INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

SISTEMAS OPERATIVOS

3004610 - 1

Servicios POSIX - procesos

SISTEMAS OPERATIVOS

3004610 - 1

Servicios POSIX

Servicios de gestión de procesos
- Identificación
- Entorno del proceso
- Creación de procesos
- Terminación de procesos
- Modificación de propiedades

Gestión de procesos

Identificación de procesos

Identificador: entero único. Tipo: pid_t (equivale a entero)

a) Servicio para obtener identificación

```
pid_t getpid (void);
```

b) Servicio para obtener la identificación del proceso padre

```
pid_t getppid (void);
```

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


int main(void) {
        printf("pid:%d ppid:%d\n", getpid(), getppid());
return EXIT_SUCCESS;
}
```

Entorno de un proceso

El entorno de un proceso está definido por una lista de variables que se pasan al proceso en el momento de comenzar su ejecución (variables de entorno). Son accesibles al proceso mediante la variable externa **environ**, que ha de ser declarada así:

```
extern char ** environ;
```

Esta lista es un vector de punteros a cadenas de caracteres, de la forma:

nombre_variable = valor

POSIX establece las siguientes variables de entorno:

- HOME (directorio de trabajo del usuario asociado al proceso)
- LOGNAME (nombre de usuario asociado al proceso)
- PATH (ruta para encontrar ejecutables)
- TERM (tipo de terminal)
- T2 (información sobre zona horaria)
- ……..

Para obtener el valor de una variable de entorno concreta, se puede usar el servicio:

```
char *getenv(const char *name)
```

Ejercicio: hacer un programa que muestre todas las variables de entorno accesibles mediante environ.

Ejercicio: hacer un programa que, usando getenv, obtenga el valor de las variable de entorno DISPLAY, HOME y PWD. Ver qué ocurre cuando se solicita el valor de una variable de entorno que no exista (por ejemplo, la variable de entorno k2r)

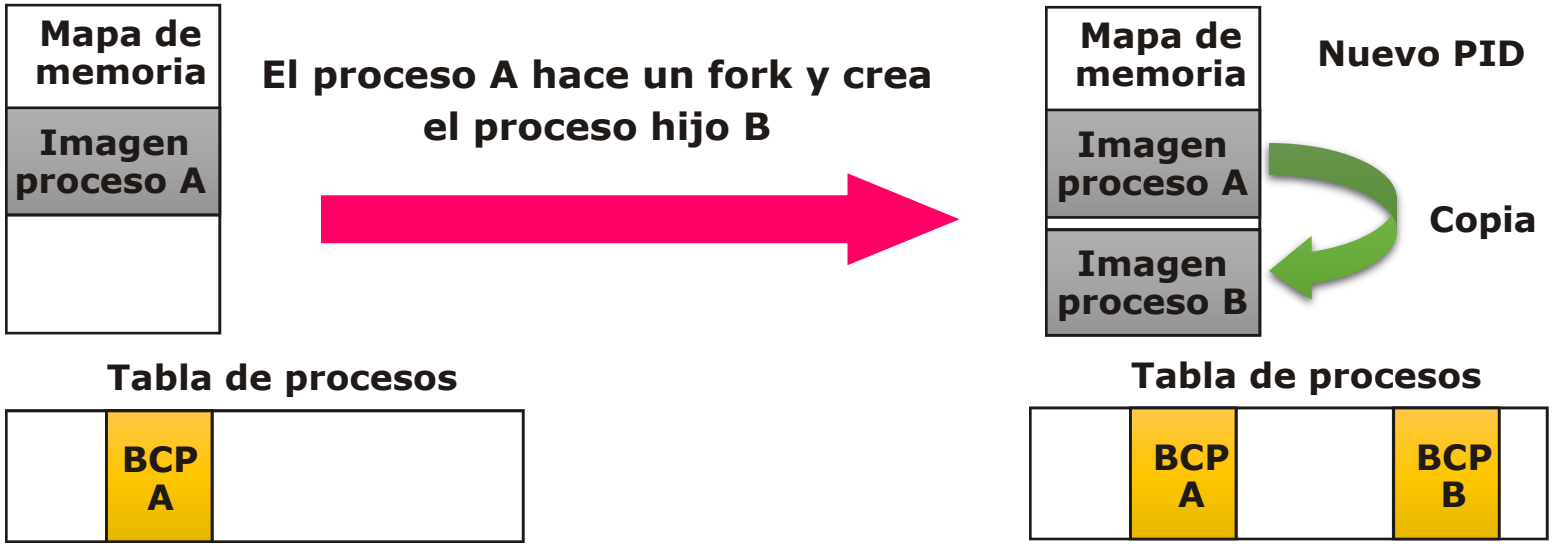
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char **environ;

int main(void) {
        //obtener toda la lista de variables de entorno
        char **list = environ;
        char *var;
        while(*list!=NULL){
                printf("%s\n", *list);
                list ++;
        }
        //obtener el valor de una variable especifica
        var = getenv("LOGNAME");
        printf("%s\n", var);
        return EXIT_SUCCESS;
}
```

a) Creación: uso del servicio *fork*. Mediante este servicio se produce la clonación del proceso padre, resultando un nuevo proceso (hijo).

pid_t fork()



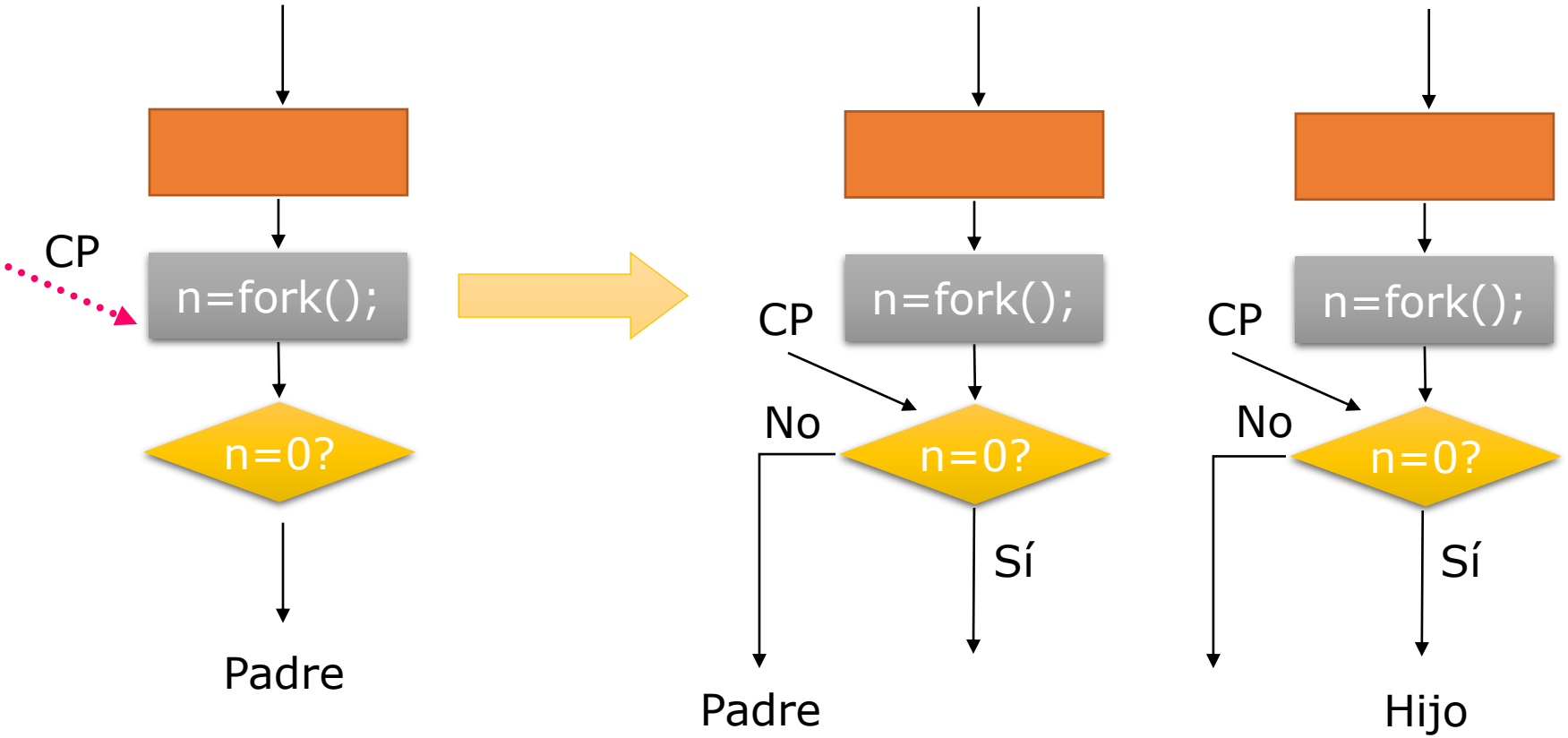
Al copiarse la imagen de memoria del proceso padre en el hijo, el texto (código), datos y la pila de ambos procesos (en el momento de la creación son iguales). Ambos procesos tendrán también los mismos valores de registros en sus BCP. Esto significa que el CP de ambos procesos será el mismo, por lo que ambos procesos ejecutarán la siguiente sentencia a la llamada a *fork*. Por tanto, ¿cómo conseguir entonces que cada uno realice una tarea diferente?

Las diferencias entre ambos procesos son:

a) Identificadores
b) Diferentes segmentos de memoria, pese a que sea copia
c) Tiempo de ejecución del hijo iniciado a 0
d) El proceso hijo no tendrá alarmas pendientes

e) El proceso hijo no tiene señales pendientes

f) El valor de retorno del fork es distinto en padre e hijo



fork() ***devuelve 0 al hijo y el padre recibe el pid del hijo*** !!!!
y ***–1 en caso de error***

Ambos procesos comparten los descriptores de los archivos usados por el proceso padre antes de la llamada a fork. Es decir, el proceso hijo tiene acceso a los archivos abiertos por el padre.

Ejercicio: hacer un programa para crear un proceso mediante la llamada a fork. Tras la llamada a fork el proceso padre mostrará los pid propio y del hijo. El proceso hijo, por su parte, mostrará el pid propio.

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

int main() {
    pid_t pid;
    int status;

    pid = fork(); // Crear un proceso hijo

    if (pid == -1) {
        printf("Error al crear proceso hijo\n");
        return 1;
    } else if (pid == 0) { // Proceso hijo
        printf("Proceso hijo: PID = %d\n", getpid());
        sleep(5); // Esperar 5 segundos
        printf("Proceso hijo terminado\n");
        exit(0);
    } else { // Proceso padre
        printf("Proceso padre: PID = %d\n", getpid());
        printf("Esperando a que el proceso hijo termine...\n");
        wait(&status); // Esperar a que el proceso hijo termine
        printf("Proceso hijo terminado con estado %d\n", WEXITSTATUS(status));
        printf("Proceso padre terminado\n");
        return 0;
    }
}
```

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

int main() {
    pid_t pid[3];
    int status;

    for (int i = 0; i < 3; i++) {
        pid[i] = fork(); // Crear un proceso hijo

        if (pid[i] == -1) {
            printf("Error al crear proceso hijo\n");
            return 1;
        } else if (pid[i] == 0) { // Proceso hijo
            printf("Proceso hijo %d: PID = %d\n", i+1, getpid());
            sleep(i+1); // Esperar entre 1 y 3 segundos
            printf("Proceso hijo %d terminado\n", i+1);
            exit(0);
        }
    }

    printf("Proceso padre: PID = %d\n", getpid());
    printf("Esperando a que los procesos hijos terminen...\n");

    for (int i = 0; i < 3; i++) {
        waitpid(pid[i], &status, 0); // Esperar a que el proceso hijo termine
        printf("Proceso hijo %d terminado con estado %d\n", i+1, WEXITSTATUS(status));
    }

    printf("Proceso padre terminado\n");
    return 0;
}
```

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

int main(void)
{
   pid_t pid;
   int i,j, status;
   int n = 10;

   printf("Proceso inicial (padre): %d\n",getpid());
   for (i = 0; i < n; i++)
   {
      pid = fork();
      if (pid == 0)
      {
             printf("Iteracion: %d - Padre: %d Hijo: %d\n",j,getppid(),getpid());
             break;
      }
   }
   if(i==n){
      for (j = 0; j < n; j++){
       wait(&status);
       printf("Hijo[%d] retorno %d\n",j,WEXITSTATUS(status));
      }
      Printf("Padre finalizado\n");
   }
return 0;
}
```

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

void main(void)
{
  pid_t pid;
  int i;
  int n = 10;

  for (i = 0; i < n; i++)
  {
    pid = fork();
    if (pid == 0)
      break;
     else{
       printf("---------------------- ITERACIÓN %d ------------------",i);
       printf("Creado el proceso con pid = %d\n",pid);
       printf("Creado por el proceso con pid = %d\n",getpid());
     }
  }

  printf("El padre del proceso con pid = %d es el proceso con pid = %d\n", getpid(), getppid());
}
```

Process control - zombies

- When a process ends, the memory and resources associated with it are deallocated.
- However, the entry for that process is not removed from the process table.
- This allows the parent to collect the child's exit status.
- When this data is not collected by the parent the child is called a "zombie". Such a leak is usually not worrisome in itself, however, it is a good indicator for problems to come.

Process control - zombies

- In some (rare) occasions, a zombie is actually desired – it may, for example, prevent the creation of another child process with the same pid.
- Zombies are not the same as *orphan processes* (a process whose parent ended and is then adopted by *init* (process id 1)).
- Zombies can be detected with **ps –el** (marked with 'Z').
- Zombies can be collected with the wait system call.

Process control

- Wait
  - pid_t wait(int *status);
  - pid_t waitpid(pid_t pid, int *status, int options);
  - The wait command is used for waiting on child processes whose state changed (the process terminated, for example).
  - The process calling wait will suspend execution until one of its children (or a specific one) terminates.
  - Waiting can be done for a specific process, a group of processes or on any arbitrary child with waitpid.
  - Once the status of a process is collected that process is removed from the process table by the collecting process.
  - Kernel 2.6.9 and later also introduced waitid(…) which gives finer control.

*pid_t wait(int *status);*

Retorna -1 si falla
- No hay procesos para esperar
- Señal

Retorna el PID del hijo que termina
Funciona como sincronización:
- Padre espera hasta que un proceso hijo termine

*pid_t waitpid(pid_t p, int *status, int opt);*

Retorna un Pid del proceso hijo o -1
Argumento pid_t p :
- -1 waits for any child (same as wait)
- 0 waits for any child in same group w/parent
- >0 waits for child process with this PID=p
- <-1 waits for any child with GID = |p|

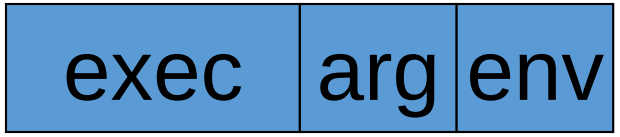
Process control

- exec*
  - int execv(const char *path, char *const argv[]);
  - int execvp(const char *file, char *const argv[]);
  - exec….
  - The exec() family of function replaces current process image with a new process image (text, data, bss, stack, etc).
  - Since no new process is created, PID remains the same.
  - Exec functions do not return to the calling process unless an error occurred (in which case -1 is returned and errno is set with a special value).
  - The system call is execve(…)

Función	Descripción	Uso
execl()	Ejecuta un programa con una lista de argumentos especificada como una serie de argumentos separados por comas.	Cuando el número de argumentos para el programa es conocido de antemano y es pequeño.
execv()	Ejecuta un programa con una lista de argumentos especificada como un arreglo de punteros a cadenas de caracteres.	Cuando el número de argumentos para el programa es conocido de antemano y es grande.
execle()	Ejecuta un programa con una lista de argumentos y un conjunto de variables de entorno especificados.	Cuando es necesario establecer o modificar variables de entorno para el programa.
execve()	Ejecuta un programa con una lista de argumentos y un conjunto de variables de entorno especificados como argumentos separados.	Cuando es necesario establecer o modificar variables de entorno para el programa y el número de argumentos para el programa es grande.
execlp()	Ejecuta un programa con una lista de argumentos especificada como una serie de argumentos separados por comas y busca el archivo del programa en los directorios de búsqueda de la variable de entorno PATH.	Cuando el número de argumentos para el programa es conocido de antemano y es pequeño, y el archivo del programa se puede encontrar en los directorios de búsqueda de PATH.
execvp()	Ejecuta un programa con una lista de argumentos especificada como un arreglo de punteros a cadenas de caracteres y busca el archivo del programa en los directorios de búsqueda de la variable de entorno PATH.	Cuando el número de argumentos para el programa es conocido de antemano y es grande, y el archivo del programa se puede encontrar en los directorios de búsqueda de PATH.

Llamada Exec

- La familia de llamadas exec tiene diferentes varianets:
  - execl, execlp, execle, execv, execvp, and execve.
- Convencion de los nombre:



Para Arg:

- **l** – Argumentos del programa en forma de lista que finalizan con NULL
- **v** – Argumentos del programa en forma de vector

Para Env:

- **p** – Variable de entorno en el PATH
- **e** – Variable de entorno en un vector enviado

Llamada Exec

```
#include <unistd.h>
int execl(const char *file, const char arg0, .. , const argn);
int execv(const char *file, char *argv[]);
```

- Si la llamada no falla el proceso ejecuta el código binario de file y no retorna
- Si falla la llamada retorna -1

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

int main() {
    pid_t pid;
    int status;

    pid = fork(); // Crear un proceso hijo

    if (pid == -1) {
        printf("Error al crear proceso hijo\n");
        return 1;
    } else if (pid == 0) { // Proceso hijo
        char *args[] = {"ls", "-al", NULL};
        execvp(args[0], args);
        printf("Error al ejecutar comando\n");
        return 1;
    } else { // Proceso padre
        wait(&status); // Esperar a que el proceso hijo termine
        printf("Proceso hijo terminado con estado %d\n", WEXITSTATUS(status));
        return 0;
    }
}
```

Ejemplo: primero.c

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

int main(int argc, char *argv[]){
   pid_t hijo[2];
   int i, status;

   for(i=0; i<2; i++){
      hijo[i] = fork();
      if(!hijo[i]){
         if(i==1){
            execl("./segundo", "1", "2", NULL);}
         printf("Hijo %d\n",i);
         break;}
       }
       if(i==2){
          for(i=0; i<2; i++){
             wait(&status);
         printf("Hijo[%d] retornÃ³ %d\n",i,WEXITSTATUS(status));
           } }
   return 0;
}
```

Ejemplo: Segundo.c

```
#include <stdio.h>

int main(int argc, char* argv[]){
        int i;
        printf("Argumentos: %d\t", argc);
        for(i=0; i<argc; i++){
                printf("%s\t", argv[i]);
        }
        printf("\n");
        return -1;
}
```