CSCI 8530: Computer Science
Advanced Operating Systems
Spring 2020
Program Assignment 2

Due on March 31, 2020

Introduction

The synchronization of processes and threads in a computer system is an important task. Any study of operating systems and concurrent programming would be significantly lacking without careful attention to this topic.

A typical introductory study of operating systems will consider various synchronization mechanisms and their applications to typical problems. Common mechanisms would certainly include hardware primitives, like atomic test and set instructions and the disabling of interrupts, and software approaches, certainly including semaphores (and variants, like mutexes), but may also include monitors, event counters, sequencers, and message passing. Typical applications would likely include classic problems like dining philosophers, readers/writers, and producer/consumer. Of course, there are many more possibilities.

An introductory study must, of necessity, omit coverage of more sophisticated (and complicated) synchronization approaches. In *OR* synchronization, a process may wait for any one of a collection of events, and take a different action depending which event actually occurred. The Microsoft Windows OS API includes the **`WaitForMultipleObjects`**[1] function, which allows a process or thread to wait for one or all of a set of objects to become signaled[2]. The semaphore operations provided by AT&T's System V UNIX OS, and also provided in Linux[3], are much more complicated, but are also extremely adaptable to different types of synchronization.

AND Synchronization

In this problem, we're concerned with *AND* synchronization[4]. This is the type of synchronization that is needed when exclusive use of each of several objects must be obtained before a process can continue. Consider two objects, *A* and *B*, that are controlled by semaphores *semA* and *semB*, respectively. Now consider two processes *X* and *Y*, each wanting to simultaneously use *A* and *B*. They might do this:

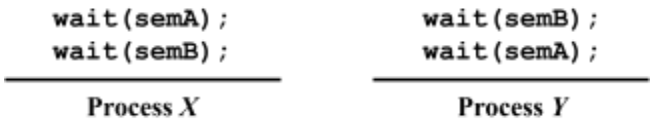
---

[1] For details, just search the web for **`MSDN`** and **`WaitForMultipleObjects`**.

[2] In the Windows API, the state of a semaphore is set to signaled when its count is greater than zero, and non-signaled when its count is zero.

[3] Try "**`man semop`**" to display the manual page; it includes references to **`semctl, semget, svipc,`** and **`sem_overview.`**

[4] An more extensive discussion of *AND* synchronization and process synchronization, in general, can be found in chapter 2 of **Operating Systems: Advanced Concepts** by Maekawa, Oldehoeft and Oldehoeft, Benjamin/Cummings, 1987.



If the two processes perform the **wait** operations in the adjacent lines simultaneously, then *deadlock* will occur. A simple solution in this case would be to restrict the order in which the resources are requested. However, this approach is not always possible. If each process requests the required resources as a group, and avoids acquiring any of them until all of them are available, then deadlock can be prevented. This eliminates the "hold and wait" condition that is required for deadlock, and is also the idea behind *AND* synchronization. The Windows **WaitForMultipleObjects** function mentioned above allows a process to wait for *all* of the objects in a set, which is an implementation of *AND* synchronization.

Project Details

In this program, you will add two new system calls to Xinu. The first will be named **swait** for "simultaneous wait," and it has two **sid32** arguments identifying two semaphores. This will first disable interrupts, and then verify the semaphore arguments are reasonable (they both exist, are allocated, and are not the same as each other). If any of these tests fail, then **SYSERR** should be returned. Look at the code for the **wait** system call for the basic ideas.

Now the process executing **swait** will check to see if both semaphores have a count greater than zero. If this is the case, then the count for each semaphore is decremented, the interrupt mask is restored, and the process returns from the system call.

If the count (in the **scount** member of each semaphore's implementation) is not greater than zero, then the process will block in the **wait** state, effectively the same as if it had done a **wait** system call on the first of the two semaphores identified by the arguments that had a count less than 1. The code for the **wait** system call provides guidance on how to achieve this. When the process awakens – that is, on return from the **resched** invocation – the count of the semaphore on which the process was waiting should be incremented. Then the **swait** system call will repeat, starting from the tests described in the preceding paragraph.

The second system call you add will be named **ssignal**, for "simultaneous signal." Like **swait,** it has two **sid32** arguments identifying two semaphores. This is simpler than **swait**, but still needs to be done carefully. As with **swait**, the validity of the semaphore arguments must be assessed. Then the count for each of the semaphores must be incremented, and if appropriate, the process that has been waiting the longest on each semaphore must be moved from the **wait** state to the **ready** state. Use caution here: the tests and the wakeups of the processes should be done in such a manner that **resched** is not invoked until both semaphores – and both blocked processes, if appropriate – have been handled. You might look at the code for **signaln** for guidance, and consider the use of **resched_cntl**.

You must also handle the exceptional case of a process being blocked as a result of calling **swait**, and then one of the semaphores on which it is waiting is deleted (using the **semdelete**

system call). Also consider what should happen if the **semreset** system call is used on one of the semaphores on which a process is waiting as a result of **swait**. Provide ample documentation and justification for the actions taken in these cases.

The Provided Code

The standard class distribution of Xinu (in file **/home/phuang/csci8530/xinu-sync.tar.gz**) is a compressed tar file containing the version of Xinu with which you are to start development of your solution to this project. It includes template versions of the code for the **swait** and **ssignal** system calls, and a new shell command named "**dine**" – a relatively simple solution to the dining philosopher's problem using **swait** and **ssignal**. Of course, that program won't work correctly until you actually implement those system calls. It should be relatively easy to understand the source code for **dine** (which is naturally in the file **shell/xsh_dine.c**), but feel free to modify it as you wish to test your work.

Evaluations

Although the solution to the dining philosopher's problem can be used to illustrate the use of the new system calls, it isn't an aggressive test of their correctness. For example, it doesn't verify that improper arguments to the calls are identified. You should develop additional tests to verify that your solution meets all requirements. Although these tests are not to be submitted for evaluation, and testing can never show the complete absence of errors, not performing any tests is likely to be inappropriately optimistic! You should be aware that the instructor's evaluation of your work will involve more than running the dining philosopher's code.

Submission

To submit your work, create a directory on Loki named **/home/myusername/csci8530-prog2-s20**, replacing **myusername** with your username. For example, if your username was **phuang**, then the directory you create should be named **/home/phuang/csci8530-prog2-s20**. Put the source code for **swait.c** and **ssignal.c** in that directory. It is also likely that you will have modified other files; if you do, make certain to include them in the submission directory. Include comments in your code that explain what's happening. That is, an individual reading your code should be able to understand its purpose and the procedure used to achieve that purpose. Turn off, disable, or remove any debugging code you may have used before submitting your work.

Make certain you include – in each file – a comment near the top of the file that includes the name or names of each author of the work and also **snapshot the results** from the console after porting the codes to the board. If there are multiple authors, submit the joint work, only once, through **a single submission directory**. For example, if there are three members in your group, only ONE of these individuals should have a submission directory. Again, make certain there is nothing other than the modified files in the submission directory (except as noted in the next paragraph).

By default, each of the modified files (with the suffix **.c**) will be copied to the system directory in an unmodified version of the Xinu system created using the standard distribution (from file

`/home/phuang/csci8530/xinu-sync.tar.gz`). If this is *not* appropriate for your submission, be certain to include explicit instructions as to where the submitted files are to be placed in the Xinu directory hierarchy; the inclusion of a **README** file would be appropriate in this case.

As always, please contact the instructor if you have questions, and periodically check the class website for any additions or corrections to this assignment.