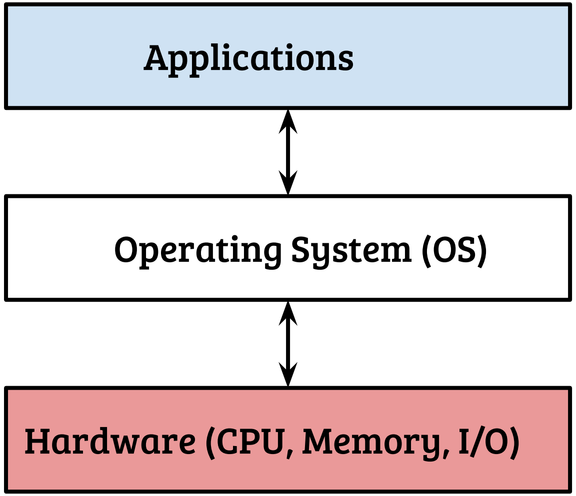
CS330: Operating Systems

Introduction

What is an Operating System?



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

What if we skip this layer? Will there be any problems at all?

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

Oh! Need a computer to show my skills.

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.

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

Oh! Need a computer to show my skills.

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

I know logic gates to ISA

Can build a small computer for my program!

What is the role of the OS?

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)

What if we skip the OS layer?

Applications

Library API

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

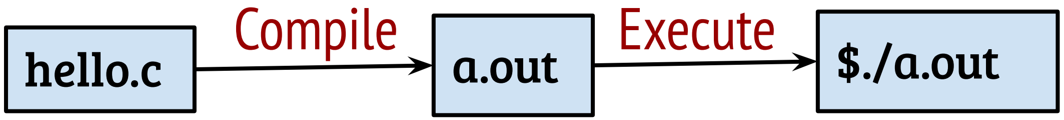
High-level programming language

Hardware (CPU, Memory, I/O)

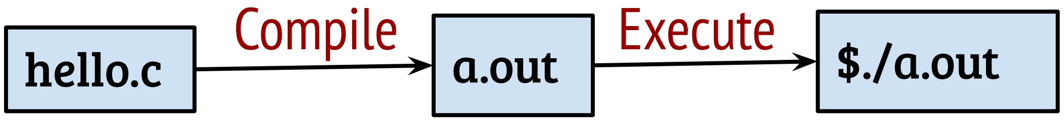
I understand only assembly code.

Conclusion: do not need the OS. Hang-on, may be there is something else!

Program execution

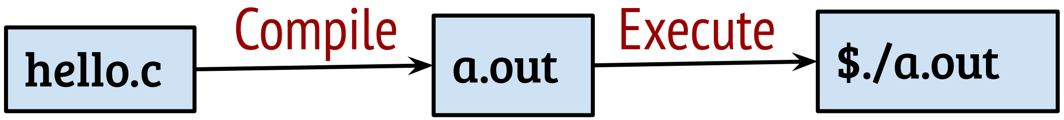


Program execution



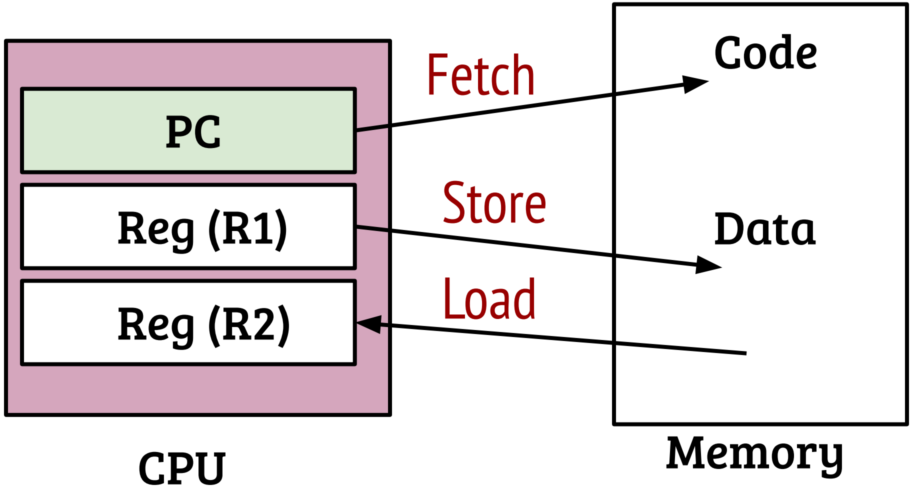
You said only CPU can execute!

Inside program execution



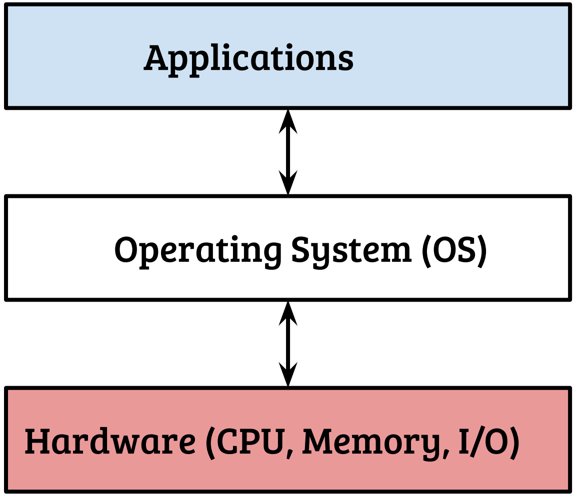
You said only CPU can execute!

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 (How?)

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

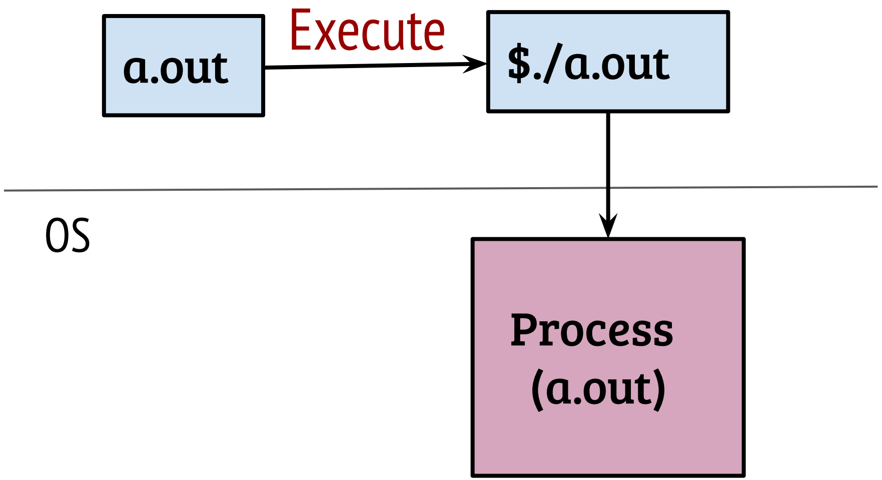
Agenda: CPU $\longrightarrow$ Process (OSTEP Ch4)

The process abstraction

- The OS creates a process when we run an executable

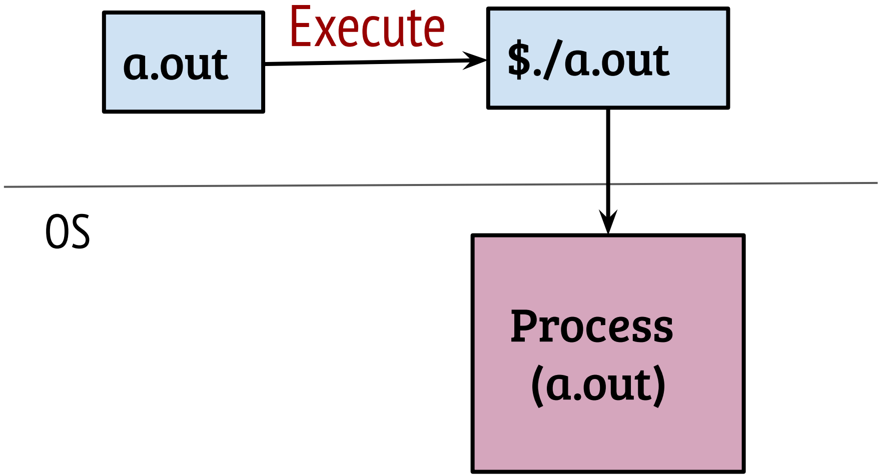
The process abstraction

- The OS creates a process when we run an executable



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

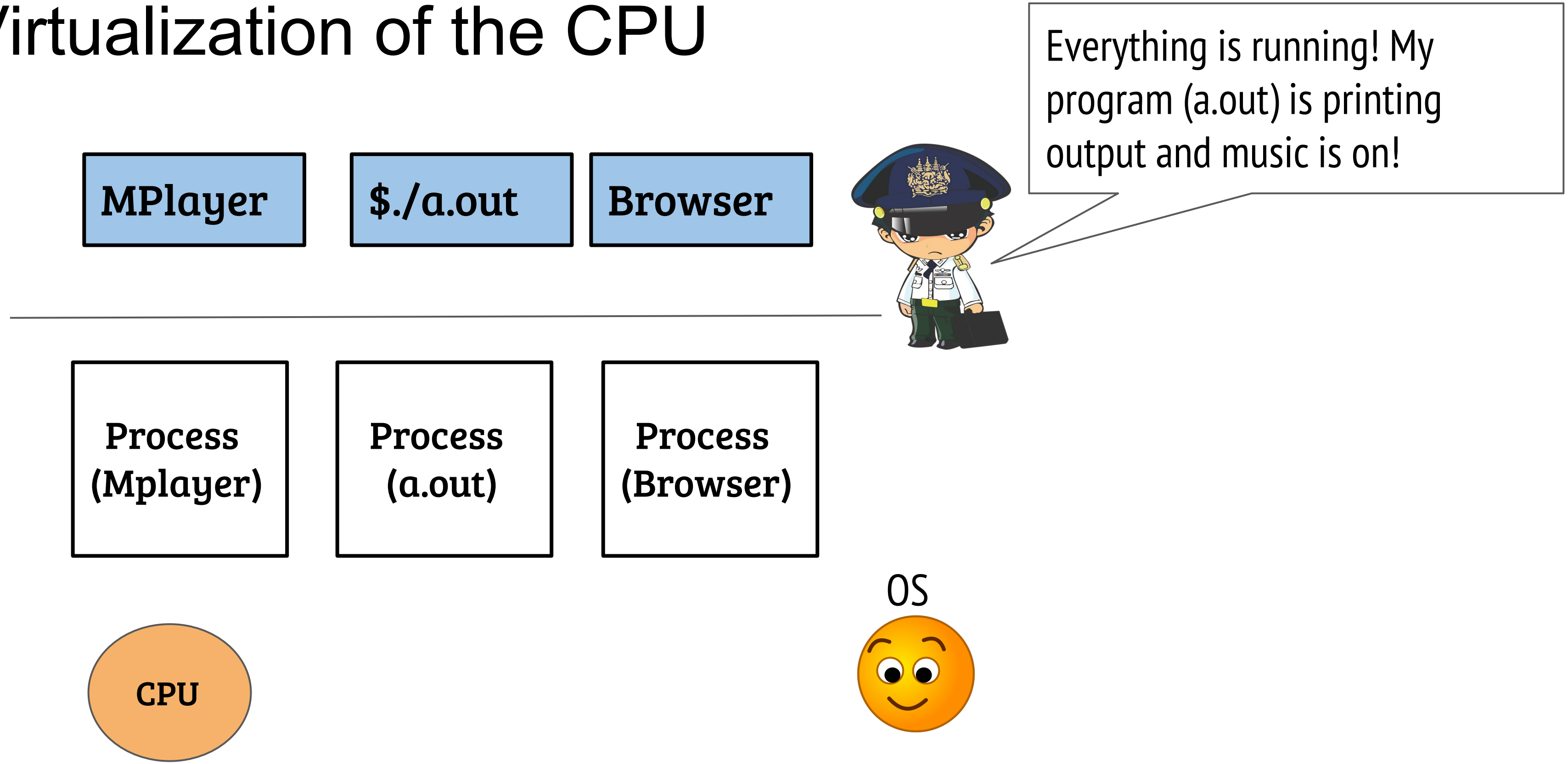
- The OS creates a process when we run an executable
- Alternatively, *A program in execution is called a process*
- Program is persistent while process is volatile
    - Program is identified by an executable, process by a PID
- Program ⟶ Process   (1 to N)
    - Many concurrent processes can execute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

MPlayer | $./a.out | Browser

Process (Mplayer) | Process (a.out) | Process (Browser)

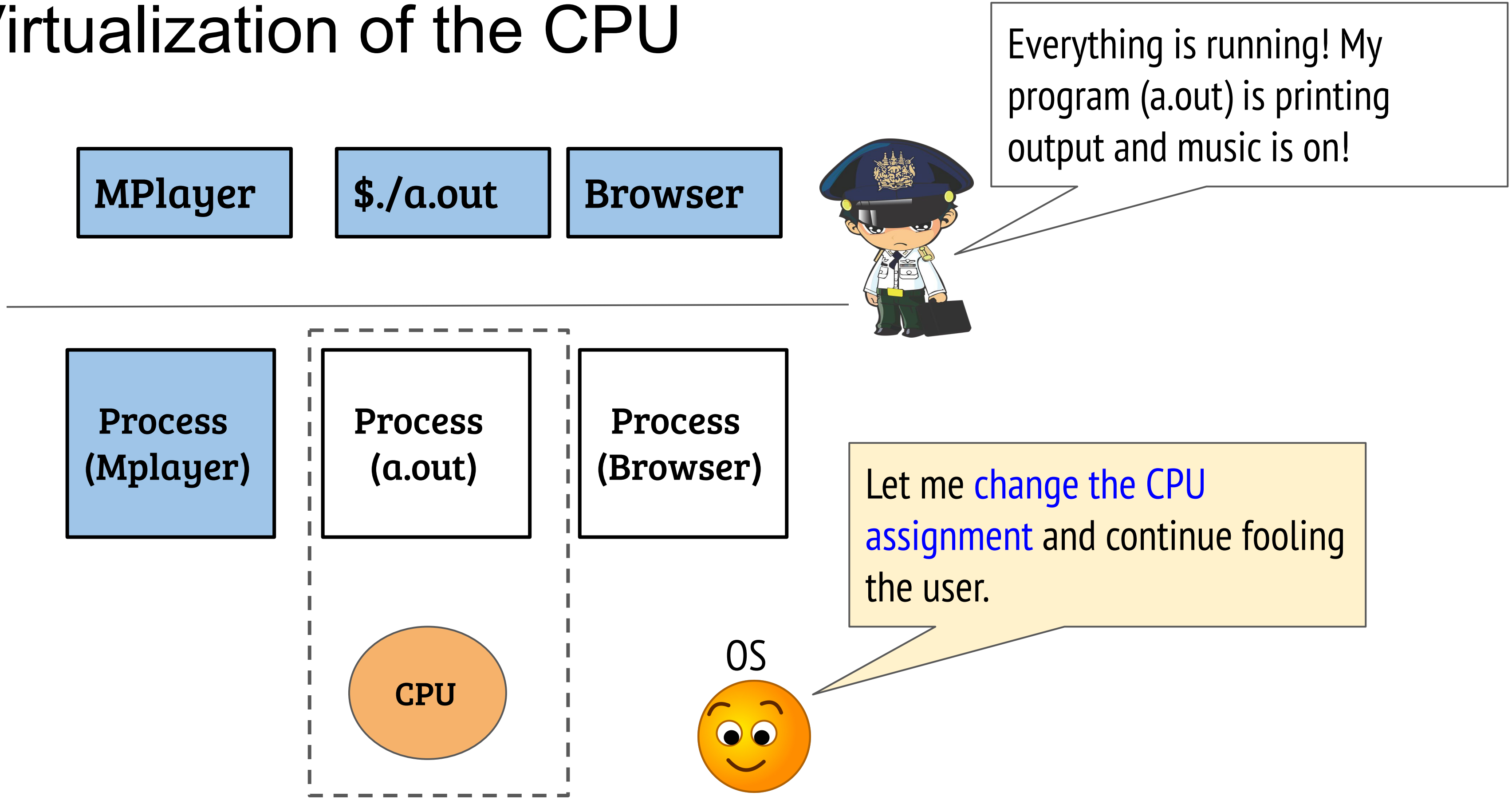
CPU

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

OS

CPU is actually assigned to MPlayer. Who cares! I have fooled the user.

Virtualization of the CPU

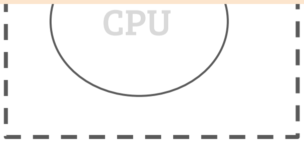


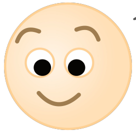
Virtualization of the CPU

MPlayer | $./a.out | Browser

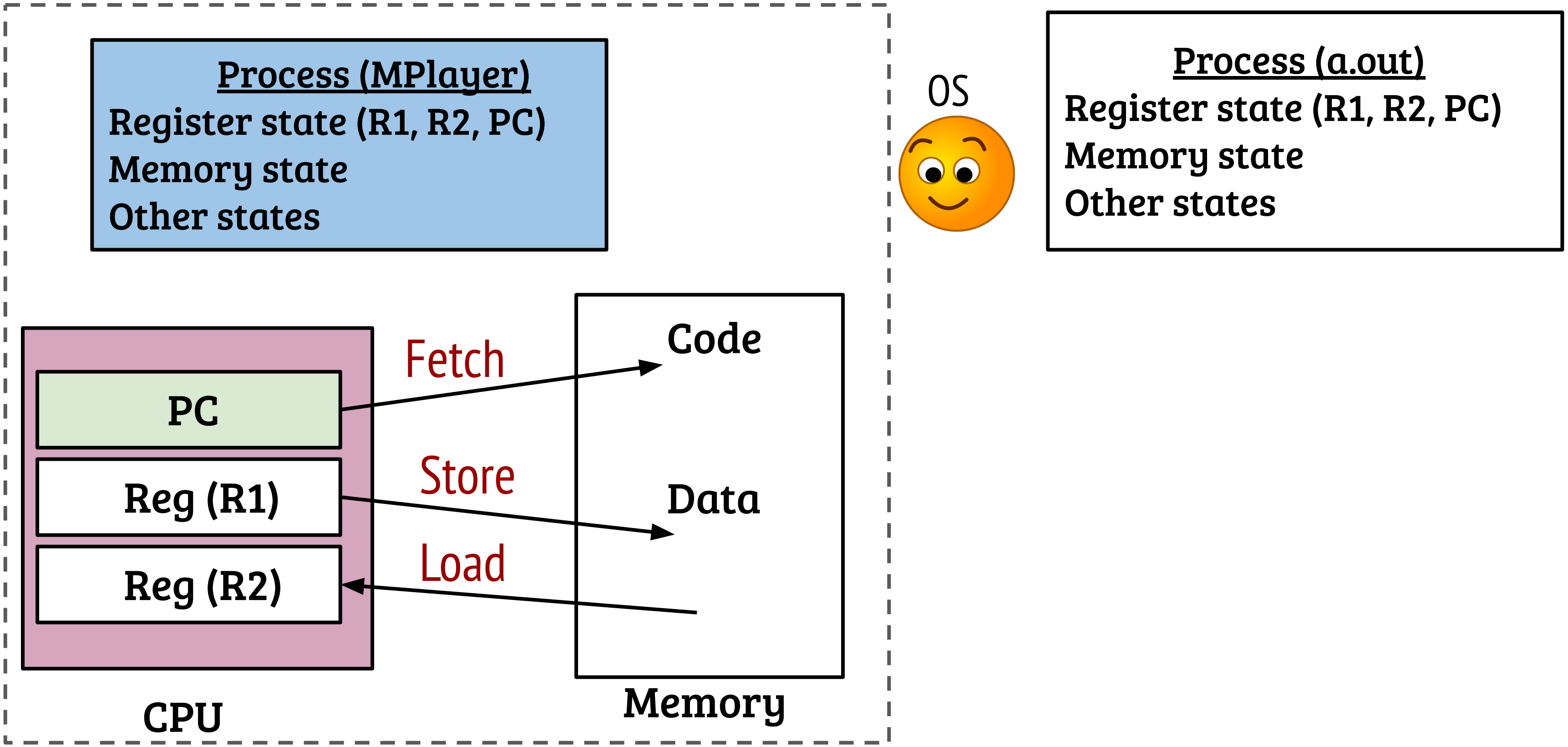
Everything is running! My program (a.out) is printing output and music is on!

- 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?

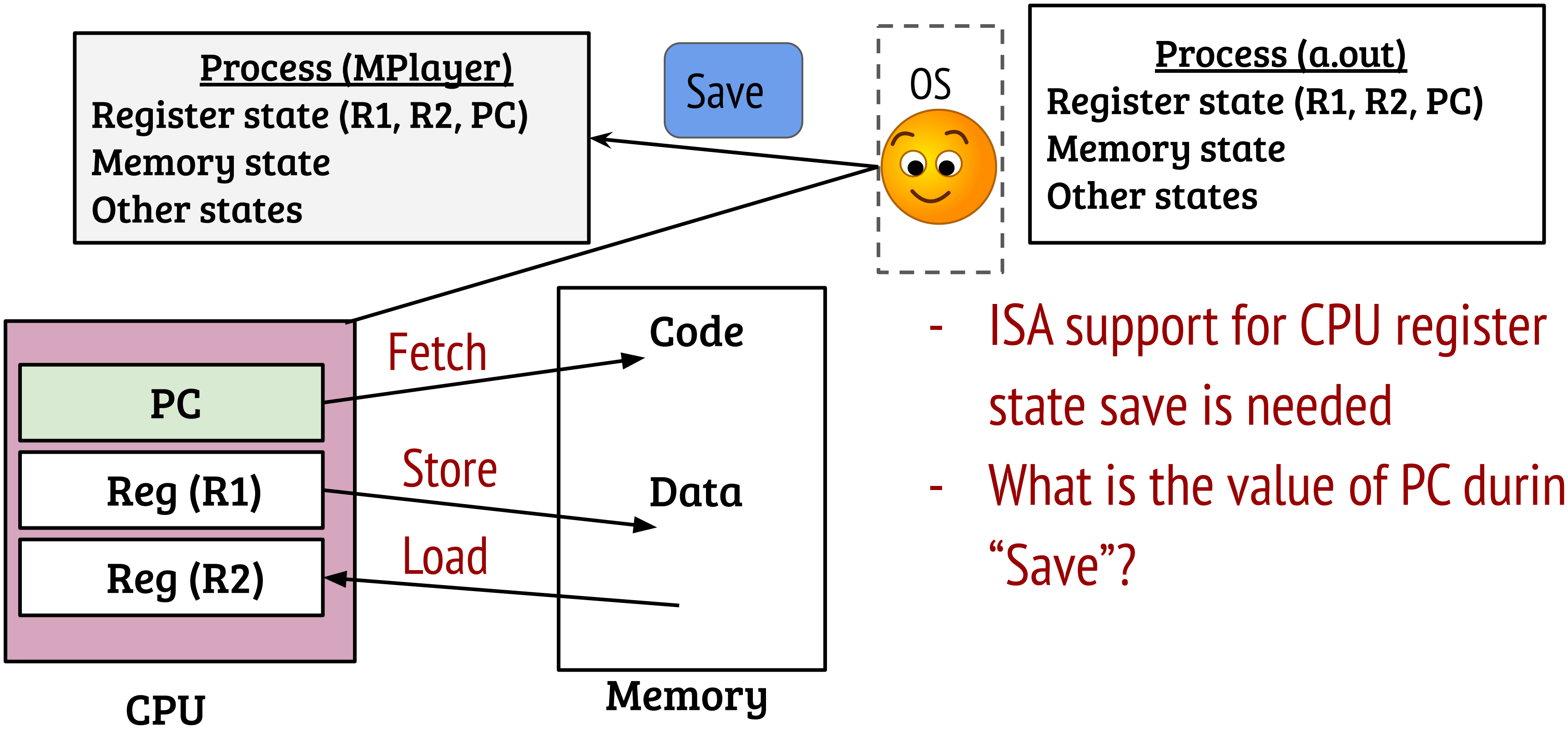




Context switch: state of a process

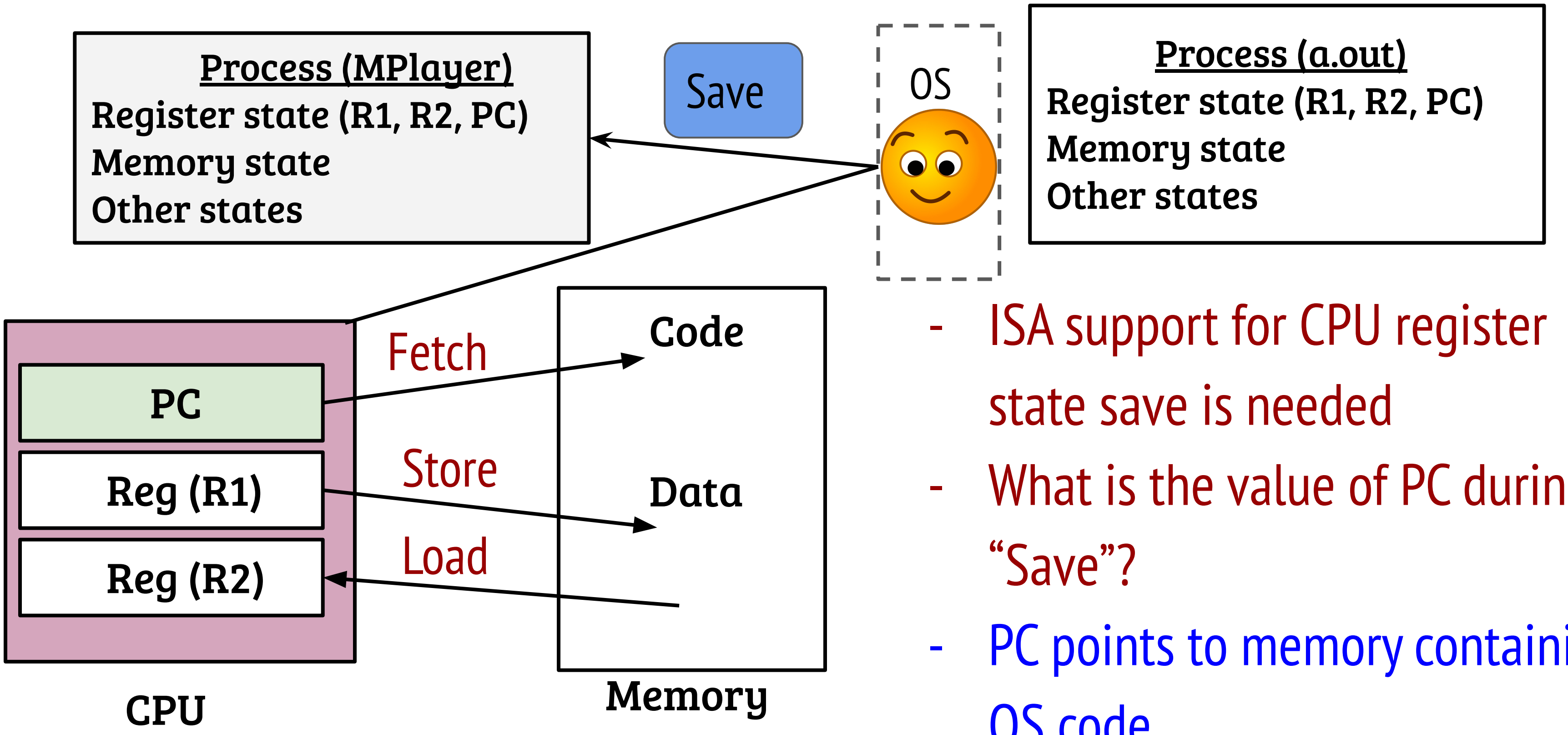


Context switch: saving the state of outgoing process



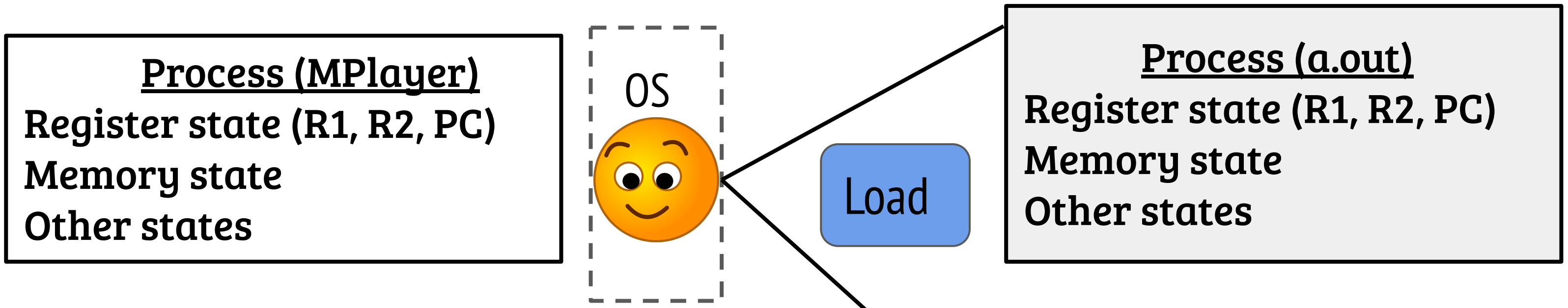
- ISA support for CPU register state save is needed
- What is the value of PC during "Save"?

Context switch: saving the state of outgoing process

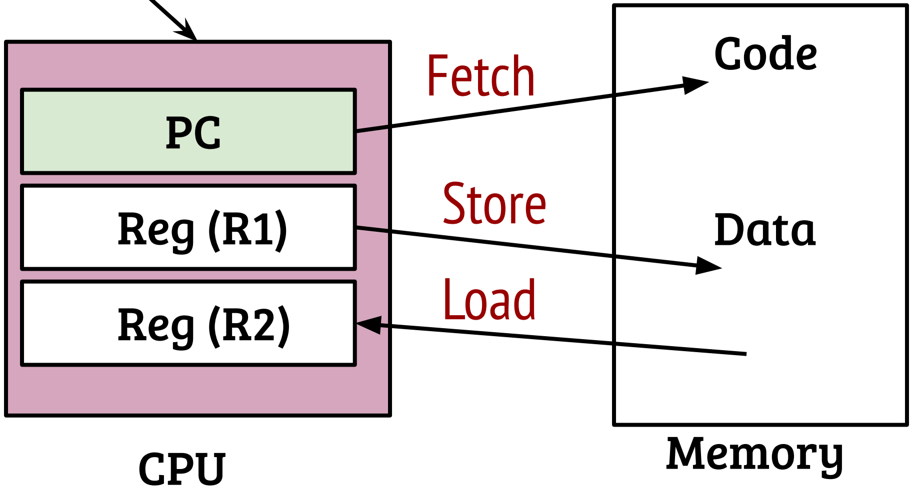


- ISA support for CPU register state save is needed
- What is the value of PC during “Save”?
- PC points to memory containing OS code

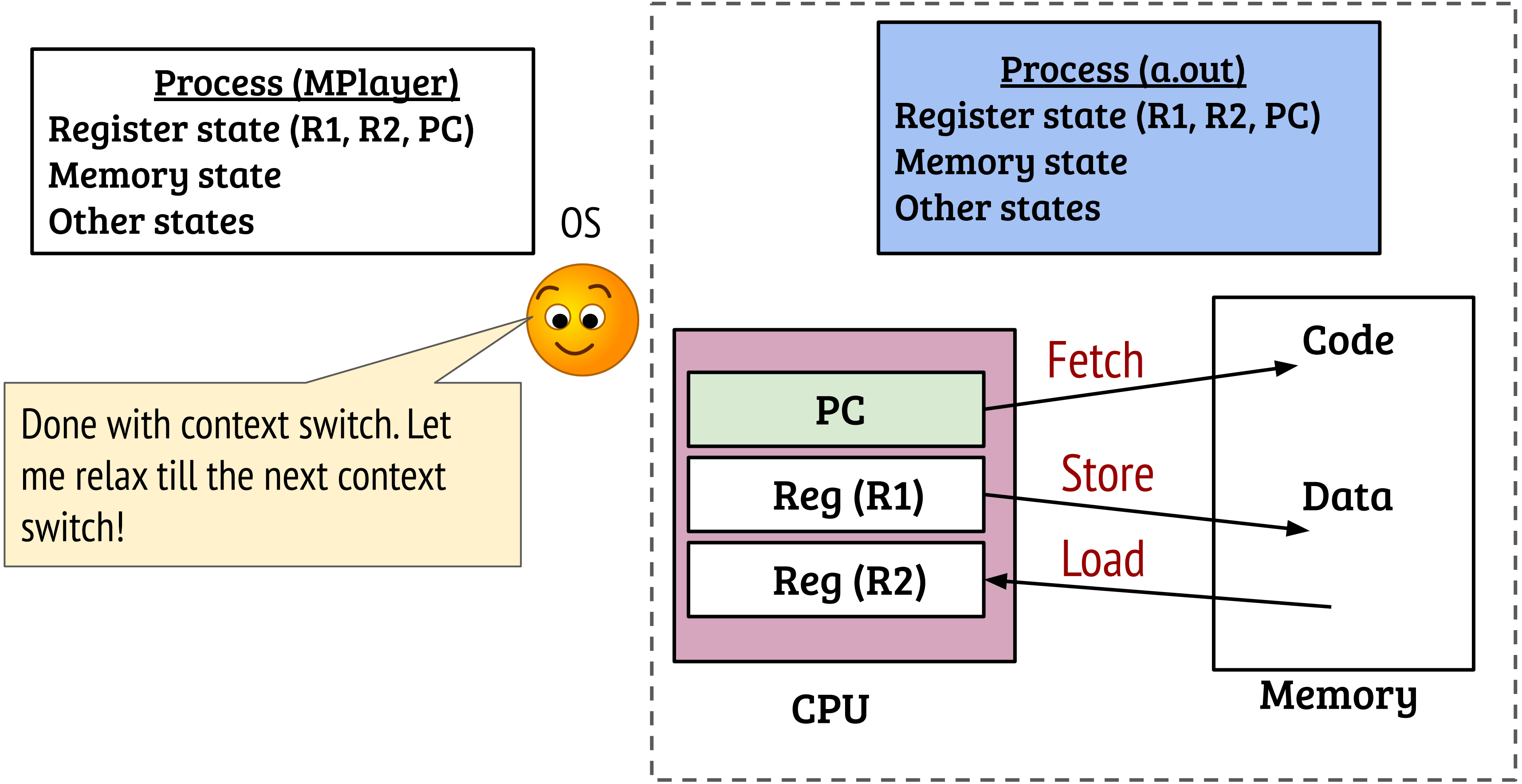
Context switch: load the state of incoming process



- 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

Everything is running! My program (a.out) is printing

- How CPU assignment is changed? (OR how context switch is performed?)
  - What happens to outgoing process? How does it come back?
- Using process scheduling, saving the *state* of the outgoing process and loading the *state* of the incoming process (will revisit)
- Overheads of context switch?
- State save and restore, cache effects
- How to decide the incoming process?
- OS implements different types of process scheduling policies

Hidden behind the abstraction!

- How does OS get the control of the 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
};
```

- All the registers shown here are used directly/indirectly during program execution
- General purpose registers (r8-r15, rax, rbx etc.) are used for storage and computation
  - Register allocation is an important aspect of a compiler

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 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
};
```

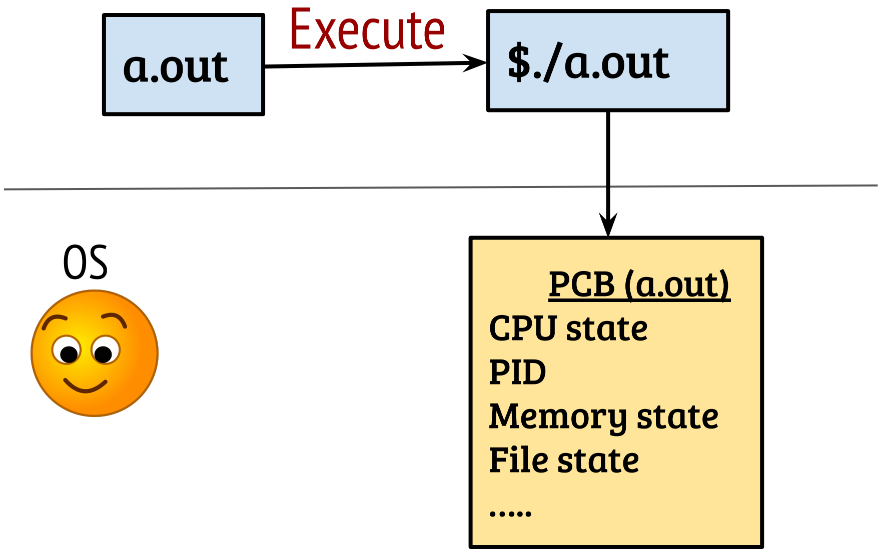
- What is a stack pointer in the context of hardware state?
- Points to the TOS address of a stack in memory, operated by *push* and *pop* instructions
- What is the use of stack?
- Makes it easy to implement function call and return, store local variables

CS330: Operating Systems

Process API: System calls

Recap: The process abstraction

- The OS creates a process when we run an executable

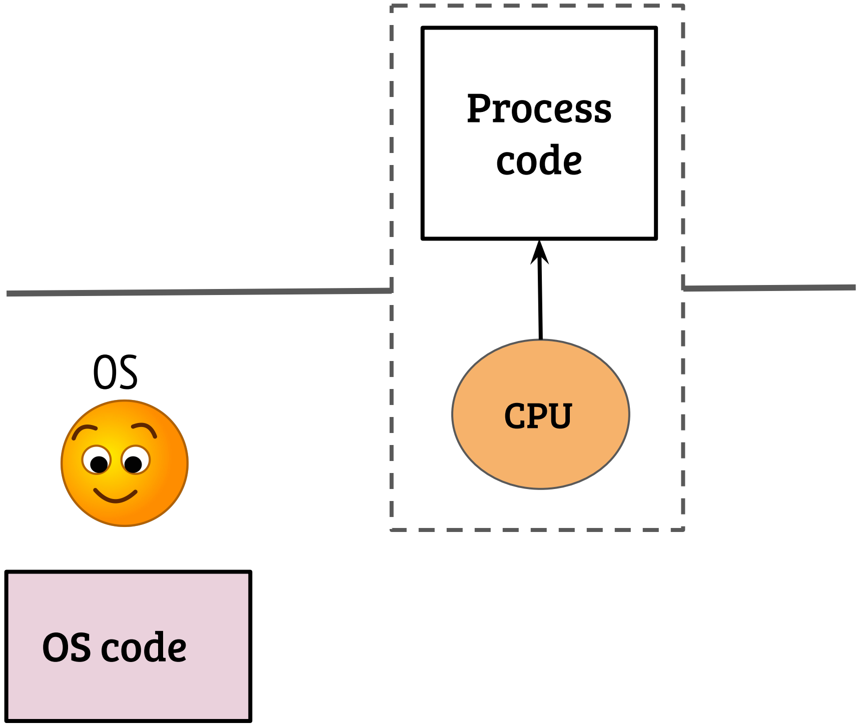


- When we execute "a.out" on a shell a process control block (PCB) is created
- Does it raise some questions related to the exact working?

Process creation: What and How?

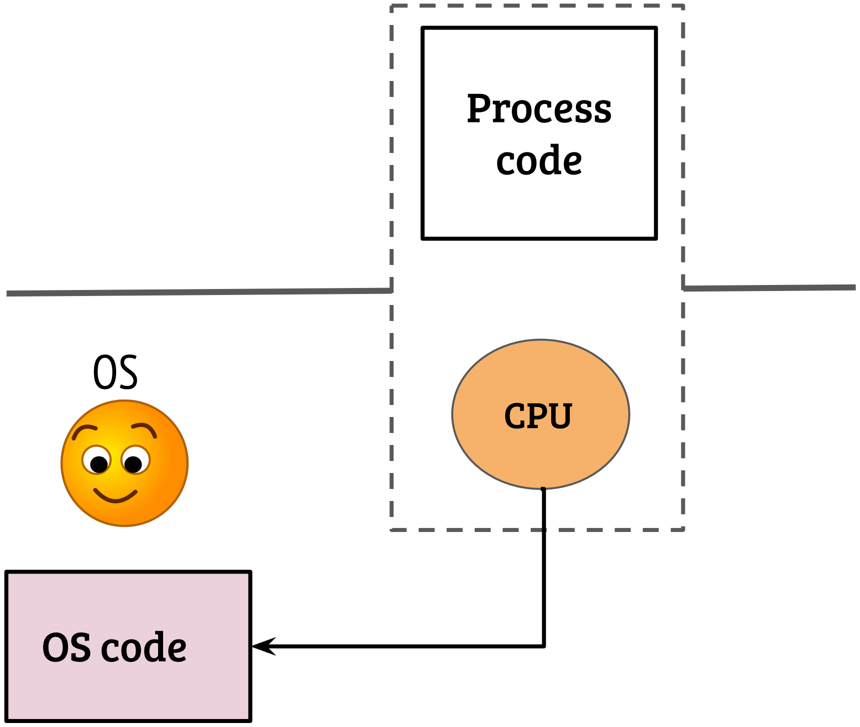
- How does OS come into action after typing "./a.out" in a shell?

System call



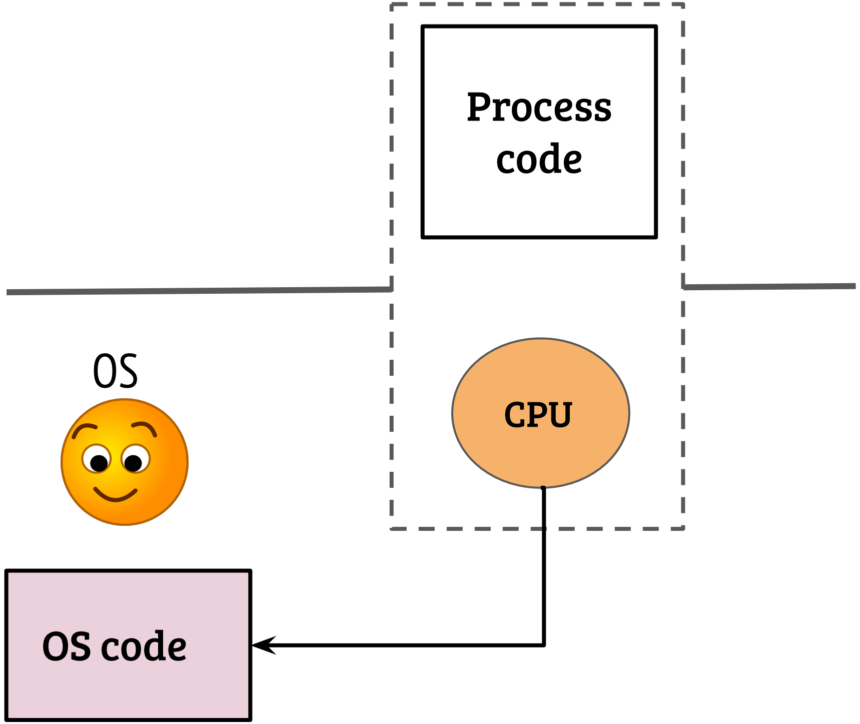
- CPU executing *user code* can invoke the *OS functions* using system calls

System call



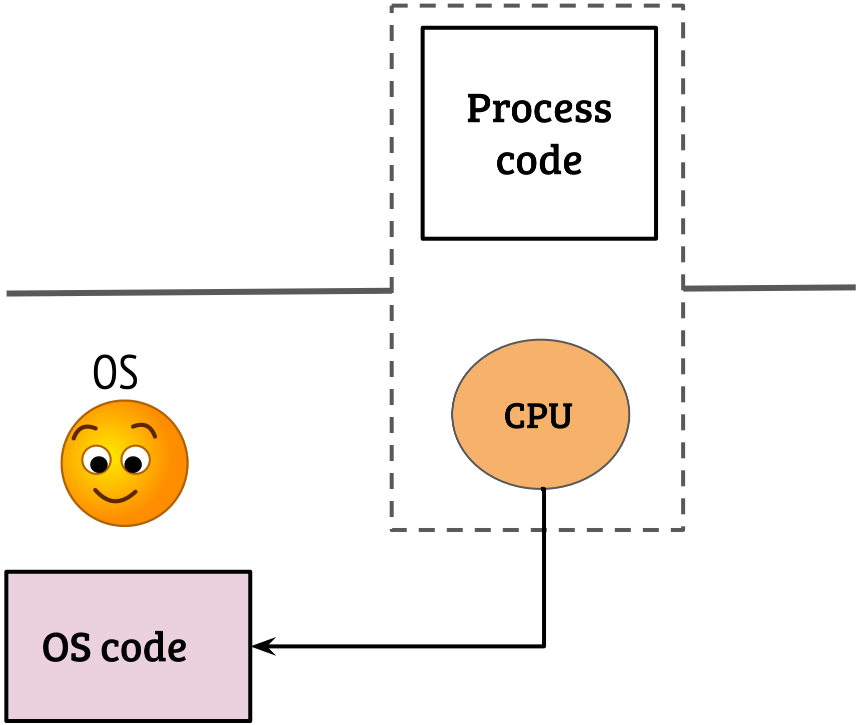
- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call

System call



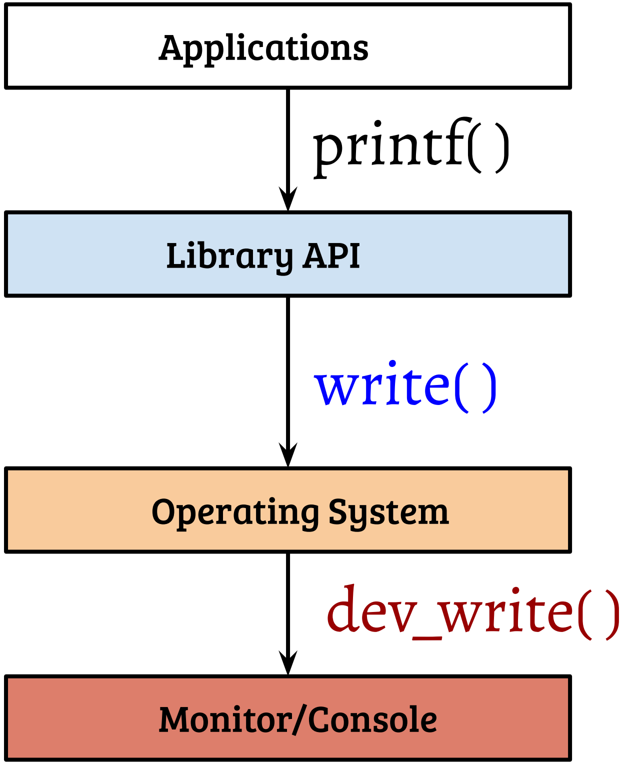
- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call
- How system call is different from a function call?

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)

System calls and user libraries



- Most system calls are invoked through wrapper library functions
- However, all system calls can be invoked directly
  - For example, in Linux systems, syscall( ) wrapper can be used (Refer: man syscall)

A simple system call: getpid( )

USER

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

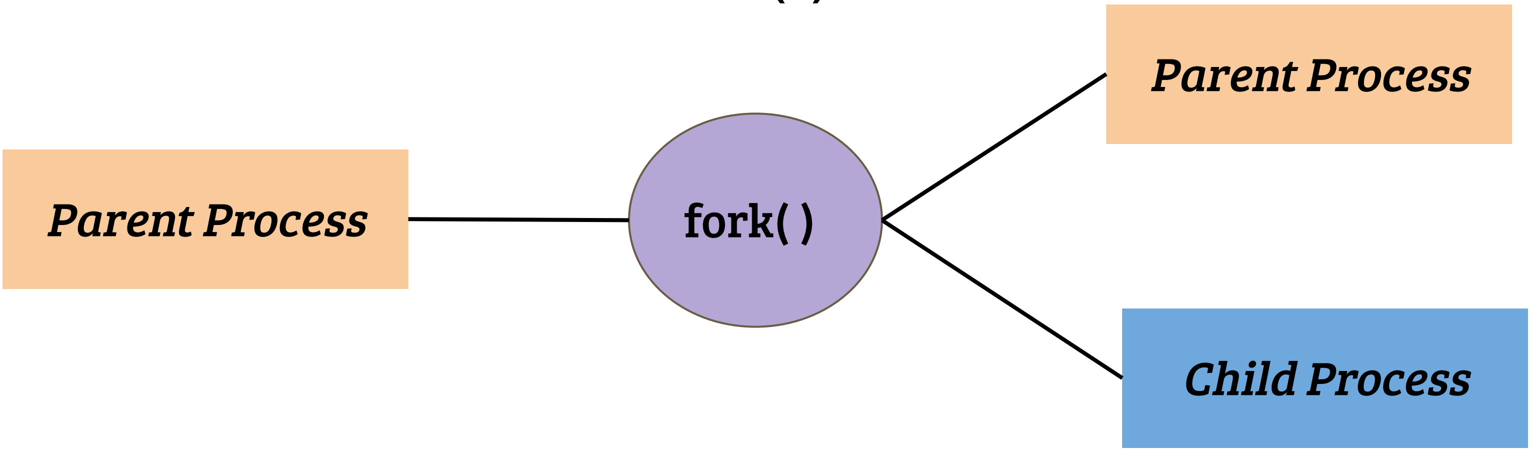
OS

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

Process creation: What and How?

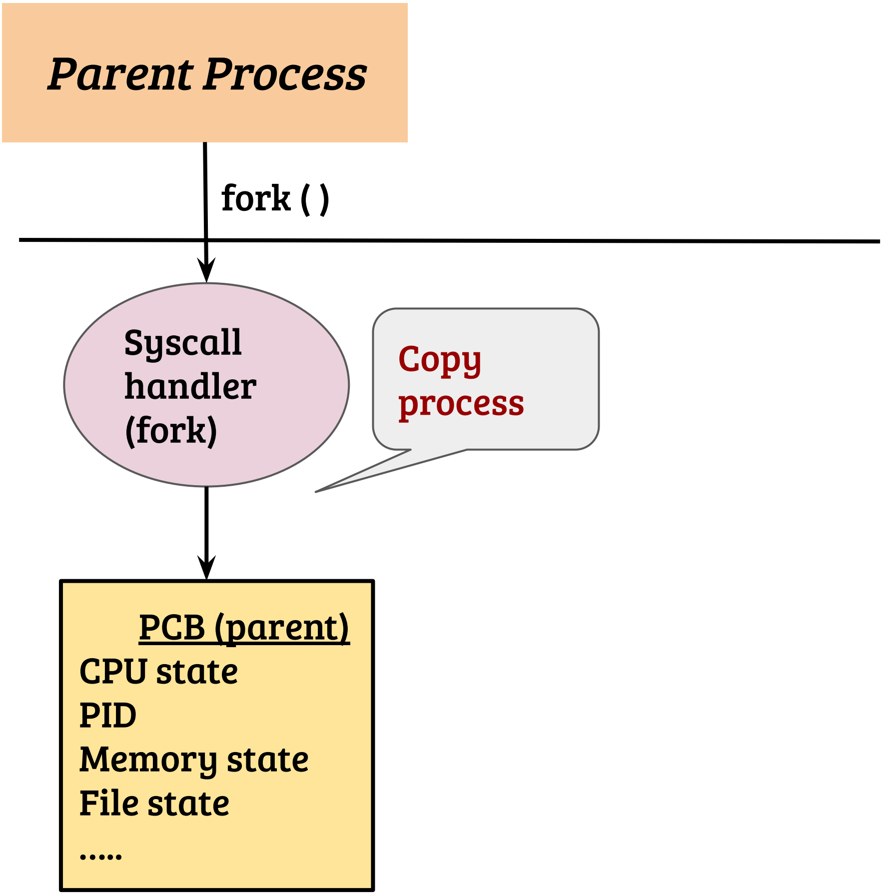
- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?

Process creation - fork( )

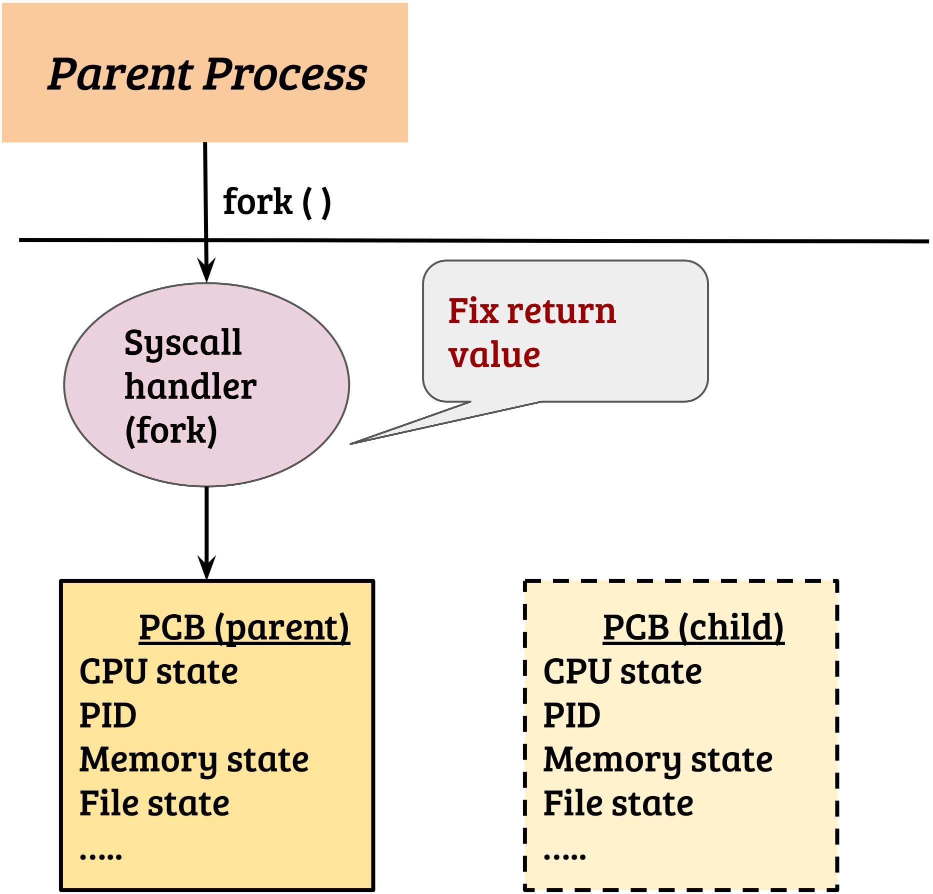


- fork( ) system call is weird; not a typical “privileged” function call
- fork( ) creates a new process; a *duplicate* of calling process
- On success, fork
    - Returns PID of child process to the caller (parent)
    - Returns 0 to the child

Typical implementation of fork

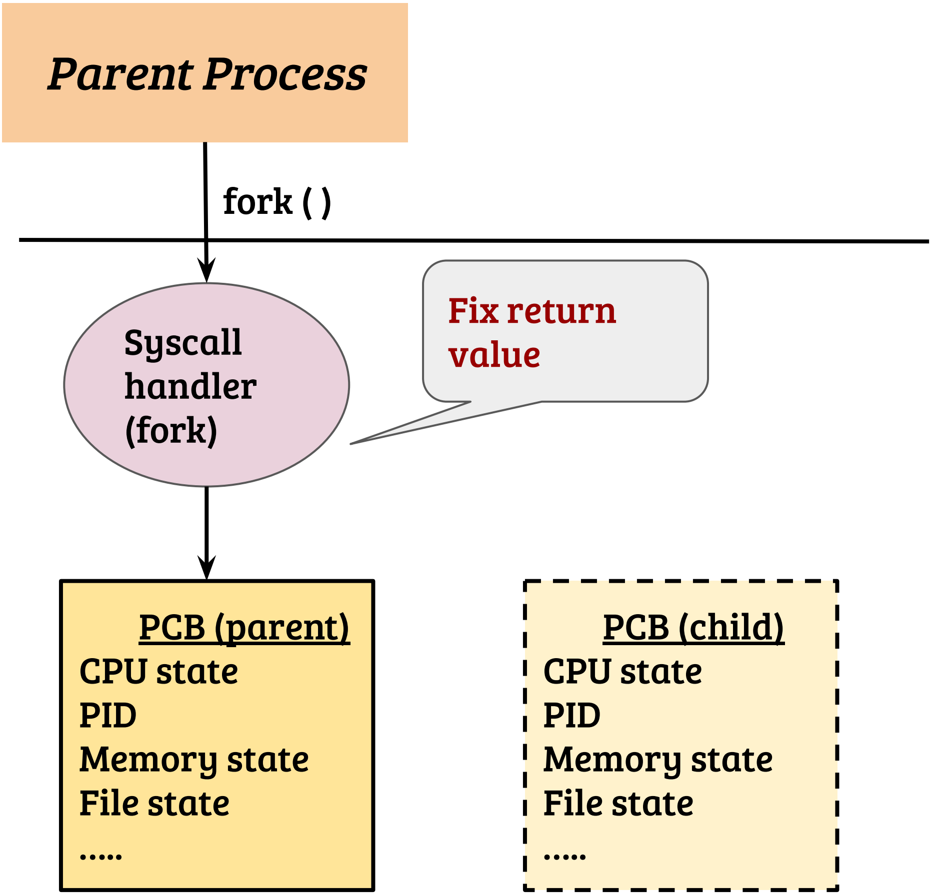


Typical implementation of fork



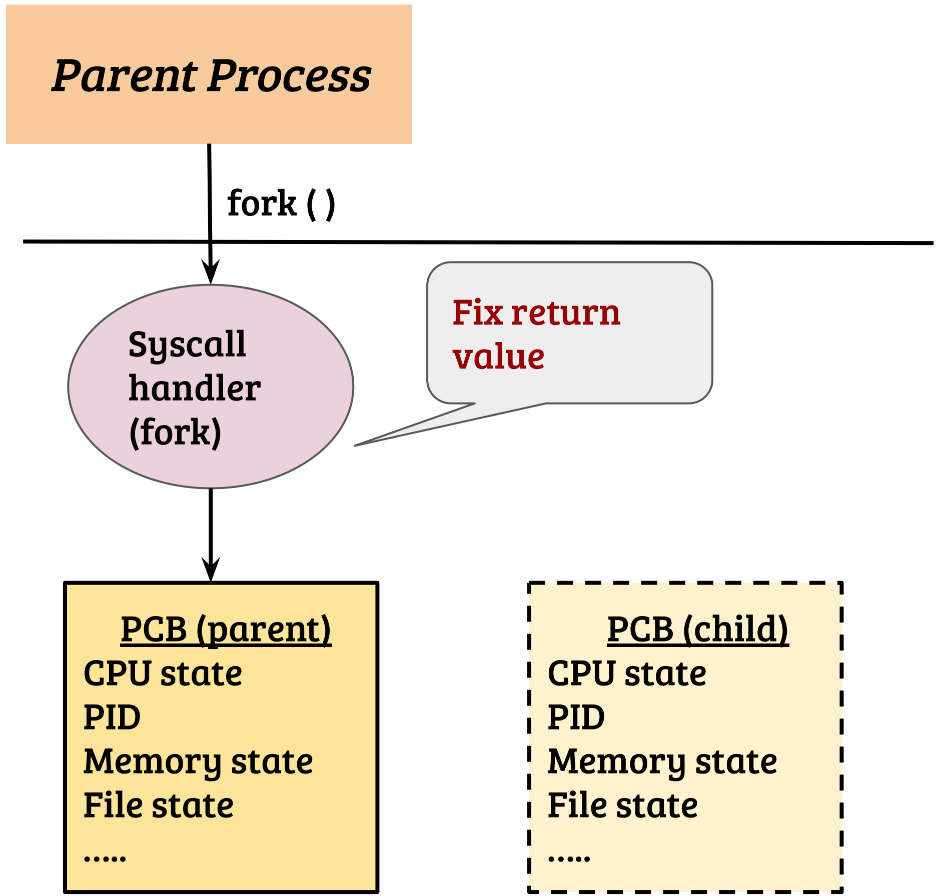
- Child should get ‘0’ and parent gets PID of child as return value. How?

Typical implementation of fork



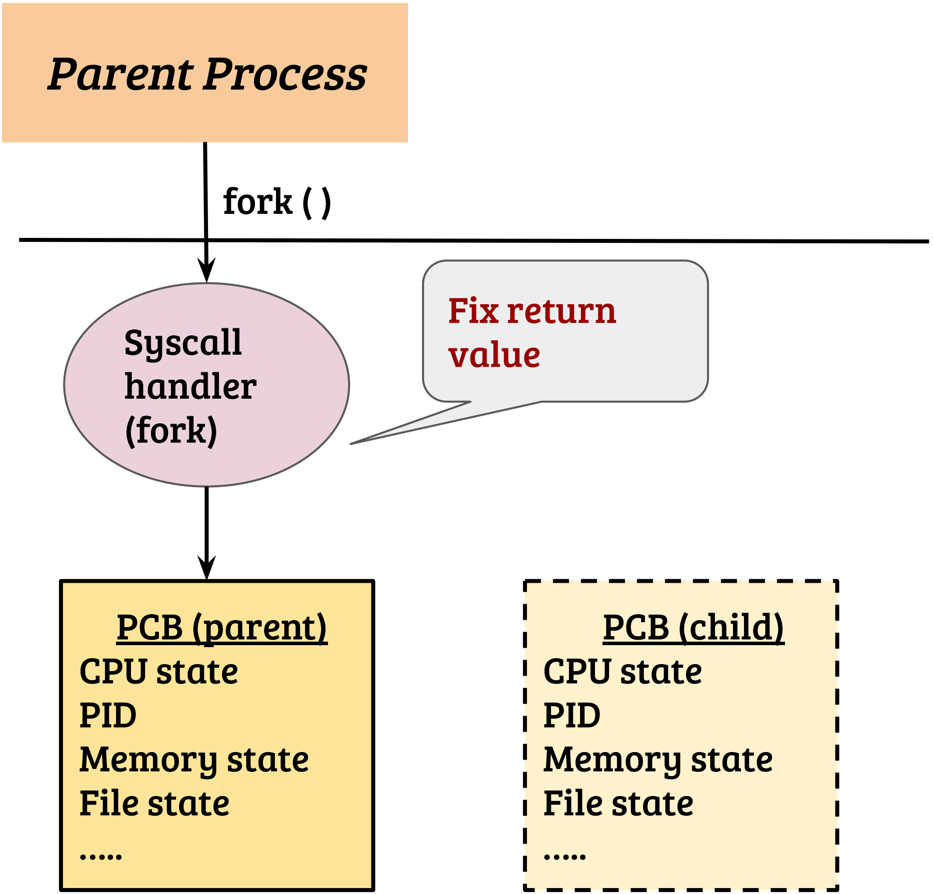
- Child should get ‘0’ and parent gets PID of child as return value. How?
- OS returns different values for parent and child

Typical implementation of fork



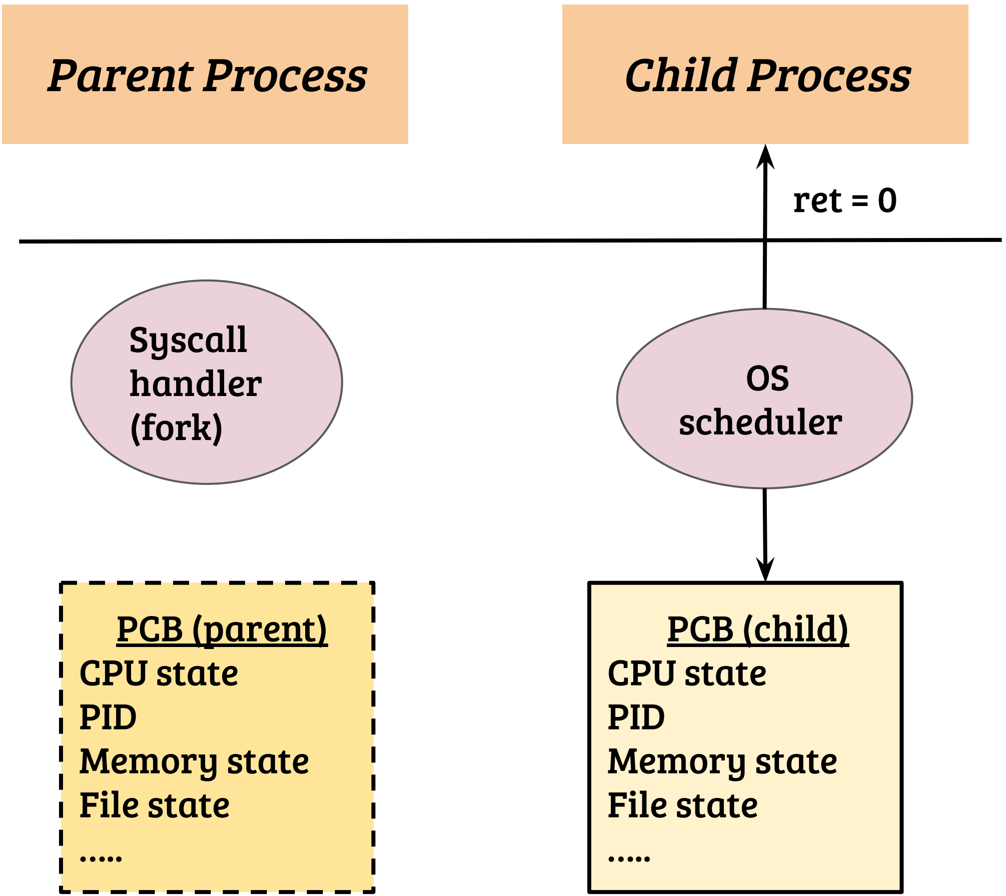
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?

Typical implementation of fork



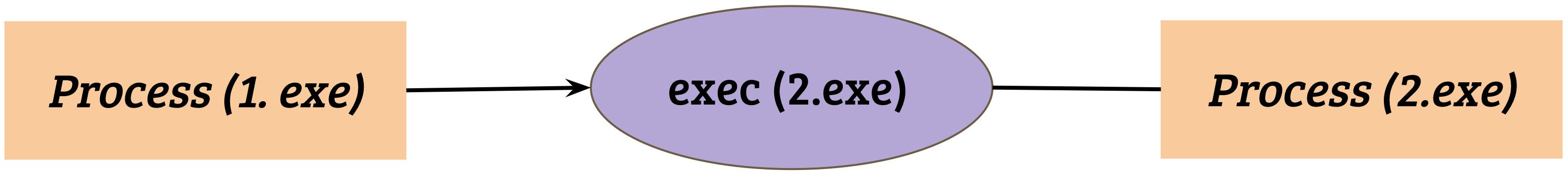
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?
- When OS schedules the child process

Typical implementation of fork



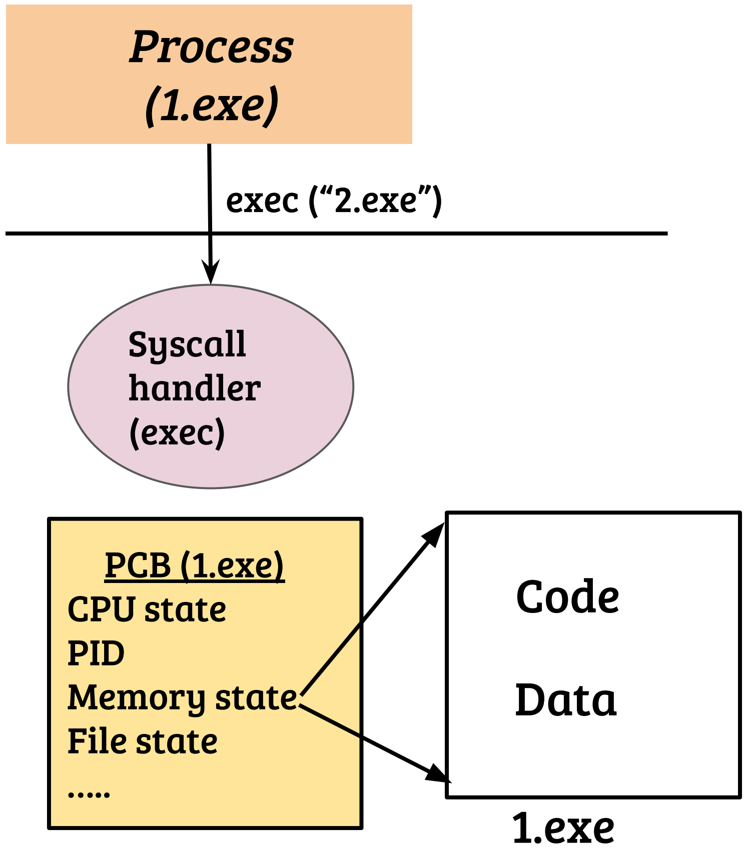
- PC is next instruction after fork( ) syscall, for both parent and child
- Child memory is an exact copy of parent
- Parent and child diverge from this point

Load a new binary - exec( )



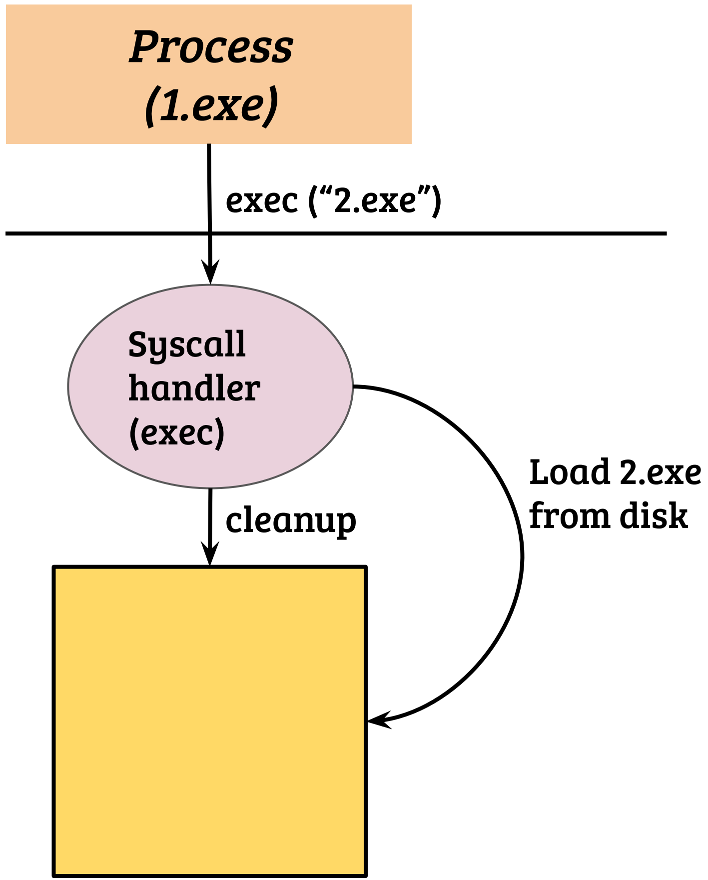
- Replace the calling process by a new executable
    - Code, data etc. are replaced by the new process
    - Usually, open files remain open

Typical implementation of exec



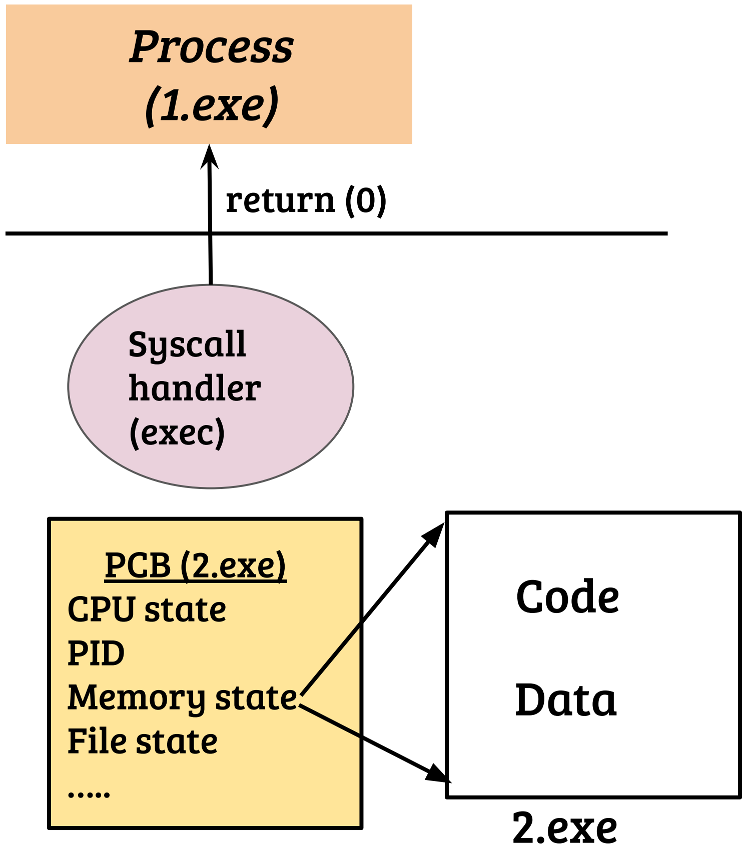
- The calling process commits self destruction! (almost)

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

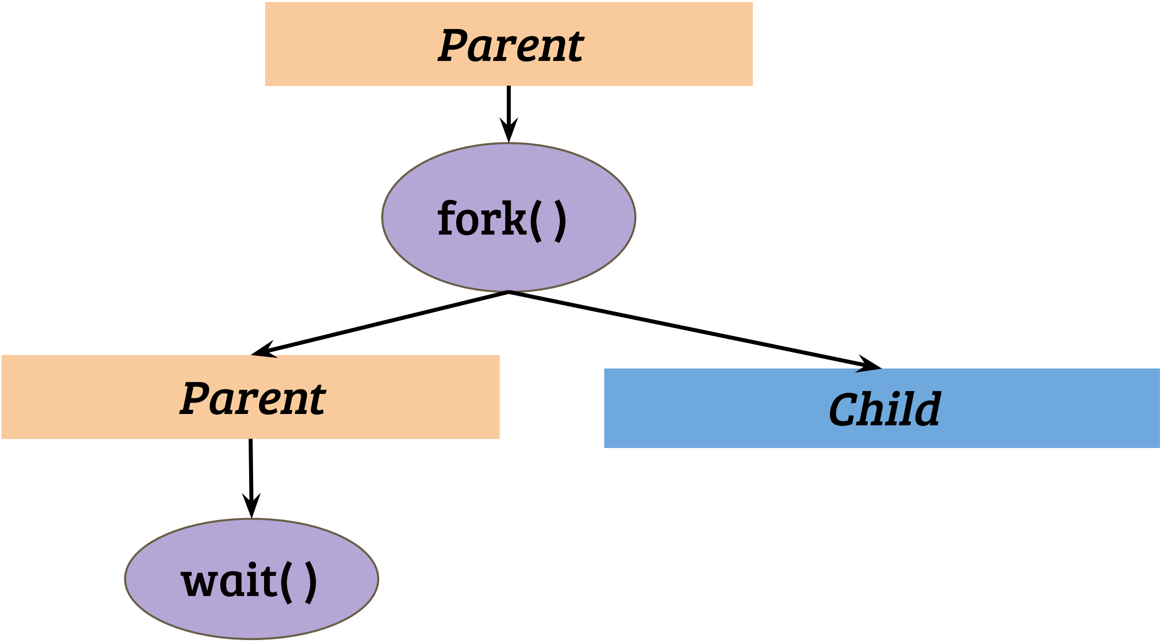


- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same
- On return, new executable starts execution
- PC is loaded with the starting address of the newly loaded binary

Process creation: What and How?

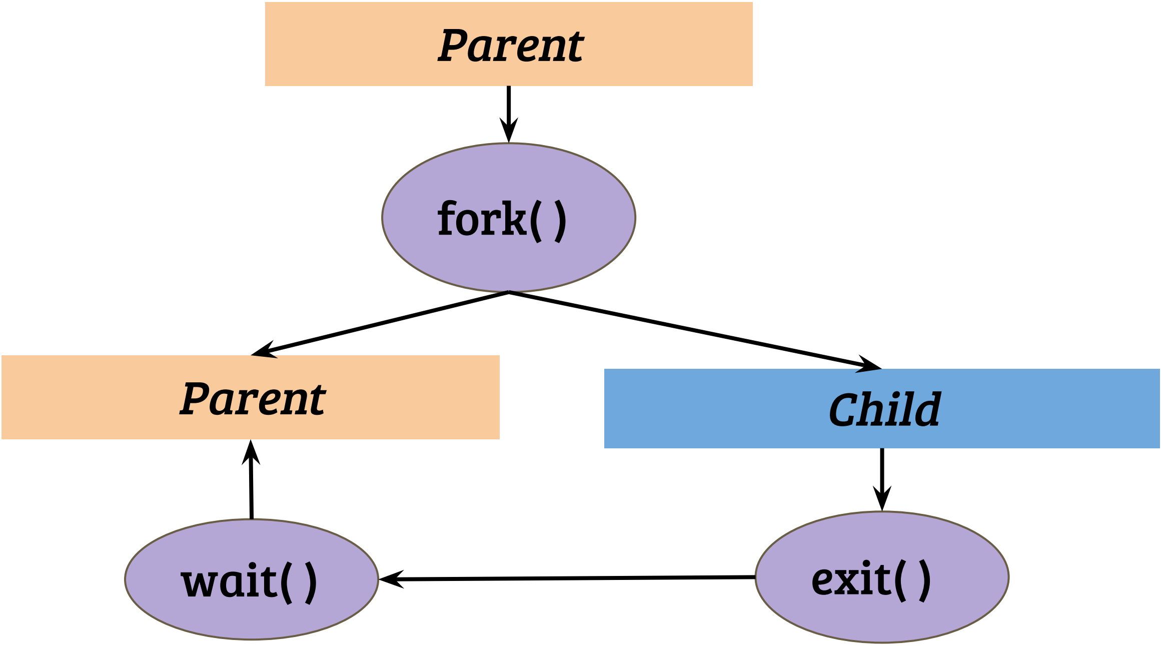
- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls? In what order?

wait( ) and exit( )



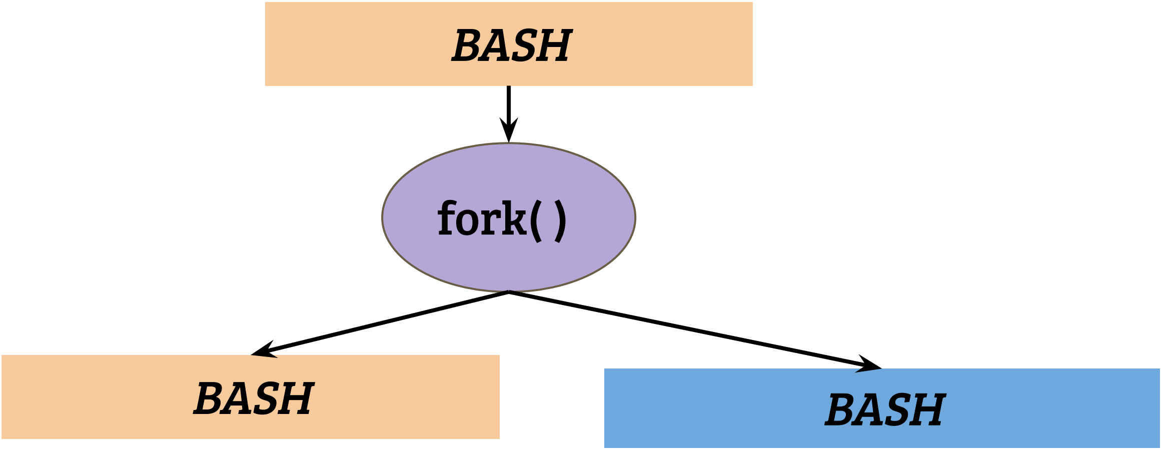
- The wait system call makes the parent wait for child process to exit

wait( ) and exit( )



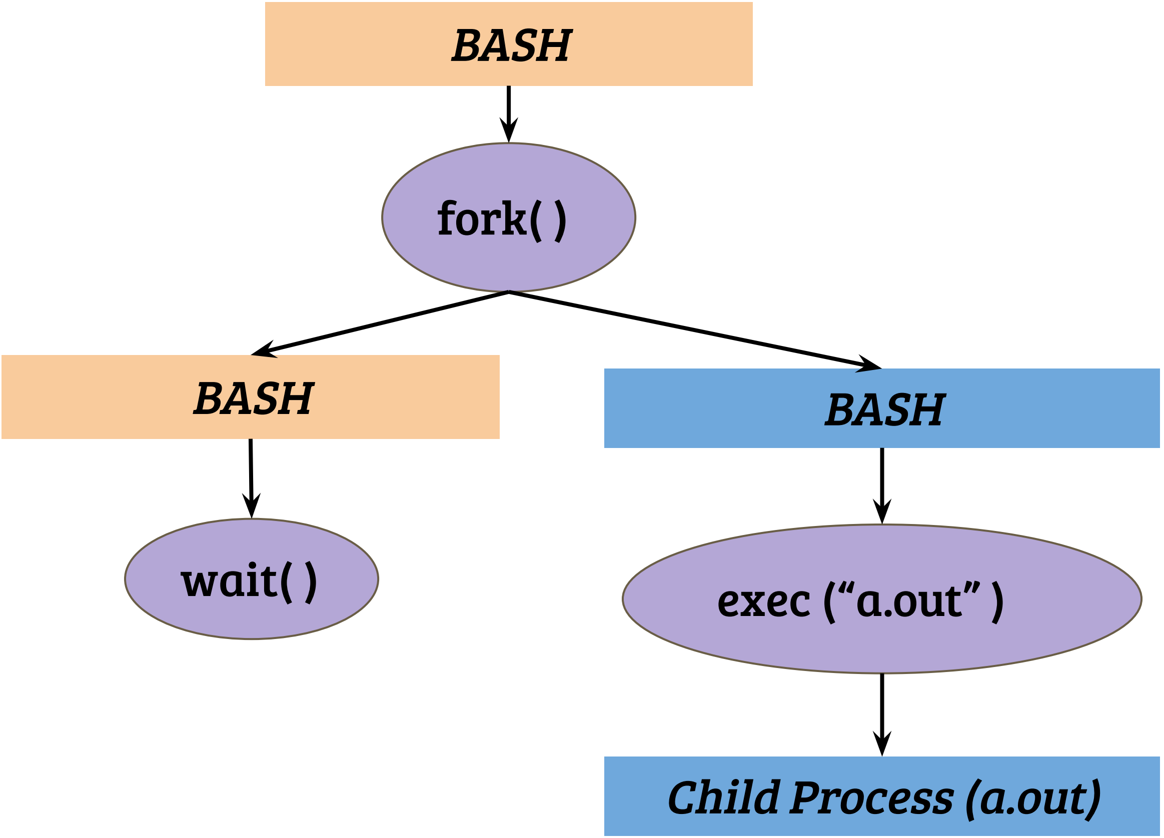
- The wait system call makes the parent wait for child process to exit
- On child exit( ), the wait() system call returns in parent

Shell command line: fork + exec + wait



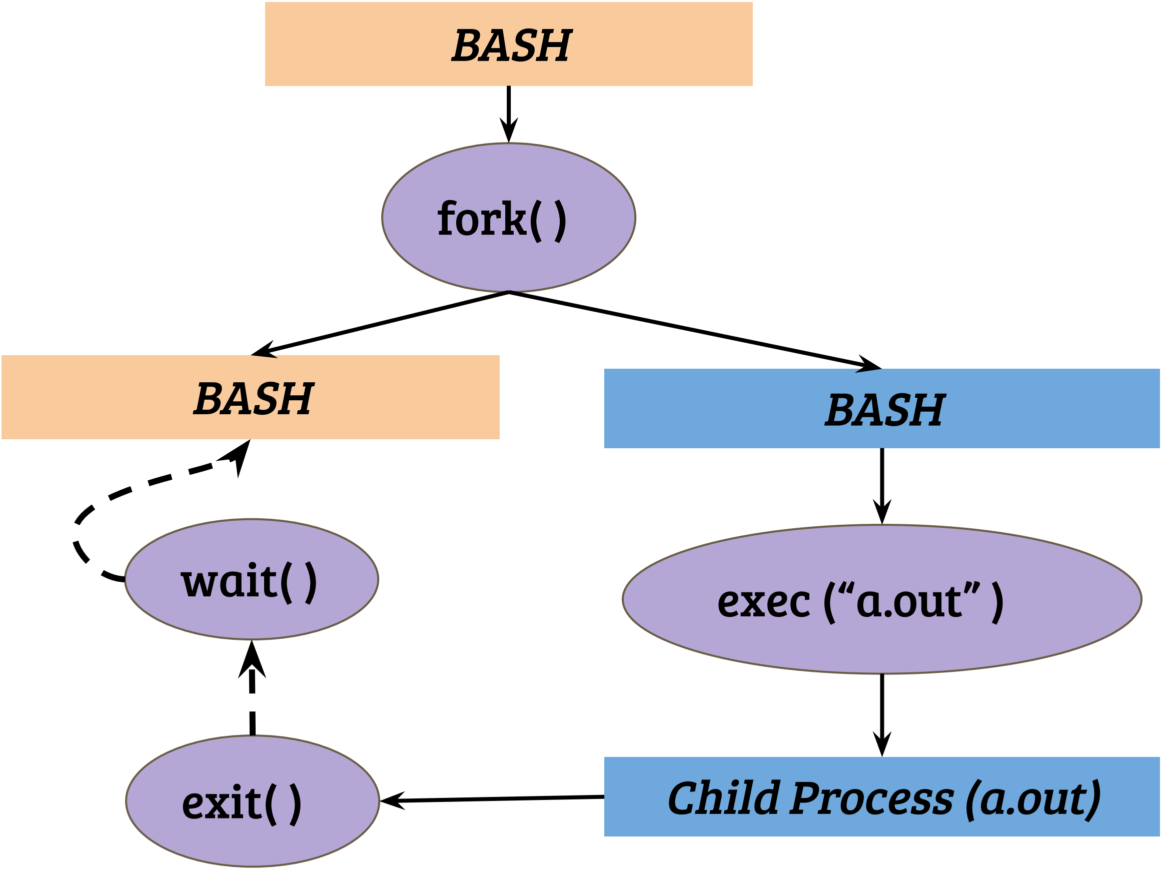
- The BASH process calls fork( )

Shell command line: fork + exec + wait



- Parent process calls wait( ) to wait for child to finish
- Child process invokes exec( )

Shell command line: fork + exec + wait

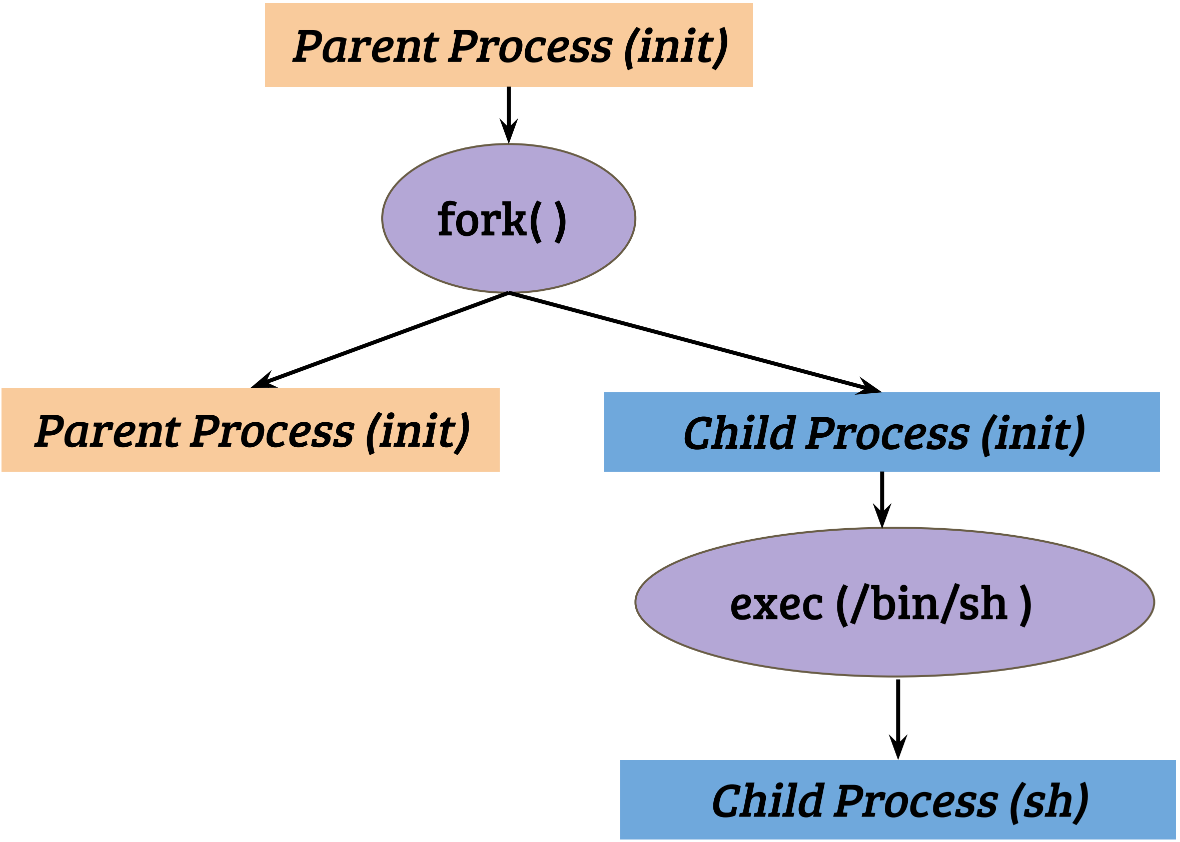


- When child exits, parent gets notified
- The BASH shell is ready for the next command at this point of time

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

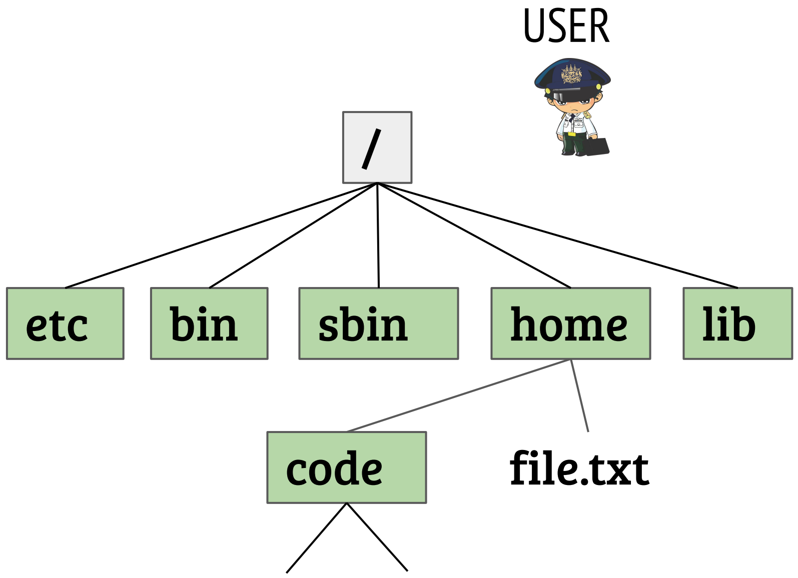
Process creation: What and How?

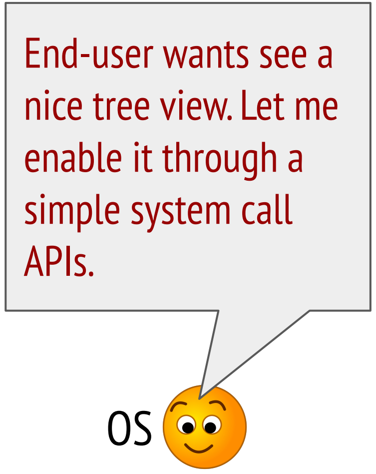
- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls?
- The shell process (bash process)
- What is the first user process?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

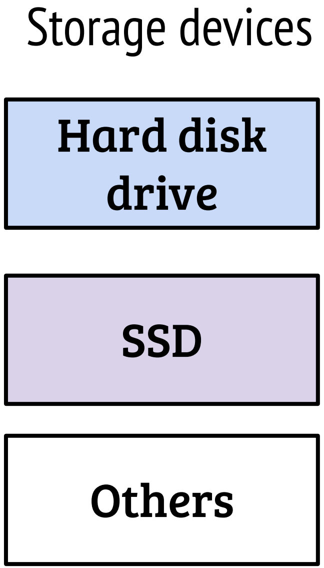
CS330: Operating Systems

Files

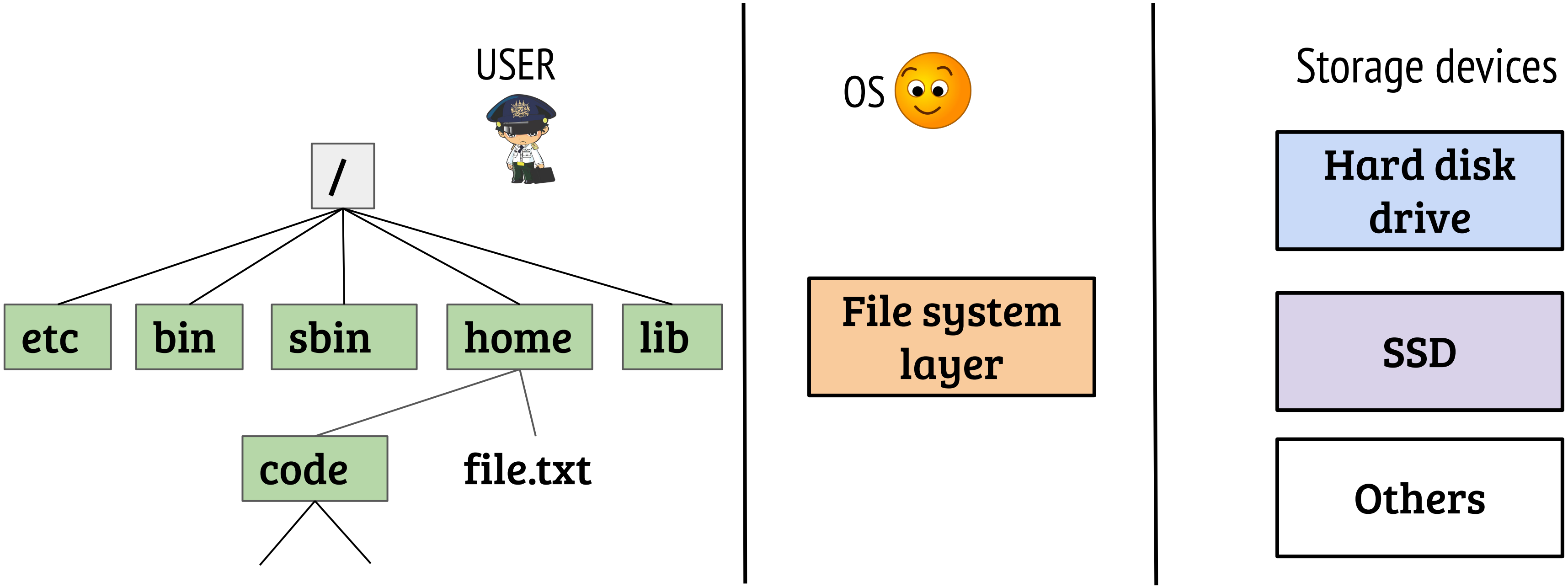
The file system





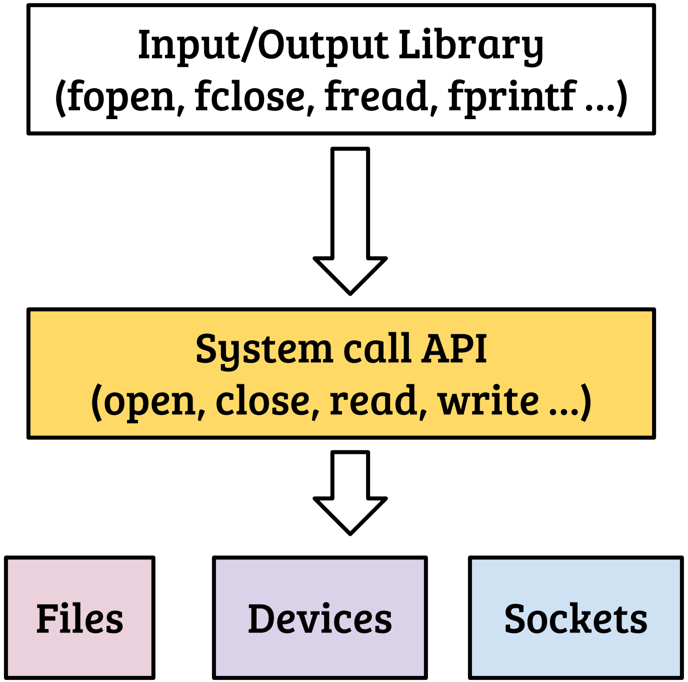


The file system



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

File system interfacing



- Processes 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.
- What is the mapping between library functions and system calls?

open: getting a handle

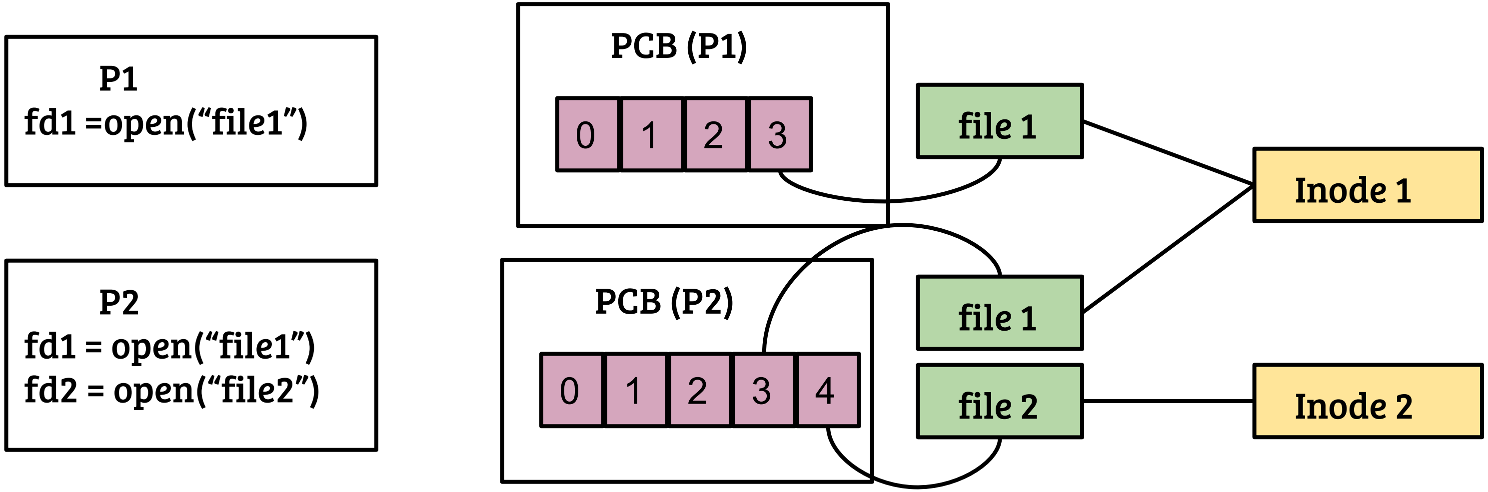
```
int open (char *path, int flags, mode_t mode)
```

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 (integer) is returned
- If flags contain O_CREAT, mode specifies the file creation mode
- Refer man page (“man 2 open")

Process view of file



- Per-process file descriptor table with pointer to a "file" object
- file object ⟶ inode is many-to-one

Process view of file

- What do file descriptors 0, 1 and 2 represent?
- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- Can multiple FDs point to the same file object?

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

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

Process view of file

- What do file descriptors 0, 1 and 2 represent?
- 0 → STDIN, 1 → STDOUT and 2 → STDERR
- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- Can multiple FDs point to the same file object?

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*

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

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

File information (stat, fstat)

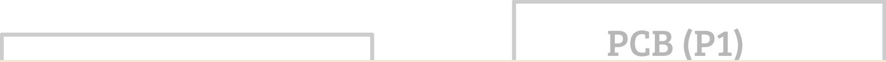
```
int stat(const char *path, struct stat *sbuf);
```

- 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)

Process view of file



- What do file descriptors 0, 1 and 2 represent?
- 0 → STDIN, 1 → STDOUT and 2 → STDERR
- What happens to the FD table and the file objects across fork( )?
    - What happens in exec( )?
- The FD table is copied across fork( ) ⇒ File objects are shared
- On exec, open files remain shared by default
- Can multiple FDs point to the same file object?

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);
```

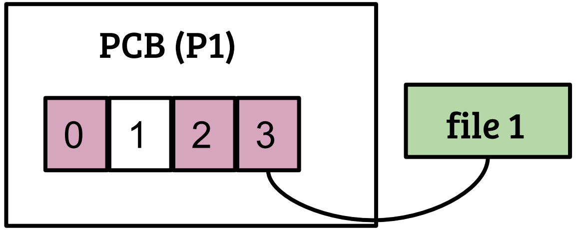
- Close newfd before duping the file descriptor oldfd
- dup2 (fd, 1) equivalent to
  - close(1);
  - dup(fd);

Duplicate file handles (dup and dup2)

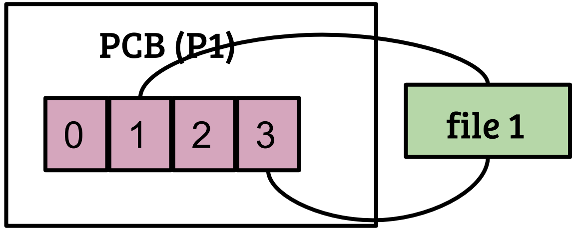
P1
fd1 =open(“file1”)

\+ 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

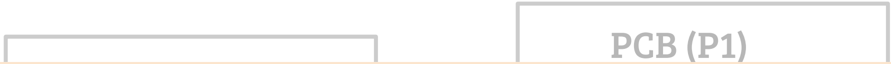
- Example: ls > tmp.txt
- How implemented?

Use of dup: shell redirection

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

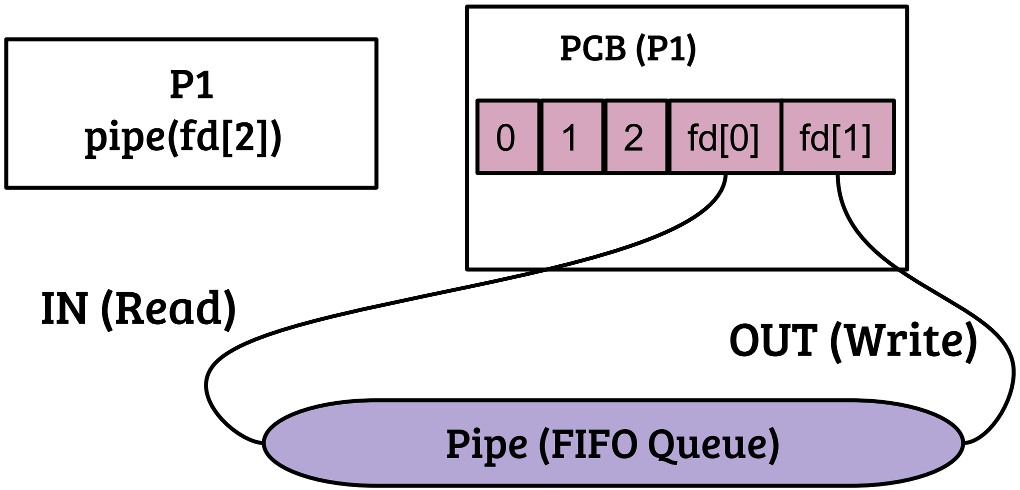
```
fd = open ("tmp.txt")

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

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

exec(ls )
```

Process view of file



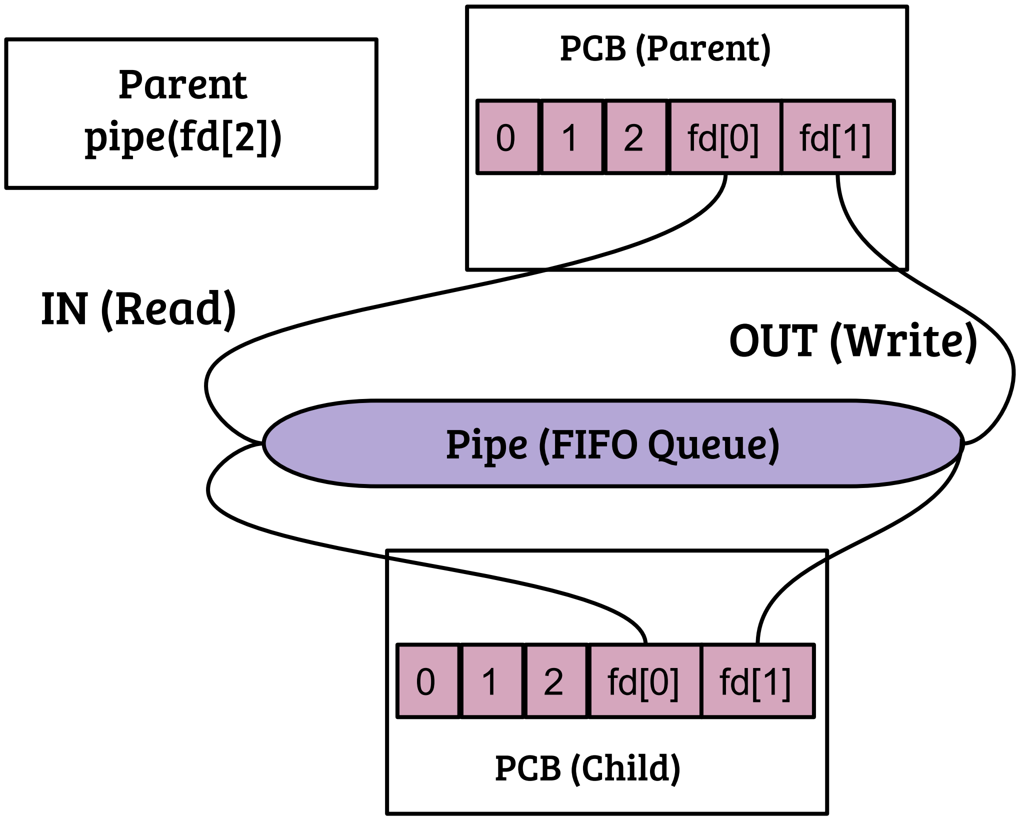
- What do file descriptors 0, 1 and 2 represent?
- 0 ⟶ STDIN, 1 ⟶ STDOUT and 2 ⟶ STDERR
- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- The FD table is copied across fork( ) ⇒ File objects are shared
- On exec, open files remain shared by default
- Can multiple FDs point to the same file object?
- Yes, duped FDs share the same file object (within a process)

UNIX pipe( ) system call



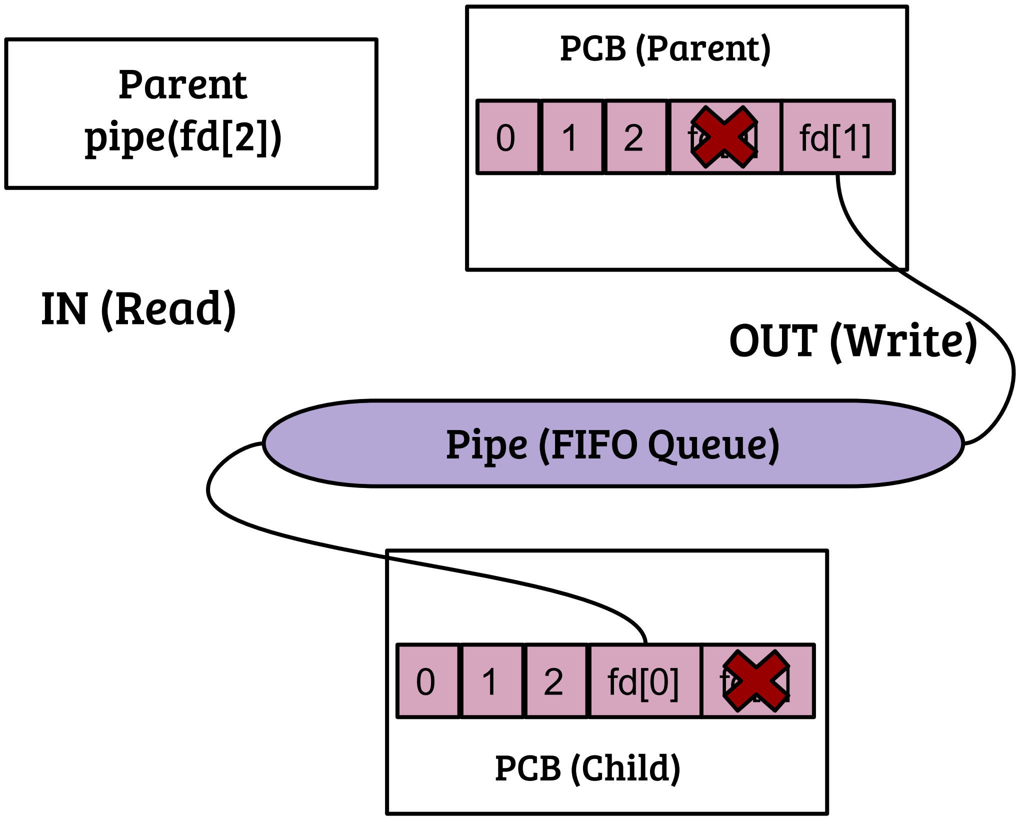
- pipe( ) takes array of two FDs as input
- *fd[0]* is the read end of the pipe
- *fd[1]* is the write end of the pipe
- Implemented as a FIFO queue in OS

UNIX pipe( ) with fork( )



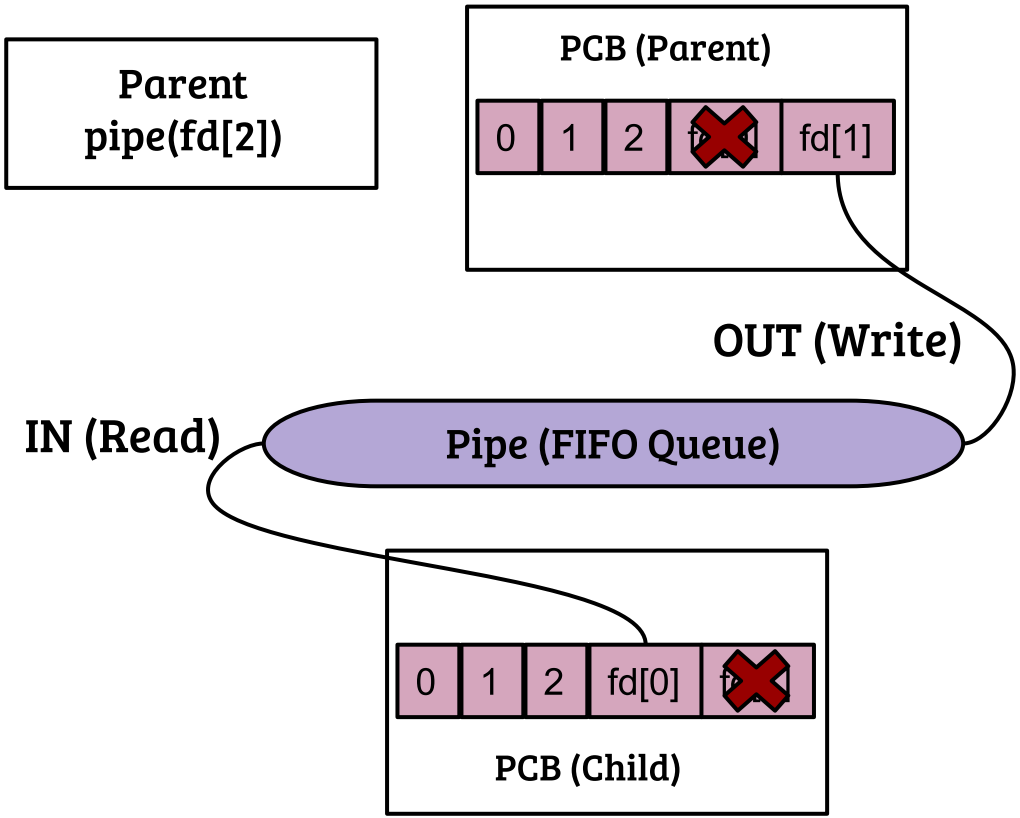
- fork( ) duplicates the file descriptors
- At this point, both the parent and the child processes can read/write to the pipe

UNIX pipe( ) with fork( )



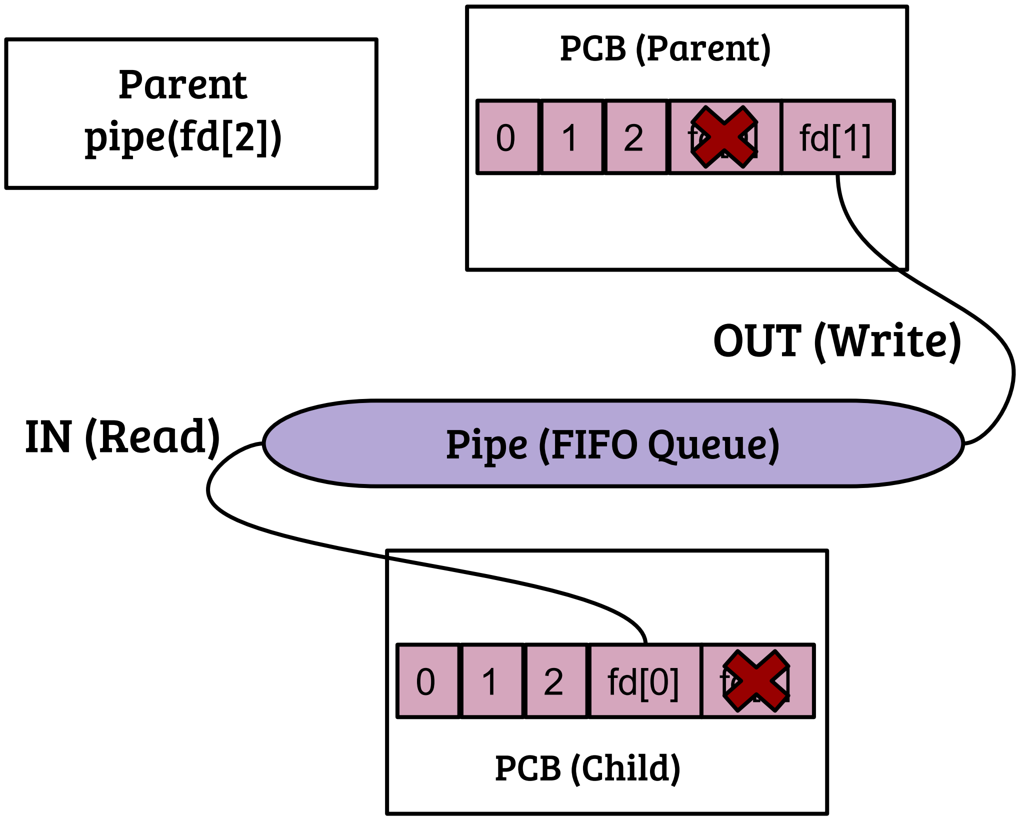
- fork( ) duplicates the file descriptors
- close( ) one end of the pipe, both in child and parent
- Result
  - A queue between parent and child

Shell piping : ls | wc -l



- pipe( ) followed by fork( )
- Parent: exec(“ls”) after making STDOUT $\longrightarrow$ out fd of the pipe (using dup)

Shell piping : ls | wc -l



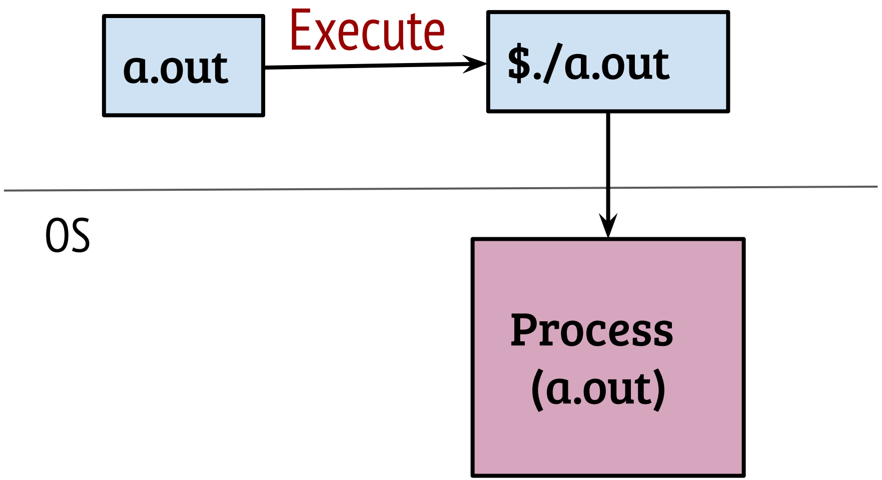
- pipe( ) followed by fork( )
- Parent: exec(“ls”) after making STDOUT ⟶ out fd of the pipe (using dup)
- Child: exec(“wc”) after closing STDIN and duping in fd of pipe
- Result: input of “wc” is connected to output of “ls”

CS330: Operating Systems

Virtual memory: Address spaces

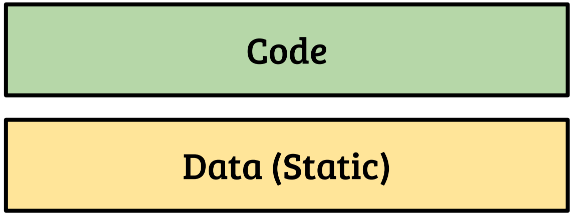
Recap: The process abstraction

- The OS creates a *process* when we run an *executable*



- Executable is a file, stored in a persistent storage (e.g., disk)
- To run, the process code and data should reside in memory
- Run-time memory allocation and deallocation should be supported

Executable file to process memory view



- A typical executable file contains code and statically allocated data
- Statically allocated: global and static variables
- Is loading the program (code and data) sufficient for program execution?

Executable file to process memory view

- A typical executable file contains code and statically allocated data
- Statically allocated: global and static variables
- Is loading the program (code and data) sufficient for program execution?
- No, we need memory for stack and dynamic allocation

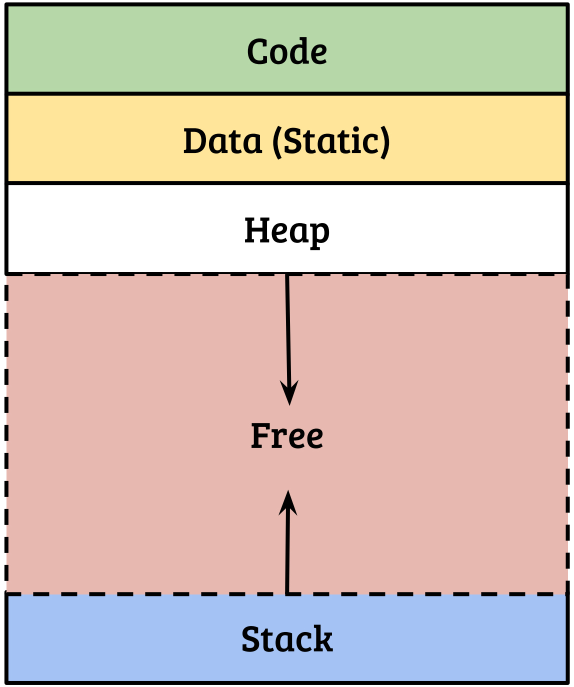
Executable file to process memory view

- A typical executable file contains code and statically allocated data
- Statically allocated: global and static variables
- Is loading the program (code and data) sufficient for program execution?
- No, we need memory for stack and dynamic allocation
- Stack: function call and return, store local (stack) variables

Executable file to process memory view

- A typical executable file contains code and statically allocated data
- Statically allocated: global and static variables
- Is loading the program (code and data) sufficient for program execution?
- No, we need memory for stack and dynamic allocation
- Stack: function call and return, store local (stack) variables
- Heap: dynamic memory allocation through APIs like *malloc( )*

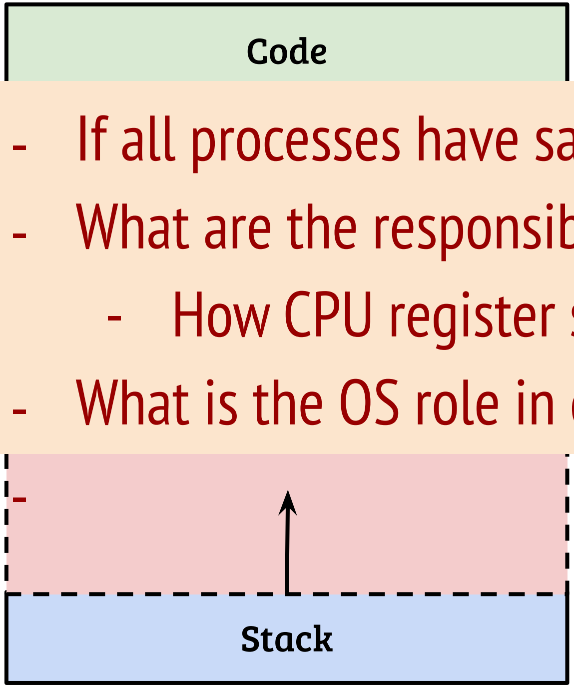
The address space abstraction



- Address space represents memory state of a process
- Address space layout is same for all the processes (convenience)
- Exact layout can be decided by the OS, conventional layout is shown

The address space abstraction

- If all processes have same address space, how they map to actual memory?
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- What is the OS role in dynamic memory allocation?



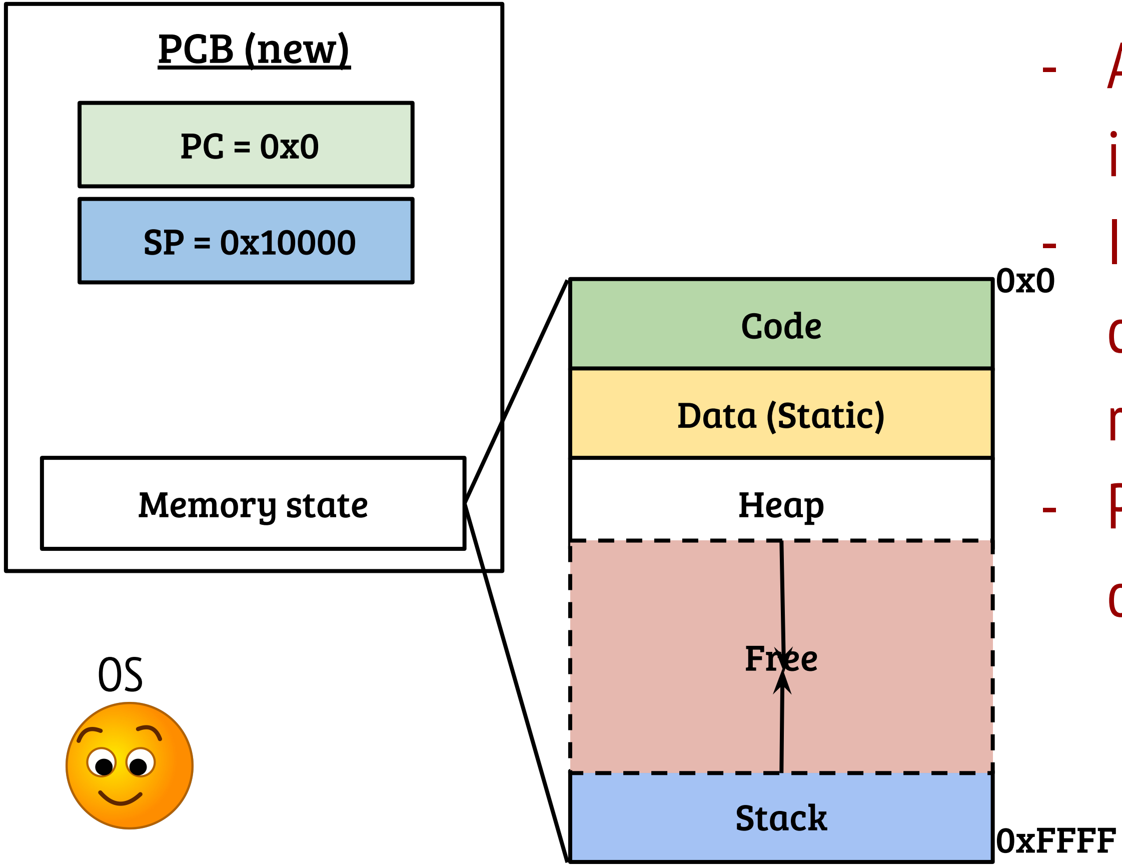
conventional layout is shown

The address space abstraction

Code

- If all processes have same address space, how they map to actual memory?
- Architecture support used by OS techniques to perform memory virtualization i.e., translate virtual address to physical address (will revisit)
- What are the responsibilities of the OS during program load?
    - How CPU register state is changed?
- What is the OS role in dynamic memory allocation?

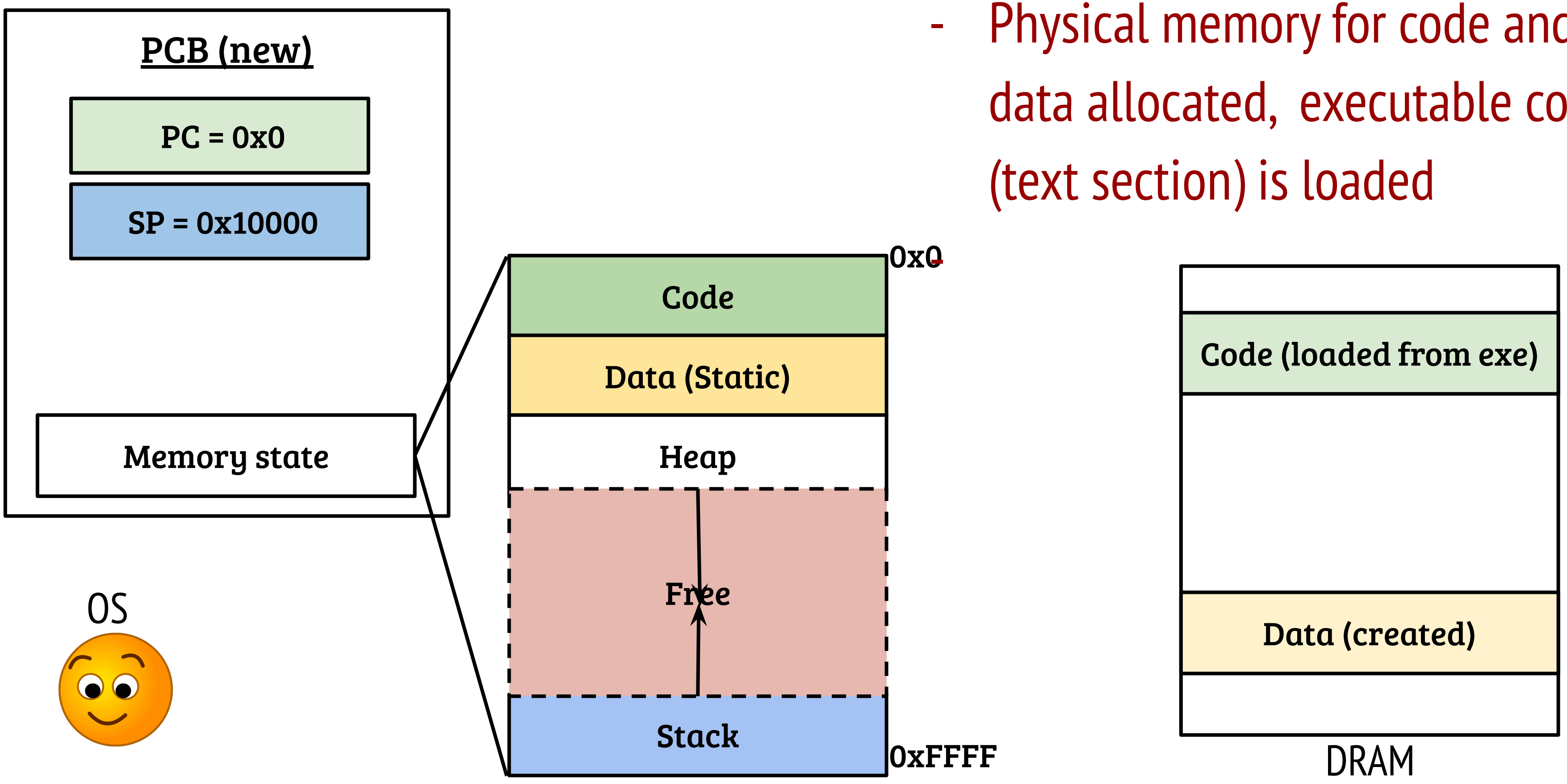
OS during program load (exec)



- A fresh address space is initialized
- In reality, parent address space copied at the time of fork( ) is reset
- PC and SP are set with addresses of code and stack, respectively

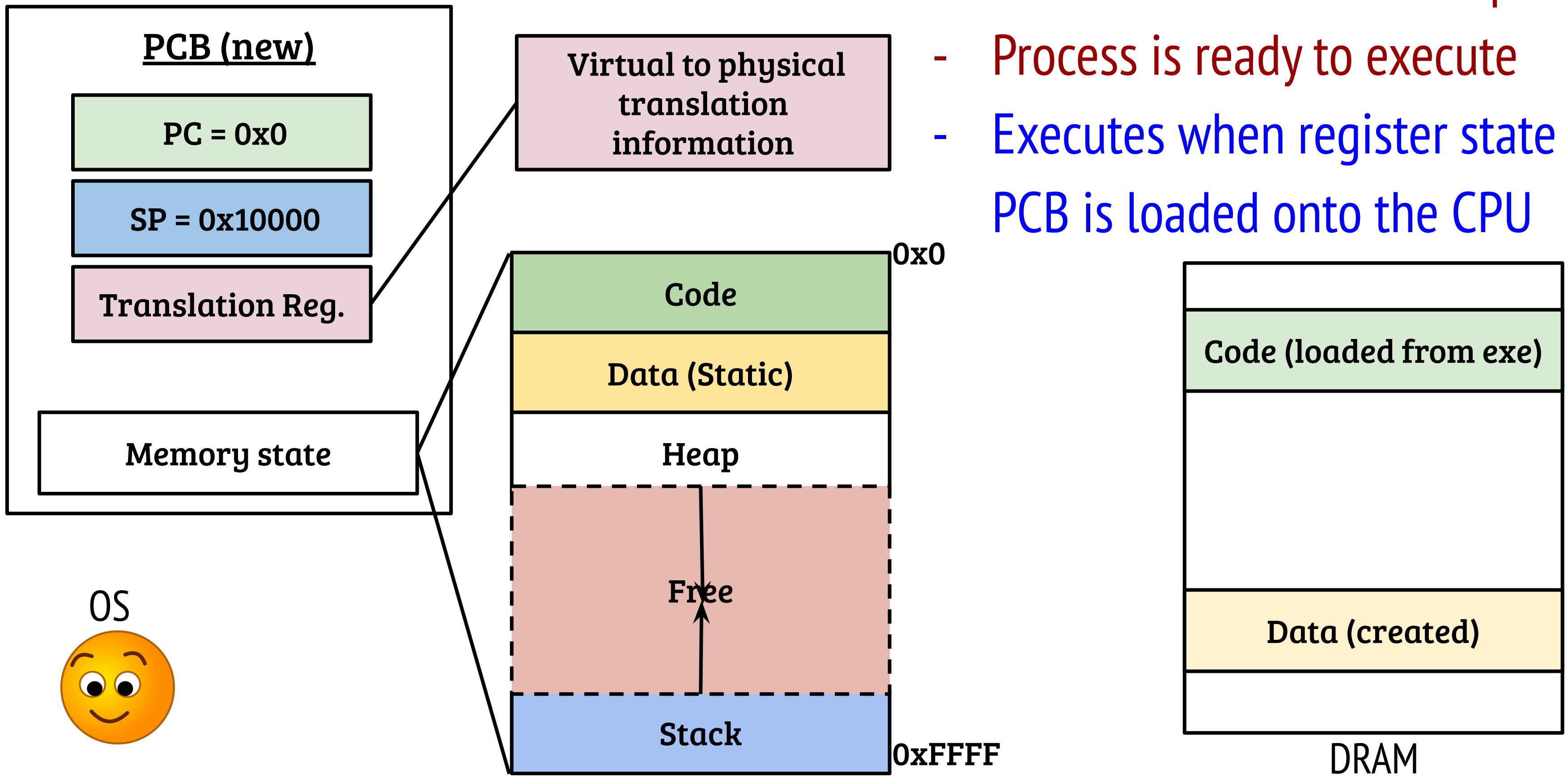
OS during program load (exec)

- Physical memory for code and data allocated, executable code (text section) is loaded



OS during program load (exec)

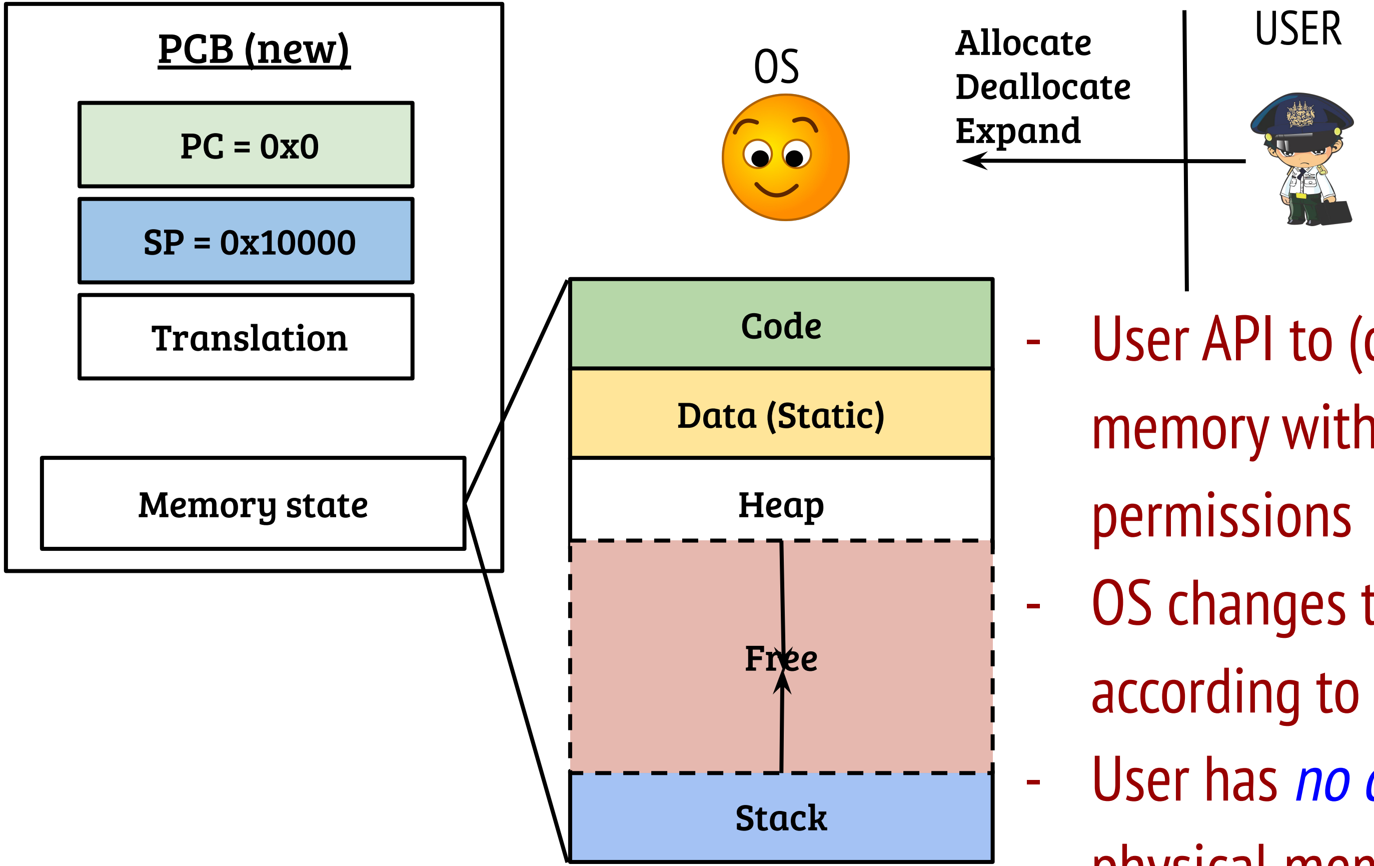
- Translation information updated
- Process is ready to execute
- Executes when register state in PCB is loaded onto the CPU



The address space abstraction

- If all processes have same address space, how they map to actual memory?
- Architecture support used by OS techniques to perform memory virtualization i.e., translate virtual address to physical address (will revisit)
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- Creating address space, loading binary, updating the PCB register state
- What is the OS role in dynamic memory allocation?

User API for memory management

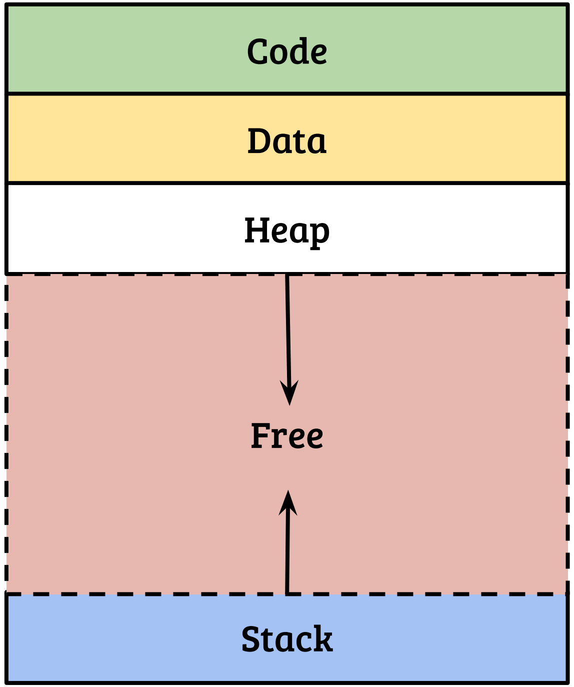


- User API to (de)allocate heap memory with different access permissions
- OS changes the *memory state* according to the user request
- User has *no direct control* on physical memory

CS330: Operating Systems

Virtual memory: Memory API

Recap: Process address space

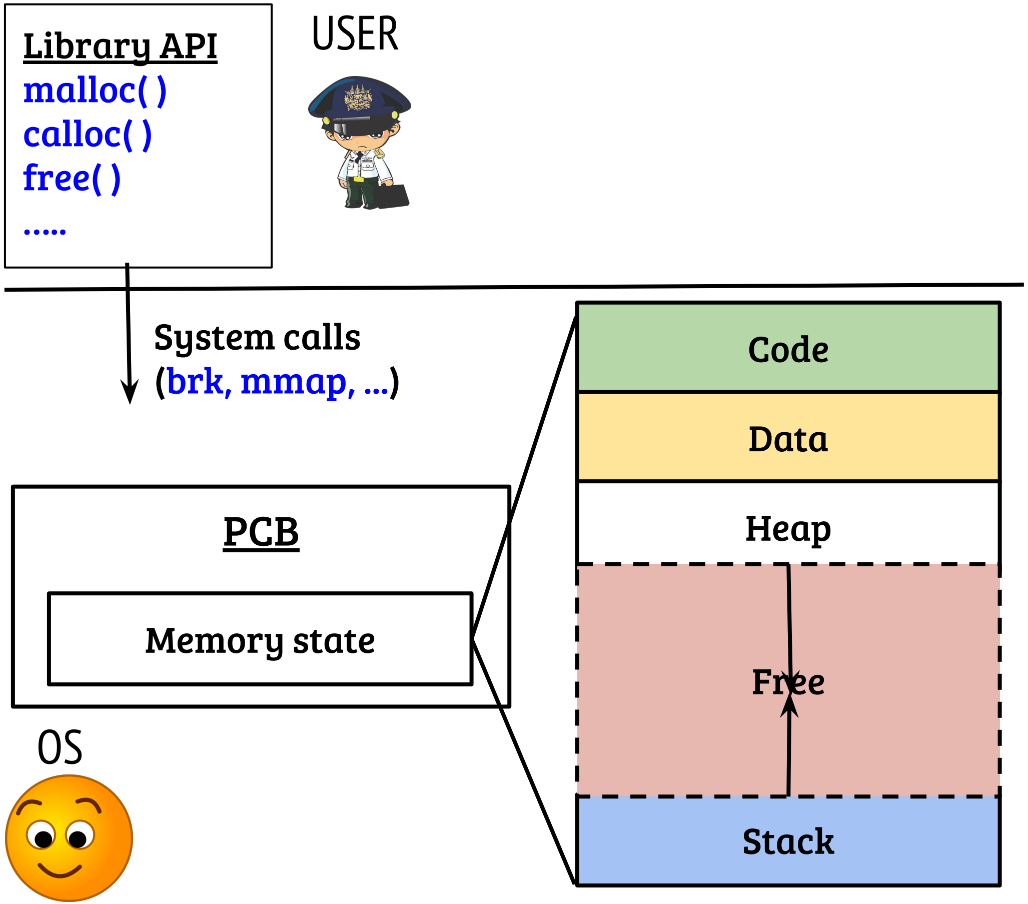


- Address space presents the same view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view

Recap: Process address space

- If all processes have same address space, how they map to actual memory?
- Architecture support used by OS techniques to perform memory virtualization i.e., translate virtual address to physical address (will revisit)
- What are the responsibilities of the OS during program load?
  - How CPU register state is changed?
- Creating address space, loading binary, updating the PCB register state
- What is the role of OS in dynamic memory allocation?

User API for memory management



- Generally, user programs use library routines to allocate/deallocate memory
- OS provides some address space manipulation system calls (week's agenda)

User API for memory management

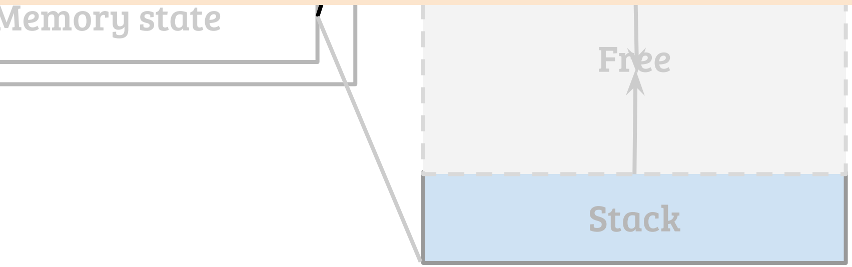
Library API
malloc( )
calloc( )

USER

- Generally, user programs

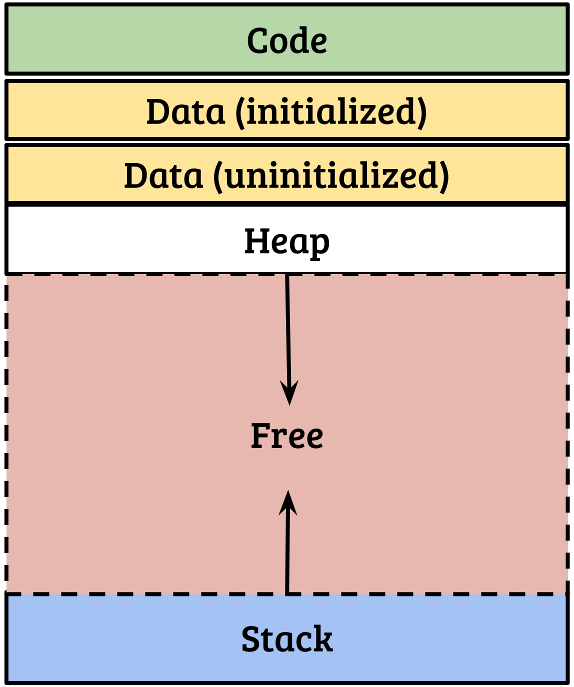
- Can the size of segments change at runtime? If yes, which ones and how?
- How can we know about the segment layout at program load and runtime?
- How to allocate memory chunks with different permissions?
- What is the structure of PCB memory state?

calls (today's agenda)



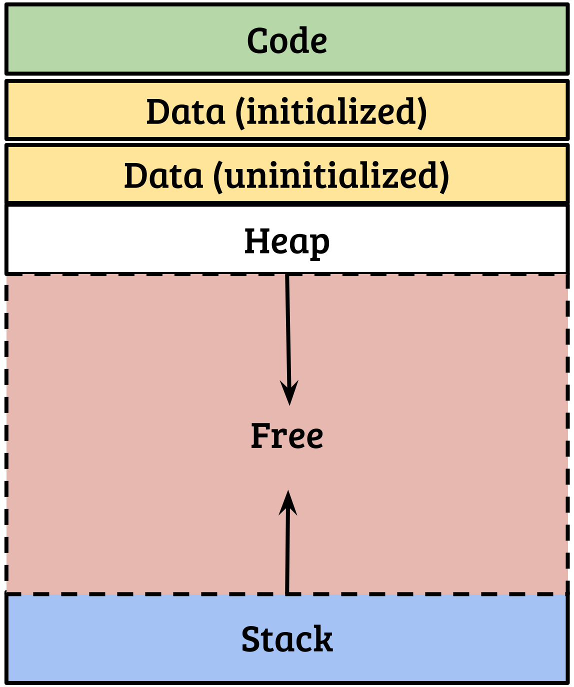
OS

Dynamically sizing the segments (UNIX)



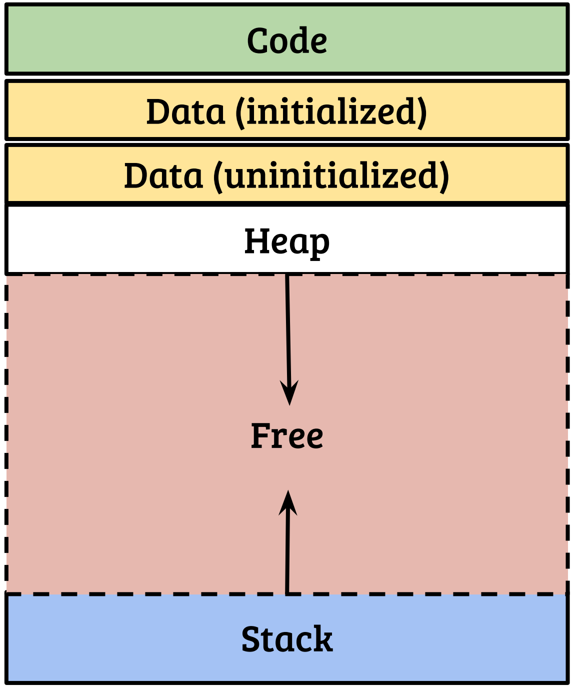
- Code segment size and initialized data segment size is fixed (at exe load)

Dynamically sizing the segments (UNIX)



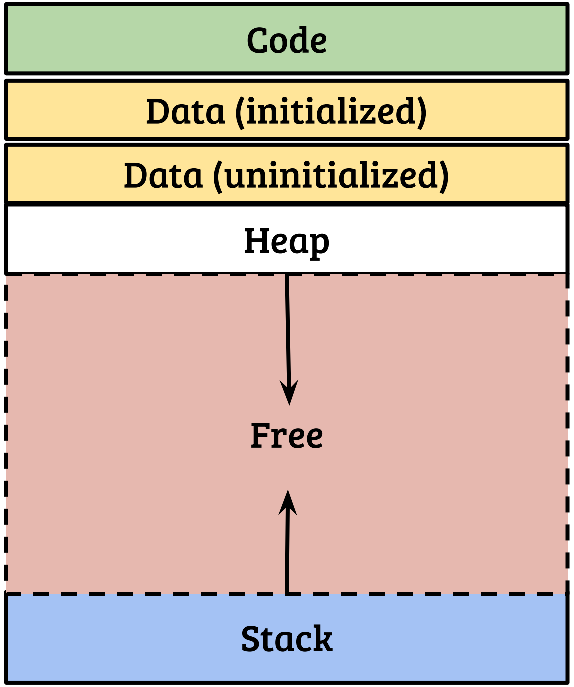
- Code segment size and initialized data segment size is fixed (at exe load)
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically

Dynamically sizing the segments (UNIX)



- Code segment size and initialized data segment size is fixed (at exe load)
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically
- Heap allocation can be discontinuous, special system calls like mmap( ) provide the facility

Dynamically sizing the segments (UNIX)



- Code segment size and initialized data segment size is fixed (at exe load)
- End of uninitialized data segment (a.k.a. BSS) can be adjusted dynamically
- Heap allocation can be discontinuous, special system calls like mmap( ) provide the facility
- Stack grows automatically based on the run-time requirements, no explicit system calls

Sliding the BSS (brk, sbrk)

int brk(void *address);

- If possible, set the end of uninitialized data segment at *address*
- Can be used by C library to allocate/free memory dynamically

void * sbrk (long size);

- Increments the program's data space by size bytes and returns the old value of the end of bss
- sbrk(0) returns the current location of BSS

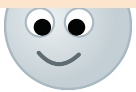
Finding the segments

- etext, edata and end variables mark the end of text segment, initialized data segment and the BSS, respectively (At program load)
- sbrk(0) can be used to find the end of the data segment
- Printing the address of functions and variables
- Linux provides the information in /proc/pid/maps

User API for memory management

Library API USER

- Can the size of segments change at runtime? If yes, which ones and how?
- Heap and data segments can be adjusted using brk and sbrk
- How can we know about the segment layout at program load and runtime?
- Using predefined variables, sbrk, proc file system (Linux)
- How to allocate memory chunks with different permissions?
- What is the structure of PCB memory state?



Stack

Discontiguous allocation (mmap)

- mmap( ) is a powerful and multipurpose system call to perform dynamic and discontiguous allocation (explicit OS support)
- Allows to allocate address space
  - with different protections (READ/WRITE/EXECUTE)
  - at a particular address provided by the user
- Example: Allocate 4096 bytes with READ+WRITE permission

```
ptr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS
|MAP_PRIVATE, -1, 0);  // See the man page for details
```

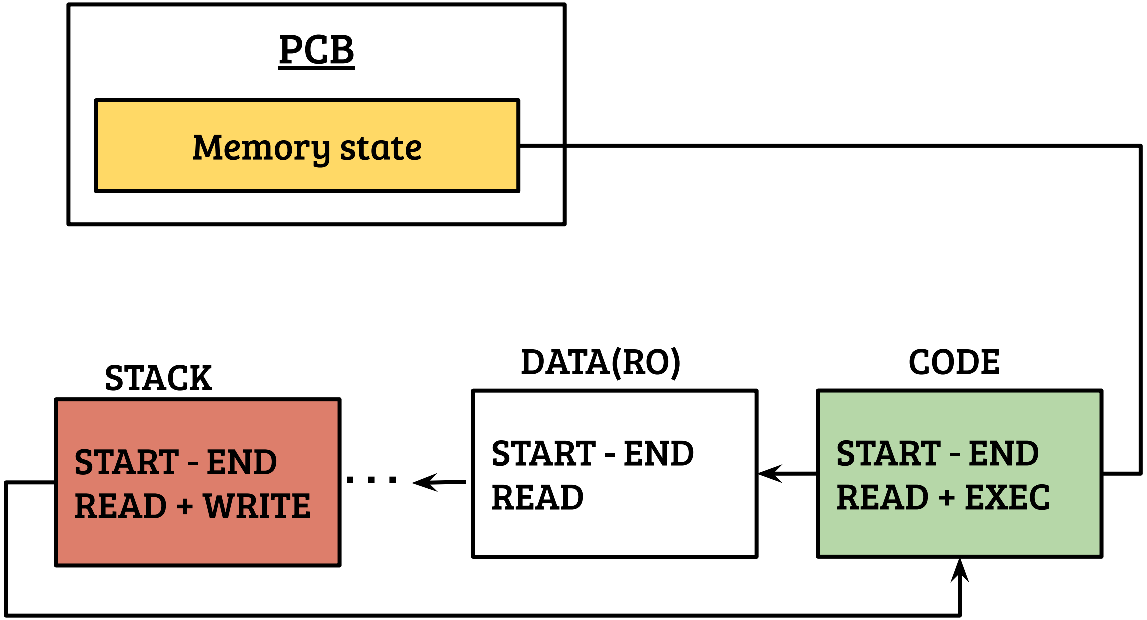
User API for memory management

Library API

USER

- Can the size of segments change at runtime? If yes, which ones and how?
- Heap and data segments can be adjusted using brk and sbrk
- How can we know about the segment layout at program load and runtime?
- Using predefined variables, sbrk, proc file system (Linux)
- How to allocate memory chunks with different permissions?
- mmap( ) supports discontinuous allocation with different permissions
- What is the structure of PCB memory state?

Memory state of PCB (example)



- Maintained as a sorted circular list accessible from PCB
- START and END never overlap between two segment areas
- Can merge/extend areas if permissions match

User API for memory management

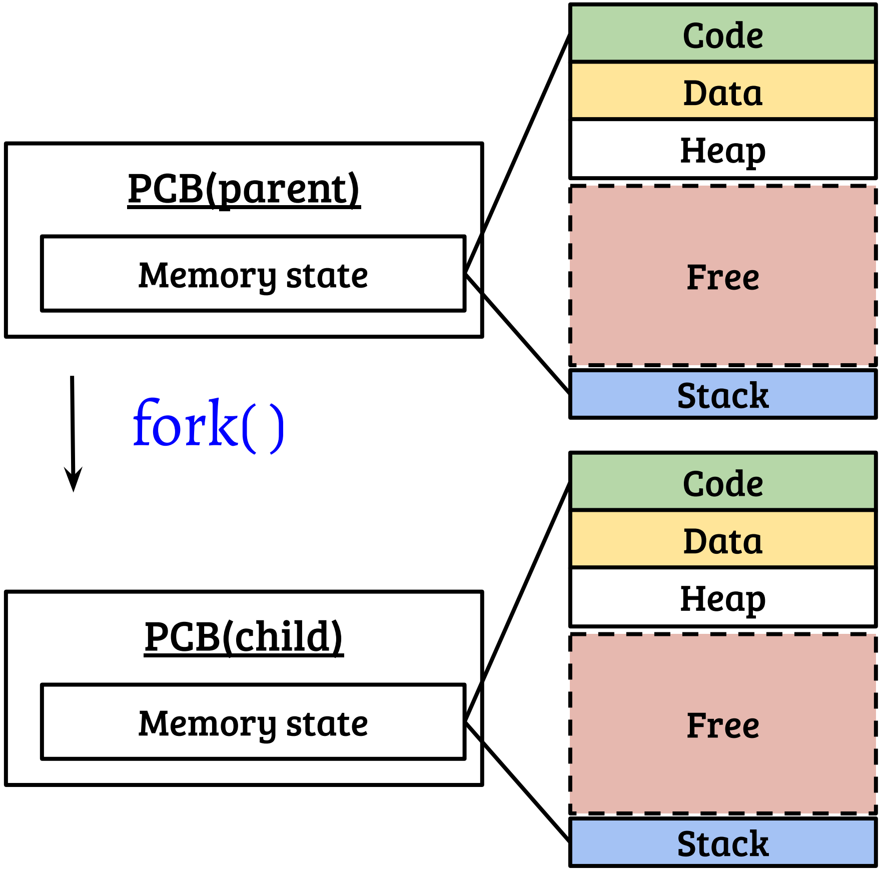
Library API

USER

- Can the size of segments change at runtime? If yes, which ones and how?
- Heap and data segments can be adjusted using brk and sbrk
- How can we know about the segment layout at program load and runtime?
- Using predefined variables, sbrk, proc file system (Linux)
- How to allocate memory chunks with different permissions?
- mmap( ) supports discontinuous allocation with different permissions
- What is the structure of PCB memory state?
- A sorted data structure of allocated areas can be used

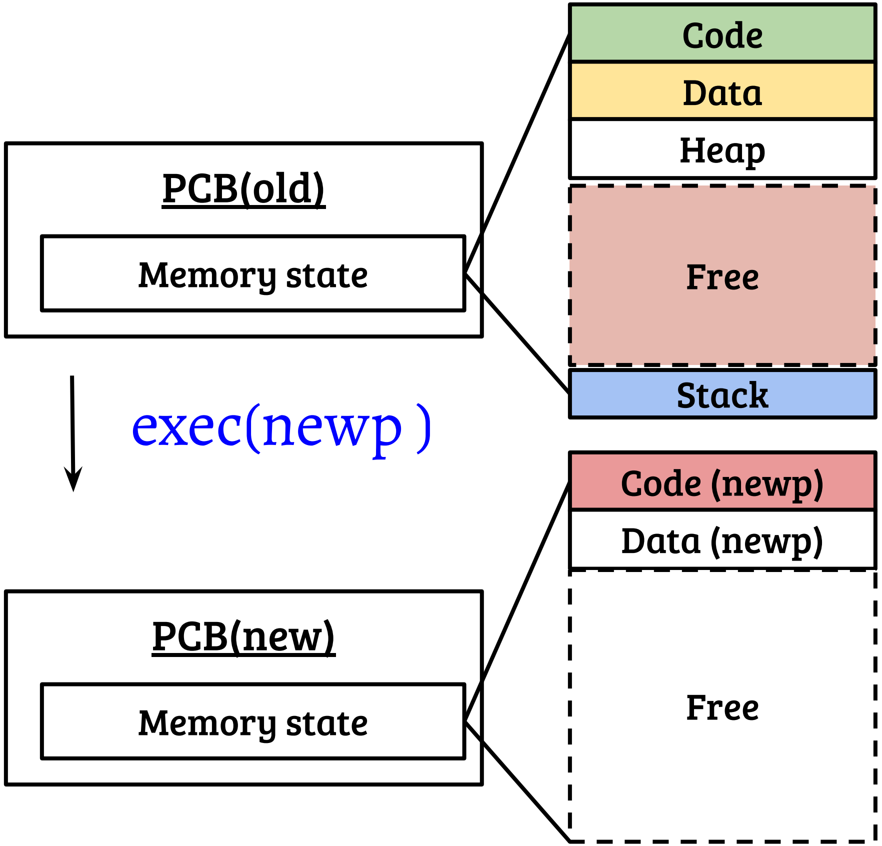
Stack

Inheriting address space through fork( )



- Child inherits the memory state of the parent
  - The memory state data structures are copied into the child PCB
- Any change through mmap( ) or brk( ) is per-process

Overriding address space through exec( )



- The address space is reinitialized using the new executable
- Changes to newly created address space depends on the logic of new process

CS330: Operating Systems

Limited direct execution

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?

Recap: virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?
  - The OS performs multiplexing of physical resources efficiently
  - Maintains mapping of virtual view to physical resource

Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, no OS mediation
  - Example: when a process is scheduled on a CPU, it should execute without OS intervention
- What is the catch?

Virtualization: Efficiency/performance

- Resource virtualization should not add any overheads
- Efficient when programs use the resource directly, infrequent OS mediation
    - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
    - Loss of control e.g., process running an infinite loop on a CPU
    - Isolation issues e.g., process accessing/changing OS data structures

Virtualization: Efficiency/performance

- Resource virtualization should not add any overheads
- Efficient when programs use the resource directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

Conclusion: Some limits to direct access must be enforced.

Limited direct execution

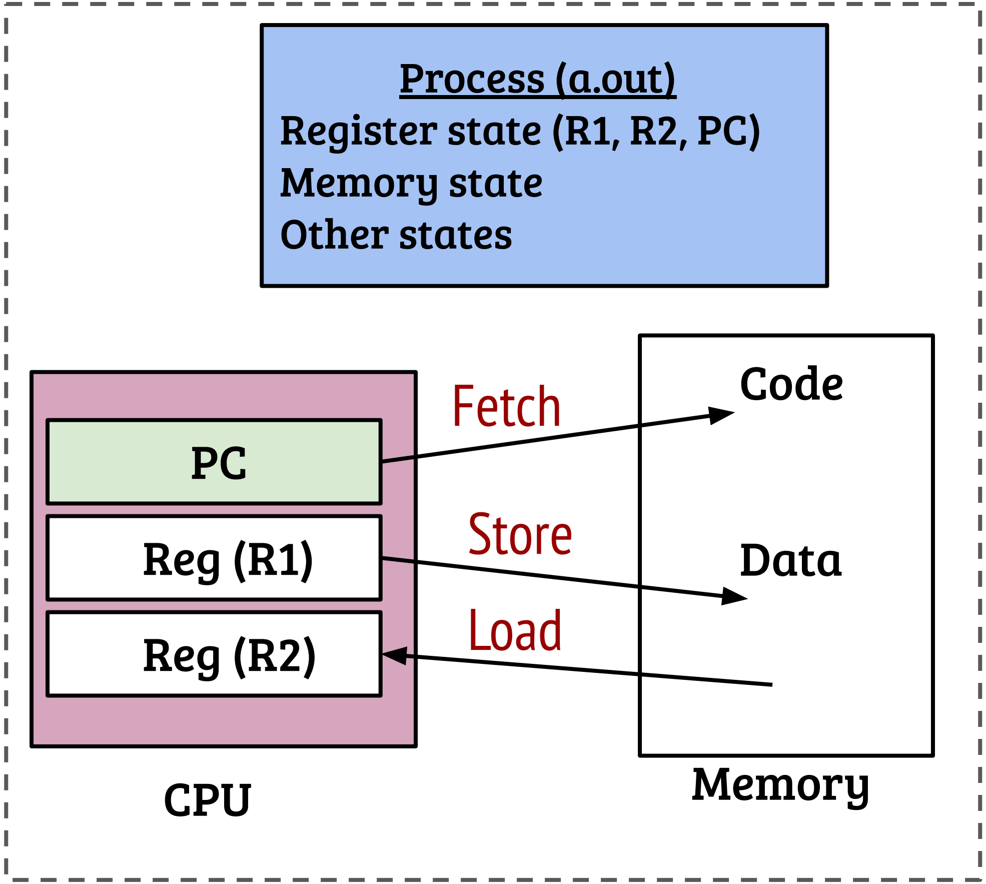
- Can the OS enforce limits to an executing process by itself?

A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

OS

I want to restrict this process accessing memory of other processes, but how? Monitoring each memory access is not efficient!



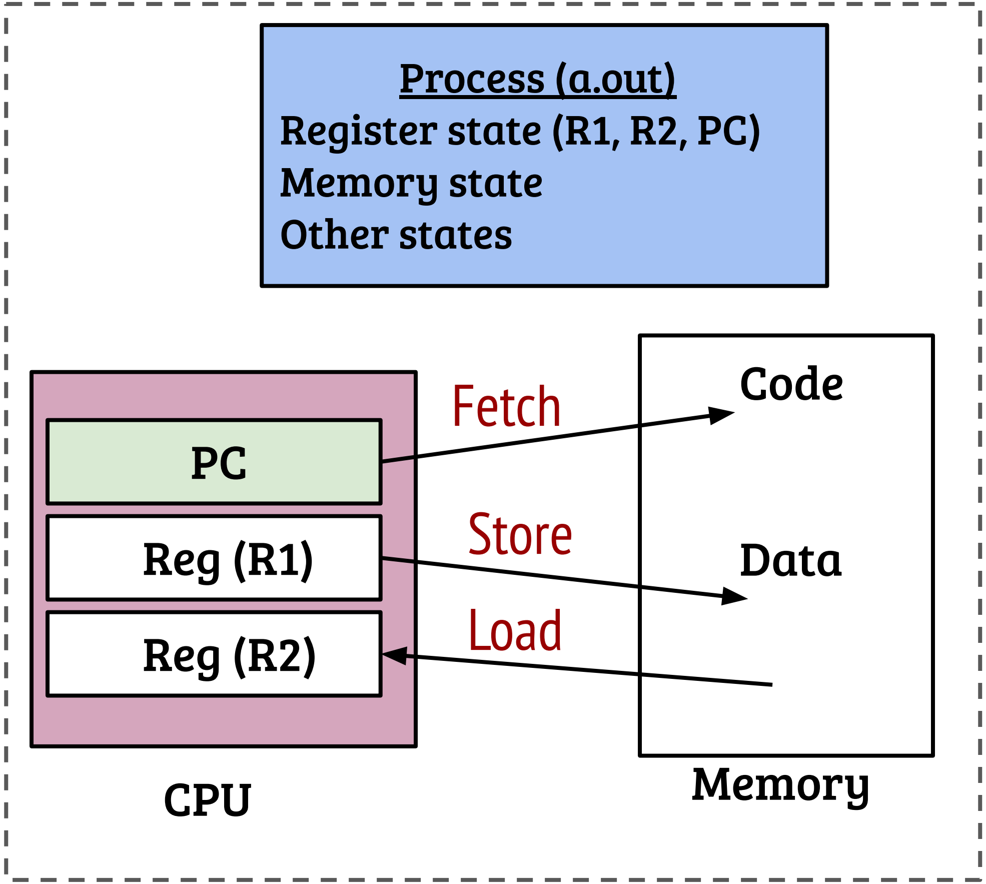
A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

Help me!

OS

I want to restrict this process accessing memory of other processes, but how? Monitoring each memory access is not efficient!



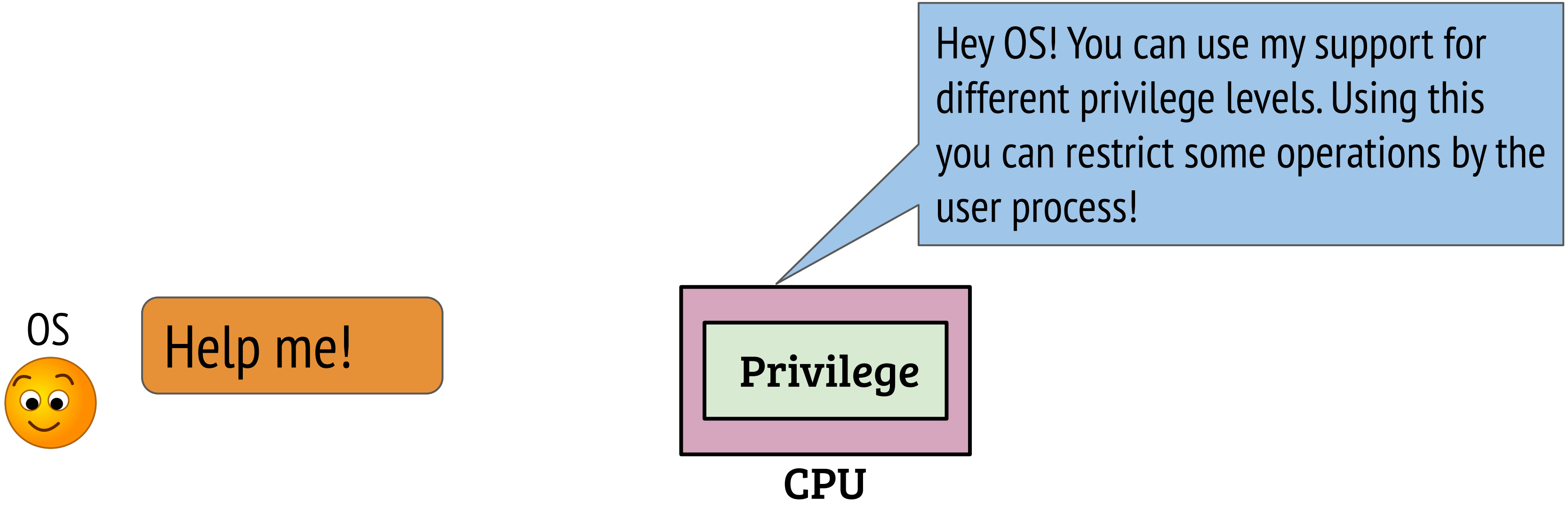
Limited direct execution

- Can the OS enforce limits to an executing process by itself?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!

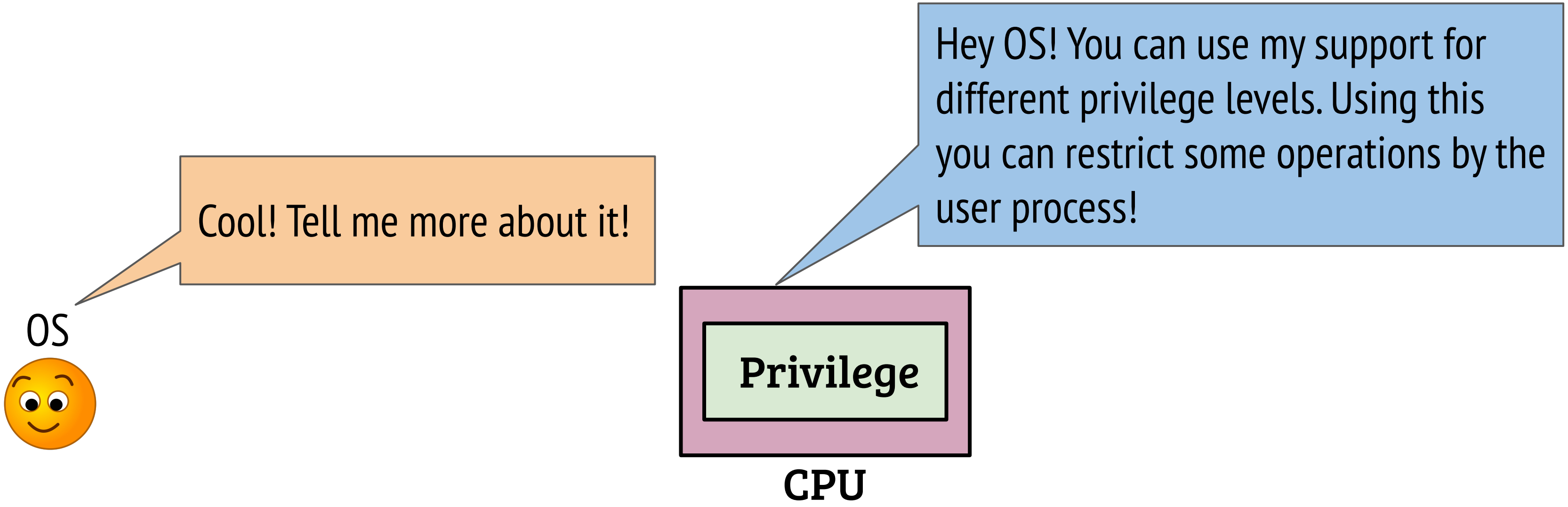
Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?

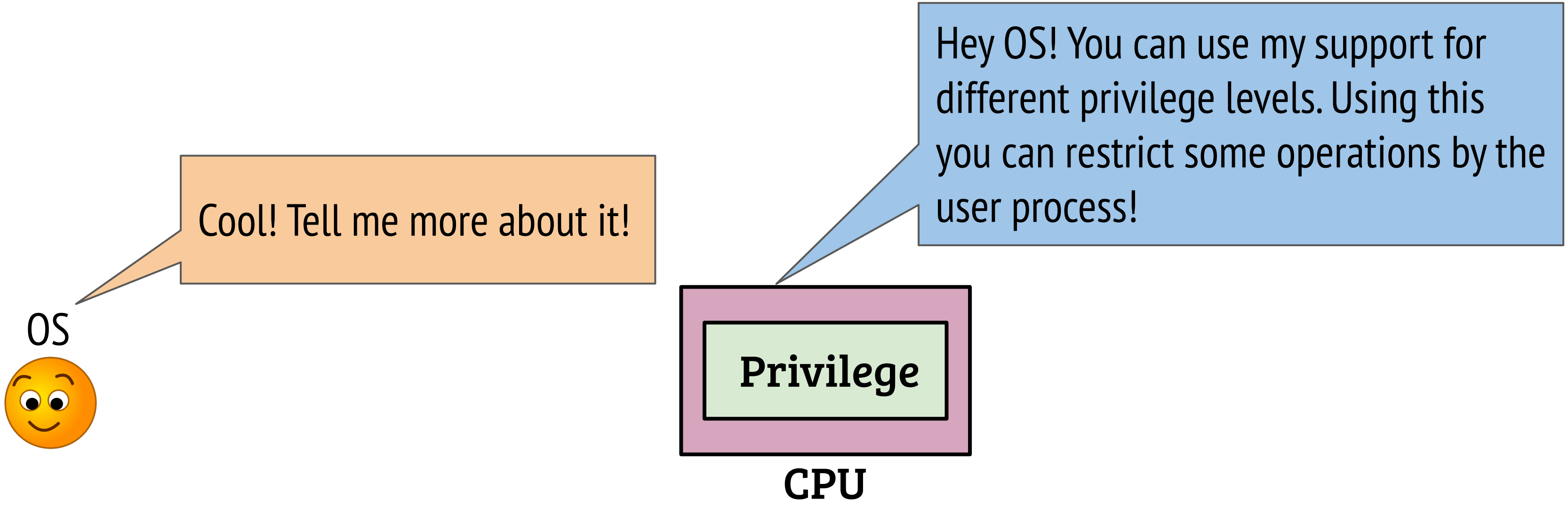
Hardware support: Privilege levels



Hardware support: Privilege levels

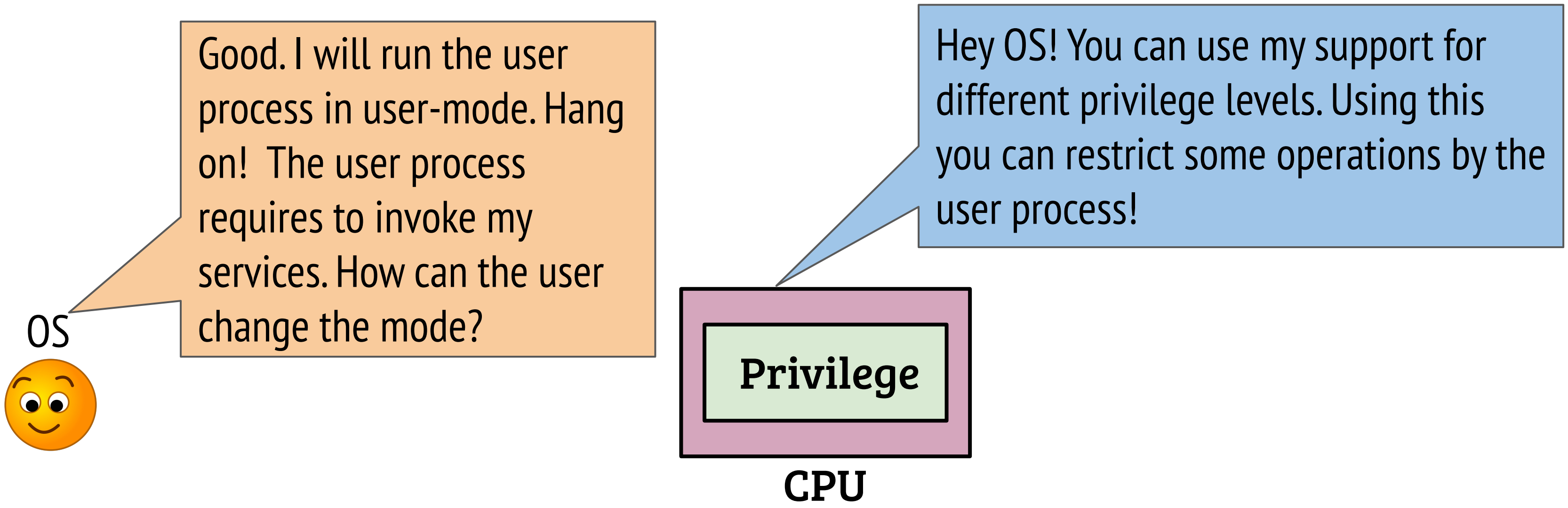


Hardware support: Privilege levels

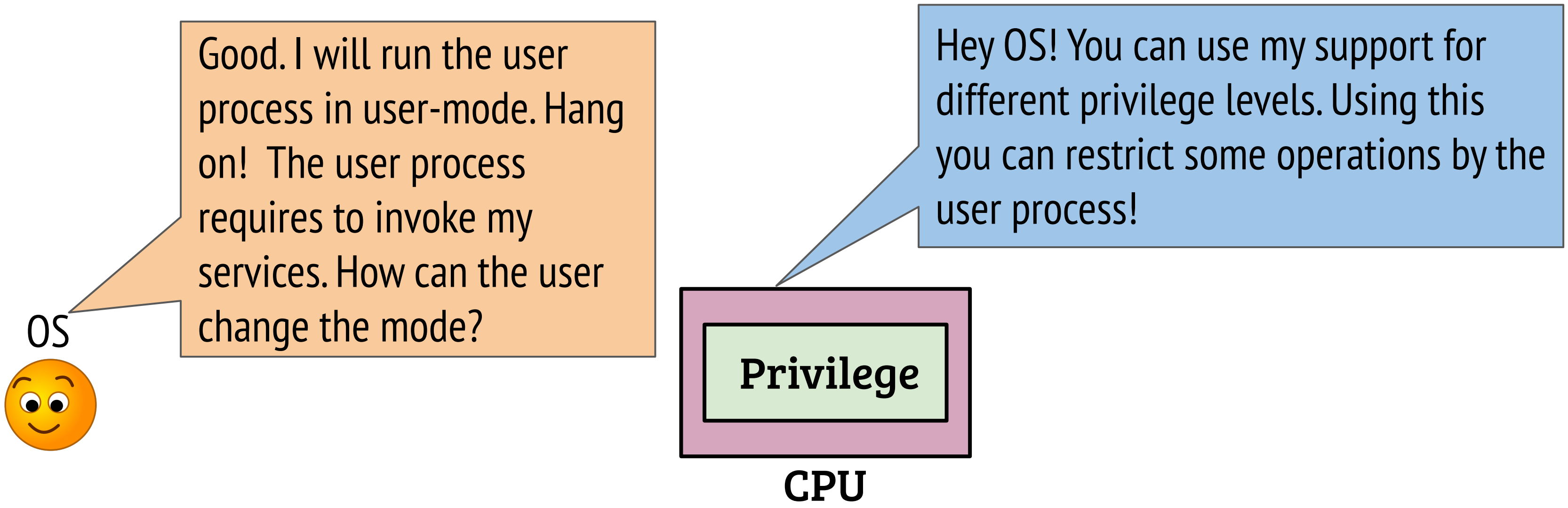


- CPU can execute in two modes: *user-mode* and *kernel-mode*
- Some operations are allowed only from kernel-mode (privileged OPs)
  - If executed from user mode, hardware will notify the OS by raising a fault/trap

Hardware support: Privilege levels



Hardware support: Privilege levels



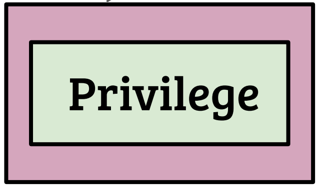
- From user-mode, privilege level of CPU can not be changed directly
- The hardware provides entry instructions from the user-mode which causes a mode switch
- The OS can define the handler for different entry gates

Hardware support: Privilege levels

Okay. You said that if the process does some mischief from the user mode, you will notify me. That means, I can define handlers for faults and exceptions too.

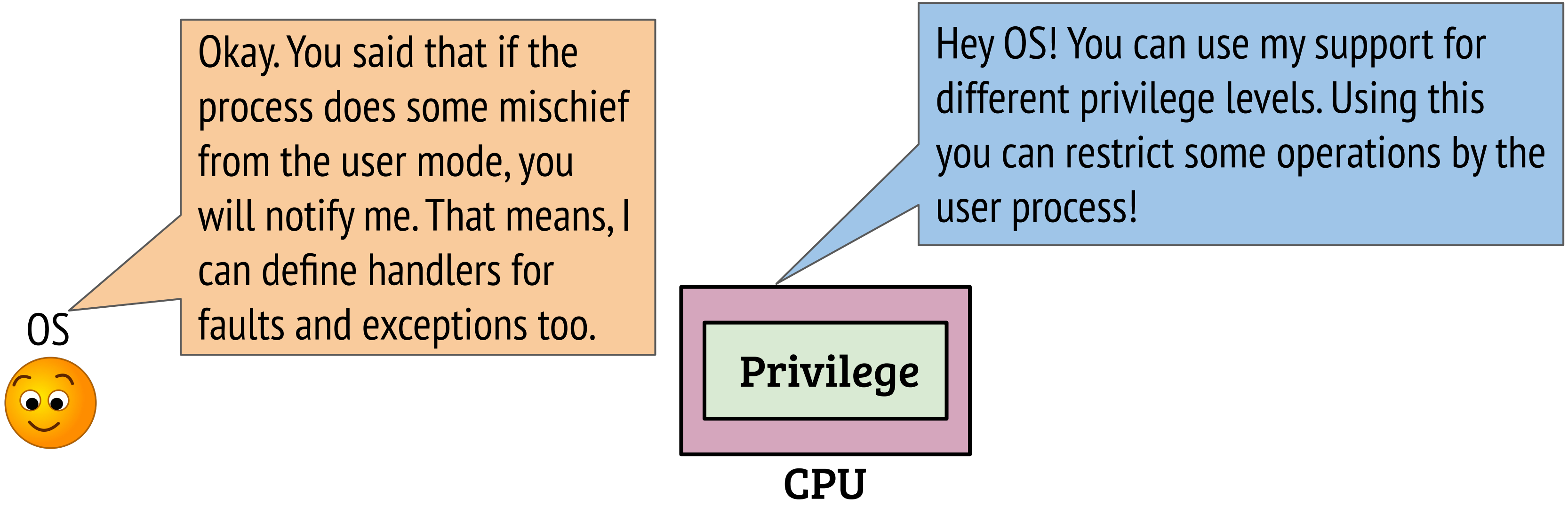
OS

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!



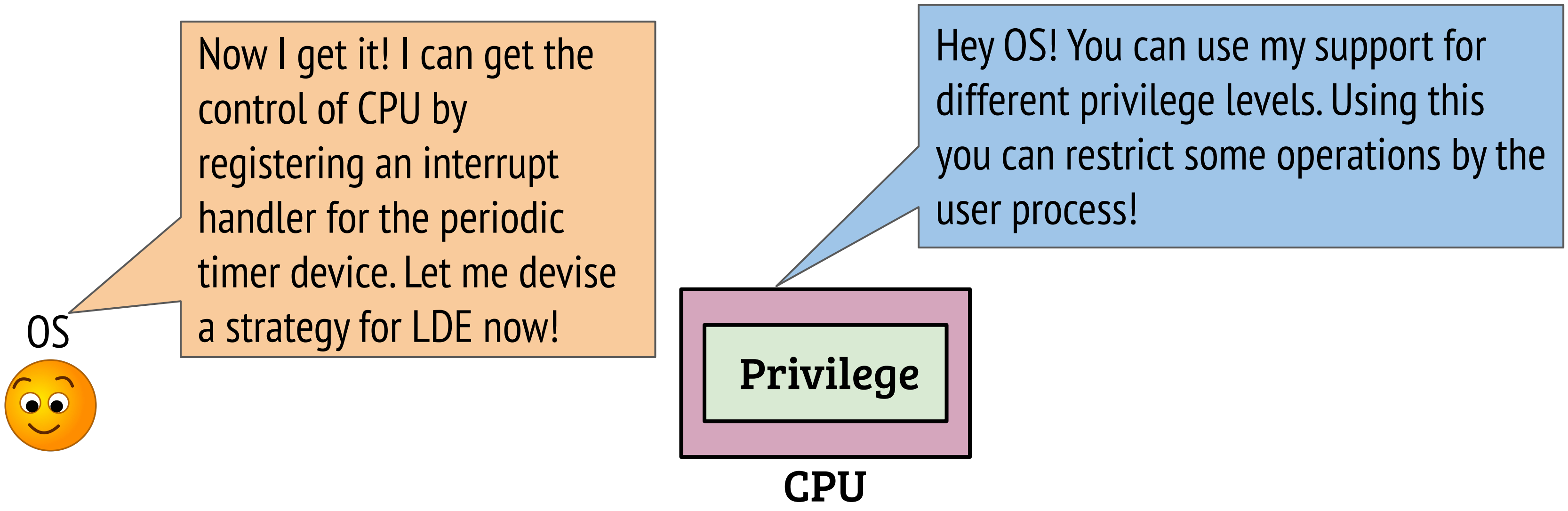
CPU

Hardware support: Privilege levels

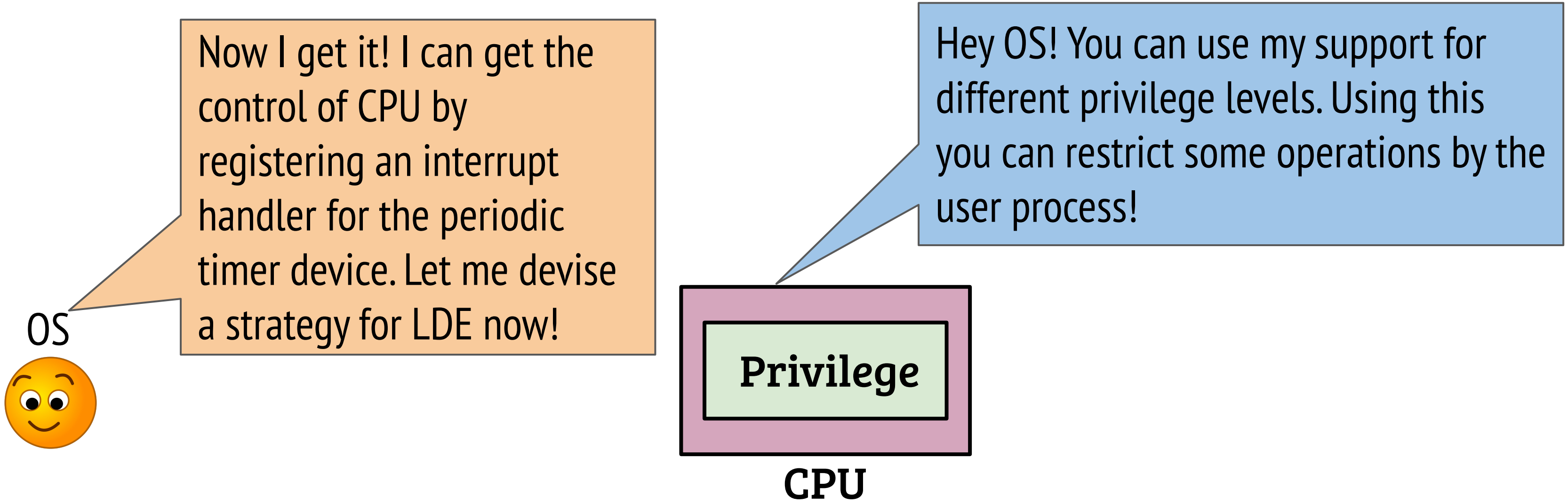


- The OS can register the handlers for faults and exceptions
- The OS can also register handlers for device interrupts
- *Registration of handlers is privileged!*

Hardware support: Privilege levels

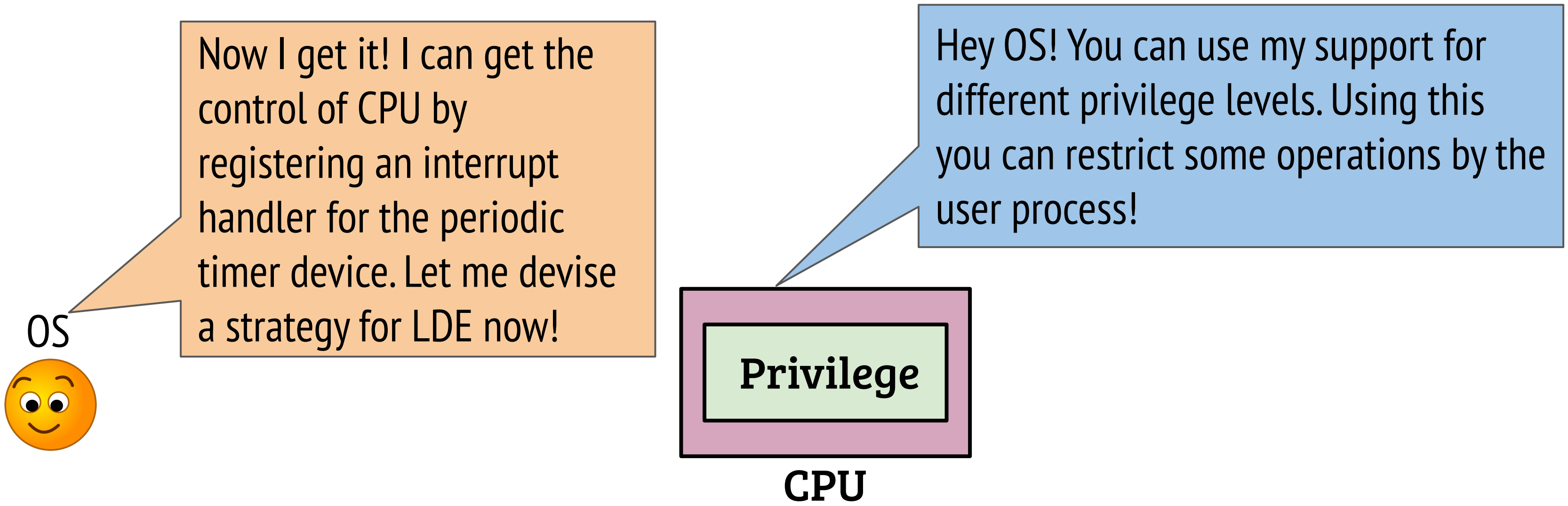


Hardware support: Privilege levels



- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts

Hardware support: Privilege levels



- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts
- The handler code is invoked by the OS when user-mode process invokes a system call or an exception or an external interrupt

Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

CS330: Operating Systems

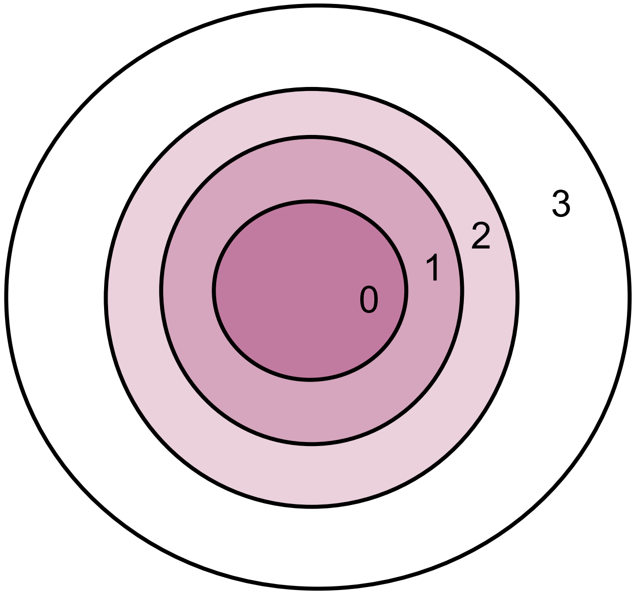
Privileged ISA (X86_64)

Recap: Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

Agenda: High-level view of x86_64 support for privileged mode

X86: rings of protection



- 4 privilege levels: 0⟶ highest, 3⟶ lowest
- Some operations are allowed only in privilege level 0
- Most OSes use 0 (for kernel) and 3 (for user)
- Different kinds of privilege enforcement
    - Instruction is privileged
    - Operand is privileged

Privileged instruction: HLT (on Linux x86_64)

```
int main( )
{
  asm("hlt;");
}
```

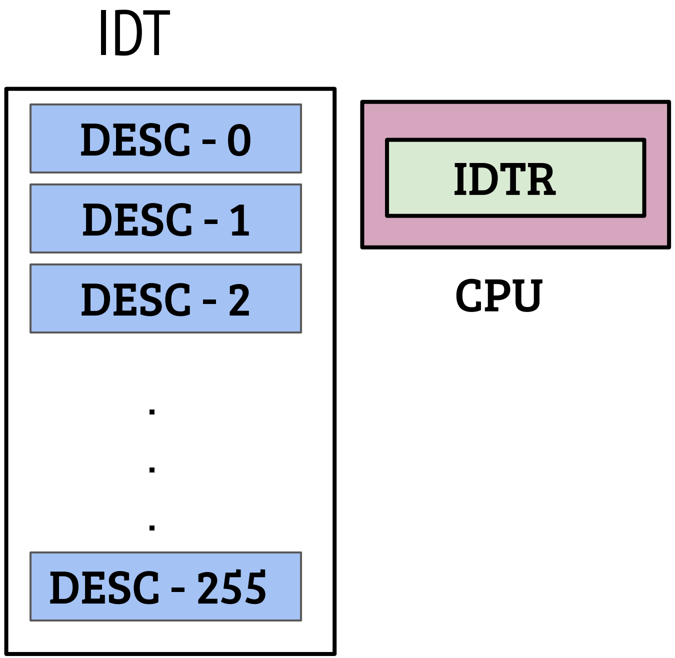
- HLT: Halt the CPU core till next external interrupt
- Executed from user space results in protection fault
- Action: Linux kernel kills the application

Privileged operation: Read CR3 (Linux x86_64)

```
#include<stdio.h>
int main( ){
  unsigned long cr3_val;
  asm volatile("mov %%cr3, %0;"
           : "=r" (cr3_val)
           :: );
 printf("%lx\n", cr3_val);
}
```

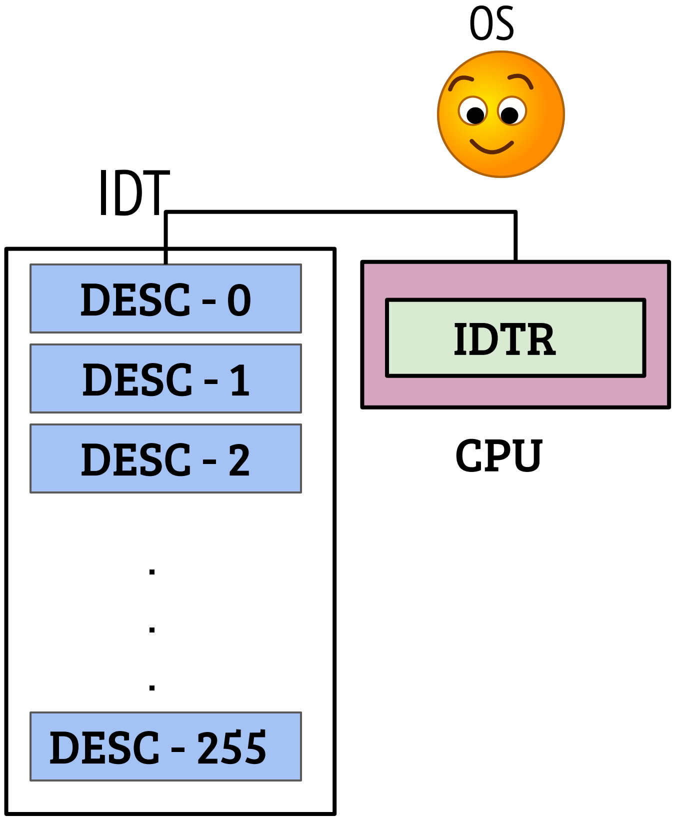
- CR3 register points to the address space translation information
- When executed from user space results in protection fault
- “mov” instruction is not privileged per se, but the operand is privileged

Interrupt Descriptor Table (IDT): gateway to handlers



- Interrupt descriptor table provides a way to define handlers for different events like external interrupts, faults and system calls by defining the descriptors
- Descriptors 0-31 are for predefined events e.g., 0 → Div-by-zero exception etc.
- Events 32-255 are user defined, can be used for h/w and s/w interrupt handling

Defining the descriptors (OS boot)



- Each descriptor contains information about handling the event
  - Privilege switch information
  - Handler address
- The OS defines the descriptors and loads the IDTR register with the address of the descriptor table (using *LIDT* instruction)

System call INT instruction (gemOS)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function. How?

System call INT instruction (gemOS)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function, How?
    - Every system call is associated with a number (defined by OS)
    - User process sends information like system call number, arguments through CPU registers which is used to invoke the actual handler

System call in gemOS

- gemOS defines system call handler for descriptor 0x80
- System call number is passed in RDI register and the return value is stored in RAX register
- Parameters are passed using the registers in the following order
  - RDI (syscall #), RSI (param #1), RDX (param #2), RCX(param #3), R8 (param #4), R9 (param #5)
- Let us write a new system call!

CS330: Operating Systems

OS mode execution

Recap: Limited direct execution support in X86

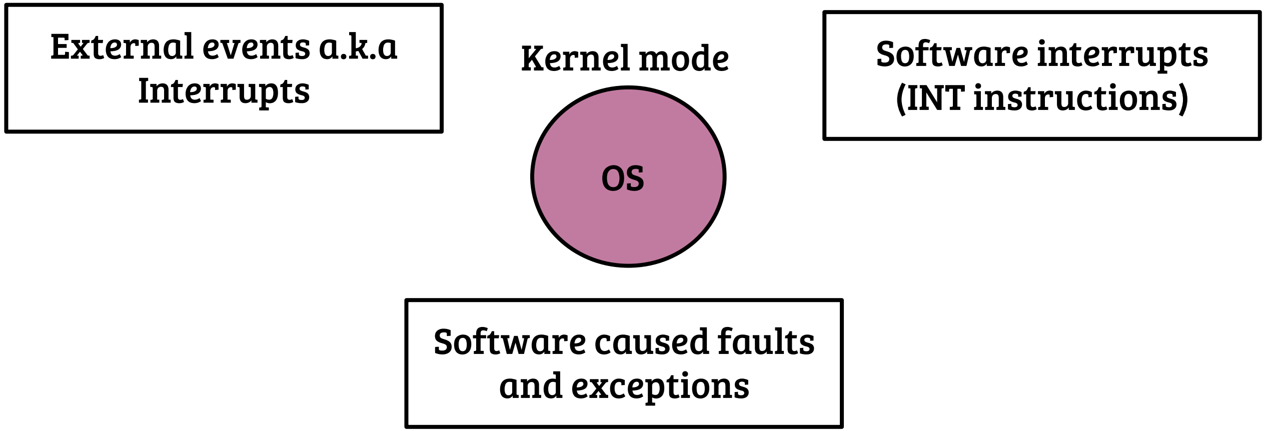
- What kind of support is needed from the hardware?
- CPU privilege levels, switching, entry points and handlers
- X86 support
  - privilege levels (ring-0 to ring-3)
  - interrupt descriptor table to define handlers for hardware and software entry points (system calls, interrupts, exceptions)
  - entry point behavior can be defined by the OS to enforce limitations on the user space execution

Recap: Limited direct execution support in X86

- What kind of support is needed from the hardware?
- CPU privilege levels, switching, entry points and handlers
- X86 support
  - privilege levels (ring-0 to ring-3)
  - interrupt descriptor table to define handlers for hardware and software entry points (system calls, interrupts, exceptions)
  - entry point behavior can be defined by the OS to enforce limitations on the user space execution

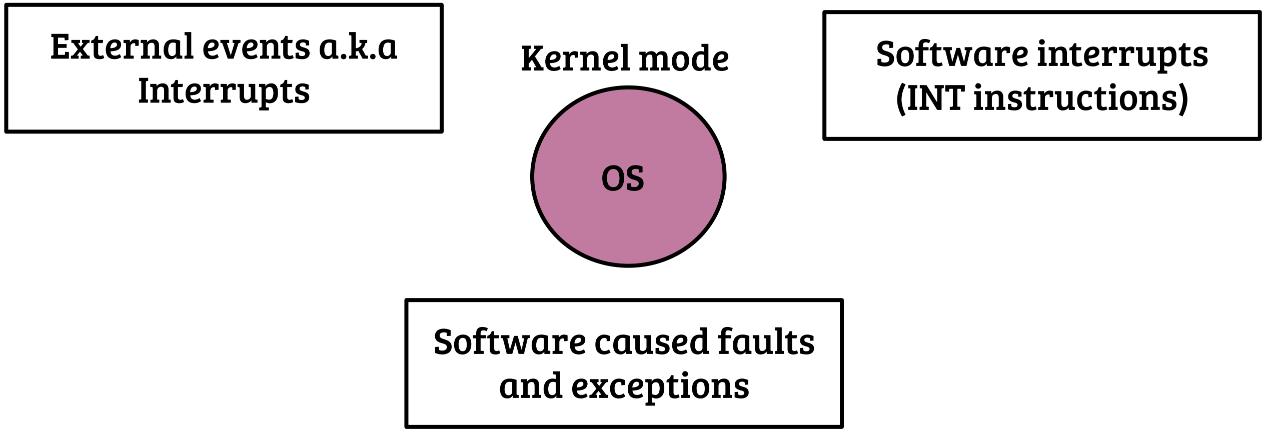
Agenda: Execution in privileged (kernel) mode

Post-boot OS execution



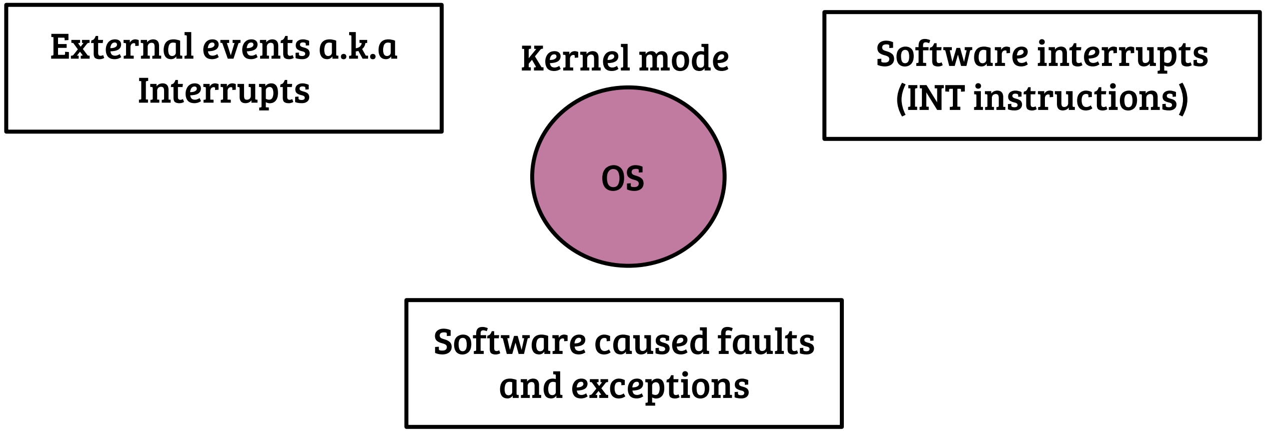
- OS execution is triggered because of interrupts, exceptions or system calls

Post-boot OS execution



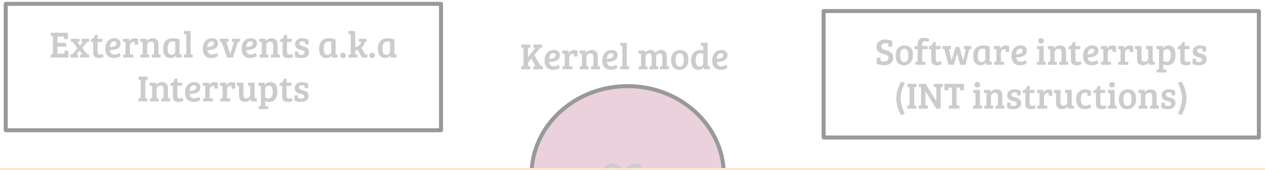
- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?

Post-boot OS execution



- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?
- The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return

Post-boot OS execution



- Does the OS need a separate stack?
- How many OS stacks are required?
- How the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

for this event to happen. What can go wrong and how to handle it?

- The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning

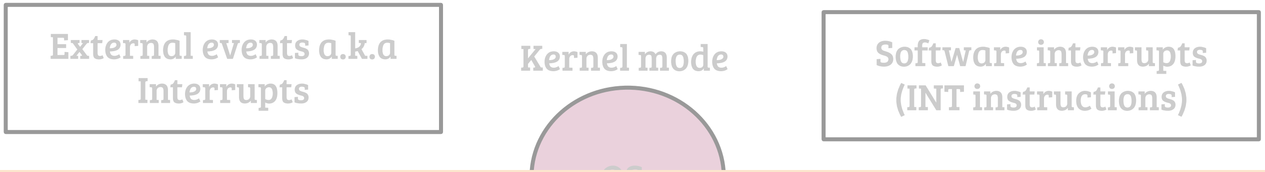
The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?
- On X86 systems, the hardware switches the stack pointer to the stack address configured by the OS

Post-boot OS execution



- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- How the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

to save the user execution state and restore it on return

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working?

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception
- What about external interrupts?

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception
- What about external interrupts?
  - Separate interrupt stacks are used by OS for handling interrupts

Post-boot OS execution

External events a.k.a

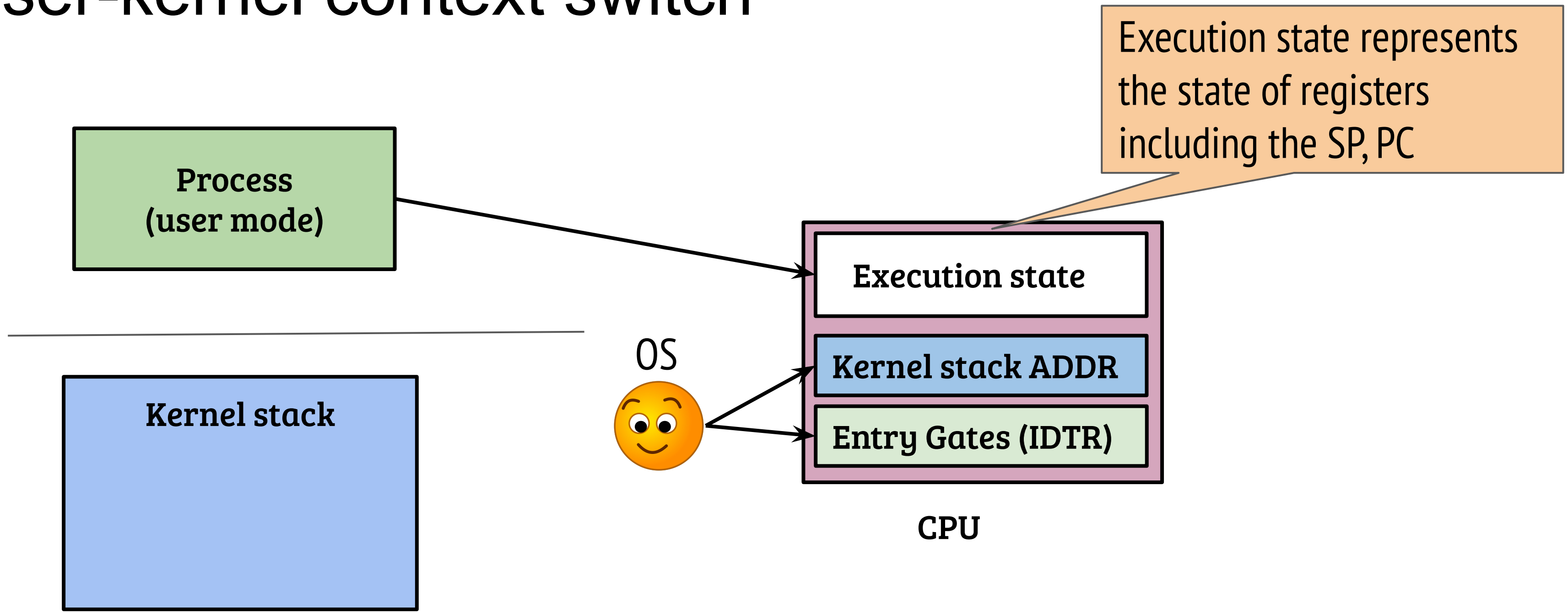
Kernel mode

Software interrupts

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How is the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

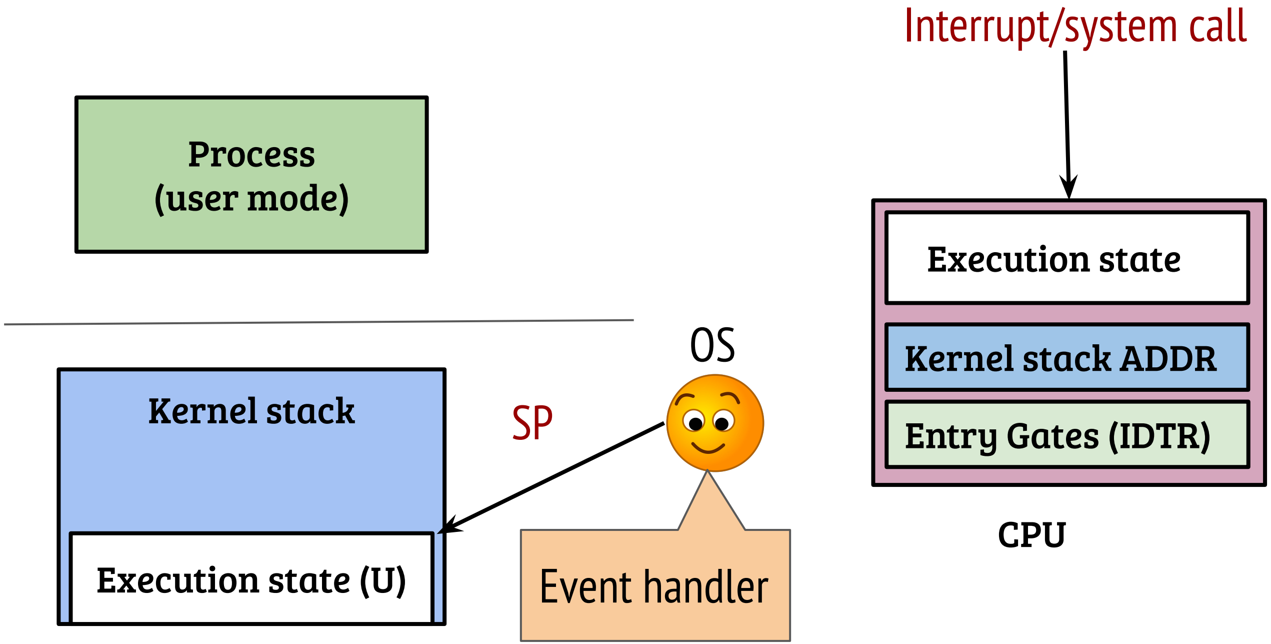
to save the user execution state and restore it on return

User-kernel context switch



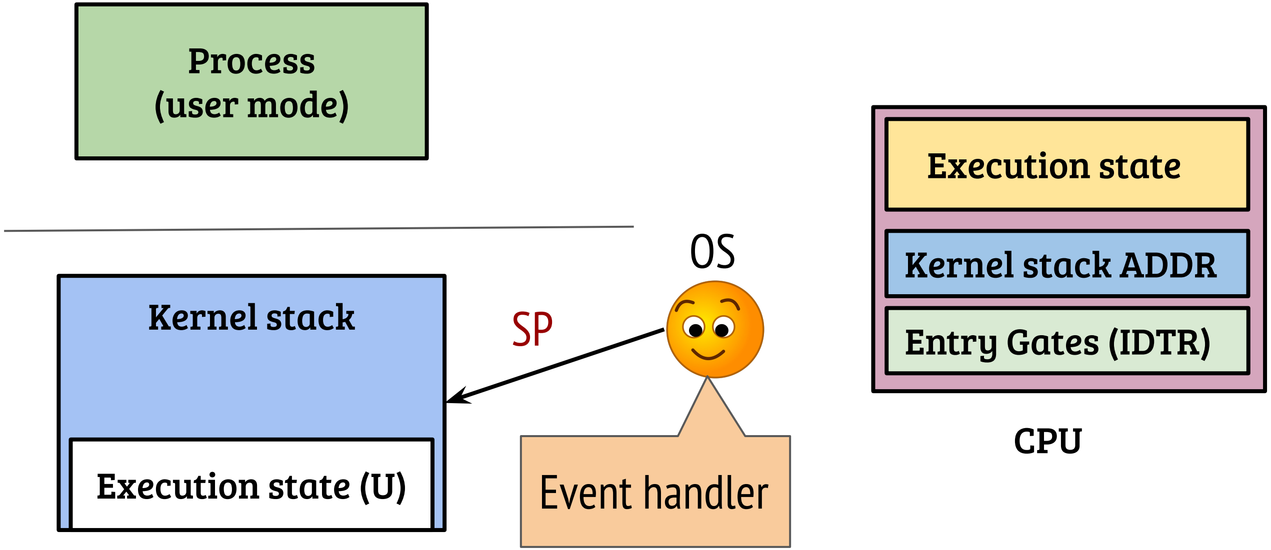
- The OS configures the kernel stack of the process before scheduling the process on the CPU

User-kernel context switch



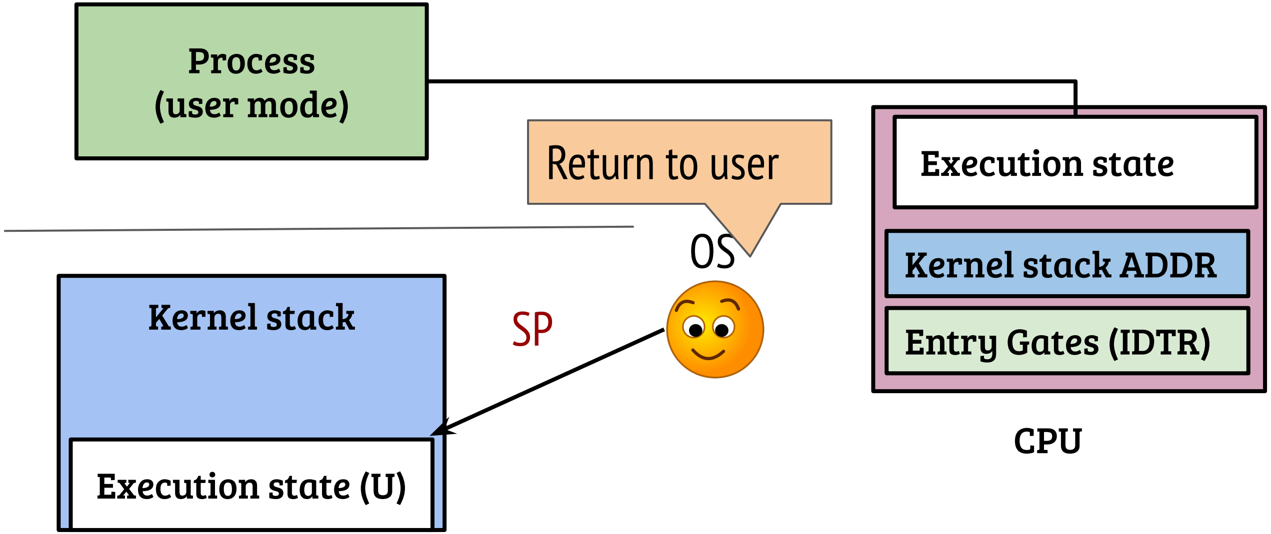
- The CPU saves the execution state onto the kernel stack
- The OS handler finds the SP switched with user state saved (fully or partially depending on architectures)

User-kernel context switch



- The OS executes the event (syscall/interrupt) handler
  - Makes uses of the kernel stack
  - Execution state on CPU is of OS at this point

User-kernel context switch



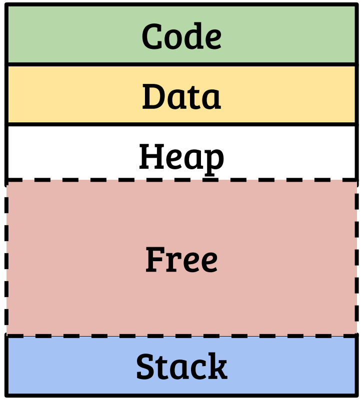
- The kernel stack pointer should point to the position at the time of entry
- CPU loads the user execution state and resumes user execution

Post-boot OS execution

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?

to save the user execution state and restore it on return

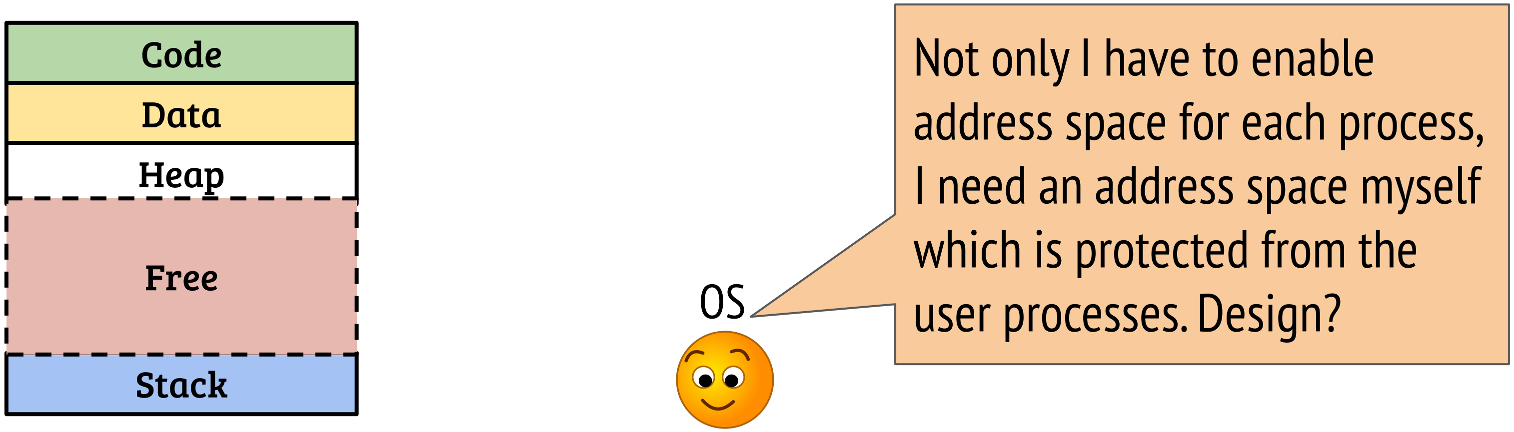
The OS address space



OS

Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

The OS address space



- Two possible design approaches
  - Use a separate address space for the OS, change the translation information on every OS entry (inefficient)
  - Consume a part of the address space from all processes and protect the OS addresses using H/W assistance (most commonly used)

Post-boot OS execution

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?
- A part of the process address space is reserved for OS and is protected

CS330: Operating Systems

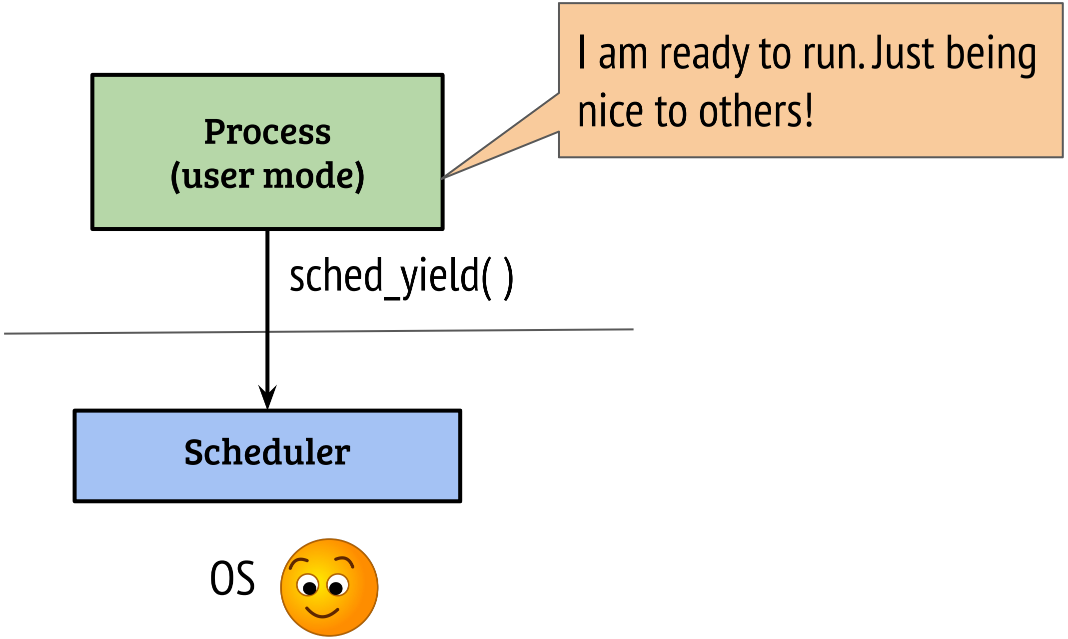
Process scheduling

Recap: OS execution, user-OS mode switch

- Which stack is used by the OS for kernel-mode execution?
- The hardware switches the SP to point it to a pre-configured per-process OS stack on mode switch
- How the user process state preserved and restored?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?
- A part of the process address space is reserved for OS and is protected

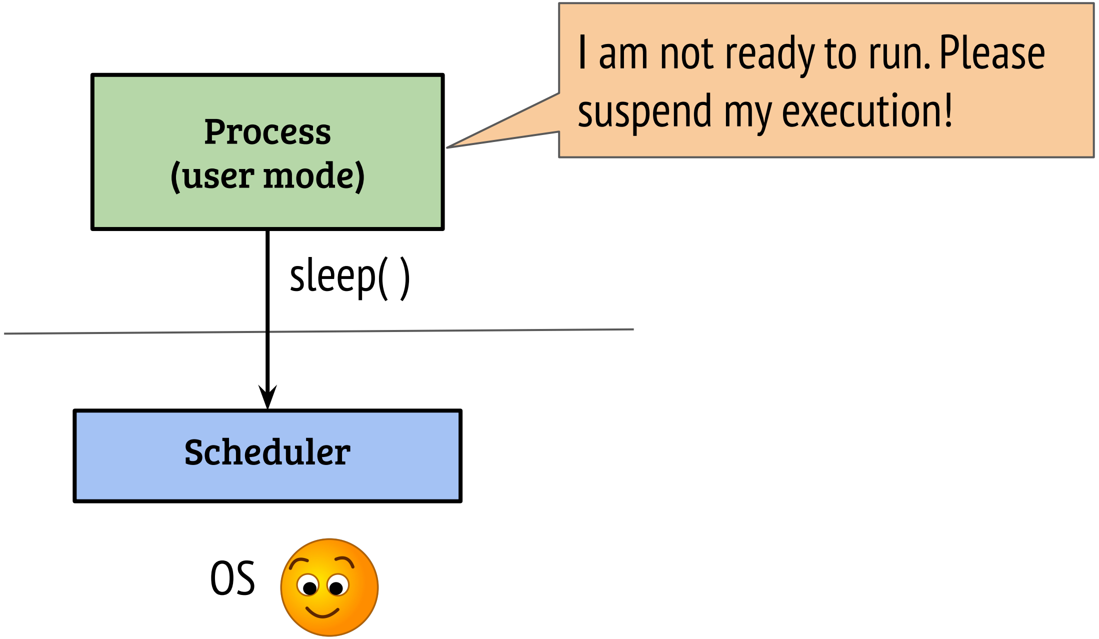
Agenda: Process context switch and scheduling

Triggers for process context switch



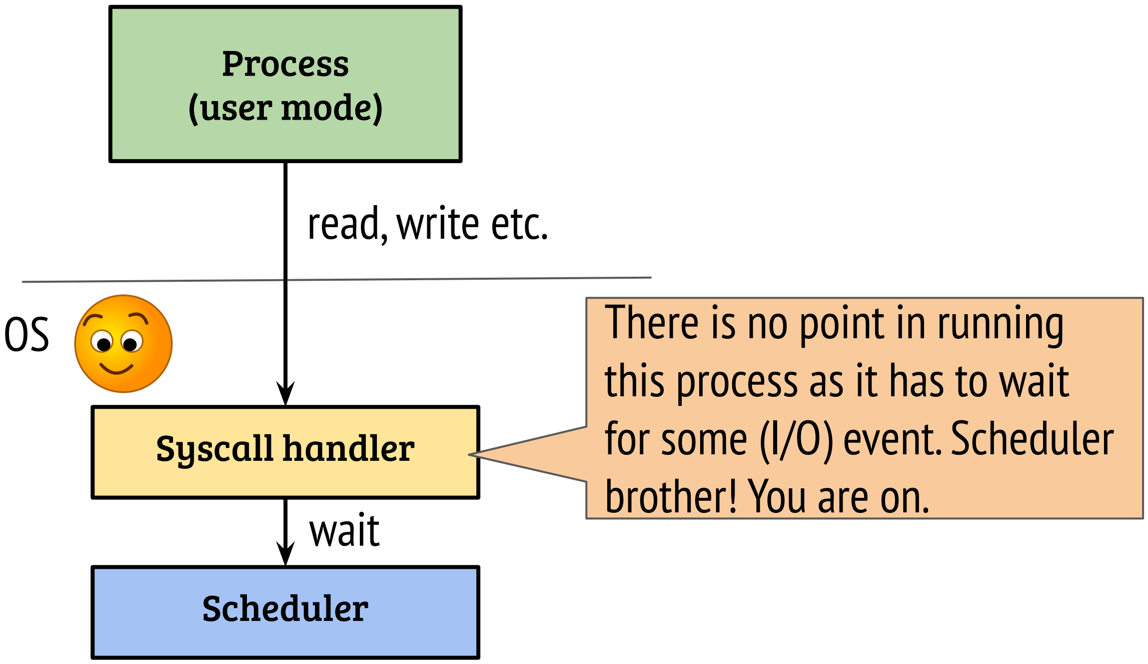
- The user process can invoke the scheduler through explicit system calls like sched_yield (see man page)

Triggers for process context switch



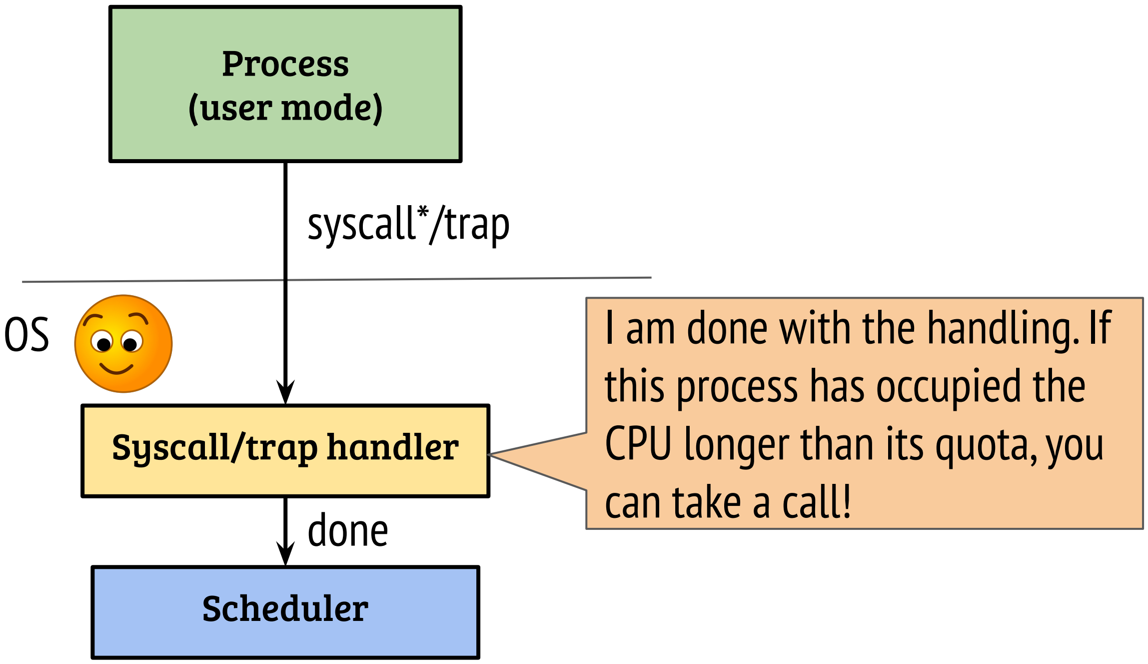
- The user process can invoke sleep( ) to suspend itself
  - sleep( ) is not a system call in Linux, it uses nanosleep( ) system call

Triggers for process context switch



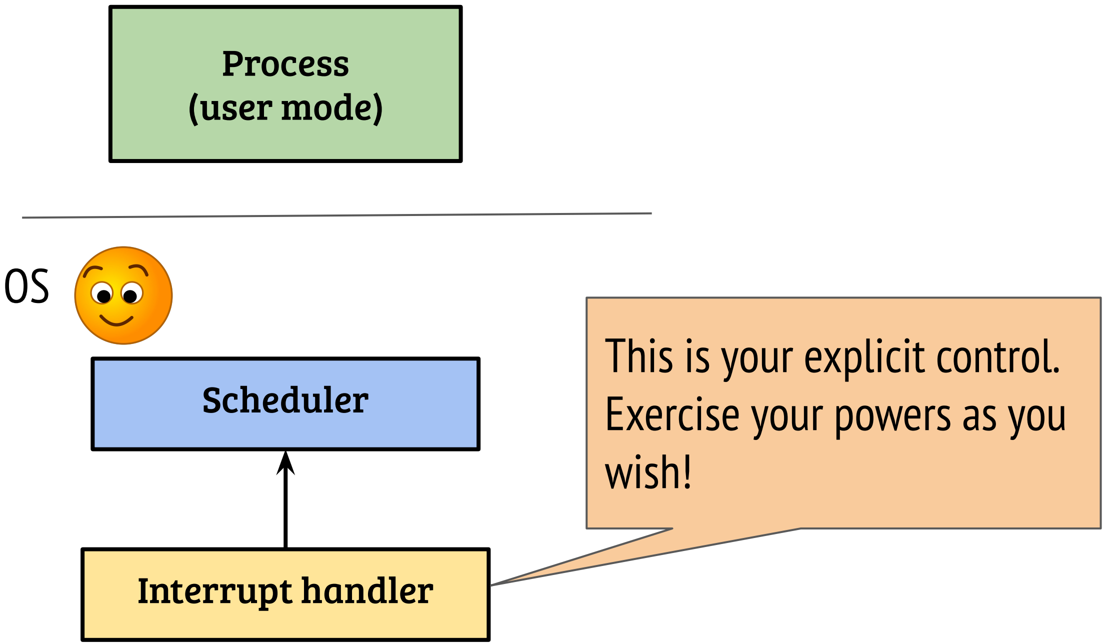
- This condition arises mostly during I/O related system calls
  - Example: read( ) from a file on disk

Triggers for process context switch



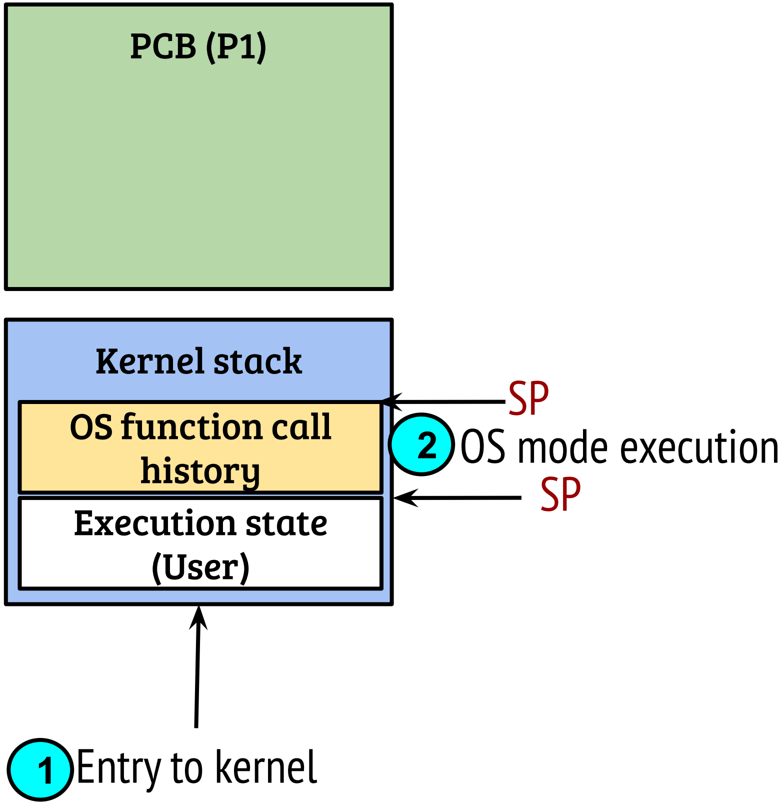
- The OS gets the control back on every system call and exception
- Before returning from syscall, the schedule can deschedule

Triggers for process context switch

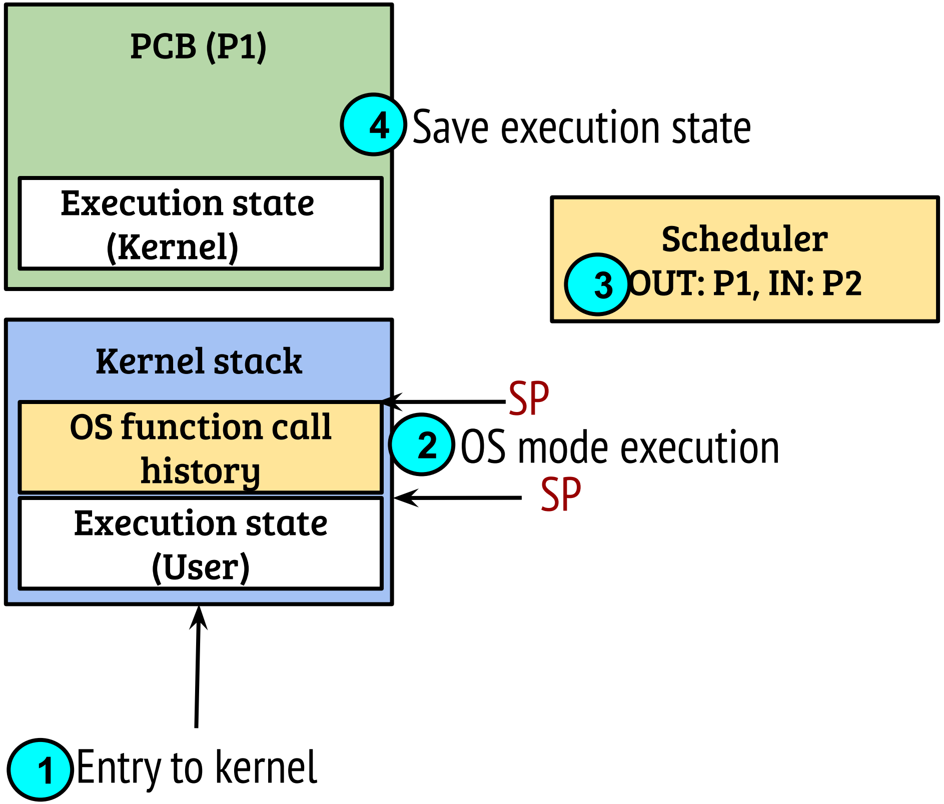


- Timer interrupts can be configured to generate interrupts periodically or after some configured time
- The OS can invoke the scheduler after handling any interrupt

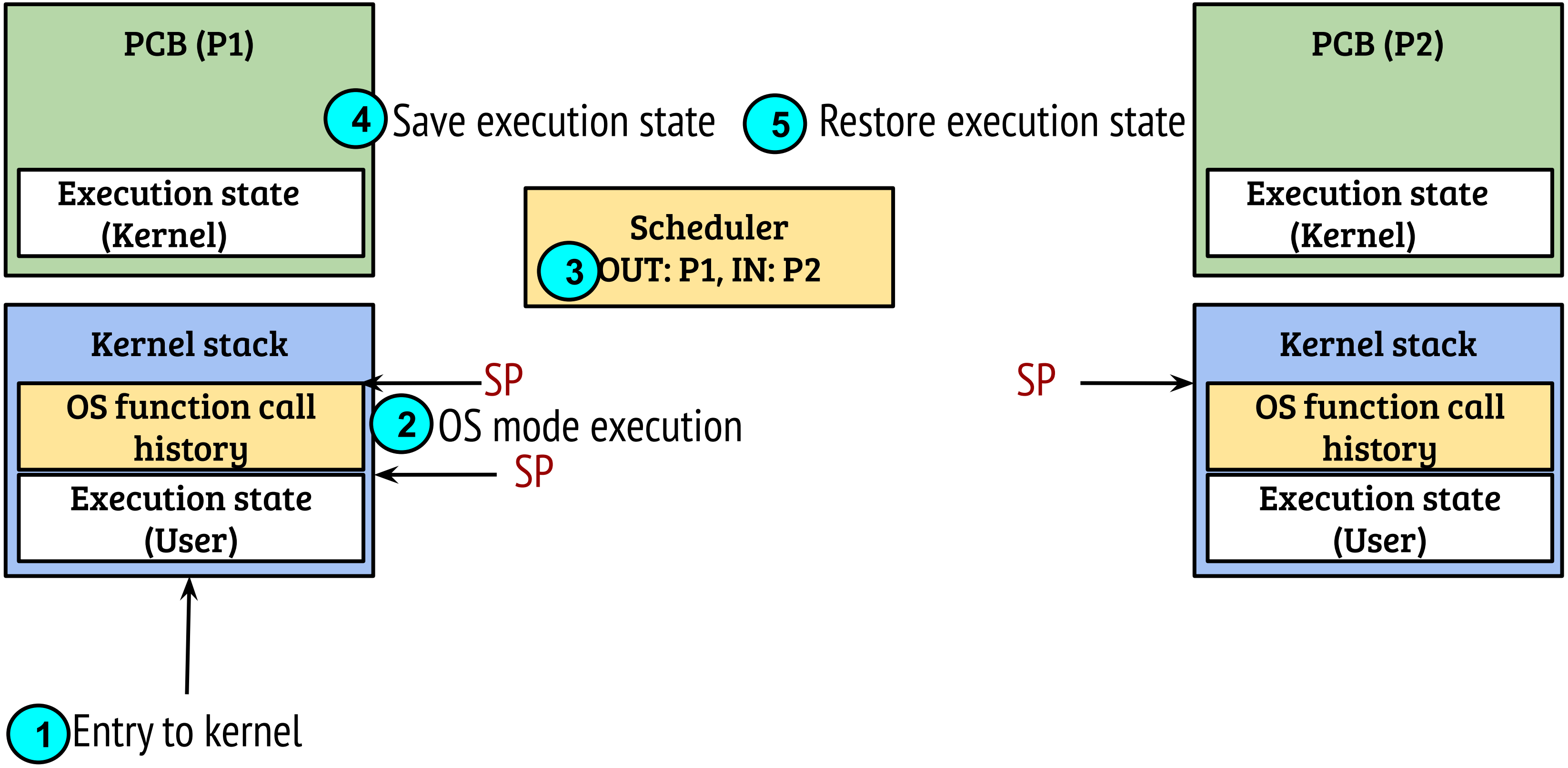
Process context switch



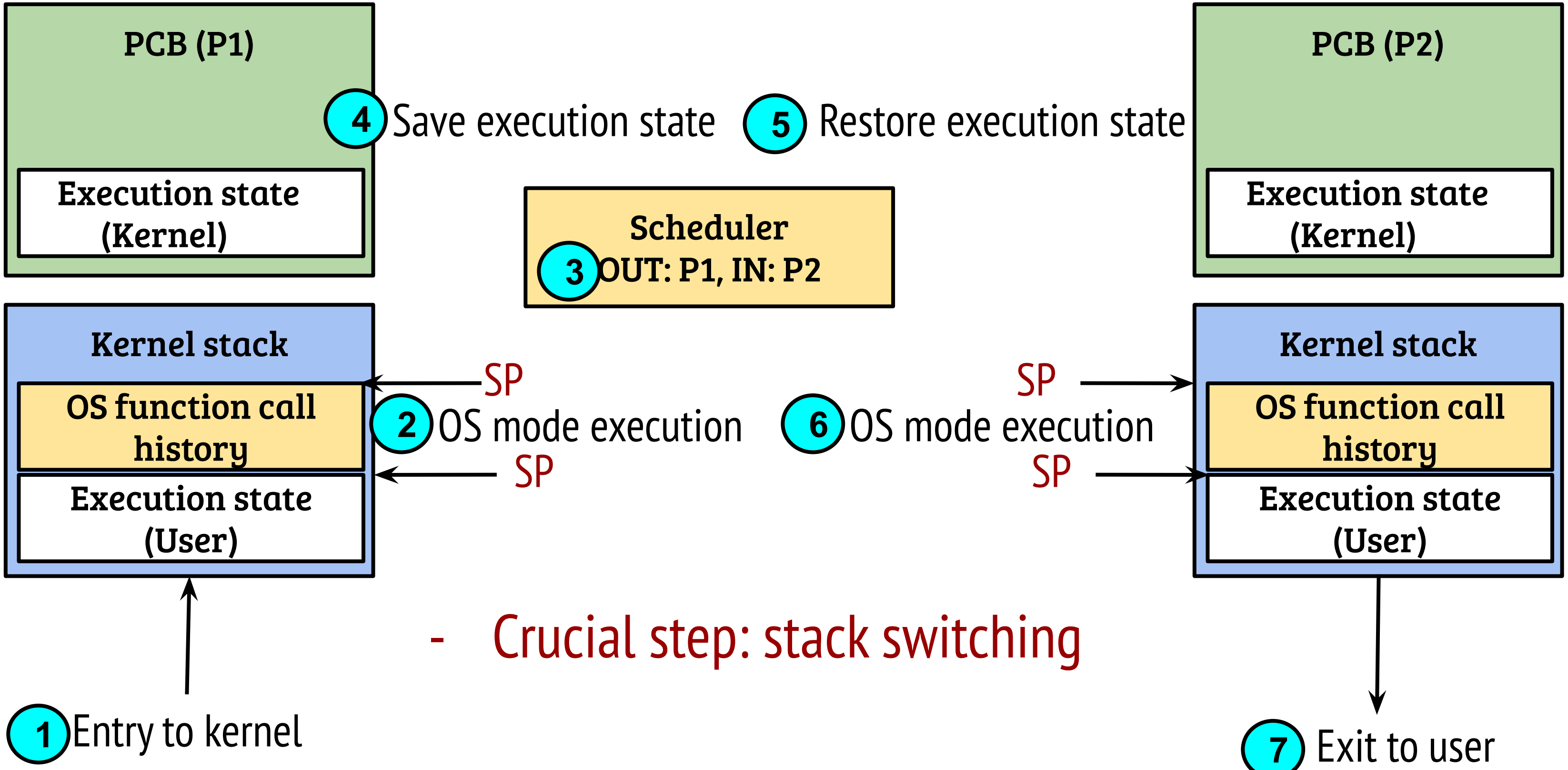
Process context switch



Process context switch

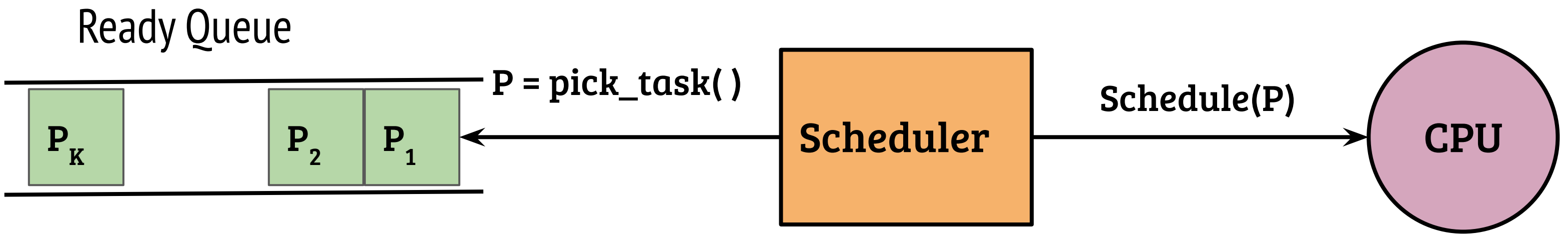


Process context switch



- Crucial step: stack switching

Scheduling



- A queue of processes ready to execute is maintained
- The scheduler decides to pick the next process based on some scheduling policy and performs a context switch
- The outgoing process is put back to ready queue (if required)

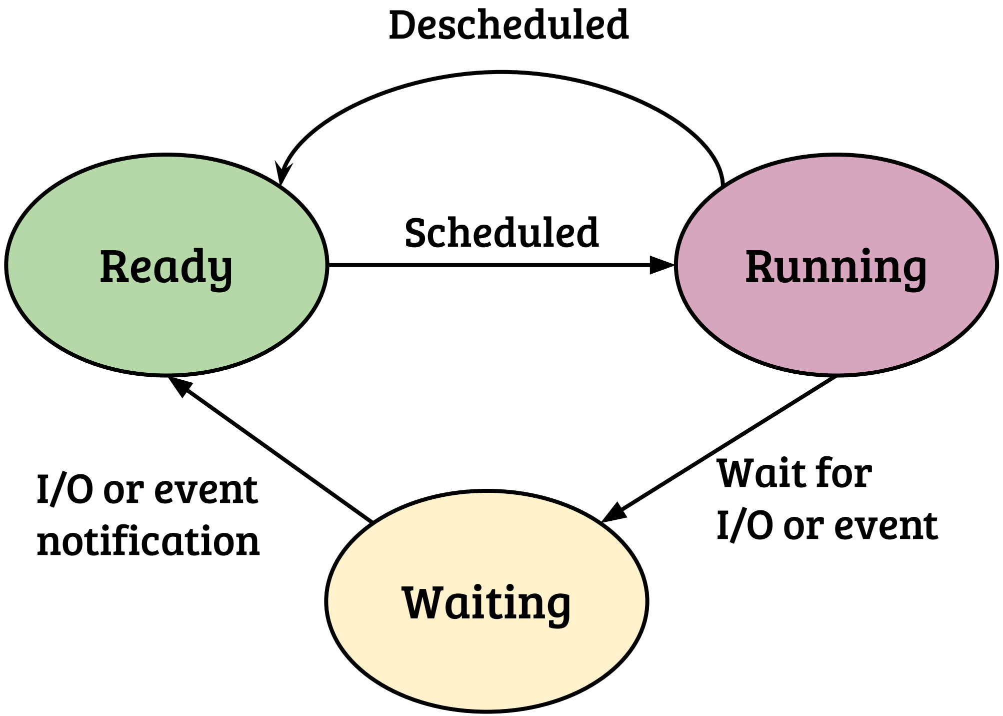
Scheduling

- How is the list of ready processes managed?
- What if there are no processes in ready queue? Can that happen?
- Can we classify the schedulers based on how they are invoked?
- What is a good scheduling strategy?

policy and performs a context switch

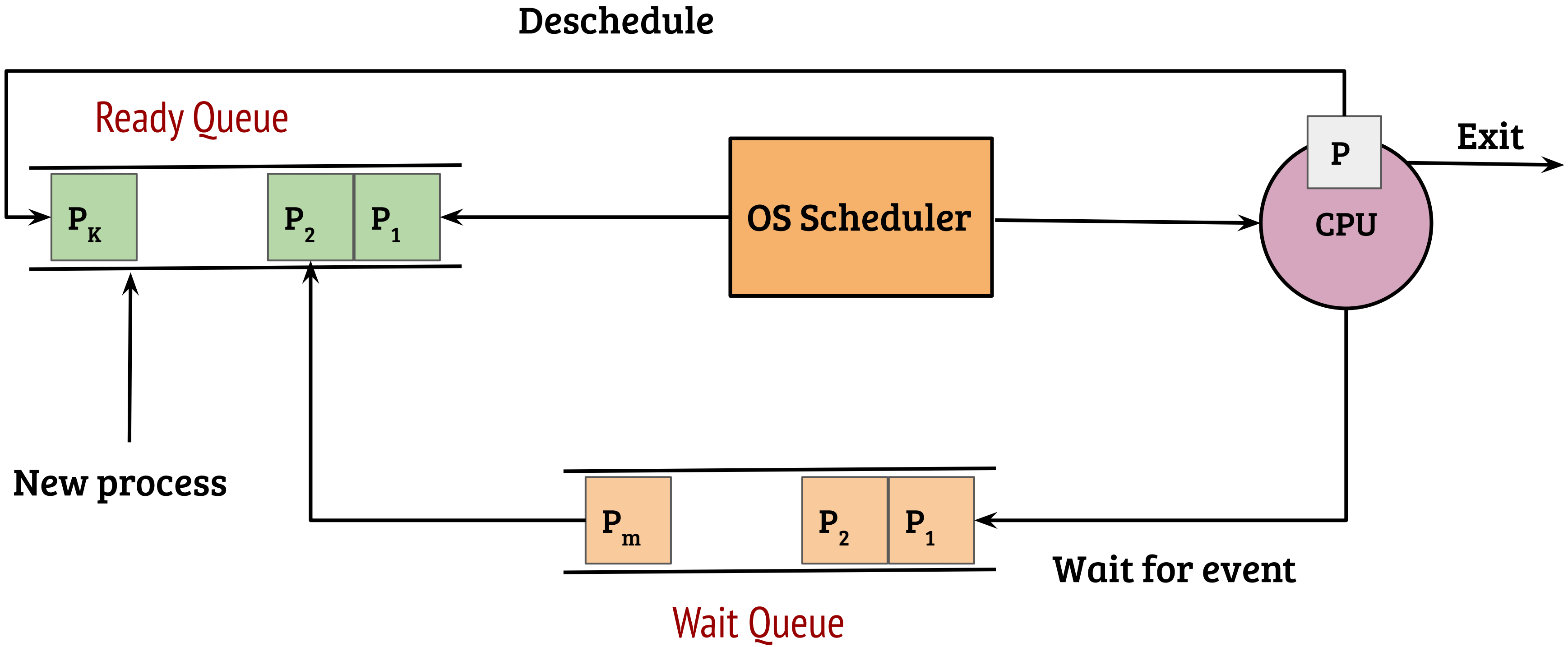
- The outgoing process is put back to ready queue (if required)

Process states and transitions



- Most processes perform a mixture of CPU and I/O activities
- When the process is waiting for an I/O, it is moved to waiting state
- A process becomes ready again when the event completion is notified (e.g., a device interrupt)

Scheduler overview



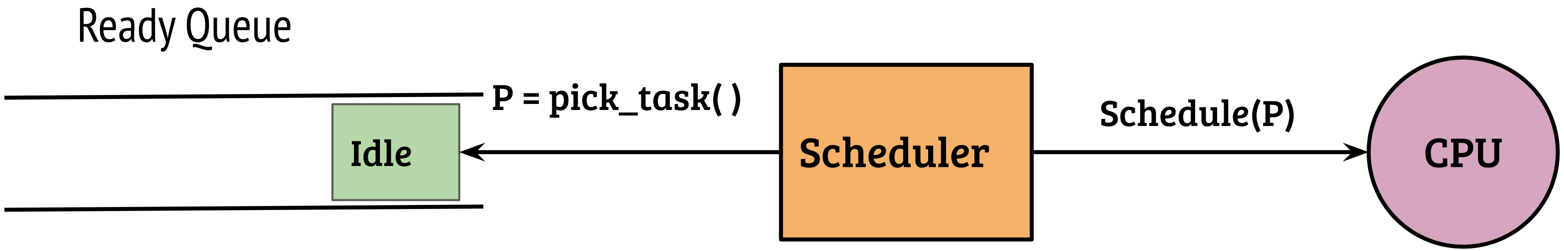
Scheduling

Ready Queue

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- Can we classify the schedulers based on how they are invoked?
- What is a good scheduling strategy?

- The outgoing process is put back to ready queue (if required)

System idle process



- There can be an instance when there are zero processes in ready queue
- A special process (system idle process) is always there
- The system idle process halts the CPU
- HLT instruction on X86_64: Halts the CPU till next interrupt

Scheduling

Ready Queue

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- There is always an idle process which executes HLT
- Can we classify the schedulers based on how they are invoked?
- What is a good scheduling strategy?

Scheduling: preemptive vs. non-preemptive

- There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
    - Process termination
    - Process explicitly yields the CPU
    - Process waits/blocks for an I/O or event

Scheduling: preemptive vs. non-preemptive

- There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
  - Process termination
  - Process explicitly yields the CPU
  - Process waits/blocks for an I/O or event
- The OS may invoke the scheduler in other conditions (preemptive)
  - Return from system call (specifically fork( ))
  - After handling an interrupt (specifically timer interrupt)

Scheduling

Ready Queue

- How is the list of ready processes managed?
- Each process is associated with three primary states: Running, Ready and Waiting. A process can moved to waiting state from running state, if needed.
- What if there are no processes in ready queue? Can that happen?
- There is always an idle process which executes HLT
- Can we classify the schedulers based on how they are invoked?
- Non-preemptive: triggered by the process, Preemptive: OS interjections
- What is a good scheduling strategy?

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
    - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
    - Objective: *Minimize waiting time*

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*
- Response time: Waiting time before first execution
  - Objective: *Minimize response time*

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*
- Response time: Waiting time before first execution
  - Objective: *Minimize response time*
- Average value of above metrics represent the average efficiency

Scheduling metrics

- Turnaround time: Time of completion - Time of arrival
  - Objective: *Minimize turnaround time*
- Waiting time: Sum of time spent in ready queue
  - Objective: *Minimize waiting time*
- Response time: Waiting time before first execution
  - Objective: *Minimize response time*
- Average value of above metrics represent the average efficiency
- Standard deviation represents fairness across different processes