UNIVERSIDAD DEL BÍO-BÍO

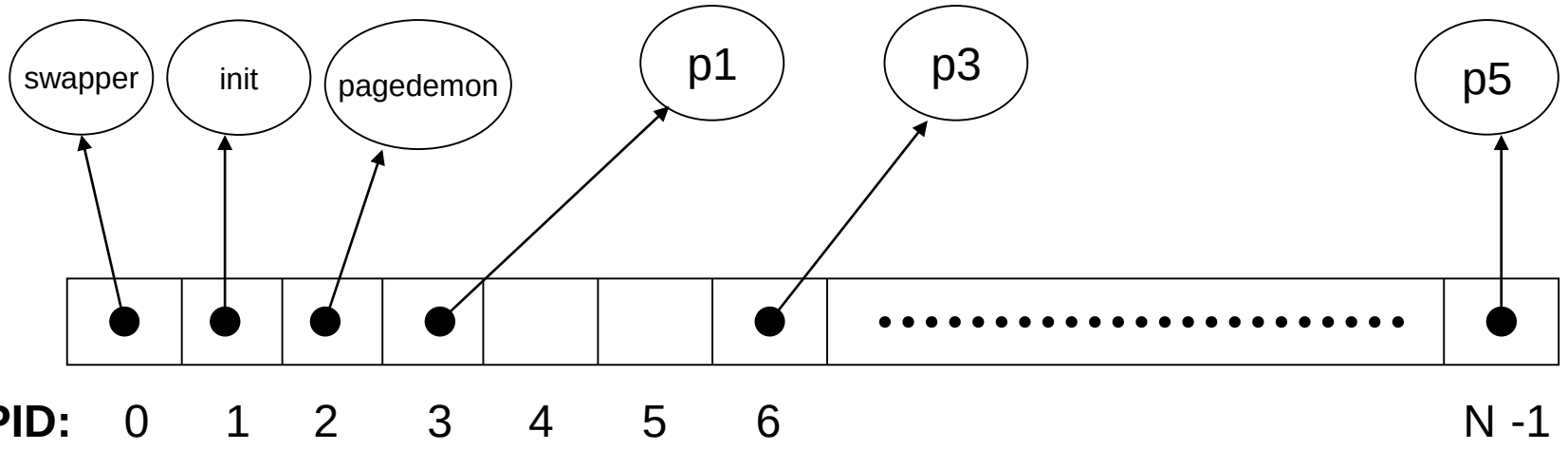
Procesos
Resumen General

Fernando Santolaya F.
fsantolaya@ubiobio.cl

Creando un proceso

• El valor retornado por *fork()* corresponde a un número entero único que identifica al proceso, llamado **PID**.

• Este PID es asignado por el kernel al proceso, para ello existe una tabla (o arrreglo) cuyos índices corresponde al PID asignado a cada proceso que se asigna a esa celda.



Creando un proceso

• ***fork()*** duplica un proceso, esto quiere decir que genera un clon con el mismo código ejecutable. Ambos procesos comparten:

- descriptores de archivos abiertos
- información del entorno
- valores de variables hasta ese punto
- etc.

•**Ambos procesos no comparten la misma zona de memoria:**

- los valores de las variables pueden cambiar de ahí en adelante.
- la misma variable puntero, en ambos, no apunta a la misma zona de memoria

• El proceso hijo también posee su propia información:

- tiene su propio PID (process ID)
- el hijo posee un identificador del padre (PPID)
- se resetea la información de uso de recursos del proceso (tiempo de CPU asignada, etc)
- se puede configurar a gusto la recepción de señales (interrupciones)

Grupo de procesos: Ejemplo2

- Código del proceso padre.

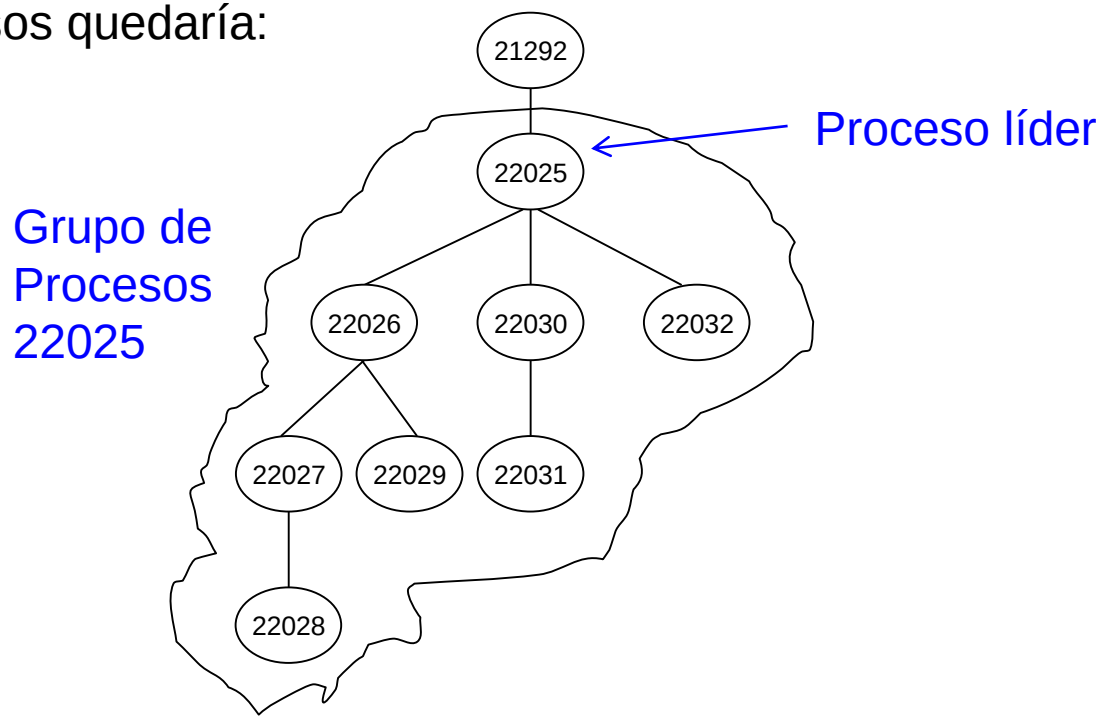
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void){
  int i;
  printf("\n\n Proceso inicial \t PID %6d \t PPID %6d \t GID %6d\n\n",
                                          getpid(), getppid(), getpgid(0));
  for(i=0; i<3; ++i)
     if (fork()==0)
        printf(("\n\n Nuevo proceso \t\t PID %6d \t PPID \t GID %6d\n\n",
                                          getpid(), getppid(), getpgid(0));
  return 0;
}
```

- Salida del programa:

```
Proceso inicial    PID    22025    PPID  21292   GID   22025
Nuevo proceso      PID    22026    PPID  22025   GID   22025
Nuevo proceso      PID    22027    PPID  22026   GID   22025
Nuevo proceso      PID    22029    PPID  22026   GID   22025
Nuevo proceso      PID    22028    PPID  22027   GID   22025
Nuevo proceso      PID    22030    PPID  22025   GID   22025
Nuevo proceso      PID    22031    PPID  22030   GID   22025
Nuevo proceso      PID    22032    PPID  22025   GID   22025
```

Grupo de procesos

- El árbol de procesos quedaría:

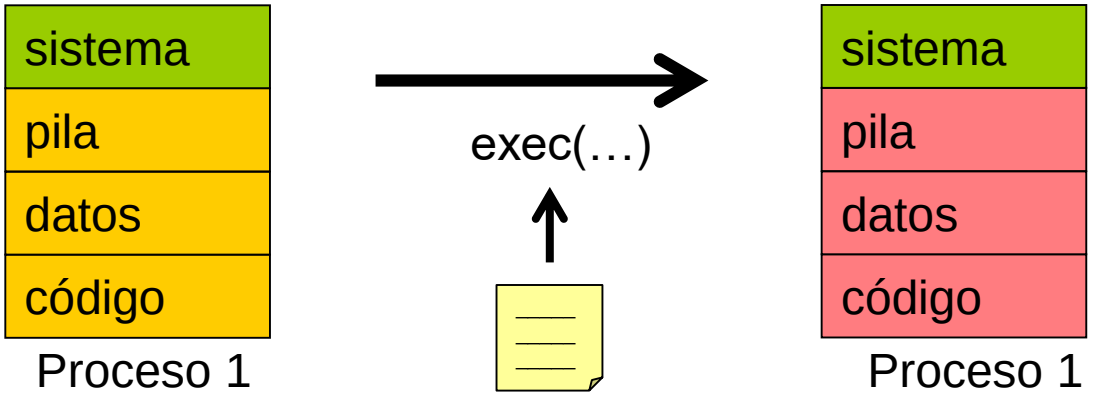


-

EXEC

¿Qué es exec?

- En Unix existe un conjunto de 6 llamadas al sistema, denominadas “la familia exec”.
- Permiten cambiar el código de un proceso (para que no sea un clon). Se reemplazan los segmentos de código, datos y pila.
- El nuevo programa comienza su ejecución en la función main().
- Si exec es exitoso entonces no retorna, porque se cambia la imagen del proceso.



Funcionamiento de exec

- **Las llamadas también se pueden clasificar en dos grupos:**

1. execl(), execlp(), execle()

   Estas llamadas pasan los argumentos de la línea de comandos del programa mediante una lista de constantes. Son útiles cuando se conoce el n° de argumentos que se van a pasar.

   - execle() permite pasar nuevos valores a variables de ambiente.
   - execlp() permite tomar el PATH por defecto del ambiente.

2. execv(), execvp(), execve()

   Estas llamadas pasan los argumentos de la línea de comandos en un arreglo de argumentos (dinámico), por lo cual es útil cuando no sabemos el n° de argumentos que hay.

   - execve() permite pasar nuevos valores a variables de ambiente.
   - execvp() permite tomar el PATH por defecto del ambiente.

Ejemplo: execlp

- Ejecutar el comando cat de Unix

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

int main(int argc, char *argv[ ]){
  if (argc > 1) {
    execlp("/bin/cat", "cat", argv[1], (char *)NULL);
    perror("error en exec ");
    exit(1);
  }
  fprintf(stderr, "Modo de uso: %s archivo_texto\n", *argv);
  exit(1);
}
```

arg[0] para cat

Nombre del archivo test.txt

- Como invocar este programa

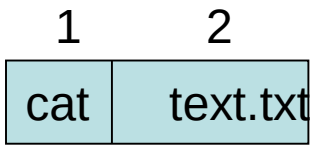
```
$ a.out  test.txt
```

execlp("/bin/cat", "cat", argv[1], (char *)NULL);

Ejemplo: execvp

- Ejecutar el comando cat de Unix

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

int main(int argc, char *argv[ ]){
  execvp(argv[1], &argv[1]);
  perror("error en exec ");
  exit(1);
}
```



- Como invocar este programa

```
$ a.out  cat test.txt
```

0 1 2

Ejemplo: fork y exec juntos

- En la mayoría de los casos (99%), fork() y exec() son utilizados en conjunto.

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

int display_msg(char *);

void main() {
  static char *msg[]= {“hola”, “que”, “tal”};
  int i;
  for (i=0; i<3; ++i)
     (void) display_msg(mesg[i]);
   sleep(2);
}

int display_msg(char *m) {
  char err_msg[25];
  switch(fork()) {
    case 0:
      execlp(“/usr/bin/echo”, “echo”, m, (char *) NULL);
      sprintf(err_msg, “%s falla en exec”, m);
      perror(err_msg);
      return 1;
    case -1:
      perror(“falla en fork”);
      return 2;
    default:
      return 0;
  }
}
```

KILL

Terminar un proceso

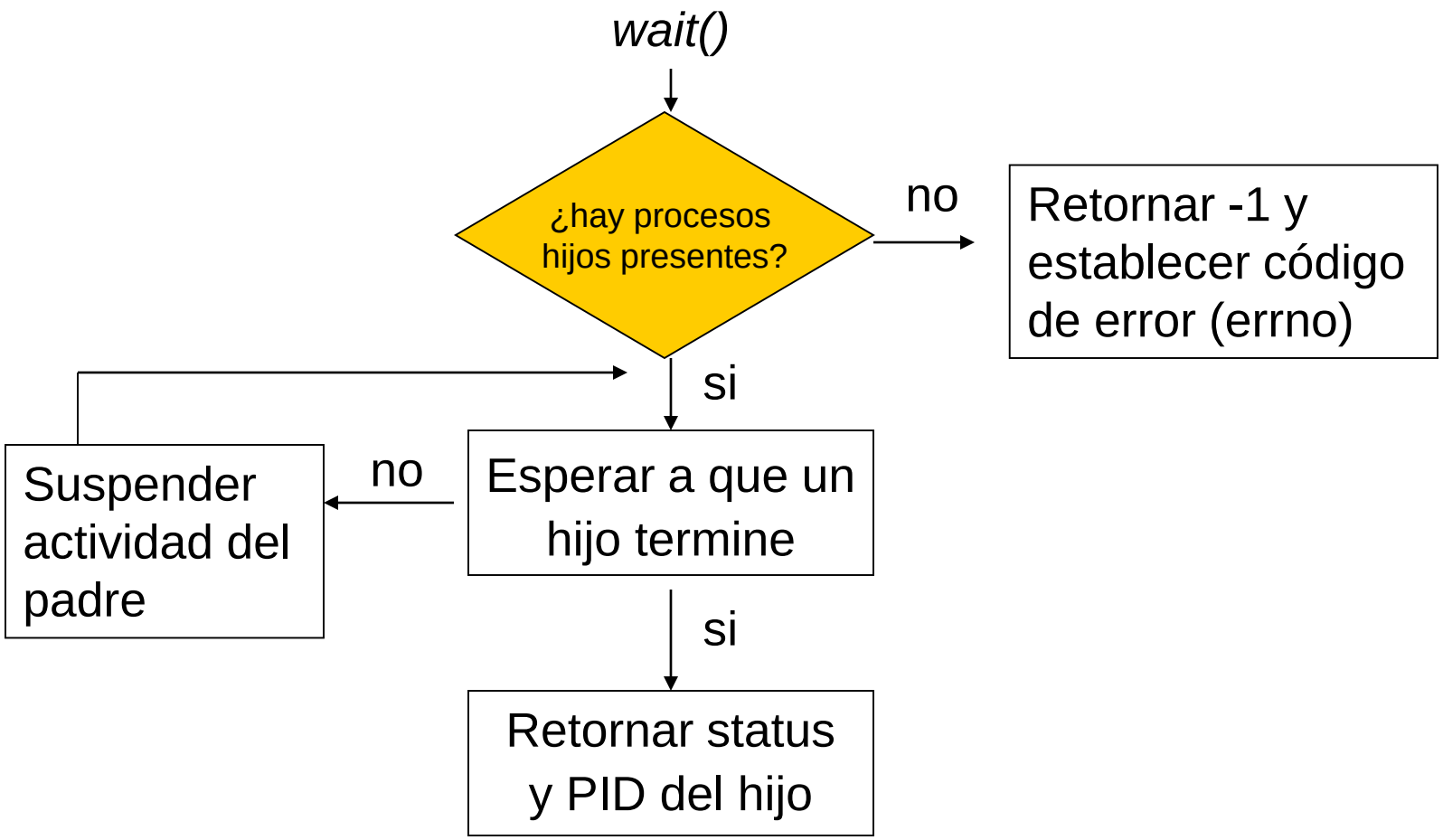
- Existen 3 maneras por las cuales puede terminar un proceso:
  - En algún punto de su código de invoca la llamada al sistema exit().
  - En algún punto del código se encuentra la instrucción return() para la función main().
  - Por errores en el programa, con lo cual termina abruptamente por fallas en su funcionamiento.

Terminar un proceso

- Cuando un proceso termina el sistema operativo realiza algunas labores:
  - Todos los descriptores de archivos abiertos son cerrados y sus buffer asociado es vaciado.
  - El padre del proceso es notificado por medio de una señal (se verá más adelante) SIGCHLD (SIG = signal, CHLD = children) que el proceso hijo terminó.
  - El padre recibe información de estatus del hijo.
  - Todos los procesos hijos al terminar su padre cambian su PPID a 1 (proceso init).

WAIT

Funcionamiento de wait



Ejemplo: wait

- Código del proceso padre.

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

int main(void){
  pid_t pid, w;
  int i, status;
  char value[3];
  for (i=0; i<3; ++i) {
    if ((pid= fork()) == 0) {
      sprintf(value, "%d", i);
      execl("hijo", "hijo", value, (char *) 0);
    } else
      printf("Hijo creado con fork %d\n", pid);
  }
  while ((w= wait(&status)) && w!=-1){
    printf("PID del proceso esperado:%d - status: %04X\n", w, status);
  }
  exit(0);
}
```

Ejemplo: wait

- Código del proceso hijo (hijo.c).

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char *argv[]){
  pid_t pid;
  int ret_value;
  pid= getpid();
  ret_value= (int) (pid % 256);
  srand((unsigned)pid);
  sleep(rand()%5);
  if (atoi(*argv + 1))%2) {
    printf("Hijo %d terminado por la señal 0009\n", pid);
    kill(pid, 9);
  } else {
    printf("Hijo %d terminado por exit(%04X)\n", pid, ret_value);
    exit(ret_value);
  }
}
```

Procesos Zombie

- Procesos han terminado antes de haber sido esperados por su padre.
- Esta situación genera un tipo de procesos llamados ZOMBIE.
- Estos ocupan una entrada en la tabla de procesos del sistema y son marcados con una letra Z.
- Los procesos ZOMBIE no pueden ser terminados con el comando kill.
- Los procesos ZOMBIE pueden ser recuperados si el padre utiliza wait() en algún momento.
- **Si por algún motivo el padre termina sin hacer wait() a sus hijos, estos son heredados por el proceso *init* el cual automáticamente invoca a wait().**

Funcionamiento de wait

- El argumento **`stat_loc`** es un puntero, por lo cual una vez que wait() retorna el PID de un proceso hijo esperado se puede consultar el status de salida de ese proceso utilizando para ello un conjunto de macros predefinidas:

#include <sys/wait.h>

WIFEXITED(stat_loc) ¿el proceso terminó normalmente? (0=false sino true)
WEXITSTATUS(stat_loc) retorna el código de salida del proceso hijo (exit(?))

WIFSIGNALED(stat_loc) ¿el proceso terminó debido a una señal?
WTERMSIG(stat_loc) retorna el código de la señal

WIFSTOPPED(stat_loc) ¿el proceso está actualmente detenido?
WSTOPSIG(stat_loc) retorna la señal que causó que el hijo se detuviera

WIFCONTINUED(stat_loc) ¿el proceso hijo continúa en ejecución?
WCOREDUMP(stat_loc) ¿se ha generado un archivo CORE?

Funcionamiento de wait

- Ejemplo usando macros:

```
...
While ((w= wait(&status)) && w!=-1) {
  if (WIFEXITED(status))
    printf("PID esperado:%d - valor retornado:%04X\n", w, WEXITSTATUS(status));
  else
    if (WIFSIGNALED(status))
      printf("PID esperado:%d - señal retornada:%04X\n", w, WTERMSIG(status));
}
...
```

- Limitaciones de wait:
  - wait() siempre retorna el status del primer hijo que termina o se detiene. (podría no ser el status del hijo que queremos).
  - wait() siempre se bloquea si la información de status no está disponible. (esto significa que el padre no puede seguir ejecutándose a la espera del término de un hijo).
  - Solución: usar otra llamada al sistema → waitpid()

Ejemplo: waitpid

- Código del proceso padre (padre.c).

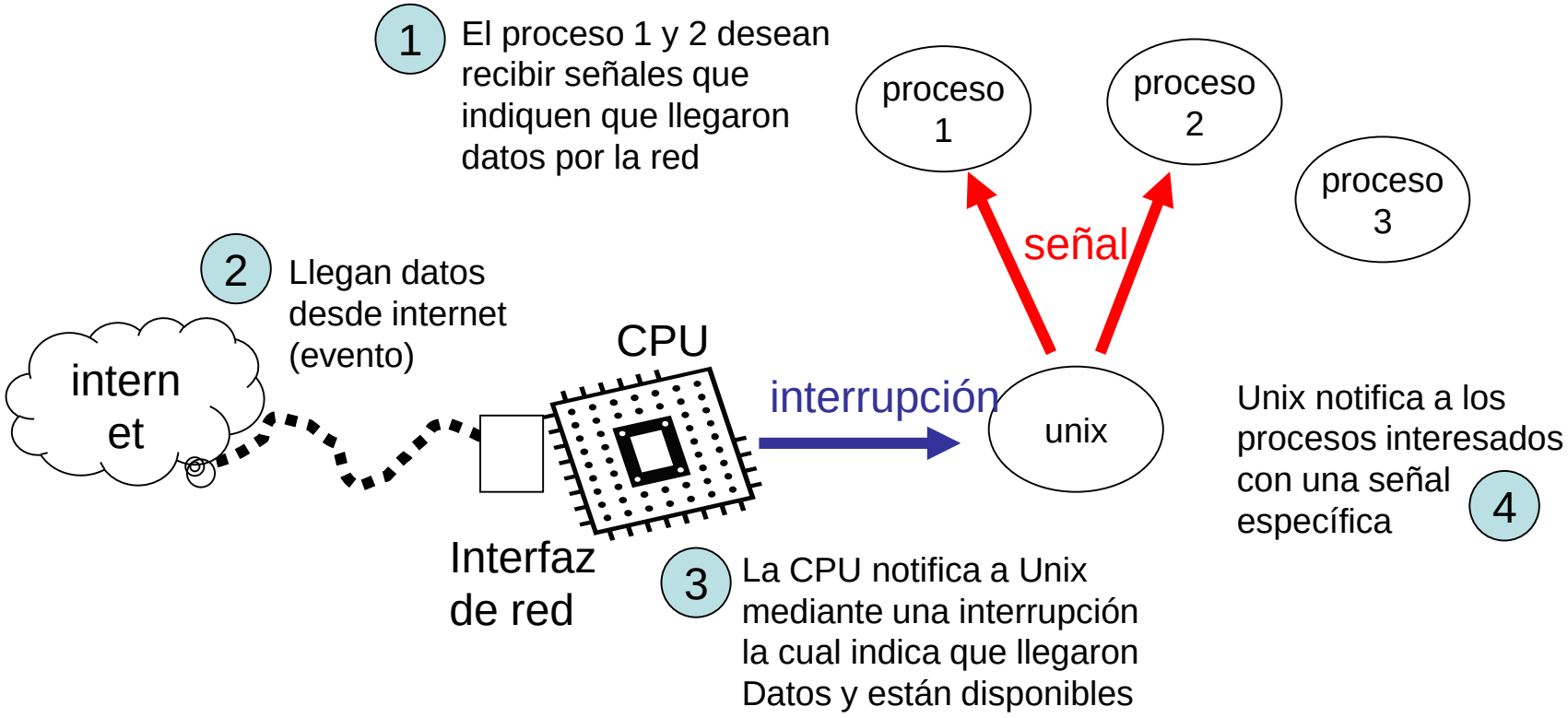
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main(void){
  pid_t pid[3], w;
  int i, status;
  char value[3];
  for (i=0; i<3; ++i) {
    if ((pid[i] = fork()) == 0) {
      sprintf(value, “%d”, i);
      execl(“hijo”, “hijo”, value, (char *) 0);
    } else
      printf(“Hijo creado con fork %d\n”, pid[i]);
  for (i=0; (w = waitpid(pid[i], &status, 0)) && w!=-1; ++i){
    if (WIFEXITED(status))
      printf(“PID esperado:%d - valor retornado:%04X\n”, w, WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
           printf(“PID esperado:%d - señal retornada:%04X\n”, w, WTERMSIG(status));
  }
  exit(0);
}
```

SEÑALES

¿Qué son las señales?

- Una señal siempre es generada por el software, ya sea el kernel o un proceso de usuario.

- Una interrupción es generada por el hardware, capturada por el kernel y enviada como “señal” a todos los procesos a los que pueda afectar.



Enviar una señal

- Para enviar una señal desde un proceso a otro o a un grupo de procesos se utiliza

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

Dónde:

pid>0 PID del proceso al cual se le envía la señal

pid=0 La señal se envía a todos los procesos del mismo grupo que el proceso que envía.

- Sig corresponde a un valor entero que representa la señal que se quiere enviar (si es cero significa señal nula → no existe señal con ese valor). Esto sólo sirve para saber si el proceso existe.

Ejemplo: Enviar una señal

```
#include <signal.h>

main () {
  int pid;
  if (pid=fork())==0) {
    while(1) { /*ciclo infinito*/
      printf("Soy el hijo\n");
      sleep(1);
    }
  }
  sleep(10);
  printf("Soy el padre\n");
  printf("Eliminando a mi proceso hijo\n");
  kill(pid, SIGTERM); /*También se puede usar SIGKILL*/
  exit(0);
}
```

Recibir una señal

- Un proceso puede controlar las acciones a realizar cuando recibe una señal.

- Para realizar esto es necesario implementar una rutina de tratamiento para la señal (llamada “manejador”) y reemplazar aquella que el sistema implementa por defecto.

- La llamada *void* (**signal(int sig, void (*action)(int)))* permite realizar esto. **sig** corresponde al número de la señal y **action** indica la acción a realizar cuando se reciba la señal.

  action puede tomar uno de estos valores:

  - SIG_IGN ignorar la señal
  - SIG_DFL se debe realizar la acción por defecto (definida por el kernel)
  - dirección es la dirección de la rutina definida por el usuario (puntero a la función)

Ejemplo: Recibir una señal

- Ejemplo: Capturar el ctrl + C desde teclado

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

void sigint(int sig);

main() {
  if (signal(SIGINT, sigint)==SIG_ERR) { /*capturar la
señal*/
    printf("Se ha producido un error\n");
    exit(-1);
  }
  while(1) {
    printf("En espera de Ctrl + C\n");
    sleep(1);
  }
}

void sigint(int sig) { /*manejador para la señal*/
  printf("señal número %d recibida \n", sig);
}
```

Ejemplo: Recibir una señal

- El programa anterior sólo captura una vez la interrupción.

- Propuesto:

  Modifique el programa anterior para que cada vez que se presione ctrl + C el programa capture la señal.

Esperar una señal con tiempo

- Sino se desea esperar indefinidamente por una señal, se puede utilizar *unsigned int sleep(unsigned int segundos),* la cual duerme el proceso por la cantidad indicada de segundos.

- sleep() retorna de inmediato cuando llega una señal, o cuando se cumple el tiempo si no llega una (retorna el número restante de segundos).

Programar el TIMER

- Todos los computadores contienen internamente un TIMER (reloj contador o cronómetro).

- Un proceso puede programar este TIMER para que se le avise en “t segundos” más que el tiempo se ha cumplido. La señal que obtiene el proceso es SIGALRM cuando se cumple el tiempo.

- La función *unsigned int alarm(unsigned int segundos)* nos provee exactamente la misma funcionalidad.

- Si hay una señal previamente programada, se cancela y se utiliza la nueva.

```
#include <unistd.h>
…
alarm (5);
…
```

Programar el TIMER

- Para capturar la señal SIGALRM es necesario utilizar la función signal() igual como se hizo anteriormente para otras señales.

- Al utilizar alarm() es necesario utilizar un manejador (usar signal()), si esto no se hace el proceso es terminado al recibir la señal.

- Propuesto:

1. Implemente un cronómetro mediante el TIMER, que muestre los segundos, minutos y la horas transcurridos desde que se invocó el programa.

2. Investigar la función *setitimer()*.