CS355 Hw 3

Due by the end of day **Sunday, 2/25**. Design document due on **Monday, 2/19** in class. Written supplementary problems (except for 39 and 40) due on **Monday, 2/26 in class**. Extra Credit: Baboon crossing programs and test outputs (due date flexible).

Programming Assignment: This is a group project and you may form a group of 2 or 3. You can base your implementation on previous work from any partner. The advantage of having an extra coder may not outweigh the overhead of coordination, in fact the project can certainly be completed by one person (it is not that much code). It is a group project because having the moral support when you are mired waist-deep in man pages can be a tremendous help. It is highly recommended that you use some form of version control, so that you do not duplicate, or worse, interfere with each other's efforts.

This week's lab each group will discuss your initial program design and receive feedback.

Extended Shell with Job Control

1. Backgrounding with `&`, e.g. `emacs -nw hw1.c&` will run the editor *emacs* in the background. Note that `-nw` specifies no window, which works more readily with a classic shell.

   When a process is executing in the background, the shell should not block and wait but rather continue accepting commands.

   A shell with forked child processes executing in the background must be able to tell when the child processes exit. It can do this either synchronously or asynchronously.

   - In the synchronous approach, the shell periodically polls for child termination. You can do this by using some of the *wait()* sytem calls with the `WNOHANG` flag. This approach is easier to implement, you might want to start with it. Eventually however, *mysh* must use asynchronous notification of job termination. The problem with polling is that if polling intervals are large, it takes too long to detect the actual termination. On the other hand, if polling frequency is increased, the overhead will be too much.
   - In the asynchronous apporach, the shell receives a signal from the OS as soon as the child exits. Consult the *signal()* man pages if needed. The signal the OS sends to notify a parent process that a child has exited is `SIGCHLD`. Thus, you must register a handler for `SIGCHLD`. You have seen signal handler registration with the system call *signal()* before. This time around you should use the system call *sigaction()*, which is part of the POSIX signal handling rountines. *sigaction()* provides a more comprehensive and reliable mechanism for controlling signals, but is (of course) harder to use.
   - Be mindful of concurrency issues that potentially arise, e.g. bad things will happen if your shell is updating a list of jobs in response to user input (i.e. add a new job) at the same time the signal handler for child process termination also attempts to update the list (i.e. remove the child job). You may delay signal handlers until the program is past a critical region by masking signals. Consult the *sigprocmask()* manpage. You may also find the POSIX signal set operations *sigemptyset(), sigaddset()* useful.

- As you might have guessed from reading the above, you need to keep a list of all jobs currently running. You must use a linked list to keep track of this.

2. Process suspension with Control-Z

   This should also be implemented with signals. The signal sent to the process is `SIGTSTP` (by the terminal driver). Your shell may need to send a `SIGSTOP` signal to the process currently executing in the foreground (depending on implementation), update the job list, and return with a new prompt.

   Related is the `bg [%<#>]` command, which resumes a job suspended by control-z in the background, as if it had originally been started with `&`. If job `#` is omitted, shell should background the last job suspended, if any.

3. `fg [%<#>]` – bring backgrounded job `#` back into the foreground.

   If bringing back a stopped/suspended process, the shell must first send a `SIGCONT` signal. If unfamiliar, play with Linux's `fg` command to see how it works. If job `#` is omitted, shell should bring back the last job backgrounded, if any.

4. Process groups

   Every process belongs to a process group and signals tend to go to all processes in the same process group. When a process is created, by default it becomes a member of the same process group as its parent process. If you leave the spawned processes in the same process group as the shell, they will all receive signals along with the shell, e.g. the terminal driver will send a `SIGTSTP` to all processes when a user types Ctrl-Z, and suspend all of them, instead of the intended one in the foreground. To set the process group of a process use the *setpgid()* system call. In general at this stage of your shell, every process should be in its own process group.

5. Foreground/terminal

   A process having access to the terminal is known as executing in the "foreground", and there can only be one. Be very careful with how you manage who has access to the terminal. There should only be two possbilities:

   (a) The shell has the foreground

   (b) The shell *foregrounds* a process group by giving it unlimited and sole access to the terminal. In this case the shell should be waiting for this particular child to exit to regain the foreground and therefore terminal

   If your shell ever lost access to the terminal because a backgrounded child somehow has access to the terminal, your shell will be completely nonresponsive. To ensure that the currently running foreground process's process group is indeed the one you want, use the *tcsetpgrp()* system call. The *tcgetpgrp()* may also be useful.

6. Built-in commands

- `jobs`
  Prints a list of jobs that are currently running or suspended/stopped. This list should include an integer (the job number), the command line that started the respective program and the status (Running or Suspended). Check with the corresponding Linux command if unfamiliar with its behavior.
- `kill [%<#>]`
  Send a `SIGTERM` to the specified job and terminate it. You will notice that this does not terminate many applications, as they register signal handlers to catch `SIGTERM`. Implement also the `-9` flag to send `SIGKILL`, which can not be caught.
- `fg [%<#>]`
  Continue the specified job in the foreground
- `bg [%<#>]`
  Resume the specified (suspended) job in the background, as if it had been started with `&` originally.

7. Obviously, your parser needs to parse the symbols `&`, `%` etc. As you implement this, remember that white spaces are not necessary to separate the symbols, but an arbitrary amount is allowed to be present.

8. Add the ability to parse and handle `;`.

9. Expanded README

   Your README must explain any particular design decisions or features of your shell. It should contain at least three parts:

   - An introduction to the basic structure of your shell
   - A list of which features are fully implemented, partially implemented and not implemented
   - A discussion of how you tested your shell for robustness and correctness

   You should get used to providing your software with an overview document which will make granting credits easier. You should write this document especially well if you are unable get things working completely but have the bones right.

10. Design document

    A design document is required from each group by Monday 2/19, where you sketch an outline of how your program is going to work. It should consist of a main control loop, calls to functions and declarations of important variables and data structures. Functions should be in prototypes only, with short comments on what the function takes, what it returns and what it is supposed to do. For most projects, this means a fleshed out `main()` with function calls to unimplemented functions. A successful design document should convince the readers that your shell works and how it works, granted that your functions will work exactly as advertised.

You should granulate your functions enough that control flow and overall functionality are convincing. In other words, an extreme case of what I don't want to see is for example, a `main()` with a single function call to a function that looks like this:

```
void do_shell(void) {
//make the shell work!
}
```

Ultimately, if you can not convince yourself that your design will work, it certainly will not convince me (or have any chances of actually working). I can not emphasize enough how important it is to try to produce a design that will not require any major changes during the final implementation. It means that your implementation will be nothing other than making individual functions work as they are designed, and group members can then divvy up the functions without overhead. The more time you spend THINKING about your design, the less time you will end up wasting on the actual coding. What distinguishes an excellent software engineer from the rest is not the ability to implement, but this elusive quality to convert a problem to an elegant programmable design. The rest is easy.

11. EXTRA CREDIT: support pipes and redirections. You need to add parsing for `|`, `<`, `>` and `>>`. Again, just like the parsing for `&` and `;`, white spaces are not necessary to separate these tokens. If you are not familiar with pipes and redirections, please play with them first on a true Linux shell to understand how they work. Systems calls you will find useful: *pipe()*, *dup()* and *dup2()*. Also keep in mind that true unix pipes (and redirections) are unlimited in length.

12. Hints:

    - As the basic shell, any crashing will be heavily frowned upon, so please test extensively!
    - Man pages are your friends!
    - The Gnu C library manuals can be found at: `http://www.gnu.org/software/libc/manual/html_mono/libc.html`. Read the sections on processes, job control and signal handling. It contains a lot more information than is required on this project, but it can be very helpful. Under the job control section it also has sample code for a partial shell, focusing on job control. Note that the implementation is somewhat different from described here, i.e. job status notification is not done via signals.

      You may in general take and make use of any publically posted code, however, you MUST credit the source. Also, taking large block of code that you do not understand will NOT help you. It only slows down your progress and makes debugging impossible.
    - Divide and conquer will make your life much easier, never try to do too much too quickly without making sure you have thoroughly tested components.
    - Utilize unit testing. The documentation requires that you explain how you tested. Remember that you need to run tests on both sides: if your implementation does the right things with good input and that it does not do wrong things with bad input.

- Signal handling is tricky, the trickiest part of this assignment. Handle with care, and test everything. Don't even think about the rest of it until you at least can properly background a child with `&`.
- When in doubt, do what Linux does. I will never fault your shell for behaving exactly like Linux!
- This is not a trivial assignment, start EARLY!

Supplementary Questions: Do the following questions from Tanenbaum.

- pages 174-180, problem numbers 43, 44, 45, 49 and 50
- pages 465-470, problem numbers 2, 21, 22, 23, 26, 30. For 26, change Process C's availability allocated vector to 11011 and max to 21311.
- Extra Credit: 39 and 40.

  If you are attempting 39 and 40, use the following assumptions in your implementation:

  1. Simulate each baboon as a separate process. You will want to set up shared memory regions with `mmap` and/or `shm_open`.
  2. Altogether, 100 baboons will cross the canyon, with a random number generator specifying whether they are eastward moving or westward moving (with equal probability).
  3. Use a random number generator, so the time between baboon arrivals is between 1 and 6 seconds. You should launch all the baboons together, but have each sleep a random number of 1 to 6 seconds, instead of generating each sequentially with 1 to 6 second delay. The former will force a lot more crowding and be more interesting.
  4. Each baboon takes 1 second to get on the rope. (That is, the minimum inter-baboon spacing is 1 second.)
  5. All baboons travel at the same speed. Each traversal takes exactly 4 seconds, after the baboon is on the rope.
  6. Use semaphores for synchronization.
  7. You should generate sufficient printouts to demonstrate correctness of program, which should include:
     - No east/west collision on rope
     - No collision getting onto the rope
     - There are mulitple baboons on the rope at a time
     - Maximum number of baboons on rope is not exceeded
     - Direction switching for the no-starvation version (40)

     At the minimum, the following events should be logged. You might need more:
     - west/east baboon [ID] arrived [system time]
     - west/east baboon [ID] starts crossing [system time]

– west/east baboon [ID] finishes crossing [system time]

Turn in your code electronically (via submit and use `-p 3`). Also turn in hardcopies of three test-run results in class. If the test-runs are long, please cut out the middle so that you do not print out too many pages. Include enough run results so that it clearly shows how your program works. And make sure you mark your test-runs starvation or non-starvation.

Deliverables:

1. Harcopy: Design document (Monday 2/19).
2. Hardcopy: Answers to the supplementary questions (Monday 2/26) in class
3. Electronic submissions of the shell programming assignment (due Sunday 2/25)
4. EC: Hardcopy test-run results for baboon-crossing (39 and 40) (TBA)
5. EC: Electronic submission of the baboon programs (TBA)