Spring 2014: CS110 Final Examination

This is a closed book, closed note, closed computer exam (although you are allowed to use your two double-sided cheat sheets, of course.) You have 180 minutes to complete all problems. You don't need to **`#include`** any libraries, and you needn't guard against any errors unless specifically instructed to do so. Understand that the majority of points are awarded for concepts taught in CS110. If you're taking the exam remotely, you can call me at 415-205-2242.

Good luck!

SUNet ID (username): _____**`@stanford.edu`**

Last Name: _____

First Name: _____

I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

[signed] _____

		Score	Grader
1. Piped Processes	[8]	_____	_____
2. Multiprocessing Redux	[12]	_____	_____
3. Priority Locking	[12]	_____	_____
4. Concurrency and Networking Redux	[18]	_____	_____
Total	**[50]**	_____	_____

Relevant Prototypes

```
// 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 retval
int pipe(int fd[]); // array should be of length 2, ignore retval
int dup2(int old, int new); // ignore retval
int execvp(const char *path, char *argv[]); // ignore retval
#define WIFEXITED(status)   // macro
#define WEXITSTATUS(status) // macro

// thread
class thread {
public:
   thread(...); // first argument is thread routine, its args come afterwards
   void join();
};

class mutex {
public:
   mutex();
   void lock();
   void unlock();
};

class semaphore {
public:
   semaphore(int count = 0);
   void wait();
   void signal();
   void signal(on_thread_exit_t); // pass on_thread_exit constant
};

class condition_variable_any {
public:
   template <typename Mutex, typename Pred> void wait(Mutex& m, Pred pred);
   void notify_one();
   void notify_all();
};
```

Problem 1: Piped Processes [8 points]

For this problem, you're to implement the **`execute`** function, which takes **two** argument vectors (e.g. full commands that have already been parsed and structured to be compatible with a call to **`execvp`**) and executes both of them—each in its own child process, with the added feature that the standard output of the first process is directly piped to the standard input of the second. **`execute`** should also return **`true`** if and only if both processes exited normally with status code of 0.

So, given a working implementation of execute, the following program should output the words within **`words.txt`** in sorted order:

```
int main(int argc, char *argv[]) {
  char *first[] = {"cat", "words.txt", NULL};
  char *second[] = {"sort", NULL};
  bool success = execute(first, second);
  printf("Things went well? %s\n", success ? "yes" : "no");
  return 0;
}
```

If **`words.txt`** contains the words **`to be or not to be`** (one word per line, in that order) then the output of the above program would be:

```
be
be
not
or
to
to
Things went well? yes
```

Implementation details:

- You must use **`fork`** and **`execvp`** to spawn the two child processes. More specifically, you may not use **`system`**, **`popen`**, **`subprocess`**, or any other Linux function that layers on top of **`fork`** and **`execvp`**.
- You must use the **`pipe`** function to create linked file descriptors to enable the inter-processes communication. (In essence, you're trying to implement command line pipes, as with **`cat words.txt | sort`**. This could easily have been required of the Assignment 3: tiny shell specification.) Recall that after a call to **`pipe(fds)`**, what's written to **`fds[1]`** is readable from **`fds[0]`**.
- **`execute`** must wait for both child processes to finish, reap their resources, and then return **`true`** if and only if both processes exited normally with exit status 0.
- You must close all unused file descriptors, so that the execution of any program calling execute can still generate a clean **`valgrind`** report.
- You can omit all of the error checking that would normally be required of a robust implementation.

Use the next page to present your implementation.

Problem 1: Piped Processes [continued]

```
// convenience function for closing both ends of a pipe
static void closeBoth(int fds[]) {
  close(fds[0]);
  close(fds[1]);
}

static bool execute(char *first[], char *second[]) {
```

Problem 2: Multiprocessing Redux [12 points]

Your answers to the following questions should be 50 words or less (about the length of this paragraph.) Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided it's clear what you're saying. Full credit will be given to clear, correct, relevant responses.

a. [3 points] The implementation of **`tsh`**—your tiny shell assignment—relied on a **`SIGCHLD`** handler to update the job list whenever a job terminated, stopped, or resumed. The installed **`SIGTSTP`** handler, on the other hand, simply intercepted and forwarded **`SIGTSTP`** on to the foreground job. Why not update the job list to reflect the job state change inside the implementation of the **`SIGTSTP`** handler?

b. [3 points] Signals set to be caught by custom signal handlers will have default signal handling behavior in the child process after **`execvp`** is called. Why can't the child process inherit the parent's installed signal handlers instead?

c. [3 points] Your shell can be configured so that a process **dumps core**—that is, generates a data file named **`core`**—whenever it crashes (via **`SIGSEGV`**, for instance.) This **`core`** file can be loaded into and analyzed within **`gdb`** to help identify where and why the program is crashing. Assuming we can modify the program source code and recompile, how might you **programmatically** cause a **`SIGSEGV`** and thus generate a core dump at a specific point in the program while allowing the process to continue executing? (Your answer might include a very short code snippet to make its point.)

d. [3 points] The **`fork`** system call creates a new process with an independent virtual address space, where all memory segments of the parent process are copied. If, however, a **copy-on-write** implementation strategy is adopted, then the physical memory backing the new virtual address space needn't actually be copied when the parent process is forked. Virtual memory pages in both parent and child can map to the same physical memory pages until one of them writes through to a page, and only then is the virtual memory page in the child's address space mapped to a different physical page. What common use case of **`fork`** from class and from the assignments would support a copy-on-write implementation strategy?

Problem 3: Priority Locking [12 points]

When multiple threads try to acquire the lock on a traditional **mutex**, only one succeeds, and all others block until the lock on the **mutex** is released. When the lock is released, any one of the blocked threads may be chosen to acquire the lock next, because as far as the **mutex** is concerned, all threads are equally worthy.

The **p_mutex** class operates much like a regular **mutex**, except that the lock on a **p_mutex** may be acquired with or without **privilege**. If a **p_mutex** is unlocked, then a thread can acquire the lock regardless of privilege. If a **p_mutex** is locked, then all other threads vying for the lock are blocked until the lock is released. When the lock is finally released, privileged threads are chosen from the blocked set before unprivileged ones are.

Here is the full interface for the **p_mutex** class:

```
class p_mutex {
 public:
  p_mutex();
  void lock(bool privileged = false);
  void unlock();

 private:
  mutex m;
  condition_variable_any cv;
  enum {
    unlocked, locked, locked_with_privilege
  } state;
  int num_privileged_waiting;
};
```

Note that the interface file is complete, and the **private** section of the class definition provides all of the data members needed to fully implement the constructor and the two methods. To be clear:

- The constructor constructs a **p_mutex** to be unlocked, with no waiting threads—privileged or otherwise.
- The **lock** method attempts to acquire the lock on the **p_mutex**, and succeeds if no other threads are waiting. Otherwise, it blocks, joining the set of other blocked threads. If **lock** is called without arguments (or **false** is supplied), then it is considered an unprivileged thread when it blocks, and won't get the lock until all blocked, privileged threads have acquired and released the lock. If **lock** is called with **true** and it blocks, then it is placed among the set of privileged threads to be considered next when the lock is released.
- **unlock** marks the **p_mutex** as unlocked, allowing some privileged thread (or if there aren't any, some unprivileged thread) to acquire the lock next.
- The implementation **must** be framed in terms of the **private** data members I've already listed, and you may not add any others or introduce any global variables.

Present your implementation of the constructor and the two **`public`** methods. You need not worry about the destructor, and you needn't do any error checking or otherwise confirm the class is being used properly (e.g. don't worry about the same thread calling **`lock`** twice in a row before unlocking, don't worry about **`unlock`** being called without a prior call to **`lock`**, etc.)

a) [1 point] Implement the constructor, which ensures that the **`p_mutex`** is configured to be unlocked (just as a regular **`mutex`** would be configured to be unlocked by its own constructor.)

b) [7 points] Present your implementation of the **`lock`** method.

c) [4 points] Present your implementation of the **`unlock`** method.

Problem 4: Concurrency and Networking Redux [18 points]

Your answers to each of the following questions should be 50 words or less. Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided it's clear what you're saying. Full credit will be given to clear, correct, on-point responses.

a. [3 points] My threading and concurrency lectures made a point to avoid busy waiting. Explain why busy waiting is generally a bad idea.

b. [3 points] There are many reasons a thread or process might be moved from a running state to a blocked state. List three of them.

c. [4 points] Explain why the multithreaded version of your RSS News Feed Aggregator is so much faster than the sequential version, and describe a type of application and computer hardware configuration where the introduction of threading would actually hurt performance.

d. [4 points] We interact with socket descriptors more or less the same way we interact with traditional file descriptors. Identify one thing you can't do with socket descriptors that you can do with traditional file descriptors, and briefly explain why not.

e. [4 points] In lecture, we presented three different siblings of the **sockaddr** record family

```
struct sockaddr {
  short sa_family;
  char sa_data[14];
};
```

```
struct sockaddr_in {
  short sin_family;
  short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
```

```
struct sockaddr_in6 {
  u_int16_t sin6_family;
  u_int16_t sin6_port;
  // other fields
};
```

The first one is a generic socket address structure, the second is specific to traditional IPv4 addresses (e.g. **171.64.64.131**), and the third is specific to IPv6 addresses (e.g. **4371:f0dd:1023:5::259**), which aren't in widespread use yet (at least not at Stanford). The addresses of socket address structures like those above are cast to (**struct sockaddr ***) when passed to all of the various socket-oriented system calls (e.g. **accept**, **connect**, and so forth). How can these system calls tell what the true socket address record type really is—after all, it needs to know how to populate it with data—if everything is expressed as a generic **struct sockaddr ***?