CPSC 415
Assignment 2

Due: Code 22:00 Monday, March 5, 2012
Writeup - Start of class Wednesday March 7
May be done in groups of two.

1 Motivation

The purpose of this assignment is to investigate process pre-emption, simple device drivers, and simple inter-process communication. In this assignment you will extend your kernel to pre-empt processes, provide a simple sleep device, and provide basic send/receive system calls to allow processes to communicate amongst each other. In doing so you will learn about how a kernel schedules a CPU, manages shared resources, and provides inter-process communication.

2 To Start

The assignment is quite doable on your own. However, if you like, you may 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 available in the assignments section of Blackboard Vista. 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.***

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 goals for this assignment:

- To add pre-emption (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 need to test each extension before proceeding to the next. The following list summarizes the extensions. For most of these extensions the dispatcher function `dispatch()` in `dispatch.c` will require modifications and consequently it is not mentioned explicitly.

Auxiliary System Calls: Add `sysgetpid()` `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()` and `consumer()` in `user.c` - improved producer/consumer.

3.1 Auxiliary System Calls

Add the two new system calls in `syscall.c` with corresponding additions to `disp.c`. The system call `sysgetpid()` returns the PID of the current process. This system call will be used in conjunction with the message passing system calls. The prototype for the system call is

```
extern unsigned int sysgetpid( void );
```

The system call `sysputs()` is to be used by processes to perform output. Since processes will become pre-emptible, 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, it can use `kprintf` to print the string. The prototype for the system call is

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

Be sure to add the above prototypes to `kernel.h`. You can use sprintf() to create formatted strings for printing. (sprintf() is like printf() except it puts the resulting string into memory as opposed to printing it.)

3.2 Changes to Process Management

In assignment 1 when a process ends it does not clean up memory. For this assignment you are to modify the PCB to keep track of where a process's stack is kept. When a process terminates the stack memory is to be freed. In addition, 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 can no longer 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 respect to the sending or receiving of messages from/to terminated processes, the PID reuse interval needs to be as large as possible.

Also, 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 me 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 or by running off the end of its code, a crash doesn't occur and and the process is cleaned up appropriately.

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. the stack is set-up so that the spot that contains the return address for subroutine calls 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. 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 code is required. You may choose either approach.

3.3 IPC System Calls

In `syscall.c` implement two more system calls: `syssend();` and `sysrecv();`. Processes will send and receive messages using these calls. The `syssend()` system call takes three parameters: a destination PID, the address of a source buffer, and the buffer's length, in bytes. The call returns the number of bytes accepted by the receiving process or -1 if the operation failed. The system call blocks until the send operation completes. The prototype for the system call is

```
extern int syssend( int dest_pid, void *buffer, int buffer_len );
```

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, the address of the destination buffer, and the size of the buffer, in bytes, as arguments. If the PID to receive from is non-zero, the underlying kernel part of the system call will either transfer the message immediately, if one is available, otherwise the underlying kernel system call code will put the process on a blocked queue until an appropriate message is available. If the PID to receive from is 0, the underlying kernel code for this system call will allow any process to send this process a message. If there is no message available, the process will be blocked. (Observe that as a result of this specification, 0

is now a reserved PID and cannot be assigned to any "real" process. The `sysrecv()` system call returns the number of bytes that were received, or -1 if the PID to receive from is invalid, and -2 if any other sort of problem is detected. (e.g. size is negative, buffer address is invalid, etc) In addition the memory location pointed to by the from address parameter is set to the PID of the process that sent the message. (Note that an actual update to this field is only required when the PID to receive from is set to 0.) The prototype for the system call is:

```
extern int sysrecv( unsigned int *from_pid, void *buffer, int buffer_len );
```

The dispatcher will need to be modified to appropriately call the `send()` and `recv()` functions to perform the IPC (see next part). As before, be sure to add these prototypes to `kernel.h`.

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, buffer, and length arguments to `send()`, as well as the sending process's process control block to this function, 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, buffer length, 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()`.

Also note that since it is possible the buffer sizes of the sender and receiver do not match, the kernel will have to ensure that data is not copied or read beyond the end of a buffer. A buffer size of 0 is permissible. If a buffer size of 0 is specified, then the address of the associated buffer is ignored by the kernel.

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.

```
                        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
                        set eax to 1
                        jump to common_entry_point
syscall_entry_point:
                        disable interrupts
                        push process state onto process stack
                        set eax to 0

common_entry_point:
                        save process stack pointer
                        switch to kernel stack
                        save system call arguments for recovery
                        save eax (is_timer flag) for recovery
                        pop kernel state from kernel stack
                        return from context switcher
```

Figure 1: The new context switcher.

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.

Since each interrupt has can 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 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.

Second, 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 another `set_evec()` call to `contextinit()`. 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 signaling 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.

3.6 Idle Process

Since, processes might become blocked for one reason or another, such as waiting to receive a message or blocked on a device, it is important to ensure that there is at least one ready process within the system 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()`, right after the root process is created. You should start with a simple idle process before trying to use `hlt`, if you choose to try `hlt`.

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 `kernel.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.

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. For example sleeping for 10,000 milliseconds may result in the process sleeping less time than that when compared to the "real" elapsed 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 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()` functin with that of `syssleep()`. The `sleep()` function always returns to the dispatcher, even though a process is being blocked and the `sysleep()` call won't complete for some time.

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 (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. 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 with a value 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 with a value 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 message with a value 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 message with a value 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 sysstop()

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

You are to handin, on paper, test output for the 8 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 receives().
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.

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. (Hand annotation is acceptable.) 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 be penalized in the implementation part of the assignment.

4.2 Assignment Questions

Answer the following questions. These questions are to be handed in as part of the write-up. These questions are worth 16 marks.

1. **[2 marks]** Suppose you had an infinitely accurate clock and that a process asks to sleep for 24 ms. Based on your kernel implementation carefully explain how long the process will actually sleep for based on this accurate clock.
2. **[4 marks]** 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 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. **[2 marks]** If two processes try 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.

4. **[2 marks]** 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. **[3 marks]** 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 in as an argument to this call are to be of the following 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 to hand in things

1. Submit, electronically, your c, h, and compile directories from the Xeros distribution. Make sure there are no .a or .o files in these directories when you hand them in. Add a file GROUPMEMBERS.txt to the .c directory to indicate who worked on this code. Include this file even if you worked along. If you have anything you want the TA to know about your assignment put in a file named README in the .c directory. Only one group member is to handin the code.

2. To have an assignment marked, each group, or individual, is to handin one paper version of the the filled in marking sheet (see next page) along with a typed, paper version of write-up. If you choose not to do a write-up you must still handin the cover page. If you don't handin this sheet your assignment will not be marked. These are due the start of class on Wednesday March 7th.

CPSC 415: Assignment 2

Student Name	Student Number	Student Signature

Kernel Used:

	Mark	
Functionality		/54
Modified process creation	/4	
Auxiliary System Calls	/4	
Messaging System	/10	
pre-emption	/4	
Idle Process	/3	
Sleep Device	/7	
Producer/Consumer	/6	
Test Cases	/16	
Code Documentation		/15
Code clarity	/10	
Function descriptions	/5	
Questions		/13
Question 1	/2	
Question 2	/4	
Question 3	/2	
Question 4	/2	
Question 5	/3	
Total		**/82**