



UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS
DIVISIÓN DE ELECTRÓNICA Y COMPUTACIÓN



PROGRAMA 7 – PAGINACIÓN SIMPLE

SEMINARIO DE SOLUCIÓN DE PROBLEMAS DE SISTEMAS OPERATIVOS

SECCIÓN: D01

CÓDIGO: 215468696

PROFESOR: VIOLETA DEL ROCIO BECERRA VELAZQUEZ

ALUMNO: DOMÍNGUEZ DURAN OSCAR ALEJANDRO

CARRERA: INGENIERÍA EN COMPUTACIÓN



23 DE JUNIO DE 2021

ACTIVIDAD DE APRENDIZAJE 14 – PAGINACIÓN SIMPLE

OBJETIVO

En este programa trabajaremos con el aspecto de la memoria, como bien recordaremos hasta antes de este programa manejábamos la memoria con tan solo un máximo de 5 procesos, considerando a los estados de ejecución, listo y bloqueado. La idea general de este programa es implementar la técnica de paginación simple para resolver la gestión de la memoria.

La paginación simple es una técnica para administrar la memoria la cual consiste en dividir la memoria en espacios de igual tamaño, a cada espacio se le denomina como *frame*. De esta manera, para que un proceso pueda acceder a memoria, este debe dividirse en partes, a cada parte se le denomina página. El tamaño de una página debe ser igual al tamaño de un marco, porque la idea es que la página enbne en el marco.

Se deberá mostrar la memoria, con la información del número de marco, la cantidad de espacios que se ocupan dentro del *frame*, quién ocupa dicho *frame* y el estado del proceso que esté ocupando el *frame*. Además, en caso de que se muestre toda la memoria en la pantalla, al teclear la tecla “A”, funcionará como una pausa. Opté por mostrar toda la memoria en la misma interfaz debido a que considero que se ve más limpio y entendible. Finalmente, se deberán conservar con todos los requisitos del programa 5, el algoritmo de planificación *Round Robin*.

DESARROLLO

Lenguaje y herramientas utilizadas:

Para este programa decidí retomar tanto a C++ como al *framework* que había estado utilizando hasta el momento, esto debido a que la complejidad de los últimos 2 programas es bastante más elevada que el resto de los programas que habíamos estado realizando, por lo que cambiar a estar alturas del semestre sería una jugada muy arriesgada, además de que, con los finales, no me daría tiempo de optar por algún nuevo lenguaje o herramienta para esta actividad. Además, considerando que era necesario cumplir con los requisitos del programa 5, me ahorraría un poco de tiempo el seguir trabajando sobre de este programa.

Funcionamiento del programa:

TAREA 1 - ARCHIVOS POR LOTES

Lo primero que se realizó para esta actividad fue modificar la clase del “Proceso” para que ahora se puedan almacenar y recuperar los valores correspondientes al tamaño del proceso y la cantidad de *frames* que este va a necesitar en base a su tamaño. El tamaño del proceso es un número aleatorio entre 5 y 25, en donde cada que nosotros fijemos el valor del tamaño en el proceso, podremos obtener cuantas páginas o *frames* vamos a abarcar con ese proceso, mediante una simple formula. Anexo captura de los métodos correspondientes:

```
void Proceso::setTamano(int value)
{
    tamano = value;
    frames = (tamano / PAGES_PER_FRAME) + (tamano % PAGES_PER_FRAME == 0 ? 0 : 1);
}

int Proceso::getTamano() const
{
    return tamano;
}

int Proceso::getFrames() const
{
    return frames;
}
```

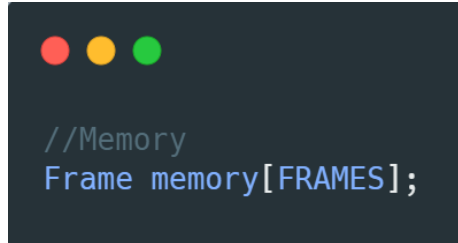
Ilustración 1 - Clase proceso con atributos para el tamaño y los frames

Posterior a la modificación de la clase del “Proceso”, se añadieron a su operador de asignación las asignaciones correspondientes de los nuevos atributos. Para poder manejar de manera adecuada la memoria, decidí tener, como primera opción una matriz de 45 x 4 (o sea, 180 espacios). Debido a que tuve muchas complicaciones tratando de llevar a cabo este acercamiento, opté por cambiar de enfoque a la solución. Me percaté de que no necesitamos saber que espacios dentro de cada página están libres o no, simplemente saber la cantidad de páginas libres que tienen; de esta manera, utilicé un simple *struct* denominado como *Frame* para poder simplificar un poco la concepción de la solución. Este *struct* almacenará la información de la cantidad de páginas libres dentro de ese *frame*, el ID del proceso que esté ocupando dicho *frame* y el estado del proceso; todo esto con la intención de poder mostrar en pantalla de manera correcta lo que está pasando en el programa. Anexo captura de pantalla del *struct* mencionado previamente:

```
struct Frame{
    int freePages = 4;
    int usedBy = FREE_PAGE;
    int state = FREE_PAGE;
};
```

Ilustración 2 - Estructura del Frame

Una vez creado ese *struct* procedí a crear, como atributo de la clase principal, un arreglo de *Frames* con un tamaño de 45. De esta manera, se maneja la misma lógica que establece la memoria, pero de una forma mucho más sencilla debido a que nos basta con tener un campo que nos indique la cantidad de páginas libres que se pueden utilizar:



```
//Memory
Frame memory[FRAMES];
```

Ilustración 3 - Arreglo de frames, representación de la memoria

Ya con la memoria lista, lo siguiente por hacer fue remplazar todas las partes en las que se necesitaba validar el tamaño máximo de la memoria y adecuarlos a que ahora ya no está sujeto a una cantidad límite, sino que la memoria se llene realmente. Para cumplir con lo anterior se crearon una serie de métodos que nos permiten saber si la memoria está llena, la cantidad de espacios que están libres, ingresar la información del proceso en la memoria, o eliminar dicha información.

Comenzando con el primer método llamado: *filledMemory*, nos retorna un *true* en caso de que la memoria esté llena, este solo nos permite validar los casos en los que todos los *frames* estén ocupados, por lo que al inicio no resulta tan útil, pero se puede aplicar para validar ciertas cuestiones a la hora de mostrar la información en pantalla o checar si un nuevo elemento puede entrar. En esta función solo contamos los *frames* en dónde no podemos ingresar información, es decir, los que estén ocupados. Anexo la parte del código correspondiente:



```
bool MainWindow::filledMemory()
{
    int i, filled = 0;

    for (i = 0; i < FRAMES - 3; i++){
        if (memory[i].freePages < PAGES){
            filled++;
        }
    }
    return filled == FRAMES - 3;
}
```

Ilustración 4 - Método para checar si la memoria está llena

El método que más me es útil es el denominado *availableFrames*, este método nos regresa la cantidad de *frames* disponibles en la memoria. De esta manera para agregar un nuevo proceso, podemos comparar si la cantidad de páginas que utiliza es menor o igual a la cantidad de *frames* libres. En el caso en el que no se pueda realizar esto, simplemente no se agrega a la memoria. En este método se contabiliza la cantidad de *frames* cuyo valor en *freePages* sea igual a 4. Anexo el correspondiente:

```
int MainWindow::availableFrames()
{
    int i, availableFrames;

    for (i = availableFrames = 0; i < FRAMES - 3; i++){
        if (memory[i].freePages == PAGES){
            availableFrames++;
        }
    }
    return availableFrames;
}
```

Ilustración 5 - Método para contar la cantidad de marcos disponibles

El otro método indispensable es el de *fillFrames*, el cual se encarga de llenar los marcos correspondientes del proceso, según el tamaño que se haya indicado. Debido a que se pueden encontrar espacios libres no contiguos, es importante checar espacio por espacio para saber en cuales *frames* pueden llenarse. Este método siempre debe ser utilizado tras validar que hay espacios suficientes, de lo contrario no funcionaría. Lo que hacemos es establecer el contador de *freePages* en 0 en el caso de que sea un *frame* completo, o le restamos a *freePages* el residuo del tamaño entre la cantidad de páginas. Anexo la parte del código correspondiente:

```
void MainWindow::fillFrames(int size, int frames, int id, int state)
{
    int i, j;

    for (i = j = 0; i < FRAMES - 3; i++){
        if (memory[i].freePages == PAGES and j < frames){
            if (j == frames - 1){
                if (size % PAGES != 0){
                    memory[i].freePages -= size % PAGES;
                }
                else{
                    memory[i].freePages = 0;
                }
            }
            else{
                memory[i].freePages = 0;
            }

            qDebug() << "Free pages = " << memory[i].freePages;
            memory[i].usedBy = id;
            memory[i].state = state;
            j++;
        }
    }
}
```

Ilustración 6 - Método para llenar los frames correspondientes a un proceso

Tanto al método pasado como a los correspondientes para eliminar o actualizar los *frames* del proceso, es necesario pasarle como argumentos, el ID del proceso y su estado, o en el caso de crearlos, les tenemos que pasar el tamaño y la cantidad de páginas que el proceso utilizará para poder llevar a cabo dicha acción. Si solo queremos actualizar la información del ID, solo se tiene que cambiar el estado de los *frames* que tengan como dueño a ese proceso. En caso de querer eliminar los *frames* de un proceso, reiniciamos sus valores para marcar que esos marcos estarán disponibles. Anexo captura de los métodos indicados:

```
void MainWindow::fillFrames(int size, int frames, int id, int state)
{
    int i, j;

    for (i = j = 0; i < FRAMES - 3; i++){
        if (memory[i].freePages == PAGES and j < frames){
            if (j == frames - 1){
                if (size % PAGES != 0){
                    memory[i].freePages -= size % PAGES;
                }
                else{
                    memory[i].freePages = 0;
                }
            }
            else{
                memory[i].freePages = 0;
            }

            qDebug() << "Free pages = " << memory[i].freePages;
            memory[i].usedBy = id;
            memory[i].state = state;
            j++;
        }
    }
}
```

Ilustración 7 - Métodos para actualizar o eliminar la información de un proceso en memoria

Para reflejar los cambios que la memoria sufre en pantalla, cree un método que me permite actualizar la información en pantalla, específicamente, en la tabla correspondiente a la representación visual de la memoria. Lo que hacemos en este método es recorrer todos los *frames* y mostrar la información correspondiente, por ejemplo, en el caso de que el *frame* 1, esté el ID del proceso 10, tenemos que mostrar en la columna correspondiente que ese *frame* está ocupado por el ID 10; lo mismo para el campo del proceso y el campo del espacio ocupado, en estos casos es solo plasmar la información de manera correcta. En el caso del número de marco, en el mismo ciclo aprovecho para mostrarlo, debido a que el iterador posee dicha información.

Anexo la parte del código correspondiente a este método para mostrar la información en pantalla:

```
void MainWindow::updateMemoryTable()
{
    int i, rows = 0;
    std::string state = "";

    for (i = 0; i < FRAMES - 3; i++, rows++){
        ui->memoryTB->setItem(rows, 0, new QTableWidgetItem(QString::number(i+1)));
        ui->memoryTB->setItem(rows, 1, new QTableWidgetItem(QString::number(PAGES - memory[i].freePages) + "/4"));
        ui->memoryTB->setItem(rows, 2, new QTableWidgetItem(QString::number(memory[i].usedBy)));
        switch (memory[i].state){
            case States::Listo:      state = "Listo";      break;
            case States::Ejecutandose: state = "Ejecución"; break;
            case States::Bloqueado:   state = "Bloqueado";  break;
            default:                  state = "Libre";
        }
        ui->memoryTB->setItem(rows, 3, new QTableWidgetItem(QString(state.c_str())));
    }

    for (i = 42; i < FRAMES; i++){
        ui->memoryTB->setItem(i, 0, new QTableWidgetItem(QString::number(i+1)));
        ui->memoryTB->setItem(i, 1, new QTableWidgetItem(QString("4/4")));
        ui->memoryTB->setItem(i, 2, new QTableWidgetItem(QString("S0")));
        ui->memoryTB->setItem(i, 3, new QTableWidgetItem(QString("")));
    }
}
```

Ilustración 8 - Método para mostrar la información de la memoria en la interfaz gráfica

Finalmente, como ya se mencionó, tuve que modificar las partes en las que se validaba que la memoria tuviese un tamaño fijo de elementos y emplear los métodos creados para poder dejar que los procesos se unieran a la memoria. Por ejemplo, para cargar los primeros procesos, se itera mientras que la cantidad de *frames* del proceso actual sea menor a la cantidad de *frames* disponibles:

```
void MainWindow::loadProcessesMemory()
{
    int process;

    for (process = 0; process < totalProcess; process++){
        Proceso p;
        p = newProcesses[0];
        p.setTiempoLlegada(globalCounter);
        if (p.getFrames() <= availableFrames){
            p.estado = States::Listo;
            fillFrames(p.getTamano(), p.getFrames(), p.getId(), p.estado);
            newProcesses.erase(newProcesses.begin());
            readyProcesses.push_back(p);
            newProcessesCount--;
        }
    }

    if (newProcessesCount == 0){
        ui->nextProcessLB->setText("Ningun proceso nuevo.");
    }
    else{
        ui->nextProcessLB->setText("Siguiente: ID - " + QString::number(newProcesses[0].getId()) + " Tamaño: " +
        QString::number(newProcesses[0].getTamano()));
    }
}
```

Ilustración 9 - Método para cargar los primeros procesos en memoria

Ya lo demás fue simplemente cambiar las validaciones para que cargara un nuevo proceso siempre que hubiera espacio dentro de la memoria. En este punto es donde entran algunos aspectos secundarios como la información del proceso próximo por entrar o presionar la tecla “A” para que pause el programa debido a que ya se muestra toda la memoria (aspectos que se pueden apreciar de manera indirecta en los fragmentos de código mostrados).

A grandes rasgos eso fue lo realizado y la manera en la abordé la solución, para esta práctica no anexo capturas de pantalla del programa siendo ejecutado debido a que prácticamente es difícil de describir todo lo que sucede comuna captura. Es por eso que decidí explayar más en el vídeo que se anexa al final.

CONCLUSIÓN

Considero que esta actividad ha sido la más complicada de realizar y no tanto por los requisitos que se plantean, si no por las fechas en las que fue asignada; al ser finales, he de admitir que me resultó complicado poder dedicarle mucho tiempo al día a esta actividad, por lo que reconozco que salió justo a tiempo debido a que pude terminar con otras materias en los últimos días y enfocar mi atención a esta actividad.

Respecto a lo abordado en el programa, considero que si ha sido uno de los programas más tediosos de llevar a cabo debido a la información de la memoria que debemos estar mostrando en pantalla. Me atoré un poco en la parte de como representar esta memoria, como ya mencioné intenté hacer que esta fuese una matriz, pero al intentar implementar los métodos descritos en el presente reporte, solo terminé más confundido, así que, tras fallar con ese primer acercamiento a la solución del problema, opté por pensar en otra manera de resolverlo la cual fue mediante el uso de ese *struct*.

Finalmente, pese a los problemas y la presión del tiempo presentada en esta actividad, he de reconocer que me fue grato realizarla y, sobre todo, estoy satisfecho con los resultados finales obtenidos. Ya solo queda un programa más, pero considero que, en este punto, estamos más que listos para abordarlo.

Link del vídeo del programa:

- [\(1006\) Programa 7 \(Paginación Simple\) - SSP Sistemas Operativos - YouTube](#)