



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



PROGRAMA 8 – PROCESOS SUSPENDIDOS

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



30 DE JUNIO DE 2021

ACTIVIDAD DE APRENDIZAJE 15 – PROCESOS SUSPENDIDOS

OBJETIVO

Para este programa se deberá de cumplir con todos los requerimientos del programa anterior, es decir, el programa 8. Como añadido principal se deberá de añadir la posibilidad de poder mandar procesos a un estado de suspendido. El estado de suspendido será simulado mediante una escritura en archivo, esta escritura puede ser mediante el método que se desee, algo a tener en consideración es que este archivo debe poder abrirse durante la ejecución del programa para poder observar los cambios que sufre. Los procesos solo pueden ser suspendidos si estos se encuentran bloqueados, por lo que, si no existe ningún proceso en la cola de procesos bloqueados, el programa no debería hacer nada.

Por otro lado, para poder regresar a memoria un proceso que esté suspendido, es decir, se encuentre en archivo, debe de existir dicho proceso en el archivo. Si el archivo estuviese vacío y se presiona la tecla correspondiente a la acción de regresar un proceso suspendido, el programa no debe hacer nada. Teniendo estos dos aspectos en consideración cabe mencionar que, si se da el caso en el que ya no hay procesos por ejecutar, pero si hay procesos suspendidos, el programa debería quedarse trabajando con un proceso nulo hasta que regresen los procesos suspendidos y sean ejecutados.

DESARROLLO

Lenguaje y herramientas utilizadas:

Para este programa final, consideré apropiado el terminar utilizando el lenguaje y herramientas que llevaba utilizando a lo largo del semestre para todos los programas, a excepción del programa del productor-consumidor, claro está. Además, he de admitir que uno de los lenguajes donde más he practicado la gestión y manejo de archivos es en C++, además de que es uno de los lenguajes más flexibles para este tipo de cosas. Para no perder demasiado tiempo (con el cual no contaba debido a la entrega de finales), decidí trabajar con C++ y el *framework* de Qt. Hasta cierto punto pensé que era necesario terminar utilizando las mismas herramientas que con las que empecé no solo para facilitar el trabajo, sino que también para poder contrastar el avance que obtuve con el desarrollo de todos los programas y también poder observar todo lo nuevo que aprendí sobre este lenguaje y herramienta.

Funcionamiento del programa:

Lo primero que se realizó para esta actividad fue la modificación de la clase “Proceso”, esto debido a que necesitábamos agregar un nuevo estado posible: el estado de suspendido. Por lo que simplemente añadí al *enum* de los estados, uno con el correspondiente nombre. Cabe mencionar que este nuevo estado me sirve más para funcionamiento interno que para la interfaz debido a que no se mostrará explícitamente en la pantalla que el estado de un proceso es suspendido, sino que simplemente este se irá a archivo cuando ese estado cambie. Anexo imagen del *enum* modificado:



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

Ilustración 1 - Estados posibles de un proceso

Posterior a esto, tuve que realizar los cambios adecuados al operador de flujo de salida de la misma clase del proceso, digo realizar cambios debido a que este operador lo sobrecargué prácticamente desde el tercer o cuarto programa como una manera de *debugear* la información del proceso, pero como ahora tenemos que escribirlo a archivo, consideré que era mejor agregar los campos correspondientes a esta sobrecarga para que realmente todo el proceso sea serializado y así cumplir con un estándar de las clases de C++. Anexo la sobrecarga del operador de flujo de salida del proceso:



```
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;
}
```

Ilustración 2 - Operador de flujo de salida del proceso para guardar en archivo

Además del operador de flujo de salida, evidentemente se necesita tener el operador de flujo de entrada para poder leer del archivo un proceso. Es parte de lo que me encanta de C++, que esta sobrecarga de operadores es tan libre que prácticamente podemos interpretar lo que leamos del archivo de cualquier manera. Cómo a qui 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. Anexo la parte del código correspondiente al operador de flujo de entrada del proceso:

```

1  std::istream &operator>>(std::istream &i, Proceso &p)
2  {
3      std::string str;
4      i >> p.id;
5      i.get();
6      getline(i,p.operacion,FIELD_SEPARATOR);
7      getline(i,str,FIELD_SEPARATOR);
8      p.resultadoOperacion = atoi(str.c_str());
9      getline(i,str,FIELD_SEPARATOR);
10     p.finalizacion = atoi(str.c_str()) == 1 ? true : false;
11     getline(i,str,FIELD_SEPARATOR);
12     p.estado = atoi(str.c_str()) == 1 ? true : false;
13     getline(i,str,FIELD_SEPARATOR);
14     p.quantum = atoi(str.c_str());
15     getline(i,str,FIELD_SEPARATOR);
16     p.tamano = atoi(str.c_str());
17     getline(i,str,FIELD_SEPARATOR);
18     p.frames = atoi(str.c_str());
19     getline(i,str,FIELD_SEPARATOR);
20     p.times[Times::Estimado] = atoi(str.c_str());
21     getline(i,str,FIELD_SEPARATOR);
22     p.times[Times::Transcurrido] = atoi(str.c_str());
23     getline(i,str,FIELD_SEPARATOR);
24     p.times[Times::Bloqueado] = atoi(str.c_str());
25     getline(i,str,FIELD_SEPARATOR);
26     p.times[Times::Llegada] = atoi(str.c_str());
27     getline(i,str,FIELD_SEPARATOR);
28     p.times[Times::Finalizacion] = atoi(str.c_str());
29     getline(i,str,FIELD_SEPARATOR);
30     p.times[Times::Retorno] = atoi(str.c_str());
31     getline(i,str,FIELD_SEPARATOR);
32     p.times[Times::Respuesta] = atoi(str.c_str());
33     getline(i,str,FIELD_SEPARATOR);
34     p.times[Times::Espera] = atoi(str.c_str());
35     getline(i,str);
36     p.times[Times::Servicio] = atoi(str.c_str());
37
38     return i;
39 }

```

Ilustración 3 - Operador de flujo de entrada para leer procesos de un archivo

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*

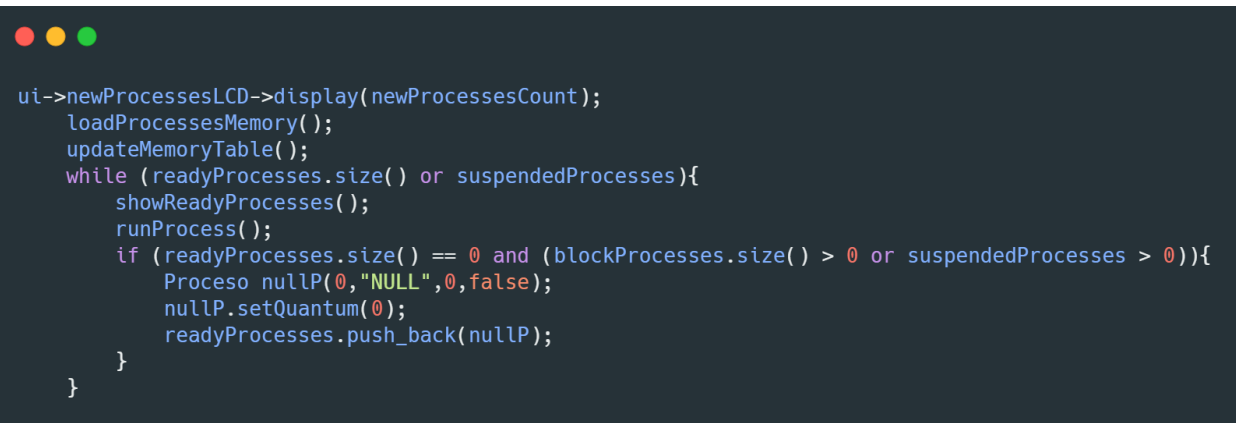
la cuál nos serviría para manipular el archivo. El nombre del archivo está definido por una macro con el texto “suspended.txt”. Cada que necesitamos tener acceso al archivo, haremos uso de esta variable llamada *file*. Además de estos, consideré necesario tener un contador de procesos suspendidos para 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. Anexo una pequeña captura de estas dos variables fundamentales:



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

Ilustración 4 - Variables para el control de los procesos suspendidos

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:



```
ui->newProcessesLCD->display(newProcessesCount);
loadProcessesMemory();
updateMemoryTable();
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);
    }
}
```

Ilustración 5 - Función para correr todos los procesos con la nueva validación

Ya 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 checo 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 este 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. Anexo el código correspondiente a la función descrita:

```
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();
            //Updating memory
            deleteFramesById(p.getId(),FREE_PAGE);
            updateMemoryTable();
            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.getTamaño()));
            }
        }
    }
    else{
        return;
    }
    showReadyProcesses();
}
```

Ilustración 6 - Función para suspender un proceso al presionar la tecla "S"

Para poder recuperar un proceso suspendido del archivo, se debe de pulsar la tecla “R”. Pulsando dicha tecla, mandamos a llamar a la función que se encarga de regresar un proceso suspendido a memoria. La primera validación que esta función realiza es checar que, si haya procesos suspendidos, en caso de que no haya simplemente retornamos vacío. En el caso de que, si exista algún proceso suspendido, abrimos el archivo en modo lectura, obtenemos el primer proceso y verificamos que la cantidad de *frames* que requiere sea menor o igual a la cantidad de *frames* disponibles en memoria, en caso de que no se pueda meter en memoria cerramos el archivo y listo. Si hay espacio en memoria entonces llenamos los *frames* correspondientes, cambiamos el estado del proceso y lo insertamos en la cola de listos. Posterior a esto, actualizamos el archivo para que ya no esté la información del proceso que se acaba de leer y regresar a la memoria.

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--;
                //Returning in ready processes
                p.estado = States::Listo;
                readyProcesses.push_back(p);
                //Updating table
                fillFrames(p.getTamaño(), p.getFrames(), p.getId(), p.estado);
                updateMemoryTable();
                //Updating file
                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.

Ilustración 7 - Función para regresar un proceso a la memoria

Una vez tenemos toda la información, procedemos a eliminar el archivo antiguo y actualizar el nombre del archivo temporal. Anexo captura de pantalla de la función descrita:

```
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();
}
```

Ilustración 8 - Función para actualizar la información del archivo cada que se regresa un proceso suspendido a memoria

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. Anexo captura de pantalla correspondiente a la función descrita:

```
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.getTamaño()));
        }
        else{
            ui->nextSuspendedLB->setText("Ningún proceso suspendido");
        }
        file.close();
    }
    else{
        return;
    }
}
```

Ilustración 9 - Función para actualizar la información de la etiqueta que muestra el siguiente proceso suspendido a entrar

Eso sería toda la explicación de que fue lo que se realizó y modificó respecto al último programa realizado. Debido a que obtener capturas de pantalla como tal del momento exacto en el que suceden las cosas puedo poner una captura de pantalla del archivo creado y su información a manera de ejemplo para poder apreciar el como la sobrecarga de operador de flujo de salida hace su trabajo:

The screenshot displays a 'SIMULADOR DE PROCESOS' application. At the top, there are two input fields: 'Nuevos procesos:' with the value '2' and 'Procesos suspendidos:' with the value '2'. Below these, it shows 'Sigiente: ID - 11, Tamaño: 11' and 'Suspendido: ID - 1, Tamaño: 5'. The main area contains three tables:

- Listos:** A table with columns ID, TME, and TT. It contains rows for IDs 4, 5, 6, and 7.
- Proceso en ejecución:** A table with columns ID, OPE, TME, TT, and TR. It contains rows for ID 3, with OPE 0*70, TME 9, TT 4, and TR 5.
- Terminados:** A table with columns ID, OPE, and RL. It is currently empty.

At the bottom, a terminal window shows the output of the simulation, including process IDs and timing information.

Ilustración 10 - Programa corriendo y mostrando la funcionalidad implementada

CONCLUSIÓN

Esta actividad no resultó ser tan complicada como en un principio me lo llegué a imaginar, de hecho, el programa previo a este si me tomó mi tiempo y mis dolores de cabeza. En este programa el mayor problema que tuve (como casi todos los programas que realicé) fue el tiempo, y no tanto porque se nos diera poco tiempo, sino que entrados los finales del semestre me era imposible el poder administrar bien mis tiempos. Las fechas de entrega se traslapaban y darle prioridad a una tarea u a otra era un dilema complicado, aunque, ahora que lo pienso, dicho problema se ve reflejado en el como se deben administrar los procesos de una computadora; muchas veces los recursos no son el problema per se, sino el como estos deben ser gestionados para ser finalizados; una reflexión irónica llegados a este último programa, pero curiosa cuanto menos.

Respecto al programa, considero que abordarlo en la manera en la que lo hice fue lo mejor no solo para la representación visual de como se almacenan los procesos en el archivo, sino que también la más interesante de implementar. Guardar la información en archivos binarios es mucho más sencillo cuando tenemos instancias de una clase con tipos de datos primitivos, pero consideré que cuando se abre un archivo binario la información es difusa y necesitamos un editor de textos que sea capaz de interpretar los bits en texto. Además, algo que me fascina de las clases de C++ es la libertad que nos brinda el poder hacer que un objeto pueda ser escrito y leído mediante la sobrecarga de operadores de flujo, así que, considero que la solución abordada fue la mejor y la más práctica para los fines del programa.

Estoy bastante satisfecho con el último programa realizado, considero que este último programa fue bastante ameno de llevar a cabo y que le agrega mucho al simulador de procesos que habíamos estado realizando a lo largo del semestre puesto que ahora si que contemplamos todos los estados que puede tener un proceso dentro del sistema operativo. En general, estoy bastante contento con los temas vistos en el curso, los programas realizados y con los resultados obtenidos a lo largo del semestre. He de reconocer que hay muchos aspectos que se pueden pulir del código que realicé en cuanto a optimización o limpieza, pero también soy consciente de que siempre codifiqué lo mejor que pude la solución a cada problema planteado y eso es lo que me llena. Sin más que añadir, agradezco la oportunidad de haber tomado este curso, siento que aprendí muchas cosas nuevas no solo de sistemas operativos, sino que pude descubrir nuevas cosas de un lenguaje y herramienta que, inocentemente, consideraba que ya dominaba,

Link del vídeo del programa:

- [Programa 8 \(Procesos suspendidos\) - SSP Sistemas Operativos - YouTube](#)