Valdez Mondragón Erik
Profesora: Ing. Patricia Del Valle Morales
Sistemas Operativos *(0840)*
Grupo 3
Universidad Nacional Autónoma de México.
octubre 25, 2018

Documentación sobre el Proyecto de software "Simulador de procesos, algoritmo Round Robin"

Definición general del proyecto y especificaciones

El desarrollo de este software consiste en crear un simulador de procesos computacionales en modo texto utilizando el algoritmo de planificación de procesos *Round Robin*. Dicho simulador le brinda al usuario la posibilidad de elegir la cantidad de procesos a simular y el *quantum* de ejecución simulado en la CPU. Por cada proceso se debe ingresar el nombre de este, el tamaño del proceso, su tiempo de llegada al sistema y la duración de su ráfaga de CPU. La información ingresada formará parte del bloque de control de cada proceso.

El objetivo del proyecto es brindar al usuario una forma intuitiva de estudiar y comprender paso a paso el funcionamiento del algoritmo de planificación *Round Robin* observando mediante una estructura de datos de colas implementadas como listas ligadas sencillas los movimientos de procesos entre el almacenamiento secundario, la memoria de acceso aleatorio y la CPU en tiempo real (segundos) y obtener promedios de tiempos de ejecución, espera y de respuesta con el fin de obtener una idea de eficiencia del algoritmo comparado con otros como FCFS, SJF, por Prioridad, etc.

Round Robin es un algoritmo de planificación de procesos simple de implementar, dentro de un sistema operativo se asigna a cada proceso una porción de tiempo equitativa y ordenada, tratando a todos los procesos con la misma prioridad. En sistemas operativos la planificación de Round Robin da un tiempo máximo de uso del procesador a cada proceso, pasado el cual es expropiado y regresado al estado de "listo". Dicha lista se planifica por FIFO (el primero en entrar será el primero en salir)

Especificación de requerimientos de software

El proyecto consta de cuatro fases de desarrollo:

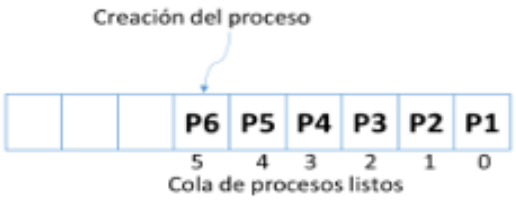
1) Creación – generación de procesos
2) Planificación de procesos utilizando el algoritmo *Round Robin*
3) Asignación dinámica de memoria
4) Planificación de CPU

Generación de procesos

Implementar una interfaz (consola) para la captura de datos de los procesos con los cuales se simulará el planificador. Los datos de los procesos serán: ID del proceso único, nombre del proceso, tamaño del proceso, tiempo requerido para su ejecución y tiempo de llegada. Todos los tiempos se simularán en milisegundos y la cola de procesos listos se implementará utilizando una lista dinámica de nodos.

Conforme se crean los procesos se irán insertando en dicha lista y, conforme se van planificando, se eliminarán de la cola.

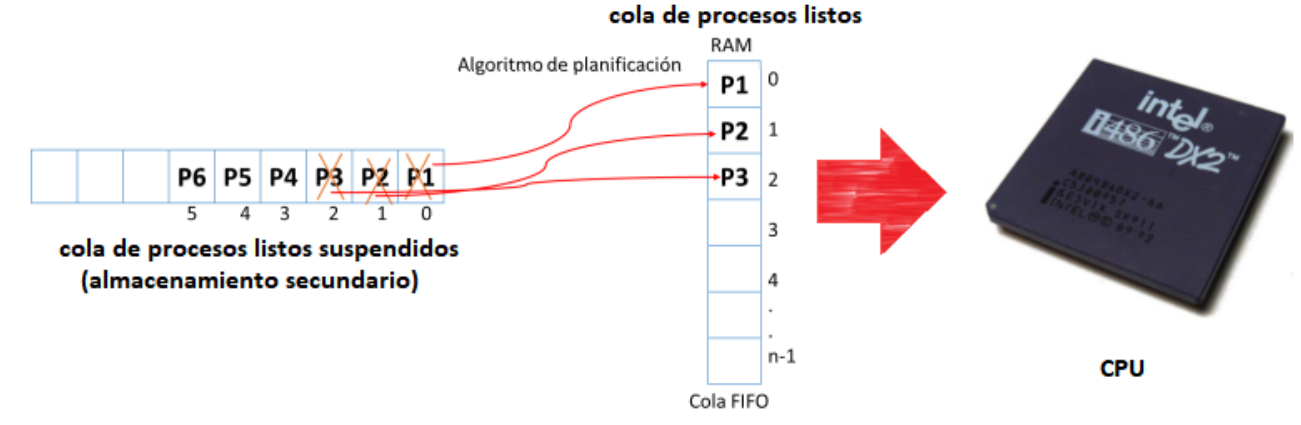
Cada cambio (insertar o borrar) realizado en esta lista, se deberá mostrar el nuevo contenido de la cola.

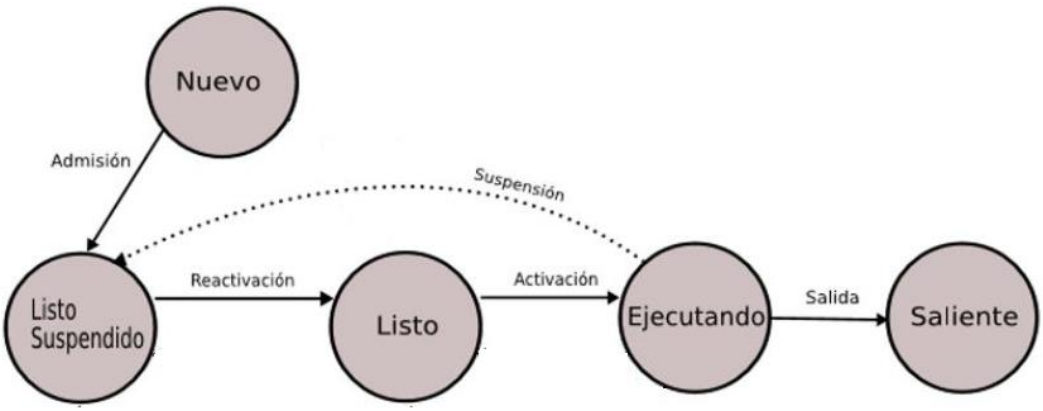


Planificación de procesos utilizando el algoritmo *Round Robin*

Se simularán el algoritmo de planificación de planificación *Round Robin* a los procesos que están en la cola de procesos listos para ser ejecutados.

Se muestra a continuación el diagrama con los niveles de planificación:





Asignación dinámica de memoria

Cuando un proceso se planifica se carga en la memoria y se crea dinámicamente su partición.

No se simulará la partición dinámica de memoria sino, bastará con validar si hay espacio suficiente para cargar el proceso (tamaño restante de memoria – tamaño del proceso a planificar) mostrando en pantalla el espacio de memoria restante una vez cargado el proceso.

Se cargarán los procesos en memoria hasta que se llene o ya no exista espacio suficiente para almacenar más procesos en estado de listos. Para ello se implementará una cola FIFO de procesos listos para ejecución (RAM). Los procesos almacenados en esta cola son los que subirán a la CPU.

Salida a pantalla:

a) Por cada cambo que exista en la lista de procesos listos para su ejecución deberá imprimir su contenido.
b) Cada vez que se libere espacio en memoria se deberá de imprimir el total del espacio disponible.

Ejemplo:

Suponer un tamaño de memoria disponible para el usuario de 1024k.

Los procesos son: P1 = 200k, P2 = 300K y P3 = 150K:

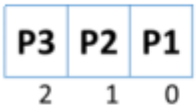
- El simulador deberá presentar a pantalla la siguiente información conforme se van simulando la carga de procesos a la memoria

Subió el proceso P1 y restan 824 unidades de memoria
Subió el proceso P2 y restan 524 unidades de memoria
Subió el proceso P3 y restan 374 unidades de memoria

- El simulador presentará a pantalla la información de las colas de procesos:

cola de procesos LISTOS (en memoria secundaria, listos para subir a RAM)

cola de procesos LISTOS PARA EJECUCIÓN (RAM)

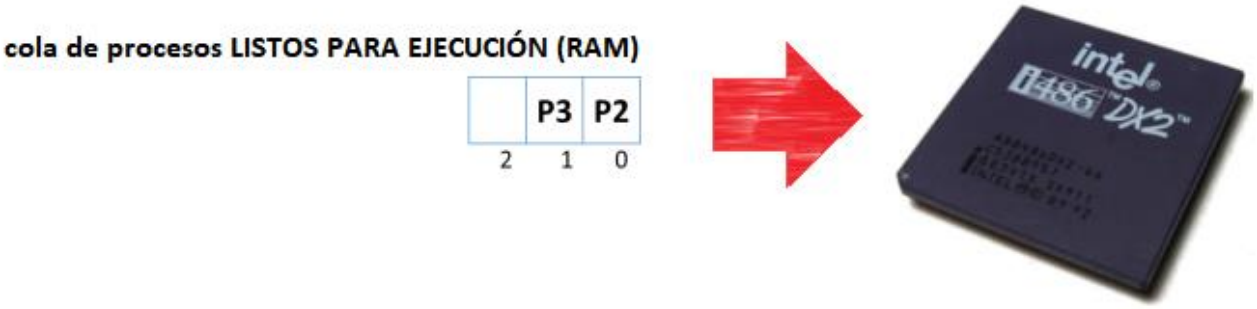


Planificación de CPU

Se simulará el algoritmo de planificación indicado en el inciso 2, se toma el primer proceso de la cola de procesos listos para ejecución y se le asigna la CPU.

Ejemplo: Simulación de la ejecución de los procesos en una CPU, para el algoritmo de *Round Robin*

CPU. Considere que P1 tiene una ráfaga de CPU de 10[ms] y el quantum por proceso es de 6[ms]



P1 en ejecución 10[ms]

P1 en ejecución 9[ms]

P1 en ejecución 8[ms]

P1 en ejecución 7[ms]

…

P1 en ejecución 4[ms] y sale de la CPU por finalizar su **quantum**

El proceso debe ejecutarse según el quantum elegido para el algoritmo de planificación y una vez que finaliza dicho tiempo, sale de la CPU y se debe formar nuevamente en la *cola de procesos LISTOS*, para esperar su próxima ejecución y subir nuevamente a la memoria (planificador mediano plazo) y después subir a la CPU (planificador a corto plazo).

Los resultados finales a obtener son los movimientos entre colas de la simulación y el cálculo de los tiempos promedio siguientes:

1) Tiempo de espera
2) Tiempo de ejecución
3) Tiempo de respuesta.

El lenguaje de programación a utilizar será Java

Obtención de requerimientos funcionales y no funcionales del proyecto

ID	Requerimiento	Tipo	Prioridad
R1	Implementar una interfaz para la captura de datos de los procesos con los cuales se simulará el planificador y un resumen de estos.	Funcional	MEDIA
R2	Implementar una interfaz para elegir el tamaño del quantum	Funcional	MEDIA
R3	La cola de procesos listos se implementará con una lista dinámica de nodos	Funcional	ALTA
R4	Programar los métodos y atributos requeridos para el manejo de las colas (insertar, borrar, insertar al final, verificar si es vacía, manejo de memoria dinámico)	Funcional	ALTA
R5	Diseñar el algoritmo de planificación Round Robin e implementarlo en una clase con sus respectivos métodos y atributos.	Funcional	ALTA
R6	Implementar colas auxiliares para un mejor manejo de la simulación	No funcional	MEDIA
R7	Mostrar en cada iteración de tiempo los movimientos existentes en las colas de LISTOS, RAM y CPU.	Funcional	ALTA
R8	Implementar interfaz para permitir al usuario omitir el detalle de ingreso memoria del proceso y tiempo de llegada cuando se desea simular solamente los movimientos entre RAM y CPU.	No funcional	BAJA
R9	Implementar opción para elegir si se requiere simulación en tiempo real (segundos) o inmediata.	No funcional	BAJA
R10	Realizar el cálculo de los tiempos de ejecución, espera y respuesta al final de la simulación	Funcional	ALTA

Las limitaciones del simulador de procesos son las siguientes

1. La memoria RAM simulada está fijada a 1024MB.
2. Si se elige simular despreciando el tamaño personalizado de cada proceso, se le asignará una memoria *por default* a cada uno de 16MB. Esto permite que entren hasta 64 procesos en la cola de RAM sin hacer uso intensivo de la cola de procesos listos (almacenamiento secundario) ya que inmediatamente los procesos que lleguen a ella se planificarán
3. No pueden simularse procesos mayores a 1024MB cada uno, pero sí menores a esa cantidad (se quedarán en el estado de "listos-suspendidos" hasta que se libere RAM.

Este proyecto es *totalmente original* y no forma parte de ningún sistema desarrollado con anterioridad salvo el uso del algoritmo de planificación utilizado.

¿Qué se requiere para poder ejecutar este software?

Este software fue creado y compilado utilizando el Kit de Desarrollo Java, versión 1.8 (JDK 1.8.0_112-b15) y NetBeans 8.2 sobre Windows 10, 64-bit.

Se requiere la instalación de J2SE en su versión 8 Update 191 o posterior disponible en https://www.java.com/es/download/

El software se entrega con su código fuente empaquetado y el ejecutable en formato JAR el cual se puede ejecutar desde la consola de Windows utilizando el siguiente comando (sustituyendo los puntos por la ruta en dónde se guarda el ejecutable):

java -jar "C:\...\RoundRobinSimulator.jar"

Definición general de este proyecto

La funcionalidad principal del software es simular la ejecución de procesos utilizando el algoritmo de planificación *Round Robin* y, el objetivo de crear este software es facilitar el entendimiento de este algoritmo. Con el desarrollo de este software se cumple con un proyecto escolar impartido en la clase de Sistemas Operativos impartido por la Ing. Patricia del Valle Morales en la Facultad de Ingeniería de la UNAM.

Este software está dirigido a quien se encuentre actualmente estudiando el funcionamiento de los sistemas operativos, en específico los temas sobre procesos y su planificación por parte de estos.

Información sobre los procedimientos

Procedimientos de desarrollo

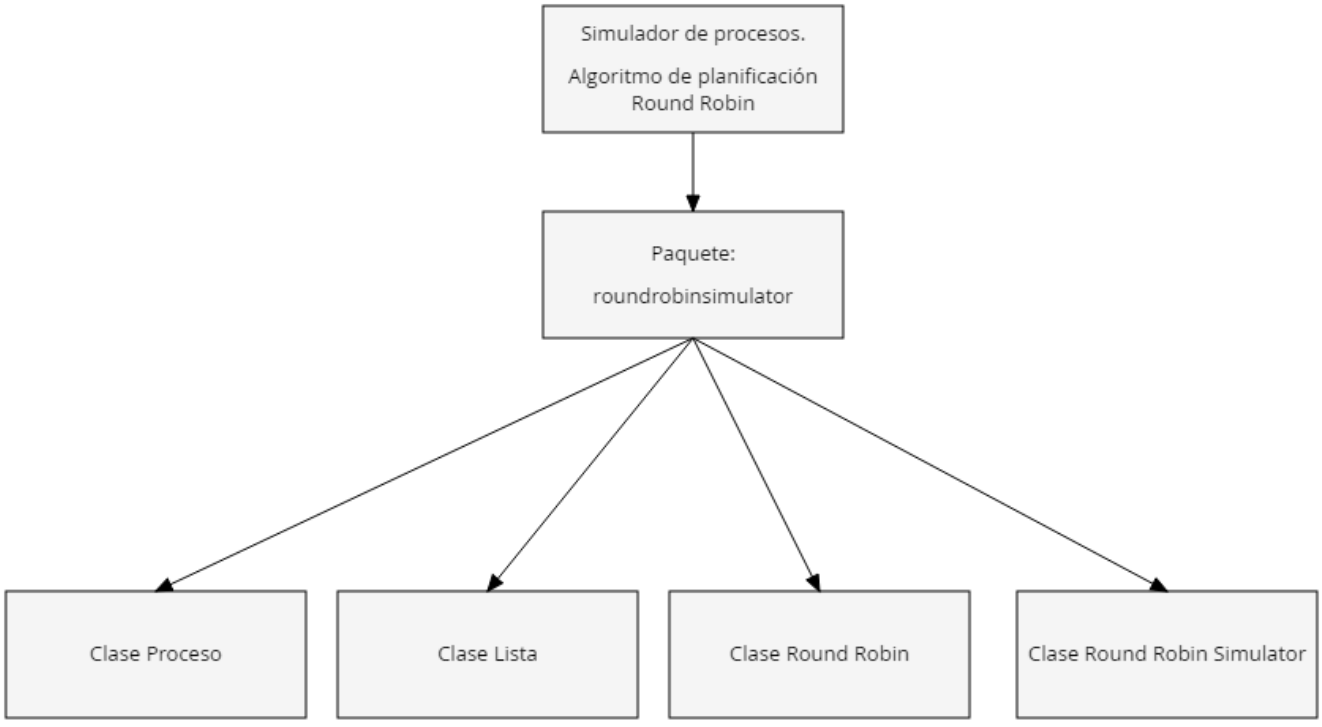
- Se utilizó el IDE NetBeans en su versión 8.2 sobre el sistema operativo Windows 10.
- La versión del Kit de Desarrollo Java fue la 1.8 (JDK 1.8.0_112-b15).
- Se utilizó un modelo de desarrollo en cascada ya que el proyecto se dividió en pequeñas fases desde un comienzo, desde la obtención de requerimientos con ligeros cambios en cuanto al algoritmo a utilizar, el diseño del algoritmo, la programación, pruebas unitarias e integrales hasta la finalización y liberación del proyecto.

Procedimientos de prueba

- La implementación y pruebas realizadas del software se hicieron sobre Windows 10 64-bit en una máquina portátil con un procesador Intel Pentium N3540 con 4GB de memoria RAM, 256GB SSD. El software no requiere grandes requerimientos de hardware para ejecutarse.
- Los parámetros de prueba se obtuvieron de los ejemplos de este algoritmo vistos en la clase de Sistemas Operativos con la Ing. Patricia del Valle Morales.

Arquitectura del sistema y funcionamiento de los módulos del sistema

El software está organizado de la siguiente manera, todo en un único paquete.



Viendo cada clase como un módulo, se detalla a continuación el funcionamiento en el sistema de cada uno de ellos.

Módulo *Proceso*

Visto como estructura de datos, este es el nodo en dónde se almacenará cada proceso y sus atributos. En sistema operativos se le denomina Bloque de Control de Proceso (BCP). Su función es proveer al simulador los procesos con los que trabajará, además de que internamente tendrá contadores y banderas requeridas para el cálculo de los tiempos de ejecución, respuesta y espera.

```
public class Proceso {
    private int id;
    private String nombre;
    private int tamanio_proceso;
    private int rafaga;
    private int t_llegada;
    private Proceso siguiente;
    private int rafaga_acumulada = 0;
    private int contador =0;
    private boolean responde = false;
    private int t_respuesta = 0;
    private int last_time_in_cpu = 0;
```

Cada atributo del objeto tiene sus respectivos *getters* y *setters* para realizar la lectura y escritura de ellos.

Este archivo se encuentra en \RoundRobinSimulator\src\roundrobinsimulator\Proceso.java

Módulo *Lista*

Es la cola que contendrá los procesos. Fue implementada como una lista ligada sencilla como se indica en el R3 y dicha cola se utilizará para simular las colas de procesos listos, en memoria principal y para CPU. Tiene métodos personalizados para agregar procesos (nodos), vaciarla, revisar si existen procesos en ella, la memoria principal disponible y restante al realizar movimientos de procesos, entre otros.

```
public class Lista{

    private Proceso inicio;
    private int tamanio;
    private int ram = 1024;
    private int rafaga_acumulada;
    private int contador_aux = 0;
```

Dicha lista queda fija a una memoria de 1024 unidades y posee contadores para calcular la ráfaga acumulada.

```
public void agregarProcesoDatos(int id, String nombre, int tamanio,
public void agregarProceso(Proceso proc){...18 lines }
public void removerProceso(){...4 lines }
public void removerProcesoRAM(){...5 lines }
public int getRam(){
    return this.ram;
}
```

Estas funciones se utilizan para llenar las listas según su origen. El método *agregarProcesosDatos()* añade los procesos según se crearon por el usuario, por lo tanto, recibe cadenas de texto y números como parte de su bloque de control de procesos.

El método *agregarProceso()* realiza la misma operación que el anterior, pero recibiendo el objeto proceso como tal. Este método se utiliza para los movimientos entre las distintas listas.

El método *removerProceso()* expulsa de la lista al objeto del tipo Proceso. *removerProcesoRam( )* realiza la misma operación que el método anterior sin antes extraer el tamaño del proceso y liberar dicha memoria RAM del simulador.

```
public void removerProceso(){
    inicio = inicio.getSiguiente();
    tamanio--;
}
public void removerProcesoRAM(){ //sacar el proceso de la ram y liberar memc
    ram = ram + inicio.getTamanio_proceso(); //ram liberada
    inicio = inicio.getSiguiente();
    tamanio--;
}
```

Este archivo se encuentra en \RoundRobinSimulator\src\roundrobinsimulator\Lista.java

Módulo Round Robin Simulator

Es el punto de partida del programa. Contiene al método *main* y es el bloque encargado de la generación de procesos, selección de opciones de simulación y llamadas al módulo de Round Robin dentro de un ciclo finito de iteraciones hasta que finalice la ejecución del algoritmo de planificación. Muestra también, en consola de texto, los movimientos entre las colas de procesos, tiempos actuales y cálculo final de los mismos, como se detalla en el R7 y R10.

```
public class RoundRobinSimulator {

    public static void main(String[] args) {

        int n;
        int quantum;
        String name;
        int size;
        int llegada;
        int rafaga;
        int time = 0;
        String opt;
        String opt2;
        String opt3 = null;

        Scanner s = new Scanner(System.in);
        Random id = new Random();

        Lista listos = new Lista();
        Lista listos_aux = new Lista();
        Lista RAM = new Lista();
        Lista CPU = new Lista();

        RoundRobin RR = new RoundRobin();
```

Las variables son para uso específico de los parámetros de configuración del simulador, mientras que también se crean cuatro instancias de la estructura *Lista* la cual ya está preparada para simular las colas de procesos listos, RAM, CPU y una cola auxiliar para usarse entre el movimiento de procesos nuevos y listos.

```
while(n!=0){
    System.out.print("\nNombre del proceso: ");
    name = s.nextLine();
    if("s".equals(opt2)){
        System.out.print("Tamaño del proceso en MB (máx 1024): ");
        size = s.nextInt();
    }
    else{
        size = 16;
    }
    if (size > RAM.getRam()){
        System.out.println("No puedes ejecutar procesos mayores a 1024 MB.");
        break;
    }
    System.out.print("Duracion de la rafaga (s): ");
    rafaga = s.nextInt();
    s.nextLine();
    if("n".equals(opt)){
        llegada = 0;
    }
    else{
        System.out.print("Tiempo de llegada (s): ");
        llegada = s.nextInt();
        s.nextLine();
    }
    listos_aux.agregarProcesoDatos(((id.nextInt(100000) % 1000)), name, size, rafaga, llegada, 0);
    n = n-1;
}
```

Dentro de un ciclo se irán adquiriendo los diferentes atributos de los procesos (nombre, tiempo de llegada, duración de ráfaga y tamaño, por default de 16 unidades de memoria), y se irán agregando a la lista auxiliar conforme se agregan a la lista ligada. Además, se tiene la restricción de no ingresar procesos mayores a 1024 unidades de memoria.

```
while(!RR.terminoSimu(listos_aux, listos, RAM, CPU)){
    System.out.println("\n" +time + "s.");
    RR.auxAListos(listos_aux, listos, RAM, time);
    if(!listos.esVacia()){
         RR.listosARAM(listos, RAM);
    }
    if(!RR.cpuVacio(RAM) || !RR.cpuVacio(CPU)){
         if(RR.cpuVacio(CPU)){
             RR.ramACPU(RAM, CPU, time);
             RR.ejecutandoEnCPU(CPU, listos, quantum,RAM, time);
        }
         else{
            int j = RR.ejecutandoEnCPU(CPU, listos, quantum, RAM, time);
            if(j==1 && !RAM.esVacia()){
                RR.ramACPU(RAM, CPU, time);
                RR.ejecutandoEnCPU_noshow(CPU, listos, quantum,RAM, time);
            }
        }
    }
        System.out.print("\n  LISTOS->");
        RR.verListaProcesos(listos);
        System.out.print("\n  RAM->");
        RR.verListaProcesos(RAM);
        System.out.println("\n  " + RAM.getRam() + "MB libres.");
        time = time + 1;

        if("s".equals(opt3)){
            Thread.sleep(1000);
        }
    }
```

Este ciclo se encarga de ejecutar el algoritmo mientras existan procesos en las cuatro listas a lo largo de la simulación.

De manera resumida, primero pasa los procesos en la cola auxiliar listos a la cola de listos conforme su tiempo de llegada. Revisa si la lista de procesos *listos* tiene elementos y los carga a memoria RAM si existe memoria disponible, sino se quedan en la cola de listos. Posteriormente revisa si la CPU está vacía y hay procesos en RAM, cargará inmediatamente el proceso a la cabeza de la cola de RAM y si la CPU está vacía, lo planifica y se ejecuta. Es en dónde comienza el conteo de ráfaga acumulada y de tiempo de respuesta inicial de dicho proceso. Existen distintas condiciones para mantener una revisión continua de los procesos en todas las colas.

```
System.out.print("\n  LISTOS->");
RR.verListaProcesos(listos);
System.out.print("\n  RAM->");
RR.verListaProcesos(RAM);
System.out.println("\n  " + RAM.getRam() + "MB libres.");
time = time + 1;

if("s".equals(opt3)){
    Thread.sleep(1000);
}
```

Al final de cada segundo de simulación se mostrará un resumen de cómo se encuentran las colas en dicho instante, la memoria libre y además, si se elige simular en tiempo real (cada segundo), se duerme el hilo principal ese tiempo para ver de una manera más clara la simulación.

```
System.out.println("\nTiempo promedio de espera = " + ((double)RR.getTiempo_final_espera()/num_proc) + "s.");
System.out.println("Tiempo promedio de ejecución = " + ((double)RR.getTiempo_final_ejecución()/num_proc) + "s.");
System.out.println("Tiempo promedio de respuesta = " + ((double)RR.getTiempo_final_respuesta()/num_proc) + "s.");
```

Al final de la ejecución, mediante método específicos de la clase *RoundRobinSimulator* se realiza el cálculo final de los tiempos promedio de espera, de ejecución y de respuesta. Dichos métodos se explicarán en el siguiente apartado de la documentación.

Este archivo se encuentra en
\RoundRobinSimulator\src\roundrobinsimulator\RoundRobinSimulator.java

Módulo Round Robin

Es quien contiene la lógica del algoritmo implementado y hace uso del módulo de Proceso y Lista para su funcionamiento. Se encarga también del cálculo principal de los tiempos requeridos y contiene métodos muy específicos para el manejo del algoritmo a detalle para facilitar el entendimiento de este.

```
public class RoundRobin {
    public int tiempo_final_ejecución;
    public int tiempo_final_respuesta;
    public int tiempo_final_espera;

    public void RoundRobin(int i){
        this.tiempo_final_ejecución = 0;
        this.tiempo_final_respuesta = 0;
    }

    public void setTiempo_final_respuesta(int i){
        this.tiempo_final_respuesta = i;
    }
```

```
void auxAListos(Lista listos_aux, Lista listos, Lista RAM, int time) {

    if(!listos_aux.esVacia()){
        Proceso copia = listos_aux.getInicio();
        while(copia!=null){
            if(copia.getT_llegada()==time){
                listos.agregarProceso(copia);
                listos_aux.setContador_aux(listos_aux.getContador_aux() + 1);
                System.out.print(" *" + copia.getNombre() + " llegó a los " + time + "s... LISTOS->");
                this.verListaProcesos(listos);
            }
            copia = copia.getSiguiente();
        }
    }
    else{}
    if(listos_aux.getContador_aux() == listos_aux.getTamanio()){
        this.resetCola(listos_aux);
    }
```

El método *auxAListos()* recibe como parámetro la lista auxiliar de procesos, la lista de procesos "listos", la lista de RAM y el tiempo actual de simulación.

La función de este método es subir a la lista de procesos *listos* los procesos ingresados por el usuario según su tiempo de llegada, comparándolo con el tiempo actual de simulación. Si eso se cumple, cargará el proceso. También mostrará en pantalla el proceso que llegó a *listos*.

```
public int listosARAM(Lista listos, Lista RAM){
    if(RAM.esVacia()){
        listos.getInicio();
        String proc_listo = listos.getInicio().getNombre();

        if(listos.getInicio().getTamanio_proceso()<=RAM.getRam()){
            RAM.agregarProceso(listos.getInicio());
            System.out.print("\n *" + listos.getInicio().getNombre() + " (" + listos.getInicio().getTa
            this.verListaProcesos(RAM);

            RAM.setRam(RAM.getRam() - listos.getInicio().getTamanio_proceso());
            System.out.print(", " +RAM.getRam() + " MB restantes.");

            listos.removerProceso();
            System.out.print("\n *" + proc_listo + " sale de LISTOS->");
            this.verListaProcesos(listos);
        }
        else{
            System.out.print("\n *No hay RAM suficiente para subir a " + listos.getInicio().getNombre(
            return 1;
        }
    }
```

El método *listosARAM()* realiza la carga de los procesos en la cola de *listos*. El método tiene dos bloques de instrucciones según el estado de la RAM (vacía o con procesos). Encola el proceso según si la lista está vacía o con elementos, muestra el tamaño de este, indica las unidades restantes de memoria y saca al proceso de la cola de listos. Si no hay memoria disponible despliega un mensaje de error.

```
public void verListaProcesos(Lista lista){
    if(!lista.esVacia()){
        Proceso copia = lista.getInicio();
        System.out.print("[");
        while(copia != null){
            System.out.print(copia.getNombre() + " ");
            copia = copia.getSiguiente();
        }
        System.out.print("\b]");
    }
    else{
        System.out.print("[ ]");
    }
}

public void verProcesoCPU(Lista lista){
    if(!lista.esVacia()){
        Proceso copia = lista.getInicio();
        System.out.print("[");
        System.out.print(copia.getNombre() + "]... (" + copia.getRafaga_acumu
       }
    else{
    System.out.print("[ ]");
    }
```

Ambos métodos únicamente imprimen la lista ligada. En el método *verProcesoCPU()* muestra además la ráfaga acumulada del proceso y si ya cumplió el quantum requerido antes de salir de la CPU.

```
public void ramACPU(Lista ram, Lista cpu, int time){
    Proceso copia = ram.getInicio();
    int nom_ultimo_ram = copia.getTamanio_proceso();

    System.out.print("\n *" + copia.getNombre() + " sale de RAM->");
    ram.removerProcesoRAM();
    this.verListaProcesos(ram);
    System.out.print(" y libera " + nom_ultimo_ram + "MB, " + ram.getRam() + " MB restantes.\n");
    cpu.agregarProceso(copia);
    System.out.print(" *" + cpu.getInicio().getNombre() + " sube a CPU->");
    this.verProcesoCPU(cpu);
}
```

ramACPU() es el planificador a corto plazo. Sube el proceso en la cola de CPU, elimina de la RAM, muestra cuánta memoria liberada hubo y qué proceso subió a CPU (muestra la cola CPU).

```
public int ejecutandoEnCPU(Lista CPU, Lista listos, int quantum, Lista RAM, int time){
if(CPU.getInicio().getContador() == 0){
    CPU.getInicio().setlast_time_in_cpu(time);
}
else{}

if(CPU.getInicio().getResponde()==false){
    CPU.getInicio().setResponde(true);
    CPU.getInicio().setT_respuesta(time);
    this.tiempo_final_respuesta = this.tiempo_final_respuesta + (CPU.getInicio().getT_respuesta() - CPU.getInicio().getT_llegada());
}
else{}
    if(CPU.getInicio().getRafaga_acumulada() == CPU.getInicio().getRafaga()){
            int sum = (time - CPU.getInicio().getlast_time_in_cpu());
            this.setTiempo_final_espera(this.getTiempo_final_espera()+ (((time - sum) - (CPU.getInicio().getRafaga() - sum)) - CPU.g
            System.out.print(" *CPU->");
            this.verProcesoCPU(CPU);
            System.out.print(" *¡Terminó su ejecución " + CPU.getInicio().getNombre() + "!, CPU ->[ ]");
            this.setTiempo_final_ejecución((time - CPU.getInicio().getT_llegada()) + this.getTiempo_final_ejecución());
            CPU.removerProceso();
            return 1;
```

Este método es sumamente importante ya que es el encargado de realizar todos los cálculos de los tiempos de ejecución, espera, tiempo final, tiempo de respuesta, incremento de la ráfaga vs el quantum del algoritmo.

La primera vez que sube un proceso a la CPU, modificará una bandera en el proceso la cual almacenará el tiempo en que subió a dicha CPU por primera vez. Esto con el fin de calcular el tiempo de espera y de respuesta.

Cuando termina su ráfaga total, al tiempo actual se le resta la última vez que subió a la CPU. Luego se calcula el tiempo final de espera mediante el tiempo actual menos el tiempo en que subió la última vez a la CPU. A lo anterior, a la ráfaga se le resta el tiempo de espera menos el tiempo de llegada. Finalmente expulsa el proceso de la CPU e indica un mensaje de finalizado.

```
else{
    System.out.print("\n  CPU->");
    this.verProcesoCPU(CPU);

    if(CPU.getInicio().getContador() == quantum){
        System.out.println(" *Finalizó su quantum de ejecución " + CPU.getInicio().getNombre() + " y sale
        CPU.setContador_aux(0);
        this.cpuAlista(CPU, listos);
        this.listosARAM(listos, RAM);
        return 1;
    }
    else{
    CPU.getInicio().setRafaga_acumulada(CPU.getInicio().getRafaga_acumulada() + 1);
    CPU.getInicio().setContador(CPU.getInicio().getContador() + 1);
    }
}
return 0;
```

En cambio, si aún no finaliza su ráfaga total pero sí su ráfaga actual según el quantum, se devuelve el proceso a la lista de *listos* para volver a ser planificado. Se suma la ráfaga acumulada al proceso y se reinicia su contador del quantum total a ejecutar.

```
public void cpuAlista(Lista cpu,Lista listos){
    System.out.print(" *" + cpu.getInicio().getNombre() + " se forma en LISTOS->" );
    listos.agregarProceso(cpu.getInicio());
    this.verListaProcesos(listos);
    cpu.quitar();
}
```

Al finalizar su quantum el proceso, sale de CPU y con este método se vuelve a formar en *listos*. Expulsa de la CPU dicho proceso.

```
public boolean terminoSimu(Lista aux, Lista listos, Lista RAM, Lista CPU){
    Proceso a = aux.getInicio();
    Proceso b = listos.getInicio();
    Proceso c = RAM.getInicio();
    Proceso d = CPU.getInicio();
    return a == null && b == null && c == null && d==null;
}
```

Finalmente, este método es el encargado de finalizar la simulación cuando todas las colas, incluyendo la auxiliar, queden vacías en su totalidad. En este momento, en la clase principal (RoundRobinSimulator) se procede a obtener los promedios de los tiempos de espera, de respuesta y de ejecución.

Este archivo se encuentra en \RoundRobinSimulator\src\roundrobinsimulator\Round Robin.java

Algunas pruebas del simulador de procesos con el Algoritmo de Round Robin

Se introducen los siguientes datos para la simulación:

(Requerimientos R1, R2 y R8)

```
SIMULADOR DE PROCESOS, ALGORITMO ROUND ROBIN
¿Cantidad de procesos a simular?: 3
Tamaño del quantum del CPU: 4
¿Con tiempos de llegada? s/n: s
¿Con tamaño personalizado en MB del proceso? s/n: n
¿La simulación será en tiempo real (segundos)? s/n: n

Nombre del proceso: P1
Duracion de la rafaga (s): 16
Tiempo de llegada (s): 8

Nombre del proceso: P2
Duracion de la rafaga (s): 6
Tiempo de llegada (s): 2

Nombre del proceso: P3
Duracion de la rafaga (s): 8
Tiempo de llegada (s): 1

*ROUND ROBIN, Q = 4
1024MB de RAM disponible.
Procesos ingresados por el usuario a simular: [P1 P2 P3]
```

Se muestra, previamente a la ejecución el tamaño del quantum, la memoria RAM disponible y el nombre de los procesos a simular.

```
0s.

  LISTOS->[ ]
  RAM->[ ]
  1024MB libres.
```

Comienzo de la simulación en 0s. Como ningún proceso llega en ese tiempo, todas las colas se encuentran vacías y no hay nada en la CPU (se encuentra ociosa).

```
1s.
 *P3 llegó a los 1s... LISTOS->[P3]
 *P3 (16MB) subió a la RAM->[P3], 1008 MB restantes.
 *P3 sale de LISTOS->[ ]
 *P3 sale de RAM->[ ] y libera 16MB, 1024 MB restantes.
 *P3 sube a CPU->[P3]... (0/8)s | Q = 0s.

  CPU->[P3]... (0/8)s | Q = 0s.
```

Nombre del proceso, (ráfaga acumulada / ráfaga total) | Quantium actual de ejecución (s).

```
  LISTOS->[ ]
  RAM->[ ]
  1024MB libres.
```

Llega el primer proceso P3 en el segundo 1. Se muestra el movimiento completo entre las listas ligadas (colas) paso a paso conforme funciona el algoritmo.

Al llegar, se forma a *listos*, se carga a la RAM, resta memoria al sistema, sale de cola *listos*. Luego, sale de la memoria RAM, libera memoria, se carga a la CPU y se muestra un resumen de su ráfaga total acumulada, ráfaga total del proceso y el quantum actual.

Finalmente se muestran a pantalla las tres colas, en la cual, al ser el primer proceso en subir, sólo hay un elemento en la CPU.

Se cumple con el requerimiento R7 de mostrar la impresión de las colas en cada movimiento posible en las colas generado por el algoritmo.

```
2s.
 *P2 llegó a los 2s... LISTOS->[P2]
 *P2 (16MB) subió a la RAM->[P2], 1008 MB restantes.
 *P2 sale de LISTOS->[ ]
  CPU->[P3]... (1/8)s | Q = 1s.

  LISTOS->[ ]
  RAM->[P2]
  1008MB libres.
```

A los dos segundos, llega el siguiente proceso y realiza la misma operación que el anterior. Sin embargo, el proceso P2 no puede expropiar a P3 ya que él tiene un tiempo en CPU asignado (Quantum = 4) y sólo lleva un segundo en la CPU. Se queda en RAM y nótese cómo disminuye la memoria (los procesos en CPU no consumen memoria en el sistema).

```
5s.

 CPU->[P3]... (4/8)s | Q = 4s.
*Finalizó su quantum de ejecución P3 y sale de la CPU ->[ ]
*P3 se forma en LISTOS->[P3]
*P3 (16MB) subió a la RAM->[P2 P3], 992 MB restantes.
*P3 sale de LISTOS->[ ]
*P2 sale de RAM->[P3] y libera 16MB, 1008 MB restantes.
*P2 sube a CPU->[P2]... (0/6)s | Q = 0s.

 LISTOS->[ ]
 RAM->[P3]
 1008MB libres.
```

En el segundo 5, el proceso P3 finalizó su ráfaga asignada. Esto es que ya igualó al valor del quantum por lo que debe salir y formarse a la cola de *listos* y volverse a cargar en RAM mientras haya unidades de memoria disponible. Inmediatamente sube P2 (liberando memoria) a la CPU que se encontraba en espera de ser planificado y comienza a acumular ráfaga en la CPU.

```
8s.
 *P1 llegó a los 8s... LISTOS->[P1]
 *P1 (16MB) subió a la RAM->[P3 P1], 992 MB restantes.
 *P1 sale de LISTOS->[ ]
  CPU->[P2]... (3/6)s | Q = 3s.

  LISTOS->[ ]
  RAM->[P3 P1]
  992MB libres.
```

Llega el último proceso P1 en el segundo 8. Se forma en la cola de listos y sube a RAM. En ella está también P3 esperando por la CPU mientras que P2 continúa ejecutándose. Cuando Q = 4, en el próximo segundo, P2 saldrá y el siguiente en cargarse será el proceso P3.

```
13s.
 *CPU->[P3]... (8/8)s | Q = 4s.
 *¡Terminó su ejecución P3!, CPU ->[ ]
 *P1 sale de RAM->[P2] y libera 16MB, 1008 MB restantes.
 *P1 sube a CPU->[P1]... (0/16)s | Q = 0s.

  LISTOS->[ ]
  RAM->[P2]
  1008MB libres.
```

Ya en el segundo 13, el proceso P3 habrá finalizado su ejecución y además cumplió su quantum de ejecución a la vez. El proceso simplemente se expulsa de la CPU, se carga el proceso siguiente que es P1 (mostrando la cola de RAM con P2 que estaba formado detrás de P1) y comienza a ejecutarlo en CPU.

Resultados de la simulación

```
30s.

  CPU->[P1]... (15/16)s | Q = 3s.

  LISTOS->[ ]
  RAM->[ ]
  1024MB libres.

31s.
 *CPU->[P1]... (16/16)s | Q = 4s.
 *¡Terminó su ejecución P1!, CPU ->[ ]
  LISTOS->[ ]
  RAM->[ ]
  1024MB libres.

Tiempo promedio de espera = 7.333333333333333s.
Tiempo promedio de ejecución = 17.333333333333332s.
Tiempo promedio de respuesta = 2.6666666666666665s.
```

En el segundo 31 finaliza la ejecución. Las colas quedan vacías y se realiza el cálculo final de los tiempos de espera, ejecución y respuesta requeridos en R10. La simulación se efectuó de manera correcta.

Conclusiones del proyecto

Ha sido un arduo trabajo de semanas para materializar este software simulador. Gracias a la clase de Sistemas Operativos, tema "planificadores de procesos" se adquirió un excelente dominio sobre el algoritmo de Round Robin. La implementación del algoritmo utilizando Java conllevó un gran esfuerzo sobre investigar nuevamente sobre clases y métodos en Java para poder recordar sobre esas herramientas que hicieron posible la creación de este proyecto.

El realizar intercambios entre colas y revisarlas a cada momento fue parte crítica del programa puesto que a veces se perdían procesos, había malos cálculos al final, no se cargaban correctamente, entre otros. Debugueando con NetBeans 8.2 se realizó, línea por línea con *breakpoints* la revisión de las variables de los objetos para detectar esos errores que hacían fallar al algoritmo.

Finalmente se compararon los ejercicios hechos en clase con los simulados y en todos ellos, este software, finalizó y calculó sin ningún error los procesos.

Se espera que este software sea utilizado a futuro para los futuros estudiantes de Ingeniería en Computación para entender, de manera gráfica la interacción y el funcionamiento paso a paso de este algoritmo basado en rondas de tiempo. Sin duda alguna fue un proyecto de nivel y que resultó en un programa de excelente calidad y con motivos educacionales.