Full Name: ....

CS 351 Spring 2012

Midterm Exam

March 14th, 2012

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.
- Show your answers in the space provided for each problem. If you make a mess, clearly indicate your final answer.
- Good luck!

1	(/10) :	
2	(/12) :	
3	(/10) :	
4	(/10) :	
5	(/10) :	
6	(/8) :	
TOTAL	(/60) :	

Problem 1. (10 points):

Multiple choice. For each of the following multiple choice problems, choose the *single best* answer by circling its corresponding letter.

1. Which of the following operations would you say consistently incurs the *most* overhead?

   (a) function call and return
   (b) `exec`-ing a new program
   (c) a call to `printf`
   (d) performing a non-local jump

2. Which of the following is *false* with respect to (proper) memory management in a C program?

   (a) every call to `malloc` should be (eventually) paired with a call to `free`
   (b) the function that `malloc`'s a structure is always responsible for `free`-ing it
   (c) `malloc` and `free` are user level functions, and do not always require trapping to the OS
   (d) if an array of structures is allocated with a single call to `malloc`, only a single call to `free` is needed to deallocate it

3. Which of the following events would be properly categorized as an *asynchronous* exception?

   (a) division by 0
   (b) delivery of a signal
   (c) reading an end-of-file (EOF) marker
   (d) a segmentation fault (e.g., caused by dereferencing a NULL pointer)

4. Due to the simplistic mechanism used within the kernel to track pending signals for a given process, it is possible that . . .

   (a) a process terminates without becoming a zombie process
   (b) a parent process does not receive a SIGCHLD signal for every terminated child process
   (c) a child process cannot be reaped by the parent with the `wait` system call
   (d) a parent process is notified of a child's termination before it has actually occurred

5. In response to a valid `open` system call for a file that is already open, the kernel ...

   (a) allocates a new open file description object
   (b) allocates a new v-node object
   (c) allocates a buffer for the purposes of I/O redirection
   (d) copies an existing file allocation table entry to a new table index

Problem 2. (12 points):

The following C structure and type definitions are used in this problem to construct a linked list:

```
    struct node {
       int val;
       struct node *next;
    };
    typedef struct node llnode;
    typedef struct node *llnodep;
```

After reading the comment that precedes each of the following functions, circle the letter corresponding to the code fragment that completes the implementation correctly.

```
/* Allocates space for a new node and initializes it
 * to the given values
 */
llnodep make_node (int val, llnodep next) {

    llnodep p = _________________________;
    p->val  = val;
    p->next = next;
    return p;
}
```

A. `malloc(sizeof(llnode))`

B. `malloc(sizeof(llnodep))`

C. `malloc(sizeof(int) + sizeof(llnode))`

```
/* Frees all nodes (individually) in a linked list starting
 * with the passed in 'head'.
 */
void free_list (llnodep head) {
   llnodep p;
   while (head != NULL) {
      p = head;

      _________________________;
      free(p);
   }
}
```

A. `free(p->next)`

B. `head = p + 1`

C. `head = head->next`

```
/* Pushes a new node containing the value 'val' onto the
 * front of the list, modifying the caller's pointer so as
 * to reflect the new list head. This function could be used
 * as follows:
 *
 *    llnodep list = NULL;
 *    push(5, &list);
 */
void push (int val, llnodep *head) {

    _____________________________;
}
```

A. `*head = make_node(val, *head)`

B. `head = make_node(val, head)`

C. `head = make_node(val, &head)`

```
/* Navigates to the end of the list whose head is passed in,
 * and adds a new node with the given value there. Assumes that
 * the list is not empty.
 */
llnodep add_to_end (int val, llnodep head) {
    llnodep p;

    ______________________________________________ ;
    p->next = make_node(val, NULL);
    return head;
}
```

A. `for (p = head; p->next != NULL; p = p->next)`

B. `for (p = head->next; p != NULL; p += 1)`

C. `for (p = head; *(p.next) != NULL; *p++)`

Problem 3. (10 points):

Examine the following programs and circle *all* corresponding combinations of output that may be produced when they are run. Note that newlines may be present in the output but are disregarded for this problem. Additionally, assume that all `printf` statements are flushed immediately.

For full credit, you must also *sketch a process tree* next to each program (2 points per problem).

```
int main () {
   fork();
   printf("0");
   if (fork() == 0) {
      /* this next exec prints "1" */
      execl("/bin/echo", "echo", "1", NULL);
      fork();
      printf("2");
   }
   printf("3");
   return 0;
}
```

a. 00112233223333   b. 0132310323   c. 013013   d. 001133   e. 013310   f. 030311

. . . .

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

   return 0;
}
```

a. 213032   b. 123023   e. 130322   f. 132203   c. 220133   d. 022133

Problem 4. (10 points):

Signal handling. Examine the two following programs and determine their corresponding outputs. If there are multiple possible outputs for a program you should list them all and briefly explain why this is the case. Assume that all `printf` statements are flushed immediately.

```
void handler (int sig) {
   int status;
   wait(&status);
   printf("%d\n", WEXITSTATUS(status));
}

int main () {
   int i;
   signal(SIGCHLD, handler);
   for (i=0; i<5; i++) {
      if (fork() == 0) {
         exit(i);
      }
   }
   sleep(10); /* sleep 10 seconds */
   return 0;
}
```

Output:

....

```
int done = 0;

void handler (int sig) {
    printf("0\n");
    done = 1;
}

int main () {
    pid_t pid;
    int status;
    signal(SIGUSR1, handler);
    if ((pid = fork()) == 0) {
        while (!done) { /* void */ }
        printf("1\n");
    } else {
        kill(pid, SIGUSR1);
        if (wait(&status) > 0)
            printf("%d\n", WEXITSTATUS(status));
    }
    if (done)
        printf("3\n");
    else
        printf("4\n");
    return 5;
}
```

Output:

Problem 5. (10 points):

A student has just started on her shell lab and is trying to get foreground processes to run correctly. She has also started to flesh out her SIGCHLD handler. Changes so far are shown below:

```
void sigchld_handler(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0) { ; }
}

void eval(char *cmdline) {
    char *argv[MAXARGS];
    int pid,
        bg = parseline(cmdline, argv);
    if (!bg) {
        if ((pid = fork()) == 0)
            execvp(argv[0], argv);
        addjob(jobs, pid, FG, cmdline);
        waitfg(pid);
    }
}

void waitfg(pid_t pid) {
    int wpid;
    do {
        wpid = wait(NULL);
    } while (wpid != pid);
    deletejob(jobs, pid);
}
```

After compiling and running her shell she's able to enter a command and see its output, but unfortunately she doesn't get her shell prompt back and is unable to enter another command.

1. What is causing the observed bug? Be specific.

2. What would be a good way for the student to fix her code? (Keep in mind that she'll want to go on to complete the lab, so the fix should be forward-looking.)

Problem 6. (8 points):

For this problem you'll complete the implementation of a program that (statically) carries out this command line invocation:

```
/bin/ls | /usr/bin/wc > lswc.out
```

The invocation combines the use of an unnamed pipe with output redirection so that the output of `/bin/ls` is sent to `/usr/bin/wc`, and the resulting output is written to a file named "`lswc.out`".

Part of the program has already been written for you. You'll need to use the following system calls, and should be sure to `close` all file descriptors necessary to ensure your program works correctly.

- `execl(char *path, char *arg0, ...)`: "execs" the program at `path` with the provided arguments. The argument list should be terminated with "`NULL`".
- `dup2(int srcfd, int dstfd)`: duplicates the value of the file descriptor `srcfd` at index `dstfd` in the file descriptor table.

```
int main () {
    int fd, pfd[2];
    fd = open("lswc.out", O_CREAT|O_RDWR); // create and open the file
    pipe(pfd); // pfd[0] is the reading end, pfd[1] the writing end

    if (fork() == 0) { // this child runs /bin/ls








    }





    if (fork() == 0) { // this child runs /usr/bin/wc








    }

    return 0;
}
```