Tejiendo con hilos y procesos

Juan Francisco Cardona M

28 de febrero de 2017

1. Procesos

Un proceso es la abstracción más importante que ofrece un sistema operativo y está se encuentra implementada en una gran parte de los sistemas operativos, pero no todos los sistemas operativos definen las mismas características para este concepto y mucho menos los mecanismos para interactuar con los procesos.

Este taller mostrará las características y las llamadas al sistema que los sistemas basados en Unix[1] y Windows ofrece para los procesos.

1.1. Procesos en Unix

La característica más relevante de los procesos en Unix, es que estos establecen una relación padre-hijo y esta relación se mantiene durante la vida del proceso. El proceso que crea un proceso se llama proceso padre y el proceso creado es el proceso hijo. Esta relación se observa en la figura 2. Esta característica establece que dentro del sistema se mantenga siempre un árbol de procesos cuyo proceso "abuelo" de todos los procesos es el proceso 1 llamado el procesos `init`.

Adicionalmente los procesos son enlazados a un dispositivo de entrada (generalmente el teclado) y a dos dispositivos de salida (la consola o la terminal para mostrar la salida normal de un proceso y la terminal o un archivo para mostrar los errores del proceso). Lo anterior se logra asignando a cada proceso tres manijas (o descriptores) para cada uno de estos recursos. En la figura 1 se puede observar la asignación de dispositivos.

Identificador	**Función**	**Dispositivo**
`stdin`	Entrada de datos	Teclado, archivo
`stdout`	Salida de datos	Terminal, consola, archivo
`stderr`	Errores del programa	Terminal, consola, archivo

La siguiente es la lista de los atributos de un proceso y su función.

[1]

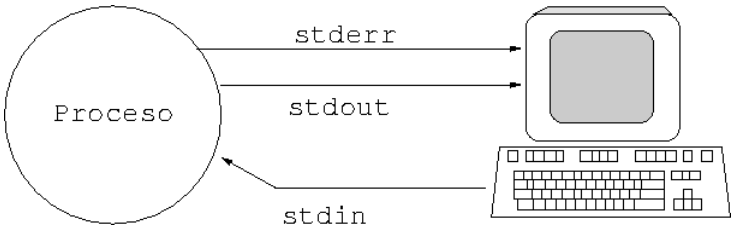

Figura 1: Un proceso en Unix y sus descriptores estándar

Atributo	Descripción	Llamada al sistema
ID de proceso	Identifica de forma única un proceso	`getpid(2)`
ID del padre	Identifica el proceso que lo creo	`getppid(2)`
ID del grupo de proceso	Identifica el proceso que lo creo	`getpgid(2)`

Un proceso hijo cuando es creado comparte los siguientes atributos con sus proceso padre:

- ID de grupo de procesos.
- Directorio actual de trabajo.
- Directorio raíz.
- bits de `umask(2)`.
- Id de usuario real.
- Id de usuario efectivo.
- Banderas de *Set-user-ID* y *Set-group-ID*.
- ID de sesión.
- Terminal de control.
- Mascara de señales y acciones registradas.
- Bandera de *Close-on-exec* para descriptores de archivos abiertos.
- Variables de ambiente.
- Memoria compartida adicionada.
- Limites de recursos.

Ejercicio 1. ¿Cuál es la diferencia entre usuario efectivo y usuario real?
Ejercicio 2. ¿Qué significado tiene las banderas *Set-user-ID* y *Set-group-ID*?
Ejercicio 3. ¿Qué significa *Close-on-exec*?

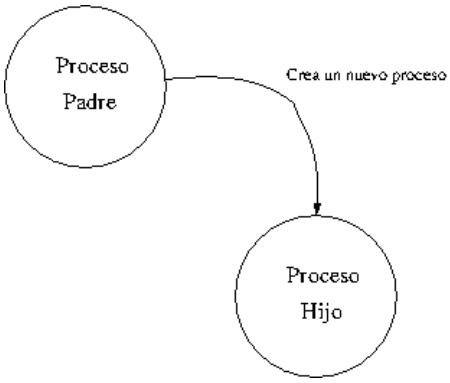

Figura 2: Relación padre-hijo entre proceso en Unix

1.2. Procesos en Windows

En Windows existen varios tipos de procesos, dependiendo del entorno de ejecución que se este trabajando y se distinguen dos muy importantes: los procesos en el ambiente Win32 y los procesos de consola. Centraremos nuestro interés en el segundo. Los procesos tipo consola son similares a los procesos en Unix pero de los no establecen una relación padre-hijo. Pero igualmente tiene asociados tres descriptores (`Handles`) a ellos que tiene las mismas funciones.

2. Creación de procesos

Una de las funciones más interesantes desempeñadas por un sistema operativo es la ejecución de procesos. Estos puede ser creados por distintas circunstancias, a petición de los usuarios, a través de procesos que deciden crear nuevos procesos, etc. Es importante entender de que manera un proceso puede solicitar crear nuevos procesos. Esta forma de creación dependerá como el sistema operativo vea el proceso, generalmente todos los procesos son programas en ejecución, pero cada sistema operativo tiene su forma específica de manejar dicha creación. Pare crear procesos, en primer lugar, se debe identificar la imagen de un ejecutable, que no es más que un archivo que representa el programa. Este archivo generalmente está ubicado dentro de un directorio del sistema de archivos. Entonces debemos indicarle a al sistema operativo la ubicación (directorio) y el nombre del archivo. En algunos casos el directorio se puede omitir por que alguno sistema operativos hacen uso de una variable de ambiente que permite establecer donde están ubicados los ejecutables, en el caso de Unix y Windows esa variable de ambiente se llama `PATH`[2]. Cada sistema operativo tiene su forma propia de crear los procesos dependiendo del modelo de procesos.

[2]El `PATH` es una variable de ambiente que contiene una lista de directorios donde será buscado el ejecutable indicado (El nombre del archivo)

2.1. Unix

En Unix la creación se hace en primer lugar invocando la llamada al sistema `fork(2)`, esta llamada realiza los siguientes pasos:

1. El *kernel* solicita espacio en el sistema de memoria virtual para el nuevo proceso.
2. La tabla de descriptores de archivos es duplicada y están disponibles las misma unidades de archivos en el proceso hijo.
3. Otros recursos compartidos tales como segmentos de memoria compartida están disponibles al proceso hijo.
4. Otros valores del hijo son reiniciados (tiempo de ejecución, el conjunto de señales pendientes son borrados y así).
5. La llamada al sistema `fork(2)` establece el valor que va a retornar a cada proceso. El proceso padre se le retornará el identificador del proceso hijo. El proceso hijo se le retornará un valor cero.

2.1.1. La llamada al sistema `fork(2)`

La sinopsis de la llamada al sistema es como se ve a continuación:

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

pid_t fork(void);
```

2.1.2. Aplicando `fork(2)`

El siguiente programa ilustra un simple programa ejemplo que crea un proceso hijo utilizando la llamada al sistema `fork(2)`.

```
1:   /* LFork.c */
2:
3:   #include <stdio.h>
4:   #include <unistd.h>
5:   #include <string.h>
6:   #include <errno.h>
7:   #include <sys/types.h>
8:
9:   int
10:  main(int argc, char *argv[]) {
11:
12:    pid_t pid;  /* ID de proceso del proceso hijo */
13:
```

```
14:    pid = fork();  /* Crea un nuevo proceso */
15:
16:    if (pid == (pid_t)(-1)) {
17:      fprintf(stderr, "%s, Fallo al hacer el fork\n",
18:       strerror(errno));
19:      exit(13);
20:    }
21:    else if (pid == 0) {
22:      fprintf(stdout, "PID: %ld: El proceso hijo iniciado, el padre es %ld.\n",
23:       (long) getpid(),  /* Identificador del proceso hijo */
24:       (long) getppid()); /* Identificador del proceso padre */
25:    }
26:    else {
27:      fprintf(stdout, "PID: %ld: PID del hijo iniciado: %ld.\n",
28:       (long) getpid(),  /* PID del padre actual */
29:       (long) pid);      /* PID del hijo */
30:    }
31:
32:    sleep(1);
33:    return 0;
34:  }
```

LFork.c

El listado muestra a `fork(2)` siendo invocado en la línea 14. Hasta el retorno de esta llamada, si es existosa los dos procesos son ejecutados. La instrucción `if` en la línea 16 examina si la llamada falló.
La línea 20 examina si el valor retornado por el `fork(2)` es cero. Si es así, las líneas 22–24 son ejecutadas por el proceso hijo. La instrucción `else` en la línea 26 indica que el valor retornado en la variable `pid` no es cero y no es `-1`, entonces las líneas 27–29 son ejecutados por el proceso hijo.
Las líneas remanentes son ejecutas por ambos procesos. La llamada al sistema `sleep(3)` es incluida aquí para que ambos procesos tengan un período de tiempo de espera.

2.1.3. La familia de llamadas al sistema exec(3)

La llamada al sistema `fork(2)` se encarga de crear un nuevo proceso con la misma imagen del proceso que lo creó. Pero, ¿cómo podemos reemplazar la imagen del ejecutable? Ya que sólo `fork(2)` nos permite ejecutar el mismo programa, requerimos de un mecanismo para reemplazar la imagen del proceso actual con una nueva imagen.

La familia de llamadas al sistema `exec(3)` permite reemplazar la imagen del proceso actual con la imagen del nuevo proceso. Existe una familia de funciones para realizar esta operación por que es necesario debido a que algunas veces se conoce el directorio y el nombre del ejecutable, otras veces no se sabe donde

está ubicado, o se conoce de antemano los parámetros que se requiere ejecutar o se deben construir en el momento de ejecutar. Para todas esas anteriores características se tiene el conjunto de llamadas `exec(3)`. La siguiente es la sinopsis de la función `exec(3)`:

```
#include <unistd.h>

extern char **environ;

int execl(const char *camino, const char *arg, ...);
int execlp(const char *fichero, const char *arg, ...);
int execle(const char *camino, const  char  *arg  ,  ...,
           char * const envp[]);
int execv(const char *camino, char *const argv[]);
int execvp(const char *fichero, char *const argv[]);
```

En el siguiente ejemplo se muestra como utilizar la función `exec(3)`. El programa `LDirectorioActual.c` se encarga de crear un nuevo proceso que ejecutará a la utilidad del sistema `ls`[3], mientras el proceso padre esperará que el proceso hijo ejecute esa utilidad, el proceso padre esperará que el hijo termine y al terminar el hijo retornará un valor de retorno a través de la llamada al sistema `exit(3)`. El padre obtendrá dicho valor a través de la llamada al sistema `wait(2)`.

```
 1: /* LDirectorioActual.c */
 2:
 3: #include <unistd.h>
 4: #include <stdio.h>
 5: #include <errno.h>
 6: #include <sys/types.h>
 7: #include <sys/wait.h>
 8: #include <string.h>
 9: int
10: main(int argc, char *argv[]) {
11:
12:   int retVal;
13:   pid_t pid;
14:
15:   pid = fork();
16:
17:   if (pid == (pid_t)(-1)) {
18:     fprintf(stderr, "%s, Fallo al hacer el fork\n",
19:      strerror(errno));
```

[3]Esta utilidad se encarga de listar el contenido de un directorio y mostrar los atributos de dichos archivos

```
20:     exit(13);
21:   }
22:   else if (pid == 0) {
23:     execl("/bin/ls", "ls", "-l", (char *) 0);
24:     // No se debe ejecutar este código
25:     fprintf(stderr, "No pudo ejecutar /bin/ls %s\n", strerror(errno));
26:   }
27:   else {
28:     wait(&retVal);
29:     // Verifica si el hijo terminó bien
30:     if (WIFEXITED(retVal)) {
31:       fprintf(stdout, "El proceso terminó bien: %d\n",
32:        WEXITSTATUS(retVal));
33:     }
34:     else if (WIFSIGNALED(retVal)) { // Fue señalizado
35:       fprintf(stderr, "La señal capturada: %d\n",
36:        WTERMSIG(retVal));
37:     }
38:     else if (WIFSTOPPED(retVal)) {
39:       fprintf(stderr, "El proceso se encuentra parado: %d\n",
40:        WSTOPSIG(retVal));
41:     }
42:   }
43: }
```

LDirectorioActual.c

En la línea 15 ejecutamos la llamada al sistema `fork(2)`, esta creará dos procesos, en la línea 17 se verificará que la operación de `fork(2)` este correcta, si no es así entre la línea 18–20 se manejará el error y se terminará. El hijo ejecutará a partir de la línea 22 y se encargará de reemplazar su imagen con la imagen de otro programa, en la línea 23 se realiza esto llamando a la función `execl(3)`. `execl(3)` es una función de parámetros variables, el primer parámetro indica la ubicación exacta del ejecutable, en este caso el programa de utilidad `ls`, el segundo parámetro indica unicamente el nombre del programa, con el que será reconocido en el sistema. Los siguientes parámetros son una lista de argumentos variables terminada en nulo `(char *) 0`, en la cual pasaremos únicamente el argumento `-l`.
Cuando se invoca alguna de las llamadas al sistema de la familia `exec(3)`, la imagen del proceso es reemplazada con un nuevo programa, si existe un problema en la ejecución del nuevo programa, no se debería ejecutar las líneas posteriores a la línea 23, en la línea 24–25 se controla este error. Mientras el padre ejecutará la líneas 28–42. En la línea 28, el padre espera a que el hijo termine ejecutando la función `wait(3)`, esta función recibe un parámetro que es la dirección de la variable entera donde almacenará el valor de retorno. Esta variable contendrá la información que indica el estado de terminación del hijo

(retornada a través de la la llamada al sistema `exit(3)`), si pudo terminar o la información de la causa de terminación del hijo (una señal, etc.); esta variable encapsula dos tipos de información, el valor de retorno del proceso y un valor si el proceso terminó bien o fue señalizado, no se puede leer dicha información de forma directa, para ello se deben utilizar unas macros especiales que toman el valor de retorno devuelto por la variable `valRet`.
En la línea 30 verifica si el hijo pudo terminar normalmente a través de las macro `WIFEXITED(3)`, esta indica si el proceso terminó bien y si es así, en las líneas 31–32 retorna el valor devuelto por el hijo a través de la macro `WEXITSTATUS(3)`.
La línea 34 se ejecuta si el proceso hijo no terminó por sus propios medios, sino que fue señalado, esto se verifica a través de la macro `WIFSIGNALED(3)`. Si eso fue así se ejecutan las líneas 35–36, para mostrar cual fue la señal que terminó dicho proceso, a través de la macro `WTERMSIG(3)`.
En la línea 38 se ejecuta si el proceso hijo no terminó por sus propios medios y no recibio una señal de terminación sino una señal de parar el proceso, por ejemplo para depurar el proceso, en este caso se verifica este estado a través de la macro `WIFSTOPED(3)`. Si el hijo fue parado, se ejecutan las líneas 39–40, para mostrar cual fue la señal que paro dicho proceso utilizando la macro `WSTOPSIG(3)`.
Ejercicio 4. Modifique el anterior programa para cuando el proceso hijo no encuentre el programa a ejecutar retorne un código de error[4], que el padre indicará que el programa terminó bien o mal.
Ejercicio 5. Modifique el anterior programa para que reciba en la línea de argumentos el nombre de programa a ejecutar y sus argumentos[5].

2.2. Windows

La creación de procesos en Windows únicamente requiere que se invoque una sola llamada `CreateProcess`, pero aunque parezca más fácil requiere que se conozca muy bien cuales parámetros deben ser utilizados.

2.2.1. La llamada `CreateProcess`

La siguiente es la sinopsis de la llamada al sistema `CreateProcess`.

```
BOOL CreateProcess(
   LPCWSTR lpszImageName,
   LPCWSTR lpszCmdLine,
   LPSECURITY_ATTRIBUTES lpsaProcess,
   LPSECURITY_ATTRIBUTES lpsaThread,
   BOOL fInheritHandles,
   DWORD fdwCreate,
   LPVOID lpvEnvironment,
```

[4] El valor retornado por la llamada al sistema `exit(3)` utiliza la siguiente convención `0` para indicar que no hubo ningún problema y cualquier valor positivo diferente de cero para indicar que existió un problema

[5] No utilice la llamada al sistema `execl(3)`, utilice una de estas funciones `execv(3)` o `execvp(3)`

```
    LPWSTR lpszCurDir,
    LPSTARTUPINFOW lpsiStartInfo,
    LPPROCESS_INFORMATION lppiProcInfo
);
```

Cuando se llama a la función `CreateProcess`, el sistema crea un espacio de direcciones virtuales de 4 GB para el nuevo proceso y carga el proceso especificado en dicho espacio de direcciones. El sistema entonces crea un subproceso (o hilo de control) para este proceso. Esto dependerá de la función de inicio dada por módulo de ejecución C. Generalmente es la función `main` en los procesos tipo consola y la funcion `WinMain` en los programas que funcionan bajo el entorno gráfico Win32.

El campo `lpszImageName` es un apuntador a una cadena que identifica el nombre del archivo ejecutable que se desea ejecutar. `CreateProcess` asume que el archivo está en el directorio actual, salvo que se específique una camino donde este se encuentra.

El campo `lpszCommandLine` especifica los argumentos que deben pasarse al proceso. Se puede obtener los comandos dentro de un proceso ya sea por las variables de la función `main` o por la llamada al sistema `GetCommandLine`.

Los campos `lpsaProcess` y `lpsaThread` identifican los atributos se seguridad que se le asignarán al nuevo objeto proceso y al nuevo objeto hilo. Se puede pasar un valor de `NULL` a estos procesos.

El campo `fdwCreate` define los indicadores que afectarán a la creación del nuevo proceso. No solamente del nuevo tipo de proceso, si no, también de la prioridad a la cual será ejecutada.

El campo `lpvEnvironment` apunta a un bloque de memoria que contiene las cadenas de entorno que serán utilizadas por los nuevos procesos. Muchas veces el valor es `NULL` que implica utilizar las mismas cadenas del proceso creador. Estas se pueden obtener a través de dos formas utilizando un tercer parámetro en la función `main` llamado `argenv` (Ver el siguiente código). O utilizando las funciones `GetEnvironmentStrings` que obtiene un apuntador a la lista completa o la función `GetEnvironmentVariable` que obtiene el apuntador a la cadena manejada por dicha variable (si existe).

```
int
main(int argc, char *argv[], *argenv[]) {
...
}
```

El campo `lpszCurDir` permite crear un nuevo proceso ubicado en un directorio de trabajo distinto. Si el valor es NULL, se utilizará el mismo directorio donde se encuentra el proceso.

El campo `lpsiStartInfo` apunta a una estructura `STARUPTINFO`.

El campo `lppiProcInfo` apunta a una estructura `PROCESS_INFORMATION` que `CreateProcess` habrá rellenado antes de retornar. La estructura se define como sigue:

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Este información se refiere a los `HANDLE` que contiene información sobre los objetos de proceso y del hilo de control incial. Los otros dos campos contiene información de los identificadores internos de los procesos y los hilos. Este campo siempre debe ser pasado a la función `CreateProcess`.

2.2.2. Utilizando la llamada `CreateProcess`

En el siguiente código se muestra como se utiliza la llamada al sistema `CreateProcess` para crear un nuevo proceso.

```
 1: /* WCreateProcess.c */
 2:
 3: #include <Windows.h>
 4:
 5: int
 6: main(int argc, char *argv[])
 7: {
 8:   STARTUPINFO startupInfo;
 9:   PROCESS_INFORMATION piProcInfo;
10:
11:   GetStartupInfo(&startupInfo);
12:
13:   if (CreateProcess(NULL, "NOTEPAD CreateProcess.c", NULL, NULL,
14:      FALSE, 0, NULL, NULL, &startupInfo,
15:      &piProcInfo)) {
16:     WaitForSingleObject(piProcInfo.hProcess, 0);
17:   }
18:   else {
19:   }
20: }
```

WCreateProcess.c

Aunque el anterior programa se ve muy sencillo hay que tener en cuenta varios detalles. En primer lugar en la línea 3 debemos incluir un único archivo de

encabezado[6] llamado `Windows.h`, en el están definidos todos los demás archivos de encabezados para las llamadas al sistema Win32.
En la línea 8 aparece la declaración de la estructura `STARTUPINFO`, esta estructura contiene información adicional sobre el proceso a crear, alguno de los miembros se puede utilizar para las aplicaciones gráficas, mientras que otros solo tiene que ver con las aplicaciones en modo consola. La siguiente es el contenido de la estructura en mención.

```
typedef struct _STARTUPINFO {
  DWORD cb;
  LPTSTR lpReserved;
  LPTSTR lpDesktop;
  LPTSTR lpTitle;
  DWORD dwX;
  DWORD dwY;
  DWORD dwXSize;
  DWORD dwYSize;
  DWORD dwXCountChars;
  DWORD dwYCountChars;
  DWORD dwFillAttribute;
  DWORD dwFlags;
  WORD wShowWindow;
  WORD cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

Podemos ver información que tiene que ver con una aplicación o con la otra. Pero nos interesa mostrar que hay tres campos que serán muy importantes para nuestro curso, estas son los campos `hStdInput` (representa la entrada estándar), `hStdOutput` (representa la salida estándar) y por último `hStdError` (representa el error estándar). En la línea 11 obtenemos la información de inicialización del proceso actual a través de la función `GetStartupInfo`, esta nos llena una estructura con la información del proceso en cuestión y puede ser utilizada para crear nuevos procesos.

En la línea 13 llamamos la función `CreateProcess` para crear un nuevo proceso que invocará al programa `Notepad`, observe que una gran parte de los parámetros pasados están en `NULL`, pero hay algo muy interesante, no se utilizó el primer parámetro, pero si se utilizó el segundo en el cual se indica la línea de comando.

En la línea 16 el programa espera que el proceso termine, utilizando el valor retornado en el último parámetro que indica el `HANDLE` del proceso creado.

[6] No es un librería

En la línea 18–19 se debería manejar el error pero no se hace en favor de implementar un ejercicio.
Ejercicio 6. Modifique el anterior programa para que retorne el código de error al no poder ejecutar el programa solicitado.
Ejercicio 7. Modifique el anterior programa para que reciba en la línea de argumentos el nombre de programa a ejecutar y sus argumentos.
Ejercicio 8. . Modifique el anterior programa para que pase el nombre del programa ejecutable en el primer parámetro.

3. Estados de los procesos

El sistema operativo debe guardar la información sobre todos los recursos que maneja y debe permitir al usuario consultar sobre el estado de ellos ya sea a través de llamadas al sistema o a través de programas del sistema. Los procesos son uno de esos recursos y por lo tanto existe manera de obtener la información sobre los mismos. Observemos en primer lugar, el comando `ps` en Unix, este comando permite listar los procesos activos del usuario que lo invoca, pero también le permite obtener sobre los procesos de otro usuario, de un grupo de usuarios, aún más, se puede obtener información sobre el estado de un proceso o de varios en general.
Ejercicio 9. ¿Cómo se obtiene la información del estado de un proceso en ejecución?
Ejercicio 10. ¿Qué significa cada uno de los estados de un proceso, son los mismos vistos en clase? **Ejercicio 11.** ¿De un usuario específico?
Ejercicio 12. ¿Qué significa cada uno de los siguientes campos: UID, EPID, PID, PPID, SZ, WCHAN, TTY, STIME y TIME?

4. Finalizar un proceso

Un proceso puede terminar de dos maneras, llegar a la última instrucción de su programa principal o suicidarse o ser asesinado.

4.1. Última instrucción

En el siguiente código el programa alcanzará su finalización ejecutando la última instrucción.

```
#include <stdio.h>

int
main() {

  printf("Hola Mundo\n");
  return 0; // Ultima línea el programa termina
```

```
}
```

Esta es una manera de terminar elegante, pero no siempre los programa terminan en la última instrucción, sino que pueden terminar en otras partes.

4.2. Suicido

Un programa termina si invoca la función `exit(3)` en Unix o la función `TerminateProcess` en Windows. La siguiente son la sinopsis de las llamadas.

```
#include <stdlib.h>

void exit(int status);


  VOID TerminateProcess( void )
```

En ambas funciones se retorna un valor que indica que el proceso ha terminado. Este valor puede ser capturado por el proceso padre u otro proceso a través de las funciones `wait(3)` en Unix y `GetExitCodeProcess`. La siguiente es la sinopsis de dichas funciones.

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

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


BOOL GetExitCodeProcess(
   HANDLE hProcess,
   LPDWORD lpExitCode
);
```

4.3. Asesinar procesos

Un proceso puede matar a otro proceso utilizando las llamadas al sistema especificas para realizarlo o utilizando utilidades del shell que suministran el mecanismo para hacerlo.

4.3.1. Unix

En Unix se utiliza el envio de señales y más apropiadamente la función `kill(2)` para asesinar un proceso. Las señales están indicadas por números y cada una de ellas tiene un significado especial. Hay varias señales para matar procesos dos de ellas son: SIG_KILL y SIG_TERM.

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

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

4.3.2. Windows

En Windows se utiliza otro mecanismo para matar procesos y se hace a través de la función `TerminateProcess`:

```
BOOL GetExitCodeProcess(
   HANDLE hProcess,
   LPDWORD lpExitCode
);
```

Ejercicio 13. Realize un programa en Linux y otro en Windows que se encarge de matar procesos únicamente dando el número del proceso.

5. Control de procesos

5.1. Señales en Unix

Las señales son un mecanismo semejante a las interrupciones, pero a diferencia de las interrupciones, que ocurren por hardware, las señales ocurren por software. Estas últimas son usadas para señalar un evento asincrónico a un proceso, como por ejemplo: indicarle a un proceso que termine a petición del usuario, o una alarma ha sido activada o un proceso quiere indicarle a otro que debe realizar alguna otra tarea.

El mecanismo para el manejo de señales es un mecanismo potente, por que permite que un proceso determine si puede o no recibir una señal en un momento determinado[7]. En el siguiente ejemplo vamos a mostrar como se pueden manejar las señales en los procesos.

```
0: /* LProcesoSinSenal.c */
1: #include <stdio.h>
2: #include <unistd.h>
3:
4: #define TIMEOUT 30
5:
6: int
7: main(int argc, char *argv[]) {
8:
```

[7] Aunque no todas las señales tienen este comportamiento, algunas señales tiene un comportamiento que no puede ser modificado

```
 9:   int valor;
10:
11:   /**
12:    * Entra en un ciclo infinito esperando ser
13:    * terminado
14:    */
15:   for (;;) {
16:     fprintf(stdout, "Esperando señal\n");
17:     sleep(TIMEOUT);
18:   }
19: }
```

LProcesoSinSenal.c

El anterior programa, cuando es convertido en un proceso, queda en un ciclo infinito. Este ciclo infinito es logrado en primer lugar a través de la instrucción `for`, observe que no tiene inicializacion, ni condición, incremento, es decir se va ejecutar por siempre; la segunda forma de como hacer que el programa consuma ciclos de CPU sin utilizar es a través de la función `sleep(3)` que pone a dormir el programa.

Como se puede terminar el anterior programa. Un usuario, lo puede hacer de muchas formas, pero todas ellas implican el envio de señales. La primera forma es presionando las teclas `Ctrl+C`, esta señal, es llamada de *break* y simplemente interrumpe un program, o puede ser a través de la señal de terminar, que son las teclas `Ctrl+Z`, o la señal de parada `Ctrl+\`. Probemos ejecutar y parar el anterior programa utilizando una de las posibles señales como la señal de *break*:

```
$ ./LProcesoSinSenal
Esperando señal
Esperando señal
Esperando señal
[Ctrl+C]
```

Otra forma de enviar señales a los procesos es a través del comando `kill(1)`. La sinopsis:

```
kill -s signal pid
```

Donde `signal` es el nombre de la señal o el número de señal y `pid` es el número del proceso. Si necesita conocer todas las señales que permite kill, ejecute el siguiente comando:

```
$ kill -l
```

Si queremos matar el anterior proceso sin utilizar `Ctrl+C`; abra una nueva terminal, e identifique el número del proceso.

```
$ ps -a | grep ProcesoSinSenal
  PID TTY          TIME CMD
  1608 pts/1    00:00:00 ssh
  2423 pts/3    00:00:00 LProcesoSinSenal
  2424 pts/2    00:00:00 ps
$ kill -s SIGINT 2423
```

Verá en la terminal donde esta corriendo el proceso lo siguiente[8]:

```
$ ./LProcesoSinSenal
Esperando senal
Llego: 2
Esperando senal
$
```

Si se desea capturar señales, son muchos los interrogantes que se deben resolver: ¿Cómo puede un programa obtener información sobre las señales que ocurren en un momento determinado? ¿Cómo puede lograr capturar una señal? ¿Cómo puede identificar las señales? ¿Cómo hace un lenguaje de programación para identificar el código encargado de manejar las señales?

Cada señal que es enviada a un programa tiene un número de identificación, se puede obtener la información acerca de las señales que el sistema operativo, permite enviarle a un proceso, mirando la información sobre el tema de señales en `signal(7)`. En algunos sistemas operativos basados en Unix, se tiene aproximadamente 32 señales[9], cada una esta identificada por un número, pero en algunos casos es mejor utilizar una constante que denota el número, por que estos números no se mantienen constante entre las diferentes versiones de Unix. Por ejemplo para capturar la señal de *break* o interrupción, no utilizamos su número que algunos sistemas operativos está definida con el valor `2`, es mejor utilizar su nemotecnico que es `SIGINT`.

Ejercicio 14. Entender cuando ocurren cada una de las señales que Linux soporta.

Ejercicio 15. Enviar cada una de las señales que ofrece Linux al proceso y observar que ocurre.

Para instalar un manejador de señales se hace a través de la llamada al sistema `signal(2)`, la siguiente es la sinopsis de dicha llamada al sistema:

```
#include <signal.h>
```

[8] El número de la señal capturada dependerá de la versión de Unix o Linux

[9] Linux soporta 64 señales

```
void (*signal(int signum, void (*manejador)(int)))(int);
```

La función recibe dos parámetros, el primero es la identificación de la señal `signum` a manejar, el segundo, es la dirección de la función manejadora de la señal, esta función tiene el siguiente formato:

```
int
manejador(int signum) {
  ⋮
}
```

`signal(2)` retorna la dirección de la función manejadora, que nos sirve para re-instalarla posteriormente.

No todas la señales se pueden manejar, algunas señales no permiten hacerlo. Por ejemplo la señal `SIGKILL`, esta se encarga de matar un proceso de forma inmediata. Es importante que esta señal no permita ser capturada, por que sino un proceso podría ejecutarse eternamente.

No todas las señales se comportan de la misma manera como las señales, que hemos visto anteriormente. Algunas señales cuando son lanzadas y no existen manejadores para ellas, terminan el programa que las lanzó, otras son ignoradas y otras paran o suspenden la ejecución del proceso.

Miremos el siguiente código que permite capturar algunas señales como: SIGINT, SIGQUIT y SIGTERM.

```
 1: /* LSenalProceso.c */
 2:
 3: #include <stdio.h>
 4: #include <unistd.h>
 5: #include <signal.h>
 6:
 7: #define TIMEOUT 30
 8:
 9: /**
10:  * Codigo manejador de señales
11:  */
12: void
13: manejador(int senal) {
14:
15:   fprintf(stderr, "Llego: %d\n", senal);
16:
17: }
18:
19: int
20: main(int argc, char *argv[]) {
```

```
21:
22:   /**
23:    * Establece los manejadores de señales
24:    */
25:
26:   signal(SIGINT, manejador);
27:   signal(SIGQUIT, manejador);
28:   signal(SIGTERM, manejador);
29:
30:    /**
31:    * Entra en un ciclo infinito esperando ser
32:    * terminado
33:    */
34:   for (;;) {
35:     fprintf(stdout, "Esperando señal\n");
36:     sleep(TIMEOUT);
37:   }
38: }
```

LSenalProceso.c

Entre las líneas 12 a la 17, se encuentra la definición del manejador de señales, este es un manejador sencillo que únicamente se encarga de mostrar que número de señal se ha capturado. Solamente se puede instalar un manejador de señal por señal, pero para varias señales puede existir un único manejador, como se muestra en el programa.

En las líneas 26 a la 28, esta la forma de instalar un manejador, este instala el mismo manejador para varias señales. Aunque el programa no lo hace es recomendable guardar la dirección retornada por el manejador de señales, para cuando no deseamos capturar la señal re-instalemos el manejador anterior.

Miremos la ejecución del anterior programa:

```
$ ./LSenalProcesos
Esperando señal
[Ctrl+C]
Llego: 2
Esperando señal
[Ctrl+/]
Llego: 3
Esperando señal
```

Ejercicio 16. Ejecutar el anterior programa y probarlo con cada una de las señales que Linux ofrece.

Ejercicio 17. Envíe varias veces la misma señal al proceso, ¿Qúe pasa?

Ejercicio 18. Modifique el anterior programa y trate de instalar más mane-

jadores (diferentes) para diferentes señales, ¿Cuáles señales pueden se capturadas? **Ejercicio 19.** ¿Cómo puedo reinstalar los manejadores anteriores? **Ejercicio 20.** Modifique el manejador y pongaló a esperar un tiempo, utilizando la función `sleep(3)` y envielé la señal que captura dicho manejador y mientras está esperando, vuelva enviar la misma señal. ¿Qué pasa? Envie una señal diferente ¿Qué pasa? ¿Cómo se podría evitar dicho comportamiento[10]?

6. Creación de hilos

Uno de los inconvenientes con la programación de procesos de un solo hilo de control, es que casi todas las llamadas son bloqueantes, es decir un proceso se bloquea hasta que la operación solicitada se lleve a cabo. ¿Cómo permitir que el proceso continue su ejecución realizando otra tarea? Es aquí donde entran los hilos. Mientras el hilo de control principal realiza una operación, un nuevo hilo puede esperar a que la operación solicitada termine.
Miremos como llevar a cabo dicha solución utilizando hilos.

6.1. Creación de hilos en POSIX

En POSIX los hilos se crean utilizando la llamada al sistema `pthread_create(3)`.

6.1.1. La función `pthread_create`

```
#include <pthread.h>

int  pthread_create(pthread_t  *thread,
                    pthread_attr_t *attr,
                    void * (*start_routine)(void *),
                    void * arg);
```

La función `pthread_create(3)` tiene los siguientes parámetros.

- El parámetro `thread` es un apuntador a tipo de dato `pthread_t` que contiene el identificador del hilo de ejecución.
- El parámetro `attr` es el apuntador a un tipo de dato `pthread_attr_t`, este indica que conjunto de atributos tendrá un hilo, entre otros: si un hilo puede obtener el estado de terminación de otro, que politica de planificación tiene, que parámetros de planificación, etc. Para ver que tipos de atributos se tienen mirar `pthread_attr_init(3)`. Se puede utilizar el valor NULL.
- El parámetro `start_routine` es un apuntador a una función donde el hilo se ejecutará dicha función debe tener la siguiente firma:

[10] `sigaction(2)`

```
void*
funcion_hilo(void *arg) {
   ...
}
```

- El parámetro `arg` es el argumento que será pasado a la función que implementa el hilo.

Para esperar que un hilo termine su ejecución, debemos utilizar la función `pthread_join(3)`:

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

`pthread_join(3)` espera que un hilo identificado por `th` termine tanto por una llamada explicita a la función `pthread_exit(3)` o la función retorne explícitamente a través de la invocación de `return` o haya sido cancelado. Si `thread_return` no es NULL, el valor de retorno del hilo `th` es almacenado en la ubicación apuntada por `thread_return`.

El siguiente es el código para crear un hilo y esperar que el termine.

```
 1: /* LCrearPThread.c */
 2:
 3: #include <pthread.h>
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6:
 7: void*
 8: funcion_hilo(void *arg) {
 9:
10:   int valor = (int) arg;
11:
12:   fprintf(stdout, "Hola Mundo del hilo %d\n", valor);
13:
14:   sleep(valor);
15:
16:   fprintf(stdout, "Va a terminar hilo %d\n", valor);
17:
18:   valor *= 100;
19:
20:   return (void *) valor;
21: }
22:
23: int
```

```
24: main(int argc, char *argv[]) {
25:
26:   int nHilos = 0;
27:   int i;
28:   pthread_t *tablaDeHilos;  // Información de los hilos
29:   int valorRetorno;
30:
31:   if (argc != 2) {
32:     fprintf(stderr, "Uso: %s nHilos\n", argv[0]);
33:     exit(1);
34:   }
35:
36:   nHilos = atoi(argv[1]);
37:
38:   if (nHilos == 0) {
39:     fprintf(stderr, "Uso: %s nHilos\n", argv[0]);
40:     exit(1);
41:   }
42:
43:   // Solicito memoria dinámica para la tabla
44:   tablaDeHilos = (pthread_t *) malloc(sizeof(pthread_t) * nHilos);
45:
46:   for (i = 0; i < nHilos; i++)
47:     pthread_create((tablaDeHilos + i),
48:      NULL,
49:      funcion_hilo,
50:      (void *) i);
51:
52:   for (i = 0; i < nHilos; i++) {
53:
54:     pthread_join(*(tablaDeHilos +i),
55:   (void **) &valorRetorno);
56:     fprintf(stdout, "Valor de retorno: %d del hilo: %ld\n",
57:      valorRetorno, *(tablaDeHilos +i));
58:   }
59:
60:   exit(1);
61: }
```

LCrearPThread.c

Para compilar el anterior programa se le indica al compilador que debe utilizar la biblioteca de hilos POSIX.

```
$ gcc -o  LCrearPThread LCrearPThread.c -lpthread
```

Para ejecutar digite el siguiente comando desde una sesión del interprete de comandos (`shell`).

```
$ ./LCrearPThread 4
```

6.2. Creación de hilos en Windows

6.2.1. La llamada al sistema `CreateThread`

```
HANDLE CreateThread(
     LPSECURITY_ATTRIBUTES lpsa,
     DWORD cbStack,
     LPTHREAD_START_ROUTINE lpStartAddr,
     LPVOID lpvThreadParm,
     DWORD fdwCreate,
     LPDWORD lpIDThread);
```

La siguiente es la lista de los parámetros:

- El parámetro `lpsa` es un apuntador a la estructura `SECURITY_ATTRIBUTES` que guarda la información sobre el atributo de seguridad. Puede ser pasado un valor de NULL.
- El parámetro `cbStack` señala el tamaño de la pila, se puede pasar un valor de 0 para que tome el valor por omisión.
- El parámetro `lpStartAddr` es el apuntador a la función que ejecutará el hilo, la función tiene la siguiente firma:

  ```
  DWORD WINAPI
  funcion_hilo(LPVOID lpvThreadParam) {
    ...
  }
  ```

- El parámetro `lpvThreadParm` es el apuntador al parámetro que le vamos a pasar a la `funcion_hilo`.
- El parámetro `fdwCreate` puede tomar uno de dos valores posibles. Si el valor es `0`, el proceso comienza a ejecutarse de inmediato. Si el valor es `CREATE_SUSPENDED`, el sistema crea el subproceso y queda preparado para la ejecución de la primera instrucción de la función `funcion_hilo`.
- El último parámetro `lpIDThread` debe tener una dirección válida a DWORD, donde `CreateThread` almacenará el ID que Windows NT asigna al nuevo subproceso. El valor de este parámetro *no puede ser* NULL.

El siguiente es el código para crear un hilo en Windows:

```
 1: #include <Windows.h>
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4:
 5: DWORD WINAPI
 6: funcion_hilo(LPVOID lpParameter) {
 7:
 8:     int valor = (int) lpParameter;
 9:     DWORD dwResultado = 0;
10:
11:     printf("");
12:     fprintf(stdout, "Hola mundo desde el hilo: %d\r\n", valor);
13:
14:     Sleep(valor * 1000); // Esta expresion esta en milisegundos
15:
16:     dwResultado = (DWORD)100 * valor;
17:     return dwResultado;
18: }
19:
20: int
21: main(int argc, char *argv[]) {
22:
23:     int nHilos;
24:     DWORD dwResultado;
25:     int i;
26:     DWORD *tablaHilos;
27:
28:     if (argc != 2) {
29:         fprintf(stderr, "Uso: %s nHilos\r\n", argv[0]);
30:         ExitProcess(10U);
31:     }
32:
33:     nHilos = atoi(argv[1]);
34:
35:     if (nHilos == 0) {
36:         fprintf(stderr, "Uso: %s nHilos\r\n", argv[0]);
37:         ExitProcess(10U);
38:     }
39:     tablaHilos = (LPDWORD) malloc(sizeof(LPDWORD) * nHilos);
40:
41:     for (i = 0; i < nHilos; i++) {
42:
43:         CreateThread(NULL,
44:                      0,
```

```
45:                     funcion_hilo,
46:                     (LPVOID) i,
47:                     0,
48:                     (tablaHilos + i));
49:     }
50:
51:     for (i = 0; i < nHilos; i++) {
52:         HANDLE hThread;
53:
54:         hThread = OpenThread(THREAD_ALL_ACCESS,
55:                              FALSE,
56:                              *(tablaHilos + i));
57:         WaitForSingleObject(hThread, 0);
58:         GetExitCodeThread(hThread, &dwResultado);
59:         fprintf(stdout, "El hilo: %ld terminó: %ld\r\n",
60:                 *(tablaHilos + i),
61:                 dwResultado);
62:     }
63:
64:     return 0;
65: }
66:
```

WCreateHilosWindows.c

Para compilar el anterior programa:

```
> cl WCreateHilosWindows.c
```

Para ejecutarlo:

```
> WCreateHilosWindows 5
```

Esto lo ejecuta con 5 hilos.

7. Atributos de los hilos

7.1. Atributos de hilos en POSIX

Cada hilo POSIX tiene un objeto de atributo asociado que representa sus propiedades. Un objeto atributo de hilo puede estar asociado a varios hilos y tiene funciones para crear, configurar y destruir objetos atributo.

```
#include <pthread.h>
```

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);

int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inherit);

int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int *inherit);

int pthread_attr_setscope(pthread_attr_t *attr,
                          int scope);

int pthread_attr_getscope(const pthread_attr_t *attr,
                          int *scope);
```

La función `pthread_attr_init(3)` inicializa un objeto atributo de hilos con los valores por omisión. La función `pthread_attr_destroy(3)` hace que el valor del objeto de atributos sea no válido. Ambas funciones llevan un único argumento que es un puntero a un objeto de atributo de hilos, de tipo `pthread_attr_t`.

Los atributos de los hilos son consultados únicamente cuando creamos un nuevo hilo. El mismo atributo puede ser utilizado para varios hilos. Modificar un objeto de atributo después de llamar un `pthread_create` no cambia los atributos de un hilo previamente creado.

Las restantes funciones para el manejo de atributos llevan dos parámetros, el primero el puntero al atributo y el segundo el valor del atributo o un puntero a un valor (eso depende de la función).

Un hilo tiene un estado de conexión o desconexión, las funciones `pthread_attr_setdetachstate(3)` y `pthread_attr_getdetachstate(3)` puede examinar y establecer el estado de desconexión de un hilo. Los posibles valores de `detachastate` controlan si el hilo está en un estado juntable (*joinable*) con el valor de `PTHREAD_CREATE_JOINABLE` o el estado de desconectado con el valor `PTHREAD_CREATE_DETACHED`. Por omisión los hilos son juntables. Los hilos desconectados invocan a `pthread_detach` cuando terminan para liberar sus recursos.

Las funciones `pthread_attr_getscope(3)` y `pthread_attr_setscope(3)` examinas y establecen el atributo `scope` (alcance de contención) que controla si el hilo compite por recursos dentro del proceso o bien en el nivel del sistema. El único valor soportado por los hilos en Linux es la implementación de `PTHREAD_SCOPE_SYSTEM` que significa que los hilos compiten por el tiempo de CPU con todos los procesos corriendo en la máquina. En particular, las prioridades de los hilos son interpretadas relativa a la prioridades de otros procesos.

La política de programación de hilo se almacena en la estructura de tipo `struct sched_param`. El submiembro `sched_policy` de `struct sched_param` contiene la política de programación. Las posibles políticas de programación son "el primero que entra es el primero que sale" (`SCHED_FIFO`), turno circular (`SCHED_RR`) o definido por la implementación (`SCHED_OTHER`). La implementación más común de `SCHED_OTHER` es una política de prioridad apropiativa.

7.2. Atributos de los hilos en Windows

7.2.1. Clases de prioridad de un proceso

Windows NT (y posteriores versiones) contemplan cuatro clase de prioridad: desocupado, normal, alta y de tiempo real. La asignación de una clase de prioridad a un proceso se realiza mediante la suma lógica (OR) de uno de los indicadores de la función `CreateProcess` cuando esta es invocada.

Clase de prioridad	**Indicador de `CreateProcess`**	**Nivel**
Desocupado	`IDLE_PRIORITY_CLASS`	4
Normal	`NORMAL_PRIORITY_CLASS`	9-17
Alta	`HIGH_PRIORITY_CLASS`	13
Tiempo real	`REALTIME_PRIORITY_CLASS`	24

7.2.2. Definición de la prioridad relativa de un hilo

Una vez que un hilo ha sido creado, este se ejecuta al nivel de la clase de prioridad del proceso que lo creó. Sin embargo, es posible cambiar el nivel de prioridad de un proceso individual. Esta nueva prioridad siempre será relativa a la prioridad del proceso al cual el hilo pertenece.

La prioridad de un hilo (o subproceso[11] es establecida a través de la llamada al subsistema Win32:

```
BOOL SetThreadPriority(
```

[11] Subproceso, proceso de bajo peso e hilos son los nombres más conocidos de los hilos

```
HANDLE hThread,
int nPriority);
```

El primer parámetro es el descriptor del subproceso cuya prioridad deseamos cambiar. El parámetro `nPriority` puede ser uno de los siguientes valores:

Identificador	`Significado`
`THREAD_PRIORITY_ABOVE_NORMAL`	La prioridad del hilo debería ser de 1 sobre la clase de prioridad del proceso
`THREAD_PRIORITY_BELOW_NORMAL`	La prioridad del hilo debería ser de 1 debajo de la clase de prioridad del proceso
`THREAD_PRIORITY_HIGHEST`	La prioridad del hilo debería ser de 2 sobre la clase de prioridad del proceso
`THREAD_PRIORITY_IDLE`	La prioridad del hilo debería ser de la menor de la clase a la que pertenece el proceso
`THREAD_PRIORITY_LOWEST`	La prioridad del hilo debería ser de 2 debajo de la clase de prioridad del proceso
`THREAD_PRIORITY_NORMAL`	La prioridad del hilo debería ser la misma del proceso

Para obtener la prioridad de un proceso se utiliza la función `GetThreadPriority`:

```
int GetThreadPriority(
    HANDLE hThread);
```

El valor que devuelve es uno de los identificadores listados anteriormente o `THREAD_PRIORITY_ERROR_RETURN` si ocurre un error en la obtención de la prioridad.

7.2.3. Suspención y activación de hilos

Cuando se crea un hilo este puede ser iniciado en estado supendido, la llamada al sistema `ResumeThread` permite activar un hilo que fue iniciado suspendido o suspendido por la llamada `SuspendThread`.

```
DWORD ResumeThread(
     HANDLE hThread);
```

El parámetro `hThread` especifica una manija para el hilo a ser restablecido. La función retorna `SUCCESS` si no hay problemas y `0xFFFFFFFF` indicando una falla.

```
DWORD SuspendThread(
     HANDLE hThread);
```

Esta funcion puede llamarla cualquier hilo para suspender a otro hilo. Un hilo suspendido no puede volver activarse.