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

Name: Zhouheng Sun Student ID: 504-625-973

1	2	3	4	5	6	7	8
1	5	5	16	10	13	0	10

60

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. p0: test & basics

--output=score

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

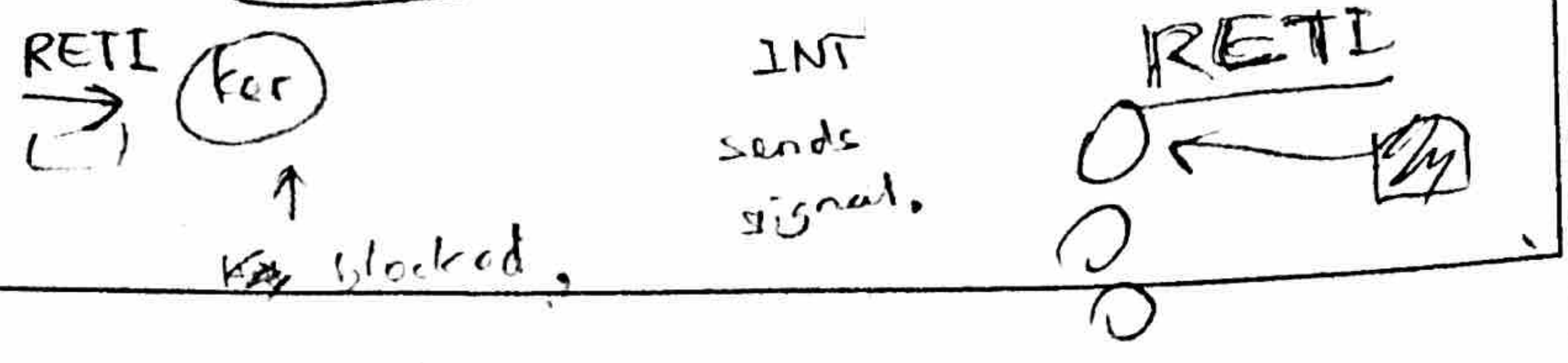
--output

What behavior should you observe and why? dup

while

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. Uhuru uses hard modularity via virtualization.
It seperates user mode from kernel mode, and the system will trap all the priviledged operations, and let the kernel inspect whether the user is doing illegal things to the system.
In this way, modularity in between processes is enforced and hard to break, rather than relying or trust ~~soft modularity~~ -2

2. "four" is written to the file "ago", in the local directory.
Because stdout redirection is performed for every --output option, ~~and~~ from left to right.
So the last file becomes the final stdout, and ~~it reads in what's piped~~ since stdin is default, stdout reads from what's typed or piped to it in shell, which is "four" in this case, and write to stdout, which is actually the file "ago".

3. It works despite great drawbacks in performance.
INT works by sending a signal to the kernel, asking it to trap and put the calling program in pause until the kernel is done, and handles control back to the user by calling RETI.

5 But: RETI is privileged and will trap.

RETI works on the opposite way: the kernel schedules a program to run, then TRAP into it, and wait until that program issues a syscall using RETI. By that point, the user program will set the appropriate registers so that syscall parameters get copied to the kernel's space during RETI, and the kernel will either do the job or abort it. Then, the kernel gets to schedule the processes by TRAPing into possibly ANOTHER program, and wait for that program to RETI, and so on.

It works in the bare minimum. It cannot, for example, do preemptive scheduling or use off-the-band communication to control malignant processes with signals and interrupts. The design is PASSIVE and UNSAFE, but it is valid as an interface.

--creat

```
4a. ./simpsh --rdonly b --creat --wronly a --append --wronly c \
    --creat wronly d --pipe --pipe --wronly /dev/null \
    --command 0 5 1 head -n 20 \
    --command 4 7 ~~3~~ sort \
    --command 6 3 8 tail
    --wait
```

3

5

```
./simpsh --append --rdwr d --pipe --wronly /dev/null
         --command ~~0~~ 2 3 cat
         --command 1 0 3 cat
```

4b. My shell Treats (...) >d as if the parentheses are removed. But in bash, a new shell is spawn for the commands in the parenthesis, and it's the ~~[illegible]~~ new shell's output that gets redirected to d. Therefore, results could differ when there's an error during (...), where bash will output error message from the new shell, along with tail's output, but mine will only print tail's output to d.

2

4c. If d is so large, that cat cannot read all of it ~~[illegible]~~ fast enough, then the second cat will ~~[illegible]~~ append a copy of d after d before the first cat reaches the EOF of the old d.
We then will have a problem, because now the first d will have to read again a portion of the same length or longer, so it will never reach EOF before file contents get updated by the second cat. If d is a file, this means generating an ever-increasing file. Or if d is a pipe, this will cause d's write buffer to be filled, ~~with cat~~ because the first cat has too much to do, and the pipes will hang forever.

5

4d. Add an option to let simpsh start another simpsh shell. Call it --simpsh, which takes three filenos only. ~~argument~~ Put the subshell's code in a file and tell --simpsh to read from it, and get its output using the other two filenos (stdout, stderr), which in our case should be piped to d.

2

4e.

simpsh ~~command 'bash -c "raise SIGSEGV' cat~~

--rdonly /dev/stdin --catch SIGSEGV
--wronly /dev/stdout --abort
--wronly /dev/stderr --command echo 'fooled ya!'
0 1 2

2

5. a.

Job	arrival	run time	waiting ~~time~~	~~turnaround~~ time
A	0	3		
B	1	2		
C	2	4		
D	3	1		

RR: ABCDABCACC. Wait time = 0 for all.
Util. = 10/10+8δ.

T2R (alternate bet' switch / continue)

ABBCCDAACC Wait time: A = 0
B = 0
C = 1
D = 1.

Util. = 10/10+5δ.

6

∴ T2R has ~~less~~ greater average wait time but greater utilization rate because of lower context switch #.

NG SUN

13/21

5b. Yes, when too many small jobs come in at the same time, blocking the larger jobs from running. But this is less of a problem than RR because larger jobs ~~[illegible]~~ can get a longer time to run before switching.

4

6a. One thread → handle_sig, another in while loop, ~~[illegible]~~ and ~~[illegible]~~ both increment n before it is written and updated. Now n will only increase by one after two lines are written.

✓ 3

6b. Same as 6a. ~~After~~

✓ 5

6c. Both printfs in handler and while loop write to stdout at the same time. Since pipes are byte-wise streamed, the sequence "n=" becomes interleaved as "nn==".

✓ 3

6d. After handle_sig is invoked once, and before the handler is remounted in while loop, another signal arrives at the interval.

✗ 0

6e. the first line of main.

Signals

✗ 2

8. It creates three child processes.

ⓒ The order of eval, is undefined.
Let's suppose the left one runs first.

ponent : calls left fork()
child #1 : sees "0 < fork()", call right fork.
child #2 : sees "0<0", return false.
child #1 : sees "0<cpid" return true.
parent : sees "cpid1< fork()", call right fork
child #3 : sees "cpid1<0", return false
parent : sees "cpid1 < cpid3", return accordingly.