ECE 469 Spring 2018 Laboratory #3

Due Sunday, March 4, 2018 at 23:59 EST

Introduction

This lab will be done in two parts similar to lab2, to be in synch with class lecture.The purpose of this lab is to become familiar with mailbox-style interprocess communication, and a lottery scheduling. In particular, you will accomplish the following:

- implement an interprocess mailbox API in DLXOS,
- test your mailbox API with the chemical reaction simulator from lab2,
- and implement a lottery scheduler in DLXOS.

As with lab2, it is highly recommended that you test the return value of EVERY function that you call to make sure it succeeds. In addition, it is extremely important to **take testing your code seriously** in this lab as there are many potential pitfalls where your code seems to work correctly in some cases, but does not in others. You can help yourself tremendously by thinking through various test cases and writing them down on a piece of paper. Then go through that list one by one and decide how to write code to test if your code successfully handles that test condition. **Please check the updated Lab Handout with details for the Process Scheduling**

Background and Review

Mailboxes

Recall from Lab 2 that processes were able to communicate via shared memory protected with synchronization primitives. Notice in this way of inter-process communication, accessing the shared data is easy; a process simply accesses the shared data no differently from accessing private data. However, properly using the synchronization primitives can be tricky; misuse of them can lead to race condition or deadlocks. An alternative mechanism for processes to communicate with each other that avoids the potential problems with using synchronization primitives is via external (to the process) "mailboxes" (supported by the OS), where some processes can send messages to, and other processes can receive messages from. As a generic interprocess communication

mechanism, a mailbox is oblivious of the specific data structures associated with the message being transmitted. It simply treats the message as a string of bytes (i.e. they are void *'s). While the upside of using mailboxes is that the programmer no longer deals with synchronization primitives, the downside is that the programmer has to explicitly "pack" and "unpack" data structures into the message buffers.

The basic mailbox functionality can be described by the following API:

Mailbox Function	**Description**	**Shared Memory/Synchronization Equivalent**
mbox_t mbox_create();	Reserves a mailbox for use, and returns a handle to that mailbox on success. Returns MBOX_FAIL on failure.	shmget() and sem_create()
int mbox_open(mbox_t handle);	Opens the given mailbox for reading and writing by the current process, in much the same way that one would open a file for reading and writing. Returns MBOX_FAIL on failure and MBOX_SUCCESS on success.	shmat()
int mbox_close(mbox_t handle)	Closes the given mailbox for a process. If there are no other processes that currently have this mailbox open, it should be de-allocated and returned to the pool of available mailboxes for the system. Returns MBOX_FAIL on	none

	failure and MBOX_SUCCESS on success.	
int mbox_send(mbox_t handle, int length, void *data)	Sends a message to the specified mailbox of length "length" from the address contained in "data." If the mailbox is full, wait until there is room. Returns MBOX_FAIL on failure and MBOX_SUCCESS on success. Note that this should fail if the calling process does not have the mailbox open.	writing to shared memory, sem_signal()
int mbox_recv(mbox_t handle, int maxlength, void *data)	Receives a message from the specified mailbox. The message is copied to the address contained in "data." If the message length is greater than maxlength an error occurs. If the mailbox is empty, wait until there is a message to be received. Returns MBOX_FAIL on failure and number of bytes written into message on success. Note that this should fail if the calling process does not have the mailbox open.	reading from shared memory, sem_wait()

All mailbox functions should be written such that they are free of synchronization problems like race conditions and deadlocks. In other words, user programs do not need to use locks, semaphores, or condition variables to send and receive messages. The handling of the locks, semaphores, or condition variables should be entirely internal to the mbox functions.

Process Scheduling in DLXOS

Process scheduling in DLXOS is currently handled according to the following algorithm:

1. Start timer:
   On initialization, start a recurring timer to go off every 0.1 seconds. This is a hardware timer, so it operates independently of the CPU.

2. Timer interrupt:
   Each time the timer goes off, it triggers an interrupt that runs the _intrhandler routine in dlxos.s

3. Save registers:
   _intrhandler pushes all relevant registers onto the system stack for the currently running process. It uses the _currentPCB global variable to identify the PCB for the current process. currentPCB is defined and maintained in process.c.

4. Handle interrupt:
   After all the registers have been saved, _intrhandler calls the dointerrupt() function in traps.c.

5. Call ProcessSchedule():
   dointerrupt() calls ProcessSchedule() in process.c each time this timer goes off.

6. Move currentPCB to end of runQueue:
   ProcessSchedule() removes the currentPCB from the head of the runQueue, and moves it to the back. It then changes the currentPCB pointer to point at the PCB on the front of the runQueue. **Note that simply changing the currentPCB pointer does not start the new process running.**

7. Return to handler:
   ProcessSchedule() returns to dointerrupt().

8. Return from handler:

dointerrupt() calls the _intrreturn function in dlxos.s.

9. Pop registers off new stack:
   _intrreturn uses the global _currentPCB variable to find the system stack, and pops all the registers back off that stack. Since currentPCB changed in ProcessSchedule(), this popping actually restores the registers with the values of the new process.

10. Resume process execution:
    _intrreturn loads r31 with the last run address of _currentPCB, and then executes the reti instruction which will load r31 into the program counter and resume execution of the new process.

It is important to note that the new PCB is actually switched in by the _intrreturn function. This means that ProcessSchedule, which is the only function that can change currentPCB, MUST be called ONLY from a trap, so that the _intrreturn function will run when it is finished.

With that said, ProcessSchedule is not only called from the timer trap. It is also called anytime a process goes to sleep (immediately after ProcessSuspend, which is also only called from a trap). Therefore, ProcessSchedule is not always called on exact integer multiples of your timer period.

Consequently, another confusing concept is that since ProcessSuspend doesn't actually change currentPCB (this is left up to ProcessSchedule), then any time a process goes to sleep it will still be the currentPCB when ProcessSchedule starts even though it is not on the runQueue. To prevent trying to remove it from the runQueue in ProcessSchedule (which would actually remove it from the waitQueue), you should check `if (currentPcb->flags & PROCESS_STATUS_RUNNABLE)`. That will be true if the currentPCB is on the runQueue, and false otherwise.

Your task this week will be to modify process.c to support lottery based scheduling rather than the round-robin scheduling algorithm laid out above. **You can define Macros such as RR_SCHED, LT_SCHED to provide support for Lottery Scheduler along with the Round-Robin Scheduler.We can set the macro during compile time to bring in support for the particular Scheduler.**

Lottery Based Scheduling

The basic DLXOS right now only has round-robin preemptive scheduling. In this lab, we will add a lottery scheduler into DLXOS.

The Basics

In this lab assignment you are to implement a lottery scheduler. A lottery scheduler assigns each process some number of tickets, then randomly draws a ticket among those allocated to ready processes to decide which process to run. That process is allowed to run for a set *time quantum*, after which it is interrupted by a timer interrupt and the process is repeated.

The number of tickets assigned to each process determines both the likelihood that it will run at each scheduling decision as well as the relative amount of time that it will get to execute. Processes that are more likely to get chosen each time will get chosen more often, and thus will get more CPU time.This is a probabilistic way of interpreting the lottery scheduler.

Note that we don't have to dole out actual tickets. We can just keep a count of how many tickets each process has {n1, n2, ..., nm}(assuming m processes),each process can maintain the no of tickets in their PCB. how many tickets have been doled out altogether (N), and the fraction of the total each process has {f1, f2, ..., fm} = {n1/N, n2/N, ..., nm/N}. Then, to decide which process we should run at each scheduling decision, we can compute a random number p between 0 and 1. If 0 < p < f1 then we run process 1. Else if f1 < p < f1+f2, we run process 2. Else if f1+f2 < p < f1+f2+f3, we run process 3. Etc.

One goal of best-effort scheduling is to give I/O-bound processes both faster service and a larger percentage of the CPU when they are ready to run. Both of these things lead to better responsiveness, which is one subjective measure of computer performance for interactive applications. CPU-bound processes, on the other hand, can get by with slower service and a relatively lower percentage of the CPU when there are I/O-bound processes that want to run. Of course, CPU-bound processes need lots of CPU, but they can get most of it when there are no ready I/O-bound processes. One fairly easy way to accomplish this in a lottery scheduler is to give I/O-bound processes more tickets - when they are ready they will get service relatively fast, and they will get relatively more CPU than other non-I/O-bound processes.

The key question is how to determine which processes are I/O-bound and which are CPU-bound. One way to do this is to look at whether or not processes block before using up their time quantum. Processes that block before using up their time quantum are doing I/O and are therefore more I/O-bound than those that do not. On the other hand, processes that do not block before using up a time quantum are more CPU-bound than those that do. So, one way to do this is to start with every process with some specified number of tickets. If a process blocks before using up its time quantum, give it another ticket (up to some set maximum, say 19). If it does not block before using up its time quantum, take a

ticket away (down to some set minimum, say 1). In this way, processes that tend to do relatively little processing after each I/O completes will have relatively high numbers of tickets and processes that tend to run for a long time will have relatively low numbers of tickets. Those that are in the middle will have medium numbers of tickets.

NOTE: DLXOS has no support for actual IO. As we know, a process doing IO will block before using up the full Quantum of CPU assigned for itself. It is going to Sleep. Another case where a blocking process can be simulated is SemWait(). A process calling SemWait() will have all probability of Sleeping, based on the Semaphore count being less than or equal to 0. Our lottery scheduler will be tested using SemWait() call.In any case the processes before going to sleep are going to invoke the scheduler to do a context switch. How can we make sure that such processes which block before using up the CPU quantum get higher priority?

HINT: Case1: Process doing an IO, Case2: Process actually sleeping due to SemWait(). There is a similarity between them. "They are going to call Scheduler as part of Sleeping." Check the _ProcessSleep assembly routine. So we can implement the dynamic ticket assignment in the scheduler itself based on whether the process has used up full quantum or not.

This system has several important parameters: time quantum, minimum and maximum numbers of tickets, speed at which tickets are given and taken away, etc.

The Details

In this project, you will modify the scheduler for DLXOS.

The prototype of create_process has been changed to facilitate passing in parameters p_nice and p_info . The new prototype is

```
void process_create(int p_nice, int p_info, char * args...);
```

For lottery scheduling, p_nice corresponds to the number of tickets given to the process. It can range from 1 to 19. For all processes, specifying a non-positive value will be treated as 1; similarly, specifying a value greater than 19 will be treated as 19. Each time the scheduler is called, it should randomly select a ticket (by number) and run the process holding that ticket. Clearly, the random number must be between 0 and *nTickets*-1, where *nTickets* is the sum of all the outstanding tickets. You may use the random() call to generate random numbers and the srandom() call to initialize the random number generator (these calls

function the same way they do in Unix).

Sleep, Yield, and the Idle process

In lab 2, when a process went to sleep, it could only be awoken by another process calling ProcessWakeup on its behalf. Therefore, if the operating system detected that there were no processes on the runQueue, it was safe to exit. If there were still processes on the waitQueue, it printed an error and exited because no one was ever going to wake up those sleeping processes.

This week, we are going to extend the "sleep" functionality by enabling processes to specify how long they want to sleep before being woken up. To support this, you must modify ProcessSchedule to check the waitQueue for any processes that need to be woken up according to the time they went to sleep and how long they wanted to sleep.

This means, of course, that the operating system can no longer simply check for no processes in the runQueues when deciding to exit. It must instead check that there are no processes on the runQueues *AND* that there are no processes that will be automatically woken up at a given time in the future.

Special Case This sounds easy enough, but now we have a problem: what PCB is the operating system going to switch to in ProcessSchedule if there are no PCB's on any runQueues? To solve this problem, we introduce the concept of an "idle" process: i.e. a process that is just an infinite while loop that gets run whenever there isn't anything else to run, but we can't exit yet. This function looks like this:

```
void ProcessIdle() {
  while(1);
}
```

Your first instinct may therefore be to write something like this at the top of ProcessSchedule():

```
void ProcessSchedule() {
  if (there are no processes on the run queue) {
    if (there are autowake processes on the wait queue) {
      ProcessIdle();
    }
  }
}
```

This won't work, however, because ProcessIdle() never returns, so you'll never schedule any new processes. What we really need is to get ProcessIdle() running with its own PCB that we can switch to when there is nothing else to run.

So, how can we run a function in the OS as a process with its own PCB (i.e. a kernel process)? ProcessFork() is already setup to do this for us. Here is the prototype for ProcessFork():

```
typedef void (*VoidFunc)();
int ProcessFork (VoidFunc func, uint32 param, int pnice, int pinfo,char *name, int isUser);
```

The first argument, `func`, is a function pointer. If we set the `isUser` parameter to 0, then ProcessFork will run the function pointed to by `func` in its own PCB. You will have to write some special code in ProcessFork to check if you are creating the idle process (i.e. `if (func == ProcessIdle)`)

You should also keep a global pointer to your idle PCB so that you can tell if a given pcb is the idle process.

Along the lines of sleep() is yield(). When a process calls yield, it is telling the operating system "I don't want to go to sleep, but I don't have anything to do right now. Let other processes run their course and then run me again." In other words, a call to yield() simply triggers an immediate call to ProcessSchedule, which moves the current process to the back of the runQueue. There need not be any change to the no of tickets.

Mutual Exclusion in Kernel Functions

You probably have noticed that the functions in the current version process.c such as like ProcessSchedule functions currently use Enable/Disable interrupts, to implement mutual exclusion, i.e. in performing certain tasks without interruption. When you modify process.c to implement the new scheduling algorithm, you should continue to use Enable/Disable interrupts to implement mutual exclusion.

In other words, you should not try to use synchronization primitives in the core OS functions in DLXOS. The explanation is a little involved. First, DLXOS can be viewed a "single process" kernel, unlike some of the OSes which consist of multiple kernel processes, which may share kernel data structures (e.g. by using locks). In other words, the single kernel process of DLXOS does not need to share any kernel data structure with some other kernel processes. Second, conceptually, by its very nature, ProcessSchedule is a special function in the kernel that should not be waiting (i.e. put to sleep) on any locks/semaphores; whenever it is invoked, it is supposed to schedule another job to run. Using locks et al. inside it and other functions in process.c creates can potentially cause it to wait.

Note that system calls are different from kernel functions such as

ProcessSchedule. System calls such as mailbox_send() and mailbox_recv() are run on behalf of user programs, in particular to allow user processes to acquire system resources. When resources are not available, it is only natural to put the user processes to sleep, and synchronization primitives are ideal for such usages.

Another example where we need to use locks in system calls will happen in Lab5, when we implement a file system for DLXOS. Since multiple concurrently running processes can potentially try to modify the same file, some locking is needed inside the implementation of the system calls for the file system.

In summary, use synchronization primitives for user-trap-only functions (i.e. system calls like mailboxes) in the DLXOS, and use Disable/Enable Interrupts for all actual kernel functionalities.

Changes to DLXOS Source

Clock Subsystem

A clock subsystem was written for DLXOS this week to better support process timing characteristics. The operation of the clock is rather simple:

1. **ClkStart()**: starts the hardware timer firing with a given period.
2. **ClkInterrupt()**: called from traps.c's dointerrupt function each time the hardware timer interrupt goes off. It increments an internal counter. This function returns 1 if enough counts have passed to warrant signalling ProcessSchedule again.
3. **ClkResetProcess()**: resets the process timer count to zero.

Note that the clock's internal counter is incremented more frequently than ProcessSchedule gets called. This allows for more fine-grained time measurements.

The concept of the number of microseconds per clock tick is commonly called a "jiffy". If the timer interrupt has fired 5 times, then 5 "jiffies" have passed. You can compute the time that has passed by multiplying the number of jiffies by the clock resolution.

IMPORTANT: you must do all your computation in your code using jiffies. Jiffies are integers, therefore the computation speed is significantly faster.

Be aware that a jiffy is different that a process quantum. A jiffy is how often the system clock fires. A process quantum is how often ProcessSchedule is run.

Therefore, a process quantum is made up of several jiffies.

pnice and pinfo

We have modified ProcessFork and the process_create() trap to accept two new arguments: pnice and pinfo. The purpose of pnice is explained in the section above on lottery scheduling. The pinfo variable can be set to either 0 or 1, and it indicates to ProcessSchedule that you want it to print the total runtime of a particular process (in jiffies) whenever that process is context switched out.

```
void process_create(char *exec_name, int pnice, int pinfo, ...);  //trap 0x432
int ProcessFork (VoidFunc func, uint32 param, int pnice, int pinfo,char *name, int isUser);
```

Note that the command-line process creation (i.e. how you start makeprocs) does not have these variables. Only the process_create trap and ProcessFork have these variables. This was done intentionally to keep the command-line argc and argv simpler.

New User Traps

All new user traps have been written in os/usertraps.s, and their prototypes are in include/usertraps.h. You will not need to write any new traps for this assignment.

Printf and printf

We have fixed some of the printf problems. First, you can now print strings with %s from user processes. Also, you can print up to 8 formatting arguments in the same call to printf, with one caveat. **Printing of floating point numbers are not officially supported in the simulator, dlxsim.**

Since you will have your first floating point number in this assignment (estcpu is floating point), you may want to print it for debugging. By luck, printing of a single floating point number actually works. If you try to print any other formatting arguments with the floating point number, it will not work. Therefore, if you have to print a mix of floating point numbers and anything else, print the floating point numbers in their own print statements and you will get the correct output.

Example and Testing code

We have changed apps/example/ to be a simple application for testing mailboxes. It is built and runs the same way as the example code from lab 2.

We have provided simple testing code in apps/userprog for testing your lottery

scheduling. This code in no way tests everything: in fact, it is only a relatively simple test. Feel free to modify this code as you wish to test the lottery scheduling completely. **Think of test cases to actually verify if the scheduler gives priority to an emulated IO process.**

Synchronization Library

We have provided our synchronization code from the solution to lab 2 as an object file. The functions work the same as the lab 2 handout described. The OS Makefile has been modified to link our synch.o into DLXOS as it is built.

Assignment

Download and untar ~ee469/labs_2018/Labs/lab3.tar.gz. This will create a `lab3/` directory for you. Put all your work in this directory structure.

1. (30 points) **Implement mailbox-style inter-process communication in DLXOS**. The functions and their descriptions are listed in the table above in the section describing mailboxes. You must put the implementations of these functions in os/mbox.c and include/os/mbox.h.

   Since there is no malloc() available in DLXOS, you will need space in which to store mailboxes and space to store mailbox messages. You must make two global arrays: one of mbox structures and one of mbox_message structures. Use the mbox_message structures to put messages into a Queue inside the mbox structures.

   To make sure that you don't run out of mailboxes unnecessarily, you need to add code to ProcessFreeResources that, upon a process's death, goes through all the mailboxes and removes it from the list of processes with that mailbox open. Then, if no other processes have that mailbox open, close the mailbox automatically.

   Be sure to do exhaustive input and sanity checking. For example, make sure that the current process has opened a given mailbox before sending, receiving, or closing it.

2. (10 points) **Test your mailbox implementation by implementing the Radeon chemical reaction simulation from lab 2 using mailboxes.** Put your solution in apps/q2. Recall that the chemical reaction equations are:

   2 H2O -> 2 H2 + O2

SO4 -> SO2 + O2
H2 + O2 + SO2 -> H2SO4

There are a few minor changes for this week's lab from lab 2.

- Do not use locks, semaphores, or conditional variables in your user code. Use only mailboxes.
- You should have one mailbox for each type of molecule.
- You should have one program for each of the 3 reactions, and the 2 generators.
- Each generator process will be responsible for generating **ONLY** 1 molecule, and each reaction process will be responsible for performing **ONLY** 1 reaction. In other words, you should not have any loops inside the generator and reaction programs. Instead, your makeprocs process will create the proper number of each type of process.

To prevent the mailboxes from being automatically deallocated when a process exits, makeprocs needs to not only create all the mailboxes, but also open them, and then wait until all child processes have exited before closing them and exiting.

3. (40 points) **Implement Lottery based scheduling in DLXOS.**
Start by implementing a lottery scheduler where every process starts with the number of tickets given by `p_nice`.Test your scheduler with the provided test case `userprog4.c`. The command for this test case is

```
dlxsim -x os.dlx.obj -a -u userprog4.dlx.obj <number>
```

The *lottery_test* program in apps is a test program to test your static lottery scheduling implementation (not dynamic scheduling). Test your dynamic lottery scheduling implementation with your own test cases. Use Macros (RR_SCHED and LT_SCHED) to support both Lottery Scheduler and Round-Robin Scheduler.

```
#ifdef RR_SCHED
  <Round-Robin Scheduler code>
#endif

#ifdef LT_SCHED
  <Lottery Scheduler code>
#endif
```

4. (10 points) **Implement CPU running time statistics in DLXOS.** If pinfo is set to 1, ProcessSchedule should print the total amount of jiffies that a process has run .You will have to change various functions in os/process.c, as well as the PCB structure in include/os/process.h to support this.

Keep in mind that you should *not* print the total time elapsed since a process started running (i.e. `ClkGetCurJiffies() - pcb->process_start_time`), but rather the cumulative time that it has gotten to run on the CPU. You also cannot simply count the number of times that it has been switched in and out, because it can be switched in and out at irregular intervals.

It should print the messages "CPUStats: Process `pid` has run for ## jiffies, priority = #", where `pid` is the process id and the priority is the no of tickets the process holds at the moment.
You can define the format string as below. #define PROCESS_CPUSTATS_FORMAT "CPUStats: Process %d has run for %d jiffies, priority = %d\n"

5. (10 points) **Implement sleep, yield, and ProcessIdle in DLXOS.** Write code to support:
   - `void sleep(int seconds);` - puts the current process to sleep for the specified number of seconds. Utilize the existing ProcessSuspend function to help you. Be sure to update a sleeping process's no of tickets properly when it is woken up. You will need to update ProcessSchedule to look for any "autowake" processes that should be woken up after a given number of jiffies. Put your code for this in os/process.c in the "ProcessUserSleep" function. You also need to update the ProcessSchedule function to NOT update the no of tickets of a process marked as sleeping.
   - `void yield();` - marks the current process as "yielding." Write your code in os/process.c. ProcessSchedule is called immediately after ProcessSleep in the yield() trap in traps.c. Therefore, you don't actually have to yield the process, you simply need to mark it as yielding, then modify ProcessSchedule to not update the tickets of a yielding currentPCB and reset the yielding flag.
   - `void ProcessIdle();` - just an infinite while loop function in os/process.c. You must modify the necessary functions in process.c to support switching to the idle process when there are no runnable processes, but there are sleeping processes that will wake up in the future. This process was described in the Background section above.

Turning in your solution

You should make sure that your `lab3/` 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 turnin 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, change to the directory containing the `lab3` directory and execute the following command:

```
turnin -c ee469 -p lab3-Y lab3
```

where `Y` stands for your lab session, so it will be lab1-1 or lab1-2 or lab1-3 depending on which lab session you are in.
Wednesday, 2:30pm = 1, Friday, 11:30am = 2, Friday, 2:30pm = 3.