Assignment 2

Name: Fredric Eriksson Sepulveda
Email: fe222pa@student.lnu.se
Course: 1DV512
Term: spring

Contents

Task 1: Comparison of FreeBSD ULE and Linux CFS Schedulers

1. Does ULE support threads, or does it support processes only? How about CFS?

Both ULE and CFS support threads, because both threads are considered processes and both schedulers can run processes

2. How does CFS select the next task to be executed?

CFS schedules the thread with the lowest vruntime to run next.

3. What is a *cgroup* and how is it used by CFS? Does ULE support *cgroups*?

threads of the same application are grouped into a structure called a cgroup “CFS then applies its algorithm on cgroups, ensuring fairness between groups of threads. When a cgroup is chosen to be scheduled, its thread with the lowest vruntime is executed, ensuring fairness within a cgroup. Cgroups can also be nested.” ULE does not support cgroups because each thread its considered independent from other threads.

4. How many queues in total does ULE use? What is the purpose of each queue?

“ULE uses two run queues to schedule threads: one runqueue contains interactive threads, and the other contains batch threads. A third runqueue called idle is used when a core is idle. This runqueue only contains the idle task.”
ULE has 3 queues in total One queue contains interactive threads The other involves with batch threads (Batch processes usually execute without user interaction) The third one is used whenever the core is idle and only runs the idle task

5. How does ULE compute priority for various tasks?

Cited answer
“To classify threads, ULE first computes a score defined as interactivity penalty + niceness. A thread is considered interactive if its score is under a certain threshold, 30 by default as in FreeBSD11.1. With a niceness value of 0, this corresponds roughly to spending more than 60% of the time sleeping. Otherwise, it is classified as batch”
“When a thread is created, it inherits the runtime and sleeptime (and thus the interactivity) of its parent. When a thread dies, its runtime in the last 5 seconds is returned to its parent. This penalizes parents that spawn batch children when being interactive.”

Each thread will be computed and given a score = (interactive penalty + niceness) Thread is interactive when: score < 30

6. Do CFS and ULE support task preemption? Are there any limitations?

“In ULE, full preemption is disabled, while CFS preempts the running thread when the thread that has just been woken up has a vruntime that is much smaller than the vruntime of the currently executing thread (1ms difference in practice).”
“In CFS, ab is preempted 2 million times during the benchmark, while it never preempted with ULE”

One of the limitations for not using preemption might be the performance, sysbench performs worse on ULE than on CFS due to not using preemption but a weakness of using preemption might be the fact that it introduces complexity to the kernel code.

7. Did Bouron et al. discover large differences in per-core scheduling performance between CFS andULE? Which definition of "performance" did they use in their benchmark, and why?

One of the major differences between is how batch threads are handled CFS uses task preemption and hence higher performance than ULE We define “performance” as follows:
for database workloads and NAS applications, we compare the number of operations per second, and for the other applications we compare “1/execution time”.
The reason for using this definition is that performance in context of database workloads & NAS applications: operations per second is the most suitable because workloads look for the amount of work a machine can perform in a specific amount of time [1] and for other applications we are mostly interested in how effective it performs under execution time.

8. What is the difference between the multi-core load balancing strategies used by CFS and ULE? Many of them faster? Does any of them typically reach perfect load balancing?

CFS allocates work across all available cores and ULE balances the amount of threads relative to the amount of cores.
For ULE “it takes more that 450 load balancer invocations or about 240 seconds to reach a balanced state.
CFS balances the load much” “CFS balances the load much faster. 0.2 seconds after the unpinning, CFS has migrated more that 380 threads from core 0.
” CFS balances faster than ULE “CFS converges faster towards a balanced load, but ULE achieves a better load balance in the long run.”
“CFS never achieves perfect load balance.
CFS only balances the load between NUMA nodes when the imbalance between the two nodes is “big enough” (25% load difference in practice).

So cores in one node can have 18 threads while cores in another only have 15.” The load core of ULE is the amount of threads it can currently run due to this strategy ULE cannot reach perfect load balancing.

Task 2: Developing a Scheduling Algorithm in Java

2.1 Task

2.1.1 Arrival Chance explained

The method ArrivalChance does the following

```
public int ArrivalChance(double probArrival, List<ScheduledProcess> processors, int id, int maxBurstTime,
    int minBurstTime, int tick, int maxArrivalsPerTick, int addProcess, int numProcesses) {
  double arrivalChance = rng.nextDouble();
  int check = 0;
  if (processors.size() < numProcesses - 1) {
    while (true) {
      if (check == maxArrivalsPerTick) {
        break;
      } else if (arrivalChance <= probArrival) {
        addProcess += 1;
      } else {
        break;
      }
      arrivalChance = rng.nextDouble();
      check += 1;
    }
  } else if (processors.size() == 9) {
    if (arrivalChance <= probArrival) {
      addProcess += 1;
    }
  }

  if (processors.size() < 10) {
    if (addProcess > 0) {
      for (int i = 0; i < addProcess; i++) {
        processors
            .add(new ScheduledProcess(id, rng.nextInt((maxBurstTime + 1 - minBurstTime)) + minBurstTime, tick));
        id += 1;
      }
    }

  }
  return id;
}
```

Adds processes by running the percentages given by rng and to add 2 then it has to have the arrival chance to be under 0.76 twice in a row, and maxArrivalsPerTick define the maximum amount of possible processors to add per tick, and if is the last processor then it tried to add one only

Then those processes are added to the process Arraylist, which returns id because id does not update outside the method

2.1.2 Index input

```
if (firstIteration) {
  index = rng.nextInt(processors.size());
  firstIteration = false;
}
```

When it is the iteration where the process list contains something then the index is assigned automatically according to the size of the list

```
if (getNextIndex) {
  while (true) {
    getNextIndex = false;
    index = rng.nextInt(processors.size());
    ScheduledProcess isAvailable = processors.get(index);
    if (registry.size() == 10) {
      break;
    } else if (isAvailable.getBurstTime() != isAvailable.getProcessingTime()) {
      break;
    }
  }
}
```

To obtain the next indexes the scheduler waits until the current process is cleared and then it goes through this code, the code is meant to go forever and provide a new available process, and since there are two ways to do this, either by creating an ArrayList that adds and deletes processes accordingly or my method that just searches for an available process, and it ends whenever the current process is incomplete or the registry ArrayList is complete.

2.1.3 ScheduledProcess class explained

```
ScheduledProcess process = processors.get(index);
if (process.getProcessingTime() < process.getBurstTime()) {
  process.setProcessingTime(process.getProcessingTime() + 1);
}
```

Processes are called through the index value and if the condition is true, then a tick is added to the processing time

```
boolean getNextIndex = false;
for (ScheduledProcess processor : processors) {
  if (processor.getProcessingTime() == processor.getBurstTime()) {

    if (!registry.contains(processor)) {

      getNextIndex = true;
      processor.setTotalWaitingTime(tick + 1 - processor.getArrivalTime());
      registry.add(processor);
    }

  }
}
```

Then a check through all processors is made to find a processor that equals the same burst time and processing time, once it's found before it is added to the registry ArrayList the registry is checked if it is found inside the registry already, once it is done the total time is set and then it is added to the registry since the current process is done and we require a new index we trigger the boolean getNextIndex.

2.1.4 preemptive Code explained

```
if (isPreemptive) {
  tqc++;
  if (tqc == timeQuantum) {
    tqc = 0;
    tqb = true;
  }
}

if (tqb) {
  tqb = false;
  while (true) {
    index = rng.nextInt(processors.size());
    ScheduledProcess isAvailable = processors.get(index);
    if (registry.size() == 10) {
      break;
    } else if (isAvailable.getBurstTime() != isAvailable.getProcessingTime()) {
      break;
    }
  }
}
// I make sure that it can pick from an available process not one that is
```

We make sure that the simulation is preemptive from the start and then we add a counter to a variable that looks for timeQuantum

Once the counter reaches timeQuantum we trigger the if cases so the scheduler is assigned a new index value that uses the same principle as in 2.1.2.

2.1.5 Non-preemptive results

```
Running non-preemptive simulation #0
Simulation results:
---------------------
 id: 0| burstTime: 7| arrivalTime: 1| waitingTime: 0|
 id: 5| burstTime: 7| arrivalTime: 4| waitingTime: 4|
 id: 3| burstTime: 4| arrivalTime: 2| waitingTime: 13|
 id: 4| burstTime: 2| arrivalTime: 3| waitingTime: 16|
 id: 9| burstTime: 7| arrivalTime: 6| waitingTime: 15|
 id: 2| burstTime: 9| arrivalTime: 2| waitingTime: 26|
 id: 7| burstTime: 9| arrivalTime: 5| waitingTime: 32|
 id: 8| burstTime: 9| arrivalTime: 6| waitingTime: 40|
 id: 1| burstTime: 9| arrivalTime: 1| waitingTime: 54|
 id: 6| burstTime: 2| arrivalTime: 5| waitingTime: 59|
-------------------------------
Execution time: 66
-------------------------------
Average waiting time: 25
```

```
Running non-preemptive simulation #1
Simulation results:
---------------------
 id: 1| burstTime: 4| arrivalTime: 1| waitingTime: 0|
 id: 4| burstTime: 8| arrivalTime: 3| waitingTime: 2|
 id: 2| burstTime: 5| arrivalTime: 2| waitingTime: 11|
 id: 0| burstTime: 10| arrivalTime: 1| waitingTime: 17|
 id: 8| burstTime: 3| arrivalTime: 5| waitingTime: 23|
 id: 6| burstTime: 2| arrivalTime: 4| waitingTime: 27|
 id: 5| burstTime: 4| arrivalTime: 3| waitingTime: 30|
 id: 9| burstTime: 6| arrivalTime: 5| waitingTime: 32|
 id: 3| burstTime: 9| arrivalTime: 2| waitingTime: 41|
 id: 7| burstTime: 4| arrivalTime: 4| waitingTime: 48|
-------------------------------
Execution time: 56
-------------------------------
Average waiting time: 23
```

```
Running non-preemptive simulation #2
Simulation results:
---------------------
 id: 1| burstTime: 9| arrivalTime: 2| waitingTime: 0|
 id: 5| burstTime: 10| arrivalTime: 5| waitingTime: 6|
 id: 0| burstTime: 5| arrivalTime: 2| waitingTime: 19|
 id: 8| burstTime: 8| arrivalTime: 13| waitingTime: 13|
 id: 7| burstTime: 9| arrivalTime: 6| waitingTime: 28|
 id: 6| burstTime: 4| arrivalTime: 6| waitingTime: 37|
 id: 4| burstTime: 4| arrivalTime: 5| waitingTime: 42|
 id: 2| burstTime: 2| arrivalTime: 3| waitingTime: 48|
 id: 9| burstTime: 3| arrivalTime: 13| waitingTime: 40|
 id: 3| burstTime: 7| arrivalTime: 3| waitingTime: 53|
-------------------------------
Execution time: 63
-------------------------------
Average waiting time: 28
```

```
Running non-preemptive simulation #3
Simulation results:
---------------------
 id: 1| burstTime: 7| arrivalTime: 1| waitingTime: 0|
 id: 0| burstTime: 7| arrivalTime: 1| waitingTime: 7|
 id: 6| burstTime: 10| arrivalTime: 9| waitingTime: 6|
 id: 3| burstTime: 3| arrivalTime: 3| waitingTime: 22|
 id: 4| burstTime: 10| arrivalTime: 5| waitingTime: 23|
 id: 5| burstTime: 10| arrivalTime: 5| waitingTime: 33|
 id: 9| burstTime: 3| arrivalTime: 12| waitingTime: 36|
 id: 7| burstTime: 10| arrivalTime: 9| waitingTime: 42|
 id: 2| burstTime: 10| arrivalTime: 2| waitingTime: 59|
 id: 8| burstTime: 8| arrivalTime: 12| waitingTime: 59|
-------------------------------
Execution time: 79
-------------------------------
Average waiting time: 28
```

```
Running non-preemptive simulation #4
Simulation results:
----------------------
 id: 0| burstTime: 6| arrivalTime: 1| waitingTime: 0|
 id: 4| burstTime: 10| arrivalTime: 4| waitingTime: 3|
 id: 7| burstTime: 3| arrivalTime: 7| waitingTime: 10|
 id: 6| burstTime: 9| arrivalTime: 7| waitingTime: 13|
 id: 5| burstTime: 7| arrivalTime: 6| waitingTime: 23|
 id: 8| burstTime: 7| arrivalTime: 8| waitingTime: 28|
 id: 1| burstTime: 5| arrivalTime: 2| waitingTime: 41|
 id: 9| burstTime: 7| arrivalTime: 8| waitingTime: 40|
 id: 3| burstTime: 3| arrivalTime: 4| waitingTime: 51|
 id: 2| burstTime: 8| arrivalTime: 2| waitingTime: 56|
------------------------------
Execution time: 66
------------------------------
Average waiting time: 26
```

2.1.6 Preemptive results

```
Running preemptive simulation #0
Simulation results:
----------------------
 id: 2| burstTime: 2| arrivalTime: 2| waitingTime: 6|
 id: 4| burstTime: 3| arrivalTime: 3| waitingTime: 9|
 id: 5| burstTime: 4| arrivalTime: 4| waitingTime: 10|
 id: 9| burstTime: 5| arrivalTime: 6| waitingTime: 16|
 id: 3| burstTime: 6| arrivalTime: 3| waitingTime: 27|
 id: 1| burstTime: 9| arrivalTime: 2| waitingTime: 28|
 id: 8| burstTime: 3| arrivalTime: 5| waitingTime: 33|
 id: 0| burstTime: 6| arrivalTime: 1| waitingTime: 42|
 id: 6| burstTime: 8| arrivalTime: 4| waitingTime: 44|
 id: 7| burstTime: 10| arrivalTime: 5| waitingTime: 42|
------------------------------
Execution time: 57
------------------------------
Average waiting time: 25
```

```
Running preemptive simulation #1
Simulation results:
----------------------
 id: 7| burstTime: 4| arrivalTime: 5| waitingTime: 9|
 id: 0| burstTime: 6| arrivalTime: 1| waitingTime: 18|
 id: 6| burstTime: 4| arrivalTime: 5| waitingTime: 21|
 id: 2| burstTime: 2| arrivalTime: 2| waitingTime: 27|
 id: 4| burstTime: 4| arrivalTime: 3| waitingTime: 29|
 id: 5| burstTime: 4| arrivalTime: 4| waitingTime: 30|
 id: 9| burstTime: 8| arrivalTime: 6| waitingTime: 30|
 id: 1| burstTime: 10| arrivalTime: 2| waitingTime: 43|
 id: 3| burstTime: 10| arrivalTime: 3| waitingTime: 46|
 id: 8| burstTime: 10| arrivalTime: 6| waitingTime: 47|
------------------------------
Execution time: 63
------------------------------
Average waiting time: 30
```

```
Running preemptive simulation #2
Simulation results:
----------------------
 id: 0| burstTime: 3| arrivalTime: 2| waitingTime: 8|
 id: 9| burstTime: 2| arrivalTime: 7| waitingTime: 8|
 id: 6| burstTime: 4| arrivalTime: 5| waitingTime: 12|
 id: 4| burstTime: 2| arrivalTime: 4| waitingTime: 17|
 id: 2| burstTime: 6| arrivalTime: 3| waitingTime: 30|
 id: 5| burstTime: 3| arrivalTime: 4| waitingTime: 33|
 id: 7| burstTime: 8| arrivalTime: 5| waitingTime: 34|
 id: 3| burstTime: 10| arrivalTime: 3| waitingTime: 37|
 id: 8| burstTime: 9| arrivalTime: 6| waitingTime: 37|
 id: 1| burstTime: 5| arrivalTime: 2| waitingTime: 47|
-------------------------------
Execution time: 54
-------------------------------
Average waiting time: 26
```

```
Running preemptive simulation #3
Simulation results:
----------------------
 id: 7| burstTime: 2| arrivalTime: 7| waitingTime: 7|
 id: 3| burstTime: 2| arrivalTime: 2| waitingTime: 22|
 id: 8| burstTime: 2| arrivalTime: 7| waitingTime: 19|
 id: 2| burstTime: 6| arrivalTime: 2| waitingTime: 26|
 id: 0| burstTime: 9| arrivalTime: 1| waitingTime: 26|
 id: 4| burstTime: 8| arrivalTime: 3| waitingTime: 31|
 id: 9| burstTime: 5| arrivalTime: 8| waitingTime: 30|
 id: 1| burstTime: 6| arrivalTime: 1| waitingTime: 42|
 id: 6| burstTime: 7| arrivalTime: 5| waitingTime: 44|
 id: 5| burstTime: 9| arrivalTime: 3| waitingTime: 45|
-------------------------------
Execution time: 57
-------------------------------
Average waiting time: 29
```

```
Running preemptive simulation #4
Simulation results:
----------------------
 id: 3| burstTime: 6| arrivalTime: 3| waitingTime: 8|
 id: 8| burstTime: 3| arrivalTime: 6| waitingTime: 13|
 id: 7| burstTime: 5| arrivalTime: 6| waitingTime: 12|
 id: 2| burstTime: 2| arrivalTime: 3| waitingTime: 24|
 id: 0| burstTime: 5| arrivalTime: 2| waitingTime: 23|
 id: 6| burstTime: 4| arrivalTime: 5| waitingTime: 28|
 id: 4| burstTime: 5| arrivalTime: 4| waitingTime: 29|
 id: 9| burstTime: 2| arrivalTime: 7| waitingTime: 34|
 id: 5| burstTime: 7| arrivalTime: 5| waitingTime: 33|
 id: 1| burstTime: 10| arrivalTime: 2| waitingTime: 39|
-------------------------------
Execution time: 51
-------------------------------
Average waiting time: 24
```

2.2 Result Discussion

2.2.1 Do the simulation results for the non-preemptive and preemptive versions of the scheduling algorithm differ from any observable patterns?

The non-preemptive average waiting time for 5 simulations = 26 ticks

The preemptive average waiting time for 5 simulations = 26.8 ticks

Regarding average waiting time and execution time, both results are very similar, However, the Important differences are found in the waitingTime and arrivalTime values.

Preemptive shows a bias towards finishing shorter burstTimes first and so it skews the waiting time towards more average values, and what I mean is that many processes fluctuate from working to passive and so it averages out the waiting times.

Non-preemptive shows no bias towards any burst Times and shows how much more direct it is at finishing certain processes.

2.2.2 Would the observable behavior for non-preemptive vs preemptive versions be different if the RNG seed was different?

Rng affects the following variables:

arrivalChance, burst times, first Index, and what index for the next available process.

So in terms of the most influential value, it would be the burst times and they could affect the behavior, assuming that the rng seed contains many processes that have short burst times or only long burst times.

Another potentially influential variable would be arrivalChance since if the probabilities were to tend to be over 0.75% then there would be higher arrivalTimes and so if no new processes are added then and affect the process's waiting times since most would be done without other processes waiting.

The other variables would make small changes that are not large enough to affect the pattern already discussed in 2.2.1.

2.2.3 What if the number of simulations was increased to, e.g., 10000?

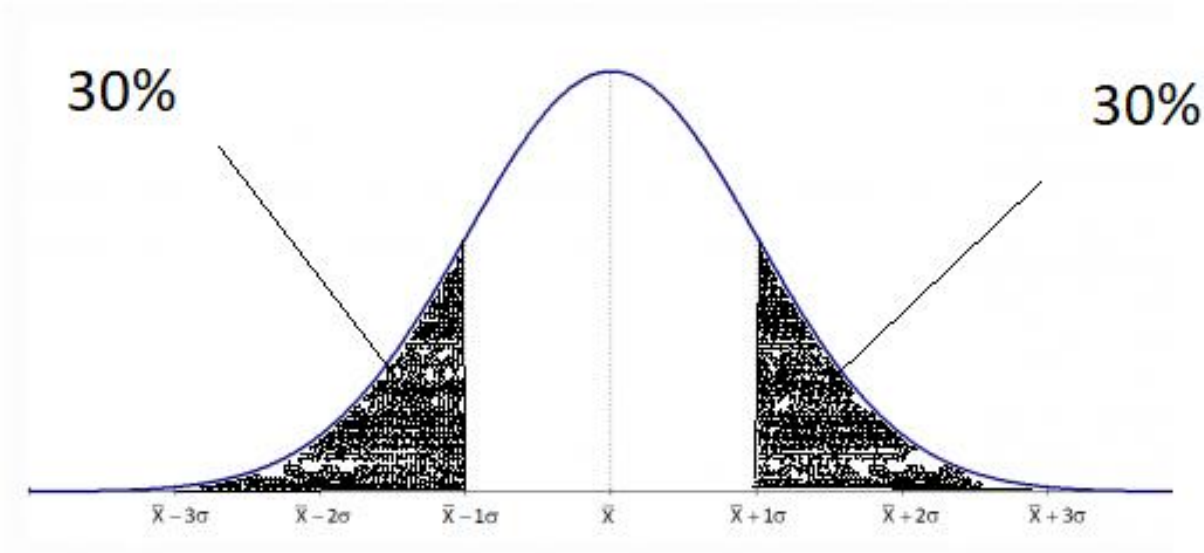
I would assume that if the values for each BurstTime in non-preemptive were adjusted in a normal distribution graph, it would look this way.

So simulation one has the following burst times [7, 7, 4, 2, 7, 9, 9, 9, 9, 2]

Each of them would be put in the graph and the longer the simulation the more expected the values to find no pattern towards any number in any position.

And waiting times will always follow the same pattern: starting with 0 and finishing with a value close to execution time.

While for preemptive would probably look this way:



Source: automateexcel.com

I presume that most of the expected values of burstTimes would fall outside the mean and be heavily skewed towards both sides.

For example [small, small, small, small, medium, medium, medium, large, large, large]

So values between 2 and 4 would fall in that 30% of the first 3 values, between 5 and 7 in between, and values 8 – 10 would fall in that other 30% of the last 3 columns.

Values are made up of course, but it falls on the rationable guess territory.

The waiting time for the last process might look similar to the average waiting time of the simulation.

2.2.4 What are the advantages and disadvantages of such a random scheduling algorithm compared to the First Come First Served (FCFS) algorithm?

Pros for FCFS: While not very practical in more complex systems, for systems that do not take much user input and are not too complex; so it usually does not have large processes, it can make it so most processes are done quickly and without creating a queue that spends memory. (The scenario described would also positively affect the other schedulers as well but FCFS benefits the most) and practically it would work efficiently enough in a batch system.

Pros for non-preemptive random scheduler: Used mostly if some processes need to take priority over others so those are done first without any waiting time in between.

Pros for preemptive random scheduler: a necessity for more complex systems, computers would render useless or very awkward to use without it because some processes such as display or speaker should take priority over a Bluetooth process let's say.

Cons for FCFS: impractical for complex systems because might take focus on some less important processes rather than others that are fundamental.

Cons for non-preemptive random scheduler: if the user requires certain processes skewed to a certain direction, can be from a complex system like a computer or a simpler system like a batch system, also if all processes are equally important then its not as good since shorter bursTimes being done last occupies up memory and makes it less practical.

Cons for preemptive random scheduler: there might be situations where a large process should take priority and this scheduler will most likely leave them last and create a situation where it might make some processes a waste of CPU. For instance, an important software update would be put last and a sound update first and once the new software update is done, the sound update would need to be installed again.