CSCI 2410 Systems Programming

Fall 2026

Instructor: Jie Zhou

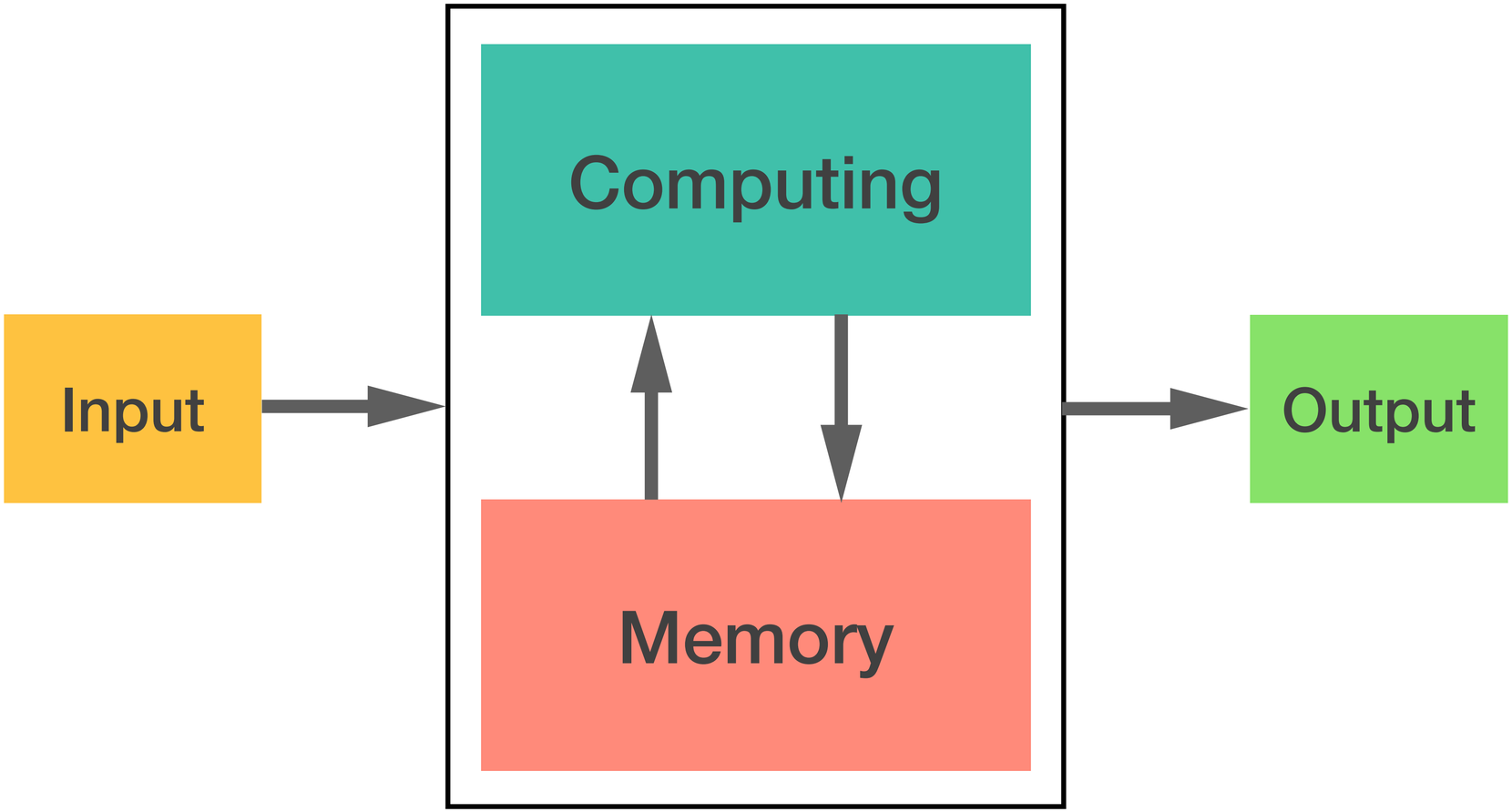
Department of Computer Science

George Washington University



Slides materials are mostly credited to Gabriel Parmer and Sibin Mohan at GWU.

Architecture of Modern Computers



In the Beginning Days

- Batch Processing
  - Single User, Single Process, Single Machine
- Submitted your code to the person who ran your program on the machine
- Problem: Lots of idle time
  - I/O waits
  - Human/operational delays
  - Debug/reseting cycles



Time Sharing System

- Each program has a “share” of the CPU time.
- The program has the impression of the *only* program.
  - Each process is sharing the computer resources.
  - Swapping in and out when share is up or waiting on other resources
  - Appearance of concurrently running process

Tackling Complexity

- Modern computers run *many* programs simultaneously.
- A program may need to communicate with another one.
- But they should *not* interfere with each other, accidentally or maliciously.
- **Problem:** How to maintain isolation between programs?

Isolation



Isolate two components from each other

- One component cannot access data/code of the other component except through a well-defined API



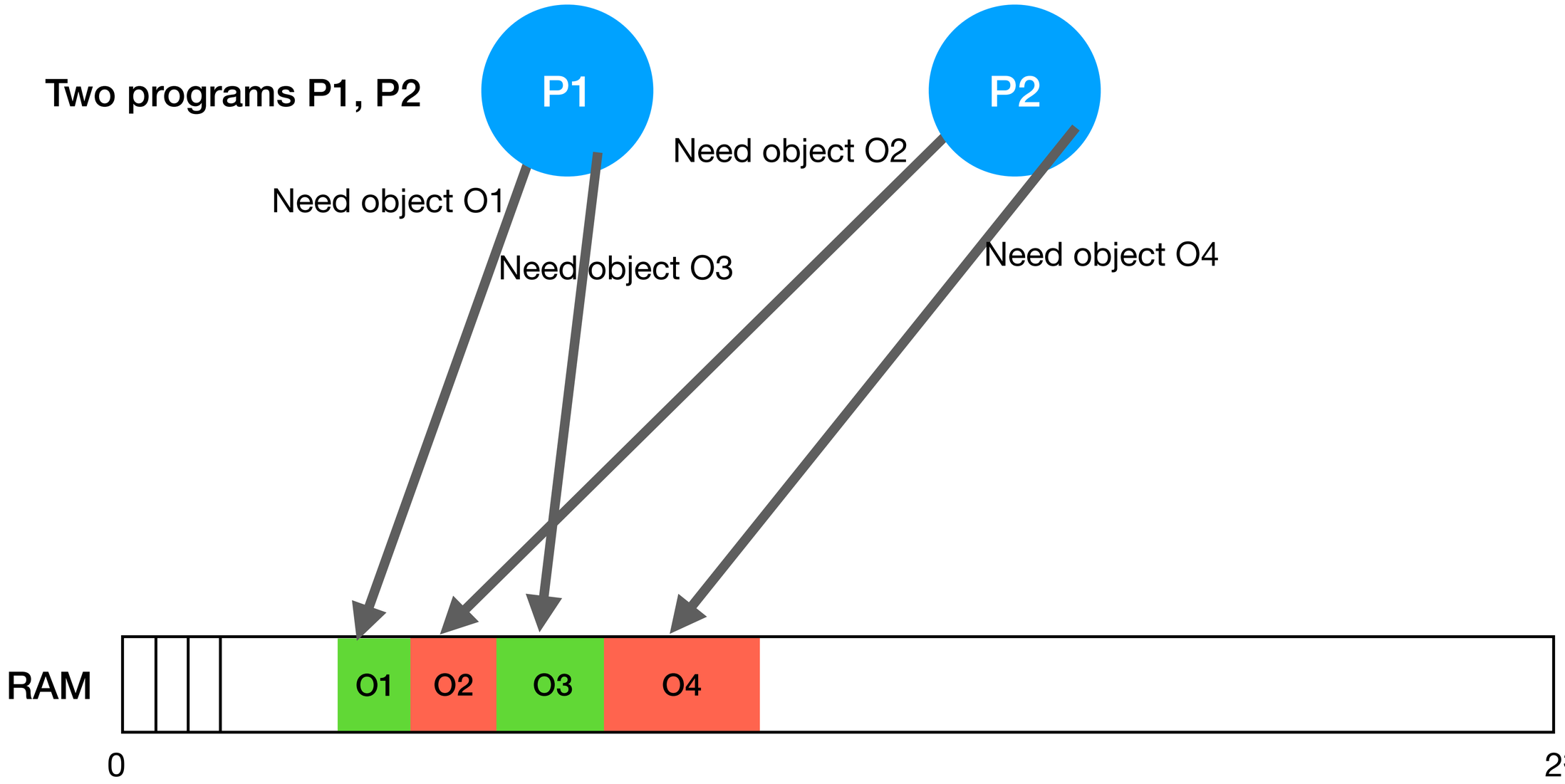
User-space application may only access the disk through the filesystem API (i.e., the OS prohibits direct block access to raw data). The OS isolates the user-space process from the disk.

- One program cannot interfere with another.
- Core provision of modern OSes and architecture

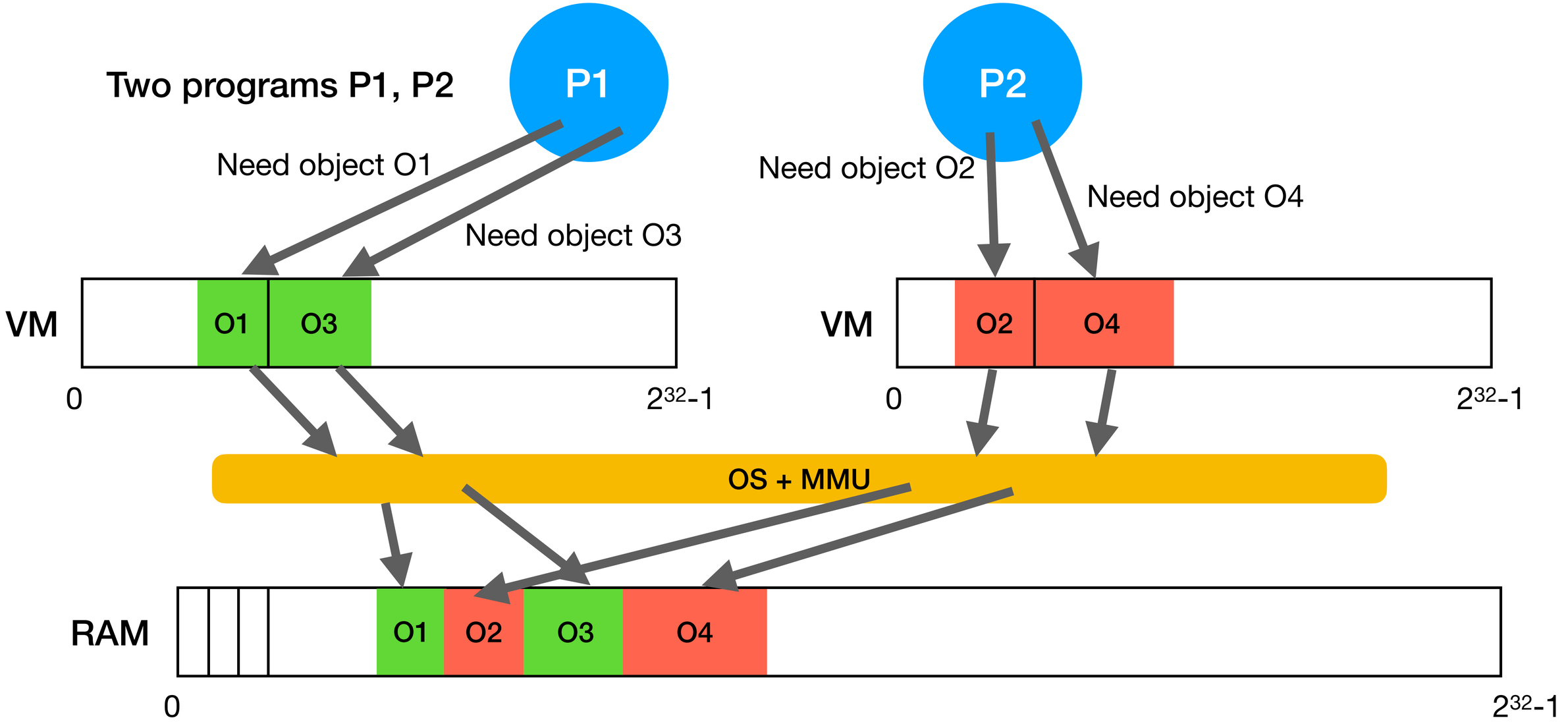
Process

- A running instance of a program, together with the state and resources needed for its execution.
  - has its own virtual address space
    - code, global/stack/heap data are exclusively owned by this process
  - has execution state such as registers and program counter
  - owns/uses resources such as open files, identified by integer descriptors

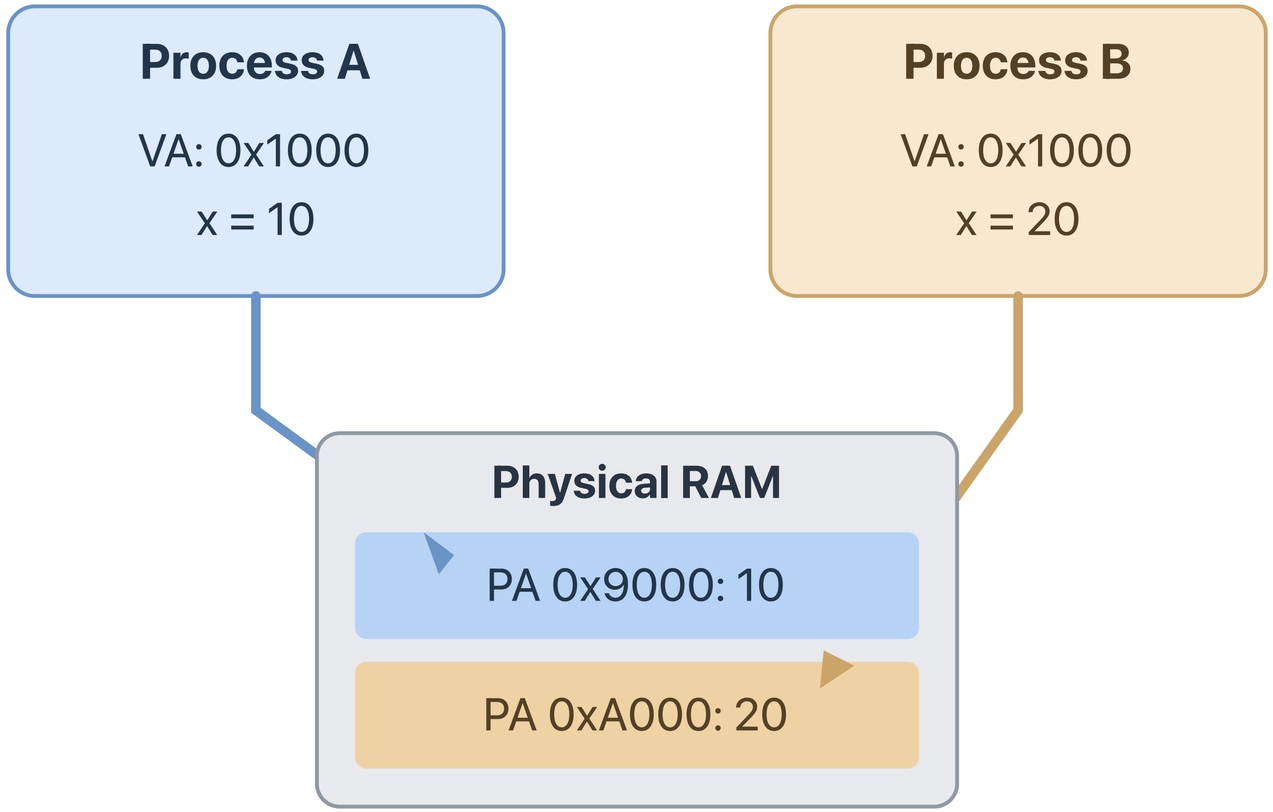
Using Main Memory Directly



Virtual Memory



Virtual Memory



Virtual Memory

- Makes each program “believe” it as the *entire* memory it itself!
- Combination of OS + hardware
- Programmers need **not** worry about other programs.
- Easier for OS to manage all programs.
- In this course—and in most of your career as a systems programmer—you will mainly deal with virtual memory.

Process

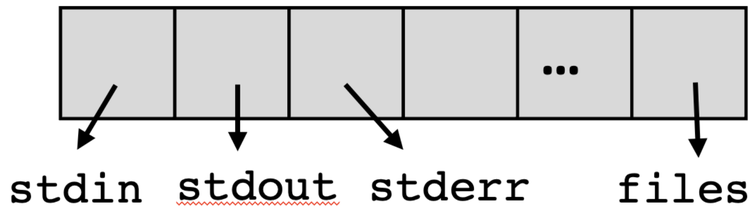
- A running instance of a program, together with the state and resources needed for its execution.
  - has its own virtual address space
  - has execution state such as registers and program counter
  - owns/uses resources such as open files
- Each process has an ID: pid
- Also has an owner
  - Usually is the user who starts the process, but can change.
- Current working directory: `pwd`
- Almost every process has a parent process.
  - PID 1 is the special root of the process tree.

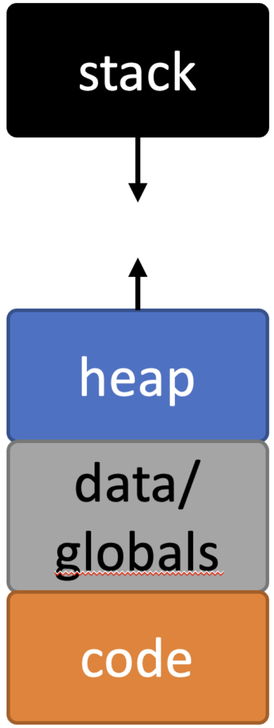
Process

pid
ppid
user: sibin
cwd: /home/sibin

command arguments
environment variables

descriptors





Unix

- A family of multitasking, multi-user OS.
- Developed at Bell Labs in the late 1960s/early 1970s
- Introduced many concepts still central to modern OS:
  - process
  - files and file descriptors
  - shells
  - pipes
- macOS is derived from a Unix OS called FreeBSD.
- Linux is a *Unix-like* OS that follows many of the same interfaces and design ideas.

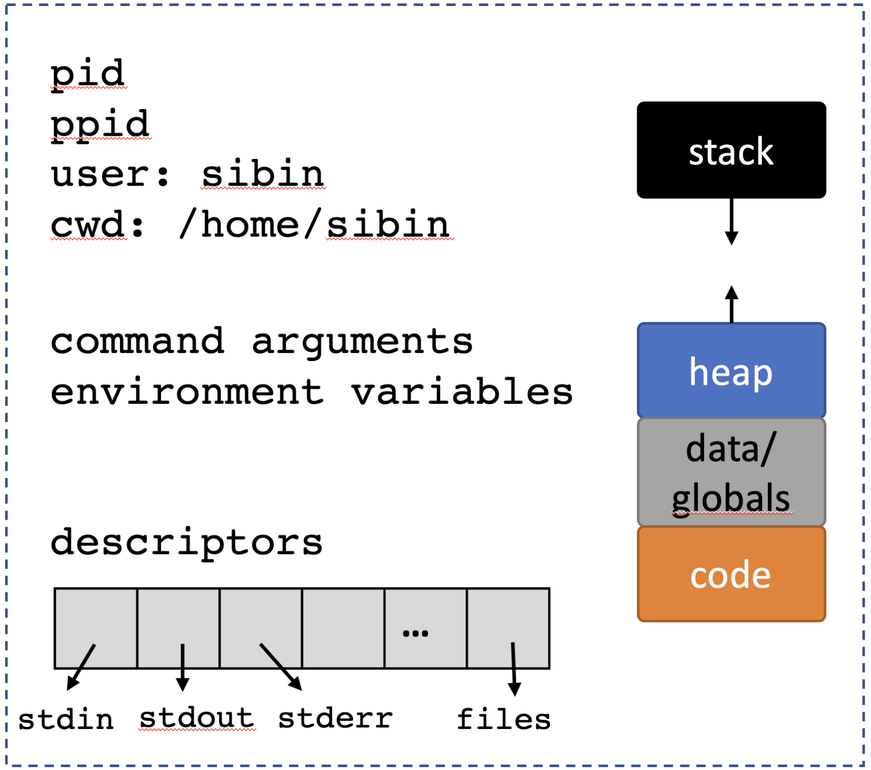
Unix Philosophy

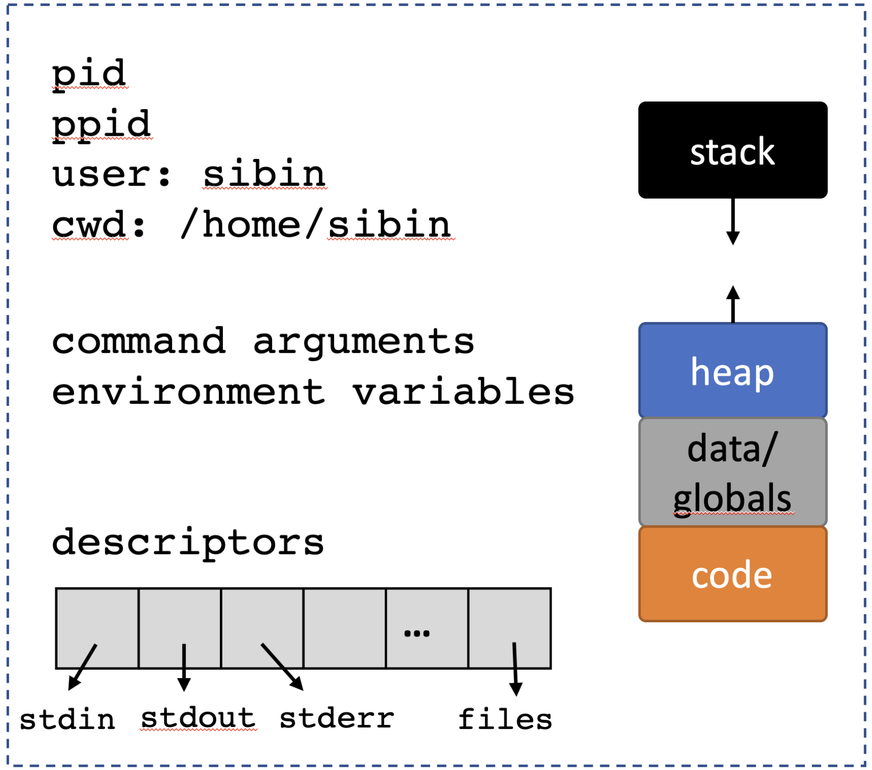
- Programs should be simple.
  - Instead of huge monolithic pieces of code, programs should be composed of smaller components.
- “Make each program do one thing well.”
- e.g., `ls`, `pwd`, `cat`

`fork()`

- Create a new, child process from the calling process.
- Defined in `<unistd.h>` (from libc)

After fork()


pid
ppid
user: sibin
cwd: /home/sibin
command arguments
environment variables
descriptors
stdin
stdout
stderr
files
stack
heap
data/
globals
code


pid
ppid
user: sibin
cwd: /home/sibin
command arguments
environment variables
descriptors
stdin
stdout
stderr
files
stack
heap
data/
globals
code

After `fork()`

- `fork()` creates a new process (child) by copying the calling process (parent).
- The child initially has almost the same *state* as the parent.
  - Code, stack, heap, global variables
  - Open files, working directory, environment variables
  - Initially share the same physical memory with its parent.
- The child has its own PID; its PPID is the parent's PID.
- After `fork()`, both processes continue executing independently.





fork()

```
pid_t fork(void);
```

- Include from <unistd.h>
- `pid_t`: process identifier defined in `<sys/types.h>`.
  - A signed integer, usually by `typedef int`

Return value	Returned in	meaning
child's PID (>0)	parent	child successfully created
0	child	inside child
-1	parent	`fork()` failed; no child created

fork() Example

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/types.h>
 4
 5 int main()
 6 {
 7     printf( "BEFORE calling fork()” ) ;
 8
 9     pid_t pid = fork() ;
10
11     printf( "AFTER calling fork()” ) ;
12
13     printf( "\n" ) ;
14     return 0 ;
15 }
```



What would be printed?

```
BEFORE calling fork()AFTER calling fork()

BEFORE calling fork()AFTER calling fork()
```

Output Buffer and Flush

- `printf()` writes to an internal buffer, which may not be immediately displayed.
- `fork()` copies the parent's unflushed buffer into the child.
- Both processes eventually flush their own buffers.
- Internally buffered streams can be *flushed*.
  - Call `int fflush(FILE *stream);`
  - Add `'\n'` to the first `printf()`. But this does not ensure immediate flushing.

fork() Example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf( "BEFORE calling fork()" ) ;
    fflush(stdout); // Flush the standard output

    pid_t pid = fork() ;

    printf( "AFTER calling fork()" ) ;

    printf( "\n" ) ;
    return 0 ;
}
```

```
BEFORE calling fork()AFTER calling fork()

AFTER calling fork()
```

Parent Process

- Parents are responsible for managing their children!



- so, they `wait()`!

wait()

```
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Called by a parent, wait for state change in a child process
- Parent can wait for
  - any child; e.g., `wait(null)` or `waitpid (-1, ….)`
  - a specific child
  - a subset of child
- A child changes state if
  - child terminates
  - stopped by a signal
  - resumed by a signal

State Change Action

Child status	Action
already changed	return immediately to parent
executing normally	parent waits/blocks for child

`wait()` return value

Return value	Meaning
pid via pid_t	pid of terminated child
-1	error

wait Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        // Do whatever it does;
    } else if (pid > 0) {
        // Parent
        int status;
        wait(&status);

        if (WIFEXITED(status)) {
            printf("Child exited with code %d\n",
                   WEXITSTATUS(status));
        }
    }
    return 0;
}
```

Process Creation/Termination/Coordination APIs

Name	Action
`fork`	create a process
`getpid`	get ID of the current process
`getppid`	get ID of the parent process
`exit`	terminate a process
`wait`	parent waiting for child to exit
`exec`	replace the current program with a new one (same pid)

Process vs Program

- Program: source code or compiled version of code
- Process: an executed/running instance of a program