Nacimiento, ejecución y muerte de un proceso

Dr. Roberto Gómez Cárdenas
ITESM-CEM, Dpto. Ciencias Computacionales

3 de marzo de 1999

El presente documento representa un borrador de lo que será el capítulo de un libro. Acaba de ser terminado y debido a la urgencia de tiempo no se presto mucha atención a la forma, pero el fondo es muy bueno. Cualquier aclaración, comentario o correción se agradecera que se le notifique al autor vía personal o a través del correo electrónico, (rogomez@campus.cem.itesm.mx).

1 Introducción

Al igual que un ser humano un proceso nace, se reproduce (como en el caso de los seres humanos no todos lo hacen) y muere. También al igual que los humanos dos procesos nunca son iguales, cada uno presenta características propias a él y al ambiente en el cual se encuentra.

En este capítulo empezaremos por enumerar las características principales de los procesos, para después explicar las diferentes llamadas de sistema relacionadas con la creación, muerte y ejecución de un proceso.

2 Características de los procesos Unix

Para poder hablar de procesos debemos dar definición de lo que es un *proceso*. Debido a la diversidad de entornos en los cuales se dearrollan los procesos no podemos encontrar una definición universal de la palabra proceso. Deitel, [Deit82], proporciona el siguiente conjunto de posibles definiciones:

- programa en ejecución
- actividad asíncrona
- *espíritu animado* de un procedimiento
- la entidad a la que se asignan los procesadores

sin embargo otra definición un poco más aceptada es la que define a un proceso como la instancia ejecutable de un programa (definición aportada por Tanembaum en [Tane83]). Debido a que este documento trata sobre los procesos dentro del ambiente del sistema operativos UNIX se adpotará dicha definición en lo que resta del documento.

3 Los estados de un proceso

.

Durante la ejecución de un proceso este pasa por diferentes estados, dependiendo del ambiente de ejecución. Estos estados permiten hacer una descripción del proceso. En forma general un proceso puede estar en tres estados:

1. **En ejecución:** el proceso esta utilizando el CPU en ese momento.
2. **Listo:** el proceso dispone de todo lo necesario para entrar en ejecución a exepción del CPU.
3. **Bloqueado:** el proceso no se puede ejecutar hasta que ocurra algún evento.

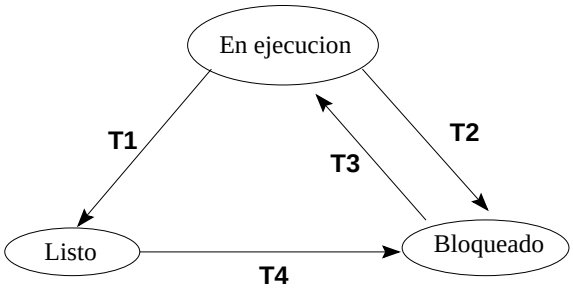

Figura 1: *Los posibles estados de un proceso cualquiera*

Los estados se encuentran ilustrados en la figura 1. Como se puede apreciar en dicha figura se tienen cuatro transiciones:

T1 *de ejecución a bloqueado:* el proceso que se encontraba ejecutando descubre que no puede continuar (p.e. el proceso lee un buffer o de un archivo vacío).

T2 *de en ejecución a listo:* se presenta cuando el "quantum[1]" del proceso ha terminado y cede el CPU a otro que esperaba por él.

T3 *de listo a en ejecución:* es todo lo contrario a la anterior, ya todos los procesos fueron atendidos y ahora es el turno del proceso que estaba en estado *listo*.

T4 *de bloqueado a listo:* la transición tiene lugar cuando el evento esperado ocurre (i.e. los datos que se esperaban llegaron). Si no existe otro proceso en estado de ejecución se produce la tercera transición en forma inmediata y el proceso comienza su ejecución. En caso contrario se queda en estado listo esperando que el CPU este disponible.

El modelo presentado es un modelo teórico-general, el cual varía según el sistema en que se este trabajando. En Unix un proceso puede estar en nueve posibles estados, (ver figura 2). La razón por la cual existen más de estos estados es que se toman en cuenta un mayor número de factores: el modo de ejecución, los recursos, los servicios otorgados, etc.

Los diferentes estados en los que un procesos puede encontrarse en unix son:

1. *usuario en ejecución* (SONPROC) proceso se encuentra ejecutando en modo usuario

[1] El termino quantum designa el tiempo que se le asigna cl CPU a un proceso.

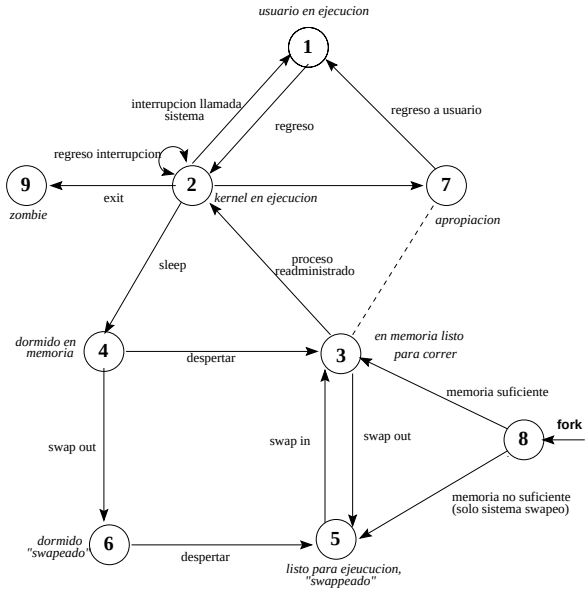

Figura 2: *Los nueve posibles estados de un proceso en Unix*

2. *kernel en ejecución* proceso se encuentra ejecutando en modo kernel
3. *en memoria listo para correr* el proceso no se esta ejecutando pero esta listo para ser ejecutado, una vez que el kernel pueda ejecutarlo.
4. *dormido en memoria* (SSTOP) el proceso esta durmiendo y reside en memoria.
5. *listo para ejecución* (SRUN)proceso esta listo para ser ejecutado, pero el intercambiador (swapper) tiene que cambiarlo a memoria principal para que el kernel lo pueda ejecutar.
6. *dormido* (SSLEEP) el proceso esta durmiendo, y el intercambiador ha cambiado el proceso a un dispositivo de almacenamiento secundario (disco) para poder hacer espacio para otro proceso en memoria principal.
7. *apropiación* (SXBRK) el proceso esta regresando de kernel a modo usuario, pero el kernel hace referencia (preemp) y hace un cambio de contexto para otro proceso
8. *creado* (SIDL) el proceso ha sido creado y esta en transición a un estado; el proceso existe pero no esta listo para correr, y no esta durmiendo.
9. *zombie* (SZOMB) el proceso terminó y el proceso que lo creó no hizo nada por recuperar el estatus de terminación del proceso.

Un proceso solo puede estar en un estado a la vez. Entre parentesis se encuentra el neumonico con que es almacenado dicho estado en la tabla de procesos.

4 Los modos de ejecución de un proceso

Durante la vida de un proceso este ejecuta un código, el cual fue escrito por el usuraio para un propósito específico. Dicho código se encuentra compuesto por palabras reservadas del lenguaje en

que se escribió y por *llamadas de sistema*. Estas llamadas son funciones definidas en bibliotecas específicas, que sirven para indicarle al kernel del sistema operativo que realice determinadas acciones: abrir un archivo, escribir en un archivo, obtener la hora del sistema, etc.

Estas llamadas son "cajas negras" que, en base a los parámetros que se le pasan, realizan una función específica y después informan del éxito o fracaso de lo que se hizo. Si es preciso, regresan algún resultado. La implementación de las llamadas depende del sistema sobre el que se este ejecutando, no es lo mismo la escritura de un archivo en AIX, Linux o Solaris. Dicha implementación solo es conocida y ejecutada por el kernel, el usuario no tiene forma alguna de modificar la forma en que se escribe en un archivo a través de la llamada *write*(), El usuario le indica a la llamada el nombre del archivo, la cantidad a escribir y lo que se va a escribir, la llamada se encarga de realizar todo esto.

Cuando un proceso se encuentra ejecutando código "controlado" por el usuario se dice que esta en *modo usuario*. Si el proceso esta ejecutando instrucciones ocultas al usuario esta en *modo núcleo*, ya que este último es el encargado de ejecutar dichas instrucciones.

Si lo vemos desde el punto de vista de las llamadas de sistema, en el momento en que el proceso ejecuta las intrucciones que definen a dicha llamada el proceso esta en *modo núcleo*. Por otro lado, cuando el usuario utiliza el resultado de esta llamada para otros cálculos el proceso se encuentra en *modo usuario*.

Podemos deducir que durante la vida de un proceso este cambia de modos de ejecución varias veces, dependiendo del número de llamadas de sistema y operaciones exclusivas al núcleo que se realizen. Esto no quiere decir que el usuario puede hacer todo lo que el quiere a través de llamadas de sistema, muchas de ellas requieren de permisos especiales para poder ser ejecutadas.Entre estas encontramos la supresión de las interrupciones, modificación del mapa de memoria, modificación de prioridades de procesos, etc. La figura 3 ejemplifica los dos modos de ejecución.

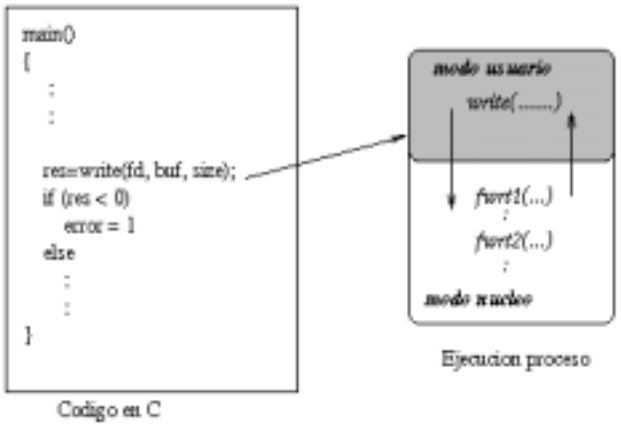

Figura 3: *Los modos de ejecución de un proceso*

5 El contexto de un proceso

En secciones anteriores se listaron los posibles estados en los que se puede encontrar un proceso. Ahora bien el estado de un proceso se define a partir de lo que es el *contexto* de un proceso. El estado

de un proceso depende del código que esta ejecutando, del valor de sus variables que comparte y de la definición de sus estructuras de datos. Debido a esto, y como se puede apreciar en la figura 4, el contexto de un proceso esta formado por tres segmentos:

- el segmento de texto o código
- el segmento de datos
- el segmento del stack

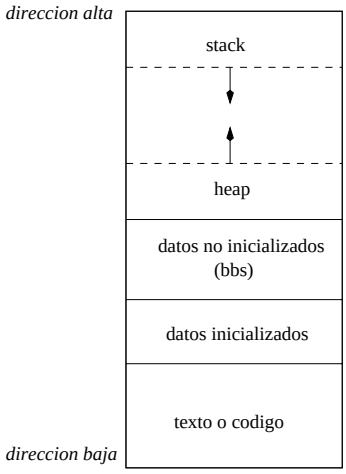

Figura 4: *Diagrama representativo del contexto de un proceso*

El *segmento de texto* contiene las instrucciones máquina a ser ejecutadas por el CPU. Usualmente puede ser compartido, de tal forma que solo una copia necesita estar en momoria para procesos usados por diferentes ususarios. Sin embargo, es frecuente que presente permisos de solo lectura para prevenir que un mismo programa modifique sus instrucciones.

El *segmento de datos* usualmente esta dividido en lo datos inicializados y no inicializados. El primero contiene variables que son especificamente inicializada en el programa, (p.e. $intcont = 0$). La segunda almacena las variables que no fueron inicializadas, y que el sistema las inicializa con cero. Mientras que el primero se conoce como segmento de datos, el último recibe el nombre de segmento *bbs*, (nombre proveniente de de un antiguo operador de ensamblador llamado *block started by symbol*).

El último de los segmentos *segmento stack* contiene toda la información relacionada con el stack, (llamadas funciones, variables locales, etc). También encontramos todas las variables dinámicas, (heap).

5.1 La estructura de un archivo ejecutable y los segmentos de textos compartibles

Diferentes procesos pueden compartir el mismo segmento de texto. Esta característica se encuentra cuando sistemas diferentes o programas de aplicación estándar son usados concurrentemente por diferentes procesos. Un buen ejemplo es el editor *vi*, el cual permite que su código ejecutable pueda ser usado por diferentes usuarios, (quienes simultáneamente están editando diferentes archivos).

Antes de hablar acerca de la compartibilidad de los segmentos de texto de un proceso, es necesario comprender la estructura de un archivo ejecutable.

Es necesario aclarar algunos malentendidos relacionados con la terminología de Unix. Un archivo que solo contienen texto ASCII puede ser "ejecutable" si sus permisos son cambiados. El texto de ASCII en el archivo es interpretado por el shell Unix como líneas de comandos. Todo lo considerado como comando valido es ejecutado. Los comandos que tienen un significado menor para el shell no son ejecutados y se producen errores de mensaje. Resumiendo, este tipo de archivo consiste de un conjunto de instrucciones tipo shell que son interpretadas. El archivo no tiene otra estructura que la de una colección de bytes.

El otro tipo de archivo ejecutable es creado a través de un compilador y un ligador. El resultado de lo anterior es un archivo cuyas instrucciones pueden ser ejecutadas inmediatamente, (ya que se trata de instrucciones tipo máquina). En esta sección, el termino archivo ejecutable designa a un archivo creado a partir de una compilación. Se asume que se trata de la estructura de un archivo nombrado *a.out*. De hecho, los manuales de Unix generalmente se refieren a la descripción de archivos ejecutables, como la descripción de la estructura de un archivo *a.out*. Un archivo ejecutable consite en un encabezado y una o más secciones; el encabezado generalmente incluye lo siguiente:

- El número mágico
- El tamaño del segmento de texto
- El tamaño del segmneto de datos
- Información acerca de la relocalización del código
- El tamaño de la tabla de símbolos para el proceso.

5.1.1 El número mágico

La información acerca de la compartibilidad de un segmento de texto es obtenida del encabezado del archivo ejecutable y es guardada en el valor del *número mágico*. Muchos procesos pueden compartir el segmento de texto de un proceso, y no se hace ningún uso especial de la información de las de las estructuras de kernel, a excepción de las tablas de página usadas para el mapeo entre memoria virtual y memoria física.

Existen diferentes opciones para el número mágico, los mneumónicos que los designan varían de sistema a sistema y estan definidos en el archivo de encabezado */usr/include/a.out.h*. Los más comunes son:

Procesos normales. Dependiendo del sistema el símbolo es denotado por ZMAGIC o A_MAGIC. El mnemónico OMAGIC también es usado frequentemente. Difiere de ZMAGIC en la forma en la que el segmento de datos está alineado para estar contiguo al segmento de texto. Un número mágico ZMAGIC significa que el segmento de datos esta alineado con la frontera del siguiente bloque disponible después del segmento de texto. Un número mágico de OMAGIC permite al segmento de texto y de datos estar contiguos.

Segmento de texto de solo lectura. El símbolo es generalmente denotado por NMAGIC o A_MAGIC2. Necesario si dos o más procesos van a compartir el mismo segmento de texto.

Segmento extensión No usado demasiado en sistemas con una memoria física grande. El símbolo es detonado como A_MAGIC4.

El encabezado de un archivo *a.out* contiene información acerca del tamaño de varias secciones del archivo. Algunos sistemas también incluyen información acerca de la relocalización de un código dentro de un archivo ejecutable, en cuyo caso, el tamaño de la información de relocalización también forma parte de la cabezera.

Notese que la compartibilidad del segmento de texto esta determinada por el número mágico. El número mágico de utilidades como *vi*, *ed*, *ex* y *sed*, los cuales compartan el mismo segmento de texto, pueden ser NMAGIC o A_MAGIC2.

El segmento de datos, como ya se había mencionado antes, contiene dos partes. La primera parte es usada para el almacenamiento de los datos inicializados, la segunda para datos no inicializados. Este método de almacenamiento explica por que la información de un arreglo puede ser alterada por una operación que exceda los límites de este.

5.1.2 La estructura de un archivo *a.out*

Se puede encontrar más información acerca de la estructura de un archivo *a.out* examinando el archivo de encabezado */usr/include/a.out.h*. Con respecto a la estructura de un archivo *a.out* se cuenta con el formato común de archivos objeto, (COFF por sus siglas en inglés). El formato es similar al archivo *a.out*, como se puede apreciar a continuación:

encabezado archivo;

encabezado sistema;

encabezado sección, (uno por sección);

datos;

información de la reubicación;

número de líneas;

tabla de símbolos.

El *encabezado de archivo* contiene el número mágico, número de sección, información sobre el tiempo y la fecha y una tabla de símbolos de información global. El *encabezado de sistema* contiene información como la de un encabezado para un archivo *a.out* así como información de reubicación global. El *segmento de datos* puede ser dividido en dos secciones; la primera sección esta destinada a los datos inicializados, cuya información puede ser proporcionada por el compilador. El tamaño de esta porción es medida en bytes y es denotada por *a_data*. La porción no inicializada del segmento de datos es denotada por *a_bss*, la cual también es medida en bytes. El valor de *a_bss* es inicializado en 0 y cambia siempre y cuando el proceso requiera usar memoria dinámica de la memoria. Algunos programadores se refieren al valor de *a_bss* como el valor de rompimiento (break value). Por último el *encabezado de sección* contiene información acerca de las direcciones físicas y virtuales de la sección, así como apuntadores a datos en ambas formas absolutas y reubicables.

Notese que un archivo objeto con formato COFF no tiene un número mágico. Esto es obvio si se quiere cambiar los permisos de un archivo objeto para poder ejecutarlo. Si depués de realizar lo anterior se teclea el nombre del archivo objeto para ejecutarlo se produce el siguiente mensaje de error:

<archivo objeto>: bad magic number

Un detalle importante es que el archivo objeto no contiene ninguna información acerca del código relocalizable. Así pues, a menos que el código sea autocontenido y no necesite ningún apoyo del sistema operativo o de alguna librería (lo cual significa que el código es trivial) el código objeto no es directamente ejecutable. Es el ligador *ld* el encargado de combinar diferentes archivos objetos para crear un archivo ejecutable.

La importancia de un formato estándar como el COFF es que permite ligar dinámicamente librerías de un código objeto con los ambientes de ejecución de un proceso. En algunos sistemas Unix, COFF ha sido substituido por otro estándar llamado ELF (Extensible Linking Format). La facilidad para ligar dinámicamente el código objeto no esta presente en todos los sistema Unix, debido a que el "ligado dinámico" del código objeto puede reducir el tamaño de los programas ejecutables.

5.1.3 Contenido archivo *a.out*

Ahora seguiremos con la discusión de los archivos *a.out*. A primera vista, la información de un archivo *a.out* parece estar oculta al usuario, ya que ese archivo tiene muchos caracteres que no se pueden imprimir. Un listado directo, vía el comando *cat*, produce confusión y puede provocar un bloqueo de la terminal. Sin embargo es posible ver el contenido usando el comando *od* (octal dump). Cuando se usa con la opción $-c$. este comando nos permite ver los contenidos del archivo, byte por byte.

El contenido de un archivo se encuentra en formato ASCII. Auxiliandose de pipes, para enviar la salida del comando *od* a un comando *more* o *pg* podremos ver el contenido pantalla por pantalla. De esta forma el comando *od* nos permite ver el número mágico directamente. Es posible usar este comando para ver el contenido de algunos archivos ejecutables en los directorios */bin* y */usr/bin/*.

En la figura 5 se presenta un ejemplo de como se puede visualizar el número mágico de un archivo objeto. Se presenta el archivo ejecutable obtenido de la compilación del programa C más simple. Este crea un archivo *a.out* cuya salida en formato ASCII se muestra en el recuadro.

```
$cat toto.c
main ()
{
}
$cc toto.c
$od -c a.out | more
0000000  001   p   004    +    206    y    x    203
0000020   034   p   003   001   013   002   ,   002   260
0000040   020   200   200   330   200   200   320   200   210  002   374
0000060   .   t   e   x   t   200   200   320   200   200   320
0000100   002   ,   320
0000120          .   d   a   t   a
0000140   200   210   002   374   200   210   002   374   002   260   002
0000160                                                              @
0000200   .   b   s   200   210   005   254   200   210   005   254
0000220           020
0000240   200   .   c   o   m   m   e   n   t
0000260   003   314     005   254
^C
$
```

Figura 5: *Ejemplo de visualización de un número mágico*

En el ejemplo el número mágico es 1, cuyo equivalente en octal es el ASCII 001. Notese el principio del segmento de datos. Los números en la columna izquierda son direcciones relativas dadas en octal. El archivo ha sido editado para mayor claridad, y los caracteres especiales que no son normalmente imprimibles han sido omitidos.

La utilidad `nm` también puede ser usada para examinar una porción del proceso que es obtenido de un archivo ejecutable: la tabla de símbolos. Esta es invocada usando la siguiente sintáxis:

```
nm nombre_archvioble_ejecutable
```

el cual produce un listado de la tabla de símbolos del proceso. Diferentes sistemas generan diferentes formatos de salida para esta utilidad. Para mayor información se puede utilizar el comando *man* del sistema donde se este trabajando.

6 La creación de un proceso

Antes de comenzar a hablar sobre la creación de procesos, nos adelantaremos un poco y comentaremos algo sobre la estructura donde se almacena la información de todos los procesos que participan en el sistema. Esta estructura se conoce como la *tabla de procesos*. Cada proceso cuenta con una entrada en la tabla de procesos, y tiene asignada un área U (U de usuario). La tabla de procesos apunta a una tabla de regiones por procesos, la cual permite a procesos independientes compartir código común. Todo esto se encuentra ilustrado en la figura 6.

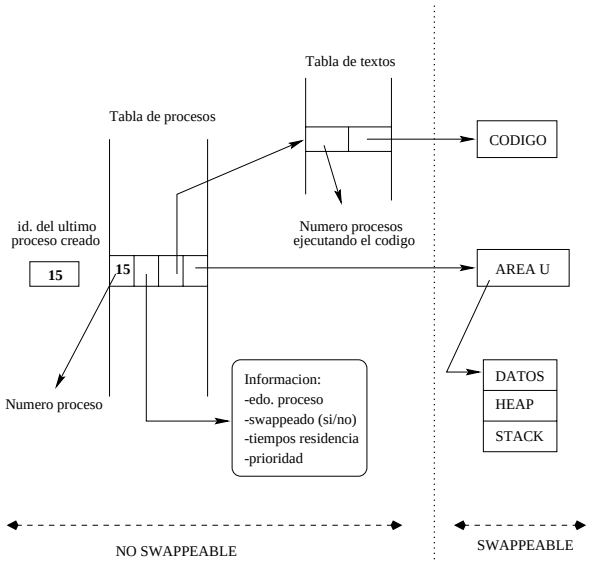

Figura 6: *Estructura datos antes creación proceso*

Todos los procesos en Unix se crean a partir de la llamada de sistema $fork()$, excepto por los procesos sistema que son creados en el arranque del sistema. Cuya sintáxis se presenta a continuación:

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

int fork();
```

El proceso que ejecuta la función es el proceso padre y el proceso creado se considera proceso hijo. Es llamada una vez (por el padre) pero regresa dos veces (una en el padre y otra en el hijo).

Al padre le regresa el identificador del proceso hijo, y al hijo le regresa el valor de cero. Si algo sale mal regresa un -1.

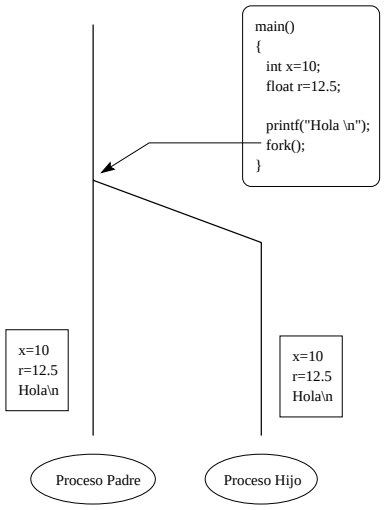

Figura 7: *La creación de un proceso a partir de fork()*

Cuando un proceso es creado, el proceso hijo copia todo el contexto de lo que es el proceso padre (ver figura 7) esto incluye las variables el heap, el stack y el código. El proceso hijo tendra su propia versión de las variables, heap y stack pero ejecuta el mismo código que el padre. A partir de donde se encuentre la llamada $fork()$ el proceso padre y el hijo ejecutarán concurrentemente (al mismo tiempo) todas las líneas del código.

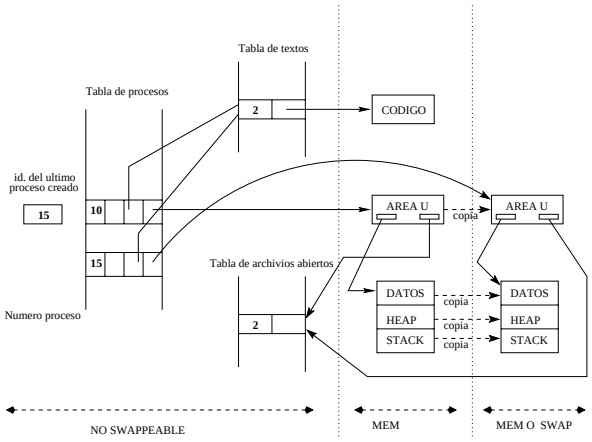

Figura 8: *Estructura datos después creación proceso*

El kernel del sistema operativo es el encargado de realizar lo anterior. En la figura 8 se ilustra una tabla de procesos que contiene un solo proceso con identificador 10. Una vez que este proceso manda llamar $fork()$ se crea un nuevo proceso con identificador 15. Los dos comparten el mismo segmento de texto (i.e. ejecutan lo mismo). Sin embargo los otros segmentos que componen al contexto del proceso se duplican y cada uno maneja su propio stack, heap y datos. Esto trae como consecuencia que cada proceso maneje su propia "copia" de variables.

El valor de regreso de la llamda $fork()$ es usado para que el programador pueda definir que parte del código debe ejecutar el hijo y que parte el padre. Ahora bien si se necesita conocer el identificador de un proceso se cuenta con las funciones *getpid*() y *getppid*(). La primera regresa el valor del identificador del proceso que la mando llamar, y la segunda la del padre del proceso que lo mando llamar. La sintáxis de las dos llamadas es:

```
#include <unistd.h>

int getpid();
int getppid();
```

6.1 Ejemplo uso $fork()$

El programa de la figura 9, ilustra el uso de la llamada de sistema $fork()$. Se presenta el programa así como el resultado de varias ejecuciones del mismo.

```
#include <stdio.h>
main()
{
  int i,m,n;

  if ((n=fork()) == -1) {
   fprintf(stderr,"Error en la creacion de procesos \n");
   exit(1);
  }
  else
    if (n == 0)   /* Es el hijo */
      for (i=0; i<3; i++)
        printf("\t\t\t Proceso  HIJO con id: %d, valor i:%d\n",getpid(),i);
    else    /* Es el padre*/
      for (i=0; i<3; i++)
        printf("Proceso PADRE con id: %d, valor i:%d\n",getpid(),i);
}
```

Figura 9: *Ejemplo de uso de la llamada fork()*

Este programa crea un proceso hijo a través de la llamada $fork()$. Como toda llamada de sistema, si regresa -1 el programa despliega un mensaje de eror y aborta el programa. Para diferenciar el código a ejecutar por el padre y el hijo, se compara el regreso de la llamada $fork()$. En este ejemplo tanto el proceso hijo como el padre realizan una iteración dentro de la cual despliega su identificador y el valor de un contador. Para diferenciar mejor la salida del proceso padre y la del proceso hijo se uso un tabulador dentro del proceso hijo.

Algunas posibles salidas de la ejecución de dicho programa se presentan en la figura 10. Esta salida ejemplifica la forma en que el hijo ejecuta una copia del contexto del proceso padre. Como puede apreciarse cada proceso maneja su propio valor de la variable i. Los dos procesos imprimen su identificador y el valor de esta variable dentro del ciclo.

Uno de los detalles que nos permite constatar este programa es el intercambio del CPU entre los procesos. No importa en que instrucción se este ejecutando cuando se le termina el quantum a un proceso, se le asigna el CPU a otro proceso para que inicie su ejecución, o continuarla si se lo habían quitado antes. Cabe la pena aclarar que las salidas presentadas se obtuvieron después de ejecutar varias veces el mismo programa.

```
rogomez@armagnac:26:>cc padhijo3.c -o padhijo3
rogomez@armagnac:27:>padhijo3
                        Proceso  HIJO con id: 1923, valor i:0
Proceso PADRE con id: 1922, valor i:0
Proceso PADRE con id: 1922, valor i:1
Proceso PADRE con id: 1922, valor i:2
rogomez@armagnac:28:>                    Proceso  HIJO con id: 1923, valor i:1
                        Proceso  HIJO con id: 1923, valor i:2
rogomez@armagnac:29:>padhijo3
                        Proceso  HIJO con id: 1929, valor i:0
                        Proceso  HIJO con id: 1929, valor i:1
                        Proceso  HIJO con id: 1929, valor i:2
Proceso PADRE con id: 1928, valor i:0
Proceso PADRE con id: 1928, valor i:1
Proceso PADRE con id: 1928, valor i:2
rogomez@armagnac:30:>padhijo3
Proceso PADRE con id: 1930, valor i:0
Proceso PADRE con id: 1930, valor i:1
Proceso PADRE con id: 1930, valor i:2
                        Proceso  HIJO con id: 1931, valor i:0
                        Proceso  HIJO con id: 1931, valor i:1
                        Proceso  HIJO con id: 1931, valor i:2
rogomez@armagnac:31:>
```

Figura 10: *Posibles salidas del programa ejemplo de uso de la llamada fork()*

6.2 Sincronización del proceso padre e hijo

Otro aspecto a considerar en el ejemplo anterior es quien termina de ejecutar primero: el padre o el hijo. Como se puede constatar nada garantiza que la ejecución del proceso hijo se hará o terminará primero que la del padre. En la primera ejecución del ejemplo anterior el proceso hijo empezó primero y terminó al último, en la segunda ejecución empezó el proceso padre y terminó el proceso hijo; en la última ejecución empezó el proceso padre y es el proceso hijo el que termina.

Si se requiere sincronizar la ejecución del padre y lo del hijo, es decir que el padre ejecute cierta parte del código cuando el hijo termine de ejecutar lo suyo, debe especificarsele al padre que *espere* a que su hijo termine. Esto último se realiza a través de la llamada de sistema *wait*(). Insertando dicha llamada en el código del padre este no hará nada hasta que su hijo haya terminado su ejecución. La sintáxis de la llamada se presenta a continuación:

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

pid_t wait(int *stat_loc);
```

Esta llamada provoca la suspensión del proceso hasta que uno de sus procesos hijos termine su ejecución. También puede ser utilizado para esperar un evento (señales). Si un proceso hijo termina antes de que el padre ejecute un *wait*(), este proceso pasa a lo que es un estado *zombie*). El regreso del *wait*() es inmediato y proporciona la siguiente información:

1. identificador del proceso a quien esperaba el proceso;
2. código de terminación del proceso esperado (parámetro $*n$);

Si no hay procesos hijos entonces el regresa un valor de -1. Si n es diferente de 0 el valor $*n$ proporciona información sobre el tipo de terminación del hijo. Más adelante se verá la forma en la que se recupera dicha información. El programa de la figura 11 presenta un programa en el cual se presenta un ejemplo de uso de *wait*(). A diferencia del programa anterior para diferenciar el código que va a ejecutar el padre y el que va a ejecutar el hijo, se compara directamente el regreso de la llamada $fork$, si es cero se trata del hijo, sino es el padre.

```
main()
{
  if (fork() == 0)
    printf("Hijo \n");
  else {
    printf("Padre \n");
    wait();
    printf("Ya termino el hijo \n");
  }
}
```

Figura 11: *Ejemplo de uso de la llamada wait()*

En la figura 12 se presenta la salida de la ejecución del programa anterior. Como se puede constatar el mensaje *Ya termino el hijo* es el último en desplegarse en todas las ejecuciones. Esto se debe a que es la instrucción que sigue a la llamada *wait*() que ejecuta el padre, lo que provoca que el padre cree el hijo, despliege *Padre*, espere que el hijo termine su ejecución e imprima el mensaje.

```
rogomez@rgomez:224:>cc exwait1.c -o exwait
rogomez@rgomez:225:>exwait1
Hijo
Padre
Ya termino el hijo
rogomez@rgomez:226:>exwait1
Hijo
Padre
Ya termino el hijo
rogomez@rgomez:227>exwait1
Padre
Hijo
Ya termino el hijo
rogomez@rgomez:228:>exwait1
Padre
Hijo
Ya termino el hijo
rogomez@rgomez:229:>
```

Figura 12: *Posibles salidas de la ejecución del programa ejemplo de wait()*

6.3 Esperando por un hijo en particular

Ahora bien si el padre creó varios hijos y desea esperar por uno en especial, puede hacer uso de la llamada *waitpid*(), en la cual especifica el identificador del proceso que espera que termine, y también se puede obtener el status de dicho proceso usando uno de los parámetros.

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

int waitpid(int pid, int *stat_loc, int options):
```

Esta llamada proporciona un método más flexible para esperar hijos. Un proceso puede esperar por un hijo en particular sin esperar por todos los hijos. Esta llamada proporciona una forma no bloqueante, por lo que un proceso puede checar periodicamente por un hijo sin bloquearse indefinidamente. Recibe tres parámetros:

1. *pid* identificador del proceso que se espera;
2. *stat_loc* apuntador para el regreso del status de terminación del proceso esperado;
3. *options* banderas

Si *pid* es -1 entonces el proceso espera por cualquier proceso, por otro lado si *pid* es mayor que cero entonces espera por el identificador del proceso identificado por *pid*. Si *pid* es cero espera por cualquier proceso cuyo identificador de grupo sea igual al proceso que realiza la llamada y si *pid* es menor a -1 se espera por cualquier proceso cuyo identificador de grupo sea igual al valor absoluto de *pid*.

El status de terminación del proceso hijo esperado es almacenado en la variable *stat_loc*. La bandera *options* es utilizado para casos especiales. Con la bandera asignada a WHOHANG la llamada se vuelve no bloqueante y regresa inmediatamente (con un valor de 0) si el proceso hijo no ha terminado. Otros posibles valores de bandera son WCONTINUED, WNOWAIT y WUNTRACED.

El regreso de *wait*() y *waitpid*() depende de la forma en que termino el hijo. Si *wait*() regresa debido a la terminación un hijo el valor de regreso es positivo y es el identificador del proceso del hijo, sino *wait*() regresa -1 y asigna un número a *errno*. Si *errno* es ECHILD entonces no hay hijos que esperar, por otro lado si *errno* es EINTR la llamada fue interrumpida por una señal.

```
main()
{
  int status
  pid_t waitreturnpid;

            :
        fork();
            :

   while (waitreturnpid =waitpid(-1, &status, WHONANG))
     if ( (waitreturnpid == -1) && (errno != ENTR) )
          break;
```

Figura 13: *Ejemplo de uso de waitpid() como wait()*

La llamada de sistema *waitpid*() puede comportarse como una llamada *wait*() con ciertos parámetros. Un ejemplo de esto se presenta en el código de la figura 13. En dicho código, independientemente de las instrucciones anteriores a la llamada *waitpid*() el proceso va a esperar la terminación de cualquier proceso debido al parámetro -1 de *waitpid*(), sin embargo el parámetro WHONANG hace que no se bloquee y que regrese cada vez que un proceso termine.

El código presentado en la figura 14 permite ejemplicar la ventaja de utilizar *waitpid*() en lugar de *wait*(). El programa crea tres procesos y los envia a dormir un determinado tiempo. El proceso padre, espera que uno de los tres termine, en este caso espera al segundo proceso. Una vez que el segundo proceso termina este despliega un mensaje. Si se quisiera esperar a cualquier otro proceso basta con cambiar el primier parámetro de *waitpid*() a *uno* o *tres* según sea el caso.

```
main()
{

  int uno, dos, tres;  /* identificadores procesos hijos */
  int retval, tmp;     /* valor regreso de la llamada y tiempo a dormir */
  int *status;         /* status del proceso hijo */

  if ( (uno = fork()) == 0) {
     tmp = (random() % 10);
     printf("Proceso uno (%d) a dormir %d segs. \n",getpid(),tmp);
     sleep(tmp);
     printf("Proceso uno termino \n");
  }
  else
    if ( (dos = fork()) == 0) {
       tmp = (random() % 10);
       printf("\tProceso dos (%d) a dormir %d segs. \n",getpid(),tmp);
       sleep(tmp);
       printf("\tProceso dos termino\n");
    }
    else
       if ( (tres = fork()) == 0) {
         tmp = (random() % 10);
         printf("\t\tProceso tres (%d) a dormir %d segs. \n",getpid(),tmp);
         sleep(tmp);
         printf("\t\tProceso tres termino\n");
       }
       else {
         printf("\t\t\tEsperando que el proceso dos termine \n");
         retval = waitpid(dos,&status,0);
         printf("\t\t\tProceso dos (%d) termino con estatus: %d \n",retval,status);
       }
}
```

Figura 14: *Ejemplo de uso de la llamada waitpid()*

6.4 Aplicaciones de la creación de procesos

Las características que presenta el crear un proceso que hereda el contexto de su padre, hace que sea útil para cualquier de los siguientes casos:

1. un proceso desea hacer copia de si mismo; mientras que el proceso realiza una operación, la copia hace otra tarea (p.e. Servidores Red).
2. un proceso quiere ejecutar otro programa proceso crea un proceso y una de las copias ejecuta lo que se quiere (p.e. shells).

7 Ejecución de un proceso

Siguiendo con la analogía procesos/seres humanos, un proceso puede hacer que el proceso hijo realize otras tareas. Un proceso ejecuta el código definido en su segmento de texto. Sin embargo es posible que ejecute otro código, esto es posible a través de alguna de las variantes de la llamada *exec*(). Esta llamada provoca que el proceso substituya "la imagen de ejecución" con la de un nuevo programa.

Lo anterior no tiene sentido si se aplica literalmente, lo que en realidad se persigue es que el hijo ejecute un código diferente al del contexto que heredó de su padre.

La llamada *exec*() presenta las siguientes características:

- el proceso que la ejecuta pierde el contexto que tenía asociado con el antiguo programa (código, datos stack y heap);
- el proceso no pierde su pid, ni su padre y continúa con su papel de hijo;
- el proceso retiene algunas partes de su contexto (p.e. todos los descriptores de los archivos abiertos);

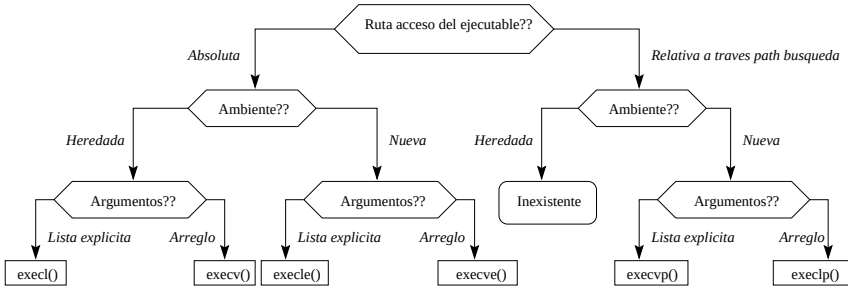

Figura 15: *Diagrama de decisión para elegir el exec()*

No se puede proporcionar una sintaxis de *exec*() ya que se cuenta con seis diferentes versiones de esta llamada. Estas versiones difieren entre sí por la forma en que se especifica el archivo a ejecutar, los parámertros del archivo a ejecutar y el ambiente en el cual se va a ejecutar (ver sección 7.2.1). En la figura 15 se presenta un diagrama que presenta las diferentes variantes y la llamada a utilizar de acuerdo a las varientes elegidas. Las posibles combinaciones varian de acuerdo a:

1. la especificación del nombre del programa a ejecutar:
   - se puede especificar con un path absoluto;
   - puede especificarse a partir de un path relativo,
2. el ambiente de ejecución:
   - se puede tratar de un ambiente nuevo:
   - se trata del mismo ambiente que se tenía
3. el paso de argumentos del programa a ejecutar
   - se puede pasar en forma de lista expliícita, terminada con un 0
   - también se puede pasar a través de un vector o arreglo

Por razones de sencillez de uso se prefirió utilizar la llamada *execlp*() para mostrar el uso de las diferentes versiones de *exec*(). Por ejemplo, si se desea ejecutar el comando sort con argumentos *-n* y *foo* la llamada sería:

```
execlp ("sort", "sort", "n", "foo", 0);
```

Vale la pena resaltar que la llamada tiene un número variable de parámetros; por lo que el argumento 0 es necesario para marcar el final de la lista. El argumento *sort* es el nombre del programa a ejecutar y el el segundo *sort* es el primer parámetro del comando sort (ver sección 7.2).

La figura 16 representa un ejemplo de un programa que utiliza la llamada *exec*(). A través de este programa un usuario selecciona uno de tres comandos (`who`, `ls` y `date`) y lo ejeucta usando la llamada *execlp()*.

```
/* Programa que imprime un menu, y ejecuta el comando seleccionado */

#include <stdio.h>

main ()
{
  /* lista comandos */
  static char *cmd []= {"who", "ls", "date"};
  int i;

  printf("0=who, 1=1s, 2= date: ");
  scanf("%d", i);
  execlp(cmd [i], cmd [i], 0);
  printf("No se encontro el comando \n");
}
```

Figura 16: *Ejemplo de uso de la llamada exec()*

Este programa ejemplifica una de las características más importantes de la llamada *execlp*(); esto es que el proceso cambia completamente de contexto al realizar la llamada. El código anterior solo ejecuta un comando a la vez, si se quisiera ejecutar otro comando se tiene que volver a ejecutar el programa. No se puede utilizar un ciclo, ya que *execlp*() no regresa (a menos que produzca un error). Si *execlp*() termina con éxito el control es transferido al nuevo programa lo que trae como consecuencia que todo el contexto del programa viejo se pierda. Para evitar lo anterior se debe mezclar el uso de las llamadas $fork()$ y $wait()$. El $fork()$ crea un hijo el cual debe realizar el *execlp*(), de esta forma el hijo ejecuta lo que se pidió, el padre espera al hijo con una llamada $wait()$ y después de terminado el comando vuelve a preguntar. De hecho este es el prinicipio básico del funcionamiento de un shell.

7.1 El resto de las llamadas *exec*()

Como se dijo anteriormente existen otras sintaxis de *exec*(). Las sintáxis de las otras llamadas *exec()* son:

```
int execl(path, arg0, arg1, ..., argn,(char)*0)
char *path, *arg0, *arg1, ..., *argn;
```

```
int execv(path, argv)
char *path, *argv[];

int execle(path, arg0, arg1, ..., argn, (char)*0, envp)
char *path, *arg0, *argl, ...., *argn, *envp[];

int execve(path, argv, envp)
char *path, *argv[], *envp[];

int execlp(file, arg0, arg1, ..., argn, (char)*0)
char *file, *arg0, *arg1, ..., *argn;

int execvp(file, argv)
char *file, *argv[];
```

En cada una de ellas *envp* representa un arreglo de apuntadores de caracteres que constituyen el ambiente en el que se va a ejecutar el nuevo programa. Dicha variable termina con un apuntador a *NULL*. El parámetro *file* representa el nombre del archivo a ejecutar en el caso de un direccionamiento relativo, *path* es en el caso de que se decida pasar todo el path para direccionar el archivo a ejecutar.

Los parámetros *arg*0, *arg*1, hasta *argn* representan los argumentos que el nuevo código va a recibir en caso de que se desean pasar en forma de lista explícita; si se desean pasar dentro de un vercot se puede utilizar el parámetro *argv*[].

7.1.1 Ejemplo uso de la llamada fork(), execlp() y wait()

La combinación de las llamadas anteriores puede ser útil para un sinúmero de aplicaciones. En realidad son muy utilizadas en el ambiente de los sistemas operativos, principalmente en el diseño e implementación de shells.

Un shell sirve de interfaz entre el usuario y el núcleo del sistema operativo. A grandes rasgos, un shell lee un comando (ya sea a través de un ratón o de la línea de comandos), interpreta lo que desea el usuario y se lo solicita el núcleo. Una vez que el núcleo realizó lo solicitado el shell informa al usuario el resultado de lo que este pidió. Esto puede ser un desplegado de error o un mensaje de que todo salió bien[2].

En Unix existen diferentes tipos de shell y todos leen los comandos a ejecutar de la línea de comandos. Así mismo todos se construyen bajo el mismo principio; el proceso padre es el shell, este lee un comando de la entrada estándar, crea un hijo para que ejecute el comando leído y espera que este termine. Cuando el hijo termina de ejecutar lo solicitado el shell vuelve a preguntar por otro comando. Si se presenta algún error durante todo el proceso anterior no se ejecuta lo solicitado y se despliega un mensaje. Entre los posibles errores estan que no se puedan crear más procesos o que el comando a ejecutar no exista.

En la figura 17 se presenta un programa que lee el nombre de un comando de la línea de comandos, crea un hijo ejecuta para que ejecute el comando y espera por él. A diferencia de un verdadero shell este programa no vuelve a preguntar por otro comando. Cuando se crea un proceso se almacena el identificador de este en la variable `chilpid`. Al crear el proceso verifica que no ocurra ningún error y ejecuta el comando pasado por la línea de comandos. Cabe la pena hacer notar que solo se ejecutan comandos que no contengan ningún parámetro. El proceso padre espera que sea exactamente el hijo

[2] En este sentido Unix es especial, ya que nunca indica que algo salió bien, solo despliega un mensaje de error en caso de que el comando no haya podido ejecutarse.

que creó el que termine. Esto lo hace comparando el regreso de la llamada *wait*() con el identificador del hijo. La condición de la parte de abajo verifica que la llamada *wait*() no sea interrumpida por una señal ajena al proceso (p.e. un `<ctrl-c>`).

```
rogomez@armagnac:162>cat toto.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

main (int argc, char *argv[])
{
  int childpid, tempo;
  int status;

  if ( (childpid = fork()) == -1) {
     fprintf(stderr, "No mas procesos disponibles\n");
     exit(1);
  }
  else
   if (childpid == 0) {
       if ( execlp(argv[1], argv[1], 0) < 0){
           fprintf(stderr,"Comando %s no conocido \n", argv[1]);
           exit(1);
       }
   }
   else
      while ( childpid != (tempo = wait(&status)) )
        if ( (tempo == -1) && (errno != EINTR) )
            break;
   exit(0);

}
rogomez@armagnac:163>cc toto.c -o toto
rogomez@armagnac:164>toto who
rogomez    console      Mar  2 10:53 (:0)
rogomez    pts/2        Mar  2 10:54
rogomez    pts/1        Mar  2 10:54
rogomez    pts/3        Mar  2 10:54
rogomez@armagnac:165>toto date
Wed Mar  3 18:47:16 CST 1999
rogomez@armagnac:166>
```

Figura 17: *Código base de un shell y ejemplo ejecución*

7.2 Los argumentos a nivel línea de comandos

En Unix un comando recibe sus parámetros a través de la línea de comandos. Muchos de los programas a desarrollar en un ambiente unix requiren que sus parámetros de entrada sean pasados a través de la línea de comandos. El lenguaje C proporciona un medio para poder llevar a cabo lo anterior. La función principal *main*() puede recibir dos argumentos: el número de argumentos a pasar en la línea de comandos y los argumentos en sí. Dichos argumentos se conocen con el nombre de *argc* y *argv*[] respectivamente.

Entre los detalles a cuidar esta el hecho de que el nombre del programa es un argumento en sí y de que la numeración de estos empieza en cero. El código de la figura 18 imprime todos los argumentos pasados al programa a través de la línea de comandos.

```
/* Programa que despliega los argumentos de la linea de comandos */

main(int argc; char *argv[])
{
   int i;

   printf("Impresion argumentos: \n");

   for(i=0; i<argc; i++)
     printf(" Argumento %d: %s \n",i,argv[i]);

}
```

Figura 18: *Ejemplo uso parámetros argc y argv*

Por último, hay que recordar que los argumentos son de tipo cadena de caracteres. Esto implica que si se requiere algun parámetro de tipo entero o flotante es necesario convertir el parámetro al tipo deseado antes de utilizarlo. Esto se puede hacer a tráves de las llamadas *atoi*() y/o *atof*().

7.2.1 Las lista de variables de ambiente

En Unix existe lo que se conoce como el ambiente del sistema. Esta ambiente define diferentes propiedades del sistema como el tipo de terminal, las salidas, el lugar donde se encuentran los programas, el directorio hogar, el prompt, etc. El ambiente se encuentra definido por lo que se conoce con el nombre de *variables de ambiente*.

Históricamente la mayoría de los sistemas Unix han proporcionado un tercer argumento a la función main que es la dirección de la lista de ambiente. Esto trae como consecuencia que la sintaxis de la función main sea:

```
int main (int argc, char *argv[], char *envp[])
```

ANSI C especifica que *main*() debe contar con dos parámetros y como este tercer argumento no proporciona un beneficio sobre la variable global *environ*, POSIX.1 especifica que *environ* puede usarse en lugar de un posible tercer argumento.

La dirección del arreglo de apuntadores esta almacenada en la variable global *environ* con la siguiente declaración:

```
extern char *environ;
```

El acceso a variables de ambiente específicas puede realizarse normalemente a través de las funciones *getenv()* y *putenv()* cuya sintaxis es:

```
char *getenv(const char *nombre);
int putenv(const char *string);
```

La primera llamada, *getenv*() busca en la lista de variables de ambiente un string de la forma "nombre=valor" y una vez que lo encuentra despliega el *valor* asociado al *nombre*. Por el otro lado,

putenv() asigna un valor a una variable de ambiente creando la variable o modificando el valor de una variable ya existente. El *string* debe ser de la forma "nombre=valor" (sin espacios).

En la figura 19 se presenta un ejemplo de como se puede representar un ambiente consistente de cinco strings.

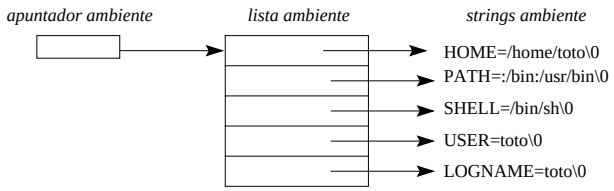

Figura 19: *Ejemplo de variables de ambiente de un proceso*

8 La muerte de un proceso

Al igual que un ser humano, lo último que hace un proceso es morir. La figura 20 se muestra un diagrama que resume el inicio y la terminación de un proceso Unix.

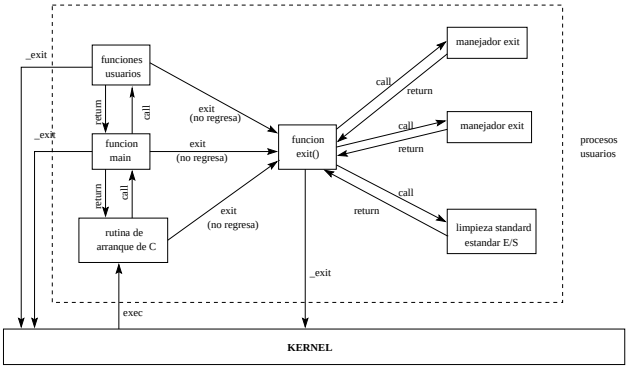

Figura 20: *Arranque y terminación de un proceso Unix*

En esta figura podemos apreciar todo las etapas por las que pasa la ejecución de un programa escrito en C. Muchos compiladores de C se las arreglan para que se llame una función especial de arranque ($fsp()$) cuando el programa es ejecutado. La función $fsp()$ se encarga de inicializar lo necesario y después llamar a la función *main*(). Cuando esta función termina o se "sale", otra función proporcionada por $fsp()$ es invocada. La forma de salirse es a través de la llamada *exit*(), la cual vacia los buffers de entrada/salida y termina llamando a la función del sistema operativo *_exit()*, que realiza la salida del programa y regresa directamente al kernel[3].

Existen cinco formas en las que un proceso puede terminar:

1. el proceso regresa de la función *main()*

[3] Tanto *exit()* como *_exit()* son llamadas de sistemas que pueden ser usadas por el usuario, la primera realiza algo de limpieza y regresa al kernel, mientras que la segunda regresa directamente al núcleo

2. el proceso ejecución la llamada *exit()*
3. el proceso realizó un *_exit* (ejecución de la última instrucción de la función *main*())
4. en alguna parte se ejecutó la función *abort*
5. el proceso recibión una señal que provocó su terminación, ya sea por parte de otro proceso, del núcleo o por el usuario.

puede decirse que las tres primeras representan una forma de terminación normal y que las dos últimas una terminación anormal.

8.1 La llamada *exit()*

Como se dijo en la sección pasada la llamada *exit*() realiza una cierta limpieza (cierra todos los streams abiertos y vacía todos los streams de salida) antes de terminar con la ejecución, y por ende vida, de un proceso. Esta llamada presenta la siguiente sintaxis:

```
#include <stdlib.h>

void exit(int status)
```

La llamada *exit*() nunca regresa al proceso que la invocó ya que regresa al kernel. La llamada envia una señal al padre (SIGCHLD) del proceso que la llamó para indicarle que terminó. Así mismo, un estatus de tipo entero es pasado por el proceso al kernel. Este estatus es el valor del parámetro `status` de la llamada. Dicho estatus puede ser recuperado por el proceso padre a través de la llamada *wait*(). Sin embargo solo la parte baja de dicho estatus puede ser usada.

Para poder explicar como se recupera dicho estatus debemos recordar la sintaxis de la llamada *wait*():

```
int wait(int n)
```

En la sección 6.2 la llamada era utilizada para sincronizar la ejecución de un proceso padre y un proceso hijo. El regreso de *wait*() es inmediato, y permite obtener el número de proceso a quien esperaba el proceso que la mando llamar (valor de regreso de la función) y el estatus de terminación del proceso al que se esperaba (a través del parámetro n). Si no hay ningún proceso al cual esperar regresa un -1. La forma de interpretar n varía dependiendo de la causa por la cual el proceso terminó:

- si la ejecución del proceso hijo fue interrumpida y abortada por alguna causa ajena al usuario, los ocho bits más significativos de n contienen el número de la señal enviada por el núcleo que causó que el proceso se abortó; y la parte baja sera igual a WSTOPFLG, (i.e. los 16 bits serán el número de señal más 128).
- si el proceso terminó debido a una llamada *exit*() los ocho bits más bajos de n tienen un valor de cero, y los ocho más significativos contienen el valor pasado como parámetro a *exit*()
- por último, si el usuario decidió abortar el proceso a través de una señal (p.e. teclear ctr-c) los ocho bits más significativos tienen un valor de cero y los menos significativos representa el valor de la señal que hizó que el proceso abortara.

Ahora bien, supongamos que un proceso *a* ejecuta un *exit*(). Si el padre de *a* no esta ejecutando un *wait*() o un *waitpid*() y no ha activado la bandera SA_NOCLDWAIT o ignora la señal SIGCHLD, entonces el proceso *a* es transformado en un proceso *zombie*. Un proceso *zombie* es un proceso inactivo que será borrado cuando el propietario del proceso salga del sistema, o cuando el padre del proceso ejecute un *wait*() o un *waitpid*(). Un proceso *zombie* solo ocupa un lugar en la tabla de procesos, y no tiene recursos asignados a nivel usuario ni a nivel núcleo.

8.1.1 Ejemplo recuperación estatus de un proceso que termino normalmente

La figura 21 presenta un código que nos permite ejemplificar lo expuesto anteriormente. En este caso un proceso crea un hijo el cual despliega un mensaje y después ejecuta un *exit*(3). Por el otro lado, el padre espera que el hijo termine su ejecución y despliega los valores regresados por la llamada *wait*(). En la parte de abajo de la figura se presenta un ejemplo de ejecución, la función *wait*() regresó el número de proceso del hijo y el número 768. El equivalente binario de 768 es 00000011 00000000 donde los bits más significativos (00000011) representan el número 3 que es el parámetro de la llamada *exit*().

```
rogomez@cognac:221>cat toto.c
main()
{
   int m, n:

   if ( fork() == 0 ) {
      printf("Proceso hijo con identificador %d \n",getpid());
      exit(3);
   }
   else {
     m = wait(&n):
     printf("Padre: fin proceso %d con estatus: %d \n",m,n);
  }
}
rogomez@cognac:222>cc toto.c -o toto
rogomez@cognac:223>toto
Proceso hijo con identificador 21322
Padre: fin proceso 21322 con estatus: 768
rogomez@cognac:224>
```

Figura 21: *Ejemplo terminación normal de un proceso*

8.1.2 Ejemplo recuperación estatus de un proceso con terminación anormal

La figura 22 presenta un ejemplo de una terminación debido a una interrupción por parte del usuario. En este caso el proceso padre crea un hijo el cual despliega su identificador y ejecuta un iteración infinita. El proceso padre ejecuta un *wait*() esperando que el hijo termine, para después desplegar los valores regresados por la llamada *wait*().

Un proceso puede abortar debido a que el usuario así lo deseó (a través de un ctrl-c), o por que el kernel detectó una operación no válida y aborta la ejecución. En el último de los casos el kernel genera un archivo de nombre *core* a través del cual es posible saber por que terminó el proceso[4].

[4] Una especie de caja negra.

Como se vió anteriormente la interpretación del parámetro n regresado por $wait(n)$ va a ser diferente si el proceso abortó por una u otra causa. En la parte de abajo se presentan dos ejecuciones diferentes del código de la figura 22. Usando el comando $kill()$ se envió una señal al proceso hijo para simular las dos posibles formas en que puede abortar, ya sea a causa del usuario o del núcleo. En el primer de los casos el proceso abortó por que pensó que el usuario tecleó un ctrl-c; el proceso padre desplegó el valor 9 correspondiente a dicha señal (SIGINT). En el segundo ejemplo se realizó una operación invalida (SIGTRAP) y regresó el valor 133, que restandole 128, nos da el 5 enviado por el comando kill.

```
rogomez@cognac:47>cat exstatus.c
main()
{
  int m,n;

  if (fork() == 0) {
     print("Identificador proceso hijo %d \n",getpid());
     for (;;);
  }
  else {
     m= wait (&n):
     print ("Padre: fin proceso %d con estatus: %d \n",m,n);
  }
}
rogomez@cognac:48>cc exstatus.c -o exstatus
rogomez@cognac:49>exstatus2 &
Identificador proceso hijo 21372
[1] 21371
rogomez@cognac:50>kill -9 21372
Padre: fin proceso 21372 con estatus: 9
rogomez@cognac:51>exstatus &
Identificador proceso hijo 21374
[1] 21373
rogomez@cognac:52>kill -5 21374
Padre: fin proceso 21374 con estatus: 133
rogomez@cognac:53>
```

Figura 22: *Ejemplo de terminación normal de un proceso*

9 FE DE ERRATAS

Debido a fallas en el editor gráfico, no fue posible corregir las siguientes figuras:

- *Figura 1: donde dice* `bloqueado` *debe reemplazarse con* `listo`*, y donde esta indicado* `listo` *debe substituirse por* `bloqueado`
- *Figura 2: al lado del círculo numerado con un 8 debe de escribirse el estado* `creado`

El autor espera que con esta información se aclaren todas las posible confusiones del lector.

Referencias

[And91] Concurrent Programming - *Andrews R. Gregory* The Benjamming/Cummings Publisihng Inc. 1991 - 1a. Edición

[Ba84] Bach - *The design of the Unix Operating System* Ed. Prentice Hall

[Bro94] Brown Chris - *Unix Distributed Programming* Prentice Hall

[Coff92] Coffin Stephen - *Sistema V Vers. 4, Manual de Referencia* 1992, McGraw Hill

[Deit82] H.M. Deitel - *Operating Systems* Ed. Addison-Wesley

[Gar94] García Márquez Fco. - *Unix Programación Avanzada* Primera Edición, 1994 Addison-Wesley Iberoamericana

[KePi84] Brian W. Kernighan, Rob Pike - *The Unix Programming Environment* 1984- Prentice Hall

[NeSnSe89] Seebass, Snyder, Nemeth - *Unix System Administration Handbook* 1989, Prentice Hall

[Rif96] Rifflet Jean-Marie - *UNIX 99 exercieses corrigés* Ediscience International 1996 1a. Edición

[Ste90] Stevens Richard W. - *Unix Network Programming* 1992 Prentice Hall

[Ste92] Stevens Richard W. - *Advanced Programming in the Unix Environnenment* Addison-Wesley Professional Computing Series 1992 1a. Edición

[Swi93] Operating Systems a practical approach - *Switzer Robert* Ed. Prentice Hall 1993 1a. Edición

[Tane83] Andrew S. Tanenbaum *Sistemas Operativos Modernos* Ed. Prentice Hall

[Vah95] Uresh Vahalia- *Unix Internals: The New Frontiers* 1995, Prentice Hall