git (fin.), Processes (start)

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW05 posted after class
  - Should be a pretty short assignment, just a “hands on” for the git stuff so that you aren’t as scared when you see it again :)
- Check-in posted after class or tomorrow
  - May be super short and just re-opens, since I am a little sick and pretty tired

Lecture Outline

- git
- processes

git reset

- **`git reset --hard <commit>`**
  - Resets the current branch to the specified commit
  - DISCARDS ANY CHANGES MADE SINCE THE SPECIFIED COMMIT
  - DANGEROUS! RARELY USE (if ever)

- **`git reset --soft <commit>`**
  - Resets the current branch to the specified commit
  - Changes made since specified commit will still be there, but “to be committed”

- Pretty useful command: **`git reset --soft HEAD~1`**
  - Undoes the most recent commit and sends changes back to “Staging”

git tags

- Allows you to mark a specific commit as important.

- git tag
  - Lists the current tags

- `git tag <tag-name> <commit-hash>`
  - Creates a tag with the specified name on the specified commit
  - If commit hash is left out, tag will be created on the same commit as HEAD

- `git push --tags`
  - To push tags to remote

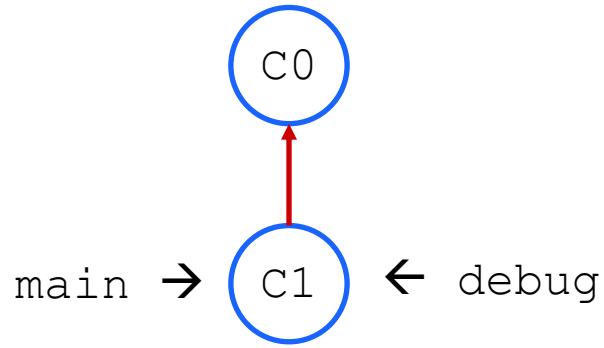
HW05 demo

- HW05 is a little different, so we will demo
  - Using the provided script
  - Tagging

- Should be posted later today, shouldn't take tooooo long we hope

git branch

- Our git tree doesn't always look like a line.
  we can create branches with diverging history.



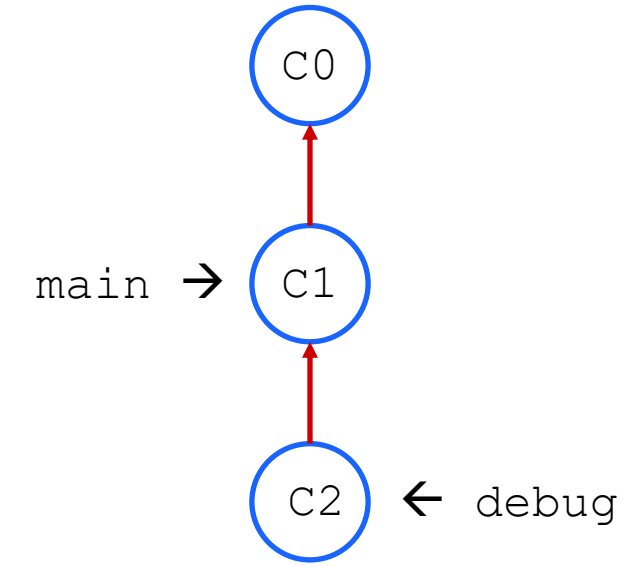
- can create branches with **`git branch <name>`**
- **`git branch --list`** to list branches
- **`git checkout <name>`** to switch head to the specified branch
  - **`git checkout -b <name>`** creates and checks out a branch with specified name

- Example:
  - **`git branch debug`**

git branch

- Our git tree doesn't always look like a line.
  we can create branches with diverging history.

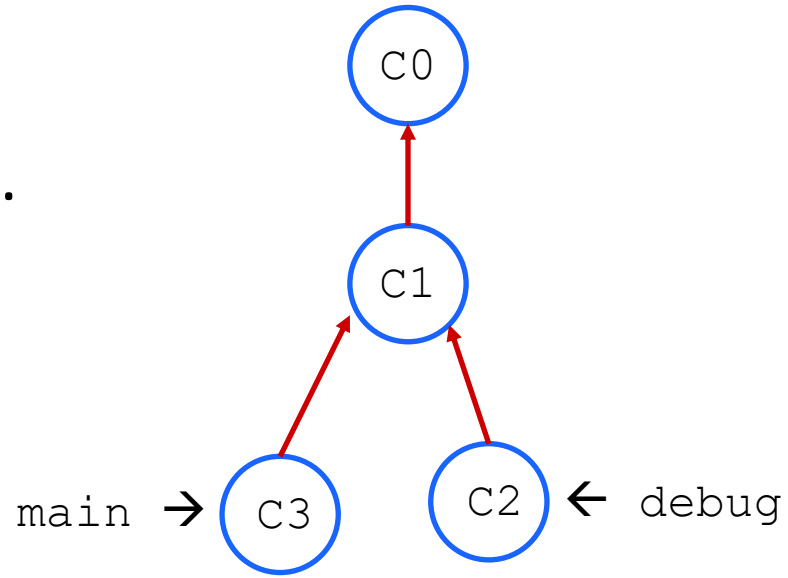
- Example:
  - **`git branch debug`**
  - **`git checkout debug`**
  - **`git commit -m "..."`**



git branch

- Our git tree doesn't always look like a line.
  we can create branches with diverging history.

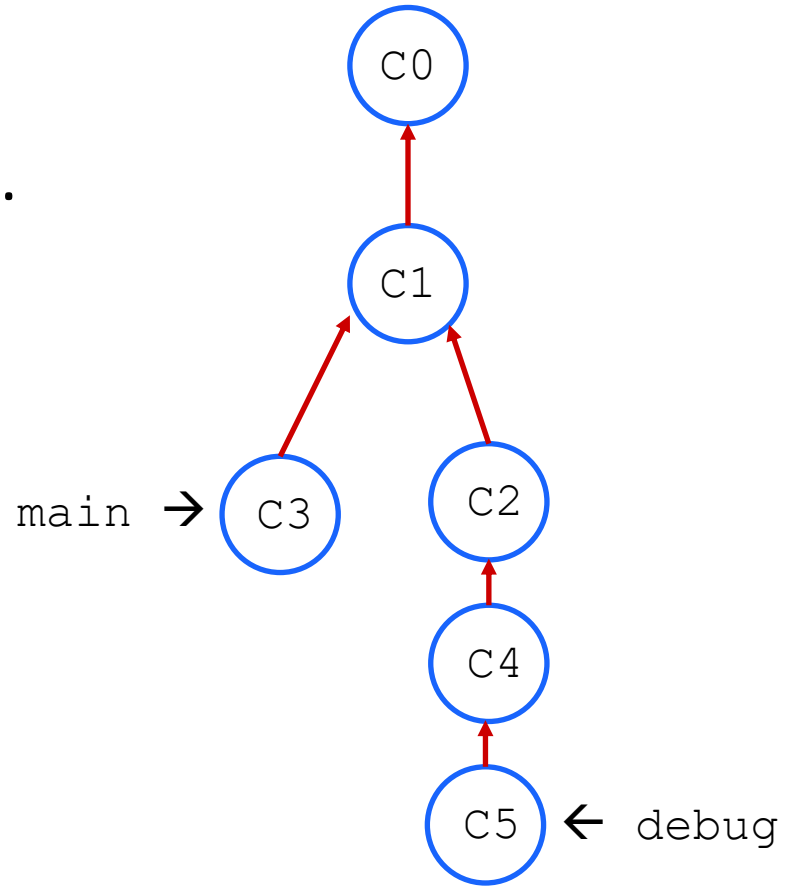
- Example:
  - **`git branch debug`**
  - **`git checkout debug`**
  - **`git commit -m "..."`**
  - **`git checkout main`**
  - **`git commit -m "..."`**



git branch

- Our git tree doesn't always look like a line.
  we can create branches with diverging history.

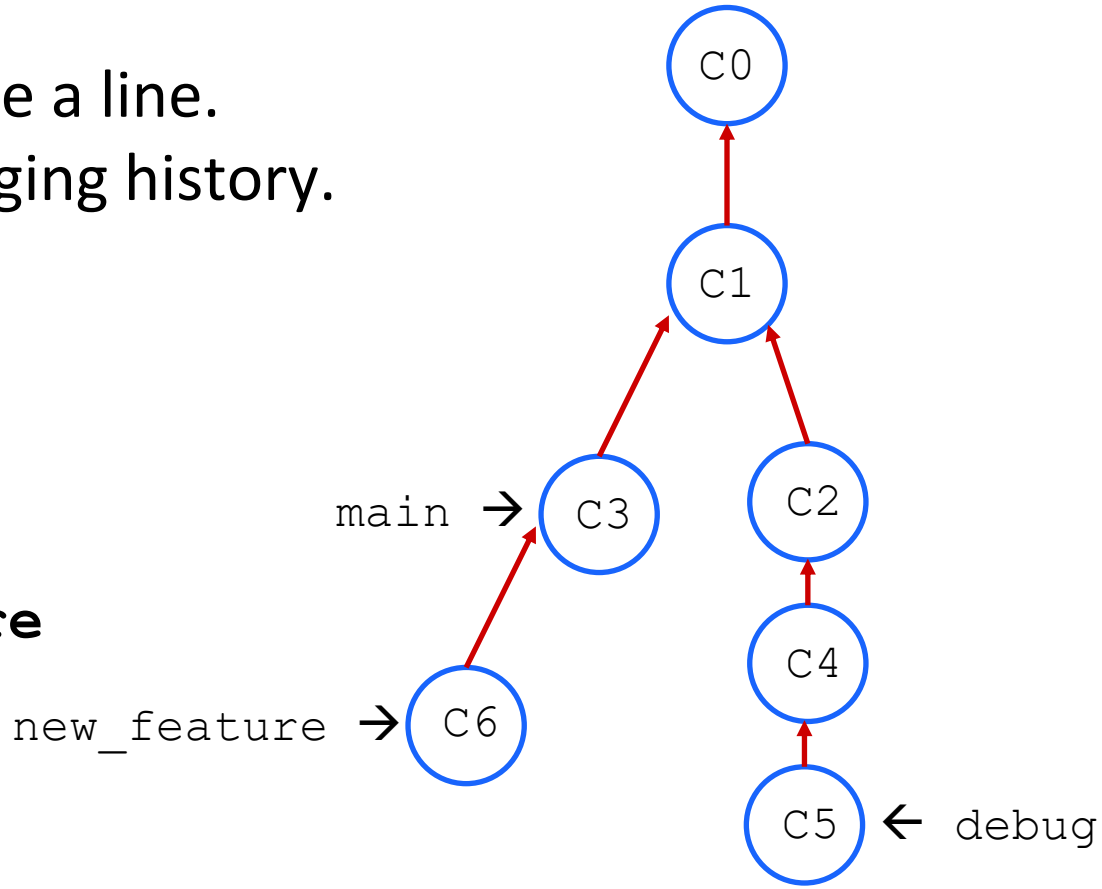
- Example:
  - **`git branch debug`**
  - **`git checkout debug`**
  - **`git commit -m "..."`**
  - **`git checkout main`**
  - **`git commit -m "..."`**
  - **`git checkout debug`**
  - **`git commit -m "..."`**
  - **`git commit -m "..."`**



git branch

- Our git tree doesn't always look like a line.
  we can create branches with diverging history.

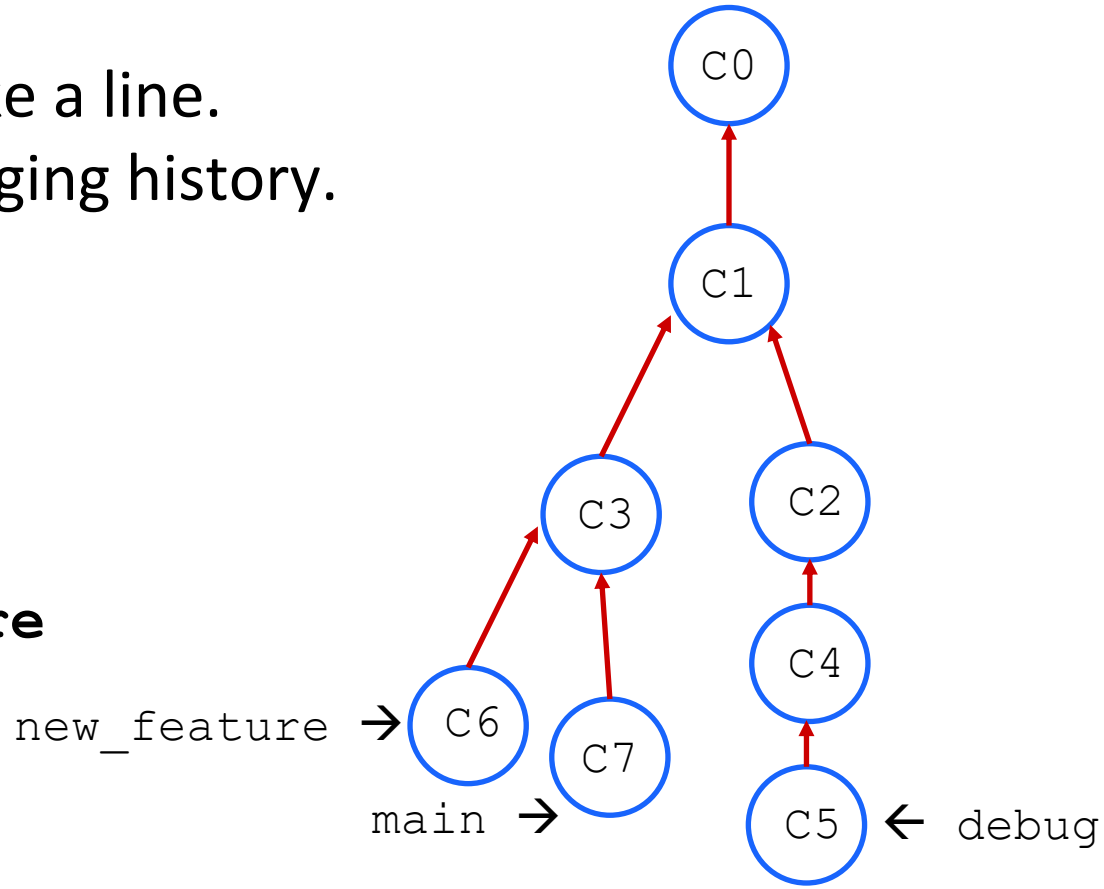
- Example:
  - …
  - **`git checkout main`**
  - **`git checkout -b new_feature`**
  - **`git commit -m "..."`**



git branch

- Our git tree doesn't always look like a line.
  we can create branches with diverging history.

- Example:
  - …
  - **`git checkout main`**
  - **`git checkout -b new_feature`**
  - **`git commit -m "..."`**
  - **`git checkout main`**
  - **`git commit -m "..."`**



git stash

- **`git stash`**: This will take your local changes, save them, while also letting your "reset" the current state of your repo to the previous commit

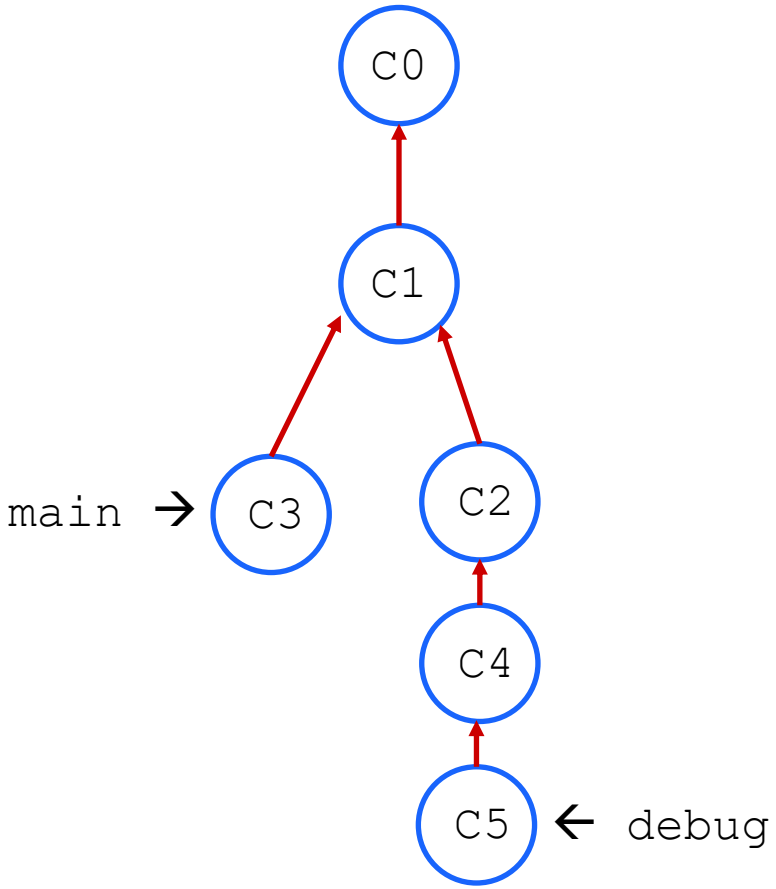
- Can inspect your changes with:
  - git stash list
  - git stash show

- To take your changes out of the stash, you use **`git stash pop`** (which also removes it from the stash). **`git stash apply`** doesn't remove from the stash.

- If pop/apply would conflict, you either resolve it manually or create a a new branch for your changes: **`git stash branch <branchname>`**

Merge branches

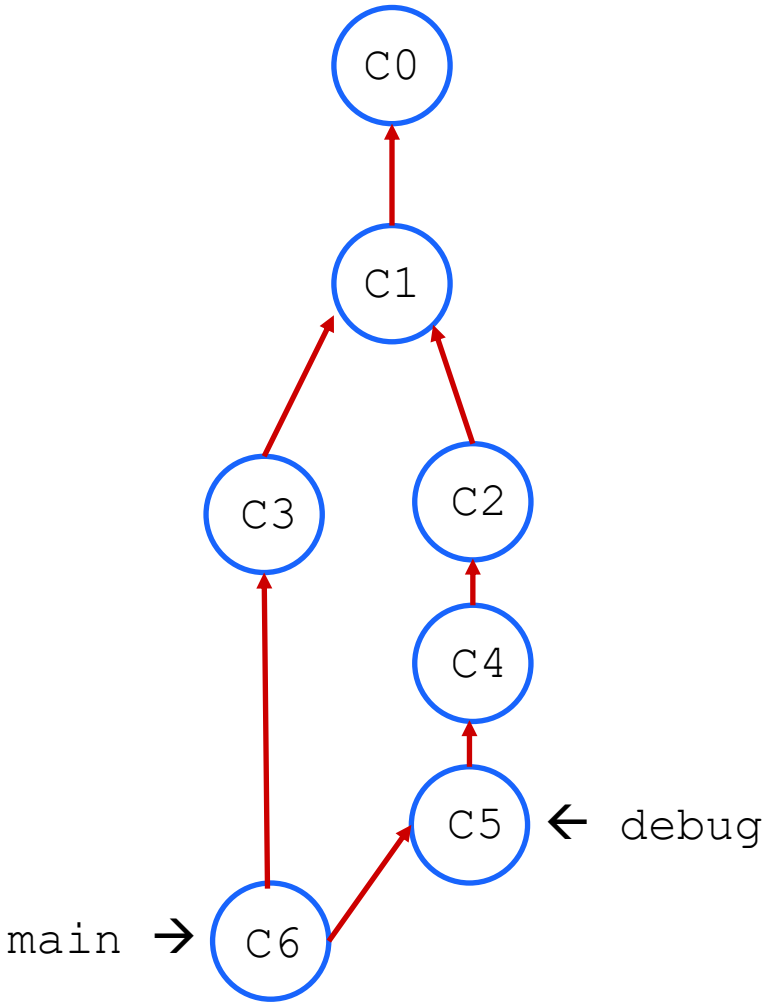
- We can merge branches if we want to include changes from one branch into another.
- Here we want to include debug into main.



Merge branches

- We can merge branches if we want to include changes from one branch into another.
- Here we want to include debug into main.

- Assume HEAD starts at main
  - **`git merge debug`**
    - May need to resolve conflicts here

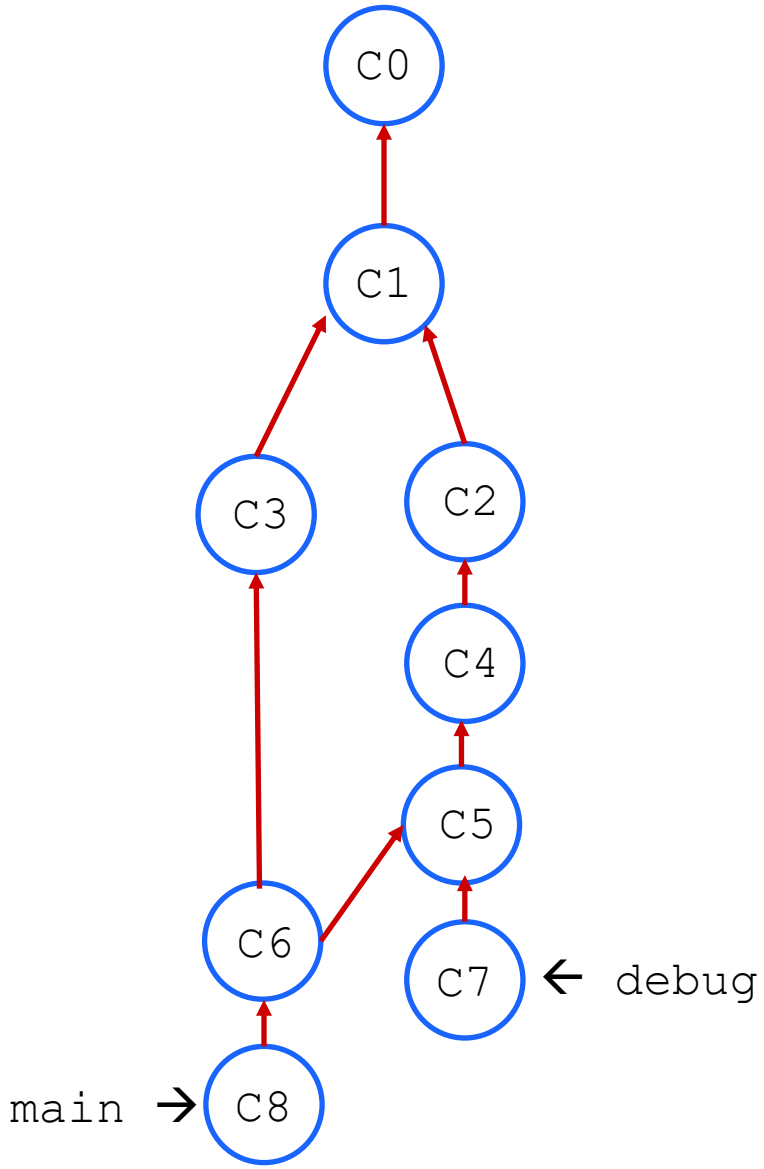


Merge branches

- We can merge branches if we want to include changes from one branch into another.
- Here we want to include debug into main.

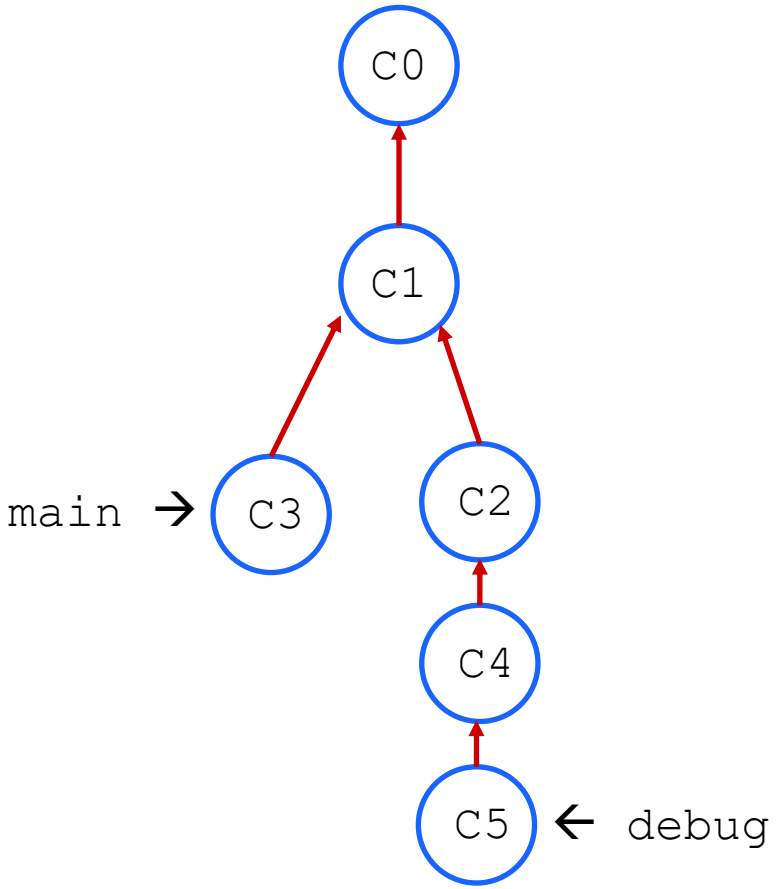
- Assume HEAD starts at main
  - **`git merge debug`**
    - May need to resolve conflicts here

- debug and main can keep growing and be merged again later



git rebase

- `git rebase` is an alternative to merge
  - Can be used to “merge” branches and also if there are issues like with merge conflicts from last lecture.

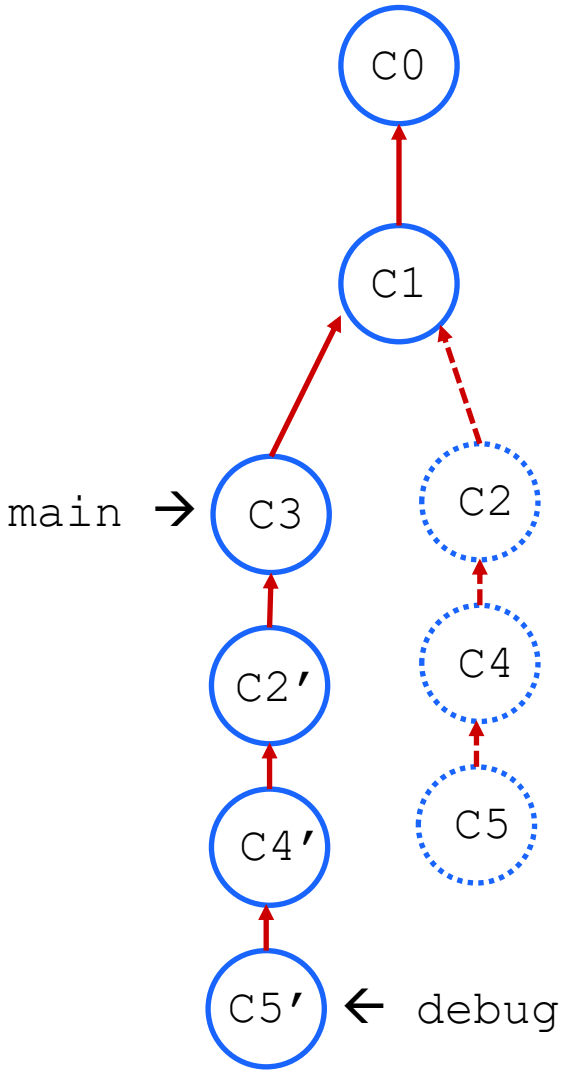


- Re-base -> “base your changes off of a new foundation.”

- If HEAD is on debug and we invoke
  - git rebase main

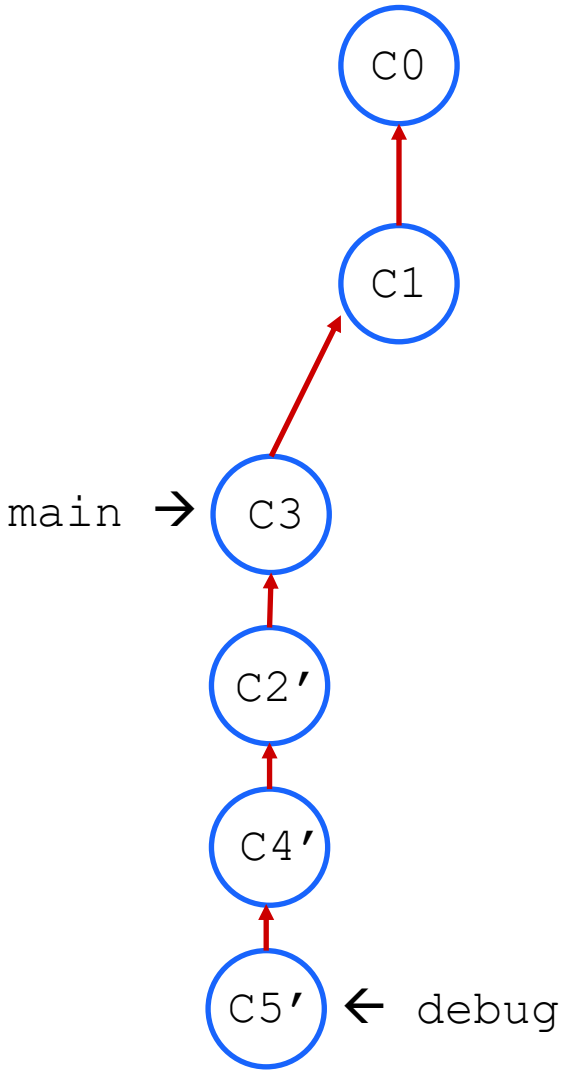
git rebase

- `git rebase` is an alternative to merge
  - Can be used to “merge” branches and also if there are issues like with merge conflicts from last lecture.
- Re-base -> “base your changes off of a new foundation.”
- If HEAD is on debug and we invoke
  - `git rebase main`



git rebase

- `git rebase` is an alternative to merge
  - Can be used to “merge” branches and also if there are issues like with merge conflicts from last lecture.
- Re-base -> “base your changes off of a new foundation.”
- If HEAD is on debug and we invoke
  - `git rebase main`



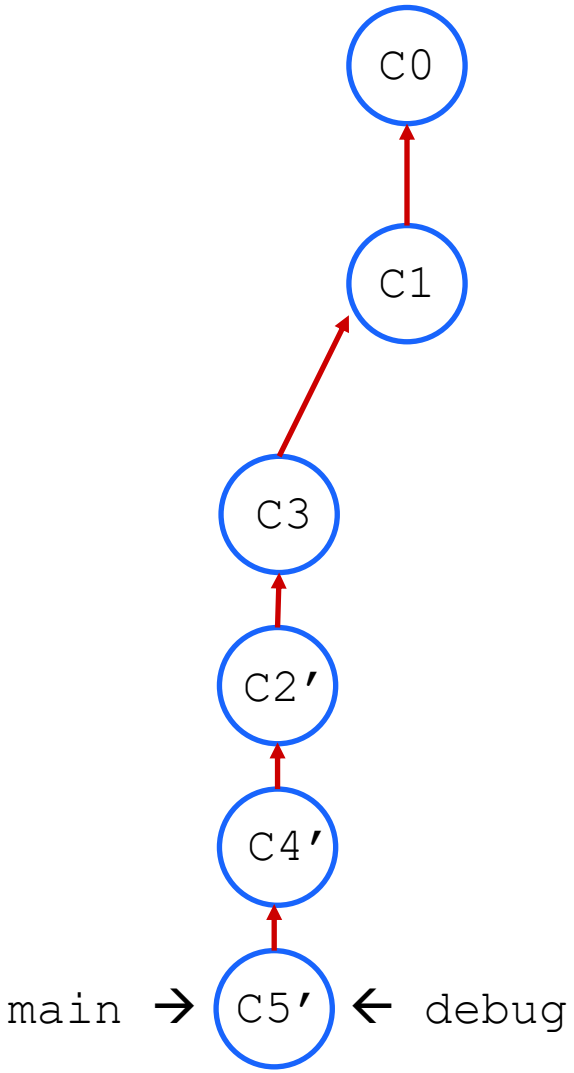
git rebase

- `git rebase` is an alternative to merge
  - Can be used to "merge" branches and also if there are issues like with merge conflicts from last lecture.

- Re-base -> "base your changes off of a new foundation."

Fast forward: only thing to do was move the reference forward

- We can "Fast forward" main to the same commit as debug:
  - `git checkout main`
  - `git merge debug`



Rebase vs merge

- When to use each?

- Merge:
  - Incorporate changes from one branch onto another

- Rebase
  - Change the starting point of this branch

Pull Request

- A feature of GitHub to propose merging* a set of commits from one branch onto another.

- Often shortened to “PR”
  - submit a PR → submit a pull request

- Gitlab has the same feature* called “merge request”
  - I will call it merge request sometimes because that is what I learned.
  - It also makes more sense as a name imo cause it almost always does a merge.

Pull Request (DEMO)

- Go to github repository, there should be a tab called “pull requests” you can use near the top
  - You choose a “compare” (source) branch and a “base” (destination) branch
  - Write a description and give it a name
  - Submit the PR and usually you assign specific people to look at it

Pull Request

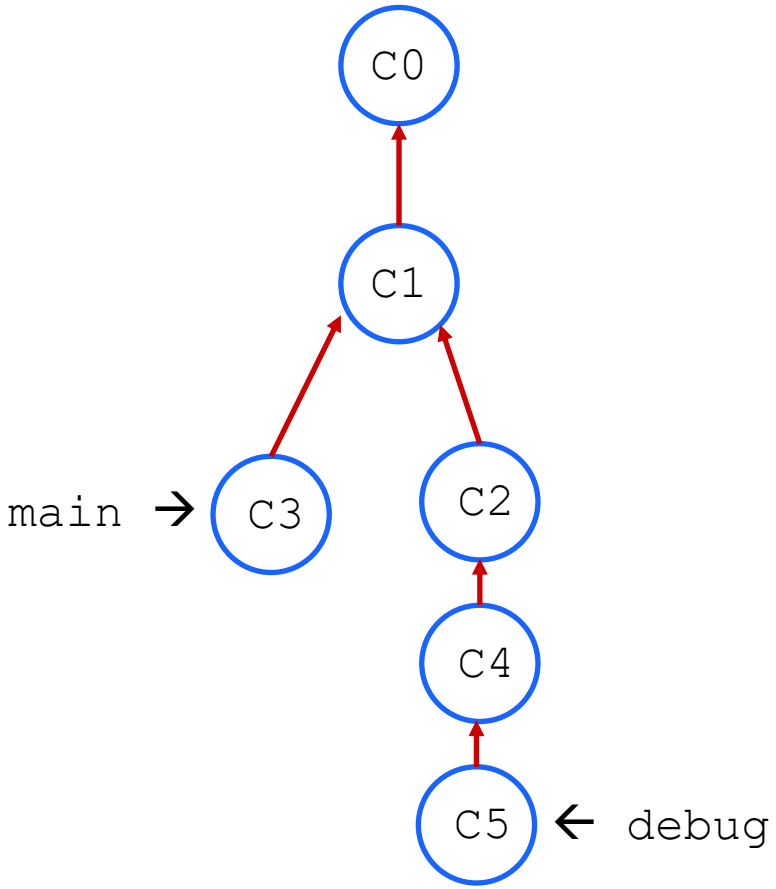
- Go to github repository, there should be a tab called “pull requests” you can use near the top
  - You choose a “compare” (source) branch and a “base” (destination) branch
  - Write a description and give it a name
  - Submit the PR and usually you assign specific people to look at it

- The branches can be rebased, merged, fast-forwarded, squashed, etc.

- A PR is just a wrapper around base git features to encourage good communication & organization:
  - You write up your changes
  - someone else reviews your changes and accepts or rejects them
    - If it is rejected, you can fix it and resubmit

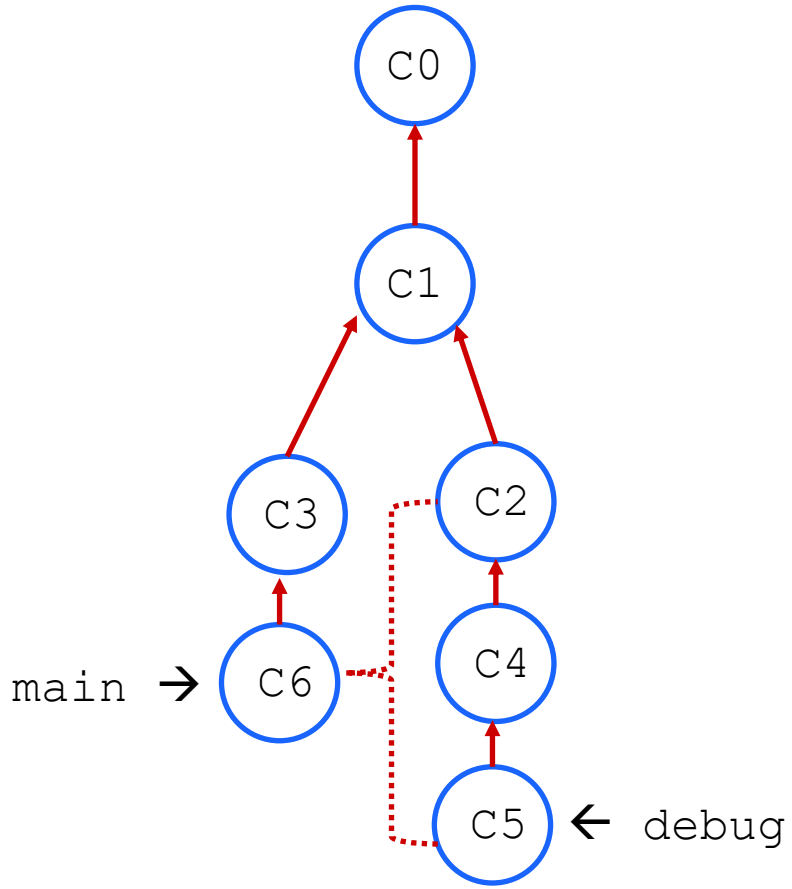
squash

- If there are many commits to merge-in, we can combine all commits into one commit



squash

- If there are many commits to merge-in, we can combine all commits into one commit
  - (assume we are on main)
  - `git merge --squash debug`
- There will not be any metadata tying the squash to the "source" commits.
  - So, we usually only use this when we are completely finished with a branch and want to merge it into another branch (e.g. main)
- Can also be done with a rebase



C6 includes all changes from c2, c4 and c5, but there is no relationship in the tree.

(The dotted line is just to explain, it won't show up in git)

Useful resources:

- Git reference: https://git-scm.com/docs
  - Has a cheat sheet: https://git-scm.com/cheat-sheet.html

- Learn Git Branching: https://learngitbranching.js.org/
  - Useful site to play around with branches and make sure you understand them
  - Pretty short imo, but still very useful
  - Has “levels” you complete
  - Can also be used as a sandbox to visualize how different commands affect branches

Ed Discussion

- See Ed Discussion

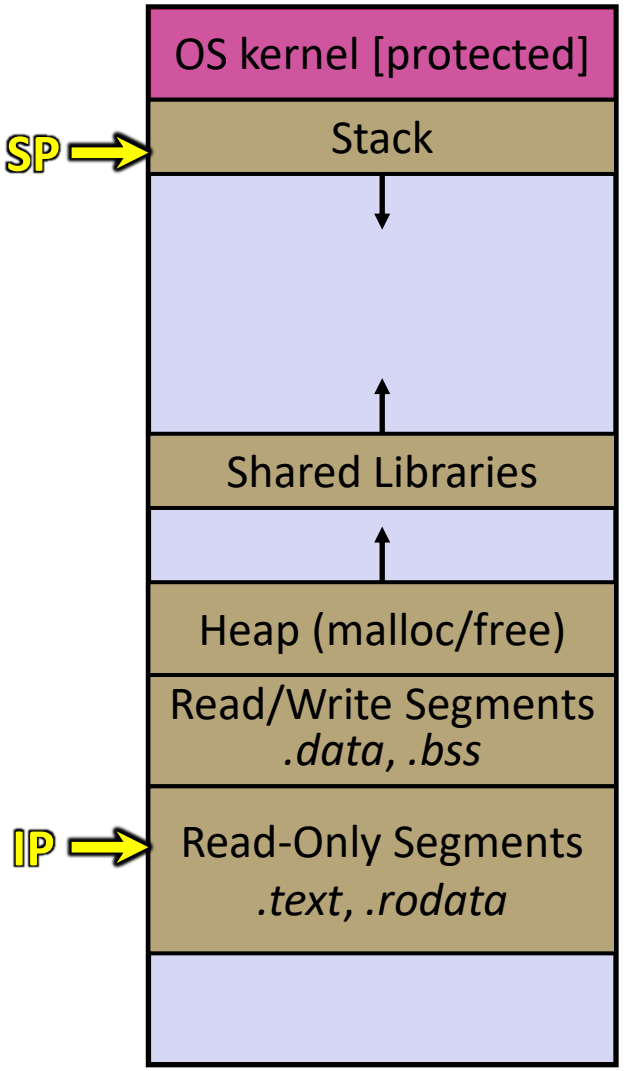
Lecture Outline

- git
- processes

Definition: Process

- Definition: An instance of a program that is being executed (or is ready for execution)

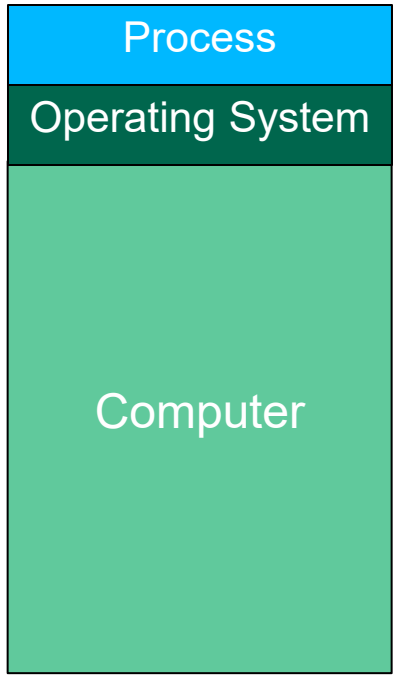
- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources



Computers as we know them now

- In CIS 2400, you learned about hardware, transistors, CMOS, gates, etc.
- Once we got to programming, our computer looks something like:

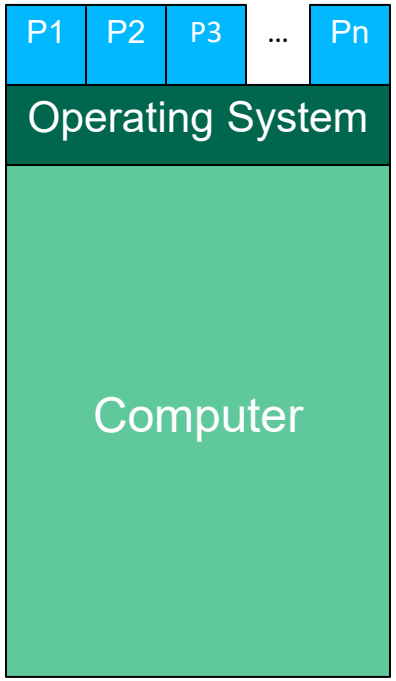
What is missing/wrong with this?



- This model is still useful, and can be used in many settings

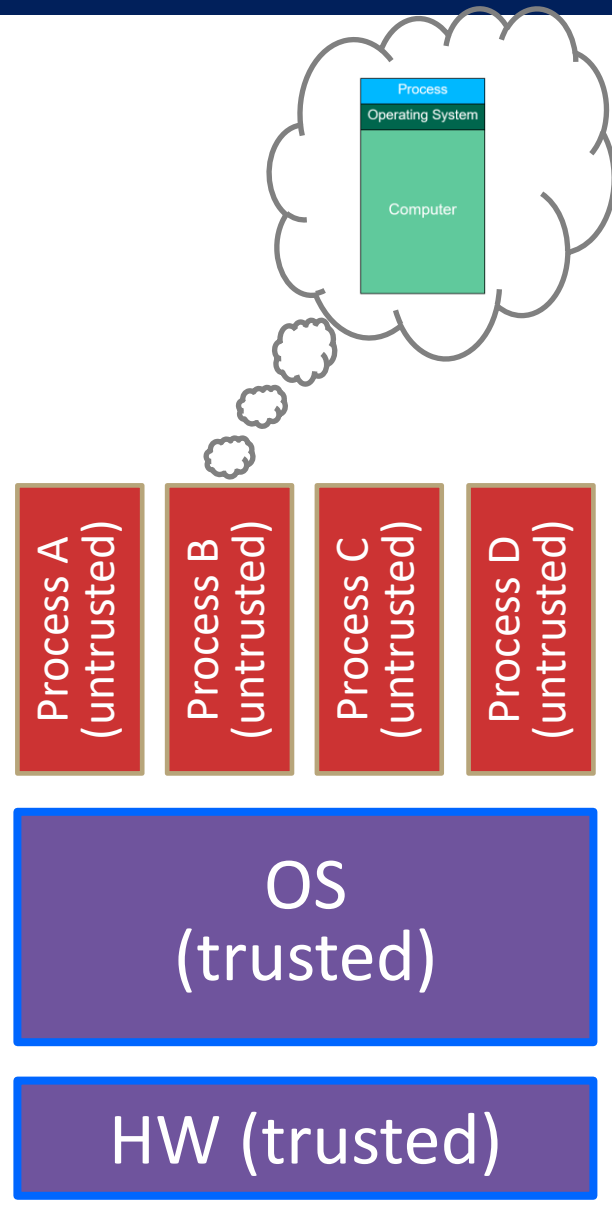
Multiple Processes

- Computers run multiple processes “at the same time”
- One or more processes for each of the programs on your computer
- Each process has its own…
  - Memory space
  - Registers
  - Resources

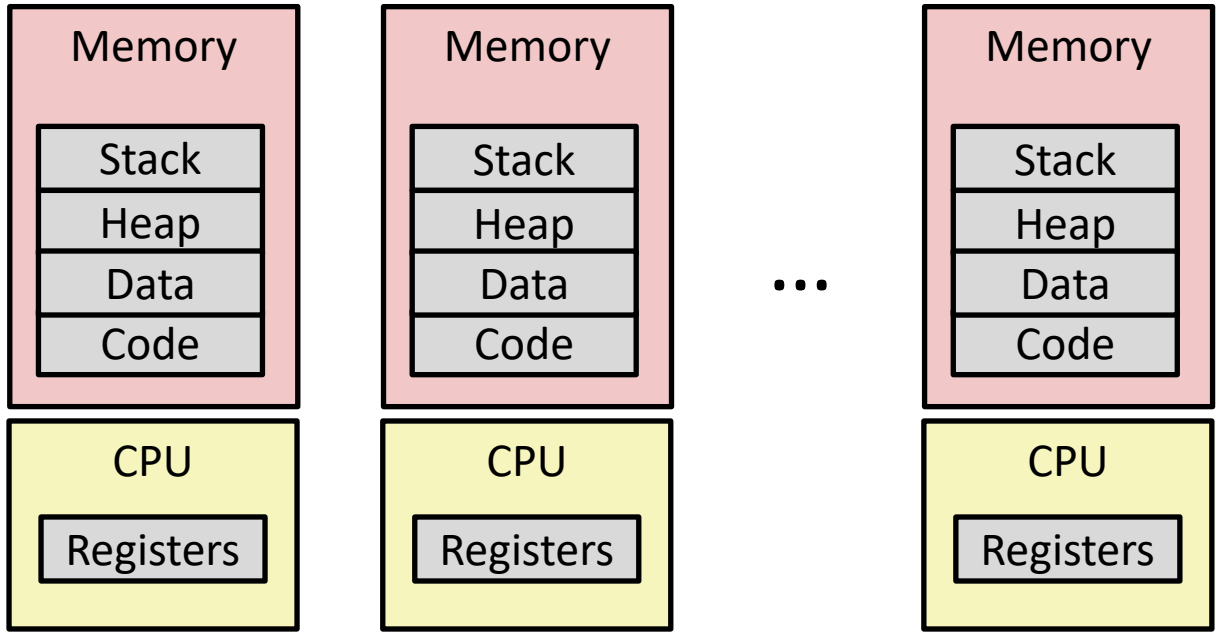


OS: Protection System

- OS isolates process from each other
  - Each process seems to have exclusive use of memory and the processor.
    - This is an **illusion**
    - More on Memory when we talk about virtual memory later in the course
  - OS permits controlled sharing between processes
    - E.g. through files, the network, etc.
- OS isolates itself from processes
  - Must prevent processes from accessing the hardware directly

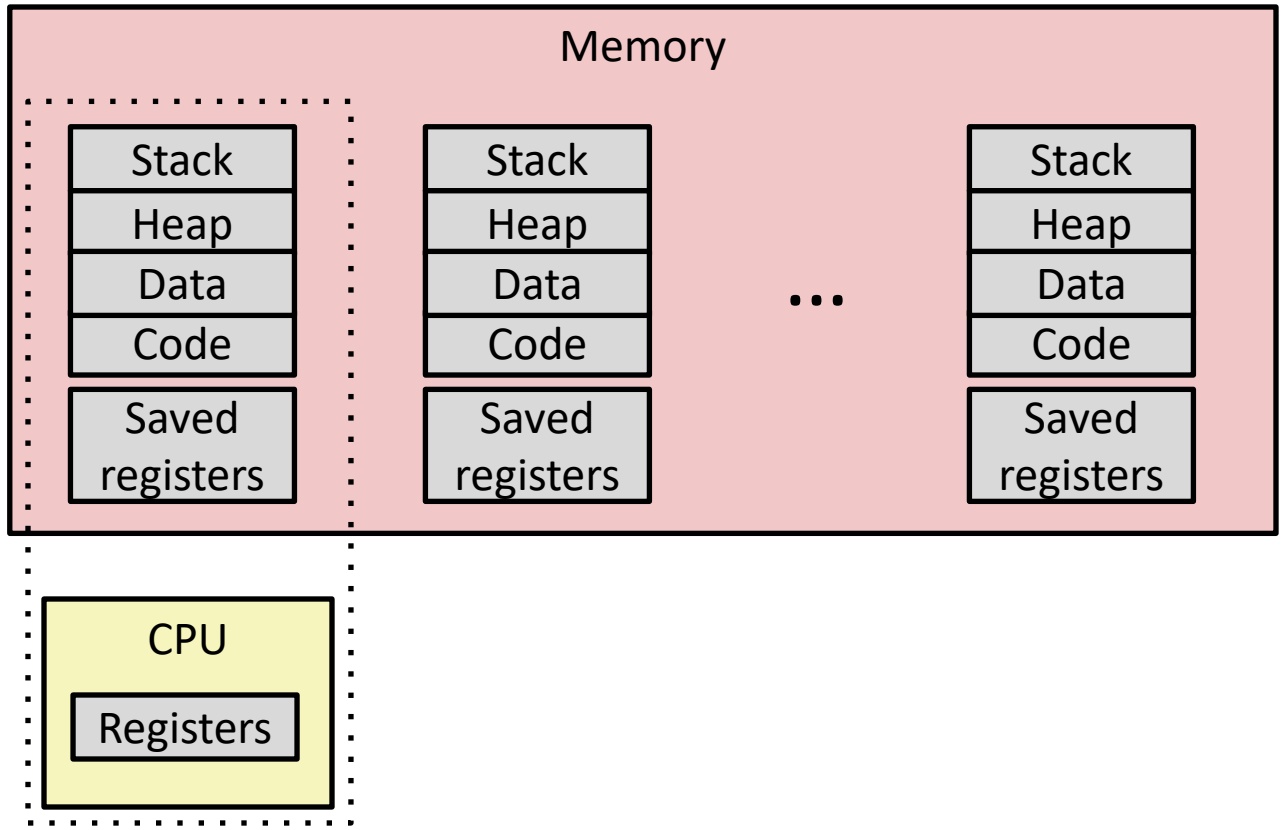


Multiprocessing: The Illusion



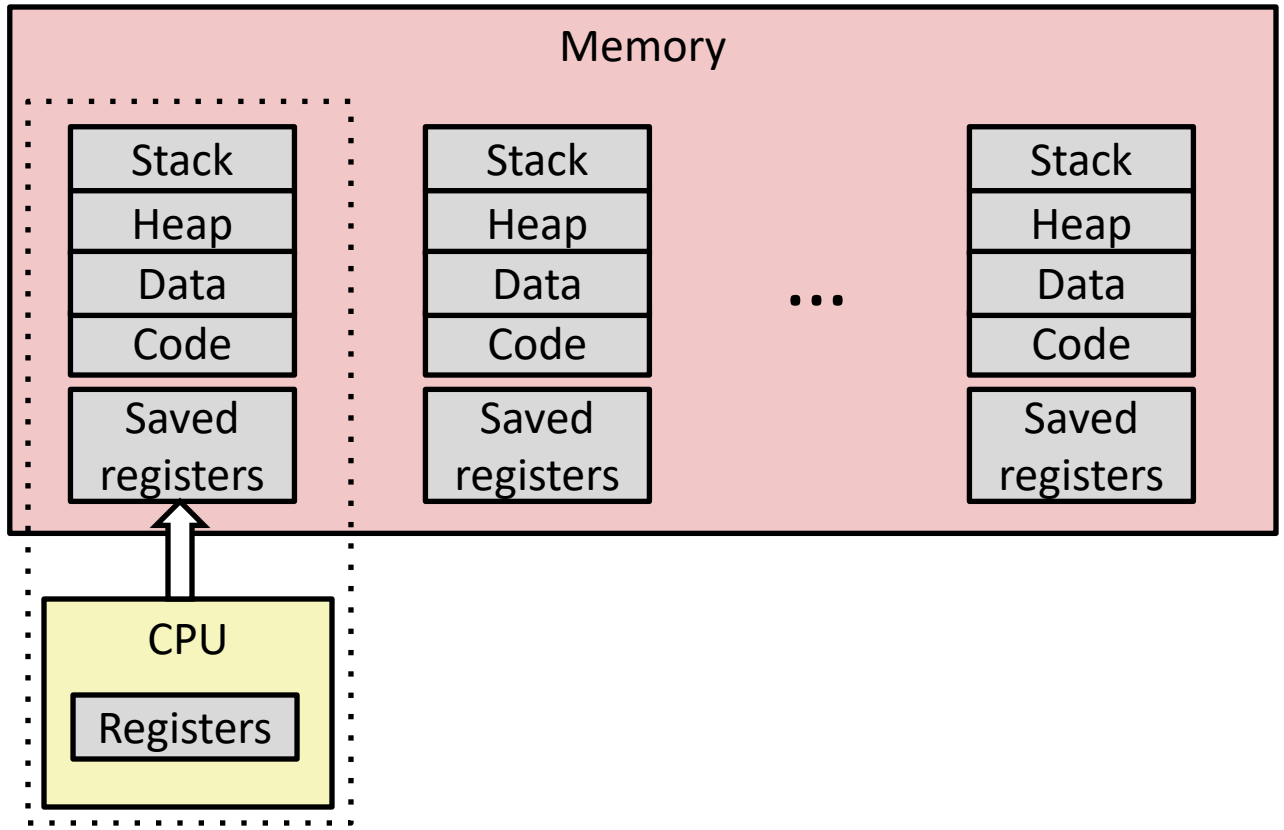
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing: The (Traditional) Reality



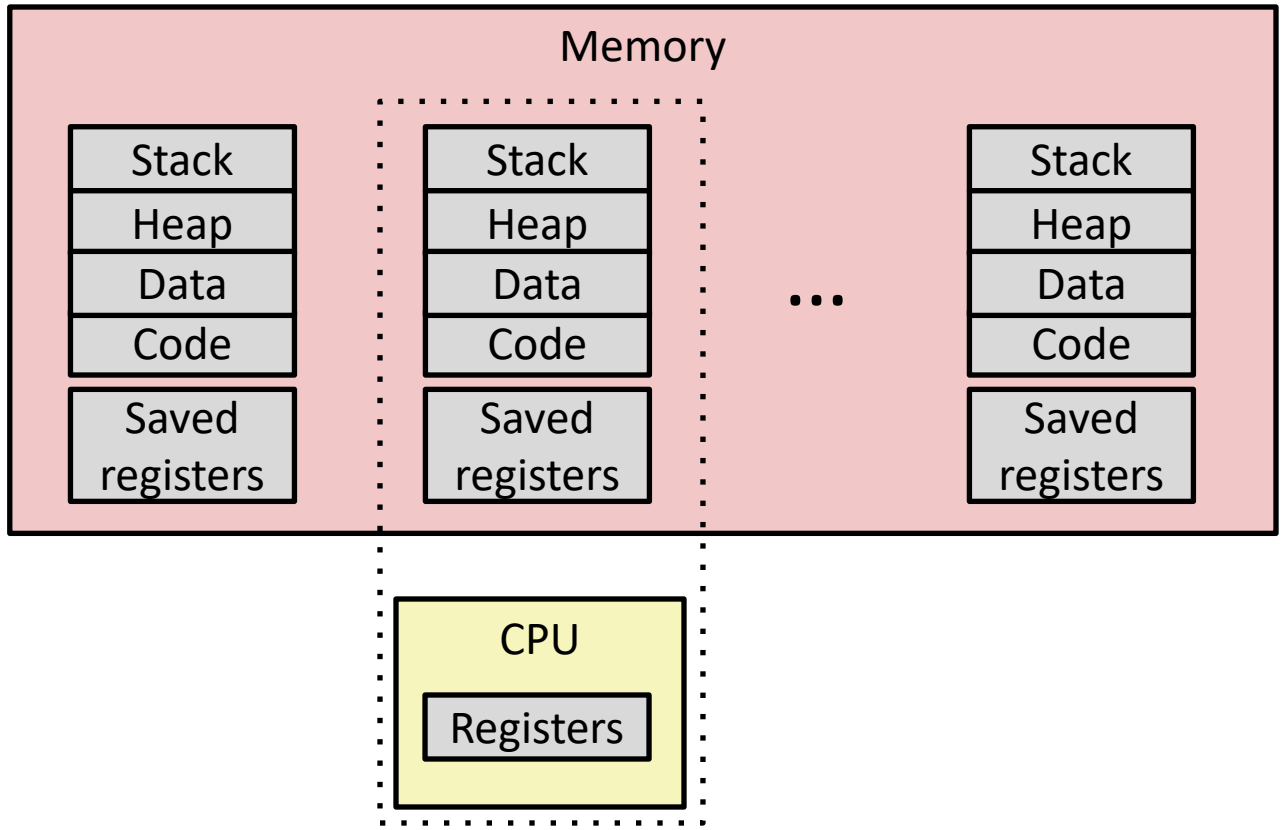
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



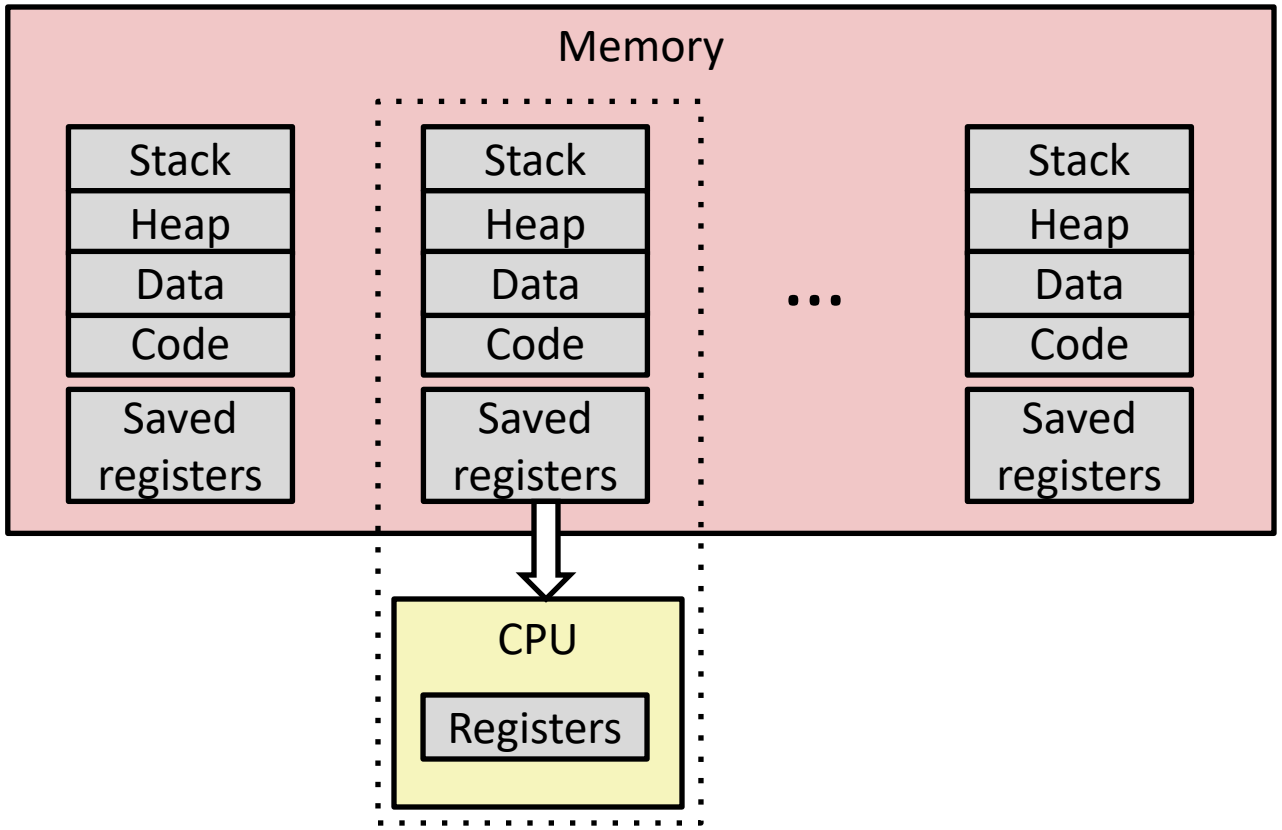
1. Save current registers in memory

Multiprocessing: The (Traditional) Reality



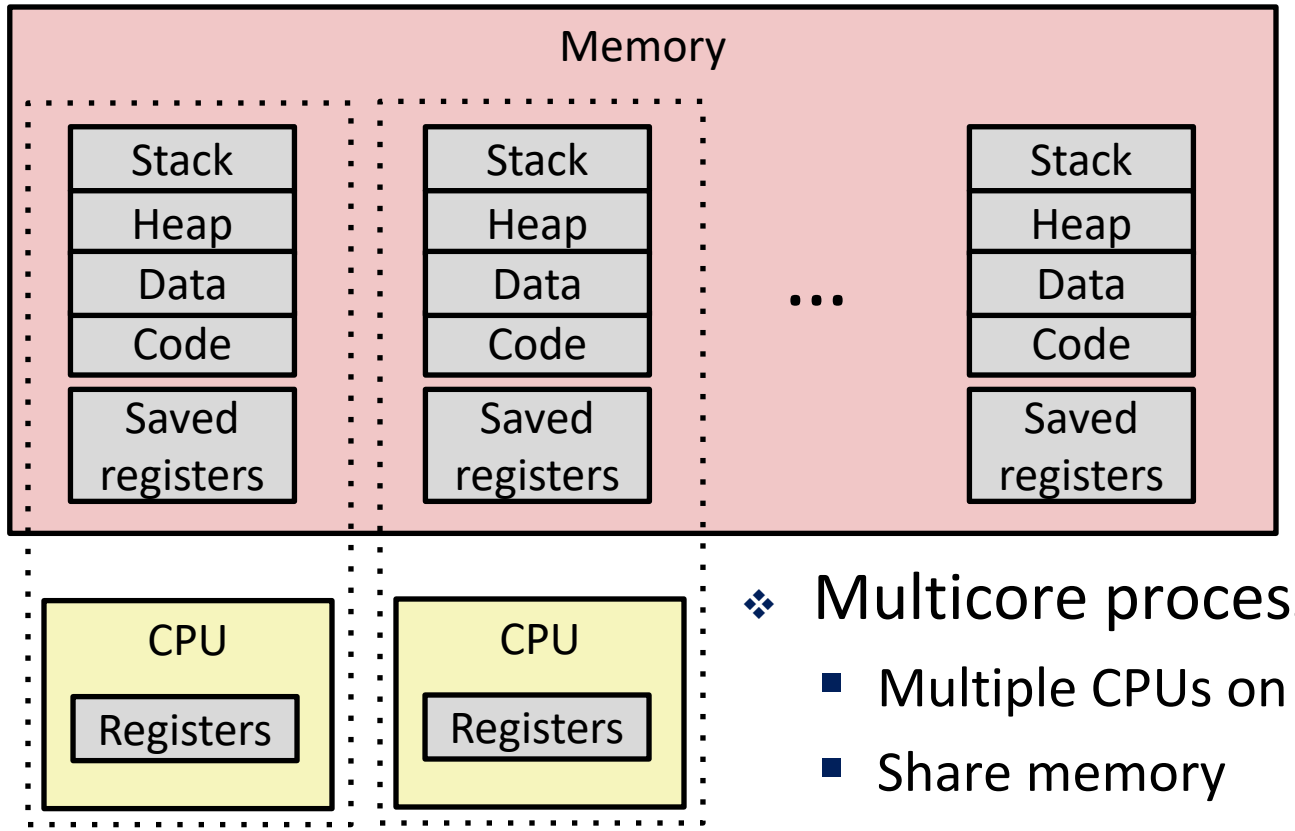
1. Save current registers in memory
2. Schedule next process for execution

Multiprocessing: The (Traditional) Reality



1. Save current registers in memory
2. Schedule next process for execution
3. Load saved registers and switch address space (context switch)

Multiprocessing: The (Modern) Reality



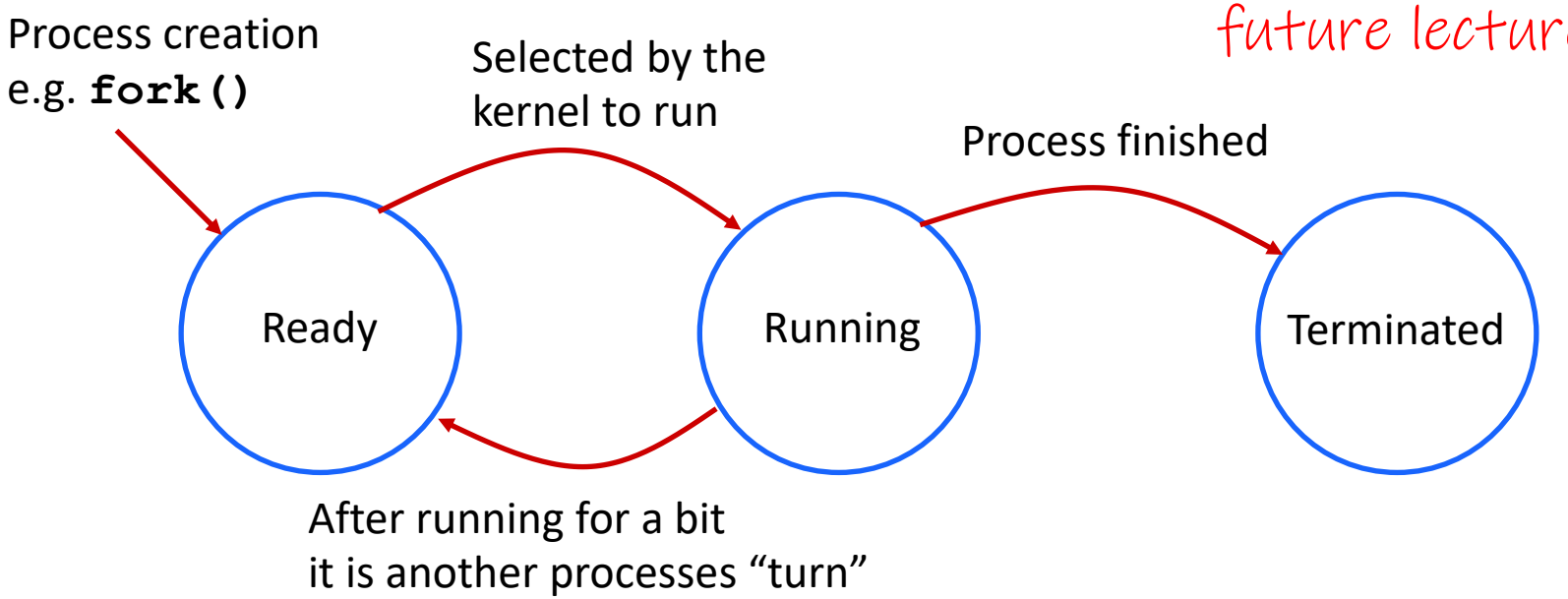
- Multicore processors
  - Multiple CPUs on single chip
  - Share memory
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
  - This is called “Parallelism”

OS: The Scheduler

- When switching between processes, the OS will run some kernel code called the “Scheduler”.
  - Switching between processes is called a **context switch**.
- The scheduler can interrupt a process mid-execution to run some other process.
- It is responsible for choosing which processes are run and does its best to be fair* (Being fair is rather complex).
- **We often simplify this to think of scheduling (and thus the order processes run) as non-deterministic.***

Process State Lifetime (incomplete)

More states in future lectures



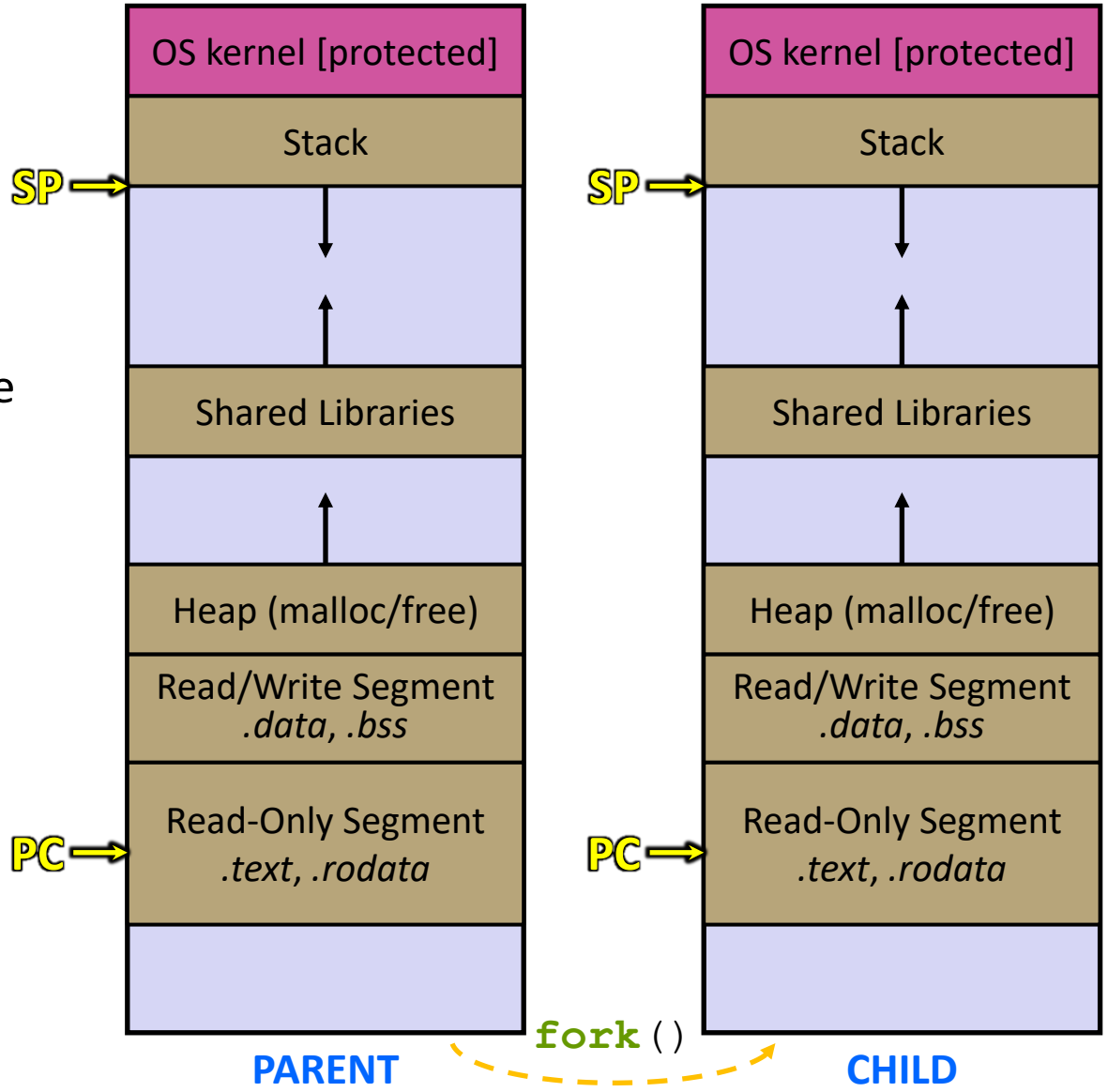
Processes can be "interrupted" to stop running. Through something like a hardware timer interrupt

Creating New Processes

- `pid_t fork();`
  - Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
    - *almost everything
  - The new process has a separate virtual address space from the parent
  - Returns a `pid_t` which is an integer type.

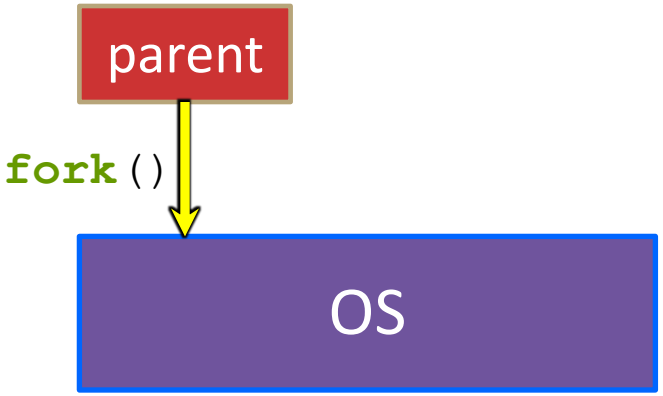
`fork()` and Address Spaces

- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



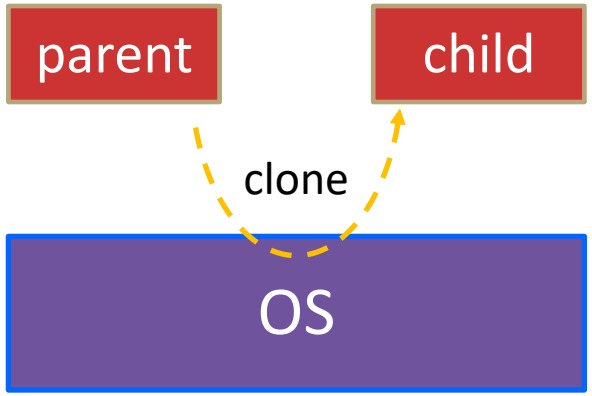
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



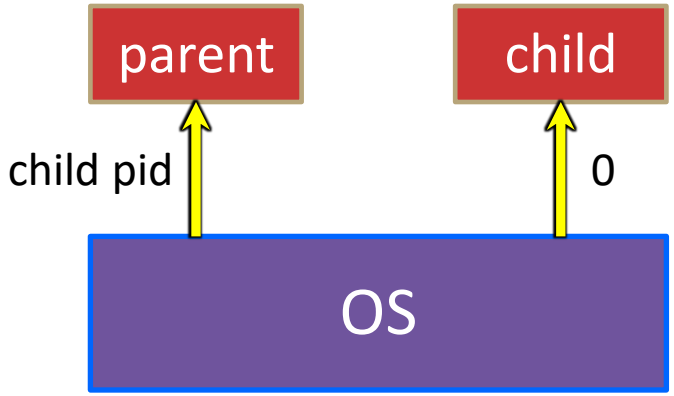
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



- Which process runs first?
  Up to the scheduler: non-determinsitic

"simple" `fork()` example

```
fork();
cout << "Hello!" << endl;
```

- What does this print?

"simple" `fork()` example

Parent Process (PID = X)

```
fork();
cout << "Hello!" << endl;
```

Child Process (PID = Y)

```
fork();
cout << "Hello!" << endl;
```

- What does this print?

- "Hello!\n" is printed twice

Ed Discussion

```
fork();
fork();
cout << "Hello!" << endl;
```

- What does this print?

Ed Discussion

```
int x = 3;
fork();
x++;
cout << x << endl;
```

- What does this print?

Ed Discussion

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }

  cout << ":)\n";
  return EXIT_SUCCESS;
}
```

Ed Discussion

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

- What does this print?

fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

fork()

fork() example

Parent Process (PID = X)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Child Process (PID = Y)

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

fork_ret = Y

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Prints "Parent"

fork_ret = 0

```
pid_t fork_ret = fork();

if (fork_ret == 0) {
  cout << "Child!" << endl;
} else {
  cout << "Parent!" << endl;
}
```

Prints "Child"

Which prints first?

fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

Always prints "Hello"

fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

Always prints "Hello"

fork() example

Parent Process (PID = X)

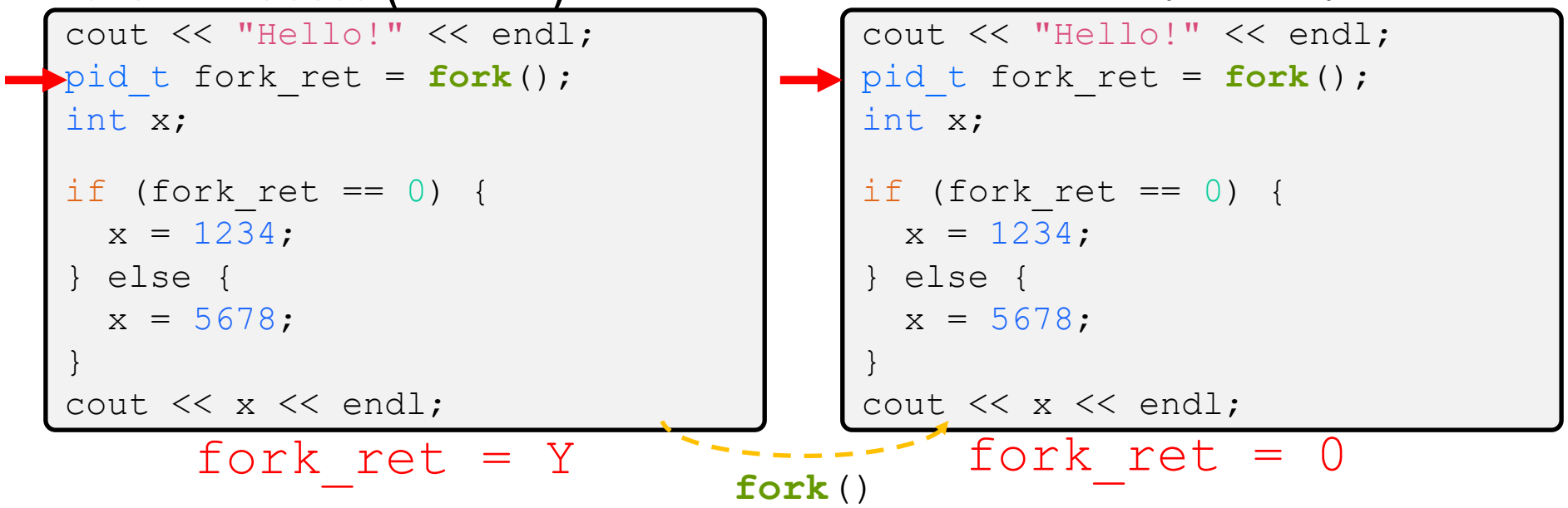
```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```



Always prints "Hello"

Does NOT print "Hello"

fork() example

Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 1234;
} else {
  x = 5678;
}
cout << x << endl;
```

Always prints "Hello"

Always prints "5678"

Always prints "1234"

Exiting a Process

```
void exit(int status);
```

- Causes the current process to exit normally
- Automatically called by **`main()`** when main returns
- Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
  - This is the same int returned by **`main()`**
- The exit status is accessible by the parent process with **`wait()`** or **`waitpid()`**.
  (more on these in a future lecture)

Ed Discussion

```
int global_num = 1;

void function() {
  global_num++;
  cout << global_num << endl;
}

int main() {
  pid_t id = fork();

  if (id == 0) {
    function();
    id = fork();
    if (id == 0) {
      function();
    }
    return EXIT_SUCCESS;
  }

  global_num += 2;
  cout << global_num << endl;
  return EXIT_SUCCESS;
}
```

- How many numbers are printed? What number(s) get printed from each process?

Ed Discussion

- Are the following outputs possible?

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
  fork_ret = fork();
  if (fork_ret == 0) {
    cout << "Hi 3!" << endl;
  } else {
    cout << "Hi 2!" << endl;
  }
} else {
  cout << "Hi 1!" << endl;
}
cout << "Bye" << endl;
```

Sequence 1:

```
Hi 1
Bye
Hi 2
Bye
Bye
Hi 3
```

Sequence 2:

```
Hi 3
Hi 1
Hi 2
Bye
Bye
Bye
```

Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent address space
  - Each process is scheduled by the OS
    - Without using some functions we have not talked about (yet), there is no way to guarantee the order processes are executed
  - Processes are created by fork() system call
    - Only difference between the parent and child immediately after fork() is their process id and the return value from fork() each process gets

execvp()

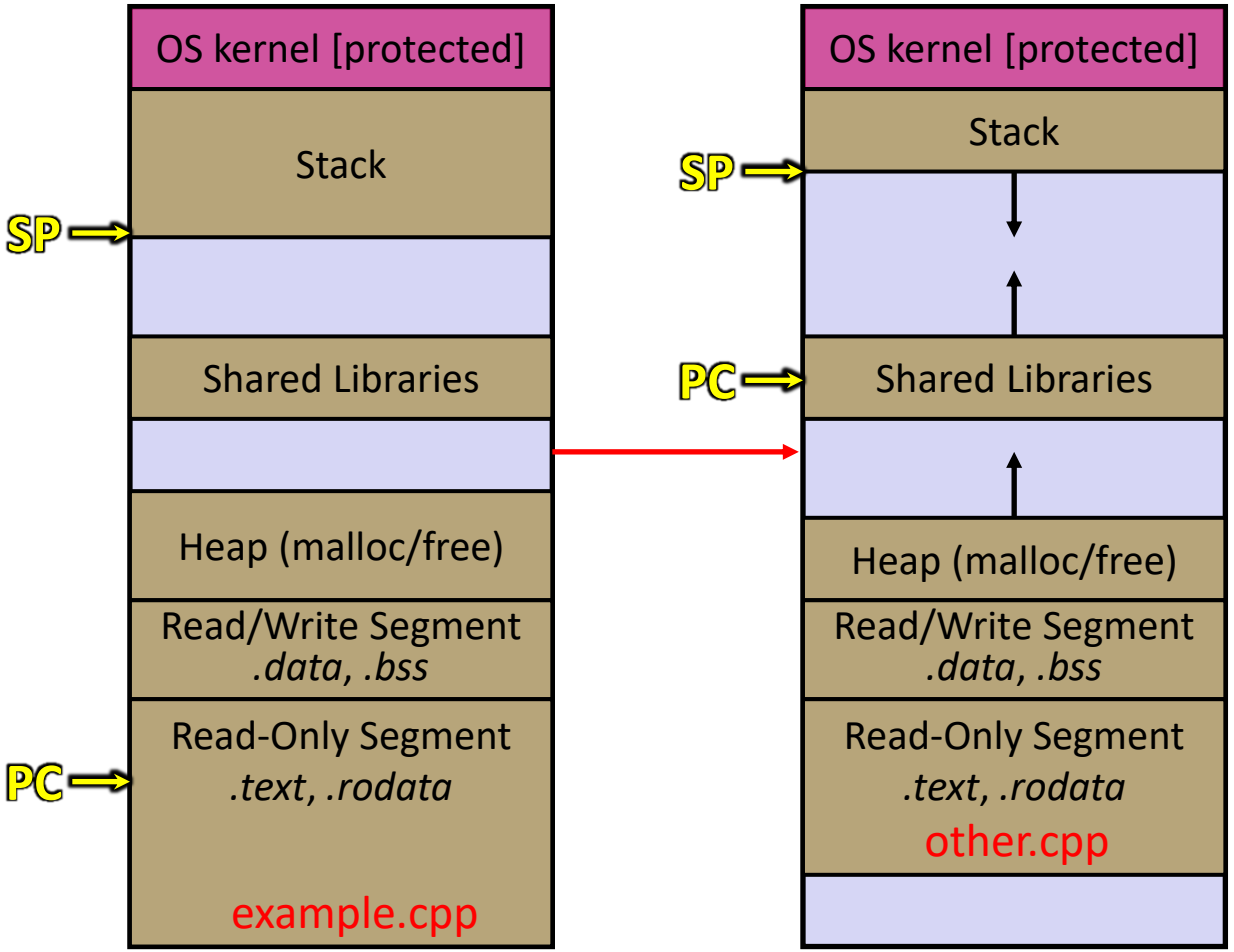
- execvp

```
int execvp(const char *file,
           char* const argv[]);
```

- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C/C++ program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL/nullptr as the last entry of the array
- Returns -1 on error. Does NOT return on success

Exec Visualization

- Exec takes a process and discards or “resets” most of it



NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens if we open some files before exec?
  - What happens if we replace stdout with a file?

- NOTE: When a process exits, then it will close all of its open files by default

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens to allocated memory when we call exec?

Ed Discussion

- In each of these, how often is `":)"` printed? Assume functions don't fail

```
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[] = {"echo",
                    "hello",
                    nullptr};
    execvp(argv[0], argv);
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

```
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    return EXIT_SUCCESS;
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

Ed Discussion

- What's wrong with this code? It tries to clone a repo and then compile the code in that repo. Assume the argv[]'s are set up correctly to do this.

```
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[] = {"git", "clone", "repo_name.git", nullptr};
    execvp(argv[0], argv);
  }

  pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[] = {"make", "-C", "repo_name", nullptr};
    execvp(argv[0], argv);
  }
  return EXIT_SUCCESS;
}
```

That's all for now!

- Next time:
  - More Processes ☺

- Hopefully you are doing well ☺