Final Exam, CS 110, Spring 2019

SUNet ID (username):_____@stanford.edu

Last Name: _____
First Name: _____

I accept the letter and spirit of the honor code.
[signed]

_____

Problem 1: Serving CGI

20 points

This question has four parts

For this problem you will create a simple file server, able to return files to a client. It will have an extra feature which is common on web servers: it will be able to run programs and send the *output* of the program back to the web page. The standard protocol for running files is called the "Common Gateway Interface," or *CGI* for short. If a web server receives a file with an extension of **`cgi`** it will *run* the file as a program, instead of simply returning the file to the client. It will capture the output of the program and send *that* to the client.

Your server will have six functions, and you will write *four* of them for this problem. Here is the documented header file for the server:

```
 *  File: cgi-server.h
 *  ------------------------------
 *  File server that runs scripts if the file
 *  suffix is .cgi
 *  The server also servers a proper Content-type: html/text
 *  when the file suffix is .html
 *  and Content-type: text/plain when the file suffix is anything els
 *
 */

#include <iostream>                // for cout, cerr, endl
#include <sstream>
#include <fstream>
#include <algorithm>
#include <sys/socket.h>            // for accept,  etc.
#include "socket++/sockstream.h"   // for sockbuf, iosockstream
#include "server-socket.h"
#include "thread-pool.h"
#include "subprocess.h"

using namespace std;

// constants and ENUM
static const unsigned short kDefaultPort = 12345;
enum ContentType {TEXT, HTML, CGI};
// to use an enum, simply refer to it as you would any integer type,
e.g.
// if (contentType == TEXT) ...

/* function: sendResponse
 * parameters:
 *   iosockstream &ss        : the client to send the response to
 *   string responseStatus   : the entire responseStatus header
 *   string payload          : the payload to send to the client
 *   ContentType contentType : self-explanatory
 * notes:
```

```
 *   This function should perform three functions:
 *     1. Send response status to client
 *     2. If the contentType is HTML, send the Content-type as
"Content-type: text/html"
 *        If the contentType is anything else other than CGI,
 *           send the Content-type as: "Content-type: text/plain"
 *        If the contentType is CGI, do not send a Content-type header
 *     3. Send the payload
*/
static void sendResponse(iosockstream &ss,
        string responseStatus,
        string payload,
        ContentType contentType);

/* function: parseRequest
 * parameter:
 *   iosockstream &ss : the client to read the request from
 * return value:
 *   string : the relative path of the file, e.g., path/filename.txt
 * notes:
 *   assume GET request is in the form
 *     GET http://webserver.com/path/filename.ext
 *   This function should read in all of the headers and return
 *     just the path and filename, without the webserver
 *   Example header to parse:
 *     GET http://webserver.com/path/filename.ext
 *     Host: webserver.com
 *     ...other headers we don't care about, ending with a blank line
 *
 *   Example return value for above header:
 *     "path/filename.ext"
*/
static string parseRequest(iosockstream &ss);

/* function: hasEnding
 * parameter:
 *   string const &fullString : the string to look in
 *   string const &ending     : the ending (e.g., ".txt")
 * return value:
 *   bool : true if the string ends with ending, false otherwise
 * notes:
 *   Helper function provided for you
*/
bool hasEnding (string const &fullString, string const &ending);

/* function: readFile (method 1)
 * parameter:
 *   string const &filename : the filename to read
 * return value:
```

```
 *   string : the entire contents of the file
 * notes:
 *   Helper function provided for you
*/
static string readFile(const string &filename);

/* function: readFile (method 2)
 * parameter:
 *   int fd : a file descriptor to read from
 * return value:
 *   string : the entire contents of the file
 *     referred to by the file descriptor
 * notes:
 *   You must use the read() system call for this function
*/
static string readFile(int fd);

/* function: serveRequest
 * parameter:
 *   int client : a file descriptor returned from the accept() system
call
 * return value:
 *   none
 * notes:
 *   once a request is parsed, if the extension of the file is .cgi,
 *     then the program should be run and the output should be sent to
 *     the client. You can assume that the program to be run does not
 *     take any arguments
 *   if the request is a file, the file should be read in and sent to
 *     the client
 *   if the file exists, the response should be "HTTP/1.0 200 OK"
 *   if the file does not exist, the response should be
 *     HTTP/1.0 404 NOT FOUND and the payload should simply be
 *     "File not found!"
*/
static void serveRequest(int client);
```

Here is the **`main`** function, written for you:

```
  int main(int argc, char *argv[]) {
    int server = createServerSocket(kDefaultPort);

    cout << "Server listening on port " << kDefaultPort << "." <<
endl;
    ThreadPool pool(4);
    while (true) {
        int client = accept(server, NULL, NULL);
        pool.schedule([client] { serveRequest(client); });
    }
    return 0;
```

```
}
```

Notes:

- You should use the **`subprocess`** function to launch the cgi programs. See the Relevant Prototypes handout for the function signature for **`subprocess`** and for **`struct subprocess_t`**.
- You can assume that there will be no payload on any request

a) Write the **`sendResponse`** function below. [4 points]

```
static void sendResponse(iosockstream &ss,
        string responseStatus,
        string payload,
        ContentType contentType) {
```

b) Write the **`parseRequest`** function below. [3 points]

```
  static string parseRequest(iosockstream &ss) {
```

c) Write the **`readFile(int fd)`** function below. [4 points]

```
// You must use the read() system call for this function
static string readFile(int fd) {
```

d) Write the **`serveRequest`** function below. [9 points]

```
static void serveRequest(int client) {
  sockbuf sb(client); // destructor closes socket
  iosockstream ss(&sb);
```

Problem 2: Signal Handling with Information:

A Process Circle

15 points

One of the limitations of using signal handlers to signal processes is that there is no information passed when a signal handler is invoked. Or is there? The function signature for a signal handler is as follows:

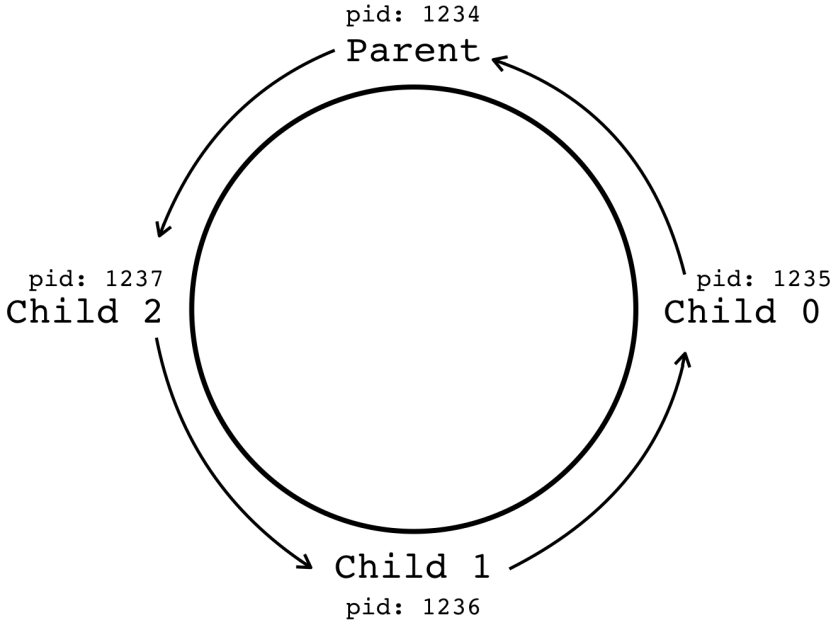
```
typedef void (*sighandler_t)(int);
```

In other words, all of our signal handlers have had the following form:

```
void sighandler(int sig) { ... } // sig is the signal number
```

Therefore, with a little bit of ingenuity, we can use the signal information to help drive the logic of the signal handler, even though there is no shared memory between processes. We will leverage the fact that there are two signals, **`SIGUSR1`** and **`SIGUSR2`** that are for any use our program chooses, and we will use the same signal handler for both signals. If the signal passed in is **`SIGUSR1`** then the logic of the handler will be based on that information, and likewise if the signal is **`SIGUSR2`**, we can have different logic.

For this problem, create a "process circle" of length **`n`**, where the parent and **`n-1`** children are in the circle. You can think of the circle as a circular linked list, if you'd like. The circle is built such that the first child signals the parent, the second child signals the first child, etc., and the parent signals the last child:



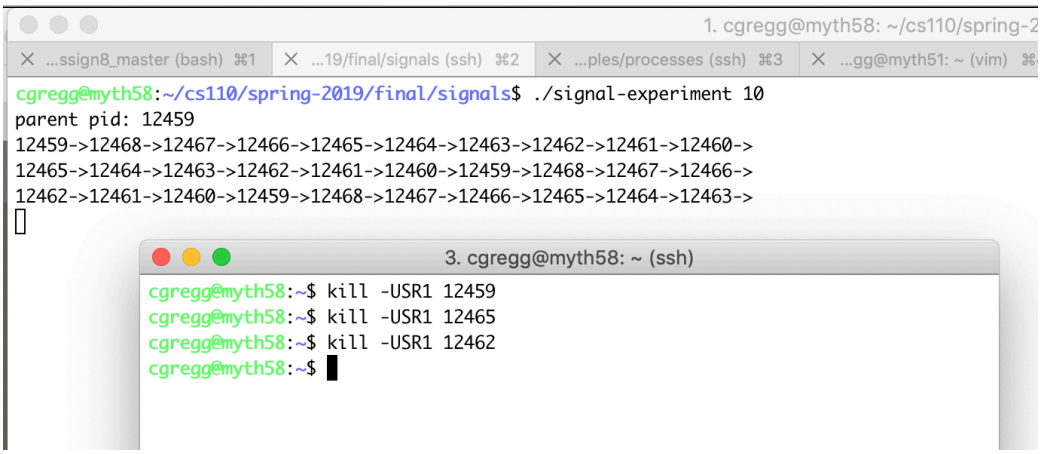
When a **`SIGUSR1`** is sent to *any* of the processes, that process should print out its **`PID`**, an arrow (**`->`**), and then signal the next process in the circle. The printing stops after the last process in the circle has printed. In other words, you have to *avoid an infinite loop*. For example, if process 1 above was signaled with **`kill(1236, SIGUSR1)`**, the following should print:

```
1236->1235->1234->1237->
```

Write the signal handler, **`usrHandler`**, and the circle creation function, **`createProcessCircle`** below.

Notes:

- Because of the infinite loop problem, you have to figure out how to ensure that the looping stops after all of the nodes in the circle have printed. This is where the two different signals will be helpful. In other words, how can you tell one node that it will need to stop if it is signaled a second time, and how can you ensure that the other nodes don't stop on the first time?
- You may use **`static`** variables inside the signal handler to retain state, but you are not allowed to create other global variables.
- The parent and each child should end up in a **`while(true) {...}`** loop, with a proper **`sigsuspend`** command to keep the process off the processor when not receiving signals. You don't need to block any signals, but you should use a **`sigsuspend`** properly in the **`while(true) {... }`** loop to avoid busy waiting. In other words: once the children have all been launched, they go into a non-busy waiting state, only to be woken up by the signal handler. The parent also ends up in a **`while(true) {...}`** loop, and also waits in a non-busy way.
- The example above should be able to be run multiple times, with different processes in the circle. In other words, you should be able to generate a circle diagram as above as many times as you wish (although the printing must be finished before trying a different starting process). See the diagram below, which shows two windows. In the larger window, the process circle program has been launched. In the smaller window, we have sent the **`SIGUSR1`** signal three times, generating the output in the larger window:



```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <errno.h>
#include "exit-utils.h"  // exitIf, exitUnless
#include "sleep-utils.h"
#include <stdbool.h>

// see main function on next page

static pid_t prevPid; // global

static void usrHandler(int sig) {
    // STUDENT CODE HERE









}
```

```
void createProcessCircle(int circleSize) {
    // STUDENT CODE HERE









}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("usage:\n\t%s circleSize\n",argv[0]);
        return -1;
    }
    signal(SIGUSR1, usrHandler);
    signal(SIGUSR2, usrHandler);

    printf("parent pid: %d\n", getpid());
    createProcessCircle(atoi(argv[1]));
    return 0;
}
```

Problem 3: Concurrency and Getting Supplies to Mars

15 points

You're writing a simulation for moving necessary supplies to support a Mars colony. There are **technician**s who organize the supplies into a container and load it onto a rocket, **astronout**s who fly the rockets full of containers to Mars, and **martian**s who unpack the containers. At the beginning of the simulation, one thread is spawned for each **technician** and one for each **astronaut**. Threads for the **martian**s are spawned as needed (this process is described below). There is only one launchpad, which is used to load the cargo, and the astronouts (who fly their own rockets) must wait for the launch pad to become available.

A **technician** is responsible for organizing just one container and getting it onto a rocket (in other words, they perform the loading, as well). First, they must round up an empty container. There are a fixed number of containers available for all technicians to use. Once the technician has a container, they **organize** it with stuff and then load it. To load, a rocket needs to be at the launch pad and have space for another container. The technician who puts the final container on the rocket tells the astronaut to go. Once their container is loaded, the technician's work is done (and the thread exits).

Each **astronaut** thread is responsible for making one trip to pick up containers, taking them to Mars, and unloading. The **astronaut** first "**refuel**s" the rocket and then goes to pick up containers. There is only one launch pad, so only one rocket can be loading at once. The astronaut hangs out while the technicians pile on the containers. Each rocket has a capacity that is established when created. The rocket is considered loaded when it is full to its capacity or it is the last rocket and there are no more containers to load. After the rocket is loaded, the astronaut "**fly**s" the rocket to Mars. On Mars, the astronaut takes each container off the rocket and dispatches a separate **martian** thread to unpack it. Once the **martian**s are dispatched, the astronaut waits for all spawned **martian** threads to finish, and only when that happens is the **astronaut** done (and the astronaut thread exits).

Each **martian** is responsible for "**unpack**ing" one container. Once the container is unpacked, it is empty and should be made available to the packers who are in need of empty containers. (Assume that each container has a corresponding container that has already made the round trip and as a container is finished on Mars, one is immediately ready back on Earth).

Several rockets can be unloaded simultaneously, and **martian**s can work in parallel. Once the **martian** unpacks their box, their job is done (and their thread exits).

Here is the starting main function for the packing simulation.

```
const static int kNumContainersToMove = 300;
const static int kNumEmptyContainers = 70;

int main(int argc, char *argv[]) {
  RandomGenerator rgen;
  vector<thread> threads;
  int totalRocketCapacity = 0;
```

```
  for (int i = 0; i < kNumContaineresToMove; i++) {
     // create technician threads
     threads.push_back(thread(technician));
  }

  while (totalRocketCapacity < kNumContainersToMove) {
    // create astronaut threads
    int thisRocketHolds = rgen.getNextInt(10, kNumEmptyContainers);
    totalRocketCapacity += thisRocketHolds;
    threads.push_back(thread(astronaut, thisRocketHolds));
  }

  for (thread& t: threads) t.join();
  return 0;
}

// simulation functions already written for you
static void organize(); // for technician, to fill container with
stuff
static void refuel();   // for astronaut, to prepare rocket
static void fly();      // for astronaut, fly to Mars
static void unpack();   // for martian, to unload container contents
```

Assume the above helpers are already written and are thread-safe, and you can just call them when you need to. Your job will be to write the **technician**, **astronaut**, and **martian** functions to properly synchronize the different activities and efficiently share the common resources.

Declare your **global** variables, **mutex**es, and **semaphore**s, and then implement the **technician** [5 points], **astronaut** [8 points], and **martian** [2 points] functions. Do not busy wait anywhere!

```
// declare your global variables here:
// STUDENT CODE HERE
```

```
void technician() {
  // STUDENT CODE HERE









}

void astronaut(int capacity) {
  // STUDENT CODE HERE

















}

void martian() {
  // STUDENT CODE HERE









}
```

Problem 4: Miscellaneous Questions

8 points

This problem has 4 parts.
Answer the four questions below. *Limit your answers to 150 words or less. Please. :)*

a) Generally, there are two different ways we use semaphores (throughout lecture and the assignments, you have seen semaphores used in many different scenarios, but the way we use semaphores in these two scenarios falls into one of these two usage patterns). Describe these two use cases, and be explicit about how they differ. [2 points]

b) Describe the difference between an "I/O Bound" program and a "CPU Bound" program. If we were to speed up the CPU on our computer, but it did not speed up the problem we were trying to solve with a program, would the program be I/O Bound or CPU Bound? [2 points]

c) Our own solution for the **`ThreadPool`** class includes "lazy" initialization of worker threads. This means that we do not actually launch any worker threads until they are needed. So, for example, if we create a 64-thread **`ThreadPool`** (e.g., **`ThreadPool tp(64)`**), and then we only scheduled ten threads to work, we would only actually create ten worker threads, at the time they are scheduled. As another example, suppose we had written the Farm problem with threads instead of processes. We could launch all farm workers at the beginning (non-lazy), or we could only launch a worker when it is needed, up to a maximum amount of total worker threads allowed (lazy). Explain the benefit of lazy thread initialization, and explain one downside to lazy thread initialization. [2 points]

d) The following is the general pattern for using a condition variable:

```
m.lock();
cv.wait(m, []{return condition > 0});
m.unlock();
```

Explain *two reasons* why a mutex is necessary for a condition variable to work properly. [2 points]

Relevant Prototypes

```
// filesystem access
int close(int fd); // ignore retval
int dup(int fd);   // ignore retval
int dup2(int oldfd, int newfd); // ignore retval
int pipe(int fds[]); // ignore retval
int pipe2(int fds[], int flags); // ignore retval, flags typically O_CLOEXEC
#define STDIN_FILENO 0
#define STDOUT_FILENO 1

// exceptional control flow and multiprocessing
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
int execvp(const char *path, char *argv[]); // ignore retval
int kill(pid_t pid, int signal); // ignore retval
typedef void (*sighandler_t)(int sig);
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
int sigemptyset(sigset_t *set); // ignore retval
int sigaddset(sigset_t *set, int sig); // ignore retval
int sigprocmask(int how, const sigset_t *set, sigset_t *old);

#define WIFEXITED(status)   // macro
#define WIFSTOPPED(status)  // macro

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

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

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

```
class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(Thunk t);
   void wait();
};

struct subprocess_t {
 pid_t pid;
 int supplyfd;
 int ingestfd;
};

subprocess_t subprocess(char *argv[],
      bool supplyChildInput,
      bool ingestChildOutput);

template <typename T>
class vector {
public:
   size_t size() const;
      void push_back(const T& elem);
      T& operator[](size_t i);
      const T& operator[](size_t i) const;
};

template <typename T>
class list {
public:
bool empty() const;
size_t size() const;
void push_back(const T& elem);
T& front();
      void pop_front();
};

template <typename U, typename V>
struct pair {
   U first;
   V second;
};

Template <typename Key, typename Value>
class map {
public:
   // iter points to pair<Key, Value>
   size_t size() const;
   iter find(const Key& k);
   Value& operator[](const Key& k);
};
```