_ Chapter 5:

- Maximum CPU utilization obtained with multiprogramming
  - _ I/O brust cycle _
- CPU↑process execution consists of a cycle of CPU execution and I/O wait
  - ↳ CPU burst followed by I/O burst
- Short term Scheduler: Selects among the processes in the ready queue, and allocates the CPU to one of them.
  - ↳ Queue may be ordered
- CPU sheduling decisions may take a place when a process:
  - ↳ 1) Switches from running to waiting state.
  - 2) Switches from running to ready State.
  - 3) Switches from waiting to ready
  - 4) Terminates
- In 1 and 4 the process stopped by her own, (nonpreemtive) while in the rest, it was forced to stop (time slice ended, need new, high priority state ready) - preempt lower priority) are and called preemptive
- Dispatcher: gives Control of the CPU to the process selected by the Short-term Scheduler
  - ↳ 1) Switching Context [Store old PCB, reload new PCB]
  - 2) Switching to user mode
  - 3) Jumping to the proper location in the user program to restart that program [defined by PC]
  - → dispatch latency: time it takes the dispatch to stop one process and start another running. (overhead)

- CPU time: time process needs from the CPU to finish
- I/O time: time a process needs to do all its I/O
- wait time: time the process spends outside the CPU (I/O + Ready)
- Start time: time the process entered ready queue.
- Finish time: time the process exits the system (terminated)
- TurnAround (TA) time: Finish time - Start time
- wait (W) time: TA - Total time

- Scheduling Criteria
  - ↳ 1) CPU utilization - keep the CPU Busy as possible = $\frac{\text{CPU work time}}{\text{CPU work time + CPU wait time}}$
    Max
  - 2) Throughput: # of processes that complete their execution per time unit
    Max : per process
  - 3) Turnaround time: amount of time to execute a particular process.
    Min
  - 4) waiting time: amount of time a process has been waiting in the ready queue
    Min
  - 5) Response time: amount of time it takes from when a request was submitted until the first response is produced. not output
    Min
    (for time-sharing environment)

- First-come, First-Served (FCFS) Scheduling

Process	Burst Time	Arr-time	Duration	Finish	TA	WTA (waited Turnaround) ↳ TA / Duration
P_1	24	0	24	24	24	1
P_2	3	0	3	27	27	9
P_3	4	0	3	30	30	10

- consider one CPU-bound and many I/O-bound processes
- Convoy effect: Short process behind long process
- Shortest Job First (SJF) Scheduling. (non-preemtive)
  1) associate with each process the length of ~~the~~ its next CPU burst.
  2) is optimal - gives minimum average ~~time~~ waiting time for a given set of processes.
  3) has a difficulty in knowing the length of the next CPU request. (could ask the user)
- Shortest remaining time first (preemtive version of SJF)

- Priority Scheduling: a priority number is associated with each process.
  - the CPU is allocated to the process with the highest priority (Preemtive or nonpreemtive)
- SJF is priority Scheduling where priority is the inverse of predicted next CPU brust time.
- Has a problem (Starvation) - low priority processes may never execute
  - a solution (Aging) - as time progresses increase the priority of the process.
- Determine length of next CPU burst.
  - How to determine shortest job? Rebeat (known); Predict
  - can only estimate the length - should be similar to the previous one.
    - then pick process with shortest pridicted next CPU brust.
  - → can be done by using the length of the previous CPU burst, using
    1) t_n = actual length of n^{th} CPU burst.
    2) τ_{n+1} = predicted value for the next CPU burst.
    3) α, $0 \leq \alpha \leq 1$ / Commonly α set to 1/2
    4) $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

- If $\alpha = 0$, Recent history does not count.
  If $\alpha = 1$, Only the actual last CPU burst counts

- Since both a and (1-α) are less than or equal to 1, each successive term has less weight than its predecessor.

- Mono-programming (the process had all CPU time)

- Round Robin (RR): Each process gets a small unit of CPU time (time quantum), after this time has elapsed, the process is preemted and added to the end of the ready queue. [time usually 10-100 millisecond]

- when having n processes in the ready queue, and the time quantum is q, each process gets 1/n of the CPU time in chunks of at most q time unit at once. [no process waits more than (n-1)q time unit]

- Timer interrupts every quantum to schedule next process.
- Performance:
  - ↳ . If q is large → FIFO
  - . If q is Small → "Overhead is high" q must be large with respect to context switch

80% of CPU bursts should be shorter than q

- Context Switch and time quantum:
  - ↳ . Contex switch = 1ms + time quantum = 1 → waste 5%
  - . Contex Switch = 1ms + time quantum = 9 → waste 10%
  - . Contex Switch = 1ms + time quantum = 99 → waste 1%
- RR is higher average turnaround than SJF, but better response. 1/10
- RR with Similar Processes: # of processes with same ratio of CPU and wait times
  - ↳ degree of MP = # of processes
- q should be large compared to context switch time: q ∈ [10,100] ms, context switch < 10 μsec
- Multilevel queue: ⇒ Ready queue is partitioned into separate queues:
  - ↳ 1) foreground (interactive) FG 2) backgroud (batch) BG

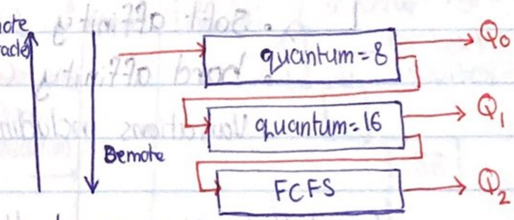
- Chapter 5 - Cont. #1

- Process permanently in a given queue (FG or BG)
  - ↳ each queue has its own scheduling algorithm
    - FG → RR
    - BG → FCFS
- Scheduling must be done between the queues
  - ↳ 1) Time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes.

    2) Fixed priority: serve all from ~~back~~ foreground and then background.
    [possibility of starvation]
- priorities from highest to lowest: (Multilevel queue scheduling).
  1) System processes
  2) interactive processes
  3) Interactive editing processes
  4) batch processes
  5) student processes ✗ شطبها الأخ

- A process can move between the various queus, aging can be implemented this way

- Multi-level feedback queue scheduler
  - ↳ 1) number of queues
    2) Scheduling algorithms for each queue
    3) method used to determine when to upgrade a process.
    4) method used to determine when to downgrade a process.
    5) method to determine which queue a process will enter when that process needs service [initially joins which queue]



- thread scheduling → distinction between user-level and kernal-level threads.
  - ↳ when threads supported, threads scheduled not processes.

- In Many-to-one and Many-to-Many models, thread library schedules user-level user threads to run on LWP.
  - ↳ Process-contention scope (PCS): Scheduling competition is within the process (not global)
    - done via priority set by Programer

&

Kernal threads are Scheduled into available CPU (System-conteniton Scope) (SCS): competition among all threads in the system.

- Pthread scheduling: API allows specifying PCS or SCS during thread Creation. (فش تكملة لقدام فش وسع)

- CPU Scheduling more complex when multiple CPUs are available
  - ↳ Homogeneous processors (they all are the same) within a multiprocessor
  - Asymmetric multiprocessor: only one processor accesses the System data Structure, allevating the need for data Sharing.
  - Symmetric multiprocessor: (SMP) each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes. (currently most common)

- processor affinity: process has affinity for processor on which it is currently running.
  - ↳ Soft affinity (Prefers to work on a certain process if available)
  - hard affinity (work on a certain process only)
  - Variations including processor Sets.

- Load Balancing in multiple-Processor Scheduling.
  - ↳ SMP need to keep all CPUs loaded for efficiency.
    - ↳ 1) Load balancing: attempts to keep workload evenly distributed
    - 2) Push migration: Periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs.
    - 3) Pull migration: idle processor pull waiting tasks from busy processor

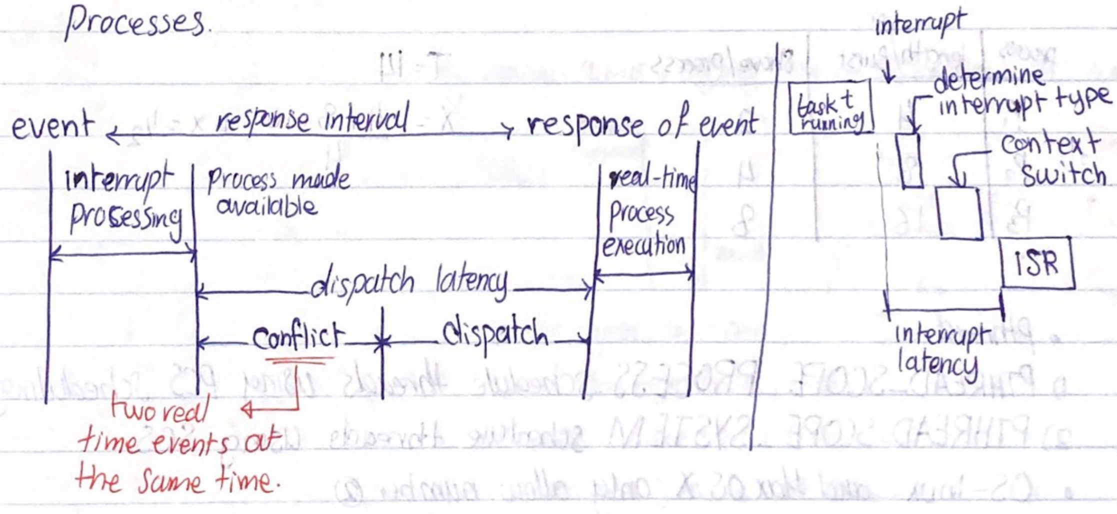
[Could be for both SMP and AMP]

- Multicore processors
  ↳ they start to replace multiple processors cores on same physical ship, this is faster and consums power less, in addition to having multiple threads per core growing.
  - takes advantage of memory stall make progress on another thread while memory retrieve happens
- Real-time CPU Scheduling
  ↳ 1) Soft real-time ~~sche~~ systems: no guarantee as to when critical real-time process will be scheduled (ex: video, speech)

  2) Hard real-time systems: task must be serviced by its deadline ex(power switch, car breaking)

- latencies that affect performance.
  ↳ 1) interrupt latency: time from arrival of interrupt to start of routine that services interrupt.
  2) dispatch latency: time for schedule to take curren process off CPU and switch to another.

- Confict phase of dispatch latency:
  ↳ 1) preemption of any process running on kernal mode.
  2) Release by low-priority process of resources needed by high-priority processes.



- Priority-based Scheduling: for real-time, Scheduller must support preemptive, Priority-based Scheduling.
  ↳ only guarantees soft real-time: no guarntee for in time work!
- for hard real-time must provide ability to meet deadlines.

- Processes have new characteristics
  - ↳ periodic ones require CPU at constant intervals.
    - ↳ 1. Has processing time t, deadline d, and a period p. $0 \leq t \leq d \leq p$
    - 2. Rate of periodic task is $1/p$.

- Rate Monotonic Scheduling: A priority is assigned based on the inverse of its period.
  - ↳ Shortest period → higher priority

- Earliest deadline First (EDF)
  - ↳ the earlier the deadline, the higher the priority.

- Proportional Share Scheduling
  - ↳ . T shares are allocated among all processes in the System
  - . an application receives N shares where N<T, this ensures each application will receive N/T of the total ~~pross~~ processor time.

process	length/Burst or	Share/process
P_1	4	2
P_2	8	4
P_3	16	8

$T = 14$

$\frac{1}{x} = \frac{4+8+16}{4} \Rightarrow x = 1/2$

- pthread:
1) PTHREAD_SCOPE_PROCESS schedule threads using PCS scheduling
2) PTHREAD_SCOPE_SYSTEM schedule threads using SCS
- OS-linux and Max OS X only allow number ②
  - ↳ can undo good scheduling algorithm efforts of guests.
- Virtualization and Scheduling: virtualization software schedules multiple guests
  - ↳ Each guest doing its own sched onto CPUs
    1) not knowing it doesn't own the CPU
    2) Can result in poor response time
    3) Can effect time-of-day clocks on guests.

Ch5 : Process Synchronization

- Processec can execute concurrently.
- Concurrent acess to shared data may result in data inconsistency.
- Solution for the Consumer-Producer problem that fills a buffer
  ↳ 1. Initially, Counter is 0
  2. ++ by the producer
  3. -- by the consumer

- Machine instruction: Atomic if cannot be ~~#~~ interrupted
- Critical section: Process may be changing common variables, updating shared table, writing shared file
  → when a process in a critical section, no other may be in its Critical Section.

⇒ the part of the program (process) that is accessing and changing shared data is called Critical Section

- Critical Section Problem ~~s~~ is to design protocols to solve this, each
  ↳ process must:
  - ask permission to enter critical section in entry section.
  - may follow critical section with exit section
  - Then perform the reminder section.

- Peterson's Solution.

- Solution to critical Section Problem:
  ↳
  - Mutual Exclusion: if a process is executing in its critical section, no other
  - ~~Progress~~ Processes ~~with~~ can be executing in their critical sections

Progress. ~~Bounded waiting~~: If no process is executing in its critical section and their exist processes that wish to enter their critical section, then the selection of the next one cannot be postponed ~~intl~~ indefinitely

- Bounded waiting: a bound must exist on the number of times that ~~the~~ othe processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- test and Set instruction
  ↳ • Executed atomically [cannot be interrupted]
    - Returns the original value of passed parameter
    - Set the new passed parameter to "True"

- Compare-and-Swap instruction / int compare-and-swap(int*value, int expected, int*new-value)
  ↳ • Executed atomically
    - Returns the original value
    - Set the variable "value" to the value of passed by parameter but only if value == expected (the swap takes place only under this condition)

- Semaphore: synchronization tool that provides more sophisticated ways (than mutex locks) for process to synchronize their activites
  ↳ only accessed via two indivisible (atomic) operations
    ↳ wait() and signal() [P() and V()]

- Counting semaphore: integer value can range over an unrestricted domain

- Binary semaphore: integer value can range only between 0 and 1
  ↳ Same as mutex luck

- Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Starvation - indefinite blocking: A process may never be removed from the semaphore queue in which it is suspended.

- Priority Inversion: Scheduling problem when lower-priorty process holds a lock needed by higher-priority process.
  → Solved via priority-inheritance protocol