Project III: Designing a Synchronization Mechanism for Your Prototype OS

1 Deadline

See course's homepage.

2 Objectives

In this project, you will design and implement a semaphore type to support race-free execution of a classic IPC problem.

3 Tasks

Based on your work in Project II, you need to finish the following tasks.

1. You will solve the *Bear and the Honey Bees* problem. It can be described as follows. There are *10* honeybees and a bear. They share a pot of honey. The pot is initially empty; its capacity is 30 equal-portions of honey. Each bee repeatedly gathers one portion of honey and puts it in the pot. In its life, each bee will gather 15 portions of honey. The bee that fills the pot awakens the bear. The bear sleeps until the pot is full, then eats the entire pot of honey and goes back to sleep. The bear wakes up to consume the honey 5 times. Note that while the bear is eating honey, no honeybees can add any honey to the pot. The bear and honeybees are represented as threads (1 bear thread and 10 honey bee threads). You are asked to develop a solution on your Altera board that simulates the actions of the bear and each honeybee. You need to instantiate semaphores for synchronization.
2. Design a semaphore type to provide mutual exclusion and synchronization for the bear and honey bees threads when they are working on the pot. Each semaphore must have its own blocking/waiting queue, as stated in the lecture. Your semaphore should support the operations listed in Table 1.
3. There should be an appropriate delay (again use a delay loop) between two consecutive attempts to add honey to the pot by a bee thread. When a thread gets scheduled, your program needs to print out whether its operation on the pot is successful or it is blocked/unblocked on a semaphore. This is applicable to both bear and bee threads.
4. After all threads finish, your program is supposed to resume the execution of prototype_os().

4 Constraints

1. A semaphore must have a counter variable and a blocking queue.
2. The atomicity of executing the critical section can only be guaranteed by your designed synchronization mechanism RATHER THAN by disabling context switching.
3. All required semaphor operations (Table 1) must be implemented.

5 Useful Hints

- Since the NIOS-II ISA does not have the "compare-and-swap" instruction, you have to provide atomicity for the semaphore type by disabling and enabling interrupt. However, the atomicity a critical section execution CANNOT be guaranteed by disabling scheduling. Instead, you need to strategically provide atomicity for your semaphore (e.g. when updating the semaphore value) and then use semaphore to provide mutual exclusion for threads
- Beware of the boundary conditions that a producer (or consumer) thread inserts (or removes) an X when the queue is full (or empty).

Table 1. Description of Required Semaphor Operations

mysem * mysem_create (int value)	It returns the starting address a semaphore variable. You can use *malloc()* to allocate memory space and initialize the internal data structure of the semaphore based on provided parameter.
void mysem_up(mysem* sem)	Increment the value of the semaphore. If one or more threads are sleeping on the semaphore, the one that has been sleeping the longest is allowed to complete its down operation. As such, after an up on a semaphore with threads sleeping on it, the semaphore value is still 0 but there is one fewer sleeping threads.
void mysem_down(mysem* sem)	Check to see if the semaphore value is greater than 0. If so, it decrements the value and just continues. If the value is 0, the thread is put to sleep without completing the down for the moment. As such, the semaphore value never falls below 0.
void mysem_delete(mysem* sem)	It deletes the memory space of a semaphore.
int mysem_waitCount(mysem* sem)	Return the number of threads sleeping on the semaphore.
int mysem_value(mysem* sem)	Return the current value of the semaphore.

6 Submission

You will submit a zipped file that contains the following components by the deadline:

1. A project report that includes:
   - the project title & names of the team members,
   - an introduction to your project goals, problem description and project management (including project timeline, teamwork, risk anticipation, resources, etc.),
   - your key ideas to solve the problem, and
   - an elaboration of how you accomplish your project tasks (including data structures, algorithms, strategy, etc.) using diagrams, flowcharts, tables or descriptions.
2. A summary of:
   - what you have accomplished in this project,
   - how it can help you understand what you have learned in class,
   - your evaluation of the work that you have done (e.g., how creative your proposed idea is, how simple and sweet your design and implementation are, how well your project is organized, etc.),
   - how the course project and its specifications can be improved, and
   - list the sources of citations appearing in your report,
3. You report should be formated as follows: 11pt Times-New-Roman fonts (the title and section headings can be larger); single-column; 1 inch margins all the way around a page; page numbers; no more than 8 pages in all; DO NOT paste source code with more than 10 lines; diagrams/charts/graphs with brief explanations are more preferable than thousands of words; submitted as a PDF file.
4. A folder containing your project source code with meaningful comments, as well as a README file that details the organization of your own source code (e.g., what each file is used for and how they function together) , as well as how to compile and run the program. Please DO NOT include any files or folders automatically generated by the NIOS-II IDE!

7 Grading Criteria

1. Project report: 30%
2. Correctness of the program: 65%
3. Detailed source code comments and README: 5%

8 Grading Rubric

1. The project can be compiled (2 pts).
2. The bear and honey bee threads can start (3 pts).
3. The bees should collectively add 150 portions into the pot (20).
4. The bear wakes up to consume the pot five times (20).
5. Each thread runs to completion (15 pts).
6. The main thread returns after all other threads have died and continues to run indefinitely (5 pts).