Week 2

Introduction to Process Management

Oana Balmau

January 10, 2023

Schedule Highlights

Lab recording on grading infrastructure will be posted on MyCourses by the end of the week. Tutorial coming soon. Have a look and try the HelloWorld if you have not done so already.

Topic	Monday	Tuesday	Wednesday	Thursday	Friday
Week 1 **Introduction**	jan 2	jan 3	jan 4 – first day of class 😊	jan 5 **Course logistics and Intro to OS**	jan 6
Week 2 **Process Management**	jan 9 Workflow: working with mimi and GitLab, Git basics	jan 10 **Intro to Process Management (1/2)** Optional reading: OSTEP Chapters 3 – 7	jan 11	jan 12 **Intro to Process Management (2/2)**	jan 13
Week 3 **Process Management**	jan 16 C Review: C Basics	jan 17 **Synchronization Primitives (1/2)** Optional reading: OSTEP Chapters 25 – 32 **add/drop deadline**	jan 18 OS Shell Assignment Released	jan 19 **Synchronization Primitives (2/2)** OS Shell Assignment Overview – with Jiaxuan	jan 20
Week 4 **Process Management**	jan 23 C Tools: GDB basics	jan 24 **Multi-process Structuring (1/2)** **Team registration deadline**	jan 25	jan 26 **Multi-process Structuring (2/2)**	jan 27
Week 5 **Process Management**	jan 30 C Review: Pointers & Memory Allocation I	jan 31 **Multithreading (1/2)** Practice Exercises Sheet: Process Management	feb 1	feb 2 **Multithreading (2/2)**	feb 3

Autograder will start sending you email

Teams and GitLab account need to be set up by this day

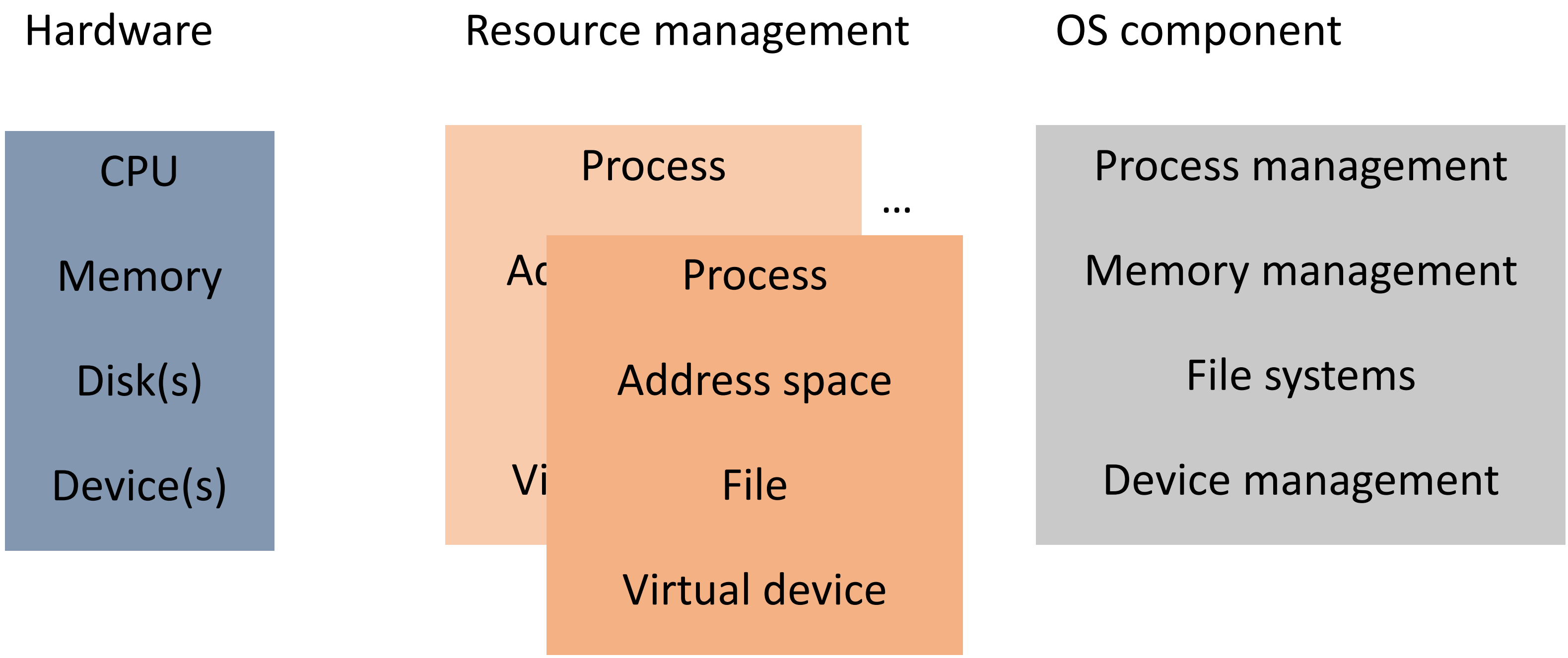
Use Ed Megathread if you have issues

Recap from Week 1

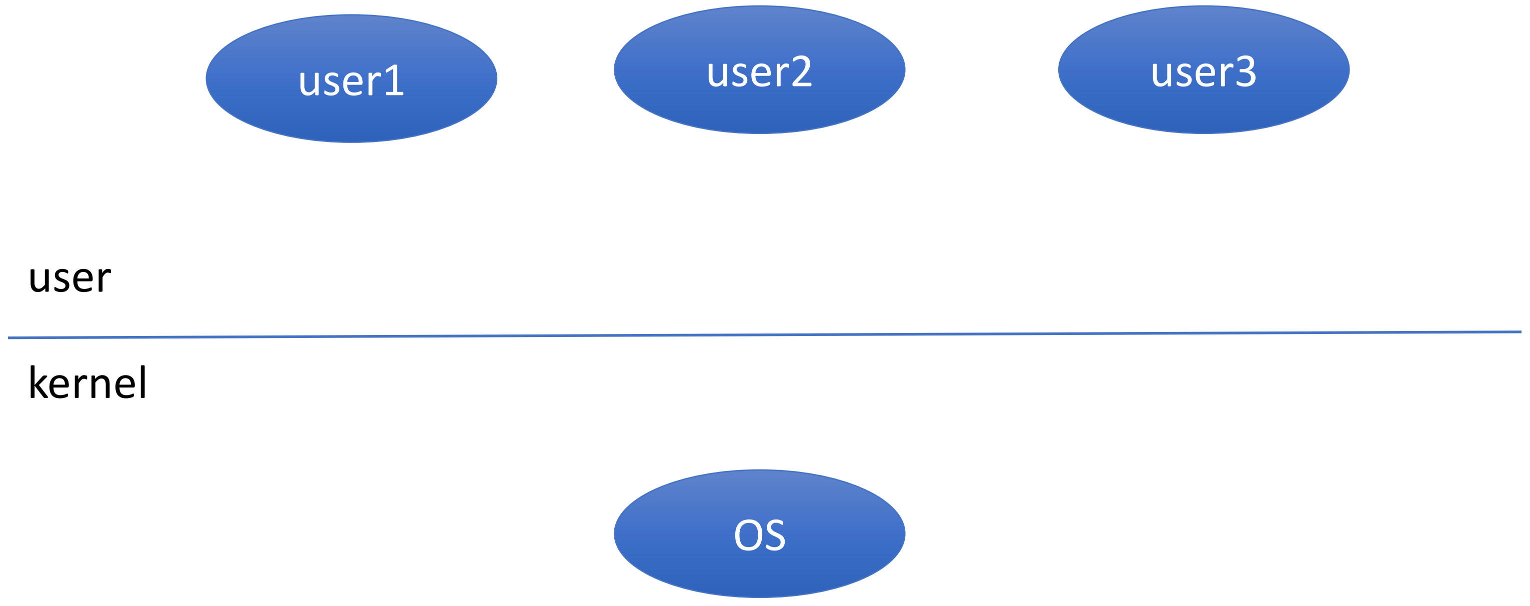
- What does the OS do?
- Where does the OS live?
- OS interfaces
- OS control flow
- OS structure

Recap from Week 1:
What does the OS do?

- **Abstraction and Resource management**



Recap from Week 1: User/OS Separation



Recap from Week 1:
Kernel mode vs User mode

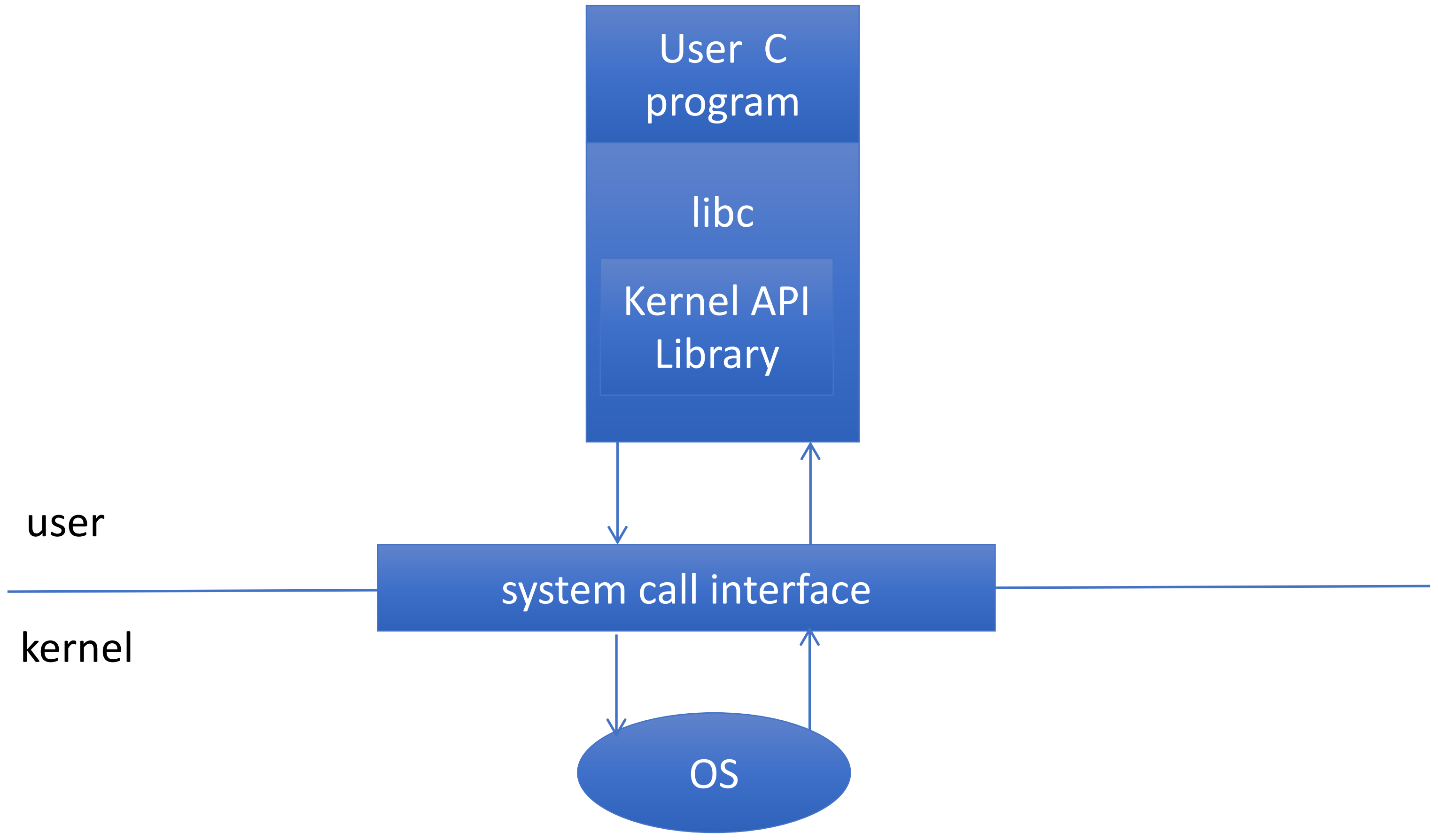
Kernel Mode

- Privileged instructions:
  - Set mode bit
  - …
- Direct access to all of memory
- Direct access to devices

User Mode

- No privileged instructions:
  - Set mode bit
  - …
- No direct access to all of memory
- No direct access to devices

Recap from Week 1:
System calls, kernel API, libc



Recap from Week 1
System Calls, Traps Interrupts

- System calls
- Traps
- Interrupts

Recap from Week 1
System Calls, Traps Interrupts

- System calls
  - Are the *only* interface from program to OS
  - Narrow interface essential for integrity of OS
- Traps
  - Trap is generated by CPU as a result of error
  - Works like an “involuntary” system call
- Interrupts
  - Generated by a device that needs attention

Recap from Week 1
OS Control Flow: Event-Driven Program

- Nothing to do

Do nothing

- Interrupt (from device)
- Trap (from process)
- System call (from process}

Start running

Key Concepts for this week

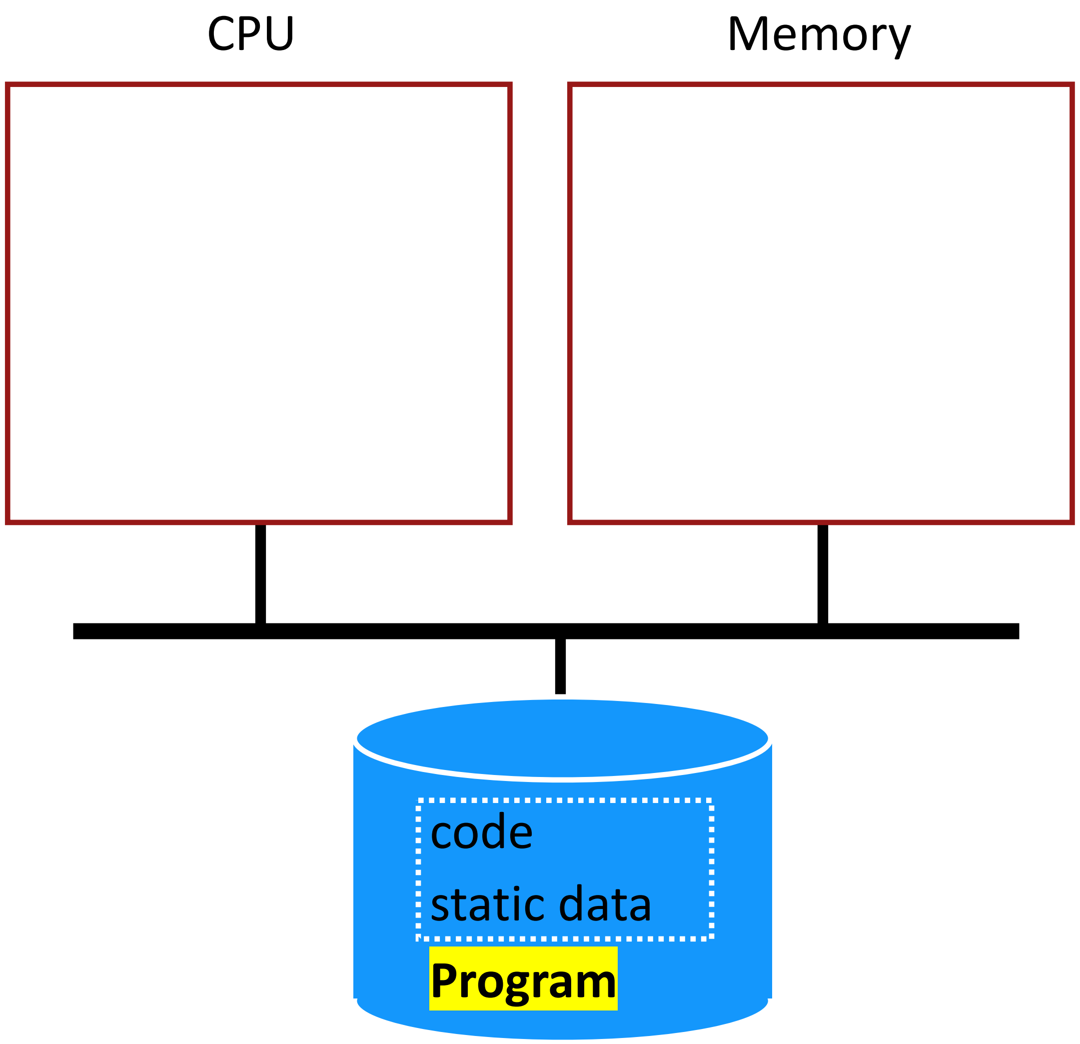
- Process
- Linux process tree
- Process switch
- Process scheduler

Process vs Program

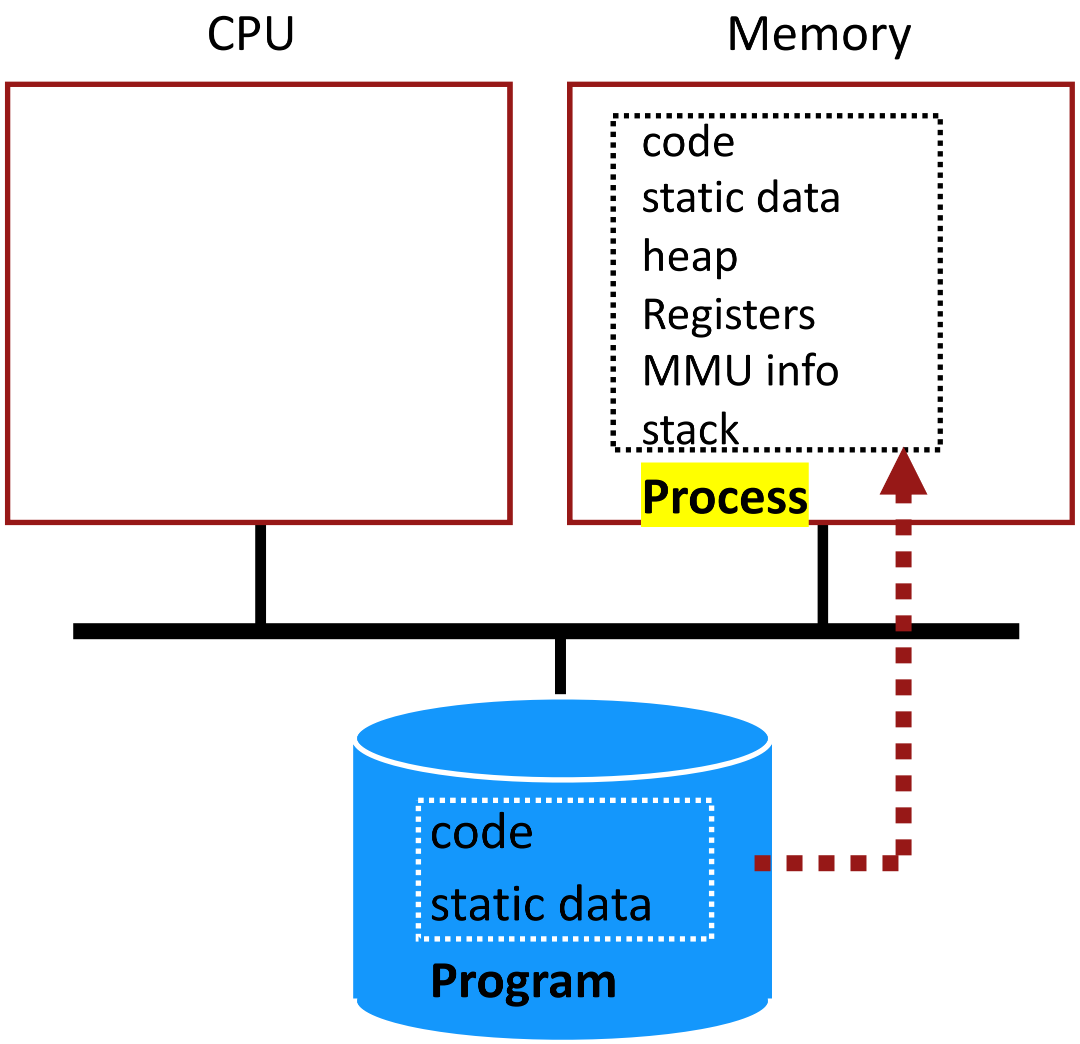
What is a Process?

- **Process = program in execution**
- Program
  - Executable code
  - Usually represented by a file on disk
- Process
  - Executing code
  - Usually represented in memory

Process Creation



Process Creation



What is a Process?

- Process: An execution stream in the context of a process state

What is a Process?

- Process: An execution stream in the context of a process state
- What is an execution stream?
  - Stream of executing instructions
  - Running piece of code
  - “thread of control”

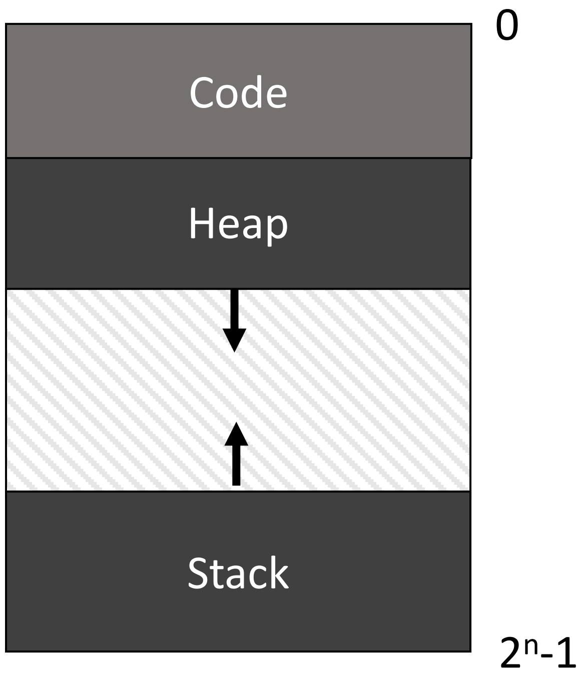
What is a Process?

- Process: An execution stream in the context of a process state
- What is process state?
  - Everything that the running code can affect or be affected by
  - Registers
    - General purpose, floating point, status, program counter, stack pointer
  - Address space
  - Heap, stack, and code
  - Open files

Address Space Review

Static and dynamic components

- Static: Code and some global variables
- Dynamic: Stack and Heap



Motivation for Dynamic Memory Allocation

Why do processes need dynamic allocation of memory?

- Do not know amount of memory needed at compile time.
- Must be pessimistic for static memory allocation.
- If statically allocate enough for worst possible case, storage is used inefficiently.

Motivation for Dynamic Memory Allocation

Why do processes need dynamic allocation of memory?

- Recursive procedures
  - Do not know how many times procedure will be nested
- Complex data structures: lists and trees
  - `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`

Dynamic Memory Allocation

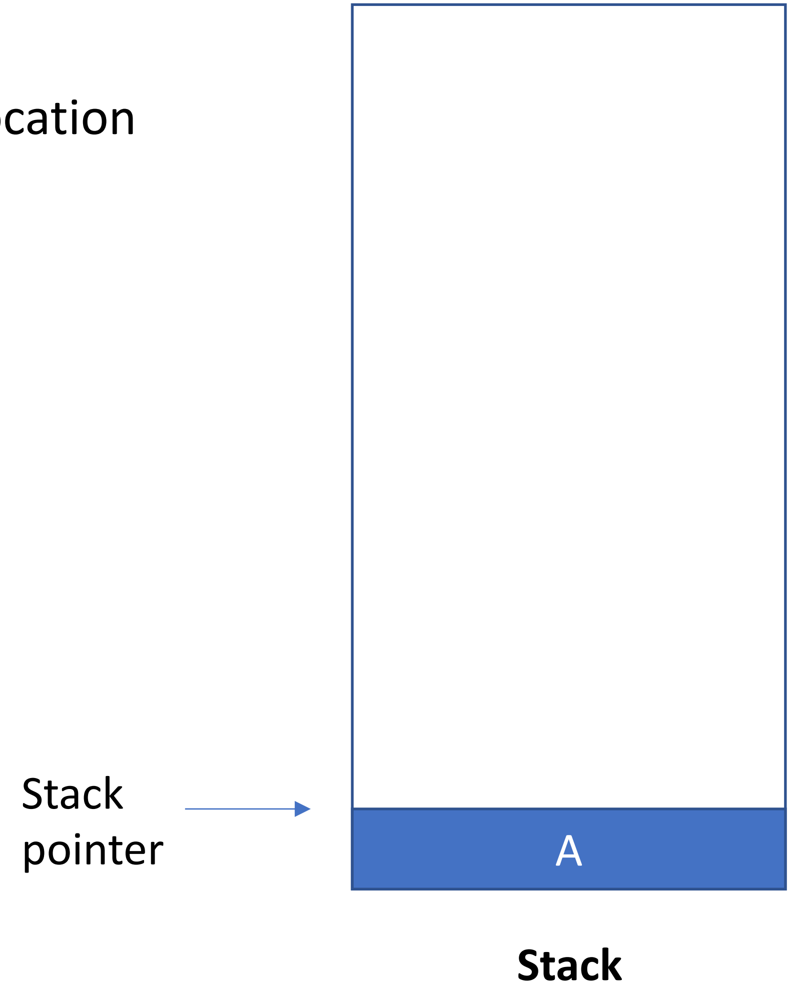
Two types of dynamic allocation

- Stack
- Heap

Stack

Memory is freed in opposite order from allocation

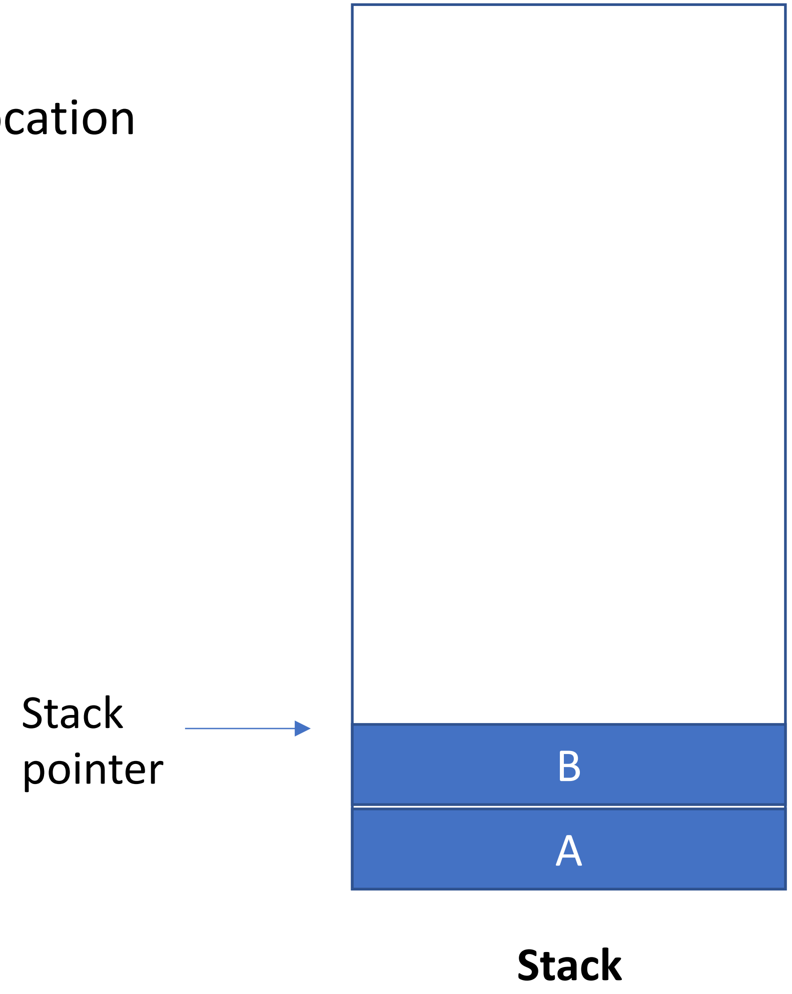
```
alloc(A);
```



Stack

Memory is freed in opposite order from allocation

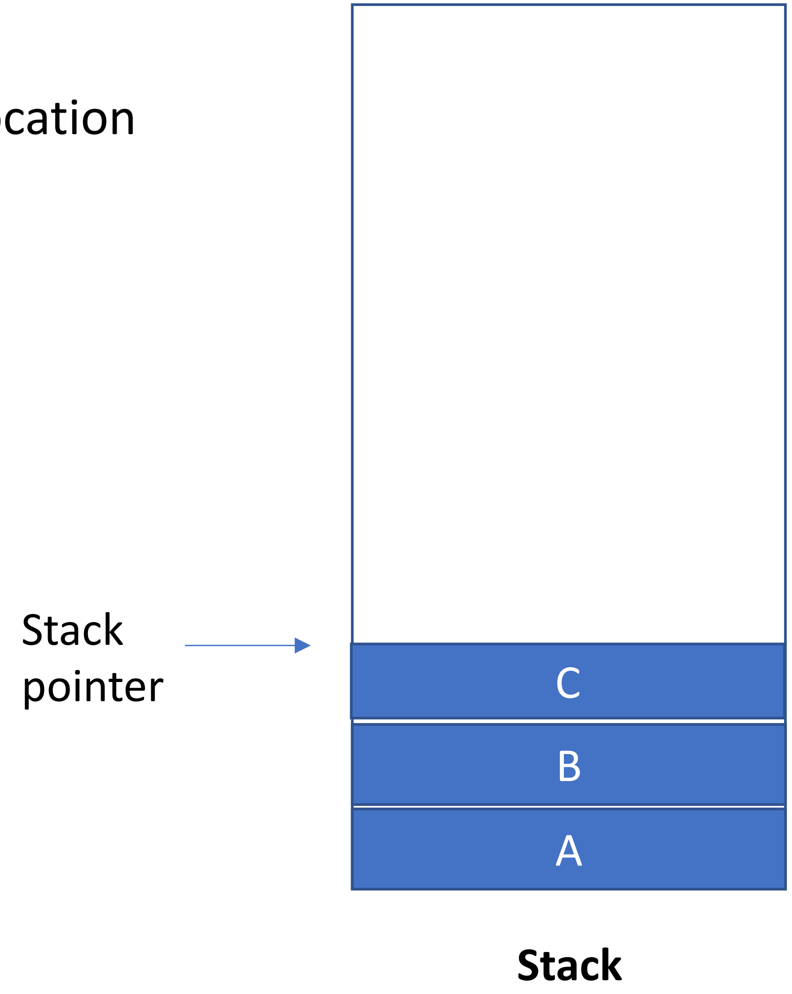
```
alloc(A);
alloc(B);
```



Stack

Memory is freed in opposite order from allocation

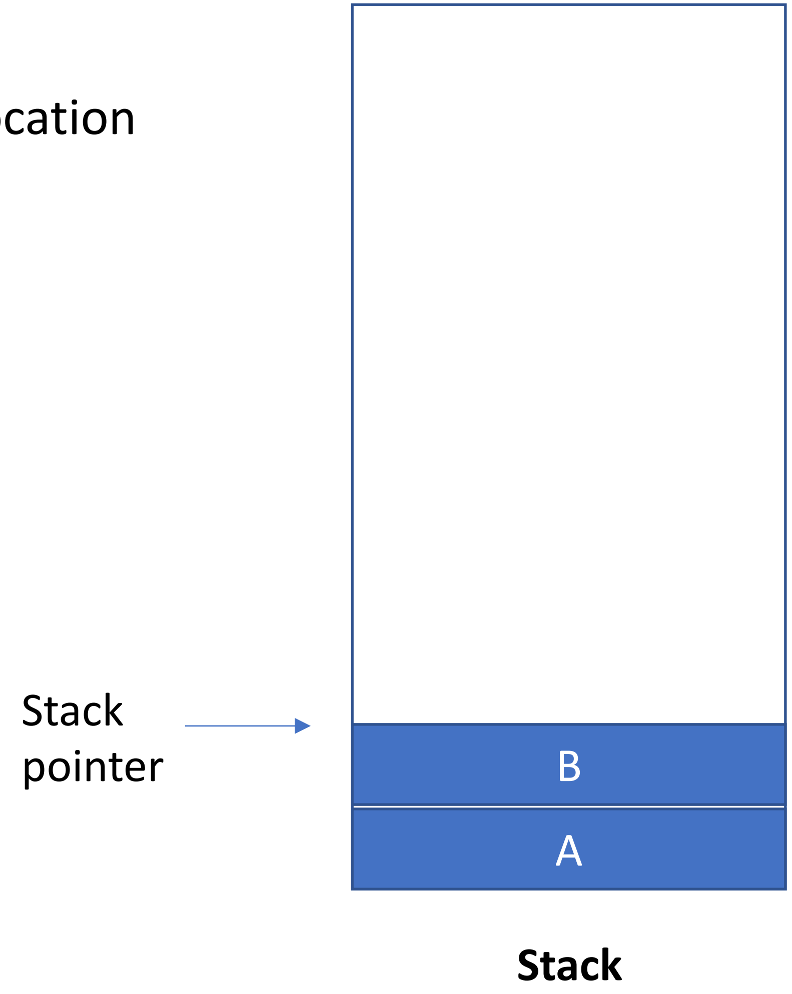
```
alloc(A);
alloc(B);
alloc(C);
```



Stack

Memory is freed in opposite order from allocation

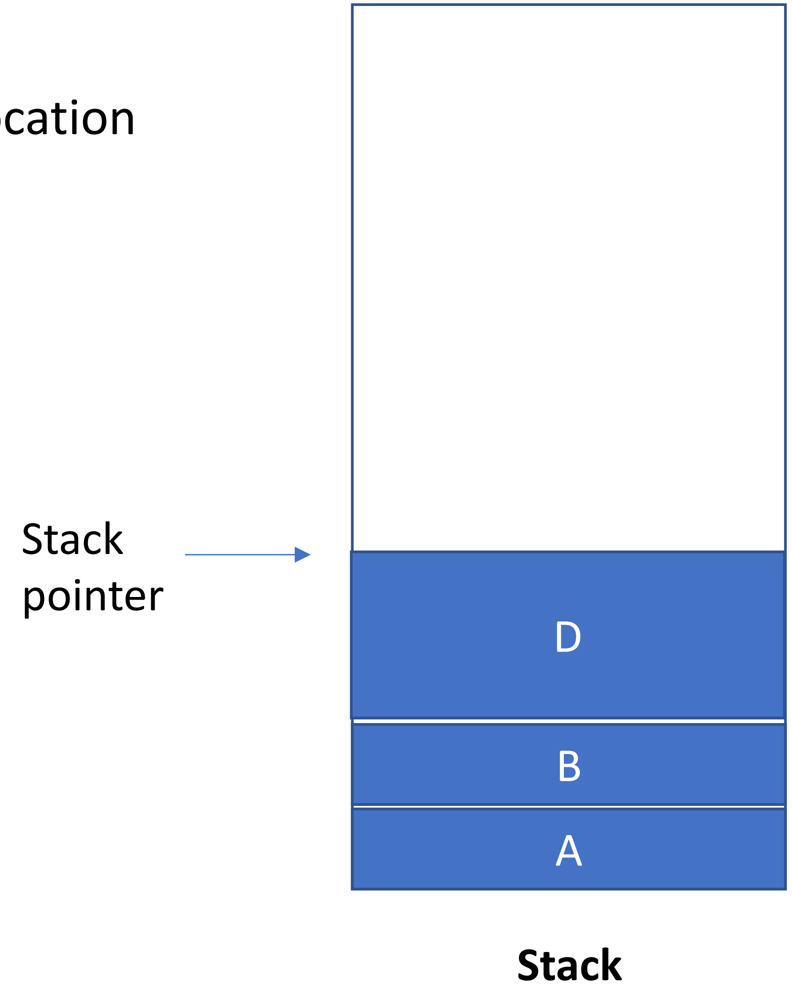
```
alloc(A);
alloc(B);
alloc(C);
free(C);
```



Stack

Memory is freed in opposite order from allocation

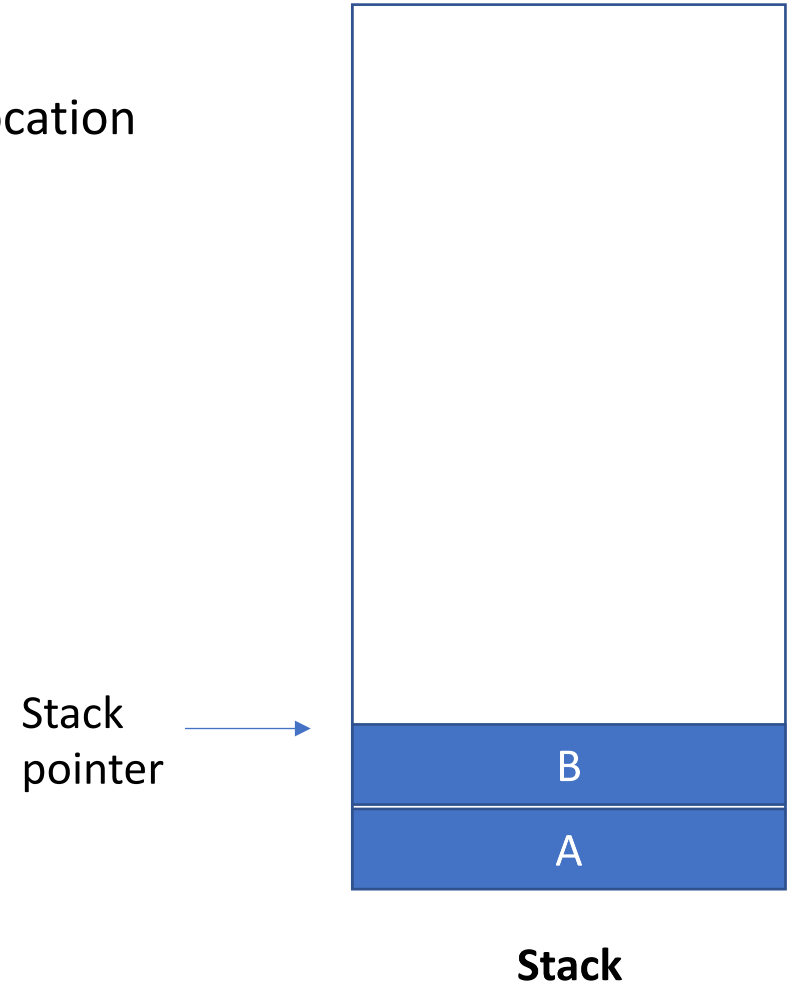
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
```



Stack

Memory is freed in opposite order from allocation

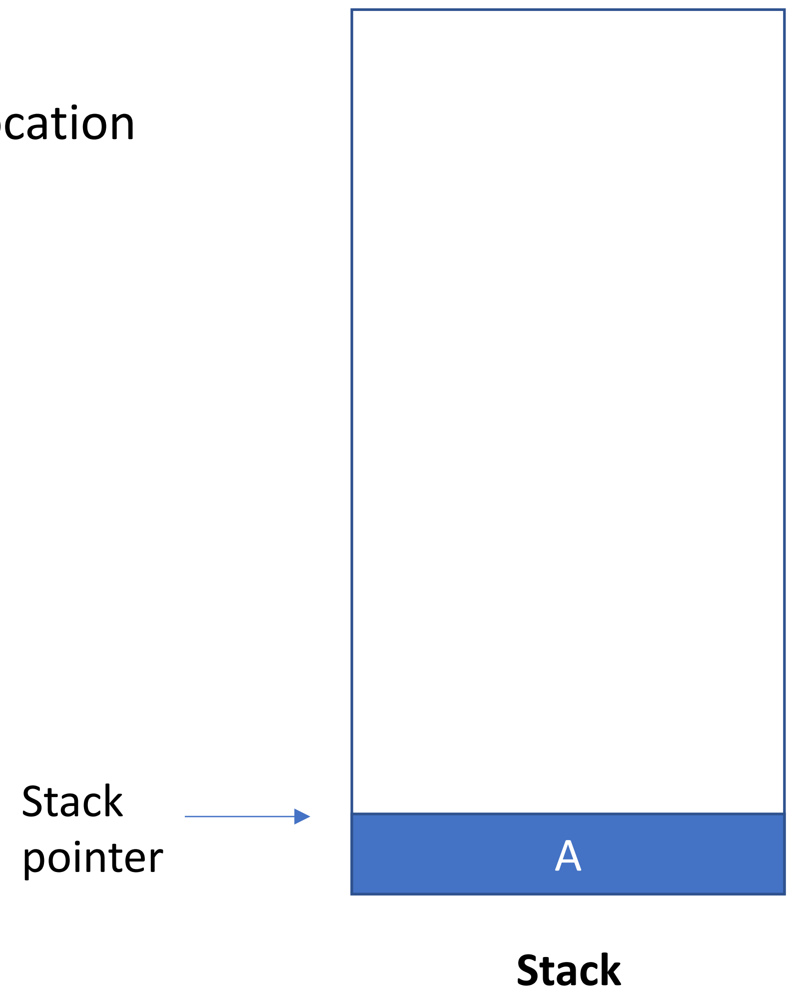
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
```



Stack

Memory is freed in opposite order from allocation

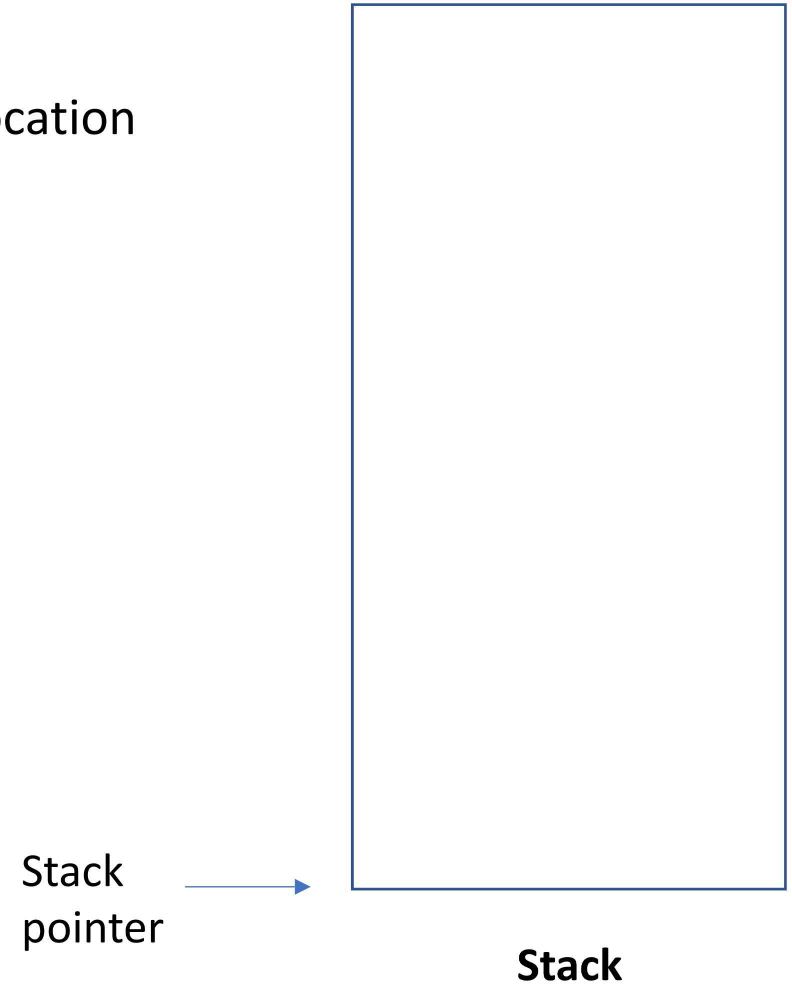
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```



Stack

Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```



Stack

Simple and efficient implementation:

- Pointer separates allocated and freed space
- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation

Stack management done automatically

Where are stacks used?

OS uses stack for procedure call frames (local variables and parameters)

```
main () {
      int A = 0;
      foo (A);
      printf(“A: %d\n”, A);
}

void foo (int Z) {
      int A = 2;
      Z = 5;
      printf(“A: %d Z: %d\n”, A, Z);
}
```

Where are stacks used?

OS uses stack for procedure call frames (local variables and parameters)

```
main () {
      int A = 0;
      foo (A);
      printf(“A: %d\n”, A);
}

void foo (int Z) {
      int A = 2;
      Z = 5;
      printf(“A: %d Z: %d\n”, A, Z);
}
```

```
Prints:

A: 2 Z: 5
A: 0
```

Heap

Allocate from any random location

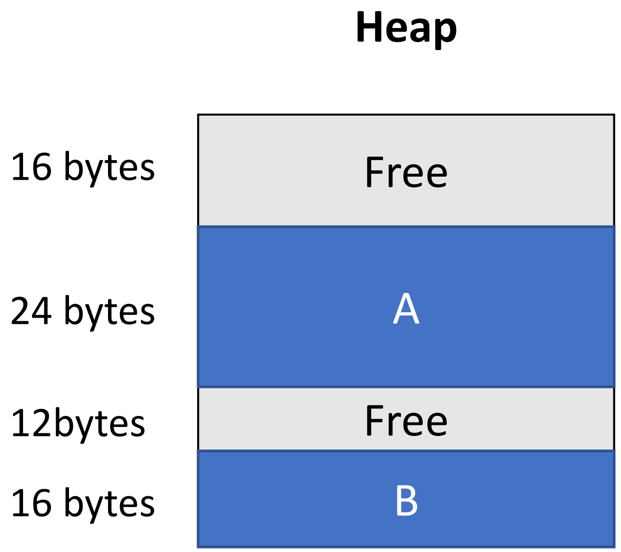
- Heap consists of allocated areas and free areas (holes)
- Order of allocation and free is unpredictable

\+ Works for all data structures

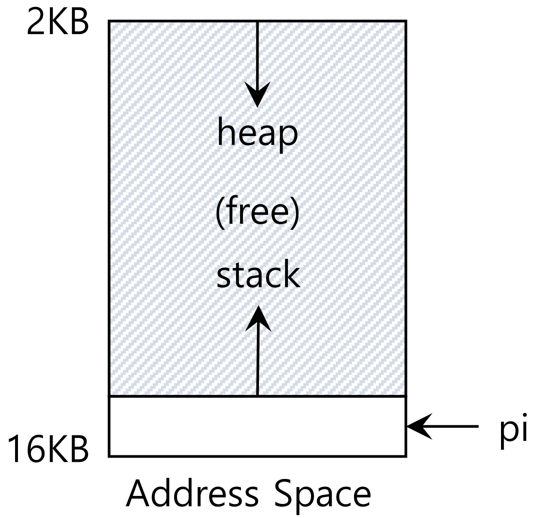
☹ Allocation can be slow

☹ Fragmentation

Programmers manage allocations/deallocations with library calls (**malloc/free**)

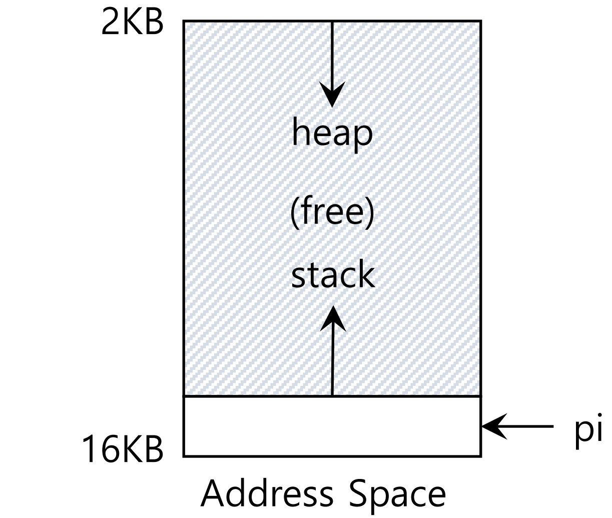


Memory allocation example

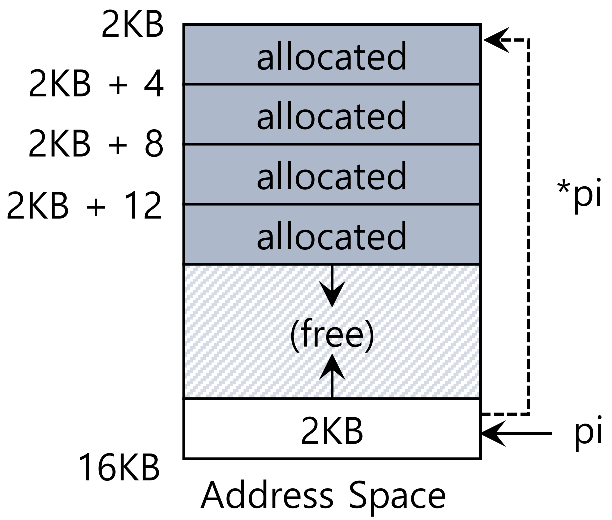



```
int *pi; // local variable
```

Memory allocation example




```
int *pi; // local variable
```




```
pi = (int *)malloc(sizeof(int)* 4);
```

Quiz: Match that Address Location

```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof int));
}
```

Possible segments: **static data, code, stack, heap**

What if no static data segment?

Address	Location
x	Static data → Code
main	Code
y	Stack
z	Stack
*z	Heap

What does a Process do?
(as far as a user is concerned)

What does a Process do?
(as far as a user is concerned)

- It can do anything

- Shell
- Compiler
- Editor
- Browser
- …
- These are all processes

Process Identification

- Each process has a unique process identifier
- Always referred to as “pid”

Basic Operations on Processes

- Create a process
- Terminate a process
  - Normal exit
  - Error
  - Terminated by another process

Linux Process Primitives

- pid = fork()
- exec(filename)
- exit()
- wait()

pid = fork()

- Creates an *identical* copy of parent
- In parent, returns pid of child
- In child, returns 0

exec(filename)

- Loads executable from file with filename

wait()

- Wait for one of its children to terminate

exit()

- Terminate the process

Typical fork()-ing Code Segment

```
if (pid = fork()) {
        wait()
}
else {
        exec(filename)
}
```

Before fork()

```
if (pid = fork()) {
      wait()
}
else {
      exec(filename)
}
```

After fork()

parent

```
if (pid = fork()) {
        wait()
}
else {
        exec(filename)
}
```

child

```
if (pid = fork()) {
        wait()
}
else {
        exec(filename)
}
```

After fork()

parent

```
if (pid_child) {
        wait()
}
else {
        exec(filename)
}
```

child

```
if (0) {
        wait()
}
else {
        exec(filename)
}
```

After fork()

parent

```
if (pid_child) {
        wait()
}
else {
        exec(filename)
}
```

child

```
if (0) {
        wait()
}
else {
        exec(filename)
}
```

After exec()

parent

```
if (pid_child) {
        wait()
}
else {
        exec(filename)
}
```

child

```
main () {

    …

    exit()
}
```

After exit()

parent

```
if (pid_child) {
        wait() //wait returns
}
else {
        exec(filename)
}
```

child

```
main () {

   …

   exit()
}
```

Outline of Linux Shell

```
forever {
  read from input

  if( logout) exit()

  if ( pid = fork() ) {
      wait()
  }

  else {
      exec( filename )
  }
}
```

Shell Operation

- New command line (!= logout)
  - Shell forks a new process and waits
  - Child executes program on command line

The Linux Process Tree

Boot

- First process after boot is the init process
- Happens by black magic

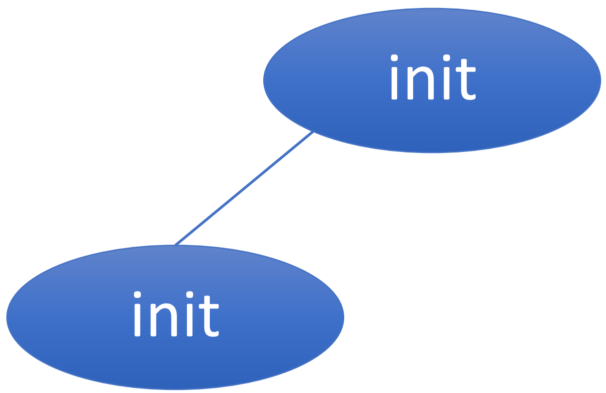
User logs in

User logs in

- Init forks and waits
- Child execs shell

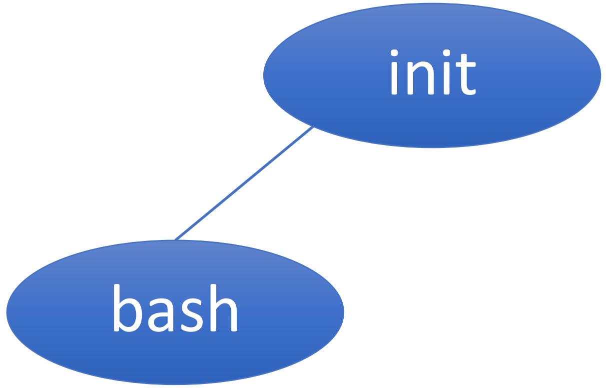
User logs in

- Init forks and waits
- Child execs shell

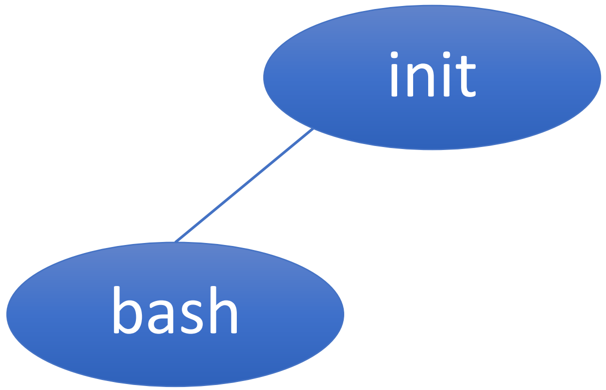


User logs in

- Init forks and waits
- Child execs shell

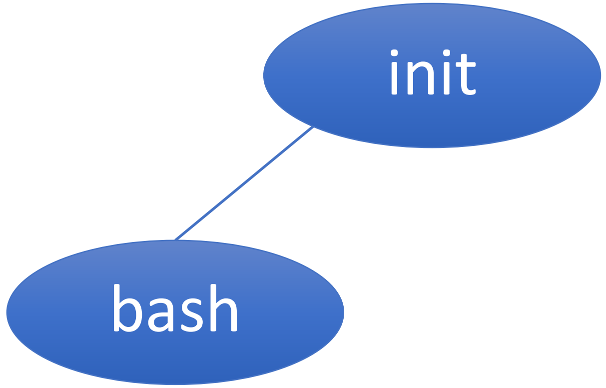


User runs make



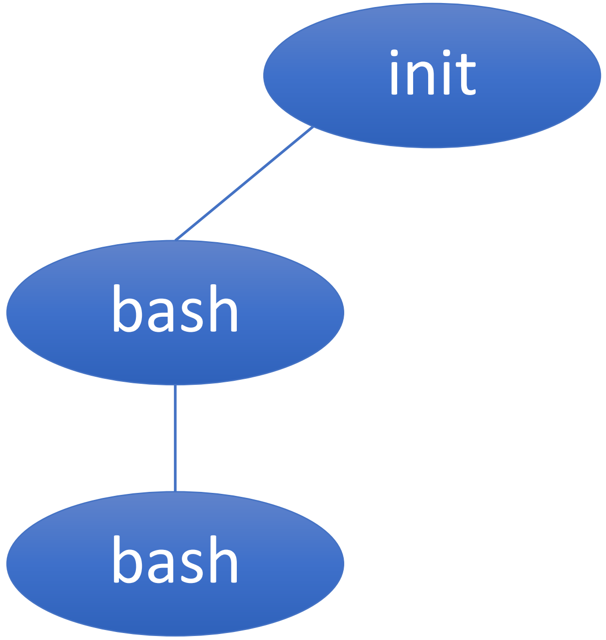
User runs make

- Shell forks and waits
- Child execs make



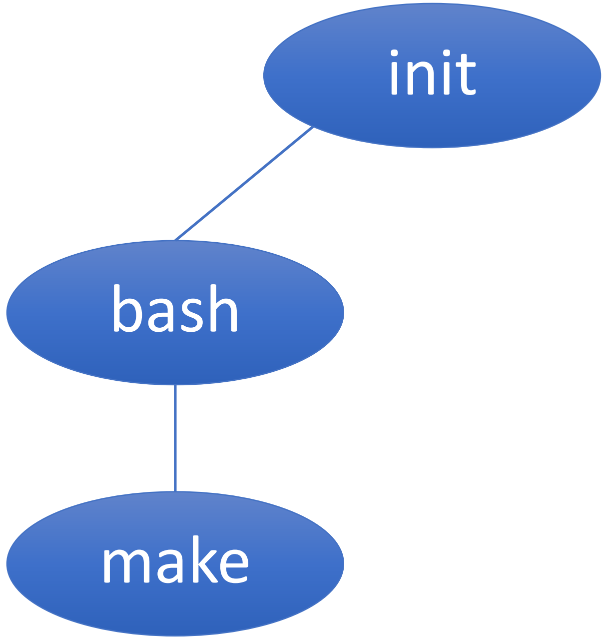
User runs make

- Shell forks and waits
- Child execs make

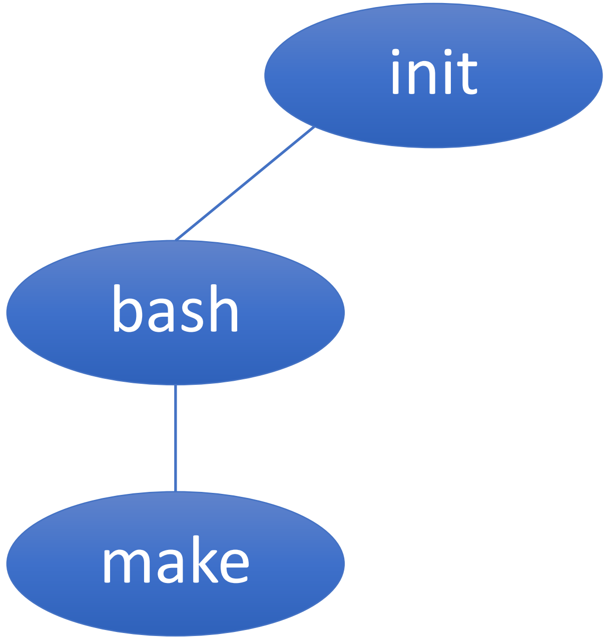


User runs make

- Shell forks and waits
- Child execs make

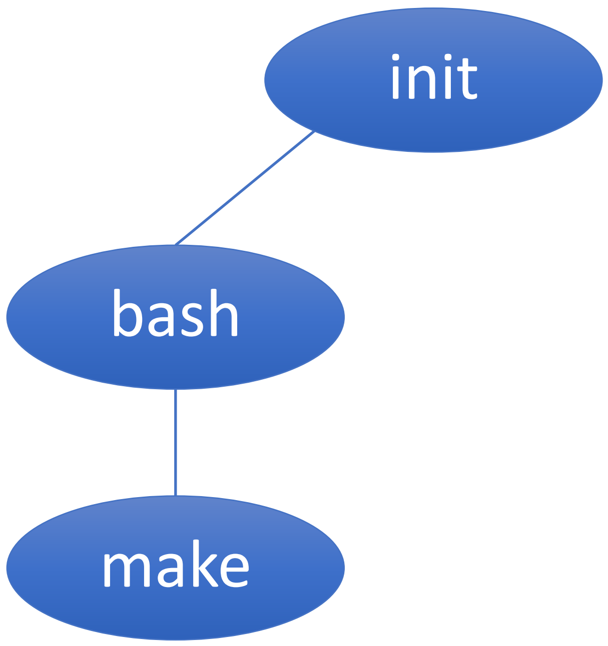


Another user logs in



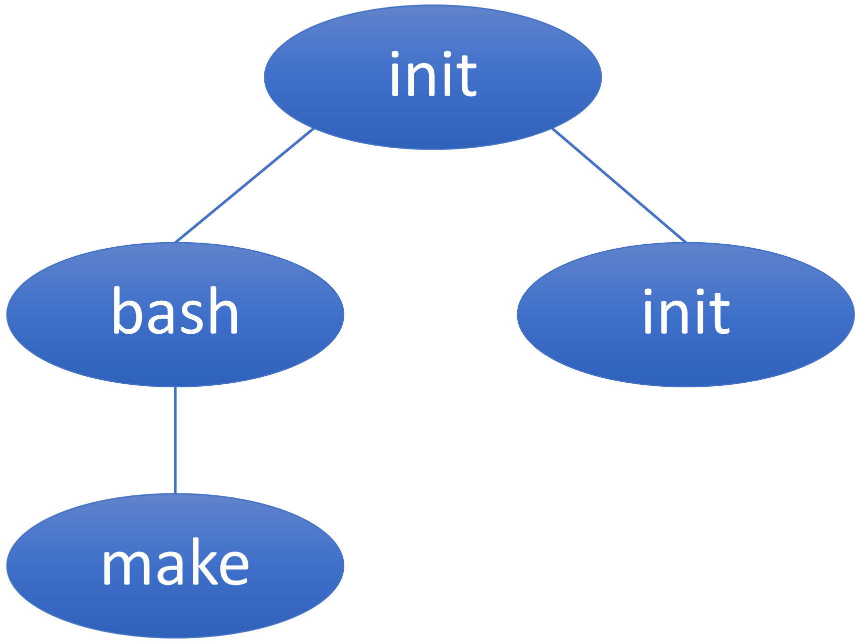
Another user logs in

- Init forks and waits
- Child execs shell



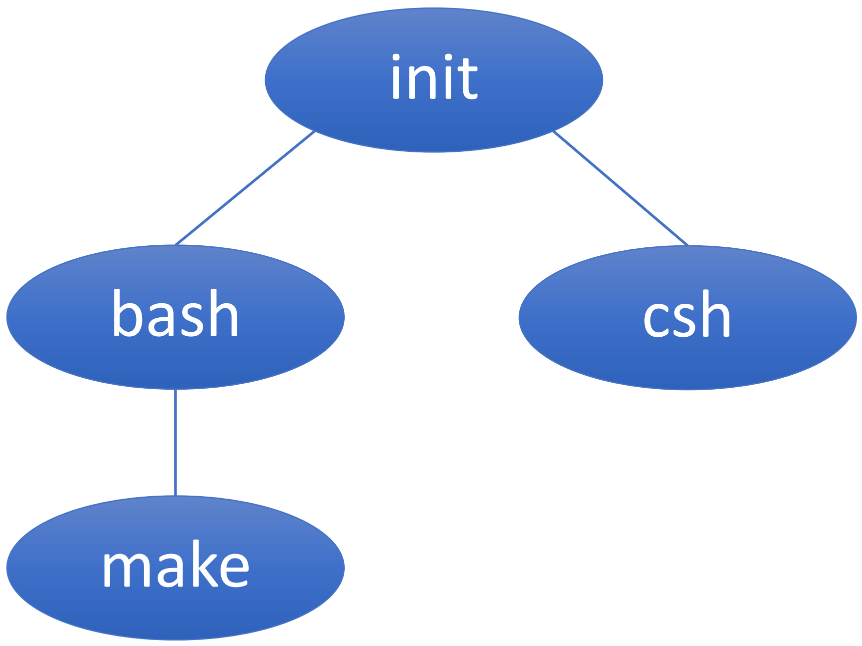
Another user logs in

- Init forks and waits
- Child execs shell

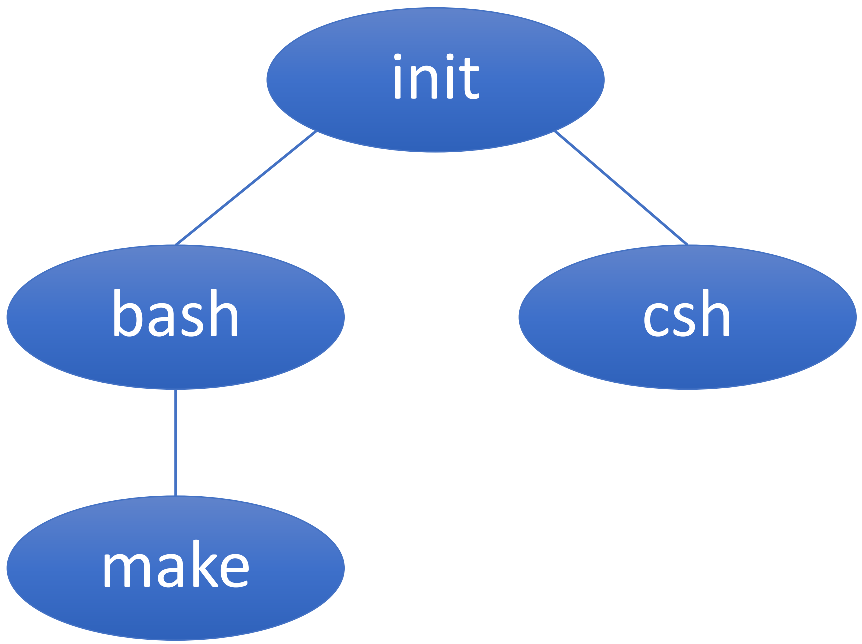


Another user logs in

- Init forks and waits
- Child execs shell

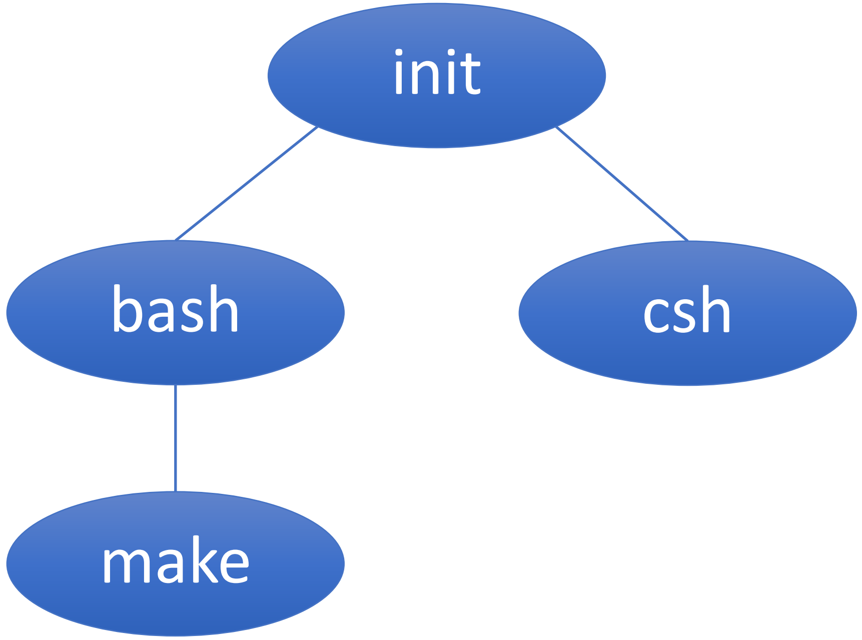


Make runs gcc



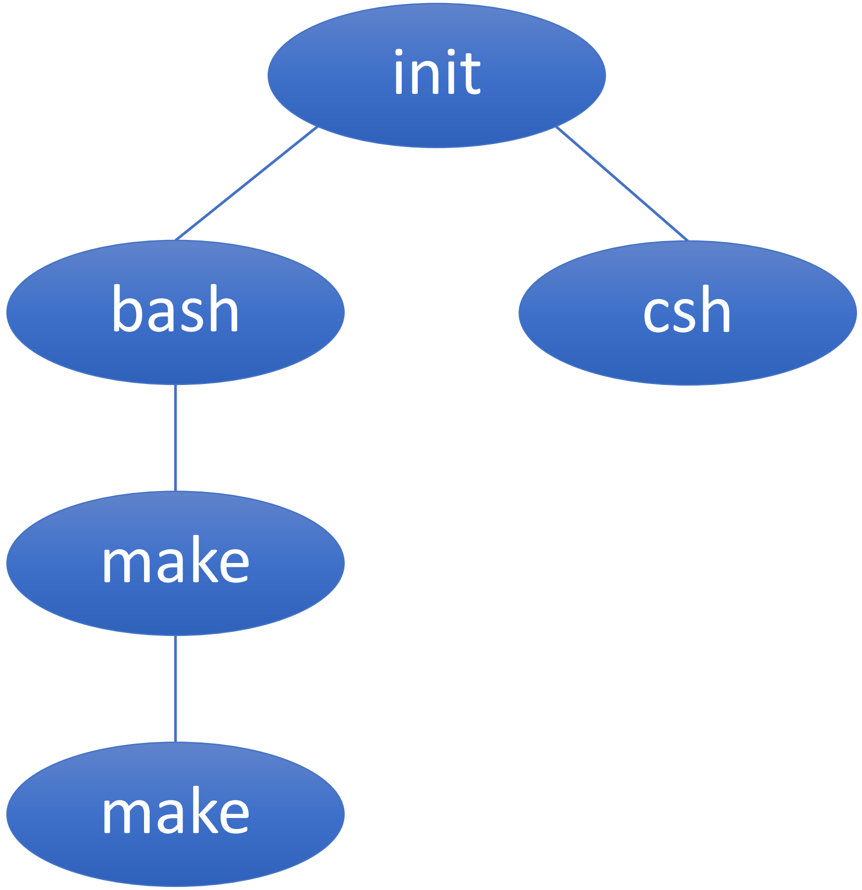
Make runs gcc

- Make forks and waits
- Child execs gcc



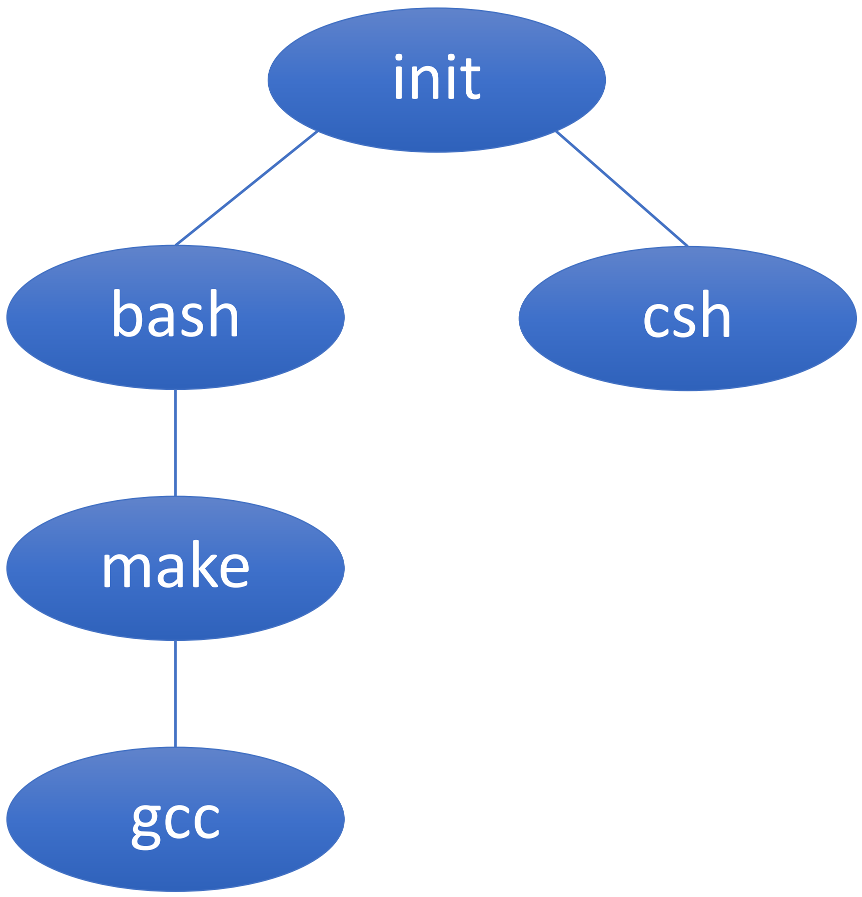
Make runs gcc

- Make forks and waits
- Child execs gcc



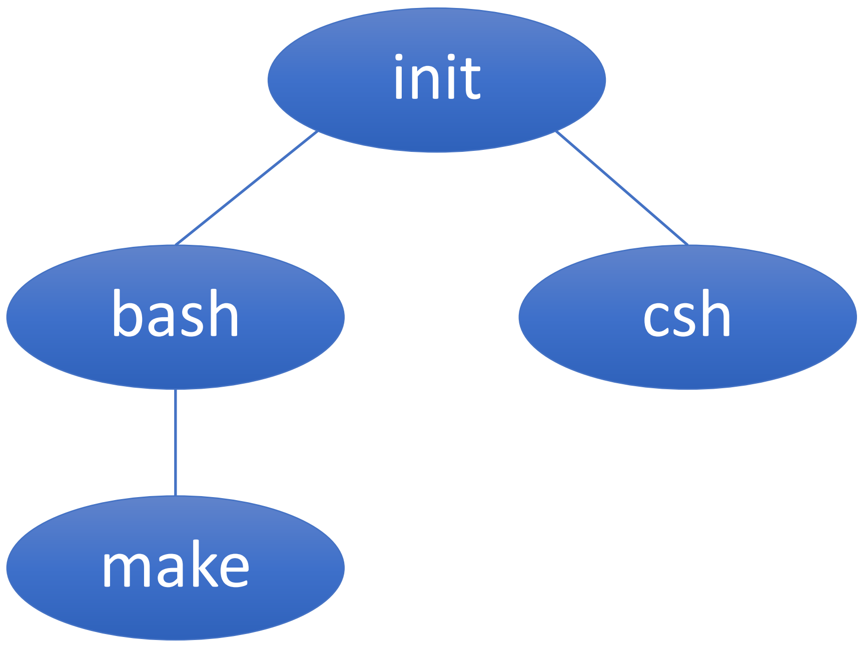
Make runs gcc

- Make forks and waits
- Child execs gcc



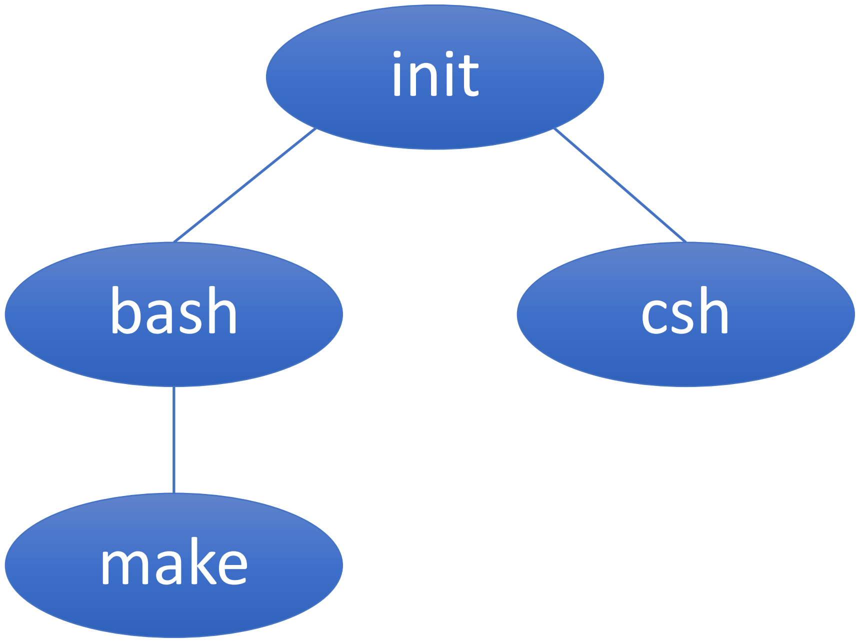
Gcc finishes

- Gcc exits
- Make returns from wait



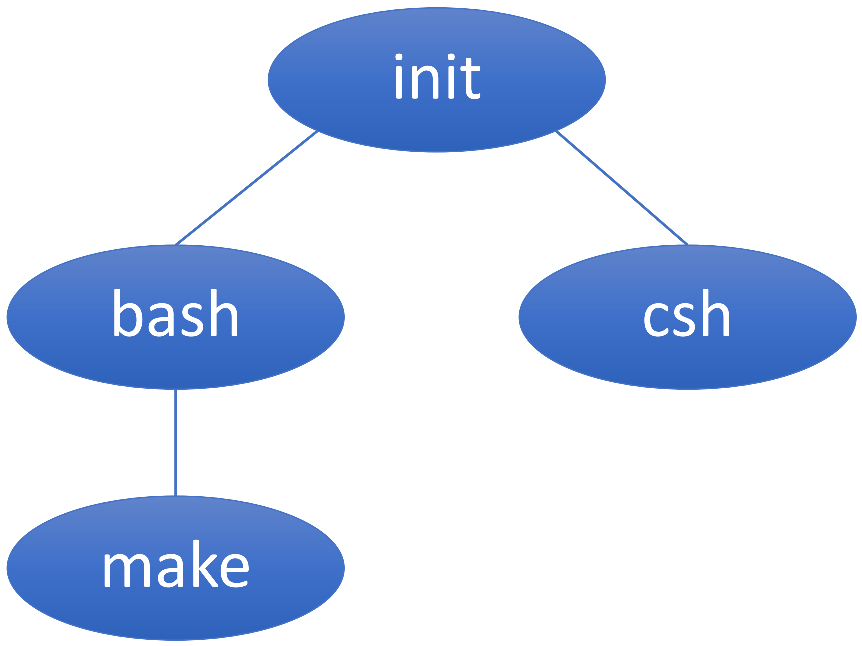
Gcc finishes

- Gcc exits
- Make returns from wait

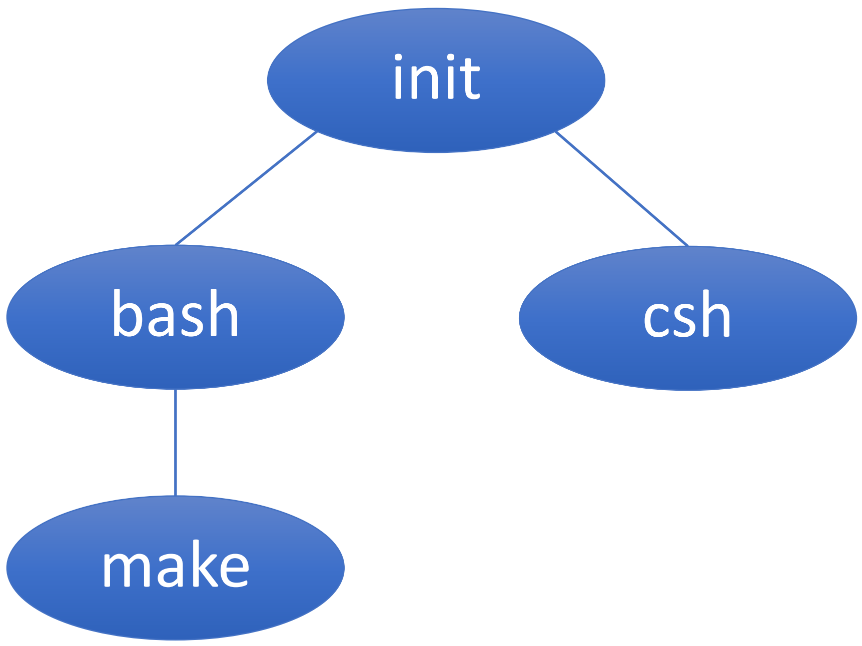


Gcc finishes

- Gcc exits
- Make returns from wait

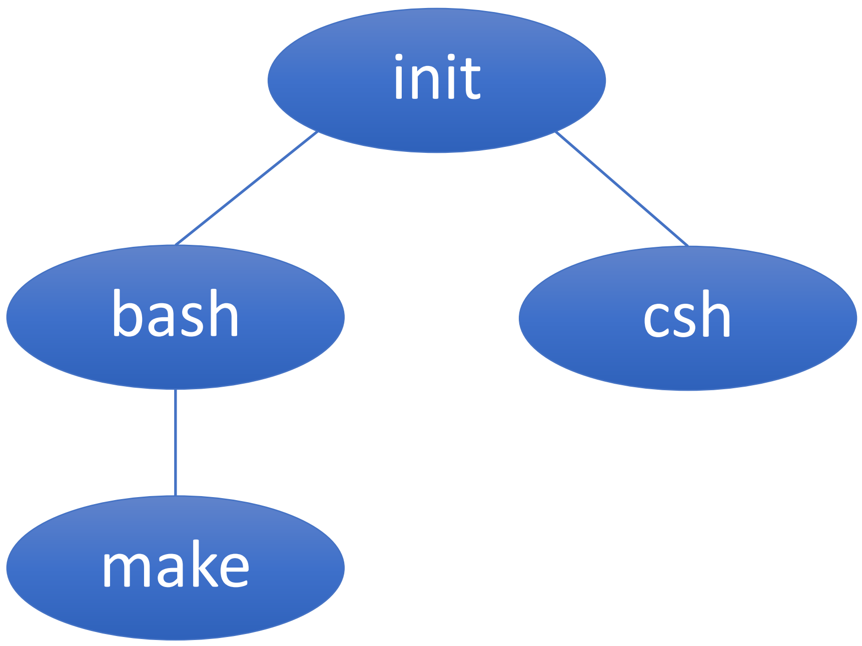


Second user logs out



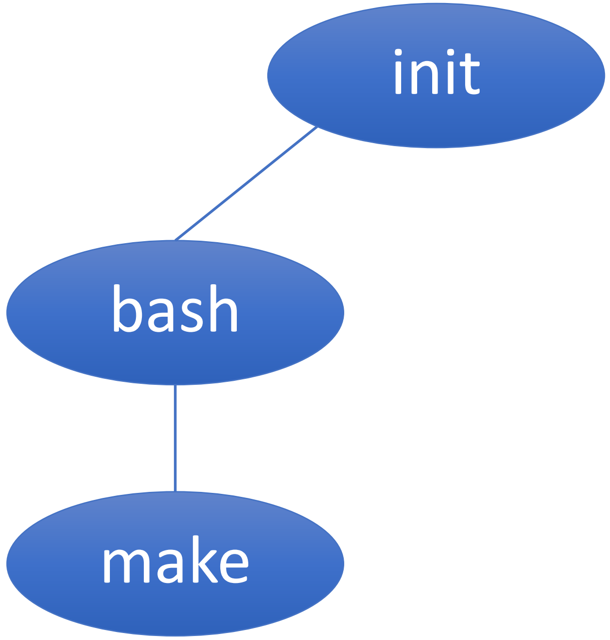
Second user logs out

- Csh exits
- Init returns from wait



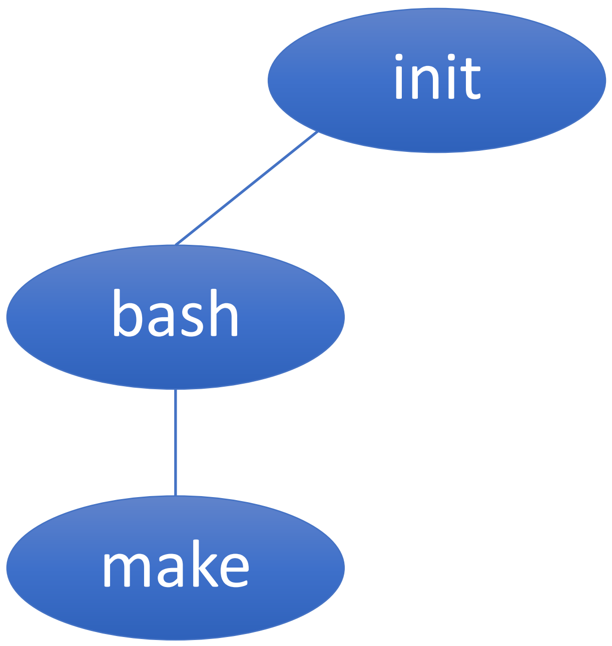
Second user logs out

- Csh exits
- Init returns from wait



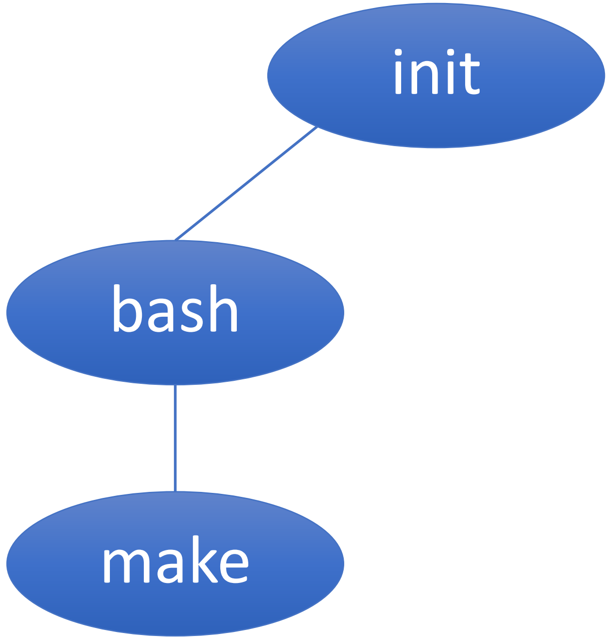
Second user logs out

- Csh exits
- Init returns from wait



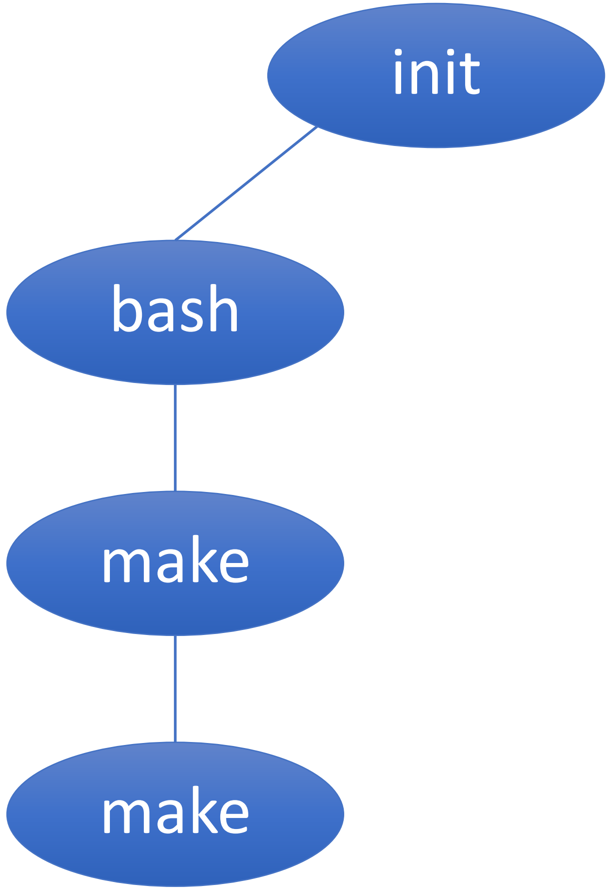
Make runs cp

- Make forks and waits
- Child execs cp



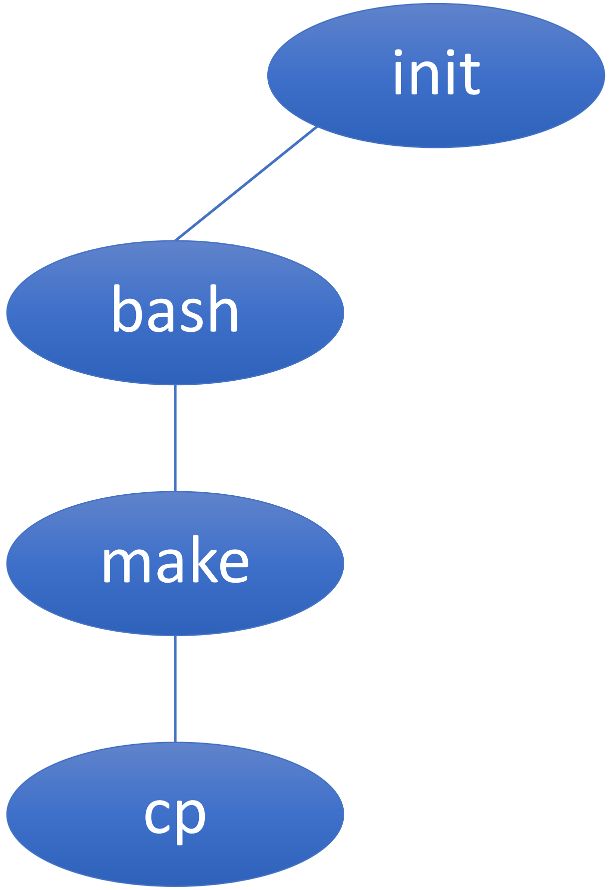
Make runs cp

- Make
  - Make forks and waits
  - Child execs cp



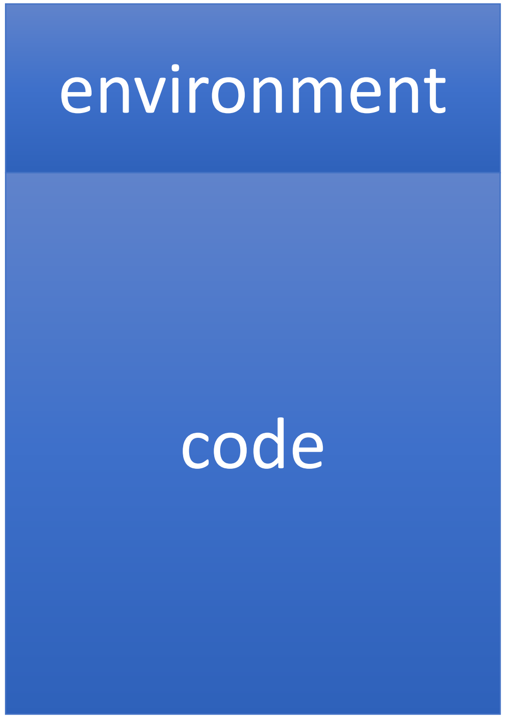
Make runs cp

- Make
  - Make forks and waits
  - Child execs cp



Why fork+exec vs. create?

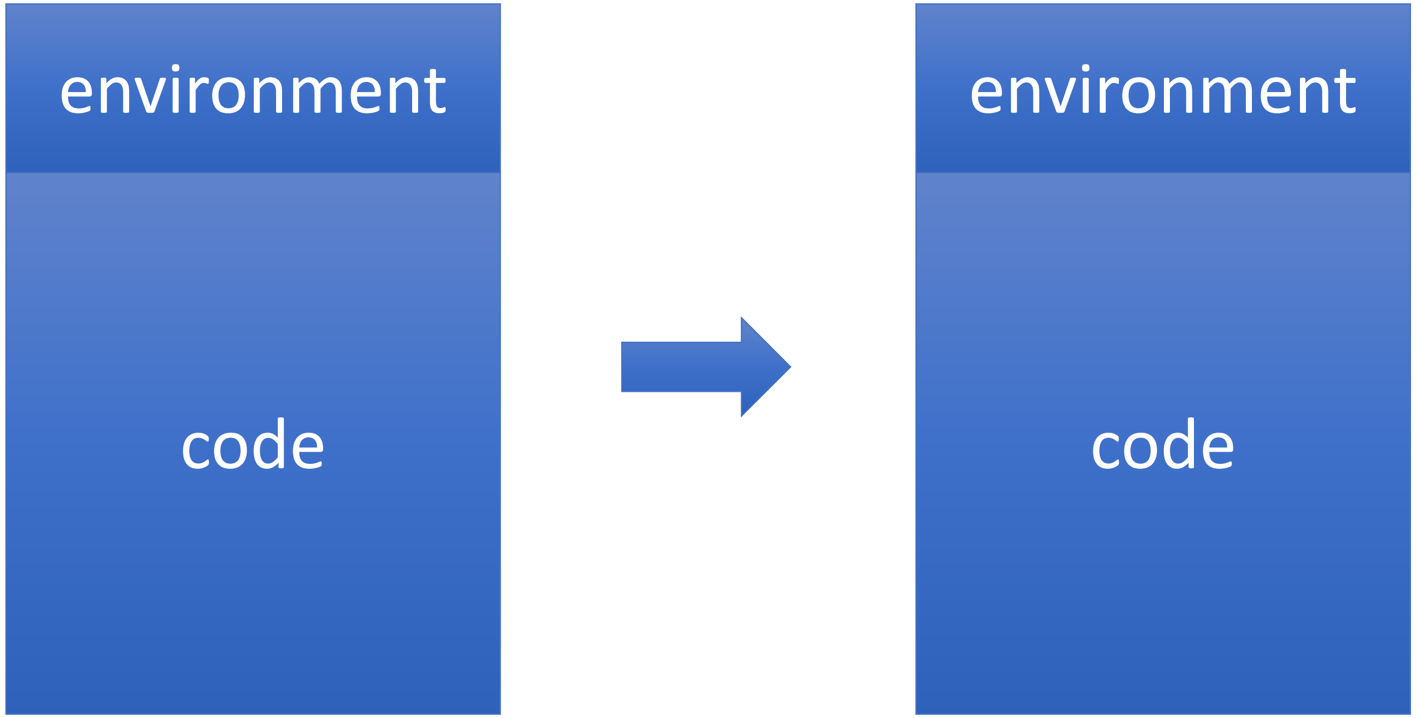
Process = Environment + Code



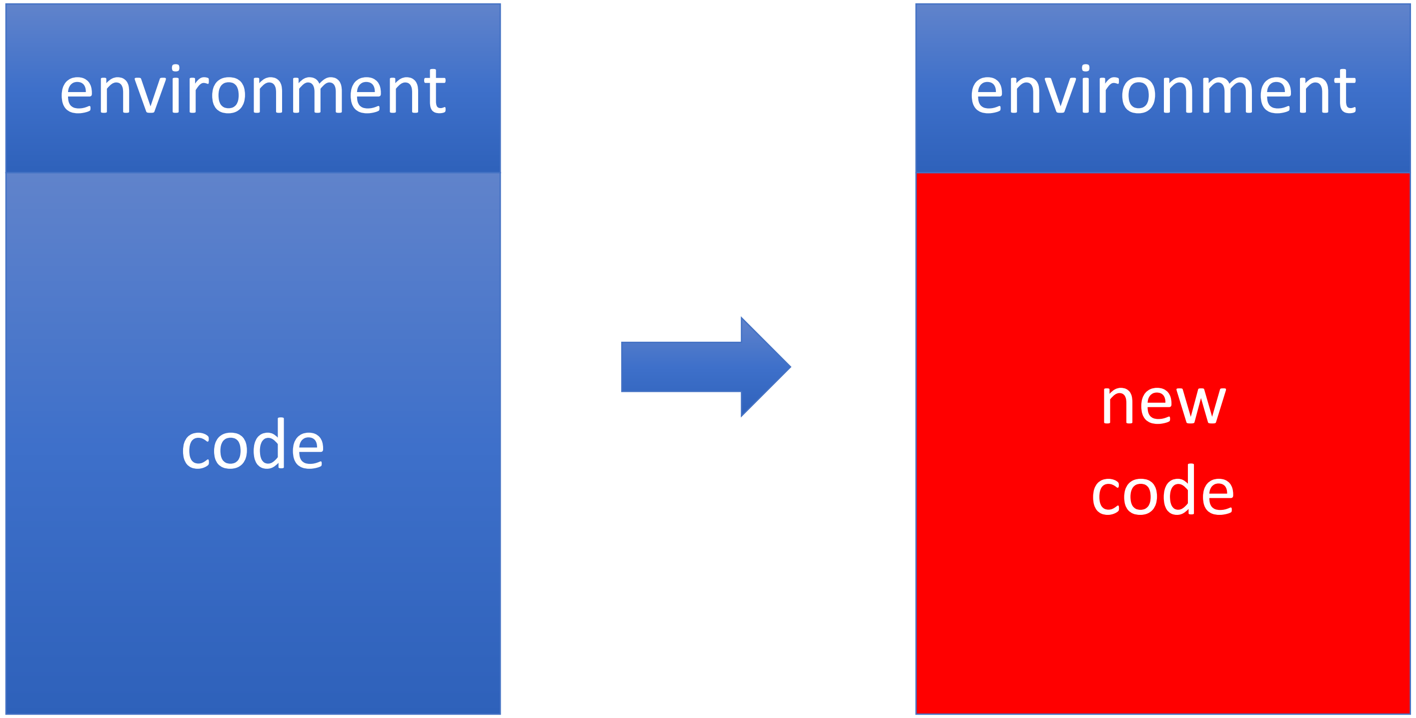
Process = Environment + Code

- Environment includes:
  - Ownership
  - Open files
  - Values of environment variables

After a fork()



After an exec() in the Child



Advantage

- Child automatically inherits environment

Given New Definition of exec

```
forever {
  read from input

  if( logout) exit()

  if ( pid = fork() ) {
      wait()
  }

  else {

      exec( filename )
  }
}
```

does it make sense to write code here?

Answer

- Yes
- Shell can manipulate environment of child
- For instance, can manipulate stdin and stdout

Let’s practice!

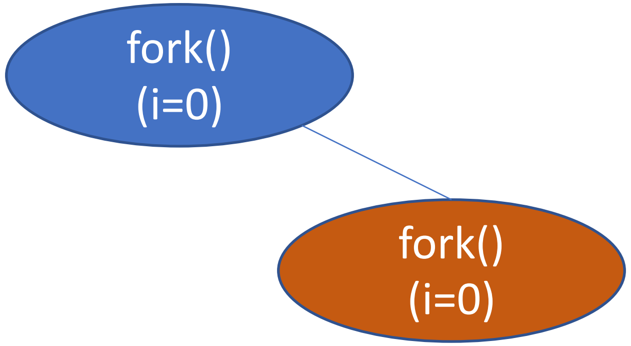
How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

```
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```

Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

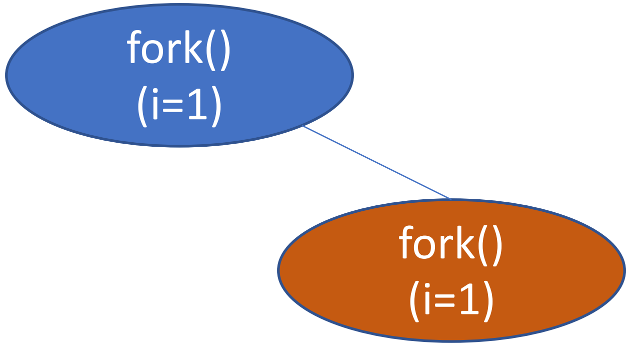
```c
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

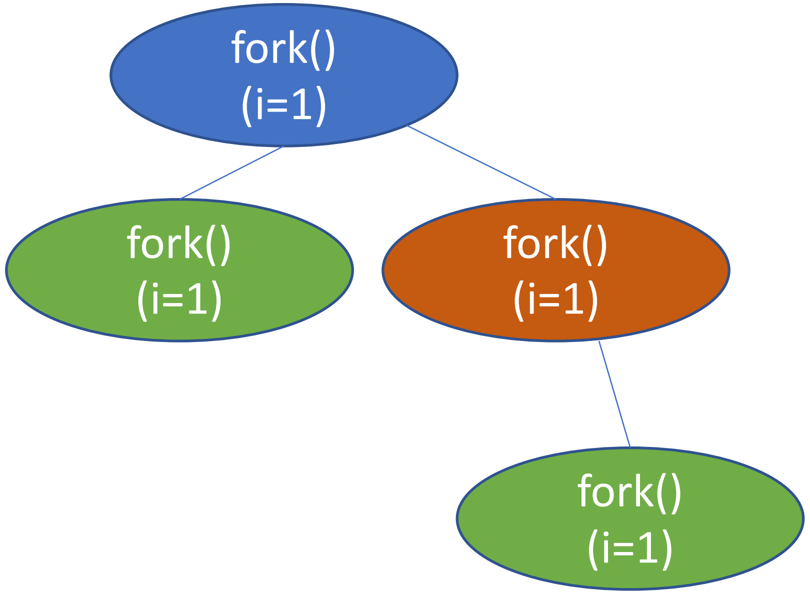
```
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

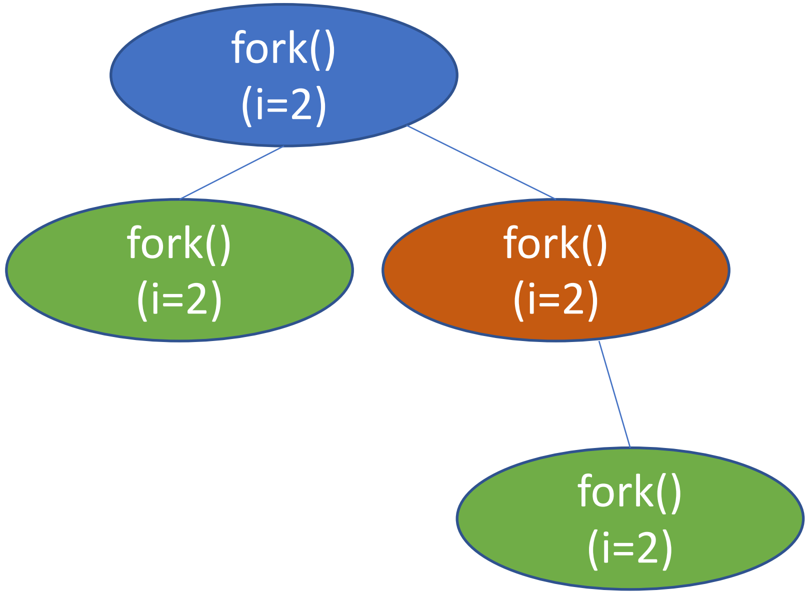
```c
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

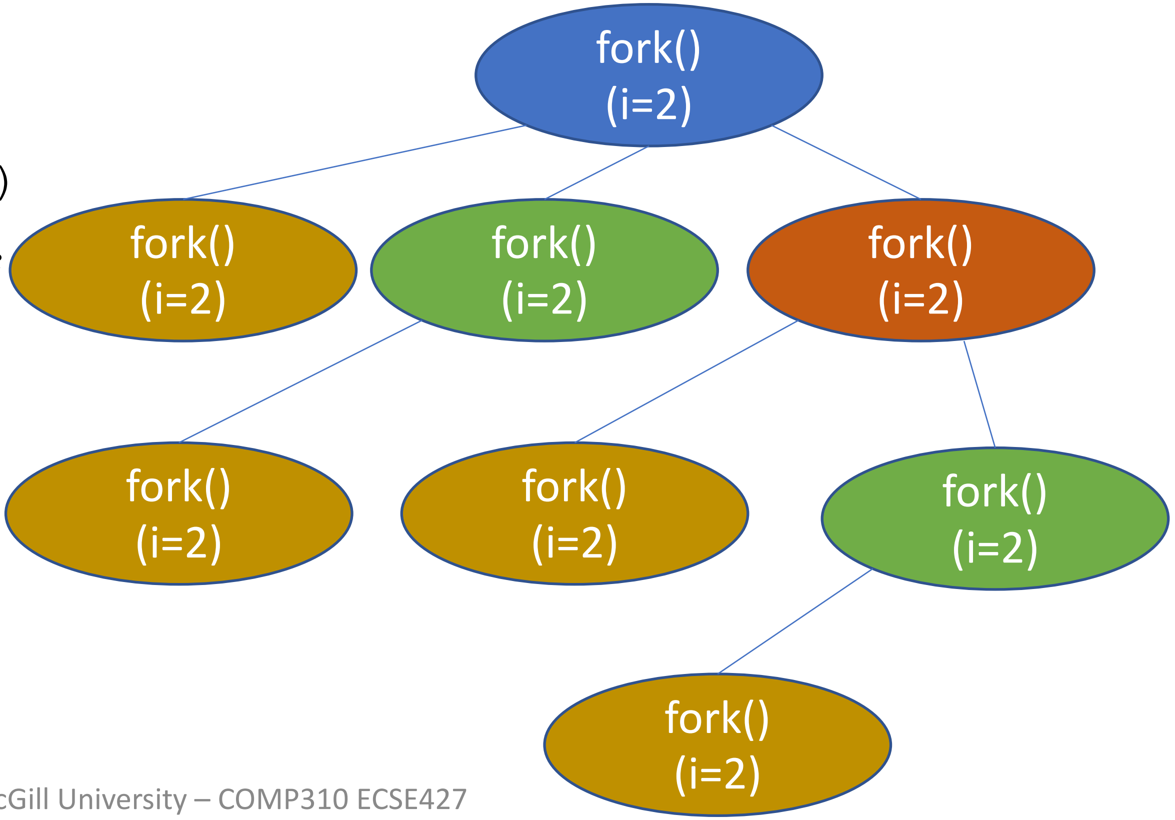
```c
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```



7 new processes created

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

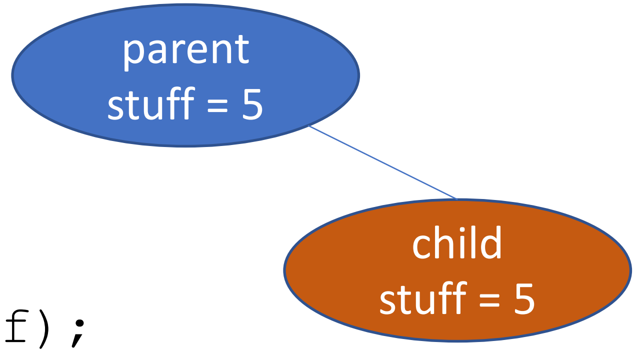
```
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```c
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```



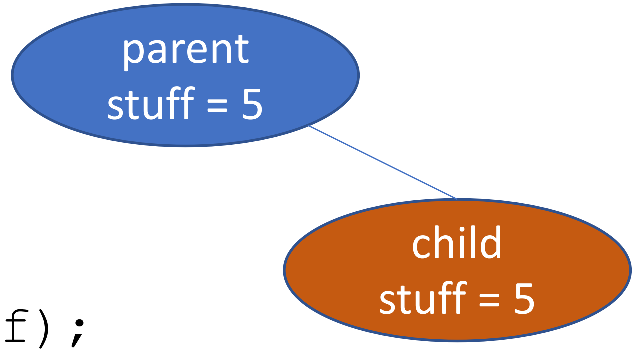
In this case, parent does not wait
→ Child and parent can be executed in any order

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```c
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```



In this case, parent does not wait
→ Child and parent can be executed in any order

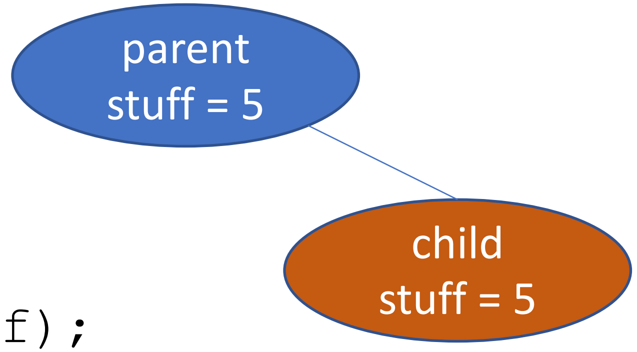
Does the order matter?

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```



Assume parent goes first:

Program prints:

The last digit of pi is 5.

The last digit of pi is 5.

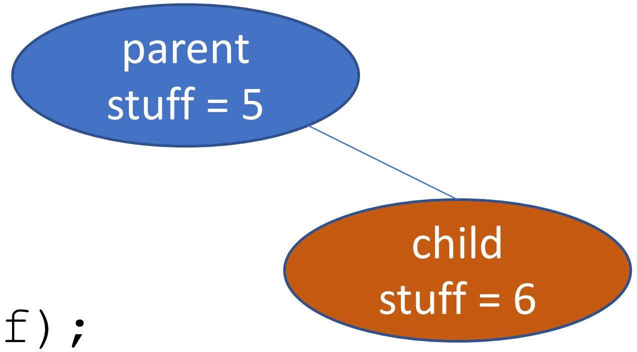
Why?

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
1 int main(void) {
2   int stuff = 5;
3   pid_t fork_ret = fork();
4   printf("The last digit of pi is %d\n", stuff);
5   if (fork_ret == 0)
6     stuff = 6;
7   return 0;
8 }
```



Assume child goes first:

Line 6 executes.

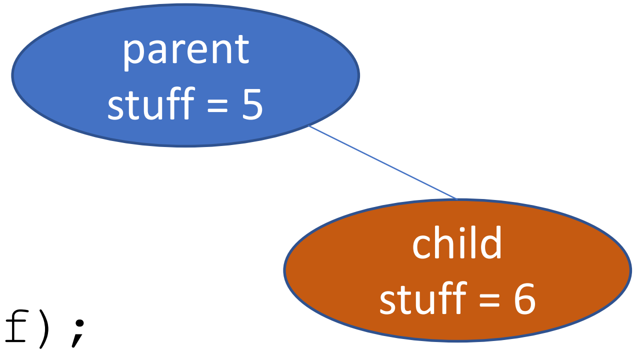
Does this affect the parent's `stuff` value?

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
1 int main(void) {
2   int stuff = 5;
3   pid_t fork_ret = fork();
4   printf("The last digit of pi is %d\n", stuff);
5   if (fork_ret == 0)
6     stuff = 6;
7   return 0;
8 }
```



Assume child goes first:

Line 6 executes.

Does this affect the parent's `stuff` value?

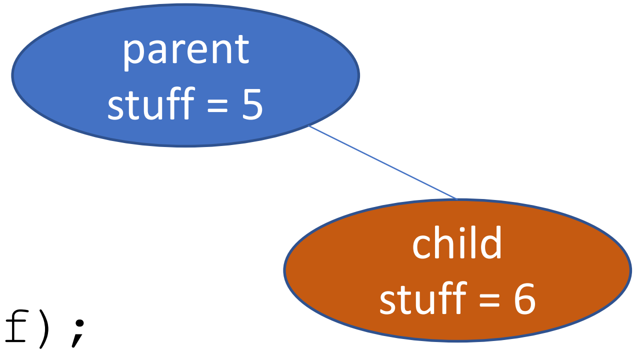
No. Child's environment is cloned.

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
1 int main(void) {
2   int stuff = 5;
3   pid_t fork_ret = fork();
4   printf("The last digit of pi is %d\n", stuff);
5   if (fork_ret == 0)
6     stuff = 6;
7   return 0;
8 }
```



Program prints:
The last digit of pi is 5.
The last digit of pi is 5.

Assume child goes first:
Line 6 executes.
Does this affect the parent's `stuff` value?
No. Child's environment is cloned.

What does a process do?
(as far as a user is concerned)

- It can do anything

- Shell
- Compiler
- Editor
- Browser
- …

- These are all processes

What does a process do?
(as far as the OS is concerned)

What does a process do?
(as far as the OS is concerned)

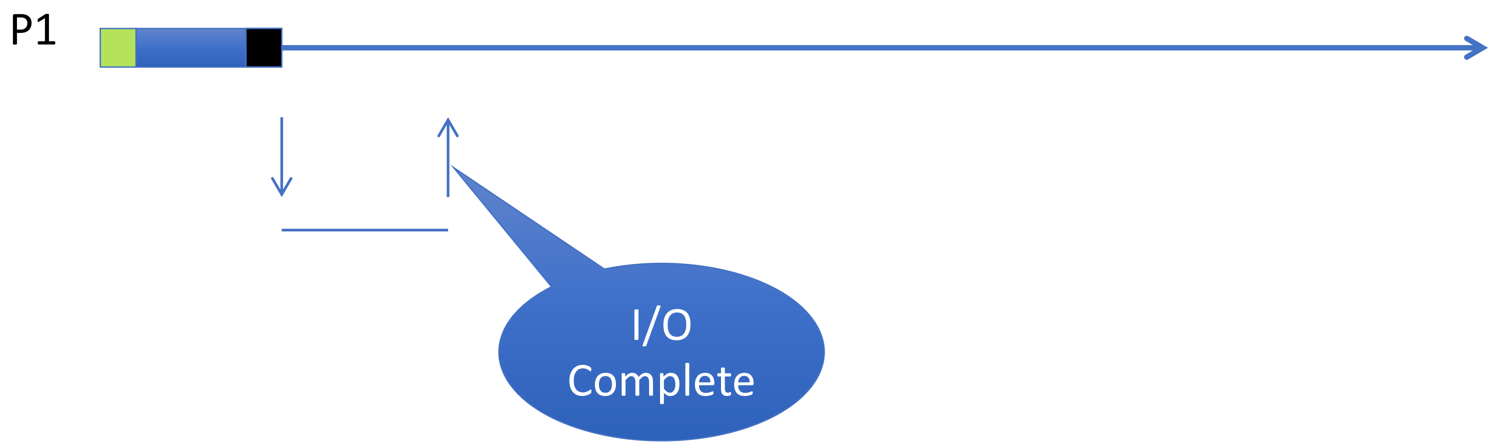
- Either it computes (uses the CPU)
- Or it does I/O (uses a device)

Single Process System

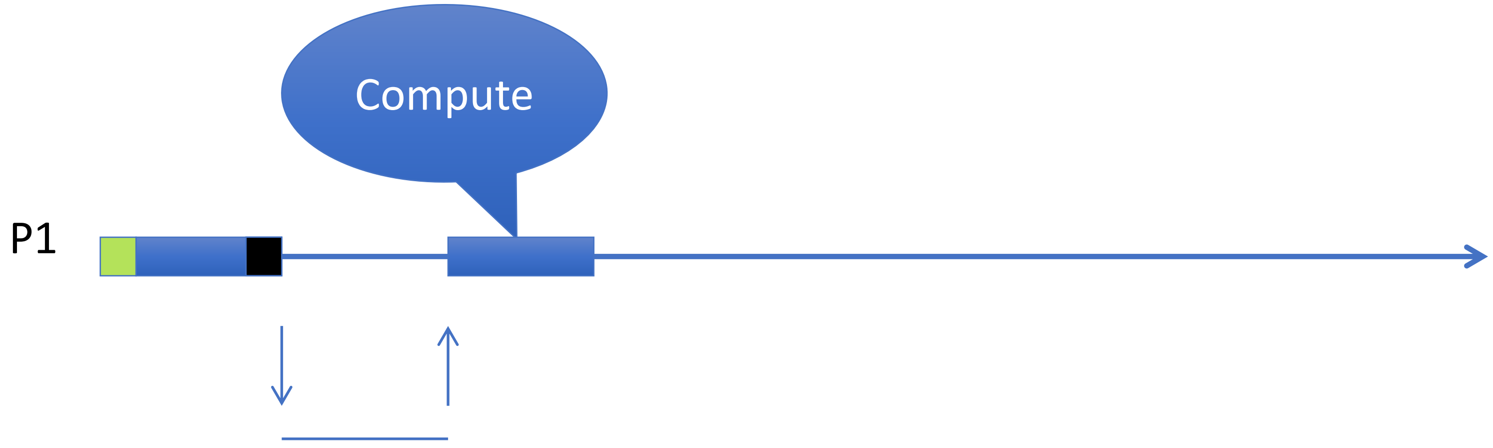
Single Process System

Single Process System

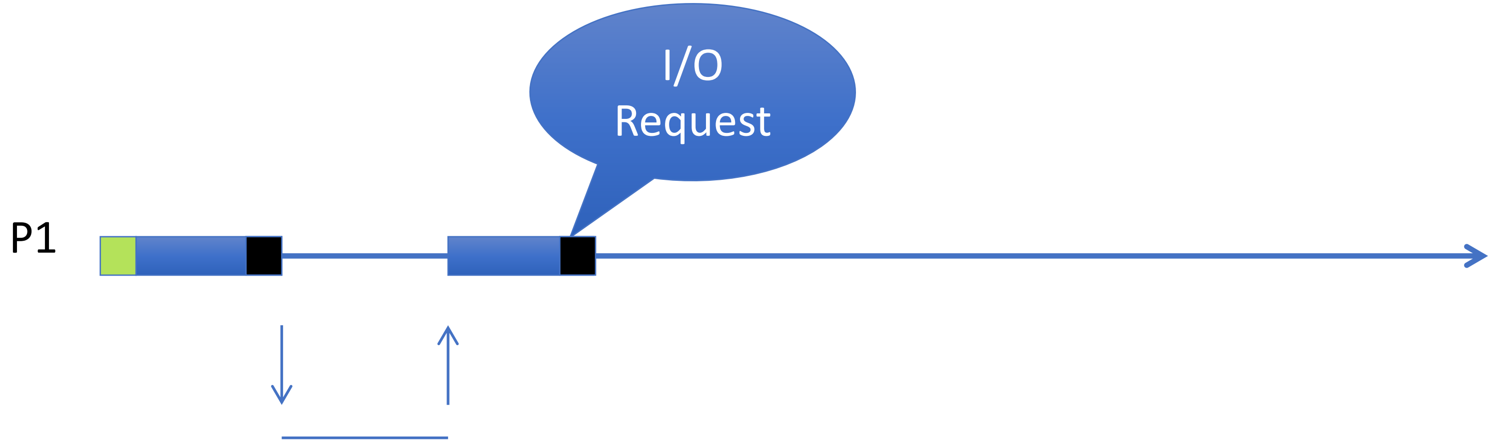
Single Process System



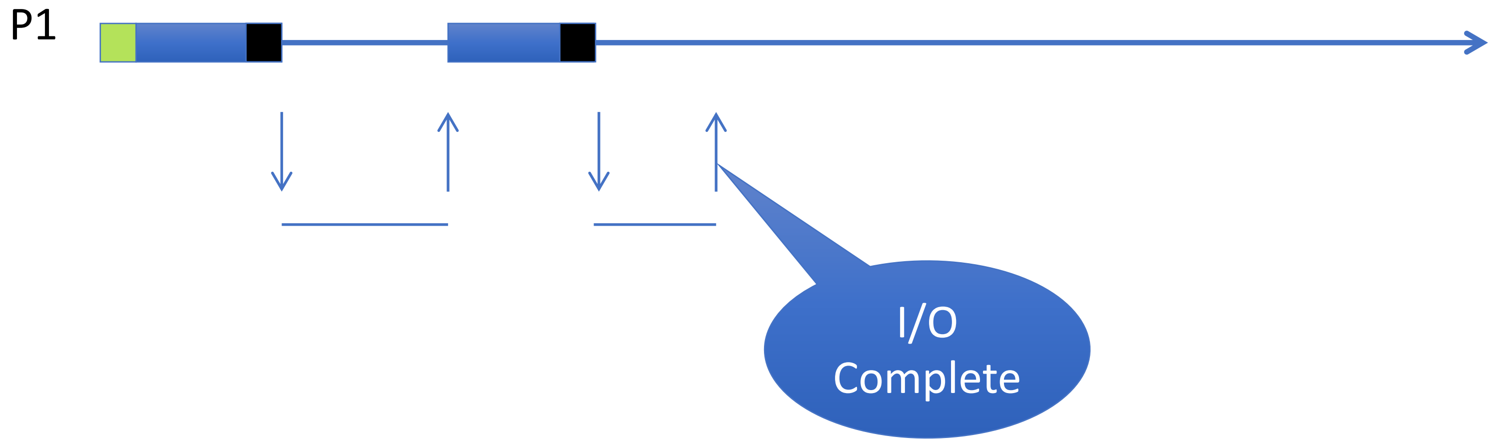
Single Process System



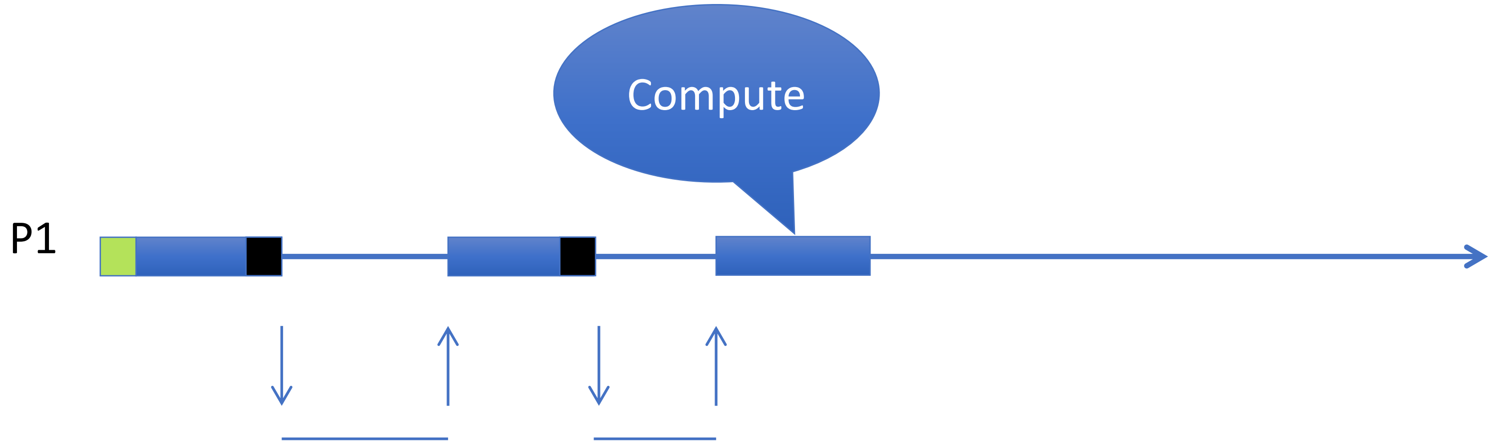
Single Process System



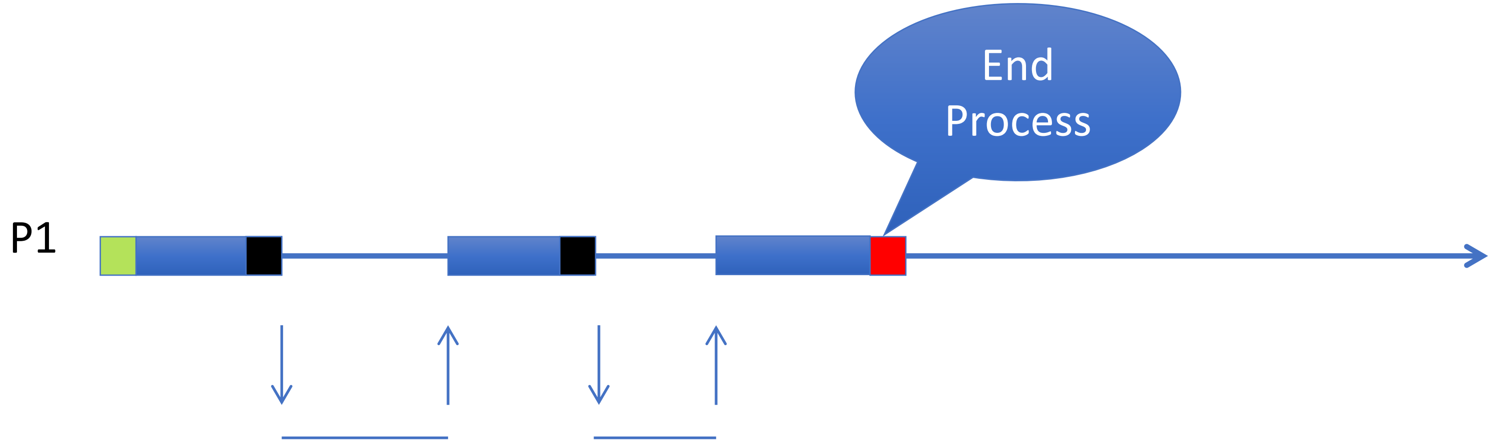
Single Process System



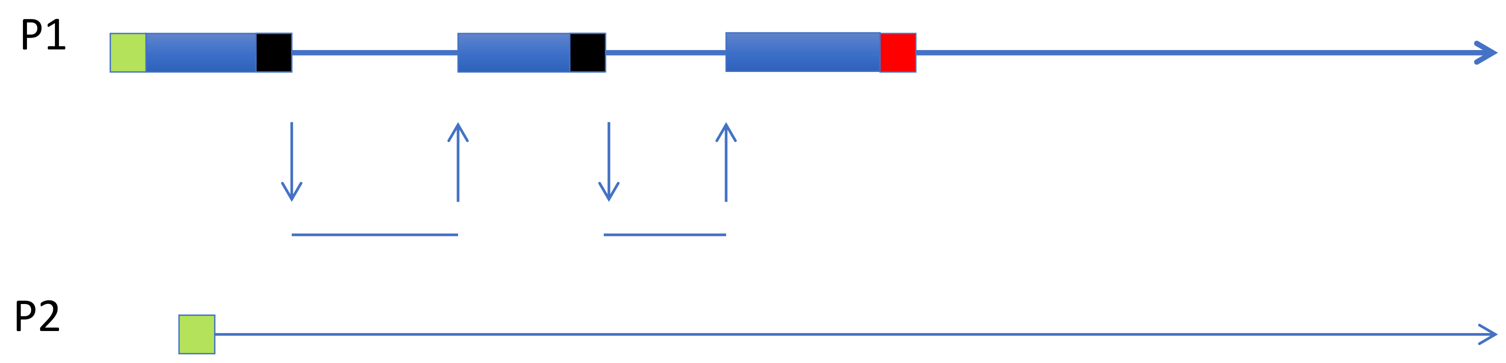
Single Process System



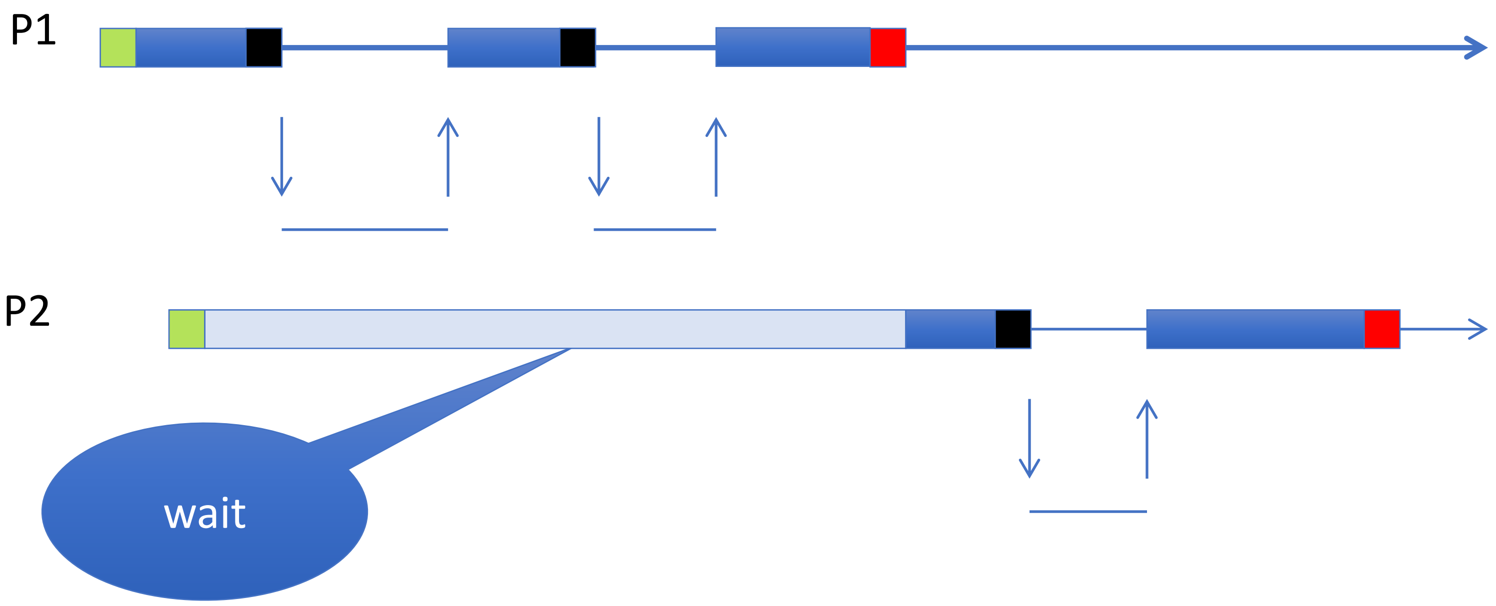
Single Process System



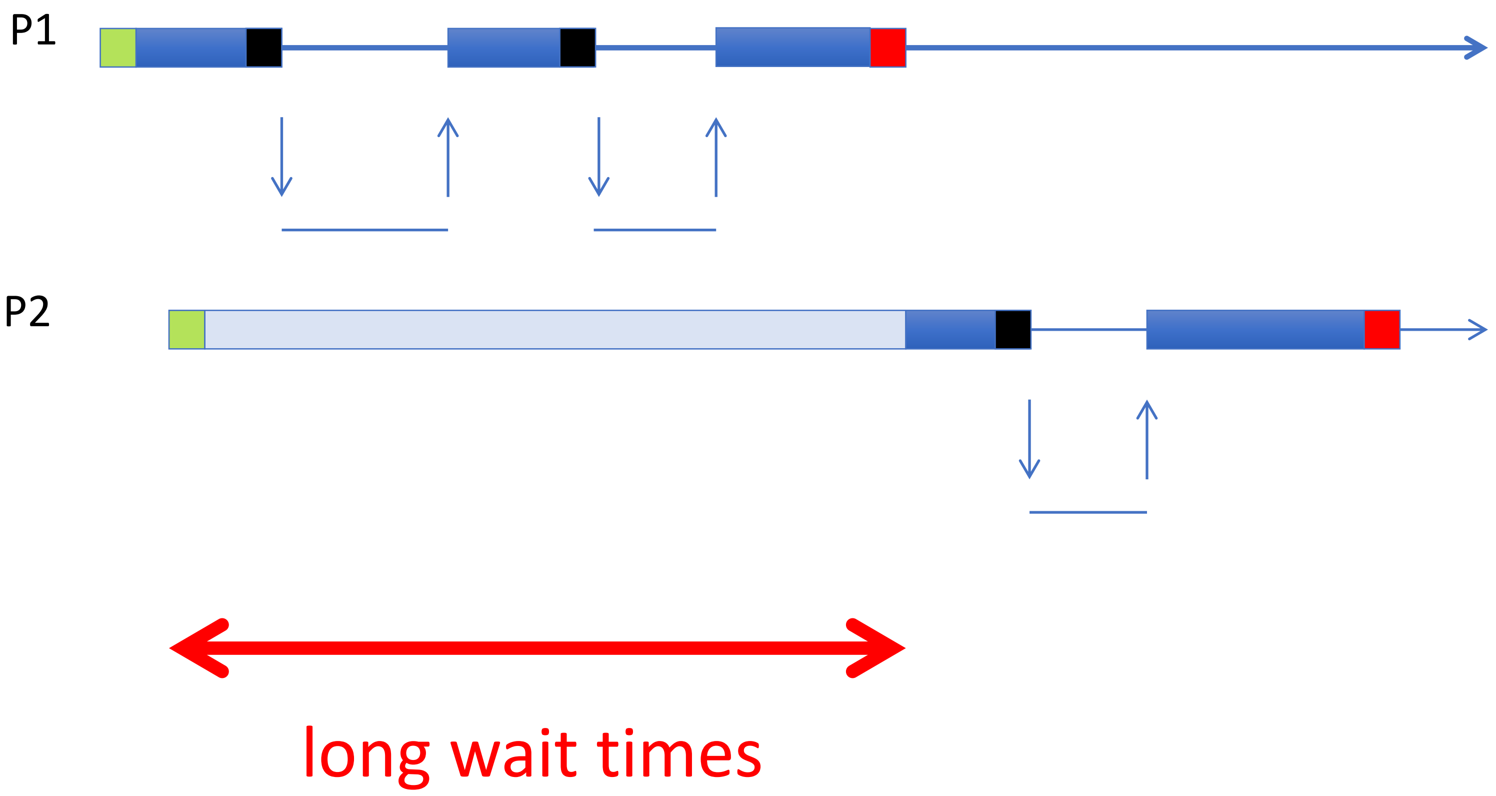
A Second Process



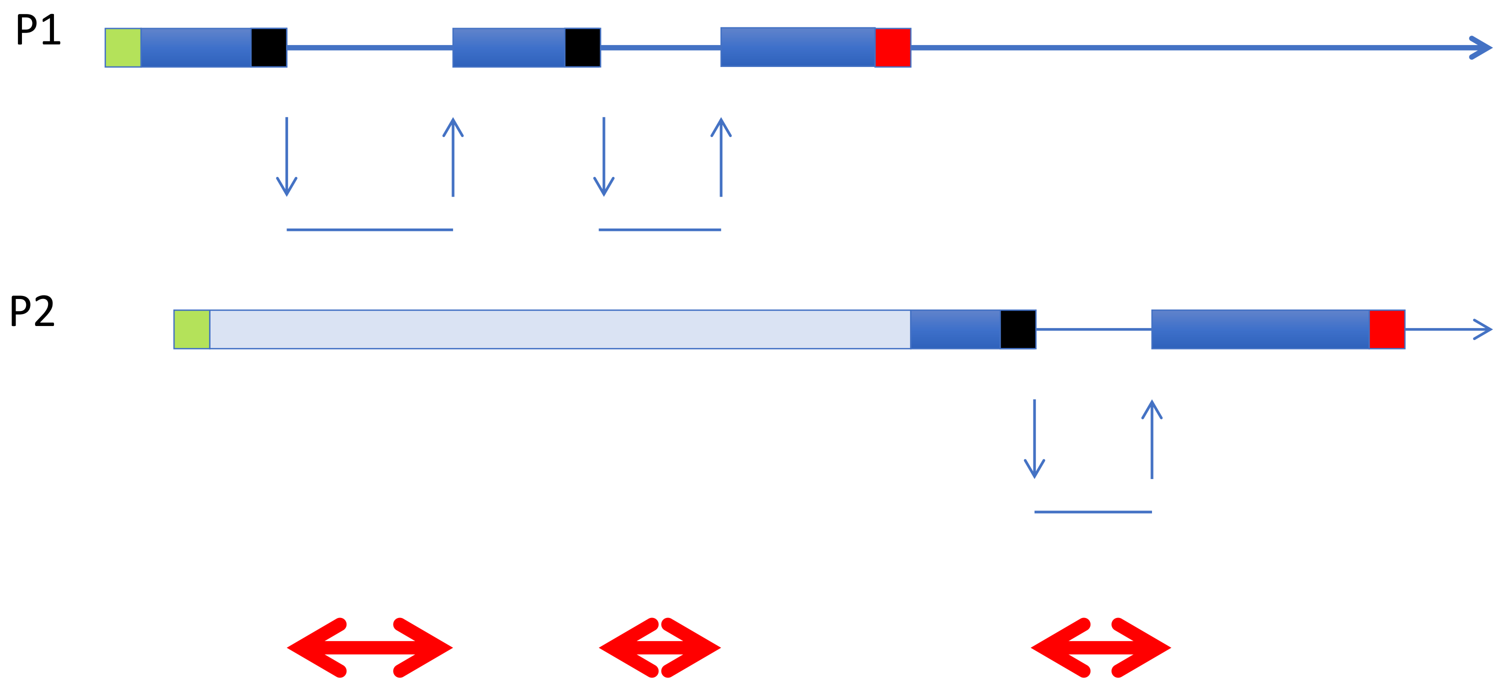
A Second Process



Two Issues



Two Issues



low utilization (long CPU idle times)

Single Process System

- Is very inefficient
  - Very poor CPU utilization
- Is very annoying
  - You can't do anything else

Further Optional Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 3 – 7 (inclusive) https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University), Prof. Natacha Crooks (UC Berkeley).