SISTEMES OPERATIUS

Pràctica 3

Breu introducció

En aquesta pràctica anem a tractar la gestió básica de procesos del sistema operatiu. En el nostre cas, gastarem una màquina Linux i el llenguatge de programació C.

Jordi Sellés Enríquez

Universitat d'Alacant – Curs 2020/2021

Índex

1. Introducció

En aquesta pràctica es pretén realitzar una simulació de gestió de memòria amb particions dinàmiques. S'han implementar els algoritmes de pitjor lloc i millor lloc. També es podrà programar en qualsevol llenguatge de programació, a gust personal.

A classe de teoria hem vist el funcionament dels dos algoritmes. El de millor lloc buscarà el lloc més xicotet de la memòria on es puga clavar, d'altra banda, el de pitjor lloc, buscarà el lloc més gran on es puga clavar el procés. Ambdós algoritmes s'han de poder veure per consola i escriure's en un arxiu i en un sistema gràfic.

Solament plantejant el problema ens estem trobant amb uns objectes principals:

- Process: Tal i com el seu nom indica, és un procés i implementa tots els mètodes necessaris per a treballar amb ell.
- Processor: Aquest seria el processador, però simplement arreplegaria els mètodes que el facen funcionar. L'algoritme l'implementarien altres processadors heretats d'aquest.
- MemoryPractice: Aquest seria l'objecte que ho faria funcionar tot.
- IViewer: Aquesta seria una interfície que declararia els mètodes necessaris en els visualitzadors.

En aquest cas, he preferit treballat en Java perquè és el llenguatge de programació que estem treballant en *Programació 3*, de manera que també el practique i estudie. A més, Java em pareixia més ràpid i pràctic per a portar a terme aquesta pràctica.

2. Revisant els objectes

2.1. Process

L'objecte *Process*, com el seu nom indica, és un procés. Internament arreplegarà les dades necessàries per a gestionar-lo. A la memòria vaig a incloure els atributs que conté i els mètodes que considere més importants.

Atributs

Els atributs que conté aquesta classe són:

- processName: és un String que emmagatzema el nom del procés.
- arrivalTime: enter on s'emmagatzema el temps d'arribada del procés.
- executionTime: enter on s'emmagatzema el temps que ha d'estar en execució.
- internalCounter: enter on s'emmagatzema el temps que porta en execució.
- neededMemory: enter on s'emmagatzema la quantitat de memòria que es necessita.
- initialPos: enter on s'emmagatzema la posició de la memòria on comença el procés.
- inExecution: booleà on s'estableix si el procés està, o no, en execució.
- xAxisNameStringSeparator, xAxisStringSeparator, endl: son atributs privats d'ajuda al mètode *toString()*.

Mètodes

Process(String processName, int arrivalTime, int executionTime, int neededMemory)

Aquest és el constructor que initzialitza un nou procés. Com podem veure, reb un String i tres enters. Es controla que els enters siguen majors o iguals que 0, menys *neededMemory*, que ha de ser estrictament major que zero. En cas de que no es complisca algun d'aquests requisits, es llançarà una *NumberFormatException*. Si el String és null, es llançarà una *NullPointerException*.

El comptador intern s'initzialitza a 0, el booleà s'initzialitza en *false* i, la posició inicial, en -1.

Process(Process p)

Aquest és un simple constructor de còpia.

Copy()

És un mètode que torna una còpia de *this*.

ChangeToExecution()

És un mètode privat que exclusivament canvia l'atribut *inExecution* a *true*.

QuitFromExecution()

Mètode públic que estableix la posició inicial com -1 i canvia a *false* l'atribut *inExecution*.

Info()

Mètode públic que torna informació detallada del procés en un String.

ToString()

Mètode públic que torna informació curta del procés en un String, amb el següent format:

> [InitPos NAME FinalPos].

2.2. Processor

L'objecte *Processor* simula un processador. Per tal que siga més visual, a l'hora de treballar el mètode *toString()*, mostra els processos que es troben en execució, els que estan a la cola i els que han acabat.

Considere que en aquest objecte no cal explicar el funcionament dels atributs, ja que son molt obvis. Però, sí cal una distinció:

L'atribut *execProcesses* és un *ArrayList* de processos. Conté un punter als objectes que hi ha en execució. D'altra banda, l'atribut *execHashList* és un vector d'enters que simula la memòria; a més, s'emmagatzema en la posició concreta el codi *hash* del procés, mentre que les posicions buides, contenen un -1.

Mètodes

A continuació, una breu descripció dels mètodes que considere més importants:

Processor(int totalMemory)

Aquest és el constructor. Rep la memòria total del processador que anem a simular. A més, s'està filtrant que siga major a 1024.

A més a més, s'inicialitzen els atributs i es buida la memòria (*execHashList*).

moveProcessFromQueueToExec(Process p)

Aquest mètode mou un procés, rebut com a paràmetre, de la cua a execució. En cas de que el procés *p* ja estiga en execució o que no estiga a la cua, llançarà la excepció *ProcessAddingException*.

moveProcessFromExecToQueue(Process p)

Aquest mètode fa la funció inversa al mètode anterior.

moveProcessFromExecToKilled(Process p)

Aquest mètode mata un procés que estiga en execució i el fica al llistat dels processos que hagen acabat.

addProcessToQueue(Process p)

Aquest mètode fica el procés *p* a la cua. Es llançarà una *ProcessAddingException* en cas de que la memòria que demana el procés siga superior a la memòria que té el processador o, menor que 100.

lookForEmpySpaces()

Aquest mètode torna un Map<Integer, Integer> amb el començament d'un espai buit (identificador del Map) i el seu tamany (value).

checkWhereProcessCanBeAdded(Process p)

Aquest és un mètode que buscarà els llocs on es pot ficar un procés. En aquest objecte, és un mètode abstracte, per tant, no està implementat. Per a veure la implementació, haurem de veure les herències d'aquest objecte, ja que serà on aquest mètode jugue el seu paper, implementant els algorismes que se'ns demanen.

2.3. ProcessorBest

Aquest serà el tipus de processador que implementarà l'algorisme de millor lloc. Per tant, aquesta classe hereta de *Processor*.

Sols té dos mètodes:

Mètodes

ProcessorBest(int totalMemory)

Constructor per paràmetres. Li envia el paràmetre *totalMemory* al constructor de l'objecte superior.

checkWhereProcessCanBeAdded(Process p)

Aquest mètode torna la posició on pot afegir-se el procés *p* implementant l'algorisme de millor lloc de la següent manera:

```java
/**
 * Check where process can be added.
 *
 * @param p the process
 * @return the int
 */
@Override
protected int checkWhereProcessCanBeAdded(Process p) {
    Objects.requireNonNull(p);
    if(this.askIfMemoryIsFull()) return -1;

    Map<Integer, Integer> emptySpaces = this.lookForEmptySpaces();

    int lastSize = this.totalMemory + 1;
    int returnKey = -1;
    Set<Integer> setKeys = emptySpaces.keySet();

    for(Integer it : setKeys) {
        if(emptySpaces.containsKey(it)) {
            int actualSize = emptySpaces.get(it);
            if(actualSize >= p.getNeededMemory()) {
                if(lastSize > actualSize) {
                    lastSize = actualSize;
                    returnKey = it.intValue();
                }
            }
        }
    }
    return returnKey;
}
```

El mètode buscarà sempre el lloc més xicotet que haja. Per això, per cada *key* del *Map*, es comprova el tamany, comparant-lo amb la variable local *lastSize*, que s'inicialitza amb el valor de la memòria total més 1. D'aquesta manera, sempre s'estarà guardant el valor més xicotet.

2.4. ProcessorWorst

Este objecte implementarà l'algorisme del pitjor lloc. El constructor és igual al del *ProcessorBest*.

Mètodes

Com el constructor és igual al de l'altre processador, vaig simplement a explicar el mètode que implementa l'algorisme:

checkWhereProcessCanBeAdded(Process p)

Aquest mètode és una sobrecàrrega del mètode de *Process*. S'implementa l'algorisme del pitjor lloc.

```java
/**
 * Check where process can be added.
 *
 * @param p the process
 * @return the int
 */
@Override
protected int checkWhereProcessCanBeAdded(Process p) {
    Objects.requireNonNull(p);
    if(this.askIfMemoryIsFull()) return -1;

    Map<Integer, Integer> emptySpaces = this.lookForEmptySpaces();

    int lastSize = 0;
    int returnKey = -1;
    Set<Integer> setKeys = emptySpaces.keySet();

    for(Integer it : setKeys) {
        if(emptySpaces.containsKey(it)) {
            int actualSize = emptySpaces.get(it);
            if(actualSize >= p.getNeededMemory()) {
                if(lastSize < actualSize) {
                    lastSize = actualSize;
                    returnKey = it.intValue();
                }
            }
        }
    }
    return returnKey;
}
```

En aquest cas, el funcionament és el mateix que a l'altre processador. El canvi el trobem a *lastSize*, que està inicialitzat a 0. D'altra banda, la condició és que *lastSize* ha de ser menor que *actualSize*.

2.5. MemoryPractice

Aquest serà l'objecte que treballe amb tot el sistema i treballa com un gestor de memòria. Una vegada que l'objecte es construeix, quidrarem al mètode *MemoryPractice.run()*. Aquest mètode és el que carregarà els processos a la cua del processador i els executarà. Si no els pot moure, capturarà l'excepció i deixarà passar la iteració i ho tornarà a intentar a la següent.

En qualsevol cas, el processador incrementarà el comptador intern de cada procés que estiga en execució en cada iteració. Finalment, si el comptador intern de un procés és igual al seu atribut *executionTime*, el procés serà eliminat de la memòria.

Cal afegir que els processos entren a la cua a través de l'objecte *ProcessorLoaderFile*, que implementa la interfície *IProcessorLoader*.

Mètodes

MemoryPractice(Processor processor, String filePath)

Aquest és el constructor. Rep un processador i un String amb la ruta de l'arxiu que anem a gastar per a que carregue els processos a la cua.

Start()

Mètode privat que carrega el *ProcessorLoaderFile*. En cas de no existir l'arxiu, llançarà una *MemoryPracticeIOException*.

Run()

El mètode run s'encarrega de fer funcionar la pràctica:

```
public void run(IViewer viewer) throws MemoryPracticeIOException, MemoryPracticeRuntimeException {
    boolean play = true;
    this.start();
    viewer.show();
    while(play) {
        this.incrementCounter();

        // KILLING PROCESSES
        ArrayList<Process> finalized = this.processor.getCopyOfExecProcesses();
        for(Process it : finalized) {
            try {
                if(it.isFinalized()) {
                    try {
                        this.processor.moveProcessFromExecToKilled(it);
                    } catch (InvalidProcessNeededMemory | ProcessAddingException e) {
                        throw new MemoryPracticeRuntimeException(e.getMessage());
                    }
                }
            } catch (ProcessExecutionTimeExceeded e1) {
                try {
                    this.processor.moveProcessFromExecToKilled(it);
                } catch (InvalidProcessNeededMemory | ProcessAddingException e) {
                    throw new MemoryPracticeRuntimeException(e.getMessage());
                }
            }

        }

        // Adding from queue:
        if(!this.processor.getQueueAsArrayList().isEmpty()) {
            Process firstInQueue = this.getProcessor().getQueueAsArrayList().get(0);
            try {
                this.processor.moveProcessFromQueueToExec(firstInQueue);
            } catch (ProcessAddingException e) {
                throw new MemoryPracticeRuntimeException(e.getMessage());
            }
        }

        if(this.isFinalized()) play = false;
        this.processor.incrementProcessesCounter();
        viewer.show();
    } // WHILE END
    viewer.close();
}
```

Aquest mètode pot llançar una *MemoryPracticeIOException* o una *MemoryPracticeRuntimeException*.

2.6. ProcessorLoaderFile

Aquesta és una classe que implementa la interfície *IProcessorLoader* i la seua funció és carregar processos al processador a través d'un arxiu. Per a que puga treballar amb l'arxiu, les línies han de tindre un format correcte, sino es llançà una excepció. El format correcte és:

process [name] [executionTime] [neededMemory]

2.7. IViewer

Aquesta interfície serveix per a implementar la manera de visuaitzar el resultat. Conté dos mètodes: show i close.

Show s'encarregarà d'emplenar l'arxiu en el que ho mostra tot, i close, de tancar-lo.

2.7.1. ViewerConsole

Aquest objecte implementa la interfície i ho mostra tot en un arxiu de text. Un exemple d'execució seria el següent:

```
 98
 99 ===== MEMORY MANAGER =====
100 === DATA ===
101 Iterations: 7
102 File Path: tests/files/runMemoryPractice2.in
103 === IN EXECUTION ===
104 [0 A 500]
105 [500 B 1000]
106 [1500 D 1900]
107 [1900 F 2000]
108 === QUEUE ===
109 === FINALIZED ===
110 [-1 C 499]
111 [-1 E 99]
```

2.7.2. ViewerGIF

Aquest objecte seria altre que implementa la interfície i ho mostraria tot en un GIF. No l'he pogut acabar per falta de temps. Possiblement faria falta un altre objecte que es gaste com a intermediari.

2.8. Simulació amb tests unitaris baix Junit4

El programa no l'he executat baix un Main ja que em pareix un poc buit. L'he executat baix tests unitaris per a poder probar el codi per parts. Els dos tests unitaris que porten a terme la pràctica son els següents:

```
/*
 * Este test trabaja con el algoritmo de mejor hueco y 5 procesos. Funciona perfectamente.
 */
@Test
public void testRunMemoryPractice2BEST_FILE() throws InvalidProcessorTypeException, MemoryPracticeIOException, MemoryPracticeRuntimeException {
    final String outFile = DIRFILES + "runMemoryPractice2BEST.data";
    mP = new MemoryPractice(ProcessorFactory.createProcessor("ProcessorBest", totalMemory), DIRFILES + "runMemoryPractice2.in");
    IViewer iv = new ViewerConsole(mP);
    PrintStream ps = standardIO2File(outFile);
    if(ps != null) {
        mP.run(iv);
        assertTrue(mP.isFinalized());
        System.setOut(System.out);
        ps.close();
    } else {
        fail("Error: No se pudo crear el fichero " + outFile);
    }

    //Se compara salida con la solución
    StringBuilder sbSolution = readFromFile(DIRFILES + "runMemoryPractice2BEST.sol");
    StringBuilder sbObtenido = readFromFile(DIRFILES + "runMemoryPractice2BEST.data");
    compareLines(sbSolution.toString(), sbObtenido.toString());
}
```

```
/*
 * Este test trabaja con el algoritmo de peor hueco y 5 procesos. Funciona perfectamente.
 */
@Test
public void testRunMemoryPractice2WORST_FILE() throws InvalidProcessorTypeException, MemoryPracticeIOException, MemoryPracticeRuntimeException {
    final String outFile = DIRFILES + "runMemoryPractice2WORST.data";
    mP = new MemoryPractice(ProcessorFactory.createProcessor("ProcessorWorst", 2000), DIRFILES + "runMemoryPractice2.in");
    IViewer iv = new ViewerConsole(mP);
    PrintStream ps = standardIO2File(outFile);
    if(ps != null) {
        mP.run(iv);
        assertTrue(mP.isFinalized());
        System.setOut(System.out);
        ps.close();
    } else {
        fail("Error: No se pudo crear el fichero " + outFile);
    }

    //Se compara salida con la solución
    StringBuilder sbSolution = readFromFile(DIRFILES + "runMemoryPractice2WORST.sol");
    StringBuilder sbObtenido = readFromFile(DIRFILES + "runMemoryPractice2WORST.data");
    compareLines(sbSolution.toString(), sbObtenido.toString());
}
```

L'objecte *ProcessorFactory* és un objecte de factoria que serveix per a crear distints tipus de processadors mitjançant *reflection*.

3. Conclusió

Una pràctica molt entretinguda i interessant. Ha sigut llarga per voler implementar la POO. Fer una ampliació als altres algorismes seria molt senzill, sols hauriem de fer dues extensions de *Processor* i implementar els algorismes.

M'ho he passat molt bé dissenyant i implementant el codi, a més de cursant la materia.

Al codi també li quedaria un poc de refracció per fer. L'objecte processor podria tindre un disseny més SOLID si implementarem l'herencia d'interfície, però era més ràpid fer-ho així.

Per si es vol consultar i revisar el codi, adjunte un enllaç al projecte de GitHub.