

**Centro Universitario de Ciencias
Exactas e Ingenierías**

Universidad de Guadalajara

REPORTE 08

Procesos Suspendidos

Alumnos:

Carbajal Armenta Yessenia Paola
Sánchez Lozano Jonathan

Códigos:

220286482
215768126

Profesora:

Becerra Velázquez Violeta del Rocío

Materia:

Seminario de Soluciones de
Problemas de Sistemas Operativos

Departamento:

Ciencias Computacionales

Carrera:

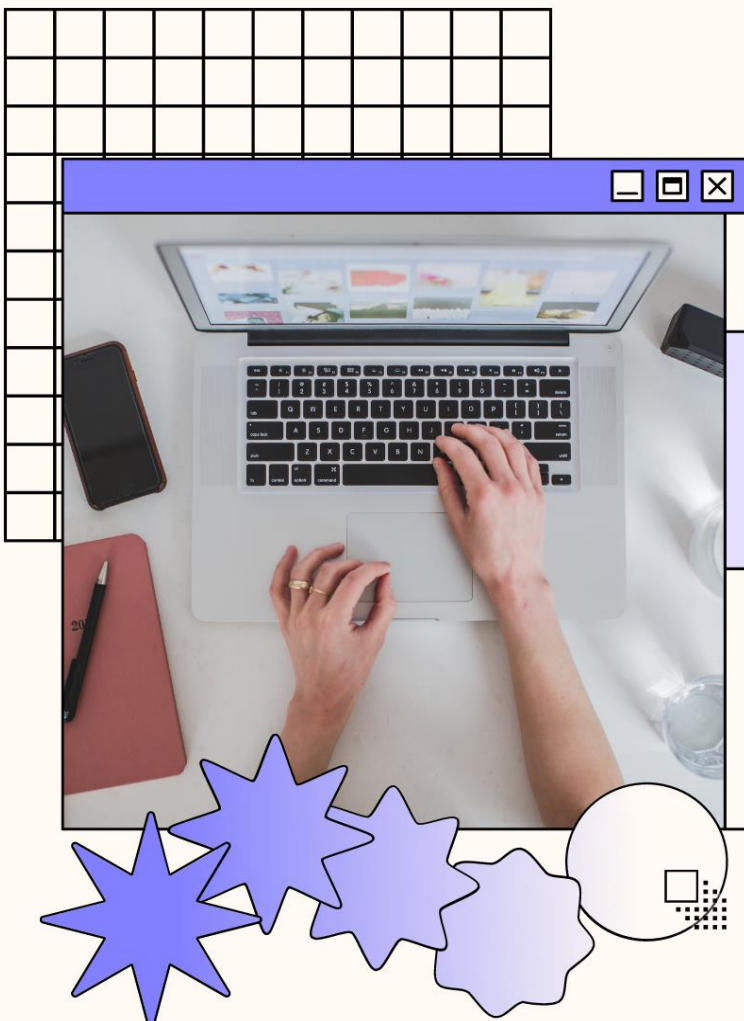
Ingeniería en Computación

NRC:

103844

Sección:

D01



Actividad No. 15

Procesos suspendidos

- Al presionar esta tecla el primero en la cola de bloqueados saldrá de memoria principal e ira a estado suspendido, es decir a disco. Debe generar un archivo con los datos de los procesos suspendidos. Si no hay procesos bloqueados no aplica.
- Si presiona esta tecla el primero en la cola de suspendidos regresara a memoria principal siempre y cuando haya espacio. Si no hay procesos suspendidos no aplica.

Objetivo

Para esta práctica se trabajó con el aspecto de la memoria, con la penúltima práctica que hicimos manejamos la memoria con solo un máximo de 5 procesos, considerando 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.

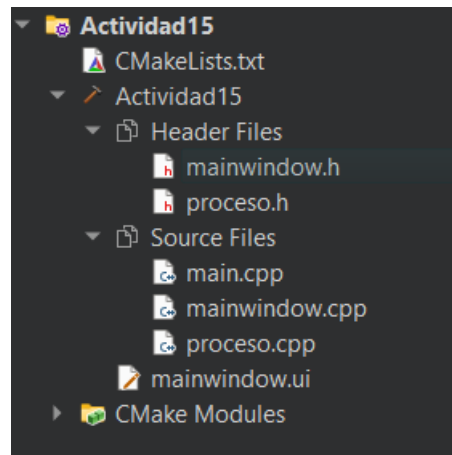
Se deberá mostrar la memoria, con la información del número de marco, la cantidad de espacios que se ocupan dentro del frame, quién ocupa dicho frame y el estado del proceso que esté ocupando el frame. Además, en caso de que se muestre toda la memoria en la pantalla, al teclear la tecla "T".

Finalmente, se deberán conservar con todos los requisitos del programa 5, el algoritmo de planificación Round Robin.

Desarrollo

Para este programa final, consideramos que lo más apropiado era terminar utilizando el lenguaje y herramientas que habíamos utilizado a lo largo del semestre para todos los programas.

Se realizaron los siguientes archivos:



Lo primero que realizamos fue la modificación de la clase “Proceso”, esto debido a que necesitábamos agregar un nuevo estado posible que es el suspendido. Por lo que simplemente añadimos al enum de los estados el nombre de la nueva variable que vamos a utilizar. Este nuevo estado nos va a servir más para el funcionamiento interno que para la interfaz debido a que no se mostrará explícitamente en la pantalla.

```
namespace States {  
    enum {  
        Nuevo,  
        Listo,  
        Ejecutandose,  
        Bloqueado,  
        Finalizado,  
        Suspendido,  
        Count  
    };  
}
```

Posterior a esto, tuvimos que realizar los cambios adecuados al operador de flujo de salida de la misma clase del proceso, realizar cambios debido a que este operador lo sobrecargamos prácticamente desde el tercer o cuarto programa como

una manera de debugear la información del proceso, pero como ahora tenemos que escribirlo a archivo, consideramos que era mejor agregar los campos correspondientes a esta sobrecarga para que realmente todo el proceso sea serializado y así poder cumplir con un estándar de las clases de C++.

```
std::ostream &operator<<(std::ostream &o, Proceso &p)
{
    o << p.getId() << FIELD_SEPARATOR;
    o << p.getOperacion() << FIELD_SEPARATOR;
    o << p.getResultadoOperacion() << FIELD_SEPARATOR;
    o << (p.getFinalizacion() ? 1 : 0) << FIELD_SEPARATOR;
    o << (p.estado ? 1 : 0) << FIELD_SEPARATOR;
    o << p.getQuantum() << FIELD_SEPARATOR;
    o << p.getTamanio() << FIELD_SEPARATOR;
    o << p.getFrames() << FIELD_SEPARATOR;
    o << p.getTiempoEstimado() << FIELD_SEPARATOR;
    o << p.getTiempoTranscurrido() << FIELD_SEPARATOR;
    o << p.getTiempoBloqueado() << FIELD_SEPARATOR;
    o << p.getTiempoLlegada() << FIELD_SEPARATOR;
    o << p.getTiempoFinalizacion() << FIELD_SEPARATOR;
    o << p.getTiempoRetorno() << FIELD_SEPARATOR;
    o << p.getTiempoRespuesta() << FIELD_SEPARATOR;
    o << p.getTiempoEspera() << FIELD_SEPARATOR;
    o << p.getTiempoServicio() << std::endl;

    return o;
}
```

Aquí primordialmente se manejan números, booleanos y cadenas. Lo que se hace es leer la cadena hasta que llegue al separador de campos y según el orden descrito en el operador de flujo de salida, será el orden en el que vayamos leyendo el proceso. Si leemos de un txt todo lo que se lea es una cadena, por lo que se le debe dar el parseo adecuado según el tipo de dato que se pretenda leer.

```

std::istream &operator>>(std::istream &i, Proceso &p)
{
    std::string str;
    i >> p.id;
    i.get();
    getline(i,p.operacion,FIELD_SEPARATOR);
    getline(i,str,FIELD_SEPARATOR);
    p.resultadoOperacion = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.finalizacion = atoi(str.c_str()) == 1 ? true : false;
    getline(i,str,FIELD_SEPARATOR);
    p.estado = atoi(str.c_str()) == 1 ? true : false;
    getline(i,str,FIELD_SEPARATOR);
    p.quantum = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.tamano = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.frames = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Estimado] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Transcurrido] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Bloqueado] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Llegada] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Finalizacion] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Retorno] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Respuesta] = atoi(str.c_str());
    getline(i,str,FIELD_SEPARATOR);
    p.times[Times::Espera] = atoi(str.c_str());
    getline(i,str);
    p.times[Times::Servicio] = atoi(str.c_str());

    return i;
}

```

Una vez tenemos la sobrecarga de estos operadores, realmente ya no queda más que modificar en la clase del proceso por lo que lo siguiente a realizar fue el contemplar las variables y funciones necesarias para implementar las funcionalidades requeridas para el correcto funcionamiento del programa. Para manejar el archivo, se incluyó una variable de tipo fstream la cual nos serviría para manipular el archivo.

El nombre del archivo está definido por una macro con el texto “suspended.txt”. Cada vez que necesitamos tener acceso al archivo, haremos uso de esta variable llamada file. Además de estos, consideré necesario tener un contador de procesos suspendidos para realizar algunas validaciones de manera más sencilla, por

ejemplo, el saber si vale la pena abrir archivo o no, o saber si existen procesos suspendidos sin necesidad de abrir y leer el archivo.

```
std::fstream file;  
int suspendedProcesses;
```

Ya con el contador y el archivo necesarios para el programa, debemos tener en cuenta una validación importante a la hora de estar ejecutando los procesos y esta es: que ya no basta con que haya procesos listos, si no que ahora también tenemos que validar que no haya procesos suspendidos. De esta manera, nuestra función para correr procesos queda de la siguiente manera:

```
void MainWindow::startProcesses()  
{  
    ui->newProcessesLCD->display(newProcessesCount);  
    ui->quantumLCD->display(quantum);  
    loadProcessesMemory();  
    showMemoryTable();  
    while (readyProcesses.size() or suspendedProcesses){  
        showReadyProcesses();  
        runProcess();  
        if (readyProcesses.size() == 0 and (blockProcesses.size() > 0 or suspendedProcesses > 0)){  
            Proceso nullP(0,"NULL",0,false);  
            nullP.setQuantum(0);  
            readyProcesses.push_back(nullP);  
        }  
    }  
}
```

Una vez con esa validación se tiene la certeza de que, si llegamos a tener algún proceso suspendido en archivo, el programa no finalizará y pondrá al proceso nulo a ejecutarse hasta que se decida regresar algún proceso suspendido a la memoria. El método que permite suspender un proceso es activado cada que se presione la tecla "S", este método verifica que la cola de procesos no esté vacía, en caso de que lo esté, simplemente retornamos vacío. Si la cola de procesos vacíos no está vacía podemos abrir el archivo, escribimos el proceso que esté el tope de la cola de procesos bloqueados, actualizamos la memoria y posteriormente eliminamos ese proceso de la cola de bloqueados. Además, actualizamos la etiqueta del siguiente proceso suspendido que puede entrar.

```

void MainWindow::suspendProcess()
{
    if (!blockProcesses.empty()){
        file.open(FILE_NAME, std::ios::out | std::ios::app);
        if (file.is_open()){
            Proceso p;
            p = blockProcesses[0];
            p.estado = States::Suspendido;
            file << p;
            file.close();

            deleteFramesById(p.getId(),FREE_PAGE);
            showMemoryTable();
            blockProcesses.erase(blockProcesses.begin());
            suspendedProcesses++;
            ui->suspendedLCD->display(suspendedProcesses);
            if (suspendedProcesses == 1){
                ui->nextSuspendedLB->setText("Suspendido: ID - " +
                    QString::number(p.getId()) +
                    ", Tamaño: " + QString::number(p.getTamanio()));
            }
        }
    }
    else{
        return;
    }
    showReadyProcesses();
}

```

Pero antes de ver la función que actualiza el archivo, veremos la estructura completa de la función que regresa un proceso a memoria:

```

void MainWindow::returnProcess()
{
    if (suspendedProcesses){
        file.open(FILE_NAME, std::ios::in);
        if (file.is_open()){
            Proceso p;
            file >> p;
            if (p.getFrames() <= availableFrames()){
                suspendedProcesses--;

                p.estado = States::Listo;
                readyProcesses.push_back(p);

                fillFrames(p.getTamanio(), p.getFrames(), p.getId(), p.estado);
                showMemoryTable();

                updateFile();
            }else{
                file.close();
            }
        }
    }
    else{
        return;
    }
    showReadyProcesses();
}

```

En el método para actualizar el archivo lo que se hace es sencillo, debido a que el puntero del archivo se queda justo donde terminó de leer el otro proceso, únicamente creamos un archivo temporal en donde vaciaremos la información de los procesos que haya desde el puntero del archivo principal en adelante. Una vez tenemos toda la información, procedemos a eliminar el archivo antiguo y actualizar el nombre del archivo temporal.

```
void MainWindow::updateFile()
{
    std::fstream tmp;
    tmp.open(TMP_FILE, std::ios::out);
    if (tmp.is_open()){
        std::string str;
        for (int i = 0; i < suspendedProcesses; i++){
            std::getline(file, str);
            tmp << str << std::endl;
        }
        tmp.close();
        file.close();
        std::remove(FILE_NAME);
        std::rename(TMP_FILE, FILE_NAME);
    }
    else{
        return;
    }
    updateNextSuspendLabel();
}
```

Finalmente, la última función implementada se encarga de actualizar la etiqueta en donde ponemos la información del ID y el tamaño del proceso suspendido siguiente a regresar en el caso en el que se presione la tecla correspondiente. En caso de que ya no haya procesos suspendidos entonces mostramos dicho mensaje, Para poder lograr esto, abrimos el archivo de manera general, leemos el proceso que esté al inicio y mostramos dicha información.


```
void MainWindow::updateNextSuspendLabel()
{
    file.open(FILE_NAME);
    if (file.is_open()){
        if (suspendedProcesses){
            Proceso p;
            file >> p;
            ui->nextSuspendedLB->setText("Suspendido: ID - " +
                QString::number(p.getId()) +
                " Tamaño: " + QString::number(p.getTamanio()));
        }
        else{
            ui->nextSuspendedLB->setText("Ningún proceso suspendido");
        }
        file.close();
    }
    else{
        return;
    }
}
```

Conclusiones


Carbajal Armenta Yessenia Paola:

Para esta práctica fue relativamente fácil implementar las nuevas funciones, puesto que solo fue añadir la funcionalidad al presionar 2 nuevas teclas, por lo que no implantó muchas dificultades como la práctica anterior, sin embargo, nos costó un poco realizarla debido al largo tiempo de espera de hacer esta práctica respecto a la pasada.


Sánchez Lozano Jonathan:

Esta práctica no supuso un gran reto ya que no fue tan difícil de hacer como pensábamos porque solo fue implementar algunas funciones nuevas y agregar dos nuevas pulsaciones de teclado, por lo que no nos tomó mucho tiempo el poder finalizar con éxito la práctica.

Enlace del código

 <https://drive.google.com/drive/folders/1CVOMYE0tFNi3u-5-7zcE5UIG4TZoqQsF?usp=sharing>

Enlace del vídeo

 https://drive.google.com/file/d/1WH8JZ9wcd1OgfSDCEFH_u-IEs4x82YSq/view?usp=sharing