Capitolo 5 : Scheduling della CPU

Table of contents

- Capitolo 5 : Scheduling della CPU
  - Tempistica dei CPU Burst
  - CPU scheduler
  - Dispatcher
- CPU Scheduling
  - Dispatcher
  - Criteri di Scheduling
- Tecniche di scheduling
  - First-Come, First-Served (FCFS)
  - Scheduling Shortest-Job-First (SJF)
  - Scheduling adottato comunemente : Round Robin (RR)
  - Priority Scheduling - basato su priorità
  - Multilevel Queue
- Thread Scheduling
- Scheduling di processori multipli
  - Sistema multithread multicore
  - Real-Time CPU scheduling
  - Scheduling Linux
  - Little's Formula

Ragionando in maniera astratta, il concetto fondamentale è che se voglio avere l'utilizzo **massimo** della CPU, devo usare la **multiprogrammazione**. Questo è un concetto fondamentale, e significa che nel momento in cui un processo si ferma perché attende il completamento, ad esempio, di un'operazione di I/O, qualche altro processo prende l'utilizzo della CPU, in modo che essa sia sempre utilizzata per computare qualcosa.
Questo concetto è adottato da **tutti** i SO moderni.

Durante la vita di un generico processo, c'è una **successione di fasi**, in cui ci sono delle sequenze **interne alla CPU**, quindi senza I/O, inframezzate da sequenze di I/O. Esiste quindi un ciclo di **burst**, ovvero una **raffica** di operazioni di CPU; nel generico processo esiste un susseguirsi di fasi in cui si lavora **internamente** alla CPU, e fasi in cui sono richieste delle **operazioni di I/O**.

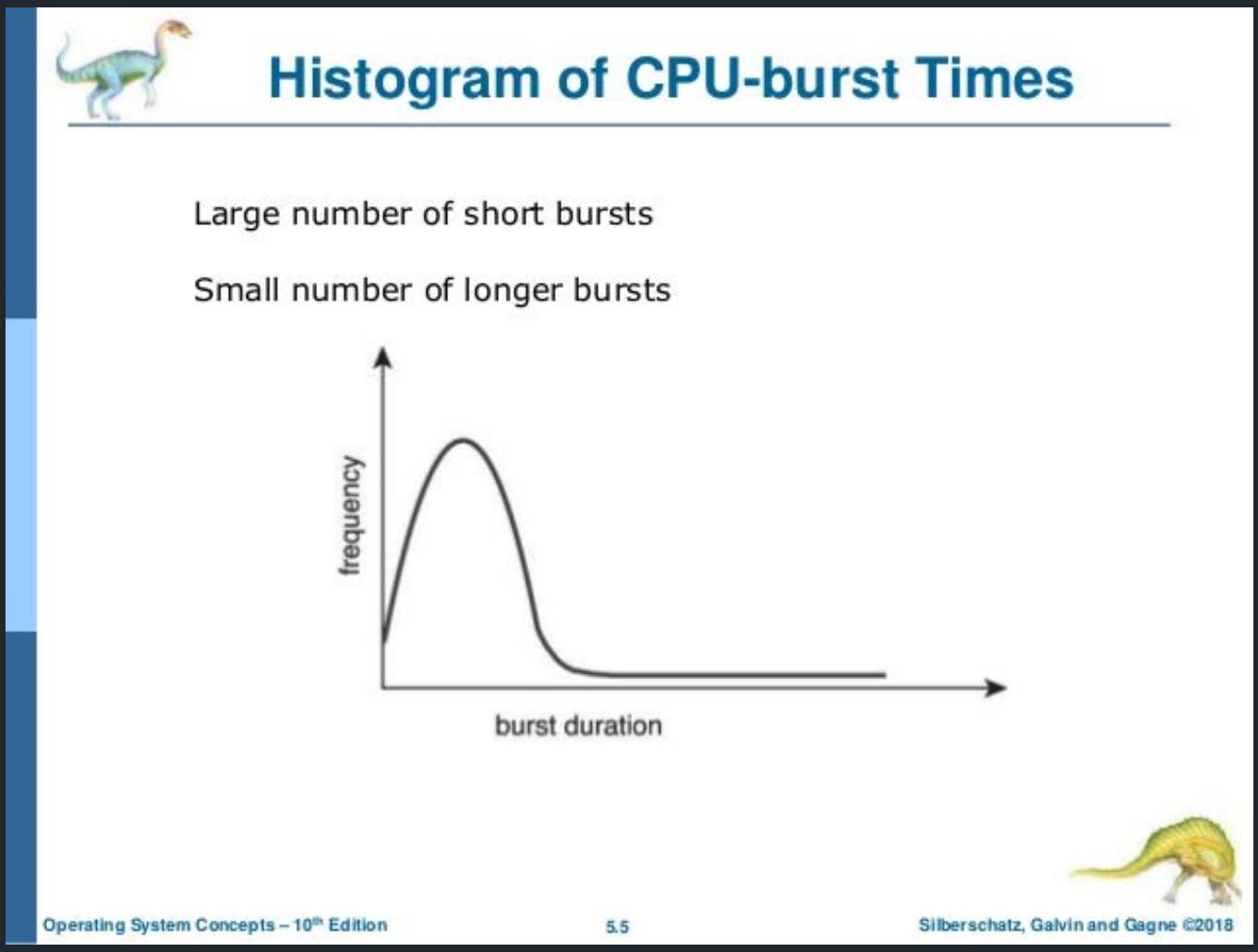
Abbiamo quindi, parlando in termini tecnici, un susseguirsi di

- **CPU** : cicli di operazioni interne alla CPU
- **I/O burst** : cicli di operazioni I/O.

L'insieme di queste operazioni è detto **CPU-I/O burst cycle**, e questo avviene per qualsiasi processo.

Tempistica dei CPU Burst

Quello che succede tipicamente, è che i burst di CPU sono abbastanza brevi:



Infatti notiamo che maggiore è la **frequenza**, e quindi la presenza di un burst, minore è il suo tempo di esecuzione (durata del burst). Di conseguenza i burst più frequenti, sono anche quelli che durano il minor tempo.

CPU scheduler

Lo scheduler è quella parte del SO che decide, tra tutti i processi presenti nella coda **Ready** (ovvero la coda dei processi pronti all'esecuzione), quello che deve essere eseguito. Nel caso di **core multipli**, vengono scelti più processi.

Qual è il momento in cui avvengono le decisioni per scegliere il processo da eseguire?

I momenti per scegliere il processo sono quattro:

1. Un processo cambia il suo stato da **running** a **waiting**:
   C'è un processo nello stato **running**, ovvero in esecuzione nella CPU. Per cambiare lo stato in **waiting**, il processo dovrebbe effettuare un'operazione di I/O, o dovrebbe deschedularsi volontariamente.
   In questo caso il processo lascia lo stato running, e quindi la CPU diventa libera per eseguire un altro processo; lo scheduler deve entrare in funzione.
2. Un processo **termina:**
   Il processo nello stato **running** effettua la system call **exit()**, che permette al processo di terminare. La CPU resta quindi libera, ed è anche questo il caso in cui lo scheduler dovrebbe entrare in gioco.

Scheduler nonpreemptive

Se lo scheduler entrasse in gioco solo in questi due casi, vorrebbe dire che esso assegna la CPU ad un altro processo, solo quando il processo precedente ha cambiato il suo stato **volontariamente**, o per l'operazione I/O, o perchè ha terminato.

In questo caso, lo scheduler, viene detto **nonpreemptive**, o in italiano, **senza prelazione**. "nonpreemptive" vuol dire che lo scheduler non toglie **forzatamente** la CPU ad un determinato processo, ma semplicemente lo sostituisco quando il processo precedente ha volontariamente lasciato spazio al prossimo.

Scheduler Preemptive

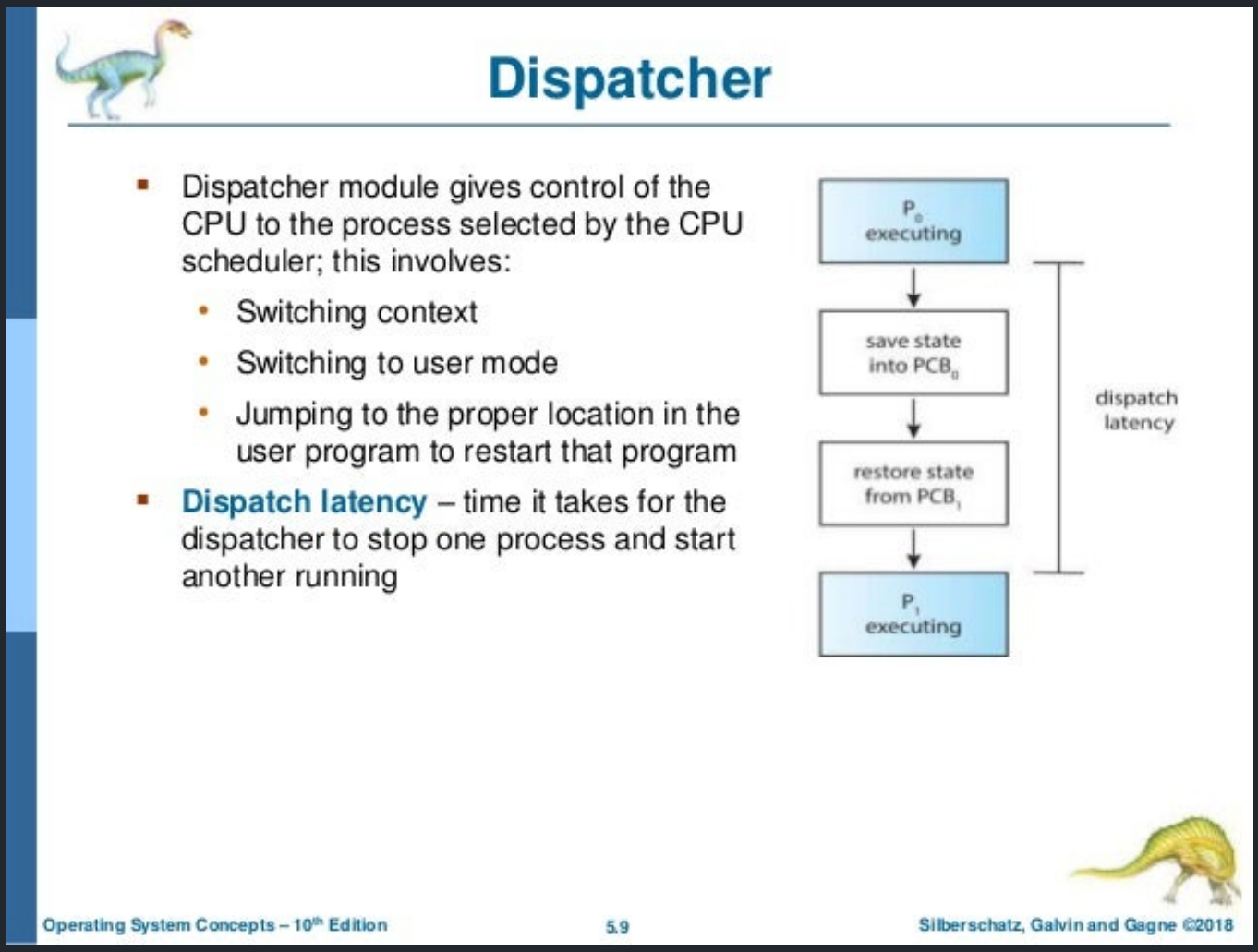
2. Il processo cambia il suo stato da **running** a **ready**:
   Significa che al processo è arrivata un'interrupt, ed è stato **momentaneamente sospeso.** Un altro motivo potrebbe essere che il tempo assegnato all'esecuzione di quel determinato processo è terminato.
   Se per caso lo scheduler prende la decisione di mandare in esecuzione un altro processo, vuol dire che quel processo **poteva** continuare ad essere eseguito, ma è stato bloccato **forzatamente**.
3. Il processo cambia il suo stato da **waiting** a **ready:**
   In questo caso il processo era in attesa di un'operazione I/O, di conseguenza lo scheduler può decidere di dare la priorità ad un altro processo, perchè, ad esempio, quest'ultimo ha una priorità maggiore.

Di conseguenza, lo scheduler che assegna la CPU, stando anche ai punti 3 e 4, è uno scheduler **preemptive**.

In alternativa, uno scheduler **nonpreemptive**, non toglierà mai la CPU ad un processo che non termina o non è in attesa; questo vuol dire che, ad esempio, un processo che è in un loop infinito (o per errore o appositamente), non si vedrà mai tolta la CPU, ma dovremo terminarlo manualmente.

Dispatcher

Il dispatcher è proprio il modulo che sospende un processo e ne manda in esecuzione un altro; esso effettua lo **switch** tra due processi.



Cosa deve fare?

1. Prendere il processo in esecuzione
2. Salvarne lo stato
3. Ripristinare lo stato del processo che riceve la CPU
4. Mandare in esecuzione un altro processo

Latenza del dispatch

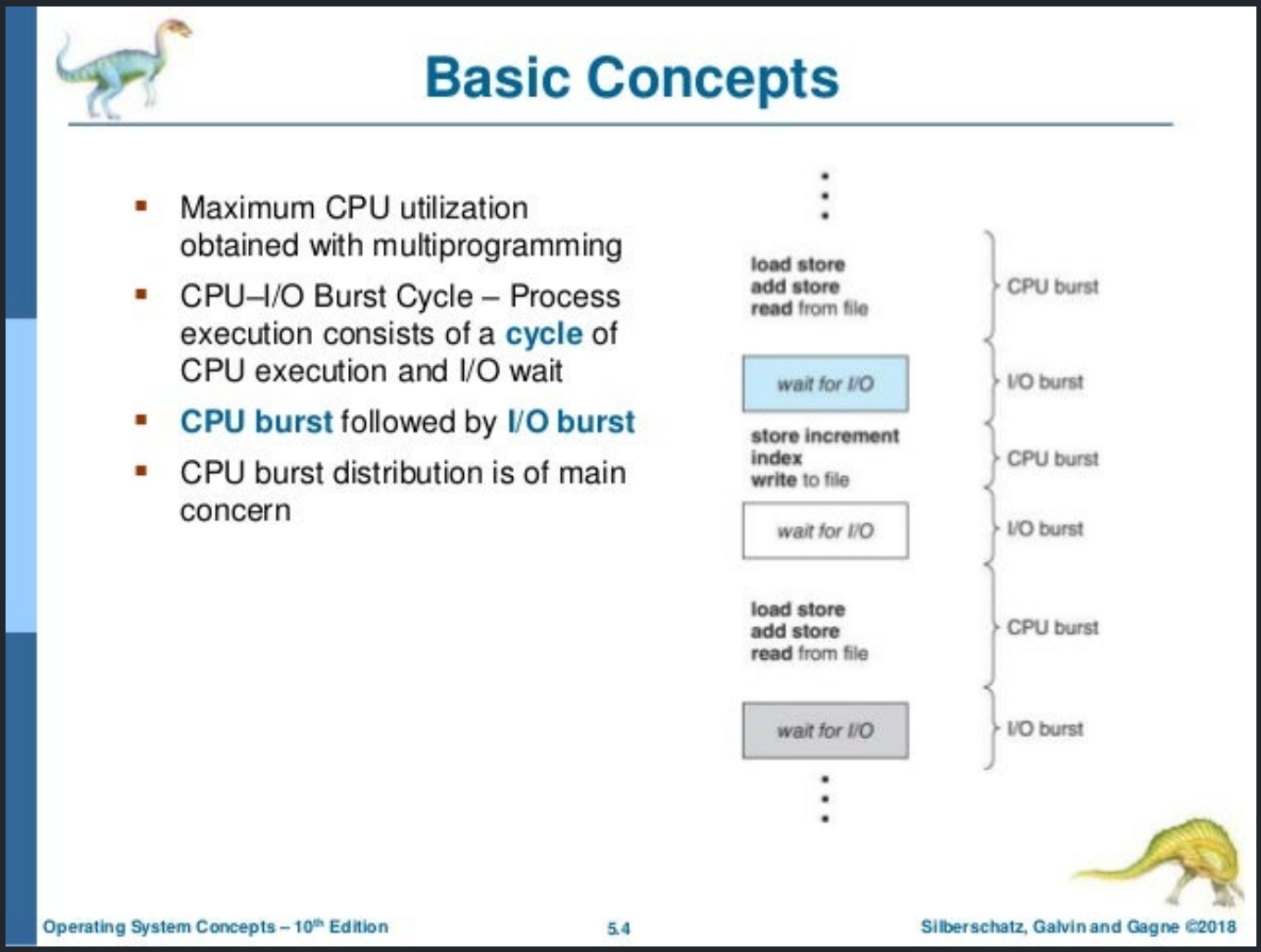
È ovviamete il tempo necessario per effettuare le operazioni elencate precedentemente. È semplice comprendere che i punti 2 e 3, sono totalmente **overhead**, proprio perchè la CPU non sta eseguendo del lavoro utile (ovvero eseguire processi), ma sta "perdendo tempo". Per questo motivo queste operazioni vanno fatte nel tempo più veloce possibile.

Bisogna chiarire che la **latenza di dispatch** vera è propria, è calcolata a partire **dopo** il punto 1 e **prima** del punto 4, quindi il tempo per eseguire i punti 2 e 3.

fine lezione 6

CPU Scheduling

Tutti i processi alternano burst di I/O a Burst di CPU. Questo perché tutti i processi **devono** effettuare delle operazioni di I/O, altrimenti sarebbero inutili. Questo vuol dire che esiste un'alternanza di operazioni interne alla CPU ed esterne.



Dispatcher

È quel modulo del sistema operativo che effettua lo switch tra due processi. Il dispatcher è rigorosamente **scirtto in assembly**, per permettergli di maneggiare i registri. La **latenza di dispatch** è il tempo necessario per effettuare lo switching tra due processi, ed esso dipende da diversi fattori come i registri da salvare, ecc.

Criteri di Scheduling

I criteri che andremo a vedere, sono dei criteri che non vengono applicati ai sistemi da noi comunemente utilizzati, ma in generale fanno riferimento ad un processo che viene eseguito su un **generico** sistema di calcolo.

Sicuramente decidere qual è il processo che deve essere assegnato alla CPU, è un problema di **ottimizzazione**; bisogna infatti scegliere il processo che ci permette di ottenere i migliori risultati.

- **Utilizzo della CPU**: La CPU **deve** essere tenuta più occupata possibile.
- **Throughput:** lo scheduling deve consentire di massimizzare il numero di processi che vengono eseguiti per **unità di tempo.**
- **Turnaround time:** tempo necessario per eseguire un particolare processo, ovvero il tempo calcolato da quando un processo entra nel sistema a quando termina.
- **Tempo di attesa**: tempo in cui il processo è in attesa la coda ready, ovvero la coda dei processi che attendono il proprio turno di essere eseguiti dalla CPU.
- **Tempo di risposta:** Nei sistemi odierni, più che i quattro punti precedenti, si ha interesse nell'ottimizzare il tempo di risposta. Questo vuol dire che bisogna ottimizzare il tempo da quando viene sottomessa una richiesta, fino a quando questa richiesta viene effettivamente eseguita.

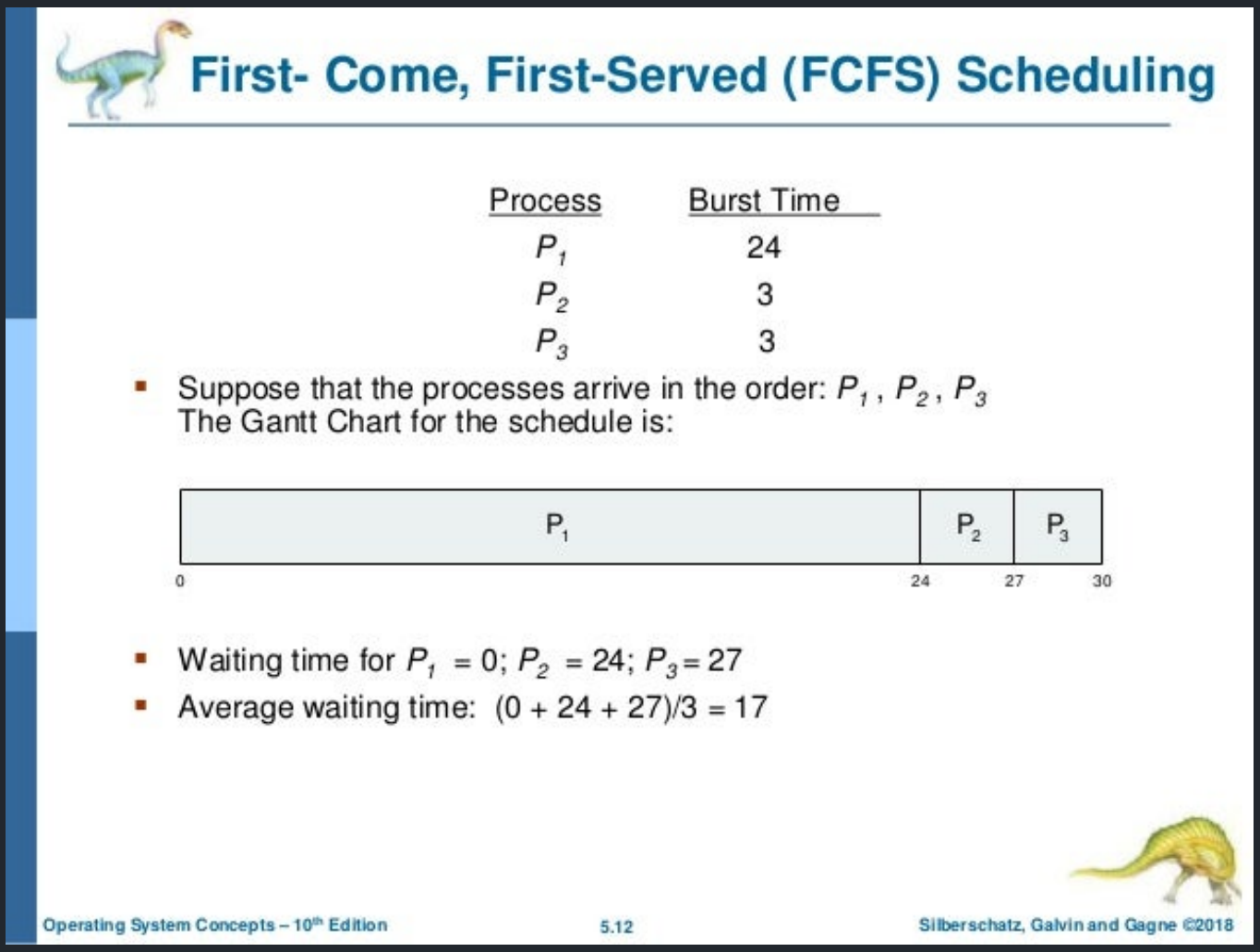
In parole spicciole, all'utente non interessa quanto tempo impiegherà **l'intera** procedura, ma interessa essere **servito** nel minor tempo possibile, anche se la sua richiesta non viene soddisfatta interamente.

Obbiettivi dei criteri:

- **Massimo** utilizzo della CPU
- **Massimo** throughput
- Turnaround time **minimo**
- Tempo di attesa **minimo**
- Tempo di risposta **minimo**

Tecniche di scheduling

First-Come, First-Served (FCFS)



È essenzialmente un **coda**: basti pensare alla coda di un supermercato, è irrilevante che una persona abbia il carrello più pieno di un'altra, verrà servita in ordine di arrivo.

Processo	Tempo Di Burst
P1	24
P2	3
P3	3

Supponiamo che i processi P_1, P_2, P_3 arrivino in ordine. Il **Diagramma di Gantt** sarà il seguente (vedi immagine). Il diagramma di Gnatt è essenzialmente un diagramma unidimensionale, dove sull'asse delle X viene posto il tempo.

Se il processo P1 arriva per primo, vorrà dire che, rispettivamente, P2 e P3 attenderanno 24 e 27 unità di tempo, prima di essere eseguiti. Quindi il tempo di attesa per ogni processo sarà:

- P_1 : 0
- P_1 : 24
- P_1 : 27

Ed il tempo di attesa medio sarà: **(0 + 24 + 27) / 3 = 17**

Se invece eseguissimo i processi in un ordine diverso, ovvero eseguendo per primo il processo "più piccolo", potremmo velocizzare i tempi di attesa:

Eseguendo in ordine: P_2 , P_3 : , P_1 otteniamo un tempo di attesa:

- P_1 : 6
- P_2 : 0
- P_3 : 3

Con un tempo di attesa medio: **(6 + 0 + 3) / 3 = 3**

Morale della favola

Il tipo di scheduling FCFS è uno scheduling completamente inefficace. Questo perchè se davanti a dei processi con un tempo di esecuzione bassa, viene posto un processo che impiega molto tempo, i diversi processi dovranno attendere molto tempo per essere eseguiti, e quindi la CPU si focalizzerà su un unico processo, invece che su molti.

Effetto convoglio

Succede quando abbiamo dei processi "mischiati", tra CPU-bound e molti I/O-Bound.

`Processi di tipo CPU bound`: Sono dei processi che utilizzano la CPU con lunghi burst, ad esempio processi di tipo scientifico.

`Processi di tipo I/O bound`: Ad esempio processi interazioni su database, dove i tempi di calcolo sono molto bassi, ma il tempo di attesa per le operazioni di I/O è molto alto.

Che succede se con una tecnica FCFS si mischiano questi due tipi di processo? Abbiamo lo stesso effetto di camion su una strada ad una singola corsia: mi trovo un camion che mi precede molto lento, l'equivalente di un processo che utilizza la CPU per molto tempo.

Scheduling Shortest-Job-First (SJF)

Bisognerebbe, ad ogni scelta, eseguire sempre il processo che richiede un tempo di CPU più basso. Se in qualche modo riuscissi a sapere la lunghezza del prossimo **burst di cpu**, potrei utilizzare questa informazione per assegnare la CPU al processo che ha il CPU burst più corto.

Questa tecnica è ottimale, ma il problema è che non è molto semplice conoscere la lunghezza del prossimo burst di CPU, infatti non c'è nessuna informazione che possa darmi questo tempo di burst.

Esempio di SJF

Facciamo finta di avere questa informazione, i processi P_1, P_2, P_3, P_4 hanno rispettivamente un tempo di burst pari a 6, 8, 7, 3.

Dovremmo quindi ordinare i processi nell'ordine: P4(3) -> P1(6+3) -> P3(6+3+7) -> P2(6+3+7+8). Abbiamo quindi un tempo di attesa medio pari a : **(3 + 16 + 9 + 0) / 4 = 7**

Come determinare il prossimo burst di CPU

Non è possibile determinare al 100% il prossimo burst, ma i processi diventano prevedibili, ed a comportarsi sempre allo stesso modo. Serve quindi un modo semplice per stimare il burst di CPU in base al comportamento passato di un processo.

Questa stima deve essere fatta in maniera tale da tenere presente sia il comportamento passato, sia il comportamento **presente**, ovvero devo tenere presente maggiormente cosa è successo **nell'immediato passato**, ed un po meno cosa è successo nel **passato remoto** del processo.

Possiamo prevedere il tempo di burst utilizzando la media dell'esponenziale:

1. τ_n = reale lunghezza dell n^{th} CPU burst
2. τ_{n+1} = valore previsto per il prossimo burst CPU
3. α, $0 < \alpha < 1$
4. Definiamo: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$

Questo significa che ogni volta devo conservarmi la previsione precedente, la combino con la formula con ciò che è stato effettivamente misurato, e con questa tecnica trovo la lunghezza prevista per il reale tempo di burst.

Vari casi:

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - La storia recente non conta
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Solo l'ultimo burst di CPU conta.
- Se espandessimo la formula, otterremmo:

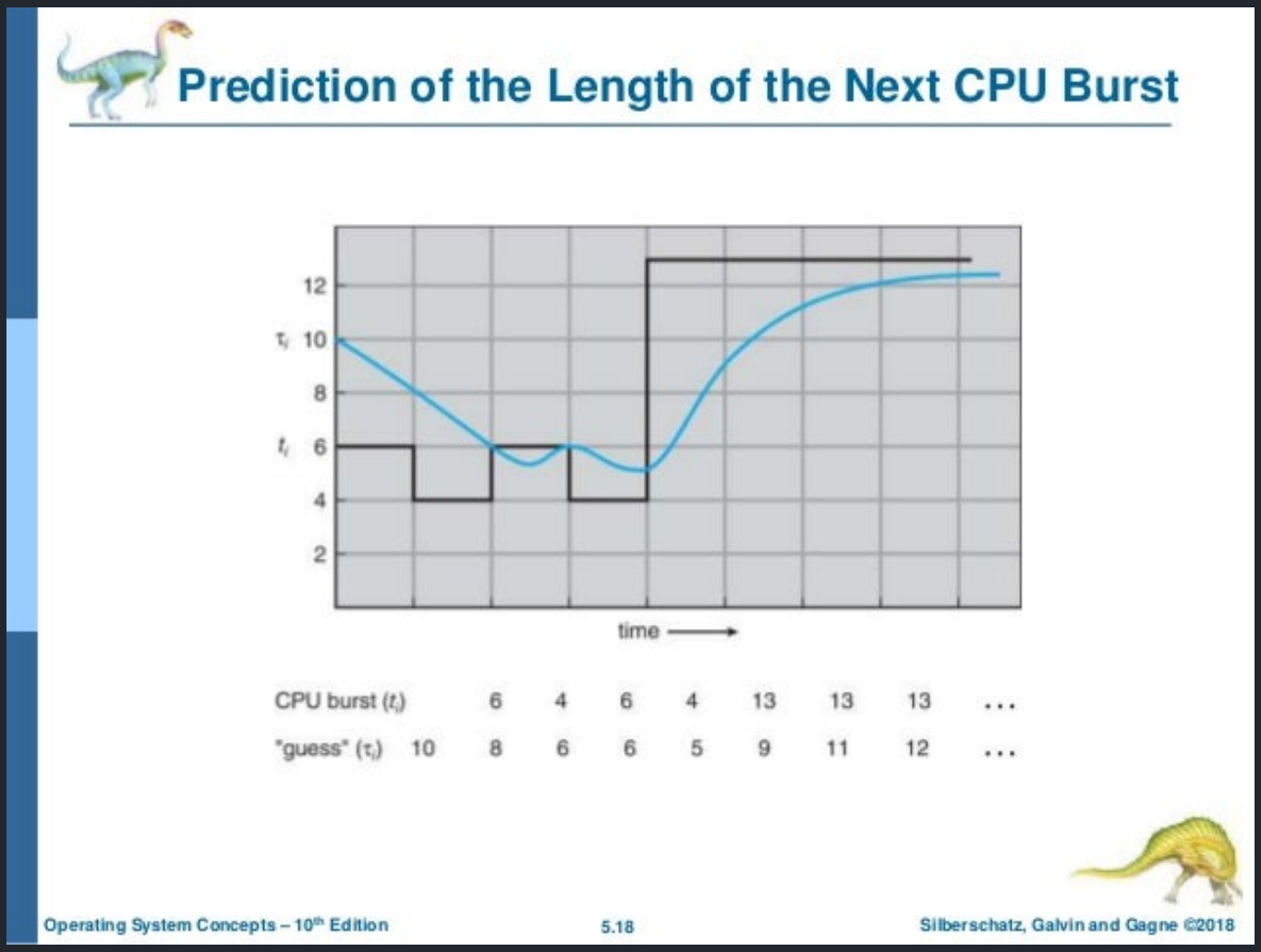
  $\tau_{n+1} = \alpha t_{n+1} + (1 - \alpha)\, t_{n-1} + \ldots$

  $(1 - \alpha)^j\, t_{n-j} + \ldots$

  $(1 - \alpha)^{n+1}\, t_0 + \ldots$

  Guardando la formula, siccome $1 - \alpha$ è un valore minore di 1, che viene elevato sempre ad un numero crescente, otteniamo che teniamo conto di ciò che è accaduto in un passato remoto via via sempre di meno.

Non è necessario usare la formula, infatti ad ogni passo devo semplicemente prendere t_n, moltiplicarlo per un numero (magari dividerlo per due visto che basta uno shift); a questo numero basta aggiungere il tempo precedente t_n e moltiplicarlo anche lui per un numero (sempre diviso due). Di conseguenza non serve una grande potenza di calcolo per eseguire questo calcolo.



In questa slide si può vedere come la previsione (in celeste) si adatta al tempo reale di CPU burst. È inevitabile che per qualche tempo la previsione sarà errata, come ad esempio nelle prime due sezioni del grafico (lungo il tempo).

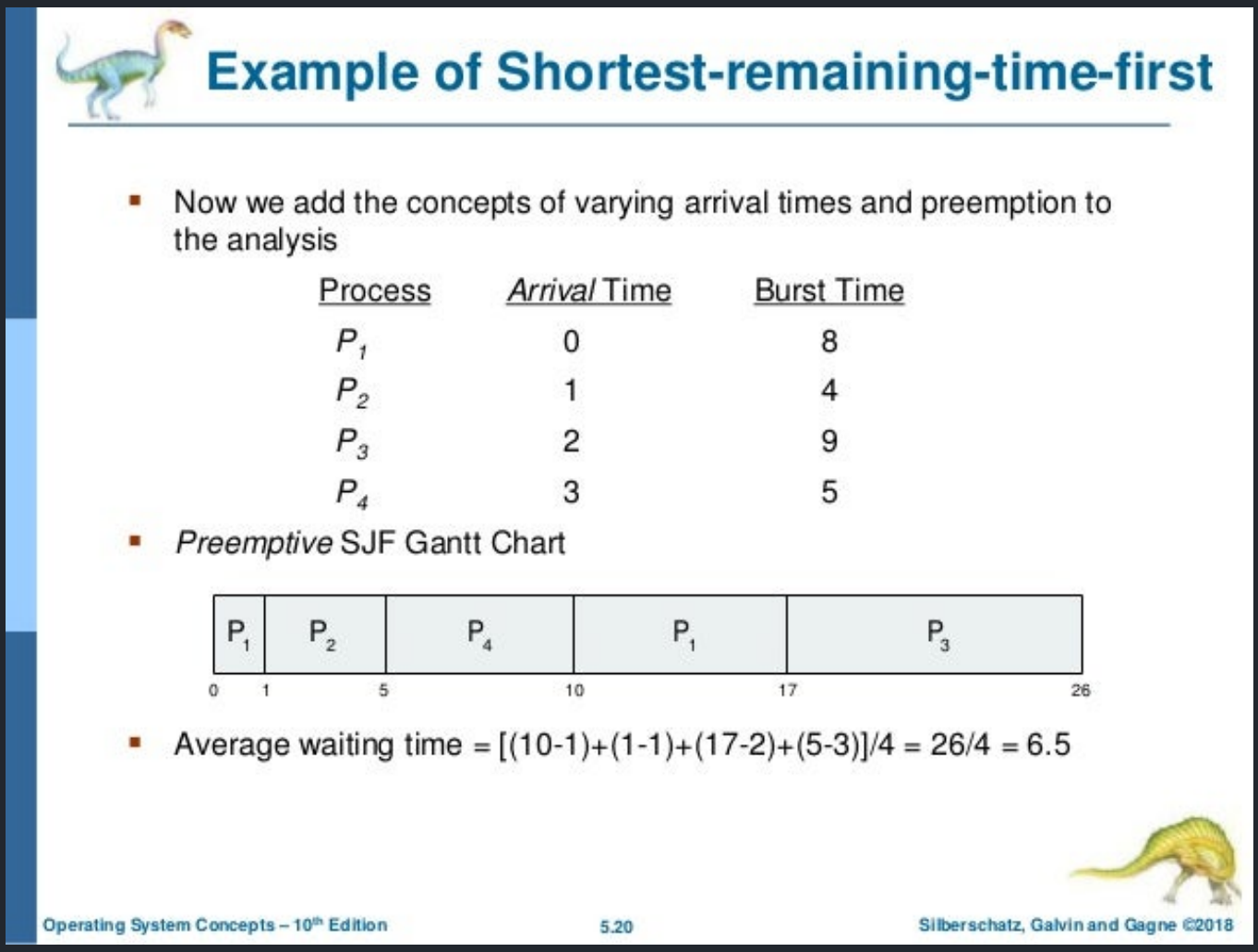
Preemptive vs nonpreemptive

Versione Nonpreemptive

Questo tipo di scheduling è non preemptive, infatti i processi vengono eseguiti interamente prima di passare ad un altro processo.

Versione Preemptive : Shortest-remaining-time-first

Nel momento in cui arriva un processo mentre la CPU è assegnata ad un dato processo, ma questo nuovo processo ha bisogno di un **burst più piccolo rispetto a quello che il processo in esecuzione deve ancora fare**, allora lo scheduler toglie la CPU al processo corrente, e la assegna al processo appena arrivato, tecnicamente più veloce da completare.



Nei Sistemi interattivi, queste tecniche di scheduling sono ragionevoli?

Risposta: assolutamente no.

Nei sistemi interattivi, come quelli che vengono utilizzati nei nostri personal computers, vogliamo avere l'illusione che tutti i processi vengano eseguiti allo stesso momento. Questo significa che a **nessun processo** deve essere permesso di utilizzare la CPU per un lungo periodo di tempo, proprio perchè lascerebbe tutti gli altri processi "bloccati".

Scheduling adottato comunemente : Round Robin (RR)

Nei sistemi interattivi sorge spontanea la soluzione di utilizzare uno scheduling a **suddivisione di tempo**, ovvero ogni processo ha una **fetta minima di tempo**, detta **time slice** che è il tempo massimo che può trascorrere con la CPU; questo tempo varia dai 10 ai 100 millisecondi (ms).

Dopo che questo tempo è esaurito, al processo viene tolta la CPU (quindi questo è uno scheduling di tipo **preemptive**) ed essa viene attribuita ad un altro processo.

Significa che in condizioni ottimali, se ogni processo ha 10ms, ogni secondo ho l'esecuzione di 10 processi diversi; all'occhio umano si ha l'impressione che tutti i processi vengono eseguiti in contemporanea.

Bisogna notare che non tutti i tipi di processo vengono schedulati per la stessa quantità di tempo, infatti alcuni processi vengono eseguiti per una piccola fetta di tempo, mentre altri per più tempo; tuttosommato si parla dell'ordine di millisecondi.

Quanto deve attendere un processo?

Se abbiamo nella coda ready **n processi**, ed il time slice è **q**, allora ogni processo riceve **1/n** del tempo della CPU in chunk di al massimo **q time** unità in una volta.

Questo vuol dire che nessun processo (nel caso peggiore) attende più di **(n-1) q** time units. In parole semplici, al massimo un processo deve attendere il completamento di tutti i processi (eseguiti al massimo per il time slice definito) che sono in coda prima di lui.

Performance

Se il time slice **q** è molto grande, essenzialmente ogni processo riesce a concludere il suo burst in un'unica esecuzione, e di conseguenza si ricade nella situazione FIFO, dove il primo processo che arriva è il primo ad essere servito.

Si ha quindi interesse a porre **q** abbastanza piccolo (ma non troppo perchè qualora q fosse davvero molto piccolo, aumenterebbe lo switching di contesto [context switch] e di conseguenza l'**overhead**).

Priority Scheduling - basato su priorità

Viene stabilito un meccanismo a priorità, si associa ad ogni processo un numero (intero) che esprime la priorità del processo, e la CPU viene assegnata al processo avente la priorità più alta.

Per motivi storici, la priorità più alta, significa numero effettivo più basso! Ovvero a numeri più bassi viene associata una priorità più alta.

Starvation

In un sistema in cui c'è un arrivo costante di processi a media-alta priorità, un processo avente una bassa priorità non verrà **mai eseguito**. In qualche modo bisognerebbe fornire un qualche tipo di meccanismo che eviti questo fenomeno.

Come si supera questo problema?

Tutti i sistemi reali prevedono una soluzione basata sul meccanismo di **aging**: le priorità non vengono più assegnate in maniera **statica**, ma via via che un processo attende, la sua priorità aumenta; in questo modo, prima o poi, il processo verrà eseguito.

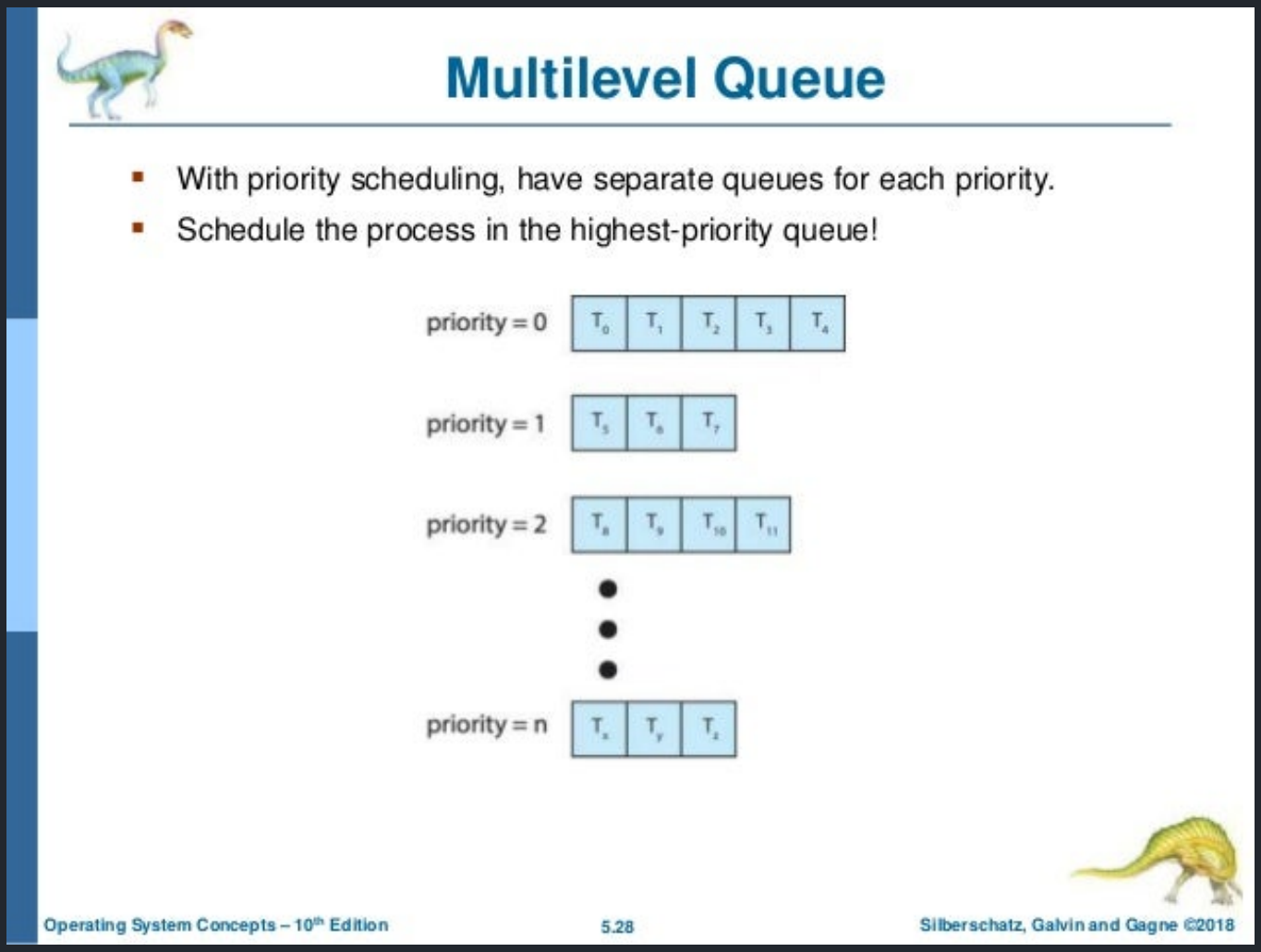
Preemptive vs nonpreemptive

Di questa tecnica, ovviamente, esiste sia la versione preemptive che nonpreemptive; nella versione preemptive, esiste si la priorità per ogni processo, ma comunque esiste un time slice dopo il quale un processo deve essere sostituito da un altro. Questo vuol dire che a **priorità uguale**, si procede con la tecnica del **Round Robin**.

Questo vuol dire che se abbiamo due processi con la stessa priorità, questi si alterneranno fino a quando uno (o entrambi) avranno terminato.

Multilevel Queue

L'idea è quella di avere delle code di processi separati, aventi priorità diverse:



Scheduliamo i processi che si trovano nella coda a più alta priorità in preferenza rispetto ad altri.

Multilevel feedback queue

Più tempo di CPU viene consumato da un processo, più il processo (o meglio la sua priorità) viene degradato, in modo che esso verrà eseguito meno spesso.

Thread Scheduling

I thread all'interno di un processo potrebbero visti come un tutt'uno con il processo stesso, quindi ricevendo lo stesso time slice di quel particolare processo, oppure potrebbero essere schedulati **separatamente**.

Process contention scope (PCS)

In questo contesto viene assegnato un tempo di CPU al processo, e poi all'interno del processo il time slice viene suddiviso tra i vari thread.

Questa tecnica serve per evitare che un processo avente molti thread utilizzi tutto il tempo di CPU a discapito degli altri processi.

System contention scope (SCS)

In questo caso il thread viene schedulato come se fosse un processo a se stante.

Tra i vari parametri che possiamo aggiungere al momento di attivazione di un thread, è possibile aggiungere anche un parametro che dice al sistema se il thread deve essere schedulato in PCS o in SCS.

Scheduling di processori multipli

🏁 05-13 01:03

Cosa succede in un sistema con più CPU?

Sono disponibili dei CPU Scheduler più complessi nel momento in cui più CPU sono disponibili. Possiamo avere una delle seguenti architetture:

- CPU multicore
- Core multithreaded: CPU singola che si riesce a dividere in più cores **virtuali** che vengono trattati come se fossero core reali della cpu.
- NUMA Systems
- Sistemi eterogenei

Sostanzialmente, lo scheduling su core multipli è una tecnica di scheduling nata con i sistemi **SMP**, ovvero i sistemi dove è presente un **multiprocessing simmetrico**.

Common ready queue - coda comune

Con questa tecnica lo scheduler pesca i thread/processi da mandare in esecuzione su diversi core da un'unica lista di ready, dove sono posti **tutti** i processi.

Pur essendo una soluzione semplice da implementare non è una grande soluzione; questo perchè una volta che un processo T0 è andato a finire sul core0, alla prossima operazione di scheduling potrebbe andare a finire su un core diverso.

Questo vuol dire che nel primo core utilizzato dal processo, è presente una cache con dati del processo, mentre se cambia core, nelle cache non troveremo dati del processo in esame.

Per-core run queues (soluzione meglio)

Con questa tecnica, abbiamo **diverse code** dove sono presenti dei processi già suddivisi, ed ogni coda verrà eseguita su un unico core della CPU. In questo modo, la cache di un processo rimane presente nel core su cui il processo è già stato eseguito precedentemente.

Cosa succede quando ad un core vengono assegnati pochi processi?

È possibile che, con la tecnica per-core, ad un core vengano assegnati pochi processi rispetto agli altri core. La soluzione è quella di, ad intervalli di tempo, **ridistribuire il carico**.

Ping-Ponging

Questo è un fenomeno che si verifica nel momento in cui effettuiamo una ridistribuizione di carico tra i vari core; poniamo il caso che il core0 abbia pochi processi in esecuzione, e che il core1 invece ne abbia molti. Il core0, però, ha molti processi in attesa (magari per operazioni I/O).

Di conseguenza, se ridistribuisco il carico dal core1 al core0, il core0 si vedrà dapprima un gran numero di processi provenienti dal core1, e poi, una volta che le operazioni I/O dei suoi processi in attesa hanno completato, si vedrà arrivare anche questi altri processi.

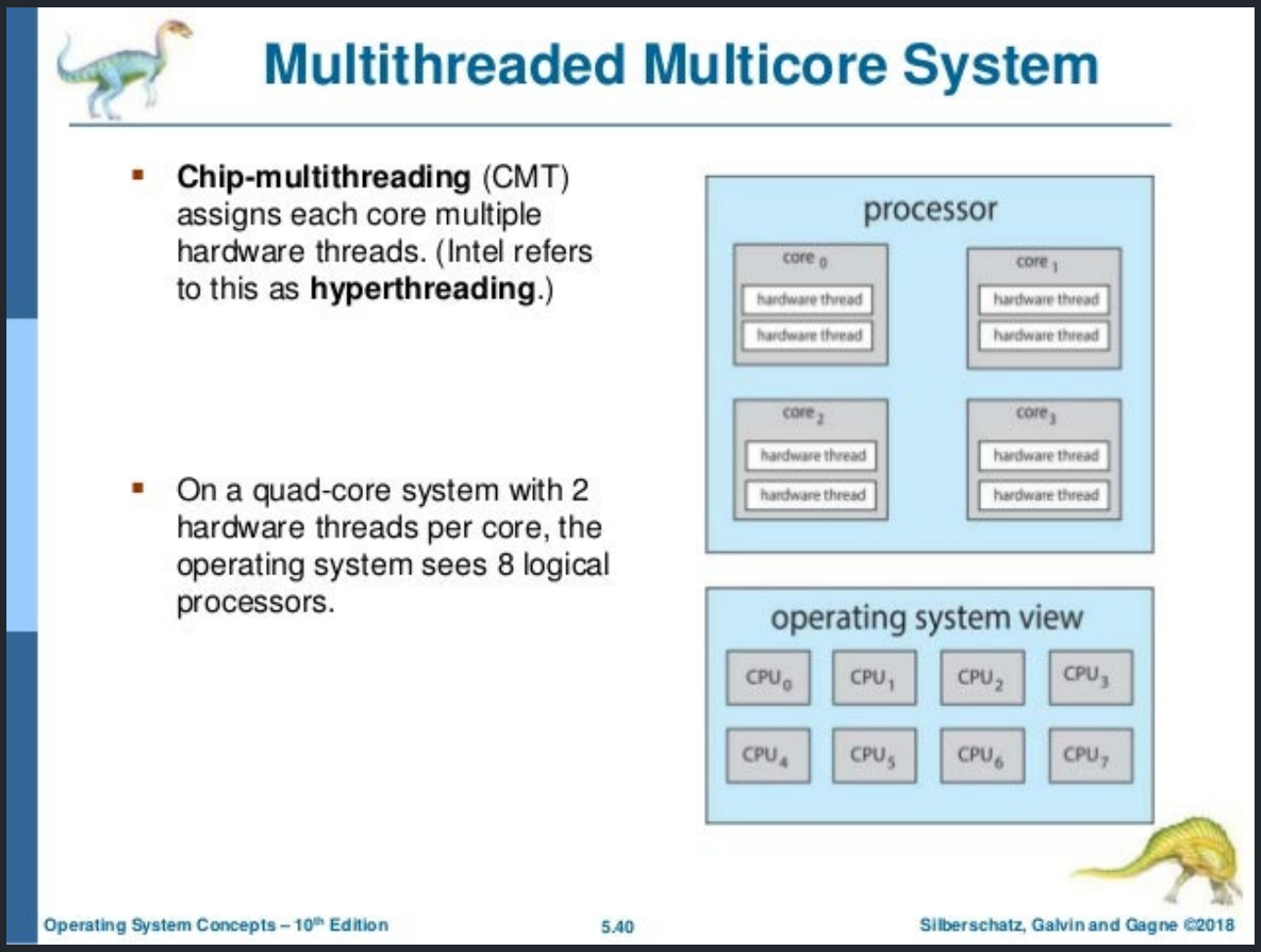
Quindi, dobbiamo effettuare **ancora una volta** la ridistribuizione di carico, e questo è noto come l'effetto di ping-ponging.

Può convenire "spegnere" un core?

Tutti i sistemi moderni, tendono ad aumentare la **frequenza** quando è presente un unico core occupato, mentre gli altri core sono "scarichi". Almeno su alcuni tipi di sistemi, si potrebbe avere interesse nell'eseguire tutti i processi su un core, in modo da ridurre i consumi energetici.

Sistema multithread multicore

In un sistema di questo tipo, abbiamo un processore con più cores (ad esempio un quadcore - 4CPU), che a loro volta vengono divisi in più **hardware thread**:



Il sistema, però, visualizza ogni thread come un processore fisico, andando quindi a visualizzare il **doppio** dei processori reali.

Processor affinity

Quando un thread viene eseguito su un processore, nel contenuto della cache di quel processore specifico, viene salvata la memoria acceduta da quel thread. Ci riferiamo a questo concetto come **affinità per un processore**.

- **Soft affinity** : Il sistema operativo cerca di mantenere l'esecuzione di un processo sullo stesso processore, ma non garantisce nulla.
- **Hard affinity:** consente al processo di specificare un insieme di processore dove esso potrà essere eseguito.

Real-Time CPU scheduling

Real-Time significa che in qualche modo devono essere rispettati dei vincoli di tempo. Esistono due tipi di sistemi:

- **Soft real-time** : questi sistemi cercano di eseguire i **task real-time critici** con la priorità maggiore, ma non garantiscono nulla.
- **Hard real-time:** i **task devono** essere serviti entro la loro **deadline** .

Scheduling basato sulla priorità

Per uno scheduling real-time, lo scheduler deve supportare il **preemptive**. I processi hanno una nuova caratteristica: quelli **periodici** richiedono la CPU ad intervalli costanti.

Scheduling earliest deadline first (EDF)

In questo tipo di scheduling, le priorità vengono assegnate a seconda delle **deadlines:**

Più la deadline è vicina, più la priorità del processo sarà alta.

Scheduling Linux

Tutti i SO, prevedono code di processi ordinate secondo priorità, dalle quali deve essere estratto velocemente il prossimo processo per ridurre quanto più possibile il tempo di **context switch** (overhead).

Si ha interesse ad avere delle **strutture dati** che supportano al meglio questo tipo di operazione.

Il tempo di scheduling di linux > 2.5 è di **ordine costante**, ovvero **O(1)**.

Little's Formula

- n = lunghezza media della coda
- W = tempo di attesa medio in coda
- λ = ritmo medio di arrivo nella coda

Questa semplice legge, ci dice che in codizioni di regime, la lunghezza media di una coda è data dalla formula:

n = λ x W

Per esempio, se con una media di 7 processi in arrivo per secondo, e normalmente 14 processi in coda, il tempo di attesa medio per processo è **2 secondi**.