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

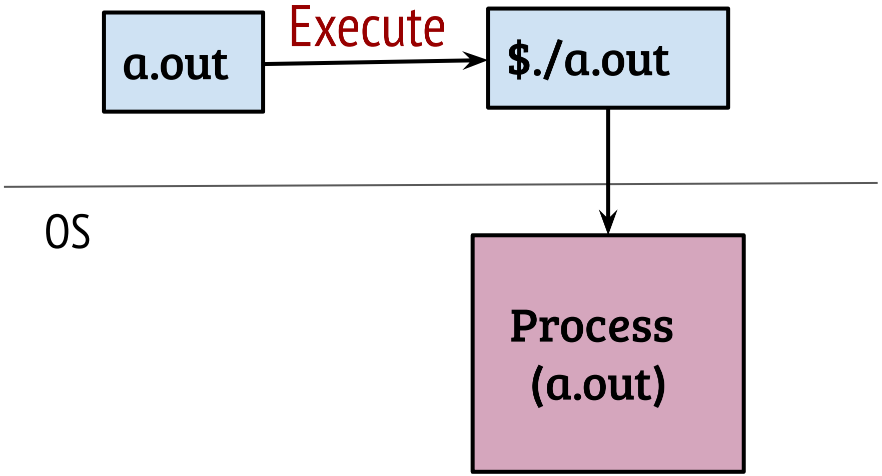
Agenda: CPU $\longrightarrow$ Process (OSTEP Ch4)

The process abstraction

- The OS creates a process when we run an executable

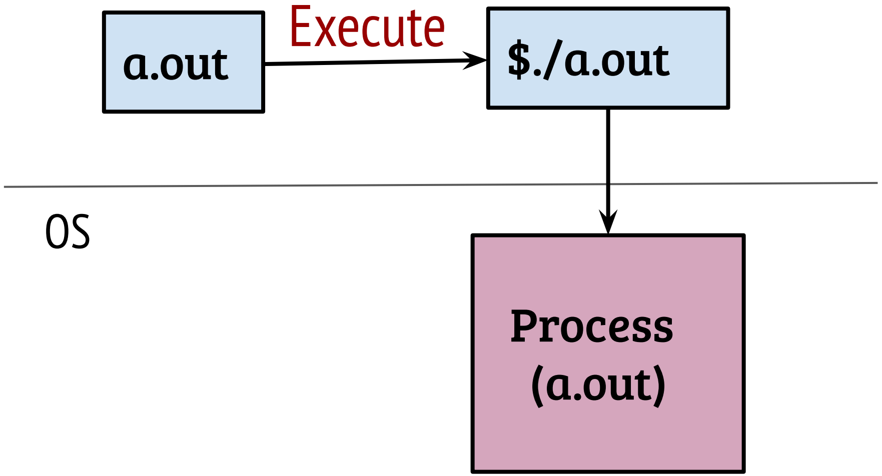
The process abstraction

- 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 $\longrightarrow$ Process (1 to N)
  - Many concurrent processes can execute the same program

What about virtualizing the CPU?

Virtualization of the CPU

MPlayer

$./a.out

Browser

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

Process (Mplayer)

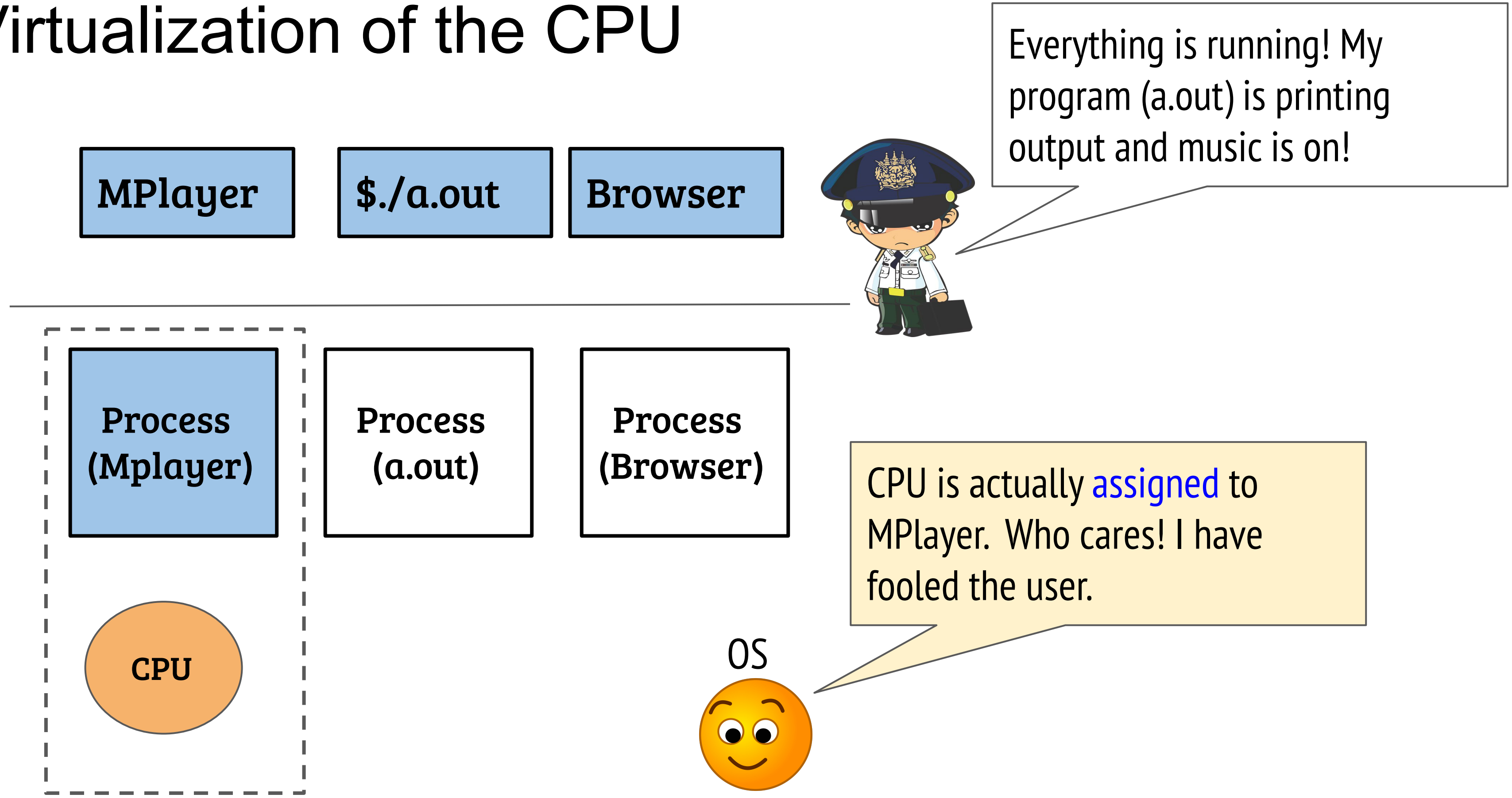
Process (a.out)

Process (Browser)

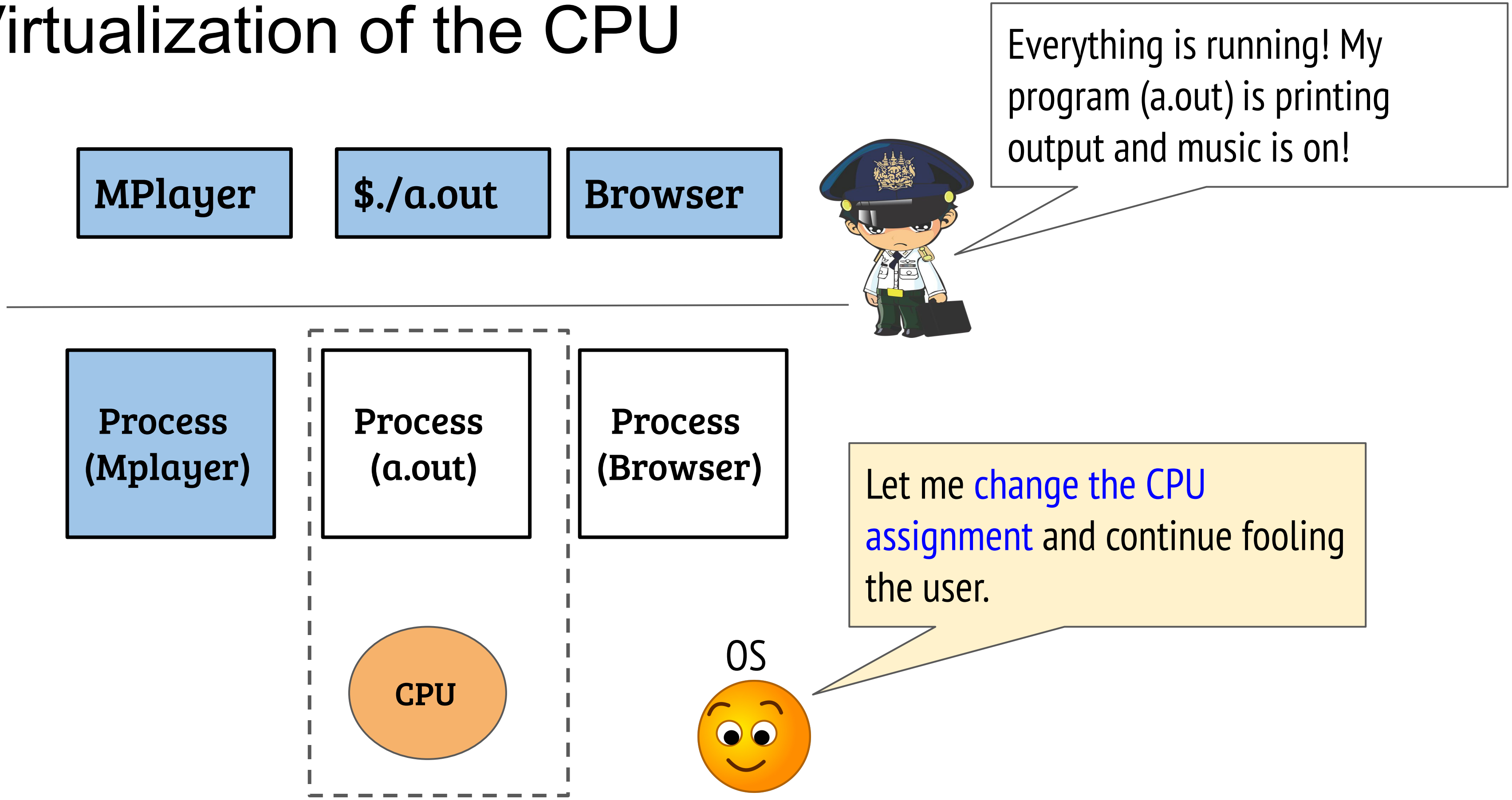
OS

CPU

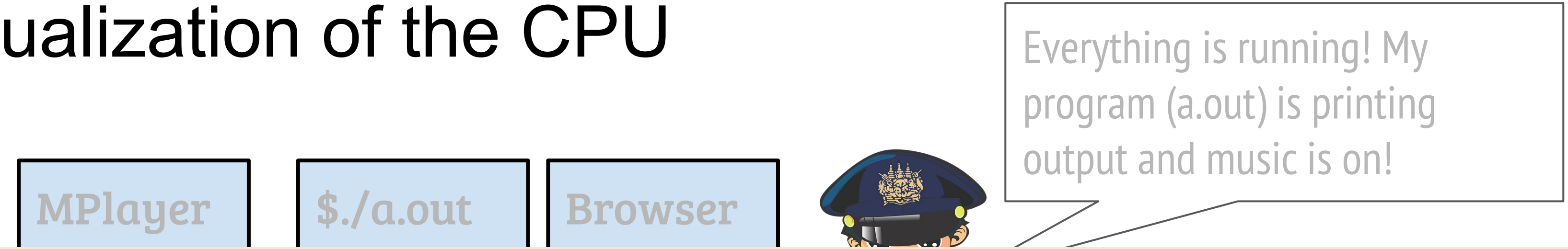
Virtualization of the CPU



Virtualization of the CPU

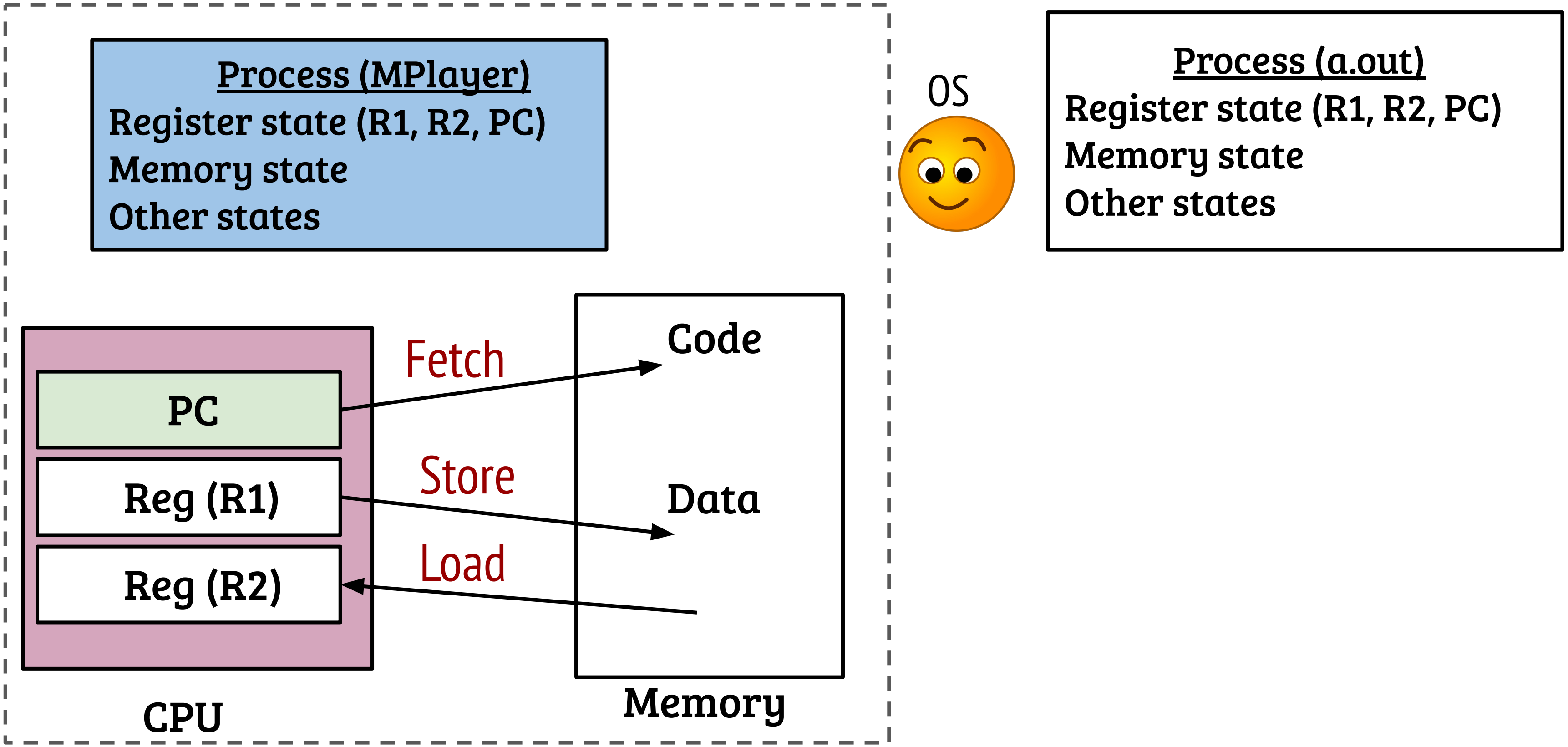


Virtualization of the CPU

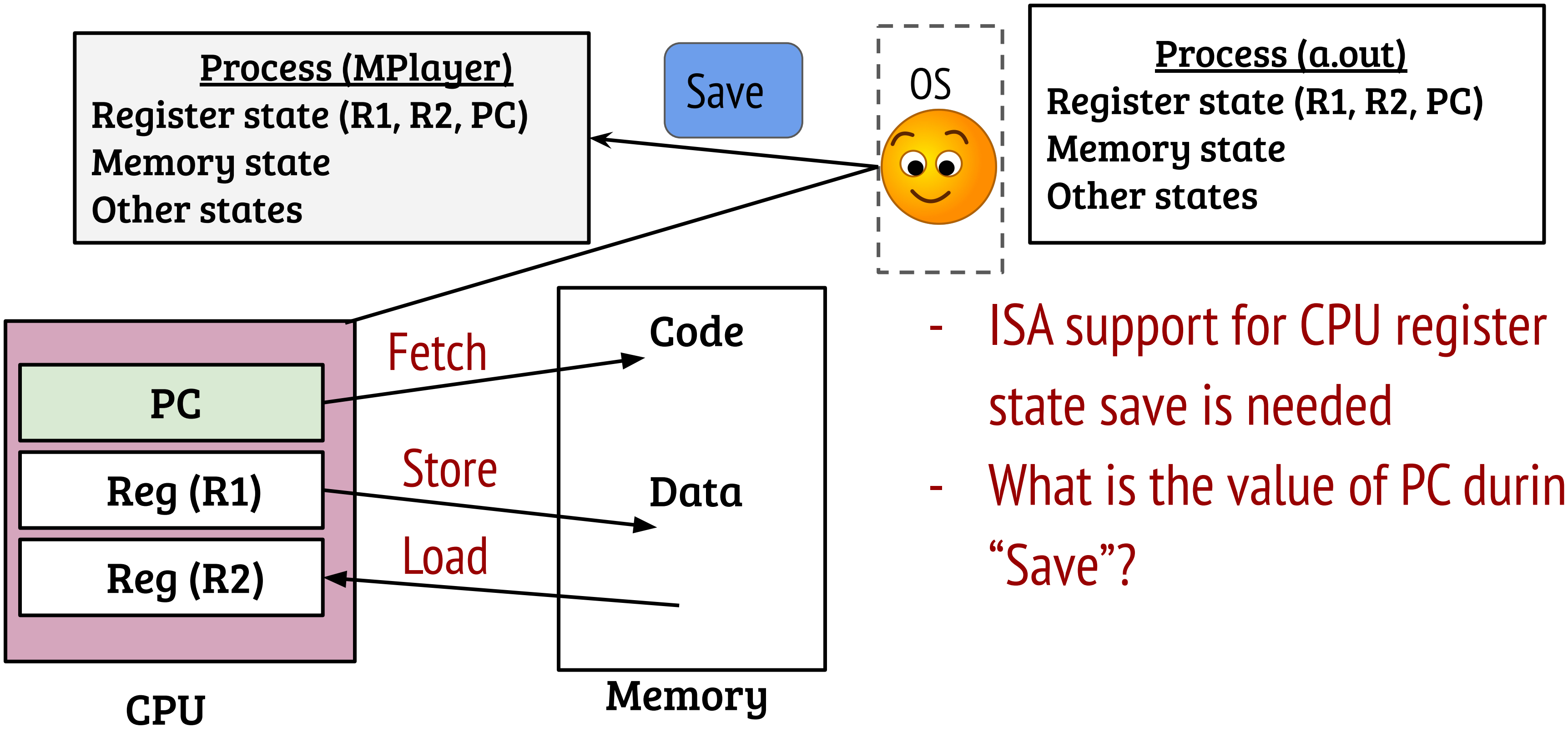


- How CPU assignment is changed? (OR how context switch is performed?)
  - What happens to outgoing process? How does it come back?
- Overheads of context switch?
- How to decide the incoming process?

Context switch: state of a process

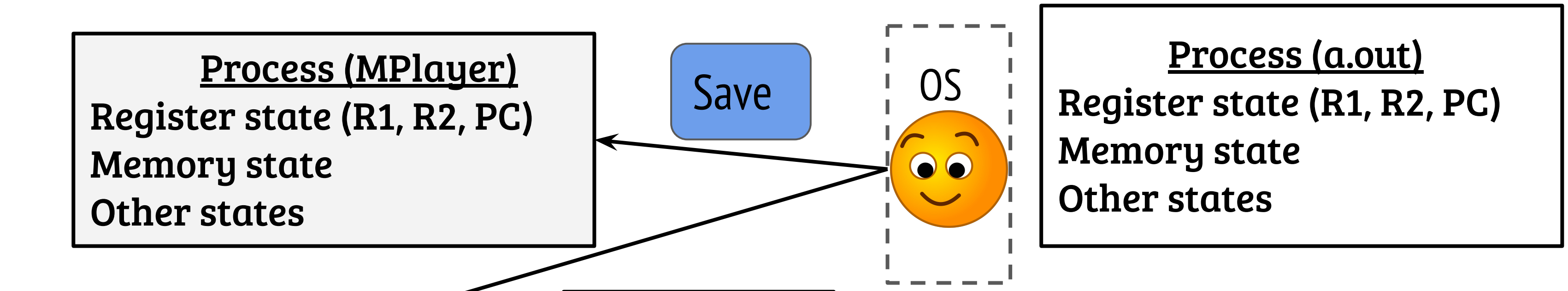


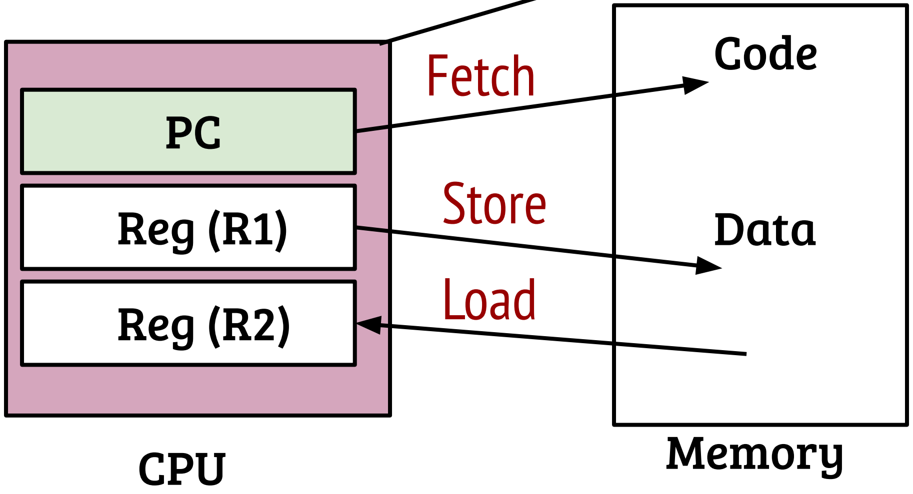
Context switch: saving the state of outgoing process



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

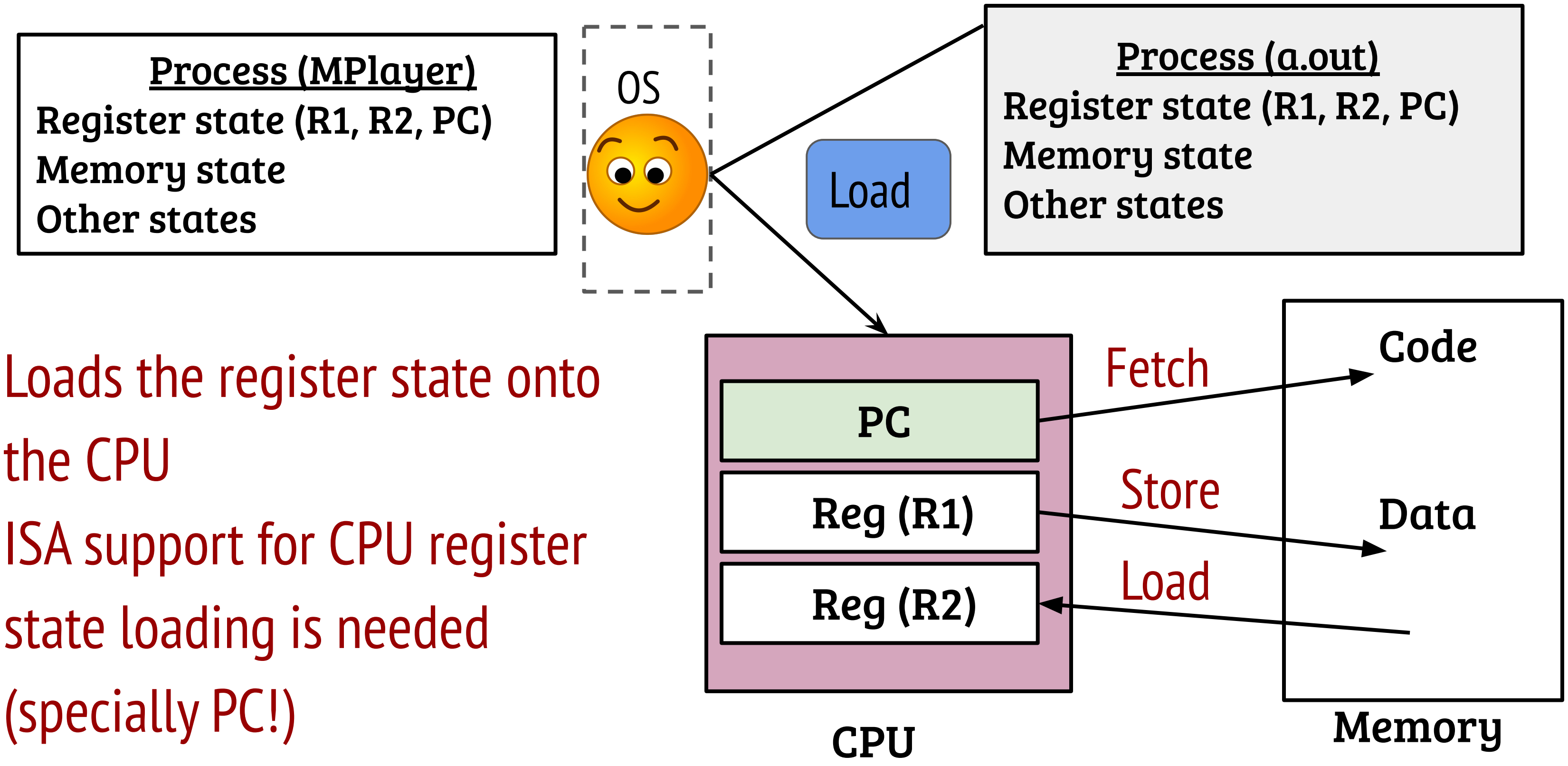
Context switch: saving the state of outgoing process





- 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

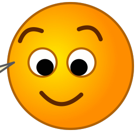


- Loads the register state onto the CPU
- ISA support for CPU register state loading is needed (specially PC!)

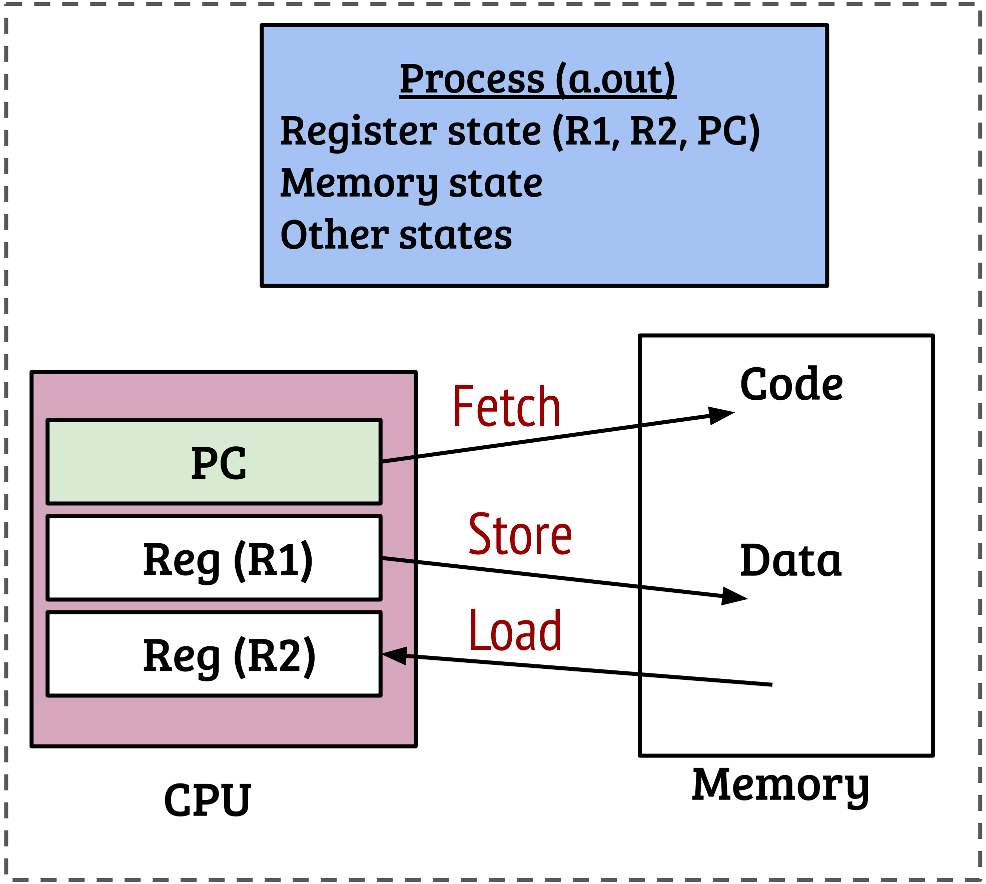
Context switch: load the state of incoming process

Process (MPlayer)
Register state (R1, R2, PC)
Memory state
Other states

OS



Done with context switch. Let me relax till the next context switch!



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