Introducción a los Sistemas Operativos

Procesos - I

I.S.O.

- ☑ Versión: Agosto 2019
- ☑ Palabras Claves: Procesos, PCB, Stack, Contexto, Espacio de Direcciones

Los temas vistos en estas diapositivas han sido mayormente extraídos del libro de Andrew S. Tanenbaum (Sistemas Operativos Modernos) y del libro de William Stallings (Sistemas Operativos: Aspectos internos y principios de diseño)

Definición de proceso

- ☑ Es un programa en ejecución
- ☑ Para nosotros serán sinónimos: tarea, job y proceso

Diferencias entre un programa y un proceso

Programa

- ☑ Es estático
- ☑ No tiene *program counter*
- ☑ Existe desde que se edita hasta que se borra

Proceso

- ☑ Es dinámico
- ☑ Tiene *program counter*
- ☑ Su ciclo de vida comprende desde que se lo “dispara” hasta que termina

El Módelo de Proceso

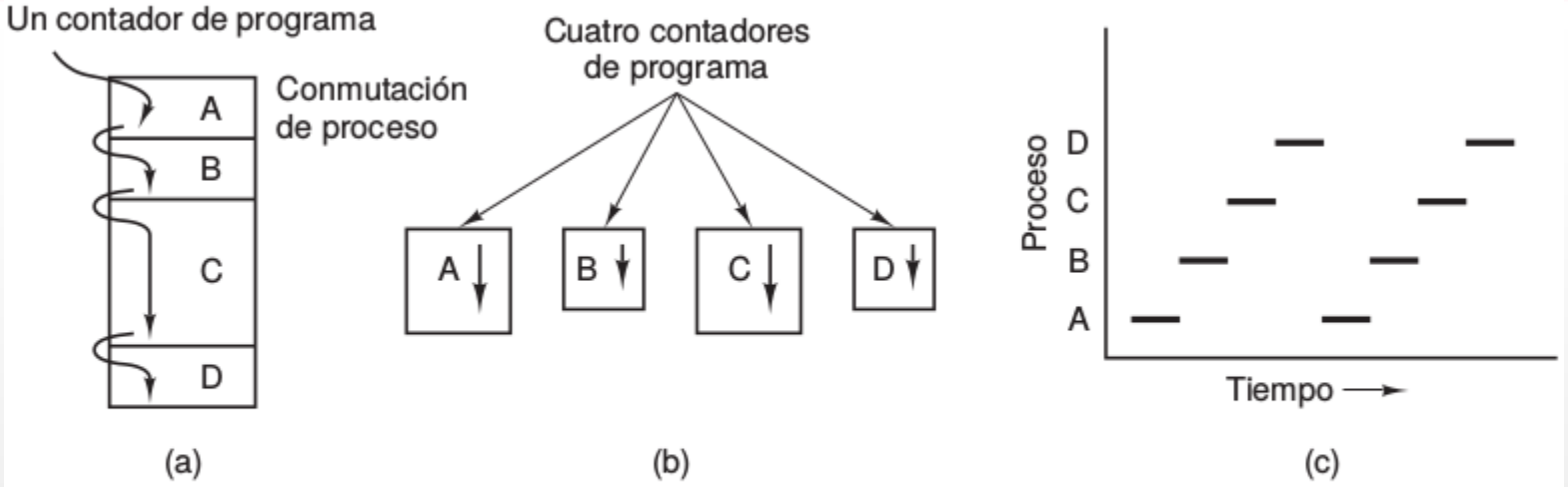


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

- ☑ Multiprogramación de 4 procesos
- ☑ Modelo conceptual de 4 procesos secuenciales e independientes.
- ☑ Solo un proceso se encontrara activo en cualquier instante. (Si tenemos una sola CPU)

Componentes de un proceso

Proceso: Entidad de abstracción

Un proceso (para poder ejecutarse) incluye como mínimo:

- ✓ Sección de Código (texto)
- ✓ Sección de Datos (variables globales)
- ✓ Stack(s) (datos temporarios: parámetros , variables temporales y direcciones de retorno)

Stacks

- Un proceso cuenta con 1 o mas stacks
  - En general: modo Usuario y modo Kernel
- Se crean automáticamente y su medida se ajusta en run-time.
- Está formado por *stack frames* que son *pushed* (al llamar a una rutina) y *popped* (cuando se retorna de ella)
- El *stack frame* tiene los parámetros de la rutina(variables locales), y datos necesarios para recuperar el stack frame anterior (el contador de programa y el valor del stack pointer en el momento del llamado)

Atributos de un proceso

- ☑ Identificación del proceso, y del proceso padre
- ☑ Identificación del usuario que lo “disparó”
- ☑ Si hay estructura de grupos, grupo que lo disparó
- ☑ En ambientes multiusuario, desde que terminal y quien lo ejecuto.

Process Control Block (PCB)

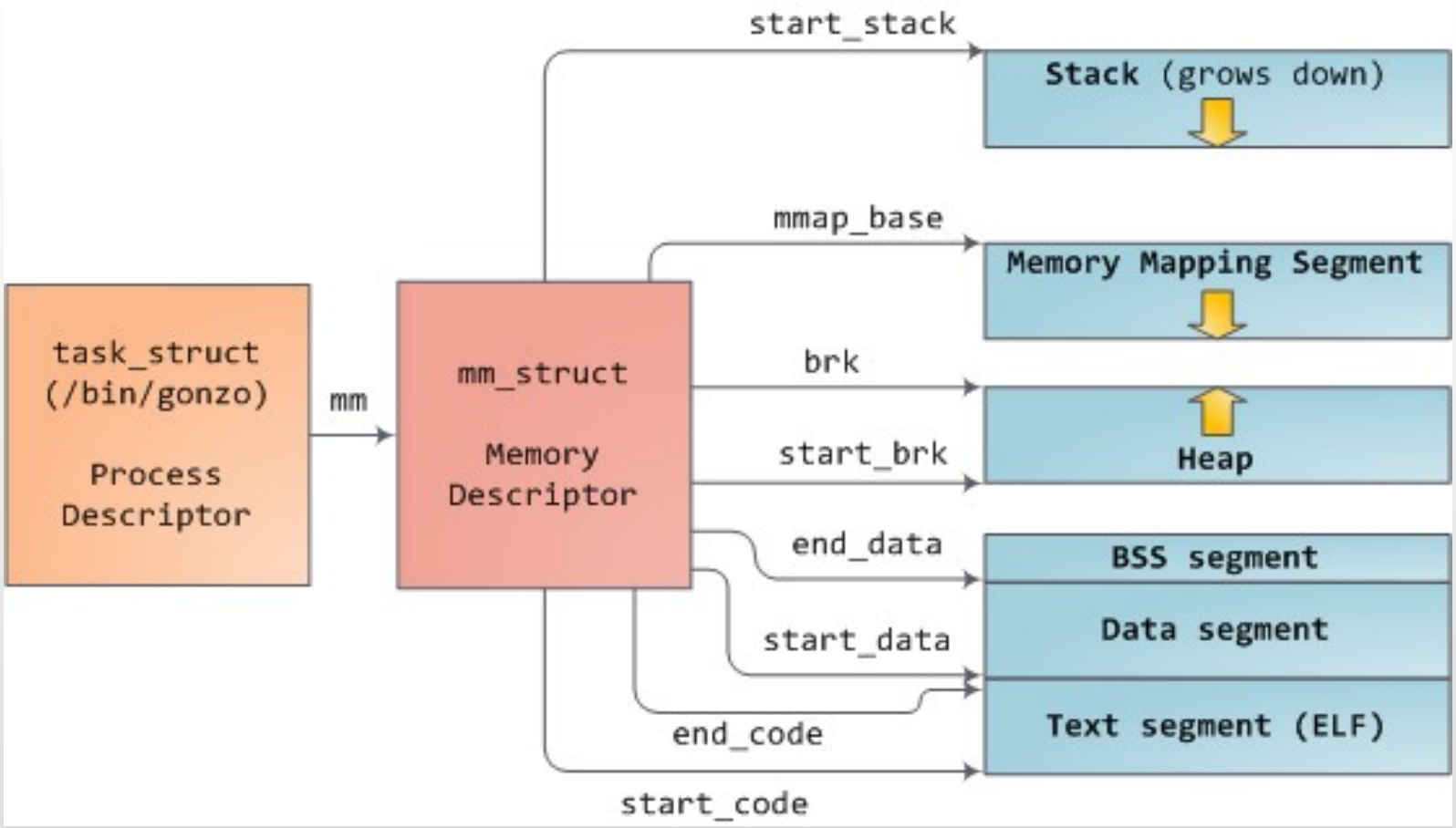
- Estructura de datos asociada al proceso (abstracción)
- Existe una por proceso.
- Es lo primero que se crea cuando se crea un proceso y lo último que se borra cuando termina
- Contiene la información asociada con cada proceso:
  - PID, PPID, etc
  - Valores de los registros de la CPU (PC, AC, etc)
  - Planificación (estado, prioridad, tiempo consumido, etc)
  - Ubicación (representación) en memoria
  - Accounting
  - Entrada salida (estado, pendientes, etc)

PCB (cont.)

Administración de procesos	Administración de memoria	Administración de archivos
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		

Campos Comunes

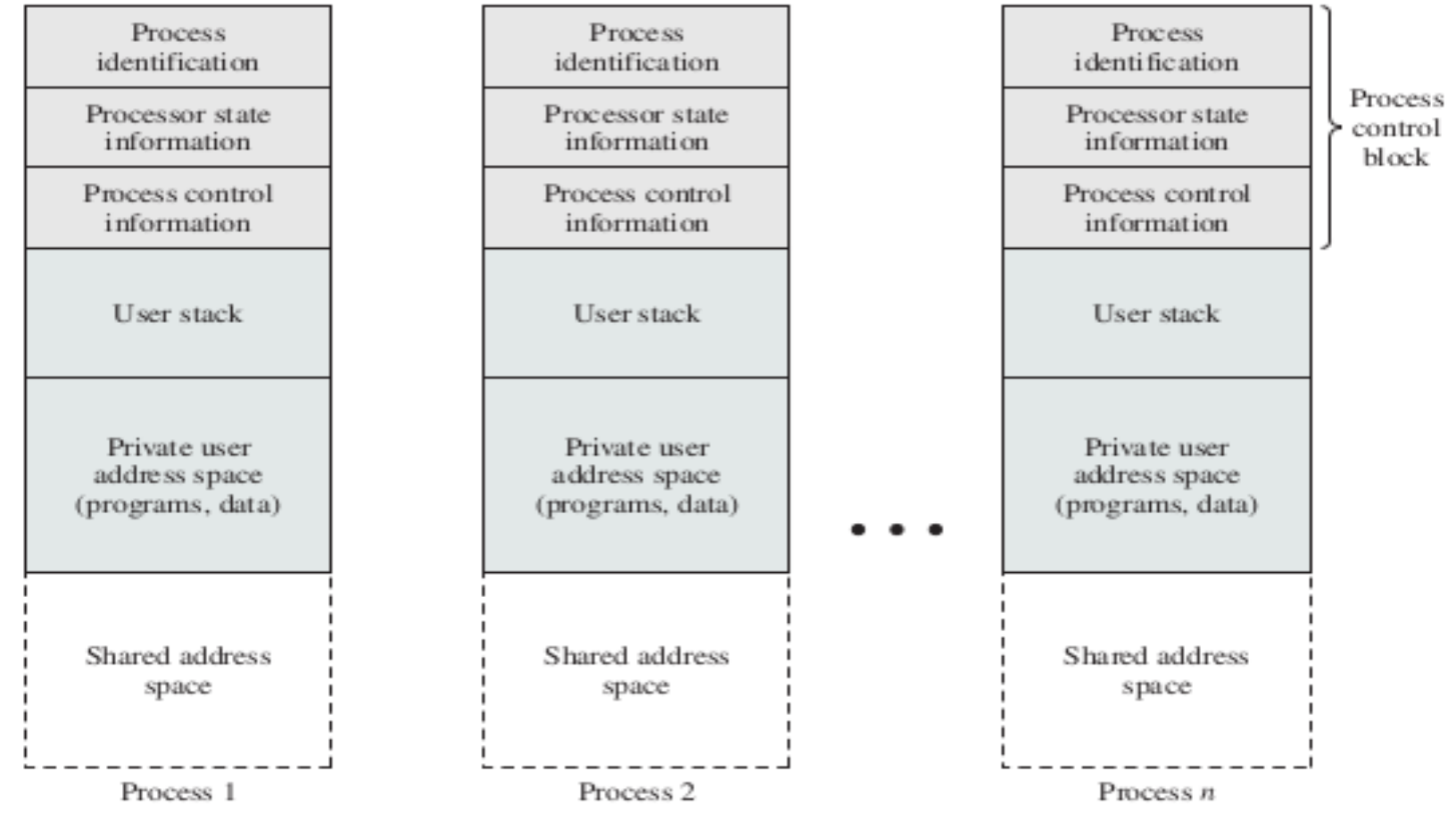
PCB (cont.)


start_stack
Stack (grows down)
mmap_base
Memory Mapping Segment
task_struct
(/bin/gonzo)
Process
Descriptor
mm
mm_struct
Memory
Descriptor
brk
start_brk
Heap
end_data
BSS segment
start_data
Data segment
end_code
Text segment (ELF)
start_code

Qué es el espacio de direcciones de un proceso?

- Es el conjunto de direcciones de memoria que ocupa el proceso
  - stack, text y datos
- **No incluye su PCB o tablas asociadas**
- Un proceso en modo usuario puede acceder sólo a su espacio de direcciones;
- En modo kernel, se puede acceder a estructuras internas (PCB del proceso por ejemplo) o a espacios de direcciones de otros procesos.

Espacio de direcciones del proceso + PCB



El contexto de un proceso

- ☑ Incluye toda la información que el SO necesita para administrar el proceso, y la CPU para ejecutarlo correctamente.
- ☑ Son parte del contexto, los registros de cpu, inclusive el contador de programa, prioridad del
- ☑ proceso, si tiene

E/S pendientes,

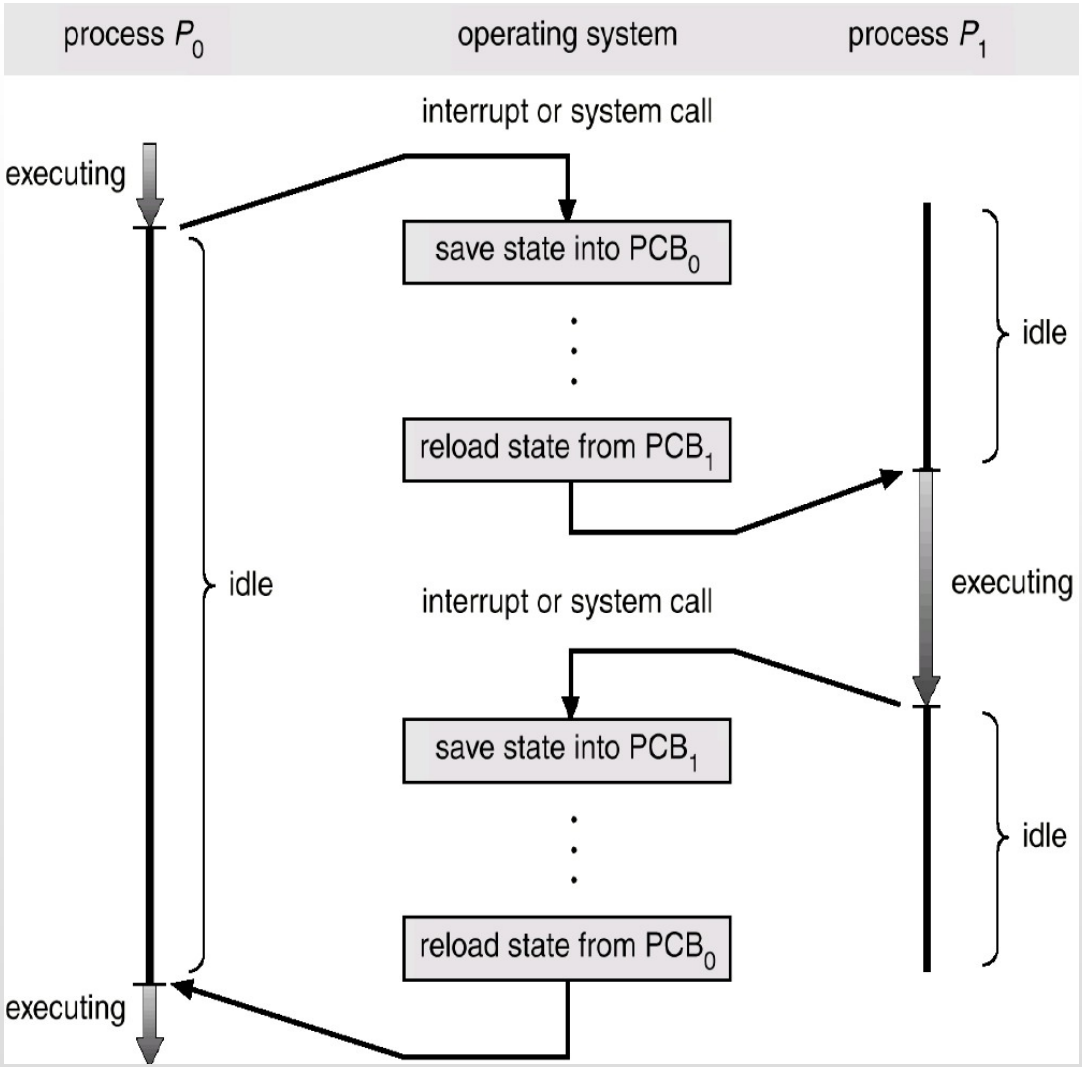
etc.

Administración de procesos	Administración de memoria	Administración de archivos
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		

Cambio de Contexto (Context Switch)

- Se produce cuando la CPU cambia de un proceso a otro.
- Se debe resguardar el contexto del proceso saliente, que pasa a espera y retornará después a la CPU.
- Se debe cargar el contexto del nuevo proceso y comenzar desde la instrucción siguiente a la última ejecutada en dicho contexto.
- Es tiempo no productivo de CPU
- El tiempo que consume depende del soporte de HW

Ejemplo de Cambio de Contexto

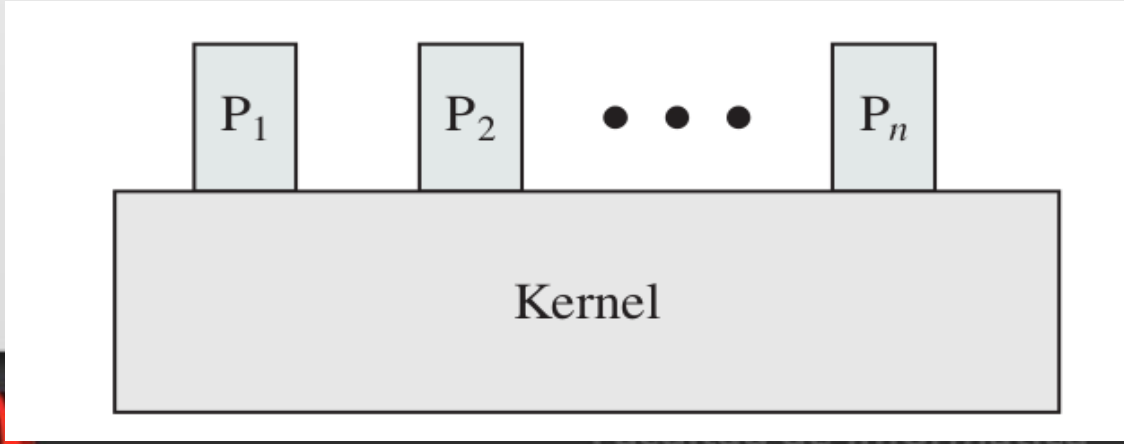


Sobre el Kernel del Sistema Opetativo

- ☑ Es un conjunto de módulos de software
- ☑ Se ejecuta en el procesador como cualquier otro proceso
- ☑ Entonces:
  - ¿El kernel es un proceso? Y de ser así ¿Quien lo controla?
- ☑ Diferentes enfoques de diseño

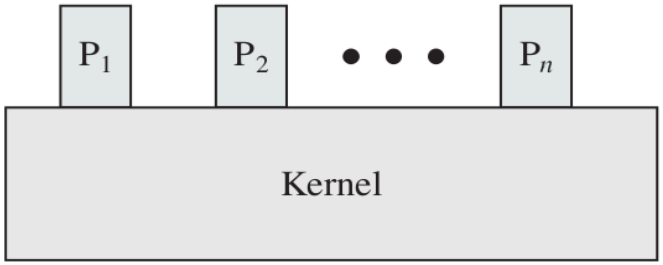
Enfoque 1 – El Kernel como entidad independiente

- El Kernel se ejecuta fuera de todo proceso
- Es una arquitectura utilizada por los primeros SO
- Cuando un proceso es “interrumpido” o realiza una System Call, el contexto del proceso se salva y el control se pasa al Kernel del sistema operativo.



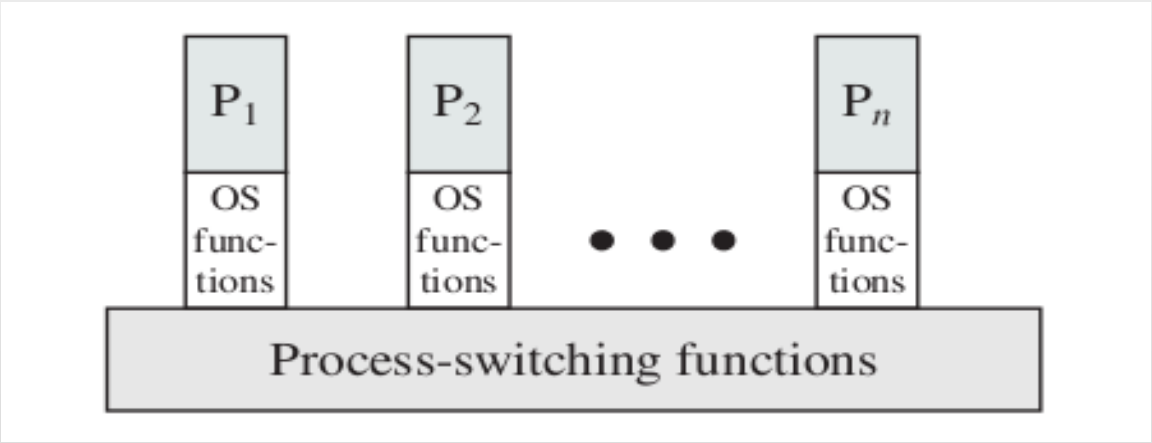
Enfoque 1 – El Kernel como entidad independiente

- El Kernel tiene su propia región de memoria
- El Kernel tiene su propio Stack
- Finalizada su actividad, le devuelve el control al proceso (o a otro diferente)
- Importante:
  - El Kernel NO es un proceso. EL concepto de proceso solo se asocia a programas de usuario
  - Se ejecuta como una entidad independiente en modo privilegiado



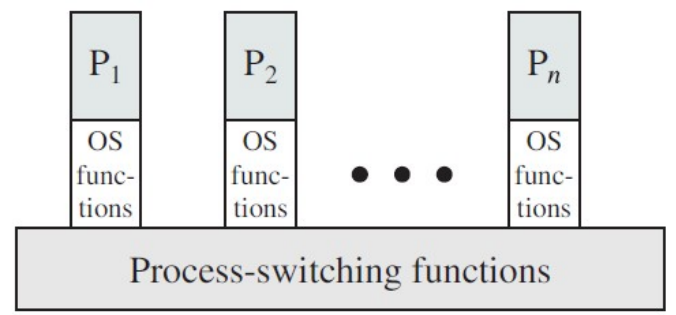
Enfoque 2 – El Kernel "dentro" del Proceso

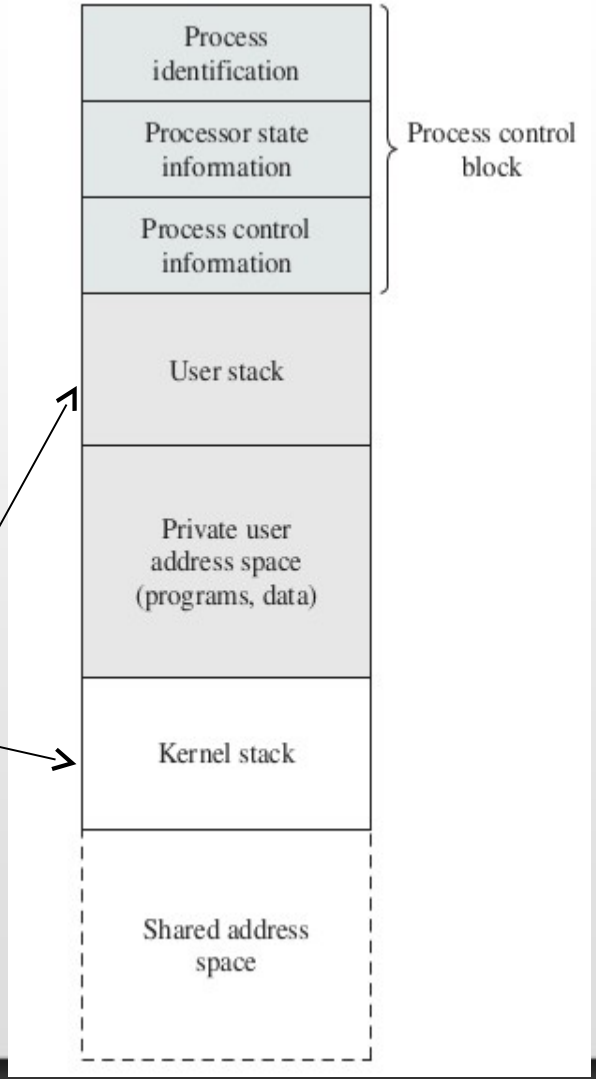
- El "Código" del Kernel se encuentra dentro del espacio de direcciones de cada proceso.
- El Kernel se ejecuta en el MISMO contexto que algún proceso de usuario
- El Kernel se puede ver como una colección de rutinas que el proceso utiliza



Enfoque 2 – El Kernel “dentro” del Proceso

- Dentro de un proceso se encuentra el código del programa (user) y el código de los módulos de SW del SO (kernel`

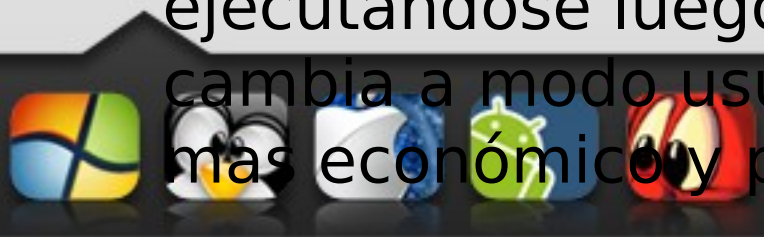




- Cada proceso tiene su propio stack (uno en modo usuario y otro en modo kernel)
- El proceso es el que se Ejecuta en Modo Usuario y el kernel del SO se ejecuta en Modo Kernel **(cambio de modo)**

Enfoque 2 – El Kernel "dentro" del Proceso

- El código del Kernel es compartido por todos los procesos
  - En administración de memoria veremos el "como"
- Cada interrupción (incluyendo las de System Call) es atendida en el contexto del proceso que se encontraba en ejecución
  - Pero en modo Kernel!!! (se pasa a este modo sin necesidad de hacer un cambio de contexto completo)
  - Si el SO determina que el proceso debe seguir ejecutándose luego de atender la interrupción, cambia a modo usuario y devuelve el control. Es mas económico y performante



Introducción a los Sistemas Operativos

Procesos - II

I.S.O.

- ☑ Versión: Agosto 2019
- ☑ Palabras Claves: Procesos, Estados, Scheduler, Long Term, Medium Term, Short Term

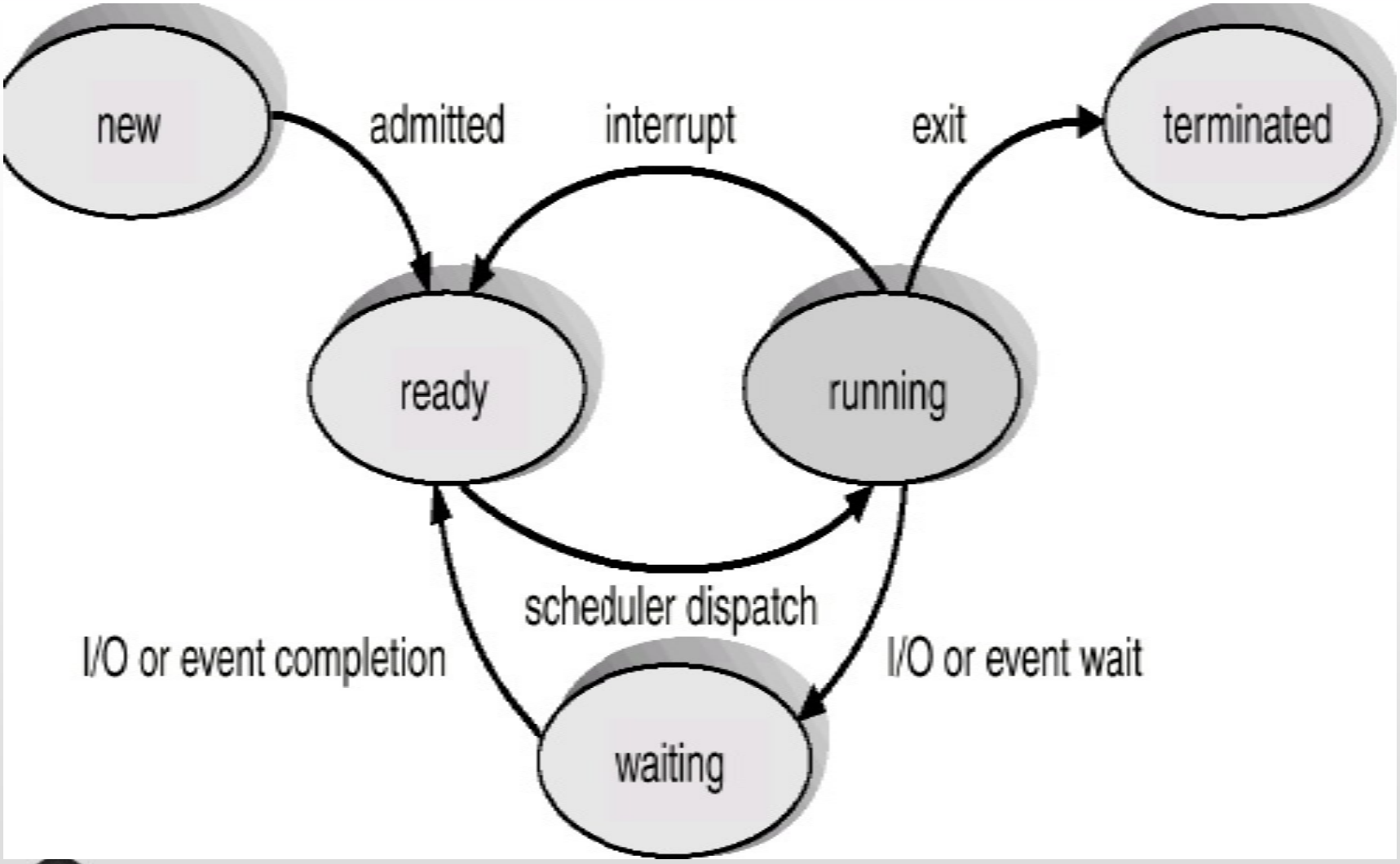
Los temas vistos en estas diapositivas han sido mayormente extraídos del libro de William Stallings (Sistemas Operativos: Aspectos internos y principios de diseño) y del libro de Silberschatz (Operating Systems Concepts)

Estados de un proceso

En su ciclo de vida, un proceso pasa por diferentes estados.

- ✓ Nuevo (new)
- ✓ Listo para ejecutar (ready)
- ✓ Ejecutándose (running)
- ✓ En espera (waiting)
- ✓ Terminado (terminated)

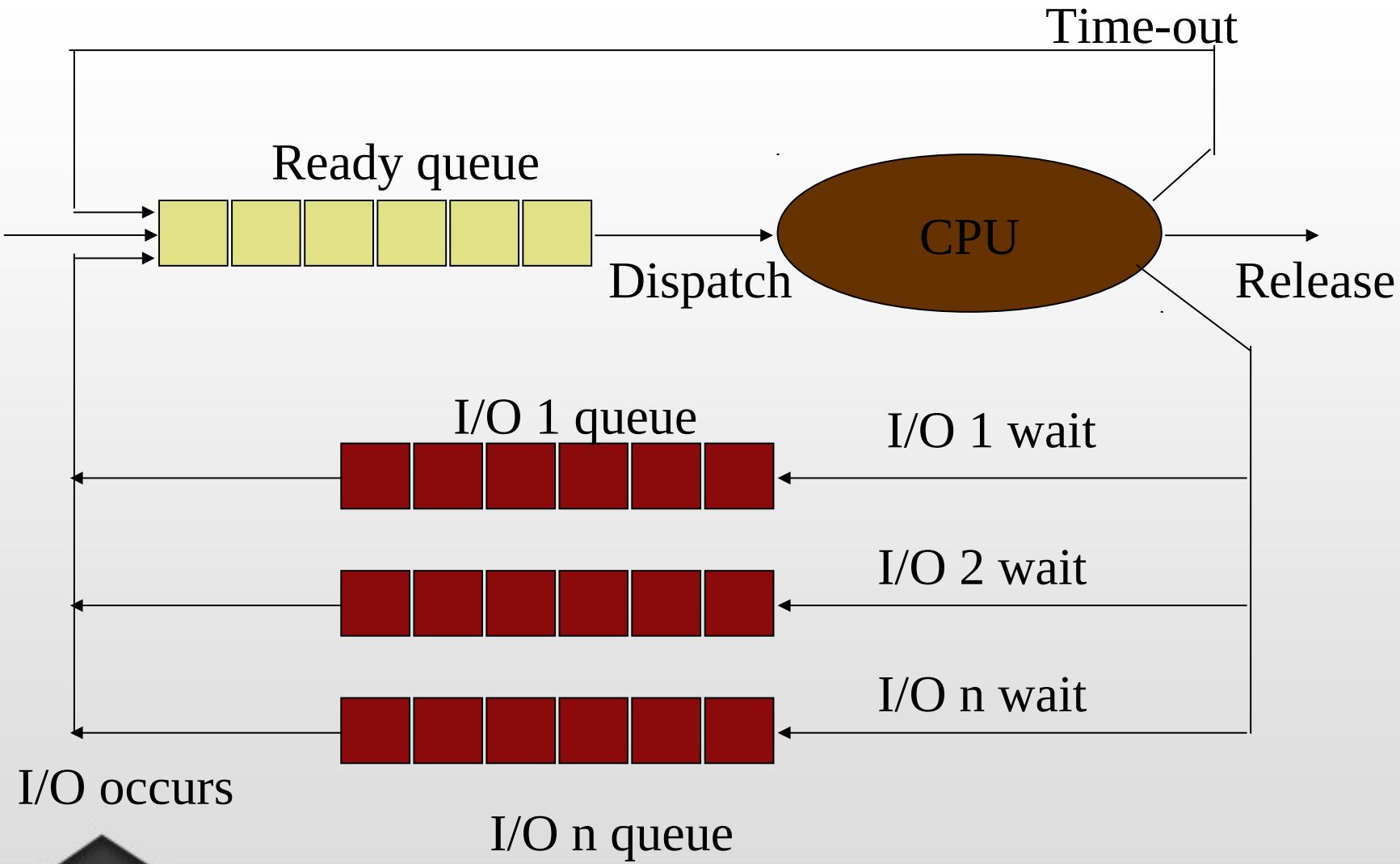
Estados de un proceso (cont.)



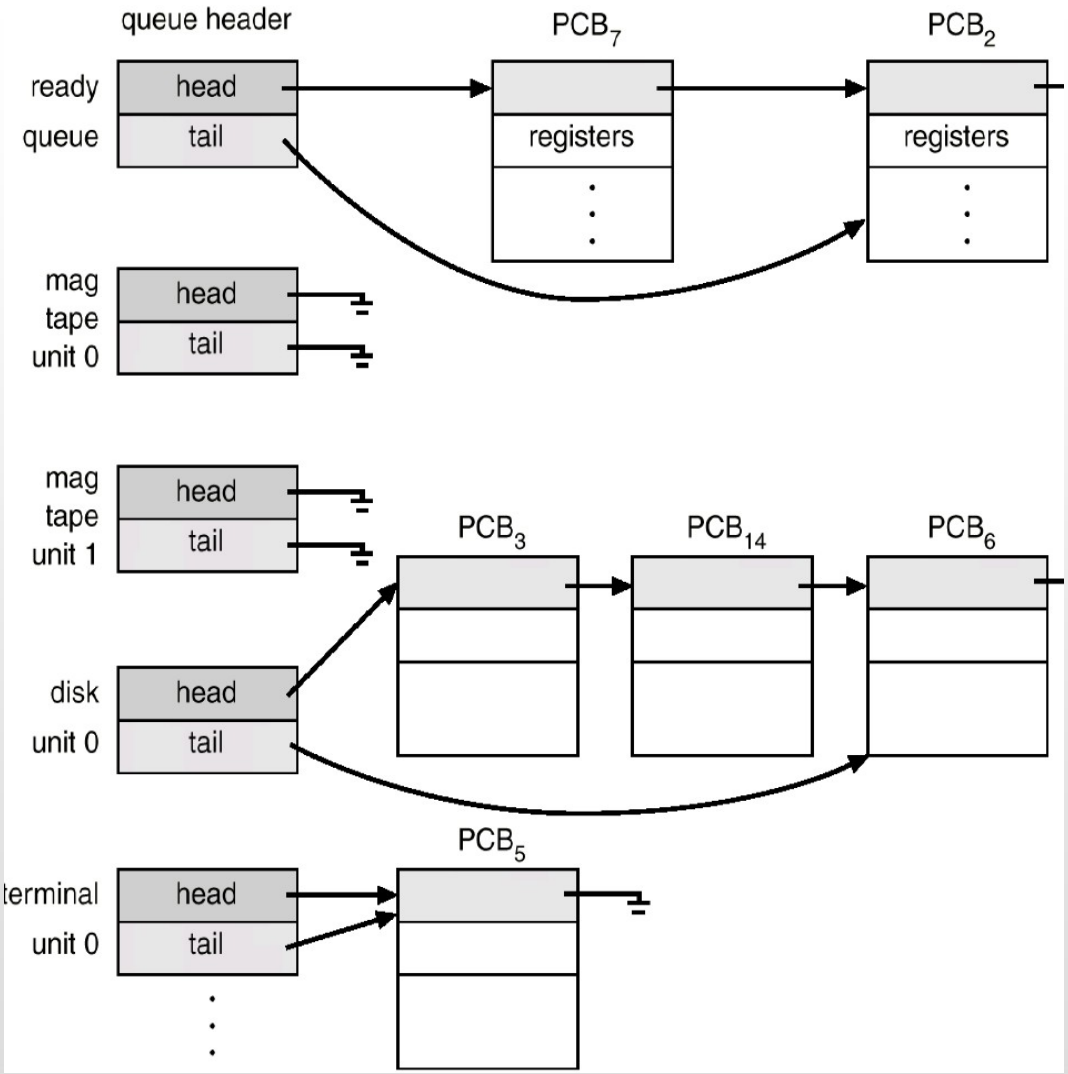
Colas en la planificación de procesos

- Para realizar la planificación, el SO utiliza la PCB de cada proceso como una abstracción del mismo
- Las PCB se enlazan en Colas siguiendo un orden determinado
- Ejemplos
  - Cola de trabajos o procesos
    - Contiene todas las PCB de procesos en el sistema
  - Cola de procesos listos
    - PCB de procesos residentes en memoria principal esperando para ejecutarse
  - Cola de dispositivos
    - PCB de procesos esperando por [redacted] I/O

Colas en la planificación de procesos (cont.)



Colas en la planificación de procesos (cont.)



Módulos de la planificación

- Son módulos (SW) del Kernel que realizan distintas tareas asociadas a la planificación.
- Se ejecutan ante determinados eventos que así lo requieren:
  - Creación/Terminación de procesos
  - Eventos de Sincronización o de E/S
  - Finalización de lapso de tiempo
  - Etc

Módulos de la planificación (cont.)

- ☑ Scheduler de long term
- ☑ Scheduler de short term
- ☑ Scheduler de medium term

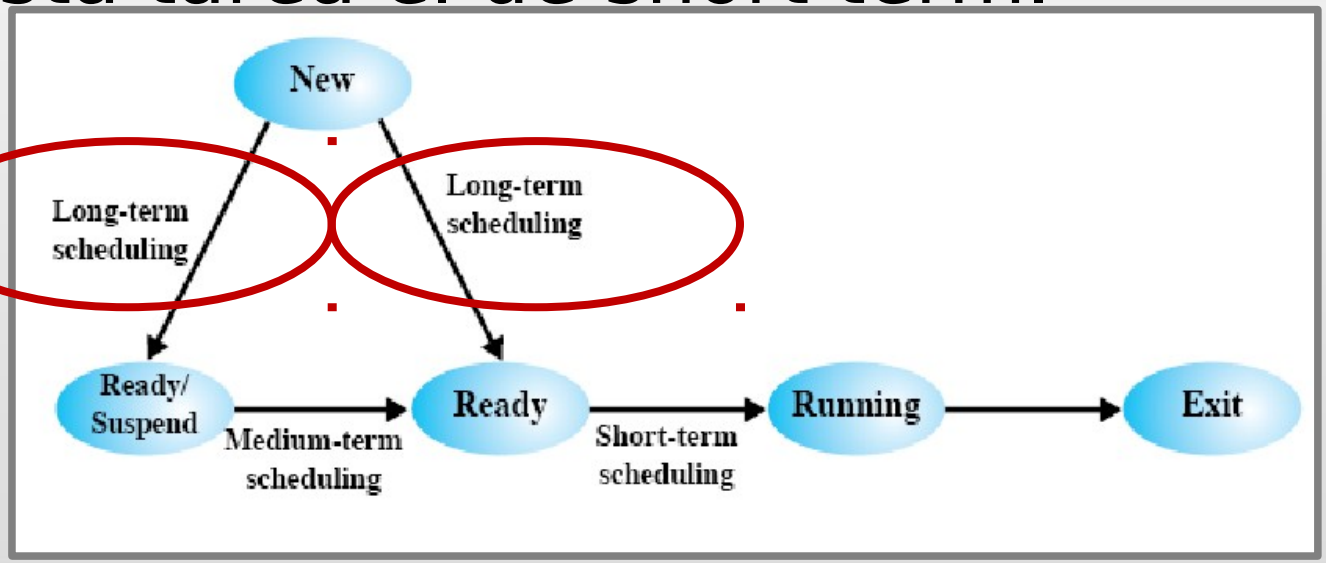
Su nombre proviene de la frecuencia de ejecución.

Módulos de la planificación (cont.)

- Otros módulos: **Dispatcher** y **Loader.**
- Pueden no existir como módulos separados de los schedulers vistos, pero la función debe cumplirse.
- **Dispatcher**: hace cambio de contexto, cambio de modo de ejecución...”despacha” el proceso elegido por el *Short Term* (es decir, “salta” a la instrucción a ejecutar).
- **Loader**: carga en memoria el proceso elegido por el *long term*.

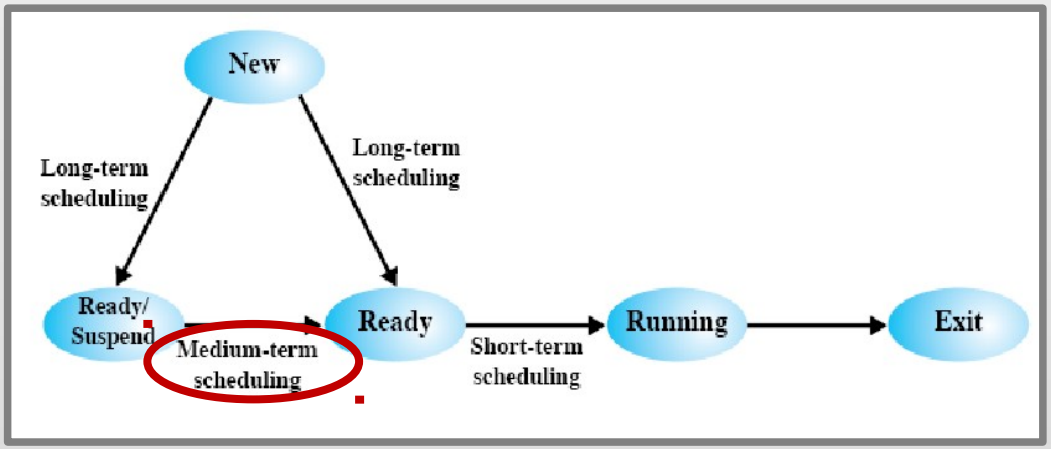
Long term Scheduler

- Controla el *grado de multiprogramación*, es decir, la cantidad de procesos en memoria.
- Puede no existir este scheduler y absorber esta tarea el de short term.



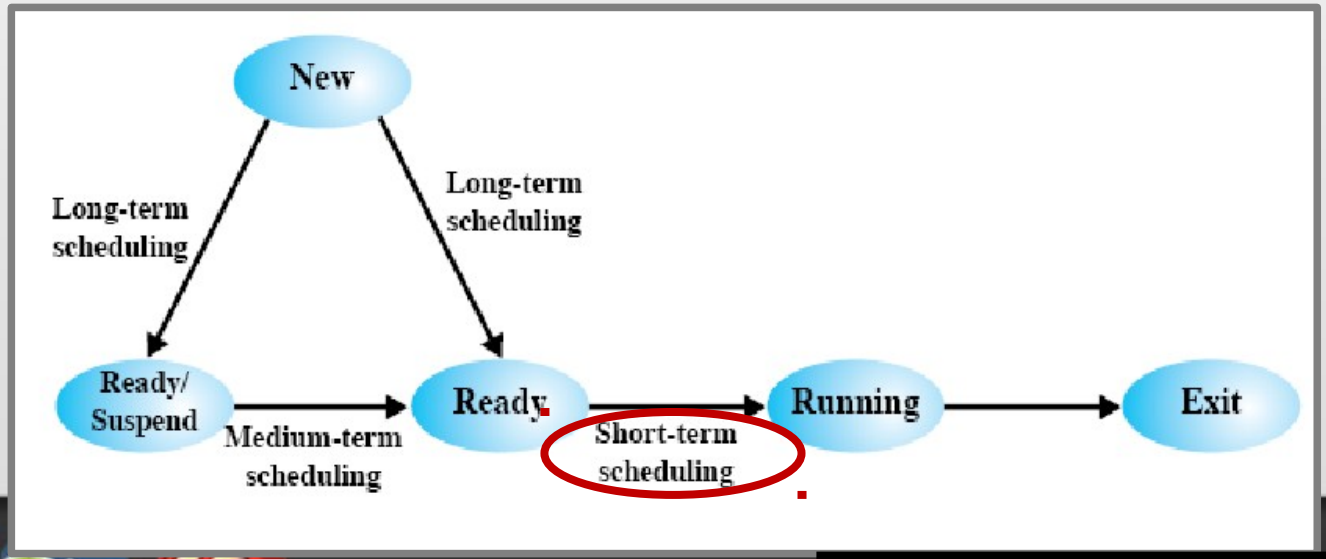
Medium Term Scheduler (swapping)

- Si es necesario, reduce el grado de multiprogramación
- Saca temporalmente de memoria los procesos que sea necesario para mantener el equilibrio del sistema.
- Términos asociados: *swap out* (sacar de memoria), *swap in* (volver a memoria).

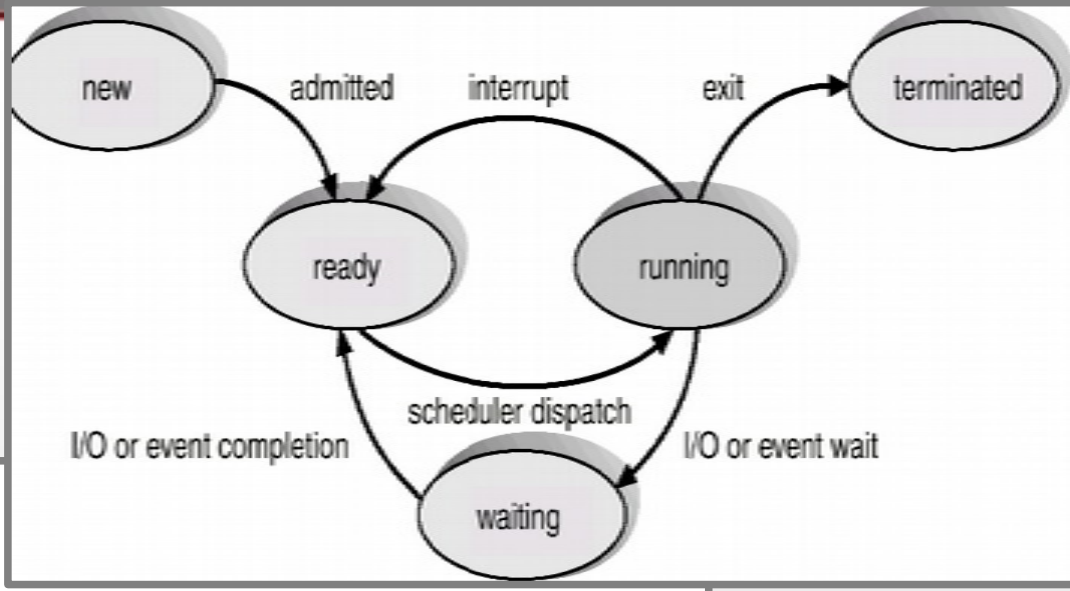


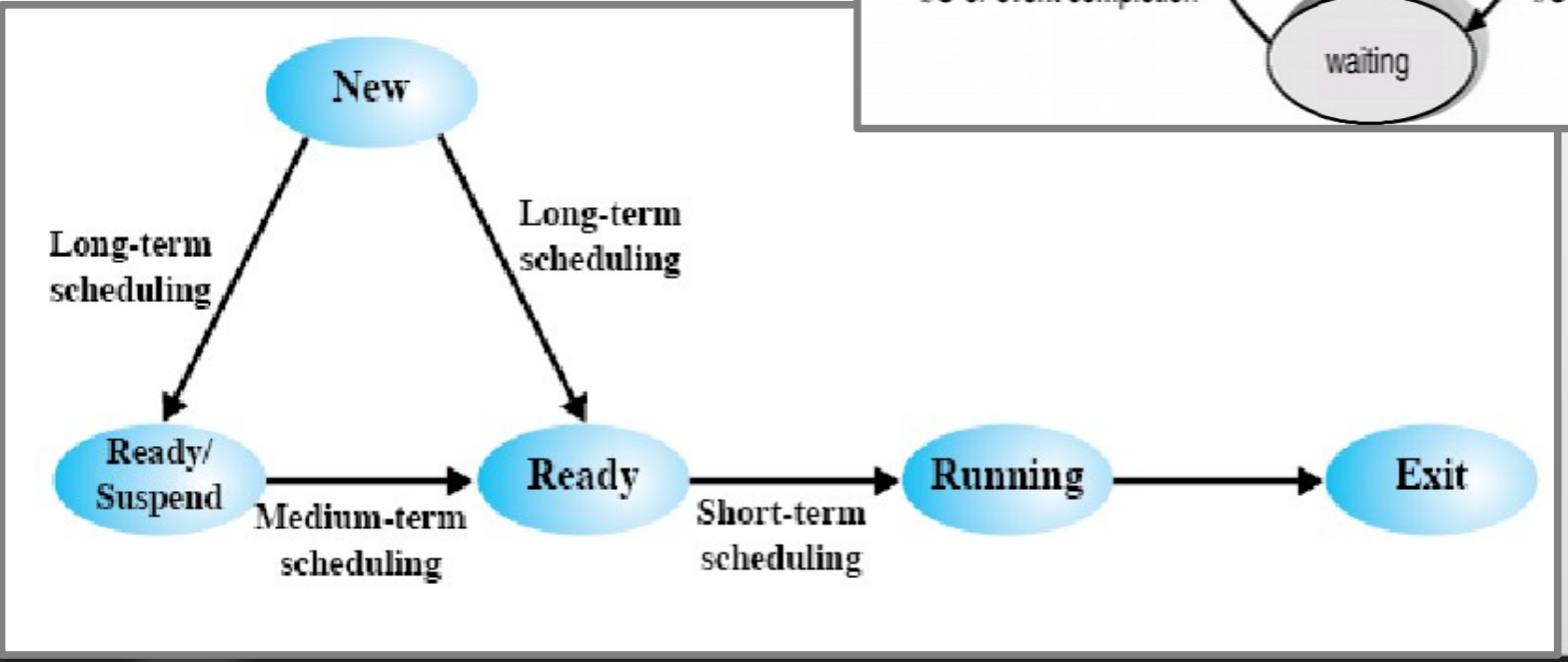
Short Term Scheduler

- Decide a cuál de los procesos en la cola de listos se elige para que use la CPU.
- Términos asociados: apropiativo, no apropiativo, algoritmo de scheduling

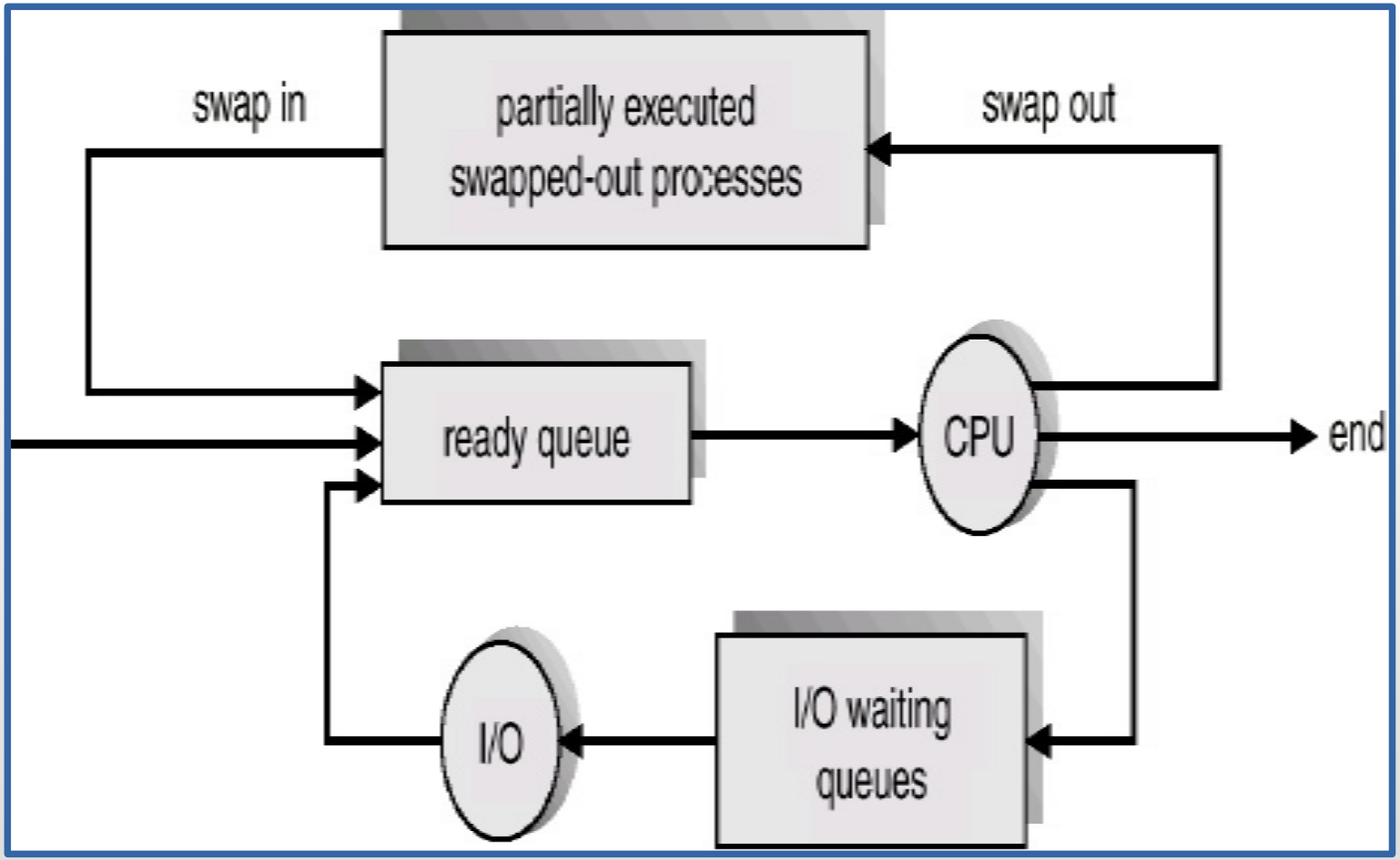


Estados y schedulers



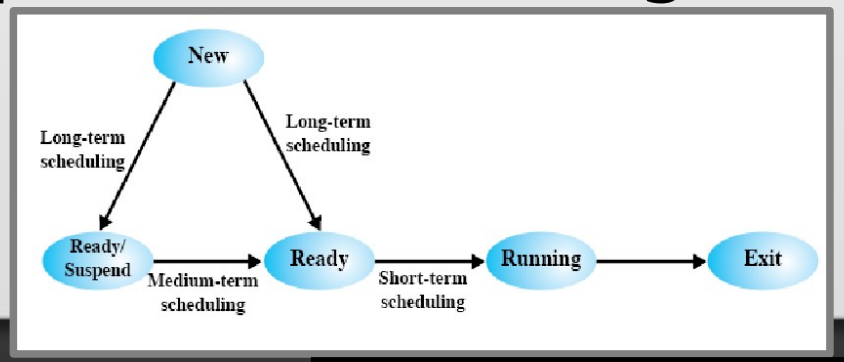


Procesos en espera y swapeados



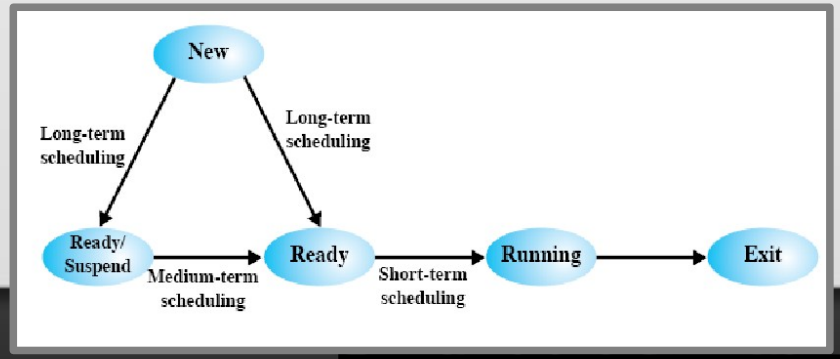
Sobre el estado nuevo (new)

- Un usuario “dispara” el proceso. Un proceso es creado por otro proceso: su proceso padre.
- En este estado se crean las estructuras asociadas, y el proceso queda en la *cola de procesos*, normalmente en espera de ser cargado en memoria



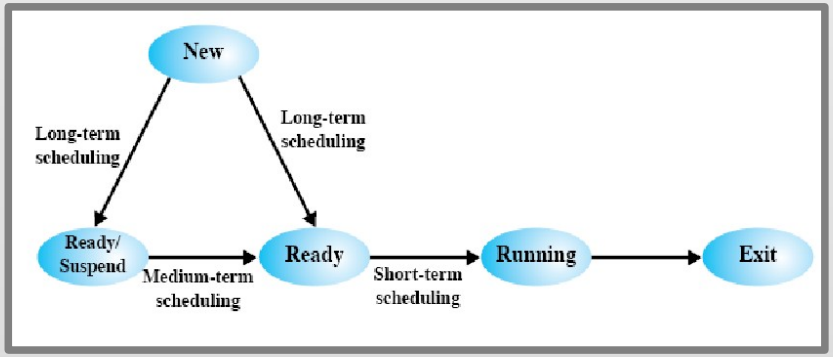
Sobre el estado listo (ready)

- Luego que el scheduller de largo plazo eligió al proceso para cargarlo en memoria, el proceso queda en estado listo
- El proceso sólo necesita que se le asigne CPU
- Está en la cola de procesos listos (ready queue)



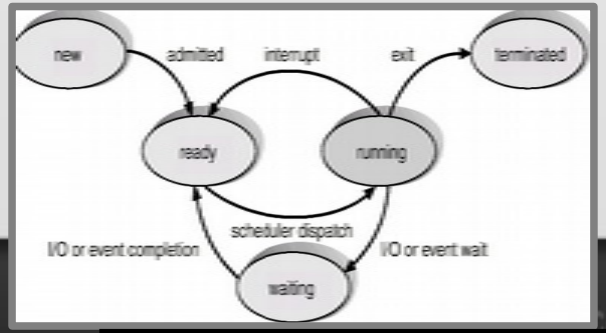
Sobre el estado en ejecución (running)

- El scheduler de corto plazo lo eligió para asignarle CPU
- Tendrá la CPU hasta que se termine el período de tiempo asignado (quantum o time slice), termine o hasta que necesite realizar alguna operación de E/S



Sobre el estado de espera (waiting)

- El proceso necesita que se cumpla el evento esperado para continuar.
- El evento puede ser la terminación de una E/S solicitada, o la llegada de una señal por parte de otro proceso.
- Sigue en memoria, pero no tiene la CPU.
- Al cumplirse el evento, pasará al estado de listo.



Transiciones

- **New-Ready**: Por elección del scheduler de largo plazo (carga en memoria)
- **Ready-Running**: Por elección del scheduler de corto plazo (asignación de CPU)
- **Running-Waiting**: el proceso “se pone a dormir”, esperando por un evento.
- **Waiting-Ready**: Terminó la espera y compite nuevamente por la CPU.

Caso especial: running-ready

- Cuando el proceso termina su quantum (franja de tiempo) sin haber necesitado ser interrumpirlo por un evento, pasa al estado de ready, para competir por CPU, *pues no está esperando por ningún evento*...
- Se trata de un caso distinto a los anteriores, porque el procesos es expulado de la CPU contra su voluntad
- Esta situación se da en algoritmos apropiativos

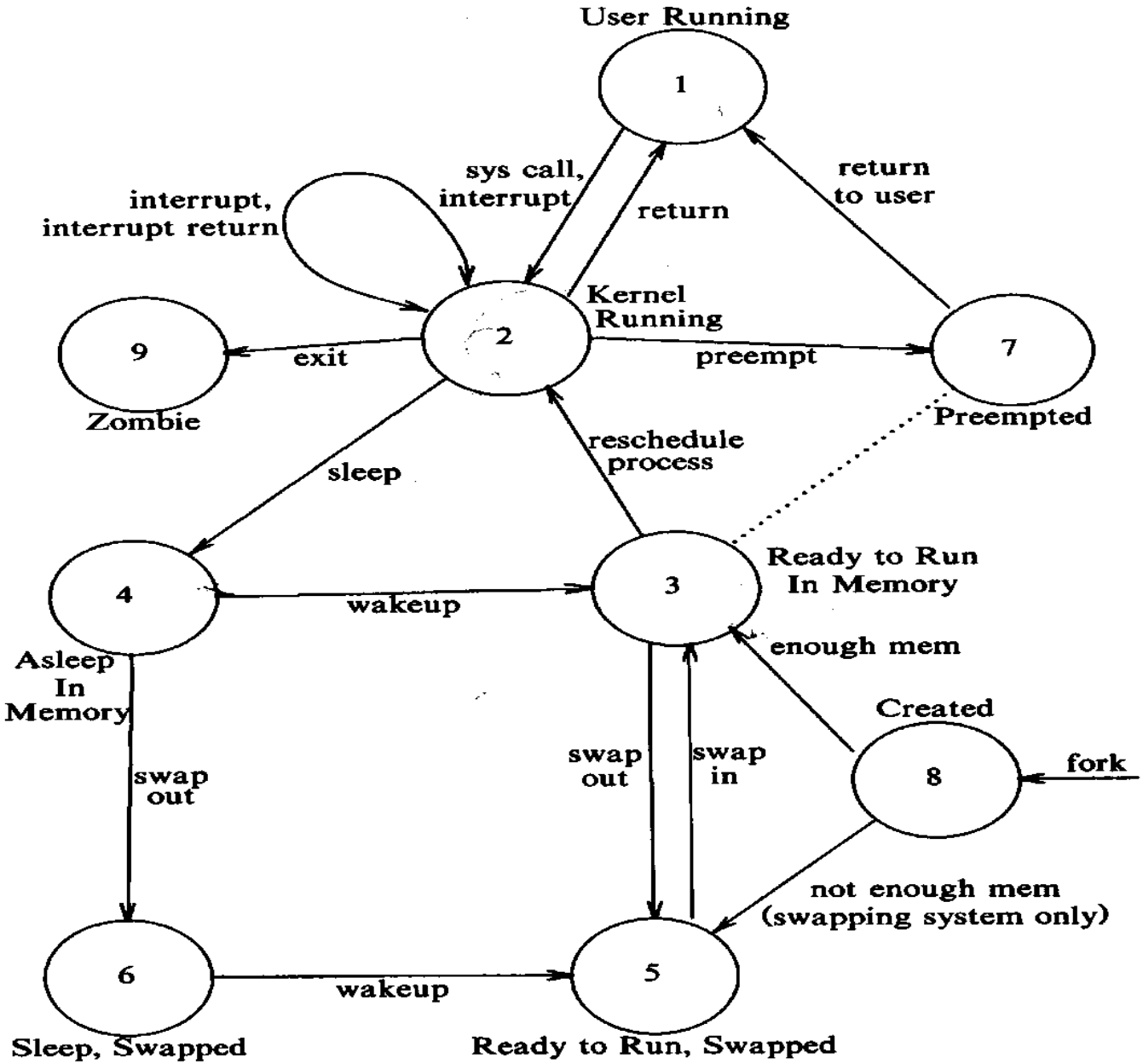


Figure 6.1. Process State Transition Diagram

Diagrama incluyendo swapping

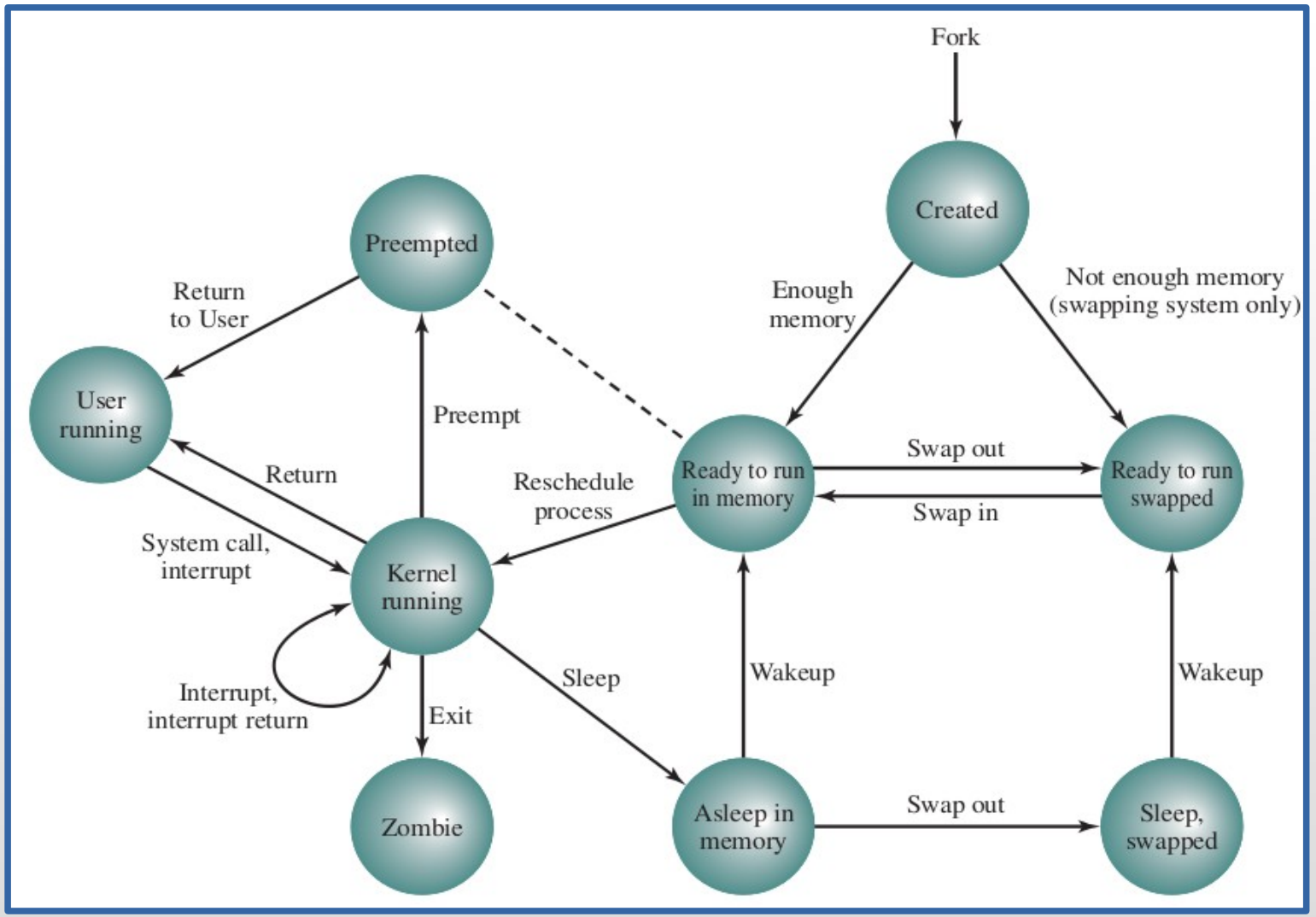
Explicación por estado

- ☑ 1. Ejecución en modo usuario
- ☑ 2. Ejecución en modo kernel
- ☑ 3. El proceso está listo para ser ejecutado cuando sea elegido.
- ☑ 4. Proceso en espera en memoria principal.
- ☑ 5. Proceso listo, pero el swapper debe llevar al proceso a memoria ppal antes que el kernel lo pueda elegir para ejecutar.

Explicación por estado (cont.)

- 6. Proceso en espera en memoria secundaria.
- 7. Proceso retornando desde el modo kernel al user. Pero el kernel se apropia, hace un context switch para darle la CPU a otro proceso.
- 8. Proceso recientemente creado y en transición: existe, pero aun no está listo para ejecutar, ni está dormido.
- 9. El proceso ejecutó la system call *exit y está en estado zombie*. Ya no existe más, pero se registran datos sobre su uso, codigo resultante del exit. Es el estado final.

Diagrama de transiciones UNIX



Introducción a los Sistemas Operativos

Procesos - III

I.S.O.

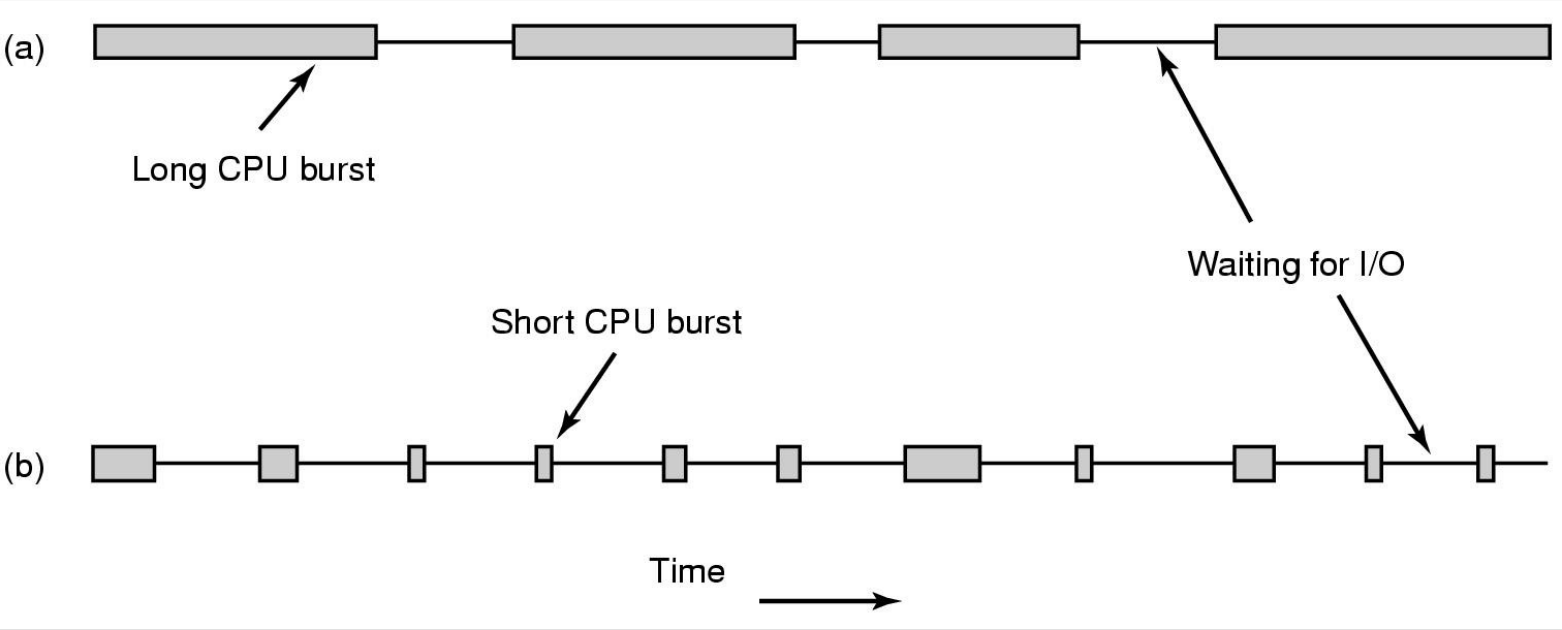
- ☑ Versión: Septiembre 2019

Palabras Claves: Procesos, Planificación, FCFS, SJF, Round Robin, SRTF, Prioridades, Algoritmos Apropiativos y Algoritmos No Apropiativos

Los temas vistos en estas diapositivas han sido mayormente extraídos del libro de Andrew S. Tanenbaum (Sistemas Operativos Modernos)

Comportamiento de los procesos

- Procesos alternan ráfagas de CPU y de I/O.



Comportamiento de los procesos (cont.)

- CPU-bound
  - Mayor parte del tiempo utilizando la CPU
- I/O-bound (I/O = E/S)
  - Mayor parte del tiempo esperando por I/O
- La velocidad de la CPU es mucho mas rápida que la de los dispositivos de E/S
  - Pensar: Necesidad de atender rápidamente procesos I/O-bound para mantener el dispositivo ocupado y aprovechar la CPU para procesos CPU-bound

Planificación

- Planificación:
  - Necesidad de determinar cual de todos los procesos que están listos para ejecutarse, se ejecutará a continuación en un ambiente multiprogramado
- Algoritmo de Planificación
  - Algoritmo utilizado para realizar la planificación del sistema

Algoritmos Apropiativos y No Apropiativos

- En los algoritmos Apropiativos (preemtive) existen situaciones que hacen que el proceso en ejecución sea expulsado de la CPU
- En los algoritmos No Apropiativo (nonpreemptive) los procesos se ejecutan hasta que el mismo (por su propia cuenta) abandone la CPU
  - Se bloquea por E/S o finaliza
  - No hay decisiones de planificación durante las interrupciones de reloj

Categorías de los Algoritmos de Planificación

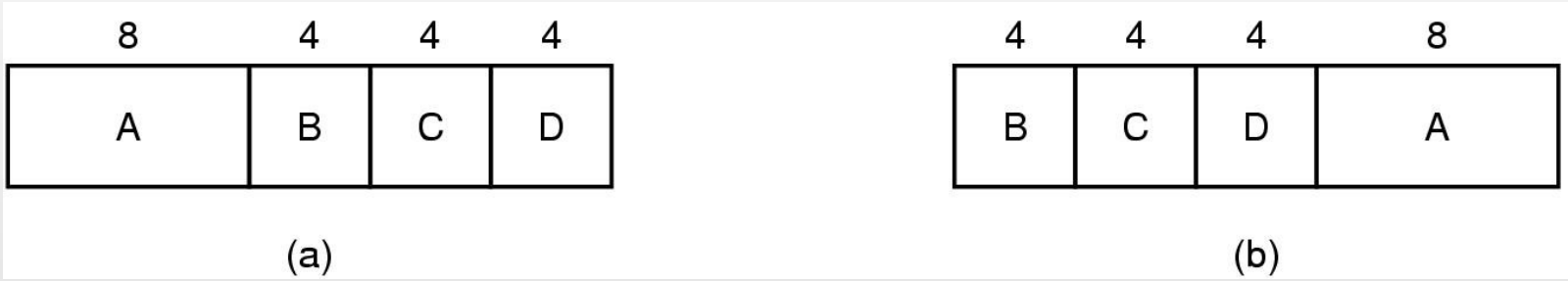
- Según el ambiente es posible requerir algoritmos de planificación diferentes, con diferentes metas:
  - Equidad: Otorgar una parte justa de la CPU a cada proceso
  - Balance: Mantener ocupadas todas las partes del sistema
- Ejemplos:
  - Procesos por lotes (batch)
  - Procesos Interactivos
  - Procesos en Tiempo Real

Procesos Batch

- No existen usuarios que esperen una respuesta en una terminal.
- Se pueden utilizar algoritmos no apropiativos
- Metas propias de este tipo de algoritmos:
  - Rendimiento: Maximizar el número de trabajos por hora
  - Tiempo de Retorno: Minimizar los tiempos entre el comienzo y la finalización
  - El Tiempo es espera se puede ver afectado
  - Uso de la CPU: Mantener la CPU ocupada la mayor cantidad de tiempo posible

Procesos Batch (cont.)

- Ejemplos de Algoritmos:
  - FCFS – First Come First Served
  - SJF – Shortest Job First

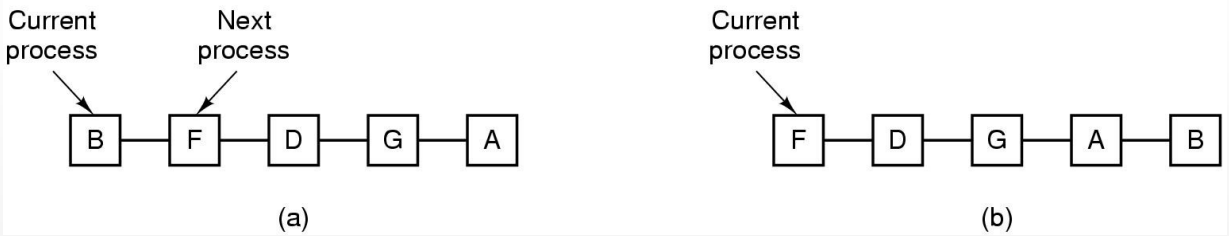


Procesos Interactivos

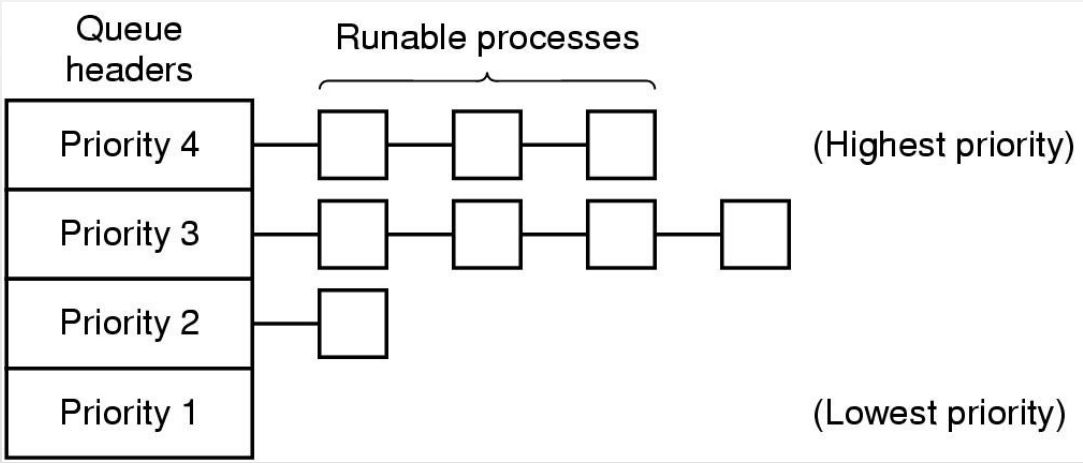
- No solo interacción con los usuarios
  - Un servidor, necesita de varios procesos para dar respuesta a diferentes requerimientos
- Son necesarios algoritmos apropiativos para evitar que un proceso acapare la CPU
- Metas propias de este tipo de algoritmos:
  - Tiempo de Respuesta: Responder a peticiones con rapidez
  - Proporcionalidad: Cumplir con expectativas de los usuarios
    - Si el usuario le pone STOP al reproductor de música, que la música deje de ser reproducida en un tiempo considerablemente corto.

Procesos Interactivos (cont.)

- Ejemplos de Algoritmos:
  - Round Robin



  - Prioridades
  - Colas Multinivel



  - SRTF – Shortest remaining time first

Política Versus Mecanismo

- Existen situaciones en las que es necesario que la planificación de uno o varios procesos se comporte de manera diferente
- El algoritmo de planificación debe estar parametrizado, de manera que los procesos/usuarios pueden indicar los parámetros para modificar la planificación

Política Versus Mecanismo (cont.)

- El Kernel implementa el mecanismo
- El usuario/proceso/administrador utiliza los parámetros para determinar la Política
- Ejemplo:
  - Un algoritmo de planificación por prioridades y una System Call que permite modificar la prioridad de un proceso (man nice)
  - Un proceso puede determinar las prioridades de los procesos que el crea, según la importancia de los mismos.

Introducción a los Sistemas Operativos

Procesos - IV

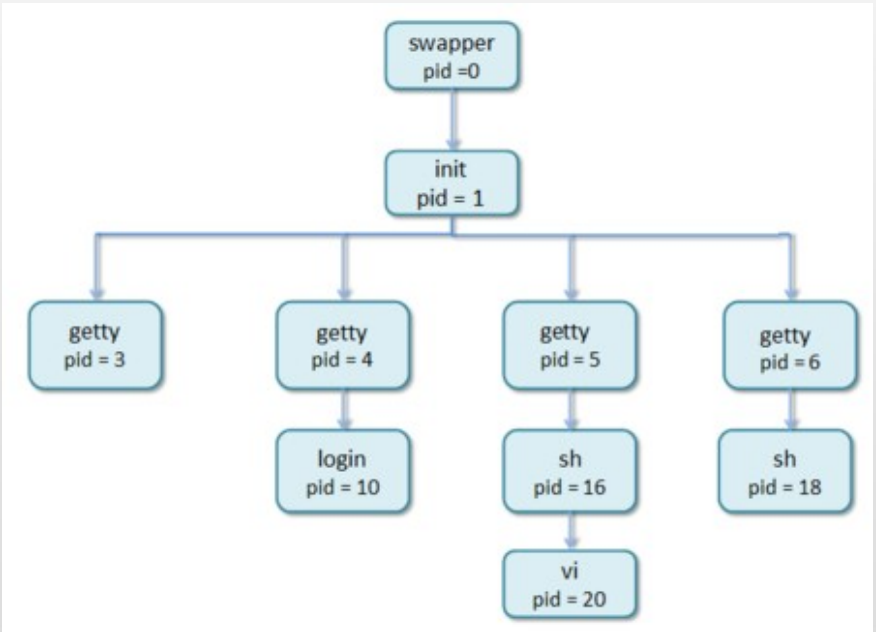
I.S.O.

- ☑ Versión: Septiembre 2019
- ☑ Palabras Claves: Procesos, Linux, Windows, Creación, Terminación, Fork, Execve, Relación entre procesos

Los temas vistos en estas diapositivas han sido mayormente extraídos del libro de Andrew S. Tanenbaum (Sistemas Operativos Modernos)

Creación de procesos

- Un proceso es creado por otro proceso
- Un proceso padre tiene uno o más procesos hijos.
- Se forma un árbol de procesos



Actividades en la creación

- ☑ Crear la PCB
- ☑ Asignar PID (Process IDentification) único
- ☑ Asignarle memoria para regiones
  - Stack, Text y Datos
- ☑ Crear estructuras de datos asociadas
  - Fork (copiar el contexto, regiones de datos, text y stack)

Relación entre procesos Padre e Hijo

Con respecto a la Ejecución:

- ☑ El padre puede continuar ejecutándose concurrentemente con su hijo
- ☑ El padre puede esperar a que el proceso hijo (o los procesos hijos) terminen para continuar la ejecución.

Relación entre procesos Padre e Hijo (cont.)

Con respecto al Espacio de Direcciones:

- ☑ El hijo es un duplicado del proceso padre (caso Unix)
- ☑ Se crea el proceso y se le carga adentro el programa (caso Windows)

Creación de Procesos

- En UNIX:
  - system call **fork()** crea nuevo proceso
  - system call **execve(),** generalmente usada después del fork, carga un nuevo programa en el espacio de direcciones.
- En Windows:
  - system call **CreateProcess()** crea un nuevo proceso y carga el programa para ejecución.

¿Como funciona fork? (1)

Padre

```
        //Instrucciones previas
PC ⟶    nue = fork()
        if nue == 0
           // hijo
        elsif nue > 0
           // padre
           // nue es el PID del hijo
        else
           // error
        end
```

PC = Program Counter

¿Como funciona fork? (2)

Padre

PC →

```
//Inst. previas
nue = fork()
if nue == 0
   // hijo
elsif nue > 0
   // padre
   // nue = PID hijo
else
   // error
end
```

Hijo

PC →

```
//Inst. previas
nue = fork()
if nue == 0
   // hijo
elsif nue > 0
   // padre
   // nue = PID hijo
else
   // error
end
```

PC = Program Counter

Ejemplo SysCall fork

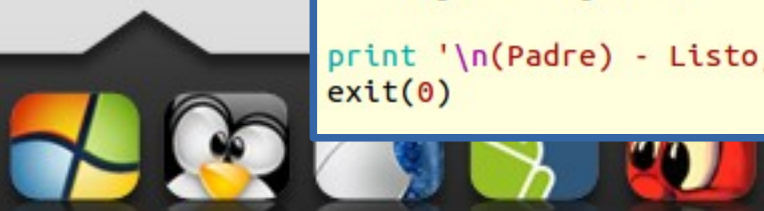
```
#
# El padre puede terminar antes que los hijos
#

import os, time
hijos = 0
print '\n\n\nSoy el PROCESO PADRE. PID: ', os.getpid() , 'y tengo', hijos, 'hijos\n'
print '\n\n\tQuiero tener un hijo? (sn)'
respuesta = raw_input()
while (respuesta <> 'n'):
    newpid = os.fork()
    if newpid == 0:
        #
        # Seccion del hijo
        #
        time.sleep(30)
        print '\t\t\t\t', os.getpid(),' - Me aburro. Me voy a jugar a la PLAY'
        exit(0)
    else:
        #
        # seccion del padre
        #
        hijos = hijos + 1
        print '\t\tTuve un hijo!!!! Tiene el PID: ', newpid

    print '\n\tQuiero tener otro hijo? (s|n)'
    respuesta = raw_input()

print '\nBueno, hasta aca llegue. Me voy a dormir. Ya con', hijos ,'hijos es suficiente'
exit(0)
```

Terminación de procesos

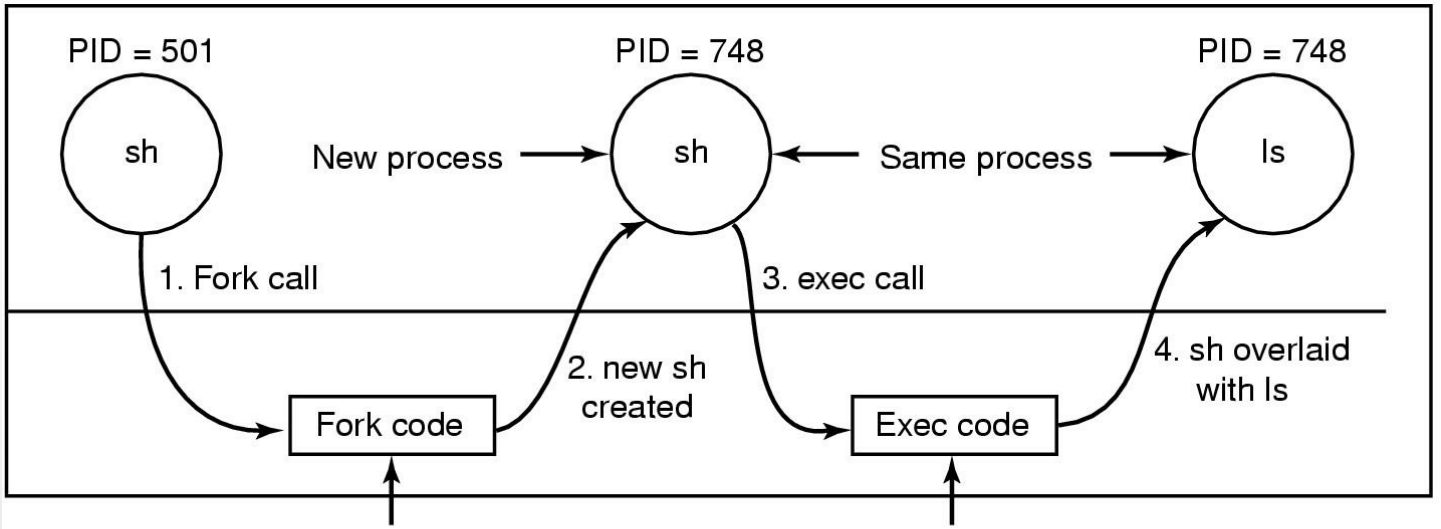
- Ante un (**exit**), se retorna el control al sistema operativo
  - El proceso padre puede esperar recibir un código de retorno (via **wait**). Generalmente se lo usa cuando se requier que el padre espere a los hijos.
- Proceso padre puede terminar la ejecución de sus hijos (**kill**)
  - La tarea asignada al hijo se terminó
  - Cuando el padre termina su ejecución
    - Habitualmente no se permite a los hijos continuar, pero existe la opción.
    - Terminación en cascada

Ejemplo SysCall fork+wait+exit

```
#
# El padre espera que terminen sus hijos antes de retirarse
#
import os, time
hijos = 0
print '\n\n\nSoy el PROCESO', os.getpid() , 'y tengo', hijos, 'hijos\n'
while True:
    newpid = os.fork()
    if newpid == 0:
        #
        # Seccion del hijo
        #
        time.sleep(30)
        print '\t(Hijo', os.getpid(), ') - Se va a jugar a la play'
        exit(0)
    else:
        #
        # Seccion del padre
        #
        hijos = hijos + 1
        print '\t(Padre) Tuve un hijo!!!! Se llama', newpid
    if raw_input( ) == 'q': break

print '\n(Padre) - VAYAN A JUGAR A LA PELOTA!!!!'

while hijos > 0:
    os.wait()
    print '\t(Padre) - Joya, uno menos para cuidar!!!\n'
    hijos = hijos - 1

print '\n(Padre) - Listo, se fueron todos, me voy a dormir'
exit(0)
```



Fork / Exec – Ejemplo



Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

Creación y Terminación de Procesos

- Un ejemplo de una CLI (command line interface) o shell

```
1  while (TRUE) {                                  /* repeat forever */
2    type_prompt( );                               /* display prompt */
3    read_command (command, parameters)            /* input from terminal */
4
5    if (fork() != 0) {                            /* fork off child process */
6      /* Parent code */
7      waitpid( -1, &status, 0);                   /* wait for child to exit */
8
9    } else {
10     /* Child code */
11     execve (command, parameters, 0);            /* execute command */
12   }
13 }
14
```

Ejemplo SysCall fork+execv

```
#
# SHELL
#
import os, time
print '''
-------------------------------------------------------
---------- Esta es la ISO DIR/LS SHELL ----------
-------------------------------------------------------
# exit (para salir)
'''
cmd = raw_input("iso:\> ")
while cmd <> 'exit':

    if cmd == '':
        cmd = raw_input("iso:\> ")
    else:
        newpid = os.fork()
        if newpid == 0:
            # Seccion del hijo
            lista = cmd.split(' ')
            os.execvp(lista[0], lista)
            print "Imprimir AAAAAAAAAA"
            exit(0)
            print "Imprimir BBBBBBBBBB"
        else:
            # Seccion del padre
            #os.wait()
            cmd = raw_input("iso:\> ")
exit(0)
```

Procesos Cooperativos e Independientes

- *Independiente*: el proceso no afecta ni puede ser afectado por la ejecución de otros procesos. No comparte ningún tipo de dato.
- *Cooperativo*: afecta o es afectado por la ejecución de otros procesos en el sistema.

Para qué sirven los procesos cooperativos?

- ☑ Para compartir información (por ejemplo, un archivo)
- ☑ Para acelerar el cómputo (separar una tarea en sub-tareas que cooperan ejecutándose paralelamente)
- ☑ Para planificar tareas de manera tal que se puedan ejecutar en paralelo.

System Calls - Unix

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Syscalls de Procesos

System calls - Windows

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semapahore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

Syscalls de Procesos

Introducción a los Sistemas Operativos / Conceptos de Sistemas Operativos

Procesos – Anexo I

Algoritmos Apropiativos y No Apropiativos

I.S.O.

- ☑ Versión: Octube 2020

Palabras Claves: Procesos, Planificación, FCFS, SJF, Round Robin, SRTF, Prioridades, Algoritmos Apropiativos y Algoritmos No Apropiativos

Los temas vistos en estas diapositivas han sido mayormente extraídos del libro de Andrew S. Tanenbaum (Sistemas Operativos Modernos)

Algoritmos Apropiativos y No Apropiativos

- La apropiación esta relaciona al recurso CPU
- En los algoritmos **Apropiativos** (preemtive) existen situaciones que hacen que el proceso en ejecución sea expulsado, por el planificador de corto plazo, de la CPU
- En los algoritmos **No Apropiativos** (nonpreemptive) los procesos se ejecutan hasta que el mismo (por su propia cuenta) abandone la CPU

Ejemplos de Algoritmos

- ☑ Apropiativos:
  - Round Robin
  - SRTF
  - Prioridades Apropiativo
- ☑ No Apropiativos:
  - FCFC
  - SJF
  - Prioridades No apropiativo

No apropiativos

- El proceso deja el estado de ejecución solo cuando:
  - Termina (Syscall Exit)
  - Se bloquea voluntariamente (SysCall wait, sleep, etc)
  - Solicita una operación de E/S (Syscall Read, Write, etc)

Apropiativos

- El proceso puede ser expulsado de la CPU según la planificación implementada:
  - Se le termina su quantum (Algoritmo round robin)
  - Llega a la cola de listos un proceso de mayor prioridad (Algoritmo prioridades apropiativo)
  - Llega a la cola de listos un proceso con menor tiempo restante (Algoritmo SRTF)

Versión: Agosto 2013
Autor: Pérez, Juan Pablo

Pasos que se suceden al llamar a una System Call

El siguiente cuadro resume, y ejemplifica, los pasos que se suceden desde el momento en que un proceso de usuario realiza un llamado a una System Call. Los pasos son generales y pueden variar en las implementaciones de cada Sistemas Operativos.

Referencias:

- ***User o Kernel Mode:*** Modo de ejecución en el que se encuentra la CPU
- ***Hard o Soft:*** Si el que realiza la operación es el Hardware o el Software
- ***Stack Utilizado:*** Indica si el stack que se esta utilizando es el de Usuario o de Kernel

User o Kernel Mode	***Hard o Soft***	***Stack Utilizado***	***Descripción***
U	S	U	1. El proceso de Usuario llama a una Syscall por medio de la GlibC
U	S	U	2. La GlibC pone los parámetros para la syscall en el Stack y eleva una interrupción
U	H	U	3. Cambia a Kernel Mode
K	H	U	4. Coloca el PC y PSW en el stack (puede que se coloquen mas registros, dependiente de la arquitectura)
K	H	U	5. Se coloca en el PC la dirección de la rutina de atención de interrupción que se extrae de la IDT y se continua la ejecución
K	S	U	6. Se sacan los parámetros a la syscall del stack
K	S	U	7. De ser necesario se pueden guardar otros mas registros del proceso actual en el Stack o en la PCB, depende de la implementación del SO
K	S	U	8. Se cambia a kernel stack, guardando la dirección del stack en User Mode en la PCB
K	S	K	9. Se colocan los parámetros para la syscall en el Stack
K	S	K	10. Se ejecuta la Syscall
K	S	K	11. Si la Syscall bloquea el Proceso
			11.1. De ser necesario se guarda mas información sobre el proceso bloqueado (registros, estados, etc.)
			11.2. Se ejecuta el Short Term Scheduller para seleccionar un nuevo proceso
			11.3. Se realiza el context switch
			11.3.1. Se cargan los registros del nuevo proceso
			11.3.2. Se acomoda la dirección del Stack, dejando apuntando a la dirección que el HW dejo

			previo a que el proceso seleccionado sea suspendido.
			11.3.3. Se acomodan los datos necesarios en la PCB o estructuras utilizadas
K	S	U	12. Se cambia a User Mode
U	S	U	13. Se ejecuta RET
U	H	U	14. Se sacan de la pila el PSW y PC
U	S	U	15. Continua la ejecución del proceso actual

Mas Información:

+ información sobre HW : http://en.wikipedia.org/wiki/X86_assembly_language#Registers
+ información sobre interrupciones: http://en.wikipedia.org/wiki/INT_%28x86_instruction%29
+ información sobre interrupciones: http://en.wikipedia.org/wiki/Interrupt
+ información sobre SysCalls: http://en.wikipedia.org/wiki/System_call
+ información syscalls en linux; http://www.tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html
+ información sobre context switch: http://wiki.osdev.org/Context_Switching
+ información sobre context switch: http://en.wikipedia.org/wiki/Context_switch
+ información sobre stack: http://wiki.osdev.org/Stack
+ información syscall en Windows: http://blogs.technet.com/b/ganand/archive/2007/12/23/how-do-transition-from-user-mode-to-kernel-mode-takes-place.aspx