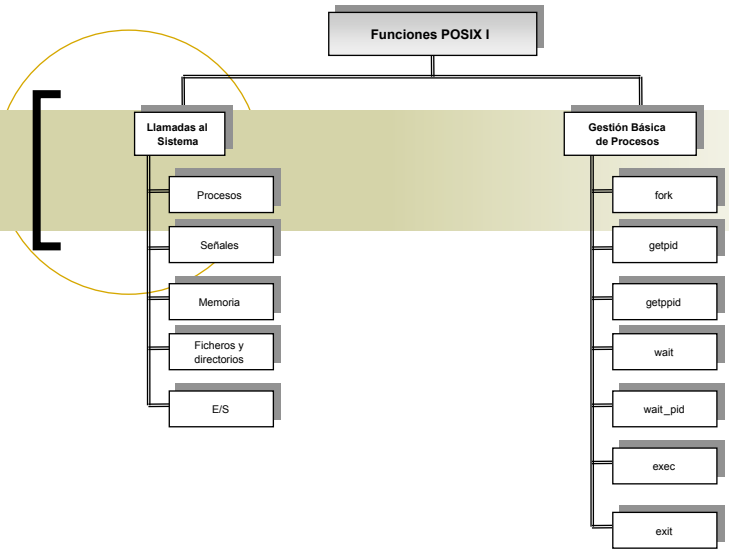


1

Funciones POSIX (I): Introducción

- ¿Que es POSIX?
  - **POSIX**: Portable Operating System Interface (IEEE)
  - Especifica la interfaz entre el sistema y el usuario.
  - Intenta estandarizar los interfaces de los SO para permitir que las aplicaciones funcionen en distintas plataformas.
  - **POSIX** se compone de distintas normas estándares llamadas "miembros"

Estándar	Descripción
POSIX.1	Interfaz del sistema para programas de aplicación (API) en lenguaje C
POSIX.1b	Rectificación 1 del API: Extensión de tiempo real en C
POSIX.1c	Rectificación 2 del API: Extensión de hilos de control
POSIX.2	Intérprete de comandos y útiles
POSIX.4	Ahora llamado POSIX.1c
POSIX.5	POSIX.1 Lenguaje ADA
POSIX.6	Seguridad
POSIX.7	Administración del sistema

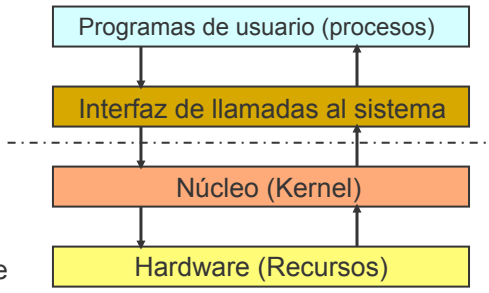
Programación de Sistemas Concurrentes

- La programación concurrente se puede realizar de dos formas
  - Utilizando un lenguaje diseñado para soportar concurrencia:
    - La concurrencia la proporciona el lenguaje
    - ADA
    - JAVA
  - Utilizando C estándar bajo un sistema UNIX que cumpla POSIX
    - La concurrencia la proporciona el SO
    - Los procesos solicitan servicios al SO (p.e. de E/S) mediante ***llamadas al sistema***

Llamadas al Sistema (I)

- Las llamadas al sistema son un conjunto de servicios básicos del SO

- Definen la interfaz entre un proceso y el SO
- Son una parte intrínseca del núcleo del SO *(Kernel)*
- ***Kernel***: colección de módulos software que se ejecutan de forma privilegiada (se tiene acceso a todos los recursos de la máquina)



Llamadas al Sistema (II)

- La implementación de las llamadas al sistema se realiza mediante interrupciones software o ***traps***
- Normalmente se proporciona al usuario una interfaz (*funciones de biblioteca*) que ocultan al usuario el mecanismo interno de las llamadas
- Tipos de llamadas en UNIX
  - Llamadas al sistema para gestión de procesos y threads
  - Llamadas al sistema para gestión de señales
  - Llamadas al sistema para gestión de memoria
  - Llamadas al sistema para gestión de ficheros y directorios
  - Llamadas al sistema para gestión de E/S

Llamadas al sistema: Ejemplo (I)

- Ejemplo de utilización de llamadas al sistema: Crear un fichero "ejemplo" (si no existe) y escribir la cadena "Esto es un ejemplo"
  - Llamadas involucradas: **open**, **write**
  - **open**: asocia un descriptor de fichero con un fichero o dispositivo lógico.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path,   int oflag, ...);
```

Etiqueta	Descripción
O_RDONLY	Abre el fichero en modo sólo lectura
O_WRONLY	Abre el fichero en modo sólo escritura
O_RDWR	Abre el fichero en modo lectura / escritura
O_APPEND	Abre el fichero en modo "añadir al final"
O_CREAT	Crea el fichero si no existe
O_TRUNC	Abre el fichero existente y lo trunca a 0

Modos de apertura de ficheros

Llamadas al sistema: Ejemplo (II)

- En caso de crear el fichero es necesario especificar el modo de apertura (permisos)
- Permisos en UNIX
  - Cada fichero en UNIX tiene permisos para el propietario del mismo, para el grupo de usuarios del propietario y para el resto
  - Los permisos para cada una de las clases anteriores son: leer (r), escribir (w) y ejecutar (x).
  - Cada fichero tiene asociados 9 bits donde cada bit a 1 o a 0 da o quita el permiso correspondiente según el patrón que se muestra a continuación

Usuario			Grupo			Otros		
r	w	x	r	w	x	r	w	x
Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Patrón de la máscara de permisos

Llamadas al sistema: Ejemplo (III)

- En vez de utilizar el valor numérico obtenido mediante la combinación de bits se pueden utilizar las siguientes etiquetas:

Etiqueta	Descripción
S_IRUSR	bit de permiso de lectura para el propietario
S_IWUSR	bit de permiso de escritura para el propietario
S_IXUSR	bit de permiso de ejecución para el propietario
S_IRWXU	lectura, escritura y ejecución para el propietario
S_IRGRP	bit de permiso de lectura para el grupo
S_IWGRP	bit de permiso de escritura para el grupo
S_IXGRP	bit de permiso de ejecución para el grupo
S_IRWXG	lectura, escritura y ejecución para el grupo
S_IROTH	bit de permiso de lectura para otros
S_IWOTH	bit de permiso de escritura para otros
S_IXOTH	bit de permiso de ejecución para otros
S_IRWXO	lectura, escritura y ejecución para otros
S_ISUID	fija el ID del usuario al momento de la ejecución
S_ISGID	fija el ID del grupo al momento de la ejecución

Etiquetas de permisos de un fichero

Llamadas al sistema: Ejemplo (IV)

- Las etiquetas de permisos se pueden combinar mediante el operador "|" ("OR" de bits)

```
fd = open ("ejemplo", O_CREAT, S_IRWXU | S_IXGRP);
```

- Llamada write: intenta escribir *nbytes* tomados del buffer *buf* en el archivo con descriptor *fildes*

```
#include <unistd.h>
ssize_t write( int fildes, const void *buf, size_t nbytes);
```

  - Devuelve el número de bytes escritos

Llamadas al sistema: Ejemplo (V)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
int main (void)
{
  int fd, bytes_escritos;                    /*descriptor de fichero*/
  char buffer[100];
  mode_t modo = S_IRWXU; /* modo de r, w y x para el propietario*/

  strcpy(buffer, "Esto es un ejemplo\n");
  if ((fd = open ("ejemplo", O_RDWR | O_CREAT, modo))== -1)
        /*abre el fichero ejemplo en modo lectura/escritura o lo
        crea si no existe */
        perror ("Error al abrir o crear el fichero");
        /*muestra un mensaje de error si no puede abrir/crear el fichero*/
  else
        bytes_escritos = write(fd, buffer, strlen(buffer));
        /* escribe buffer de tamaño sizeof(buffer) en fd */
  exit(0);
}
```

Llamadas al sistema: Ejemplo (VI)

- Análogamente se puede leer el contenido del fichero "ejemplo" de la siguiente forma

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#define BLKSIZE  100                /*tamaño del buffer*/
int main(void)
{
  int fd, bytes_read;
  char buffer[BLKSIZE];
  if ((fd = open("ejemplo", O_RDWR))==-1)
        perror("Error al abrir el fichero");
  else
  {
        bytes_read = read(fd, buffer, BLKSIZE);
        buffer[bytes_read]='\0';
        printf("%s son %d \n", buffer, bytes_read);
  }
  exit(0);
}
```


Gestión de Procesos en POSIX

Llamada	Función
fork	Crea un proceso
getpid	Obtiene el identificador del proceso
getppid	Obtiene el identificador del proceso padre
wait	Detiene un proceso hasta que alguno de sus hijos termina
waitpid	Detiene un proceso hasta que un determinado hijo termina
exec	(Conjunto de llamadas) Cambia la imagen de memoria de un proceso
exit	Fuerza la terminación del proceso que la invoca
kill	Solicita la terminación de otro proceso

Llamada al sistema *fork* (I)

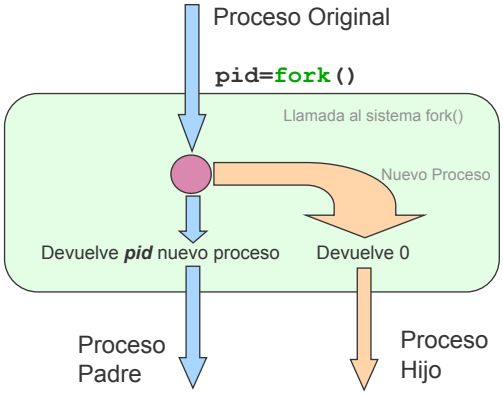
- Provoca la creación de un nuevo proceso
- El nuevo proceso hijo es una copia exacta del padre
- Sintaxis

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

pid_t fork(void);
```

- Devuelve 0 al proceso hijo y el PID del hijo al padre
- El proceso hijo hereda del padre los siguientes atributos
  - UID (identificador de usuario) y GID (identificador de grupo)
  - Variables de entorno y argumentos
  - Descriptores de ficheros
  - Directorio de trabajo actual y raíz
  - Máscara de modo de creación de ficheros.

Llamada al sistema *fork* (II)

- Bifurcación *fork*:



Llamada al sistema *fork* (III)

- El proceso difiere del padre en los siguientes atributos
  - El proceso hijo tiene su propio identificador PID
  - El hijo tiene su propio PPID (*Parent Process ID*)
  - Se inicializa la información de contabilidad del proceso hijo
- Errores posibles al ejecutar *fork*
  - Se excede el número máximo de procesos permitido en el sistema
  - Se excede el número de procesos máximo permitido a un usuario
  - Memoria insuficiente

Llamadas al sistema *getpid* y *getppid*

- *getpid*: Obtiene el PID del proceso que la invoca
- *getppid*: Obtiene el PID del padre del proceso que la invoca
- Sintaxis:

```
#include <sys/types.h>
#include <unistd.h>
uid_t getpid (void);
uid_t getppid (void);
```

- El tipo *uid_t* no es más que un entero definido en *types.h*

Llamada al sistema *fork*: Ejemplo

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

int main (void)
{
  pid_t pid;
  pid = fork();
```

```
[/u0/sitr/sitr001/procesos]p1
Soy el hijo, mi PID es 15673 y mi PPID es 15672
Soy el padre, mi PID es 15672 y el PID de mi    hijo es 15673
[/u0/sitr/sitr001/procesos]
```

Bifurcación

```
  switch (pid)
  {
        case -1: perror ("No se ha podido crear el hijo");
                 break;
        case 0:  printf("Soy el hijo, mi PID es %d y mi PPID es %d\n",
                                 getpid(), getppid());
                 break;

        default: printf ("Soy el padre, mi PID es %d y el PID de mi
                          hijo es %d\n", getpid(), pid);
  }

  exit(0);
}
```


Llamada al sistema *exit*

- Fuerza la terminación de un proceso devolviendo un código de error
- Está implícita en la terminación de todos los procesos

```
#include <stdlib.h>

void exit( int status);
```

- Al finalizar un proceso se recuperan todos los recursos asignados al proceso
- Si el padre del proceso que termina está ejecutando *wait* se le notifica la terminación y el código de terminación
- Si el padre no ejecuta *wait* el proceso se transforma en un proceso *zombie* (huérfano) hasta que es adoptado por ***init***

Llamada al sistema *wait*

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

pid_t wait( int *stat_loc)
pid_t waitpid( pid_t pid, int *stat_loc, int options);
```

- *wait* suspende la ejecución del proceso que la llama hasta que finaliza alguno de sus hijos (devuelve el PID del hijo)
- En caso de error (el proceso no tiene hijos o estos ya han terminado) devuelve -1
- *stat_loc* es un puntero a un entero. Si es distinto de NULL en la llamada, en él se guarda el estado devuelto por el hijo.
- *waitpid* proporciona métodos más flexible y potentes para esperar a los hijos (se puede esperar a uno en particular)

Llamada al sistema *wait* : Ejemplo

```
#include <sys/types.h>
#include <stdlib.h>
#include <sys/wait.h>
int main(void)
{
   pid_t childpid, childdead;
   int i;
   childpid = fork();
   if (childpid == -1)
   {
         perror("fork no pudo crear el hijo");
         exit(1);
   }
   else  if (childpid == 0)
   {
      printf ("Soy el hijo (PID %d) y voy a contar hasta 100000 \n", getpid());
      for (i=0;i<100000; i++) {}
   }
   else
   {    printf("Soy el padre (PID %d) y voy a esperar a mi hijo (PID %d)\n",
                                                   getpid(),childpid);
      if ((childdead = wait(0))==-1)
         perror("No he podido esperar al hijo");
      else
         printf("Mi hijo con pid %d, ha muerto\n",childdead);
   }
   exit(0);
}
```

```
[/u0/sitr/sitr001/procesos]p2
Soy el hijo (PID 18296) y mi padre es (PID 18295), voy a contar hasta 100000
Soy el padre (PID 18295) y voy a esperar a mi hijo (PID 18296)
Mi hijo con pid 18296, ha muerto
[/u0/sitr/sitr001/procesos]
```

Bifurcación

Familia de llamadas al sistema ***exec***

- La familia de llamadas ***exec*** cambia la imagen de memoria de un proceso por el contenido de un fichero ejecutable
- La forma habitual de utilizar la combinación ***fork/exec*** es dejar que el hijo ejecute ***exec*** mientras el padre continua la ejecución normal
- Existen 6 variaciones de ***exec*** que se distinguen por la forma en que son pasados los argumentos de la línea de comandos y el entorno que utilizan.
- Las seis variaciones se pueden agrupar en dos grupos
  - Llamadas ***execl*** : *execl, execlp execle*
  - Llamadas ***execv***: *execv, execvp, execve*

Llamadas ***execl***

- Las llamadas *execl* (*execl, execlp, execle*) pasan los argumentos de la línea de comando del programa mediante una lista.
- Son útiles si se conoce el número de argumentos que se van a pasar

```
#include <uninst.h>
int execl (const char *path, const char *arg0, ...,
                const char argn, char /*NULL*/);
int execle (const char *path, const char *arg0, ...,
                const char argn, char /*NULL*/, char
int execlp (const char *file, const char *arg0, ...,
                const char argn, char /*NULL*/);
```

- *execle* permite pasar nuevos valores de variables de entorno
- *execlp* permite tomar el path por defecto del entorno

Llamadas *execv*

- Pasan los argumentos de la línea de comando en un array de argumentos

```
#include <uninst.h>

int execv (const char *path, const char *argv[]);
int execve (const char *path, const char *argv[],
    const char *envp[]);
int execvp (const char *file, const char *argv[]);
```

- *execve* permite pasar nuevos valores de variables de entorno
- *execvp* permite tomar el path por defecto del entorno

Llamada *execl* : Ejemplo (I)

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

int main (void)
{
  pid_t childpid;
  childpid = fork();
  if (childpid == -1)
  {
          perror("Error al crear el hijo");
          exit(1);
  }
  else if (childpid ==0)
  {
          printf("Soy el hijo y voy a ejecutar ls -l\n");
          if (execl("/bin/ls", "ls", "-l", NULL) <0)
          {
            perror("Error al ejecutar ls\n");
            exit (1);
          }
  }
  else
  {
          printf("Soy el padre, espero al hijo y termino\n");
          wait(0);
  }
  exit(0);
}
```

```
Soy el hijo y voy a ejecutar ls -l
total 50
-rwxr-xr-x   1 sitr001  alu        24256 Oct 20 20:31 hola
-rw-rw-rw-   1 sitr001  alu           65 Oct 20 20:31 hola.c
-rwxr-xr-x   1 sitr001  alu        24988 Nov 20 13:57 p3
Soy el padre, espero al hijo y termino
```

Bifurcación

Llamada *execv* : Ejemplo (II)

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

int main (int argc, char *argv[])
{
  pid_t childpid;
  childpid = fork();
  if (childpid == -1)
  {
          perror("Error al crear el hijo");
          exit(1);
  }
  else if (childpid ==0)
  {
          if(argc>1)
          {
             printf("Soy el hijo y voy a ejecutar el comando %s\n",argv[1]);
             if (execvp(argv[1], &argv[1]) <0)
                     perror("Error al ejecutar el comando");
          } else printf("Soy el hijo y no has escrito ningún comando\n");
  }
  else
  {
          printf("Soy el padre, espero al hijo y termino\n");
          wait(0);
  }
  exit(0);
}
```

Bifurcación

```
[/u0/sitr/sitr001/programas]p4 ls
Soy el hijo y voy a ejecutar el comando ls
hola     hola.c   p3       p4
Soy el padre, espero al hijo y termino
```


Llamada al sistema *kill*

```
#include <sys/types.h>
#include <sys/signal.h>

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

- *kill* envía una señal (*sig*) al proceso identificado por *pid*. La señal para terminar un proceso es SIGKILL (9)
- En caso de error devuelve -1
- El proceso a terminar debe pertenecer al mismo usuario o tener privilegios sobre él.

- *Comando Consola:*
  kill [-s signal] pid

Llamada al sistema *kill* : Ejemplo

```
#include <sys/types.h>
#include <stdlib.h>
#include <sys/signal.h>
int main(void)
{
   pid_t childpid;
   int res;
   childpid = fork();
   if (childpid == -1)
   {
          perror("fork no pudo crear el hijo");
          exit(1);
   }
   else  if (childpid == 0)
   {
          printf ("Soy el hijo (PID %d) y me voy a colgar\n");
          while(1) {}
   }
   else
   {
          printf("Soy el padre y voy a terminar a mi hijo\n");
          sleep(2);  /* espero 2 segundos -> Estado Suspendido */
          if ((res = kill(childpid, SIGKILL))==-1)
                 perror("no he podido terminar al hijo");
          else
                 printf("Mi hijo con pid %d, ha muerto\n",childpid);
   }
   exit(0);
}
```

```
Soy el hijo (PID 21024) y me voy a colgar
Soy el padre y voy a terminar a mi hijo
Mi hijo con pid 21024, ha muerto
```

Bifurcación

Ejercicio:

Implementar una aplicación concurrente que calcule el cuadrado de los 20 primeros números naturales y almacene el resultado, repartiendo la tarea entre dos procesos:

- Crear dos procesos Hijos:
  - 1º. Realiza la operación sobre los números impares
  - 2º. Realiza la operación sobre los números pares
- El proceso padre espera la terminación de los hijos, '***obtiene***' el resultado de cada hijo y muestra los valores ordenados en pantalla

Ejercicio: crear dos procesos hijo

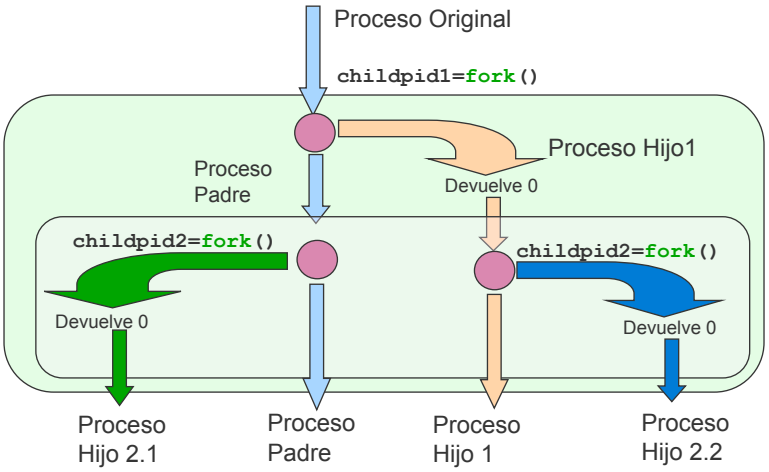
1ª Opción -> Incorrecta

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

int main(void)
{
   pid_t childpid1 childpid2;
   int i;

   childpid1 = fork();
   childpid2 = fork();

   if ((childpid1 == -1) || (childpid2 == -1)
   {
           perror("fork no pudo los procesos hijo");
           exit(1);
   }

   // ..........
}
```

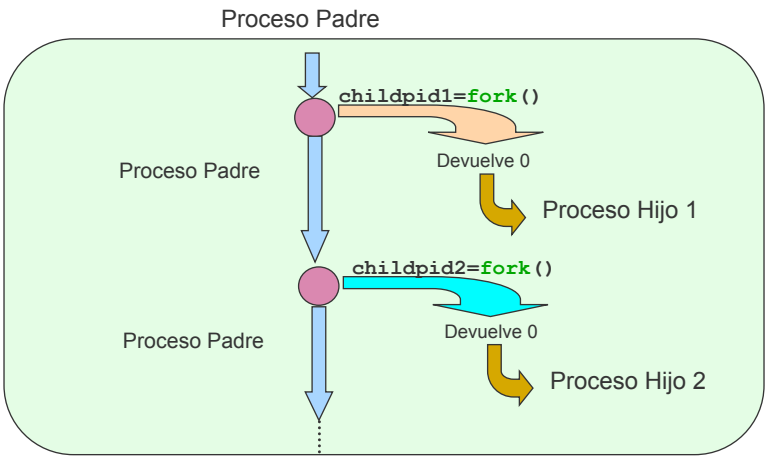
Doble Bifurcación

=> 4 procesos

Doble Bifurcación: Incorrecta

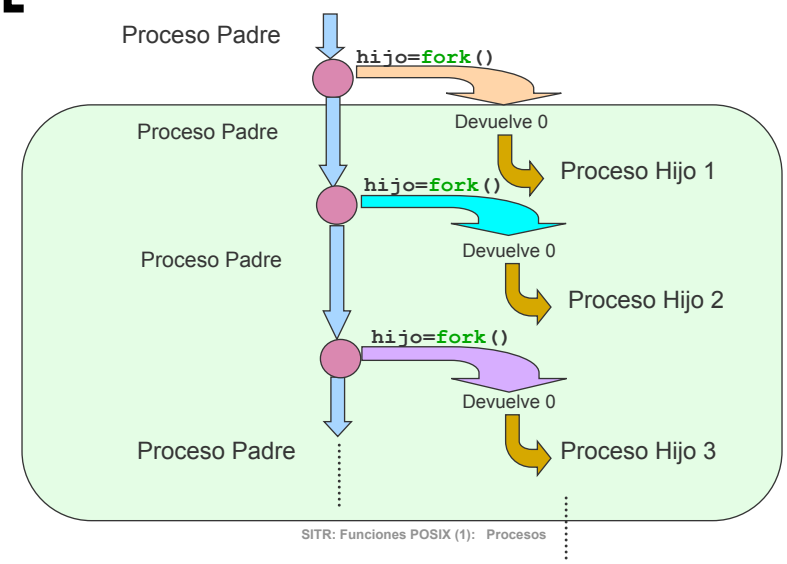


Doble Bifurcación: Correcta




```
int main(void)
{
  pid_t childpid1 childpid2;
  printf ("Soy el padre, mi PID es %d y voy a crear el primer hijo.\n", getpid());

  childpid1 = fork();
  switch (childpid1)
  {
      case -1: perror ("No se ha podido crear el hijo");
               break;
      case 0:  printf("Soy el hijo1, mi PID es %d y voy a calcular los números impares %d\n",
                              getpid());
                    // Cálculo potencias números impares
               break;
      default: printf ("Soy el padre, mi PID es %d y voy a crear el segundo hijo.\n",
                              getpid());
               childpid2 = fork();

               if (childpid2 == -1)
               {    perror("fork no pudo crear el hijo");   break;  }
               else  if (childpid2 == 0)
               {
                      printf ("Soy el hijo2, mi PID es %d y voy a calcular
                                     los números pares %d\n", getpid());
                    // Cálculo potencias números pares
               }
               else
               {
                    printf("Soy el padre y voy a esperar que terminen los hijos\n");
                    wait(0); // espera al primer hijo que termine
                    wait(0); // espera al siguiente hijo que termine
                    // Juntar resultados y mostrar en pantalla
               }
  }
  exit(0);
}
```

Creación Sistemática de Procesos



Crea n-1 procesos hijo

```
pid_t hijo;

/* Crea los procesos */
for (i = 1; i<n; ++i)
{
        hijo = fork();
        if (hijo ==-1) {
                perror("No se puede crear el proceso");
                exit(-1);
        }
        else if(hijo==0)
                break; // si es un hijo terminar el for (solo el padre crea los hijos)
}
```

Bifurcación

```
/* i actúa como identificador del proceso en el padre i==n */
printf("i:%d  proceso ID:%d  padre ID:%d\n",i, getpid(), getppid());

// El Padre espera que terminen los hijos
if(i==n)
{
        printf("Soy el Padre [PID:%d] y voy a esperar a los hijos\n", getpid());
        for(i=1;i<n;i++)   wait(0);
}
```

Ejercicio: Compartir Datos

- Espacio de memoria separado
  - Las variables de cada proceso son independientes y no accesibles al resto de procesos
- Soluciones:
  - **Ficheros**:
    - almacenar los resultados en ficheros accesibles al resto de procesos
  - **Memoria Compartida**:
    - Espacio de memoria especial compartida por varios procesos
  - **Mensajes**:
    - Notificaciones entre procesos en la misma máquina
  - **Sockets**:
    - Puertos de comunicación entre procesos distribuidos
  - **Threads**:
    - Espacio de memoria compartido para todos los hilos de ejecución de un proceso