Operating System:
Processes

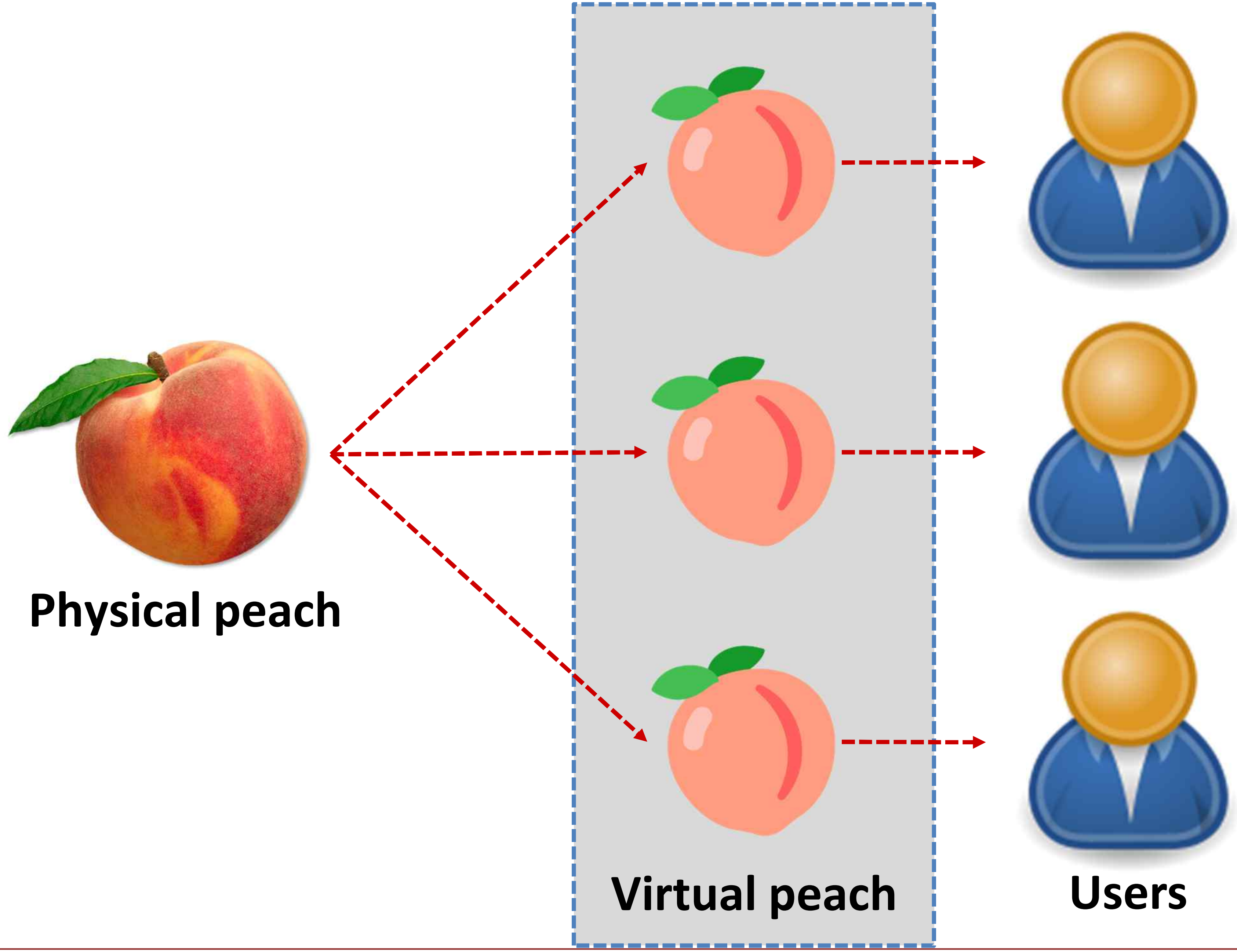
Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

Virtualization

A Dialogue on Virtualization

- Professor: And thus we reach the first of our three pieces on operating systems: virtualization
- Student: But what is virtualization, oh noble professor?
- Professor: Imagine we have a peach

A Dialogue on Virtualization (Cont'd)



A Dialogue on Virtualization (Cont'd)

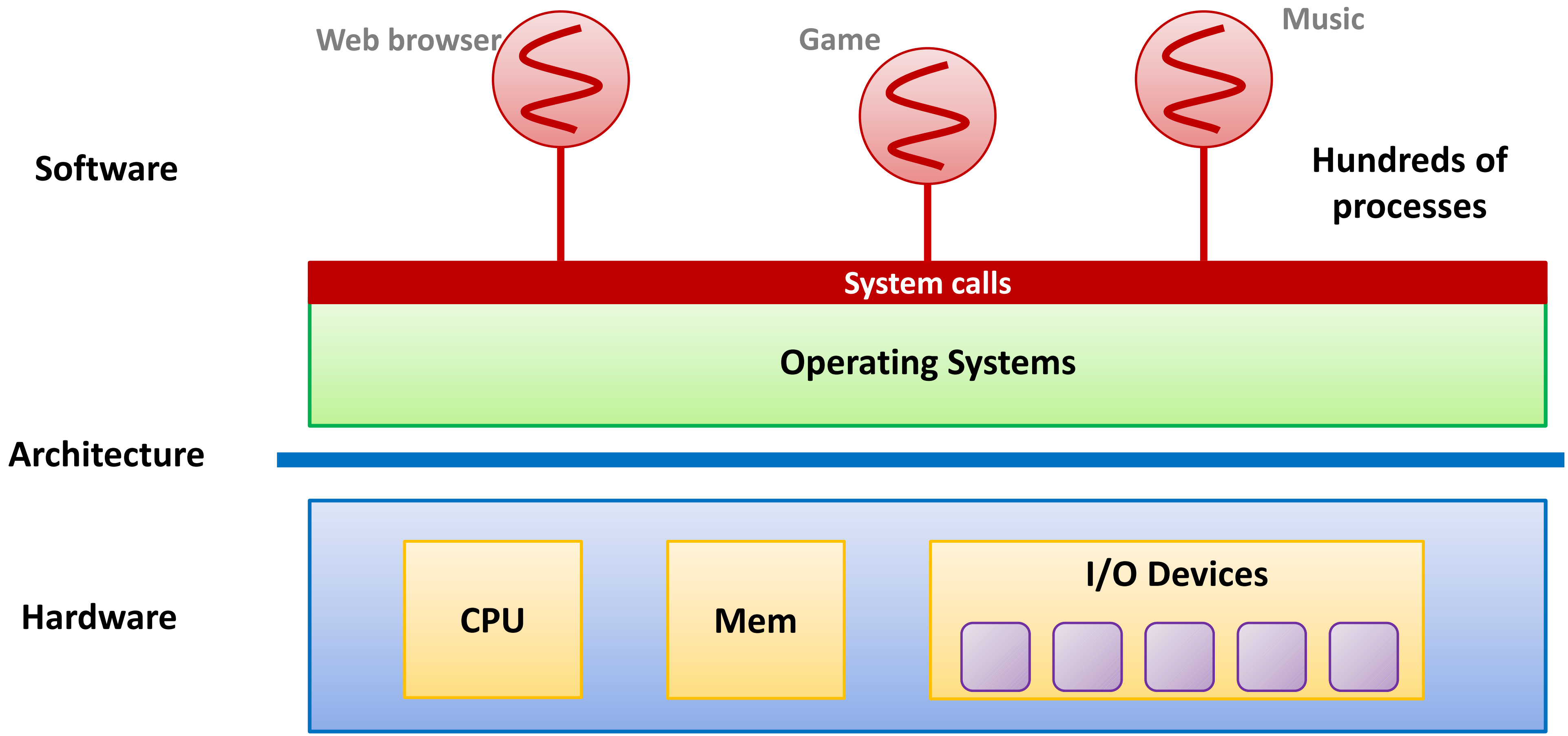
- Student: Well, if I was sharing a peach with somebody else, I think I would notice

- Professor: Ah yes! Good point. But that is the thing with many eaters; most of the time they are napping or doing something else, and thus, you can snatch that peach away and give it to someone else for a while. And thus we create the illusion of many virtual peaches, one peach for each person!

CPU Virtualization

가상화를 하기 위한 요구 : context switching
기능 : scheduling

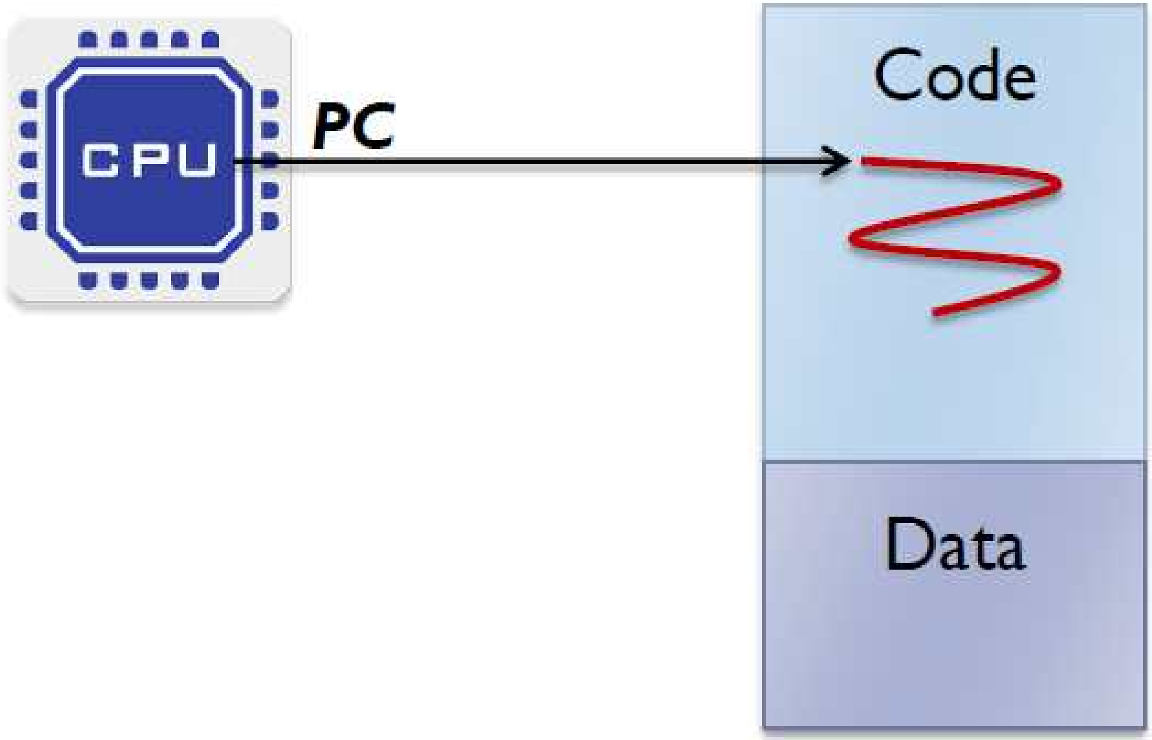
The Crux of the Problem:
How to Provide the Illusion of Many CPUs?



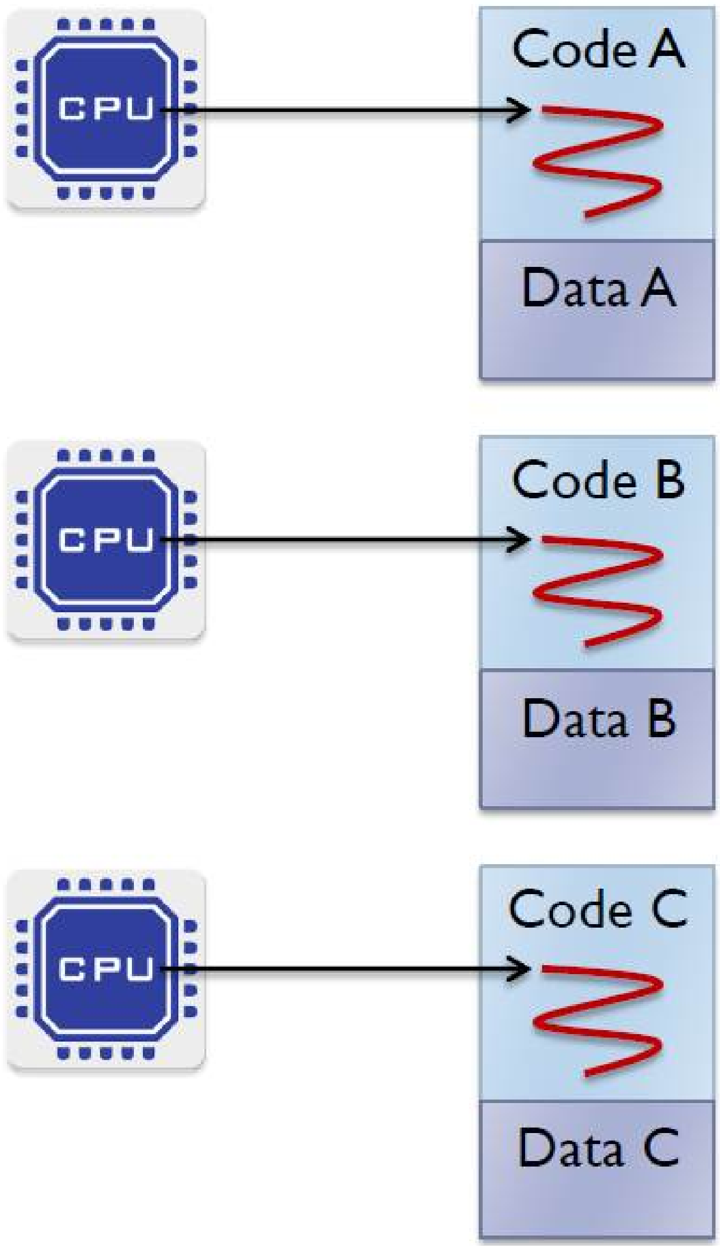
Running a Process

instruction program counter

Fetch ***I*** ← Mem[***PC***]
Decode ***I***
Execute ***I***
Update ***PC***

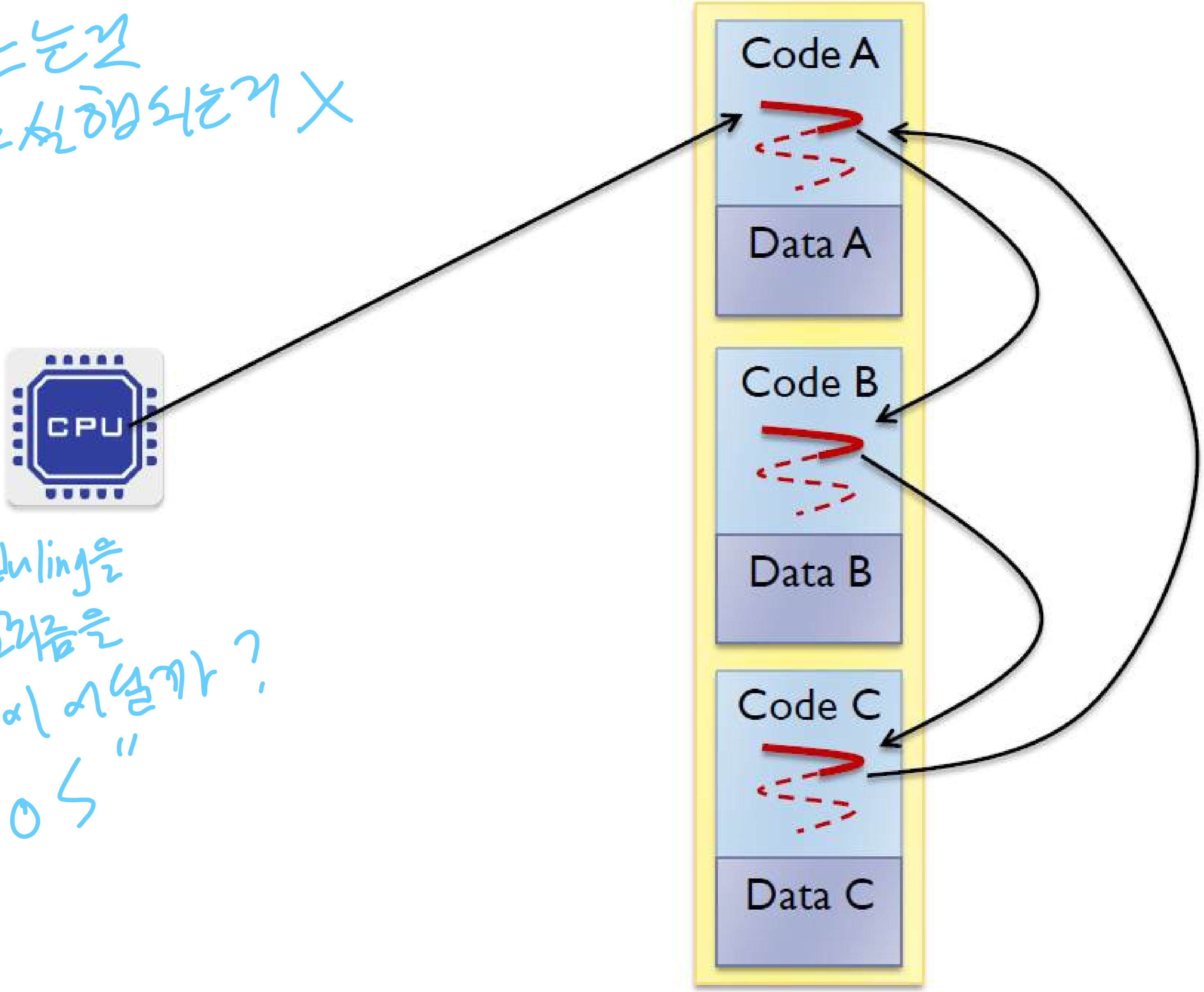


Running Multiple Processes



Interleaving Multiple Processes

process가 중간중간
끼어드는걸
하나가 종료후 실행되는거 X

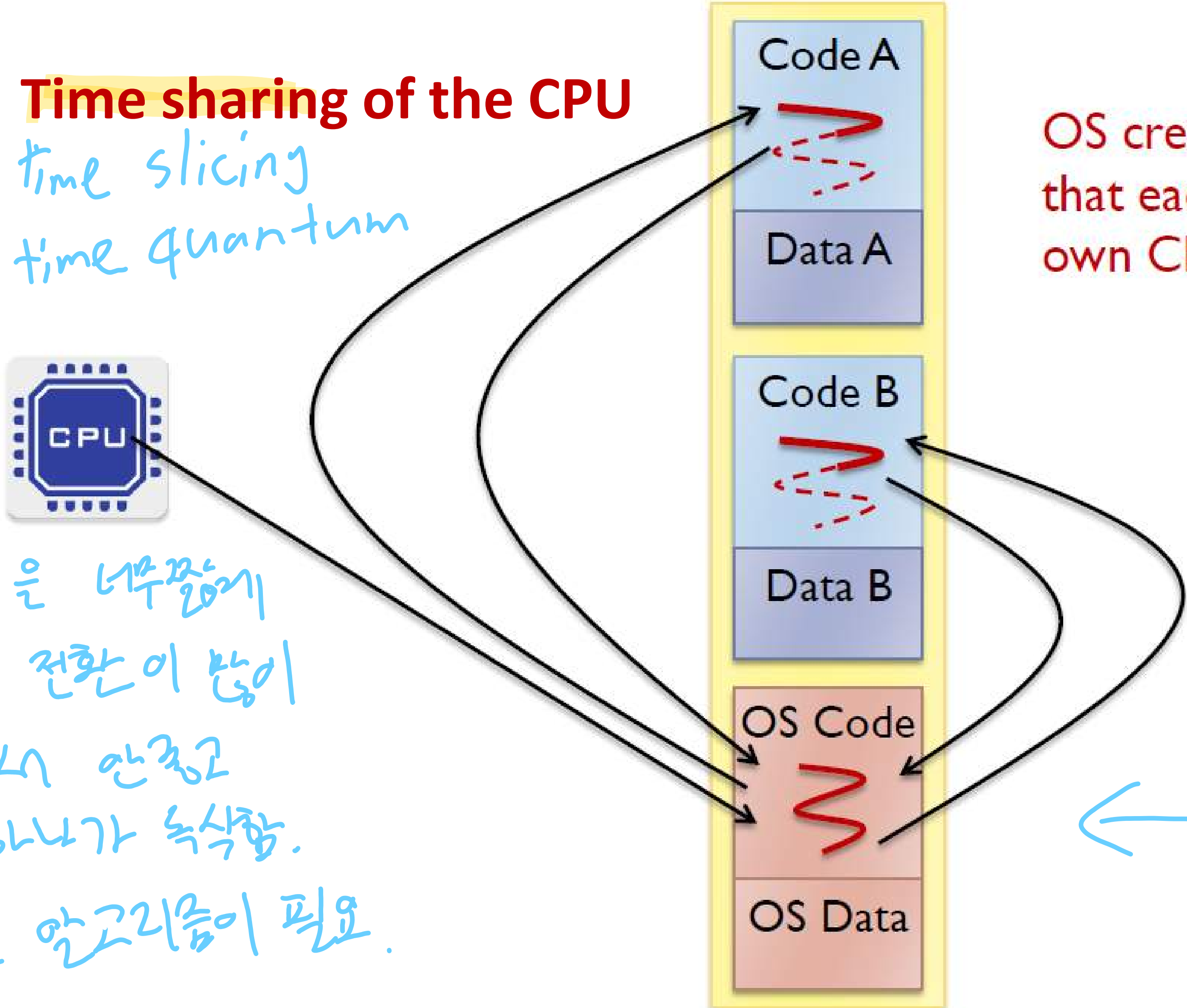


이러한 scheduling을
진행할 알고리즘을
가지는 곳이 어딜까?
"OS"

Virtualizing the CPU

Time sharing of the CPU

time slicing
time quantum



OS creates the illusion that each process has its own CPU (and memory)

사용시간을 너무 짧게 잡으면 전환이 많이 일어나서 안좋고 길면 하나가 독식함. 알맞는 알고리즘이 필요.

각각의 프로세스가 각각 CPU를 할당받은 것처럼 만들어줌.

A Process

A process is a running program

- Comprising of a process:
  - Memory (address space)
    - Code(Text)
    - Data
    - Stack
    - Heap
  - Registers
    - Program counter
    - Stack pointer

Process API

- These APIs are available on any modern OS
  - Create
    - ➢ Create a new process to run a program
  - Destroy
    - ➢ Halt a runaway process
  - Wait
    - ➢ Wait for a process to stop running
  - Miscellaneous Control 비정상적인 종료 .
    - ➢ Some kind of method to suspend a process and then resume it
  - Status running 인지 block 인지 등등 . . .
    - ➢ Get some status info about a process

Process Creation

1. **Load** a program code into memory, into the address space of the process
   - Programs initially reside on disk in *executable format*
   - OS perform the loading process lazily
     - ➢ Loading pieces of code or data only as they are needed during program execution

lazily의 의미는 코드의 일부분만 메모리에 가져오고 필요할때 마다 가져온다는것

2. The program's run-time **stack** is allocated

정적영역.

   - Use the stack for *local variables*, *function parameters*, and *return address*
   - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

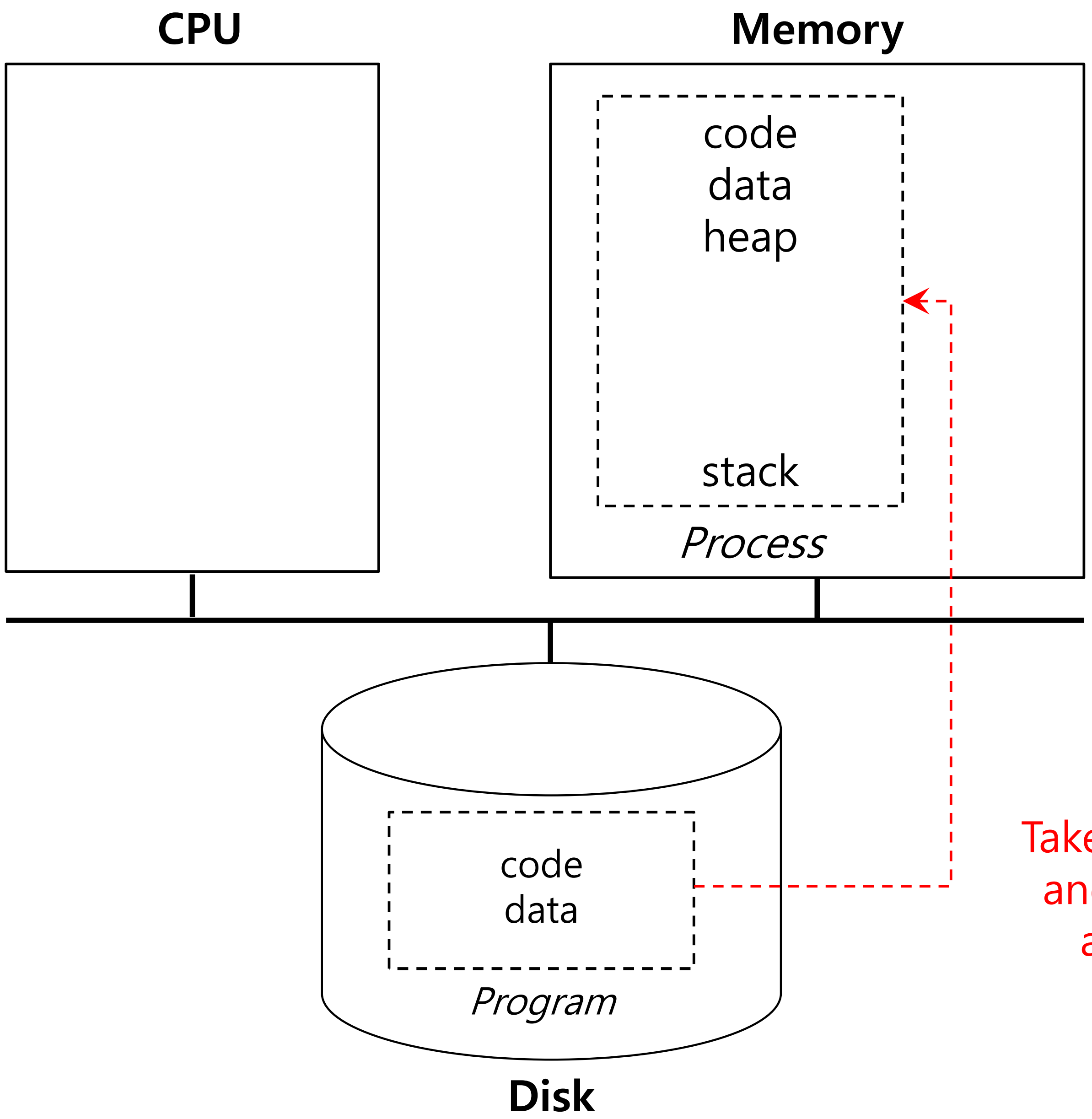
Process Creation (Cont.)

3. The program's **heap** is created 동적할당영역.
   - Used for explicitly requested dynamically allocated data
   - Program request such space by calling `malloc()` and free it by calling `free()`

4. The OS do some other initialization tasks
   - input/output (I/O) setup
     - ➢ Each process by default has three open file descriptors
     - ➢ Standard input, output and error

5. **Start the program** running at the entry point, namely `main()`
   - The OS *transfers control* of the CPU to the newly-created process

Loading: From Program To Process

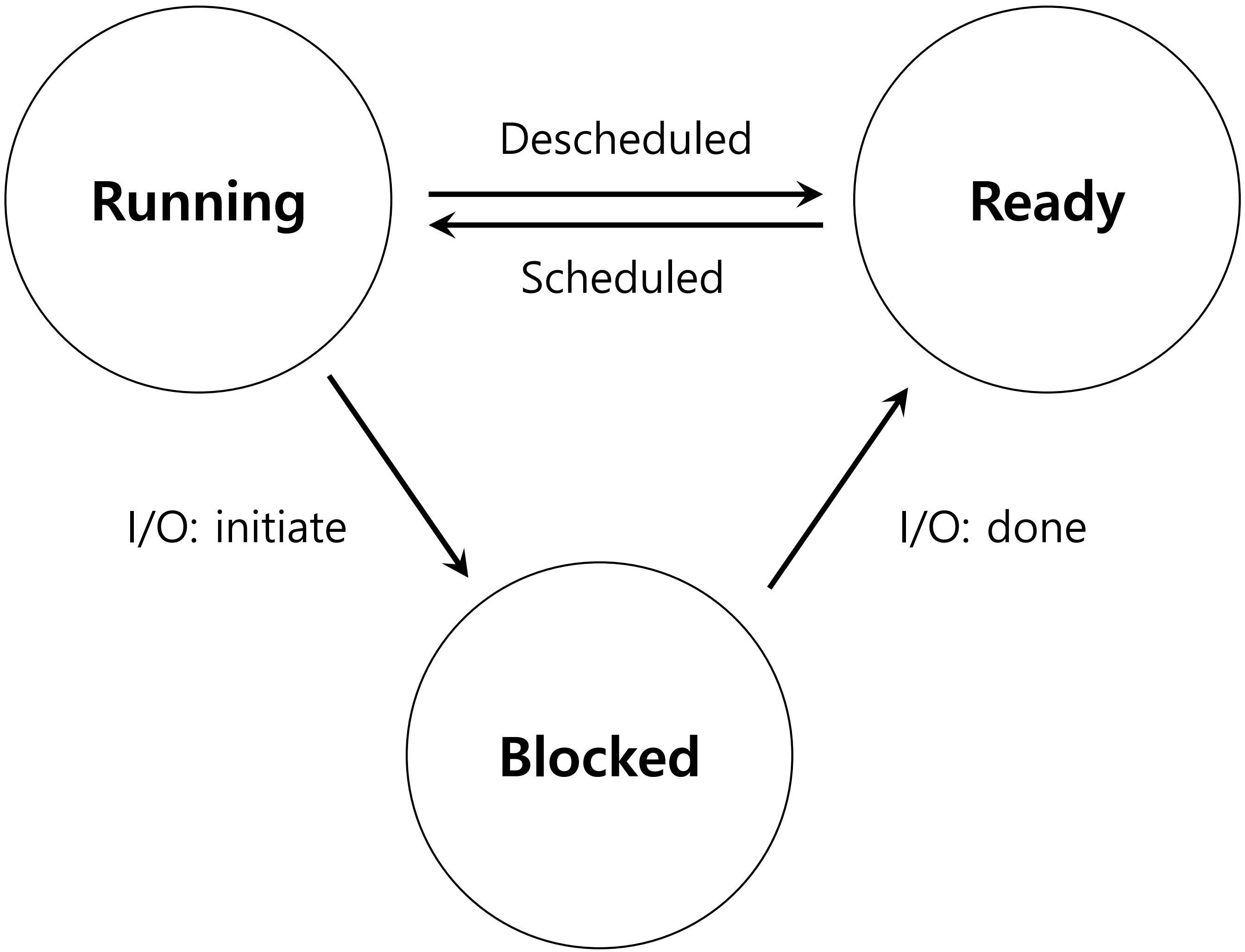


Loading:
Takes on-disk program and reads it into the address space of process

Process States

- A process can be one of three states
  - Running　　CPU를 할당받아서 실행중인 상태.
    - ➢ A process is running on a processor
  - Ready　　다른 프로세스가 사용중이라 대기중인 상태.
    - ➢ A process is ready to run but for some reason the OS has chosen not to run it at this given moment
  - Blocked　　특정한 일을 수행하기위한 상태
    - ➢ A process has performed some kind of operation
    - ➢ When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor　I/O 요청 시에는 CPU가 필요없기에.

Process State Transition



Tracing Process State: CPU Only

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Tracing Process State: CPU and I/O

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Data structures

- The OS has some key data structures that track various relevant pieces of information
  - Process list
    - Ready processes
    - Blocked processes
    - Current running process
  - Register context
- PCB (Process Control Block)
  - A C-structure that contains information about each process

Implementing Processes

- PCB (Process Control Block) or Process Descriptor
  - Contains all of the information about a process
    - ➢ CPU registers
    - ➢ PID, PPID, process group, priority, process state, signals
    - ➢ CPU scheduling information
    - ➢ Memory management information
    - ➢ Accounting information — 실제 해당프로세스가 얼마나 사용되었는지
    - ➢ File management information
    - ➢ I/O status information
    - ➢ Credentials
  - `task_struct` in Linux
    - ➢ 3248 bytes as of Linux 3.2.0

Example) The xv6 kernel Proc Structure

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;   // Index pointer register
    int esp;   // Stack pointer register
    int ebx;   // Called the base register
    int ecx;   // Called the counter register
    int edx;   // Called the data register
    int esi;   // Source index register
    int edi;   // Destination index register
    int ebp;   // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

Example) The xv6 kernel Proc Structure (Cont.)

```c
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                    // Start of process memory
    uint sz;                      // Size of process memory
    char *kstack;                 // Bottom of kernel stack
                                  // for this process
    enum proc_state state;        // Process state
    int pid;                      // Process ID
    struct proc *parent;          // Parent process
    void *chan;                   // If non-zero, sleeping on chan
    int killed;                   // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;            // Current directory
    struct context context;       // Switch here to run process
    struct trapframe *tf;         // Trap frame for the
                                  // current interrupt
};
```

Context Switch

- When CPU switches to another process, the system must
  - save the state of the old process, and
  - load the saved state for the new process

- Context switch time is pure overhead
  - System does not any useful work while switching

전환하는 시간이 오버헤드이니 time slicing이 너무 짧아지면 X.

- Context switch time depends on hardware
  - The register set is different

Context Switch (Cont.)

- CPU switch from process to process