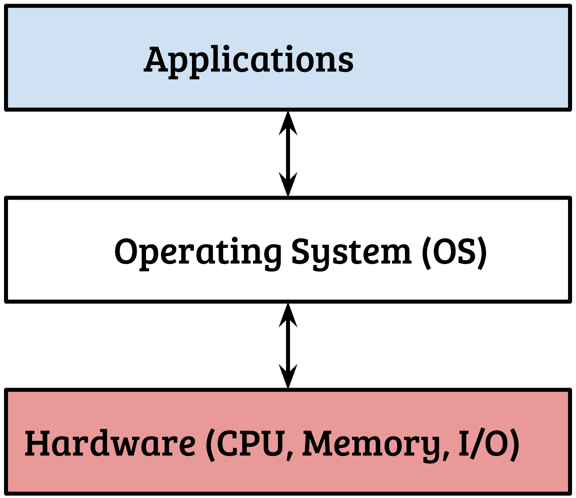
CS330: Operating Systems

Introduction

What is an Operating System?



- Operating system is a software layer between the hardware and the applications
- What are the functions of this middleware?
  - Why is this intermediate layer necessary?

What if we skip the OS layer?

Browser	Word processor	Your own application

User libraries

Logic
Programming (C, Python etc.)
Data structures and Algorithms

Can build applications

Can even build libraries

What if we skip the OS layer?

Browser

Word processor

Your own application

User libraries

Logic
Programming (C, Python etc.)
Data structures and Algorithms

Oh! Need a computer to show my skills.

Can build applications

Can even build libraries

What if we skip the OS layer?

Browser

Word processor

Your own application

User libraries

Logic
Programming (C, Python etc.)
Data structures and Algorithms

Oh! Need a computer to show my skills.

I know logic gates to ISA

Can build a small computer for my program!

What if we skip the OS layer?

Browser

Word processor

Your own application

User libraries

Logic
Programming (C, Python etc.)
Data structures and Algorithms

Oh! Need a computer to show my skills.

I know logic gates to ISA

Can build a small computer for my program!

Conclusion: do not need the OS!

What if we skip the OS layer?

Applications

Library API

(Will) know compilers to convert high-level language code to machine assembly

Hardware (CPU, Memory, I/O)

High-level programming language

I understand only assembly code.

What if we skip the OS layer?

Applications

Library API

High-level programming language

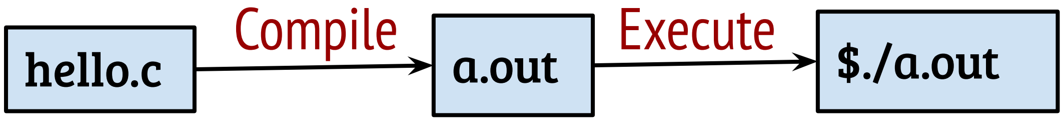
(Will) know compilers to convert high-level language code to machine assembly

I understand only assembly code.

Hardware (CPU, Memory, I/O)

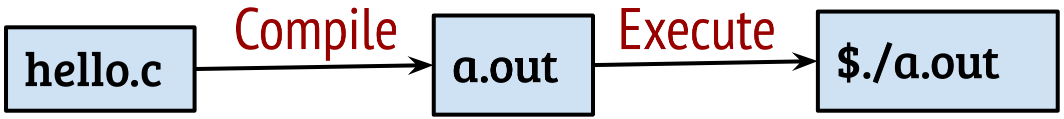
Conclusion: do not need the OS!

Program execution



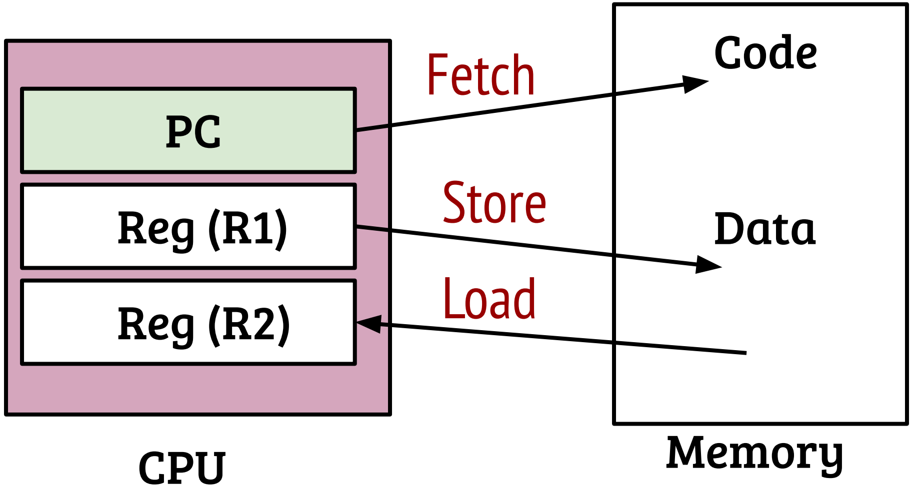
You said only CPU can execute!

Inside program execution



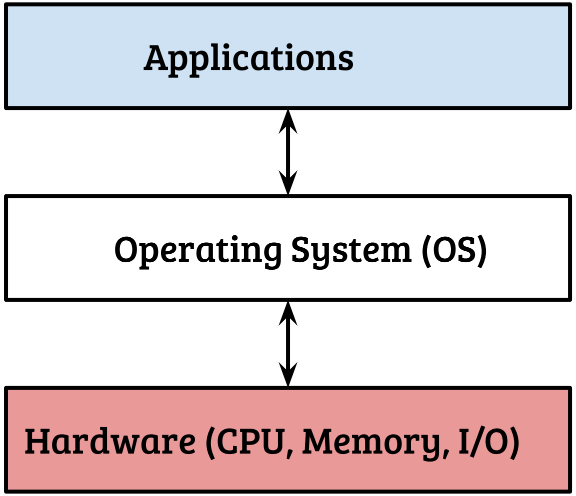
You said only CPU can execute!

CPU execution (from CS220)



- Loads instruction pointed to by PC
- Decode instruction
- Load operand into registers
- Execute instruction (ALU)
- Store results

What is an Operating System?



- OS bridges the *semantic gap* between the notions of application execution and real execution
  - OS loads an executable from disk to memory, allocates/frees memory dynamically
  - OS initializes the CPU state i.e., the PC and other registers
  - OS provides interfaces to access I/O devices
- OS facilitates hardware resource sharing and management

Resource virtualization

- OS provides virtual representation of physical resources
  - Easy to use abstractions with well defined interfaces
  - Examples:

Physical resource	Abstraction	Interfaces
CPU	Process	Create, Destroy, Stop etc.
Memory	Virtual memory	Allocate, Free, Permissions
Disk	File system tree	Create, Delete, Open, Close etc.

What is virtualization of resources?

- Definition [1] “Not physically existing as such but made by software to appear to do so.”
- By implication
  - OS multiplexes the physical resources
  - OS manages the physical resources
- Efficient management becomes more crucial with multitasking

1. Oxford dictionary : https://en.oxforddictionaries.com/definition/virtual

Design goals of OS abstractions

- Simple to use and flexible
- Minimize OS overheads
    - Any layer of indirection incurs certain overheads!
- Protection and isolation
- Configurable resource management policies
- Reliability and security

Next lecture: The process abstraction

CS330: Operating Systems

Process

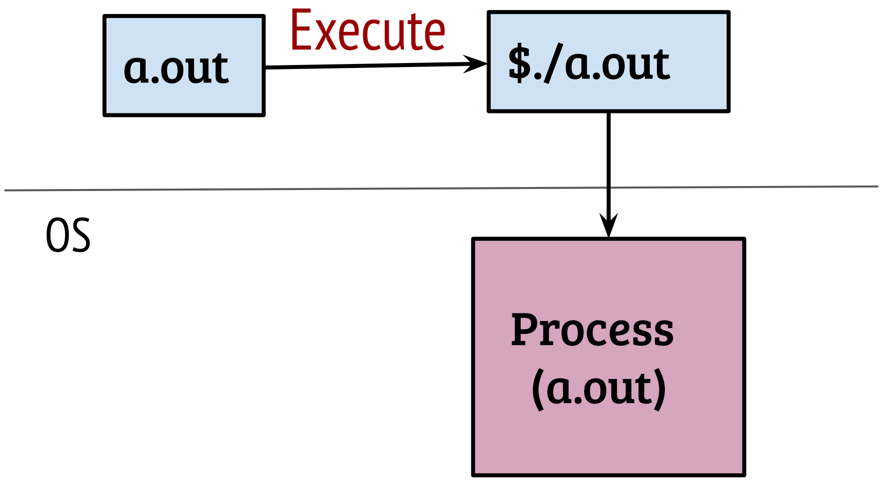
Recap

- OS bridges the *semantic gap* between the notions of application execution and real execution
- How?
  - By virtualizing the physical resources
  - Creating abstractions with well defined interfaces

- Today's agenda: CPU $\longrightarrow$ Process

The process abstraction

- The OS creates a process when we run an executable



- Process is represented by a data structure commonly known as process control block (PCB)
- Linux ⟶ task_struct
- gemOS ⟶ exec_context

The process abstraction

- The OS creates a process when we run an executable
- Alternatively, *A program in execution is called a process*

The process abstraction

- The OS creates a process when we run an executable
- Alternatively, *A program in execution is called a process*
- Program is persistent while process is volatile
    - Program is identified by an executable, process by a PID

The process abstraction

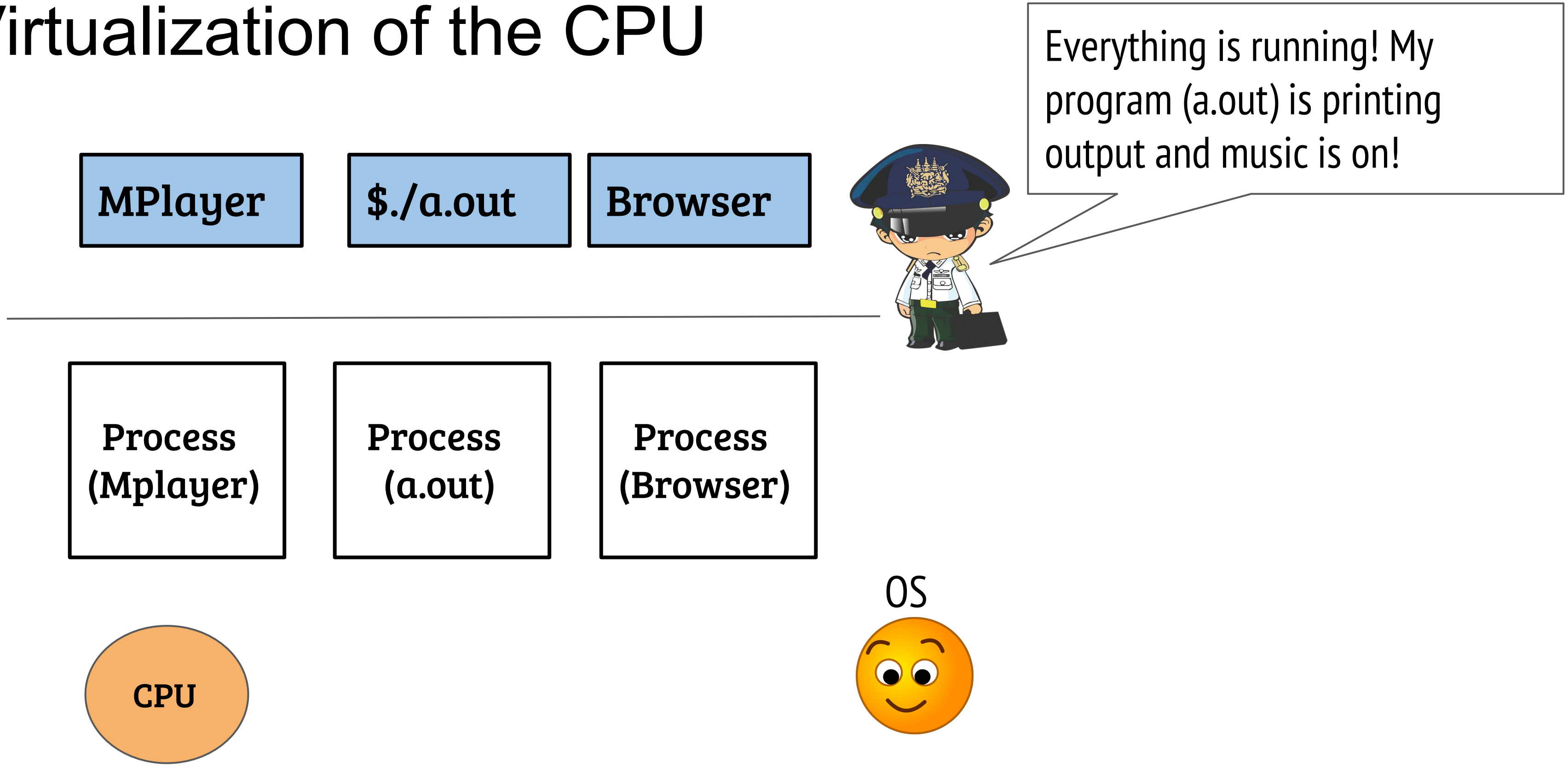
- The OS creates a process when we run an executable
- Alternatively, *A program in execution is called a process*
- Program is persistent while process is volatile
    - Program is identified by an executable, process by a PID
- Program ⟶ Process   (1 to N)
    - Many concurrent processes can execute the same program

The process abstraction

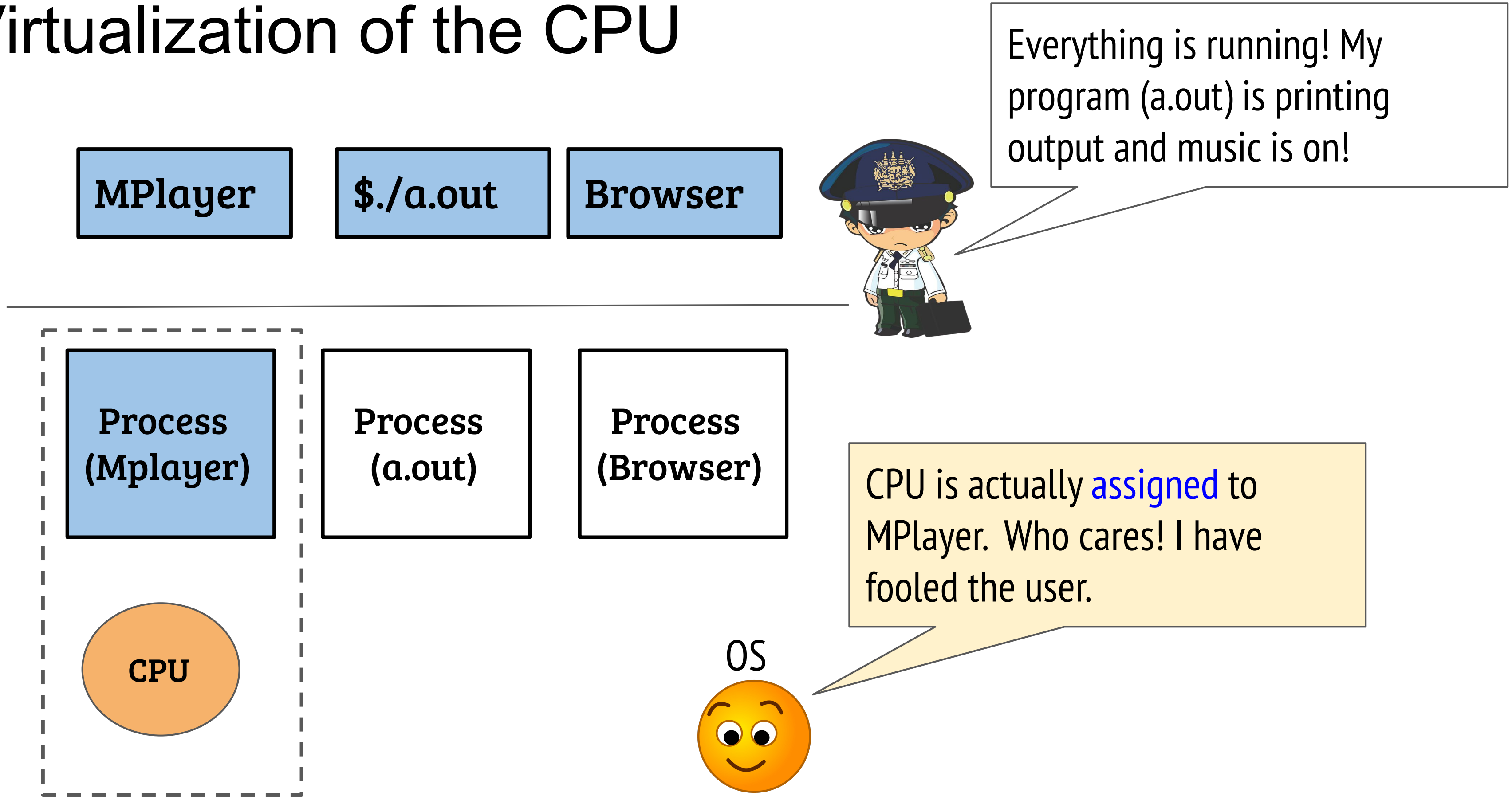
- The OS creates a process when we run an executable
- Alternatively, *A program in execution is called a process*
- Program is persistent while process is volatile
    - Program is identified by an executable, process by a PID
- Program $\longrightarrow$ Process (1 to N)
    - Many concurrent processes can execute the same program

What about virtualizing the CPU?

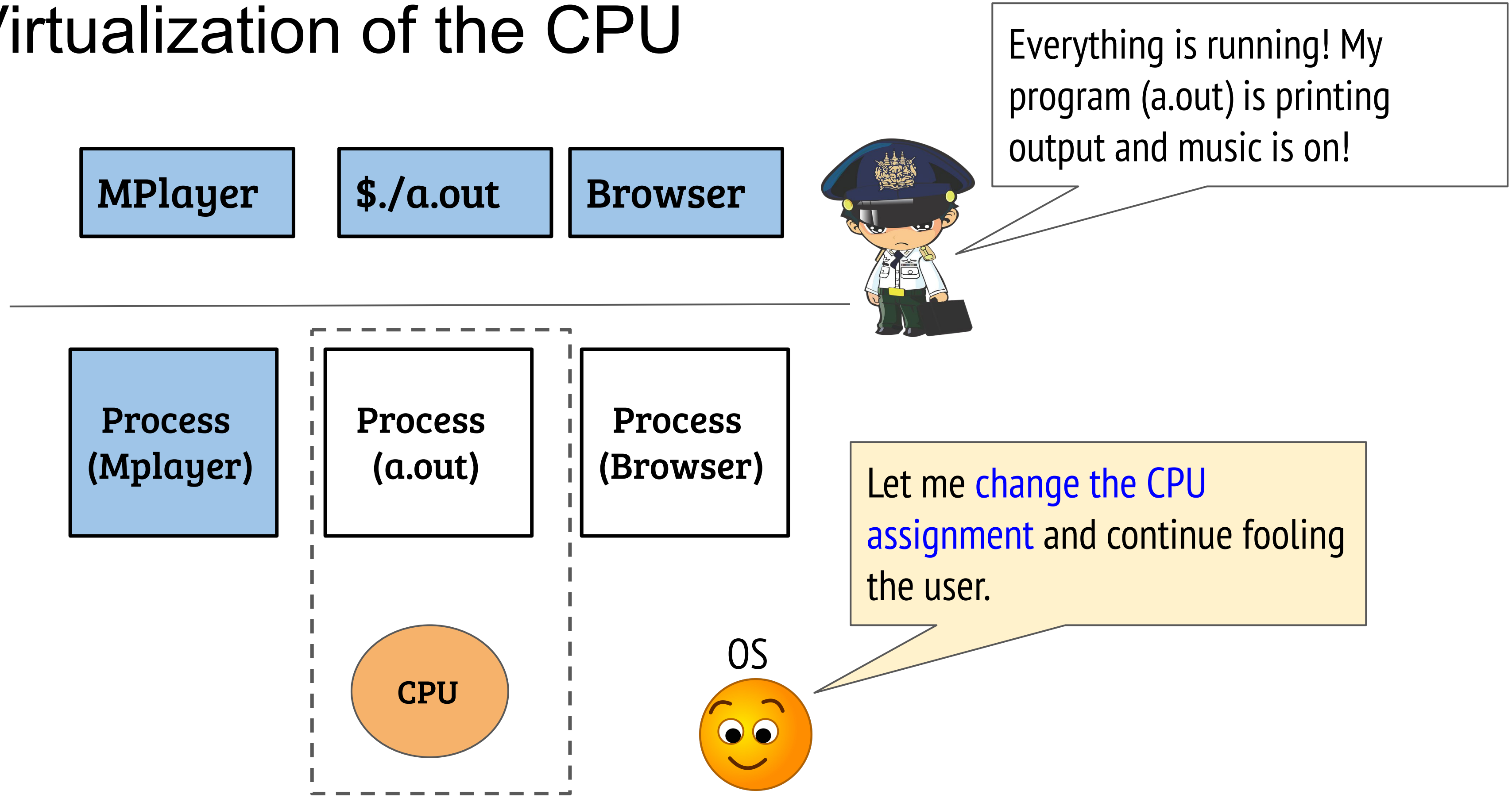
Virtualization of the CPU



Virtualization of the CPU



Virtualization of the CPU



Virtualization of the CPU

MPlayer

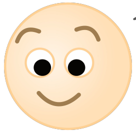
$./a.out

Browser

Everything is running! My program (a.out) is printing output and music is on!

- How CPU assignment is changed? (OR how context switch is performed?)
  - What happens to outgoing process? How does it come back?
- Overheads of context switch?
- How to decide the incoming process?

CPU



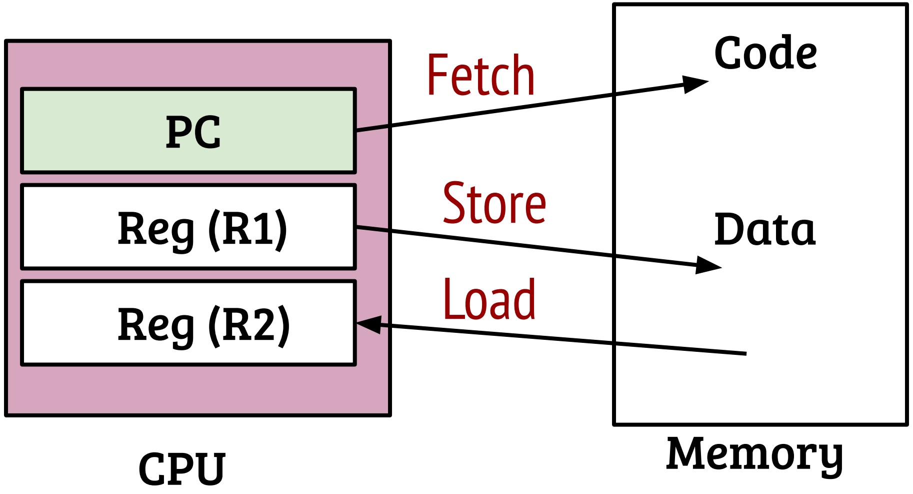
Context switch: state of a process

Process (MPlayer)

Register state (R1, R2, PC)

Memory state

Other states



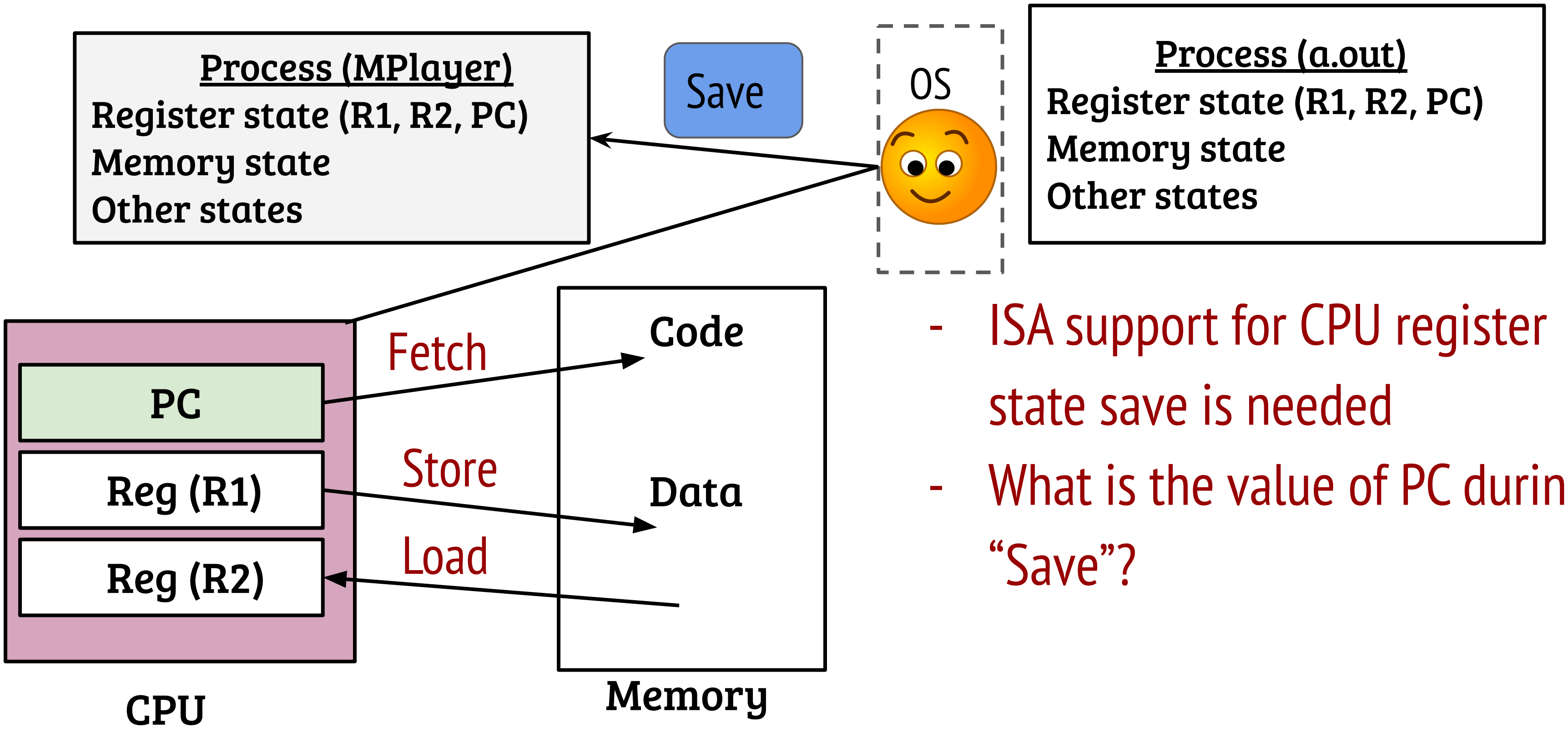
Process (a.out)

Register state (R1, R2, PC)

Memory state

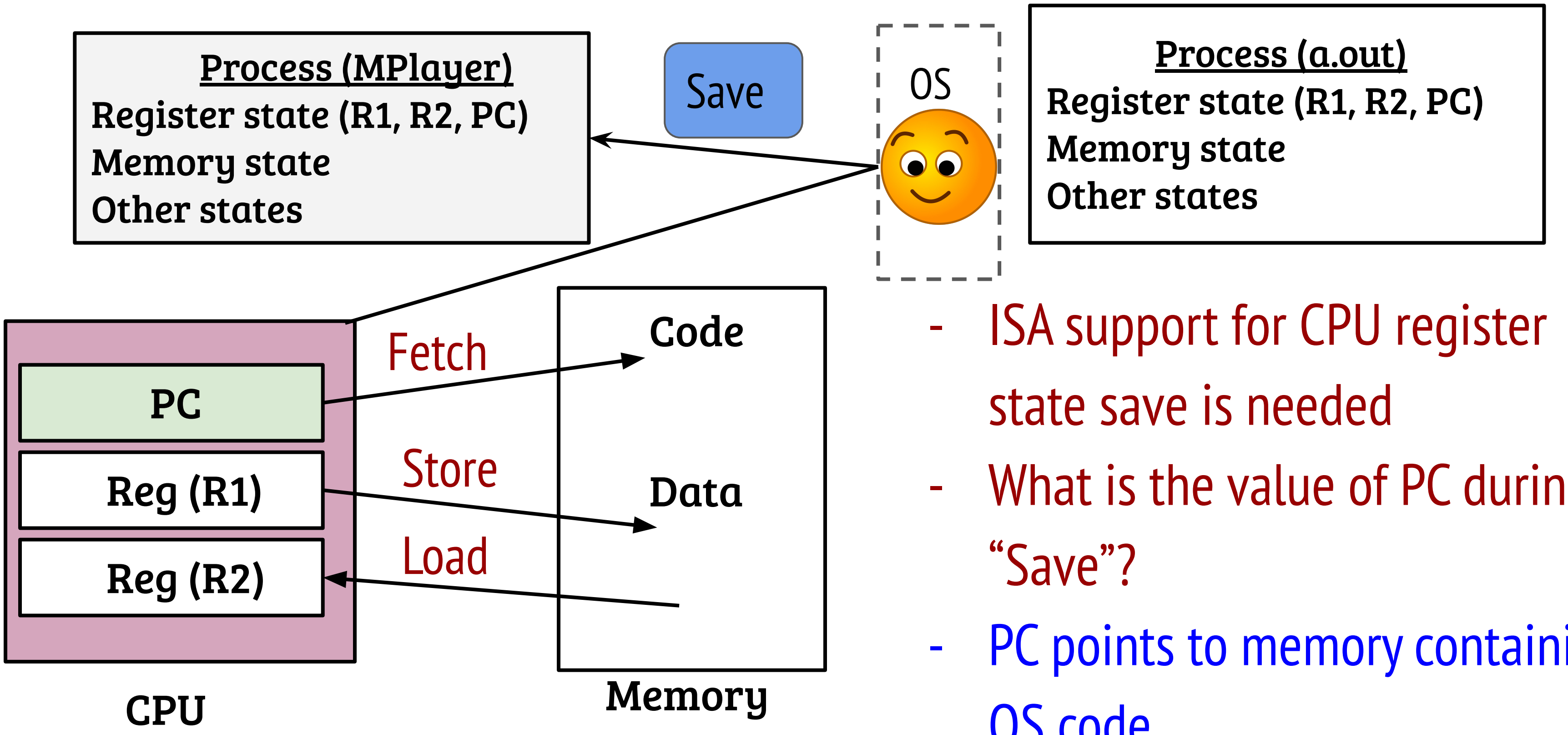
Other states

Context switch: saving the state of outgoing process



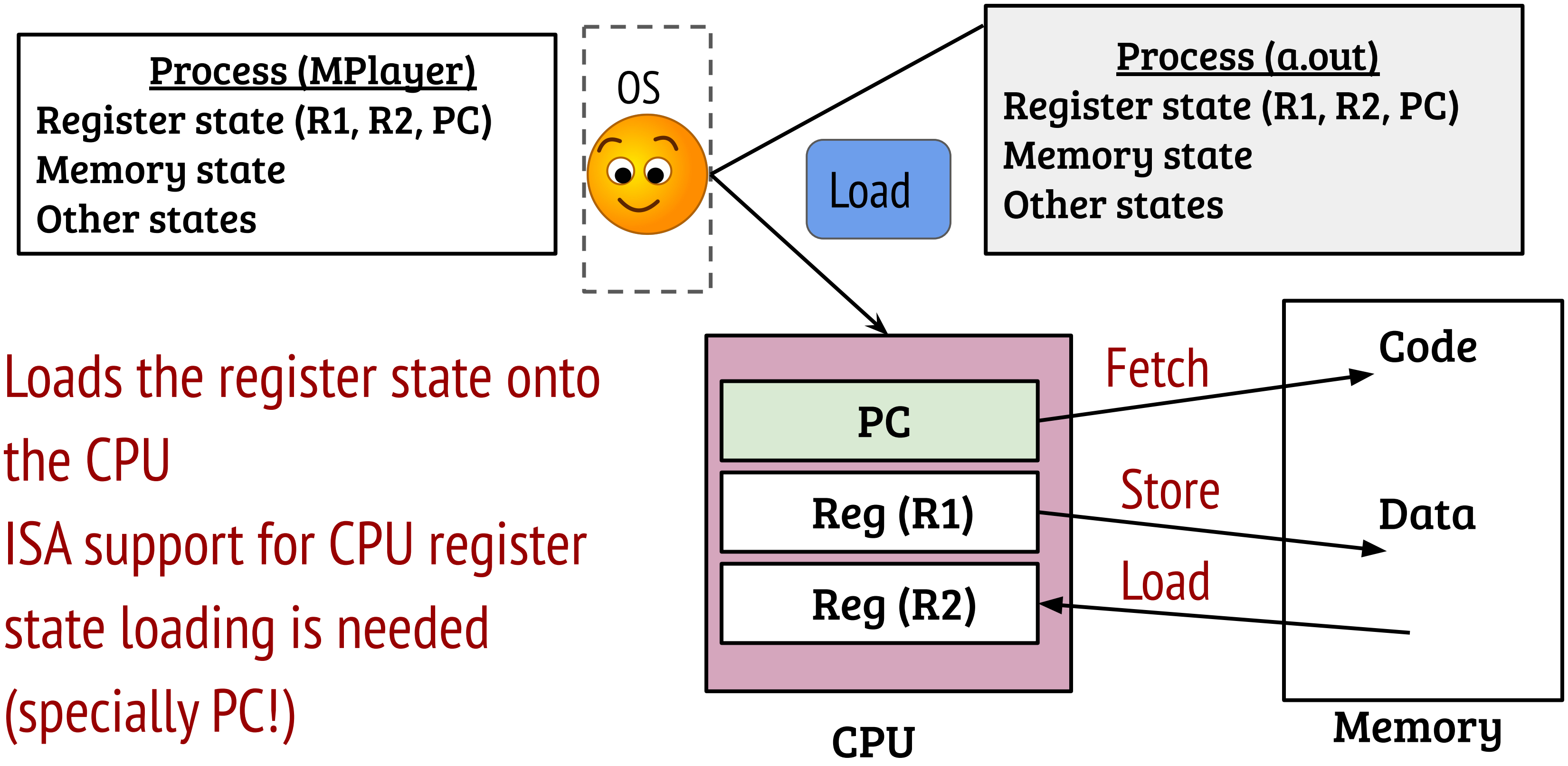
- ISA support for CPU register state save is needed
- What is the value of PC during “Save”?

Context switch: saving the state of outgoing process



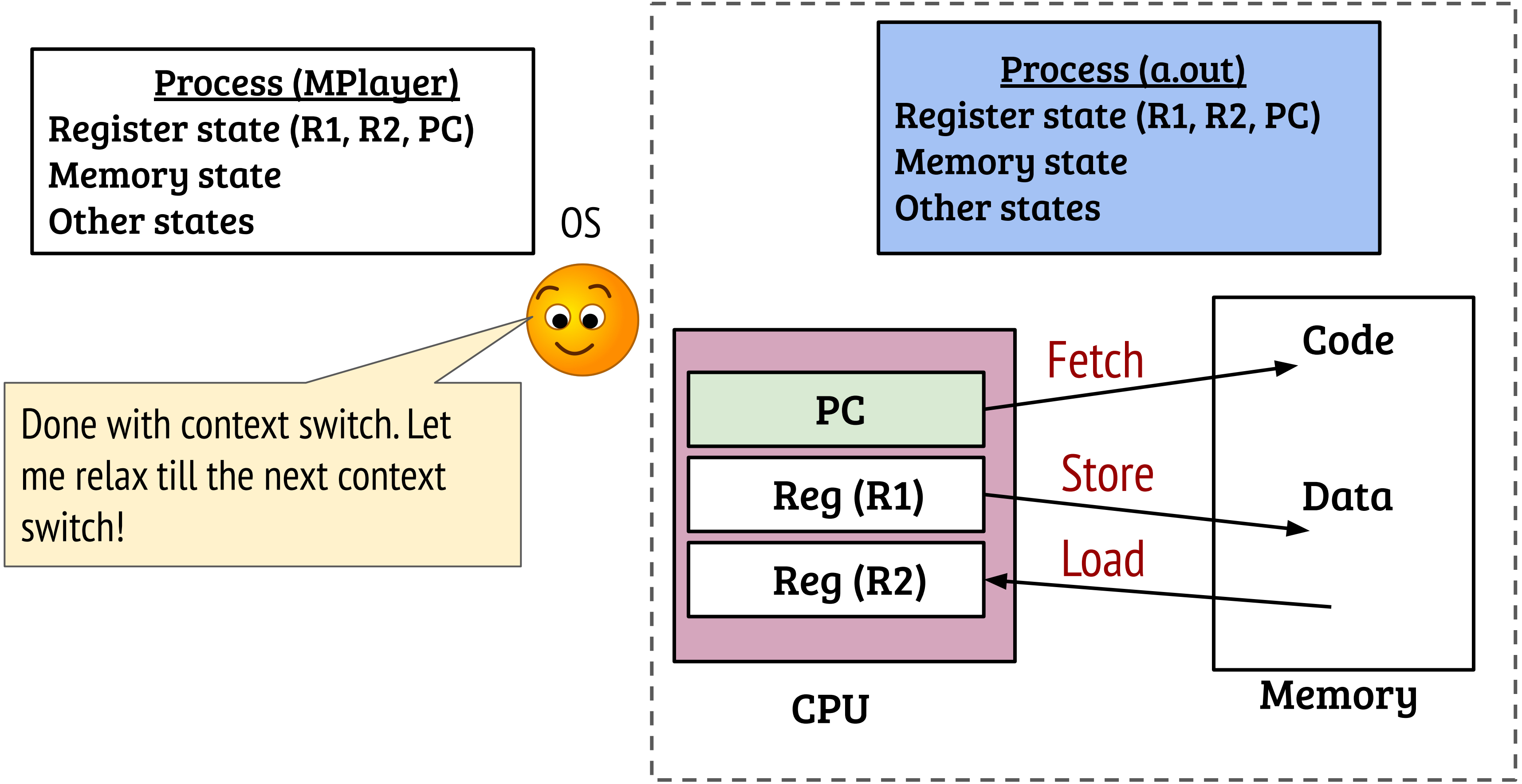
- ISA support for CPU register state save is needed
- What is the value of PC during "Save"?
- PC points to memory containing OS code

Context switch: load the state of incoming process



- Loads the register state onto the CPU
- ISA support for CPU register state loading is needed (specially PC!)

Context switch: load the state of incoming process



Virtualization of the CPU

MPlayer

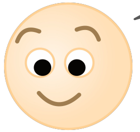
$./a.out

Browser

Everything is running! My program (a.out) is printing output and music is on!

- How CPU assignment is changed? (OR how context switch is performed?)
  - What happens to outgoing process? How does it come back?
- Overheads of context switch?
- How to decide the incoming process?

CPU



Hidden behind the abstraction!

- How does OS get the control of the CPU?
- In short, the OS configures the hardware to get the control. (will revisit)

Hidden behind the abstraction!

- How does OS get the control of the CPU?
- In short, the OS configures the hardware to get the control. (will revisit)
- How the OS knows which process is “ready”?
    - Why the process may not be ready?

Hidden behind the abstraction!

- How does OS get the control of the CPU?
- In short, the OS configures the hardware to get the control. (will revisit)
- How the OS knows which process is “ready”?
    - Why the process may not be ready?
- A process may be “sleeping” or waiting for I/O. Every process is associated with a state i.e., ready, running, waiting (will revisit).

Hidden behind the abstraction!

- How does OS get the control of the CPU?
- In short, the OS configures the hardware to get the control. (will revisit)
- How the OS knows which process is “ready”?
    - Why the process may not be ready?
- A process may be “sleeping” or waiting for I/O. Every process is associated with a state i.e., ready, running, waiting (will revisit).
- What is the memory state of a process?
    - How memory state is saved and restored?

Hidden behind the abstraction!

- How does OS get the control of the CPU?
- In short, the OS configures the hardware to get the control. (will revisit)
- How the OS knows which process is “ready”?
    - Why the process may not be ready?
- A process may be “sleeping” or waiting for I/O. Every process is associated with a state i.e., ready, running, waiting (will revisit).
- What is the memory state of a process?
    - How memory state is saved and restored?
- Memory itself virtualized. PCB + CPU registers maintain state (will revisit)

Example: hardware state of X86_64 (in gemOS)

```
struct user_regs{
u64 rip;        // PC
u64 r15 - r8;
u64 rax, rbx, rcx, rdx, rsi, rdi;
u64 rsp;       // stack pointer
u64 rbp;       // base pointer
};
```

- What is a stack in the context of hardware state? What is its use?

Example: hardware state of X86_64 (in gemOS)

```
struct user_regs{
u64 rip;        // PC
u64 r15 - r8;
u64 rax, rbx, rcx, rdx, rsi, rdi;
u64 rsp;       // stack pointer
u64 rbp;      // base pointer
};
```

- What is a stack pointer in the context of hardware state?
- Points to the TOS address of a stack in memory, operated by *push* and *pop* instructions

Example: hardware state of X86_64 (in gemOS)

```
struct user_regs{
u64 rip;        // PC
u64 r15 - r8;
u64 rax, rbx, rcx, rdx, rsi, rdi;
u64 rsp;        // stack pointer
u64 rbp;        // base pointer
};
```

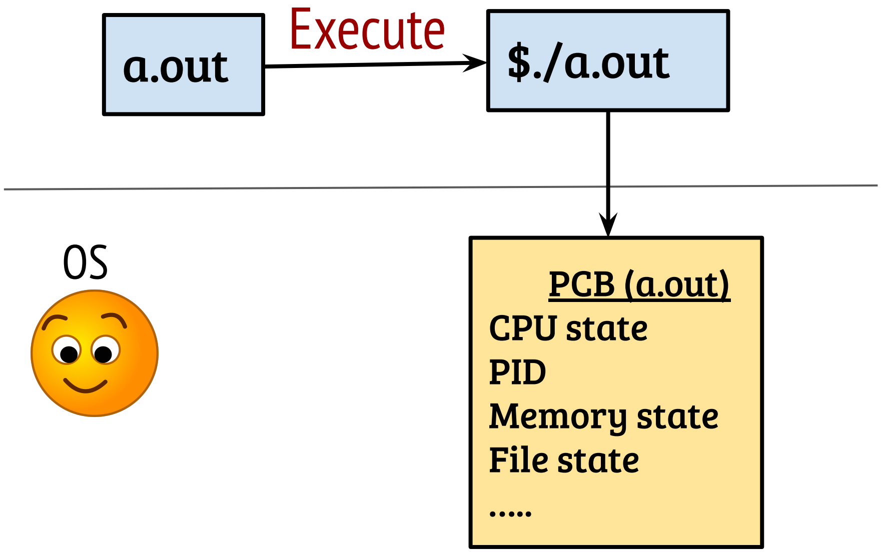
- What is a stack pointer in the context of hardware state?
- Points to the TOS address of a stack in memory, operated by *push* and *pop* instructions
- What is the use of stack?
- Makes it easy to implement function call and return

CS330: Operating Systems

Process API

Recap: The process abstraction

- The OS creates a process when we run an executable

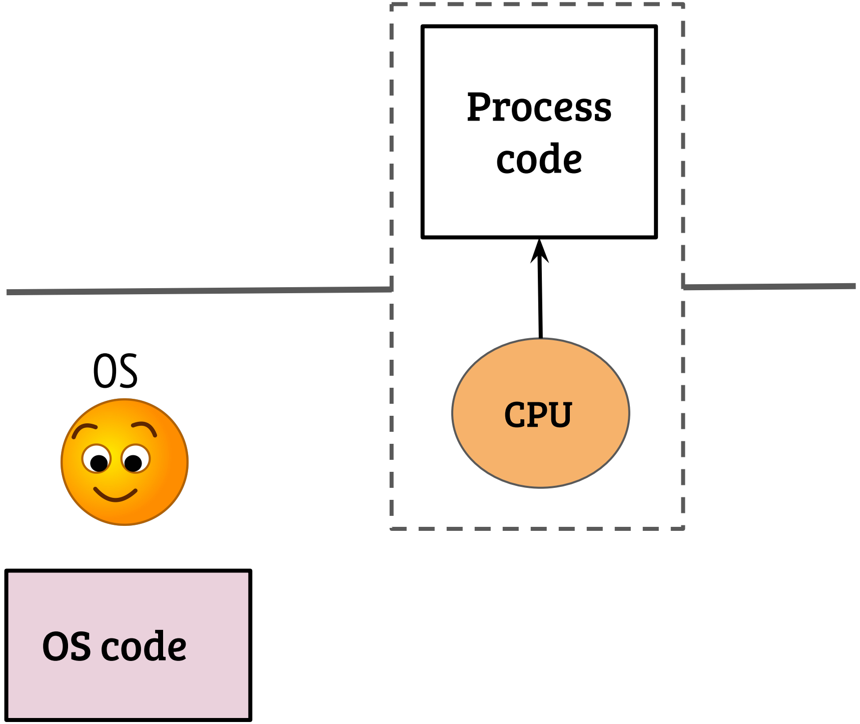


- When we execute "a.out" on a shell a process control block (PCB) is created
- Does it raise some questions related to the exact working?

Process creation: What and How?

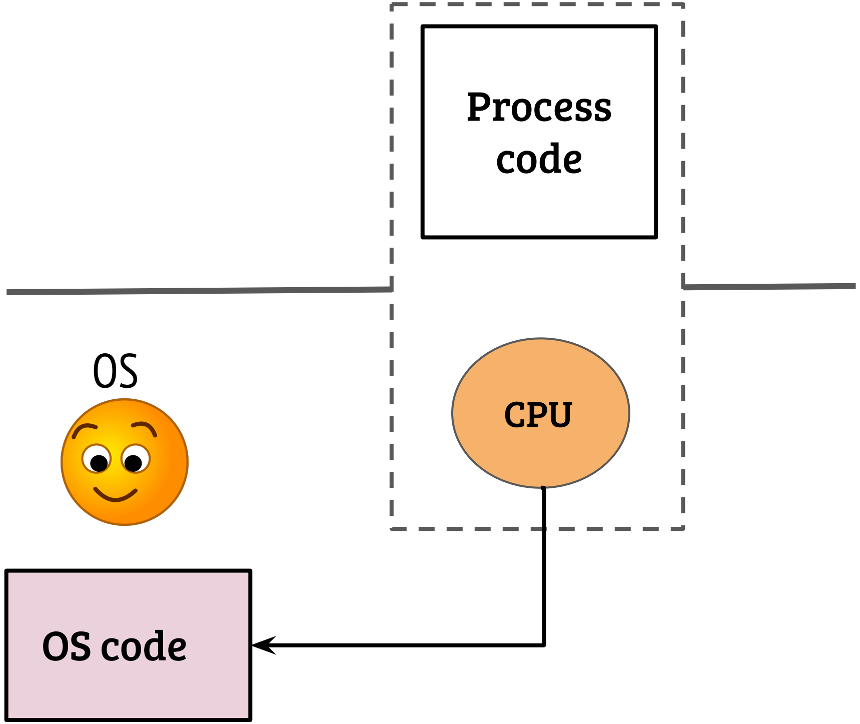
- How does OS come into action after typing "./a.out" in a shell?

System calls



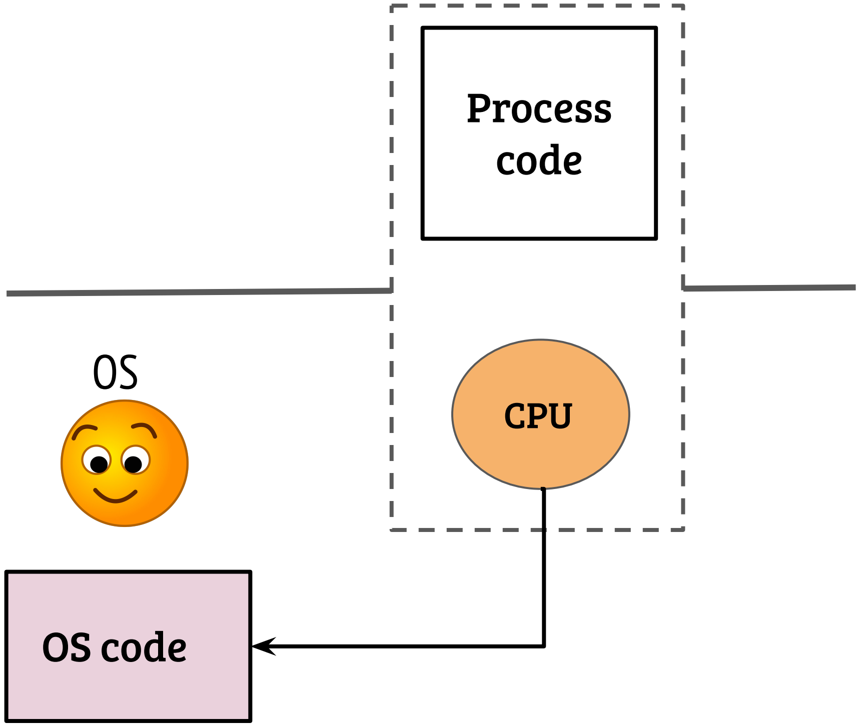
- CPU executing *user code* can invoke the *OS functions* using system calls

System call



- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call
- How system call is different from a function call?

System call



- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call
- How is system call different from a function call?
- Can be thought as an invocation of privileged functions (will revisit)

A simple system call: getpid( )

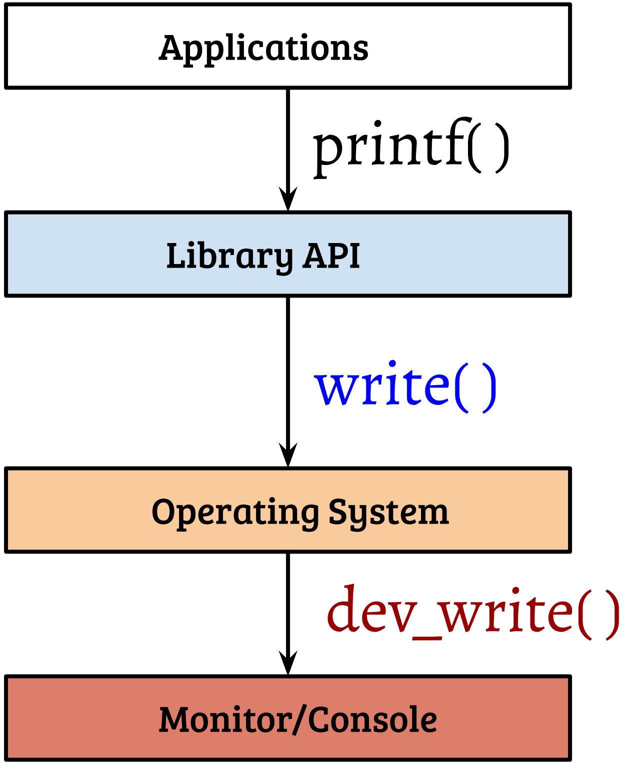
USER

```
main( )
{
  printf("%d\n", getpid( ));
}
```

OS

```
pid_t getpid( )
{
    PCB *current = get_current_process( );
    return (current → pid);
}
```

System calls and user libraries

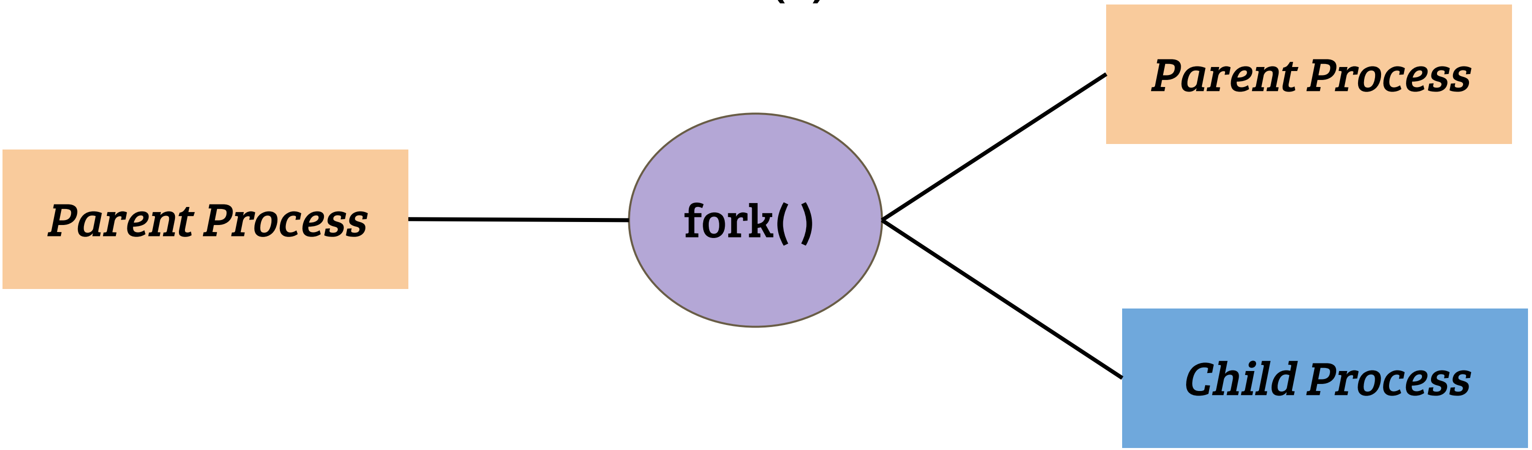


- Most system calls are invoked through wrapper library functions
- However, all system calls can be invoked directly (using syscall( ))

Process creation: What and How?

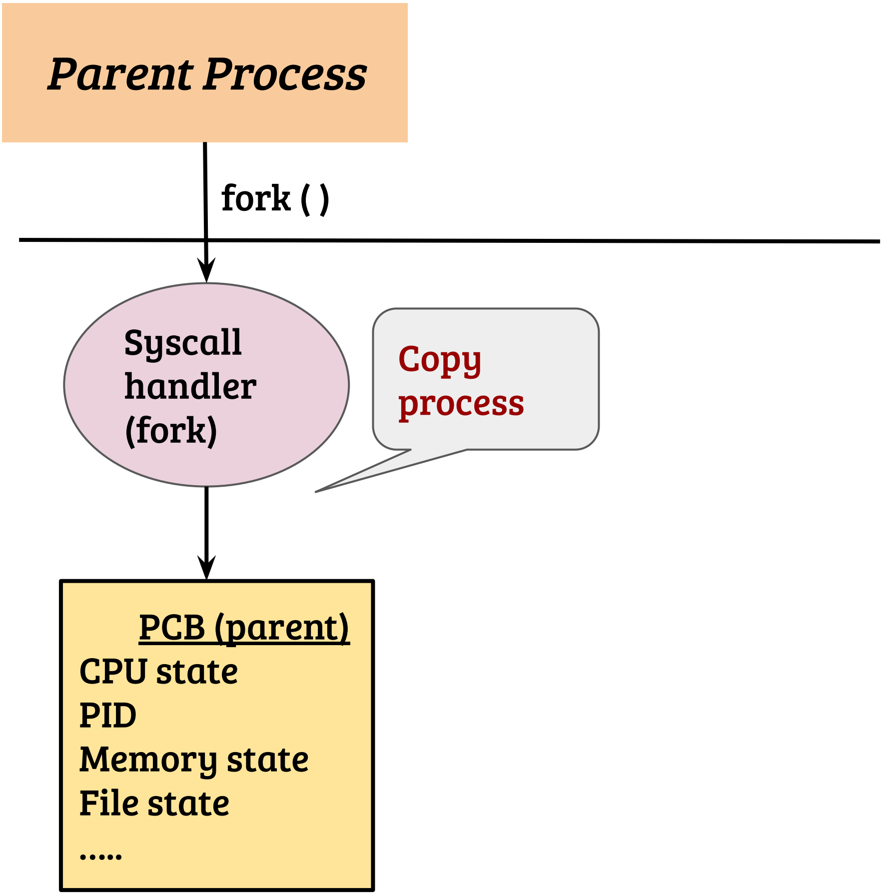
- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?

Process creation - fork( )

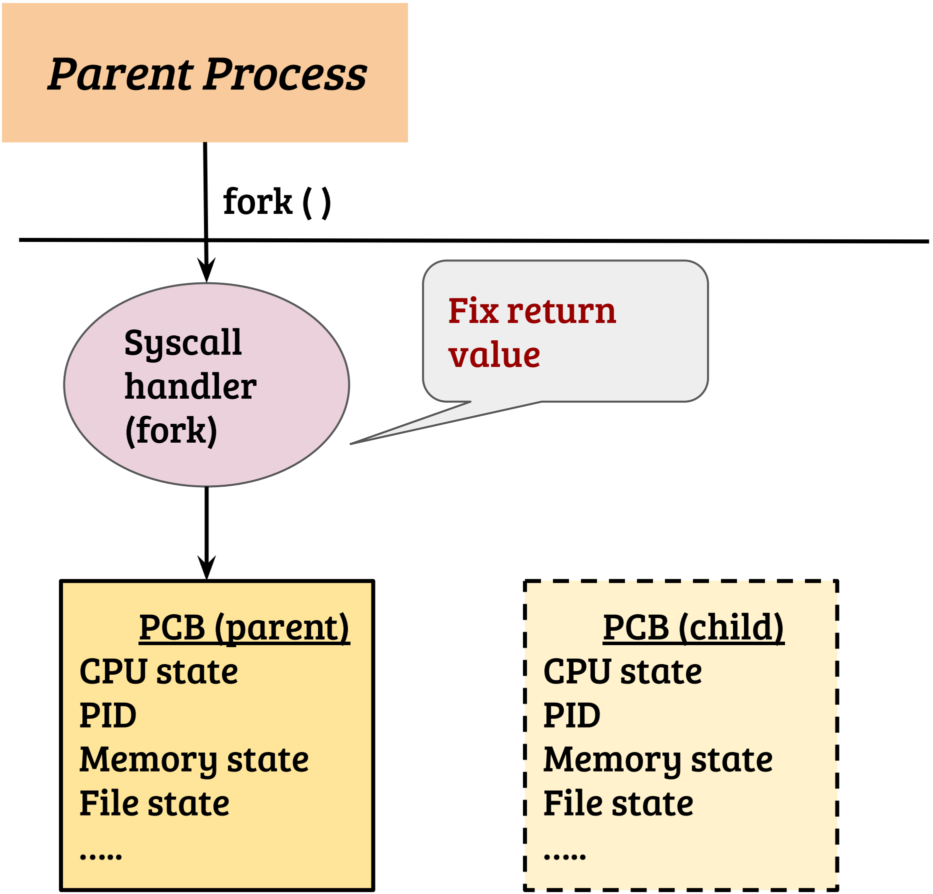


- fork( ) system call is weird; not a typical "privileged" function call
- fork( ) creates a new process; a *duplicate* of calling process
- On success, fork
    - Returns PID of child process to the caller (parent)
    - Returns 0 to the child

Typical implementation of fork

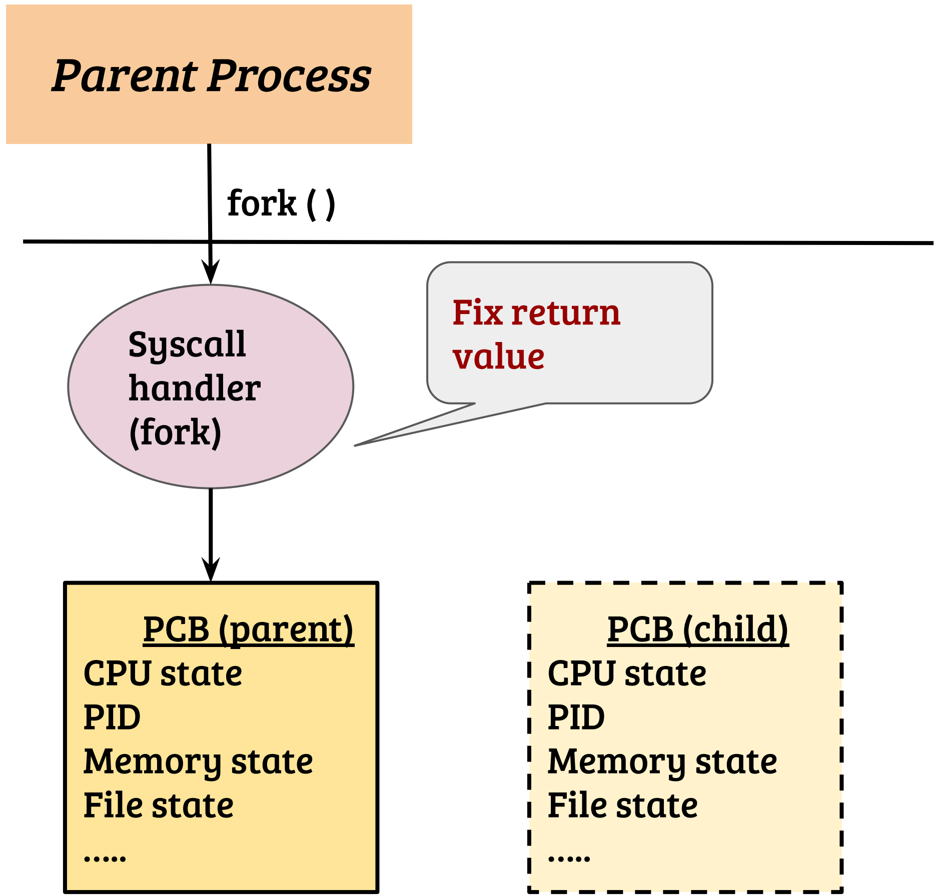


Typical implementation of fork



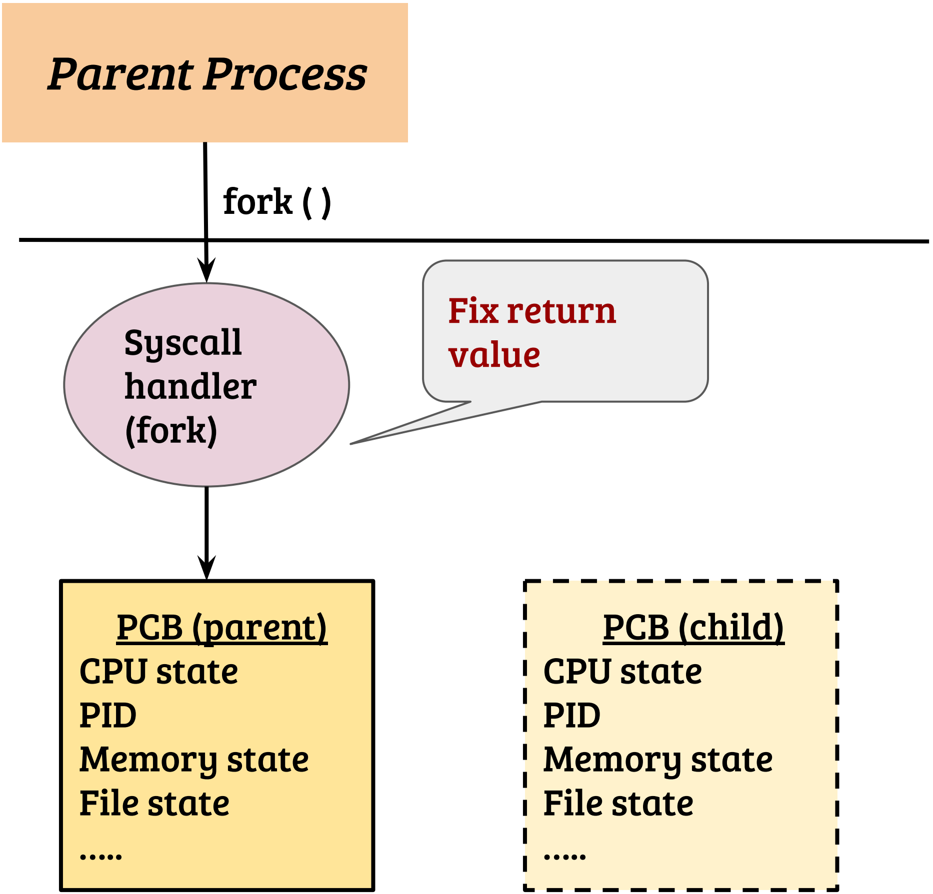
- Child should get ‘0’ and parent gets PID of child as return value. How?

Typical implementation of fork



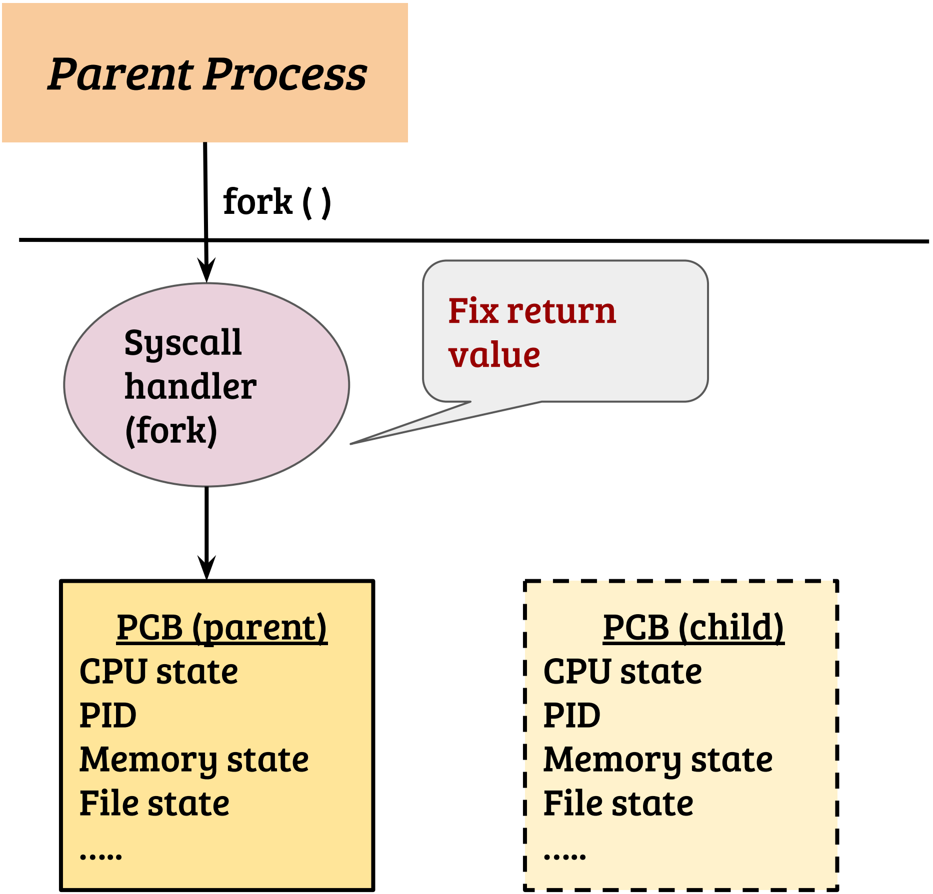
- Child should get ‘0’ and parent gets PID of child as return value. How?
- OS returns different values for parent and child

Typical implementation of fork



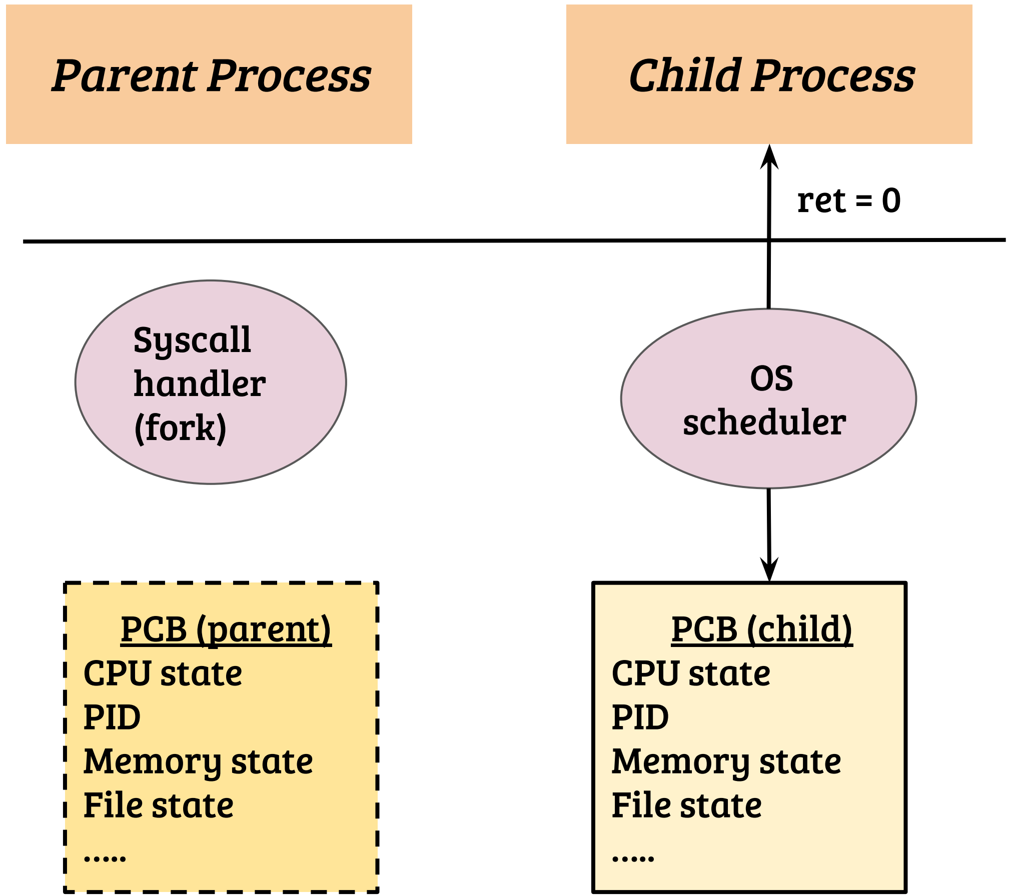
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?

Typical implementation of fork



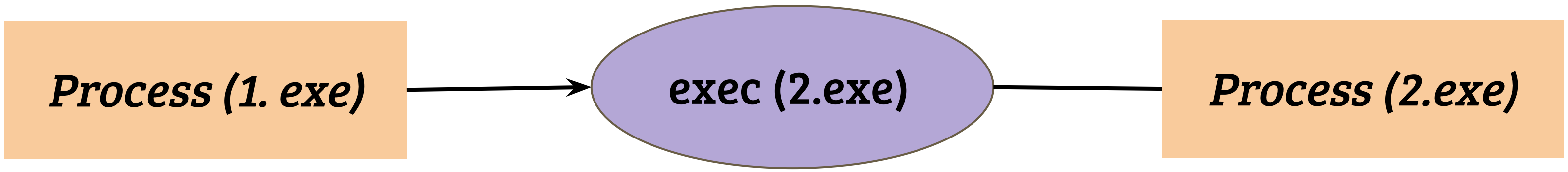
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?
- When OS schedules the child process

Typical implementation of fork



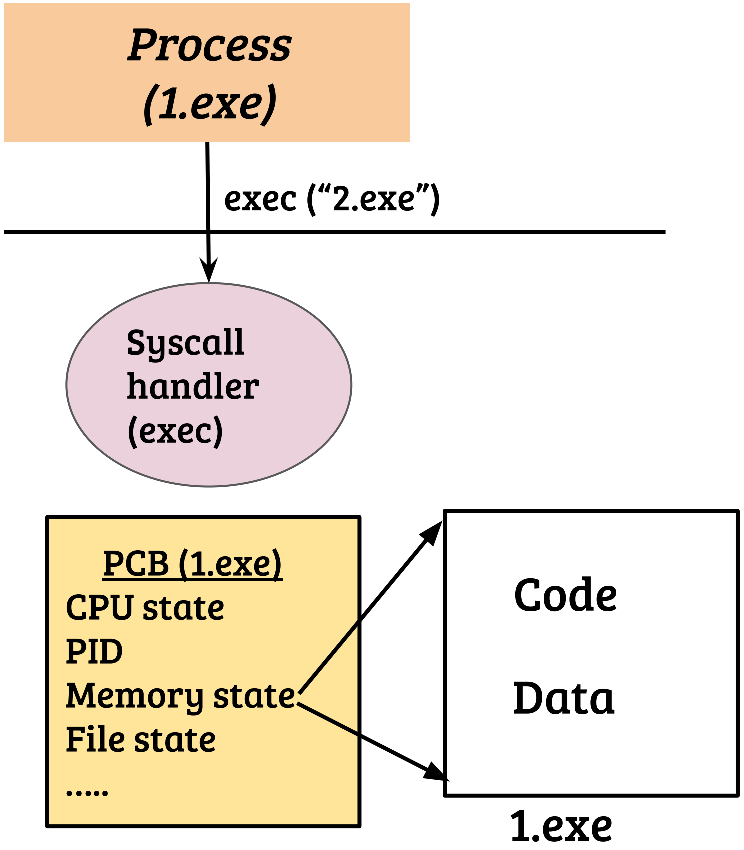
- PC is next instruction after fork( ) syscall, for both parent and child
- Child memory is an exact copy of parent
- Parent and child diverge from this point

Load a new binary - exec( )



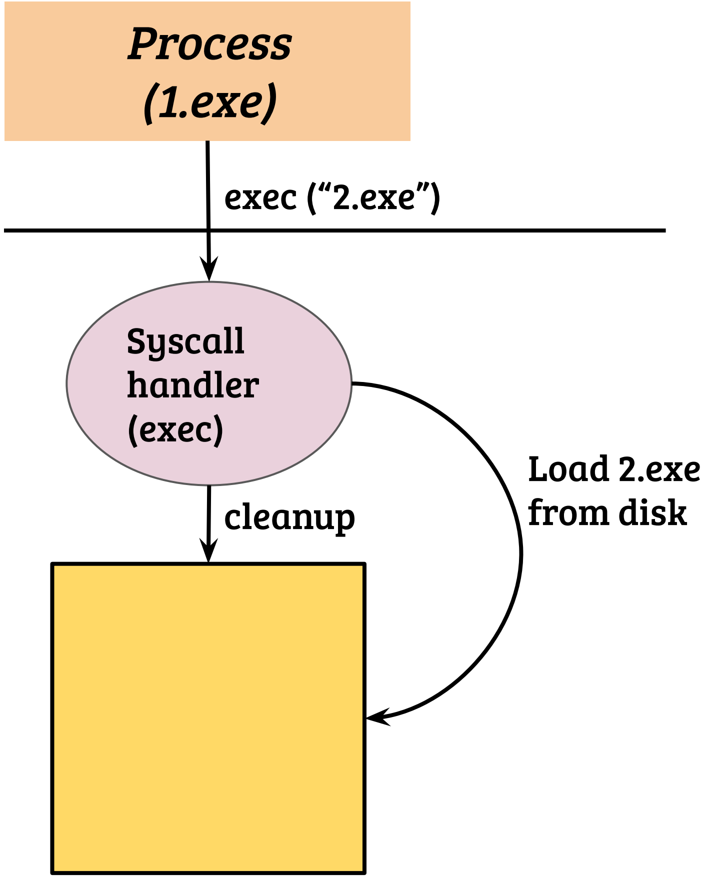
- Replace the calling process by a new executable
    - Code, data etc. are replaced by the new process
    - Usually, open files remain open

Typical implementation of exec



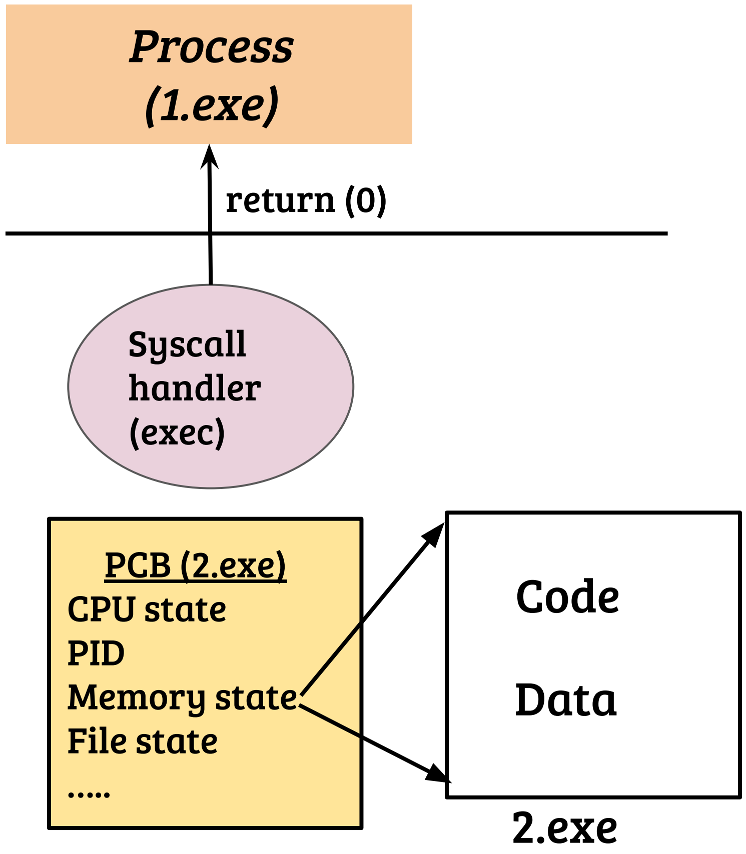
- The calling process commits self destruction! (almost)

Typical implementation of exec



- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same

Typical implementation of exec

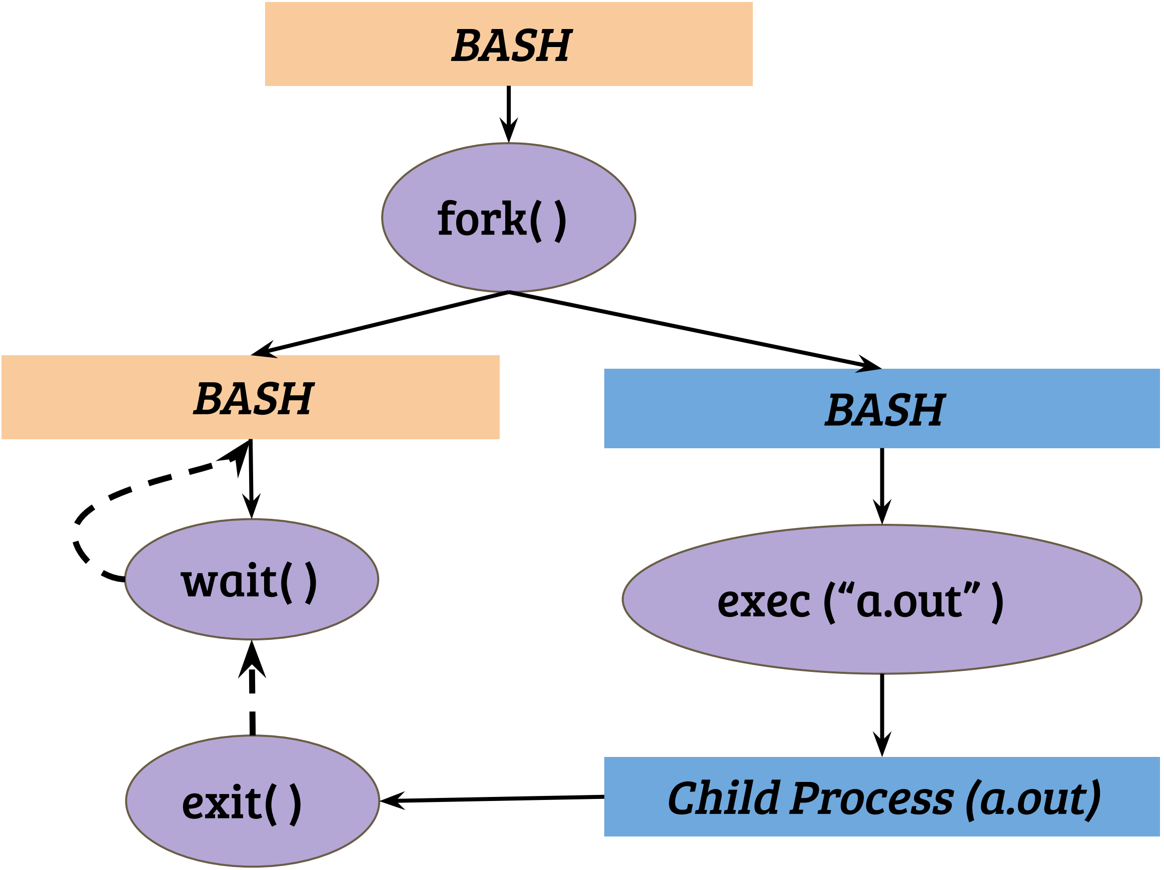


- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same
- On return, new executable starts execution
- PC is loaded with the starting address of the new process

Process creation: What and How?

- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls? In what order?

Shell command line: fork + exec + wait

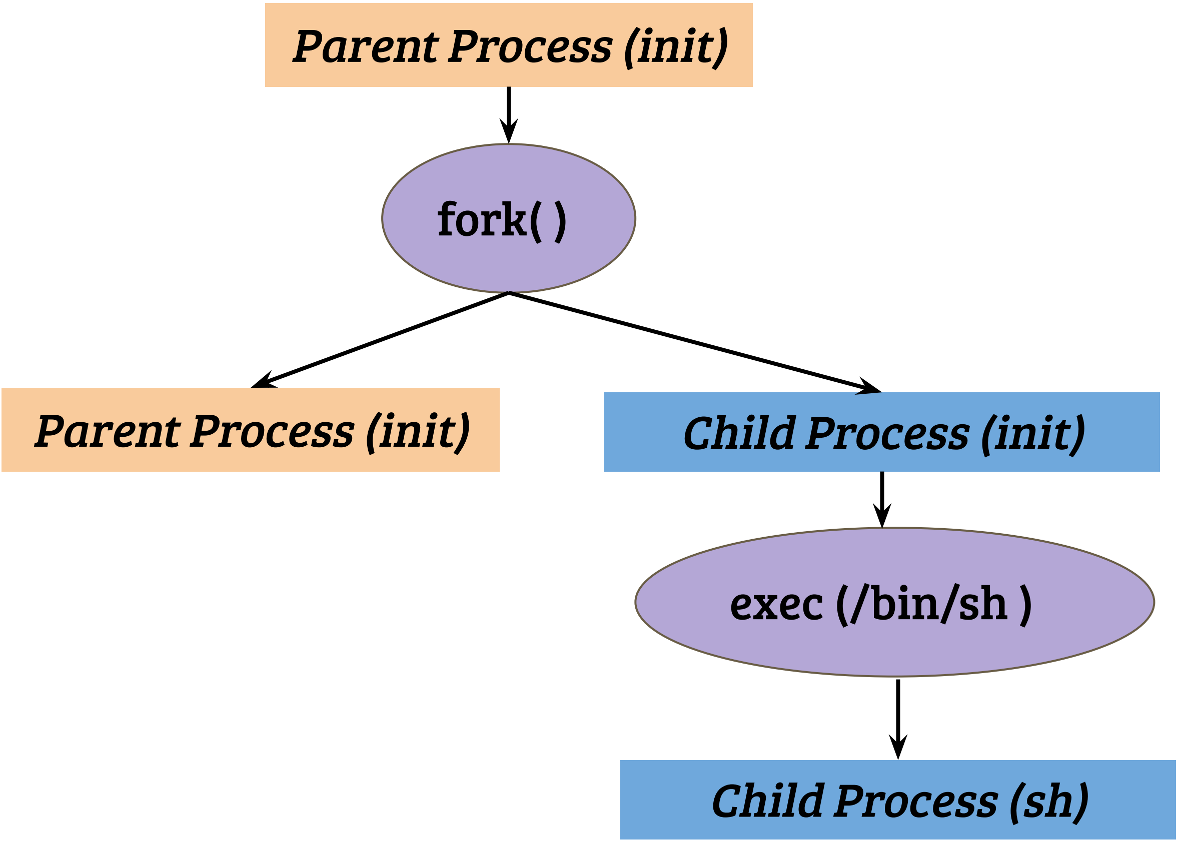


- Parent process calls wait( ) to wait for child to finish
- When child exits, parent gets notified

Process creation: What and How?

- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls? In what order?
- The shell process (bash process)
- What is the first user process?

Unix process family using fork + exec



- Fork and exec are used to create the process tree
- Commands: ps, pstree
- See the /proc directory in linux systems

Process creation: What and How?

- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls?
- The shell process (bash process)
- What is the first user process?
- In Unix systems, it is called the *init* process
- Who creates and schedules the init process?

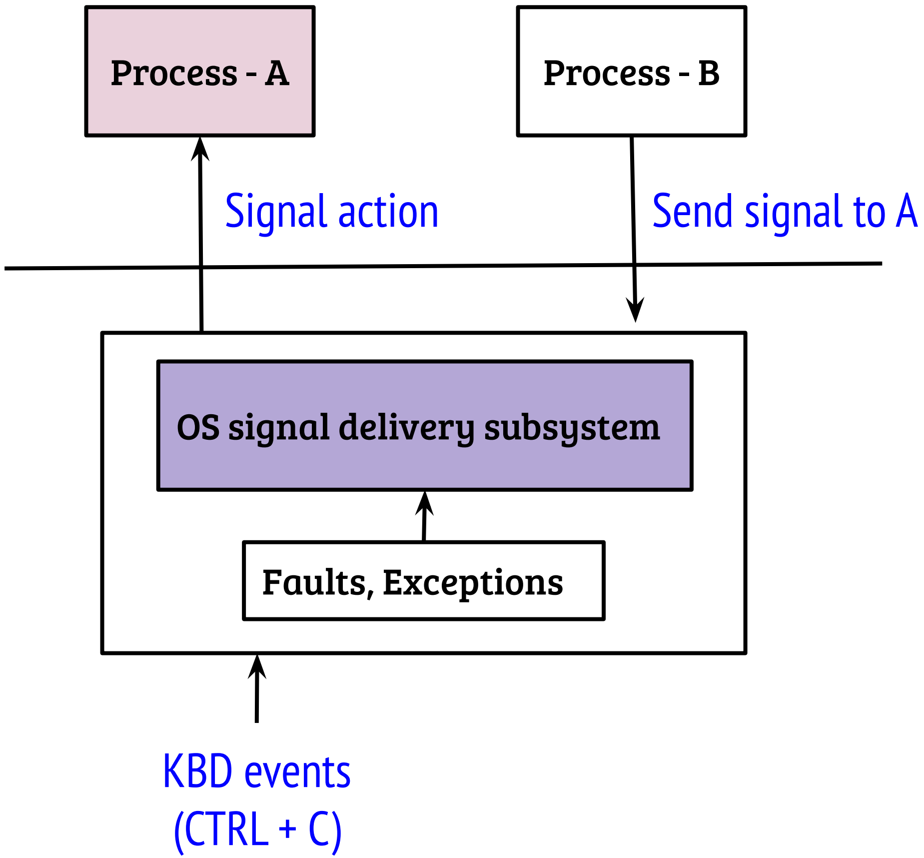
CS330: Operating Systems

Signals

Recap: Process creation

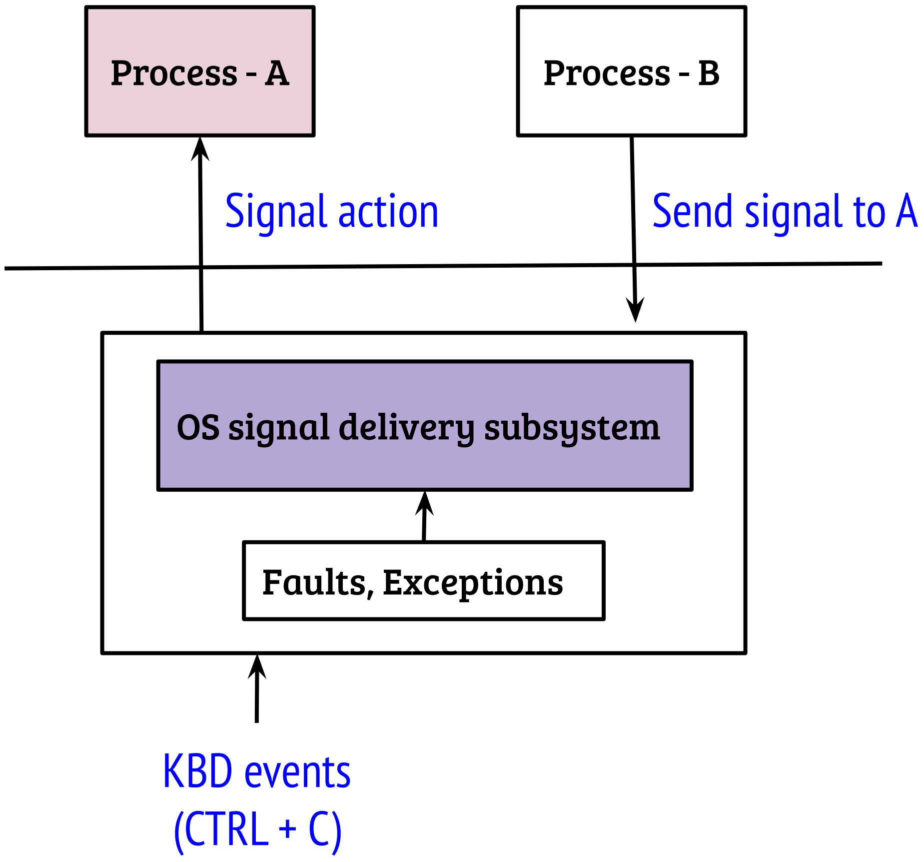
- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls?
- The shell process (bash process)
- What is the first user process?
- In Unix systems, it is called the *init* process
- Who creates and schedules the init process?
- After boot, the OS creates a PCB and loads the init executable

OS signal mechanism



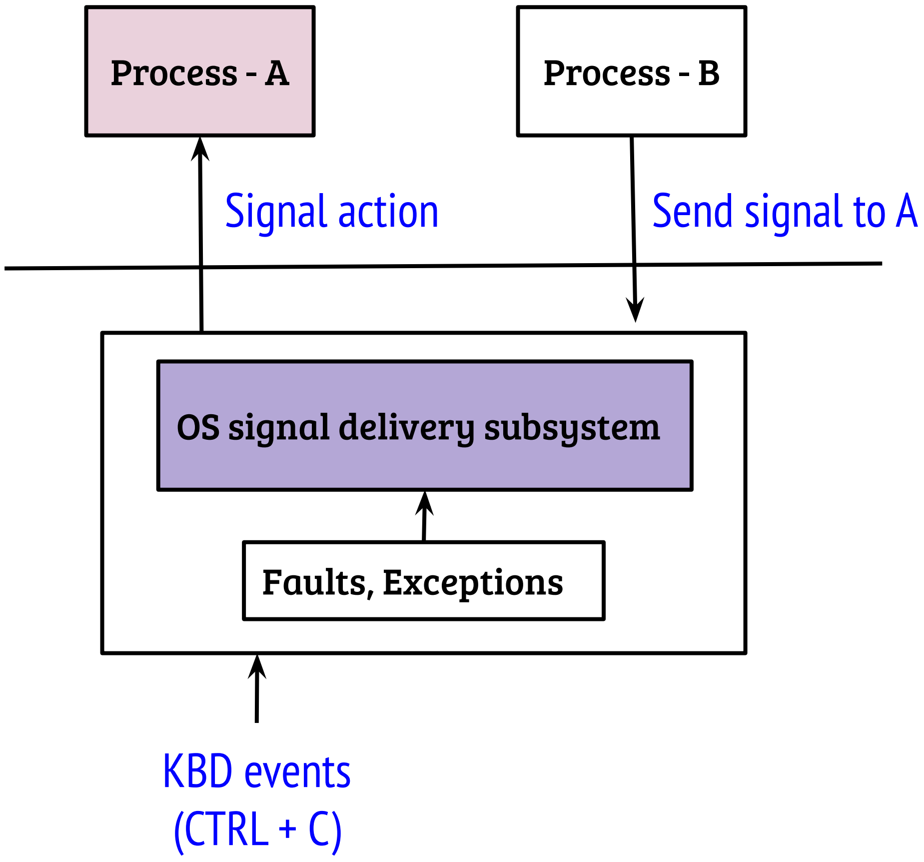
- A signal can be sent from two sources
  - Hardware events
  - Another process
- Step(1): Signal handler is registered (by process-A)

OS signal mechanism



- A signal can be sent from two sources
  - Hardware events
  - Another process
- Step(1): Signal handler is registered (by process-A)
- Step(2): OS logic invoked through syscall or hardware event

OS signal mechanism



- A signal can be sent from two sources
  - Hardware events
  - Another process
- Step(1): Signal handler is registered (by process-A)
- Step(2): OS logic invoked through syscall or hardware event
- Step(3): Signal handler is invoked

Signal semantics

Destination process

signal (signum, handler)

SIGHUP
SIGINT
SIGALRM
SIGUSR1
SIGCHLD
....

1. **SIG_IGN: Ignore the signal**
2. **SIG_DFL: Default action**
3. **Function address: custom handling**

- If signal handler not registered, process is terminated (mostly)
- SIGKILL and SIGSTOP → no custom actions, why?

Signal semantics

Destination process

signal (signum, handler)

SIGHUP
SIGINT
SIGALRM
SIGUSR1
SIGCHLD
....

1. **SIG_IGN: Ignore the signal**
2. **SIG_DFL: Default action**
3. **Function address: custom handling**

- If signal handler not registered, process is terminated (mostly)
- SIGKILL and SIGSTOP → no custom actions, why?
- OS or root user should have some way to kill (rouge) processes

Signal semantics

Source process

kill (pid, signum)

PID Of target process

SIGHUP
SIGINT
SIGALRM
SIGUSR1
SIGCHLD
....

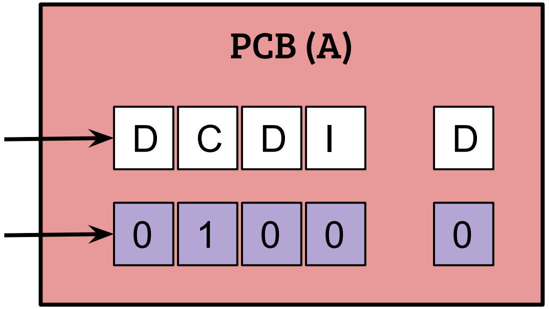
- If pid == 0, signal is sent to all processes in the process group
- Must have permissions to send signals → same user or root user

OS support for signals

```
void sighandler(int signo){
   printf("Signal received\n");
}
main( ){
     signal(1, sighandler);
     while(1)  do_something( );
}
```

USER

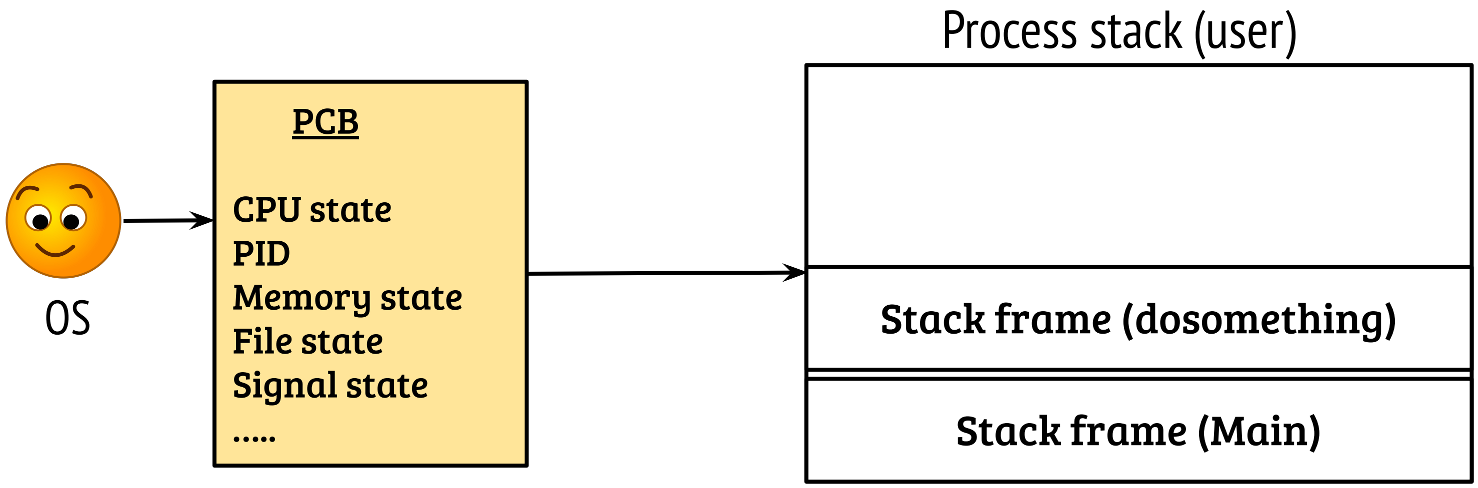
OS



D - default
I - Ignore
C - Custom

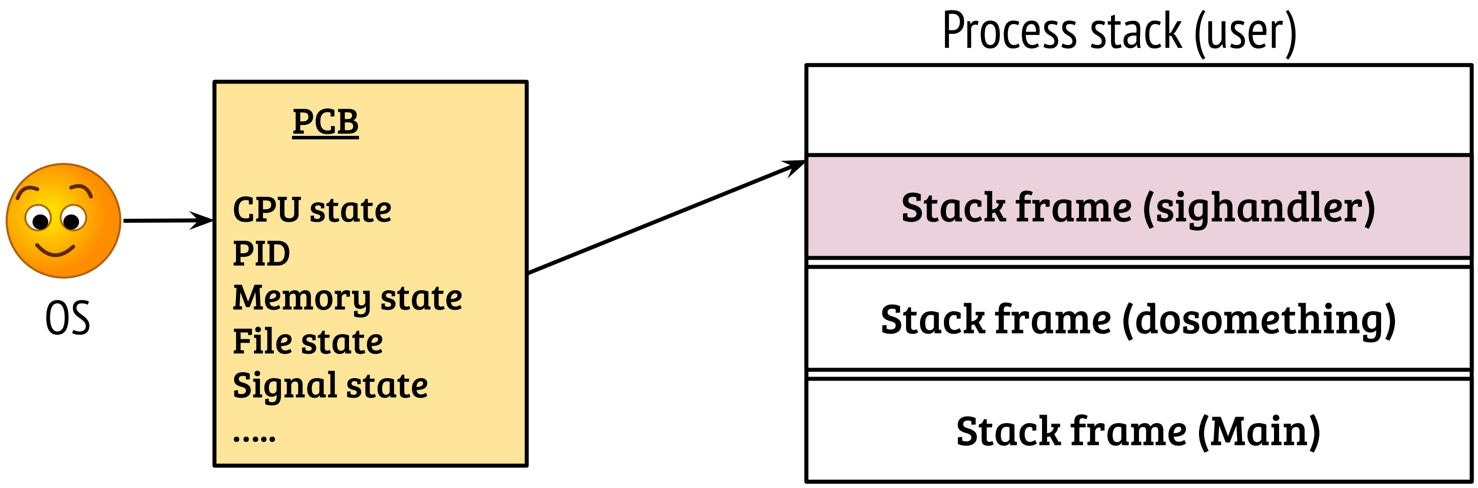
- Signal handlers are registered by the process
- Signal pending is modified
  - *kill* system call
  - OS event handler
- How are signals delivered?
- When are signals delivered?

Signal delivery (invoking the handler)



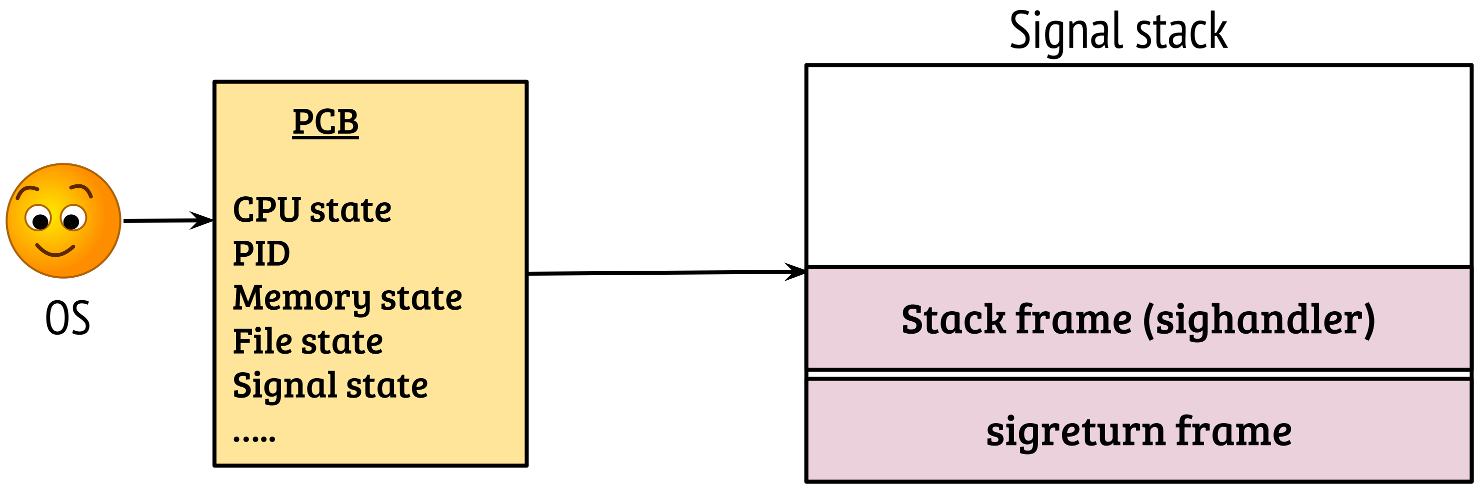
- How to invoke the signal handler?
- Current state: PC points to the last execution address, SP points to the TOS

Signal delivery using the user stack



- How to invoke the signal handler?
- Current state: PC points to the last execution address, SP points to the TOS
- Mimic a function call by modifying user stack and PC ⟶ sighandler

Signal delivery using signal stack (Linux)



- OS allocates a stack before invoking the handler, *remembers the old stack*
- Creates a stack frame to invoke an *sigreturn* system call to free the stack
- Original stack restored by OS *sigreturn* handler

OS support for signals

```
void sighandler(int signo){
   printf("Signal received\n");
}
main( ){
    signal(1, sighandler);
    while(1)  do_something( );
}
```

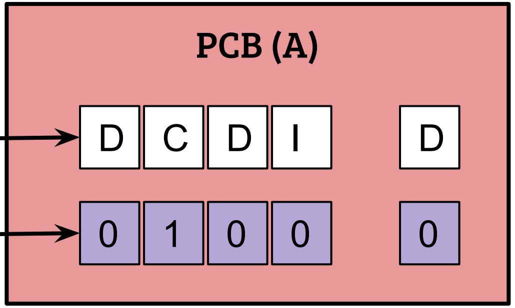
USER

- How are signals delivered?
  - By modifying the user stack and PC address
  - Using an alternate temporary (signal) stack
- When are signals delivered?

OS 

Signal handlers

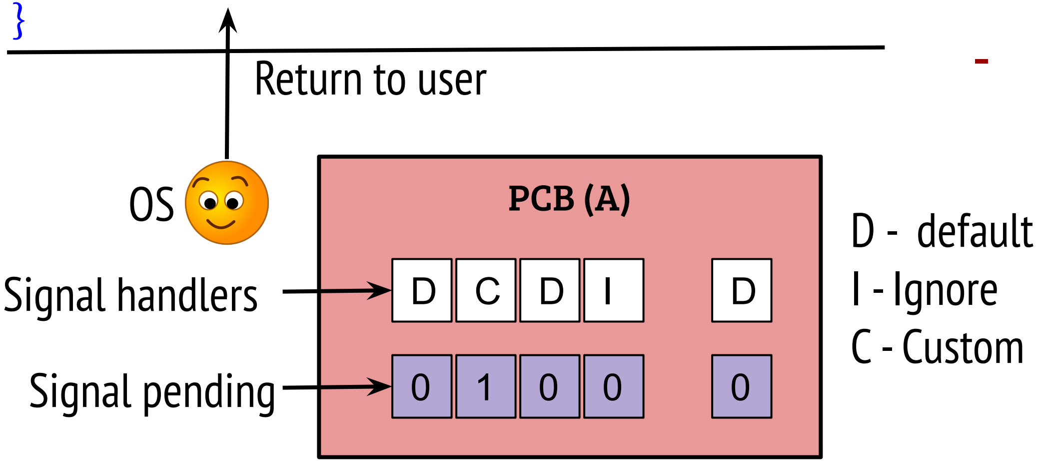
Signal pending



D - default
I - Ignore
C - Custom

OS support for signals

```
void sighandler(int signo){
   printf("Signal received\n");
}
main( ){
    signal(1, sighandler);
    while(1)  do_something( );
}
```



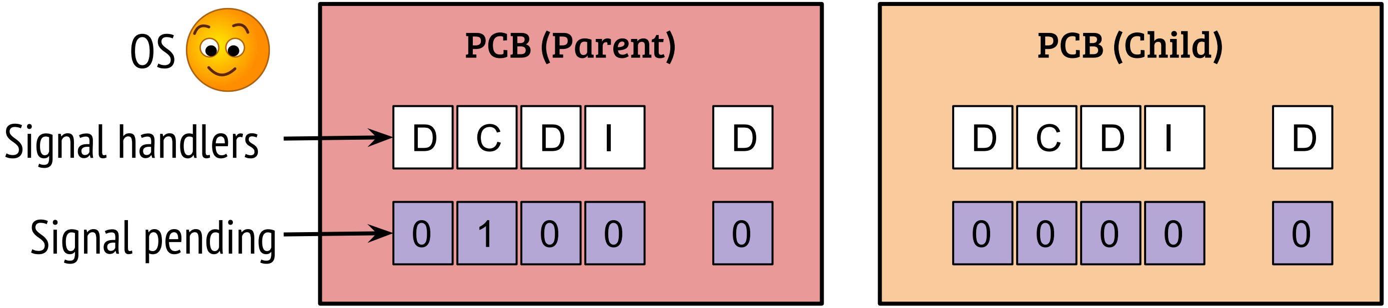
- How are signals delivered?
  - By modifying the user stack and PC address
  - Using an alternate temporary (signal) stack
- When are signals delivered?
  - On return to user space

Signal and fork

```
main(){
    signal(1, sighandler);
    fork();
    .......
}
```

USER

- Child inherits the signal handlers
- Note that, signal pending is not copied
- What is the logic?



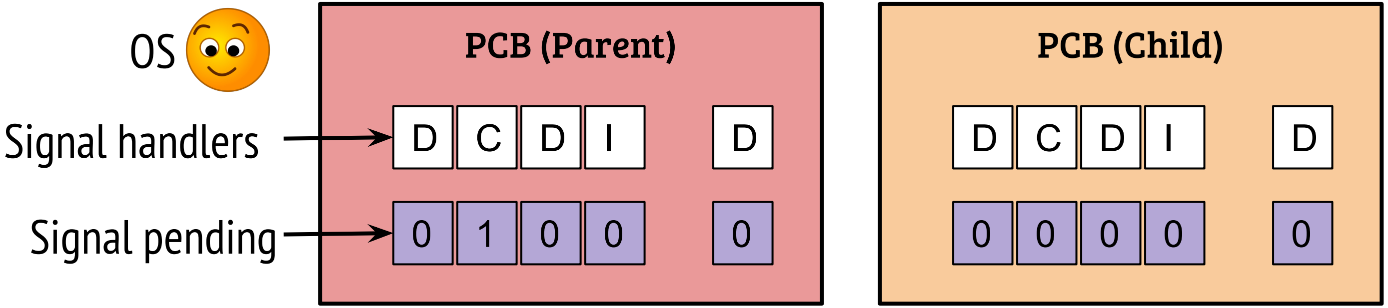
D - default   I - Ignore   C - Custom

Signal and fork

```
main( ){
    signal(1, sighandler);
    fork( );
    .......
}
```

USER

- Child inherits the signal handlers
- Note that, signal pending is not copied
- What is the logic?
  - Signal intended for the parent, not for child

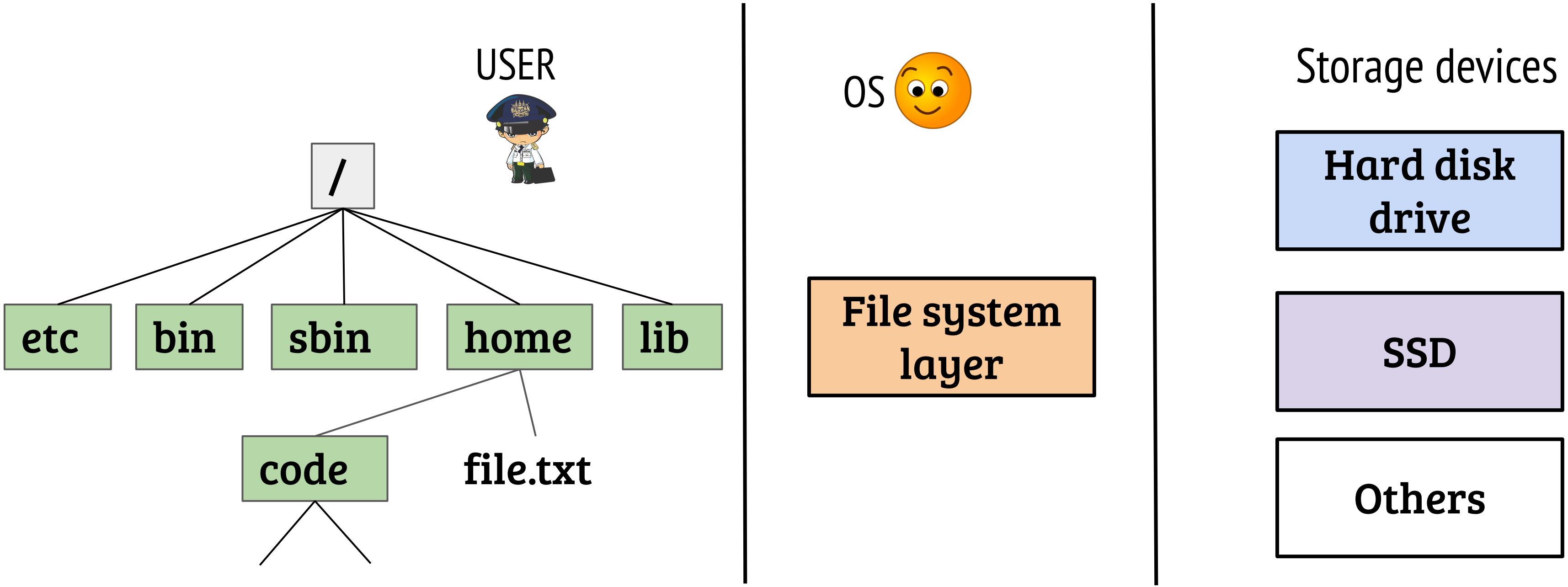


D - default   I - Ignore   C - Custom

CS330: Operating Systems

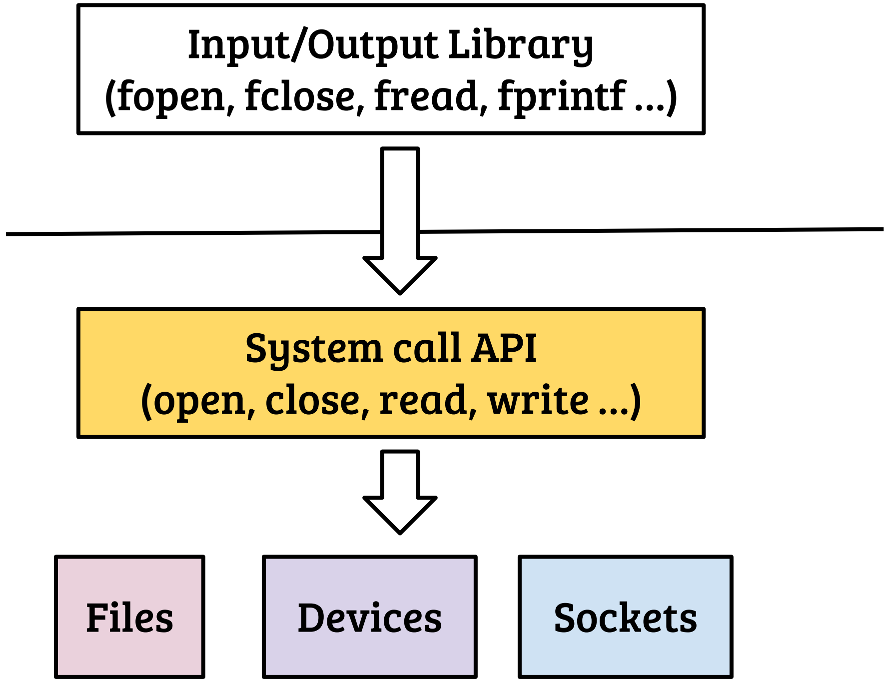
Files

The file system



- File system is an important OS subsystem
  - Provides abstractions like files and directories
  - Hides the complexity of underlying storage devices

File system interfacing



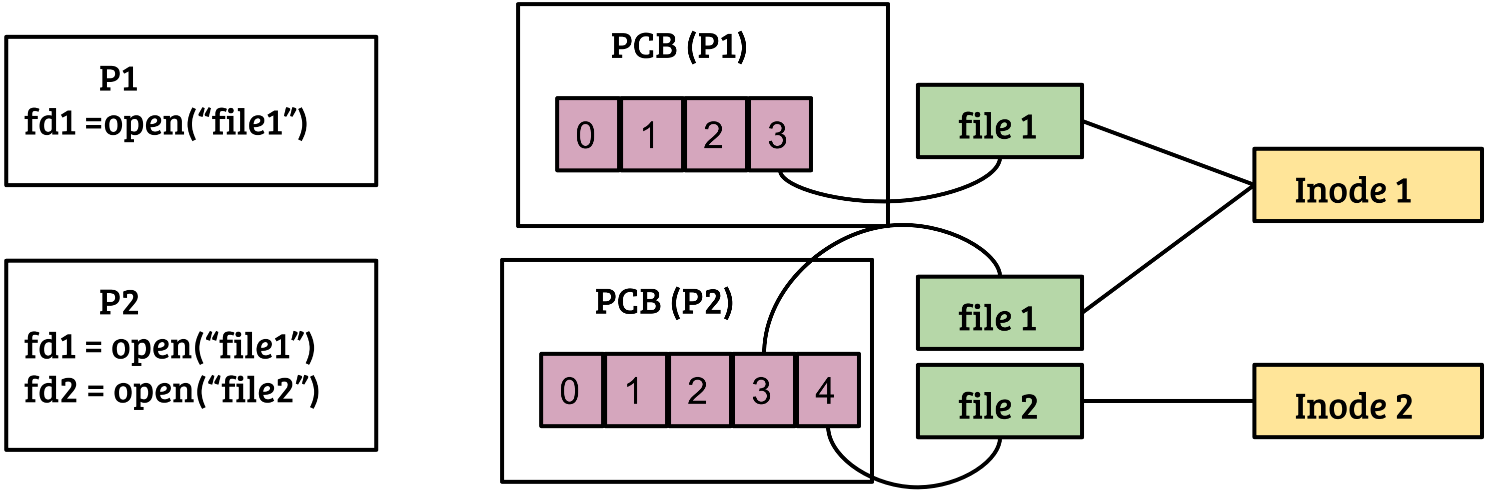
- User process identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.

open: getting a handle

```
int open (char *path, int flags, mode_t mode)
```

- Access mode specified in flags : O_RDONLY, O_RDWR, O_WRONLY
- Access permissions check performed by the OS
- On success, a file descriptor (int) is returned
- If flags contain O_CREAT, mode specifies the file creation mode

Process view of file



- Per-process file descriptor table with pointer to a "file" object
- fd ⟶ file (many-to-one), file ⟶ inode (many-to-one)
- On fork( ), child inherits open file handles
- 0, 1, 2 are STDIN, STDOUT and STDERR, respectively

File information (stat, fstat)

```
int stat(const char *path, struct stat *sbuf);
```

- Returns the information about file/dir in the argument path
- The information is filled up in structure called stat

```
struct stat sbuf;
stat("/home/user/tmp.txt", &sbuf);
printf("inode = %d size = %ld\n", sbuf.st_ino, sbuf.st_size);
```

- Other useful fields in *struct stat* : st_uid, st_mode (Refer stat man page)

Read and Write

ssize_t read (int fd, void *buf, size_t count);

- fd → file handle
- buf → user buffer as read destination
- count → #of bytes to read
- read ( ) returns #of bytes actually read, can be smaller than count

ssize_t write (int fd, void *buf, size_t count);

- Similar to read

lseek

```
off_t lseek(int fd, off_t offset, int whence);
```

- fd → file handle
- offset → target offset
- whence → SEEK_SET, SEEK_CUR, SEEK_END
- On success, returns offset from *the starting of the file*
- Examples
  - lseek(fd, SEEK_CUR, 100) → forwards the file position by 100 bytes
  - lseek(fd, SEEK_END, 0) → file pos at EOF, returns the file size
  - lseek(fd, SEEK_SET, 0) → file pos at beginning of file

Duplicate file handles (dup and dup2)

```
int dup(int oldfd);
```

- The dup() system call creates a “copy” of the file descriptor oldfd
- Returns the lowest-numbered unused descriptor as the new descriptor
- The old and new file descriptors represent the same file

Duplicate file handles (dup and dup2)

```
int fd, dupfd;
fd = open("tmp.txt");
close(1);
dupfd = dup(fd);      //What will be the value of dupfd?
printf("Hello world\n");  // Where will be the output?
```

Duplicate file handles (dup and dup2)

```
int fd, dupfd;
fd = open("tmp.txt");
close(1);
dupfd = dup(fd);      //What will be the value of dupfd?
printf("Hello world\n");  // Where will be the output?
```

- Value of dupfd = 1 (assuming STDIN is open)
- "Hello world" will be written to tmp.txt file

Duplicate file handles (dup and dup2)

```
int dup2(int oldfd, int newfd);
```

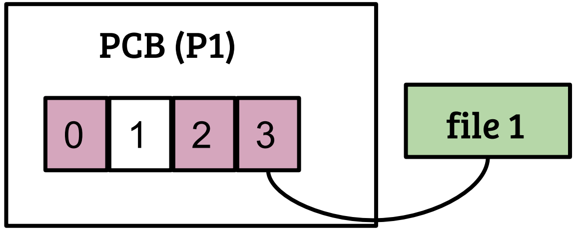
- Close newfd before duping the file descriptor oldfd
- dup2 (fd, 1) equivalent to
  - close(1);
  - dup(fd);

Duplicate file handles (dup and dup2)

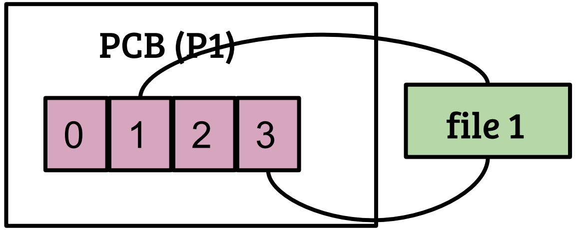
P1
fd1 =open("file1")

+ dup(fd1)

Before dup( )



After dup( )



- Lowest numbered unused fd (i.e., 1) is used (Assume STDOUT is closed before)
- Duplicate descriptors share the same file state
- Closing one file descriptor *does not* close the file

Use of dup: shell redirection

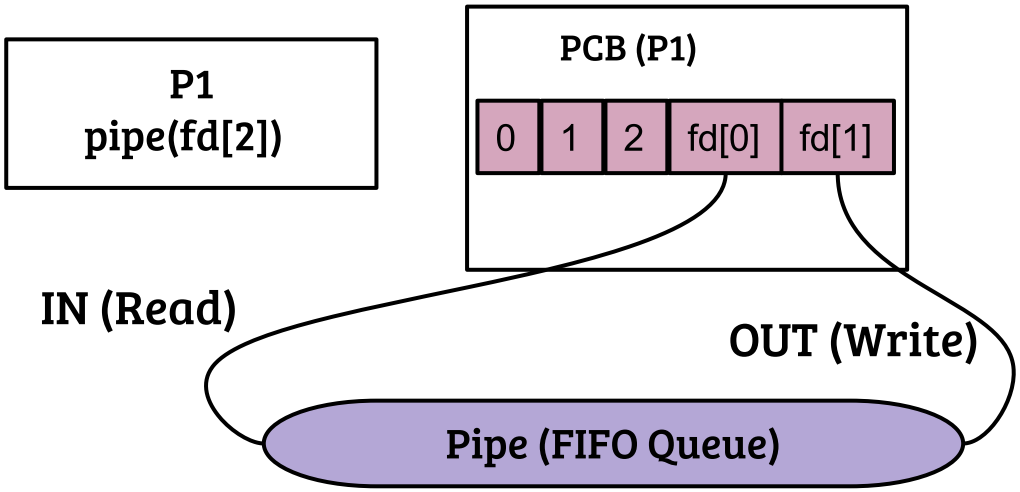
- Example: ls > tmp.txt
- How implemented?

Use of dup: shell redirection

- Example: ls > tmp.txt
- How implemented?

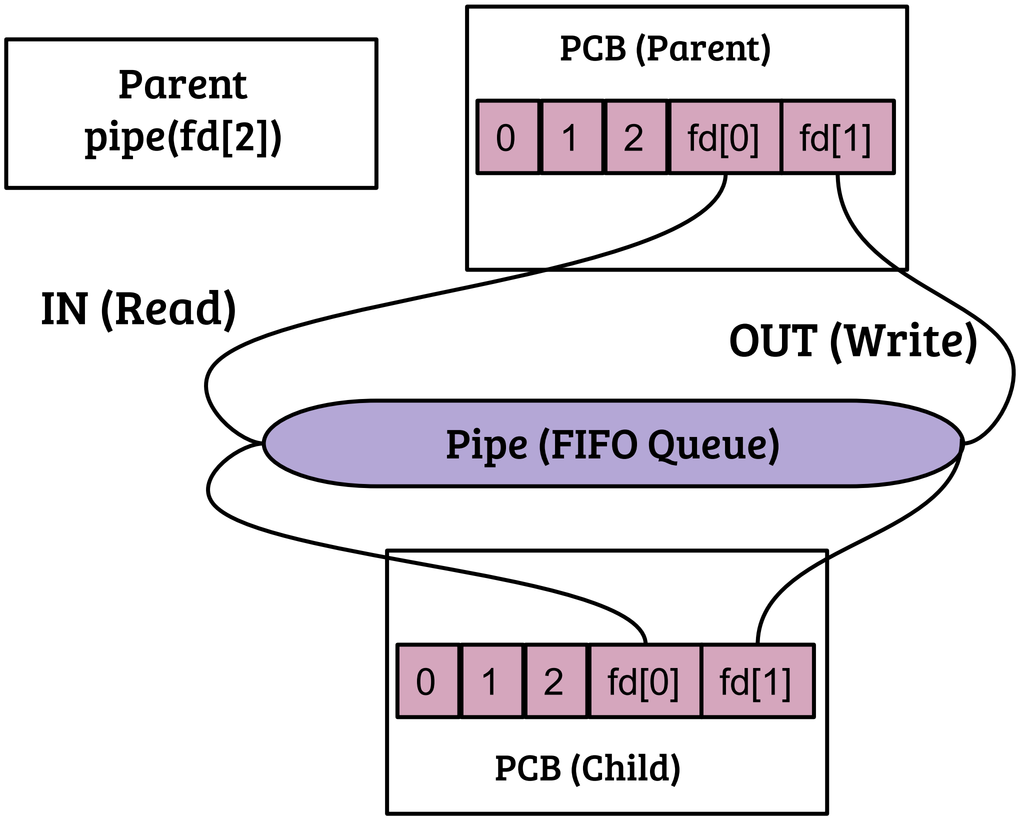
```
fd = open ("tmp.txt")

close( 1); close(2);      // close STDOUT and STDERR

dup(fd); dup(fd)          // 1→ fd, 2 → fd

exec(ls )
```

UNIX pipe( ) system call



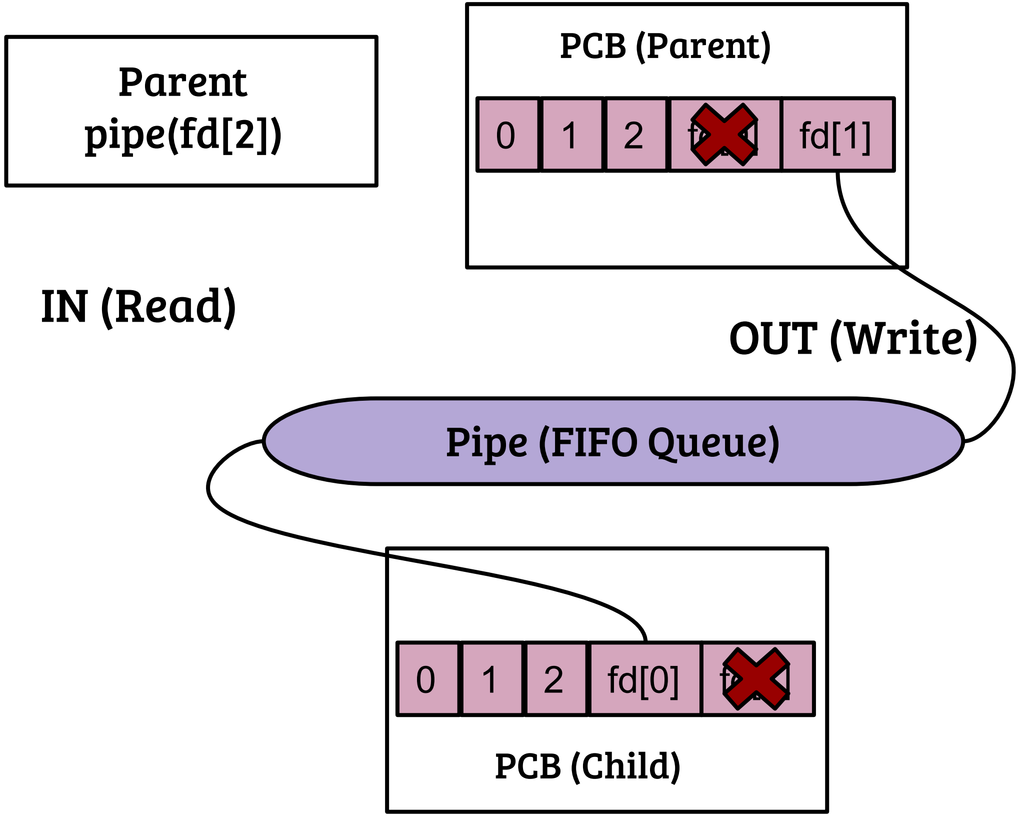
- pipe( ) takes array of two FDs as input
- *fd[0]* is the read end of the pipe
- *fd[1]* is the write end of the pipe
- Implemented as a FIFO queue in OS

UNIX pipe( ) with fork( )



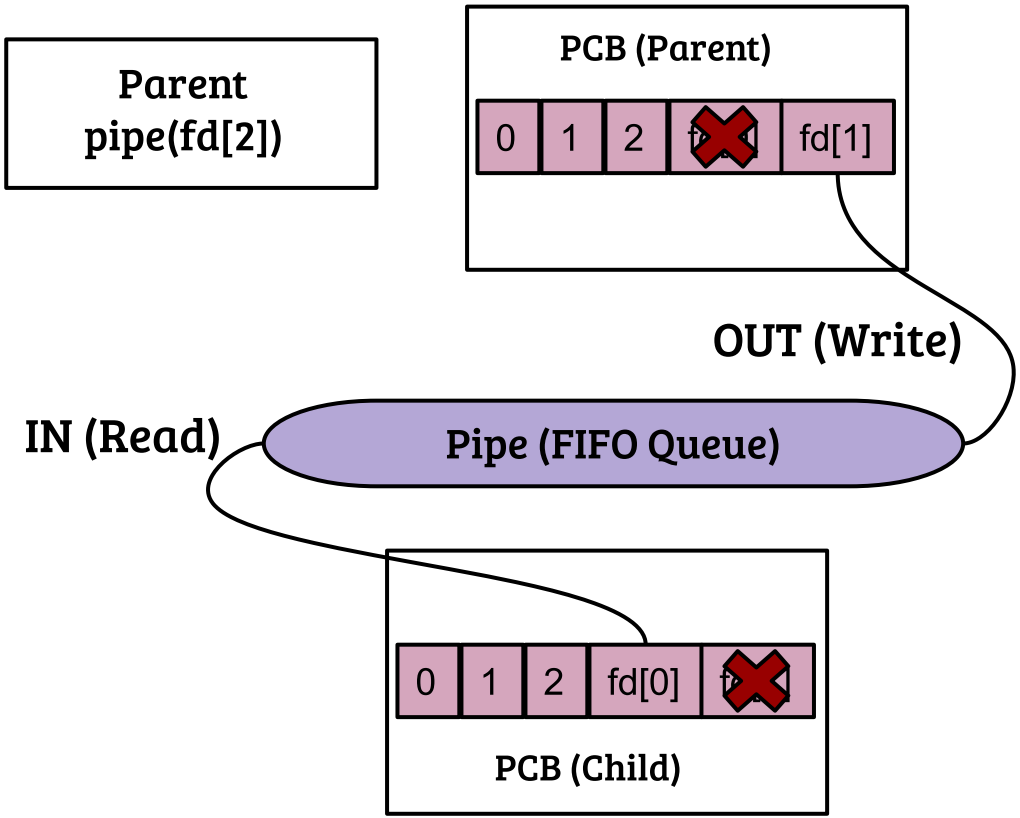
- fork( ) duplicates the file descriptors
- close( ) one end of the pipe, both in child and parent
- Result: a queue between parent and child

UNIX pipe( ) with fork( )



- fork( ) duplicates the file descriptors
- close( ) one end of the pipe, both in child and parent
- Result
  - A queue between parent and child
  - A cleaner queue

Shell piping : ls | wc -l



- pipe( ) followed by fork( )
- exec("ls") after closing STDOUT and duping OUT fd of pipe
- exec("wc") after closing STDIN and duping IN fd of pipe
- Result: input of "wc" is connected to output of "ls"