Lab Handout 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
```

Problem 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.
- You've seen **`dprintf`** in lecture and in the **`assign2`** handout, and it presumably contributed to many **`farm`** implementations. 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.
- 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;
}
```

- 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.
    - Had I accidentally passed in **`&additions`** to the **`sigsuspend`** call instead of **`&existing`**, the farm could have deadlocked. Explain why.
    - Had I accidentally omitted the **`sigaddset`** call and not blocked **`SIGCHLD`**, **`farm`** could have deadlocked. Explain how.
    - 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.

    ```
    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);
    }
  }
```

Problem 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.
- 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.
- 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.
- 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.

- 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.

Problem 3: Threads vs Processes

According to our ever-observant CAs, a good number of students are pondering the pros and cons of threads versus processes. Some have even asked what the pros and cons are. 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?
- What are the advantages of a private address space?
- What are the disadvantages?
- What programming directives have we used in prior assignments and discussion section handouts to circumvent address space privacy?
- In what cases do the processes whose private address spaces are being publicized have any say in the matter?
- 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?
- Can a process be used to execute multiple executables? Restated, can it **`execvp`** twice to run multiple programs?
- Threads are often called lightweight processes. In what sense are they processes? And why the lightweight distinction?
- Threads are often called virtual processes as well. In what sense are threads an example of virtualization?
- 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?

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?
- What happens if you pass a thread id that isn't a process id to **`waitpid`**?
- What happens if you pass a thread id to **`sched_setaffinity`**?
- What are the advantages of requiring that a thread always be assigned to the same CPU?

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?
- Why might you prefer multiprocessing over multithreading if both are reasonably good options?
- What happens if a thread within a larger process calls **`fork`**?
- What happens if a thread within a larger process calls **`execvp`**?

Problem 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?

- What's the difference between a **`mutex`** and a **`semaphore`** with an initial value of 1? Can one be substituted for the other?
- What is the **`lock_guard`** class used for, and why is it useful?
- 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?
- 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.
- 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).
- What's the multiprocessing equivalent of the **`mutex`**?
- What's the multiprocessing equivalent of the **`condition_variable_any`**?
- 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();
}
```