1	2	3	4	5	6	7	8
1	0	2	1	11	11	8	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 (); }
```

1) Ubuntu uses hard modularity because there is a key difference between privileged and unprivileged instructions. If a process/thread attempts to use these unprivileged instructions in the user mode, then the OS will block or end the process because the thread/process must be given access to the system mode to use these instructions. This is what hard modularity looks like, where as soft modularity, the OS would not have enforced the thread/process from using system calls in user mode.

-2 soft modularity

2) If you run the following command where lab0 implements project zero, the word "four" is used as an input for lab 0 and there are five outputs phrases. We will observe that the four will be ignored since we did not look for an input for lab 0, and the program will just print out "Score and seven years ago" each word on a new line.

-5

3) If Xunil issues a system call by returning from interrupt rather than executing an interrupt instruction, the OS will not work and will be called completely crazy. This is so because when an OS does Int, the PS and PC are stored so that the OS knows where to return the execution of the program after the system call is done. If we do RETI instead, the OS will try to pull some previously stored PS & PC value into the current PC/PS registers (that is the purpose of RETI, to restore PC/PS after a system call) and XUNIL will start the program at the wrong place

after finishing the system call (and breach the program).
Ex. close(1); ← PC was 2, PS was 2000
{... code}
read(...); ← PC was 15, PS was 2500, but the RETI restore PC=2 & PS=2000 from previous stack due to forth -> or it will restore garbage values

4 a) (head -n 20 2>a <b | sort 2>>c | tail) >d
cat <d || cat >>d

1 fd in total

./simpsh --append --wronly a --wronly b --pipe --append --wronly c --pipe --append --rdwr d --pipe --command 1 3 0 head -n 20 --command 2 6 4 sort --command 5 7 9 cat --wait --command 7 9 8 cat --command 8 7 6 cat --wait

* Note, I have used file a (fd 0) as the stderror file several times. Also the --wait at the end takes care of closing file descriptors for pipes so no ∞ loops

b) My translation will differ from the original in that for the >d at the end of line 1, and all of line 2, any errors will go to stderror, while in my simpsh, any errors will be appended to file a. Furthermore, in terms of behavior, my program has an extra "cat" command to put the output of (head -n, ...) into d. This will mean my program is slightly slower than bash. Lastly, rather than closing file descriptors after each command for pipes (some ∞ loops), I call --wait at the end which follows the TAs implementation to close all fds so no infinite loops ☺. post on piazza!!

2

c) A scenario where the above script or simpsh will loop indefinitely is if the pipes never receive an end of file, then they will continue to try to fread or

write and will wait forever. This can happen if we don't have --close in simpsh (I went about this with --wait), or if d is a file that keeps on growing with no eof to tell the pipe it is done.

4

d) A minimal change to simpsh that would allow it to translate faithfully to the bash script would be to allow multiple inputs and let it close files descriptor (pipes properly.

0

e) A scenario where a single invocation of simpsh would crash it and cause it to dump core and then output then message "Fooled ya!" to standard output would be if you were to catch a signal like segfault that would cause it to dump core, but in your signal handler, you output to Standard Output "Fooled ya!". Another way is if you do --command echo "Fooled ya!" and then cause it to segfault and dumpcore, the child process that was forked may be delayed by a bit, and would output "Fooled ya!" to standard output even after simpsh has crashed & dumped core.

5

5) a) With T2R, where a process running has 0.5 (50%) chance to run again, or to go do the back), the utilization and average wait times for RR vs T2R differ such that the utilization of CPU would be slightly higher for T2R since it has less overhead than their RR for switching contexts since half the time, it will be running the

current process. In terms of average wait time, RR will be better since processes know after a fixed # of quantums, they will get a chance to run, while not so true for T2R.

σ_1 = delta for diff process,
σ_2 = delta for same process

Ex. a → arrive 0 → total = 1
b → arrive 0 → total = 2
c → arrive 0 → total = 3
Quantum = 1

6 RR → $A\sigma_1, B\sigma_1, C\sigma_1, B\sigma_1, C\sigma_1, C$ → $6q + 4\sigma_1 + \sigma_2$ avg wait = $\frac{0+1+2}{3} = 1$

T2R → $A\sigma_1, B\sigma_2, B\sigma_1, C\sigma_2, C\sigma_1, C$ → $6q + 2\sigma_1 + 3\sigma_2$ $\frac{0+1+3}{3} = 4/3$

$\sigma_2 < \sigma_1$, so T2R gets more CPU time and thus more utilization, But avg wait time went from 1 to 4/3.

5) With T2R, starvation would not be possible, because even if there are multiple long jobs in the queue, the probability that the long job would run 3 times in a row is 1/8, 4 times in a row is 1/16, so on, so forth. Eventually, the jobs after will be able to run, and won't be blocked out forever. (They may be blocked for several more quantums than before, so avg wait time is more).

6) a) A race condition where the program can output more than 256 lines would be if one thread read a value like 2 for n, but before it could increment and store it again, another thread also reads 2, outputs a line, and stores 3. The first thread then outputs a line, increments n to 3 as well, and stores. Now, we have 2 lines printed instead of 1 for n=2, so if this happens multiple times, we would get more than 256 lines in total. only 1 thread here

2

1 thread

b) If one thread picks n=5 and increments it to 6, outputs the line, but before it can store 6 for the other thread to pick it up, the other thread interrupts, picks up the value of 5, increments it to 6, outputs n=6 as well and then stores it. Thus, 2 lines with same n values.

4

c) There can be a line outputting two == signs if the char buffer for the output of one thread was interrupted after the =, like "Got a signal! n=" and the other thread appends its content to this string buffer, so the final output looks like "Got a signal! n= Looping n= 3\n=2\n"

3

2 equal signs; rest of first thread

d) A race condition that may dump core is if two threads attempt to both return at the same time. We cannot call _Exit() after the return 0, meaning the program would crash and dump core.

(Exit(2)) can/should only be called once.

X 0

Cannot

e) You can remove line 9 technically if you just want it to loop 256 times and print out the value of n each time. However, this would change the first output from "Got a signal" to "Looping" and so on so forth. If you wanted that specific order, then I would not remove any lines because then you would not get order.

"Caught... Looping... Caught..." etc etc.

2

a) If we remove line 8, we would not be waiting for the disk to be ready before continuing on to copy disk cache to our program. This means we might not yet a valid output from mem location 'a' of the disk.

b) If we change line 3 to 0xfff, then we would be sending too many bits into register 0x1f3 which would be masked by the next line (s>>8) && 0xff. This would lead to an incorrect read from the disk and overlap of data.

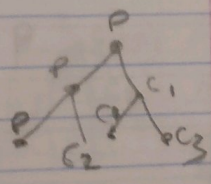
c) If you change 3 & 4, you would not get any difference since s>>8 or s>>16 will not change s itself (would need s>>=8 would actually change it).

where is d>? 0

d) If you put a copy of line 1 after line 9, you would merely delay your read i.e. sector so that it only exits when the disk is ready again to be read from (& so only performance decrease).

it is ready

e) int main (void) {return fork() < fork();}
↳ fork returns 0 for child, and positive for parent
-1 0

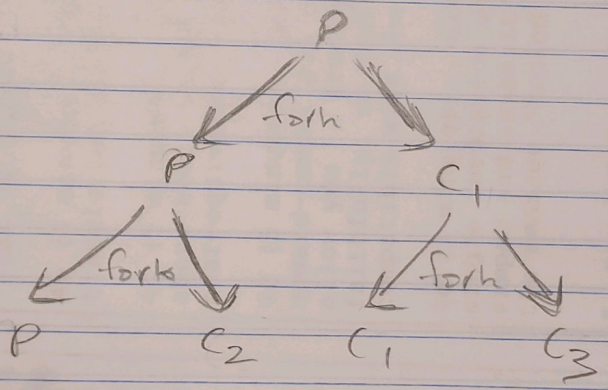


This is in essence a type of fork bomb. The parent process will return a value of greater than 0 and the child is 0, but the parent will call fork() and then call fork again, creating two children. The parent will then do id < id, which is false, so it will return false but for the child, it has done the left hand fork, and must call fork() again to evaluate the right hand side. This looks a bit like the tree diagram

and could continue forever (hence a fork bomb).

↳ Sequence of system calls:
fork fork return Parent (P in diagram) fork()
return Child 1 (C1 in diagram) fork return (2 ...
so on, so forth

>> lots of forks!



etc because of while loop