Assignment 3: All Things Multiprocessing

This assignment was designed by Jerry Cain

We've progressed through a good amount of material with multiprocessing, pipes, and interprocess communication, and by the end of Monday's lecture you'll know all about signals and signal handlers. Rather than building one large program, I'd like you to code up a few different things with the idea that you'll learn more by tackling multiple problems and leveraging your understanding of the material in multiple domains. All of these programs should be coded directly within a single repository, which you can get by typing the following:

```
cgregg@myth55$ git clone /usr/class/cs110/repos/assign3/$USER assign3
```

There are four problems in total, and by the end of Wednesday's lecture, you'll be outfitted with all of the material needed to tackle the first three of them without much drama. The final problem—the prime factorization farm—will require material we won't cover until the Monday, but the first three will keep you busy until then. The good news is that you have 10 days to get this assignment up and operational.

Due Date: Sunday, April 28, 2019 at 11:59 p.m.

Short Etude (Op. 25, No. 9): Implementing pipeline in C

Your first task is to implement the **pipeline** function. This **pipeline** function accepts two argument vectors, and assuming both vectors are legit, spawns off twin processes with the added bonus that the standard output of the first is directed to the standard input of the second. Here's the interface you're coding to:

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]);
```

For simplicity, you can assume that all calls to pipeline are well-formed and will work as expected. In other words, **argv1** and **argv2** are each valid, **NULL**-terminated argument vectors, and that **pids** is the base address of an array of length two. Further assume that all calls to **pipe**, **dup2**, **close**, **execvp**, and so forth succeed so that you needn't do any error checking whatsoever. **pipeline** should return without waiting for either of the child processes to finish (i.e. your **pipeline** implementation should **not** call **waitpid** anywhere), and the ids of the daisy-chained processes are dropped into **pids[0]** and **pids[1]**. Also, ensure that the two processes are running in parallel as much as possible, so that **pipeline({"sleep", "10", NULL}, {"sleep", "10", NULL}, pids)** takes about 10 seconds instead of 20.

You should place your implementation of **pipeline** in **pipeline.c**, and you can rely on **pipeline-test.c** and the **pipeline-test** executable it compiles to exercise your implementation. The **pipeline-test.c** test harness I start you off with is small, so you should add many more tests of your own to prove that your **pipeline** is coded to specification.

Note that this first problem is standalone and doesn't contribute to anything else that follows (although the concept of a pipeline will come back in Assignment 4). And as an added bonus, you needn't do any error checking for this exercise (although you may if you think it'll help you arrive at a working solution more quickly).

Short Etude (Op. 25, No. 8): Implementing subprocess in C++

Your next task is to implement an even more flexible **subprocess** than that implemented in lecture. The most important part of the **subprocess.h** interface file is right here:

```
static const int kNotInUse = -1;
struct subprocess_t {
  pid_t pid;
  int supplyfd;
  int ingestfd;
};

/**
 * Function: subprocess
 * --------------------
 * Creates a new process running the executable identified via argv[0].
 *
 * argv: the NULL-terminated argument vector that should be passed
 * to the new process's main function
 * supplyChildInput: true if the parent process would like to pipe content
 * to the new process's stdin, false otherwise
 * ingestChildOutput: true if the parent would like the child's stdout to be
 * pushed to the parent, false otheriwse
 */
subprocess_t subprocess(char *argv[],
                        bool supplyChildInput,
                        bool ingestChildOutput) throw (SubprocessException);
```

Read through the **subprocess.h** documentation to see how this new **subprocess** should work, and place your implementation in **subprocess.cc**. Should any of the system calls needed to implement your **subprocess** routine fail (either in the parent or in the child), you should throw a **SubprocessException** around an actionable error message. Inspect the **subprocess-exceptions.h** file for the full, inlined definition.

Use the test harness supplied by **subprocess-test.cc** to exercise your implementation, and by all means add to the **subprocess-test.cc** file to ensure that your implementation is bulletproof. When looking at **subprocess-test.cc**, you'll also get a little bit of a reminder how **try**/**catch** blocks work. You'll want to add your own tests to **subprocess-test.cc** to ensure that all the (**true**, **true**), (**true**, **false**), (**false**, **true**), and (**false**, **false**) combinations for (**supplyChildInput**, **ingestChildOutput**) all work as expected.

Note that your implementation here is formally C++, since the two larger exercises that follow this one are also to be written in C++, and they each need to link against your **subprocess** implementation without drama. We're switching to C++ pretty much from this problem forward, because C++ provides better support for strings and generics than C does. C++ also provided native support for some threading and concurrency directives we'll be relying on a few weeks, and I'd rather ease you into the language now than do so when we branch into the multithreading topic. Truth be told, your C++ implementation of **subprocess** will look as it would have in pure C, save for the fact that you're throwing C++ exceptions to identify errors.

Your fully functional **subprocess** routine is used by code I wrote for the next exercise (the one requiring you to implement **trace**) and by the starter code I've given you for the final exercise (the one requiring you implement the prime factorization **farm**).

One trick: you'll want to investigate **pipe**'s lesser known sibling, **pipe2**. In particular, you can use **pipe2** to create a pipe just as **pipe** does, while further ensuring that all pipe endpoints are automatically closed when a process calls **execvp** by passing **O_CLOEXEC** as the second argument to **pipe2**. Type **man pipe2** at the prompt and search for **O_CLOEXEC** for more details.

Long Etude (Op. 10, No. 4): Implementing trace in C++

The **trace** utility—like the **strace** utility discussed in this week's discussion section handout—is a diagnostic tool used to monitor the execution of a command. The process running **trace** is called the **tracer**, and the process being monitored is called the **tracee**. The tracer intercepts and records each of the tracee's system calls, and when run in simple mode, prints system call opcodes and raw return values. When run in full mode, **trace** prints the name of each system call, its arguments, and its return value, with additional error information about system calls that fail.

Consider, for example, the following nonsense program (drawn from **simple-test5.cc** in your **assign3** repo).

```
int main(int argc, char *argv[]) {
  write(STDOUT_FILENO, "12345\n", 6);
  int fd = open(__FILE__, O_RDONLY);
  write(fd, "12345\n", 6);
  close(fd);
  read(fd, NULL, 64);
  close(/* bogusfd = */ 1000);
  return 0;
}
```

A simple trace of the command **./simple-test5** might look like this:

```
cgregg@myth55$ ./trace --simple ./simple-test5
syscall(59) = 0
syscall(12) = 14434304
// many lines omitted for brevity
syscall(1) = 123456
syscall(2) = 3
syscall(1) = -9
syscall(3) = 0
syscall(0) = -9
syscall(3) = -9
syscall(231) = <no return>
Program exited normally with status 0
cgregg@myth55$
```

It may look like a bunch of random numbers, but the numbers in parentheses are system call opcodes (59 is the opcode for **execve**, 12 is for **brk**, 1 is for **write**, 2 is for **open**, 3 is for **close**, 0 is for **read**, and 231 is **exit_group**) and the numbers after the equals signs are return values (that 6 is the number of characters just published by **write**, the -9's communicate **write**'s, **read**'s, and **close**'s inability to function when handed closed, incompatible, or otherwise bogus file descriptors, and **exit_group** never returns).

When run in full mode, **trace** publishes much, much more detailed information, as with:

```
cgregg@myth55$ ./trace ./simple-test5
execve("./simple-test5", 0x7ffef5945540, 0x7ffef5945550) = 0
brk(NULL) = 0x1761000
// many lines omitted for brevity
write(1, "12345", 6) = 123456
open("simple-test5.cc", 0, 6) = 3
write(3, "12345", 6) = -1 EBADF (Bad file descriptor)
close(3) = 0
read(3, NULL, 64) = -1 EBADF (Bad file descriptor)
close(1000) = -1 EBADF (Bad file descriptor)
exit_group(0) = <no return>
Program exited normally with status 0
cgregg@myth55$
```

You can see the return values, if negative, are always -1. When the return value is -1, the value of **errno** is printed after that in **#define** constant form (e.g. **EBADF**), followed by a specific error message in parentheses (e.g. **"Bad file descriptor"**). If the return value is nonnegative, then that return value is simply printed and **errno** is ignored as irrelevant.

Overview of trace

trace is more challenging than **pipeline** and **subprocess**, so it requires a detailed specification. To ensure you have everything you need to successfully implement a fully operation **trace**, I provide a good amount of reading with lots of code. Don't be intimidated by the length of this section. I'm just taking care to ensure that everything is spelled out as clearly as possible.

For **trace** to monitor the progress of the tracee and publish the full list of its system calls, it must be able to manipulate the tracee to temporarily freeze as it's entering or exiting each system call. Each time the tracee stalls, the tracer extracts call or return information from the tracee's registers. Only after this information has been extracted is the tracee restarted. By repeatedly doing exactly this (freeze, extract, restart, repeat), the tracer gradually accumulates a series of CPU snapshots as the tracee oscillates in and out of its system calls. When run in simple mode, these CPU snapshots are printed without much translation. When a full digest of system call information is needed, the register values need to be translated to full system call names, strongly typed argument lists (e.g., **int**s, C strings, pointers), and properly interpreted return values.

Unless you've taken systems programming courses more advanced that CS107 or CS107E, you're likely unfamiliar with the idea of writing code to manipulate the execution of another process as we're describing here. The implementation of **trace**, as we'll see, will rely on a special function—itself a system call, as it turns out—called **ptrace**. A quick glance at **ptrace**'s man page provided the following:

*The **ptrace** system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.*

The full prototype of **ptrace** looks like this:

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

The first argument to **ptrace** is always some constant (e.g. **PTRACE_TRACEME**, **PTRACE_SYSCALL**, **PTRACE_PEEKUSER**, etc.), and the choice of constant determines how many additional arguments are needed. The constant supplied as the first argument further dictates what **ptrace** actually does.

The Starter Code

The code we start you out with is far from a feature-complete product. But it does just enough to illustrate how one process can manipulate a second using **ptrace**. When the initial version of **trace.cc** is compiled and invoked to profile **simple-test5**, we see the following:

```
cgregg@myth55$ ./trace --simple ./simple-test5
syscall(59) = 0
cgregg@myth55$
```

The version of **trace** we provide prints information about the first system call. That, of course, needs to change, but the code we give you at least demonstrates how to take and print the first of the many CPU snapshots you'll ultimately need to take.

Go ahead and open the starter version of **trace.cc** included in your **assign3** repo, or view copy of it by clicking right here. We'll expend some energy stepping through the starter code and explaining what it does. By doing so, you'll be in a position to quickly implement an initial version of **trace** that fully supports simple mode, and that'll pave the way for the follow-up work needed to implement the full version.

Let's discuss the first few lines of the starter code, inlined below:

```
int main(int argc, char *argv[]) {
  bool simple = false, rebuild = false;
  int numFlags = processCommandLineFlags(simple, rebuild, argv);
  if (argc - numFlags == 1) {
    cout << "Nothing to trace... exiting." << endl;
    return 0;
  }
```

The starter version ignores whatever **simple** and **rebuild** are set to, even though the code you write will eventually rely on them. The implementation of **processCommandLineFlags** resides in **trace-options.cc**, and that implementation parses just enough of the full command line to figure out how many flags (e.g. **--simple** and **--rebuild**) sit in between **./trace** and the command to be traced. **processCommandLineFlags** accepts **simple** and **rebuild** by reference, updating each independently

depending on what command line flags are supplied. Its return value is captured in a variable called **numFlags**, and that return value shapes how **execvp** is invoked in the code that follows.

The next several lines spawn off a child process that ultimately executes the command of interest:

```
pid_t pid = fork();
if (pid == 0) {
  ptrace(PTRACE_TRACEME);
  raise(SIGSTOP);
  execvp(argv[numFlags + 1], argv + numFlags + 1);
  return 0;
}
```

A new process is created via **fork**, and the child process:

- calls **ptrace(PTRACE_TRACEME)** to inform the OS that it's happy being manipulated by the parent,
- calls **raise(SIGSTOP)** and awaits parental permission to continue, and
- calls **execvp(argv[numFlags + 1], argv + numFlags + 1)** to transform the child process to run the command to be profiled. We're used to seeing **argv[0]** and **argv** as the two arguments, but **argv[0]** is **./trace**. Here, **execvp**'s first argument needs to be the name of the executable to be monitored, and we get to that by hurdling over **./trace** and all of the command line flags. Provided the **execvp** succeeds, the child process effectively reboots itself with a new executable and proceeds through its **main** function, largely unaware that it'll be halted every time it crosses the user-code/kernel-code threshold.

The **return 0** at the end is relevant in the event that **argv[numFlags + 1]** names an executable that either doesn't exist or can't be invoked because of permission issues. We need to ensure the child process ends in the event that **execvp** fails, else its execution will flow into code designed for the tracer, not the tracee.

The tracer circumvents the code specific to the child and executes the following three lines:

```
waitpid(pid, &status, 0);
assert(WIFSTOPPED(status));
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);
```

The **waitpid** call halts the tracer until the child process has granted permission to be traced and self-halted. The **assert** statement confirms that the child self-halted, and the fancy **ptrace** line instructs the operating system to set bit 7 of the signal number—i.e., to deliver **SIGTRAP | 0x80**—whenever a system call trap is executed.

Now consider these lines, which work to advance the tracee to run until it's just about to execute the body of a system call:

```
  while (true) {
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, &status, 0);
    if (WIFSTOPPED(status) && (WSTOPSIG(status) == (SIGTRAP | 0x80))) {
      int syscall = ptrace(PTRACE_PEEKUSER, pid, ORIG_RAX * sizeof(long));
      cout << "syscall(" << syscall << ") = " << flush;
      break;
    }
  }
```

Here's a breakdown of what each line within the **while** loop does:

- **ptrace(PTRACE_SYSCALL, pid, 0, 0)** continues a stopped tracee until it enters a system call (or is otherwise signaled).
- The **waitpid(pid, &status, 0)** call blocks the tracer until the tracee halts.
- If the tracee stops because it's entering a system call, then the **WIFSTOPPED(status)** you will certainly produce **true**. If the child stopped because it's entering a system call, then the tracee would be signaled with **SIGTRAP | 0x80**, as per the above discussion of what **ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD)** does. If either of the two tests **&&**'ed together fails, then the tracee halted for some other reason, and the tracer restarts the tracee as if it never halted.
- Eventually, the tracee stops because it's entering its first system call—that is, the tracee stop just after the system call opcode has been placed in **%rax**, any additional arguments are placed in **%rdi**, **%rsi**, **%rdx**,**%r10**, **%r8**, and **%r9**, as needed, and the software interrupt instruction—**int 0x80**, as per last week's discussion section handout—has been executed.
  The **ptrace(PTRACE_PEEKUSER, pid, ORIG_RAX * sizeof(long))** is another flavor of **ptrace**, and this one extracts the value in **%rax** just as the tracee was forced off the CPU. (For reasons I won't go into, the value of **%rax** is clobbered by the user-to-kernel mode transition, but a pseudo-register **%orig_rax** preserves the value, specifically for system call traces like we're managing here.)
- For the moment, we're ignoring system call parameters, so we're content to extract just the system call opcode from **%orig_rax** and print it within the requisite string published by **cout << "syscall(" << syscall << ") = " << flush**.

Once we've flushed the characters to the console, we break from the for loop and advance on to a second—one designed to publish the system call's raw return value.

```
  while (true) {
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, &status, 0);
    if (WIFSTOPPED(status) && (WSTOPSIG(status) == (SIGTRAP | 0x80))) {
      long retval = ptrace(PTRACE_PEEKUSER, pid, RAX * sizeof(long));
      cout << retval << endl;
      break;
    }
  }
```

The structure of the loop is precisely the same as the first, so line-by-line commentary isn't really necessary. When **waitpid** truly returns because the system call has just exited, we extract the return value from **%rax** (which hasn't been clobbered—so **RAX** is correct, not **ORIG_RAX**), and the raw return value is published to round out the line. Note that **retval** is a **long** instead of an **int**; system call return values can be pointers, so all 64 bits matter sometimes. The system call opcode, however, is always small enough to fit in an **int**, which is why I go with an **int** instead of a **long** in the first **while** loop.

The rest of the starter **trace.cc** file is placeholder and should be removed as you develop a fully operational **trace**. It exists simply to kill the tracee and wait for it to die before allowing the tracer to return. That's why the starter code prints information about the first system call, but none of the others.

```
  kill(pid, SIGKILL);
  waitpid(pid, &status, 0);
  assert(WIFSIGNALED(status));
  return 0;
}
```

Implementing simple trace

The very first thing you'll need to do is to cannibalize the starter version of **trace.cc** to support simple mode, and to print out the full sequence of system call opcodes and return values, as with this:

```
cgregg@myth55:~$ ./trace --simple ./simple-test5
syscall(59) = 0
syscall(12) = 14434304
// many lines omitted for brevity
syscall(1) = 123456
syscall(2) = 3
syscall(1) = -9
syscall(3) = 0
syscall(0) = -9
syscall(3) = -9
syscall(231) = <no return>
Program exited normally with status 0
cgregg@myth55:~$
```

Of course, you have no idea how many system calls a monitored process makes, so you'll need to update the code to repeatedly halt on system call enter, exit, enter, exit, and so forth, until you noticed that the tracee truly exits in the **WIFEXITED** sense.

Words of Wisdom

- As you implement a working version of simple **trace**, you'll likely want to unify the two **while** loops in the starter code to a single function that can be called from two different places. The only reason I replicated the code in the starter **trace.cc** file was to simplify the handout discussion of how it works.
- You may assume that the traced programs are never delivered any non-trap signals and never execute any signal handlers. It wouldn't be that much more difficult to support arbitrary signals and signal handlers, but it'd just be more code that wouldn't add much to the point of the problem.

- System calls don't predictably return the same values with each execution, and one system call's return value may be be passed as an argument to another system call down the line. The **sanitycheck** and the **autograder** are sensitive to all this—at least for the system calls relevant to the all of the test programs I've exposed—and use regular expressions that match and replace parameters and return values that are permitted, and even likely, to vary from test run to test tun.
- Make sure that everything up through an including the equals sign (e.g. **syscall(231) =**) is published and flushed to the console before you allow the tracee to continue. Doing so will help align your output with that of the sample executable, and will make the **sanitycheck** and **autograder** tools happy. This is important when the system calls themselves print the the console (e.g. **write(STDOUT_FILENO, "12345\n", 6)**), so that the system call output interleaves with **trace**'s output in a predictable way.

Implementing full trace

Once you're convinced you have simple mode working, you'll want to tackle full mode. The overall architecture of the program is largely the same. But each time the tracee halts because it's entering or exiting a system call, you print more detailed information. In particular, the first five lines of a simple trace might produce the following:

```
cgregg@myth55:~$ ./trace --simple ./simple-test3
syscall(59) = 0
syscall(12) = 29536256
syscall(21) = -2
syscall(21) = -2
syscall(2) = 3
```

whereas the first five lines of a full trace of the same command might produce this:

```
cgregg@myth55:~$ ./trace ./simple-test3
execve("./simple-test3", 0x7ffdd1531690, 0x7ffdd15316a0) = 0
brk(NULL) = 0x1f90000
access("/etc/ld.so.nohwcap", 0) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", 4) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", 524288, 1) = 3
```

When you need to print full system call information, you need to print the name of the system call instead of its opcode. You also need to print out all of the arguments in a parentheses-delimited, comma-separated list. You'll see that sometimes that arguments are C strings, sometimes they're pointers, and sometimes they're **int**s. Of course, you need to print the return value as you do in simple mode, except that in some circumstances the raw return value is interpreted as a pointer and not an **int**. And when a system call fails, you normalize the return value to a generic -1, but synthesize and print the equivalent **errno** **#define** constant and the corresponding error string. Note that the *real* **errno** global hasn't been set just yet; that comes a few instructions after the system call exits. But you can easily compute what **errno** *will soon be* (and what should be printed by **trace**) by taking the absolute value of the raw return value and using that. The error string is produced by a call to **strerror**, which takes an **errno** value (again, that's the absolute value of the raw return value).

How do you convert system call opcodes to system call names? How does, for instance, 59 become **execve**? The opcode-to- function-call information is housed within the first of three maps populated by a successful

call to **compileSystemCallData**, which is documented in **trace-system-calls.h**. That first map is guaranteed to include every single legitimate opcode as a key. Each key maps to the corresponding system call name, so that 0 maps to **read**, 1 maps to **write**, 2 maps to **open**, 3, maps to **close**, and so forth. You can rely on the information in this map—which you should only construct if you're running in full mode—to convert the 0's to **read**s and the 59's to **execve**'s, etc. (If you're interested, you can look inside the implementation of **compileSystemCallData** within **trace-system-calls.cc**, and you'll see that the map is interest here is built by parsing **/usr/include/x86_64-linux-gnu/asm/unistd_64.h**.)

How do you print argument lists like those contributing to **execve("./simple-test3", 0x7ffdd1531690, 0x7ffdd15316a0)**, **brk(NULL)**, and **access("/etc/ld.so.nohwcap", 0)**? By consulting third of the three maps initialized by a call to **compileSystemCallData**. This map contains system call signature information, and can be used to determine that, say, **execve** takes a C string and two pointers, **access** takes a C string followed by an integer, and that **open** takes a C string and two integers. The number of arguments in the signature determines how many of the registers **%rdi**, **%rsi**, **%rdx**,**%r10**, **%r8**, and **%r9** are relevant. The entries in the map also convey how the data in each of those registers—extracted using **ptrace** and **PTRACE_PEEKUSER**—should be interpreted. All arguments are **int**s, pointers (which should be printed with a leading **0x**, unless it's the zero pointer, in which case it should be printed as **NULL**), or the base address of **'\0'**-terminated character array, aka a C string. Because the C strings reside in the tracee's virtual address space, you'll need to use **ptrace** and **PTRACE_PEEKDATA** to extract the sequence of characters in chunks until you pull in a chunk that includes a **'\0'**. Presented below is a partial implementation to a helper function—one that contributed to my own solution—to rope in a C string from the tracee's virtual address space:

How do you print system call return value information when running in full mode? If the raw return value is

```
static string readString(pid_t pid, unsigned long addr) {
  // addr is a char * read from an argument register via PTRACE_PEEKUSER
  string str;
  // start out empty
  size_t numBytesRead = 0;
  while (true) {
    long ret = ptrace(PTRACE_PEEKDATA, pid, addr + numBytesRead);
    // code that analyzes the sizeof(long) bytes to see if there's a \0 inside
    // code that extends str up to eight bytes in length, but possibly less
    // if ret included a \0 byte
    if (<ret includes a '\0'>) return str;
    numBytesRead += sizeof(long);
  }
}
```

nonnegative, then you print that return value as in **int** for all system calls except **brk** and **mmap**, in which case the raw return values should be interpreted and printed as 64-bit pointers, with a leading **0x**. When the raw return value is negative, you should print the system call's return value as -1. After the -1, you should print the synthesized value of **errno** (e.g. **EBADF** or **ENOENT**), and the error message describing that **errno** , which is produced by a call to **strerror**. The relevant **#define** constant, spelled out as a string, can be retrieved from a map surfaced by a call to a function I wrote for you named **compileSystemCallErrorStrings**, which is documented in **trace-error-constants.h**. A raw return value of -2, for instance, becomes (-1,**"ENOENT"**, **"No such file or directory"**), and a raw return value of -9 (which we see a bunch in the simple trace of **simple-test5**) becomes (-1,**"EBADF"**, **"Bad file descriptor"**). The map surfaced

by **compileSystemCallErrorStrings** links 2 and 9 to **"ENOENT"** and **"EBADF"** , respectively, and **strerror(2)** and **strerror(9)** return **"No such file or directory"** and **"Bad file descriptor"**, respectively. You'll want to read through the documentation within **trace-error-constants.h** to see how to convert **errno** codes to **#define** constant spellings, and you'll want to inspect the man page for **strerror** to see how it works and what you need to **#include** so you can call it without drama.

Once you piece everything together, your **trace** executable should be able to produce the following:

```
cgregg@myth55$ ./trace ./simple-test2
execve("./simple-test2", 0x7ffe088cfb20, 0x7ffe088cfb30) = 0
brk(NULL) = 0x23a1000
access("/etc/ld.so.nohwcap", 0) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", 4) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", 524288, 1) = 3fstat(3, 0x7ffeb60ac940) = 0
mmap(NULL, 0x22ff8, 0x1, 0x2, 0x3, 0) = 0x7fb153093000close(3) = 0
access("/etc/ld.so.nohwcap", 0) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", 524288, 1393262952) = 3
read(3, 0x7ffeb60acaf8, 832) = 832
fstat(3, 0x7ffeb60ac990) = 0
mmap(NULL, 0x1000, 0x3, 0x22, 0xffffffffffffffff, 0) = 0x7fb153092000
mmap(NULL, 0x3c99a0, 0x5, 0x802, 0x3, 0) = 0x7fb152ac7000
mprotect(0x7fb152c87000, 2097152, NULL) = 0
mmap(0x7fb152e87000, 0x6000, 0x3, 0x812, 0x3, 1835008) = 0x7fb152e87000
mmap(0x7fb152e8d000, 0x39a0, 0x3, 0x32, 0xffffffffffffffff, 0) = 0x7fb152e8d000
close(3) = 0
mmap(NULL, 0x1000, 0x3, 0x22, 0xffffffffffffffff, 0) = 0x7fb153091000
mmap(NULL, 0x1000, 0x3, 0x22, 0xffffffffffffffff, 0) = 0x7fb153090000
arch_prctl(<signature-information-missing>) = 0
mprotect(0x7fb152e87000, 16384, 0x1) = 0
mprotect(0x600000, 4096, 0x1) = 0
mprotect(0x7fb1530b6000, 4096, 0x1) = 0
munmap(0x7fb153093000, 143352) = 0
exit_group(0) = <no return>
Program exited normally with status 0
cgregg@myth55$
```

Understanding ptrace

The full list of **ptrace** constants I use for my own solution are presented right here:

- **PTRACE_TRACEME**: Used by the tracee to state its willingness to be manipulated and inspected by its parent. No additional arguments are required, so a simple call to **ptrace(PTRACE_TRACEME)** does the trick.

- **ptrace(PTRACE_SYSCALL, pid, 0, 0)** instructs a stopped tracee to continue executing as usual until it either exits, is signaled, is caught entering a system call, or is caught exiting one. The tracer relies on **waitpid** to block until the tracer stops or exits.
- **ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD)** instructs the kernel to set bit 7 of the wait status to be 1 for all **SIGTRAP**s associated with a tracee's system call.
- **PTRACE_PEEKUSER**: Used by the tracer to inspect and extract the contents of a tracee's register at the time of the call. Only the first three arguments are needed, and any value passed through **data** is ignored. A call to **ptrace(PTRACE_PEEKUSER, pid, RSI * sizeof(long))**, for instance, returns the contents of the tracee's **%rsi** register at the time of the call (provided the supplied **pid** is the process id of the tracee, of course). There are constants for all registers (**RAX**, **RSI**, **RBX**, **RSP**, etc), and the third argument is supposed to be scaled by the size of a word on that processor (which is, by definition, the size of a **long**).
- **PTRACE_PEEKDATA**: Used by the tracer to inspect and extract the word of data residing at the specified location within the tracee's virtual address space. A call to **ptrace(PTRACE_PEEKDATA, pid, 0x7fa59a8b0000)** would return the eight bytes residing at address **0x7fa59a8b0000** within the tracee's virtual address space, and a call to **ptrace(PTRACE_PEEKDATA, pid, ptrace(PTRACE_PEEKUSER, pid, RDI * sizeof(long))** would return the eight bytes residing at another address, which itself resides in **%rdi**). If you know the contents of a register is an address interpretable as the base address of a **'\0'**-terminated C string, you can collect all of the characters of that string by a sequence of **PTRACE_PEEKDATA** calls, as implied by the partial implementation of **readString** I shared above.

More Words of Wisdom

- Take time to digest everything you just read. Make sure you truly understand what every line in the starter code does.
- When run in **full** mode, you'll need access to a collection of maps storing system call function names, system call signatures, and return types. You'll also need to know what **#define** constants should be printed (in text form) when system calls fail. We provided you with several libraries that crawl over the system header files to extract information about system call numbers and **errno** constants, and over a reasonably large number of source files implementing the Linux kernel. You're welcome to peruse the **trace-error-constants.cc** and **trace-system-calls.cc** source files, but you really only need to read through the corresponding interface files to understand what they do for you. The function that crawls over the Linux kernel code base assumes your **subprocess** implementation is solid, so you'll need to make sure you get that working before you can expect the support libraries to work.
- The very first time you run **trace**, you should expect it to take a while to read in all of the prototype information for the linux kernel source tree. All of the prototype information is cached in a local file after that (the cache file will sit in your repo with the name **.trace_signatures.txt**), so trace will fire up much more quickly the second time. Should you want to rebuild the prototype cache for any reason (e.g. you learn your **subprocess** implementation is borked and allowed a bogus **.trace_signatures.txt** file to be generated), you can invoke **trace** with the **--rebuild** flag, as with **./trace --rebuild --simple ./simple-test5**.
- Some system call signature information isn't easily extracted from the Linux source code, so in some cases, the system call signature map populated may not include a system call (e.g. **arch_prctl** comes to mind). If the prototype information is missing from the map, print **<signature-information-missing>** in place of a parameter list, as with **arch_prctl(<signature-information-missing>)**.
- All arguments are either **int**s, **char ***s, or general pointers. The **long** returned by **ptrace** needs to be truncated to an **int**, converted to a C++ **string**, or reinterpreted as a **void *** before printing it.
- This particular program certainly relies on signals, but there are no exposed signal handler functions in my own solution. You'll get practice with signal handlers on the final Assignment 3 problem, and even more practice with Assignment 4.
- My own solution has been compiled into an executable named **trace_soln**, and it's accessible from **./samples**. Feel free to play with it and compare what it generates to what yours does.
- The return value of **trace** is always the return value of the tracee. Running the sample executable on **simple-test3** should make it clear what I'm expecting.

Scherzo No. 4 Implementing farm in C++

Your final challenge is to harness the power of a computer's multiple cores to manage a collection of executables, each running in parallel to contribute its share to a larger result. For the purposes of this problem, we're going to contrive a scenario where the computation of interest—the prime factorization of arbitrarily large numbers—is complex enough that some factorizations take multiple seconds or even minutes to compute. The factorization algorithm itself isn't the focus here, save for the fact that it's potentially time consuming, and that should we need to compute multiple prime factorizations, we should leverage the computing resources of our trusty **myth** cluster to multiprocess and generate output more quickly.

Consider the following Python program called **factor.py**:

```
self_halting = len(sys.argv) > 1 and sys.argv[1] == '--self-halting'
pid = os.getpid()
while True:
   if self_halting: os.kill(pid, signal.SIGSTOP)
   try: num = int(raw_input()) # raw_input blocks, eventually returns a single line from stdin
   except EOFError: break; # raw_input throws an EOFError when EOF is detected
   start = time.time()
   response = factorization(num)
    stop = time.time()
   print ' %s [pid: %d, time: %g seconds]' % (response, pid, stop - start)
```

You really don't need to know Python to understand how it works, because every line of this particular program has a clear C or C++ analog. The primary things I'll point out are:

- Python's **print** operates just like C's **printf** (and it's even process-safe)
- **raw_input** reads and returns a single line of text from standard input, blocking indefinitely until a line is supplied (chomping the **'\n'**) or until end-of-file is detected
- **factorization** is something I wrote; it takes an integer (e.g. **12345678**) and returns the prime factorization (e.g. **12345678 = 2 * 3 * 3 * 47 * 14593**) as a string. You'll see it when you open up **factor.py** in your favorite text editor.
- The **os.kill** line prompts the script to stop itself (but only if the script is invoked with the **'--self-halting'** flag) and wait for it to be restarted via **SIGCONT**

The following should convince you our script does what you'd expect (I'm using **bash**, and I've cleaned up the formatting just a bit, so it looks prettier):

```
cgregg@myth59:~$ printf "1234567\n12345678\n" | ./factor.py
1234567 = 127 * 9721 [pid: 28598, time: 0.0561171 seconds]
12345678 = 2 * 3 * 3 * 47 * 14593 [pid: 28598, time: 0.512921 seconds]
cgregg@myth59:~$ time printf "1234567\n12345678\n123456789\n1234567890\n" | ./factor.py
1234567 = 127 * 9721 [pid: 28601, time: 0.0521989 seconds]
12345678 = 2 * 3 * 3 * 47 * 14593 [pid: 28601, time: 0.517912 seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 28601, time: 5.18094 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 28601, time: 51.763 seconds]
real    0m57.535s
user    0m57.516s
sys    0m0.004s
cgregg@myth59:~$ printf "1001\n10001\n" | ./factor.py —self-halting
cgregg@myth59:~$ kill -CONT %1
1001 = 7 * 11 * 13 [pid: 28625, time: 0.000285149 seconds]
cgregg@myth59:~$ kill -CONT %1
10001 = 73 * 137 [pid: 28625, time: 0.00222802 seconds]
cgregg@myth59:~$ kill -CONT %1
cgregg@myth59:~$ kill -CONT %1
-bash: kill: (28624) - No such process
cgregg@myth59:~$ time printf "123456789\n123456789\n" | ./factor.py
123456789 = 3 * 3 * 3607 * 3803 [pid: 28631, time: 5.1199 seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 28631, time: 5.1183 seconds]
real    0m10.260s
user    0m10.248s
sys    0m0.008s
```

This last test may look silly, but it certainly verifies that one process is performing the same factorization twice, in sequence, so that the overall running time is roughly twice the time it takes to compute the factorization the first time (no caching here, so the second factorization does it all over again).

My **factorization** function runs in O(n) time, so it's very slow for some large inputs. Should you need to compute the prime factorizations of many large numbers, the **factor.py** script would get the job done, but it may take a while. If, however, you're **ssh**'ed into a machine that has multiple processors and/or multiple cores (each of the **myth**s has eight cores), you can write a program that manages several processes running **factor.py** and tracks which processes are idle and which processes are deep in thoughtful number theory.

You're going to write a program—a C++ program called **farm**—that can run on the **myth**s to leverage the fact that you have eight cores at your fingertips. **farm** will spawn several workers—one for each core, each running a self-halting instance of **factor.py**, read an unbounded number of positive integers (one per line, no error checking required), forward each integer on to an idle worker (blocking until one or more workers is ready to read the number), and allow all of the workers to cooperatively publish all prime factorizations to

standard output (without worrying about the order in which they're printed). To illustrate how **farm** should work, check out the following test case:

```
cgregg@myth59:~$ time printf
"1234567890\n1234567890\n1234567890\n1234567890\n1234567890\n1234567890\n123
4567890\n1234567890\n" | ./farm
There are this many CPUs: 8, numbered 0 through 7.
Worker 4245 is set to run on CPU 0.
Worker 4246 is set to run on CPU 1.
Worker 4247 is set to run on CPU 2.
Worker 4248 is set to run on CPU 3.
Worker 4249 is set to run on CPU 4.
Worker 4250 is set to run on CPU 5.
Worker 4251 is set to run on CPU 6.
Worker 4252 is set to run on CPU 7.
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4249, time: 95.5286 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4252, time: 95.5527 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4245, time: 95.5824 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4247, time: 95.5851 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4248, time: 95.6578 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4250, time: 95.6627 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4251, time: 95.6666 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 4246, time: 96.254 seconds]
real    1m36.285s
user    12m42.668s
sys    0m0.128s
```

Note that each of eight processes took about the same amount of time to compute the identical prime factorization, but because each was assigned to a different core, the real (aka perceived) time is basically the time it took to compute the factorization just once. How's that for parallelism!

Note that prime factorizations aren't required to be published in order—that makes this all a little easier—and repeat requests for the same prime factorization are all computed from scratch.

Your **farm.cc** implementation will make use of the following C++ record, global constants, and global variables:

```
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;
```

The **main** function we give you includes stubs for all of the helper functions that decompose it, and that **main** function looks like this:

```
int main(int argc, char *argv[]) {
   signal(SIGCHLD, markWorkersAsAvailable);
   spawnAllWorkers();
   broadcastNumbersToWorkers();
   waitForAllWorkers();
   closeAllWorkers();
   return 0;
}
```

This final problem can be tricky, but it's perfectly manageable provided you follow this road map:

- First implement **spawnAllWorkers**, which spawns a self-halting **factor.py** process for each core and updates the global **workers** vector so that each worker contains the relevant **subprocess_t** allowing **farm.cc** to monitor it and pipe prime numbers to it. You can assign a process to always execute on a particular core by leveraging functionality outlined in the **CPU_SET** and **sched_setaffinity** man pages (i.e. type in **man CPU_SET** to learn about the **cpu_set_t** type, the **CPU_ZERO** and **CPU_SET** macros, and the **sched_setaffinity** function).
- Implement the **markWorkersAsAvailable** handler, which gets invoked to handle all pending **SIGCHLD** signals. Call **waitpid** to surface the pid of the child that recently self-halted, and mark it as available.
- Implement a **getAvailableWorker** helper function, which you'll use to decompose the **broadcastNumbersToWorkers** function in the next step. You should never busy wait. Instead, investigate **sigsuspend** (by typing **man sigsuspend**) as a way of blocking indefinitely until at least one worker is available.
- Flesh out the implementation of **broadcastNumbersToWorkers**. I'm giving you a tiny hint here—that **broadcastNumbersToWorkers** keeps on looping until EOF is detected. You can restart a stopped process by sending it a **SIGCONT** signal.

```
static void broadcastNumbersToWorkers() {
  while (true) {
    string line;
    getline(cin, line);
    if (cin.fail()) break;
    size_t endpos;
    /* long long num = */ stoll(line, &endpos);
    if (endpos != line.size()) break;
    // you shouldn't need all that many lines of additional code
  }
}
```

- Implement **waitForAllWorkers**, which does more or less what it says—it waits for all workers to self-halt and become available.
- Last but not least, implement the **closeAllWorkers** routine to uninstall the **SIGCHLD** handler and restore the default (investigate the **SIG_DFL** constant), coax all workers to exit by sending them EOFs, and then wait for them to all exit.

Your implementation should be exception-safe, and nothing you write should orphan any memory.

Submitting your work

Once you're done, you should run **./tools/sanitycheck** all of your work as you normally would and then run **./tools/submit**.

Grading

Your assignments will be rigorously tested using the tests we expose via sanitycheck plus a whole set of others. I reserve the right to add tests and change point values if during grading I find some features aren't being exercised or properly rewarded by the below distribution, but I'm fairly certain the breakdown presented below will be a very good approximation regardless.

Clean Build (2 points)

Pipeline Tests (20 points)

- Ensure that **pipeline** implementation works with exposed pipeline tests and test strategies discussed in the handout: 6 points
- Ensure that **pipeline** implementation works with custom tests not exposed in the handout or by sanitycheck: 14 points

Trace Tests (40 points)

- Ensure that **trace** works in simple mode on exposed **simple-test1**: 10 points
- Ensure that details associated with simple and full **trace** work as expected (string extraction is one example of something that's independently tested): 10 points
- Ensure that **trace** works on much larger tracee programs not exposed by sanity or by the handout: 20 points

Farm Tests (28 points)

- Ensures that farm properly factors all numbers and distributes numbers across workers in order to maximize parallelism: 25 points
- Ensure that farm works with one edge case scenario that's technically valid input: 3 points