Autumn 2014: CS110 Final Examination Solution

Solution 1: **SIGCALL** and System Call Traces [Median was 8 out of 10]

```
static pid_t pid = 0;
static void handleSIGCALL(int sig) {
  while (true) {
    const char *name = getsyscall(pid);
    if (name == NULL) return;
    printf("%s\n", name);
  }
}

int main(int argc, char *argv[]) {
  signal(SIGCALL, handleSIGCALL);
  pid = fork();
  if (pid == 0) {
    int devnull = open("/dev/null", O_WRONLY);
    dup2(devnull, STDOUT_FILENO);
    dup2(devnull, STDERR_FILENO);
    close(devnull);
    execvp(argv[1], argv + 1);
  }

  int status;
  waitpid(pid, &status, 0);
  return WEXITSTATUS(status);
}
```

Problem 1 Criteria: 10 points

- Calls **`fork`** and dispatches the child to execute the program to be traced: 1 point
- Opens **`"/dev/null"`**, and does so under the jurisdiction of the **`pid == 0`** test: 1 point
- Binds both **`STDOUT_FILENO`** and **`STDERR_FILENO`** to **`devnull`**'s resources: 1 point
- Closes **`devnull`** to release the file descriptor: 1 point
- Calls **`execvp`** in the correct place: 1 point
- Calls **`execvp`** with the correct arguments: 1 point
- Properly blocks on **`waitpid`** (in **`main`**, otherwise more complex than it needs to be: 1 point
- Returns the traced executable's return value: 1 point
- Includes a **`while (true)`** loop around repeated calls to **`getsyscall(pid)`**, and breaks on **`NULL`**: 1 point
- Prints out each system call name: 1 point

There were other isolated deductions imposed for errors we couldn't possibly have anticipated that just didn't map to the criteria.

Solution 2: Multiprocessing Redux [Median was 4 out of 12]

a. Consider the following program called **`conduit`** (this is the entire implementation):

```
int main(int argc, char *argv[]) {
  while (true) {
    sleep(1); // sleep one second
    int ch = fgetc(stdin); // pulls a single character from stdin
    if (ch == -1) return 0;
    putchar(ch); // presses the char ch to stdout
    fflush(stdout);
  }
}
```

When I type the following line in at the command prompt on a **`myth`** machine, I create a background job with five processes.

```
myth15> echo abcdefghij | conduit | conduit | conduit | conduit &
[1] 20686 20687 20688 20689 20690
```

- [2 points] How would each of the four **`conduit`** processes terminate if I send a **`SIGKILL`** (the kill-program signal) to pid 20687 five seconds after launching the job?

  The first **`conduit`** is forcibly terminated, and the three **`conduit`** processes after it all terminate gracefully (**`fgetc`** returns -1, since the write end of the pipes are gracefully closed.): 2 points for the right answer, 0 points for the wrong answer (don't require they mentioned that 20687 is killed, since that's obvious and the student might not see the point in writing it down.)

- [2 points] How would each of the four **`conduit`** processes terminate if I instead send a **`SIGKILL`** to pid 20689 five seconds after launching the job?

  The last **`conduit`** exits gracefully: 1 point
  The first two **`conduit`**s each abort because of uncaught **`SIGPIPE`** signals (the read end of their pipes have been closed, so attempts to write to **`stdout`** fail and prompt **`SIGPIPE`**s to be issued.): 1 point for this

b. [4 points] Typically, each page of a process's virtual address space maps to a page in physical memory that no other virtual address space maps to. However, when two processes are running the same executable (e.g. you have two instances of **`emacs`** running,) some pages within each of the two processes' virtual address spaces can map to the same exact pages in physical memory. Name two segments (the **heap** is an example of a segment) of a processes' virtual address spaces that might be backed by the same pages of physical memory, and briefly explain why it's possible.

Any read-only segment in a process's virtual address space can be aliases to the same physical address space, because nothing can be changed behind the back of the same segment in another virtual address space: 2 points for recognizing this.

Two examples: text (**`.text`**) segment, read-only global text (**`.rodata`**) segment: 1 point for each. (The shared library segment is also a legitimate answer.)

c. [4 points] Recall that the stack frames for system calls are laid out in a different segment of memory than the stack frames of normal (i.e. user program) functions. How are the stack frames for system calls set up, and how are the values passed to the system calls received when invoked from user functions?

Parameters are transferred from user stack to kernel stack through registers: 2 points for recognizing this

More detail: for x86-32, an integer identifying which system call is being invoked is placed in %eax, and additional arguments are placed in %ebx, %ecx, etc.

The stack frames are set up by a trap handler, which is executed when the calling function executes a trap machine instruction (after populating the registers with the system call id and the relevant arguments.): 2 points for recognizing that some sort of signal is used to execute the system call.

More detail: on x86-32, the trap machine instruction is int 0x80.

Solution 3: Concurrent and Evaluation [Median was 8 out of 10]

```
static const size_t kPoolSize = 64;
static ThreadPool pool(kPoolSize);
static bool concurrentAnd(const vector<BoolExpression>& expressions) {
  mutex m;
  condition_variable_any cv;
  size_t count(expressions.size());
  bool result = true;
  for (size_t i = 0; i < expressions.size(); i++) {
    pool.schedule([&, i]() {
        bool b = expressions[i].evaluate();
        lock_guard<mutex> lg(m);
        if (!b) result = false;
        count--;
        if (count == 0) cv.notify_one();
      });
  }

  lock_guard<mutex> lg(m);
  cv.wait(m, [&count]() { return count == 0; });
  return result;
}
```

Problem 3 Criteria: 10 points

- Declares a **bool** that's defaulted to true (or declares a **vector<bool>** with the expectation that all expression results are stored somewhere): 1 point
- Declares the **mutex** needed to protect against race conditions: 1 point
- Declares the **condition_variable_any** to manage communication that all expressions have been evaluated: 1 point
- Shares everything by reference or by value (as needed) with the thunks: 1 point (if they go with an index variable like **i**, that needs to be shared by value.)
- Calls the evaluate method without any kind of lockdown: 1 point
- Properly uses the **mutex** to guard whatever it needs to guard (at the very least, it needs to guard something like **count**, but in the case of a solution like mine, it needs to guard **result** as well): 1 point
- Properly constructs the condition that must be met before **concurrentAnd** can return: 1 point
- Properly acquires the lock on the condition—whatever it is—before calling **wait**: 1 point
- Properly unlocks all **mutex**es: 1 point
- Properly notifies the **condition_variable_any** when the condition is met: 1 point

There were other isolated deductions imposed for errors we couldn't possibly have anticipated that just didn't map to the criteria.

Solution 4: Concurrency and Networking Redux [Median was 9 out of 18]

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. There are a very limited number of scenarios where busy waiting **<u>is</u>** a reasonable approach to guarding a critical region. Briefly describe one such scenario in enough detail that someone just learning about threading and concurrency would understand why busy waiting might make more sense.

- System must be multi-core and/or multi-processor, so that the thread holding the resource could release the resource while the busy waiting thread is spinning.
- The second thread holding the resource is expected to hold it for a very, very short time (e.g. the resource should be freed up before the busy waiting thread's processor quantum is over).

Criteria: 2 points for saying one but not both, 3 points for saying both

b. Explain why a line as simple as **`i++`**, where **`i`** is a simple **`int`**, might be thread-safe on some architectures but not thread-safe on others, even if the implementer fails to use concurrency directives like the **`mutex`** or the **`semaphore`**.

On some architectures, **`i++`** compiles to a single assembly code instruction when **`i`** is a global variable (e.g. x86, location of global is defined prior to execution.) However, **`i++`** might compile to three or more assembly code instructions (e.g. **`i`** is a C++ reference parameter aliasing another integer whose location is not known at load time.) Criteria: 1 point for framing discussion in terms of atomicity, 3 points for providing clear examples as I do, 0 points otherwise.

c. Briefly explain the primary advantage of using a **`lock_guard<mutex>`** over exposed calls to **`mutex::lock`** and **`mutex::unlock`**.

The **`lock_guard`** constructor acquires a **`mutex`** lock, and the **`lock_guard`** destructor, which is called no matter how the surrounding scope ends, always calls **`unlock`**. The **`lock_guard`** is ideal when there are multiple exit paths (early returns, throw exceptions, etc) and the **`mutex`** needs to be unlocked regardless. Criteria: The above argument is the only one that really works for me, and that's worth two points. You can give 1 point if they argue it's less code (technically true), but that doesn't require CS110 knowledge.

d. When I updated **createClientSocket** for the **http-proxy** assignment, I replaced the call to **gethostbyname** with a call to **gethostbyname_r**, which has the following prototype:

```
struct hostent {
  char *h_name;       // real canonical host name
  char **h_aliases;   // NULL-terminated list of host name aliases
  int h_addrtype;     // result's address type, typically AF_INET
  int length;         // length of the addresses in bytes (typically 4, for IPv4)
  char **h_addr_list  // NULL-terminated list of host's IP addresses
};

int gethostbyname_r(const char *name, struct hostent *ret,
                    char *buf, size_t buflen,
                    struct hostent **result, int *h_errnop);
```

This second, reentrant version is thread-safe, because the client shares the location of a *locally* allocated **struct hostent** via argument 2 where the return value can be placed, thereby circumventing the caller's dependence on shared, statically allocated, global data. Note, however, that the client is expected to pass in a large character buffer (as with a locally declared **char buffer[1 << 16]**) and its size via arguments 3 and 4 (e.g. **buffer** and **sizeof(buffer)**). What specific purpose does this buffer serve?

Memory for the arrays that extend from the locally allocated **struct hostent** must also be placed somewhere, and it can't be globally allocated (else it's not reentrant) or dynamically allocated (since there's no expectation to **free** any memory). Criteria: 1 point for anything on par with some additional-temporary-space argument, and 3 points for understanding that the space is for the arrays extending from the locally allocated **struct hostent** must be placed somewhere as well.

e. With Assignment 7, a worker establishes a new network connection to the server for every single message it sends. Briefly describe how the implementation of your MapReduce worker and server would need to change if a single, open connection were maintained per worker for the lifetime of the map or reduce phases.

   The worker would open a single client connection to the server and leave it open until it's notified by the server that there aren't any more jobs. The server would need to maintain a pool of threads—one for each worker—to concurrently manage the back-and-forth messaging with each. Criteria: 1 point for the worker's single connection, 2 points for the server's **`ThreadPool`** of open client connection conversations, 3 points total.

f. Your MapReduce server took the responsibility of actually spawning the workers via a combination of threading, calls to **`system`**, and the **`ssh`** user program. This worked for our implementation because we had at most 8 workers at any one time. In practice, MapReduce implementations manage thousands of workers across thousands of machines. Why does our implementation not scale to the realm where there are thousands of workers instead of at most 8 (even if the **`myth`** cluster actually had thousands of machines)? What changes can realistically be made to the implementation to deal with thousands of workers instead of just 8?

   The maximum number of processes per user is typically in the hundreds, and the number of threads per process is limited to the low hundreds. Criteria: 2 points

   One possibility: Connect to each of the workers on the default ssh port, and speak ssh protocol to launch worker processes. Leave those connections open (a process can maintain tens of thousands of open connections) until it's heard all workers have finished, then close all connections. Criteria: 2 points for something solid, 1 point for something vague, 0 points otherwise.