CS3210: Operating Systems

Lab 1 Tutorial

Lab session general structure

- Session A - overview presentation (30 min)
  - Concepts, tutorial, and demo
- Session B - group activity (30 min)
  - Each student will get his/her hands dirty on tutorials
  - We will provide a README and/or source code
  - Divide class in three groups
  - Note: README is only for practice
- Session C (20 min)
  - Q&A lab

Lab1 goals

Understanding the tools required for OS development

Part 1 - Git source control and its internals

Part 2 - QEMU and debugging with QEMU

Part 3 - Basics of boot process and JOS makefile

Source Control - Git Basics

Why version control?

- Basic functionality
  - Keep track of changes made to files
  - Merge the contributions of multiple developers
- Accountability
  - Who wrote the code?
  - Do we have the rights to it?
- Software branches
  - Different software versions, ensure bug fixes shared
- Record keeping
  - Commit logs may tie to issue tracking system or be used to enforce guidelines

Why version control?

Setting up Git

In this class we will use Git heavily for development and for general assignment submission.

Let's walk through basic commands and then the internals. First, prevent your commits from having terrible `<user@hostname>` annotations.

Update your global config, one time only. Applies to all repos.

```
$ git config --global user.name "George P. Burdell"
$ git config --global user.email "burdell@gatech.edu"
```

Or, later go edit ~/.gitconfig manually:

```
$ cat ~/.gitconfig
[user]
  name=George P. Burdell
  email=burdell@gatech.edu
```

Why version control?

Setting up Git

Generating SSH keypair

Skip if you've already done this...

Yes, https works but this is much better.

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/vagrant/.ssh/id_
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/vagrant/.ssh/ic
Your public key has been saved in /home/vagrant/.ssh/id_rsa
```

Generally speaking, good to set a password.
You can always use `ssh-agent` to cache later.
Now...

```
$ cat ~/.ssh/id_rsa.pub

<copy contents to clipboard>
```

Getting started

Your lab repo:

```
git@github.gatech.edu:cs3210-spring2017/cs3210-lab-<username>.git
```

This is your own repo. It is readable and writeable only by you, the TAs, and the instructors. Let's clone it.

```
$ mkdir ~/cs3210
$ cd ~/cs3210
$ git clone git@github.gatech.edu:cs3210-spring2017/cs3210-lab-<user>.git lab
$ Cloning into lab...
$ cd lab
```

Note: If you are using Vagrant, it is convenient to clone into the folder containing your Vagrantfile. This folder is auto mounted by vagrant in the guest OS. Then you can use handy tools (gitk, git-gui) on the host OS for doing your commits.

Committing and pushing your changes

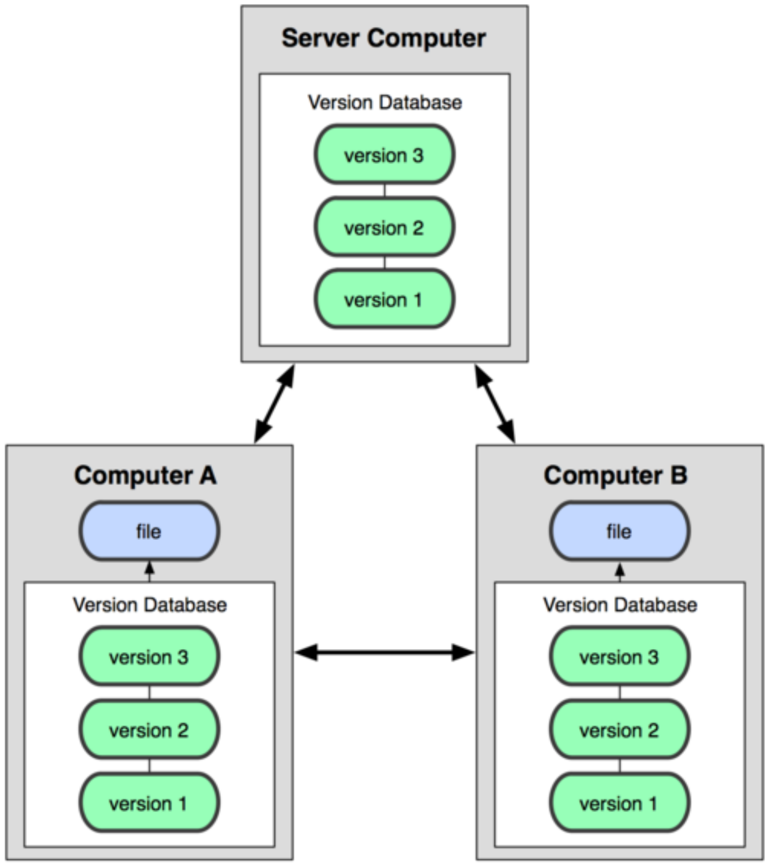
Bringing code under control, committing, pushing

```
$ git add code1.c
$ git commit -m "my solution for lab1 exercise 9"
$ git push origin master */ push your code to github.gatech */
```

Turning assignment in...

```
$ git tag "lab1_v1"
$ git push origin lab1_v1
```

Git - Distributed Version Control

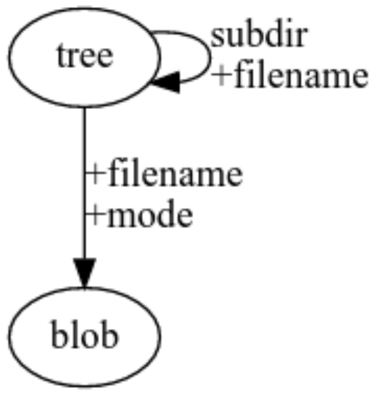


Git Internals - blob

- Git is a DAG (directed acyclic graph) of different type of objects
- Objects are stored compressed and identified by an SHA1
- Blob: simplest object, just a bunch of bytes, often a file

Git Internals - trees and blobs

- Directories are represented by a tree object
- They point to blob objects or subtrees



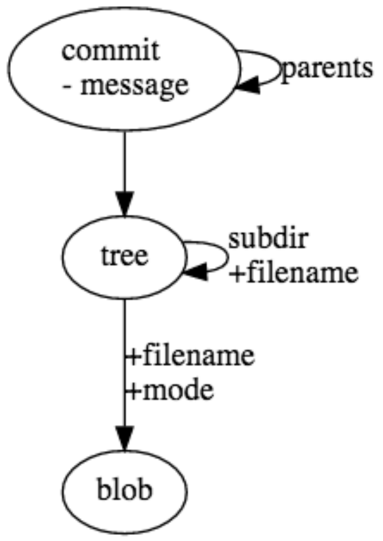
Git Internals - blobs and trees

```
$ find .git/objects type f
.git/objects/02/b365d4af3f74b0b1f18c41507c82b3ee571
.git/objects/37/ce98f6635fa1192d843bcaa4622537b2eb87  Tree
.git/objects/f0/5245cba7f23f998a5e372812d1a390375314c
```

```
$ git catfile p 37ce98f6635fa1192d85243bcaa4622537b2eb87
100644 blob 5fe92a0481023dfa3d2e64a0556dda3bbb852e5d     init.scm
100644 blob 20fa5e19fcb963f8a4ff249a815413153fb6b4e3    opdefines.h
10644 blob 69c742cc2544e336230d637b8115d69f0c050720   scheme.h
100644 blob badef17026a45893a7b3174db325e868c3a688b7     scheme.c
```

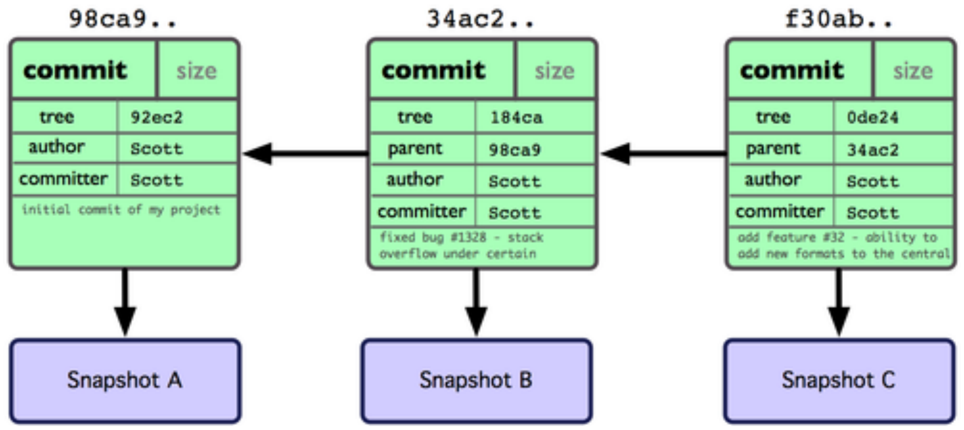
Git Internals - commit

- Commit refers to a tree that represents the state of the files at the time of the commit



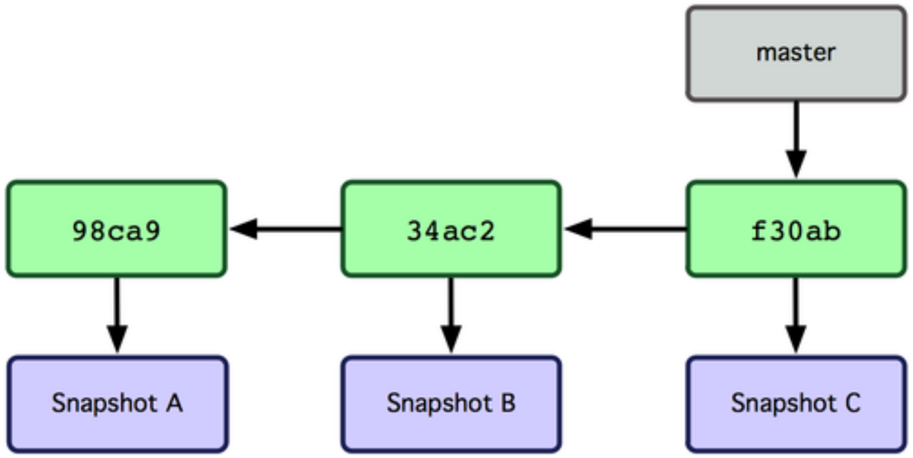
Git Internals - commit

It also refers to 0..n other commits that are its parents



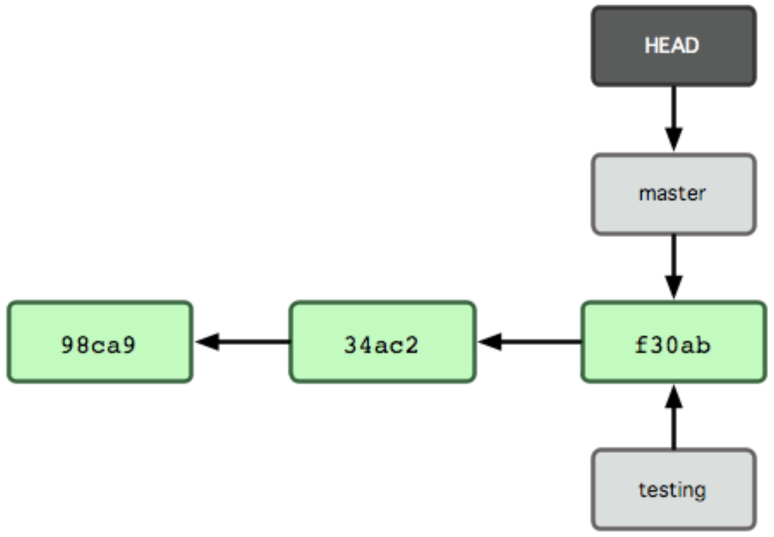
Git Internals - commit

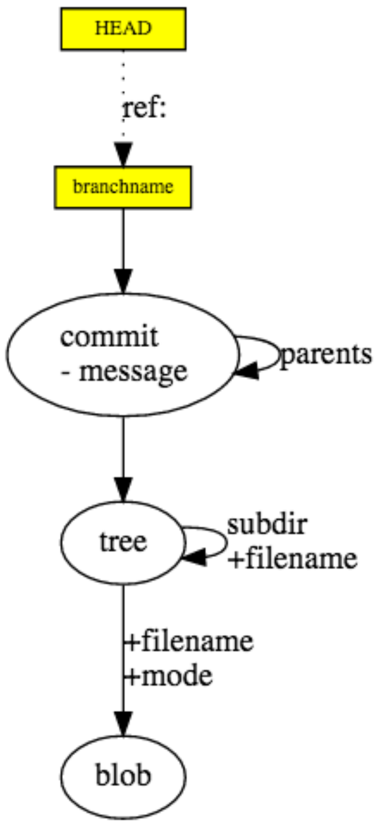
A branch is a pointer to a commit



Git Internals - commit

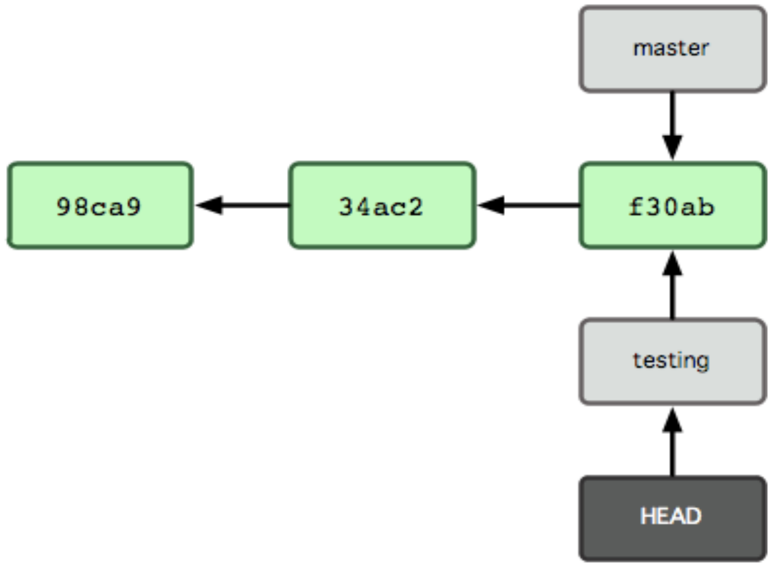
The files in the working directory reflect HEAD





Git internals - creating branch

```
$ git checkout -b testing
```



X86 Assembly

Why x86 assembly?

- All labs require understanding of assembly instructions
- We need to understand what instructions are executed during the boot process, which all written in assembly

The book "PC Assembly Language" is an excellent resource to understand the basics.

See: https://tc.gtisc.gatech.edu/cs3210/2017/spring/refs.html

We will not be covering it today in the class

QEMU emulator

PC Emulator

- Debugging and modifying real PC boot is hard
- So, we use a program that faithfully emulates a PC
- We can track, debug when our kernel boots
- So what does the emulator PC require?
    - A working OS!
    - Let's discuss the internals

What is QEMU?

Modes:

- System-mode emulation- emulation of a full system
    - User-mode emulation- launch processes compiled for another CPU(same OS)
    - Ex. execute arm/linux program on x86/linux
- Popular uses:
    - For cross-compilation development environments
    - virtualization, device emulation, for kvm
    - Android Emulator(part of SDK)

What is QEMU?

- QEMU is a user-level processor emulator
- Simulation vs. Emulation
    - Simulation - for analysis and study
    - Emulation - for usage as substitute

Building CS3210 kernel for emulator

```
$ cd lab
$ make
```

Successful build generates our CS3210 kernel:

`check kern/kernel.img`

Next we install our PC emulator - QEMU:

```
$ sudo apt-get install qemu
```

When done, we can boot our PC:

```
$ make qemu
```

Starting QEMU

```
$ make qemu-gdb
```

You will see the following printed on the screen

```
$ qemu-system-i386 -drive file=
obj/kern/kernel.img, index=0,media=disk,format=raw
-serial mon:stdio -gdb tcp::26001 D qemu.log
```

We will next discuss

- Boot procedure
- Using QEMU with gdb to understand boot procedure

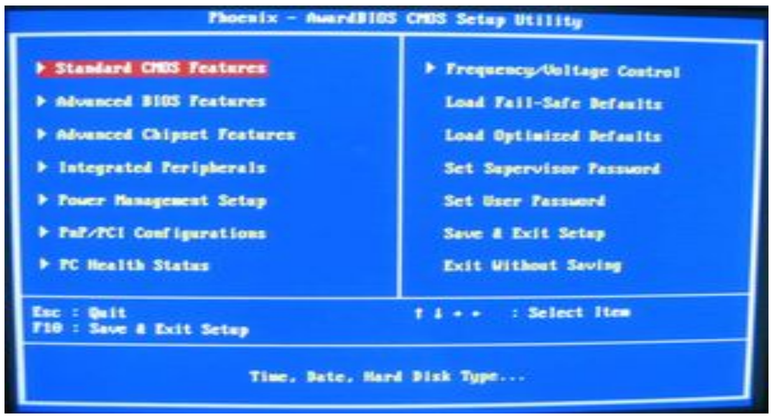
How does computer startup?

- Booting is a bootstrapping process that starts operating systems when the user turns on a computer system
- A boot sequence is the set of operations the performs when it is switched on that load an operating system

Understanding OS booting

What is BIOS

- BIOS refers to the software code run by a computer when first powered on
- The primary function of BIOS is code program embedded on a chip that recognizes and controls various devices that make up the computer



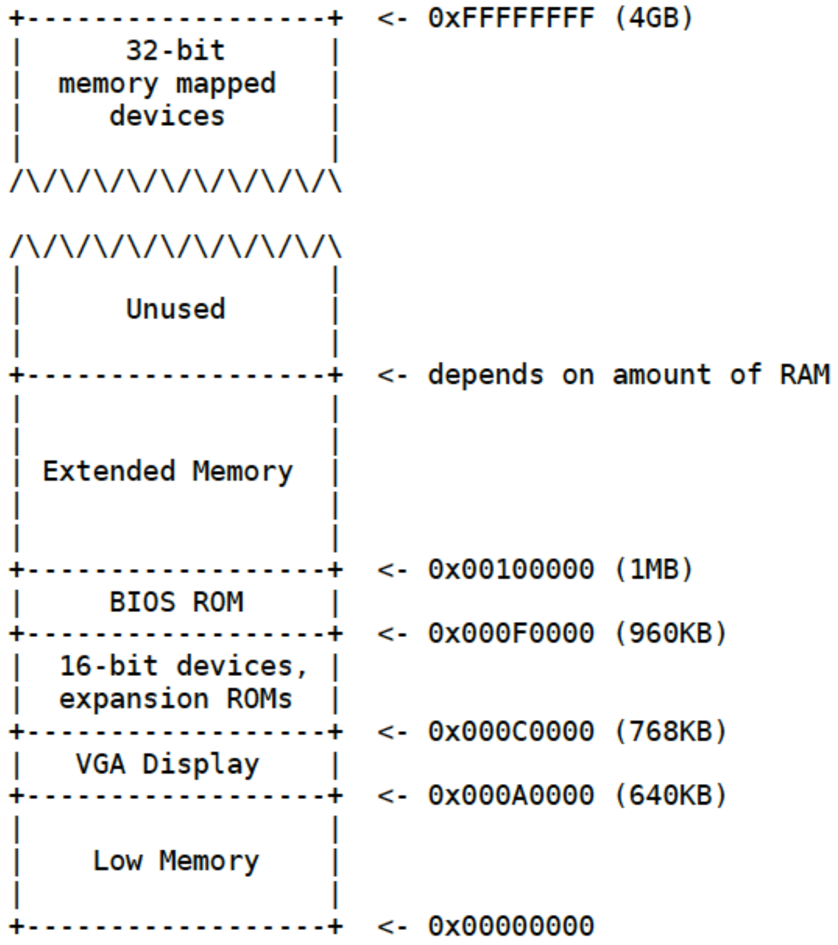
```
Phoenix - AwardBIOS CMOS Setup Utility

▶ Standard CMOS Features          ▶ Frequency/Voltage Control
▶ Advanced BIOS Features            Load Fail-Safe Defaults
▶ Advanced Chipset Features         Load Optimized Defaults
▶ Integrated Peripherals            Set Supervisor Password
▶ Power Management Setup            Set User Password
▶ PnP/PCI Configurations            Save & Exit Setup
▶ PC Health Status                  Exit Without Saving

Esc : Quit                        ↑ ↓ → ←   : Select Item
F10 : Save & Exit Setup

              Time, Date, Hard Disk Type...
```

Graphic Initial PC address is 0xffff0



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

Booting sequence - high-level steps first

1. Turning on the computer
2. CPU jumps to address of BIOS (0xFFFF0)
3. BIOS runs POST (Power-On Self Test)
4. Finds a bootable device
5. Loads and executes boot sector form MBR
6. Loads OS

Boot sector

- OS is booted from a hard disk, where the master boot Record (MBR) contains the primary boot loader
- The MBR is a 512byte sector, located in the first sector on the disk (sector 1 of cylinder 0, head 0)
- After the MBR is loaded into RAM, the BIOS yields control to it

Boot loader

- Boot loader is the code ultimately responsible for loading your kernel
- In JOS, you can find the boot-loader implementation in boot/main.c The boot loader does two important steps:

1. Switches processor from real mode to 32-bit (Why?)
2. Reads the kernel from the hard disk

How can we debug PC booting?

- GDB is the GNU program debugger
- GDB provides some helpful functionality
  - Allows you to stop your program at any given point.
  - You can examine the program state when stopped.
  - Change things in your program, so you can experiment with correcting the effects of a bug.
- So, let's see a demo for debugging our PC emulator

JOS: Boot loader (main.c and boot.S)

- Both boot.S and main.c correspond as JOS's boot loader
- It should be stored in the first sector of the disk
- The 2nd sector onward holds the kernel image
- In JOS source, bootmain() function is where it all starts
- Function readsect() reads the first sector (boot loader)

Cscope Walking through the source kernel

- Cscope can be a particularly useful tool if you need to wade into a large code base
- Fast, targeted searches rather than randomly grepping through the source files by hand

To recursively parse a directory, use

```
$ cscope -R -p X
```

X represents the number of levels of subdirectories

Cscope Walking through the source kernel

Commonly used Cscope options:

- Find this C symbol: (functions or symbols to be searched)
- Find this global definition: (function definition)
- Find functions called by this function: (callee's of a func)
- Find functions calling this function: (caller's of a func)
- Find this egrep pattern: (search by grepping)
- Find this file: (locate a file)

Getting hands dirty

```
$ git clone git@github.gatech.edu:cs3210-spring2017/cs3210-pub
$ cd cs3210pub/tut/tut1
$ less README
```

Looks like...

```
# Lab 1 - Tools

## Part 1. Getting hands dirty with git

In this part you will try the basics required for updating,
committing, and submitting your code via git. For more
details about git at http://gitimmersion.com/index.html

1. Install git

    $ sudo apt-get install git  (in ubuntu)

2. Set your user name and email

    $ git config --global user.name "Your Name"
    $ git config --global user.email "your_email@whatever.com"
```