Capitolo 9 : Memoria Principale

Table of contents

Background

Un programma per essere eseguito, non può restare nel disco, ma deve essere caricato in **memoria centrale** del sistema. Un programma non viene quindi eseguito dal **disco**, ma viene portato interamente, o in parte, nella **RAM**;

Questo anche perchè la **CPU** può accedere solo a **cache e memoria centrale.** L'unico modo per la CPU di accedere al disco è tramite la RAM: il controllo deve portare ciò che serve dal disco alla memoria, e solo a questo punto la CPU può lavorarci.

Cosa vede la memoria?

La memoria vede uno **stram di richieste** che arrivano dalla CPU, che possono essere richieste di **lettura** o richieste di **lettura e scrittura**.

La CPU fornisce un **indirizzo** che servirà a prelevare ciò che è presente in quella cella di memoria.

Velocità

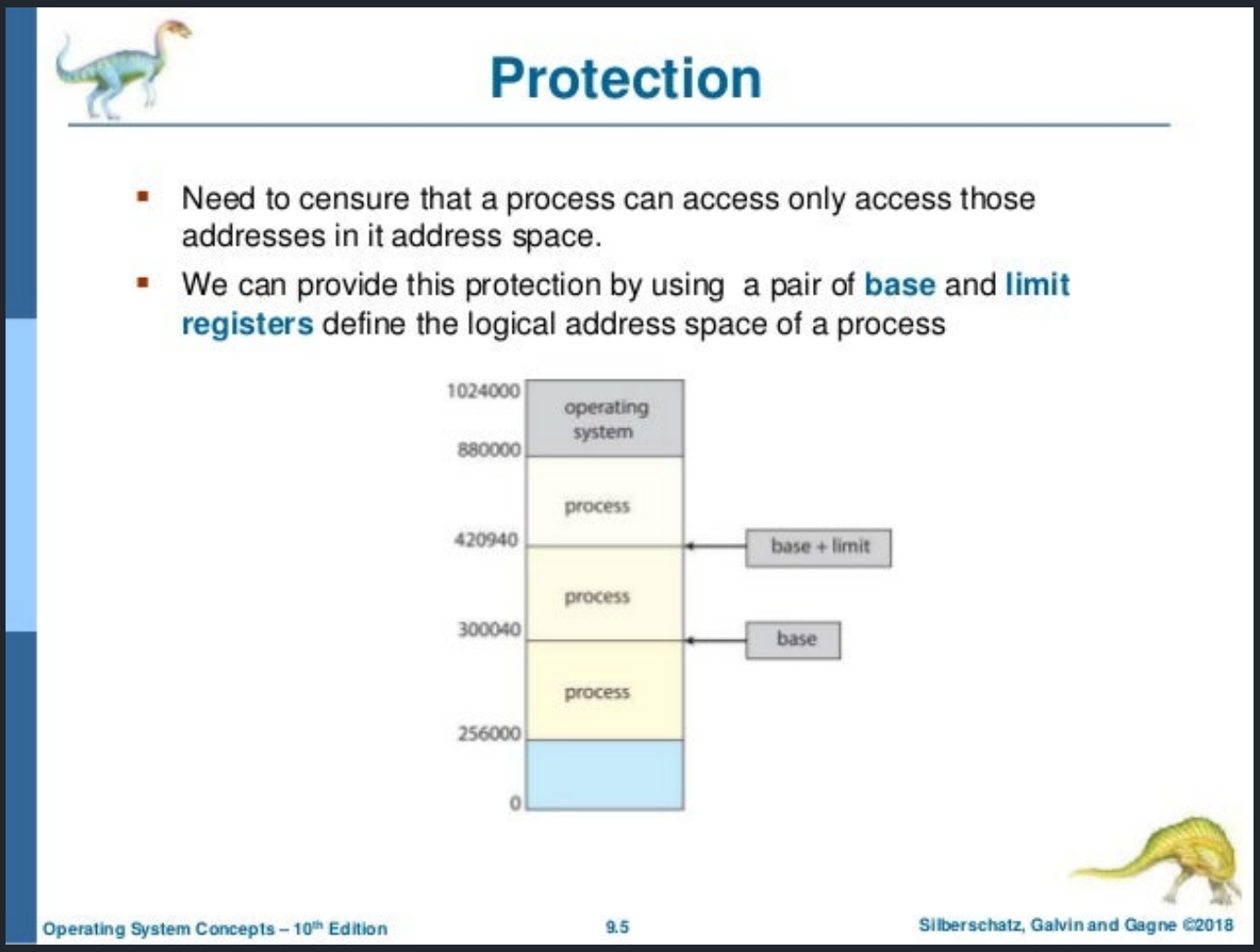
Mentre l'accesso, da parte della CPU, ai registri è tipicamente veloce (dell'ordine di **un solo ciclo di clock**), l'accesso alla RAM è molto lento, dell'ordine di diversi cicli di clock.

Per questo motivo tra i registri e la memoria sono presenti diverse celle di **cache** che attenuano il problema.

Sistema di protezione

La gestione della memoria deve prevedere un sistema di protezione:

I processi non devono essere lasciati liberi di poter accedere al di fuori del proprio spazio indirizzi; ogni processo, infatti, riceve un proprio spazio indirizzi, ovvero un "pezzo" della ram del sistema, e all'interno di quello spazio può fare ciò che vuole, ma non può accedere a celle di memoria di altri processi, o addirittura, del sistema operativo.



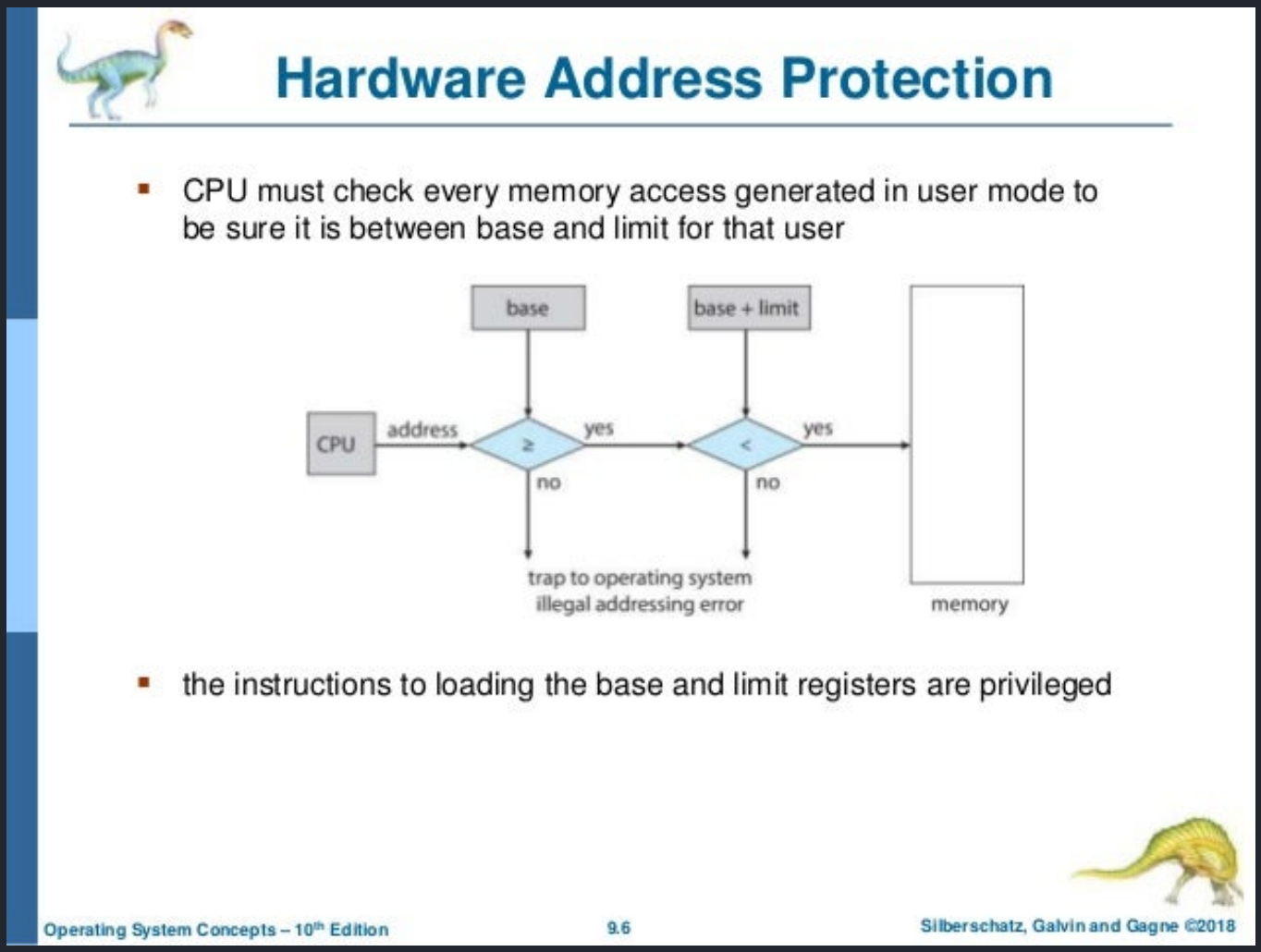
Se la memoria del sistema è composta dal rettangolo intero, ed una sua sezione è composta dal sistema operativo, e tanti rettangoli compongono i diversi processi, come garantisco che un dato processo non vada ad accedere allo spazio indirizzi di un altro processo, o addirittura del SO?

Essenzialmente serve un meccanismo **hardware**; deve essere hardware perchè via software non posso verificare che ogni indirizzo che la CPU fornisce sia un indirizzo **lecito**, serve quindi qualcosa a basso livello.

Basterebbe l'utilizzo di un hardware che fa uso di due registri:

- **base** : dice dove inizia la fetta di memoria del processo
- **base + limit** : dice il massimo indirizzo a cui il processo può accedere.

Protezione **Hardware**



- La CPU fornisce un indirizzo
- In hardware ogni indirizzo proveniente da CPU deve essere verificato per vedere se è compreso nella fetta oppure no.
- Se è compreso, l'indirizzo passa e viene utilizzato; se non è compreso avviene una **trap** e si verifica quindi un'eccezione.
  Questo è proprio quello che accade quando, programmando, aggiungiamo ad un array un elemento in più di quello che può contenere, e ci viene quindi restituito un errore *segmentationFault*

Address Binding

Quando scriviamo un programma utilizziamo dei nomi simbolici (variabile x, i, ecc.). Nel momento in cui il programma va in esecuzione, questi nomi simbolici non esistono più, ma vengon trasformati in un certo indirizzo di memoria che gli è stato associato.

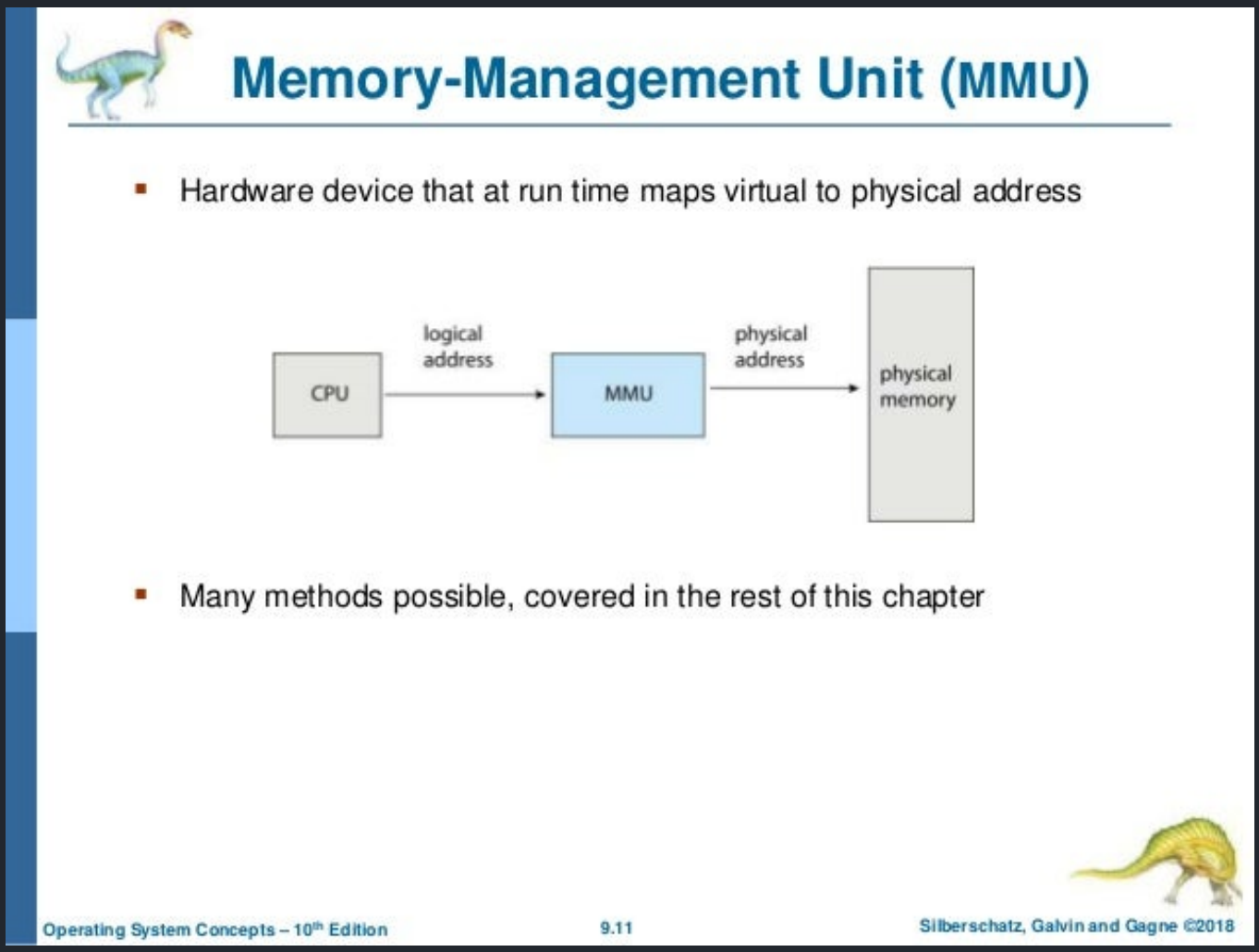
Viene quindi effettuata **un'associazione** tra i nomi simbolici presenti nel codice, e gli indirizzi effettivi di memoria.

Quando viene fatta l'associazione?

1. **Al momento della compilazione** : scrivo x nel mio programma, ed il **compilatore** decide che x corrisponde ad un certo indirizzo fisico in memoria.
   In un sistema moderno questo non accade; infatti un eseguibile non conosce ancora la zona di memoria in cui verrà posto, quindi il compilatore ragiona su "indirizzi di fantasia" che verranno poi convertiti in indirizzi fisici.
2. **Al momento di caricamento in memoria** (caso antico): Nel momento in cui carico il programma in una zona di memoria significa che devo modificare tutti gli indirizzi che ha fornito il compilatore, in modo che il programma giri in quella specifica area di memoria.
   Il binding viene effettuato, quindi, al momento del **caricamento** ; si dice che il **codice sia rilocabile** .
   Anche in questo caso questa non è una soluzione utilizzata nei sistemi moderni.
3. **Al momento di esecuzione** (Soluzione attuale): il binding tra i nomi simbolici delle variabili, e gli indirizzi effettivi della memoria, **viene ritardato fino al momento dell'esecuzione** .
   Questo significa che solo nel momento in cui il codice viene eseguito, l'indirizzo viene modificato per far riferimento ad una certa area di memoria.

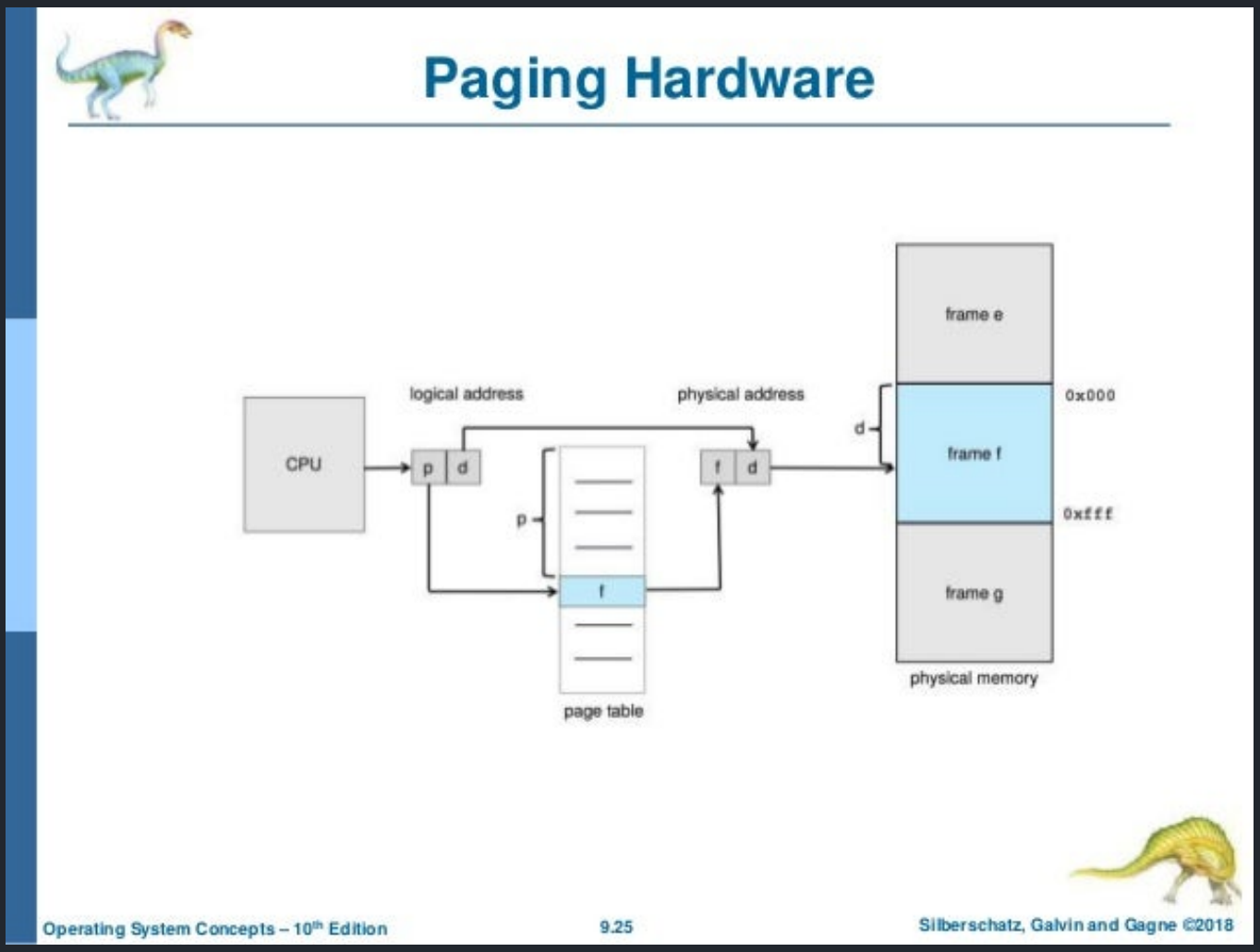
Memory Management Unit (MMU)

L'unità che gestisce la modifica degli indirizzi si chiama MMU. E' un componente **hardware** che, inizialmente (verso gli anni 60), era un componente hardware **esterno**, letteralmente posto **tra CPU e memoria fisica**; nei Chip moderni, L'MMU è posto all'interno del processore, e svolge anche compiti più complicati.



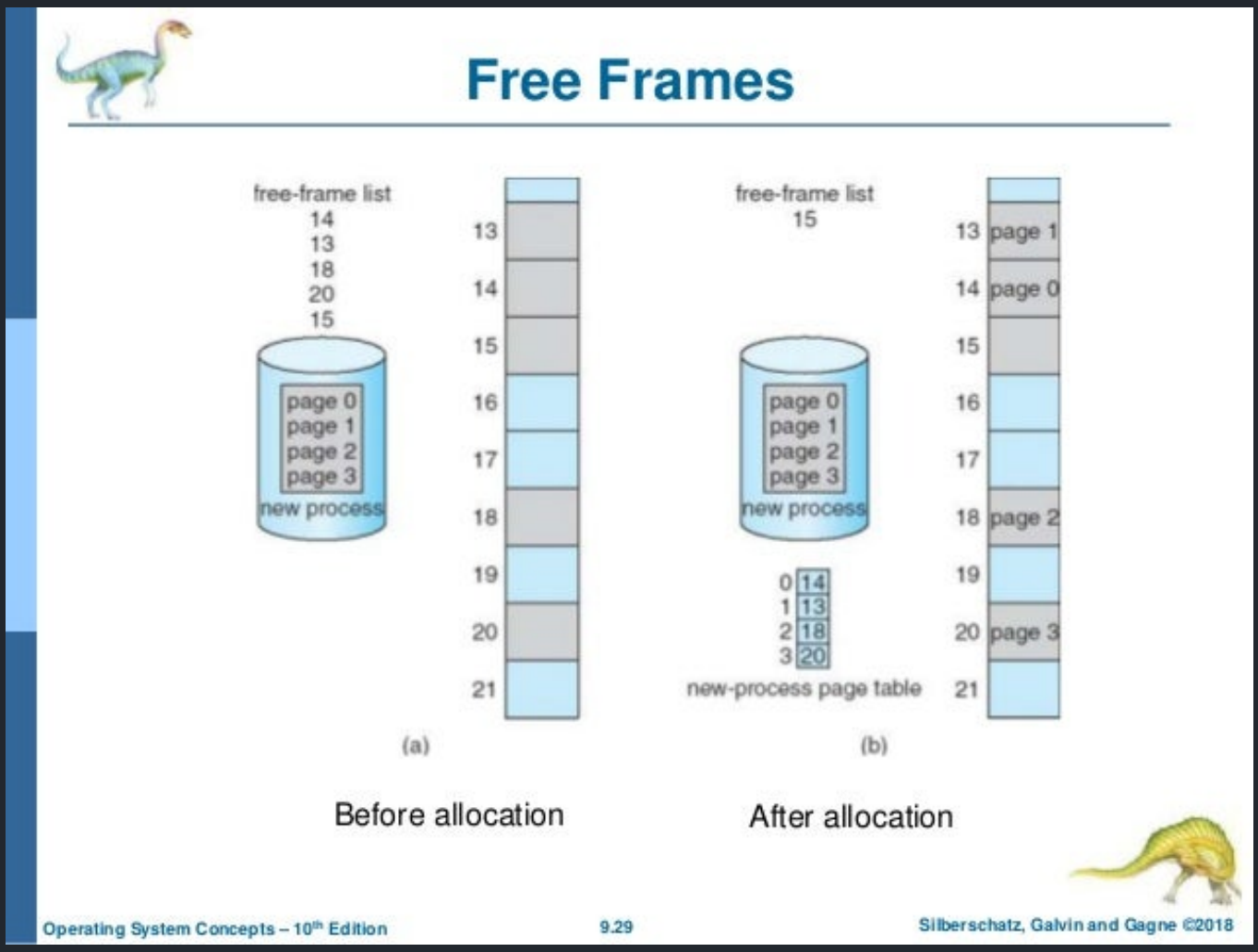
FINE LEZIONE 10

##



Per capire meglio...

Entro nell'hardware paging:



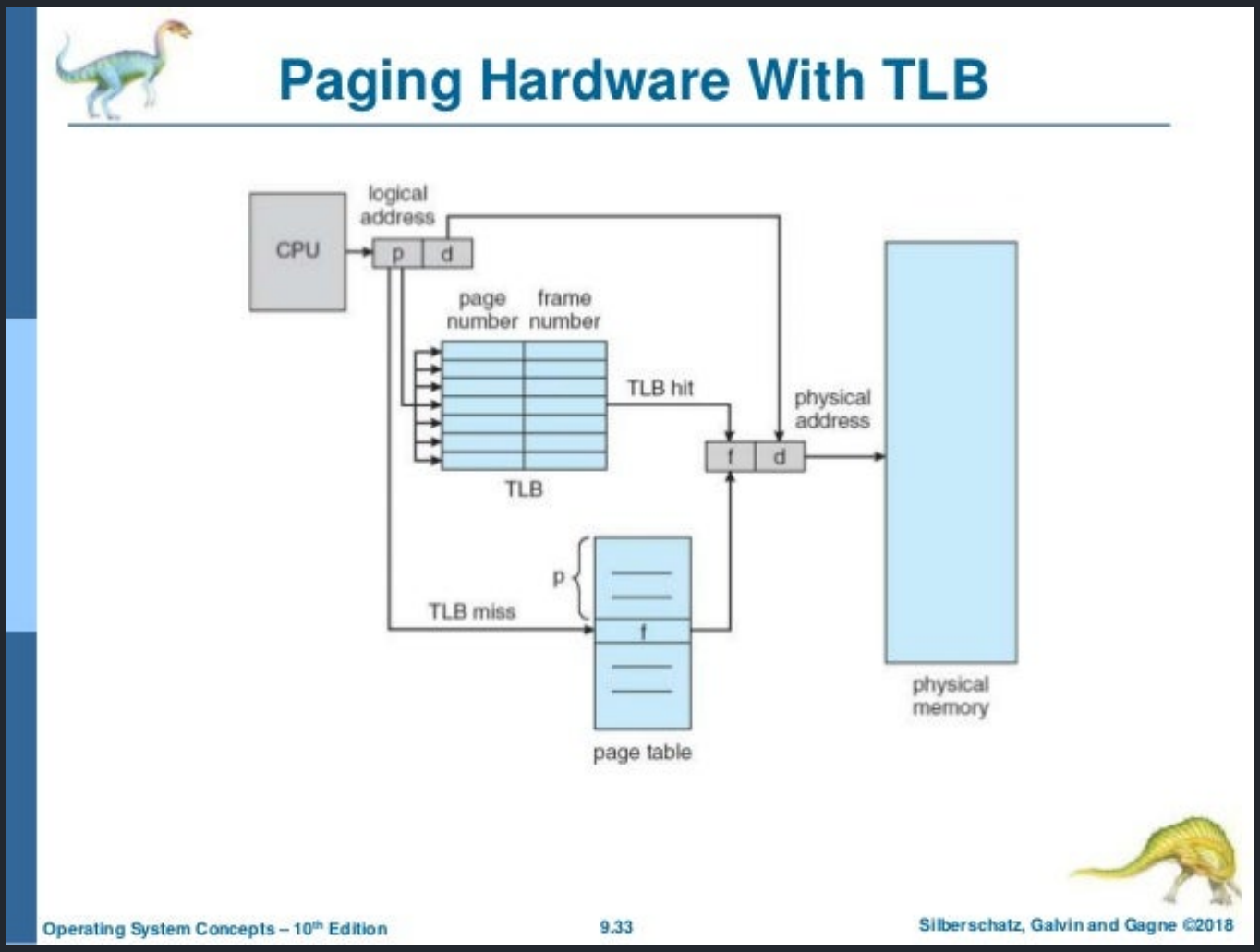
Se nel TLB (cache) pongo una coppia **pagina-frame** utilizzo la pagina come **chiave** di accesso, ed ottengo quindi il numero del frame.

Siccome il TLB non è ad accesso random, ma di tipo cache (quindi più veloce) non aumento di troppo di accesso in memoria.

Se non trovo l'indirizzo nel TLB?

Se non riesco a trovare l'indirizzo di memoria nella cache, non c'è molto altro da fare, e devo accedere alla memoria *the old fashioned way* e quindi sprecare tempo.

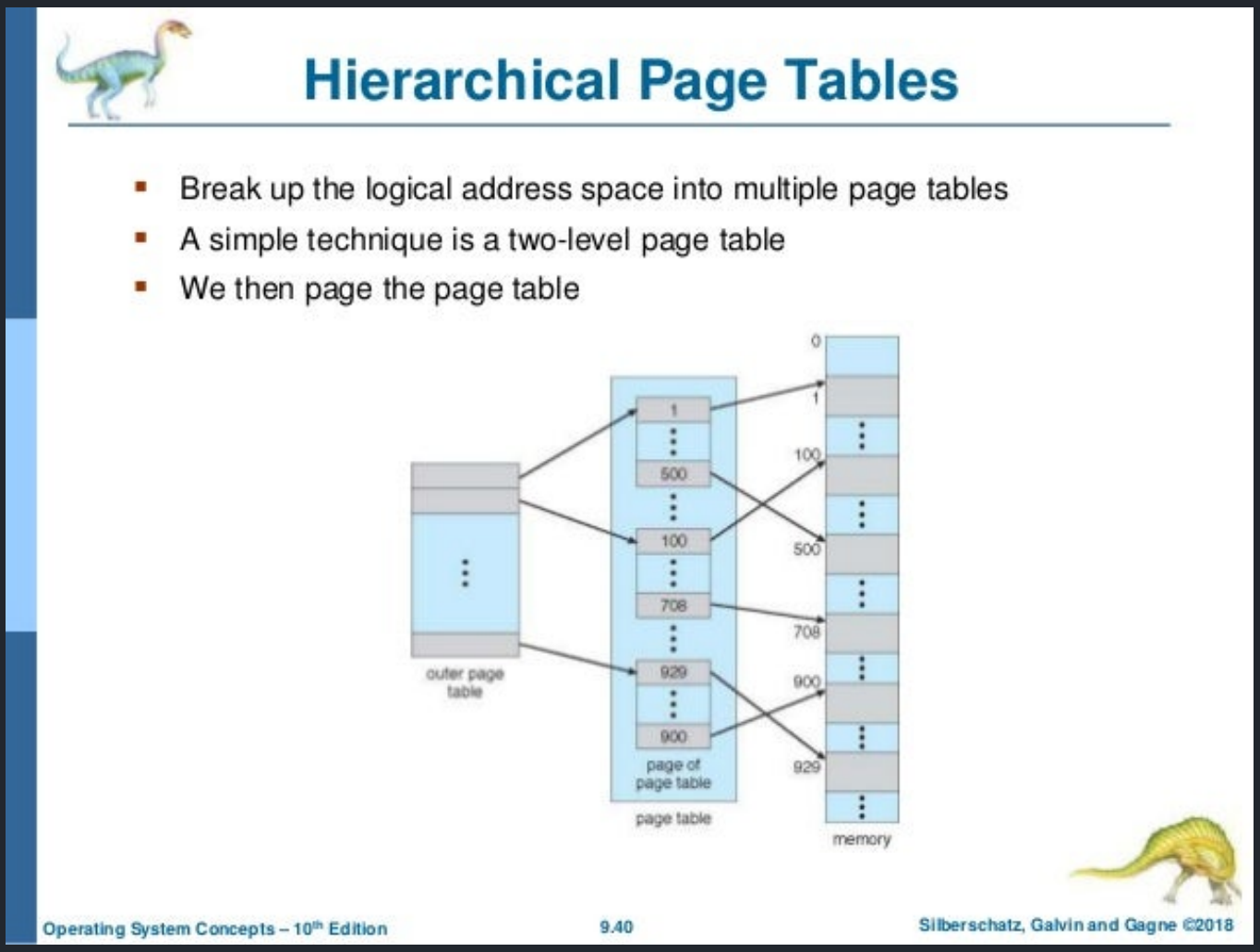
Tuttavia, se quella è la prima volta che effettuo l'accesso a quella specifica cella di memoria, probabilmente in un prossimo futuro mi servirà nuovamente, quindi viene salvata nel TLB per un accesso più veloce.



####

Tabelle delle pagine gerarchich

All'atto pratico possiamo avere "diverse" tabelle delle pagine, organizzate in modo gerarchico:



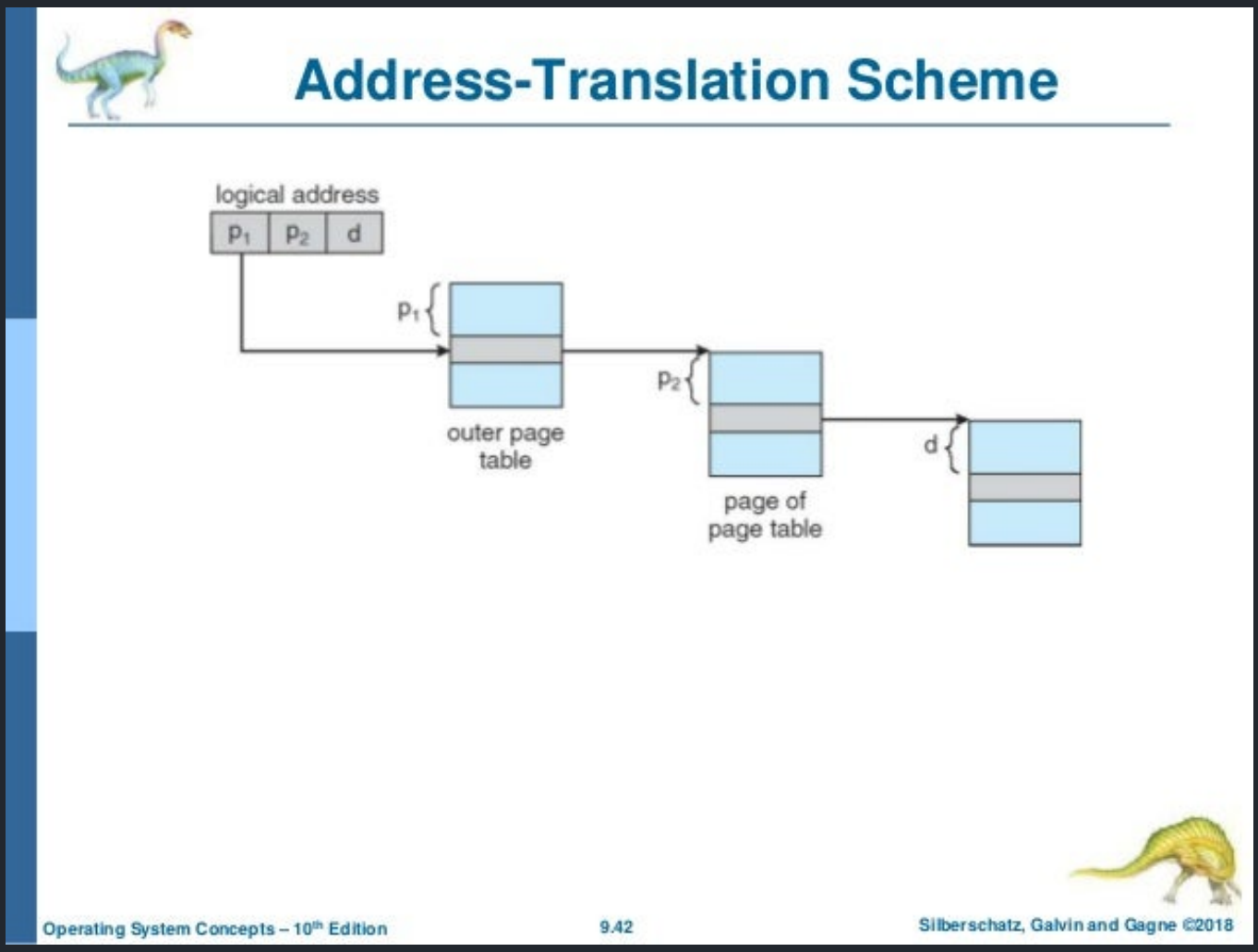
In questo caso abbiamo una tabella di primo livello che punta ad una di secondo livello, la quale punta alla tabella effettiva.

La prima tabella è chiamata **tabella delle pagine esterne**, che è quella esposta; la seconda è chiamata **tabella delle pagine** (l'effettiva tabella delle pagine), la quale punta direttamente alla memoria fisica.

I vantaggi

Questo tipo di tabelle vanno a risolvere il problema dell'unica tabella enorme; infatti queste essendo suddivise in ordine gerarchico, sono nettamente più piccole.

Traduzione degli indirizzi nelle tabelle gerarchiche



Prima l'indirizzo a 32 bit era diviso in due parti, in questo caso, che abbiamo diverse tabelle, l'indirizzo deve essere diviso ulteriormente.

Avremo quindi che i **bit più significativi** servono a posizionarmi all'interno della **outer page table**; i **bit intermedi** servono per la **tabella delle pagine** ed i bit meno significativi servono per la memoria effettiva.

🏁 05-21 1:06

Paging - continuo

Caricamento dinamico

Il loading dinamico è una tecnica non utilizzata più tanto spesso ai giorni d'oggi; Il programma veniva "spezzettato" in tanti moduli che venivano caricati in memoria uno alla volta, in modo da risparmiare memoria ed utilizzare programmi relativamente grandi che non entravano interamente nella ram.

Contiguous Allocation

Come fa il sistema dove allocare il programma e come gestire la memoria in modo da far convivere i vari programmi?

Una soluzione è quella di scegliere uno spazio di allocazione che sia **contiguo**, ovvero che ogni processo ha il suo spazio indirizzi, e si trovano tutti allo stesso posto.

Un'operazione del genere è complicata, perchè è esattamente come

L'allocazione dinamica della memoria:
C'è uno spazio di memoria, ed ogni processo ha bisogno di un "pezzo" e lo rilascia a suo piacimento. Per questo motivo è chiamato **heap, ovvero "mucchio"**.
Funziona quindi in modo diverso dallo stack, dove la memoria deve essere rilasciata in un ordine ben preciso.

Il problema della memoria dinamica, è che i processi rilasciando la memoria a loro piacimento, ci ritroveremo con dei "buchi" di memoria libera, che non solo sono troppo piccoli, ma anche "sparpagliati".

Per intenderci, questo è quello che succede quando utilizziamo la **free()** della **malloc** in C.

Soluzione - Garbage Collection

Quando abbiamo lo spazio in memoria suddiviso in tanti piccoli pezzi, che sono troppo piccoli per essere utilizzati, entra in gioco il **garbage collection.**

Quello che il garbage collector effettua, in parole povere, è spostare tutta la memoria (o meglio ciò che la occupa) da una parte, in modo da avere lo spazio libero tutto concentrato in un punto.

Questo non è un problema semplice da risolvere, anzi, è uno dei più difficili dell'informatica.

Frammentazione

- **Frammentazione Esterna** ci sono dei frammenti (buchi) di memoria, ma sono troppo piccoli per essere utilizzati (perchè non continui).
  Addirittura anche 1/3 della memoria diventa inutilizzabile. Si parla di frammentazione **esterna** quando ho della memoria non utilizzata, ma che non può essere usata. Questo è il problema più diffuso.
- **Frammentazione interna** Si parla di frammentazione **interna** quando ho della memoria che **non è utilizzata, ma marcata come occupata.**

Come ridurre la frammentazione?

Come già detto precedentemente, possiamo ridurre la frammentazione tramite la **compattazione**, ovvero porre tutta la memoria libera in un **unico grande blocco**.

La compattazione è possibile solo quando la rilocazione è dinamica, e viene effettuata a tempo di esecuzione.

Inoltre abbiamo il **problema dell'I/O**, ovvero se avessimo delle locazioni di memoria che ospitano **buffer per I/O**, questi non possono essere spostati, perchè altrimenti i dispositivi I/O non li troverebbero più.

Soluzione migliore per risolvere la frammentazione

Sostanzialmente, la frammentazione si verifica perchè alloco **spazi di dimensioni variabili** i quali vengono allocati in tempi diversi, per cui diventa difficile trovare dello spazio libero, che viene sprecato.

La **paginazione** è ciò che risolve il problema:

Paging

L'idea è di allocare spazi che siano di dimensione **fissa**; divido la memoria fisica in spazi di una certa dimensione detti **frames**. Lo spazio indirizzi del processo che devo allocare in memoria, viene così diviso in tante fette, che sono esattamente della dimensione del frame; queste fette sono dette **pagine**.

L'idea è quindi quella di porre le pagine all'interno dei frame, in modo da non **sprecare spazio**. Ovviamente anche in questo caso un minimo di spazio resta inutilizzato, infatti si presume che il frame che occupa **l'ultima pagina** della memoria del processo, non sia riempito completamente

La grandezza di un **frame, è una potenza di 2**, compresa tra **512 bytes** e **16 Mb**; la grandezza dei frame (e quindi delle pagine) dipende dal processore.

Il trucco

Il vero punto di forza di questa tecnica, è che tutti questi "moduli", **non devono essere continui**!

Grazie ad un sistema di **risoluzione degli indirizzi**, viene effettuata, al momento dell'accesso in memoria, una traduzione da **indirizzo logico a fisico**, che permette di avere, per un processo, delle pagine poste a distanza tra loro.

Quindi: i vantaggi

Come detto, il primo grande vantaggio è quello di non dover necessariamente sprecare la memoria, proprio perchè non è necessaria **l'allocazione dinamica di memoria**, e quindi il massimo di memoria che non viene utilizzata, è la restante parte del frame che non viene occupata dall'ultima pagina di un processo. Questa parte di memoria non utilizzata, è un esempio di **frammentazione interna**, proprio perchè la memoria è ritenuta occupata, ma non lo è.

Il secondo grande vantaggio è il fatto di poter posizionare le varie pagine di un processo ovunque io voglia (o più probabilmente dove vuole il sistema), senza dover avere l'allocazione della memoria di un processo **continua**.
Questo è possibile grazie al fatto che avviene una "traduzione" da indirizzo logico ad indirizzo fisico; infatti la **CPU continua a vedere uno spazio degli indirizzi continuo**.

Schema della traduzione degli indirizzi

Ho degli indirizzi di memoria; l'indirizzo in memoria è una **stringa di bits**.

Supponiamo di avere delle pagine fatte "a fette" nello spazio indirizzi che parte da 0, e supponiamo di avere delle pagine di grandezza **1k** (1024b).

Se prendo un indirizzo **logico** a 32 bit, come posso **logicamente dividerlo?**

`'0' in binario a 32 bit:` 0000 0000 0000 0000 | 0000 0000 0000 0000

Supponiamo di dividere il numero a 32 bit in modo che i 10 bit meno significativi rappresentino la **posizione all'interno della pagina in cui mi trovo**.

Mentre invece i 22 bit più significativi rappresentino **il numero della pagina in cui mi trovo**

`'1023' in binario a 32 bit:`
0000 0000 0000 0000 0000 00 | 11 1111 1111

In questo modo, tutti e 22 i bit più alti rappresentano 0, ovvero la **prima pagina**, mentre i 10 bit più bassi rappresentano gli indirizzi, che vanno da 0 a 1023.

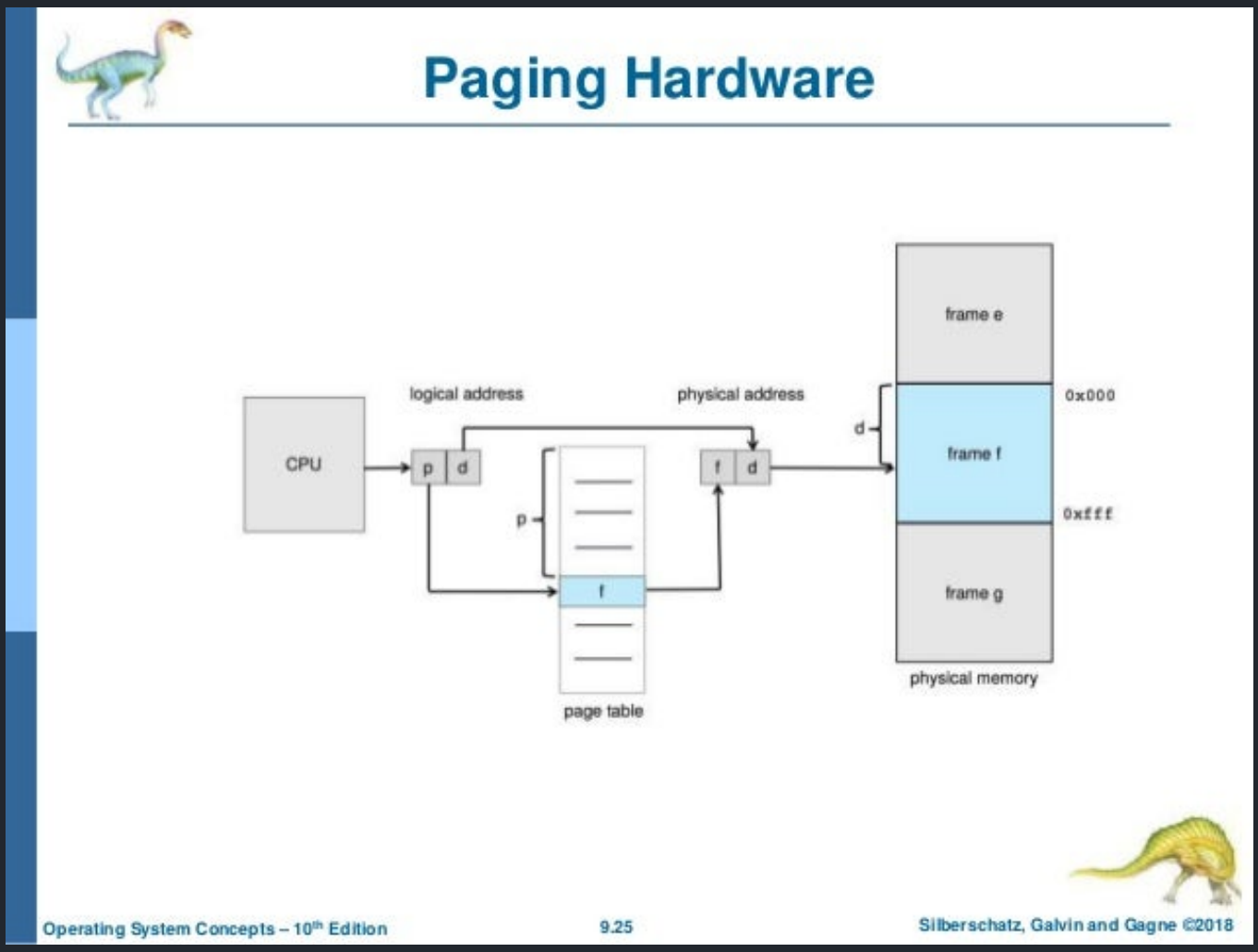
Se l'indirizzo cresce di 1, ci portiamo al valore 1024, rappresentato in questo modo:

`'1024' in binario a 32 bit:`
0000 0000 0000 0000 0000 01 | 00 0000 0000

In questo caso la parte più significativa diventa ...0000 01, mentre la parte meno significativa torna a 0; in altre parole sono passato alla pagina successiva (1) dove ancora una volta gli indirizzi andranno da 0 a 1023.

Ovviamente la posizione della pagina deve essere a potenza di 2.

Paging hardware



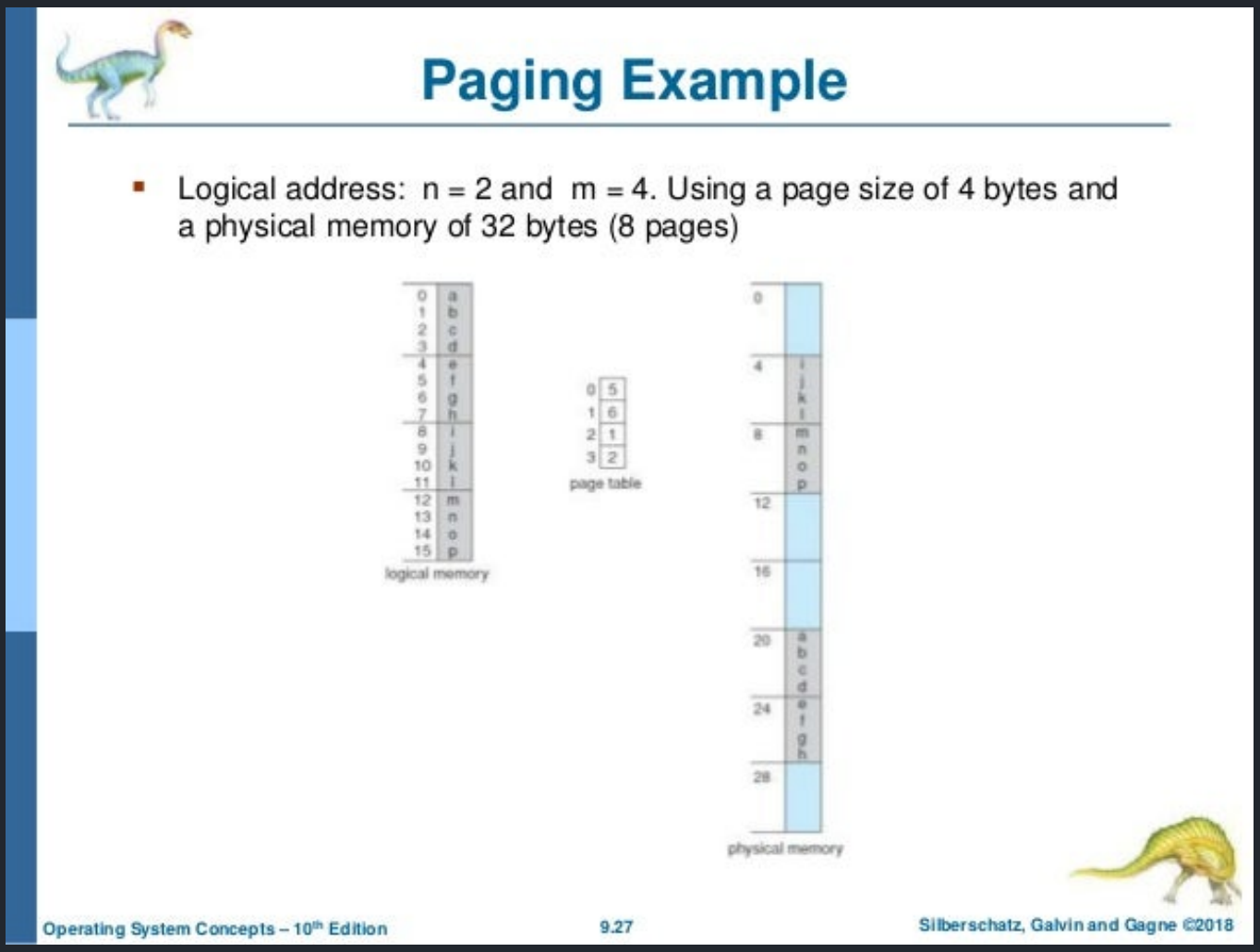
Nel momento in cui carico il processo in memoria, devo mettere nella tabella delle pagine, le posizioni in cui sono stati posti i numeri di frame in cui le pagine sono state poste.

La CPU mi da un indirizzo, utilizzo la parte più significativa come **numero di pagina**, per spostate di *i* posizioni nella tabella.

Arrivato nella *i-esima* posizione della tabella, trovo il numero del frame.

Quindi...

Ogni processo ha la **propria** tabella delle pagine, ed in ogni posizione è scritto il numero del frame in cui la pagina è stata caricata.



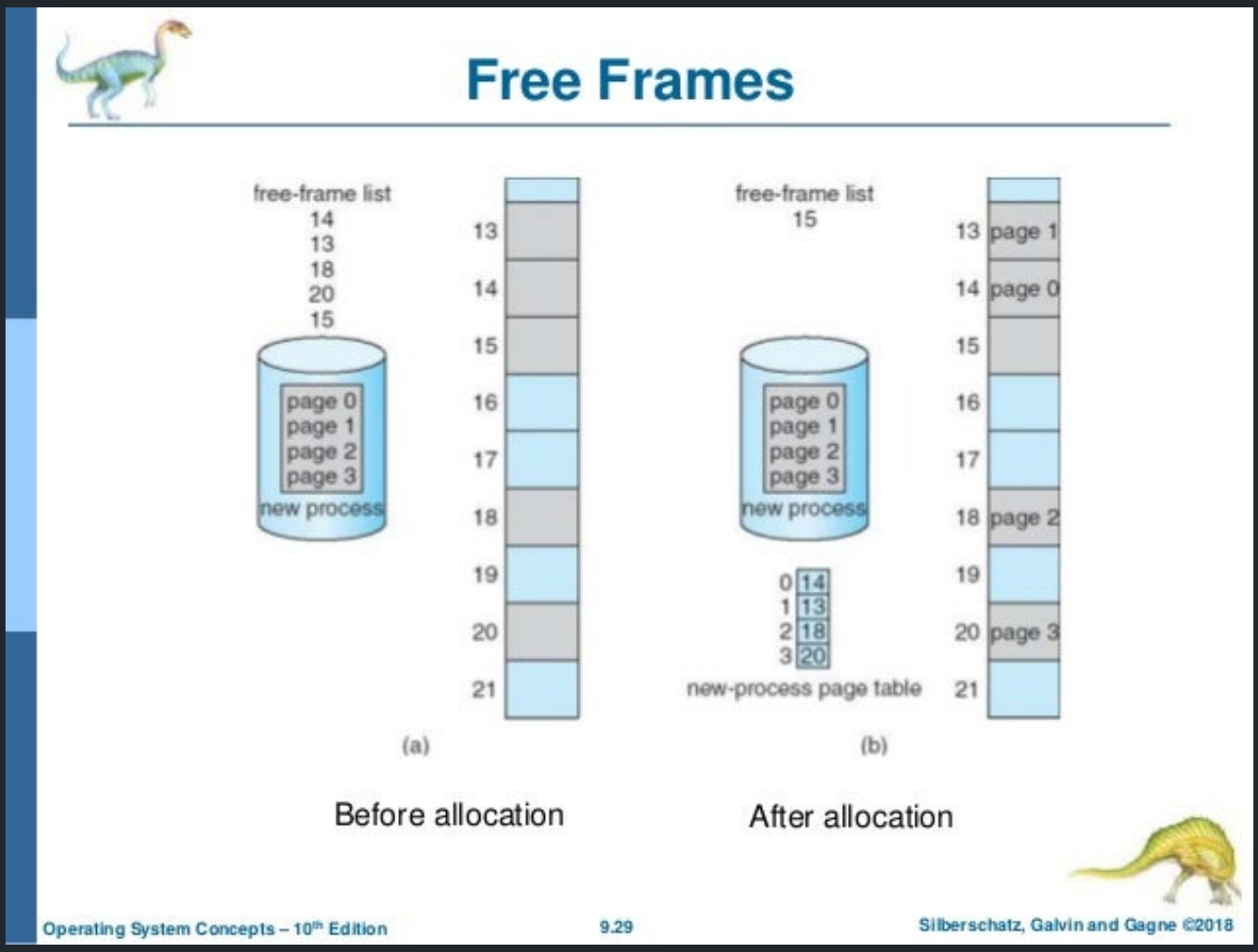
Paging - Calcoliamo la frammentazione

Con delle pagine di 2048B (2k), un processo di 72,766B, richiede 35 pagine, e restano quindi 1,086b persi.
Mediamente la dimensione persa è di circa 1/2 frame;

Con il tempo la dimensione delle pagine tende a crescere, infatti **solaris** supporta due grandezze di pagina: 8KB e 4MB.

Frames Disponibili

Siccome non c'è interesse nell'usare dei frames consegutivi, basta una semplice lista di frames liberi per consentire al sistema di trovare immediatamente uno spazio in memoria dove allocare lo spazio richiesto da un processo.



Come si implementa una tabella delle pagine - TLB

Ogni volta che devo effettuare un accesso in memoria, in realtà deve essere eseguito un accesso supplementare che legga sulla tabella delle pagine.

Quindi essenzialmente, anche in questo caso la CPU "perde tempo".

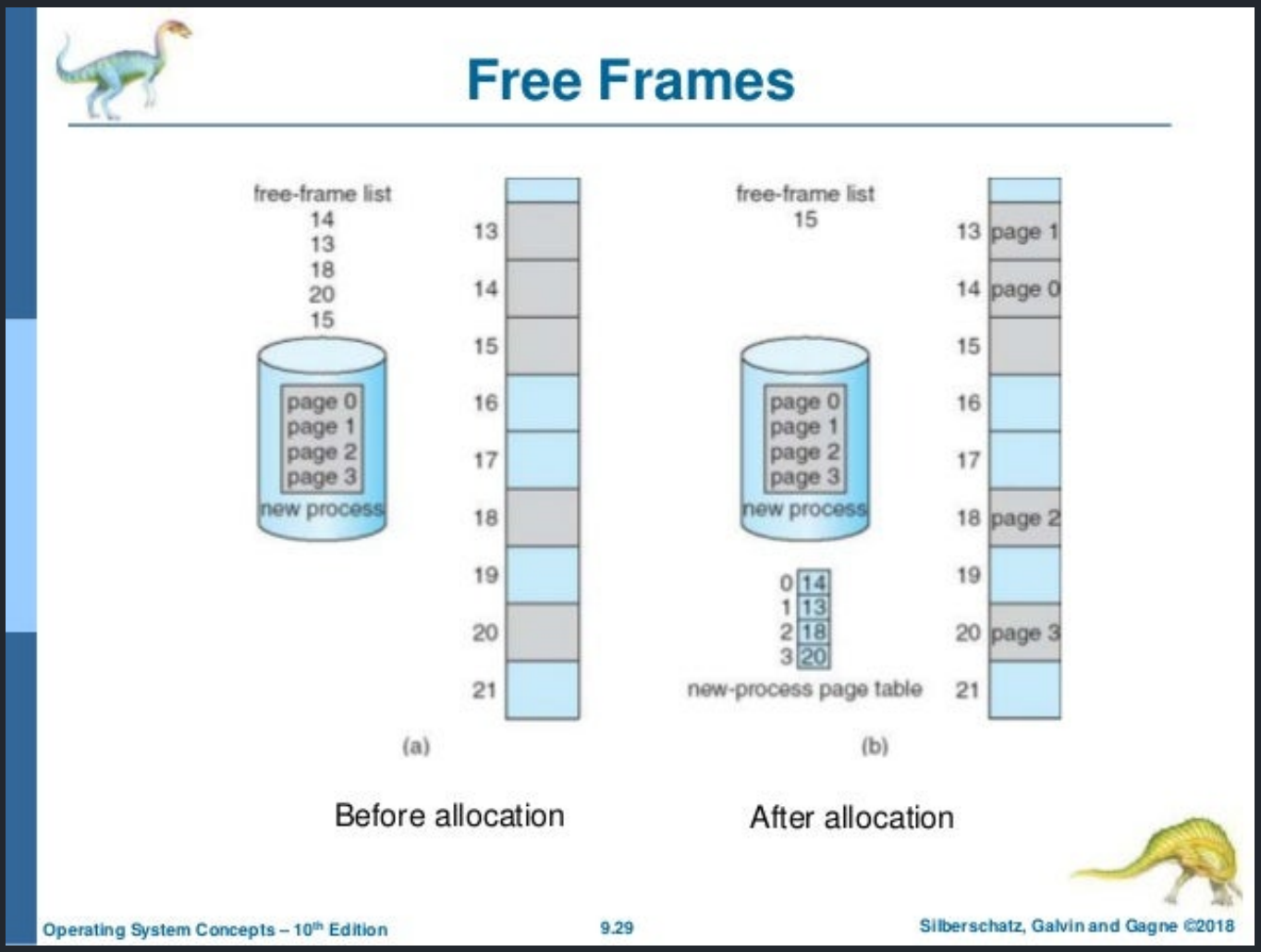
Come possiamo risolvere il problema?

Visto che i processi hanno i propri indirizzi **relativamente** vicini, e quindi probabilmente nella stessa pagina (dettata dai 22 bit più significativi), possiamo utilizzare **una sorta di cache**, che con una **singola entry** mi permette di contenere tutto ciò che è contenuto **in un'intera pagina**, ovvero 1024 indirizzi, coprendo l'intera pagina.

Questa particolare forma di cache, viene detta **TLB**, anche detta **memoria associativa**; questo perchè grazie al fatto che avendo poche entry (pagine), è possibile avere una memoria di tipo associativo (da 64 a 1024 entries).

Per capire meglio...

Entro nell'hardware paging:



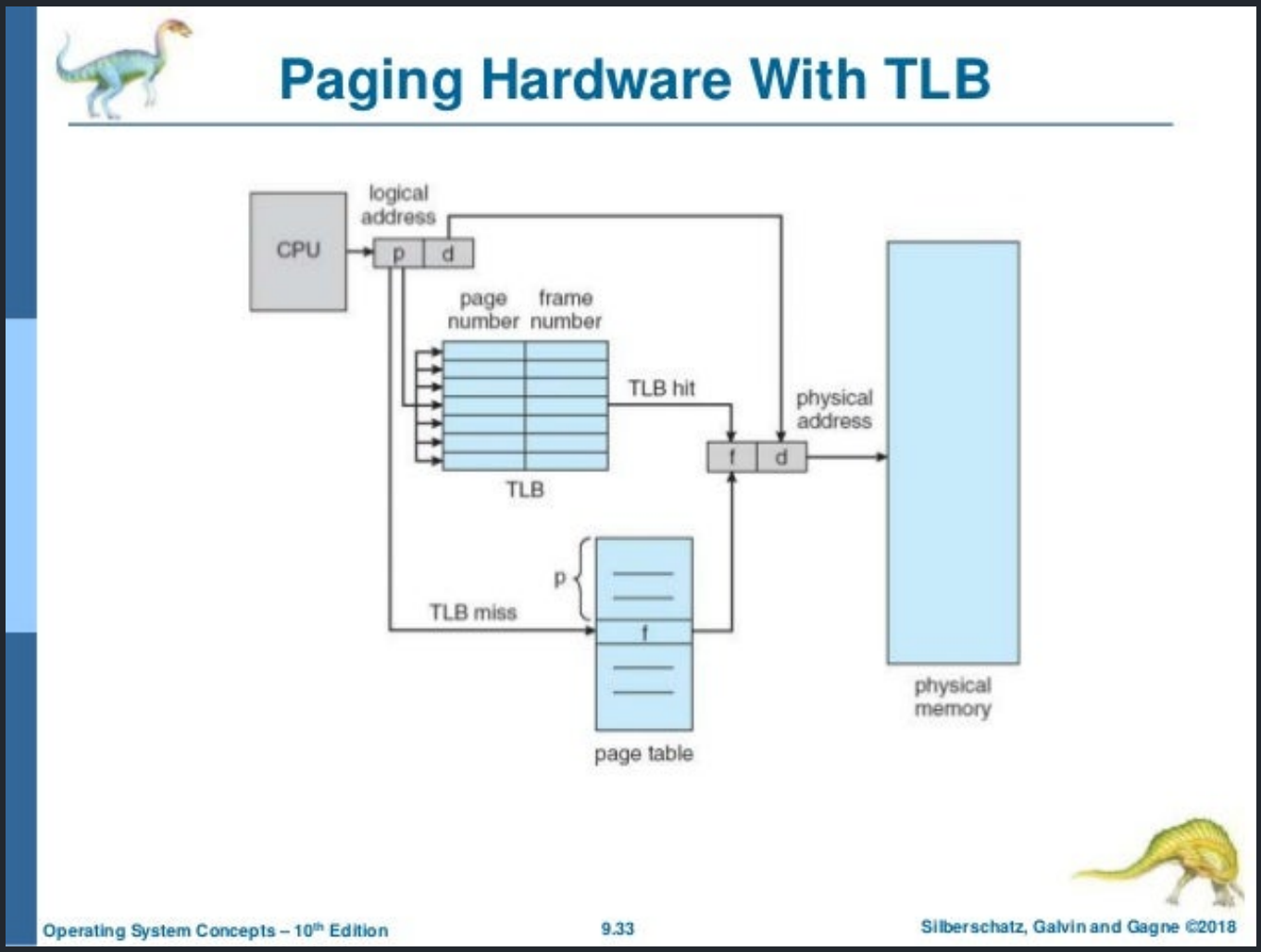
Se nel TLB (cache) pongo una coppia **pagina-frame** utilizzo la pagina come **chiave** di accesso, ed ottengo quindi il numero del frame.

Siccome il TLB non è ad accesso random, ma di tipo cache (quindi più veloce) non aumento di troppo di accesso in memoria.

Se non trovo l'indirizzo nel TLB?

Se non riesco a trovare l'indirizzo di memoria nella cache, non c'è molto altro da fare, e devo accedere alla memoria *the old fashioned way* e quindi sprecare tempo.

Tuttavia, se quella è la prima volta che effettuo l'accesso a quella specifica cella di memoria, probabilmente in un prossimo futuro mi servirà nuovamente, quindi viene salvata nel TLB per un accesso più veloce.



Quando si svuota il TLB?

Il TLB, ovvero la tabella coppia chave-valore (pagina-numero frame) deve essere svuotata ogni qualvolta il processo perde la CPU.

Questo per l'ovvio motivo che il TLB contiene la tabella degli indirizzi associata a quello specifico processo, e non al nuovo.

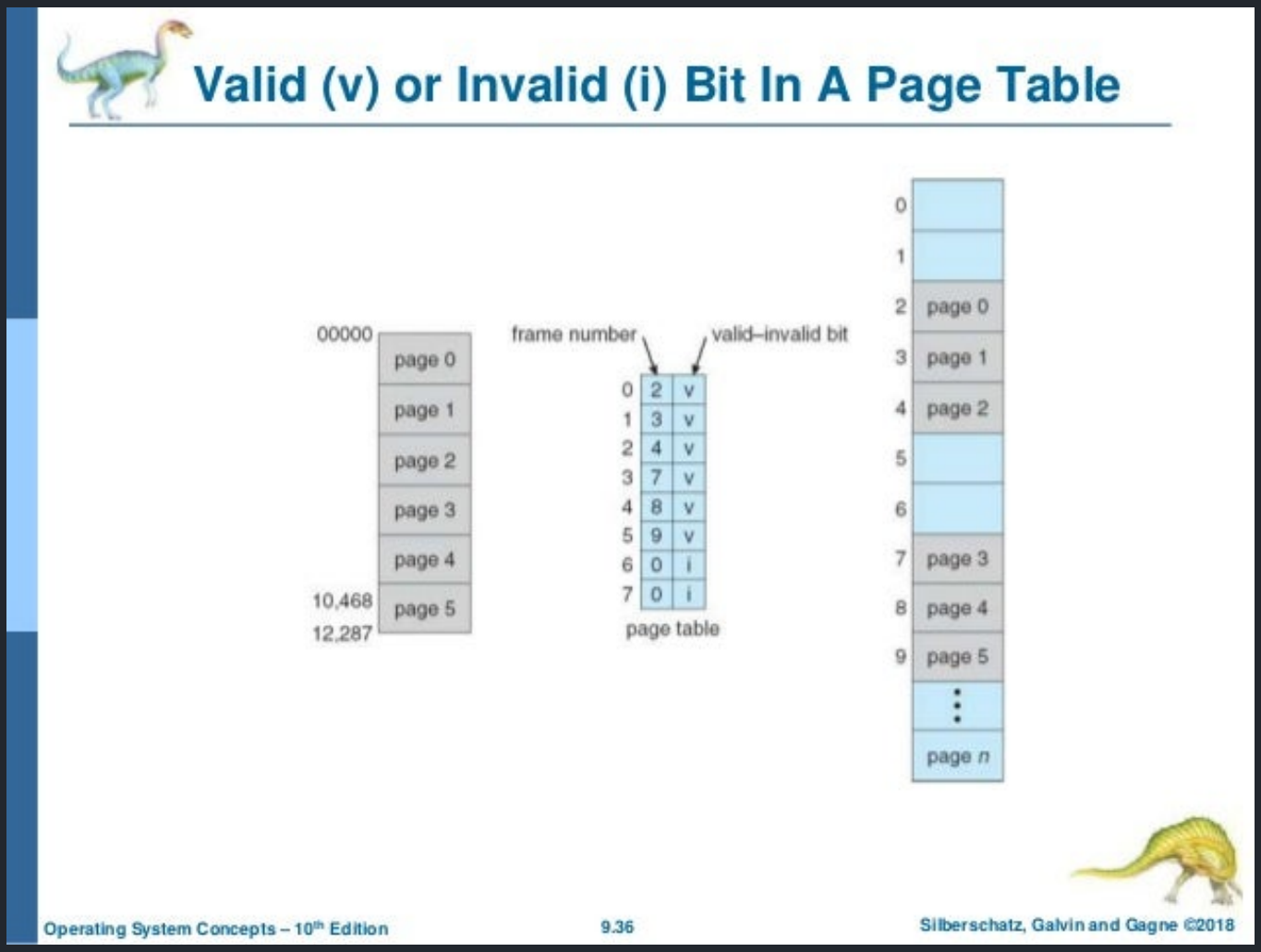
Questo è uno dei motivi per cui il context switching tra un processo e l'altro è **lento**; perchè quando arriva un altro processo e trova il TLB vuoto (perchè è stato svuotato dal processo precedente), i primi accessi alla memoria saranno tutti **miss**; essendo tutti miss, i tempi prima di "riempire" il TLB, sono lunghi.

🏁 05-21 00:54

Pagine condivise

Posso condividere una parte di memoria tra processi?

Sicuramente lo schema è fattibile se ogni processo ha la **propria tabella delle pagine**; se però voglio che dei processi abbiano un'area di memoria comune, come gestisco il tutto?



Supponiamo che il frame n1 sia un frame associato a più processi, ed è quindi un'area shared che i processi possono utilizzare.

Questo frame deve comparire, sulla tabella delle pagine, **sia su un processo che sull'altro**.

E' certamente fattibile gestire la condivisione delle pagine.

Struttura di una tabella delle pagine

Quanto è grande una tabella delle pagine? Molto!

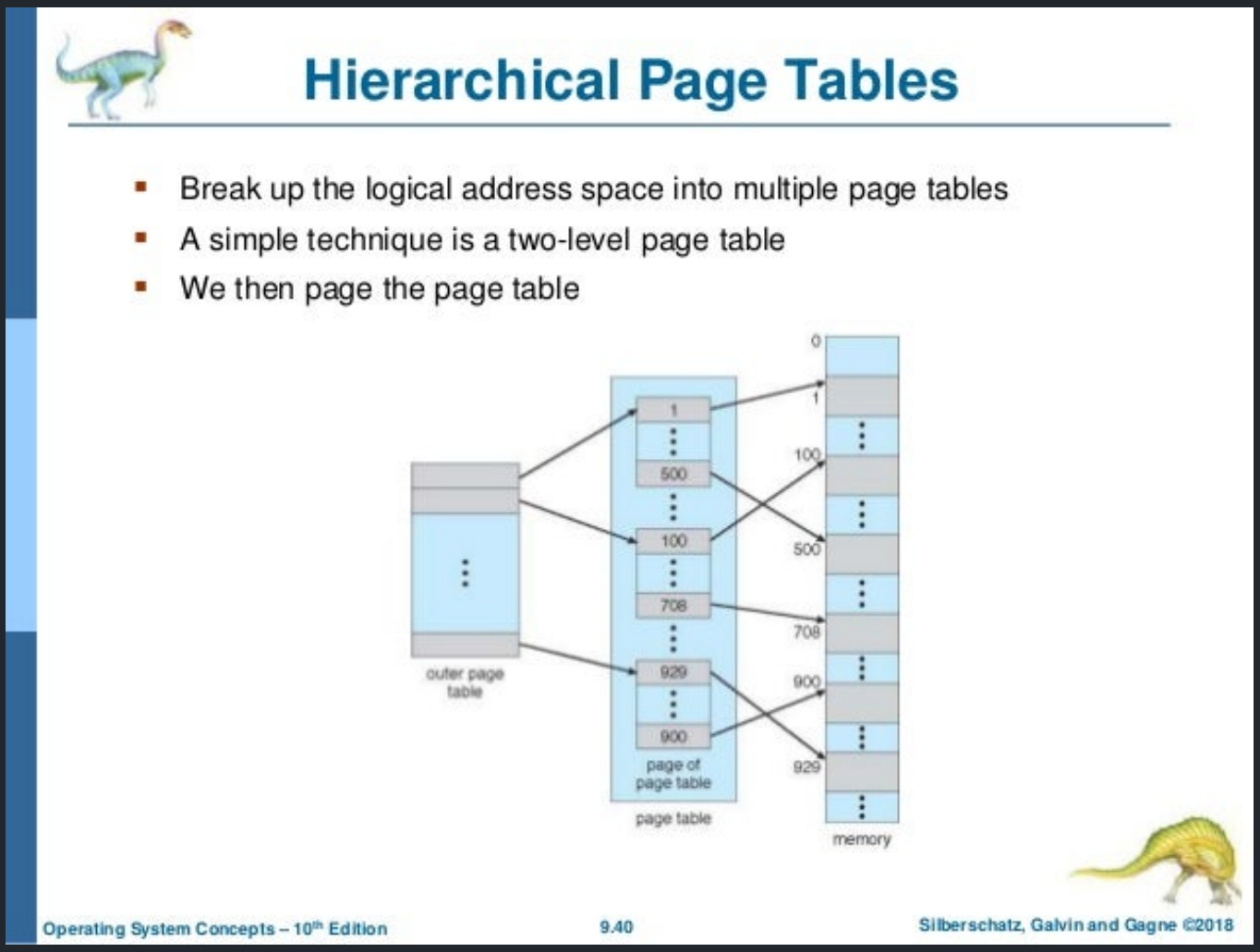
La tabella deve avere un entry per ogni possibile numero di pagina; supponiamo di avere delle pagine della grandezza di 4KB (212), se l'indirizzo è fornito con 32 bit, possiamo avere 232 / 212 = 220 ; dove i 12 bit meno significativi vengono usati come **offset** nella pagina, e i 20 bit più significativi sono il **numero delle pagine**. Con 20 bit posso avere più di **un milione*** di pagine!

Se ogni entry è di 4 bytes, ogni processo potrebbe avere 4MB di indirizzi fisici per solo la tabella delle pagine, che è un sacco di spazio.

Se invece di avere delle tabelle di 4MB, ho tante tabelle più piccole (per processo) il sistema riesce a gestire meglio lo spazio della memoria.

Tabelle delle pagine gerarchiche

All'atto pratico possiamo avere "diverse" tabelle delle pagine, organizzate in modo gerarchico:



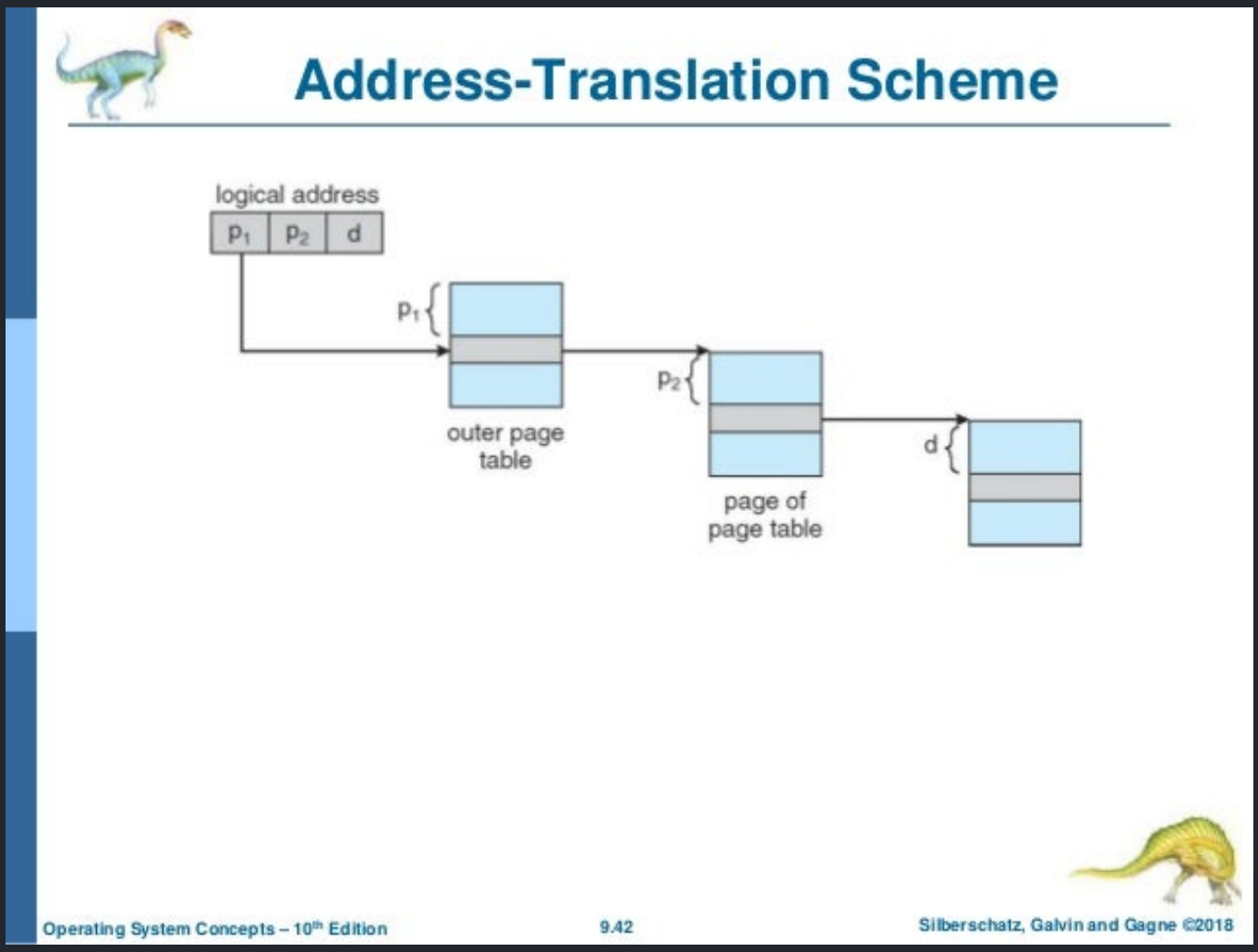
In questo caso abbiamo una tabella di primo livello che punta ad una di secondo livello, la quale punta alla tabella effettiva.

La prima tabella è chiamata **tabella delle pagine esterne**, che è quella esposta; la seconda è chiamata **tabella delle pagine** (l'effettiva tabella delle pagine), la quale punta direttamente alla memoria fisica.

I vantaggi

Questo tipo di tabelle vanno a risolvere il problema dell'unica tabella enorme; infatti queste essendo suddivise in ordine gerarchico, sono nettamente più piccole.

Traduzione degli indirizzi nelle tabelle gerarchiche



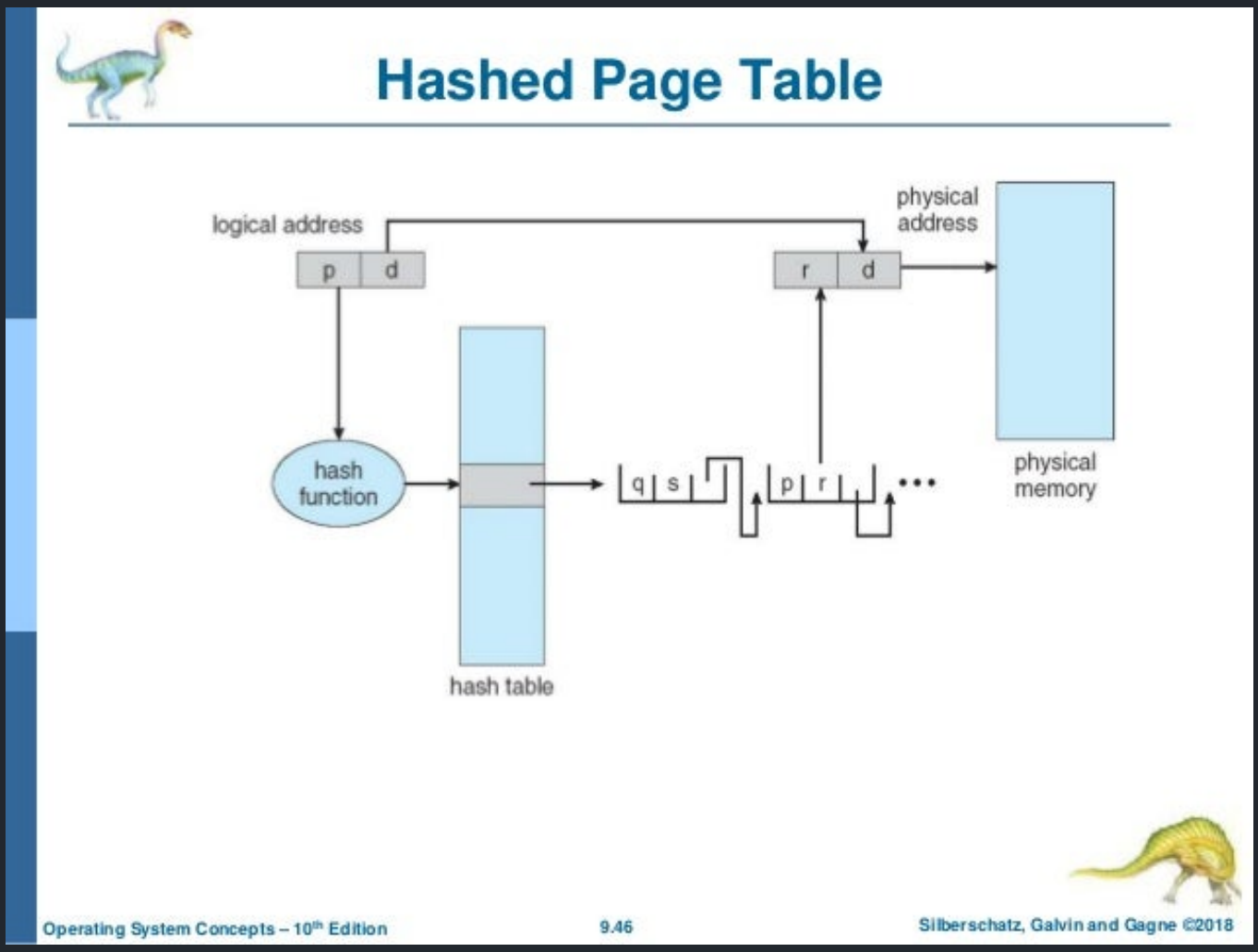
Prima l'indirizzo a 32 bit era diviso in due parti, in questo caso, che abbiamo diverse tabelle, l'indirizzo deve essere diviso ulteriormente.

Avremo quindi che i **bit più significativi** servono a posizionarmi all'interno della **outer page table**; i **bit intermedi** servono per la **tabella delle pagine** ed i bit meno significativi servono per la memoria effettiva.

🏁 05-21 1:06

Tabelle telle pagine Hashed

Molti processori utilizzano una soluzione basata su **tabelle hashed**, ovvero il campo **chiave** (numero di pagina), in hardware viene calcolata una traduzione (mediante una funzione di hashing, sperando non ci siano collisioni), in modo da poter indicizzare in modo molto più veloce:



Per evitare le collisioni utilizziamo un'array di liste con la tecnica del separate chaining.

Tecnica della tabella delle pagine invertita

Ogni processo ha la sua tabella delle pagine; ogni tanto questa copre **tutto lo spazio indirizzabile**.

L'idea della tabella delle pagine invertita, è quella di "ragionare al contrario": invece di avere per ogni processo la traduzione **pagina-frame**, avere invece **un'unica tabella** per tutti in cui vengono riportate tutte le appartenenze dei frame (i processi posseggono i frame di memoria).

Questa è una soluzione brillante perchè permette di ridurre di molto la grandezza della tabella delle pagine, la quale viene mappata sulla grandezza **fisica** (e non quella massima installabile nel sistema), e quindi non c'è possibilità che essa "ecceda".

Inoltre la tabella è unica, quindi comune a tutti i processi.

Il problema

Ovviamente il problema risiede nella **ricerca**, che non può assolutamente essere **sequenziale**, e di conseguenza sarà necessario adottare un certo meccanismo che trasformi il **pid** di ogni processo in un **hash**, dove poi andare a salvare il frame.

Segmentazione

La paginazione è una tecnica ottima per ridurre la quantità di memoria sprecata, ma non è l'unica soluzione possibile.

Storicamente, ancora prima della paginazione, il metodo adottato era quello della **segmentazione**.

Svantaggio della paginazione

Lo svantaggio maggiore della tecnica vista prima (paginazione), è il fatto di dover dividere lo spazio indirizzi di un processo in grandezza uguale alla dimensione delle pagine; questo significa che non è detto che in un'unica pagina capitino lo stesso tipo di dati: ad esempio lo stack potrebbe trovarsi diviso in due pagine.

La soluzione potrebbe essere quella di dividere la memoria non in **pagine**, ma in **segmenti**, in modo tale che in ogni segmento vengano posti dei dati che "stanno bene tra di loro"; potremmo avere quindi una divisione in segmenti del tipo:

- Programma principale
- procedure
- funzioni
- metodi
- oggetti
- ...

Il problema della segmentazione

Il problema di questo metodo, è che ovviamente tutti questi segmenti sono di **dimensione variabile**, il che significa che quando vado a salvare in memoria i segmenti, troviamo nuovamente il problema della gestione dinamica della memoria!

Come risolverlo?

Un indirizzo deve essere composto nel seguente modo:

< numero-segmento, offset >

Inoltre, Invece della tabella delle pagine, Abbiamo la **tabella dei segmenti**:
Entro nella tabella con il numero di segmento, prendo l'indirizzo di inizio, sommo l'offset ed ottengo l'indirizzo finale (traduzione simile alla paginazione).

Protezioni

Avere delle protezioni con la segmentazione è più semplice, perchè posso, giustamente, dichiarare un segmento come **sola lettura**, il segmento **non è eseguibile**, ecc.

Le protezioni sono molto importante, perchè molti degli exploit che bucano il sistema operativo, funzionano proprio in questo modo:

viene iniettato (injected) del codice esterno mediante un meccanismo, e questo viene eseguito (ad esempio dallo stack), da una zona che **non è l'area text**!

Andando a dire, quindi, che il codice può essere eseguito solo nel **segmento che contiene l'area text**, abbiamo una protezione aggiuntiva contro gli exploit.

Swapping

In un epoca passata (ancora una volta), la memoria principale era molto piccola. Quando l'utente voleva alternare molti processi (interattivi, con diversi terminali remoti) velocemente, capitava spesso che la memoria *finisse*.

Per risolvere questo problema, si usava la tecnica dello **swapping**, ovvero, per fare spazio in memoria, si prendeva **l'intero processo con il suo spazio indirizzi**, e lo scaricava su una memoria di appoggio; questa memoria era il disco più veloce presente all'interno della macchina.

In questo modo si fa spazio per un nuovo processo da eseguire.

Questa tecnica è ancora in uso?

Questa tecnica, ai nostri giorni, non viene praticamente più utilizzata; Microsoft chiama la memoria virtuale "**Spazio Di Swap**" impropriamente, in questo schemo lo swap vero e proprio si verifica quando **l'intero processo esce dalla ram ed un intero processo entra**.

Cosa si inventa intel dopo la segmentazione?

Multics era un sistema operativo abbastanza elaborato, così elaborato da far "perdere le speranze" ai suoi programmatori, che decisero di creare un nuovo sistema operativo molto più semplice: **UNIX**.
