ECE 469 Spring 2020 Laboratory #2

Changelog

2/7/2020: Added sample output. Jump

Introduction

The purpose of this lab is to become familiar with the problems of process synchronization and their solutions using OS-provided synchronization primitives.

In particular, you will solve the classic producer-consumer problem using the following different combinations of synchronization primitives:

- locks (binary semaphores)
- locks and condition variables

In addition to solving the producer-consumer problem, you will also implement one of the synchronization primitives inside the OS. In particular, for this lab, we will provide you with the implementation code for semaphores and locks in synch.c and synch.h, and you need to write the code to implement condition variables in synch.c and synch.h.

It is highly recommended that you check the return value of every function to ensure that it succeeded. It is very difficult to debug synchonization primitives, but you can help yourself substantially with proper print statements and return value tests.

In addition, we have provided you with several new Makefiles this week. You will need to understand what these Makefiles do in order to modify them for your assignment.
To be in synch with class lecture, this lab will be done in two parts. There are 5 questions in total. You are required to answer questions 1,2 during the first week of the lab. You shall work on questions 3,4,5 during the next two weeks.

Background and Review

Material Related to Synchronization Programming

An introduction to programming with threads is an excellent tutorial on the subject. You should find it useful additional reading. Note that DLXOS does not support threads. However, it does support shared memory, which makes the differences between thread programming and process programming trivial.

Shared Memory

Synchronization is only necessary when two processes or threads need to share information. Threads implement this information sharing process natively because individual threads share global address space. Therefore, any information that needs to transfer from one thread to another simply needs be put into a global variable, and other threads can read from it.

Since DLXOS does not have thread support, a new mechanism has been introduced to enable two processes to communicate. This mechanism involves allowing two or more processes to share part of their address space through two functions:

1. **request a handle to a chunk of shared memory:** a user program that wishes to use shared memory calls a trap to signal the operating system. The trap then returns a handle that identifies a physical shared memory segment allocated for this call. If two processes wish to share the same memory segment, they must both have the same handle. Handles can be shared between processes in DLXOS through command-line arguments when one process forks another process. This means a practical way to have some processes share a memory segment is to have a single parent process request a handle to a shared memory segment, and then have the parent process pass that handle to all of its child processes during forking.
2. **request the address of a chunk of shared memory given its handle:** Since a process only sees and deals with its own virtual address space, before accessing a shared physical memory segment, we need to create a mapping between part of a process's address space and the shared physical memory segment. This is achieved via the following API:

   ```
   buffer *b = (buffer *)get_shared_memory_address_from_handle(handle);
   b->item1 = 5;
   ```

   Effectively, the above API maps the shared physical memory segment back into the address space of the calling process, so the process can access the shared memory in ordinary ways, as shown in the example second line above.

   More details of how this "mapping" stuff works will be explained in Lab 4 when we deal with virtual memory. For now, here is an intuitive explanation. When you compile two programs, userprog1 and userprog2, all the addresses in those programs are virtual, e.g., the address space always starts at address 0x0. However, if you run userprog1 and userprog2 at the same time, they cannot both start in the physical memory at 0x0, because they would interfere with each other. Therefore, the operating system must allocate a piece of distinct physical memory for each process, and then create a mapping between the each individual process's address space to the physical memory region allocated for it. So, address 0x0 of userprog1 may actually reside at 0xF2341 in the physical memory, and address 0x0 of userprog2 may actually reside at 0xDDE7. This concept of mapping is called "dynamic address translation" which is part of what "virtual memory" does.

As a general rule, any time a process accesses shared memory, it must first make sure that no other processes are modifying that memory. (Otherwise, a context switch can happen anywhere during process execution, and the outcome of that memory access operation becomes uncertain.) This can be accomplished via the synchronization primitives covered in this lab. **Be very wary of any access to a shared memory location that is not surrounded with some kind of synchronization primitive.**

In this lab, we will use shared memory to share semaphore handles, lock handles, condition variable handles, and circular buffers among processes.

Circular Buffers

The classic producer-consumer problem is generally implemented through the use of a shared circular buffer. A circular buffer has three components:

- head
- tail
- and buffer space to hold data.

The buffer space is usually an array, and the head and tail members are indices to that array. The head index is defined as the index of the first available slot in the buffer, and the tail index is defined as the index of the first full slot in the buffer. The term "circular" refers to the fact that when either the head or tail index is incremented beyond the end of the buffer array, it is reset to the first slot in the array.

Before a consumer process removes an item from the buffer, it must first check to see if the buffer is empty. The empty condition, as a result of the definition of the head and tail indices, is when the head and tail indices point to the same location in the buffer:

```
if (head == tail) {
  // buffer is empty
} else {
  // buffer is not empty
}
```

Before a producer process can put an item into the buffer, it must first check to see if the buffer is full. The full condition is when the head index points at the buffer slot immediately behind the tail index. In other words, the buffer is full when the addition of one more item would put head == tail, therefore making the buffer appear empty:

```
if ((head + 1) % BUFFER_SIZE == tail) {
  // buffer is full
} else {
  // buffer is not full
}
```

For more information on circular buffers, a simple Google search for "circular buffer" is very helpful.

Changes to DLXOS Source

In order to support this week's lab, several things have been changed from last week. You will have to copy this new version of DLXOS (version lab2.tar.gz) for these changes to be visible to you. The tar file of this new version of DLXOS is located at `~ee469/labs_2020/Labs/lab2.tar.gz`

Directory Structure:

The following directories have been added:

```
apps/example/            example user code is here
             include/    application-specific include files are here
```

```
            makeprocs/  example code to spawn processes is here
            spawn_me/   example code of a spawned process is here
apps/q1/    put your answers for question 1 here
apps/q2/    put your user code for question 2 here
apps/q3/    put your user code for question 3 here
apps/q4/    put your user code for question 4 here
apps/q5/    put your user code for question 5 here
```

Shared Memory:

We have added shared memory support to DLXOS. User programs can request up to 32 chunks (a.k.a. "pages") of 64KB of memory that they can share with other processes. Any given process can only have up to 16 chunks of memory allocated to it at any given time. This support manifests itself in your programs as two functions:

- `uint32 shmget();`
  shmget is a trap which returns a unique handle to a 64KB page of shared memory. This handle can be passed to any processes that want to share this same memory space.
- `void *shmat(uint32 handle);`
  shmat is a trap which maps the shared memory page represented by "handle" to the calling process's memory space. It returns a pointer to the shared memory space, similar to how a call to malloc() in a standard C program would return an address of memory space.

The OS implementation of the shared memory API can be found in lib/share_memory.o. This file should be linked against the operating system when compiling. The user traps for the two functions listed above are found in os/usertraps.s with all the other user traps. NOTE: do NOT link lib/share_memory.o with your user programs. It should ONLY be linked with the operating system.

Details on the new API functions can be found here. You must read that document in order to know what functions are available and how they should be called.

Synchonization primitives

We have implemented semaphores and locks in synch.c and synch.h, as well as user traps for these functions in usertraps.s. To be able to use these traps in your programs, you must #include include/lab2-api.h. This file also declares the shared memory API functions.

Command-Line Arguments to User Programs

Since you will need a way to pass shared handles between multiple processes, we have added support in DLXOS to pass command-line arguments to user programs. All arguments listed on the command line after the name of the user program will be passed directly to the user program via the standard argc/argv method. If you are unfamiliar with argc/argv, a quick Google search will be helpful. For example, if I run the user program "makeprocs.dlx.obj" with this command:

```
$ dlxsim -x os.dlx.obj -a -u makeprocs.dlx.obj 3
```

then the command line argument "3" will be available in the main function of makeprocs.c. We support up to 10 command line arguments to any given process. You can also pass command line arguments when a program creates other programs via the process_create() function. Refer to the example user program apps/example/makeprocs/makeprocs.c for an example of how to pass command-line arguments through process_create().

Example User Program:

We have provided an example user program to help you understand how to structure your user programs and to show you how to use the new shared memory and semaphore API's. This example program consists of two separate processes. The first process, "makeprocs", forks multiple copies of the second process, "spawn_me". To run the example program, you can type "make run" from apps/example, apps/example/makeprocs, or apps/example/spawn_me.

The "makeprocs" program expects one command line argument: the number of spawn_me processes to create. It is the responsibility of makeprocs to allocate the shared memory handle and any semaphores or locks prior to creating any spawn_me processes, and then pass these handles to spawn_me.

There are two methods of passing handles to the forked processes. The first, illustrated in makeprocs, is to convert them each to a string and pass each string as a separate command-line argument to spawn_me through process_create(). This method is the most straighforward, although it can become unweildy as the number of handles increases. The second method involves putting all the shared handles into the shared memory page, and then just passing the handle of the shared memory page to the forked processes. All child processes can then access the other handles by mapping the shared memory. This process is less straightforward, but it scales much more nicely as the number of shared handles increases.

After allocating the shared memory handle, makeprocs puts some information into the shared memory buffer to be read by spawn_me to ensure that it is working properly. It also creates a semaphore to indicate when all of its forked processes have completed. It is necessary for makeprocs to remain running until all other processes have completed in order to ensure that the operating system does not de-allocate the shared memory page. It then creates the spawn_me processes via process_create(), and finally waits on the procs_completed semaphore before exiting.

The spawn_me process maps the shared memory page, reads the data put there by makeprocs, and then signals the procs_completed semaphore before exiting.

Makefiles:

As you all discovered last week, it can be very time-consuming to have to keep typing all those commands to compile your programs. Therefore, now that you understand how to compile programs, we are giving you Makefiles this week to make things go faster.

The first Makefile we've provided is located in os/Makefile. This Makefile tells the "make" program how to compile the operating system and its libraries. Since all the intermediate object files tend to clutter up the source directory, the Makefile will first create a "work/" directory where it will store all the intermediate object files. Once os.dlx.obj is created, it will copy it to the bin/ directory. Also, once usertraps.o is created, it

will copy that file to the lib/ directory to be linked against user programs. A standard "make clean" will remove all the intermediate files and force the operating system to be rebuilt from scratch on the next call to "make".

We have also provided a special target, "make run", that you can type from the os/ directory, and it will cd into the apps/example directory and run "make run" there, which will run the user program for you. You are welcome to modify this target in any way that is convenient to you (add debugging flags, for instance).

A NOTE ON "MAKE RUN": the "make run" command ONLY cd's to the bin/ directory and starts the simulator with your user program. If either the OS executable or your user program's executable do not exist, the simulator will simply not print anything to the screen and never exit. This means you have to explicilty make the operating system AND a user program before "make run" will work correctly.

There are also a set of Makefiles provided for compiling your user programs. They have been designed such that you can simply copy them to any new application directories, and you'll only have to change a couple of lines in order to compile any new programs.

The Make system for user programs starts with apps/Makerules. This file contains almost the same things as the Makefile in os/Makefile, except it doesn't pass the _osinit function to the assembler as the bootloader. So, it will create the work/ directory and copy the user program executable to the bin/ directory in much the same way. This file is in the main apps directory because it is generic for all applications. It is called "Makerules" rather than "Makefile" to indicate that it should not be used as a standalone Makefile, but rather included in each application's Makefile.

In apps/example/Makefile, you will find a wrapper Makefile that cd's into each individual code directory for the example application (directories are makeprocs/ and spawn_me/), and runs make in those directories. It also defines the actual "make run" command that will run the example user program.

In apps/example/Makerules, there are some application-specific rules that are common among all sub-diectories of the example application. The following items are specified here:

- HDRS=usertraps.h: this is a list of all header files in the main include/ directory that are included in this user program
- FINALHDRS += ../include/spawn.h: this line lists all the application-specific header files that any files in this application #include. Note that you need the "+=" here because FINALHDRS may already exist from other Makerules files. Also note that the relative directory path "../include" actually refers to a starting location of one directory below this Makerules file. That may seem confusing at first, but just remember that a Makerules file is intended to be included in other Makefiles, and those other Makefiles in this case are one directory below the Makerules file.
- INCDIR += -I../include: this line adds the application-specific include directory to the list of compiler flags indicating where to search for include files. This explains why you need the "-I" part: that's the GCC flag indicating the include search path.
- top: default: this line sets the top-level target (the one that is built when you type "make"), and sets its dependency as "default", which is the top-level target in the generic Makerules file.
- run: this line defines the "make run" target for each of the application sub-directories, reducing the number of times you have to change directories to test your program.

In apps/example/makeprocs/Makefile, there are only a few lines. This Makefile contains only those things that are specific to the makeprocs program, namely the list of sources (SRCS), and the name of the executable file (EXEC). You'll need to change these when write new user applications. There is a similar Makefile in apps/example/spawn_me/Makefile that does the same for the spawn_me application.

Assignment

1. (10 Points) Read queue.c, queue.h, synch.h, synch.c and answer the following questions. Please write your answers in apps/q1/Step1.txt.
    - (2 points) How many semaphores are available in DLXOS?
    - (2 points) How many locks are available in DLXOS?
    - (2 points) How is a semaphore created in DLXOS?
    - (2 points) Explain how sem_signal and sem_wait work.
    - (2 points) What is a handle and how is it used to access semaphores in DLXOS?

2. (10 points) Using the above shared memory APIs, implement the following producer/consumer communication through a circular buffer using only locks to prevent race conditions and deadlocks. Note that you should still use a sempahore to signal the parent process that all child processes have completed. The sample code "makeprocs" will serve as an example on how to use semaphore to signal the parent process. The solution to this problem can be expressed as follows:

    The producer places characters from the string "Hello world" into a shared circular buffer one character at a time. If the buffer is full, the producer will wait until there is space available. The consumer pulls one character out of the shared circular buffer and prints it to the screen. If the buffer is empty, it must wait until there is something in the buffer to print.

    To make the solution simpler, you can assume there are the same number of consumers and producers. In other words, if each producer will put strlen("Hello World") characters into the buffer, then each consumer can remove exactly strlen("Hello World") characters out of the buffer. If we did not have this restriction, then it is more difficult for the consumer to know when to exit. This way, it can simply choose to exit when it has read strlen("Hello World") characters from the buffer.

    Your solution must follow the convention of the example code, where a "makeprocs" process creates the producers and consumers, and then waits until all producers and consumers have exited before it exits. makeprocs should have the following program invocation:
    `dlxsim -x os.dlx.obj -a -u makeprocs.dlx.obj <number of producers and consumers>`
    for example,
    `dlxsim -x os.dlx.obj -a -u makeprocs.dlx.obj 3`
    will create 3 producers and 3 consumers.

    Each time a producer puts a character into the buffer, it should print:

    ```
    Producer X inserted: H
    ```

    Each time a consumer takes a character out of the buffer, it should print:

```
Consumer Y removed: H
```

where X and Y are the process id's of the producer and consumer, respectively.

All your user code should be in apps/q2/. You should create the following subdirectories: apps/q2/include/, apps/q2/makeprocs, apps/q2/producer, and apps/q2/consumer. What goes in those directories should follow the convention of the apps/example directory.

3. (30 points) Implement condition variables in os/synch.c using the existing locks and queues. The user-level interface is already provided in os/usertraps.s. Look at the code for locks and semaphores to see how to implement the process waiting queue. As a hint, recognize that a condition variable is essentially a semaphore without an internal counter, so the code is almost the same. You will find markers in include/os/synch.h and os/synch.c where your code should go.

   All your code for this problem (excluding any extra test code you may write) should be in os/synch.c and include/os/synch.h.

4. (10 points) Implement the same producer/consumer problem using locks and condition variables. All code for this problem should go in the apps/q4 directory with the same structure as the previous producer/consumer solutions.

5. (40 points) You are given a hypothetical situation below. In the atmosphere on the planet Radeon, there are 3 primary chemical reactions that can occur:

   1) 2 H2O -> 2 H2 + O2 2) SO4 -> SO2 + O2 3) H2 + O2 + SO2 -> H2SO4

   The Radeon Atmospheric Sciences (RAS) department has asked us to help out with writing an atmosphere monitoring program. They require a program that
   a) Injects molecules of H2O (water) and SO4 (sulfate) into the atmosphere
   b) Monitors the formation of the O2 (oxygen) and H2SO4 (sulfuric acid) molecules
   c) Prints out messages at every stage

   Reaction 1) should happen whenever there are two H2O molecules available
   Reaction 2) should happen whenever there is one SO4 molecule available
   Reaction 3) should happen whenever there is one H2 molecule, one O2 molecule, SO2 molecule available.

   In order to solve this problem, you may think of each reaction as a separate process, and each molecule as a separate semaphore. Also, the "injection" routines which create the original H2O and SO4 molecules are each a separate process, for a total of 5 processes. When a given atom or molecule is created, you can signal the semaphore for that atom or molecule, and when you want to consume an atom or molecule, you should wait on the associated semaphore.

   When a H2O or SO4 molecule is injected, you should print a message indicating such. You should also print a message when the final O2 molecule and H2SO4 molecule are created.

Your program should have a makeprocs process that can be invoked like this:

```
$ dlxsim -x os.dlx.obj -a -u makeprocs.dlx.obj <number of H2O molecules> <number of SO4 molecules>
```

As in all other problems, makeprocs should not exit until all of the 5 other processes have finished. In order to support this, you must pass the expected number of "consumable" molecules to each of the "consumer" processes: i.e. the 3 processes representing the reactions. You will therefore have to compute this based on the reaction formulas for each reaction and the original number of H2O molecules and SO4 molecules.

You should only print the minimum number of messages necessary to describe the underlying reactions. You must print one message for each molecule created.
Refer to the following sample output (added 2/7/2020):
If the input is 3 H2O and 3 SO4 molecule the output can be as given below.
Creating 3 H2Os and 3 SO4s.
H2O injected into Radeon atmosphere, PID: 30
H2O injected into Radeon atmosphere, PID: 30
SO4 injected into Radeon atmosphere, PID: 29
SO4 injected into Radeon atmosphere, PID: 29
SO4 injected into Radeon atmosphere, PID: 29
2 H2O -> 2 H2 + O2 reacted, PID: 28
H2O injected into Radeon atmosphere, PID: 30
SO4 -> SO2 + O2 reacted, PID: 27
(1) H2 + O2 + SO2 -> H2SO4 reacted, PID: 26
SO4 -> SO2 + O2 reacted, PID: 27
(2) H2 + O2 + SO2 -> H2SO4 reacted, PID: 26
SO4 -> SO2 + O2 reacted, PID: 27
1 H2O's left over. 0 H2's left over. 2 O2's left over. 1 SO2's left over. 2 H2SO4's created.
(End of sample output. added 2/7/2020)

Use **ONLY** semaphores to implement the synchronization, for the following reason. To simplify your solution, you do not need to store a buffer of actual molecules as they are produced and consumed (i.e., no need to write code to simulate the actual reaction process); you only need to use the s emaphores to simulate the conditions for the reactions. Therefore, since you will not need to access a shared buffer, there is no need to use locks in the solution.

Put all your code for this problem in apps/q5. Your directory structure should be such that there is one directory for each distinct type of process, similar to the structure for the previous producer/consumer problems.

Turning in your solution

You should make sure that your `lab2/` directory contains all of the files needed to build your project source code. You should include a README file that explains:

- how to build your solution,

- anything unusual about your solution that the TA should know,
- and **a list of any external sources referenced while working on your solution**.

DO NOT INCLUDE OBJECT FILES in your submission! In other words, make sure you run "make clean" in all of your application directories, and in the os directory. Every file in the handing directory that could be generated automatically by the DLX compiler or assembler will result in a 5-point deduction from your over all project grade for this lab.

When you are ready to submit your solution, submit it via Blackboard in a zip/tar file. The name of zip file clearly should mention lab number and the group number. For eg :- For lab2 , it should be ee469lab2_g12.tar.gz where lab2 is lab number and g12 is the group number.

---

ee469@ecn.purdue.edu