COM S 352: Introduction to Operating Systems
Midterm Exam
Spring 2023

Cover Sheet

Student Name: Answers

_____

Format:

- Time: 60 mins
- Points: 100
- Question Types: matching, true/false and short answer

Instructions:

- You may use 1 (one) letter sized sheet of paper (front and back), that you have prepared yourself with notes before the exam, as a "cheat sheet" during the exam.
- You may not consult classmates, electronic devices or resources other than the cheat sheet during the exam.
- Questions of clarification should be asked directly to an instructor or TA.

Question	**Points**
1	/24
2	/30
3	/9
4	/6
5	/7
6	/8
7	/6
8	/10
Total	/100

1. (24 pts, 3 pts each) For each description on the left, select the best matching term on the right, each term is used only once but some will not be used.

_P__ has the job of loading a page from the swap space to physical memory

_I__ causes a trap

_J___ code only one thread is allowed to execute at a time

_O__ describes a child process whose parent process terminates before itself

_D__ where frames are located

_H__ a space allocated when a new thread is created

_L__ technique to use the CPU efficiently

_B__ a scheduling policy that has the goal of shortest turnaround time

A. FIFO
B. SJF
C. RR
D. physical memory
E. semaphore
F. scheduler
G. condition variable
H. stack
I. making a system call
J. mutual exclusion
K. address space
L. multiprogramming
M. lottery
N. zombie
O. orphan
P. page fault handler

2. (30 pts, 2 pts each) Which of the following statements are true? Write T or F for true or false.

_F__ Running a multi-thread process requires a computer with multiple CPU cores.

_F__ Increasing the level of multiprogramming is a way to reduce thrashing.

_F__ Reading a pipe without any content returns NULL immediately.

_F__ FIFO is a commonly used replacement policy in modern Operating Systems.

_F__ The URL algorithm replacement policy takes advantage of a feature found on some computer hardware.

_T__ The address space of a process may contain multiple stack sections.

_F__ TLB is used for managing free space in physical frame.

_T__ In the CFS scheduler, when the virtual runtime of process is weighted to increase more rapidly than its real runtime, the scheduler is less likely to pick it to run.

_F__ Calling sem_wait() on a semaphore always results in a thread being blocked.

_T__ A process being removed from running when its time slice (quantum) has expired is an example of preemption.

_T__ Threads in the same process share the same heap section.

_T__ A loop reading from randomly chosen indexes of an array, which is a couple of pages in size, is an example of spatial locality.

_F__ A benefit of larger page size is a reduction in internal fragmentation.

_F__ The TestAndSet machine instruction available on some CPUs is used to avoid spinning loops.

_T__ When using a SJF (shortest job first) scheduling policy, it is possible for a process to prevent other jobs from running by staying in the CPU for as long as it wants.

3. (9 pts) Consider the following set of jobs, with arrival times and the length of CPU bursts (job runtimes) given in milliseconds.

	Arrival Time	CPU Burst
A	0	15
B	2	15
C	4	8
D	6	5

Suppose a scheduler produces the schedule shown in the Gantt chart below:

| A | C | D | C | A | B |

0 4 6 11 17 28 43

a) The average response time is __6.5___. (show calculation below)

(0 + (28-2) + (4-4) + (6-6))/4 = 26/4 = 6.5

b) The average turnaround time is __21.75__. (show calculation below)
((28-0) + (43-2) + (17-4) + (11-6))/4 = (28+41+13+5)/4 = 21.75

c) The scheduling policy that is most likely being used is _STCF___. (select FIFO, SJF, STCF or RR)

4. (6 pts) Recall what we did in Project 1.B. Which of the following steps are necessary in order to add a new system call X to xv6-riscv?
_____a,b,c,e,g,h_____

a) declare a system call number in syscall.h

b) declear a function named sys_X in syscall.c

C) add sys_X to syscall table in syscall.c

d) make change to struct proc in proc.h

e) implement sys_X in proc.c or some other program file in the kernel folder

f) update Makefile

g) add an entry for X to usys.pl

h) declare system call X to user.h

i) add a program file that calls system call X to user folder

5. (7 pts) Consider a TLB and a page table below. Assume a page size of 2KB (i.e., 2,048 bytes).

TLB

VPN	PFN	Valid
1	2	1
2	0	0

Page Table

VPN	PFN	Valid
0	3	1
1	2	1
2	1	1
3	0	0
4	0	0
5	8	1
6	12	1

What is the physical address in memory that will be read when a program reads a byte from virtual address 4100? (must show full work, answer can be left in formula form)

Page #: 4100/2048 = 2, offset: 4100 % 2048 = 4

TLB miss (entry for VPN#=2 is not valid), look up page table find: PFN = 1

Physical addres: 1*2048 + 4 = 2052

6. (8 pts) Given the reference string of page accesses:

3 2 0 1 3 0 2 3

and a physical memory with 3 page frames, how many page faults result when using the LRU replacement policy? Show your work.

The number of page fault is _____6_____ (show work below)

3	2	0	1	3	0	2	3
M	M	M	M	M	H	M	H
	3	3	3	2	0	1	2
		2	2	0	1	3	3
			0	1	3	0	0

7. (6 pts) Let first() and second() be two functions executed by threads thrd1 and thrd2 respectively. Complete the code below (following "**To do**" lines) to make sure thrd2 starts executing second() only after thrd1 has completed executing first(). You are free to use mutexes, condition variables, semaphores, or any combinations of them in the code.

Note: the answers in red and blue are two versions

```
//To do: declare shared variables, if applicable
sem_t sem;
pthread_mutex_t mutex;
Int flag;
pthread_cond_t cond;

//To do: add code to initialize the shared variables, if applicable
sem_init(&sem, 0);
mutex = PTHREAD_MUTEX_INITIALIZER;
flag=0; //first() hasn't executed
cond = PTHREAD_COND_INITIALIZER;

//routine for Thrd1
void *thrd1(void *arg){
        //To do: add code that should be run before calling first(), if applicable

        first();
        //To do: add code that should be run after calling first(), if applicable
        sem_post(&sem);
        pthread_mutex_lock(&mutex);
        flag=1;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
}

//routine for Thrd2
void *thrd2(void *arg){
        //To do: add code that should be run before calling second(), if applicable
        sem_wait(&sem);
        pthread_mutex_lock(&mutex);
        while(flag==0) pthread_cond_wait(&cond, &mutex);
        pthread_mutex_unlock(&mutex);

        second();
        //To do: add code that should be run after calling second(), if applicable
}
```

8. (10 pts) Consider an extended version of Producer/Consumer Problem:

- 3 buffer spaces are shared by producers and consumers (initially they are all empty);
- each producer generates 2 items at a time;
- each consumer consumes 1 item at a time.

Complete the code below (following "**To do**" lines) to ensure:

- mutual exclusion in buffer access;
- a producer cannot put any of its 2 items unless until at least 2 buffer spaces are empty;
- a consumer cannot get an item if all spaces are empty.

You are free to use mutexes, condition variables, semaphores, or combinations of them.

```
//declare shared variables

        int buf[3];

        //To do: declare more shared variable(s) if applicable

        Sem_t mutex, full, empty;



//initialize shared variables

        for(int i=0; i<3; i++) buf[i]=0;

        //To do: add more initialization if applicable

        Sem_init(&mutex,0);

        Sem_init(&full,0);

        Sem_init(&empty,3);


//function for put, which should be called only when at least 2 buffer spaces are empty

void put() {
```

```
        int n = 0;
        for(int i=0; i<3 && n<2; i++) {
                if(buf[i] == 0) {
                        buf[i]=1;
                        n+=1;
                }
        }
}

//function for get, which should be called only when at least one buffer space is available
        void get() {
                for(int i=0; i<3; i++) {
                        if(buf[i] == 1) {
                                buf[i]=0;
                                break;
                        }
                }
        }


//Producer thread's routine
        void *producer(void *arg) {
                while(1) {
                        //To do: add code executed before putting its items to buffer, if applicable
```

```
                sem_wait(&empty);
                sem_wait(&empty);
                sem_wait(&mutex);
                put();
                //To do: add code executed after putting its items to buffer, if applicable
                sem_post(&mutex);
                sem_post(&full);
                sem_post(&full);
        }
    }
//Consumer thread's routine
    void *consumer(void *arg) {
        while(1) {
            //To do: add code executed before getting an item from buffer, if applicable
            sem_wait(&full);
            sem_wait(&mutex);
            get();
            //To do: add code executed after getting an item from buffer, if applicable
            sem_post(&mutex);
            sem_post(&empty);
        }
    }
```