ECE 469 Spring 2020 Laboratory #3

Changelog

2/29/20: Minor revisions for clarification. Formatted as ~~deletion~~ and addition . To locate all the revisions, search for "updated 2/29/20" in the webpage.

I. Introduction

The purpose of this lab is to become familiar with mailbox-style interprocess communication, and priority-based process scheduling. In particular, you will accomplish the following:

- implement an interprocess mailbox API in DLXOS,
- test your mailbox API with a chemical reaction simulator,
- and implement BSD 4.4-style priority scheduling 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.

Similar to lab2, lab sessions in the first week of lab 3 will cover questions 1 and 2, and you are strongly encouraged to finish them in the first week. The second week of lab 3 will cover questions 3, 4 and 5, which are heavier, and you have the second and the third weeks to work on them.

II. 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. Be aware if you allocate a new lock in mbox_create(): you may run out of available locks with repeated invocations toÂ mbox_create() (updated 2/29/20)	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. De-allocating means setting it as no longer in use and removing all queued up messages. Returns MBOX_FAIL on failure and MBOX_SUCCESS on success. (updated 2/29/20)	none
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	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, and error occurs. If the mailbox is empty, wait until there is a	reading from shared

maxlength, void *data)	message to be received. Returns MBOX_FAIL on failure and size of the msg return in bytes on success. Note that this should fail if the calling process does not have the mailbox open.	memory, sem_wait()

All mailbox functions should be written such that they are free of synchonization problems like race conditions and deadlocks. In other words, user programs should 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 _intrhander 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 priority scheduling rather than the round-robin scheduling algorithm laid out above.

BSD 4.4 Priority Scheduling

As you can see, the current round-robin process scheduling algorithm is severely lacking for a modern operating system. It gives all processes equal time on the CPU, whether they need/deserve it or not. Imagine if your current operating system ran this way: processes in the background would severly slow down any processes in the foreground.

Also, processes that spend a lot of time waiting on I/O requests (like file reading, network communication, etc.) have a severe disadvantage because they go to sleep before using their entire allocated CPU time. Therefore, processes which only churn out raw CPU computations will run much faster than processes that have to wait on I/O requests. A better scenario would allow processes to "catch up" if they repeatedly do not use up their entire allocated window on the CPU. One "CPU window" is generally referred to as a *process quantum* which is defined as the time between periodic calls to ProcessSchedule. Therefore, one process quantum is equal to the period of your timer interrupts.

There are many solutions to these problems. What follows here is a description of how these problems were solved in the BSD 4.4 kernel, the basis for popular operating systems like OpenBSD, NetBSD, FreeBSD, and OS/X.

To allow some processes more time on the CPU than others, the concept of *priority* is introduced. Each process has a number from 0 to 127 associated with it. A lower number means a higher priority. In other

words, a process with priority 10 should get much more CPU time than a process with priority 80. There are 2 types of processes: kernel processes (servicing I/O requests, handling low-level drivers, etc.) and user processes. Kernel processes can have priorities from 0 to 49, and user processes can have priorities from 50 to 127. **Note that DLXOS does not have kernel processes.**

In order to run higher priority processes more often, instead of having a single runQueue, there are 32 runQueues. Processes with a priority from 0-3 go in queue 0, 4-7 go in queue 1, 8-11 go in queue 2, and so on. In other words, the queue number for a given priority can be computed as:

```
queue_number = priority / PRIORITIES_PER_QUEUE;
```

where PRIORITIES_PER_QUEUE = 4. **Important: this only works with integer division.** Note that just because only 4 priority levels are assigned to each queue does *not* mean that a maximum of only 4 processes can exist in a given queue. Multiple processes can have the same priority.

To decide which process to run, you start at the lowest-numbered (highest-priority) queue (queue 0), and look for any processes in there. If you don't find any, move up one queue. Continue this until you find the queue with the highest priority that contains processes.

Within each priority queue, processes are scheduled in a standard round-robin fashion. When a process is done executing, it is moved to the back of its corresponding priority queue. So, once you find the highest priority queue with an existing process, you run the process on the front of that queue.

This method will never run processes in lower-priority (higher-numbered) queues until all higher-priority (lower-numbered) queues are empty. Processes can specify their relative priority when they are created by using a parameter known as the "nice" value. If a process wants to be "nice" and allow other processes more time, it gives a high "nice" value. We'll call this value `pnice`. BSD 4.4 allows pnice to range from -19 to 20 for kernel processes, and 0 to 20 for user processes.

That solves the problem of giving some processes more time and others less, but we still have the problem of process "starvation", where some processes can take up too much time, and others then get too little. To solve this, BSD 4.4 recomputes the priority of each process as it is switched in and out, and as it sleeps and wakes. So, as a process uses more CPU time, it will gradually get a lower priority (higher number) to enable "starved" processes to run. In addition, as processes become starved for CPU time, they will get a higher priority (lower number). We will designate the estimated cpu usage as `estcpu`, which is a floating-point (double) variable type. **`priority` is an integer variable type.**

Priorities are computed and changed according to the following formulas:

```
if (process has used an entire window of CPU time) estcpu++;
priority = BASE_PRIORITY + estcpu/4 + 2*pnice;                (1)
if (10 process quanta have passed (0.1 second)) {
  for all processes in all runQueues {
    estcpu = [ estcpu * (2*load)/(2*load + 1) ] + pnice;      (2)
    recompute priority according to equation (1) above;
  }
```

```
}
```

BASE_PRIORITY is the minimum priority that a process can have. This is set to 50 for all user processes, and 0 for all kernel processes. You must ensure that a user process's priority never goes below this number. `load` is the average of the sum of the lengths of the run queue and the short-term sleep queue (i.e. processes waiting on disk I/O). For simplicity in this assignment, assume that `load` is always 1, which makes `(2*load)/(2*load+1) = 2.0/3.0`.

Equation (2) decays the estcpu of all processes as they run for longer periods of time. Since lower numbers mean higher priorities, "decaying" the number means that processes that have been in the runQueue for a long time gradually get higher priorities.

The only remaining question is how to handle processes that go to sleep before they use their entire window of CPU time. First, these processes will not have their estcpu value incremented when they go to sleep or wakeup. In addition, since they missed out on all the decays of every process's estcpu values, we should go ahead and let them "catch up" on the decays that they missed. Therefore, when a process wakes up, the following algorithm is executed:

```
if (sleeptime >= 10 process quanta (0.1 second, 100 jiffies)) {
  int num_windows_asleep = sleeptime / (TIME_PER_CPU_WINDOW * CPU_WINDOWS_BETWEEN_DECAYS);
  estcpu = estcpu * [ (2*load)/(2*load+1) ] ^ (num_windows_asleep);
}
```

where TIME_PER_CPU_WINDOW = 0.01 seconds (10 jiffies) and CPU_WINDOWS_BETWEEN_DECAYS = 10. Note that the "^" symbol here means exponent, not XOR.

Your task is to implement this priority scheduling alorithm in DLXOS.

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 noone 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.

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. In fact, if we can get ProcessIdle into a PCB with priority 127, with a little finessing of the lowest-priority (highest-numbered) queue, our scheduling algorithm will already switch to ProcessIdle for us when there is nothing else on the runQueue.

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)`), and set it's base_priority to the maximum possible priority, 127, to ensure that it always keeps the maximum priority.

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

Since the bottom level queue is a round-robin queue (just like all the other queues), you'll need to add a little code in ProcessSchedule when determining which process to run. The algorithm looks like this:

```
pcb = ProcessFindHighestPriorityPCB();
if (pcb == idlePCB) {
  move pcb to back of bottom queue
  pcb = ProcessFindHighestPriorityPCB();
}
```

After you've moved the pcb to the back of the bottom queue, if it is still the highest priority PCB, then it is the only PCB and you can go ahead and switch to it.

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.

yield() gets a little more tricky when dealing with priority queues. You must not increment the estcpu of a process that has yielded in ProcessSchedule. Also, for simplicity, when a process yields, it will just be put at the back of the priority queue that goes with its current priority. If there are no other processes on that particular priority queue, it will just get to run again right away.

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 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 resouces 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.

III. 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 BSD 4.4 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/prio_test for testing your priority queues. This code in no way tests everything: in fact, it only contains a few relatively simple tests. Feel free to modify this code as you wish to test your priority queues.

Synchronization Library

We have provided our synchonization 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.

IV. Assignment

Download and untar ~ee469/labs_2020/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. You may use semaphores, locks, or condition variables in the mbox functions. (updated 2/29/20)

   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 chemical reaction simulation ~~from lab 2~~ using mailboxes.** The first argument provided will be the number of S2 inserted, and the second argument will be the number of CO inserted into the system. (updated 2/29/20)
   Put your solution in apps/q2. Recall that there are 2 primary chemical reactions that occur in the atmosphere:

   1) S2 -> S + S

2) 4 CO -> 2 O2 + 2 C2
3) S + 2 O2 -> SO4

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. The only exception to this rule is one semaphore which may be used for keeping track of the number of processes that have completed. (updated 2/29/20)
- 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. (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 using the printf format string that is #define'd in include/os/process.h. 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.

   For the "prio" field in the format string, just print 0 for now until you implement priority scheduling later.

   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.

4. (40 points) **Implement BSD 4.4 priority scheduling in DLXOS.**
   Implement the full BSD 4.4 scheduling algorithm as outlined above. You will need to modify functions in os/process.c and the PCB structure in include/os/process.h, among other things.

   It is highly recommended that you write helper functions in process.c to make your implementation of the scheduling algorithm easier to understand. Here are some suggested function prototypes. You'll have to figure out what they should do from their names as you think about the priority scheduling problem. You don't have to write these, but your life will probably be easier if you do:

   - `void ProcessRecalcPriority(PCB *pcb);`
   - `inline int WhichQueue(PCB *pcb);`
   - `int ProcessInsertRunning(PCB *pcb);`
   - `void ProcessDecayEstcpu(PCB *pcb);`
   - `void ProcessDecayEstcpuSleep(PCB *pcb, int time_asleep_jiffies);`
   - `PCB *ProcessFindHighestPriorityPCB();`
   - `void ProcessDecayAllEstcpus();`
   - `void ProcessFixRunQueues();`
   - `int ProcessCountAutowake();`

- `void ProcessPrintRunQueues();`

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 priority 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 "ProcessSleep" function. You also need to update the ProcessSchedule function to NOT update the priority/estcpu 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 estcpu 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.

V. Submitting 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, submit it via Blackboard in a zip/tar file.The name of tar file should clearly mention lab number and the group number.For eg:- For lab3, it should be ee469lab3_g12.tar.gz where lab3 is the lab number and g12 can be any group number(students can find their group number in Blackboard announcements

---

ee469@ecn.purdue.edu