CS 344: Operating Systems I

03.15: Recap – summary

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Announcement

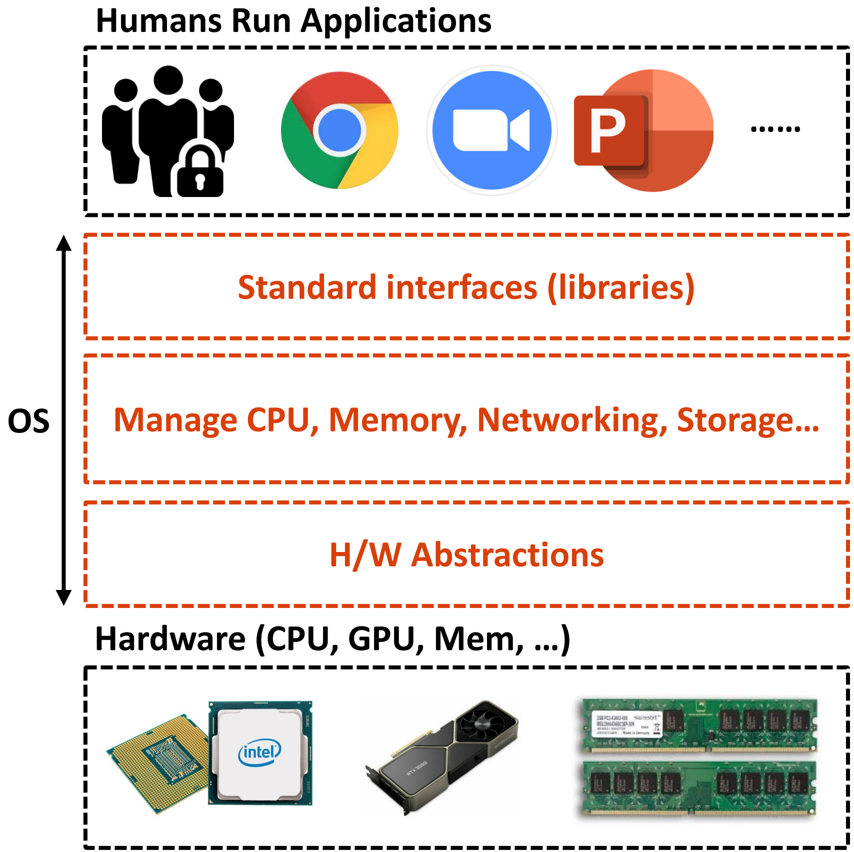
- Upcoming deadlines
  - 3/15: Programming assignment V
  - 3/17: Extra credit opportunity (Linus Torvalds article)
  - 3/20: Midterm quiz IV
  - 3/20: The other three extra credit opportunities
  - 3/22: Late submissions for programming assignments only

Outline

- Part I:
  - Process
  - Threads
  - Scheduling basics
- Part II:
  - Files and I/Os
  - File system basics
- Part III:
  - IPC
  - RPC
  - Networking
- Part IV:
  - Synchronization
  - Rust

Operating systems

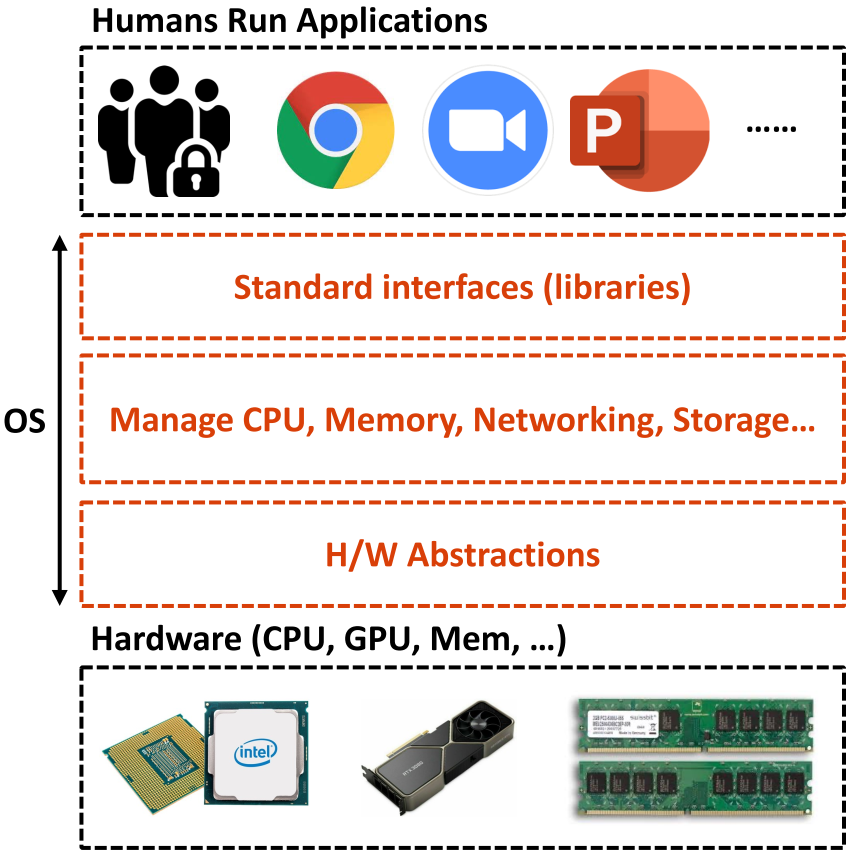
- **Operating systems (OS)**
  - Computer software that lies between hardware and user applications



Operating systems

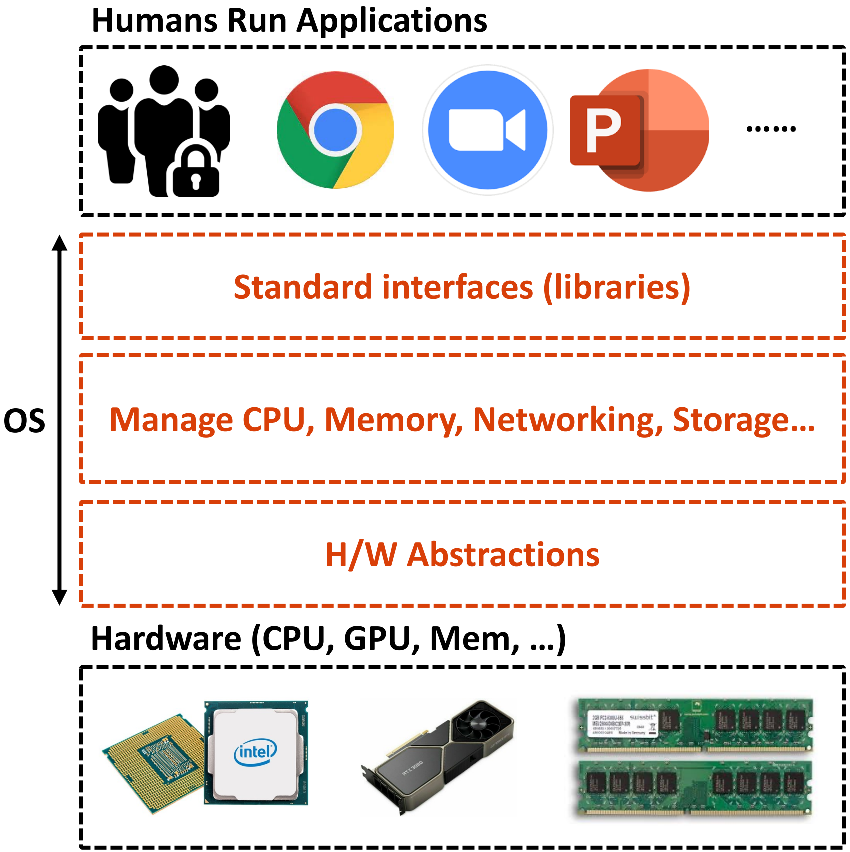
- **Operating systems (OS)**
  - Computer software that lies between hardware and user applications

- **Why do we learn OS?**
  - To understand better how computers think (how you can run your programs in OS)



Operating systems

- **Operating systems (OS)**
  - Computer software that lies between hardware and user applications
- **Why do we learn OS?**
  - To understand better how computers think (how you can run your programs in OS)
- **Functionalities of modern OS**
  - Manage resources
  - Provide abstractions
  - Provide standard interface (e.g., C libraries)



Program

- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute

You

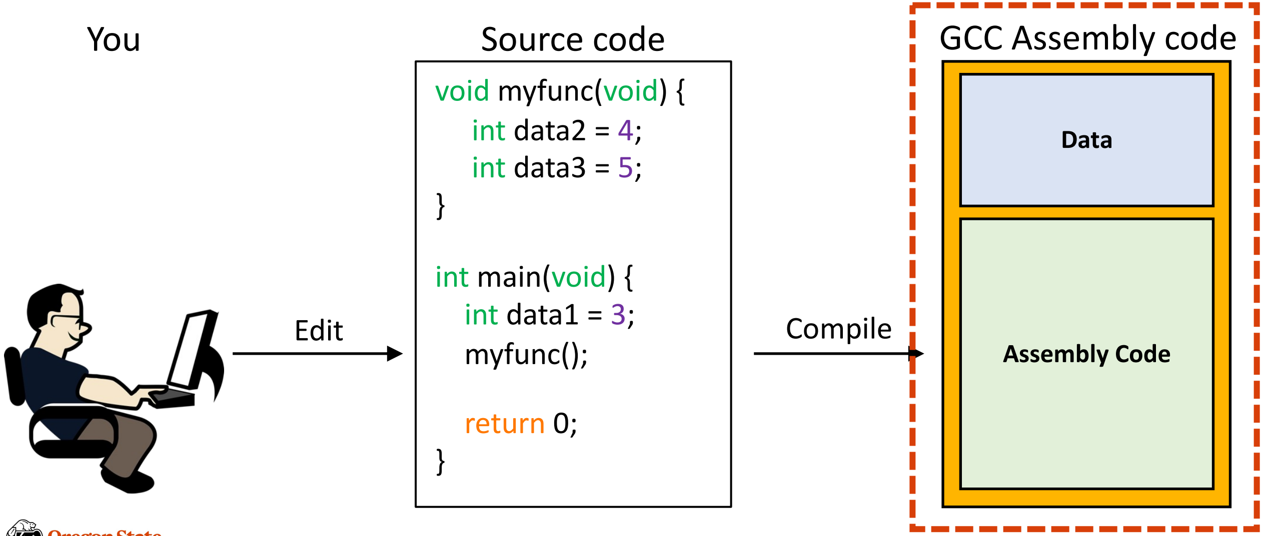
Edit →

Source code

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

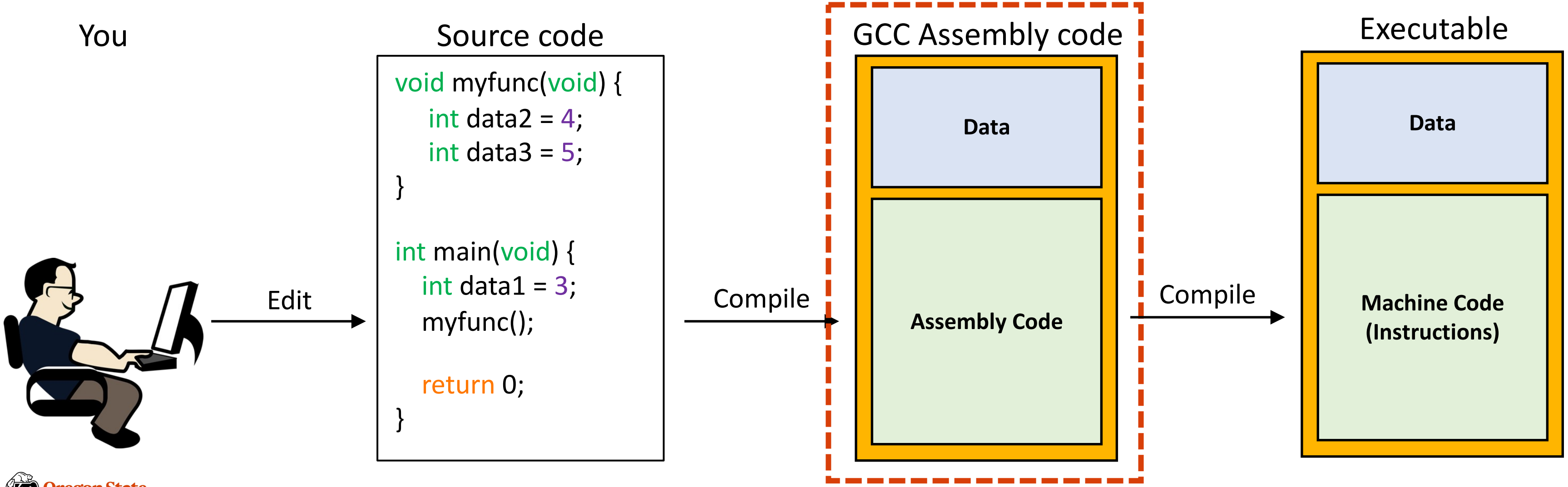
Program

- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute



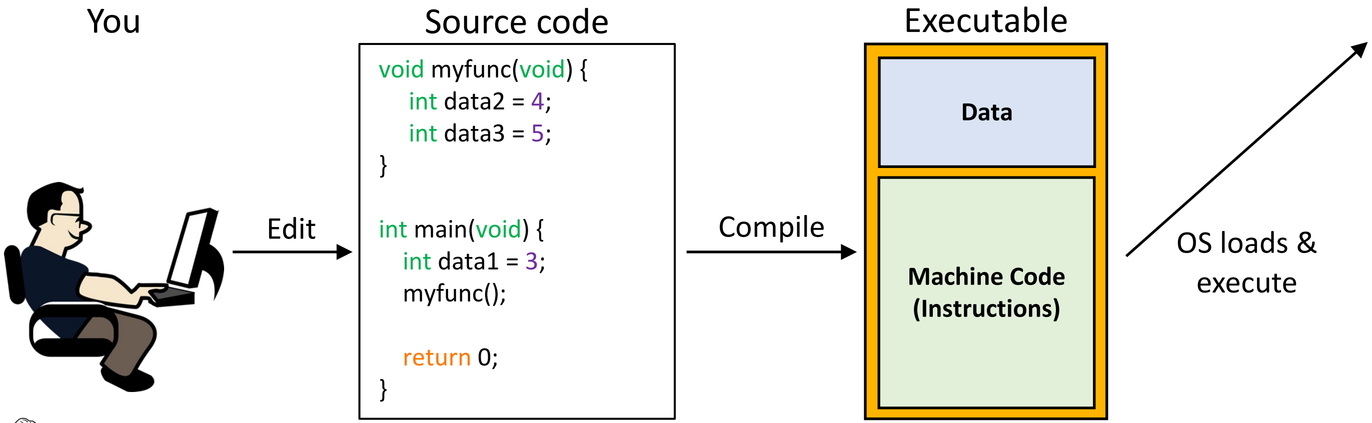
Program

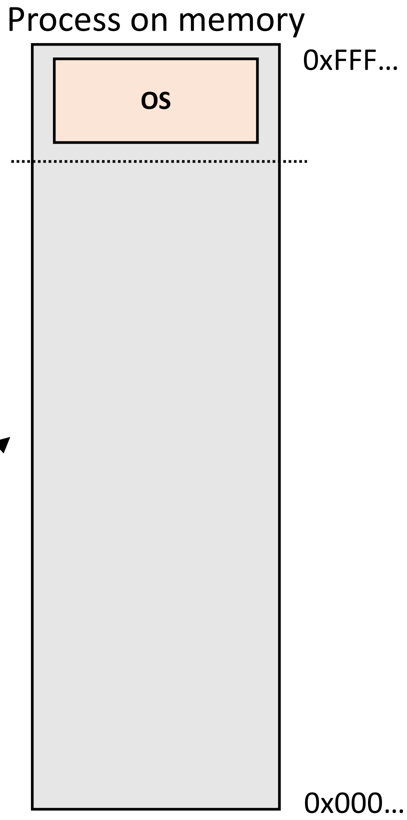
- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute



Provide abstraction: A process

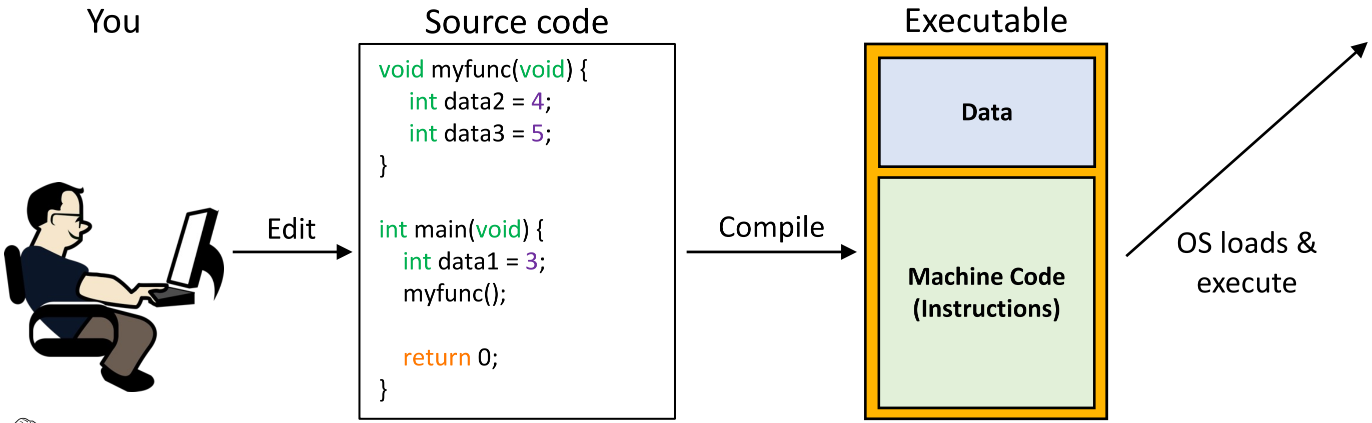
- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Process segments:**
    - **Code**, **data**, **heap** and **stack**

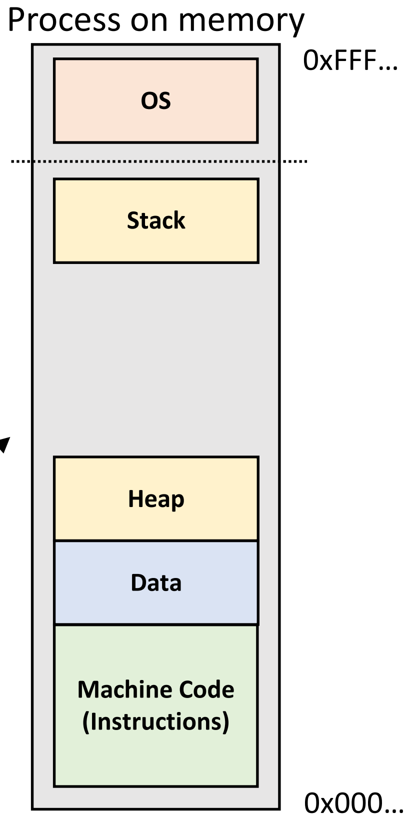




Provide abstraction: a process

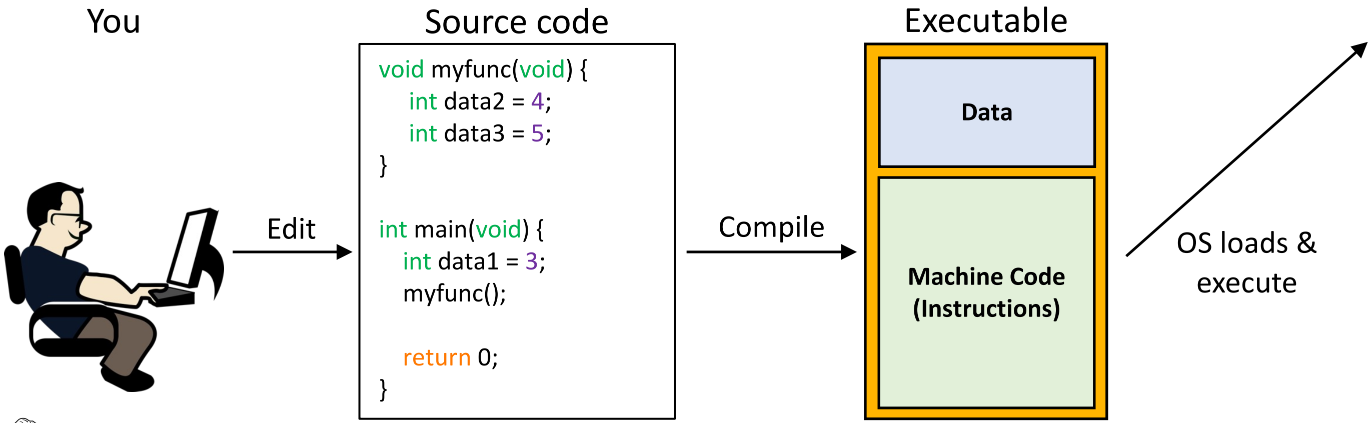
- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Process segments:**
    - **Code**, **data**, **heap** and **stack**

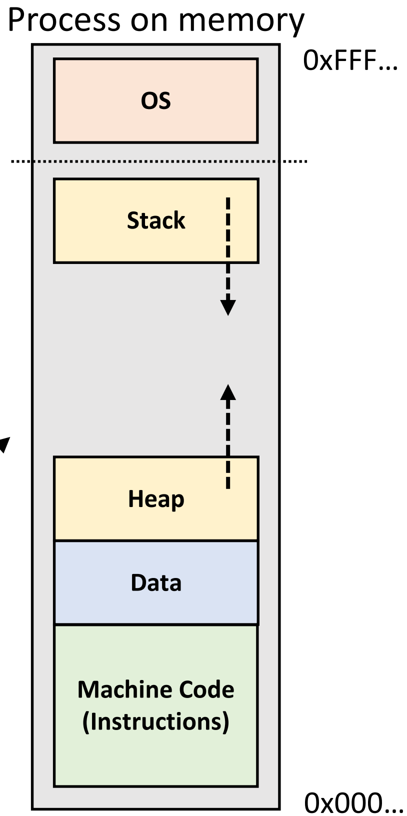




Provide abstraction: a process

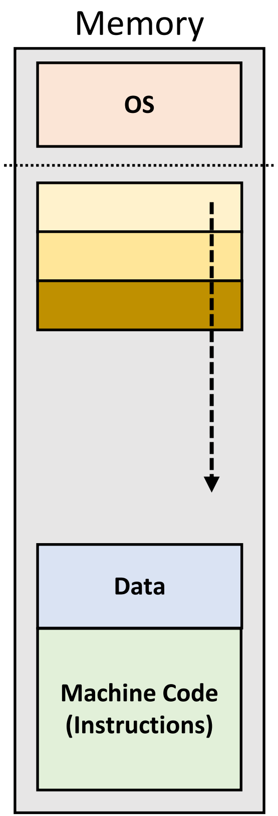
- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Process segments:**
    - **Code**, **data**, **heap** and **stack**





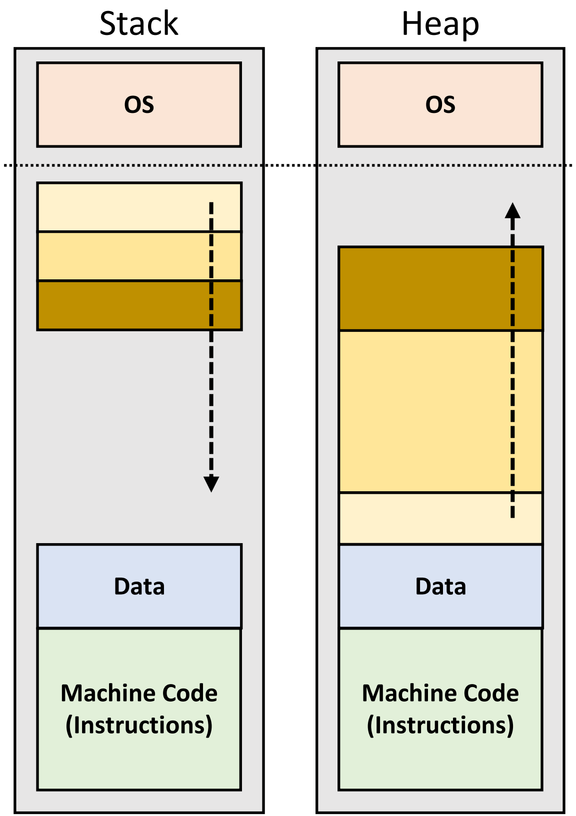
Stack and heap segments

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Stack
    - **OS controls** the memory allocations (size)
    - Store data in Last in first out (**LIFO**) manner
    - Stack mostly holds data initialized within a function



Stack and heap segments

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur



Process creation: system call

- System call
  - **Definition:** a user-level function call to request a service from the OS
  - **Example:** when we allocate memory with "malloc()"

User program

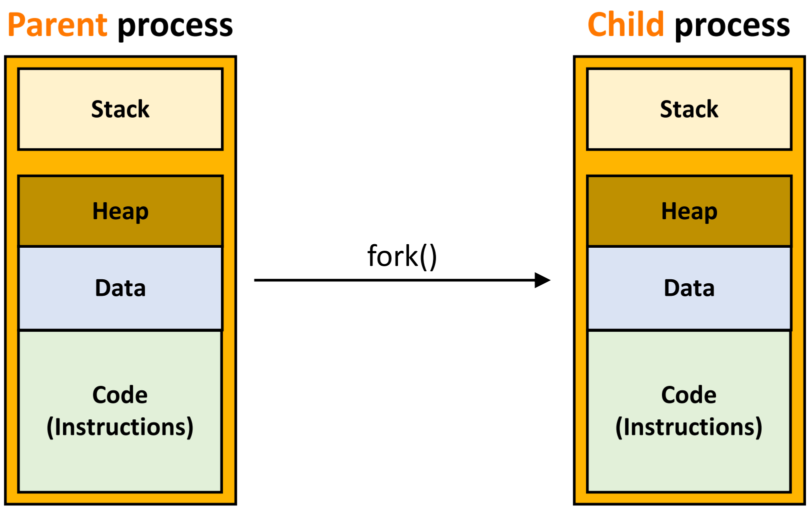
```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);
   free(data2);
}

int main(void) {
   int num = 100;
   myfunc();

   return 0;
}
```

OS Kernel

Kernel allocates the memory space that has the size of 5 bytes, and return the starting address of this memory to the user's program

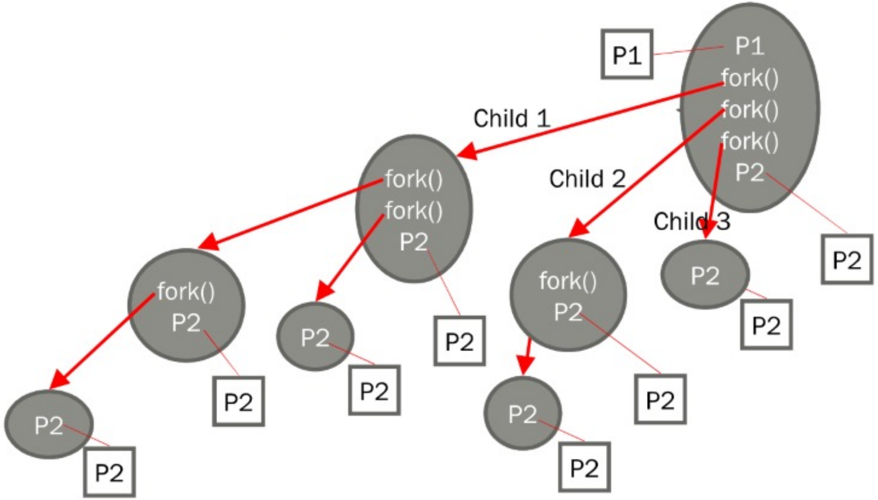
Process creation: fork system call

- fork() system call
  - **Operation:**
    - Create a new process that is an exact copy of the calling process
    - Return the process ID (PID) of a new process (and if it's in child, returns 0)



Process management

- **fork() tree**
  - OS manages processes with a tree
  - Use ($ pstree) command to see the tree!
  - Root of the fork() tree (in Linux)
    - PID=0: **Sched** (swapper) process
    - PID=1: **Init** process



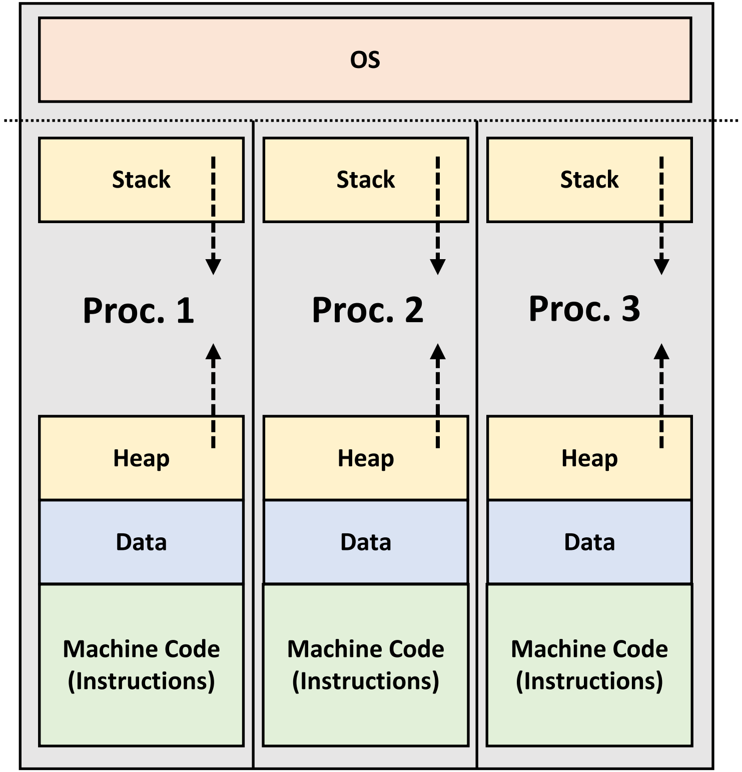
- Properties
  - User processes always have a parent
  - If we kill the parent, all the child processes will be killed, too (an exception, any process launched by $ nohup or $ disown)
  - PIDs allocated by OS increases as we fork() more

Thread

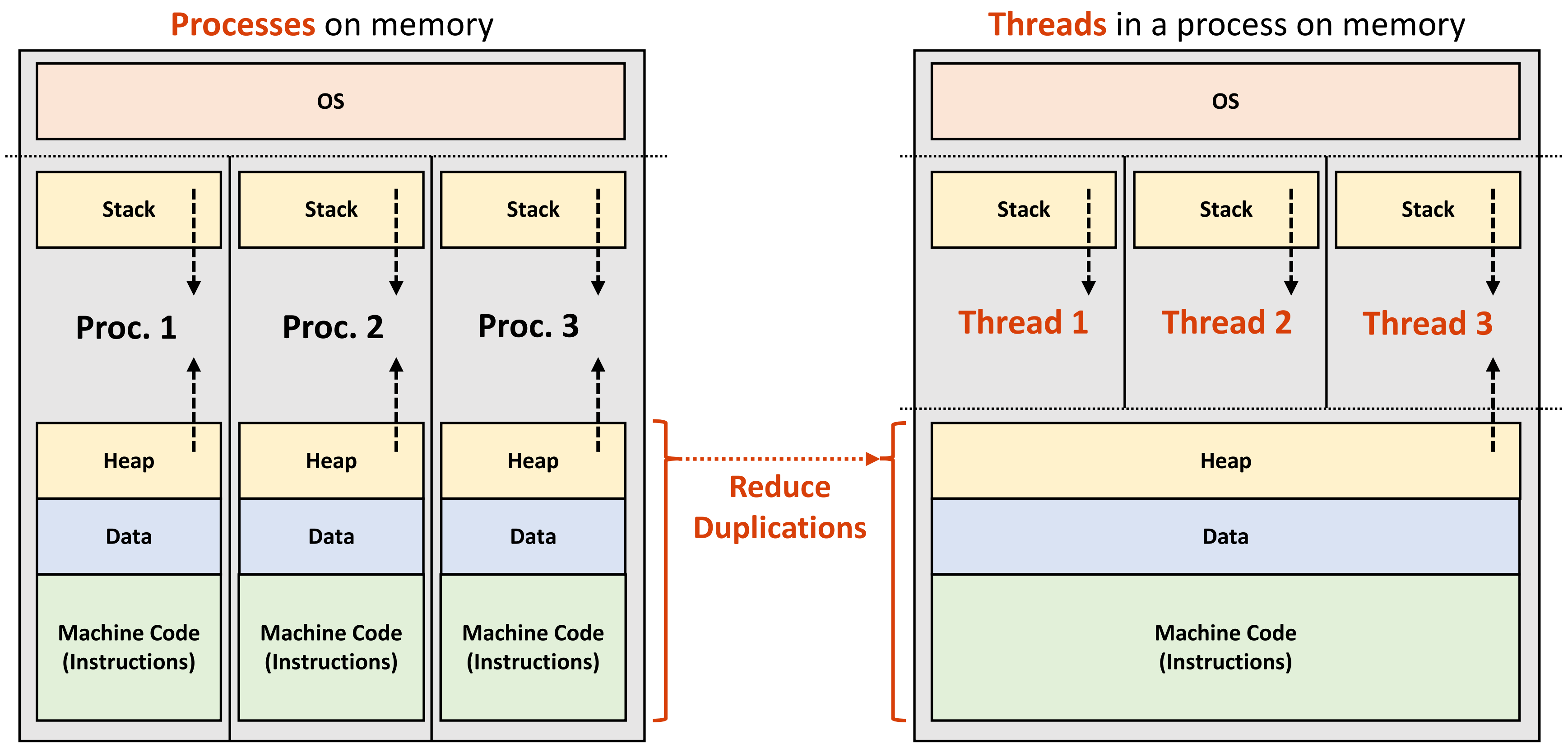
- Thread
  - **Definition:** a smallest schedulable execution context
  - **Terminology:**
    - Smallest: it’s much light-weight than a process
    - Schedulable execution context: one thread can run on a CPU at a time

Thread vs. process

Processes on memory



Thread vs. process – cont'd

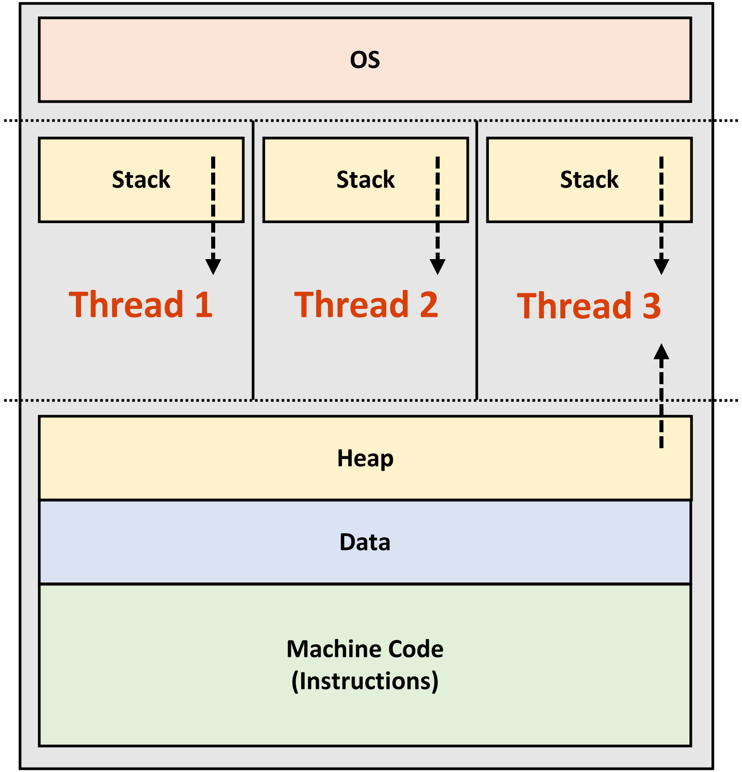


Thread vs. process – cont'd

- Threads share:
  - **Code** and **data** segments
  - **Heap** memory (ex. global variables)
  - Open files (ex. I/O access points)

- Threads **do not** share:
  - **Stack** segments, e.g.:
    - arguments passed when we launch them
    - local variables we initialize within them
    - return address, when they terminate (OS II)
  - Running contexts, e.g.:
    - thread state
    - stack pointer
    - …

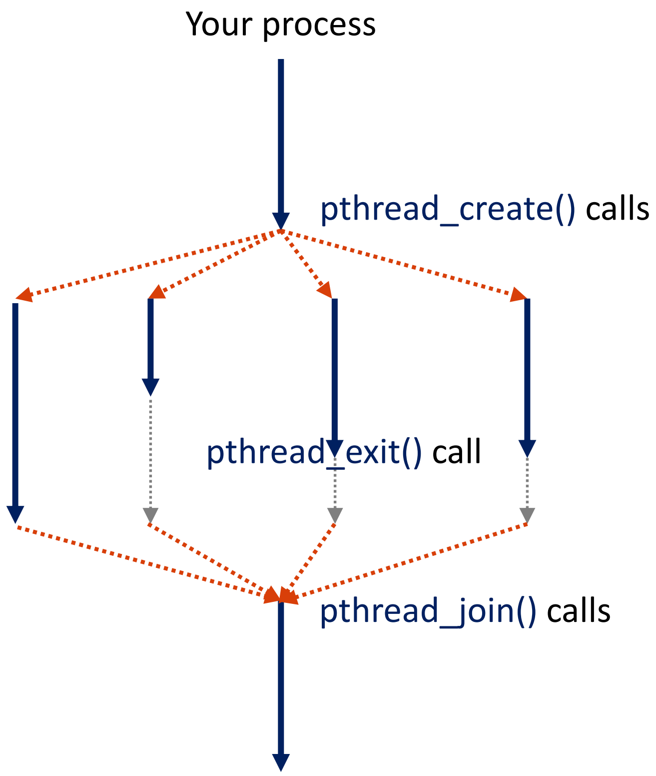
Threads in a process on memory



THREAD CREATION: THREAD-SPECIFIC SYSTEM CALLS

- Thread-specific system calls
  - **pthread_create**(thread, attribute, subroutine, subroutine-arguments);
    - Create a new thread executing the *subroutine* in the current process
    - Returns zero if it's successful, otherwise it returns errno
  - **pthread_exit**(return-value);
    - Terminate the thread and returns the *return-value* to any successful join
    - Note: If a thread terminates, it will be automatically called and always return success
  - **pthread_join(**thread, return-value-loc**);**
    - Suspend execution of the calling thread until the *thread* terminates
    - Once the thread terminates, the function will copy the return value to *return-value-loc*
    - Returns zero if it's successful, otherwise it returns an error

Thread programming pattern: Fork-Join



- **Fork - Join** Pattern
  - Fork: Main process creates a set of sub-(or child)-threads that runs a function
  - Each thread exits if the function returns
  - Join: Main waits until all the threads exit
- **Example:** download a large file
  - Splits a file into smaller chunks
  - Create a thread for downloading each
  - Sum-up all the downloaded chunks and combine them to create a single large file

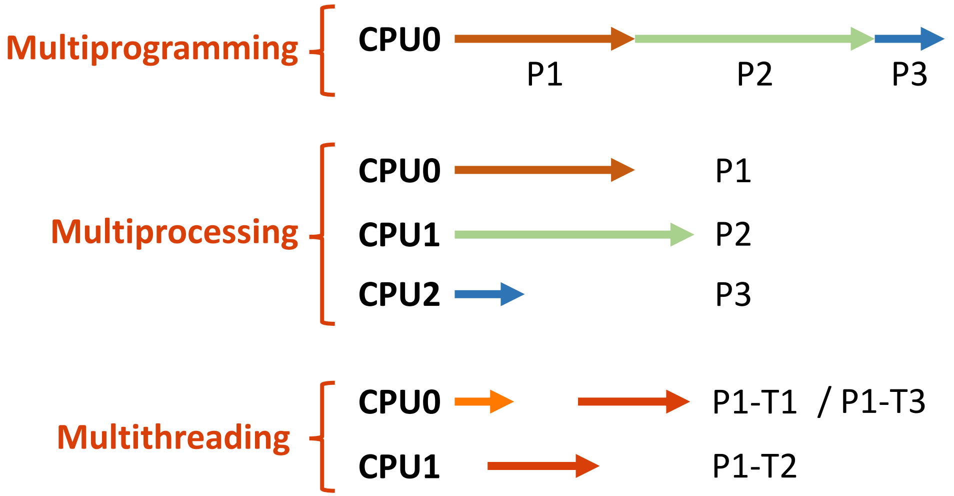
Thread management

- **(Linux)** OS
  - A thread is treated as the same as a process
  - (Linux) thread control block ≈ process context

- A thread can have **three states**:
  - **Ready:** a thread is created and ready to run, but not running now
  - **Running:** a thread running now
  - **Blocked:** a thread is unable to run (terminated or errors)

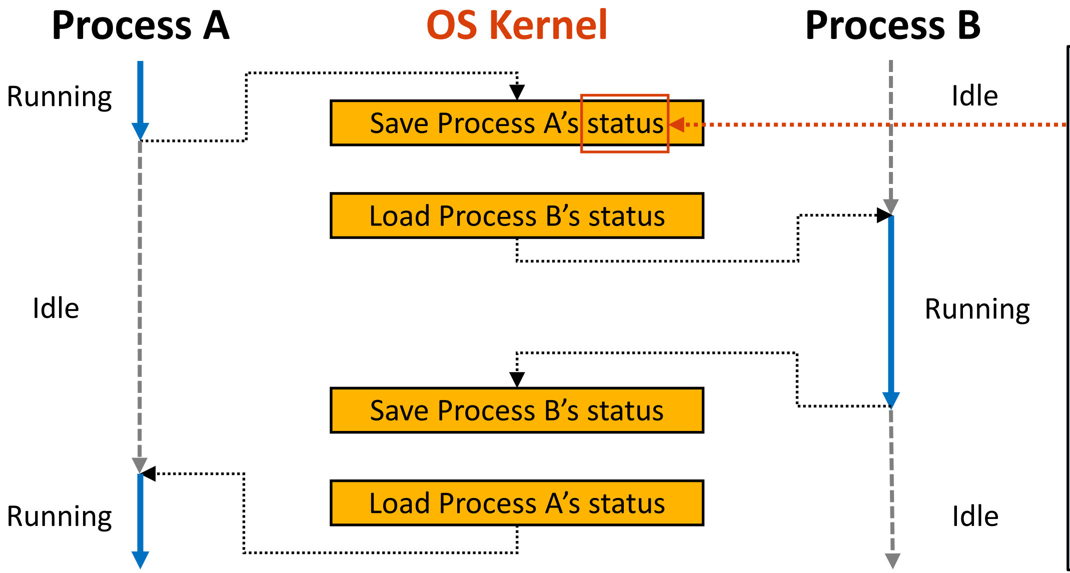
Scheduling: terminology

- **Three confusing terms:**
  - Multiprogramming vs. multi-processing vs. multi-threading
    - Multi-programming: multiple jobs (or processes)
    - Multi-processing: multiple processors (CPUs)
    - Multi-threading: multiple threads



Scheduling: context switch

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



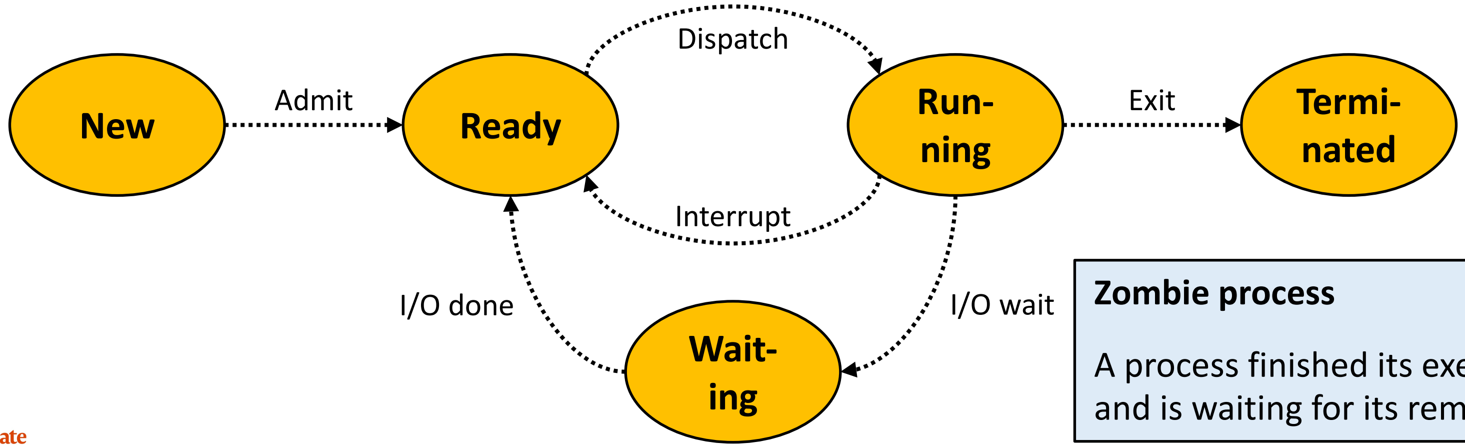
Recall: Process control block

A structure in OS that contains a set of information required to run a process on a CPU. Recall that Linux has *task_struct*.

- CPU#
- Program counter
- Instruction pointer
- Heap/stack pointer
- Process state [!]
- …

Scheduling: process states

- A process can have **five states**:
  - **New:** a process (or thread) is being created (by fork())
  - **Ready:** the process is waiting to run
  - **Running:** the process is running on a CPU(or CPUs)
  - **Waiting:** the process is waiting for some events to occur (*e.g.*, a data loaded from storage)
  - **Terminated:** the process has finished execution; waiting for removal
- State transition (life cycle):



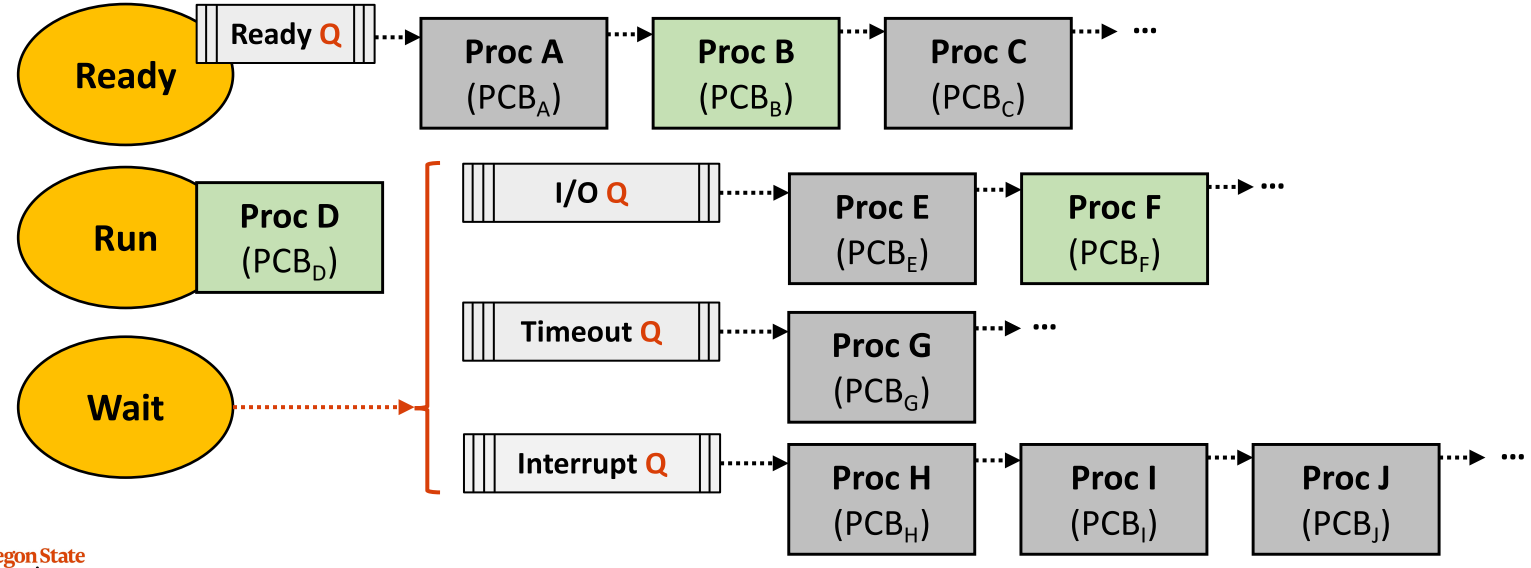
Zombie process

A process finished its execution and is waiting for its removal

Scheduling in OS

- **Scheduling**
  - **Definition:** an OS activity that schedules processes in different states
  - **Note:** OS implements queues to hold multiple processes in the same state

- **Illustration (single CPU)**

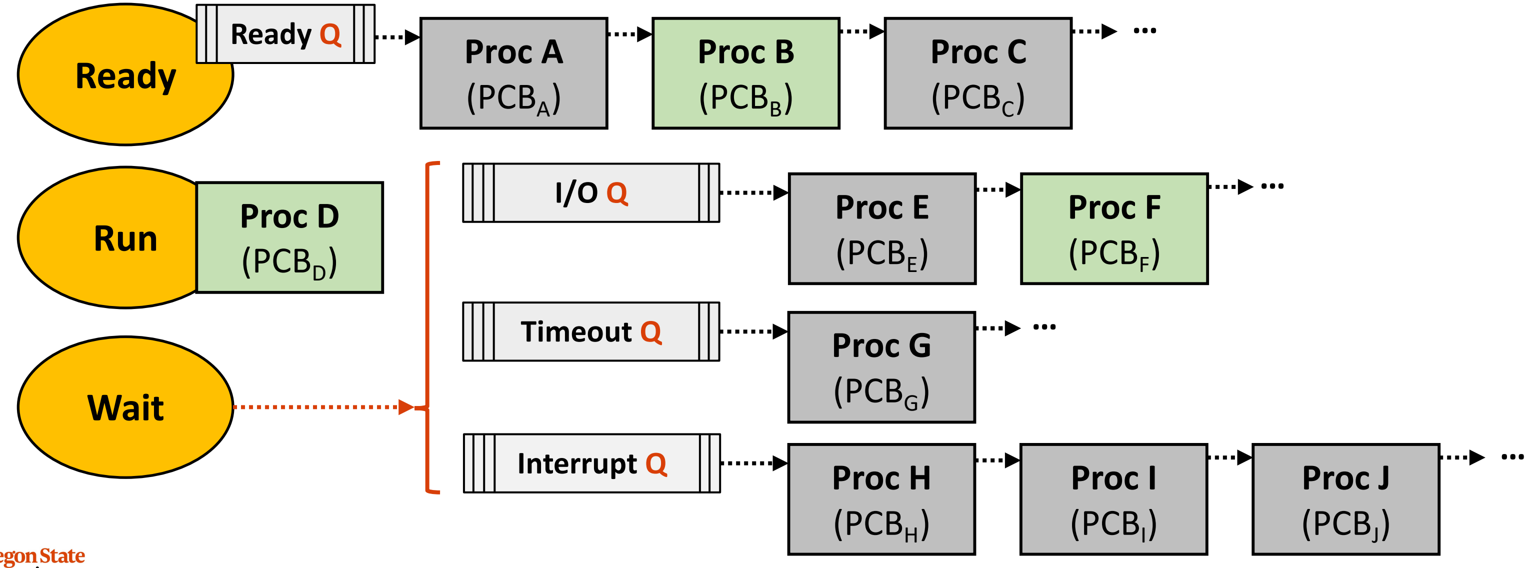


Scheduling in OS: Example

- **Scheduling**
  - **Definition:** an OS activity that schedules processes in different sta
  - **Note:** OS implements queues to hold multiple processes in the sa
- **Illustration (single CPU)**

Illustrated Example

1. Kicks out Proc D (timeout)
2. Runs Proc B
3. Puts Proc F in the ready Q
   (I/O has done, in this case)



Scheduling: OS scheduler

- **(OS) Scheduler:**
  - **Definition:** An OS task (process) that manages the process scheduling activity

- **Implementation**

```
while ( <some condition,
          but eventually will be infinite>) {

   RunProcess( curProc );
   newProc = chooseNextProc();
   saveCurrentProc( curProc );
   LoadNextState( newProc );

}
```

  - It is also a process (an *infinite* loop)
  - The scheduler process terminates if we *stop* (turn-off) a computer

Scheduling: OS scheduler – cont'd

- **What triggers OS scheduling?**

```
while ( <some condition,
            but eventually will be infinite>) {

    RunProcess( curProc );
    newProc = chooseNextProc();   <······· Yield or interrupt triggers this code line
    saveCurrentProc( curProc );
    LoadNextState( newProc );

}
```

  - RunProcess(): a CPU executes the machine code of "curProc"
  - chooseNextProc(): OS kernel selects the next process to run
  - saveCurrentProc(): OS kernel saves the CPU's state to "curProc"
  - loadNextState(): OS kernel stores "newProc" state to the CPU

Outline

- Part I:
  - Process
  - Threads
  - Scheduling basics
- Part II:
  - Files and I/Os
  - File system basics
- Part III:
  - IPC
  - RPC
  - Networking
- Part IV:
  - Synchronization
  - Rust

File and directory

- File
  - **Definition:** a named collection of data (*e.g.*, movie.csv containing movie data)
  - **POSIX** : a sequence of data bytes
  - ***NIX OS** : **everything** is a file

- Directories
  - **Definition** : a folder containing files and directories

Users, groups, and permissions

- Users and groups
  - **User** : owner of a file or a directory
  - **Group** : the group where users are
  - **Others:** all the other users

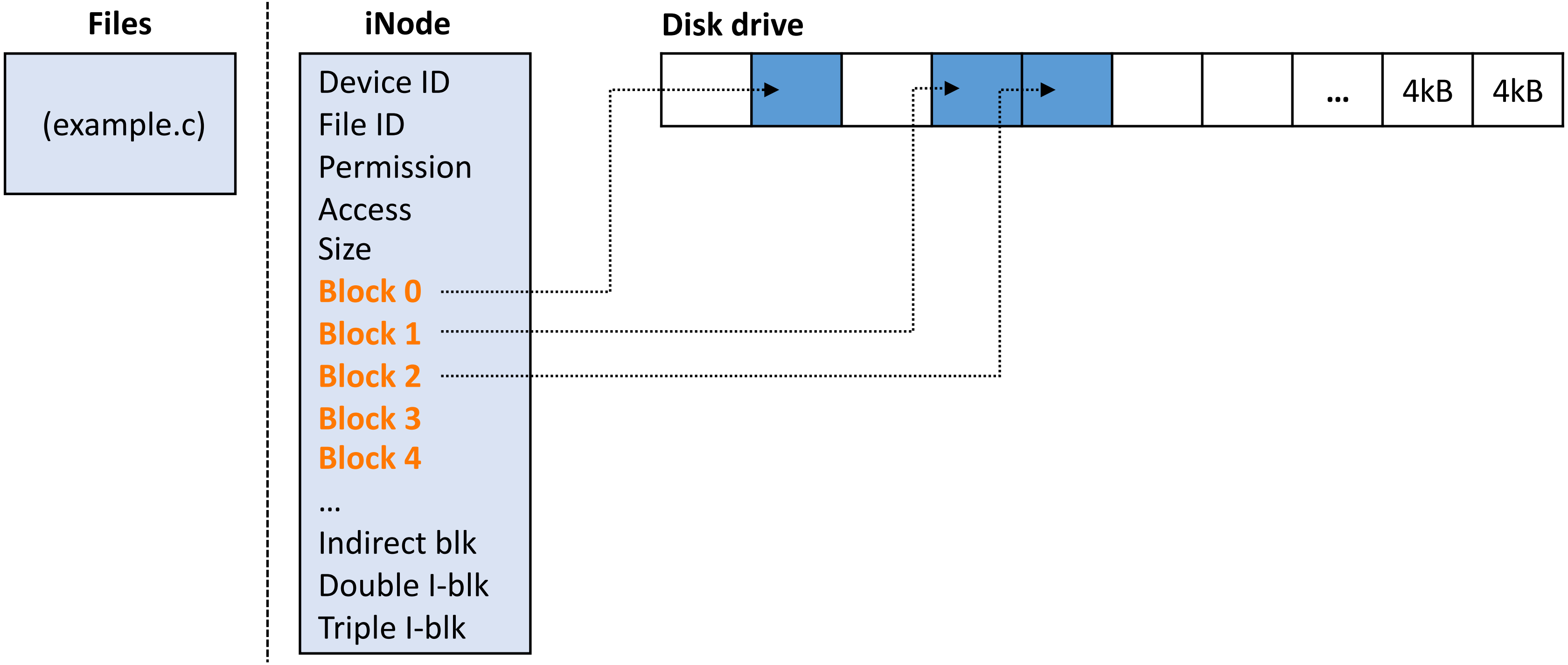
- Permissions
  - **Read** : one can read files and directories with ‘r’ permission
  - **Write** : one can write files and dirs. with ‘w’ permission
  - **Execute:** one can execute files and dirs. with ‘x’ permission

File system structure

- Basic components
  - File : a named collection of data
  - Directory: a file that holds other files as data
- Access control, permission
  - Access control: user, group, and others (u, g, o)
  - Permission : read, write, and execute (r, w, x)
- Filesystem structure
  - iNode: a data-structure that describes a file-system object
  - Block : a unit of data storage, the size is defined by OS (e.g., 4kB)

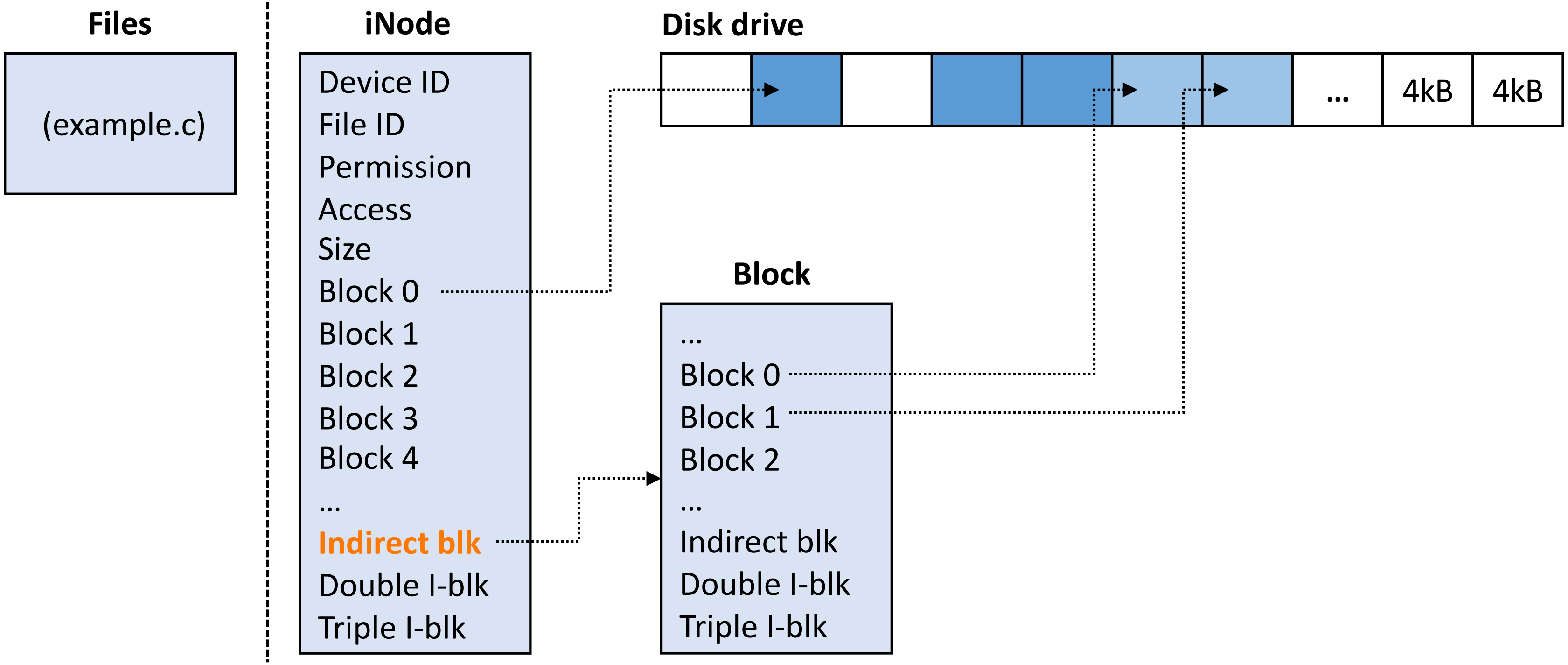
Filesystem structure overview

- A file stored in a filesystem (12 blocks ≈ 48kB)



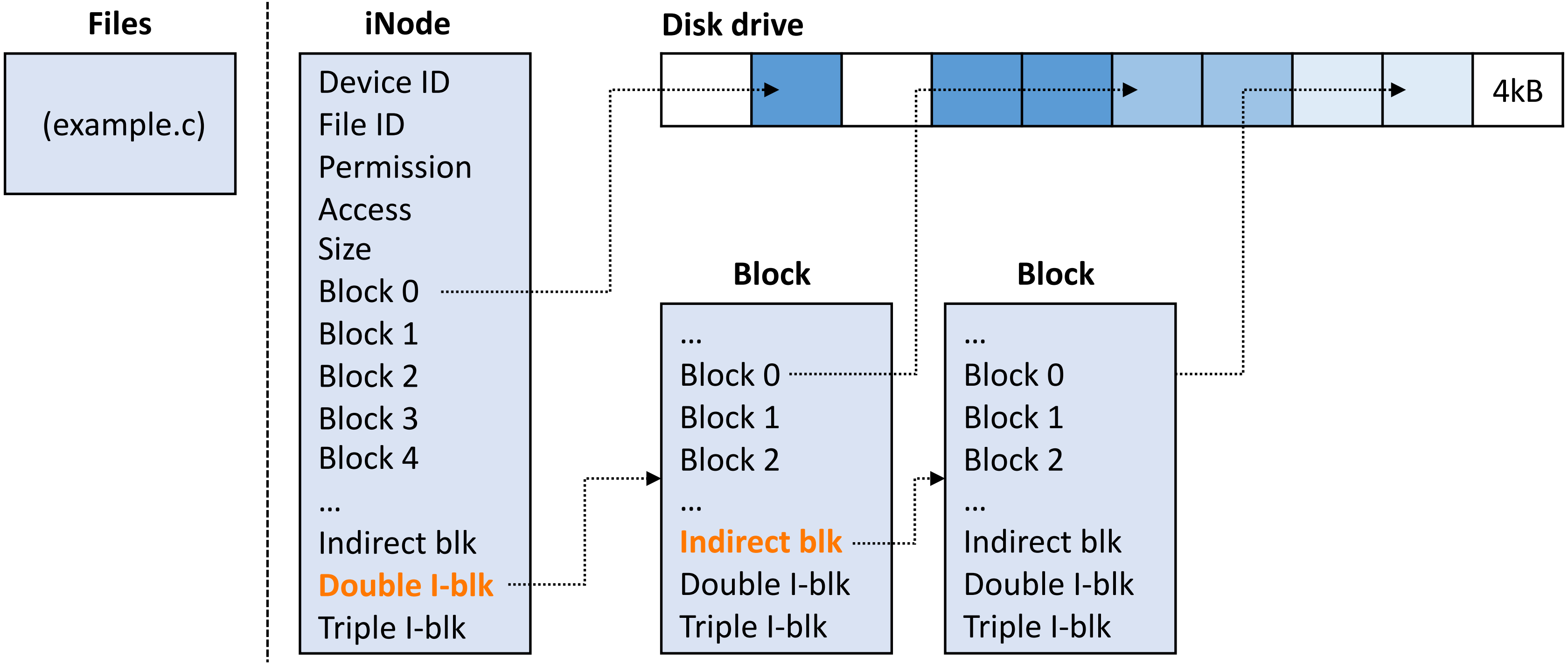
Filesystem structure overview – cont'd

- A (larger) file stored in a filesystem (indirect block ≈ 4MB + 4kB)



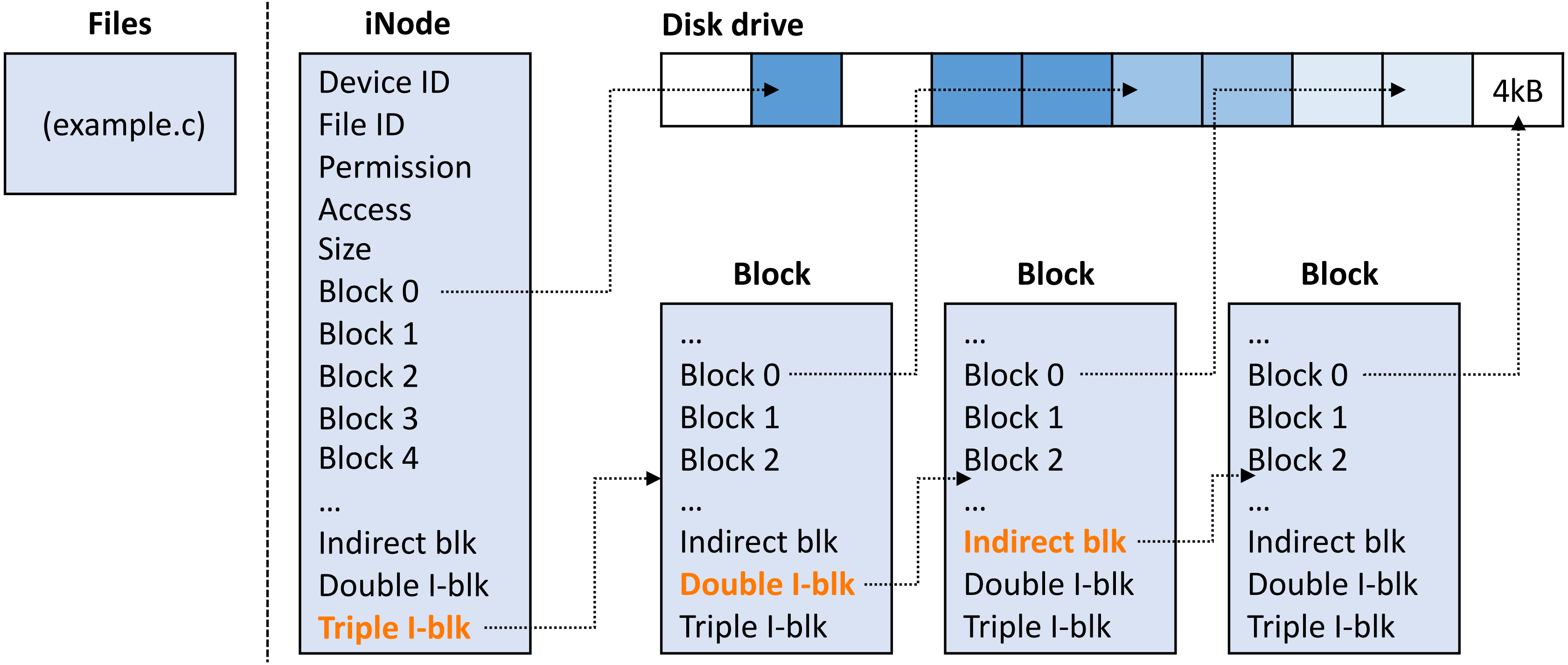
Filesystem structure overview – cont'd

- A (larger) file stored in a filesystem (double I-blk ≈ 4GB +4MB +4kB)



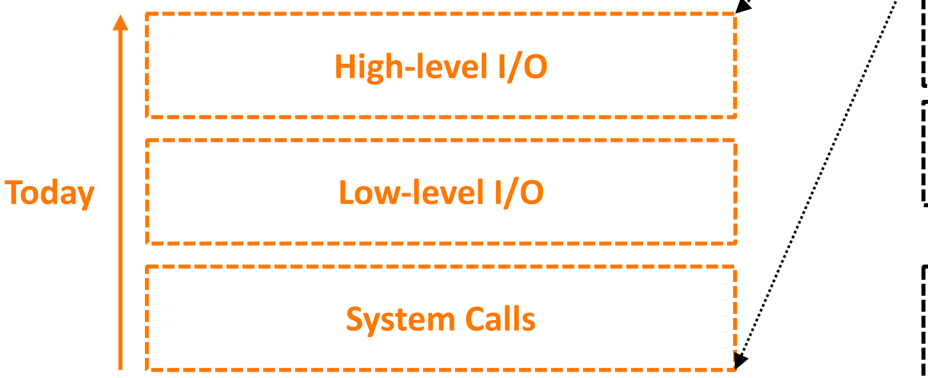
Filesystem structure overview – cont'd

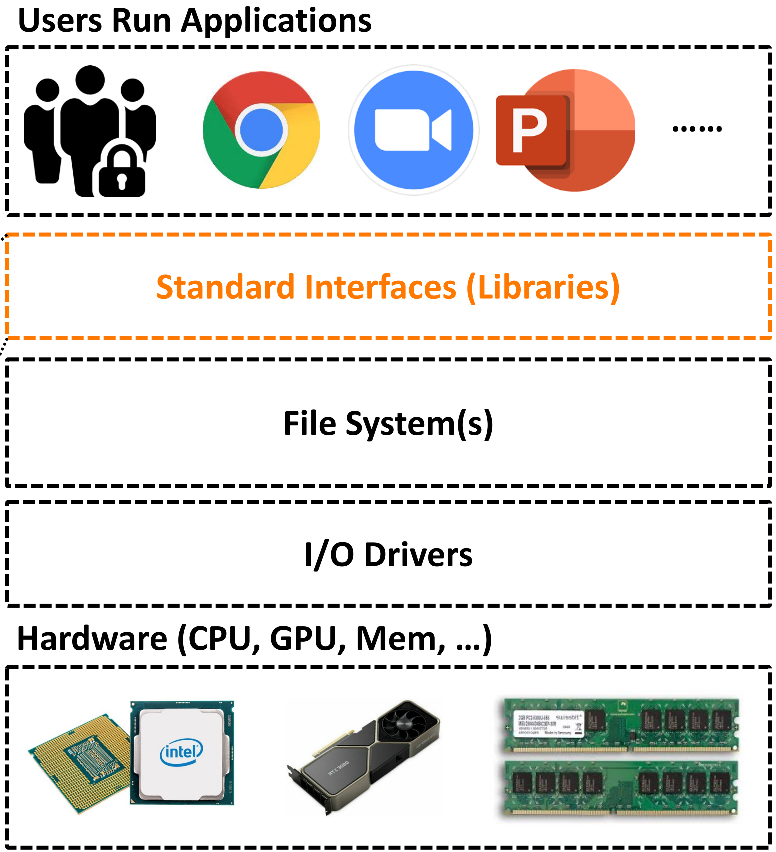
- A (largest) file stored in a filesystem (triple I-blk ≈ 4TB +4GB +4MB +4kB)



I/O

- I/O
  - **Definition :** input and output
  - **Def (*NIX):** any operation that read/write system services (*NIX OS: everything is a file)





Low-level I/O

- File descriptors (fd)
  - **Definition** : an integer that uniquely identifies an open file in Linux
  - **System calls:** (fctrl.h)
    - int open(const char *filename, int flags, mode_t *mode)
    - int create(const char *filename, mode_t *mode)
    - int close(int *fd)

  - **Standard file descriptors:**
    - STDIN_FILENO : **0**
    - STDOUT_FILENO: **1**
    - STDERR_FILENO : **2**

Low-level I/O – cont'd

- Basic functions
  - ssize_t read(int fd, void *buffer, size_t maxsize)
  - ssize_t write(int fd, const void *buffer, size_t size)
  - off_t lseek(int fd, off_t offset, int whence)

- Descriptions
  - read(): reads data from an open file using its file descriptor
    - Read **up to maxsize bytes**; returns less bytes if the data < maxsize
    - Return the number of bytes it read (0 means **EOF**, and negative values are errors)
  - write(): writes data to an open file using its file descriptor
    - Returns the number of bytes it wrote
  - lseek(): repositions the file offset within the kernel
    - (lseek != fseek) fseek holds a position in the FILE pointer

High-level I/O

- File as a stream
  - **Definition:** an unformatted sequence of bytes **with a position**
  - **Functions :**
    - FILE *fopen(const char *filename, const char *mode)
    - int fclose(FILE *fp)

  - **Standard streams:**
    - FILE *stdin   : normal source of input, can be redirected
    - FILE *stdout: normal source of output; redirection can be done
    - FILE *stderr : output errors

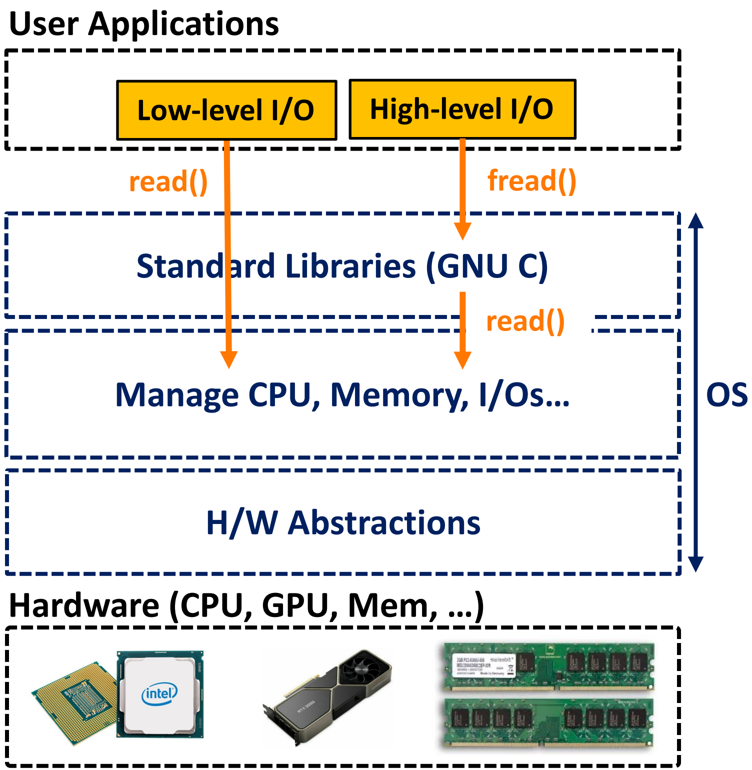
High-level I/O – cont'd

- Character(byte)-level API
  - int fputc(int c, FILE *fp)
  - int fputs(const char *s, FILE *fp)
  - int fgetc(FILE *fp)
  - char *fgets(char *buf, int n, FILE *fp)

- Block-level API
  - size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *fp)
  - size_t fwrite(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *fp)

Low-level I/O vs. high-level I/O

- Low-level I/O uses system calls, while high-level I/Os are **not**
  - **System calls**
    - They directly request OS services/resources
    - e.g., open(), read(), write(), and close()
  - **Standard libraries in C**
    - They are offered by C libraries
    - C libraries eventually do system calls
    - e.g., fopen(), fread(), fwrite(), and fclose()



Outline

- Part I:
  - Process
  - Threads
  - Scheduling basics
- Part II:
  - Files and I/Os
  - File system basics
- Part III:
  - IPC
  - RPC
  - Networking
- Part IV:
  - Synchronization
  - Rust

IPC: Signals

- **Signals**
  - **Definition:**
    - (Formal) an asynchronous mechanism to notify an event to a process
    - (Informal) **notifications** between processes or a process and a thread

- **Signals in Linux**
  - 32 non-real-time signals (0 to 31)
  - 31 real-time signals (32 to _NSIG [link])

IPC: Signals

- **Signals**
  - **Definition:**
    - (Formal) an asynchronous mechanism to
    - (Informal) notifications between processes

- **Signals in Linux**
  - 32 non-real-time signals (0 to 31)
  - 31 real-time signals (32 to _NSIG [link])

- **Signals we might know**
  - SIGINT : To terminate (CTRL+C)
  - SIGKILL : To terminate immediately (kill -9)
  - SIGSEGV: If segmentation fault happens
  - …

#	Signal Name	Default Action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard, Control-C	Yes
3	SIGQUIT	Dump	Quit from keyboard, Control-\	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timer clock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated or got a signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

IPC: Signal internals

- Signal from Process A -> Process B
  - **OS kernel**
    - Checks if Process B has pending signals
    - Pauses the execution of Process B
    - Invokes do_signal()
    - do_signal() call invokes handle_signal()
  - **Process B**
    - Run code in signal_handler
    - Return back to kernel: sigreturn()
  - **OS Kernel**
    - Resume Process B

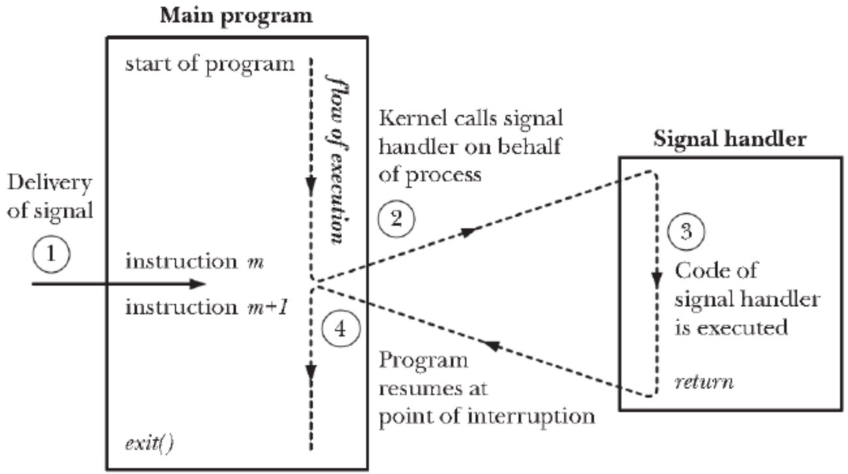


Figure 20-1: Signal delivery and handler execution

IPC: Pipes

- **PIPE:**
  - **Definition:** a unidirectional data channel, used for inter-process communication
  - **Conceptually:**
    - A file shared between two process (only one can write, and the other can only read)
    - Note: a file descriptor can be shared (**aliased**) between two process
      - **To write:** write(**writefd**, wbuf, wlen);
      - **To read :** read(**readfd**, rbuf, rmax);

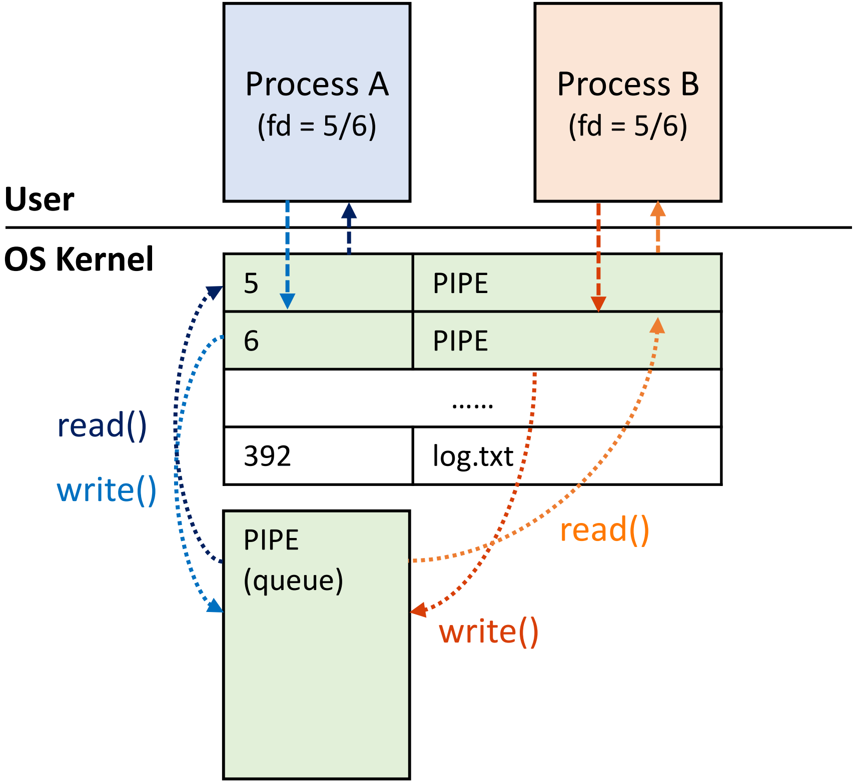
IPC: Pipe – cont'd

- Data structure
  - **Queue** in memory
  - **(Rule)** If Proc A writes data, the data will be in the kernel queue until Proc B reads it

- OS kernel's queue control:
  - Queue can be **full/empty**
    - If the queue is full, OS kernel asks Proc A (write) to wait
    - If the queue is empty, OS kernel asks Proc B (read) to wait

IPC: PIPE

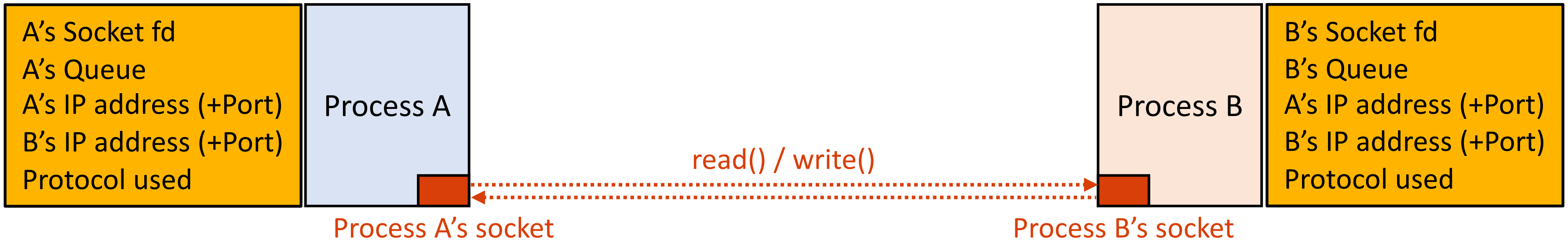
- PIPE between two processes
  - Process A creates a pipe (fd=5/6)
  - A can read/write with the pipe
  - Process A fork()
  - Process B is created (a child)
  - Process B can read/write from (fd=5/6)



RPC: Socket

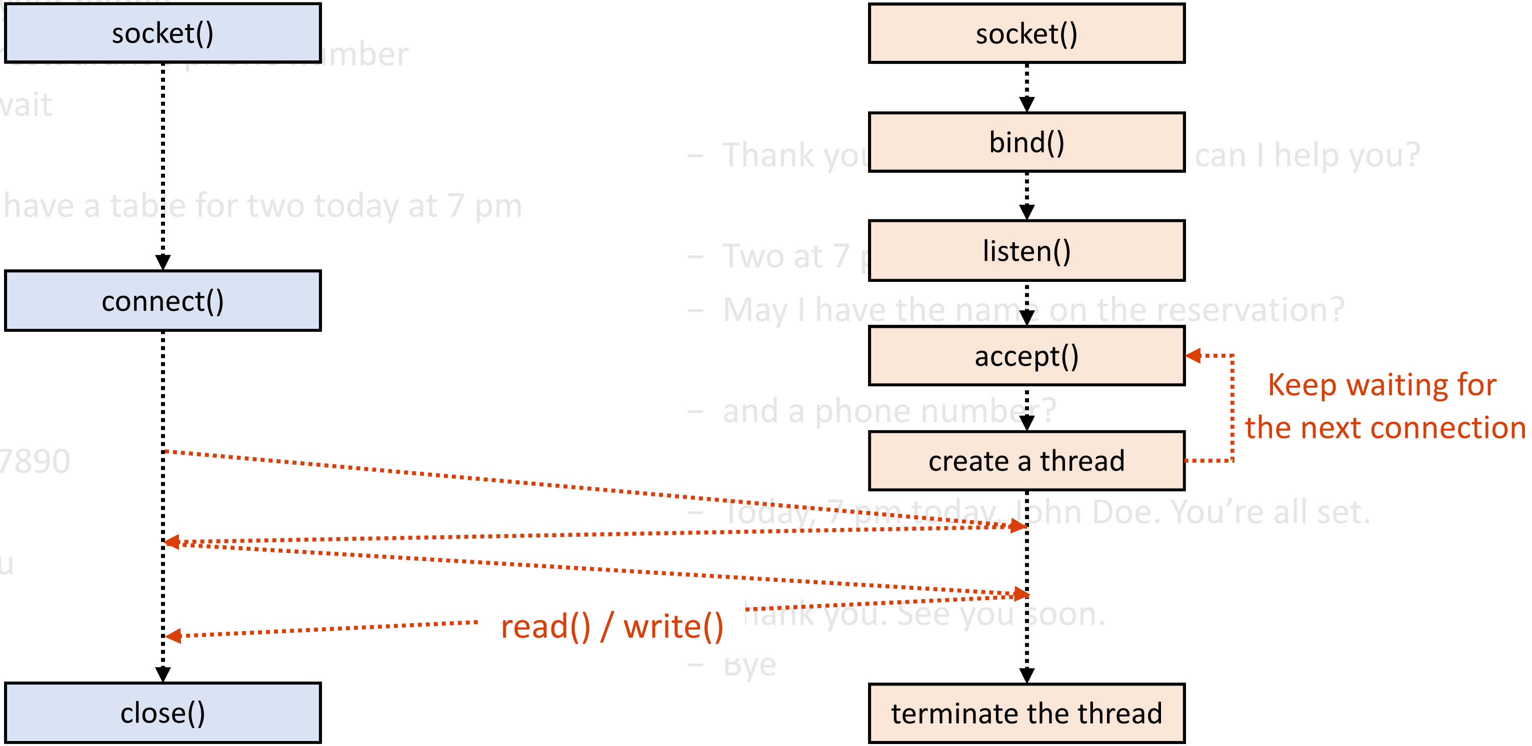
- Socket
  - **Definition:** an *abstract* structure for sending and receiving data
  - **TL; DR:** a *bi-directional* pipe

- Socket components
  - A structure (① a file descriptor and ② a queue)
  - IP addresses (③ source and ④ destination addresses)
  - ⑤ Protocols (*e.g.*, TCP/IP or UDP) to use



RPC: Socket – procedure

- **Caller (Client)**
- **Callee (Server)**



RPC: Socket – server.c

Bind the socket to the address
> Any IP (of the host)
> Port # 8080

AF_INET (IPv4)
SOCK_STREAM (bi-directional)

SO_REUSEADDR
SO_REUSEPORT
opt (optional value)

Listen incoming connections
> Use the socket fd
> Allow 3 connections (max.)

Start accepting connections
> Use the socket fd
> Use the address specified
> Return the fd (accepted)

```
... omit the includes

#define  BUF_SIZE        1024
#define  PORT            8080

int main(void) {
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
   char buffer[BUF_SIZE] = { 0 };
   char* hello = "Hello (server)!";

// create socket (returns a sockfd for reading/writing)
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
   perror("socket failed");
   exit(EXIT_FAILURE);
}

// configure the socket by setting the options
if (setsockopt(server_fd, SOL_SOCKET,
                  SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
   perror("setsocketopt failed");
   exit(EXIT_FAILURE);
}

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;  // bind to any address
address.sin_port = htons(PORT);         // format the port num

// attach socket to the port 8080
if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
   perror("bind failed");
   exit(EXIT_FAILURE);
}

if (listen(server_fd, 3) < 0) {
   perror("listen failed");
   exit(EXIT_FAILURE);
}

if ((new_socket = accept(server_fd,
                          (struct sockaddr*)&address,
                          (socklen_t*)&sizeof(address))) < 0) {
   perror("accept");
   exit(EXIT_FAILURE);
}

valread = read(new_socket, buffer, 1024);
printf("%s\n", buffer);
send(new_socket, hello, strlen(hello), 0);
printf("Message sent (server)\n");
return 0;
}
```

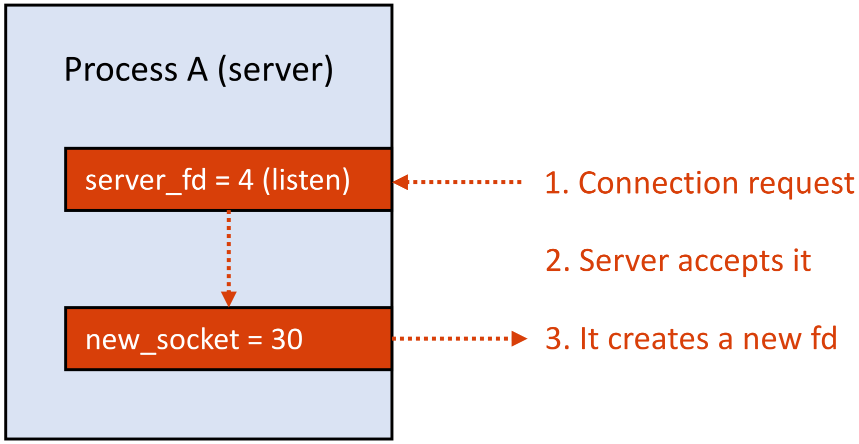
RPC: Socket – server.c

Bind the socket to the address
> Any IP (of the host)
> Port # 8080

```
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;  // bind to any address
address.sin_port = htons(PORT);        // format the port num

// attach socket to the port 8080
if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
   perror("bind failed");
   exit(EXIT_FAILURE);
}

if (listen(server_fd, 3) < 0) {
   perror("listen failed");
   exit(EXIT_FAILURE);
}

if ((new_socket = accept(server_fd,
                         (struct sockaddr*)&address,
                         (socklen_t*)&sizeof(address))) < 0) {
   perror("accept");
   exit(EXIT_FAILURE);
}

valread = read(new_socket, buffer, 1024);
printf("%s\n", buffer);
send(new_socket, hello, strlen(hello), 0);
printf("Message sent (server)\n");
return 0;
}
```

Listen incoming connections
> Use the socket fd
> Allow 3 connections (max.)

Start accepting connections
> Use the socket fd
> Use the address specified
> Return the fd (accepted)



socket fd != new_socket

Design choice:
We want to *separate* the file descriptor for listening connection requests (socket_fd) from the file descriptor used for communicating with the client (new_socket)

RPC: Socket – client.c

```c
#define  IPADDR  "127.0.0.1"
#define  PORT     8080
#define  BUFSIZE 1024

int main(void)
{
   int sock = 0, valread;
   struct sockaddr_in serv_addr;
   char* hello = "Hello (client)";
   char buffer[BUFSIZE] = { 0 };

   // create a socket
   if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
      printf("Error: socket creation error\n");
      return -1;
   }

   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(PORT);

   // convert IP addresses from text to binary
   if (inet_pton(AF_INET, IPADDR, &serv_addr.sin_addr) <= 0) {
      printf("Error: invalid address, address not supported\n");
      return -1;
   }

   if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
      printf("Connection Failed\n");
      return -1;
   }

   send(sock, hello, strlen(hello), 0);
   printf("Message sent (client)\n");
   valread = read(sock, buffer, BUFSIZE);
   printf("%s\n", buffer);

   return 0;
}
```

AF_INET (IPv4)
SOCK_STREAM (bi-directional)

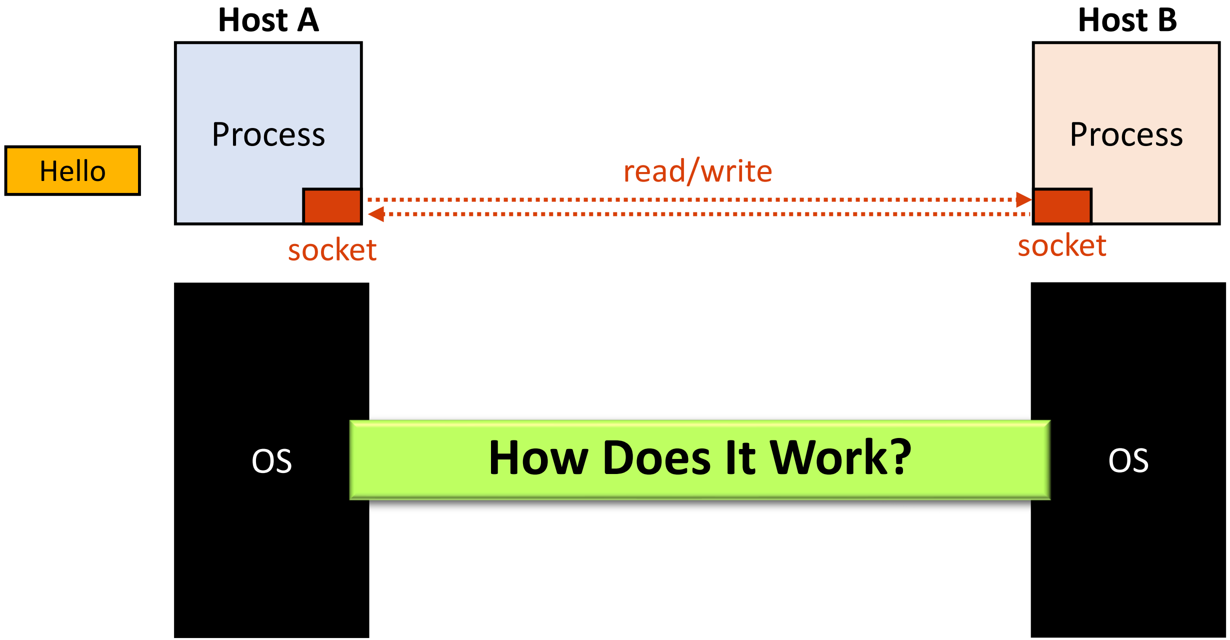
Connect to the server, running on the IP address we specify "127.0.0.1"

Execution result

```
$ gcc -o server server.c
$ gcc -o client client.c
$ ./server &
$ ./client

Message sent (client)
Hello (client)
Message sent (server)
Hello (server)
```

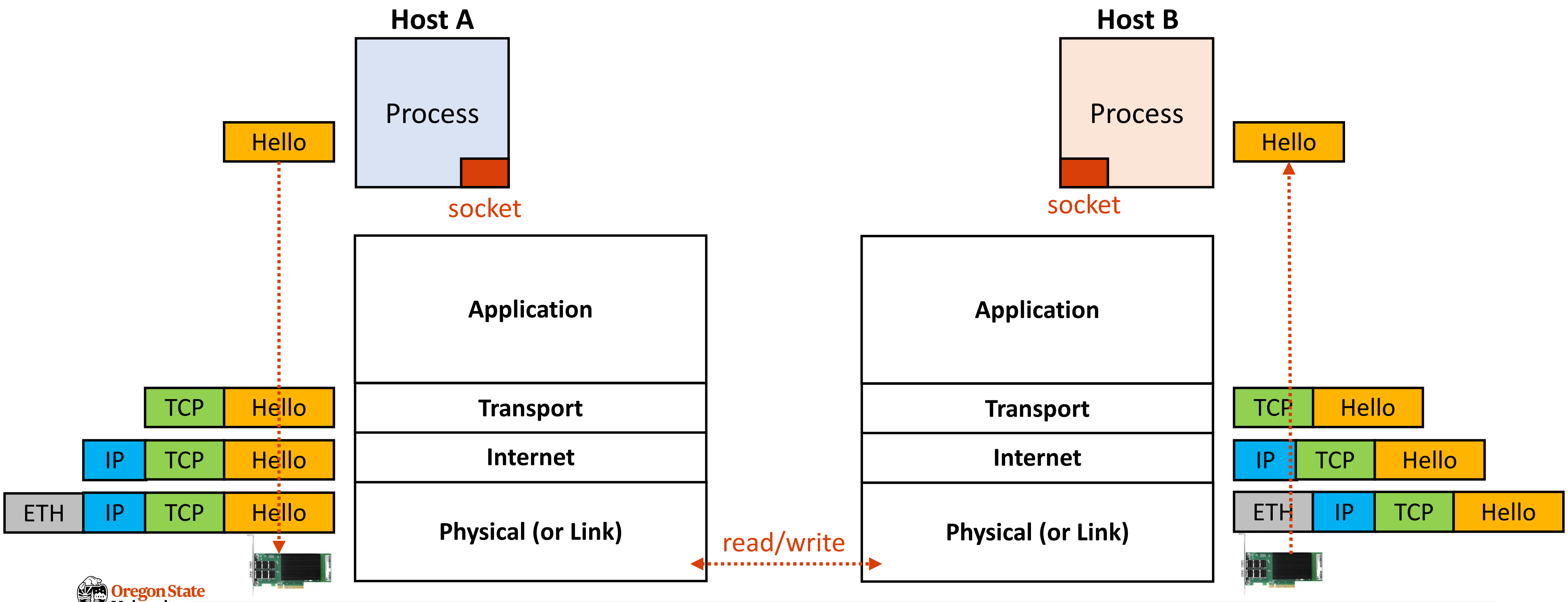
(Computer) Networking

- Networking
  - **Definition:** two or more applications on different computers (hosts) exchanging data



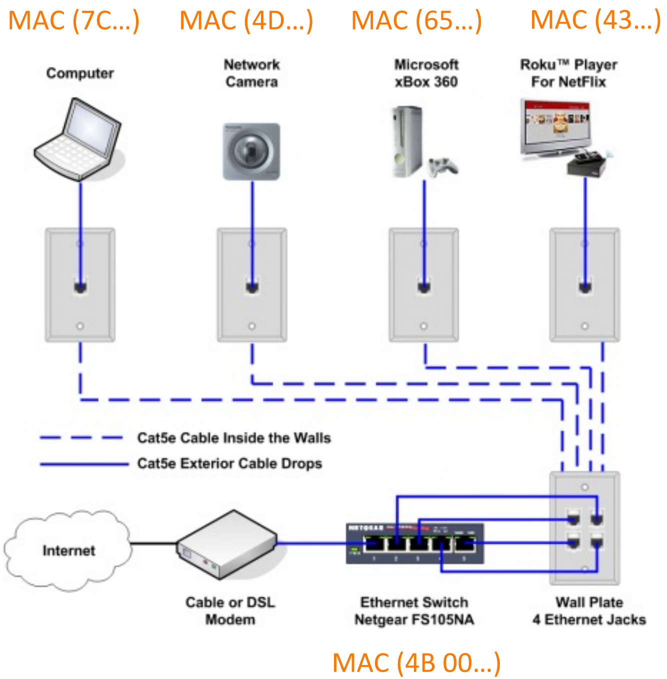
Networking: packet encapsulation

- In the TCP/IP 4-layer model



Networking: Ethernet (Physical Layer)

- Ethernet Protocol (~80s)
  - Each network device (NIC) has 48-bit **MAC address**
  - Each NIC is connected via Ethernet **cable**
  - ETH **header** contains:
    - (64 bit) Preamble (0x111111111… or a unique data)
    - (48-bit) Destination MAC address
    - (48-bit) Source MAC address
    - (16-bit) Type
    - (up to 1500 bytes) Data
    - (32-bit) CRC for error correcting



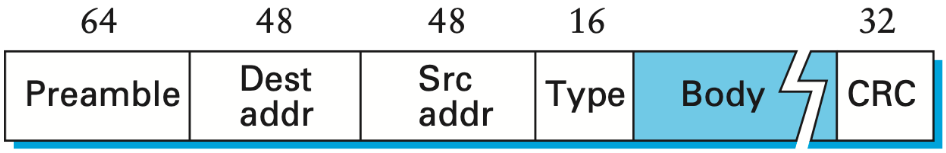
ETH	IP	TCP	Hello

Physical (or Link) ← read/write → **Physical (or Link)**

ETH	IP	TCP	Hello

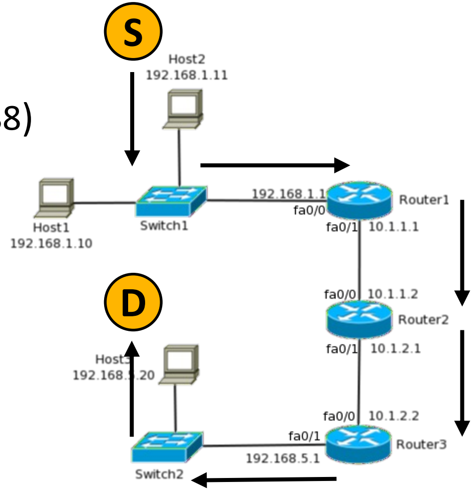
Networking: Ethernet (Physical Layer)

- Ethernet Protocol (~80s)
  - Each network device (NIC) has 48-bit **MAC address**
  - Each NIC is connected via Ethernet **cable**
  - ETH **header** contains:
    - (64 bit) Preamble (0x111111111… or a unique data)
    - (48-bit) Destination MAC address
    - (48-bit) Source MAC address
    - (16-bit) Type
    - (up to 1500 bytes) Data
    - (32-bit) CRC for error correcting



Networking: IP layer

- Internet Protocol (IP)
  - IP allows us to connect multiple networks
  - Each host has a unique IP address
    - **IPv4**: 32-bit address (e.g., 147.56.28.101)
    - **IPv6**: 128-bit address (e.g., 2001:db8:3333:4444:5555:6666:7777:8888)
  - IP data (packets) is **routed** based on **destination IP**



IP	TCP	Hello

Internet
Physical (or Link)

read/write

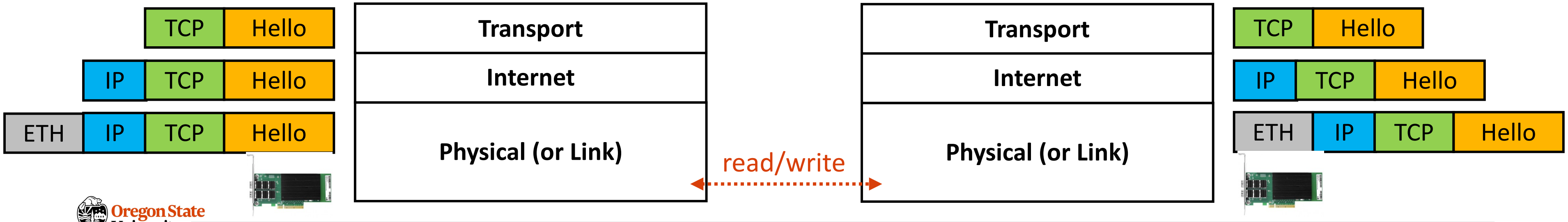
Internet
Physical (or Link)

IP	TCP	Hello

Oregon State University

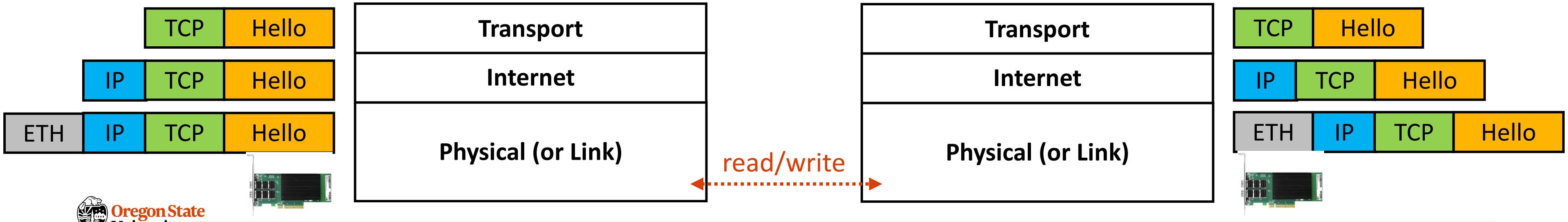
Networking: transport layer

- TCP vs UDP **Protocol**
  - Transmission Control Protocol: **TCP** Packet
    - (16-bit, for each) Source and destination ports
    - (32-bit) Sequence number
    - (32-bit) Acknowledgement number
    - Others: flags, checksums, window-size, pointer, …
  - User Datagram Protocol: **UDP** Packet
    - (16-bit, for each) Source and destination port
    - (16-bit, for each) Length and checksum



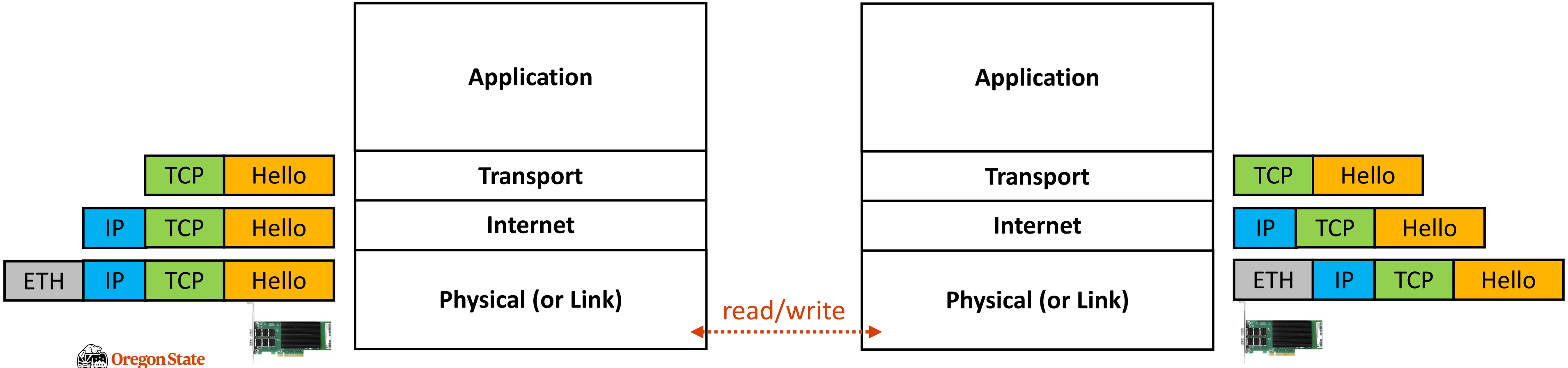
Networking: transport layer

- TCP vs UDP **Protocol**
  - TCP requires an established connection, but UDP is not (broadcast)
  - TCP can use sequences, but UDP is not
  - TCP is like a PIPE; data won’t be lost, but UDP will (can lose data)
  - TCP guarantees delivery, but UDP does not
  - TCP is slower than UDP (suppose that we deliver all the packets)



Networking: Application Layer

- Application layer
  - Support various user-defined or OS-defined protocols (on top of TCP/UDP)
  - **TCP-based :** HTTPS, HTTP, SMTP, POP, FTP, …
  - **UDP-based:** Video streaming, conferencing, DNS, VoIP, …



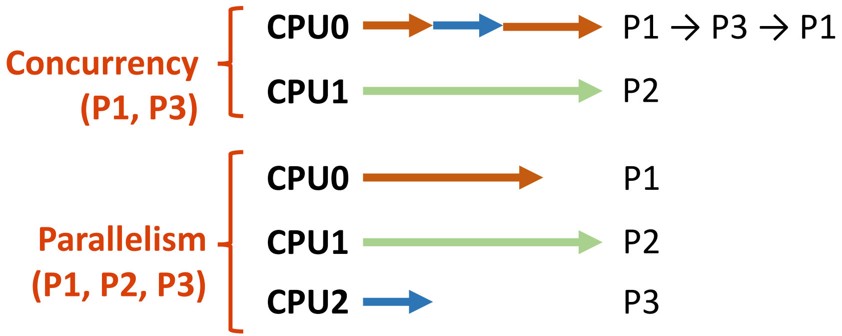
Outline

- Part I:
  - Process
  - Threads
  - Scheduling basics
- Part II:
  - Files and I/Os
  - File system basics
- Part III:
  - IPC
  - RPC
  - Networking
- Part IV:
  - Synchronization
  - Rust

Synchronization: terminology

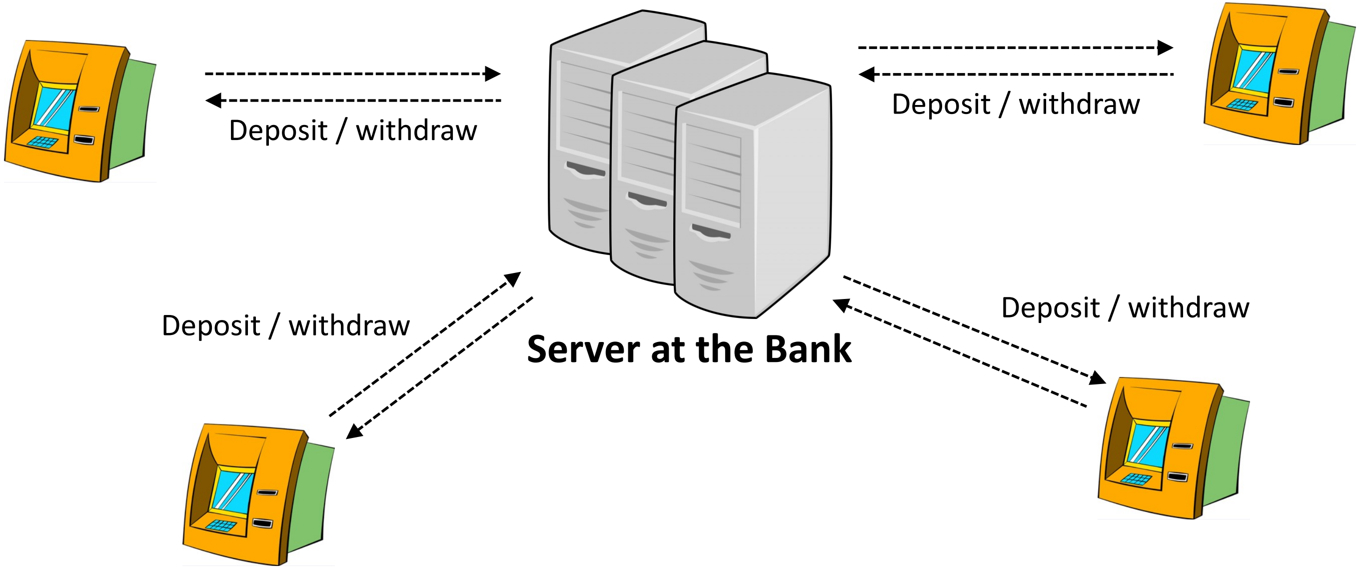
- **Concurrency vs. parallelism:**
  - Concurrency: handling multiple processes (or threads) at once
  - Parallelism: running multiple processes (or threads) *simultaneously*

- **Example:**
  - On the CPU0
    - P1 and P3 can execute *concurrently*
    - P1 and P3 is *not* running in parallel
  - On the CPU0 and CPU1
    - P1 and P2 runs in parallel



Synchronization: ATM bank server problem

- **ATM bank's server**
  - The server(s) takes care of multiple deposit / withdrawal requests
  - Bank want to make sure all the transactions are correct



Synchronization: concurrent ATM bank server in C

- **Threaded ATM bank server**
  - Receive a request
  - Create a thread for processing it
  - Multiple threads can co-exist

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;         // code only reaches here if the server terminates
}
```

Synchronization: Race Condition

- **Race condition:**
  - **Definition:** an undesirable scenario; performs multiple operations on a shared resource
  - **Example:** two "deposit" threads, running *concurrently*, increase the balance

How Can We Make Sure My Balance Is $700 at the End?

Synchronization: atomic operation

- **Solution approach:**
  - Deposit() is *indivisible*
  - Make sure to execute “Deposit()” at once

- **Atomic operation:**
  - Code should be executed w/o interrupt
  - **TL; DR:** Code should be run *at once*

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

Synchronization: Mutual Exclusion (Mutex)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         …
   }
}

void Deposit(accountId, amount) {
   pthread_mutex_lock(&foo_mutex);         // lock before the atomic op.
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
   pthread_mutex_unlock(&foo_mutex);       // unlock after the atomic op.
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;
   pthread_mutex_init(&deposit_lock, NULL);

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
```

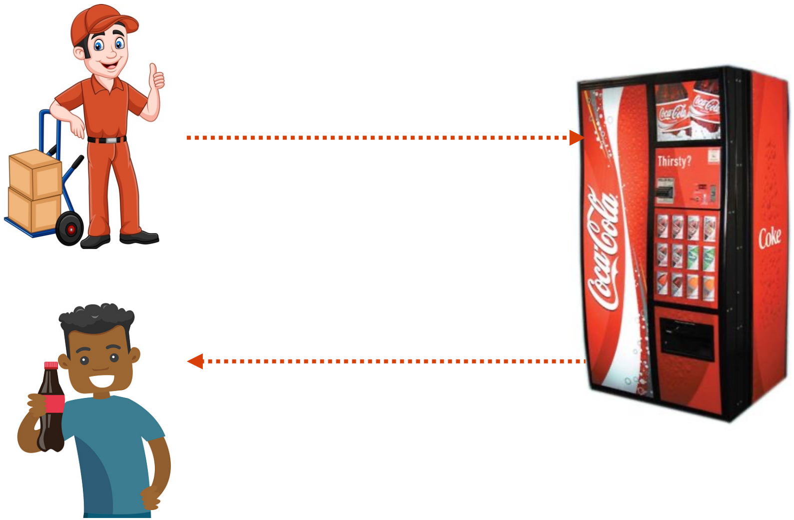
Synchronization: Mutual Exclusion (Mutex)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code
- Critical section
  - A code section protected by lock & unlock

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      …
  }
}

void Deposit(accountId, amount) {
  pthread_mutex_lock(&foo_mutex);       // lock before the atomic op.
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
  pthread_mutex_unlock(&foo_mutex);     // unlock after the atomic op.
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;
  pthread_mutex_init(&deposit_lock, NULL);

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;        // code only reaches here if the server terminates
```

SYNCHRONIZATION PROBLEM: A COKE MACHINE

- **A coke machine**
  - Two workers (or threads):
    - Producer: fills the coke machine
    - Consumer: takes cokes from the machine



Synchronization problem: a coke machine w. mutex

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke
- **Problem:**
  - Producer/consumer can wait forever
  - “Busy-wait” does *not* guarantee running

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      while (machine == full) {};
      pthread_mutex_lock(&machine);
      enqueue(acoke, coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

void consumer_fn() {
   while (1) {
      while (machine == empty) {};
      pthread_mutex_lock(&machine);
      acoke = dequeue(coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   ….

   return 0;                    // code only reaches here if the machine is broken
}
```

Synchronization: semaphore

- Semaphore
  - **Definition:** a variable used to control access to a shared resource
  - **TL; DR:** Mutex + Variable + Signal

- Semaphore operations
  - **P():** *wait* until a semaphore becomes positive and *decrease* it by 1
  - **V():** *increase* a semaphore by 1 and *wake up* any thread that waits by P()

Synchronization: Coke Machine

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke
- **Solution:**
  - Use semaphore
  - P() is sem_wait()
  - V() is sem_post()

```
sem_t mutex;
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
    while (1) {
        sem_wait(&slots_empty);
        sem_wait(&mutex);
        enqueue(acoke, coke_machine);
        sem_post(&mutex);
        sem_post(&slots_filled);
    }
}

void consumer_fn() {
    while (1) {
        sem_wait(&slots_filled);
        sem_wait(&mutex);
        acoke = dequeue(coke_machine);
        sem_post(&mutex);
        sem_post(&slots_empty);
    }
}

int main(void) {
    int ret;
    ret = sem_init(&mutex, 0, 1);
    ret = sem_init(&slots_empty, 0, 64);
    ret = sem_init(&slots_filled, 0, 0);
    ....

}
```

The semaphore only allows one thread to enqueue (or dequeue)

It decreases "filled slot" by one

It increases "empty slot" by one, and wakes up any thread *(i.e., producer thread)* by sending a *signal* to that thread

Initialize with the # resources
1) Mutex := lock := 1
2) Empty slots := 64 (capacity)
3) Filled slots := 0 (empty at first)

SYNCHRONIZATION: A COKE MACHINE

- **Example scenario**
  - Initially the coke machine is empty
  - Consumer tries to get a coke
    - It decreases "slots_filled" by one
    - "slots_filled" becomes -1
    - The thread sleeps
  - Producer runs
    - It decreases "slots_empty" by one
    - It adds a coke to the machine
    - It signals the thread waiting by "slots_filled"
  - Consumer wakes up and run

```
sem_t mutex;
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
  while (1) {
    sem_wait(&slots_empty);
    sem_wait(&mutex);
    enqueue(acoke, coke_machine);
    sem_post(&mutex);
    sem_post(&slots_filled);
  }
}

void consumer_fn() {
  while (1) {
    sem_wait(&slots_filled);
    sem_wait(&mutex);
    acoke = dequeue(coke_machine);
    sem_post(&mutex);
    sem_post(&slots_empty);
  }
}

int main(void) {
  int ret;
  ret = sem_init(&mutex, 0, 1);
  ret = sem_init(&slots_empty, 0, 64);
  ret = sem_init(&slots_filled, 0, 0);
  ....

}
```

The semaphore only allows one thread to enqueue (or dequeue)

It decreases "filled slot" by one

It increases "empty slot" by one, and wakes up any thread *(i.e., producer thread)* by sending a *signal* to that thread

Initialize with the # resources
1) Mutex := lock := 1
2) Empty slots := 64 (capacity)
3) Filled slots := 0 (empty at first)

Monitor

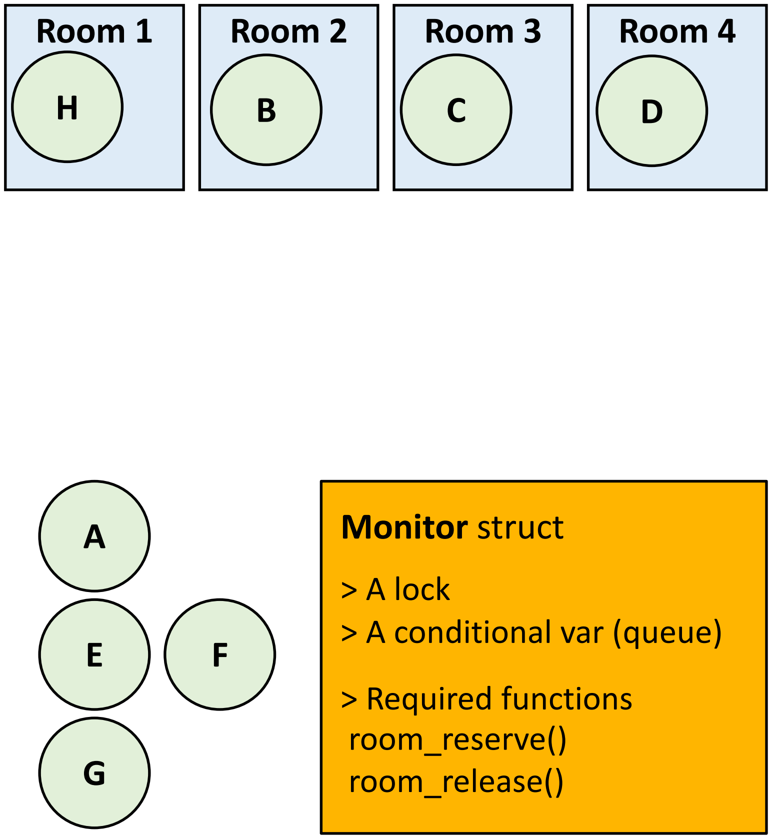
- Monitor:
  - **Def:** a synchronization *object*
    - Conditional variable
    - Monitoring mechanism

- Supported operations:
  - wait(&lock): release lock and sleep
  - signal(): wake up one waiting worker
  - broadcast(): wake up *all* waiting jobs

Monitor

- Monitor:
  - **Def:** a synchronization *object*
    - Conditional variable
    - Monitoring mechanism

- Supported operations:
  - wait(&lock): release lock and sleep
  - signal(): wake up one waiting worker
  - broadcast(): wake up *all* waiting jobs



Monitor in C

- Monitor:
  - **Def:** a synchronization *object*
    - Conditional variable
    - Monitoring mechanism

- Supported operations:
  - wait(&lock): release lock and sleep
  - signal(): wake up one waiting worker
  - broadcast(): wake up *all* waiting jobs

monitor.h

```
#ifndef  MONITOR_H
#define  MONITOR_H

#define  NUM_ROOMS    4

void reserve_a_room(int room_num, struct user_t* employee);
struct user_t* release_a_room(int room_num);

#endif
```

monitor.c

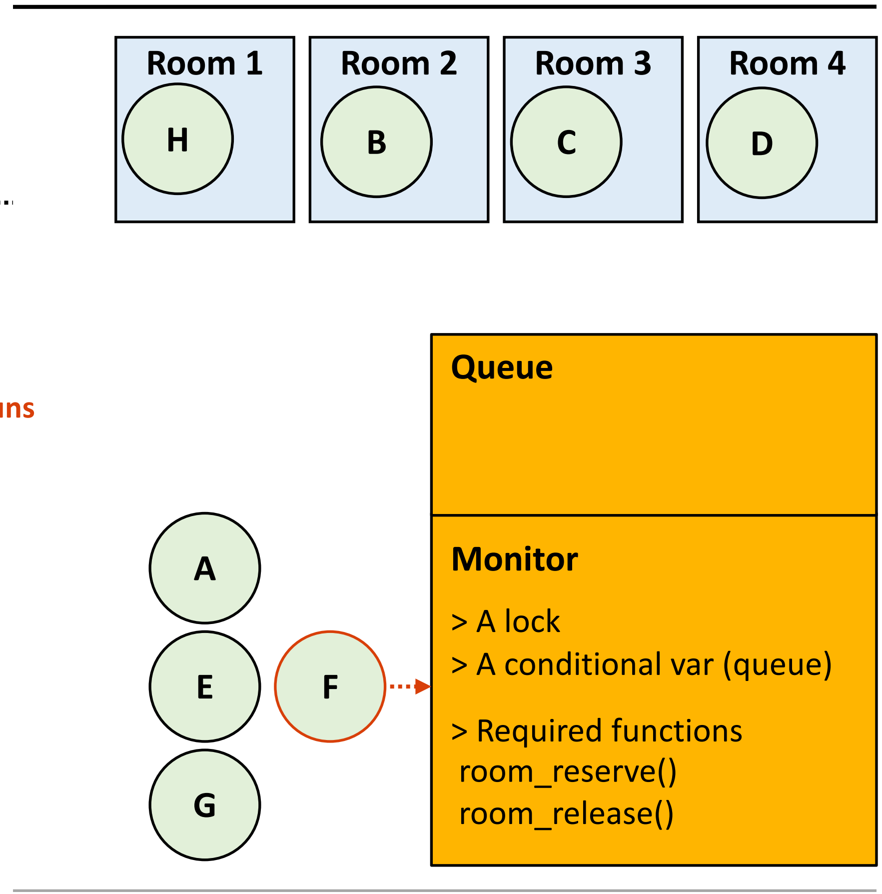
```
static lock monitor_lock;                          // lock
static struct queue wait_queue;                    // conditional variable
static struct room_t meeting_rooms[4];

void reserve_a_room(int room_num, struct user_t* employee) {
  acquire(&monitor_lock);
  while (meeting_rooms[room_num] != empty) {
    wait(&wait_queue, &monitor_lock);              // wait + unlock + sleep
  }
  room_assign(room_num, employee);
  release(&monitor_lock);
}

struct user_t* release_a_room(int room_num) {
  acquire(&monitor_lock);
  employee = room_empty(room_num);
  signal(&wait_queue);                             // wake up one of them
  release(&monitor_lock);
  return employee;
}
```

monitor.h

```
#ifndef  MONITOR_H
#define  MONITOR_H

#define  NUM_ROOMS     4

void reserve_a_room(int room_num, struct user_t* employee);
struct user_t* release_a_room(int room_num);

#endif
```

monitor.c

```
static lock monitor_lock;                    // lock
static struct queue wait_queue;              // conditional variable
static struct room_t meeting_rooms[4];

void reserve_a_room(int room_num, struct user_t* employee) {        <······· Runs
   acquire(&monitor_lock);
   while (meeting_rooms[room_num] != empty) {
      wait(&wait_queue, &monitor_lock);      // wait + unlock + sleep
   }
   room_assign(room_num, employee);
   release(&monitor_lock);
}

struct user_t* release_a_room(int room_num) {
   acquire(&monitor_lock);
   employee = room_empty(room_num);
   signal(&wait_queue);                      // wake up one of them
   release(&monitor_lock);
   return employee;
}
```

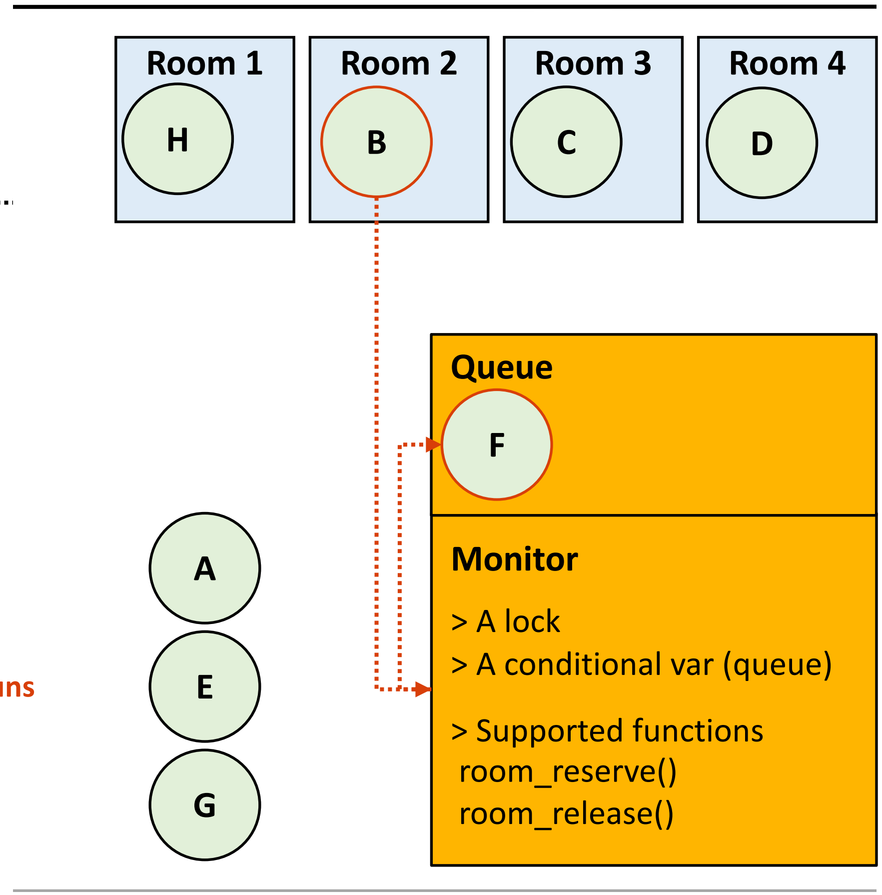


monitor.h

```c
#ifndef  MONITOR_H
#define  MONITOR_H

#define  NUM_ROOMS     4

void reserve_a_room(int room_num, struct user_t* employee);
struct user_t* release_a_room(int room_num);

#endif
```

monitor.c

```c
static lock monitor_lock;                    // lock
static struct queue wait_queue;              // conditional variable
static struct room_t meeting_rooms[4];

void reserve_a_room(int room_num, struct user_t* employee) {
   acquire(&monitor_lock);
   while (meeting_rooms[room_num] != empty) {
     wait(&wait_queue, &monitor_lock);       // wait + unlock + sleep
   }
   room_assign(room_num, employee);
   release(&monitor_lock);
}

struct user_t* release_a_room(int room_num) {        <······· Runs
   acquire(&monitor_lock);
   employee = room_empty(room_num);
   signal(&wait_queue);                      // wake up one of them
   release(&monitor_lock);
   return employee;
}
```



Rust: A trade off between control and safety


```c
…
#define  BUFSIZE             20

int main(void) {
   char *buf;
   char *str = "Hello world!";

   // initialize the memory space
   buf = (char *) malloc( sizeof(char) * BUFSIZE );

   // copy the string to the buffer
   strncpy(buf, str, BUFSIZE);

   // print the string
   printf("Buffer contains: %s.\n", buf);

   return 0;
}
```

```python
…import

if __main__ == "__main__":
   buf = ""
   str = "Hello world!"

   // copy the string
   buf += str

   // print out it
   print ("{}".format(buf))
   # done.
```

Example:

- **C:** More control over mem. allocation, but less safe
- **Python:** Less control, but more safe

Rust!

- Rust
  - A programming language designed for (memory) safety and performance

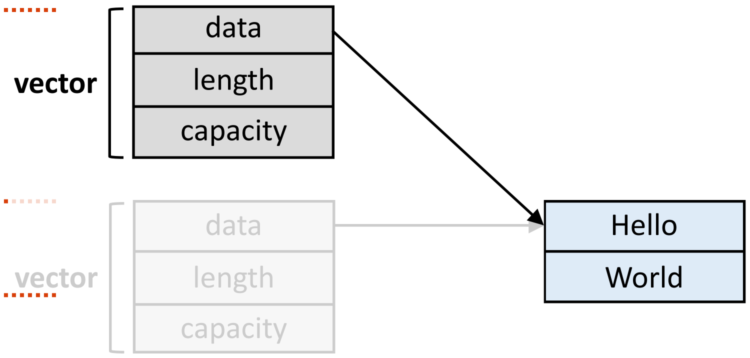
- Rust addresses
  - Runtime performance (unlike Python or Java, Rust does not use GC)
  - Memory leaks (no explicit allocation/de-allocation)
  - No data-race condition

- Rust concept
  - Ownership and borrowing
  - Concurrency
  - Unsafe code

Rust ownership

- Ownership
  - **Definition:** a set of rules how a Rust program manages memory
  - Rust rules:
    - Each value in Rust has a variable “owner”
    - There can be only one owner at a time
    - If the owner goes out of scope, the value will disappear
  - Ownership example:

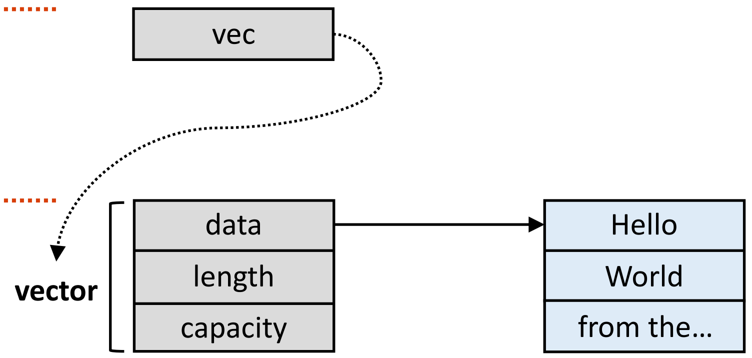
```
fn take(vec: Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    take(vec);

    vec.push(String::from("from the other side!"))
}
```



Rust borrowing

- Borrowing
  - **Definition:** a way to access data without taking ownership over it
  - Borrowing example:

```
fn borrow(vec: &Vec<String>){
   println!("{:?}", vec);
}

fn main() {
   let mut vec = Vec::new();
   vec.push(String::from("Hello "));
   vec.push(String::from("World "));
   borrow(&vec);

   vec.push(String::from("from the other side!"))
}
```



Rust concurrency

- Concurrency
  - Shared **read-only** accesses
  - Concurrency example:

Results:

```
$ ./main
Decrease the balance -100
Increase the balance 300
Final balance 200
```

Note:

"balance" is a read-only shared variable
"new_balance" only exists in each thread
No effect on the actual "balance" in main

```rust
use std::thread;

fn main() {
    let mut balance = 200;
    let mut threads = vec![];

    // deposit thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance += 100;
        println!("Increase the balance {}", new_balance);
    }));

    // withdrawal thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance -= 300;
        println!("Decrease the balance {}", new_balance);
    }));

    for thread in threads {
        let _ = thread.join();
    }
    println!("Final balance {}", balance);
}
```

Rust concurrency

- Concurrency
  - Shared **read-only** accesses
  - Shared **mutable** accesses
  - Concurrency example:

Mutable by threads:
- Mutex: mutable if we lock() the variable
- Arc : send-able to multiple threads

Deposit thread:
- Line 1: clone the Arc instance; point to the same.
- Line 2: lock and get the balance value
- Line 3: increase 100 (cf. access with *)

Withdrawal thread:
- The same as the deposit thread
- Decrease the balance by $300

```rust
use std::thread; use std::sync::{Arc,Mutex};

fn main() {
  let balance = Arc::new(Mutex::new(200));
  let mut threads = vec![];

  // deposit thread
  let balance4deposit = Arc::clone(&balance);
  threads.push(thread::spawn(move || {
    let mut new_balance = balance4deposit.lock().unwrap();
    *new_balance += 100;
    println!("Increase the balance {}", new_balance);
  }));

  // withdrawal thread
  let balance4withdrawal = Arc::clone(&balance);
  threads.push(thread::spawn(move || {
    let mut new_balance = balance4withdrawal.lock().unwrap();
    *new_balance -= 300;
    println!("Decrease the balance {}", new_balance);
  }));

  for thread in threads {
    let _ = thread.join();
  }

  println!("Final balance {}", *balance.lock().unwrap());
}
```

Rust concurrency

- Concurrency
  - Shared **read-only** accesses
  - Shared **mutable** accesses
  - Concurrency example:

Results:

$ ./main
Increase the balance 300
Decrease the balance 0
Final balance 0

Note:

"balance" is a mutable shared variable
"new_balance" points to the mutable variable
Require to wrap with Arc for sending to threads
Modify the value is only available after lock()

```
use std::thread; use std::sync::{Arc,Mutex};

fn main() {
   let balance = Arc::new(Mutex::new(200));
   let mut threads = vec![];

   // deposit thread
   let balance4deposit = Arc::clone(&balance);
   threads.push(thread::spawn(move || {
      let mut new_balance = balance4deposit.lock().unwrap();
      *new_balance += 100;
      println!("Increase the balance {}", new_balance);
   }));

   // withdrawal thread
   let balance4withdrawal = Arc::clone(&balance);
   threads.push(thread::spawn(move || {
      let mut new_balance = balance4withdrawal.lock().unwrap();
      *new_balance -= 300;
      println!("Decrease the balance {}", new_balance);
   }));

   for thread in threads {
      let _ = thread.join();
   }

   println!("Final balance {}", *balance.lock().unwrap());
}
```

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Unsafe code in rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static variable:
- "anumber" can be accessible in any code in this file

Create 10 threads:
- Each thread prints the thread index and "anumber"

Results:

```
$ ./main
Thread 0: anumber is 10
Thread 4: anumber is 10
Thread 5: anumber is 10
Thread 2: anumber is 10
Thread 8: anumber is 10
…
```

```rust
use std::thread;

static anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      println!("Thread {}: anumber is {}", tidx, anumber);
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }
}
```

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static variable:
- "anumber" can be accessible in any code in this file

Create 10 threads:
- It will return a Rust **compilation error**
- Rust prevents us from directly modifying static mut
- Rust prohibits us from even just accessing it

```
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      println!("Thread {}: anumber is {}", tidx, anumber);
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static (mutable) variable:
- We want "anumber" can be **modified** in any code

Create 10 threads:
- Use "unsafe" keyword if we modify "anumber"
- "unsafe" means we understand the consequences
- Now each thread will increase "anumber" by 10

Print out the static mutable:
- Use "unsafe" even for just printing out

```
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      unsafe {
        anumber += 1;
        println!("Thread {}: anumber is {}", tidx, anumber);
      }
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }

  unsafe {
    println!("The final anumber is {}", anumber);
  }
}
```

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

A variable:
- "s" contains the address of the string "123"

A (pointer) variable:
- "ptr" is the pointer for the string "123"
- "ptr" is "constant" and the type of "u8"

Dereference the pointer values:
- "ptr.offset(#)" is the same as *(ptr + 1) in C
- "as char" converts the output of "ptr.offset" as char
- It causes a **compilation error (Rust prevents this)**

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    println!("{}", *ptr.offset(1) as char);
    println!("{}", *ptr.offset(2) as char);
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

A variable:
- "s" contains the address of the string "123"

A (pointer) variable:
- "ptr" is the pointer for the string "123"
- "ptr" is "constant" and the type of "u8"

Access the pointer values:
- Use "unsafe" to do the pointer arithmetic
- "unsafe" means we understand the consequences
- It causes a **compilation error (Rust prevents this)**

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    unsafe {
        println!("{}", *ptr.offset(1) as char);
        println!("{}", *ptr.offset(2) as char);
    }
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Access the out-of-bound values:
- "*ptr.offset(3)" accesses the 4th character [?!]

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    unsafe {
        println!("{}", *ptr.offset(1) as char);
        println!("{}", *ptr.offset(2) as char);
        println!("{}", *ptr.offset(3));
    }
}
```

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

An external function:
- The function "abs" is defined in C (not in Rust)

Use of the external function:
- A **compilation error** (cannot call "abs" *directly*)
- Not sure whether the abs implementation is safe

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

An external function:
- The function "abs" is defined in C (not in Rust)

Use of the external function:
- Use "unsafe" to call the "abs" function
- Not sure whether the abs implementation is safe

```rust
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

This term

- Part I:
  - Process
  - Threads
  - Scheduling basics
- Part II:
  - Files and I/Os
  - File system basics
- Part III:
  - IPC
  - RPC
  - Networking
- Part IV:
  - Synchronization
  - Rust

Thank You!

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab