Organización de Computadoras

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Segundo Cuatrimestre de 2014

Proyecto Nº 1
Programación en Lenguaje C

El objetivo del proyecto es implementar un simulador de puestos de atención en un establecimiento.

En este establecimiento hay n puestos de atención (identificados con un número entre 1 y n) que atienden diferentes trámites. Cada trámite tiene asociado un tiempo de duración y hay un orden de importancia entre ellos. Cada puesto de atención tiene asociada una cola de espera que se actualiza de la siguiente forma:

- Cuando un cliente llega al establecimiento se ubica en el primer puesto libre que encuentra. De no encontrar un puesto libre se ubica al final de la cola del puesto que menos gente tenga esperando. Un puesto está libre si no está atendiendo a nadie y además no tiene clientes esperando. Frente a diferentes puestos en las mismas condiciones, los clientes prefieren los puestos de menor número de identificación.
- Cuando un puesto se libera atiende a la persona que está primero en su cola de espera. Si su cola de espera estuviera vacía, atiende al último de la cola de espera adyacente que más gente tenga.
- Si dos clientes llegan en el mismo momento, se ubica primero el que tenga el trámite de mayor importancia.

En la simulación hay diferentes tipos de eventos que ocurren secuencialmente en el siguiente orden:

1. *Liberación de puestos:* Si un puesto está atendiendo a un cliente, continúa atendiéndolo. Si se terminó la operación entonces el puesto se libera.
2. *Comienzo de atención:* Si un puesto no está atendiendo a nadie y hay clientes esperando en su cola, entonces atiende al primero de la cola.
3. *Arribo de clientes:* Arriban los clientes y se ubican en las colas de espera o en puestos vacíos.
4. *Robo de clientes:* Si un puesto sigue sin atender a nadie, busca atender al último cliente de una cola vecina.

Ejemplo

Supongamos que el establecimiento tiene 3 puestos de atención y los trámites disponibles (ordenados de mayor a menor importancia) son los siguientes:

Trámite	Duración
pago	5
impresión	4
reclamo	15
contratación	7

Si los puestos abren en el tiempo 0 y los clientes llegan en el siguiente orden:

Tiempo	Trámite
1	reclamo
2	impresión
2	contratación
3	contratación
5	impresión
6	pago

Las colas de espera se actualizarán de la siguiente forma:

Tiempo	Puesto	Atiende	Cola de espera
0	1:		[]
	2:		[]
	3:		[]
1	Llega un reclamo a puesto 1		
	1:	reclamo	[]
	2:		[]
	3:		[]
2	Llega una impresión al puesto 2		
	Llega una contratación al puesto 3		
	1:	reclamo	[]
	2:	impresión	[]
	3:	contratación	[]
3	Llega una contratación a puesto 1		
	1:	reclamo	[contratación]
	2:	impresión	[]
	3:	contratación	[]
5	Llega una impresión al puesto 2		
	1:	reclamo	[contratación]
	2:	impresión	[impresión]
	3:	contratación	[]
6	Se libera el puesto 2		
	El puesto 2 atiende al siguiente de la cola		
	Llega un pago al puesto 2		
	1:	reclamo	[contratación]
	2:	impresión	[pago]
	3:	contratación	[]
9	Se libera el puesto 3		
	El puesto 3 roba un cliente al puesto 2		
	1:	reclamo	[contratación]
	2:	impresión	[]
	3:	pago	[]
10	Se libera el puesto 2		
	El puesto 2 roba un cliente de puesto 1		
	1:	reclamo	[]
	2:	contratación	[]
	3:	pago	[]
14	Se libera el puesto 3		
16	Se libera el puesto 1		
17	Se libera el puesto 2		

Entradas a la simulación

Los datos de entrada a la simulación se reciben en diferentes archivos:

- Un archivo que se ingresa como parámetro a la aplicación donde se especifica qué tramites hay disponibles y cuánto tiempo demora la atención de cada trámite. Además, el orden en que se listen estos trámites establecerá el orden de importancia entre ellos: el primero en la lista será el trámite de mayor importancia, mientas que el último de la lista será el de menor importancia.

  En el archivo se especifica un trámite por línea de la forma:

  `<nombre trámite><tab><número entero>`↲

  En el ejemplo anterior este archivo contendrá las líneas:

  ```
  pago          5↲
  impresión     4↲
  reclamo       15↲
  contratación  7↲
  ```

- Por cada trámite hay un archivo de nombre "`<nombre-trámite>.txt`" que lista los tiempos en los que llegan clientes con ese trámite particular.

  En el ejemplo anterior, habrá 4 archivos que contendrán los siguientes números:

pago.txt	reclamo.txt	impresion.txt	contratacion.txt
6↲	1↲	2↲	2↲
		5↲	3↲

Salida de la simulación

La salida de la simulación mostrará para todos aquellos tiempos en los que haya algún cambio en los puestos o en las colas de espera, el estado de todos los puestos de atención, de todas las colas de espera y los eventos ocurridos.

La salida debe tener el siguiente formato (véase la tabla de actualización de colas del ejemplo):

```
TIEMPO t
Listado y descripción de los eventos:
- Liberación de puestos
- Comienzo de atención de clientes de la cola de espera propia
- Arribo de un trámite nuevo a un puesto en particular
- Comienzo de atención de clientes robados a una cola de espera ajena
 1:   trámite que está atendiendo el puesto 1   [trámites en la cola de espera del puesto 1]
 2:   trámite que está atendiendo el puesto 2   [trámites en la cola de espera del puesto 2]
...
 n:   trámite que está atendiendo el puesto n   [trámites en la cola de espera del puesto n]
```

Sintaxis

El programa implementado (simulador) debe conformar la siguiente especificación al ser invocado desde la línea de comandos:

```
$ simulador  [-h |  [-f <número entero>]
                    -i <archivo entrada> -n <número entero>
                    [-o <archivo salida>]]
```

Los parámetros entre corchetes denotan parámetros opcionales. El significado de las diversas opciones es el siguiente:

- En caso de no especificar parámetros, de que haya algún error en los parámetro o de especificar el parámetro `-h` como primer argumento en la línea de comandos, el programa debe mostrar una pequeña ayuda por pantalla, la cual consiste en un breve resumen del propósito del programa junto con una reseña de las diversas opciones disponibles.
- Si se especifica el parámetro `-f <número entero>` debe realizarse la simulación hasta cumplir el tiempo especificado por el parámetro. De no estar presente este parámetro debe realizarse la simulación hasta que se haya atendido a todos los clientes.
- El parámetro `-i <archivo entrada>` es obligatorio y especifica los trámites disponibles para atender.
- El parámetro `-n <número entero>` es obligatorio e indica la cantidad de puestos de atención disponibles.
- El parámetro `-o <archivo salida>` indica el nombre del archivo en el que se escribirá la salida de la simulación. De no especificarse la salida debe mostrarse por consola.

Implementación

Para implementar el proyecto se debe utilizar las siguientes estructuras dinámicas

- una lista ordenada simplemente enlazada
- un *Deque* implementado con una lista doblemente enlazada.

Ante eventuales errores en los datos de entrada el programa debe terminar correctamente mostrando un mensaje de error significativo.

Además, *debe hacerse un uso eficiente de la memoria, liberando el espacio reservado para todos los datos que ya no sean necesarios.*

1. TDA Lista Ordenada

Implementar en C utilizando punteros un TDA Lista Ordenada cuyos elementos sean punteros genéricos. El orden de los elementos en la lista se especifica al momento de la creación a través de una función de comparación. La lista debe ser simplemente enlazada sin centinelas. La implementación debe contener las operaciones:

- `tListaOrdenada crearLista(int (*f)(void*,void*))` Crea una lista vacía y la devuelve. El orden de los elementos insertados estará dado por la función de comparación `int f(void*,void*)`. Se considera que la función `f` devuelve -1 si el orden del primer argumento es menor que el orden del segundo, 0 si el orden es el mismo, y 1 si el orden del primer argumento es mayor que el orden del segundo.
- `void insertar(tListaOrdenada* L, void* x)` Inserta el elemento `x` ordenado en la lista `L`.
- `tNodo* siguiente(tNodo* pos)` Devuelve la posición siguiente a la posición `pos`. Devuelve `NULL` si `p` es `NULL` o si no existe posición siguiente.
- `tNodo* primera(tListaOrdenada L)` Devuelve la primera posición de la lista `L`.
- `tNodo* ultima(tListaOrdenada L)` Devuelve la última posición de la lista `L`.
- `void* elemento(tNodo* pos)` Devuelve un puntero al elemento que ocupa la posición `pos` de la lista.

- `int listaVacia(tListaOrdenada L)` Devuelve verdadero ($\neq 0$) si la lista está vacía, y falso (=0) si la lista contiene al menos un elemento.

donde los tipos `tNodo` y `tListaOrdenada` están definidos de la siguiente manera:

- en el header:

```
struct Nodo;
typedef struct Nodo tNodo;
struct Lista {
    tNodo* ppio; //puntero al primer nodo de la lista
    tNodo* fin; //puntero al último nodo de la lista
    int cant; // cantidad de elementos de la lista
    int (*comparador)(void*,void*);
};
typedef struct Lista tListaOrdenada;
```

- en la implementación:

```
struct Nodo {
    void * elem;
    struct Nodo* next;
};
```

2. TDA Deque

Una cola doblemente terminada (*double–ended queue* o *deque*) es un dato abstracto más general que las pilas o las colas convencionales: los elementos pueden insertarse o eliminarse de ambos extremos. Se pueden distinguir dos tipos de *deques* restringidas:

- *Deque* de entrada restringida es aquella en la que las eliminaciones se pueden realizar en los dos extremos, pero las inserciones se realizan en un solo extremo.
- *Deque* de salida restringida es aquella en la que las inserciones se pueden realizar en ambos extremos, pero las eliminaciones se realizan en un solo extremo.

Implementar en C el TDA Deque de entrada restringida como una lista doblemente enlazada cuyos elementos sean punteros genéricos.

- `tDeque* crearDeque()` Crea una *Deque* vacía y la devuelve.
- `void insertarEnDeque(tDeque* D, void * x)` Inserta el elemento `x` al final de la cola.
- `void* eliminarAtras(tDeque* D)` Elimina y devuelve el primer elemento de la Deque.
- `void* eliminarAdelante(tDeque* D)` Elimina y devuelve el último elemento de la Deque.
- `int dequeVacio(tDeque* D)` Devuelve verdadero ($\neq 0$) si la Deque está vacía, y falso (0) si contiene al menos un elemento.
- `int ocupacionDeque(tDeque * D)` Devuelve la cantidad de elementos que tiene la Deque.
- `void * deque2array(tDeque * D)` Devuelve un arreglo con todos los elementos de la Deque.

donde los tipos `tNodo` y `tDeque` están definidos de la siguiente manera:

- En el *header*:

```
typedef struct Deque tDeque;
```

- En la implementación

```
typedef struct Nodo {
    void* elem;
    struct Nodo* sig;
    struct Nodo* ant;
} tNodo;

struct Deque {
    tNodo* ppio; //puntero al primer nodo de la cola
    tNodo* fin; //puntero al último nodo de la cola
    int cant;
};
```

3. TDA Trámite

Representa un trámite que llega a las colas de espera. Mantiene el nombre del cliente, la duración del trámite que quiere realizar y el momento de llegada. La estructura de datos debe especificarse en el *header* y es la siguiente:

```
typedef struct Tramite {
    char * nombre;
    int duracion;
    int llegada;
} tTramite;
```

A pesar de ser visible la estructura, debe agregar todas las operaciones necesarias para modificar y consultar un trámite sin necesidad de acceder directamente a la estructura.

Sobre la implementación

- Los archivos fuente principales se deben denominar **simulador.c, deque.c, listaordenada.c y tramite.c** respectivamente. Recordemos que al tratarse de librerías, también se deben adjuntar los respectivos archivos de encabezados **deque.h, listaordenada.h y tramite.h**, los cuales han de ser incluidos en los archivos fuente de los programas que hagan uso de estas librerías.
- Es importante que durante la implementación del proyecto se haga un uso cuidadoso del espacio de memoria, tanto para la reservar (`malloc`) la memoria necesaria, como para liberar (`free`) la memoria de variables que ya no sean necesarias.
- Se debe respetar los nombres de tipos y los encabezados de funciones especificados en el enunciado. *Los proyectos que no cumplan esta condición quedarán automáticamente desaprobados*
- *Los proyectos que incluyan código que no compile quedarán automáticamente desaprobados.*

Sobre el estilo de programación

- El código implementado debe reflejar la aplicación de las técnicas de programación modular estudiadas a lo largo de la carrera.
- En el código, entre eficiencia y claridad, se debe optar por la claridad. Toda decisión en este sentido debe constar en la documentación que acompaña al programa implementado.
- El código debe estar *indentado* y adecuadamente comentado.

Sobre la documentación

Los proyectos que no incluyan documentación estarán automáticamente desaprobados.
La documentación debe:

- estar dirigida principalmente a usuarios con conocimientos de programación.
- explicar brevemente los programas realizados, así como las decisiones de diseño tomadas, y toda otra observación que se considere pertinente.
- incluir explicación de todas las funciones implementadas, indicando su prototipo y el uso de los parámetros de entrada y de salida.

Sobre la entrega

- Las comisiones deben estar conformadas por exactamente 3 alumnos registrados con la cátedra. La fecha límite para informar a la cátedra quiénes son los integrantes de cada comisión es el **Jueves 9 de octubre de 2014**.

  No se aceptarán cambios a último momento en los integrantes de las mismas.
- El código fuente del proyecto deberá ser enviado en un archivo *zip* por mail a la asistente de la cátedra hasta las **24:00hs** del día **Lunes 27 de octubre de 2014**. Tanto el asunto del mail como el nombre del archivo comprimido debe ser el siguiente:

  [OC2014] Proyecto 1 - apellido de los integrantes de la comisión

  Toda comisión que no cumpla este punto estará automáticamente desaprobada.
- El día **Martes 28 de octubre de 2014**, de **10 a 12hs**, en el aula 3 de Palihue deberá entregarse un folio plástico (no se aceptarán carpetas), cerrado con cinta adhesiva, conteniendo los siguientes elementos:
  - Una carátula que identifique claramente a los integrantes de la comisión.
  - Una impresión en doble faz de los archivos ".c" y ".h" enviados por mail.
  - La documentación del proyecto impresa en doble faz.

  No se aceptarán discrepancias entre el código fuente impreso y el enviado por mail.