


# Sistemas Operativos

Universidad Complutense de Madrid

2020-2021

## Módulo 3.1: Gestión de Procesos

Juan Carlos Sáez

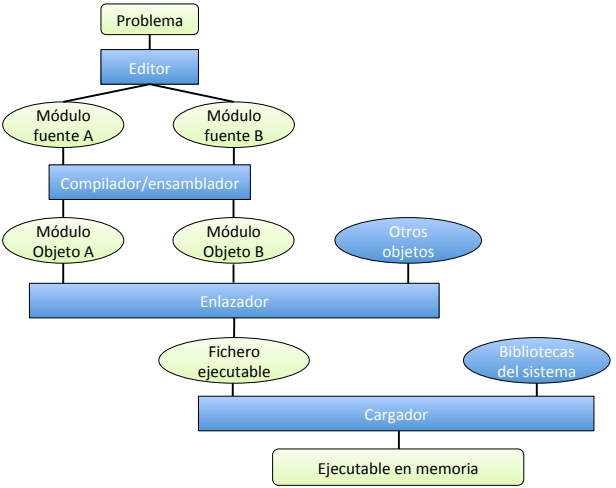

# Contenido

# Contenido

# Concepto de Proceso

- **Programa**: fichero ejecutable en un dispositivo de almacenamiento permanente
- **Proceso**: Programa en ejecución
  - Conjunto de recursos que permiten la ejecución del programa
    - Mapa de memoria
    - Ficheros abiertos
    - Hilos de ejecución
    - ...
- Un proceso puede constar de uno o varios hilos de ejecución
  - Hilo: flujo de ejecución de instrucciones independiente
  - La unidad mínima planificable en una CPU

# Preparación del código de un proceso



# Recuerda: comandos útiles

- `gcc`
  - Compilador C de GNU
  - Realiza todas las etapas

```
$ gcc --save-temps hello.c -o hello.o
```

- `ldd`
  - Modo de uso: `ldd <fichero-ejecutable>`
  - Permite ver las librerías dinámicas con las que hemos enlazado
- `nm` / `objdump`
  - Permiten visualizar partes de un ejecutable

# Entorno del proceso

- **Entorno:** tabla que se pasa al proceso en su creación
  - Cada entrada es un par (*nombre*,*valor*)
  - Se puede consultar con el comando `env`

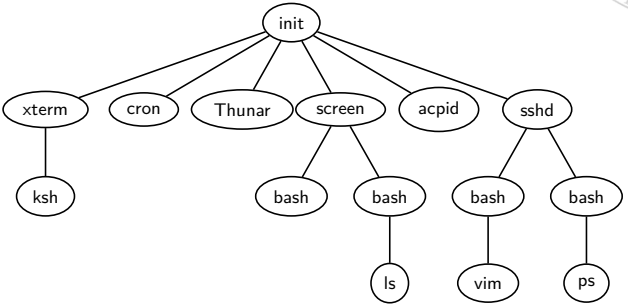
**Ejemplo**

```
$ env
PATH=/usr/bin:/home/joe/bin
TERM=vt100
HOME=/home/joe
PWD=/home/joe/books/OperatingSystems
TIMEZONE=EDT
...
```

- El contenido inicial de la tabla se hereda del proceso padre
  - Puede cambiarse mediante:
    1. Func. de la biblioteca estandar de "C": `getenv()`, `putenv()`, `unsetenv()`
    2. Comandos del shell: `echo $NAME`, `export NAME=val`, `unset NAME`

# Jerarquía de procesos (UNIX)

- En UNIX los procesos se crean mediante clonación de un proceso existente
  - fork()
- Familia de procesos
  - Proceso padre
  - Proceso hijo
  - Proceso hermano
  - ...
- Vida de un proceso
  - Crea
  - Ejecuta
  - Muere o termina



# Consulta procesos activos

- `ps`
  - Permite ver la información de todos los procesos en ejecución
  - `man ps` para consultar las múltiples opciones
- `top` (*table of processes*)
  - Muestra los procesos en ejecución, refrescando la información periódicamente
  - Permite interactuar con los procesos (p.ej., enviar señales)
- `pstree`
  - Permite visualizar el árbol de procesos

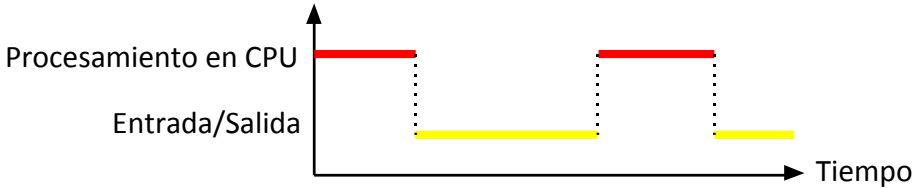
# Usuarios y Grupos

- Todo proceso del sistema tiene asociado un usuario propietario
- Usuario: Persona autorizada a utilizar un sistema
  - Se identifica en la autenticación mediante:
    - Nombre de usuario
    - Clave (password)
  - Internamente el SO le asigna un "UID" (user identification)
- Superusuario (*root*)
  - Tiene todos los derechos
  - Administra el sistema
- Grupo de usuarios
  - Los usuarios se organizan en grupos
  - Todo usuario ha de pertenecer al menos a un grupo
  - La existencia de grupos simplifica la administración del sistema

# Contenido

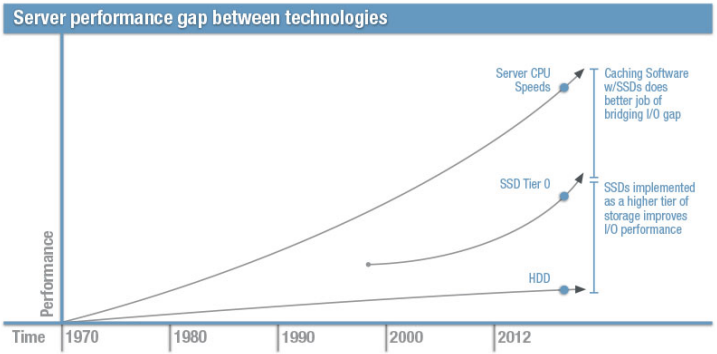
# Motivación de la multitarea

- En los antiguos SSOO tipo batch (años 50/60) los procesos se ejecutaban en serie
  - Cola de procesos del sistema
- Cuando proceso en ejecución realiza operación de E/S la CPU se quedaba inactiva
  - Uso de E/S programada (espera activa)



# Motivación de la multitarea

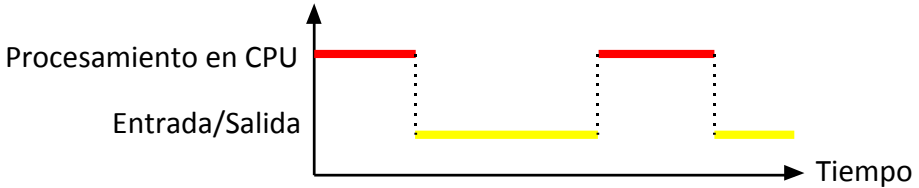
- Desde la aparición de la tecnología CMOS hasta nuestros días, la CPU es cada vez más rápida que los dispositivos de E/S
  - E/S programada se vuelve más ineficiente




*Source: RTC magazine*

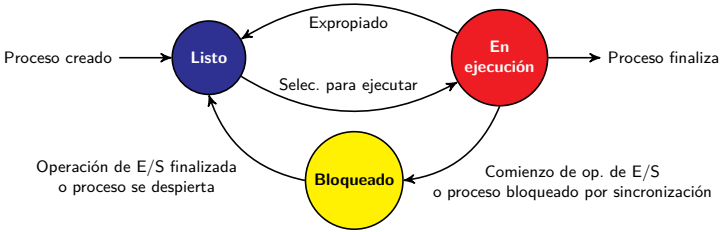
# Base de la multitarea

- Explota el hecho de que los procesos alternan fases de E/S de procesamiento en CPU
- Paralelismo real entre E/S y CPU
  - Muchos procesos pueden realizar operaciones de E/S en paralelo
- Necesario almacenar en memoria la imagen de memoria de varios procesos



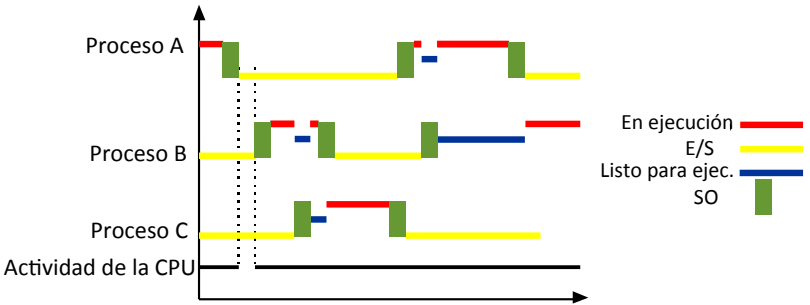
# Estados básicos de un proceso

1. **En ejecución** (uno por procesador/core)
2. **Bloqueado** (en espera de completar E/S o por motivos de sincronización)
3. **Listo para ejecutar**




- **Planificador:** Componente del SO que decide qué proceso se ejecuta en cada procesador y en qué instante

# Ejecución en un sistema multitarea



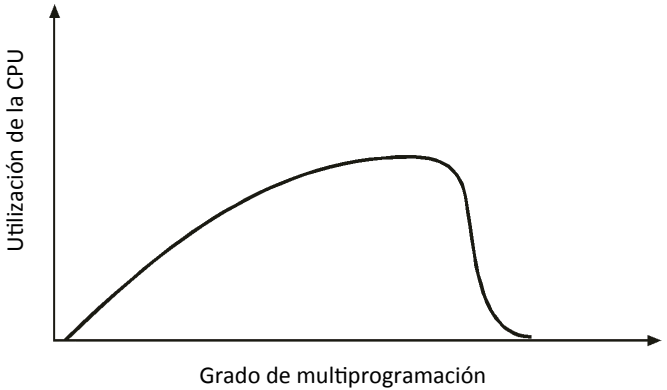

- Tarea/proceso *idle*


# Ventajas de la multitarea

- Optimiza el uso de la CPU
- Permite el servicio interactivo simultáneo de varios usuarios de forma eficiente
- Facilita la programación, dividiendo los programas en procesos (modularidad)

# Grado de multiprogramación

- **Grado de multiprogramación:** número máximo de procesos activos que un sistema puede gestionar de forma eficiente
- Para sistemas con memoria virtual:



# Contenido

# Información de un proceso

1. **Estado del procesador**: contenido de los registros del modelo de programación
2. **Imagen de memoria**: contenido de los segmentos de memoria en los que reside el código y los datos del proceso
3. **Bloque de control del proceso (BCP) o Descriptor de proceso**
   - Estado actual del proceso
   - Almacenamiento para el estado del procesador
     - Actualizado cuando proceso no se está ejecutando en la CPU
   - Identificadores pid, uid, etc.
   - Prioridad
   - Segmentos de memoria (espacio de direcciones)
   - Ficheros abiertos
   - Temporizadores
   - Señales
   - ...

# Estado del procesador

- Está formado por el contenido de todos sus registros:
  - Registros generales
  - Contador de programa
  - Puntero de pila
  - Registro de estado
  - Registros especiales
- Cuando un proceso se está ejecutando su estado del procesador reside en los registros del computador
  - El estado del procesador de un proceso que no está en ejecución reside en el BCP

# Imagen de memoria

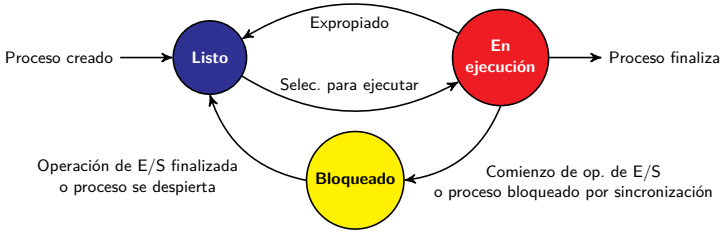
- La imagen de memoria está formada por el conjunto de regiones de memoria que un proceso está autorizado a utilizar
  - Términos equivalentes: *espacio de direcciones*, *mapa de memoria*
- La imagen de memoria, dependiendo del computador, puede estar referida a memoria virtual o memoria física
- Si un proceso genera una dirección que esta fuera del espacio de direcciones el HW genera una excepción que el SO captura
  - Típicamente, el SO mata el proceso cuando esto ocurre
  - Soporte HW $\rightarrow$ *Memory Management Unit* (MMU)

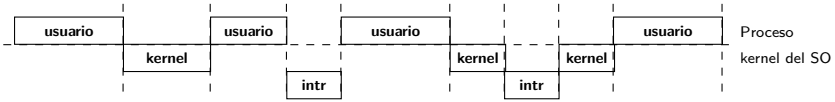
# Información del BCP

- **Información de identificación**:
  - PID del proceso, PID del padre (PPID)
  - ID de usuario y grupo reales (uid/gid reales)
  - ID de usuario y grupo efectivos (uid/gid efectivos)
- **Estado del procesador**
- **Información de control del proceso**:
  - Información de planificación y estado
  - Descripción de los segmentos de memoria del proceso
  - Recursos asignados (ficheros abiertos, ...)
  - Recursos de comunicación entre procesos
  - Punteros para estructurar los procesos en listas o colas
    - *task list*, árbol de procesos, *run queues*, *wait queues*

# Contenido

# Estados del proceso



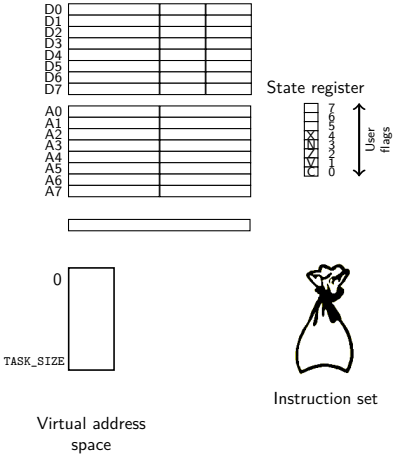



# Cambio de modo del procesador

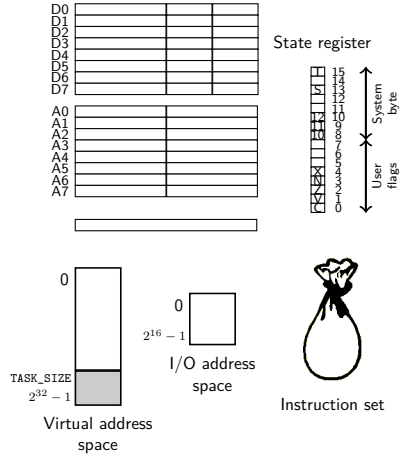
- Cuando se produce una interrupción o excepción mientras un proceso se ejecuta en modo usuario, el procesador cambia de modo de ejecución (a modo kernel)
- Al producirse la interrupción/excepción:
  - Se salva el valor de los registros (estado del procesador) en la pila de kernel del proceso
  - Se pasa a ejecutar la rutina de tratamiento de interrupción/excepción, bajo el control del SO
- Al finalizar la rutina, si el proceso actual sigue "en ejecución":
  - Restaura el estado de los registros del procesador
  - Termina con una instrucción RETI (retorno de interrupción). Dos acciones simultáneas
    1. Restituye el registro de estado (bit de nivel de ejecución)
    2. Restituye el contador de programa (para el nuevo proceso)

# Modo usuario vs. modo kernel

## Modo usuario




## Modo kernel



# Cambio de contexto

- El cambio de contexto es el conjunto de acciones que realiza el SO para cambiar el proceso que está actualmente en ejecución en una CPU
- Acciones:
  1. Salvar el contexto del proceso saliente (registros del modelo de programación) en el BCP
  2. Cambiar el estado del proceso saliente (En ejecución -> Otro Estado)
  3. Intercambio de los espacios de direcciones
     - Tabla de páginas + Segmentos de memoria que puede usar un proceso
     - En x86. CR3 (Puntero al directorio de páginas) + LDTR (Dir. de Local Descriptor Table)
     - Invalidación de entradas de la TLB si lo exige la arquitectura
  4. Cambiar el estado del proceso entrante, (Listo -> En ejecución)
  5. Restaurar su contexto (BCP -> registros) y volver a modo usuario

# Cambio de contexto



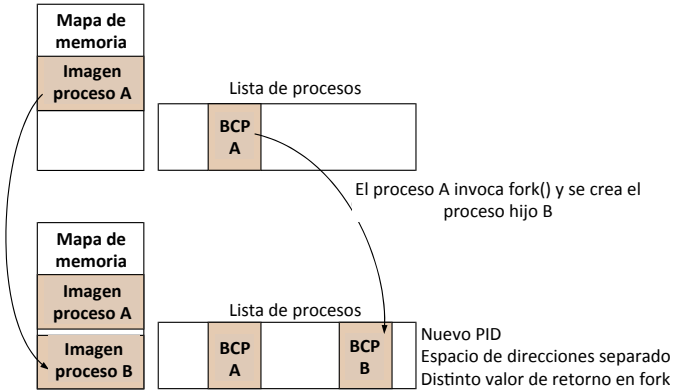
- Puede llegar a ser una operación bastante costosa
  - Reconstrucción del estado de la cache
  - Sobrecarga asociada a la activación del espacio de direcciones
- El cambio de modo de ejecución del procesador no siempre desencadena un cambio de contexto
  - Si el proceso actual sigue listo para ejecutar después de procesar la interrupción/excepción y el planificador no decide expropiar el proceso, no habrá cambio de contexto

# Servicios de gestión de procesos (POSIX)

- Identificación de procesos
- Entorno de un proceso
- Creación de procesos
- Cambio del programa de un proceso
- Esperar la terminación de un proceso
- Finalizar la ejecución de un proceso
- Información sobre procesos

# Servicios POSIX: `fork()`

- Crea un proceso clonando al padre



# Servicios POSIX: `fork()`

## Servicio

```
pid_t fork(void);
```

## Devuelve

- En caso de éxito retorna dos veces, una en el proceso hijo y otra en el padre
- Retorna 0 en el hijo y el PID del hijo en el padre
- En caso de fallo retorna solo una vez (valor de retorno -1)

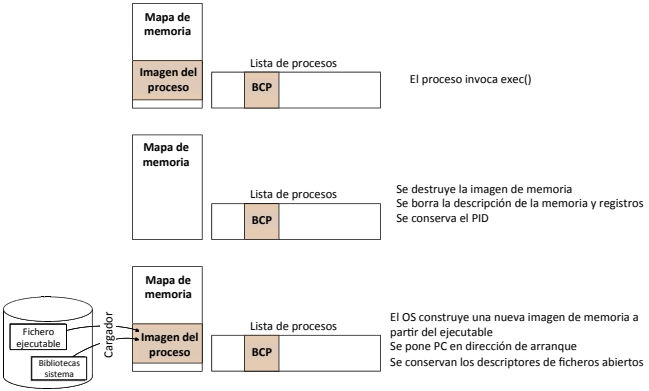
```
void main() {
   pid_t pid;
   ..
   pid = fork();
   if (pid == 0) {
      /* Proceso hijo */
      ...
   } else if (pid>0){
      /* Proceso padre */
      ...
   } else{
      /* Error */
   ...
   }
...
}
```

## Descripción

- Crea un proceso hijo que ejecuta el mismo programa que el padre
- El proceso hijo sólo tiene un hilo
- Hereda los ficheros abiertos (se copian los descriptores)

# Servicios POSIX: `exec()`

- Cambia el programa de un proceso



# Servicios POSIX: `exec()`

## Funciones disponibles

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

## Argumentos:

- `path`, `file`: nombre/ruta del ejecutable
- `arg`: argumentos del programa

## Descripción:

- Devuelve -1 en caso de error, en caso contrario **no retorna**
- El mismo proceso ejecuta otro programa
- Los ficheros abiertos permanecen abiertos
- Las señales con la acción por defecto seguirán por defecto, las señales con manejador tomarán la acción por defecto

# Servicios POSIX (ejemplos)

## Ejecutando `ls -l`

- Ejecutable ubicado en `/bin`
  - Ejecutar comando "`which ls`" para obtener ruta completa
  - Asumimos que `/bin` está en el `PATH` (`echo $PATH`)

```
/* Primera alternativa */
execl("/bin/ls","/bin/ls","-l",NULL);

/* Segunda alternativa */
execlp("ls","ls","-l",NULL);

/* Tercera alternativa */
char* arguments[]={"ls","-l",NULL};
execvp(arguments[0],arguments);
```

# Terminación de un proceso: `exit()`

## Servicio:

```
int exit(int status);
```

## Argumentos:

- Código de retorno para el proceso padre

## Descripción:

- Finaliza la ejecución del proceso.
- Se cierran todos los descriptores de ficheros abiertos
- Se liberan todos los recursos del proceso

# Espera terminación de proceso hijo: `wait()`

## Servicio:

```
#include <sys/types.h>
pid_t wait(int *status);
```

## Argumentos:

- `status`: parámetro de salida. Código de terminación del proceso hijo

## Descripción:

- Devuelve el identificador del proceso hijo o -1 en caso de error.
- Permite a un proceso padre esperar hasta que termine un proceso hijo. Devuelve el identificador del proceso hijo y el estado de terminación del mismo.

# Ejemplo

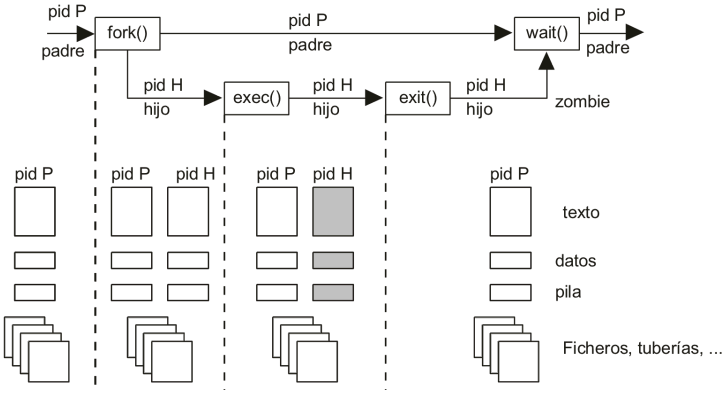
```
void main(){
   int a=0;
    int status;
    pid_t pid;

    pid = fork();
    if (pid == 0) {
    /* proceso hijo */
        a++;
    } else if (pid>0){
    /* proceso padre */
        wait(&status);
        a=a+2;
     } else{
        fprintf(stderr,"Can't fork()\n");
        exit(1);
     }

    printf("My PID is %d, a=%d\n",getpid(),a);
    exit(0);
}
```

- ¿Qué imprimirá por pantalla el proceso hijo con `printf()`?
- ¿y el padre?
- ¿Ocurriría lo mismo si declaráramos la variable *a* como variable global?

# Uso normal de los servicios



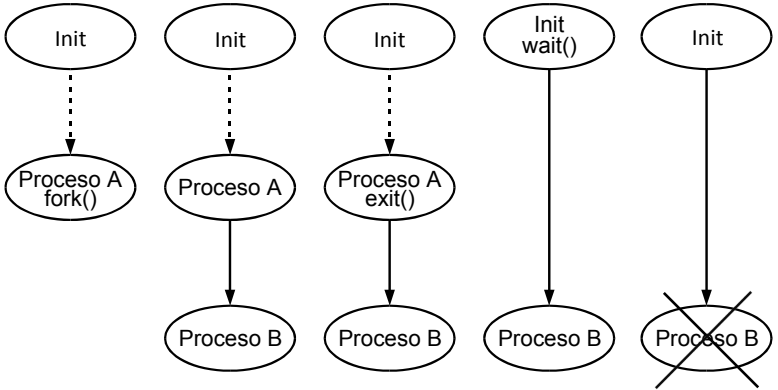
# Programa de ejemplo

```c
#include <sys/types.h>
#include <stdio.h>

/* programa que ejecuta el comando 'ls -l' */
void main() {
    pid_t pid;
    int status;
    pid = fork();
    if (pid == 0) { /* proceso hijo */
        execlp("ls","ls","-l",NULL);
        exit(1);
    } else { /* proceso padre */
        while (pid != wait(&status));
        exit(0);
    }
}
```

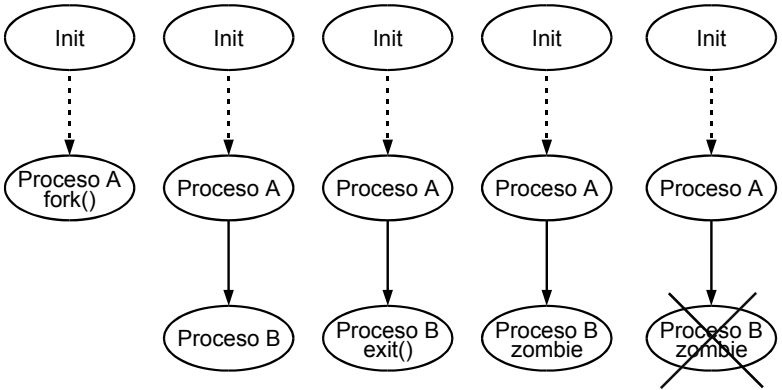
# Ciclo de vida de un proceso (I)

- El padre muere → INIT acepta los hijos



# Ciclo de vida de un proceso (II)

- Zombie: el hijo muere y el padre no invoca `wait()`



# Contenido

# Semántica de coutilización

- Cualquier mecanismo de acceso a ficheros tiene problemas cuando varios usuarios trabajan con el fichero simultáneamente
- La semántica de coutilización especifica el comportamiento del acceso simultáneo a un fichero desde varios procesos
  - Conjunto de reglas que establecen también cuando los cambios realizados por un proceso en un fichero son visibles desde otro proceso diferente

## Tipos de semánticas

- UNIX
- Semántica de Sesión
- Semántica de versiones
- Ficheros inmutables

# Semántica UNIX

- Cada fichero se expone como un "objeto global" a todos los procesos
  - Las escrituras son inmediatamente visibles para todos los procesos con el fichero abierto
  - El resultado de dos escrituras consecutivas a la misma región del fichero (desde varios procesos) es la segunda operación
- Los procesos emparentados comparten la visión lógica de los ficheros abiertos, incluyendo el puntero de posición
  - La herencia de ficheros sólo afecta a los ficheros abiertos por el proceso padre antes de invocar `fork()`

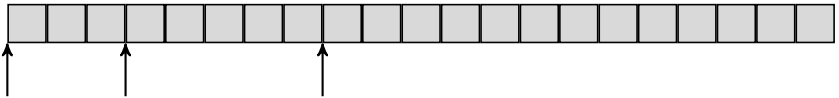
# Semántica UNIX: Ejemplo

**Proceso 1 (padre)**

```
char buf[MAX_SIZE];
int fd = open("myfile.dat", O_RDONLY);
fork();
read(fd,buf,5)
```

**Proceso 2 (hijo)**

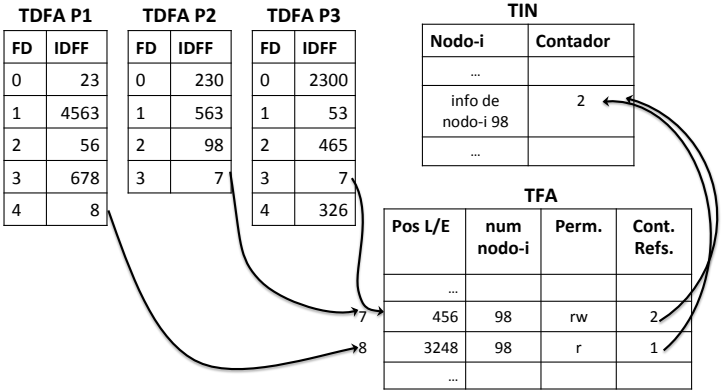
```
read(fd,buf,3)
```



1. El proceso $P1$ abre un fichero $F$ y crea un proceso hijo $P2$ con `fork()`
2. El proceso $P2$ hereda $F$: (a) puede acceder a $F$ directamente y (b) cualquier cambio realizado en el puntero de posición realizado por $P1$ o $P2$ es visible desde cualquiera de los procesos

# Tablas del Sist. de Gestión de Ficheros

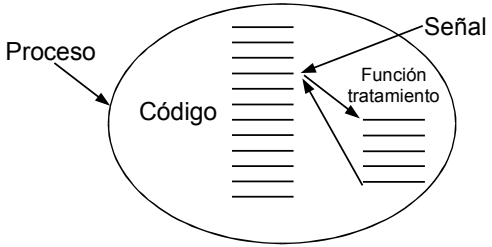
- Tabla intermedia de nodos-i (**TIN**):Tabla global que incluye los nodos-i de los ficheros en uso
- Tabla intermedia de posiciones (**TFA**): Tabla global que incluye las distintas representaciones lógicas de los ficheros en uso
- Tabla de descriptores de ficheros abiertos (**TDFA**): Una por proceso




# Contenido

# Concepto de señal

- Las señales son interrupciones al proceso




- Envío o generación:
  - Proceso → Proceso (con mismo uid) con `kill()` o mediante el comando `kill`
  - SO → Proceso

# Señales

- Hay muchos tipos de señales, según su origen
  - `SIGILL` instrucción ilegal
  - `SIGALRM` vence el temporizador
  - `SIGKILL` mata al proceso
- El SO las transmite al proceso
  - El proceso debe estar preparado para recibirla
    - Especificando un manejador de señal con `sigaction()`
    - Enmascarando la señal con `sigprogmask()`
  - Si no está preparado → acción por defecto
    - El proceso, en general, muere
    - Hay algunas señales que se ignoran o tienen otro efecto
- `pause()`: para el proceso hasta que recibe una señal

# Envío de señales: ejemplo

**Terminal 1**

```
osuser@debian:~$ sleep 50
^C
osuser@debian:~$ sleep 50

osuser@debian:~$ ^C
osuser@debian:~$ ^C
osuser@debian:~$ echo $$
5711
osuser@debian:~$ ^C
osuser@debian:~$ ^C
osuser@debian:~$
```

**Terminal 2**

```
osuser@debian:~$ ps -ef | grep sleep
osuser    5756  5711  0 09:19 pts/3    00:00:00 sleep 50
osuser    5758  5667  0 09:19 pts/2    00:00:00 grep sleep
osuser@debian:~$ kill -s SIGINT 5756
osuser@debian:~$ kill -s SIGINT 5711
osuser@debian:~$ kill -s SIGINT 5711
```

*La variable built-in $$ de BASH almacena el PID del proceso shell*

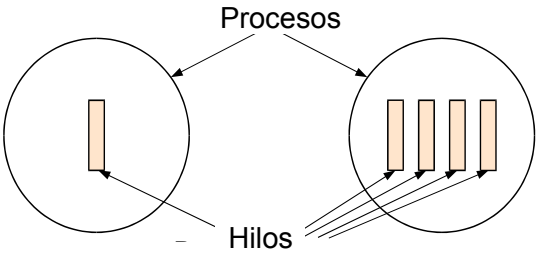
# Servicios POSIX para señales

- `int kill(pid_t pid, int sig);`
  - Envía la señal `sig` al proceso con PID=`pid`.
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - Permite indicar qué acción se realizará (`act`) cuando el proceso recibe la señal `signum`. La acción previamente configurada se almacena en `oldact`
- `int pause(void);`
  - Bloquea el proceso hasta la recepción de una señal
- `unsigned int alarm(unsigned int seconds);`
  - Genera la recepción de la señal SIGALRM pasados `seconds` segundos
- `int sigprocmask(int how, const sigset_t *set, sigset_t *oset);`
  - Usado para examinar/modificar la máscara de señales del proceso

# Contenido

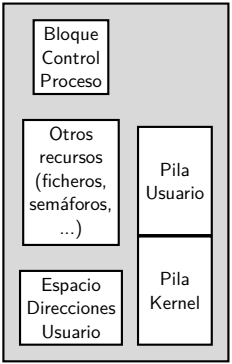
# Hilos o threads

- Por Hilo
  - Contador de programa, Registros
  - Pila
  - Estado (ejecutando, listo o bloqueado)
  - Bloque de control de thread
- Por proceso
  - Espacio de direcciones de memoria
  - Variables globales
  - Ficheros abiertos
  - Procesos hijos
  - Temporizadores
  - Señales y semáforos

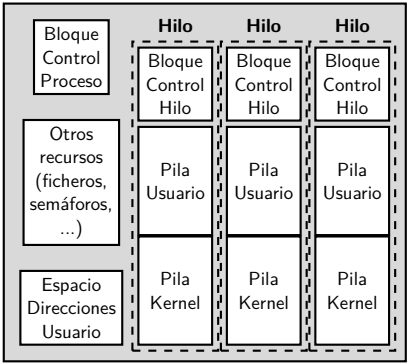


# Mono-hilo vs Multi-hilo

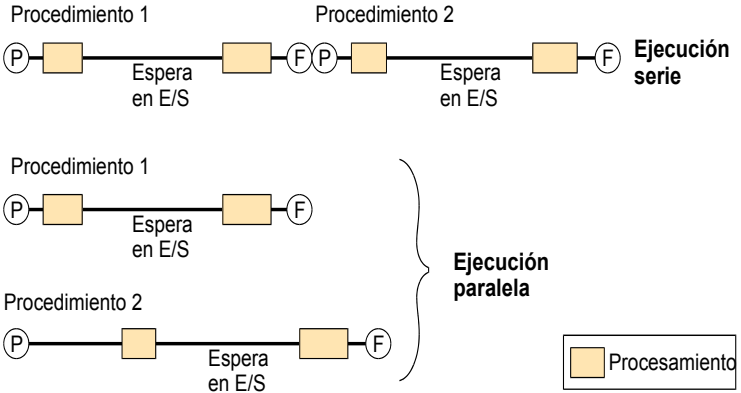
**Modelo de proceso monohilo**



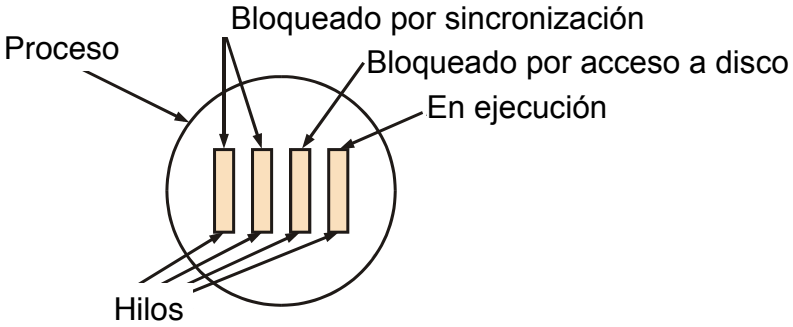

**Modelo de proceso multihilo**



# Paralelización usando hilos



# Estados de un hilo



# Ventajas threads vs. procesos

- Operaciones relacionadas con creación, destrucción, planificación y sincronización son más costosas para procesos que para hilos
  - Hasta un orden de magnitud
- El cambio de contexto entre hilos de un mismo proceso es menos costoso que entre hilos de distintos procesos
  - En el primer caso no es necesario cambiar el espacio de direcciones "activo" de usuario

# Diseño con hilos

- Permite separación de tareas
- Permite división de tareas
- Aumenta la velocidad de ejecución del trabajo
- Paralelismo
- Llamadas al sistema bloqueantes
- Programación concurrente (memoria compartida)
  - Variables o estructuras de datos compartidas
  - Funciones reentrantes
    - Imaginar llamadas simultáneas a la misma función (no necesariamente de forma coordinada)
  - Mecanismos de sincronización entre hilos
    - mutexes, semáforos,...
  - Simplicidad vs. exclusión en el acceso

# Alternativas al diseño multihilo



- Proceso con un solo hilo
  - No hay paralelismo
    - Llamadas al sistema bloqueantes
  - Paralelismo gestionado por el programador
    - Llamadas al sistema no bloqueantes
- Múltiples procesos convencionales cooperando
  - Permite paralelismo
  - No comparten variables
    - Necesario emplear mecanismo de comunicación entre procesos (proporcionado por el SO)
  - Mayor sobrecarga de ejecución

# API de POSIX Threads

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*func)(void *), void *arg)`
  - Crea un hilo que ejecuta `func` con argumento `arg` y atributos `attr`
    - Los atributos permiten especificar: tamaño de la pila, prioridad, política de planificación, etc.
    - Existen diversas llamadas para modificar los atributos
- `int pthread_join(pthread_t thid, void **value)`
  - Suspende la ejecución de un hilo hasta que termina el hilo con identificador `thid`
    - Devuelve el estado de terminación del hilo
- `int pthread_exit(void *value)`
  - Permite a un hilo finalizar su ejecución, indicando el estado de terminación del mismo
- `pthread_t pthread_self(void)`
  - Devuelve el identificador del thread que ejecuta la llamada

# API de POSIX Threads (Cont.)



- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`
  - Establece el estado de terminación de un hilo.
    - Si `detachstate = PTHREAD_CREATE_DETACHED` el hilo liberará sus recursos cuando finalice su ejecución.
    - Si `detachstate = PTHREAD_CREATE_JOINABLE` no se liberarán los recursos, es necesario utilizar `pthread_join()`.

# Programa de ejemplo

```c
#include <stdio.h>
#include <pthread.h>
#define MAX_THREADS 10

void* func(void* arg) {
    printf("Thread %d \n", pthread_self());
    pthread_exit(0);
}

int main(void) {
    int j;
    pthread_attr_t attr;
    pthread_t thid[MAX_THREADS];

    pthread_attr_init(&attr);

    for(j = 0; j < MAX_THREADS; j ++)
        pthread_create(&thid[j], &attr, func, NULL);

    for(j = 0; j < MAX_THREADS; j ++)
        pthread_join(thid[j], NULL);

    return 0;
}
```