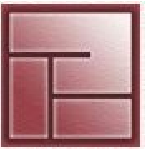
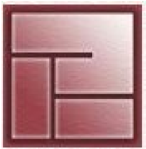


Instituto
Politécnico
Superior

PROCESOS



Definición de Proceso

- Término introducido por los diseñadores de Multics en los 60 como término más general que trabajo (job)

¿Qué es un proceso?

- Es la unidad de trabajo en un sistema de tiempo compartido

- Un proceso es básicamente un programa *en ejecución*
  Consta del programa ejecutable y la pila o stack del programa, su contador de programa, apuntador de pila y otros registros, descriptores de archivo, etc.

- **Una de las principales responsabilidad del SO es la de controlar la ejecución de los procesos**

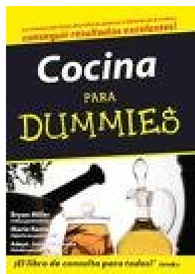
Instituto
Politécnico
Superior

Definición de Proceso

Diferencias: programa – proceso

Receta

Programa



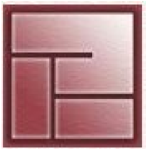
Cocinero

Sistema Operativo

Actividades para llevar a cabo la receta

Proceso

- Un programa es una entidad pasiva, un proceso es activo
- Un programa instancia (pone los valores iniciales) a un proceso
- Puede haber dos o más procesos asociados a un mismo programa



Segmentos o secciones de un Proceso

TEXTO
Instruction Segment

- Instrucciones en lenguaje máquina
- Es read-only

DATO
User Data Segment

- Datos inicializados y no inicializados
- Malloc / heap
- Stack: variables auto y parámetros de funciones como argc, argv[ ] y env[ ]

SISTEMA
System Data Segment

- File descriptors (**EJEMPLO**)
- Directorio de trabajo y terminal
- Información de recursos usados y límites
- Real user ID: dueño del proceso
- Effective user IDs (setuid, setgid)
- Real y effective GID
- Handlers de señales
- Argumentos y valores de retorno de syscalls
- Bloque de Control de Proceso (PCB)

- El espacio de direcciones (*address space*) de un proceso está conformado por los segmentos de **TEXTO** y **DATO**
- El segmento de **SISTEMA** es mantenido por el sistema operativo y es accesible a través de *syscalls*
- Un programa pone valores iniciales a los segmentos de **TEXTO** y **DATO**
- Dos procesos de un mismo programa se consideran dos secuencias de ejecución diferentes, las secciones de **TEXTO** son equivalentes, los otros segmentos varían

Instituto Politécnico Superior

Identificadores de proceso (Credentials)

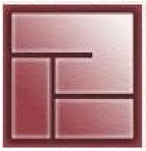
- Todo proceso tiene un identificador único
  `PID` (Process ID)
  - valor entero positivo asignado al momento de la creación del proceso
  - asignación secuencial

- Casi todos los procesos tienen un proceso padre
  `PPID` (Parent Process ID)

- Por lo general en UNIX/Linux existen los procesos
  - **`scheduler`** (**`PID=0`**) / En Linux es "Tarea no existente"
  - **`init`** (**`PID=1`**) adm. los procesos `login`, ancestro de la mayoría de los procesos
  - **`pager`** (**`PID=2`**) memory pager – Memoria virtual

- Máximo PID

```
/proc/sys/kernel/pid_max

# sysctl kernel.pid_max
# sysctl -w kernel.pid_max=4194303
```



Identificadores de proceso

```
#include <stdio.h>
#include <unistd.h>  //declara funciones del estándar POSIX

int main () {
   printf("PID=%d\n", getpid());
   printf("PPID=%d\n", getppid());

   return 0;
}
```

Prototipos:

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

<sys/types.h>

```
$ gcc -W -Wall pid.c

$ echo $$
10705

$ ./a.out
PID=10834
PPID=10705
```

Instituto Politécnico Superior

Tabla de Procesos y la u-área

- Estructuras de datos fundamentales relacionadas con los procesos
- Todo proceso que esté o no en memoria, esté ejecutándose o no, contiene
  - una entrada en la Tabla de Procesos (se crea y elimina cuando lo hace el proceso)
  - una u-área

Tabla de Procesos

- Estructura tipo array o lista enlazada de tamaño fijo
- Impone el límite del número máximo de procesos
- Tabla que está permanentemente cargada en memoria
- Contiene información (de identificación y funcional) de cada proceso
- Campos
  - Estado del Proceso
  - Información que permite localizar al proceso y su u-área (en memoria o disco)
  - UID, PID, PPID
  - Señales
  - Temporizadores
  - Descriptor de evento (indica que evento espera el proceso cuando está "dormido")

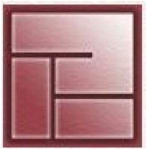


Tabla de Procesos y la u-área

U-área

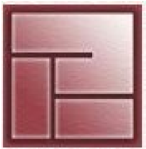
- Sólo está en memoria cuando el proceso correspondiente también lo está
- Puede ser *swapeado* a disco
- Sus datos sólo pueden ser manipulados por el kernel
  El kernel accede únicamente al u-área del proceso en ejecución
- Campos más importantes
  - un puntero a la entrada de la tabla de procesos hacia el proceso que refiere
  - UID y EUID
  - directorio actual del proceso
  - Tabla de *file descriptors*
  - información de como reaccionar a las señales recibidas
  - *El **Bloque de Control de Proceso (PCB)** que almacena el contexto del hardware salvado cuando el proceso no está en ejecución (registros de CPU, Program Counter)*

Instituto Politécnico Superior

/proc

- Pseudo file system (**procfs**) basado en memoria (no ocupa espacio en disco)
- Es dinámico, contiene un directorio por proceso en ejecución nombrado con su PID

- Contiene diversa información sobre cada proceso:
  - `/proc/PID/cmdline` comando que lo inició
  - `/proc/PID/cwd`, un enlace simbólico (*symlink*) al working directory
  - `/proc/PID/exe`, un symlink al ejecutable
  - `/proc/PID/fd`, directorio que contiene un symlink por cada *file descriptor*
  - `/proc/PID/status`, archivo con información básica incluyendo estado de ejecución y uso de memoria
  - `/proc/PID/task`, directorio con *hard links* a toda tarea iniciada por este proceso
  - `/proc/PID/maps`, el mapa de memoria que muestra cuales direcciones actualmente visibles por el proceso están mapeadas en regiones de RAM o archivos



Comandos

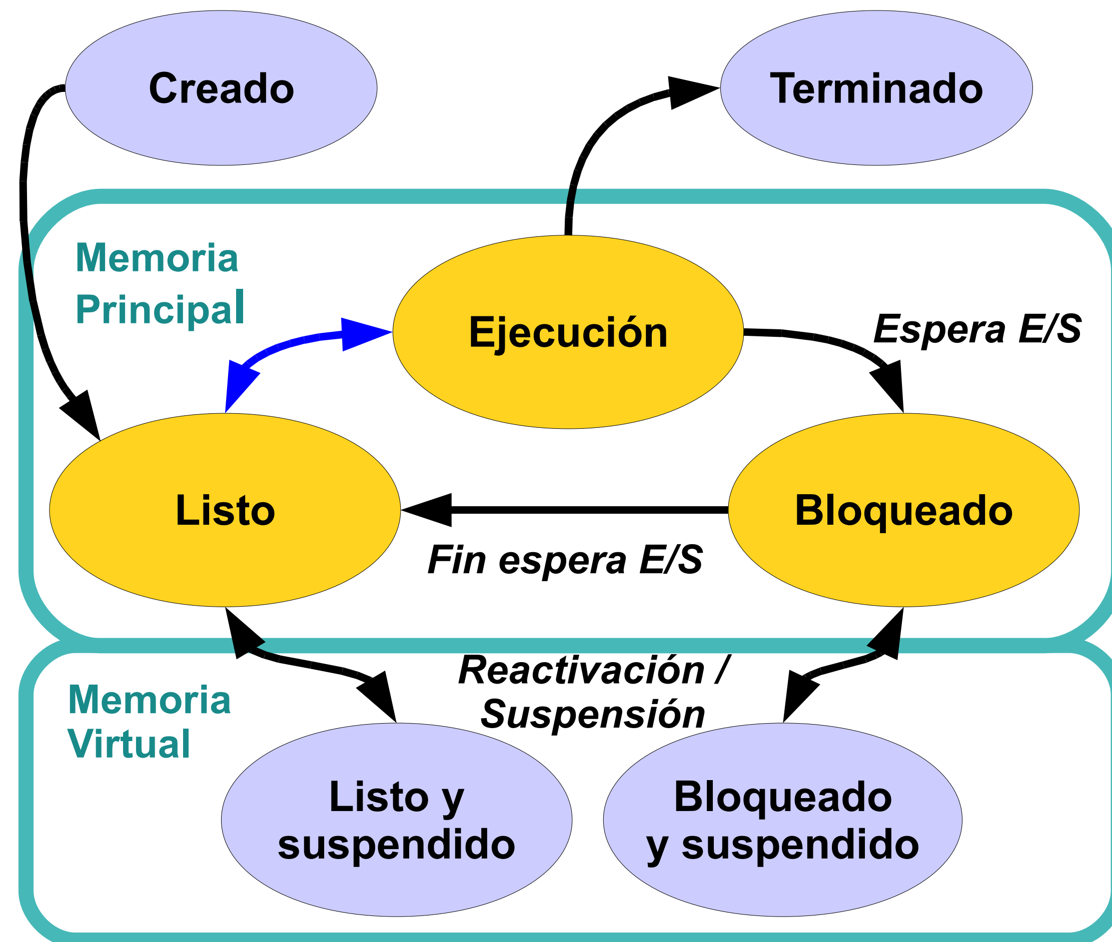
- Process Status (**`ps`**, **`top`**)
- Process tree (**`pstree`**, **`ptree`**)
- Señales: **`kill`**, **`killall`**
- Prioridades: **`nice`**, **`renice`**
- Otros comandos: **`bg`**, **`fg`**, **`jobs`**

Estados de un Proceso

- Un proceso se encuentra en un único estado en cada momento
- A medida que el proceso se ejecuta va cambiando de estado

<u>*GRAFO DE ESTADOS*</u>

- Un grupo de procesos puede estar en **Ejecución**, tantos como admitan los procesadores
- Muchos procesos pueden estar en **Listo** y **Bloqueado**
  Listas o Colas de Planificación
- Transición **azul** determinada por el **Scheduler** o **Planificador**
  Se realiza mediante un cambio de contexto
- Un proceso abandona el estado de **Ejecución**:
  - De manera *cooperativa*
  - De manera *preemptiva*

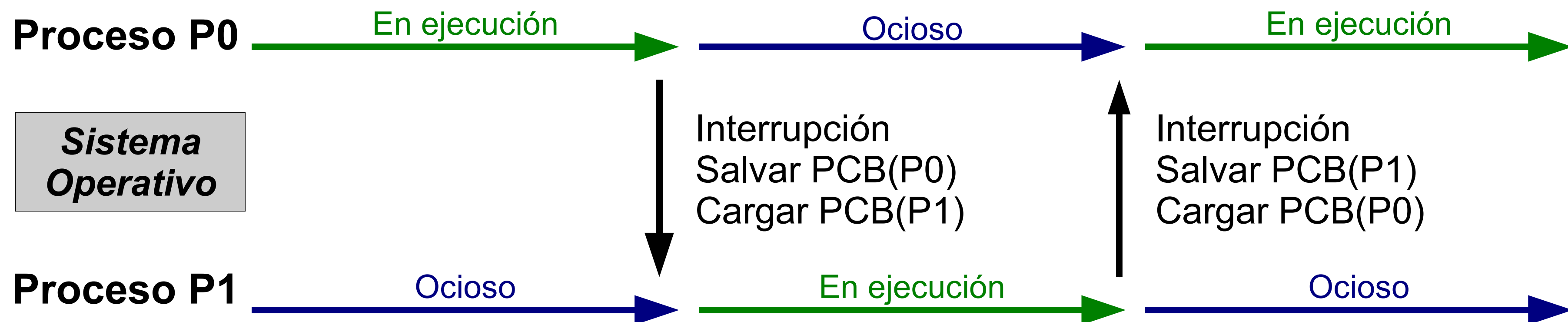


suspendido = swapeado

Instituto Politécnico Superior

Cambio de Contexto

- Consiste en las operaciones que realiza el SO para cambiar de estado a los procesos a fin de detenerlos y posteriormente reanudarlos
- Es el conjunto de dos operaciones:
  - Se resguarda el estado del proceso en el correspondiente PCB
  - Se ejecuta la rutina de tratamiento de interrupción (handler) del SO



- El Planificador o *Scheduler* es la parte del SO que escoje cuales procesos se ejecutan y durante cuanto tiempo
- El Activador o *Dispatcher* es la parte del SO que “pone a ejecutar” los procesos

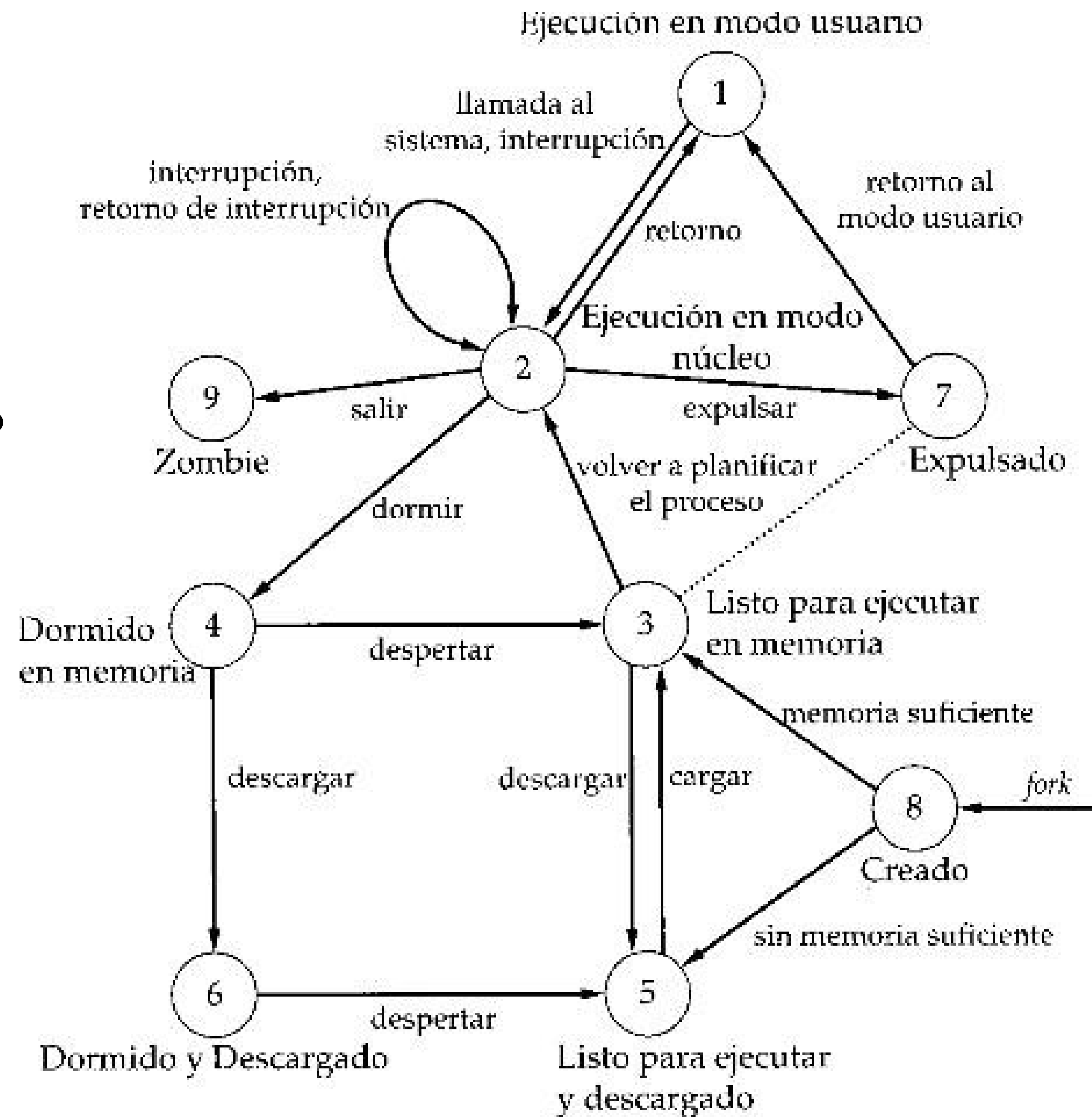
Instituto Politécnico Superior

Procesos en UNIX

- Cada proceso posee dos partes
  - Contexto del hardware
    stack pointer, contador de programa
    palabra de estado, registros
  - Parte de usuario

- Estados

```
1. Ejecución en modo usuario
2. Ejecución en modo núcleo
3. Listo en memoria
4. Bloqueado en memoria
5. Listo suspendido
6. Bloqueado suspendido
7. Apropiado (preempted)
8. Creado
9. Zombie
```



Instituto Politécnico Superior

Tipos de Proceso: Zombie o Defunct

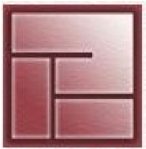
- Proceso que *ha completado su ejecución* pero aun está presente en la *tabla de procesos.*
  Entrada necesaria para que el proceso (padre) que inició el proceso devenido en zombie pueda leer su ***exit status***

- Un proceso zombie es "el certificado de muerte" de un proceso.

- Los procesos zombies que existen por más de instante indican típicamente un bug en el proceso padre (en el código del programa) o sólo una mala decisión del momento en que se recoge el exit status de un hijo

- La presencia de pocos zombies no es problema pero puede agravarse bajo sobrecarga en el sistema

- No es un problema de saturación de memoria pues no tienen memoria *allocada* (excepto lo que ocupa la entrada en la tabla de procesos).
  Sino del uso de identificadores (por la cantidad y la localización)



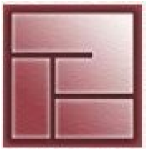
Exit status

- Es un valor numérico pasado de un proceso hijo a un proceso padre cuando el primero finaliza la ejecución de una tarea específica delegada

- Cuando un proceso hijo finaliza su ejecución, puede terminar invocando a la llamada al sistema **`exit`**.

  Esta llamada emite el entero al proceso padre, el cual a su vez recupera el valor mediante la llamada al sistema **`wait`**.

Más información
http://en.wikipedia.org/wiki/Exit_status



Tipos de Proceso: Zombie o Defunct

- Localización:

```
# ps -elf | grep Z
```

- Para removerlos

```
# kill -CHLD <PID-padre>
```

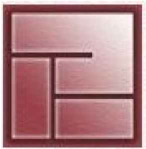
si no funciona (y se puede)

```
# kill -9 <PID-padre>
```

Man page

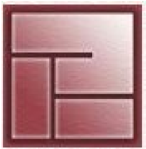
zombies(5)

http://en.wikipedia.org/wiki/Man_page#Manual_sections



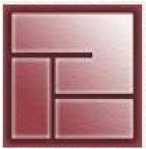
Tipos de Proceso: Huérfano

- Proceso que aun sigue en ejecución pero cuyo padre ha terminado
- No es un proceso zombie, continúa en ejecución, es un “proceso vivo”
- Estos procesos son adoptados inmediatamente por el proceso **`init`** (PID=1), el "proceso *reaper*" (to reap = cosechar)
  Así evitan convertirse en zombies.
- init además periódicamente ejecuta una llamada **`wait`** sobre los procesos zombies de los cuales es padre, ignorando su exit status.



Tipos de Proceso: Daemons

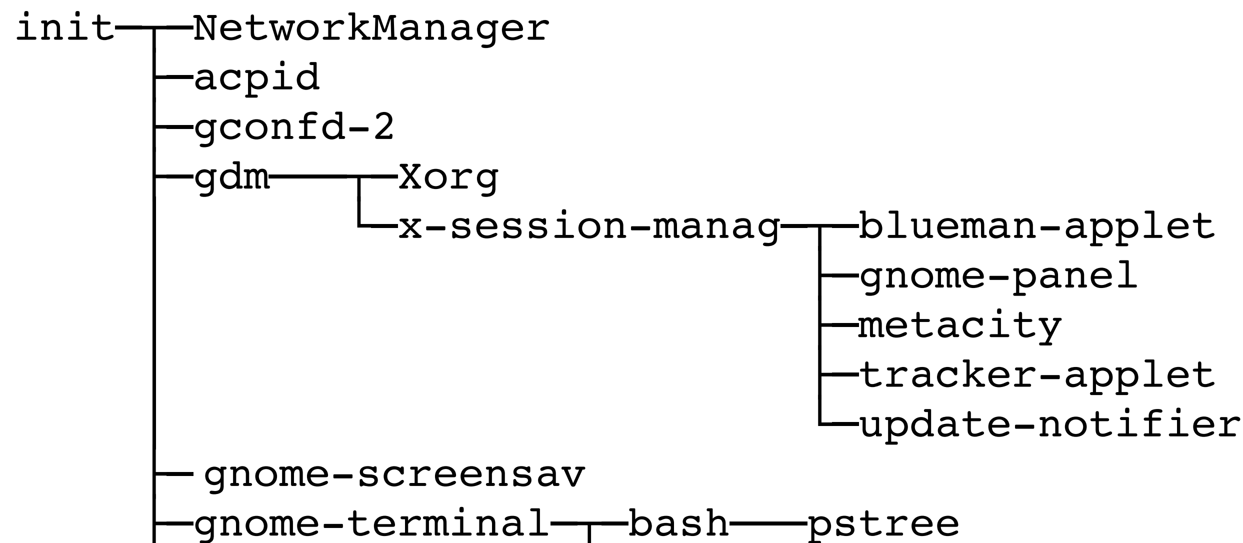
- Procesos que se ejecutan contínuamente en background.
  No están asociados a una terminal, en los comandos como tty tienen “?”
- En Linux/UNIX por convención finalizan su nombre con “d”
- Usualmente se lanzan de manera automática en boot-time y finalizan cuando el servicio que implementan ya no es más necesario
- Ejemplos
  - `xinetd`
  - `sshd`
  - `ftpd`
  - `httpd`
  - `mysqld`



Jerarquía de Procesos

- El SO crea procesos mediante rutinas internas, para ello ofrece primitivas a los usuarios

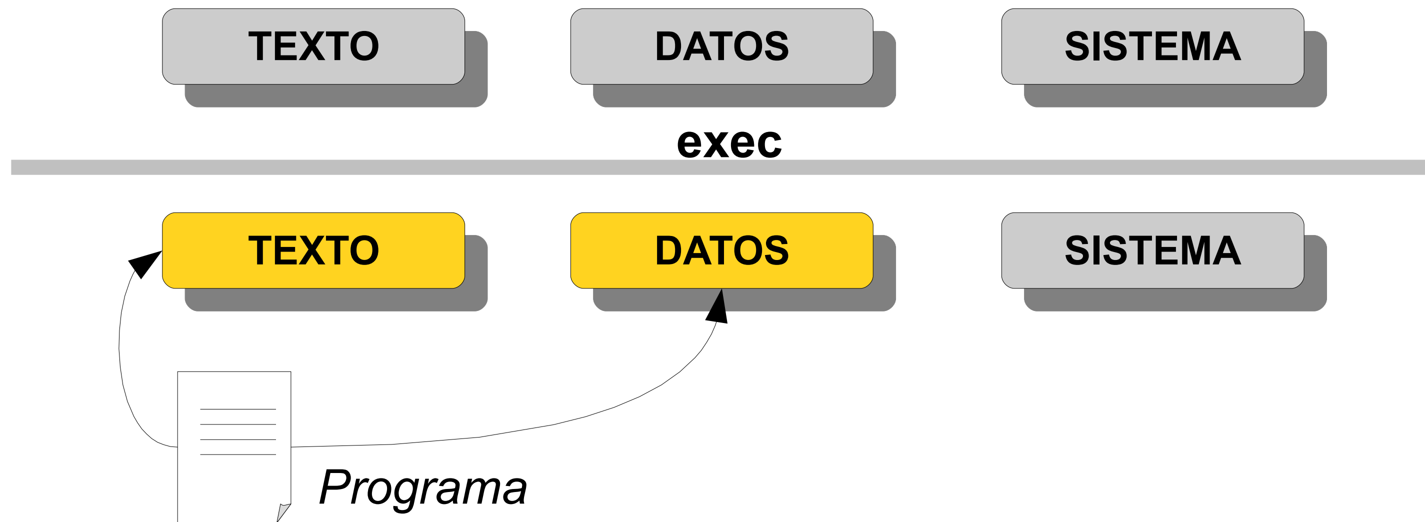
- Cuando un proceso crea a otro procesos se dice que ha creado un *proceso hijo*, el primero es llamado *proceso padre*



Instituto Politécnico Superior

Familia de funciones `exec`

- Concepto
  - El proceso que invoca esta función es reemplazado por el recientemente creado
  - Se trata de un reemplazo de procesos, no incrementa el número de procesos
  - Es un reemplazo de procesos (imagen del proceso), el invocante se auto-termina



- El objetivo es ejecutar una imagen nueva del proceso

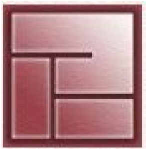
Familia de funciones `exec`

- **Exec** es una familia de funciones definidas en **`<unistd.h>`**

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

Notas:

- Todas son front-ends de ***`execve`***
- `path` que indica la nueva imagen, `file` compone un path (si contiene / se usa como `path`, sino su prefijo es obtenido con PATH)
- La "`l`" viene de lista de argumentos `{arg1, arg2, ..., argn, NULL}` estos argumentos se transfieren al programa invocado
- "`v`" indica el uso de un vector de punteros a cadena, el *vector* (array) debe terminar en `NULL`
- "`e`" viene de *environment* (entorno) es una serie de variables con valores, sintaxis similar a la lista de argumentos



Familia de funciones `exec`

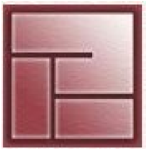
Valor de retorno:

- En caso de éxito no hay valor de retorno ***¿por qué?***
- En caso de error retorna -1 y se establece *errno* para indicar el tipo de error
  Razones
  - `[E2BIG]`: Se supera la cantidad de memoria (bytes) para los argumentos y el environment
  - `[EACCES]`: problemas de permisos
  - `[ENOENT]`: imagen no existe
  - etc.

Enlaces útiles
http://es.wikipedia.org/wiki/Errno.h
http://en.wikipedia.org/wiki/Perror

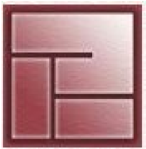
Ejemplo:

```
cat errno.c
./errno
/usr/include/asm-generic  (errno-base.h y errno.h)
```



- Exec: Ejemplos

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

int main() {
   int ret;
   char *args[]={"ls", "-l", NULL};

   ret = execv("/bin/ls", args);

   if (ret == -1) {
      printf("errno = %d\n", errno);
      perror("execv");
      return -1;
   }

   printf("Codigo inalcanzable");
   return 0;
}
```



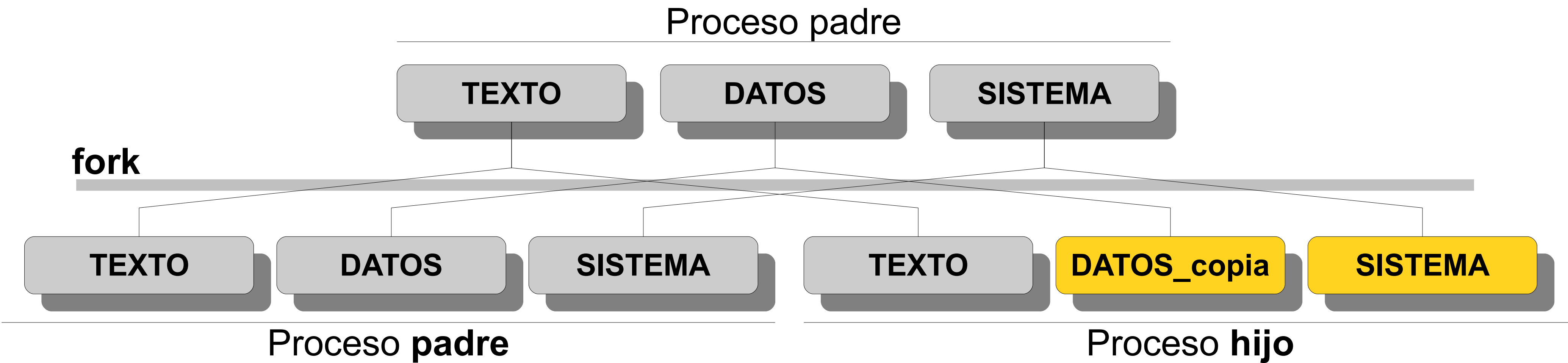
Exec: Ejemplos

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

int main() {
    int ret;
    char *args[]={"sleep", "1000", NULL};
    char *env[]={"LOGNAME=gdm", "PWD=/opt", NULL};

    ret = execve("/bin/sleep", args, env);

    if (ret == -1) {
        printf("errno = %d\n", errno);
        perror("execve");
        return -1;
    }

    printf("Codigo inalcanzable");
    return 0;
}
```

```
$ cat /proc/PID/environ
LOGNAME=gdmPWD=/opt
```

Instituto Politécnico Superior

- **`fork`**
- Crea un proceso nuevo (proceso hijo)
- Este proceso es una “copia” del proceso que invoca a fork

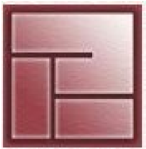


- Prototipo

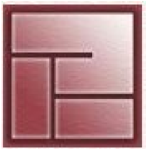
```
pid_t fork (void)
```

- Llamada exitosa
  - Retorna el `PID` del proceso hijo al proceso padre
  - Retorna 0 al proceso hijo
- En error retorna -1 y setea errno (`EAGAIN`, `ENOSYS`)

- El proceso hijo es una copia del proceso padre, excepto `PID` y `PPID`




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

int main() {
   pid_t pid_hijo = fork();
   int var = 5;

   switch(pid_hijo) {
     case 0: {   //proceso hijo
        var = 10;
        printf("Proceso hijo\n");
        printf("Hijo: PID=%d, PPID=%d - Variable=%d\n", getpid(), getppid(), var);
        break;
     }
     case -1: {
        printf("errno = %d\n", errno);  perror("fork");
        break;
     }
     default: {  //proceso padre
        printf("Proceso padre\n");
        printf("Padre: PID=%d, PPID=%d - Variable=%d\n", getpid(), getppid(), var);
        break;
     }
  }
  return 0;
}
```



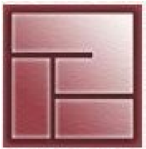
fork

- Mediante fork se crea un espacio de direcciones *diferente* para el proceso hijo *copy-on-write* (`http://es.wikipedia.org/wiki/Copy-on-write`)

- El environment, los resource limits, umask, current working directory, root directory, signal masks y otros recursos del proceso padre son duplicados y asignados al proceso hijo

- No debería (sin protección) ejecutarse código perteneciente al proceso hijo que dependa de la ejecución del código del proceso padre, ni viceversa

- Es necesario emplear herramientas de sincronización



`vfork`

- En lugar de hacer una copia del espacio de direcciones la comparte. Esto reduce la sobrecarga generada por duplicar este bloque
- Se usa generalmente en un ciclo ***fork – exec – fin***
- El proceso padre al invocar `vfork()` espera por la finalización del proceso hijo (con `exec()` o `_exit()`)
- Dados los avances (cow) en la implementación de fork(), vfork es muy poco empleada pues es innecesaria, en ciertas implementaciones vfork invoca a fork.
- Posible problema, ¿qué ocurre si el hijo no finaliza?

Instituto Politécnico Superior

`fork` bomb

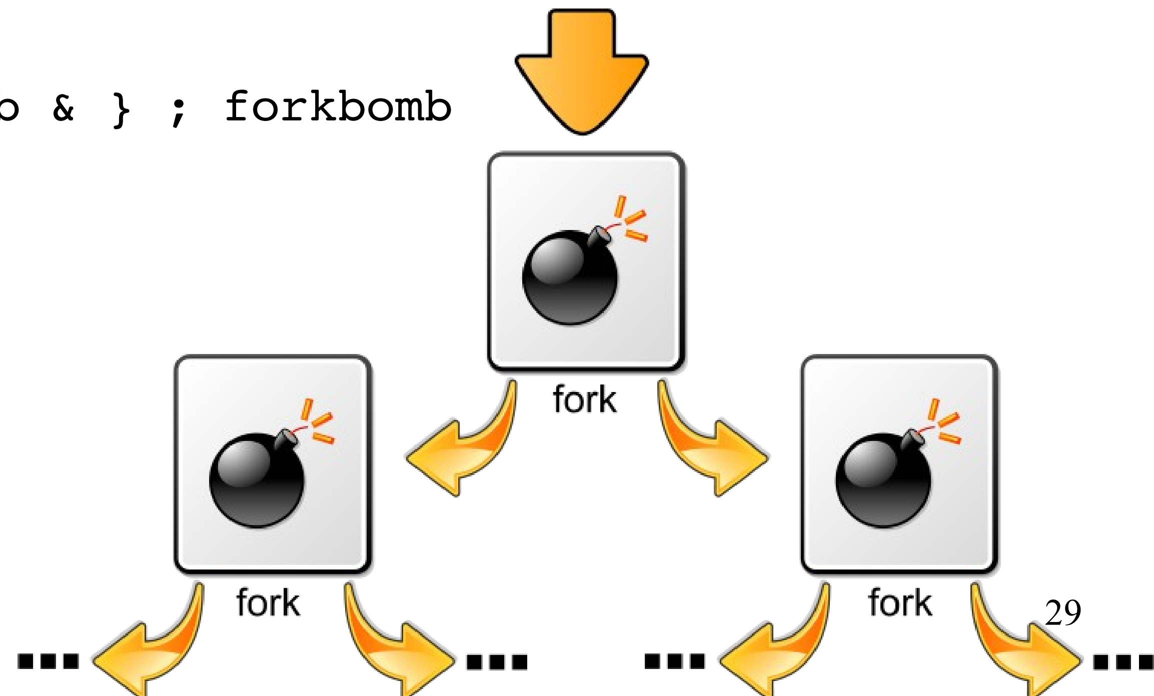
- Tipo de ataque estilo *denegación de servicio*
- Wabbit (http://en.wikipedia.org/wiki/Wabbit)
- Incapacita sistemas basándose en la presunción de que el número de procesos que un sistema puede ejecutar simultáneamente es finito.
- Crear muchos procesos muy rápido (fork's recursivos)
- Fork bombs usan espacio en las tablas de proceso + tiempo de CPU + memoria

Ejemplos

1) `forkbomb(){ forkbomb|forkbomb & } ; forkbomb`

2)

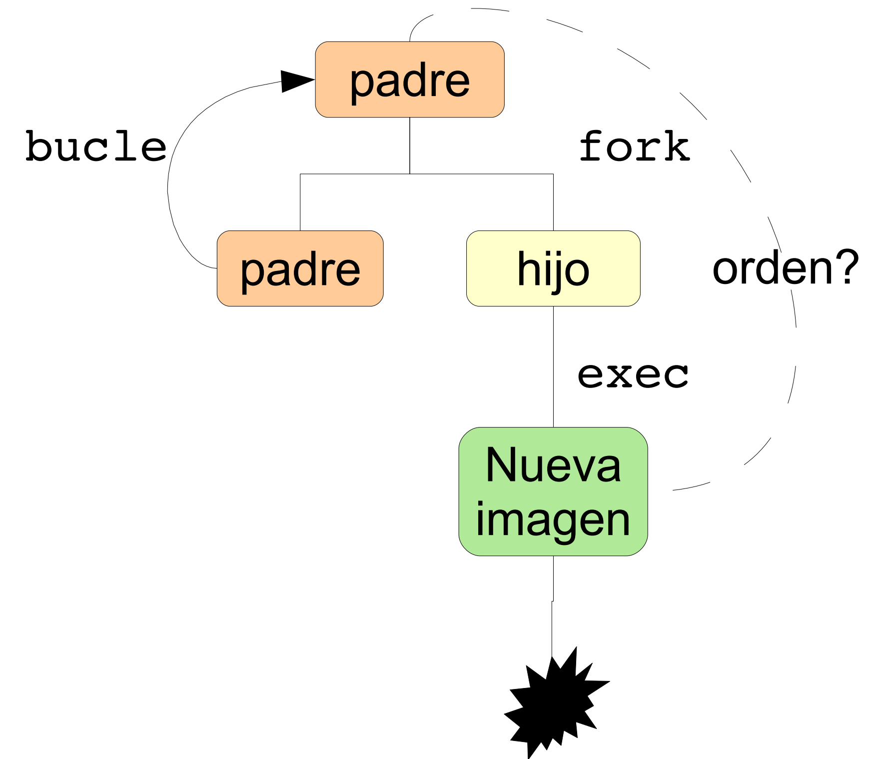
```
#include <unistd.h>

int main() {
  for(;;)
    fork();
  return 0;
}
```



- Recuperación & Prevención

Shell de juguete

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

int main() {
  char comando[1000];

  for (;;) {
      printf("$ ");
      gets(comando);

      if (fork()) ;
      else execl(comando, comando, NULL);
  }

  return 0;
}
```



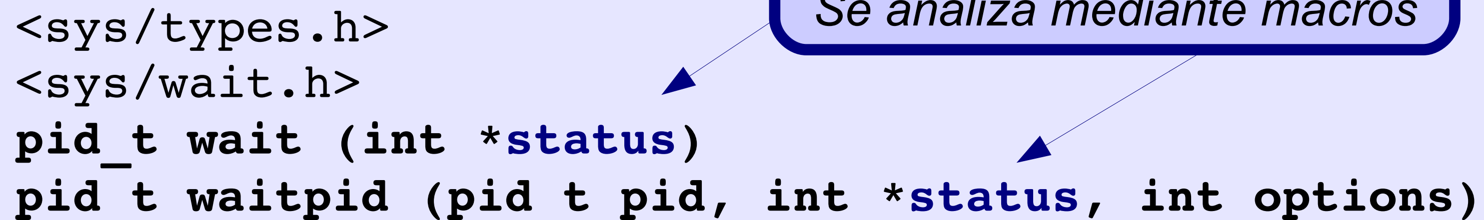
- Es necesario sincronizar los procesos, que el padre aguarde por la finalización del proceso hijo

- Emplearemos las funciones wait

wait

- El proceso padre debe aguardar por la finalización del proceso hijo para recoger su exit status y evitar la creación de procesos zombies

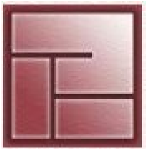
- Prototipos




```
<sys/types.h>
<sys/wait.h>
pid_t wait (int *status)
pid_t waitpid (pid_t pid, int *status, int options)
```

- `wait` bloquea al proceso invocante hasta que el proceso hijo termina.
  Si el proceso hijo ya ha terminado retorna inmediatamente.
  Si el proceso padre tiene múltiples hijos la función retorna cuando uno de ellos termina.

- `waitpid` Posee opciones para bloquear al proceso invocante a la espera de un proceso en particular en lugar del primero.



- **waitpid**

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

pid

- `< -1`: esperar por cualquier hijo que tenga `GID = |pid|`
- `-1`: esperar por cualquier hijo que tenga `GID = |pid|`
- `0`: esperar por cualquier hijo cuyo $GID = GID_{padre}$
- `> 0`: esperar por el hijo con `PID = pid`

options (es una máscara OR)

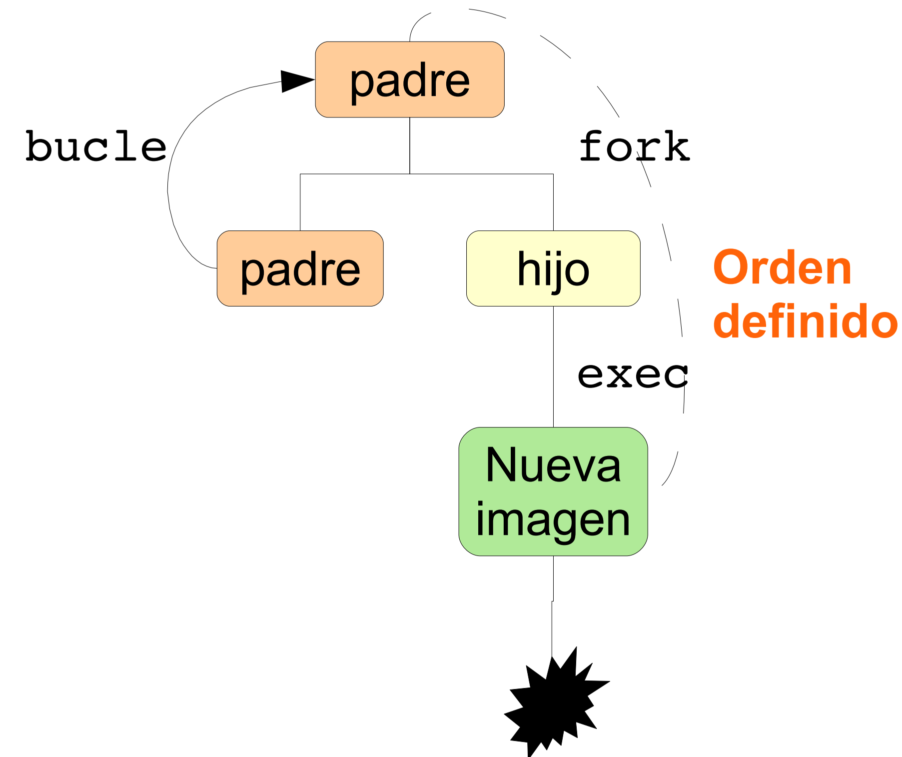
WHOHANG, WUNTRACED, WCONTINUED, WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WCOREDUMP, WIFSTOPPED, WSTOPSIG, WIFCONFIRMED

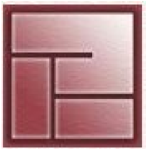
```
int *status
wait(&status) ≡ waitpid(-1, &stauts, 0)
```

Instituto Politécnico Superior

Shell de juguete - **Procesos sincronizados**

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/types.h>


int main() {
  char comando[1000];
  int *status;

  for (;;) {
      printf("$ ");
      gets(comando);

      if (fork()) wait(status);
      else execl(comando, comando, NULL);
  }

  return 0;
}
```





Función `system`

- Función que ejecuta un comando de shell

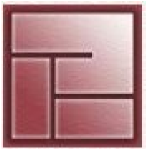
- Prototipo:

```
<stdlib.h>
int system (const char *string)
```

- Función implementada utilizando `fork – waitpid – exec`

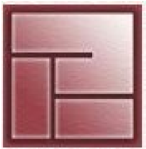
- La cadena contiene el comando y se ejecuta mediante `/bin/sh -c string (string es el comando)`

- Retorna
  . el *exit status* del shell como si fuese retornado por `waitpid`
  . -1 si ocurre un error en `fork` o `waitpid`
  . 127 si la ejecución del shell falla



Función `system`

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

int main()
{
    system("ls -l");
    printf("Command done!\n");

    return 0;
}
```



Terminación de procesos

- Un proceso puede terminar por varias razones:
  - su función main ejecuta la sentencia **`return`**
  - llamando a la función **`exit()`**
  - llamando a la función **`_exit()`**
  - llamando a **`abort()`**
  - es finalizado por una señal

Terminaciones normales (return, exit(), _exit())

Terminaciones anormales (abort(), señal)

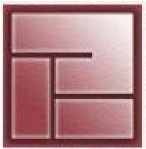
- Más allá del porque finaliza un proceso el kernel ejecuta estas tareas típicas:
  - se cierran archivos abiertos
  - se liberan recursos (identificadores, entradas en tabla de proceso, etc.)
  - se realizan otras tarea de “limpieza”

Función `exit()`

- Función que forma parte de la biblioteca estándar de C
- Prototipo

```
<stdlib.h>
int exit (int status)
```

- Invoca internamente a la llamada al sistema `_exit()`
- Brinda el status al proceso padre (que invocó una función de la familia wait)
  Los valores posibles son `0`, `EXIT_SUCCESS` y `EXIT_FAILURE`
- Se ejecutan todas las funciones registradas con `atexit()`
  `int atexit (void (*ptr) (void))` // no debe fallar ninguna función registrada
  en el orden inverso de registración.
- Programa *exit_atexit*
- Puede ocurrir un ciclo *exit() – atexit(f) – f { exit() }*
  Por ello la función *f* debe utilizar la función de C _exit()



Función `_exit()`

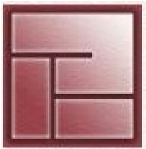
- Prototipo

```
<unistd.h>
int _exit (int status)
```

- Finaliza de inmediato el proceso que la invoca

- No se ejecutan las funciones registradas con `atexit()`

- Todos los file descriptors abiertos que pertenezcan al proceso se cierran
  Todos los procesos hijos son heredados por el proceso *init*
  Al proceso padre se le envía la señal `SIGCHLD`



Función `abort()`

- Prototipo:

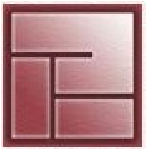
```
<stdlib.h>
void abort (void)
```

- Termina un proceso anormalmente

- Produce que el proceso invocante realice un volcado de memoria (dump), eventualmente empleado por los depuradores

- Esta función invoca `exit(EXIT_FAILURE)`

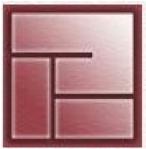
- Es invocada por la macro `assert`



Ejemplo (Generador de procesos zombies)

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

int main() {
    pid_t pids[5];
    int i;

    for (i = 4; i >= 0; --i) {
        pids[i] = fork();
        if (pids[i] == 0) { /* Hijo */
            sleep(i+1);
            _exit(0);
        }
    }

    for (i = 4; i >= 0; --i)
        waitpid(pids[i], NULL, 0);

    return 0;
}
```

```
while :; do
   sleep 1; clear; ps -elf | grep Z;
done
```

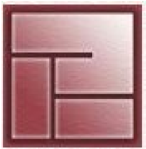


Señales

- Mediante señales un proceso puede finalizar a otro
- Las señales son interrupciones generadas por software y proveen un mecanismo para la manipulación de eventos asincrónicos
- Pueden generarse por muchos motivos:
  - Ctrl+C, Ctrl+Z, Ctrl+\
  - Terminación de proceso hijo
  - Excepciones: división por cero, segmentation fault
  - Las genera el kernel
  - Mediante la llamada al sistema `kill`
- Cada señal tiene un nombre con prefijo SIG y un número entero positivo asociado, estos valores están definidos en `signal.h`

```
SIGHUP 1 (hangup) - SIGINT 2 (interrupt)
SIGQUIT 3 (quit) - SIGILL 4 (illegal instruction)
SIGABRT 6 (used by abort) - SIGKILL 9 (hard kill)
SIGALRM 14 (alarm clock)
SIGCONT 19 (continue a stopped process)
SIGCHLD 20 (to parent on child stop or exit)
```

- Las señales, número y nombre, como su tratamiento están estandarizadas por POSIX

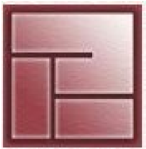


Envío de señales

- Función de C, prototipo

```
<sys/types.h>
<signal.h>
int kill (pid_t pid, int sig)
```

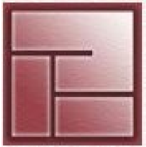
- Se envía la señal **`sig`** al proceso cuyo PID es **`pid`**
- La señal se envía satisfactoriamente si el proceso que envía y el que recibe son del mismo usuario, o bien si el proceso que envía es del root
- `kill()` funciona de forma diferente dependiendo del valor de pid:
  - `pid > 0`, la señal se envía al proceso cuyo PID es pid
  - `pid = 0`, la señal se envía a todos los procesos que pertenecen al mismo grupo del proceso
  - `pid = -1`, la señal se envía a todos procesos cuyo UID real es igual al UID efectivo del proceso que la envía.
    Si el proceso que la envía tiene UID efectivo root, la señal es enviada a todos los procesos, excepto al proceso 0 (swapper) y 1 (init).
  - `pid < -1`, la señal es enviada a todos los procesos cuyo ID de grupo coincide con el valor absoluto de pid.



Llamadas al sistema y acciones

- Llamadas al sistema
  - **kill**: envío de una señal a un proceso
  - **raise**: auto-envío de una señal
  - **alarm**: planificar envío de una señal de alarma
  - **signal**: registrar handler para una señal
  - **pause**: espera hasta recepción de una señal

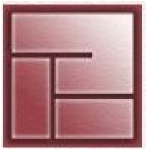
- Los procesos frente a una señal pueden:
  - ignorarla
  - invocar una rutina (handler) por defecto
  - invocar una rutina registrada por el programador



Ejemplo (Finalización de procesos)

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  pid_t pid_hijo;
  int* status_hijo;

  pid_hijo = fork();

  if (pid_hijo) {
    /* Padre */
    printf("Mi hijo es el proceso %d, espero...", pid_hijo);
    wait(status_hijo);

    /* Determinar el motivo de la terminacion */
    if (WIFEXITED(*status_hijo))
      printf("\nTerminacion normal del proceso hijo con exit code %d.\n",
              WEXITSTATUS(*status_hijo));

    if (WIFSIGNALED(*status_hijo)) {
       int signal_id = WTERMSIG(*status_hijo);
       char *signal_name = strsignal(signal_id);
       printf("\nTerminacion debido a la señal %d (%s).\n", signal_id, signal_name);
    }
  } else {
```

```
/* Hijo */
    int i = 0;
    sleep(5);

    while(i++ < 120) {
      fprintf(stdout, "%d - ", i++);
      sleep(1);
    }
  printf("Hijo termina normalmente");
  _exit(status_hijo);
  }

  return 0;
}
```



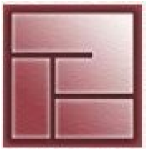
Referencias del ejemplo

`WIFEXITED(status)`
evalúa a true si el hijo terminó normalmente.

`WEXITSTATUS(status)`
evalúa los ocho bits menos significativos del exit status del hijo.
Esta macro sólo debe usarse si `WIFEXITED` devuelve un valor distinto de cero.

`WIFSIGNALED(status)`
evalúa a true si el proceso hijo terminó a causa de una señal no capturada.

`WTERMSIG(status)`
devuelve el número de la señal que provocó la muerte del proceso hijo.
Esta macro sólo puede ser evaluada si `WIFSIGNALED` devolvió un valor distinto de cero.

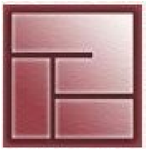


Raise

- Envía una señal al proceso invocante
- Prototipo:

```
int raise (int sig)
```

- Retorna -1 en caso de error y se establece errno
- Es equivalente a `kill(getpid(), sig)`



Signal

- Registra un handler para una cierta señal

- Prototipo

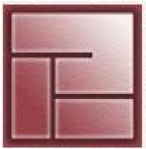
```
sighandler_t signal(int signum, sighandler_t handler)
```

- Parámetros

  signum: número de señal que se quiere enviar
  handler: función para administrar la señal recibida

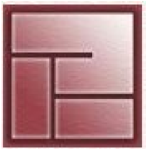
```
typedef void (*sighandler_t)(int)
```

- En caso de error retorna `SIG_ERR` (-1)



Signal

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

void sigproc(int sig) {
   printf("Ud presiono ctrl-c \n");
   printf("recibida la señal numero %d\n", sig);
}

void quitproc(int sig) {
   printf("Ud presiono ctrl-\\ \n");
   printf("recibida la señal numero %d\n", sig);
   exit(0);
}

int main() {
  signal(SIGINT, sigproc);
  signal(SIGQUIT, quitproc);
  printf("ctrl-c deshabilitado use ctrl-\\ para terminar \n");

  for(;;);

  return 0;
}
```

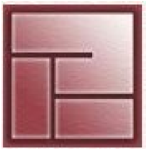


Alarm

- Envía, al proceso invocante, la señal SIG_ALRM pasados n segundos
- Prototipo

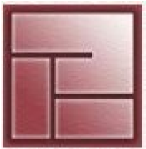
```
<unistd.h>
int alarm(unsigned int segundos)
```

- Retorna, -1 en error, en caso de éxito número de segundos pendientes de espera de la llamada previa a **`alarm()`**
- **`segundos`**=0, se cancela el envío registrado por el uso previo de **alarm** se cancela
  Las invocaciones no son acumulables, sólo se puede mantener una alarma por proceso en cada momento



- **Alarm**

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

void handler( int sig ) {
   printf("Recibida la alrma %d\n.", sig);
   exit(0);
}

int main() {
  signal(SIGALRM, handler);
  alarm(5);

  printf("zzzzzz\n");
  for (;;) ;
  return 0;
}
```



Pause

- Detiene al proceso invocante hasta que reciba una señal
- Prototipo:

```
int pause (void)
```

- Retorna, si lo hace, el valor -1 y establece errno a `EINTR`, indicando que la señal fue recibida y handled correctamente