# CS360 Final Exam - April 30, 2014 - James S. Plank

## Question 1 - 16 points

Use the "Answer Sheet for Question 1" for this one.

Draw the state of memory addresses 0xb8000 through 0xb80fc under the following conditions:

- Pointers are four bytes.
- **malloc()** and **free()** work as described in lecture notes.
- The user has called **malloc()** and **free()** a bunch. Currently, the user holds six pointers that were return values of **malloc()**, and he/she has not called **free()** on them. They are as follows:
  1. **malloc(5)** returned 0xb80f8.
  2. **malloc(12)** which returned 0xb8008.
  3. **malloc(8)** which returned 0xb8028.
  4. **malloc(8)** which returned 0xb8050.
  5. **malloc(16)** which returned 0xb8078.
  6. **malloc(20)** which returned 0xb80b8.
- The free list has a minimum number of nodes, and the addresses of the nodes on the free list are in ascending order.
- The free list is doubly linked and NULL terminated.

If you know a value of memory, then fill it in. If you don't know the value, leave it blank.

## Question 2 - 12 points

Explain to me what **setjmp()** and **longjmp()** do. Illustrate your explanation with a small program where the program calls **setjmp()** and **longjmp()**, and you explain its output. Don't be fancy with this -- have it all occur in **main**().

The prototypes are:

```
int  setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

## Question 3 - 16 points

You are writing the code for a matchmaking service called "2014mingle.com." You don't wish to be politically incorrect, so rather than match people by gender, you're going to match them by keyword. You've written your IOS and Android apps, which clients will use, and you've filmed your commercial -- "2014 Mingle. Can you hear your heartstrings tingle?"

Time to write the server. The server is going to serve a socket and fork off a thread for every client. Each thread is going to do some initialization, and then call the procedure **find_match()** with the following prototype:

```
void find_match(Person *me, char *keyword);
```

A **Person** struct has all of the contact information for a person, plus:

- A pointer to a condition variable in the field **cond**. This condition variable is unique to the person.
- A field called **match**, which is of type **Person ***. This is initialized to be NULL.

When **find_match()** is called, the variable **me** is pointing to the contact information for the client that is calling it. You need to find a match for this client. To help you, you get to use two global variables: A JRB named **tree**, and a pointer to a mutex called **lock**. You may assume that they have both been initialized correctly.

When **find_match()** is called, you are going to search the tree for a match. If you find one, you'll need to set the **match** field for both clients and have them return. If you don't find a match, you should insert the client into the tree and have him/her wait for a match.

**Part A:** Go ahead and write **find_match()**.

**Part B:** Is it possible in your code for the following to occur:

- Clients A and B match on a keyword.
- Clients C and D subsequently match on the same keyword.
- Clients D returns from **find_match()** before client A does.

If so, tell me how that happens, in terms of mutexes and condition variables and the order in which operations occur. If not, convince that it cannot.

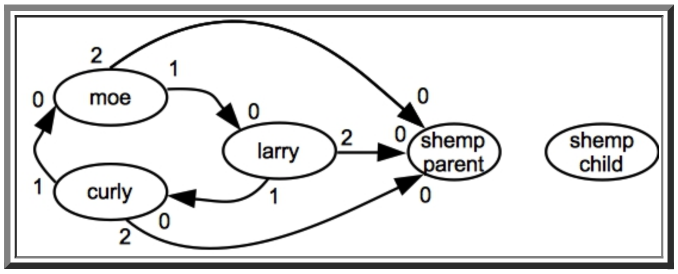
## Question 4 - 5 points

On at least one student's **chat_server** lab, he would pass each gradescript individually, but his server seg faulted in **gradeall**, around case 50. The reason is that he forgot to call **pthread_detach()**. Explain why the seg fault occurred, and why calling **pthread_detach()** fixed the problem.

## Question 5 - 24 points

You are taking CS402, and your design team consists of you, Larry, Moe and Curly. Professor Birdwell has made you all sign non-disclosure agreements, but for the life of you, you don't know why Larry, Moe or Curly would want to disclose their disastrous programming skills to anyone! Whatever, you have taken the design lead, and you have decided that the four of you will each write your own C programs. Moe's will be compiled into the executable **moe**; Larry's will be compiled into the executable **larry**, and Curly's will be compiled into the executable **curly**. None of the executables will have command line arguments, and none of them will be multi-threaded.

Your program (named **shemp.c**, of course), is going to be the master program. It is going to create processes so that when the system runs, it is going to look as follows:



As you can see:

- The shemp program will fork a child. We'll call the two processes, "shemp parent" and "shemp child," but they are going to share code.
- Moe's standard output goes to Larry's standard input.
- Larry's standard output goes to Curly's standard input.
- Curly's standard output goes to Moe's standard input.
- Moe, Larry and Curly's standard error will all go standard input of "shemp parent." Shemp parent is going to read its standard input, process it a bit and write to standard output.
- Moe, Larry and Curly will all be children of "shemp child."

You may wonder why we're splitting **shemp** into a parent and a child? The reason is as follows. Larry, Moe and Curly can't be trusted. Their processes are supposed to talk to each other forever, but in reality, they may seg fault or go into infinite loops. The child is there to detect when any of them die. When that happens, it is going to kill the others and then exit. When the others are dead, the parent will be able to detect it. It will then print "NFS not responding, still trying" and go into an infinite loop. That way, when you demo your project, professor Birdwell will hopefully be suckered into thinking that we have network problems, and he won't realize that disaster has occurred.

Fun as it would be, I'm not making you write **shemp.c**. However, I'm sure you would write it flawlessly, using only the system calls **fork()**, **execlp()**, **close()**, **wait()**, **pipe()**, **dup2()** and **kill()**.

Now, answer the following questions:

- **Question 5A**: How many times is "shemp-parent" going to call **fork**()? Why?
- **Question 5B**: How many times is "shemp-child" going to call **fork**()? Why?
- **Question 5C**: How many times is "shemp-parent" going to call one of the **exec**()'s? Why?
- **Question 5D**: How many times is "shemp-child" going to call one of the **exec**()'s? Why?
- **Question 5E**: "Shemp-Child" is going to call **pipe()** three times. How many times is "shemp-parent" going to call **pipe**()? Why?
- **Question 5F**: How many times is "shemp-parent" going to call **close()**? For which file descriptors?
- **Question 5G**: How many times is "shemp-child" going to call **close**()? For which file descriptors?
- **Question 5H**: How many times is the **larry** process going to call **dup2**()? Why?
- **Question 5I**: During the demo, disaster indeed occurs. **Larry** goes into an infinite loop and stops reading and writing. **Curly** doesn't bother reading from standard input, and instead goes into an infinite loop writing to standard output. And **moe** seg faults. How does "shemp child" detect that **moe** has died on a seg fault?
- **Question 5J**: After detecting that **moe** has died, "shemp child" kills **larry** and tries to kill **curly**, but **curly** is already dead. Why?
- **Question 5K**: How does "shemp-parent" detect that **larry**, **moe** and **curly** are dead?
- **Question 5L**: When "shemp-parent" goes into its infinite loop, you want to make sure that there are no zombie processes. How many times must it call **wait()** to do that? Why? (In your explanation, explain why **larry**, **moe** and **curly** do not end up as zombies).

## 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 JRB.h

```
extern JRB make_jrb();   /* Creates a new rb-tree */
extern JRB jrb_insert_str(JRB tree, char *key, Jval val);
extern JRB jrb_find_str(JRB root, char *key);  /* Returns NULL if it's not there */
extern void jrb_delete_node(JRB node);  /* Deletes and frees a node (but not the key or val) */
```