CPSC 415
Assignment 2

This assignment is, ideally, to be done in pairs. Working alone is strongly discouraged.

1 Motivation

The purpose of this assignment is to investigate process pre-emption, simple device drivers, and simple inter-process communication (IPC). In this assignment you will extend your kernel to pre-empt processes, provide a simple sleep device, and provide a basic message passing IPC service. In doing so you will learn about how a kernel schedules a CPU, manages shared resources, handles interrupts, provides inter-process communication.

2 To Start

Although the assignment is doable on your own, you are very strongly encouraged to work with a partner. Your starting point is the kernel resulting from the first assignment. If you did not complete assignment 1, or don't wish to use your kernel, you may use a kernel from someone else in the class, with their permission, or the solution kernel you will get when you clone the assignment repo. You are also free to merge you or your partner's assignment 1 solution with the sample solution if you prefer. Your assignment must compile on the undergrad Linux machines and run in the Bochs emulator on those machines.

In this assignment you will modify the code in several modules and add a couple more modules to extend the kernel. ***Note: If you use the supplied solution, the*** *kfree()* ***function is not fully implemented and does not actually free memory. Consequently, don't get too exuberant with*** `kmalloc()` ***and expect*** `kfree()` ***to help you out.***

The provided assignment 1 solution kernel compiles without any compiler warnings except for routines in the libx directory. Your code, regardless of the kernel you start with, must also compile without warnings except for files in the libx directory and maybe a warning about an unused variable in ctsw.c to hold the kernel stack pointer. You are not allowed to change the command line options to the compiler to achieve this. (You should check the sample solution to see how it avoids having the unused variable message.)

Note: Sometimes you may see the message *Clock skew detected. Your build may be incomplete.* I have never managed to figure out just what conditions cause this message to be generated, nor has a problem with the build ever been observed when the message is printed. If in doubt do a *make clean* and rebuild the code. You might get the message again, but it won't produce the required zImage file if there was an actual problem.

2.1 Assignment 2 - GIT Usage

For this assignment everyone is required to use a GIT repository on the department's stash (bitbucket) server to manage the source code for their assignment. Furthermore you are required to regularly "push" your changes to the server. Consult the assignment 1 description for the basic GIT references and commands to do this.

2.2 Using your own kernel or someone else's

If you plan on using the supplied solution kernel, skip the step in the next paragraph.

After completing the steps in the previous section copy all the modified **.c**, and **.h** files from assignment 1 that you intend to use overtop of the supplied files. Use the git **add**, **rm** (if needed), **commit** and **push** commands to commit the changes and push them to the repository. You should also tag this set of files as your starting point. (Check the GIT references to figure out how to do this if you didn't do it in A1.)

NOTE: You are expected to regularly check-in the work you have done throughout the assignment. To encourage this part of the grade for this assignment will be based on a regular and reasonable check-ins of your code. If working in pairs it is expected that both partners will contribute to the assignment and check-in code. It is accepted that the checking in of code may not be 50/50, but if only one person is checking in code then questions might be raised about who is doing the work and consequently who deserves the marks.

3 Objectives

You will extend the kernel from the first assignment. **In your documentation you must state which kernel you are using, be it your own, the provided solution kernel, or someone else's.**

There are three main implementation goals for this assignment:

- To add pre-emption. (i.e. time slicing);
- To implement a simple sleep device;
- To implement an inter-process communication system.

In addition, you will need to implement a couple of auxiliary system calls to help things work smoothly.

There are numerous ways to tackle this assignment. One way is to implement the kernel extensions in the order they are described in this document. Another order would be to implement section 3.3 after completing 3.7. If you use this latter approach you may want to disable pre-emption while working on 3.3, but don't forget to turn pre-emption back on and re-test things.

Regardless of the order you choose, you need to test each extension before proceeding to the next. The list below summarizes the extensions. For most of these extensions the dispatcher function `dispatch()` in `dispatch.c` will require modifications and consequently it may not be explicitly mentioned.

Auxiliary System Calls: Add `sysgetpid()`, `syskill()` `sysputs()` in `syscall.c` - some useful system calls.

IPC system calls: `syssend()` and `sysrecv()` in `syscall.c` - used by processes for IPC.

Messaging system: `send()` and `recv()` in `msg.c` - used by the kernel for servicing the IPC requests.

Pre-emption: `contextswitch()` and `contextinit()` in - `ctsw.c` for timer interrupts and time slicing.

Idle process: `idleproc()` in `init.c` - to avoid running out of processes.

Sleep system call: `syssleep()` in `syscall.c` - used by a process to sleep a specified time.

Sleep device: `sleep()` and `tick()` in `sleep.c` - code in the kernel for processing sleep request and timer ticks.

Producer/consumer: `producer()` (really root) and `consumer()` in `user.c` - improved producer/consumer.

Note you may also need to update various .h files to add add constants, data structures, prototypes etc. Such updates will not always be explicitly mentioned in the assignment description.

3.1 Auxiliary System Calls

Add the four new system calls in `syscall.c` with corresponding additions to `disp.c`. The system call `sysgetpid()` returns the PID of the current process and `syskill()` takes a PID as an argument, and terminates the process. To simplify things, a PID is of type PID_t which is a typedef of an unsigned int.

The prototype for `sysgetpid()` is

```
extern PID_t sysgetpid( void );
```

The system call `sysputs()` is to be used by processes to perform output. Since processes will become preemptable, and the screen is a shared resource, access to the screen must be synchronized via the `sysputs()` system call. The system call takes a null terminated string that is to be displayed by the kernel. Since the kernel cannot be pre-empted, the kernel side of `sysputs()`can use `kprintf` to print the string. The prototype for the system call is

```
extern void sysputs( char *str );
```

You can use sprintf() to create formatted strings for fancier printing. (sprintf() is like printf() except it puts the resulting string into memory, as opposed to printing it.)

The `syskill()` call terminates the process identified by the argument to the call. The call returns 0 on success. The call fails with -1 if the target process does not exist. It is okay for a process to call kill on itself. In this case the call does not return and the process terminates. The prototype is

```
extern int syskill( PID_t pid );
```

The final call you are to implement is `syssetprio()` which allows a process to set its priority. The prototype for the call is:

```
extern int syssetprio( int priority );
```

In this system a process can have a priority from 0 to 3 with 3 being the lowest priority and 0 being the highest priority. By default a process is created with priority 3. The result returned by this call is -1 if the call failed because the requested priority was out of range, otherwise it returns the priority the process had when this call was made. There is one special case and that is priority -1. If the requested priority is -1 then then call simply returns the current priority of the process and does not update the process's priority.

3.2 Changes to Process Management

For this assignment you must clean-up resources that are directly associated with a process. This includes, among potentially other things, the stack and any data that might be part of the message passing system.

One of the resources associated with a process is its PCB and as a result it must be possible to reuse a PCB. A side effect of this is that you will have to be careful with respect to how you assign PIDs. A PID cannot simply be the index into the PCB table. A good PCB selection algorithm will make it possible to index into the PCB in small, constant time while still making it easy to determine whether or not a PID is valid. In addition, to minimize the problems with process interactions based on PIDs, the PID reuse interval needs to be as large as possible.

In assignment 1 if a *process* runs off the end of its code (i.e. doesn't call `sysstop()` and just keeps going) or executes the C language `return` statement bad things happen. (If you don't believe this just try adding a `return` to one of your processes in assignment 1.) In this assignment you are to modify the process creation code so that if a process does a return, either explicitly through the **return** statement or by running off the end of its code, the process is cleaned up appropriately and the process/system does not crash or restart.

There are a couple of ways to do this. One is to set-up the stack of the process so that when the process "returns" it transfers control to `sysstop()`. (i.e. set-up the stack so that the spot that contains the return address for the function call now contains the address of sysstop().) Alternatively you could introduce a "wrapper" process. This process takes as an argument the function address passed in as the argument to `syscreate()`. Now each time a process is created the wrapper process is what is created. The body of this process will call the real process (i.e. the address of the user function passed to syscreate()), and when the "user process" returns, the "wrapper process" will call `sysstop()`. For this approach to work, when the wrapper process is created the stack will have to be constructed such that when the wrapper process runs it can retrieve the address of the user function to "call" (i.e. run). That means the arguments will have to be placed in the appropriate place of this initial stack. It should be noted that with both of these two approaches no assembly language code is required. You may choose either approach or develop something of your own.

3.3 IPC System Calls

In `syscall.c` implement two more system calls: `syssend();` and `sysrecv()`. Processes will send and receive unsigned long integer messages using these calls. As part of implementing these calls the dispatcher will need to be modified to appropriately call the `send()` and `recv()` functions, described later. As before, be sure to add these prototypes to `xeroskernel.h`.

3.3.1 syssend()

```
extern int syssend( unsigned int dest_pid, unsigned long num );
```

The `syssend()` system call takes two parameters: a destination PID, and the integer to send. The call returns 0 on success and a negative value if the operation failed. The system call blocks until the send operation completes. The send operation fails returning -1 if the destination process terminates before the matching recieve is performed or -2 if the process does not exist. If the process tries to send a message to itself, it returns -3 and it returns -100 if any other problem is detected.

3.3.2 sysrecv()

```
extern int sysrecv( unsigned int *from_pid, unsigned int * num);
```

The `sysrecv()` system call takes a pointer to an integer, from_pid, which specifies the address of a variable containing the PID of the process to receive from, and num, the address of the integer to store the received value into. If the PID to receive from is non-zero then it specifies the PID to receive from. If the PID is 0 then the process is willing to receive from any process. When the PID is non-zero the system must handle the following scenarios:

1. The process to receive from is blocked on a send to this process. The integer being sent is transferred and both processes are put on their respective ready queues.

2. The process to receive from is not blocked on a send to this process. This process blocks until the matching send is performed. If the process terminates before this happens then then the process is unblocked and -1 returned.

3. The process to receive from does not exist. In this case -2 is returned immediately.

4. If the process tries to recieve from is itself then -3 is returned immediately.

In the situation when the from_pid is 0 then the system must handle the following scenarios:

1. A process is waiting to send. In this case the integer is transferred and both processes are put on their respective ready queues.

2. No process is ready to send to this process. The process is then blocked until a process sends to this process. In this case the location point to by from_pid will be set to the PID of the process doing the send.

3. If address for from_pid is invalid, then -5 is returned.

In both cases if the address of *num* is invalid -4 is returned and if the receiving process is the only process, other than the idle or null process in the system then -10 is returned. When a message is received 0 will be returned. If any other problem is detected then -100 is returned.

3.4 Messaging System

In the file `msg.c` implement the following two functions: `send()` and `recv()`. These functions implement the kernel side of the system calls `syssend()` and `sysrecv()` respectively. The `send()` function is called by the dispatcher upon receipt of a `syssend()` request to perform the actual work of the system call. If `send()` is called and the receiving process is not blocked on a matching receive, then the sending process is blocked until the matching receive call is made. If the receiving process is blocked and ready to receive the message, then the message is copied into the receive buffer, and both processes are placed on the ready queue. You will probably need to pass the PID, and buffer location to`send()`, as well as the sending process's process control block, however the exact number and type of parameters is a design decision left to you. The function will also need to perform appropriate parameter checking before doing the actual processing.

The `recv()` function is called by the dispatcher upon receipt of a `sysrecv()` request. If `recv()` is called before the matching send, the receiving process is blocked until the matching send occurs. If the matching send has occurred (this implies the sending process is blocked), the message is copied into the receive buffer and both processes are placed on the ready queue. If the PID is 0, then the earliest outstanding send to the receiving process (the earliest unreceived send) is the matching send to the receive. As with `send()`, you will probably need to pass the PID, buffer, and process control block of the receiving process, but again that is a design decision left to you. Also, in the case when the receiving process is willing to receive from any process, the memory location in the process space will need to be updated to reflect the PID of the process doing the sending. Like `send()`, this function must perform appropriate parameter checking before doing the actual processing.

If a process doing a `syssend()` is blocked waiting to complete the send, and the target process terminates, then `syssend()` will return a -1 to indicate that the PID is invalid. If a process performing a `sysrecv()` is blocked waiting for a message from a specific process, and that process terminates, then -1 is returned by `sysrecv()`.

3.5 Pre-emption

In order to pre-empt a process the kernel must get control of the processor. Since the current process implicitly has control of the CPU, a hardware timer interrupt is used to transfer control to the kernel on a regular basis. In order to make our kernel pre-emptive, four steps need to be performed. First the context switcher needs to be modified to handle both the hardware timer interrupt, and the software system call interrupt; this is outlined in Figure 1. This figure is a broad outline of the processing and does not identify precisely what needs to be done and how.

Immediately after an interrupt occurs, the interrupts need to be disabled since we are not building a re-entrant kernel. (In the `set_evec()` code provided, when the interrupt vector is set-up, a trap gate is specified instead of an interrupt gate so disabling interrupts isn't done automatically. Note that the context switcher draft 2 assumes that the interrupts are automatically disabled. However, a couple of slides later it is pointed out that one might have to disable interrupts explicitly.) As a result, this means that interrupts need to be deferred while the kernel is running. Consequently, the first instruction executed by the ISR needs to disable interrupt checking by the CPU. The `cli` instruction does this. It is not necessary to re-enable interrupts before the `iret` is performed because the restoration of the saved `eflags` register will have the interrupt bit appropriately set. If you forget to do this you probably will not encounter any problems, or at least very rarely. It

```
ContextSwitcher
                        push kernel state onto the kernel stack
                        save kernel stack pointer
                        switch to process stack
                        save return value of system call
                        pop process state from the process stack
                        iret

timer_entry_point:
                        disable interrupts
                        push process state onto process stack
                        save some sort of indication that this is a timer interrupt
                        jump to common_entry_point
syscall_entry_point:
                        disable interrupts
                        push process state onto process stack
                        save some sort of indication that this is system call

common_entry_point:
                        save process stack pointer
                        switch to kernel stack
                        pop kernel state from kernel stack
                        recover entry point type to determine return type.
                        return from context switcher
```

Figure 1: The new context switcher.

turns out that if you don't do the `cli` there is about a 5 instruction window where problems could occur if a hardware interrupt goes off. (Think about why the window is so small or what would cause the window of vulnerability to close after about 5 instructions.)

Since each interrupt has its own vector (i.e. address of the ISR code to jump to), we can easily distinguish between a system call trap and a hardware interrupt, like the timer, by having different ISRs. Ideally we want to reuse as much code as possible between the hardware interrupt and system call trap processing code. A hardware interrupt, unlike an exception used for a system call, requires that ALL registers contain what they had before the interrupt so special care needs to be taken.

Although which registers can be clobbered during the kernel's processing of a system call is design specific, it is the case that the return value of the system call will be in register `eax`. If the context switcher returns the value of a system call in `eax`, and the same code is used to return from an interrupt (which should be the case) then the return value of a hardware interrupt needs to be the value of `eax` when the interrupt occurred.

To have hardware interrupts fit in with our context switch model, one approach is to make a hardware interrupt appear like a system call. To do this the context switcher needs to return an indication to the dispatcher that a timer interrupt occurred. It is recommended that you define a `TIMER_INT` request identifier along with all the other system call request identifiers that a dispatcher

can receive and have that returned by the context switcher when a timer interrupt occurs.

Next, modify the dispatcher to handle the `TIMER_INT` request. This request should do the same thing as the `YIELD` request: place the current process at the end of the ready queue, de-queue the next process in the ready queue, and run it. In the next section you will have to add an additional line of code to this request handler, so don't share it with the yield code.

Third, change the initial flags being set by the context initializer in `create.c`. Instead of 0, the interrupt bit should be enabled, i.e., the initial flags should be $0x00003200$. (The 0x3000 will prevent processes from accessing privileged I/O space, not that that should be an issue.)

Finally, you must add a `set_evec()` call as part of configuring the timer interrupt. The timer is on IRQ0 which translates to interrupt number 32. Make sure the handler is the timer entry point in the context switcher and not the syscall entry point. You will also have to initialize the timer and enable the corresponding interrupt on the PIC (i8259). To do this use the `initPIT( divisor )` function. This function takes one argument, the rate at which the timer should trigger the interrupt. For example, specifying a value of 100 will mean that the interrupt will be triggered every 100th of a second (every 10 milliseconds). Thus, a process will be given a time slice of 10 milliseconds, before being pre-empted. This is the recommended time slice for the system. Thus, after the call to `set_evec()` you should add

```
initPIT( 100 );
```

When a timer interrupt occurs, the hardware has to be re-armed in order for the interrupt to occur again. This is done by signalling an end of interrupt (EOI). To signal an EOI use the provided function `end_of_intr()`. This function should be called, probably from `dispatch()` as the last operation in the servicing of the `TIMER_INT` request. You will notice very quickly if you forget to do this.

3.5.1 Priorities

In this system processes have priorities and the scheduling policy is that higher priority processes (i.e. lower priority number) are always run first and that round-robin scheduling is used within a priority. You are not to add support to avoid CPU starvation or to deal with the priority inversion problem. If you are using the supplied kernel, observe that the code to manage the ready queue is in disp.c and that there is a single queue. Consequently you will need to modify the queue management routines to support multiple ready queues. It is okay to change the next() and ready() functions to take additional arguments, perhaps the number of the queue they are working on.

3.6 Idle Process

Since processes might become blocked for one reason or another, such as waiting to receive a message or sleeping, among other things, it is important to ensure that there is at least one ready process available at all times. As you know, bad things happen if the kernel runs out of processes to schedule. In order to prevent this, you should create an idle process. Its only job is to be ready and is simply an infinite loop. You will need to modify the dispatcher so that the idle process will run **only** if no other process is available. Instead of just having the idle process execute an infinite loop with an empty body, you might want to try executing the `hlt` instruction in the body of the loop. (You don't have to use `hlt`, it is just something you can try if you like.)

The prototype for the process should be

```
extern void idleproc( void );
```

and it should be created by `initproc()`. You should start with a simple idle process before trying to use `hlt`, if you choose to try `hlt`. If you ever schedule the idle process, and don't have the timer interrupt working, control will never return to the kernel.

3.7 Sleep Device

The sleep device will allow processes to sleep for a requested number of milliseconds. First, add a system call `syssleep()` to `syscall.c` that takes one unsigned int parameter and returns an unsigned int. The single parameter is the number of milliseconds the process wishes to sleep. For example, to sleep for a second the call would be `syssleep(1000);` The prototype for the calls is

```
extern unsigned int syssleep( unsigned int milliseconds );
```

and should be added to `xeroskernel.h`. The return value for `syssleep()` is 0 if the call was blocked for the amount of time requested otherwise it is the amount of time the process still had to sleep at the time it was unblocked. In this assignment there is no mechanism to unblock the process early, but in assignment 3 there will be. So, at this point one would always expect to get back 0, although in A3 that won't be the case.

Modify the dispatcher appropriately to recognize the system call, and have it call the corresponding kernel `sleep()` function that you will write. To the `TIMER_INT` request service code add a call to a function called `tick()`. This function will also be written by you and it notifies the sleep device code that a clock tick has occurred. Since Bochs is an emulator, it is sometimes the case that the machine the emulator is running on can be emulated faster than the original machine actually ran. As a result, time will pass faster on these machines than in "real" or "wall clock" time. An attempt has been made to minimize this problem through configuration options in the bochsrc file in the xeros directory, but don't be overly concerned if things do not match wall clock time.

To implement the kernel side of `syssleep()`, implement in `sleep.c`, the two functions `sleep()` and `tick()`. The `sleep()` function places the current process in a list of sleeping processes such that its place in the list corresponds to when it needs to be woken. Observe that the clock tick has a certain granularity and that the parameter to `syssleep()` has a finer granularity than a clock tick. To deal with this problem the kernel will treat the value of the parameter to syssleep() as the minimum amount of time to sleep for and wake a process on the next clock tick after this minimum amount of time has elapsed. To simplify the managing of the times and sleep queue it would be a good idea to convert, within the kernel, the amount of time to sleep into "clock ticks" (time slices) and work from there. (Example conversions: 10ms = 1 tick, 11ms = 2 ticks, 0 ms = 0 ticks, 1ms = 1 tick etc). As an example, assuming times have been converted to ticks, if there are three processes in the sleep list, that need to be woken after 5, 7, and 8 time slices respectively, and the current process wants to sleep for 6 time slices, then it should be placed second in the sleep list. You should try to come up with an efficient way of doing this. (Hint: delta list). The process is NOT to be placed on the ready queue until it is woken. After the `sleep()` call completes, the dispatcher selects the next process in the ready queue to run. Don't confuse the behaviour of the kernel `sleep()` function with that of `syssleep()`. The `sleep()` function always returns to the dispatcher, even though a process is being blocked but the `sysleep()` call won't return to the application until at least the specified time has elapsed.

The `tick()` kernel function notifies the sleep device that another time slice has occurred. The function should wake up any processes that need to be woken up by placing them on the ready queue associated with their priority (and marking them as ready), and updating internal counters.

3.8 The Extended Producer-Consumer Problem

At this point you should have a pre-emptive multitasking kernel and you should have extensively tested it. Implement an extended Producer-Consumer type application that does the following:

Note - In the following each piece of printout is to be on a line by itself and start with '"Process xxx" where xxx is the process ID of the process printing the message. The times are to be the times that result from calling sleep appropriately. (The times you specify to sleep should be the actual ones that would be used if the timer on the emulator accurately captured wall clock elapsed time. You do not have to try to select sleep values that result in the process actually sleeping that amount of time according to the "wall clock.")

Create a root process that:

- Prints a message saying it is alive.
- Creates 4 processes and as each process is created prints a message saying it has created a process and what that process's PID is. Each of these processes does the following:
  - Prints a message indicating it is alive.
  - Sleeps for 5 seconds.
  - Does a receive from the root process.
  - Prints a message saying the message has been received and the number of milliseconds it is to sleep.
  - Sleeps for the number of milliseconds specified in the received message.
  - Prints a message saying sleeping has stopped and it is going to exit.
  - Runs off the end of its code.
- The root process then sleeps for 4 seconds.
- Upon waking it sends, in the order listed, the following messages:
  - To the 3rd process created, sends a message (plain text ASCII form) with a number indicating that the receiver should sleep for 10 seconds. (These values are sent as the number of milliseconds to sleep.)
  - To the 2nd process created, sends a message (plain text ASCII form) with a number indicating that the receiver should sleep for 7 seconds. (These values are sent as the number of milliseconds to sleep.)
  - To the 1st process created sends a raw binary message containing a 32 bit integer indicating that the receiver should sleep for 20 seconds. (These values are sent as the number of milliseconds to sleep.)
  - To the 4th process created, sends a raw binary message containing a single 32 bit integer indicating that the receiver should sleep for 27 seconds. (These values are sent as the number of milliseconds to sleep.)

- Attempts to receive a message from the 4th process created.
- Prints the return status of the attempt to receive a message from the 4th process.
- Attempts to send message to the 3rd process created and prints the result of that send attempt.
- Calls syskill() on itself.

4 Write-up

Your write-up is to be a well organized, typed document containing testing and assignment questions sections as described in the next sections.

4.1 Testing Documentation

Your testing documentation is to a plain text ASCII file with the name *Testing.txt*. A file with this name has been provided for you.

You are to handin test output for the 10 test scenarios outlined below. Each test scenario is to be different (i.e. exercises a different code path in the kernel.)

1. Two tests associated with sends.
2. Two tests associated with receive.
3. One test of a send failure not demonstrated in the producer consumer problem.
4. Two tests of receive failures not demonstrated
5. One test demonstrating that time-sharing is working.
6. Two tests of syskill();

The documentation is to organized in sections, with each section corresponding to one test. The sections are to organized in the order of the tests described above. Each section is to indicate the purpose of the test and provide a brief description of how the test was performed (i.e. what was the scenario, how it was actually done.) You are then to provide a sample run for the test. It is acceptable to remove large areas of repeated data provided you indicate this and you may also annotate the output. Finally, indicate if this test was a pass or a fail. Note that it acceptable for your program to fail a test. In such a case you will get full marks for the testing portion of the assignment but may not get full marks in the implementation part of the assignment.

4.2 Some Questions to ponder

You do not have to hand in the answers to these questions. However they are representative of the types of questions you might get on an exam.

1. In the implementation section, the instructions have the use of `set_evec()` occurring before the call to `initPIT()`. Could the order of these two calls be reversed (i.e. could `initPIT()` be called before `set_evec()`? Carefully explain/justify your answer.

2. The functionality of semaphores can be achieved by having a "semaphore" process and then using message sends and receives. Outline in pseudo code

   - What the semaphore process (assume one process per semaphore) would have to do
   - What the P() function (i.e. get semaphore) would do
   - What the V() function (i.e. release semaphore) would do

3. If two processes try to send to each other at the same time they will deadlock. Describe how the kernel needs to be structured/organized to detect this **specific** scenario. Do not consider more complicated scenarios than the one specified.

4. Would your solution be able to handle the situation when process A sends to B, B sends to C and C sends to A? Explain.

5. Currently the function signature for new processes is such that the function takes no arguments. Suppose that the `syscreate()` call were changed so that all functions passed to `syscreate()` expect two arguments. The function being passed to create now has the form:

   ```
   function(int argc, int**argv)
   ```

   Draw and label as precisely as possible a picture of what a process's stack would look like when it is put onto the ready queue for the first time. The picture you draw is to assume that the changes required by the "Changes to Process Management" section have been implemented.

5 How the TA's will Test Your Code

To test your code, the TA's will do three things. First, they will compile and run your code as is, verifying that the producer/consumer seems to work. Second, they may replace your `user.c` with ours, recompile the code and run it. Our code will test your kernel. Third, the TA's will visually inspect your code.

6 What to Hand In

Handing in is done by simply making sure that your git repository with all appropriately committed changes has been pushed to the Stash server. You will be marked on the last push and any late penalty computed will be based on the time of that push. If you have added any files to your code base don't forget to add them to the repo before doing your commit and push. It can never hurt to clone the repo after you have handed things in and then test that your assignment builds and runs as expected on the undergrad Linux machines.