SMASH (SMAll SHell)

Daniel Calabria

smash is designed to be a small and simple shell.

Compiling

`make clean all` will build smash. Upon successful completion, the final binary will be located under `bin/smash`.

The following targets are offered by the Makefile:

- `clean` : Cleans build environment.
- `all` : Compiles smash with default options.
- `debug` : Compiles smash with heavy debugging output, by `-DDEBUG`.
- `tests` : Executes test scripts, located under `tests/`.
- `ec` : Builds with `-DEXTRA_CREDIT`, enabling additional functionality.
- `ectests` : Executes test scripts associated with `ec`, located under `tests/`.

Usage

Usage: `smash [-d] [FILE]`

Options

- `-d` : Enables debugging output. This will print a message whenever a child has begun executing, and a message when a child is detected to have finished (either normally terminating, or by receipt of a signal).
- `FILE` : if a file is specified on the command line, smash will execute non-interactively, processing the given file as input.

Using smash

After launching the shell, you will be greeted by a `smash>` prompt. You can then attempt to execute programs and processes as you normally would, in addition to also executing a selection of builtin commands.

Foreground and Background Jobs

Jobs in smash will run either in the foreground or background. A job can be specified to be a background job on the command line by the inclusion of the `&` character at the end of the input line. By default, programs will be launched in the foreground unless specified to be a background job.

Pressing C-z will suspend a job running in the foreground, such that you may once again interact with the shell and enter commands. The process is placed into a stopped state until the delivery of a `SIGCONT` signal.

Pressing C-c will interrupt a job running in the foreground, such that the job recieves a `SIGINT` and processes it as normal.

Builtin Commands

The shell has support for a selection of builtin commands.

- `echo [args...]` : Outputs the argument list to standard output.
- `cd [path]` : Changes to the specified directory.
- `pwd` : Outputs the current working directory of the shell.
- `jobs` : Outputs a list of currently running and recently ended jobs handled by the shell. If a job has completed, either the exit code it returned upon completion or the signal number which caused its termination is reported along with it.
- `fg [jobid]` : Places the specified job in the foreground, and resumes execution of it if it not already running.
- `bg [jobid]` : Places the specified job in the background, and resumes execution of it if it not already running.
- `kill [-N] [jobid]` : Sends the signal number `N` to the process group with the specified jobid.
- `C-d` (^D, Ctrl-D) keypress : This will send an EOF character to the input line, causing the shell to terminate if it is on an empty line.

In addition, any line beginning with a `#` is ignored, and any text encountered after a `#` is encountered in a line is discarded. Effectively, anything after a `#` in a line is treated as a comment and not evaluted.

Environment Variables

smash has basic support for the use of environment variables. Any command argument which begins with a `$` is treates as an environment variable, and an attempt is made to `getenv(3)` on the variable name. If no such variable is found, then an empty string (not `NULL`) is used in its place, instead.

It is possible to use these environment variables in the evaluation and execution of both builtin commands (`echo $HOME`) and actual programs (`ls $HOME`).

smash also supports the reporting of the exit code of the last completed foreground process. This is bound to the `$?` variable, and is updated whenever a foreground job exits. Note that this does not update the variable if a job is terminated via a signal, or if the program crashes.

IO Redirection

smash supports redirection in the following formats:

- `cmd > FILE` : Redirect standard output to `FILE`, truncating it if it exists.
- `cmd >> FILE` : Redirect standard output to `FILE`, appending to it if it exists.
- `cmd 2> FILE` : Redirect standard error to `FILE`, truncating it if it exists.
- `cmd < FILE` : Redirect standard input from `FILE`.

Note that the syntax of the forms `cmd > FILE` and `cmd >FILE` are both supported for all of these redirection options.

Non-Interactive Mode

In the event that smash is given a filename argument when being launched, or a shell script contains a shebang that points to the smash binary, smash will run in non-interactive mode. The main difference between non-interactive and interactive modes is that jobs can not be background in

non-interactive mode (since the shell itself is "backgrounded" already). Aside from that, when launched in non-interactive mode, smash will real all lines from the input file one by one, launching and completing each job as it processes it. When smash detects EOF or EOT, it terminates cleanly.

General Overview

When smash first starts, it performs various initializations (setting command line options, checking if this is supposed to be an interactive or non-interactive session, etc). Also of note is that, during initialization, smash uses `sigaction(3)` to set all process control signals to `SIG_IGN`, so that the shell itself may ignore these signals, which will be later unignored by the child processes which it launches so that they may be handled.

After this, the program goes into its main loop wherein it will prompt the user for input, read the input from the user (through the use of `pselect(2)` and `read(2)` to only read 1 character at a time, and to avoid shenanigans with blocking IO).

The command line is then parsed. While parsing, we maintain three things: 1. the entire command line, as entered raw by the user. This is then split into... 2. each individualized command (in the case of pipelining), wherein the entire command line (from 1) is split around `|` characters. Each individual command is further divided into... 3. each individualized component, wherein each command (from 2) is split around any whitespace characters. This effectively turns each component into a single word/token.

We then check to determine if this is a builtin command or not. If this is a builtin command, we simply execute that builtin command and perform the next loop iteration.

If this is not a builtin command, the shell will then launch the program. The shell will `fork(2)`, and the child process will reset all signal handlers to `SIG_DFL` so that the child process may handle them as normal. The child will set the process group id of itself.

In the case that this is a foreground command, or that we are foregrounding a previously backgrounded job, we check to see if there is a need to give control of the terminal to the child process using `tcsetpgrp(3)`. We also save the terminals attributes so that it may later be restored upon child exit and flush all output from the terminal, using `tcgetattr(3)` and `tcsetattr(3)`.

On the Reaping of Children

For the sake of simplicity, smash does not rely on asynchronous notification of child process' signals. Instead, we perform non-blocking `waitpid(2)` calls, directly before and after prompting the user for input. Since the rest of the main loop is simply bookkeeping, the only potential problem would occur while the user is typing input. Since this is an interactive program which relies heavily on user input, it was decided to be better to not have messages about completed/stopped/etc jobs being output to the console while the user is attempting to type input. Instead, just wait until after the user has finished entering input, then see if there are any children which need to be handled.

Jobs

Each user input not corresponding to a builtin command eventually makes it way into the jobs list. The jobs list is maintained as a singly linked list of all jobs that the shell has processed thus far.

A user can check the status of all jobs with the builtin `jobs` command. All jobs which have terminated since the last time the `jobs` command was invoked will be output (along with their exit

codes or terminating signal numbers) before being removed from this list. This allows the user to check the statuses of past jobs without having to worry about missing the output message on the terminal, since they are free to invoke the `jobs` command at their leisure.

Jobs begin in a `NEW` state. As a particular job progresses through its life cycle, it may change to `RUNNING`, `SUSPENDED`, `EXITED` (for successful completion with an exit code), `ABORTED` (for termination via signal), or `CANCELED` (wherein it has been aborted, but has not yet been reaped).

Each job is tied to a particular user input and a `job_t` is maintained which contains some information about each job (status, exitcode, the user input parsings, jobid, process group id, any terminal attributes which may need to be restored later on, background/foreground status).

Command Line option `-t` When specified, this option will track the execution times of child processes. The shell will report the real, user, and system times utilized by the process during its execution. This is accomplished through the use of the `wait4(2)` system call, which populates a `struct rusage` which will end up containing the system and user times used by the child process. In order to determine the real time which was used by the program, we save the results of a `gettimeofday(2)` call on child process launch, and again when the child is reaped. The `timersub(3)` function is then used to calculate the difference between these two values, which is the total of the real time that the process was executing.

Support for Wildcard Characters and Tilde Expansion smash will attempt to resolve wildcard and tilde characters while launching processes. If an argument to a child process is found to start with either a * or a ~, then we rely on the use of `glob(3)` to resolve these. `glob(3)` will build an array which can be used to represent `argv` for a program.

The support for tilde expansion allows users to pass home directory paths without having to provide a full path (`cd ~user` or `ls ~root`).

Support for pipelining In addition to standard input/output redirection via <, >, », and 2>, smash also supports pipelining between programs. Given the syntax `prog1 | prog2 | prog3`, prog1's standard output will be tied to prog2's standard input, and prog2's standard output is tied to prog3's standard input.

This is accomplished with the use of `pipe(2)` to create a pipe, then by using `dup2(2)` to copy the file descriptor of the appropriate end of the pipe to be in place of `STDIN_FILENO` or `STDOUT_FILENO`, as necessary.

In addition, we also set each individual process' process group id to be the same as any other process' in the same pipeline. This has the effect of allowing us to `killpg(3)` the entire pipeline easily, as well as simplifying the effect of C-c.

Note that it is possible to provide a malformed command line (which may not act as expected) if you provide both pipes and standard redirection in the same input line. In the case where a program is given a >to redirect its output, as well as being the command preceding a `|`, the `|` should take precedence, as the rest of the pipeline depends on the output of the command. Similarly, in the case where a program is given a `<` to redirect its input, as well as being the command following a `|`, the `|` should take precedence.