Sistemas Operativos
Proyecto 1

Fernando Sisul (LU: 81236) Tomás Touceda (LU: 84024)

29 de Octubre de 2010

Índice general

Capítulo 1

Preguntas teóricas

1.1. Windows

1.1.1. Creación de procesos

Los procesos en Windows se crean utilizando la función CreateProcess, a esta se le pasa en los parámetros el nombre del binario a ejecutar en el nuevo proceso junto con diferentes flags para especificar distintas características del mismo, como atributos del proceso, atributos de los hilos que crea el proceso, el entorno del proceso, y el directorio actual donde se encontrará el proceso. La llamada devuelve en el último parámetro un puntero a una estructura que contiene información del proceso creado, y retorna cero si falla, o un número distinto de cero en caso contrario.

La diferencia más grande con la creación de procesos en sistemas UNIX es que en Windows se utiliza una función wrapper de las system calls específicas (CreateProcess) y en sistemas UNIX se deben ejecutar una serie definida de llamadas al sistema. Esta diferencia hace que la creación de procesos en Windows sea transparente a cualquier cambio en la definición de las system calls que suceda entre versiones del sistema operativo.

1.1.2. Visualización de procesos

Los procesos en los sistemas Windows se pueden visualizar utilizando el Administrador de tareas que provee el sistema, este se activa con las teclas Ctrl+Alt+Del, y permite manipular de forma básica todos los procesos que se encuentren ejecutándose al momento en que se lanza el Administrador de tareas.

1.1.3. Sysinternals Process Monitor

La aplicación Process Monitor es una herramienta de monitoreo avanzada para Windows que muestra en tiempo real la actividad del sistema de archivos, el registro del sistema y de los procesos e hilos en ejecución. Esta herramienta provee funcionalidades para visualizar información de procesos, los stacks de cada hilo, realizar logs en archivos.

1.1.4. Sysinternals Process Explorer

Process Explorer muestra información acerca de los recursos y bibliotecas dinámicas que utilizan los distintos procesos en ejecución, además también muestra los archivos que cada proceso ha abierto.

1.2. Linux

1.2.1. Directorio /proc

El sistema de archivos proc contiene una jerarquía de archivos que representan el estado actual del kernel y sus estructuras de datos. Este tipo de sistema de archivos es denominado sistema de archivos virtual. Dentro del directorio /proc se puede encontrar información detallada del hardware del sistema y de los procesos que estan ejecutandose actualmente. Este contenido no se guarda en ningún dispositivo fisico sino que se construye dinámicamente cada vez que se solicita al kernel que lo muestre o cuando queremos visualizar el contenido de sus archivos o directorios. Es por esto que estos datos no existen una vez que el equipo se apaga.

1.2.2. Creación de procesos

1. fork()

```
pid_t fork(void);
```

   Crea un nuevo proceso duplicando el proceso invocador, referido como proceso padre. El nuevo proceso, referido como proceso hijo, es un duplicado exacto del proceso padre, exceptuando, entre otras, el process ID, el parent process ID, locks de memoria del padre, etc.

2. clone()

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

   Crea un nuevo proceso, de manera similar a fork(). A diferencia de fork(), esta llamada permite al proceso hijo compartir parte de su contexto de ejecución con el proceso padre, como el espacio de memoria, tabla de descriptores de archivos y tabla de manejadores de señales.

1.2.3. Manipulación de procesos

1. signal()

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

   Establece la disposicion de la señal signum al controlador handler, que puede ser SIG_IGN (ignora la señal), SIG_DFL (realiza la accion default asociada con la señal), o la direccion de una funcion definida por el programador (un manejador de señales).

1.2.4. System calls fork y exec

Las diferencia entre los system calls fork y exec es que la primera realiza lo explicado en el inciso anterior y la familia de funciones exec() reemplaza la imagen del proceso actual por una nueva imagen de proceso. Hay una gran variedad de funciones exec:

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

El primer argumento de estas funciones es la ruta al archivo que va a ser cargado en memoria y ejecutado, el resto varia de función a función.

Capítulo 2

Algoritmos

2.1. Consideraciones iniciales

Para compilar cada ejercicio se deberá ejecutar un comando dentro de la carpeta src de la forma:

```
cd ejercicio4a; make
```

siendo "ejercicio4a" el ejercicio en cuestión.

2.2. Creación de procesos

2.2.1. Ejercicio 4A

Modo de uso

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio4A".

La aplicación escribe dentro de un archivo llamado "output_ejercicio4A", en él cada proceso escribe un mensaje identificándose a sí mismo.

El orden de finalización de cada proceso es completamente aleatorio (dependiente del scheduler y del resto de los procesos que se encuentran ejecutándose en ese momento), ya que en el código no se utiliza explícitamente ninguna función para que un proceso espere a otro.

Algoritmo

```
Inicializar mutex en 1;
fork();
if(es_hijo) {
   Escribir en el archivo la identidad del primer hijo
} else {
   fork();
   if(es_segundo_hijo)
      Escribir en el archivo la identidad del segundo hijo
   else
      Escribir en el archivo la identidad del padre
}
```

2.2.2. Ejercicio 4B

Modo de uso

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio4B".

El orden de finalización de cada hilo es completamente aleatorio (dependiente del scheduler y del resto de los procesos que se encuentran ejecutándose en ese momento), el proceso padre va a finalizar luego de los hilos ya que explícitamente el padre espera a que terminen.

Algoritmo

```
Proceso
Crear los threads
Escribir en la consola que es el padre
Esperar que terminen los hilos

Hilos
Escribir en la consola que numero de hijo es
```

2.2.3. Ejercicio 4C

Modo de uso

En este ejercicio el Makefile no solo se genera un ejecutable llamado ejercicio4C, sino que también se genera uno llamado "job4c" que será utilizado por el primero y no deberá ser llamado explícitamente.

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio4C".

La aplicación escribe dentro de un archivo llamado "salida.txt", en el que se operará como se especificó en el enunciado del ejercicio.

El orden de finalización de cada proceso hijo es completamente aleatorio, pero el proceso padre, al utilizar la función wait, espera a que terminen sus hijos antes de finalizar.

Algoritmo

```
Crear el archivo "salida.txt" o eliminar su contenido
Crear la memoria compartida
Asignar el espacio de memoria al semaforo
Inicializar el semaforo en 1
for(i=0 hasta i=2){
   Creo un hijo
   if (es_el_hijo){
      Reemplazo la imagen ejecutable actual por la de job4c
   }
}
Esperar a que terminen los hijos
Abrir el archivo para escritura sin borrar el contenido
Escribir en el mismo que se finalizo la actividad

job4c
Acceder al espacio de memoria compartida
Asignar el espacio de memoria al mutex
for(i=0 hasta i=9999){
   Esperar el semaforo compartido
   Escribir en el archivo y en la consola
   Liberar el semaforo compartido
}
```

2.2.4. Ejercicio 5A

Modo de uso

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio5A".

ACLARACIÓN: Como ya se habló con ayudantes de la cátedra, si bien el código cicla 10000 veces imprimiendo cada letra, como se especifica en el enunciado, por alguna razón cuya respuesta no se encontró, algunas letras se imprimen un número arbitrario más de veces.

Algoritmo

```
void main() {
   Crear los 5 threads para que utilicen la funcion "thread()";
   fork();
   if(es_proceso_hijo)
      Imprimir un mensaje indicando la creacion existosa;
   else
      Esperar a que terminen los threads;
}

void thread(letra) {
   Imprimir 10000 veces letra;
}
```

2.2.5. Ejercicio 5B

Modo de uso

En este ejercicio, el Makefile genera los siguientes binarios: ejercicio5B1, ejercicio5B2, ejercicio5B3, que corresponden a cada inciso del ejercicio.

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio5B1", o "./ejercicio5B2", "./ejercicio5B3".

Algoritmo

Inciso 1:

```
Proceso
Definir un arreglo de 5 semaforos {A,B,C,D,E} tratada como cola circular
Inicializar los semaforos de la siguiente manera A=1, B=0, C=0, D=0, E=0
Crear los 5 hilos asignandoles la misma funcion


Hilos
for(){
   Esperar mi semaforo
   Imprimir mi letra
   LIbera el semaforo siguiente
}
```

Inciso 2:

```
/* Se utiliza una variable que define el turno en el que se encuentra la
   secuencia.
   turno = 1 -> Tengo que imprimir 2 veces (B o C)
   turno = 0 -> (B o C) ya se imprimio una vez, tengo que imprimirla una
                vez mas y continuar con la secuencia normalmente */
Proceso
Definir un arreglo de 4 semaforos {A,B,D,E} tratada como cola circular
```

```
Inicializar los semaforos de la siguiente manera A=1, B=0, D=0, E=0
Crear los 5 hilos asignandoles la misma funcion


Hilos
/* B y C comparten semaforo ya que compiten por imprimir */
for(){
    Esperar mi semaforo
    Imprimir mi letra
    Dependiendo del turno en el que me encuentre liberar el semaforo
}
```

Inciso 3:

```
/* Se utilizan 2 variables que definen el turno en el que se encuentra la
    secuencia. Dadas las 3 condiciones que se pueden dar
    (turno, turnoD)
    (0, 0) -> El proximo turno no es de D, ni hay que imprimir 2 veces (A o B)
    (0, 1) -> El proximo turno no es de D, pero si hay que imprimir 2 veces (A o B)
    (1, 1) -> El proximo turno es de D, pero tengo que imprimir una vez mas (A o B) */

Proceso
Definir un arreglo de 4 semaforos {A,C,D,E} tratada como cola circular
Inicializar los semaforos de la siguiente manera A=1, C=0, D=0, E=0
Crear los 5 hilos asignandoles la misma funcion


Hilos
/* A y B comparten semaforo ya que compiten por imprimir */
for(){
    Esperar mi semaforo
    Imprimir mi letra
    Dependiendo del turno en el que me encuentre liberar el semaforo
}
```

2.2.6. Ejercicio 6A

shmget

```
int shmget(key_t key, size_t size, int shmflg);
```

Crea un nuevo segmento de memoria compartida con la key y el tamaño que se especifica por parámetro.

shmat

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Se enlaza al segmento de memoria compartida identificado por el shmid que se pasa por parámetro.

shmdt

```
int shmdt(const void *shmaddr);
```

Desenlaza el segmento de memoria compartida ubicado en la dirección especificada en el parámetro shmaddr.

2.2.7. Ejercicio 6B

Modo de uso

En este ejercicio, el Makefile genera los siguientes binarios: ejercicio6B y ejercicio6B_2.

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio6B", y luego en otra instancia de la consola en mismo directorio "./ejercicio6B_2". No importa el orden en que se ejecuten, ejercicio6B escribe primero un mensaje y luego espera la respuesta, y ejercicio6B_2 primero lee un mensaje y luego escribe la respuesta. Este comportamiento se realiza infinitas veces.

Para comprobar que el envío de mensajes se realiza correctamente estos poseen un id aleatorio para identificarlos.

Algoritmo

NOTA: Como ambos programas funcionan de la misma forma, simplemente se invierte el comportamiento inicial en el envío de mensajes, solo se presenta el algoritmo para ejercicio6B.c, y simplemente invirtiendo el orden en que se envia y reciben mensajes se puede obtener el algoritmo de ejercicio6B_2.

```
Se crea el segmento de memoria compartida para los semaforos utilizados o
se enlaza al mismo si ya esta creado;

Si fue creado, se inicializan los semaforos;

Se crea el segmento de memoria compartida para el buffer de mensajes o
se enlaza al mismo si ya esta creado;

id = numero random;
Se espera a que el otro proceso haya leido el mensaje enviado;
Se envia el mensaje con el id aleatorio;
Se espera a que el otro proceso responda el mensaje;
Se lee la respuesta;
```

2.2.8. Ejercicio 6C

msgget

```
int msgget(key_t key, int msgflg);
```

Este system call devuelve el identificador de la cola de mensajes asociado al argumento key. Una nueva cola de mensajes se crea si key tiene el valor IPC_PRIVATE o key no es IPC_PRIVATE, no existe una cola de mensajes con llave key e IPC_CREAT es especificado en msgflag.

msgsnd

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Este system call se utiliza para enviar un mensaje a la cola de mensajes. El proceso invocador tiene que tener permisos de escritura en la cola de mensajes para poder enviarlo. Este anexa una copia del mensaje apuntado por msgp a la cola identificada por msqid.

msgrcv

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Este system call se utiliza para leer mensajes de la cola de mensajes. El proceso invocador tiene que tener permisos de lectura en la cola de mensajes para poder leerlo. Este anexa el mensaje al buffer msgp el mensaje leido de la cola de mensajes identificada por msqid. El argumento msgtyp especifica el tipo de mensaje que se lee.

2.2.9. Ejercicio 6D

Modo de uso

Binarios generados por el Makefile: ejercicio6D1 y ejercicio6D2.

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio6D1", y luego en otra instancia de la consola en mismo directorio "./ejercicio6D2". No importa el orden en que se ejecuten, cada binario se encarga de la sincronización completa.

Algoritmo

```
ejercicio6D1
Intentar crear o acceder a la cola de mensajes
Enviar un mensaje
Esperar a recibir una respuesta
Mostrar la respuesta recibida

ejercicio6D2
Intentar crear o acceder a la cola de mensajes
Esperar a recibir un mensaje
Mostrar el mensaje recibido
Enviar un mensaje de respuesta
```

2.2.10. Ejercicio 6E1

Modo de uso

Binarios generados por el Makefile: ejercicio6E1_productor y ejercicio6E1_consumidor.

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio6E1_productor", y luego "./ejercicio6E1_consumidor". El orden es indistinto, ya que la sincrionización se encargará de mantener el comportamiento de las aplicaciones consistente.

NOTA: Dado que se utiliza memoria compartida, y el manejo del buffer de mensajes se realiza en forma de pila para facilitar el consumo de mensajes y la producción.

Algoritmo

```
productor() {
   Se crea el segmento de memoria compartida para los semaforos utilizados o
   se enlaza al mismo si ya esta creado;

   Si fue creado, se inicializan los semaforos;

   Se crea el segmento de memoria compartida para el buffer de mensajes o
   se enlaza al mismo si ya esta creado;

   while(1) {
      id = numero random;
      Se espera a que haya un lugar vacio para producir;
      Se produce un mensaje con id aleatorio;
      Se avisa que hay un nuevo mensaje producido;
   }
}
```

```
consumidor() {
   Se crea el segmento de memoria compartida para los semaforos utilizados o
   se enlaza al mismo si ya esta creado;

   Si fue creado, se inicializan los semaforos;

   Se crea el segmento de memoria compartida para el buffer de mensajes o
   se enlaza al mismo si ya esta creado;

   while(1) {
      Se espera a que haya un mensaje para consumir;
      Se consume el mensaje;
      Se avisa que esta ese lugar vacio;
   }
}
```

2.2.11. Ejercicio 6E2

Modo de uso

Binarios generados por el Makefile: ejercicio6E2_productor y ejercicio6E2_consumidor.

Dentro del directorio donde se compiló el ejercicio, para ejecutarlo se podrá hacerlo con el comando "./ejercicio6E2_productor", y luego "./ejercicio6E2_consumidor". El orden es indistinto, ya que la sincrionización se encargará de mantener el comportamiento de las aplicaciones consistente.

NOTA: Dado que se utiliza una cola de mensajes, y el manejo del buffer de mensajes se realiza en forma de cola para facilitar el consumo de mensajes y la producción aprovechando las herramientas que provee la librería standard de C.

```
Ambos programas intentan crear o acceder a una cola de mensajes compartida
y a un espacio de memoria compartido para los semaforos.
La cola de mensajes se supone circular, es decir, el productor produce los
mensajes en 1,2,... BUFFSIZE,1,2,... y el consumidor consume los mensajes en ese
mismo orden.

ejercicio6E2_productor
Enviar un mensaje que confirme la inicializacion
for(){
   Esperar el semaforo que indica cuanto espacio libre hay en la cola de mensajes
   Produzco un mensaje
   Aumento el semaforo que representa cuantos mensajes hay en la cola de mensajes
   Aumento el indice de escritura en 1 y calculando este valor modulo BUFFSIZE + 1 (
       para evitar el 0)
}

ejercicio6E2_consumidor
Esperar un mensaje que confirme la inicializacion
for(){
   Esperar el semaforo que indica cuantos mensajes hay en la cola de mensajes
   Consumo el mensaje
   Aumento el semaforo que representa cuanto espacio libre hay en la cola de mensajes
   Aumento el indice de lectura en 1 y calculando este valor modulo BUFFSIZE + 1 (para
       evitar el 0)
}
```