Present bit: support swapping pages

Assume a hardware managed TLB. what happens during a translation?

extract VPN → checks the TLB for a match
- → TLB hit → ✓
- → TLB miss

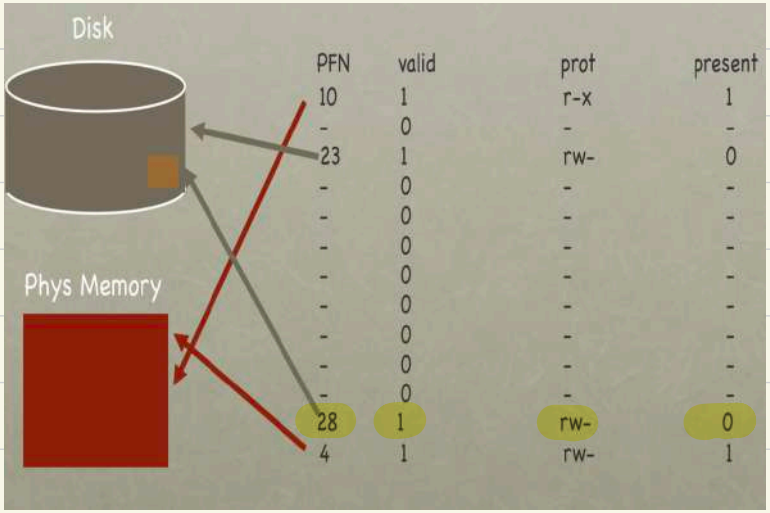
(but this page might not be present in physical memory)

indicated by Present bit.

the hardware locates the PTE

Present = 1
- go to the physical address,
- execute instruction

Present = 0
- ⇒ "Page-fault"
- ↳ OS is invoked to service the page-fault.
- page-fault handler runs



Note: Even with a hardware-managed TLB, OS handles page-fault

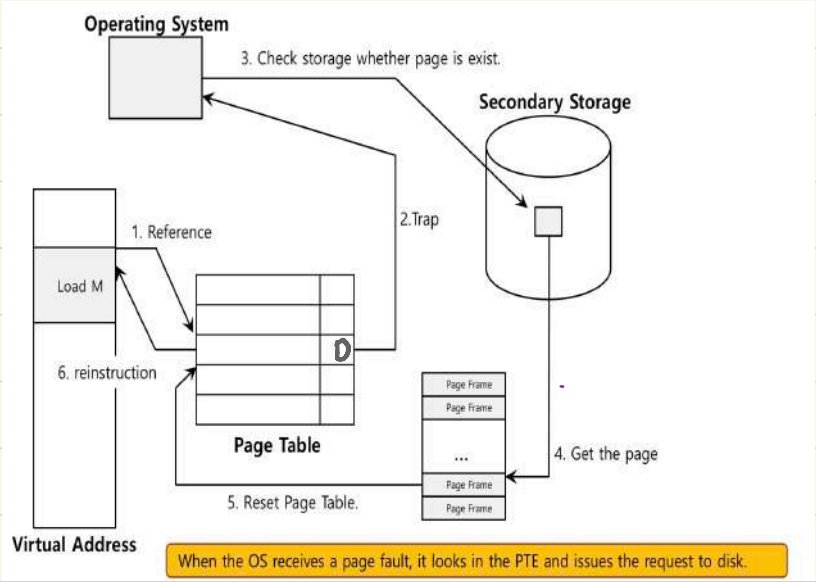
The page-fault

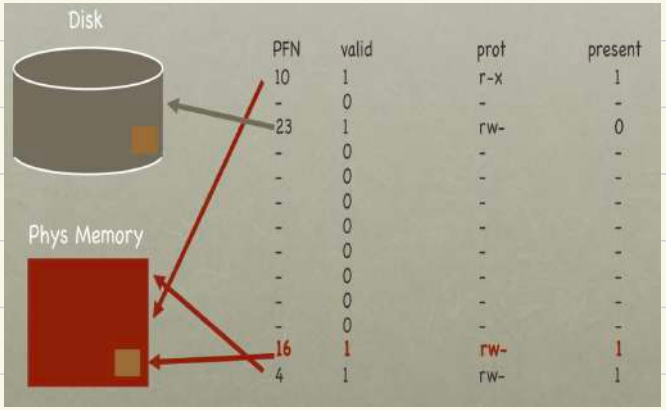
How does the OS knows where to find the page?

↳ use the bits reserved for PFN in PTE for "disk address".

i.e., if page-fault,

* OS issues the request to disk to fetch the page into memory (trap)
when the disk I/O completes





* OS updates the PFN in PTE, sets the present bit to 1

* retry the instruction → would result in a TLB miss, update the TLB, find the translation.

Note: Disk I/O is slow (micro seconds for memory vs milliseconds for disk)
If the CPU just sat idle waiting for the disk, it is a wastage of resources.

Therefore,

- the OS blocks the faulting process and puts it into the blocked queue.
- the scheduler picks another ready process and runs it on the CPU
- when the I/O completes, the faulting process becomes ready again.

But, what if the memory is already full?

↳ need to remove some page in order to bring the new page

page-out page-in

For this, we need some page-replacement policy.

Beyond Physical Memory: Policies

Main memory → holds a subset of pages

↳ works like a "cache"

Goal: pick a replacement policy that

minimize # times that we have to fetch a page from disk.

↳ cache miss.

maximize # times a page that is accessed is found in memory

↳ cache hit.

Performance measure: Average memory access time (AMAT) for a program,

$$AMAT = T_M + P_{miss} \cdot T_D$$

- T_M: time to access memory
- P_{miss}: prob of a cache miss
- T_D: time to access the disk.

eg:- suppose the hit rate is 90% $\Rightarrow P_{miss} = 0.1$.

let $T_D = 10$ ms (milli seconds) and $T_M = 100$ ns (nano seconds)

(1 ns $= 10^{-6}$ ms)

($1\ \mu s = 10^{-3}$ ms) micro second

then $AMAT = 100\text{ ns} + 0.1 * 10\text{ ms}$

$= 1.0001\text{ ms} \approx 1\text{ ms}$

hit rate → 99% $\Rightarrow AMAT = 100\text{ ns} + 0.01 * 10\text{ ms} = 0.1001\text{ ms} \approx 0.1\text{ms}$

hit rate → 99.9% $\Rightarrow AMAT = 100\text{ ns} + 0.001 * 10\text{ ms} = 0.1\ \mu s + 10\ \mu s = 10.1\ \mu s \approx 10\ \mu s$

roughly 100 times faster!

Goal: Develop a smart replacement policy that minimizes P_{miss}!

1. The optimal replacement policy : replace the page that will be accessed furthest in the future.

Let cache size = 3

6 hits 5 misses

$$\Rightarrow \text{Hit rate} = \frac{\text{Hits}}{\text{Hits} + \text{misses}}$$

$$= \frac{6}{6+5} = 54.5\%.$$

Advantages: Guaranteed to minimize number of page faults.

Disadvantages: Requires that OS predicts the future

$\Rightarrow$ not practical;

but as this policy is optimal, it can be used for comparison:!

↳ we will get to know, how close we are to "perfect"!

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

(2) A simple policy: FIFO (first-in, first-out)

Intuition: first referenced long time ago, so probably done with it now

Advantages: * fair; all pages receive equal residency
* easy to implement

Disadvantage: Some pages may always be needed.

Hit rate = $\frac{4}{11}$ = 36.4 %

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

(3) Another simple policy: Random (picks a random page to replace)

- it doesn't really try to be too intelligent in picking which blocks to evict.

- Random depends entirely upon how lucky it gets in its choice.

A common problem in FIFO and Random?
→ it might kick out an important page, one that is about to be referenced again.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

(4) Using History: LRU (replace page not used for longest time in past)

Intuition: use past to predict the future.

Hit rate = $\frac{6}{11}$ = 54.5%, which is almost close to optimal!

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Exercise

compare the performance of above policies in different workload examples.

(reference: 22.6, OSTEP)

Belady's Anomaly

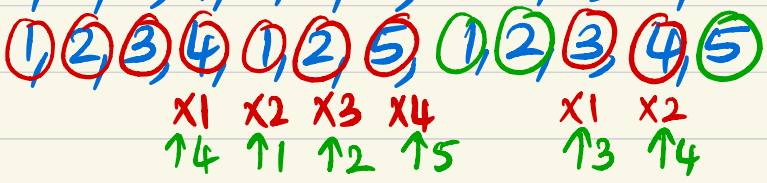
Question: whether cache hit rate increase when the cache gets larger?

Not necessarily in FIFO!

eg:- consider the memory reference stream,

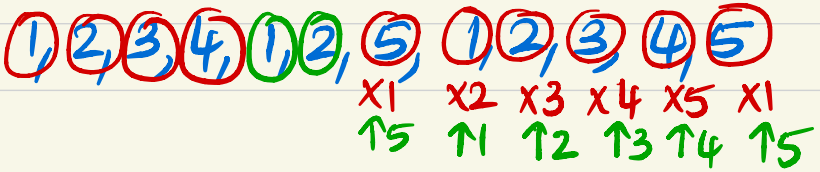
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

cache size: 3:



3 hits

cache size 4:



only 2 hits!

What about LRU?

LRU does not suffer from this problem, due to its stack property.

the set of pages in memory with N frames under LRU is always a subset of the set of pages in memory with N+1 frames.

Why this is true for LRU?

* Imagine you maintain a stack of all pages ordered by recency of use
  top = most recently used, bottom = least recently used
* If you allow N frames, you keep the top N pages from this stack.
* If you allow N+1 frames, you keep the top N+1 pages.

$\Rightarrow$ set of pages in memory with N frames $\subseteq$ set of pages in memory with (N+1)-frames.

(The above property is not true for FIFO)

Implementing historical policies

Note that implementing FIFO is easy → maintain a queue, need to update only when a page is evicted / a new page is added.
- page is evicted: remove the page in the first-in side
- a new page is added: add it to last-in side

How to implement LRU? → need to keep track of least recently used
⇒ need accounting work on every memory reference.
↳ How to do this accounting?

Use hardware support: when a page is accessed, add a time stamp to PTE, or maintain a list of frames with time stamp.
- when a page is accessed, the hardware can set the time field to the current time.
- when a page needs to be replaced, OS scan all the time fields in the system to find the least-recently used page.

Problem with this approach: suppose RAM size is 4GB and each page is of size 4KB, # pages $= \frac{2^{32}}{2^{12}} = 2^{20} = 1048576$
> 1 million pages!

⇒ finding LRU page is expensive!

Question: since the perfect LRU is expensive, can we approximate it in some way? YES!

need hardware support → use bit (or reference bit)

one bit per physical page → can be added in PTE or separate list

when a physical page is accessed → Hardware updates use bit to 1.
(hardware will never reset the use bit to 0, but OS does, using the following clock algorithm).

* all frames are arranged in a circle (like hours on a clock)
* a hand (pointer) moves around the clock to select victims

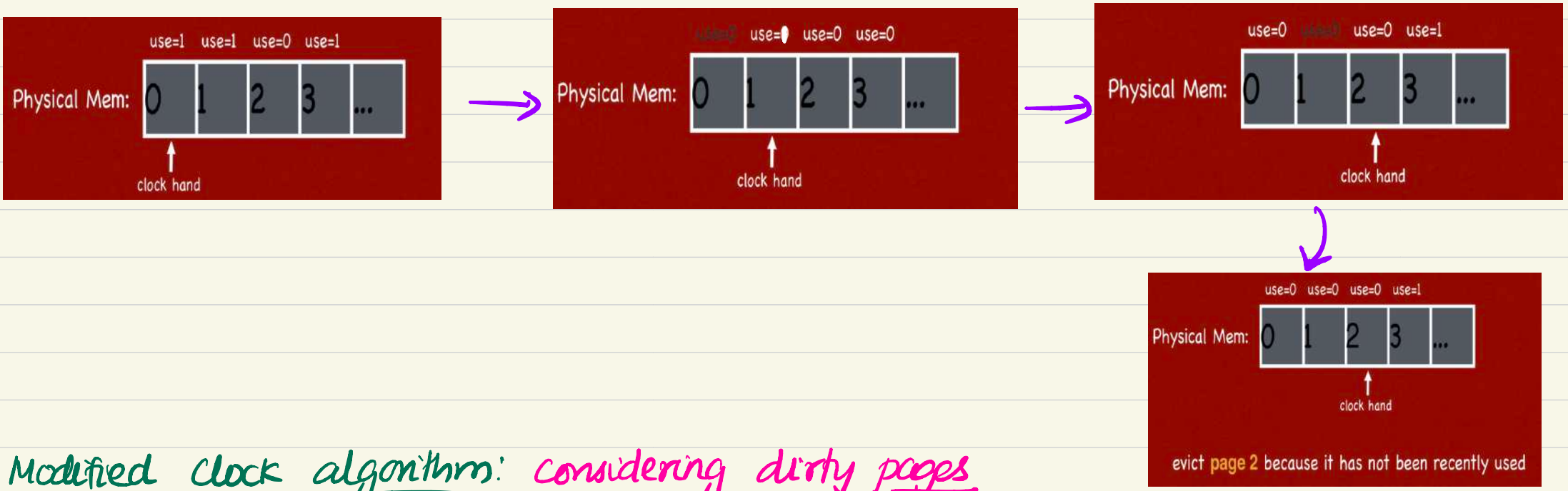
Steps: when a replacement is needed.

victims: pages to be evicted.

1. check the frame at the clock hand:
   - if use = 1, clear it to 0, move hand forward (giving a second chance to this page as it is recently used)
   - if use = 0, choose this frame for replacement.

2. continue until a victim is found.

Example: clock: looking for a victim page.



Modified clock algorithm: considering dirty pages

Idea: dirty = 1 ⇒ page has been modified → need to be written back to disk while evicting, which is expensive.

O/w (dirty = 0) ⇒ eviction is free.

∴ Some systems prefer to evict clean pages over dirty pages.

↳ How to do this? incorporate dirty bit in the replacement algorithm.

ii, Consider <use bit, dirty bit> as an ordered pair.

① <0,0> – not recently used and clean – best page to replace
② <0,1> – not recently used, but modified (has to be written to disk)
③ <1,0> – recently used, but clean
④ <1,1> – recently used and modified.

first page in lowest non-empty classes is selected as victim.

Other policies ① when to bring a page into memory?

- Demand paging: a page is brought only when it is needed.
- Prefetching: OS could guess that a page is about to be used and bring it in ahead of time.

eg:- A system may assume if a code page P is brought into memory, then code page P+1 will likely soon be accessed.

② How the OS writes pages out to disk?

one at a time

✓more effective!

clustering or grouping.

collect a number of pending writes together in memory and write them to disk in one write.

What should the OS do when thrashing happens?

when system spends more time handling page faults than actually executing useful instructions

ie, working set of active pages doesn't fit in physical memory. and page faults happen continuously!

OS can detect thrashing by noticing - high page fault rate
- low CPU utilization.

How to deal with it → earlier systems: admission control: decide not to run a subset of processes.

(better to do less work well than to try to do everything poorly)

→ Some versions of Linux: run an out of memory killer chooses a memory intensive process and kills it.

Refer chapters 21 and 22, OSTEP book.