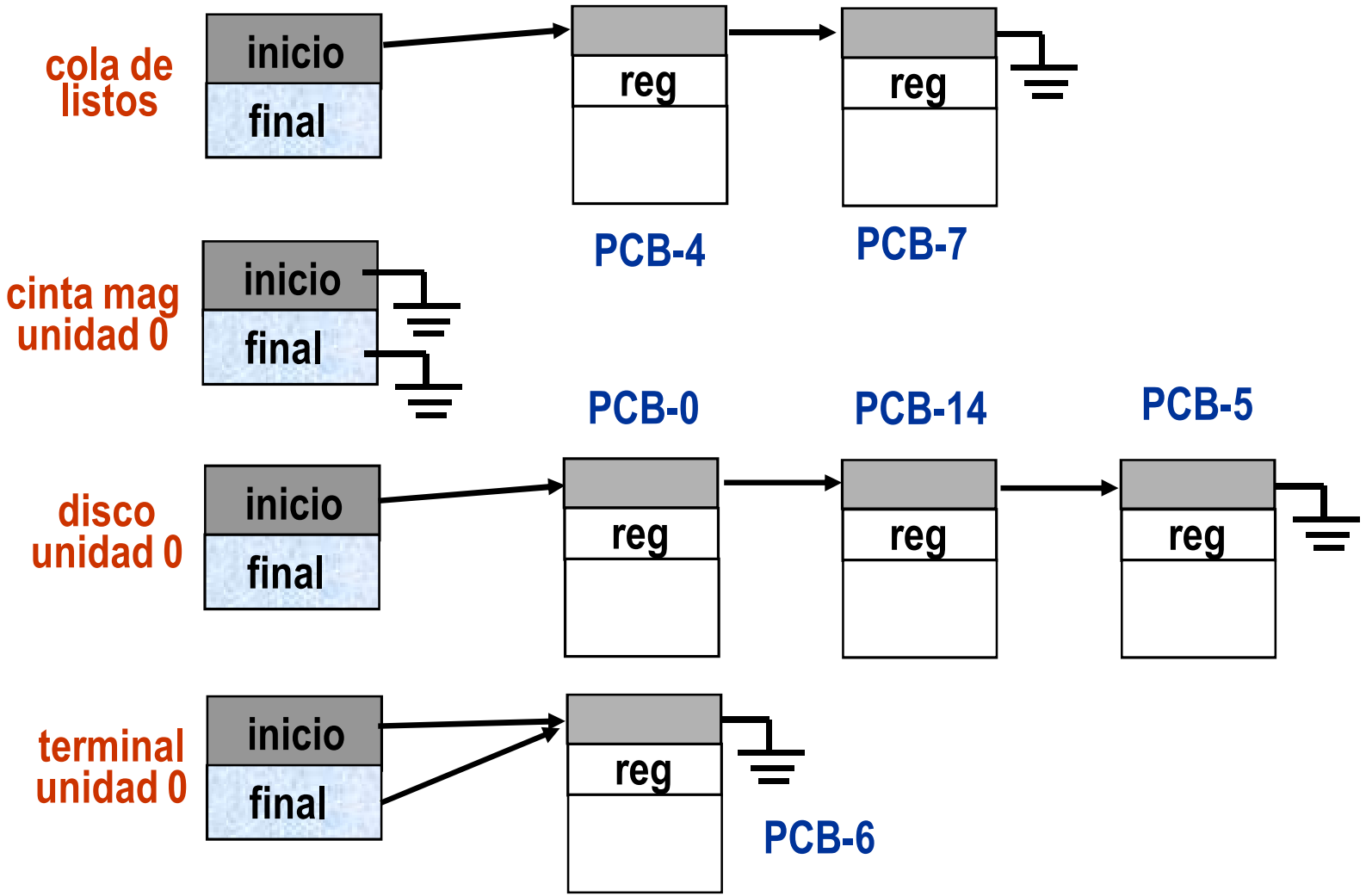
Imagen de un proceso en Memoria



Pasaje de CPU de un proceso a otro



Colas Listos y de Dispositivos I/O

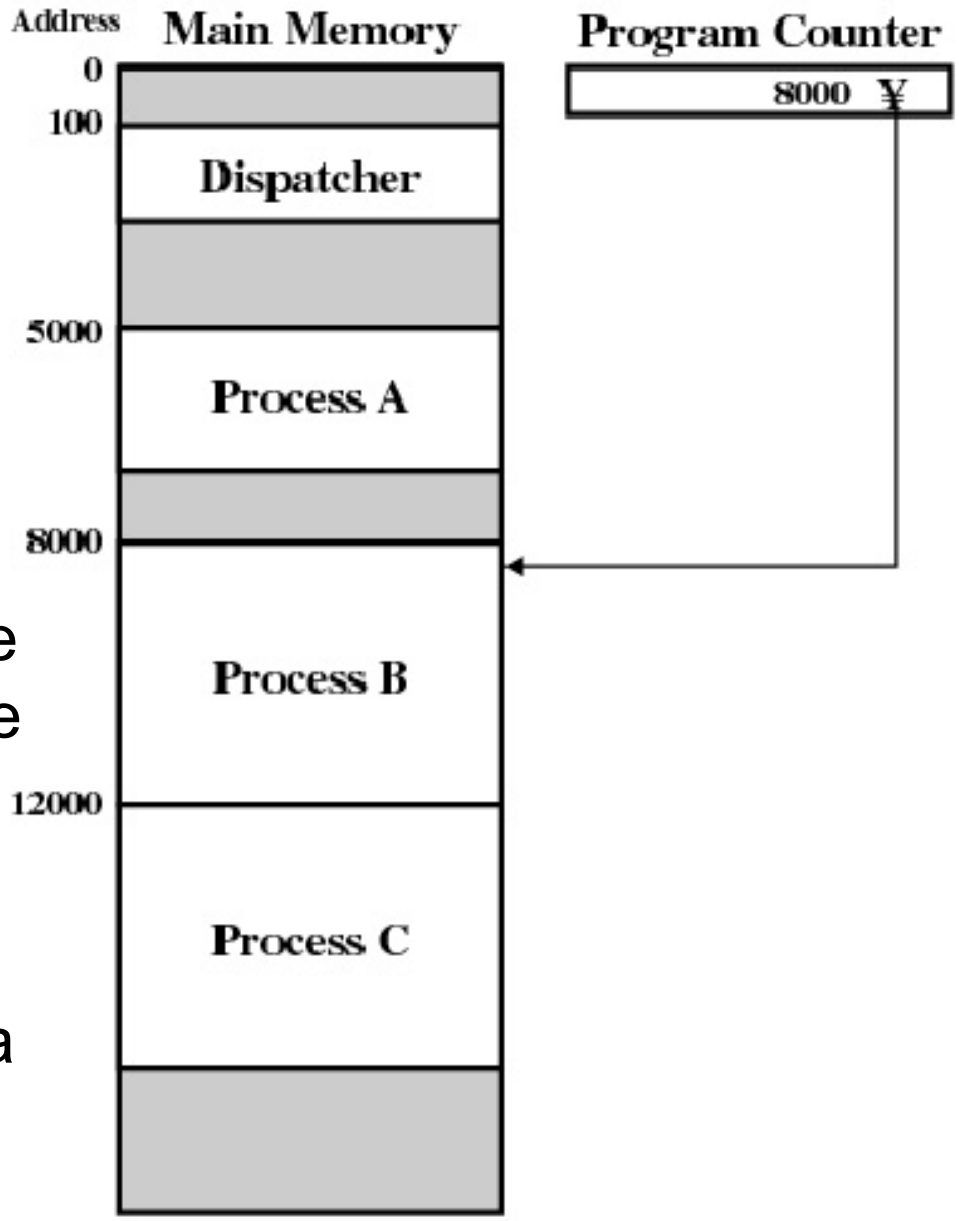


Administración de Procesos

- **Scheduler:** agrega (nuevos) y elimina procesos (terminados) a la tabla de procesos.
- **Dispatcher (planificador de corto término):** controla la asignación de *slices* de tiempo a los procesos referenciados en la tabla de procesos.
  - El final del slice se marca con una INT.
- Nótese que existen otras definiciones de scheduler / dispatcher.

Ejemplo de Procesos en mem

- No hay memoria virtual
- Tres programas cargados en memoria.
- Cada proceso representa a un programa.
- Existe un programa que asigna el procesador de un proceso a otro.
- Cada proceso tiene un tiempo de ejecución luego de lo cual ingresa el siguiente proceso



Traza de los tres procesos

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

```
1       5000
2       5001
3       5002
4       5003
5       5004
6       5005
--------------------Time out
7       100
8       101
9       102
10      103
11      104
12      105
13      8000
14      8001
15      8002
16      8003
-----------------I/O request
17      100
18      101
19      102
20      103
21      104
22      105
23      12000
24      12001
25      12002
26      12003
27      12004
28      12005
--------------------Time out
29      100
30      101
31      102
32      103
33      104
34      105
35      5006
36      5007
37      5008
38      5009
39      5010
40      5011
--------------------Time out
41      100
42      101
43      102
44      103
45      104
46      105
47      12006
48      12007
49      12008
50      12009
51      12010
52      12011
--------------------Time out
```

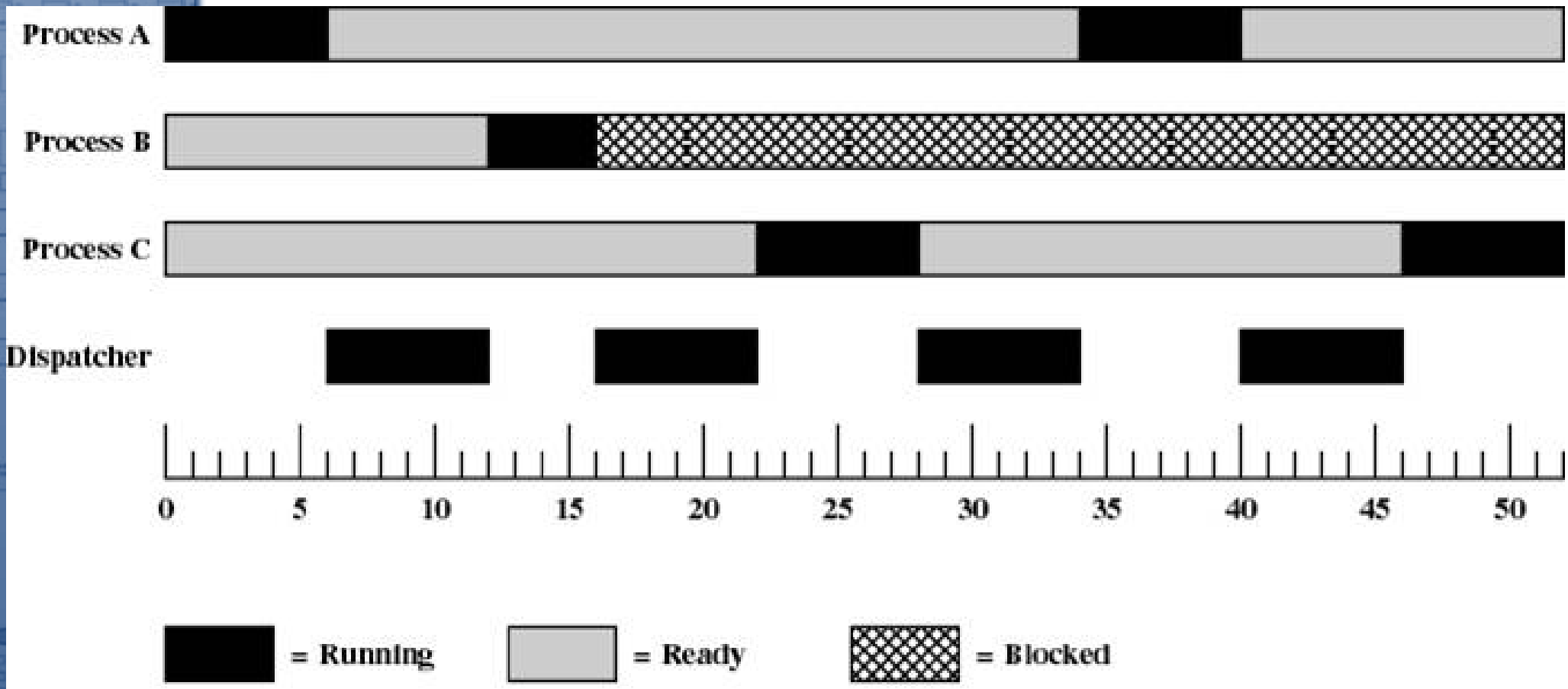
Traza de tres procesos

100 dirección de inicio para el programa distribuidor (dispatcher)

Las áreas sombreadas indican ejecución del proceso dispatcher.

La primera y tercera columna cuenta el ciclo de instrucción

La segunda y cuarta columna muestra la dirección de la que se está ejecutando.


Process A
Process B
Process C
Dispatcher
0
5
10
15
20
25
30
35
40
45
50
= Running
= Ready
= Blocked

Planificadores (1)

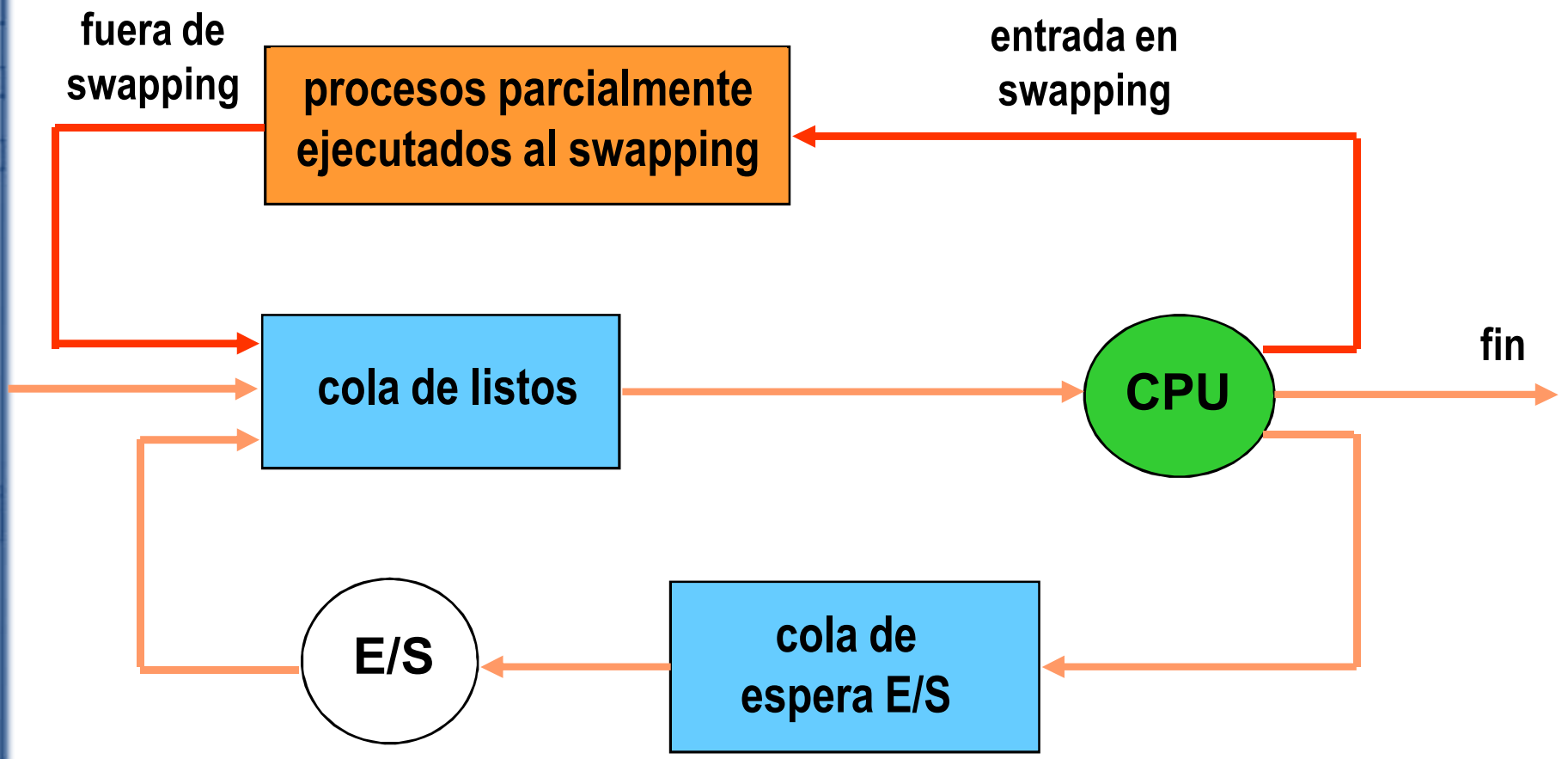
- Planificador de largo término

  (o planificador de jobs) – selecciona qué procesos deberían ser puestos en la cola de listos.
- Planificador de corto término

  (o planificador de CPU / dispatcher) – selecciona qué procesos deberían ser ejecutados próximamente por el CPU.

Planificadores (2)

Planificación de mediano término



Planificadores (3)

- El planificador de corto término es invocado muy frecuentemente (milisegundos) ⇒ (debe ser muy rápido).
- El planificador de largo término es invocado menos frecuentemente (segundos, minutos) ⇒ (puede ser lento).
- El planificador de largo término controla el *grado de multiprogramación*.
- Los procesos pueden ser descriptos como:
  - ***Procesos limitados por E/S*** – pasa más tiempo haciendo E/S que computaciones, ráfagas (burst) de CPU muy cortas.
  - ***Procesos limitados por CPU*** – pasa más tiempo haciendo computaciones que E/S, ráfagas (burst) de CPU muy largas.

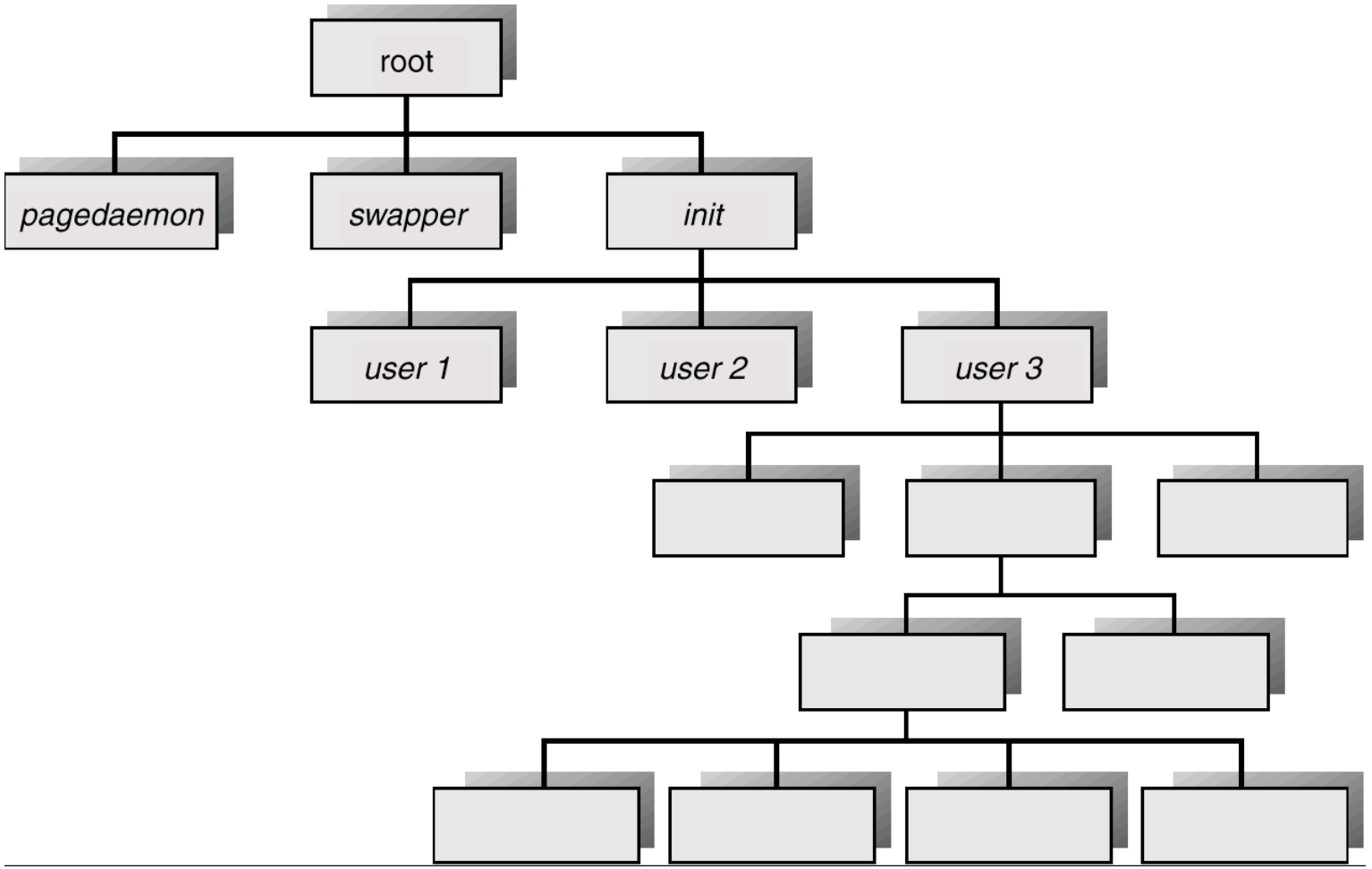
Cambio de contexto

- Cuando la CPU conmuta a otro proceso (*context switch*), el sistema debe salvar el estado del viejo proceso y cargar el estado para el nuevo proceso.
- El tiempo que lleva el cambio de contexto es *overhead*: el sistema no hace trabajo útil mientras está conmutando.
- El tiempo depende (entre otros) del soporte de hardware.

Creación de Procesos

- Procesos padres crean procesos hijos, los cuales, a su vez crean otros procesos hijos, formando un árbol de procesos (ver pstree).
- Recursos compartidos, 3 alternativas:
  1. Padres e hijos comparten todos los recursos.
  2. Los hijos comparten un subconjunto de los recursos de los padres.
  3. Padres e hijos no comparten ningún recurso.
- Ejecución, 2 alternativas:
  1. Padres e hijos ejecutan concurrentemente.
  2. Los padres esperan hasta que los hijos terminan.

Árbol de Procesos en UNIX



pstree sobre un servidor GNU/Linux

```
init─┬─amavisd───2*[amavisd]
     ├─auditd─┬─audispd───{audispd}
     │        └─{auditd}
     ├─avahi-daemon───avahi-daemon
     ├─brcm_iscsiuio───2*[{brcm_iscsiuio}]
     ├─clamd───{clamd}
     ├─courierlogger───authdaemond───5*[authdaemond]
     ├─crond
     ├─dbus-daemon
     ├─dovecot─┬─2*[dovecot-auth]
     │         ├─3*[imap-login]
     │         ├─pop3
     │         └─3*[pop3-login]
     ├─fail2ban-server───2*[{fail2ban-server}]
     ├─gam_server
     ├─gpm
     ├─httpd───20*[httpd]
     ├─2*[iscsid]
     ├─klogd
     ├─mailmanctl───8*[python]
     ├─master─┬─anvil
     │        ├─pickup
     │        ├─qmgr
     │        ├─smtpd
     │        ├─tlsmgr
     │        └─trivial-rewrite
     ├─6*[mingetty]
     ├─mysqld_safe───mysqld───29*[{mysqld}]
     ├─postgrey
     ├─saslauthd───4*[saslauthd]
     ├─spamd───2*[spamd]
     ├─sshd─┬─sshd───sshd───bash───pstree
     │      └─sshd───sshd───bash
     ├─syslogd
     ├─udevd
     ├─vmtoolsd
     └─vsftpd
```

Terminación de Procesos

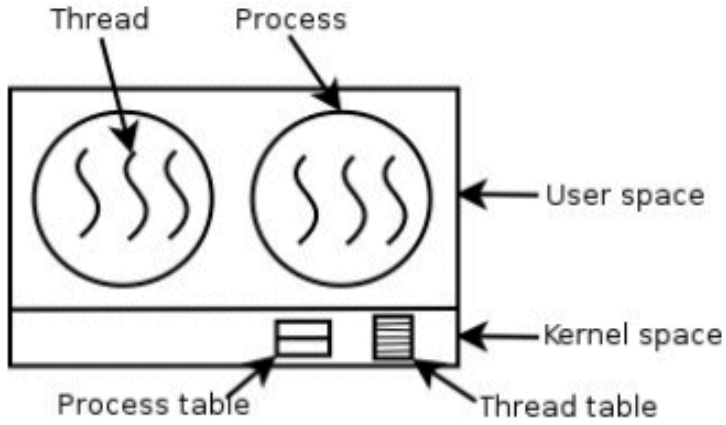
- El proceso ejecuta la última sentencia y espera a que el SO haga algo (**exit**).
  - Los datos de salida del hijo se pasan al padre (vía **wait**).
  - Los recursos de los procesos son liberados por el SO.
- El padre puede terminar la ejecución del proceso hijo (**abort**).
  - El hijo ha excedido los recursos asignados.
  - La tarea asignada al hijo ya no es requerida.
  - El padre está terminando.
    - El SO no permite (por default) a los hijos continuar si su padre termina.
    - Terminación en cascada.

Procesos Cooperativos

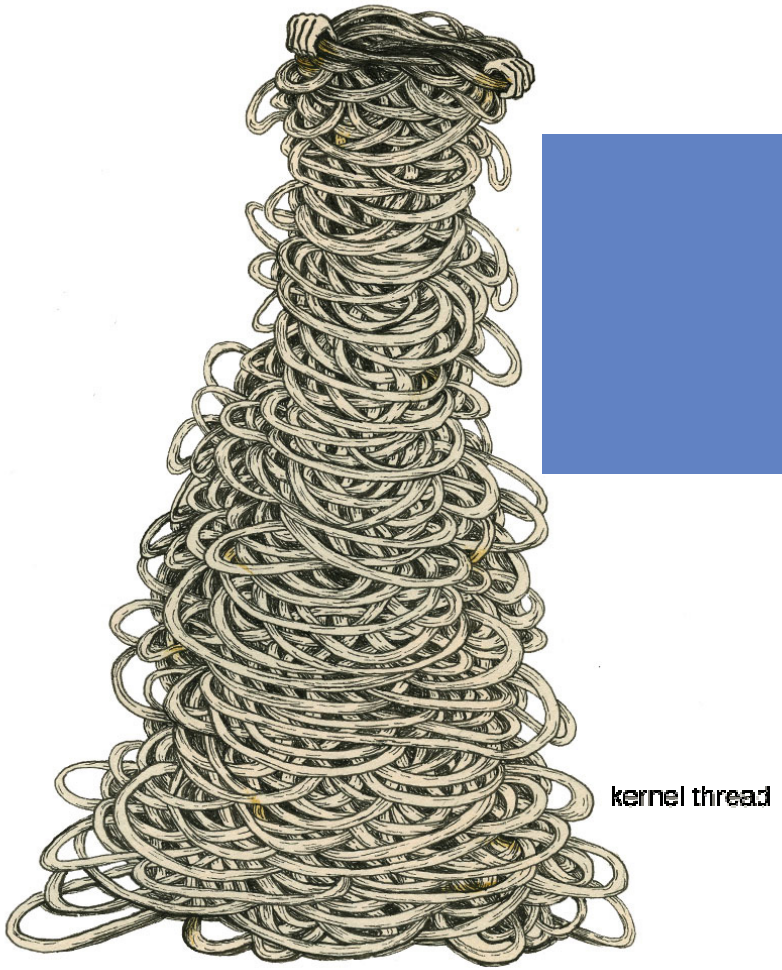
- Un proceso ***independiente*** no puede afectar ni ser afectado por la ejecución de otro proceso.
- Un proceso ***cooperativo*** puede afectar o ser afectado por la ejecución de otro proceso.
- Ventajas de los procesos cooperativos
  - Información compartida.
  - Aceleración de la computación.
  - Modularidad.
  - Conveniencia.

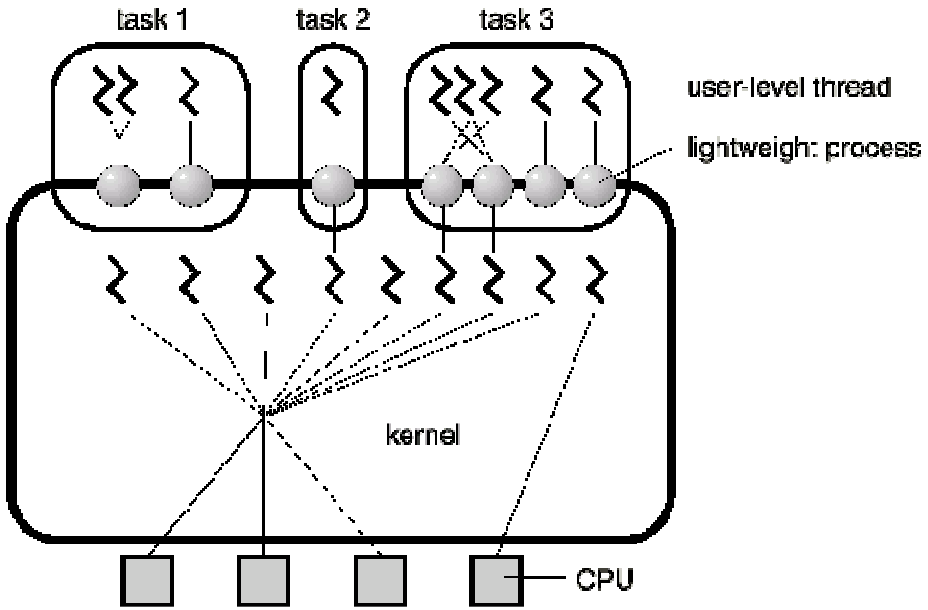
Problema del Productor-Consumidor

- Paradigma procesos cooperativos, el proceso ***productor*** produce información que es consumida por un proceso ***consumidor***.
  - *buffer ilimitado:* no tiene límites prácticos en el tamaño del buffer.
  - *buffer limitado:* supone que hay un tamaño fijo de buffer.



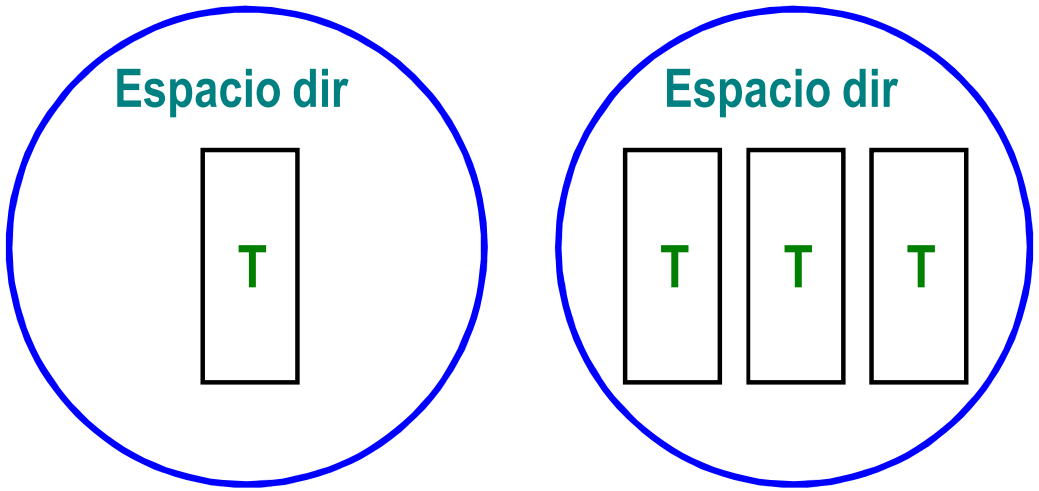
Hilos de Control





THREADS

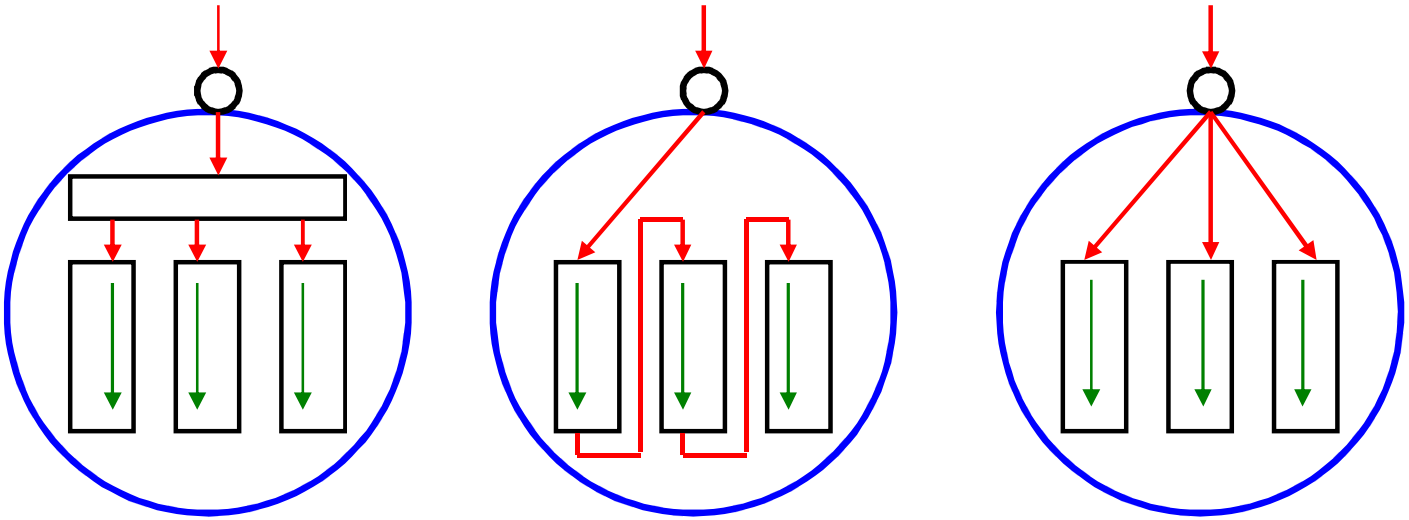
Uso de ***threads*** (*lightweight processes*)



- Igual espacio de direcciones.
- No hay protección entre los mismos (y no es necesaria).
- Estado, stack.

Modelos de Organización de Threads

- Modelo Despachador-Trabajador
- Modelo "Pipeline"
- Modelo "Team"



Procesos

Ejemplo: Servidor de Archivos

- Un *thread* simple.
- *Threads* múltiples.
- Máquina de estados finitos.

Modelo	Características
Threads	Paralelismo, llamadas al sistema bloqueantes
Proceso simple	No paralelismo, llamadas al sistema bloqueantes
Máquinas de Estados Finitos	Paralelismo, llamadas al sistema no bloqueantes

Motivaciones para su uso

1. La sobrecarga de crear un nuevo proceso es considerable frente a la creación de un nuevo *thread* dentro de un proceso.
2. La conmutación entre *threads* compartiendo el mismo espacio de direcciones es más “barata” que entre procesos.
3. Los *threads* permiten paralelismo al ser combinados con ejecución secuencial y llamadas a sistemas bloqueantes.
4. Compartir recursos puede hacerse más eficiente y natural entre *threads* de un proceso que entre procesos mismos a causa de compartir el mismo espacio de direcciones.

Aspectos de Diseño de Threads

Creación de threads

- estática
- dinámica

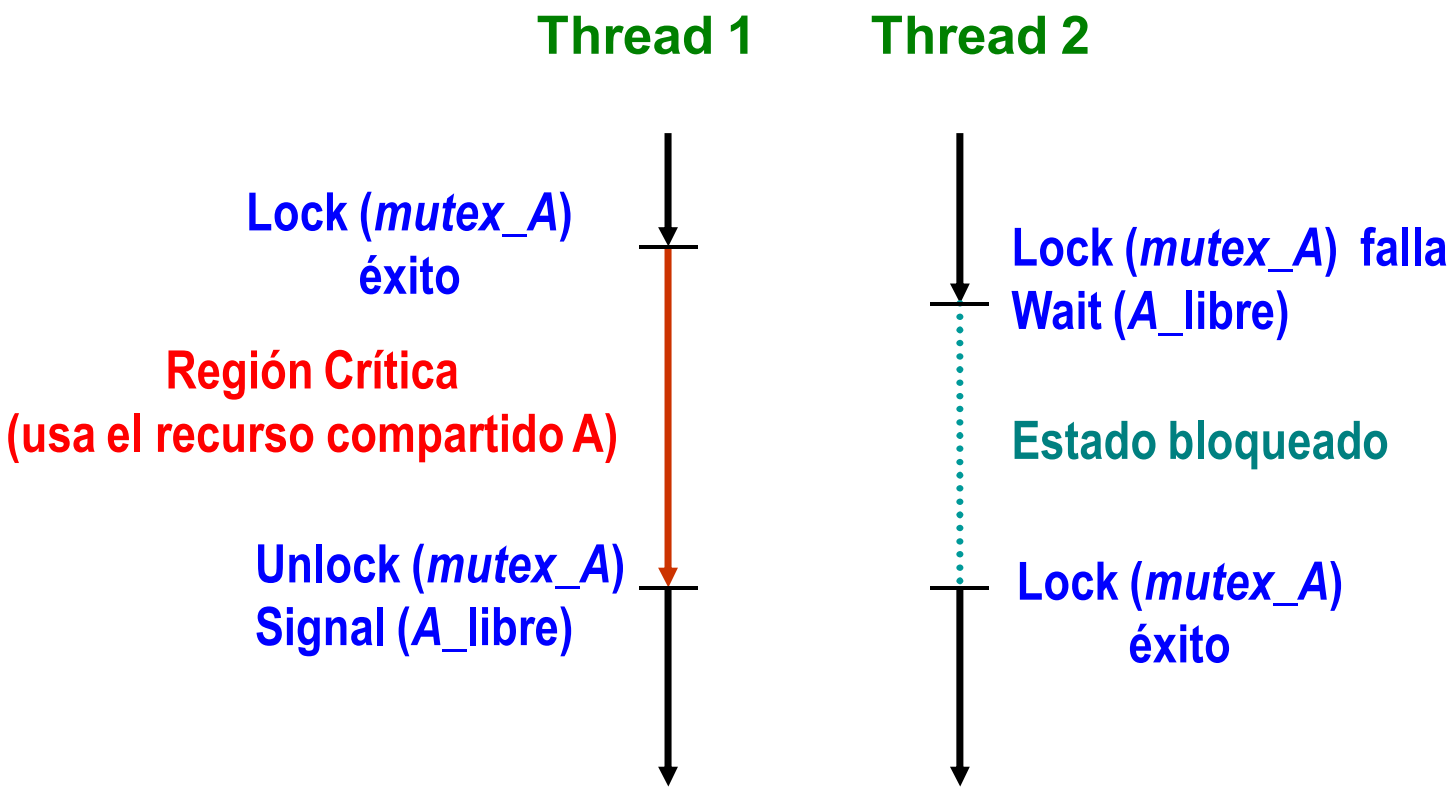
Terminación de threads

- pueden terminar convencionalmente
- pueden autodestruirse o ser destruidos

Sincronización de Threads

- ✓ usa variables *mutex*
- ✓ si encuentra *mutex=lock*
  - a) el *thread* se bloquea y espera en una cola por la variable *mutex*.
  - b) un *cod-status* indica la situación. El *thread* puede continuar con otra tarea o esperar.

Threads

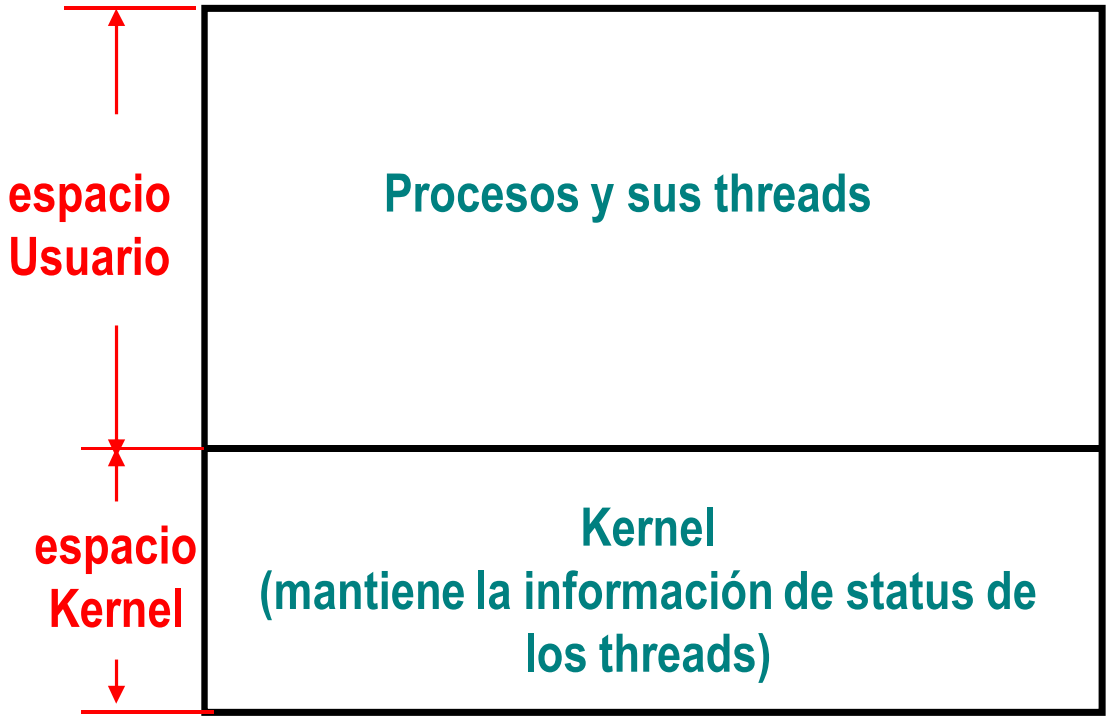

Thread 1
Thread 2
Lock (*mutex_A*)
éxito
Región Crítica
(usa el recurso compartido A)
Unlock (*mutex_A*)
Signal (*A_libre*)
Lock (*mutex_A*) falla
Wait (*A_libre*)
Estado bloqueado
Lock (*mutex_A*)
éxito

Planificación de Threads

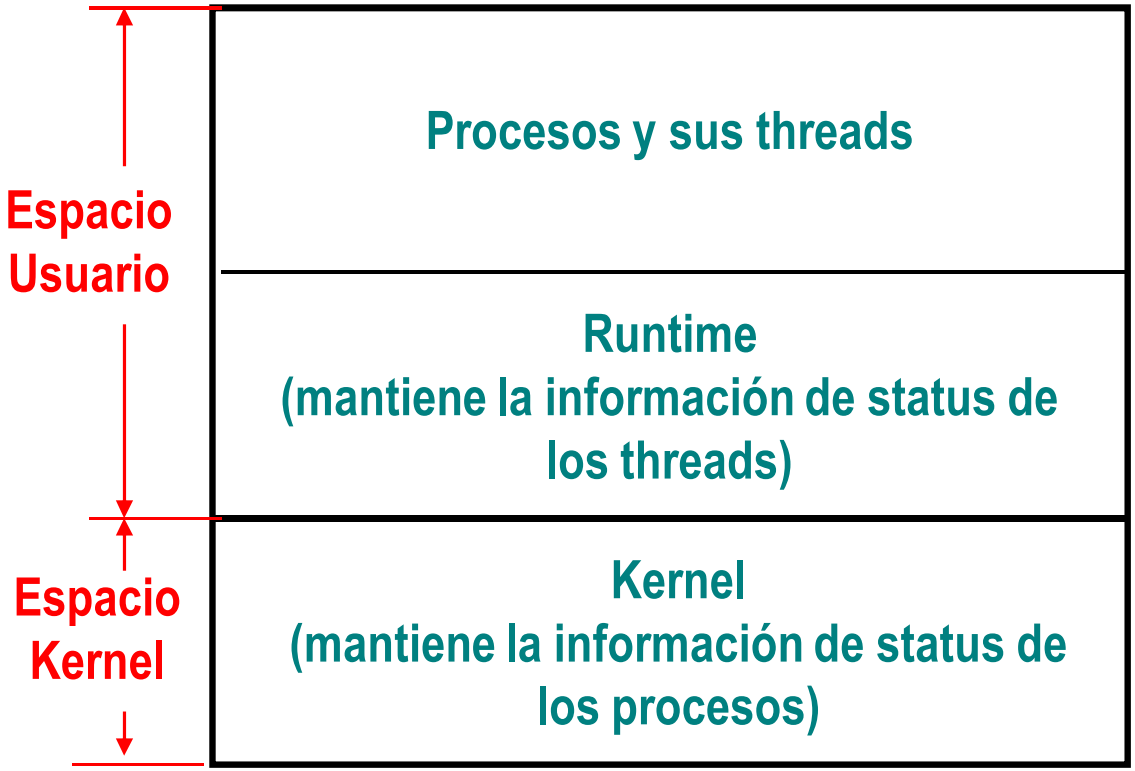
- ✓ facilidad para asignar prioridad.
- ✓ flexibilidad para cambiar el *quantum*.
- ✓ planificación forzada.
- ✓ planificación afín.

Implementación

Espacio del kernel



Espacio de Usuario



Ventajas y desventajas de las distintas alternativas (1)

- ✓ **Ventaja del nivel-usuario:** puede construirse encima del sistema operativo que no soporta *threads*. En el otro caso deben meterse en el *kernel*.
- ✓ **Ventaja del nivel-usuario:** puede usar su propio esquema de planificación. No es posible en el nivel-kernel.

Ventajas y desventajas de las distintas alternativas (2)

- ✓ Ventaja del nivel-usuario: el cambio de contexto de un *thread* en el nivel-usuario es más rápido que en el nivel-kernel. Es hecho por el *runtime*, en el nivel-kernel requiere un *trap*.
- ✓ Ventaja del nivel-usuario: la escalabilidad es mayor. Las tablas a nivel-kernel son mantenidas dentro del *kernel*.
- ✓ Desventaja del nivel-usuario: trabajan en multiprogramación pura.
- ✓ Desventaja del nivel-usuario: las *system-calls* generan problemas de bloqueo.

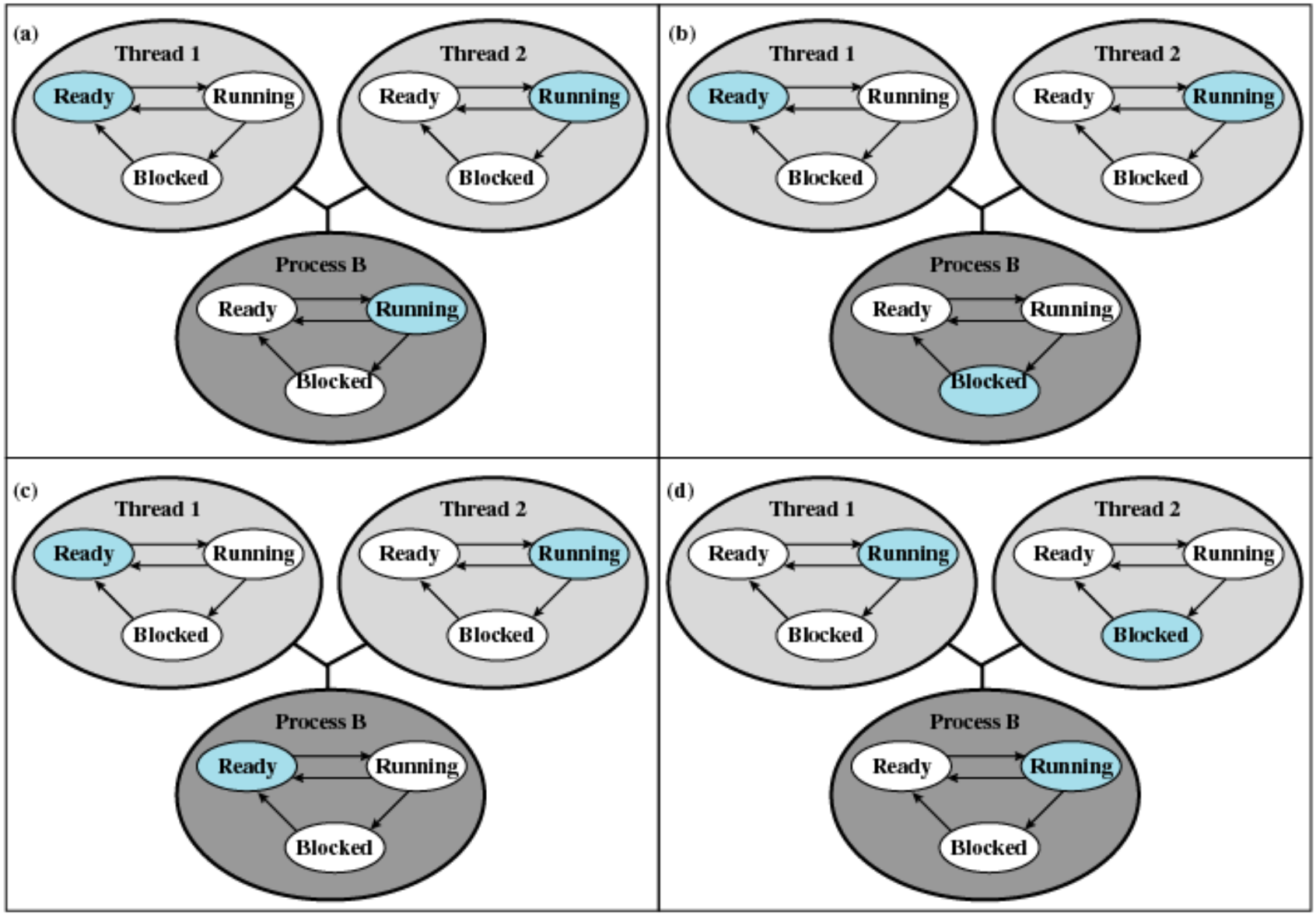
Beneficios del Uso de Threads

- Capacidad de Respuesta
- Compartir Recursos
  - Dado que los *threads* dentro de un mismo proceso comparten memoria y archivos, pueden comunicarse unos con otros sin invocar al kernel
- Economía
  - Toma menos tiempo crear un nuevo *thread* que un proceso
  - Menos tiempo terminar un *thread* que un proceso
  - Menos tiempo en conmutar entre dos *threads* dentro del mismo proceso
- Aprovechan las arquitecturas multiprocesador

Threads a Nivel de Usuario

- Manejo de threads hecho por librerías a nivel de usuario.
- Ejemplos
  - POSIX *Pthreads*
  - + Native POSIX Thread Lib (NPTL)
  - Mach *C-threads*
  - Solaris *threads*

Relaciones entre estados de threads nivel usuario y estados de procesos



1) La aplicación corriendo en el thread 2 hace una llamada a sistema que bloquea B (por ejemplo: E/S). Esto transfiere el control al kernel que pone al proceso B en estado bloqueado (espera) pero de acuerdo a la estructura de datos mantenida por la librería de threads, el thread 2 permanece en "corriendo". (b)

2) Cuando el proceso B ha agotado su tiempo pasa al estado de listo, pero de acuerdo a la estructura de datos mantenida por la librería de threads, el thread se mantiene en estado "corriendo". (c)

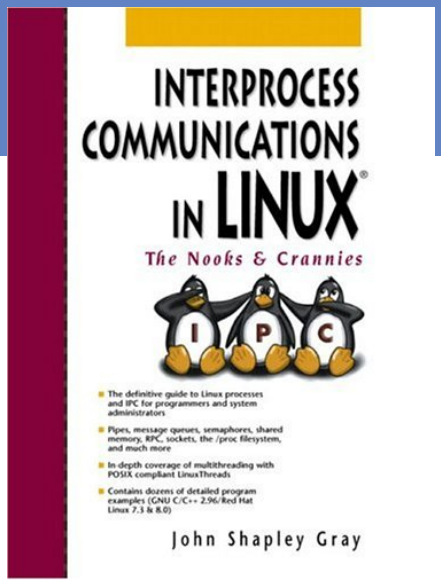
3) El thread 2 ha alcanzado un punto donde necesita alguna acción por parte del thread 1. El thread 2 entra en estado de bloqueado y el thread 1 pasa a "corriendo". El proceso B que los contiene se mantiene en estado "corriendo". (d)

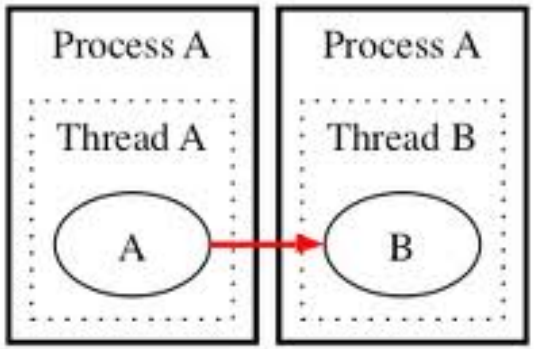
Threads a Nivel Kernel

- Soportados por el Kernel

- Ejemplos
  - Windows 98/2000/XP/Seven
  - Solaris
  - Tru64 UNIX

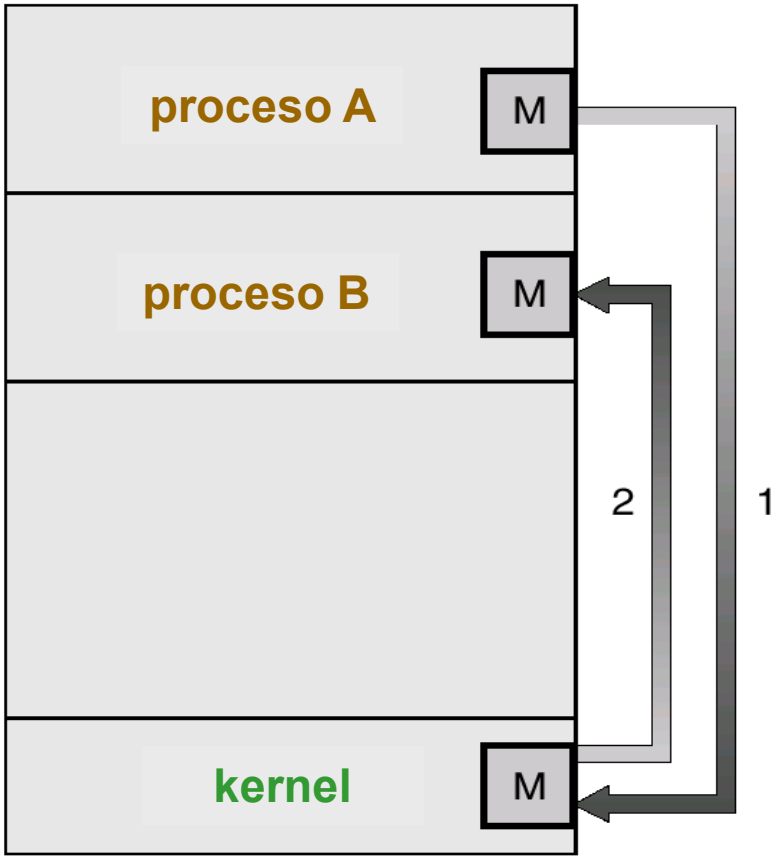
Comunicación entre Procesos (IPC)



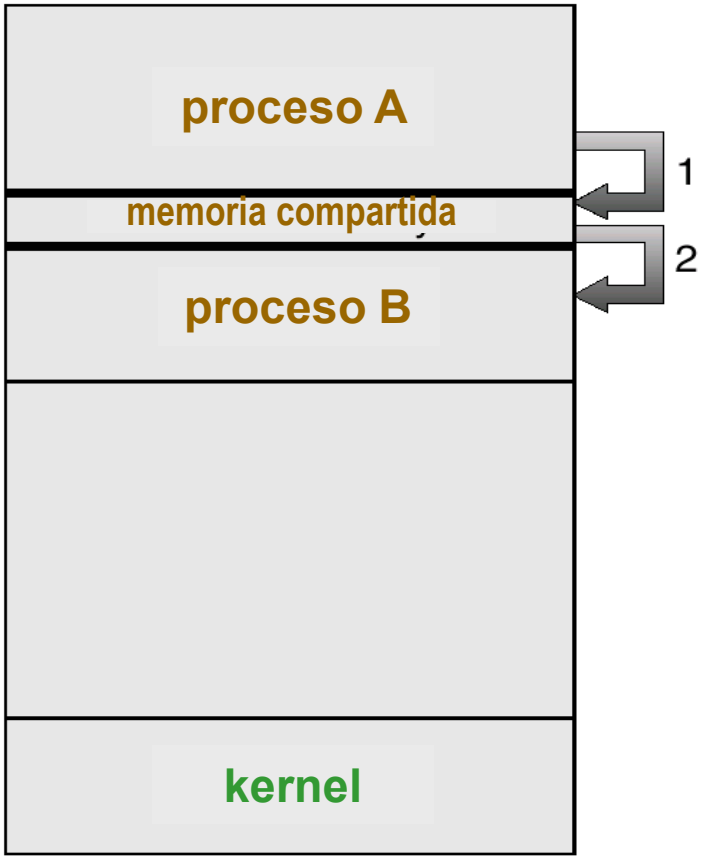


Modelos de Comunicación

Pasaje de mensajes



Memoria compartida



Comunicación entre procesos (IPC)

- Mecanismo de los procesos para comunicarse y sincronizar sus acciones.
- Sistema de mensajes – los procesos se comunican uno con otro sin necesidad de variables compartidas.
- Las facilidades de IPC provee dos operaciones:
  - **send** (*mensaje*) – mensaje de tamaño fijo o variable
  - **receive** (*mensaje*)
- Si *P* and *Q* desean comunicarse, necesitan:
  - Establecer un *vínculo de comunicación* entre ellos
  - Intercambiar mensajes via send/receive
- Implementación de un vínculo de comunicación
  - lógico (p.e., propiedades lógicas)
  - físico (p.e., memoria compartida, canal hardware)

Comunicación Directa

- Los procesos deben nombrar al otro explícitamente:
  - **send** (*P, mensaje*) – envía un mensaje al proceso P.
  - **receive** (*Q, mensaje*) – recibe un mensaje del proceso Q.
- Propiedades del vínculo de comunicación
  - Los vínculos son establecidos automáticamente.
  - Un vínculo está asociado con exactamente un par de procesos que se comunican.
  - Entre cada par existe exactamente un vínculo.
  - El vínculo puede ser unidireccional, pero es usualmente bidireccional.

Comunicación Indirecta

- Los mensajes son dirigidos y recibidos desde *mailboxes* (también conocidos como *ports*).
  - Cada mailbox tiene una única identificación.
  - Los procesos pueden comunicarse sólo si comparten un mailbox.
- Propiedades de un vínculo de comunicación
  - El vínculo se establece solo si los procesos comparten un mailbox común.
  - Un vínculo puede ser asociado con muchos procesos.
  - Cada par de procesos puede compartir varios vínculos de comunicación.
  - Los vínculos pueden ser unidireccionales o bidireccionales.

Sincronización

- El pasaje de mensajes puede ser bloqueante o no bloqueante.
- **Bloqueante** es considerado **sincrónico**
- **No bloqueante** es considerado **asincrónico**
- La primitivas ***send*** y ***receive*** pueden ser bloqueantes o no bloqueantes.

Buffering

- La cola de mensajes asociada al vínculo se puede implementar de tres maneras.
  1. **Capacidad 0 mensajes**
     El emisor debe esperar al receptor (***rendez-vous***).
  2. **Capacidad limitada** – longitud finita de *n* mensajes.
     El emisor debe esperar si el vínculo está lleno.
  3. **Capacidad ilimitada** – longitud infinita
     El emisor nunca espera.

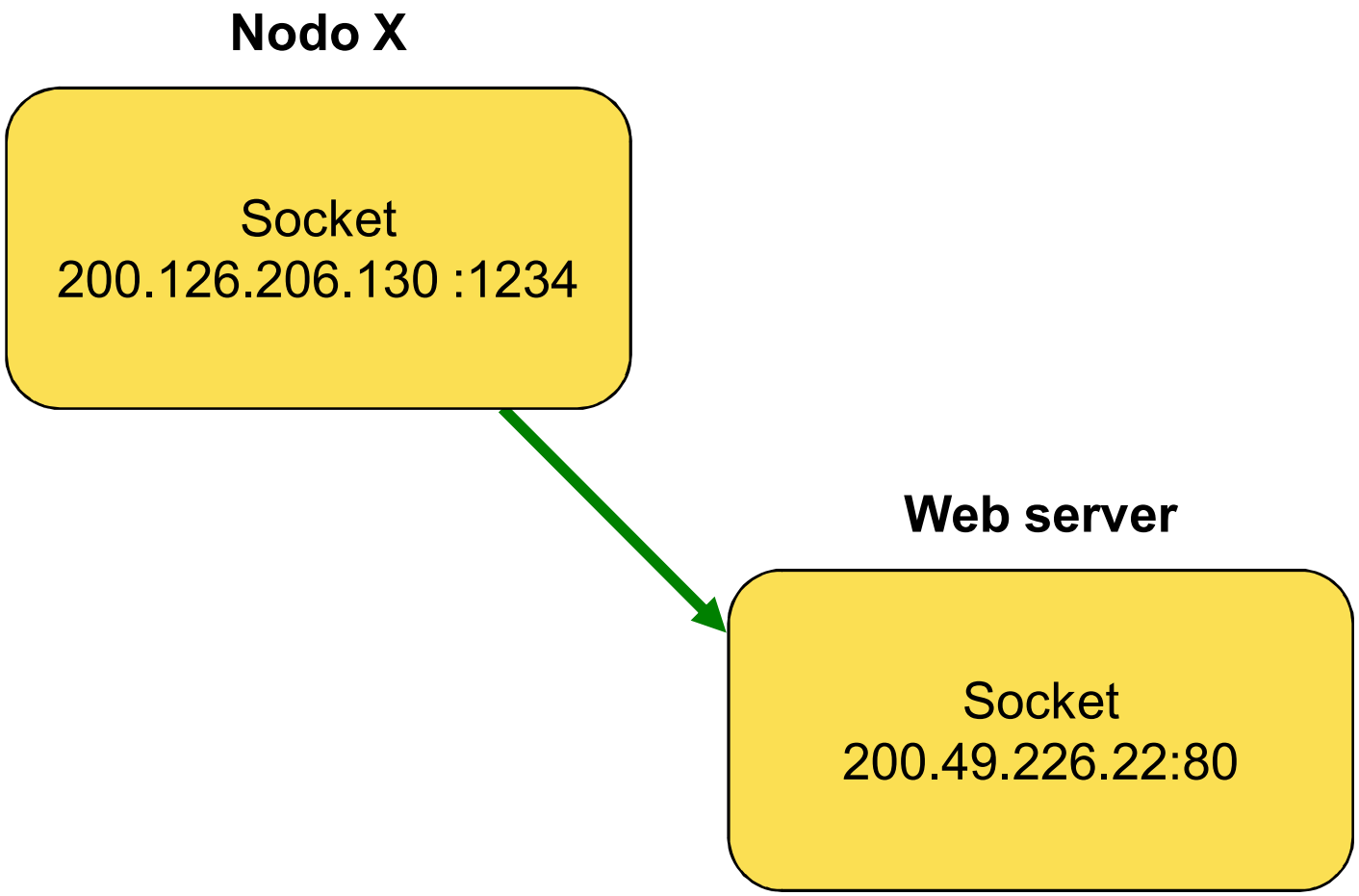
Comunicación Cliente-Servidor

- Sockets
- Llamadas a Procedimientos Remotors (**RPC**:Remote Procedure Calls)
- Invocación a Métodos Remotos (**RMI**:Remote Method Invocation (Java))

Sockets

- Un ***socket*** es definido como un *punto final de comunicación*.
- Concatenación de dirección IP y pórtico
- El socket 200.49.226.22:80 se refiere al port 80 en el host 200.49.226.22.
- La comunicación se lleva a cabo entre un par de sockets.

Comunicación por Sockets

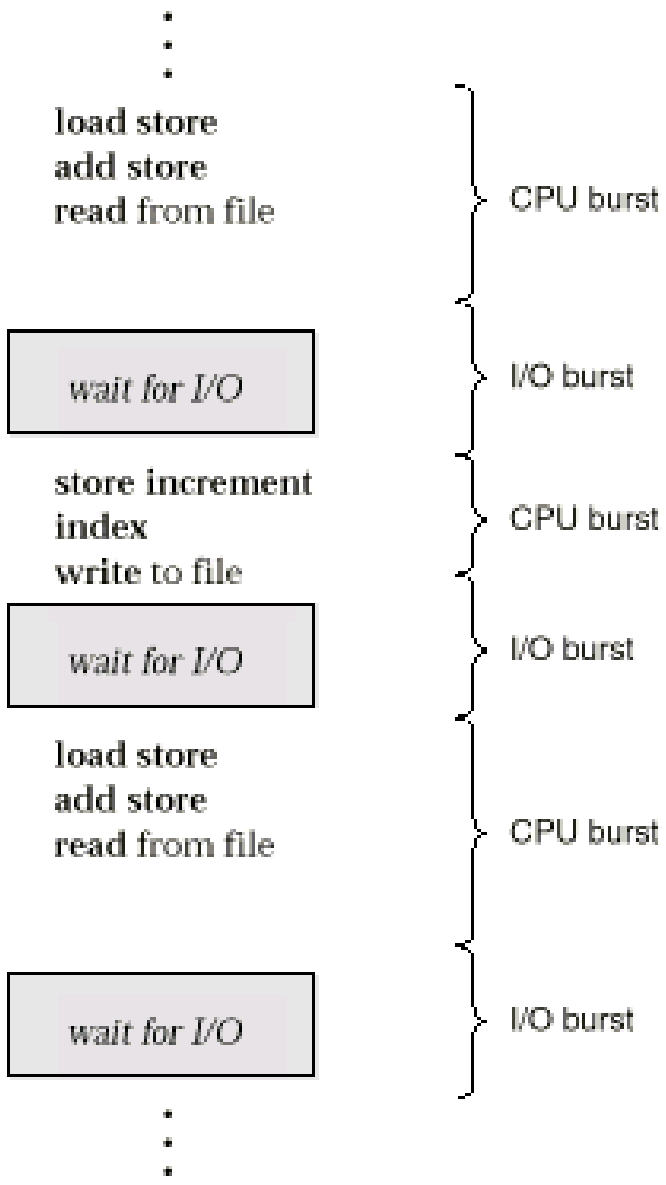


Criterios y Algoritmos de Planificación

Criterios Básicos

- Máxima utilización de CPU obtenida con multiprogramación.
- **Ciclo CPU–ráfagas de E/S** – La ejecución de procesos consiste de *ciclos* de ejecución de CPU y esperas en E/S.
- Distribución de ráfagas de CPU.

Secuencia Alternada de Ráfagas de CPU y E/S



Planificador de CPU

- Selecciona entre los procesos en memoria que están listos para ejecutar, y asigna la CPU a uno de ellos.
- La decisión de planificar la CPU puede tener lugar cuando un proceso:
  1. Conmuta de ejecutando a estado de espera.
  2. Conmuta de ejecutando a estado de listo.
  3. Conmuta de espera a listo.
  4. Termina.
- La planificación de 1 y 4 es *no apropiativa*.
- Las otras planificaciones son *apropiativas*.

Despachador

- El módulo despachador pasa el control de la CPU al proceso seleccionado (por el planificador de corto término); esto implica:
  - cambio de contexto.
  - conmutación a modo usuario.
  - salta a la dirección apropiada en el programa de usuario para reiniciarlo.
- *Latencia de despacho* – tiempo que toma al despachador para detener un proceso e iniciar otro.

Criterios de Planificación

- Utilización de CPU – mantener la CPU tan ocupada como sea posible.
- Procesamiento total (*Throughput*) – número de procesos que completan sus ejecución por unidad de tiempo.
- Tiempo de retorno (*Turnaround*) – cantidad de tiempo para ejecutar un determinado proceso.
- Tiempo de espera – cantidad de tiempo que un proceso ha estado esperando en las colas.
- Tiempo de respuesta – cantidad de tiempo que transcurre desde que fue hecho un requerimiento hasta que se produce la primer respuesta, **no salida!**

Criterios de Optimización

- Maximizar la utilización de CPU.
- Maximizar el procesamiento total.
- Minimizar el tiempo de retorno.
- Minimizar el tiempo de espera.
- Minimizar el tiempo de respuesta.

Planificación Primero-Entrar, Primero-Servido (FCFS)

Ejemplo:

Proceso	Tiempo de Ráfaga
P_1	24
P_2	3
P_3	3

Suponer que los procesos llegan en el orden: P_1 , P_2 , P_3

El diagrama de Gantt para la planificación es:

Tiempo de espera para P_1 = 0; P_2 = 24; P_3 = 27

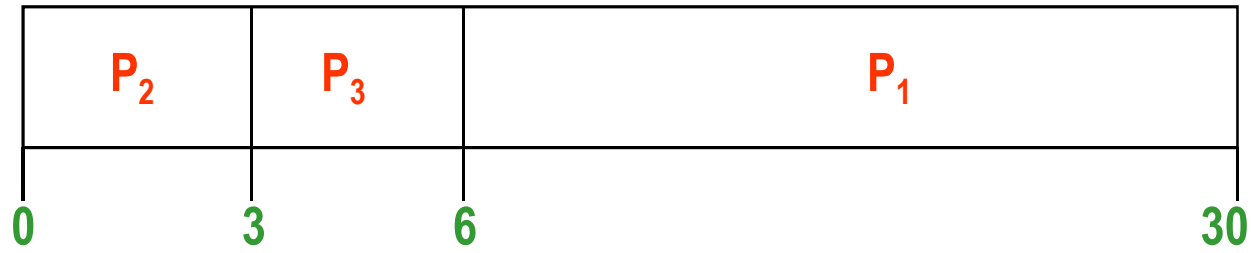
Tiempo medio de espera: (0 + 24 + 27) / 3 = 17

Planificación FCFS (Cont.)

Suponer que los procesos llegan en el orden

$$P_2 , P_3 , P_1 .$$

- El diag. de Gantt para la planificación es:



- Tiempo de espera para $P_1 = 6; P_2 = 0; P_3 = 3$
- Tiempo medio de espera: (6 + 0 + 3) / 3 = 3
- Mucho mejor que el caso anterior.
- *Efecto Convoy* los procesos cortos delante de los procesos largos.

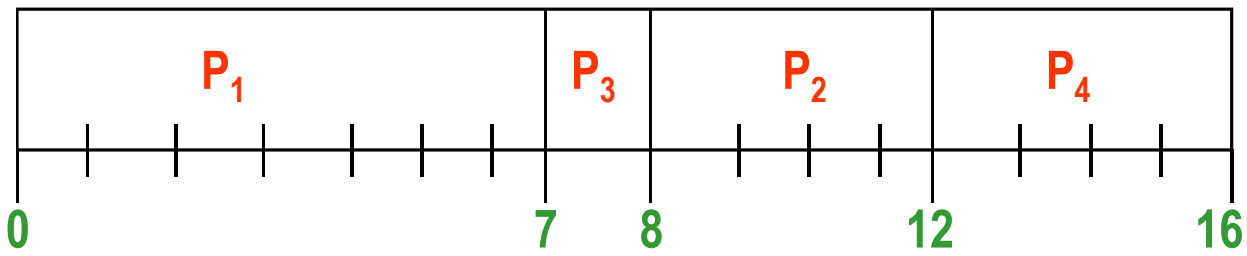
Planificación Job-Más Corto Primero (SJF)

- Se asocia con cada proceso la longitud de su *próxima* ráfaga de CPU. Se usan estas longitudes para planificar los procesos con el tiempo más corto.
- Dos esquemas:
  - No apropiativo – una vez que la CPU es dada a un proceso, no puede ser apropiada hasta que el mismo complete su ráfaga de CPU.
  - Apropiativo – si un nuevo proceso llega con una longitud de ráfaga de CPU menor que el resto del tiempo de ejecución que le queda al proceso que está ejecutando entonces se apropia de la CPU. Este esquema es conocido como El Tiempo Remanente Más Corto Primero (SRTF).
- **SJF es óptimo** – da el mínimo tiempo de espera promedio para un dado conjunto de procesos.

Ejemplo de SJF No Apropiativo

Proceso	Tiempo de llegada	Ráfaga
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (no apropiativo)

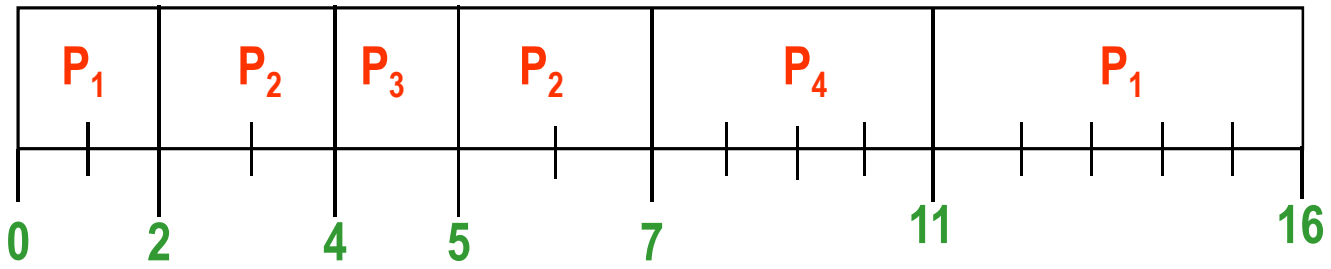


- Tiempo medio de espera = (0 + 6 + 3 + 7)/4 = 4

Ejemplo SJF Apropiativo

Proceso	Tiempo de llegada	Ráfaga
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (apropiativo)



- Tiempo medio de espera = (9 + 1 + 0 +2)/4 = 3

Planificación por Prioridad

- Con cada proceso se asocia un número (entero).
- La CPU es asignada al proceso con prioridad más alta (entero más pequeño ⇒ más alta prioridad o el entero más grande, *depende de la convención*).
  - Apropiativo
  - No apropiativo
- SJF es un algoritmo planificador con prioridad.
- **Problema** ⇒ **Inanición** – los procesos de baja prioridad pueden no llegar a ejecutarse nunca.
- **Solución** ≡ **Envejecimiento** – se incrementa en el tiempo la prioridad de los procesos en espera.

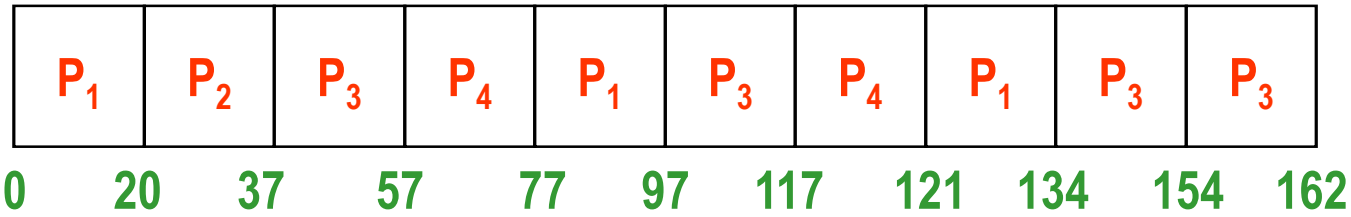
Round Robin (RR)

- Cada proceso toma una pequeña unidad de tiempo de CPU (***quantum***), usualmente 10-100 milisegundos. Luego de este tiempo el proceso es quitado de la CPU y agregado a la cola de listos.
- Si hay *n* procesos en la cola de listos y el tiempo del quantum es *q*, entonces cada proceso toma 1/*n* del tiempo de CPU en *slices* (rebanadas) de a lo sumo *q* unidades de tiempo a la vez. Los procesos no esperan más de (*n*-1)*q* unidades de tiempo.
- Rendimiento
  - *q* largo ⇒ Primero-Entrar, Primero-Salir
  - *q chico* ⇒ *q* debe ser grande con respecto al cambio de contexto, sino la sobrecarga es demasiado grande.

Ejemplo: RR con Quantum = 20

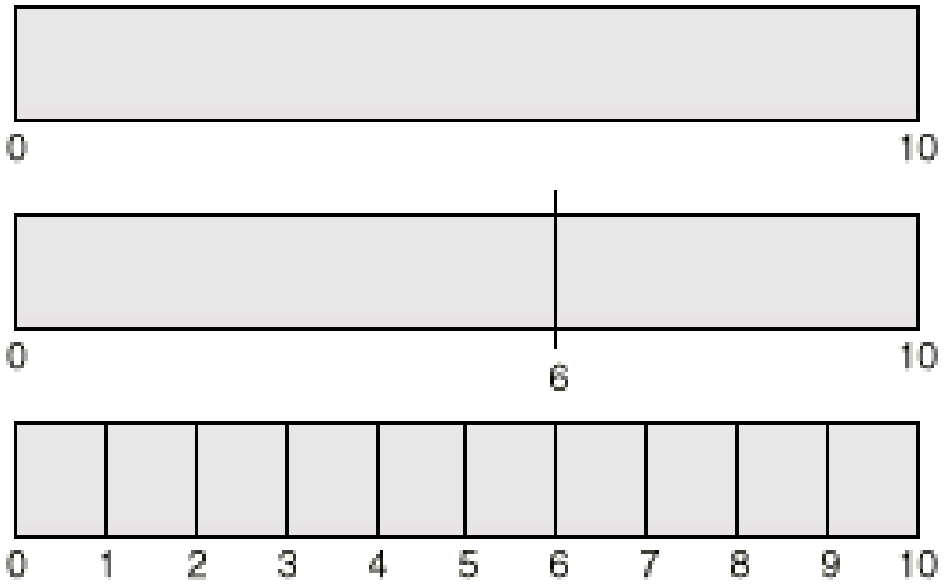
Proceso	Ráfaga
P_1	53
P_2	17
P_3	68
P_4	24

- El diagrama de Gantt:



- Típicamente, más tiempo de retorno promedio que SJF, pero mejor *respuesta*.

Quantum PEQUEÑO Incrementa los Cambios de Contexto



quantum	cambios de contexto
12	0
6	1
1	9

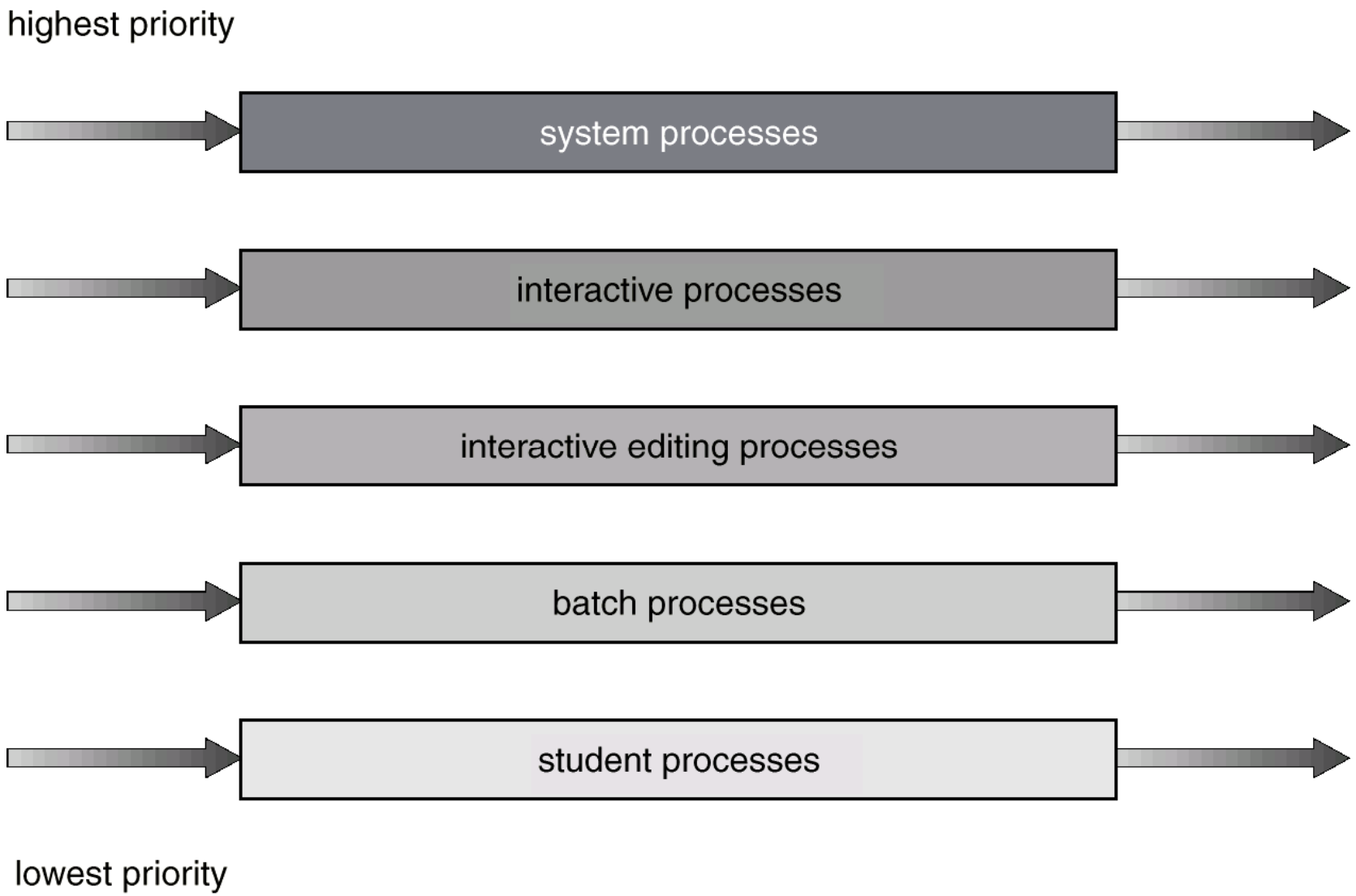
Colas Multinivel (1)

- La cola de listos está particionada en colas separadas:
  - foreground (*interactive*)
  - background (*batch*)
- Cada cola tiene su propio algoritmo de planificación,
  - foreground – RR
  - background – FCFS

Colas Multinivel (2)

- La planificación debe ser hecha entre las colas.
  - Planificación con prioridad fija; p.e., ejecutar desde el foreground y luego del background. Posibilidad de inanición.
  - *Slice* de tiempo – cada cola tiene una cierta cantidad de tiempo de CPU que puede planificar entre sus procesos; p.e., 80% en foreground en RR, 20% en background mediante FCFS.

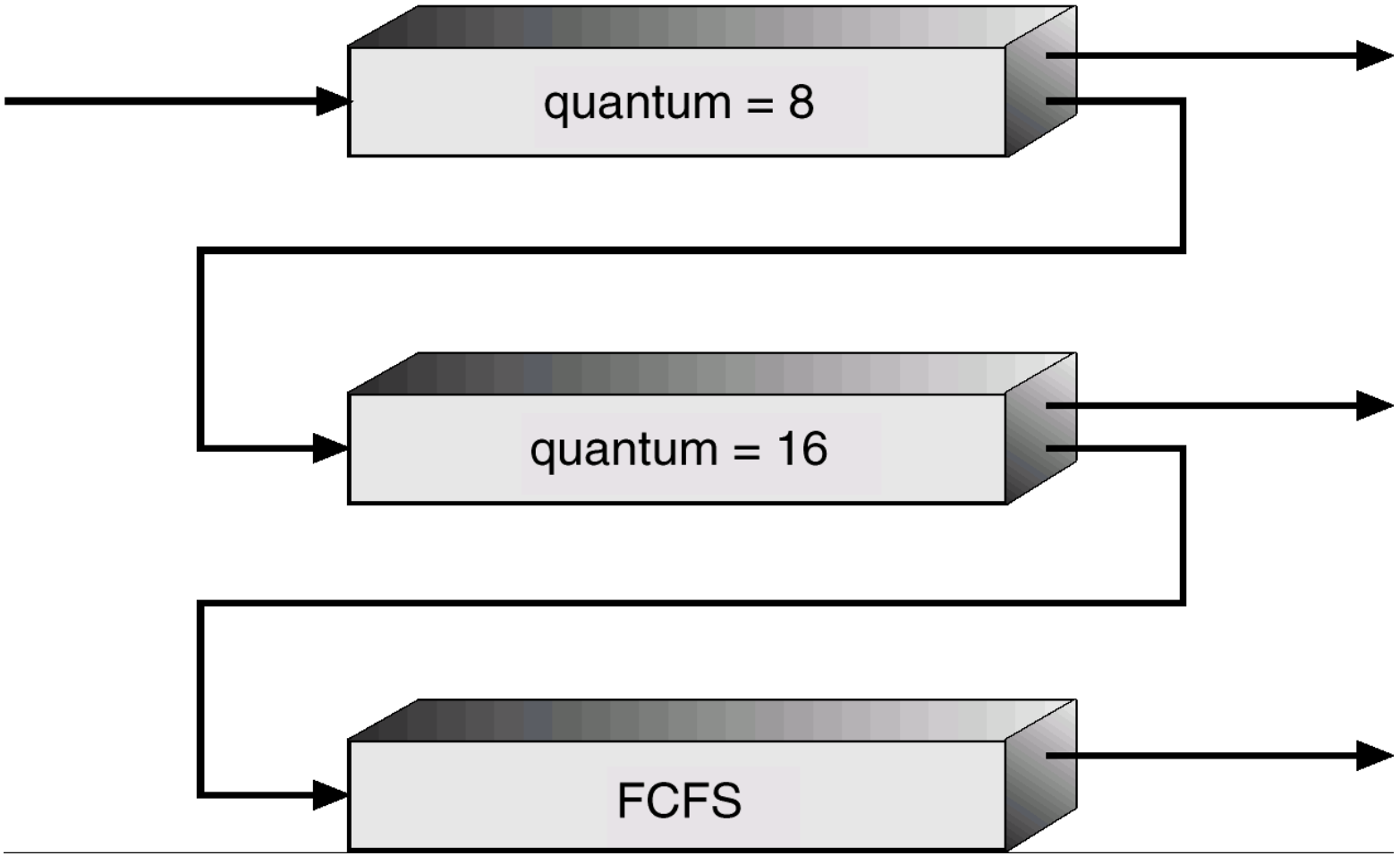
Planificación de Colas Multinivel



Colas Multinivel Realimentadas

- Un proceso puede moverse entre varias colas.
- El planificador de colas multinivel realimentadas está definido por los siguientes parámetros:
  - Número de colas.
  - Algoritmos de planificación para cada cola.
  - Método usado para determinar cuando mejorar un proceso.
  - Método usado para determinar cuando degradar un proceso.
  - Método usado para determinar en que cola entra un proceso cuando necesita servicio.

Colas Multinivel Realimentadas



Planificación Tiempo Real

- *Sistemas de Tiempo Real Duro* – requiere completar tareas críticas en una cantidad de tiempo garantizado.
- *Computación de Tiempo Real Blando* – requiere que los procesos críticos reciban prioridad sobre otros.

Planificación en UNIX (y Linux)

La planificación tradicional en UNIX emplea colas multinivel (los niveles se definen en bandas de prioridades) usando Round Robin en cada una de ellas:

$$P_j(i) = \text{Base}_j + \frac{CPU_j(i)}{2} + \text{nice}_j \qquad (1)$$

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} \qquad (2)$$

Planificación en UNIX (y Linux)

$CPU_j(i)$ = Mide la utilización del procesador por el proceso *j* en el intervalo *i*.

$P_j(i)$ = Prioridad del proceso *j* en el comienzo del intervalo *i*; valores bajos implican prioridades altas.

$Base_j$ = Prioridad base del proceso *j*.

$nice_j$ = Factor de ajuste controlable por el usuario

(1) Es utilizada para ajustar dinámicamente la prioridad (producto del uso de CPU).

(2) Es usada para implementar el "envejecimiento" cuando el proceso espera. Así evita la inanición.

Planificación en UNIX (y Linux)

La prioridad de cada proceso es computada cada segundo (en los primeros UNIX, hoy es cada *quantum*).

El propósito de la prioridad base es dividir todos los procesos en bandas de niveles de prioridad.

Los componentes ***CPU*** y ***nice*** se utilizan para prevenir que los procesos migren fuera de su banda asignada (dada por la prioridad base).

Estas bandas son utilizadas para optimizar el acceso a los dispositivos que se manejan con bloques de información (discos, cintas, CD, etc) y permitir al sistema operativo responder rápidamente a las llamadas a sistema.

Planificación en UNIX (y Linux)

En orden decreciente de prioridad, las bandas son:

- Swapper.
- Control de dispositivos de E/S en bloques.
- Manipulación de archivos.
- Control de dispositivos de E/S por caracteres.
- Procesos de usuarios.

Dentro de la banda de procesos de usuario, el uso de la historia de ejecución tiende a penalizar a los procesos limitados por procesador a expensas de los procesos limitados por E/S.

Planificación con Múltiples Procesadores

- Nos enfocaremos únicamente en procesadores múltiples idénticos (homogéneos).

- **Multiprocesamiento asimétrico**
  - Sencillo (un master), sólo un procesador accede a las estructuras internas del sistema.
- **Multiprocesamiento simétrico**
  - Cada procesador efectúa self-sched, i.e. puede tener su propia cola ready.

Balance de Carga

- En sistemas SMP es fundamental mantener la carga bien balanceada entre todos los procesadores para aprovechar la multiplicidad de los mismos.
- Existen dos formas para llevar a cabo el balance de carga:
  - Migración Push.
  - Migración Pull.

Multithreading Simétrico

- Esta tecnología provee múltiples procesadores lógicos (en lugar de físicos).
- En la jerga Intel es **hyperthreading**
  - Disponible desde Pentium IV.
- Se crean múltiples procesadores lógicos sobre un procesador físico (tipo threads!).
- No es tecnología por SW sino HW!
- Cada procesador lógico tiene su propio PSW y es responsable por la gestión de sus interrupciones.

Threads

(extra slides)

Department of Computer Science
Rutgers University

http://www.cs.rutgers.edu/~ricardob/courses/cs416/web/

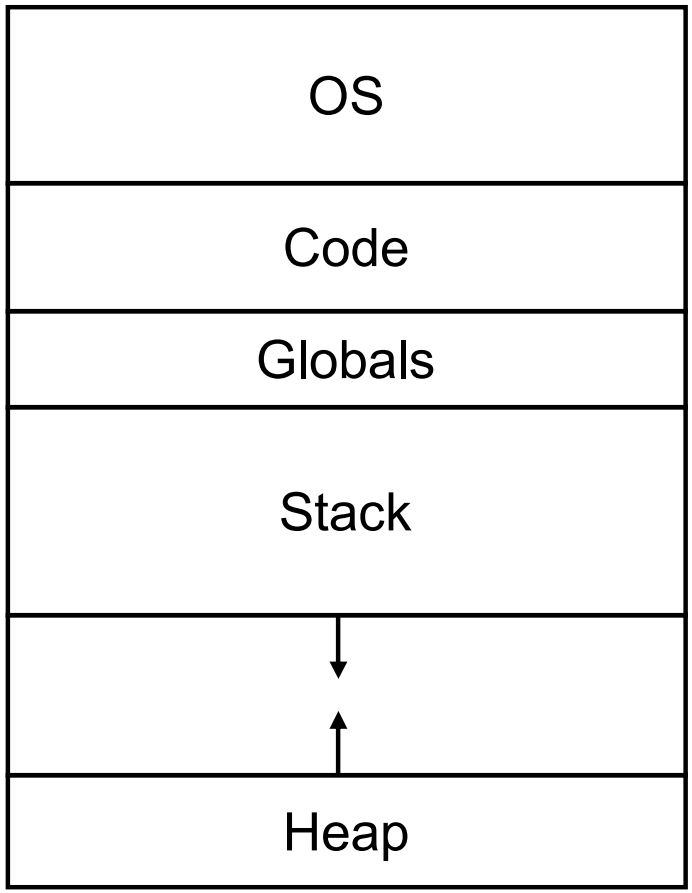
Threads

- Thread of execution: stack + registers (which includes the PC)
  - Informally: where an execution stream is currently at in the program and the routine call chain that brought the stream to the current place
  - Example: A called B which called C which called B which called C
  - The PC should be pointing somewhere inside C at this point
  - The stack should contain 5 activation records: A/B/C/B/C
  - Thread for short
- Process model discussed last time implies a single thread

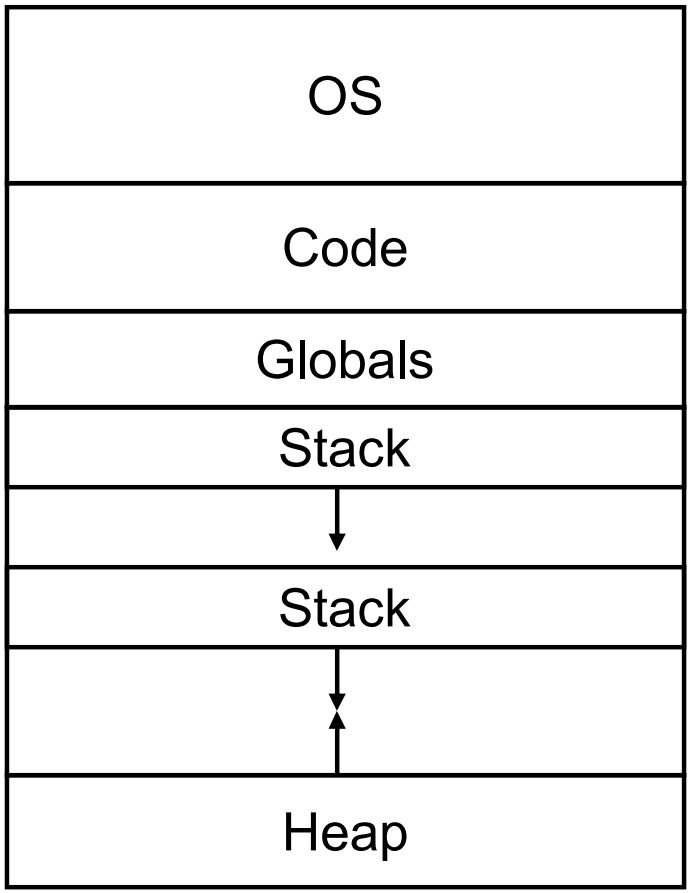
Multi-Threading

- Why limit ourselves to a single thread?
  - Think of a web server that must service a large stream of requests
  - If only have one thread, can only process one request at a time
  - What to do when reading a file from disk?
- Multi-threading model
  - Each process can have multiple threads
  - Each thread has a private stack
    - Registers are also private
  - All threads of a process share the code, globals, and heap
    - Objects to be shared across multiple threads should be allocated on the heap or in the globals area

Process Address Space Revisited



(a) Single-threaded address space

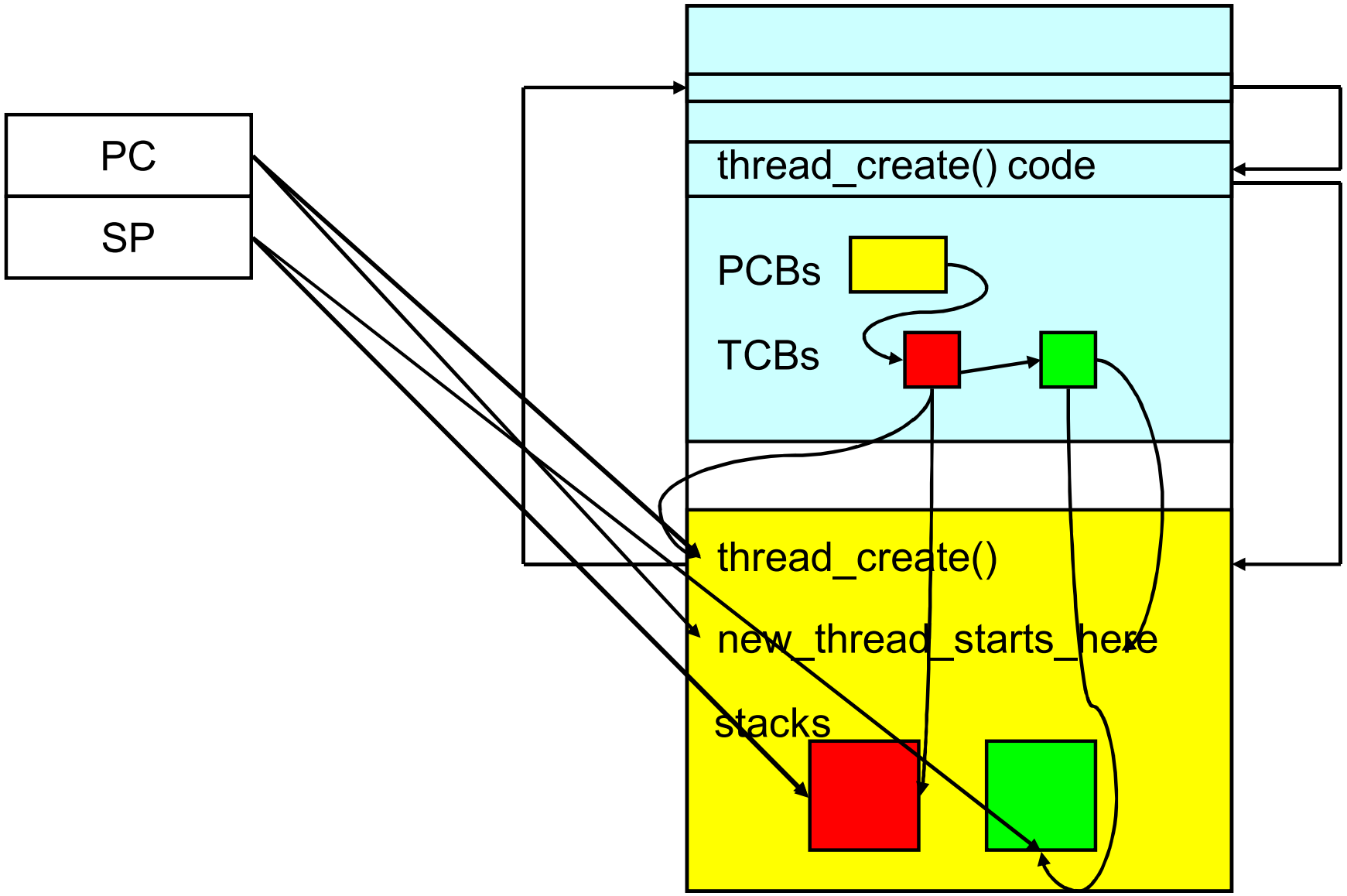


(b) Multi-threaded address space

Multi-Threading (cont)

- Implementation
  - Each thread is described by a *thread-control block* (TCB)
  - A TCB typically contains
    - Thread ID
    - Space for saving registers
    - Thread-specific info (e.g., signal mask, scheduling priority)
- Observation
  - Although the model is that each thread has a private stack, threads actually share the process address space
  - ⇒ There's no memory protection!
  - ⇒ Threads could potentially write into each other's stack

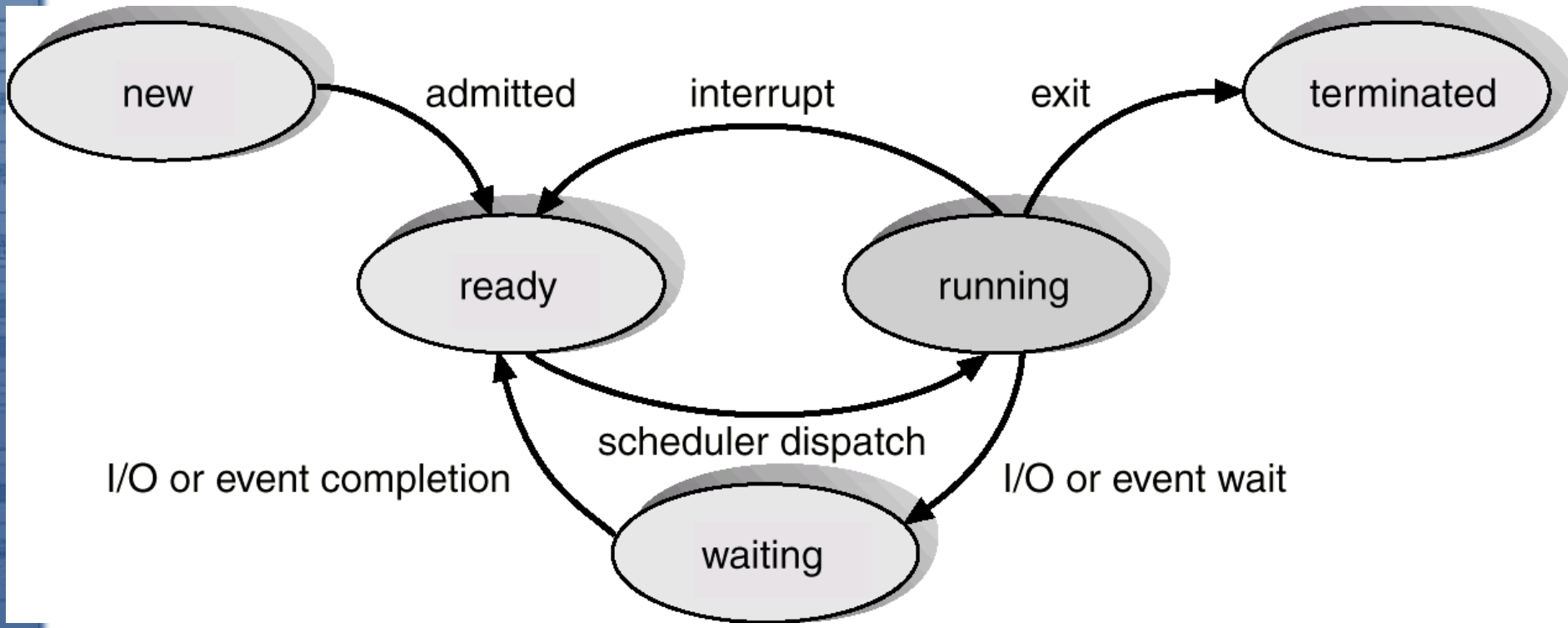
Thread Creation



Context Switching

- Suppose a process has multiple threads on a machine with a single non-multithreaded CPU core … what to do?
  - In fact, even if we only had one thread per process, we would have to do something about running multiple processes …
- We *multiplex* the multiple threads on the core
- At any point in time, only one thread is *running* (again, assuming a single non-multithreaded core)
- At some point, the OS may decide to stop the currently running thread and allow another thread to run
- This switching from one running thread to another is called *context switching*

Diagram of Thread State



Context Switching (cont)

- How to do a context switch?
- Save state of currently executing thread
  - Copy all “live” registers to thread control block
  - For register-only machines, need at least 1 scratch register
    - points to area of memory in thread control block that registers should be saved to
- Restore state of thread to run next
  - Copy values of live registers from thread control block to registers
- When does context switching take place?

Context Switching (cont)

- When does context switching occur?
  - When the OS decides that a thread has run long enough and that another thread should be given the CPU
  - When a thread performs an I/O operation and needs to block to wait for the completion of this operation
  - To wait for some other thread
    - Thread synchronization: we'll talk about this lots in a couple of lectures

How Is the Switching Code Invoked?

- user thread executing → clock interrupt → PC modified by hardware to "vector" to interrupt handler → user thread state is saved for restart → clock interrupt handler is invoked → disable interrupt checking → check whether current thread has run "long enough" → if yes, post asynchronous software trap (AST) → enable interrupt checking → exit clock interrupt handler → enter "return-to-user" code → check whether AST was posted → if not, restore user thread state and return to executing user thread; if AST was posted, call context switch code

- Why need AST?

How Is the Switching Code Invoked? (cont)

•user thread executing → system call to perform I/O → PC modified by hardware to "vector" to trap handler → user thread state is saved for restart → OS code to perform system call is invoked → disable interrupt checking → I/O operation started (by invoking I/O driver) → set thread status to waiting → move thread's TCB from run queue to wait queue associated with specific device → enable interrupt checking → exit trap handler → call context switching code

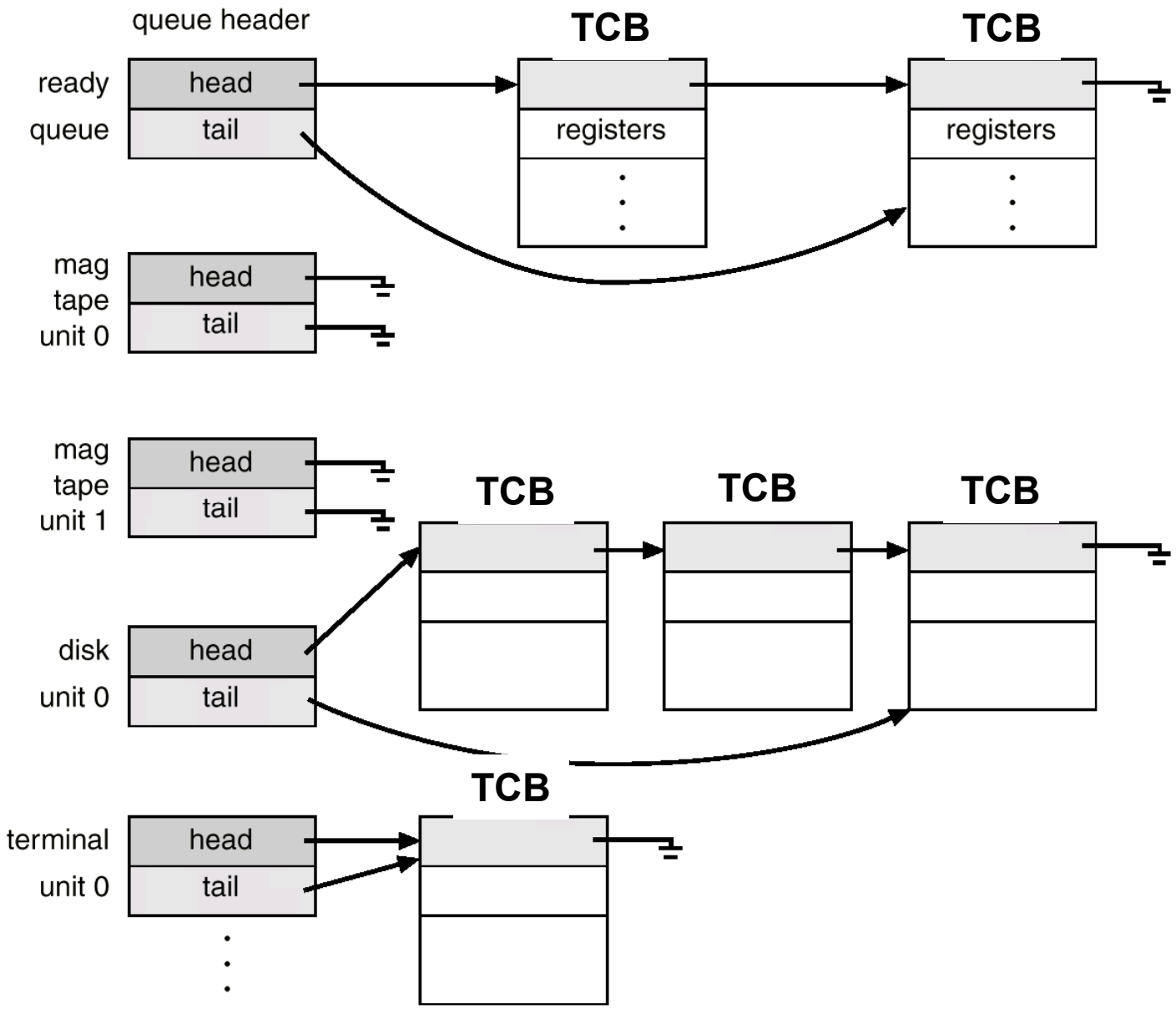
Context Switching

- At entry to CS, the return address is either in a register or on the stack (in the current activation record)
- CS saves this return address to the TCB instead of the current PC
- To thread, it looks like CS just took a while to return!
  - If the context switch was initiated from an interrupt, the thread never knows that it has been context switched out and back in unless it looks at the “wall” clock

Context Switching (cont)

- Even that is not quite the whole story
- When a thread is switched out, what happens to it?
- How do we find it to switch it back in?
- This is what the TCB is for. System typically has
  - A run queue that points to the TCBs of threads ready to run
  - A blocked queue per device to hold the TCBs of threads blocked waiting for an I/O operation on that device to complete
  - When a thread is switched out at a timer interrupt, it is still ready to run so its TCB stays on the run queue
  - When a thread is switched out because it is blocking on an I/O operation, its TCB is moved to the blocked queue of the device

Ready Queue And Various I/O Device Queues



Switching Between Threads of Different Processes

- What if switching to a thread of a different process?
- Caches, TLB, page table, etc.?
  - Caches
    - Physical addresses: no problem
    - Virtual addresses: cache must either have process tag or must flush cache on context switch
  - TLB
    - Each entry must have process tag or must flush TLB on context switch
  - Page table
    - Typically have page table pointer (register) that must be reloaded on context switch

Threads & Signals

- What happens if kernel wants to signal a process when all of its threads are blocked?
- When there are multiple threads, which thread should the kernel deliver the signal to?
  - OS writes into process control block that a signal should be delivered
  - Next time any thread from this process is allowed to run, the signal is delivered to that thread as part of the context switch
  - What happens if kernel needs to deliver multiple signals?

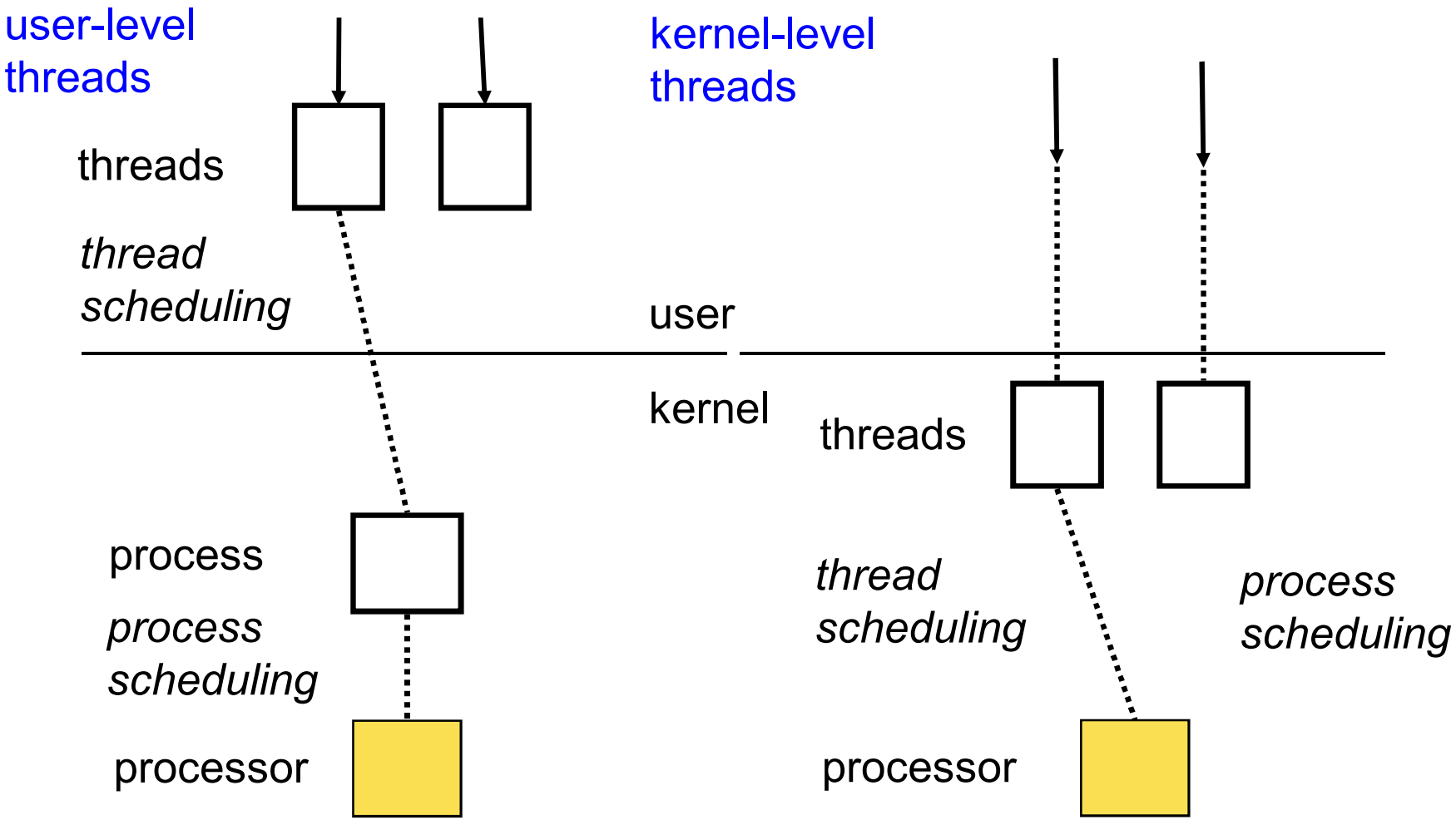
Thread Implementation

- Kernel-level threads
  - Kernel sees multiple execution contexts
  - Thread management done by the kernel
- User-level threads
  - Implemented as a thread library which contains the code for thread creation, termination, scheduling and switching
  - Kernel sees one execution context and is unaware of thread activity
  - Can be preemptive or not

User-Level vs. Kernel-Level Threads

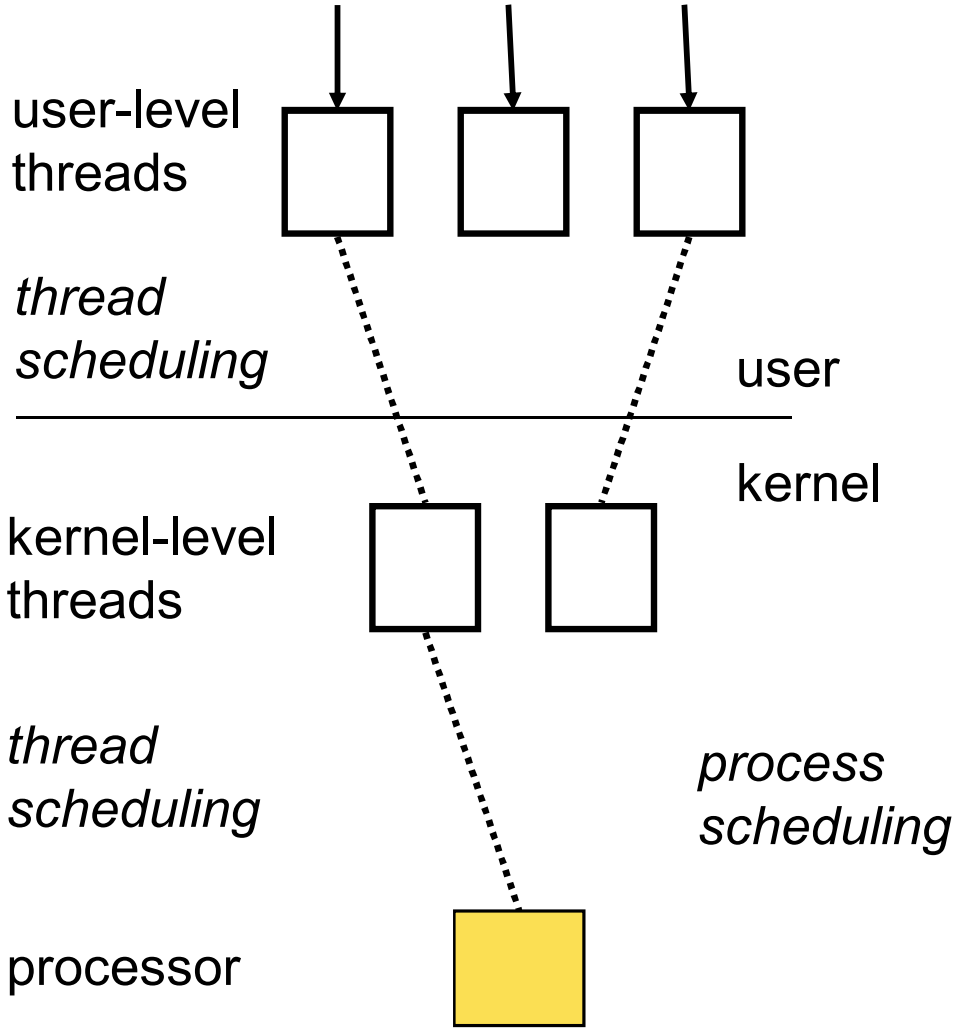
- Advantages of user-level threads
  - Performance: low-cost thread operations (do not require crossing protection domains)
  - Flexibility: scheduling can be application-specific
  - Portability: user-level thread library easy to port
- Disadvantages of user-level threads
  - If a user-level thread is blocked in the kernel, the entire process (all threads of that process) are blocked
  - Cannot take advantage of multiprocessing (the kernel assigns one process to only one processor)

User-Level vs. Kernel-Level Threads



User-Level vs. Kernel-Level Threads

- No reason why we shouldn't have both
- Most systems now support kernel threads
- User-level threads are available as linkable libraries



Thread Implementation in Real OSes

- Lightweight processes (Solaris only)
  - LWPs create an extra layer between user-level and kernel-level threads
  - An LWP runs in user-space on top of a kernel-level thread; multiple user-level threads can be created on top of each LWP
  - LWPs of the same process share data
- Process ~= Thread (Linux only)
  - The schedulable entities are processes (called “tasks” in Linux lingo)
  - A process can be seen as a single thread, but a process can contain multiple threads that share code + data
  - In the Pthreads library (Native POSIX Thread Library or NPTL) for Linux, each thread created corresponds to a kernel schedulable entity

Threads vs. Processes

- Why multiple threads?
  - Can't we use multiple processes to do whatever that is that we do with multiple threads?
    - Of course, we need to be able to share memory (and other resources) between multiple processes …
    - But this sharing is already supported
  - Operations on threads (creation, termination, scheduling, etc) are cheaper than the corresponding operations on processes
    - This is because thread operations do not involve manipulations of other resources associated with processes
  - Inter-thread communication is supported through shared memory without kernel intervention
  - Why not? Have multiple other resources, why not threads

Thread/Process Operation Latencies

Operation	User-level Thread (µs)	Kernel Threads (µs)	Processes (µs)
Null fork	34	948	11,300
Signal-wait	37	441	1,840

VAX uniprocessor running UNIX-like OS, 1992.

Operation	Kernel Threads (µs)	Processes (µs)
Null fork	45	108

2.8-GHz Pentium 4 uniprocessor running Linux, 2004.

Pthreads

Operating Systems

Hebrew University of Jerusalem

How to Compile

- #include <pthread.h>
- gcc myprog.c –o myprog –l pthread

thread creation

```
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void* (*start_routine)(void*),
                   void *arg);
```

- Create a thread and run the start_routine
- attr is usually NULL, don't mess with it

Example

```
#include <pthread.h>
int val = 0;

void *thread(void *vargp) {
    val  = (int)vargp;
}

int main() {
    int i;
    pthread_t tid;
    pthread_create(&tid, NULL, thread, (void *)42);
    pthread_join(tid, NULL);
    printf("%d\n",val);
}
```

sched_yield

```
#include <sched.h>
#include <unistd.h>
int sched_yield (void);
```

- Yield the processor to another thread
- Useful on uni-processor

Who am I

- pthread_t pthread_self(void)
- Uses:
  - Debugging
  - Data structures indexed by thread
- pthread_equal: compare two pthread_t

Relationships

- Marriage:
  - pthread_join – I will wait for you forever
- Good bye
  - pthread_exit – I am going away now
- Death
  - pthread_cancel – please die
- Divorce:
  - pthread_detach – never talk to me again

pthread_join

- int pthread_join(pthread t, void *data)
- Wait until the thread exits and return the exit data. This call blocks!
- Performs a detach after the join succeeds

- Return values:
  - 0: successful completion
  - EINVAL: thread is not joinable
  - ESRCH: no such thread
  - EDEADLK: a deadlock was detected, or thread specifies the calling thread

pthread_exit

- void pthread_exit(void *data)

- Stops execution of this thread
- Return data to anyone trying to join this thread
- Don’t call from the main thread, use exit()

Example

```
#include <pthread.h>
void *thread(void *vargp) {
    pthread_exit((void*)42);
}
int main() {
    int i;
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, (void **)&i);
    printf("%d\n",i);
}
```

pthread_cancel

- Die!
- int pthread_cancel(pthread_t thread)

- return values:
  - 0: ok
  - EINVAL: thread is invalid
  - ESRCH: no such thread

pthread_cancel

- Three phases
  - Post a cancel request
  - Deliver to the target thread at the next cancellation point
  - Call cleanup routines and die

pthread_detach

- I never want to see this thread again.
- int pthread_detach(pthread_t thread)

- Return values:
  - 0 - ok
  - EINVAL – thread is not joinable
  - ESRCH – no such thread

- On some systems, exit does not cause the program to exit until all non-detached threads are finished

Review: Exiting threads

- Four options
  - Exit from start routine
  - Call pthread_exit
  - Call exit
  - Killed by pthread_cancel

Review

- pthread_create
- pthread_join
- pthread_detach
- pthread_exit
- pthread_cancel
- pthread_self
- pthread_equal
- sched_yield

Coming Next