Three Key Ideas

Requirements

- A good OS should …
    - … maximize the utilization of resources (efficiency)
        - Undesirable: a CPU is allocated to execute a user program, but the CPU is often waiting for I/O operations during the allocated time period, when there are other user programs waiting to be executed
    - … allocate resources fairly to multiple requesters (fairness)
        - Undesirable: one requester uses CPU for long time while others cannot use it at all

Requirements

- A good OS should …

    … provide responsive support to real-time requests (responsiveness)

    - Undesirable: enter a character on your keyboard, but it appears on the monitor after 10 seconds

    … prevent the errors made by a single user/program from affecting others (i.e., the OS and other user/programs) (reliability)

    … prevent misuse of resources by user/program (security)

3 Key Ideas in OS Design

- **Multiprogramming** for Efficiency
- **Timesharing** (**Multitasking**) for fairness and responsiveness
- **Dual-mode** execution for resource protection
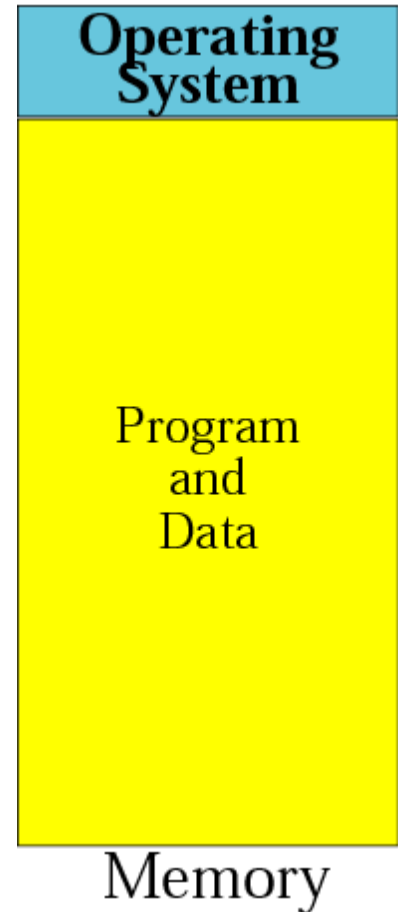
Multiprogramming

Mono-programming
- OS loads a program (job), runs it, and then replaces it with the next program.
- Only one job is in memory for execution.

Problems:
- When one job is being run (i.e., haven't finished), no other program can be executed.
- When the job issues an I/O request (e.g., open a file, read data from a file), it cannot continue until the request is fulfilled – the job blocks on I/O.

Mono-programming cannot keep CPU and I/O devices busy

Operating System
Program and Data

Memory

Multiprogramming

Basic idea of multiprogramming:

- Keep multiple jobs in memory.
- When one job blocks on I/O, the operating system:
  - Starts the I/O operation.
  - Switches to another job that is ready to execute.
  - Now the CPU and I/O device are executing in parallel.
- When the I/O device has completed a request, it generates an interrupt to inform the CPU.
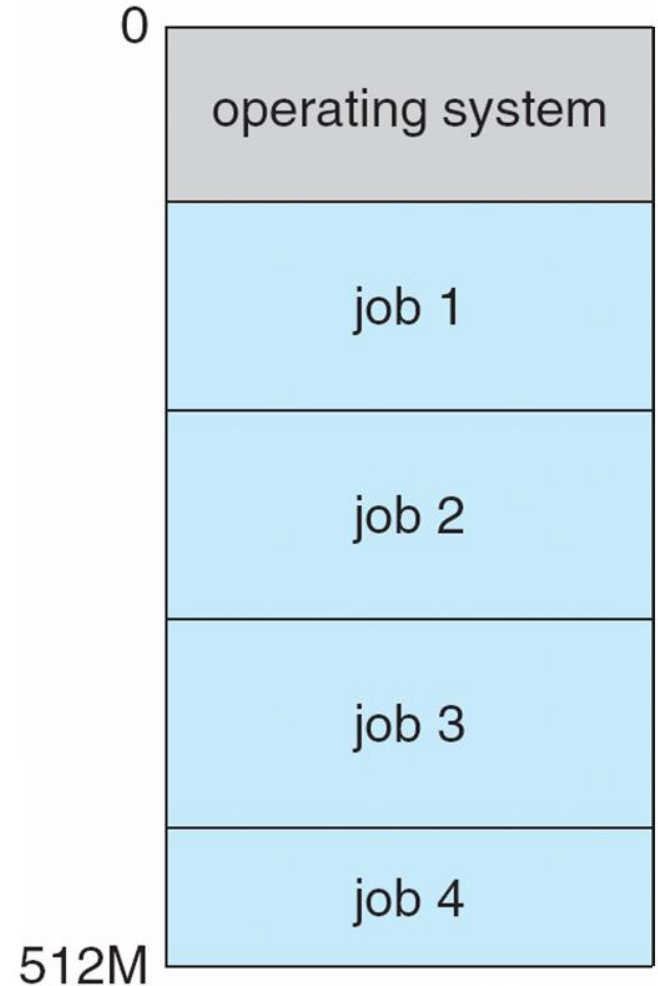
operating system

job 1

job 2

job 3

job 4

0

512M

Comparisons

Suppose there are two programs:

- A: Computation (100ms); I/O on device 1(200ms); Computation(50ms)

- B: Computation (50ms); I/O on device 2 (200ms); Computation(100ms)

How much time will it take to complete both programs when

(1)   Mono-programming is used?

(2)   Multi-programming is used?
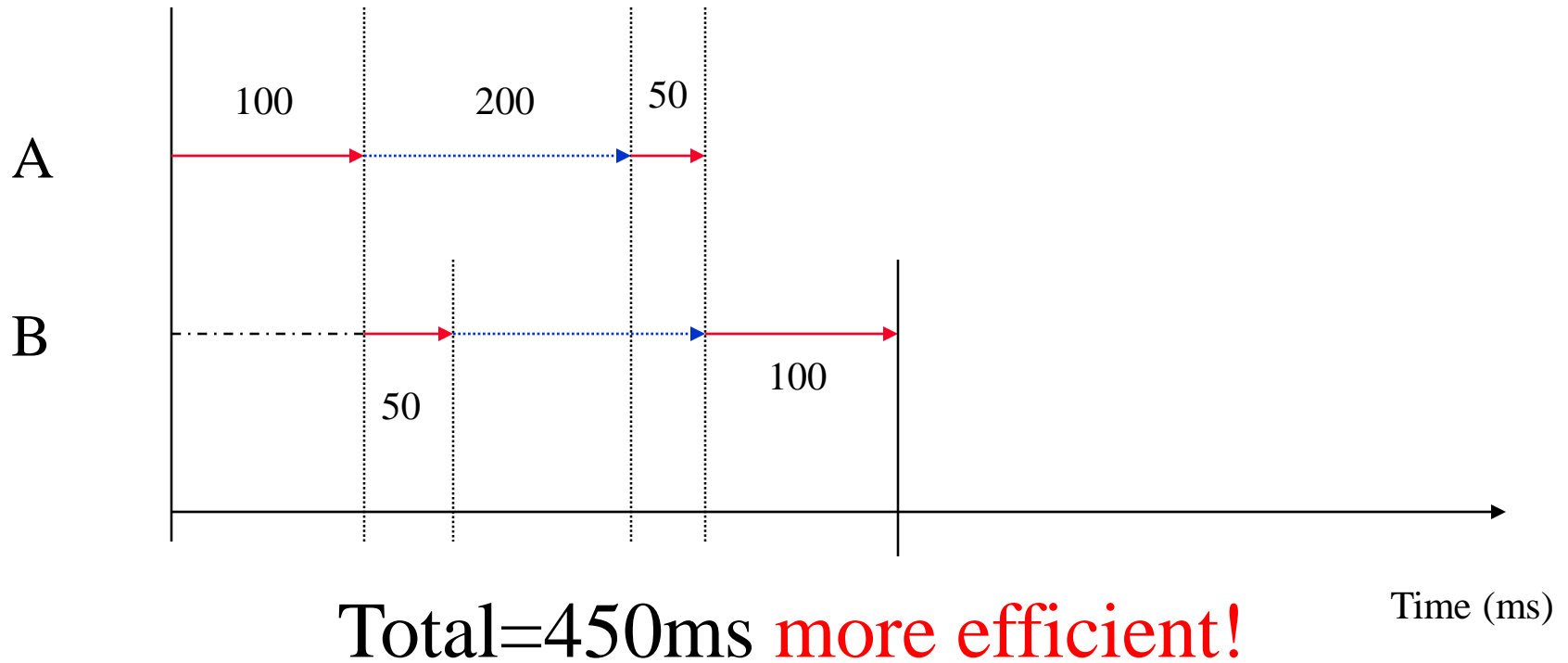
Assume there is only one CPU.

Mono-Programming

| 100 | 200 | 50 | 50 | 200 | 100 |

time

Total=700ms

Multi-Programming

Total=450ms more efficient!
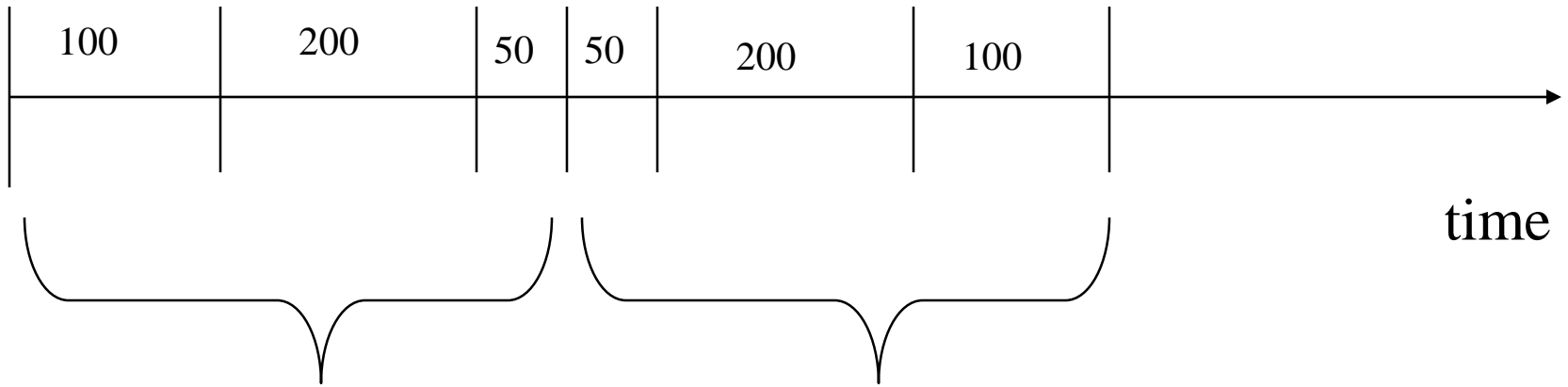
Multiprogramming

- First developed for batch systems in the 60s.
  - Go to the computer center and give them your program (stored on punch cards).
  - The computer operator "batched" several jobs together and loaded them into the computer.
  - Come back at 5:00pm to get the results of your program.
- Batch systems are *non-interactive*.

Time-Sharing (Multitasking)

- Logical extension of multiprogramming termed *multitasking*.
- Typical scenario:
  - Multiple users each uses a terminal to interact with computer.

Internet

Pyrite
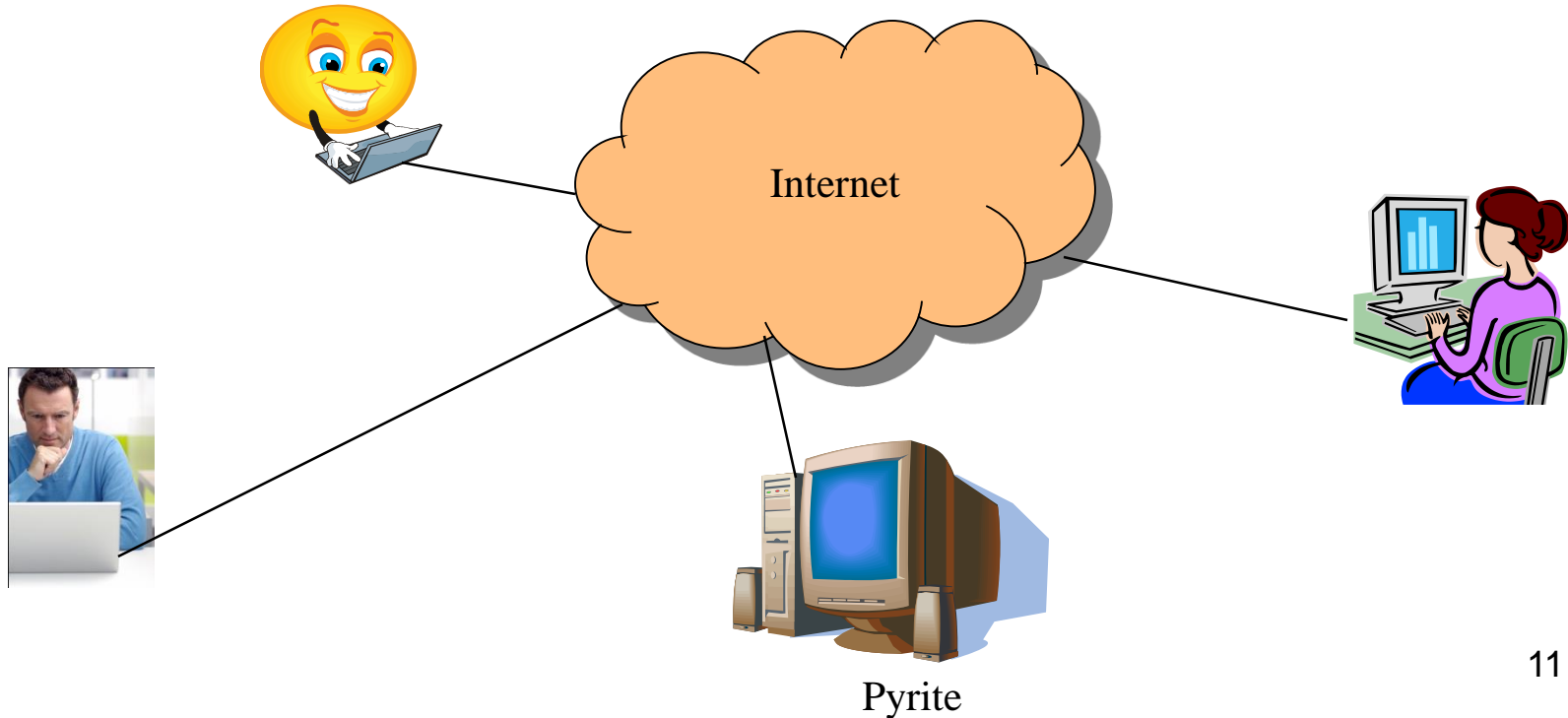
Time-Sharing (Multitasking)

- Each user has at least one program (job) in memory
- Switching between users' jobs every a certain time interval
- Goal is to give the illusion that each user is exclusively using the machine.
- Short response time and fairness in using the resources are achieved.

Internet

Pyrite

12
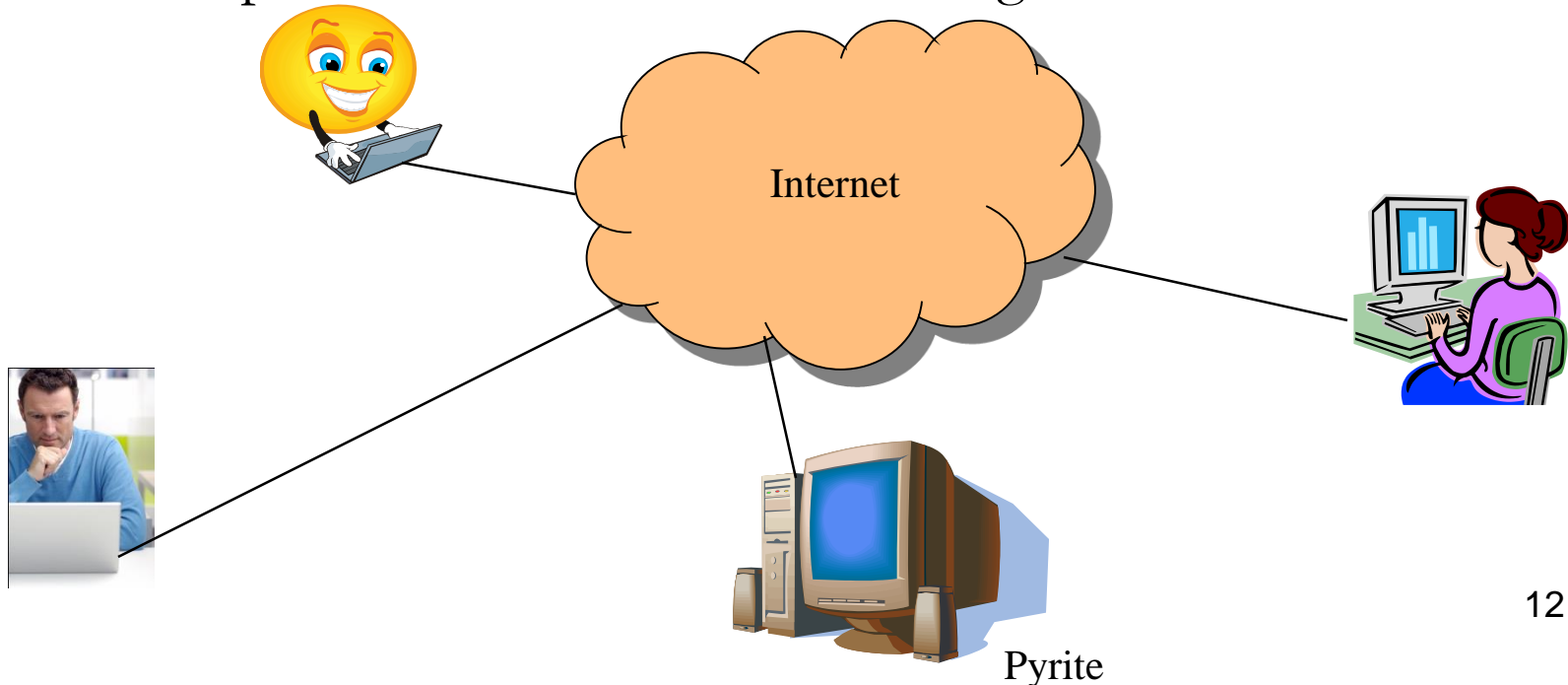
Time-Sharing (Multitasking)

- Not just for multitasking between different users …
- Think that you are writing a paper on your computer while the computer is playing music
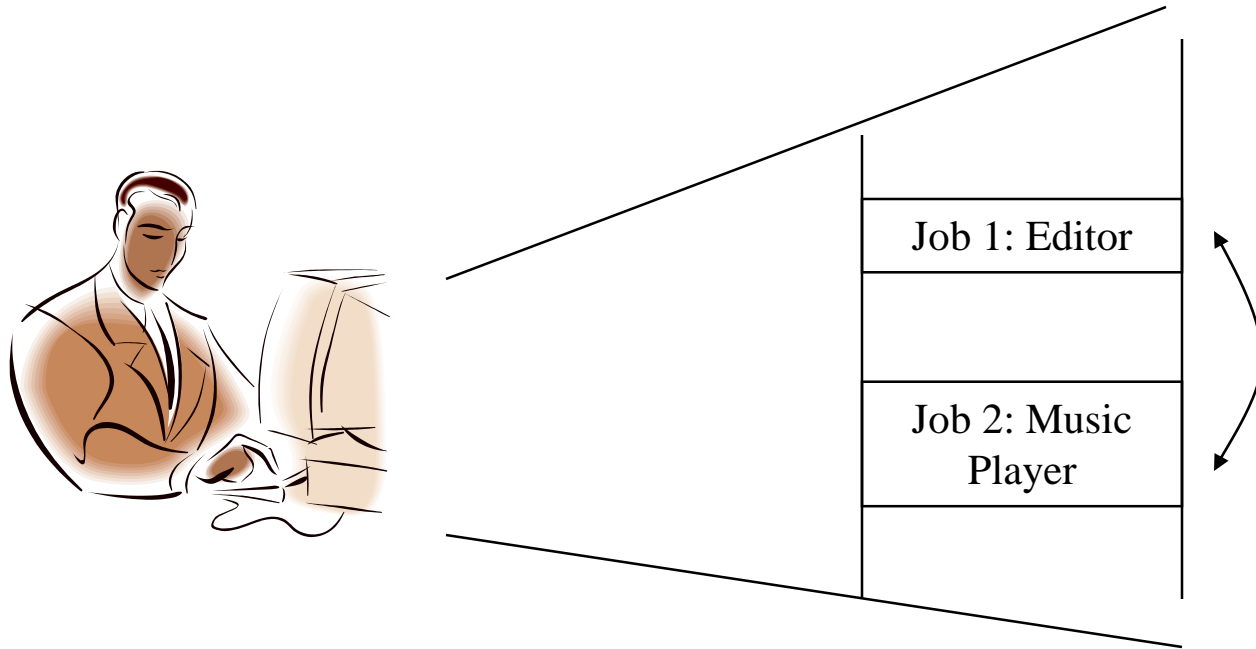
Job 1: Editor

Job 2: Music Player

Comparisons

Suppose there are two programs:

- A: Computation (100ms); I/O on device 1(200ms); Computation(50ms)

- B: Computation (50ms); I/O on device 2 (200ms); Computation(100ms)

How much time will it take to complete both programs when

(1)   Mono-programming is used?

(2)   Multi-programming is used?

(3)   Multi-tasking is used (assume: job switch occurs every 25ms)?

Assume there is only one CPU.

Mono-Programming

| 100 | 200 | 50 | 50 | 200 | 100 |
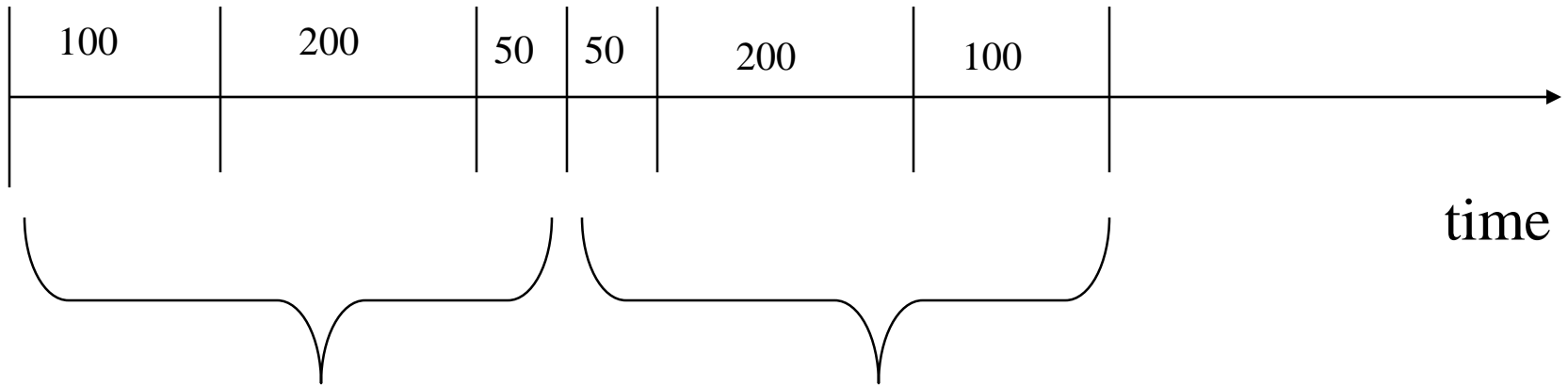
time

Total=700ms

Multi-Programming
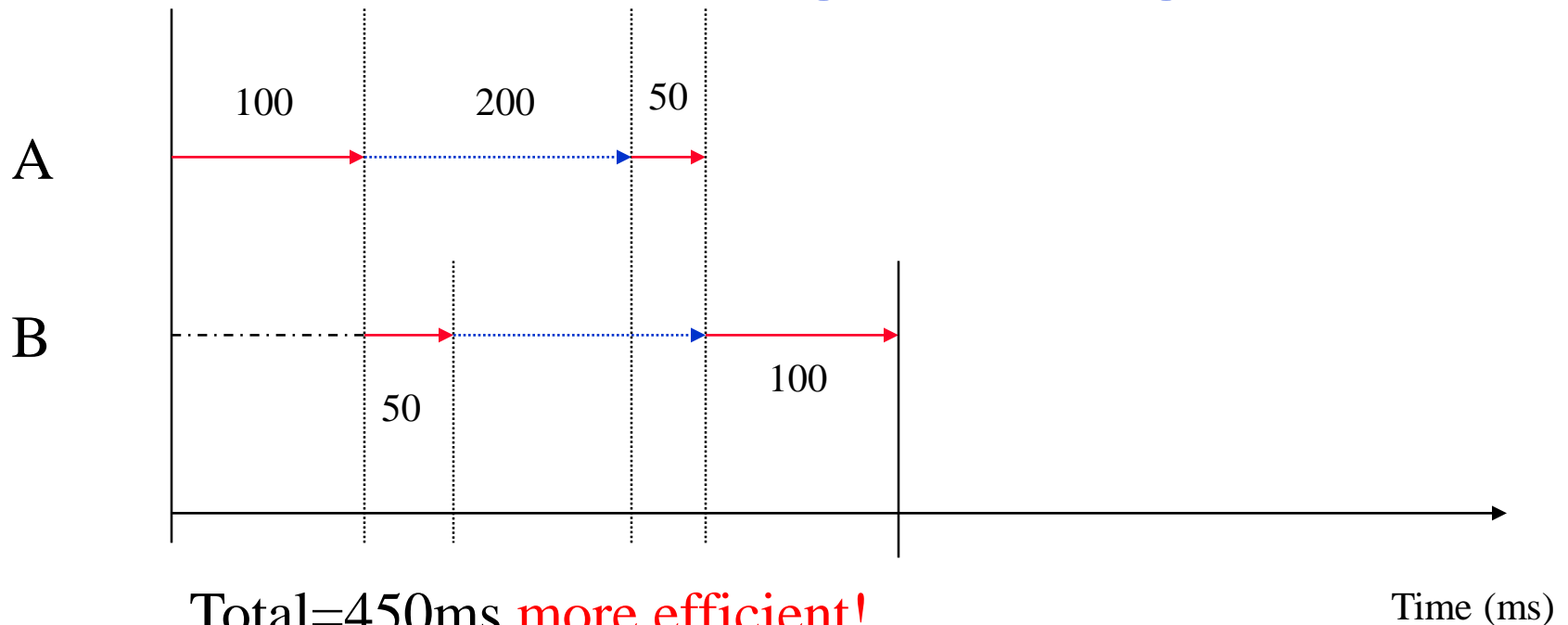
Total=450ms more efficient!

Finish time for job A = 350ms

Finish time for job B = 450ms

Multi-Tasking

→ 25ms

Total=450ms more efficient!
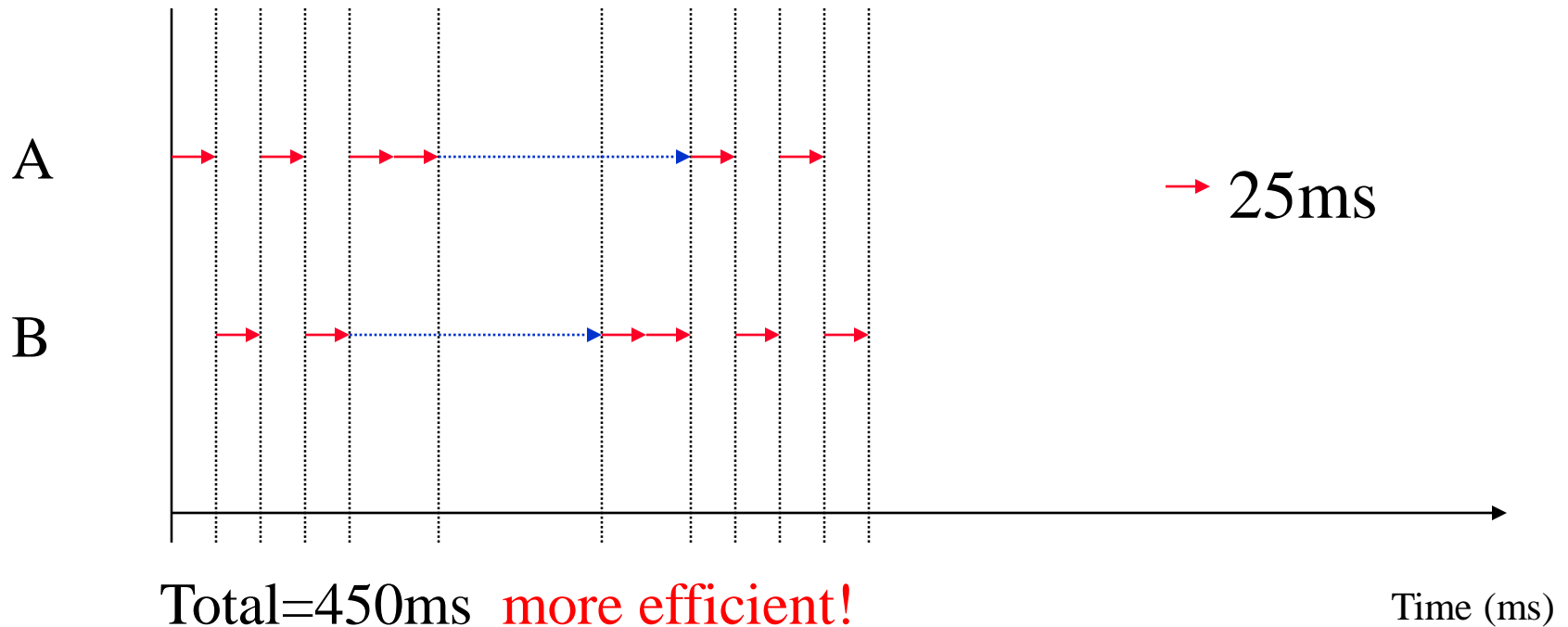
Finish time for job A = 425ms

Finish time for job B = 450ms more fair!

Time (ms)

Protection of System Resources

- What resources should be protected?
  - I/O Devices
  - Memory
  - CPU
  - Files
  - Operating System
- The protection is provided by using both software and hardware mechanisms

Dual-mode Operation

- Most of a contemporary computer system can run in one of at least two modes
  - **User mode** and **kernel mode**
- Some instructions (inappropriate use of which may cause damage to system resources and other user code) are designated as **privileged**, only executable in kernel mode
  - These instructions should be part of OS kernel
- Other instructions can be executed in the user (or kernel) mode and thus can be part of user program.
- Reason for dual-mode: code executed in kernel mode are parts of the OS, and are better tested (and hence more reliable) than user code.

Protection of I/O Devices

- All I/O instructions are privileged instructions.
- User programs can only request the OS kernel to access I/O device on behalf of them (the mechanism is termed system calls).

Example: What's behind printf?

C program:

…
printf("Hello World");
…

C library

…
void printf(…)
{
 …
 send a request (i.e., make a system call) to OS kernel
  …
}

User Mode

OS kernel

Execute the following instruction (within the driver of the monitor):

OUT xx, "Hello World"

A Privileged Instruction

Kernel Mode

21

# Switch between Modes

- When a user program is executed, the system is normally in user mode. It switches to kernel mode when …

    - … an interrupt is triggered by hardware (hardware interrupt)

    - … a software error creates an exception (software interrupt)

    - … user code requesting to operate on system resources (for example, to access an I/O device) creates a trap (i.e., makes a system call)
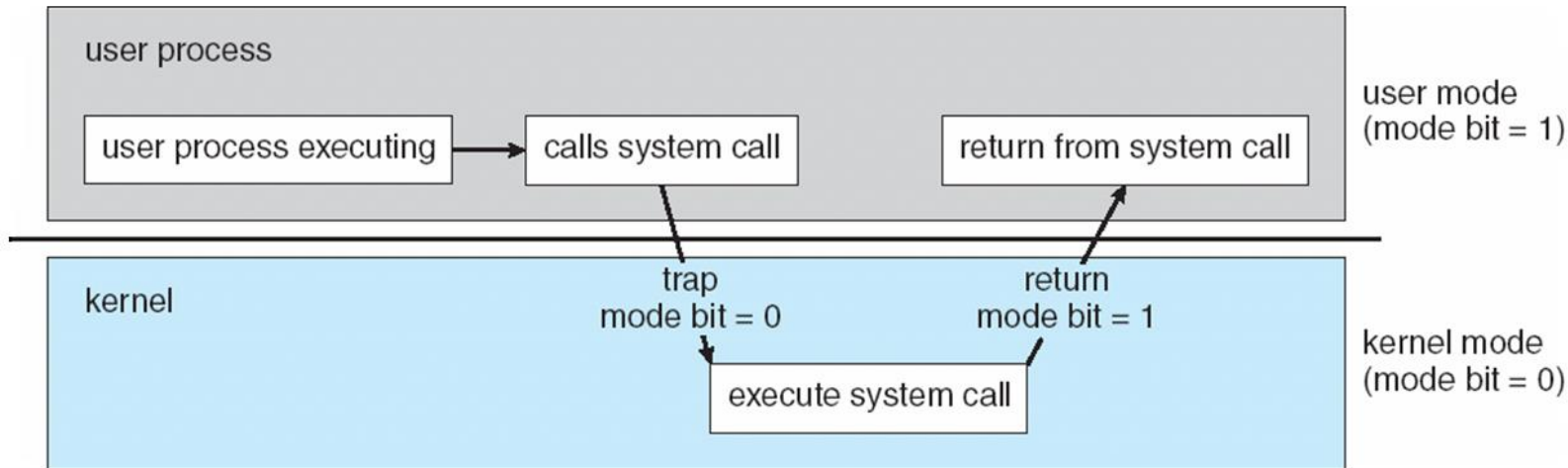
Memory Protection

- Must provide memory protection for kernel data structures (e.g., the interrupt vector, interrupt service routines) and other applications address space.

- Two registers that determine the range of legal addresses a user program may access:
  - **Base register** – holds the smallest legal physical memory address.
  - **Limit register** – contains the size of the range

- Memory outside the defined range of a user program is protected from being accessed by the program.
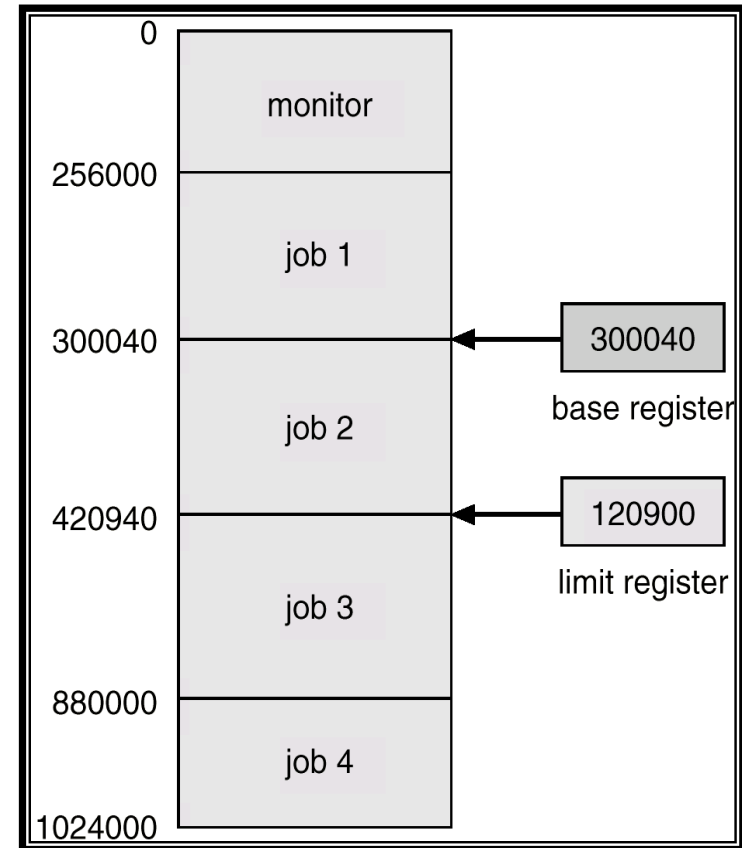
Hardware Address Protection
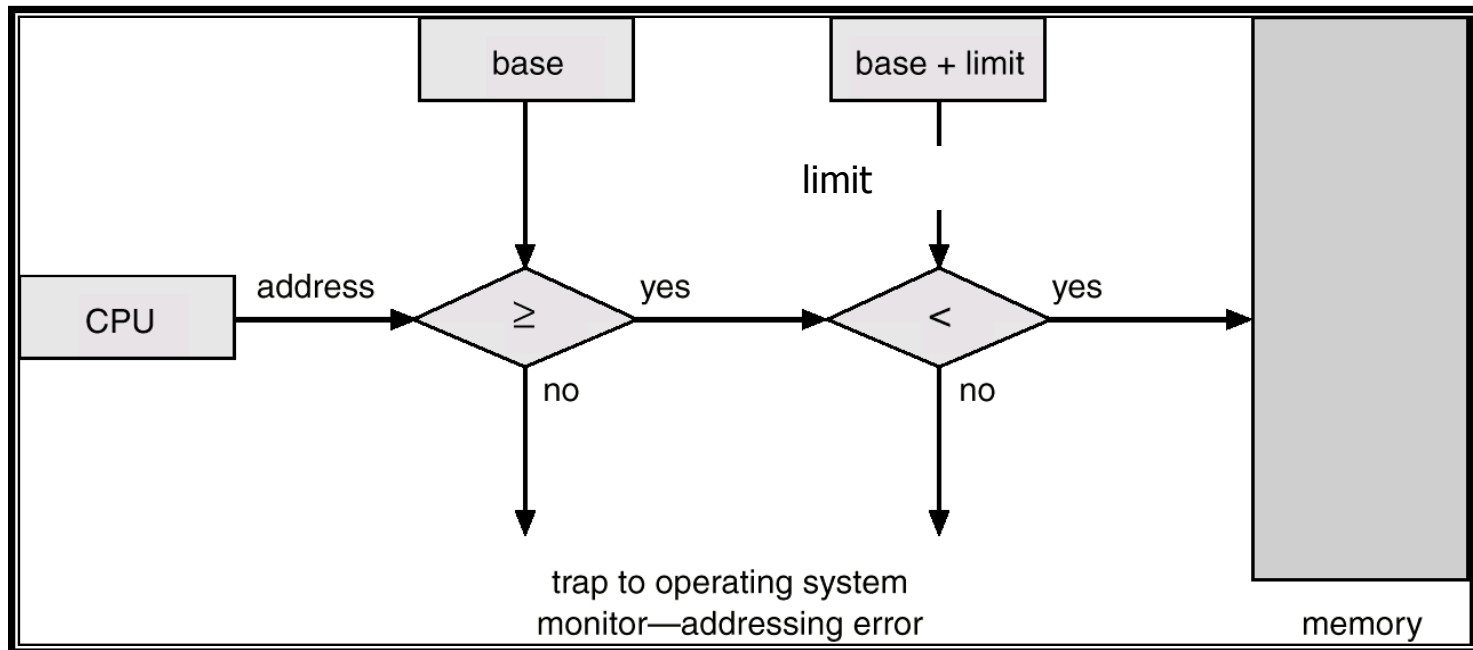
CPU (and OS) Protection

- Goal: Keep user from monopolizing CPU.
  - Ensure OS regains control of CPU every now and then.
- *Timer* – interrupts computer after a specified period to ensure operating system maintains control.

Set timer

Timer fires & triggers interrupt

Timer fires & triggers interrupt

Timer fires & triggers interrupt
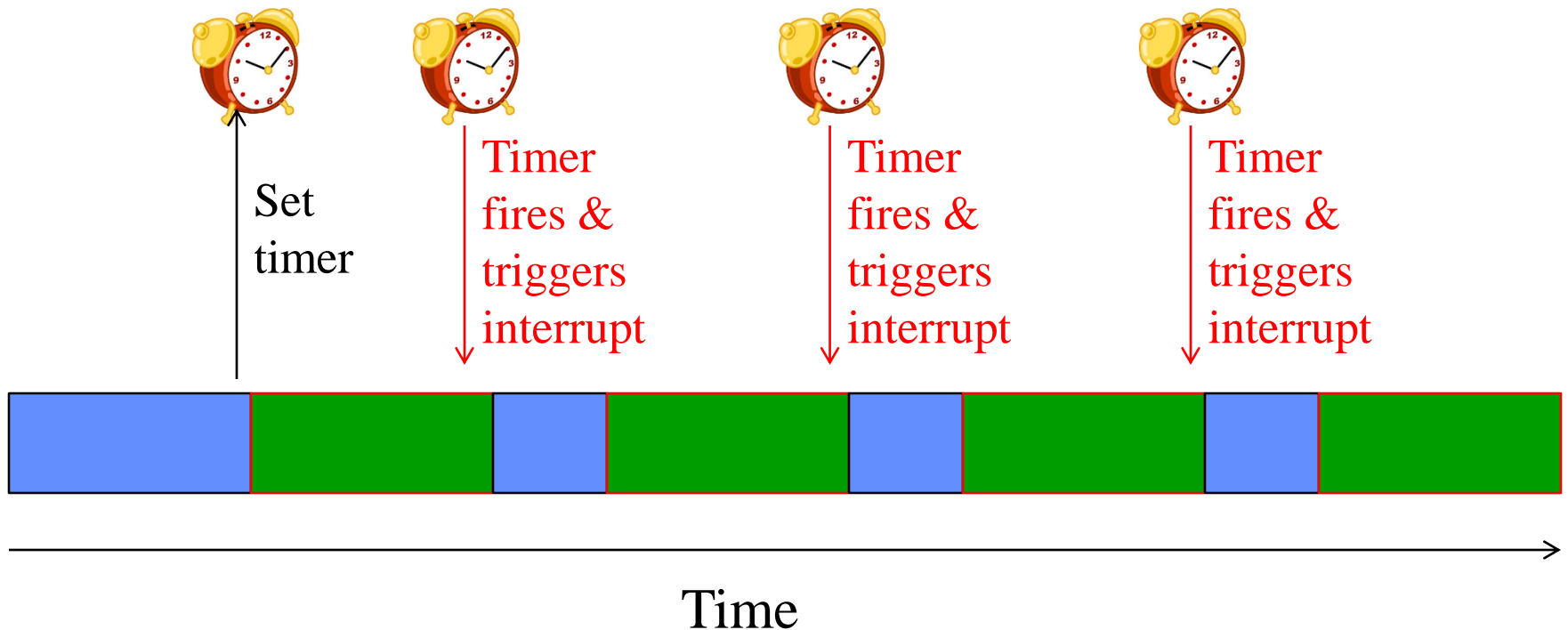
Time

Summary

- Mono-programming vs Multi-programming vs Multi-tasking (Time-sharing)
- User vs Kernel mode; Privileged instructions
- How to protect I/O devices
- How to protect Memory
- How to protect CPU and OS

System Call

A View of Operating System Services

user and other system programs

GUI	batch	command line
user interfaces		

system calls

program execution	I/O operations	file systems	communication	resource allocation	accounting

| error detection | | | | protection and security |

services

operating system

hardware
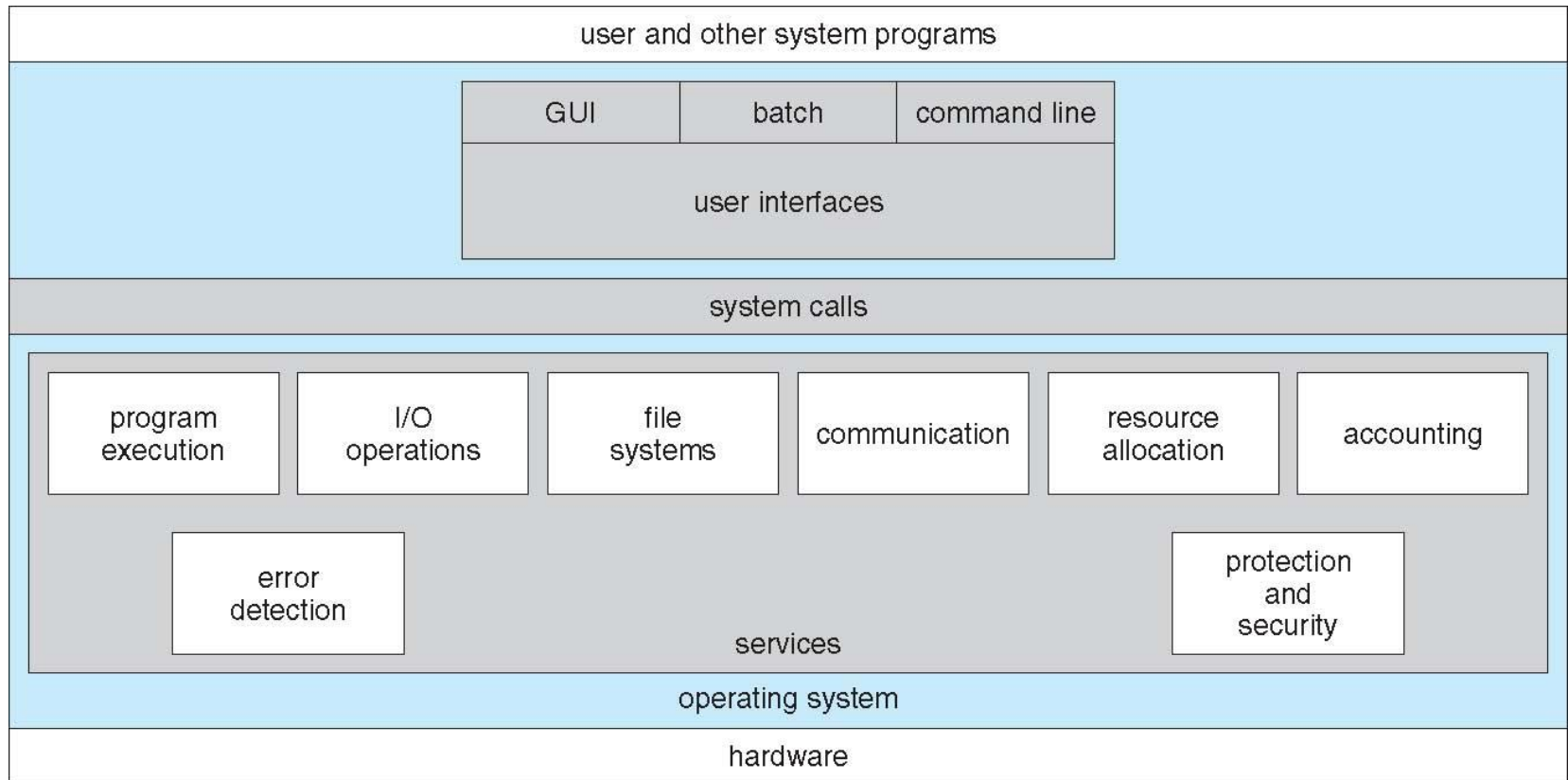
System calls:

Programming interface to the services provided by the OS

System Call Implementation

In kernel

- Associates a number with each system call
- Maintains a table indexed according to these numbers; each entry of the table points to the system service code
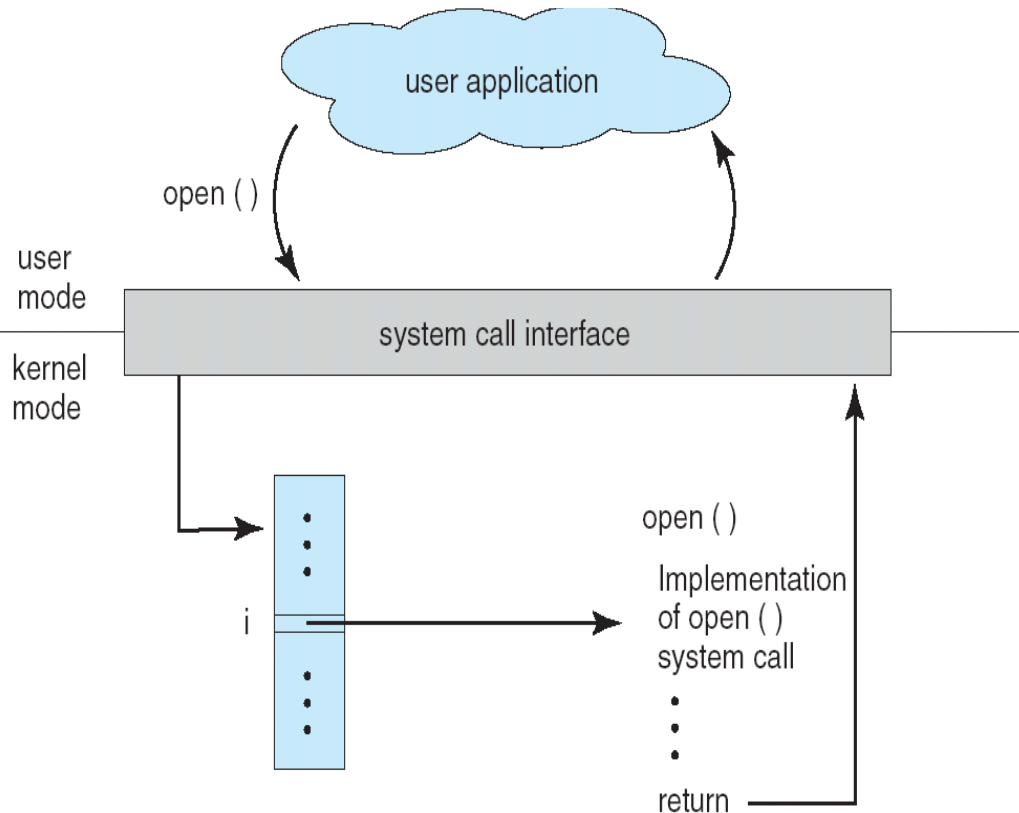
System Call Implementation
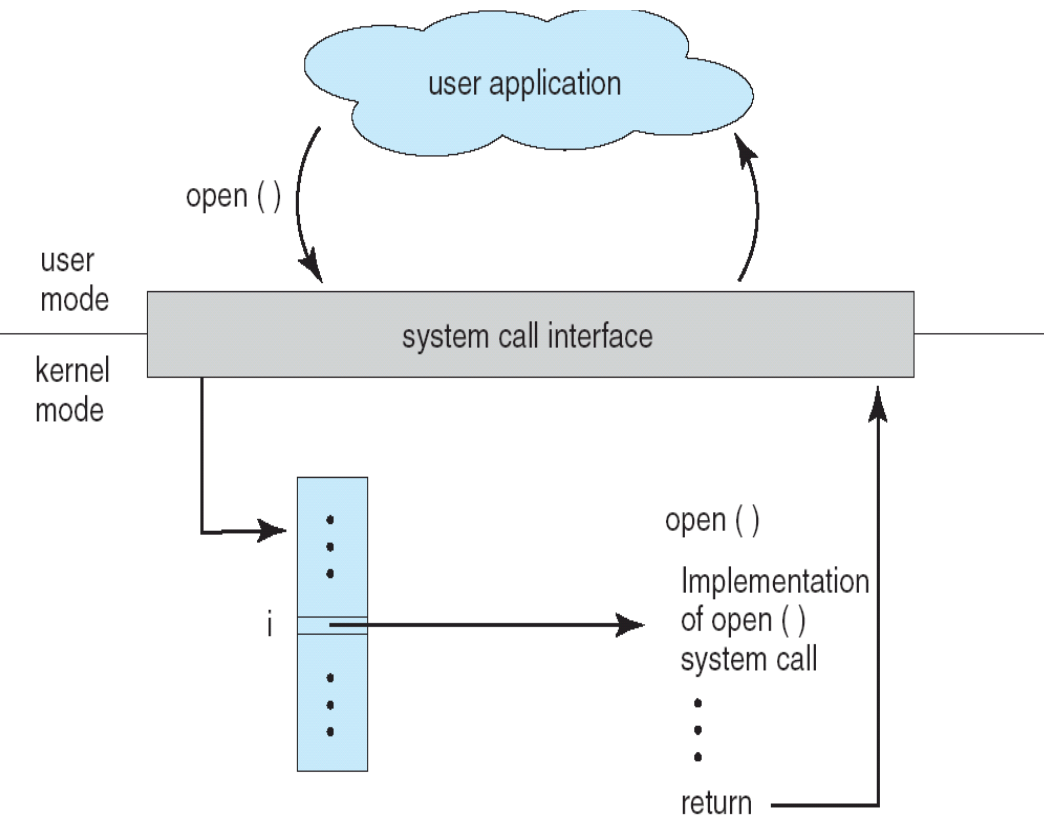
In User mode:

System-call interface is provided by the run-time support system (i.e., part of the libraries of a compiler)

System Call Implementation
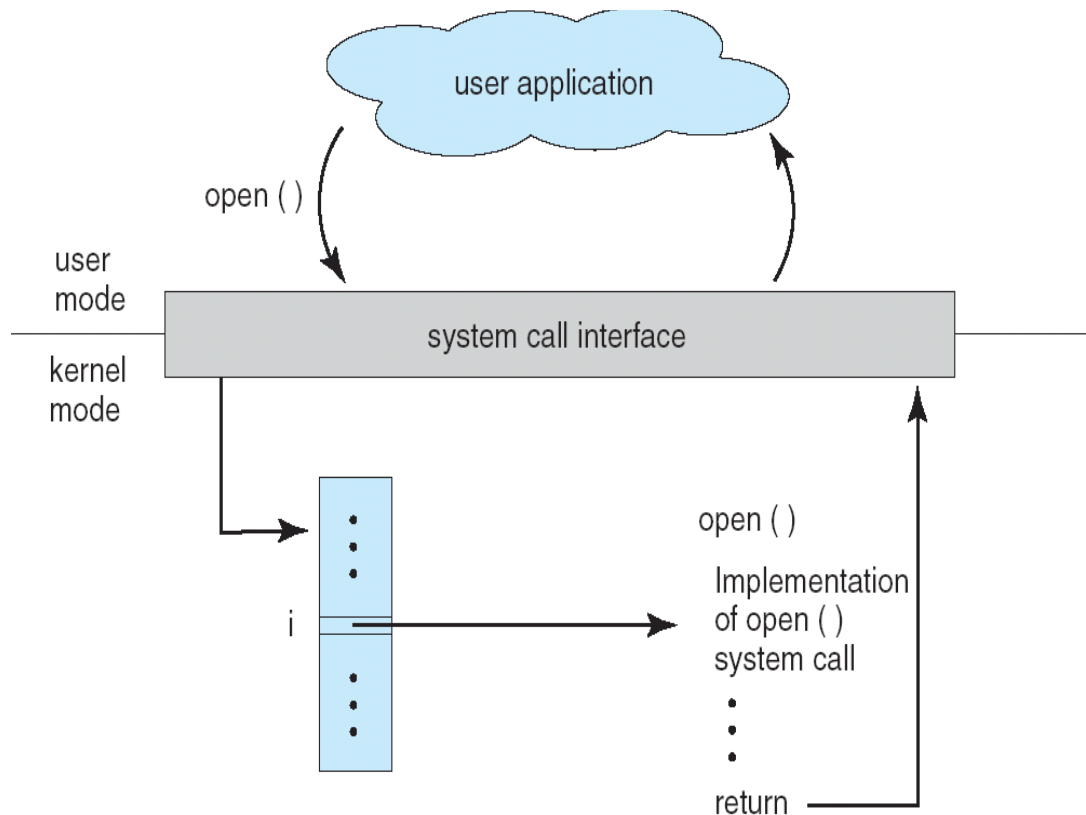
The system call interface includes stub functions that can invoke the implementation code of intended system calls in OS kernel and return status of the system calls and any return values

System Call Implementation
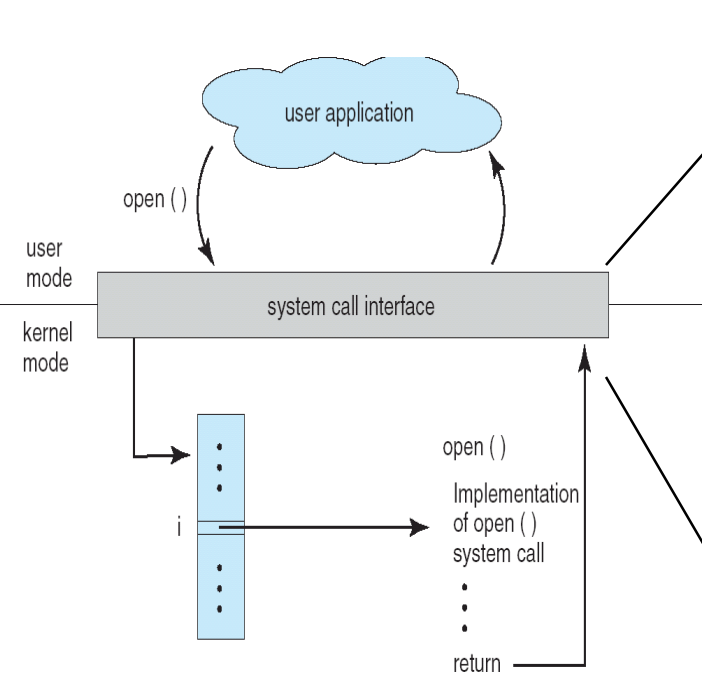
Example: In system call interface (library), stub function for open:

```
open(){
    … load arguments…
    eax ← _NR_open; //NR_open is the internal ID of the system call

    INT 80h;//interrup is triggered
    …
}
```

Types and Examples of System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
    - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
    - Simplest: pass the parameters in *registers*

System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
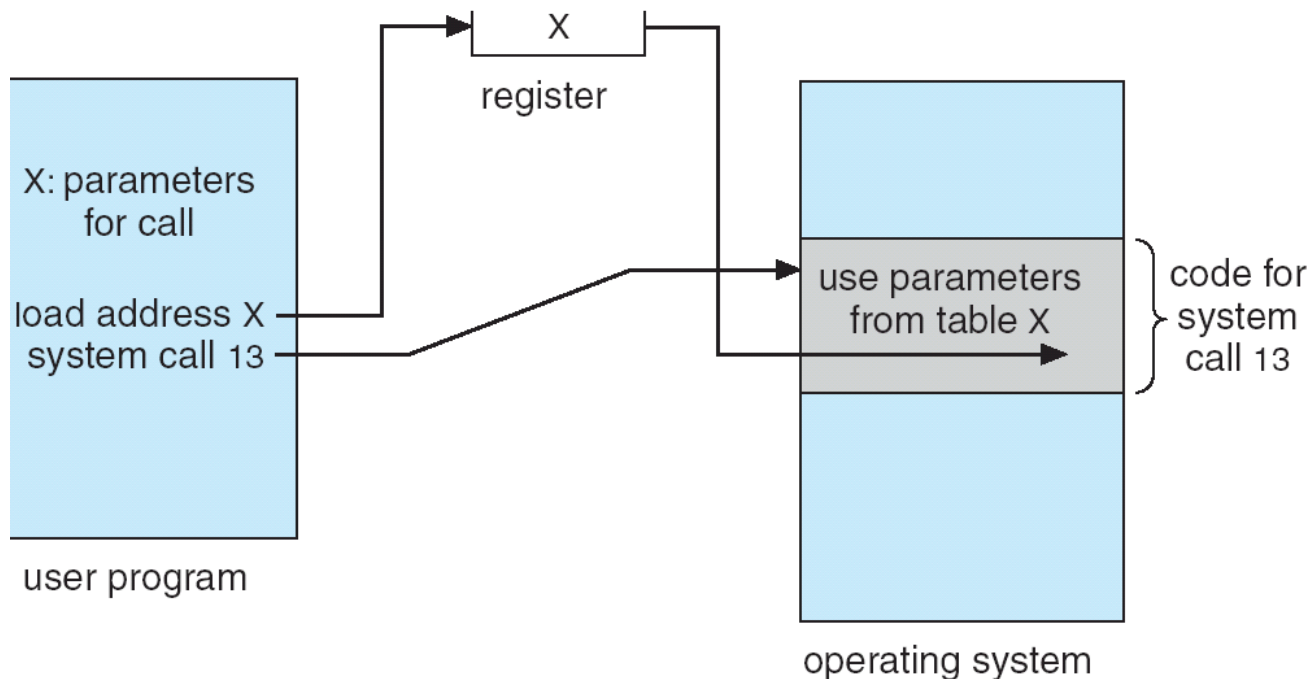
System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

System Calls vs API

- System services are mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call

- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

- Why use APIs rather than system calls? Portability! Transparency!

System Calls vs API

- The caller needs know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result
  - More details of OS interface hidden from programmer by API (provided by programming language libraries)

Standard C Library Example

C program invoking printf() library call, which calls write() system call


```
#include <stdio.h>
int main ( )
{
    •
    •
    •
    printf ("Greetings");
    •
    •
    •
    return 0;
}
```

user mode

kernel mode

standard C library

printf
↓
write

write ( )

write ( )
system call
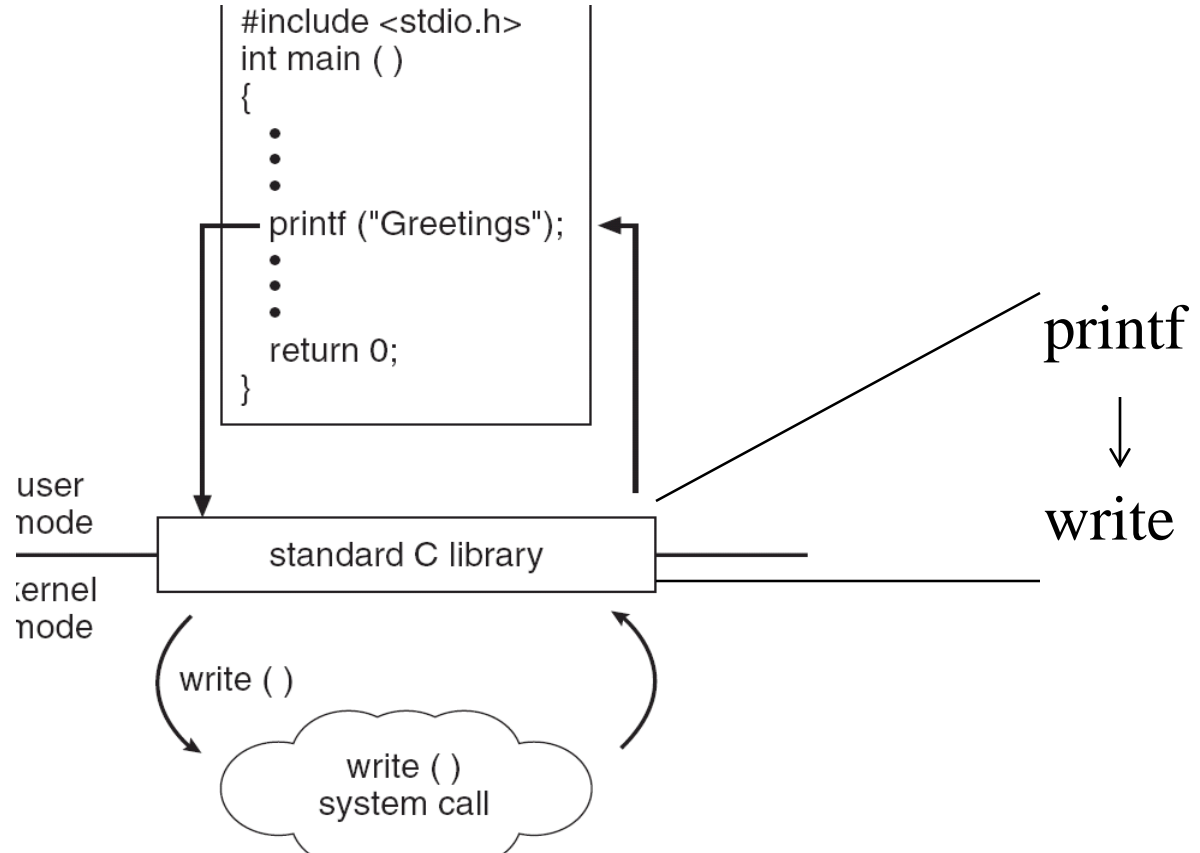
System Programs

- System programs provide a convenient environment for program development and execution. They fall into these categories:
    - File manipulation: ls, mkdir, rm, cd, …
    - Status information of the system: date, uname, df, …
    - File modification: cat, chmod, ..
    - Programming language support: cc, gcc, …
    - Program loading and execution: ld, link, …
    - Communications: talk, message, …
- Implementation: Some are simply user interfaces to system calls; others are considerably more complex
- Most users' view of the operation system is defined by system programs, not the actual system calls

Example

System program: mkdir

Usage: mkdir [OPTION] … DIRECTORY …
Create the DIREcTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.
-m, --mode=MODE set file mode (as in chmod), not a=rwx – umask
-p, --parents no error if existing, make parent directories as needed
-v, --verbose print a message for each created directory
-Z, --context=CTX set the SELinux security context of each created
            directory to CTX
    --help   display this help and exit
    --version   output version information and exit

(run "mkdir –help" can get the above)

Example

SYNOPSIS
    #include <sys/stat.h>
    #include <sys/types.h>
    int mkdir(const char *pathname, mode_t mode);

DESCRIPTION
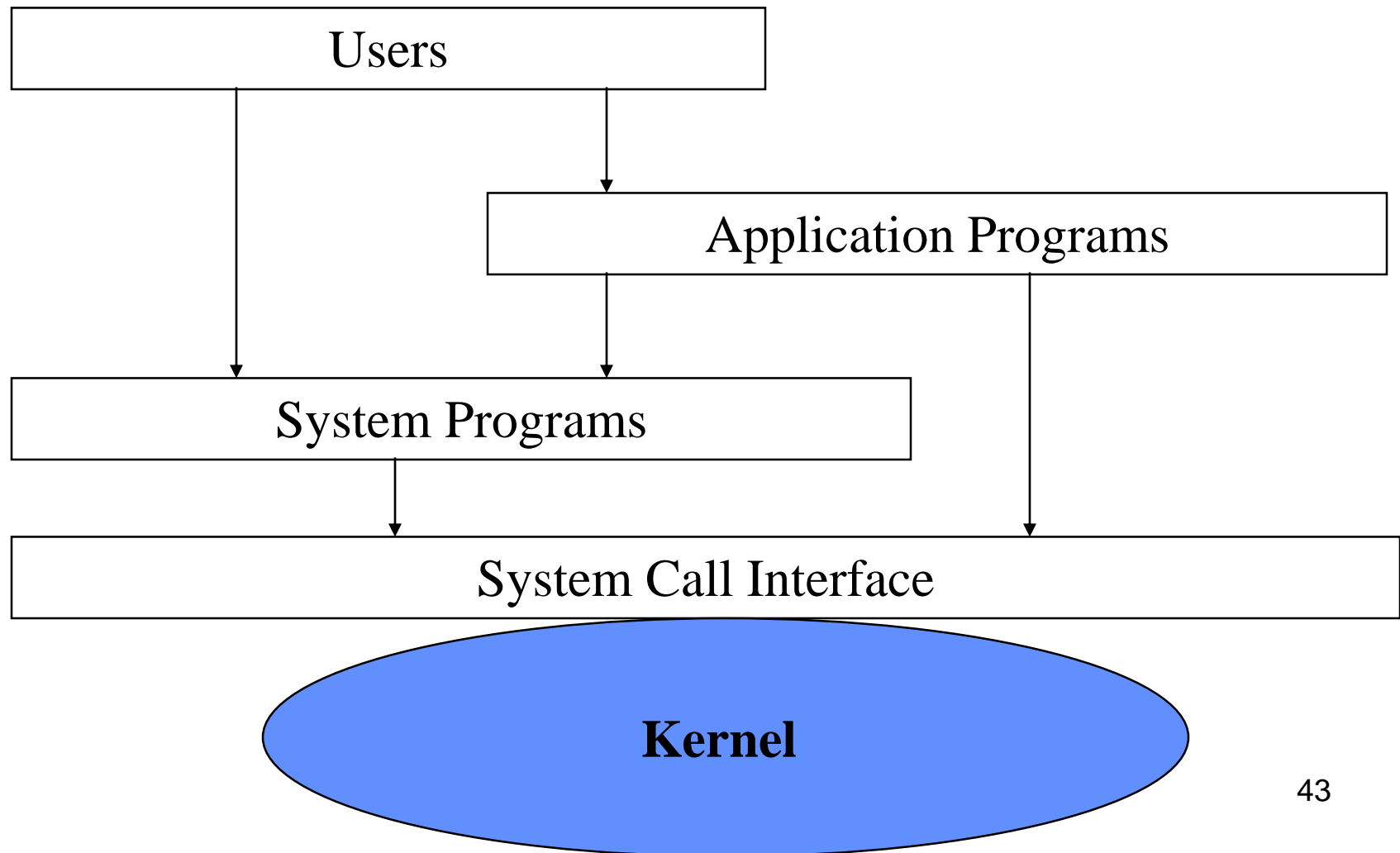   mkdir() attempts to create a directory named pathname.

    The argument mode specifies the permissions to use. It is modified by
    the processes umask in the usual way: the permission of the created
    directory are (mode & ~umask & 0777). Other mode bits of the created
    directory depend on the operating system. For Linux, see below.

    … ….

(run "man 2 mkdir" can get the above)

Relation between Users, Apps, System Calls, and System Programs

```
┌──────────────────────────────────────┐
│                Users                 │
└──────────────────────────────────────┘
        │                    │
        │                    ▼
        │        ┌──────────────────────────────────────┐
        │        │        Application Programs           │
        │        └──────────────────────────────────────┘
        │                    │
        ▼                    ▼
┌──────────────────────────────────────┐
│           System Programs            │
└──────────────────────────────────────┘
        │                    │
        ▼                    ▼
┌──────────────────────────────────────────────────────────┐
│              System Call Interface                        │
└──────────────────────────────────────────────────────────┘

                    Kernel
```

Summary

- System call vs procedure all
- System call vs programming language API
- Implementation details of system calls
- System programs vs Application programs