Tema 3

GESTIÓN Y COMUNICACIÓN DE PROCESOS

Índice

Índice

Definición Proceso

- Un proceso es la instancia de un programa en ejecución.
  - Una unidad de propiedad de los recursos:
    - un espacio de direcciones virtuales para la imagen del proceso
    - el control de otros recursos (archivos, dispositivos E/S...)
  - Una unidad de ejecución
    - Un proceso es un camino de ejecución a través de uno o más programas
    - Pueden ser intercalados con otros procesos
    - Estado de ejecución (Ready, Running, Blocked,...) y una prioridad de ejecución

Características procesos

- Estas dos características se tratan por separado por en la mayoría de sistemas operativos:
  - La unidad de propiedad de los recursos que normalmente se conoce como un **proceso** o **tarea**
  - La unidad de ejecución se suele hacer referencia a un **hilo** de ejecución o un "**proceso ligero**"

Gestión Proceso

- Gestión proceso por parte del SO:
  - Creación y eliminación.
  - Garantizar la ejecución y finalización.
  - Controlar errores y excepciones
  - Asignación de recursos.
  - Comunicación y sincronización.

- Ejemplo ejecución proceso: vi.

Estructura Procesos

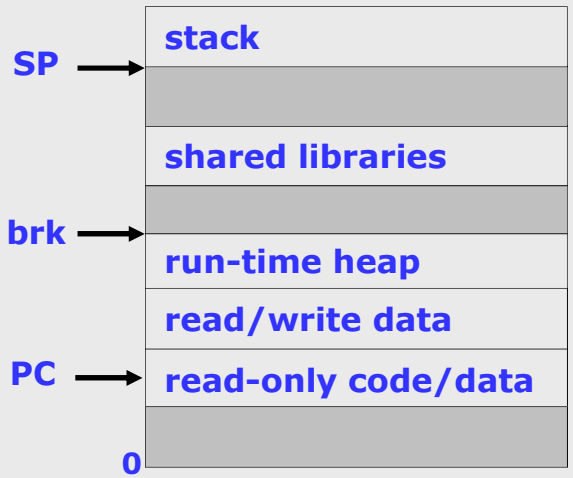
- Un proceso tiene:
  - Un espacio virtual de direcciones que contiene la imagen del proceso
  - Acceso protegido a los procesadores, a otros procesos (comunicación entre procesos), archivos y otros recursos de E/S

Contexto Proceso

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Código, datos, y pila



Estado de los procesos

- **Nuevo**:

  Proceso que todavía no se ha creado del todo (sin PCB).
- **Inactivo**:

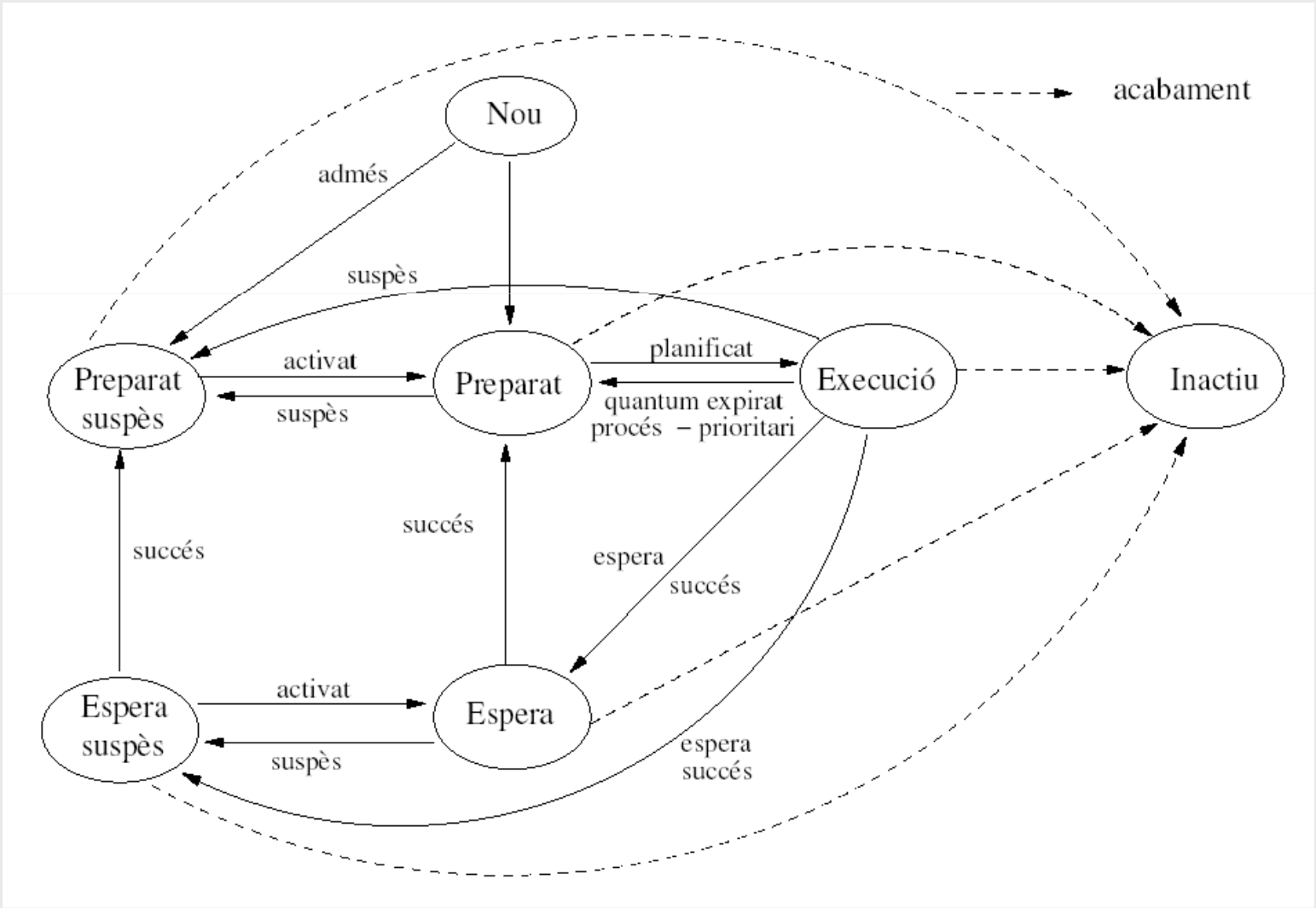
  Cuándo un proceso ha finalizado.
- **Preparado**:

  Cuando un proceso tiene todos los recursos necesarios para poder ejecutarse (excepto la CPU).
- **Ejecución**:

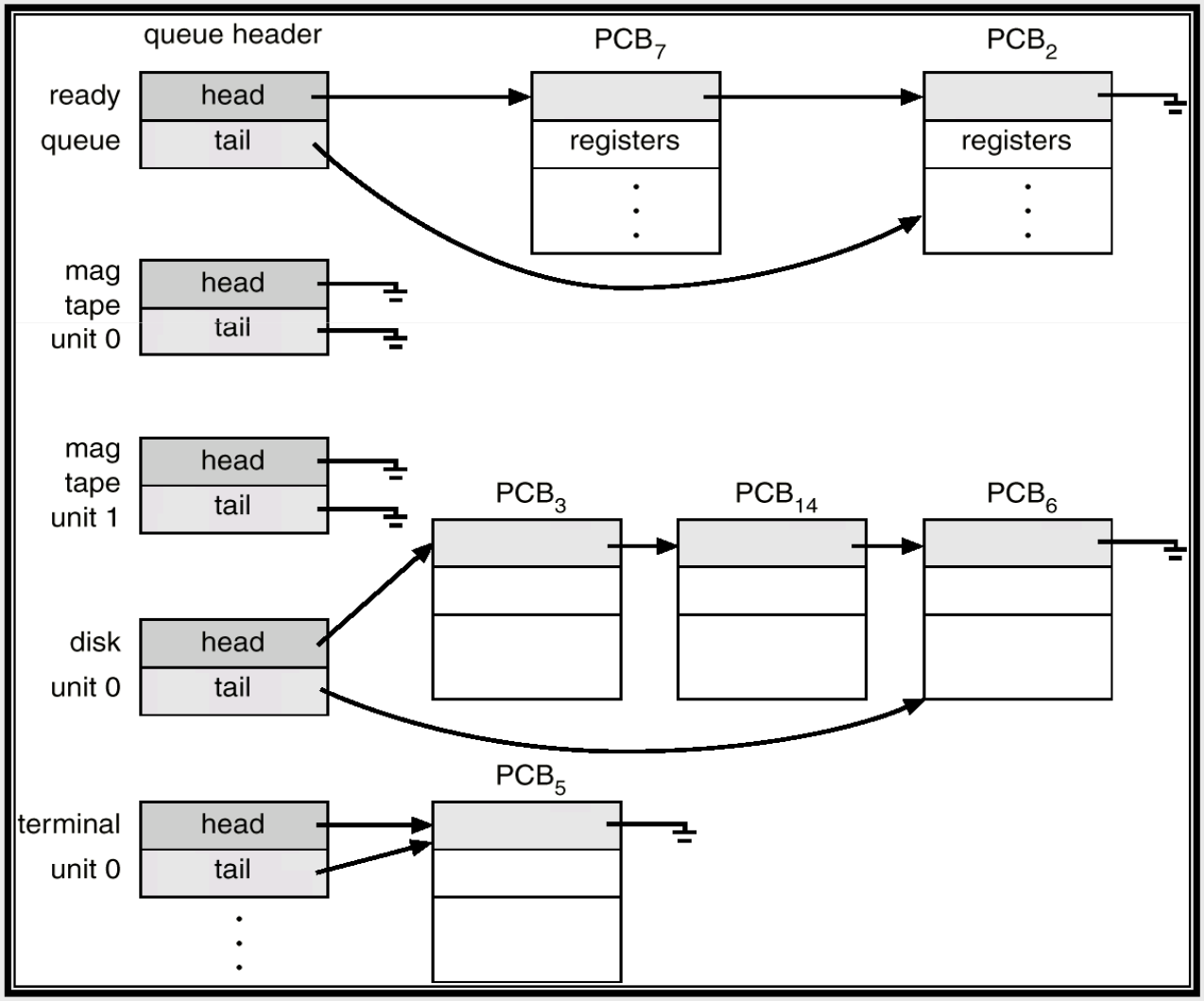
  Cuándo un proceso tienen asignada la CPU.
- **Espera**:

  Cuándo al proceso le falta algún recurso para poder ejecutarse.

Diagrama de transición de estados

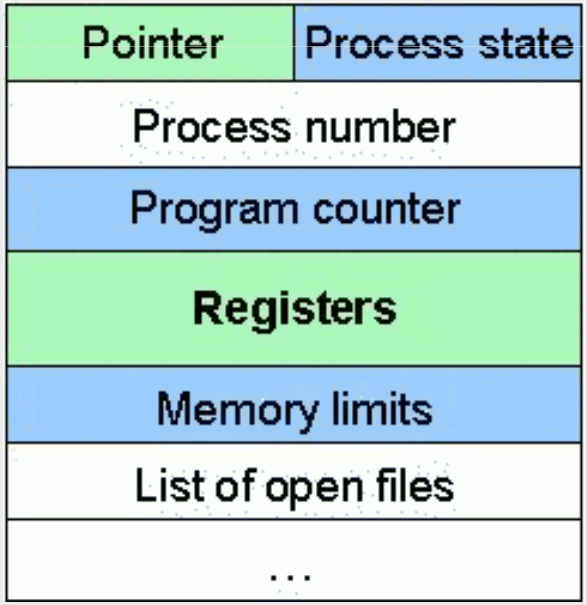


Implementación colas de procesos



PCB (Process Control Block)

- El PCB es una estructura de datos mediante la cual el S.O. supervisa y controla un proceso.
- Información PCB:
  - Punteros.
  - Estado del proceso.
  - Identificadores.
  - Tabla de ficheros abiertos.
  - Recursos asignados.
  - Contexto registros CPU.
  - Información sobre gestión memoria.
  - Información de contabilidad.
  - Información de planificación de CPU.



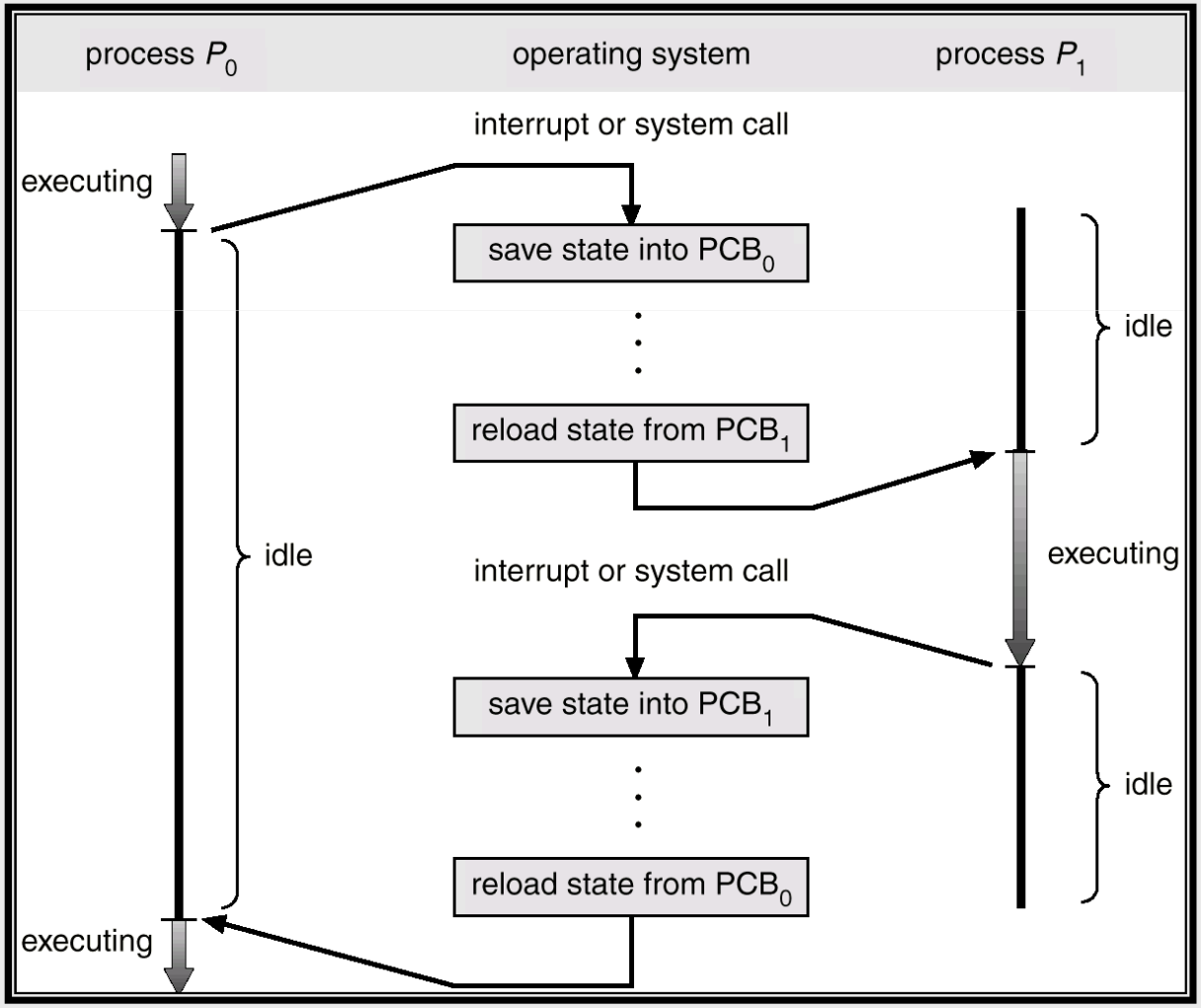
Multiprogramación

- Consiste en la gestión de más de un programa en memoria principal y su ejecución de forma concurrente.
- De esta forma si el proceso que tiene la CPU necesita esperar por un recurso, otro proceso puede aprovecharla mientras tanto.
- Para soportar la multiprogramación los procesos deben encontrarse el estado del procesador tal y como lo dejaron la última vez que se ejecutaron.

Contexto de un proceso

- El contexto de un proceso contiene toda la información necesaria para poder parar y reanudar la ejecución de un proceso:
  - Contexto de usuario.
  - Contexto de registros.
  - Contexto de sistema.
- El contexto de un proceso se salva en su PCB.
- La **conmutación de contexto** es la acción por la cual el SO saca un proceso de la CPU, para colocar a otro.

Intercambio procesos en la CPU



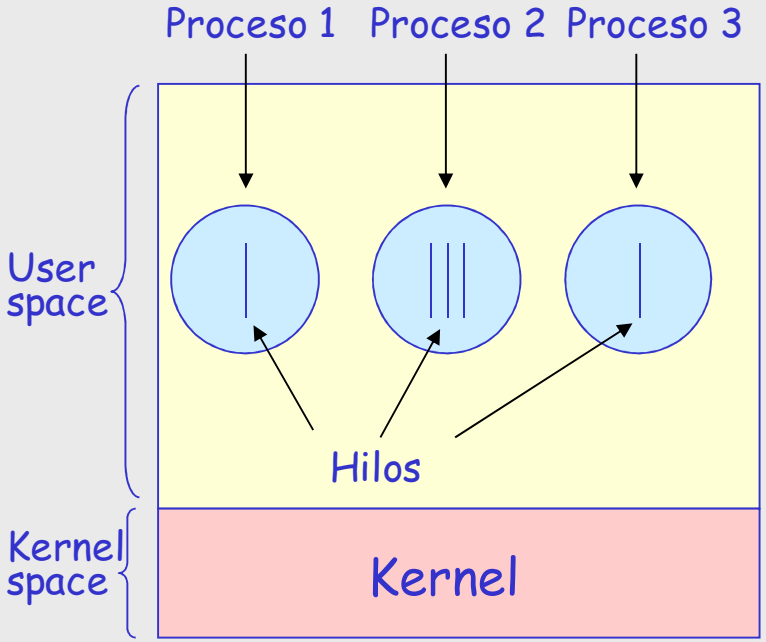
Concurrencia

- La concurrencia es la existencia simultánea de varios procesos en ejecución (no implica ejecución simultánea => paralelismo).
- El objetivo de la programación concurrente es aumentar la eficiencia de la aplicación y con ello la productividad del sistema informático.
- Niveles de concurrencia:
  - A nivel de procesos.
  - A nivel de hilos de ejecución.

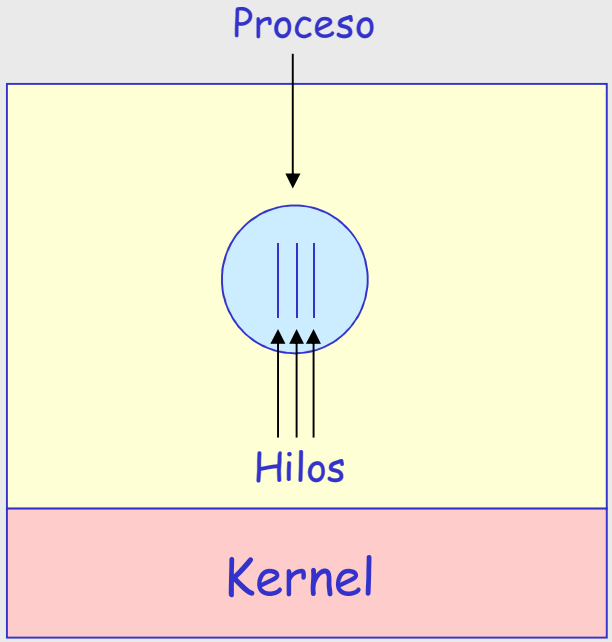
Índice

Hilos de ejecución

- Un hilo ó proceso ligero define una camino de ejecución independiente dentro de un proceso.
- Un proceso puede tener múltiples hilos de ejecución.

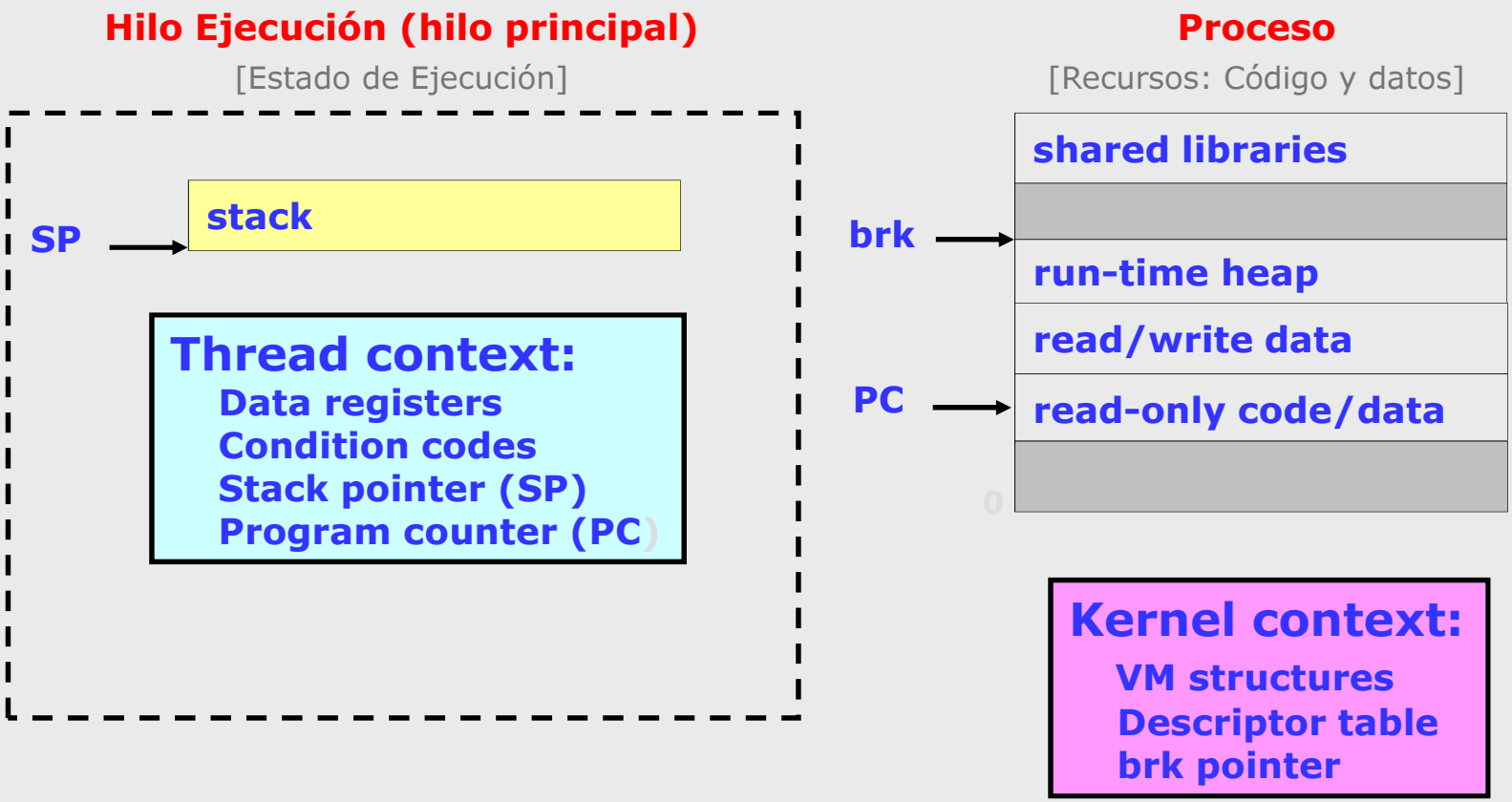


Tres procesos, cada uno con uno o más hilos

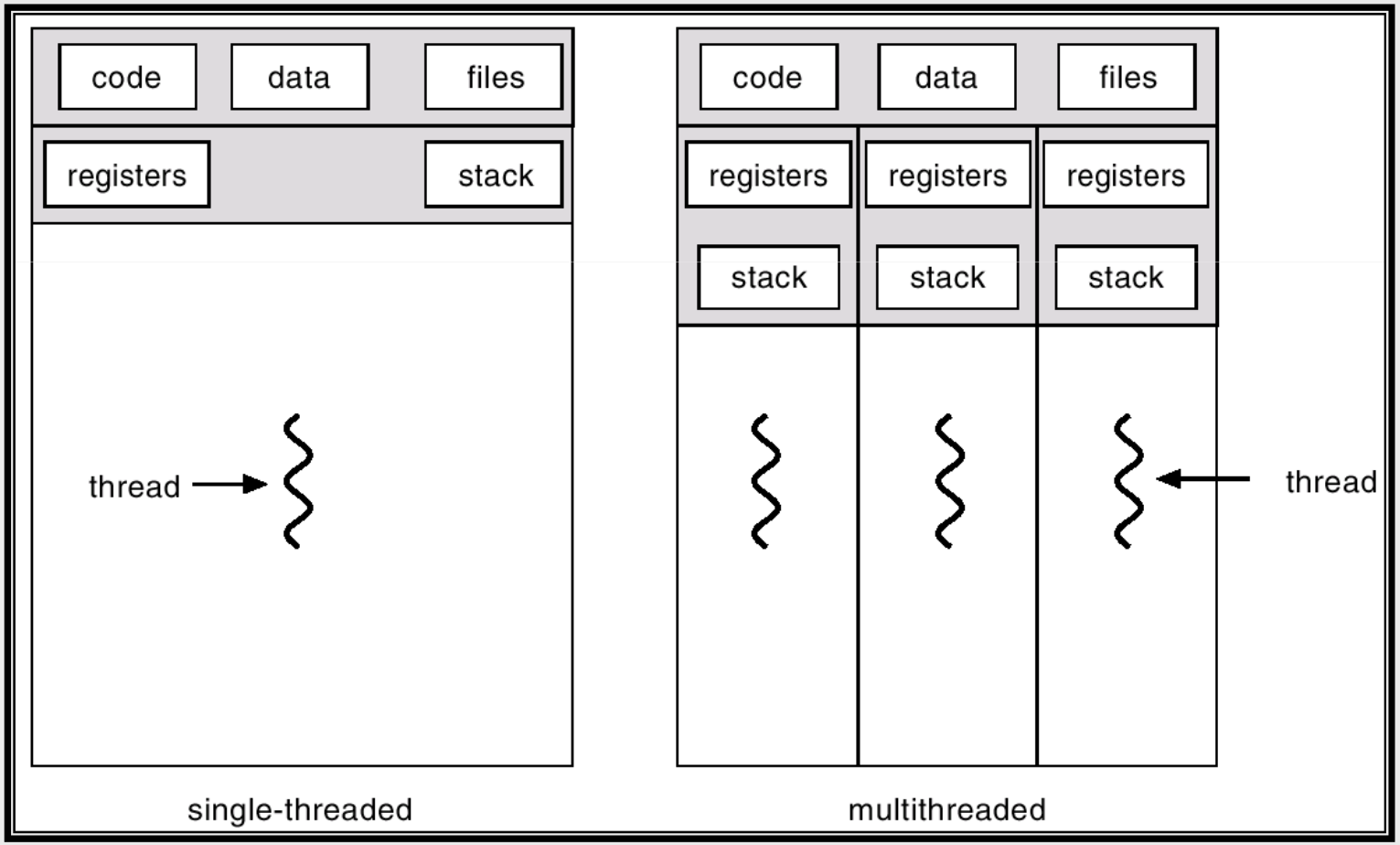


Un proceso con tres hilos

Estructura hilos ejecución



Procesos con uno ó varios hilos de ejecución



Estructura hilos ejecución

- Los hilos de un proceso comparten:
  - Espacio direccionamiento (código/variables globales)
  - Recursos asignados (ficheros abiertos, semáforos,...)
  - Entorno (señales, variables de entorno)
- Cada hilo de ejecución tiene:
  - Estado de ejecución.
  - Contexto del procesador.
  - Pila de ejecución (variables locales).
  - Memoria dinámica (heap).

Hilos & procesos

- Ventajas hilos de ejecución:
  - Los hilos requieren menos recursos del sistema operativo (código/datos/entorno está compartido).
  - Un hilo se puede crear de forma más rápida que un proceso.
  - El cambio de contexto para un hilo es menos costoso.
- Desventajas hilos de ejecución:
  - Los hilos comparten memoria, es necesario sincronizar el acceso a las variables globales para evitar inconsistencias en los datos (desarrollo aplicaciones difícil).
  - No existe protección entre los hilos.

Porque usar hilos

- **Beneficios**
  - Mayor rendimiento
    - Método sencillo para sacar provecho de arquitecturas multi-core
  - Mejor utilización de recursos
    - Reduce latencia (incuso en sistemas con un procesador)
  - Compartición eficiente de los datos
    - Compartir datos a través de la memoria es más eficiente que pasar mensajes
- **Riesgos**
  - Incrementa la complejidad de la aplicación
  - Difícil depurar (condiciones de carrera, interbloqueos, etc.)

Tipos hilos de ejecución (I)

- Hilos de ejecución en el nivel de usuario.

  Los hilos se implementan mediante librerías en el nivel de usuario, sin involucrar al SO, ni requerir llamadas al sistema.

  - Ventaja:

    La conmutación entre hilos en el nivel de usuario se puede realizar de forma independiente del SO y de forma muy rápida.

  - Desventajas:
    - El bloqueo de un hilo en un proceso implica el bloqueo de todos sus hilos.
    - Planificación no equitativa.
  - Ejemplo: POSIX Threads y SUN Threads.

Tipos hilos de ejecución (II)

- Hilos de ejecución al nivel del kernel.

  El kernel del SO soporta directamente los hilos de ejecución.

  - Ventajas:
    - Planificación independiente y equitativa.
    - Bloqueo a nivel de hilos de ejecución.
  - Desventajas:
    - La conmutación entre hilos es más costosa ya que requiere una interrupción y acceder al núcleo.
  - Ejemplo: Windows, Linux, Solaris, ...

Caso de Estudio: Unix

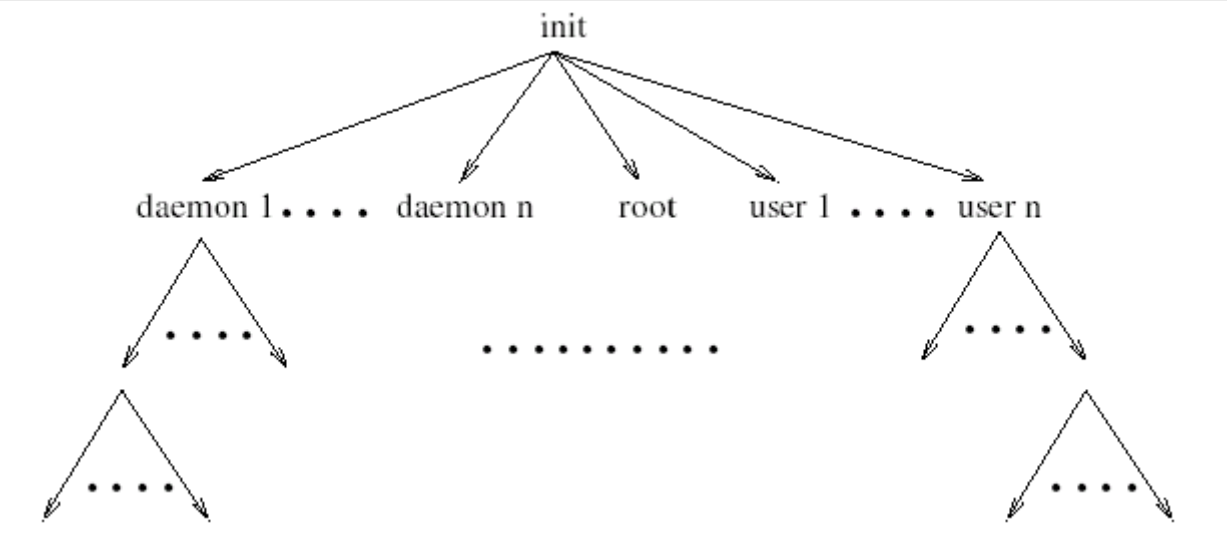
- Gestión de procesos en Unix
  - Se realiza a través de las llamadas al sistema
  - Unix permite a los programas:
    - **Gestión de procesos.**
    - Sincronización procesos / hilos.
    - Gestión de excepciones (señales).
    - Gestión de hilos de ejecución (threads).

Gestión de procesos

- Un proceso es la entidad que utiliza Unix para modelar un programa en ejecución.
- Unix es un sistema operativo multitarea con lo que ha de proveer infraestructura para ejecutar un número arbitrario de procesos al mismo tiempo.
- Esto lo realiza a dos niveles:
  - A nivel del kernel del sistema operativo soportando la multiprogramación.
  - A nivel del entorno de ejecución (llamadas de sistema) de forma que los programadores puedan acceder a estas características.

Árbol de procesos Unix

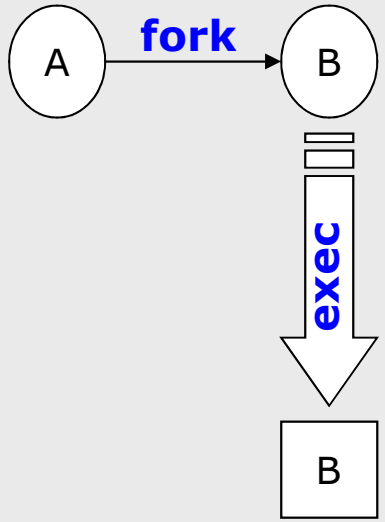
- Cuando el sistema arranca, crea automáticamente el proceso init.
- El proceso init es el responsable de crear (mediante fork y exec) todos los procesos iniciales del sistema.



Pasos de creación de un proceso

- Supongamos que el proceso A (○) quiere crear el proceso B, para ejecutar otro programa (□).

  1. **Fork**: El proceso A llama a la función fork y crea una copia exacta de si mismo. El nuevo proceso B tiene un PID diferente a A.
  2. **Exec**: El proceso B ejecuta la función exec y reemplaza el programa en ejecución por otro especificado.



Creación de procesos

- La única forma de crear un proceso en Unix es mediante la llamada de sistema fork().

```
pid = fork()
```

- La llamada fork hace que el proceso actual se duplique.
- Los dos procesos resultantes son una copia casi idéntica, únicamente difieren en el valor devuelto por la llamada fork():
  - Al proceso padre se le devuelve el pid del proceso creado.
  - Al proceso hijo devuelve el pid nulo.
  - En caso de error devuelve pid = -1 y errno tendrá el código del error producido.

Ejemplo fork (I)

```
if ((pid=fork ()) == -1)
{
   /* Error en la creación del proceso hijo */
   perror("Error en la llamada fork");
}
else if (pid==0)
{
   /* Código que tiene que ejecutar el proceso hijo */
}
else /* pid>0 */
{
   /* Código que tiene que ejecutar el proceso padre */
}
```

Similitudes entre procesos duplicados

- El código es el mismo.
- El proceso hijo hereda los ficheros abiertos del proceso padre.
  - Padre e hijo comparten la sesiones de trabajo con estos ficheros (posición escritura, buffer, etc....)
- Directorio actual del proceso es el mismo.
- Los valores iníciales de las variables globales y locales del hijo se corresponde con los valores que tenían esas variables antes de crear el proceso hijo.

Diferencias entre procesos duplicados

- El identificador de proceso (pid).
- Mapa de memoria es diferente (código, datos, pila).
  - Los segmentos de memoria de datos son diferentes, por lo tanto una vez se reanude la ejecución de ambos procesos la modificación de cualquier variable afectará únicamente al proceso que la realicé.
  - La pila del proceso es diferente.
- El valor de retorno de la función fork().

Ejemplo fork (II)

```
main() {
    int i=0;  char *mensaje="Prueba fork: ";
    switch ( fork() )
    {
        case –1:  /* Código de control de errores */
            perror("Error al crear el proceso hijo");  exit (1);
            break;
        case 0:    /* Código del hijo */
            while (i<10) printf ("%s. Soy el hijo %d.\n", mensaje,i++);
            exit(0);
            break;
        default:   /* Código del padre */
            i=10;
            while (i<100) printf ("%s. Soy el padre %d.\n", mensaje,i++);
            wait(NULL);
            break;
    }
}
```

¿Valor inicial variable i en ambos procesos?

¿Valor final variable i ambos procesos?

¿Algún otro problema en alguno de los procesos?

Ejercicio: Jerarquía Procesos

- A partir del siguiente código:

```
1.  int x=1, pid;
2.  pid=fork();
3.  if (pid!=0){
4.        x++;
5.        pid=fork();
6.  }
7.  x++;
8.  fork();
9.  sprintf(msg,“Proceso %d -> X=%d.\n",getpid(),x);
10. write(1,msg,strlen(msg));
```

  - Se pide:
    1. ¿Cuántos procesos tiene la aplicación?
    2. ¿Qué jerarquía de procesos?
    3. ¿Que mensajes se imprimen por pantalla?

Terminación de procesos

- Cuando un proceso termina en Unix, ejecuta una rutina especial para terminar, notifica al sistema y devuelve un código de salida que identifica la razón por la que el proceso ha terminado.
- En Unix, antes de que un proceso desaparezca totalmente se requiere que su terminación sea reconocida por el proceso padre.
- Este reconocimiento se realiza mediante las llamadas al sistema **wait** y **exit**.

exit

- La llamada exit termina la ejecución de un proceso, devolviendo el valor de retorno al proceso padre.
- El retorno desde la función principal (main) produce el mismo efecto que la llamada exit.
- Sintaxis:

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

Consecuencias exit

- El contexto del proceso se descarga de memoria.
- La tabla de descriptores de ficheros del proceso y los ficheros abiertos son cerrados siempre que no existan más procesos que los estén utilizando/compartiendo.
- Si el proceso padre está ejecutando una llamada wait se le notifica la terminación del proceso hijo y se le envían los 8 bits menos significativos de status (código de retorno).
- Si el proceso padre no está ejecutando una llamada wait el hijo se queda en estado **zombie** esperando la sincronización con el padre.

wait

- La llamada a sistema wait suspende la ejecución del proceso que la invoca hasta que alguno de sus procesos hijos termine.
- La llamada devuelve el pid del proceso hijo finalizado y el estado de finalización en el parámetro de entrada.
- Sintaxis:

```
pid_t wait (int *status);
```

- También se puede suspender la ejecución para esperar la finalización de un proceso en concreto, mediante la llamada al sistema waitpid.
- Sintaxis:

```
pid_t waitpid (pid_t pid,int *status);
```

Relación exit y wait

- Si el proceso hijo ha finalizado mediante la llamada exit(status) los 8 bits más significativos contienen el valor de status.
- Si el proceso hijo se ha parado, los 8 bits más significativos contienen el identificador de la señal que ha provocado la parada del proceso.
- Si el proceso hijo ha finalizado debido a la llegada de una señal los 8 bits más significativos contienen contienen el id. de la señal.

Ejemplo wait/exit

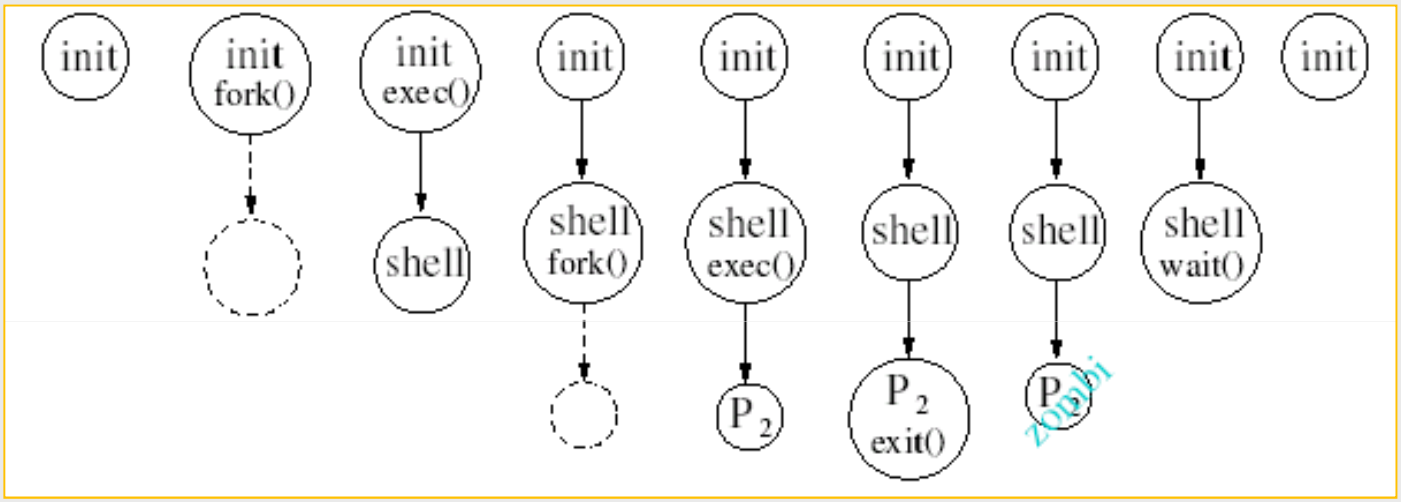
```
if ((pid=fork ()) == -1) {
    /* Error en la creación del proceso hijo */
    perror("Error en la llamada fork");
}
else if (pid==0) {
    /* Código que tiene que ejecutar el proceso hijo */
    exit(10);
}
else {  /* pid>0 */
    /* Código que tiene que ejecutar el proceso padre */
    pid = wait (&estado);
}
```

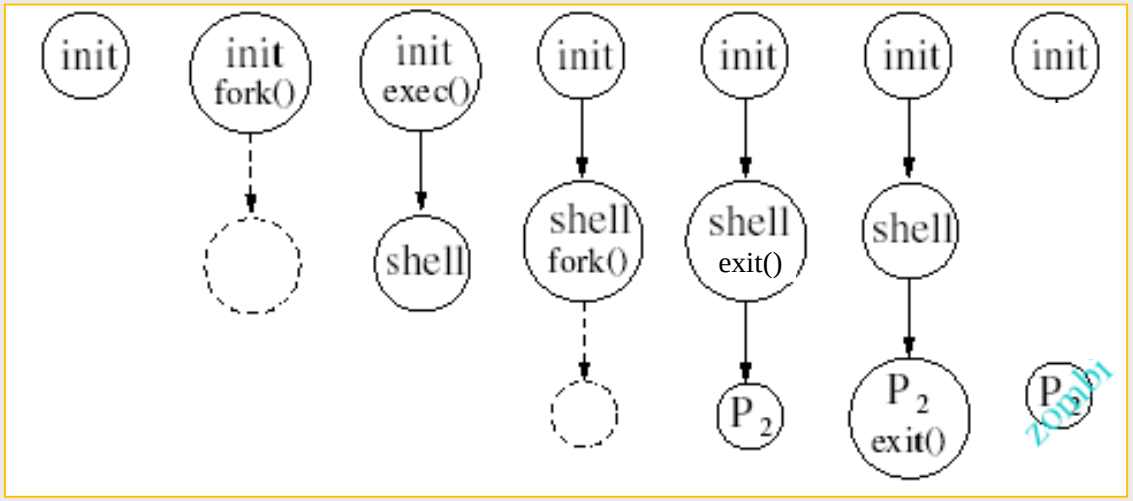
Procesos zombies

- ¿Que es un proceso Zombie?

  Un proceso zombie es un proceso que ha terminado, pero cuyo proceso padre no ha confirmado esta terminación.
- ¿Cuándo se produce un Zombie?
  - Cuando el proceso hijo finaliza pero el proceso padre (aún en ejecución) todavía no ha realizado el wait.
  - Cuando el proceso padre finaliza antes que el hijo y sin realizar un wait.
- ¿Consecuencias?
  - Con la primera causa el estado de zombi no es usual y dura poco tiempo.
  - Con la segunda causa, si el SO no lo controla el proceso permanece en estado de zombie indefinidamente, pudiéndose llegar a saturar el sistema con infinidad de procesos zombies.

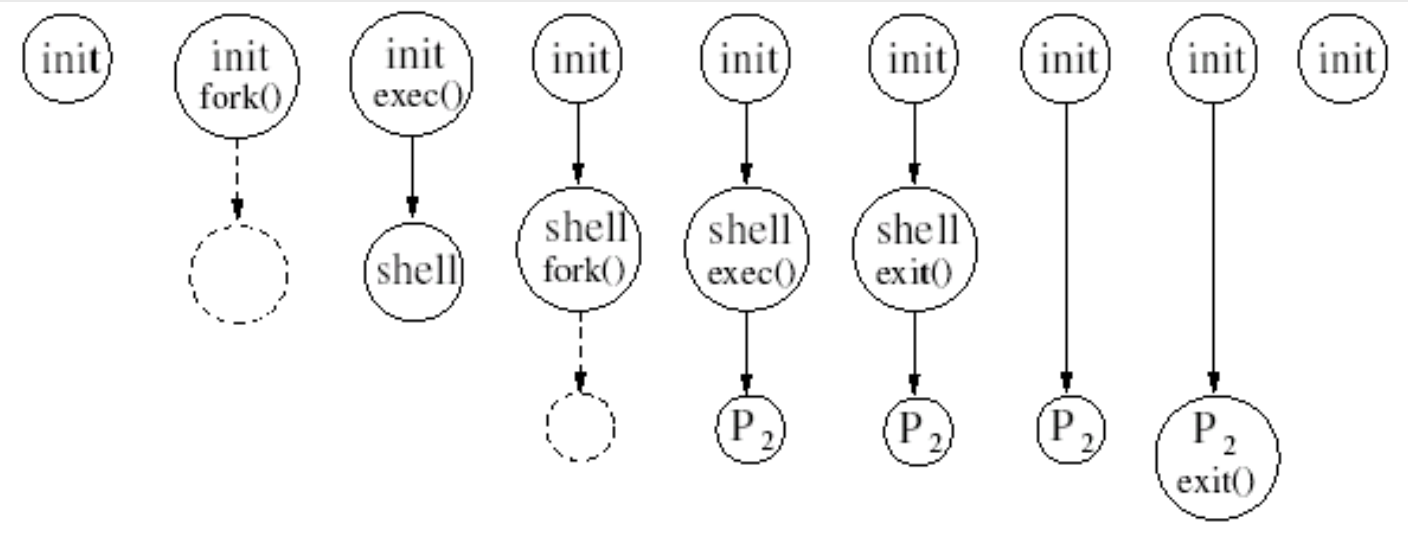
Ejecución con proceso Zombie





Solución procesos zombies

- El sistema operativo detecta esta condición y hace que los procesos hijos pasen a depender del proceso init.
- Podría decirse que el proceso init “adopta” a los procesos huérfanos (cuyo padre a muerto antes que ellos).



Ejercicio: Creación Zombis

1. Implementar un programa en C que genere estos dos tipos de procesos zombis.
   - Podéis utilizar las siguientes llamadas al sistema:
     - `pid_t fork(void);`
     - `void exit(int);`
     - `pid_t wait(int *status)`
     - `unsigned int sleep(unsigned int segundos);`
1. ¿Qué duración tienen dichos procesos zombis?
2. ¿Como evitar su aparición?

Consulta características procesos

- **`int getpid(void)`**

  Devuelve el pid del proceso.
- **`int getppid(void)`**

  Devuelve el pid del proceso padre.
- **`int getuid(void)`**

  Devuelve el identificador de usuario real propietario del proceso.
- **`int geteuid(void)`**

  Devuelve el identificador de usuario efectivo propietario del proceso.
- **`int getpgrp(void)`**

  Devuelve el identificador del grupo pid del proceso padre.
- **`int getgid(void)`**

  Devuelve el identificador de grupo real del proceso.
- **`int getegid(void)`**

  Devuelve el identificador de grupo efectivo del proceso.

Modificación características procesos

- **`int setpgid(int id,int gid)`**

  Actualiza el identificador de grupo de procesos.
- **`int setuid(int id)`**

  Actualiza el identificador de usuario real y efectivo del proceso (solo root).
- **`int seteuid(int id)`**

  Actualiza el identificador de usuario efectivo del proceso.
- **`int setruid(int id)`**

  Actualiza el identificador de usuario real.
- **`int setgid(int id)`**

  Actualiza el identificador de grupo real y efectivo del proceso.
- **`int setegid(int id)`**

  Actualiza el identificador de grupo efectivo del proceso.
- **`int setrgid(int id)`**

  Actualiza el identificador de grupo real.

Ejercicio: Procesos

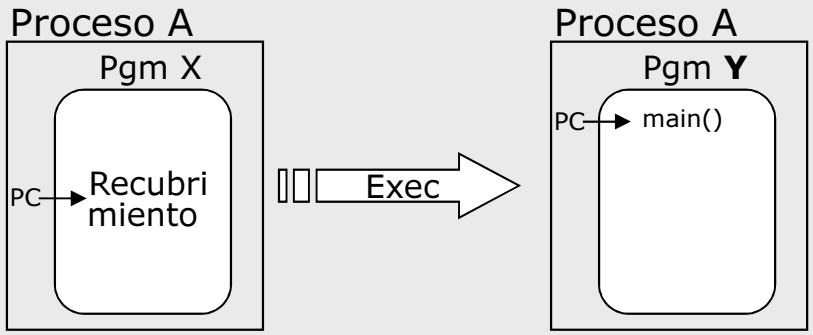
- Donat el codi següent expliqueu raonadament quants cops s'executa l'ordre wait, i quants d'aquests cops la seva execució retorna un -1.

```
1.  id = fork();
2.  wait(st);
3.  if (id == 0)
4.          fork();
5.  wait(st);
```

a) ¿Número de procesos?
b) ¿Jerarquía de procesos?
c) ¿Número de invocaciones a wait? ¿Cuántas fallan y porque?

Recubrimiento (exec)

- El procedimiento de recubrimiento consiste en cargar un programa en la zona de memoria del proceso que ejecuta la llamada.
- Se sobrescriben los segmentos del programa antiguo con los del nuevo.
- Cuando se produce un recubrimiento se inicia la ejecución del nuevo programa a partir de su función main, recibiendo los parámetros de entrada especificados.



Exec (I)

- El recubrimiento se puede realizar utilizando la familia de llamadas exec de la librería unistd.h.
- Sintaxis:

```
int execl(const char *camino, const char *arg,
...);
int execv(const char *camino, char *const
argv[]);
```

- Los parámetros se pasan en formato cadena.
  - execl permite especificar cada uno de los argumentos individualmente
  - execv se agrupan todos los parámetros en un vector.

Exec (II)

- Las llamadas exec* solo devuelven el control al programa que las invoca en caso de error. En caso contrario se ejecuta el nuevo programa.
- En caso de error, se devuelve un -1 y en errno estará el código de error producido.
- El programa recibe los valores de entrada a través de los parámetros argc y argv de la función main.
  - En unix, el primer parámetro (argv[0]) siempre contiene el nombre del programa a ejecutar.

Ejemplo recubrimiento

```
/*  Programa que permite ejecutar un comando de forma
    diferida */
void main(int argc, char *argv[])
{
    /* Retraso la ejecución del comando  tantos segundos
    como indique el primer argumento */
    sleep(atoi(argv[1]));

    /* Ejecutamos el argumento introducido por el usuario */
    execv(argv[2], &argv[2]);
    perror("No se ha podido ejecutar el comando");
    exit(-1);
}
```

Índice

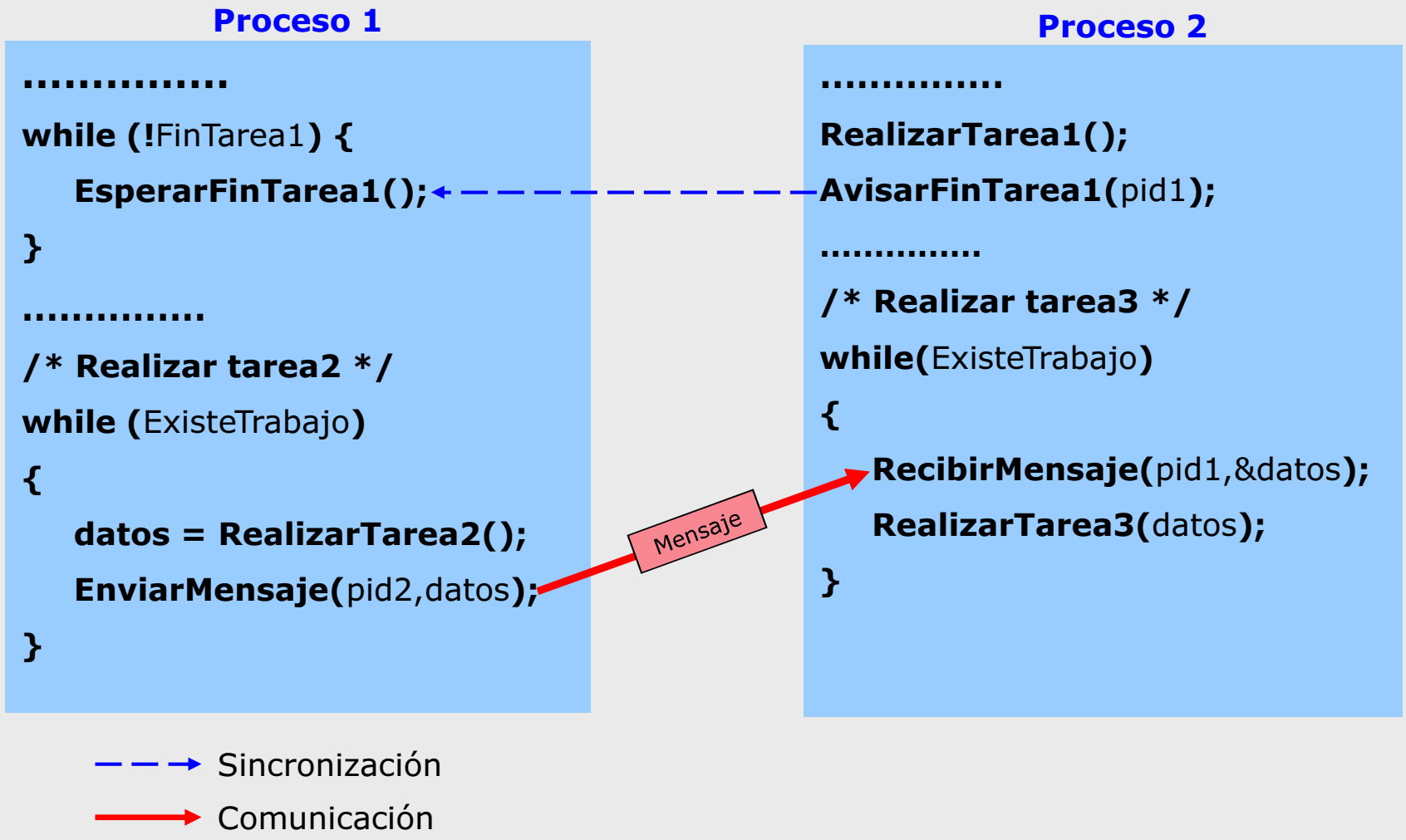
Tipos de procesos

Independientes	Cooperantes
Su estado no es compartido	Su estado es compartido
Son deterministas	Su funcionamiento no es determinista
Son reproducibles	Su funcionamiento puede ser no reproducible
Compiten por los recursos	Comparten recursos
Ejemplo: programa que calcule los 1000 primeros números primos	Ejemplo: programa distribuido para la simulación del clima

Comunicación & sincronización

- En los sistemas multiprogramados/distribuidos los procesos cooperan en el cálculo y la realización de tareas, compartiendo recursos e información.
- En estos sistemas los procesos pueden:
  - Intercambiar información entre ellos (comunicación),
  - Acceder a recursos compartidos ó ponerse de acuerdo en el orden de realización de las distintas tareas (sincronización).
- **Comunicación**: Intercambio de información (al menos un byte).
- **Sincronización**: Ponerse de acuerdo para realizar una determinada tarea ó acceder a un recurso compartido.

Esquema sincronización / comunicación



Esquemas de comunicación entre procesos

- **Paso de mensajes**.
  - Mecanismo que permite integrar tareas de sincronización y comunicación entre procesos situados en una misma máquina ó en máquinas distribuidas.
  - Un mensaje es un conjunto de datos intercambiado por dos ó mas procesos.
- **Memoria compartida**.
  - Los procesos se comunican a través de variables ó zonas de memoria compartida.
  - Se utiliza para comunicar entre si procesos en una misma máquina ó sistema multiprocesador.
  - Se necesita controlar el acceso a la información compartida (por software ó semáforos) para garantizar el resultado correcto de la ejecución concurrente (condiciones de carrera).

Características mecanismos de comunicación

- Identificación
  - Mecanismos de nombrado
    - Sin nombre
    - Con nombre local
    - Con nombre de red
  - Identificador destino
    - Directo
    - Indirecto
- Flujo de datos
  - Unidireccional
  - Bidireccional
- Buffering
  - Sin buffering
  - Con buffering
- Sincronización
  - Síncrono (bloqueante)
  - Asíncrono (no bloqueante)

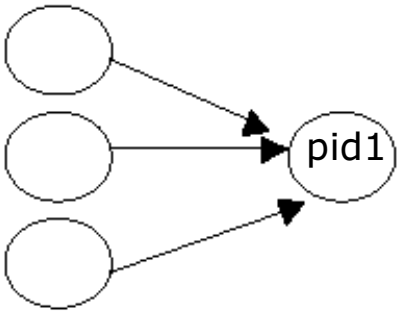
Identificación destino

- **Directa**: Se especifica explícitamente el proceso origen y el destino de la comunicación.

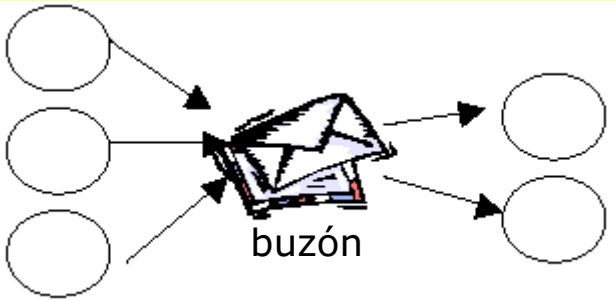
  El mensaje se envía a un proceso concreto (especificando su pid). El receptor puede indicar ó no el proceso del cual quiere recibir un mensaje.

- **Indirecta**: Los mensajes se envían a una zona intermedia (buzón ó puerto), sin identificar explícitamente el destino.

send(*pid1, mensaje*)



send(*buzón, mensaje*)



Sincronización comunicaciones

- **Síncrona**:
  - El emisor y el receptor tienen que finalizar la transferencia de información en el mismo momento.
  - El emisor y el receptor se quedan bloqueados hasta que la transmisión finaliza.
  - Es fácil de implementar y requiere pocos recursos.
- **Asíncrona**:
  - El emisor y el receptor transmiten y reciben el mensaje en distintos instantes de tiempo.
  - No se bloquea el proceso emisor y el SO tiene que guardar el mensaje temporalmente hasta que el receptor lo reclame.
  - Aumenta el grado de concurrencia, pero requiere la gestión del almacenamiento temporal para los mensajes.

Mecanismos de comunicación

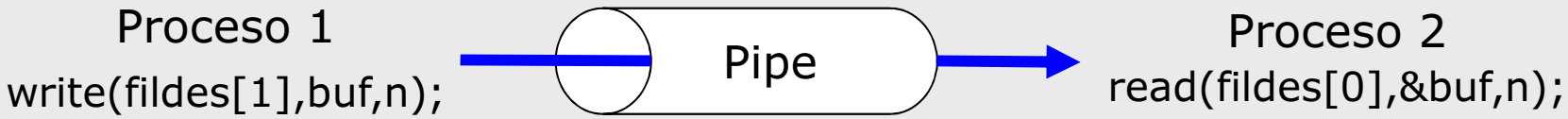
- Ficheros
- Pipes
- FIFOS (pipes con nombre)
- Cola de mensajes
- Sockets
- Memoria compartida (IPC)

Pipes (tuberias)

- Mecanismo de comunicación y sincronización sin nombre
  - Sólo puede utilizarse entre procesos que tengan parentesco
- Identificación indirecta: dos descriptores de fichero

```
int pipe(int fildes[2]);
```

- Flujo de datos: unidireccional
- Con buffering
- No trabajan a través la red (misma máquina)



FIFOS

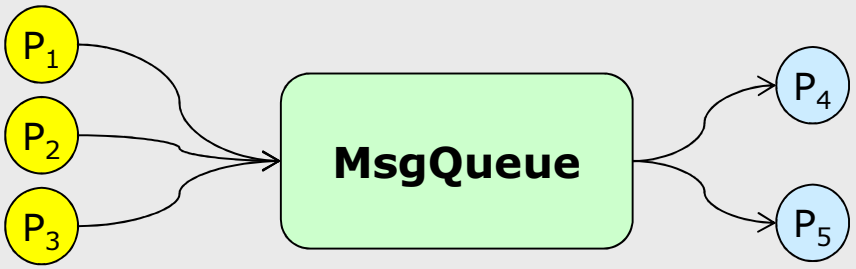
- Igual que los pipes
- Mecanismo de comunicación y sincronización **con nombre** local
  - Ruta y nombre archivo.
- Misma máquina

Servicios

- `mkfifo(char *name,mode_t mode);`
  - Crea un FIFO con nombre *name*
- `open(char *name, int flag);`
  - Abre un FIFO (para lectura, escritura o ambas)
  - Bloquea hasta que haya algún proceso en el otro extremo
- Lectura y escritura mediante *read()* y *write()*
  - Igual semántica que los pipes
- Cierre de un FIFO mediante *close()*
- Borrado de un FIFO mediante *unlink()*

Cola de mensajes

- Comunicación mediante paso de mensajes de la librería IPC.
  - Misma máquina
  - Identificación: indirecta con identifcador especial (IdCola)
  - Mecanismo de nombrado: con nombre local
  - Buffering: Sí
  - Unidireccional
  - Sincronización: bloqueante y no bloqueante



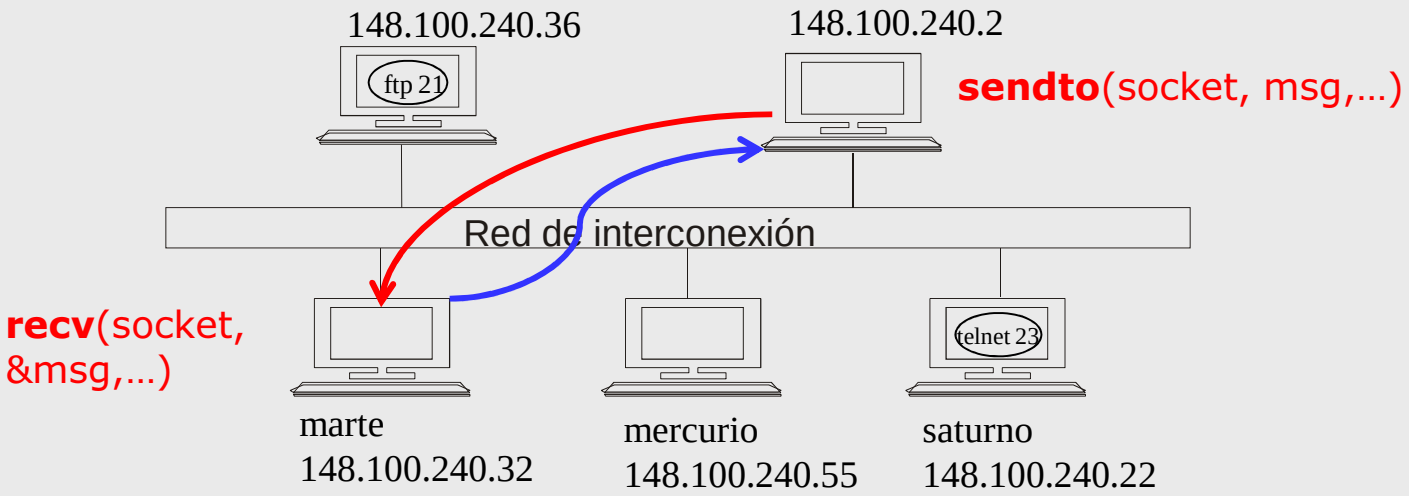
- Primitivas básicas:
  - **int msgsnd(int** *msqid*, **const void** **msgp*, **size_t** *msgsz*, **int** *msgflg*);**

    Envía el mensaje msgp con tamaño *msgsz* a la cola *msgid*
  - **ssize_t msgrcv(int** *msqid*, **void** **msgp*, **size_t** *msgsz*, **long** *msgtyp*, **int** *msgflg*);**

    Recibe un mensaje de la cola *msgid y lo guarda* en msgp.

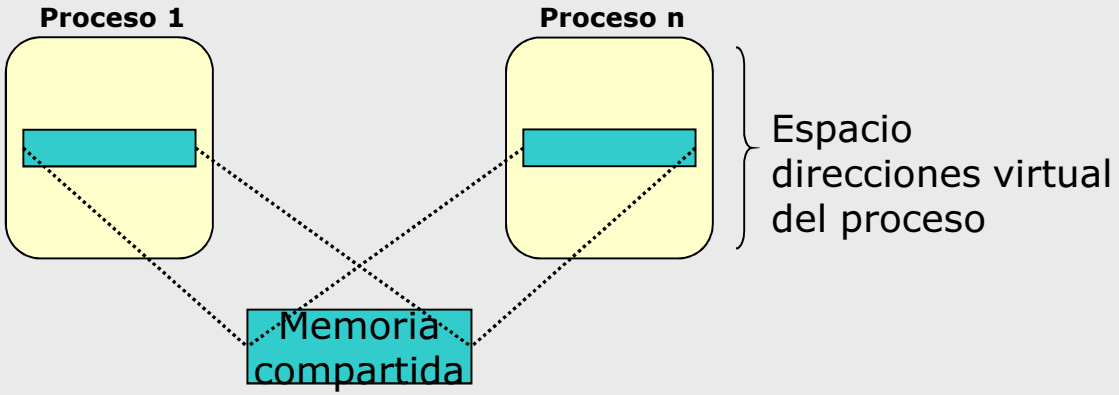
Sockets

- Mecanismo de paso mensajes que permite comunicar:
  - Procesos del mismo computador
  - Procesos conectados a través de una red
- Tipos de direcciones:
  - Direcciones locales: dominio UNIX
  - Direcciones de red (TCP/IP)
    - Dirección de red (dirección IP) + Puerto



Memoria compartida

- La comunicación por memoria compartida:
  - **Espacio de direcciones único**: hilos de ejecución de un mismo proceso.
  - **Múltiples espacios de direcciones**: zona de memoria accesible por varios procesos.
- En ambos casos hay que controlar/sincronizar el acceso a los datos compartidos para asegurar la consistencia de los mismos (evitar las condiciones de carrera)



Mecanismos de Sincronización

- Construcciones de los lenguajes concurrentes (join, …)
- Servicios del sistema operativo:
  - Señales
  - Pipes
  - FIFOS
  - Paso de mensajes
  - Semáforos
  - Mutex y variables condicionales

Caso de Estudio:Comunicación y sincronización en Unix

- **Pipes (paso de mensajes)**
- Señales
- IPC (Inter-Process Comunication)
  - Cola Mensajes (paso de mensajes)
  - Memoria Compartida
  - Semáforos (sincronización)

Pipes (Tuberías)

- Un pipe es un dispositivo lógico destinado a comunicar y sincronizar procesos.
- La comunicación mediante las tuberías es unidireccional y unioperacional, solo permite realizar una operación (lectura/escritura) al mismo tiempo.
- Funciona como una cola FIFO de caracteres de longitud fija en la cual los procesos pueden leer ó escribir.

Pipes (Tuberías)

- Los pipes se implementan dentro del sistema operativo como fichero regular del tipo FIFO (el primer carácter que entra es el primero que se lee).
- Los pipes no tienen un nombre asociado que sea visible en el sistema de ficheros.

Creación pipes

- Los pipes se crean en el momento que se abre mediante la llamada al sistema pipe (unistd.h)

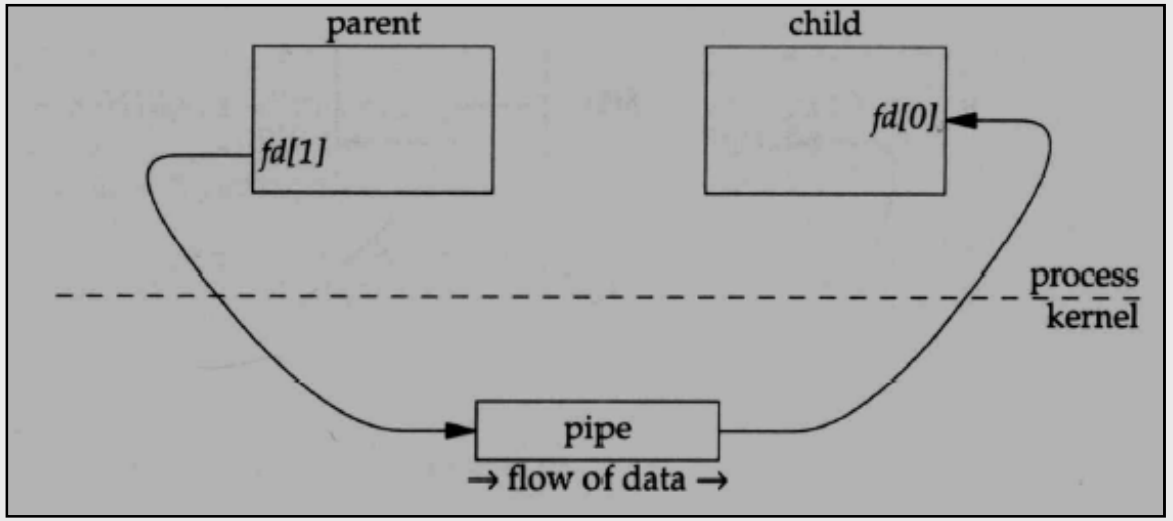
  Sintaxis:

  int pipe(int desc[2]);

- La llamada a pipe crea dos descriptores de fichero: uno de lectura (desc[0]) y otro de escritura (desc[1]).
- La información que se escribe en desc[1] se lee por desc[0]
- Una vez creado el pipe, los procesos que los quieran utilizar deben heredar estos descriptores de su padre.

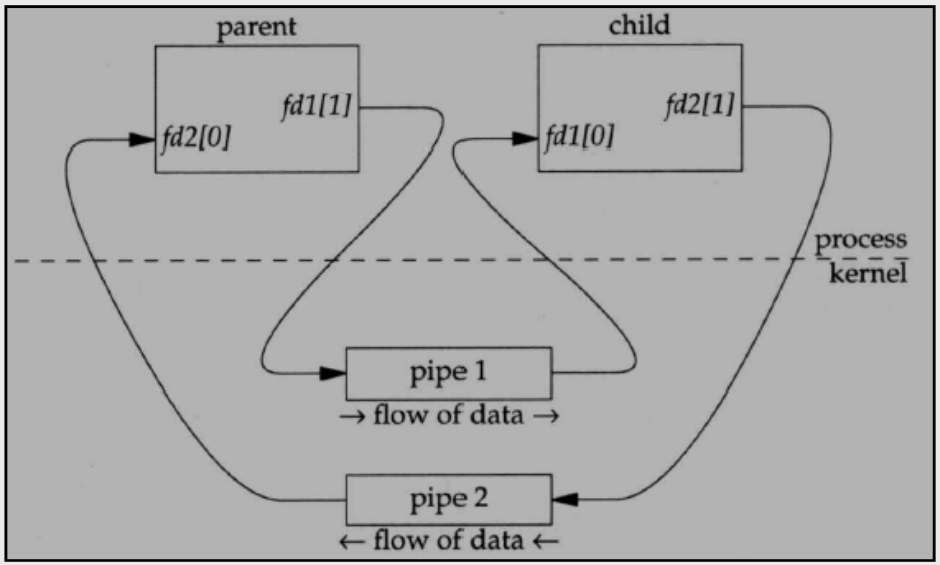
Comunicación unidireccional

- Pasos a realizar:
  - El proceso padre crea una pipe.
  - Se llama a fork creando una copia del padre (y duplica los descriptores de ficheros asociados al pipe)
  - El proceso padre cierra la lectura de la salida del pipe.
  - El proceso hijo cierra la escritura a la entrada del pipe.



Comunicación bidireccional

- Pasos a realizar:
  - Crear pipe1 (fd1[0] y fd1[1]) y pipe2 (fd2[0] y fd2[1]).
  - El padre cierra la lectura desde la salida del pipe1.
  - El padre cierra la escritura a la pipe2.
  - El hijo cierra la lectura desde la salida del pipe2.
  - El padre cierra la escritura a la pipe1.



Escritura de un pipe

- La escritura en los pipes se realiza utilizando las llamadas a sistema estándar de escritura a ficheros: write(), fprintf(), ect...
  - Si un proceso escribe en una tubería que tenga el descriptor de lectura cerrado, la llamada write falla y se envía la señal SIGPIPE.
  - Si un proceso escribe menos bytes de los que admite el pipe, la escritura se realizará de forma atómica.
  - Si se intenta escribir sobre un pipe lleno el proceso se queda bloqueado hasta que aparezca hueco en el pipe (debido a que otro proceso haya leído datos del pipe) que le permita finalizar la escritura.

Lectura de un pipe

- La lectura de los pipes se realiza utilizando también las llamadas a sistema estándar de lectura de ficheros: read(), fscanf(), ect…
  - Si un proceso lee de un pipe que tiene el descriptor de escritura cerrado se devuelve un 0, indicando la condición de final de fichero.
  - Si un proceso lee un pipe vació se bloquea hasta que los datos esten disponibles.
  - Si un proceso intenta leer más datos que los disponibles en un pipe, se leen los datos disponibles y se devuelve el número de bytes leídos.

Ejemplo pipes: Filtrado (I)

```
main ()
{
    int fp[2];  char *c="99 76 15\n";
    if (pipe(fp)==-1) {
        perror("Creación Pipe:"); exit(-1); }
    switch (fork())
    {
        case 0:  /* Hijo */
        /* Cerramos descriptores no utilizados */
        close(fp[1]);
        while (read(fp[0], &num, sizeof(int))>0)
        {
            if (num%2) printf(" %d \n", num);
        }
        close(fp[0]);
        exit(-1);
    }
```

Ejemplo pipes: Filtrado (II)

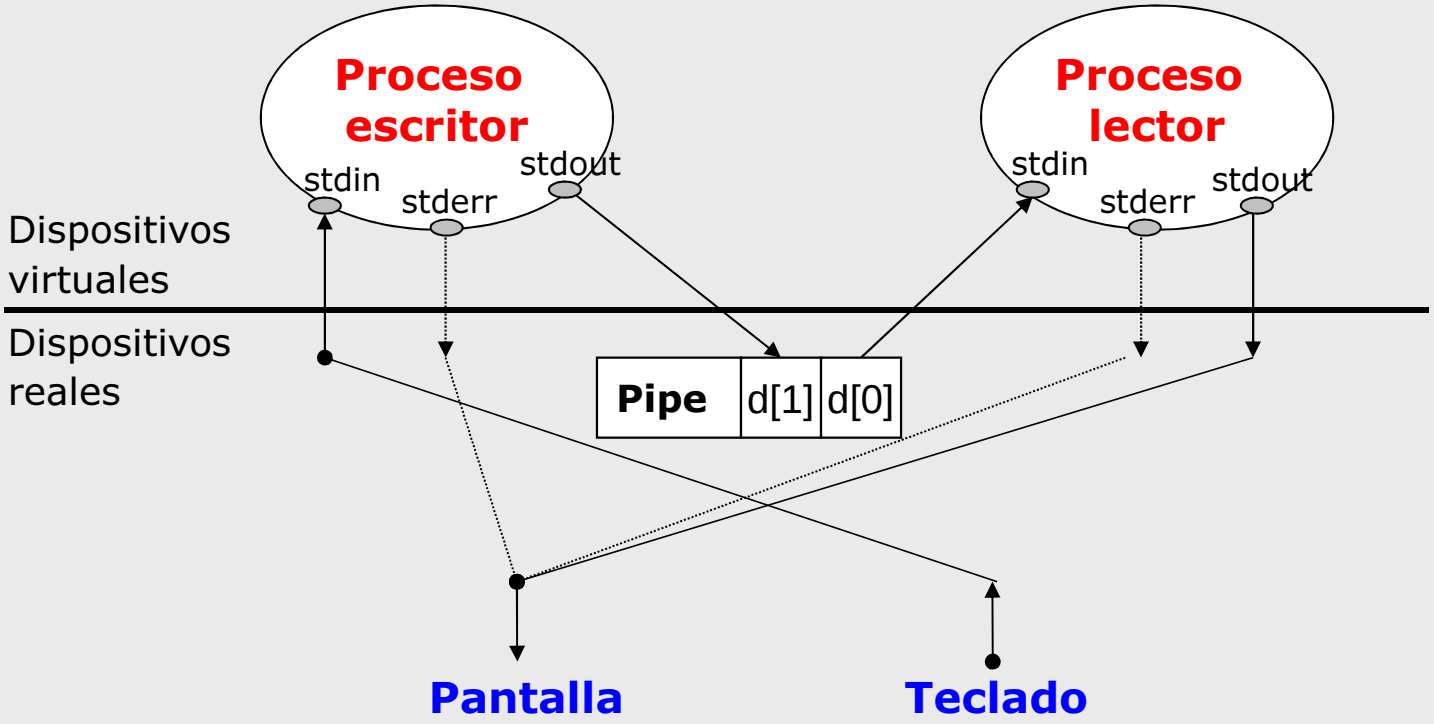
```
        /* Padre */
        /* Cerramos el descriptor que no vamos a utilizar */
        close(fp[0]);
        /* Instalar rutina tratamiento SIGPIPE */
        signal(SIGPIPE, Cerrar);
        /* Escritura pipe */
        while ( scanf("%d",&num)>0 )
        {
            if ( write(fp[1], &num, sizeof(int))<0 )
            {
                perror("Escritura pipe"); exit(1);
            }
        }
       close(fp[1]);   exit(0);
}

void Cerrar(int sig)
{
     close(fp[1]); exit(0);
}
```

Redirección pipes

- Si el proceso hijo realiza un recubrimiento (exec) no es posible utilizar las variables de los pipes heredadas para realizar la comunicación.
  - Después del recubrimiento el proceso hijo no tiene las mismas variables y no heredan los valores del padre.
- Para que dos programas diferentes puedan utilizar la comunicación mediante pipes tienen que definir a priori en que descriptores se van a recibir los pipes:
  - Se suele utilizar los descriptores estándar (stdin, stdout), para redirigir los pipes y poder realizar la comunicación.

Redirección pipes



Duplicados

- La forma más sencilla de heredar los pipes es mediante la creación de procesos clonados (fork).
- Para utilizar pipes con distintos ejecutables (recubrimiento) es necesario redireccionar la salida y la entrada estándar de los procesos hacia los pipes.
- Esta redirección se consigue cerrando los descriptores estándar y duplicando los descriptores del pipe mediante la llamada a sistema dup (unistd.h).
  - Sintaxis:

```
int dup(int fdv);
int dup2(int fdv, int fdn);
```

- Dup usa el descriptor libre más pequeño para duplicar el descriptor fdv. Dup2 hace que fdn sea la copia de fdv, cerrando primero fdn si es necesario.

Ejemplo pipes: Redirección (I)

```
int fp[2];

if (pipe(fp)==-1) {
    perror("Creación Pipe:"); exit(-1); }
switch (fork())
{
    case 0:         /* Hijo */
        /* Redirecciono la entrada del hijo al desc. lectura del pipe */
        close(fp[1]);      /* Cerramos descriptores no utilizados */
        close(0);
        dup(fp[0]);
        close (fp[0]);     /* Cerramos descriptores no utilizados */
        execlp("sort",” sort",NULL);
        exit(-1);
```

Ejemplo pipes: Redirección (II)

```
    default:        /* Padre */
            /* Redirecciono la salida del padre al desc. escritura del pipe */
            close(fp[0]);      /* cerramos los descriptores que no vayamos
                                  a utilizar */
            close(1);
            dup(fp[1]);
            close (fp[1]);     /* Cerramos descriptores no utilizados */
            execlp(”ls","ls”,”-la”,NULL);
            exit(-1);
}
```

Ejemplo: Redirección y Escritura (I)

```
main ()
{
    int fp[2];  char *c="99 76 15\n";
    if (pipe(fp)==-1) {
        perror("Creación Pipe:"); exit(-1); }
    switch (fork())
    {
        case 0:  /* Hijo */
        /* Redirecciono la entrada del hijo al desc. lectura del pipe */
        close(0);
        dup(fp[0]);
        /* Cerramos descriptores no utilizados */
        close(fp[1]);
        close (fp[0]);
        execlp("Filtro"," Filtro",NULL);
        exit(-1);
    }
```

Ejemplo: Redirección y Escritura (II)

```
/* Padre */
/* Redirecciono la salida del padre al desc. escritura del pipe */
close(1);
dup(fp[1]);
/* Cerramos los descriptores que no vamos a utilizar */
close(fp[0]);
close(fp[1]);
fprintf(stdout,"1 2 3 4 5 7 19");
printf("20 24 12");
fflush(stdout);
write(1,c,9);
close(1);
}
```

Ejemplo: Filtro

```
main ()
{
    int num;

    while (scanf(“%d”,&num)!=EOF)
    {
        if (num%2)
            printf(“ %d \n”, num);
    }

    return(0);
}
```

Caso de Estudio:Comunicación y sincronización en Unix

- Pipes (paso de mensajes)
- IPC (Inter-Process Comunication)
  - Cola Mensajes (paso de mensajes)
  - Memoria Compartida
  - Semáforos (sincronización)

Señales

- Las señales son excepciones software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono ó situación especial.
- Cada proceso tiene asignada una acción específica a ejecutar para cada tipo de señal que puede recibir.
- Se pueden cambiar las acciones específicas a ejecutar. Si no se especifica ninguna, el kernel del sistema la acción por defecto que normalmente implica finalizar el proceso.
- También se puede notificar a un programa que ignore ciertas señales.

Tipos de señales

1	HUP	El terminal de lectura se ha colgado.
2	INT	Se ha enviado una interrupción mandada desde el teclado.
3	QUIT	Se ha enviado el comando QUIT desde el teclado.
9	KILL	Señal para abortar el proceso.
10	BUS	Error en el bus al acceder a memoria.
11	SEGV	Segmentation fault.
15	TERM	Señal para terminar el proceso.
17	STOP	Señal de parada de proceso.
18	TSTP	Señal de parada de teclado.
19	CONT	Señal de continuación despues de la parada.
28	WINCH	La ventana ha cambiado de tamaño.
30	USR1	Definida por el usuario.
31	USR2	Definida por el usuario.

xx: Pueden variar según la versión de UNIX

Envío de señales

- Para enviar una señal a un proceso ó un grupo de procesos se utilizar la llamada kill (signal.h).

  Sintaxis:

  ```
  int kill(pid_t id, int sig);
  ```

- Si pid>0 → Identifica al proceso receptor.

  Si pid=0 → Se envía a todos los procesos del mismo grupo que el emisor.

  Si pid=-1→ Se envía a todos los procesos cuyo id. real es igual al id. efectivo del emisor.

  Si pid<-1→ Se envía a todos los procesos cuyo id. de grupo de procesos que coincide con el valor absoluto de pid.

Tratamiento de las señales

- Para especificar qué tratamiento debe realizar un proceso al recibir una señal se emplea la llamada signal (signal.h)
- Sintaxis:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t action);
```

- La llamada devuelve un apuntador a una función y requiere dos parámetros:
  - Sig: Identificador de la señal que se quiere desviar.
  - Action: Acción que queremos que se invoque cuando se reciba la señal.

    Este parámetro puede utilizar 3 tipos de valores:
    - SIG_DFL: Indica que la acción a realizar es la acción por defecto.
    - SIG_IGN: Indica que la señal se debe ignorar.
    - Dirección función: apuntador a la rutina de tratamiento de la señal del usuario.

Puntualizaciónes signal

- La llamada a signal devuelve el valor que tenía *action*, para poder restaurarlo posteriormente. Si ha ocurrido un error la llamada devuelve el valor SIG_ERR.
- La llamada a la rutina de tratamiento de la señal es asíncrona.
- La rutina de tratamiento recibe como parámetro el identificador de la señal generada.
- La mayoría de las rutinas de tratamiento, una vez invocadas, restauran la rutina por defecto.

Ejemplo señales (I)

```
void gestor_sigint(int sig);

main ()
{
    if (signal(SIGINT, gestor_sigint)==SIG_ERR) {
        perror("Signal");  exit(-1);
    }
    while(1) {
        printf("Esperando Ctrl-C \n");
        sleep(999)
    }
}
void gestor_sigint(int sig)
{
    printf("Señal número %d recibida.\n",sig);
}
```

Ejemplo señales (b)

```
void gestor_sigint(int sig)
{
    printf("Señal número %d recibida.\n",sig);

    /* Instalamos la misma rutina de gestión*/
    if (signal(SIGINT, gestor_sigint)==SIG_ERR)
    {
        perror("Signal");
        exit(-1);
    }
}
```

Espera de señales

- Se puede bloquear un proceso hasta que se reciba una señal mediante la llamada pause (signal.h).
- Sintaxis:

```
int pause( );
```

- Siempre devuelve -1.

Temporizadores

- Se puede planificar la generación de la señal SIGALRM una vez transcurrido un determinado tiempo mediante la llamada alarm (unistd.h).
- Sintaxis:

```
int alarm(long int seg);
```

- Pasado seg segundos el S.O. le envía una señal de alarma (SIGALRM) al proceso.
- Si seg==0, entonces desactiva la alarma activa y devuelve el tiempo que le quedaba pendiente.

Ejemplo alarm / wait

```
/* Programa que permite ejecutar un comando de forma de forma
    diferida */
void main(int argc, char argv[])
{
    /* Retraso la ejecución del comando  tantos segundos como
    indique el primer argumento*/
    alarm(atoi(argv[1]));
    pause();

    /* Ejecutamos el argumento introducido por el usuario*/
    execv(argv[2], &argv[2]);
    perror("No se ha podido ejecutar el comando");
    exit(-1);
}
```

Problema: Señales

- Realizar un reloj que muestre las horas, minutos y segundos, mediante 3 programas que se sincronicen con señales.
- Cada uno de los procesos se encargara de gestionar y mostrar un campo del reloj (hora, minutos ó segundos).
- Solo el primero de los procesos (segundos) puede utilizar la señal SIGALARM.