CS110 Final Examination Solution: Winter 2015

Solution 1: Filesystems and Process Control Redux [21 points]

Each of the following questions is accompanied by two to five choices, only one of which correctly answers it. Clearly circle the letter (a, b, c, d, etc.) of that correct answer.

- [3 points] In the following code snippet, the parent process opens a file twice, and then a forked child reads a single character.

```
int fd1 = open("foo.txt", O_RDONLY);
int fd2 = open("foo.txt", O_RDONLY);
if (fork() == 0) {
   char ch;
   read(fd1, &ch, 1);
}
bar(fd1, fd2);
```

  Which of the following statements is true of the parent process just as it's calling **`bar`**?

  a. **`fd1`** points to the first character while **`fd2`** points to the second character of **`foo.txt`**.
  [✓ b.] **`fd1`** points to the second character while **`fd2`** points to the first character of **`foo.txt`**.
  or
  [✓ c.] **`fd1`** and **`fd2`** each point to the first character of **`foo.txt`**.
  d. **`fd1`** and **`fd2`** each point to the second character of **`foo.txt`**.

- [3 points] **`stat`** and **`lstat`** surface different information for what type of file?

  a. regular files
  b. directories
  [✓ c.] symbolic links
  d. none of the above

- [3 points] Assume a process blocks both **`SIGCHLD`** and **`SIGTSTP`** signals. While blocked, the kernel sends it a **`SIGCHLD`**, a **`SIGTSTP`**, and a second **`SIGCHLD`** in that order. What signals does the executable receive after it unblocks the two signals?

  a. none, since the signals were blocked at the time they were sent, they were discarded.
  b. just a single **`SIGCHLD`**, since all subsequent signals are discarded.
  [✓ c.] a single **`SIGCHLD`** and a single **`SIGTSTP`**, since the second **`SIGCHLD`** is discarded.
  d. all three signals, since no signals are discarded.

- [3 points] Which of the following events does not generate a signal?

    a. integer division by zero.
    b. a **NULL** pointer dereference.
    c. a child process terminates after its original parent process terminates.
    d. **read** from a socket where the write end has been closed.
    [✓ e.] **write** to a socket where the read end has been closed.

- [3 points] Assume that an **int a** is stored at virtual (64-bit, hexadecimal) address of **0x7fff5fbffa40** and that an **int b** is stored at virtual (64-bit) address of **0x7fff5fbffa80**. If the size of a page is **0x1000** (hexadecimal) bytes, which of the following is true?

    [✓ a.] **a**'s physical address is numerically lower than **b**'s physical address
    b. **a**'s physical address is numerically higher than **b**'s physical address
    c. **a**'s physical address is might be higher than than **b**'s physical address, but not necessarily

- [3 points] Assuming that a page size is 4000 (decimal) bytes and that integers are 4 bytes, which of the following statements is true?

    a. all **int**s of an array of 100 integers are guaranteed to reside in the same page of physical memory
    [✓ b.] the **int**s of an array of length 1002 could collectively reside in three different pages of physical memory
    c. the first **int** of an array is guaranteed to be at a lower physical address that the second int of the same array
    d. none of the above statements are true

- [3 points] Which of the following (presumably valid) system calls is guaranteed to move a process into the blocked state?

    a. **read**
    b. **accept**
    c. **waitpid**
    d. all of the above
    [✓ e.] none of the above

Solution 2: Downloading Music [14 points]

You have access to the following thread-safe function that downloads a single song from wherever it is that only music retailers house all of their music:

```
bool download(const string& title);
```

`download` returns **`true`** if and only if the song was downloaded successfully, and **`false`** if the download failed.

Write a function called **`downloadAll`** that, given an arbitrarily long **`vector`** of song titles, downloads all of the songs. Rather than downloading the songs sequentially, enlist the services of a single **`ThreadPool`** of size 4 to download songs in parallel. If after the **`ThreadPool`** drains you detect one or more songs failed to download, wait two seconds and attempt to download just those songs a second time. If after a second round some still fail to download, try a third time. If after three times some songs haven't been downloaded, give up.

Your implementation of **`downloadAll`** should not rely on any global variables, and it should return the number of songs that were eventually downloaded. Use this and the next page to present your answer.

```
static int downloadAll(const vector<string>& titles) {
   ThreadPool pool(4);
   vector<string> missing = titles;
   vector<string> failed;
   mutex m;
   for (size_t i = 0; i < 3; i++) {
      for (size_t i = 0; i < missing.size(); i++) {
         pool.schedule([&, i] {
            if (download(missing[i])) return;
            lock_guard<mutex> lg(m);
            failed.push_back(missing[i]);
         });
      }
      pool.wait();
      missing = failed;
      if (missing.empty()) break;
      sleep(2);
   }

   return titles.size() - missing.size();
}
```

Problem 2 Criteria: 14 points

- Properly tracks those songs that have yet to be download: 3 points
- Properly guards against any race conditions while compiling list of missing songs: 3 points
- Properly waits for all threads each round to finish: 2 points
- Properly captures all variables in surrounding scope as necessary: 3 points
- Properly implements three rounds (short-circuiting isn't technically necessary): 2 points
- Properly returns the number of songs that were downloaded: 1 point

Solution 3: Concurrency and Networking Redux

Your answers to 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, complete, relevant responses.

a. [3 points] Your first implementation of **`news-aggregator`** managed the following:

- spawned a separate thread for each RSS feed
- within each RSS feed thread, spawned a separate thread for each article
- required that each RSS feed thread join against all article threads it spawned
- limited the number of active connections to any one server (e.g. **`www.nytimes.com`**) to be at most 6 at any one time

This last requirement was imposed just for article downloads, but not for feed downloads. Describe a scenario where **`news-aggregator`** would deadlock if the per-server limit of 6 were imposed for both RSS feed and article downloads combined.

If there are six or more feeds being drawn from the same server (say, www.nytimes.com) and all are waiting for their www.nytimes.com articles (that aren't permitted to download) to download, then you'll get deadlock.

b. [3 points] Your second version of **`news-aggregator`** essentially relied on the following code snippet, which you can assume works to specification. (Assume **`feedsPool`** and **`articlesPool`** are each **`ThreadPool`**s of size 10 and 64, respectively).

```
RSSFeedList feedList(feedListURL);
feedList.parse();
for (const pair<string, string>& p: feedList.getFeeds()) {
   feedsPool.schedule([p] {
      RSSFeed feed(p.first);
      feed.parse();
      for (const Article& article: feed.getArticles()) {
         articlesPool.schedule([article] {
            addToIndex(article); // assume addToIndex does the right thing
         });
      }
   });
}

feedsPool.wait();
articlesPool.wait();
queryIndex(); // assumes entire index has been built and is read-only
```

Identify one major problem that might be introduced if the two calls to **`wait`** are exchanged?

```
articlesPool.wait();
feedsPool.wait();
queryIndex();
```

*It's possible for all of the feedlist thunks to be scheduled, but for those thunks to make no progress (so no article thunks get scheduled) as the main thread advances to the **`articlesPool.wait`** call. Execution could pass right through both **`wait`** calls into **`queryIndex`**, which assumes the index is read-only.*

c. [3 points] The **`condition_variable_any`** class requires that a locked **`mutex`** be passed to its **`wait`** method. What exactly is that locked **`mutex`** protecting?

It's guarding the execution of the condition.

d. [3 points] Someone designing the **`condition_variable_any`** class could argue that the **`mutex`** should be supplied in the unlocked state, and that the implementation of **`wait`** should acquire a lock on the supplied **`mutex`**, wait until the condition is met, release the lock on the **`mutex`**, and return. Were this the case, then the implementation of **`semaphore::wait`** would look like this:

```
void semaphore::wait() {
   cv.wait(m, [this]() { return value > 0; });
   lock_guard<mutex> lg(m);
   value--;
}
```

In the context of the new implementation of **`semaphore::wait`** above, explain why this new behavior of **`condition_variable_any::wait`** is broken.

*If the lock is released before wait is called, two previously waiting threads might both return because another thread promoted **`value`** from 0 to 1, and both would be permitted to decrement value to 0 and then -1.*

e. [3 points] Your home desktop machine has three public IP addresses: 100.45.14.15, 100.45.14.16, and 100.45.14.17. You're implementing a single server running in a single process, but you'd like clients to be able to connect to your server through 100.45.14.15 (port 12345) and 100.45.14.16 (port 34567), but not through 100.45.14.17. Briefly describe how.

*Create two server sockets (not to **`IADDR_ANY`**, but to specific IP addresses and companion ports) and spawn two threads—one for each server socket—to accept connections and handle their own incoming requests (perhaps using a shared **`ThreadPool`**, but not required).*

f. [3 points] An HTTP server would like to limit the number of requests from any one IP address to 10 connections per second. Briefly explain how this could be implemented.

*Rely on the client IP address-surfacing parameters of **`accept`**, and maintain a **`map<string, int>`** of IP address/connection count pairs. If the connection count* *for an IP address ever hits 10 in a single second, just ignore it. Clear the map every second.*

g. [3 points] Your map-reduce system in Assignment 7 was designed to manage the group-by-key phase on the server. Real map-reduce systems use the map and reduce workers to collectively manage the hashing, collation, sorting, and grouping (i.e. the four phases of a real group-by-key effort) of intermediate files. Which phases of the group-by-key effort must be done (or are more easily done) by the mappers, which must be done (or are more easily done) by the reducers, and which ones can be really be done by either equally easily?

The hashing absolutely needs to be done by the mappers, since they are the components that produce the keys. The collation, sorting, and grouping should be done by the reducers, since they need to operate on sets of keys that all hash to the same value.

h. [3 points] Servers generally relegate the incoming request to either a separate thread or a separate process so that the primary thread of execution can more immediately loop around to accept new client connections. List two advantages of using threads instead of processes, and list one advantage of using processes instead of threads.

Advantage of threads over processes:

- *Easier to share resources between threads than processes*
- *Less expensive to create threads, more expensive to create processes*
- *(many other legitimate advantages)*

Advantage of processes over threads:

- *Private address spaces, much harder to leak information across process boundaries.*

i. [1 point] What was your favorite assignment of the seven?

The correct answer was Assignment 5. I hope that's what you put down. Oh, and supersmall typo: this was 1,000,001 points. Sorry about that.