Memories

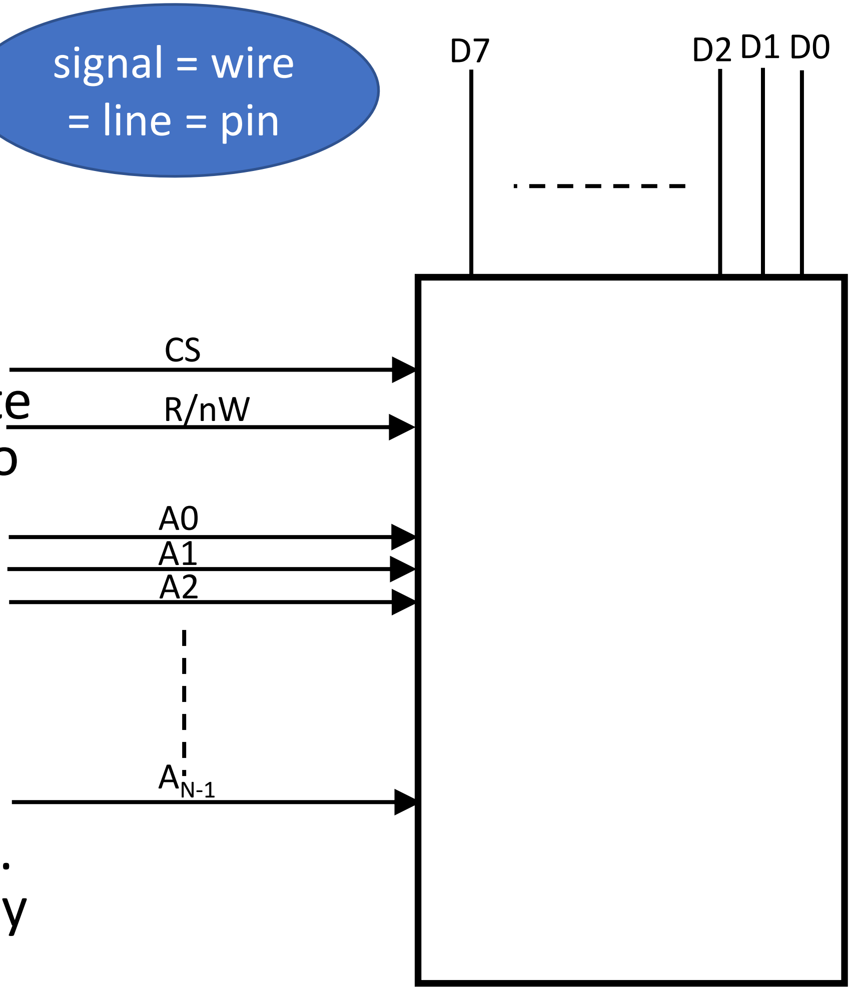
- The basic unit of computer storage is the **bit**.
    - A bit can contain one of two values, 0 and 1.
    - All other storage in a computer is based on collections of bits.
- A **byte** is 8 bits
    - On most (if not all) computers it is the smallest accessible (i.e. via reads/writes) unit storage. For example, most computers don’t have an instruction to move a bit but do have one to move a byte.
- A **word** is the computer’s native unit of data. Thus it is architecture-dependent
    - In a 64-bit system, a memory word is 8 bytes. CPU registers are 8 bytes too.
    - In a 32-bit system, a memory word is 4 bytes. CPU registers are 4 bytes.

    - A computer is most efficient when it accesses external memory one word at a time.
        - It can still access smaller unites from the main memory, e.g. a single byte.

Memories (main memory)

- A memory is digital chip that stores data generally accepts:
  - CS (**chip select**) signal - selects the entire chip. If this signal is not asserted, then all other signals do not matter.
  - **Address** signals (pins) – select a memory byte within the chip, where data will be written to or read from.
    - The address signals combine to form an **address bus**.
  - R/nW (**read, not write**) signal – tell the chip whether data will be written to or read from the chip.
- The **Data bus** (D7 – D0) is an input/output signal.
  - If R/nW is high, then the data bus is driven by the memory chip → Memory produces data
  - If R/nW is low, then the data bus is driven by the CPU → Memory accepts data.

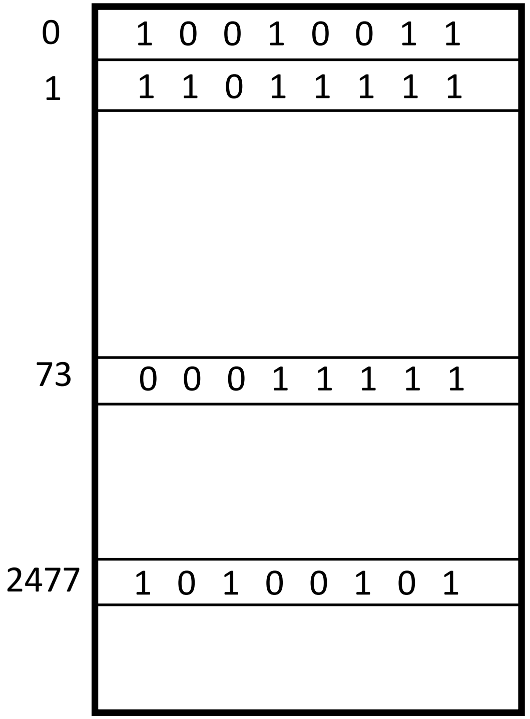
signal = wire = line = pin



A byte-addressable memory

Memories – cont.

- For N address bits 2^N bytes may be addressed.
  - Ex: 16-bit address bus may address 64K bytes
  - 20-bit address bus may address ? Bytes
- Memories may also be **word-addressable**, e.g. In a 64-bit system, a memory may be addressed as 8 bytes (64 bits).
  - In such case, the lower address bits may not be used. How many?
- Alternatively, it may be byte-addressable (8-bits).

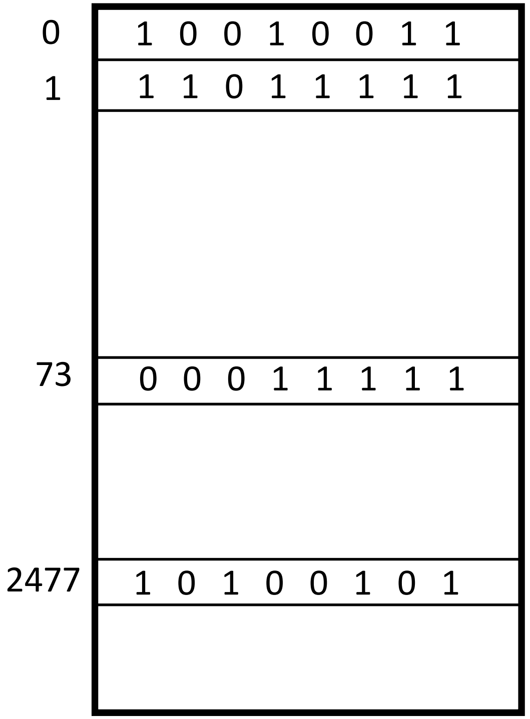


0	1 0 0 1 0 0 1 1
1	1 1 0 1 1 1 1 1
73	0 0 0 1 1 1 1 1
2477	1 0 1 0 0 1 0 1

A byte-addressable memory

Memories – cont.

- Volatile vs non-volatile
  - Read-only memory (**ROM**) is a form of non-volatile memory.
  - Electrically erasable programmable read-only (**EEPROM**) and Flash memories are non-volatile and may be re-programmed.
    - Write times take significantly longer than read times.
  - Random Access Memory (**RAM**) is volatile.
  - ROM, EEPROM and flash memories may be randomly accessed, RAM is just the name commonly used to refer to volatile memory.

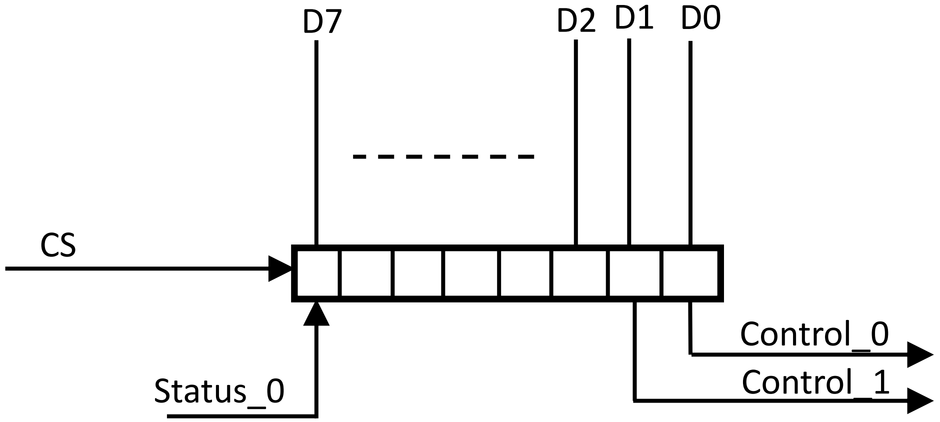


0	1 0 0 1 0 0 1 1
1	1 1 0 1 1 1 1 1
73	0 0 0 1 1 1 1 1
2477	1 0 1 0 0 1 0 1

A byte-addressable memory

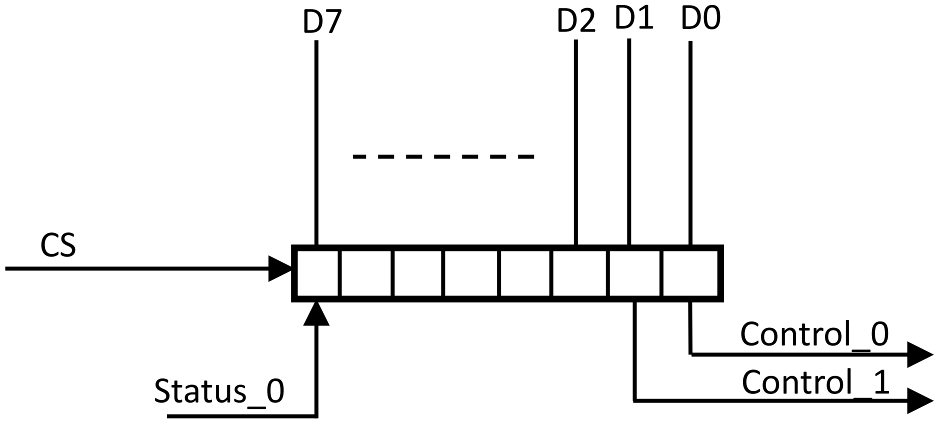
Registers

- Similar to memories in that they store information, but with an additional feature:
  - The information **<u>feeds to some external logic function</u>**.
- As a result, a register bit may be used as:
  - An output/control bit that controls some digital function, or
  - An input/status bit that reflects the status of another digital function
- <u>Status bit example:</u> PACKET_XFER_COMPLETE, SENSOR_TRIGGERED, etc.
- <u>Control bit example:</u> PACKET_SEND, MOTOR_EN, MOTOR_ROTATE_RIGHT, etc.



Registers – cont.

- **<u>A CPU register</u>** is inside the CPU and accessible as an operand in the machine instruction
- **<u>A device register</u>** is most commonly memory-mapped to some address within the CPU’s address space (aka memory space);
    - Thus, **each register has a single unique memory address** through which you can access (read/write) that register.

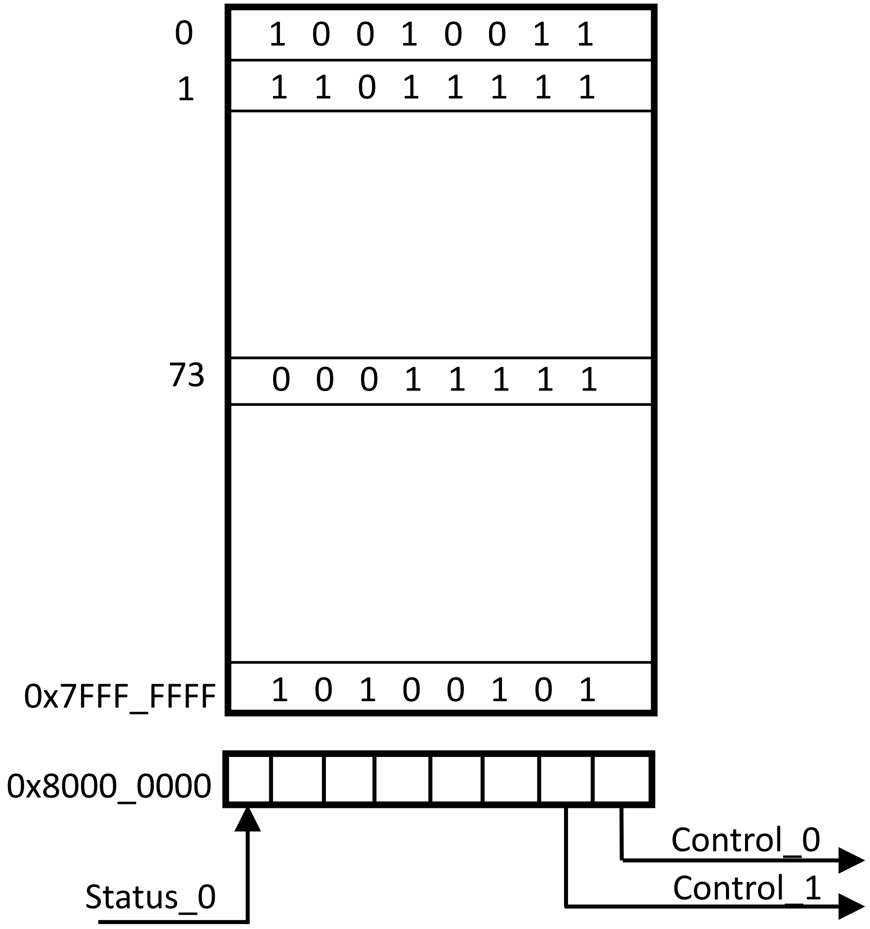


registers are not located inside of any memory though

Registers – Example

- A 32-bit system with a 2GB memory, and a single control register.

- How do we use “polling” to wait for a status bit to be asserted (logic 1 in this case).
  - We need to create a loop that checks the value of “Status_0” bit, which is bit 7 in the diagram.
  - Assume the register is located at memory-mapped address 0x80000000



```
unsigned char status;
volatile unsigned char *addr = 0x80000000;
do {
    status = *addr;
} while ((status & 0x80) == 0); // Stay in the loop till condition is false
```

I/O Operation

- I/O devices and the CPU can operate or execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer (usually memory mapped)
- CPU, and thus software, communicate with devices by:
    - Writing to control registers (to instruct the controller to do an operation)
    - Reading from status registers (to determine if the operation has completed)
    - Moving data from main memory to local buffers and vise versa.
- The CPU waits till the operation is done via one of two methods:
    - **Polling:** The CPU continuously reads the status till it detects operation is complete.
    - **Interrupt-driven-I/O:** CPU moves on and executes other tasks. Device controller informs CPU of completion by asserting an interrupt pin.
- A **Device Driver** (part of the kernel software) exists for each device controller to manage I/O
    - Provides uniform interface between controller and rest of the kernel

Storage Definitions and Notation Review

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes.
A **kilobyte**, or **KB**, is 1,024 bytes
a **megabyte**, or **MB**, is $1{,}024^2$ bytes
a **gigabyte**, or **GB**, is $1{,}024^3$ bytes
a **terabyte**, or **TB**, is $1{,}024^4$ bytes
a **petabyte**, or **PB**, is $1{,}024^5$ bytes
an **exabyte**, or **EB**, is $1{,}024^6$ bytes

Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

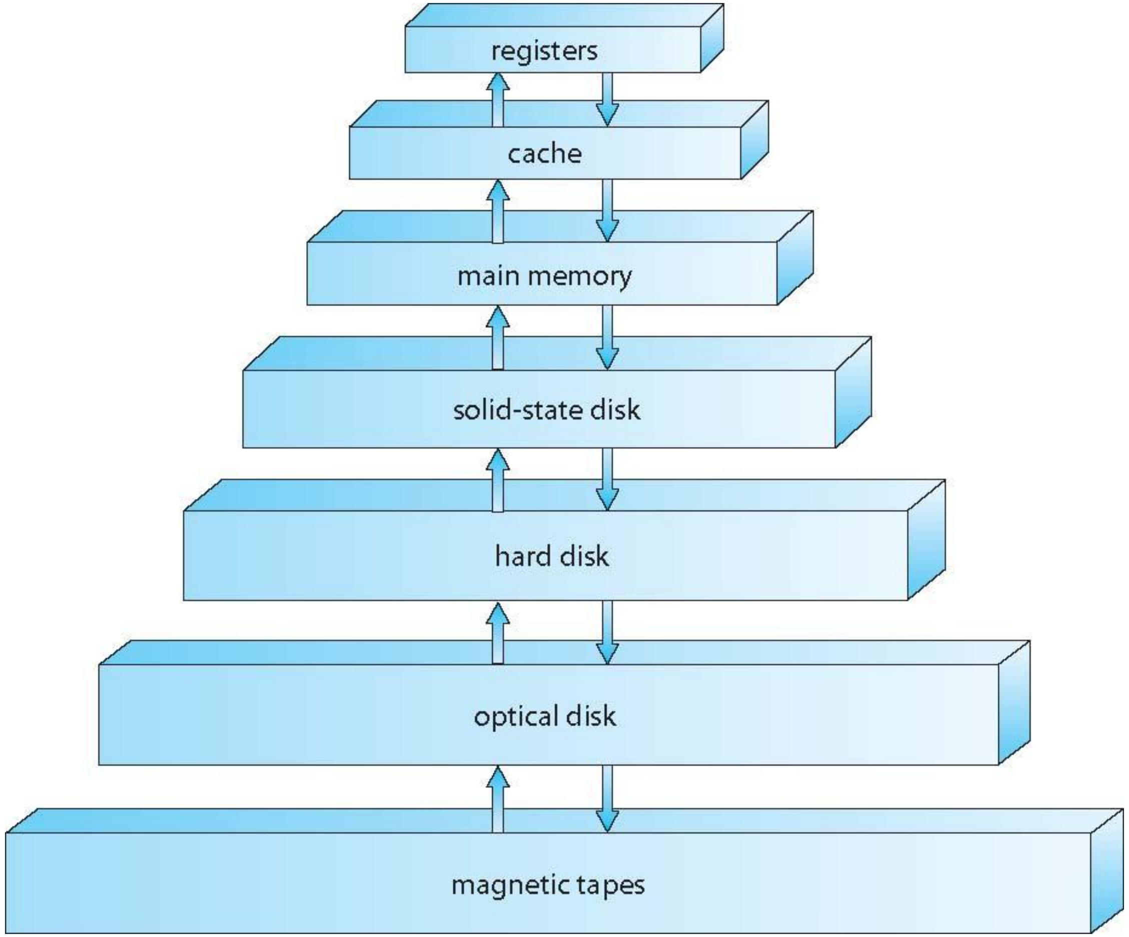
Storage Structure

- Main memory – is the only storage media that the CPU can access **directly**:
  - RAM: random access memory, volatile
  - ROM: read only memory, non-volatile (e.g. ROM, EEPROM or flash)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity, however, the CPU can only access this memory **indirectly** via a device controller (using its control/status and data interfaces)
  - **Hard disks** – rigid metal or glass platters covered with magnetic recording material.
    - Disk surface is logically divided into tracks, which are subdivided into sectors.
  - **Solid-state disks** – faster than hard disks, also nonvolatile
    - Becoming more popular

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost

  (usually, the larger the memory, the slower it is)
- **Caching** – As a concept, it means copying information into faster storage system;
  - Main memory can be viewed as a cache for secondary storage
  - CPU's internal cache memory is a cache for the main memory



Caching

- Important principle, performed at **many levels** in a computer (in hardware, operating system, software)
- Information in use is copied from slower to faster storage temporarily
  - Faster storage (cache) checked first to determine if information is there
    - If it is, information used directly from the cache (**cache hit**).
    - If not, data copied to cache and used there (**cache miss**).
- Why is it advantageous to use cache?
- Cache management is an important design problem
  - Cache size (affects speed + cost)
  - Replacement policy (e.g. LIFO, LRU, etc.)

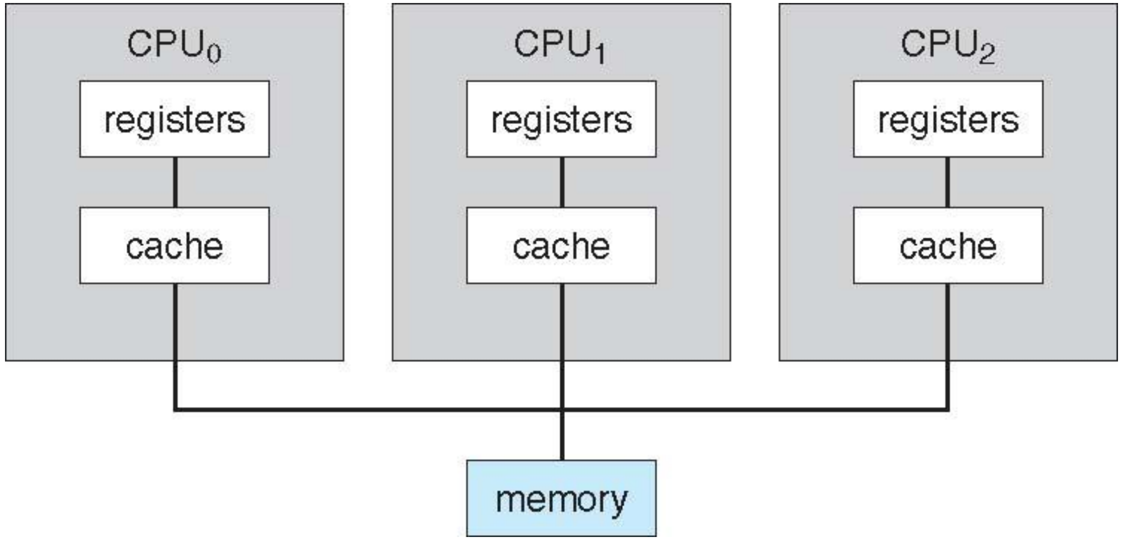
Computer-System Architecture

- Many systems use a **single general-purpose processor**
  - Also often referred to as **application processor**.
  - Most systems have special-purpose processors as well (e.g. a GPU or a DSP), but these do not make the system a multiprocessor system.

- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput**
    2. **Economy of scale**
    3. **Increased reliability** – graceful degradation or fault tolerance

Computer-System Architecture – cont.

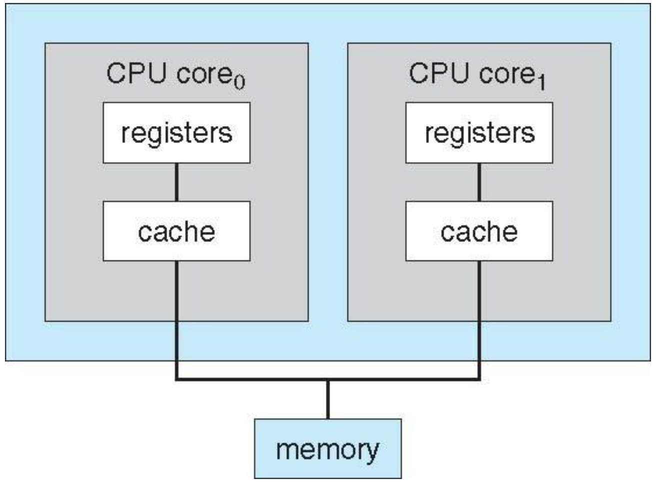
- **Multiprocessors** - Two types:
  1. **Asymmetric Multiprocessing** – processors are not treated as equal.
     - Processors may be dedicated to specific tasks
     - e.g. boss and worker processors
  2. **Symmetric Multiprocessing (SMP)** – all processors are treated equally
     - Single instance of the OS.
     - Each processor is **capable** of performing any task, such as handling interrupts, running the OS kernel, running applications, etc.



Symmetric multiprocessors

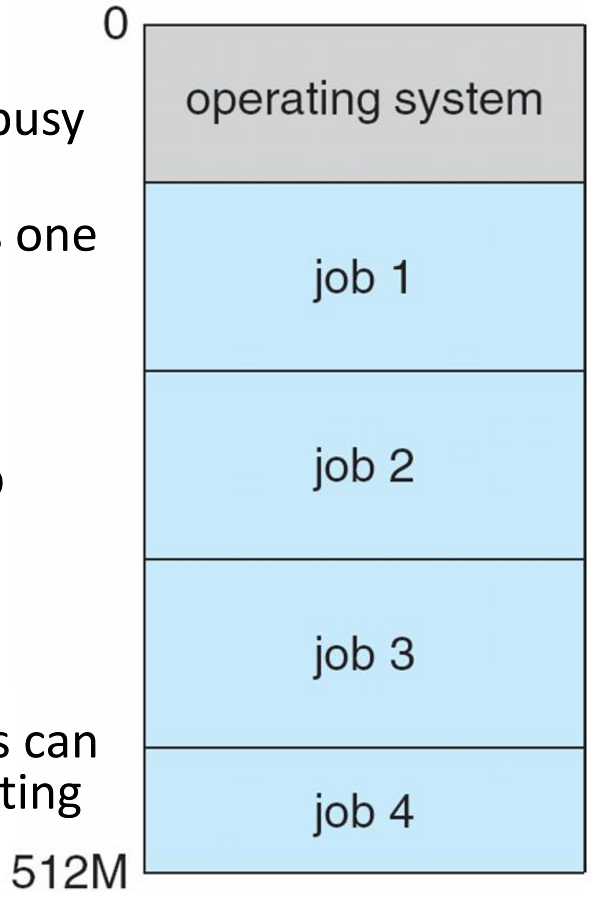
Multi-Core Designs

- A **multicore** system may have multiple cores in a single chip, and is thus a multiprocessor system.
- Systems may be built of multiple chips, each with multiple cores.



Operating System types

- **Multiprogramming batch systems**
  - A bit historical
  - Needed for efficiency: Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
  - Typically used **non-preemptive = cooperative** multitasking



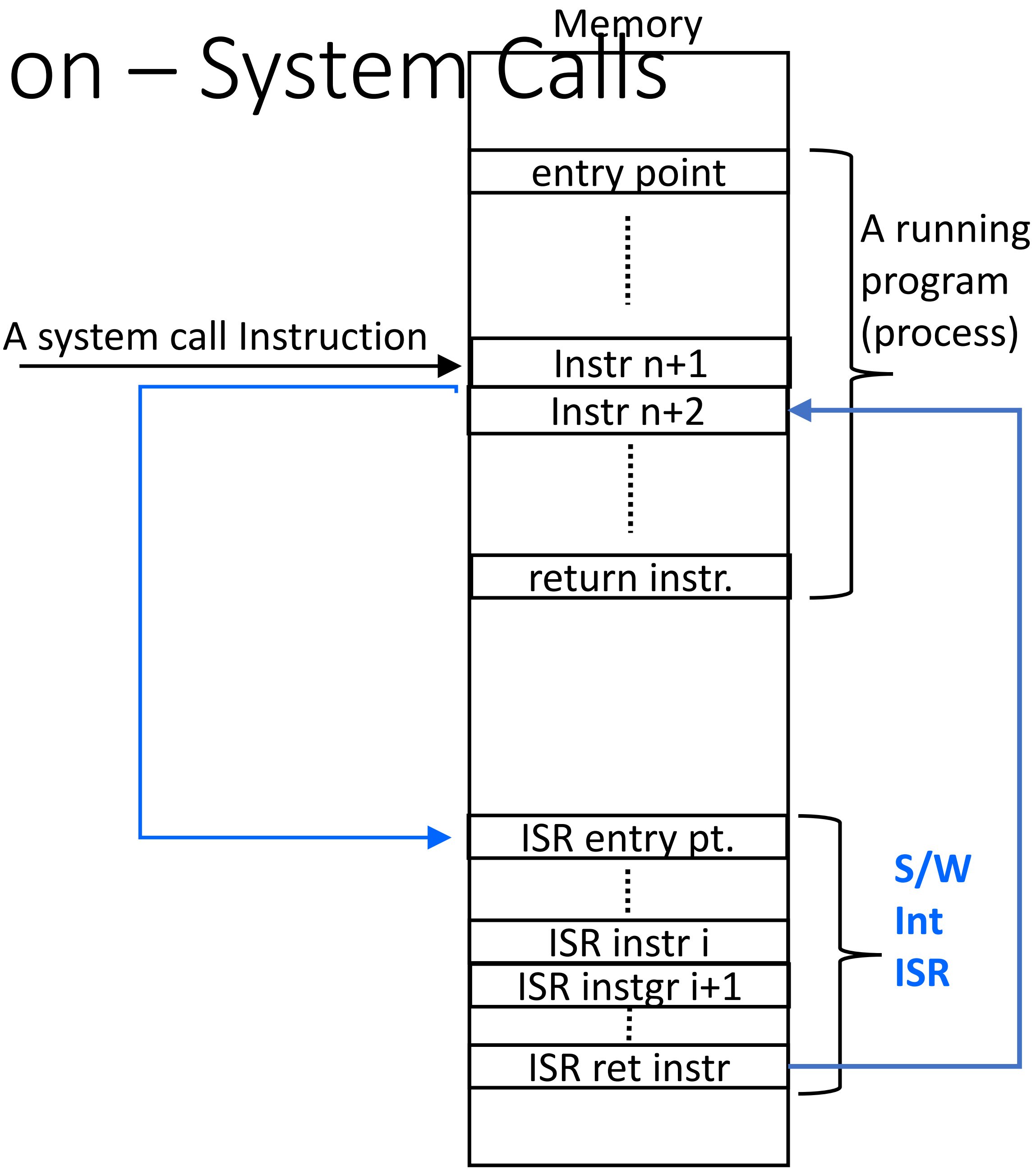
- **Timesharing/interactive systems**
  - Logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing (**response time** should be < 1 second) -> **preemptive**
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling** selects one of them

Operating-System Invocation

- The operating system is invoked via interrupts. Invocation may be due to either:
  - Hardware interrupts by one of the devices.
    - A **timer** interrupt is a hardware interrupt caused by an on-chip timer, and is used to preempt applications and invoke the OS kernel on regular intervals (called **OS tick**)
      - The interrupt interval is usually 1 to 50 mS.
    - Interrupts **from device controllers**.
  - Software interrupts**:**
    - Operating system services are requested using the **trap** instruction (aka **system request)**, which is a software interrupt.
    - Illegal operations:
      - Software error (e.g., division by zero) causes an exception
      - Illegal instruction or illegal access also cause exceptions.
- Note that the operating system may also be running its own threads (kernel threads).
  - The **OS scheduler** thus schedules **user jobs/processes** AND **kernel threads**

Operating-System Invocation – System Calls

- If a program wishes to make a system call, it embeds a "software trap" or "system call" instruction in its machine code.
  - SWI – ARM CPUs (or **svc** in newer ARM CPU's)
  - INT – intel CPUs
- When the CPU encounters such instruction, it pauses the running process and starts executing the corresponding ISR
  - CPU looks up the Interrupt Vector Table (IVT) and finds the entry point for the corresponding ISR **AND** switches to kernel mode
- When the system call handler finishes, it returns from interrupt (switching back to user mode) then resumes the previously interrupted/paused process.

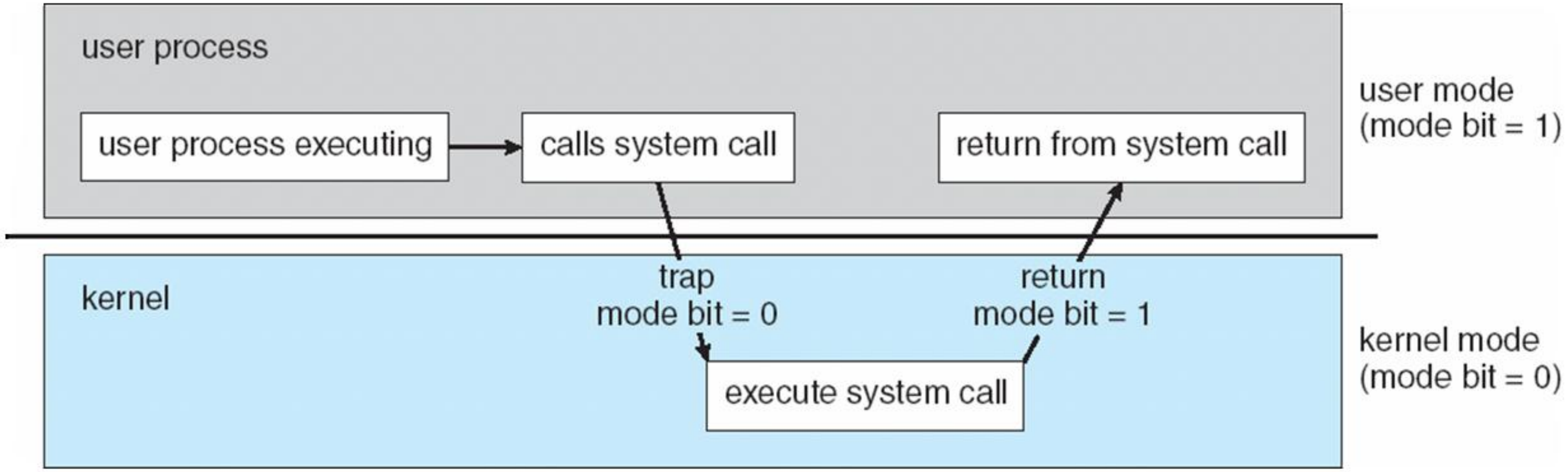


Operating-System Invocation – System Calls

- **Dual-mode** operation of CPU allows OS to **protect itself** and other system components
  - **Mode bit** provided by the CPU hardware determines whether the CPU is in **User mode** or **kernel mode.**

    Provides ability to distinguish when system is running user code or kernel code

    - Some **instructions** designated as **privileged**, only executable in kernel mode
    - Some **memory locations** may be configured to be only accessible in kernel mode.
  - **System call** (using the "software trap" instruction) changes mode to kernel-mode.
  - **Return** from a system call resets the mode bit back to user-mode



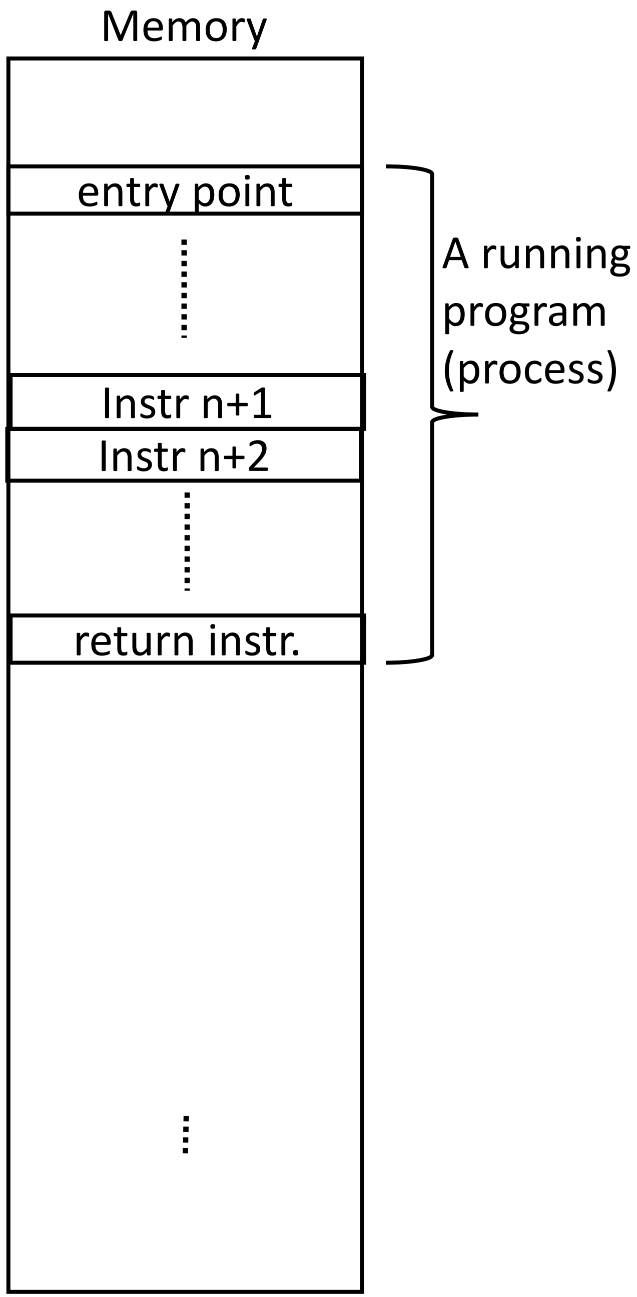
Transition from user mode to kernel mode

Operating-System Invocation – System Calls (cont.)

- Increasingly CPUs support multi-mode operations:
  - Privileged/kernel modes: e.g. interrupts, kernel threads, etc.
  - Non-privileged/user modes: e.g. user threads/processes, **virtual machine manager** (**VMM**) mode for guest **VMs**, etc.

Operating-System Invocation – Timer interrupts

- A process may decide to run for a long period of time
  - e.g. an MRI image processing process may take several minutes of continuous processing with requesting any I/O operations
- Other – possibly interactive - processes need to run every < 1 sec

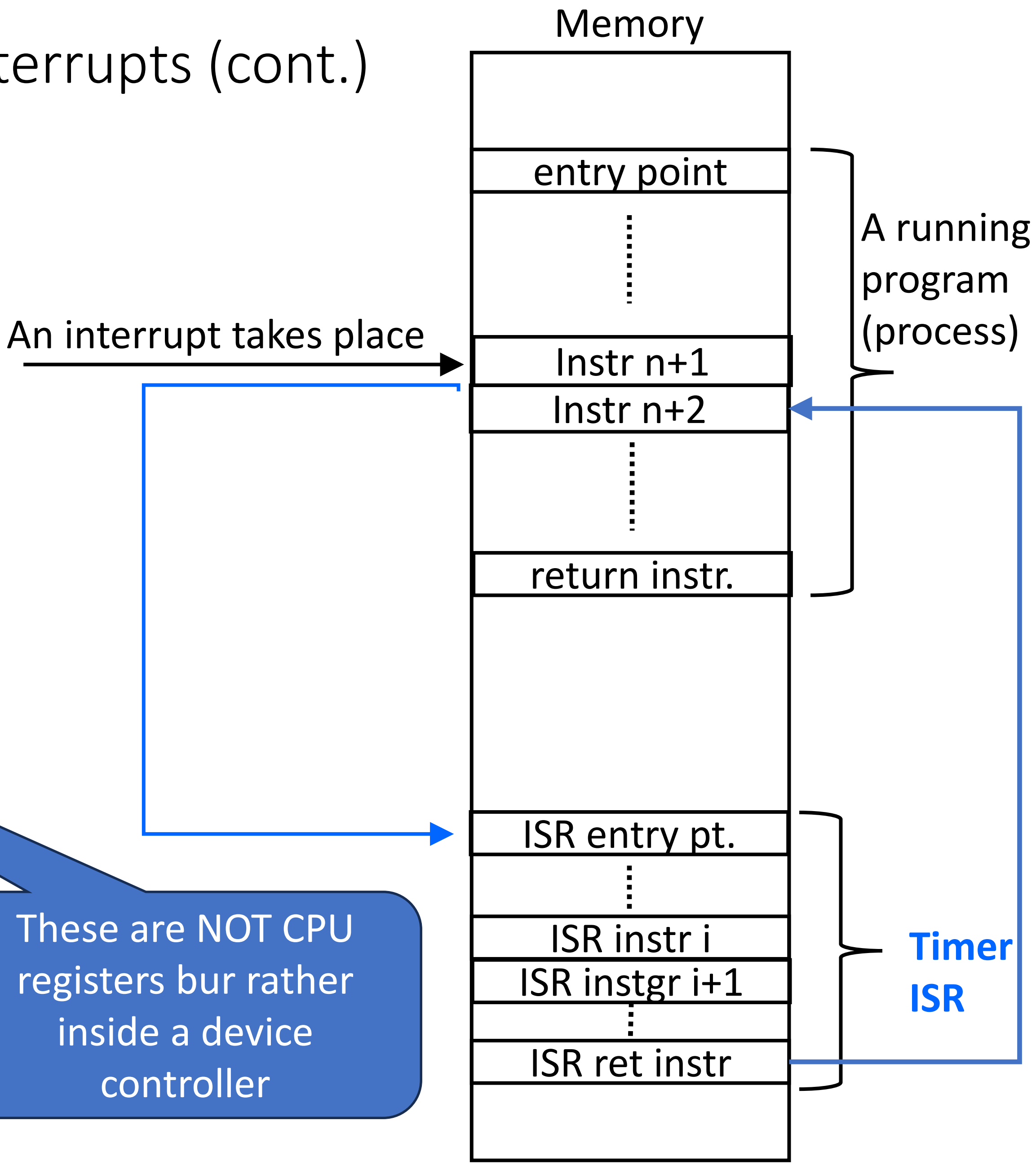


A process may run for a very long time without requesting I/O operations

Operating-System Invocation – Timer Interrupts (cont.)

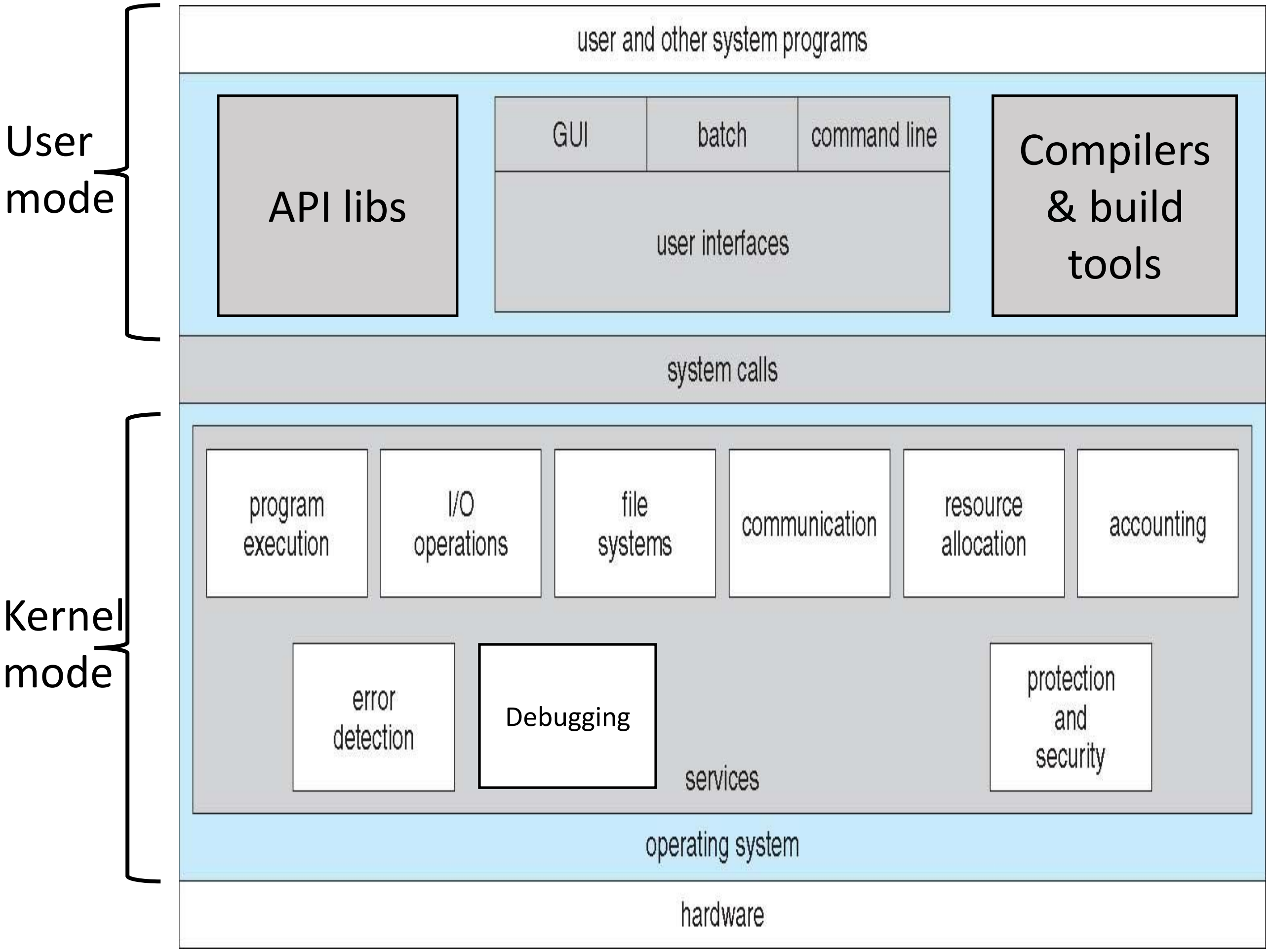
- **Solution:** Timer to prevent infinite loop / process hogging CPU resources (**→ preemptive multitasking**)
  - Timer is set to interrupt the CPU after some time period (e.g. 1 – 50 ms)
    - The interrupt is handled by the OS kernel.
    - The timer registers are memory-mapped to a memory area that can be accessed only in privileged mode.
  - The kernel sets up the timer for the next interrupt (timer tick) before scheduling a process to run.
    - This is in order to regain control or **preempt** a running process that exceeds its allotted time

These are NOT CPU registers bur rather inside a device controller



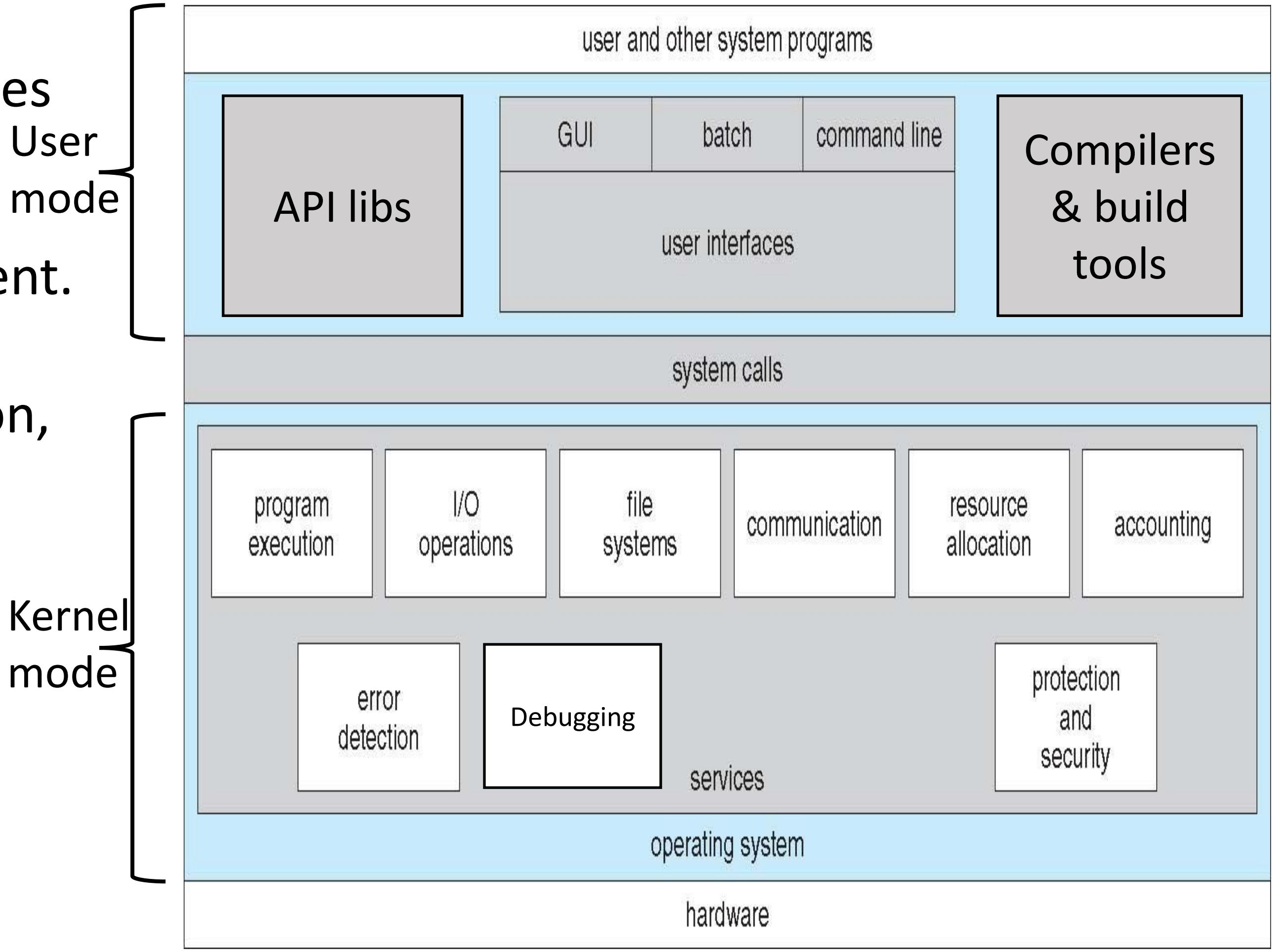
2.1 Operating System Services

- One of the operating systems’ main tasks is to provide **an environment and services** for application programs to run:
  - **API Libraries** – e.g. libc or pthreads
  - **Compilers and build tools**
  - **User interface** - Almost all operating systems have a user interface (**UI**). Varies between:
    - **Command-Line** (**CLI**),
    - **Graphics User Interface** (**GUI**),
    - **Batch**
  - **Program execution** - The system must be able to **load** a program into memory and to run that program and also **end** its execution (normally or abnormally - indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device



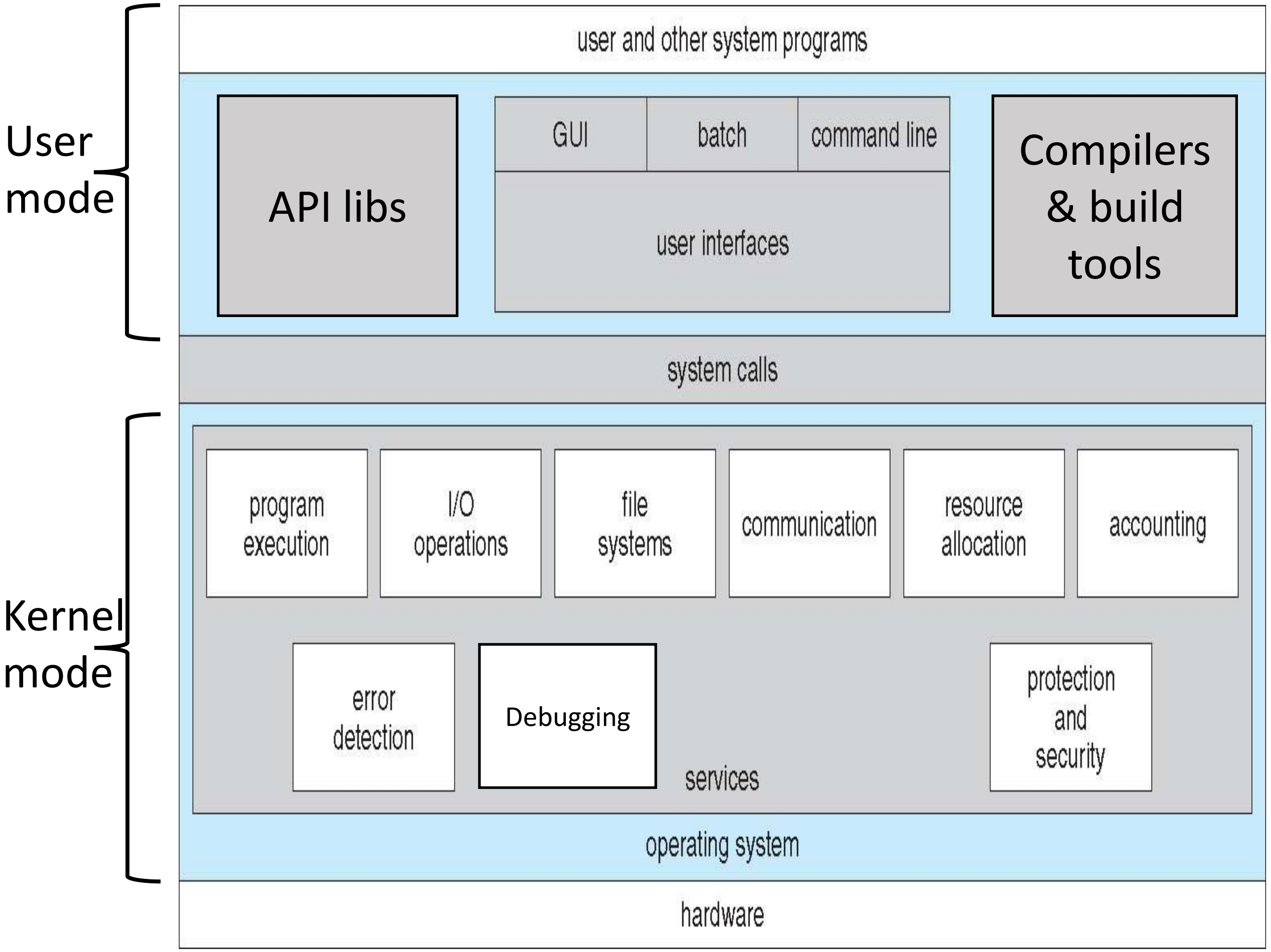
Operating System Services (Cont.)

- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Interprocess Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets or messages moved by the OS)



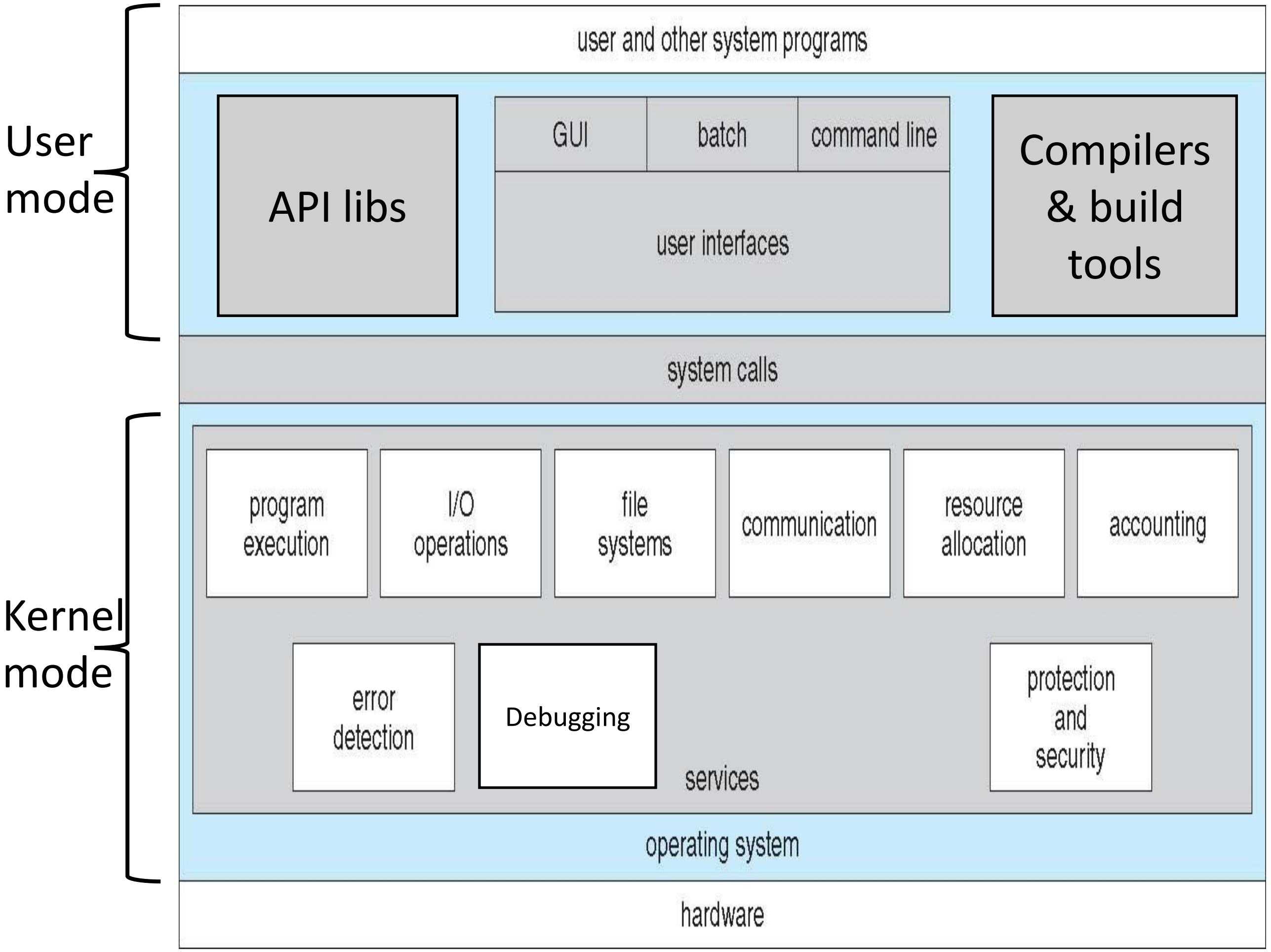
Operating System Services (Cont.)

- **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU, memory hardware, or I/O devices.
    - May alternatively occur in user programs or even the kernel.
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
- **Debugging:** OS provides debugging facilities which can greatly enhance the user’s and programmer’s abilities to efficiently use the system.
    - In Linux, the kernel’s debug component provides this facility using the “**ptrace**” and “**debug**” system calls.
    - User mode debuggers (e.g. **gdb**) use this kernel facility to debug user programs.



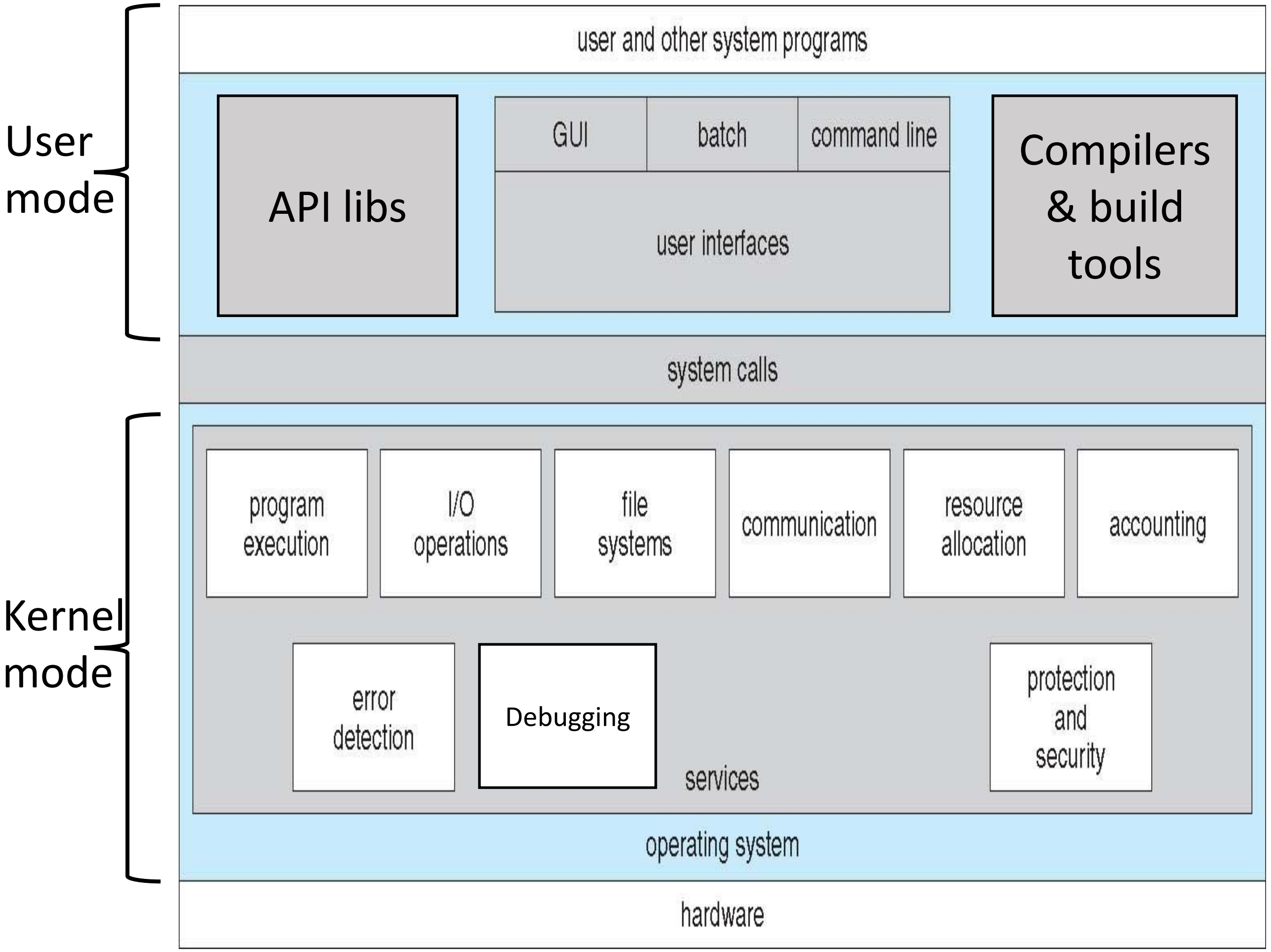
Operating System Services (Cont.)

- The second main goal of an OS is to provide functions that ensure the efficient and secure operation of the system:
  - **Resource allocation -** When multiple jobs (or processes) are running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Accounting -** To keep track of which **users/processes** use how much and what kinds of computer **resources** (may collect statistics useful for researchers or system admins for detecting system misuse or intrusion).



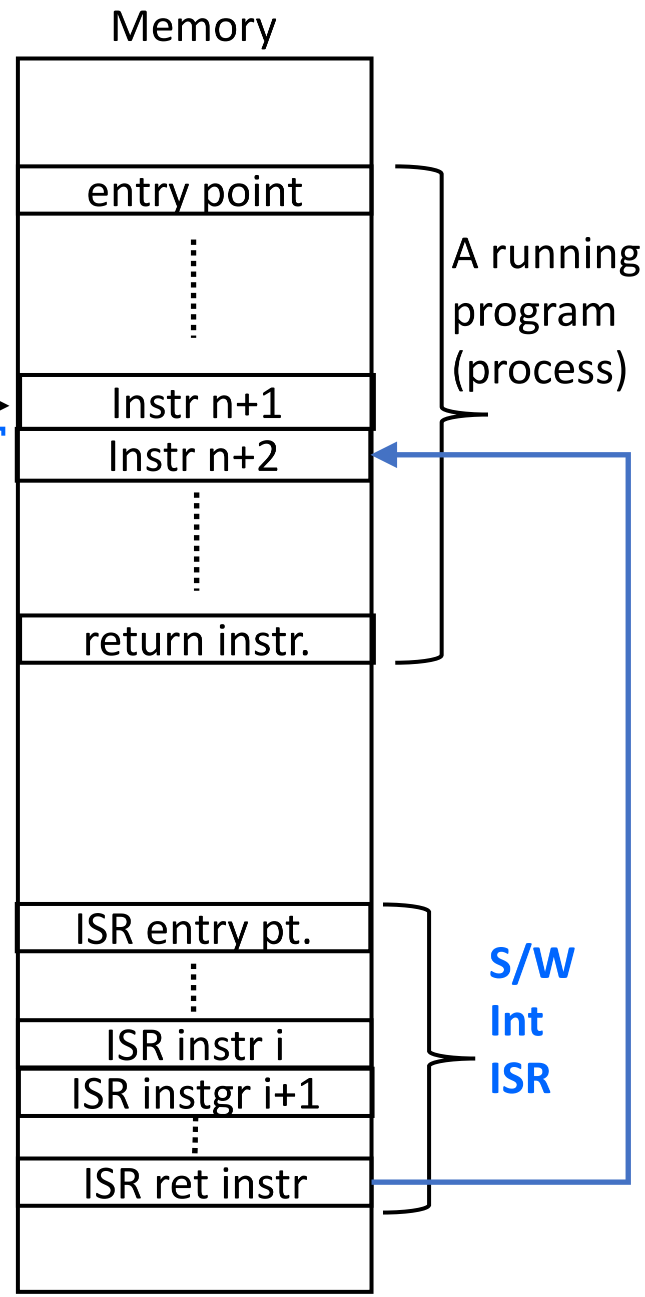
Operating System Services (Cont.)

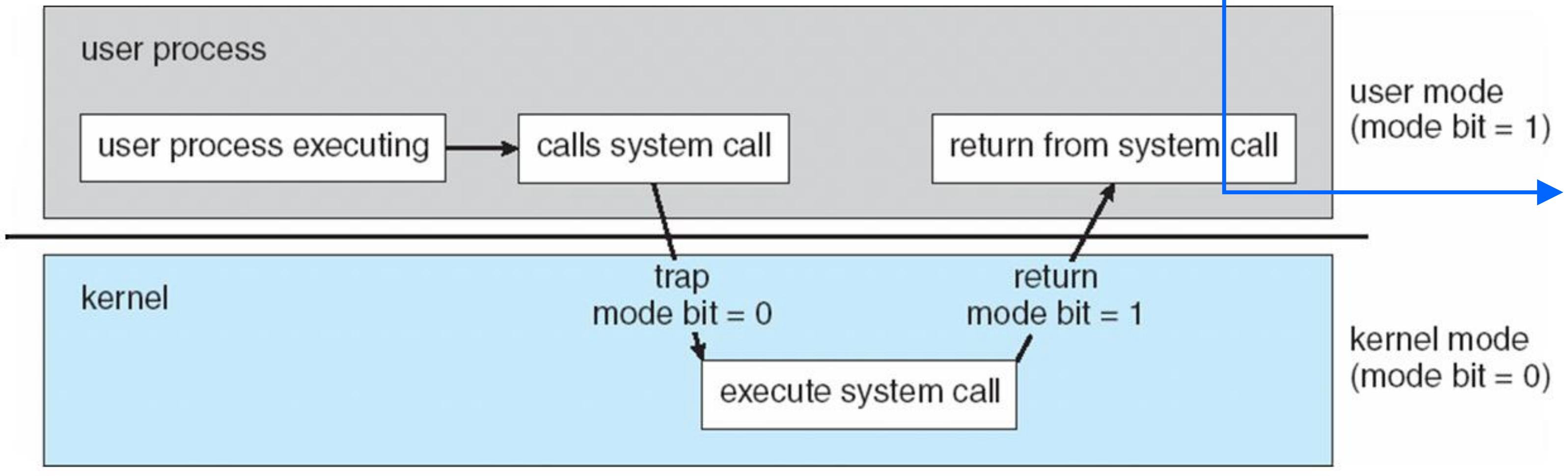
- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
    - Extends to defending external I/O devices from invalid access attempts
  - **Security** of the system from outsiders requires user authentication.



2.3 System Calls

- User-mode program's interface to the services provided by the OS
  - User mode programs **cannot** directly call driver or operating system functions. Instead they need to use system calls.
  - System calls may also be referred to as “supervisor call”, “trap” or sometimes “software interrupt”
  - The application program uses a special machine instruction to perform a system call:
    - SWI – ARM CPUs (or **svc** in newer ARM CPU’s)
    - INT – intel CPUs

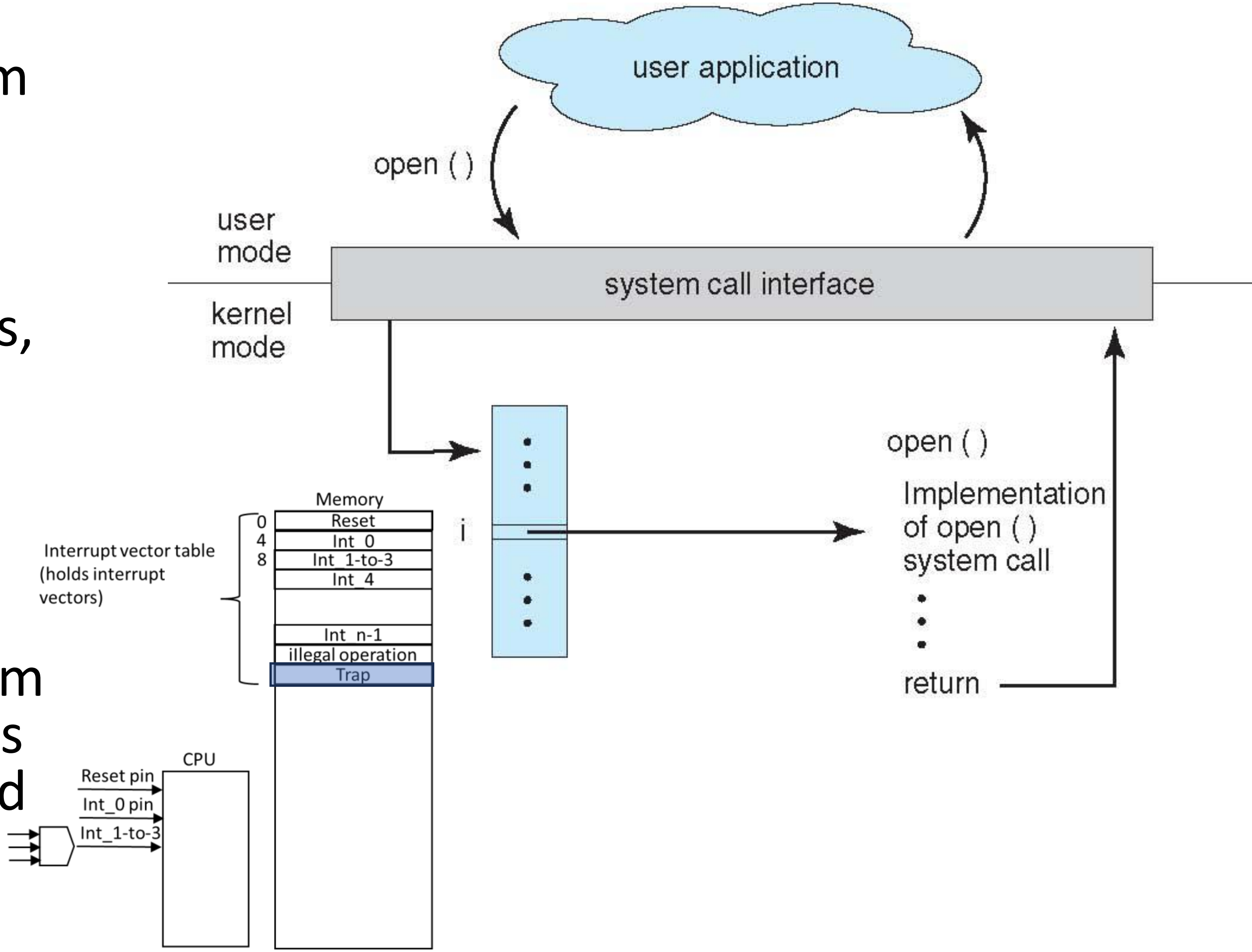




Transition from user mode to kernel mode

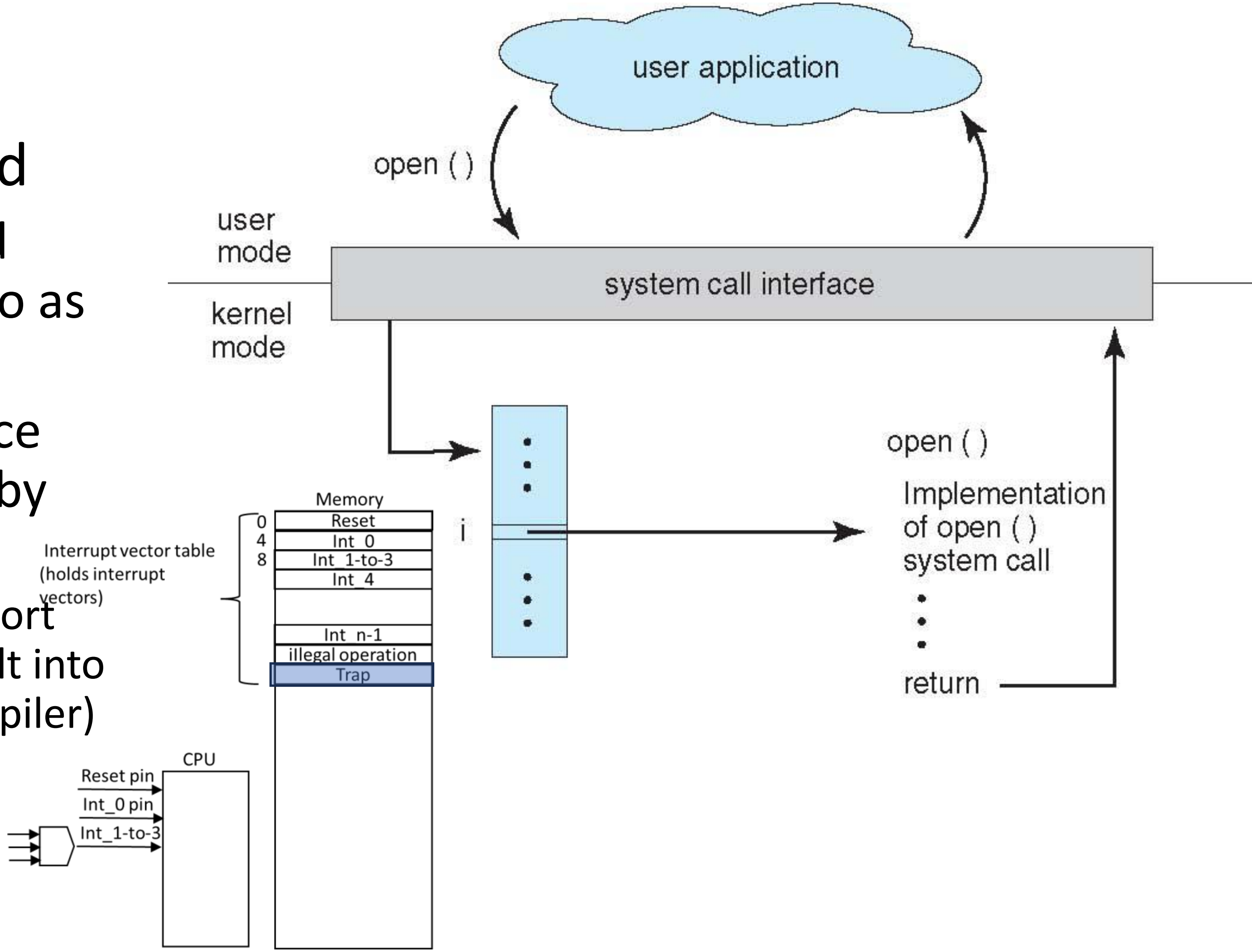
System Call Implementation

- Typically, a number associated with each system call
- **System-call interface** maintains a table indexed according to these numbers, the **system call table**:
  - Contains pointers to subroutines that implement system calls, e.g. “open”
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values



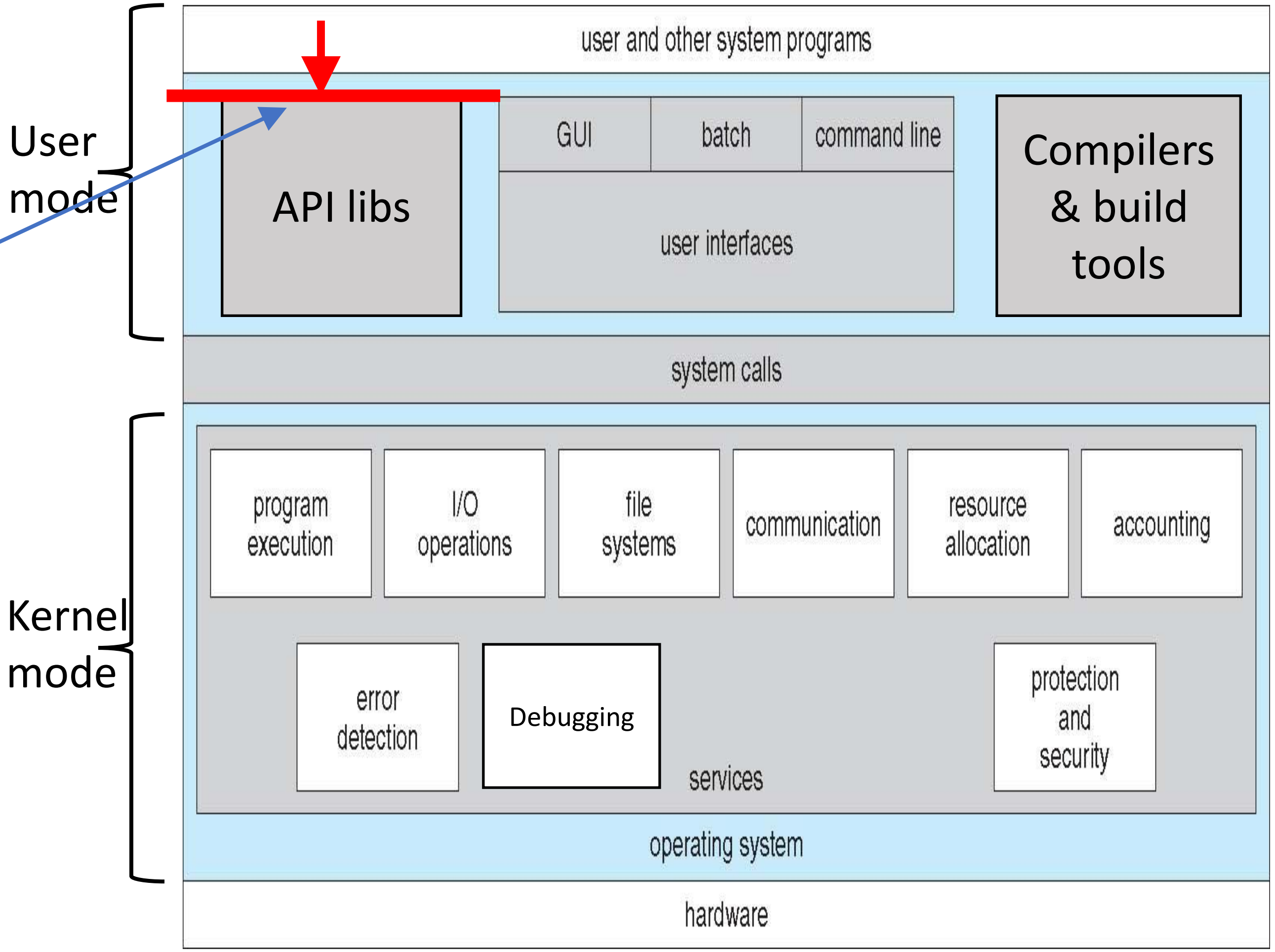
System Call Implementation (cont.)

- The caller need not know anything about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result of the call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)



System Calls Implementation – cont.

- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) libraries (e.g. libc for unix/linux) rather than direct system call use



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

Note that unless otherwise stated, the system-call names used throughout this course are generic

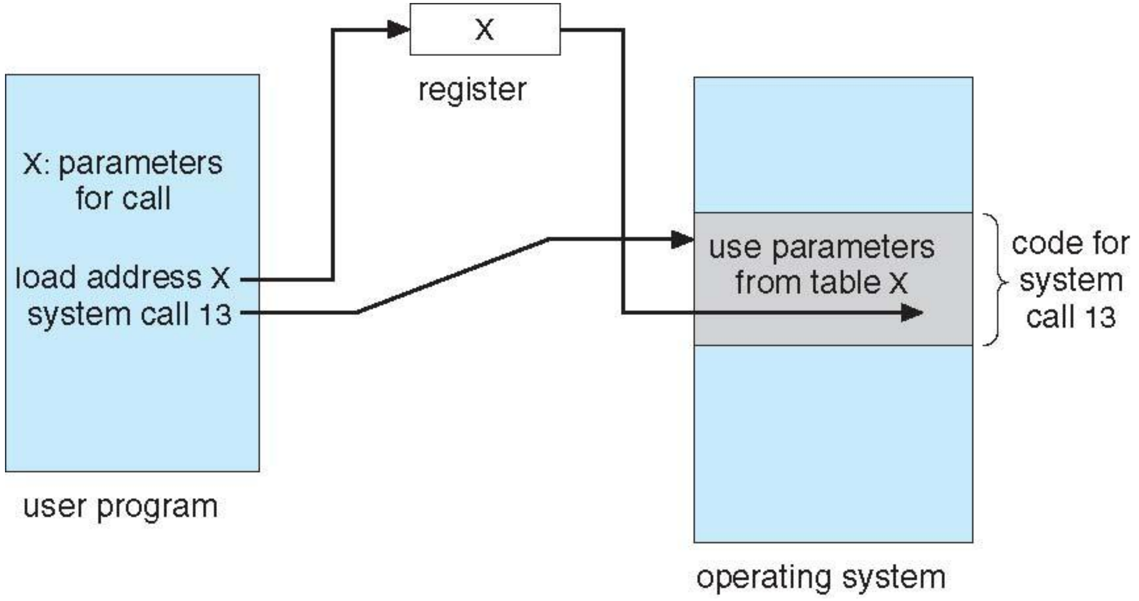
System Call Parameter Passing

- Often, we need to pass parameters to a system call:
  - Exact type and number vary according to system call routine called
- Three general methods used to pass parameters to the OS
  1. Simplest: pass the parameters in registers
     - Disadvantage: In some cases, may be more parameters than registers

System Call Parameter Passing – cont.

2. Parameters stored in a block in memory, and address of block passed as a parameter in a register
   - This approach taken by Linux and Solaris (a Unix system)
3. Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system's kernel

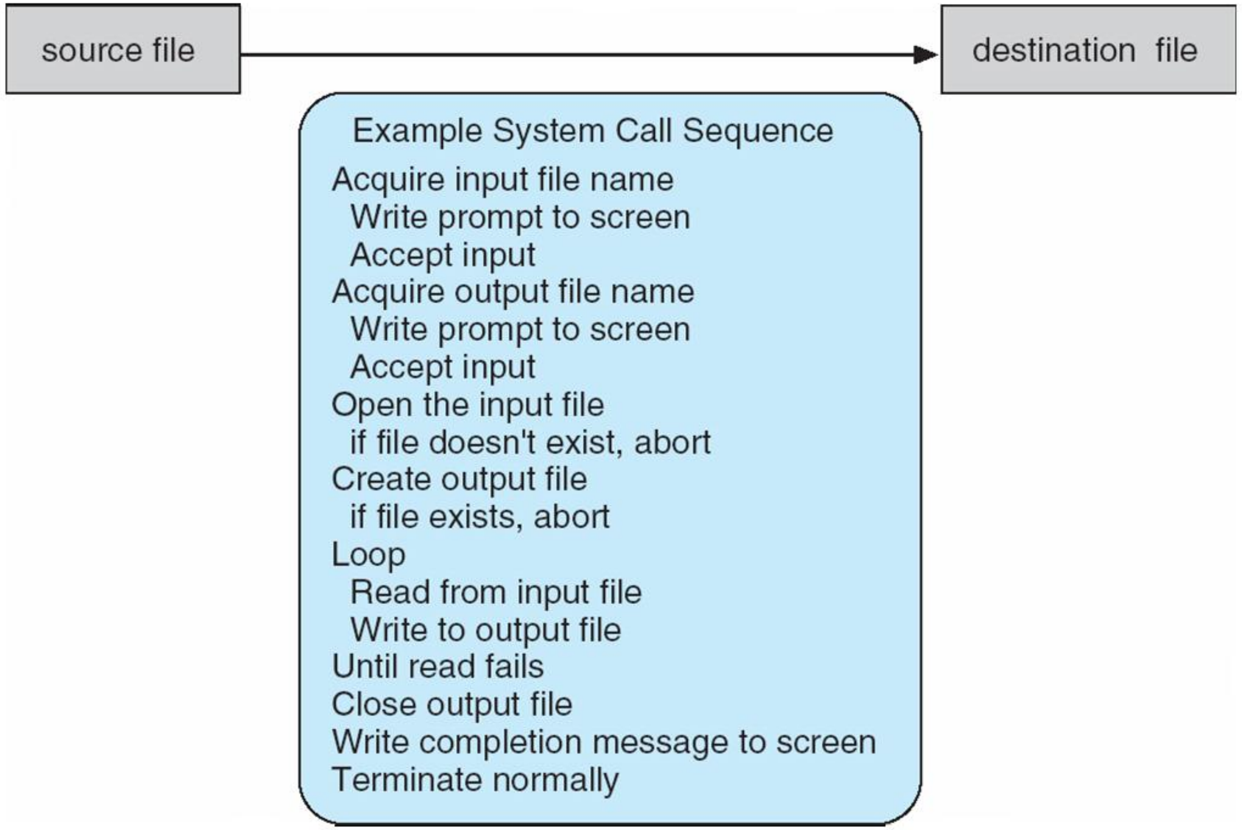
- Unlike the registers method, the block and stack methods do not limit the number or length of parameters being passed



Parameter Passing via memory block/table

Example of System Calls

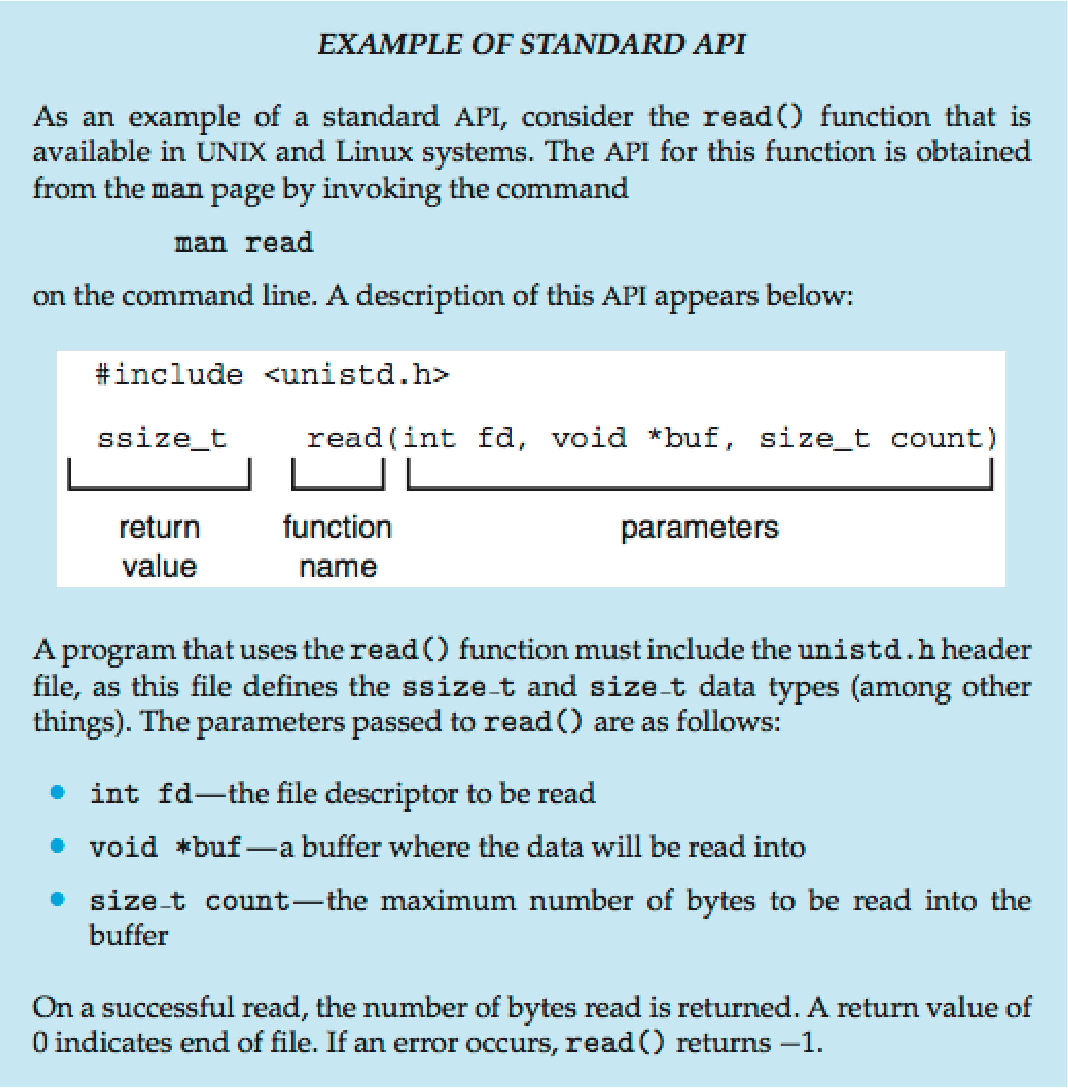
- The example below demonstrates the steps needed to copy the contents of one file to another file → A sequence of system calls results



Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Example of POSIX Standard API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value

function name

parameters

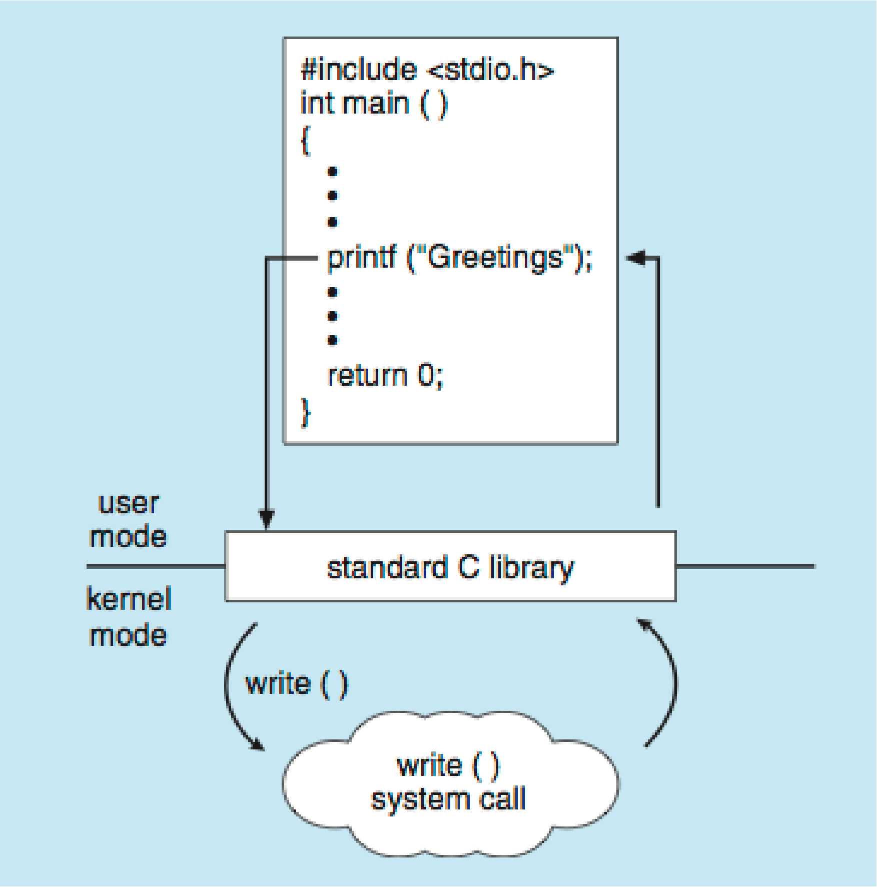
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Example using standard C library

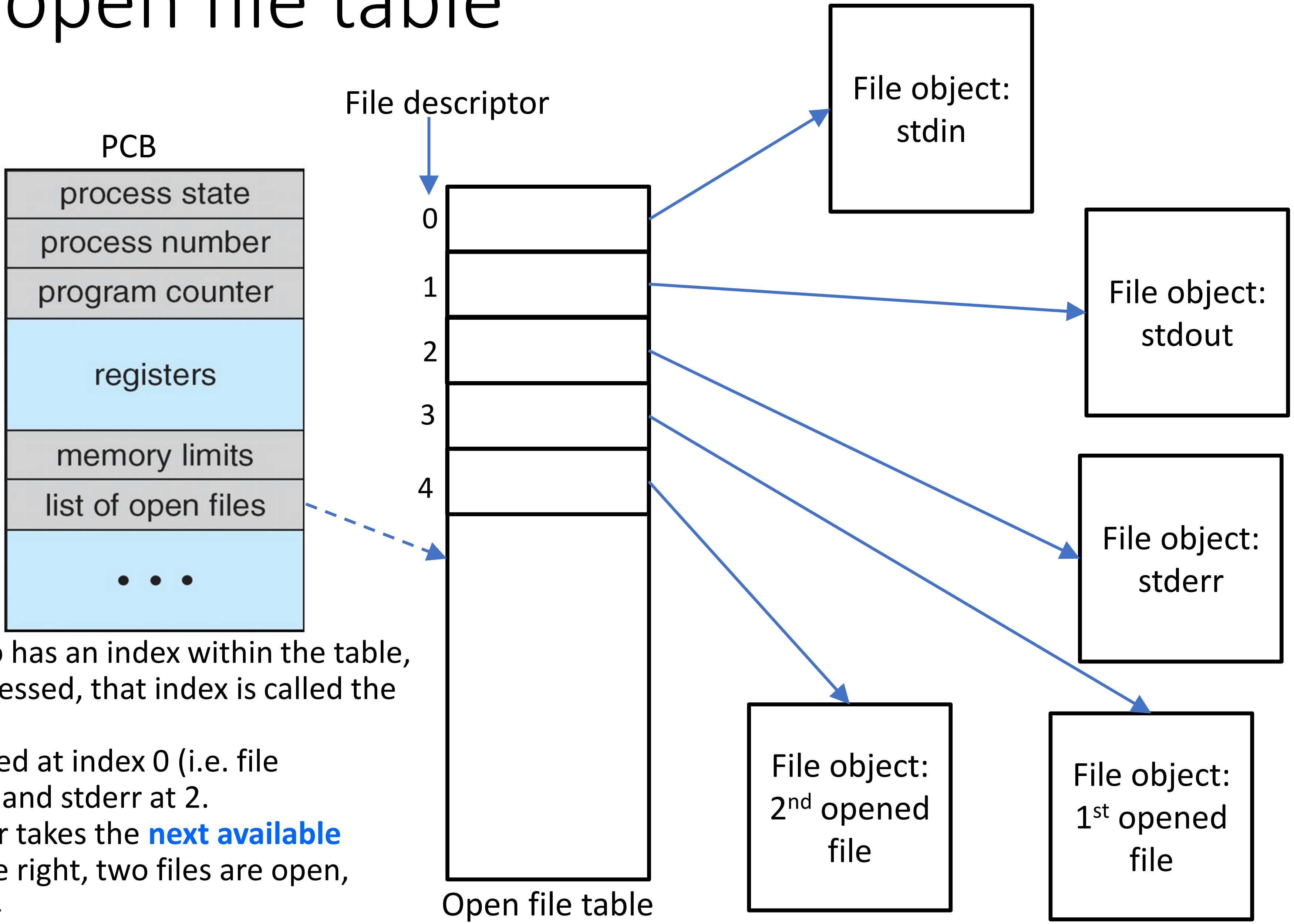
- C program invoking printf() library call, which calls write() system call



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

Per-process open file table

- In Linux, every process is represented by an object, called the process control block (PCB)
- The PCB contains a **pointer that eventually points to the open file table**, which contains info about open files.
- Each entry within the open file table has some info about the open file + a pointer that eventually points to an object representing that file (**File object**)



- Each entry in that table also has an index within the table, through which it can be accessed, that index is called the **file descriptor**.
- The pointer to stdin is located at index 0 (i.e. file descriptor = 0), stdout at 1, and stderr at 2.
- Any file you open thereafter takes the **next available index** (in the example to the right, two files are open, with file descriptors 3 and 4