Tarea 2: Multiestructuras

Programación 3
Instituto de Computación
Facultad de Ingeniería de la Universidad de la República
Curso 2010
Versión 1.1

1. Objetivos

- Manejo de estructuras de datos avanzadas.
- Implementación eficiente de operaciones.
- Manejo de memoria dinámica y punteros.

2. Conocimientos previos

- Metodología de programación estructurada.
- Estructuras de datos avanzadas.
- Análisis de algoritmos.
- Conocimientos básicos de C*.

3. Descripción del problema

Un Sistema Operativo (SO) es el software que interactúa entre el hardware y las aplicaciones de los usuarios. Su responsabilidad es gestionar los distintos recursos de la computadora. Las estructuras internas que posee son variadas y se manejan de forma muy eficiente, por estos motivos es de interés modelar una versión simplificada de un SO.

Se modelarán los siguientes conceptos: Programa, Proceso, Recurso y Procesador.

1. Un programa es un código ejecutable (una lista acotada de instrucciones) con un nombre asociado.
2. Un proceso es un programa en ejecución, y sólo puede estar en uno de los siguientes estados: *listo*, *corriendo* y *bloqueado*.
   a) Un proceso está en estado *listo* cuando un procesador puede ejecutar su próxima instrucción.

   b) Un proceso está en estado *corriendo* cuando está siendo ejecutado por un procesador.
   c) Un proceso está en estado *bloqueado* cuando está esperando por un recurso al cual trató de acceder pero éste estaba siendo usado por otro proceso.

3. Un recurso puede ser cualquier dispositivo de la computadora, como un teclado o una impresora.

4. Un procesador es quién ejecuta efectivamente el código de un programa, instrucción por instrucción, el procesador no es un recurso.

Nuestro SO debe soportar cualquier cantidad de programas, procesos y procesadores. Los recursos son acotados y son identificados por una letra de la A a la Z, por lo tanto hay 26 como máximo.

3.1. Instrucciones

Las instrucciones son variables de tipo char y son las siguientes:

- '!': El proceso deja el procesador actual y cambia su estado a *listo*.
- '@': El proceso termina la ejecución y se elimina. Debe liberar todos los recursos que posee.
- '_': No hace nada.

La letra de un recurso en mayúscula significa el pedido de obtención del recurso, en minúscula lo libera. Entonces:

- 'A' .. 'Z': Pide el recurso nombrado. Si algún otro proceso lo tiene, debe cambiar su estado a *bloqueado* hasta que se lo den.
- 'a' .. 'z': Libera el recurso nombrado, y se lo otorga al primer proceso que lo había pedido después de él.

3.2. Especificación del TAD SO

- *iniciar_SO*: Inicializa las estructuras internas. Se debe pasar la cantidad de procesadores disponibles. Los identificadores de los procesadores empiezan en 0 y son consecutivos en forma creciente.
- *crearPrograma_SO*: Crea un programa en base al nombre y al código pasados. Esta operación debe ejecutar en $O(log(A))$ peor caso. Siendo *A* la cantidad de programas en el sistema.
- *imprimirProgramas_SO*: Imprime en pantalla los nombres de los programas en orden lexicográfico, cada línea debe contener el nombre del programa. Esta operación debe ejecutar en $O(A)$ peor caso. Siendo *A* la cantidad de programas en el sistema.

- *ejecutar_SO*: Crea un proceso en base al nombre de un programa y lo pone en estado *listo*. El identificador del proceso empieza en 1 y se va asignando en forma creciente. Esta operación debe ejecutar en $O(log(A))$ peor caso. Siendo *A* la cantidad de programas en el sistema.

- *imprimirProcesosBloqueados_SO*: Imprime en pantalla los procesos en estado *bloqueado*. Esta operación debe ejecutar en $O(B)$ peor caso, siendo *B* la cantidad de procesos en estado bloqueado.

  La impresión se realiza en el orden cronológico definido por cuando el proceso cambió de estado, respetando la siguiente sintaxis:

  ```
  PID=%A Archivo=%B:%C Esperando=%D Tiene=%E
  ```

  - %A se sustituye por el identificador del proceso.
  - %B se sustituye por el nombre del archivo.
  - %C se sustituye por el número de línea a ejecutar. Las lineas de los archivos empiezan a numerarse desde 1.
  - %D se sustituye por el nombre (en minúscula) del recurso.
  - %E se sustituye por la lista de nombres de recursos (en minúscula) separados por coma. Se respeta el orden en el cual fueron pedidos.

  Por ejemplo:

  ```
  PID=1 Archivo=icq.exe:4 Esperando=d Tiene=a,b,c
  ```

- *imprimirProcesosListos_SO*: Imprime en pantalla los procesos en estado *listo*. Esta operación debe ejecutar en $O(L)$ peor caso, siendo *L* la cantidad de procesos en estado listo.

  La impresión se realiza en el orden cronológico definido por cuando el proceso cambió de estado, respetando la siguiente sintaxis:

  ```
  PID=%A Archivo=%B:%C Tiene=%D
  ```

  - %A se sustituye por el identificador del proceso.
  - %B se sustituye por el nombre del archivo.
  - %C se sustituye por el número de línea a ejecutar.
  - %D se sustituye por la lista de nombres de recursos (en minúscula) separados por coma. Se respeta el orden en el cual fueron pedidos.

  Por ejemplo:

  ```
  PID=2 Archivo=msn.exe:2 Tiene=d,f
  ```

  Si no tiene ningún recurso, no muestra ninguna lista, por ejemplo:

  ```
  PID=2 Archivo=msn.exe:2 Tiene=
  ```

- *ejecutarInstruccion_SO*: Ejecuta la próxima instrucción del proceso asociado al procesador dado. Si el procesador no cuenta con ningún proceso, tratará de buscar uno *listo* (respetando el orden FIFO) y le cambiará su estado a *corriendo*. Esta operación debe ejecutar en $O(1)$ peor caso.

  Debe imprimir en pantalla un resumen de lo realizado, respetando la siguiente sintaxis:

  ```
  CPU=%A PID=%B I=%C
  ```

  - %A se sustituye por el número del procesador.
  - %B se sustituye por el identificador del proceso.
  - %C se sustituye por la instrucción ejecutada.

  Por ejemplo:

  ```
  CPU=0 PID=2 I=D
  ```

- *terminarProceso_SO*: Termina el proceso indicado, liberando todos los recursos asignados a él y asignándolos a quienes correspondan, además de liberar toda la memoria asociada. Esta operación debe ejecutar en $O(1)$ caso promedio.
- *liberar_SO*: Libera toda la memoria de las estructuras intermedias.

3.3. Consideraciones

- Un proceso puede pedir tantos recursos como necesite.
- Un recurso puede ser pedido tantas veces como procesos hayan más uno.
- Un recurso es obtenido por un proceso a la vez.
- Un mismo programa puede ser ejecutado tantas veces como se desee.
- Siempre se debe respetar el orden (FIFO) de los procesos en estado *listo* cuando se asigna un proceso a un procesador.
- Siempre se debe respetar el orden (FIFO) de los procesos cuando se haya liberado un recurso por un proceso y éste tenga que elegir uno de los procesos bloqueados por ese recurso.
- La última instrucción de cualquier programa es '@'.
- Cuando termina un proceso puede pasar que no haya liberado todos los recursos, en este caso el SO debe liberarlos.
- El SO debe controlar que los procesos liberen solamente los recursos pedidos. En el caso que un proceso no respete esto, no se imprime nada en pantalla, ni se modifica el estado del recurso. Hay un ejemplo sobre esto en la Sección 3.5.3.

- En el caso que el procesador no tenga nada para ejecutar no imprime nada. También existe un ejemplo sobre esto en la Sección 3.5.3.

3.4. Archivo de cabecera

```
1  /*
2   * Tarea 2
3   *
4   * Programación 3
5   * Instituto de Computación
6   * Facultad de Ingeniería
7   * Universidad de la República
8   *
9   * $Rev: 189 $
10  *
11  * Nombres de recursos posibles:
12  * A B C D E F G H I J
13  * K L M N O P Q R S T
14  * U V W X Y Z
15  */
16 #ifndef SO_H
17 #define SO_H
18
19 #define SO_ERROR 0
20 #define SO_OK          1
21
22 struct SO;
23
24 /*
25  * Inicializa las estructuras internas con 'maxcpus' CPUs.
26  * En caso de error devuelve NULL
27  */
28 struct SO* iniciar_SO(unsigned int maxcpus);
29
30 /*
31  * Agrega al SO un programa con nombre 'nombre' y su
32  * código asociado 'codigo'.
33  * Pre: - No existe un programa con nombre 'nombre'
34  *      - Todo código termina con '@'
35  * Retorna SO_OK en caso de éxito, SO_ERROR en caso contrario.
36  */
37 int crearPrograma_SO(struct SO *, const char* nombre, const char *codigo);
38
39 /*
40  * Imprime los nombres de los programas
41  * que hay en el SO.
42  */
43 void imprimirProgramas_SO(struct SO* so);
44
45 /*
46  * Crea un proceso ejecutando el programa con nombre 'nombre'.
47  * Retorna el identificador del proceso, si no existe el programa
48  * devuelve 0.
49  */
50 unsigned int ejecutar_SO(struct SO *, const char* nombre);
51
52 /*
53  * Imprime la información relacionada a los
54  * procesos en estado bloqueado.
55  */
56 void imprimirProcesosBloqueados_SO(struct SO*);
57
58 /*
59  * Imprime la información relacionada a los
60  * procesos en estado listo.
61  */
62 void imprimirProcesosListos_SO(struct SO*);
63
64 /*
65  * Ejecuta la próxima instrucción del proceso en el
```

```
66   * procesador 'cpu'
67   * Pre: cpu < maxcpus
68   */
69  void ejecutarInstruccion_SO(struct SO*, unsigned int cpu);
70
71  /*
72   * Termina el proceso con identificador 'pid'
73   * Devuelve SO_OK en caso de éxito, SO_ERROR en caso contrario.
74   */
75  unsigned int terminarProceso_SO(struct SO*, unsigned int pid);
76
77  /*
78   * Libera toda la memoria asociada a la estructura SO.
79   */
80  void liberar_SO(struct SO *);
81
82  #endif
```

3.5. Ejemplos

3.5.1. Ejemplo de ejecución 1

Veamos el siguiente código:

```
1   /*
2    * Tarea 2: Ejemplo 1
3    *
4    * $Rev: 216 $
5    *
6    **/
7   #include <stdlib.h>
8   #include <stdio.h>
9
10  #include "so.h"
11
12  int main() {
13          struct SO *so = iniciar_SO(1);
14          unsigned int pidMSN, pidICQ;
15
16          crearPrograma_SO(so,"msn.exe","AB!_@");
17          crearPrograma_SO(so,"icq.exe","!___@");
18
19          // El proceso con identificador pidMSN pasa a
20          // estado LISTO
21          pidMSN = ejecutar_SO(so,"msn.exe");
22          // El proceso con identificador pidICQ pasa a
23          // estado LISTO
24          pidICQ = ejecutar_SO(so,"icq.exe");
25
26          /*
27           * El SO asigna el CPU-0 al proceso pidMSN,
28           * lo pasa a estado CORRIENDO
29           * Ejecuta la instrucción 'A'
30           * Asigna el recurso 'A' al proceso pidMSN
31           * */
32          ejecutarInstruccion_SO(so,0);
33
34          /*
35           * Ejecuta la instrucción 'B'
36           * Asigna el recurso 'b' al proceso pidMSN
37           * El proceso pidMSN tiene los recursos: a y b
38           * */
39          ejecutarInstruccion_SO(so,0);
40
41          /*
42           * Ejecuta la instrucción '!'
43           * El SO libera el CPU-0, y cambia el
44           * estado del proceso pidMSN a LISTO
45           * */
46          ejecutarInstruccion_SO(so,0);
47
48          /*
```

```
49           * El SO asigna el CPU-0 al proceso pidICQ,
50           * lo pasa a estado CORRIENDO.
51           * Ejecuta la instrucción '!'
52           * El SO libera el CPU-0, y cambia el estado
53           * del proceso pidICQ a LISTO
54           * */
55          ejecutarInstruccion_SO(so,0);
56
57          /*
58           * Ejecuta la instrucción '_'
59           * */
60          ejecutarInstruccion_SO(so,0);
61
62          /*
63           * Ejecuta la instrucción '@'
64           * El SO libera el CPU-0 y terminar el proceso
65           * liberando los recursos a y b
66           * */
67          ejecutarInstruccion_SO(so,0);
68
69          /*
70           * Ejecuta la instrucción '_'
71           * */
72          ejecutarInstruccion_SO(so,0);
73
74          /*
75           * Imprime los programas
76           * */
77          imprimirProgramas_SO(so);
78
79          liberar_SO(so);
80          return EXIT_SUCCESS;
81  }
```

Su salida correspondiente es:

```
CPU=0 PID=1 I=A
CPU=0 PID=1 I=B
CPU=0 PID=1 I=!
CPU=0 PID=2 I=!
CPU=0 PID=1 I=_
CPU=0 PID=1 I=@
CPU=0 PID=2 I=_
Programas:
icq.exe
msn.exe
```

3.5.2. Ejemplo de ejecución 2

Veamos el siguiente código:

```
1   /*
2    * Tarea 2: Ejemplo 2
3    *
4    * $Rev: 216 $
5    *
6    **/
7   #include <stdlib.h>
8   #include <stdio.h>
9
10  #include "so.h"
11
12  int main() {
13          // 2 CPUs identificadas como 0 y 1.
14          struct SO *so = iniciar_SO(2);
15
16          crearPrograma_SO(so,"notepad.exe","ABCD__abc@");
17          crearPrograma_SO(so,"msn.exe","D!____@");
18
19          // Devuelve 1
20          ejecutar_SO(so,"notepad.exe");
21          // Devuelve 2
```

```
22          ejecutar_SO(so,"msn.exe");
23
24          // Asigna a la CPU 0 el proceso con PID 1 y ejecuta 'A'
25          ejecutarInstruccion_SO(so,0);
26
27          // Ejecuta 'B' del proceso con PID 1
28          ejecutarInstruccion_SO(so,0);
29
30          // Ejecuta 'C' del proceso con PID 1
31          ejecutarInstruccion_SO(so,0);
32
33          // Asigna a la CPU 1 el proceso con PID 2 y ejecuta 'D'
34          ejecutarInstruccion_SO(so,1);
35
36          // Ejecuta 'D' del proceso con PID 1,
37          // lo pasa a bloqueado.
38          // Libera la CPU 0
39          ejecutarInstruccion_SO(so,0);
40
41          // Imprime proceso con PID 1
42          imprimirProcesosBloqueados_SO(so);
43
44          // Ejecuta '!' del proceso con PID 2,
45          // lo pasa a listo.
46          ejecutarInstruccion_SO(so,1);
47
48          // Imprime proceso con PID 2.
49          imprimirProcesosListos_SO(so);
50
51          // Imprime los procesos con PID 1
52          imprimirProcesosBloqueados_SO(so);
53
54          // Termina el proceso con PID 1, y libera
55          // toda la memoria asociada
56          terminarProceso_SO(so,1);
57
58          // No imprime nada
59          imprimirProcesosBloqueados_SO(so);
60
61          // Asigna a la CPU 0 el proceso con PID 2 y ejecuta '_'
62          ejecutarInstruccion_SO(so,0);
63
64          // Imprimo los programas
65          imprimirProgramas_SO(so);
66
67          // Libera toda la memoria asociada
68          liberar_SO(so);
69          return EXIT_SUCCESS;
70  }
```

Su salida correspondiente es:

```
CPU=0 PID=1 I=A
CPU=0 PID=1 I=B
CPU=0 PID=1 I=C
CPU=1 PID=2 I=D
CPU=0 PID=1 I=D
PID=1 Archivo=notepad.exe:4 Esperando=d Tiene=a,b,c
CPU=1 PID=2 I=!
PID=2 Archivo=msn.exe:2 Esperando= Tiene=d
PID=1 Archivo=notepad.exe:4 Esperando=d Tiene=a,b,c
CPU=0 PID=2 I=_
Programas:
msn.exe
notepad.exe
```

3.5.3. Ejemplo de ejecución 3

Veamos el siguiente código:

```
1   /*
2    * Tarea 2: Ejemplo 3
```

```
3    *
4    * $Rev: 218 $
5    *
6    **/
7
8   #include <stdlib.h>
9   #include "so.h"
10
11  int main() {
12          struct SO *so = iniciar_SO(1);
13
14          crearPrograma_SO(so,"emacs","A!@");
15          crearPrograma_SO(so,"vim","a@");
16
17
18          ejecutar_SO(so,"emacs");
19          ejecutar_SO(so,"vim");
20
21          /*
22           * Ejecuta instruccion 'A' del proceso con PID 1
23           */
24          ejecutarInstruccion_SO(so,0);
25
26          /*
27           * Ejecuta instruccion '!' del proceso con PID 1
28           */
29          ejecutarInstruccion_SO(so,0);
30
31          /*
32           * Ejecuta instruccion 'a' del proceso con PID 2
33           */
34          ejecutarInstruccion_SO(so,0);
35
36          /*
37           * Ejecuta instruccion '@' del proceso con PID 2
38           */
39          ejecutarInstruccion_SO(so,0);
40
41          /*
42           * Ejecuta instruccion '@' del proceso con PID 1
43           */
44          ejecutarInstruccion_SO(so,0);
45
46          /*
47           * No existe ningún proceso listo.
48           * No hace nada. No imprime nada.
49           */
50          ejecutarInstruccion_SO(so,0);
51
52          liberar_SO(so);
53          return EXIT_SUCCESS;
54  }
55
```

Su salida correspondiente es:

```
CPU=0 PID=1 I=A
CPU=0 PID=1 I=!
CPU=0 PID=2 I=@
CPU=0 PID=1 I=@
```

4. Lenguaje a utilizar

El lenguaje a utilizar en este trabajo será C con las siguientes extensiones:

- Operadores new y delete.
- Pasaje de parámetros por referencia (uso de &).
- Declaración de tipos como en C++ para registros y enumerados.
- Sobrecarga de funciones.
- Uso de cin y cout.
- Uso del tipo bool predefinido en C++.

5. Qué se espera

Para cada módulo de cabecera (.h) con los prototipos de las operaciones solicitadas, debe entregarse un módulo (.cpp) con la implementación de dichas operaciones. Debe respetarse estrictamente los prototipos especificados, esto es: nombre de la operación, tipo, orden y forma de pasaje de los parámetros y tipo de retorno.

Los **módulos de cabecera pueden** bajarse de la página web del curso. Estos módulos no forman parte de la entrega, y por lo tanto, **no deben ser modificados**. **Los módulos deben funcionar en el ambiente MinGW instalado en facultad**. Se espera que todos los módulos compilen **sin errores** utilizando las flags "-Wall" , "-Werror" y "-O1", se ejecuten **sin colgarse** y den los **resultados correctos**.

6. Forma de la entrega

Se deberá entregar únicamente 1 archivo (respetando las mayúsculas en los nombres):

- so.cpp

correspondiente a la implementación del sistema especificado. No se podrá entregar otra cosa que no sea este archivo.

La primera línea del archivo debe contener un comentario (/* ... */) con la cédula del autor, sin puntos ni dígito de verificación. Por ejemplo, si la cédula es 1.234.567-8, la primera línea de cada archivo deberá ser exactamente:

```
/* 1234567 */
```

7. Advertencia sobre el manejo de la memoria

Cuando un programa contiene errores en el manejo de la memoria, su comportamiento puede ser inestable. Esto implica que algunas veces funciona correctamente y otras no. En ciertos casos esto puede inducir a creer (erróneamente) que ciertos programas, que en realidad son incorrectos, funcionan correctamente. Este aspecto es influenciado, entre otras cosas, por el sistema operativo en el que se ejecuten los programas. Recomendamos tener sumo cuidado con este punto y testear los módulos el sistemas operativos Windows XP. CON LA VERSIÓN DE MINGW INSTALADA EN LAS SALAS DE INFORMÁTICA DE LA FACULTAD.

8. Sobre la individualidad del trabajo

El laboratorio es INDIVIDUAL. Los estudiantes pueden estudiar en grupo y consultarse mutuamente, pero NO pueden realizar en grupo las tareas de codificación, escritura, compilación y depuración del programa.

Los trabajos de laboratorio que a juicio de los docentes no sean individuales serán eliminados, con la consiguiente pérdida del curso, para todos los involucrados. Además todos los casos serán enviados a los órganos competentes de la Facultad, lo cual puede acarrear sanciones de otro carácter y gravedad para los estudiantes involucrados.

No existirán instancias en el curso para reclamos frente a la detección por parte de los docentes de trabajos de laboratorio no individuales, independientemente de las causas que pudiesen originar la no individualidad. A modo de ejemplo y sin ser exhaustivos: utilizar código realizado en cursos anteriores u otros cursos, perder el código, olvidarse del código en lugares accesibles a otros estudiantes, prestar el código o dejar que el mismo sea copiado por otros estudiantes, dejar la terminal con el usuario abierto al retirarse, enviarse código por mail, etc. Asimismo, se prohíbe el envío de código al grupo de noticias del curso, dado que el mismo será considerado como una forma de compartir código y será sancionada de la manera más severa posible.

Es decir que se considera a cada estudiante RESPONSABLE DE SU TRABAJO DE LABORATORIO Y DE QUE EL MISMO SEA INDIVIDUAL. NO CONFIAR en el borrado automático del directorio pub de las salas Windows. Es decir antes de cerrar sesión borrar todos los archivos del directorio.

9. Fecha de entrega

El trabajo debe entregarse el día **lunes 13 de setiembre de 2010 antes de las 22:00 horas**. La entrega se realizará mediante un formulario que se habilitará oportunamente en la página web del curso.