Assignment 2: All Things Multiprocessing

We've progressed through a good amount of material with multiprocessing, pipes, interprocess communication, 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:

```
myth02> cp -r /home/assigments/assign2 ~/
```

There are three problems in total, you'll be outfitted with most of the material needed to tackle them without much drama. The part 4 is additional question, you can choose to complete it.

Due Date: Saturday, November 30, 2019 at 11:59 p.m.

PART 1: 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 daughter processes with the added bonus that the standard output of the first is directed not to the console but rather to the standard input of the second. Here's the interface you're coding to:

```c
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 yourpipelineis 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).

PART 2: 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:

```
/*
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);
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** (the .cc extension means that the code within it is C++) 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. Be sure 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 of the entire assignment (the one requiring you implement the prime factorization farm).

PART 3: Implementing trace in C++

trace is a systems programming tool that helps us profile the execution of a secondary process and present information about all of the system calls—that is, function calls into the kernel—that the secondary executable makes. Specifically, the secondary process makes calls to system calls you certainly know about (e.g. **open, stat, read, write, close, sleep**) and ones you probably don't (**mmap, mprotect, ioctl, recv, getdents**).

The process running trace is called the **tracer**, and the process being profiled is called the **tracee**.

trace can be invoked in two different modes: simple and full (and your implementation needs to support both). When run in simple mode, **trace** publishes information about all of the tracee's system calls via a bare-bones presentation. Only system call codes and raw return values are posted—nothing about arguments, data types, or specific error information. To illustrate, consider the following program (you'll see it in your repo as **simple-test5.cc**):

```
int main(int argc, char *argv[]) {
  write(STDOUT_FILENO, "12345\n", 6);
  int fd = open(__FILE__, O_RDONLY);
  write(fd, "12345\n", 5); close(fd);
  read(fd, NULL, 64);
  close(/* bogusfd = */ 1000);
  return 0;
}
```

Assuming the above has been compiled into an executable called **simple-test5**, its bare-bones trace might look like this:

```
myth5> ./trace --simple ./simple-test5
syscall(59) = 0
syscall(12) = 35303424
// many lines omitted for brevity syscall(1) = 12345
6
syscall(1) = 6
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 myth5>
```

There are a lot of magic numbers there, but I promise that the numbers in parentheses are system call numbers (**59 is for execve, 0 is for read, 1 is for write, 2 is for open, 3 is for close, 12 is for brk, 231 is for exit_group**) and the numbers after the equals signs are return values (that 6 is the number of characters published by **write**, the -9's express **write's**, **read's**, and **close's** inability to function when handed closed, incompatible, or otherwise bogus file descriptors, and **exit_group** never returns (gulp!)).

When run in full mode (i.e. without the --simple flag), trace pulls out all of the stops and prints oodles of information about each of calls:

```
myth5> ./trace ./simple-test5
execve("./simple-test5", 0x7ffeb3d1a460, 0x7ffeb3d1a470) = 0
brk(NULL) = 0xbbcffe04
// many lines omitted for brevity
write(1, "12345", 6) = 12345
6
open("simple-test5.cc", 0, 6) = 3
write(3, "12345", 5) = -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
```

```
myth5>
```

You can see the return values, if negative, are always -1, but that some **errno** value is printed after that in **#de ! ne** constant form, followed by a specific error message. It turns out that **#de ! ne** constant is always mapped to the absolute value of the system call's return value (so in this case, 2), and you can easily get the more detailed error message by passing the constant (in this case 2) to the strerror function.

PART 4: Implementing farm in C++ (Additional Question)

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 Unix cluster machines 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_inputreads 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 here, where the time builtin clocks the entire pipeline):

```
real 0m16.806s
user 0m16.793s
sys 0m0.008smyth02> printf "1234567\n12345678\n" | ./factor.py
1234567 = 127 * 9721 [pid: 14391, time: 0.100041 seconds]
12345678 = 2 * 3 * 3 * 47 * 14593 [pid: 14391, time: 1.03848
seconds]
myth02> time printf "1234567\n12345678\n123456789\n1234567890\n" |
./factor.py
1234567 = 127 * 9721 [pid: 14440, time: 0.108153 seconds]
12345678 = 2 * 3 * 3 * 47 * 14593 [pid: 14440, time: 1.04659
seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 14440, time: 10.1917 seconds]
1234567890 = 2 * 3 * 3 * 5 * 3607 * 3803 [pid: 14440, time: 102.537
seconds]
real 1m53.911s
user 1m53.929s
sys 0m0.033s
myth02> printf "1001\n10001\n" | ./factor.py --self-halting
myth02> kill -CONT %1
1001 = 7 * 11 * 13 [pid: 15973, time: 0.000144005 seconds]
myth02> kill -CONT %1
10001 = 73 * 137 [pid: 15973, time: 0.000889063 seconds]
myth02> kill -CONT %1
```

```
myth02> kill -CONT %1
kill: No such job.
myth02> time printf "123456789\n123456789\n" | ./factor.py
123456789 = 3 * 3 * 3607 * 3803 [pid: 2143, time: 8.39598 seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 2143, time: 8.39575 seconds]
real 0m16.806s
user 0m16.793s
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 (the virtual machine has eight!), 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. To illustrate how farm should work, check out the following test case:

```
real 0m10.667s user 0m41.197s
sys 0m0.099srye01> time printf
"123456789\n123456789\n123456789\n123456789\n" | ./farm
There are this many CPUs: 8, numbered 0 through 7. Worker 25528 is
set to run on CPU 0.
Worker 25529 is set to run on CPU 1.
Worker 25530 is set to run on CPU 2.
Worker 25531 is set to run on CPU 3.
Worker 25532 is set to run on CPU 4.
Worker 25533 is set to run on CPU 5.
Worker 25534 is set to run on CPU 6.
Worker 25535 is set to run on CPU 7.
123456789 = 3 * 3 * 3607 * 3803 [pid: 25528, time: 10.2493 seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 25531, time: 10.3282 seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 25530, time: 10.4229 seconds]
123456789 = 3 * 3 * 3607 * 3803 [pid: 25529, time: 10.6165 seconds]
```

```
real 0m10.667s
user 0m41.197s
sys 0m0.099s
```

Note that each of four processes took about the same amount of time to compute the identical prime factorizations, but because each of the four processes was assigned to different cores, the real (aka perceived) user time was under 11 seconds. Note that prime factorizations aren't required to be published in order, and repeat requests for the same prime factorization are all computed from scratch, without any caching.

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

```
static const size_t kNumCPUs = sysconf(_SC_NPROCESSORS_ONLN);
static vector<worker> workers(kNumCPUs); // space for kNumCPUs,
zero-arg constructed workers
static size_t numWorkersAvailable = 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); // space for kNumCPUs,
zero-arg constructed workers
static size_t numWorkersAvailable = 0;
```

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:

- Advance on tospawnAllWorkers, which spawns a self-haltingfactor.pyprocess 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 whenever one of the child processes self-halts (prompting the kernel to SIGCHLD signal the parent). 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 known to be available.
- Flesh out the implementation of broadcastNumbersToWorkers. I'm giving you a tiny hint here—that broadcastNumbersToWorkers keeps on looping until either EOF is detected (or until the farm user messes up and deviated from the required input format). Investigate the SIGCONT signal as the means to restart another stopped process.

```
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
  }
}
```

- ImplementwaitForAllWorkers, 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), cajole all child processes to exit by sending them EOFs, and then wait for them to all actually exit.

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