Systems Programming

-- Chapter 10: Shell Building --

Franz Kurfess

Spring 2004

0.5

Chapter Overview

<subtitle>Shell Building</subtitle>

building a simple shell for a Unix system

- **Shell Overview**
- **Parsing**
- **Launching Processes**
- **Inter-Process Communication**
- **Releasing Processes**
- **Signal Handling**

Shell Overview

important aspects of shells

Shells in Unix

even in GUI-oriented systems, *shells* are very important for user interaction

a shell is a process that interacts with the user

the shell process creates new processes for many commands issued by the user

some commands are built-in, i.e. directly handled by the shell, and not handed over to separate processes

the shell is responsible for setting up communication structures for child processes that require it, e.g. via pipes or intermediary files

the shell is also responsible for some of the signal handling since it is the process that the user primarily interacts with

Shell as Command Line

a shell is also known as *command line*: it accepts commands from the user, usually in a line-oriented fashion

after accepting an input line, the shell has to determine the commands, options, and arguments issued by the user

this is done by a *parsing* component (e.g. parseline)

once the commands are identified, the shell sets up the communication infrastructure (e.g. via pipes), and then launches new processes to execute the commands via fork() and exec()

in most cases, the shell just waits until a command issued via a child process is completed

programs that are intended to run for a longer time period may be launched in the background, making

Shell as Command Line (Continued)

the shell available to the user again

after a child process terminates, the shell needs to release the process via wait()

the user can terminate the shell, usually through a command such as exit, logout, quit or a special key combination such as `Control-D`

the following slides contain mostly practical hints, and are not necessarily a systematic, in-depth treatment of that issue

Parsing

- identifying commands, options, and arguments
- commands may interact with each other, often via pipes
- redirection operators may be used for input and out-put

Parsing Hints

strictly separating all commands and arguments by spaces uses up more space in the input line, but makes parsing much easier

the input line is separated into *tokens*

the tokens are examined for their purpose:

- **command**
- **option for a command, usually indicated by a dash (-)**
- **argument for a command**
- **pipe operator (|)**
- **redirection operators (< and >)**

the parsing component produces intermediate structures that are used by the shell to set up the processes, and if necessary the communication infra-

Parsing Hints (Continued)

structure between processes

Launching Processes

the shell creates child processes for most commands issued by the user

Process Creation

new processes are created via fork()

the children inherit most of the parent process' properties, such as open file descriptors, or the signal mask

Process Management

the shell must keep a list of the processes it launched

since the shell is the main point of contact with the user, it may have to assist the children with the handling of signals

Cleaning Up

before a process exits, it should clean up after itself

in particular, open file descriptors need to be closed

Releasing Child Processes

- the shell has to release each child process it created via wait()
- wait() may get interrupted by a the signal handler, so it is important to examine its return value to make sure that the child actually exited; otherwise, the shell may lose track of the number of children that are still around
- after the child process has finished executing the command, it is advisable to flush `stdout` via fflush(); otherwise there might still be some output from the old process in the buffer used by `stdout`

Inter-Process Communication

commands issued by the user may exchange information

this is frequently done via *pipes*, e.g. as in ps -aux | sort

Pipes between Processes

- the most frequent setup is to direct the output of one process directly to another process
- alternatively, a file can be used as intermediate repository
- the function pipe() sets up two file descriptors connected by a buffer
- the writing process deposits its data into the `write` end of the pipe (its input, or head)
- the data are temporarily stored in a buffer
- the reading process obtains its input data from the read end of the pipe (its output, or tail)
- if the pipe's buffer is full, the writing process receives a signal indicating that it should not send any more data

Pipes between Processes (Continued)

- **if the pipe is empty, the reading process sleeps until data arrive**

Pipes and File Descriptors

dealing with pipes requires the management of several file descriptors

it is important to close the unneeded file descriptors, ideally as soon as it is known that they are not needed anymore

since a process has a limit on the number of open file descriptors, it can happen that this limit is reached if file descriptors are not closed

after a fork(), the child has a copy of all file descriptors from the parent process; the ones it doesn't need should be closed right away

a process reading from a pipe will get an end of file (EOF) only after *all* open file descriptors to the `write` end have been closed; if the `write` end of the parent process isn't closed, the pipeline may hang forever

Pipes and File Descriptors (Continued)

Releasing Processes

the shell is responsible for the release of its children after their termination

Termination of Child Processes

since exec() only returns in case of an error, it is a good idea to print out an error message and exit in that case

a child process terminates normally when the program invoked via exec ends

you can assume that those programs terminate properly

the shell must have a wait() or waitpid() for each child process in order to release them properly

Signal Handling

figuring out who is responsible for what signals

Signals and the Shell

child processes running in the foreground are largely responsible for dealing with signals from the user since they are in control of the terminal

processes in the background are less accessible to user signals: kill -SIGNALNAME pid instead of key-board sequences

in more complex situations, the shell may have to help coordinate signals between processes

child processes inherit the signal mask from their parent process; if the parent process blocks signals, the children won't listen to those signals unless they are explicitly unblocked

Implementation and Testing

there are many different strategies for implementing a shell

here is the approach I recommend

Overall Design

before you start coding, develop an overall design for your shell

this can be a sketch of building blocks, some pseudocode, or a more formal design document such as a UML diagram or specification

the components identified above are probably a good starting point

Implementation

once you have the overall design, start with the implementation of the main components

concentrate on one component, and try to make it work before proceeding to the next one

for components that need to interact with each other, use function prototypes or similar techniques to make sure that they work together

Testing

test often, and as thoroughly as feasible

fixing an error in a component you thought is working correctly when you're working on a different component is most likely to be more tedious and time-consuming than fixing it while you're working on that particular component

after a component is completed, try it out in combination with the ones it is supposed to interact with

when you're stuck, take a break

Chapter Summary

<subtitle>Shell Building</subtitle>

Shells in Unix

a *shell* is an important interaction component between the user and the system

since a user typically types one or more commands in a line, and then sends it to the shell by pressing the `RETURN` key, shells are also known as *command lines*

for most commands issued by the user, the shell launches separate child processes

shells frequently use the *redirection operators* (< and >) for more flexible ways of dealing with input and output for commands

***pipes* are often used to direct the output of one command to be used as input by another command**

redirection and pipes can be handled by rearranging file descriptors for the processes involved

Shells in Unix (Continued)