AN OPERATING SYSTEM SYNCHRONIZATION IMPLEMENTATION

(Due Date: 10.12.2017 Sunday – 23:55)

CMPE322 project series is a set of projects that teaches how to construct an operating system, named *BUnix (written in institutional colors of Boğaziçi University ☺)*, from scratch. In the first project, you scheduled processes which are independent from each other. In addition to your implementation of the first project, now these processes have to use the same critical resources, thus you also have to synchronize them with binary semaphores.

Scenario in the Second Project

BUnix manages a hardware which has some critical resources that only one process at a time is allowed to use. So, you inform your BUnix coders that if they need to use a critical section, they have to write "waitS_[Semaphore_No]" to not permit other processes to use that critical resource. Also, you remind them that they should not forget to write "signS_[Semaphore_No]" to release this critical section after it is completed. You allow them to use at most 10 semaphores which are numbered from 0 to 9. After all, they compile their new codes which is in the form that is presented in Table 1.

INSTRUCTION NAME	INSTRUCTION EXECUTION TIME (ms)
instruction_1	20
instruction_2	20
waitS_0	0
instruction_3	50
instruction_4	20
signS_0	0
waitS_2	0
instruction_5	30
instruction_6	30
signS_2	0
instruction_7	30
. . .	. . .
exit	10

Table 1 – A Text Based Program Code File Example

The Synchronization Code

In addition to the first project, we want you to implement the following parts in your second project:

- Implement a wait queue for each semaphore. These wait queues should be a FIFO again.
- Implement "waitS" operation:
    - If it is not used by any process allow the current process to use it and lock the semaphore.
    - Otherwise remove this process from the ready queue and send it to the corresponding wait queue.

    - Prevent the starvation of a process in wait queue. Allow a process to use this semaphore before any other process wants to use this semaphore. For example, P1 calls waitS_0 and starts to use critical section. Then P3 calls waitS_0 before P1 calls signS_0 which means that you have to send P3 to the wait queue of the semaphore 0. You have to send any other process (P1 included) to this wait queue if they call waitS_0 before P3 calls signS_0.
- Implement “signS” operation:
    - Allow the use of this semaphore again.
    - If there is any process in the wait queue of semaphore, remove it from the wait queue (do not forget that this queue is a FIFO!) and send it to the ready queue.
    - Continue to run this process.
- An output mechanism that generates a file to present the ready queue and wait queues whenever it is changed by the algorithm. The format of the files should be same as the first project.
- Ready queue output name should be “output.txt” again. Wait queue output names should be “output_[Semaphore_No].txt”, e.g. “output_0.txt”, “output_1.txt”. If a semaphore queue is not used according to test input files, you don’t have to create it.

Restrictions in Test Input Files

The following restrictions are added to reduce complexity of the implementation.

- In our test definition files:
    - P1 will always arrive at time 0
    - Process arrival times will increase in each row.
    - There will be always a process in the ready queue according to test definition files. So, you don't have to worry about what to do in "idle CPU" state.
- If a running process and a new arrived process should have been enqueued at the same time, first enqueue the new arrived process and then the running process. For example:

  140::HEAD-P2-P1-TAIL

  250::HEAD-P1-P3-P2-TAIL

  P2 consumes its dedicated time slot at 250 and P3 has arrived at the same time. First enqueue P3 and then P2.

- Your output file does not have to show the ready queue when a new process has arrived.

  For example:

  P1 has arrived at time 0

  P2 has arrived at time 120

  P3 has arrived at time 250

  Output:

  0::HEAD-P1-TAIL

  120::HEAD-P1-P2-TAIL

140::HEAD-P2-P1-TAIL

250::HEAD-P1-P3-P2-TAIL

290::HEAD-P3-P2-TAIL

You do not have to show the second line (time 120). On the other hand, you have to show the fourth line (time 250) when P3 is arrived but also P1 starts to execute by CPU.

- Execution time of waitS and signS operations are always 0.
- Your BUnix coders are really good coders that never call another waitS command before releasing the first one. So, don't be afraid from a deadlock. You will never see a code like Table-2.

INSTRUCTION NAME	INSTRUCTION EXECUTION TIME (ms)
waitS_0	0
instruction_3	50
waitS_2	0
instruction_5	30
signS_0	0

Table 2 – A Code Example that may cause a deadlock

Development Platform

You have to implement your design in the Linux Platform with GCC/G++ Compiler. We strongly advise you to install Ubuntu which has pre-installed GCC/G++ compiling tools. We will test your code in this platform and if your compiled code is not compatible with the test platform, you might be invited for a demo, if necessary.

Provided Files

The following files are given together with the project:

- The process definition file (number of processes and their arrival times will be modified to test your code).
- Four program code files (number of instructions, their names and their execution times will be modified to test your code. Only exception is the last instruction name is always "exit" and its execution time is 10 milliseconds).
- The expected output file.
- Note: The length of the time slot is a constant value in *BUnix* operating system (100 milliseconds).

Project Requirements

- Your project should have generated the expected output file for modified process definition file and program code files. (80%)
- The source code documentation. The documentation depends on your design style but at least having comments that explain your code is strongly advised. (20%)