LLamadas al Sistema Unix

Sistemas Operativos

Escuela de Ingeniería Civil Informática

Crear Procesos
fork(), getpid()

UNIVERSIDAD DEL BÍO-BÍO

¿Qué son las Llamadas al Sistema?

- Una llamada al sistema (system call):
  - Es una interface de programación para los servicios proveídos por el S.O.
  - Se utilizan para transferir el control entre el código del usuario y el del sistema (modo usuario, modo kernel).

  - Típicamente escrita en lenguajes de alto nivel (C o C++)
  - La mayoría de los programas las usan mediante APIs de alto nivel.
  - Las 3 APIs más comunes son:
    - Win32 para Windows
    - POSIX para Unix, Linux y Mac OS X
    - Java API para JVM

Formato General de las Llamadas al Sistema

- Una llamada al sistema:
  - Se invoca mediante una función.
  - Siempre retorna un valor (con información del servicio o del error que se ha producido en su ejecución).

- El valor de retorno puede ser ignorado, pero es recomendable siempre revisarlo.

- Cuando existe un error:
  - Se retorna un valor de -1.
  - Se revisa una variable externa *errno*, donde se indica un código de error más específico (*errno.h*).
  - La variable *errno* no cambia de valor después de una llamada al sistema que retorna con éxito, por tanto es importante tomar dicho valor justo después de la llamada al sistema y únicamente cuando éstas retornan error.

Tipos de Llamadas al Sistema

- Control de procesos.
- Manipulación de archivos.
- Manipulación de dispositivos.
- Mantenimiento de información.
- Comunicaciones.

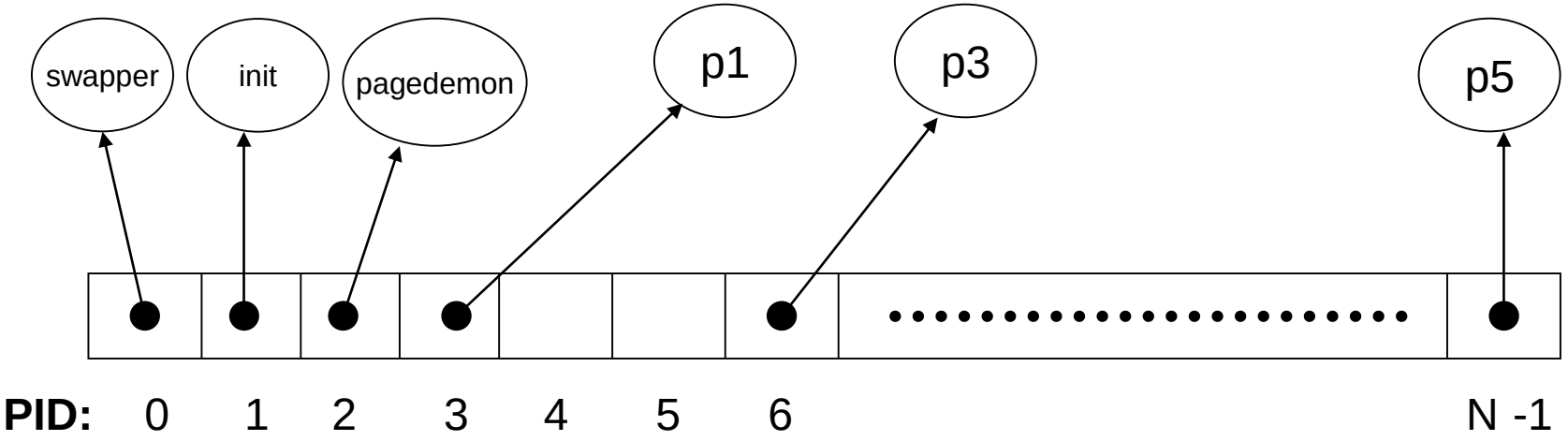
Crear un Proceso - *fork()*

- Es de vital importancia que Unix posea algún mecanismo para crear nuevos procesos.
- Existen 3 procesos creados por el kernel al iniciar el sistema (bootstrapping), ellos son:
  - **swapper** (demonio que maneja la memoria swap)
  - **init** (proceso padre de todos los demás)
  - **pagedemon** (demonio servidor de páginas de memoria virtual)
- Para que el usuario pueda crear un nuevo proceso, debe utilizar la llamada al sistema `fork()`.
- El proceso que crea uno nuevo se denomina “padre” y el nuevo “hijo”.

Archivo cabecera	`#include <sys/types.h>` `#include <unistd.h>`		
Formato	`pid_t fork(void);`		
Salida	Exito	Fallo	Valor en errno
	0 al hijo PID del hijo al padre	-1	Si

PID de un Proceso

- El valor retornado por *`fork()`* corresponde a un número entero único que identifica al proceso, llamado `PID`.
- Este `PID` es asignado por el kernel al proceso, para ello existe una tabla (o arrreglo) cuyos índices corresponde al `PID` asignado a cada proceso.

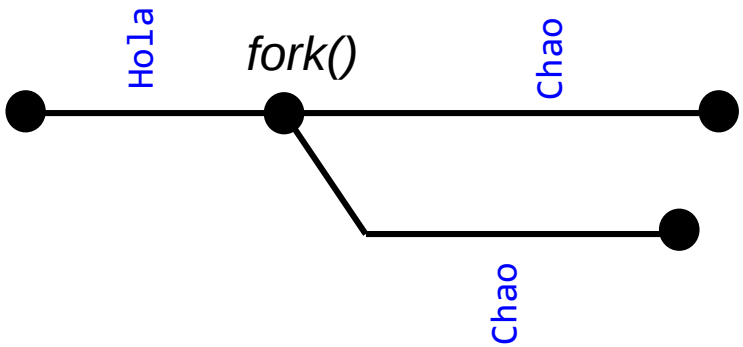


- Por defecto el sistema asigna las tres primeras celdas a los procesos creados al inicio, por lo tanto, esas celdas nunca se asignan a otro proceso de usuario. El resto se asigna secuencialmente.

Ejemplo 1: Crear un Proceso

- Código del proceso padre.

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

int main(void){
  printf("Hola\n");
  fork();
  printf("Chao\n")
  return 0;
}
```

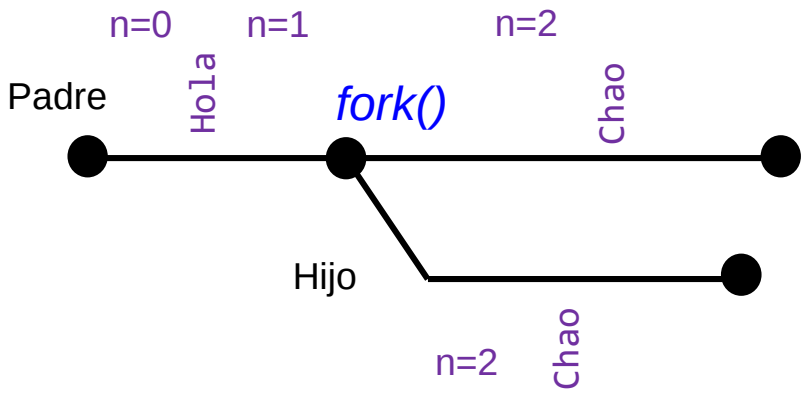


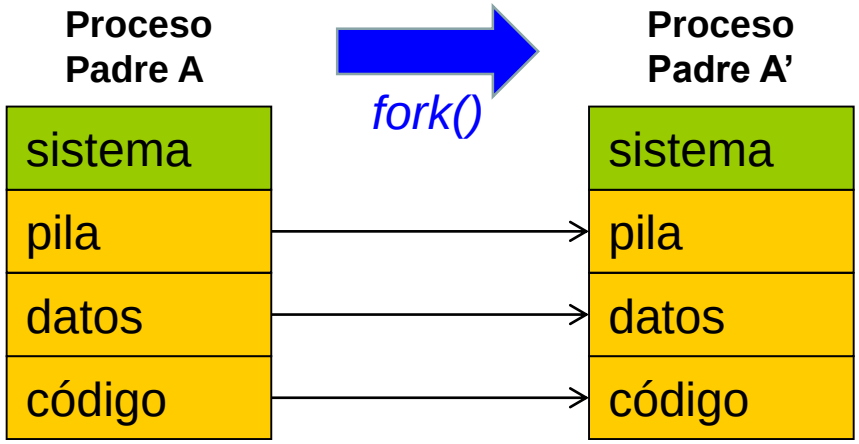
- Salida del programa:

```
$./a.out
Hola
Chao
Chao
```

¿Cómo funciona fork()?

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

int main(void){
  int n= 0;
  printf("Hola\n");
  n++;
  fork();
  n++;
  printf("Chao\n")
  return 0;
}
```





Los segmentos de pila, datos y código son idénticos. Al modificar algún segmento se duplica en otra zona de memoria (**copy-on-write**)

¿Cómo funciona fork()?

- `fork()` duplica un proceso, esto quiere decir que genera un clon con el mismo código ejecutable. Ambos procesos comparten:
  - Descriptores de archivos abiertos
  - Información del entorno
  - Valores de variables hasta ese punto
  - etc.
- Padre e hijo no comparten la misma zona de memoria, esto significa:
  - Los valores de las variables pueden cambiar de ahí en adelante.
  - La misma variable puntero, en ambos, no apunta a la misma zona de memoria.
- El proceso hijo también posee su propia información:
  - Tiene su propio PID (process ID)
  - Posee un identificador del padre (PPID)
  - Se resetea la información de uso de recursos del proceso (por ejemplo: tiempo de CPU asignada)
  - Se puede configurar a gusto la recepción de señales (interrupciones)

Obtener el PID de un Proceso

- Si un proceso, tal como un hijo, desea saber su `PID` puede utilizar la llamada al sistema *`getpid()`*:

Archivo cabecera	*#include <sys/types.h>* *#include <unistd.h>*		
Formato	*pid_t getpid(void);*		
Salida	Exito	Fallo	Valor en errno
	PID del proceso que la invoca	-1	Si

- Ejemplo:

```
printf("Mi PID es %d \n", getpid());
```

- Desafortunadamente **no hay llamada al sistema para obtener los PIDs de los hijos de un proceso determinado**, por lo cual los PIDs se deben almacenar al momento de crear cada proceso hijo.

Grupo de Procesos

- Cada proceso pertenece a un grupo de procesos que es identificado por un valor entero (`PGID`).
- Cuando un proceso genera procesos hijos, el sistema operativo automáticamente crea un grupo de procesos.
- El padre inicial es conocido como “líder”. El `PID` del proceso líder es el mismo que el `PGID`.
- Los grupos son útiles para enviar señales a varios procesos de una sola vez. Por ejemplo, una señal podría ser `kill`, la cual termina el proceso al cual se le envía.
- Un proceso puede saber su `PGID` mediante la llamada al sistema:

```
pid_t getpgid(pid_t  pid);
```

  - *`pid`* corresponde al `PID` del proceso del cual se desea saber su `PGID`. Si es cero se interpreta como el proceso actual.

Ejemplo 2: Grupo de Procesos

- Código del proceso padre.

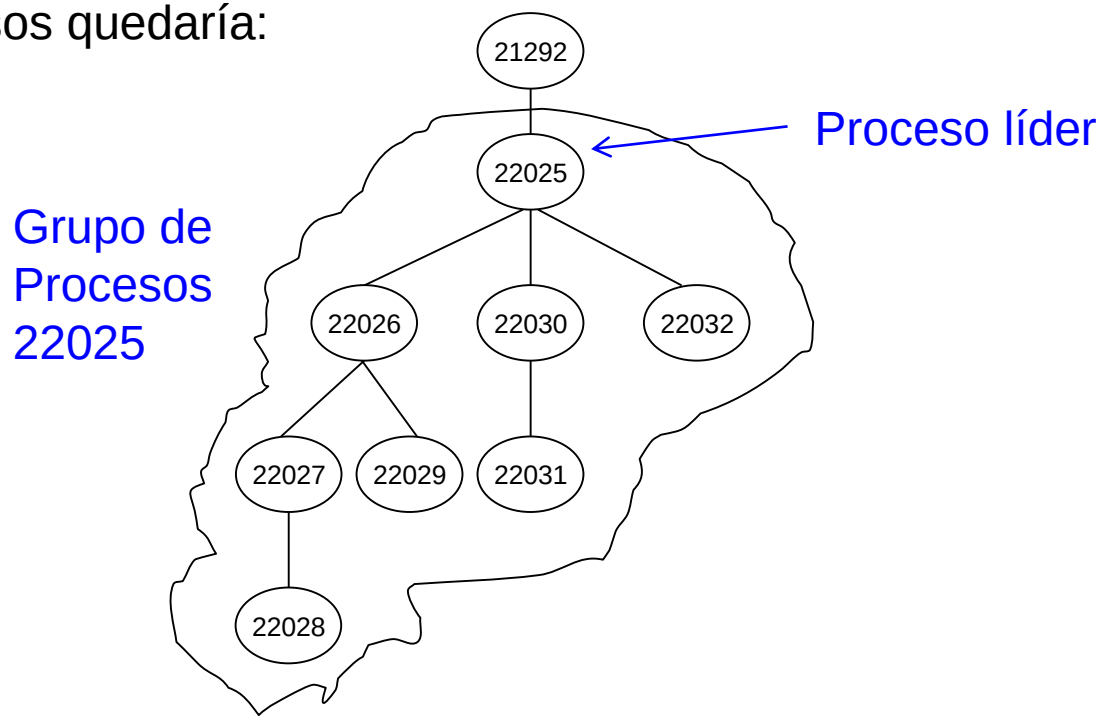
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void){
  int i;
  printf("\n\n Proceso inicial \t PID %6d \t PPID %6d \t GID %6d\n\n",
                                        getpid(), getppid(), getpgid(0));
  for(i=0; i<3; ++i)
     if (fork()==0)
        printf(("\n\n Nuevo proceso \t\t PID %6d \t PPID \t GID %6d\n\n",
                                        getpid(), getppid(), getpgid(0));
  return 0;
}
```

- Salida del programa:

```
Proceso inicial     PID    22025    PPID  21292   GID   22025
Nuevo proceso       PID    22026    PPID  22025   GID   22025
Nuevo proceso       PID    22027    PPID  22026   GID   22025
Nuevo proceso       PID    22029    PPID  22026   GID   22025
Nuevo proceso       PID    22028    PPID  22027   GID   22025
Nuevo proceso       PID    22030    PPID  22025   GID   22025
Nuevo proceso       PID    22031    PPID  22030   GID   22025
Nuevo proceso       PID    22032    PPID  22025   GID   22025
```

Grupo de Procesos

- El árbol de procesos quedaría:



- Un proceso puede cambiar su grupo mediante la llamada ***setpgid()***

propuesto: investigar

Terminar un Proceso

- Todo proceso que es creado, en algún momento debe terminar.
- Existen **3 maneras** por las cuales puede terminar un proceso:
  - En algún punto de su código de invoca la llamada al sistema `exit()`.
  - En algún punto del código se encuentra la instrucción `return()` para la función `main()`.
  - Por errores en el programa, con lo cual termina abruptamente por fallas en su funcionamiento.
- La llamada al sistema `exit()` permite dar término a un proceso:

Archivo cabecera	*#include <stdlib.h>*		
Formato	*void exit(int status);*		
Salida	Exito	Fallo	Valor en errno
	No retorna	No retorna	No

Terminar un Proceso…

- Cuando un proceso termina el sistema operativo realiza algunas labores:
  - Todos los descriptores de archivos abiertos son cerrados y su buffer asociado es vaciado.
  - El padre del proceso es notificado por medio de una señal (se verá más adelante) `SIGCHLD` (**SIG** = signal, **CHLD** = children) que el proceso hijo terminó.
  - El padre recibe **información de estatus** del hijo.
  - Todos los procesos hijos al terminar su padre cambian su `PPID` a 1 (**proceso init**).