# CS360 Final - May 2, 2019 - James S. Plank

Put all answers on the answer sheet. In all of these questions:

- Don't worry about header files in any of this work. Just assume that they are there.
- In your code, error check, and simply exit if you see an error. You don't need to print anything out.
- Assume that lines of text are less than 1000 characters.

---

## Question 1 (16 points)

Please write the **NAME**, **SYNOPSIS**, **DESCRIPTION** and **RETURN VALUES** parts of the man page for **fork()**. For reference, here are those sections for the **pipe()** man page. Don't worry about the **include** statement in the **SYNOPSIS**.

```
NAME
     pipe -- create descriptor pair for interprocess communication

SYNOPSIS
     #include <unistd.h>

     int pipe(int fd[2]);

DESCRIPTION
     The pipe() system call creates an I/O mechanism called a pipe
     and returns two file descriptors, fd[0] and fd[1].  fd[0] is
     opened for reading and fd[1] is opened for writing.  When the
     pipe is written using the descriptor fd[1] up to {PIPE_BUF} (see
     sysconf(2V)) bytes of data are buffered before the writing
     process is blocked.  A read only file descriptor fd[0] accesses
     the data written to fd[1] on a FIFO (first-in-first-out) basis.

     Read calls on an empty pipe (no buffered data) with only one
     end  (all write file descriptors closed) return an EOF (end
     of file).

     A SIGPIPE signal is generated if a write on a pipe with only
     one end is attempted.

RETURN VALUES
     pipe() returns:

     0    on success.

     -1   on failure and sets errno to indicate the error.
```

## Question 2 - 16 points

Using pthreads and "sockettome", write the program **telnet.c**. This program takes two command line arguments -- a host and a port, and makes a client connection to a server which is serving a socket on that host/port. Once the connection is established, **telnet** does two things -- read lines from standard input and write them to the server, and read lines from the server and write them to standard output. Don't worry about SIGPIPE.

## Question 3 - 16 points

This question concerns something we covered in class called "longjmp-ing up the stack." Please explain what this is, giving an example program that does it, and explain why it is bad, both in general and for the example.

# CS360 Final - May 2, 2019 - James S. Plank

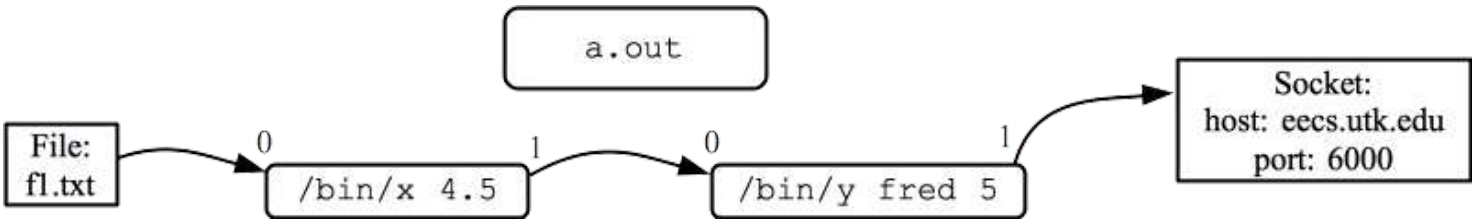
## Question 4 (26 points)

Write a program, that, when you compile it to **a.out** and run it, creates the situation pictured below. Some detail:

- In the picture, there are three processes, represented by rounded rectangles. The **a.out** process should be the parent of the other two.
- The processes should be executing the executables and command line arguments shown in the rounded rectangles.
- In the picture, arrows represent input/output, either between processes, to/from files, or to/from a socket served by another machine.
- The arrows are labeled with the file descriptor numbers to/from the processes.
- The **a.out** process should not exit until all processes shown are complete.
- There should be no file descriptors open whose values are higher than two.

I suggest that you do this on scratch paper first, and then copy it over to the answer sheet, so that it is neat.

Also, if you want, you can write some description of what you are trying to do, which may help me grade if your code is not clear.




---

## Question 5 (26 points)

On the next page is a multi-threaded program that uses locks and condition variables. Take a little time to read it and familiarize yourself with what it does. Suppose you compile it to **a.out**. When you run it, you specify what the threads to on the command line. For example, if I run

```
UNIX> a.out SP
```

The output will be "A", because there will be one thread, whose **id** is "A", which will sleep for a second, then print "A" and exit.

For this question, I am going to give you 8 separate executions of this program. You are to tell me **all** of the possible outputs. Do this by choosing from the multiple choice answers on the answer sheet -- they are arranged alphabetically, so that you have an easier time finding them. If there is deadlock, just assume that the program exits at that point.

**Run 1:** `a.out P P P`

**Run 2:** `a.out PP PP`

**Run 3:** `a.out SPA XP`

**Run 4:** `a.out XP YP SPBSPA`

**Run 5:** `a.out PE PE PE`

**Run 6:** `a.out XP YP SPAB`

**Run 7:** `a.out SPAA XP XP`

**Run 8:** `a.out XP AP`

## Question 5A

Suppose I wanted make the program above print "A B A B A B", printing one character per second. Obviously, I could do:

```
UNIX> a.out PSSPSSP SPSSPSSP
```

However, if I make use of the condition variables, I can do this so that there are exactly five S's on the command line (there are 9 S's on the command line above). Give me a command that does this.

# CS360 Final - May 2, 2019 - James S. Plank

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

struct ts {                                 /* This is the struct passed to each thread. */
  pthread_mutex_t *lock;                    /* The lock and cv's are shared among all threads. */
  pthread_cond_t *cv1, *cv2;
  char *command;                            /* Each thread gets its own command string and id. */
  int id;
};

void *thread(void *arg)                      /* Each thread locks the mutex, and then runs its */
{                                            /* own set of commands.  The commands are characters */
  struct ts *T;                              /* in the command string.  Each thread has an id, */
  int i;                                     /* represented by a character 'A', 'B', 'C', etc. */

  T = (struct ts *) arg;

  pthread_mutex_lock(T->lock);
  for (i = 0; T->command[i] != '\0'; i++) {
    if (T->command[i] == 'S') {
      pthread_mutex_unlock(T->lock);
      sleep(1);
      pthread_mutex_lock(T->lock);
    }
    if (T->command[i] == 'X') pthread_cond_wait(T->cv1, T->lock);
    if (T->command[i] == 'Y') pthread_cond_wait(T->cv2, T->lock);
    if (T->command[i] == 'A') pthread_cond_signal(T->cv1);
    if (T->command[i] == 'B') pthread_cond_signal(T->cv2);
    if (T->command[i] == 'P') { printf("%c ", T->id); fflush(stdout); }
    if (T->command[i] == 'E') { printf("\n"); exit(1); }
  }
  pthread_mutex_unlock(T->lock);
  return NULL;
}

int main(int argc, char **argv)
{
  pthread_mutex_t lock;
  pthread_cond_t cv1, cv2;
  int i;
  pthread_t *tids;
  struct ts *T;
  void *dummy;

  pthread_mutex_init(&lock, NULL);   /* Create the lock and two CV's */
  pthread_cond_init(&cv1, NULL);
  pthread_cond_init(&cv2, NULL);

  T = (struct ts *) malloc(sizeof(struct ts)*(argc-1));       /* Create the threads.  Each thread */
  tids = (pthread_t *) malloc(sizeof(pthread_t)*(argc-1));    /* has its own copy of the struct, */
  for (i = 0; i < argc-1; i++) {                              /* with its unique id and command. */
    T[i].id = 'A' + i;                                        /* The struct shares the lock and */
    T[i].lock = &lock;                                        /* condition variables. */
    T[i].cv1 = &cv1;
    T[i].cv2 = &cv2;
    T[i].command = argv[i+1];
    pthread_create(tids+i, NULL, thread, (void *) (T+i));
  }
  for (i = 0; i < argc-1; i++) pthread_join(tids[i], &dummy);  /* Wait for the threads to exit. */
  printf("\n");
  return 0;
}
```

# Prototypes of various useful system and library calls

```
int wait(int *stat_loc);
int dup2(int fildes, int fildes2);
int pipe(int fildes[2]);

int open(const char *path, int oflag, ...);
int close(int fildes);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);

char *strcpy(char *destination, char *source);
char *strdup(char *source);
int   strcmp(char *s1, char *s2);
```

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

int  setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
int  sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

```
int execl(const char *path, const char *arg, ...);  /* End the argument list with NULL */
int execlp(const char *file, const char *arg, ...);  /* End the argument list with NULL */
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

---

# Prototypes of Standard IO Library Calls

```
char  *fgets(char *s, int size, FILE *stream); /* Returns NULL on EOF */
int    fputs(const char *s, FILE *stream);     /* Returns EOF when unsuccessful */
int    fflush(FILE *stream);                   /* Returns 0 on success, EOF on failure */
FILE  *fdopen(int fd, char *mode);             /* Returns NULL on failure */

int fgetc(FILE *stream);                       /* Returns EOF on EOF */
int fputc(int c, FILE *stream);                /* Returns EOF when unsuccessful */

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);

int atoi(char *s);   /* Converts a string to an integer - returns zero if unsuccessful */
```

---

# Prototypes from Pthreads

```
typedef void *(*pthread_proc)(void *);
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   pthread_proc start_routine, void *arg);

int       pthread_join(pthread_t thread, void **value_ptr);
void      pthread_exit(void *value_ptr);
int       pthread_detach(pthread_t thread);
pthread_t pthread_self();

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

---

# Prototypes from sockettome.h

```
extern int serve_socket(int port);
extern int accept_connection(int s);
extern int request_connection(char *hn, int port);
```