

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



Universidad de Zaragoza



Proceso \neq programa

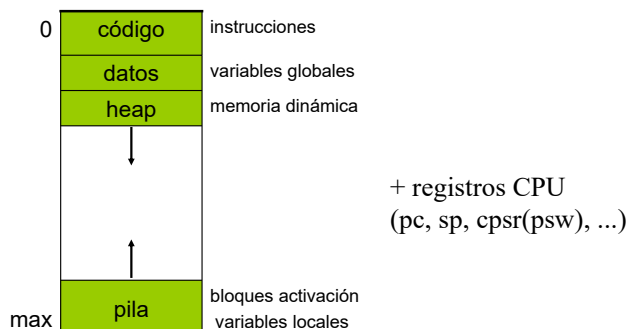
- Programa es una entidad pasiva
 - Fichero ejecutable con instrucciones y descripción de variables globales que está en disco
- Proceso es una entidad activa
 - pc que indica la siguiente instrucción que hay que ejecutar
 - Un conjunto de recursos asignados: CPU, MEM, E/S
- El programa se convierte 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)



Universidad de Zaragoza



Partes de un Proceso



Universidad de Zaragoza



Estados de un proceso

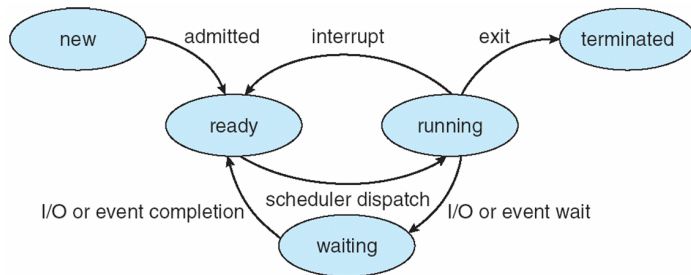
- **Nuevo:** El proceso se está creando (asignando recursos)
- **Ejecución:** La CPU está ejecutando instrucciones del proceso
- **Bloqueado:** El proceso está esperando la ocurrencia de algún evento (E/S, señal, mensaje)
- **Preparado:** El proceso está esperando la asignación de CPU (scheduler)
- **Terminado:** El proceso ha acabado (liberando recursos).
- En un instante dado, la CPU sólo está ejecutando 1 proceso
 - Si CPU de n núcleos, 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.



Universidad de Zaragoza



Estados de un proceso



Evolución de un proceso

Ráfagas de ejecución separadas por E/S:

Repetir
Trabajo en CPU
Entrada / Salida
Hasta FIN

→

Programar controlador
Esperar respuesta

Tiempos de E/S usualmente largos

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



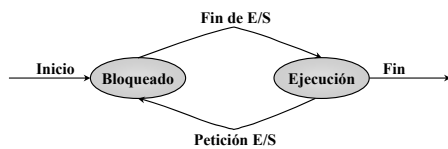
Procesos limitados por CPU: muy poca E/S



■ Cálculo

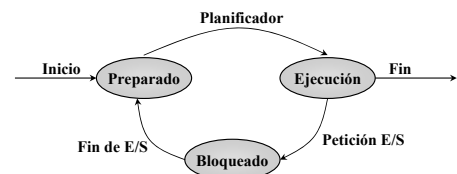
□ Entrada/Salida

Estados de un proceso (1)



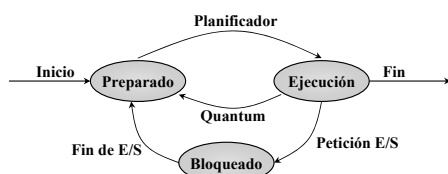
- Sistema Operativo Monoprogramado
 - E/S: pérdida de tiempo de CPU
- Sistema Operativo Multiprogramado
 - Varios procesos en el sistema
 - E/S bloqueante: cesión de la CPU a otro proceso

Estados de un proceso (2)



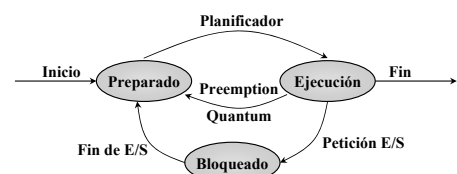
- Planificador (Scheduler)
 - Selecciona un proceso entre los *Preparados*
- El proceso en *Ejecución* deja CPU cuando pide E/S
 - Otros procesos pueden esperar indefinidamente
 - No apto para Sistemas *Interactivos* (solo para *Lotes*)

Estados de un proceso (3)



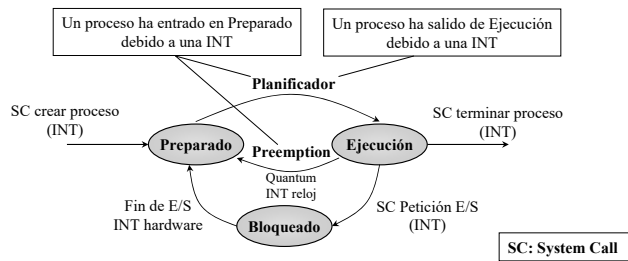
- Quantum
 - Tiempo máximo de permanencia en *Ejecución*
 - Controlado por la rutina de interrupción del reloj
 - El proceso en *Ejecución* deja CPU cuando pide E/S o cuando agota su tiempo de Quantum
 - Tiempo de espera de otros procesos limitado
 - Apto para Sistemas *Interactivos* (*Tiempo Compartido*)

Estados de un proceso (4)



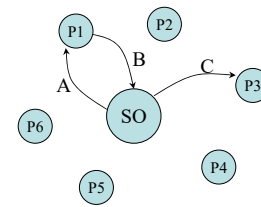
- Preemption
 - Si $Prioridad(Preparado) > Prioridad(Ejecución)$
=> expulsar proceso de *Ejecución*
- Otros autores entienden por Preemption:
 - Un proceso lógicamente ejecutable cede la CPU a otro
 - Por agotamiento de Quantum
 - Por mayor Prioridad

SO Como gestor de eventos



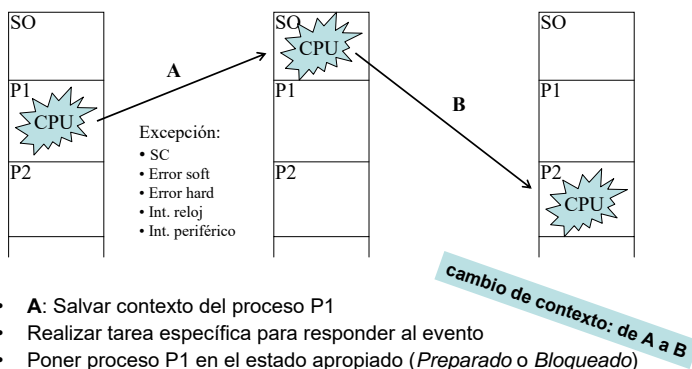
- Todo cambio de estado proviene de una Interrupción
 - El Sistema genera eventos (interrupciones)
 - En cada evento, el Sistema Operativo toma el control y gestiona los cambios de estado pertinentes

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)

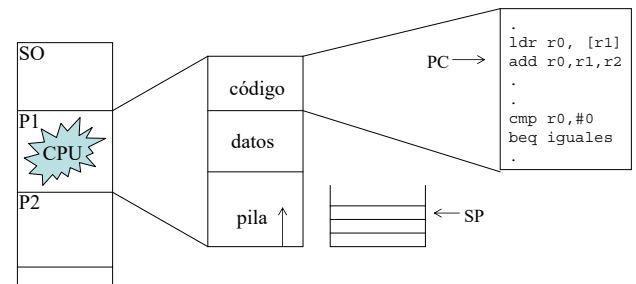
Dinámica del SO



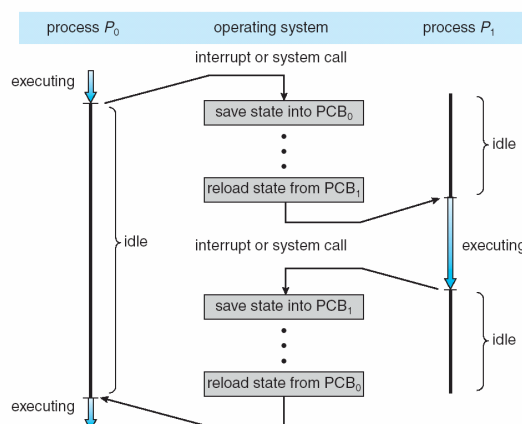
- A: Salvar contexto del proceso P1
- Realizar tarea específica para responder al evento
- Poner proceso P1 en el estado apropiado (*Preparado* o *Bloqueado*)
- Seleccionar proceso entre *Preparados* (P2): Planificador (Scheduler)
- B: Restaurar contexto del proceso P2 (Dispatcher)

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, cps, 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

- 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

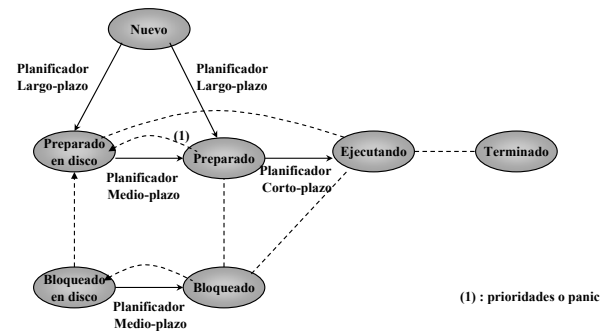
- Largo plazo
 - Decide sobre la creación de nuevos procesos, determina qué programas son admitidos por el sistema para ser procesados
 - Controla el grado de multiprogramación
 - Mayor número de procesos en el sistema implica menor porcentaje de tiempo dedicado a cada proceso
- Medio plazo
 - Swapping: 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



Universidad de Zaragoza



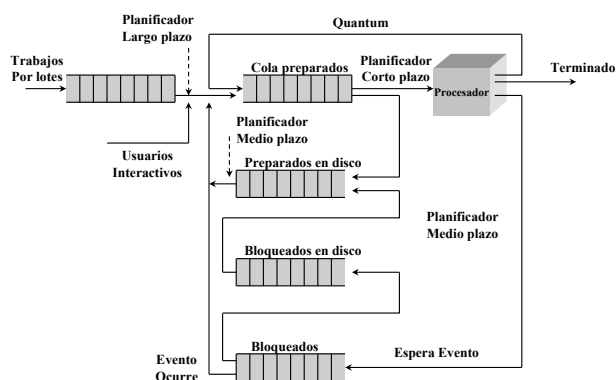
Planificador y estados de un proceso



Universidad de Zaragoza



Diagrama de colas para la planificación



Universidad de Zaragoza



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



Universidad de Zaragoza



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



Universidad de Zaragoza



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



Universidad de Zaragoza



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



Universidad de Zaragoza



Planificación por Prioridades

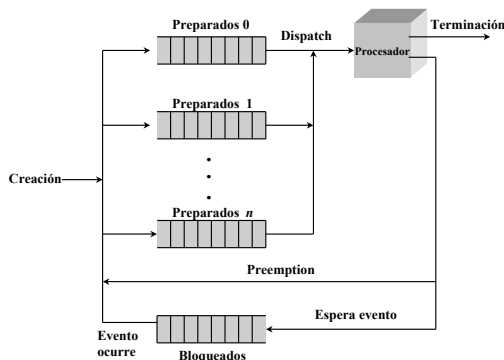
- Cada proceso tiene asignada una prioridad
- El Planificador elige siempre el proceso con mayor prioridad entre los preparados
 - Implementación típica: una cola de preparados para cada prioridad
- Los procesos de baja prioridad pueden sufrir inanición
 - A veces los procesos cambian su prioridad en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



Universidad de Zaragoza



Colas de prioridad



Universidad de Zaragoza



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



Universidad de Zaragoza



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



Universidad de Zaragoza



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

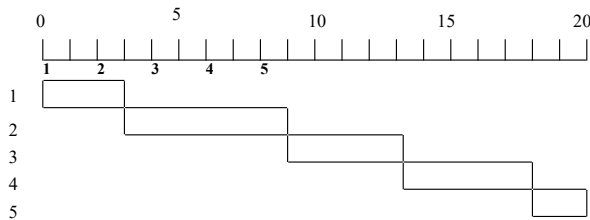
Proceso	Tiempo de llegada	Tiempo de servicio
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Universidad de Zaragoza



First-Come-First-Served (FCFS)



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



Universidad de Zaragoza



First-Come-First-Served (FCFS)

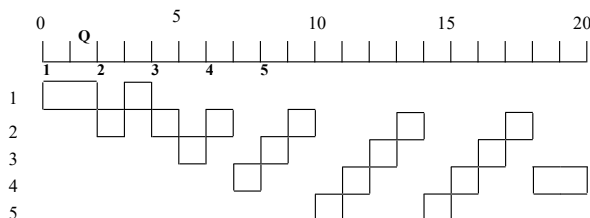
- Un proceso corto puede ser obligado a esperar durante mucho tiempo antes de entrar a ejecutarse
- Favorece a los procesos CPU-bound
 - Los procesos con mucha E/S deben esperar hasta que los CPU-bound abandonan el procesador
- Minimiza el número de cambios de contexto



Universidad de Zaragoza



Round-Robin



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



Universidad de Zaragoza



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

$$\%T_{cc} = 100 * \frac{T_{cc}}{T_{cc} + Q} \quad (\text{Mínimo})$$

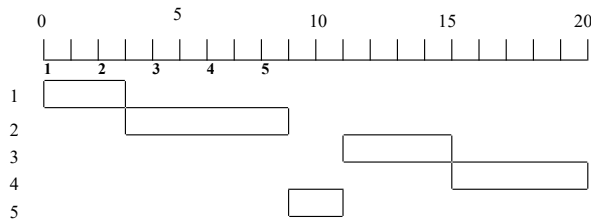
- Tcc disminuye cuando aumentan prestaciones
- Si Q constante => %Tcc disminuye, tiende a cero
- Se puede disminuir Q para mejorar el servicio sin aumentar la pérdida de prestaciones



Universidad de Zaragoza



Shortest Process Next



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



Universidad de Zaragoza



Shortest Process Next

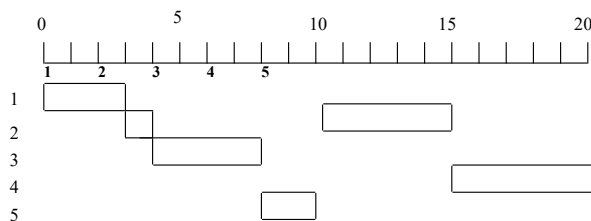
- Minimiza el tiempo medio de espera (óptimo)
- El comportamiento de los procesos largos es menos previsible
- Los procesos largos pueden sufrir inanición
- Debe estimar el tiempo de proceso
- Si el tiempo estimado para un proceso no es el correcto, el SO puede abortarlo



Universidad de Zaragoza



Shortest Remaining Time



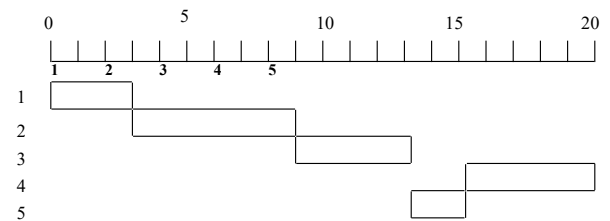
- Versión preemptive de la política shortest process next
- Debe estimar el tiempo de proceso
- Posible inanición de procesos largos



Universidad de Zaragoza



Highest Response Ratio Next (HRRN)



- Elige el siguiente proceso según mayor ratio

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

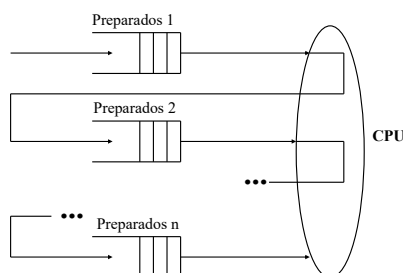
- No preentive
- Resuelve problemas inanición



Universidad de Zaragoza



Colas con realimentación



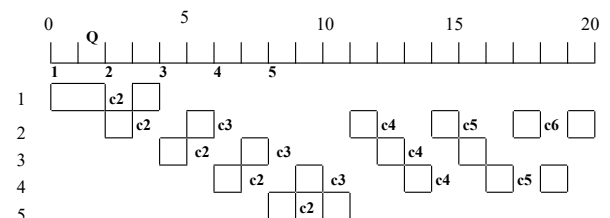
- Varias colas con distinta prioridad
 - Los procesos pasan de unas a otras en función de su comportamiento temporal



Universidad de Zaragoza



Colas con realimentación



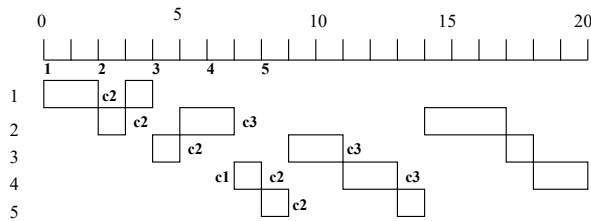
- Penaliza a los procesos que más tiempo llevan en ejecución



Universidad de Zaragoza



Colas con realimentación



- Quantum variable según cola

- $Q(c1)=1, Q(c2)=2, \dots$

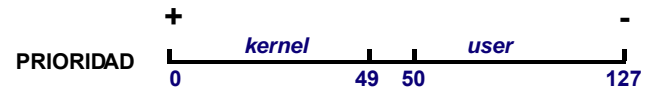


Universidad de Zaragoza



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



Universidad de Zaragoza



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 realimentación (configurable por el administrador)



Universidad de Zaragoza



Solaris Dispatch Table

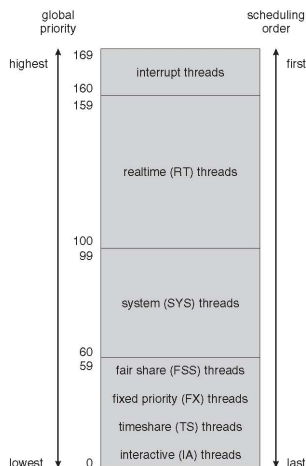
priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59



Universidad de Zaragoza



Planificación en Solaris



Universidad de Zaragoza



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



Universidad de Zaragoza



Procesos: Llamadas al Sistema

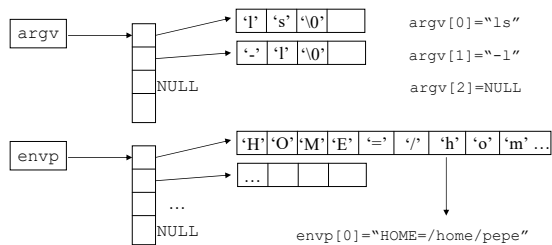
[Ste05]: capítulo 8



Estructura de Programa en C

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

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



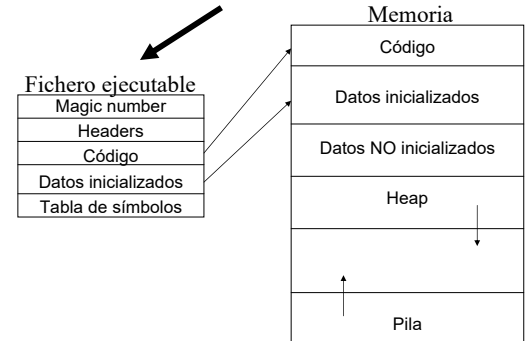
Universidad de Zaragoza

2



Imagen de un proceso

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



Universidad de Zaragoza

3



UNIX: Llamadas asociadas

- Cada proceso en Unix se identifica por un pid (process identifier) 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)



Universidad de Zaragoza

4



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 */
```



Universidad de Zaragoza

5



```
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 ejecución\n");
fork()
 padre hijo

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

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



- 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 exec1(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 exec1p(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)



- 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()`**



```
#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



Voluntaria: <code>exit(0..255)</code>	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
```

```
*p estado:
```

Terminación voluntaria del hijo

estado	0
8 bits	8 bits

Terminación involuntaria del hijo

--	--	--

1: core (bit 7) N° señal



- Caso más frecuente

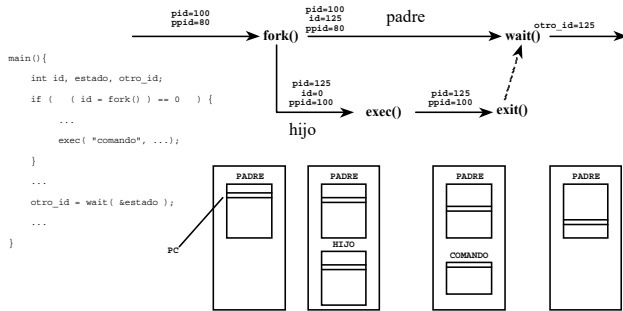
- Proceso hijo termina y padre está esperando en wait()
 - Padre recibe información sobre la causa de la muerte del hijo y continúa ejecutando
 - Hijo desaparece

- 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



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("\n\tSoy el padre: %d\n", getpid() );
    printf("\n\tFork 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>

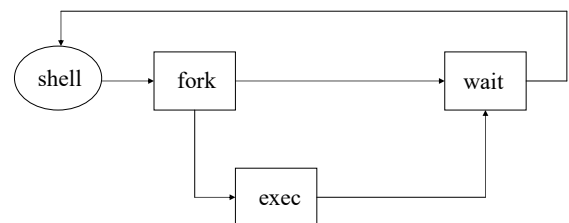
int main() {
    int pid;
    int estado;
    switch(fork()) {
        case -1: perror("fork"); exit(1); /* error */
        case 0 : /* hijo */
            sleep(2);
            printf("Soy el hijo %d y me muero...\n",getpid());
            exit(15);
        default: /*padre*/
            sleep(10); /* 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 l-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)

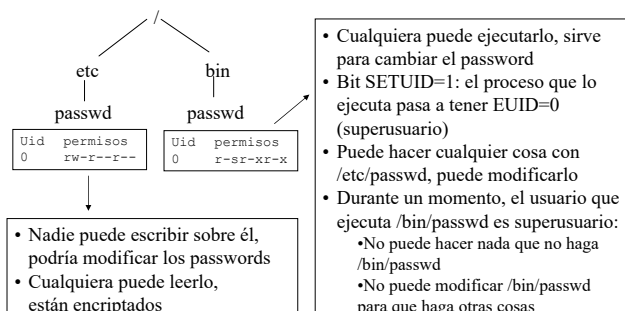
Resumen de Identificadores y Marcas (2)

- Para cada proceso
 - PID: identificador de proceso - getpid()
 - Process Group ID: Id. de grupo de procesos getpgid()
 - setpgid() PGID=PID (proceso líder)
 - setpgid(pid, pgid)
 - 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



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 execl 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 execl 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 *argv[BUFSIZE];
    int i;
    for(;;){ /* bucle infinito */
        fprintf( stderr, "\n$ " );
        if (gets(s)==NULL) {
            printf("logout\n");
            exit(0);
        }
        argv[0] = strtok( s, " " ); /* argv[0] comando a ejecutar */
        if( 0 == strcmp( argv[0], "quit" )){ /*cierto si argv[0]="quit" */
            printf("logout\n");
            exit( 0 );
        }
        for( i = 1; (argv[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( argv[0], &argv[0]);
        fprintf(stderr, "\nNo se puede ejecutar %s\n", argv[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 *argv[BUFSIZE];
    int i, parate, pid;
```



Ejecución asíncrona: ej1001.c (2)

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

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

    if( 0 == strcmp( argv[0], "soo" ) )
        parate = TRUE;
    else
        if( 0 == strcmp( argv[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
- Características de la solución
- Solución monoprocesador
- Soporte hw a la solución: swap y test&set
- Mutex
- Semáforos

Problema

Acceso concurrente a variable compartida

r0=@i i=0

Proceso 1: i=i+1

Proceso 2: i=i-1

Proceso 1

Proceso 2

```

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

```

```

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

NO

OK

NO

tiempo

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, \dots, 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)



Universidad de Zaragoza

5



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



Universidad de Zaragoza

7



Soporte hw a la exclusión mutua

- Instrucciones habituales
 - Swap:** Intercambio **atómico** $\text{var} \leftrightarrow \text{var}$ o $\text{reg} \leftrightarrow \text{var}$
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** $\text{reg} \leftrightarrow \text{mem}$ 32 bits
`swp r0, r0, [r1]` ; intercambio atómico $r0 \leftrightarrow \text{Mem32}[r1]$
 - swpb:** Intercambio **atómico** $\text{reg} \leftrightarrow \text{mem}$ 8 bits
`swpb r0, r0, [r1]` ; intercambio atómico $r0[7:0] \leftrightarrow \text{Mem8}[r1]$



Universidad de Zaragoza

9



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 ($i \neq j$) pueden entrar en sus SCs antes de que lo haga P_i

Exclusión mutua → proporcionada por HW

Progreso + espera acotada → proporcionada por SW



Universidad de Zaragoza

6



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



Universidad de Zaragoza

8



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



Universidad de Zaragoza

10



Universidad de Zaragoza

10

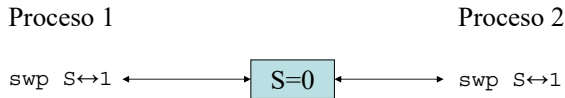


Mutex bajo nivel (ARM v4)

```

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



Universidad de Zaragoza

11



Semáforos

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

Proceso 1	Proceso 2
wait(S); //SC signal(S);	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**



Universidad de Zaragoza

12



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



Universidad de Zaragoza

13



Semáforos

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



Universidad de Zaragoza

14



Sistemas Operativos

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



Universidad de Zaragoza



Universidad de Zaragoza



Características

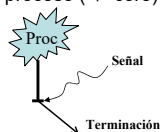
- Comunican eventos especiales a procesos en ejecución
 - Son interrupciones software
 - No contienen información. El proceso señalado no conoce la identidad del proceso señalador
 - No es la forma habitual de comunicar procesos. Pueden servir para sincronizar

¿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)

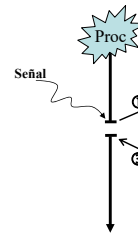
- El proceso señalado puede:
 - ① No querer manifestar su disposición frente a la señal
actúa el comportamiento por **defecto**
normalmente terminación del proceso (+/- core)



- 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,
SIGQUIT,
14, ...

SIG_DFL=0
SIG_IGN=1
rutina_captura

① defecto
② ignorar
③ capturar
- Devolución: anterior comportamiento ó -1 si error

```
#define SIG_ERR (void (*)(int)) -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 = \emptyset
 - 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

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

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

Solución?



Universidad de Zaragoza



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);  
}
```

CUIDADO!!

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



Universidad de Zaragoza



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



Universidad de Zaragoza



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:

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

si llega señal => ¡el proceso acaba!

no siempre funciona!



Universidad de Zaragoza



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");  
}
```



Universidad de Zaragoza



Problemas con las señales (2): pelosisig.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"

SOLUCIÓN?



Universidad de Zaragoza



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 leído\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	NO	SI
<i>para qué?</i> – ejercicio pelosisig.c		
• Desprotección del proceso en la rutina de captura	SI	opcional
<i>por qué?</i> – reset al DFT en captura		
• Posibilidad de <i>restart</i> 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)	Llamar siempre a sigemptyset o sigfillset para asegurarse correcta inicialización de sigset
int sigfillset (sigset_t *set)	
int sigaddset (sigset_t *set, int signo)	Funciones sobre sets (tipo sigset_t definido en signal.h)
int sigdelset (sigset_t *set, int signo)	
int sigismember (sigset_t *set, int signo)	



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 cjo. señales bloqueadas
SIG_UNBLOCK quita *set del cjo. señales bloq.
SIG_SETMASK pone nuevo cjo. señales bloq.
si oset != NULL devuelve en *oset el cjo. 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:
}

SA_RESTART (4)      reinicio automático de SC      defecto: 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

- 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)
- Bloqueo de señales imposible con no seguras
 - Es posible con sigprocmask**
- 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");
}
```



Sistemas Operativos

Gestión de la Entrada/Salida

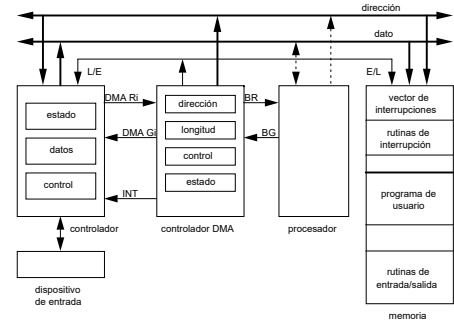
[SGG]: Cap. 10 al 13
[Sta]: Cap 10 a 12
[Ste05]: Cap. 3 a 5



Gestión de la entrada/salida

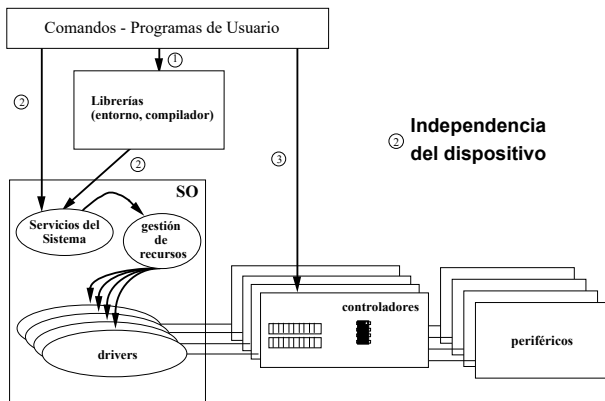
- **Hardware** de e/s. Recordatorio
- **Software** de la e/s. *Drivers*
- Tipos de dispositivos
- *Disco*: directorio, fichero, asignación de espacio
- Sistema de Ficheros UNIX
- UNIX: Estructura de un disco
- Tablas en memoria del Sistema de Ficheros

Fundamentos del *hardware* de e/s

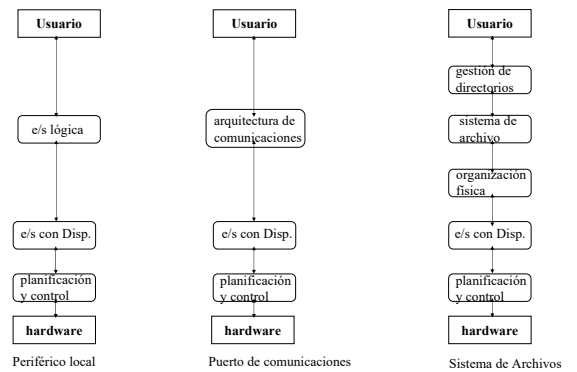


- Sincronización por INTerrupciones
- Transferencia por Direct Memory Access

Análisis del *software* de la e/s



Software de la e/s (2 de 2)



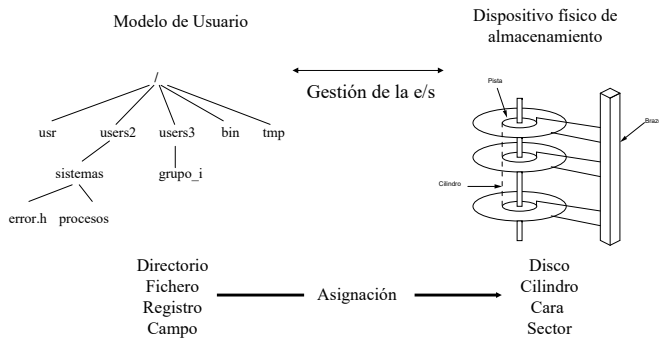
Drivers

- Dispositivos lógicos mantenidos por el SO
- *Software* que trabaja con la e/s dependiente del dispositivo
- Un *driver* por cada dispositivo o por una clase si están muy relacionados
 - UNIX: número mayor, número menor
- Emite los comandos a los registros del K_D
- Verifica que se ejecuten bien
- Conoce todos los detalles sobre el D y el K_D
 - @ de registros del K_D , utilidad de los registros, mecánica del D, sectores, pistas...

Tipos de Dispositivos

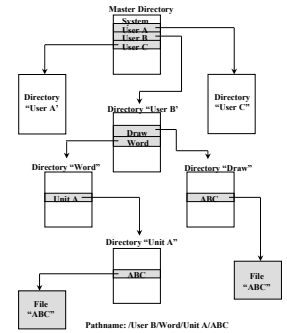
- De bloque. *Discos*
 - direccionable (bloque), con operaciones de localización
- De caracteres. *Terminales, impresoras, ratón...*
 - no direccionables, sin operaciones de localización

Abstracción del disco



Directorio: Espacio de nombres

- Un nivel
 - Todos los ficheros en un único directorio
- Dos niveles
 - Un directorio por usuario
- Jerarquía
 - Cada usuario puede organizar sus ficheros en subdirectorios



Fichero

- Unidad lógica de almacenamiento secundario
 - Tipo abstracto de dato def. e implementado por el S.O.
 - Colección de información relacionada
 - Para el usuario es la unidad mínima de almacen. secund.
- Atributos :
 - nombre, identificador, tipo, propietario, protección, tamaño, fechas, localización, etc.
- Operaciones:
 - crear, borrar, leer, escribir, truncar, reposicionar
- Tipos de acceso a la información:
 - secuencial, directo, indexado, hash, ...

Fichero: operaciones

- Operaciones vs. Tipo de acceso
 - Secuencial:
 - rebobinar, leer_siguiente, escribir_siguiente
 - Directo:
 - leer(n), escribir(n), posicionar(n)
 - leer_siguiente, escribir_siguiente
 - Indexado, Hash
 - Leer(clave), escribir(clave), posicionar(clave)
 - leer_siguiente, escribir_siguiente

Asignación de espacio

- A cada fichero se le debe asignar espacio físico en un dispositivo
- El sistema debe guardar la información necesaria para recuperar el contenido del fichero
- El sistema debe controlar el espacio libre de cada dispositivo
- El espacio se asigna como una o varias porciones

Tamaño de las porciones

- Fijo/Variable
 - Variable minimiza el espacio perdido
 - Fijo simplifica la asignación de espacio
 - Se usa fijo: BLOQUE
- Grande/Pequeño
 - Grande minimiza tiempo de acceso
 - Grande minimiza información de acceso
 - Pequeño minimiza el espacio perdido

Métodos de Asignación

- Asignación contigua
- Asignación encadenada
- Asignación indexada

Asignación contigua

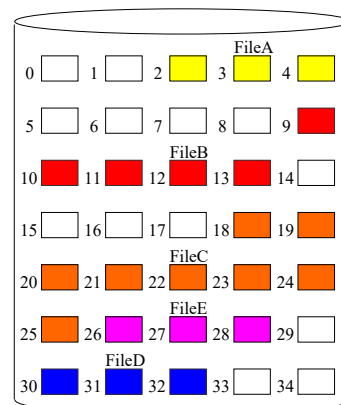


Tabla de localización

File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	30	3
FileE	26	3

- Fácil localización: inicio y tamaño
- Fácil acceso secuencial y directo
- Difícil asignación
- Pérdida de espacio
- Problemas con cambios de tamaño



Asignación encadenada

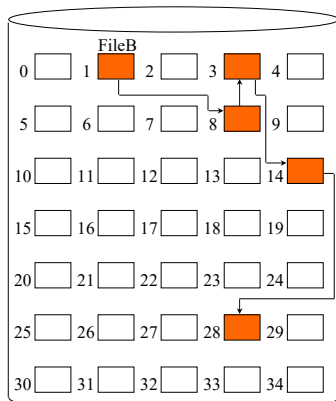


Tabla de localización

File Name	Start Block	Length
...
FileB	1	5
...

- Localización: inicio y tamaño
- Acceso secuencial
- Imposible acceso directo
- Fácil asignación
- Sin pérdida de espacio

La información se puede centralizar
Modelo DOS: FAT



Asignación indexada

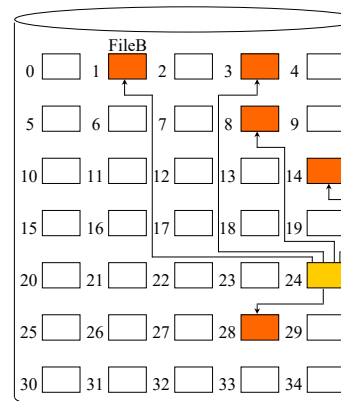


Tabla de localización

File Name	Index Block
...	...
FileB	24
...	...

- Localización: índice
- Acceso secuencial
- Acceso directo
- Fácil asignación
- Sin pérdida de espacio

Problema: tamaño del índice



Sistema de ficheros UNIX

- Ficheros sin estructura
 - Son secuencias de bytes
- Tipos de ficheros
 - Ordinario: definido por el usuario
 - Directorio: contiene una lista de parejas nombres de fichero -- Inodo
 - Dispositivo: usado para el acceso a dispositivos de entrada/salida
 - FIFO, pipes o "named pipes": Ficheros para comunicación entre procesos

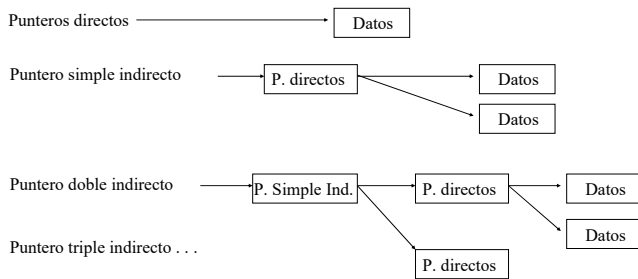
UNIX: I-node

- Estructura de datos que almacena toda la información relativa a un fichero
 - Número de I-node
 - Tipo fichero
 - Protección
 - UID, GID
 - Tamaño
 - Fecha último cambio
 - Número de links
 - Información sobre localización de los bloques del fichero



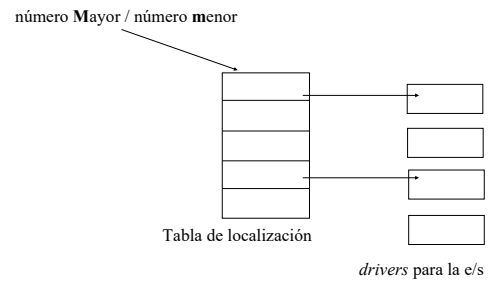
UNIX: I-node, localización

- Asignación indexada con varios niveles

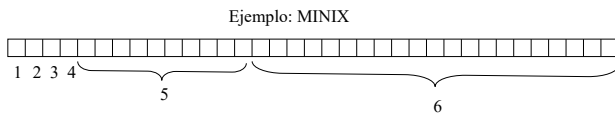


UNIX: I-node, localización

- Tipo de fichero Dispositivo



UNIX: Estructura de un disco



- 1. Bloque de arranque
- 2. Superbloque
- 3. Mapa de bits de nodos-l
- 4. Mapa de bits de bloques (o zonas)
- 5. Nodos-l
- 6. Bloques de datos

Ejercicio: asignación en UNIX

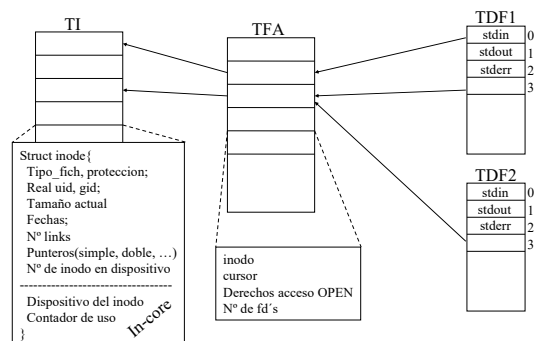
- Recuerda:
 - Estructura de directorio: árbol
 - Directorio: fichero con parejas <nombre, i-nodo>
 - Localización de la información de un fichero
 - I-nodo: índice con varios niveles
 - Estructura de un disco
 - Tabla de I-nodos y bloques de datos
- Ejercicio: ¿ Cuantos bloques de disco se han de leer para encontrar un byte de un fichero? Por ej.
 - Byte: 450
 - Fichero: /users3/sistemas/error.h

Tablas en memoria del Sistema de Ficheros

- Guardan información de los ficheros en uso
 - Razones: eficiencia, implementación del cursor, tipos de acceso
- El usuario debe avisar al sistema que desea trabajar con un fichero: llamada `open()`
- El usuario debe avisar cuando deja de trabajar con un fichero: llamada `close()`
- Tablas en memoria: 2 tablas globales
 - I-nodos de todos los ficheros en uso (los abiertos) o los recientemente usados ...
 - Para el acceso secuencial se necesita un cursor. Si queremos que varios procesos trabajen al mismo tiempo con un fichero:
 - varios cursores, distintos permisos de acceso
- Tablas en memoria: 1 tabla por proceso que contiene punteros a las entradas de TFA que está usando

TDF: Tabla de Descriptores de Fichero

Tablas en memoria del Sistema de Ficheros (2)



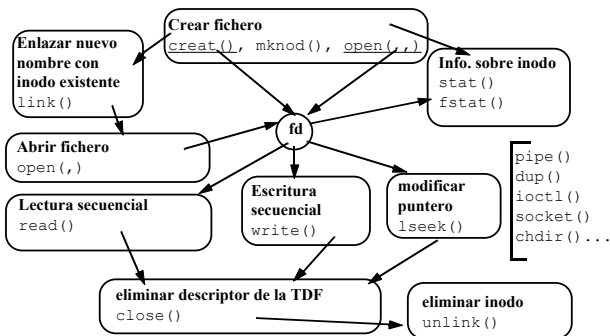
Ficheros: Llamadas al Sistema

- Esquema de llamadas sobre ficheros
- Manejo de ficheros existentes: `creat()`, `open()`, `close()`, `read()`, `write()`, `lseek()`. Ejemplos
- Llamada `mknod()`
- Información sobre ficheros: `stat()`, `fstat()`
- Llamadas `link()` y `unlink()`
- E/S standard vs. llamadas al sistema
- `forkfiles.c`

[Ste05]: cap. 3, 4, 5



Esquema de llamadas sobre ficheros



Universidad de Zaragoza

2

**creat()**

- Sintaxis:** `# include <fcntl.h>`
`int creat (char * path, mode_t mode)`
- Acción:** crea un fichero nuevo o reescribe uno existente
El fichero queda abierto sólo para escritura
path nombre del fichero a crear (absoluto o relativo)
Si *path* no existe: asigna/inicializa i-nodo
crea entrada en TFA (cursor al principio)
nueva entrada directorio `<path,i-nodo>`
Si *path* existe: trunca el fichero
mode permisos de acceso del fichero (según umask)
Devuelve: primer fd libre en TDF (apunta a TFA) ó -1 (error)



Universidad de Zaragoza

3

**open()**

- Sintaxis:** `# include <fcntl.h>`
`int open (char * path, int oflag)`
- Acción:** abre un fichero existente
path nombre del fichero a abrir (absoluto o relativo)
oflag modo de apertura del fichero:
`O_RDONLY, O_WRONLY, O_RDWR` (0,1,2)
sino modo de apertura indefinido
+ OR con `O_APPEND` (cursor final antes de escribir), `O_CREAT...`
se identifica el i-nodo
se crea entrada en TFA (cursor al principio, *oflag*)
- Devuelve:** primer fd libre en TDF (apunta a TFA) ó -1 (error)
`creat(pathname,mode) = open(pathname,O_WRONLY | O_CREAT | O_TRUNC,mode)`
Si queremos crear fichero pero poderlo leer también
`open(pathname,O_RDWR | O_CREAT | O_TRUNC,mode)`

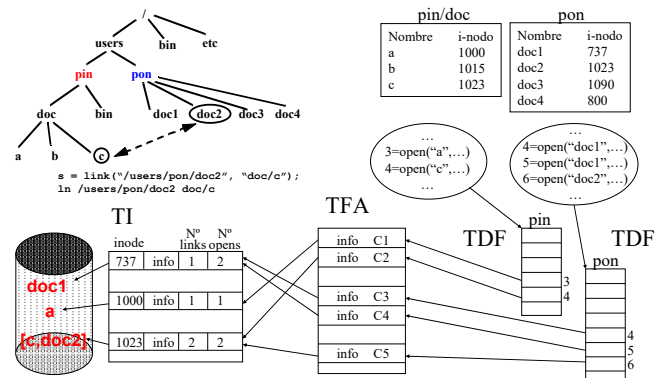


Universidad de Zaragoza

4



Tablas en memoria del Sistema de Ficheros (3)



Universidad de Zaragoza

5



close()

- Sintaxis:** `# include <unistd.h>`
`int close (int fildes)`
- Acción:** cierra el descriptor de fichero *fildes*

queda libre la entrada *fildes* de TDF
decrementa *nfilides* en TFA
`si (nfilides == 0)`
queda libre entrada en TFA
decrementa *nopens* en TI
`si (nopens == 0)`
`si (nlinks == 0)` borra fichero;
elimina inodo de TI;
- Devuelve:** 0 si bien ó -1 (errno)
- Cuando termina un proceso el kernel cierra todos los ficheros de su TDF y actualiza tablas TFA y TI



/* ej42.c */

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#define DIM 300

main( argc, argv )
int argc; char *argv[];
{
    int fd[DIM], contador, i;

    if( (fd[0] = creat( argv[1], 0777 )) == -1 ){
        perror( "creat" ); exit(1);
    }
    printf( "Abro el %d\n", fd[0] );
    for( i = 1; ( fd[i] = open( argv[1], O_RDONLY )) != -1 ; i++ )
        printf( "Abro el %d\n", fd[i] );
    contador=i;
    if( errno == EMFILE )
        printf( "\tTotal fich. abiertos: %d\n", (contador+3) );
    else{ perror( "open" ); exit(1); }
    for( i=0; i<contador; ++i)
        close( fd[i] );
    unlink( argv[1] );
}
```

Crea tantos descriptors de fichero como permite el sistema (OPEN_MAX=256)

EMFILE: agotada TDF (ENFILE: agotada TFA)



lseek()

- Sintaxis:** `# include <unistd.h>`
`off_t lseek (int fildes, off_t offset, int whence)`
- Acción:** mueve el cursor de lectura/escritura de un fichero
se modifica la posición del cursor *offset* bytes
hacia delante o hacia detrás (según signo de *offset*)
desde un punto de referencia (*whence*)
whence posición desde la que se aplica el offset
`SEEK_SET / L_SET / 0` principio fichero (*offset* >= 0)
`SEEK_CUR / L_CUR / 1` posición actual
`SEEK_END / L_END / 2` posición final
- Devuelve:** nueva posición del cursor respecto del principio
-1 si error (errno)
- Ejemplo:** Para saber la posición actual del cursor:
`off_t cursor;`
`cursor = lseek(fd, 0, SEEK_CUR);`



read()

- Sintaxis:** `# include <unistd.h>`
`ssize_t read (int fildes, void *buf, size_t nbyte)`
- Acción:** lee *nbyte* bytes del fichero asociado al descriptor *filides* y los almacena en *buf*
la lectura comienza en la posición indicada por el cursor (en TFA)
se modifica la posición del cursor con el número de bytes realmente leídos
- Devuelve:** número de bytes realmente leídos
0 si ya no hay más bytes que leer
-1 si error (errno)



write()

- Sintaxis:** `# include <unistd.h>`
`ssize_t write (int filides, const void *buf, size_t nbyte)`
- Acción:** escribe *nbyte* bytes del buffer *buf* en el fichero asociado al descriptor *filides*
la escritura comienza en la posición indicada por el cursor (en TFA)
se modifica la posición del cursor con el número de bytes realmente escritos
se actualiza el tamaño del fichero en i-nodo
- Devuelve:** número de bytes realmente escritos
-1 si error (errno)



/* ej10.c */

Copia el contenido de fich1 en fich2

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

main( argc, argv )
int argc; char *argv[];
{
    int fdfnt, fddst;
    void copia();

    if( argc != 3 ){
        printf( "Uso: %s fich1 fich2 ", argv[0] ); exit( 1 ); }

    if ( (fdfnt = open( argv[1], O_RDONLY )) == -1 )
        { fprintf( stderr, "\tError open\n" ); exit( 1 ); }

    if ( (fddst = creat( argv[2], 0666 )) == -1 )
        { fprintf( stderr, "\tError creat\n" ); exit( 1 ); }

    copia( fdfnt, fddst );
    exit( 0 );
}

void copia ( fnt, dst )
int fnt, dst;
{
    int cuenta;
    char buf[BUFSIZ];

    while ( (cuenta = read( fnt, buf, sizeof( buf ) )) > 0 )
        write( dst, buf, cuenta );
}
```

Probar:
• ej10 fich_existe fich_nuevo
• ej10 ej10.c nuevo.c
• ej10 ej10 nuevo
• ej10 . direct
• ej10 fich_existe /dev/tty

en <stdio.h>



/* reverse.c */

```
#include <stdio.h>
#include <fcntl.h>
#include "error.h"

main(argc,argv)
{
    char c;
    int i, fdnt;
    long where;

    if(argc != 2){ printf( "Uso: %s fichero_a_invertir" argv[0]; exit(1); }

    if((fdnt = open( argv[1], O_RDONLY )) == -1) syserr("open");
    if((where = lseek( fdnt, -1L, SEEK_END )) == -1) syserr("lseek");

    while(where >= 0){
        read(fdnt, &c, 1);
        write(1, &c, 1);
        where = lseek( fdnt, -2L, SEEK_CUR );
    }
}
```

Invierte el contenido de un fichero

formato long



Universidad de Zaragoza

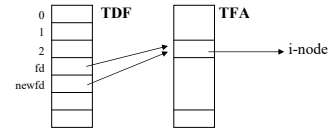
12



dup()

- Sintaxis:** `#include <unistd.h>`
`int dup(int fd);`
- Acción:** duplica `fd` con un nuevo descriptor (`newfd`)
`fd` es un descriptor ya asignado con `open`, `creat`, `pipe`, `dup`... => apunta a TFA
tras `dup`, `fd` y `newfd` apuntan a la misma entrada en TFA
- Devuelve:** el siguiente descriptor libre en TDF ó -1 si error

`newfd = dup(fd);`



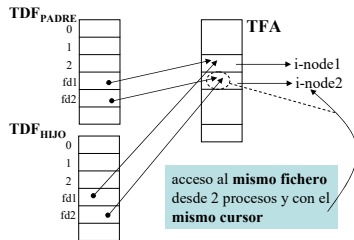
Universidad de Zaragoza

13



Tablas vs fork() y exec()

- Tras hacer un **fork()** el proceso HIJO recibe una copia de la Tabla de Descriptores de Fichero del proceso PADRE:
=> - la TDF se *duplica* entera
- TODAS las entradas de la TDF_{HIJO} apuntan a la misma entrada en TFA que la correspondiente entrada de la TDF_{PADRE}



- Tras hacer un **exec()** el proceso *ejecutado* mantiene la TDF del proceso *ejecutor*

acceso al mismo fichero desde 2 procesos y con el mismo cursor



Universidad de Zaragoza

14



/* forkfiles.c */

```
#include <fcntl.h>
#include <stdio.h>
#include "error.h"

int sfd, tfd;
char c;

main(argc, argv)
{
    int argc; char **argv;

    if( argc != 3 ) exit( 1 );
    if( (sfd = open( argv[1], O_RDONLY )) == -1 ) syserr( "open" );
    if( (tfd = creat( argv[2], 0777 )) == -1 ) syserr( "creat" );

    if( fork() == -1 ) syserr( "fork" );
    copy();
    exit( 0 );
}

copy()
{
    for(;;){
        if( read( sfd, &c, 1 ) != 1 ) return;
        write( tfd, &c, 1 );
    }
}
```

Padre e Hijo acceden a los mismos ficheros a través del cursor



Universidad de Zaragoza

15



stat(), fstat()

- Sintaxis:** `#include <sys/stat.h>`
`int stat (char *path, struct stat *buf)`
`int fstat (int fd, struct stat *buf)`
- Acción:** recogen información del fichero asociado al nombre `path` (`stat`) o al descriptor `fd` (`fstat`)
se lee el i-nodo para rellenar la estructura `buf`

fichero abierto

```
struct stat {
    dev_t st_dev; /* ID of dev containing a directory entry for this file */
    ino_t st_ino; /* Inode number */
    mode_t st_mode; /* File type & Permission bits */
    nlink_t st_nlink; /* Number of links */
    uid_t st_uid; /* User ID of file owner */
    gid_t st_gid; /* Group ID of file group */
    dev_t st_rdev; /* mayor/minor IDs; defined only for char or blk spec files */
    off_t st_size; /* File size (bytes) */
    time_t st_atime; /* Time of last access */
    time_t st_mtime; /* Last modification time */
    time_t st_ctime; /* Last file status change time */
    blksize_t st_blksize; /* best I/O block size (8192) */
    ... /* hay mas campos */
};
```

- Devuelve:** 0 si bien ó -1 (erro)



Universidad de Zaragoza

16



link()

- Sintaxis:** `#include <unistd.h>`
`int link (char *path1, char *path2)`
- Acción:** crea para `path2` una nueva entrada en el directorio con el mismo i-nodo de `path1`:
`<path1, i-nodo i>` `<path2, i-nodo i>`
(número de links del i-nodo i)++;
- Devuelve:** 0 si bien ó -1 (erro)

fichero existente

fichero nuevo



Universidad de Zaragoza

17



unlink()

- **Sintaxis:** `# include <unistd.h>`
`int unlink (char *path)`
- **Acción:** elimina la entrada en el directorio del fichero *path*:
~~<path, i-nodo>~~
`nlinks = nlinks-1 (en i-nodo);`
`si (nlinks == 0) {`
 `si (nopens (en TI) == 0)`
 `borra fichero;`
 `si el borrado se hará en el último close;`
}
- **Devuelve:** 0 si bien ó -1 (erro)



Universidad de Zaragoza

18



mknod()

- **Sintaxis:** `# include <sys/stat.h>`
`int mknod (char * path, mode_t mode, dev_t dev)`
- **Acción:** crea tipos de ficheros especiales (directorios, dispositivos, FIFOs...) uso restringido al super-usuario excepto en FIFOs
- path* nombre del fichero a crear
- mode* tipo y permisos de acceso del fichero (+ umask)
constantes simbólicas definidas en *stat.h*:
 `S_IFDIR, S_IFBLK, S_IFCHR, S_IFIFO ...`
 `S_IRWXU, S_IRWXG, S_IRWXO ...`
- dev* solo para ficheros dispositivos (caracteres, bloques)
 número mayor ← y → número menor (ls -l /dev)
 clase de dispositivo ← drivers → número dentro de una clase
- **Devuelve:** 0 si bien ó -1 si error (erro).

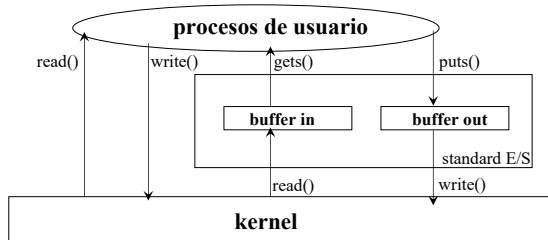


Universidad de Zaragoza

19



E/S estándar vs. llamadas al sistema



- Las funciones de biblioteca agrupan la E/S (menos SC's)
 - Fully buffered: E/S real se realiza cuando el buffer se llena o se ejecuta fflush
 - Line buffered: E/S real cuando llega caracter fin de línea ('\n' o ASCII 10)
Pensado para dispositivos de líneas (terminales, ...)
 - Unbuffered: E/S real sin uso de buffers



Universidad de Zaragoza

20



E/S estándar vs. llamadas al sistema

- Por defecto
 - stdin y stdout son:
 - Line buffered si están dirigidas a una terminal o
 - Fully buffered en caso contrario
 - stderr es unbuffered
- Se puede modificar el comportamiento de las funciones de biblioteca con setbuf() o setvbuf()
 - Cambiar tamaño de los buffers (por defecto BUFSIZ=1024)
 - fully buffered (_IOFBF)
 - line buffered (_IOLBF)
 - unbuffered (_IONBF)



Universidad de Zaragoza

21



E/S estándar vs. llamadas al sistema

Llamadas al sistema	Funciones de biblioteca C
<pre> int fd; /*file descriptor*/ fd = open(nombre,...); read(fd, ...); write(fd, ...); lseek(fd, ...); ... close(fd); </pre>	<pre> FILE *fp; /*file pointer*/ fp = fopen(nombre,...); fgets(fp, ...); fprintf(fp, ...); fseek(fp, ...); ... fclose(fp); </pre>



Universidad de Zaragoza

22



E/S estándar vs. llamadas al sistema

Llamadas al sistema	Funciones de biblioteca C
<pre> Entrada/salida estandar 0=STDIN_FILENO 1=STDOUT_FILENO 2=STDERR_FILENO son file descriptor read(0, ...); write(1, ...); write(2, ...); ... close(1); </pre>	<pre> Entrada/salida estandar stdin stdout stderr son file pointer fgets(stdin, ...); fprintf(stdout, ...); fprintf(stderr, ...); ... fclose(stdout); </pre>



Universidad de Zaragoza

23



E/S estándar vs. llamadas al sistema

- Qué ocurre con los buffers cuando:
 - **fork:** se copia el Buffer_{PADRE} al Buffer_{HIJO}
(si había cosas pendientes de escribir, se escriben 2 veces)
 - **exec:** desaparece el Buffer
(si había cosas pendientes de escribir, se pierden)
 - **exit:** se vacía el Buffer (equivalente a fflush)
- **terminación involuntaria:**
desaparece el Buffer
(si había cosas pendientes de escribir, se pierden)



Ejemplo: ficheros, E/S estándar, redirecciones

```
#include<stdio.h>
main() {
    int id,estado,fd;

    printf("linea de texto n 1\n");
    if((id=fork())==0) {
        close(1);
        creat("salida.dat",0777);
        write(1,"linea de texto n 2\n",19);
        exit(1);
    }
    else {
        while(wait(&estado)!=id);
        write(1,"linea de texto n 3\n",19);
        exit(0);
    }
}
```



Sistemas Operativos

Redireccionamiento



Redireccionamiento

- Procesos FILTRO
- Redirección entrada/salida/errores
- Ejemplos
- ¿Cómo hace el *Shell* para redireccionar?



Procesos FILTRO



FILTRO: Proceso que LEE datos de la entrada estándar (*stdin*), los TRANSFORMA y ESCRIBE los datos transformados en la salida estándar (*stdout*, *stderr*)



Esa convención inicial se puede cambiar REDIRECCIONANDO



Redirección

- Desde el intérprete de comandos:
 - Redirección de la salida:
who > quien_hay.tmp
 - Redirección de la entrada:
sort < quien_hay.tmp
 - Redirección de la salida de errores:
cc -o ej10 ej10.c 2> error_10
 - Redirección de la entrada y la salida:
sort < quien_hay.tmp > quien_hay



/* ej110.c */

```
#include "error.h"
#include <fcntl.h>
#include <stdio.h>

main() {
    int fd;
    if ( fd = creat( "quien_hay.tmp", 0640 ) == -1 )
        if ( errno == EACCES ) {
            printf( "fichero existe sin permiso escritura\n" );
            exit( 0 );
        }
        else syserr( "creat" );
    close( fd );
    close( 1 );
    fd = open( "quien_hay.tmp", O_WRONLY );
    if ( fd == 1 )
        fprintf( stderr, "Salida Redireccionada\n" );
    else {
        fprintf( stderr, "Virus\n" );
        exit( 1 );
    }
    execl( "/bin/who", "who", 0 );
    syserr( "execl" );
}
```

who > quien_hay.tmp

se puede mejorar?



Universidad de Zaragoza

4



Ejercicios

- ej111.c:
Programa que realice las siguientes redirecciones:
sort < quien_hay.tmp > quien_hay
(quien_hay.tmp es el fichero generado en ej110.c)
- ej112.c:
Programa que ejecute el comando:
date >> quien_hay
(recordar: >> añade la salida del comando date al final del fichero quien_hay. Usar open(... O_APPEND) o lseek())



Universidad de Zaragoza

5



¿Cómo hace el shell para redireccionar?

```
switch ( fork() )
{
    . . .
    case 0:          /* HIJO */
        if ( r_in )   redireccionar_entrada();
        if ( r_out )  redireccionar_salida();
        if ( r_out_app redireccionar_salida_append();

        exec( . . . . . );
    default:
        . . .
}
```

hereda TDF y por lo tanto las redirecciones...

La separación de fork() y exec() simplifica el tratamiento de señales y los redireccionamientos. Además, permite que el código quede perfectamente localizado



Universidad de Zaragoza

6



Sistemas Operativos

Comunicación entre Procesos



Universidad de Zaragoza

2



Comunicación entre Procesos

- Tuberías. Tipos
- PIPES. Llamadas asociadas
- Ejemplos de uso
- Creación de pipes desde el shell
- Más ejemplos

Tuberías (1 de 2)

Dos tipos:

- UNAMED PIPES O PIPES
- NAMED PIPES, FIFOS O NAMED FIFOS

CARACTERÍSTICAS COMUNES

- Unidireccionales (un proceso lee y el otro escribe)
- Implementación: i-nodo del que se utilizan los punteros directos
=> capacidad limitada y dependiente de la implementación:
mínimo 4096 bytes



[Ste94]: capítulo 15



Universidad de Zaragoza

1



Universidad de Zaragoza

2



Tuberías (2 de 2)

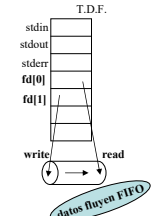
DIFERENCIAS

UNAMED	NAMED
comunican procesos emparentados	comunican cualquier proceso
No están representados en el Sistema de Ficheros mediante un nombre	Tienen nombre en el Sistema de Ficheros, como un fichero cualquiera
Se crean con pipe()	Se crean con mknod()
Muy utilizados, especialmente en el <i>shell</i> : who sort lp -oq	Muy poco utilizados



PIPES: Llamadas asociadas (1)

- Sintaxis:** `# include <unistd.h>`
`int pipe(int fd[2]);`
- Acción:** crea una pipe
asigna i-nodo
crea 2 entradas en TFA



en *fd* se almacenan los 2 primeros fd's libres:

- *fd*[0] abierto para lectura en pipe
- *fd*[1] abierto para escritura en pipe
- lo escrito en *fd*[1] se lee por *fd*[0]
- apuntan a las correspondientes entradas en TFA
- para manejar la pipe como si fuese un fichero normal (read, write)

Prob. clásico del productor-consumidor

- Devuelve:** 0 si bien ó -1 si error

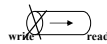
Buffer en memoria (más rápido)
16 KiB en *hendrix*



PIPES: Transmisión de datos

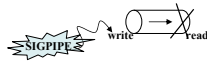
read()

- Se leen datos del extremo de lectura
- Si pipe vacía => bloquea
- Si pipe vacía y extremo de escritura cerrado => no bloquea y devuelve 0 (final de datos)



write()

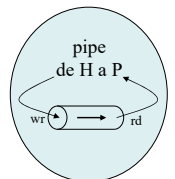
- Se escriben datos en el extremo de escritura
- Si pipe llena => bloquea
- Si no está abierto el extremo de lectura => señal SIGPIPE



Ejemplo de uso de pipes

```
#include "error.h"
main(argc,argv)
int argc;char *argv[];
{
    int fpipe[2], file, n; char buf[512];
    if(argc!=3) syserr("Numero de parametros");
    pipe(fpipe);
    switch(fork()) {
        case -1: syserr("fork");
        case 0: file=open(argv[1],0);
            while((n=read(file,buf,sizeof(buf)))!=0)
                write(fpipe[1],buf,n);
            printf("Fichero leído\n");
            break;
        default: close(fpipe[1]);
            file=creat(argv[2],0600);
            while((n=read(fpipe[0],buf,sizeof(buf)))!=0)
                write(file,buf,n);
            printf("Fichero copiado\n");
    }
    exit(0);
}
```

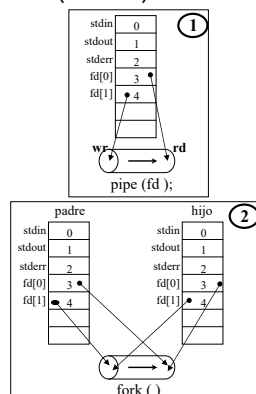
para copiar un fichero perdiendo el tiempo



Pipes y redirecciones (1 de 4)

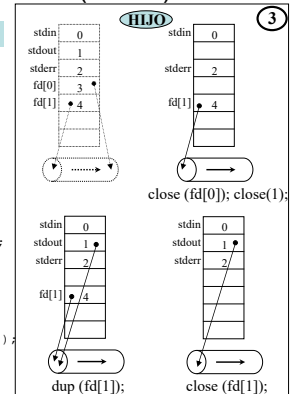
```
#include "error.h"
main()
{
    int fd[2];
    pipe(fd);
    switch(fork()) {
        case -1: syserr("fork");
        case 0: close(fd[0]);
            close(1);
            dup(fd[1]);
            close(fd[1]);
            execlp("who", "who", 0);
        default: close(fd[1]);
            close(0);
            dup(fd[0]);
            close(fd[0]);
            execlp("sort", "sort", 0);
    }
}
```

who | sort



Pipes y redirecciones (2 de 4)

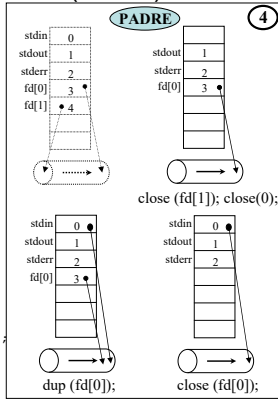
```
#include "error.h"
main()
{
    int fd[2];
    pipe(fd);
    switch(fork()) {
        case -1: syserr("fork");
        case 0: close(fd[0]);
            close(1);
            dup(fd[1]);
            close(fd[1]);
            execlp("who", "who", 0);
        default: close(fd[1]);
            close(0);
            dup(fd[0]);
            close(fd[0]);
            execlp("sort", "sort", 0);
    }
}
```



Pipes y redirecciones (3 de 4)

```
#include "error.h"
main()
{ int fd[2];

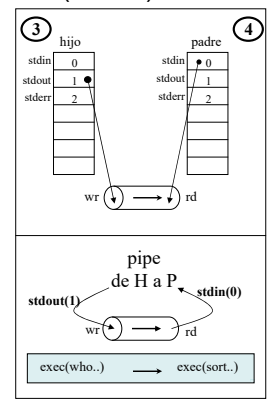
  pipe( fd );
  switch( fork() ) {
  case -1: syserr( "fork" );
  case 0: close( fd[0] );
          close( 1 );
          dup( fd[1] );
          close( fd[1] );
          execlp( "who", "who", 0 );
  default: close( fd[1] );
           close( 0 );
           dup( fd[0] );
           close( fd[0] );
           execlp( "sort", "sort", 0 );
  }
}
```



Pipes y redirecciones (4 de 4)

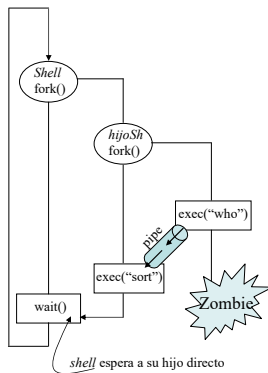
```
#include "error.h"
main()
{ int fd[2];

  pipe( fd );
  switch( fork() ) {
  case -1: syserr( "fork" );
  case 0: close( fd[0] );
          close( 1 );
          dup( fd[1] );
          close( fd[1] );
          execlp( "who", "who", 0 );
  default: close( fd[1] );
           close( 0 );
           dup( fd[0] );
           close( fd[0] );
           execlp( "sort", "sort", 0 );
  }
}
```



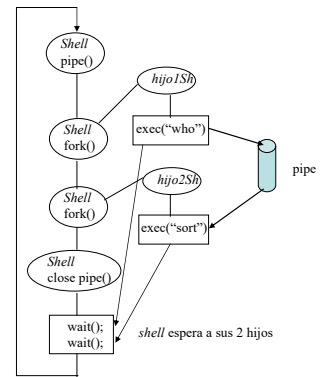
Creación de pipes desde el shell

```
for( ; ; ){
  lectura de comandos...;
  parsing...;
  switch( fork() ) {
  case -1: ...
  case 0: pipe( fd );
          switch( fork() ) {
          case -1: ...
          case 0: redirecciones;
                  exec ( "who" );
          default: redirecciones;
                  exec ( "sort" );
          }
  default: wait( estado );
  }
}
```



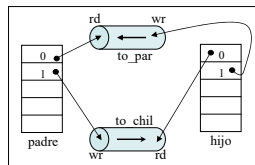
Creación de pipes desde el shell

```
for( ; ; ){
  lectura de comandos...;
  parsing...;
  pipe( fd );
  switch( fork() ) {
  case -1: ...
  case 0: redirecciones;
          exec("who");
  default: switch( fork() ) {
          case -1: ...
          case 0: redirecciones;
                  exec ( "sort" );
          }
  }
  close fd[0] fd[1];
  wait( estado ); /* 1er hijo */
  wait( estado ); /* 2º hijo */
}
```



Comunicación bidireccional con 2 pipes

```
#include <string.h>
char string[] = "hello world";
main()
{ int count, i, to_par[2], to_child[2];
  char buf[256];
  pipe(to_par); pipe(to_child);
  if( fork() == 0 ){
    close(0); dup(to_child[0]);
    close(1); dup(to_par[1]);
    close(to_par[1]); close(to_child[0]);
    close(to_par[0]); close(to_child[1]);
    for( ; ; ){
      if ( (count = read(0, buf, sizeof(buf))) == 0 )
        exit(1);
      write(1, buf, count);
    }
  }
  close(1); dup(to_child[1]);
  close(0); dup(to_par[0]);
  close(to_child[1]); close(to_par[0]);
  close(to_child[0]); close(to_par[1]);
  for( i = 0 ; i < 15 ; i++ ){
    write(1, buf, count);
    read(0, buf, sizeof(buf));
  }
}
```



Otro ejemplo

```
/* primer.c */
main() {
  int id, fd[2];
  id = fork();
  pipe(fd);
  switch ( id ) {
  case -1: exit(1);
  case 0: close(0);
          dup(fd[0]);
          close(fd[0]);
          close(fd[1]);
          execlp("segun", "segun", 0);
          exit(1);
  default: close(1);
           dup(fd[1]);
           close(fd[0]);
           close(fd[1]);
           execlp("segun", "segun", 0);
           exit(1);
  }
  exit(0);
}
```

```
/* segun.c */
main()
{ char c;

  while ( read(0, &c, 1) != 0 )
    write(2, &c, 1);
  write(1, &c, 1);
  exit(0);
}
```

No funciona

Gestión de Memoria

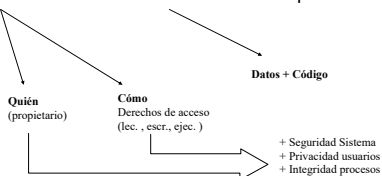
- Objetivos
- Clasificación
- Programas enteros y contiguos
 - Particiones de tamaño fijo
 - Particiones de tamaño variable
- Programas troceados
 - Segmentación
 - Paginación
- Memoria virtual paginada

[SGG]: capítulo 8-9



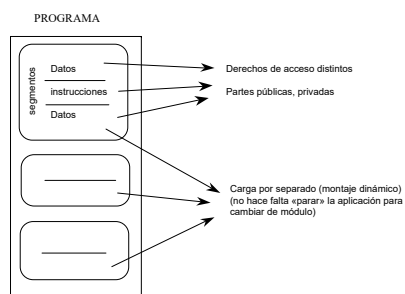
Objetivos de un sistema multiprogramado

- Mayor utilización del Procesador y de las E/S (+ procesos en estado **PREPARADO**)
- Permitir la **Comunicación y Sincronización** entre procesos
- Facilitar al S.O. la carga/movimiento de procs. en memoria
- **LIBERAR AL PROGRAMADOR/** ORA de detalles físicos (Tam. RAM etc)
- **PROTECCIÓN / COMPARTICIÓN** entre procesos o usuarios

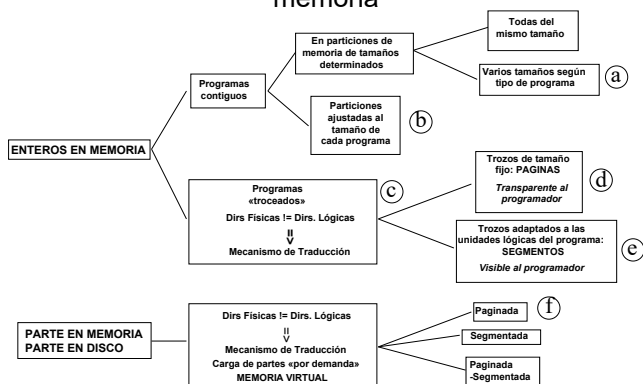


Objetivos de un sistema multiprogramado (2)

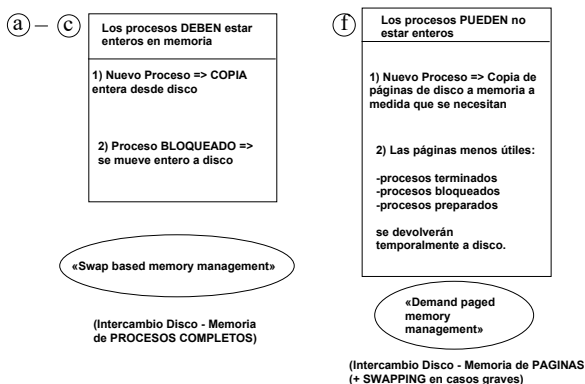
- **SOPORTAR CONCEPTOS** de la programación en alto nivel:



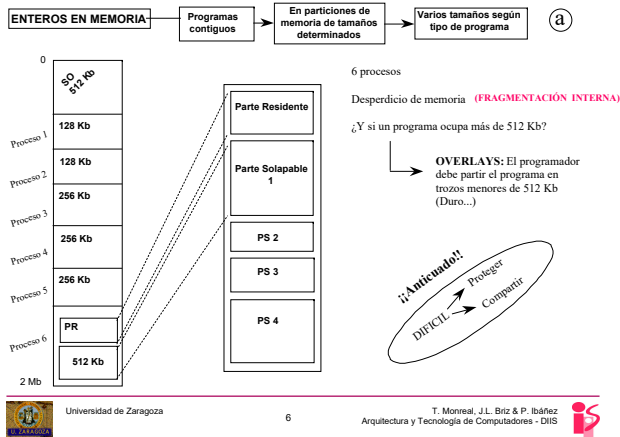
Cómo pueden residir los programas en memoria



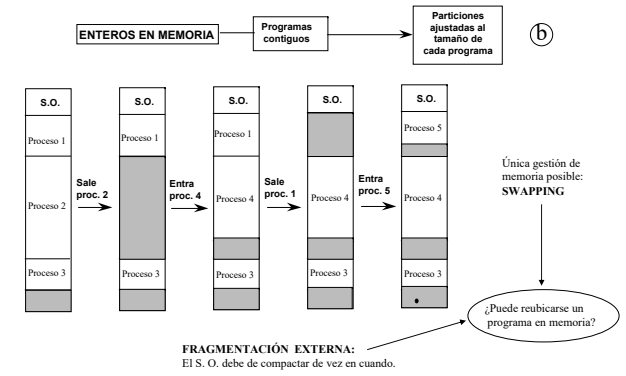
Dos posibilidades básicas



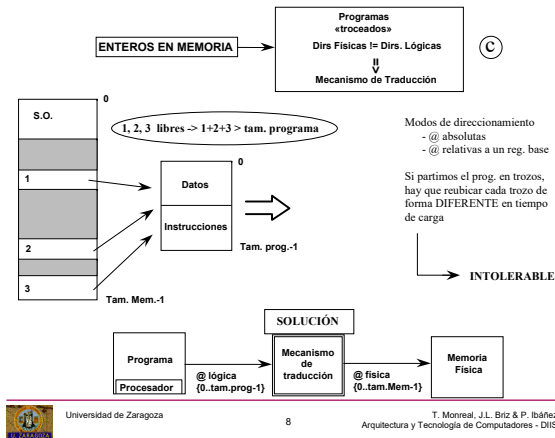
Particiones de tamaño fijo



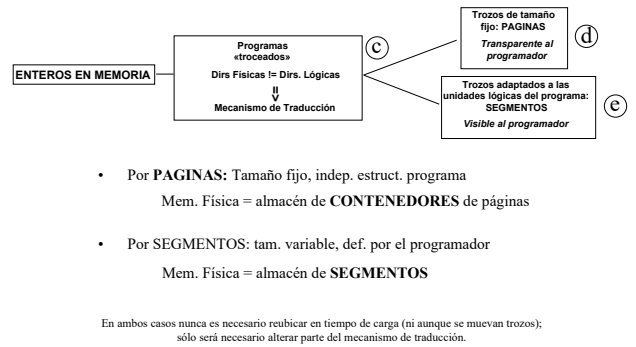
Particiones de tamaño variable



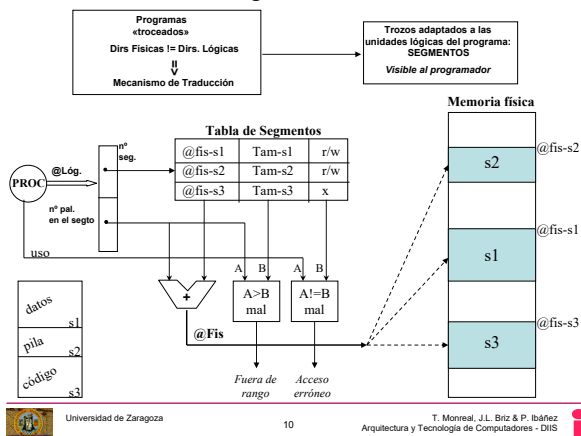
Programas troceados



Dos formas de partir un programa



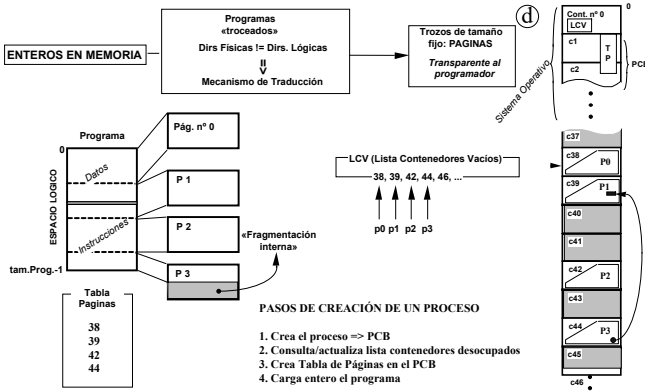
Segmentación



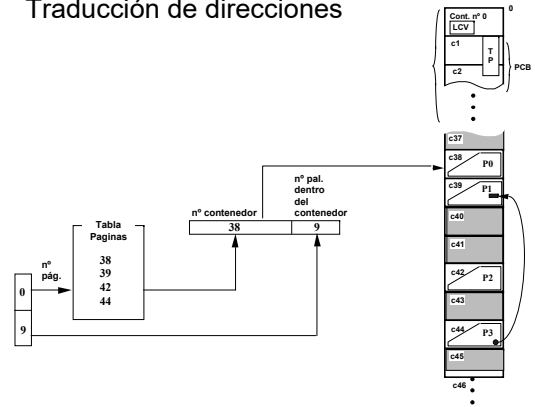
Segmentación: características

- Segmentos definidos por el programador
- Fragmentación Externa
- Crecimiento dinámico fácil (pila, heap)
- Cada segmento puede estar en un fichero distinto
- El espacio lógico está formado por DOS dimensiones, el programador «ve» las dos dimensiones:
 - nº de segmento.
 - nº de palabra dentro del segmento.
- Coste temporal:
 - 2 accesos a tabla (< l/e, tam_s1> ; < @Fis_s1>)
 - 1 SUMA (Inicios de Segmento NO ALINEADOS)
 - 1 acceso a @ Física

Paginación

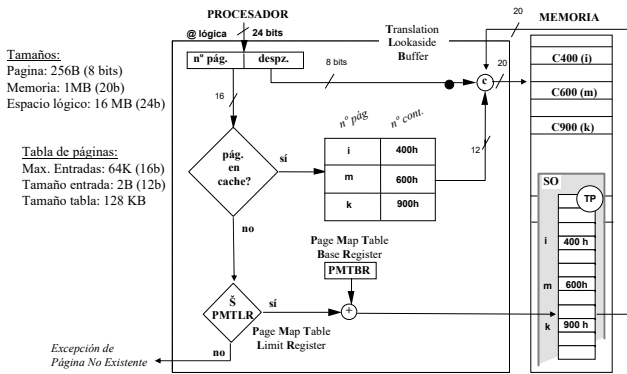


Paginación: Traducción de direcciones



Paginación: Traducción de direcciones (TLB)

Aceleración Hardware del Mecanismo de Traducción por Páginas



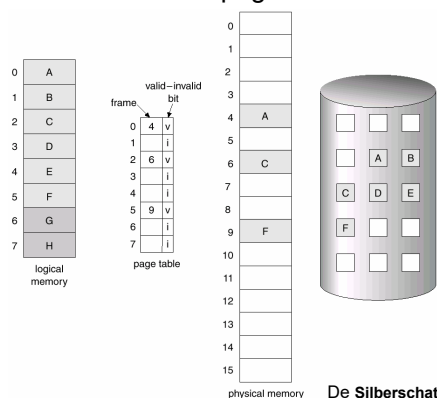
Paginación: protección y compartición

- Protección
 - Un proceso sólo "ve" a través de su tabla de traducción
 - Derechos de acceso en cada página (lec, escr, ejec)
 - (Almacenados en cada entrada de la tabla de páginas (TP))
- Compartición
 - 2 entradas, de 2 TP distintas, pueden apuntar al mismo contenedor físico
 - Compartición a nivel de página

Memoria virtual paginada

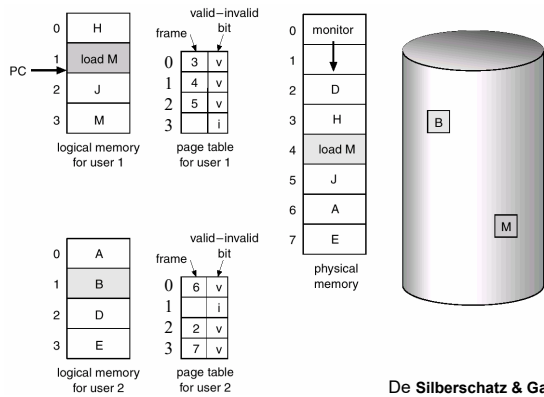
- Paginación: $EF > EL$; $EF > \text{suma}(ELi)$
- Alternativas para conseguir: $\text{suma}(ELi) > EF$
 - Swapping
 - Memoria virtual paginada: mecanismo hard/soft que permite cargar páginas bajo demanda
- Hard: detección de fallo de página y reanudación
 - Mecanismo de traducción debe detectar "página no presente en memoria física"
 - Procesador debe ser capaz de detenerse a mitad de una instrucción y posteriormente reanudar su ejecución
- Soft: rutina de servicio al fallo de página
 - Bloquea al proceso, selecciona contenedor destino, inicia transferencia DMA a contenedor destino, planifica siguiente proceso, cede el control

Detección de fallo de página: bit de validez



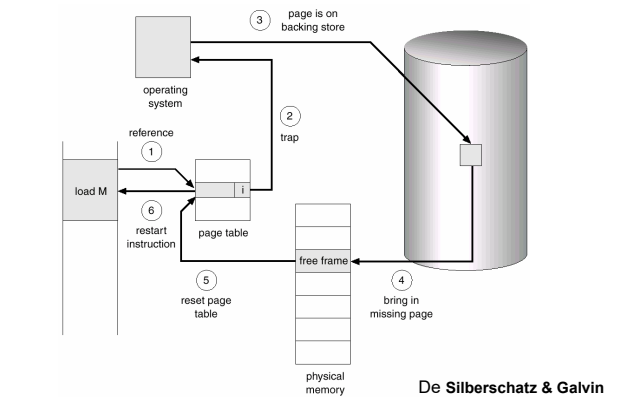
De Silberschatz & Galvin

Ejemplo memoria virtual



De Silberschatz & Galvin

Servicio al fallo de página



De Silberschatz & Galvin

Sistemas Operativos

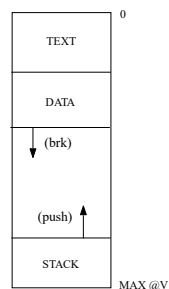
Gestión de Memoria en UNIX

- Regiones y Tablas
- Sistemas basados en SWAP
- Sistemas basados en Demand Paging
- Ejemplo fork()
 - fork() y Swap
 - fork() y Demand Paging
- Llamadas asociadas
- Funciones de librería, ejemplos
- Mapeo de ficheros en memoria: ejemplos

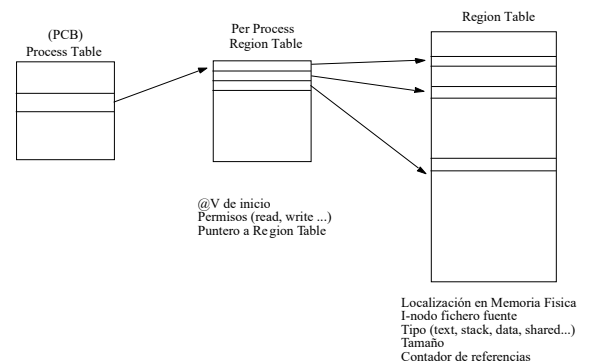
Gestion de Memoria

Gestión de memoria en UNIX: Regiones

- Región:
 - Area contigua del espacio virtual de un proceso
 - Tratada como un único objeto
 - Uniforme en cuanto a permisos, shared...



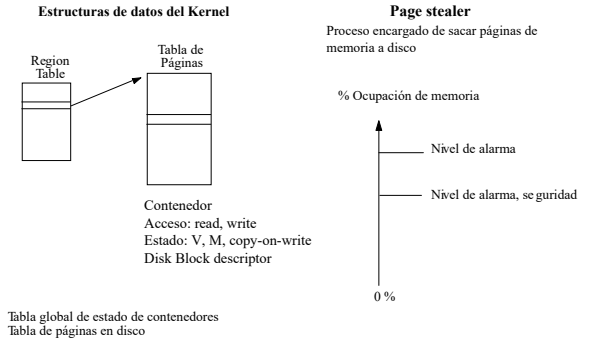
Gestión de memoria en UNIX: Tablas



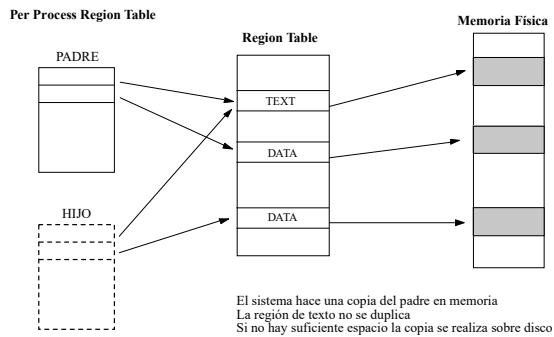
Sistemas basados en SWAP

- Espacio en disco(swap device):
 - una partición de disco, almacena procesos expulsados
 - gestión de espacio por particiones variables
- Razones de expulsión de un proceso
 - fork(), brk(), crecimiento de pila
 - Se requiere espacio para recuperar otro proceso
- Solo se copia a disco la parte del espacio virtual usada
- Proceso swapper
 - Es un proceso mas
 - Devuelve procesos de disco a memoria
 - Entra en ejecución cada cierto tiempo
 - Mira si hay procesos "preparados en disco"
 - Busca espacio en memoria y los copia
 - Si no lo hay busca víctima y la expulsa

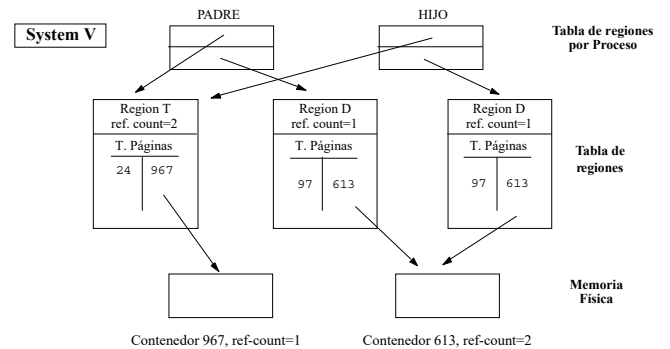
Sistemas basados en Demand Paging



fork() y Swap



fork() y Demand Paging



fork() y Demand Paging

BSD

- Crea una copia de todas las páginas privadas:
 - Data, Stack
- Ofrece una llamada alternativa a fork()
- vfork(): no copia ni las tablas de página
 - Trata las regiones de datos como la de texto
 - Padre e hijo comparten memoria física
 - Solo debe usarse para llamar a exec() inmediatamente

Llamadas asociadas

- brk(end_ds)
 - Cambia límite superior del segmento de datos a end_ds
- old_end_ds= sbrk(increment)
 - Suma increment bytes al límite superior del segmento de datos
 - Increment puede ser negativo
 - devuelve el valor antiguo de end_ds
- brk() puede fallar por varias razones:
 - Al ampliar la región se colisiona con otra
 - Al ampliar la región se sobrepasa el espacio virtual máximo del proceso
 - Faltan recursos:
 - No hay espacio físico en memoria o en disco (dependiendo de la implementación)

Funciones de librería

```
#include <stdlib.h>
```

- `void * malloc(size_t size);`
 - Reserva espacio para `size` bytes. No inicializa el espacio
- `void * calloc(size_t nelem, size_t elsize);`
 - Reserva espacio para `nelem` elementos de `elsize` bytes
 - Inicializa el espacio reservado con ceros
- `void * free(void * ptr);`
 - Libera el espacio apuntado por `ptr`. Este espacio no queda liberado para el Kernel
 - Hay una estructura de gestión de espacios propia de la librería
 - No mezclar llamadas a `brk()` con funciones de librería.
- `void * realloc(void * ptr, size_t newsize);`
 - Cambia el tamaño del bloque apuntado por `ptr` al valor `newsize` (sin inicializar si aumenta el espacio)
 - Devuelve puntero al inicio de bloque, puede ser distinto del original
 - Si `ptr=NULL` actúa como `malloc()`
 - Si `size=0` actúa como `free()`



Ejemplo malloc(), calloc()

```
main(argc, argv)
int argc; char *argv[];
{
    int x[1000], y[1000]; /* reserva 1000 elem */
    int i, n, res=0;

    n=atoi(argv[1]);
    leer_vector(x,n);
    leer_vector(y,n);
    for(i=0;i<n;i++) /* solo se usan n */
        res+=x[i]*y[n-i-1];
    printf("res=%d\n",res);
}
```



Ejemplo malloc(), calloc()

```
main(argc, argv)
int argc; char *argv[];
{
    int *x,*y;
    int i, n, res=0;
    n=atoi(argv[1]);
    x=malloc(n*sizeof(int)); /* reserva n*4 bytes */
    y=calloc(n,sizeof(int)); /* reserva espacio n int */
    leer_vector(x,n);
    leer_vector(y,n);
    for(i=0;i<n;i++)
        res+=x[i]*y[n-i-1];
    printf("res=%d\n",res);
}
```

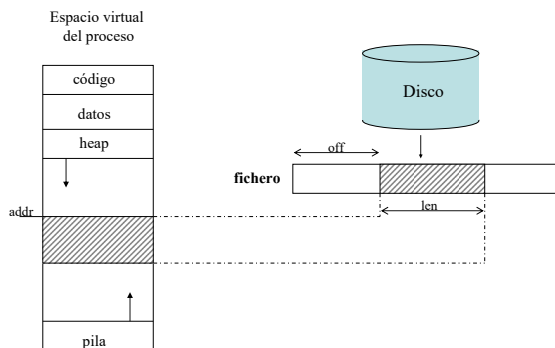


Ejemplo realloc()

```
main()
{
    int *p, *q;
    ...
    p=calloc(1000,4);
    ...
    q=p+50;
    ...
    p=realloc(p,8000);
    ...
    printf("p[50]=%d\n",*q);
}
```



Mapeo de ficheros en memoria



Mapeo de ficheros en memoria: mmap()

- Permite mapear un fichero de disco en un buffer de memoria
- Para realizar operaciones E/S sin read/write
- Fichero previamente abierto

```
#include <sys/mman.h>
void * mmap(addr, len, prot, flag, filedes, off)
char *addr;
size_t len;
int prot, flag, filedes;
off_t off;
```
- `addr=0`: colócalo donde quieras (recomendable)
- `addr!=0`: hint (intenta colocarlo en `addr`) `addr` debe ser múltiplo del tamaño de página
- `len`: número de bytes mapeados
- `off`: a partir de que punto del fichero se mapea (normalmente debe ser múltiplo del tamaño de página)



Mapeo de ficheros en memoria: mmap()

```
#include <sys/mman.h>
void * mmap(addr, len, prot, flag, filedes, off)
char * addr;
size_t len;
int prot, flag, filedes;
off_t off;
```

- **filedes**: descriptor del fichero, tiene que estar abierto
- **prot**: intención de uso: debe respetar los del open()
PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
ejemplo: PROT_READ | PROT_WRITE
- **flag**: MAP_FIXED: addr pasa de hint a obligación
MAP_SHARED: un store sobre la región = write sobre fichero
MAP_PRIVATE: un store provoca una copia privada
el fichero nunca se modifica
- **mmap devuelve**:
 - Si todo va bien -> @ comienzo zona de mapeo
 - Si hay error: MAP_FAILED ((void *)-1)

/* reverse.c */

```
#include <stdio.h>
#include <fcntl.h>
#include "error.h"

main(argc,argv)
int argc; char *argv[];
{
    char c;
    int i, fdnt;
    long where;

    if(argc != 2){ printf( "Uso: %s fichero_a_invertir" argv[0]; exit(1); }

    if((fdnt = open( argv[1], O_RDONLY )) == -1) syserr("open");
    if((where = lseek( fdnt, -1L, 2 )) == -1) syserr("lseek");

    while(where >= 0){
        read(fdnt, &c, 1);
        write(1, &c, 1);
        where = lseek( fdnt, -2L, 1 );
    }
}
```

Invierte el contenido de un fichero

formato long



Universidad de Zaragoza

16

T. Monreal, J.L. Briz & P. Ibáñez
Arquitectura y Tecnología de Computadores - DIS



Universidad de Zaragoza

17

T. Monreal, J.L. Briz & P. Ibáñez
Arquitectura y Tecnología de Computadores - DIS



Ejemplo mmap(): mreverse (1de2)

```
/* mreverse.c Invierte el contenido de un fichero.*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h> /* mmap() */
#include <fcntl.h>
#include "error.h"

main(argc,argv)
int argc;
char *argv[]; {
    int fdnt;
    long fsize;
    char *src;

    if(argc != 2){
        printf( "Uso: %s fichero_a_invertir", argv[0];
        exit(1);
    }
}
```

Ejemplo mmap(): mreverse (2de2)

```
if((fdnt = open(argv[1], O_RDONLY)) == -1)
    syserr("open del primer fichero");
if((fsize=lseek(fdnt,0,SEEK_END)) == -1)
    syserr("lseek al final del fichero");

src=mmap(0,fsize,PROT_READ,MAP_SHARED,fdnt,0);
if (src == MAP_FAILED) syserr("mmap error for input");

fsize--;
while(fsize >= 0){
    write(1, &src[fsize], 1);
    fsize--;
}
}
```



Universidad de Zaragoza

18

T. Monreal, J.L. Briz & P. Ibáñez
Arquitectura y Tecnología de Computadores - DIS



Universidad de Zaragoza

19

T. Monreal, J.L. Briz & P. Ibáñez
Arquitectura y Tecnología de Computadores - DIS

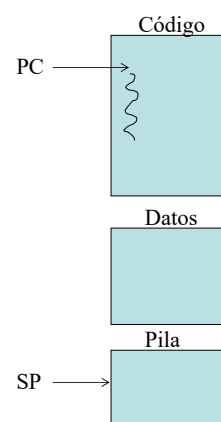


Gestión de threads (hilos)

Sistemas Operativos
Pablo Ibáñez

[Sta]: capítulo 4
[SGG]: capítulo 4

Modelo de proceso



Registros

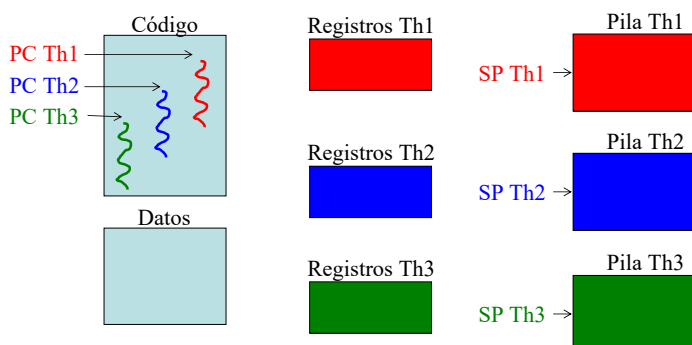
- Programación/ejecución secuencial
 - Simple
 - Válida para resolver muchos problemas
- En algunos casos es un modelo muy limitado. Ejemplos
 - Aplicación con varias actividades concurrentes de E/S y/o cálculo
 - Disponibilidad de varios procesadores
- Alternativa: programación/ejecución concurrente
 - Pensemos en varios procesos

- Aplicación con E/S concurrentes
 - Recibe peticiones (read, write, ...) de procesos de otras máquinas
 - realiza las operaciones sobre disco y responde las peticiones
- Versión secuencial simple: servicio de peticiones en serie
 - Bucle que lee petición, realiza llamada al sistema bloqueante, y responde
 - Muy bajas prestaciones
- Versión secuencial optimizada: código complejo
 - Usa llamadas al sistema no bloqueantes para recibir peticiones y para realizar las operaciones en disco
 - guarda memoria de las operaciones pendientes de respuesta, que posiblemente llegarán en desorden
- Solución concurrente
 - Cada petición crea un proceso que realiza la operación sobre disco y responde

- Aplicación con varias actividades concurrentes: entrada de texto, corrector ortográfico, formato en pantalla
- Cada vez que se introduce nuevo texto desde teclado hay que ejecutar el código del corrector ortográfico. Opciones:
 - Se ejecuta corrector y después se continúa: se desatiende la lectura de teclado durante un tiempo
 - Se ejecuta corrector a la vez que se atiende teclado: se complica mucho la programación de la aplicación
- Además, word tiene que realizar otras tareas como la organización del texto en pantalla
 - espaciado entre caracteres, líneas y párrafos
 - formatos de letra, ...
- Solución concurrente: cada actividad un proceso
 - Cada actividad se programa de forma independiente
 - Se despierta solo cuando se requiere, distintas prioridades, ...

- Ejemplo: sumar los elementos de un vector muy grande
- Disponibilidad de varios procesadores
- Objetivo: aumentar rendimiento
 - Repartir el trabajo entre varios procesos
 - Cada proceso trabaja sobre un trozo del vector
 - Cada proceso ejecuta en un procesador
- Ejemplos 1 y 2
 - independientes de la plataforma, sirven para uno o varios cores
 - Objetivo principal: facilitar la programación
- Ejemplo 3
 - Solo tiene sentido en un sistema multicore,
 - Casi siempre con numero de threads \leq numero de cores
 - Objetivo: rendimiento
 - Complica la programación

- Los procesos (tal como los hemos visto hasta ahora):
 - **Son propietarios de recursos** – espacio de memoria para almacenar su imagen, ficheros abiertos, tratamiento de señales, ...
 - **Son la unidad de ejecución/planificación** – siguen un camino de ejecución, el SO les otorga tiempo en CPU
- En el modelo proceso/hilo, estas dos características se tratan de forma independiente por el sistema operativo
 - Los procesos **son propietarios de recursos**
 - Los hilos **son la unidad de ejecución/planificación**
 - Un proceso puede tener varios hilos



- Hilo: unidad básica de utilización de la CPU
 - Comprende un ID de hilo, un PC, un conjunto de registros y una pila
 - Comparte con otros hilos: código, datos y recursos de sistema asignados al programa (ficheros abiertos, señales, ...)

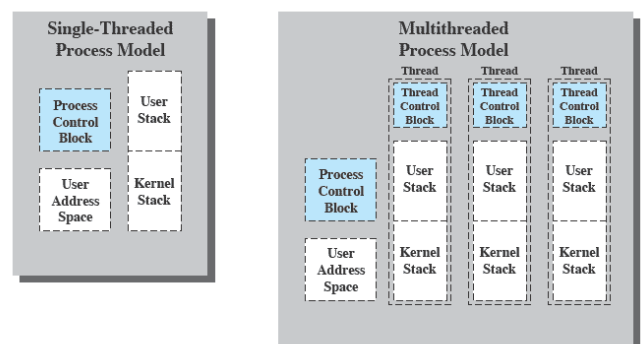
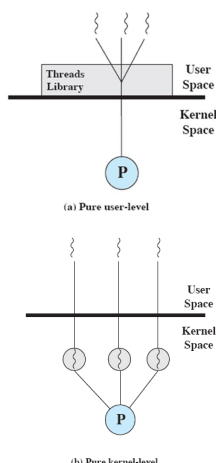


Figure 4.2 Single Threaded and Multithreaded Process Models

- Hilos de usuario (User Level Thread, ULT)
 - La gestión de hilos la lleva la propia aplicación (biblioteca)
 - El SO no da ningún soporte, no conoce los hilos
- Hilos de sistema (Kernel level Thread, KLT)
 - también llamados: kernel-supported threads, lightweight processes
 - El SO conoce y gestiona hilos, planifica a nivel de hilo



Situación inicial

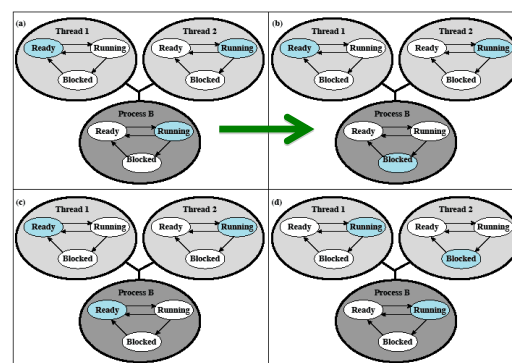


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Th2 SC
Bloquea
Al proceso B

Hilos de usuario: planificación en dos niveles

Hilos de usuario: planificación en dos niveles

Situación inicial

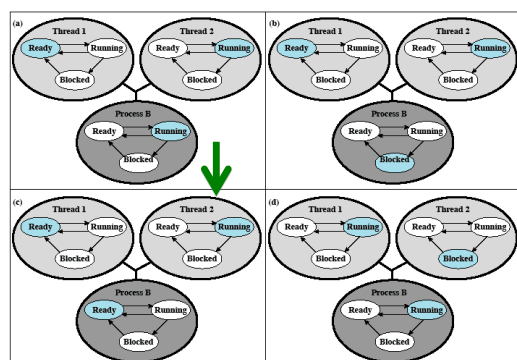


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Quantum proceso B

Situación inicial

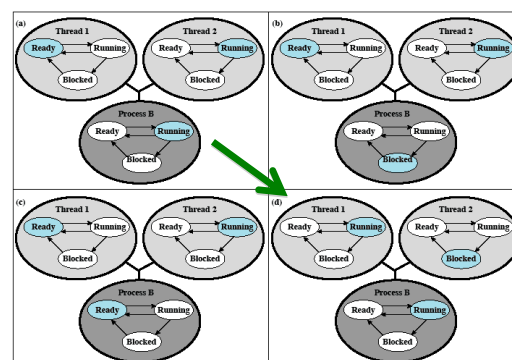


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Th2 sincroniza
con Th1 y se
Bloquea

Hilos de sistema: Ventajas/Desventajas

Bibliotecas de hilos

- Si un hilo se bloquea, no bloquea a todo el proceso. El kernel puede poner en ejecución otros hilos del mismo proceso
- En un sistema multiprocesador, el kernel puede poner simultáneamente en ejecución varios hilos del mismo proceso
- Tienen mayor coste en cambio de contexto de hilo
- Tienen mayor coste para crear, señalar/esperar, ...

Latencia en uSg. VAX/UNIX	ULT	KLT	proceso
Crear	34	948	11.300
Señalar-esperar	37	441	1.840

- Contienen la colección de funciones ofrecidas al usuario para trabajar con hilos
 - Crear, terminar, funciones de sincronización,...
- Ejemplos:
 - POSIX Pthreads (existe como ULT y KLT)
 - Windows Threads (KLT)
 - Java Threads (se implementa sobre la librería del host)

- Veremos Pthreads en el siguiente tema

- Además de `fork()`, linux ofrece llamada `clone()`
- `Clone()` permite determinar el grado de compartición entre padre e hijo

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

SC's Pthreads

[Ste05]: cap. 11



SC's Pthreads

- Identificación de threads
- Creación de threads
- Terminación de threads
- Threads + `fork()`, `exec()`
- Ejemplo

Identificación de threads

- Tipo de dato `pthread_t` para el identificador de thread (unsigned int en Solaris)

```
#include <pthread.h>
```

```
pthread_t pthread_equal(pthread_t tid1,  
pthread_t tid2);
```

- Devuelve (`#0, TRUE`) si `tid1==tid2`
- Devuelve (`0, FALSE`) si `tid1!=tid2`

```
pthread_t pthread_self(void);
```

- Devuelve el identificador del thread que llama (similar a `getpid()` en procesos)



Creación de threads

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *tidp,  
const pthread_attr_t *attr,  
void *(*start_rtn)(void *),  
void *arg);
```

- Devuelve: 0 si OK, número de error si falla
 - `tidp`: Variable donde pone el identif. de thread creado
 - `attr`: Atributos de creación del thread (NULL para atributos por defecto).
 - `start_rtn`: función donde comienza la ejecución del thread
 - `arg`: argumento(s) de la función `start_rtn`

Terminación de threads

- Si un thread ejecuta `exit`, termina el proceso entero (todos los threads).
- Para que un thread termine sin acabar el proceso entero debe hacer:
 - Retornar de su rutina de comienzo. El valor devuelto es el código de terminación del thread.
 - Thread cancelado por otro thread en el mismo proceso (`pthread_cancel`)
 - Ejecutar `pthread_exit`

```
#include <pthread.h>
int pthread_exit(void *rval_ptr);
```

 - El puntero `rval_ptr` está disponible para el resto de threads por medio de `pthread_join`



Terminación de threads

```
#include <pthread.h>

int pthread_join(pthread_t tid,
    void **rval_ptr);
```

- Devuelve: 0 si OK, número de error si falla
- El thread que ejecuta `pthread_join` se bloquea hasta que el thread `tid` termina (similar a `waitpid` en procesos). `rval_ptr` vale:
 - Si el thread `tid` simplemente retorna de su función `start`, `rval_ptr` toma el valor devuelto
 - `rval_ptr=PTHREAD_CANCELED` si el thread `tid` es cancelado
 - Si no interesa el valor devuelto por el thread `tid`, poner `NULL` en `rval_ptr`



Threads + fork() exec()

- Si un thread hace `fork()`, dos posibilidades
 - Nuevo proceso con solo un thread (el que hace `fork`)
 - Sería lo lógico si luego se llama a `exec()`
 - Nuevo proceso con todos los threads
 - Sería lo lógico si no hay `exec()` posterior
- En `hendrix (solaris 10)`
 - `fork()` duplica unicamente el thread que llama a `fork()`
 - `forkall()` nueva SC para replicar todos los threads
- Si un thread hace `exec()`, funciona como siempre, es decir, desaparecen todos los threads y se sustituye por el ejecutable indicado en `exec()`



Thr_n.c

```
#include <stdio.h> <stdlib.h> <pthread.h> "error.h"
struct arg {
    int ini; /* estructura para pasar arg/res al thread */
    int fin; /* indice inicial */
    int res; /* indice final */
    int res; /* resultado devuelto */
};
int n,n_thr,*v; /* var global compartidas por todos los threads */

void main(int argc, char *argv[]) {
    int i,tam,error,suma;
    pthread_t *tid;
    struct arg *param;

    if (argc != 3) {printf("Uso: thr_n <int> <n_thr>\n"); exit(1);}
    n=atoi(argv[1]); n_thr=atoi(argv[2]);

    v=calloc(n,sizeof(int)); /* reserva vector a sumar */
    tid=calloc(n_thr,sizeof(pthread_t)); /* reserva tid_threads */
    param=calloc(n_thr,sizeof(struct arg)); /* reserva parametros */

    for (i=0;i<n;i++) v[i]=1; /* inicializacion vector a sumar */
    tam=n/n_thr; /* tamaño trozo para cada thread */
    printf("Calculando S(%d) en %d threads -> tam=%d\n",n,n_thr,tam);
```



Thr_n.c

```
/* creando threads para suma parciales */
for (i=0;i<n_thr;i=i+1) {
    param[i].ini=tam*i; /* param1 del thread */
    param[i].fin=tam*(i+1); /* param2 del thread */
    error=pthread_create(&tid[i],NULL,start,&param[i]);
    if (error!=0) syserr(pthread_create);
}

suma=0;
if (n_thr*tam<n) /* falta sumar el resto */
    for (i=n_thr*tam;i<n;i=i+1) suma=suma+v[i];

for (i=0;i<n_thr;i=i+1) { /* espera terminacion threads */
    pthread_join(tid[i],NULL);
    suma=suma+param[i].res; /* extrae resultado del thread */
}
printf("Terminado. S(%d)=%d\n",n,suma);
}
```



Thr_n.c

```
void *start(void *p) {
    pthread_t tid;
    int i,ini,fin,tmp;

    ini=((struct arg *)p)->ini; /* ini=arg1 */
    fin=((struct arg *)p)->fin; /* fin=arg2 */
    tmp=0;
    for (i=ini;i<fin;i=i+1) tmp=tmp+v[i]; /* calcula la suma parcial */
    ((struct arg *)p)->res=tmp; /* almacena resultado */
    pthread_exit(NULL); /* acaba "sin devolver" resultado */
}
```

