**Universidad de Antioquia**

**Facultad de Ingeniería**

**Sistemas Operativos y Laboratorio**

Laboratorio 3-1: Procesos

# 1. Llamados al sistema

Los procesos de usuario se ejecutan en el modo menos privilegiado de la máquina (modo usuario) donde sólo pueden ejecutar algunas operaciones básicas. Muchas de las operaciones privilegiadas relacionadas principalmente con los servicios del sistema operativo tienen que ser solicitadas al kernel mediante los **llamados al sistema**.

Los llamados al sistema son la interfaz por la cual los procesos acceden a las funciones del Sistema Operativo. Los llamados al sistema se encuentran implementados en el kernel y se ejecutan en **modo privilegiado**. En la escritura de programas, un llamado al sistema **se invoca como una función**, sin embargo, a diferencia de una función ordinaria, cuando un programa llama a una función del sistema, los argumentos son empaquetados y manejados por el kernel, el cual toma el control de la ejecución hasta que el llamado se completa.

Un llamado al sistema no es una llamado a una función ordinaria, y requiere un procedimiento especial para **transferir el control al kernel**. Básicamente el compilador genera una instrucción especial de máquina que, al ejecutarse, produce una **interrupción de software** (“trap” o excepción sincronica) para que se realice un cambio de modo en el procesador (de “usuario” a “privilegiado”) y el kernel del sistema operativo realice las acciones restringidas necesarias para prestar el servicio solicitado.

En el caso de los sistemas Unix/Linux, los llamados al sistema se encuentran inmersos en funciones pertenecientes a diferentes librerías. Por ejemplo, la función “time”, de la librería “time.h”, contiene el llamado para consultar el reloj del sistema; y la función “close”, de la librería “unistd.h”, contiene el llamado para cerrar un archivo.

En [este enlace](https://wiki.deimos.fr/images/3/3c/LINUX_System_Call_Quick_Reference.pdf) se encuentra una tabla de referencias con los principales llamados al sistema del sistema operativo Linux, esta tabla es de gran utilidad para el desarrollo de todas las prácticas siguientes, así que se recomienda su estudio.

## 1.1. Ejemplo de llamado al sistema

Para utilizar los llamados al sistema se puede hacer uso de dos diferentes métodos: la función “syscall” o la función de la librería correspondiente.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 | #include <syscall.h> #include <unistd.h> #include <stdio.h> #include <sys/types.h>  int main(void) {  long ID1, ID2;  */\*--------------------------------\*/*  */\* DIRECT SYSTEM CALL \*/*  */\* SYS\_getpid(func no. is 20) \*/*  */\*--------------------------------\*/*  ID1 = syscall(SYS\_getpid);  printf("syscall(SYS\_getpid) = %ld\n", ID1);   */\*-----------------------------------\*/*  */\* "libc" WRAPPED SYSTEM CALL \*/*  */\* SYS\_getpid(func no. is 20) \*/*  */\*-----------------------------------\*/*  ID2 = getpid();  printf("getpid() = %ld\n", ID2);  return 0; } |
| **Código 1.** Ejemplo de llamados al sistema | |

El Código 1 muestra un ejemplo de la invocación del llamado del sistema “*getpid*”. En el primer caso se empleó el llamado indirecta con la función “syscall”[[1]](#footnote-1) (Línea 12), mientras que en el segundo caso se hizo el llamado al sistema con la función de la librería estándar “getpid”[[2]](#footnote-2) (Línea 19).

# 2. Procesos

Un proceso se puede definir como una **instancia en ejecución** de un programa, igualmente, concebimos un proceso como el mecanismo del sistema operativo que **abstrae** el uso del recurso de procesamiento. En general los procesos son componentes fundamentales de un sistema operativo, por ende es necesario conocer las herramientas que el sistema operativo nos ofrece para gestionar su funcionamiento.

El sistema operativo lleva el control de los procesos a través de la **tabla de procesos**, donde se almacena una estructura de datos llamada el **bloque control de procesos** (PCB , por sus siglas en inglés: *Process Control Block*). En el PCB se encuentra información relevante del proceso como su identificador, su proceso padre, su estado, el valor del registro *program counter* (PC), el valor del registro *stack pointer* (SP), las referencias a su imagen de memoria, y, en general, todo aquello que se debe conocer del proceso para que pueda iniciarse nuevamente desde el punto donde este se detuvo para ceder la CPU a otro proceso (ver Figura 1).

|  |
| --- |
| Process State |
| Process Number |
| Program Counter |
| Registers |
| Memory Limits |
| List of Open files |
| . . . |
| Figura 1. Proceso |

En los sistemas Linux, la estructura “task\_struct” es la que contiene esta información. La declaración de la estructura puede verse en el archivo “sched.h” del código del kernel en su versión 3.0.4.

## 2.1. Servicios POSIX para la gestión de procesos

A continuación se realizará una revisión de los principales servicios que ofrece la especificación **POSIX**[[3]](#footnote-3) para la administración de procesos.

* **Identificación de procesos:**

Se identifica a cada proceso por medio de un número único con representación entera. Este es conocido como el identificador del proceso (que es de tipo “pid\_t”). La siguiente tabla muestra las principales funciones empleadas para este fin:

|  |  |
| --- | --- |
| **Función** | **Descripción** |
| pid\_t getpid(void) | Este servicio devuelve el identificador del proceso que realiza la llamada. |
| pid\_t getppid(void) | Devuelve el identificador del proceso padre |
| uid\_t getuid(void) | Devuelve el identificador del usuario real (usuario que ejecuta el proceso). |
| uid\_t geteuid(void) | Devuelve el identificador del usuario efectivo (usuario en el que se ejecuta el proceso). |
| gid\_t getgid(void) | Devuelve el identificador del grupo real. |
| gid\_t getegid(void) | Devuelve el identificador del grupo efectivo. |
| **Tabla 1.** Servicios POSIX para identificar procesos | |

* **Gestión de procesos**

Este grupo de funciones permite la creacion y manipulacion del estado de los procesos. La siguiente tabla describe brevemente estas funciones:

|  |  |
| --- | --- |
| **Función** | **Descripción** |
| pid\_t fork(void) | Permite crear procesos: Se realiza una clonación del proceso que lo solicita (conocido como proceso padre). El nuevo proceso se conoce como proceso hijo. |
| int execl (char \*path, char \*arg, …) | Familia de funciones que permiten cambiar el programa que está ejecutando el proceso. |
| void exit (int status) | Termina la ejecución de un proceso y envía el valor de “status” al padre. Es similar a “return” de la función “main”. |
| pid\_t wait (int \*status)  pid\_t waitpid(pid\_t pid, int \*status, int options) | Permite que un proceso padre espere hasta que finalice la ejecución de un proceso hijo. El proceso padre se queda bloqueado hasta que termina el proceso hijo. Ambas llamadas permiten obtener información sobre el estado de terminación. |
| int sleep (unsigned int seconds) | Suspende el proceso durante un número de segundos. El proceso despierta cuando ha transcurrido el tiempo o cuando el proceso recibe una señal. |
| **Tabla 2.** Servicios POSIX para gestionar procesos | |

## 2.2. Ejemplos de identificacion y gestion de procesos

### 2.2.1. Creación de procesos

#### Usando fork

En la creación de procesos hay dos elementos claves:

* El proceso padre
* El proceso hijo

Crear un proceso consiste en realizar una copia en memoria de un proceso (*el padre*) por medio de una llamada al sistema. Dicha copia (*el hijo*) posee una copia de toda la información del padre excepto que posee su propio **PID** (Identificador del proceso) y **PPID** (Identificador del proceso padre).

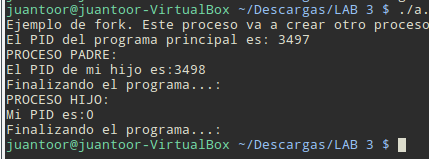
|  |  |
| --- | --- |
| **Función** | Fork |
| **Uso** | #include <unistd.h> pid\_t fork(void); |
| **Descripción** | La función ”fork()” crea un nuevo proceso hijo como duplicado del proceso que la invoca (*padre*). Cuando la ejecución de la función es exitosa se retorna un valor diferente para el *padre* y para el *hijo*:   * Al padre se le retorna el PID del hijo. * Al hijo se le retorna 0.   El valor retornado será -1 en caso de que la llamada falle y no se pueda crear un proceso hijo. |
| **Ejecución** | fork(); |
| **Resultado** | Pp: Proceso padre con PID 0  Ph: Proceso creado (hijo) con PID 1  Pp  (0)  Pp  (0)  Ph  (1)  fork() |
| **Tabla 4.** Función “fork” | |

**Ejemplo:** Compile y ejecute el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | #include <unistd.h> #include <stdio.h>  int main(int argc, char \*argv[]) {  pid\_t valor\_retornado;  printf("Ejemplo de fork. Este proceso va a crear otro proceso\n");  printf("El PID del programa principal es: %d\n", (int)getpid());  switch(valor\_retornado = fork()) {  case -1: *// Caso de error*  printf("Error al crear el proceso");  return -1;  case 0: *// Codigo ejecutado por el hijo*  printf("PROCESO HIJO:\n");  printf("Mi PID es:%d\n", (int)valor\_retornado);  break;  default: *// Codigo ejecutado por el padre*  printf("PROCESO PADRE:\n");  printf("El PID de mi hijo es:%d\n", (int)valor\_retornado);  }  *// Código ejecutado tanto por el padre como el hijo*  printf("Finalizando el programa...:\n");  return 0; } |
| **Código 3.** Uso de “fork” | |

**Responda:**

* ¿Cuál es la salida del programa anterior y por qué?



#### Múltiples hijos

Un proceso puede crear o tener múltiples hijos llamando repetidamente la función fork. Y estos hijos pueden tener o crear otros procesos (*nietos*). La recomendación es tener un proceso padre que cree a todos los hijos que se necesiten. A continuación se muestran algunas formas de trabajar con varios procesos:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | #include <stdio.h> #include <stdlib.h> #include <unistd.h> int main () {  pid\_t pid\_hijo1;  pid\_t pid\_hijo2;  pid\_t pid\_hijo3;  pid\_hijo1 = fork(); *// Creo el primer hijo*  if (pid\_hijo1 == 0) { *// Hijo 1*  printf("Soy el hijo 1");  sleep (5);  } else { *// Padre*  pid\_hijo2 = fork(); *// Creo al segundo hijo*  if (pid\_hijo2 == 0) { *// Hijo 2*  printf("Soy el hijo 2");  sleep (5);  } else { *// Padre*  pid\_hijo3 = fork(); *// Creo al tercer hijo*  if (pid\_hijo3 == 0) { *// Hijo 3*  printf("Soy el hijo 3");  sleep (5);  } else { *// Padre*  printf("Soy el padre");  sleep (5);  }  }  }  return 0; } |
| **Código 4.** Múltiples llamados a “fork” | |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 | #include <stdio.h> #include <stdlib.h> #include <unistd.h>  int main () {  int i;  int numHijos = 3;  pid\_t pid;  for (i = 0; i < numHijos; i++) {  pid = fork();  if (pid == -1) {  */\* Error \*/*  printf("No fue posible crear un hijo\n");  return -1;  }  if (pid == 0) {  printf("Soy el hijo #%d con PID: %d\n",i+1, getpid());  exit(0);  }  }  return 0; } |
| **Código 5.** Llamando a “fork” en un ciclo. | |

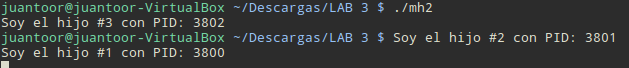
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | #include <stdio.h> #include <stdlib.h> #include <unistd.h>  int main () {  int pid;  int numHijos = 5;  int numProceso;  for(numProceso = 0; numProceso < numHijos; numProceso++) {  pid = fork();  if (pid == -1) {  *// Imprimir algún mensaje de error*  } else if(pid == 0) {  break;  }  }  if (pid == 0) {  *// Lógica del hijo*  printf("Soy el hijo #%d\n", numProceso);  }  else {  printf("Soy un padre perezoso\n");  } } |
| **Código 6.** Llamando a “fork” en un ciclo (otra forma, no muy usada). | |

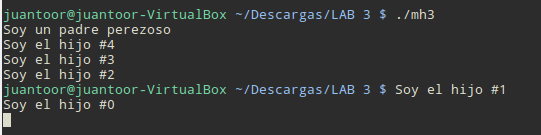
|  |  |
| --- | --- |
|  | En la siguiente jerarquía de procesos, cada uno de los procesos hijos debe aumentar el valor de una variable e imprimir ese valor en pantalla:  fork  fork  fork  fork  Vista simplificada  Note que en la figura anterior el proceso padre (0) creó 3 procesos hijos (1, 3, 4) y un proceso “nieto” (2). A continuación se muestra el código asociado el problema anterior. |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 | #include <unistd.h> #include <stdio.h>  int main(int argc, char \*argv[]) {  pid\_t pid\_h1, pid\_h2, pid\_h3;  pid\_t pid\_n;  int i = 0;   pid\_h1 = fork();  if(pid\_h1 == 0) {  i++;  pid\_n = fork();  if(pid\_n==0) {  i++;  printf("NIETO: i = %d\n",i);   }else {  printf("HIJO 1: i = %d\n",i);   }   }else {  pid\_h2 = fork();  if(pid\_h2 == 0) {  i++;  printf("HIJO 2: i = %d\n",i);   }else {  pid\_h3 = fork();  if(pid\_h3 == 0) {   i++;  printf("HIJO 3: i = %d\n",i);   }else {   printf("PAPA: i = %d\n",i);   }  }  }  return 0; } |
| **Código 7.** Hijos y Nietos | |

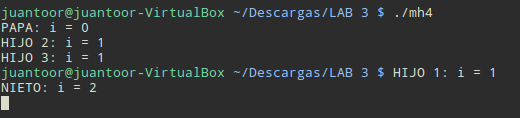
**Responda:**

* ¿Qué diferencias nota entre los tres ejemplos anteriores?
* ¿Cuál es la salida de estos ejemplos y por qué?









Algunas de las “anomalías” presentes en la ejecución de los ejemplos anteriores serán explicados en la siguiente sección.

### 2.2.2. Gestión de procesos

#### Terminación de procesos con exit

Así cómo es posible crear procesos también es posible finalizarlos. Para ello existen dos maneras: por medio del uso de la función “exit” o usando la función “kill”. Veamos primero la función “exit”:

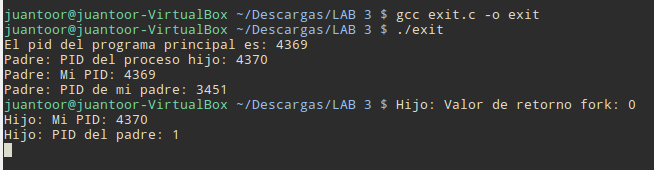
|  |  |
| --- | --- |
| **Función** | exit |
| **Uso** | #include <stdlib.h> void exit(int status); |
| **Descripción** | Esta función causa la terminación normal de un proceso. La variable entera status es empleada para transmitir al proceso padre la forma en que el proceso hijo ha terminado. Por convención este valor suele ser 0 si el programa termina de manera exitosa u otro valor cuando la terminación de este es anormal. |
| **Ejecución** | exit(8); |
| **Tabla 5.** Función “exit” | |

Compile y ejecute el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 | #include <unistd.h> #include <stdio.h> #include <stdlib.h>  int main(int argc, char \*argv[]) {  pid\_t pid\_hijo;  printf("El pid del programa principal es: %d\n",(int)getpid());   switch(pid\_hijo=fork()) {   case -1: */\* Código ejecutado en caso de error\*/*  printf("Error al crear el proceso");  return -1;  case 0: */\* Código ejecutado por el hijo \*/*  printf("Hijo: Valor de retorno fork: %d\n",(int)pid\_hijo);  printf("Hijo: Mi PID: %d\n",(int)getpid());  printf("Hijo: PID del padre: %d\n",(int)getppid());  exit(0);  printf("Esta instrucción nunca se ejecutara en el proceso hijo\n");  break;  default: */\* Código ejecutado por el padre \*/*  printf("Padre: PID del proceso hijo: %d\n",(int)pid\_hijo);  printf("Padre: Mi PID: %d\n",(int)getpid());  printf("Padre: PID de mi padre: %d\n",(int)getppid());  }  return 0; } |
| **Código 8.** Uso de “exit” | |

Al ejecutar el código anterior, se espera que salga algo similar a la captura mostrada en la siguiente figura:

|  |
| --- |
| pid |
| **Figura 2.** Efecto de llamada a “exit” |



Observe bien la figura anterior y notará que una vez invocada la función “exit” el proceso hijo deja de ejecutarse, por eso las instrucciones ubicadas después de esta llamada nunca se ejecutarán.

#### Esperar que un hijo termine:

Si observa la Figura 2 notará algo interesante; el proceso padre culmina antes de que el hijo lo haga. Pues bien, existen ocasiones en las cuales es deseable que el proceso padre espere a que el proceso hijo culmine y es allí donde entra en juego la función “**wait**”. Básicamente lo que hace esta función es permitir esperar que la ejecución de un proceso hijo finalice y permitir al padre esperar recuperar información sobre la finalización del hijo. La siguiente tabla resume esta función:

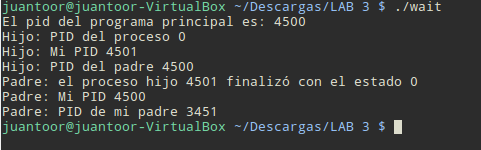
|  |  |
| --- | --- |
| **Función** | wait |
| **Uso** | #include <sys/types.h> #include <sys/wait.h> pid\_t wait(int \*status);  **Dónde:**   * **Valor retornado:** Entero que contiene el PID del proceso hijo que finalizó o -1 si no se crearon hijos o si ya no hay hijos por los cuales esperar. * **Status:** Puntero a la dirección donde la llamada al sistema debe almacenar el estado de finalización, o valor de retorno del proceso hijo (parámetro utilizado en la llamada exit). |
| **Descripción** | Esta función suspende la ejecución del proceso padre hasta que su hijo termine. |
| **Tabla 6.** Función “wait” | |

Compile y ejecute el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <sys/wait.h>  int main(int argc, char \*argv[]) {  pid\_t pid\_hijo;  int estado;  printf("El pid del programa principal es: %d\n",(int)getpid());  switch(pid\_hijo=fork()) {  case -1: */\* Codigo ejecutado en caso de error\*/*  printf("Error al crear el proceso");  return -1;  case 0: */\* Codigo ejecutado por el hijo \*/*  printf("Hijo: PID del proceso %d\n",(int)pid\_hijo);  printf("Hijo: Mi PID %d\n",(int)getpid());  printf("Hijo: PID del padre %d\n",(int)getppid());  exit(0);  printf("Esto nunca se ejecutará en el proceso hijo\n");  break;  default: */\* Código ejecutado por el padre \*/*  wait(&estado);  printf("Padre: el proceso hijo %d finalizó con el estado %d \n",(int)pid\_hijo, estado);  printf("Padre: Mi PID %d\n",(int)getpid());  printf("Padre: PID de mi padre %d\n",(int)getppid());  }  return 0; } |
| **Código 9.** Uso de “wait” | |

Al ejecutar el código anterior la salida esperada es algo como la siguiente:

|  |
| --- |
| pid3.png |
| **Figura 3.** Efecto de llamada a “wait” |



Si se compara la Figura 2 con la Figura 3 podrá notar que ya hay algo diferente y es que una vez invocada la función “wait”, el proceso padre no continua la ejecución de las instrucciones siguientes hasta que el proceso hijo culmine su ejecución.

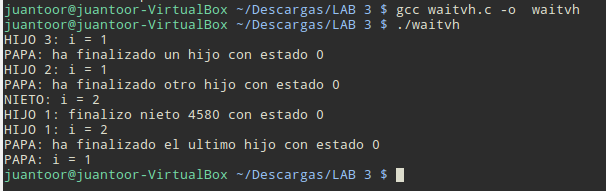
Ahora bien, si hay varios procesos hijos, el proceso padre queda bloqueado hasta que uno de ellos culmina. Al finalizar uno de ellos, se liberan todos los recursos que tengan asociados, recuperándose el valor de retorno devuelto para que pueda ser accesible desde el proceso que realizó la llamada. El siguiente código clarifica un poco esto:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  51  52  53  54  55  56  57  58  59 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <wait.h>  int main(int argc, char \*argv[]) {  pid\_t pid\_h1, pid\_h2, pid\_h3;  int status\_h1, status\_h2, status\_h3;  pid\_t pid\_n;  int status\_n;  int i = 0;   pid\_h1 = fork();  if(pid\_h1 == 0) {  i++;  pid\_n = fork();  if( pid\_n == 0 ) {  i++;  printf("NIETO: i = %d\n",i);   }  else {  wait(&status\_n); *// Papa (hijo 1) esperando hijo (nieto)*  i++;  printf("HIJO 1: finalizo nieto %u con estado %d\n", pid\_n, status\_n );  printf("HIJO 1: i = %d\n",i);   }   }  else {  pid\_h2 = fork();  if(pid\_h2 == 0) {  i++;  printf("HIJO 2: i = %d\n",i);   }  else {  pid\_h3 = fork();  if(pid\_h3 == 0) {   i++;  printf("HIJO 3: i = %d\n",i);   }  else {   *// El papa decidió esperar todos los hijos al final*  wait(&status\_h1); *// Papa esperando un hijo*  printf("PAPA: ha finalizado un hijo con estado %d\n", status\_h1);  wait(&status\_h2); *// Papa esperando otro hijo*  printf("PAPA: ha finalizado otro hijo con estado %d\n", status\_h2);  wait(&status\_h3); *// Papa esperando el ultimo hijo*  printf("PAPA: ha finalizado el ultimo hijo con estado %d\n", status\_h3);  i++;  printf("PAPA: i = %d\n",i);   }  }  }    return 0; } |
| **Código 10.** Uso de “wait” con varios hijos | |

Como se puede ver en el código anterior, hay una invocación a “wait” por cada uno de los hijos esperados.

**Responda:**

* ¿Qué diferencias nota entre el Código 8 y el Código 10?
* ¿Cuál es la salida de estos ejemplos y por qué?



#### Terminación de procesos con kill

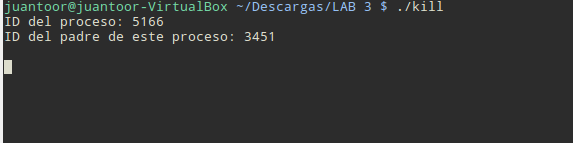
Es posible terminar abruptamente con la vida de un proceso, para ello se emplea la función “kill”, la cual a diferencia de la función “exit” termina de manera forzada dicho proceso.

“kill” también es un comando en consola, el cual se emplea pasando como argumento el PID del proceso que se desea culminar. Este comando trabaja enviando una señal de terminación (*SIGTERM*) la cual causa que el proceso culmine a menos que el programa tenga un *handler* para gestionar esta señal o que *SIGTERM* (la señal) se encuentra enmascarada. En lo que respecta a la función la siguiente tabla resume sus mayores atributos:

|  |  |
| --- | --- |
| **Función** | Kill |
| **Uso** | #include <sys/types.h> #include <signal.h> int kill(pid\_t pid, int sig); |
| **Descripción** | Esta función es empleada para enviar una señal a cualquier proceso o grupo de procesos.  El argumento pid es el PID del proceso que se desea killear mientras que el argumento sig está asociado a la señal que se desea enviar (KILL, TERM, TRAP, ALRM, SIGINT). |
| **Tabla 7.** Función “kill” | |

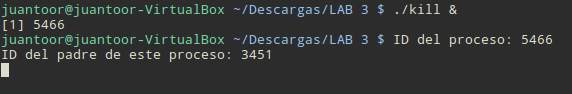
Compile y ejecute el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <signal.h>  int main(int argc, char \*argv[]) {  printf("ID del proceso: %d\n", (int)getpid());  printf("ID del padre de este proceso: %d\n", (int)getppid());  for(;;) {  pause();  }  return 0; } |
| **Código 11.** Programa con ciclo infinito | |



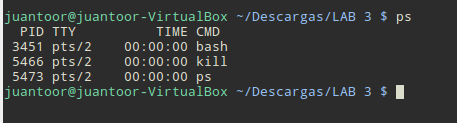
Luego compile el código y ejecútalo en **background**[[4]](#footnote-4) (esto para que no se bloquee la consola hasta que el programa termine y pueda ejecutar otros comandos), a continuación se muestra como:

|  |
| --- |
| pid_for.png |
| **Figura 4.** Salida en pantalla |



Si se invoca el comando “ps”, se puede ver la información más relevante de los procesos que actualmente se están ejecutando tal y como se muestra la siguiente figura:

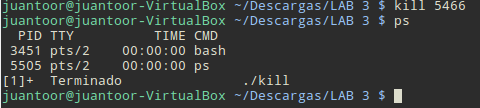
|  |
| --- |
| ps.png |
| **Figura 5.** Ejecución del comando “ps” |



Como se puede notar de la figura anterior, se despliegan 3 procesos la consola (**bash**), el proceso del comando **ps** y el proceso del programa que acaba de ser compilado y que ahora se ejecuta (**a.out**). Como el programa tiene un ciclo infinito vamos a enviarle una señal de terminación con el comando kill tal y como se muestra a continuación:

|  |
| --- |
| kill.png |
| **Figura 6.** Ejecución del comando kill |

Como se puede notar en la figura anterior, el programa es culminado (si vuelve a ejecutar el comando ps notará que ya no aparece este proceso). Note una cosa importante, el comando kill[[5]](#footnote-5) se invocó pasando el **pid** del proceso a matar (32067) y el número[[6]](#footnote-6) de la señal a enviar (9). Una forma alternativa de invocar este comando sería: **kill -KILL 32067**, donde **KILL** es una constante del sistema que es igual al mismo número 9.



Anteriormente se vio el uso de “kill” como comando, ahora veamos cómo es su empleo como función, para ello codifique y compile el siguiente código fuente:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <signal.h> #include <sys/types.h>  int main(int argc, char \*argv[]) {  pid\_t my\_pid;  my\_pid = getpid();  printf("Mi ID del proceso es: %d \n",(int)my\_pid);  printf("El ID del padre del proceso es: %d \n",(int)getppid());  printf("Hola mundo. \n");  printf("Hola mundo. \n");  printf("Hola mundo. \n");  printf("Hola mundo. \n");  printf("Hasta la vista baby. \n");  kill(my\_pid,9);*//Forma alternativa:kill(pid\_hijo,SIGKILL);*   printf("Hasta la vista baby. \n");  printf("Hasta la vista baby. \n");  printf("Hasta la vista baby. \n");  return 0; } |
| **Código 12.** Uso de “kill” | |

### 2.2.3. Procesos Zombies y Procesos Huérfanos

El sistema operativo Linux define dos tipos de procesos que se pueden dar dentro de este sistema operativo: el proceso con estado zombie o el proceso “huérfano”.

El proceso zombie se presenta cuando se requiere que las estructuras de datos del sistema operativo asociadas a un proceso continúen existiendo a pesar de que el proceso haya finalizado. Un proceso huérfano se refiere a un proceso que, debido a la terminación de su proceso padre, requiere ser reasignado a la jerarquía de procesos del sistema operativo.

Después de que un proceso hijo es creado por su padre haciendo uso de la función “fork” pueden suceder una de las siguientes cosas:

* Que el proceso padre espere a que el proceso hijo culmine haciendo uso de la función “wait”. En el caso normal, cuando el proceso hijo termina, se le notifica su terminación al padre y se le manda el valor a la variable *status*. Ahora bien, también podría suceder que el proceso padre se queda a la espera de que el hijo acabe y que éste, en efecto, ya ha culminado. Más exactamente, que **el proceso hijo finalice antes de que el proceso padre llame la función “wait”**. Cuando esto sucede, si las estructuras de datos del proceso hijo son liberadas, el proceso padre no podría recoger el estado de salida de su hijo. Para evitar este caso el sistema operativo utiliza el estado **zombie** para mantener estos datos disponibles sin importar que el proceso ya haya finalizado.
* Que el proceso padre no espere a que su hijo culmine, de tal manera que si **el proceso padre culmina primero** el proceso hijo será un **proceso huérfano**. El sistema operativo realiza un proceso de reparentalización (*reparenting*), en el cual un nuevo proceso padre “adopta” el proceso hijo (generalmente lo hace el proceso **init** o el proceso de sesión **upstart**).

Compile y ejecute el siguiente código el cual crea un proceso zombie. Elija como nombre del ejecutable *make-zombie*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 | #include <stdlib.h> #include <sys/types.h> #include <unistd.h> int main () {  pid\_t child\_pid;  */\* Creacion del proceso hijo. \*/*  child\_pid = fork ();  if (child\_pid > 0) {  */\*Este es el proceso padre el cual duerme por 20 segundos.\*/*  sleep (20);  }  else {  */\* Este es el proceso hijo el cual culmina inmediatamente.\*/*  exit (0);  } return 0; } |
| **Código 13.** make-zombie.c | |

Ejecute el programa anterior y una vez hecho esto, ejecuta en otra pestaña el comando ps –o pid,ppid,stat,cmd*.* Notará una salida algo similar como la de la siguiente figura:

|  |
| --- |
| make-zombie.png |
| **Figura 7.** Salida en pantalla de ps |

Observe además del proceso *make-zombie* la existencia de otro proceso (el hijo zombie: [make-zombie] <defunct>) cuyo código de estado es Z indicando que es zombie.

**Ejercicio:** En el Código 13 intercambie las instrucciones que ejecuta el padre y el hijo. Es decir, haga que el padre finalice inmediatamente ejecutando la instrucción exit y que el hijo espere durante 20 segundos ejecutando la instrucción sleep. ¿Qué nota ahora de diferente? ¿Cuál es el valor del campo **ppid** para el proceso hijo?

## 2.3. Ejecución de nuevos programas

### 2.3.1. Familia de funciones exec

Con anterioridad se trató la función **fork** la cual permitía la creación de un nuevo proceso el cual era una copia del proceso padre, la limitante al respecto era que al ser el nuevo proceso una copia del padre, lo que en realidad se estaba ejecutando era otra instancia de un mismo programa, esto impone una limitante la cual se traduce en la siguiente pregunta: ¿Es posible realizar la ejecución de nuevos programas?.

Pues bien, afortunadamente existe una nueva función con la cual esta limitante puede ser superada, la función **exec**. Esta función reemplaza el programa que se está ejecutando en un proceso por otro programa. Cuando un programa llama una función **exec**, el proceso inmediatamente cesa de ejecutar el programa y empieza ejecutando un nuevo programa desde el principio (asumiendo que la llamada **exec** no encontró un error). Dentro de la familia de funciones exec, hay funciones que varían levemente en sus capacidades y como son invocadas, la siguiente tabla trata esto con más detalle:

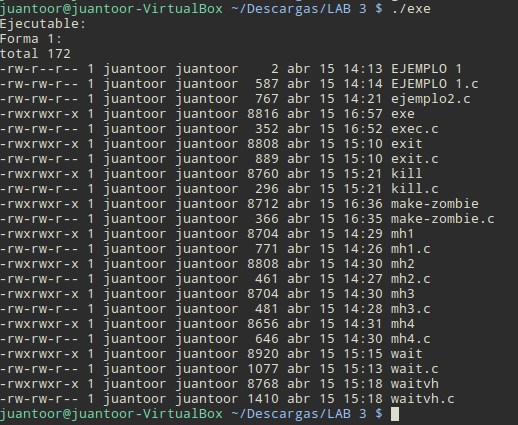
|  |  |
| --- | --- |
| **Función** | exec |
| **Uso** | #include <unistd.h>  int execl(const char \*path, const char \*arg,...); int execlp(const char \*path, const char \*arg,...); int execle(const char \*path, const char \*arg,  ...,char \*const envp[]); int execv(const char \*path, char \*const argv[]); int execvp(const char \*file, char \*const argv[]);  **Donde**   * **path** o **file**: Cadena de caracteres que contiene el nombre del nuevo programa a ejecutar con su ubicación, **/bin/cp** por ejemplo. * **const char \*arg**: Lista de uno o más apuntadores a cadenas de caracteres que representan la lista de argumentos que recibirá el programa llamado. Por convención, el primer argumento deberá contener el nombre del archivo que contiene el programa ejecutado. El último elemento de la lista debe ser un apuntador a NULL. * **char \*const arg[]**: Array de punteros a cadenas de caracteres que representan la lista de argumentos que recibirá el programa llamado. Por convención, el primer argumento (arg0) deberá tener el nombre del archivo que contiene el programa ejecutado y el último elemento deberá ser un apuntador a NULL. * **char \*const envp[]**: Array de apuntadores a cadenas que contienen el entorno de ejecución (variables de entorno) que tendrá accesible el nuevo proceso. El último elemento deberá ser un apuntador a NULL. |
| **Descripción** | Esta familia de funciones, reemplaza la imagen de memoria actual del proceso con una nueva imagen de memoria. En caso de que la llamada a la función sea correcta esta no retorna nada, si hay una falla el valor retornado será -1. |
| **Tabla 8.** Familia de funciones “exec” | |

Compile y ejecute el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 | #include <unistd.h> #include <stdio.h> int main(int argc, char \*argv[]) {  printf("Ejecutable: \n");  char \*args[] = {"/bin/ls", "-l", ".", NULL};  printf("Forma 1: \n");  execl("/bin/ls", "/bin/ls", "-l", ".", NULL);  printf("Forma 2: \n");  execv("/bin/ls", args);  printf("Forma 3: \n");  execvp("/bin/ls", args);  return 0; } |
| **Código 14.** Uso de “exec” | |

**Responda:**

* ¿Qué hace el programa anterior?
* ¿Qué tiene de raro la salida?



* Tome el código anterior y pártelo en 3 programas donde cada uno de estos debe colocar cada una de las diferentes invocaciones de la funciones de la familia **exec**; esto es, el código programa 1 debe usar **execl**, el programa 2 **execv** y el programa 3 **execvp**.

### 2.3.2. Usando fork y exec

Al ejecutar el programa Código 14 se puede notar que solo se ejecuta el primer llamado al **exec** (**execl**), los otros dos llamados (**execv** y **execvp**) nunca se ejecutan, porque una vez culmina la ejecución del primer **exec** el proceso invocador es sobreescrito por el nuevo programa invocado.

Recuerde que, cuando analizamos la función **fork** encontramos que se podían crear copias de procesos y que esas copias en realidad siempre ejecutaban el mismo programa.

Así, según lo anterior tenemos una limitante, por un lado podemos crear copias pero estas ejecutan siempre lo mismo, y por otro lado podemos ejecutar un programa nuevo con llamado a una de las funciones de la familia **exec**, pero una vez hecho esto solo se puede ejecutar un solo programa. Pues bien, es posible solucionar estas limitantes mediante el uso combinado de las funciones **fork** y **exec** tal y como se muestra en la siguiente figura:

|  |
| --- |
|  |
| **Figura 9.** Uso combinado de fork y exec |

El efecto de usar estas dos funciones combinadas es que permiten que un programa pueda correr subprogramas. Como se muestra en la figura anterior, para correr un subprograma (nuevo programa invocado) dentro de un programa, lo primero que tiene que hacer el proceso padre es invocar la función fork para crear un nuevo proceso hijo (el cual es una copia del padre), y luego ese proceso hijo invoca la función exec para empezar el nuevo programa. Lo anterior permite que el programa que realiza la invocación, continúe en ejecución en el lado de ejecución del proceso padre mientras que el programa llamado es reemplazado por el subprograma en el proceso hijo.

El siguiente fragmento de código muestra el esqueleto de cómo se hace uso de estas llamadas en conjunto:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 | *// ...* if (fork == 0) {  *// Este es el hijo*  execvp(path, args); *// Llamado a exec para ejecutar subprograma* } else {  *// Este es el padre*  *// Llamado a wait para esperar a que el hijo termine*  *// (opcional: depende de la situación)*  wait(&status);  } |
| **Código 15.** Llamada en conjunto | |

Realizar un programa que invoque los comandos date y ls (ls debe listar el contenido del directorio raíz). El padre debe imprimir una vez que los dos subprocesos han culminado la frase “**Hasta la vista baby**”. A continuación se muestra el código asociado al ejemplo anterior:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <wait.h>   int main(int argc, char \*argv[]) {  pid\_t pid\_h1, pid\_h2;  int status;  pid\_h1 = fork();  if(pid\_h1 == 0) {  *// Proceso hijo el cual ejecuta el comando ls*  execl("/bin/ls","/bin/ls","/",NULL);  }  else {  pid\_h2 = fork();  if(pid\_h2 == 0) {   *// Proceso hijo que ejecuta el comando date*  execl("/bin/date","/bin/date",NULL);  }  else {  *// Proceso padre*  wait(&status); *// wait para esperar un proceso*  wait(&status); *// wait para esperar el otro proceso*  printf("Hasta la vista baby\n");  }  }  return 0; } |
| **Código 16.** Hasta la vista baby | |

No solo es posible invocar comandos, también se pueden invocar ejecutables hechos por nosotros. Por ejemplo, supóngase que usted compilo un programa el cual imprimía la frase hola mundo, al realizar esto, usted generó el ejecutable con nombre **myExe.out** el cual se encuentra en el directorio de trabajo actual, ahora bien, usted desea invocar este ejecutable desde otro programa con la función exec, el siguiente fragmento de código muestra cómo se desarrolla esta tarea:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 | *// ...* if (fork == 0) {  *// Este es el hijo*  execl("./myExe.out", "./myExe.out", NULL); *// Ejecutar subprograma* } else {  *// Este es el padre*  *// Llamado a wait para esperar a que el hijo termine*  *// (opcional: depende de la situacion)*  wait(&status);  } ... |
| **Código 17.** Ejemplo de llamada en conjunto | |

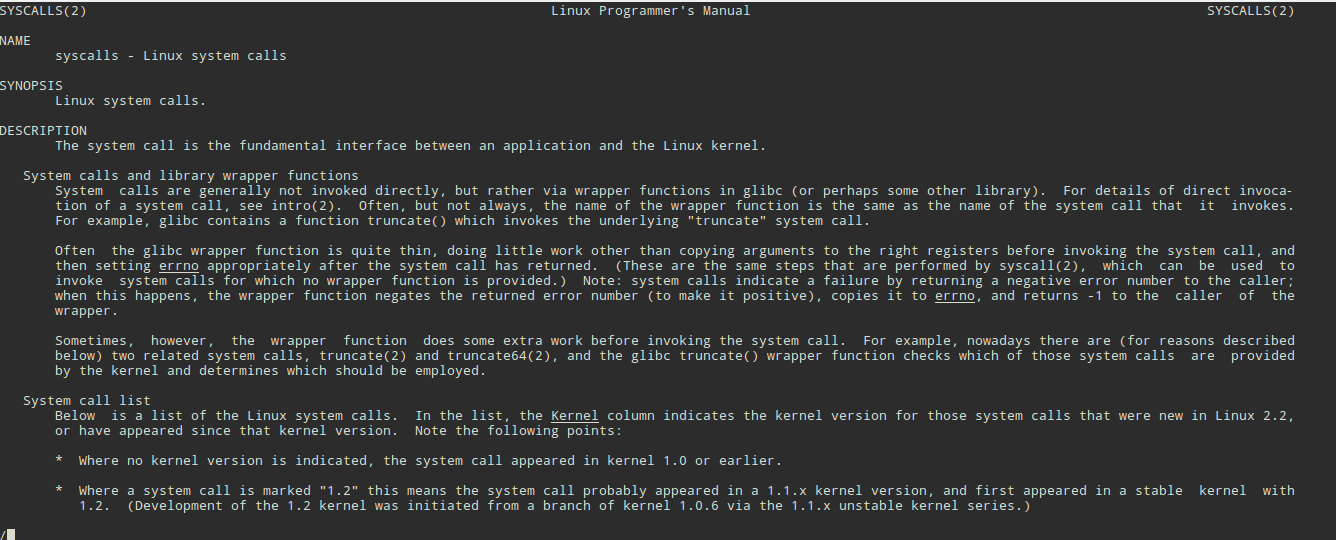
**Ejercicio:**

* Realizar un programa que ejecute un comando que no existe y luego ejecute el comando **ls -l**.

## 3. Taller

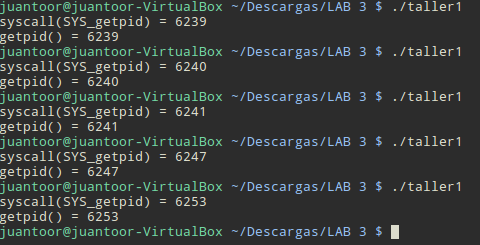
**1.** Escriba en consola “man syscalls” y responda: ¿Qué contiene esta llamada al sistema?

R// http://man7.org/linux/man-pages/man2/syscalls.2.html

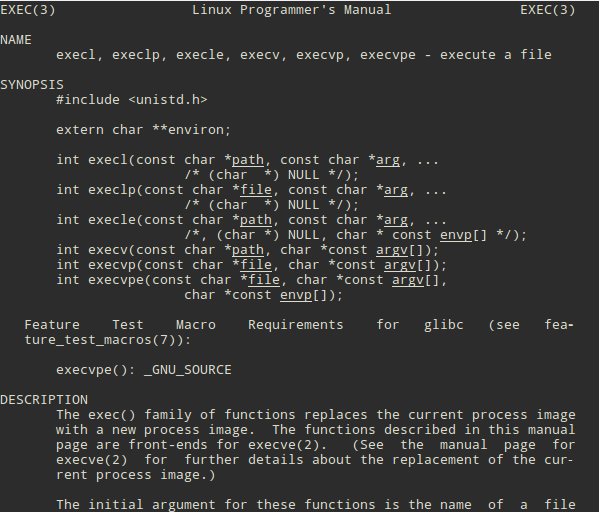


**2.** Explique qué hace el siguiente programa:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 | #include <syscall.h> #include <unistd.h> #include <stdio.h> #include <sys/types.h>  int main(void) {  long ID1, ID2;  */\*--------------------------------\*/*  */\* DIRECT SYSTEM CALL \*/*  */\* SYS\_getpid(func no. is 20) \*/*  */\*--------------------------------\*/*  ID1 = syscall(SYS\_getpid);  printf("syscall(SYS\_getpid) = %1d\n", ID1);   */\*-----------------------------------\*/*  */\* "libc" WRAPPED SYSTEM CALL \*/*  */\* SYS\_getpid(func no. is 20) \*/*  */\*-----------------------------------\*/*  ID2 = getpid();  printf("getpid() = %1d\n", ID2);  return 0; } |



**3.** Ejecute el comando “man execl”. Liste las funciones e indique qué hace cada una de ellas.



La familia de funciones exec reemplaza la imagen del proceso en curso

con una nueva. Las funciones descritas en esta página del Manual son

interfaces para la primitiva execve(2). (Consulte la página del Manual

de execve para información detallada acerca del reemplazo del proceso

en curso.)

El primer argumento de estas funciones es el camino de un fichero que

va a ser ejecutado.

El const char \*arg y puntos suspensivos siguientes en las funciones

execl, execlp, y execle pueden ser contemplados como arg0, arg1, ...,

argn. Todos juntos, describen una lista de uno o más punteros a

cadenas de caracteres terminadas en cero, que representan la lista de

argumentos disponible para el programa ejecutado. El primer argumento,

por convenio, debe apuntar al nombre de fichero asociado con el fichero

que se esté ejecutando. La lista de argumentos debe ser terminada por

un puntero NULL.

Las funciones execv y execvp proporcionan un vector de punteros a

cadenas de caracteres terminadas en cero, que representan la lista de

argumentos disponible para el nuevo programa. El primer argumento, por

convenio, debe apuntar al nombre de fichero asociado con el fichero que

se esté ejecutando. El vector de punteros debe ser terminado por un

puntero NULL.

La función execle también especifica el entorno del proceso ejecutado

mediante un parámetro adicional que va detrás del puntero NULL que

termina la lista de argumentos de la lista de parámetros o el puntero

al vector argv. Este parámetro adicional es un vector de punteros a

cadenas de caracteres acabadas en cero y debe ser terminada por un

puntero NULL. Las otras funciones obtienen el entorno para la nueva

imagen de proceso de la variable externa environ en el proceso en

curso.

Algunas de estas funciones tienen una semántica especial.

Las funciones execlp y execvp duplicarán las acciones del shell al

buscar un fichero ejecutable si el nombre de fichero especificado no

contiene un carácter de barra inclinada (/). El camino de búsqueda es

el especificado en el entorno por la variable PATH. Si esta variable

no es especificada, se emplea el camino predeterminado

``:/bin:/usr/bin''. Además, ciertos errores se tratan de forma

especial.

Si a un fichero se le deniega el permiso (la función intentada execve

devuelve EACCES), estas funciones continuarán buscando en el resto del

camino de búsqueda. Si no se encuentra otro fichero, empero, retornarán

dando el valor EACCES a la variable global errno .

Si no se reconoce la cabecera de un fichero (la función intentada

execve devuelve ENOEXEC), estas funciones ejecutarán el shell con el

camino del fichero como su primer argumento. (Si este intento falla, no

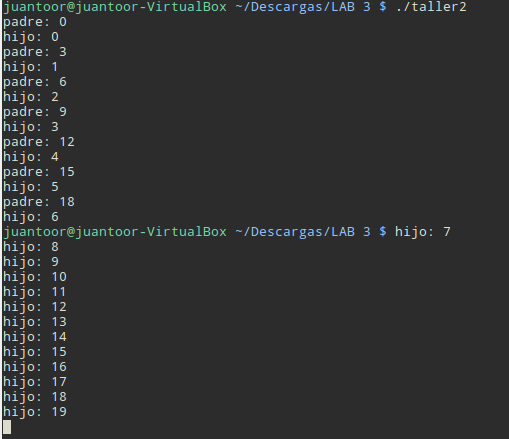
se busca más.)

**4.** Compile y ejecute el siguiente programa:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <unistd.h>  int main(void) {  int fd;  pid\_t pid;  int num;  if ((pid = fork()) < 0) {  perror("fork falló");  exit(-1);  } else if (pid == 0) {  for (num=0; num<20; num++) {  printf("hijo: %d\n", num);  sleep(1);  }  } else {  for (num=0; num<20; num+=3) {  printf("padre: %d\n", num);  sleep(1);  }  } } |

**Responda:**

* ¿Qué significa el retorno de la función fork?
* ¿Cuál es la salida esperada en pantalla?



* ¿Cómo es posible que la sentencia printf reporte valores diferentes para la variable num en el hijo y en el padre?

**5.** Modifique los Códigos del 3 al 8 (añadiendo donde sea necesario un llamado a wait) para que los programas se ejecuten como realmente se esperaría.

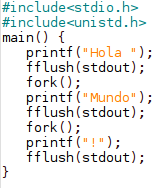
**6.** Dado el siguiente código:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | #include<stdio.h> #include<unistd.h> main() {  printf("Hola ");  fork();  printf("Mundo");  fork();  printf("!"); } |

**Resuelva:**

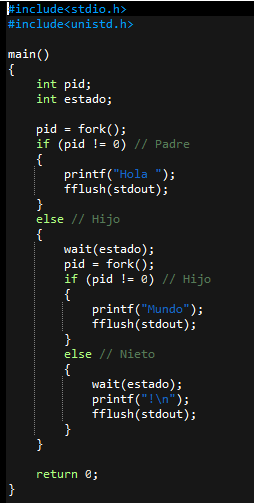
* Sin ejecutarlo dibuje la jerarquía de procesos del programa y determine cuál es la posible salida en pantalla.
* Compile y ejecute el programa. ¿Es la salida en consola la que usted esperaba? ¿Cuál puede ser la razón de esto? (ayuda: función fflush: fflush(stdout);)





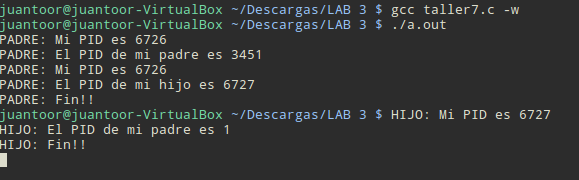


* Modifique el programa de tal manera que se creen exactamente 3 procesos, el padre imprime “Hola”, el hijo imprime “Mundo” y el hijo del hijo imprime “!”, exactamente en ese orden.



**7.** Dado el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26  27 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <sys/wait.h> main() {  pid\_t pid;  int status;  printf("PADRE: Mi PID es %d\n", getpid());  printf("PADRE: El PID de mi padre es %d\n", getppid());  pid = fork();  if(pid == 0){  sleep(5);  printf("HIJO: Mi PID es %d\n", getpid());  printf("HIJO: El PID de mi padre es %d\n", getppid());  printf("HIJO: Fin!!\n");  }  else {  printf("PADRE: Mi PID es %d\n", getpid());  printf("PADRE: El PID de mi hijo es %d\n", pid);  *// wait(&status);*  *// printf("PADRE: Mi hijo ha finalizado con estado %d\n", status);*  printf("PADRE: Fin!!\n");  }  exit(0); } |



**Responda:**

* ¿Cuál es la principal función de sleep en el código anterior?
* ¿Quién es el padre del padre? Use este comando:

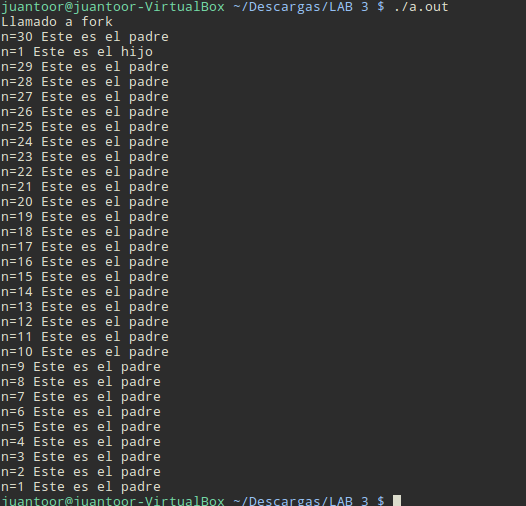
**ps -alf**

* ¿Por qué el proceso hijo imprime el id del padre como 1? ¿Es el que usted espera de acuerdo la jerarquía de procesos?
* Retire el comentario de las líneas de la función wait y la siguiente función printf. ¿Cuál es el identificador del padre ahora? ¿Para qué sirve la función wait? ¿Qué retorna en status?

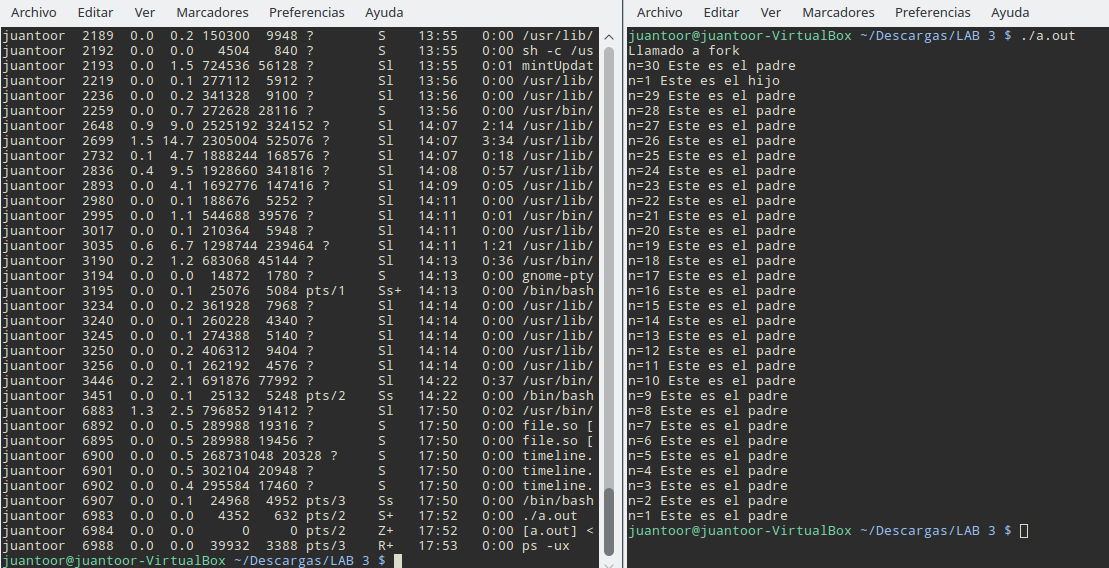
**8.** Proceso Zombie

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | #include <sys/types.h> #include <unistd.h> #include <stdio.h> #include <stdlib.h> int main() {  pid\_t pid;  char \*message;  int n;  printf("Llamado a fork\n");  pid = fork();  switch(pid) {  case -1:  perror("fork falló");  exit(1);  case 0:  message = "Este es el hijo";  n = 1;  break;  default:  message = "Este es el padre";  n = 30;  break;  }  for(; n > 0; n--) {  printf("n=%d ",n);  puts(message);  sleep(1);  }  exit(0); } |

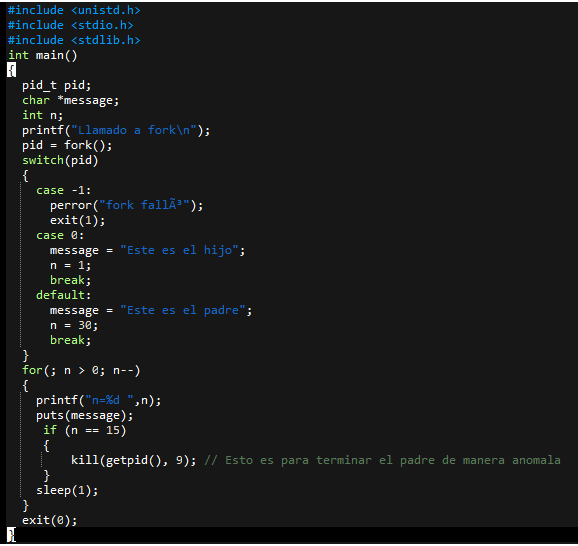
Cuando un proceso hijo termina, su asociación con el padre continúa mientras el padre termina normalmente o realiza el llamado a wait. La entrada del proceso hijo en la tabla de procesos no es liberada inmediatamente. Aunque el proceso no está activo el proceso hijo reside aún en el sistema porque es necesario que su valor de salida exista en caso de que el proceso padre llame wait. Por lo tanto él se convierte en un proceso zombie.

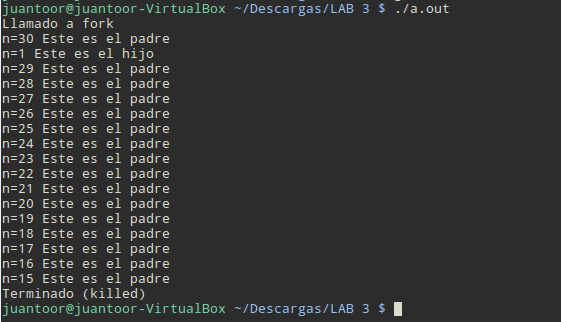


* Realice el comando **ps –ux** en otra terminal mientras el proceso hijo haya finalizado pero antes de que el padre lo haga. ¿Qué observa en las líneas de los procesos involucrados?



* ¿Qué sucede si el proceso padre termina de manera anormal?





**9.** Familia de funciones execl: execl, execlp, execle, exect, execv y execvp y todas las que realizan una función similar empezando otro programa. El nuevo programa empezado, sobrescribirá el programa existente, de manera que nunca se podrá retornar al código original a menos que la llamada execl falle.

*Programa 1:*

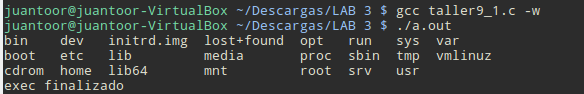
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 | #include <unistd.h> #include <stdio.h> #include <sys/types.h> #include <sys/wait.h> main() {  int pid;  if ((pid = fork()) == 0) {  execl("/bin/ls", "ls", "/", 0);  }  else {  wait(&pid);  printf("exec finalizado\n");  } } |

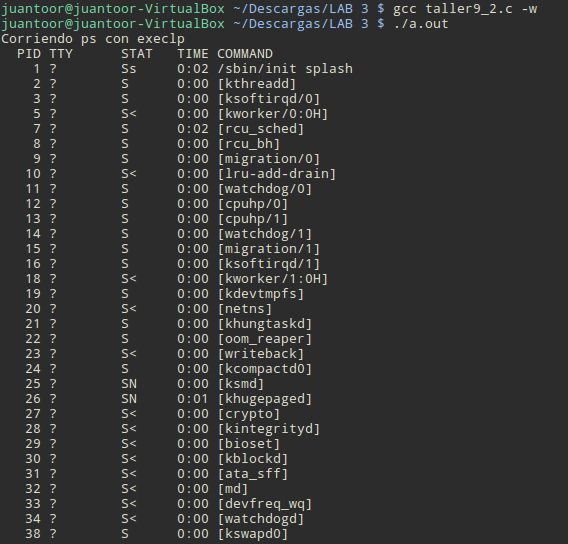
*Programa 2:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 | #include <unistd.h> #include <stdio.h> #include <stdlib.h> int main() {  printf("Corriendo ps con execlp\n");  execlp("ps", "ps", "-ax", 0);  printf("Echo.\n");  exit(0); } |

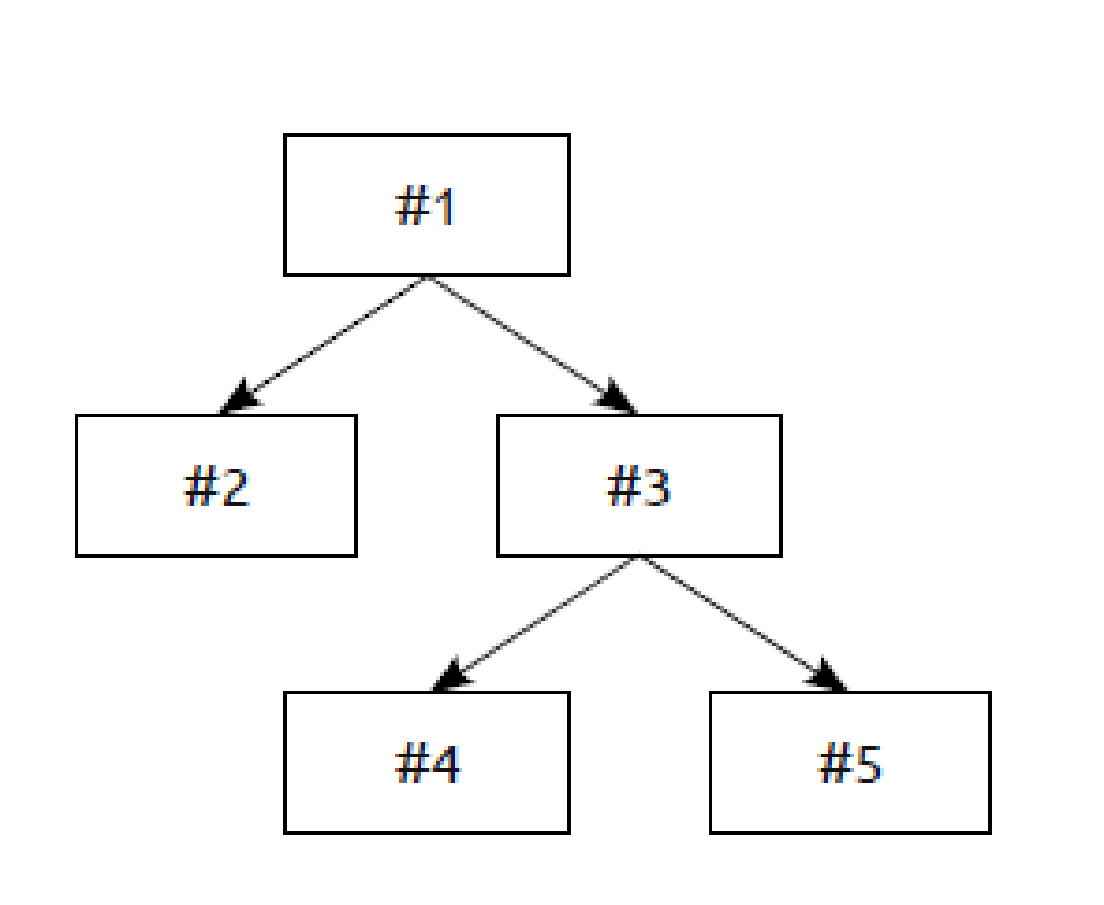
**Resuelve:**

* ¿Qué es lo que hace cada uno de los programas anteriormente mostrados?



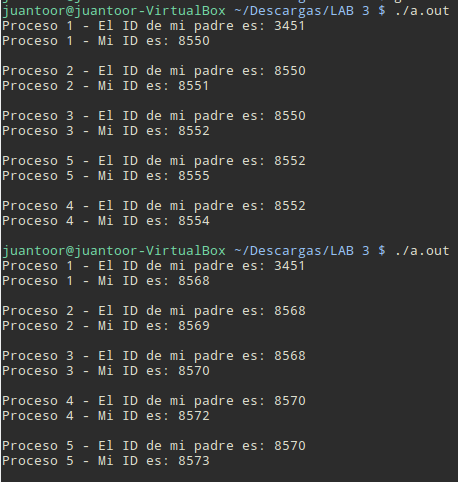


**10.** Haga un programa que cree 5 procesos donde el primer proceso es el padre del segundo y el tercero, y el tercer proceso a su vez es padre del cuarto y el quinto:



El programa debe tener la capacidad de:

* Verificar que la creación de proceso con fork haya sido satisfactoria.
* Imprimir para cada proceso su id y el id del padre.
* Imprimir el id del proceso padre del proceso 1.
* A través de la función system imprimir el árbol del proceso y verificar la jerarquía (pstree).

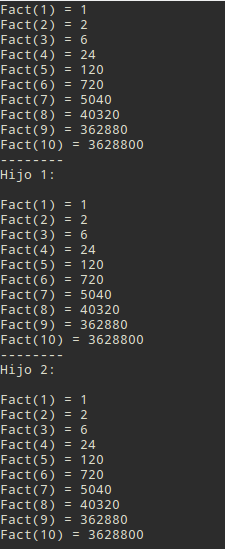


**11.** Codifique un programa que haga lo siguiente:

* Cree 3 procesos diferentes.
* Cada uno de los procesos hijos, calculará por recursión el factorial de los enteros entre 1 y 10, imprimirá los resultados en pantalla y terminará.
* El mensaje impreso por cada proceso debe ser lo suficientemente claro de modo que sea posible entender cuál es el proceso hijo que está ejecutando la operación factorial.
* Una salida tentativa se muestra a continuación (esto no quiere decir que el orden en que se despliegue sea el mismo):

**HIJO1: fact(1) = 1  
HIJO2: fact(2) = 1  
HIJO2: fact(2) = 2  
HIJO1: fact(2) = 2**

* El proceso padre tiene que esperar a que los hijos terminen.



**12.** Realice un programa llamado *ejecutador* que lea de la entrada estándar el nombre de un programa y cree un proceso hijo para ejecutar dicho programa.

**13.** Dado el siguiente fragmento de código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | #include<stdio.h> #include<error.h> #include<stdlib.h> #include<fcntl.h> int main(int argc, char \*argv[]) {  int fd;  int pid;  char ch1, ch2;  fd = open("data.txt", O\_RDWR);  read(fd, &ch1, 1);  printf("En el padre: ch1 = %c\n", ch1);  if ((pid = fork()) < 0) {  perror("fork fallo");  exit(-1); *//Sale con código de error*  } else if (pid == 0) {  read(fd, &ch2, 1);  printf("En el hijo: ch2 = %c\n", ch2);  } else {  read(fd, &ch1, 1);  printf("En el padre: ch1 = %c\n", ch1);  }  return 0; } |

* Cree manualmente el archivo *data.txt* con el siguiente contenido:

**hola**

* Ejecute el programa, capture en pantalla la salida producida. ¿Por qué el programa produce la salida vista? ¿Qué sucede con un padre que abre un archivo, lo hereda?

## 4. Referencias

* Tim Jones, 2010. Kernel command using Linux system calls. <https://www.ibm.com/developerworks/linux/library/l-system-calls/>
* Linux Syscall Reference. https://syscalls.kernelgrok.com

1. Sintaxis: syscall(int number, …); Para mas informacion puede consultar: <http://man7.org/linux/man-pages/man2/syscall.2.html> [↑](#footnote-ref-1)
2. Para mas informacion puede consultar: <http://linux.die.net/man/2/getpid> [↑](#footnote-ref-2)
3. Siglas de “Portable Operating System Interface for uniX”. [↑](#footnote-ref-3)
4. Para ello use el símbolo ampersand (&) después del nombre del ejecutable [↑](#footnote-ref-4)
5. Para más información sobre el comando kill:

   <http://lowfatlinux.com/linux-kill-manual.html>

   <http://es.wikipedia.org/wiki/Kill>

   HYPERLINK "http://es.wikipedia.org/wiki/Kill" [↑](#footnote-ref-5)
6. Para más información puede consultar las siguientes URLs:

   <http://es.wikipedia.org/wiki/Se%C3%B1al_(inform%C3%A1tica)>

   http://en.wikipedia.org/wiki/Signal.h [↑](#footnote-ref-6)