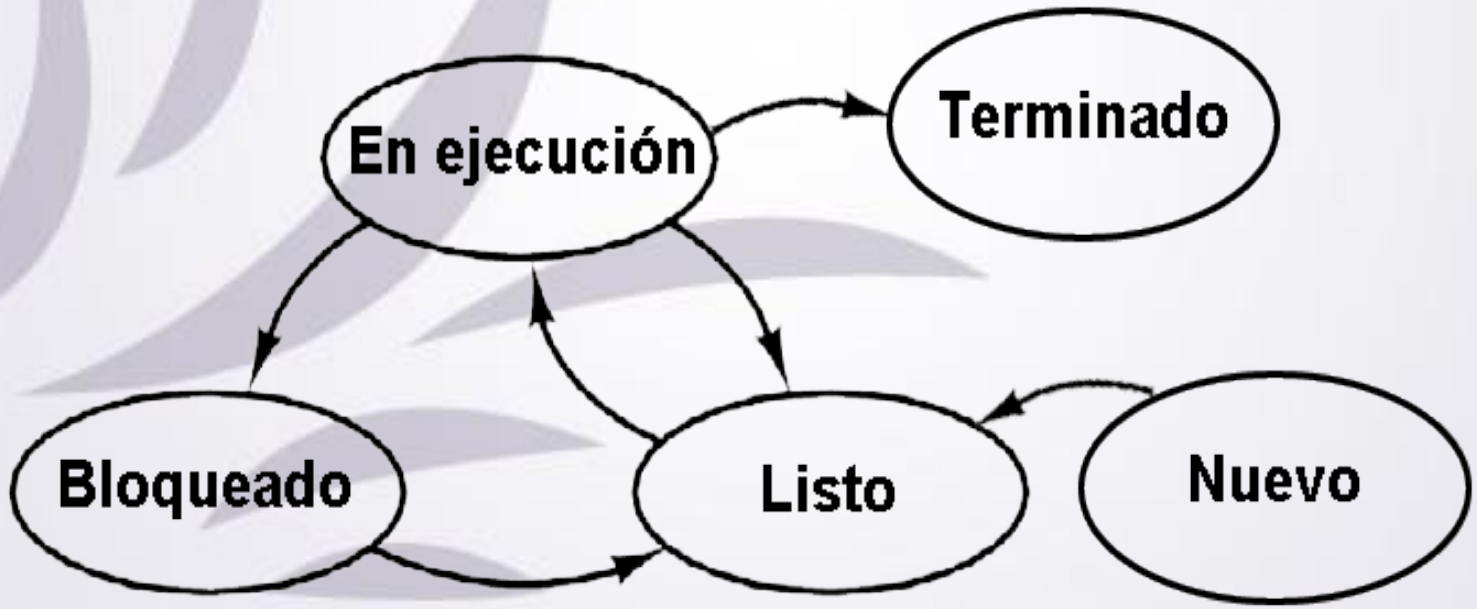
Procesos

Qué es un proceso?

Un proceso es un programa en ejecución que tiene asociado los registros utilizados, el contador de programa, el puntero de pila, las variables y otros registros de hardware.

Estados de un proceso



Creación Procesos

Los cuatro principales sucesos que provocan la creación de nuevos procesos son:

1. La inicialización del sistema.
2. Un proceso ejecuta una llamada al sistema para de creación de un nuevo proceso.
3. La petición por parte del usuario de la creación de un nuevo proceso.
4. El inicio de un trabajo en batch.

Finalización de Procesos

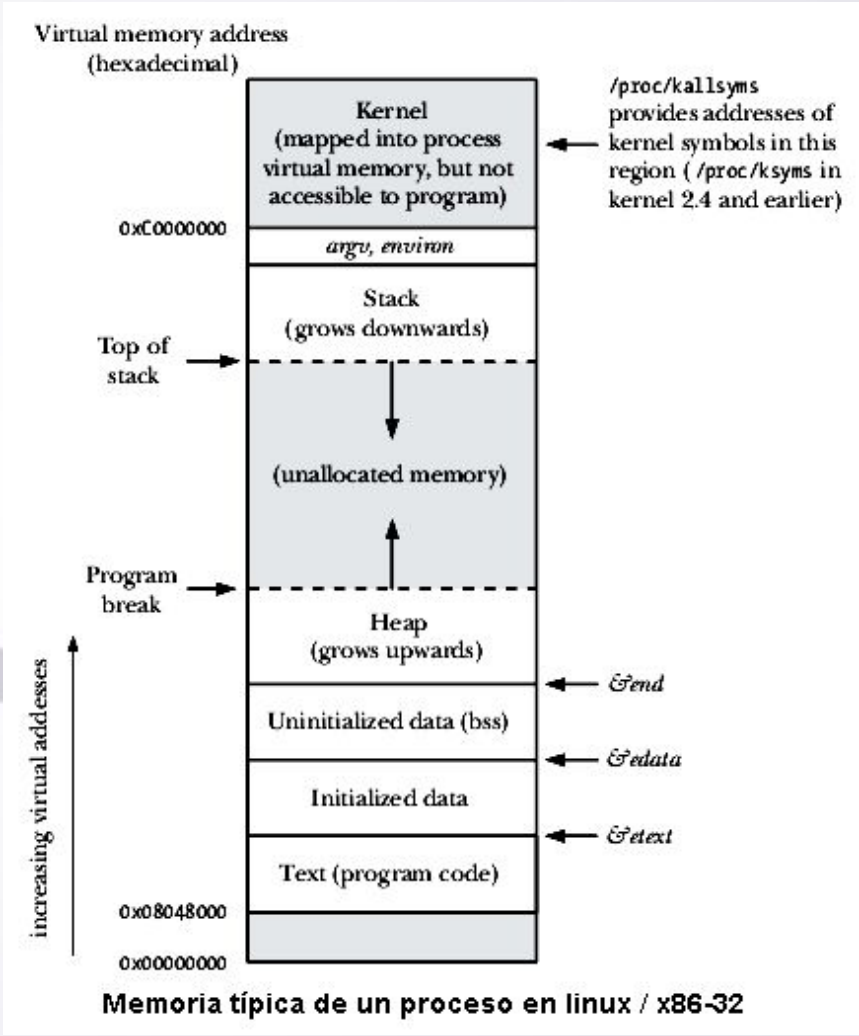
Los cuatro principales sucesos que provocan la finalización de un procesos son:

1. El proceso completa su trabajo y termina (voluntariamente).
2. El proceso detecta un error y termina (voluntariamente).
3. El sistema detecta un error fatal del proceso y fuerza su terminación.
4. Otro proceso fuerza la terminación del proceso (por ejemplo en UNIX mediante la llamada al sistema kill).

Memoria de un Proceso

La memoria de un proceso se divide en segmentos:

- **Segmento *Text***: instrucciones en lenguaje de máquina que ejecuta el proceso. Tamaño constante.
- **Variables inicializadas**. Tamaño constante.
- **Variables no inicializadas**. Tamaño constante.
- **La pila** (*stack*): contiene un marco de datos por cada llamada a función. Varía dinámicamente. El puntero a pila (*stack pointer*) indica la dirección superior de la pila.
- **El *heap***. Contiene variables que son creadas en tiempo de ejecución. Varía dinámicamente.



Memoria típica de un proceso en linux / x86-32

Jerarquia de Procesos

En linux existe una jerarquía de procesos:

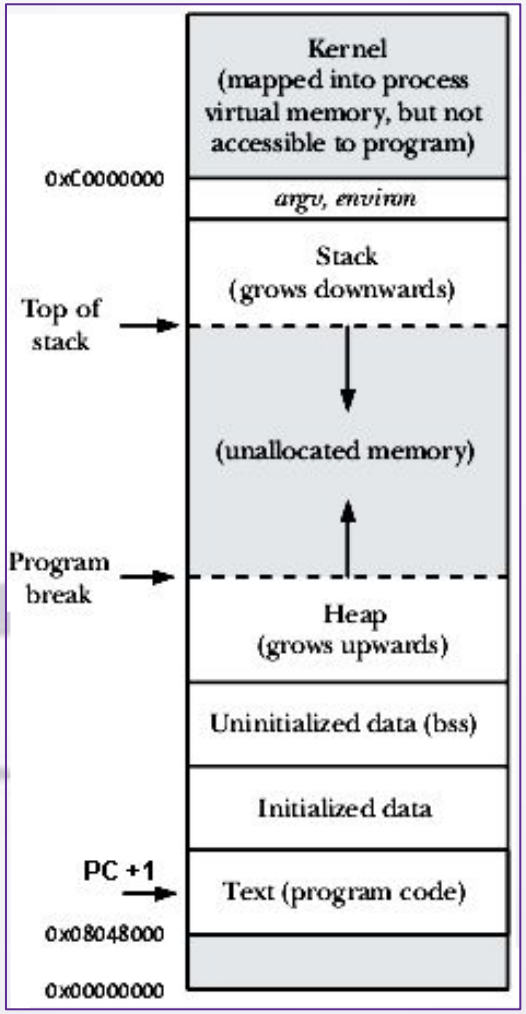
- Un proceso A crea a otro proceso B, el proceso A es el padre y el proceso B es el hijo.
- Un proceso, todos sus hijos y demás descendientes forman un grupo de procesos.
- Los procesos se identifican con un ID de proceso, PID.

En Windows no hay jerarquías de procesos.

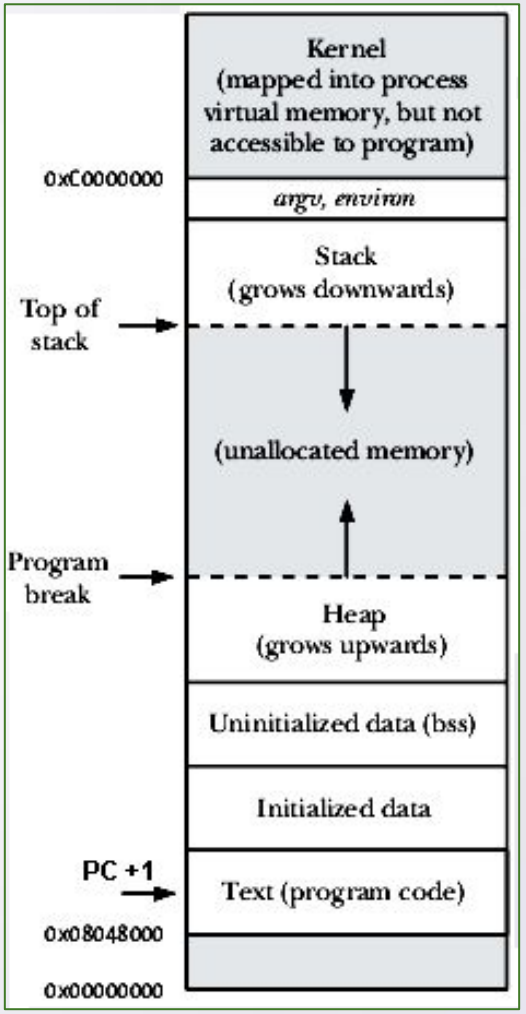
Procesos Hijos

Al crear un proceso hijo:
El nuevo proceso hijo es casi una copia exacta del proceso padre, obtiene copias de la pila del padre, los datos, el heap y el segmento de texto.
El padre e hijo se diferencia en el resultado de la llamada fork(). Al proceso padre le devuelve el PID del hijo creado y al proceso hijo le devuelve cero.

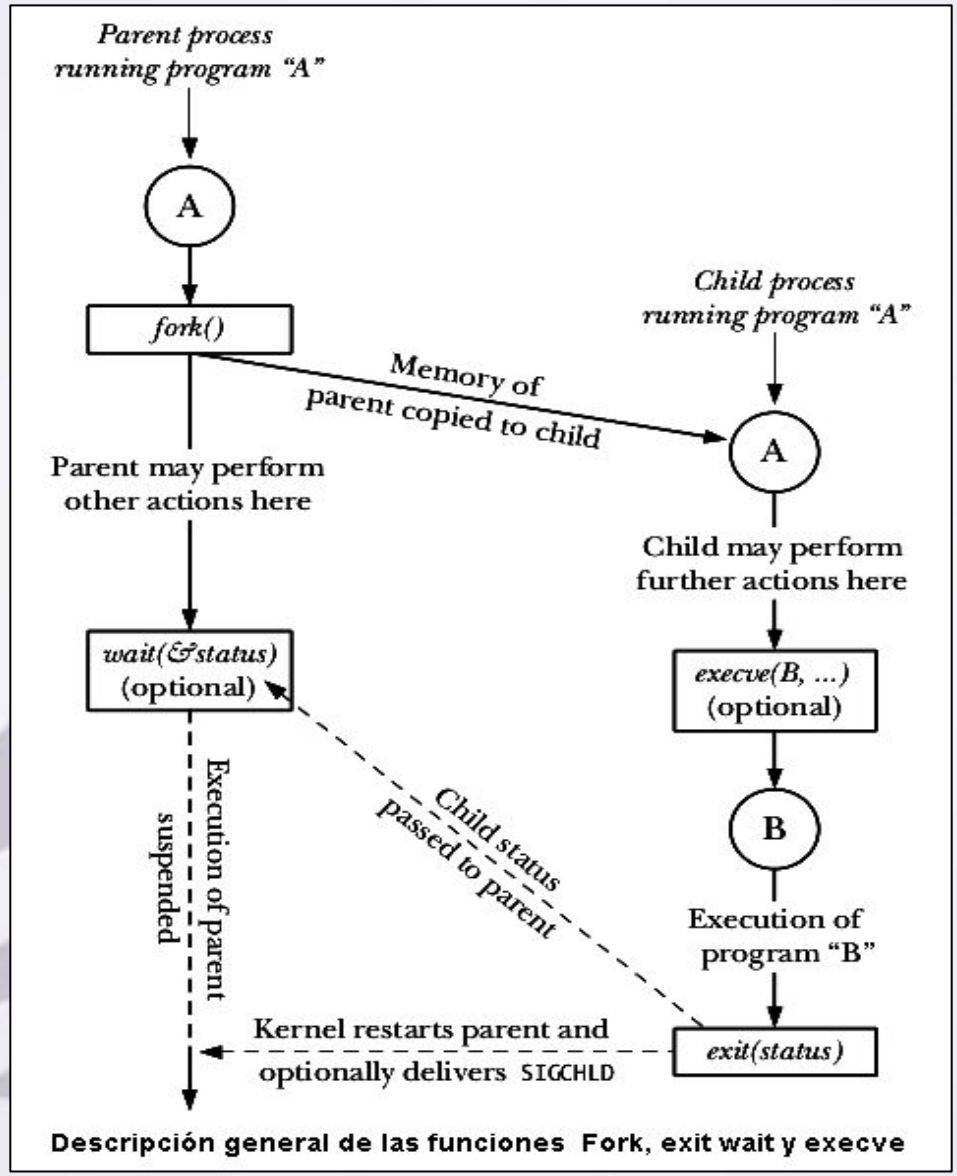
Memoria proceso padre



Memoria proceso hijo



Procesos Hijos



Descripción general de las funciones Fork, exit wait y execve

Creación de un proceso: fork()

La llamada a sistema fork() permite a un proceso (proceso padre) crear otro proceso (proceso hijo).

```
#include <sys/types.h>    //biblioteca en la que definido el tipo pid_t
#include <unistd.h>       //biblioteca en la que definida fork
pid_t fork(void);
```

fork() devuelve:

- Al padre el ID de proceso hijo creado
- Al hijo 0
- -1 en caso de error

El nuevo proceso hijo es casi una copia exacta del proceso padre: obtiene copias de la pila del padre, los datos, el heap y el segmento de texto

Funciones getpid() y getppid()

```
#include <unistd.h>
pid_t getpid(void);
```

getpid() nos devuelve el pid del proceso actual

```
#include <unistd.h>
pid_t getppid(void);
```

getppid() nos devuelve el pid del proceso padre

Ejemplo:

```
printf (“soy el pid: %d y mi papa es: %d\n",getpid(),getppid());
```

Función exit()

La función exit(status), termina un proceso por lo que todos los recursos (memoria, descriptores etc) utilizados por el proceso quedan disponibles.

El argumento de la función es un entero que determina el estado de terminación del proceso. Usando la llamada wait(), el padre puede obtener ese estado.

```
#include <unistd.h>

void exit(int status);
```

Llamada a sistema wait()

La llamada a sistema wait(&status) tiene dos propósitos, si un proceso hijo aún no termina, esperarlo hasta que termine, una vez terminado el hijo obtener el estado de terminación en status.

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

pid_t wait(int *status);
```

Devuelve el ID del proceso hijo terminado o -1 en caso de error

Comando pstree

pstree –p nos muestra los PID de los procesos

```
usuario@nombre_pc:~$  pstree -p

init(1)-+-NetworkManager(4530)---{NetworkManager}(5168)
|-NetworkManagerD(4544)
|-acpid(4328)
|-atd(5328)
|-avahi-daemon(4586)---avahi-daemon(4587)
|-bonobo-activati(5662)---{bonobo-activati}(5666)
.
.
.
|-gnome-terminal(7118)-+-bash(7123)-+-pstree(7524)
|                      |            `-watch(7162)
|                      |-gnome-pty-helpe(7122)
|                      `-{gnome-terminal}(7124)
`-inetd(4927)
```

Procesos huérfanos y zombies

¿Qué pasa cuando un proceso padre termina antes que un proceso hijo?

El proceso hijo se convierte en huérfano y se le asigna como nuevo padre un proceso superior o el proceso Init cuyo PID es 1

¿Qué pasa cuando un proceso hijo termina antes que un proceso padre?.

El proceso hijo se convierte en zombie. Es borrado de memoria, excepto su PID y su valor de retorno (status).

Un zombie no puede ser “matado” por kill.

Cuando el padre hace un wait(), el zombie es sacado de memoria por completo.

Funciones de la biblioteca exec()

Todas estas funciones se encuentran en capas en la parte superior de execve(), y se diferencian unas de otras sólo en la forma en la que se especifican el nombre del programa, la lista de argumentos, y el entorno del nuevo programa.

La llamada a sistema **execve**(pathname, argv, envp) carga un nuevo programa en la memoria de un proceso.
execve(nombre de ruta, con la lista de argumentos argv y lista de entornos envp)

El texto del programa existente se descarta
La pila, los datos y los segmentos del heap son nuevamente creados

Función execl()

Ejecuta un nuevo programa en el mismo proceso. Se crea un proceso imagen sustituyendo el programa actual por el nuevo.

execl() requiere que el programador especifique los argumentos como una lista de strings dentro de la llamada. El puntero NULL debe terminar la lista de argumentos, así las llamadas pueden localizar el final de la lista.

```
#include <unistd.h>
int execl(const char *pathname, const char *arg, ...
                » /* , (char *) NULL */);
```

No devuelve nada en caso de éxito y -1 en caso de error.

Ejemplo:

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

Bibliografía

- Tanenbaum, Andrew S. Sistemas Operativos Modernos, 3era Edición. Prentice Hall. 2009. Capítulo 2.1.

- Kerrisk, Michael. *The linux programming Interface.* 2011. **Capítulos 6, 24.1, 24.2, 25.1, 25.2, 26**.

Tipos de dato “algo_t”

En programación en C es común crear tipos de datos que están relacionados con el propósito de la variable creada. El tipo de dato `pid_t` se crea al hacer,

```
typedef int pid_t;
```

`typedef` es una palabra reservada del lenguaje C.

Luego, en el programa se podrá hacer,

```
pid_t my_pid;
```

Esta instrucción es equivalente a hacer,

```
int my_pid;
```

Otros tipos de datos comunes en C: `int8_t, uint8_t, int32_t, uint32_t, time_t.`

Puntero nulo

Un puntero nulo (NULL) es un puntero que no apunta a ninguna dirección de memoria. Algunos usos del puntero nulo son:

1) Para inicializar un puntero cuando aún no tiene asignada una dirección de memoria válida.

```
int * p = NULL;
```

2) Para pasar un puntero nulo como argumento de una función cuando no queremos pasar ninguna dirección de memoria válida.

```
wait(NULL);
```

c) Comprobar si un puntero es nulo antes de acceder a cualquier variable del puntero.

```
if (NULL == p) { ........ }
```