CS39002: Operating Systems Lab
Spring 2013

Assignment 5
Due: April 1, 2013, 1 pm
(Really due on April 1!! Don't come and tell us "April Fool" on that day!! ☺)

Consider a synchronization mechanism called ***ticket-and-eventcount*** which works as follows. Assume that each resource in the system has a unique string name for the resource known a-priori (for ex. PRINTER etc.). The OS internally maintains a data structure for each resource X with three fields: an integer variable *next_ticket* initialized to 1, an integer variable *eventcount* that is initialized to 1, and a queue *Q* of 2-tuples <*p*, *n*> where *p* is a process id, and *n* is an integer value. Three operations are supported on each resource X:

1. *ticket(X)**:*** This function returns the current value of *next_ticket* and increments it by 1 (so the first process that calls *ticket(X)* gets 1, the second calling process gets 2 and so on).
2. *await(X, n)*: This function returns with 0 if the *eventcount* variable associated with X has value *n*. If the value is less than *n*, an entry <*p*, *n*> is added to the queue *Q* for X, where *p* is the process id of the process, and the process *p* goes to sleep.
3. *advance(X)*: This function increments the *eventcount* variable for X by 1. If the new value after increment is *n*, then the OS checks if there is any entry in *Q* for X of the form <*p*, *n*>. If present, the process *p* is woken up. The function returns 0 on success.

For each of the functions, the function returns -1 if there is an error (for ex., passing a device name that does not exist).

In this assignment, you will write an OS process that will support this mechanism for user processes to use for their synchronization needs.

Each application that wants access to a resource X takes a ticket for the resource by calling *ticket(X)*. It then waits for the *eventcount* variable for X to become equal to the ticket value it got. For ex., if the *ticket(PRINTER1)* returned 5 to a process, the process will wait till the *eventcount* for PRINTER1 is 5. The process then executes (simulate execution time of a process by calling *sleep(2)* and then printing the process id and resource name), and increments the *eventcount* variable for that resource by 1 by calling *advance(X)*. The next process that is waiting for that resource, if any, then executes. So a typical user process using a resource named DISK will look like (no error checking shown)

```
val = ticket(DISK);
await(X, val);
// use resource DISK here
advance(DISK);
```

The OS Process

The OS is simulated by a separate process. The OS process is the first process to be started, and it creates and initializes all necessary shared memories and semaphores.

For this assignment, our system has three resources named DISK, PRINTER, and PORT. Each resource name is internally mapped by the OS to a unique integer id. For this assignment, DISK has id 1, PRINTER has id 2, and PORT has id 3. The OS process first creates and initializes a shared memory block named DEVICES that is an array mapping from resource names to the corresponding integer ids.

The OS process then creates two other shared memory blocks named COMMAND and RESULT. Basically, any user process will ask the OS process to do something by placing a command along with its parameters (in a specific format) in the COMMAND array (a circular array of commands). The OS will return the result (in a specific format again) of that command in the RESULT array (a circular array). The user process, on placing the command in the COMMAND array, will wait (go to sleep) for the result to come in the RESULT array. Thus, the OS process acts as the single consumer for the COMMAND array, and the single producer for the RESULT array. The user processes act as the producers of the COMMAND array and the consumers of the RESULT array. The COMMAND and the RESULT array will be protected by two sets of semaphores {L, M, N} (you should not need more than 3 semaphores to control a circular queue) and {X, Y, Z} respectively. The names of these semaphores are once again known to every process for use in *ftok* calls (see the names you should use at the end). Each entry of the RESULT array will have an additional integer to indicate if it is empty or not.

Size of the COMMAND and RESULT array are each 10 (i.e., 10 commands or 10 results). This should be #defined at the top of your program.

The structure of each command has the following form - an integer for the process id of the user process, an integer id for the command (well-known and fixed a-priori), an integer for the device id (again, known a-priori), and an integer parameter. The last integer parameter is used only for the *await(X, n)* call to pass the parameter *n*. The three commands we support have the following ids: *ticket* (id = 1), *await* (id = 2), *advance* (id = 3). Thus, if a process with process id 6534 makes the *ticket(PRINTER)* call, the command that is placed in the command array is <6534, 1, 2, _>, where '_' can be any arbitrary value as it will not be used for the *ticket( )* call.

Similarly, a result has the following form - an integer for the process id of the requesting process, and a single integer result. We will assume that a user process can issue at most one command to the OS at one time, it will issue a next command only after receiving the result of the previous one.

Your OS process should catch the SIGINT signal (sent when Ctrl+C is pressed to terminate it) and the signal handler should delete all shared memories and semaphores before the OS process terminates.

Placing/Processing Commands and Results

Your application will call *ticket(X)*/*await(X,n)*/*advance(X)* as before, where X is a string resource name. All these calls will now be implemented as functions in a static library call. This call will first map the DEVICES array using *ftok/shmget/shmat*, and look up the id corresponding to the

resource X. It will then map the COMMAND and RESULT array using *ftok/shmget/shmat* (and the semaphores using *ftok/semget*), prepare an appropriate command structure in the format specified for the command, and place it in the COMMAND array. As usual, it will use the semaphores {L, M, N} to control access to the array. It will then go to sleep (look up the *pause( )* call) waiting for the result to come in the result array.

The OS process will wait on the semaphores on the command array, and if there is a command, it will pick up the command from the head of the queue, process it, and write a result back in the RESULT array. The OS process will write in the first free entry of the RESULT array, and mark that entry as not-empty. The OS process will then send a SIGUSR1 signal (see the *kill( )* call) to the user process that put the command to wake it up. All information regarding the resources (like the current *eventcount* value, *Q* contents etc.) is maintained as local variables of the OS process, and are not directly accessible to any other process in any way. As usual, it will use the semaphores {X, Y, Z} to control access to the array. The OS process will behave as follows for the different commands:

1. For *ticket(X)*, it will write the integer return value (or -1 on error) in the RESULT array in the format described earlier, and then send a SIGUSR1 signal to the user process.
2. For *await(X, n)*, if the value of the *eventcount* variable for X is *n*, it will write 0 (or -1 on error) in the RESULT array. If the value is not 0, it will insert <*p,n*> in the *Q* array for X, where *p* is the process id of the user process that put the command, and write 1 (or -1 on error) in the RESULT array. In both cases, it will then send a SIGUSR1 signal to the process.
3. For *advance(X)*, it will first increment the *eventcount* variable for X. It will then scan the *Q* array for X to see if there is any process waiting for the new value of the *eventcount* variable. If yes, it will send a SIGUSR2 signal to that process. It will then write 0 (or -1 on error) in the RESULT array and send a SIGUSR1 signal to the user process.

A user process waiting for a result, on being woken up by a SIGUSR1 signal by the OS, will scan the entire RESULT array to pick up its result, and mark that entry empty. When it gets the result back, it takes the following action depending on the result:

1. For *ticket(X) a*nd *advance(X)* calls, it sends the result (the integer value returned in the result) back to the original call. For *ticket(X)* this is the integer value returned from the functions, or -1 on error. For *advance(X)*, this can be 0 for success or -1 for error.
2. For *await( )* call, if the result is 0 or -1, it will return the result back to the original call. If the result is 1 (meaning the process is to block), it will block again using a *pause( )* call. It will get unblocked on receiving a SIGUSR2 signal later, at which time it should return 0 to the original call.

In all cases, it also detaches all shared memory it attached with before returning. As usual, it will use the semaphores {X, Y, Z} to control access to the array.

Note that the user process knows nothing about all these things going on in the background, it just waits on the *ticket(X)* call the same as if it has made the call directly to your real OS. Once the library is written, the user process need not have anything to do with semaphores etc., that detail is hidden in the library. The user process will just need to link to the library.

Look up the *ar* command to build a static library under Linux. You can link with the library using the -l option of cc.

The following lines show calls to generate the keys for the shared memory/semaphores to be used:

- *ftok(".", 'D')* for shared memory DEVICES
- *ftok(".", 'C')* for COMMAND
- *ftok(".", 'R')* for RESULTS
- *ftok(".", 'L')* for semaphore L
- *ftok(".", 'M')* for semaphore M
- *ftok(".", 'N')* for semaphore N
- Similarly for semaphores X, Y, Z

You must follow the following specifications:

1. Have 3 resources named DISK, PRINTER, and PORT
2. All your *ftok()* calls must have "." as the first parameter
3. The static library must be named "libOS.a"

Submit the following files: *os.c* for the OS process, *ticket.c* for the library functions, and *user*.c for a sample user file that calls all the functions. We will test with our user programs, so make sure to follow specifications exactly.