EURECOM
Sophia Antipolis

# Exam
# Operating Systems - OS - Fall 2019

Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr

February, 2020

**Authorized documents**: closed books and notes, except one A4 sheet (i.e. two pages) with handwritten notes. The grading takes into account the fact that you don't have any other document with you.
A grade is provided for each question (beware: be sure to organize your time with regards to the grading policy). 1 additional point is given for general appreciation, including writing skills and readability.

## 1 Course knowledge (5 points, ∼30 minutes)

***a***. What is the difference between a user-level thread and a kernel-level thread? Which scheduling policy is applied to user-level threads? [2 points]

*User-level threads are threads which are managed by a library running in user mode. For instance, Java green threads are handled by the Java Virtual Machine, i.e. the Operating System is not aware these threads run in the system. The library is responsible for scheduling these threads. Thus, the scheduling policy depends on the used library. Java green threads are scheduled with a round-robin approach, and with fixed priorities.*

*On the contrary, kernel threads are managed by the Operating System that has routines to create them, schedule them and destroy them. Kernel threads can use the support offered by processors (multi-core, hyper-threading).*

***b***. What are the two reasons for getting a page fault? When a page fault occurs, what are the actions taken by the Operating System for these two reasons, respectively? [3 points]

*A page fault occurs when a memory request (read, write) to a given page is performed by a process to the MMU, and this page is not present in physical memory: the MMU*

*informs the OS of this situation via a trap. The two reasons for this situation can be:*

- *The page does not exist, i.e. the process is trying to access to an invalid address. In this case, the OS kills the faulty process (e.g. "Page/segmentation fault".*
- *The page has been swapped out, e.g. on disk. There, the OS must swap in the swapped out page. For this, it may need to swap out another page in order to free space in main memory. If the swap out/in process is expected to last, the process is put on I/O wait and another process might be scheduled meanwhile.*

## 2 Deadlocks (6 points, ∼30 minutes)

***a***. Define what is a deadlock. [1 point]

*A deadlock is a situation in which a process waits for a resource that will never be available. For instance, a process tries to lock a mutex that will never be freed.*

***b***. Give an example of a program[1] that sometimes leads to a deadlock situation, and sometimes not. You should use at least two threads and at least one shared object protected by at least one mutex in your program. The deadlock situation should be linked to the shared object and the mutex. [5 points]

*A simple program that may lead to a deadlock is to use two mutex objects, m1 and m2, two processes P1 and P2, and a shared variable (a). P1 first tries to lock m1 and then m2, and P2 tries to lock m2 and then m1. Let us assume P1 runs totally before P2: it will be able to lock m1, then m2. Let us now assume that P1 locks m1, and then P1 is preempted. P2 locks m2. Now, P1 is blocked since it waits for m2, and P2 is blocked because it waits for m1: this is a deadlock situation. We thus have a program that may sometimes lead to a deadlock situation.*

```
P1()
mutex_lock(m1)
mutex_lock(m2)
a++
mutex_unlock(m2)
mutex_unlock(m1)
```

```
P2()
mutex_lock(m2)
mutex_lock(m1)
a--
mutex_unlock(m1)
mutex_unlock(m2)
```

## 3 Load balancing in Linux (10 points, ∼60 minutes)

**Answer to the following questions after having understood the text that is provided after the questions.**

***a***. List three main objectives of the Linux scheduler? How can load balancing support these objectives? [3 points]

[1] Only pseudo code is requested

- ***Fairness***. *Tasks with the same similarities should be offered the same amount of computation power. Tasks with the same needs are to be placed in the same group, e.g. Real-Time, Batch, etc. One way to ensure fairness is to use round-robins.*

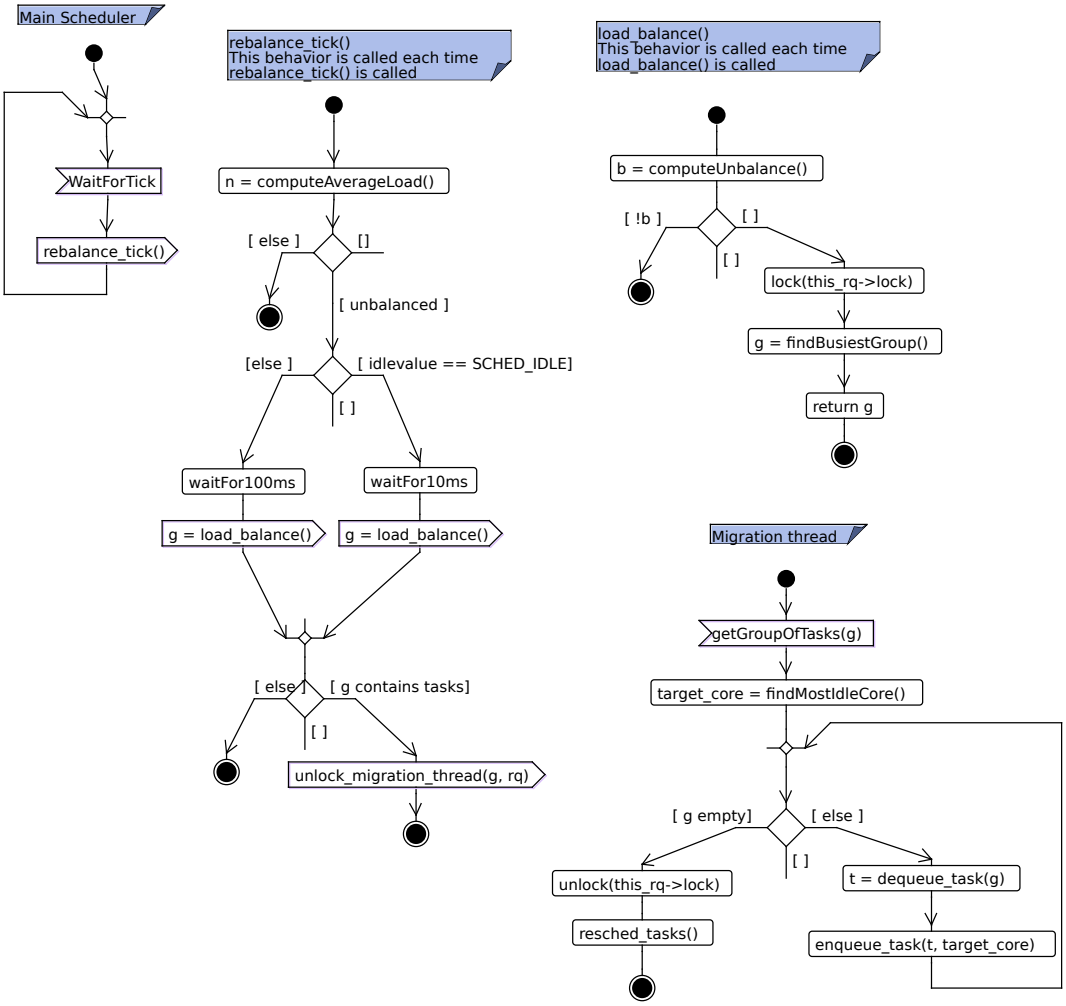
- ***Interactivity***. *The Linux scheduler boosts the priority of tasks that are interactive. The priority is inversely proportional to the use of the time quantum.*

- ***Keeping the CPU busy***. *The Linux scheduler targets the maximum CPU usage.*

*Task balancing helps enforcing the previous aspects. Balancing supports fairness because if a task is in the queue of a busy CPU, it may be moved to a less busy CPU, which is more fair. If an interactive task were to be mapped on a very busy CPU, the interaction it offers may be worse than the one that could be obtained with a task mapped on a idle processor. CPU usage is also maximized by having all CPU busy at the same time.*

***b***. Draw a figure (e.g. a flowchart) that explains how the main elements involved in the load balancing of the Linux kernel interact. [5 points]

*The following flowchart shows the main aspects of the different functions and their corresponding behavior according to the provided text. The busiest_lock has been left apart since we do not know when it is locked. The "10 ms" and "100 ms" are just there to illustrate the different delays according to the situation.*




***c***. Why is it necessary to re-schedule the system after tasks have migrated? [2 points]

*When a task that has just been added to a new core, this task may be more urgent —said differently: has a higher priority— than the currently running one → the scheduler has to be called. This ensures fairness with regards to tasks priorities.*

The following text is taken from: Lim, G., Min, C., Eom, Y.: Load-Balancing for Improving User Responsiveness on Multicore Embedded Systems. In: 2012 Linux Symposium (July 2012) 10.

---

The current SMP scheduler in Linux kernel periodically executes the load-balancing operation to equally utilize each CPU core whenever load imbalance among CPU cores is detected.

At every tick, the scheduler_tick() function calls rebalance_tick() function to adjust the load of the run-queue that is assigned to each CPU. There balance_tick() function

determines the number of tasks that exist in the run-queue. It updates the average load of the run-queue by accessing nr_running of the run-queue descriptor and cpu_load field for all domains from the default domain to the domain of the upper layer. If the load imbalance is found, the SMP scheduler starts the procedure to balance the load of the scheduling domain by calling load_balance() function. It is determined by idle value in the sched_domain descriptor and other parameters how frequently load-balancing happens. If idlevalue is SCHED_IDLE, meaning that the run-queue is empty, rebalance_tick() function frequently calls load_balance() function. On the contrary, if idlevalue is NOT_IDLE, the run-queue is not empty, and rebalance_tick() function delays calling load_balance() function.

For example, if the number of running tasks in the run-queue increases, the SMP scheduler inspects whether the load-balancing time of the scheduling domain belonging to physical CPU needs to be changed from 10 milliseconds to 100 milliseconds.

When load_balance() function moves tasks from the busiest group to the run-queue of other CPU, it calculates whether Linux can reduce the load imbalance of the scheduling domain. If load_balance() function can reduce the load imbalance of the scheduling domain as a result of the calculation, this function gets parameter information like this_cpu,this_rq,sd, and idle, and acquires spin-lock called this_rq->lock for synchronization. Then, load_balance() function returns sched_group descriptor address of the busiest group to the caller after analyzing the load of the groupin the scheduling domain by calling find_busiest_group() function. At this time, load_balance() function returns the information of tasks to the caller to move the tasks into the run-queue of local CPU for the load-balancing of scheduling domain.The kernel moves the selected tasks from the busiest run-queue to this_rq of another CPU.

After turning on the flag, it wakes up migration thread. The migration thread scans the hierarchical scheduling domain from the base domain of the busiest run-queue to the top in order to find the most idle CPU. If it finds relatively idle CPU, it moves one of the tasks in the busiest run-queue to the run-queue of relatively idle CPU (calling move_tasks() function). If a task migration is completed, kernel releases two previously held spin-locks (busiest->lock and this_rq->lock), and finally it finishes the task migration. dequeue_task() function removes a particular task in the run-queue of other CPU. Then, enqueue_task() function adds a particular task into the run-queue of local CPU. At this time, if the priority of the moved task is higher than the current task, the moved task will preempt the current task by calling resched_task() function to gain the ownership of CPU scheduling.

---