

**Centro Universitario de Ciencias
Exactas e Ingenierías**

Universidad de Guadalajara

REPORTE 03
**Algoritmo de
planificación FCFS**

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:

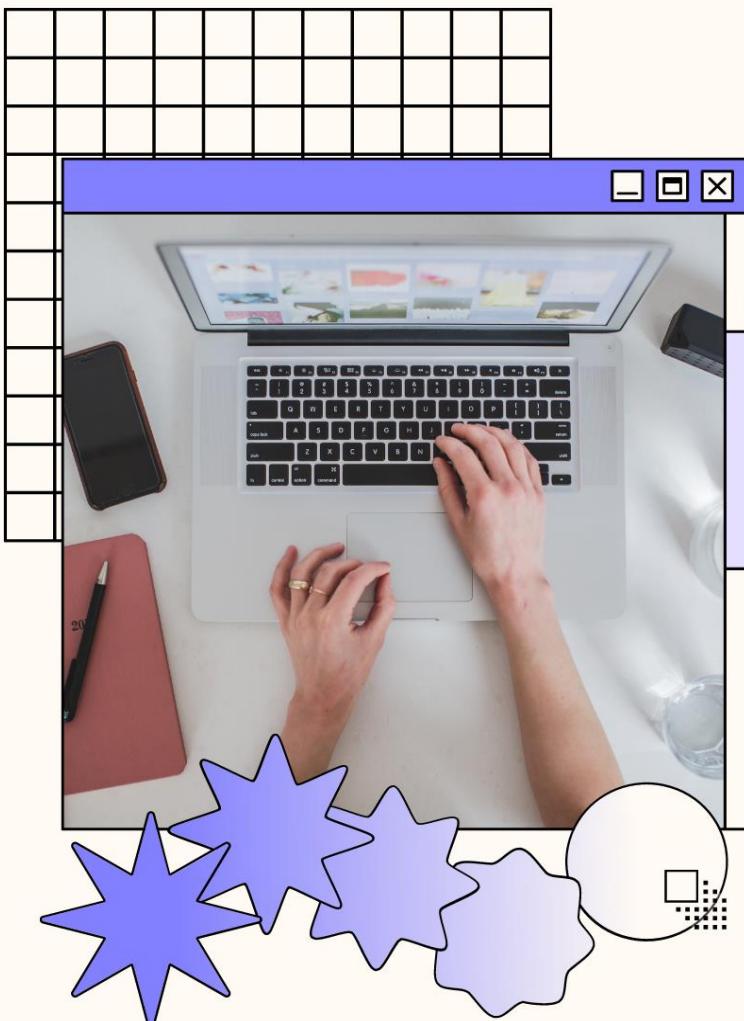
Ingeniería en Computación

NRC:

103844

Sección:

D01



Guadalajara, Jal. a 25 de septiembre del 2022

Actividad No. 6

Algoritmo de planificación FCFS (First Come First Server)

Introducir desde teclado N procesos, estos serán los que conformen el número de procesos a terminar llevando a cabo la implementación de un algoritmo de planificación FCFS, el cual consiste en ejecutar automáticamente aplicaciones y procesos de cola en orden de llegada.

Objetivo

Para este programa respecto al último que realizamos, es que se deja de lado todo lo relacionado con los lotes, es decir, ya no vamos a trabajar con todos los procesos separados de 3 en 3 sino que ahora todos los procesos serán independientes de pertenecer a un lote. Este cambio representa una significativa diferencia en la manera en la que se debe abordar el programa, ya que ahora se deben contemplar más factores, por ejemplo, todos los tiempos que se generan en el ciclo de vida de cada proceso.

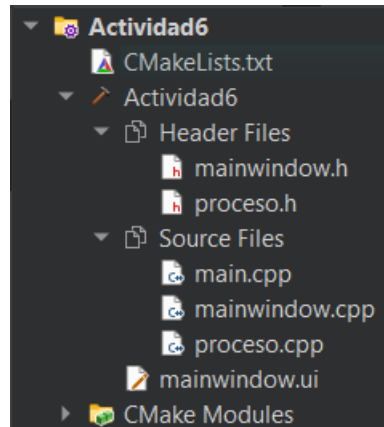
Desarrollo

Para el desarrollo de la práctica decidimos usar el lenguaje de programación C++ en complemento con el framework QT. Seleccionamos este lenguaje ya que, además de ser con el que más cómodo nos sentimos, nos da un campo más amplio a la hora de manejar con instrucciones de nivel un poco más bajo, en comparación de Python u otros lenguajes de nivel superior que tienden a obviar algunas partes.

Se desarrolló el programa con programación orientada a objetos y con la funcionalidad de una interfaz gráfica para una mayor comodidad, se crearon

instancias para cada para cada proceso y así poder almacenarlos por lotes para facilitar su proceso y así poder simularlo de una manera más real.

Se terminaron creando los siguientes archivos:



Los cambios que sufrió nuestro programa para esta actividad fue que los procesos eran almacenados en un vector dentro de vectores, en donde cada vector representaba un lote.

```
vector<Proceso> newProcesses;  
vector<Proceso> readyProcesses;  
vector<Proceso> blockProcesses;  
vector<Proceso> finishedProcesses;
```

Como al inicio solo podemos cargar 3 procesos, basta con tomar los primeros 3 de este arreglo y pasarlos a la cola de procesos listos; lo cual significa que serían eliminados del vector correspondiente al ser añadidos a la cola. Realizamos una función que se encargue de lo anterior, así como de asignar el valor correspondiente al contador de procesos nuevos:

```
void MainWindow::loadProcessesMemory(){  
    for ( int i = 0; i < MAX_PROCESSES_IN_MEMORY and i < totalProcess; i++ ){  
        Proceso p;  
        p = newProcesses[0];  
        p.setTiempoLlegada(globalCounter);  
        newProcesses.erase(newProcesses.begin());  
        readyProcesses.push_back(p);  
    }  
    totalProcess > MAX_PROCESSES_IN_MEMORY  
        ? newProcessesCount = newProcessesCount - MAX_PROCESSES_IN_MEMORY  
        : newProcessesCount = 0;  
}
```

La clase correspondiente a moldear un proceso sufrió una serie de cambios, siendo el más relevante la incorporación de un arreglo con todos los tiempos necesarios. La clase "Proceso" termina teniendo los atributos utilizados en programas anteriores con el añadido del arreglo y una bandera, la cual será descrita más adelante. Adjunto imagen correspondiente a los atributos de la clase:

```
class Proceso
{
private:
    int id;
    string operacion;
    int resultadoOperacion;
    bool finalizacion;
    bool ejecutado;
    int tiempoMaximoEstimado;
    int times[Times::Count] = {0};
```

Cuando un proceso empieza a ejecutar lo primero que se hace es tomar el elemento tope de la cola de procesos listos, por cada proceso que vamos a correr, tenemos que marcar que esté se ha ejecutado mediante su bandera, ya que es necesario asignar su tiempo de respuesta, el cual se calcula tomando el contador global en ese momento y restando él tiempo de llegada, este tiempo solo debe ser calculado una sola vez cuando este esté se ejecuta por primera ocasión. Lo anterior está descrito en las siguientes líneas de código:

```
void MainWindow::runProcess()
{
    qDebug() << "Ejecutado: " << readyProcesses[0].getEjecutado();

    readyProcesses[0].setFinalizacion(SUCCESSFUL_FINISH);
    if ( !readyProcesses[0].getEjecutado() ){
        readyProcesses[0].setEjecutado(true);
        qDebug() << "ID: " << readyProcesses[0].getId();
        qDebug() << "Tiempo llegada: " << readyProcesses[0].getTiempoLlegada();
        readyProcesses[0].setTiempoRespuesta( globalCounter - readyProcesses[0].getTiempoLlegada() );
        qDebug() << "Tiempo Respuesta: " << readyProcesses[0].getTiempoRespuesta();
    }

    tT = readyProcesses[0].getTiempoTranscurrido();
    tR = readyProcesses[0].getTiempoEstimado() - tT;
    interrupted = error = false;
```

Estas partes del código es para poder incrementar o decrementar los tiempos asignados cuando los procesos están en bloqueados:

```

if ( readyProcesses[0].getId() != NULL_PROCESS ) {

    while (tT < readyProcesses[0].getTiempoEstimado() and tT != ACTION_CODE ){
        tR--;
        tT++;
        readyProcesses[0].setTiempoTranscurrido(tT);
        ui->processRuningTB->setItem(3,0,new QTableWidgetItem(QString::number(readyProcesses[0].getTiempoTranscurrido())));
        ui->processRuningTB->setItem(4,0,new QTableWidgetItem(QString::number(tR)));

        if ( !blockProcesses.empty() ) {
            incrementBlockedTimes();
        }
        showBlockedProcesses();
        globalCounter++;
        ui->globalCountLCD->display(globalCounter);
        ui->newProcessesLCD->display(newProcessesCount);
        delay();
    }
}

```

```

void MainWindow::incrementBlockedTimes()
{
    for (auto it = blockProcesses.begin(); it != blockProcesses.end();){
        (*it).setTiempoBloqueado((*it).getTiempoBloqueado() + 1);

        if ((*it).getTiempoBloqueado() == BLOCKED_TIME){
            Proceso p;
            p = *it;
            p.setTiempoBloqueado(0);
            readyProcesses.push_back(p);
            blockProcesses.erase(it);
            showReadyProcesses();
        }
        else{
            it++;
        }
    }
}

```

Esta sección del código fue para poder añadir un nuevo proceso a la cola de los procesos listos:

```

if ( newProcesses.size() ) {
    Proceso p;
    p = newProcesses[0];
    newProcesses.erase(newProcesses.begin());
    p.setTiempoLlegada(globalCounter);
    readyProcesses.push_back(p);
    newProcessesCount--;
}

```

En el siguiente apartado de código, se puede observar el método que utilizamos para poder calcular los tiempos necesarios para cada proceso:

```

void MainWindow::calculateProcessesTimes(){
    for (int i = 0; i < (int)finishedProcesses.size(); i++ ) {
        Proceso& p = finishedProcesses[i];
        p.setTiempoRetorno(p.getTiempoFinalizacion() - p.getTiempoLlegada());
        p.setTiempoServicio(p.getTiempoTranscurrido());
        p.setTiempoEspera(p.getTiempoRetorno() - p.getTiempoServicio());
    }
}

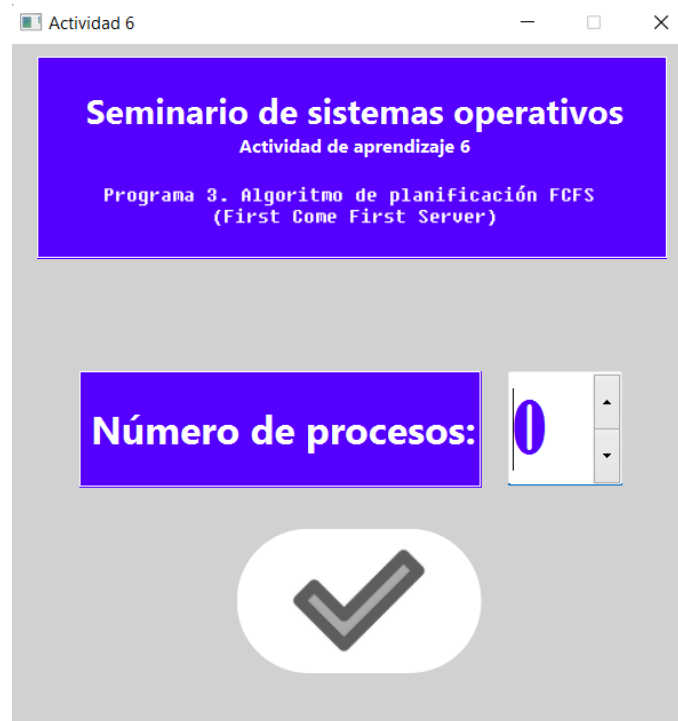
```

Algo que añadimos es que en la actividad es que cuando no hay ningún proceso en ejecución se crea un proceso nulo y cuando un proceso va a entrar, el proceso nulo en automático se elimina para no afectar al programa:

```
while ( readyProcesses.size() ){  
  
    showReadyProcesses();  
    runProcess();  
    if ( readyProcesses.size() == 0 and blockProcesses.size() > 0 ){  
        Proceso nullP( 0, "NULL", 0, false );  
        readyProcesses.push_back(nullP);  
    }  
  
}
```

```
while ( readyProcesses.size() < 2 ) {  
    incrementBlockedTimes();  
    globalCounter++;  
    showBlockedProcesses();  
    ui->globalCountLCD->display(globalCounter);  
    ui->newProcessesLCD->display(newProcessesCount);  
    delay();  
}
```

Finalmente, nos encontramos con la interfaz gráfica, donde se editaron los objetos y el diseño de estos para obtener una GUI visualmente amigable e intuitiva de usar y así obtener nuestras pantallas finales de ejecución.



Conclusiones

Carbajal Armenta Yessenia Paola:

Es una actividad de bastante importancia en el funcionamiento del algoritmo FCFS, pues nos ayuda a comprenderlo de una mejor manera si queremos comprender las interrupciones y los estados que se manejan dentro de un planificador sencillo como lo es FCFS.


Cabe recalcar que, a pesar de haber hecho cambios en el código del programa, ya teníamos una base previa para realizar esta actividad debido a que el programa anterior ya poseía varias de estas opciones, y solamente era ir almacenando sus tiempos y manejar su estado.

Sánchez Lozano Jonathan:

Con esta actividad nos dimos cuenta de cómo es el funcionamiento del algoritmo FCFS, ya que nos ayuda a entender el funcionamiento de las interrupciones y cómo se manejan los procesos de manera independiente sin tener que estar dentro de un conjunto o dentro de un lote.

No se nos hizo tan complicada la actividad, ya que de lo que teníamos de la anterior fuimos modificando conforme se fuera requiriendo para llegar a lo establecido, aunque sí hubo varios momentos en los que se nos dificultó un poco porque no sabíamos cómo hacer las cosas, pero con investigación pudimos resolverla sin mayor problema.

Enlace del código

 https://drive.google.com/drive/folders/1ujxTwF8uUcradMWE7goA3_BURH46oPc?usp=sharing

Enlace del vídeo

 https://drive.google.com/file/d/1Du1OBjHOx7OIBqIOwUwaIRujVVTI_bRB/view?usp=sharing