CS360 Final Exam
May 4, 2017
James S. Plank

Instructions

- There are six questions. You must answer all six questions.
- Put your answers on separate sheets of paper. Do not hand in the exam.
- Put your name and utk email *(xxx@vols.utk.edu)* on all of your answer sheets.
- I have put my time estimates on how long you should take on each question. Use them to gauge your timing.

Things to make your life easier

- I have all sorts of function prototypes on the last page of this exam.
- Do not bother writing down any **#include** statements.
- You do not need to error check any system or library calls with the exception of **open()**, **fopen()** and the **exec** calls.
- You may abbreviate **pthread_mutex_t** as **PMT**.
- You may abbreviate **pthread_cond_t** as **PCT**.
- You may assume that the following two procedures exist -- these will save you some time:

```
PMT *new_mutex()
{
  PMT *m;
  m = (PMT *) malloc(sizeof(PMT));
  pthread_mutex_init(m, NULL);
  return m;
}
```

```
PCT *new_cond()
{
  PCT *c;
  c = (PCT *) malloc(sizeof(PCT));
  pthread_cond_init(c, NULL);
  return c;
}
```

CS360 Final Exam - May 4, 2017 - James S. Plank

Question 1 - 20 points (15 minutes)

Write the **jassem** assembly code for the C procedure to the right. As always, don't optimize your code.

```
int g(int j, int k)
{
  int i;

  for (i = j; i > k; i = g(i, j)) j += 5;
  return j;
}
```

Question 2 - 20 points (30 minutes)

Let us recall the **malloc()** lab. In this lab, you wrote **mymalloc.c**, which implemented the following five procedures:

- **void *my_malloc(size_t size)**: This returns a buffer of at least **size** bytes to the user. The number of bytes is a multiple of 8, and the pointer is aligned on an 8-byte quantity. 8 bytes of bookkeeping are added to the front of this buffer. The first four of these contain the size of the memory block (bookkeeping, padding and all). **my_malloc()** maintains a free list of memory blocks, and if it can satisfy **my_malloc()** from the free list, it does so.
- **void my_free(void *p)**: This assumes that **p** was returned to the caller from a previous call to **my_malloc()**. It returns the memory block to the free list.
- **void *free_list_begin()**: This returns a pointer to the first byte of the first free memory block on the free list. If there are no blocks on the free list, then **free_list_begin()** returns NULL.

```
int main()
{
  unsigned int *x, *fp[20];
  void *z;
  int i;
  /* More variable
    declarations are here */

  x = (unsigned int *) my_malloc(13);
  my_free(my_malloc(73));

  i = 0;
  z = free_list_begin();
  while (z != NULL)
    if (i == 20) {
      fprintf(stderr, "ERROR.\n");
      exit(1); }
    }
    fp[i] = (unsigned int *) z;
    i++;
    z = free_list_next(z);
  }
  /* More code is here */
}
```

- **void *free_list_next(void *p)**: This assumes that **p** is a pointer to the first byte of a free memory block. It returns a pointer to the first byte of the next free memory block on the free list. If **p** points to the last free memory block, then **free_list_next(p)** returns NULL.
- **void coalesce_free_list():** Ignore this.

Now, suppose I am writing one of the gradescript programs for **mymalloc**. I have the beginnings of such a program above to the right. It makes two **my_malloc()** calls, one **my_free()** call, one **free_list_begin()** call, and potentially multiple **free_list_end()** calls.

Your job is to tell me how you would go about completing this program to check for errors in **mymalloc.c**. Don't write code -- tell me in nice, precise English, what errors you are looking for, and how you would check for them. If you know of numbers to use, then use them -- don't give me, for example, vague things like "check to see that **b** is legal" when you know that **b** should be an exact value, like 40. In that case, you would say "check to see that **b** equals 40." (Note that this is just an example -- I know that there is no variable **b** in the code above).

You are not allowed to make any more calls to the procedures in **mymalloc.c**.

Don't bother checking to see if **x**, or any of the **z** are going to seg fault. Assume that pointers are 4 bytes.

CS360 Final Exam - May 4, 2017 - James S. Plank

Question 3 - 20 points (22 minutes)

Write a program that does what the shell does when it executes the following command:

```
UNIX> a.out 500 < input.txt | sed -e s/X/Y/ > output.txt
```

(You should use **execlp()** rather than **execvp()**, to make your life easier).

Question 4 - 20 points (28 minutes)

Write a program that takes a number *n* on the command line, and creates *n* threads, with ID's 0 through *n-1*. Each thread needs to go into an infinite loop, where at each iteration of the loop, it calls:

- **void perform_calculation(int id)**. Each thread can perform this at the same time. During this procedure, each thread communicates with a server to get the information for its calculation, and as a result, updates some global variables that pertain only to that thread.
- **void update_state(int id)**. This is where the thread-specific state is merged with some global state. This procedure should not be called simultaneously by multiple threads. Moreover, thread 0 should call this before thread 1, and thread 1 should call it before thread 2, etc. Thread 0 should not call this for the second time before thread *n-1* has called it for the first time, etc.

You need to write the **main()** that initializes information and creates threads, and the procedure that the threads run. I make two **typedef**'s below. You should use these in your procedures in such a way that **perform_calculation()** and **update_state()** are synchronized properly among the threads. You are not allowed to use **sleep()** calls or busy waiting.

You should assume that **perform_calculation()** and **update_state()** have been written by someone else -- you just call them and assume that they work.

Here are your **typedefs** (remember, you can abbreviate **pthread_mutex_t** as **PMT** and **pthread_cond_t** as **PCT**, and use **new_mutex()** and **new_cond()** as specified on the cover page of this exam.).

```
typedef struct {
  PMT *m;
  PCT **c;
  int n;                /* Number of threads */
  int turn;             /* Whose turn is next */
} Shared;

typedef struct {
  int id;              /* Thread id, from 0 to (n-1) */
  Shared *s;
} Info;
```

CS360 Final Exam - May 4, 2017 - James S. Plank

Question 5 - 18 points (17 minutes)

You are on a job interview, and the interviewer asks you the following questions. Please answer them in the best possible way, so that you can get the job!

- "Explain to me what SIGPIPE is, and how it gets generated."
- "How do you program so that your program recognizes when SIGPIPE gets generated, and then takes the appropriate actions?"
- "Explain how to use the C stdio library to help you with SIGPIPE."

Question 6 - 2 points (1 minute, 13 seconds)

What is the output of the following program?

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

int main()
{
  unsigned int i, j;

  i = 0x64827c54;
  j = (i << 12);
  printf("0x%08x\n", j);
  exit(0);
}
```

Prototypes of various useful system and library calls

```
int fork();
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);
```