**CSci 5103 (Spring 2016)**

John Erickson   
eric0870  
2336359  
UNITE Student

**Assignment 1**

**Part A (100 points)**

**Due February 1, 2016**

*This part of the assignment must be done* ***individually****.*

**Problem 1 (10 points):** Which of the following instructions should **NOT** be allowed in user mode?

a) Disable all interrupts.

b) Read the time-of-the-day clock.

c) Set the time-of-the day clock.

d) Change the base-bound register values.

e) Set a timer.

f) Turn off timer interrupt.

g) Read from kernel memory.

h) Write to kernel memory.

i) Fetch an instruction from kernel memory.

j) Switch from user to monitor mode.

The only instruction above that should be allowed in user mode is the switch from user to monitor mode. This is handled by hardware via a kernel interrupt. Some of these instructions may be available to the user program via System calls, which would transition the mode from user to kernel prior to implementing the instruction. These instructions include reading and setting the time of day clock. The rest of the instructions from the list should not be available to user programs.

**Problem 2 (20 points):** What are the key differences between traps, system calls, exceptions, and interrupts?

First some definitions:

Exception: An Exception is an event triggered by a program instruction. An Exception interrupts the current program execution and needs to be serviced by the kernel. Examples of Exceptions include error conditions (divide by zero, illegal memory access, etc) and non-error conditions (page faults, etc).

Trap: A Trap is a special type of Exception. The kernel users a Trap instruction to switch from user mode to kernel mode in order to execute a System Call.

System Call: System Calls make up the interface between the kernel and the user program. They provide a mechanism for the kernel to manage system resources for user programs without allowing the user program direct access to the resources. Common features supported via System Calls include process management, file management, memory allocation, etc.

Interrupt: An interrupt is an event triggered by hardware. Interrupts may be sourced internally (timer, FIFO level trigger, etc) or received from an external device. Regardless of the source, an interrupt occurs asynchronous to the executing program. Interrupts are routed to an interrupt handler to be dispositioned. Depending on its priority, an interrupt may be serviced immediately or queued by the interrupt handler.

Key differences between traps, system calls, exceptions and interrupts include:

* Exceptions, Traps, and System Calls are all initiated by program instructions, as opposed to interrupts which occur asynchronous to programs and don’t necessarily have any relation to the currently executing program.
* Interrupts are serviced in the system context as opposed to exceptions where are serviced in the process context.

**Problem 3 (21 points):** For each of the system calls, identify at-least 3 conditions that cause it to fail: *fork*, *open*, *exec*, *waitpid*, *read, write*, *unlink,*

|  |  |
| --- | --- |
| **System Call** | **Failure Condition** (from Linux man pages man7.org) |
| fork | * there is not sufficient memory to copy the parents data for the child * the total number of processes for the user ID exceeds the max allowed * the fork() call is not supported by the hardware |
| open | * invalid input arguments * access to the requested file is not allowed * max number of files open exceeded by process |
| exec | * invalid input arguments * user does not have permission to execute desired file * file pointer argument is outside the users accessible address space |
| waitpid | * invalid input arguments * process specified by pid argument does not exist or is not a child of calling process * WNOHANG was not set and an unblocked signal or a SIGCHLD was caught |
| read | * invalid input arguments * file descriptor argument is not a valid file descriptor or is not open for reading * buffer pointer argument is outside the users accessible address space |
| write | * invalid input arguments * file descriptor argument is not a valid file descriptor or is not open for writing * buffer pointer argument is outside the users accessible address space |
| unlink | * write access to the directory containing the file pointed to by the pathname argument is not allowed for user * the file pointed to by pathname argument is in use * the pathname argument is outside users accessible address space |

**Problem 4 (30 points):** A file whose file descriptor is fd contains the following sequence of bytes:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f (length = 17 including terminating newline char)

The program makes the following sequence of system calls (assume all variables have been initialized properly):

// set fd offset to 3, subsequent read() call points to ‘3’ (byte 4)

1 lseek (fd, 3, SEEK\_SET);

2 read (fd, &buffer1, 4); // read 3456 into buffer1, increment offset by 4

3 fprintf (stderr, "%s", buffer1); // print: 3456

// set fd offset to current + 3, subsequent read() call points to ‘a’ (byte 11)

4 lseek (fd, 3, SEEK\_CUR);

5 read (fd, &buffer2, 4); // read abcd into buffer2

6 write (2, buffer2, 4); // print to shell: abcd

7 lseek (fd, 10, SEEK\_END); // set fd offset to end of file + 10 (byte 27)

8 write(fd, buffer1, strlen(buffer1)); // write 4 bytes from buffer1 to file at byte 27

a. What is the output after executing lines 3 and 6?

3456abcd

b. What is the difference between *fprintf()* and *write()*? What does the operating system do for each function?

fprintf() is included in C Standard Library stdio.h and provides high level features like character conversions, while write() is a C System Call and simply copies bytes from a buffer to a stream.

The fprintf() instruction is handled inline with the executing process, there is no kernel intervention required.

For a write() instruction, the OS

* creates an activation record on the process stack,
* loads the code for the write system call into a register and executes the trap instruction,
  + causes switch to kernel mode
* kernel trap handler executes write system call
* after write function completes, kernel returns control back to caller, switching back to user mode

c. List the contents of the file after line 8 has finished execution. What is the length of the file?

0123456789abcdef\0\0\0\0\0\0\0\0\03456

**Problem 5 (10 points):** What are the advantages and disadvantages of implementing threads in user space?

From Tanenbaum 2.2.4

Advantages of threads in user space

* multi-threaded programs can be implemented on OS that doesn’t support threads
* thread switching doesn’t require trapping to the kernel, saving cycles
* each process can have a custom scheduler
* scalability, a large number of threads is more easily accommodated in the user space

Disadvantages of threads in user space

* using blocking system calls can block the entire process
* a page fault will result in the entire process being blocked
* without a clock interrupt to aid in scheduling, a single thread can run forever
* the types of applications that lend themselves to multi-threading often require numerous system calls, adding overhead of select system calls to avoid process blocking

**Problem 6 (9 points)**: What is the difference between hyperthreading and multi-core CPUs?

Hyperthreading is a means of creating virtual processing units from a single CPU, to appear as multiple CPU’s to the OS. The virtual CPU’s share the same physical resources, thus only can run at a time. This still improves performance, maximizing physical CPU resources. Multi-core CPU’s introduces multiple physical processor cores on the same chip. With each core processing instructions simultaneously and independently, this allows for true parallelization of processing.

**CSci 5103 (Spring 2016)**

**Assignment 1 Part B (100 points) Due February 1, 2016**

*This part of the assignment must be done* ***individually****.*

**Objective:**

The objective of this assignment is to understand the following topics:

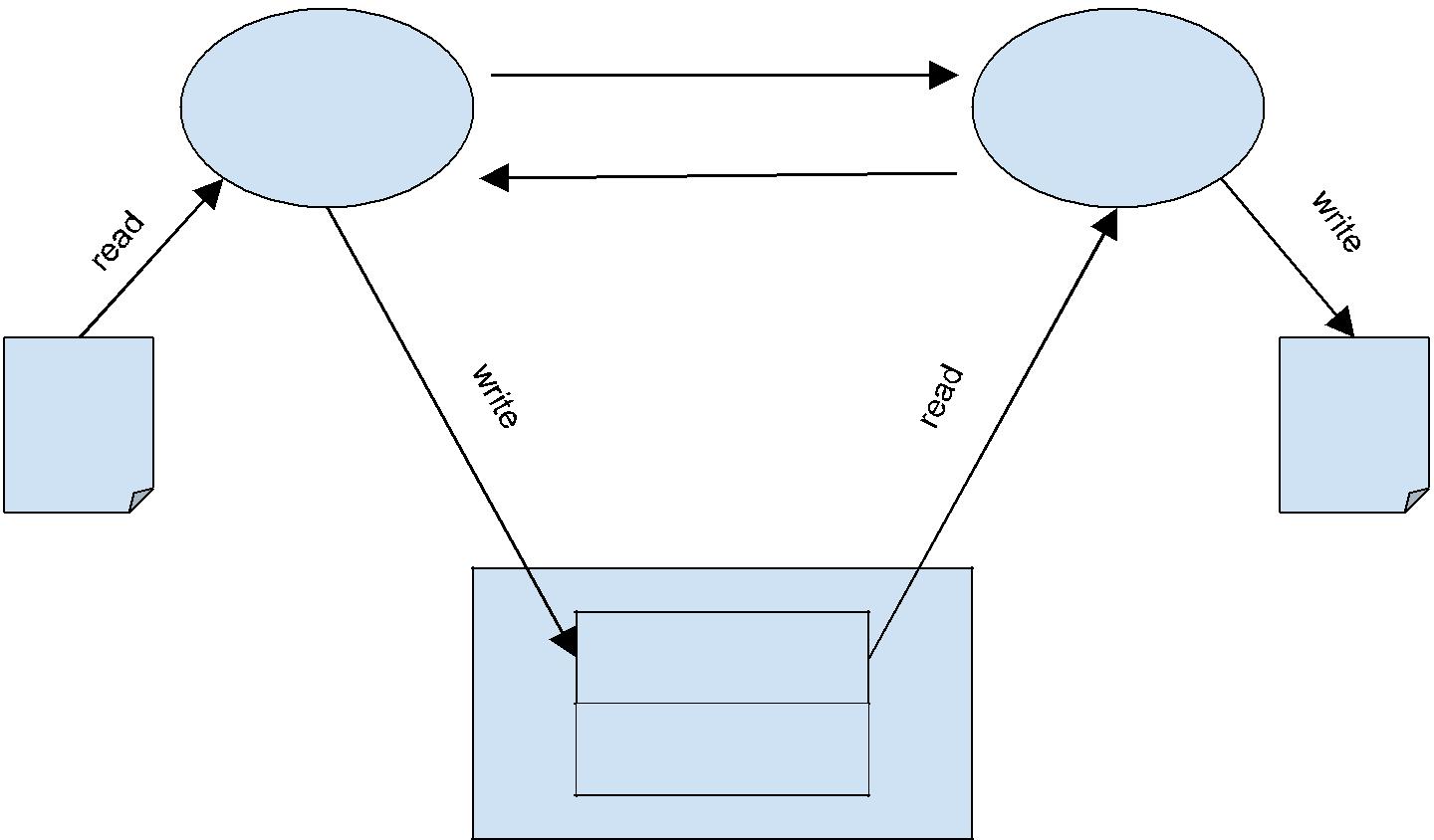
* Process creation and management on a UNIX system using system functions such as *fork(),* *exec(), waitpid().*
* Shared memory operations using *shmget(), shmat(), shmdt()*
* Signal handling using *sigemptyset(), sigaddset(), sigprocmask(), sigwaitinfo(), sigqueue()* and other related functions
* Timing operations

**Problem statement:**

In this assignment you will create a region of shared memory between two processes to act as a medium for data transfer. These two processes will communicate in the producer-consumer mode. The producer process will read a file, transfer its contents to the consumer through the shared memory, and the consumer process will store the received data in a new file called **output**. Your program also needs to measure and report the total time taken for data transfer. Your programs must use POSIX real-time signals for communication and synchronization.

The shared memory region must contain the following:

**buffer**: A character array of size **1024 bytes**, used to store the data to be transferred. **time**: Timestamp indicating the time of insertion



real-time signal

producer consumer

real-time signal

|  |  |
| --- | --- |
| input | output |
| file | file |

buffer

time



shared memory

The producer process must read data from a file and write it into the buffer in the shared memory region. It may be necessary to repeat this step multiple times as the file may be much larger than the size of buffer. The producer process must also obtain the current time and store it in the shared memory region. The producer process must then send a real-time signal to the consumer process to indicate that the buffer is full and ready for consumption. The producer process must also send the size of available data in shared memory to the consumer process along with the signal. The producer process must then wait to receive a real-time signal from the consumer process before proceeding. After the entire file has been transferred, the producer process must send a real-time signal to the consumer process along with an accompanying integer value of -1 to indicate completion of data transfer. The producer process must then wait for the consumer process to terminate, report its exit status and then exit gracefully.

The consumer process is responsible for storing the received data in a file called ‘**output’.** The consumer process must wait to receive a real-time signal from the producer process. The reception of the signal indicates that either data is available for consumption or that data transfer is complete. If the consumer process receives an integer whose value is non-negative, it indicates that the buffer has valid data for consumption. It must read those many bytes of data from the buffer and write it to the output file. It must also measure the total time the buffer contains valid data. It may do so by aggregating the time difference between the current time and the time stored in the buffer. The consumer process must then send a real-time signal to the producer process to indicate that the buffer is empty. If the accompanying integer value is -1, it indicates end of data transfer. At this point, the consumer process must report the total time taken and exit gracefully.

Write a C program named prodcon.c which contains code for both the producer and the consumer processes. When executed, the program must create a shared memory segment and fork a child process. The parent process should act as a producer and the child process should act as the consumer. The input file would be given as a command line argument to the program. Provide a makefile which can compile your program to produce an executable called **prodcon**.

**Usage:**

$ prodcon <input\_filename>

**Recipe:**

The general structure of the producer would be as follows:

...

Open the file specified by argv[1] for reading

…

while (not end of file) {

read a chunk of data into memory read the current time

write the chunk into the shared memory write the timestamp into the shared memory send a real-time signal to the consumer

suspend execution until a real-time signal is received

}

// data transfer complete

send a real-time signal to the consumer with an accompanying integer value of -1 wait for the consumer process to terminate

...

exit

End of producer

The general structure of the consumer would be as follows:

…

Open the file “output” for writing

…

while(...) {

suspend execution until a real-time signal is received

store the integer value sent along with the real-time signal in a variable called ‘size’

if ‘size’ is -1: Close the output file, report the aggregate time taken for download and exit gracefully

read ‘size’ bytes from the shared memory region read the current time (A)

read the timestamp from the shared memory region (B) compute the time difference between (A) & (B) and aggregate

send a real-time signal to the producer process with an accompanying integer value of 0

}

End of consumer

**Grading criteria:**

Process creation and management - fork/wait/waitpid: **20%**

Shared memory operations: **20%**

Complete data transfer: **25%**

Timing computation and reporting: **10%**

Correct use of real-time signals: **20%**

Comments/Documentation: **5%**

**Note:**

Please ensure that your program can run on the  [CSELabs UNIX machines](http://cselabs.umn.edu/labs/unix_machines).

You will lose significant points for not following instructions, failing to submit a makefile or a readme file.

You will also lose significant points if the output file does not match the input file (Hint: Use the ‘diff’ command to compare your input and output files)

**Submission Instructions:**

Submit a single archive file (.zip/.tar/.rar) containing

* A PDF document for your answers to Part A
* The source files, header files and a Makefile for Part B.
* A README file containing your name, your x500 and the CSELabs machine on which you tested your code.