Práctica 1: Procesos en Unix - Linux

75.59 - Técnicas de Programación Concurrente I

Ejercicios

1. Explicar qué se entiende por el *ambiente* de un proceso.
2. Escribir un programa para listar las variables de ambiente. Mostrar diferentes alternativas para lograr el resultado deseado.
3. Diseñar un TDA y su implementación en C para que actúe como una tabla de símbolos.
4. Escribir un programa que agrega una variable a la tabla de símbolos y la imprime.
5. *a*) ¿Cuáles son todos los *ids* asociados a un proceso Unix?
   b) El *i-nodo* de un archivo contiene los permisos del mismo. ¿Cuáles pueden ser los permisos de un archivo? Definirlos.
6. Escribir un programa que crea un proceso hijo. Tanto el proceso padre como el hijo están escribiendo algún mensaje al usuario en pantalla.
7. Escribir un programa que muestre la utilización de las llamadas `exit()` y `wait()`.
8. Explicar qué hace el siguiente código:

```
1  #include<stidio.h>
2
3  main () {
4          if ( fork() == 0 ) {
5                  printf ( "PID = %d, PPID = %d, ID de grupo = %d\n",getpid(),getppid(),
                       getpgrp() );
6                  sleep ( 10 );
7                  printf ( "PID = %d, PPID = %d, ID de grupo = %d\n",getpid(),getppid(),
                       getpgrp() );
8                  setpgrp ();
9                  printf ( "PID = %d, PPID = %d, ID de grupo = %d\n",getpid(),getppid(),
                       getpgrp() );
10                 exit ( 0 );
11         }
12
13         sleep ( 5 );
14         printf ( "PID = %d, PPID = %d, ID de grupo = %d\n",getpid(),getppid(),getpgrp() );
15         exit ( 0 );
16 }
```

9. Unix utiliza el tiempo de calendario tanto como el tiempo de ejecución. El tiempo de calendario se utiliza para el acceso, modificación y estado de los cambios de los archivos, para registrar el tiempo de acceso de un usuario, etc. Estudiar cuáles son los tipos de datos relacionados con el tiempo calendario y las siguientes funciones, elaborando un programa que muestre su aplicación: `time()`, `gettimeofday()`, `localtime()`, `gmtime()`, `ctime()`, `getdate()` y `difftime()`.
10. Las facilidades relacionadas con el tiempo de ejecución sirven para medir intervalos de tiempo y tiempos de ejecución de los procesos. Estudiar las funciones `clock()` y `time()`, así como también los tipos de datos `struct timespec` y `struct tms`. Ver la cabecera `<sys/times.h>`.

11. Escribir un programa que reciba, a través de la línea de comandos, la cantidad de procesos hijos que debe crear. Cada proceso hijo debe dormir un número aleatorio de segundos comprendido entre 0 y 30. El proceso padre debe esperar la terminación de cada uno de sus hijos. A medida que los hijos vayan terminando, el padre presentará en pantalla el *pid* de cada uno de ellos y la cantidad de segundos que ha estado durmiendo cada hijo.

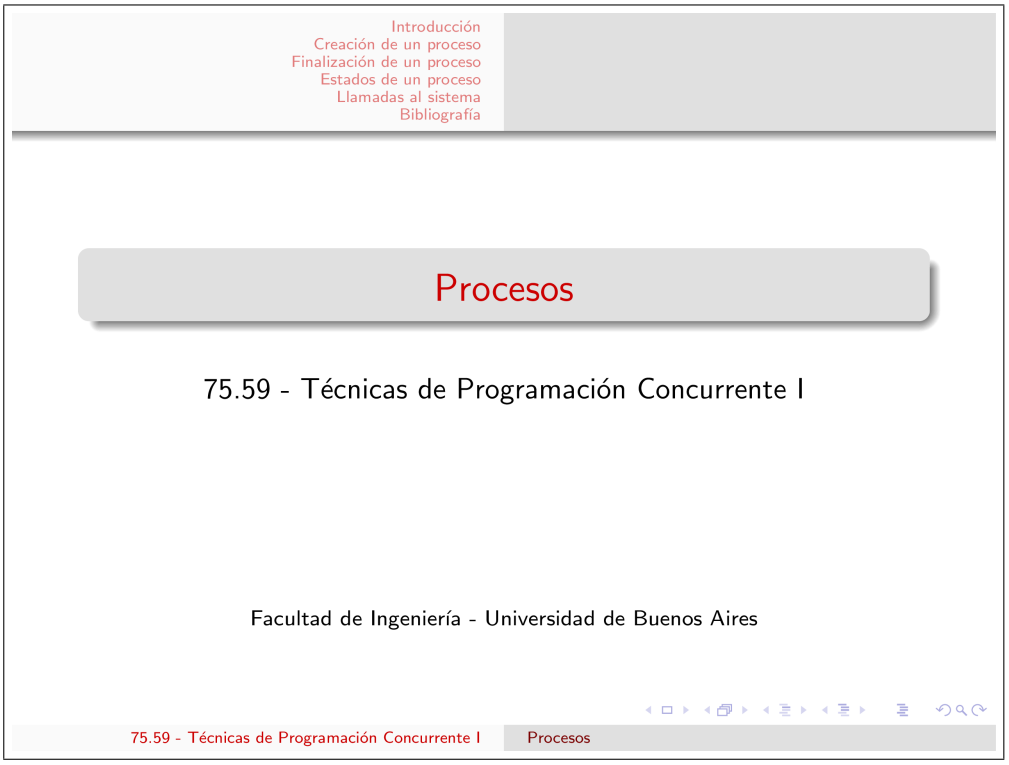
12. Escribir un programa que consulte el tamaño de un archivo y que genere un proceso que informe al usuario cuando el archivo supere un determinado tamaño. (Ayuda: estudiar `<sys/stat.h>`, en particular `int stat(const char* nombArch,struct stat* buf)` que obtiene información acerca del archivo apuntado por `nombArch` e `int fstat(int desearch,struct stat* buf)`, la cual obtiene la misma información que la anterior, pero sólo el archivo abierto con descriptor `desearch` (tal y como lo devuelve `open()`) es examinado.

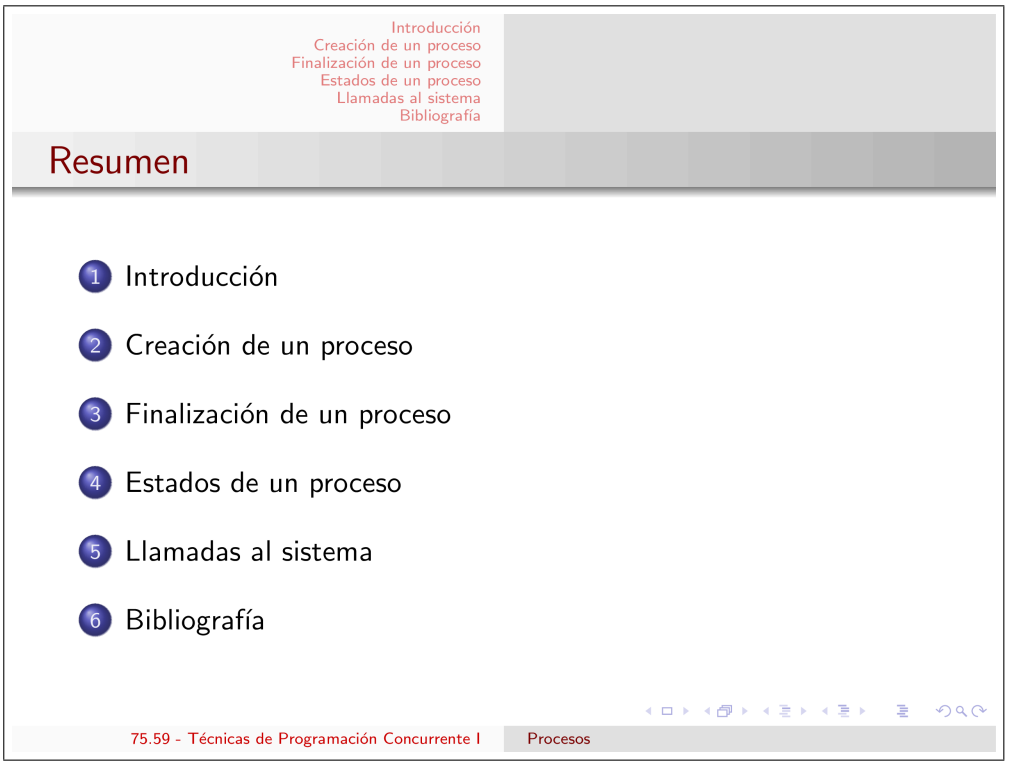
13. Explicar el siguiente código:

```
1  #include <fcntl.h>
2  int fdr, fdw;
3  char c;
4
5  main ( int argc, char *argv[] ) {
6          if ( argc != 3 )
7                  exit ( 1 );
8          if ( (fdr = open(argv[1],O_RDONLY)) == -1 )
9                  exit ( 1 );
10         if ( (fdw = creat(argv[2], 0666)) == -1 )
11                 exit ( 1 );
12         fork ();
13         rw ();
14         exit ( 0 );
15 }
16
17 void rw () {
18         for ( ;; ) {
19                 if ( read(fdr,&c,1) !=1 )
20                         return;
21                 write ( fdw,&c,1 );
22         }
23 }
```

14. Investigar llamadas al sistema similares a `exec()` y `fork()` en otro sistema operativo. Comparar sus características y destacar ventajas y desventajas.

Apuntes


Introducción
Creación de un proceso
Finalización de un proceso
Estados de un proceso
Llamadas al sistema
Bibliografía
Procesos
75.59 - Técnicas de Programación Concurrente I
Facultad de Ingeniería - Universidad de Buenos Aires
75.59 - Técnicas de Programación Concurrente I
Procesos


Introducción
Creación de un proceso
Finalización de un proceso
Estados de un proceso
Llamadas al sistema
Bibliografía
Resumen
1 Introducción
2 Creación de un proceso
3 Finalización de un proceso
4 Estados de un proceso
5 Llamadas al sistema
6 Bibliografía
75.59 - Técnicas de Programación Concurrente I
Procesos

Introducción (I)

Un proceso está formado por:

- Programa: instrucciones que conforman el programa a ejecutar
- Datos del usuario: espacio de memoria modificable por el usuario, por ejemplo: datos propios del programa, heap
- Pila del sistema: se utiliza para almacenar parámetros y direcciones de retorno durante el llamado a subrutinas
- Estructuras de datos del kernel:
  - Fila en la tabla de procesos: PCB (Process Control Block)

Introducción (II)

Información de control que el SO almacena en el *PCB*:

- Datos de identificación del proceso
  - Identificador del proceso
  - Identificador del proceso que lo creó
  - Identificador del usuario y del grupo
- Datos del estado del proceso: permite suspender y retomar el proceso
  - Registros de propósito general de la CPU
  - Stack pointer
  - Instruction pointer
- Datos de control del proceso
  - Estado del proceso y otra información de *scheduling* (p.ej. prioridad)
  - Estructura del proceso: identificadores de los hijos
  - Datos de IPC (señales, mensajes)
  - Contadores de tiempo de uso de CPU

Creación de un proceso

¿En qué momento se crea un proceso?

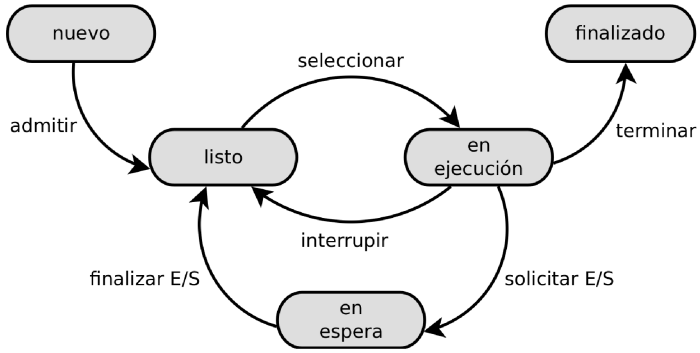
- Inicialización del sistema
- Ejecución de una llamada al sistema por parte de otro proceso que está en ejecución (en breve veremos cuál es)
- Por pedido del usuario
- Inicio de un *batch job*

Finalización de un proceso

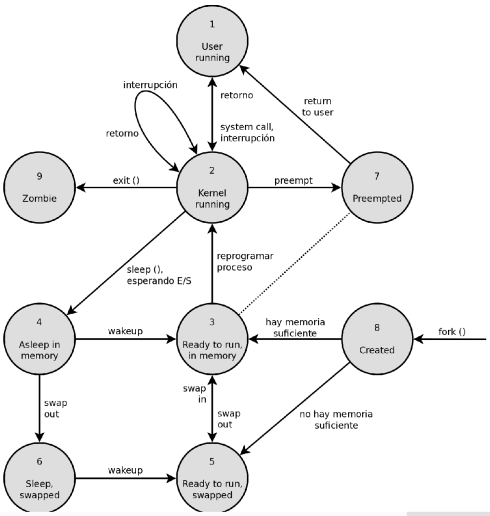
¿En qué condiciones puede finalizar un proceso?

- Finalización normal (voluntario): el proceso terminó su tarea
- Finalización con error (voluntario)
- Error fatal (involuntario): generalmente debido a un error de programación
- Finalizado por otro proceso (involuntario): comando *kill*

Estados de un proceso (genérico)



Estados de un proceso en UNIX



1: El proceso se está ejecutando en modo usuario

2: El proceso se está ejecutando en modo kernel

3: El proceso no se está ejecutando, pero está listo para ejecutarse ni bien el kernel lo programe

4: El proceso está durmiendo y reside en la memoria principal

5: El proceso está listo para ejecutarse, pero debe ser swappeado a memoria principal antes que el kernel pueda programarlo

6: El proceso está durmiendo y es swappeado a memoria secundaria

7: El proceso está retornando a modo usuario pero el kernel lo interrumpe y realiza un cambio de contexto para programar otro proceso

8: El proceso está creado pero aún no está listo para ejecutarse

9: El proceso ejecutó la system call exit() y ahora es un zombie. El proceso ya no existe, pero deja un registro que contiene el código de terminación para que lo lea el padre

Llamadas al sistema (I)

Creación de un proceso: función *fork()*

- Sirve para crear un proceso realizando una copia del proceso que la invoca. El proceso que la invoca se llama *padre*, mientras que el proceso creado se llama *hijo*.
- Cuando se llama a `fork()`, el kernel realiza lo siguiente:
  1. Reserva un lugar para el hijo en su tabla de procesos
  2. Asigna un identificador único al proceso hijo (*process id*)
  3. Realiza una copia del contexto del proceso padre
  4. Retorna el identificador del proceso hijo al padre, y cero al hijo
- Al finalizar la llamada, ambos procesos tienen una copia exacta del contexto, excepto por el valor de retorno

Llamadas al sistema (II)

Creación de un proceso: función *fork()*

- Algunas propiedades que el hijo *hereda* del padre:
  - *User id* real y efectivo
  - Grupo de procesos
  - Valor de *nice*: usado para calcular la prioridad del proceso
  - Directorio actual
  - Directorio raíz: si el padre ejecutó `chroot()` para cambiar su directorio raíz, el hijo hereda el directorio raíz cambiado
  - Tabla de archivos abiertos

Llamadas al sistema (III)

Ejecución de otro programa: función *exec()* (toda la familia)

- Reemplaza el espacio de memoria (código y stack) del proceso actual con una copia de un archivo ejecutable
- Luego de llamar a *exec()*, el espacio de memoria anterior ya no es accesible
- Suele usarse en conjunto con *fork()*

Otras funciones:

- `sleep()`: el proceso suspende su ejecución hasta que pase el tiempo dado
- `getpid()`: retorna el *process id* del proceso que la llama
- `getppid()`: retorna el *process id* del proceso padre

Llamadas al sistema (IV)

Finalización de un proceso: función *exit()*

- El valor del estado (parámetro de `exit()`) se retorna al proceso padre
- Si el proceso creó hijos, el kernel cambia el padre de los hijos al proceso *init*
- El kernel libera todos los recursos asignados al proceso
- El proceso pasa a estado *zombie*: queda el registro en la tabla de procesos del kernel
- Cuando el padre lee estado mediante *wait()*, el registro se elimina

Llamadas al sistema (V)

Esperar por la finalización de un proceso: funciones *wait()* y *waitpid()*

- Permiten sincronizar la ejecución del padre con los hijos
- El padre suspende su ejecución hasta que finalice la ejecución de alguno de sus hijos (*wait()*) o hasta que termine algún hijo en particular (*waitpid()*)
- El padre puede obtener el código de finalización del hijo y conocer el motivo de la finalización
- El kernel busca en la tabla de procesos un hijo del proceso que esté en estado *zombie* y retorna el código de finalización

Bibliografía

- *Modern Operating Systems*, Andrew S. Tanenbaum, Cuarta edición
- *Sistemas Operativos*, William Stallings, Segunda edición
- *The Design of the Unix Operating System*, Maurice Bach
- Manuales del sistema operativo

Fuentes de los ejemplos

Listado 1: Crear un proceso

```
#ifdef EJEMPLO_1

#include <iostream>
#include <unistd.h>
#include <stdlib.h>

using namespace std;

int main () {

        pid_t id = fork ();

        if ( id == 0 ) {

                cout << "Hijo: Hola, soy el proceso hijo.  Mi process ID es " << getpid() << endl;
                cout << "Hijo: El process ID de mi padre es " << getppid() << endl;

                exit ( 0 );

        } else {

                cout << "Padre: Hola, soy el proceso padre.  Mi process ID es " << getpid() << endl;
                cout << "Padre: El process ID de mi hijo es " << id << endl;

                exit ( 0 );
        }
}

#endif
```

Listado 2: Crear un proceso y sincronizar con wait()

```
#ifdef EJEMPLO_2

#include <iostream>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <sys/wait.h>

using namespace std;

int calcularRandom ();


int main () {

        pid_t id = fork ();

        if ( id == 0 ) {

                int tiempoAdormir = calcularRandom ();
                cout << "Hijo: Hola, soy el proceso hijo.  Voy a dormir " << tiempoAdormir << " segundos" << endl;
                sleep ( tiempoAdormir );
                cout << "Hijo: ya me desperte y termino" << endl;

                exit ( 0 );

        } else {

                cout << "Padre: Hola, soy el proceso padre.  Espero a que mi hijo se despierte" << endl;
                int estado;
                wait ( (void*) &estado );
                cout << "Padre: mi hijo se desperto y termino" << endl;

                exit ( 0 );
        }
}


int calcularRandom () {
        srand ( time(NULL) );
```

```
41          int resultado = rand() % 10 + 1;
42          return resultado;
43  }
44
45  #endif
```

Listado 3: Crear dos procesos hijos (forma incorrecta)

```
1   #ifdef EJEMPLO_3
2
3   #include <unistd.h>
4   #include <stdlib.h>
5   #include <iostream>
6
7   using namespace std;
8
9
10  int main () {
11
12          cout << "Proceso padre: " << getpid() << endl;
13          pid_t id1 = fork ();
14
15          if ( id1 == 0 ) {
16                  cout << "Hijo: Hola, soy primer el proceso hijo.  Mi process ID es " << getpid
                        () << " (" << getppid() << ")" << endl;
17          }
18
19          pid_t id2 = fork ();
20
21          if ( id2 == 0 ) {
22                  cout << "Hijo: Hola, soy segundo el proceso hijo.  Mi process ID es " << getpid
                        () << " (" << getppid() << ")" << endl;
23          }
24
25  }
26
27  #endif
```

Listado 4: Crear dos procesos hijos (una de las formas correctas)

```
1   #ifdef EJEMPLO_4
2
3   #include <unistd.h>
4   #include <stdlib.h>
5   #include <iostream>
6
7   using namespace std;
8
9
10  int main () {
11
12          cout << "Proceso padre: " << getpid() << endl;
13          pid_t id1 = fork ();
14
15          if ( id1 == 0 ) {
16                  cout << "Hijo: Hola, soy primer el proceso hijo.  Mi process ID es " << getpid
                        () << " (" << getppid() << ")" << endl;
17          } else {
18
19                  pid_t id2 = fork ();
20
21                  if ( id2 == 0 ) {
22                          cout << "Hijo: Hola, soy segundo el proceso hijo.  Mi process ID es "
                                << getpid() << " (" << getppid() << ")" << endl;
23                  }
24          }
25
26  }
27
28  #endif
```

Listado 5: Utilizar `new` y `delete` (forma incorrecta)

```
1   #ifdef EJEMPLO_5
2
3   #include <iostream>
4   #include <stdlib.h>
5   #include <unistd.h>
6
```

```
7  using namespace std;
8
9  int main () {
10         int* valor = new int;
11
12         pid_t pid = fork ();
13
14         if ( pid == 0 ) {
15                 *valor = getpid ();
16                 cout << "El valor del pid es " << *valor << endl;
17                 exit ( 0 );
18         } else {
19                 *valor = getpid ();
20                 cout << "El valor del pid es " << *valor << endl;
21                 delete valor;
22                 exit ( 0 );
23         }
24 }
25
26 #endif
```

Listado 6: Utilizar `new` y `delete` (forma correcta)

```
1  #ifdef EJEMPLO_6
2
3  #include <iostream>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  using namespace std;
8
9  int main () {
10         int* valor = new int;
11
12         pid_t pid = fork ();
13
14         if ( pid == 0 ) {
15                 *valor = getpid ();
16                 cout << "El valor del pid es " << *valor << endl;
17                 delete valor;
18                 exit ( 0 );
19         } else {
20                 *valor = getpid ();
21                 cout << "El valor del pid es " << *valor << endl;
22                 delete valor;
23                 exit ( 0 );
24         }
25 }
26
27 #endif
```