Gestión de Procesos

Gestión de Procesos

- Estados de un proceso
- SO Como gestor de eventos
- SO Como gestor del recurso *Procesador*
- Contexto de un Proceso y PCB

[SGG]: capítulo 3.1
(y algo 3.2.3)

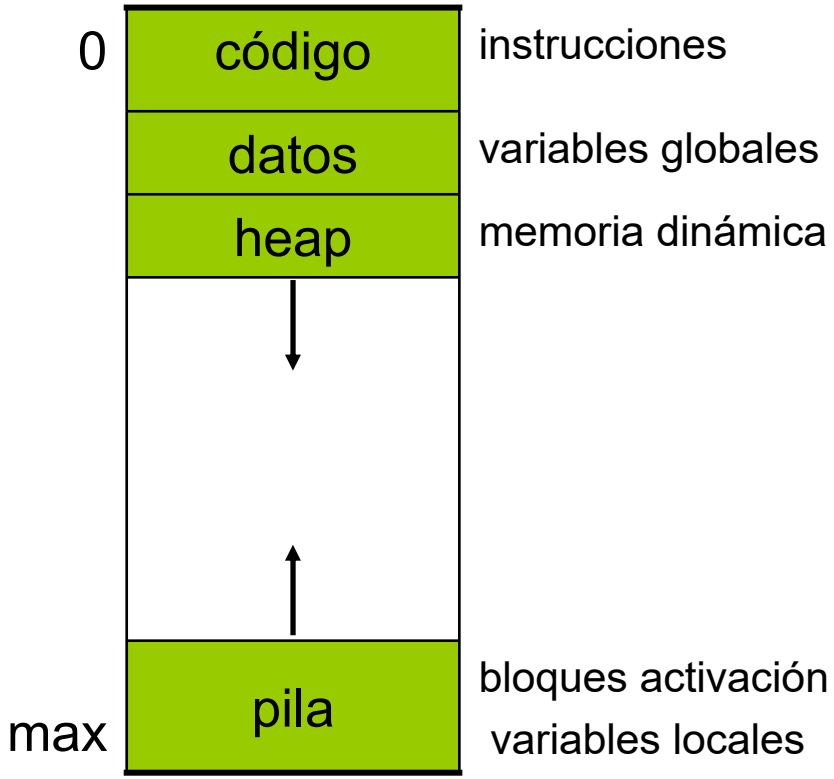
Proceso

- Definición de proceso:
  - Programa en ejecución
  - Unidad de trabajo en un SO moderno
- Nombres alternativos: Trabajo, tarea
- En un SO moderno se están ejecutando de forma concurrente
  - Procesos de usuario (modo usuario)
  - Procesos del sistema (modo kernel)
- En relación con los procesos, el SO se encarga de:
  - Crearlos y destruirlos (tanto de sistema como de usuario)
  - Planificación de procesos (scheduling)
  - Proveer mecanismos para sincronizar, comunicar y evitar bloqueos entre procesos

Proceso ≠ programa

- Programa es una entidad pasiva
  - Fichero ejecutable con instrucciones y descripción de variables globales que está en disco
- Proceso es una entidad activa
  - pc que indica la siguiente instrucción que hay que ejecutar
  - Un conjunto de recursos asignados: CPU, MEM, E/S
- El programa se convierte en proceso cuando se carga en memoria y se le asignan recursos para su ejecución.
- Dado un programa, puede haber varios procesos distintos (p. ej. navegadores, editores, etc. del mismo o distintos usuarios) Aunque estos procesos tienen la misma sección de código, difieren en la de datos, pila, registros, heap.
- Formas de ejecutar:
  - Doble clic en icono (GUI = Graphical User Interface)
  - Nombre fichero ejecutable (CLI = Command Line Interpreter = Intérprete Comandos)

Partes de un Proceso



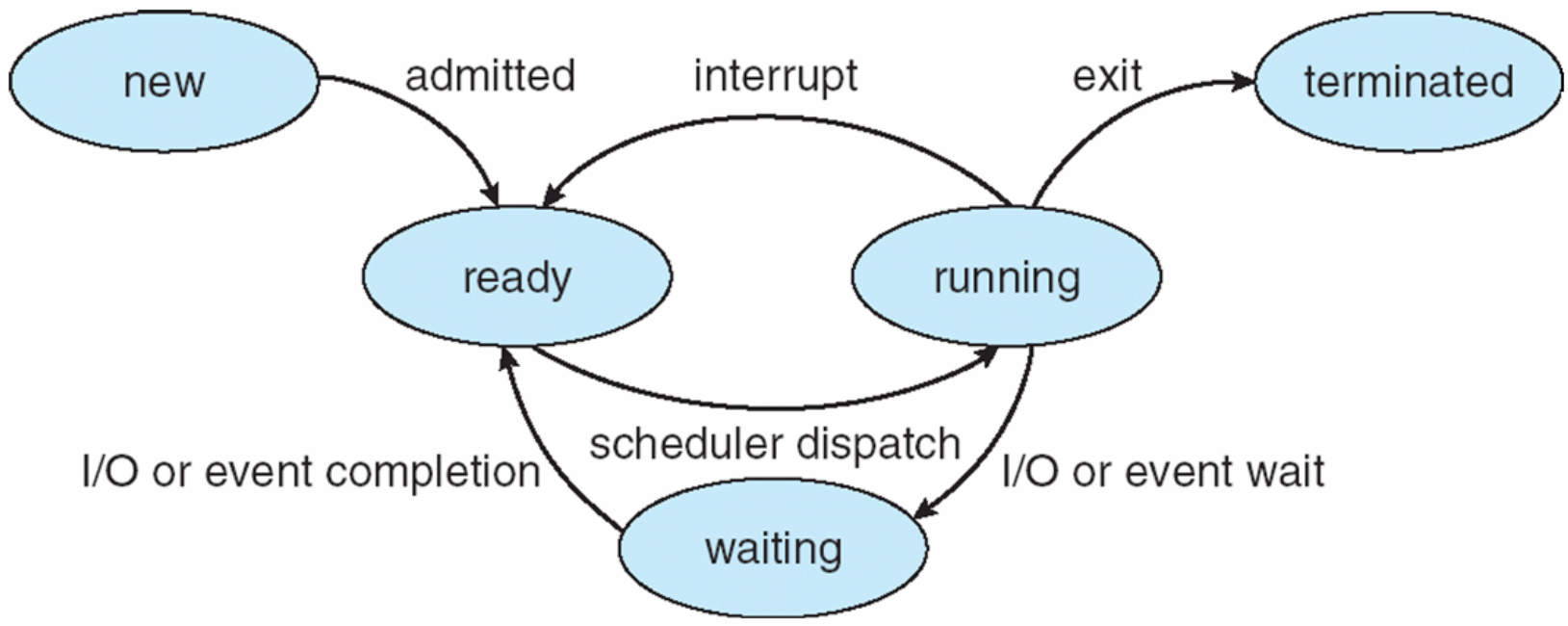
+ registros CPU
(pc, sp, cpsr(psw), ...)

Estados de un proceso

- **Nuevo:** El proceso se está creando (asignando recursos)
- **Ejecución:** La CPU está ejecutando instrucciones del proceso
- **Bloqueado:** El proceso está esperando la ocurrencia de algún evento (E/S, señal, mensaje)
- **Preparado:** El proceso está esperando la asignación de CPU (scheduler)
- **Terminado:** El proceso ha acabado (liberando recursos).

- En un instante dado, la CPU sólo está ejecutando 1 proceso
  - Si CPU de n nucleos, n procesos en ejecución en un instante determinado)
- Pero puede haber muchos más preparados para ejecutar en el momento en el que el proceso actual se bloquee, agote su quantum o llegue otro más prioritario.

Estados de un proceso



Evolución de un proceso

Ráfagas de ejecución separadas por E/S:

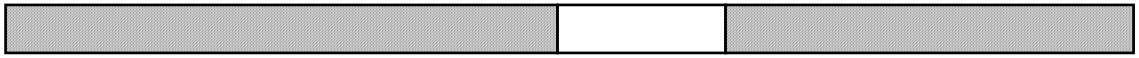
Tiempos de E/S usualmente largos

Repetir
Trabajo en CPU
Entrada / Salida
Hasta FIN

→ **Programar controlador**
Esperar respuesta

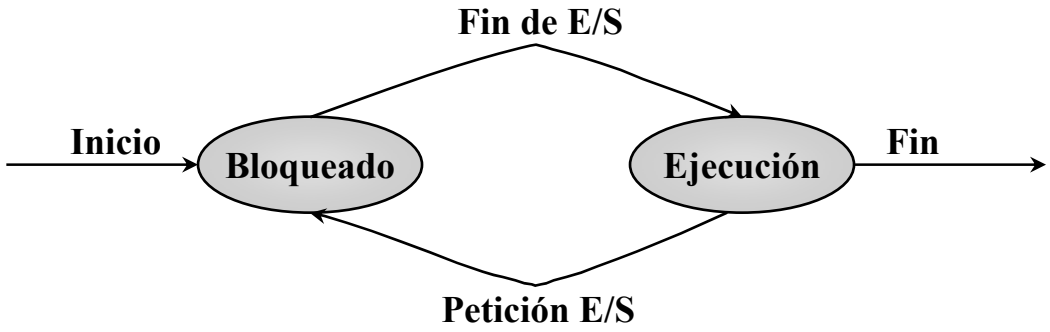
Procesos limitados por E/S: E/S largas y frecuentes

Procesos limitados por CPU: muy poca E/S



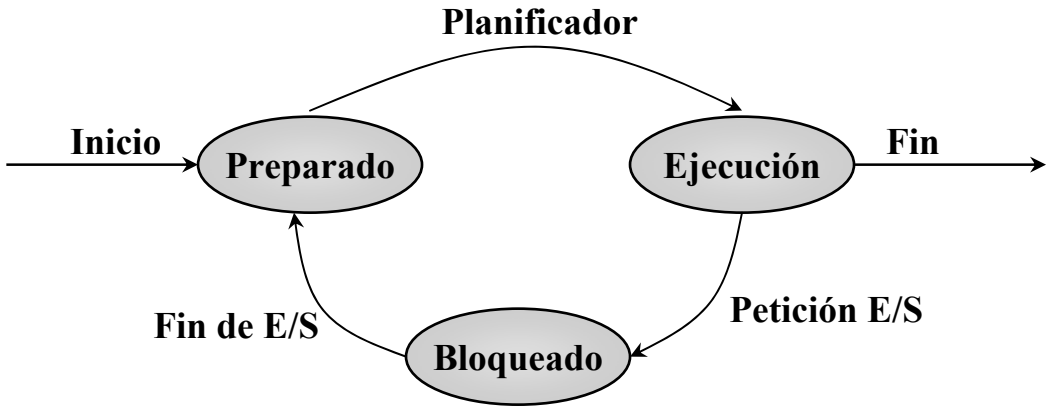
Cálculo **Entrada/Salida**

Estados de un proceso (1)



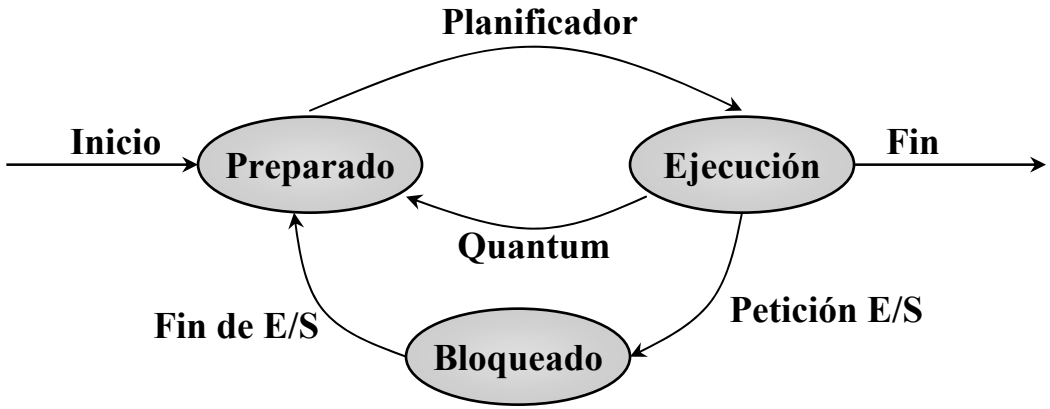
- Sistema Operativo Monoprogramado
  - E/S: pérdida de tiempo de CPU
- Sistema Operativo Multiprogramado
  - Varios procesos en el sistema
  - E/S bloqueante: cesión de la CPU a otro proceso

Estados de un proceso (2)



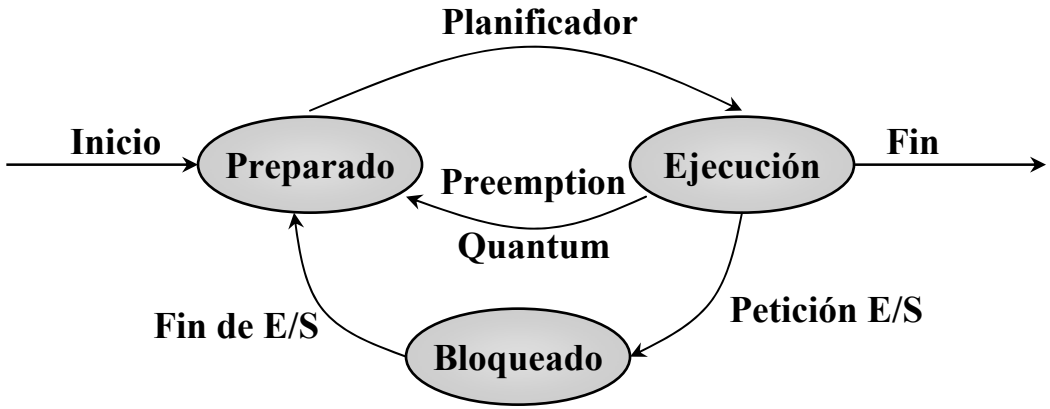
- Planificador (Scheduler)
  - Selecciona un proceso entre los *Preparados*
- El proceso en *Ejecución* deja CPU cuando pide E/S
  - Otros procesos pueden esperar indefinidamente
  - No apto para Sistemas *Interactivos* (solo para *Lotes*)

Estados de un proceso (3)



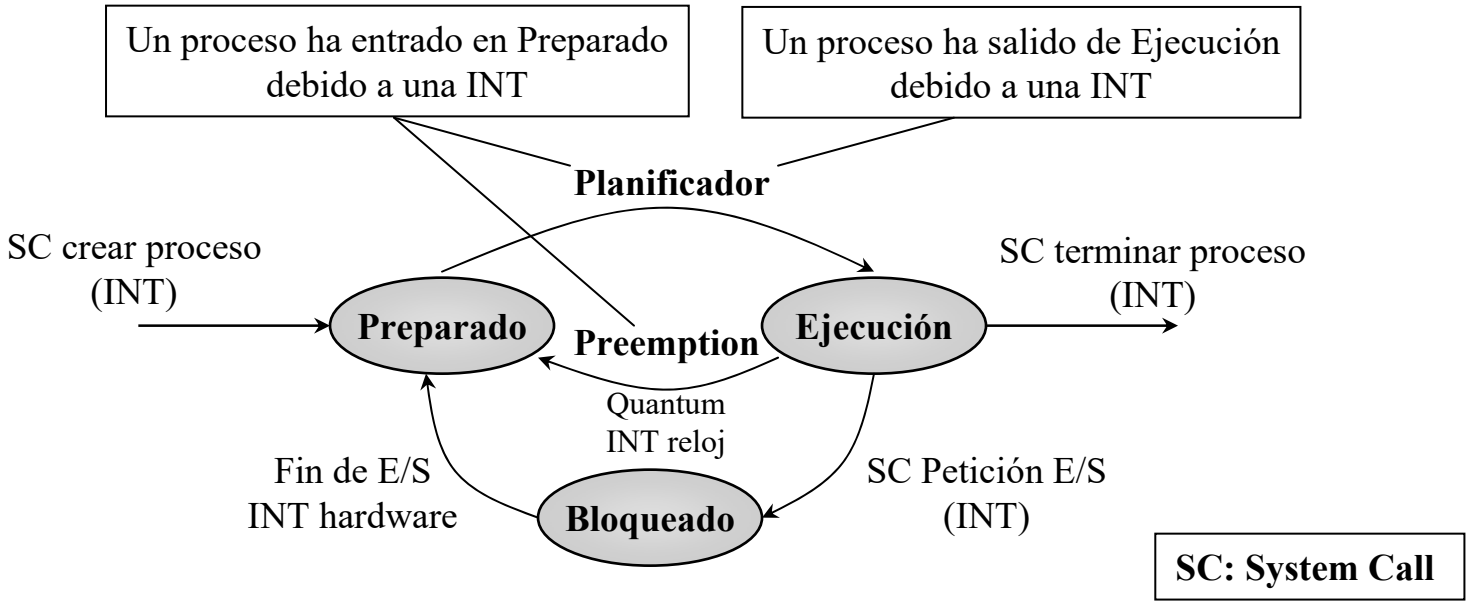
- Quantum
  - Tiempo máximo de permanencia en *Ejecución*
  - Controlado por la rutina de interrupción del reloj
  - El proceso en *Ejecución* deja CPU cuando pide E/S o cuando agota su tiempo de Quantum
  - Tiempo de espera de otros procesos limitado
  - Apto para Sistemas *Interactivos* (*Tiempo Compartido*)

Estados de un proceso (4)



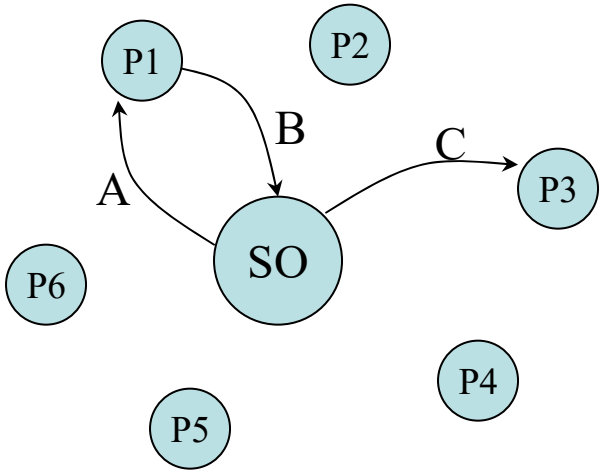
- Preemption
  - Si Prioridad(*Preparado*) > Prioridad(*Ejecución*)
    => expulsar proceso de *Ejecución*
- Otros autores entienden por Preemption:
  - Un proceso lógicamente ejecutable cede la CPU a otro
    - Por agotamiento de Quantum
    - Por mayor Prioridad

SO Como gestor de eventos



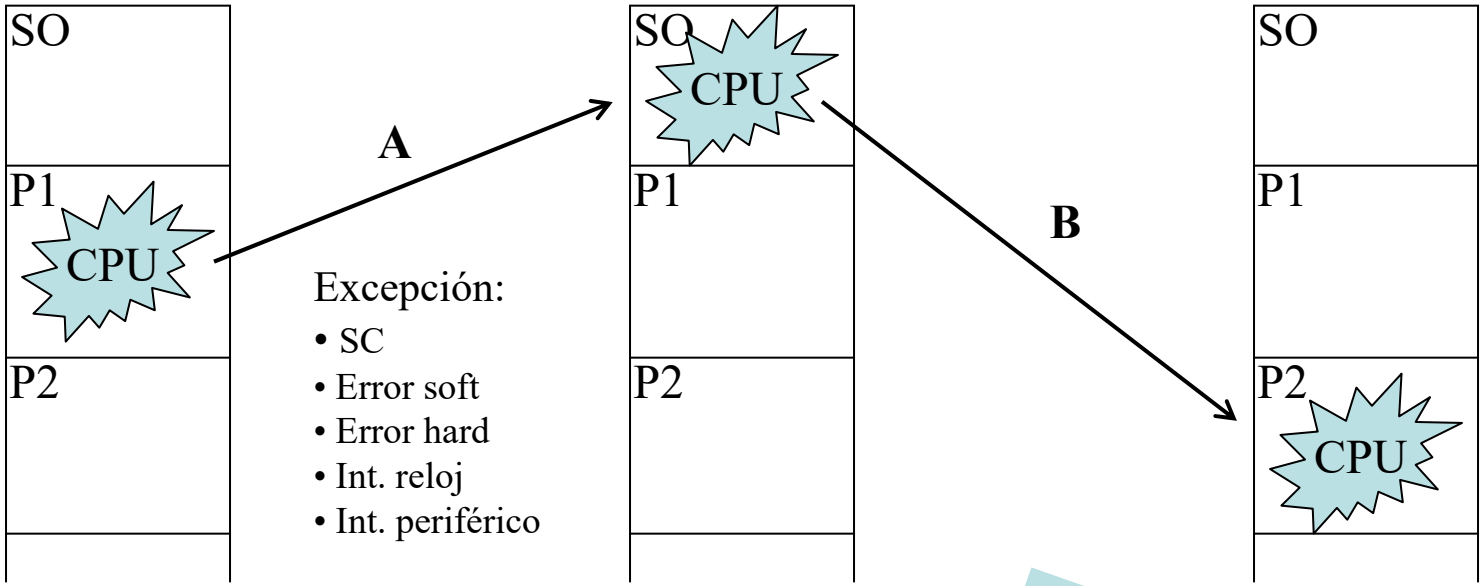
- Todo cambio de estado proviene de una Interrupción
  - El Sistema genera eventos (interrupciones)
  - En cada evento, el Sistema Operativo toma el control y gestiona los cambios de estado pertinentes

SO Como gestor del recurso *Procesador*



- A: El SO cede la CPU a un proceso
- B: El proceso cede la CPU al SO (SC, int. hardware, …)
  - Los procesos no pueden pasarse el control de CPU entre si
  - Devuelven siempre el control al SO (voluntaria o forzosamente)

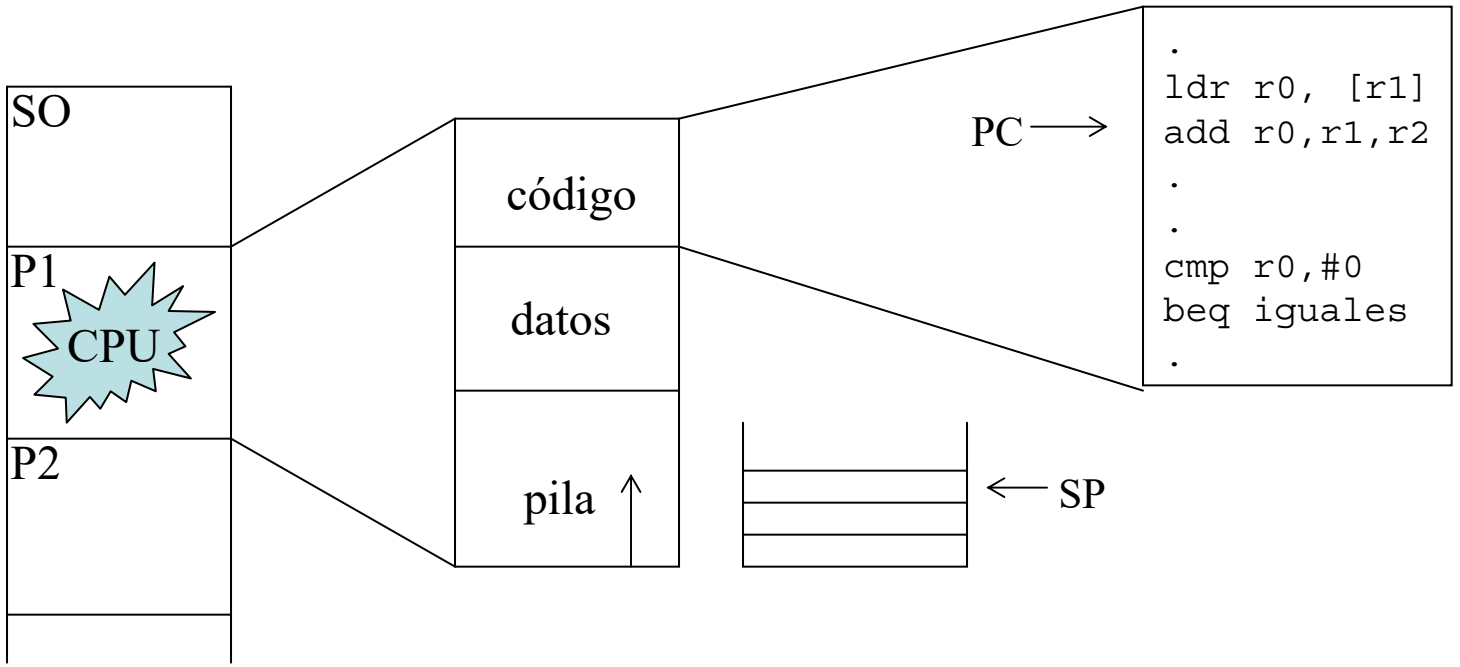
Dinámica del SO



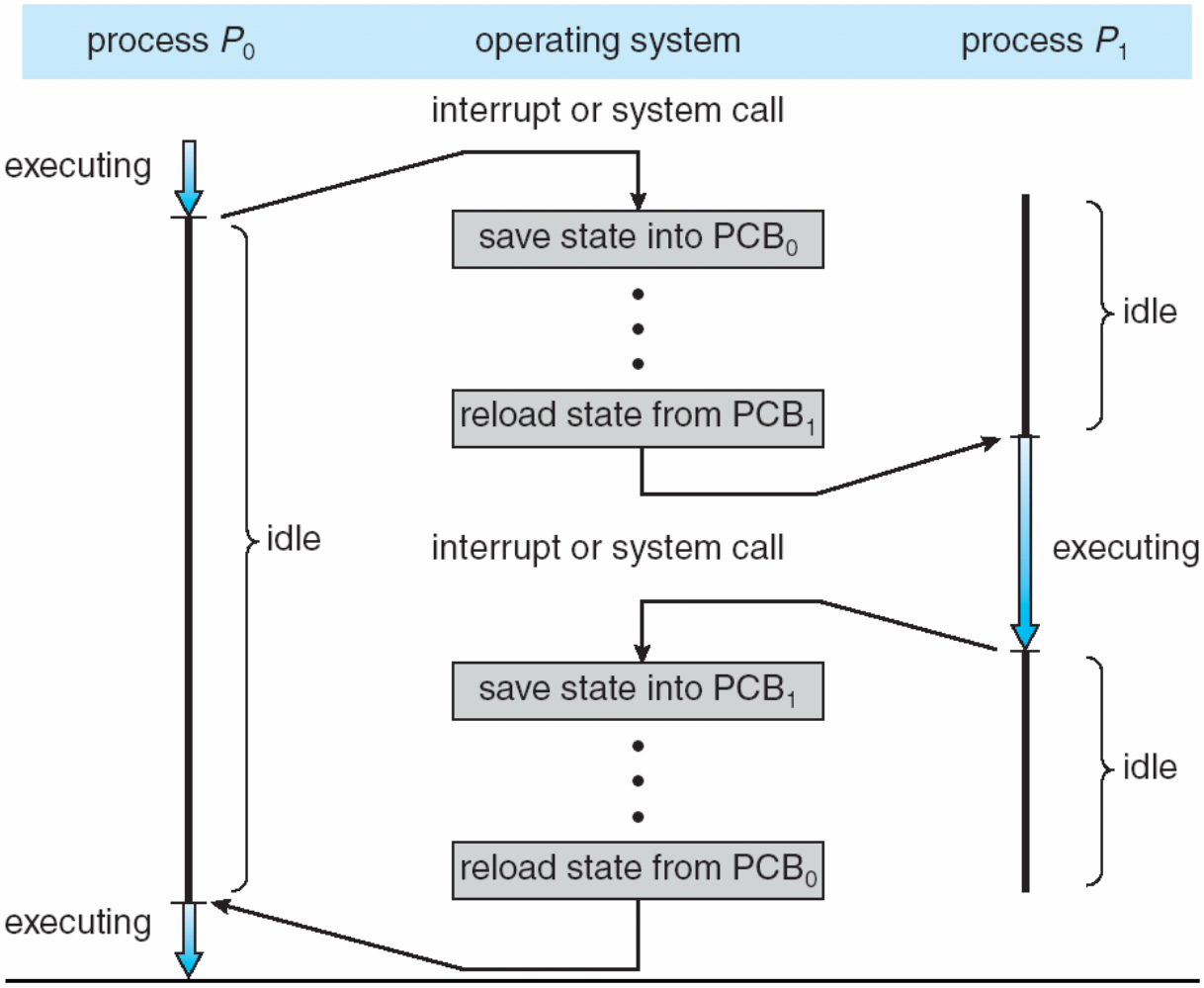
- **A**: Salvar contexto del proceso P1
- Realizar tarea específica para responder al evento
- Poner proceso P1 en el estado apropiado (*Preparado* o *Bloqueado*)
- Seleccionar proceso entre *Preparados* (P2): Planificador (Scheduler)
- **B**: Restaurar contexto del proceso P2 (Dispatcher)

Contexto de un Proceso

- Todo lo que debemos salvar cuando la CPU deja de ejecutar un proceso, para poder retomar su ejecución en otro momento



Cambio de contexto



Bloque de Control de Proceso (PCB)

- Process Control Block:
  contiene toda la información relativa a un proceso

  - Identificador
  - Estado
  - Prioridad
  - Padre e hijos
  - contexto I
    - Punteros a zonas de memoria
  - contexto II
    - pc, cprs, registros
  - Info E/S
    - Peticiones pendientes
    - Disp. E/S asignados
    - Ficheros abiertos
  - Info Contabilidad
    - Tiempos acumulados
    - Fechas
    - Cuotas (memoria, tiempo, …)

- Cada vez que un proceso cambia de estado, el SO anota los cambios en los campos apropiados de su PCB
- Tabla de Procesos: un struct PCB por proceso

Planificación para monoprocesadores

Planificación para monoprocesadores

- Tipos de Planificador
- Tipos de Sistema Operativo
- Criterios para el Planificador de corto
- Planificación por Prioridades
- Otras políticas de planificación
  - FCFS, RR, SPN, SRT, HRRN, Colas con Realimentación
- Planificación en UNIX

[Sta05]: capítulo 9

[SGG05]: capítulo 5

Tipos de Planificador

- Largo plazo
  - Decide sobre la creación de nuevos procesos, determina qué programas son admitidos por el sistema para ser procesados
  - Controla el grado de multiprogramación
  - Mayor número de procesos en el sistema implica menor porcentaje de tiempo dedicado a cada proceso
- Medio plazo
  - Swapping: intercambio de procesos entre memoria y disco
- Corto plazo
  - Se invoca cada vez que ocurre un evento
    - Interrupción de reloj, Interrupción de E/S, Llamada al sistema operativo, señales
  - Decide qué proceso de los preparados será el siguiente en ejecutarse

Planificador y estados de un proceso

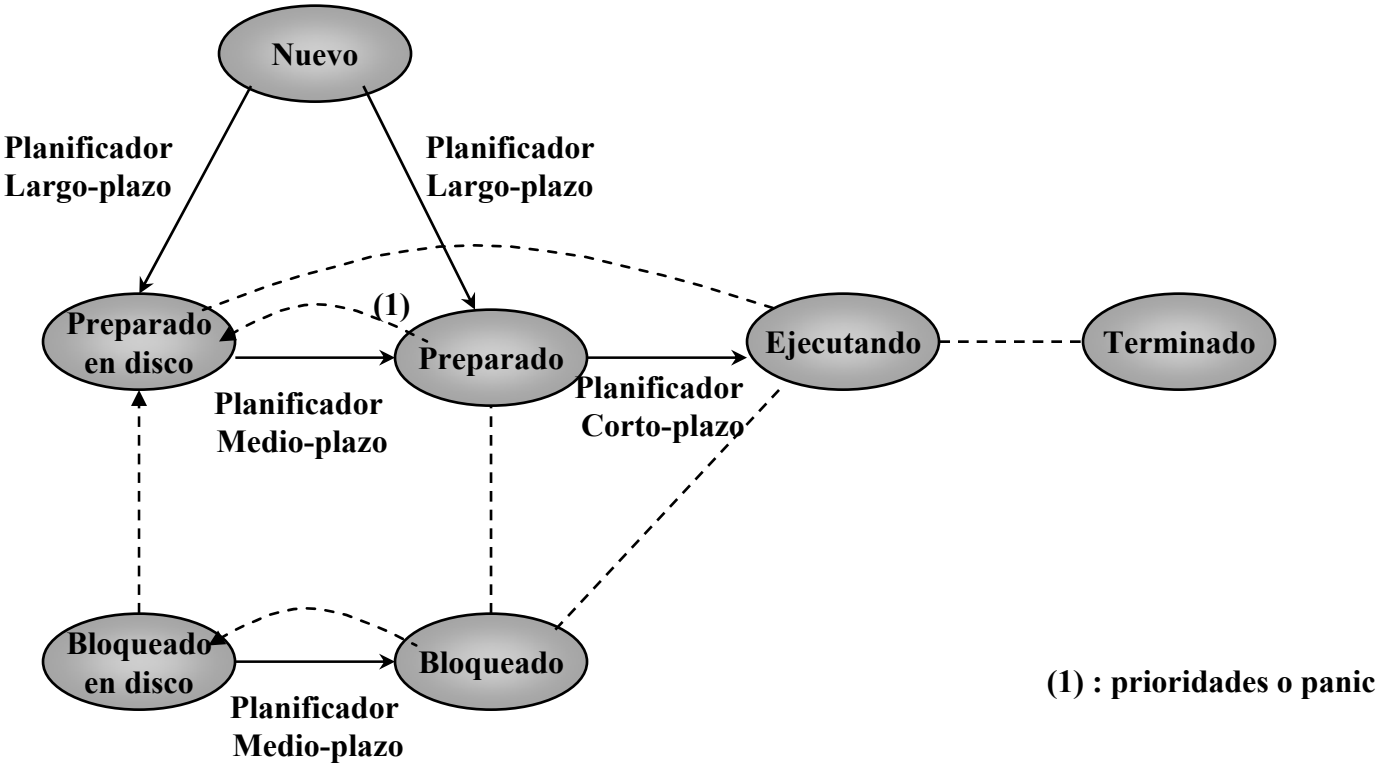
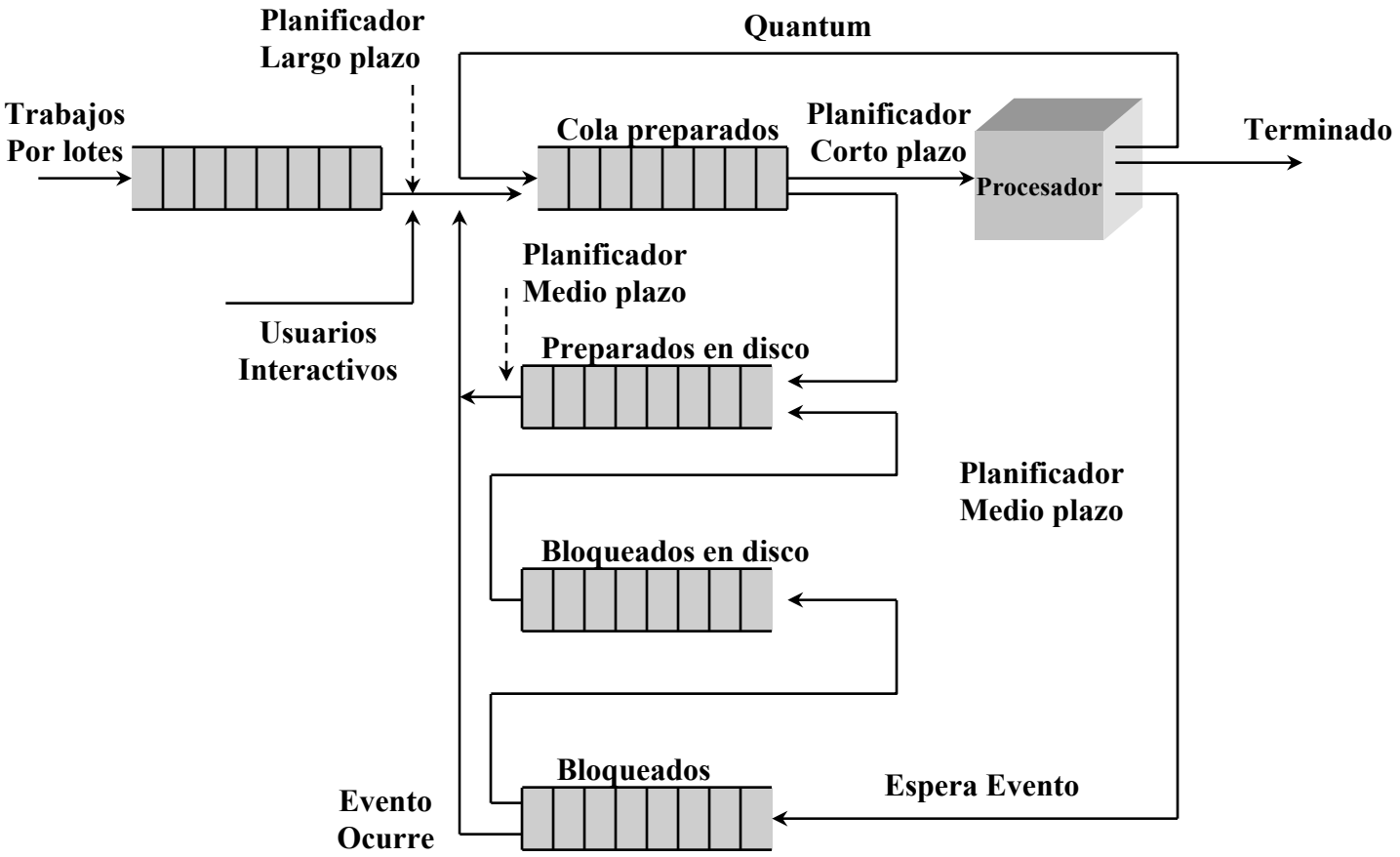


Diagrama de colas para la planificación



Tipos de Sistema Operativo

- SO por Lotes
  - Se agrupan programas,datos y órdenes de control para formar “trabajos” a ejecutar por el ordenador
    - No permiten interacción hombre - máquina
    - Buscan la máxima eficiencia del sistema
- SO de Tiempo Compartido
  - Pensados para entornos muy interactivos
  - El sistema pretende dar la imagen a cada usuario de que tiene toda la máquina para él
- SO de Tiempo Real
  - Entornos en los que han de ser aceptados y procesados un gran número de sucesos externos con mayor o menor urgencia
    - Control industrial, comunicaciones, control de vuelo, …
- SO de Plazo Fijo
  - Ciertos trabajos tienen un tiempo específico para ser terminados. Se han de terminar antes de ese momento

Criterios para el Planificador de corto plazo

- Orientados al usuario
  - Tiempo de respuesta
    - Tiempo desde que se emite una petición hasta que se recibe la primera respuesta del sistema
  - Tiempo de retorno
    - Tiempo desde que se emite una petición hasta que se completa su servicio
  - Previsibilidad
    - El tiempo en que se ejecuta un trabajo debe ser independiente de la carga del sistema

Criterios para el Planificador de corto plazo

- Orientados al sistema
  - Productividad (throughput)
    - Maximizar el número de trabajos terminados por unidad de tiempo
  - Utilización del procesador
    - Maximizar el porcentaje de tiempo que el procesador está ocupado
  - Recursos equilibrados
    - Mantener ocupados todos los recursos del sistema
    - Favorecer los procesos que no usen recursos sobrecargados

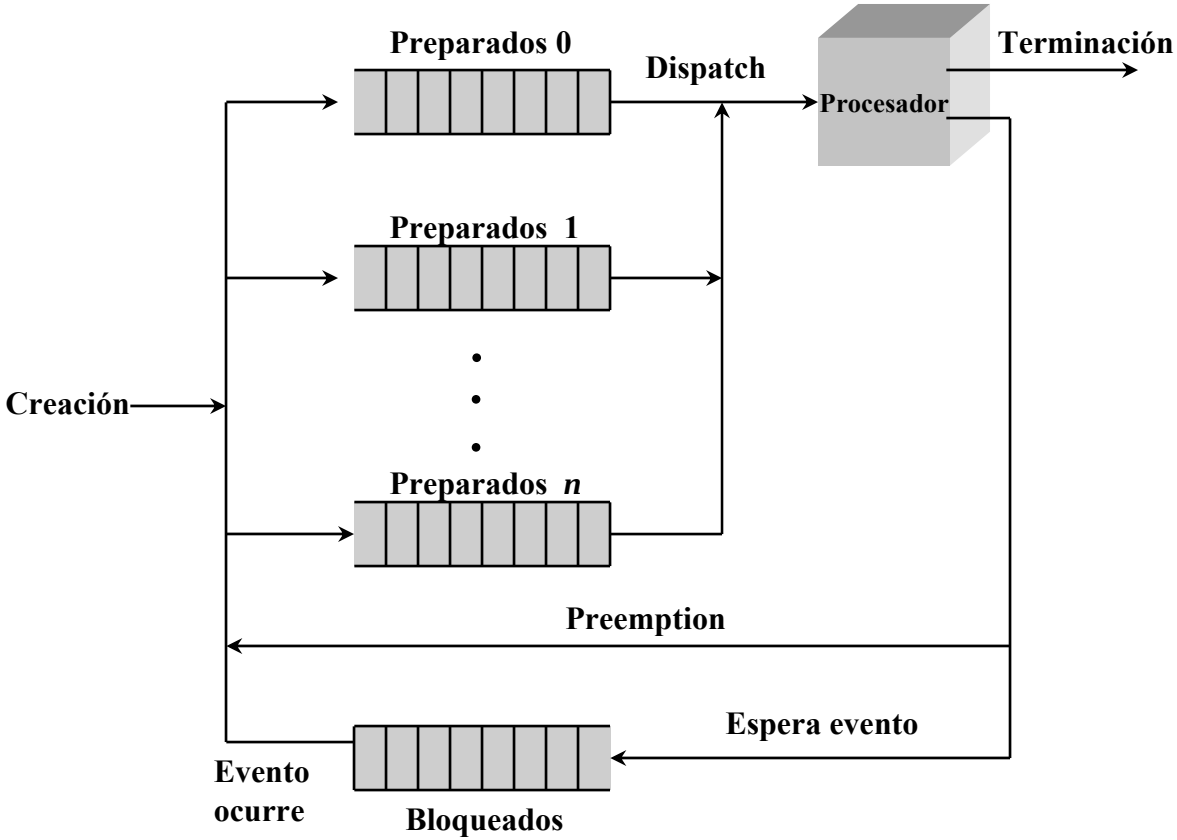
Criterios para el Planificador de corto plazo

- Otros criterios
  - Plazos
    - Cada tarea tiene asociado un plazo de terminación
  - Prioridades
    - Cada tarea tiene asignada una prioridad
  - Equidad
    - En ausencia de otras directrices, todos los procesos deben tratarse de forma equitativa

Planificación por Prioridades

- Cada proceso tiene asignada una prioridad
- El Planificador elige siempre el proceso con mayor prioridad entre los preparados
  - Implementación típica: una cola de preparados para cada prioridad
- Los procesos de baja prioridad pueden sufrir inanición
  - A veces los procesos cambian su prioridad en función de su historia en el sistema

Colas de prioridad



Asignación de prioridades

- Interna / Externa
  - Definida a partir de parámetros obtenidos por el propio SO
  - Definida por el administrador en base a tipo de usuario, importancia del trabajo, cuota que se paga, …
- Estática / Dinámica
  - Estable durante toda la vida del proceso
  - Varía durante la ejecución en función del patrón de uso de recursos, de la ocupación del sistema, …
- Se gana o se compra
  - El proceso gana o pierde prioridad dependiendo de su comportamiento
  - El proceso adquiere un cierto nivel de prioridad pagando por ello al propietario de la máquina

Modo de Decisión

- No preemption
  - Cuando un proceso consigue el procesador, no lo abandona hasta que termina o se bloquea en espera de una operación de E/S
- Preemption
  - Un proceso en estado de ejecución puede ser interrumpido por el Sistema Operativo, pasando a estado preparado
  - Evita que un proceso pueda monopolizar el procesador durante demasiado tiempo

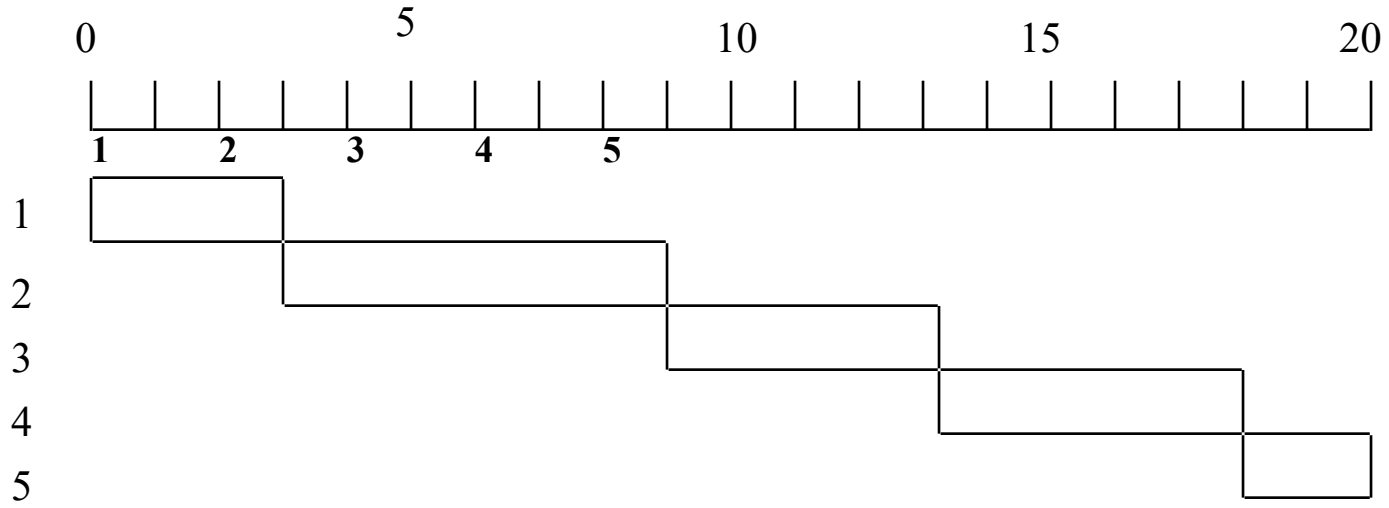
Otras políticas de planificación

- First Come First Served (FCFS)
- Round-Robin (RR)
- Shortest Process Next (SPN)
- Shortest Remaining Time (SRT)
- Highest Response Ratio Next (HRRN)
- Colas con Realimentación

Ejemplo

Proceso	Tiempo de llegada	Tiempo de servicio
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

First-Come-First-Served (FCFS)

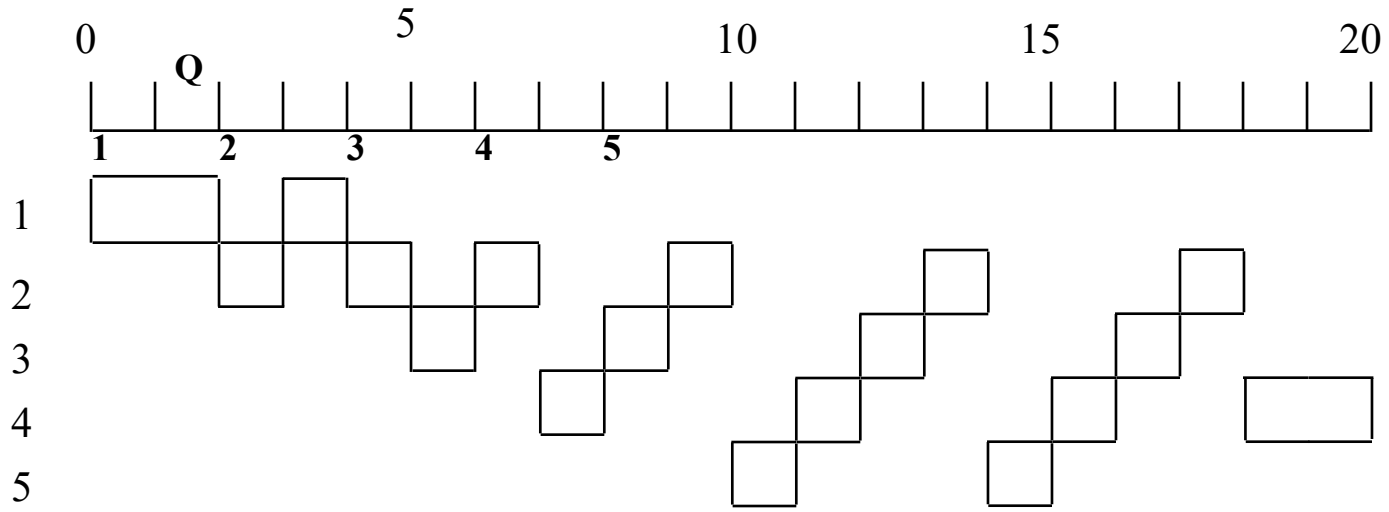


- Cuando un proceso abandona la CPU, el proceso más viejo de la cola de Preparados pasa a ejecutarse

First-Come-First-Served (FCFS)

- Un proceso corto puede ser obligado a esperar durante mucho tiempo antes de entrar a ejecutarse
- Favorece a los procesos CPU-bound
  - Los procesos con mucha E/S deben esperar hasta que los CPU-bound abandonan el procesador
- Minimiza el número de cambios de contexto

Round-Robin



- Usa preemption basada en un reloj
- Quantum (Q): tiempo máximo de permanencia en ejecución que se da a un proceso

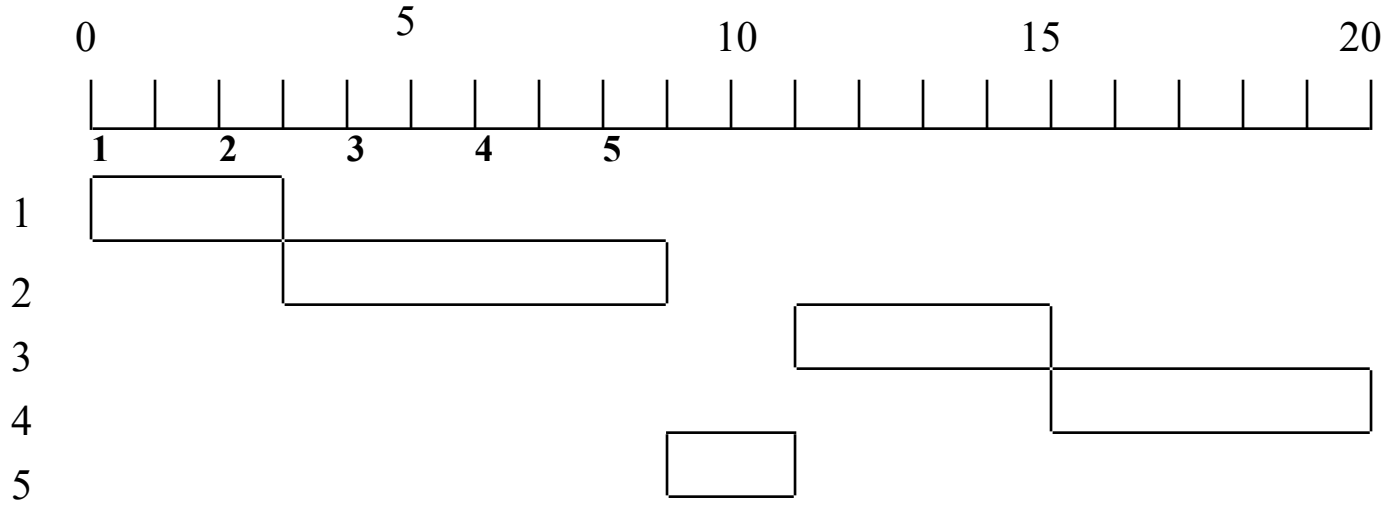
Round Robin: tamaño del Quantum

- Mayor quantum, mejor throughput (pensar por qué)
  - Caso extremo: quantum infinito. Óptimo en supercomputación...
- Menor quantum, mejor tiempo de respuesta (pensar por qué)
- En general, más cambios de contexto = menor rendimiento = mayor satisfacción de los usuarios en tiempo compartido
- Porcentaje de tiempo perdido en Cambio de Contexto

$$\%Tcc = 100 * \frac{Tcc}{Tcc + Q} \qquad \textbf{(Mínimo)}$$

  - Tcc disminuye cuando aumentan prestaciones
  - Si Q constante => %Tcc disminuye, tiende a cero
  - Se puede disminuir Q para mejorar el servicio sin aumentar la pérdida de prestaciones

Shortest Process Next

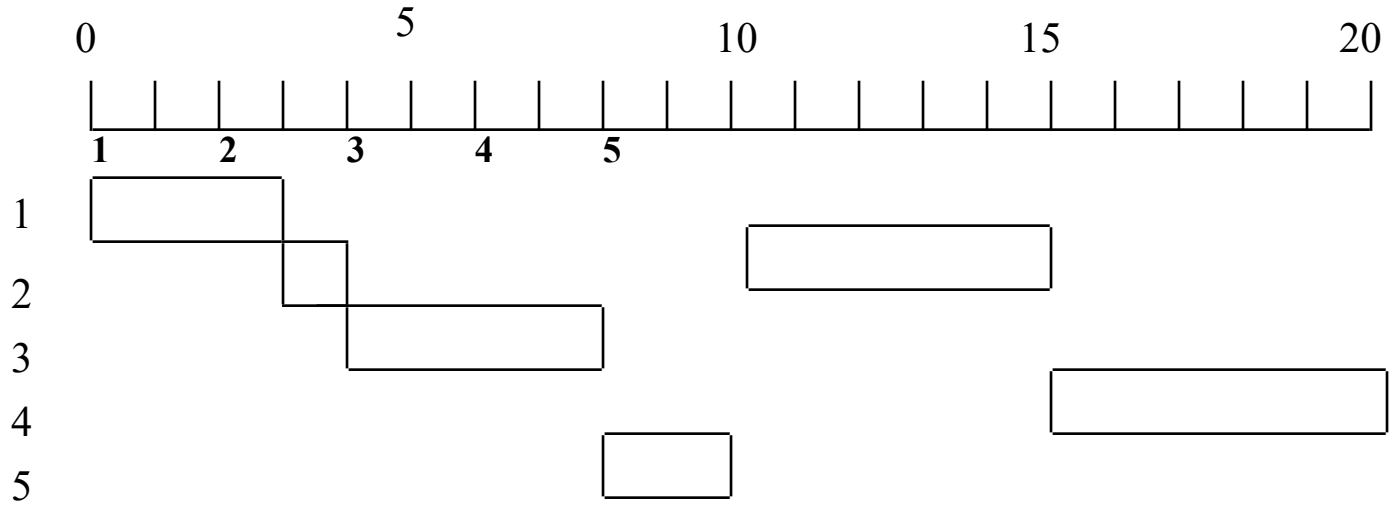


- Política no preemption
- Se selecciona al proceso con menor tiempo en ejecución previsto
- Los procesos cortos se adelantan a los largos

Shortest Process Next

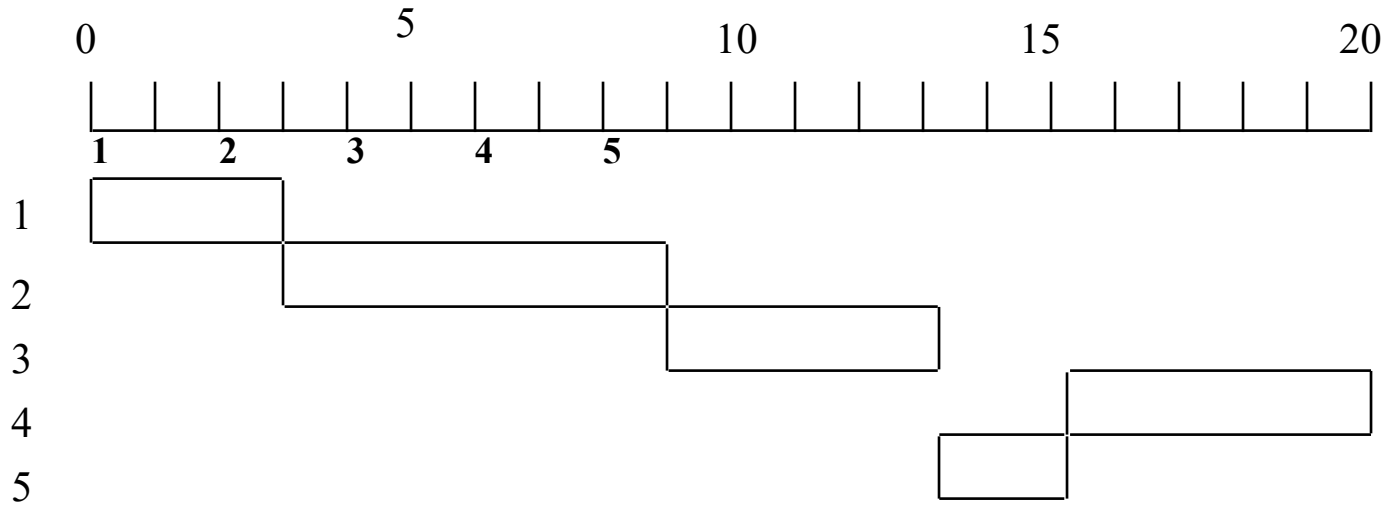
- Minimiza el tiempo medio de espera (óptimo)
- El comportamiento de los procesos largos es menos previsible
- Los procesos largos pueden sufrir inanición
- Debe estimar el tiempo de proceso
- Si el tiempo estimado para un proceso no es el correcto, el SO puede abortarlo

Shortest Remaining Time



- Versión preemptive de la política shortest process next
- Debe estimar el tiempo de proceso
- Posible inanición de procesos largos

Highest Response Ratio Next (HRRN)

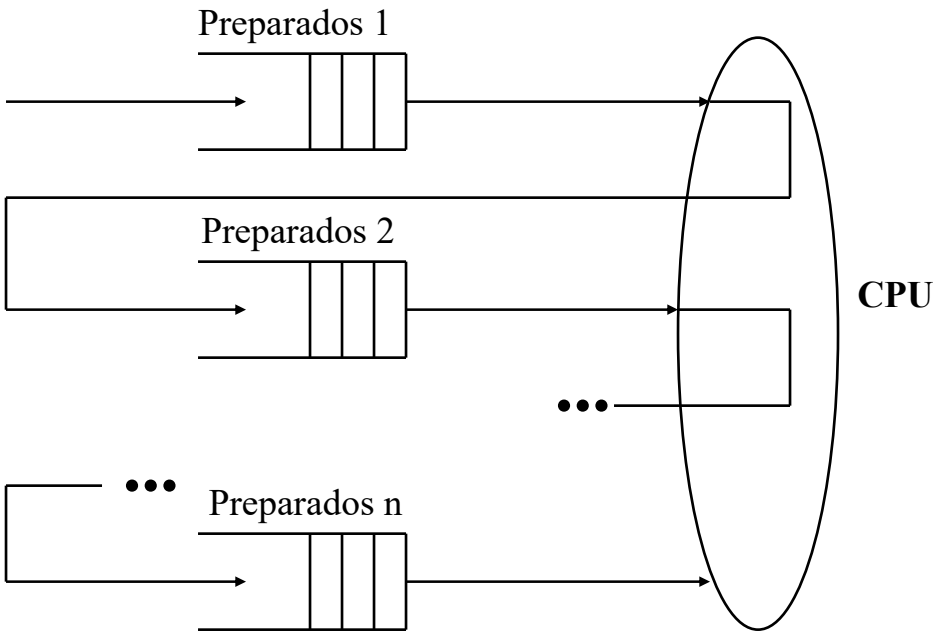


- Elige el siguiente proceso según mayor ratio

$$\frac{\text{Tiempo de espera en las colas + tiempo de servicio}}{\text{tiempo de servicio}}$$

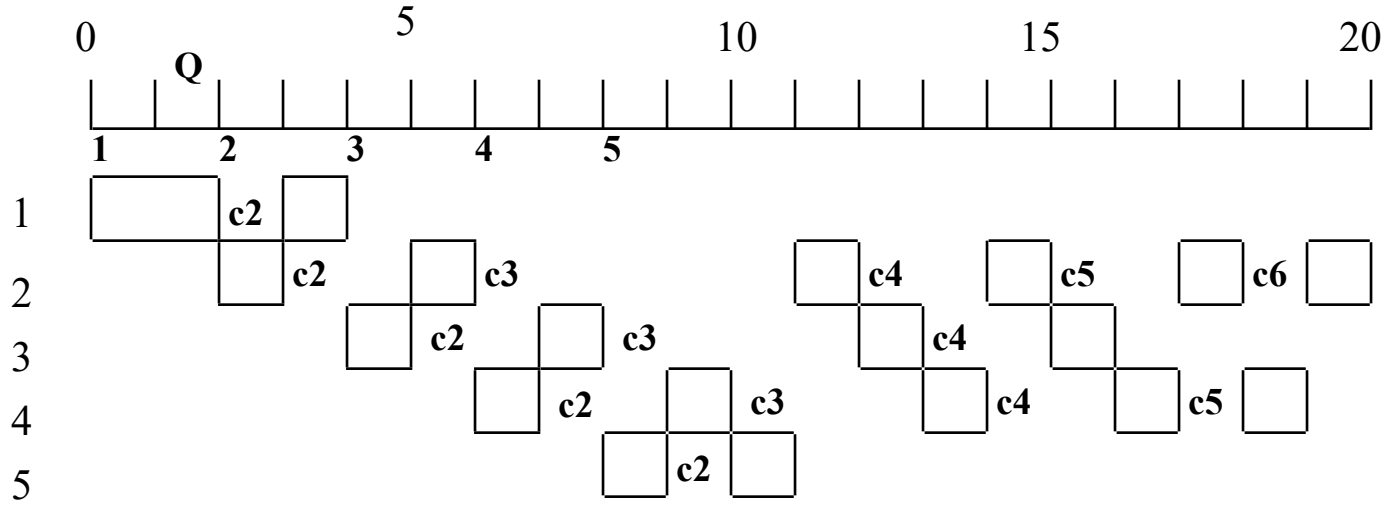
- No preentive
- Resuelve problemas inanición

Colas con realimentación



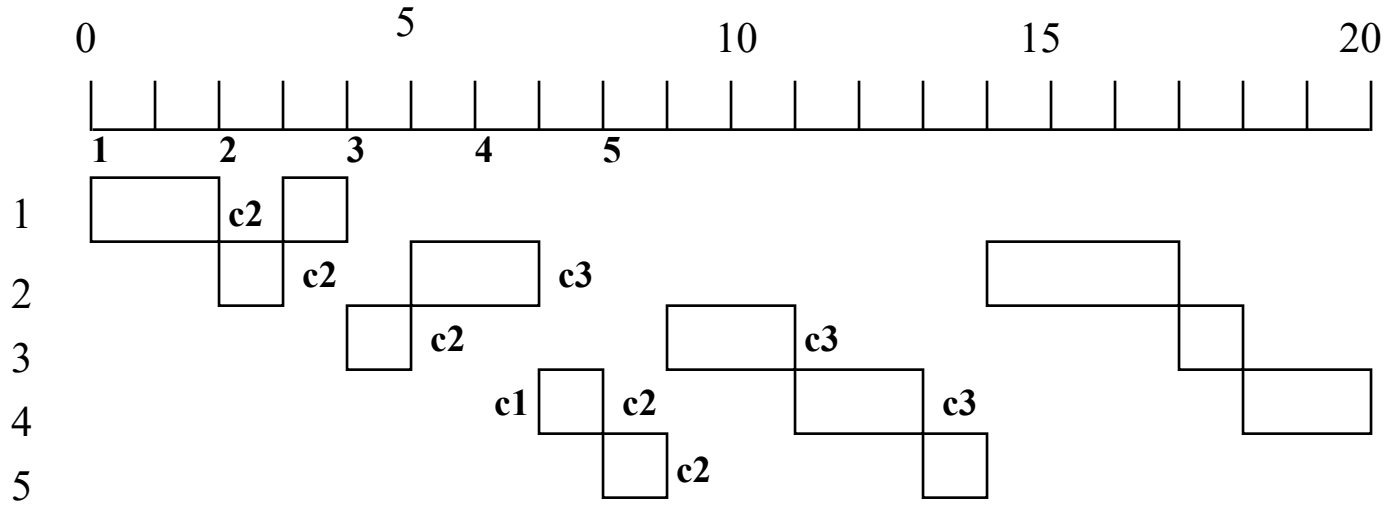
- Varias colas con distinta prioridad
  - Los procesos pasan de unas a otras en función de su comportamiento temporal

Colas con realimentación



- Penaliza a los procesos que más tiempo llevan en ejecución

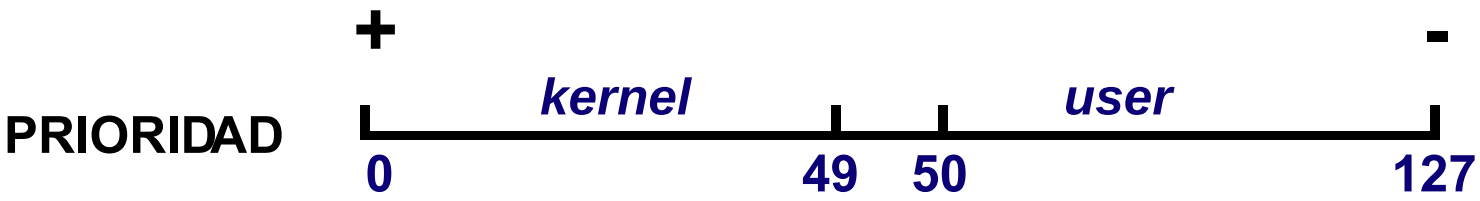
Colas con realimentación



- Quantum variable según cola
  - Q(c1)=1, Q(c2)=2, …

Planificación en UNIX

- Prioridad variable + Round Robin
- Cálculo de prioridad
  - Valor numérico: A mayor valor menor prioridad



  - Prioridad base + NICE + prioridad variable
  - *Prioridad base* de usuario: 50
  - *NICE*: valor positivo que el usuario puede añadir a sus programas para “molestar menos”
  - *Prioridad variable*: función de la carga del sistema y del uso de CPU que el proceso ha hecho recientemente

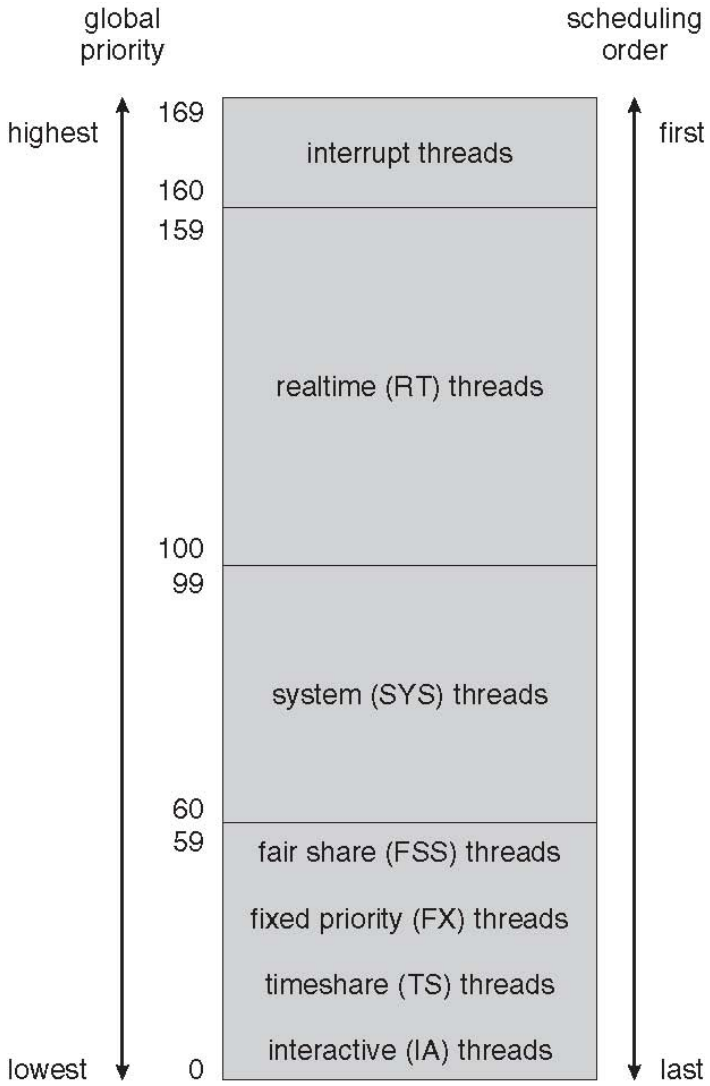
Planificación en Solaris (hendrix)

- Planificación basada en prioridades
- Seis clases de procesos (prioridad de + a -)
  - Tiempo real
  - Sistema
  - Parte justa (fair share)
  - Prioridad fija
  - Tiempo compartido (por defecto)
  - Interactivo
- Cada proceso está en una única clase en un momento dado
- Cada clase tiene su propio algoritmo de planificación
- Tiempo compartido es una cola multinivel con realimentacion (configurable por el administrador)

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Planificación en Solaris



Planificación en Solaris (Cont.)

- El planificador convierte las prioridades particulares de cada clase en una prioridad global por proceso
  - El proceso con más prioridad es el siguiente en ejecutarse
  - Se ejecuta hasta (1) bloqueo, (2) quantum, (3) preempted por un proceso más prioritario
  - Si hay varios procesos de la misma prioridad, se selecciona por RR

Sistemas Operativos

Procesos: Llamadas al Sistema

Procesos: Llamadas al Sistema

- Recuerda: main, imagen de un proceso
- Llamadas fork() y exec()
- Terminación de procesos: llamadas exit() y wait()
- Resumen de identificadores y marcas
- Intérpretes de comandos

[Ste05]: capítulo 8

Estructura de Programa en C

```
main(int argc, char *argv[], char *envp[])
```

- Lista de argumentos: argv
- Lista de variables de entorno: envp

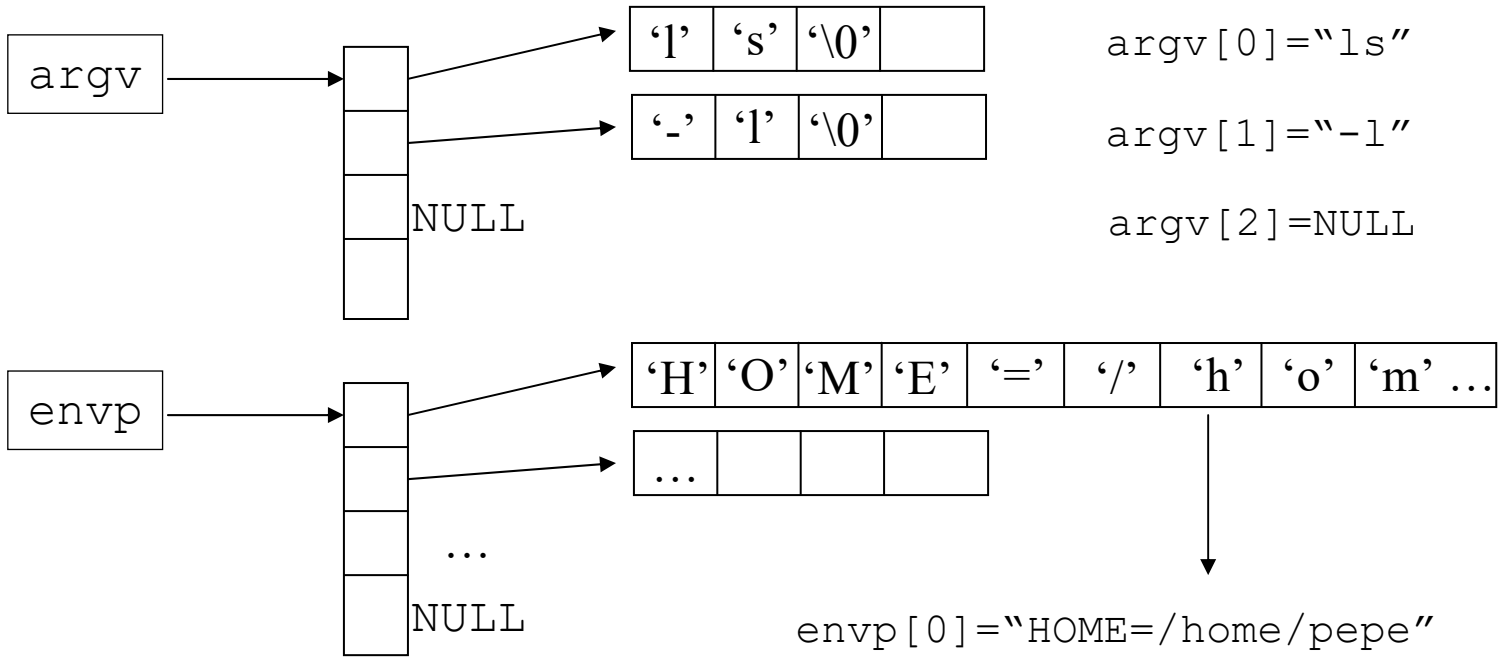
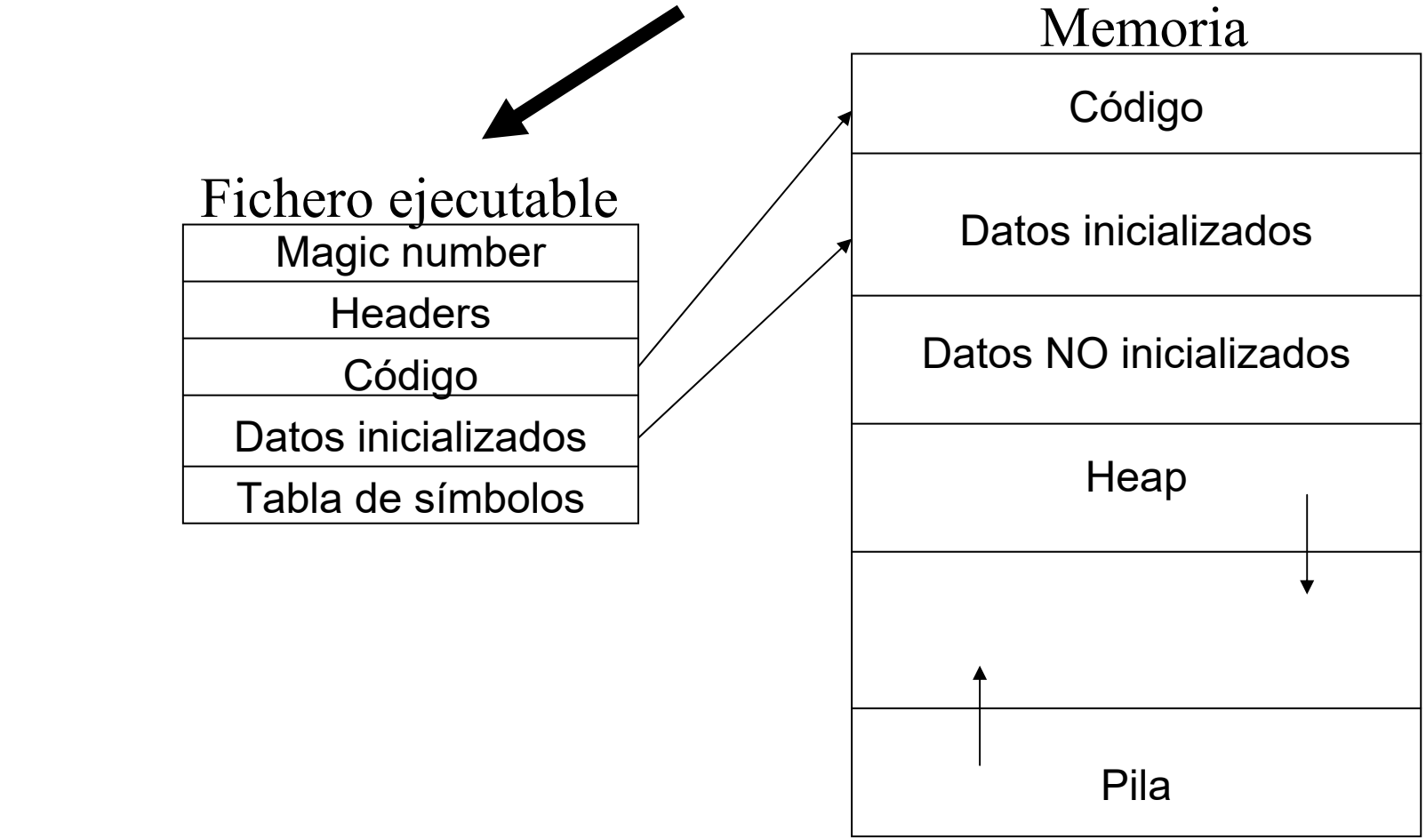


Imagen de un proceso

```
cc fichero.c -o fichero
```



UNIX: Llamadas asociadas

- Cada proceso en Unix se identifica por un pid (process identificator) número natural ≥ 0

  Algunos PIDs: 0 -> scheduler, 1-> init

- Gestión de procesos
  - `fork() -> crear nuevo proceso (padre crea hijo)`
  - `exit() -> terminar (voluntariamente) proceso`
  - `wait() -> esperar terminación proceso (hijo)`
- Carga y ejecución
  - `exec() -> ejecutar programa`
- Información
  - `getpid() -> obtener PID de proceso (pid)`
  - `getppid() -> obtener PID del proceso padre (ppid)`

fork()

- Crea un nuevo proceso
- La llamada `fork()` crea un duplicado (hijo) del proceso que la realiza (padre)
- Los dos procesos continúan ejecutando a partir de la llamada a la función `fork()`
- `fork()` devuelve el PID del hijo al proceso padre
- `fork()` devuelve 0 al proceso hijo
- `fork()` devuelve -1 si falla. Razones de fallo
  - Demasiados procesos en el sistema (causa exterior)
  - Demasiados procesos de usuario (CHILD_MAX, <limits.h>, 25 en hendrix)

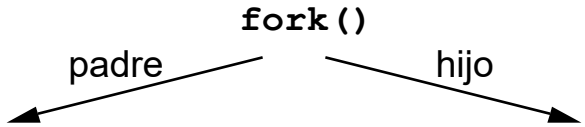
```
#include <unistd.h>

pid_t fork(void);    /* pid_t es un int
                        /usr/include/sys/types.h */
```

Ejemplo **fork()**

```
main(){
    int id;
    printf("Comienza la ejecución\n");
    id=fork();
    if (id==0) printf("Soy el hijo\n");
    else printf("Soy el padre\n");
    /* ejecutado por ambos */
    printf("Termina la ejecución\n");
    exit(0);
}
```

```
printf("Comienza la ejecucion\n");
```




```
id=pid del hijo
if (id==0) printf("soy el hijo\n");
else printf("soy el padre\n");
printf("Termina la ejecucion\n");
exit(0);
```

```
id=0
if (id==0) printf("soy el hijo\n");
else  printf("soy el padre\n");
printf("Termina la ejecucion\n");
exit(0);
```

exec()

- Pone en ejecución un programa (en fichero ejecutable)

```
main(){
   execl("/bin/ls","ls","-l",0);
   printf("Error\n");
}
```

- La llamada `exec()` sustituye la imagen del proceso que la realiza por la almacenada en un fichero ejecutable
  - En el ejemplo anterior, el `printf()` sólo se ejecuta si falla `exec()`
- La nueva imagen empieza a ejecutarse por la función `main()`
- La identidad del proceso no cambia. Sigue teniendo el mismo identificador, el mismo tiempo de CPU consumido, …
- Varios formatos (ver libro Stevens 2005):
  - `execl()`, **`execlp()`**, `execle()`
  - `execv()`, **`execvp()`**, **`execve`**`()`

exec()

- l -> argumentos del comando en lista
- v -> argumentos del comando en vector (como argv[])
- e -> nuevas variables de entorno en vector (como envp[])
- no e -> variables de entorno del usuario
- p -> filename (fichero ejecutable (usa variable PATH))
- no p -> pathname (trayectoria completa (no usa PATH))

```
#include <unistd.h>
int execl(const char *pathname,const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0, ...
                        /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp []);
int execlp(const char *filename, const char *arg0,... /*(char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

Devuelven: –1 si hay error, nada si hay éxito (lógico)

wait()

#include <sys/wait.h>

pid_t wait(int *p_estado)

- Bloquea un proceso hasta que termine un hijo suyo (uno cualquiera si hay varios).
- El resultado de ejecutar wait() puede ser:
  - Bloqueo del proceso padre si todos sus hijos siguen ejecutandose
  - Retorna inmediatamente con el estado de terminación de un hijo (si esa información esta disponible)
  - Retorna inmediatamente -1 (error) si el proceso padre no tiene hijos
- waitpid(pid_hijo,*p_estado,options) -> Igual que wait, pero esperando a un hijo particular
  - Permite opciones adicionales para controlar bloqueos

Terminación de procesos

Voluntaria: `exit(0..255)`	Involuntaria: llega señal

- En ambos casos, el padre puede recibir información sobre la causa de la muerte del hijo mediante `wait()`

```
void exit(estado)
int estado
```

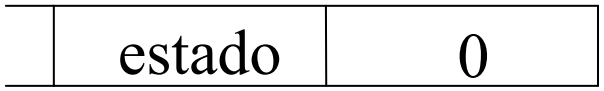
- No devuelve control nunca
- Estado: 0..255 (0: terminación normal)

```
int wait(p_estado)
int *p_estado
```

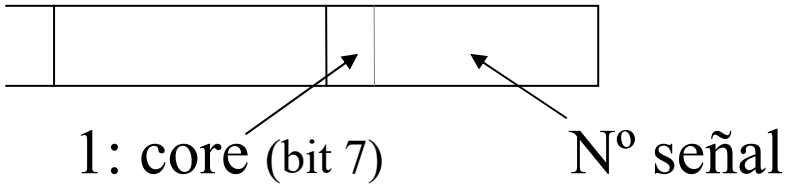
- Devuelve PID del hijo terminado
- Devuelve -1 en caso de ERROR

`*p_estado:`

Terminación voluntaria del hijo



Terminación involuntaria del hijo

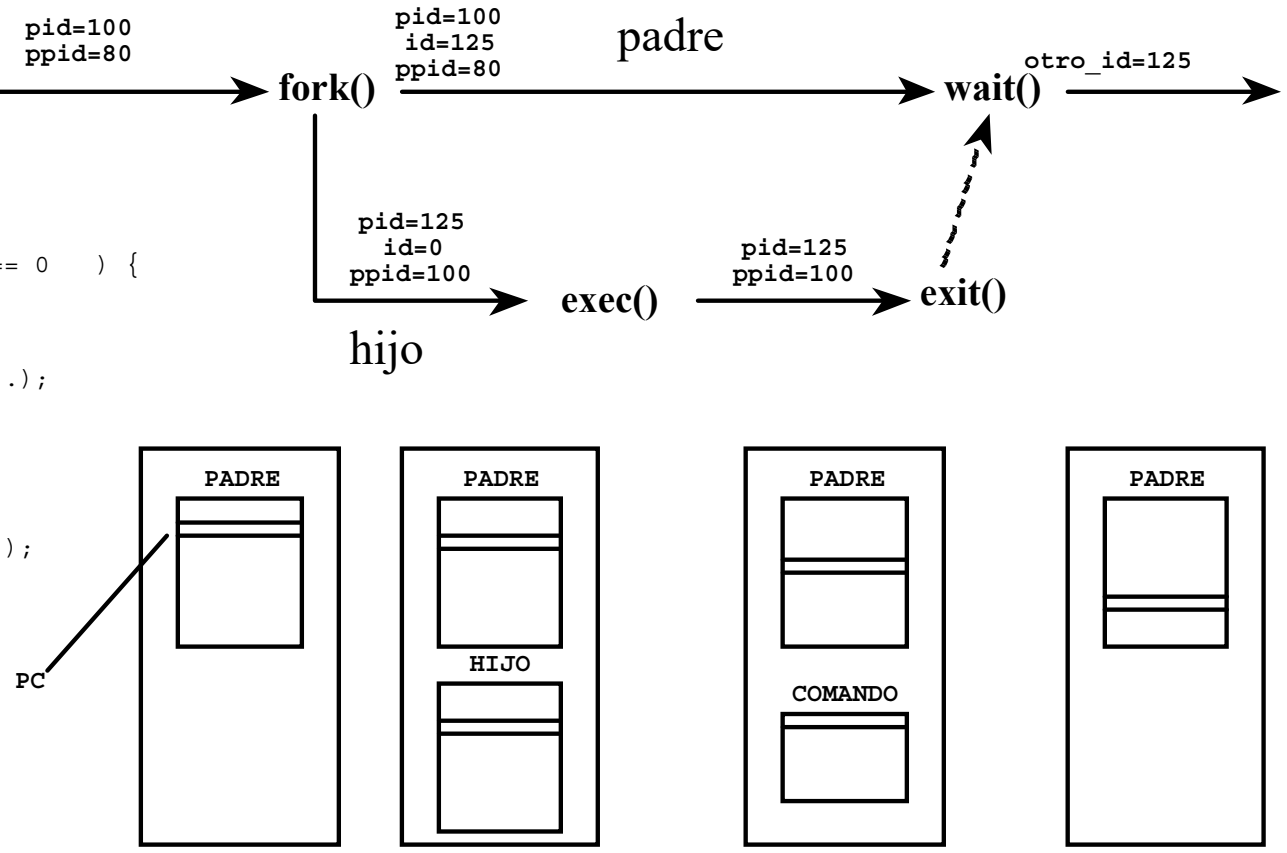


Información de hijo a padre

- Caso más frecuente
  - Proceso hijo termina y padre está esperando en wait()
    - Padre recibe información sobre la causa de la muerte del hijo y continúa ejecutando
    - Hijo desaparece
- Casos especiales
  - Proceso termina y padre no está en wait()
    - Proceso Zombie hasta que padre ejecute wait() o termine
  - Padre termina dejando hijos activos
    - Hijos adoptados por init (pid=1)
  - Padre ejecuta wait() y no existen hijos
    - Wait devuelve -1 y errno=10 (ECHILD: “No children”)

`fork()`, `exec()`, `wait()` : Ejemplo

```
main(){
    int id, estado, otro_id;
    if (   ( id = fork() ) == 0   ) {
        ...
        exec( "comando", ...);
    }
    ...
    otro_id = wait( &estado );
    ...
}
```



Ejemplo: **ej701.c**

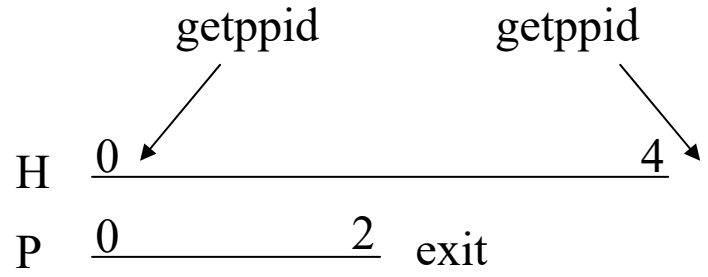
```
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "error.h"

void main() {
  int pidh,err;
  printf( "Inicio prueba\n" );
  pidh=fork();
  if ( pidh == 0 ) {           /* hijo */
    printf("\n\tSoy el hijo: %d\n", getpid() );
    printf("\n\tFork me devuelve: %d\n", pidh );
    exit( 0 );
  }
  /* padre */
  fprintf( stderr, "Antes de sleep\n");
  sleep( 1 );
  err=wait(NULL);
  if ( err == -1 ) syserr("wait");
  printf("\nSoy el padre: %d\n", getpid() );
  printf("\nFork me devuelve: %d\n", pidh );
}
```

Ejemplo de hijo huérfano (**ej81.c**)

```
#include <errno.h>

int main() {
    int pid;
    switch(fork()) {
    case -1: perror("fork"); exit(1);  /* error */
    case 0 :    /* hijo */
       pid=getppid();
       printf("pid padre antes = %d\n",pid);
       sleep(4); /* damos tiempo a que muera el padre.*/
       pid=getppid();
       printf("pid padre despues = %d\n",pid);
       exit(15);
    default:
       sleep(2);
       exit(0);
    }
}
```



Ejemplo de hijo zombie (**ej82.c**)

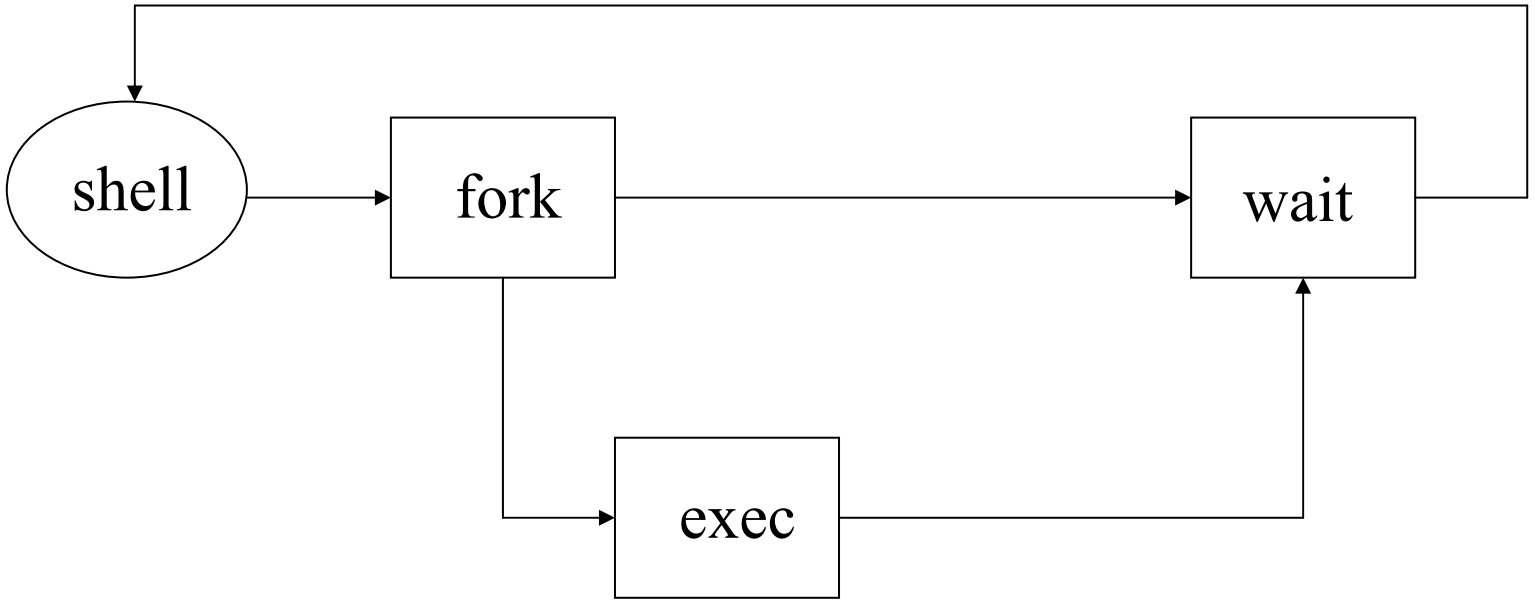
```
#include <errno.h>

int main() {
    int pid;
    int estado;
    switch(fork()) {
    case -1: perror("fork"); exit(1); /* error */
    case 0 :    /* hijo */
        sleep(2);
        printf("Soy el hijo %d y me muero...\n",getpid());
        exit(15);
    default: /*padre*/
        sleep(10); /* para que el hijo termine y quede zombie */
        if(-1==wait(&estado)){perror("wait"); exit(1);}
        printf("Estado hijo = %x\n", estado);
        exit(0);
    }
}
```

Arranque de UNIX

- ROM
- Carga del bootstrap
- Carga del resto del sistema
- Ejecuta kernel, gestor memoria, gestor ficheros
- exec(init) PID=1
- init ejecuta cada línea de /etc/inittab (fork y exec)
- Algunas líneas ejecutan /etc/getty (una por terminal)
- getty escribe “login:” y espera
  - Ejecuta /bin/login pasándole el username leído (exec)
- /bin/login escribe “password:” y espera
  - Comprueba password en /etc/passwd
  - Ejecuta el shell indicado en /etc/passwd (exec)
  - Shell es hijo de init, cuando acabe shell init lanzara otro getty

Estructura de shell



Algoritmo general de fork()

- Reservar PCB para el nuevo proceso
  - Comprobar que el número de procesos del usuario no excede el máximo
  - Buscar entrada libre en la tabla de procesos y un PID único
  - Marcar el estado del hijo como siendo creado
- Crear contexto del nuevo proceso
  - Copiar datos del padre al PCB del hijo
  - Inicializar los campos que difieran: tiempo, PID, ...
  - Reservar espacio en memoria y duplicar las zonas del padre
- *Modificar sistema de ficheros*
  - *Copiar la tabla de descriptores del padre sobre el hijo*
  - *Incrementar contadores de la tabla de ficheros abiertos*
- Devolver control
  - Devolver PID del hijo al proceso padre y cero al hijo como resultado de la función
  - Colocar a los dos procesos en estado preparado
  - Llamar al Scheduler

Algoritmo general de exec()

- Conseguir imagen del programa a ejecutar
  - Conseguir i-node del fichero ejecutable
  - Verificar número mágico y permiso de ejecución
  - Lectura de cabeceras. Comprobación de que es cargable
- Modificar contexto del proceso
  - Salvar temporalmente los parámetros de exec()
  - Liberar las regiones de memoria del proceso
  - Reservar espacio en memoria para las nuevas regiones
  - Cargar los contenidos del fichero ejecutable
  - Modificación de algunos campos del PCB:
    - Contador de programa, puntero de pila, ...
- Devolver control
  - Cargar variables de entorno y parámetros de la llamada exec() a la pila del proceso
  - Poner al proceso en estado preparado
  - Llamar al Scheduler

Resumen de Identificadores y Marcas (1)

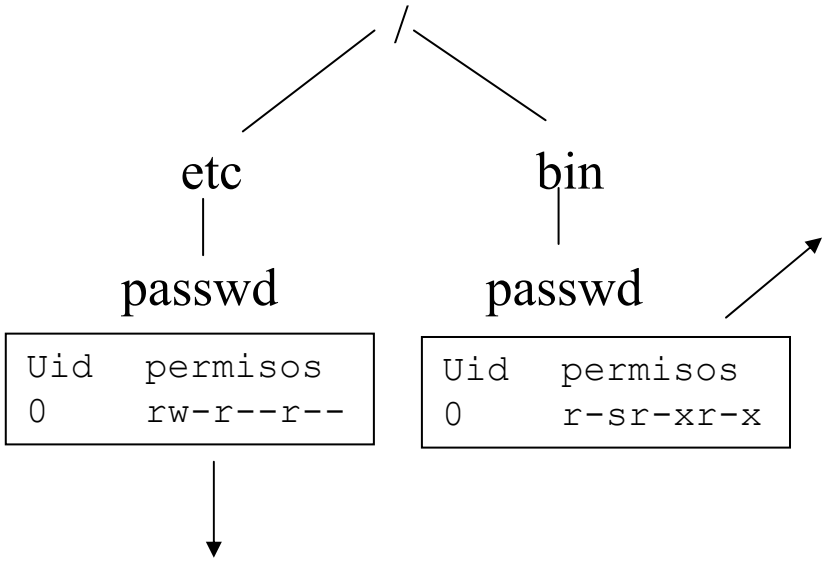
- Para cada usuario
  - UID (user id.): id. de usuario (nº natural)
    - UID=0 -> superusuario (root)
  - GID (group id.): id. de grupo al que pertenece (nº natural)
- Para cada fichero
  - Número de I-node: identificador numérico
  - UID del propietario
  - GID del propietario
  - Permisos de acceso (user-group-other)
  - Bits set-user-id, set-group-id: Permiten cambiar el usuario/grupo efectivo de un fichero.
  - Sticky bit: Permite gestión adecuada de ficheros de distintos usuarios en directorios compartidos (por ejemplo /tmp)

Resumen de Identificadores y Marcas (2)

- Para cada proceso
  - PID: identificador de proceso - getpid()
  - Process Group ID: Id. de grupo de procesos getpgrp()
    - setpgrp() PGID=PID (proceso líder)
    - setpgid(pid, pgrp)
  - Real User ID: UID del usuario que ha ejecutado el proceso getuid()
  - Real Group ID: GID del mismo usuario getgid()
  - Effective User ID: al ejecutar un programa, geteuid()
    SI (set-user-id==1 en el fichero ejecutable)
    EUID = UID del propietario del fichero (Ej: passwd)
    SINO EUID = RUID.
  - Effective Group ID: igual que EUID aplicado a grupo getegid()

Resumen de Identificadores y Marcas (3)

- Ejemplo de uso del bit set-user-id



- Nadie puede escribir sobre él, podría modificar los passwords
- Cualquiera puede leerlo, están encriptados

- Cualquiera puede ejecutarlo, sirve para cambiar el password
- Bit SETUID=1: el proceso que lo ejecuta pasa a tener EUID=0 (superusuario)
- Puede hacer cualquier cosa con /etc/passwd, puede modificarlo
- Durante un momento, el usuario que ejecuta /bin/passwd es superusuario:
  - No puede hacer nada que no haga /bin/passwd
  - No puede modificar /bin/passwd para que haga otras cosas

Herencia en fork()

- UID, GID, EUID, EGID, PGID
- Entorno, variables
  - cwd, umask, …
- *Comportamiento ante señales y mascara*
- *Tabla de descriptores de fichero*
- Otros
  - Segmentos de memoria compartida, límites de recursos del sistema, close-on-exec flag, terminal de control, …
- Diferencias entre padre e hijo
  - Valor devuelto por fork()
  - PID
  - Contadores de tiempo a cero en el hijo
  - *Ficheros bloqueados por el padre no lo están en el hijo*
  - *No se heredan las señales pendientes (ej. Alarmas)*

Herencia en exec()

- PID, PPID, PGID
- UID y GID (reales)
- Variables de entorno (salvo en execle y execve)
- *Señales pendientes*
- *Tabla de descriptores de fichero (salvo bit close-on-exec)*
- Otros
  - Bloqueo de ficheros, tiempo de ejecución acumulado, …
- En un exec cambia:
  - EUI, EGID (bits set-userID, set-groupID)
  - *Descriptores de fichero (bit close-on exec)*
  - *Comportamiento ante señales (no captura)*
  - Variables de entorno (en execle y execve)

Intérprete de comandos: **ej71.c**

```
#include <stdio.h> <stdlib.h> <unistd.h> <sys/wait.h>
#include "error.h"

int main(int argc,char *argv[]) { //comando sin parámetros
   switch ( fork() ){
   case -1:       /* error */
     fprintf( stderr, "no se puede crear proceso nuevo\n" );
     syserr( "fork" );
   case 0:        /* hijo */
     execlp(argv[1], argv[1], 0);
     printf( "No se puede ejecutar %s\n", argv[1]);
     syserr( "execlp" );
   default:         /* padre */
     if ( wait( NULL ) == -1 )
        syserr( "wait" ); /* trat. erroneo? */
     printf( "Ejecutado %s \n", argv[1] );
   }
}
```

Bucle de ejecución: **ej73.c (1)**

```
#include <stdio.h> <string.h> <stdlib.h> <unistd.h> <sys/wait.h>
#include "error.h"

#define BUFSIZE 1024
int main( ) {
    static char s[BUFSIZE];
    char *argt[BUFSIZE];
    int i;
    for(;;){      /* bucle infinito */
      fprintf( stderr, "\n_$ " );
      if (gets(s)==NULL) {
          printf("logout\n");
          exit(0);
      }
      argt[0] = strtok( s, " " );  /* argt[0] comando a ejecutar */
      if( 0 == strcmp( argt[0], "quit" )){ /*cierto si argt[0]="quit" */
          printf("logout\n");
          exit( 0 );
      }
      for( i = 1; (argt[i] = strtok( NULL, " \t" )) != NULL; i++ );
```

Bucle de ejecución: **ej73.c (2)**

```
        switch ( fork() ){
        case -1:        /* error */
            fprintf(stderr, "\nNo se puede crear proceso nuevo\n");
            syserr( "fork" );

        case 0:         /* hijo */
            execvp( argt[0], &argt[0]);
            fprintf(stderr,"\nNo se puede ejecutar %s\n", argt[0]);
            syserr( "execvp" );

        default:        /* padre */
            if( wait( NULL) == -1) syserr("wait");

        }/*switch*/
    }/*for*/
}/*main*/
```

Ejecución asíncrona: **ej1001.c (1)**

```
/* ej1001.c
   * Shell elemental con bucle de lectura de comandos con
     parametros
   * Uso: [arre|soo] [comando][lista parametros]
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "error.h"

#define BUFSIZE 1024
#define TRUE 1
#define FALSE 0

int main( ) {
   static char s[BUFSIZE];
   char *argt[BUFSIZE];
   int i, parate, pid;
```

Ejecución asíncrona: **ej1001.c (2)**

```
while(1){
    fprintf( stderr, "\n_$ " );
    gets( s );
    argt[0] = strtok( s, " " );
    if( 0 == strcmp( argt[0], "quit" )){
      printf("logout\n");
      exit( 0 );
    }

    for( i = 1; (argt[i] = strtok( NULL, " \t" )) != NULL; i++ );

    if( 0 == strcmp( argt[0], "soo" ) )
      parate = TRUE;
    else
      if( 0 == strcmp( argt[0], "arre" ) )
          parate = FALSE;
      else{
          printf( "\n Mande?" ); continue;
      }
```

Ejecución asíncrona: **ej1001.c (3)**

```
switch ( pid=fork() ){
  case -1:      /* error */
      printf(stderr,"\nNo se puede crear proceso nuevo\n");
      syserr( "fork" );

  case 0:       /* hijo */
      if(!parate) sleep(3);/*para que se note mas*/
      execvp( argt[1], &argt[1]);
      fprintf(stderr,"\nNo se puede ejecutar %s\n", argt[1] );
      syserr( "execvp" );

  default:      /* padre */
      if(parate)
          while( pid != wait( NULL))
               if( pid == -1 ){
                        fprintf(stderr,"\nMuy raro. Bye\n");
                        exit( 1 );
               }
  }/*switch*/
}/*while*/
}/*main*/
```

Procesos: Sincronización

[SGG05]: capítulo 6

Resumen

- Problema
- Definiciones básicas
- Problema de sección crítica
- Caracteristicas de la solución
- Solución monoprocesador
- Soporte hw a la solución: swap y test&set
- Mutex
- Semáforos

Problema

Acceso concurrente a variable compartida

Proceso 1: `i=i+1`
Proceso 2: `i=i-1`

`r0=@i` `i=0`

Proceso 1

```
ldr r1,[r0]
add r1,r1,#1
str r1,[r0]
```

Proceso 2

```
ldr r1,[r0]
sub r1,r1,#1
str r1,[r0]
```

Ejemplo práctico: Número de enlaces de un fichero en unix

Problema generalizable a:
- Varios procesos (n)
- Multiprocesadores

Posibles resultados

`r0=@i` `i=0`

Proceso 1	Proceso 2	Proceso 2	Proceso 2
			`ldr r1,[r0]`
`ldr r1,[r0]`			
	`ldr r1,[r0]`		`sub r1,r1,#1`
`add r1,r1,#1`			
	`sub r1,r1,#1`		`str r1,[r0]`
`str r1,[r0]`			
	`str r1,[r0]`	`ldr r1,[r0]`	
		`sub r1,r1,#1`	
		`str r1,[r0]`	

`i=-1` `i=0` `i=1` tiempo

NO OK NO

Definiciones básicas

- **Problema: El acceso concurrente a datos/recursos compartidos puede llevar a inconsistencias en los datos/recursos**
- **Condición de carrera** (race condition): Situación como la anterior cuyo resultado depende del orden concreto de ejecución de las instrucciones
- **Sección crítica**: Zona de código donde un proceso accede a recursos compartidos con otros procesos
- **Procesos cooperantes**: Aquellos cuyo resultado puede afectar o verse afectado por la ejecución de otros procesos.

Problema de sección crítica

- Sistema con n procesos $P_1, P_2, \ldots, P_n$
- Recursos compartidos (variables, ficheros, etc.)
- Cada proceso tiene una sección crítica (SC) en el que quiere acceder a los recursos compartidos

- Diseñar un protocolo que permita a los procesos compartir los recursos sin incurrir en inconsistencias (condiciones de carrera)

Solución al problema de SC

Debe satisfacer 3 propiedades:

- **Exclusión mutua**: Que no entren dos procesos en sus SCs de forma simultánea
- **Progreso**: Si ningún proceso está en su SC y un subconjunto de procesos quiere entrar a sus SCs, la decisión se toma entre ellos y sin postponerla indefinidamente
- **Espera acotada**: Si un proceso P_i quiere entrar en su SC, hay una cota o límite en el número de veces que otros procesos P_j `(i≠j)` pueden entrar en sus SCs antes de que lo haga P_i

Exclusión mutua → proporcionada por HW

Progreso + espera acotada → proporcionada por SW

Esquema de los procesos

```
do {
    Sección de entrada
       Sección crítica
    Sección de salida
       resto
} while(TRUE)
```

Sección de entrada: Obtener **llave** para entrar a la SC
Sección de salida: Liberar **llave**
Resto: Zona de código local del proceso

Solución para un solo procesador

- **Inhabilitar interrupciones** durante el acceso a la variable compartida
  - Asegura el acceso en exclusión mutua
- **Problema**: Ineficiente en multiprocesadores

- **Lo importante: Asegurar a un proceso el acceso y modificación de la variable compartida sin interrupción por otro proceso**

- **Operación atómica**: Conjunto de instrucciones que se ejecutan sin interrupción.
  - O se ejecutan todas o ninguna
  - Una vez comenzada la operación atómica debe terminar sin interrupción

Soporte hw a la exclusión mutua

- Instrucciones habituales

  Swap: Intercambio **atómico** var ↔ var o reg ↔ var

  En esencia es 2 ó 1 ldr+str atómico

  Test&set: Lectura y modificación **atómico** de var

  En esencia es un ldr+str atómico

- En ARM v4

  `swp`: Intercambio **atómico** reg ↔ mem 32 bits

  ```
  swp r0,r0,[r1]    ;intercambio atómico r0 ↔ Mem32[r1]
  ```

  `swpb`: Intercambio **atómico** reg ↔ mem 8 bits

  ```
  swpb r0,r0,[r1]   ;intercambio atómico r0[7:0] ↔ Mem8[r1]
  ```

Mutex

- Variable booleana S
- Operaciones **atómicas**:
  - **Lock**: Coger llave
  - **Unlock**: Dejar llave

Proceso 1

```
lock(S);
  //SC
unlock(S);
```

Proceso 2

```
lock(S);
  //SC
unlock(S);
```

```
void lock(S) {
  while (S!=0) esperar;
  S=1;
}
```

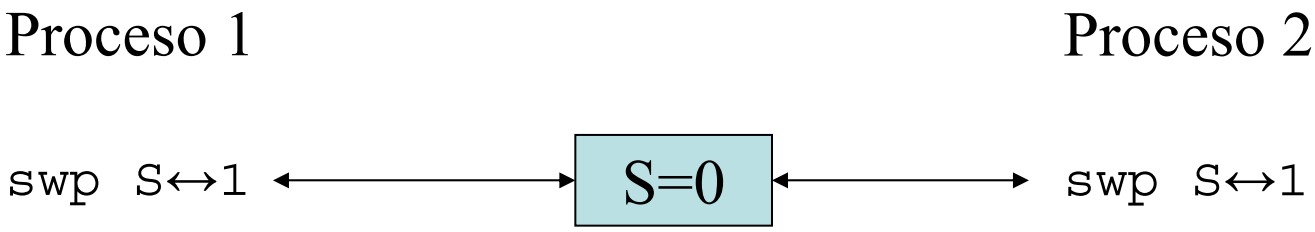
```
void unlock(S) {
  S=0;
}
```

- Problema: **Espera activa** de los procesos que quieran coger la llave y no puedan

Mutex bajo nivel (ARM v4)

```
Lock(*S)
      ;r0=@S
      mov r1,#1
whi   swp r1,r1,[r0]   ;S↔1 atómico
      cmp r1,#0
      bne whi
```

```
Unlock(*S)
   ;r0=@S
   mov r1,#0
   str r1,[r0]
```



Sólo el proceso que ejecuta primero swp toma la llave
Los otros procesos quedan en espera activa

Semáforos

- Generalización del mutex a varias llaves.
- Variable int S
- Operaciones **atómicas**:
  - **Wait**: Coger llave
  - **Signal**: Dejar llave

Proceso 1

```
wait(S);
  //SC
signal(S);
```

Proceso 2

```
wait(S);
  //SC
signal(S);
```

```
void wait(S) {
  while (S≤0) esperar;
  S=S-1;
}
```

```
void signal(S) {
  S=S+1;
}
```

- Problemas: **Espera activa + malos usos**

Semáforos

- Reducir la **espera activa** (ahora muy corta)
- Se le añade al semáforo una cola de procesos bloqueados

```
void wait(S) {
  S.valor--;
  if (S.valor<0) {
    añadir_proceso(S.cola);
    bloquear_proceso();
  }
}
```

```
void signal(S) {
  S.valor++;
  if (S.valor<=0) {
    P=sacar_proceso(S.cola);
    desbloquear(P);
  }
}
```

- Si `S.valor>0` Número de procesos que pueden pasar el semáforo
- Si `S.valor<0` Número de procesos bloqueados → **espera inactiva**

- Sigue habiendo **espera activa** en el acceso en exclusión mutua a las variables del semáforo (`S.valor` y `S.cola`), pero es muy corta
- Semáforos normalmente implementados en el kernel, lo que facilita su uso

Semáforos

- Ejercicio: Implementar el wait y signal en ARM v4 con swp eliminando condiciones de carrera

Señales

Señales (signals)

- Características
- ¿Quién puede enviar señales?
- Comportamientos
- Tipos y llamadas al sistema asociadas
- Ejemplo sencillo señales
- Terminología e Implementación
- Fork y exec *versus* señales
- Ejemplos
- Problemas con las señales
- Señales seguras

[Ste05]: capítulo 10

Características

- Comunican eventos especiales a procesos en ejecución
  - Son interrupciones software
  - No contienen información. El proceso señalado no conoce la identidad del proceso señalador
  - No es la forma habitual de comunicar procesos. Pueden servir para sincronizar

¿Quién puede enviar señales?

- Un proceso recibe una señal originada por:

  – El kernel en respuesta a una condición hardware violenta:

    SIGSEGV intento de acceso a @ fuera de su espacio

    SIGFPE error en aritmética FP

  – El kernel en nombre del propio proceso:

    el proceso se quiere programar una alarma

  – El kernel en nombre del usuario:

    se generan desde el teclado

    Ctrl \ señala a todos los procesos creados por el usuario

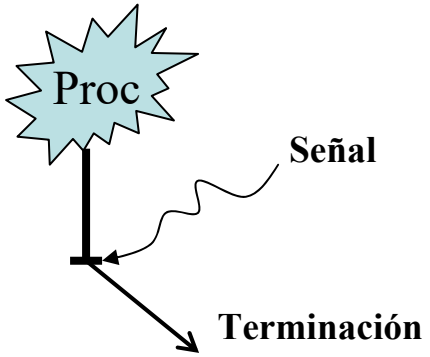
  – El kernel en nombre de otro proceso:

    se generan con kill por parte del proceso señalador

Comportamientos (1)

- El proceso señalado puede:

① No querer manifestar su disposición frente a la señal
actúa el comportamiento por **defecto**
normalmente terminación del proceso (+/- core)

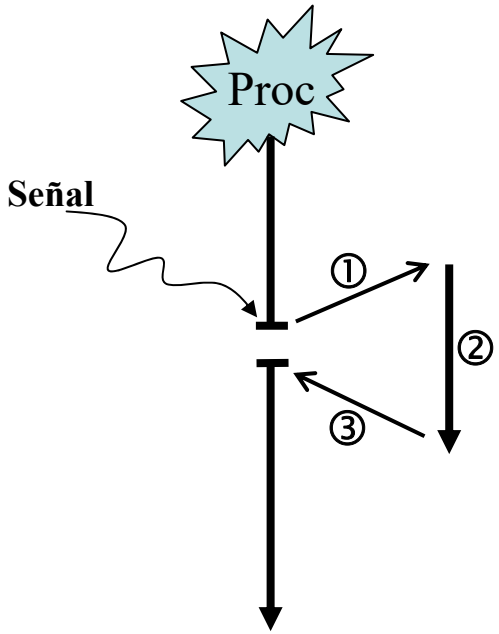


② Manifestar su disposición a **ignorar** la señal
el proceso se hace inmune a la señal

Comportamientos (2)

③ Querer **capturar** la señal

actúa el comportamiento programado



① **Llamada a la rutina de tratamiento (signal handler)**

② **Ejecución y posible indicación de querer capturar de nuevo la señal**

③ **Retorno y restauración del contexto (pila de usuario)**

Tipos

- Declaradas en <signal.h>
- Constantes que empiezan por SIG
- Todas asociadas a un entero positivo
- Lista completa en /usr/include/sys/iso/signal_iso.h

Nombre	Núm.	Defecto	Observaciones
SIGINT	2	terminación	Ctrl C
SIGQUIT	3	terminación + core	Ctrl \
SIGKILL	9	terminación	no ignorar ni capturar
SIGPIPE	13	terminación	escritura en pipe cerrada
SIGALRM	14	terminación	se fija con alarm
SIGTERM	15	terminación	como SIGKILL pero si ignorar
SIGUSR[1,2]	16,17	terminación	definidas por el programador
SIGCHLD	18	ignorada	muerte del proceso HIJO
SIGSTOP	23	detener	no ignorar ni capturar

Llamadas al sistema (1): **signal**

- Sintaxis: # include <signal.h>
  void (* **signal** (*sig*, *func*)) (int)
  int *sig*; void (* *func*) (int);
- Acción: Cambia comportamiento para la señal
- Uso: signal (señal, comportamiento);

SIGINT,	SIG_DFL=0	①	defecto
SIGQUIT,	SIG_IGN=1	②	ignorar
14, ...	rutina_captura	③	capturar

- Devolución: anterior comportamiento ó –1 si error

#define SIG_ERR (void (*)()) -1

Llamadas al sistema (1): **signal** (cont.)

- Si programamos capturar la señal:

  En la rutina de captura: void func (int)

  ① reset del comportamiento al defecto
  ② ejecución del código de la rutina
  ③ retorno y restauración del estado

  un solo parámetro para func = número de la señal

Si la señal llega durante la ejecución de una SC bloqueante: la interrumpe, devuelve –1 y errno = EINTR

Llamadas al sistema (2): **kill**

- Sintaxis: # include <signal.h>
  int **kill** (pid_t *pid*, int *sig*)
- Acción: envía una señal a un proceso
  (ruid_señalado = euid_emisor)
- Uso: kill (pid_señalado, señal);
- Devolución: 0 si bien, –1 si error

Llamadas al sistema (2): **kill** (cont.)

- *pid* > 0 destino = *pid*

- *pid* = 0 destino = todos los procesos con *process group id* igual al del emisor
  (todos los procesos de su grupo)

- *pid* = -1 a) si (emisor != superusuario)
  destino = todos los procesos con ruid igual al euid del emisor
  b) si (emisor == superusuario)
  destino = todos los procesos (! PID=0, PID=1)

- *pid* < -1 destino = procesos cuyo *process group id* igual al valor absoluto de *pid*

Llamadas al sistema (3): **alarm**

- Sintaxis: # include <unistd.h>
  unsigned int **alarm** (unsigned int *sec*)
- Acción: programa la recepción de SIGALRM para dentro de *sec* segundos

- Uso: alarm (5); /* alarm clock = 5 */
  alarm (0); /* cancelación */

  Ojo !!: solo un alarm clock activo a la vez

  Si se quiere capturar, signal() antes de alarm()
- Devolución: segs que quedan del anterior alarm
  0 la primera vez

Llamadas al sistema (4): **pause**

Sintaxis: # include <unistd.h>

int **pause** (void)

Acción: suspende al proceso hasta la llegada de una señal cualquiera capturada (si defecto: terminar, si ignorar → sigue pause)

- Uso: pause ( );

Devolución: –1 si la señal se captura y se retorna del signal_handler (errno = EINTR)

Ejemplo sencillo señales

```
#include <signal.h>
void fcaptura();
main(){
   signal(SIGALRM, fcaptura);
   alarm(10);
   pause();
   printf("Trabajo terminado\n");
}
void fcaptura( ){ };
```

SIG_DFL
SIG_IGN
rutina de captura

devuelve?

reprogramación:
signal(SIGALRM,fcaptura);

Terminología

- Una señal es GENERADA para un proceso cuando ocurre el suceso que causa la señal
- Una señal es TRATADA en un proceso cuando la acción programada para la señal se lleva a cabo en el proceso receptor
- Una señal está PENDIENTE desde que se genera hasta que es tratada

Implementación

- Campos añadidos en la struct PCB del proceso

  – Conjunto de señales que el proceso tiene pendientes:

    Señales pendientes: 1 bit por señal

    consultada cada vez que el proceso entra en ejecución

  – Indicación para cada señal del comportamiento programado con signal

    ① #define SIG_DFL (void (*)()) 0

    ② #define SIG_IGN (void (*)()) 1

    ③ @ rutina de captura

fork() *versus* señales

- El proceso HIJO:
  - No hereda las señales pendientes en el proceso PADRE:

    señales pendientes en el HIJO = Ø
  - Hereda todos los comportamientos programados por el proceso PADRE:

    ① defecto
    ② ignorar
    ③ capturar

exec() *versus* señales

- El proceso ejecutado:
  - Mantiene todas las señales que están pendientes:

    mismo struct PCB

  - No mantiene todos los comportamientos programados:

    ① defecto
    ② ignorar
    ③ capturar pasa a defecto: Ya que ha desaparecido el código de la rutina de captura

/* ej90.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main(){
    void newhandler(int);
    void (* syshandler)(int);

    syshandler = signal(SIGALRM, newhandler);
    alarm(5);
    printf("Me bloqueo esperando la alarma\n");
    pause();
    printf("Ya me he despertado\n");
    exit(0);
}
```

```
/* Rutina mia de servicio */
void newhandler(int n)
{

    printf("En la rutina de servicio %d\n",n);
};
```

podemos imprimir el parámetro...

Ejercicio: /* autolesion.c */

```
#include <stdlib.h>
#include <signal.h>
#include "error.h"
#define MAL (void (*)(int)) -1

void main(){
    int i, *ip;
    void func(), captura();

    ip = (int *) func;
    for( i= 1; i<41; i++ )
      if ((i!=9)&&(i!=23))   /* no podemos capturar SIGKILL (9)
                                ni SIGSTOP(23) */
        if (signal( i, captura ) == MAL) syserr( "signal" );
    *ip = 1;      /* PROVOCA EXCEPCION Y ENVIO DE SEÑAL SIGSEGV(11)*/
    printf( "Asignado ip\n" );
    func();
}
```


```
void func(){}

void captura(int n) {
  printf( "Capturada %d\n", n );
  exit( 1 );
}
```

SIGCHLD / SIGCLD

Señal enviada a un proceso cuando muere/para un hijo.

Por defecto se **ignora**

ej1001.c: zombies en modo asíncrono

Solución?

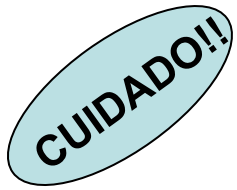
- SIGCHLD (BSD, POSIX)
  - como el resto de señales (si se ignora, deja zombies)

En hendrix existen los 2 nombres, con el mismo valor = 18, con este comportamiento

- SIGCLD (System V, **Solaris**)
  - signal (SIGCLD, SIG_IGN)
    - NO deja zombies
    - Si se llama a wait => bloquea hasta muerte de todos los hijos y luego devuelve -1
  - signal (SIGCLD, captura)
    - si ya hay un zombie => llamar a captura() YA
    - sino se llamará a captura() cuando muera uno

Función de captura para SIGCLD

```
void captura () {
    int pid, estado;
    signal (SIGCLD, captura);
    pid = wait (&estado);
    printf ("Ha terminado PID=%d con estado %x \n",
            pid, estado);
}
```



Implementaciones antiguas de Sistem V puede entrar en bucle infinito de llamadas a captura sin ejecutar wait.
En Solaris 10 (hendrix) no pasa

Problemas con señales

- Las señales expuestas hasta ahora son las conocidas como señales no seguras (non reliable signals).
  - Se recomienda en programas modernos no utilizarlas
  - Es necesario conocerlas para analizar o cambiar programas viejos, por ello las seguiremos estudiando
- Problemas que dan las señales no seguras:
  1. Al capturar una señal el comportamiento de la misma cambia a SIG_DFL
  2. Posibles bloqueos por llegadas de señales en momentos no adecuados (por ej. antes de un pause)
  3. Interrupción de SCs bloqueantes

Problemas con las señales (1)

- Recordar:

  con *signal* hay reset al SIG_DFL en la rutina de captura

- Solución (no siempre funciona):

  reprogramar el comportamiento de captura en la rutina:

si llega señal => ¡el proceso acaba!

```
void rutina_captura(){
        signal(SIGUSR1, rutina_captura);
        ...
}
```

no siempre funciona!

Problemas con las señales (1): /* sig_dfl.c */

Al capturar señal, el comportamiento pasa a SIG_DFL

```
#include <stdlib.h>, <unistd.h>,<signal.h>,"error.h"
#define MAL (void (*)(int))-1
#define ESPERA 1

int main() {
   if (signal(SIGINT,captura)==MAL) syserr("signal");
   while (1) pause();
}

void captura(int n) {
   void (*ff)(int);
   printf( "\nSeñal capturada %d\n", n );
   sleep(ESPERA);
   if (signal(SIGINT,captura)==MAL) syserr("signal");
   printf( "Nuevo comportamiento\n");
}
```

Problemas con las señales (2): pelosig.c

```
#include <signal.h>
#include <stdio.h>

main(){
int pid;

    signal(SIGUSR1,f);
    if((pid=fork())==0){
        pid=getppid();
        while(1){
            fprintf(stderr,"h");
            kill(pid,SIGUSR1);
            pause();
        }
    }
    else{
        pause();
        while(1){
            fprintf(stderr,"p");
            kill(pid,SIGUSR1);
            pause();
        }
    }
}
```

```
void f(){
    signal(SIGUSR1,f);
    fprintf(stderr,"-");
}
```

Dos posibles resultados:

1) h-p-h-p- … -h-p- (acabado en p-)
2) h-p-h-p- … -h- (acabado en h-)

en los dos casos, al final "se cuelga"

Problemas con las señales (3): /* ej901.c */

Interrupción de llamadas al sistema bloqueantes

```
#include <stdio.h>, <stdlib.h>, <unistd.h>
#include <signal.h>, "error.h"

#define MAL (void (*)()) -1

void main(){
   if( signal(SIGALRM, trap) == MAL )
       syserr("signal");
   alarm(2);
   getchar();
   printf("Caracter leido\n");
   pause();          /* ¿? */
   puts("Aquí seguiría el programa...\n");
}
```

De señales no-seguras a señales seguras

Cosas que mejoran de *non-reliable signals* a *reliable signals*:

	non-reliable	reliable
• Posibilidad de dejar señales bloqueadas para qué? – ejercicio pelosig.c	NO	SI
• Desprotección del proceso en la rutina de captura por qué? – reset al DFT en captura	SI	opcional
• Posibilidad de *restart* SC bloqueantes interrumpidas	NO	opcional

Funciones no reentrantes => efectos laterales que siguen existiendo

mala programación!

Soluciones

- Necesitamos poder recordar la GENERACION de una señal para TRATARLA más tarde
  - Significado: poder retrasar la acción programada para la señal
- Necesitamos poder BLOQUEAR señales durante el tiempo que no queremos TRATARLAS
- Se puede hacer con señales seguras
  - signal_sets Conjunto de señales
  - sigprocmask SC para bloquear/desbloquear señales
  - sigpending SC para saber señales pendientes del proceso
- Entre sigprocmask y pause aun puede llegar una señal
  - Se resuelve ejecutando sigprocmask y pause de forma atómica, es decir, con **sigsuspend**

Signal sets

- Manipulación del conjunto de señales

recordar:	máscara de bits para señales pendientes, 1 bit por señal
si bit i = 0 => señal número i $\notin$ set	si bit i = 1 => señal número i $\in$ set

int sigemptyset (sigset_t *set)

int sigfillset (sigset_t *set)

int sigaddset (sigset_t *set, int signo)

int sigdelset (sigset_t *set, int signo)

int sigismember (sigset_t *set, int signo)

Llamar siempre a sigemptyset o sigfillset para asegurarse correcta inicializacion de sigset

Funciones sobre sets

(*tipo* sigset_t *definido en signal.h*)

Signal mask y sigpending

- Signal mask: máscara indicando qué señales se deben BLOQUEAR para un proceso

Para modificar/consultar *signal mask*

int **sigprocmask** (int how, sigset_t *set, sigset_t *oset)

how: SIG_BLOCK añade *set a cjto. señales bloqueadas

SIG_UNBLOCK quita *set del cjto. señales bloq.

SIG_SETMASK pone nuevo cjto. señales bloq.

si oset != NULL devuelve en **oset* el cjto. previo de señales bloqueadas

int **sigpending** (sigset_t *set)

Para consultar qué señales están pendientes

devuelve en **set* el conjunto de señales pendientes

sigsuspend()

- Modifica signal mask (sigprocmask) y hace pause de forma atómica

int **sigsuspend** (sigset_t *sigmask)

  - asigna sigmask a *signal mask*
  - bloquea al proceso
    hasta que llega una señal NO bloqueada NI ignorada
  - restaura la máscara previa

sigaction()

- Versión *reliable* de signal (sustituye y amplía a signal)

int **sigaction** (int signo, struct sigaction *act,
struct sigaction *oact)

  - signo: número de la señal sobre la que fijar el comportamiento
  - act: nuevo comportamiento (input)
  - oact: anterior comportamiento (output)

struct sigaction

```
struct sigaction {
    void (*sa_handler)();     /* SIG_IGN, SIG_DFL o captura */

    sigset_t sa_mask;         /* señales a bloquear durante la captura */
                              añadidas a signal mask (restaurado al
                              acabar) */
    int sa_flags;             /* para programar las opciones:
}
```

		defecto:
SA_RESTART (4)	reinicio automático de SC	no reinicio
SA_RESETHAND (2)	reset a DFL en captura	no reset
SA_NODEFER (16)	no bloqueo de la propia señal durante la función de captura	bloqueo

Solución de problemas con las señales

1. Reset al SIG_DFL en la rutina de captura modificado en señales seguras:
   - **Es opcional con sigaction**
   - NO hay reset a SIG_DFL en la rutina de captura (4.2 BSD y SVR3)
2. Bloqueo de señales imposible con no seguras
   - **Es posible con sigprocmask**
3. Un proceso captura señal mientras está bloqueado en una SC "lenta", la SC es interrumpida, devuelve –1 y errno = EINTR. Si se quería reiniciar la SC había que hacerlo a mano.
   - **Es opcional con sigaction** (sa_flags de struct sigaction) para ciertas SCs (ioctl, read, readv, write, writev, wait, waitpid):
     - interrupción SIN reinicio (no poner flag SA_RESTART)
     - interrupción CON reinicio automático (SA_RESTART)

Problemas con las señales (seguras y no seguras)

- Funciones no reentrantes (ej. *malloc*)
  - un proceso hace una llamada a *malloc*
  - durante el servicio de *malloc* llega una señal a **capturar**
  - en la rutina de captura se vuelve a llamar a *malloc*

  Si la función no es reentrante (ej. usa variables *static*) la 2ª llamada puede provocar problemas a la 1ª
- En [Ste05] Fig 10.4 (pag 306) tabla de funciones reentrantes, es decir, que se pueden llamar con “menos” problemas desde rutina de captura.
- En general:

  ¡Cuidado con los efectos laterales

  provocados por una rutina de captura!

señal que en captura modifica errno

```
if (wait(NULL) == -1)
        if(errno == ECHILD);
        ...
```

Ejercicio

Reescribir el siguiente código usando las llamadas de señales seguras.

Deben programarse las señales de forma que su comportamiento sea lo más parecido posible al código original

```
void captura(){
   printf("Funcion de captura...\n");
}

main(){
   signal(SIGALRM, captura);
   alarm(5);
   pause();
   printf("Fin programa\n");
}
```