Full Name: _____

CS 351 Spring 2015

Midterm Exam

March 25^{th}, 2015

Instructions:

- Write your full name on the front, and make sure that your exam is not missing any sheets.
- This exam is closed-book, closed-notes. Calculators are neither needed nor permitted.
- If you make a mess, clearly indicate your final answer.

1 (/20) :	
2 (/12) :	
3 (/8) :	
4 (/8) :	
5 (/8) :	
TOTAL (/56) :	

Problem 1. (20 points):

Circle the letter next to the *single best* answer for each question.

1. What accurately describes the C type declaration `int *arr[10]`?

   a. array of 10 pointers to integers

   b. pointer to an array of 10 integers

   c. array of 10 pointers to functions returning integers

   d. pointer to a function that takes an array and returns an integer

2. Given that `p` is declared to be of type `(struct foo *)`, which of the following is equivalent to the statement `p++;`?

   a. `p = *(p + 1);`

   b. `p = (struct foo *)((char *)p + 1);`

   c. `p = (struct foo *)((char *)p + sizeof(p));`

   d. `p = (struct foo *)((char *)p + sizeof(struct foo));`

3. Which of the following correctly **frees** the space allocated by the statement `void *p = malloc(N * sizeof(struct foo))`?

   a. `free(p);`

   b. `free(*p)`

   c. `free(p[N]);`

   d. `for (i=0; i<N; i++) free(p + i);`

4. Which of the following pieces of information is **not** maintained by the kernel on behalf of each running process?

   a. its process ID

   b. its group ID

   c. the set of blocked signals

   d. the previous function's return address

5. Which of the following is an example of a **synchronous** exception?

   a. a system call

   b. a function call

   c. a keyboard interrupt (e.g., `^C`)

   d. someone tripping over the power cord

6. Which of the following types of exceptions is usually triggered **intentionally**?

   a. a trap
   b. a fault
   c. an abort
   d. a floating point exception

7. Which of the following is **not** inherited by a child process across a `fork` (from its parent)?

   a. its group ID
   b. its signal handlers
   c. the set of pending signals
   d. the currently running program

8. Which of the following may **not** be retained by a process across a (successful) call to `exec`?

   a. its group ID
   b. its signal handlers
   c. the set of blocked signals
   d. the set of pending signals

9. Under which of the following conditions will the kernel take over the responsibility of reaping (i.e. *adopt*) a process?

   a. when it terminates due to a signal
   b. when its parent process has blocked the `SIGCHLD` signal
   c. when its parent process has already terminated
   d. when its children have all terminated

10. Which of the following actions is likely to be taken by the shell program in a child process after `fork`-ing, but before `exec`-ing?

    a. blocking the `SIGCHLD` signal
    b. registering a handler for the `SIGCONT` signal
    c. setting the group ID equal to the process ID
    d. adding the process ID to the array of job structures

Problem 2. (12 points):

Consider the following structure definitions:

```
typedef struct queue queue_t;
typedef struct node node_t;

struct node {
    node_t *next;
    void *val;
};

struct queue {
    node_t *head;
    node_t *tail;
};
```

For this problem you are to complete a handful of methods that implement a queue data structure. The following program demonstrates their use:

```
queue_t *q = make_queue();

enqueue(q, "supercali");
printf("%s", dequeue(q));

enqueue(q, "fragi");
enqueue(q, "listic");
enqueue(q, "expiali");
enqueue(q, "docious");

char *c = NULL;
while ((c = dequeue(q))) {
    printf("%s", c);
}

free(q);
```

When run, the program's output is as follows:

```
supercalifragilisticexpialidocious
```

Note that there are no memory leaks. Complete the implementations of `make_queue`, `enqueue` and `dequeue` on the following page.

```
/* allocates and returns an empty queue */
queue_t *make_queue () {

    queue_t *q = malloc(sizeof(________________));
    q->head = q->tail = NULL;
    return q;
}


/* add val at the tail of the queue q */
void enqueue (queue_t *q, void *val) {
    if (q->head == NULL) {

        _______________ = _______________ = malloc(sizeof(________________));

        _______________ = NULL;
    } else {

        _______________ = _______________ = malloc(sizeof(________________));

        _______________ = NULL;
    }
    q->tail->val = val;
}


/* removes and returns the item at the head of the queue q,
 * or NULL if q is empty. */
void *dequeue (queue_t *q) {
    if (q->head == NULL) {
        return NULL;
    }

    void *val = ________________;

    node_t *p = ________________;

    q->head = ________________;
    free(p);

    if (q->head == NULL) {
        q->tail = NULL;
    }

    return val;
}
```

Problem 3. (8 points):

For each of the following programs:

1. Sketch the corresponding process tree next to it
2. Circle all of the possible outputs beneath it

Assume that all `printf` statements are immediately flushed. Also, recall that when `wait` is called by a process with no children, it returns immediately (with `-1`).

```
int main() {
    int i;
    for (i=0; i<2; i++) {
        fork();
        printf("%d", i);
    }
    printf("2");
}
```

a. 0011221122 b. 0121201212 c. 0212021112 d. 0011112222 e. 0111012222

```
int main() {
    printf("1");
    fork();
    printf("2");
    wait(NULL);
    if (fork() == 0)
        printf("3");
    else
        printf("4");
    return 0;
}
```

a. 1234243 b. 1234342 c. 1224343 d. 1223344 e. 1243234

Problem 4. (8 points):

For each of the following programs, circle its possible output(s) next to it. Sketching a process tree is not necessary, though you may do so if it helps you reason about things. Assume that all `printf` statements are immediately flushed. Recall that `pause` blocks a process until after signal arrives.

```
int glob = 1;
int done = 0;

void handler(int sig) {
    glob = glob * 2;
    printf("3\n");
    done = 1;
}

int main() {
    signal(SIGCHLD, handler);
    if (fork() == 0) {
        exit(0);
    }
    int val = glob;
    printf("1\n");
    glob = val + 1;
    printf("2\n");
    while (!done) ; /* busy loop */
    printf("%d\n", glob);
}
```

a. `3124`

b. `3123`

c. `1322`

d. `1324`

e. `1232`

f. `1234`

```
void handler(int sig) {
    int stat;
    wait(&stat);
    if (WIFEXITED(stat))
        printf("%d", WEXITSTATUS(stat));
}

int main() {
    int i;
    pid_t pids[5];
    signal(SIGCHLD, handler);
    for (i=0; i<5; i++)
        if ((pids[i] = fork()) == 0) {
            printf("1");
            while (1) ; /* inf loop */
        }
    for (i=0; i<5; i++)
        kill(pids[i], SIGKILL);
    sleep(1);
    return 2;
}
```

a. `1212121212`

b. `1111122222`

c. `1`

d. `12`

e. `11111`

f. *no output*

Problem 5. (8 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 his 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;
    while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0)
        if (WIFEXITED(status))
            deletejob(jobs, pid);
}

void 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.