Memòria virtual i traducció d'adreces

Lluís Garrido – lluis.garrido@ub.edu

Grau d'Enginyeria Informàtica

Traducció d'adreça: memòria paginada

Recordem el codi `exemple_fork.c`

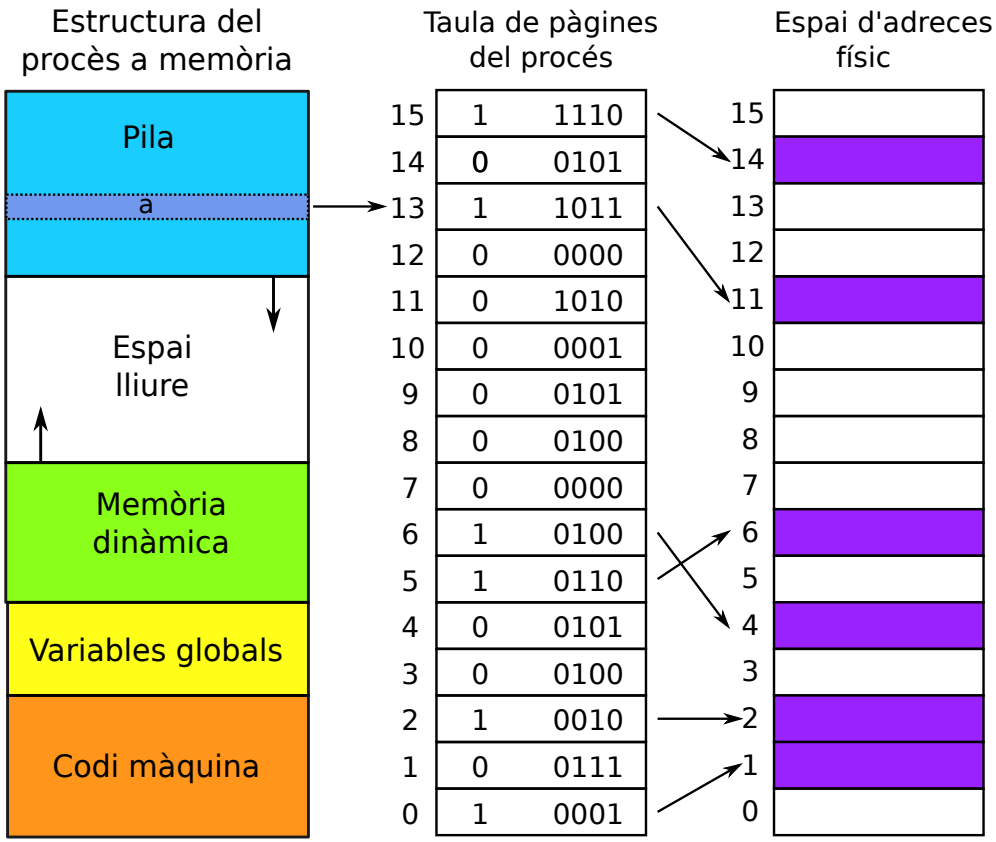
```
Direccio d'a abans fork:  140723702953032
Fill valor d'a:  2
Fill direccio d'a:  140723702953032
Pare valor d'a:  1
Pare direccio d'a:  140723702953032
```

Tot i que es canviï de valor la variable "a" al pare i/o al fill, l'adreça de memòria no canvia. Això és perquè s'estan imprimint per pantalla les adreces virtuals!

Pel cas del `fork` es fa servir, a més, una tècnica anomenada Copy-On-Write (COW) per optimitzar el `fork`.

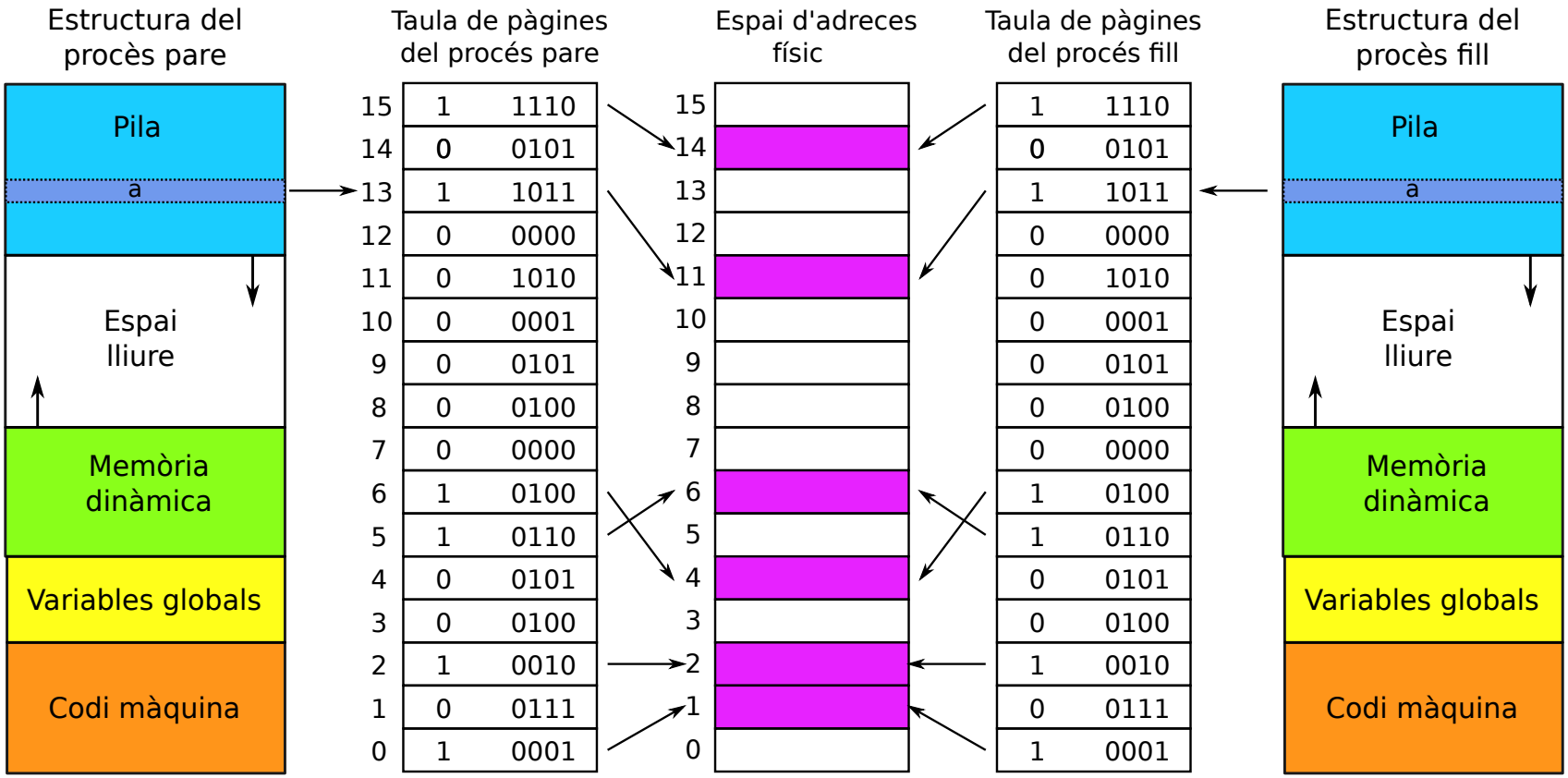
Traducció d'adreça: memòria paginada

Mapa de memòria abans de realitzar el fork



Traducció d'adreça: memòria paginada

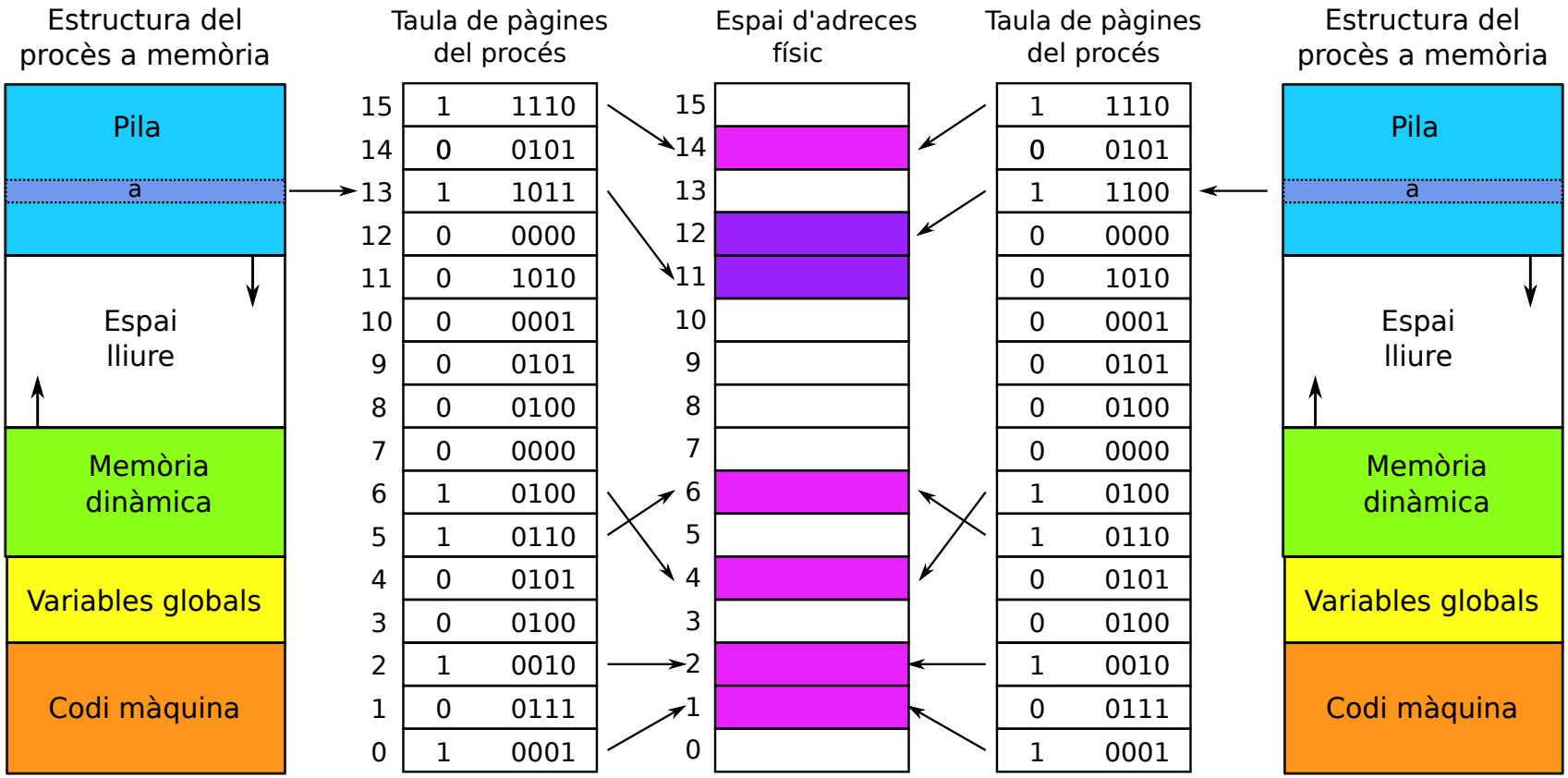
Mapa de memòria després de realitzar el fork



1. El sistema operatiu duplica la taula del procés (però no pas la informació física associada).
2. A la taula es marquen les pàgines físiques com a només de lectura.

Traducció d'adreça: memòria paginada

Mapa de memòria després de realitzar l'assignació a = 2 al fill.



1. En realitzar el fill a = 2 es produeix una fallada de pàgina. El SO captura la fallada i duplica les pàgines físiques.
2. Es retorna de la fallada i el procés continua l'execució com si res hagué passat...

Traducció d'adreça: memòria paginada

Tècnica Copy-On-Write (COW)

- Les pàgines es marquen com a de lectura i es copien només quan hi accedim per escriptura. Només es copien aquelles pàgines que es modifiquen, la resta són compartides entre pare i fill.

Compartició de memòria

- Observar que el codi màquina es comparteix entre pare i fill (ja que no es modifica).
- De forma genèrica, compartir zones de memòria (llibreries, codi exectuable, ...) és senzill a memòria paginada. El sistema operatiu allibera els marcs associats així que el darrer procés que hi ha referència deixa de fer servir aquella part de memòria.

Traducció d'adreça: memòria paginada

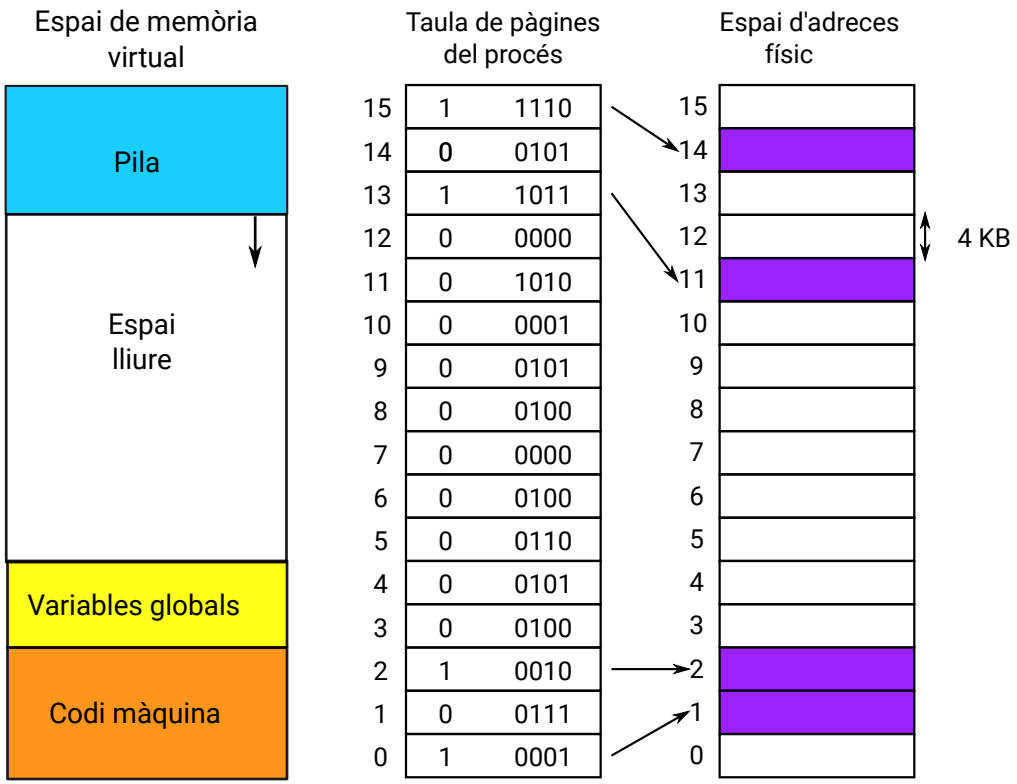
Què passa en fer un malloc?

- Aquesta funció només inicialitza el mapa de memòria virtual.
- L'assignació de marcs de pàgina físics es fa a mesura que hi accedim realment.

Veure codi `exemple_malloc.c`.

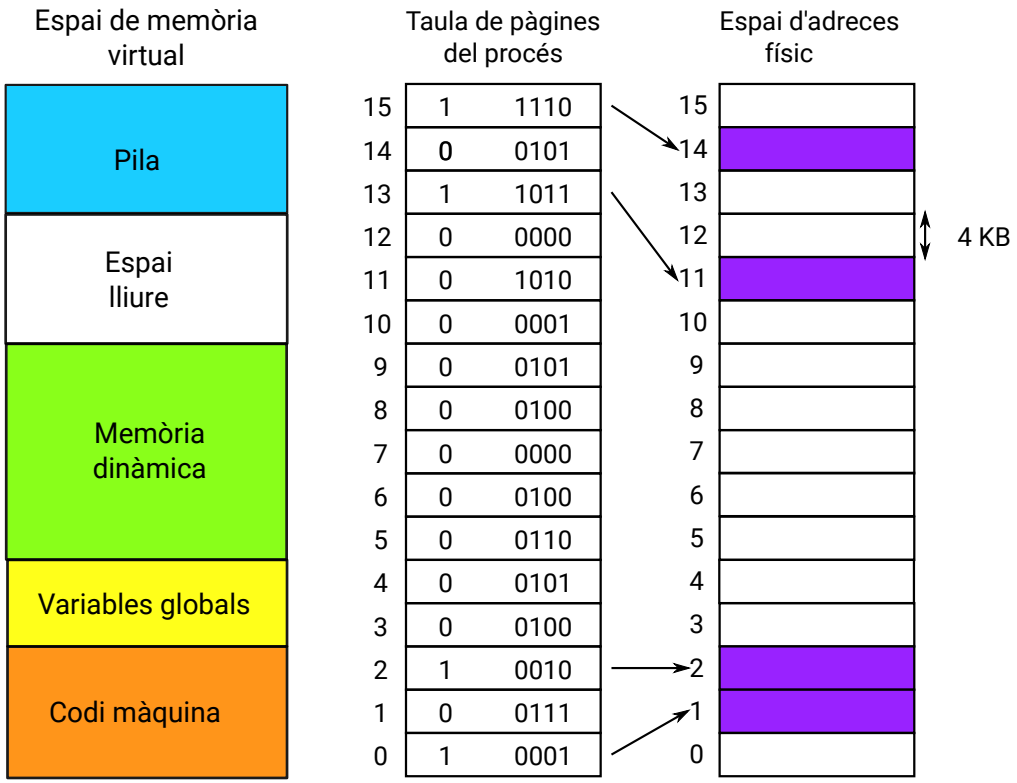
Traducció d'adreça: memòria paginada

Mapa de memòria abans del malloc (amb arquitectura de 16 bits)



Traducció d'adreça: memòria paginada

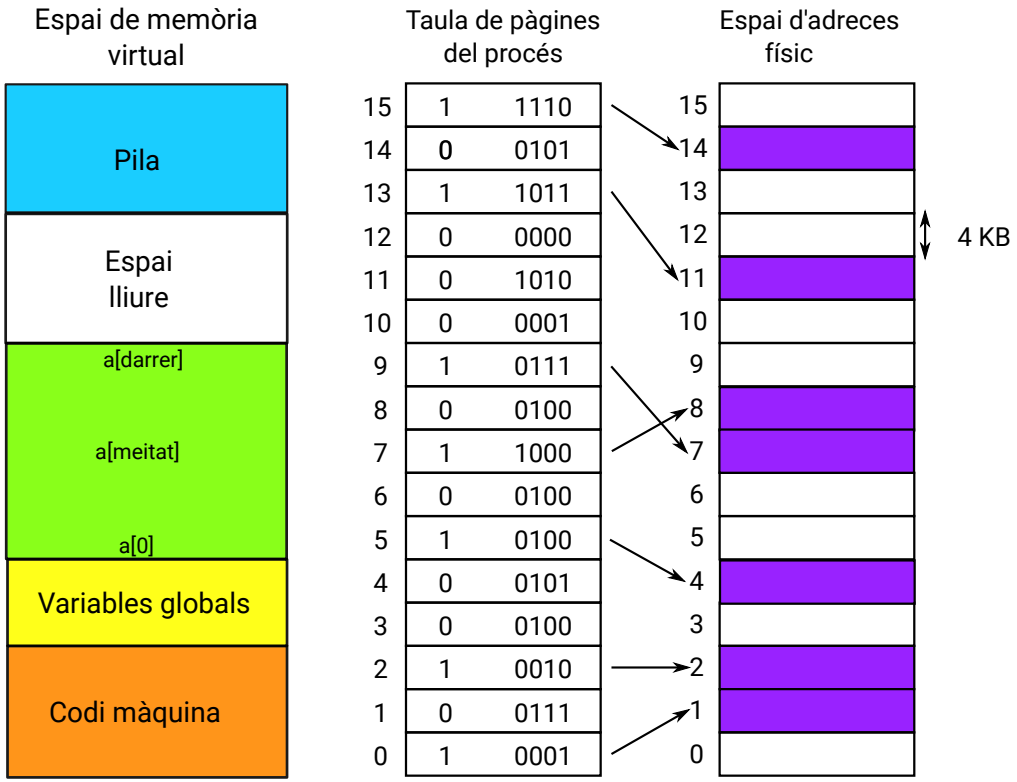
Mapa de memòria després del malloc (amb arquitectura de 16 bits)



Un cop fet el malloc només s'ha fet la reserva a l'espai virtual, no pas a l'espai físic.

Traducció d'adreça: memòria paginada

Mapa de memòria després dels accessos (arquitectura de 16 bits)



1. Si l'adreça no existeix a l'espai físic es produeix una fallada de pàgina. El SO "cercar" una pàgina disponible a l'espai físic.
2. Es retorna de la fallada i es realitza l'accés...

Traducció d'adreça: memòria paginada

Per al codi `exemple_malloc.c`.

- En cridar `malloc` només es fa la reserva a l'espai virtual, no pas a l'espai físic. El sistema operatiu assigna espai físic en accedir a una determinada posició del vector.
- Per a l'exemple donat hem reservat molts GBytes però físicament només s'assignen 3×4 Kbytes de memòria! El sistema operatiu ha de decidir quin marc de pàgina físic utilitzar.

Traducció d'adreça: memòria paginada

La memòria paginada resol els principal problema de la memòria segmentada: trobar zones de memòria lliures.

- Per això el sistema operatiu només ha de mantenir una taula (un bitmap) amb els marcs de pàgina lliures. Per trobar una pàgina lliure només cal recórrer la taula i buscar un bit que indiqui que la pàgina és lliure.
- La memòria física és limitada: pot arribar un moment en què tots els marcs de pàgines físiques estiguin ocupats. Si un procés en requereix nous marcs, el sistema operatiu ha de decidir quins marcs "descarta" de l'espai de memòria física. Ho veurem!
- En fer un `exec` el sistema operatiu no carrega a memòria tot l'executable: només inicialitza el mapa de memòria virtual indicant que les pàgines no es troben a memòria física. A mesura que es produeixen "excepcions" per executar el codi, el sistema operatiu carrega de disc la pàgina de codi executable corresponent...

Traducció d'adreça: memòria paginada

Altres punts a tractar

- En un sistema x86 de 32 bits els marcs físics tenen mida de $2^{12} = 4096$ bytes. Això vol dir que la taula té $2^{20} = 1048576$ entrades. Si cada entrada ocupa 4 bytes significa que una taula ocupa 4Mbytes, i hi ha una per a cada procés!
- Què passa en un sistema de 64 bits? La taula ocuparia un munt a memòria! Podríem augmentar la mida del marc físic (que a l'actualitat continua sent de 4KB), però malbarataríem memòria RAM. Aleshores, què?

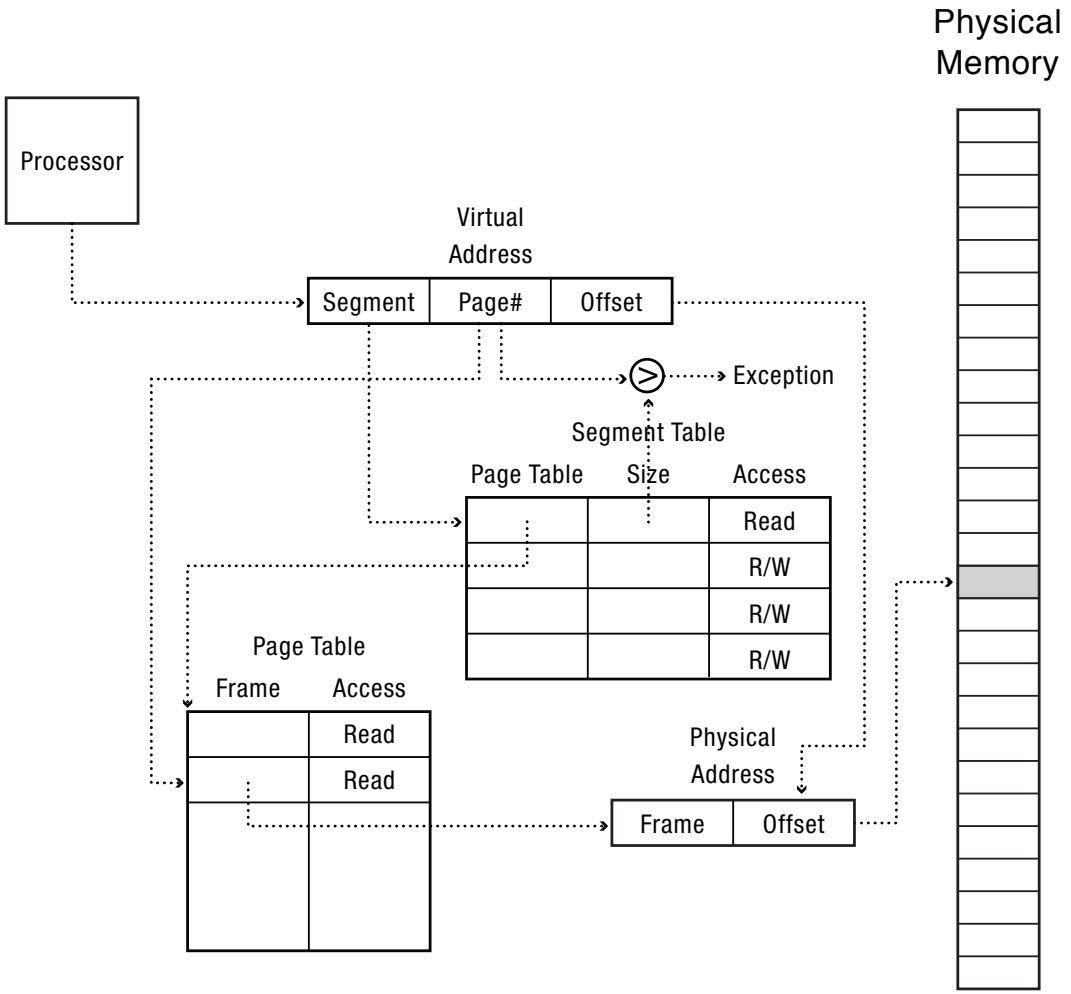
Traducció d'adreça: memòria paginada

La solució en sistemes de 64 bits és no fer servir una taula! Es pot fer servir un arbre o una taula de hash... tots dos s'utilitzen en sistemes reals. De forma genèrica són esquemes de traducció multinivell.

- Tots els sistemes multinivell utilitzen l'esquema de paginació al nivell més fi (a les fulles). La diferència està en com arribar, a traves dels nivells, al nivell més fi.
- Anem a veure aquests esquemes d'arbre: paginació segmentada, paginació multinivell i paginació segmentada multinivell.

Traducció d'adreça: memòria paginada

Amb paginació segmentada l'arbre té dos nivells: el 1r nivell, el segment, apunta a una taula, del 2n nivell, que apunta als marcs de pàgina físics.



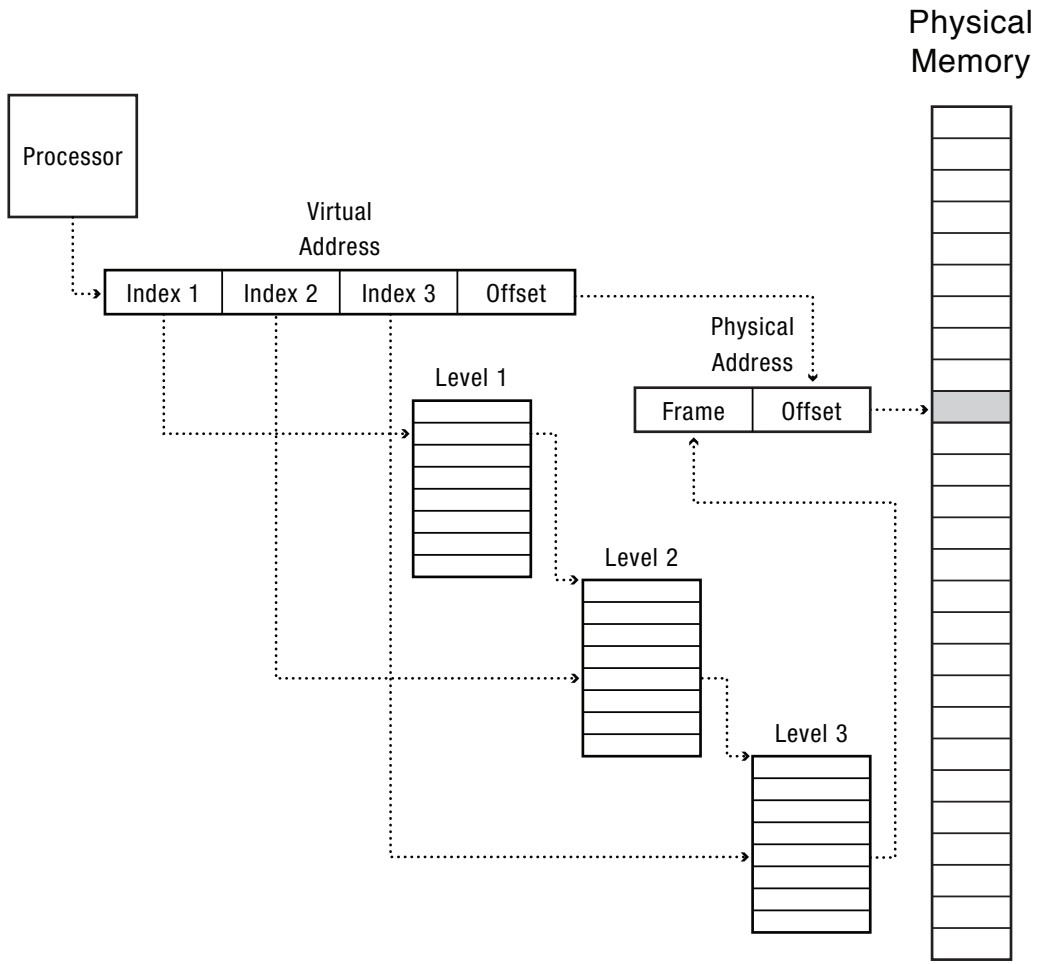
Traducció d'adreça: memòria paginada

Característiques

- Les propietats d'accés (lectura, escriptura, execució, ...) es poden establir al nivell de segment.
- En sistemes de 32 bits, per exemple, es poden assignar els 10 primers bits al segment, els 10 següents a l'índex al marc de pàgina, i els 12 darrers a l'offset dintre de la pàgina (marc de pàgina de 4096 bytes).

Traducció d'adreça: memòria paginada

Amb paginació multinivell l'arbre té múltiples nivells. Només s'ubiquen les parts de l'arbre que realment són utilitzades pel procés.



Traducció d'adreça: memòria paginada

Es poden combinar els dos esquemes anteriors per obtenir un esquema de paginació multinivell segmentada: cada segment conté una taula multinivell. És l'esquema utilitzat pels sistemes x86, tant per 32 com per 64 bits.

- Per qüestions històriques, el número de "segment" s'emmagatzema en un registre de CPU separat.
- En sistemes de 32 bits, s'utilitza un esquema de dos nivells: 10 bits al primer nivell, 10 al segon i 12 per l'offset.
- En sistemes de 64 bits, actualment, només s'utilitzen 48 bits de l'espai d'adreces virtual. Això permet adreçar 128 Terabytes. S'utilitzen 4 nivells d'arbre ($48 = 9+9+9+9+12$). Només s'ubiquen les parts de l'arbre que realment són utilitzades pel procés.

Traducció d'adreça: memòria paginada

Tot això funciona molt bé, però... les taules que fan la traducció es guarden a memòria RAM.

- Suposem que volem accedir al valor d'una variable "a". Quants accessos a memòria fan falta per accedir al valor de la variable "a"? Penseu-hi...
- Suposem que volem accedir al valor d'un element d'un vector, "a[10]". Quants accessos a memòria fan falta pera accedir al valor de "a[10]"? Penseu-hi...

Com arreglem aquesta "ineficiència" a l'hora d'accedir a adreces de memòria?

Traducció d'adreça: eficiència en la traducció

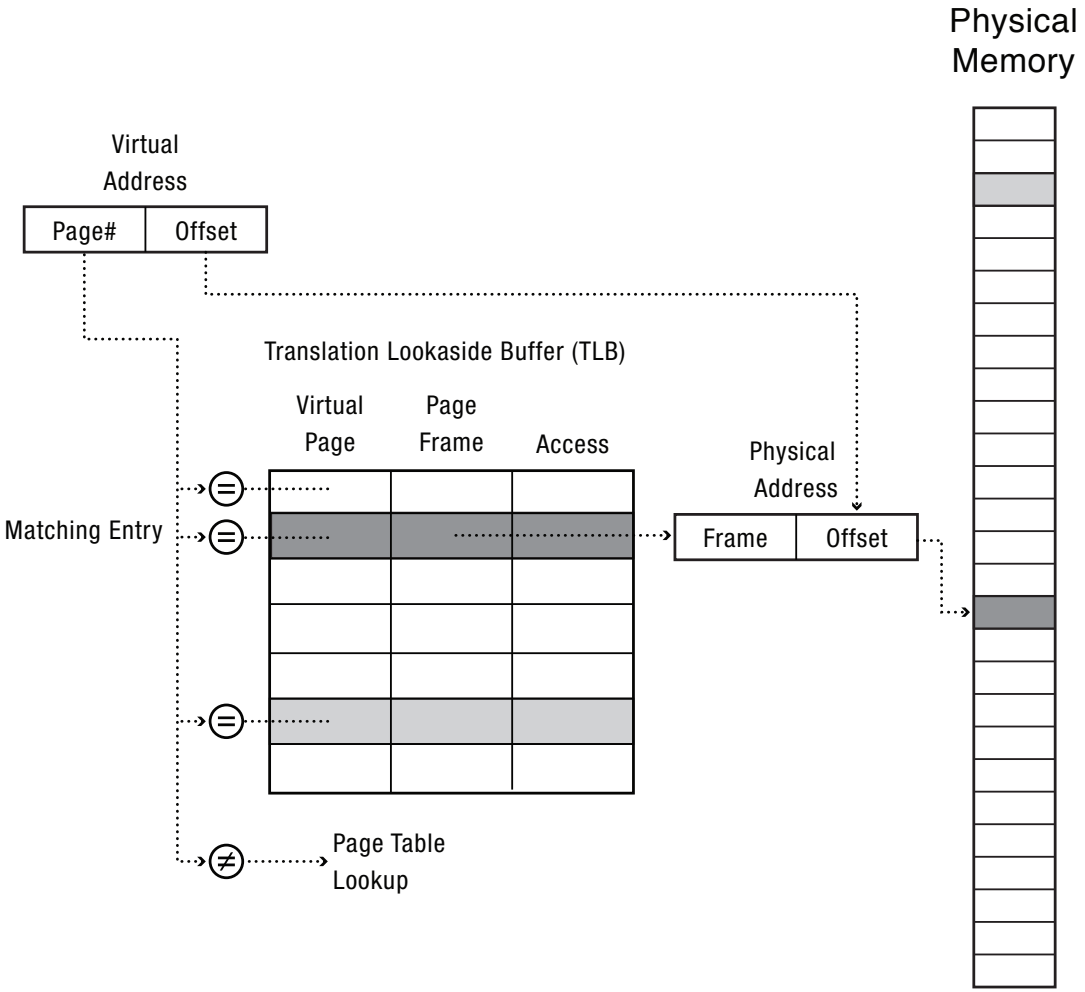
Com podem millorar l'eficiència de la traducció sense perdre en la versatilitat que ens ofereix el sistema de traducció?

- Suposem que s'executa una instrucció emmagatzemada en una determinada posició de memòria. S'espera que la següent instrucció que s'executi sigui la que està a la posició següent de memòria (a l'espai virtual).
- Suposem que accedim a un vector de dades. Normalment els accessos a les posicions del vector es realitzen de forma correlativa.

Per evitar haver d'accedir cada cop a l'estructura multinivell es pot utilitzar una memòria cau (*cache*)! Es guardarà a una taula separada la traducció per les pàgines accedides més "habitualment".

Traducció d'adreça: eficiència en la traducció

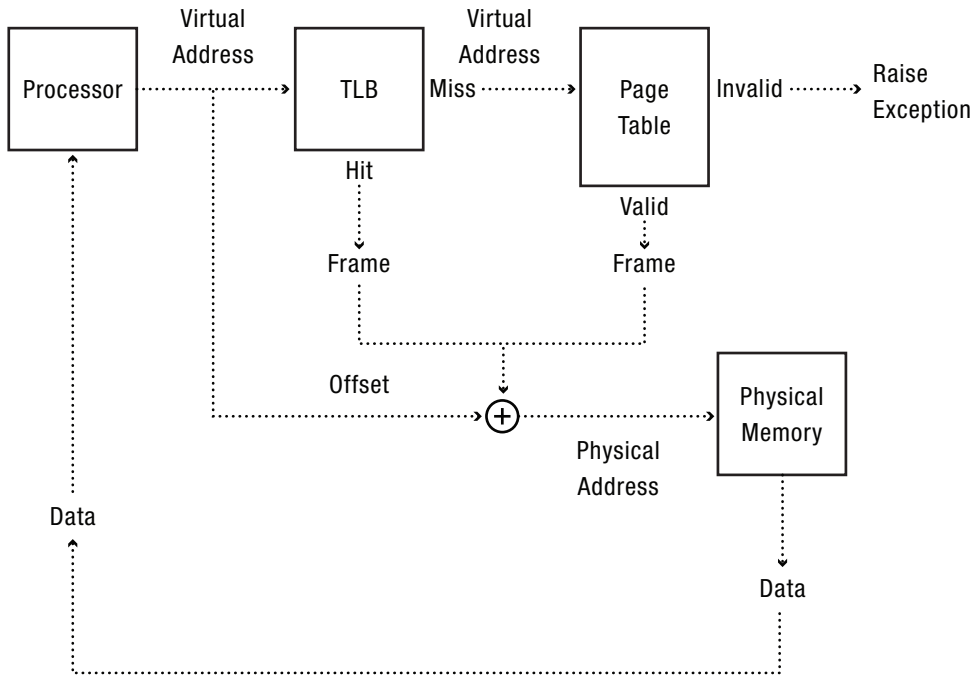
La Translation Lookaside Buffer (TLB) implementa aquesta funcionalitat (típicament) a nivell de maquinari.



Traducció d'adreça: eficiència en la traducció

Es comproven simultàniament totes les entrades de la TLB amb l'adreça a traduir.

- Si hi ha una coincidència, un *TLB hit*, s'utilitza l'entrada corresponent i no cal fer servir la informació multinivell.
- Si no hi ha coincidència, un *TLB miss*, el maquinari realitza la traducció multinivell tal com hem vist anteriorment.



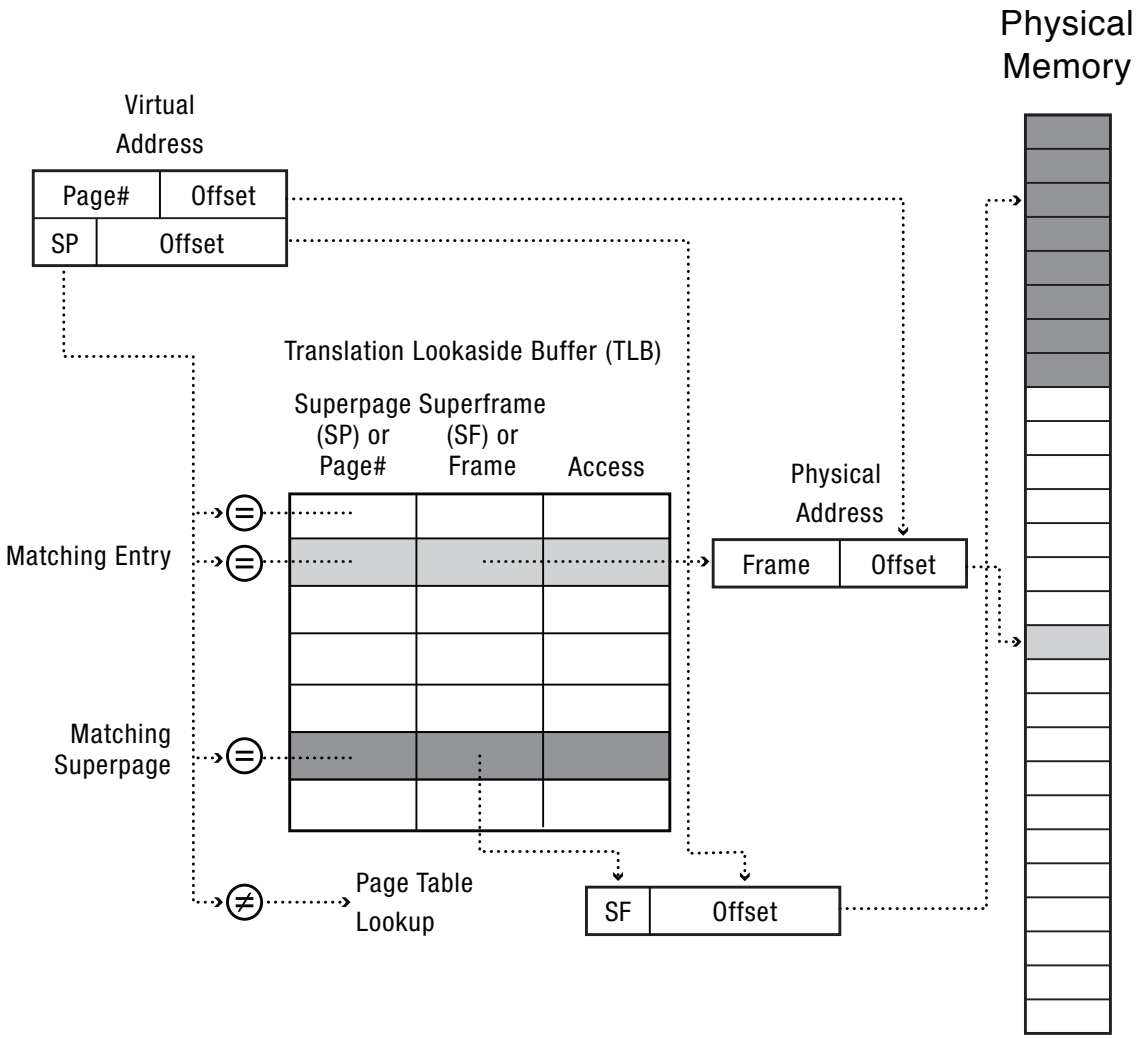
Traducció d'adreça: eficiència en la traducció

Es pot millorar encara més l'eficiència...

- Amb l'esquema anterior cada "entrada" de la TLB es correspon a una fulla de l'arbre, 4096 bytes.
- A l'actualitat es permeten "superpàgines" a la TLB: una superpàgina es correspon a múltiples pàgines contigües a memòria física. En concret, per a sistemes x86
  - Ens podem estalviar el quart nivell de l'arbre si es pot mapar una regió de $2^{9+12} = 2^{21} = 2$Mbytes de forma contigua.
  - De forma similar, ens estalviem també el tercer nivell de l'arbre si es pot mapar una regió de $2^{9+9+12} = 2^{30} = 1$Gbyte de forma contigua.

Traducció d'adreça: eficiència en la traducció

Esquema de la traducció amb "superpàgines"



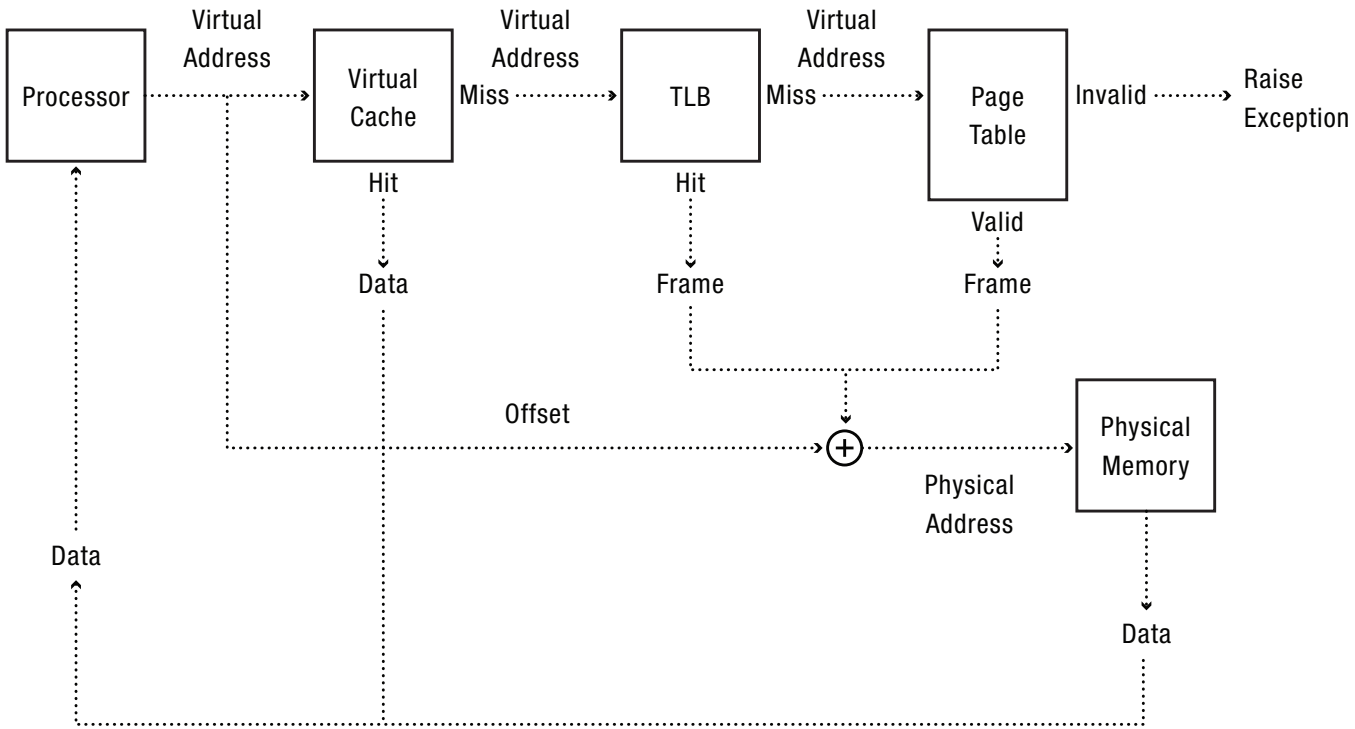
Traducció d'adreça: eficiència en la traducció

Quina és la tasca del sistema operatiu?

- El sistema operatiu té capacitat per gestionar la TLB. Pot decidir introduir o eliminar elements (per exemple, superpàgines).
- En fer un canvi de context caldria invalidar la TLB ja que la informació que hi ha és per al procés. Actualment, però, la TLB és capaç de mantenir una còpia de la TLB de diversos processos.

Traducció d'adreça: eficiència en la traducció

L'eficiència es pot augmentar encara més si utilitzem una *cache* de dades abans de la TLB.



Traducció d'adreça: eficiència en la traducció

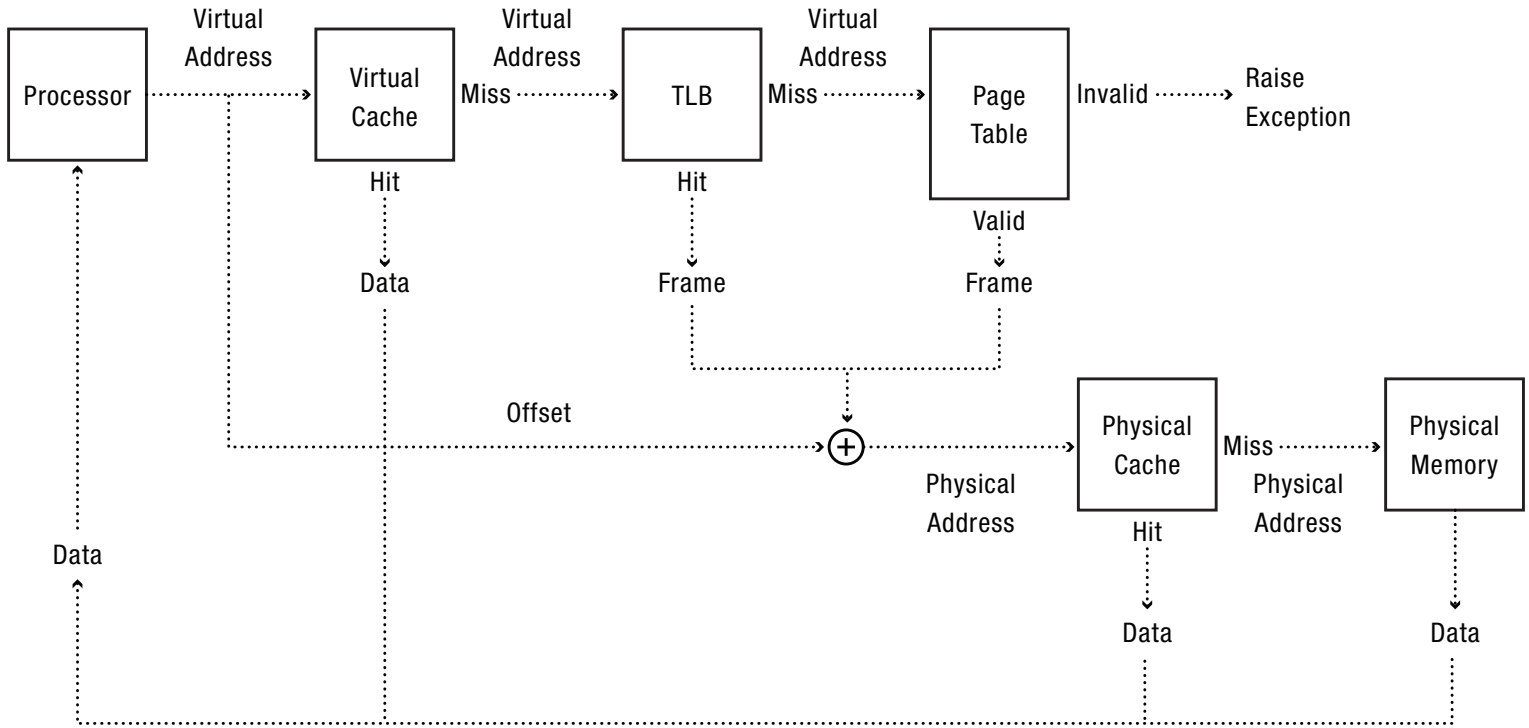
Aquesta *cache*

- Està indexada per adreces virtuals i emmagatzema una còpia del contingut de la memòria física associada.
- Si es troba l'adreça virtual a la *cache* es retorna la dada demanada i no cal consultar ni la TLB ni el sistema multinivell per generar l'adreça física.
- Acostuma a ser una memòria petita i molt ràpida.

És el que habitualment s'anomena "*cache* de 1r nivell" i està incorporada en pràcticament tots els processadors.
Provar el codi `matmul.c` amb diferents mides de la matriu. El temps de càlcul depèn molt de si la matriu cap o no a la memòria cau.

Traducció d'adreça: eficiència en la traducció

Podem augmentar encara més l'eficiència... Moltes arquitectures incorporen també una *cache* que es troba després de la TLB.



Traducció d'adreça: eficiència en la traducció

Una vegada la TLB o la traducció multinivell ens ha fet la traducció de memòria virtual a memòria física, aquesta *cache*

- Està indexada per adreces físiques i emmagatzema una còpia del contingut de la memòria física associada.
- Si es troba l'adreça física a la *cache*, es retorna la dada demanada.

Quin sentit té tenir una *cache* indexada per adreces físiques?

- En produir-se un "TLB miss", cal fer servir les taules multinivell. Les taules s'emmagatzemen a memòria física i, per tant, els valors dels diferents nivells poden guardar-se en aquesta *cache*.

Aquesta cache és el que habitualment es coneix com a "*cache* de 2n nivell" (una per cada *core* d'un processador) o "*cache* de 3r nivell" (compartit entre els *cores* del processador).

Contingut

El contingut d'aquestes transparències és

1. Concepte de traducció d'adreça. Els conceptes bàsics associats
2. Traducció d'adreça flexible. Com es pot dissenyar el maquinari per proveir de la màxima flexibilitat al sistema operatiu.
3. Traducció eficient d'adreça. Quins mecanismes es poden utilitzar per fer una traducció eficient de l'adreça?
4. Traducció d'adreces a nivell de programari. Cada vegada més s'implementa la funcionalitat de traducció a nivell de programari. Què tal tenir en compte?

Centrem-nos en el darrer punt

Protecció de memòria a nivell de programari

Cada dia apareixen més sistemes en què el programari complementa els mecanismes de maquinari

- Idealment podríem utilitzar un intèrpret de codi ensamblador: agafa la instrucció de memòria, la interpreta, comprovar si les direccions són vàlides, ... això seria molt lent.
- Quines tècniques podem fer servir aleshores per executar codi en un entorn restringit?
- A l'actualitat la protecció es realitza pràcticament sempre a nivell de maquinari. Quin sentit té fer-ho a nivell a programari?

Protecció de memòria a nivell de programari

Avantatges d'implementar la protecció a nivell de programari

- Simplificació del maquinari: amb una implementació a nivell de programari disposaríem de molta més flexibilitat.
- Protecció a nivell d'aplicació: una aplicació es podria protegir a l'hora d'executar codi de tercers (navegadors web, ...)
- Protecció a nivell de nucli: a l'actualitat també s'executa codi no fiable al nucli del sistema operatiu (gestors o "*driver*" de dispositius).
- Portabilitat: es permet que l'aplicació executi a múltiples dispositius diferents (smartphone, tablet, netbook, ...) amb les mesures de seguretat necessàries.

El terme *sandbox* es fa servir per denotar la protecció a nivell de programari necessària per executar codi no fiable.

Protecció de memòria a nivell de programari

Una forma ben senzilla d'implementar protecció es assegurar que totes les aplicacions estan escrites en el mateix llenguatge de programació

- Diversos dels primers ordinadors personals implementaven així els mecanismes de protecció: Xerox va desenvolupar el llenguatge Mesa, inspirador del Java; altres ordinadors només executaven Lisp, i d'altres Smalltalk, precursor del Python.
- A l'actualitat moltes aplicacions utilitzen llenguatges propis per controlar la seguretat: per exemple, Javascript, Python, ...
- Cal assegurar però que l'intèrpret i les llibreries que utilitzen són fiables, cosa que no és fàcil. Una errada en l'intèrpret o la llibreria pot ser aprofitada per realitzar un atac.
- Interpretar codi és lent: per això moltes llibreries es compilen directament en codi ensamblador. Amb la qual cosa tornem al nostre problema inicial... com protegim?

Protecció de memòria a nivell de programari

Alguns enfocs

- Escriure tot el codi en un llenguatge segur i compilar el codi en instruccions que executen directament al processador. En el sistema Xerox Alto es van implementar tant el sistema operatiu com les aplicacions en Modula-2. Però cal assegurar-se que el compilador és fiable!
- La majoria de sistemes combinen actualment la protecció a nivell de programari i de maquinari. Per exemple, Microsoft Windows executa el seu navegador en un procés especial amb permisos restringits per assegurar seguretat.

Protecció de memòria a nivell de programari

A l'actualitat existeixen múltiples llenguatges de programació... com podem protegir aillar una aplicació fent servir programari (sense maquinari), de forma independent al llenguatge de programació?

- Seria molt interessant poder-ho: els navegadors, el sistema operatiu, aplicacions de bases de dades, ... acostumen a utilizar codi de tercers.
- Google i Microsoft tenen productes, Native Client i AppDomain respectivament que, a partir d'un codi ensamblador qualsevol, el modifiquen per assegurar que no accedeixen a posicions de memòria fora de l'espai que tenen assignat. Tècnicament estem executant el codi dins d'un *sandbox*. Segons Google, la "degradació" per executar el codi modificat al processador és menys d'un 10%.

Protecció de memòria a nivell de programari

També s'està treballant en implementar *sandboxes* a partir de codi intermedi en comptes de codi ensamblador

- El codi intermedi un llenguatge associat a una màquina abstracte que permet analitzar programes d'ordinador. El codi itermedi es portable entre diverses arquitectures.
- El sandbox genera el codi segur a partir d'aquest codi intermedi.
- La màquina virtual de Java es un tipus de sandbox: el codi Java es tradueix a codi intermedi. En temps d'execució es pot comprovar que les instruccions estan contingudes dins del sandbox.