

Full Name: _____

CS 351 Fall 2009 Midterm Exam

October 14, 2009

Instructions:

- This is a closed-book, closed-notes exam.
- Calculators are **not** permitted.
- Write your full name on the front, and make sure your exam is not missing any pages.
- The problems are of varying difficulty: dispatch the easy ones first and tackle the harder ones later.
- Good luck!

Problem 1	(/20)	:
Problem 2	(/10)	:
Problem 3	(/10)	:
Problem 4	(/10)	:
TOTAL	(/50)	:

Problem 1 (20 points)

Circle the single best answer to each of the following multiple choice problems.

1. Which of the following operations should not require operating system kernel intervention?
 - a. signal delivery
 - b. stack allocation
 - c. context switches
 - d. process creation
2. After performing a `fork`, which of the following is true of the child process?
 - a. the child will belong to a separate process group from the parent
 - b. the child's signal handlers will be reset to the default set
 - c. the child will automatically close all inherited open files (except for FDs 0, 1, and 2)
 - d. the child will be allocated its own set of signal vectors (pending & blocked)
3. Without the `exec` system call,
 - a. it would be impossible to create child processes
 - b. it would be impossible to distinguish between parent and child processes after a `fork`
 - c. parent and child processes would always follow the same control flow
 - d. parent and child processes would always be executing the same program
4. Which of the following would work as the definition of a pointer to a signal handler?
 - a. `void (*f)(int)`
 - b. `int *f (int)`
 - c. `void f (int*)`
 - d. `void **f`
5. To stop a running foreground process you might use the Ctrl-Z key combination to send the shell the `SIGTSTP` signal — how would you classify this keypress?
 - a. a fault
 - b. a synchronous interrupt
 - c. an asynchronous interrupt
 - d. a trap

6. Zombie process are created
 - a. only when a parent process terminates before its children
 - b. only when a child process terminates before its parent
 - c. when either a parent or child process terminates
 - d. only if the shell forgets to reap its children
7. Non-local jumps (with `setjmp` and `longjmp`) allow the programmer to “jump” from the context of some function `foo` to another function `bar`, providing
 - a. `bar` precedes `foo` in the call stack
 - b. `foo` precedes `bar` in the call stack
 - c. `bar` is defined before `foo` in the program source code
 - d. `bar` is installed as a signal handler for the process
8. A severe limitation of unnamed pipes is that
 - a. only processes that know their names can communicate over them
 - b. only parent and child processes can communicate
 - c. they can only be used for IPC when created by a common ancestor
 - d. they do not support synchronization
9. One argument for the implementation of system-wide open file descriptions is that
 - a. separate processes can share a common file position
 - b. separate processes can read from the same file
 - c. they implicitly support file locking mechanisms
 - d. a process may use them to redirect I/O before `exec`-ing a new program
10. Standard library (buffered) I/O is *not* a good choice when:
 - a. performing lengthy reads on a short-count prone file type
 - b. many small write operations need to be performed separately
 - c. read and write operations are to be interleaved on a short-count prone file
 - d. complex data formatting is required

Problem 2 (10 points):

For each of the following two programs, you are to:

- draw a corresponding process tree (showing outputs and any synchronization points), and
- circle all the listed outputs that could be produced by the program

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

process tree:

- a. 00112233 b. 01301322 c. 00221133 d. 00132132 e. 01230123 f. 00231231
-

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

process tree:

- a. 00112233 b. 01301322 c. 00221133 d. 00132132 e. 01230123 f. 00231231

Problem 3 (10 points):

Fill in the blanks to complete the following function, which robustly reads **n** bytes from the file descriptor **fd** into the buffer pointed to by **buf**. The only time a short count should be returned is when the read system call returns 0 prematurely (note that you may assume no read errors occur).

```
int robust_read (int fd, char *buf, int n) {  
    int nleft = _____,  
        nread;  
  
    while (_____) {  
        nread = read(fd, buf, _____);  
        if (nread == 0)  
            break;  
  
        nleft = _____  
        buf   = _____  
    }  
  
    return _____  
}
```

Problem 4 (10 points):

A student just starting to work on the shell lab comes up with the following code for her SIGCHLD handler.

```
void sigchld_handler(int sig) {
    pid_t pid;
    int status;
    struct job_t *j;

    while ((pid = wait(&status)) > 0) {
        j = getjobpid(jobs, pid);
        j->state = UNDEF;
        j->pid = j->jid = 0;
    }
}
```

After testing her code, however, the student finds that after creating a number of background jobs, sometimes things work just fine, and sometimes her shell hangs for long periods of time.

Can you identify the problem and suggest a fix?