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]

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]

2 Deadlocks (6 points, ∼30 minutes)

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

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]

[1] Only pseudo code is requested

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]

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]

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

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.
