Llamadas al Sistema para la Creación de Procesos

Transparencias realizadas por M. Curiel. Se utiliza material ya publicado en la WEB
y elaborado por todos los prof. que hemos dado el curso.

Llamada al Sistema Fork

- La única forma que existe en UNIX para crear un proceso es duplicando un proceso ya existente. El proceso *init* es el ancestro de todos los procesos que se crean durante una sesión.
- El proceso que hace una solicitud para la creación de otro proceso se conoce como proceso padre, y el proceso creado se conoce como proceso hijo.

Llamada al Sistema Fork

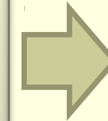
- Después de la duplicación, ambos procesos (padre e hijo) son idénticos salvo en el PID (Process ID). Es decir, ambos procesos tienen el mismo espacio de direcciónes: el área de datos, la pila y ambos continúan ejecutando el mismo código.
- Es posible que el proceso hijo reemplace posteriormente el código que ejecuta su padre por otro programa.

Llamada al Sistema Fork

- Un proceso crea otro proceso utilizando la llamada al sistema fork

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

pid_t fork(void)



Librerías que se deben Incluir.

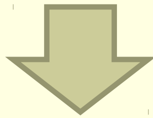
Procesos

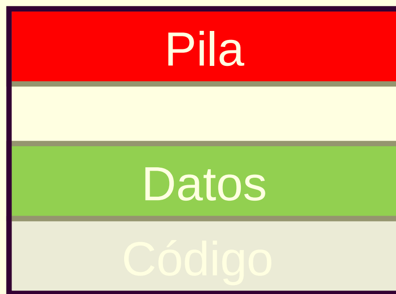
Ejemplo:

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

main( )
{
        ...
        if ((childpid=fork())==0) {
                fprintf(stderr,"Soy el hijo con PID %d\n",getpid())
        else if (childpid > 0)
                fprintf(stderr,"Soy el padre con PID %d\n",
getpid());

}
```

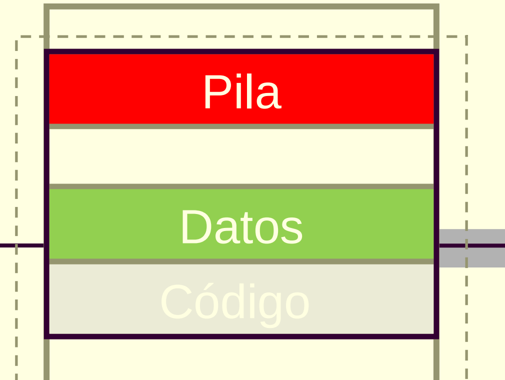
Devuelve 0 al hijo y el PID del hijo
Al padre.

Imagen en la Memoria de Procesos Unix





Proceso 1

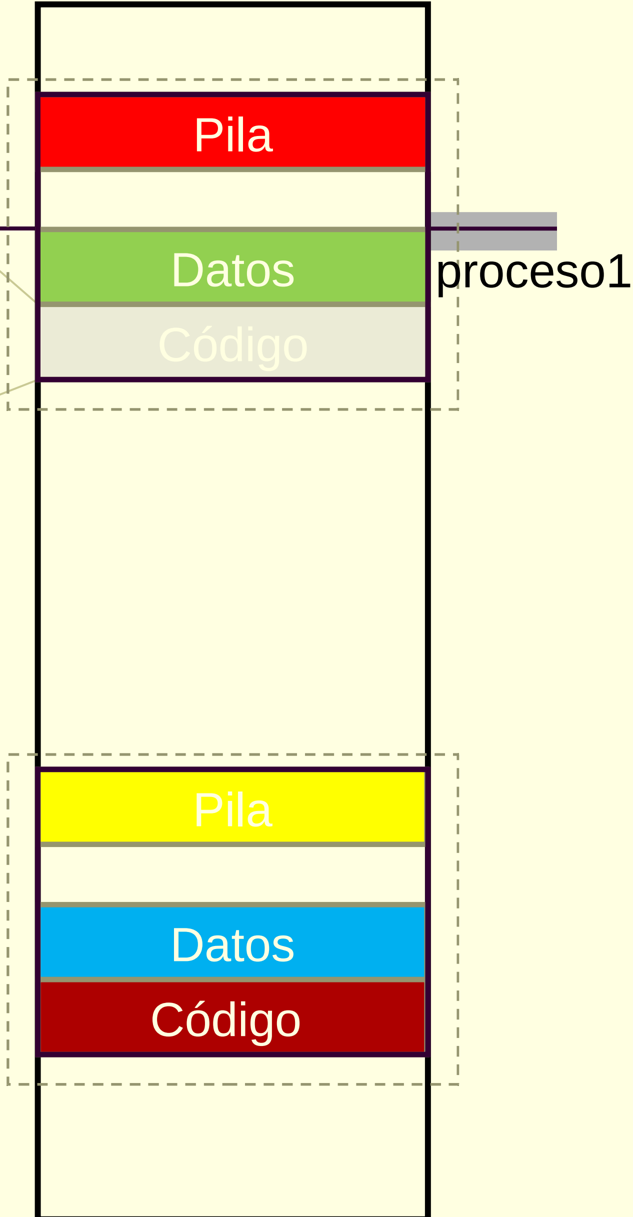


Proceso 2

Proceso 3

PC

```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf("soy el padre"),
else
 printf("soy el hijo);

}
```



Detalle de la Memoria Principal: Código del Proceso 1, antes de ejecutar la llamada al Sistema fork()

```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf(“soy el padre”),
else
 printf(“soy el hijo);

}
```

PC

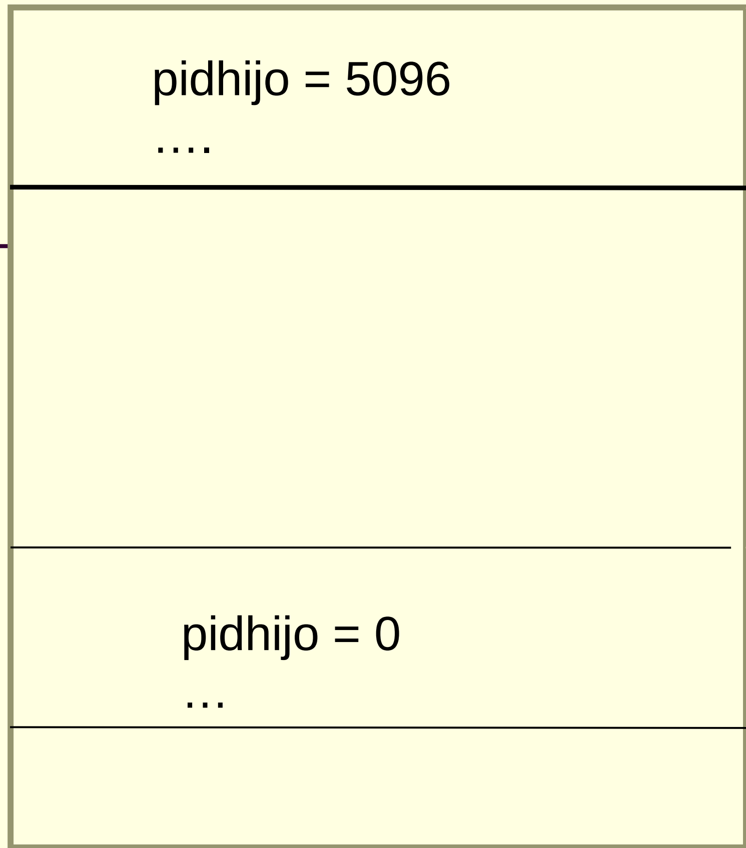
Pila
Datos
Código
Padre

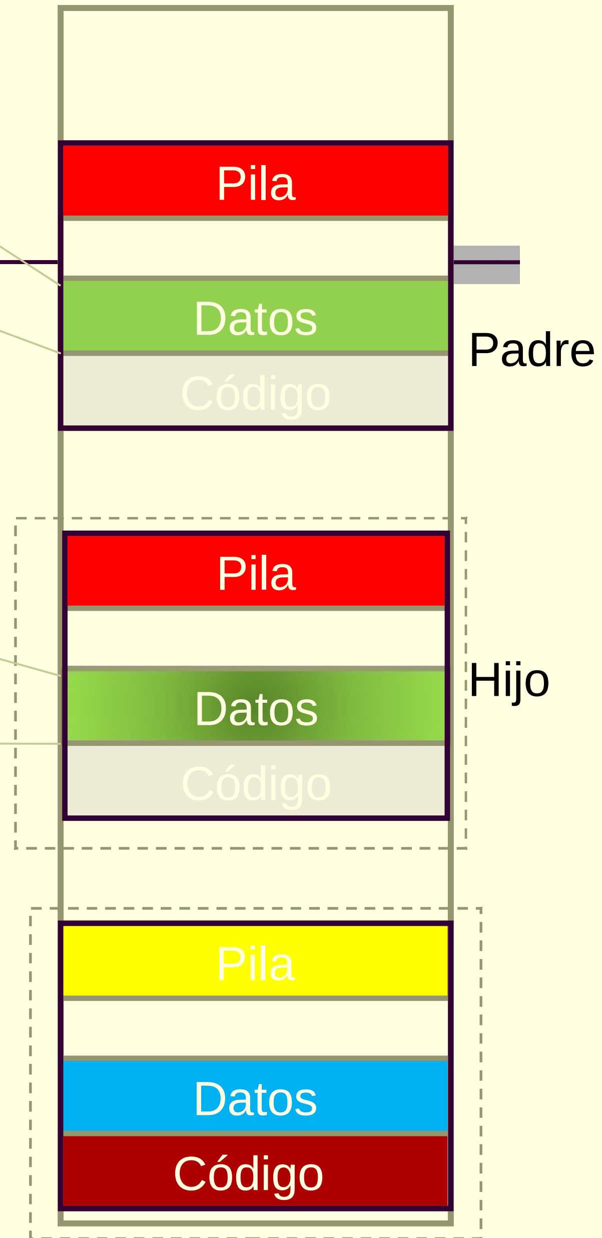
Pila
Datos
Código
Hijo

Pila
Datos
Código

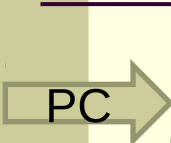
Memoria Principal

Detalle de la Memoria Principal:
Qúe pasa en la memoria cuando se ejecuta la llamada al sistema *fork*?.

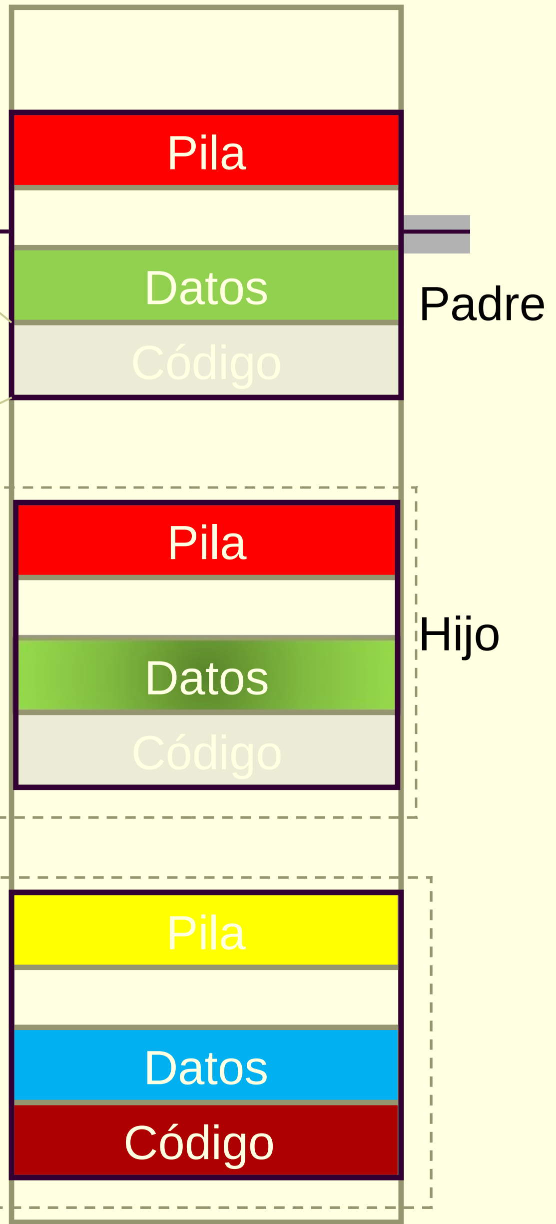




Detalle de la Memoria Principal:
Valor de la variable *pidhijo* en ambos procesos




```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf(“soy el hijo”),
else
 printf(“soy el padre);

}
```



Detalle de la Memoria Principal: Ejecución del proceso padre después del *fork*

```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf(“soy el hijo”),
else
 printf(“soy el padre);

}
```

PC

Pila
Datos
Código
Padre

Pila
Datos
Código
Hijo

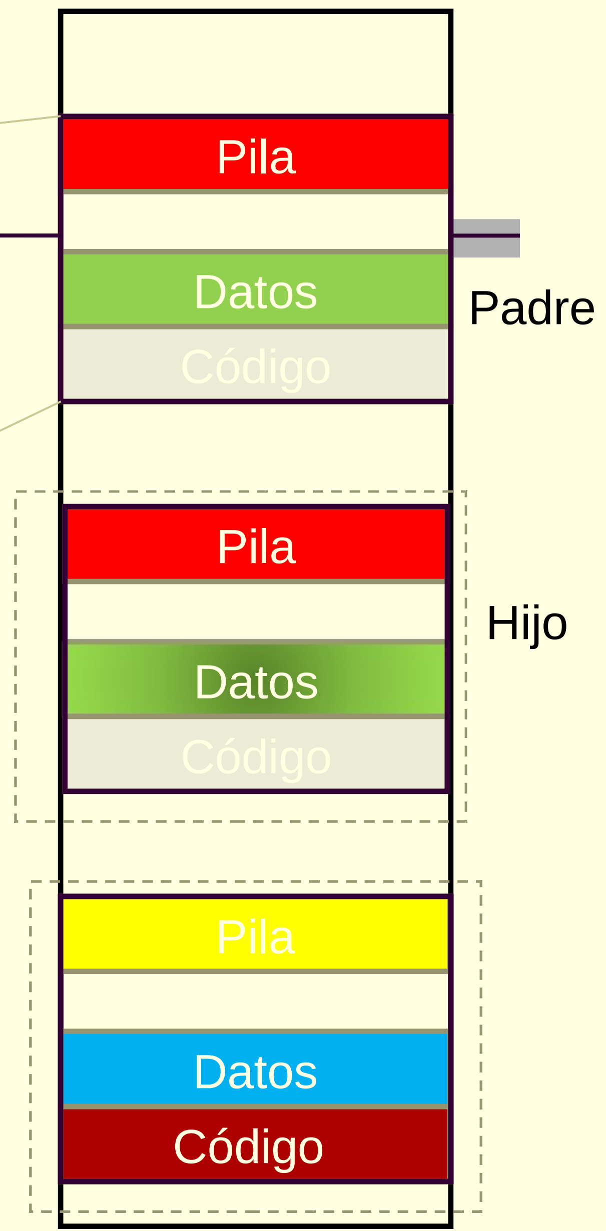
Pila
Datos
Código

Memoria Principal

Detalle de la Memoria Principal: Ejecución del proceso Padre después del *fork*

```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf(“soy el hijo”),
else
 printf(“soy el padre);

}
```

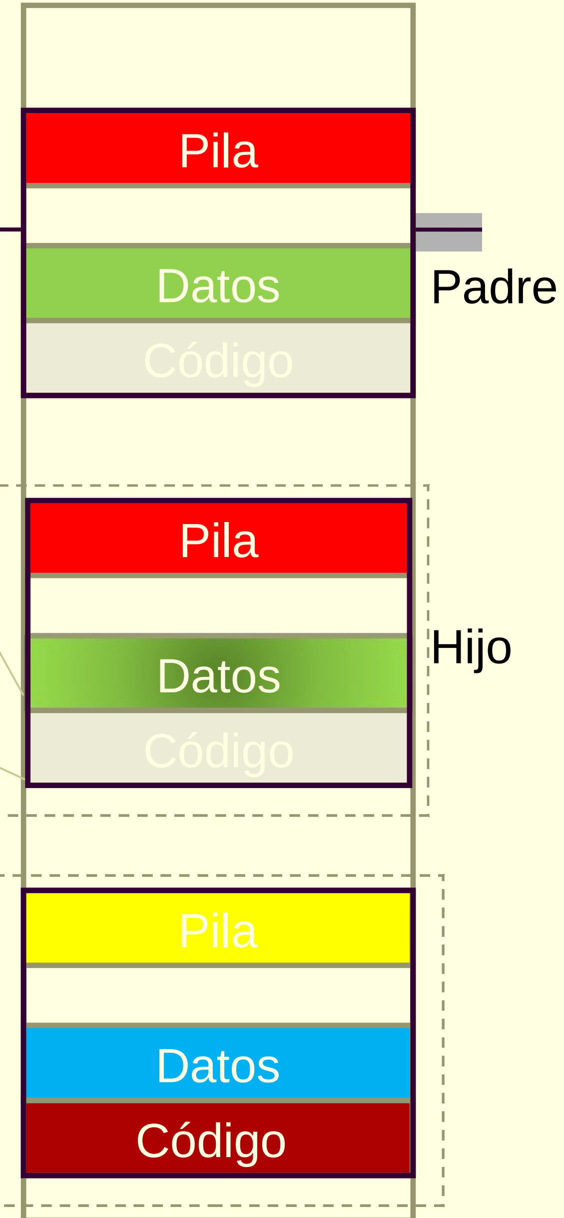
PC



Detalle de la Memoria Principal: Ejecución del proceso hijo después del *fork*

PC

```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf(“soy el hijo”),
else
 printf(“soy el padre);

}
```



Detalle de la Memoria Principal: Ejecución del proceso hijo después del *fork*

```
main() {

….
pidhijo = fork();
if (pidhijo == 0) {
 printf(“soy el hijo”),
else
 printf(“soy el padre”);

}
```

PC

Pila

Datos

Código

Padre

Pila

Datos

Código

Hijo

Pila

Datos

Código

Memoria Principal

Detalle de la Memoria Principal: Ejecución del proceso hijo después del *fork*

Qué hereda el proceso hijo?

- Cada proceso tiene un identificador (PID). Las llamadas al sistema getpid() y getppid() permiten obtener el PID del propio proceso y el de su padre, respectivamente.
- El hijo hereda la mayor parte de los atributos del padre incluyendo el ambiente y los privilegios. También hereda alguno de los recursos del padre tales como archivos y dispositivos abiertos.
- Los tiempo de uso de CPU para el hijo son inicializados en 0.

Qué hereda el proceso hijo??

- El hijo no obtiene los bloqueos que el padre mantiene.
- Si el padre ha colocado una alarma, el hijo no recibe notificación alguna al momento que esta expira.
- El hijo su ejecución comienza sin señales pendientes

Ejemplos del uso de la llamada al sistema fork.

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

main() {
        x = 0;
        fork();
        x = 1;
}
```

Después del fork existen dos procesos independientes, cada uno tiene su propia copia de la variable *X*.

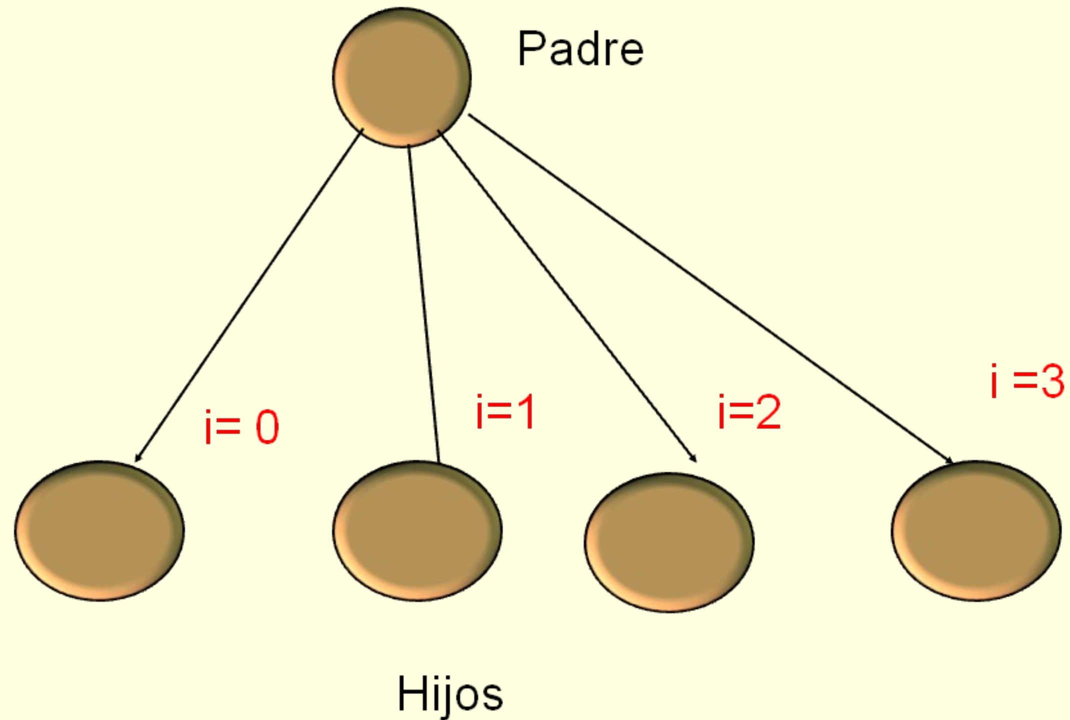
No están modificando la misma localidad de memoria.

Ambos procesos están ejecutando la misma instrucción sobre su copia de la Variable X

Abanico de Procesos

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
main() {

   int status, i,nprocesos;
   pid_t childpid;

   nprocesos = 4;
   for (i = 0; i < nprocesos; ++i) {
     if ((childpid = fork()) < 0) {
       perror("fork:");
       exit(1);
     }
     // Codigo que ejecutaran los hijos
     if (childpid == 0) {
       printf("Soy el hijo con pid %ld\n", getpid());
       exit(0);
     }
   } //fork

} // main
```

Abanico de Procesos



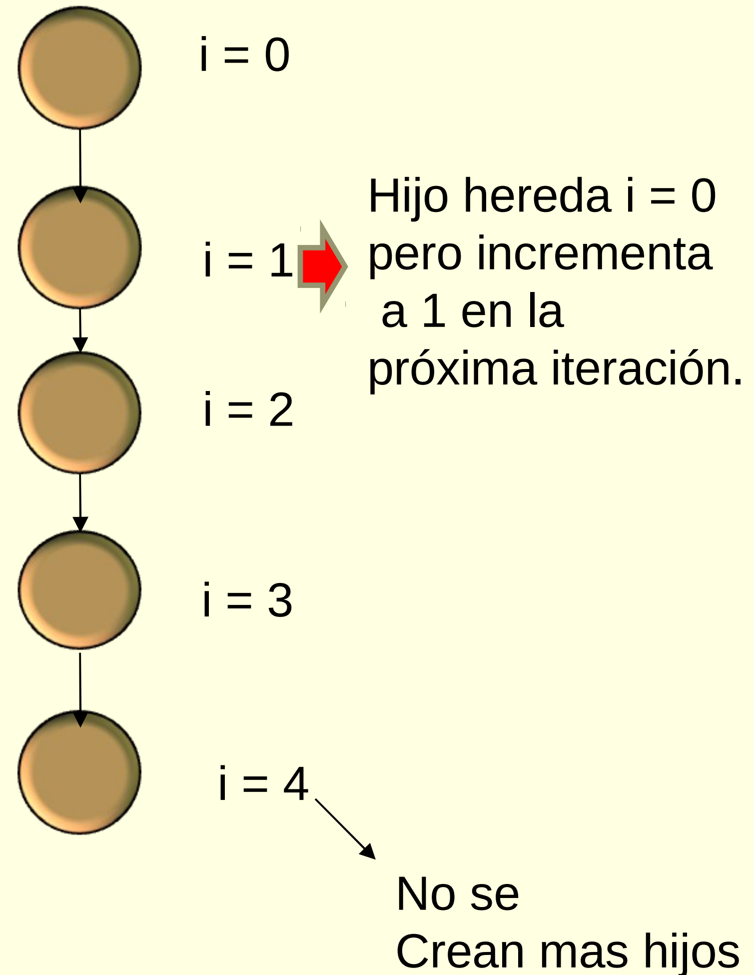
Valor de la variable i heredado por cada uno de los procesos creados.

Salida del Programa

```
// Salida al ejecutar el programa
Soy el hijo con pid 3788
Soy el hijo con pid 3790
Soy el hijo con pid 3789
Soy el hijo con pid 3791
```

Cadena de Procesos

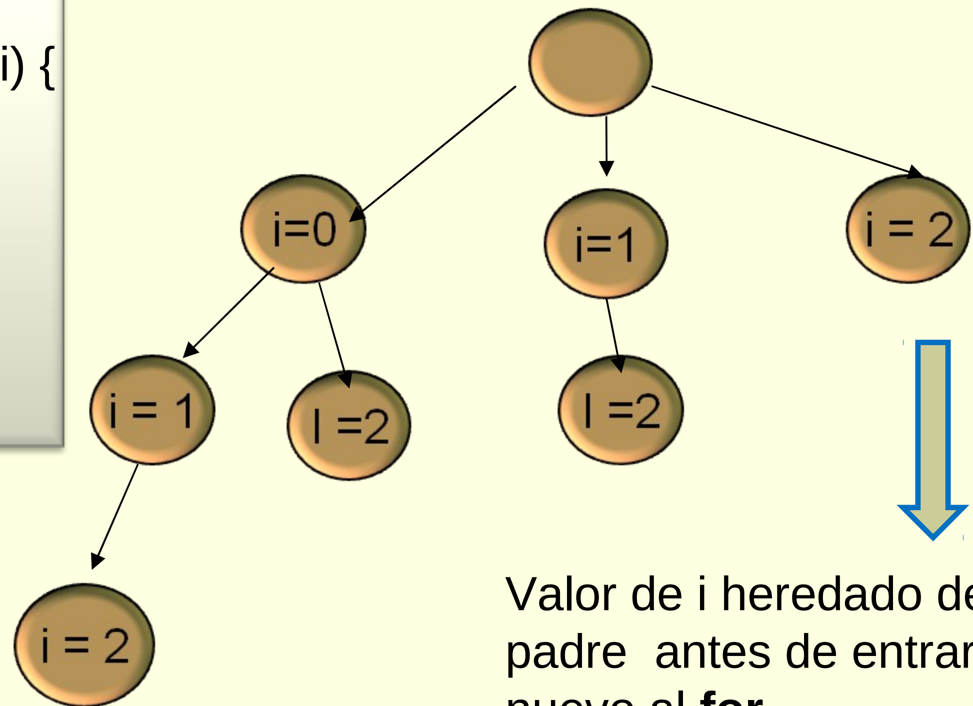
```
nprocesos = 4;
for (i = 0; i < nprocesos; i++) {
  if ((childpid = fork()) < 0) {
    perror("fork:");
    exit(1);
  }
  // Codigo que ejecutaran los hijos
  if (childpid > 0)  //los padres se mueren,
                     // el hijo sigue en el for.
     break;
}
```




```
For: CMP i, nprocesos
       BEQ   i , 4, salir
……. //codigo fork

      ADD i,i, 1
      JMP For
```

Árbol de Procesos (Padres e hijos crean procesos)

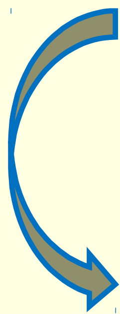
```
nprocesos = 3;
  for (i = 0; i < nprocesos; ++i) {
    if ((childpid = fork()) < 0) {
      perror("fork:");
      exit(1);
    }
}
```



Llamada al sistema:Wait

- Después de que un padre crea un proceso hijo, ambos continúan ejecutándose de forma concurrente.
- Si un padre desea esperar porque el hijo termine, debe ejecutar wait o waitpid.

*pid_t wait(int *status)*
*pid_t waitpid(pid_t pid, int *status, int opciones)*



Prototipo de las funciones de librería.

Wait, WaitPid

- La llamada al sistema *wait* detiene al proceso que la invoca hasta que un hijo de éste termine (cualquiera). *wait* regresa de inmediato si el proceso que la invoca no tiene hijos o si el hijo ya ha terminado.
- Si *wait* regresa debido a la terminación de un hijo, el valor devuelto es igual al PID del proceso que finaliza. De lo contrario devuelve –1 y pone un valor en *errno*.

Wait, WaitPid

- *status* es un apuntador a una variable entera. El proceso que invoca *wait debe colocar como parámetro la dirección de una variable entera. En esta variable se almacenará* el estado devuelto por el hijo (bits más significativos).

*pid_t wait(int *status)*

La librería POSIX especifica ciertas macros para analizar el estado devuelto por el hijo (WIFEXITED, WEXITSTUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED y WSTOPSIG).

Wait, WaitPid

- La función de librería *waitpid* proporciona un método para esperar por un hijo en particular. Tiene tres parámetros. Si *pid* es igual a –1, *waitpid* espera por cualquier proceso. Si *pid* es positivo entonces espera al hijo cuyo PID es *pid*. El segundo parámetro es la variable donde se devolverá el status, y el último permite especificar, por ejemplo, si el padre se bloqueará o no al esperar por los hijos.

*pid_t waitpid(pid_t pid, int *status, int opciones)*

Ejemplos: Abanico con Wait

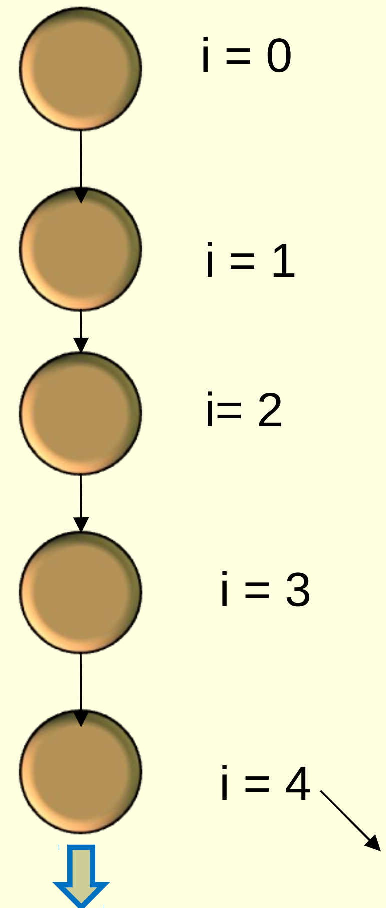
```
nprocesos = 5;
  for (i = 0; i < nprocesos; ++i) {
    if ((childpid = fork()) < 0) {
      perror("fork:");
      exit(1);
    }
    // Codigo que ejecutaran los hijos
    if (childpid == 0) {
      printf("Soy el hijo con pid %ld\n", getpid());
      exit(0);
    }
  }
  // El padre espera  que terminen todos los hijos que creo.
  for (i = 0; i < nprocesos; ++i)
    wait(&status);
  printf("El padre termina\n");

}
```

Abanico que no es concurrente

```
for (i = 0; i < nprocesos; ++i) {
    if ((childpid = fork()) < 0) {
      perror("fork:");
      exit(1);
    }
    // Codigo que ejecutaran los hijos
    if (childpid == 0) {
      printf("Soy el hijo con pid %ld\n", getpid());
      exit(0);
    }
    wait(&status); // evita la concurrencia
  }
```

Cadena con el Wait

```
nprocesos = 5;
  for (i = 0; i < nprocesos; ++i) {
    if ((childpid = fork()) < 0) {
      perror("fork:");
      exit(1);
    }
    // Codigo que ejecutaran los hijos
    if (childpid > 0)
       break;
  }
  while ((childpid = wait(&status)) != -1);
   printf("Soy el hijo con pid %ld, Mi padre es %ld\n",
getpid(), getppid());
```



El primer proceso que sale del *wait* (le retorna -1), es el primero en i mprimir el mensaje.

Salida del Programa

```
//Salida al ejecutar el programa
Soy el hijo con pid 3798, Mi padre es 3797
Soy el hijo con pid 3797, Mi padre es 3796
Soy el hijo con pid 3796, Mi padre es 3795
Soy el hijo con pid 3795, Mi padre es 3794
Soy el hijo con pid 3794, Mi padre es 3793
Soy el hijo con pid 3793, Mi padre es 3730
```

Ejemplo utilizando el valor de status

```
  if ((child_pid = fork()) < 0) {
      perror("fork:");
      exit(1);
  }
    // Codigo que ejecutara el hijo
  if (child_pid == 0) {
    //….
      exit(0);
    }
  // El padre espera el hijo  que creo.
  while((child_pid = wait(&status)) == -1);
  if (!status)
    printf("El hijo %ld termino normalmente, el estado devuelto es 0\n", (long)
child_pid);
  else if (WIFEXITED(status))
    printf("El hijo %ld termino normalmente, el estado devuelto es %d\n", (long)
           child_pid, WEXITSTATUS(status));
  else if (WIFSIGNALED(status))
    printf("El hijo %ld termino debido a una senal no atrapada\n", (long) child_pid);
```

Se debe incluir wait.h

Utilizando el Valor de Status

```
R = 0;
/* Creacion de los 2 procesos en abanico */
   for (i = 0; i < 2; i++) {
      if ((childpid = fork()) < 0) {
         perror("fork:"); return(1);
      }
      /* Codigo que ejecutan los hijos */
      if (childpid == 0) return(2);
   }

/* El padre espera por todos los hijos y colecta los resultados */
   for (i = 0; i < 2; i++) {
      wait(&status);
      printf("\n\n El proceso %d retorna: %d",i,WEXITSTATUS(status));
      if (WIFEXITED(status)) R = R + WEXITSTATUS(status);
   }

  printf("\n\n Resultado: %d\n\n", R);
```

Procesos Zombies y Huérfanos

- Proceso Zombie: Es un proceso que terminó pero, aún su padre no ha invocado la llamada al sistema wait para recuperar su status de terminación. Los zombies permanecen en el sistema hasta que algún proceso los "espere".
- Proceso Huérfano: Si un padre termina y no espera a uno de sus hijos, el hijo se convierte en huérfano y es adoptado por el proceso *init*. El proceso *init* espera periódicamente por todos los descendientes de modo que en algún momento los procesos huérfanos desaparecen del sistema.

Ejemplo de un Proceso Zombie

```
main( )
{
        ...
        if ((pid=fork() > 0) {
            while (1)
                    sleep(1000);
        }
        else
            exit(42);
}
```

$ zombie.exe &
[1] 13545
$ ps
PID TT STAT TIME COMMAND
13535 p2 S 0:00 -ksh (ksh) ...the Shell
13545 p2 S 0:00 zombie.exe ... the parent process
13546 p2 Z 0:00 <defunct> ... the zombie child process
13547 p2 R 0:00 ps

$ kill 13545
[1] Terminated

Ejemplo de un Proceso Huérfano

```
#include <stdio.h>
main() {
    pid_t pid;
    printf(“I'm the original process with PID %d y PPID %d\n”, getpid(), getppid());
   pid = fork();
   if (pid > 0) {
       printf(“I'm the parent process with PID %d y PPID %d\n”, getpid(), getppid());
       printf(“My child's PID is %d\n”, pid);
   } else {
       sleep(5); /* Make sure that the parent terminates first */
       printf(“The child finishes\n”);
   }
   exit(0);
}
```

Terminación de Un Proceso (exit)

- Cuando termina un proceso, el sistema de operación cancela temporizadores y señales pendientes, libera recursos de memoria virtual y otros tipos de recursos, cierra archivos abiertos, etc. y actualiza las estadísticas necesarias.
- Además, notifica al proceso padre si éste realizó una llamada al sistema *wait*.
- La llamada toma un parámetro entero *status* que indica el estado de terminación del programa. Para indicar una terminación normal se hace uso del valor 0 para *status*. Los valores de *status* distinto de 0 y definidos por el programador indican errores.

Llamadas al Sistema Exec

- La llamada *fork* duplica al proceso que la invoca, pero muchas veces hay aplicaciones que requieren que el proceso hijo ejecute un código diferente al de su padre.
- La familia de llamadas *exec* permite sustituir el proceso que invoca la llamada por otro código ejecutable.

Exec: Prototipos

int execl(const char *path, const char *arg0,...,const char *argn, NULL)
int execlp(const char *file, const char *arg0,...,const char *argn, NULL)
int execle(const char *file, const char *arg0,...,const char *argn, NULL, const char *envp[])

path corresponde a la ruta de acceso y el nombre del programa, especificado ya sea como un nombre con el camino absoluto o relativo al directorio de trabajo
argl corresponde a los argumentos de la línea de comandos seguidos de un apuntador NULL.
envp[] se pasa el ambiente al nuevo programa. Las variables de ambiente son un conjunto de tuplas (nombre, valor). Cada par es representado por una cadena de la forma NAME=VALUE, y a este conjunto se le conoce como ambiente del programa. Ejemplo: HOME: contiene el directorio home del usuario. Para acceder directamente al ambiente se usa la variable global

```
extern char *environ[];
```

Esto provee el ambiente como un vector de apuntadores que apunta a cada elemento del ambiente del programa. La última entreda contiene NULL.

Exec

- execl y execlp se diferencian en que el último utiliza la variable de ambiente PATH para buscar el ejecutable.

Exec

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

path corresponde a la ruta de acceso y el nombre del programa, especificado ya sea como un nombre con el camino absoluto o relativo al directorio de trabajo

argv[] corresponde a un arreglo de argumentos

Ejemplos

```
#include <stdio.h>
main() {
    printf("I'm process %d and I'm about to exec an ls -l\n", getpid());
    execl("/bin/ls", "ls", "-l", NULL);
    printf("ESta línea nunca debiera ejecutarse");
}
```

```
$ exec.exe
I'm process %d and I'm about to exec an ls -l
total 92
-rw-r--r-- 1 mcuriel cs 1226 Feb 6 16:56 archivo.txt
-rw-r--r-- 1 mcuriel cs 3512 Feb 6 16:52 archivo.txt~
drwxr-xr-x 2 mcuriel cs 4096 Jan 28 11:27 Consultas
-rw-r--r-- 1 mcuriel cs 20393 Jan 23 16:37 ejemplosop
```

Ejemplos Exec

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
Int main(void )  {
  int status;
  pid_t chilpid;
   ...
  if ((childpid=fork())==-1) {
     perror("Error al ejecutar fork");
     exit(1);
  } else if (childpid==0) {
     if (execl("/bin/ls","ls","-l",NULL) < 0) {
        perror("Falla en la ejecucion de exec");
        exit(1);
     }
  }
 }
```

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

int main(void) {

  int status, fd1;
  if (!fork()) {
                printf ("Un ejemplo con el exec\n");
                execlp("sleep", "sleep", "5",  NULL);
                 perror("Hubo un error\n");
  }
  wait(&status);

}
```

```
void main(int argc, char *argv[])
{
     int status;
     pid_t chilpid;
     ...
     if ((childpid=fork())==-1) {
        perror("Error al ejecutar fork");
        exit(1);
     } else if (childpid==0) {
        if (execvp(argv[1],&argv[1]) < 0) {
            perror("Falla en la ejecucion del comando");
            exit(1);
        }
    } else {
        while (childpid != wait(&status))
        if (childpid == -1) break;
             exit(0);

 }
}
```

$ ejecutar ls -l

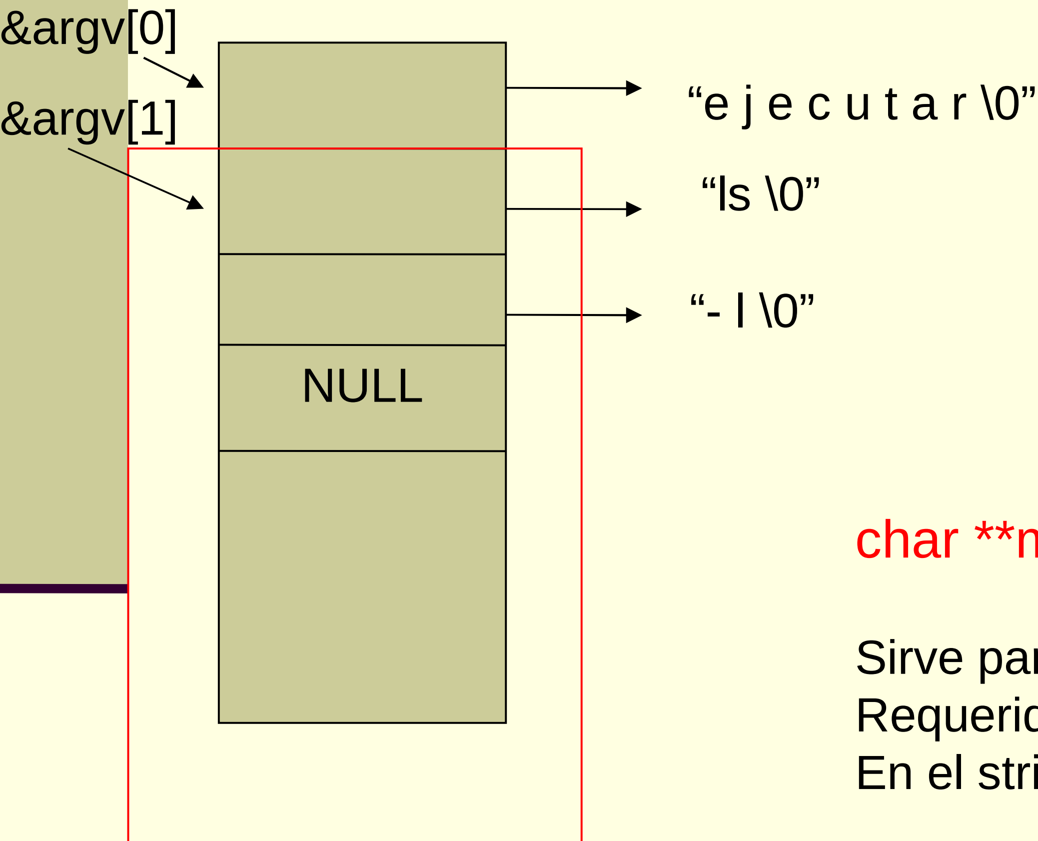
argv0

argv1

Se pasa un arreglo de la forma de los argv, pero a partir de la primera posición

Nombre del programa a ejecutar, se usa la variable PATH para encontrar el ejecutable

Argumentos del main()



char **makeargv(char *s)

Sirve para construir un arreglo de la forma
Requerida por las llamadas al sistema exec.
En el string “s” uno coloca todos los argumentos

Bibliografía

- K. Robbins and Steven Robbins. UNIX: Programación Práctica. Guía para la concurrencia, la Comunicación y los Multihilos. Prentice Hall. 1997.
- Grahan Glass. Unix for Programmers and Users. Prentice Hall. 1993.

Procesos

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

void main(int argc, char *argv[])
{
int status;
int chilpid;

     ...

if ((childpid=fork())==-1) {
        perror("Error al ejecutar fork");
        exit(1);
     } else if (childpid==0) {
        if (execvp(argv[1],&argv[1]) < 0) {
            perror("Falla en la ejecucion del comando");
                exit(1);
         }
      } else
         while (childpid != wait(&status))
               if (childpid == -1) && errno !=EINTR))
                   break;
      exit(0);
}
```