User program and OS interaction Multiprocessing

Jinyang Li & Shuai Mu

What we've learnt so far

- Machine instructions
  - compiler translates C to x86 instructions
  - x86 instructions are executed by CPU hardware only
- Dynamic memory allocator
  - realized as a library implementation
- Virtual memory
  - each process has its own virtual address space
  - VM is realized by a combination of hardware mechanism and OS implementation
    - MMU performs address translation
    - OS populates page table

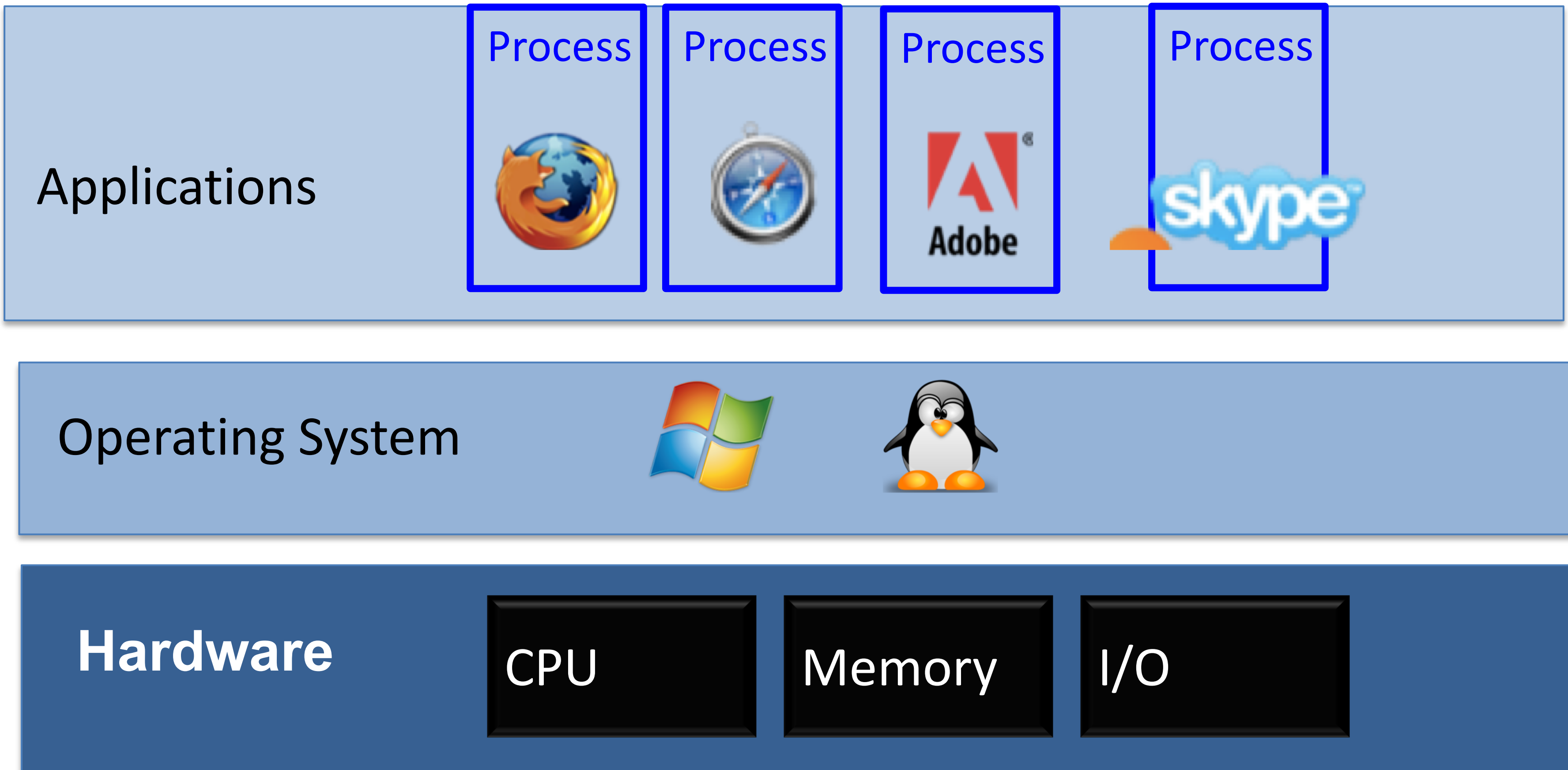
Today's lesson plan

1. Interaction between user programs and OS
2. Multiprocessing

Interaction between user programs and OS

I mean OS kernel

Applications, OS, Hardware



The role of OS

Purpose of the OS software

1. Manage resources among running programs
2. Hide messy hardware details

Concrete jobs of the OS

1.1 Scheduling (give each process the illusion of exclusive CPU use)
1.2 VM management (give each process the illusion of exclusive memory use)
2. file systems, networking, I/O

Process

- Process is an instance of a running program
  - when you type `./a.out`, a process is launched
  - when you type Ctrl-C, the process is killed
- Each process corresponds to some state in OS
  - process identifier (process id)
  - user id
  - status (e.g. runnable or blocked)
  - saved rip and other registers
  - VM structure (including its page table)

Only OS can modify these data

How to protect the OS from user processes?

- Hardware provides privileged vs. non-privileged mode of execution

  also called supervisor/kernel mode

  also called user mode

- OS runs in privileged mode
  - can change content of CR3 (points to root page table)
  - can access VA marked as supervisor only
  - …
- User programs run in non-privileged mode
  - cannot access kernel data structures because they are stored in VA marked as supervisor only

How to get into privileged mode?

Hardware provides 3 controlled mechanisms to switch from non-privileged to privileged execution:

1. Traps
   – syscalls (user programs explicitly ask for OS help)
2. Exception (caused by the current running program)
   – e.g. divide by zero, page fault
3. Interrupt (caused by external events)
   – timer, device events e.g. keyboard press, packet arrival

How to get out of privileged mode?

- OS uses the special hardware instruction `iret`
- OS may return to the same program or context switch to execute a different program

#1 Traps:
Syscall: User → OS

- User programs ask for OS services using syscalls
  - it's like invoking a function in OS
- Each syscall has a known number

0	**read**
1	write
2	open
3	close
...	
57	fork
59	execve
60	exit
62	kill

C library wraps these syscalls to provide file I/O

linux syscall number

Syscall: user → OS

user code

```
movq %rax, %rbx
inc %rbx
…
syscall 2
movq %rax, %r8
add %r8, %r9
…
```

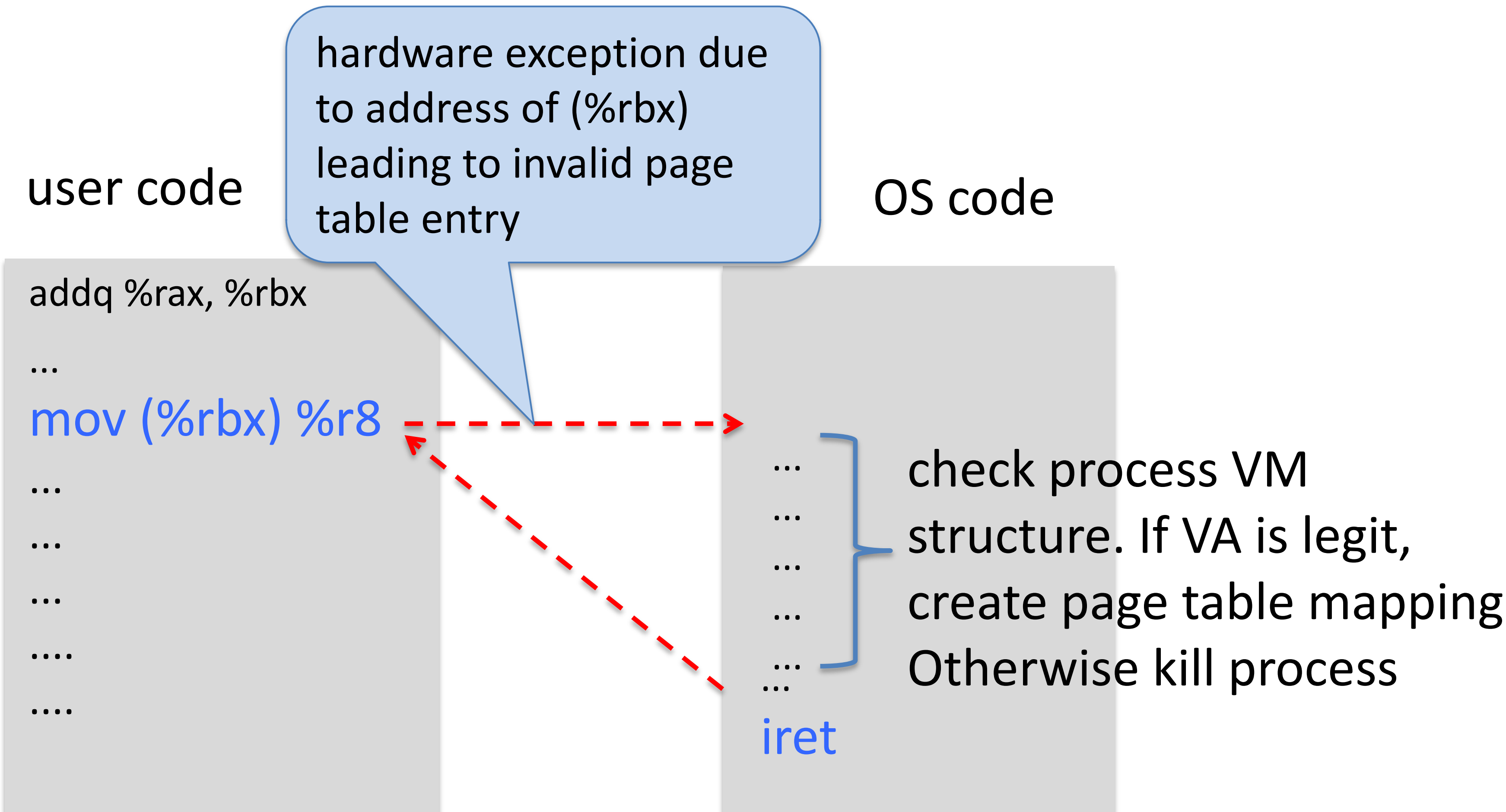
OS code

```
…
…
…
…
…
…
…
…
…
iret
```

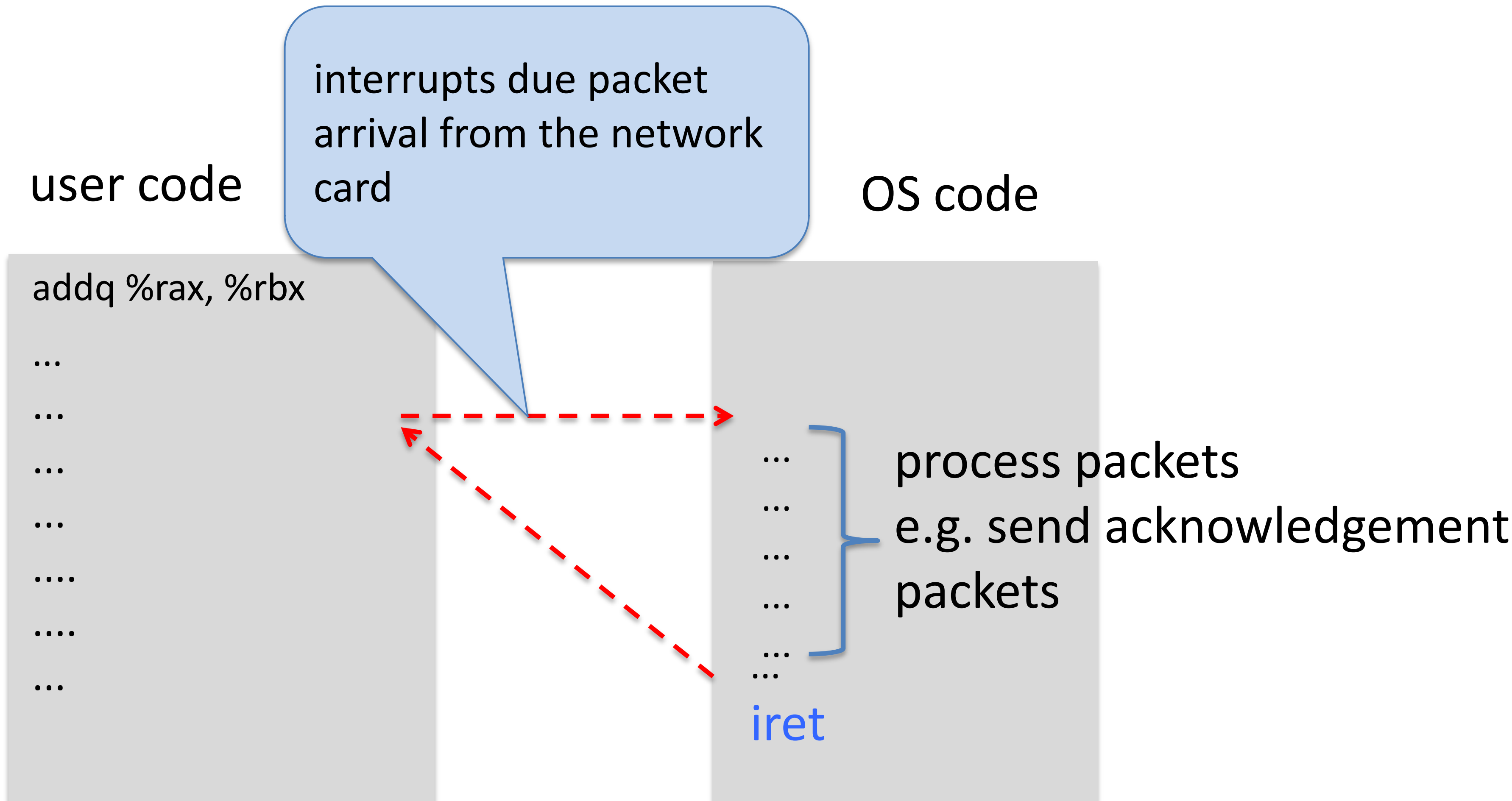
code to open the requested file

Assuming OS wants to execute the same process next; it does not have to

#2 exceptions:
OS takes control upon exceptions



#3 interrupts:
OS takes control upon interrupts

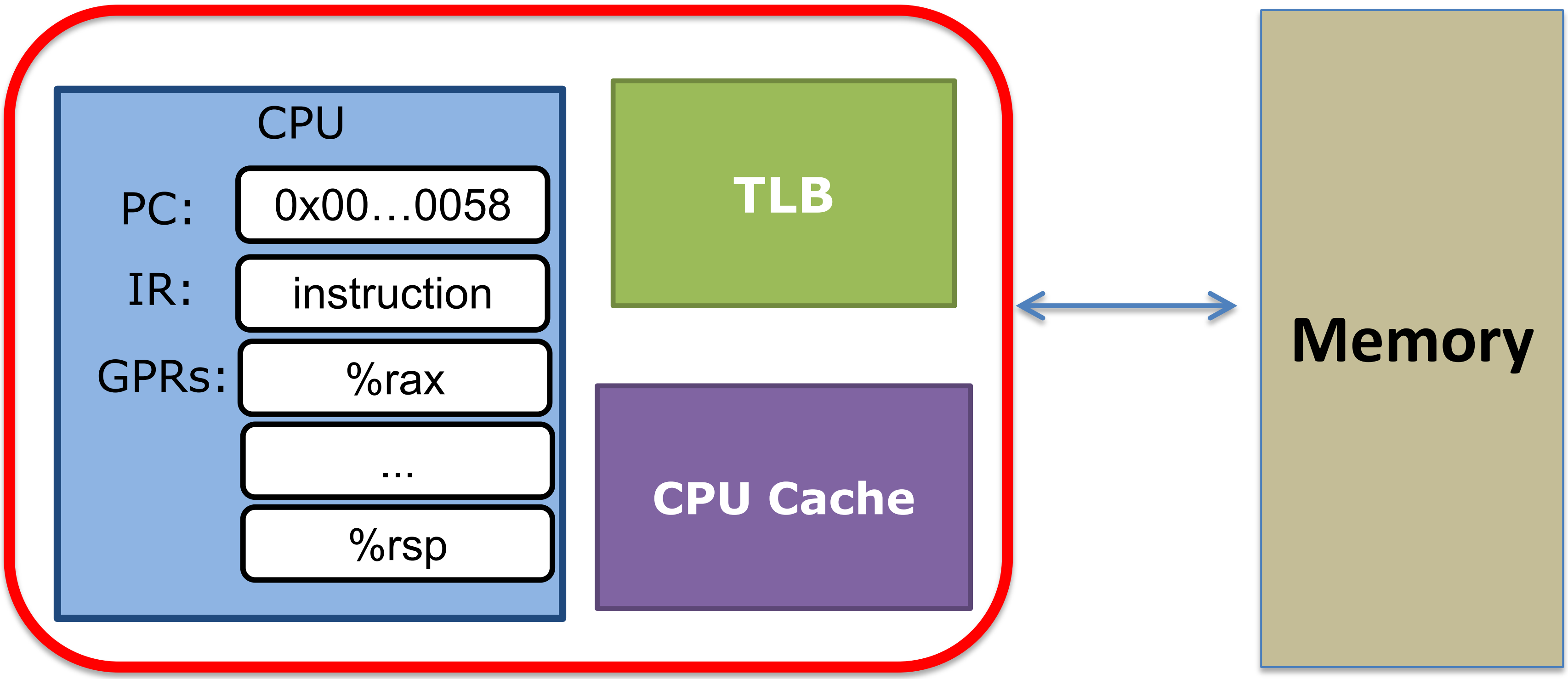


Multi-processing

Goal of multi-processing

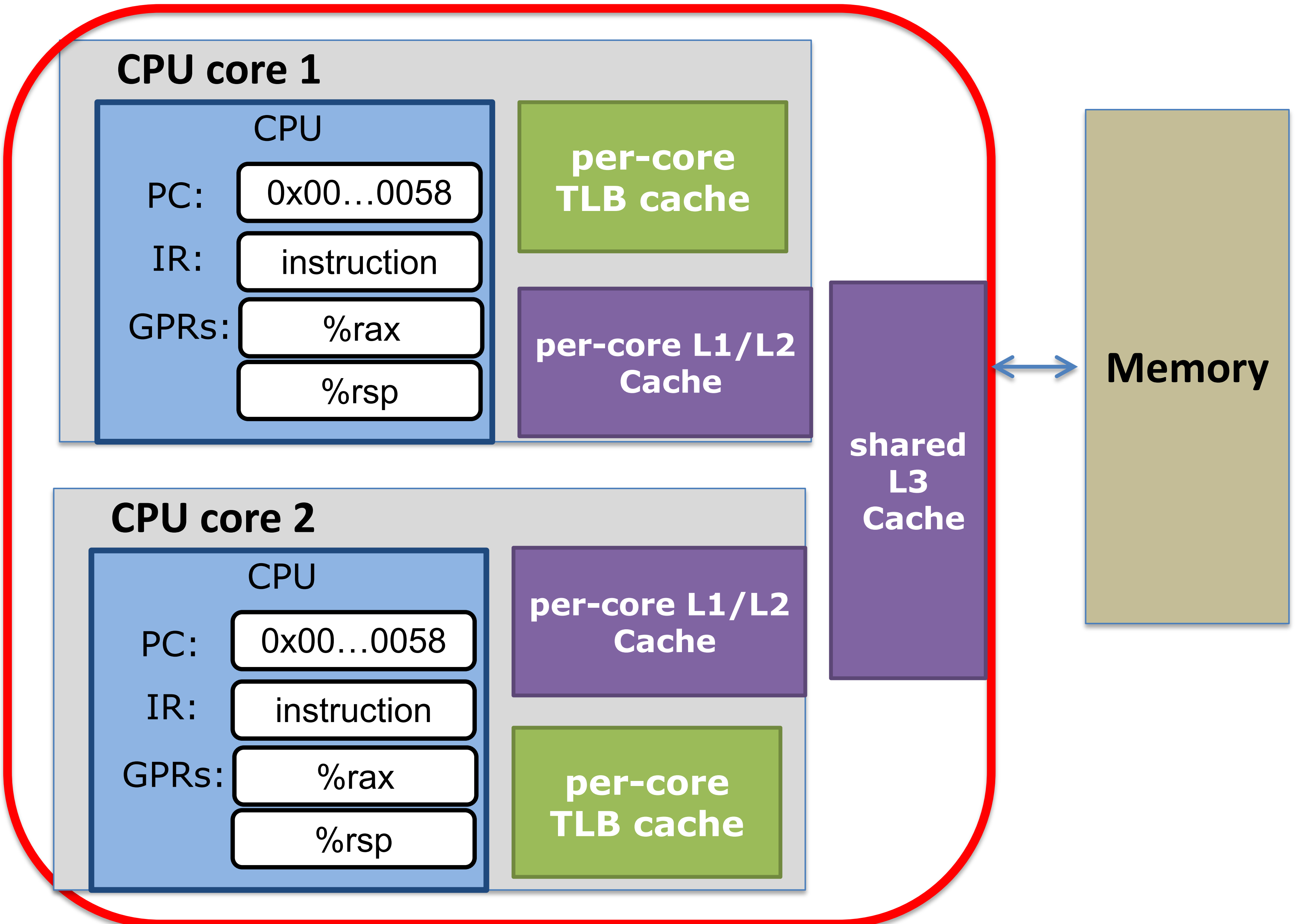
- Run multiple processes “simultaneously”
- Why?
  - listening to music while writing your lab
  - Running a web server, a database server, a PHP program together

Modern CPUs have multiple cores



Your mental model of the CPU as a single core machine

Modern CPUs have multiple cores

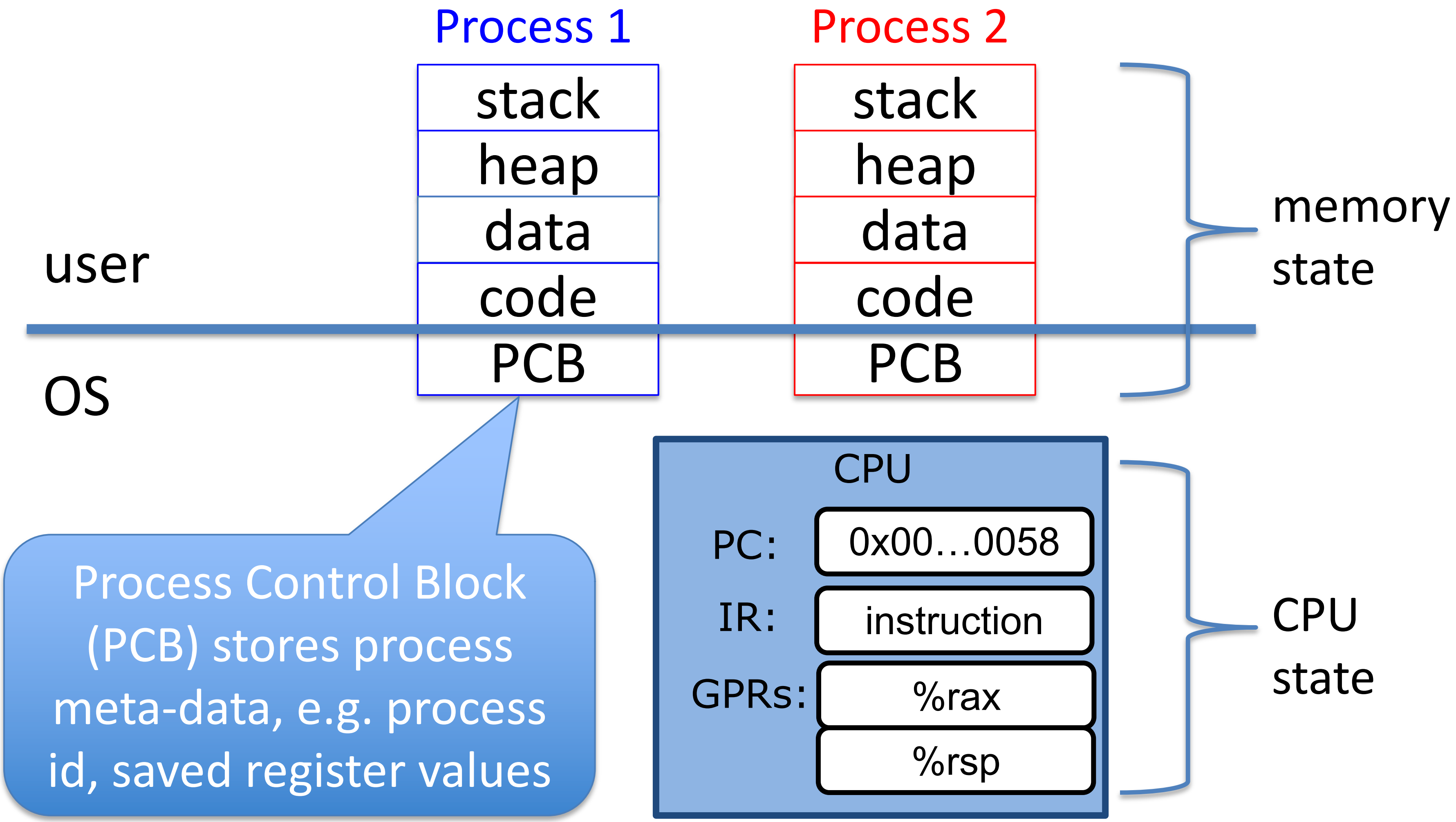


How to multi-process?

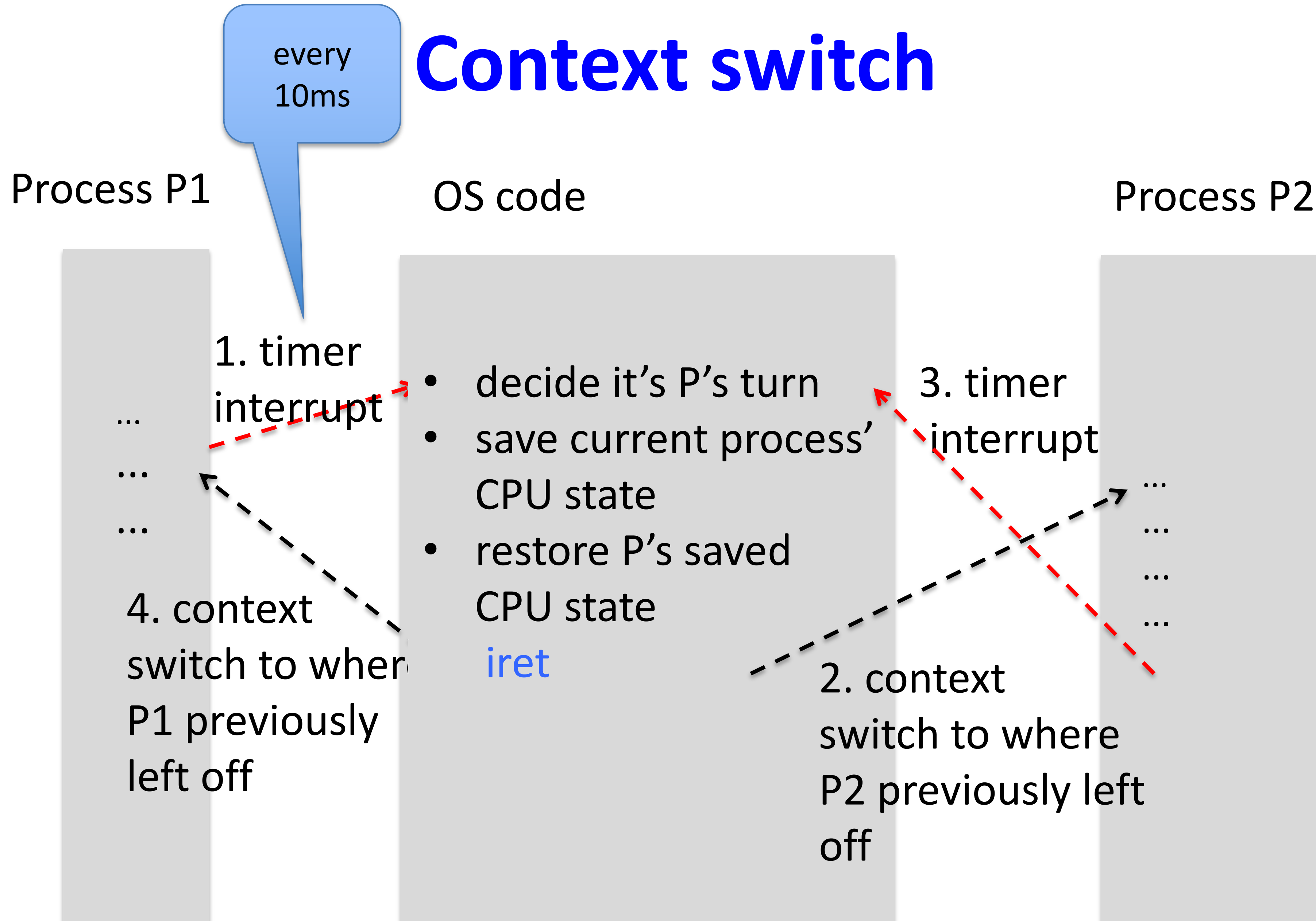
- Execute one process exclusive on each core?
  - 2 cores → 2 processes only 

- How to “simultaneously” execute more processes than there are cores?

Multiprocessing (e.g. on a single core machine)



Context switch



Creating and killing processes

- One process creates another process via syscall `fork()`
  - All processes are created by some processes (a tree).
  - The first process is a special one (`init`) and is created by OS.
  - When launching a program via command-line, the shell program creates the process

The fork syscall

- OS creates a new child process (almost completely) identical to the parent process
- Same code, data, heap, stack, register state except different return values of the fork syscall
- Returns child process’s id in parent process
- Returns zero in the child process

“called once, returned twice”

Example fork call

```
void main()
{
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent, child pid=%d\n", pid);
  }
}
```

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...\n");
  }
}
```

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...\n");
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...\n");
  }
}
```

output:

```
In parent...
```

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

output:

In parent...

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf(“In child”);
  } else {
    printf(“In parent...\n”);
  }
}
```

output:

In parent...

Example fork call

process 1

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...\n");
  }
}
```

process 2

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("In child");
  } else {
    printf("In parent...\n");
  }
}
```

output:

```
In parent...
In child
```

Notes on fork

- Execution of parent and child are concurrent
  - interleaving is non-deterministic.
  - In the example, both outputs are possible

```
In parent...
In child
```

```
In child
In parent...
```

- Parent and child have separate address space (but their contents immediately after fork are identical)

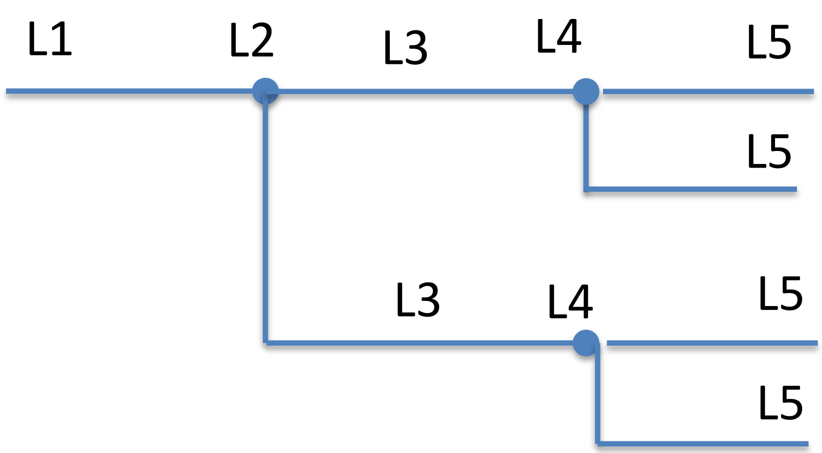
Another fork example

```
    void main()
    {
1:      printf("hello\n");
2:      fork();
3:      printf("world\n");
4:      fork();
5:      printf("bye\n");
    }
```

How many processes are created in total?

Another fork example

```
void main()
{
    L1: printf(“hello\n”);
    L2: fork();
    L3: printf(“world\n”);
    L4: fork();
    L5: printf(“bye\n”);
}
```

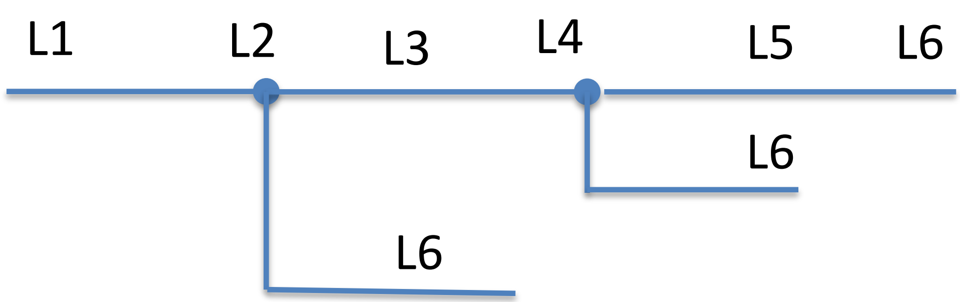


What are the possible printouts?

hello	hello	hello
world	world	world
world	bye	world
bye	bye	world
bye	world	bye
bye	bye	bye
bye	bye	bye

Exercise

```
void main()
{
   L1: printf("hello\n");
   L2: if (fork() == 0) {
   L3:   printf("big\n");
   L4:   if (fork() == 0) {
   L5:     printf("world\n");
          }
        }
   L6:  printf("bye\m");
}
```



What are the possible printouts?

hello	hello	hello
big	bye	bye
world	big	big
bye	bye	bye
bye	world	bye
bye	bye	world

Parent and child have separate address space with (initially) idential content

```
void main()
{
    int total = 0;
    pid_t pid = fork();
    assert(pid >= 0);
    total++;
    if (pid == 0)
        printf("child %d\n", total);
    else
        printf("parent %d\n", total);
}
```

What are the possible printouts?

child 1
parent 1

child 1
parent 2

parent 1
child 2

parent

total=0

Parent and child have separate address space with (initially) idential content

```
void main()
{
    int total = 0;
    pid_t pid = fork();
    assert(pid >= 0);
    total++;
    if (pid == 0)
        printf("child %d\n");
    else
        printf("parent %d\n");
}
```

What are the possible printouts?

child 1 parent 1	child 1 parent 2	parent 1 child 2

parent

total=0

child

total=0

Parent and child have separate address space with (initially) idential content

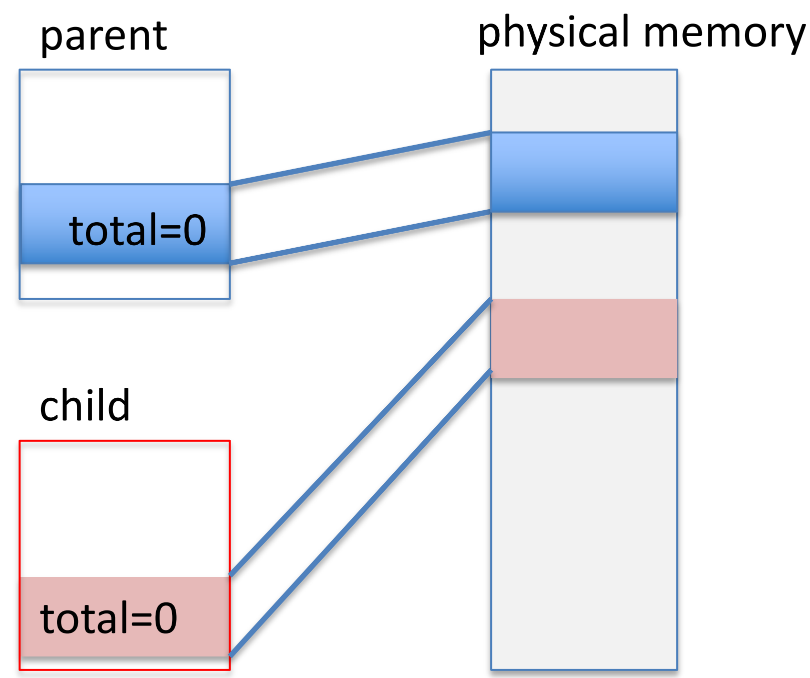
```
void main()
{
    int total = 0;
    pid_t pid = fork();
    assert(pid >= 0);
    total++;
    if (pid == 0)
        printf(“child %d\n”);
    else
        printf(“parent %d\n”);
}
```

What are the possible printouts?

child 1
parent 1

child 1
parent 2

parent 1
child 2



wait: synchronize with child

- Parent process could wait for the exit of its child process(es).
  - int waitpid(pid_t pid, int * child_status, ...)
- Good practice for parent to wait
  - Otherwise, some OS process state about the child cannot be freed even after child exits
  - leaks memory

Exercise

What are the possible printouts?

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("child");
  } else {
    printf("parent");
  }
}
```

child	parent
parent	child

Exercise

```
void
main() {
  pid_t pid = fork();
  assert(pid >= 0);
  if (pid == 0) {
    printf("child");
  } else {
    waitpid(pid, NULL, 0);
    printf("parent");
  }
}
```

What are the possible printouts?

child parent	~~parent child~~

execv: load program in current process

- int execv(char *filename, char *argv[])
  - overwrites code, data, heap, stack of existing process (retains process pid)
- called once, never returns

Example

```
void main() {
   pid_t pid;
   pid = fork();
   if (pid == 0) {
      execv("/bin/echo", "hello");
      printf("world\n");
   }
   waitpid(pid, NULL, 0);
   printf("bye\n");
}
```

Never executed because execv has replaced process's memory with that of the echo program

How many processes are created in total? output?

2

hello bye