UNIVERSIDAD DE MÁLAGA

Programación de Sistemas y Concurrencia

Dpto. de Lenguajes y Ciencias de la Computación

Examen Septiembre 2018

APELLIDOS_____ NOMBRE_____

DNI _____ ORDENADOR _____ GRUPO/TITULACIÓN_____

Bloque Programación C (3,5 puntos)

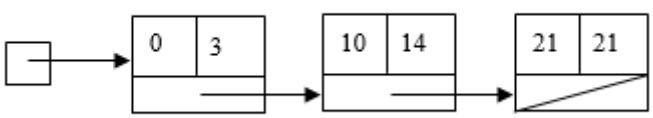
Se pide realizar, utilizando memoria dinámica, una simulación del comportamiento del uso que diferentes procesos hacen de una memoria caché para mejorar su ejecución. Dicha memoria caché se compone de clusters,.

Cada vez que un proceso demanda memoria indica la cantidad de clusters que necesita. Para que sea posible atender su solicitud es necesario que existan clusters contiguos disponibles. Evidentemente, no pueden existir clusters que sean utilizados por más de un proceso. Para la realización de la simulación se supone que la memoria caché tiene un número MAX de clusters y que si un proceso solicita varios clusters, deben estar todos contiguos.

Cuando un proceso no necesita utilizar la memoria se realizará una liberación de todos los clusters que hayan sido utilizados. Se ha pensado utilizar las siguientes estructuras de datos para gestionar la memoria caché:

```
typedef struct T_Nodo* T_Cluster;
struct T_Nodo {
      unsigned dir_inicio;
      unsigned dir_fin;
      T_Cluster sig;
};
```

en la que cada nodo de la estructura dinámica nos indica la zona disponible de clusters (desde el primer cluster `dir_inicio` hasta el último cluster `dir_fin`, ambas incluidas) disponible en la memoria cache. Inicialmente existirá un único nodo con `dir_inicio = 0` y `dir_fin = MAX-1`. Esta lista de nodos estará ordenada por el valor `dir_inicio`. En un momento determinado esta estructura puede tener el aspecto de la siguiente figura:



En esta situación los clusters disponibles son el 0, 1, 2, 3, 10, 11, 12, 13, 14 y 21.

Nota: Para simplificar el ejercicio se permite que la estructura no mantenga los nodos de forma compacta, es decir, la lista puede contener dos nodos consecutivos de manera que la dir_fin+1 del primer nodo coincida con dir_inicio del segundo nodo.

Se desean implementar las siguientes operaciones:

//Crea la estructura utilizada para gestionar los clusters libres. Se crea un solo nodo desde 0 a MAX-1 **(Valoración: 0,5 puntos sobre 10)**

void crearLista(T_Cluster *cache);

//Comprueba si existe espacio en memoria caché disponible para almacenar un proceso que necesita memoria de tamaño *clusters*. Si hay espacio devuelve el número del primer cluster donde se almacenaría, en caso contrario devuelve un valor -1 Es importante recordar que para almacenarse en memoria el número de clusters deben ser contiguos**. Hay que tener en cuenta que la estructura puede no ser compacta**. **(Valoración: 2 puntos sobre 10)**

int existeEspacio(T_Cluster cache, unsigned clusters)

//Modifica la estructura de memoria disponible si el número de clusters solicitados están disponibles. La variable ok valdrá 1 si se pueden reservar todos y 0 en caso contrario. La variable dir almacena la dirección del primer cluster asignado al proceso. Hay que tener en cuenta que la estructura puede no ser compacta. **(Valoración: 2,5 puntos sobre 10)**

void reservaMemoriaProceso(T_Cluster *cache, unsigned clusters, unsigned *ok, unsigned *dir);

//Se modifica la estructura de memoria disponible, teniendo en cuenta que se liberan clusters contiguos a partir del cluster número "dir" (incluida). No hace falta que la estructura que quede sea compacta. Es decir, al liberar estos clusters, puede haber dos nodos seguidos en el que el valor dir_fin de un determinado nodo sea una unidad inferior al valor dir_inicio del siguiente nodo. **(Valoración: 2,5 puntos sobre 10)**

void liberar(T_Cluster *cache, unsigned dir, unsigned clusters);

//Muestra en pantalla una lista indicando los códigos de cluster que están libres **(Valoración: 0,5 puntos sobre 10)**

void imprimir_libres(T_Cluster cache);

//Destruye la estructura completa, liberando todos los nodos de la misma de memoria. **(Valoración: 0,75 puntos sobre 10)**

void destruir(T_Cluster *cache);

//Crea el fichero binario CLUSTERS.BIN donde se almacenan los valores numéricos de los clusters ocupados en memoria. **(Valoración: 1,25 puntos)**
void volcadoUsados(T_Cluster cache);

Anexo. Los prototipos de las funciones de lectura y escritura binaria en ficheros de la biblioteca <stdio.h> son los siguientes (se dan por conocidos los prototipos de las funciones de <stdlib.h> que necesites, como free o malloc):

`FILE *fopen(const char *path, const char *mode);`
Abre el fichero especificado en el modo indicado ("rb" para lectura binaria y "wb" para escritura binaria). Devuelve un puntero al manejador del fichero en caso de éxito y NULL en caso de error.

`unsigned fwrite(const void *ptr, unsigned size, unsigned nmemb, FILE *stream);`
Escribe `nmemb` elementos de datos, cada uno de tamaño `size`, al fichero `stream`, obteniéndolos desde la dirección apuntada por `ptr`. Devuelve el número de elementos escritos.

`int fclose(FILE *fp);`
Guarda el contenido del buffer y cierra el fichero especificado. Devuelve 0 en caso de éxito y -1 en caso de error.

Bloque Concurrencia (6,5 puntos)

Ejercicio Semáforos binarios

En una empresa hay un aseo unisex que es compartido por hombres y mujeres. Cada trabajador (hombre o mujer) puede estar en el aseo, puede estar esperando para entrar en el aseo o puede estar trabajando. La regla para usar el aseo es muy simple: no puede haber nunca un hombre y una mujer utilizando el aseo simultáneamente; sin embargo, cualquier número de personas del mismo sexo pueden estar utilizando el aseo a la vez.

Implementa este ejercicio utilizando semáforos binarios para sincronizar a los procesos, sin preocuparse por la justicia de la solución. Utiliza el esqueleto que se encuentra en el campus virtual. La clase Aseo tiene los siguientes métodos:

```
/**
* El hombre id quiere entrar en el aseo.
* Espera si no es posible, es decir, si hay alguna mujer en ese
* momento en el aseo
*/
public void llegaHombre(int id)

/**
* La mujer id quiere entrar en el aseo.
* Espera si no es posible, es decir, si hay algun hombre en ese
* momento en el aseo
*/
public void llegaMujer(int id)

/**
* El hombre id, que estaba en el aseo, sale
*/
public void saleHombre(int id)
```

```
/**
* La mujer id, que estaba en el aseo, sale
*/
public void saleMujer(int id)
```

Ejercicio Monitores (Métodos sincronizados/Locks)

La parada del autobús de la Escuela de Informática funciona de un modo un poco peculiar. Cuando un pasajero llega a la parada del autobús siempre se sube **al siguiente** autobús que llegue. Esto significa que si no hay ningún autobús cuando un viajero llega a la parada, entonces se subirá en el próximo que venga. Sin embargo, si, cuando el viajero llega a la parada, hay un autobús, el viajero no puede subirse en ese autobús, sino que tiene que esperar al siguiente. Dicho de otro modo, cuando un autobús llega a la parada se suben **todos** los viajeros que están esperando **en ese momento** (para simplificar la solución suponemos que el autobús tiene una capacidad ilimitada). Cualquier otra persona que llegue después de que haya llegado el autobús tiene que esperar al siguiente. Esto significa también que si el autobús llega y no hay nadie en la parada, el autobús sale vacío.

Implementa una solución a este problema utilizando métodos sincronizados o locks. En tu solución supón que en la parada del autobús hay dos grupos de viajeros, uno principal y otro secundario:

- Si el autobús no está en la parada, todos los viajeros están en grupo principal, y el grupo secundario está vacío
- Si el autobús está en la parada, los viajeros del grupo principal, que estaban antes de que llegara el autobús, se están subiendo al mismo; y el grupo secundario contiene a los viajeros que han llegado después del autobús y tienen que esperar al siguiente. Cuando el autobús se va, el grupo secundario se convierte en el principal y viceversa.

En el campus virtual se encuentra el esqueleto de la solución que tienes que implementar (en la solución se ha supuesto que los grupos se identifican con números enteros). Observa que sólo se implementa la subida de los pasajeros al autobús. Una vez que se han subido, se supone que se bajan de forma instantánea y que el autobús vuelve a estar vacío.

La clase Parada contiene los siguientes métodos:

```
/**
* El pasajero id llama a este metodo cuando llega a la parada.
* Siempre tiene que esperar el siguiente autobus en uno de los
* dos grupos de personas que hay en la parada
* El metodo devuelve el grupo en el que esta esperando el pasajero
*
*/

public int esperoBus(int id)
```

```java
/**
* Una vez que ha llegado el autobús, el pasajero id que estaba
* esperando en el grupo i se sube al autobus
*
*/
public void subeAutobus(int id,int i)

/**
* El autobus llama a este metodo cuando llega a la parada
* Espera a que se suban todos los viajeros que han llegado antes
* que el, y se va
*
*/
public void llegoParada()
```