Proyecto 2 - Sistemas Operativos

María Castillo Acuña, Francy Jelvez Jen, Pedro Palacios Rossi, Diego Venegas Anabalón

November 22, 2024

1 Introducción

A continuación se presentan detalles sobre nuestro proyecto, un simulador que maneja una cola circular de tamaño dinámico compartida entre múltiples hebras y un simulador de memoria virtual con paginación y algoritmos de reemplazo de páginas. El repositorio de nuestro proyecto se encuentra en este enlace.

2 Implementación Parte I

Crear un simulador que maneje una cola circular de tamaño dinámico compartida entre múltiples hebras, se asume que los ítems que se agregan y extraen son de tipo entero.

a) Supuestos

- Usamos integers como los datos almacenados
- los consumidores siempre tienen un tiempo de espera maxima de t segundos

b) Requerimientos

1. El uso de primitivas de sincronización se hace mediante la clase `mutex` y la declaración de la variable mtx que es de ese tipo, en la clase monitor, luego con la sentencia `unique_lock<mutex> lck(mtx);` en los métodos agregarElemento y extraerElemento se asegura que solo una hebra ya sea productora o consumidora ejecute la función que le corresponde a la vez, y se hace uso la primitiva `condition_variable` mediante la declaración de `cv` que se usa en las siguientes sentencias con su respectiva explicación: `cv.wait(lck, [this] return numElementos < numSize; )` cuya función es hacer esperar a la hebra productora si la cola está llena, hasta que tenga espacio disponible , `if (cv.wait_for(lck, tiempoEspera, [this] return numElementos > 0; ))` acá el consumidor debe esperar hasta que existan items para extraer o que el tiempo de espera acabe, `cv.notify_all();` por último se encarga de notificar a las hebras cuando se agregan y extraen elementos en las funciones agregarElemento() y extraerElemento().

2. Cuando todos los productores terminen de agregar elementos, los consumidores deben esperar por un tiempo máximo en segundos (definido como parámetro de entrada a su aplicación) y terminar. Para cumplir con este requerimiento en la función extraerElemento(), los consumidores esperan por un tiempo máximo (tiempoEspera) gracias al uso del método cv.wait_for del objeto cv de clase "condition_variable", en la sentencia `if (cv.wait_for(lck, tiempoEspera, [this] return numElementos > 0; ))`, que se ocupa de hacer que las hebras consumidoras esperen hasta que hayan items en la cola duarante el tiempo designado, si no encuentra items entonces retorna -1.

3. Para manejar la duplicación y reducción del tamaño de la cola (array dinámico) a la mitad, se realizó el siguiente procedimiento:

    - Duplicar tamaño de la cola: Este proceso se realiza mediante la función duplicarCapacidad(), que crea una nueva cola llamada "colaDuplicada" con el doble de la capacidad de la cola actual, luego, se copian los elementos de la cola original ,"cola", a colaDuplicada,

después de copiar los elementos, con la función move se pasan los elementos de colaDuplicada a cola. Luego, se actualiza la variable numSize que representa la capacidad de la cola, asignándole el doble de su valor, además, el frente de la cola una vez duplicada, se restablece al primer elemento y el final de la cola es la cantidad de elementos que tiene la cola "numEelementos". Cuando se agregan elementos a la cola con la función agregarElemento(), en esta misma, primero se verifica con un condicional, si la capacidad actual de la cola "numSize" es igual a la cantidad de elementos actuales "numElementos", en caso de que los valores sean iguales se llama a la función duplicarCapacidad(), para aumentar el tamaño de la cola y que se puedan agregar más elementos.

- Reducir tamaño de la cola a la mitad: Este proceso se realiza mediante la función dividirCola(), que crea una nueva cola llamada "mediaCola" con la mitad de la capacidad de la cola actual, luego, se copian los elementos de la cola original ,"cola", a mediaCola, después de copiar los elementos, con la función move se pasan los elementos de mediaCola a cola. Luego, se actualiza la variable numSize que representa la capacidad de la cola, asignándole la mitad de su valor, además, el frente de la cola una vez reducida a la mitad, se restablece al primer elemento y el final de la cola es la cantidad de elementos que tiene la cola "numEelementos". Cuando se extraen elementos de la cola con la función extraerElemento(), en esta misma, primero se verifica con un condicional, si la cantidad de elementos actuales de la cola (numElementos) es igual a un cuarto de la capacidad actual (numSize) de la cola "numSize", en caso de que los valores sean iguales se llama a la función mediaCola(), para disminuir el tamaño de la cola a la mitad.

4. Para manejar la construcción del archivo `log` que registre los cambios de tamaño de la cola, se creó un objeto de clase ofstream llamado archivo y luego en el constructor de la clase Monitor se abre/crea el archivo que es llamado log.txt y verifica si el archivo se abrió correctamente, de ser cierto,la terminal mostrará el mensaje "archivo abierto" y se escribirá en el archivo la capacidad inicial de la cola, de lo contrario, se imprimirá en la terminal el mensaje "error al abrir el archivo". Luego, cuando se quiere agregar un ítem a la cola con la función agregarElemento() y se verifica que la capacidad actual de la cola es igual a su número de elementos (con el condicional `if (numElementos >= numSize)`), además de duplicarse la cola, se registra en el archivo el evento, mediante la sentencia: `archivo << "Se duplico la capacidad de la cola y ahora es:" << numSize << endl`. Al ejecutar la función extraerElemento(), si se verifica que el numero de elementos (numElementos) de la cola es un cuarto (con el condicional `if (numElementos <= numSize / 4)`) de la capacidad actual de la cola (numSize), además de reducirse a la mitad e tamañon de la cola, se registra en el archivo el evento, mediante la sentencia: `archivo << "Se redujo a la mitad la capacidad de la cola y ahora es:" << numSize << endl;`.

5. La lectura de los parámetros de entrada se hace mediante la variable argc que permite identificar el número de argumentos que tiene el comando de entrada, con **argv se crea un arreglo de punteros a cadenas donde cada elemento es uno de los argumentos de la entrada y atoi se usa para castear a entero el valor "numérico" de cada argumento de entrada, y luego asignarlo a las variables p (número de productores), c (número de consumidores), s (tamaño inicial de la cola), t (tiempo de espera), esta asignación se realiza en la función main con el uso de la siguiente fracción del código:

```
int p = 0, c = 0, s = 0, t = 0;
for (int i = 1; i + 1 < argc; i++) {
    if (strcmp(argv[i], "-p") == 0) {
        p = atoi(argv[i + 1]);
    } else if (strcmp(argv[i], "-c") == 0) {
        c = atoi(argv[i + 1]);
    } else if (strcmp(argv[i], "-s") == 0) {
        s = atoi(argv[i + 1]);
    } else if (strcmp(argv[i], "-t") == 0) {
        t = atoi(argv[i + 1]);
    }
}
```

```
}
```

Donde en el ciclo for se revisa cada argumento del comando de entrada con `strcmp(argv[i], "cadena")` para identificar el tipo de parámetro que se está leyendo y luego en la siguiente posición con `parametro = atoi(argv[i + 1])` se le asigna el valor a la variable del parámetro correspondiente.

3 Implementación Parte II

Crear un simulador de memoria virtual con paginación y algoritmos de reemplazo de páginas.

a) Descripción de la implementación

La implementación desarrollada consiste en una clase llamada `Manejador`, diseñada para simular diferentes **algoritmos de reemplazo de páginas** en sistemas de memoria virtual. Esta herramienta permite evaluar y comparar la eficiencia de los algoritmos en términos de **fallos de página** bajo distintas configuraciones de marcos y secuencias de referencia de páginas.

Manejador

El `Manejador` es la clase principal que coordina y gestiona los procesos relacionados con los algoritmos de reemplazo de páginas. Dentro de esta clase se encuentran las tablas de páginas, que permiten realizar búsquedas rápidas, y los marcos, que representan los espacios de memoria disponibles para almacenar las páginas cargadas. Además, el `Manejador` contiene las estructuras de datos específicas necesarias para la ejecución de cada algoritmo, como colas para `FIFO` y `LRU`, vectores para `LRU con reloj simple` y arreglos para el algoritmo `Óptimo`.

Tabla Hash

La `TablaHash` es una estructura de datos diseñada para mapear páginas virtuales (`npv`) a marcos físicos (`nmp`) de manera eficiente, utilizando un esquema de direccionamiento abierto basado en listas enlazadas.

Cada posición de la tabla se calcula mediante una función de hashing que distribuye las páginas virtuales entre los diferentes índices de la tabla. Las colisiones se resuelven mediante listas enlazadas, donde las páginas que caen en el mismo índice se organizan en una cadena.

La clase proporciona tres operaciones principales:

- `insertItem`: Inserta una nueva página virtual y la asocia con un marco físico.
- `findItem`: Busca una página virtual en la tabla y retorna el marco físico asociado o `-1` si no está presente.
- `deleteItem`: Elimina una página virtual de la tabla, liberando su espacio.

Finalmente, incluye un destructor que garantiza la correcta liberación de la memoria dinámica al eliminar todas las listas enlazadas asociadas. Esta implementación asegura un acceso rápido y eficiente a las páginas necesarias durante la simulación de reemplazo de páginas.

Algoritmos

El sistema implementa cuatro algoritmos de reemplazo de páginas que permiten gestionar eficientemente la memoria principal cuando se presentan fallos de página. A continuación, se describe el funcionamiento de cada uno:

FIFO (First-In, First-Out)

El algoritmo `FIFO` se basa en la idea de reemplazar la página que ha permanecido más tiempo en memoria. Para lograr esto, se utiliza una cola donde las páginas se insertan en el orden en que son cargadas y se eliminan por el frente cuando es necesario liberar espacio. Si la página solicitada ya se encuentra en memoria, no se realizan modificaciones. Si hay espacio disponible, la nueva página se agrega al final de la cola. En caso contrario, se reemplaza la página ubicada al frente de la cola con la nueva página solicitada.

LRU (Least Recently Used)

El algoritmo `LRU` selecciona para reemplazo la página que no ha sido utilizada durante el mayor tiempo. Para implementar esta lógica, se utiliza una lista que almacena las páginas en memoria, ordenándolas por el momento de su última referencia. Cuando una página es solicitada, si ya está en memoria, se actualiza su posición en la lista, moviéndola al final. Si no está en memoria y hay espacio disponible, la página se agrega al final de la lista. En caso de necesitar reemplazo, se elimina la página al frente de la lista, que corresponde a la menos recientemente utilizada, y se introduce la nueva página.

LRU con reloj simple

Este algoritmo es una variante optimizada de `LRU`, donde las páginas se gestionan utilizando un vector de bits y un puntero circular, simulando un reloj. Cada página en memoria tiene asociado un bit que indica si ha sido utilizada recientemente. Cuando ocurre un fallo de página, el puntero avanza hasta encontrar una página cuyo bit esté en 0, indicando que no ha sido referenciada recientemente. Esa página es reemplazada por la nueva, y su bit se actualiza a 1. Si el bit está en 1, este se resetea a 0, y el puntero avanza nuevamente.

Óptimo

El algoritmo `Óptimo` realiza reemplazos basándose en el conocimiento futuro del uso de las páginas, seleccionando para reemplazo la página que no será utilizada durante el mayor tiempo en el futuro. Para implementar este comportamiento, se analiza el patrón de referencias a páginas en el resto de la simulación. Si la página solicitada ya está en memoria, no se realizan cambios. Si hay espacio disponible, la página se agrega. En caso de que sea necesario un reemplazo, se identifica la página cuyo próximo uso esté más lejano o que no sea utilizada nuevamente, y esta se reemplaza por la nueva página.

Interfaz de Usuario

El programa principal (`main`) permite a los usuarios configurar la simulación mediante argumentos de línea de comandos:

- `-m`: Número de marcos disponibles.
- `-a`: Algoritmo a utilizar (`Optimo`, `FIFO`, `LRU`, `LRU_reloj_simple`).
- `-f`: Archivo que contiene las referencias de páginas.

El archivo de referencias se lee y se procesa para simular el acceso a las páginas. Al final, el programa muestra:

- Total de fallos de página.
- Fallos por falta de datos (*cold misses*).
- Fallos por capacidad (*capability misses*).