Sessional Quiz

Date: 22/02/2023

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

L-3/T-II CSE 314 Operating Systems Sessional

Time: 50 minutes **Marks: 60**

Student Name: _____ Student ID: _____

1. Write the correct answer in the assigned box. $(5 \times 1 = 5)$

(a)	(b)	(c)	(d)	(e)

Write your answers in the table above.

(a) How many levels does an xv6 pagetable have?
   (i) 1
   (ii) 2
   (iii) 3
   (iv) 4

(b) What is the maximum size of virutal address space can xv6 support?
   (i) 2^{64} bytes
   (ii) 2^{54} bytes
   (iii) 2^{49} bytes
   (iv) 2^{39} bytes

(c) How much RAM does xv6 have by default?
   (i) 64 MB
   (ii) 128 MB
   (iii) 256 MB
   (iv) 512 MB

(d) In which mode do system calls run in xv6?
   (i) Kernel
   (ii) User
   (iii) Machine
   (iv) Supervision

(e) In default xv6, what happens when a process requires more memory than available RAM?
   (i) RAM size is increased
   (ii) The process sleeps until more memory is available
   (iii) Kernel panics
   (iv) Some pages are swapped to disk

2. xv6 does not keep any username. You have to implement a very simple system to read and update the username. The default username should be *'xv6'*. You can assume a username cannot be longer than 32 characters.

The following user program reads and writes the username.

```
//...headers...
int
main()
{
    char uname[100];
    read_uname(uname); // sys-call reading username from kernel
    printf(uname); // prints "xv6" when first executed
    strcpy(uname, "myxv6");
    write_uname(uname); // sys-call writing username to kernel
    read_uname(uname); // reading username from kernel
    printf(uname); // prints "myxv6"
    return 0;
}
```

Now, complete the following program. [Focus more on the general setup than exact names and argument types of required xv6 functions.]

```
// Write any global variables required here



// The following system-call copies username from kernel to userspace
void
sys_read_uname()
{









}
// The following system-call copies username from userspace to kernel
void
sys_write_uname()
{









}
```

3. Consider the two following functions run by two individual threads: (8)

Thread 1	Thread 2

void* printA()
{
while (1)
printf("A");
}
	void* printB()
	{
	while (1)
	printf("B");
	}

Add semaphores to the code such that at any moment the number of A or B differs by at most 1. The solution should allow (but not be limited to) strings such as: ABBAABBABA....

Do not forget to indicate the initial value of the semaphores.

4. The following is a set of three interacting processes that can access two shared semaphores: (2+2+2+3+3 =

semaphore U = 3;

semaphore V = 0;

Process 1	Process 2	Process 3

L1:
wait(U);
printf("C");
post(V);
goto L1;
	L2:
	wait(V);
	printf("A");
	printf("B");
	post(V);
	goto L2;
		L3:
		wait(V);
		printf("D");
		goto L3;

Answer the questions below assuming that once execution begins, the processes will be allowed to run until all 3 processes are stuck in a wait() statement, at which point execution is halted.

i Assuming execution is eventually halted, how many C's are printed when the set of processes runs? Why?

ii Assuming execution is eventually halted, how many D's are printed when this set of processes runs? Why?

iii What is the smallest number of A's that might be printed when this set of processes runs?

iv Is CABACDBCABDD a possible output sequence when this set of processes runs? Why/why not?

v Is CABABDDCABCABD a possible output sequence when this set of processes runs? Why/why not?

5. You have designed and coded an $\mathcal{O}(n^2)$ naïve algorithm in `naive.py` that always produces correct results to a problem. Now, to optimize, you have designed and coded another $\mathcal{O}(n \log n)$ algorithm in `rareBug.py` which you suspect, is not always right and may produce wrong output sometimes.

   Now you want to find out one such rare case where your optimized solution fails (i.e., produces different output than your naïve solution). You wrote a program `generator.py` that generates random valid (small enough for your naïve algorithm) input on each run. Your task is to write a bash program that will run until it finds an input from the `generator.py` that produces a different result from the two solutions. When your program exits, it should print the number of attempts it needed to find the failing input and the failing input should be saved in a file named `input.txt`. You can assume that all the files are in the same directory where your shell script lives. You might find the `diff -q file1 file2` command helpful that returns zero when two files match exactly, non-zero otherwise. (5)

6. You have a CPU-intensive program `intense.py` that you want to run 1000 times in your machine. In order to get the most out of your multi-core system, you do not want to launch them one by one, so you thought of launching all of them in parallel using the following script.

```
for i in {1..1000}; do
    python intense.py & # & makes the process run in background without blocking
done
```

But, the load is too much for your machine and you noticed it is spending most of its time in context switching. So, you decided to limit the maximum number of parallel execution to 5, i.e., at any point of time, a maximum of 5 instances of `python intense.py` should exist. So, you should launch 5 of them, wait for all of them to finish, then launch another 5 of them simultaneously and so on. Write a bash program to achieve this.

You may assume `intense.py` will be in the same directory as your script. For your reference, the `wait` command in bash blocks until all of the existing background processes exit. (5)

7. Consider a scheduling algorithm that uses a weighted round-robin approach to assign CPU time to tasks. Let there be n tasks, each with a weight w_i and an execution time e_i, where $i = 1, 2, \cdots, n$. The algorithm assigns CPU time to tasks in a round-robin fashion, but the amount of time given to each task in a given round is proportional to its weight. That is, if task i is scheduled in a round, it will receive $\frac{w_i}{\sum_{k=1}^{n} w_k}$ of the available CPU time in that round.

   Prove that this scheduling algorithm is fair, in the sense that each task will receive a proportion of the CPU time that is at least as large as its weight. In other words, prove that for any task i, the proportion of the total CPU time that it receives is $\geq \frac{w_i}{\sum_{k=1}^{n} w_k}$. (10)