MONASH University
Information Technology

# FIT9134
# Computer architecture and operating systems

**Week 6**

**Operating Systems IV:**
**Process Management Concepts**

# Next 3 weeks …

- the materials covered in the next ***3*** lectures (Weeks 7-9) are very important for the Shell Scripting lab tasks in the final 4 labs. Please make sure you attend these lectures; otherwise you will have difficulties completing those lab tasks.

# Processes.

- A process is a ***program in execution***.
  - consists of executable program, its data & stack, its ***Program Counter*** (***PC***), ***Stack Counter*** (***SP***) and other info necessary to run (or restart) the program.

- Process control is an important part of multi-tasking O/S –this involves allocating memory, cpu time, I/O devices, etc

- Modern O/S breaks processes down further - into threads – which are smaller individually executable pieces of a process. This makes more efficient use of the CPU.

- Information about processes is stored in a ***Process Table*** (a data structure of O/S ).

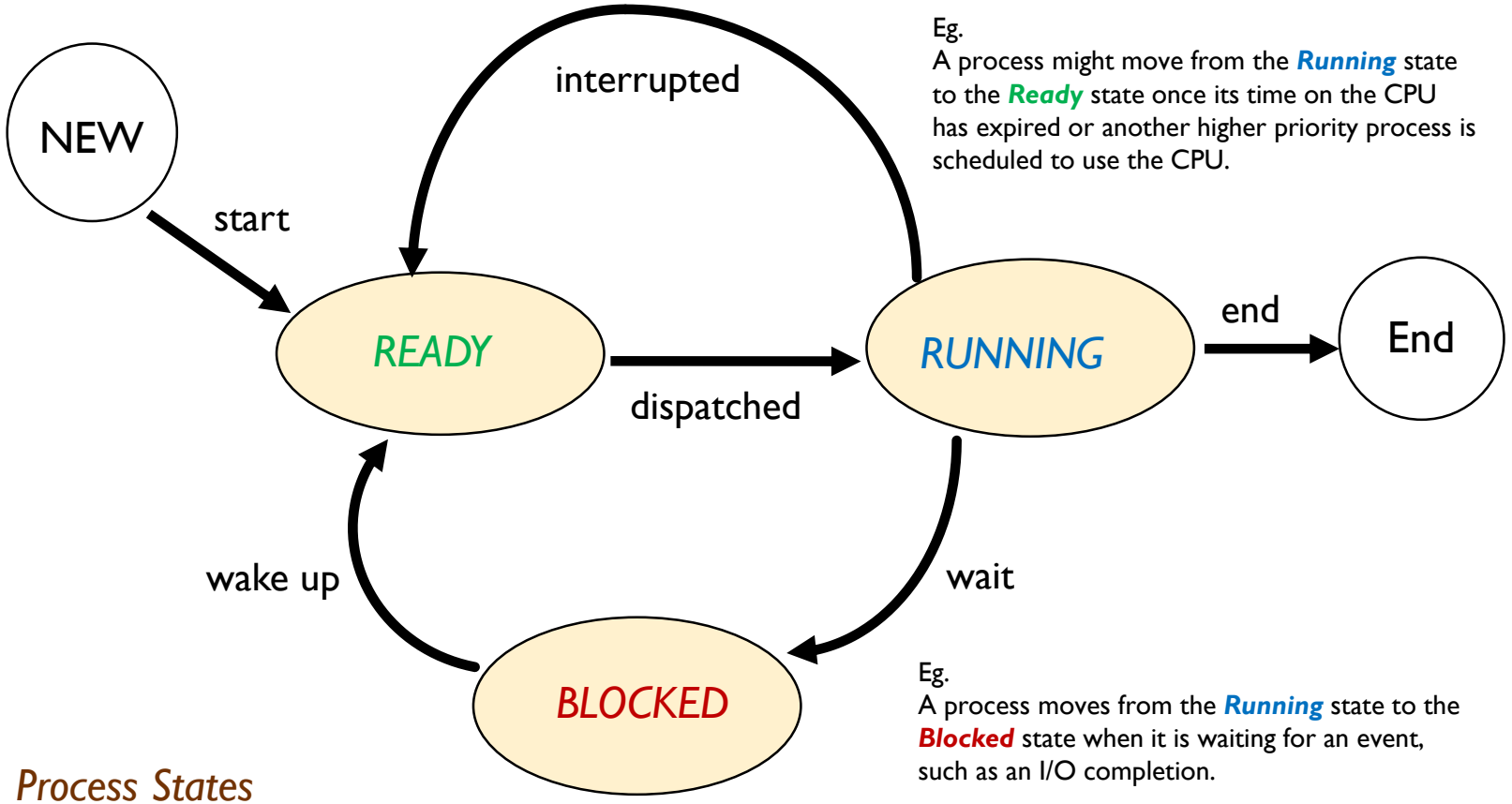
# Process States

A process can be in one of 3 main "***states***" :

**Ready**: waiting for a processor to become available

***Running***: instructions are being executed

***Blocked*:** waiting for some event, eg. I/O completion




Eg.
A process might move from the ***Running*** state to the ***Ready*** state once its time on the CPU has expired or another higher priority process is scheduled to use the CPU.

Eg.
A process moves from the ***Running*** state to the ***Blocked*** state when it is waiting for an event, such as an I/O completion.

*Process States*

# Processes

- A process may create a new process as it runs - this is called ***forking*** or ***spawning*** a new process.
- The original process is called the ***parent*** process, the new one the ***child*** process.
- The ***Unix*** `ps` (try the `-ael` option) command can show information about currently running processes.

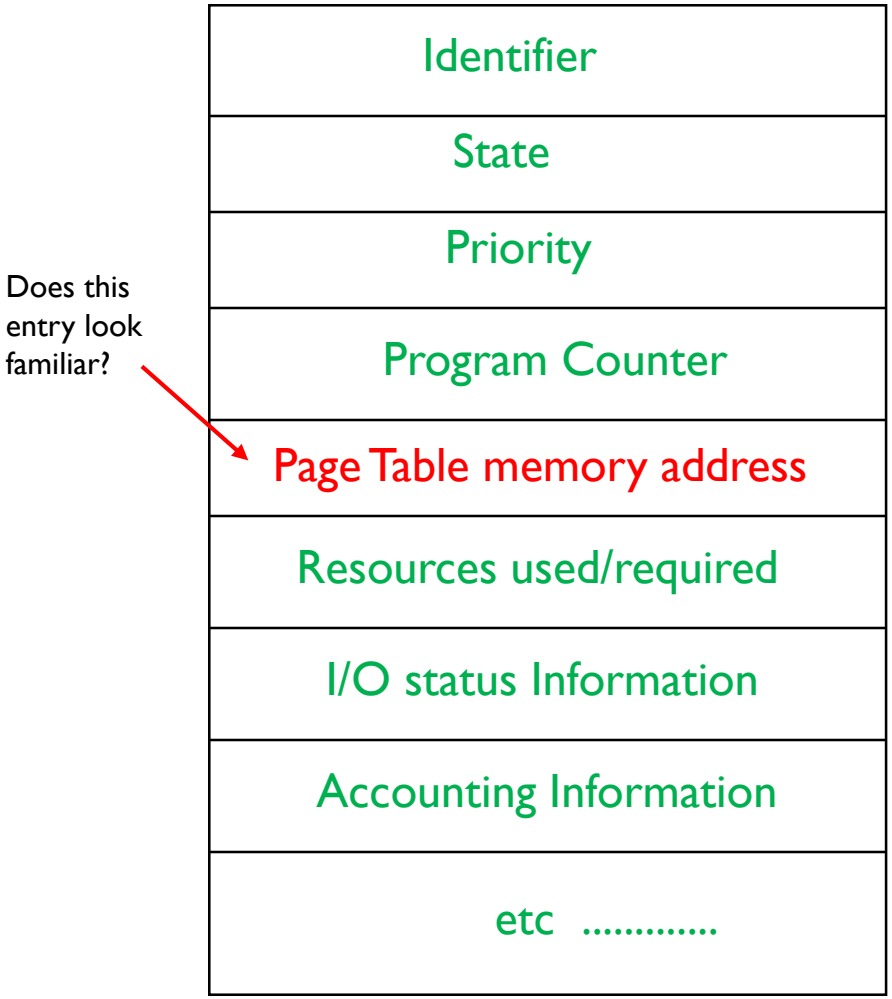
# Process Control Block

- Usually there are more processes than processors. Concurrency achieved by interleaving processes i.e. allocating each process a fraction of the CPU time.

- When a process is interrupted, its current state must be saved, for it to be resumed later. This info is stored in a "Process Control Block", which forms one entry in the Process Table .

- Process Control Block (PCB)
  - is an in-memory data structure created by O/S
  - is used to identify process
  - stores status of process
  - stores its 'volatile environment' (e.g. register values)

# Process control block (PCB)




- A ***PCB*** exists for every process in the system.
- A ***Process Table*** contains ***PCBs*** for all the processes.
- Entries in the Process Table may be linked together to form a list, or stored in an array; each entry in the list (or in the array) is for one PCB.

# Scheduling

- CPU scheduling is divided into two separate components - the ***long-term*** and ***short-term*** schedulers.

- ***Long-term*** (or ***high-level***) scheduler decides which processes will be admitted into the system's *Ready Queue*. Decision based on memory availability and system load. Important in batch systems, but less so in most modern interactive systems.

- ***Short-term*** (or ***low-level***) scheduler works with processes ***already in memory*** and ready to run. A ***Dispatcher*** then decides which one to run next.

# High-Level Scheduler

- If there is not enough memory to hold all processes, high-level scheduler will swap jobs from disk to memory and vice versa

- Criteria that the high-level scheduler could use may include:

  - how long the process has been swapped in/out
  - how much CPU time has the process recently had
  - how big is the process
  - how high is the process priority
  - etc

# Low-Level Scheduler

- A running process may need to stop for I/O, or is interrupted for some reason. The dispatcher will then choose the next process to run.

- An operating system may use ***pre-emptive*** or ***non-pre-emptive*** low-level scheduling.

# Scheduling algorithms

- different algorithms favour different types of processes, and different criteria may be used to determine the "best" algorithm

- examples of criteria which may be used are:
  - minimize response time for interactive processes
  - ensure fairness for all processes
  - maximize throughput
  - prevent CPU starvation

# Pre-emptive Vs Non-Pre-emptive Scheduling

- **Non-Pre-emptive** scheduling means system relies on a (well-written) process itself to relinquish CPU when it finishes.
  - Eg. Windows 3.1, Windows 95 (16-bit), "Classic" MacOS, etc

- **Pre-emptive** scheduling means the O/S controls how the CPU is shared by the processes.
  - Eg. Unix, Linux, Windows NT/2000/XP/Vista/8/9/10, Mac OS X, etc

# Non-pre-emptive scheduling

- Non-pre-emptive algorithms are more applicable to batch systems. Differ from pre-emptive as processes will only stop executing when they decide to stop.

- Examples of non-pre-emptive algorithms :

  - First-in, first-out, where processes are executed in the order they arrive
  - Shortest job first, which maximizes throughput, but may result in job starvation
  - Priority scheduling, where priorities are assigned based on criteria such as resources requested, or financial criteria

# Pre-emptive Scheduling

- With pre-emptive scheduling, computer uses an inbuilt clock to ensure no process runs for too long. Pre-emptive scheduling is more common in interactive systems, but involves much more overhead. *Most modern O/S's use pre-emptive scheduling*.

- Eg: an internal clock creates interrupts 50-100 times/sec. The O/S dispatcher runs at each clock interrupt to decide on next process to execute.

- Different algorithms can be used to achieve maximum efficiency and response.

# Pre-emptive scheduling algorithms

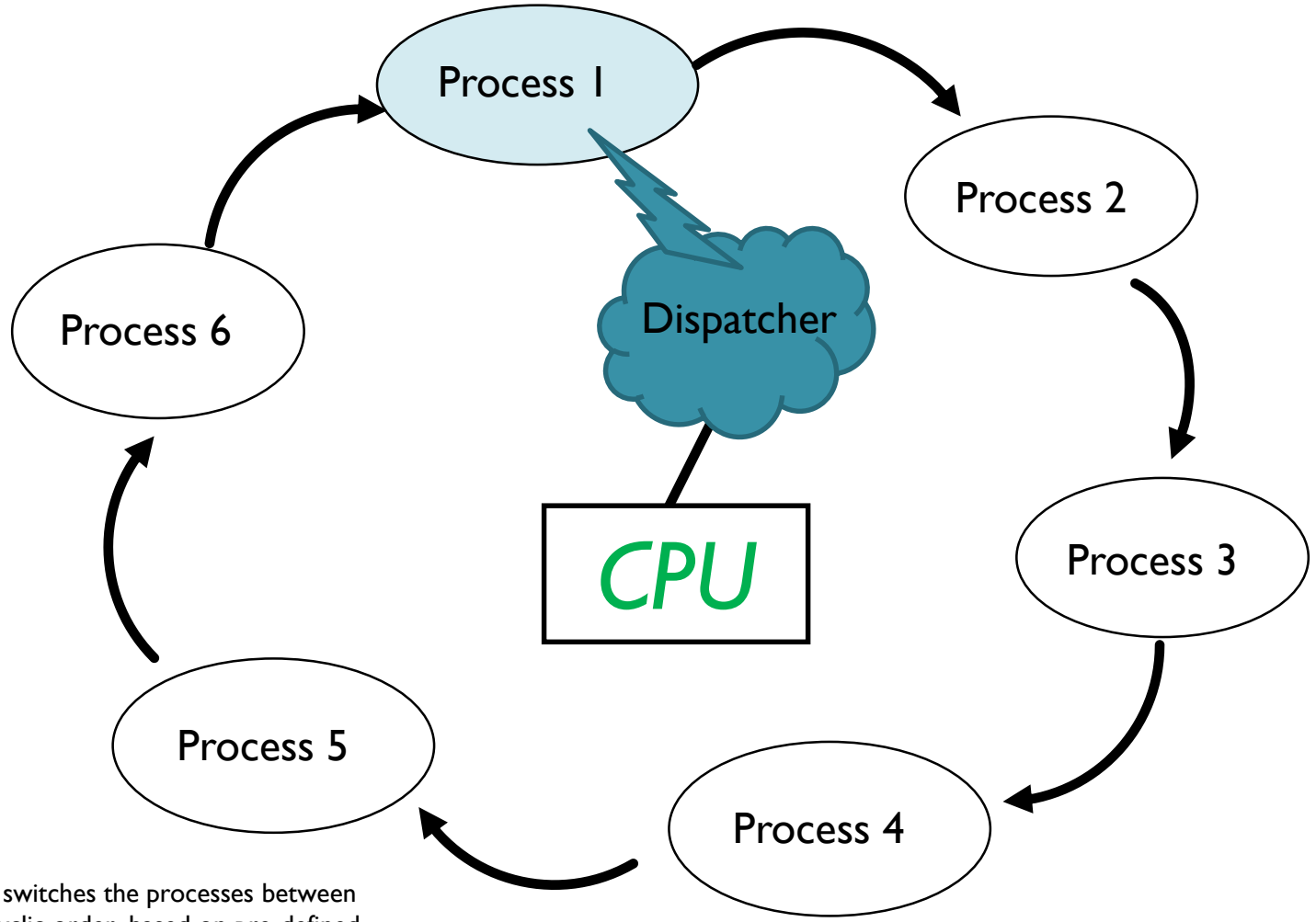
***Round Robin*** :

- All processes assigned equal *time quantum* to run. All ready-to-run processes are maintained in circular linked-list, and take turn to use CPU.

- how long should a reasonable time quantum be?

  Eg.
  - If process switch takes 5msec, 20 ms quantum means 25% of CPU time spent just switching processes.
  - If 500 ms quantum is used - very slow response time to interactive users.
  - Usually, quantum of ~100 msec is used.

# Round Robin Scheduling




The Scheduler switches the processes between the CPU in a cyclic order, based on pre-defined time quantums.

# Round Robin problems

- Round Robin does not allow definition of "more important processes", ie. priority

- Round robin also indirectly penalizes processes that frequently use I/O resources, by always returning them to back of queue even if used only small % of quantum. This is because I/O always takes longer to complete, hence such processes have higher chance of waiting/blocking.

# Other Scheduling Algorithm

***Priority Scheduling :***

- Processes given initial priority level. Usually multiple priority classes exist. Runnable processes maintained in priority queues, with round robin used within queue.

- To prevent CPU starvation of low priority jobs, may need to temporarily boost priority (eg. if they have been waiting for long periods). Once process has had its share of CPU time, its priority drops back to normal.

# Dynamic Priority Scheduling

- Another variation of priority scheduling is to assign priorities dynamically, using some formula.

- For instance, based on fraction of the last time quantum used (f), priority formula could be **1/f** (ie. more time used now, lesser priority later). This would favour interactive users and I/O bound jobs (these tends to spend more time in blocked states & use less CPU time quantum, then the **1/f** formula will give them higher priorities) rather than CPU bound jobs.

# Process Management

- When dealing with multiple processes sharing the same CPU, we must consider 3 important situations :

1) ***Mutual Exclusion*** - ensuring that non-shareable resources (e.g. printers) are accessed by only one process at a time.
2) ***Synchronization*** - for processes which must cooperate (e.g. chat programs) with one another.

# Process Management

3) ***Deadlock*** - when two or more processes want to use a non-shareable resource held by each other.

- for example:
  - User A opens file1 with lock
  - User B opens file2 with lock
  - User A wants to open file2 but cannot..
  - User B wants to open file1 but cannot..

➔ "Deadlock" situation!

# Next time (Week 10)…

- In order to implement *mutual exclusion*, *synchronization* and deal with *deadlocks*, some form of process-coordination is required.
- In addition, processes often need to exchange information.
- Both of these goals are met by ***Inter-Process Communication*** mechanisms (***IPC***).