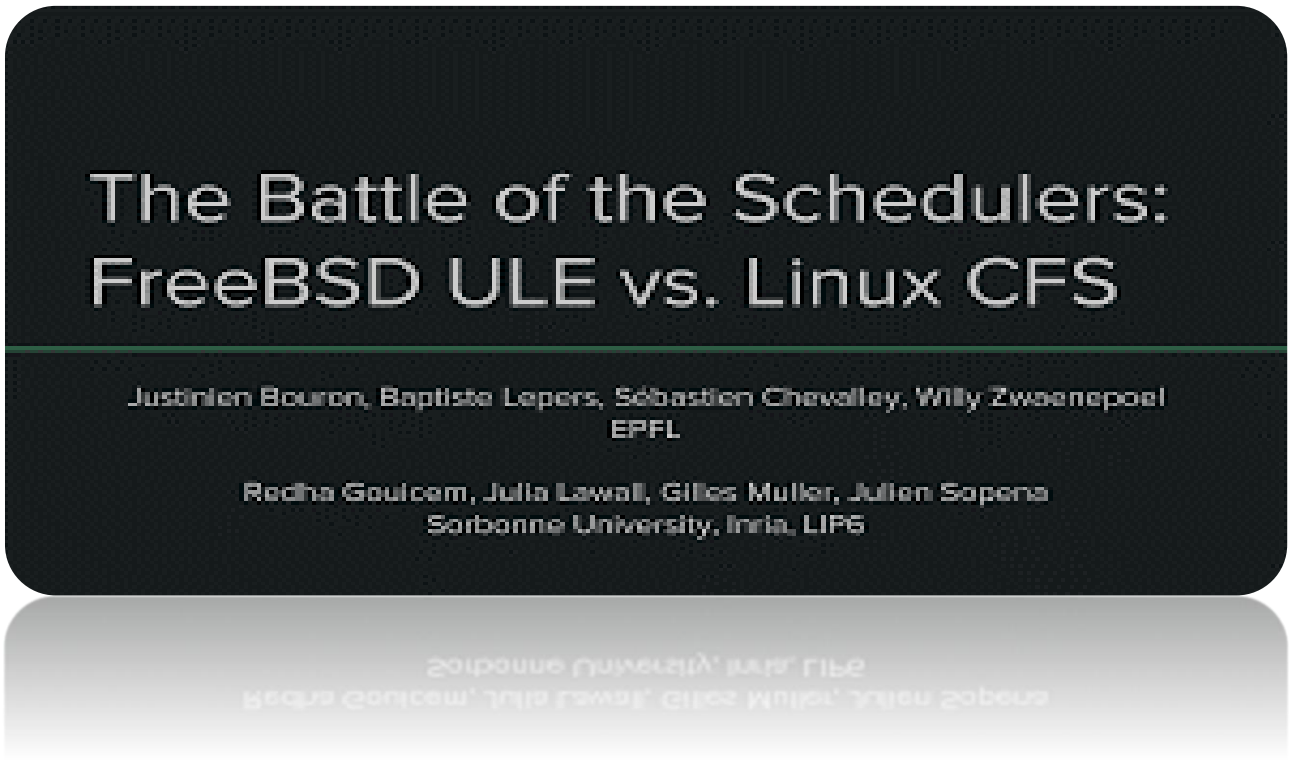
Linnéuniversitetet
Kalmar Växjö

20HT – 1DV512 – Operating Systems
Individual Assignment 2



i

Student: Sami Mwanje

ID/mail: mm223kk@student.lnu.se

Assignment date: 2020-12-06

Hand in date: 2021-06-18

Sami Mwanje
mm223kk@student.lnu.se

1. **Does ULE support threads, or does it support processes only? How about CFS?** To answer this, we must first shortly answer what a thread is. Threads are virtual cores created by programming instructions. Threads are designed to allow a single CPU core to appear to be split into two cores. A processor that contains two cores will be able to use two threads and a processor with four cores will be able to use eight threads. A thread is created when a process is started/executed. How many threads that are created depends on how the process is designed. How many threads that can be used depends on the CPU. A single threaded process can only run on one CPU (core). A multithreaded process can run on several CPUs (cores). A multithreaded process can share code, data, files and so on with threads within it. This enables the user to run more tasks at the same time without slowing down the performance. Every newly created/executed process will be assigned a core. If there is no available core the virtual cores (threads) will be used. If there are no available cores or threads the started/executed process will have wait in line using a scheduling algorithm. Both ULE and CFS are schedulers. ULE is designed for free BSD and CFS is designed for Linux. Both CFS and ELE are designed so they can handle a various number of threads. ULE has 2960 lines of code and CFG has 17900 lines of code. Both ULE and CFS have their advantages and disadvantages.

2. **How does CFS select the next task to be executed?** The completely fair schedular (CFS) uses three main ideas to select which task that will be executed next. Nice value, Virtual run time (vruntime) and priority. Priority is in some way a calculation of the nice value and vruntime. Linux uses a various amount of scheduling classes. When it comes to selecting which task to run next the Linux schedular selects the highest priority scheduling class, and within in that it schedules the highest priority task. In this exercise the highest priority scheduling class is CFS. When it comes to prioritizing which task to run next, the CPU assigns a proportion of the time it takes to process each individual task. The amount of time is calculated using the nice value. The nice

value can be a number from -20 to +19. A lower nice value will indicate a higher priority and a higher nice value will indicate a lower priority. The nice value is not the only calculation that occur when it comes to selecting priorities. Priority is also calculated by using Virtual run time (vruntime). The CFS records how long each task has run and allocates it as vruntime. The vruntime has a decay rate which is connected to how high prioritized a task is. The higher priority a task has the higher is the decay factor, the lower priority a task has the lower the decay rate is. For a task to have a so called “normal priority” the nice value must 0. If the nice value is 0 the vruntime will be identical to physical run time. If a zero nice value task runs for 200 milliseconds the vruntime will be 200 milliseconds. However, if a task that has lower priority runs for 200 milliseconds the vruntime will be higher than 200 milliseconds. This means that if a task with a high priority runs for milliseconds the vruntime will be less than 200 milliseconds. When it then comes for the CFS to decide which task to run next it selects the task with the lowest vruntime. To describe the process more accurate we would say that the executed task is within the most prioritized cgroup, and in that specific cgroup the executed task is the most prioritized task (thread). More about csgroups in the next exercise.

3. **What is a *cgroup* and how is it used by CFS? Does ULE support *cgroups*?** Linux divides all tasks (threads) in groups called cgroups (control groups). Every cgroup holds a various number of threads. Which threads that are going to belong to a specific cgroup is based on how prioritized the thread is. In recent exercise we discussed how this prioritization occurs with nice value and vruntime. Tasks with a higher prioritization belong to a higher prioritized cgroup, and tasks with a lower prioritization belong a lower prioritized cgroup. Even within the cgroups tasks are scheduled based on their priority. There can even me nested cgroups, which means cgroups within cgroups, even here a priority order occurs. So, let us say the computer is executing a various

number of tasks. To choose which task to execute next the CFS uses the sum of vruntime of the threads within a cgroup and schedules the cgroup with the lowest vruntime. The CFS will execute the task (thread) with lowest vruntime within the scheduled cgroup. As we can see there is a risk that a task or cgroup gets underprioritized. To resolve this issue so called “starvation”, the CFS schedules all tasks (threads) within a cgroup for only a given time. If less than eight threads are scheduled to a core, the default time period will be 48ms. If more than eight threads are scheduled to a single core the time period will be equal to “6 * number of threads ms”. CFS tries to avoid having a large difference between different threads vruntime inside a cgroup. This avoids a thread with low vruntime from running for a time that leads other threads to starvation. A rule is that the difference of vruntime for two threads cannot be larger than 6ms. This must be done because higher prioritized threads have a more share of the time period within the cgroup. If a thread would have a much bigger share of the time period than other threads within the croup, other thread would starve. Therefore, a newly created thread starts with a vruntime that is equal to the maximum vruntime of the threads waiting in the runqueue for current cgroup. A thread that is woken up will be assigned a vruntime that is minimum the threads in the schedule queue. This will mean that threads that sleep longer will get scheduled first. So, newly awaken threads do not have to wait for a long time before getting scheduled. This can for example be an input device that waits for the user to interact with it. Here the user may want the system to react as fast as possible, though the device may have not been used for a while. Unlike CFS ULE does not use control groups. It is built on three run queues that schedules threads. These three run queues contain interactive, batch and idle threads. The CFS does not see threads as a cgroup, it sees each thread independently. Though the CFS does keep threads in the queue in some sort of group (FIFO). More about them in the next exercise.

Sami Mwanje
mm223kk@student.lnu.se

4. **How many queues in total does ULE use? What is the purpose of each queue?** As stated in the last exercise ULE uses three different queues. Interactive run queue for interactive threads, batch queues for considered batch files. More about them in the next exercise. And idle for idle threads. Batch processes have their own queues because they often execute without user interference, the latency it takes to schedule the process itself is not important here. The main goal for the ULE is to give priority to the most interactive tasks, but also have some sort of fairness when doing so.

5. **How does ULE compute priority for various tasks?** To define “interactive”, ULE uses an interactive penalty metric. This penalty has a value between 0 and 100. If the value is less than 50 the thread is considered less interactive, which means that it spends more time sleeping than being awake. An interactive value above 50 means that the thread spends more time awake than sleeping. The interactive value can be calculated with using a function that depends on s and r. R is the time the thread has spent running and S is the time the thread has spent sleeping. The ULE keeps track of a thread’s sleeping and waking history for the past 5 seconds. When a thread is newly created it inherits r and s from its parent. When it comes to classifying the hierarchy of a thread ULE uses “interactive penalty value + nice value”. As stated in the last exercise the nice value is a number from -20 to +19 (for CFS). A lower nice value will indicate a higher priority, and a higher value will indicate a lower priority for the threads. A value less than 30 means that the process is interactive and spends 60 % of its time sleeping. If this is not the case the ULE will classify to process as a batch. As we can clearly see a negative nice value gives a process a more interactive classification. We can also see that if a thread has the interactive penalty value 0, it will be considered very interactive. Unlike CFS that uses cgroups of threads that are considered to have the same priority

value, ULE uses FIFO inside run queues. Every interactive value has its FIFO, and these are located inside the run queue. A thread is added to the FIFO with the corresponding threads priority inside the run queue. This FIFO uses an index that corresponds to its priority inside the run queue. When a thread is going to run it is taken from the FIFO with the highest priority, and which thread that is going to run here depends on first in first out. Unlike interactive threads, batch files depend on their run time. The more they run the lower priority value they get. They are also located inside a FIFO waiting to be scheduled, which FIFO they get depends on running time priority + nice value, this FIFO is located in the batch queue. The ULE always searches the run queue first for interactive processes, if the run queue is empty, it will continue to search the batch queue, if both are empty the system will be considered idle. This may occur a starvation on the batch queue due to the under prioritization, but this is very rare because interactive processes sleep more than 60 % of the time.

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

   Preemption means that an executing process can be interrupted in order to be resumed on the same execution line at a later time. This interrupt is often controlled by schedulers and it lets other processes execute while another one is resting in order to gain faster execution time for the required processes. ULE only allow preemption for kernel threads, there is no full preemption support for normally running threads. CFS however preempts running threads when their vruntime is “much greater” than a newly awakened thread. In practice at least 1 ms difference means “much greater”. This means that threads scheduled on ULE have to wait for the core to first execute the running thread before another one can be executed. ULE however can execute one thread interrupt it and then execute another. Threads running on ULE nor sleep or woken up. Threads running on CFS sleep and wakes up all the time. This means that preemption may slow down some applications if it has to wait for another

thread before continue executing. But it also may speed up other applications like “Sysbench” due to the preemption.

7. **Did Bouron et al. discover large differences in per-core scheduling performance between CFS and ULE? Which definition of "performance" did they use in their benchmark, and why?** The largest difference that was found was that CFS’s goal was to even out the amount of work on all the cores while ULE was trying to even out the number of threads on all the cores. Which one that is faster depends on the processes that is running on the core. We could see big differences between the two schedulers when executing Sysbench and Fibo. Sysbench was executed in 235s on CFS, and 143s on ULE. This did not mean that ULE was better when it comes to per-core scheduling, it meant that these two schedulers were working in different ways. CFS shared 50 % of the work between Fibo and Sysbench while, ULE starved Fibo under the execution of Sysbench. When it comes to single applications ULE could be sometimes faster than CFS. For example, Sysbench that uses 129 threads was faster on ULE than CFS. As stated, earlier ULE uses interactive penalties, which means that some threads will be prioritized. The interactive penalties are inherited from their parent. When Sysbench is created it inherits its interactive penalty from the bash process that forked it. Bash process are mostly sleeping which means they have a very low interactivity penalty. Threads created early on Sysbench have a low interactive penalty while the threads created later on will be starved and never executed. This however is very effective on applications where all threads have the same task. When it came to a single threaded application like “Scimark” CFS was performing 36 % faster. This application uses Java threads in order to run, which can be a problem for the ULE scheduler. When Scimark is launched the Java threads within it gets marked as interactive. This means that the core will focus on the interactive threads before executing the application thread itself, and that leads to a higher execution time. The lack of full preemption in ULE is what leads to this result. If there was preemption in ULE,

the core would focus on executing the main thread for Scimark which would have the highest priority.

8. **What is the difference between the multi-core load balancing strategies used by CFS and ULE? Is any of them faster? Does any of them typically reach perfect load balancing?** As stated in the earlier exercise the largest difference between CFS and ULE is that CFS's goal is to even out the amount of work on all the cores while ULE is trying to even out the number of threads on all the cores. The load balance on the cores occurs when threads are newly created or just woken up. Load balancing happens more often on CFS than on ULE. CFS uses a balancing by hierarchical while ULE is trying to even out the number of threads on each core. When comparing these two we can see that CFS balances the load much faster, but CFS never reaches perfect load balance, while ULE balances much slower but reaches perfect balanced state. When launching a new process, ULE always aims to fork the threads on the core with the lowest number of threads. A test with the application "c-ray" proves that ULE takes 11 seconds to have runnable threads while CFS only takes 2. This is because the time it takes for ULE to wake up these threads. As stated, earlier ULE runs a thread while the others are waiting, while the first thread is being executed, the second thread is waiting for the first thread to wake it up and so on. On CFS all threads are woken up fairly, this makes the threads quicker runnable. But after this fast-awakening time the CFS schedular ends up with a bad load balance, so in the end both the ULE and CFS perform the same when it comes to "c-ray". ULE performs 2.75 % "better" than CFS, but this better should not be taken for granted. The test results shows that it depends on the processes that are being executed, rather than the core(s) that they are being executed. ULE can be slower in a single threaded java application while it is faster in a multi-threaded application. A summary of the differences between these two schedulers is that the CFS scheduler has lack of load balancing, while

the ULE schedular has lack of preemption. So, which one is better? Well, that depends.[1][2]

Task 2:

- **The Do the simulation results for the non-preemptive and preemptive versions of the scheduling algorithm differ with any observable patterns?**
  The non-preemptive simulation types got the average waiting ticks: **33.3, 39.6, 47.7, 49.5 and 56.7** for each simulation, which gave an average of **45.36 ticks per run**. Each simulation average run time was: **378.9, 57.5, 49.3, 52.4 and 73.6ms**. This gives an average run time of **58.2ms**. The first run took 378.9 ms because of system warm up and so on. So, I counted it as 58.2ms (average of the other four).

  The preemptive version got the average waiting ticks **50.7, 50.8, 50.9, 46.1 and 44.6 ticks per simulation**. This gives an average of **48.62** ticks per run. The average run time for each simulation was **40.1, 38.1, 37.0, 31.7 and 72.3ms**. This gave an average run-time of **43.84**ms.

  When observing the average waiting ticks, we first monitor an average of **45.36** ticks per run on the non-preemptive simulation, while the preemptive gets an average of **48.84** ticks per run. This does not show a clear difference, but when looking closer on the non-preemptive and comparing it with the preemptive it can be observed t that the preemptive sometimes went down to a number so low as **33.3 ticks**. This means that on a very good run where almost two processes are randomly scheduled for every tick and burst times are not so high, the average waiting time for a process can be very low. The non-preemptive simulation also shows an “unstable” behavior. The lowest

[1] Bouron et al. - The Battle of the Schedulers - FreeBSD ULE vs. Linux CFS (2018).pdf
[2] Abraham-Silberschatz-Operating-System-Concepts-10th-2018

Sami Mwanje
mm223kk@student.lnu.se

average waiting tick is **33.3** while the highest is **56.7**. This tells us that the performance can differ a lot depending on which processes that can queued first and its burst time.

The simulation times shows some larger differences. The non-preemptive average is **58.2** ms while the preemptive average is **43.84ms**. In the preemptive version all processes got executed for two ticks before getting preempted. The non-preemptive executed a process until it was complete. This means that the code behind this may be a bit more time consuming. The preemptive is a bit more forward with less checks because it only holds a process for a maximum of two ticks.

- **Would the observable behavior for non-preemptive vs preemptive versions be different, if the RNG seed was different? What if the number of simulations was increased to, e.g., 10000?** When running the first simulation the simulation time was **378.9ms.** This means that the first time the program is running, is always the slowest. If the program were running for a longer time the faster, it would have become. So, if the simulation gets increased the simulation time would decrease. Though I do not think that this would have any affect on the ticks because they are independent from how many times the program run. The program does the same tasks no matter how many times it is running. It will only depend on the random burst times, schedules, and executions. This means if I would use a more advanced seed the simulation time would increase for the preemptive. The ticks would also increase. First of all, generating a more advanced seed takes more time. Large burst times and randomness may increase the simulation time for both preemptive and non-preemptive. It all depends on how random seed values the new seed will create. Burst times that cannot be divided by two like 1, 3, 7 and 9 can be time and tick consuming for the preemptive while the non-preemptive have

no problem with these. While burst times that can be divided by two can going smoothly on the preemptive. So, it all depends on the randomness.

- **What are the advantages and disadvantages of such a random scheduling algorithm compared to the First Come First Served (FCFS) algorithm?** The first come first can be fairer at a first glance. Looking closer at both of them we can see that the random schedular can be fairer than FCFS at some rare (random) times. With good random values and some luck on the choice of execution the random schedular can be the fastest. The FCFS always schedules the processes that can first. Not matter burst time. So, the processes waiting on the FCFS will always have the lowest waiting time if their burst time is not too high. Then FCFS with preemption may be needed. As long as the preemption exist the FCFS is the clear winner. Without preemption that FCFS can be as random as the random schedular. A process that came in first with a burst time of 20 loads while a process with a burst time of two have to wait. In the random schedular there is no meaning with coming first. The only meaning is to be chosen randomly first for execution. This can be a process with a burst time of 20 or a burst time of two. Fairness is more noticeable when preemption is enabled or the are no processes that are waiting for arrival. The more processes that are waiting to arrive the more unfair is the random schedular. It may schedule the processes that came first or the processes that can last or even the processes in middle. So, a large number of choices is not good for the random schedular. Even the preemptive version of the random schedular is unfair. The main task here for each process is to be random chosen as many times as possible until completion. These choices can sometime be fair and other times very unfair. It all depends really depends on the burst time of the processes for both the random schedular and the FCFS. A process with a low burst time will always have a lower waiting time, while a process with a high

burst time will always have a higher waiting time on both FCFS and Random schedular. In the end you may realize that the FCFS is also pretty random.

Output:

```
Running non-preemptive simulation #0
Simulation results:
----------------------
Process ID: 0, Burst time: 8, Waiting time: 52, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 5, Waiting time: 54, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 8, Waiting time: 50, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 5, Waiting time: 52, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 5, Waiting time: 51, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 5, Waiting time: 50, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 7, Waiting time: 47, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 3, Waiting time: 50, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 7, Waiting time: 45, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 7, Waiting time: 44, Arrival time: 9 (ticks)
Total simulation time: 378.9 ms.
Average waiting time: 49.5 (ticks)
Running non-preemptive simulation #1
Simulation results:
----------------------
Process ID: 0, Burst time: 5, Waiting time: 53, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 7, Waiting time: 50, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 7, Waiting time: 49, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 8, Waiting time: 47, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 8, Waiting time: 46, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 7, Waiting time: 46, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 4, Waiting time: 48, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 6, Waiting time: 45, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 2, Waiting time: 48, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 4, Waiting time: 45, Arrival time: 9 (ticks)
Total simulation time: 57.5 ms.
Average waiting time: 47.7 (ticks)
Running non-preemptive simulation #2
Simulation results:
----------------------
Process ID: 0, Burst time: 4, Waiting time: 38, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 2, Waiting time: 39, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 5, Waiting time: 35, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 2, Waiting time: 37, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 3, Waiting time: 35, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 8, Waiting time: 29, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 3, Waiting time: 33, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 6, Waiting time: 29, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 4, Waiting time: 30, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 5, Waiting time: 28, Arrival time: 9 (ticks)
Total simulation time: 49.3 ms.
Average waiting time: 33.3 (ticks)
Running non-preemptive simulation #3
Simulation results:
----------------------
Process ID: 0, Burst time: 5, Waiting time: 44, Arrival time: 0 (ticks)
```

Sami Mwanje
mm223kk@student.lnu.se

```
Process ID: 1, Burst time: 2, Waiting time: 46, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 3, Waiting time: 44, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 9, Waiting time: 37, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 7, Waiting time: 38, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 5, Waiting time: 39, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 8, Waiting time: 35, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 2, Waiting time: 40, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 2, Waiting time: 39, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 6, Waiting time: 34, Arrival time: 9 (ticks)
Total simulation time: 52.4 ms.
Average waiting time: 39.6 (ticks)
Running non-preemptive simulation #4
Simulation results:
----------------------
Process ID: 0, Burst time: 9, Waiting time: 59, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 4, Waiting time: 63, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 8, Waiting time: 58, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 6, Waiting time: 59, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 5, Waiting time: 59, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 6, Waiting time: 57, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 9, Waiting time: 53, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 9, Waiting time: 52, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 8, Waiting time: 52, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 4, Waiting time: 55, Arrival time: 9 (ticks)
Total simulation time: 73.6 ms.
Average waiting time: 56.7 (ticks)
Running preemptive simulation #0
Simulation results:
----------------------
Process ID: 0, Burst time: 9, Waiting time: 45, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 4, Waiting time: 49, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 9, Waiting time: 43, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 2, Waiting time: 49, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 3, Waiting time: 47, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 2, Waiting time: 47, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 8, Waiting time: 40, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 4, Waiting time: 43, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 6, Waiting time: 40, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 2, Waiting time: 43, Arrival time: 9 (ticks)
Total simulation time: 72.3 ms.
Average waiting time: 44.6 (ticks)
Running preemptive simulation #1
Simulation results:
----------------------
Process ID: 0, Burst time: 5, Waiting time: 56, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 6, Waiting time: 54, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 5, Waiting time: 54, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 7, Waiting time: 51, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 3, Waiting time: 54, Arrival time: 4 (ticks)
```

Sami Mwanje
mm223kk@student.lnu.se

```
Process ID: 5, Burst time: 8, Waiting time: 48, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 4, Waiting time: 51, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 6, Waiting time: 48, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 9, Waiting time: 44, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 5, Waiting time: 47, Arrival time: 9 (ticks)
Total simulation time: 40.1 ms.
Average waiting time: 50.7 (ticks)
Running preemptive simulation #2
Simulation results:
----------------------
Process ID: 0, Burst time: 7, Waiting time: 54, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 6, Waiting time: 54, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 3, Waiting time: 56, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 5, Waiting time: 53, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 7, Waiting time: 50, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 3, Waiting time: 53, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 3, Waiting time: 52, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 7, Waiting time: 47, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 9, Waiting time: 44, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 7, Waiting time: 45, Arrival time: 9 (ticks)
Total simulation time: 38.1 ms.
Average waiting time: 50.8 (ticks)
Running preemptive simulation #3
Simulation results:
----------------------
Process ID: 0, Burst time: 4, Waiting time: 57, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 3, Waiting time: 57, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 8, Waiting time: 51, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 5, Waiting time: 53, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 2, Waiting time: 55, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 7, Waiting time: 49, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 9, Waiting time: 46, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 2, Waiting time: 52, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 9, Waiting time: 44, Arrival time: 8 (ticks)
Process ID: 9, Burst time: 7, Waiting time: 45, Arrival time: 9 (ticks)
Total simulation time: 37.0 ms.
Average waiting time: 50.9 (ticks)
Running preemptive simulation #4
Simulation results:
----------------------
Process ID: 0, Burst time: 5, Waiting time: 51, Arrival time: 0 (ticks)
Process ID: 1, Burst time: 6, Waiting time: 49, Arrival time: 1 (ticks)
Process ID: 2, Burst time: 3, Waiting time: 51, Arrival time: 2 (ticks)
Process ID: 3, Burst time: 2, Waiting time: 51, Arrival time: 3 (ticks)
Process ID: 4, Burst time: 8, Waiting time: 44, Arrival time: 4 (ticks)
Process ID: 5, Burst time: 3, Waiting time: 48, Arrival time: 5 (ticks)
Process ID: 6, Burst time: 5, Waiting time: 45, Arrival time: 6 (ticks)
Process ID: 7, Burst time: 8, Waiting time: 41, Arrival time: 7 (ticks)
Process ID: 8, Burst time: 8, Waiting time: 40, Arrival time: 8 (ticks)
```

Sami Mwanje
mm223kk@student.lnu.se

```
Process ID: 9, Burst time: 6, Waiting time: 41, Arrival time: 9 (ticks)
Total simulation time: 31.7 ms.
```
Average waiting time: 46.1 (ticks)

[i] https://blog.desdelinux.net/wp-content/uploads/2012/01/freebsd_logo.png

Sami Mwanje
mm223kk@student.lnu.se