CS110 Midterm Exam

Question Booklet

CS110 Winter 2020 – Instructors: Chris Gregg and Nick Troccoli

You may not use any internet devices. You will be graded on functionality – but good style saves time and helps graders understand what you were attempting. You have 80 minutes. We hope this exam is an exciting journey.

Note: DO NOT WRITE in this booklet. Only work in the answer booklet will be graded.

1) Pipe Queue **20 Points/80 Total**

Note: you may need to scroll horizontally to fully to view blocks of code.

Using what you know about UNIX pipes, use a *single* pipe in a single process to finish writing the C++ class below called `PipeQueue`, that acts as a queue data structure for C++ strings. In other words, strings are enqueued and dequeued with first-in-first-out behavior. Here's an example of how this data structure might be used:

```
[CODE START]
int main(int argc, char *argv[]) {
    PipeQueue q;
    q.enqueue("Hello,");
    q.enqueue("World!");
    while (!q.empty()) {
        cout << q.dequeue() << endl;
    }
    return 0;
}
[CODE END]
```

As you are well aware, C-strings are null-terminated, but in order to know the size of a C-string, you must read one character at a time until you find the null-terminator. Another type of string, called a *length-prefixed* string is used in some programming languages (most notably, in the *Pascal* language), and for this problem, you will use this type of string. A length-prefixed string encodes the length of the string at the beginning of the data for the string. Once the size is read, the number of bytes remaining (containing the actual string data) can be read.

In Linux, the default capacity of the write end of a pipe is 2^{16} (65,535) bytes, meaning that we are safe using a 16-byte unsigned short for the length of a string in our queue. Therefore, when enqueuing, the strings should be prefixed by a 2-byte value for the length of the string data, and when dequeueing, the first two bytes will be the 2-byte prefix, as well. *Note: we will not take points off for endianness issues for the prefix (though if you were to code this up for real you would need to check this carefully). You simply need to ensure that both bytes end up being written to, and read from, the pipe at the appropriate time*.

For `enqueue`, you may use `write`, `dprintf`, or `stdio_filebuf` and `ostreams`, and you must ensure that all bytes are correctly written to the pipe. Note, however, that you **must use `write` to write the 2-byte short for the length** - the reason for this is if you use `dprintf` or other

similar functions, it will write the number as a string with one character per digit, whereas we wish to write the number as a 2 byte short. You may assume that `write` writes all requested bytes.

For `dequeue`, you must use the `read` function, and *you are not allowed to use any loops* in the method - doing so will result in no credit. You may assume that `read` reads all requested bytes.

You must ensure all file descriptors (other than STDIN, STDOUT and STDERR) are closed when no longer used. You may assume the following:

- all system calls succeed
- there will be space in the pipe when enqueueing
- there will be at least one element in the queue when dequeueing

Here is the provided interface and partial implementation. You will complete the `constructor`, `destructor`, `enqueue`, and `dequeue` functions. You should only add code within the specified methods.

```
[CODE START]
class PipeQueue {
    public:
        PipeQueue(); // constructor
        bool empty();

        /** 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.
         *  Updates state to reflect a new string was
         *  enqueued.
         *
         *  @param s: a C++ string to enqueue
         *  @return: void
         */
        void enqueue(string s);

        /** 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.  Updates state to
         * reflect a new string was dequeued.
         *
         *  @param: none
```

```
         *  @return: a C++ string dequeued from the queue
         */
        string dequeue();

        ~PipeQueue(); // destructor

    private:
        int fds[2];
        size_t size;
};

bool PipeQueue::empty() {
    return size == 0;
}

PipeQueue::PipeQueue() : size(0) {
    // TODO: write constructor
}

PipeQueue::~PipeQueue() {
    // TODO: write destructor
}

void PipeQueue::enqueue(string s) {
    // TODO: write enqueue function
}

string PipeQueue::dequeue() {
    // TODO: write dequeue function

}
[CODE END]
```

```
[ANSWER START]
PipeQueue::PipeQueue() : size(0) {
    // TODO: write constructor
}

PipeQueue::~PipeQueue() {
    // TODO: write destructor
}

void PipeQueue::enqueue(string s) {
    // TODO: write enqueue function
}

string PipeQueue::dequeue() {
    // TODO: write dequeue function
}
[ANSWER END]
```

2) Fork Chat **20 Points/80 Total**

Note: you may need to scroll horizontally to fully to view blocks of code.

In this problem, you're going to complete a *chat* program **in C** that allows two users logged onto the same machine (e.g., `myth55`) to chat with each other by each running their own instance of the program. The program works by using temporary files that are readable by each user, and it achieves asynchronous behavior (i.e., a user can receive a message while typing) by spawning a child process via `fork`.

The program works as follows:

1. To launch the program, a user with ID `userid` provides ID `other_id` of the person they want to chat with as a command-line argument. Two files are opened (or created):
    - `/tmp/[userid]` that the user will write chat text into, and that the other user will read from (`[userid]` is replaced with the user's actual `userid`, e.g., `cgregg`).
    - `/tmp/[other_id]` that the other user will write chat text into, and that the user will read from (`[other_id]` is replaced with the command line argument, e.g., `troccoli`).
2. The initial parent process acts as the writer, and forks a child process that will act as the reader:
    - The reader process reads lines from `/tmp/[other_id]` and prints `Message: [MESSAGE]` (where [MESSAGE] is the message just read) on the screen after each line read.
    - The writer process reads from `stdin` and writes one line at a time to `/tmp/[userid]`.
3. When the user types `ctrl-d` on a line on its own, the program gracefully ends, meaning both the reader and writer processes. The other user's program continues to run until that user also types `ctrl-d`. In order to terminate, the child must receive a signal from the parent (in this case, a `SIGUSR1` signal, as defined in part (E) below). **All memory must be correctly cleaned up and all file descriptors except for `stdin`, `stdout` and `stderr` must be closed when the child process exits.**

The starter code for this problem is shown below. You will answer questions about it and complete portions of it in the following parts. For writing code, you should fill in code only in the locations specified, and should not add code outside these locations. You may assume that all system calls succeed.

```
[CODE START]
#define kMaxLine 256
```

```
static int read_fd;

int read_line(int fd, char *buf, int buf_len) {
    // TODO: Your code here
    return 0;
}

void sigusr1_handler(int sig) {
    // TODO: Your code here
}

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! Say something!\n\n");

    /* afterwards, read_filename and write_filename point
     * to malloc'ed memory storing the corresponding filenames.
     * Its implementation is omitted, but assumed to work.
     */
    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

        // TODO: Your code here
    } 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

        // TODO: Your code here
    }
    return 0;
}
[CODE END]
```

Part A.

Answer the following questions about the starter code:

1. The `read_fd` file descriptor is defined globally. Why might this be the case?
2. The comment above the call to `populate_filenames` says that it uses `malloc` to provide memory for `write_filename` and `read_filename`. Based on what we know about virtual memory and `fork()`, how must we go about freeing this memory?
3. The first thing the child process does is to open a read-only file and create it if it doesn't exist. If the program we are chatting with is the one writing, why does this program need to create the file if it doesn't exist?

```
[ANSWER START]
1.
2.
3.
[ANSWER END]
```

Part B.

Write the `read_line` function for this program. The function should populate `buf` with characters read from `fd` until receiving a newline, or until the end of the file. The string in `buf` should be a properly null-terminated C-string.

```
[CODE START]
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

    // TODO: Your code here
}
[CODE END]
```

```
[ANSWER START]
int read_line(int fd, char *buf, int buf_len) {
    // TODO: Your code here
}
[ANSWER END]
```

Part C.

Write the code for the parent process (after the `fork`) below. The parent process continually reads lines from `STDIN_FILENO` and writes each line (including a newline) to `write_fd`. Lines are limited to `kMaxLine` characters, including the null-terminator.

If the user types `ctrl-d`, this closes `stdin`, and the program should gracefully finish.

Before the parent process returns:

- All memory must be correctly cleaned up.
- All file descriptors except for `stdin`, `stdout` and `stderr` must be closed.
- The parent process must send a `SIGUSR1` signal to the child to notify it to exit.
- The parent must wait for the child to finish before itself exiting.

```
[CODE START]
} 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
                                  // of file in case
                                  // of resumed chat
    // TODO: Your code here

}
[CODE END]
```

```
[ANSWER START]
} else {
    int write_fd;
    write_fd = open(write_filename, O_WRONLY | O_CREAT, 0666);
    lseek(write_fd, 0, SEEK_END);

    // TODO: Your code here

}
[ANSWER END]
```

Part D.

Write the signal handler for the `SIGUSR1` signal.

```
[CODE START]
void sigusr1_handler(int sig) {
    // TODO: Your code here

}
[CODE END]
```

```
[ANSWER START]
void sigusr1_handler(int sig) {
    // TODO: Your code here
}
[ANSWER END]
```

Part E.

Write the code for the child process below. The child process continually reads lines from `read_fd` and prints the following for each line read (where [MESSAGE] is the message read):

```
Message: [MESSAGE]
```

Lines are limited to `kMaxLine` characters, including the null-terminator. If an attempt to read a line returns 0 bytes, then the process does not print anything, and instead sleeps for 10000 microseconds, using the `usleep(1000)` function.

In order to terminate, the child must receive a signal from the parent (in this case, a `SIGUSR1` signal, as defined in part (E) below). **All memory must be correctly cleaned up and all file descriptors except for `stdin`, `stdout` and `stderr` must be closed when the child process exits.**

```
[CODE START]
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
    // TODO: Your code here

}
[CODE END]
```

```
[ANSWER START]
if (pid_or_zero == 0) { // child
    read_fd = open(read_filename, O_RDONLY | O_CREAT, 0666);
    lseek(read_fd, 0, SEEK_END);

    // TODO: Your code here
}
[ANSWER END]
```

3) Signal Mystery **20 Points/80 Total**

Note: you may need to scroll horizontally to fully to view blocks of code.

Assume that the user `sahami` runs the following program. List all potential outputs. Assume that all system calls succeed (including `execvp`). You should also assume that calls to `fprintf` are atomic. You may not make *any* assumptions about when a process is definitely running on, or definitely off the processor.

Note: The `whoami` program prints the current userid (in this case, `sahami`) and a newline to `stdout`.

```
[CODE START]
int pid_or_zero;

void sig_handler(int sig) {
    fprintf(stderr, "I am the %s\n",
            pid_or_zero == 0 ? "child" : "parent");
}

int main(int argc, char *argv[]) {
    signal(SIGUSR1, sig_handler);

    int fds[2];
    pipe(fds);
    pid_or_zero = fork();

    if (pid_or_zero == 0) {
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        char *argv[] = {"whoami", NULL};
        kill(getppid(), SIGUSR1);
        execvp("whoami", argv); // assume this always works
        kill(getppid(), SIGUSR1);
        exit(0);
    }
    kill(pid_or_zero, SIGUSR1);
    close(fds[1]);
```

```
    waitpid(pid_or_zero, NULL, 0);

    char userid[9]; // max userid is 8
    for (int i = 0; i < 9; i++) {
        read(fds[0], userid + i, 1);
        if (userid[i] == '\n') {
            userid[i] = '\0';
            break;
        }
    }
    printf("%s\n", userid);
    close(fds[0]);
    return 0;
}
[CODE END]
```

```
[ANSWER START]
List the possible outputs:

Output 1:


Output 2:


etc...
[ANSWER END]
```

4) Short Answer **20 Points/80 Total**

Note: you may need to scroll horizontally to fully to view blocks of code.

Part A.

For assignment 2, we saw that a directory is modeled the same as a file, except its contents are a list of unsorted directory entries. A fellow filesystem designer suggests storing the entries in sorted order. In 3 sentences or less for each, explain 1 benefit and 1 drawback in terms of performance of this change.

```
[ANSWER START]

[ANSWER END]
```

Part B.

We have seen that an inode stores only 8 block numbers, and if a file has more block numbers than this, we must use approaches such as singly and doubly indirect blocks to store them. A fellow filesystem designer suggests adding more space in each inode to store block numbers. In three sentences or less for each, explain 1 benefit and 1 drawback in terms of performance of increasing the number of block numbers stored in an inode.

```
[ANSWER START]

[ANSWER END]
```

Part C.

For each bullet below, specify the range of file block numbers (as a number or mathematical expression) that could be stored there, assuming the same filesystem implementation as in assign2. File block number is defined the same as in assign2 - it is a data block index for the file, starting at 0; so the first block of file data has number 0, the second block of file data has number 1, etc. As an example, the first block number entry in an inode using direct addressing stores file block number 0, and the second block number entry in an inode using direct addressing stores file block number 1. You do not need to explain your answers.

- The first singly-indirect block in an inode using indirect addressing
- The doubly-indirect block in an inode using indirect addressing

```
[ANSWER START]

[ANSWER END]
```

Part D.

Name the primary difference between threads and processes, and explain why this difference could be seen as a benefit and as a drawback for each of threads and processes.

```
[ANSWER START]

[ANSWER END]
```

Part E.

Your coworker has implemented a program using threads that they need your help debugging. Specifically, they say that *sometimes* the program infinite loops, but they are not sure why! You take a look at their code, which is shown below:

```
[CODE START]
 1 void worker(int& counter) {
 2      while (true) {
 3          if (counter == 10) break;
 4          counter += 1;
 5      }
 6 }

 7 int main(int argc, char *argv[]) {
 8      int counter = 2;
 9      thread worker1(worker, ref(counter));
10      thread worker2(worker, ref(counter));
11      worker1.join();
12      worker2.join();
13      return 0;
14 }
[CODE END]
```

- Explain the race condition that makes it possible for the code above to infinite loop.
- Your coworker learns that they should add a mutex lock around a critical region to prevent the race condition. Assuming we declare a mutex in `main` that we pass by reference to each of the threads, note the single line number immediately before which we should lock, and the *two* line numbers immediately after which we should unlock.

```
[ANSWER START]

[ANSWER END]
```

5) Function Reference

Note: you may need to scroll horizontally to fully to view blocks of code.

```
[CODE START]
// filesystem access
int open(const char *path, int oflag, ...); // returns descriptor

ssize_t read(int fd, char buffer[], size_t len); // returns num read,
 0 at eof

ssize_t write(int fd, char buffer[], size_t len); // returns num writ
ten

int printf(const char *format, ...); // prints to STDOUT

int dprintf(int fd, const char *format, ...); // prints to file descr
iptor

int fprintf(FILE *stream, const char *format, ...); // prints to stre
am

int close(int fd); // ignore retval

int pipe(int fds[]); // argument should be array of length 2, ignore
retval

int pipe2(int fds[], int flags); // common flag: O_CLOEXEC

int dup2(int old, int new); // ignore retval

#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

// exceptional control flow and multiprocessing
pid_t fork();

pid_t waitpid(pid_t pid, int *status, int flags);
```

```
typedef void (*sighandler_t)(int sig);

sighandler_t signal(int signum, sighandler_t handler); // ignore retv
al

int sigsuspend(const sigset_t *mask); // ignore retval

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); // i
gn. retval

int execvp(const char *path, char *argv[]); // ignore retval

int kill(pid_t pid, int sig); // ignore retval

int setpgid(pid_t pid, pid_t pgid); // ignore retval

#define WIFEXITED(status) // macro
#define WIFSTOPPED(status) // macro
#define WEXITSTATUS(status) // macro
[CODE END]
```