CS 537: Introduction to Operating Systems (Summer 2017)

University of Wisconsin-Madison
Department of Computer Sciences

Midterm Exam 1

July 7, 2017

3 pm - 5 pm

There are **twelve (12) total numbered pages** with **ten (10) questions**.

There are many easy questions and a few hard questions in this exam. You may want to use a **easiest-question-first scheduling policy**. This will help you to answer most questions on this exam without getting stuck on a single hard question.

Good luck with your exam!

Please write your **FULL NAME** and **UW ID** below.

NAME:_____

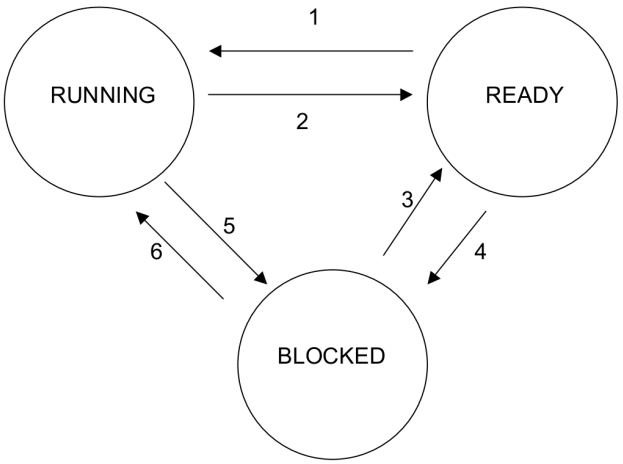
UW ID:_____

Grading Page

Question	**Points Scored**	**Maximum Points**
1		10
2		10
3		10
4		10
5		10
6		10
7		10
8		10
9		10
10		10
Total		**100**

1. Process: State Transitions

What conditions cause a process to transition between the 3 states shown below?
e.g., If one of the arrows indicates that a process is being descheduled, then you are expected to write what condition(s) may cause the process to be descheduled. If a particular state transition doesn't happen, label it N/A and explain why it doesn't happen.



Arrow 1:

Arrow 2:

Arrow 3:

Arrow 4:

Arrow 5:

Arrow 6:

2. System Calls vs Timer Interrupts

The following are the list of events (in no particular order) that may happen when a system call or a timer interrupt happens while a user process is executing.

a. Process A: trap into OS via int $64
b. Process A: calls read()
c. Process B: continues execution
d. OS: return from trap (into B)
e. OS: handle trap
f. OS: call switch routine
g. OS: send read to disk
h. OS: restore registers(B) from PCB(B)
i. OS: switch to kernel-stack (B)
j. OS: save registers(A) to PCB(A)
k. OS: put A to sleep (i.e. A's state = blocked)
l. Hardware: restore registers(B) from kernel-stack(B)
m. Hardware: jump to B's PC
n. Hardware: save registers(A) to kernel-stack(A)
o. Hardware: timer interrupt
p. Hardware: move to user mode
q. Hardware: jump to trap handler
r. Hardware: move to kernel mode

What are the events that may happen as per the limited direct execution protocol? Choose and **sort** them in order for the following two scenarios. Just write the **lower-case letters** for each event in order.

1. Process A switches to B, because it calls **read()** and **blocks**.

2. Process A switches to B, because a **timer interrupt** happens.

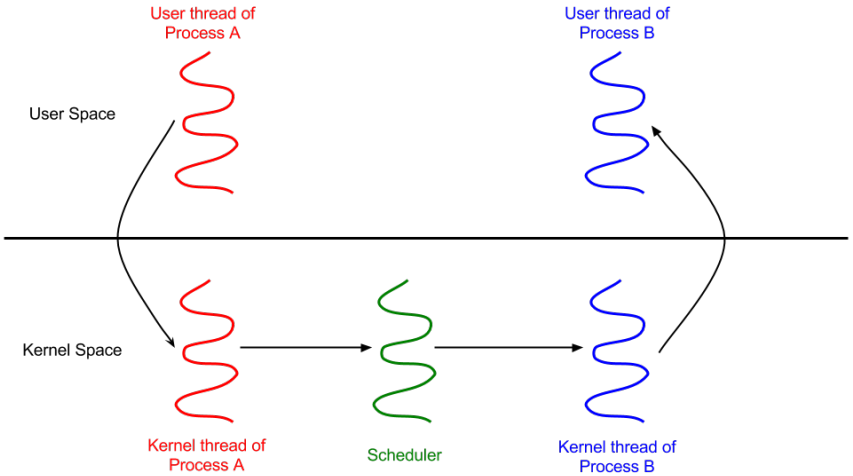
3. Context Switch

a. When a context switch happens (e.g. from process A to process B), the hardware saves the *registers(A)* to *kernel-stack(A)* (in a structure called *trap frame*) and the OS stores the *registers(A)* to *PCB(A)*. Similarly, to begin executing process B, the OS restores *registers(B)* from the *PCB(B)* and the hardware restores the *registers(B)* from the *kernel-stack(B)*. Why do **both** the hardware and the OS save/restore the registers of a process during a context switch?

b. The control flow during a context switch from process A to process B is shown in the figure below.

(a) How many times during this process does the value of **stack pointer** (%esp register) change?

(b) What are the different stacks that the stack pointer points to? You may assume that initially the stack pointer was pointing to process A's user stack.



4. Basic Scheduling

Assumptions: Processes A, B, and C are **CPU-bound** (no I/O). There is **no overhead** for context switching. In round robin scheduling, when a **new process** enters the system, it is placed at the **front** of the **round robin queue**.

a. Different scheduling policies have different behaviors. Can you tell them apart by looking at their schedules only? All policies below are scheduling the same set of processes. The workload is shown below.

Process	Arrival Time	Service Time
A	0	5
B	1	3
C	2	1

Abbreviations: **FIFO** - First In First Out, **SJF** - Shortest Job First, **STCF** - Shortest Time to Completion First, **RR** - Round Robin.

Write the abbreviations of a policy on the side of its corresponding schedule for the given workload.

Schedule	Policy
A B C B B A A A A	
A A A A A C B B B	
A B C A B A B A A	
A A A A A B B B C	

b. Calculate the **schedule**, **average turn around time** and the **average response time** for the workload given below under different scheduling policies.

Process	Arrival Time	Service Time
A	0	2
B	1	5
C	2	2

Policy	Schedule (e.g., ABCABC)	Avg. Turnaround Time	Avg. Response Time
FIFO			
SJF			
STCF			
RR			

5. MLFQ

Assumptions:

1. Processes A, B, and C are **CPU-bound** (no I/O).
2. There is **no overhead** for context switching.
3. When a **new process** enters a particular queue, it is placed at the **end** of the round robin queue.

Consider a **Multi-Level Feedback Queue (MLFQ)** Scheduler with **3 queues** as shown below. After a time interval of **10 seconds**, all processes are **boosted up** to the **top-most queue** with the highest priority.

queue #	priority	Round Robin time slice
2	2 (highest)	1
1	1	2
0	0 (lowest)	3

The table below gives the details of the **workload**.

Process	Arrival Time	Service Time
A	0	10
B	3	3
C	5	4

a. Fill the table below with the details of which process will be scheduled for each time unit and at which priority level. In this table, for each time unit n, write which process will be scheduled from time n to n + 1. e.g., For the cell with time unit 0, fill in the name of the process that will be scheduled from time 0 to 1.

	Time																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Priority 2																	
Priority 1																	
Priority 0																	

b. What is the turn around time and the response time for each of the three processes in the above schedule? Fill in the values in the table below.

Process	Turnaround Time	Response Time
A		
B		
C		

6. fork, exec, wait!

Assumptions:

1. The 2 programs (parent.c, child.c) are present in a directory named **/home/cs537**.
2. The programs are compiled from the same directory **/home/cs537** as shown below.
   $ gcc -o parent parent.c -Wall
   $ gcc -o child child.c -Wall
3. An executable named **bad** is NOT present anywhere in the computer and hence it's an invalid program name.
4. **pwd** is a valid program which prints the current working directory.
5. The 2 programs **compile and execute successfully** without any errors/warnings.
6. You may assume that the necessary **header files** are included even though they are not shown here.
7. You may assume that **fork()** always succeeds.

parent.c

```
int main() {
    printf("Madras\n");
    int rc = fork();
    if (rc == 0) {
        char *args[2];
        args[0] = "./child";
        args[1] = NULL;
        execvp(args[0], args);
        printf("Bangalore\n");
        exit(1);
    } else if (rc > 0) {
        wait(NULL);
        printf("Leh\n");
    }
    return 0;
}
```

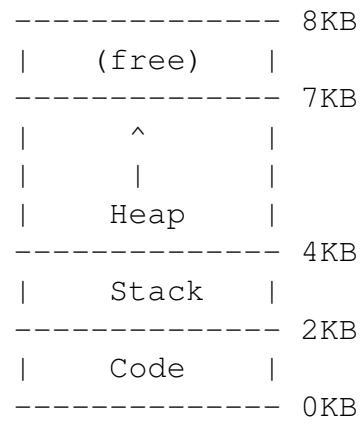
child.c

```
int main() {
    printf("Madison\n");
    int rc = fork();
    if (rc == 0) {
        char *myargs[2];
        myargs[0] = "./bad";
        myargs[1] = NULL;
        execvp(myargs[0], myargs);
        printf("SFO\n");
        myargs[0] = "pwd";
        execvp(myargs[0], myargs);
    } else if (rc > 0) {
        wait(NULL);
        printf("NYC\n");
    }
    return 0;
}
```

What is the **output** when we run **./parent** from the directory **/home/cs537**?

7. Base and Bounds

Consider a machine that uses the **base and bounds technique** for memory virtualization and a slightly **different address space** as shown below. Note that the **stack** is of fixed size and it immediately follows the code section. The heap starts at the end of the stack and grows upward. In this configuration, there is only one direction of growth, towards higher regions of the address space. The figure below is NOT drawn to scale.



Parameters:
Size of the virtual address space = 8KB, size of the physical memory = 32KB, base register = 16KB, and bounds register = 7KB. References to any address within the bounds would be considered legal; references above this value are out of bounds and thus the hardware would raise an exception.

a. For each **virtual address**, either write down the **physical address** it translates to OR write down that it is a **segmentation fault** (an out-of-bounds address).

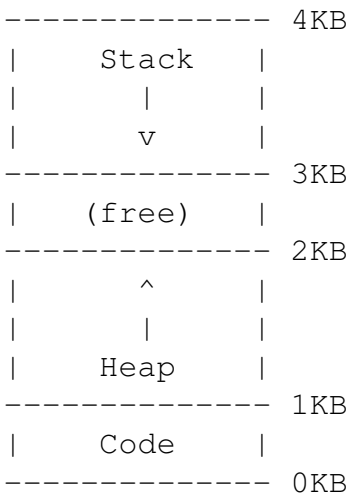
Virtual Address	**Physical Address (in hex or decimal) OR Seg Fault**
0x1000 (decimal: 4096)	
0x1C00 (decimal: 7168)	
0x0000 (decimal: 0)	
0x2000 (decimal: 8192)	
0x0800 (decimal: 2048)	

b. For each physical address, write the corresponding virtual address (in hex or decimal), and the logical segment (code, stack, heap) that the virtual address belongs to.

Physical Address	**Virtual Address**	**Segment**
0x5800 (decimal: 22KB)		
0x4400 (decimal: 17KB)		
0x4C00 (decimal: 19KB)		

8. Segmentation

Assume a machine that uses **segmented virtual memory**. There are **three segments** namely, code, heap, and stack. Note that the stack and the heap grow towards each other, and the stack grows towards lower addresses.The **top two bits** (MSBs) in the virtual address are used to identify the segment. 00 for code, 01 for heap, and 11 for stack. The figure below is NOT drawn to scale.



Parameters:
Size of virtual address space = 4KB, size of physical memory = 8KB,
size of stack segment = 1KB, size of heap segment = 1KB
size of code segment = 1KB.

Assume that the following virtual to physical address translations are valid.

Virtual Address	**Physical Address**
0x064 (decimal: 100)	0x1064 (decimal: 4196)
0x7D0 (decimal: 2000)	0x1FD0 (decimal: 8144)
0xFA0 (decimal: 4000)	0x0FA0 (decimal: 4000)

Find the value(in hex or decimal) of the **base** & **bounds** register for the 3 segments.

Segment	**Base Register**	**Bounds Register**
Code		
Heap		
Stack		

9. Paging

In this question, we consider address translation in a system with a simple linear page table (an array of Page Table Entries or PTEs).

Parameters:

- Size of virtual address space = 32KB
- Page size = 4KB
- Size of physical memory = 64KB

Answer the following questions based on the parameters described above.

a. Number of **bits** needed for the Virtual Page Number (**VPN**): ————————

b. Number of **bits** needed for the Physical Frame Number (**PFN**): ————————

c. Number of **bits** needed for the **offset**: ————————

d. Number of **virtual pages** in a process' address space:————————

e. Number of **physical frames** in this machine's physical memory: ————————

f. The **page table** for a process in this machine is given below. The **format** of the page table is simple. The **high-order (left-most) bit** is the **VALID** bit. If the bit is 1, the rest of the entry is the PFN. If the bit is 0, the page is not valid.

Page Table (from entry 0 down to the max size)

0x8000000C
0x00000000
0x00000000
0x80000006
0x80000004
0x00000000
0x8000000E
0x80000009

For each **virtual address**, write down the **physical address** it translates to OR write down that it is **a segmentation fault** (an out-of-bounds address).

Virtual Address	**Physical Address (in hex or decimal) OR Seg Fault**
0x3F2D (decimal: 16173)	
0x1F38 (decimal: 7992)	
0x0CE3 (decimal: 3299)	
0x5C1F (decimal: 23583)	

10. TLBs

The following are the sequence of steps that take place during each memory reference. You may assume that the system uses a linear page table and has a Translation-Lookaside Buffer **(TLB) that is managed by the Operating System**.

a. Mark (by **circling** the correct answer), in the timeline below, which steps are done by the Operating System (OS) and which are done by the Hardware (HW).

Extract the Virtual Page Number (VPN) from the Virtual Address (VA)	HW	OS
Check if the TLB holds the translation for this VPN	HW	OS
On a **TLB hit**, extract the Page Frame Number (PFN) from the relevent TLB entry	HW	OS
Concatenate the PFN with the offset from the original VA and form the Physical Address (PA)	HW	OS
Access the memory location corresponding to the PA	HW	OS
On a **TLB miss**, raise a TLB miss exception	HW	OS
Run the TLB miss trap handler	HW	OS
Check Page Table (PT) for Page Table Entry (PTE)	HW	OS
Extract the PFN from the PTE	HW	OS
Update the TLB using special privileged instructions	HW	OS
Return from trap	HW	OS
Retry the **current** instruction	HW	OS

b. When a context switch happens on a system with a TLB (assuming the TLB only has a valid bit, VPN, and PFN fields), then the contents of the TLB become invalid for the next process that is about to run.

i. What support does the **OS** provide for context switching to work with TLBs?

ii. What support does the **hardware** provide for context switching to work with TLBs?