Lab Solution 4: `assign2` Redux and Threads

Lab by Jerry Cain.

Before starting, go ahead and clone the `lab4` folder, which contains a working solution to Problem 2. My expectation is that you spend the majority of your time on Problems 1, 3, and 4, but Problem 2 is there for you to play with at the end of section or on your own time.

```
poohbear@myth55:~$ git clone /usr/class/cs110/repos/lab4/shared lab4
poohbear@myth55:~$ cd lab4
poohbear@myth55:~$ make
```

Solution 1: `assign2` Redux

Here are a collection of short answer questions drilling your understanding of `subprocess` and `farm`. It's not uncommon for students to get working solutions to assignments and still not be entirely clear why they work. These questions are here to force you to think big picture and understand the systems concepts I feel are important.

- Your Assignment 2 implementation of `subprocess` required two pipes–one to foster a parent-to-child communication channel, and a second to foster a child-to-parent communication channel. Clearly explain why a single pipe shouldn't be used to support both communication channels.
    - *Because both child and parent would need to write to `fds[1]`, and there'd be no generic way to codify whether material in the pipe is intended for child or parent, so one could accidentally read what's intended for the other.*
- You've seen `dprintf` in lecture and in the `assign2` handout, and it presumably contributed to most if not everyone's `farm` implementation. Explain why there's a `dprintf` function, but there's no analogous `dscanf` function. Hint: Think about why `dprintf(fd, "%s %d\n", str, i)` would be easy to manage whereas `dscanf(fd, "%s %d\n", str, &i)` wouldn't be. Read the first few lines of the `man` pages for the traditional `fprintf` and `fscanf` functions to understand how they operate.
    - *With `dprintf`, the entire string can be constructed (and its length computed) before the underlying `write` calls. `dscanf` might need to read an extra character (e.g. the space in `"1234 "`) to confirm a placeholder like `%d` has been matched, and there's no way to revert that extra read. `fscanf`, other the other hand, is framed in terms of `FILE *`s, and those `FILE *`s address data structures that include not only the underlying descriptor, but also a cache of the most recently read block of characters, which can be consulted and used for future calls to `fscanf` on the same `FILE *`.*
- Consider the implementation of `spawnAllWorkers` below. Even though it rarely causes problems, the line in **bold** *italics* technically contains a race condition. Briefly describe the race condition, and explain how to fix it.

```
struct worker {
   worker() {}
   worker(char *argv[]) : sp(subprocess(argv, true, false)), available(false) {}
   subprocess_t sp;
   bool available;
};

static const size_t kNumCPUs = sysconf(_SC_NPROCESSORS_ONLN);
static vector<worker> workers(kNumCPUs);
static size_t numWorkersAvailable;

static const char *kWorkerArguments[] = {
   "./factor.py", "--self-halting", NULL
};
```

```
static void spawnAllWorkers() {
   cout << "There are this many CPUs: " << kNumCPUs << ", numbered 0 through "
        << kNumCPUs - 1 << "." << endl;
   for (size_t i = 0; i < workers.size(); i++) {
      workers[i] = worker(const_cast<char **>(kWorkerArguments));
      assignToCPU(workers[i].sp.pid, i); // implementation omitted, irrelevant
   }
}

int main(int argc, char *argv[]) {
    signal(SIGCHLD, markWorkersAsAvailable); // markWorkersAsAvailable is correct
    spawnAllWorkers();
    // other functions, assume all correct
    return 0;
}
```

  - *The right hand side constructs a temporary that launches a process that self-halts before content of temporary is copied into* ***`workers[i]`****. If that happens, the* ***`SIGCHLD`*** *handler is invoked and crawls over an array that doesn't have the pid yet, and sadness ensues.*
  - *Solution is to block* ***`SIGCHLD`*** *before the bold, italic line and then unblock after.*

- While implementing the **`farm`** program for **`assign2`**, you were expected to implement a **`getAvailableWorker`** function to effectively block **`farm`** until at least one worker was available. My own solution relied on a helper function I called **`waitForAvailableWorker`**, which I present below. After analyzing my own solution, answer the following questions:
  - Assuming no signals are blocked at the time **`waitForAvailableWorker`** is called, clearly identify when **`SIGCHLD`** is blocked and when it is not.
    - *TL;DR:* ***`SIGCHLD`*** *isn't blocked prior to the* ***`sigprocmask`*** *call, within the call to* ***`sigsuspend`****. It's blocked everywhere else.*
    - *More detail: The first three lines create a singleton mask containing just* ***`SIGCHLD`****, and the fourth line blocks it. The* ***`while`*** *loop test is evaluated while the* ***`SIGCHLD`*** *block is in place, which is exactly what we want, because we don't want its evaluation to be interrupted by the execution of* ***`markWorkersAsAvailable`****. If the test passes, the call* ***`sigsuspend`*** *simultaneously lifts the block on* ***`SIGCHLD`*** *(remember:* ***`existingmask`*** *is empty) and puts the process to sleep until some (or rather, any) signal is received, at which point any installed handler (presumably the* ***`SIGCHLD`*** *handler, since we expect* ***`SIGCHLD`*** *to be the signal that comes in) executes with high priority. After any handler exits,* ***`sigsuspend`*** *returns while simultaneously restoring the mask that was in place before* ***`sigsuspend`*** *was called, and re-evaluates the test again with a block on* ***`SIGCHLD`****. That process repeats until the* ***`while`*** *loop test fails, at which point* ***`waitForAvailableWorker`*** *returns, leaving the block on* ***`SIGCHLD`*** *in place.*
  - Had I accidentally passed in **`&additions`** to the **`sigsuspend`** call instead of **`&existing`**, the farm could have deadlocked. Explain why.
    - ***`numWorkersAvailable == 0`*** *could pass, and then* ***`sigsuspend`*** *would force* ***`farm`*** *to deadlock, as only* ***`SIGCHLD`*** *signals are coming in and capable of changing* ***`numWorkersAvailable`****, and they're blocked.*
  - Had I accidentally omitted the **`sigaddset`** call and not blocked **`SIGCHLD`**, **`farm`** could have deadlocked. Explain how.
    - ***`numWorkersAvailable == 0`*** *passes,* ***`farm`*** *is swapped off CPU, all* ***`kNumCPUs`*** *workers self-halt, all* ***`kNumCPUs`*** ***`SIGCHLD`****s handled by one* ***`SIGCHLD`*** *handler call,* ***`farm`*** *descends into* ***`sigsuspend`****, and no additional* ***`SIGCHLD`****s ever arrive to wake* ***`farm`*** *up.*
  - In past quarters, I saw a bunch of students who lifted the block on SIGCHLD before the two lines in bold instead of after. As it turns out, executing **`numWorkersAvailable--`** after the block is lifted can cause

problems, but executing **`workers[i].available = false`** actually can't. Explain why the placement of the **`--`** is more sensitive to race conditions than the Boolean assignment is.

- *Execution of **`--`** could be interrupted and confused by execution of **`++`** within **`SIGCHLD`** handler.*
- *When Boolean assignment is executed, relevant worker is halted, so interruption by **`SIGCHLD`** handler would hop over **`workers[i]`**, as its pid can't possibly have been returned by handler's **`waitpid`** call until **`workers[i]`** is continued.*

```
static sigset_t waitForAvailableWorker() {
   sigset_t existing, additions;
   sigemptyset(&additions);
   sigaddset(&additions, SIGCHLD);
   sigprocmask(SIG_BLOCK, &additions, &existing);
   while (numWorkersAvailable == 0) sigsuspend(&existing);
   return existing;
}

static size_t getAvailableWorker() {
   sigset_t existing = waitForAvailableWorker();
   size_t i;
   for (i = 0; !workers[i].available; i++);
   assert(i < workers.size());
   numWorkersAvailable--;
   workers[i].available = false;
   sigprocmask(SIG_SETMASK, &existing, NULL); // restore original block set
   return i;
}
```

- The first quarter I used this assignment, a student asked if one could just use the **`pause`** function instead, as the second version of **`waitForAvailableWorker`** does below. The zero-argument **`pause`** function doesn't alter signal masks like **`sigsuspend`** does; it simply halts execution until the process receives any signal whatsoever and any installed signal handler has fully executed. This is conceptually simpler and more easily explained than the version that relies on **`sigsuspend`**, but it's flawed in a way my solution in the preceding bullet is not. Describe the problem and why it's there.

```
static sigset_t waitForAvailableWorker() {
   sigset_t mask;
   sigemptyset(&mask);
   sigaddset(&mask, SIGCHLD);
   sigprocmask(SIG_BLOCK, &mask, NULL);
   while (numWorkersAvailable == 0) {
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      pause();
      sigprocmask(SIG_BLOCK, &mask, NULL);
   }
}
```

  - *The program can deadlock. How? After **`SIGCHLD`** is unblocked, all workers become available before **`pause`** gets called, **`numAvailableWorkers`** becomes maximum value, **`main`** execution flow descends into **`pause`** and no more signals are ever sent.*

Solution 2: Multithreaded `quicksort`

`quicksort` is an efficient, divide-and-conquer sorting algorithm whose traditional implementation looks like this:

```
static void quicksort(vector<int>& numbers, ssize_t start, ssize_t finish) {
  if (start >= finish) return;
  ssize_t mid = partition(numbers, start, finish);
  quicksort(numbers, start, mid - 1);
  quicksort(numbers, mid + 1, finish);
}

static void quicksort(vector<int>& numbers) {
  quicksort(numbers, 0, numbers.size() - 1);
}
```

The details of how `partition` works aren't important. All you need to know is that a call to `partition(numbers, start, finish)` reorders the elements between `numbers[start]` and `numbers[finish]`, inclusive, so that numbers residing within indices `start` through `mid-1`, inclusive, are less than or equal to the number at index `mid`, and that all numbers residing in indices `mid + 1` through `stop`, inclusive, are strictly greater than or equal to the number at index `mid`. As a result of this reorganization, we know that, once `partition` returns, the number residing at index `mid` actually belongs there in the final sort.

What's super neat is that the two recursive calls to `quicksort` can execute in parallel, since the sequences they operate on don't overlap. In fact, to make sure you get some practice with C++ threads right away, you're going to cannibalize the above implementation so that each call to `quicksort` spawns off threads to recursively sort the front and back portions at the same time.

- Descend into your clone of the shared `lab4` directory, and execute the sequential `quicksort` executable to confirm that it runs and passes with flying colors. Then examine the `quicksort.cc` file to confirm your understanding of `quicksort`. You can ignore the details of the `partition` routine and just trust that it works, but ensure you believe in the recursive substructure of the three-argument `quicksort` function.
  - *Nothing to report here. Hopefully you believe in the recursive substructure! :)*
- Now implement the `aggressiveQuicksort` function, which is more or less the same as the sequential `quicksort`, except that each of the two recursive calls run in independent, parallel threads. Create standalone threads without concern for any system thread count limits. Ensure that any call to `aggressiveQuicksort` returns only after its recursively guided threads finish. Test your implementation to verify it works as intended by typing `./quicksort --aggressive` on the command line.
  - My own solution is below. The `ref` is needed to be clear that a reference to numbers (and not a copy) is passed as the first argument to the `aggressiveQuicksort` thread routine. You'd think that the compiler could examine the prototype of the thread routine being installed, but it doesn't. It only looks at the prototype of the `thread` constructor, which relies on the C++ equivalent of `...` to accept zero or more arguments beyond the thread routine function name.

    ```
    static void aggressiveQuicksort(vector<int>& numbers, ssize_t start, ssize_t
      if (start >= finish) return;
      ssize_t mid = partition(numbers, start, finish);
      thread front(aggressiveQuicksort, ref(numbers), start, mid - 1);
      thread back(aggressiveQuicksort, ref(numbers), mid + 1, finish);
      front.join();
      back.join();
    }

    static void aggressiveQuicksort(vector<int>& numbers) {
      aggressiveQuicksort(numbers, 0, numbers.size() - 1);
    }
    ```

- Tinker with the value of `kNumElements` (initially set to the 128) to see how high you can make it before you exceed the number of threads allowed to coexist in a single process. You don't need to surface an exact number, as a ballpark figure is just fine.
  - *I tried 2500 and everything seemed to work without crashing. When I went with 5000, the test application aborted with an error message that included `"terminate called after throwing an instance of 'std::system_error'"`, which is the abort error message you'll generally see when too many threads (and therefore, too many subdivisions of the finite stack segment into thread stacks) exist at any one time. 4500 led to a crash, and so did 4250. When I tried 4000, it succeeded, so the actual number is probably somewhere in between 4000 and 4250.*
  - *In reality, the number of threads permitted to exist at any one time is just above 2000. (Actually, my own testing seems to indicate that 2041 is the limit.) . Note that `quicksort`ing an array of 4000 numbers doesn't require 2000 threads exist at any one moment, even if the total number of threads created over the lifetime of the sort is much greater than 2000. Some threads die before other threads are created.*
- Leveraging your `aggressiveQuicksort` implementation, implement the recursive `conservativeQuicksort` function so it's **just as parallel**, but the second recursive call isn't run within a new thread; instead, it's run within the same thread of execution as the caller. Test your implementation to verify it works as intended by typing in `./quicksort --conservative` on the command line.
  - *My own solution is below. Provided you understand my `aggressiveQuicksort` implementation, I suspect my `conservativeQuicksort` will be self-explanatory. Consider is a hybrid of sequential and aggressive.*

```
static void conservativeQuicksort(vector<int>& numbers, ssize_t start, ssize_
  if (start >= finish) return;
  ssize_t mid = partition(numbers, start, finish);
  thread front(conservativeQuicksort, ref(numbers), start, mid - 1);
  conservativeQuicksort(numbers, mid + 1, finish);
  front.join();
}

static void conservativeQuicksort(vector<int>& numbers) {
  conservativeQuicksort(numbers, 0, numbers.size() - 1);
}
```

- Time each of the three versions by using the `time` utility as you probably did in Assignment 2 while testing `farm`. Are the running times of the parallel versions lower or higher than the sequential versions? Are the running times what you expect? Explain.
  - *The timing results may surprise you and make you question whether we should ever use threading.*

```
poohbear@myth61:~/lab4$ echo $SHELL
/bin/bash
poohbear@myth61:~/lab4$ time ./quicksort --sequential
Trial #1000: SUCCEEDED!

real    0m0.025s
user    0m0.020s
sys     0m0.004s
poohbear@myth61:~/lab4$ time ./quicksort --aggressive
Trial #1000: SUCCEEDED!

real    0m14.698s
user    0m0.968s
sys     0m37.452s
poohbear@myth61:~/lab4$ time ./quicksort --conservative
Trial #1000: SUCCEEDED!
```

```
real    0m7.271s
user    0m0.400s
sys     0m18.424s
poohbear@myth61:~/lab4$
```

- *The issue here is that quicksort is what's considered **CPU-bound**, which is systems speak that means the vast majority of an algorithm's work is computational and requires CPU time to add, compare, move, and so forth. By introducing threading, we do enlist all eight of the myth's cores. But we also introduce an enormous amount of overhead (thread selection, context switches between threads, and so forth) so that the more parallelism we introduce, the longer the algorithm takes.*
- *In general, you rarely want the number of threads doing CPU-bound work in parallel to be more than the number of cores (or maybe twice the number of cores). The work that needs to be done using any one of the CPUs is the same regardless of whether one thread does it or 100 threads do it.*
- *Rest assured that we'll soon rely on threading to manage **I/O-** or **network-bound** algorithms, where the majority of the time is spent sleeping (off of the CPU) and waiting for I/O events. Even in these scenarios, you rarely want the number of threads to be more than a small multiple of your CPU and core count. So forgive this extreme example where we create as many threads as we want to. In general, we don't and won't do that. We just allow it during the initial window where we're just learning about threads.*

Solution 3: Threads vs Processes

According to some CAs, a good number of students are pondering the pros and cons of threads versus processes. To provide some answers, we'll lead you through a collection of short answer questions about multiprocessing and multithreading that focuses more on the big picture.

- What does it mean when we say that a process has a private address space?
  - *It means that its range of addresses are, by default, private to the owning process, and that it's impossible for an another arbitrary, unprivileged process to access any of it.*
- What are the advantages of a private address space?
  - *The fact that it's private, of course. Most other programs can't accidentally or intentionally examine another process's data.*
- What are the disadvantages?
  - *The fact that it's private, of course. Address space privacy makes it exceptionally difficult to exchange data with other processes, and works against any efforts to breezily distribute work over multiple CPUs using multiprocessing.*
- What programming directives have we used in prior assignments and discussion section handouts to circumvent address space privacy?
  - *Memory maps (via `mmap` and `munmap`) were used in the parallel `mergesort` section example to share large data structures between many processes. Signals and pipes have been used to foster communication between multiple processes in many assignments and section examples.*
- In what cases do the processes whose private address spaces are being publicized have any say in the matter?
  - *Processes have little say about pipes, since they're generally set up and used to rewire standard input, standard output, and standard error before `execvp` is used to bring a new executable into the space of running processes.*
  - *Don't like signals? You can install `SIG_IGN` to ignore most of them and/or block them for the lifetime of the executable if you want to. The only signals that can't be ignored or fully blocked are `SIGKILL` and `SIGCONT`.*
  - *Memory mapped segments can be forced onto child processes, but those memory mapped segments are pretty much ignored the instant a virtual address space is cannibalized by an `execvp` call.*

- When architecting a larger program like `farm` or `stsh` that relies on multiprocessing, what did we need to do to exchange information across process boundaries?
    - *`farm` relied on pipes to forward data to each of the workers, and it relied on `SIGTSTP`, `SIGCHLD`, `SIGCONT`, `SIGKILL`, and a `SIGCHLD` handler to manage synchronization issues. And in some sense, `farm` relied on the `execvp`'s string arguments to influence what each of the workers should be running.*
    - *`stsh` relied on pretty much the same thing, albeit for different reasons. `stsh` relies on signals and signal handlers to manage job control, terminal control transfer to be clear who's responding to keyboard events and any signals associated with them, and pipes to foster communication between neighboring processes in a pipeline.*
- Can a process be used to execute multiple executables? Restated, can it `execvp` twice to run multiple programs?
    - *Nope, not in general. A program like `stsh` can fork off additional processes, each of which `execvp`s, but that's not the same thing. Bottom line is that process spaces can't be reused to run multiple executables. They can only be transformed once.*
- Threads are often called lightweight processes. In what sense are they processes? And why the lightweight distinction?
    - *Each thread of execution runs as if it's a miniature program. Threads have their own stack, and they have access to globals, dynamic memory allocation, global data, system calls, and so forth. They're relatively lightweight compared to true processes, because you don't need to create a **new** process when creating a thread. The thread manager cordons off a portion of the full stack segment for the new thread's stack, but otherwise the thread piggybacks off existing text, heap, and data segments previously establish by `fork` and `execvp`. (For more info, go ahead and read the man page for `clone`).*
- Threads are often called virtual processes as well. In what sense are threads an example of virtualization?
    - *Virtualization is an abstraction mechanism used to make a single resource appear to be many or to make many resources appear to be one. In this case, the process is subdivided to house many lightweight processes, so that one process is made to look like many.*
- Threads running within the same process all share the same address space. What are the advantages and disadvantages of allowing threads to access pretty much all of virtual memory?
    - *It's a double-edged sword, as you'll learn with Assignments 5 and 6. Because all threads have access to the same virtual address space, it's relatively trivial to share information. However, because two or more threads might want to perform surgery on a shared piece of data, directives must be implanted into thread routines to ensure data and resources are shared without inspiring data corruption via race conditions, or deadlock via resource contention. That's a very difficult thing to consistently get right.*

Each thread within a larger process is given a thread id, also called a **tid**. In fact, the thread id concept is just an extension of the process id. For singly threaded processes, the pid and the main thread's tid are precisely the same. If a process with pid 12345 creates three additional threads beyond the main thread (and no other processes or threads within other processes are created), then the tid of the main thread would be 12345, and the thread ids of the three other threads would be 12346, 12347, and 12348.

- What are the advantages of leveraging the pid abstraction for thread ids?
    - *To the extent that the OS can view threads as processes, the OS can provide services on a per-thread basis instead of just a per-process one.*
- What happens if you pass a thread id that isn't a process id to `waitpid`?
    - *Poorly documented, but the takeaway here is that `waitpid` does not consider it to be an error if a thread id is passed to `waitpid`. `waitpid`'s man page includes some information about how `waitpid` and threads interact with one another.*
- What happens if you pass a thread id to `sched_setaffinity`?
    - *It actually works, and informs the OS that the identified thread should only be run on those CPUs included in the provided `cpuset_t`.*

- What are the advantages of requiring that a thread always be assigned to the same CPU?
    - *Each CPU typically has a dedicated L1 cache that stores data fetched by instructions run on that CPU. CPU-specific L1 caches are more likely to retain content relevant to the thread if the same thread keeps coming back instead of being arbitrarily assigned to other CPUs.*

In some situations, the decision to rely on multithreading instead of multiprocessing is dictated solely by whether the code to be run apart from the main thread is available in executable or in library form. But in other situations, we have a choice.

- Why might you prefer multithreading over multiprocessing if both are reasonably good options?
    - *Ease of data exchange and code sharing.*
- Why might you prefer multiprocessing over multithreading if both are reasonably good options?
    - *Protection, privacy, insulation from other processes' errors.*
- What happens if a thread within a larger process calls `fork`?
    - *On Linux, the process space is cloned, but the only surviving thread is the one that called `fork`. In general, you don't want a single thread of many to call `fork` unless you have an exquisite reason to do so.*
- What happens if a thread within a larger process calls `execvp`?
    - *It transforms the entire process to run a new executable. The fact that the pre-`execvp` process involved threading is irrelevant once `execvp` is invoked. It's **that** drastic.*

Solution 4: Threading Short Answer Questions

Here are some more short answer questions about the specifics of threads and the directives that help threads communicate.

- Is `i--` thread safe on a single core machine? Why or why not?
    - *`i--` is not thread safe if it expands to two or more assembly code instructions, as it does in x86_64. For a local variable not already in a register, the generated code might look like this:*

```
mov   (%rbp), %rax
sub   $1, %rax
mov   %rax, (%rbp)
```

    - *If the thread gets swapped off the processor after the first or second of those two lines and another thread is scheduled and changes `i`, the original thread will eventually resume and continue on with stale data.*
    - *Some architectures **are** capable of doing an in-memory decrement (or, more broadly, a memory-memory instruction) in one instruction, so i-- **could** compile to one assembly code instruction on some architectures **if** i is a global variable, and in a single core world where only one thing can happen at a time. But in general, you write code to be portable and architecture-agnostic, so you would need to operate as if i--, even where i is a global, is thread-unsafe.*
- What's the difference between a `mutex` and a `semaphore` with an initial value of 1? Can one be substituted for the other?
    - *The `semaphore` could be used in place of the `mutex`, but not vice versa. First off, the thread that acquires the `lock` on a `mutex` must be the one to unlock it (by C++ specification), whereas one thread can `signal` a `semaphore` while a second thread waits on it. There's also nothing to prevent a `semaphore` initialized to 1 from being signaled to surround an even higher number (like 2, or 2 million), whereas a `mutex` can really only be in one of two states.*
- What is the `lock_guard` class used for, and why is it useful?

- *It's used to layer over some kind of mutex (typically a `mutex`, but possibly some other mutex classes like `timed_mutex` and `recursive_mutex`) so that it's locked when it needs to be locked (via the `lock_guard` constructor) and unlocked when the `lock_guard` goes out of scope (via its destructor). It's useful because it guarantees that a mutex that's locked through a `lock_guard` will be unlocked no matter how the function, method, or enclosing scope ends–in particular, when a function has two or more exits paths (some early returns, some exceptions being thrown, and so forth).*

- What is busy waiting? Is it ever a good idea? Does your answer to the good-idea question depend on the number of CPUs your multithreaded application has access to?
  - *Busy waiting is consuming processor time waiting for some condition to change, as with `while (numAvailableWorkers == 0) {;}`. In a single CPU machine, it makes sense for a process or thread that would otherwise busy wait to yield the processor, since the condition can't possibly change until some other thread gets the processor and makes changes to the variables that are part of the condition. That's what `sigsuspend` (for processes) and `conditional_variable_any::wait` (for threads) are for. In a few scenarios where multithreaded code is running on a machine with multiple CPUs, it's okay to busy wait **if and only** if you know with high probability that another thread is running at the same time (on another CPU) and will invert the condition that's causing the first thread to busy wait.*

- As it turns out, the semaphore's constructor allows a negative number to be passed in, as with `semaphore s(-11)`. Identify a scenario where -11 might be a sensible initial value.
  - *If one thread adds twelve thunks to a `ThreadPool` and then needs to wait on just those twelve threads (and none of the others that happen to be in the `ThreadPool` for other reasons), then you could implement this one of two ways:*
    - *Initialize a `semaphore` to surround a 0, share that `semaphore` with each of the twelve thunks, have each thunk `signal` the `semaphore` once just as it's exiting, and require the parent thread to `wait` on that `semaphore` 12 times, **or***
    - *Initialize a `semaphore` to surround a -11, share that `semaphore` with each of the twelve thunks, have each thunk `signal` the `semaphore` once just as it's exiting, and require the parent thread to `wait` on that `semaphore` just once! `wait` still requires that the encapsulated `count` be positive before it decrements and returns, so this works just as well without prompting the parent to keep to waking make it through each of twelve `wait` calls.*

- What would the implementation of `semaphore::signal(size_t increase = 1)` need to look like if we wanted to allow a `semaphore`'s encapsulated value to be promoted by the `increase` amount? Note that `increase` defaults to 1, so that this version could just replace the standard `semaphore::signal` that's officially exported by the `semaphore` abstraction (the implementation of which is presented at the end of the lab handout).
  - *The key line in signal would need to replace the `++` with a `+=` increase. It would also need to decide whether the condition variable needs to be notified by checking to see if the encapsulated value pre-increment was less than or equal to 0 and the value post-increment is positive. If so, then `notify_all` needs to be invoked.*

- What's the multiprocessing equivalent of the `mutex`?
  - *It's not a perfect parallel, but if I had to choose one, it'd be signal masks. We used signal masks to remove the possibility of interruption by signal, which is the only thing that can otherwise introduce a race condition into a sequential program. (Another answer is something I referenced in an earlier solution, even though I didn't teach it in CS110. Investigate the `flock` function and think about how it could be used to prevent interprocess race conditions on shared memory mapped segments.)*

- What's the multiprocessing equivalent of the `condition_variable_any`?
  - *`sigsuspend` is the equivalent of `conditional_variable_any::wait`, and a signal outside the `sigsuspend` signal mask is the equivalent of `conditional_variable_any::notify_[one|all]`.*

- The `semaphore` implementation we produced in lecture is repeated below. Many students have asked whether the very last line of `semaphore::signal` could have called `notify_one` instead of `notify_all`. Does is matter? Explain.

```
semaphore::semaphore(int value) : value(value) {}

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

void semaphore::signal() {
  lock_guard<mutex> lg(m);
  value++;
  if (value == 1) cv.notify_all();
}
```

- *It definitely matters. Assume signal is implemented in terms of* ***`notify_one`*** *instead of* ***`notify_all`****, and consider a four-thread test program with a single semaphore* ***`s`*** *(initialized to 0), threads 1 and 2 calling* ***`s.wait()`****, threads 3 and 4 calling* ***`s.signal()`****. The following scheduling pattern—one that's very possible—leads to thread 2 blocking forever:*
  - *Threads 1 and 2 each progress and block on* ***`s.wait()`****.*
  - *Thread 3 signals* ***`s`****, which notifies* ***`cv`*** *to wake thread 1 (but not 2) up.*
  - *Thread 4 signals* ***`s`****, without notifying* ***`cv`****.*
  - *Thread 1 passes through* ***`s.wait()`****, but thread 2 is still asleep under* ***`cv`****., even though the semaphore value is 1.*