UCLA Computer Science 111 (Winter 2016)
Midterm
100 minutes total, open book, open notes,
no automated devices

Name:_____ Student ID:_____

1	2	3	4	5	6	total
2	17	4	4	23	8	58

Consider the following program, slightly modified from the bootloader discussed in class.

```
// MBR code
enum { pa = 0x20000, pn = 80 };
for (int i = 0; i < pn; i++)
  read_ide_sector(i + 100, pa + i * 512);
goto *pa;

enum { ide = 0x1f0 };
void wait_for_ready(void) {
  while ((inb(ide + 7) & 0xc0) != 0x40)
    continue;
}

void read_ide_sector(int s, int a) {
  wait_for_ready();
  outb(ide + 2, 1);
  for (int i = 0; i < 4; i++)
    outb(ide + 3 + i, (s >> (8 * i)) & 0xff);
  outb(ide + 7, 0x20);
  wait_for_ready();
  insl(ide + 0, a, 128);
}

// WC program

int cws(char *buf, int bufsize, bool *inword) {
  int w = 0;
  for (int i = 0; i < bufsize; i++) {
    bool alpha = isalpha((unsigned char)buf[i]);
    w += alpha & !*inword;
    *inword = alpha;
  }
}
```

c0
11000000

40
01000000

12

00100000

```
enum { oo = 200 };
void display_cws(int nwords) {
  char *screen = (char *) 0xB8000 + oo;
  do {
    screen[0] = (nwords % 10) + '0';
    screen[1] = 7;
    screen -= 2;
  } while (nwords /= 10 != 0);
}

void main(void) {
  long long int nwords = 0;
  int s = 50000;
  bool inword = false;
  int len;
  do {
    char buf[513];
    buf[512] = 0;  //end of the buffer
    read_ide_sector(s++, (int)buf);
    len = strlen(buf);
    nwords += cws(buf, len, &inword);
  } while (len == 512);
  display_cws(nwords);
}
```

1a (4 minutes). The WC program calls a function defined in the MBR code, but these are separate programs. Is this a typo, possible but a bad idea, or a good idea? Briefly explain.

1b (4 minutes). The WC program calls two functions that are not defined anywhere. Explain how this should be implemented, consistently with your answer to (a).

2a (8 minutes). This version of the code declares and uses four named constants pa, pn, ide, and oo, which were not in the original code. For each of these four constants, explain what it does, whether and why you might want to change it, and what you might change it to.

2b (20 minutes). Do the same thing with the remaining mystery constants in this the code. That is, give names to as few constants as possible, and replace each mystery constant with an expression that does not involve mystery constants. For each named constant, explain what it does, whether and why you might want to change

nd what you might change it to. (Please use er names than "pa", "pn", and "ide" -- those es were deliberately chosen to make the evious subquestion obscure!)

3. Suppose we spent a million dollars to write the code in problem 1 (i.e., suppose our application developers were reeeaally inefficient), and suppose we want to run the same program in a less-paranoid environment; something like SEASsnet GNU/Linux server, say. We could rewrite the application from scratch, but that would cost us another million dollars. So instead, we'd like to modify GNU/Linux to run this expensive program unmodified. One more thing: suppose our Linux kernel hackers are much more productive and are paid by a different company, so we don't need to worry about how much work they do.

3a (8 minutes). Describe in general a simple way to modify the GNU/Linux kernel on an x86-64 machine so that the above program will run unmodified. Assume that the program is compiled in the same way as before, that it is located on the same spot on the same drive, and that the data are in the same place as before. Do not worry about security on the GNU/Linux side, in order to keep things as simple as possible. For simplicity's sake, do not run the program in a virtual machine: let it use the native hardware as before.

3b (10 minutes). Now, suppose we start worrying about security on the GNU/Linux side. We still want to run the program unmodified, and let it access the main disk and display, but we don't want it to access any other device. We are willing to give up some simplicity for this extra security, but we still want to keep things simple and native when possible, without using virtual machines. Describe how you'd go about this.

4 (10 minutes). Suppose the program in problem 3 takes a long time because the disk is big and slow. Explain how the GNU/Linux scheduler could arrange for the single CPU to be shared between the program and other programs that are also running, doing other things. If this might require changes to the scheduler, explain what those changes might be. How does your answer vary depend on whether you assume the solutions in (3a) and (3b)?

5a (3 minutes). Now, suppose the user also develops some paranoia about the reliability of the hardware, and decides to run the program three times and make sure the count is the same each time. How would you modify the program in a minimal way so that there would be three versions of the program, one for each region of the display it would update?

5b (8 minutes). If the program takes 100 seconds, the procedure described in (a) will take ~300 seconds. But the paranoid user doesn't want to wait that long, and instead wants to run the three instances of the program in parallel. And the user is willing to pay your expensive application programmers to modify the program to do that, though these modifications should be minimized to keep the costs down. Explain the race conditions that would cause correctness problems if they just ran naively in parallel, and what the consequences of these race conditions would be.

5c (8 minutes). Use the Goldilocks principle to identify the critical sections of your three programs, to prevent these race conditions.

5d (8 minutes). Assuming the critical sections are enforced somehow, give a scenario whereby the combined application (all three programs) finishes in about 101 seconds. And give a scenario whereby the combined application finishes in about 500 seconds. Which scenario is more likely and why?

6 (8 minutes). Suppose you want a scheduler that is really unfair, while still keeping utilization as high as possible. Can preemption help you achieve your goal of maximizing unfairness? Or can you maximize unfairness well enough without having to resort to preemption? Briefly explain.

3

1a) This is possible, but it probably is a bad idea because the MBR is not intended to store regular programs. There are only 446 bytes available so code stored on it is not very extensible. WC would need to define the MBR memory address and load its program into memory.

1b) WC should include a header file with macros to define the memory locations of these functions, which may be implemented natively in the ~~hardware~~.

3

2a) pa - start location of the file on disk. It may be useful to change if we want to read another file at some other memory address.

pn - total # of disk sectors to read. It would need to change to accomodate large files, or shrink for small files.

ide - location of disk controller status register. Generally should not change, unless a different status byte needs ~~to be set, maybe a second disk.~~

✓

ou - location to start printing on screen (+200 would be somewhere in the middle). Change this if you want to move the word count to another part of the display, say 400 to move further down.

the program and other programs that are also

2b) 0xc0 → disk status mask
Only check the 1st 2 bits for disk access status by masking it w/ this constant. Change if you only need to check a single status bit, for example, 0x80.

0x40 → disk status is ready (bit pattern 01000000)
01 as the 1st 2 bits means the disk is ready to use. Change if you want to check for another status, like busy writing/reading.

512 → size of disk sector
Change if your disk uses another sector size, like 1024.

0xff → sector number mask
bitwise AND this to change only the sector number in the disk controller register. Change this if you want to accommodate larger numbers and the disk supports it.

0x20 → busy status flag
Pattern of busy flag for disk. Change to set another disk status.

128 → size of read
Change to read more or fewer bytes at a time from disk.

0xB8000 → display output address
Memory buffer location for the graphics controller. Change if you want to display at another point onscreen, like 0xB8080.

513 / 512 → buffer size
Change to make the buffer larger or smaller, to balance memory usage with read speed. for example, 257, 256 or 1025, 1024.

3b) Create a compatibility layer within the kernel that re-defines all the external functions that the program calls. Make sure the functions return identical results, but are coded at a higher level so that system calls are wrapped in additional safety checks. Then add a check in the kernel so that when WC runs, its requests are redirected to this compatibility layer, which will simulate direct access to the hardware.

2 ?

1

3a) Check when the program is run within the kernel. Then save the kernel state and suspend it, and switch execution to the WC program. Signal the CPU to run in full kernel mode, so WC can access any memory location without triggering segfaults. Once WC finishes, Linux can resume execution.

2 ?

4) The scheduler will need to be modified to handle low-level processes that aren't fully controlled by the kernel. Assuming solution 3b was used, the compatibility layer can signal an interrupt while WC is executing and switch control back to Linux. When another program is executing, the kernel can first trigger a context switch and then hand over control to WC + the layer. If 3a is used instead, then preemptive multitasking can't be used, and WC must cooperate within the function calls.

4

(ler)

When WC calls a function, then the function can cede control to the kernel and normal program scheduling can resume.

5a) Define the display address as a preprocessor macro, so it can be quickly changed and recompiled for each version.

3

5b) Two or even all three could try to read from disk simultaneously. The disk controller would be unaware of the parallelism, so it would serve requests arbitrarily and one instance could get another's data, or the read request would simply fail. The disk's status register could also get messed up with garbled bits, from multiple instances all trying to set them partially overwriting another instance's request. The resulting word count outputs would be highly variable and incorrect since they processed garbage data.

8

5c) Each line of code calling read_ide_sector() should be a critical section, to ensure only one request is sent at a time to the disk controller. This is a good balance between locking each insl and outb, which still causes race conditions, and locking the entire program, which makes parallelism useless.

8

5d) Each program is processing the data while only one is reading from disk, so nothing is waiting for another.

4

500 sec - Every instance is trying to read at the same

time, so they have to "wait in line" for their turn.

The 500 sec situation is more likely because the read requests are likely to overlap at some points, so waiting occurs more often.

8

f) Yes, preemption is not necessary for unfairness. An SJF algorithm is enough to keep long jobs waiting for short ones to finish w/o preempting. They are opposite goals.

it does, whether and why you might want to change