Assignment 4: stsh — stanford shell

Kudos to Randal Bryant and Dave O'Hallaron of Carnegie Mellon for assignment inspiration and for parts of this handout. Huge thanks to Truman Cranor for writing the command line parser using tools you'll learn in CS143, which you should all take someday because it's amazing.
You've all been using shells to drive your conversations with UNIX systems since the first day you logged into a **`myth`**. It's high time we uncover the shell's magic by leveraging the **`simplesh`**s we built in lecture together and extending the final version to support process control, job lists, signals, pipelines, and I/O redirection—all while managing the interprocess concurrency problems that make a shell's implementation a genuinely advanced systems programming project. There's lots of neat code to write, and with your smarts and my love to guide you, I'm confident you can pull it off.

Due Date: Sunday, November 3rd at 11:59 p.m.

All coding should be done on a **`myth`** cluster machine, as that's where we'll be testing all **`assign4`** submissions. You should clone the master git repository we've set up for you by typing:

```
myth62$ git clone /usr/class/cs110/repos/assign4/$USER assign4
```

Doing so will create an **`assign4`** directory within your AFS space, and you can descend into that **`assign4`** directory and code there. There's a working **`sanitycheck`** for this assignment, and your repo includes a soft link to a fully functional solution. When in doubt about how something should work, just run my solution (which can be found at **`./samples/stsh_soln`**) to see what it does and imitate the solution.

Builtin `stsh` Commands

Your Assignment 4 shell needs to support a collection of builtin commands that should execute without creating any new processes. These builtins are:

- **`quit`**, which exits the shell and abandons any jobs that were still running. If there are any extra arguments after the **`quit`**, just can just ignore them.
- **`exit`**, which does the same thing as **`quit`**. Extraneous arguments? Just ignore them.
- **`fg`**, which prompts a stopped job to continue in the foreground or brings a running background job into the foreground. **`fg`** takes a single job number (e.g. **`fg 3`**). If the provided argument isn't a number, there's no such job with that number, or the number of arguments isn't correct, then throw an **`STSHException`** around an actionable error message and allow your shell to carry on as if you never typed anything.
- **`bg`**, which prompts a stopped job to continue in the background. **`bg`** takes a single job number (e.g. **`bg 3`**). If the provided argument isn't a number, there's no such job with that number, or the number of arguments isn't correct, then throw an **`STSHException`** around an actionable error message and allow your shell to carry on as if you never typed anything.

- **`slay`**, which is used to terminate a single process (which may have many sibling processes as part of a larger pipeline). **`slay`** takes either one or two numeric arguments. If only one number is provided, it's assumed to be the pid of the process to be killed. If two numbers are provided, the first number is assumed to be the job number, and the second is assumed to be a process index within the job. So, **`slay 12345`** would terminate (in the **`SIGKILL`** sense of terminate) the process with pid 12345. **`slay 2 0`** would terminate the first process in the pipeline making up job 2. **`slay 13 8`** would terminate the 9th process in the pipeline of processes making up job 13. If there are any issues (e.g., the arguments aren't numbers, or they are numbers but identify some nonexistent process, or the argument count is incorrect), then throw an **`STSHException`** around an actionable error message and allow your shell to carry on as if you never typed anything.
- **`halt`**, which has nearly the same story as **`slay`**, except that its one or two arguments identify a single process that should be halted (but not terminated) if it isn't already stopped. If it's already stopped, then don't do anything and just return.
- **`cont`**, which has the same story as **`slay`** and **`halt`**, except that its one or two arguments identify a single process that should continue if it isn't already running. If it's already running, then don't do anything and just return. When you prompt a single process to continue, you're asking that it do so in the background.
- **`jobs`**, which prints the job list to the console. If there are any additional arguments, then just ignore them.

`quit`, **`exit`**, and **`jobs`** are already implemented for you. You're responsible for implementing the others and ensuring the global job list is appropriately updated.

Getting Started

Inspect the `stsh.cc` file we give you. This is the only file you should need to modify. The core `main` function you're provided looks like this:

```
int main(int argc, char *argv[]) {
  pid_t stshpid = getpid();
  installSignalHandlers();
  rlinit(argc, argv); // configures stsh-readline library so readline works
properly
  while (true) {
    string line;
    if (!readline(line)) break;
    if (line.empty()) continue;
    try {
      pipeline p(line);
      bool builtin = handleBuiltin(p);
      if (!builtin) createJob(p); // createJob is initially defined as a
wrapper around cout << p;
    } catch (const STSHException& e) {
      cerr << e.what() << endl;
      if (getpid() != stshpid) exit(0); // if exception is thrown from child
process, kill it
    }
  }

  return 0;
}
```

The general idiom here is the same used in the **`simplesh`** examples from lecture, except that this version is in C++, and it'll support many more features. The **`readline`** function prompts the user to enter a command and a pipeline record is constructed around it. **`readline`** and **`pipeline`** (which is a different from the **`pipeline`** function you implemented for Assignment 3) are implemented via a suite of files in the **`stsh-parser`** subdirectory, and for the most part you can ignore those implementations. You should, however, be familiar with the type definitions of the **`command`** and **`pipeline`** types, though, and they are right here:

```
const size_t kMaxCommandLength = 32;
const size_t kMaxArguments = 32;
struct command {
  char command[kMaxCommandLength + 1]; // '\0' terminated
  char *tokens[kMaxArguments + 1]; // NULL-terminated array, C strings are
all '\0' terminated
};

struct pipeline {
  std::string input;   // empty if no input redirection file to first command
  std::string output;  // empty if no output redirection file from last command
  std::vector<command> commands;
  bool background;

  pipeline(const std::string& str);
  ~pipeline();
};
```

Check out what the initial version of **`stsh`** is capable of before you add any new code.

Setting Milestones

The best approach to implementing anything this complex is to invent a collection of milestones that advance you toward your final goal. Never introduce more than a few lines of code before compiling and confirming that the lines you added do what you expect. I repeat: **Never introduce more than a few lines of code before compiling, testing, and confirming that the additional lines do what you expect.** View everything you add as a slight perturbation to a working system that slowly evolves into the final product. Try to understand every single line you add, why it's needed, and why it belongs where you put it.

Here is a sequence of milestones I'd like you to work through in order to get started:

- Descend into the `stsh-parser` directory, read through the `stsh-readline.h` and `stsh-parse.h` header files for data type definitions and function/method prototypes, type `make`, and play with the `stsh-parse-test` to gain a sense of what `readline` and the `pipeline` constructor do for you. In general, the `readline` function is like `getline`, except that you can use your up and down arrows to scroll through your history of inputs (neat!). The `pipeline` record defines a bunch of fields that store all of the various commands that chain together to form a pipeline. For example, the text `cat < /usr/include/stdio.h | wc > output.txt` would be split into two `command`s—one for the `cat` and a second for the `wc`—and populate the `vector<command>` in the pipeline with information about each of them. The `input` and `output` fields would each be nonempty, and the background field would be `false`.
- Get a pipeline of just one command (e.g. `stsh> sleep 5`) to run in the foreground until it's finished. Rely on a call to `waitpid` to stall `stsh` until the foreground job finishes. Ignore the job list, ignore the `SIGCHLD` handler, don't worry about background jobs, pipelining, or redirection. Don't worry about programs like `emacs` just yet. Focus on these executables instead: `ls`, `date`, `sleep`, as their execution is well understood and predictable.
- Establish the process group ID of the job to be the PID of the process by investigating the `setpgid` system call. When you run `stsh` from the standard Unix shell, note that `stsh` is running in the foreground. If your shell then creates a child process, by default that child will also be a member of the foreground process group as well, and you don't want that. Since typing ctrl-c sends a `SIGINT` to every process in the foreground group, typing ctrl-c will send a `SIGINT` to your shell, as well as to every process that your shell created, and you don't want that. The notion of a group ID isn't all that important yet, because at the moment, a process group consists of only one process. But we'll eventually be dealing with jobs comprised of many processes, and we'll want a single process group id to represent all of them.
- Read through `stsh-job-list.h`, `stsh-job.h`, and `stsh-process.h` to learn how to add a new foreground job to the job list, and how to add a process to that job. Add code that does exactly that to the `stsh.cc` file, right after you successfully fork off a new process. After your `waitpid` call returns, remove the job from the job list. If it helps, inline `cout << joblist;` lines in strategically chosen locations to confirm your new job is being added after `fork` and being removed after `waitpid`.
- Implement the `SIGCHLD` handler to reap the resources of a foreground process after it exits, and suspend `stsh`'s main thread of execution using a `sigset_t`, `sigprocmask`, `sigsuspend`, and related functions until an examination of the job list proves that the

foreground job is no longer the foreground job. Your call to **`waitpid`** should be moved into the **`SIGCHLD`** handler, and that should be the only place in your entire solution—even the final one you submit—with a **`waitpid`** call.

- Install functions to activate ctrl-z and ctrl-c on your keyboard to stop and kill the foreground process instead of the shell itself. If, for instance, a **`sleep 500`** is running as the foreground, you may want to kill the process by pressing ctrl-c. When you press ctrl-c, the OS sends a **`SIGINT`** to your shell, which unless handled will terminate the shell itself. If, however, you install a function to handle **`SIGINT`**, then that handler can (and should) forward the **`SIGINT`** on to the foreground job, should one exist. The story for ctrl-z is similar, except ctrl-z prompts the OS to send your shell a **`SIGTSTP`**. If you install a custom handler to intercept a **`SIGTSTP`**, you can forward the **`SIGTSTP`** on to the foreground job, should one exist. In these cases, the **`kill`** function is your friend and will contribute to your implementation of both of these handlers.
- Implement the **`fg`** builtin, which takes a stopped process—stopped presumably because it was running in the foreground at the time you pressed ctrl-z—and prompts it to continue, or it takes a process running in the background and brings it into the foreground. The **`fg`** builtin takes job number, translates that job number to a process group ID, and, if necessary, forwards a **`SIGCONT`** on to the process group via a call to **`kill(-groupID, SIGCONT)`**. Right now, process groups consist of just one process, but once you start to support pipelines (e.g. **`cat words.txt | sort | uniq | wc -l`**), you'll want **`fg`** to bring the entire job into the foreground, and if all relevant processes are part of the same process group, you can achieve this with a single **`kill`** call. Of course, if the argument passed to **`fg`** isn't a number, or it is but it doesn't identify a real job, then you should throw an **`STSHException`** that's wrapped around a clear error message saying so.
- Update the **`SIGCHLD`** handler to detect state changes in all processes under **`stsh`**'s jurisdiction. Processes can exit gracefully, exit abnormally because of some signal, or be terminated by ctrl-c. Processes can halt because of a signal or a ctrl-z. And processes can continue because of a signal or an **`fg`** builtin. The job list needs to remain in sync with the state of your shell's world, and **`waitpid`** is the perfect function to tell you about changes. You're already familiar with **`WNOHANG`**, **`WUNTRACED`**, **`WIFEXITED`**, **`WEXITSTATUS`**, etc. Read through **`waitpid`**'s man page to get word on some Linux-specific flags and macros that tell you about processes that have started executing again. Buzzwords include **`WCONTINUED`** and **`WIFCONTINUED`**, so read up on those.
- Add support for background jobs. The **`pipeline`** constructor already searches for trailing &'s and records whether or not the pipeline should be run in the background. If it does, then you still add information about the job to the job list, but you immediately come back to the shell prompt without waiting.
- Add support for **`slay`**, **`halt`**, **`cont`** (the process-oriented commands) and **`bg`** (which prompts all processes in a single job to continue in the background), and use some of the sample user programs I include in your repo (**`int`**, **`fpe`**, **`tstp`**, **`spin`**, **`split`**) to test all of these.

The following are additional milestones you need to hit on your way to a fully functional **`stsh`**.

Each of these bullet points represents something larger.

- Add support for foreground jobs whose leading process (e.g. **`cat`**, **`more`**, **`emacs`**, **`vi`**, and other executables requiring elaborate control of the console) requires control of the terminal. You should investigate the **`tcsetpgrp`** function as a way of transferring terminal control to a process group, and update your solution to always call it, even if it fails. If **`tcsetpgrp(STDIN_FILENO, pgid)`** succeeds, then it'll return 0. If it fails with a return value of -1 but it sets **`errno`** to **`ENOTTY`**, that just means that your **`stsh`** instance didn't have control of the terminal or the authority to donate it to another process group. (This will happen if your shell is being driven by **`stsh-driver`**) If it fails with a different **`errno`** value, then that's a more serious problem that should be identified via an **`STSHException`**. If **`stsh`** succeeds to transferring control to the foreground process group, then **`stsh`** should take control back when that group falls out of the foreground (perhaps because it exits, or it stops, or something else).
- Add support for pipelines consisting of two processes (i.e. binary pipelines, e.g **`cat /usr/include/stdio.h | wc`**). Make sure that the standard output of the first is piped to the standard input of the second, and that each of the two processes are part of the same process group, using a process group ID that is identical to the pid of the leading process. The **`pipeline`** function from Assignment 3 (which, again, is different from the **`pipeline`** type here) was a paying-it-forward nod to the basic pipeline functionality needed right here. You needn't go nuts on the error checking: You can assume that all system calls succeed, with the exception of **`execvp`**, which may fail because of user error (misspelled executable name, file isn't an executable, lack of permissions, etc.). You might want to include more error checking if it helps you triage bugs, assert the truth of certain expectations during execution, and arrive at a working product more quickly, but do all that because it's good for you and not because you're trying to make us happy. (Hint: the **`conduit`** user program I dropped in your repo starts to become useful as soon as you deal with nontrivial pipelines. Try typing **`echo 12345 | ./conduit --delay 1`** in the standard shell to see what happens, and try to replicate the behavior in **`stsh`**.)
- Once you get your head around pipelines of one and two processes, work on getting arbitrarily long pipeline chains to do the right thing. So, if the user types in **`echo 12345 | ./conduit --delay 1 | ./conduit | ./conduit`**, four processes are created, each with their own pid, and all in a process group whose process group id is that of the leading process (in this case, the one running echo). **`echo`**'s standard out feeds the standard in of the first **`conduit`**, whose standard out feeds into the standard in of the second **`conduit`**, which pipes its output to the standard input of the last **`conduit`**, which at last prints to the console. I'm calling this out separately from the binary pipeline discussed in the previous bullet point, because the code needed to realize pipelines of three or more doesn't have as much in common with the binary pipeline code as you might think. Yes, there are **`pipe`**, **`setpgid`**, **`dup2`**, **`close`**, and **`execvp`** calls, but figuring out how to get one of the inner processes to redefine what **`STDIN_FILENO`** and **`STDOUT_FILENO`** are connected to is very tricky, and this trickiness doesn't present itself in the binary pipeline.
- Finally, add support for input and output redirection via < and > (e.g. **`cat < /usr/include/stdio.h | wc > output.txt`**). The names of input and output redirection files are surfaced by the **`pipeline`** constructor, and if there is a nonempty

string in the input and/or output fields of the pipeline record, that's your signal that input redirection, output redirection, or both are needed. If the file you're writing to doesn't exist, create it, and go with 0644 (with the leading zero) as the octal constant to establish the `rw-r-r--` permission. If the output file you're redirecting to already exists, then truncate it using the `O_TRUNC` flag. Note that input redirection always impacts where the leading process draws its input from and that output redirection influences where the caboose process publishes its output. Sometimes those two processes are the same, and sometimes they are different. Type `man 2 open` for the full skinny on the `open` system call and a reminder of what flags can be bitwise-OR'ed together for the second argument.

Shell Driver

The `stsh-driver` program (there's a copy of it in your repo) executes `stsh` as a child process, sends it commands and signals as directed by a *trace file*, and allows the shell to print to standard output and error as it normally would. The `stsh` process is driven by the `stsh-driver`, which is why we call `stsh-driver` a driver.

Go ahead and type ./`stsh-driver -h` to learn how to use it:

```
Usage: ./stsh-driver [-hv] -t <trace> -s <shell> [-a <args>]
Options:
  -h         Print this message
  -v         Output more information
  -t <trace> Trace file
  -s <shell> Version of stsh to test
  -a <args>  Arguments to pass through to stsh implementation
myth62$
```

We've also provided several trace files that you can feed to the driver to test your `stsh`. If you look drill into your repo's `samples` symlink, you'll arrive at `/usr/class/cs110/samples/assign4`, which includes not only a copy of my own `stsh` solution, but also a directory of shared trace files called `scripts`. Within `scripts` , you'll see `simple`, `intermediate`, and `advanced` subdirectories, each of which contains one or more trace files you can use for testing.
Run the shell driver on your own shell using trace file `bg-spin.txt` by typing this:

```
myth62$ ./stsh-driver -t ./samples/scripts/simple/bg-spin.txt -s ./
stsh -a "--suppress-prompt --no-history"
```

(the `-a "--suppress-prompt --no-history"` argument tells `stsh` to not emit a prompt or to use the fancy `readline` history stuff, since it confused the `sanitycheck` and `autograder` stuff).
Similarly, to compare your results with those generated by my own solution, you can run the driver on `./stsh_soln` shell by typing:

```
myth62$ ./stsh-driver -t ./samples/scripts/simple/bg-spin.txt -s ./
samples/stsh_soln -a "--suppress-prompt -no-history"
```

The neat thing about the trace files is that they generate the same output you would have gotten had you run your shell interactively (except for an initial comment identifying the output as something generated via **`stsh-driver`**). For example:

```
myth62$ ./stsh-driver -t ./samples/scripts/advanced/simple-
pipeline-1.txt -s ./samples/stsh_soln -a "--suppress-prompt --no-
history"
# Trace: simple-pipeline-1
# ------------------------
# Exercises support for pipes via a foreground pipeline with
# just two processes.
stsh> /bin/echo abc | ./conduit --count 3
aaabbbcccdddeeefffgggghhhiiijjj
```

The process ID's listed as part of a trace's output will be different from run to run, but otherwise your output should be exactly the same as that generated by my solution.

Tips and Tidbits

- Read every word of Chapters 1 and 2 in your B&O reader (or Chapters 8 and 10 in the full textbook).
- Your implementation should be in C++ unless there's no way to avoid it. By that, I mean you should use **`C++`** strings unless you interface with a system call that requires C strings, use **`cout`** instead of **`printf`**, and so forth. Understand that when you redefine where **`STDOUT_FILENO`** directs its text, that impacts where **`printf`** (which you're not using) and **`cout`** (which you are) place that text.
- We have reasonably low expectations on error checking for this assignment, but we do have some. I want you to focus on how to leverage system directives to get a fully functional shell working, but I don't want you to examine the return value of every system call when it's more or less impossible for them to fail. In general, your error checking should guard against ***user*** error—attempts to invoke a nonexistent executable, providing out-of-range arguments to command-line built-ins, and so forth. In some cases—and I point out those cases in this handout—you **do** need to check the return value of a system call or two, because sometimes system call "failure" (the air quotes are intentional) isn't a true failure. You've seen situations where **`waitpid`** returns -1 even though everything was fine, and that happens with a few other system calls as well.
- All unused file descriptors should be closed.
- When creating a **`pipeline`**—whether it consists of a single process, two processes, or many, many processes—you need to ensure that all of the pipeline processes are in the same process group, but in a different process group than the **`stsh`** instance is. To support this, you should investigate **`setpgid`** to see how all of the sibling processes in a pipeline can be added to a new process group whose id is the same as the pid of the leading process. So, if a pipeline consists of four processes with pids 4004, 4005, 4007, and 4008, they would all be placed in a process group with an ID of 4004. By doing this, you can send a signal to every process in a group using the **`kill`** function, where the first argument is the negative of the process group id (e.g. **`kill(-4004, SIGTSTP)`**) .

In order to avoid some race conditions, you should call **`setpgid`** in the parent and in each of the children. Why does the parent need to call it? Because it needs to ensure the process group exists before it advances on to add other processes in the pipeline to the same group. Why do child processes need to call it? Because if the child relies on the parent to do it, the child may **`execvp`** (and invalidate its own pid as a valid **`setpgid`** argument) before the parent gets around to it. Race conditions: deep stuff.

- You do not need to support pipelines or redirection for any of the builtins. In principle, you should be able to, say, redirect the output of **`jobs`** to a file, or pipe the output to another process, but you don't need to worry about this for Assignment 4. Our tests will never mix builtins with pipes and redirection.
- Every time your **`SIGCHLD`** handler learns of a stopped or terminated process, it's possible the surrounding job is impacted. Investigate the **`STSHJobList::synchronize(STSHJob& job)`** method, which scrutinizes the supplied job to see if it should be removed from the job list or if it should demote itself to the background.
- We've defined one global variable for you (the **`STSHJobList`** called **`joblist`**). You may not declare any others unless they are constants.
- Make sure pipelines of multiple commands work even when the processes spawned on their behalf ignore standard in and standard out, e.g. **`sleep 10 | sleep 10 | sleep 10 | sleep 10 > empty-output.txt`** should run just fine—the entire pipeline lasts for about 10 seconds, and **`empty-output.txt`** is created or truncated and ends up being 0 bytes.
- When an entire pipeline is run as a background job, make sure you print out a job summary that's consistent with the following output:

```
stsh> sleep 10 | sleep 10 | sleep 10 &
[1] 27684 27685 27686
```

- We will be testing your submission against many more traces than the ones we provide, so be sure to define lots and lots of additional traces to verify your shell is unbreakable!

Submitting your work

Once you're done, you should test all of your work as you normally would and then run the infamous submissions script by typing **`./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, but I'm fairly certain the breakdown presented below will be a very good approximation regardless.

Basic Tests (47 points)

- Clean Build: 2 points
- Ensure that most basic of builtins (`quit`, `exit`, `jobs`) work properly: 15 points
- Ensure that basic foreground and background processes work: 10 points
- Ensure that `SIGINT` and `SIGTSTP` handlers work on single process pipelines: 20 points

Intermediate Tests (50 points)

- Ensure that `bg` and `fg` builtins are properly supported: 15 points
- Ensure that `SIGINT` and `SIGTSTP` signals play well with stopped processes, restarted processes, and `fg` and `bg` builtins: 30 points
- Confirm error checking to guard against user error (e.g. mistyped commands, mistyped builtins, etc): 5 points (the one test is exposed via sanity)

Advanced Tests (70 points)

- Ensure that pipelines with two processes work well: 15 points
- Ensure that pipelines with three or more processes work well: 25 points
- Ensure redirection works for pipelines of just 1 process: 15 points
- Ensure redirection works for pipelines with two or more processes: 15 points