UCLA Computer Science 111 (Winter 2017) Midterm
100 minutes total
Open book, open notes, closed computer

Name: Vansh Gandhi Student ID: 204621797

1	2	3	4	5	6	7	8
1	5	2	11	11	9	13	10

72

1 (3 minutes). Does Ubuntu use soft or hard modularity? Briefly explain.

2 (5 minutes). Suppose you run the following command, where 'lab0' implements Project 0.

```
echo four | \
  lab0 --output=score --output=and \
       --output=7 --output=years --output=ago
```

What behavior should you observe and why?

3 (7 minutes). Suppose the x86-based kernel Xunil is like the Linux kernel but reverses the usual pattern for system calls: in Xunil, an application issues a system call by executing an RETI (RETurn from Interrupt) instruction rather than by executing an INT (INTerrupt) instruction. Other than this difference in instruction choice, Xunil is supposed to act like Linux.

Is the Xunil idea completely crazy, or is it a valid (albeit unusual) operating system interface? Briefly explain.

4a (9 minutes). Translate the following shell script to simpsh as well as possible. Your translation should simply invoke simpsh with appropriate arguments.

```
#! /bin/sh
(head -n 20 2>a <b | sort 2>>c | tail) >d
cat <d | cat >>d
```

4b (4 minutes). How and why will your translation differ in behavior from the original?

4c (5 minutes). Give a scenario whereby the above shell script, or its simpsh near-equivalent, will loop indefinitely.

4d (5 minutes). Propose minimal upward-compatible changes to simpsh that will allow you to translate the above script to simpsh faithfully, so that its behavior is 100% compatible with the standard shell.

4e (5 minutes). Give a scenario involving a single invocation of simpsh that can first crash simpsh and cause it to dump core, and then output the message "Fooled ya!" to standard output.

5. Round Two Robin (T2R) scheduling is a preemptive scheduling algorithm, like Round Robin (RR) scheduling, but it differs in that when a quantum expires and two or more processes are in the system, then T2R does not always move the currently-running process to the end of the run queue; instead, with probability 0.5, T2R lets the currently-running process continue to run for another quantum, so that other processes continue to wait in the queue.

5a (6 minutes). Compare RR to T2R scheduling with respect to utilization and average wait time; give an example.

5b (5 minutes). Is starvation possible with T2R scheduling? Briefly explain.

6. Suppose you compile and run the following C program in a terminal session that operates on a SEASnet GNU/Linux server:

```
 1  #include <signal.h>
 2  #include <unistd.h>
 3  #include <stdio.h>
 4  static unsigned char n;
 5  void handle_sig (int sig) {
 6    printf ("Got signal! n=%d\n", n++);
 7  }
 8  int main (void) {
 9    signal (SIGINT, handle_sig);
10    do {
11      printf ("looping n=%d\n", n++);
12      signal (SIGINT, handle_sig);
13    } while (n != 0);
14    return 0;
15  }
```

Give race-condition scenarios by which this program could possibly do the following:

6a (3 minutes). Output more than 256 lines.

6b (5 minutes). Output successive lines containing "n=N" and "n=N" strings where N is the same integer in both lines.

6c (3 minutes). Output a line containing two "=" signs.

6d (5 minutes). Dump core.

6e (5 minutes): Which lines or lines of the program can you remove without changing the program's set of possible behaviors? Briefly explain.

7. Consider the following implementation of read_sector:

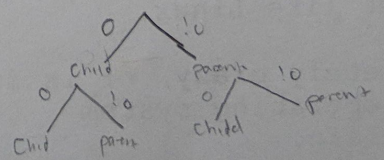
```
void wait_for_ready (void) {
  while ((inb (0x1f7) & 0xC0) != 0x40)
    continue;
}

void read_sector (int s, char *a) {
  /*1*/ wait_for_ready ();
  /*2*/ outb (0x1f2, 1);
  /*3*/ outb (0x1f3, s & 0xff);
  /*4*/ outb (0x1f4, (s>>8) & 0xff);
  /*5*/ outb (0x1f5, (s>>16) & 0xff);
  /*6*/ outb (0x1f6, (s>>24) & 0xff);
  /*7*/ outb (0x1f7, 0x20);
  /*8*/ wait_for_ready ();
  /*9*/ insl (0x1f0, a, 128);
}
```

What, if anything, would go wrong if we did the following? Briefly explain. Treat each proposed change independently of the other changes.

7a (3 minutes). Remove /*8*/.
7b (3 minutes). In /*3*/, change 0xff to 0xfff.
7c (3 minutes). Interchange /*3*/ and /*4*/.
7d (3 minutes). Interchange /*6*/ and /*7*/.
7e (3 minutes). Put a copy of /*1*/ after /*9*/.

8 (10 minutes). What does the following program do? Give a sequence of system calls that it and its subprocesses might execute.

```
#include <unistd.h>
int main (void) { return fork () < fork (); }
```



1) Ubuntu uses hard modularity. This is because there are some special instructions that can only be accomplished in Kernel mode. Therefore, you have to use the defined interface in order to execute any of those instructions. —> soft modularity

Therefore, Ubuntu is implementing the virtualization approach (via hardware) to hard modularity through use of traps.

2) You should see four in Score, but not in and, 7, years, or ago.

This is because labO reads from stdin, which in this case is the output of echo. Then it goes through all the arguments and runs dup from stdout to the file specified. This will work for the first file; however, at the end of duping the first file, we close the file descripter for stdout. Therefore, any further calls to dup with an old fd of 1 would fail. Files and, 7, years, and ago will still be created. —0

3) This implementation of xunil would not work. A RETI instruction would not be able to take in a number, and so would not be able to look anything up in the trap table. Furthermore, you wouldn't know whether you actually want to return from an interrupt or you want to start an interrupt. There would be no way of actually knowing. RETI is privileged, so you "know". —2 ← put it in a register.

4a) ./simpsh --creat --wronly a --rdonly b --pipe --append --creat --wronly d --pipe --pipe --pipe --append --wronly c
--command 1 6 0 head -n 20
--command 5 8 11 sort
--command 7 3 4 tail
--command 2 10 12 cat
--command 9 11 13 cat --wait
./simpsh --append --rdwr d
--command 0 0 0 cat --wait

5

4b) My behavior of the translation differs from the original in that it replaces writing and reading from d with a pipe. Then, it calls simpsh again, this time with the sole purpose of appending d's file contents to itself. Also, any unspecified stderrs are simply redirected to stdout.

2

4c) This would loop the script or simpsh indefinitely if file d was deleted between calls or if one of the write ends of a pipe was not closed in simpsh

0

4d) To make it work properly, if simpsh could handle more commands after the --wait flag, then you could wait for some file operations to finish before proceeding to the next command to run. This would make it unnecessary to call simpsh twice, and would also prevent deletion of intermediary files. To fix the stderr issue, there needs to be a way to print directly to console instead of file.

2

4e) If you printed "Fooled Ya!" which didn't fill up the buffer, then did a null dereference, then the program would dump core and flush all its buffers, resulting "Fooled ya!" being written to the screen after.

2

5a) The average wait time would go up. This is because the order of processes doesn't change, and processes don't run for a shorter amount of time, only a longer amount of time, on average 1.5 x longer. So, each process runs 1.5x as long on average, increasing the wait time.

RR: A|B|C|D|A|B|C|D|A|B|C|D

R2R: A|B|B|C|D|D|A|A|B|B|C|D|D|A|C|C

In this case, the wait time of D increased from 3+3δ to 4+3δ Processes would have higher utilization time since they have more opportunity to use the processor once they've started running

6

19/21

5b) Yes, starvation is possible. Since the scheduler decides whether to let the current running process run for another quantum based on probability, there is a chance, however small, that the result of this probability is such that it lets the current process keep running for an indefinite number of quantums.

√

6a) If the loop is on its last iteration, and then a SIGINT arrives, it would call the interrupt handler, increase n by 1, which would overflow back to 0, and the handler would print the 257th line.

3

6b) If the program receives a SIGINT after it has called printf in the loop but has not incremented n yet, then the signal handler would see the same value of n. This occurs because an interrupt can occur at any time.

√

5

6c) If the program was in the middle of printing out a line in the loop and receives a SIGINT before it has printed out the newline charater, then the signal handler will also print out an =, and it would be on the same line as the previous equals.

3

6d) If the SIGINT occurs because of an issue with printf. Printf is not async-signal-safe. Why?

3

6e) The signal call inside the loop is unnecessary. The signal hander was already set once outside the loop, and the specified function will remain the signal handler unless it is explicitly been overwritten by ignoring signal or using default signal behavior.

5

7a) You would try reading from the disk before it was ready to be polled for data. You would get bad or incomplete data. Behavior would be undefined.

2

7b) This would technically be an incorrect mask. But, since the registers take only the lowest 8 bits anyways, and we haven't shifted 3 at all, there would be no difference

3

7c) Nothing would go wrong, you would just write to the device/register in a different order.

3

7d) This would cause the disk to start reading before all relevant data was written regarding which data we want back from the disk. Would be ~~undefined behavior~~, or bad data.

2

7e) The disk might be ready immediately or it might add latency to the program. No data itself is being modified, so this shouldn't cause an issue.

3

8) This program creates 3 new processes. It calls fork 3 times.

1) Parent process calls fork(); (There are now 2 processes)
2) Parent process calls 2nd fork() in comparison statement. (Now 3 total processes. 1 from original parent & 2 children). In the child that was just created, it has reached the end of the program, so it finishes. Same with the parent process.
3) The child created in step 1 goes through step 2, except now this child is acting as the parent.

If at any point fork() fails, then the resulting child process would not be created and neither would any potential children