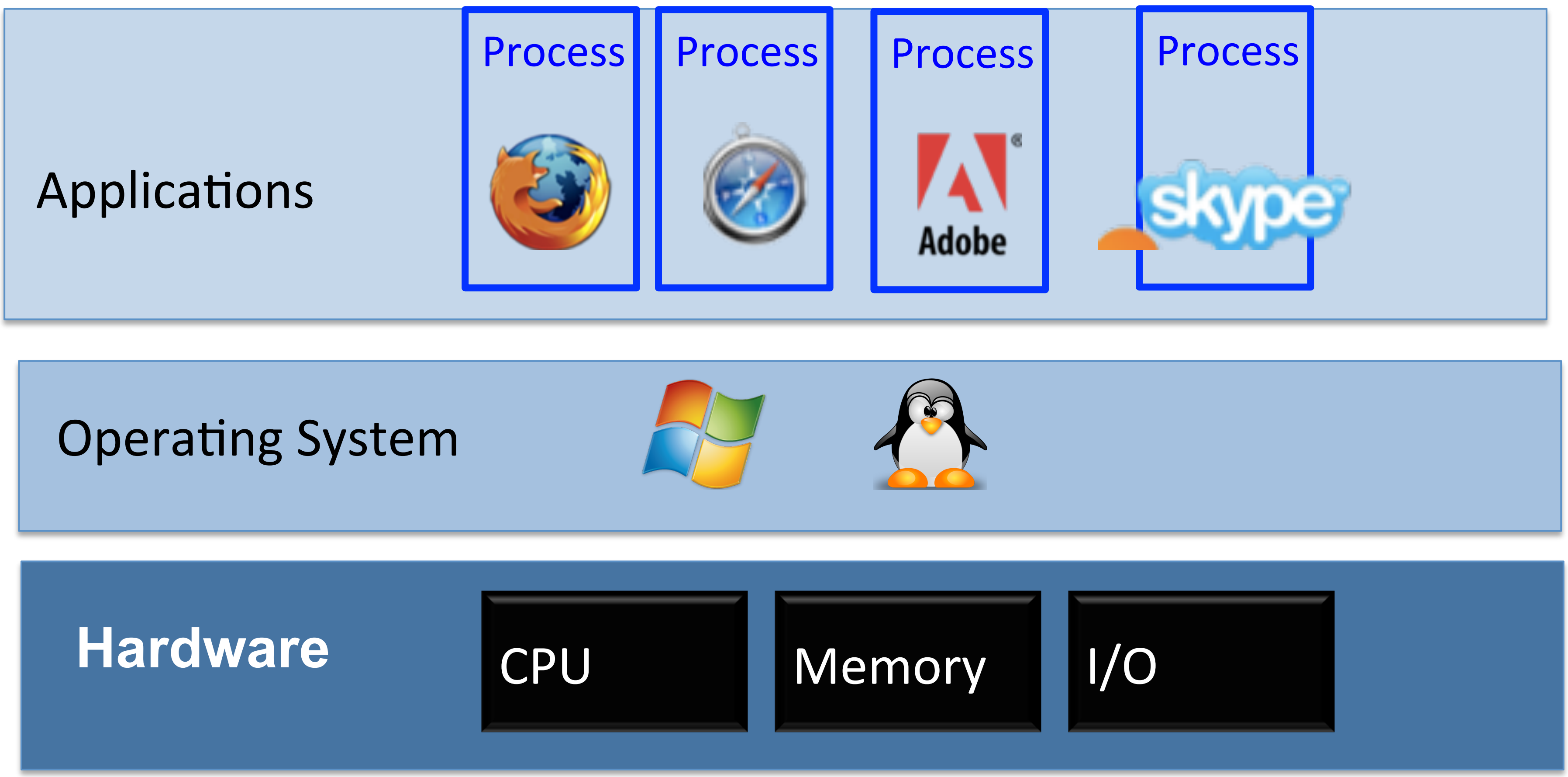
User program and OS interaction Multiprocessing

Jinyang Li

User program and OS interaction

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`, OS launches a process
  - when you type Ctrl-C, the foreground 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. Exception
   – e.g. divide by zero, page fault
2. Traps
   – syscalls (user programs explicitly ask for OS help)
3. Interrupt
   – timer, device events such as keyboard prcess, 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

Syscall: User → OS

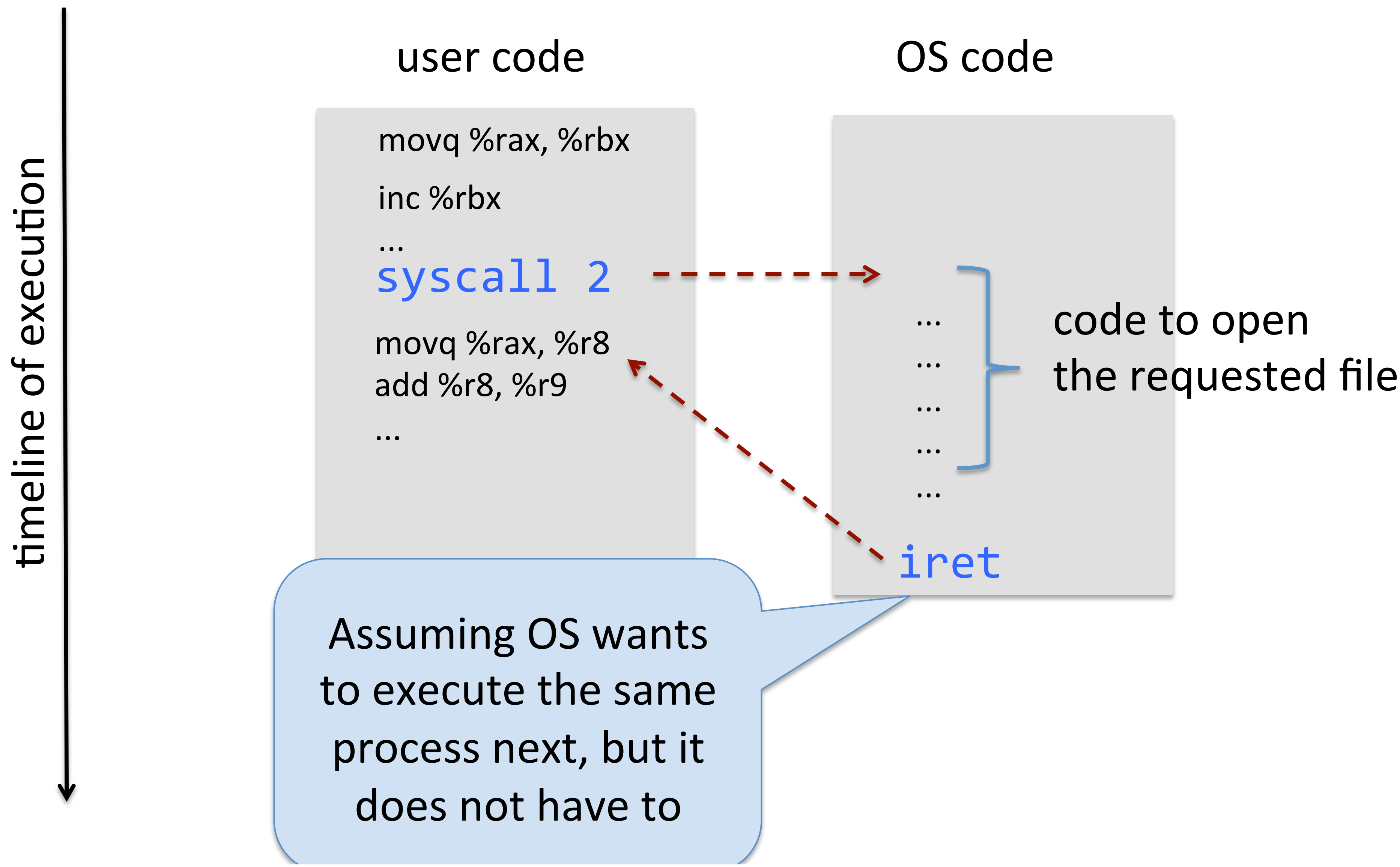
- User programs ask for OS services using syscalls
  - it's like invoking a function in OS
- Each syscall has a known number

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

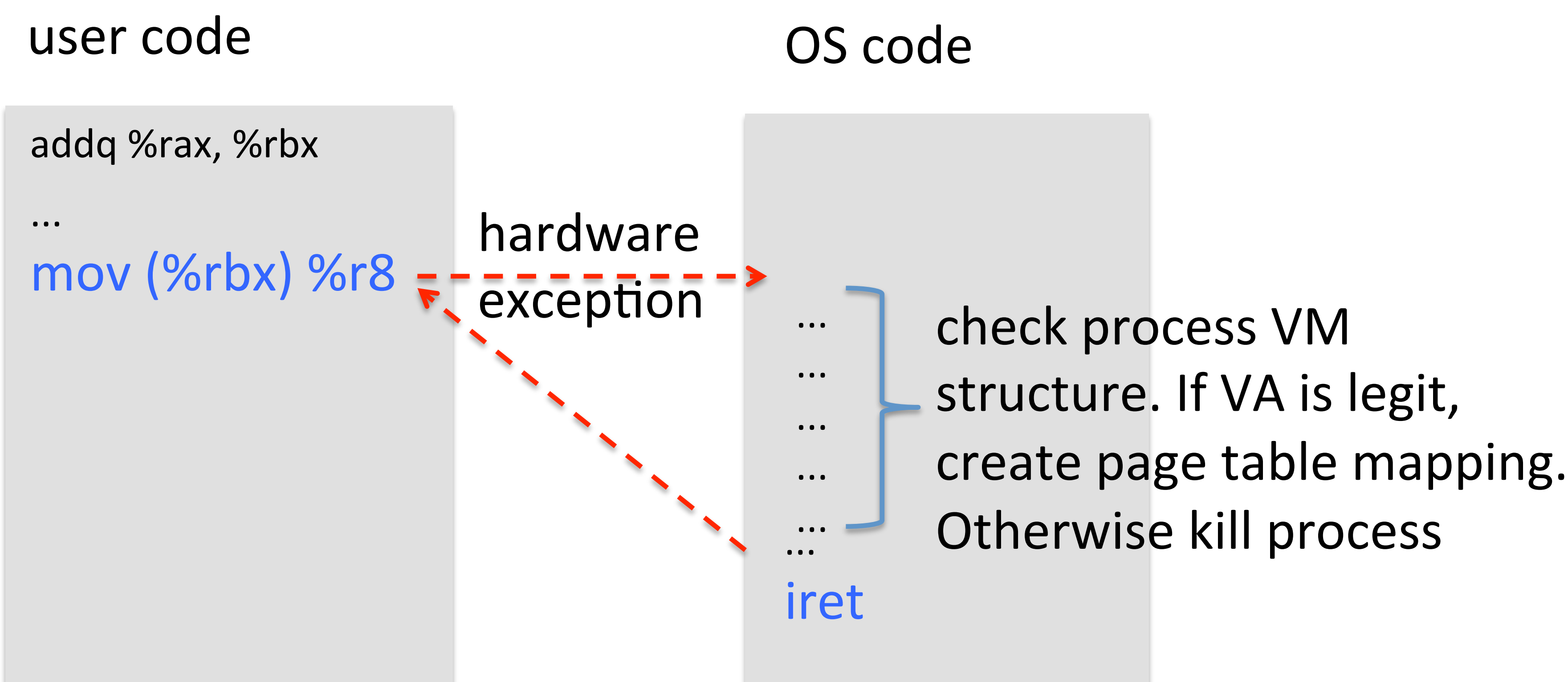
C library wraps around these syscalls to provide file I/O

linux syscall number

Syscall: user → OS



OS also takes control upon exceptions

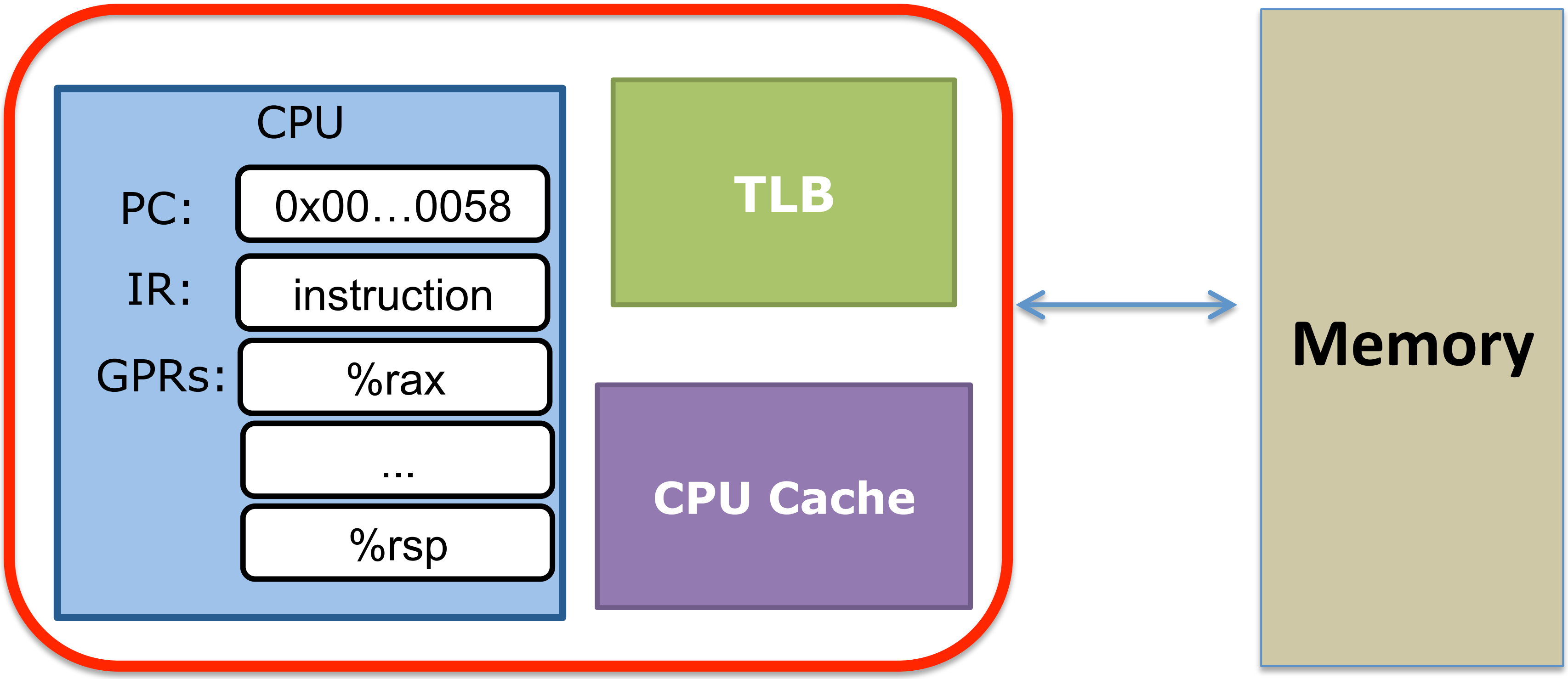


Multi-processing

Goal of multi-processing

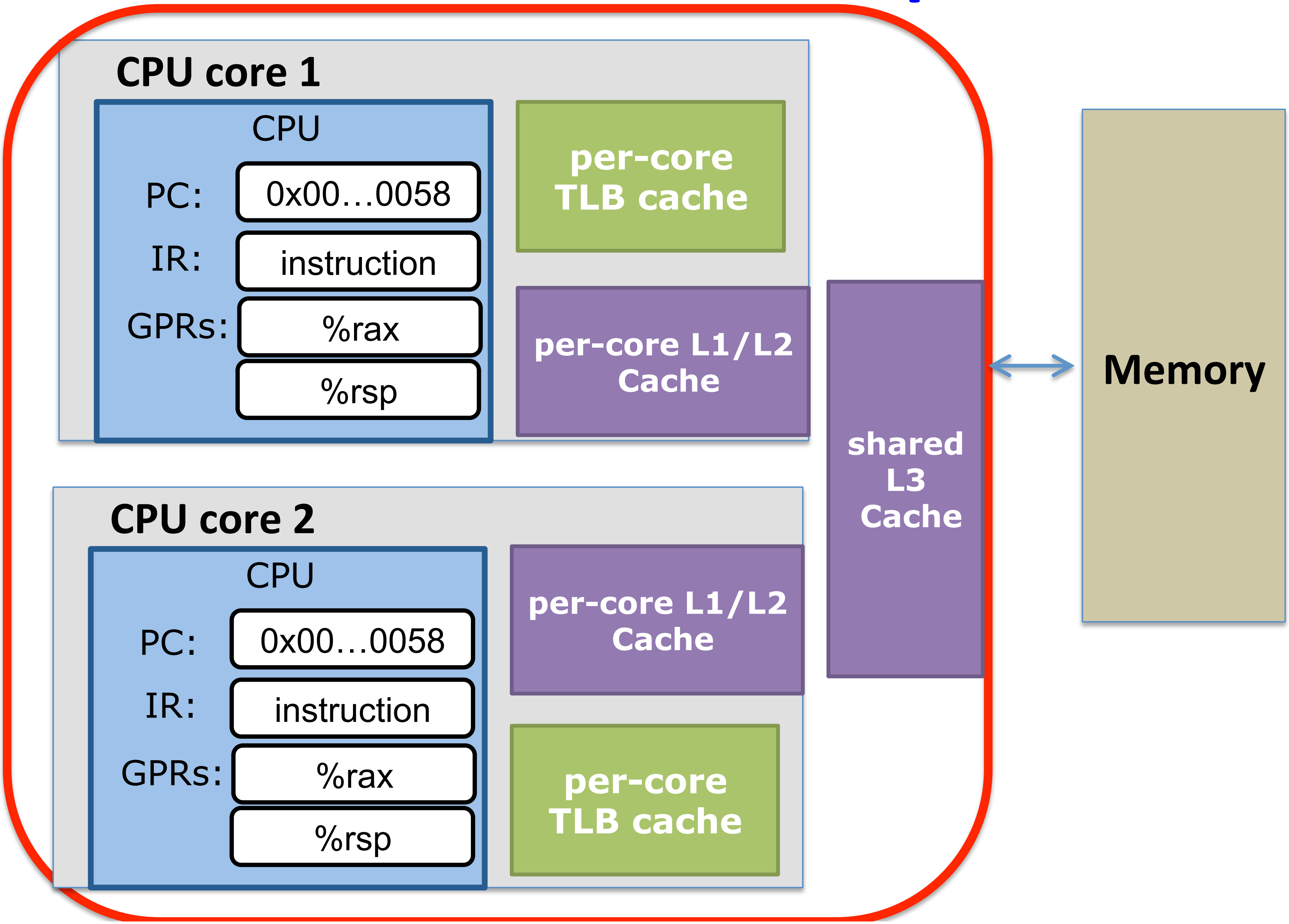
- Run multiple processes “simultaneously”
- Why?
  - listening to music while writing your lab
  - Running a web server, a database server, a PHP program together

Modern CPUs have multiple cores



Your mental model of the CPU as a single core machine

Modern CPUs have multiple cores

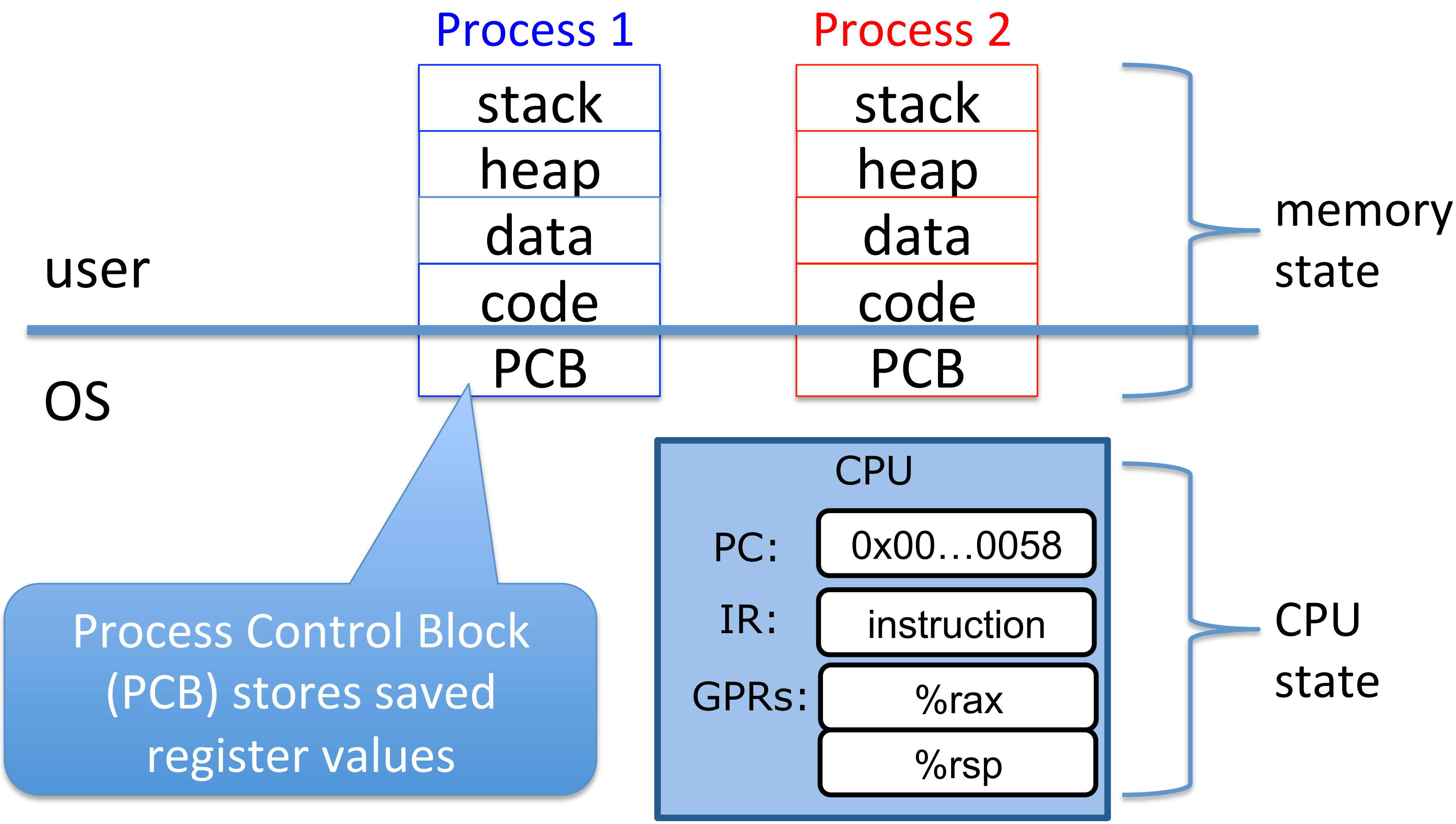


How to multi-process?

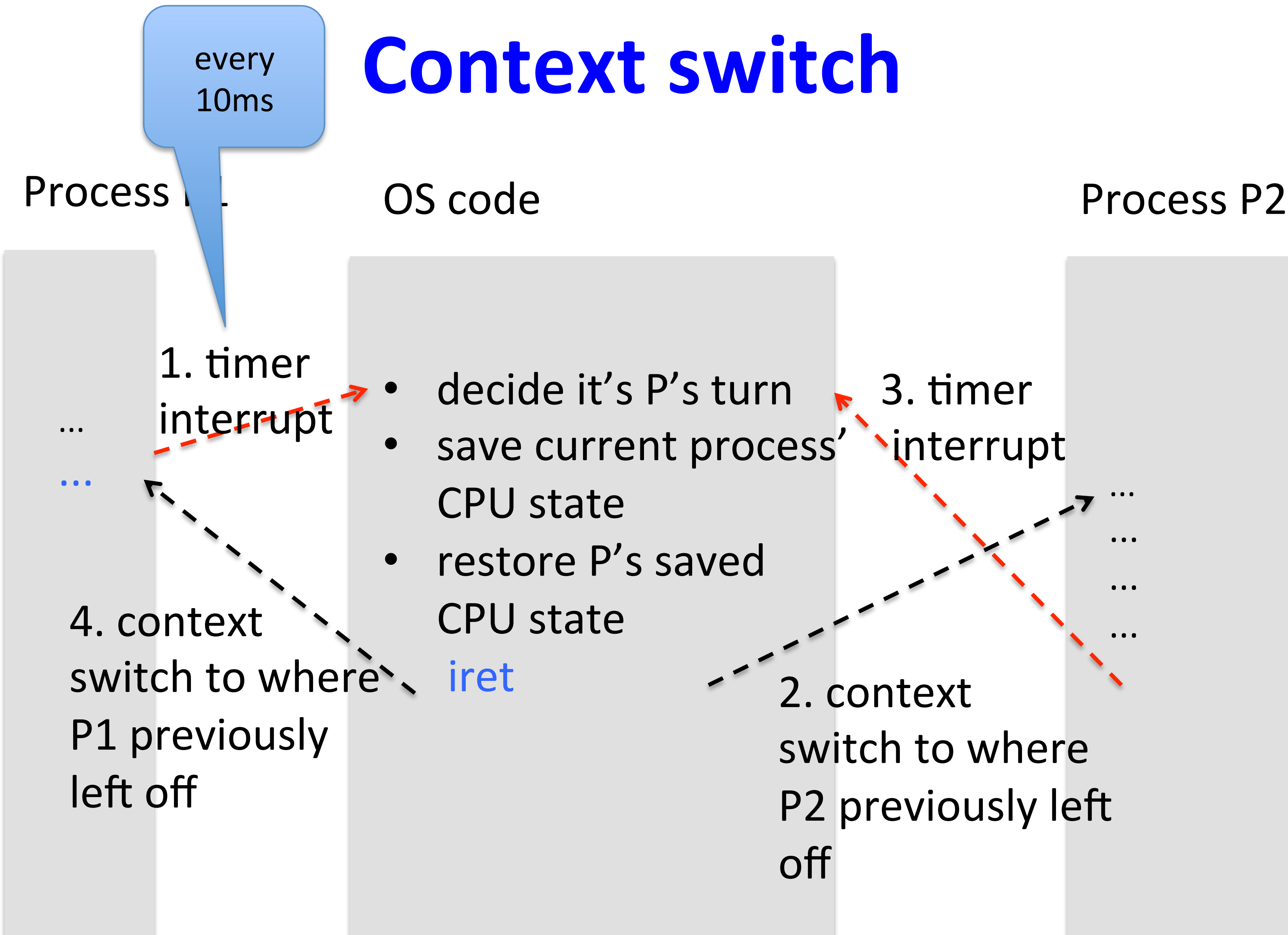
- Execute one process exclusive on each core?
  - 2 cores → 2 processes only 

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

Multiprocessing (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.
  - Launch a program via command-line (the shell program creates the process)

The fork syscall

- 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)

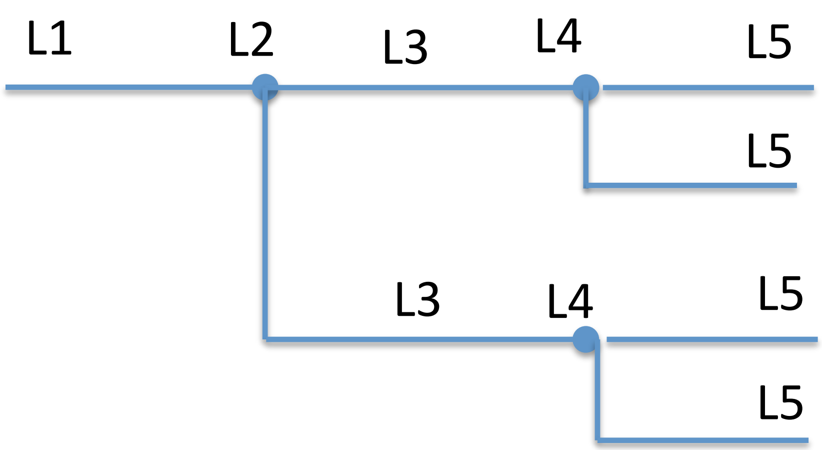
Another fork example

```
void main()
{
    printf("hello\n");
    fork();
    printf("world\n");
    fork();
    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?

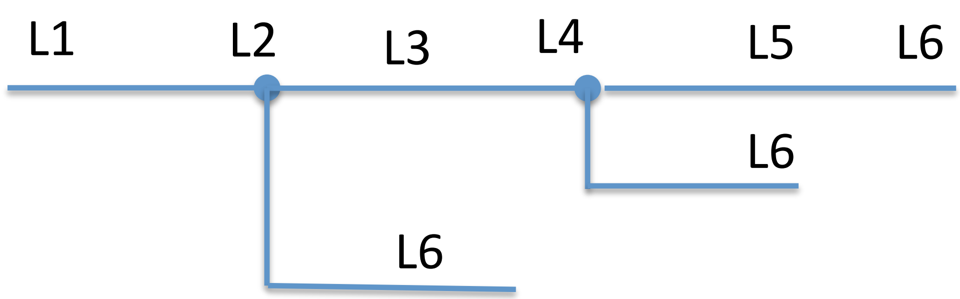
✔	✔	✗
hello	hello	hello
world	world	world
world	bye	world
bye	bye	world
bye	world	bye
bye	bye	bye
bye	bye	bye

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



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