Tarea 2

Matías García
Matías Valdebenito
Diego Rebollo
Oliver Peñailillo

Contenido

1 Objetivos **2**

2 Sincronización de hebras **4**
2.1 Monitor 4
2.2 Cola 6

3 Memoria virtual **8**
3.1 Tabla de páginas 8
3.2 Algoritmos de reemplazo 10
3.2.1 Optimo 11
3.2.2 FIFO 12
3.2.3 LRU 13
3.2.4 LRU Reloj Simple 14

1 Objetivos

El objetivo del presente trabajo es profundizar en conocimientos y habilidades relacionadas a la programación multihebra, sincronización y manejo de memoria virtual. Para ello, se desarrollarán dos simuladores independientes, los cuales abordarán problemáticas presentes en el ámbito de programación multihebra y mecanismos de sincronización, así como gestión y manejo de memoria virtual.

Primeramente, se implementará un simulador que maneje una cola circular de tamaño dinámico, compartida entre múltiples hebras tanto productoras como consumidoras, mediante la utilización de una estructura tipo monitor, la cual permita un correcto manejo de la concurrencia. Así mismo, la segunda implementación tendrá como objetivo la simulación de memoria virtual con paginación, siendo capaz de usar y aplicar diversos algoritmos de reemplazo de páginas. Concretamente, estos algoritmos serán: FIFO, LRU, LRU Reloj Simple y Óptimo.

Ambos simuladores serán implementados en C++.

Link del repositorio: Repositorio

Previo a abordar la implementación de cada simulador, es conveniente considerar el siguiente archivo de cabecera, el cual define algunas configuraciones y alias de tipos, con el propósito de facilitar la escritura de código y legibilidad:

```
#ifndef TYPE_H
#define TYPE_H

#include <ostream>
#include <string>

#define SELF (*this)
#define NULLPTR (nullptr)

namespace cpp = std;

using ostream = cpp::ostream;
using string = cpp::string;

#endif
```

2 Sincronización de hebras

2.1 Monitor

Su estructura es la siguiente:

```
class QueueTS : public Queue
{
    public:
        QueueTS(int size, int maxWait, Logger* logger);

        Item pop();
        void push(Item item);

    private:
        int maxWait;

        Logger* logger;

        cpp::mutex mutex;
        cpp::condition_variable hasItems;
};

#endif
```

A continuación se explicarán algunos de sus métodos principales:

La implementación del método *push* es la siguiente:

```
void QueueTS::push(Item item)
{
    ulock lock(SELF.mutex);

    int capacity1 = SELF.capacity();
    Queue::push(item);
    int capacity2 = SELF.capacity();

    if(SELF.logger ≠ NULLPTR)
    {
        cpp::stringstream message;

        message << "PUSHED: " << item;

        if(capacity1 ≠ capacity2)
        {
            message << " ";
            message << "[" << capacity1 << " → " << capacity2 << "]";
        }

        SELF.logger → print(message.str());
    }

    cpp::cerr << SELF << '\n';

    lock.unlock();

    SELF.hasItems.notify_one();
}
```

En términos generales, este método agrega un elemento a la cola de manera segura utilizando sincronización para evitar condiciones de carrera.

Primero, adquiere un bloqueo sobre el mutex para garantizar que ninguna otra hebra modifique la cola simultáneamente. Luego, obtiene y guarda la capacidad de la cola antes y después de insertar el ítem, para detectar si la capacidad ha cambiado. Si la cola se redimensiona, este cambio se incluye en un mensaje de registro junto con la información del ítem agregado, el cual se pasa al objeto logger para su almacenamiento o impresión. Después de insertar el ítem, el método imprime el estado actual de la cola en la salida estándar para fines de monitoreo. Finalmente, el bloqueo del mutex se libera automáticamente y se notifica a uno de los consumidores, si es necesario, indicando que hay elementos disponibles para procesar.

La implementación del método *pop* es la siguiente:

```
Item QueueTS::pop()
{
    ulock lock(SELF.mutex);

    while(SELF.size() == 0)
    {
        SELF.hasItems.wait_for(lock, SELF.maxWait * cpp::chrono::seconds(1));

        if(SELF.size() == 0)
            throw -1;
    }

    int capacity1 = SELF.capacity();
    Item item = Queue::pop();
    int capacity2 = SELF.capacity();

    if(SELF.logger ≠ NULLPTR)
    {
        cpp::stringstream message;

        message ≪ "POPPED: " ≪ item;

        if(capacity1 ≠ capacity2)
        {
            message ≪ " ";
            message ≪ "[" ≪ capacity1 ≪ " → " ≪ capacity2 ≪ "]";
        }

        SELF.logger → print(message.str());
    }

    cpp::cerr ≪ SELF ≪ '\n';

    lock.unlock();

    return item;
}
```

Este método permite extraer un elemento de la cola de manera segura, manejando la concurrencia con el uso del mutex. Primero, se adquiere el bloqueo sobre el mutex para evitar que otras hebras modifiquen la cola al mismo tiempo. Luego, se verifica si la cola está vacía. Si es así, la hebra se bloquea esperando la notificación de que hay elementos disponibles. Al extraer el elemento, se registra el estado antes y después de la operación para comprobar

si la capacidad de la cola ha cambiado. Si se produce un cambio en la capacidad, se incluye esta información en un mensaje de registro que es enviado al logger para su seguimiento. Además, se imprime el estado actual de la cola en la salida estándar. Finalmente, el mutex se libera automáticamente, y el elemento extraído es devuelto a la función que lo invoca.

2.2 Cola

Su estructura es la siguiente:

```
#ifndef QUEUE_H
#define QUEUE_H

#include "Item.hxx"
#include "Type.hxx"

class Queue
{
    public:
        Queue(int size);
        ~Queue();

        int size();
        int capacity();

        Item pop();
        void push(Item item);

        friend ostream& operator<<(ostream& stream, const Queue& queue);

    private:
        int first;
        int final;

        int limit;

        Item* items;

        void resize(int size);
};

#endif
```

La implementación del método *pop* es la siguiente:

```
Item Queue::pop()
{
    assert(SELF.size() ≥ 1);

    Item item = SELF.items[SELF.first];

    SELF.first += 1;
    SELF.first %= SELF.limit;

    if(SELF.size() == (SELF.limit / 4))
        SELF.resize(SELF.limit / 2);

    return item;
}
```

Extrae y devuelve el primer elemento de la cola. Si el tamaño de la cola llega a 1/4 de su capacidad, se redimensiona a la mitad de su tamaño actual llamando a *resize*().

La implementación del método *push* es la siguiente:

```
void Queue::push(Item item)
{
    if(SELF.size() == (SELF.limit - 1))
        SELF.resize(SELF.limit * 2);

    SELF.items[SELF.final] = item;

    SELF.final += 1;
    SELF.final %= SELF.limit;
}
```

Se agrega un nuevo elemento al final de la cola. Si la cola está llena, se redimensiona a un tamaño doble, mediante el uso del método *resize*().

La implementación de *resize* es la siguiente:

```
void Queue::resize(int size)
{
    assert(size ≥ SELF.size());

    Item* aux = SELF.items;

    SELF.items = (Item*) malloc(sizeof(Item) * size);

    for(int i = 0; i < SELF.size(); i++)
        SELF.items[i] = aux[(i + SELF.first) % SELF.limit];

    SELF.final = SELF.size();
    SELF.first = 0;

    SELF.limit = size;

    delete aux;
}
```

Este método ajusta el tamaño de la cola, ya sea aumentando o reduciendo su capacidad. Primero verifica que el nuevo tamaño sea adecuado para contener los elementos actuales. Luego, asigna un nuevo bloque de memoria y copia los elementos desde el arreglo antiguo al nuevo, respetando el orden de los elementos en la cola. Después, actualiza los índices $first$ y $final$, que indican las posiciones de los elementos en la cola, y finalmente, libera la memoria del arreglo antiguo para evitar fugas de memoria. Este método permite que la cola crezca o se reduzca dinámicamente según sea necesario.

3 Memoria virtual

Para abordar la implementación del simulador de memoria virtual, se explicarán sus componentes principales, es decir, la implementación de la tabla de páginas y algoritmos de reemplazo.

3.1 Tabla de páginas

Para computar la tabla de páginas, se creó la siguiente clase:

```
#ifndef PAGE_TABLE_H
#define PAGE_TABLE_H

#include <fstream>
#include <set>
#include <unordered_map>

#include "Algorithm.hxx"
#include "Type.hxx"

class PageTable
{
    public:
        PageTable(const int frameCount, Algorithm* algorithm);
        ~PageTable();

        int getMissCount();

        void accessPage(const int npv);

    private:
        int missCount;
        int frameCount;

        Algorithm* algorithm;

        cpp::set<int> freeFrames;
        cpp::unordered_map<int, int> table;

        friend ostream& operator<<(ostream& stream, const PageTable& pageTable);
};

#endif
```

Notar que algunso de sus elementos principales son:

- Método $getMissCount()$: Retorna el número de fallos de página
- Método $accessPage()$: Maneja la referencia a una página virtual

- Miembro *algorithm*: Algoritmo de reemplazo encargado de manejar fallos de página ante falta de marcos libres
- Miembro $freeFrames$: Estructura que almacena marcos de página libres
- Miembro *table*: Estructura que representa internamente la tabla de páginas, asociando cada número de página virtual a su número de marco de página correspondiente

Ante cada referencia a página virtual realizada, se utiliza el método $accessPage()$, cuya implementación es la siguiente:

```
void PageTable::accessPage(const int npv)
{
    if(!SELF.table.contains(npv))
    {
        SELF.missCount++;

        int nmp = -1;

        if(SELF.table.size() ≥ SELF.frameCount)
        {
            int npvToRemove = SELF.algorithm → chooseVirtPage(npv, table);

            nmp = SELF.table[npvToRemove];

            SELF.table.erase(npvToRemove);
        }

        else
        {
            nmp = *SELF.freeFrames.begin();

            SELF.freeFrames.erase(nmp);
        }

        if(nmp == -1)
            throw -1;

        SELF.table[npv] = nmp;
    }

    SELF.algorithm → registerAccess(npv);
}
```

De manera general, el método comienza verificando si la página referenciada está presente en la tabla de páginas. En caso de que esté, se registra su acceso directamente. Por otro lado, en caso contrario, se produce un fallo de página, el cual es manejado, primeramente, aumentando el contador de fallos de página. Luego, si la tabla de páginas no está llena, se le asigna un

marco de página libre, y en caso negativo, se hace uso del algoritmo de reemplazo que se esté usando y, dependiendo de cual sea y qué criterio ocupe, se obtiene una página víctima. Así, se libera su marco de página y se elimina de la tabla, siendo reemplazada por la nueva página. Por último, se registra su acceso.

Es necesario notar que el registro de acceso y la elección de la página víctima son dependientes del algoritmo de reemplazo que se esté usando, y serán explicados en la sección siguiente.

3.2 Algoritmos de reemplazo

Como se mencionó anteriormente, se implementaron cuatro algoritmos de reemplazo, los cuales son: Optimo, FIFO, LRU, LRU Reloj Simple.

Se realizó una abstracción de la siguiente forma:

```
#ifndef ALGORITHM_H
#define ALGORITHM_H

#include <unordered_map>

#include "Type.hxx"

class Algorithm
{
    public:
        virtual int chooseVirtPage(const int npv, const cpp::unordered_map<int, int>& table) = 0;
        virtual void registerAccess(const int npv) = 0;
};

#endif
```

Cada uno de los algoritmos cuenta con los métodos $chooseVirtPage()$ y $registerAccess$. El primero se encarga de aplicar el criterio de búsqueda de la página víctima, y retorna su número de página virtual. Así mismo, el segundo método registra el acceso de cada página virtual. Por supuesto, cada algoritmo le da una implementación distinta a cada método.

A continuación, se detallará cada uno de los algoritmos:

3.2.1 Optimo

Este algoritmo aplica el siguiente principio: "La mejor página víctima es aquella que se referenciará más lejos en el futuro".

Su implementación es la siguiente:

```
class OPT : public Algorithm
{
    public:
        OPT(const cpp::vector<int>& references) : references(references)
        {
            SELF.position = 0;
        }

        int chooseVirtPage(const int npv, const cpp::unordered_map<int, int>& table) override
        {
            int access = 0;
            int newNpv = -1;

            for(const auto& [u, v] : table)
            {
                int firstAccess = SELF.references.size();

                for(int i = position; i < SELF.references.size(); i++)
                {
                    if(references[i] == u)
                    {
                        firstAccess = i;

                        break;
                    }
                }

                if(firstAccess > access)
                {
                    access = firstAccess;
                    newNpv = u;
                }
            }

            return newNpv;
        }

        void registerAccess(const int npv) override
        {
            SELF.position++;
        }

    private:
        int position;

        cpp::vector<int> references;
};
```

En términos generales, cuando se debe elegir una página víctima, se recorren las páginas presentes en la tabla y se busca la próxima aparición de cada una en $references$, almacenando su distancia $firstAccess$. Así, aquella página cuyo acceso sea el más lejano respecto a *position* es la seleccionada para ser reemplazada.

Por último, luego de cada referencia, se registra su acceso avanzando *position*, para así reflejar el estado actual de la lista de accesos.

3.2.2 FIFO

Este algoritmo aplica el siguiente principio: "La mejor página víctima es aquella que se referenció primero entre las actuales, pues es la que lleva más tiempo en memoria".

Su implementación es la siguiente:

```
class FIFO : public Algorithm
{
    public:
        int chooseVirtPage(const int npv, const cpp::unordered_map<int, int>& table) override
        {
            assert(!SELF.queue.empty());

            int newNpv = SELF.queue.front();

            SELF.queue.pop();

            return newNpv;
        }

        void registerAccess(const int npv) override
        {
            if(!SELF.npvsInQueue.contains(npv))
            {
                npvsInQueue.insert(npv);
                SELF.queue.push(npv);
            }
        }

    private:
        cpp::set<int> npvsInQueue;
        cpp::queue<int> queue;
};

#endif
```

Se utiliza una *queue* que almacena las páginas en el orden en el que fueron referenciadas. Al buscar una página víctima, simplemente se extrae la página que se encuentra al frente de la cola, y esta corresponderá a la víctima. Además, para evitar páginas duplicadas, se mantiene un *set* que guarda la información sobre qué páginas están en la cola.

Luego de cada referencia, si la página no está en el *set*, se registra su acceso agregándola a la *queue* y al propio *set*.

3.2.3 LRU

Se aplica el principio: "La mejor página víctima es aquella que no ha sido referenciada durante más tiempo"

Su implementación es la siguiente:

```
class LRU : public Algorithm
{
    public:
        int chooseVirtPage(const int npv, const cpp::unordered_map<int, int>& table) override
        {
            assert(!SELF.list.empty());

            int newNpv = SELF.list.back();

            SELF.list.pop_back();

            return newNpv;
        }

        void registerAccess(const int npv) override
        {
            auto position = cpp::find(SELF.list.begin(), SELF.list.end(), npv);

            if(position ≠ SELF.list.end())
                SELF.list.erase(position);

            SELF.list.push_front(npv);
        }

    private:
        cpp::list<int> list;
};

#endif
```

Para rastrear el uso de las páginas, se utiliza una *list*. Luego, al buscar una página víctima, se devuelve la página que se encuentra al final de la *list*, pues esta representa a la página menos recientemente usada.

Luego de cada referencia, se registra su acceso colocando o moviendo la página referenciada al frente de la *list*, indicando que es la más recientemente usada. En el caso de que la página ya esté en *list*, se elimina de su posición actual antes de ser añadida al inicio.

3.2.4 LRU Reloj Simple

Se aplica un principio similar al algoritmo anterior, con la diferencia de que este utiliza un bit de uso para aproximar la página más recientemente usada.

Su implementación es la siguiente:

```
class CLK : public Algorithm
{
    public:
        CLK(const int size) : accessed(size, false)
        {
        }

        int chooseVirtPage(const int npv, const cpp::unordered_map<int, int>& table) override
        {
            int newNpv = -1;

            while(newNpv == -1)
            {
                if(table.contains(SELF.position))
                {
                    if(SELF.accessed[SELF.position])
                        SELF.accessed[SELF.position] = false;

                    else
                    {
                        newNpv = SELF.position;
                        SELF.position = npv;

                        break;
                    }
                }

                SELF.position += 1;
                SELF.position %= accessed.size();
            }

            return newNpv;
        }

        void registerAccess(const int npv) override
        {
            SELF.accessed[npv] = true;
        }

    private:
        int position = 0;

        cpp::vector<bool> accessed;
};
```

Como se mencionó, esta aproximación emplea un bit de uso para indicar si una página ha sido referenciada recientemente, y un puntero circular (*position*) que recorre las páginas en memoria. Cuando se debe seleccionar una página víctima, se van continuamente verificando los bits de uso de cada página apuntada por *position*. Si el bit de uso está activado, se desactiva y

avanza al siguiente, hasta que encuentre una página con bit de uso desactivado, la cual corresponderá a la víctima.

Finalmente, para registrar el acceso, se marca el bit de uso asociado a la página referenciada como activado.