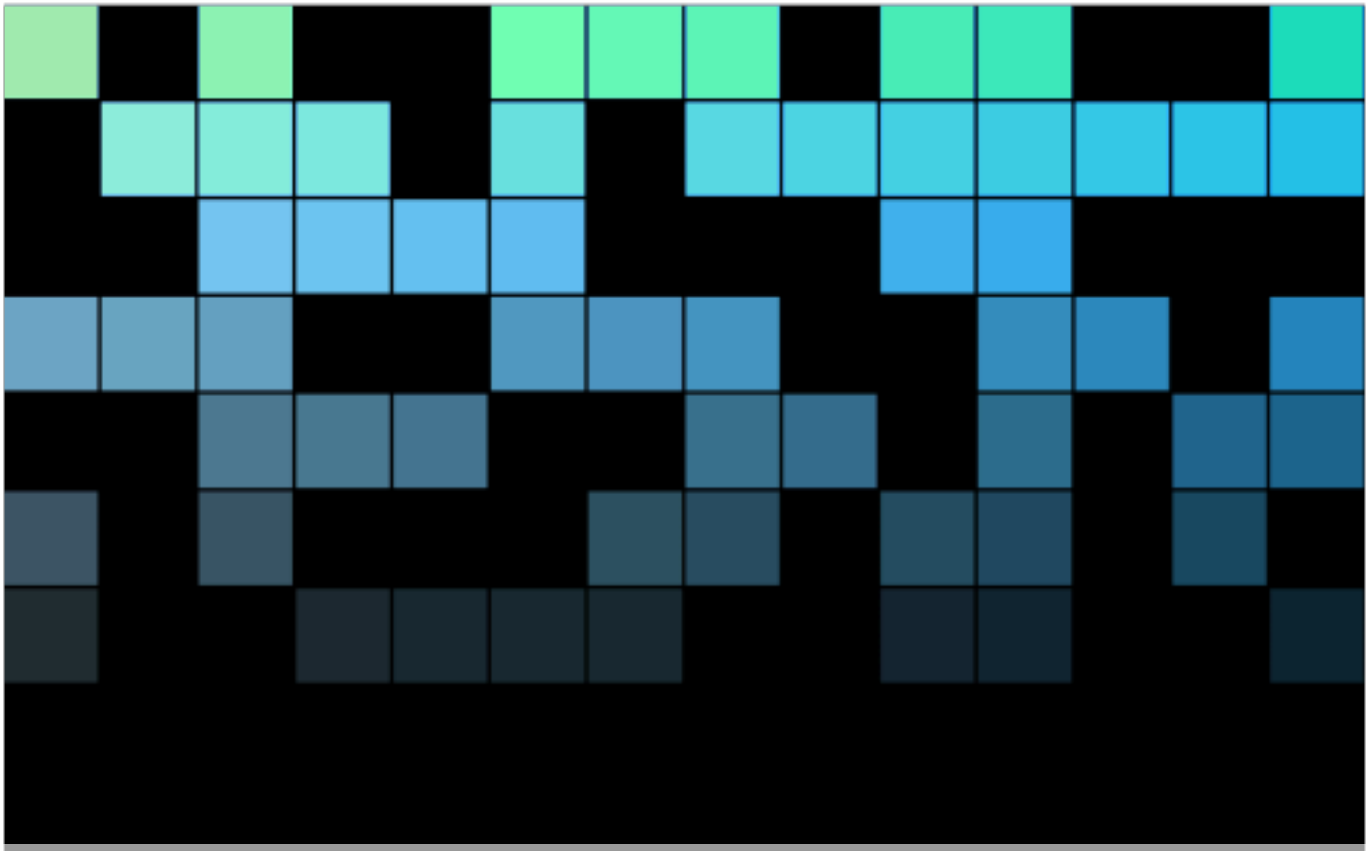
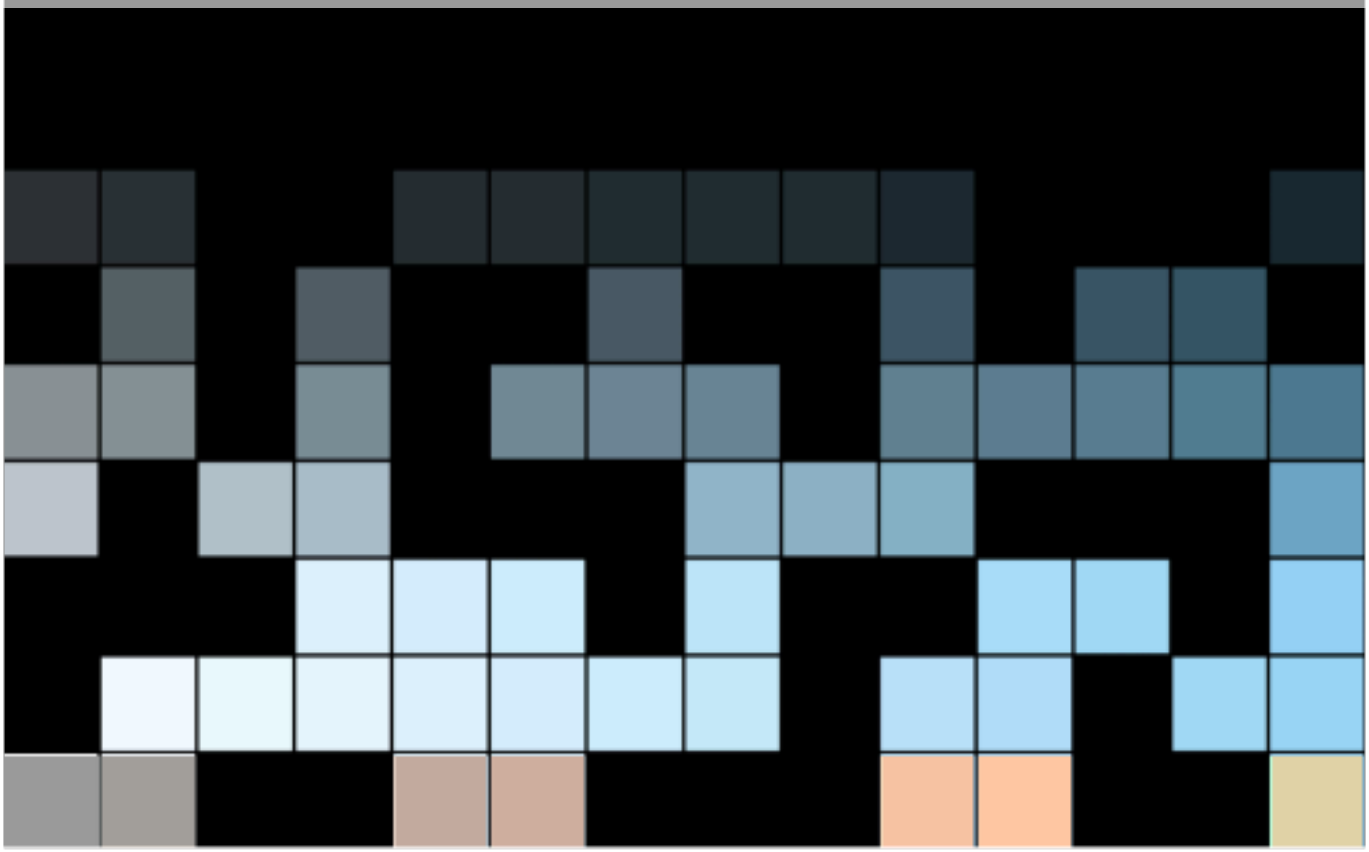


Unix Systems Programming

Ramesh Yerraballi



Copyright (c) 2017 Ramesh Yerraballi

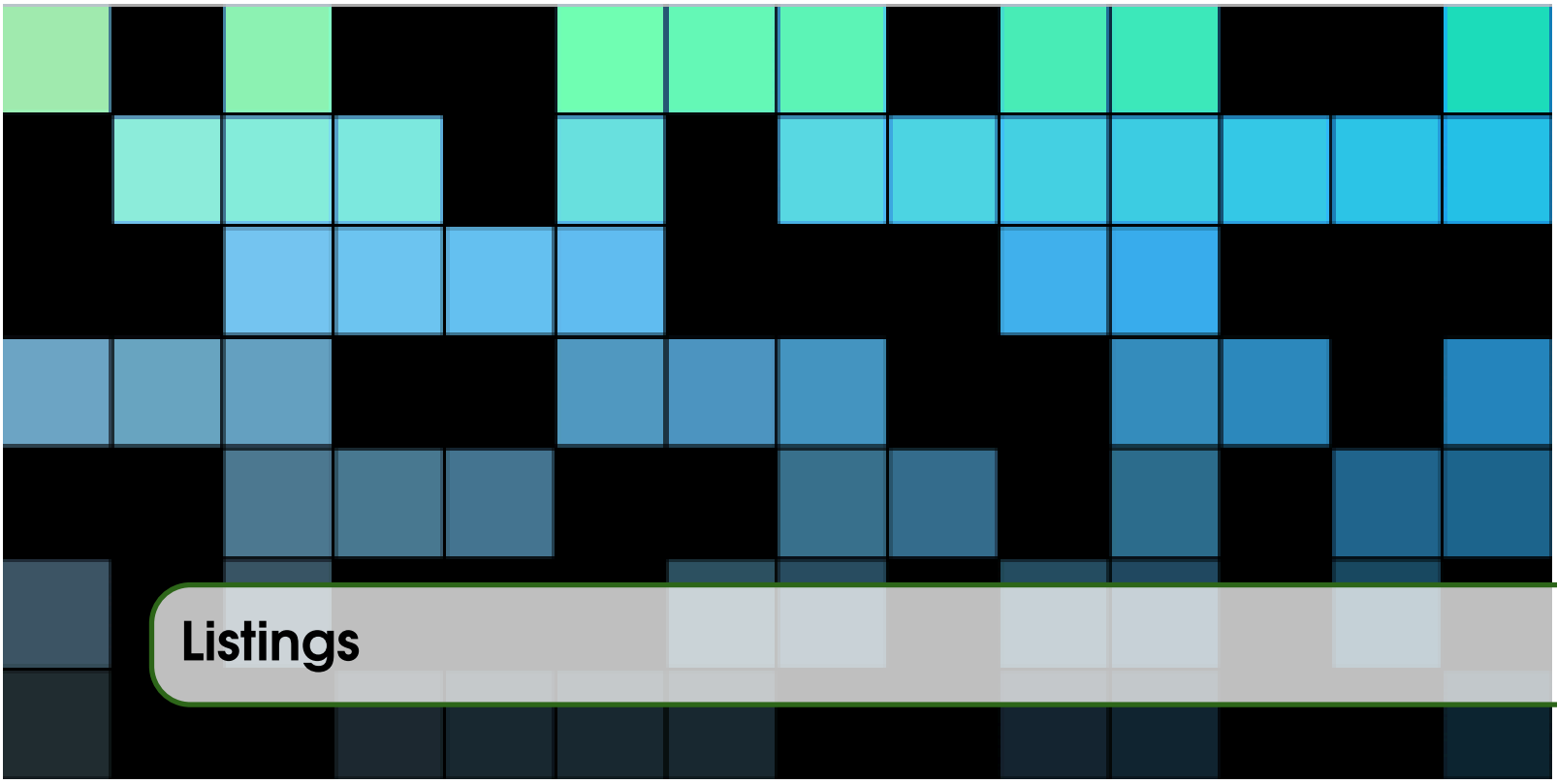
SELF-PUBLISHED

Licensed under the Creative Commons Attribution-NonCommercial 3.0 You may obtain a copy of the License at `http://creativecommons.org/licenses/by-nc/3.0`.
First printing, March 2017

Preface

This did not start as a book and may never make it as one. It simply is something I needed to write to help my students get started, become comfortable and eventually, fully embrace as their preferred operating system. That is the hope but for now its a work in progress.

Contents



Listings

0. Introduction to Unix

In 1969, a group of Bell Labs researchers, Ken Thompson, Dennis Ritchie and others, developed the first Unix Operating system. The original Unix OS was designed for the DEC PDP-7 computer and written in assembly language but a subsequent rewrite in C for the PDP-11 in 1970 brought Unix to the world at large. Bell Labs made the source code available to universities on generous licensing terms which triggered independent developments. Notable among this was the development at UC Berkeley, released under the name Unix 4.xBSD. The Bell Labs versions of Unix were released under the name System V Unix. As the open source community at premier universities embraced Unix, it morphed and evolved into several strains. Eventually, in the mid 1980, IEEE sponsored a project to standardize Unix, thus creating POSIX.

> **Aside** **POSIX** - The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems. [**Wikipedia**]

Today, the most popular version of Unix is the GNU/Linux which is comprised of the enabling tools from GNU (gcc compiler, coreutils, debugger, binutils, shell etc.) and the Linux Kernel developed by Linus Torvalds. Various distributions offer the GNU/Linux packaging as a complete Operating system suite with some custom applications thrown for good measure. Examples include Arch Linux, CentOS, Debian, Fedora, Gentoo Linux, Linux Mint, openSUSE and Ubuntu.

Desktop Linux distributions often provide a rich user experience with a desktop environment like GNOME, KDE, LXDE or Xfce. These dictate the look-and-feel of the actual "windowing" environment used to interact with the underlying operating system.

In contrast to desktop distributions, server distributions tend to omit the bells and whistles of a windowing system, focusing instead on software like, web servers, databases and programming languages. Notable among such server distributions is the LAMP software bundle which includes, Linux, the Apache HTTP server, the MySQL relational database and the PHP programming language support.

For the rest of the book I will assume you have a desktop Linux distribution of your choice running either natively on your PC or inside a Virtualization platform like Virtual Box `https://www.virtualbox.org/`.

Also note that this is not a text on how to use the Unix operating system but rather how to program in it. Specifically, we will program in the C programming language utilizing the POSIX API that the underlying Unix operating system exports.

0.1 The POSIX API

The POSIX standard (`http://pubs.opengroup.org/onlinepubs/9699919799/`) is a extensive corpus that specifies a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. It is intended to be used by both application developers and system implementors.

In this text we will focus only on the operating system interface in the form of system calls. System calls are a mechanism by which an application programmer accesses features of the operating system that provide abstractions of, processes, memory, files and i/o devices. Note that the role of the operating system is to manage the resources provided by the underlying hardware and make them accessible to the application programmer as abstractions that hide the low-level implementation details. For example, the OS provides an abstraction called a *process* that encapsulates within it an address space with one or more threads executing within and associated resources. The application programmer simply needs to know what system calls are used for the creation and management of a process. For example, Table 1 shows a partial list of system

System Call	Description
`pid_t fork(void)`	Clone the current process
`exec()`	Execute a new image inside a process
`pid_t wait(int *status)`	Wait for termination of a process
`pid_t getpid(void)`	Get process identifier
`void exit(int status)`	Terminate the current process

Table 1: A sample of the POSIX Process Management system calls

calls pertaining to process management. The POSIX standard does not just specify that a `fork()` system call ought to be supported but, pins down the actual C function prototype for the `fork()` system call and what effect its execution must have. In other words, both the form (the syntax) and the function (the semantics) must the met to be POSIX compliant.

0.2 Unix Basics

As the focus of this text is primarily on System calls, I will assume a certain familiarity with the use of the Unix operating system at the shell level. Specifically familiarity with the bash shell and common command line utilities is assumed. It will also be useful to have some working knowledge of the Linux system, the boot sequence, the filesystem structure, commonly used commands, operating at the command-line using features like pipes and i/o redirection, how makefiles work and, simple shell scripting.

First a few notes on notation:

- The shell command prompt is shown as $
- All typed commands are shown in the `monofaced` font
- Outputs of commands are shown in monospaced font with a gray background
- Comments are *italicized*
- The username of the user in examples is assumed to be `younix`.

0.2.1 The Boot Procedure

The Linux system boot procedure involves several stages:

- The BIOS runs first. It finds the Master Boot Record (MBR) on the first sector of a bootable disk (/dev/hda or /dev/sda) and runs it.
- The MBR is too small (512 bytes) to hold an entire operating system, what it holds instead is a "Primary Boot Loader". This loader has information about where the Grand Unified Bootloader (GRUB) is loacated, which it then loads and runs.
- The GRUB interface allows you to choose between multiple Kernel images (Windows/Linux/MacOS) etc., that have been installed on your PC. When Linux is one of choice GRUB loads and runs the Linux kernel and initrd images.
- The first process that the kernel runs is the `init` process (with process id 1). The initrd image has a temporary root file system (with some drivers) which is mounted by the kernel during startup. Eventually, this is replaced by the a real file system.
- Under most Linux distributions the init daemon process `systemd` manages all other daemons that essentially make the system operational.

> **Aside** **Daemon** - In multitasking computer operating systems, a daemon is a computer program that runs as a background process, rather than being under the direct control of

an interactive user. Traditionally, the process names of a daemon end with the letter d, for clarification that the process is, in fact, a daemon, and for differentiation between a daemon and a normal computer program. For example, syslogd is the daemon that implements the system logging facility, and sshd is a daemon that serves incoming SSH connections [**Wikipedia**]

For the most part, users do not pay much attention to the boot procedure unless there is a fault. If there are no faults then the systemd daemon launches several services (daemons) including a logging daemon, utilities to control basic system configuration like the hostname, date, locale, maintain a list of logged-in users and running containers and virtual machines, system accounts, runtime directories and settings, and daemons to manage simple network configuration, network time synchronization, log forwarding, and name resolution.

0.2.2 The Logind Daemon

The last service that systemd runs before handing over control to user interaction is the logind service after which the Login splash screen is displayed. The username and password entered are checked by the logind service for authentication of credentials. Once the user passes authentication she is presented with a login shell or a full windowing system depending on how your particular system is setup. If you are in a Login shell and not the full windowing system, you simply have to run the following command to launch the desktop environment:

```
$ startx
```

At this point you are inside the desktop environment. Your primary mode of interaction will be a shell but other applications like a Browsers, Editors, IDEs and such can be launched from here. A Terminal is a graphical interface that gives a user access to a shell. The default shell on most Linux systems is Bash and that is what I will assume your system is setup with as well. If you know enough about shells that you have a different preference (say zsh) then you can skip the next section.

0.2.3 Work Environment

The environment of your shell is determined during a two stages, the first stage is at login and the second is upon launch of a new interactive shell (for example when a Terminal is launched). When bash is invoked at login, it first reads and executes commands from the file /etc/profile, if that file exists. After reading that file, it looks for ~/.bash_profile, ~/.bash_login, and ~/.profile, in that order, and reads and executes commands from the first one that exists and is readable.

When an interactive shell that is not a login shell is started, bash reads and executes commands from /etc/bash.bashrc and ~/.bashrc. Where, ~ represents your home directory, which is

typically `/home/younix` for our example user. Note that files and directories that are prefixed by a dot (the period character: `.`) are considered "hidden" and therefore do not show up on a listing using the `ls` command.

The most important environment variable you will need to manipulate at some point is the `PATH` variable. It determines where the shell ought to look for and in what order, for the commands you ask it to run:

```
$ echo $PATH
/usr/sbin:/usr/bin:/sbin:/bin
```

This may be what the system set your `PATH` variable by default. The echo command is printing this variable (note the $ in front to refer to the varaible name "PATH" and not just the string "PATH"). Say you installed some software (say called `foo`)in the `/usr/local/bin` directory and you want the shell to find it.

```
$ foo
foo:  command not found
$ export PATH=/usr/local/bin:$PATH
$ foo
foo's output
```

You can add the export statement to your `~/.profile` so it comes to effect on login. Edit the file to do this or:

```
$ echo "export PATH=/usr/local/bin:\$PATH" >> ~/.profile
```

The >> operator (two greater than symbols) appends the echoed string to the target file. Related operators > and < do output and input redirection:

```
$ echo "Howdy Joe" > greet
$ cat greet
Howdy Joe
$ echo "Fine Thank you" >> greet
$ cat greet
Howdy Joe
Fine Thank you
$ wc -w < greet
5
```

Checkpoint 0.1 Which symbol represents a user's home directory in Unix? **Solution: [1.1.]**

The main thing to remember about these dotfiles is, the commands in `~/.profile` file are executed only once on the user's login, but `~/.bashrc` is executed each time a new shell is opened.

0.3 Doing More in the shell

Most seasoned Unix users spend a good bit of time at the shell prompt because it is possible to do just about everything that your computer can do, from the shell prompt. Yes, including web browsing (`lynx`) but lets be practical, what about cat pictures!

Here are a few concepts that will help you get more out of your shell experience.

0.3.1 Installing software

You were probably given an option to choose application and utility software to install at the time of your Linux installation. This may restrict what commands or utilities you can run. However, you can add new software as the need arises using a package install utility. Three notable package installers in the Linux world are `apt`, `yum` and `pacman`. Debian (.deb files) based distributions like Ubuntu and Linux Mint use `apt` and RPM (.rpm) based distributions like Redhat use `yum`. Arch Linux uses `pacman` which works with `.tar` files.

All pretty much do the same thing which is search and install software for repositories from the Internet. I will assume you have a Debian based distribution and so `apt` will be your package installer. Say you wanted to use a tool called `tree` that lets you see a hierarchical tree view of a directory:

```
$ tree
The program 'tree' is currently not installed.  You can install
it by typing:
sudo apt install tree
$ sudo apt install tree
[sudo] password for younix:
```

Installation of `tree` and any other packages that tree depends on follows...

To search for packages in repositories by keyword you can use `apt-cache search keyword`.

0.3.2 The file system

The filesystem is at the heart of a user's experience in the Unix environment. Different variants of Unix use different implementations (ext4, HFS+, UFS, ZFS) of the filesystem. However, common to all of them is the hierarchical (tree like) structure with the root node represented by the name `"/"`. The nodes in the filesystem can be of three different types: regular files, directories and

special files. We will look at special files in some detail later, for now we focus on just directories and files.

```
$ cd /
$ tree -d -L 1
```

show directories only and restrict to depth of level 1

```
.
├── bin
├── boot
├── cdrom
├── dev
├── etc
├── home
├── lib
├── lib64
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

Lets get a better sense of the file system and some common commands used to manipulate and navigate it.

```
$ cd
```

cd with no argument takes you home

```
$ pwd
/home/younix
$ mkdir testdir
$ cd testdir
$ touch foo
```

touch changes a file's timestamps; creates it if it does not exist

```
$ ls -la
total 8
drwxrwxr-x 2 younix younix 4096 Aug 16 01:27 .
drwxr-xr-x 6 younix younix 4096 Aug 16 01:27 ..
-rw-rw-r- 1 younix younix 0 Aug 16 01:27 foo
```

There are 8 columns in the output:

- *The first column shows file permissions*
- *The second column shows the number of hard links*
- *The third and the fourth ones are owner and group names*
- *The fifth is file size*

- *The sixth and seventh are date and time of last modification*
- *the last is the name of the file.*

```
$ cd ..
$ pwd
/home/younix
```

Note that every directory has two entries in it, the current directory named "`.`" and the parent directory named "`..`". A directory is simply a file which contains mappings of filenames to the corresponding file structures called i-nodes (more later). This mapping is sometimes referred to as a "link". This is the connotation of the word "link", the second column of the output of the `ls -la` command in the example above. The hard link count is the number of entries in the directory.

Checkpoint 0.2 Do all directories have parent directories? **Solution: [1.2.]**

0.3.3 whereis which and whatis

As you progress on your path to getting proficient with your Unix environment you will ask questions like, what does this command do? How can I get more details about running this command? Where is this command located? If there are multiple commands by the same name, which one will be run? Responding to these questions requires getting acquainted with these four commands: `whatis`, `man`, `whereis`, `which`.

```
$ whatis find
find (1) - search for files in a directory hierarchy
$ man find

FIND(1)        General Commands Manual         FIND(1)

NAME
find - search for files in a directory hierarchy

SYNOPSIS
find [-H] [-L] [-P] [-D  debugopts] [-Olevel] [starting-point...]
[expression]

DESCRIPTION
This manual page documents the GNU version of find. GNU find searches
the directory tree rooted at each given starting-point by evaluating
the given expression from left to right, according to the  rules of
```

```
precedence (se section OPERATORS), until the outcome is known (the
left hand side is false for and operations, true for or), at which
point find moves on to the next file name. If no starting-point is
specified, '.' is assumed.
...
...

$ whereis find
find:  /usr/bin/find /usr/share/man/man1/find.1.gz
$ which find
/usr/bin/find
```

`whatis` gives an one-line brief about what the command does and `man` gives you the entire manual on the command. The distinction between `whereis` and `which` is subtle. `whereis` tells you where all resources pertaining to the command are located, i.e., the executable, the documentation etc. `which` on the other hand tells you which resource will be executed if you were to type the command at the shell prompt. Knowing which of the possible commands by the same name will be chosen to run can save you headaches arising from naming multiple utilities by the same name. Of course the one that gets run is the one located earlier in the environment variable `PATH`.

Aside **A Newbie's Trojan Horse** - The environment variable `PATH` determines the order in which the shell looks for the requested command to execute. So, if there were two executables called `bar` in two different directories, say `/usr/bin` and `/usr/local/bin` and both of these directories are in your `PATH`

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

then the shell will choose to run `/usr/local/bin/foo` as `/usr/local/bin` appears before in the `PATH`.

Here comes the Trojan Horse. Say a newbie decided to add the current directory to her `PATH` as:

```
$ export PATH=.:$PATH
$ echo $PATH
.:/usr/local/bin:/usr/bin:/bin
```

Now, say you `cd` into a directory (`/home/trudy/welcome`) where Trudy planted a Trojan Horse, a executable cunningly named `ls`. This executable `ls` has some malicious code that runs before invoking the system command `/bin/ls`:

```
$ cd /home/trudy/welcome
```

```
$ ls
```

You will see the directories contents but, **alas! you've been had.** So, the moral of the story here is, if you have to add `.` to your path, make it the last thing in your path:

```
$ export PATH=$PATH:.
```

Checkpoint 0.3 Say you have an executable called `dothis` in the current working directory that you would like to run. You get this response:

```
$ dothis
dothis:  command not found
```

why? **Solution: [1.3.]**

0.3.4 Finding files

Searching for files by their names or by their contents is one of the most common things programmers do. The find command is a power tool that makes the task of searching for a needle in a haystack, a breeze. You can find a good tutorial on find here: `http://www.thegeekstuff.com/2009/03/15-practical-linux-find-command-examples/`

Here I will demonstrate two simple uses. Say you want to find all files with the `pdf` extension under the current directory tree. Lets create some files and subdirectories for demonstration purposes.

```
$ pwd
/home/younix
$ cd testdir
$ touch a.pdf
$ touch aa.pdf
$ mkdir bdir
$ touch bb.pdf
$ find .  -name '*.pdf'
./bdir/bb.pdf
./a.pdf
./aa.pdf
```

Note that the quotes around the pattern to search are necessary

For our second example lets ask find to run a command on each of the file found using a pattern match. More specifically we will use find to remove all files under directory `project` named `a.out` or `*.o`. This might be a common cleanup that programmers do to remove object files.

```
\$ find ./project \( -name a.out -o -name '*.o' \) -atime +7 -exec rm {} \;
```

The text contained between `\(` and `\)` specifies the pattern to match. The `-o` is an OR operator to tell to look for either of the two patterns. The text that follows `-exec` is the command to run where the `{}` is replaced with the name of the found file and `\;` marks the end of the arguments passed to the command (`rm` here).

0.3.5 Redirection

The shell has access to standard I/O through file descriptors 0,1 and 2. File descriptor 0 represents the standard input (`stdin`) which is the connected to the input of the terminal in which the shell is open. File descriptor 1 represents the standard output (`stdout`) which is the connected to the output of the terminal in which the shell is open. File descriptor 2 is the standard error (`stderr`) which remains connected to the terminal even if the stdin is redirected. `stderr` is used by commands to print diagnostic messages which you may not want redirected to the output.

We saw the use of redirection before where a command's input and/or output can be redirected to a file instead of the standard input and/or output. For example the command `ls` by itself prints the listing to the terminal (`stdin`). If we instead want the output to be redirected to a file, we run the command as `ls > filename`. The same can be done with input redirection and output appending (`>>`). The actual syntax for output redirection is `[n]>filename`, where `n` is an optional file descriptor number. Omitting it will default to 1. Similarly the syntax for output redirection is `[n]<filename`, where `n` defaults to 0 if omitted.

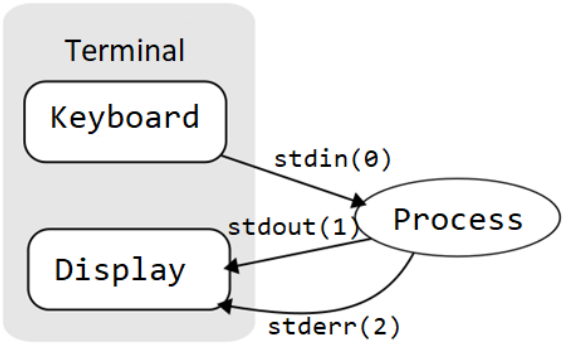


Figure 1: Standard I/O for a Process

That infamous pattern: `2>&1`: As mentioned before redirecting the `stdout` of a command to a file does not redirect the `stderr`:

```
$ ls
aa.pdf a.pdf bdir foo
```

Let us get the stats of the file aa.pdf and the directory named bd and redirect it to a file called outfile

```
$ stat aa.pdf bd > outfile
stat:  cannot stat 'bd':  No such file or directory
```

The stats on file aa.pdf have been correctly written to the outfile but as there is no file by name "bd", the stat command printed an error message to the stderr

```
$ cat outfile

File: 'a.pdf'
Size: 0   Blocks: 0  IO Block: 4096   regular empty file
Device: 801h/2049d Inode: 266636      Links: 1
Access: (0664/-rw-rw-r--) Uid: (1000/younix) Gid: (1000/younix)
Access: 2017-08-16 23:05:20.481275377 -0500
Modify: 2017-08-16 23:05:20.481275377 -0500
Change: 2017-08-16 23:05:20.481275377 -0500
Birth: -
```

If we wanted the error message to also go to the outfile we do this:

```
$ stat aa.pdf bd > outfile 2>&1
$ cat outfile

File: 'a.pdf'
Size: 0   Blocks: 0  IO Block: 4096   regular empty file
Device: 801h/2049d Inode: 266636      Links: 1
Access: (0664/-rw-rw-r--) Uid: (1000/younix) Gid: (1000/younix)
Access: 2017-08-16 23:05:20.481275377 -0500
Modify: 2017-08-16 23:05:20.481275377 -0500
Change: 2017-08-16 23:05:20.481275377 -0500
Birth: -
stat: cannot stat 'bd': No such file or director
```

The breakdown of the command is as follows: We first told the shell to duplicate the outfile to the stdout, so anything written to the output has to go to the outfile; We are then asking that the stderr be redirected to the stdout which is already being redirected to outfile

Checkpoint 0.4 What happens if we were to switch the order of redirections in the above example:

```
$ stat aa.pdf 2>&1 > outfile
```

Solution: [1.4.]

0.3.6 Filters

Just as we can redirect input or output of a command to a file, we can also direct the output of one *command* to the be the input of another *command* by using **pipes**. For example, say we want to get a count of the number of subdirectories in a given directory:

```
$ mkdir cdir
$ ls -l

total 12
-rw-rw-r-- 1 younix younix    0 Aug 16 23:14 aa.pdf
-rw-rw-r-- 1 younix younix    0 Aug 16 23:05 a.pdf
drwxrwxr-x 2 younix younix 4096 Aug 16 23:11 bdir
drwxrwxr-x 2 younix younix 4096 Aug 17 12:37 cdir
-rw-rw-r-- 1 younix younix    0 Aug 16 01:33 foo
-rw-rw-r-- 1 younix younix  402 Aug 17 11:58 outfile

$ ls -l | grep "^d" | wc -l
2
```

The equivalent without pipes would have been:

```
$ ls -l > tmpfile1
$ grep "^d" < tmpfile1 > tmpfile2
$ wc -l < tmpfile2
$ rm tmpfile1 tmpfile2
```

The regular expression pattern `"^d"` *passed to grep tells it to search for lines that start with the letter d*

0.3.7 Background Process

The normal behavior of command execution at the shell prompt is for the shell to run the command and then return and be ready for the next command. Therefore, if we were to run a command that takes a long time or a command that does not terminate then the shell will wait for a long time or never return to the shell prompt as the case may be. The shell offers a way to run a command and put in the "background" so the shell can return immediately with the command prompt (i.e., not wait for the the command to finish). This is done by placing the ampersand (&) operator at the end of the command.

```
$ mkdir cdir
$ ls -l

total 12
-rw-rw-r-- 1 younix younix    0 Aug 16 23:14 aa.pdf
-rw-rw-r-- 1 younix younix    0 Aug 16 23:05 a.pdf
drwxrwxr-x 2 younix younix 4096 Aug 16 23:11 bdir
drwxrwxr-x 2 younix younix 4096 Aug 17 12:37 cdir
-rw-rw-r-- 1 younix younix    0 Aug 16 01:33 foo
```

```
-rw-rw-r-- 1 younix younix  402 Aug 17 11:58 outfile

$ ls -l | grep "^d" | wc -l
2
```

The equivalent without pipes would have been:

```
$ ls -l > tmpfile1
$ grep "^d" < tmpfile1 > tmpfile2
$ wc -l < tmpfile2
$ rm tmpfile1 tmpfile2
```

The regular expression pattern `"^d"` *passed to grep tells it to search for lines that start with the letter d*

0.3.8 Job Control

Every running entity in Unix does so within a program context called a *process.* Recall that the first process that ran was the *init* process (aka *systemd*) which has a process id value of 1. All processes after that are either created by the *init* process or by one of the processes created by *init* or one of the processes created by a process that was created by the *init* process and so on. Interacting with these processes is the essence of job control.

The Unix terminology the *init* process is the common ancestor to all processes. When a process `P` creates another process `C`, we refer to process `C` as the *child* of the *parent* process `P`. Use `pstree` to view the process tree rooted of course at `systemd`.

The bash shell one runs inside a Terminal is itself a process. Further, for every command one types in the shell, the shell creates a child process and executes the command inside the context of the child process.

We will learn more details about processes but for now we just see, how to view processes and their resource usage and, send signals to processes. The two commands to do these tasks are `ps` and `kill` respectively. If you want to get a live updated view of processes and their resource usage use the `top` utility. Two other commands that are easier to use counterparts of `ps` and `kill` are `pgrep` and `pkill`.

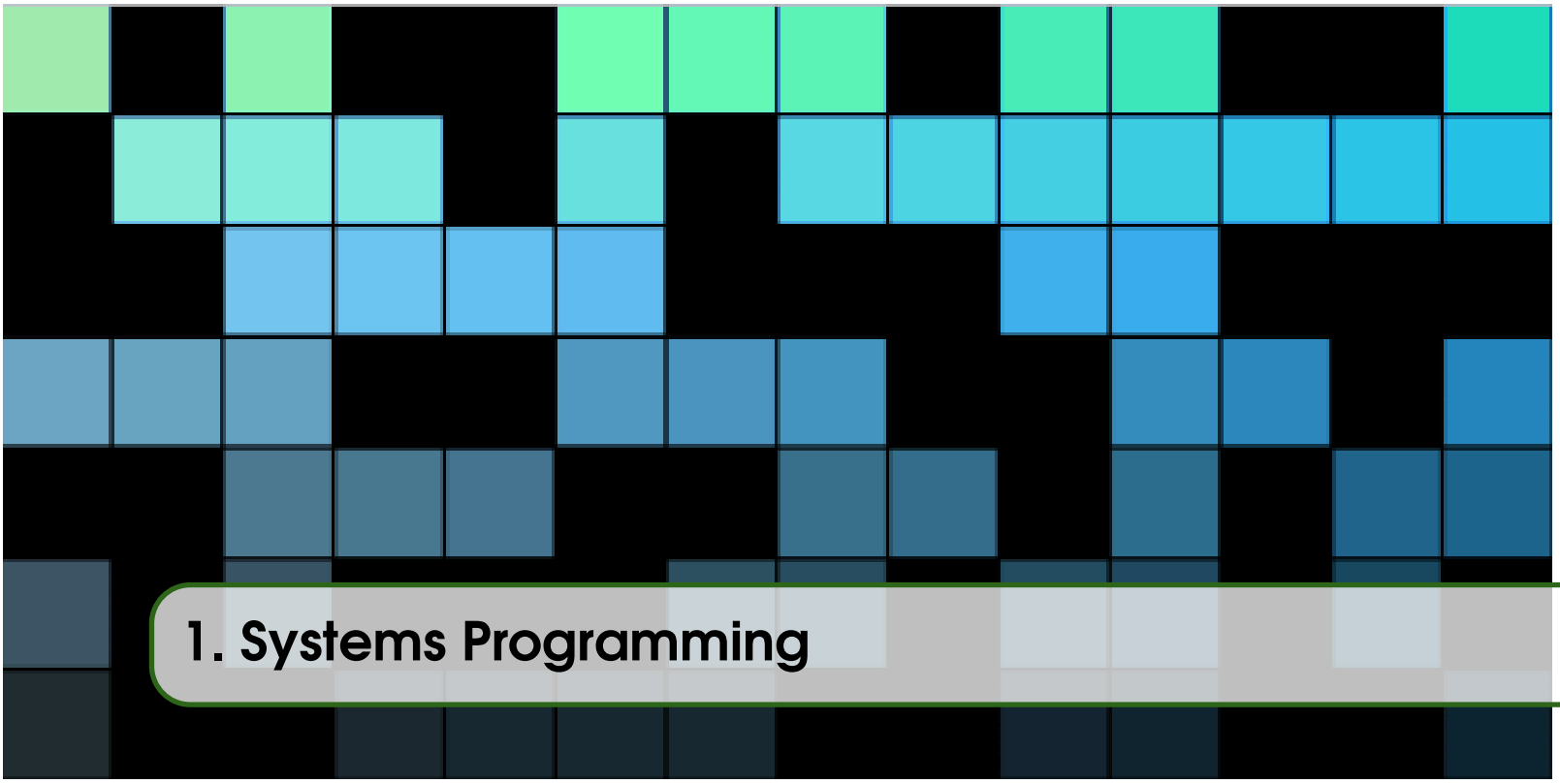
```
$ ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 14:49 ?        00:00:01 /sbin/init splash
root         2     0  0 14:49 ?        00:00:00 [kthreadd]
root         3     2  0 14:49 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 14:49 ?        00:00:00 [kworker/0:0H]
root         7     2  0 14:49 ?        00:00:01 [rcu_sched]
...
```

```
...

To get the pid's of by using their names use pgrep
$ pgrep bash


1975
1998
2045

Gives the pid's of all the bash shells running
```

The command `kill` is a misnomer because its use goes further than just terminating a process. It is used to send a signal to a process, where `SIGKILL` is just one of many possible signals. Running `kill -l` gives you the list of all signals you can send to a process. The most important ones are `SIGHUP`, `SIGINT` `SIGTERM` and `SIGKILL`. The `SIGHUP` signal is typically used to get a process' attention to make it do something its preprogrammed to do demand. For example, say you modified the configuration file for your apache webserver (the daemon `httpd`) and would like the server to reload the updated configuration. Instead of restarting the server (stop followed by start), you can send it a `SIGHUP`.

```
$ pgrep httpd 1028
$ kill -SIGHUP 1028
```

Alternatively, one can use the one line equivalent: `pkill -SIGHUP httpd`, which assumes there is only one process with the name `httpd`.



1. Systems Programming

An operating system serves as an interface between application programs and the underlying hardware. This interface (API) is a collection of *system calls* and, writing programs that make use of this API is referred to as systems programming. When the API is based on a standard like POSIX, it makes your code portable across all operating systems that conform to the standard. Popular operating systems like, Linux, FreeBSD, MacOSX and Windows are all POSIX compliant. Though we will focus on Linux here, all the concepts covered here translate to these other operating systems as well.

We will use the C programming language for our systems programming. Other programming languages like, python, perl and java can also be used but programming is C has the advantage that the Linux OS itself is written in C. The GNU C compiler, `gcc` is at the heart of software development.

1.1 C Basics

I am assuming the reader has programming background in C and is familiar with the following concepts:

- Pre-processor directives, `#include`, `#define` and macros:

```
#include <stdio.h>
#define INT\_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

- C variable declarations, typedefs, assignment(`variable = expression`), expressions.

- `if-else`, `while`, and `for` statements.
- Pointers (address-of: & and dereferencing: `*`), Call-by-value and Call-by-reference:

```
void swap(int* a, int* b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}
```

```
int x = 42;
int y = 22;
swap(&x, &y);
```

The values of `x` andy after the swap call are `22` and `42` respectively.

- Arrays - Declaration and use

```
int A[10]; // A is array of 10 int's on the stack
int* B = calloc(10, sizeof(int));
           // Bis array of 10 int's on the heap
int *C[3]; // C is an array of 3 pointers to int
int (*D)[3]; // D is a pointer to an array of 3 ints
char
```

- Structs for composite data types

```
struct foo_s {
  int a;
  char b;
};

struct bar_s {
  char ar[10];
  foo_s baz;
};

bar_s biz;          // bar_s instance
biz.ar[0] = 'a';
biz.baz.a = 42;
bar_s* boz = &biz; // bar_s ptr
boz->baz.b = 'b';
```

- Strings and `<string.h>` library of functions, `strcat`, `strcmp`, `strlen`, `strcpy` and `strtok`.

1.2 Program Development

Lets take a simple example to demonstrate the procedure we will use for software development involving editing, compiling and running our software.

Here is our first example involving two C files, `hello.c` and `PrintDate.c`. The first has the `main()` function and the second has a function called `Today()` that is called from `main()`. You may use editors like `emacs` or `vim` to create these files and edit their contents.

```c
/**
 * @brief  Program to demonstrate Development Process
 * This is part of an example with two other files
 * Today.c, Makefile
 * that are used to demonsrate program developement
 */
#include <stdio.h>

extern void Today(void);
int main()
{
  printf("Howdy_Sys_Programmer\n");
  Today();
}
```

Listing 1.1: hello.c

```c
#include <stdlib.h>

/**
 * @brief  Prints today's date
 * @param  none
 * @return none
 */
extern void Today(void){
  system("date");
}
```

Listing 1.2: PrintDate.c

We can compile these files to form our application (executable called `hello`) and then run it in the shell. The first command invokes the *gcc* compiler to build our executable (`-o hello`), telling it compile both files `hello.c` and `PrintDate.c` because `hello` depends on both these files.

```
$ gcc -o hello hello.c PrintDate.c
$ ./hello
Howdy Sys Programmer
Fri 18 Aug 02:20:39 CDT 2017
```

We can automate the compilation process using makefile which also has the added benefit of: (i) specify dependencies, (ii) compile only when needed, (iii) compile only those files that changed and (iv) customize for a particular environment. We will use the following Makefile for our simple example.

```
1  # The program we want to build, what it depends on
2  # and how to build it
3  hello : hello.o PrintDate.o
4          gcc -o hello hello.o PrintDate.o
5  # hello.o depends on hello.c and its built
6  # by running the command: gcc -c hello.c
7  hello.o : hello.c
8          gcc -c hello.c
9  # build PrintDate.o
10 PrintDate.o : PrintDate.c
11         gcc -c PrintDate.c
12 # What to do if make is run as:
13 #   make clean
14 # remove all object and executables
15 clean:
16         rm *.o hello
```

Listing 1.3: Makefile

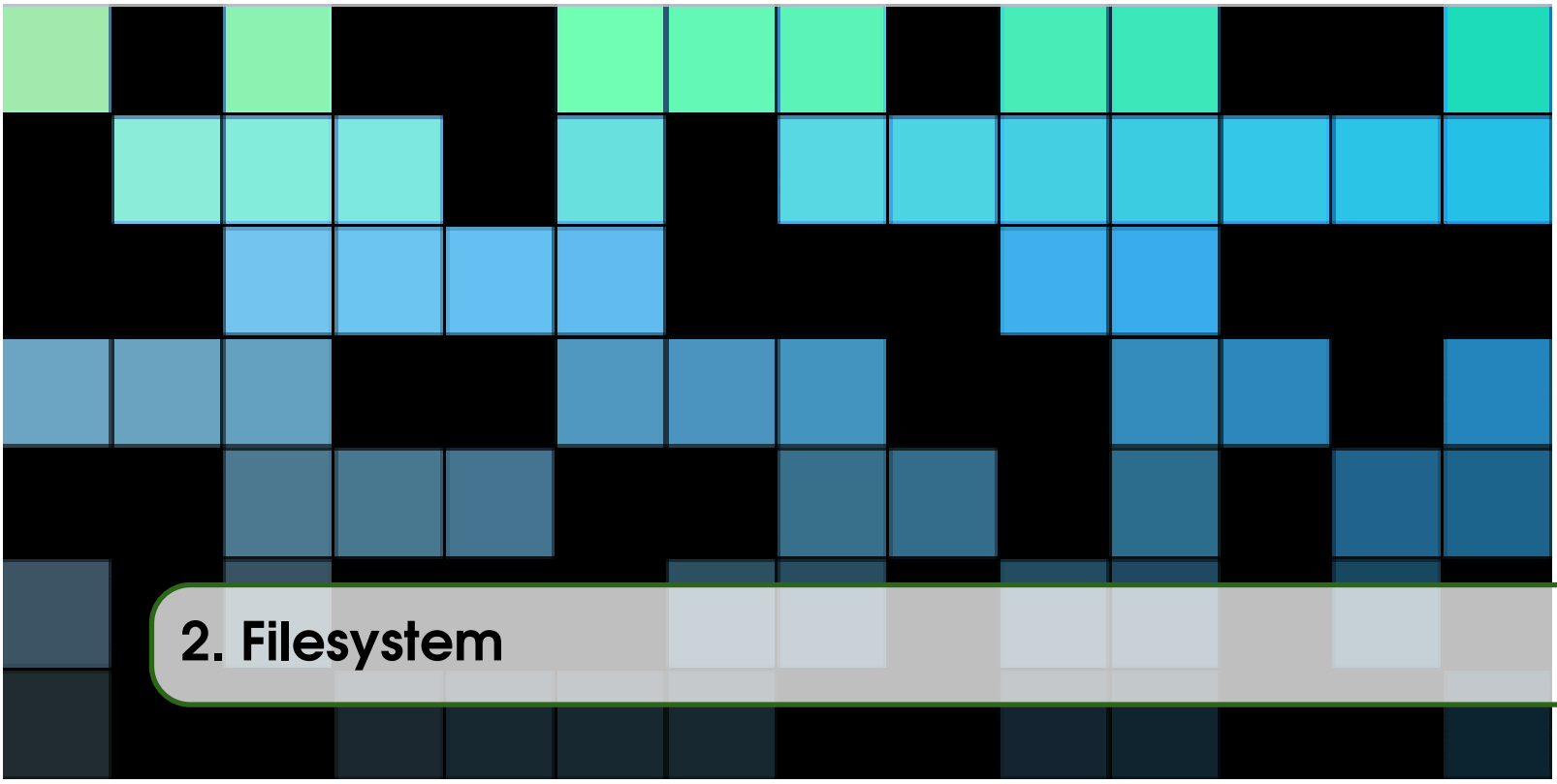
```
$ make
gcc -c hello.c
gcc -c PrintDate.c
gcc -o hello hello.o PrintDate.o
```

Say we edit the file hello.c and change the string Howdy to Hello. Now:

```
$ make
```

```
gcc -c hello.c
gcc -o hello hello.o PrintDate.o
```

skips building PrintDate.o as its timestamp is newer than PrintDate.c, which implies that it is up to date



2. Filesystem

2.1 File Descriptors vs File Pointers

2.2 Low-level I/O

open, read, write, lseek, close, unlink

2.3 Standard I/O

fopen, fread, fwrite, fseek, fclose, fflush, fscanf

2.4 Miscellaneous Calls

dup2, getc, putc, getchar, putchar, readline

3. Processes and Threads

A Unix *process* is a fundamental entity that represents a program in a state of execution. A program file located on a storage media (say disk) is a static entity which when executed triggers the creation of a process which has "state" in memory. There are two main pieces that make up this state:

- The Program Control Block (PCB) associated with this process that is a data structure maintained by the Kernel.
- The address space that maps:
  - the code, heap, stack segments that are private to the process
  - and, the kernel

3.1 Fork and exec

what you have to #include: sys/types.h, unistd.h, sys/wait.h types used: pid_t for identifying processes via PID functions:

```
pid_t fork(void)
The caller process creates a clone of itself which will run independently after the
pid_t wait(int *status)
blocks until one of the children exits, stores its exit status in status and return
pid_t waitpid(pid_t wpid, int *status, int options)
Does the same as wait(), but waiting for a specific child identified by wpid. When wp
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage)
```

```
this is the most powerful function for getting child processes informations. It
pid_t getpid(void)
Returns the Process ID of the caller. Always successful.
pid_t getppid(void)
Returns the parent's Process ID of the caller. Always successful.
macros (for waitpid() and wait4()):
WIFEXITED(status)
returns true (non-zero) if status denotes a normal process exit status
WIFSIGNALED(status)
returns true if the process terminated after receiving a signal
WIFSTOPPED(status)
returns true if the process is not running but has been stopped, and can be res
WEXITSTATUS(status)
returns the process exit status information carried by status
WTERMSIG(status)
returns the number of the signal which made the process terminate. If the proce
WSTOPSIG(status)
returns the number of the signal which put the process into stop.
```

3.2 Process Groups

3.3 pThreads

4. Inter Process Communication

4.1 Signals

```c
/* file: sig_ex1.c
   author: Ramesh Yerraballi
   This is a simple example of signals
   The program sets up functions that will be called in response to the two
   signals (SIGINT and SIGTSTP). The actions performed are print and exit or just print
*/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/*
These are the POSIX signals, their numbers and what they are for
      Signal     Signo     Action   Comment
       SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2       Term    Interrupt from keyboard (Ctrl-C)
       SIGQUIT       3       Core    Quit from keyboard
       SIGILL        4       Core    Illegal Instruction
       SIGABRT       6       Core    Abort signal from abort(3)
       SIGFPE        8       Core    Floating point exception
       SIGKILL       9       Term    Kill signal
       SIGSEGV      11       Core    Invalid memory reference
```

```
22        SIGPIPE      13       Term    Broken pipe: write to pipe with no
23                                       readers
24        SIGALRM      14       Term    Timer signal from alarm(2)
25        SIGTERM      15       Term    Termination signal
26        SIGUSR1   30,10,16    Term    User-defined signal 1
27        SIGUSR2   31,12,17    Term    User-defined signal 2
28        SIGCHLD   20,17,18    Ign     Child stopped or terminated

30        SIGCONT   19,18,25    Cont    Continue if stopped
31        SIGSTOP   17,19,23    Stop    Stop process
32        SIGTSTP   18,20,24    Stop    Stop typed at terminal (Ctrl-Z)
33        SIGTTIN   21,21,26    Stop    Terminal input for background process
34        SIGTTOU   22,22,27    Stop    Terminal output for background process

36        The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
37 To send a signal to a process from the shell:
38    prompt>> kill -<Signal or Signo> <process_pid>
39 */

41 static void sig_int(int signo) {
42   printf("caught SIGINT\n");
43   exit(0);
44 }

46 static void sig_tstp(int signo) {
47         printf("caught SIGTSTP\n");
48 }

50 static void sig_handler(int signo) {
51   switch(signo){
52   case SIGINT:
53         printf("caught SIGINT\n");
54         exit(0);
55         break;
56   case SIGTSTP:
57         printf("caught SIGTSTP\n");
58         break;
59   }

61 }

63 int main(void) {
64 // You can register separate signal handlers like here:
```

```
65   /*  if (signal(SIGINT, sig_int) == SIG_ERR)
66     printf("signal(SIGINT) error");
67   if (signal(SIGCHLD, sig_chld) == SIG_ERR)
68   printf("signal(SIGCHLD) error"); */

70   // OR you can have a commong signal handler and do the appropriate action
71   // based on the signal number
72   if (signal(SIGINT, sig_handler) == SIG_ERR)
73     printf("signal(SIGINT) error");
74   if (signal(SIGTSTP, sig_handler) == SIG_ERR)
75     printf("signal(SIGTSTP) error");

77   while(1){}
78 }
```

Listing 4.1: signals/sigex1.c

```
1  /* file: sig_ex2.c
2     Author: Ramesh Yerraballi
3     This is a more complex example of signals
4     Create a child process and communicate with it using a pipe
5     The parent writes to the pipe and child reads from the pipe.
6     What the parent writes is what the user is typing on the keyboard
7     when the child read two identical characters in a row
8     it sends a SIGUSR1 signal to the parent. The parent acknowleges
9     the signal and quits
10 */
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <signal.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <sys/wait.h>

18 static void sig_handler(int signo) {
19   printf("doh\n");
20   exit(0);

22 }

24 int main(void) {

26   int pipefd[2], status;
```

```
27    char ch[2]={0,0},pch=128;

29    if (pipe(pipefd) == -1) {
30      perror("pipe");
31      exit(-1);
32    }

34    int ret = fork();
35    if (ret > 0){
36      // Parent
37      if (signal(SIGUSR1, sig_handler) == SIG_ERR)
38      printf("signal(SIGUSR1)␣error");
39      close(pipefd[0]); //close read
40      while (read(STDIN_FILENO,ch,1) != 0){
41        write(pipefd[1],ch,1);
42      }
43      wait(&status);

45    }else {
46      // Child
47      close(pipefd[1]); // close the write end
48      while (read(pipefd[0],ch,1) != 0){
49        printf("%c",ch[0]);sync();
50        if (pch == ch[0]){
51          //printf("sending SIGUSR1 to parent\n");
52          kill(getppid(),SIGUSR1);
53          exit(0);
54        }
55        pch = ch[0];
56      }

58    }

60  }
```

Listing 4.2: signals/sigex2.c

```
1   /* file sig_ex3.c: This is a more complex example of signals
2      Author: Ramesh Yerraballi
3      Attempt to mimic:
4        prompt>> top | grep firefox
5      The parent creates  a pipe and two child processes with the
6      write end of the pipe serving as the stdout for top and
```

```
7     the read end serving as the stdin for grep.
8     The first child that exec's top creates a new session with itself a member leader
9     of the process group in it. The process group's id is same as the child's pid (pid_ch1).
10    The second child that exec's grep joins the process group that the first child created
11    Now when a Ctr-c is pressed the parent relays a SIGINT to both children using
12       kill(-pid_ch1,SIGINT); alternative you could call killpg(pid_ch1,SIGINT);
13    The two child processes receive the SIGINT and their default behavior is to terminate.
14    Once they do that the parent reaps their exit status (using wait), prints and exits.
15    When a  Ctrl-z is pressed the the parent relays a SIGTSTP to both children using
16       kill(-pid_ch1,SIGTSTP);
17    The parent's waitpid() call unblocks when the child receives the STOP signal. The parent
18    waits for 4 secs and resumes the the child that STOPped. This happens once for each of
19    the two children.
20 */
21 #include <stdio.h>
22 #include <stdlib.h>
23 #include <signal.h>
24 #include <unistd.h>
25 #include <sys/types.h>
26 #include <sys/wait.h>
27 #include <string.h>

29 int pipefd[2];
30 int status, pid_ch1, pid_ch2, pid;

32 static void sig_int(int signo) {
33   printf("Sending signals to group:%d\n",pid_ch1); // group id is pid of first in pipeline
34   kill(-pid_ch1,SIGINT);
35 }
36 static void sig_tstp(int signo) {
37   printf("Sending SIGTSTP to group:%d\n",pid_ch1); // group id is pid of first in pipeline
38   kill(-pid_ch1,SIGTSTP);
39 }

41 int main(void) {

43   char ch[1]={0};

45   if (pipe(pipefd) == -1) {
46     perror("pipe");
47     exit(-1);
48   }
```

```
50  pid_ch1 = fork();
51  if (pid_ch1 > 0){
52    printf("Child1 pid = %d\n",pid_ch1);
53    // Parent
54    pid_ch2 = fork();
55    if (pid_ch2 > 0){
56      printf("Child2 pid = %d\n",pid_ch2);
57      if (signal(SIGINT, sig_int) == SIG_ERR)
58        printf("signal(SIGINT) error");
59      if (signal(SIGTSTP, sig_tstp) == SIG_ERR)
60        printf("signal(SIGTSTP) error");
61      close(pipefd[0]); //close the pipe in the parent
62      close(pipefd[1]);
63      int count = 0;
64      while (count < 2) {
65        // Parent's wait processing is based on the sig_ex4.c
66        pid = waitpid(-1, &status, WUNTRACED | WCONTINUED);
67        // wait does not take options:
68        //    waitpid(-1,&status,0) is same as wait(&status)
69        // with no options waitpid wait only for terminated child processes
70        // with options we can specify what other changes in the child's status
71        // we can respond to. Here we are saying we want to also know if the child
72        // has been stopped (WUNTRACED) or continued (WCONTINUED)
73        if (pid == -1) {
74          perror("waitpid");
75          exit(EXIT_FAILURE);
76        }

78        if (WIFEXITED(status)) {
79          printf("child %d exited, status=%d\n", pid, WEXITSTATUS(status));count++;
80        } else if (WIFSIGNALED(status)) {
81          printf("child %d killed by signal %d\n", pid, WTERMSIG(status));count++;
82        } else if (WIFSTOPPED(status)) {
83          printf("%d stopped by signal %d\n", pid,WSTOPSIG(status));
84          printf("Sending CONT to %d\n", pid);
85          sleep(4); //sleep for 4 seconds before sending CONT
86          kill(pid,SIGCONT);
87        } else if (WIFCONTINUED(status)) {
88          printf("Continuing %d\n",pid);
89        }
90      }
91      exit(1);
92    }else {
```

```
93        //Child 2
94        sleep(1);
95        setpgid(0,pid_ch1); //child2 joins the group whose group id is same as child1's pid
96        close(pipefd[1]); // close the write end
97        dup2(pipefd[0],STDIN_FILENO);
98        char *myargs[3];
99        myargs[0] = strdup("grep");   // program: "grep" (word count)
100       myargs[1] = strdup("firefox");   // argument: "firefox"
101       myargs[2] = NULL;           // marks end of array
102       execvp(myargs[0], myargs);  // runs word count
103     }
104   } else {
105       // Child 1
106     setsid(); // child 1 creates a new session and a new group and becomes leader -
107               //   group id is same as his pid: pid_ch1
108     close(pipefd[0]); // close the read end
109     dup2(pipefd[1],STDOUT_FILENO);
110     char *myargs[2];
111     myargs[0] = strdup("top");   // program: "top" (writes to stdout which is now pipe)
112     myargs[1] = NULL;
113     execvp(myargs[0], myargs);  // runs top
114   }
115 }
```

Listing 4.3: signals/sigex3.c

4.2 Pipes

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>

5  int
6  main(int argc, char *argv[])
7  {
8      int pipefd[2];
9      pipe(pipefd);

11     dup2(0,pipefd[0]);
12     dup2(1,pipefd[1]);

14     while(1){
```

```
15        putchar(getchar());
16      }
17  }
```

Listing 4.4: pipes/echo.c

```
1  /* Example demonstrates a simple use of a Pipe to communicate between a parent and a c
2     A pipe is a uni-directional communication mechanism.
3     Here the parent writes a string to the child, which the child prints to the screen
4     Original Source: Advanced Programming in the Unix Enviroment, by Richard Stevens
5                      http://www.apuebook.com/apue3e.html
6     Annotated by: Ramesh Yerraballi
7     System Calls used:  fork, wait, pipe
8  */
9  #include <sys/wait.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <unistd.h>
13 #include <string.h>

15 int
16 main(int argc, char *argv[])
17 {
18     int pipefd[2];
19     pid_t cpid;
20     char buf;

22     if (argc != 2) {
23         fprintf(stderr, "Usage: %s <string>\n", argv[0]);
24         exit(EXIT_FAILURE);
25     }
26     // pipe takes a pointer to a 2-element array (pipefd[2]) as input and returns the
27     // the read and write end of the pipe in pipefd[0] and write end in pipefd[1]
28     if (pipe(pipefd) == -1) {
29         perror("pipe");
30         exit(EXIT_FAILURE);
31     }

33     cpid = fork();
34     if (cpid == -1) {
35         perror("fork");
36         exit(EXIT_FAILURE);
37     }
```

```
39      if (cpid == 0) {    /* Child reads from pipe */
40          close(pipefd[1]);          /* Closes unused write end */
41
42          while (read(pipefd[0], &buf, 1) > 0)
43              write(STDOUT_FILENO, &buf, 1);
44
45          write(STDOUT_FILENO, "\n", 1);
46          close(pipefd[0]);
47          _exit(EXIT_SUCCESS);
48
49      } else {            /* Parent writes argv[1] to pipe */
50          close(pipefd[0]);          /* Close unused read end */
51          write(pipefd[1], argv[1], strlen(argv[1]));
52          close(pipefd[1]);          /* Reader will see EOF */
53          wait(NULL);                /* Wait for child */
54          exit(EXIT_SUCCESS);
55      }
56  }
```

Listing 4.5: pipes/pipeex1.c

```
1   /* Example demonstrates use of a pipe that mimics the use of pipes in a shell
2      Takes the name of the file you want to view using a pager program
3      like "more" or "less"; Displays the file's contents by: Creating a
4      child process that execs the pager. The parent passes the name of
5      the file to the child
6      Original Source: Advanced Programming in the Unix Enviroment, by Richard Stevens
7                       http://www.apuebook.com/apue3e.html
8      Annotated by: Ramesh Yerraballi
9      System Calls used:  fork, exec, wait, pipe, dup2
10
11  */
12
13  #include <sys/wait.h>
14  #include <stdio.h>
15  #include <stdlib.h>
16  #include <unistd.h>
17  #include <string.h>
18
19  #define MAXLINE 4096                    /* max line length */
20
21  #define DEF_PAGER       "/bin/more"             /* default pager program */
```

```
23 int
24 main(int argc, char *argv[])
25 {
26     int         n;
27     int         fd[2];
28     pid_t       pid;
29     char        *pager, *argv0;
30     char        line[MAXLINE];
31     FILE        *fp;

33     if (argc != 2) {
34         fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
35         exit(EXIT_FAILURE);
36     }

38     if ((fp = fopen(argv[1], "r")) == NULL)
39         fprintf(stderr,"can't open %s", argv[1]);
40     if (pipe(fd) < 0)
41             fprintf(stderr,"pipe error");

43     if ((pid = fork()) < 0) {
44         fprintf(stderr,"fork error");
45     } else if (pid > 0) {
46         close(fd[0]);           /* close read end */
47         /* parent copies contents of file given in argv[1] to pipe */
48         while (fgets(line, MAXLINE, fp) != NULL) {
49             n = strlen(line);
50             if (write(fd[1], line, n) != n)
51                 fprintf(stderr,"write error to pipe");
52         }

54         if (ferror(fp))
55             fprintf(stderr,"fgets error");

57         close(fd[1]);   /* close write end of pipe for reader */

59         if (waitpid(pid, NULL, 0) < 0)
60             fprintf(stderr,"waitpid error");
61         exit(0);
62     } else {
63         close(fd[1]);   /* close write end */
64         if (fd[0] != STDIN_FILENO) {
```

```
65              if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
66                  fprintf(stderr,"dup2 error to stdin");
67              close(fd[0]);       /* don't need this after dup2 */
68          }

70          /* get arguments for execl() */
71          if ((pager = getenv("PAGER")) == NULL)
72                  pager = DEF_PAGER;
73          if ((argv0 = strrchr(pager, '/')) != NULL)
74              argv0++;            /* step past rightmost slash */
75          else
76              argv0 = pager;      /* no slash in pager */

78          if (execl(pager, argv0, (char *)0) < 0)
79              fprintf(stderr,"execl error for %s", pager);
80      }
81      exit(0);
82  }
```

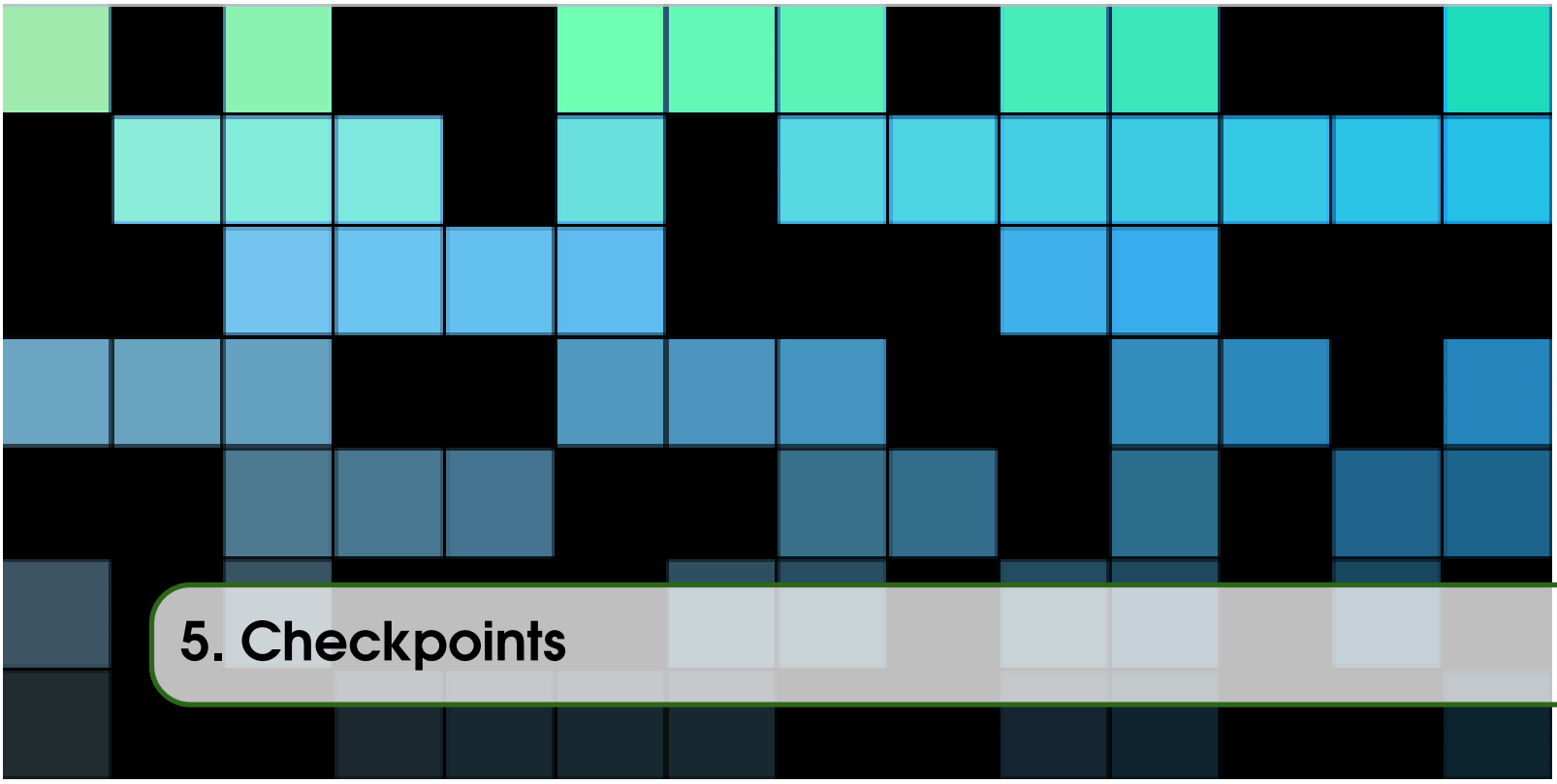
Listing 4.6: pipes/pipeex2.c

4.3 Unix Domain Sockets

4.4 Shared Memory

4.5 Network Sockets

Appendix



5. Checkpoints

1. Chapter 1
   1.1. The tilde symbol ~
   1.2. Yes, the only exception is the root (`/`)
   1.3. Your `PATH` does not have `.` in it. If you don't want to bother with changing your `PATH` environment variable then, you can run it like so:
   ```
   $ ./dothis
   ```

   1.4. Only the stdout will be redirected to outfile because the stderr was duplicated from the stdout before the stdout was redirected to outfile. So, the order of redirections matters.

Book References