Lab1 tutorial – CS 3210

https://tc.gtisc.gatech.edu/cs3210/2016/lab/lab1.html

Lab session general structure

Session A - overview presentation (30 min)

- About concept, 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

Setting up Git

In this class, we will use Git

Lets walkthrough basic commands and then the internals

Update your config, one time only

```
$ git config --global  user.name “john.lastname”
$ git config --global user.email john@gatech.edu
```

Getting started

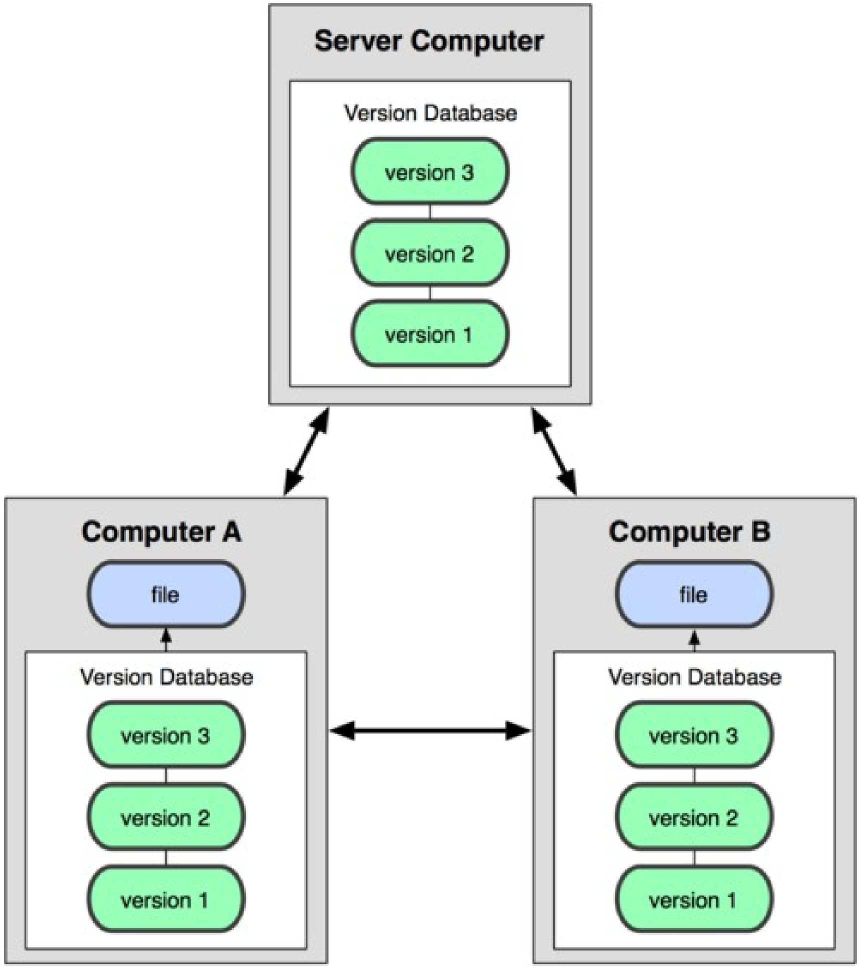
Course git repo: git://tc.gtisc.gatech.edu/cs3210-lab

```
$ mkdir ~/cs3210
$ cd ~/cs3210
$ git clone git://tc.gtisc.gatech.edu/cs3210-lab lab
$ Cloning into lab...
$ cd lab
```

Committing your changes

```
$ git add code1.c  /*  Tells git to track a file */
$ git commit -am 'my solution for lab1 exercise 9'

$ make handin LAB=1
```

Git - distributed version control



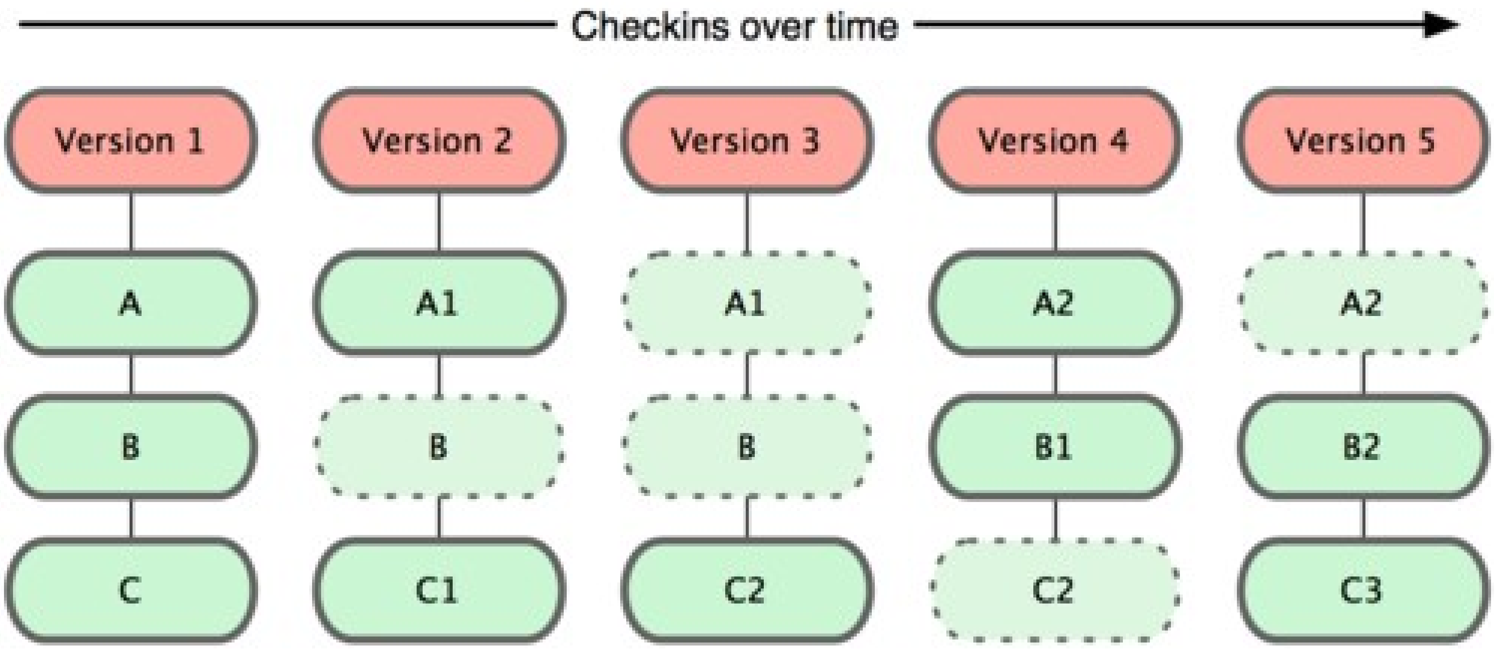
Git internals - blob

Git is a DAG (directed acyclic graph) of different type of objects

Objects are stored compressed and identified by an SHA-1

Blob: simplest object, just a bunch of bytes, often a file

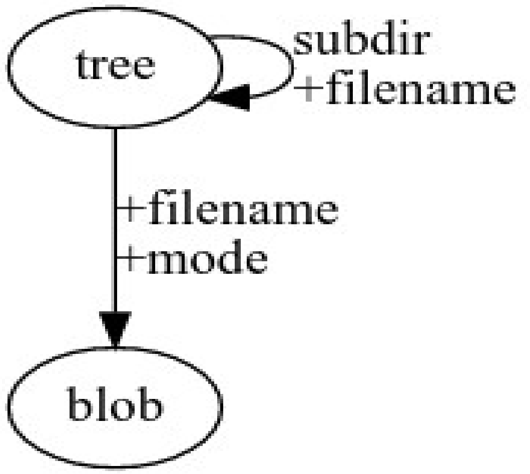
Git - distributed snapshots



Git internals - trees and blobs

Directories are represented by a tree object

They point to blob objects or subtrees



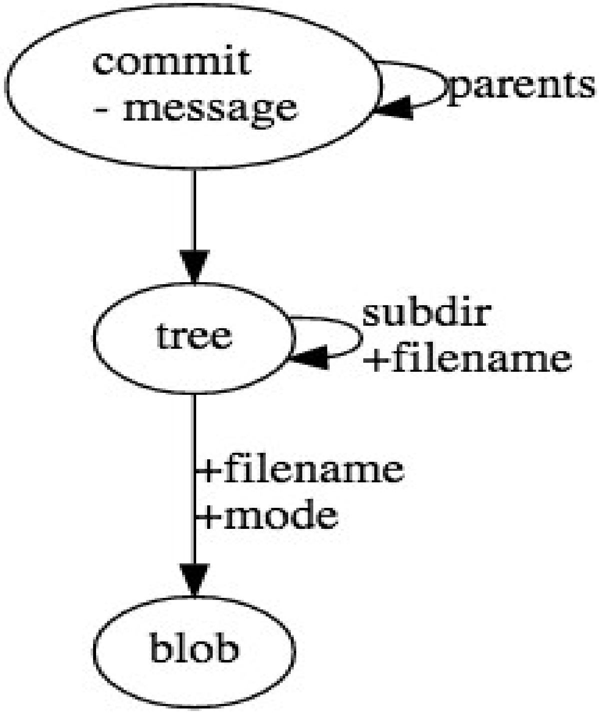
Git internals – blobs and trees

```
$ find .git/objects -type f

.git/objects/02/b365d4af3f74b0b1f18c41507c82b3ee571
.git/objects/37/ce98f6635fa1192d843bcaa4622537b2eb87 - Tree
.git/objects/f0/5245cba7f23f998a5e372812d1a390375314c
```

```
$ git cat-file -p 37ce98f6635fa1192d85243bcaa4622537b2eb87

100644 blob 5fe92a0481023dfa3d2e64a0556dda3bbb852e5d    init.scm
100644 blob 20fa5e19fcb963f8a4ff249a815413153fb6b4e3    opdefines.h
10644 blob 69c742cc2544e336230d637b8115d69f0c050720    scheme.h
100644 blob badef17026a45893a7b3174db325e868c3a688b7    scheme.c
```

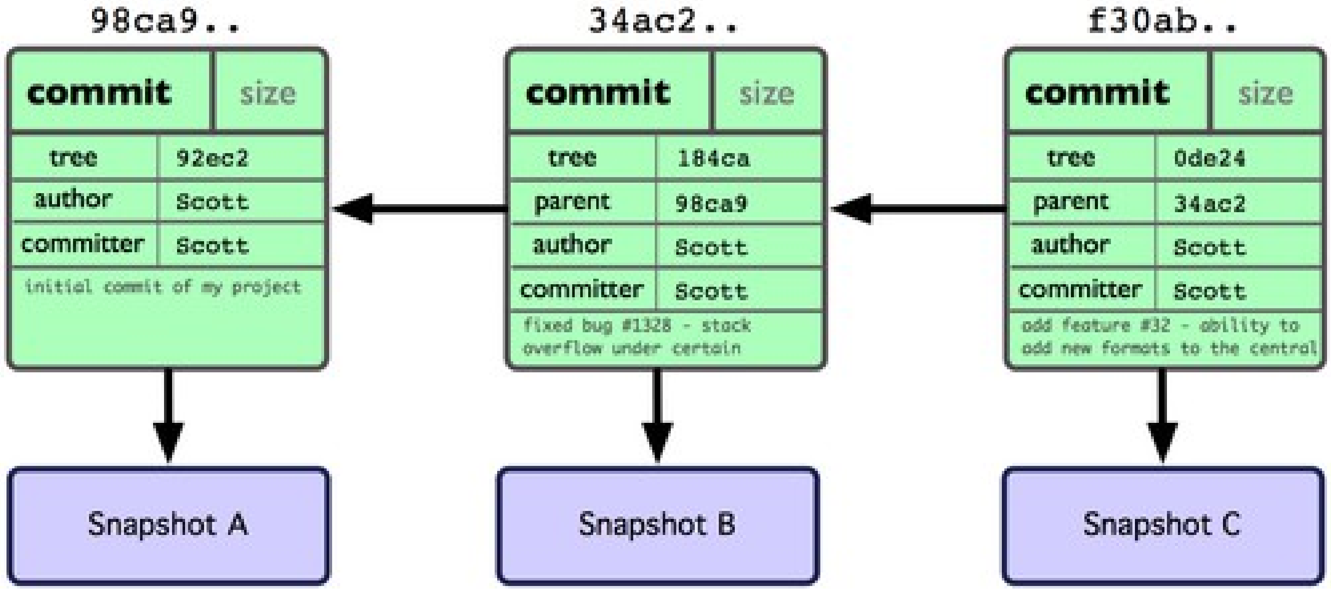
Git internals - commit

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



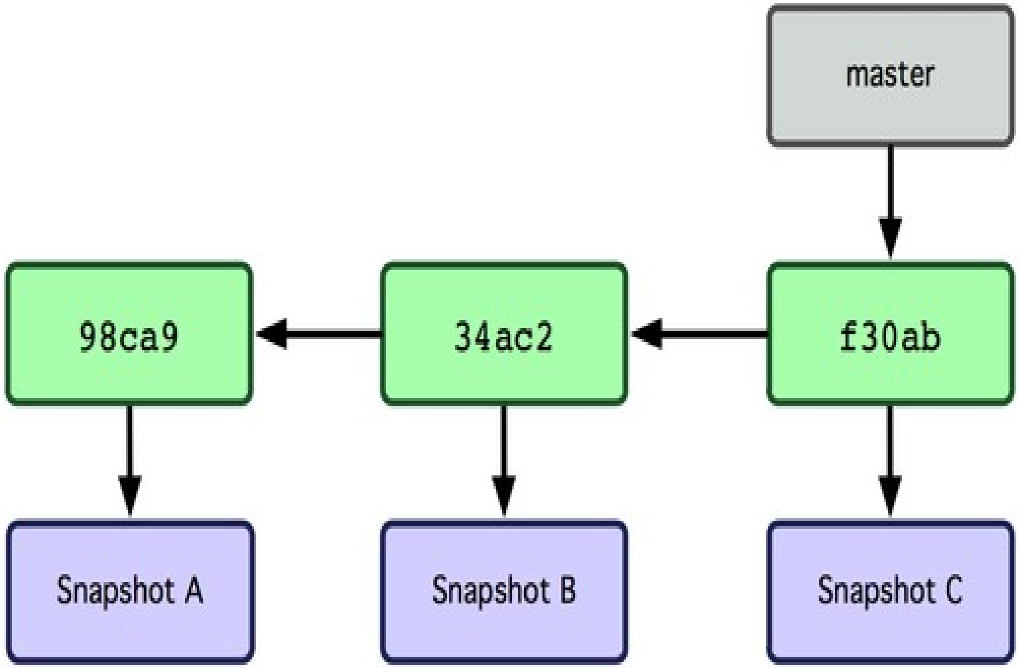
Git internals - commit

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



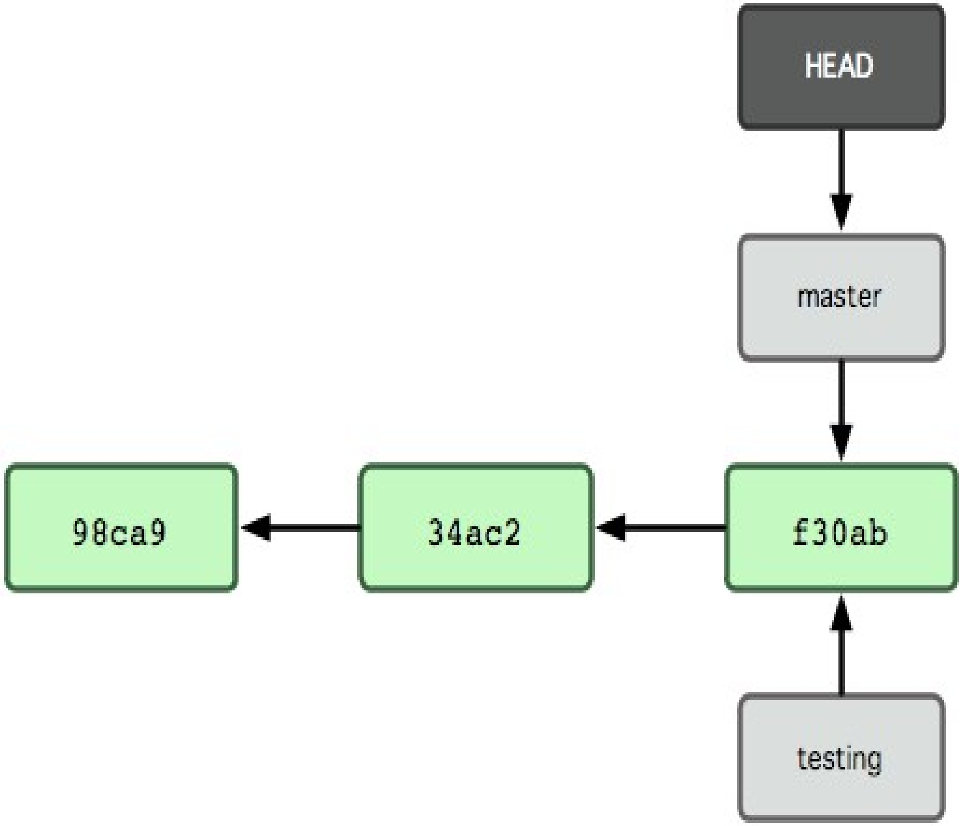
Git internals - commit

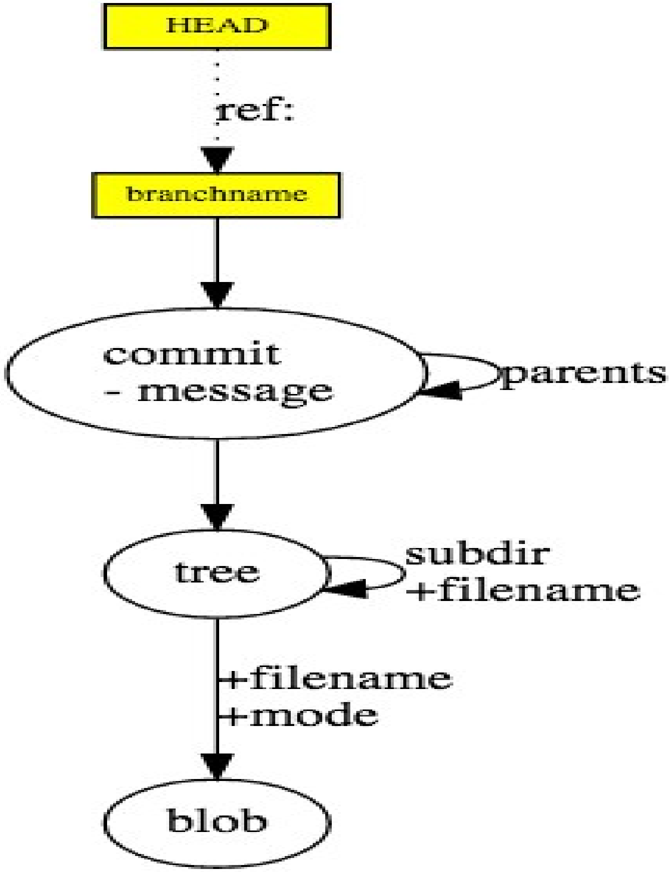
A branch is a pointer to a commit.



Git internals - commit

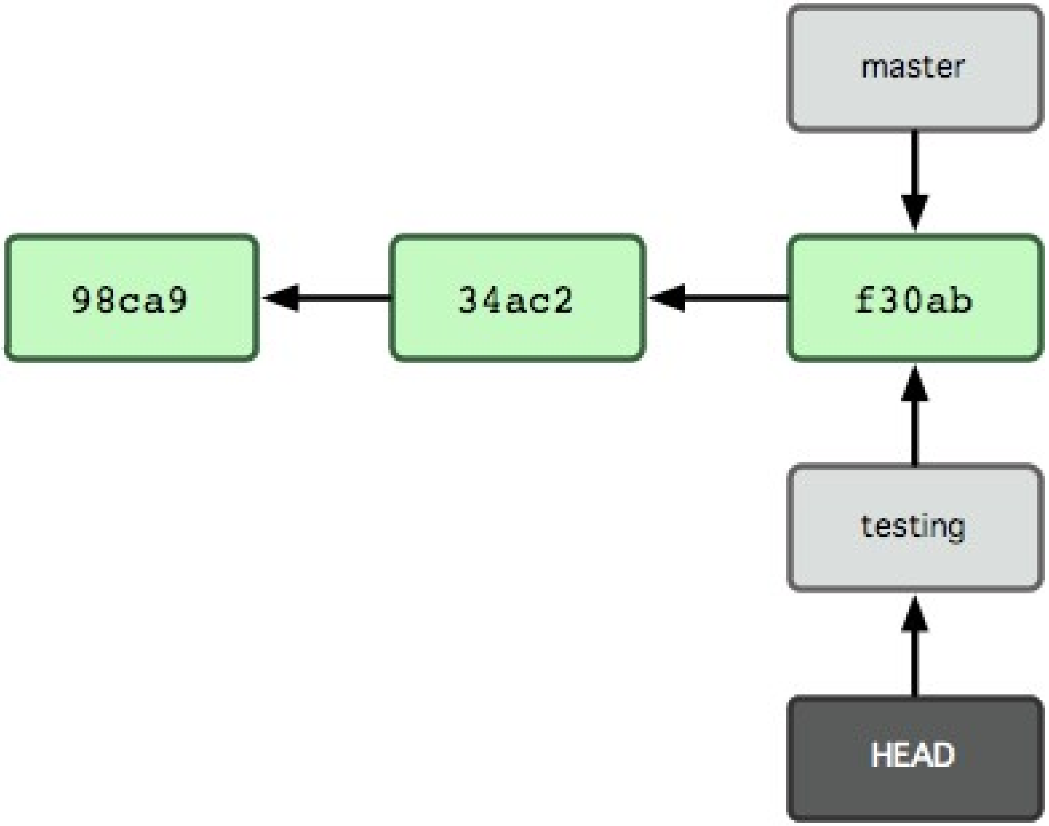
The files in the working directory reflect HEAD





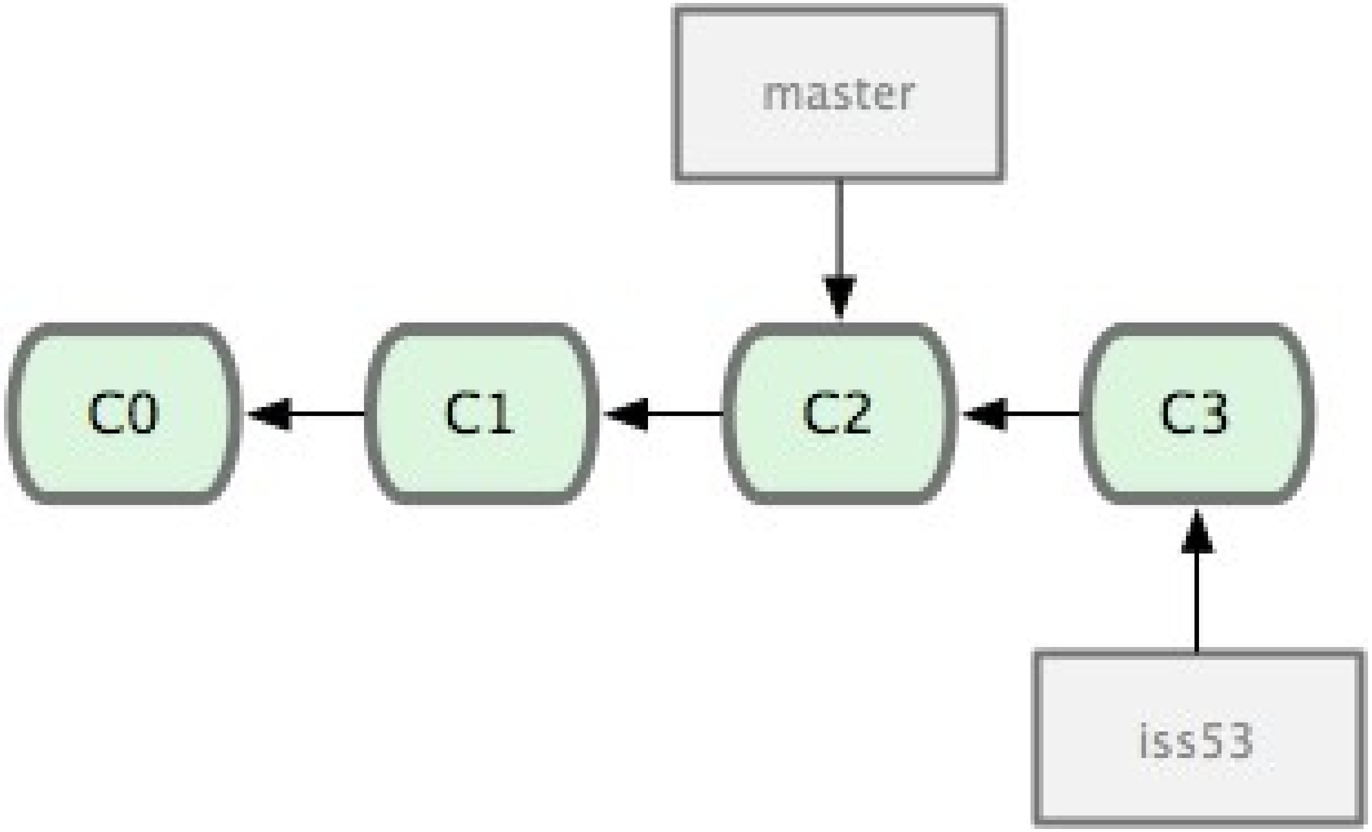
Git internals - creating branch

git checkout testing



Git internals - creating branch

git commit -m "commit is53"



X86 Assembly

Why x86 assembly?

All labs require understanding of assembly instructions

We need to understand what instructions are executed during the boot

The book “PC Assembly Language” is an excellent resource to understand the basics

https://tc.gtisc.gatech.edu/cs3210/2016/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:

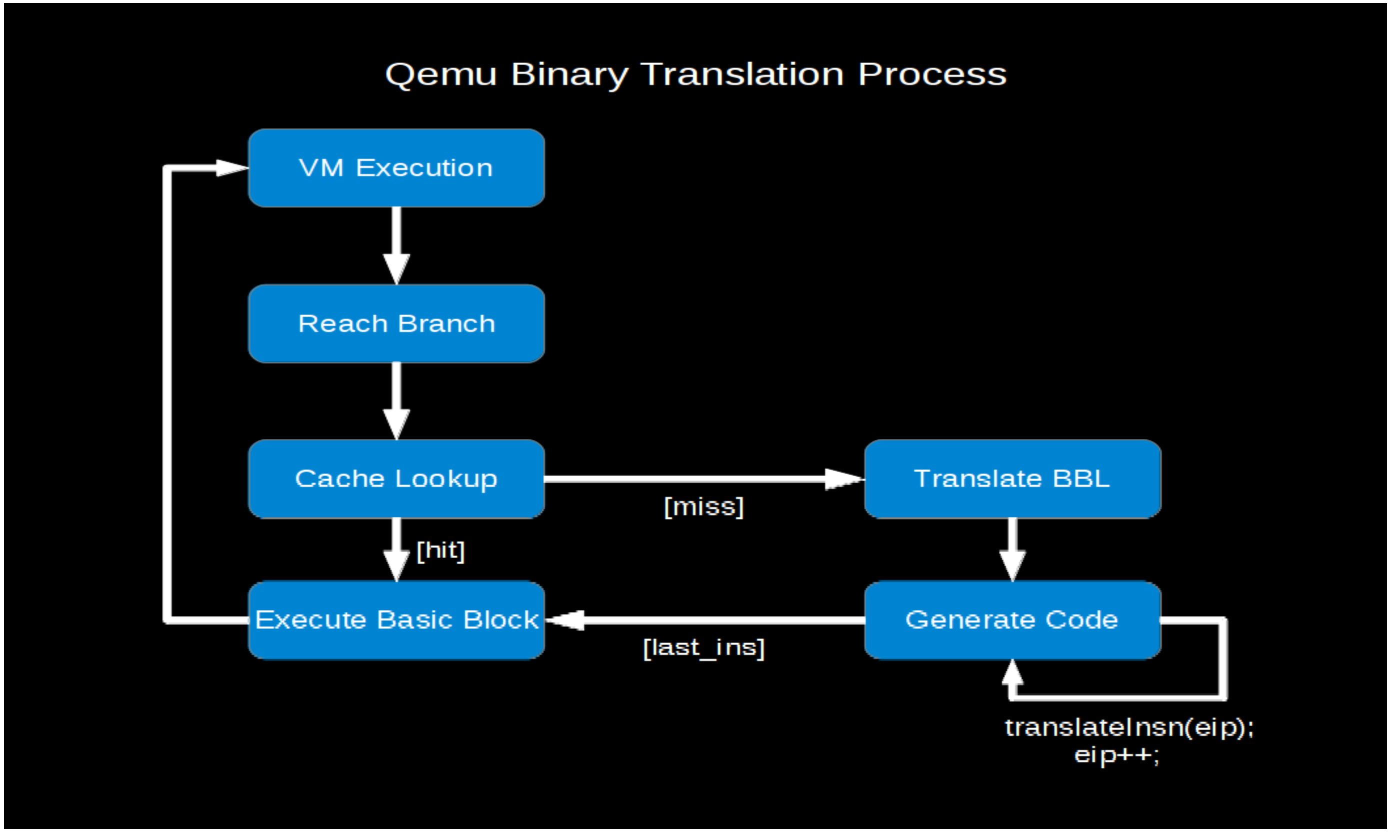
- System-mode emulation – emulation of a full system
- User-mode emulation – launch processes compiled for another CPU(same OS)
  - Ex. execute arm/linux program on x86/linux

Popular uses:

- For cross-compilation development environments
- virtualization, device emulation, for kvm
- Android Emulator(part of SDK)

Dynamic translation

Target CPU instruction → Host CPU instruction

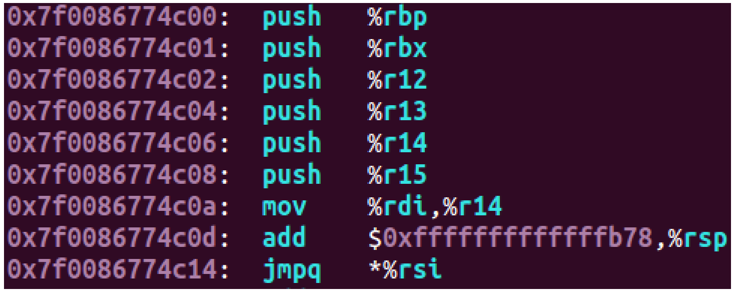


What is QEMU?

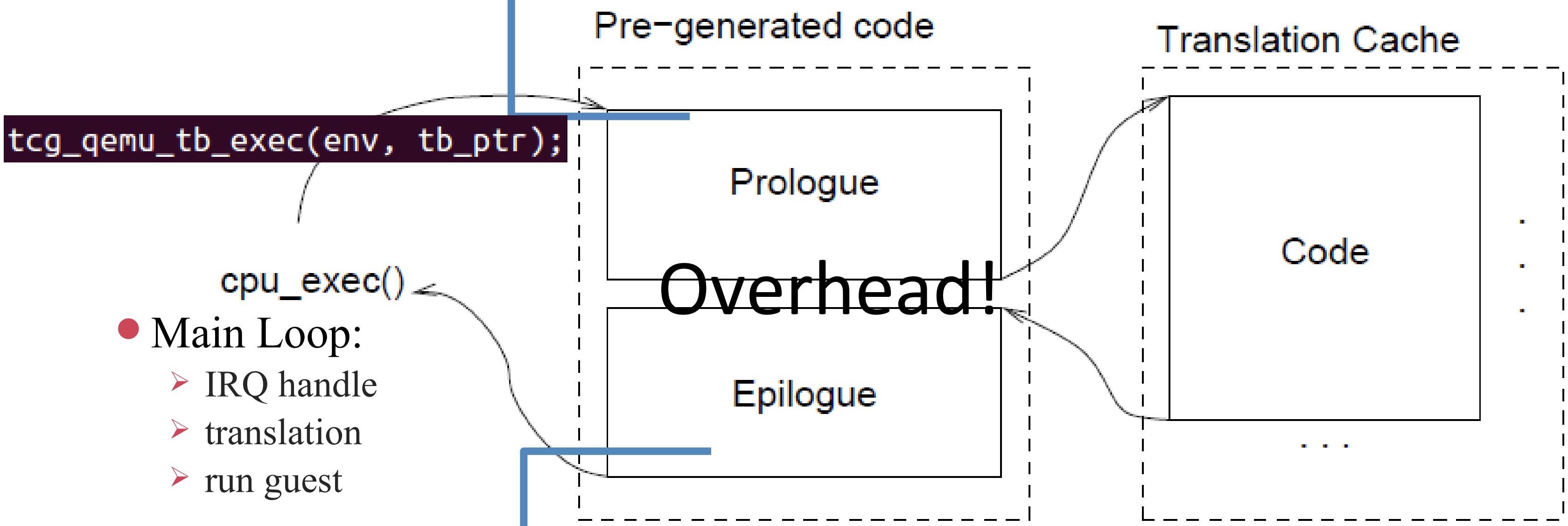
- QEMU is a user-level processor emulator
- Simulation vs. Emulation
  - Simulation – for analysis and study
  - Emulation – for usage as substitute

Translation and execution

initialize the process or and jump to the host code



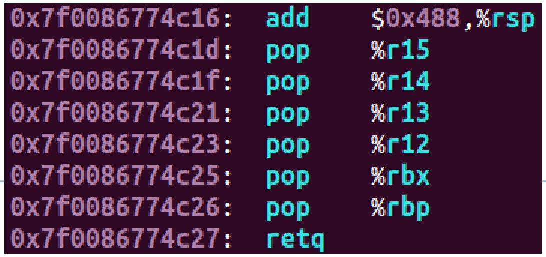
```
0x7f0086774c00:  push   %rbp
0x7f0086774c01:  push   %rbx
0x7f0086774c02:  push   %r12
0x7f0086774c04:  push   %r13
0x7f0086774c06:  push   %r14
0x7f0086774c08:  push   %r15
0x7f0086774c0a:  mov    %rdi,%r14
0x7f0086774c0d:  add    $0xfffffffffffffb78,%rsp
0x7f0086774c14:  jmpq   *%rsi
```



```
tcg_qemu_tb_exec(env, tb_ptr);
```

cpu_exec()

- Main Loop:
  - IRQ handle
  - translation
  - run guest



```
0x7f0086774c16:  add    $0x488,%rsp
0x7f0086774c1d:  pop    %r15
0x7f0086774c1f:  pop    %r14
0x7f0086774c21:  pop    %r13
0x7f0086774c23:  pop    %r12
0x7f0086774c25:  pop    %rbx
0x7f0086774c26:  pop    %rbp
0x7f0086774c27:  retq
```

restore normal state and return to the main loop

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

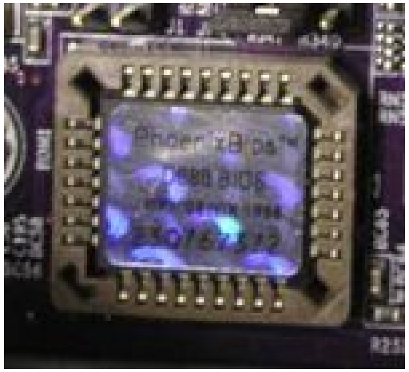
A boot sequence is the set of operations the performs when it is switched on that load an operating system

Understanding the OS booting

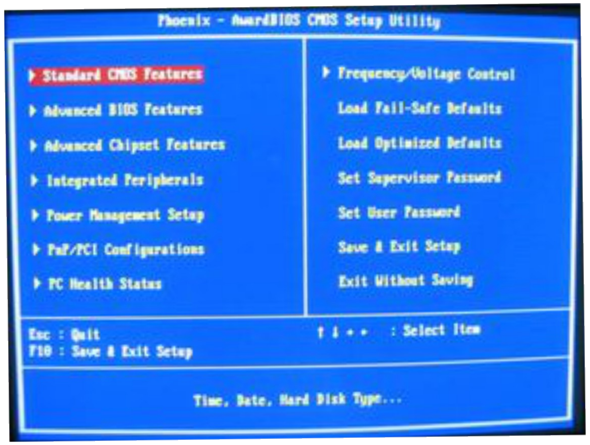
What is BIOS

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.



BIOS on board



BIOS on screen

PC physical address space and BIOS loading

```
+------------------+  <- 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
```

Our emulator PC is 32 bit.
64 bit – beyond 4GB address

- Initial PC address in our emulator **0xffff0**

Flash vs ROM differences?

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 512-byte sector, located in the f irst 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 a code 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

.

QEMU generic features?

Self-modifying code

Precise exception

> Process state corresponds to sequential execution when an interrupt occurs

FPU - software emulation of host FPU instructions

Dynamic translation to native code => speed

Boot sector

OS is booted from a hard disk, where the master boot Record (MBR) contains the primary boot loader

The MBR is a 512-byte sector, located in the f irst sector on the disk (sector 1 of cylinder 0, head 0)

After the MBR is loaded into RAM, the BIOS yields control to it

.

.

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://tc.gtisc.gatech.edu/cs3210-pub
```

```
cd cs3210-pub/tut/tut1
```

Open README file