Capitolo 10: Memoria Virtuale

Table of contents

Nei sistemi moderni non si utilizza ne la paginazione ne la segmentazione, ciò che realmente viene usato è la **memoria virtuale suddivisa a pagine.**

Background

Il programma per poter essere eseguito deve essere caricato in memoria centrale del sistema, perchè dal disco non può essere eseguito nulla.

La domanda sorge spontanea: serve che **tutto** il programma venga caricato in memoria?

- Probabilmente ci sono dei "pezzi" che non mi serviranno: ad esempio nel programma c'è una porzione di codice che serve ad un compito specifico che non viene utilizzato spesso, non è meglio lasciarlo sul disco e portarlo in memoria solo quando serve? ovviamente si.
- Mi serve tutto il programma **contemporaneamente** in memoria? Ovviamente no.

Se riuscissi a portare in memoria solo dei "pezzi" di programmi così grandi, il vantaggio sarebbe quello di non **sprecare spazio in memoria** con porzioni di codice che probabilmente non verrà mai eseguito. Come conseguenza, ho il fatto che posso mandare in esecuzione **più programmi in esecuzione**.

L'idea

L'idea, quindi, è quella di avere la maggior parte del programma sul disco, e portare solo **su richiesta** in memoria la parte di codice che realmente mi serve in quel momento.

Questa può essere vista come **una forma di caching**.

Tutti i sistemi operativi utilizzano questa tecnica per il caricamento dei programmi in memoria.

I vantaggi

I vantaggi sono:

- Ogni programma occupa meno spazio durante la sua esecuzione, quindi posso **eseguire un maggior numero di programmi.**
- E' richiesta una minore quantità di operazioni I/O per caricare e scambiare programmi nella memoria, quindi **ogni programma utente gira più velocemente** .
- Visto che in memoria vengono caricate solo delle porzioni di programma, **posso avere un programma anche più grande della memoria fisica a disposizione!**

In parole povere

La memoria virtuale realizza l'estrema separazione tra la vista **logica della memoria** e l'indirizzo **fisico**; la CPU continua a vedere degli indirizzi **fisici** di memoria, anche se magari quella porzione di codice (o atro) è sul disco.

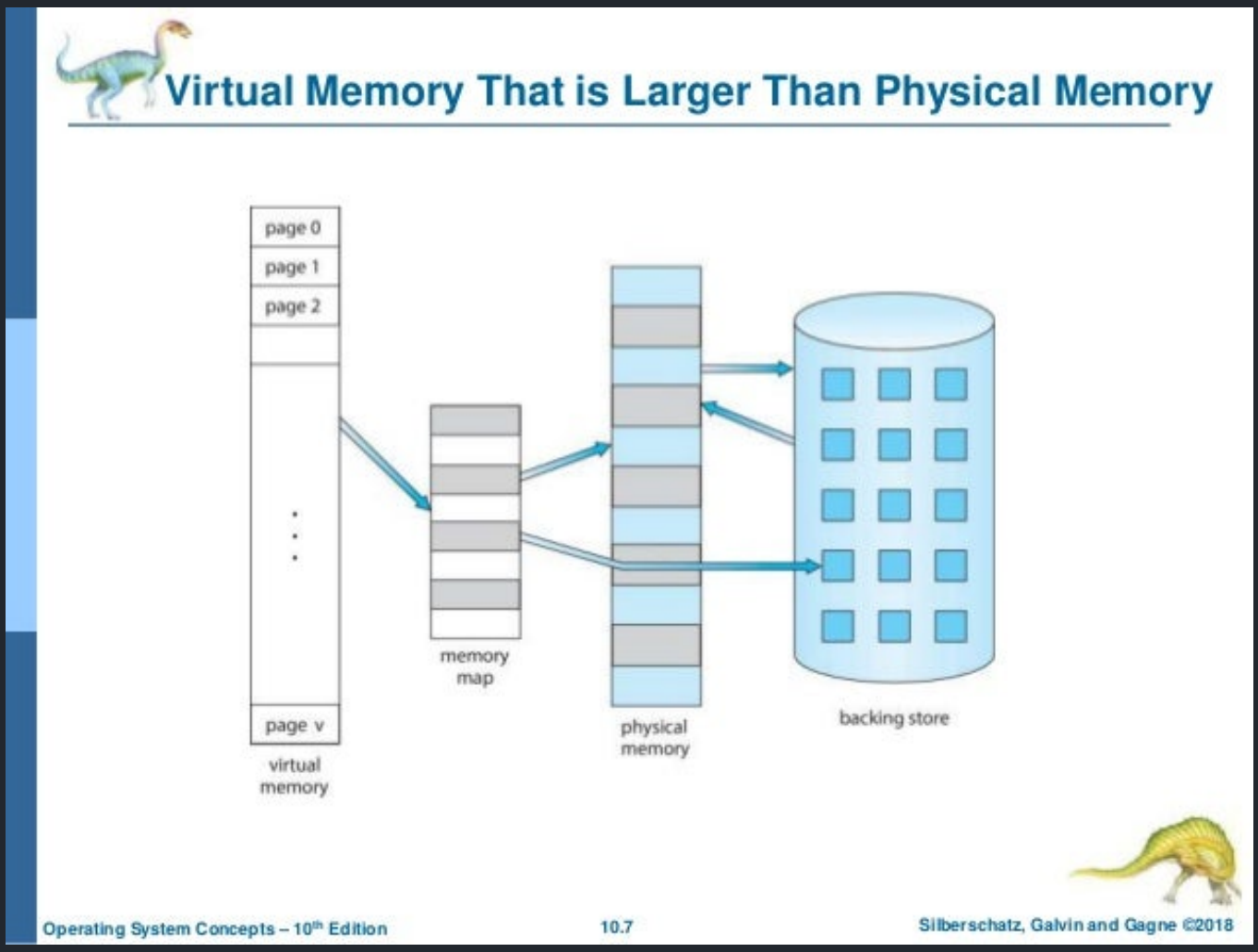
L'idea che solo parte del programma è in memoria, permette che spazio **logico** possa essere **molto più grande** dello spazio **fisico**.

Cosa vede la CPU

La CPU vede uno spazio indirizzi **virtuale**, che solitamente inizia da 0. Nel frattempo, la memoria **fisica** è organizzata in **pagine (quindi frames)**.

La memoria virtuale può essere implementata in due modi (che abbiamo visto precedentemente):

- **Pagine** (soluzione utilizzata nei sistemi odierni)
- **Segmenti**



In questo schema si può notare come la memoria virtuale sia quella esposta, che fa riferimento alla **mappa della memoria**; nella mappa sono presenti gli indirizzi, che potrebbero far riferimento a dei dati realmente presenti nella **memoria fisica**, oppure potrebbero essere **sul disco**.

Si nota inoltre, che c'è uno scambio di dati, ma bisogna notare che i dati sul disco, per essere eseguiti, devono prima arrivare in **RAM**;

Attenzione: questo scambio di dati tra RAM e disco, non è uno **swap**; questo perchè per essere uno swap **tutto lo spazio indirizzi di un processo esce dalla ram, e tutto lo spazio indirizzi di un altro entra.**

In questo caso, quelle che vengono scambiate sono delle **pagine**.

Demand Paging

Una memoria virtuale implementata a pagine (non a segmenti), utilizza il **demand paging**, ovvero la **paginazione su richiesta**; le pagine vengono quindi portate in ram solo quando servono.

Quando una pagina è richiesta?

Una pagina è richiesta quando la CPU deve accedere ad un indirizzo (di memoria) contenuto nella pagina.

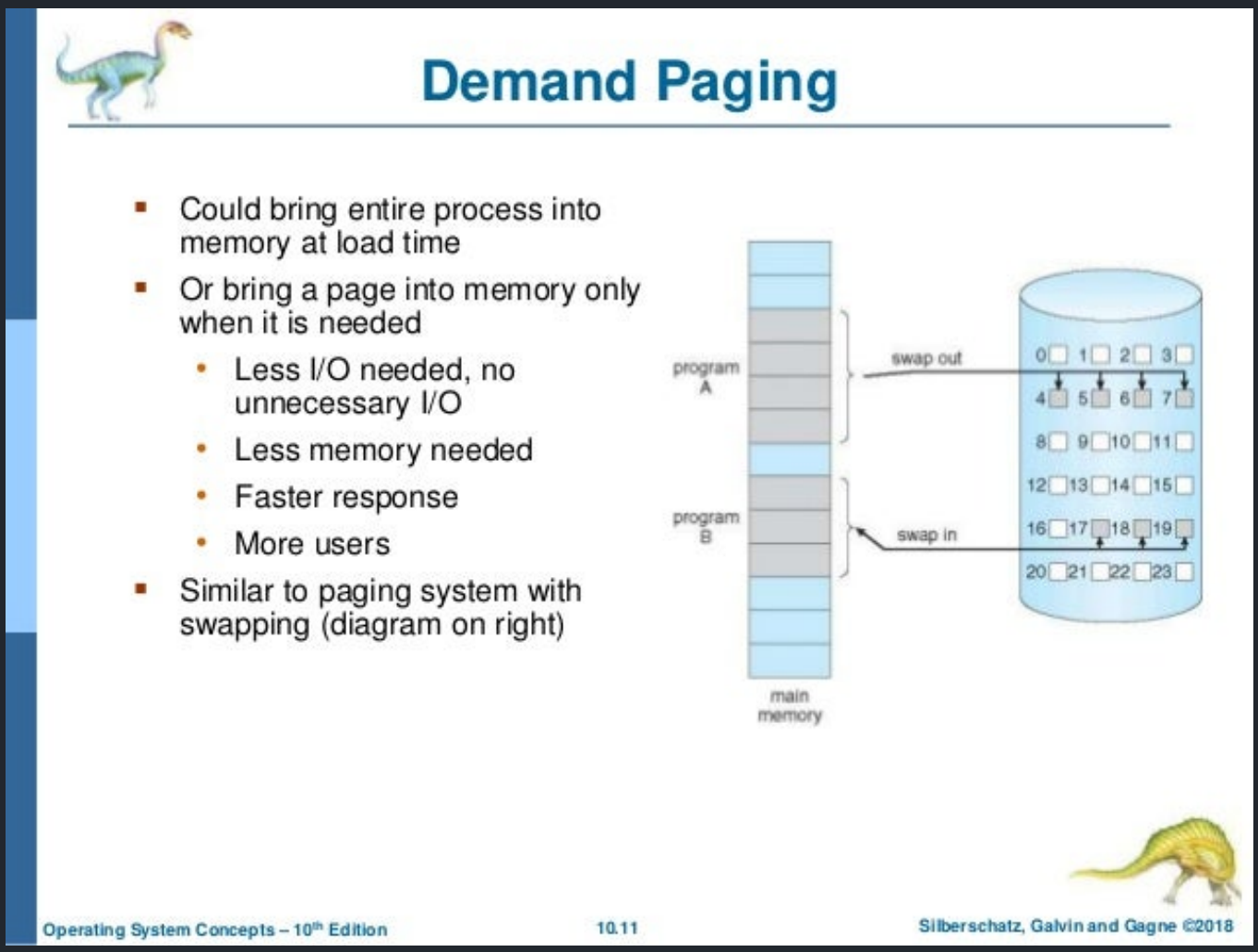
Per le pagine non esistenti, nella tabella delle pagine viene scritto "i" - invalido; questo valore "i" può essere anche utilizzato per le pagine che esistono ma non sono state caricate in memoria, per cui non esiste il numero di pagina (frame) corrispondente.

Il processo è il seguente:

- La CPU rilascia un indirizzo
- L'MMU tenta la traduzione ma nel TLB non trova nulla perchè l'indirizzo non è ancora presente
- Consulta la tabella delle pagine
- Trovato l'indirizzo si reca in quella posizione ma trova "i"; abbiamo due possibilità (determinate dal SO):
    - La pagina non esiste per quel processo
    - **La pagina esiste ma è sul disco e va caricata in memoria** .

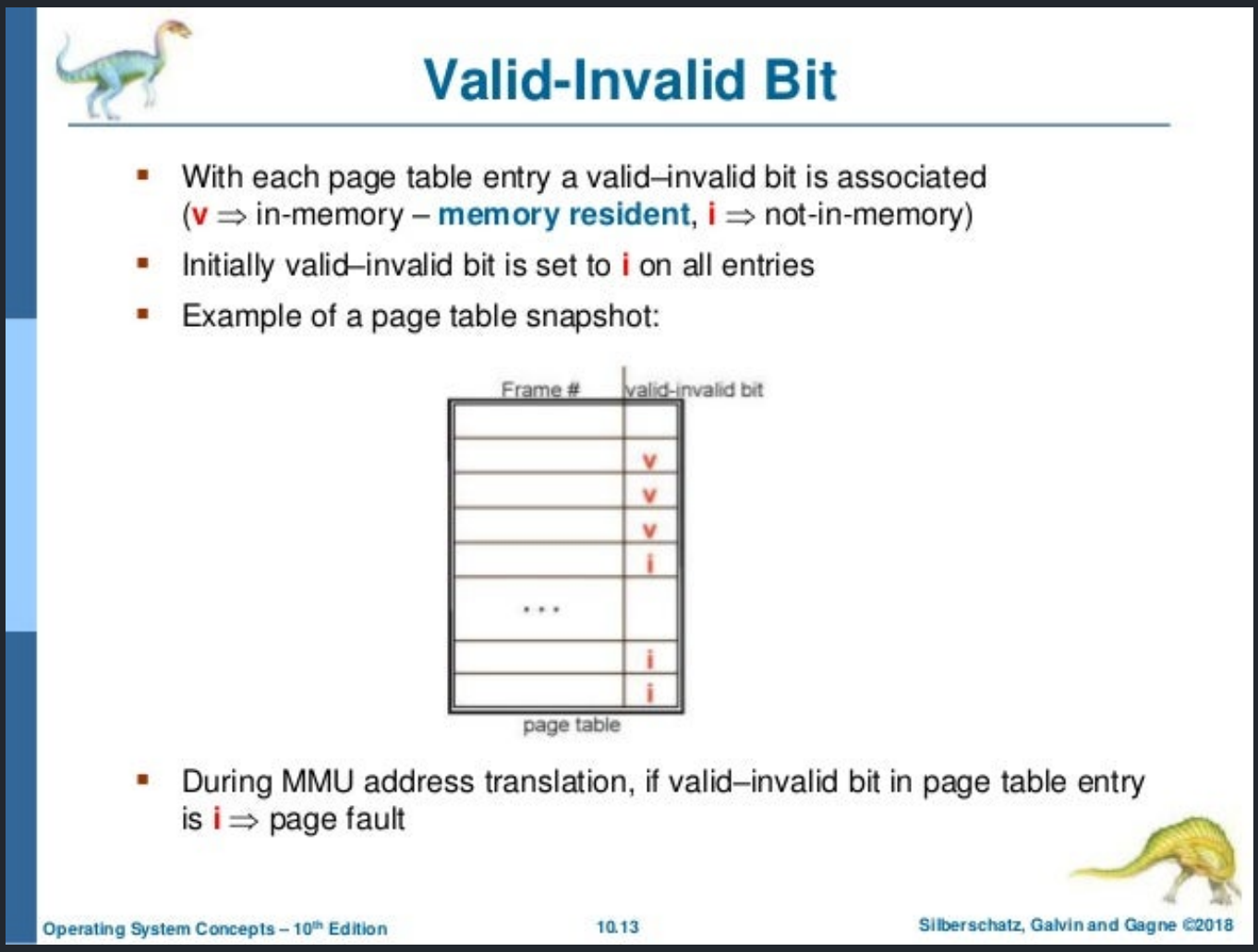
Non si parla di swapping

Come detto anche prima, questi vari processi non sono categorizzati come **swapping**, proprio perchè per essere uno swap, l'intero spazio indirizzi di due processi diversi devono essere scambiati tra loro;



In questa immagine, infatti, si nota che il programma A che viene rimosso dalla ram, ma **non tutto** il programma B viene caricato.

Bit Valid-Invalid

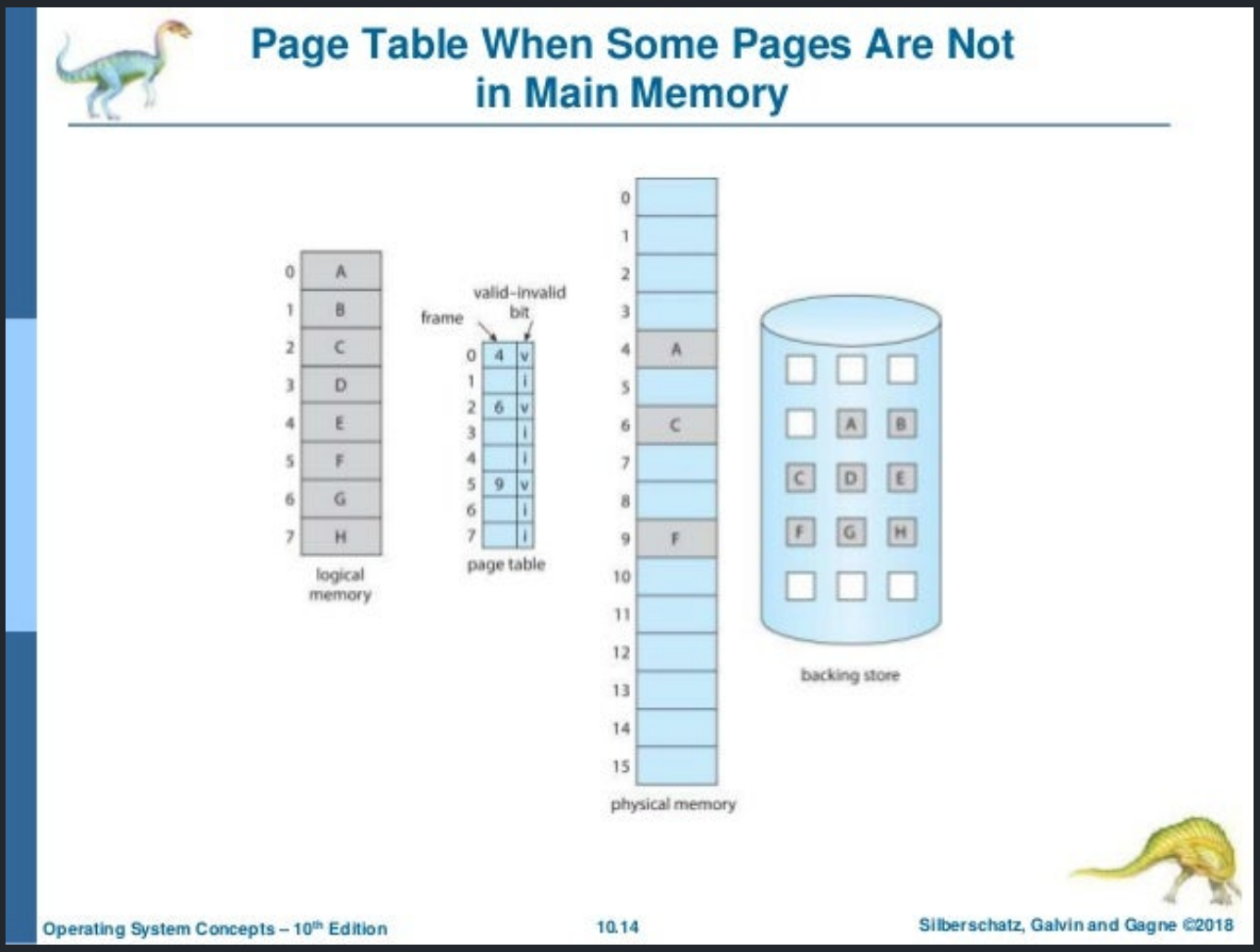


Nella tabella delle pagine è presente il bit **valid-invalid**; quando un frame è marcato come **valid** vuol dire che quel frame è caricato in memoria ed il sistema è perfettamente a conoscenza della sua locazione in memoria.

Se invece un frame è marcato come **invalid** (i), potrebbe essere che quella pagina **non esiste per quel processo**, oppure **la pagina non è stata caricata in memoria**, perchè non utilizzata fino a quel momento.

Quando si effettua la traduzione dell'indirizzo tramite MMU, se quel frame trovato si ha una **page fault**, ovvero un "errore" nella traduzione pagina-frame

Disposizione delle pagine in memoria



Come si vede in questa immagine, le pagine che sono in **backing store**, ovvero sul disco, sono le pagine che sono marcate con "i", invalid; questo **non perchè non esistono, ma perchè ancora non sono state caricate in memoria**.

Come si gestisce un Page Fault

Quando si ha un page fault, a differenza dei sistemi di caching visti precedentemente, si attua un meccanismo misto tra hardware e software: quando si trova un indirizzo marcato con "i", scatta l'intervento del **sistema operativo** (soluzione software).

Cosa fa il sistema operativo in questi casi?

Il SO deve controllare su una sua tabella interna per capire se "i" in quel caso significa che si ha una pagina **inesistente**, oppure non è ancora stata caricata in memoria.

Se la pagina si rivela **inesistente**, il processo **deve essere abortito**, perchè potrebbe aver sfondato un array o altri errori; se ci facciamo caso, quando si programma male e si sfonda un array, si ha come errore **segmentation fault**, anche se non abbiamo più la tecnica della segmentazione! Questo errore deriva da quando i SO adottavano una divisione della memoria a segmenti, e storicamente, è rimasto invariato.

Se invece la pagina si trova ancora sul disco, deve portarla in memoria:

- Deve trovare un frame libero.
- Effettuare lo swap della pagina lanciando **un'operazione sul disco**
- portare la pagina in memoria
- modificare la tabella delle pagine ponendo sia l'indirizzo corretto del frame sia aggiornando il bit da invalid a **valid**.
- Rilanciare l'istruzione che ha provocato il **page fault**.

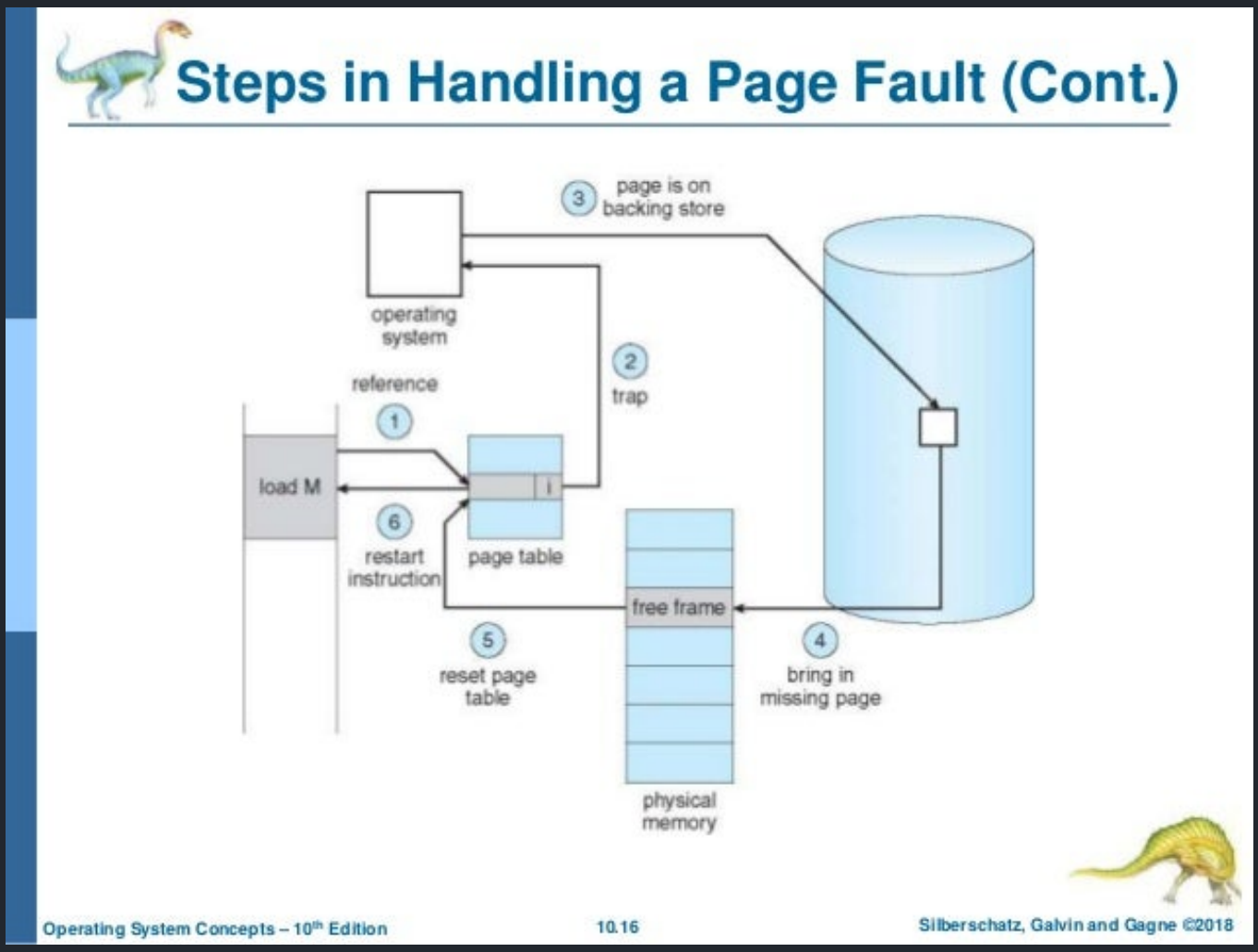
Questo significa che dopo il page fault, l'istruzione deve ripartire come se nulla fosse. Un processore che ha delle istruzioni molto complicate (ovvero che per eseguire una singola istruzione compiono diverse operazioni) non si trovano bene con questa tecnica!

Serve un processore che **non faccia modifiche permanenti** finquando l'istruzione non è stata completata, perchè solo allora si ha la **certezza di poter andare avanti** con la prossima istruzione.

Quindi...

I processori destinati all'utilizzo della memoria virtuale, per via del fatto che con un **page fault** devono riavviare da zero l'istruzione che l'ha provocato, **non effettuano modifiche permanenti** finchè l'istruzione non è stata eseguita.

Passi nella gestione di un Page Fault



1. **Riferimento alla memoria** -> tabella delle pagine e trovo invalid
2. **Trap** ovvero un software interrupt che chiama il **siste operativo** ; il sistema operativo decide se il riferimento è davvero invalid oppure non è ancora stato caricato in RAM
3. **Pagina in backing store** (sul disco e non su RAM)
4. **Caricamento della pagina mancante in RAM**

5. **Reset della tabella delle pagine** ponendo l'indirizzo corretto del frame e il bit su **valid**
6. **Restart dell'istruzione** .

Morale della favola

Questa operazione non è delle più veloci. Infatti, la CPU mentre questo processo avviene (ovvero mentre la pagina viene caricata in RAM), passa ad un altro processo.

Quando la CPU torna a questo processo, ripartirà direttamente dall'istruzione che ha causato il Page Fault.

Rimpiazzo della pagina

Preveniamo la "**over allocation**" (allocazione eccessiva) della memoria andando a modificare la routine del page-fault in modo da includere un rimpiazzo della pagina.

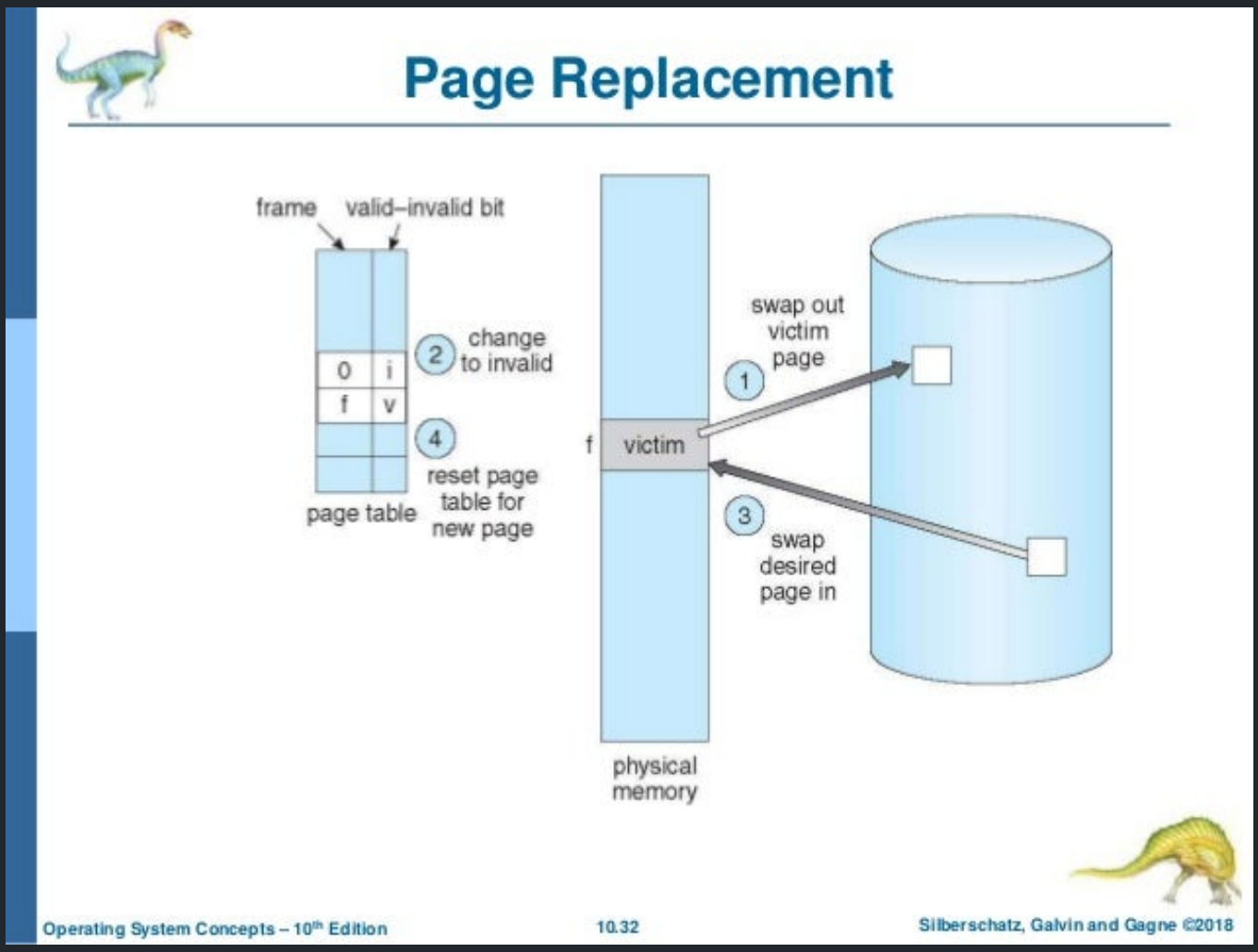
La pagina uscente deve essere riscritta in **backing store** (memoria di massa); si può però utilizzare un **bit di modifica** in modo da ridurre i tempi.

Il tempo per l'operazione di swap si divide in tempo di **swap out** e tempo di **swap in**; se la pagina che esce, è una pagina **non modificata** rispetto a quando è entrata, sul disco è presente la stessa pagina presente in memoria, quindi è inutile salvarla nuovamente.

Quindi, posso tenere traccia delle pagine modificate grazie ad un **dirty modify bit** che mi dice se la pagina è stata modificata, andando a salvare solo le pagine modificate.

Procedura basilare per il rimpiazzo di una Page

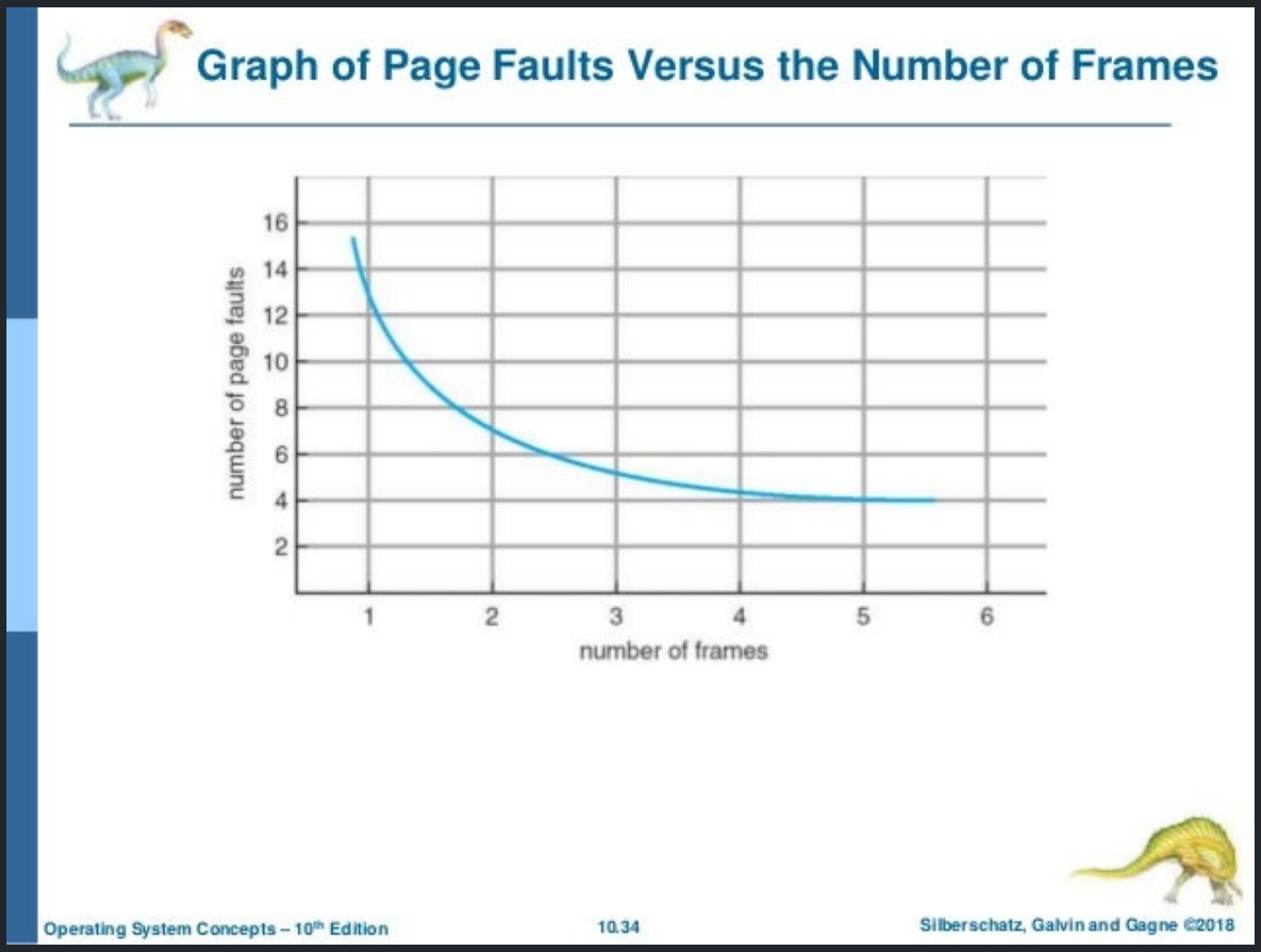
1. Trovare la locazione della pagina sul disco
2. Trovare un frame libero, se presente;
    1. se è presente un frame libero, utilizzarlo
    2. Se non c'è un frame libero, usare un algoritmo di rimpiazzo di pagine per selezionare un **frame vittima**
        1. scrivere il frame vittima sul disco **solo se dirty (modificato)** .
3. Portare la pagina desiderata nel frame appena liberato (o libero già da prima), aggiornare la pagina e la tabella dei frames.
4. Riprendere il processo riavviando l'istruzione che ha causato la trap.



Algoritmi di rimpiazzo frames e pagine

Per ottenere le migliori prestazioni possibili dovrei essere in grado di sapere a priori quali saranno le pagine che mi serviranno, in modo da caricarle nel momento giusto senza rallentamenti.

Se ad esempio so che una pagina mi servirà tra non molto tempo, di certo non la sposto sul disco per caricarne una nuova, ma ne sceglierò un'altra che non mi servirà per un po' di tempo.



Nel momento in cui aumenta il numero di frames a disposizione del processo, il numero di page fault si riduce. Questo è ragionevole, siccome minore è lo spazio a disposizione del processo (dove poter caricare le proprie pagine) più aumenta la probabilità di page fault (ovvero che qualche pagina non sia già presente in memoria).

Se aumento lo spazio a disposizione in RAM fisica il numero di page fault si riduce; il SO tende ad "accontentare" i processi che verificano troppi page fault, in modo da concedergli **più frames**.

Algoritmo FIFO

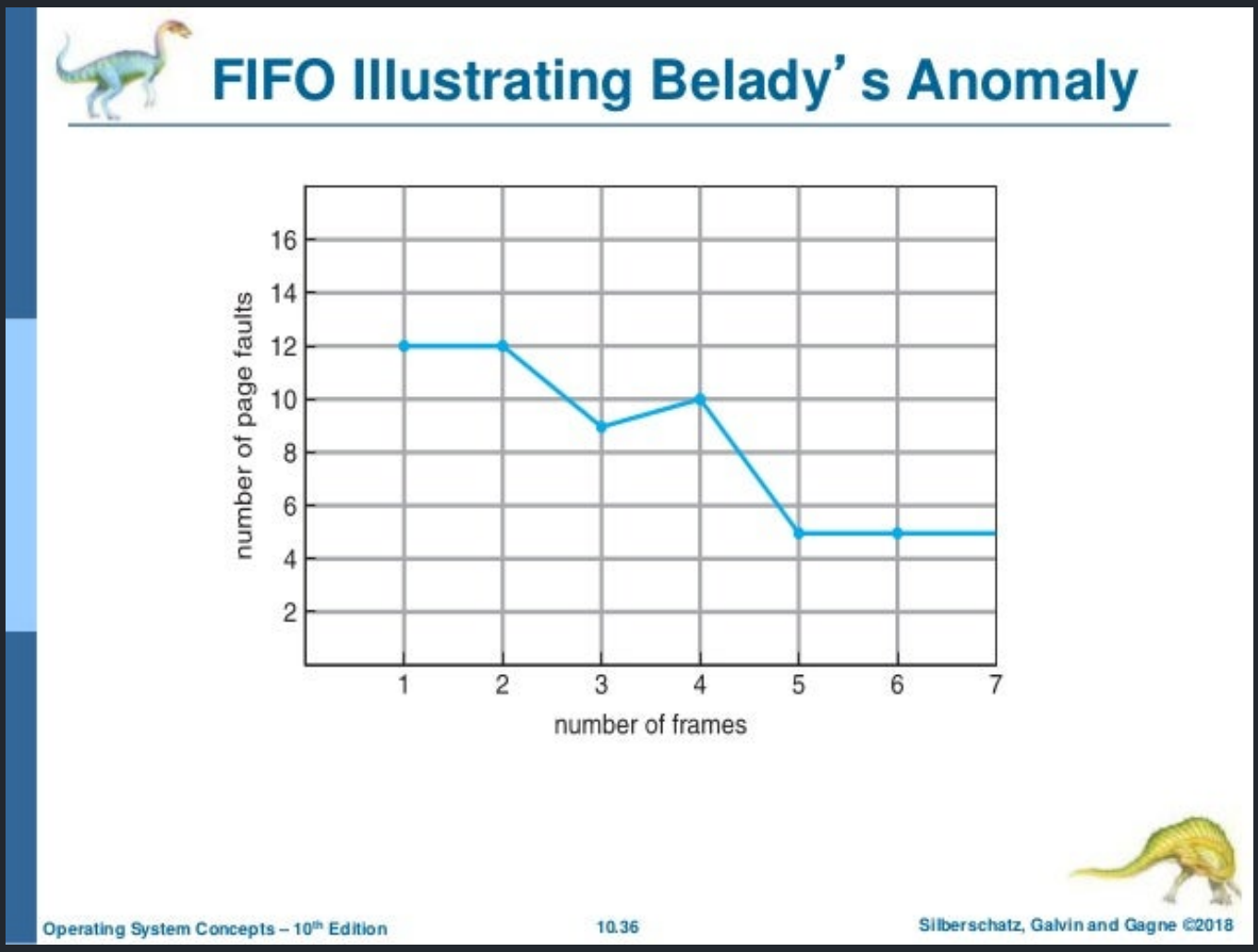
Un algoritmo FIFO prende e sceglie come vittima la **pagina caricata in memoria da più tempo**. Per il semplice motivo che la pagina è in memoria da molto tempo, essa viene scelta per essere spostata sul disco.

Non esiste alcuna garanzia che questa pagina fosse la pagina giusta da togliere, infatti questa pagina potrebbe essere una pagina molto importante del programma, dove potrebbe, ad esempio, risiedere lo **stack**.

Questo sistema non è assolutamente adatto.

L'altro grande problema di questo algoritmo è il seguente:

se fornisco ad un processo, invece di 3, 4 frames, il tasso di **page fault**, invece di diminuire (come dovrebbe), aumenta!
Questa anomalia viene detta **anomalia di Belady**, ed in pratica è dovuta all'utiizzo dell'algoritmo fifo, ed in pratica si verifica un picco di page fault, ovvero con l'aumento di frames, aumentano anche i page fault; inevitabilmente con un aumento ulteriore dei frames questo picco si risolve.



Questa anomalia da particolarmente fastidio, perchè impedisce al SO di reagire ad un tasso di page fault elevato.

Algoritmo ottimale

Per ridurre il più possibile il tasso di **page fault** dovremmo utilizzare un **algoritmo ottimale**, ovvero togliere dalla RAM la pagina che non servirà per più tempo, ma ovviamente non possiamo sapere qual è questa pagina.

Ovviamente questo tipo di algoritmo è solo "immaginario".

Algoritmo Least Recently Used (LRU)

LRU - pagina meno utilizzata recentemente: non sapendo cosa potrà succedere nel futuro, l'algoritmo tenta di utilizzare ciò che è successo nel passato:

l'algoritmo deduce dal fatto che una pagina non viene **utilizzata** da tanto tempo per scegliere la vittima da spostare sul disco.

Questo algoritmo sembra simile a quello FIFO, ma c'è una sostanziale differenza: questo algoritmo sposta la pagina meno utilizzata recentemente (quella che non uso da più tempo), mentre il FIFO spostava la pagina presente in RAM da più tempo.

Il meccanismo di scelta della vittima, è **hardware**, e non software.

Il problema

Il problema è che anche in questo caso, l'algoritmo LRU è troppo complicato da implementare.

Se infatti volessi scegliere la vittima **con precisione** sarei costretto, in hardware, di tenere traccia di tutti gli accessi in modo da avere una **classifica** ordinata dalla pagina utilizzata da più tempo a meno tempo, aggiornando la classifica ogni volta che **effettuo un accesso in memoria**.

Morale della favola

Quello che accade nella realtà non è una scelta completamente casuale (della vittima), ma utilizzare degli algorittmi **LRU approssimati**.

Algoritmi LRU Approssimati

Per realizzare questo tipo di algoritmo, vengono utilizzati i **bit di riferimento**;

Questi sono dei bit che sistematicamente, quando scatta il timer di sistema, vengono azzerati. Ogni volta che i bit di riferimento vengono azzerati, la prossima volta che entrerà in funzione il SO, troverà dei bit di riferimento **alti SOLO per le pagine che sono state utilizzate.**

In questo modo è possibile capire quali pagine, nell'ultimo intervallo di tempo, sono state utilizzate maggiormente.

Su questo meccanismo si basa l'algoritmo che la maggior parte dei SO utilizzano, detto **algoritmo della seconda chance, o algoritmo dell'orologio**:

Algoritmo dell'orologio

In pratica c'è un''indicatore'' che ci dice quale pagina deve essere spostata sul disco; nel momento in cui si verifica un **page fault** avviene il seguente meccanismo:

1. Se il bit di riferimento della pagina è basso (0), la pagina viene scelta per essere spostata sul disco.
2. Se il bit di riferimento della pagina è alto (1):
    1. il bit di riferimento viene posto a zero, lasciando la pagina in RAM
    2. Se al prossimo passaggio la pagina viene trovata con il bit di riferimento basso (0) viene spostata sul disco.

Questo algoritmo è detto ''della seconda possibilità'' perchè nel momento in cui per una pagina è giunto il momento di essere spostata sul disco, ma è stata utilizzata di recente, le si da **una seconda possibilità** azzerando il bit di riferimento.

Se alla seconda passata il bit verrà trovato basso, la pagina verrà spostata definitivamente.

Versione migliorata dell'algoritmo

una versione migliorata di questo algoritmo potrebbe essere quella dove la vittima viene scelta non solo rispetto a quanto è stata usata ultimamente (bit di riferimento == 0), ma anche se essa è una pagina che non è stata modificata, in modo da non dover nemmeno **effettuare lo swap out**, e quindi rimuovendola semplicemente dalla memoria.

🏁 05-26 00:22

Algoritmi di Page Buffering

Potrebbe accadere che la pagina appena spostata sul disco possa riservirmi immediatamente dopo.

Si può pensare ad una posizione "intermedia", che funzioni come un "cestino" dove le pagine "eliminate" vengono poste temporaneamente primad di essere definitivamente spostate su disco.

Questa soluzione richiede una maggiore quantità di memoria, ma offre prestazioni migliori.

Allocare Frames

Qual è il numero di frames che va assegnato ad ogni processo?

Ogni processo ha bisogno di un **numero minimo** di frames. Ad esempio, il vecchio **IBM 370** ha la possibilità , con un'unica istruzione chiamata *SS MOVE*, di generare **6 possibili page fault**!

- Istruzione è di 6 bytes, potrebbe essere a cavallo di 2 pagine
- 2 pagine per il campo di partenza

- 2 pagine per il campo di destinazione

Se, ad un sistema del genere, vengono concessi **meno di 6 frames** c'è il rischio che l'istruzione generi sempre dei page fault, senza mai riuscire ad essere eseguita.

Questo è il motivo per cui ogni processo ha bisogno di un **numero minimo** di frames.

Il **numero massimo** dipende sempre dalla memoria disponibile, ma ci sono delle guidelines:

- Allocazione fissa: ovvero sempre lo stesso numero di frames per ogni processo
- Allocazione a priorità

Allocazione fissa

Divido il numero di frames a disposizione per il numero di processi, dando un numero uguale di frames per ogni processo.

L'evidente problema di questo metodo è che alcuni processi avranno uno spazio indirizzi molto piccolo, ed altri con uno spazio indirizzi molto grande.

Allocazione proporzionale

Simile all'allocazione fissa, dove divido in frames la memoria disponibile; in questo caso, però, i vari frames vengono **mappati** sui processi a seconda della dimensione del processo: i processi **più grandi** riceveranno un maggior numero di frames.

Allocazione globale vs locale

- **Rimpiazzo globale**: il processo seleziona un frame di rimpiazzo dall'insieme di **tutti i frames**; un processo può ricevere un frame da un altro.
  - Il tempo di esecuzione del processo può essere molto vario
  - Ha un **throughput** maggiore, quindi più comune
- **Rimpiazzo locale**: ogni processo seleziona solo dal proprio insieme di frames allocati.
  - La performance per-processo è più affidabile
  - Possibilmente la memoria viene "poco utilizzata".

Trashing

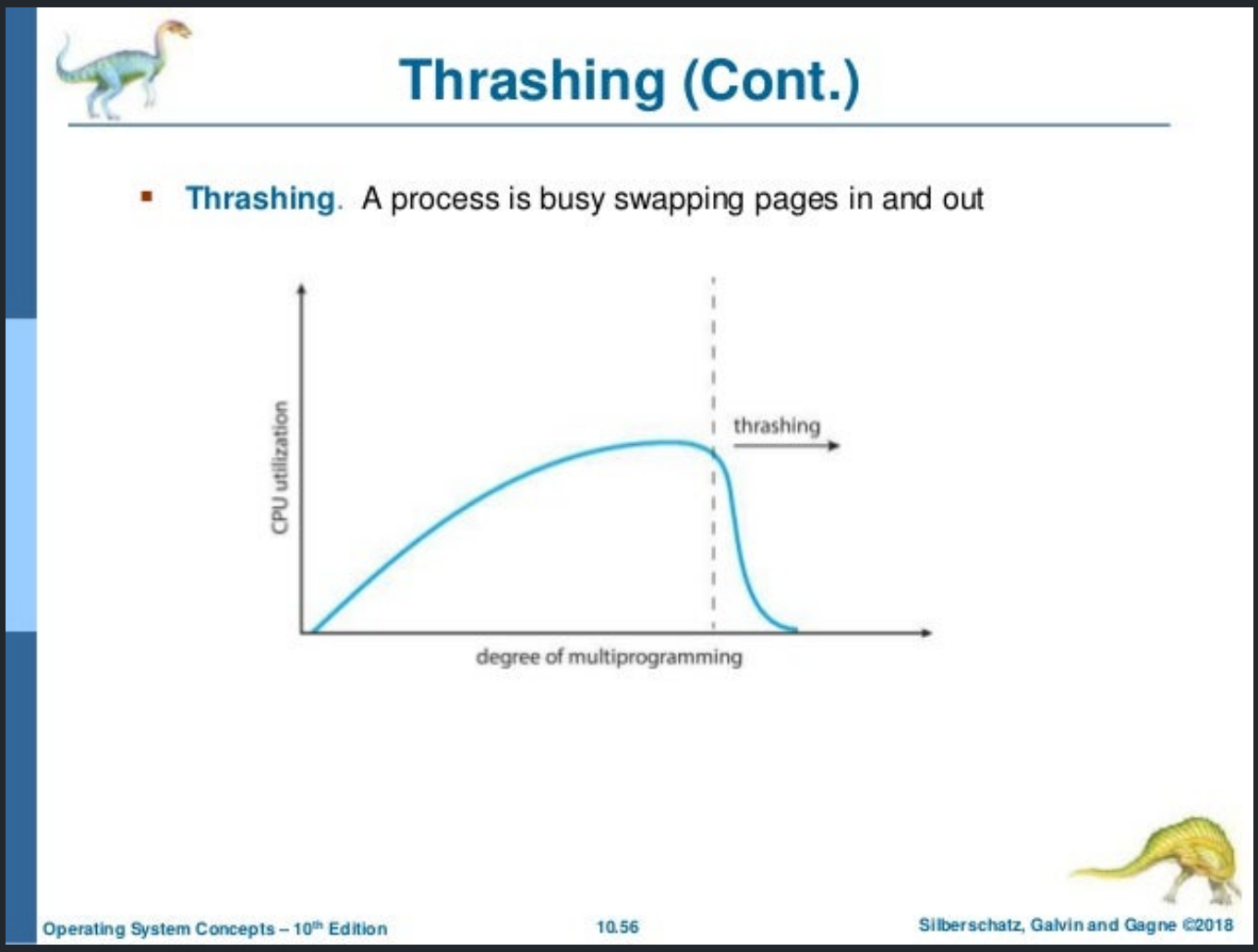
Se c'è un processo avente troppo pochi frames a disposizione, la probabilità di page fault che si verificheranno per quel processo sarà molto alta.

Di conseguenza possiamo definire il **trashing** come un processo impegnato a scambiare pagine tra RAM e disco.

Questo fenomeno va ad intaccare anche gli altri processi, proprio perchè effettuando continuamente degli scambi di pagine da RAM a disco, terà sempre occupato il disco.

Questa situazione porta ad una bassa percentuale di CPU utilizzata, che il sistem **potrebbe vedere come fattore positivo**, e lanciare un altro processo. Questo è disastroso, proprio perchè ci sarà ancora più scarsità di frames per accontentare il processo che sta andando in trashing.

Questo fenomeno è particolarmente "pericoloso" nei **sistemi batch**, ovvero quei sistemi dove l'interazione unamana è quasi inesistente, ed il sistema si occupa di tutto.



I sistemi batch, decidono di lanciare processi nel momento in cui la CPU è sottoutilizzata, e per questo motivo questi sono i sistemi più soggetti a questo fenomeno.

Sistemi come Mac OS o Windows non soffrono di questo problema, per via del fatto che è l'utente a scegliere il programma (e quindi processi) da lanciare.

Perchè si verifica il trashing

Solitamente si verifica perchè un processo utilizza troppe pagine.

Il trucco risiede nel riuscire ad assegnare al processo un numero di frame che lo "accontenti" durante la durata di tutta la sua vita. Dobbiamo quindi cercare di capire quale sarà il numero **minimo** di frames che gli serve affinchè non inizi a generare page faults.

Considerazioni finali sulla Virtual Memory

Prepaginazione

Il concetto è iniziare a dare ad un processo un certo numero di pagine, nel momento in cui esso viene avviato, in modo ch non abbia dei page fault immediatamente.

Dimensione della pagina

Spesso la pagina è a dimensione fissa per un certo processore; solitamente **le pagine sono di grandezza pari alle potenze di 2**.

Nel tempo (storicamente) con l'aumento della dimensione della memoria fisica (RAM) è andata aumentando anche la dimensione delle pagine.

TLB Reach

Il TLB contiene pochi entry, più la pagina è grande, più il TLB (giustamente) riuscirà a coprire una porzione maggiore della memoria.

Struttura del programma

Se un programma ha una struttura che gli permette di occupare spazio in memoria in locazioni più o meno consegutive, il numero di page fault diminuisce; questo succede perchè abbiamo una maggiore probabilità di trovare dati che ci interessano in una singola pagina.

Un programma scritto male inevitabilmente sarà costretto ad effettuare un numero di page fault elevatissimo; Bisogna quindi accedere alla memoria ad **indirizzi vicini tra loro**.

I/O Interlock

E' necessario, a volte, bloccare delle pagine in memoria.

Questo perchè potrebbero contenere un buffer dal quale sta arrivando della data da unità I/O, quindi quelle pagine devono essere **bloccate in RAM**.

Per quanto riguarda la sicurezza, supponiamo di avere un programma che gestisce informazioni mantenute criptate; se il frame dove vengono mantenute in chiaro queste informazioni viene assegnato ad un altro processo senza che venga azzerato (come normalmente dovrebbe accadere), quel processo può leggere i dati in chiaro.
I programmi che gestiscono informazioni assolutamente private, sono costretti a bloccare le pagine segnalando al SO di non spostare le pagine.
