Lab Handout 4: `assign3` Redux and Threads

Students are encouraged to share their ideas in the [#lab4]() Slack channel. SCPD students are welcome to reach out to me or Hemanth directly if they have questions that can't be properly addressed without being physically present for a discussion section. The lab checkoff sheet for all students–both on-campus and off–can be found [right here]().

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 Problem 1, which is a collection of nontrivial short answer questions that verify your understanding of Assignment 3, but Problem 2 is there for you to play with at the end of section or on your own time.

```
poohbear@myth15:~$ hg clone /usr/class/cs110/repos/lab4/shared lab4
poohbear@myth15:~$ cd lab4
poohbear@myth15:~$ make
```

Problem 1: Short Answer Questions

Here are a collection of short answer questions drilling your understanding of subprocess, trace, 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 3 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 `assign3` 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.
- 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.

```
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, all correct
   return 0;
}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, all correct

    return 0;

}
```

- While implementing the **farm** program for **assign3**, 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 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;
}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);
   }
}
```

- Your implementation of `trace` relied on `ptrace`'s ability to read system call arguments from registers via the `PTRACE_PEEKDATA` command. When a system call argument was a C string, you needed to rely on repeated calls to `ptrace` and the `PTRACE_PEEKUSER` option to pull in characters, eight bytes at a time, until a zero byte was included. At that point, the entire `'\0'`-terminated C string could be printed.

  Was this more complicated than need be? If, after all, the argument register contains the base address of a `'\0'`-terminated character array, why can't you just `<<` the `char *` to `cout` and rely on `cout` to print the C string of interest?

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) {
 quicksort(numbers, 0, numbers.size() - 1);
}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 it 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 3 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.