If our program will run on Linux, we should learn about Linux

Process abstraction. Daemons

How programs learn what to do: rc files, environment variables, arguments

Along the way... many useful Linux commands and bash features
RECAP

We saw that when our word-count program was running, parallelism offered a way to get much better performance from the machine, as much as a 30x speedup for this task.

In fact, Linux systems often have a lot of things running on them, in the background (meaning, “not talking to the person typing commands on the console.”)
STUFF THAT WAS RUNNING ON A SERVER

At a random moment on a server similar to the setup in the CS engineering server pool, what happens to be running?

Ken made these next two slides to show you
kern@ken:/etc#$: lish
Ssh compute30-fractus.cs.cornell.edu
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.0.0-58-generic x86_64)

* Documentation:  https://help.ubuntu.com
* Management:  https://landscape.canonical.com
* Support:  https://ubuntu.com/advantage

System information as of Mon Sep 14 20:23:26 UTC 2020

System load:  1.0  Users logged in:  1
Usage of /:  15.8% of 1.72TB  IP address for enp9s0:  128.84.139.25
Memory usage:  2%  IP address for enp17s0F1:  102.168.9.30
Swap usage:  0%  IP address for ib0:  192.168.99.30
Processes:  721

* Kubernetes 1.19 is out! Get it in one command with:

    sudo snap install microk8s --channel=1.19 --classic

    https://microk8s.io/ has docs and details.

* Canonical Livepatch is available for installation.
  - Reduce system reboots and improve kernel security. Activate at:
    https://ubuntu.com/livepatch

61 packages can be updated.
1 update is a security update.

*** System restart required ***
Last login: Mon Sep 14 20:21:30 2020 from 10.41.250.60
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

kern@compute30:~$ who
lorezo_roza pts/0 2020-08-24 14:35 (128.84.139.10)
lorezo_roza pts/1 2020-09-11 14:13 (128.84.139.10)
ken pts/2 2020-09-14 20:23 (10.41.250.60)
kern@compute30:~$
WHAT’S WITH THE ?????? STUFF?

... apparently those are escaped newline characters!

In fact any “non-printing” characters are shown as ?
LET'S SUMMARIZE SOME OF WHAT WE SAW

In addition to the Linux operating system “kernel”, Linux had many helper programs running in the background.

We used the term daemon programs for these. The term is a reference to physics, but a bit obscure.

A daemon program is launched during startup (or periodically) and doesn’t connect to a console. It lives in the background.
YOU CAN ALSO CREATE BACKGROUND TASKS OF YOUR OWN

One way to do this is with a command called “nohup”, which means “when I log out ("hang up"), leave this running.”

A second is with a command named “disown”.

- When you log out, bash kills any background jobs that you still own.
- If you “disown” a job, it leaves it running
ONE REASON FOR DAEMONS: PERIODIC TASKS

In production systems, many things need to happen periodically.

Linux and C++ have all sorts of features to help.

- Within Linux, a tool called “cron” (for “chronological”) runs jobs on a schedule that you can modify or extend.
- Example: Once every hour, check for new photos on the camera and download them.
HOW CRON WORKS

There is a file in a standard location called the “crontab”, meaning “table of jobs that run chronologically”

Each line in the file uses a special notation to designate when the job should run and what program to launch.

The program itself could be in any language and can even be a Linux “bash script” (also called a “shell script”).
HOW AT WORKS

Very similar to cron, but for a one-time command

The “atd” waits until the specified time, then runs it

Whereas cron is controlled from the crontab file, at is used at the command-line.
HOW DO THESE PROGRAMS KNOW WHAT WE WANT THEM TO DO?

On Linux, programs have three ways to discover runtime parameters that tell them what to do.

- Arguments provided when you run the program, on the command line
- Configuration files, specific to the program, that it can read to learn parameter settings, files to scan, etc.
- Linux environment variables. These are managed by bash and can be read by the program using “getenv” system calls.
PROGRAMS CONTROLLED BY CONFIGURATION FILES

In Linux, many programs use some sort of configuration file, just like cron is doing. Some of those files are hidden but you can see them if you know to ask.

➢ In any directory, hidden files will simply be files that start with a name like “.bashrc”. The dot at the start says “invisible”

➢ If you use “ls -a” to list a directory, it will show these files. You can also use “echo .*” to do this, or find, or ....
A FEW COMMON HIDDEN FILES

Bash replaces “~” with the pathname to your home directory

`~/.bashrc` – The Bourne shell (bash) initialization script

`~/.vimrc` – A file used to initialize the vim visual editor

`~/.emacs` – A file used to initialize the emacs visual editor

`/etc/init.d` – When Linux starts up, the files here tell it how to configure the entire computer

`/etc/init.d/cron` – Used by cron to track periodic jobs
ENVIRONMENT VARIABLES

The bash configuration file is used to set the environment variables.

Examples of environment variables on Ubuntu include

- **HOME**: my “home directory”
- **USER**: my login user-name
- **PATH**: A list of places Ubuntu searches for programs when I run a command
- **PYTHONPATH**: Where my version of Python was built
ENVIRONMENT VARIABLES

The bash configuration file is used to set the environment variables. Examples of environment variables on Ubuntu include:

- HOME: my “home directory”
- USER: my login user-name
- PATH: A list of places Ubuntu searches for programs when I run a command
- PYTHONPATH: Where my version of Python was built

Other versions of Linux, like CentOS, RTOS, etc might have different environment variables, or additional ones. And different shells could use different variables too!
EXAMPLE, FROM KEN’S LOGIN

HOSTTYPE=x86_64
USER=ken
HOME=/home/ken
SHELL=/bin/bash
PYTHONPATH=/home/ken/z3/build/python/
PATH=/home/ken/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/games:/usr/local/games
SO... LET’S WALK THROUGH THE SEQUENCE THAT CAUSES THESE TO BE “USED”

We will review

1) How Linux boots when you restart the computer
2) How bash got launched (this is when it read .bashrc)
3) How a command like “c++” gets launched
WHEN UBUNTU BOOTS

Ubuntu is a version of Linux. It runs as the “operating system” or “kernel”. But when you start the computer, it isn’t yet running.

Every computer has a special firmware program to launch a special stand-alone program call the “bootstrap” program. In fact this is a 2-stage process (hence “stage ½ bootloader”)

This stand-alone program then reads the operating system binary from a file on disk into memory and launches it.
WHAT ABOUT UBUNTU ON WINDOWS?

Microsoft Windows has a “microkernel” on which they can host Ubuntu as a kind of application. (Same with MacOS, via bootcamp)

In effect, both Windows and Ubuntu are “apps” that in turn can host and run additional programs, all on your machine.

Early virtualization approaches of this kind were slow, but over time they have become highly performant. Now we do this all the time.
UBUNTU LINUX STARTS BY SCANNING THE HARDWARE

Linux figures out how much memory the machine has, what kind of CPU it has, what devices are attached, etc.

It accesses the same disk it booted on to learn configuration parameters and also which devices to activate. For these activated devices, it loads a “device driver”.

Then it starts the “init” daemon.
The init daemon is the “parent” of all other processes that run on an Ubuntu Linux system. /etc/init.d told it what to initially do at boot time.

It launched **cron** and the **at** daemon, and it also launches the application that allows you to log in and have a bash shell connected to your console.

The rlogin daemon allows remote logins, if you configured Ubuntu to permit them. If firewalls and IP addresses allow, you can then use rlogin to remotely connect to a machine, like I did to access compute30 on Fractus.
WHEN YOU LOG IN

The login process sees that “ken” is logging in.

It checks the secure table of permitted users and makes sure I am a user listed for this machine – if not, “goodbye”!

In fact I am, and I prefer the bash shell. So it launches the bash shell, and configures it to take command-line input from my console. Now when I type commands, bash sees the string as input.
BASH INITIALIZES ITSELF

The .bashrc file is “executed” by bash to configure itself for me.

I can customize this (and many people do!), to set environment variables, run programs, etc — it is actually a script of bash commands, just like the ones I can type on the command line.

By the time my command prompt appears, bash is configured.
WHEN WE LAUNCH PROGRAMS...

Bash (or cron, or whatever) looks for the program to launch using the PATH variable as guidance on where to look. A special Linux operation called “fork” followed by “exec” runs it.

The program is now active and will read the environment plus any arguments you provided to know what to do. Some programs fail at this stage because they can’t find a needed file in the places listed in the relevant path, or an argument is wrong.
I log in, and then edit a file using vim (Sagar prefers emacs). So:

1. init ran a login daemon.
2. That daemon launched bash.
3. Bash initialized using .bashrc, then gave a command-line prompt
4. When I ran “vim”, bash found the program and ran it, using PATH to know where to look. “which vim” would tell me which it found.
5. Vim initialized itself, and created a visual editing window for me.
First, just to explain about “prompts”, bash has a command prompt that it shows when it is waiting for a command:

ken@compute30: echo “Hello world”

Even if my slide doesn’t show a prompt, it is really there. You can customize it to show anything you like (your computer name, the folder you are in, etc). On old Linux systems, it was “% ”
First, just to explain about “prompts”, bash has a command prompt that it shows when it is waiting for a command:

```
ken@compute30: echo “Hello world”
```

Even if my slide doesn’t show a prompt, it is really there. You can customize it to show anything you like (your computer name, the folder you are in, etc). On old Linux systems, it was “% ”
In a bash script, you can always set environment variables using the special bash command “export” (or the older “setenv”):

```
export PATH=/bin
```

Normally you want to “add” a directory to path. To do this you expand the old value:

```
export PATH=$PATH:$HOME/myapp/bin
```

This says that in my home directory is a directory myapp/bin with programs I might want to run. Bash will now look there, too.
In fact bash allows a shorthand version too

```
% PATH=$PATH:$HOME/myapp/bin
```

or even

```
% PATH=$PATH:~/myapp/bin
```

# ~ is short for $HOME

Why so many notations? Linux evolved over 40 years... people got tired of typing “export” or “setenv” or $HOME
DIRECTORIES, FILES

Linux organizes files into a tree. Even a directory is actually a special kind of file. Use “ls –l” to see details about a file.

Chdir (“cd”) to enter a directory.
- “/” is the root of the file system tree.
- “.” refers to the current directory.
- “..” is a way to access the parent directory.
- In the bash shell, “~” refers to your home directory.

RULES ABOUT FILE NAMES

Linux directories limit the length of a file name to 255 chars.

The maximum length of a pathname, from the root, is 4096.

Alphanumeric and a few characters like . _ -

Unlike Windows and Mac, don’t use spaces in file names.
Processes

When you launch a process (like from bash), it gets executed and has a process id.

The “ps” and “top” commands let you see what you have running.

You can kill a process in various ways: ^C, kill pid, logging out (there is also a way to prevent this, called “nohup”)
There are hundreds of them!

In fact you have to install them, in batches, because they use so much space if you install everything.

Learn about each command using its “manual” page. Just google it, like “Linux find command” (or “man 1 find”)
COMMANDS ARE REALLY EXECUTABLE FILES: READ/WRITE/EXECUTE FILE “PERMISSIONS”

Each file in Linux has permissions, visible via “ls -l”. Permissions are shown as [dlcb]rwxrwxrwx. The d, if present, means that this file is a directory. The other letters are for special types of files.

The next three are permissions for the user who created the file.

The next three are for other users in the owner’s “group”.

The last three are for users outside these two categories.
SPECIAL FILES (S/D/C/B/R...)  

Linux uses file names to refer to devices like the disk, or your camera (if you attach it) or your computer display and keyboard.

There are also files types with other special meanings:

- Links: a way to give a file a second name (an “alias”)
- c or b: character (keyboard) or block (disk) devices
- r: “raw”. A way to access a device “directly”.
THE PERMISSIONS THEMSELVES

Read means “allowed to see the contents”. For a file, this means the bytes. For a directory, this means you can list the files in the directory.

Write means “allowed to make changes”. For a directory this means creating or deleting files.

Execute is very complicated...
EXECUTE: THEY RAN OUT OF BITS SO THEY GAVE IT MULTIPLE MEANINGS

If the file is a program, execute means “run the program”

If the file is a “shell file”, execute means “launch the bash program (or it could be some other shell), and tell it to run it.

If the file is a directory, “execute” means “can access files in it”. Note: this means you can sometimes read or run a file that you wouldn’t be able to “see” by listing the directory it is in!
Linux has the concept of a “superuser”. Used when installing programs

Running a command using “sudo” can “override” the normal restrictions. You’ll need this to install extra commands.

Be aware that you can also break Linux easily by changing settings or modifying/removing a file that matters.
REMEMBER THE DAEMONS?
KILLING THEM IS RISKY!

Sometimes a computer seems very busy, or even stuck, and novice users will check for what is running and kill it.

With “sudo” you can kill anything! Like a daemon-killing sword…

… but you need to know what you are killing. Linux depends on many of the background daemons!
SOME DIRECTORIES TO KNOW ABOUT

The current working directory: this is where you are right now, and where files created by commands or programs will be put by default.

For example, if you compile fast-wc.cpp and name the executable fast-c, you could run it by typing ./fast.wc.

If “.” is in PATH, then you can just type fast-wc
SOME DIRECTORIES TO KNOW ABOUT

/tmp is a place for programs to put temporary files needed while executing. These are automatically deleted if you forget to do so (on reboot).

/dev/null: a black hole. We’ll see a use for it soon!

A fun one: You can configure Linux to have a temporary file system entirely in memory (“RAM”). Called /ramfs
Linux treats each storage device (including “ramdisk”) as a separate entity.

A storage device can be “raw” meaning “blocks of bytes” or it can have a file system on it (a tree data structure). At boot time there is just one storage device with an active file system.

The “mount” command attaches a storage device with a file system on it to your directory structure, so that you can access the files in it.
MORE DIRECTORIES TO KNOW ABOUT

/bin and /usr/bin: Standard places where programs are put. Of course you can add more places by installing programs or building your own, and modifying the search PATH variable.

/include: The header files for system calls and standard libraries.

/etc, /init.d: Configuration files used by Linux itself, and the ones used by daemons like cron.
HOW DO PEOPLE LEARN THIS STUFF?

Linux is “self documented”! You can buy a book... but no need!

The Linux “man” program is a user manual for Linux, and has sections covering commands (man 1 find, for example), system calls (man 2 open), libraries (man 3) ...

Bash has a “help” command that will print these same pages. help, by itself, lists all available commands. help find would print the man page for the find command.
You’ve seen: **bash, vim/emacs [pick one], cat, ls, chdir, mkdir, rm, rmdir, more, find, tr, sort, uniq, cron, rlogin, c++, which, sudo**

apt and apt-get are used to install packages. Many “missing” things just need to be installed. For example this sequence:

- `ken@compute30% sudo apt-get update` // updates everything
- `ken@compute30% sudo apt-get upgrade` // adds optional features
- `ken@compute30% sudo apt-get install g++` // installs GNU C++ compiler

Ken’s .bashrc file set the prompt to the machine he is on, followed by “% ”
SOME REALLY USEFUL COMMANDS TO LEARN

**ps**: Used to see what processes are running

**who**: Used to see if other people are on this same machine

**top**: Used to see the “heavy hitters” among active processes

**apt/apt-get**: Used to install packages like the GNU C++ compiler, Python, Java, Eclipse

**tr and sed**: Two “editors” controlled by command-line options

**tar**: Makes a single big file from a list of files or a directory

**gzip**: Compresses a big file
SOME REALLY USEFUL COMMANDS TO LEARN

**C++**: Compiler for C++ programs

**gdb**: Debugger used with C++ programs. Requires `c++ -g`

**time**: Measures how long something takes to run.

- It breaks it down: wall-clock time (“real”), time spent running processes (“user”) and time spent in the Linux kernel (“sys”)

**gprof**: Fancy tool to understand where your code was spending time. Requires a special `c++` command-line argument.
-std=c++17 means “permit use of C++ 17 features”
-g means “I’m still debugging”. Don’t combine with –O3
-O3 means “apply heavy optimizations”
-pg means “I plan to run the gprof profiler”
-Wxxx means “warn about xxx…” (many options)
-pthreads means “I’m using C++ threads”
-o xxx means “name the compiled program xxx”

```c
g++ -std=c++17 -O3 -Wall -Wpedantic -pthread -o fast-wc fast-wc.cpp
```
In Linux, each command you execute runs as a “process”. All the commands I showed you run in the “foreground”.

A process will have some source of input (stdin), output (stdout) and some place for error messages (stderr).

We say that a process is in the foreground if console input is currently controlled by that process. A background process can run, but will pause if it reads console input.
HOW TO RUN A BACKGROUND PROCESS

In bash, just give the command line but put a single & at the end.

(Note: double &, as in &&, means something else).

Another option: run a command, then use “^Z” and say “bg”. Bash will freeze the command (^Z), then restart it in the background.
^C VERSUS ^Z

^C kills the foreground process. There is also a command, “kill” to terminate a background process, e.g.: kill %1

^Z “freezes” a process. It halts but is restartable. To restart it, type the process “number” (%1) or “bg” or “fg”.

- bg puts it in the background. You can run other commands.
- fg puts in the foreground. It is connected to the console.
- “jobs” command lists things you’ve put in the background
^S, ^Q, ^O

^S pauses the screen display of output, but not the process.

^Q resumes the screen output.

^O redirects console output to a black hole (/dev/null). ^O is a toggle: typing ^O again restores console output.
Many editors use the “ESC” character to mean “drop out of visual editing mode into command mode”

In vim, “:” lets you do this for a single command.

^D is used to say “no more input”. Applications that read console input will see an “end of file”
FILE NAME EXTENSIONS

In Windows and Mac systems, we get used to the idea that files have types like “powerpoint” (name.pptx), PDF (name.pdf), image (name.jpg or name.jpeg).

In Linux, file name extensions are optional, but some are common, like name.cpp, name.h or name.hpp, etc.

You can rename a file: Many people rename a.out (default executable name) with something sensible like “myWordCount”
If we write

```bash
find . | wc
```

This means “find all files in this directory and its children and list file names. Here we piped the output into the “wc” command, which will counts lines (the number of files!) and characters.

```bash
find . > file_list
```

means “create a file called file_list containing the output”. If you use `>>` it means “append the output to the end of the file”.
HEAD, MORE, TAIL

The “head” command shows just the first lines of a file, or of the input received via a pipe.

More shows one page of its input at a time. Type “q” to quit.

Tail is like head, but shows the end of the file.
EXAMPLE

Ken often compiles programs this way:

```
g++ myprogram.cpp | & more
```

| & means “pipe output, including any error reports”. With |, error messages go to the console (not to the target of the pipe)

This lets him see any errors, but “pauses” after each full page.
You can also send the contents of a file into a program:

- `more < file_list`
  Shows the data in file_list one page at a time

- `find . > file_list &`
  Runs that same find command “in the background”

- `fg`
  Pulls it back into the “foreground” (and waits for it)
DO YOU REMEMBER THE TIMED WORD-COUNT RACE FROM LECTURE 1?

Bash has a built-in timing capability:

```
time command
```

But of course printing our sorted list of counts would be the main time spent. So I used

```
time command > /dev/null
```

This timed the command but “threw away” the actual output!
You can take it to the next level by creating a file with bash commands and then setting the execute permission bit for it.

Now if you “run” that file name, it runs the script of commands!

Bash supports variables, loops, conditional tests, simple math, string manipulations. You can even pipe program output into a bash variable. Very flexible and useful!
CMAKE SCRIPTS

Similar to bash scripts, but controlled by a “makefile” (and you have to actually run cmake as a command). Again, many fancy options

A basic makefile has the form

```
something:   files it depends on
            command(s) to rebuild it
```

Example

```
iconwriter:  iconwriter.cpp iconwriter.hpp
            g++ -O3 iconwriter.cpp -o iconwriter
```

Cmake will rebuild “iconwriter” if the .cpp or .hpp file has changed
Our class is working with C++ on Linux, so we need to become familiar with Linux. Linux ≅ kernel + device drivers + daemons + standard programs like initd and bash.

Today we reviewed some Linux concepts and tools as seen by the bash user who might be creating a C++ application.

In future lectures we will see some of the Linux system calls, that a program (in any language) can use to talk directly to the kernel.