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

$\frac{80}{1280}$

Name: Anderson Huang Student ID: 104299133

1	2	3	4	5	6	total
7	10	7	4	13	8	49

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 ~~< bu~~fsize; i++) {
    bool alpha = isalpha((unsigned char)buf[i]);
    w += alpha & ~~!*~~inword; 1b
    *inword = alpha;
  }
}
```

$2 \cdot 10^4$
$2(2^{?})$
$2(2^{16})$
$2^{2^{17}}$
$2^{1?}$
$2^{??}$

```
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); 1b
    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

This

-c, and what you might change it to. (Please use better names than "pa", "pn", and "ide" -- those names were deliberately chosen to make the previous 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.