CS110 Midterm Answer Key

Winter 2020
Instructors: Chris Gregg and Nick Troccoli

1. Pipe Queue (20 points)

```
PipeQueue::PipeQueue() : size(0) {
    pipe(fds);
}

PipeQueue::~PipeQueue() {
    close(fds[0]);
    close(fds[1]);
}

void PipeQueue::enqueue(string s) {
    /** The enqueue method writes a length-prefixed string to
     *  the pipe file descriptor backing the queue.
     *  The prefix should be a 2-byte (16-bit) number, and the
     *  endianness of the number must be accounted for in the
     *  dequeue() method.
     *  You may use write, dprintf, or stdio_filebuf and ostreams
     *  to write your data to the pipe, but you must ensure that
     *  all bytes are correctly sent.
     *
     *  @param s: a C++ string to enqueue
     *  @return: void
     */

    // send two-byte size
    uint16_t len = (uint16_t)s.size();
    write(fds[1], &len, 2);
    dprintf(fds[1], "%s", s.c_str());
    size++;
}

string PipeQueue::dequeue() {
    /** The dequeue method reads a length-prefixed string
     *  from the pipe file descriptor backing the queue.
     *  The prefix will be a 2-byte (16-bit) number, with
```

```
     *  an endianness as defined in the enqueue() method.
     *  You should use the read() function to read in both
     *  the prefix and the string data.
     *  Note: NO loops are allowed for this method. A solution
     *  with loops will receive zero points.
     *
     *  @param: none
     *  @return: a C++ string dequeued from the queue
     */

    // read two bytes of length
    uint16_t len;
    read(fds[0], &len, 2);
    char strBuf[len + 1]; // needs null terminator

    read(fds[0], strBuf, len);
    strBuf[len] = '\0';
    size--;
    return string(strBuf);
}
```

2. Fork-Chat (20 points)

Short Answer;

1. Read_fd must be accessed by the child in both the signal handler (for closing it when terminated) and the main code (for reading from file and printing out messages).
2. When we fork a new process, it gets a copy of the virtual address space, meaning that if we heap-allocate memory and then spawn a new process, we must free that memory in both the parent and child processes.
3. We do not know whether our child process or the other user's parent process will be the first to open (and possibly create) the file, and if our child process is first we must create the file so we have something to read from.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <string.h>
#include <signal.h>

#define kMaxLine 256

int read_fd;


int read_line(int fd, char *buf, int buf_len) {
    // reads up to and including a newline (or EOF) from fd
    // replaces newline with null terminator
    // only reads up to buf_len - 1 bytes
    // @param fd: the file descriptor to read from
    // @param buf: the buffer to populate
    // @param buf_len: the size of the buffer
    // @return: the number of bytes read, not including
    //          the newline. Can be 0 if at end of file
    int bytes_read = 0;
    while (bytes_read < buf_len - 1) {
        int actually_read = read(fd, buf + bytes_read, 1);
        if (actually_read == 0 || buf[bytes_read] == '\n') break;
        bytes_read++;
    }
    buf[bytes_read] = '\0';
    return bytes_read;
}

void sigusr1_handler(int sig) {
    close(read_fd);
    exit(0);
}

void populate_filenames(char *otheruserid, char **read_filename, char
**write_filename) {
    const char *tmp_dir = "/tmp/";
    *read_filename = (char *)malloc(strlen(otheruserid) +
strlen(tmp_dir) + 1);
    strcpy(*read_filename, tmp_dir);
    strcat(*read_filename, otheruserid);

    char *myuserid = getenv("USER");
```

```
    *write_filename = (char *)malloc(strlen(myuserid) +
strlen(tmp_dir) + 1);
    strcpy(*write_filename, tmp_dir);
    strcat(*write_filename, myuserid);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage:\n\t%s other_id\n", argv[0]);
        return 0;
    }

    char *write_filename; // will need to free
    char *read_filename; // will need to free

    printf("Welcome to chat! Go ahead and say something
already!\n\n");
    populate_filenames(argv[1], &read_filename, &write_filename);

    umask(0);
    pid_t pid_or_zero = fork();
    if (pid_or_zero == 0) { // child
        read_fd = open(read_filename, O_RDONLY | O_CREAT, 0666);
        lseek(read_fd, 0, SEEK_END); // start cursor at end of file
                                     // in case of resumed chat
        free(write_filename); // no longer needed
        free(read_filename);
        signal(SIGUSR1, sigusr1_handler);
        while (true) {
            char buf[kMaxLine];
            int bytes_read = read_line(read_fd, buf, kMaxLine);
            if (bytes_read == 0) {
                usleep(1000);
            } else {
                printf("Message: %s\n", buf);
            }
        }
    } else {
        int write_fd;
        write_fd = open(write_filename, O_WRONLY | O_CREAT, 0666);
        lseek(write_fd, 0, SEEK_END); // start cursor at end of file
                                      // in case of resumed chat
        free(read_filename);
```

```
        free(write_filename);

        char buf[kMaxLine];
        while (read_line(STDIN_FILENO, buf, kMaxLine) != 0) {
            dprintf(write_fd, "%s\n", buf);
        }
        close(write_fd);
        kill(pid_or_zero, SIGUSR1);
        waitpid(pid_or_zero, NULL, 0);
    }
    return 0;
}
```

3. Signal mystery (20 points)

Three possible outputs:

I am the child
I am the parent
sahami

I am the parent
I am the child
sahami

I am the parent
sahami

4. Short Answer (20 points)

a) A benefit is quicker lookup of an entry, as we can use binary search to search for an entry name rather than looping over each entry. A drawback is when we add or rename

directory entries, we must potentially re-sort the elements, which is extra work as opposed to the unsorted version.

b) One benefit of this is files are stored more directly, as we can fit more block numbers directly (or indirectly) in an inode. A drawback of this is more space may be taken up on disk by non-file data.

c) The first singly-indirect block in this case would hold file block numbers 0 - 255, inclusive. The doubly-indirect block in this case would hold block numbers 7*256 through 7*256 + 256*256 - 1, inclusive.

d) The primary difference is threads share the same virtual address space, while processes have copies of their virtual address space. This could be seen as a benefit for threads in that it allows for easier cross-thread communication, but a drawback in that it allows for more data races between threads. This could be seen as a benefit for processes in that it provides more isolation and security, but a drawback in that it makes inter-process communication more difficult.

e) The race condition is that it's possible for both threads to check whether counter is equal to 10 at roughly the same time (before either of them has a chance to increment it). Thus, both threads could believe that counter is 9, and go ahead and increment it, at which point counter will be 11. Even with overflow, the program could continue like this. Thus, the program would infinite loop. To fix this, we must lock the mutex before line 3 (so that another worker thread can access it once we start looping), and unlock it after lines 4 (so that another worker thread can access it while we are looping) and 5 (so that we unlock before terminating).