CS 110 Spring 2019 Midterm Answer Key

1. Double Trouble

```
/* Function: writeBuf
 * ------------------
 * Writes the contents of buf with size len to the file descriptor fdOut
 * @param fdOut - the file descriptor where the data will be written
 * @param buf - the buffer to write
 * @param len - the number of bytes in the buffer
 */

void writeBuf(int fdOut, char *buf, ssize_t len) {
    ssize_t bytesWritten = 0;
    while (bytesWritten < len) {
        bytesWritten += write(fdOut, buf + bytesWritten, len - bytesWritten);
    }
}

/* Function: dualEcho
 * ------------------
 * Writes the all input from fdIn to both fdOut1 and fdOut2
 * @param fdIn - the file descriptor to read from (until EOF)
 * @param fdOut1 - the first output file descriptor (ignore if -1)
 * @param fdOut2 - the second output file descriptor (ignore if -1)
 */
void dualEcho(int fdIn, int fdOut1, int fdOut2) {
    // read from fdIn and echo to fdOut1 and fdOut2
    // if either fdOut1 or fdOut2 are -1, don't write
    char buf[1024];
    while (true) {
        ssize_t bytesRead = read(fdIn, buf, sizeof(buf));
        if (bytesRead <= 0) break;
            if (fdOut1 != -1) {
                writeBuf(fdOut1, buf, bytesRead);
            }
            if (fdOut2 != -1) {
                writeBuf(fdOut2, buf, bytesRead);
            }
    }
}

int main(int argc, char **argv)
{
    if (argc < 3) {
        printf("Usage:\n\t %s prog1 prog2\n",argv[0]);
        return 0;
    }
    char *progA[] = {argv[1], NULL};
    char *progB[] = {argv[2], NULL};
    // your code here:
    int fdsSupplyA[2];
    int fdsSupplyB[2];

    pid_t pids[2];

    pipe2(fdsSupplyA,O_CLOEXEC);
    pids[0] = fork();
    if (pids[0] == 0) { // child
        // for supply
        // child will read from pipe
        close(fdsSupplyA[1]); // no need to write

        // duplicate STDIN
        dup2(fdsSupplyA[0],STDIN_FILENO);
        close(fdsSupplyA[0]); // no need for this any more

        execvp(progA[0], progA);
        printf("Execvp for progA failed!\n");
        exit(0);
    }

    pipe2(fdsSupplyB,O_CLOEXEC);
    pids[1] = fork();
    if (pids[1] == 0) { // child
        // for supply
        // child will read from pipe
        close(fdsSupplyB[1]); // no need to write

        // duplicate STDIN
        dup2(fdsSupplyB[0],STDIN_FILENO);
        close(fdsSupplyB[0]); // no need for this any more

        execvp(progB[0], progB);
        printf("Execvp for progB failed!\n");
        exit(0);
    }

    // parent will write to supply
    close(fdsSupplyA[0]); // no need to read
    close(fdsSupplyB[0]); // no need to read

    dualEcho(STDIN_FILENO, fdsSupplyA[1], fdsSupplyB[1]);
    close(fdsSupplyA[1]);
    close(fdsSupplyB[1]);

    waitpid(pids[0], NULL, 0);
    waitpid(pids[1], NULL, 0);

    return 0;
}
```

1d. Neither works.

A: If a child produces more than 64KB output its output pipe fills up. The child’s next write() blocks (until parent consumes). Since child is blocked, it cannot read() from its stdin and its pipe buffer fills causing parent to block on write(). Since parent writes its full stdin to the children before consuming from the children's output pipes, parent will get hung indefinitely on write().

B: The self-stopped program will not read from it's stdin. Eventually it's pipe buffer will fill up and then the parent's write() calls will block causing the parent process to hang indefinitely.

2. Signal Puzzle

a. Six possible outputs:

```
*******************

I made it here!
child!
parent!


*******************

parent!
child!

******************

child!
parent!

******************

child!
I made it here!
parent!

*****************

I made it here!
parent!
child!

******************

parent!

******************
```

b. **If exit(0) was removed, there would be another “parent!” printed, and there would also be a “And I made it here!” printed.**

3. Sleep Sort

a.
```
void sleepsort(vector<int> numbers) {
    for (int n : numbers) {
        pid_t pid = fork();
        if (pid == 0) {
            sleep(n);
            //usleep(n);
            printf("%d\n",n);
            exit(0);
        }
    }
    for (size_t i=0; i < numbers.size(); i++) {
        waitpid(-1, NULL, 0);
    }
}
```

b. This is a giant race condition because all of the sleeps are happening at the same time, and except for the fact that they are different amounts of seconds, you don't really know when each one will stop first. Because of the granularity of the `sleep()` value in seconds, the program will work as expected.

c. Possible reasons for `usleep()` to make the program fail:

- The looping will take more than some number of microseconds, and for small integers, there just won't be enough granularity to have distinct sleep times
- Even if the microsecond timing is perfect, there are still very tight race conditions, and things like `printf` will take enough microseconds to destroy timing granularity.

4. File Systems

a. Pros of a 4096 byte inumber:
- Files fit into fewer blocks, meaning that there is less overhead per file.
- It takes less time to read files, because there are fewer blocks to look through
- There would be less disk fragmentation because files would have fewer blocks

Con of a 4096 bytes inumber:
- Every file takes at least 4096 bytes, so if there are a lot of small files, this can waste space

b. Larger inode
- Possible uses for extra 32-bytes
    - more direct or indirect blocks, so files can be bigger
    - more metadata about the files
    - for very small files, keep the file data itself in the inode
- Tradeoffs to having larger inode
    - metadata is overhead, which takes away from usable payload space
    - more complicated handling of files could be inefficient

c. 2-byte -vs- 4-byte block number

- Main limitation to a 2-byte inumber: file count is limited to 2^16 (64K) flies for the entire file system

- Tradeoff for 4-byte block number: Needs more metadata space (2x) to hold the numbers themselves.

d. A directory is just a file in the sense that it is accessed like any other file (with direct and indirect blocks), and it simply holds payload data that refers to other files. It isn't a separate part of the file system, and there is no need for the filesystem code to treat a directory differently when reading and writing to their associated files.

5. Safer SIGINT

Robust solution (will handle three in a row if the last two are within 500ms of each other):

```
void sigint_handler(int sig) {
    // start: variable to hold the start time between calls to this function
    static int start;
    if (!first_sigint) {
        first_sigint = true;
        start = get_ms_time();
    } else {
        int end = get_ms_time();
        int duration = (end - start);
        if (duration < kTimeThreshold) {
            okay_to_quit = true;
        } else {
            start = end;
        }
    }
}
```

Other solution (will capture two in a row, but not three):

```
void sigint_handler(int sig) {
    // start: variable to hold the start time between calls to this function
    static int start;
    if (!first_sigint) {
        first_sigint = true;
        start = get_ms_time();
    } else {
        int duration = (get_ms_time() - start);
        first_sigint = false;
        if (duration < kTimeThreshold) {
            okay_to_quit = true;
        }
    }
}
```