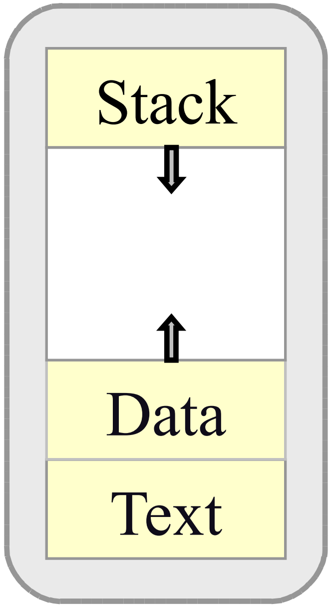
Un proceso es una instancia de un programa que está en ejecución

Cada instancia tiene su propio espacio de direcciones y estado de ejecución

Cuando se ejecuta un programa, el sistema de operación copia el módulo ejecutable en una imagen del programa en memoria principal



Prof. Angela Di Serio

BY NC SA

Unix identifica los procesos por un valor entero (process ID o PID)

Además cada proceso tiene un PID del proceso que lo creó (proceso padre)

Si el padre termina antes que su hijo, éste es adoptado por un proceso del sistema

```
#include <unistd.h>

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

```
printf("My process ID is %ld\n", getpid());
```

Prof. Angela Di Serio

CC BY NC SA

El comando ps muestra información de los procesos

Se ejecuta a través de la línea de comando de UNIX

Por defecto muestra información de los procesos del usuario

UID – user ID
PID – process ID
PPID – parent process ID
C – (obsolete)

STIME – starting time of the process
TTY – controlling terminal
TIME – cumulative execution time
CMD – command name

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Jan08 ?        00:00:01 init [2]
root         2     0  0 Jan08 ?        00:00:00 [kthreadd]
root         3     2  0 Jan08 ?        00:00:01 [ksoftirqd/0]
root         6     2  0 Jan08 ?        00:00:00 [migration/0]
root         7     2  0 Jan08 ?        00:00:00 [watchdog/0]
root         8     2  0 Jan08 ?        00:00:00 [migration/1]
root        10     2  0 Jan08 ?        00:00:00 [ksoftirqd/1]
root        12     2  0 Jan08 ?        00:00:00 [watchdog/1]
```

Prof. Angela Di Serio

BY NC SA

Creación de Procesos

Un proceso crea otro a través de la llamada al sistema fork()

El proceso que invoca el fork se conoce como padre y el nuevo proceso es el hijo

El hijo recibe una copia de la imagen en memoria del proceso padre. Cada proceso tiene su propio espacio de direcciones

Ambos procesos continuan con la instrucción que sigue después del fork

Prof. Angela Di Serio

cc BY NC SA

```
#include <unistd.h>

pid_t fork(void);
```

El valor de retorno del fork se usa para determinar quién es el proceso padre y quién es el hijo

El valor de retorno para el hijo es 0 (cero) mientras que el padre recibe el PID del hijo

Prof. Angela Di Serio

Bitvise xterm - garon.tip - auliya.ldc.usb.ve:22

```
FORK(2)                    Linux Programmer's Manual                   FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a new process by duplicating the calling process.  The new process, referred
       to as the child, is an exact duplicate of the calling process, referred  to  as  the  parent,
       except for the following points:

       *  The  child has its own unique process ID, and this PID does not match the ID of any exist-
          ing process group (setpgid(2)).

       *  The child's parent process ID is the same as the parent's process ID.

       *  The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

       *  Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to
          zero in the child.

       *  The child's set of pending signals is initially empty (sigpending(2)).

       *  The child does not inherit semaphore adjustments from its parent (semop(2)).

       *  The child does not inherit record locks from its parent (fcntl(2)).
```

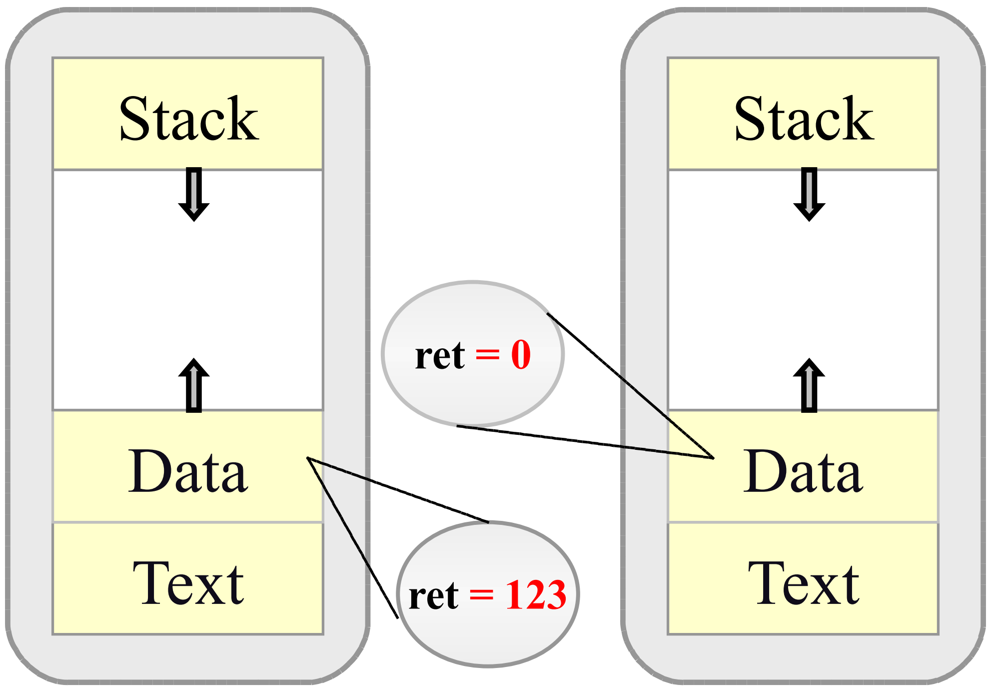
Prof. Angela Di Serio

```
       *  The child process is created with a single thread—the one that called fork().  The
          entire  virtual  address space of the parent is replicated in the child, including
          the states of mutexes, condition variables, and other pthreads objects; the use of
          pthread_atfork(3) may be helpful for dealing with problems that this can cause.

       *  The child inherits copies of the parent's set of open file descriptors.  Each file
          descriptor in the child refers to the same open file description (see open(2))  as
          the corresponding file descriptor in the parent.  This means that the two descrip-
          tors share open file status flags, current  file  offset,  and  signal-driven  I/O
          attributes (see the description of F_SETOWN and F_SETSIG in fcntl(2)).

       *  The  child  inherits  copies of the parent's set of open message queue descriptors
          (see mq_overview(7)).  Each descriptor in the child refers to the same  open  mes-
          sage  queue description as the corresponding descriptor in the parent.  This means
          that the two descriptors share the same flags (mq_flags).

       *  The child inherits copies of the parent's  set  of  open  directory  streams  (see
          opendir(3)).   POSIX.1-2001  says  that the corresponding directory streams in the
          parent and child may share the directory stream positioning; on  Linux/glibc  they
          do not.

RETURN VALUE
       On success, the PID of the child process is returned in the parent, and 0 is returned
       in the child.  On failure, -1 is returned in the parent, no child process is created,
       and errno is set appropriately.

ERRORS
       EAGAIN fork()  cannot allocate sufficient memory to copy the parent's page tables and
              allocate a task structure for the child.

       EAGAIN It was not possible to create a new process because the caller's  RLIMIT_NPROC
              resource  limit  was encountered.  To exceed this limit, the process must have
              either the CAP_SYS_ADMIN or the CAP_SYS_RESOURCE capability.
```

Prof. Angela Di Serio

cc BY NC SA

fork()



Prof. Angela Di Serio

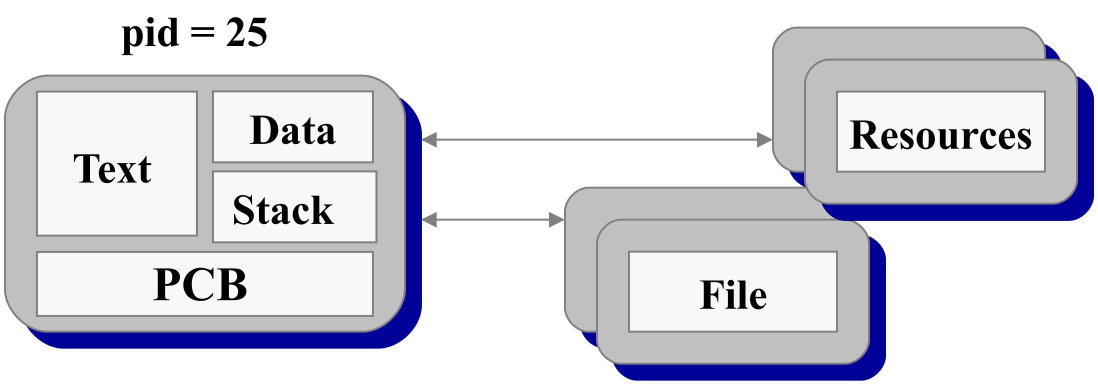
El proceso hijo hereda del padre una copia idéntica de su memoria

Registros de CPU

Todos los archivos abiertos

La ejecución prosigue de forma concurrente a partir de la instrucción que sigue el fork

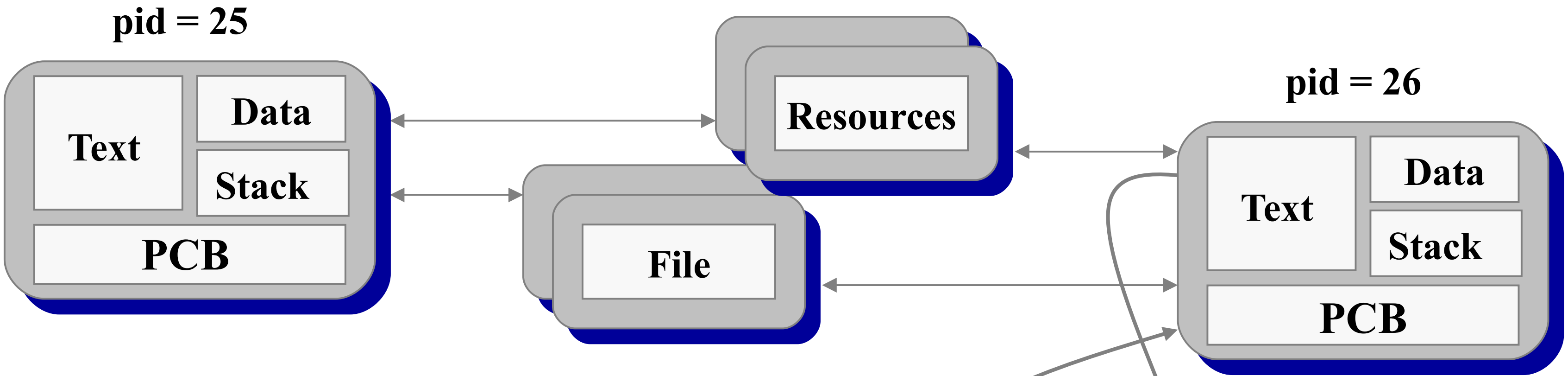
BY NC SA




```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
             <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

UNIX

Prof. Angela Di Serio



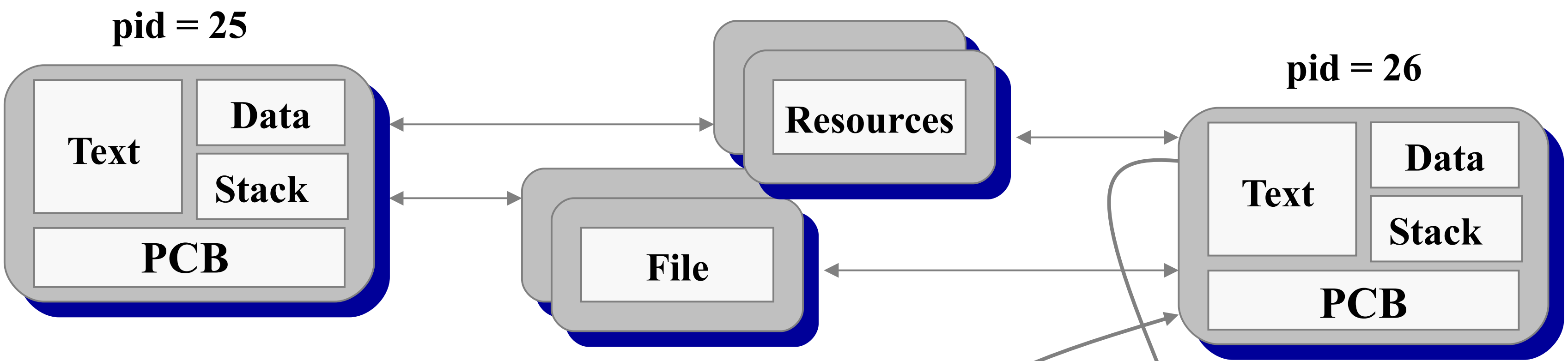

```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

UNIX

ret = 0

```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

Prof. Angela Di Serio



ret = 26

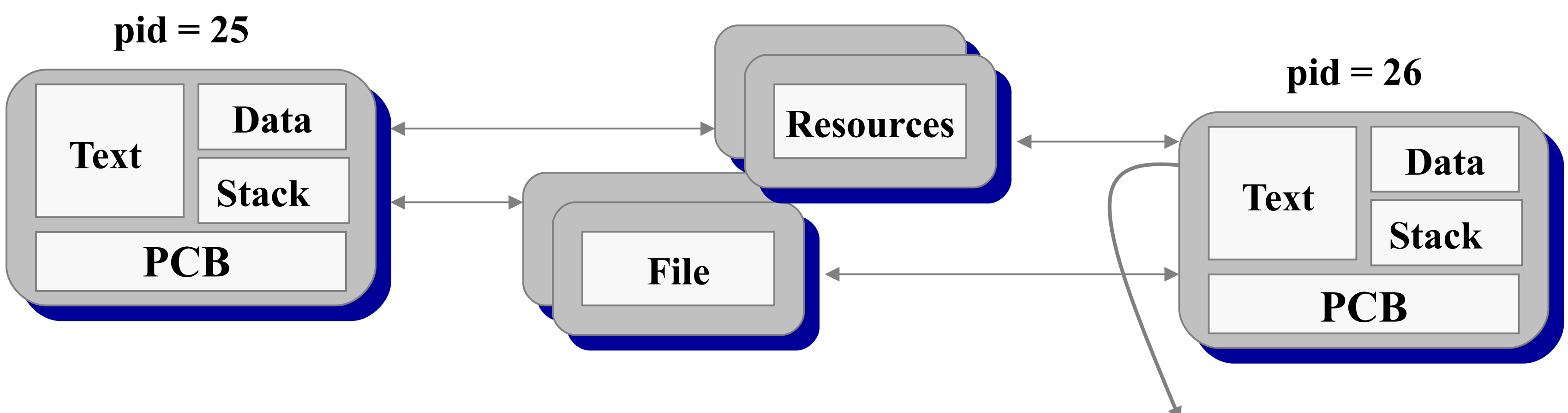
```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

UNIX

ret = 0

```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

Prof. Angela Di Serio




```
ret = fork();
switch(ret)
{
    case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
            <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

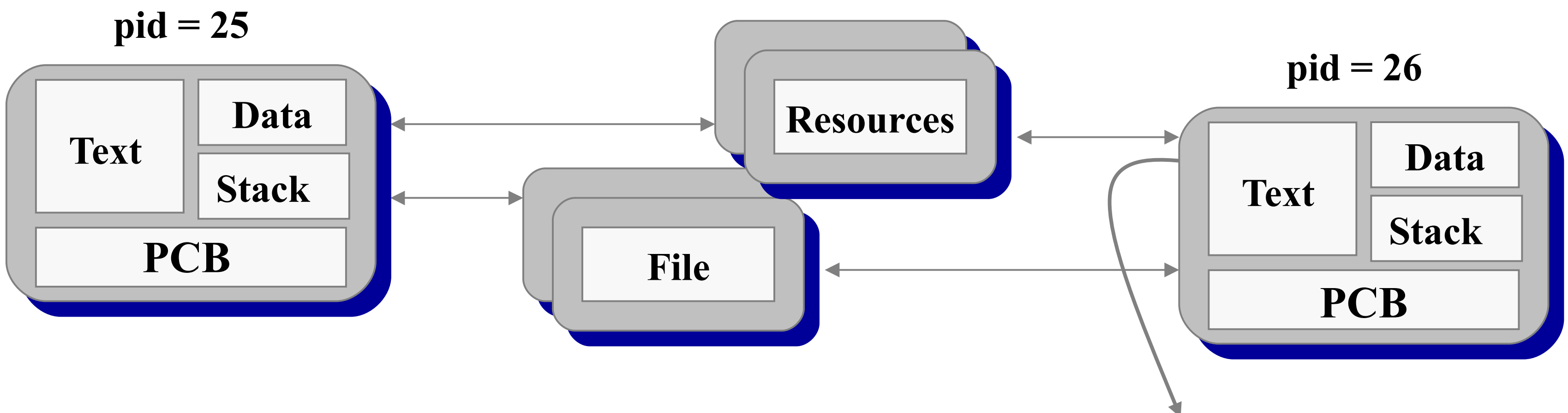
ret = 26

UNIX

```
ret = fork();
switch(ret)
{
    case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
            <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

ret = 0

Prof. Angela Di Serio



ret = 26

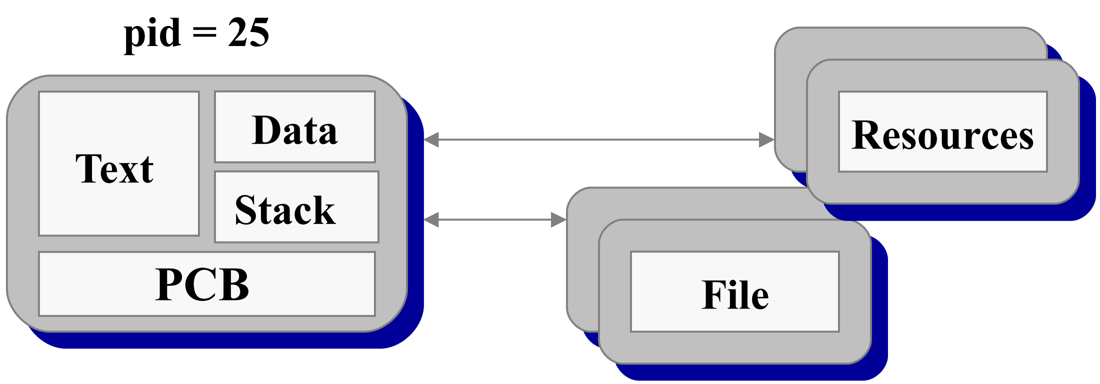
```
ret = fork();
switch(ret)
{
   case -1:
          perror("fork");
          exit(1);
   case 0:  // I am the child
          <code for child >
          exit(0);
   default:  // I am parent ...
          <code for parent >
          wait(0);
}
```

UNIX

ret = 0

```
ret = fork();
switch(ret)
{
   case -1:
          perror("fork");
          exit(1);
   case 0:  // I am the child
          <code for child >
          exit(0);
   default:  // I am parent ...
          <code for parent >
          wait(0);
}
```

Prof. Angela Di Serio




```
ret = fork();
switch(ret)          ret = 26
{
    case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
            <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            Wait(0);
        <....>
}
```

UNIX

Prof. Angela Di Serio

Para finalizar la ejecución, el hijo puede invocar la llamada al sistema exit(*resultado*)

Qué hace un exit?

Qué hace un exit?

Salva el resultado= argumento de la llamada exit
Cierra todos los archivos abiertos, conexiones
Libera la memoria
Chequea si el padre está vivo
Si el padre vive, mantiene el valor de retorno hasta que el padre lo solicita con un wait. En este caso el proceso hijo no muere y entra en un estado de zombie o defunct
Si el padre no está vivo, el hijo termina (muere)

Prof. Angela Di Serio

BY NC SA

```
EXIT(3)                    Linux Programmer's Manual                    EXIT(3)

NAME
       exit - cause normal process termination

SYNOPSIS
       #include <stdlib.h>

       void exit(int status);

DESCRIPTION
       The  exit() function causes normal process termination and the value of status & 0377
       is returned to the parent (see wait(2)).

       All functions registered with atexit(3) and on_exit(3) are  called,  in  the  reverse
       order  of  their  registration.   (It  is  possible for one of these functions to use
       atexit(3) or on_exit(3) to register an additional function to be executed during exit
       processing;  the new registration is added to the front of the list of functions that
       remain to be called.)  If one of these functions does  not  return  (e.g.,  it  calls
       _exit(2),  or  kills  itself  with a signal), then none of the remaining functions is
       called, and further exit processing (in particular, flushing of stdio(3) streams)  is
       abandoned.   If  a  function  has  been  registered multiple times using atexit(3) or
       on_exit(3), then it is called as many times as it was registered.

       All open stdio(3) streams are flushed and closed.  Files created  by  tmpfile(3)  are
       removed.

       The  C  standard  specifies two constants, EXIT_SUCCESS and EXIT_FAILURE, that may be
       passed to exit() to indicate successful or unsuccessful termination, respectively.

RETURN VALUE
       The exit() function does not return.

CONFORMING TO
       SVr4, 4.3BSD, POSIX.1-2001, C89, C99.
```

El proceso padre puede querer esperar a que sus procesos hijos finalicen

Se utiliza la llamada al sistema wait() para esperar por un proceso hijo

El proceso padre se bloquea hasta que alguno de sus hijos termine. La llamada retorna el PID del proceso que finalizó o -1 si no hay procesos hijos

Para esperar por un proceso en particular se usa waitpid()

Prof. Angela Di Serio

cc BY NC SA

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

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

```
WAIT(2)                       Linux Programmer's Manual                       WAIT(2)

NAME
       wait, waitpid, waitid - wait for process to change state

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

       pid_t wait(int *status);

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

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       waitid():
           _SVID_SOURCE || _XOPEN_SOURCE >= 500 || _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED
           || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L

DESCRIPTION
       All  of  these  system  calls  are  used  to wait for state changes in a child of the calling
       process, and obtain information about the child whose state has changed.  A state  change  is
       considered  to  be: the child terminated; the child was stopped by a signal; or the child was
       resumed by a signal.  In the case of a terminated child, performing a wait allows the  system
       to release the resources associated with the child; if a wait is not performed, then the ter-
       minated child remains in a "zombie" state (see NOTES below).
```

Prof. Angela Di Serio

```
    If a child has already changed state, then these calls return  immediately.   Otherwise  they
    block  until  either  a child changes state or a signal handler interrupts the call (assuming
    that system calls are not automatically restarted using the SA_RESTART flag of sigaction(2)).
    In  the  remainder  of  this page, a child whose state has changed and which has not yet been
    waited upon by one of these system calls is termed waitable.

wait() and waitpid()
    The wait() system call suspends execution of the calling process until one  of  its  children
    terminates.  The call wait(&status) is equivalent to:

        waitpid(-1, &status, 0);

    The  waitpid()  system call suspends execution of the calling process until a child specified
    by pid argument has changed state.  By default, waitpid() waits only for terminated children,
    but this behavior is modifiable via the options argument, as described below.

    The value of pid can be:

    < -1   meaning  wait  for  any  child process whose process group ID is equal to the absolute
           value of pid.

    -1     meaning wait for any child process.

    0      meaning wait for any child process whose process group ID is  equal  to  that  of  the
           calling process.

    > 0    meaning wait for the child whose process ID is equal to the value of pid.
```

Prof. Angela Di Serio

BY NC SA

```
If  status  is not NULL, wait() and waitpid() store status information in the int to which it
points.  This integer can be inspected with the following  macros  (which  take  the  integer
itself as an argument, not a pointer to it, as is done in wait() and waitpid()!):

WIFEXITED(status)
       returns  true  if  the  child  terminated  normally,  that  is,  by calling exit(3) or
       _exit(2), or by returning from main().

WEXITSTATUS(status)
       returns the exit status of the child.  This consists of the least significant  8  bits
       of the status argument that the child specified in a call to exit(3) or _exit(2) or as
       the argument for a return statement in main().  This macro should only be employed  if
       WIFEXITED returned true.

WIFSIGNALED(status)
       returns true if the child process was terminated by a signal.

WTERMSIG(status)
       returns  the  number  of  the signal that caused the child process to terminate.  This
       macro should only be employed if WIFSIGNALED returned true.

WCOREDUMP(status)
       returns true if the child produced a core dump.  This macro should only be employed if
       WIFSIGNALED  returned  true.   This  macro is not specified in POSIX.1-2001 and is not
       available on some UNIX implementations (e.g., AIX, SunOS).  Only use this enclosed  in
       #ifdef WCOREDUMP ... #endif.

WIFSTOPPED(status)
       returns  true  if  the child process was stopped by delivery of a signal; this is only
```

Prof. Angela Di Serio

BY NC SA

getpid retorna el identificador del proceso

```
pid = getpid();
```

getppid retorna el identificador del padre

Procesos Zombies pueden ser vistos usando el comando **ps** .

En el caso de procesos zombies aparecerá <**defunct**> en la columna de comando

Prof. Angela Di Serio

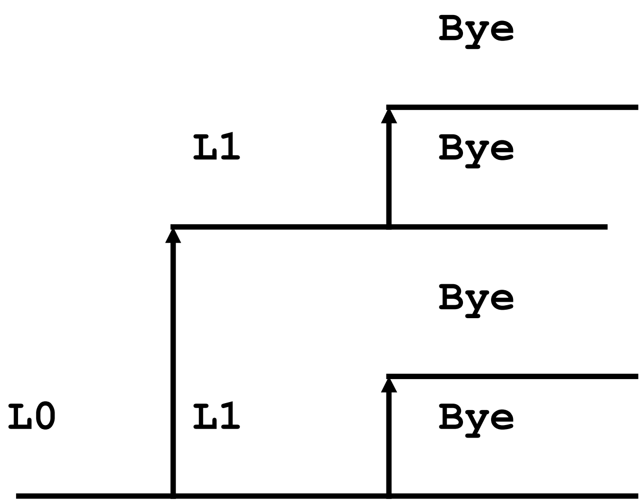
CC BY NC SA

```
int main()
{
   pid_t pid;
   int x = 1;

   pid = fork();
   if (pid != 0) {
     printf("parent: x = %d\n", --x);
     exit(0);
   } else {
     printf("child: x = %d\n", ++x);
     exit(0);
   }
}
```

Prof. Angela Di Serio

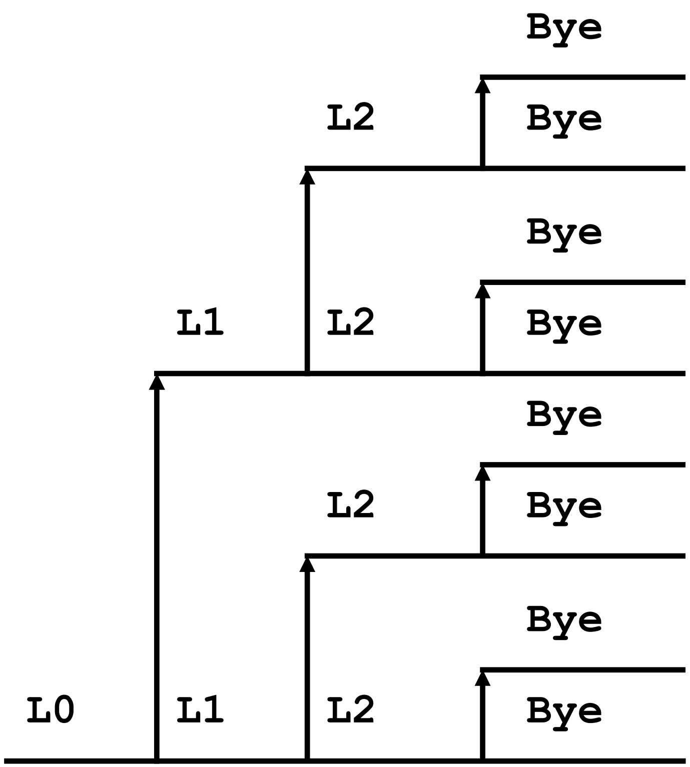
cc BY NC SA

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Prof. Angela Di Serio

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Prof. Angela Di Serio

BY NC SA

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```

Prof. Angela Di Serio

cc BY NC SA

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```

execv ejecuta un archivo transformando el proceso llamador en un nuevo proceso . Después de la ejecución correcta de execv no hay retorno al proceso llamador

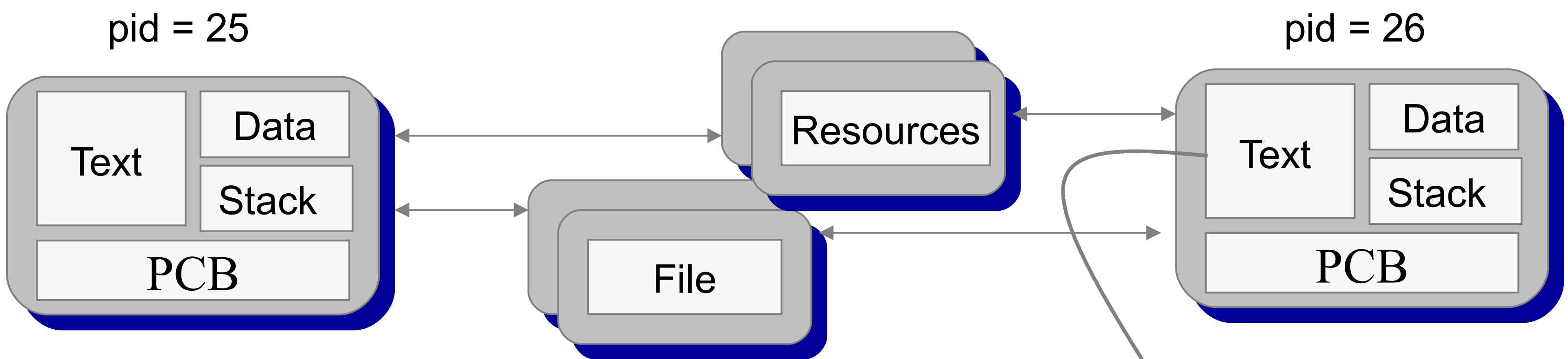
execv(const char * path, char * const argv[])

path camino completo al archivo a ser ejecutado

argv arreglo de argumentos para el programa a ejecutar. Cada argumento es una cadena de caracteres terminado con el caracter nulo. El primer argumento es el nombre del programa y el último es NULL

Prof. Angela Di Serio

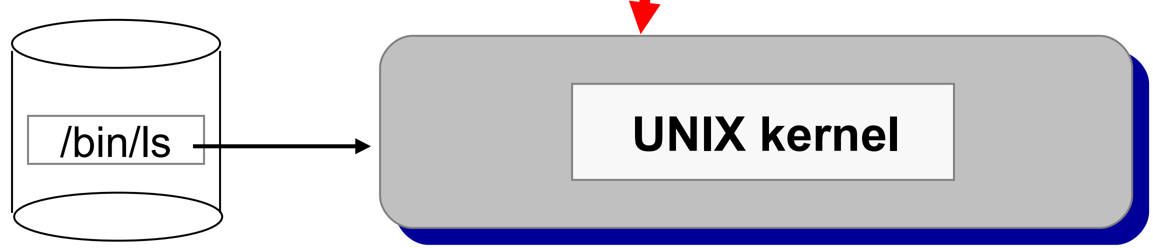
BY NC SA

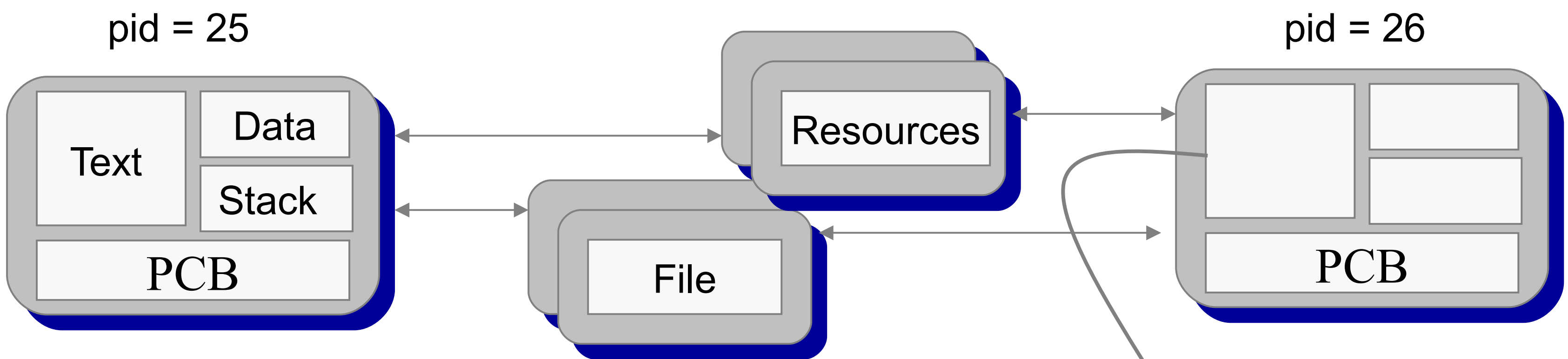



```
char * argv[ ] = {“/bin/ls”, 0};
<…>
int cpid = fork( );
if (cpid = = 0) {
    execv(argv[0], argv);
    exit(0);
}
<parent code>
wait(&cpid);
```

cpid = 0

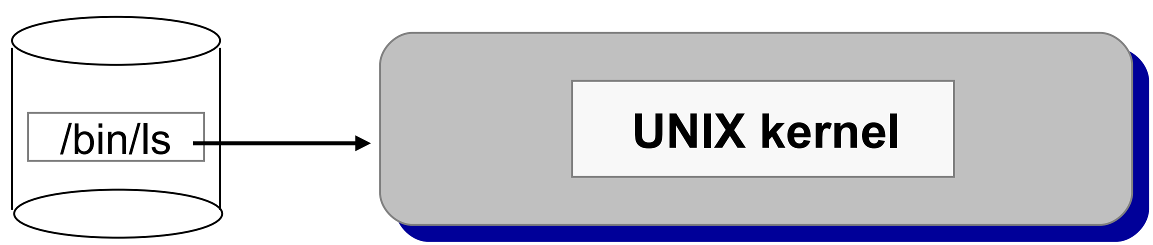
```
char * argv[ ] = {“/bin/ls”, 0};
<…>
int cpid = fork( );
if (cpid = = 0) {
    execv(argv[0], argv);
    exit(0);
}
<parent code>
wait(&cpid);
```



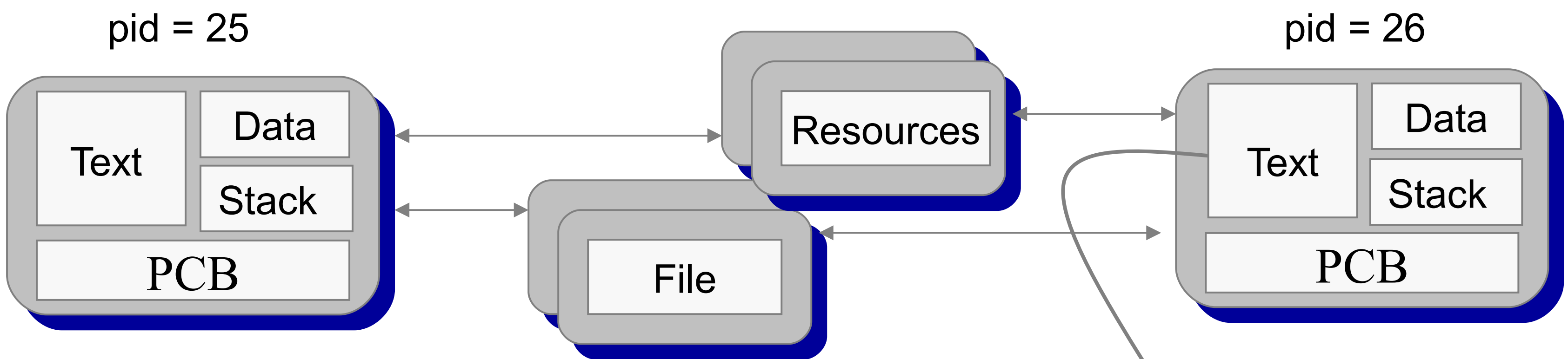



```
char * argv[ ] = {"/bin/ls", 0};
<…>
int cpid = fork( );
if (cpid = = 0) {
    execv(argv[0], argv);
    exit(0);
}
<parent code>
wait(&cpid);
```

Exec destruye la imagen del proceso llamador. Una nueva imagen es construida a partir del ejecutable ls

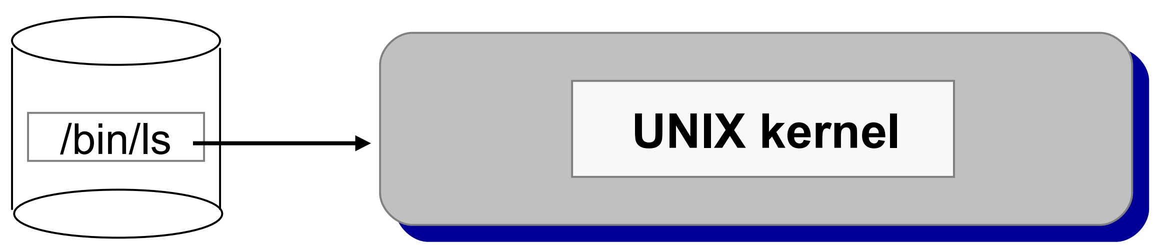


Prof. Angela Di Serio




```
char * argv[ ] = {"/bin/ls", 0};
<…>
int cpid = fork( );
if (cpid = = 0) {
    execv(argv[0], argv);
    exit(0);
}
<parent code>
wait(&cpid);
```

```
<first line of ls>
<…>
<…>
<…>
exit(0);
```



Prof. Angela Di Serio

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

char * argv[] = {"/bin/ls", "-l", 0};
int main()
{
   int pid, status;

   if ( (pid = fork() ) < 0 )
   {
        printf("Fork error \n");
        exit(1);
   }
   if(pid == 0)  { /* Child executes here */
        execv(argv[0], argv);
        printf("Exec error \n");
        exit(1);
   } else      /* Parent executes here */
        wait(&status);
   printf("Hello there! \n");
   return 0;
}
```

Execl similar a execv pero los argumentos para el nuevo programa se pasan como una lista y no un vector

```
execl("/bin/ls", "/bin/ls", "-l", 0);
```

es equivalente a

```
char * argv[] = {"/bin/ls", "-l", 0};
execv(argv[0], argv);
```

execl es usado principalmente cuando se conoce el número de argumentos a ser pasados

Prof. Angela Di Serio

```
int childPid;
char * const argv[ ] = {…};

main {
    childPid = fork();
    if(childPid == 0)
    {
        // I am child ...
        // Do some cleaning, close files
        execv(argv[0], argv);
    }
    else
    {
        // I am parent ...
        <code for parent process>
      wait(0);
    }
}
```

Prof. Angela Di Serio

```
EXEC(3)                    Linux Programmer's Manual                    EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *path, const char *arg, ...);
       int execlp(const char *file, const char *arg, ...);
       int execle(const char *path, const char *arg,
                  ..., char * const envp[]);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                  char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE
```

Prof. Angela Di Serio

BY NC SA

```
DESCRIPTION
       The  exec()  family of functions replaces the current process image with a new process image.
       The functions described in this manual page are front-ends for execve(2).   (See  the  manual
       page for execve(2) for further details about the replacement of the current process image.)

       The initial argument for these functions is the name of a file that is to be executed.

       The  const char *arg and subsequent ellipses in the execl(), execlp(), and execle() functions
       can be thought of as arg0, arg1, ..., argn.  Together they describe a list  of  one  or  more
       pointers  to  null-terminated  strings that represent the argument list available to the exe-
       cuted program.  The first argument, by convention, should point to  the  filename  associated
       with  the  file  being executed.  The list of arguments must be terminated by a NULL pointer,
       and, since these are variadic functions, this pointer must be cast (char *) NULL.

       The execv(), execvp(), and execvpe() functions provide an array of  pointers  to  null-termi-
       nated strings that represent the argument list available to the new program.  The first argu-
       ment, by convention, should point to the filename associated with the  file  being  executed.
       The array of pointers must be terminated by a NULL pointer.

       The  execle() and execvpe() functions allow the caller to specify the environment of the exe-
       cuted program via the argument envp.  The envp argument is an array of pointers to  null-ter-
       minated strings and must be terminated by a NULL pointer.  The other functions take the envi-
       ronment for the new process image from the external variable environ in the calling process.

   Special semantics for execlp() and execvp()
       The execlp(), execvp(), and execvpe() functions duplicate the actions of the shell in search-
       ing  for an executable file if the specified filename does not contain a slash (/) character.
       The file is sought in the colon-separated list of directory pathnames specified in  the  PATH
       environment  variable.  If this variable isn't defined, the path list defaults to the current
```

Prof. Angela Di Serio

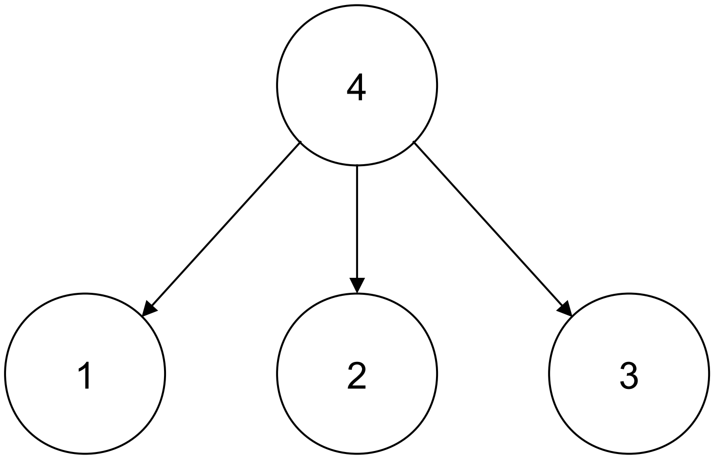
cc BY NC SA

```
int main (int argc, char *argv[])
{
pid_t childpid = 0;
int i, nbrOfProcesses;

if (argc != 2)
   {   /* Check for valid number of command-line arguments */
   fprintf(stderr, "Usage: %s <processes>\n", argv[0]);
   return 1;
   } // End if
nbrOfProcesses = atoi(argv[1]);  // Convert string to an integer
for (i = 1; i < nbrOfProcesses; i++)
   {
   childpid = fork();
   if (childpid == -1)
      {
      perror("Fork failed");
      exit(1);
      }  // End if
   else if (childpid == 0)  // The child
      {
      printf("i:%d  process ID: %4ld  parent ID: %4ld  child ID: %4ld\n",
             i, getpid(), getppid(), childpid);
      sleep(2); // Sleep two seconds
      exit(0);
      } // End if
   else // The parent
      continue;
   } // End for
printf("i:%d  process ID: %4ld  parent ID: %4ld  child ID: %4ld\n",
             i, getpid(), getppid(), childpid);
return 0;
} // End main
```

Prof. Angela Di Serio

BY NC SA

```
% a.out 4
i:1  process ID: 2736  parent ID:  120  child ID:    0
i:2  process ID: 3488  parent ID:  120  child ID:    0
i:4  process ID:  120  parent ID:   40  child ID:  512
i:3  process ID:  512  parent ID:  120  child ID:    0
```

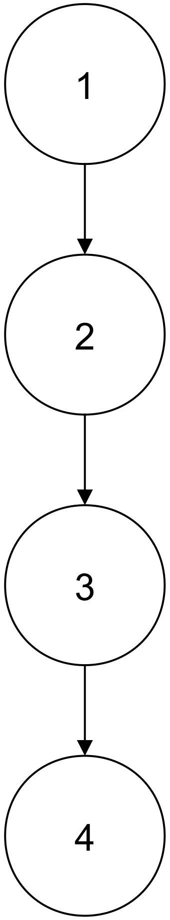


CC BY NC SA

```
int main (int argc, char *argv[])
{
pid_t childpid = 0;
int i, nbrOfProcesses;

if (argc != 2)
   {   /* Check for valid number of command-line arguments */
   fprintf(stderr, "Usage: %s <processes>\n", argv[0]);
   return 1;
   } // End if

nbrOfProcesses = atoi(argv[1]);  // Convert character string to integer
for (i = 1; i < nbrOfProcesses; i++)
   {
   childpid = fork();
   if (childpid == -1)
      {
      perror("Fork failed");
      exit(1);
      } // End if
   else if (childpid != 0)  // True for a parent
      break;
   } // End for

// Each parent prints this line
fprintf(stderr, "i: %d  process ID: %4ld  parent ID: %4ld  child ID: %4ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);
sleep(5); // Sleep five seconds
return 0;
} // End main
```


BY NC SA

```
% a.out 4

i: 1  process ID:  496  parent ID:   40  child ID: 3232
i: 2  process ID: 3232  parent ID:  496  child ID:  320
i: 3  process ID:  320  parent ID: 3232  child ID: 2744
i: 4  process ID: 2744  parent ID:  320  child ID:    0
```



Prof. Angela Di Serio