UCLA Computer Science 111 (Fall 2010)
Midterm
100 minutes total, open book, open notes

Name: [redacted] Student ID: [redacted]

1	2	3	4	5	6	7	8	total
4	6	10	6	21	5	8	5	65

1a (5 minutes). In the low level bootstrap code given in class, suppose we omit the line "outb(0x1f7, 0x20);". What will go wrong, and why?

1b (5 minutes). In the low level bootstrap code that prints to a screen, suppose we uniformly replace "uint16_t" with "char". What will go wrong, and why?

2a (5 minutes). Give an example of how the waterbed effect applies to bootstrapping.

2b (5 minutes). Give an example of an economy of scale in scheduling.

3 (10 minutes). As shown in class, double buffering can help performance in some cases. Would it help even more to triple buffer? quadruple buffer? Why or why not?

4 (8 minutes). Give realistic C code showing why critical sections are sometimes needed even in a single-threaded environment, where there is only one process in the system and that process has just one thread.

5. Consider the following code:

```
 1  int sortIO(void) {
 2    pid_t p = fork();
 3    switch (p) {
 4      case -1:
 5        return -1;
 6      case 0:
 7        execvp("/usr/bin/sort",
 8               (char *[]) {"sort", NULL});
 9        _exit(1);
10        break;
11      default:
12      {
13          int status;
14          if (waitpid(p, &status, 0) < 0)
15            return -1;
16          if (!WIFEXITED(status)
17              || WEXITSTATUS(status) != 0)
18            return -1;
19          return 0;
20      }
21    }
22  }
```

For each of the following proposed changes, explain the consequences of the change. Consider each change independently.

5a (3 minutes): Swap lines 6 and 11.

5b (3 minutes): Omit lines 16 through 18.

5c (3 minutes): Omit line 5.

5d (4 minutes): Swap lines 4 and 6.

5e (4 minutes): Change "_exit" to "exit".

5f (5 minutes): Insert "fork();" between lines 2 and 3.

5g (3 minutes): Change line 7's "sort" to "cat".

Suppose I use an unusual x86 C compiler in which every function call is a system call. To call a function FOO, my compiler generates all the code that an x86 compiler normally generates, but instead of issuing a CALL FOO instruction, it pushes FOO onto the stack, and then executes an INT 92 instruction. By convention, the 92nd interrupt vector pops FOO from the stack and then jumps to FOO.

6a (7 minutes). Will this idea work at all on Linux? If so, explain; if not, explain what would need to change with Linux to get it to work.

6b (8 minutes). Assuming the idea works, will this sort of thing help us to attain hard modularity instead of soft modularity? Explain.

7 (12 minutes). Suppose we use a round-robin (RR) scheduling policy, except that whenever a quantum expires, we choose a process at random. Call this approach Random Round Robin ("RRR"). Compare RRR to ordinary RR. Which one is fairer? Which one has lower average turnaround time? Which one has lower average response time? Explain, using an example set of jobs to schedule.

8 (10 minutes). A precondition for lock() is that a process cannot attempt to gain a lock that it already has. Suppose a process violates this precondition: give a specific example of what can go wrong, using the implementation of 'lock that was discussed in class. Similarly, give an example of what can go wrong with unlock() if its similar precondition is violated.

Aniket Dey
803 532 544

1) a) The disk would never ~~read~~ start reading. +4
The program has to set the 0x1F7 ~~byte~~ to cell to 0x20 to instruct the ~~prog~~ disk to read. The ~~progr~~ program will then wait indefinietly for the disk to finish the read it never started. Will read in ~~gab~~ garbage values into the buffer.

b) If we replace uint16_t with char, we are +0 replacing a 16 bit data structure with an 8 bit one. ~~If there are buffers that have 16 bit size,~~ Any reads we do would contain half the data then before. In the end this would slow the program greatly as ~~more programs~~ ~~bytes have to be read off the disk~~ more disk reads are required

1 byte ⇒ color
1 byte ⇒ data

2) a) Our word count program was simple and secure, but we gave up useability for anything other then word count.

4

b) When there are thousands of threads wanting to use one core, the scheduler becomes a bottleneck for the threads. Why? e.g., RR?

2

3) triple buffering and quadruple buffering would not be useful due to dependent cases? The disk can't read section 2 until it finished reading section 1. OK. The check sum, which benefitted from double buffering, can't execute until the read it is computing for is completed. The third buffer (or higher) would sit idle ~~for~~ as neither an extra read or check sum would be able to go there.

+10

xchg %eax, (%ebx)

4) fd = open ("/path_name", O_RDWR | O_CREAT, 0666);
foo (fd); //does some reads with fd
write (buffer, fd);
close (fd);

6

If an interrupt comes from the kernel while foo is running, and for some reason the kernel unlinks fd, we'd be writing to a non existent file when we return to the code. Making this a critical section would eliminate this problem.

5) a) The child would wait on process id 0, which would result in an error and it would return ~~status~~ -1. The parent would die either running sort, or calling _exit(1). either way, the program would die.

-1

b) The program would not care if the ~~process it~~ child exited, or exited with status 0. The child could've crashed, and the parent would act ~~like~~ like nothing happened.

c) If the fork fails, the main program would run ~~exit~~ the execvp code and die. (Or, run _exit(1), either way it dies).

d) The ~~program~~ fork fails, the main program runs the execvp code, and dies. If the fork works, the child process would continue running and the parent would wait forever for the child to complete. (2 instances of same program)

e) exit can be interrupted, while _exit cant. ~~pot~~
-2 Potential race condition while exit runs.

f) bothe the child and the parent would fork again.
-1 Both childs would run execvp and die and bothe parents would wait on the firstchild to finish.

g) The child will run 'cat' with the 'NULL' argument.
The child dies, and cat outputs nothing to stdout.

6) a) This code would work in Linux, because there are no rules that prevent x86 instructions to be ran from off the stack
2 Security features might need to be disabled though, as allowing stack code might warrant the risk of buffer overflows.
FOO = an address

b) This will help with hard modularity. Common code
3 previously shared by processes, can now be saved on their individual ~~stack~~ stacks, and run independently.

7) context switch time k Quantum 2
3 process
A time 3
B time 5
C time 4

RR: AA : BB : CC : A : BB : CC : B

respone: ~~1~~+k

0
1+k
2+2k

$\frac{3+3k}{3} =$

turnaround: 7+3k
12+6k
11+5k

$\frac{28+14k}{3} =$

9.33+4.66k

RRR: responce A: 0
B: $(\frac{1}{3})(1+k)+(\frac{1}{2})(1+k)$

8 4

2 Round Robin is fairer because a job is less likely to starve
Round Robin will have better response time because RRR might start a job that came later at the end of a quantum
1 (?) RR and RRR will have equal turnaround time because of context switches.

8) If a process gains a lock it already has, it will have two copies of the same lock. ~~If~~ If a file read has a lock on a file, and it ~~gets~~ a second lock, when the process unlocks the file, it may only unlock it once. The second lock would still exist and no process will be able to write to the file, thereby deadlocking it.

ϕ

Double unlocking can induce race conditions. If the read program unlocks its ~~read~~ read lock on a file and then unlocks it again, it may be unlocking the ~~lock~~ read lock ~~of~~ in another process. Now any process ~~can~~ can write ~~the~~ to the previously locked file ~~write~~ while its still being read from. Therefore we have a race condition.

+5