Full Name:_____

Roll Number:_____

IT215: Systems Software, Winter 2016-17
Second In-Sem Exam (Time allowed: 1.5 hrs)
March 23, 2017

Instructions:

- Make sure your exam is not missing any sheets, then write your name and roll number on the top of this page.
- Clearly write your answer in the space indicated. None of the questions need long answers.
- For rough work, do not use any additional sheets. Rough work will not be graded.
- Assume IA32 machine running Linux.
- The exam has a maximum score of 30 points. It is CLOSED BOOK. Notes are NOT allowed.
- Anyone who copies or allows someone to copy will receive F grade.

Problem 1 (16):
Problem 2 (/2):
Problem 3 (/2):
Problem 4 (/2):
Problem 5 (/4):
Problem 6 (/4):
TOTAL (/30):

Problem 1. (16, 2 each points):

Circle the *single best* answer to each of the following questions. You will get -1 points for a wrong answer. If you circle more than one answer, you will lose the mark for the corresponding question.

Assume that none of the system or library calls fail.

1. Which of the following is NOT a possible output of the following program?

```
int x = 0;
void *A(void *arg) {
   int i;
   for (i = 0; i < 2; i++)
      x++;
}

int main() {
   pthread_t a, b;
   pthread_create(&a, NULL, A, NULL);
   pthread_create(&b, NULL, A, NULL);
   pthread_join(a, NULL);
   pthread_join(b, NULL);
   printf("%d\n", x);
   return 0;
}
```

   (a) 4

   (b) 3

   (c) 2

   (d) 1

2. Assuming there is a file named `foo` in the current directory, what is displayed on the screen after you execute the following command line at the bash shell prompt:

```
echo hi > foo ; wc foo > foo ; cat foo
```

   (a) `0 0 0 foo`

   (b) `1 1 1 foo`

   (c) `1 1 2 foo`

   (d) `2 2 4 foo`

3. In an empty directory, what is displayed on the screen after you execute this command line:

```
ls nosuchfile 2>out
```

   (a) nosuchfile not found

   (b) no output

   (c) nosuchfile 2 not found

   (d) 2 not found

4. Suppose there are two threads, `A` and `B`, which share binary semaphores `r` and `s`. At the starting time, `r=0`, `s=1`. The two threads execute the steps shown below.

```
/* Thread A */                    /* Thread B */
P(r);                             Statement B.1;
P(s);                             V(r);
Statement A.1;                    Statement B.2;
V(s);                             P(s);
Statement A.2;                    Statement B.3;
V(r);                             V(s);
```

   Which of the following statements is FALSE?

   (a) Statement `A.1` cannot start until statement `B.1` is completed.

   (b) Statements `A.1` and `B.2` can execute simultaneously.

   (c) Statement `B.3` cannot start until operation `A.1` is completed.

   (d) Statements `A.2` and `B.3` can execute simultaneously.

5. The `pthread_join` call is most similar to the system call:

   (a) `fork`

   (b) `waitpid`

   (c) `exit`

   (d) `execve`

6. What is the effect of the following x86 instruction:

```
mov 8(%ebp), %ecx
```

   (a) Add 8 to the contents of `ebp` and store the sum in `ecx`.

   (b) Add 8 to the contents of `ebp`, treat the sum as a memory address and store the contents at that address in `ecx`.

   (c) Add 8 to the contents of the memory location whose address is stored in `ebp` and store the sum in `ecx`.

   (d) Add the contents of `ebp` to the contents of memory address 8 and store the sum in `ecx`.

Consider the code below.

```
int balance = 0;

void *mythread(void *arg) {
    int result = 0;
    result = result + 200;
    balance = balance + 200;
    printf("Result is %d\n", result);
    printf("Balance is %d\n", balance);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, NULL);
    pthread_create(&p2, NULL, mythread, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
}
```

7. When value of `result` will be printed by thread `p1`?

   (a) Due to a race condition, different values may be printed on different runs of the program.

   (b) 0

   (c) 200

   (d) 400

8. What value of `balance` will be printed by thread `p1`?

   (a) Due to a race condition, different values may be printed on different runs of the program.

   (b) 0

   (c) 200

   (d) 400

Problem 2. (2 points):

What is displayed on the screen after you execute the following command at the bash shell prompt? Explain.

```
echo echo echo | bash
```

Problem 3. (2 points):

Assume values of type `int` are allocated 4 bytes. Given the following C code:

```
union {
   int wn;
   char cn[4];
} num;

num.wn = foo();
if (num.cn[3] & 0x80)
    printf("Sign bit on\n");
```

A. If `foo()` returns the value -129 (0xFFFF FF7F) on a big-endian machine, will the "Sign bit on" message get printed? (Circle one)

YES NO

B. If `foo()` returns the value -129 (0xFFFF FF7F) on a little-endian machine, will the "Sign bit on" message get printed? (Circle one)

YES NO

Problem 4. (2 points):

Let A and B be threads and let s and t be binary semaphores. Initially, both s and t are 1. The two threads execute the steps shown below.

```
/* Thread A */

P(s);   _______
P(t);   _______

/* Access Data */

V(s);   _______
P(s);   _______

/* Access Data */

V(s);   _______
V(t);   _______
```

```
/* Thread B */

P(s);   _______
P(t);   _______

/* Access Data */

V(t);   _______
V(s);   _______
```

Show a feasible sequence of calls to the P or V operations that will result in a deadlock. Place an ascending sequence number (1, 2, 3, and so on) next to each operation in the order that it is **called**, even if it never returns. For example, if a P operation is called but blocks and never returns, you should assign it a sequence number.

Note that there may be several correct solutions to this problem.

Problem 5. (4 points):

Having just added code to block signals before forking a child process and unblock them after adding the process to the job queue, a student is surprised to discover a mysterious race condition in her shell implementation that causes it to occasionally segfault when running foreground processes. Below are relevant function declarations and bodies.

```
/* add a job to the job list */
int addjob(struct  job_t *jobs, pid_t pid, int state, char *cmdline);
/* find a job (by PID) on the job list; return NULL if not found */
struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
/* delete a job whose PID=pid from the job list */
int deletejob(struct job_t *jobs, pid_t pid);

void sigchld_handler(int sig) {
    pid_t pid;
    int status;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
       if (WIFEXITED(status))
          deletejob(jobs, pid);
}

int eval(char *cmdline) {
    /* var declarations & cmdline parsing not shown */
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, NULL);
    if ((pid = fork()) == 0) {
        sigprocmask(SIG_UNBLOCK, &mask, NULL);
        execvp(argv[0], argv);
    }
    addjob(jobs, pid, (bg == 1? BG : FG), cmdline);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    if (!bg)
        waitfg(pid);
}

void waitfg(pid_t pid) {
    struct job_t *j = getjobpid(jobs, pid);
    while (j->pid == pid && j->state == FG)
        sleep(1);
}
```

Explain how, precisely, the race condition plays out in order to cause the segfault, and explain how you would go about fixing it. Assume all system calls execute without error.

Problem 6. (4 points):

Throughout this question, remember that it might help to draw a picture. Consider the following C code:

```
void foo(int a, int b, int c, int d) {
    int buf[16];
    buf[0] = a;
    buf[1] = b;
    buf[2] = c;
    buf[3] = d;
}

void bar() {
    foo(0x15213, 0x18243, 0xdeadbeef, 0xcafebabe)
}
```

When compiled, it gives the following assembly:

```
00000000 <foo>:
   0: 55                        push   %ebp
   1: 89 e5                     mov    %esp,%ebp
   3: 83 ec 40                  sub    $0x40,%esp

   6: 8b 45 08                  mov    _____(%ebp),%eax
   9: 89 45 c0                  mov    %eax,-0x40(%ebp)

   c: 8b 45 0c                  mov    _____(%ebp),%eax
   f: 89 45 c4                  mov    %eax,-0x3c(%ebp)

  12: 8b 45 10                  mov    _____(%ebp),%eax
  15: 89 45 c8                  mov    %eax,-0x38(%ebp)

  18: 8b 45 14                  mov    _____(%ebp),%eax
  1b: 89 45 cc                  mov    %eax,-0x34(%ebp)
  1e: c9                        leave
  1f: c3                        ret

00000020 <bar>:
  20: 55                        push   %ebp
  21: 89 e5                     mov    %esp,%ebp
  23: 83 ec 10                  sub    $0x10,%esp
  26: c7 44 24 0c be ba fe ca   movl   $0xcafebabe,0xc(%esp)
  2e: c7 44 24 08 ef be ad de   movl   $0xdeadbeef,0x8(%esp)
  36: c7 44 24 04 43 82 01 00   movl   $0x18243,0x4(%esp)
  3e: c7 04 24 13 52 01 00      movl   $0x15213,(%esp)
  45: e8 fc ff ff ff            call   foo
  4a: c9                        leave
  4b: c3                        ret
```

Note that in `foo` (C version), each of the four arguments are accessed in turn. Recall that the current `%ebp` value points to where the pushed old base pointer resides, and immediately above that is the return address from the function call. Write into the gaps in the disassembly of `foo` the offsets from `%ebp` needed to access each of the four arguments `a`, `b`, `c`, and `d`. (Hint: Look at how they are arranged in `bar` before the call.)

(Blank page for rough work.)