CSE2431 – Lecture Topic 2
Process (part 2)

Process (Part 2)

Instructor: Luan Duong, Ph.D.

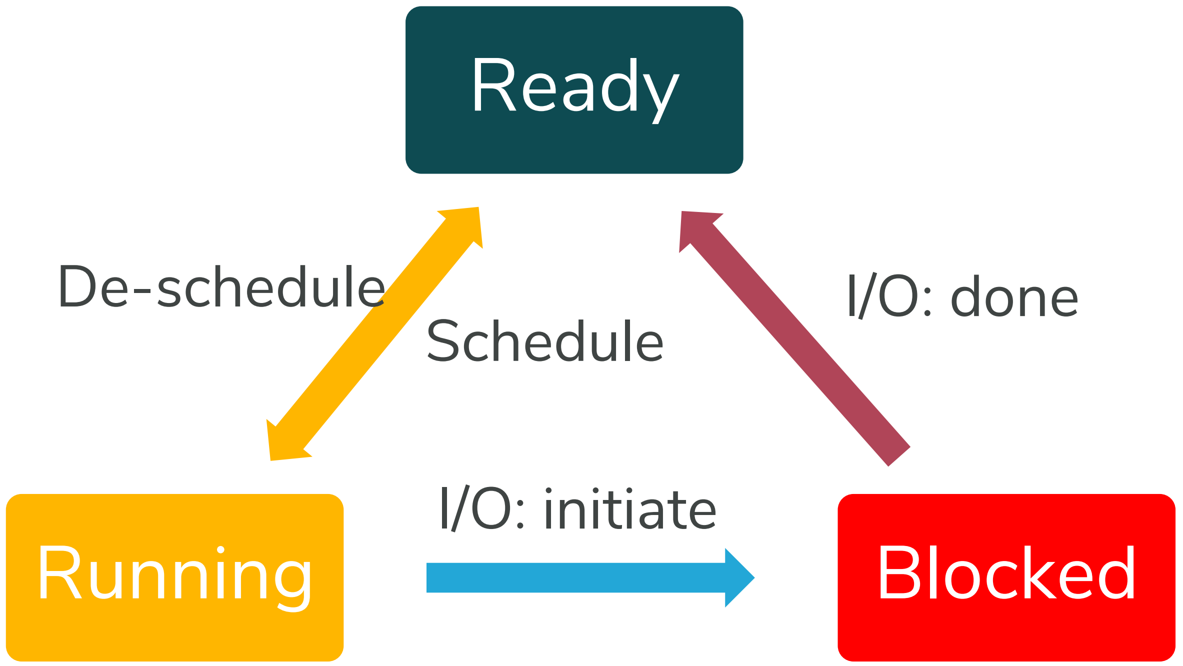
CSE 2431: Introduction to Operating Systems

Reading: Chapter 4-5 in required textbook

Lecture materials referred from previously taught course by Dr. Yang Wang and Dr. Adam C. Champion

Last Lecture: Process status

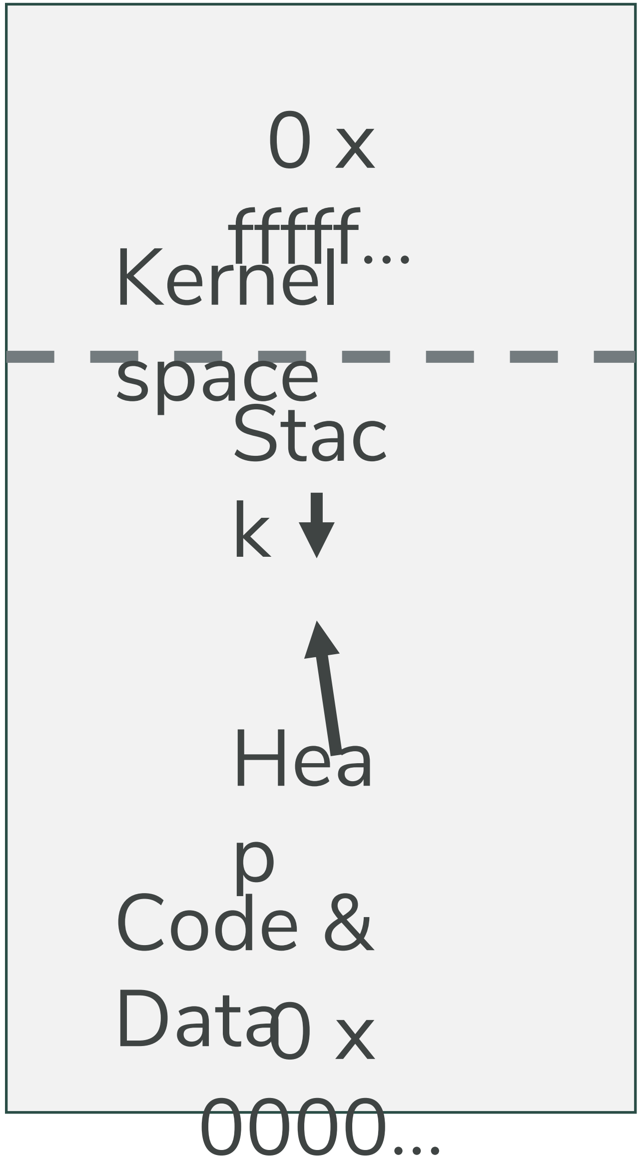
- Running: a process is being executed by a CPU
- Ready: a process is ready to run but is not running
- Blocked: a process is waiting on some event to take place



Last lecture: Process creation & memory

- *fork()*, *wait()* and *exec()*

```
int rc = fork();
if (rc < 0) {
   // fork failed
   fprintf(stderr, "fork failed\n");
   exit(1);
} else if (rc == 0) {
   // child (new process)
   printf("child (pid:%d)\n", (int) getpid());
} else {
   // parent goes down this path (main)
   printf("parent of %d (pid:%d)\n",
          rc, (int) getpid());
}
```

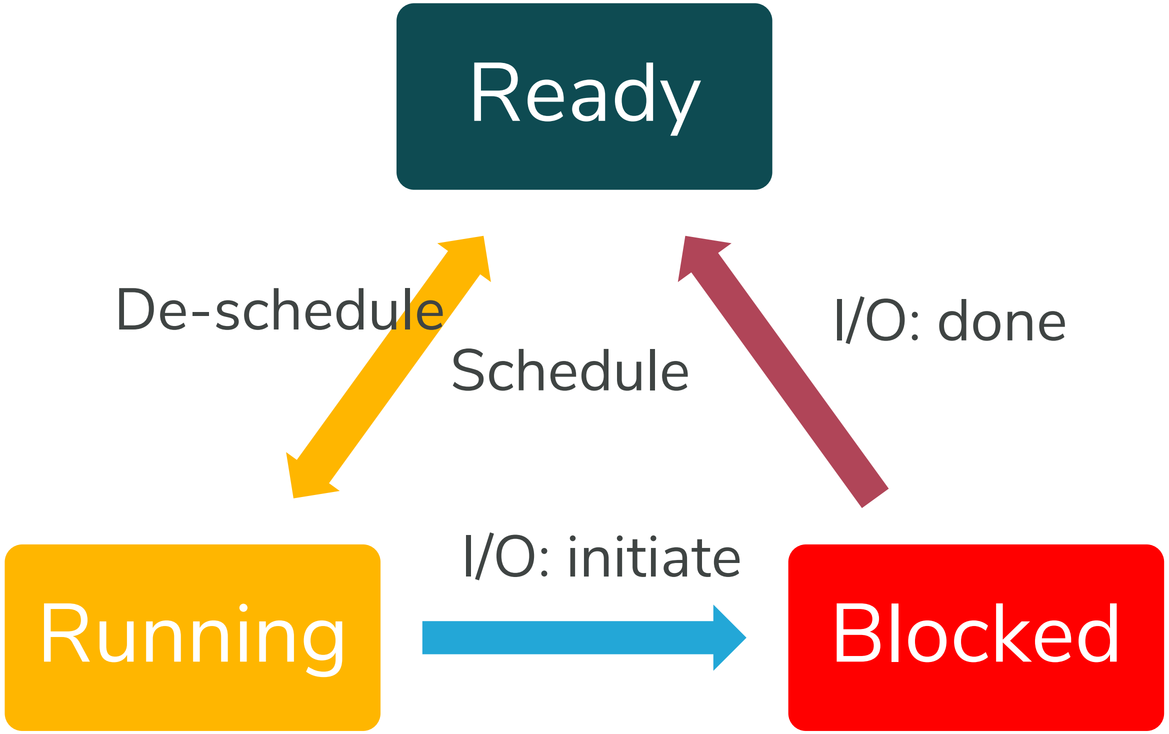


Outline: Process

- What is a process?
- Process States; Process Control Block (PCB)
- Process Creation; *fork* command
- Process Memory Layout
- Process Scheduling
- Context Switch
- Inter-Process Communication
- Client-Server Communication

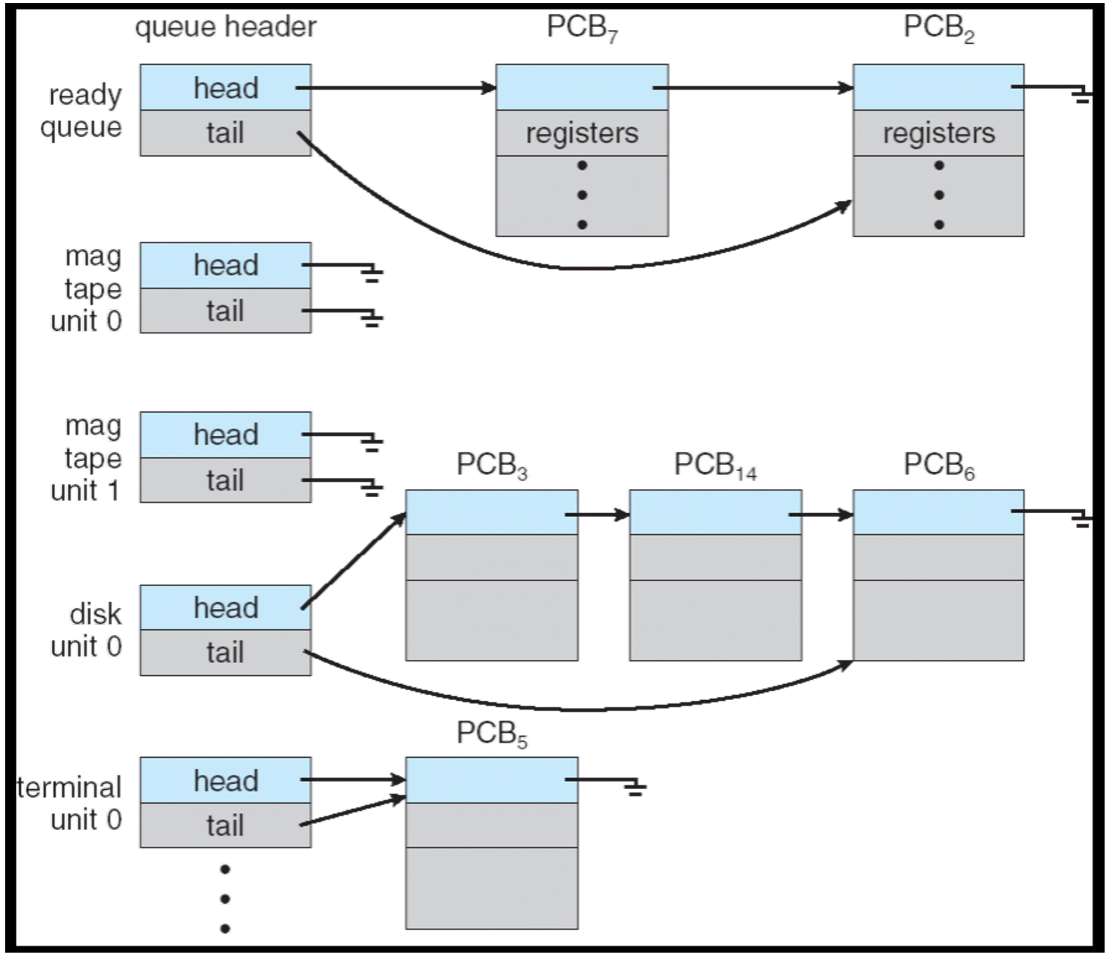
Process scheduling

- Remember this?
  - A process is the basic unit for virtualization.
  - OS virtualizes CPU by **time sharing**
    - OS runs one process for some time, stops it, then runs another process, …
    - Each process becomes slower, but users cannot tell, because computers are much faster than human beings
  - How can OS control the behavior of a process?



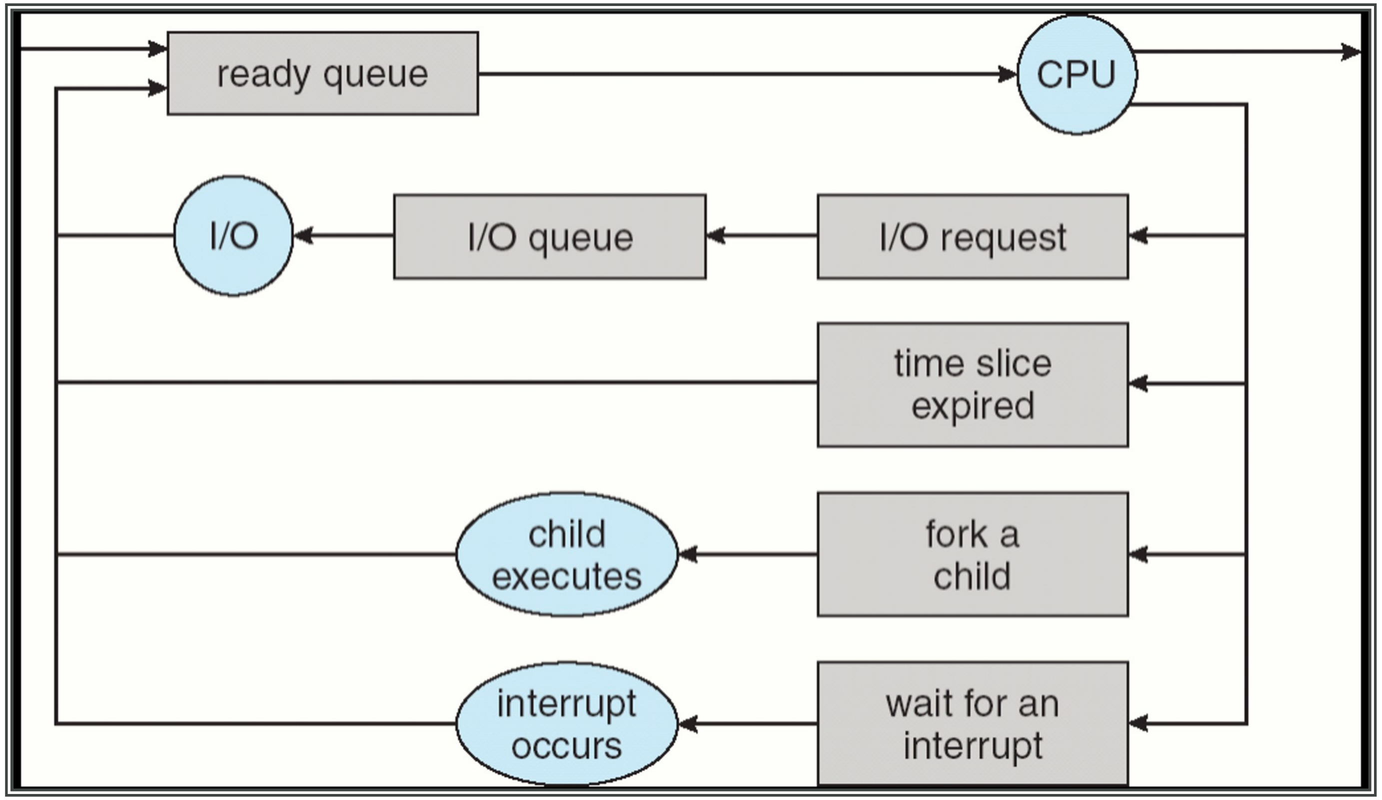
Process scheduling

- **Job queue:** set of all processes in the system
- **Ready queue:** set of all processes in main memory, ready and waiting to run
- **Device queues:** set of processes waiting for an I/O device
- **Processes migrate among various queues**



Process scheduling

- **Job queue:** set of all processes in the system
- **Ready queue:** set of all processes in main memory, ready and waiting to run
- **Device queues:** set of processes waiting for an I/O device
- **Processes migrate among various queues**



Process scheduling

- **Long-term scheduler** (or job scheduler)
  - Selects which processes should be brought into the ready queue
  - Invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - Controls the degree of multiprogramming
  - Should balance different types of processes:
    - **I/O-bound process** spends more time doing I/O than computation; many short CPU bursts
    - **CPU-bound process** spends more time doing computation; few long CPU bursts
- **Short-term scheduler** (or CPU scheduler)
  - Selects which process should be executed next and allocates CPU
  - Invoked very frequently (milliseconds) $\Rightarrow$ must be fast!

Process scheduling

- Solution 1: Interpreter

Process

- A process sends its instructions to the interpreter

Interprete r

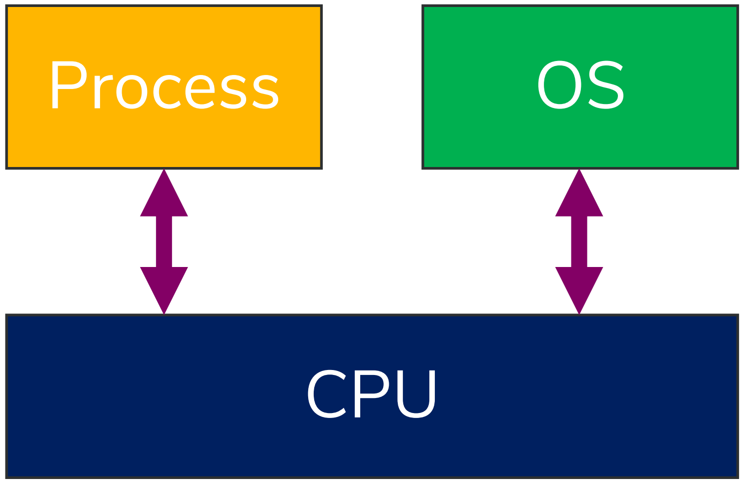
- Interpreter checks these instructions
- If appropriate, the interpreter translates them into Hardware instructions.
- Disadvantage: overhead for checking and translation

CPU

- CPU: Handle the process(es) given from the interpreter
- Advantage: easy to control process behavior(s)
- Approach used by Java and Python, but OS wants better performance

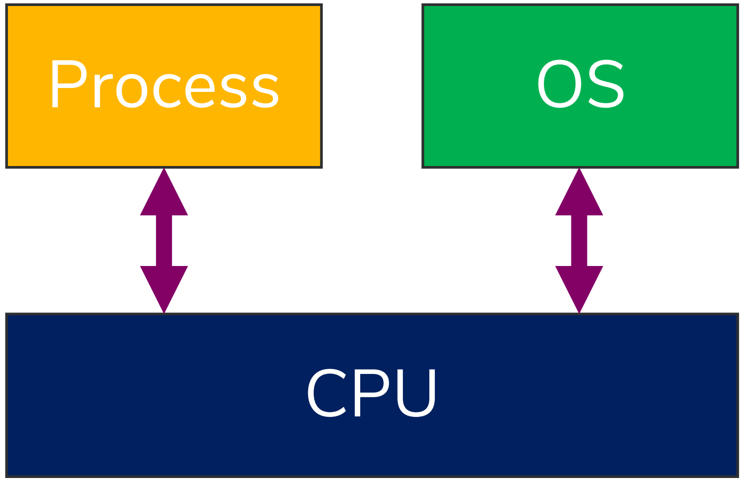
Process scheduling

- Solution 2: Limited Direct Execution
  - A process sends its instructions directly to the CPU
  - OS collaborates with CPU to control the behavior(s) of the process.
  - Advantage: low overhead
  - Disadvantages: hard to control process behavior(s)
  - OS usually uses this approach



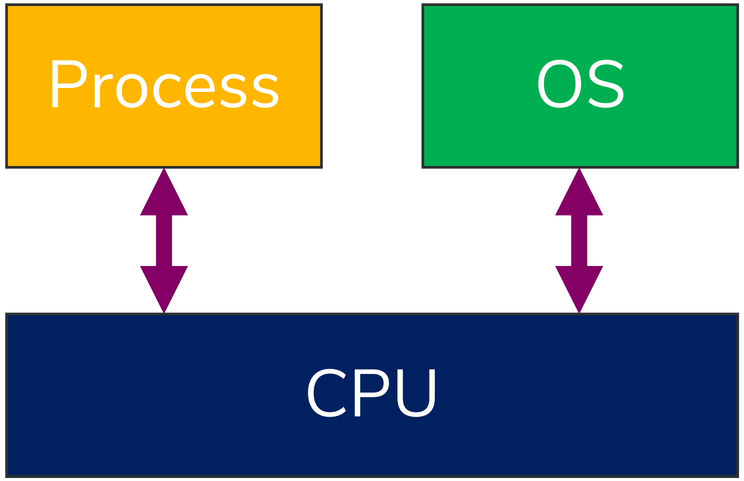
Process scheduling

- Solution 2: **Challenges**
  - When a process starts to run, it gains **full control** of the CPU
  - How to prevent a process from doing bad things?
    - A program can be malicious or simply buggy
    - It may try to read/write/delete other processes' data
    - This may violate virtualization
  - How to pause a process and switch to another process?
    - A malicious or buggy program may not want to give up CPU resources
    - Ideally, we should not need to worry about this when writing a program



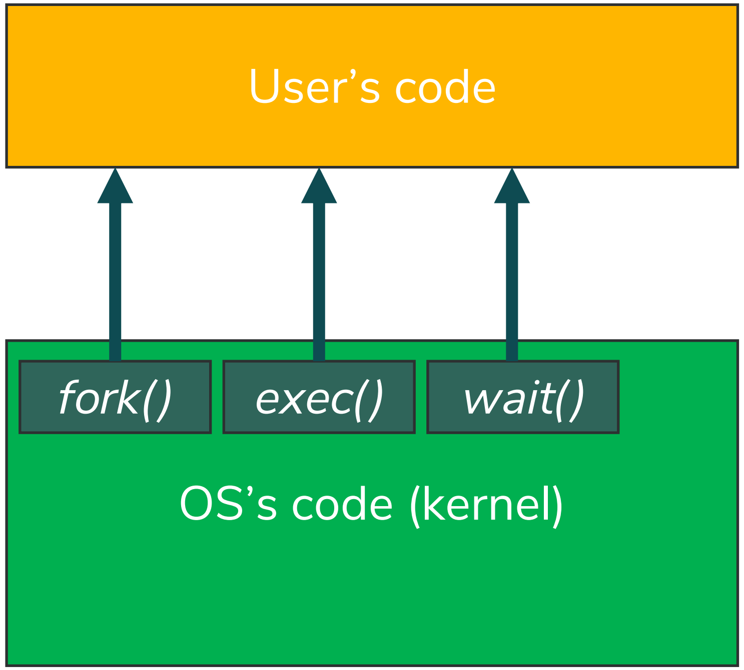
Process scheduling

- **Solution?**
  - 'Dangerous' instructions will be called **restricted operations** or **privileged instructions**.
  - What instructions are not 'dangerous'?
    - Normal operations such as arithmetic (add, sub, multiple)
    - Most memory-related operations (load, jump, conditional jump, etc.)
  - What are 'dangerous' instructions
    - i.e. write a file
  - Solution: system calls!



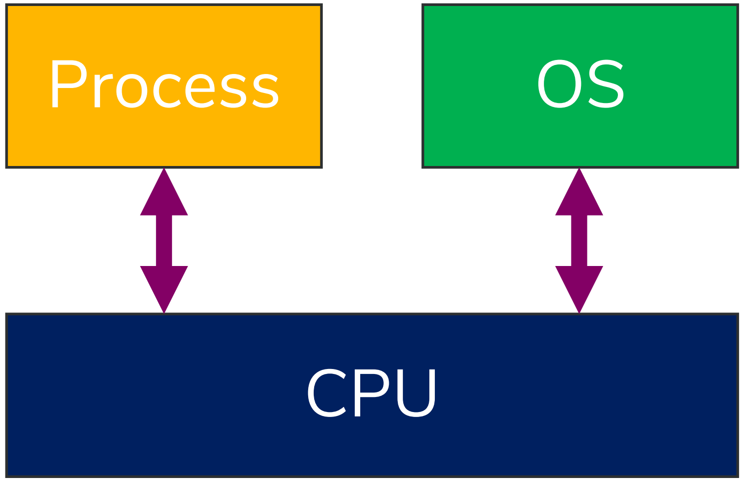
Process scheduling

- **Solution**
  - User's code cannot execute restricted operations
  - OS expose some functions to user's code
  - OS code can execute restricted operations:
    - Define functionalities for users: fork/exec/wait; file create/open/read/write/close, …
    - Execute restricted operations with appropriate checks (file write can only access data belonged to the user)
    - Programmer calls a system call like a library function.



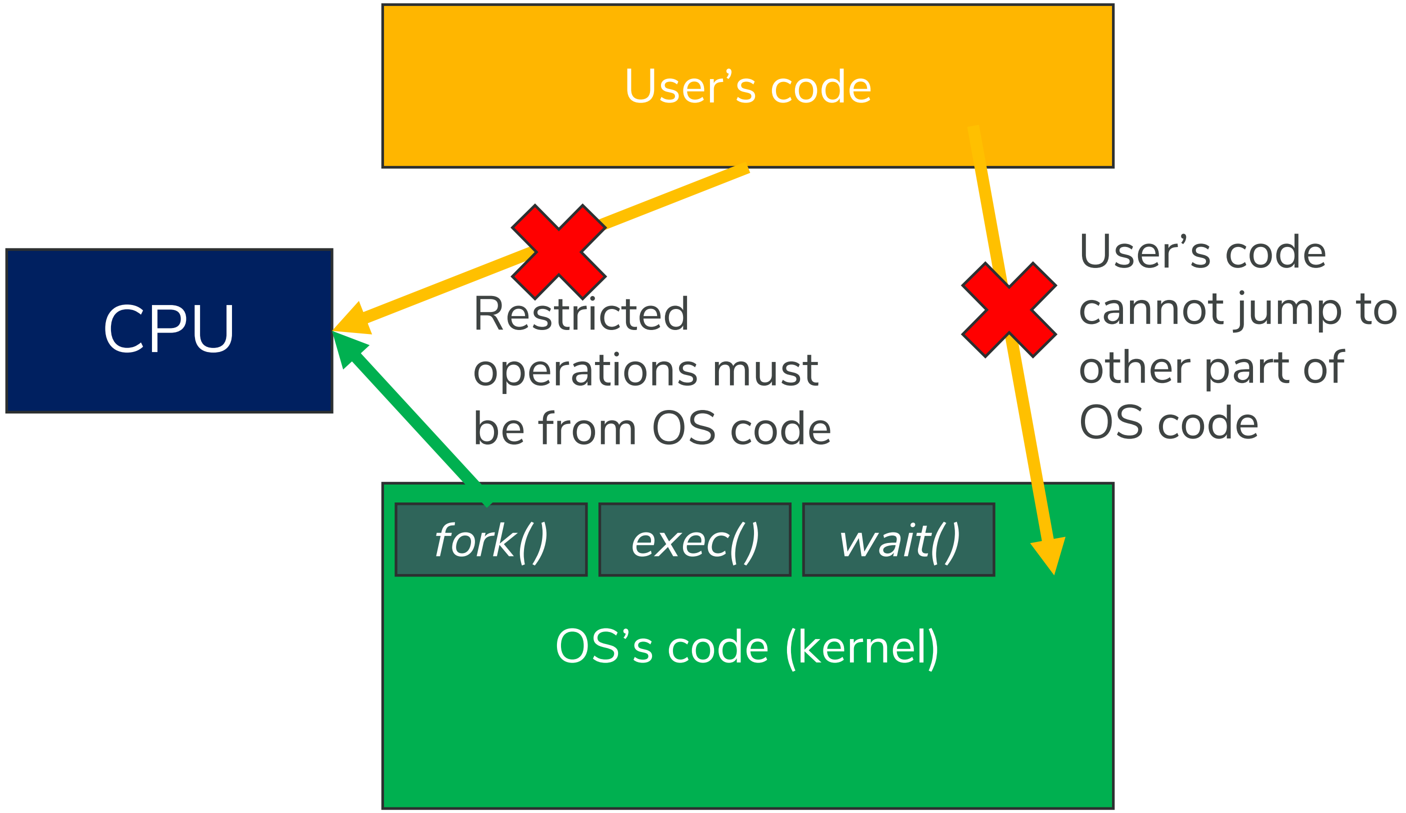
Process scheduling

- **Is the problem really solved?**
  - **NO!**
  - We already prevent a user program from executing restricted operations arbitrarily.
  - However, we still assume a program is not buggy and not malicious.
  - A buggy/malicious program can still issue restricted operations without making system calls?
    - Remember this? How can CPU tell whether a restricted operation is from a system call or not? CPU only sees a sequence of instructions



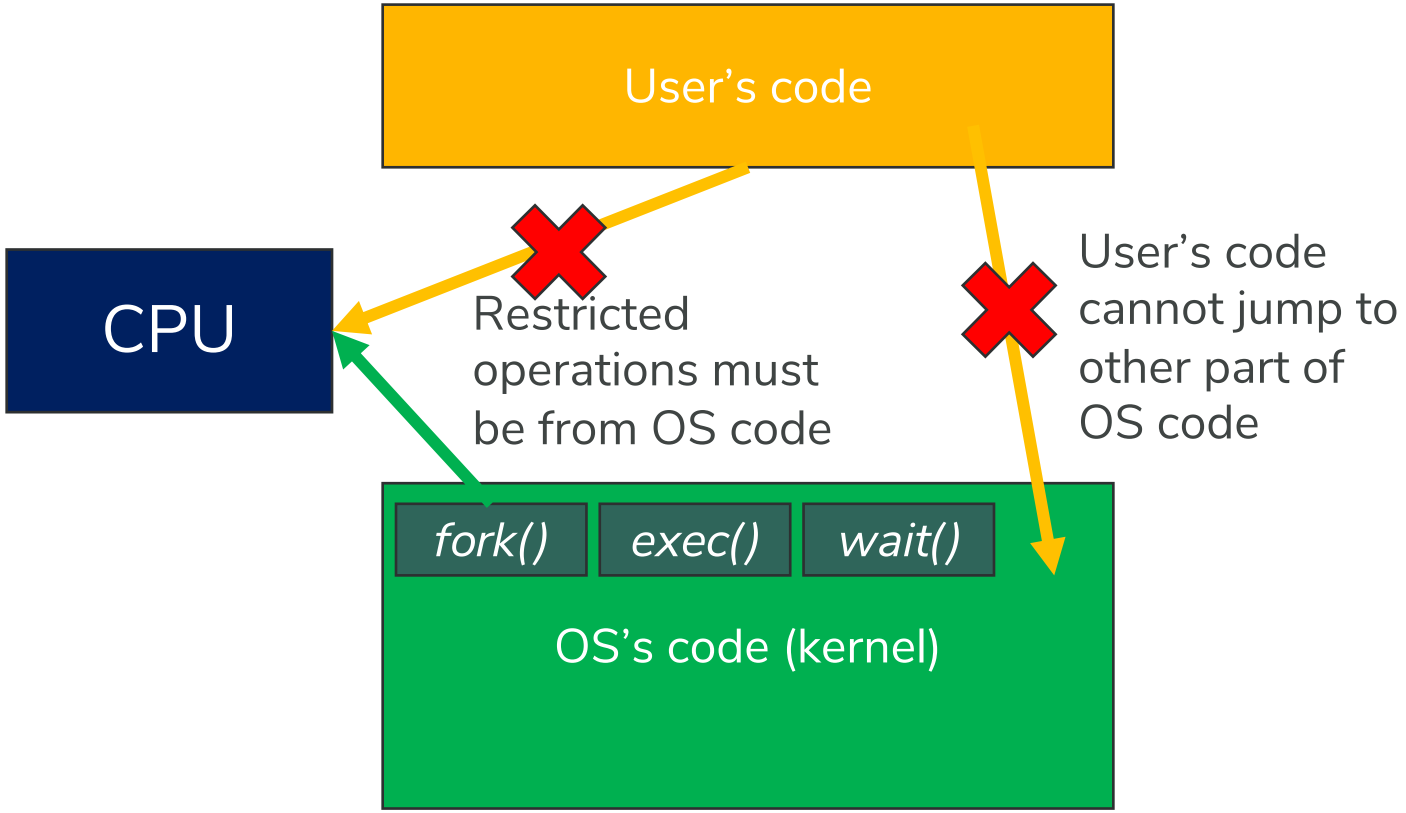
Process scheduling

- Requirement?
  - User's code **cannot** execute restricted operations.
  - OS code **can** execute restricted operations.



Process scheduling

- CPU's supports:
  - Trap-table
  - Kernel-mode and user-mode
  - Trap and return-from trap instruction



Process scheduling

- <u>Trap-table:</u>
  - Normal function calls are implemented by “jump” instruction. Inside a process, code can jump to arbitrary location!
  - When calling an OS function, this cannot happen arbitrarily
  - OS exposes locations of all system calls in a **trap-table**
  - CPU ensures that user’s code can only jump to locations defined in trap-table.

Process scheduling

- Kernel-mode and user-mode
  - To ensure only OS code can execute restricted operations
  - CPU maintains a bit
  - When this bit is set to **True** (**kernel mode**), OS code (including system call code) is executing
    - It can do anything including executing restricted operations.
  - When this bit is **False** (**user mode**), user code is executing
    - Only non-restricted operations are allowed
  - Have we really solved the problem with this?

Process scheduling

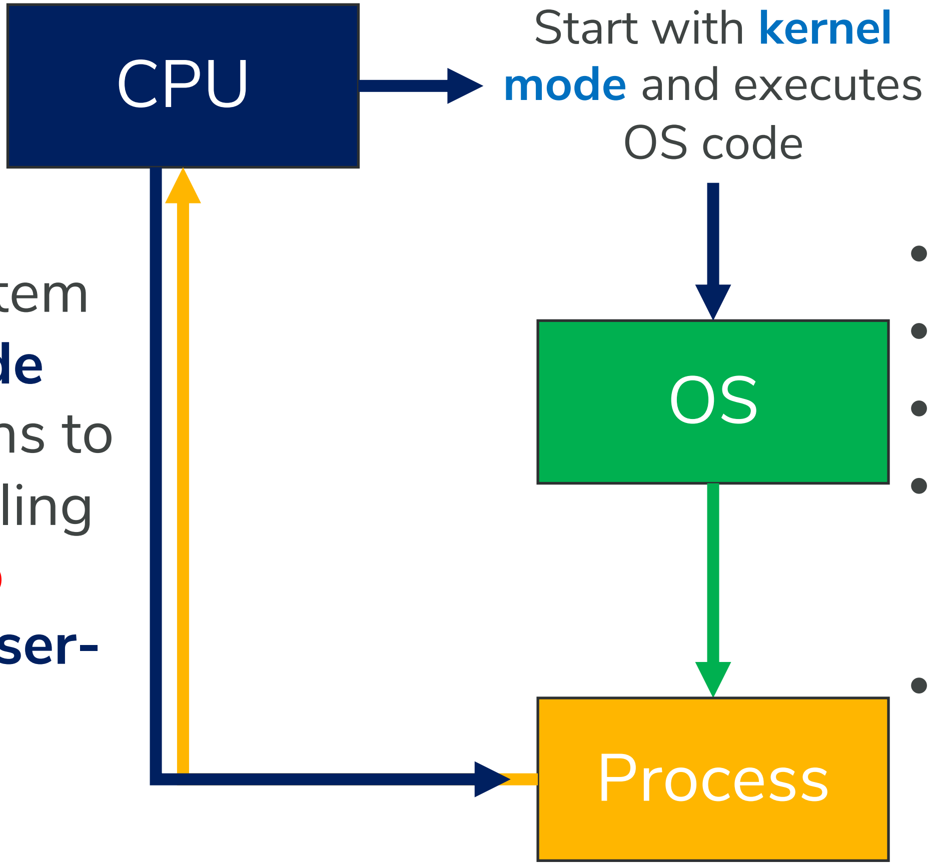
- <u>Kernel-mode and user-mode</u>
  - Suppose CPU provides a new instruction "**change kernel bit**"
    - Should it be a restricted operation or not?
  - Can we make it non-restricted?
    - NO. A malicious code can set kernel bit to True and executed restricted operations.
  - Should we make it restricted?
    - Sure, but how can the CPU check that it is from OS code?
  - How can CPU knows whether the kernel bit itself is set up appropriately?

Process scheduling

- Trap- and return-from-trap instruction
  - Modern CPU provides a special trap instruction that performs the following two operations **atomically**.
    - **Switch** from user mode to kernel mode
    - **Jump** to an address defined in the trap table
  - Key idea: we allow user's code to change kernel bit, but **after** doing so, the user's code **must jump** to a **system call**!
    - Then it is safe to make trap non-restricted.
  - OS makes sure a system call **switches to user mode** before returning
    - CPU provides a special return-from-trap instruction to do that

Process scheduling

- At boot time:



- **Executes** the system call in **kernel mode**
- System call returns to **user mode** by calling **return-from-trap**
- Process runs in **user-mode** again

- Sets up trap table
- Performs other initializations
- Switches to **user-mode**
- Starts a program/process in user-mode

- May make system call, jumping to a location in trap table and switching to **kernel mode**, by using the trap instruction

System call vs. normal function call

System call	Function call
Must run in **kernel mode**	Can run in any mode
Made by a **trap instruction**	Made by a jump instruction
Must be registered in the **trap table**	Can be put in any location (in mem)
Comes with **two context switches**	Only with one context switch

Outline: Process

- What is a process?
- Process States; Process Control Block (PCB)
- Process Creation; *fork* command
- Process Memory Layout
- Process Scheduling
- **Context Switch**
- Inter-Process Communication
- Client-Server Communication

How to pause a process?

- Remember when a process is running, OS is not running

- Cooperative approach: OS trusts the process to behave reasonably
  - OS can take control when the process is making a system call
  - OS provides an explicit “yield” system call
  - Problem: it violates virtualization; malicious or buggy processes can still halt the whole system by not making any system calls

- Non-cooperative approach: OS takes control anyway
  - Need hardware support

How to pause a process?

- Special hardware support—timer
  - A timer will trigger a timer event periodically
  - When a timer event is triggered, CPU will automatically jump to a predefined timer event handler
  - In this way, CPU provides a backdoor for OS.

- An OS can implement its own timer event handler and register it to the CPU (much like registering a system call)
  - An OS can implement the process switching logic there

How to resume a process?

- Remember a CPU decides the next instruction to execute by the program counter (PC) register
  - So, to resume a process, an OS should set PC to the location where the process is paused
- There are some other registers the OS needs to recover.
- In principle, an OS should save the state of a process when pausing it and load the state when resuming the process.
  - This is called a **context switch**.
  - It's very much like saving, loading PC game states

Context Switching

- Switches CPU from one process to another
- Performed by scheduler (dispatcher) with a special hardware support (timer)
- It includes:
  - Saving the state of the old process (such as registers);
  - Loading the state of the new process;
  - Flushing memory cache;
  - Changing memory mapping [Translation Lookaside Buffer (TLB)]

Context Switch Workflow

① Process 1

```
0x00: load r1, &a
0x04: inc r1
0x08: store r1, &a
0x0C: load r1, &a
0x10: cmp r1, 1
0x14: JE 0x20
0x18: add r1, 2
0x1C: store r1, &a
0x20: ……
```

CPU

PC=0x08

② Timer event occurs

③ **Save** PC=0x08

Context Switch Workflow

① Process 1

```
0x00: load r1, &a
0x04: inc r1
0x08: store r1, &a
0x0C: load r1, &a
0x10: cmp r1, 1
0x14: JE 0x20
0x18: add r1, 2
0x1C: store r1, &a
0x20: ……
```

CPU

PC=0x08

② Timer event occurs

Process 2

```
0x50: ……
0x54: ……
0x58: ……
0x5C: ……
0x60: ……
0x64: ……
0x68: ……
0x6C: ……
0x60: ……
```

③ **Save PC=0x08**

④ Timer event handler

30

Context Switch Workflow

① Process 1

```
0x00: load r1, &a
0x04: inc r1
0x08: store r1, &a
0x0C: load r1, &a
0x10: cmp r1, 1
0x14: JE 0x20
0x18: add r1, 2
0x1C: store r1, &a
0x20: ……
```

② **Save** PC=0x08

CPU

PC=0x08

③ Timer event occurs
④ Jump to timer-event handler
⑥ OS resumes Process 1 **or** may switch to Process 2

Process 2

```
0x50: ……
0x54: ……
0x58: ……
0x5C: ……
0x60: ……
0x64: ……
0x68: ……
0x6C: ……
0x60: ……
```

⑤ Timer event handler

Context Switch (more)

- Context switches can occur when
  - A program makes a system call
  - A timer event is triggered
  - ……
- Context switches have additional overhead
  - OS needs to save the state of current process and load the state of another process
  - Frequent context switches can be bad for performance

Context Switching

- **Context switch is expensive** (1–1000 μsec)
  - No useful work is done (pure overhead)
  - Can become a bottleneck
  - **Real-life analogy?**

Summary: Process (part 2)

- Process Scheduling
- Context Switching
- Inter-Process Communication