# Spring 2014: CS110 Final Examination Solution

## Solution 1: Piped Processes

```
static bool execute(char *first[], char *second[]) {
  int fds[2];
  pipe(fds);
  pid_t pids[2];

  if ((pids[0] = fork()) == 0) {
    dup2(fds[1], STDOUT_FILENO); // descriptor 1 publishes to write end of pipe
    closeBoth(fds); // don't need read end or old descriptor to write end
    execvp(first[0], first); // gut process and inject new image
  }

  if ((pids[1] = fork()) == 0) {
    dup2(fds[0], STDIN_FILENO); // descriptor 0 pulls from read end of pipe
    closeBoth(fds); // don't need write end or old descriptor to read end
    execvp(second[0], second); // gut process and inject new image
  }

  closeBoth(fds); // close pipe at parent, only child processes needed it
  int statuses[2];

  for (size_t i = 0; i < 2; i++)
    waitpid(pids[i], &statuses[i], 0);

  for (size_t i = 0; i < 2; i++)
    if (!WIFEXITED(statuses[i]) || WEXITSTATUS(statuses[i]) != 0)
      return false;

  return true; // both succeeded
}
```

## Problem 1 Criteria: 8 points

- Properly declares an array of length 2 and populates it with two file descriptors via a call to **`pipe`**: 1 point
- Properly calls **`fork`** twice to create copies of the original into which **`first[0]`** and **`second[0]`** can be installed: 1 point
- Properly uses dup2 to remap the correct ends of the pipe to **`stdin`** and **`stdout`**: 1 point (don't worry about argument order—I always forget as well)
- Properly relies on **`execvp`** to cannibalize the process images to execute **`first[0]`** and **`second[0]`**: 1 point
- Closes both ends of the original pipe in the child processes: 1 point
- Closes both ends of the original pipe in the parent process: 1 point
- Properly waits for the two child processes to finish: 1 point
- Properly returns **`true`** if and only if both processes exit normally with status code of 0: 1 point

**Solution 2: Multiprocessing Redux**

a. The implementation of **`stsh`** 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?

   The foreground process may install a custom **`SIGTSTP`** handler and execute that without suspending, so a **`SIGCHLD`** is never (and shouldn't be) issued. (**Criteria**: 3 points for correct answer, 1 point for something relevant but oblique/obscure, 0 points otherwise.)

b. 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?

   Because the machine code for the signal handlers exists in the parent's text segment, not in the child's. Even if the text image of the child could be updated, you don't want to support the injection of arbitrary code into another application—infinite security risk. (Image a shell that launches make with a **`SIGTSTP`** handler that, oh, executes **`rm -fr ~/*`**. (**Criteria**: 3 points for correct answer, 1 point for something relevant but oblique/obscure, 0 points otherwise.)

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.)

   Add the following line of code (or something equivalent) at the exact point where you want a core dump to be generated:

```
if (fork() == 0) { int *p = NULL; *p = 0; }
```

   (**Criteria**: 3 points for correct answer, 1 point for something way more complicated than necessary, 0 points otherwise.)

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 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?

> In practice, when **`fork`** is called to create a child process, **`execvp`** is called very, very soon. We shouldn't make a deep copy of mapped virtual address space when the virtual address space is within milliseconds of being cannibalized and repurposed with a new process image. (**Criteria**: 3 points for correct answer, 1 point for something correct but obscure, 0 points otherwise.)

**Solution 3: Priority Locking**

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.)

```
p_mutex::p_mutex(): state(unlocked), num_privileged_waiting(0) {}
```

**Problem 3a Criteria: 1 point**

- Properly initializes primitive data members: 1 point

b) Present your implementation of the **`lock`** method. You may use the top portion of the next page as well.

```
void p_mutex::lock(bool privileged) {
  lock_guard<mutex> lg(m);
  if (privileged) {
    num_privileged_waiting++;
    cv.wait(m, [this]{ return state == unlocked; });
    state = locked_with_privilege;
  } else {
    cv.wait(m, [this] {
      return state == unlocked && num_privileged_waiting == 0;
    });
    state = locked;
  }
}
```

or

```
void p_mutex::lock(bool privileged) {
  lock_guard<mutex> lg(m);
  if (privileged) {
    num_privileged_waiting++;
    cv.wait(m, [this]{ return state == unlocked; });
    num_privileged_waiting--;
  } else {
    cv.wait(m, [this]{
      return state == unlocked && num_privileged_waiting == 0;
    });
  }
  state = locked;
}
```

**Problem 3b Criteria: 7 points**

- Properly acquires **`m`** at beginning: 1 point
- Properly releases **`m`** at end: 1 point
- Properly dispatches between two scenarios, if needed: 1 point
- Properly waits for correct condition to be satisfied: 2 points (1 point for each scenario)
- Properly tracks number of privileged threads: 1 point
- Properly updates state: 1 point

c) And finally, present your implementation of the **`unlock`** method.

```
void p_mutex::unlock() {
  lock_guard<mutex> lg(m);
  if (state == locked_with_privilege) num_privileged_waiting--;
  state = unlocked;
  cv.notify_all();
}
```

or (alternatively, if second **`lock`** implementation above is used)

```
void p_mutex::unlock() {
  lock_guard<mutex> lg(m);
  state = unlocked;
  cv.notify_all();
}
```

**Problem 3c Criteria: 4 points**

- Properly acquires and releases **`m`**: 1 point
- Properly updates state: 1 point
- Properly demotes count on number of privileged threads: 1 point (note that it's possible they did this in lock): 1 point
- Properly calls **`notify_all`** (or, if they know only unprivileged threads are waiting, **`notify_one`**): 1 point

**Solution 4: Concurrency and Networking Redux**

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

   Busy waiting allows a thread to occupy the CPU when it has nothing meaningful to do. It's better to block thread off the processor indefinitely until the thread manager is notified that the thread actually has a chance to do meaningful work. (**Criteria**: 3 points for correct answer, 1 point for obscure answer or one with correct and incorrect information, 0 for incorrect response.)

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

- Thread is blocked on a locked **mutex**, a condition that's failing, or a **semaphore** value of 0.
- Thread or process is blocked because it's in a slow I/O system call (**read**, **accept**, etc.)
- Thread or process is blocked because it's waiting for a timer interrupt (as a result of a call to **sleep** or **sleep_for**)

(**Criteria**: 1 point for each distinct reason provided. Only read first three, and don't give two points for same category, e.g. **mutex** and **semaphore** listed separately.)

c. 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.

It's faster because you overlay all of the dead times associated with the acquisition of multiple network connections. An application that's mostly or all CPU-bound (e.g. a local file tree crawl) that runs on a single-processor machine would not benefit from threading, since there same number of machine instructions need to be executed whether it's within one thread or distributed across many. (**Criteria**: Solid answer to first question is worth 1 point, answer to second question is worth 2 points—all other nothing, based on quality.)

d. 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.

Socket descriptors are not seek-able (or informally, we can't skip over bytes or rewind). This is because the bytes accessible through a socket aren't permanently stored anywhere as they are with files. (**Criteria**: 3 points for solid answer [might be different than mine], 1 point for obscure answer, 0 points for one that's incorrect.)

e. 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 *`**?

> The first field—the two-byte **`short`** under a variety of names—is examined with the expectation that it identifies the larger type around it.
>
> (**Criteria**: Give 3 points for correct answer, 1 point if they think the length-oriented argument identifies the type, and 0 points otherwise.)