Project 2: Synchronization

The goal of this assignment is to implement semaphores as a kernel service, and to use those semaphores to implement locks and condition variables as user-level routines.

Basics

To implement semaphores in the kernel, you'll need to modify the process manager (servers/pm), adding system calls and code to create, use, and destroy semaphores. These calls will be accessible to user programs via system calls, so you'll also need to implement several library functions to call the process manager (don't worry—there are plenty of examples of how to do this). Once you have semaphores working, you'll need to implement locks and condition variables using semaphores; this code will run entirely at user level.

Details

Your code will run in two places: the process manager server (servers/pm) and user-level libraries. We'll cover system calls separately.

Process Manager Server

You'll have to modify the process manager server to handle the system calls described below. To do this, add a separate file to the code in servers/pm that contains the code for semaphores. You'll need to write routines to handle five operations: allocating a semaphore, getting values for a semaphore, the up() and down() calls on a given semaphore, and freeing a semaphore. Semaphores are identified by a positive (non-zero) integer that's provided when the semaphore is created; if the call to create a semaphore is passed the value 0, the kernel chooses an identifier for the semaphore. When the user is done with a semaphore, the code must explicitly release the semaphore, allowing the process manager to reuse it.

The process manager only has to handle up to 100 semaphores at any given time, so you can simply create an array that holds the semaphores the kernel may allocate to use. However, semaphores must be reused once they're freed—there's no limit to the number of times a semaphore may be allocated and then freed. Semaphores are identified by a positive (non-zero) integer, and no two active semaphores may have the same identifier. However, identifiers can be reused after a semaphore is released.

Semaphores behave the way we've discussed in the course. Processes can do up() and down() on a semaphore, and must sleep if the value of the semaphore is negative. When up() is called on a semaphore, a single process is awakened; processes that wait on a semaphore should be awakened first-come, first-served.

System Calls

Your system must support five system calls. All semaphore system calls except seminit() generate the EINVAL error if there's no active (allocated) semaphore with the identifier passed. Error code values can be found in sys/errno.h. System calls themselves are defined in lib/libc/sys-minix. You can use these short C routines as a model for your own system calls, though you need not include them in libc if you don't want to. Instead, you could make them part of semaphore.c if you want. In particular, look at priority.c for examples of how to make a system call to the process manager.

- int seminit (int sem, int value)
  This call initializes a new semaphore. The semaphore is identified by the integer passed in sem; if sem is 0, the kernel chooses an identifier. Note that an identifier can be any positive integer between 1 and 2^{31}; it need not be in the range 1–100. The call returns the identifier for the semaphore—either the value passed or the one the kernel chose—or 0 if an error occurred. The initial value for the semaphore is passed in value. On an error, the semaphore is not initialized. Error conditions (with error codes) include:

  EAGAIN

No free semaphores in the kernel (there is a limit of 100 active semaphores in the kernel).

EINVAL
Semaphore identifier is negative.

EINVAL
Semaphore initial value is not in the range $-1000 \leq value \leq 1000$.

EEXIST
Semaphore identifier is already in use for an *active* semaphore (one that was freed with semfree() is not active).

•int semvalue (int sem)
This call returns the current value of the semaphore whose identifier is passed. If there are *n* processes waiting on the semaphore, the call should return -n. If the semaphore has a value of 0, the next process to call semdown() would wait, but no process is currently waiting. If an error occurs, return 0x8000000.

•int semup (int sem)
This call does UP on the semaphore whose identifier is passed. This call never blocks. If there's at least one process waiting on this semaphore, semup()causes one waiting process to be awakened. The call returns 1 if successful, 0 otherwise.
Note: while a semaphore can't be initialized outside the range $-1000 \leq value \leq 1000$, it may be incremented (or decremented) to a value outside this range, up to $\pm 10^6$. If the semup() call would result in a value above 10^6, return 0 and set the error to **EOVERFLOW**.

•int semdown (int sem)
This call does DOWN on the semaphore whose identifier is passed. If the semaphore value would go below zero, the call blocks until the value goes above zero again. The call returns 1 if successful, 0 otherwise.
Note: while a semaphore can't be initialized outside the range $-1000 \leq value \leq 1000$, it may be decremented (or incremented) to a value outside this range, up to $\pm 10^6$. If the semup() call would result in a value below -10^6, return 0 and set the error to **EOVERFLOW**.

•int semfree (int sem)
This call frees a semaphore that's currently allocated, making the slot in the kernel available for reuse. A semaphore may not be released if there are processes waiting on it; the process that wants to free the semaphore must call semup() if necessary to ensure that there are no waiting processes before freeing it. The call returns 1 if successful, 0 otherwise. Error codes include:

EBUSY
The semaphore has processes waiting on it.

In addition to the system calls, you'll need to create an include file, semaphore.h, that can be used by user-level programs that wants to use the semaphore-related system calls.

Locks and Condition Variables

While semaphores are implemented in the kernel, locks and condition variables can be built from semaphores in a user-level library. You need to implement locks and condition variables with Mesa semantics. This includes struct lock, struct cond, and the following calls:

- lock_init (struct lock *l)
- lock_acquire (struct lock *l)
- lock_release (struct lock *l)
- cond_init(struct cond *cnd, struct lock *l)
- cond_wait(struct cond *cnd)
- cond_signal (struct cond *cnd)

You should create a file lockcond.h that can be included by programs that want to use locks and condition variables, and a file lockcond.c that contains the code for locks and condition variables.

This part of the assignment should be straightforward, since the implementation for the lock and condition variable functions is similar to the one discussed in class.

Testing

Testing your code is an important part of computer science, and operating systems design is no exception—if anything, testing is *more* important for operating systems because of the consequences of making a mistake.

You're strongly encouraged to write programs to test both your kernel-level semaphore code and your user-level lock and condition code. If you write these programs (and you should), please turn them in. If your code doesn't work, you can get some credit back by showing that you tried to find bugs, but didn't find the ones that we found. If you don't test your code and it doesn't work, you'll receive relatively little credit for simply typing in a bunch of C code.

You may want to document your testing code with information about what kinds of things it tests. You should also make sure you test error conditions, especially for kernel code.

Additional Synchronization Problems

In addition to the kernel and user-level code above, we assigned two synchronization problems on Tuesday, May 8th (see below). These problems must be implemented using the kernel semaphore calls you've implemented. We're assigning them after the rest of the assignment to give you a chance to focus on kernel code, rather than first tackling the (perhaps easier) user-level synchronization problems.

Deliverables

You must turn in a compressed (gzipped) tar file of your project directory, including your design document. Please ensure that the directory contains no machine-generated files when you create the tar file; you might want to add a special target (usually called clean to your makefile to handle this task). In addition, include aREADME file (place it in the project directory that you tar up) to explain anything unusual to the teaching assistant. You should have at least three subdirectories in your project directory, one for the process manager, one for user-level library code, and one for the user-level synchronization problems. You may also want a directory for files you use for testing (highly recommended). **Do not submit object files, assembler files, or executables.** Every file in the submit directory that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade. Also, the names and UCSC accounts of all group members must be in each file you've created or modified.

Your design document should be called design.txt (if in plain text), or design.pdf (if in Adobe PDF) and should reside in the top-level project directory. Formats other than plain text or PDF are not acceptable; please convert other formats (Word, LaTeX, HTML) to PDF. Your design should describe the design of your assignment in enough detail such that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

Project Groups

As you already know, this project *must* be done in a group of 2–3 students; you've already been emailed the names of your group members. Only one of you should turn in the project online; the others should turn in a single file listing the names and UCSC email accounts of all project partners, and indicating which of them turned in the file.

In addition to turning in the group project, *each* project member must turn in a short (1–2 paragraph) summary of their contributions to the group effort. While we expect to give the same grade to every member of a group, we reserve the right to give differential grades if it's clear that group members aren't pulling their weight. This does *not* mean that everyone in the group has to contribute

identically, but it *does* mean that a group member who does little or no work risks getting a lower grade.

If a group decides to use grace days to turn in their project late, **each student on the project must use a grace day for every day the project is late (e. g., a three-student project turned in three days late requires nine grace days, three from each student).** Recall that late projects (those turned in after the due date, which may have been extended by using grace days) receive a grade of 1 point, regardless of quality.

Hints

- Go to section for help on how to approach this assignment.
- Do your design document first, after looking through the kernel code to see what you need to do.
- Finish your design by Tuesday, May 1st. Go over it with the professor or TA if you have questions or concerns.
- Be paranoid in your code: check parameters in the kernel, not only in the user-level software library.
- Get kernel-level code working first; the user-level code for locks and condition variables is straight-forward. Do the user-level synchronization problems *last*. It's really difficult to debug synchronization problems when semaphores don't work.
- Write simple user-level programs to test your semaphore implementations. Run them a lot—you may have Heisenbugs.
- **START EARLY!**

Synchronization Problems

Your goal for this sub-assignment is to code up solutions to these synchronization problems using the semaphores that you've already implemented. Your code should run as a set of independent processes that communicate *solely* using the semaphores that you've implemented and one or more shared files for shared variables.

While you could use locks and condition variables for these problems, your locks and condition variables run at user level, and are thus more difficult to share between processes. Instead, rely upon semaphores.

You should have a brief design document for each problem, listing the approach you're using and pseudocode for each type of participant. You don't need to go into detail on semaphores; that's for your regular design document.

For each assignment, you'll need to create code for individual threads corresponding to the multiple actors. The code should be executable as a regular program, and take as many arguments as needed—typically, one argument per semaphore with an ID for the semaphore. You should write a separate program that sets things up or cleans up from the set of processes for a problem; this includes creating and freeing semaphores as well as setting up any files for shared variables (storing values in one or more files is probably the best way to do shared variables in MINIX).

Problem 1: No Starving Writers

In class, we discussed the readers/writers synchronization problem. We noted that a writer might starve if a continuous stream of readers kept arriving, never releasing the system to the writer.

Your goal for this problem is to write the code for a reader and for a writer for the readers/writers problem so that writers cannot starve. In particular, a writer should go before all readers that arrive after the writer arrives. The system must otherwise work the same way it does currently. Hopefully, your semaphores keep waiting processes in a FIFO (simple FCFS queue), since that's essential for non-starvation.

In this problem, as with all others, you should use a random number between 20000–200000 to determine how many microseconds to sleep while in each state (use the usleep() system call), and you should have each actor (reader or writer) print a message as they enter each state (waiting or using the file).

Problem 2: Raising Kids on an Alien World

For this problem, we return to the world of three genders. Every resident of the planet has a (single) gender, and they're not monogamous. However, they do triple-bond to procreate, and the bond lasts until the offspring is born. You should write code for a single resident of the planet that takes an argument whose value is 1, 2, or 3 (the gender of the resident). Residents meet in a shared location and wait patiently in line (asleep) until there's one of each gender. When there is, the gender 3 resident determines how long it will take to raise a child (random number between 20000–200000 microseconds) and notifies the other two. They all sleep for that period of time, awaken, and then spend a random amount of time (1000–400000 microseconds) waiting before rejoining the lines to reproduce again. Since this is a very poor planet, there are only two nurseries, so only two child-raising groups may be active at a time.

You may assume that there is at least one resident of each gender in the system.

Extra credit

For 10 points of extra credit, implement the data structures for semaphores so that the process server supports an essentially unlimited number, and so that access time for the semaphore structure is constant (i. e., use a dynamically-sizable hash table for semaphores).

Extra credit points will only be given to programs with at least 85 points of regular credit—spend your time getting the basic assignment working first, and only then work on the extra credit.