Operating Systems
Project 2: Scheduling Algorithms

Nardine Basta - 4-626
Christine Maher - 4-1083
Frederic-Gerald - 4-1805

General Classes Implemented:

Note: For more details, view code comments.

main.java: this is where the program will start, it contains a main method that uses command line arguments to execute an algorithm accordingly. Running the program without arguments at all will output information about how to use it and which algorithms are named what.

user.java: this class implements the user. Each user has a name, priority, and minimum and maximum ticket ranges (used in the lottery scheduling).

process.java: this class implements the process. Each process has a name, an arrival time, a run time and an owner (user).

These classes are the bare bones of the program. They are used by the algorithms to represent their respective descriptions. Other more algorithm specific class implementations are described in their algorithm's page.

Multi-level Feedback Scheduling:

Note: For more details, view code comments.

The multi-level feedback scheduling algorithm is basically a table of levels of priorities and cells of processes. Each level has a quantum of running time double it's previous, starting from quantum 1 at level 1. Any process that starts is scheduled at level 1, then when it is done with it's quantum, it is moved down to the next priority level and any other pending processes in the very first level are ran and acted upon accordingly. When a process is at the last level and is finished with it's quantum, it gets re-scheduled at that last level, it doesn't start at the beginning. Multi-level feedback scheduling has a starvation problem because if processes keep on entering at the first level, those at the last level will never run.

Classes implemented that are specific to multi-level feedback scheduling:

process_queue.java: this represents a “level” in the table of processes. Processes are taken from it's start and are scheduled to it's end. It was written to take some complications out of the scheduling algorithm itself.

mlf.java: represents the multi-level feedback algorithm. Basically, it will initialize the table for processing and an input list of processes to run then will start a timer and run the algorithm.

Description of the simulation:

At each “clock tick” or “timer incrementation”, it checks the input list of processes for those that have their arrival time equal to it's current time and will enqueue them into the queue at the first level. Processes are taken out of the levels (only if the levels aren't empty, but if they are, the scheduler moves to the next level) and ran for the level's respective quantum of time. At each running quantum, processes are being scheduled (those who have their arrival time equal to the current time). The algorithm keeps running the processes in the level until both the input list or processes and the table are empty (which means there are no more processes to run).

Lottery Scheduling:

Note: For more details, view code comments.

The lottery scheduling algorithm is based on statistical information. Each user acts as a lottery player and running a process is the prize. The users just get numbers that represent their tickets. The scheduler then picks up a number randomly, checks which user has the ticket with the number and runs one his processes for a certain quantum of time. This way, eliminates process and user starvation.

Classes implemented that are specific to lottery scheduling:

lot.java: Implements the actual algorithm. Note that min_ticket and max_ticket are used from user.java as the range of tickets the user has.

Description of the simulation:

There is a list of users, a list of running processes, a list of input processes (that aren't yet scheduled), a current time, a maximum number of tickets (depends on the number of users) and a user. When the scheduler starts, it schedules processes with arrive time that are equal to the current time for running from the input processes list into the running processes list (this is also done at every time the timer is incremented). The process gets users from the scheduled processes and puts them in the user list (to keep track of the lottery players) and assigns a range of numbers to each one of them respectively to their priority (the higher the priority, the more numbers a user gets). The scheduler then selects a random number (that is, of course, smaller than the largest number given to users) and looks for the first process in the running processes list that belongs to the user with the number, removes the process from the running list, runs it for a quantum of time and re-enqueues it into the end of the running list (this way, per-user-process starvation won't happen). The algorithm keeps running until there are no running nor waiting to run processes.

Fair-Share Scheduling:

Note: For more details, view code comments.

The fair chair ensure that the cpu time is fairly distributed among users
If a user has a big number of processes and another has half the number for example
The second user runs two processes each time the 1st one runs 1 process

the priority each user is taken care of
If a sure has a higher priority he tends to have more number of quantum for his processes
The processes of each user run in a round robin manner

Our implementation of the fair chair scheduling is one class

It is composed of 5 methods

The logic of the problem is as follows
We have 3 parallel linked lists
The first one is jobs which holds the processes
The second one is already ran which holds a string of true or false that signify is that this process has within the user's processes has ran before to guarantee that each turn a different process run to be fair within the users' processes.
The third linked list is current_owners which holds the names of the users that haven't ran yet any processes when a user runs a process the node with his name is removed to guarantee that he won't run any further process.
When the list has a size of 0 it is re initialized with the method get_current_owners.

The five methods are the following:
get_current_owners()
for initializing a list with the current users.

LinkedList already_ran()
This method is for initializing a linked list that goes in parallel with the linked list of the processes to make sure that each user's turn a differnt one of his processes runs. being false means that that process didn't run before.

check_processes()
Makes sure that the process who has the turn to run hasn't ran before which means that all the user's processes has ran before i.e. something like round robin. If the process with the current index is not allowed to run because it has already ran and the ore more processes within this user's processes that haven't ran yet. It returns the number of the process that is allowed to run.

Run()
This method is for running the processes the priority here is handled as follows if a user has a priority x the process having the turn will run for x quantum.
if the process has a runtime= 0 it is removed from the list.

scheduel()

This method has a nested loop. The bigger loop makes sure that the algorithm runs until there are no more jobs.
The inner loop makes sure that the whole list is traversed and the index is moved along the list to point to the jobs.
If the indexed process's arrival time is less than or equal to the clock it calls the method check process and then the run method with the index returned by the check processes method.
If not it reruns the previously ran process in order
If the process has a runtime= 0 it is removed from the list
It stops when the size of the processes list is 0

Limitations

1- the process list must be ordered according to the arrival time which makes sense in the physical applications because the lastly arrived processes are just attached to the end of the list
2- for guaranteeing the priority when a user has a priority x its process runs for x quantum . id the process finishes before taking all of the x quantum.the turn doesn't necessarily switch to a process of the same user which is not correct and doesn't realize the fairness. We worked on that but actually we ran out of time and couldn't complete it.