Autumn 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.) 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 should you have questions.

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. **`SIGCALL`** and System Call Traces	[10]	_____	_____
2. Multiprocessing Redux	[12]	_____	_____
3. Concurrent **`and`** Evaluation	[10]	_____	_____
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 open(const char *pathname, int flags); // returns descriptor
int close(int fd); // ignore retval
int dup2(int old, int new); // ignore retval
int execvp(const char *path, char *argv[]); // ignore retval
#define WEXITSTATUS(status) // macro

// vector
template <typename T>
class vector {
public:
   vector();
   vector(size_t count, const T& elem = T());
   size_t size() const;
   const T& operator[](size_t index) const; // shorthand is v[index]
}

// thread
class ThreadPool {
public:
   ThreadPool(size_t size);
   ~ThreadPool();
   void schedule(const function<void(void)>& thunk);
   void wait();
};

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

class semaphore {
public:
   semaphore(int count = 0);
   void wait();
   void signal();
};

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

Problem 1: `SIGCALL` and System Call Traces [10 points]

Imagine the set of system calls has been extended to include a new one called **`getsyscall`**, which has the following prototype:

```
const char *getsyscall(pid_t pid);
```

`getsyscall` accepts the id of a child process and returns the name of a system call recently invoked by that child. More specifically, the names of the child's system calls are queued up, behind the scenes in FIFO order, and **`getsyscall`** can be repeatedly called to surface the names of all system calls in the order they were invoked. If all of a child's system calls made thus far have been surfaced, **`getsyscall`** returns **`NULL`**.

Further imagine the set of signals (e.g. **`SIGCHLD`**, **`SIGTSTP`**, etc.) has been extended to include a new type, **`SIGCALL`**. The kernel sends a **`SIGCALL`** to a process every time a child process makes a system call (e.g. **`fork`**, **`execvp`**, **`open`**, **`read`**, **`accept`**, etc.).

By default, the **`SIGCALL`** signal is ignored, but a custom signal handler (e.g. **`handleSIGCALL`**, where you specify the implementation) can be installed. A custom **`SIGCALL`** signal handler is invoked whenever the kernel has **`SIGCALL`**'ed the parent one or more times since the handler last executed.

These new directives (**`SIGCALL`**, **`getsyscall`**) can be used to implement a program called **`trace`**. In particular, **`trace`** takes the name of an executable and the arguments that should be passed to that executable's **`main`** function and runs that executable in a child process. The child process runs as it normally would, redirecting **`STDOUT_FILENO`** and **`STDERR_FILENO`** to a file called **`"/dev/null"`**[1], but the **`trace`** executable itself lists out the sequence of system calls made by the child.

Example time: I'm able to run my own Assignment 7 solution this way:

```
jerry> mr --config odyssey-partial.cfg
Mapper executable: word-count-mapper
Reducer executable: word-count-reducer
// output omitted for brevity
Remote ssh command on myth15 executed and returned a status 0.
Reduction of all grouped, intermediate chunks now complete.
Server has shut down.
All done!
```

If I'm curious what system calls are made when **`mr`** executes (and don't care to see to its output), I can execute this instead:

[1] When one wants to discard standard output and/or standard error, one typically redirects one or both of them to a file called **`/dev/null`**. For this problem, we don't want the child process's output to interfere with **`trace`**'s.

```
jerry> trace mr --config odyssey-partial.cfg
execve
brk
access
mmap
open
stat
open
stat
// output omitted for brevity
futex
munmap
write
exit_group
jerry>
```

To be clear, the execution of **mr**, when invoked with **--config** and **odyssey-partial.cfg** as arguments, involves calls to **execve**, **brk**, **access**, **mmap**, **open**, **stat**, and so forth, all of which are system calls. Not surprisingly, some system calls are invoked several times, so you'd expect to see **open**, **read**, **write**, **accept**, and **close** listed many, many times in this particular **trace**'s output.

For this problem, you're to present the entire **main** function needed to implement **trace**. Here are some assumptions to make and constraints to be met:

- You can assume that **trace** gets at least one argument, and that argument is the name of another legitimate executable.
- You may declare a single global variable of type **pid_t**, which can be used to store a single process id. (Actually, I declared it for you. You can't declare any others.)
- You should assume that all system calls succeed (i.e. you needn't do any error checking.)
- You should assume that the child executable returns normally (e.g. returns some value from its own **main**, or calls **exit** with some legitimate value; there's no need to check for abnormal termination.)
- The child process being traced should have its output (both standard and error) redirected to **"/dev/null"**. (**"/dev/null"** should be opened as a regular file with the flag **O_WRONLY**.)
- **trace** should wait until the child process terminates, and the child process's exit status should be trace's exit status.
- Your solution should implement a custom **SIGCALL** handler and install it.
- Your solution may not busy wait anywhere.

Use the next page to present your entire program. An unnecessarily complicated solution will not get full credit, even if it's correct.

```
/**
 * File: trace.c
 * -------------
 * Implements an executable that traces all of the system calls
 * made by another executable.
 *
 * myth8> ls -lta /usr/class/cs110/repos/assign1
 * // lists all of the SUNet IDs of those who were
 * // enrolled when the first assignment went out
 * myth8> trace ls -lta /usr/class/cs110/repos/assign1
 * // executes ls -lta /usr/class/cs110/repose/assign1 in a
 * // child process, and lists the sequence of
 * // system calls ls uses while executing with its two arguments
 */

static pid_t pid = 0;
static void handleSIGCALL(int sig) {









}

int main(int argc, char *argv[]) {
   signal(SIGCALL, handleSIGCALL);
```

Problem 2: Multiprocessing Redux [12 points]

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. [4 points] Recall that one can route the standard output of one process to the standard input of a second using | (the vertical bar) on the command line. In fact, we can cascade pipes—that's what they're called—so that the standard output of the first process sources the standard input of a second, the standard output of the second sources the standard input of the third, and so forth.

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
```

The **echo** process, which immediately prints and flushes **abcdefghij** to the standard input of the first **conduit** process in the pipeline, has a process id of 20686. The first **conduit** process—the one fed by **echo**—has a process id of 20687, the second has a process id of 20688, and so forth.

- [2 points] Assume I send a **SIGTSTP** to process id 20687 after two seconds. What state will the other four processes be in 20 seconds later, assuming I don't send any other signals?

- [2 points] Assume I send a **SIGTSTP** to process id 20690 after two seconds. What state will the other four processes be in 20 seconds later, assuming I don't send any other signals?

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.

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?

Problem 3: Concurrent `and` Evaluation [10 points]

Some programming languages—not C, C++, or Java, but others like Haskell, Erlang, and Scheme—could support the concurrent evaluation of all of the conjuncts contributing to a compound Boolean expression, as with **`a and b + c > 0 and !isPrime(n)`**. Of course, the overall value of the Boolean expression is true if and only if each of the conjuncts evaluates to true. If each of the conjuncts is computationally expensive, but the machine has multiple processors and/or multiple cores, each of the conjuncts can be evaluated concurrently to speed things up.

Assume a programming language models the notion of a Boolean expression like this:

```
class BoolExpression {
public:
  bool evaluate() const; // evaluates the expression and returns true or false
  // constructor and other methods omitted
private:
  // implementation details omitted
};
```

You're to implement the **`concurrentAnd`** function, which accepts a **`vector`** of **`BoolExpression`**s, evaluates them, and returns **`true`** if and only if all evaluate to **`true`**.

Some constraints and tips:

- Each of the supplied **`BoolExpression`**s should be evaluated using a global **`ThreadPool`** called **`pool`**. (I've already declared this for you.)
- You can assume **`BoolExpression::evaluate`** and **`ThreadPool::schedule`** are each thread-safe.
- You should **not** implement any form of short-circuit evaluation—i.e. you should evaluate **all** of the supplied **`BoolExpression`**s, even if one of them evaluates to **`false`** early on.
- Your implementation should use a **`condition_variable_any`** to efficiently block until all contributing **`BoolExpression`**s have been evaluated. In particular, you may not busy wait anywhere.
- Your implementation can make use of other basic concurrency directives (**`mutex`**, **`lock_guard`**, etc.), except that you may not use the **`atomic`** class.
- You can assume that **none** of the **`BoolExpression`**s in the supplied **`vector`** recursively depend on **`concurrentAnd`**. However, there may be other calls to **`concurrentAnd`** executing concurrently.
- You may not declare any global variables other than the one I've declared for you.
- The capture clause **`[&, i, j]`** captures all variables in the surrounding scope by reference, except for the variables **`i`** and **`j`**, which are captured by value.

Use the next page to present your implementation. An unnecessarily complicated solution will not receive full credit, even if it's correct.

```
/**
 * Function: concurrentAnd
 * -----------------------
 * Concurrently evaluates each of the BoolExpressions, and returns
 * true if and only if all BoolExpressions evaluate to true.  This version
 * does not support short circuit evaluation.
 */

static const size_t kPoolSize = 64;
static ThreadPool pool(kPoolSize);
static bool concurrentAnd(const vector<BoolExpression>& expressions) {
```

Problem 4: Concurrency and Networking Redux [18 points]

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] There are a very limited number of scenarios where busy waiting **is** 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.

b. [3 points] 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`**.

c. [2 points] Briefly explain the primary advantage of using a **lock_guard<mutex>** over exposed calls to **mutex::lock** and **mutex::unlock**.

d. [3 points] 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 purpose does this buffer serve?

e. [3 points] With Assignment 6, 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.

f. [4 points] 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 32 (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 32?