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

61

Name: Allan (Xiang) Chen   Student ID: 704624388

1	2	3	4	5	6	7	8
1	5	3	6	11	21	14	0

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 (); }
```

Allan (Xiong) Chen

CS111 Midterm Winter 2017

1. Ubuntu uses hard modularity for executing privileged instructions. User code cannot execute privileged instructions directly. Instead, they must make system calls, which execute the INT instruction and causes the CPU to trap into kernel code.

−2 / soft modularity

2. The observed behavior should be the word "four" written to the beginning of the file "ago", possibly overwriting the first five bytes of the file if the file was nonempty.

−0

3. Assuming the effect of the RETI and INT instructions are not modified from what was discussed in class, then this would NOT work. When we execute system calls, the OS is supposed to save the current state of the user program such as the instruction pointer onto the stack. However, RETI does the opposite: it pops whatever information was on the stack into these registers, likely overwriting these registers with garbage values and crashing the OS.

3

0 1 2 3

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

6

b). The simpsh version of the script has a race condition where two commands can be writing to d at the same time, so the file d's contents at the end will not be as same as the shell script, which does sequential writes to d.

0

c). If simpsh has a bug where it forgets to close both ends of each pipe before waiting, then the process will loop indefinitely waiting for children to finish when the children are in fact waiting for the parent to close the pipes.

0

d). Change simpsh so that if --wait is specified in the middle of the arguments, it will wait for all commands that came before the --wait command

0

4e). ./simpsh

0

5. a. Let A have run time 4 quantums, arrival time 0
Let B have run time 1 quantum, arrival time 1.

RR: ABAAA Average wait time RR: $\frac{0+0}{2} = 0$
T2R: AABAA Average wait time T2R: $\frac{0+1}{2} = \frac{1}{2}$

assume T2R lets A run for another quantum

6

T2R has higher average wait time than RR because certain jobs can now use more than one quantum per run, forcing other ~~proce~~ jobs to wait longer

Assuming that T2R takes slightly more CPU cycles than RR in order to calculate a random probability, then the utilization of RR is higher, since the RR scheduler consumes less CPU time and allows more "real" user work to be done on the CPU.

actually, randomization is cheaper than much context switching these days

b) Assuming that each job can only extend its quantum once, then starvation will not occur. Even if all jobs extend their quantum once every time they're run, we'll be effectively doubling the quantum for every job, so each job will eventually get a chance to run.

5

21/21 3

(and n=05)

6 a. If, after line 11 has been executed and SIGINT is received, then n will be incremented again, skipping past n=0.

b. When line 11 prints n=N to the screen but has not incremented n yet, SIGINT is received, causing the signal handler to print out the same n=N.

5

c. Just before printf() writes '\n' to the buffer, SIGINT is called and prints out "Got signal! n=N\n" on the same line

3

d. Since printf calls malloc, it is not reentrant, So if printf() on line 11 was in the middle of a malloc call and SIGINT is received, then malloc() will be called again the printf in the signal handler, before the first malloc finished, thereby corrupting heap data structures and causing the program to dump core.

5

e. line 12 can be removed, since it is redundant to register the same signal handler twice - nothing is changed by doing so

5

7 a. We would try to read data from the disk controlle before the disk has retrieved the sector's data, which means we would read garbage values, or the read may fail
incorrect

3

b. Nothing would go wrong. Since outb takes a char as its second argument, outb would truncate all but the least significant 8 bits of S anyways.

3

c. Nothing would go wrong. We would simply be writing the second least significant byte of the sector number into the proper location before writing the least significant byte into the proper location. The sector number is still correctly written to the controller

3

d. The command to read from sector would be issued before all the bytes of the sector number have been written to the disk controller. So the disk would return data from the wrong sector or simply return an error.

3

e. Nothing serious would go wrong (except we may need to wait a little longer if the disk (becomes busy again)).

it will not

2

8 fork() fork() exit(1)

−10