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

S3

Name: Aaron Chi Student ID: [redacted]

1	2	3	4	5	6	7	8
1	0	2	4	7	12	0	4

13

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 >>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 (); }
```

CS 111 midterm

1. Ubuntu uses hard modularity w/ a virtualizable processor. It has application & kernel classes. Applications can trap into kernel using syscall, where the OS can perform privileged operations. Application does not have direct access to privileged instructions

-2 soft modularity

2. getopt will return an error because you are not allowed to specify more than one output file. Also stdin will be set to Null because "echo four" cannot be piped as input to another command. Need to create a file a.txt with "four" then cat a.txt | labu ...

-5

3. This means that OS is in kernel mode initially, switches to user mode when INT instruction, then performs application operations. RTI switches back to kernel mode. It can work but performance should be slower than Linux because majority of operations are not privileged, so not necessary to let kernel mode be default.

2

ordinary op has same speed either way...

4. a) simsh --rdonly b --creat --trunc --wronly a
--creat --append --wronly --creat --trunc
--wronly --pipe --pipe --creat --trunc --wronly
--command 0 5 1 tread -n 20 --command 4 7 2 sort
--command 6 3 8 tail --wait
simpsh --append --rdwr --pipe --append --wronly
--command 0 2 3 cat --command 1 0 3 cat

b) simpsh behaves differently, we must create another file
e to store stderr for tail & cat commands. Need to
run simpsh twice to open file d for different
behaviors (> vs >>) ->
c) file b has less than 20 lines but no EOF

d) allow specifying stderr file stream default instead
of always redirecting stderr to a file. Allow multiple
separate sets of file open and commands to be run from
one instance of simpsh.

e) simpsh --catch --abort
(not sure of actual value of SIGSEGV, let 135 represent
SIGSEGV)

```
char *p = NULL // global variable
... signal (135, sig_handler);
// code for abort
*p = '5'; printf("Exoledya!");
// signal handler
static void sig_handler (int sig) {
    p = malloc (sizeof (char));
}
```

5. a)

Job	arrive	run
A	0	20
B	2	20
C	2	20

RR | A | A | A | B | C | A | B | C

T2R | A | A | C | B | A | C | B | A

2 utilization is the same for RR and T2R but wait ~~time is better,~~ because newly added jobs have a chance to be placed at front of queue, decreasing wait time.

5 b) Starvation is possible because if new jobs continuously arrived and get placed in front of queue, older jobs may never get chance to run again.

12/12 on 1

a) n++ performs n = 1+n, but n on right side is a dependent read. If n is updated between the time ~~you read~~ n and do assignment, then you can print the same line multiple times.

4 ~~b) similar scenario as in (a).~~ Say n = 5. In T1, you read value of 5 from n, then print 5 in T1, then read value of 5 from n in T2, then print 5 in T2, then assignment of n in T1 and T2 only 1 thread

c) One thread is in signal handler printf and ~~one thread in main printf~~, output overlaps and prints two = sign in a row. Same thread

2

d) SIGINT recered in the moment before n is assigned to new value in any thread X 0

e) Line 12, this signal call is already caused by signal in line 9. ✓

5

4th

7. a) Disk may not be ready when we perform inst, and get unpredictable result. 3

b) No errors, only the lowest order 8 bits are read. 3

c) No errors, same data still being written to same locations in memory. 3

d) Line 7 issues the write command to status & command register, data from line 6 is not written. 3

e) No errors, but unnecessary waiting because no disk operations occurred since last wait. 3

8. This program creates ~~two~~ child processes and compares their pid's. If 1st pid less than 2nd pid, return 1, else return 0. Fork is the only system call here.

SO 7

-6 4 problems