COEN 346 Programming Assignment 3

Student: Sun He Li 26624847
William Cui 27314167

Due Date: April 7, 2019

High level descriptions

In this assignment, we are trying to simulate how a virtual memory is managed in an operating system with the concurrency controls.

In order to simulate the main memory and disk space, the size of Main memory (Page size) is given in a input file called" memconfig.txt", the disk space is called "vm.txt" and it's considered has a very large space.

We created 4 classes to simulate how concurrent processes are scheduled with access to the virtual memory and accessing the disk.

Command.java

This is a class to define the commands to be fetch into the process.

`public Command(String name, String ID, int Value)` is a constructor method for the class

`public void print()` is a method to print a command

VMM.java

This is a class to simulate the virtual memory management. It mainly contains 4 methods with 3 API services

```
public static void readSize()
```

To read the page size of the main memory from "memconfig.txt".

After reading the size, we initialized the main memory as empty.

```
public static void Store(String id, int value)
```

This is an API service used to store a variable into a virtual memory(either main memory or disk), here we used another method `public static  int checkFreeSpace()` which tells whether if there is a free spot in the main memory and return the free spot index when there is free spot. Getting to know this helps us to decide where to store the give variable. Store it into the memory at respective spot when there is free space or store it into a disk when there’s no space in memory. At the end of the service, we update the last access time.

```
public static void Release(String id)
```

This is another API service used to release a given variable from the storage (memory or disk). That goes to two directions: first, we check the variable id, see if it’s in the main memory, if true, we simply set the empty parameter to be true for the corresponding spot. (here the logic is simple, we created a parallel array stores the emptiness status of the memory, which will indicate whether the corresponding spot is free or not). Second when it’s not in main memory we check the disk to find the variable and remove it from disk, this requires access to the disk file ”vm.txt”. At the end of the service, we update the last access time.

```
public static int Lookup(String id)
```

This is an API service used to search if a variable is in the memory and return its value or -1 if it doesn’t exist. There are two possibilities when we find the variable, first is a hit when checking main memory, we simply return the value and update the last access time. Or when we find it in the disk, it goes two situations: if there’s a free space in main memory, we move this variable into the free spot and return the value, again, update the last access time. On the other hand, when memory is full, we swap the variable with the item which has the smallest last access time in the main memory and return the value Again, we have to update the last access time.

```
private static void writeVM(String tempString)
```

This is a method used to generate the “outpu.txt” which contains the running information of each process and instructions being fetching with the variable ids and values. It also shows information when there ‘s a memory management action of swapping.

Process.java

This Process class implements Runnable and is to simulate the running a process.

```
public Process(int a,int b,int r,String name)
```

default constructor for Process class.

`public int getAT(),getBT(),getFT(),getReadyForNextTime()`

Some getter methods to return the value of parameters

```
@Override
public void run()// this is the thread will be running
```

To simulation of running a process:

Once it's arrived and ready to run, it waits for schedule's clock signal and start it's running. The running procedure includes pick up next command, decoding the commands and fetching the command(calling respective API services).

`public void getNextCommand()`

this method is invoke by run() to pick up next command in the "command.txt" file

Scheduler.java

This is a the main class of the project, it reads the "processes.txt" and makes a schedule for the processes to run concurrently.

`public static void main(String[] args)`

This is considered as the main process of the program, any other processes is running concurrently with this process.

`private static void readProcesses()`

This is a method to read the process information from "processes.txt".

Work Flow

Run the program, we first call the `readProcesses(); method` to access the "processes.txt" file. After gathering all the information, we store them in an array of Process objects (`Process[] processes)` ,at the same time for each process, we generate a thread corresponding to it and store them in an array of processThread(`Thread[] processThreads`).

Then we call the `readSize();` from VMM class to know the size of the main memory for further use.

Next, we use a while loop to run all the threads, it contains:

1. A for loop which starts all the thread when the clock equals to the arrive time.
   `processThreads[i].start();`

this invokes the run() in the process which will make process to execute the current commands, depends on the decoding, the respective API services will be called and executed.

2. A for loop which schedules the processed to pick up next commands.
   `processes[i].getNextCommand();`
3. A for loop to join all threads to ensure they are running concurrently.
   `processThreads[i].join();`

At the end, after finishing all the processes, we write all the threads running information, instruction being fetched with the names and values into the "output.txt" file.

Conclusion and Discussion

Ensuring mutual exclusion is crucial in concurrency processing. In this assignment, we stored all the comments inside a static queue in the process class and the code does fetching and removing the first command at the same time. Putting them in one static queue ensured that all processes are looking at the same command list to be executed instead of one for each individual process which might overlap the commands to be executed and runs the same commands as the same time. And doing the fetching and removing at the same time made sure that if one command is being taken by a process, no other processes would be allowed to grab it and run it.