UNIVERSITY OF PORTSMOUTH

Operating Systems and Internetworking

OSI – M30233 (OS Theme)

Week 5- Processes and Scheduling

Dr Tamer Elboghdadly

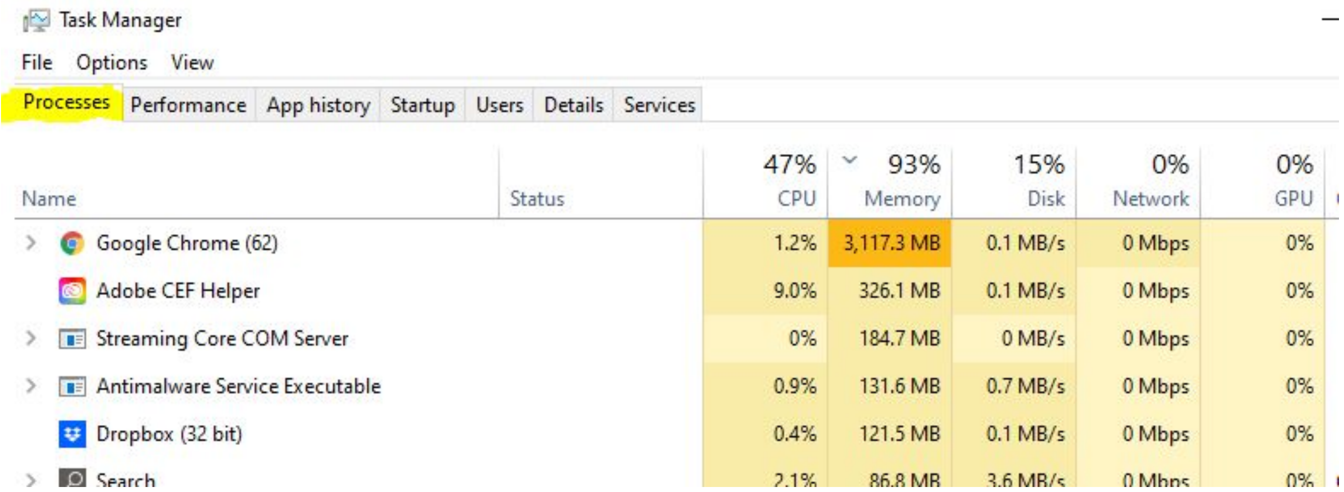
Processes and Scheduling

- **Plan:**
  - Components of processes
  - Multitasking basics
    - Process state
    - Context switching and scheduling
  - System calls

Processes

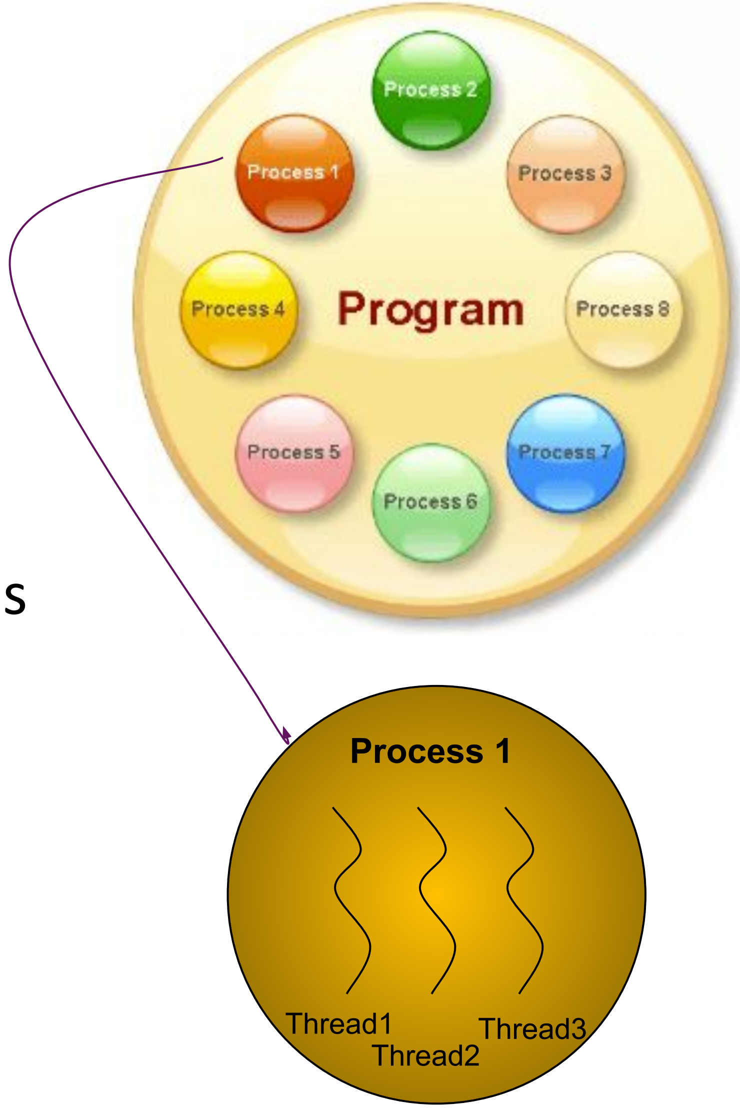
Process Concept

- A *process* is a *program in execution*.
  - On a PC, each program in each window will be running in a different process.
  - In MS Windows, for example, Task Manager will identify many other invisible processes, running in the background.



Processes vs Programs

- A process is *not* a program:
  - A program is a **passive** entity – a sequence of instructions.
  - A process is an **active** entity – it is "doing things". It is an executing part of a program.
  - Several copies of the same program may be running concurrently – each in a distinct process.
- A process also contains *execution state*.



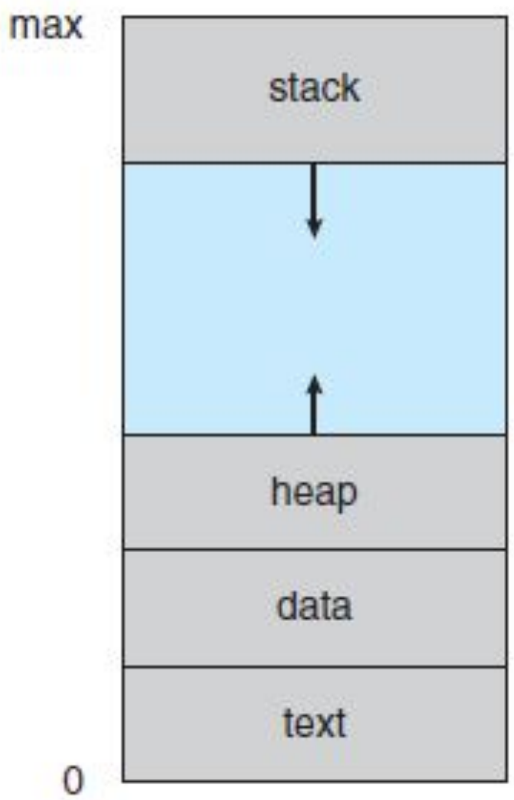
Read more here or here

UNIVERSITY OF PORTSMOUTH

Memory layout of a Process

- The memory layout of a process is typically divided into multiple sections, Include:

  - *Text Section:* the executable code
  - *Data Section:* contains **global and static** variables
  - *Heap:* memory that is dynamically allocated during program run time
  - *Stack:* temporary data storage when invoking functions (such as function parameters, return addresses, and **local** variables)



What Makes a Process?

- Execution state of a process includes:
  - *program counter* (point reached in programme), a *stack* and a *data section*

(A *thread* also has these, but inherits the data section from the process it belongs to.)

- The *address space* of a process is another of its defining properties - loosely, we may say:

Process = Thread(s) + Address Space

UNIVERSITY OF PORTSMOUTH

MULTITASKING

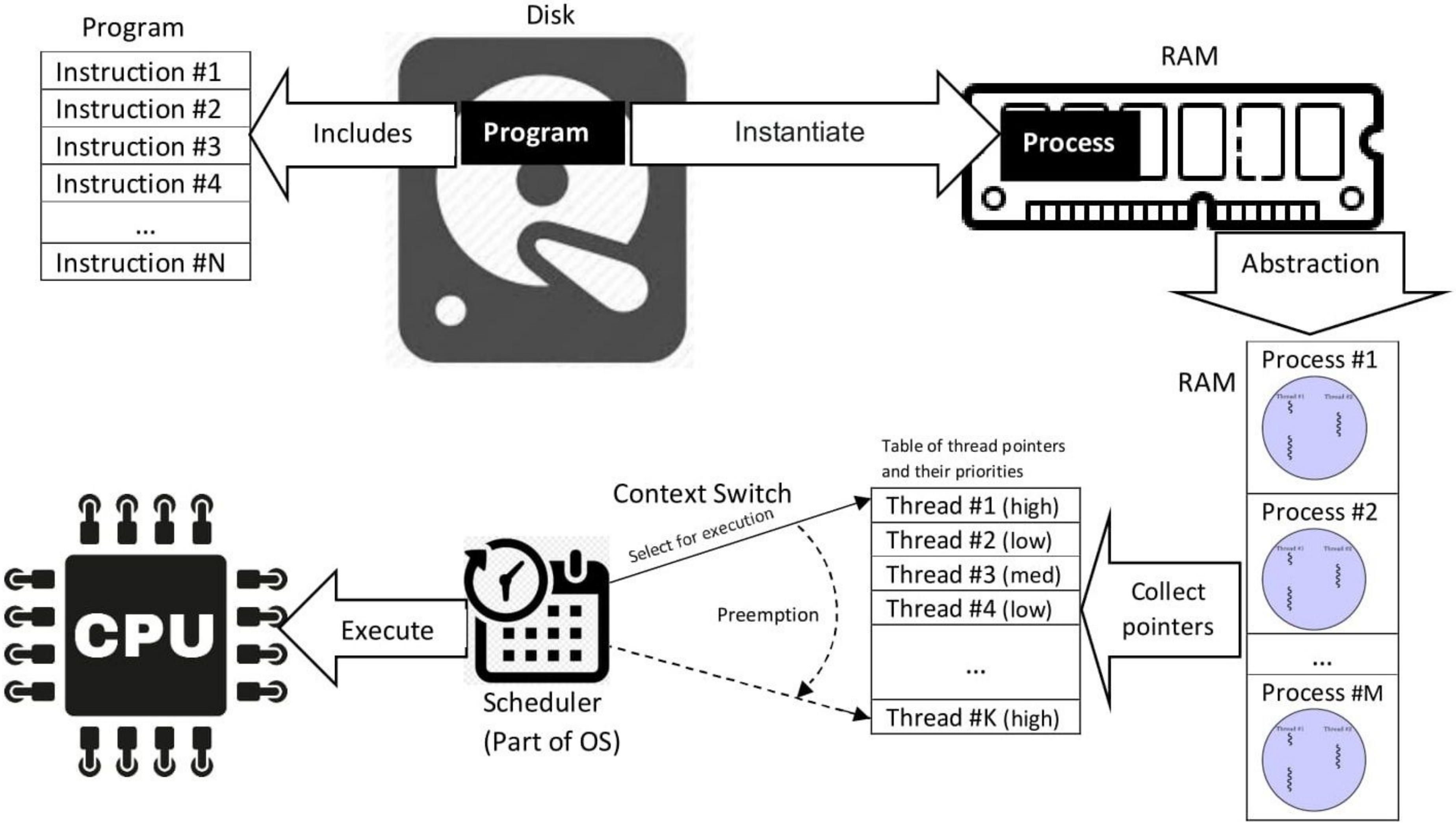
Multitasking

- The operating system is responsible for *sharing* the physical *CPU resource* between processes.
- It does this by *time-sharing*.
  - CPU is allocated *in turn* to active processes.
- *Context switching* between processes happens frequently enough that processes *appear* to run concurrently. *Context switching* is the process of storing the state of a process and switch the CPU to another process.
  - Maximum *quantum* of time a process runs before switching might be around 10ms.
  - Details depend on OS scheduling policy.

Time Sharing

- While one process is waiting for I/O, the CPU can be reallocated to another process that has work to do.
  - Improves utilization of CPU
- Thus, reasons why OS should **context switch** between processes include:
  - Process has been executing on CPU for allotted quantum - it is time to give another process a go.
  - Process needs to wait for I/O – relinquish CPU to another process.

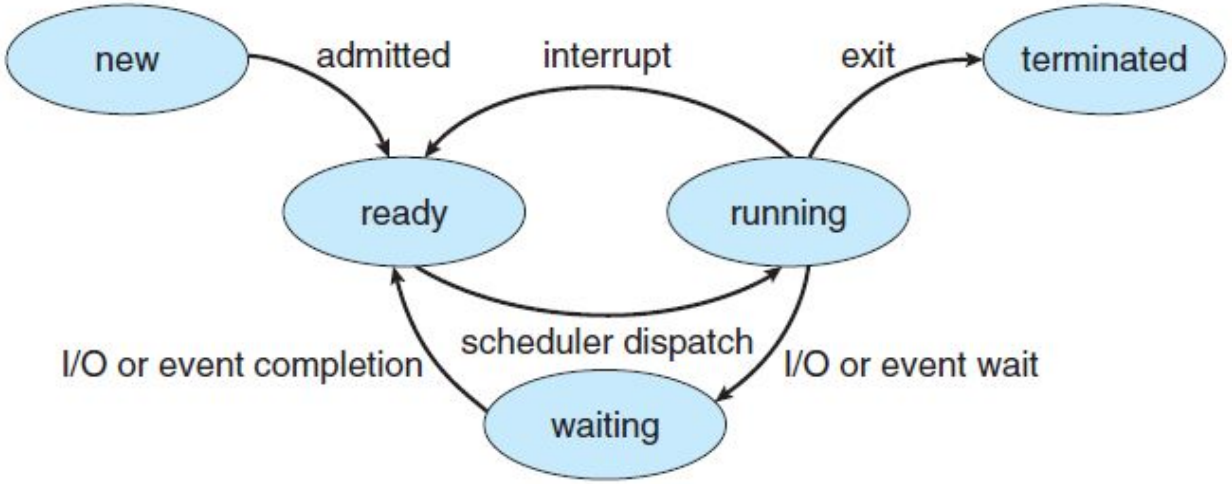
Slide 19


Program
Instruction #1
Instruction #2
Instruction #3
Instruction #4
...
Instruction #N
Includes
Disk
Program
Instantiate
RAM
Process
Abstraction
RAM
Process #1
Process #2
...
Process #M
Collect
pointers
Table of thread pointers
and their priorities
Thread #1 (high)
Thread #2 (low)
Thread #3 (med)
Thread #4 (low)
...
Thread #K (high)
Context Switch
Select for execution
Preemption
Scheduler
(Part of OS)
Execute
CPU

Process States

A process may be in one of the following states:

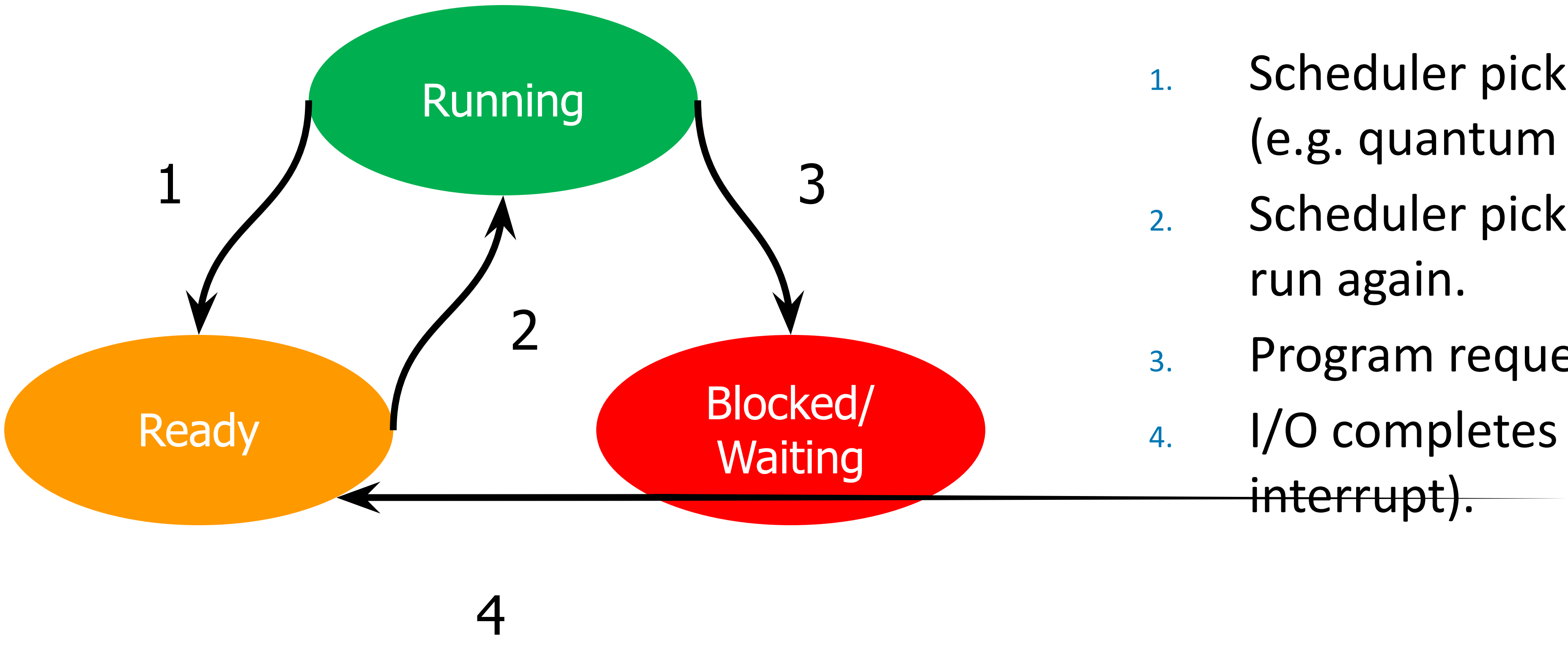
- *New:* The process is being created.
- *Ready*: The process is waiting to be assigned to a processor.
- *Running*: Instructions are being executed.
- *Blocked/Waiting*: The process is waiting for some event to occur(such as an I/O completion or reception of a signal)
- *Terminated*: The process has finished execution.



UNIVERSITY OF PORTSMOUTH

Process States

Example:



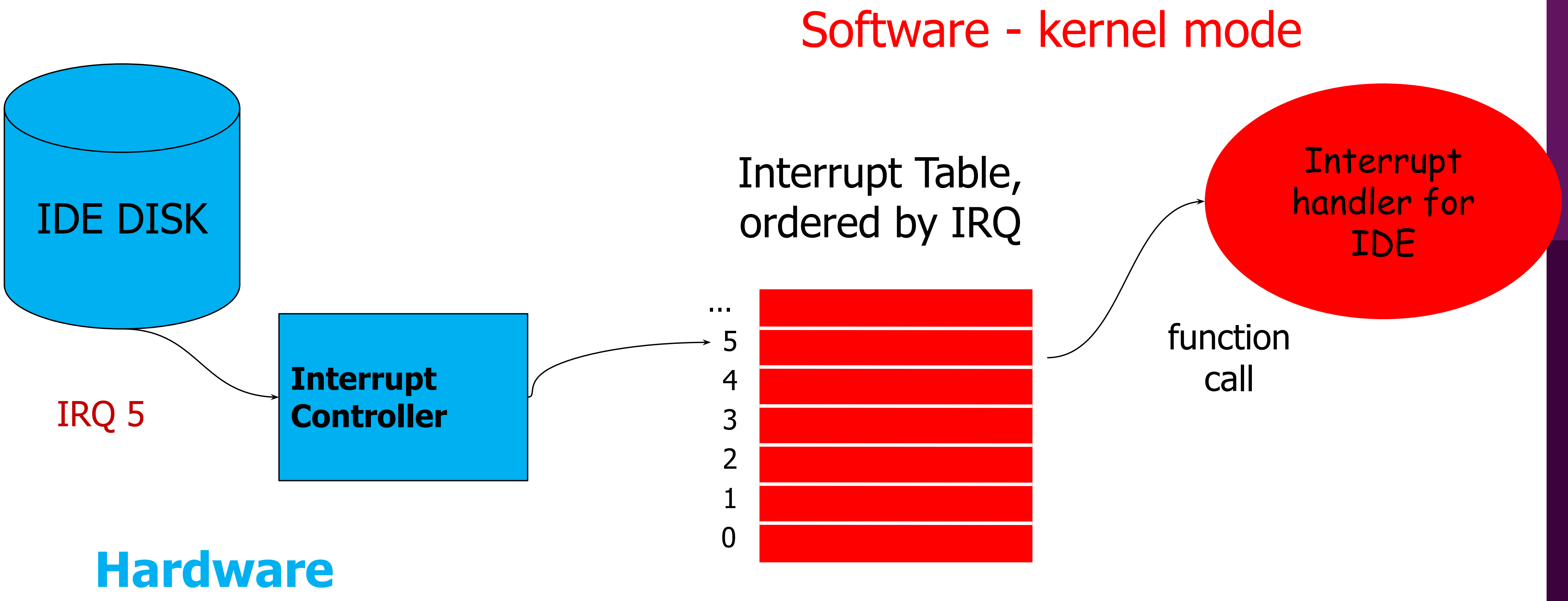
1. Scheduler picks another process (e.g. quantum is up).
2. Scheduler picks this process to run again.
3. Program requests I/O.
4. I/O completes (typically after an interrupt).

Process Transitions

- Transition 4 from *blocked* to *ready* typically driven by interrupt from an I/O device.
- Transition 1 from *running* to *ready* is also driven by regular interrupts, from the *system clock*.
- And typically there is another special kind of interrupt ("trap") on transition 3, from *running* to *blocked* state
  - See "system calls", later.
- All these interrupts are dealt with by *interrupt handlers*, installed by OS.

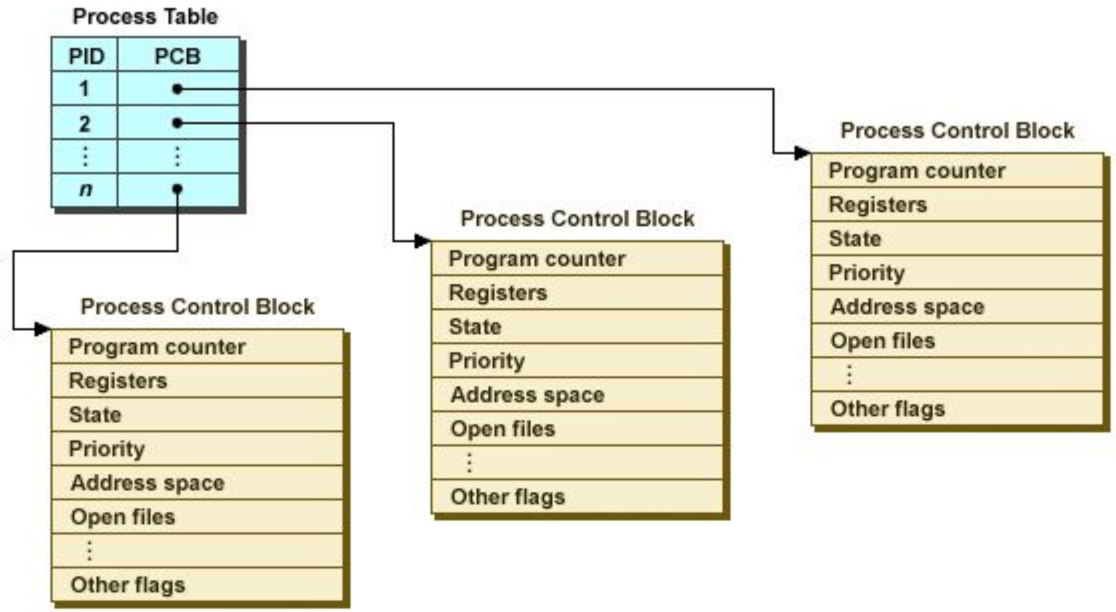
[Check Slide 12]

Interrupt Handling (transition 4)



Process Table

- The operating system maintains a data structure (*process table*) in memory with slots for every running process.
- Slot for each process is called a *Process Control Block* (PCB).



UNIVERSITY OF PORTSMOUTH

Process Control Block (PCB)

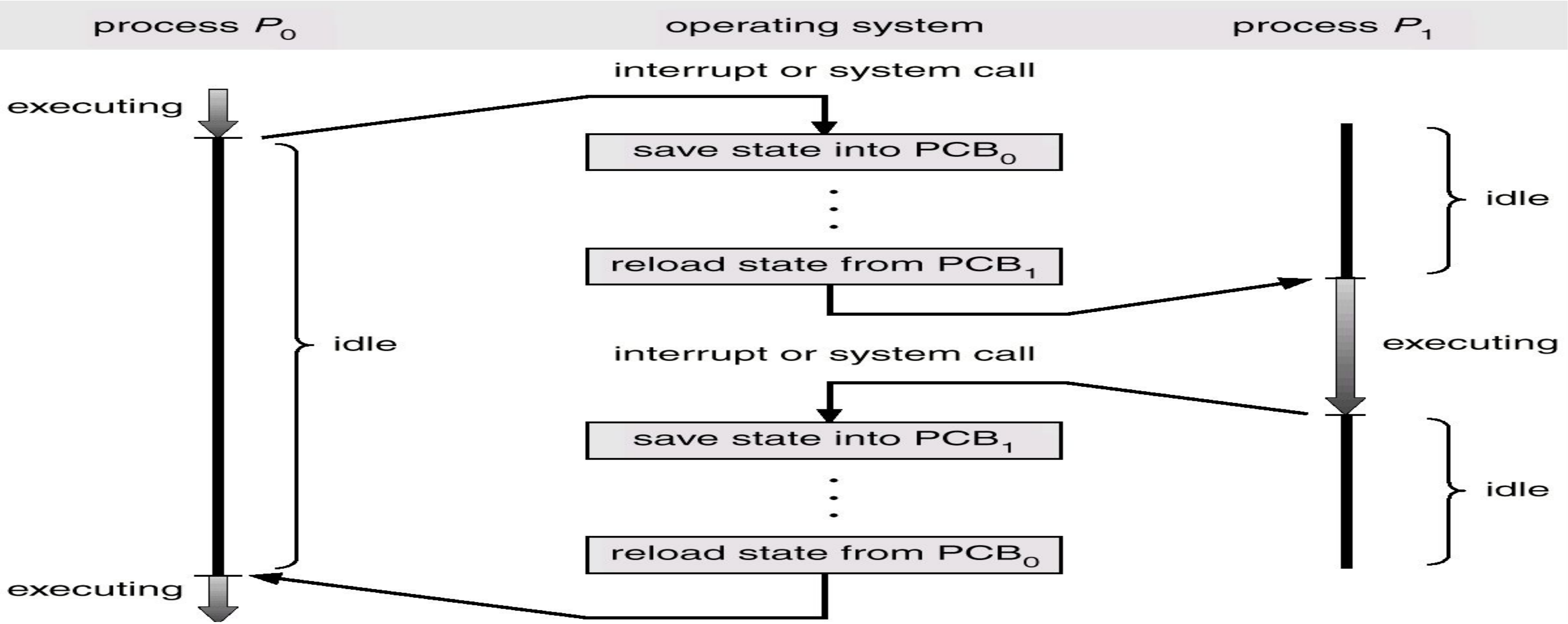
- For processes *not* presently in *running* state, a PCB includes:
  - Contents of all machine registers at time process was interrupted
    - general purpose registers, *program counter*, program status word, etc
  - Pointers to data structures associated with *memory management* for the process (see later lectures).
  - Any other information needed to restore the process in exactly the state it was in when last *running*.
- PCB does *not* contain *program variables* – these are assumed still in process' memory.

Process Control Block Example†

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

†Tanenbaum, MOS, Fig 2-4

CPU switch between processes

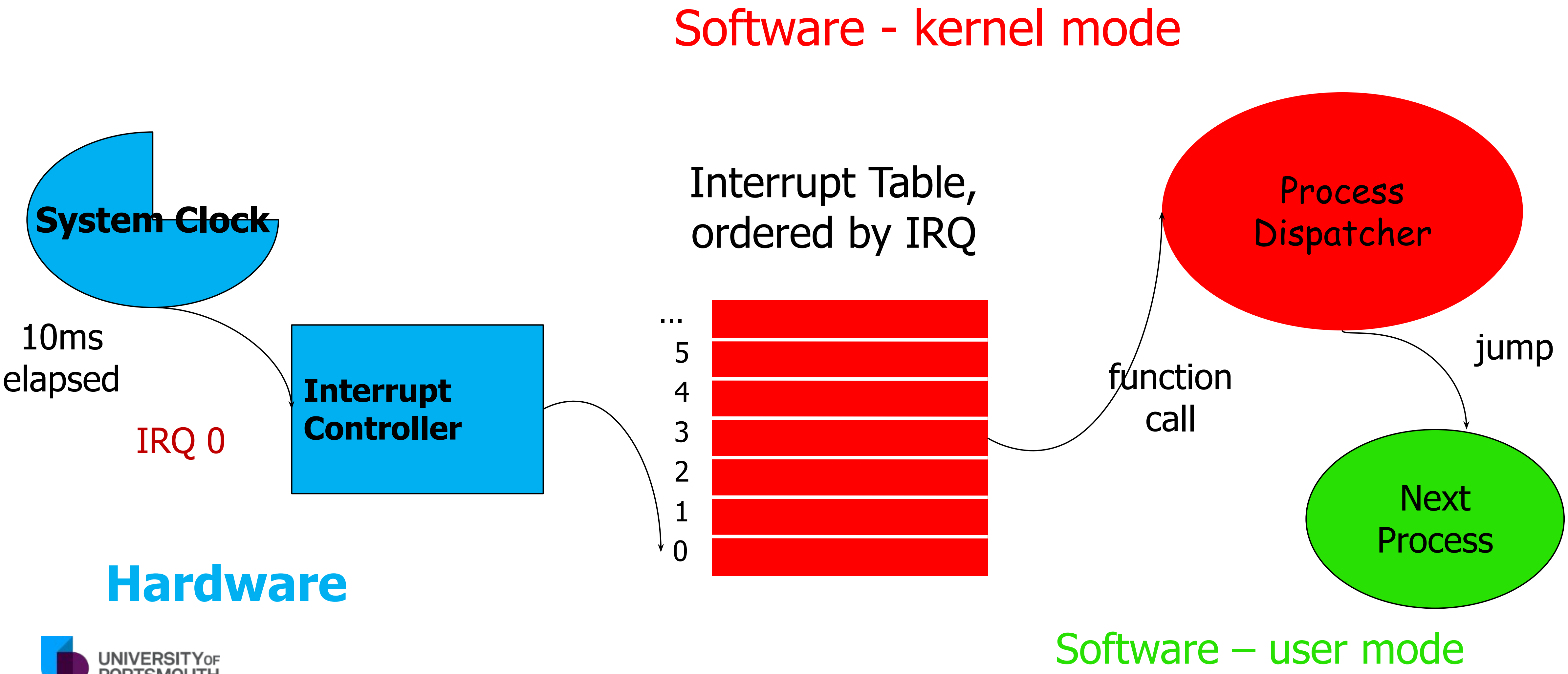


UNIVERSITY OF PORTSMOUTH

Dispatcher

- Part of the OS called the *dispatcher* gives control of the CPU to the process selected by the *scheduler*.
- This involves:
  - Stopping the currently running process,
  - Storing the hardware *registers* (PC, SP, etc.), and any other state information in *that* process' PCB,
  - Loading the hardware registers with the values stored in the *selected* process' PCB, and restores any other state information,
  - Switching to *user mode*,
  - Jumping to the proper location in the user program to restart that program.
- These steps are collectively known as *context switching*.

Time-sharing Context Switch



Multitasking Summary

- The first step of context switching is typically a call to an *interrupt handler*.
  - This starts with a fragment of machine code that saves states of registers, etc
- Process table slot for the formerly running process is added to the back of a *queue*
  - Either of *blocked* or *ready* processes.
- The scheduler chooses a new process to run from the front of *ready* queue.
  - Associated state from the process table is loaded to the CPU, and that process is resumed.

SCHEDULING

UNIVERSITY OF PORTSMOUTH

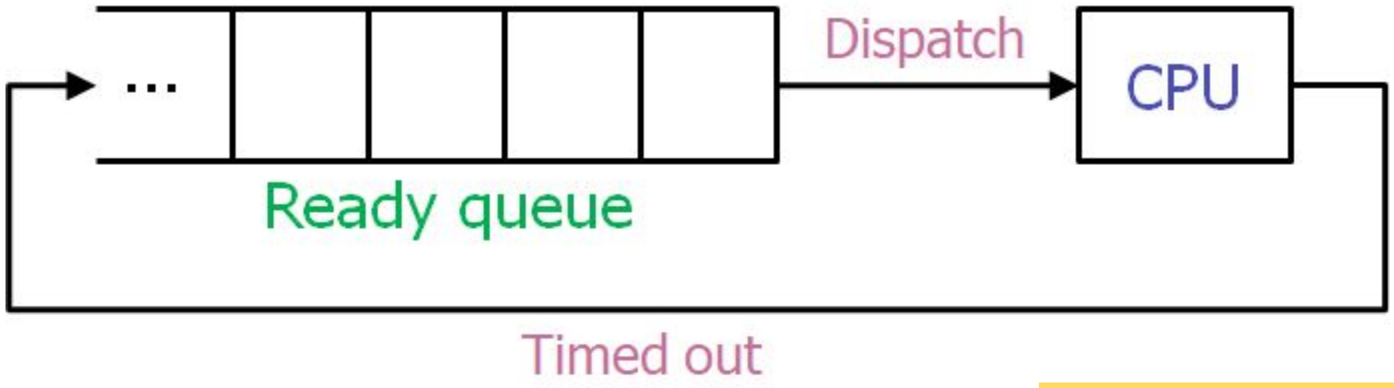
Scheduling Issues

- Before a process is selected as next to run on the CPU, a *scheduler* needs to address the following:
  - *When* should the CPU be given to another process?
  - Under *what* circumstances should the CPU be given to another process (scheduling policy) ?
  - *How long and in what order* should the CPU be given to another process (scheduling mechanism) ?
- The algorithm used to implement the mechanism is known as a *scheduling algorithm*.

UNIVERSITY OF PORTSMOUTH

Scheduler

- Conceptually, *ready* processes are on a *ready queue*.
- Assumes all processes are in memory,
- Scheduler selects a process from the ready queue and allocates the CPU to it.



Watch the short video on moodle [here]

UNIVERSITY OF PORTSMOUTH

More on Queues

- Ready queue is usually implemented using “linked list” data structure.
  - A linked list of pointers to PCB’s.
  - May be ordered by priority to give preferential treatment to high-priority processes.
- Since there is generally contention for I/O, *each device also has a queue*
  - Different from ready queue; these are where the process “goes” while it is blocked on I/O.
- *Likewise* there may process queues associated with semaphores, etc.
  - *For more on scheduling, see appendix to this week’s lecture, on Moodle.*

SYSTEM CALLS

Invoking the OS

- *User programs* (or *applications*) run in processes.
- The scheduler and other parts of the operating system will run, sandwiched between "time slices" of user processes.
  - While an application is running, the CPU is in *user mode*, and has limited power.
  - In between times, while the OS is running, the CPU is often in *kernel mode*, and has "unlimited" power.
- How does the user ask the OS to perform privileged instructions (e.g. I/O) on its behalf?

System Calls

- Practically, the user invokes OS through *system calls*.
  - To the programmer, system calls *look* like ordinary *function calls*.
  - They perform actions like creating new processes and performing input or output.
- In *UNIX* and derivatives, there are only a few dozen system calls.
  - These are standardized in *POSIX*. In a sense the set of system calls *defines* the OS.
  - Unfortunately Windows has many more (*Win32*)

UNIVERSITY OF PORTSMOUTH

Important System Calls†

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

† Tanenbaum, MOS, Fig 1-23

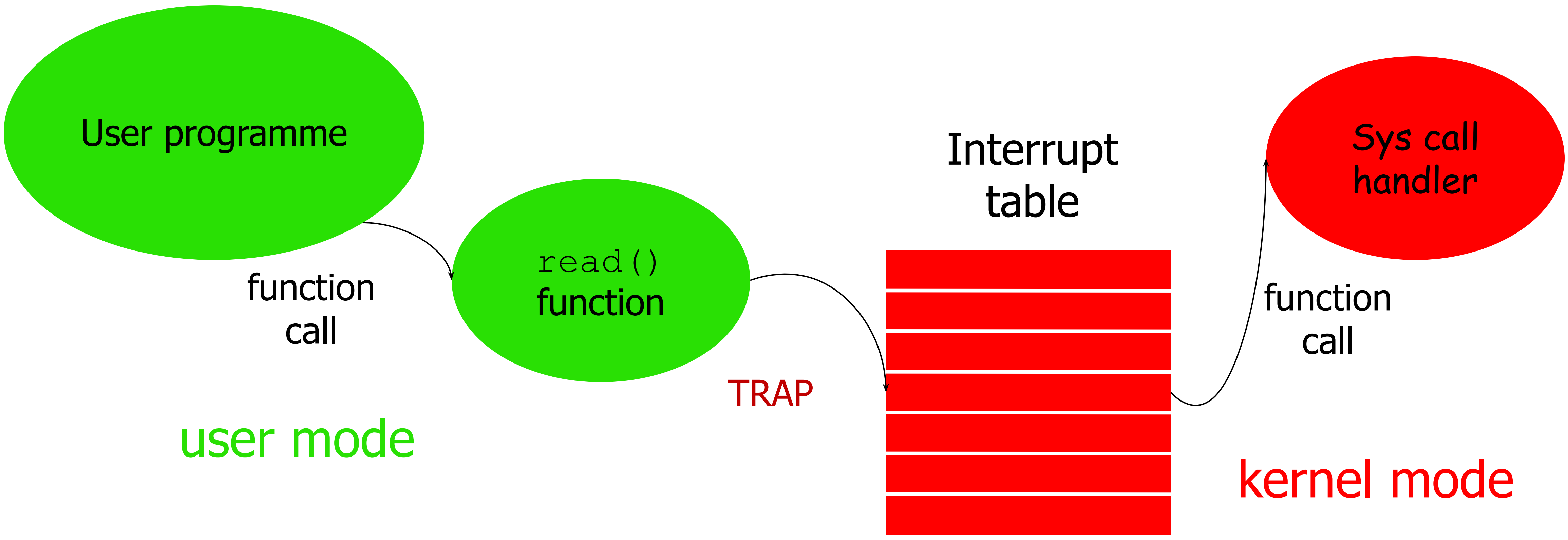
Implementation of System Calls

- The function call made by the user can't *directly* change the processor to kernel mode to perform I/O (for example)
  - A program running in user mode is forbidden from changing relevant parts of *program status word*.
- Instead a *software interrupt* ("*trap*") is raised.
  - Handled similarly to a hardware interrupt.
  - On a Pentium, the library code for the system call executes an instruction something like

    INT SYSVEC

    where SYSVEC is a constant (interrupt level).

First Stages of a read() call



Remarks

- This is mechanism of transition labelled 3 in earlier “Process States” diagram.
  - The system call handler may call into a *device driver*, that talks to some I/O device.
  - Later, when data transfer completes, a *hardware interrupt* goes through another entry in the *interrupt table*, and eventually the user process is rescheduled.
- OS initially populates the interrupt table with suitable kernel mode handler functions
  - on a Pentium using the LIDT instruction (“Load Interrupt Descriptor Table Register”)

UNIVERSITY OF PORTSMOUTH

Interrupt Handlers are All?

- In an interesting sense the OS's interrupt handlers run the whole kernel OS.
- Any switch to kernel mode that triggers kernel OS activity starts with an "interrupt" of one kind or another, followed by a jump into some table of handlers.
- In particular, all phases of process scheduling are driven by such interrupts.
  - Check arcs 1, 3, 4 in process states diagram. Arc 2 happens as the last step of "handling the interrupt".

UNIVERSITY OF PORTSMOUTH

Summary

- Started with definitions of an OS process
- Discussed *process states*.
- Went on to consider how multitasking is implemented by OS – *context switching* and *scheduling*.
- Finally considered implementation of *system calls*.
- *Next lecture*: *Inter-process Communication*

UNIVERSITY OF PORTSMOUTH

Further Reading

- Andrew S. Tanenbaum, “*Modern Operating Systems*”, 4th Edition, Pearson, 2014 (MOS), Chapter 2.
- Andrew S. Tanenbaum and Albert S. Woodhull “Operating Systems Design and Implementation”, 3rd Edition, 2009 (MODI)
  - Chapters 1 and 2 contain details of how system calls and interrupt handlers work, in the context of MINIX.
- "Operating System Concepts", 10th Edition, Abraham Silberschatz, Peter Galvin, Greg Gagne

UNIVERSITY OF PORTSMOUTH

Questions?