CSC 4410 Operating Systems | Spring 2018

Programming Assignment 2

Florida Southern College | *Due: February 28*

The purpose of this programming assignment is to familiarize you with semaphores. In this assignment you will implement semaphores, similar to the first form of semaphore we saw in class, using files. You will then use this type of semaphore to solve a classical synchronization problem. A note on error checking: System calls have return values that indicate failure. Be sure to check for those values so that your program can terminate gracefully when a system call fails.

Implementing Semaphores with Files

When `creat(2)` makes a new file, the file is opened for writing even if the permission argument doesn't allow writing. When `creat(2)` truncates an existing file, however, it fails if there is no write permission. This allows a file to be used as a semaphore since such a call to `creat(2)` succeeds if and only if the file doesn't exist. You can use the `unlink(2)` system call to delete the file. One idiosyncracy of using the `creat(2)` system call for this purpose is that you can exhaust the available file descriptors. For this reason, **you should `close(2)` the file descriptor associated with a file as soon as the file has been successfully created by a `creat(2)` call**.

Write two functions: lock(semaphore-name) and unlock(semaphore-name) using the method described above. Think about how lock and unlock relate to the semaphore operations wait and signal. Make use of the `sleep(3c)` function to reduce the busy-wait time in the lock routine.

Dining Philosophers

Use the semaphore routines created in the above section to implement a **deadlock-free solution** to the Dining Philosophers problem. Your solution should allow for multiple philosophers to eat simultaneosuly as long as mutual exclusion is not violated. Each philosopher will begin in the "thinking" state. Each time a philosopher becomes hungry or starts to eat or think, the program should display a line of text to standard out. The philosophers should continue to cycle through the states until you explicitly terminate the program.

For example, a sample of your output might look like this:

Philosopher 1 hungry.
Philosopher 1 eating.
Philosopher 4 hungry.
Philosopher 1 sated.
Philosopher 1 thinking.
Philosopher 4 eating.
Philosopher 4 sated.