Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 12: Multilevel paging and eviction

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Midterm.
2. Paging.
3. Multilevel paging.
4. Translation lookaside buffer.
5. Eviction.

Agenda

1. Midterm.
2. Paging.
3. Multilevel paging.
4. Translation lookaside buffer.
5. Eviction.

Midterm exam

Online using *Crabster.org*
Wed Jun 24 3:00 to 5:00 pm EDT.

If you need an accommodation, please let us know soon.

Material for midterm:

1. All the lecture topics from start until end of lecture 9 on deadlock.
2. All the labs on these topics.
3. Projects 1 and 2.

Midterm exam

Two sample exams posted on web page.

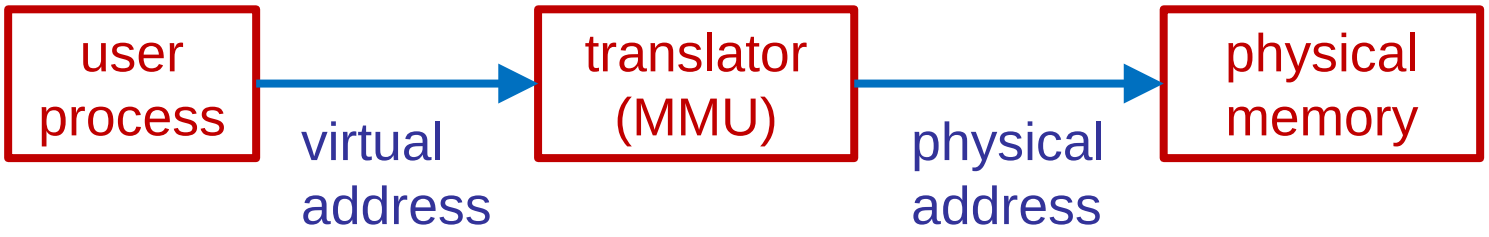
Solutions in lab section this Friday.

Review session Sat Jun 20 12:00 noon to 3:00 pm EDT.

Agenda

1. Midterm.
2. Paging.
3. Multilevel paging.
4. Translation lookaside buffer.
5. Eviction.

Dynamic address translation



Break the requirement that the process space be contiguous.

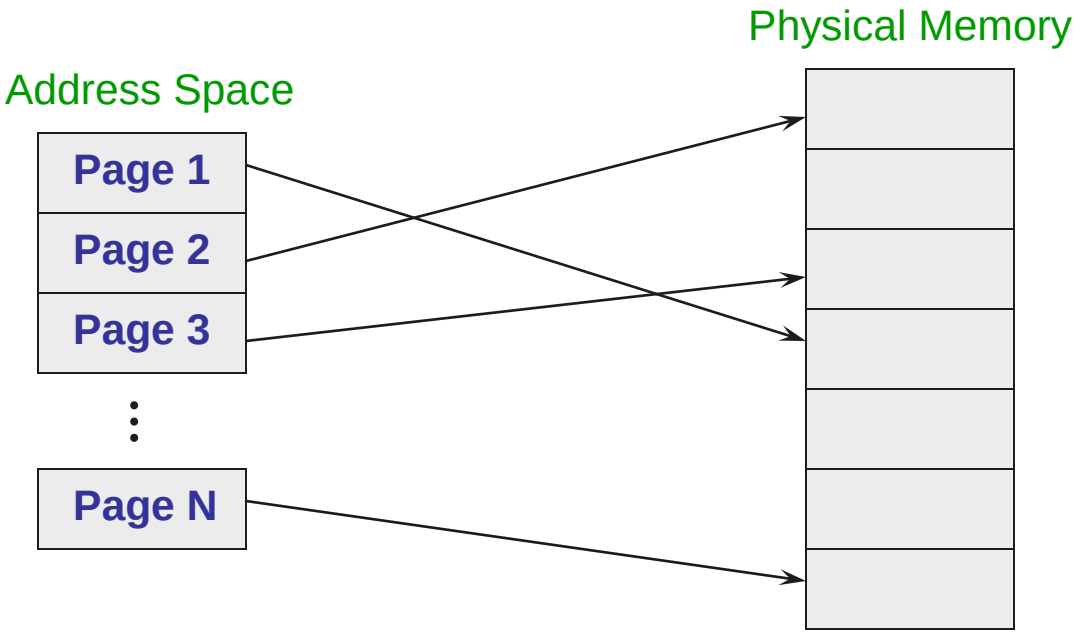
MMU strategies we'll discuss:

1. Base and bounds.
2. Segmentation.
3. Paging.

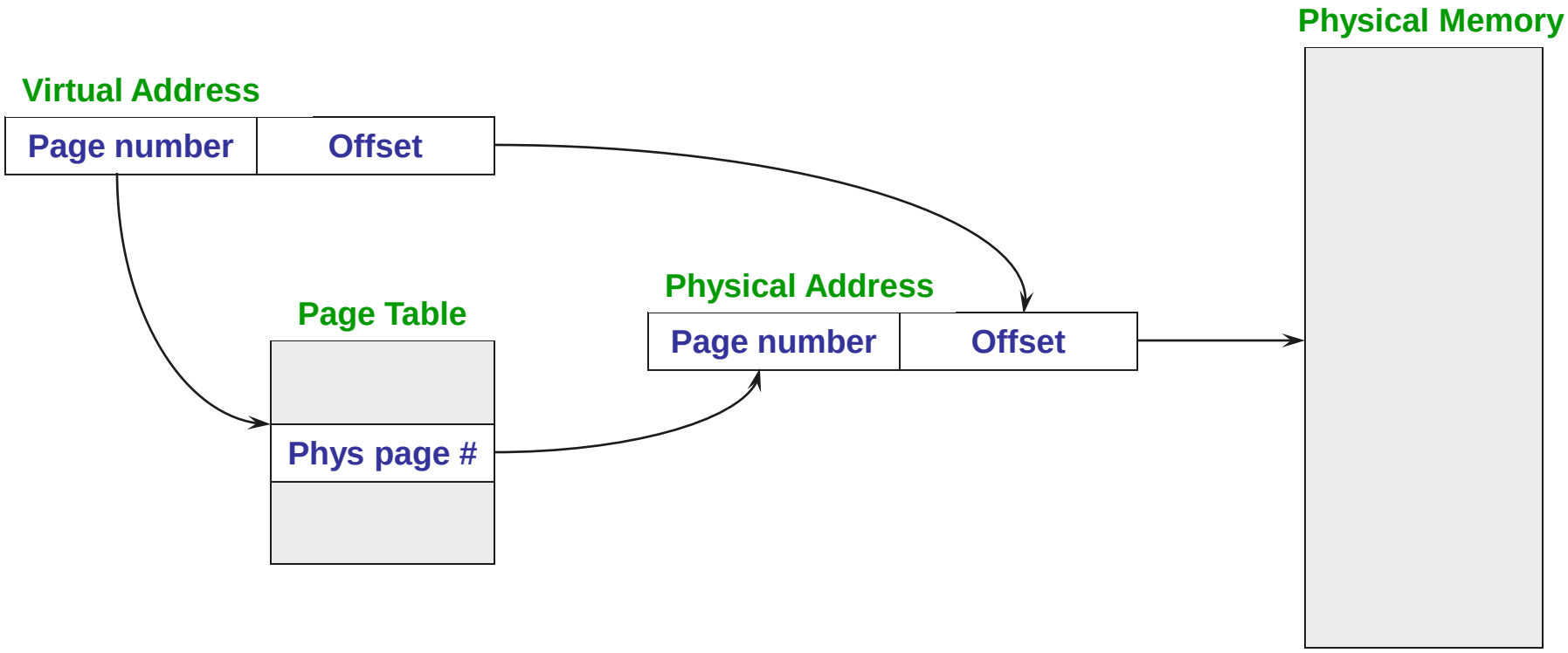
Paging

Allocate phys. memory in fixed-size units (pages)

Any free physical page can store any virtual page



Page Lookups



Paging

Each virtual page can be in physical memory or “paged out” to disk.

Pages can also have different protections (e.g., read, write, execute).

```
MMU_translation( )
   {
   if ( virtual page is
          invalid or non-resident or protected )
      trap to OS fault handler;
   else
      {
      physical page # =
         pageTable[ virtual page # ].physPageNum;
      physical address =
         concat( Physical page #, offset );
      }
   }
```

Paging

Virtual page #	Physical page #	Resident	Protection
0	105	0	RX
1	15	1	R
2	283	1	RW
3	invalid		
...	invalid		
1048575	invalid		

Valid → virtual page is legal for process to access.

Resident → virtual page is valid and in physical memory.

Error to access invalid page, but not to access non-resident page.

Paging

Virtual page #	Physical page #	Resident	Protection
0	105	0	RX
1	15	1	R
2	283	1	RW
3	invalid		
...	invalid		
1048575	invalid		

Causes of a page fault:

1. Invalid page or disallowed by protection bits, often a user bug.
2. Not resident and must be brought in by the OS.

Page table size

Page size is typically 4 KB or 8 KB.

Some architectures support multiple page sizes.

Each process with a 32-bit address space with 4-byte page table entries requires:

$$\left(\frac{2^{32}}{4096}\right) * 4 = 4\ MB$$

A 64-bit address space with 8-byte entries requires:

$$\left(\frac{2^{64}}{4096}\right) * 8 = 3.6 * 10^{16} = 36\ PB$$

Paging

Pros

1. Simple memory allocation.
2. Flexible sharing.
3. Easy to grow address space.

Cons

1. Large page table size.

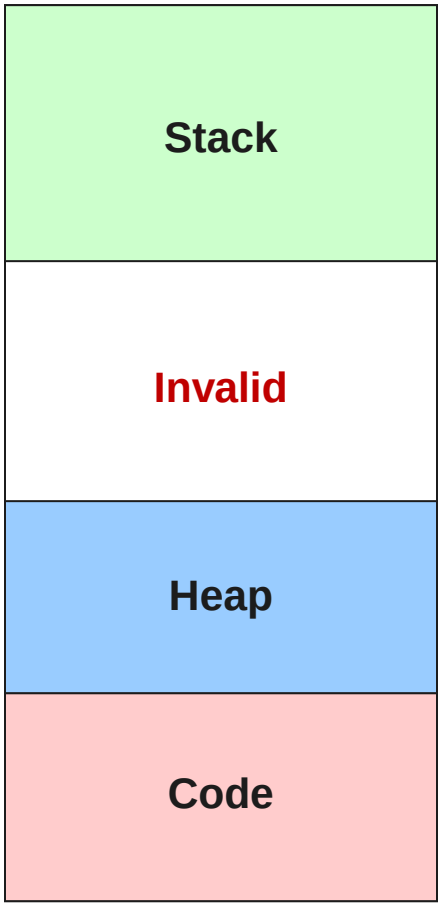
But the vast majority of all page table entries will be marked invalid.

Sparse Address Space

Most processes use only a tiny fraction of their 32 or 64-bit address space.

They usually have a huge hole in the middle.

So we only need to represent that part of the page table that isn't marked invalid.



Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048572	invalid
1048573	1078
1048574	48136
1048575	60

Agenda

1. Midterm.
2. Paging.
3. Multilevel paging.
4. Translation lookaside buffer.
5. Eviction.

Multi-level Paging

A standard page table is a simple array.

Multi-level paging generalizes this into a tree, filling in only the parts of the tree that aren't marked invalid.

With multilevel paging, a lot of the entries in any given page table will be null.

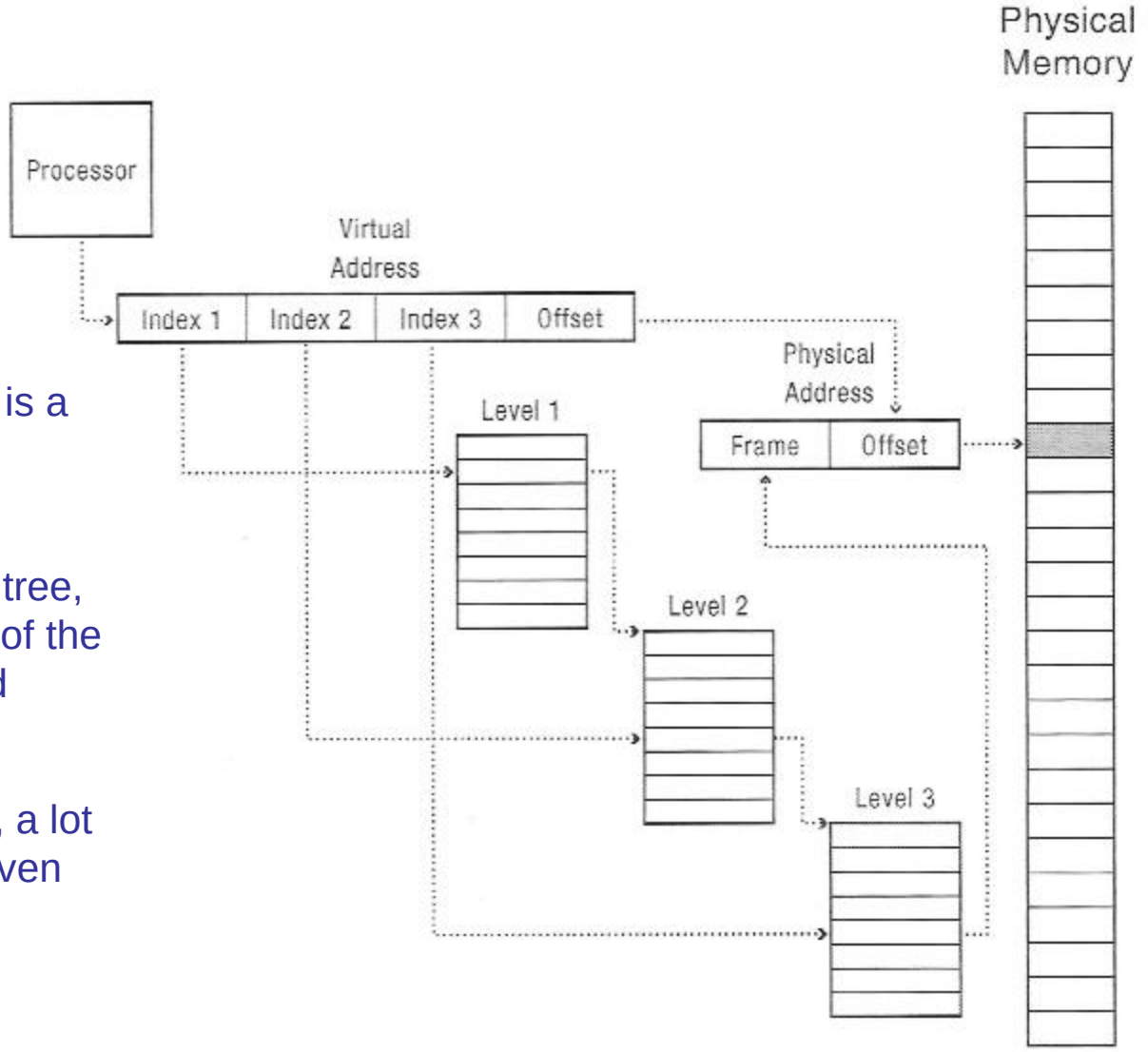


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

When a process starts, a new L1 page table is allocated, then filled in with L2 and L3 leaves as new pages are made valid.

When a process ends, the entire tree of L1, L2 and L3 tables is deleted.

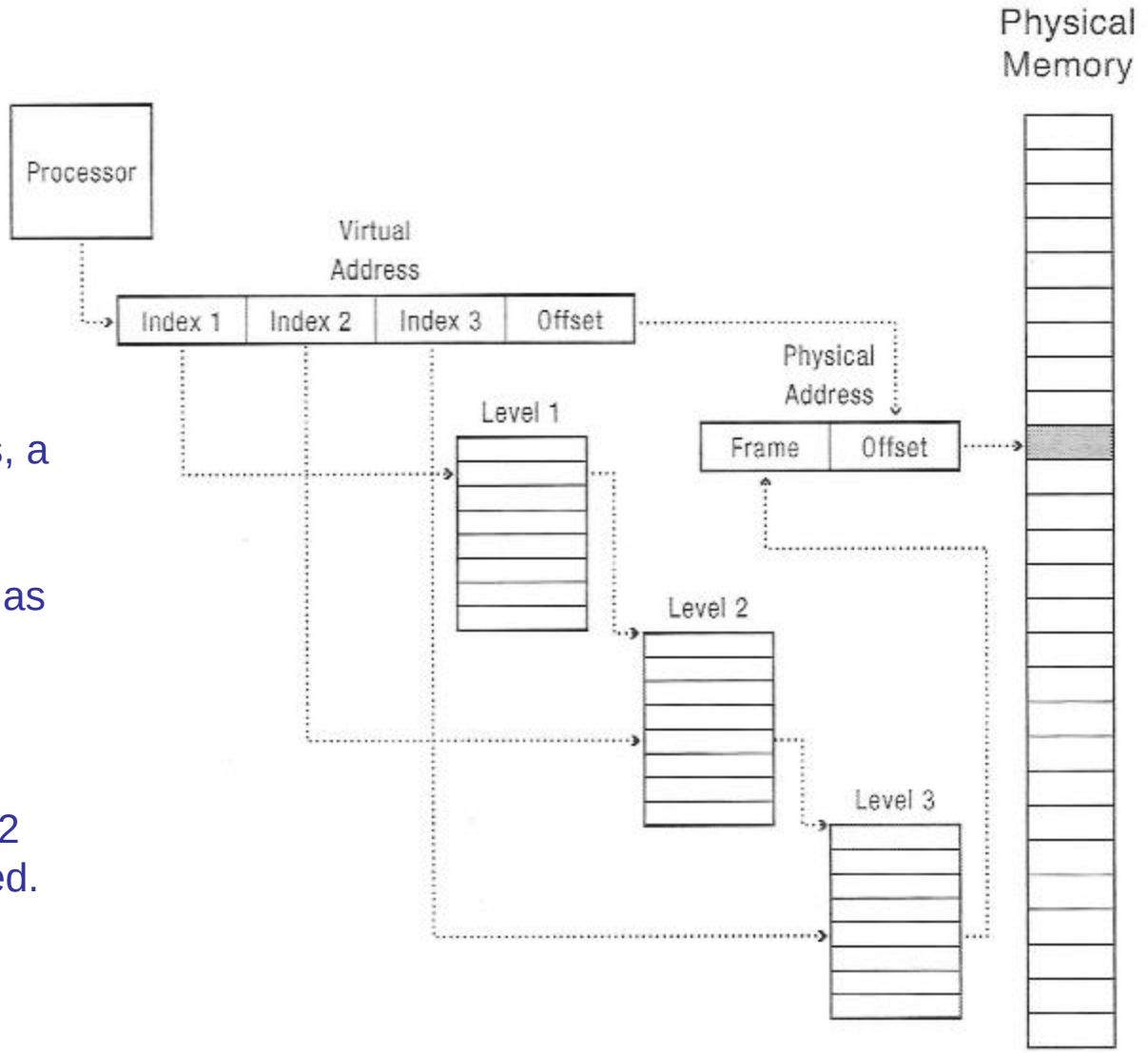
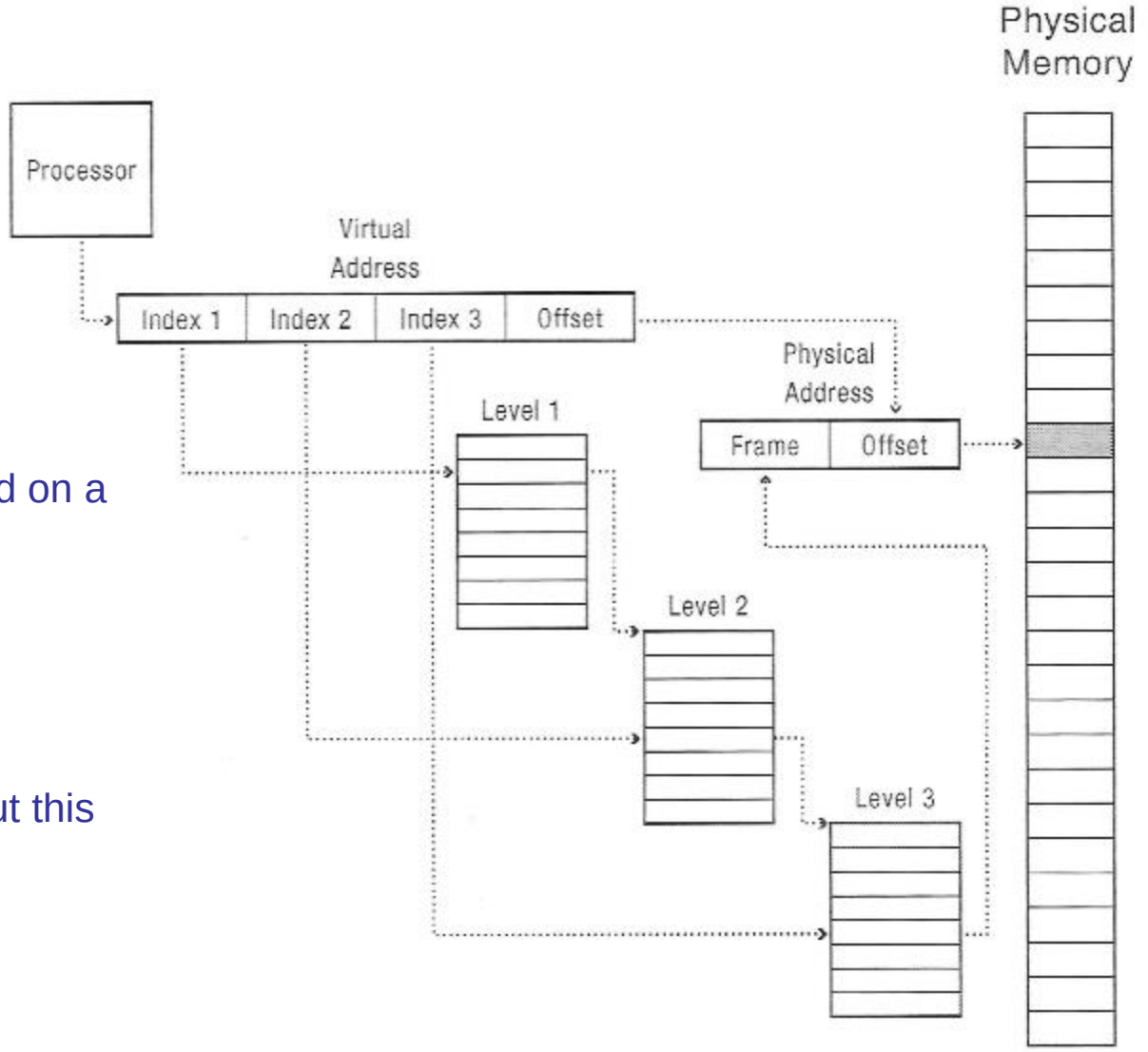


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging



Questions:

What must be changed on a context switch?

How would you share memory between processes?

What's not to like about this strategy?

Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging

What must be changed on a context switch?

Pointer to a level 1 page table.

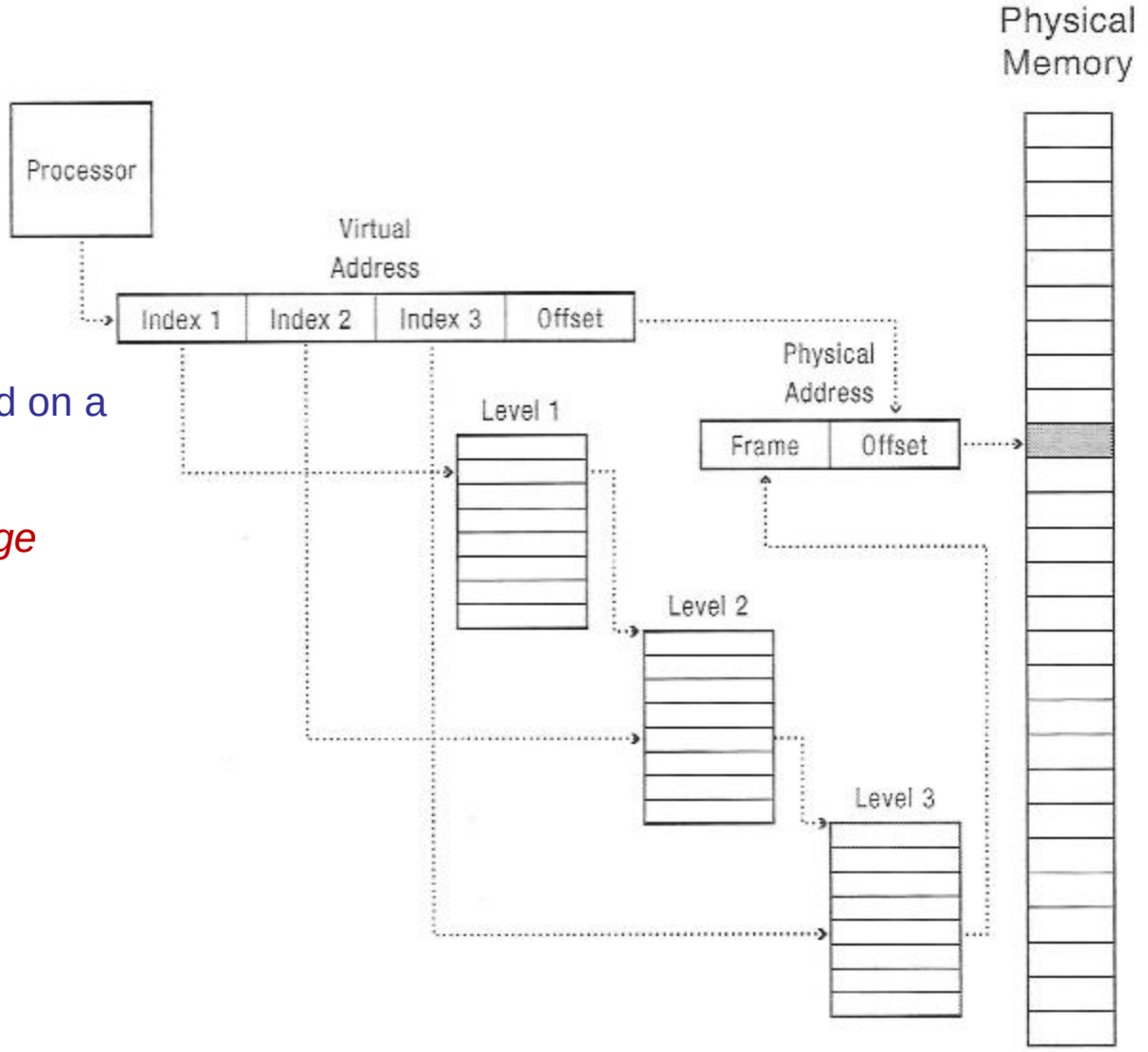


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging

How would you share memory between processes?

Share either individual pages or large blocks of pages by sharing a level 1, 2 or 3 entry.

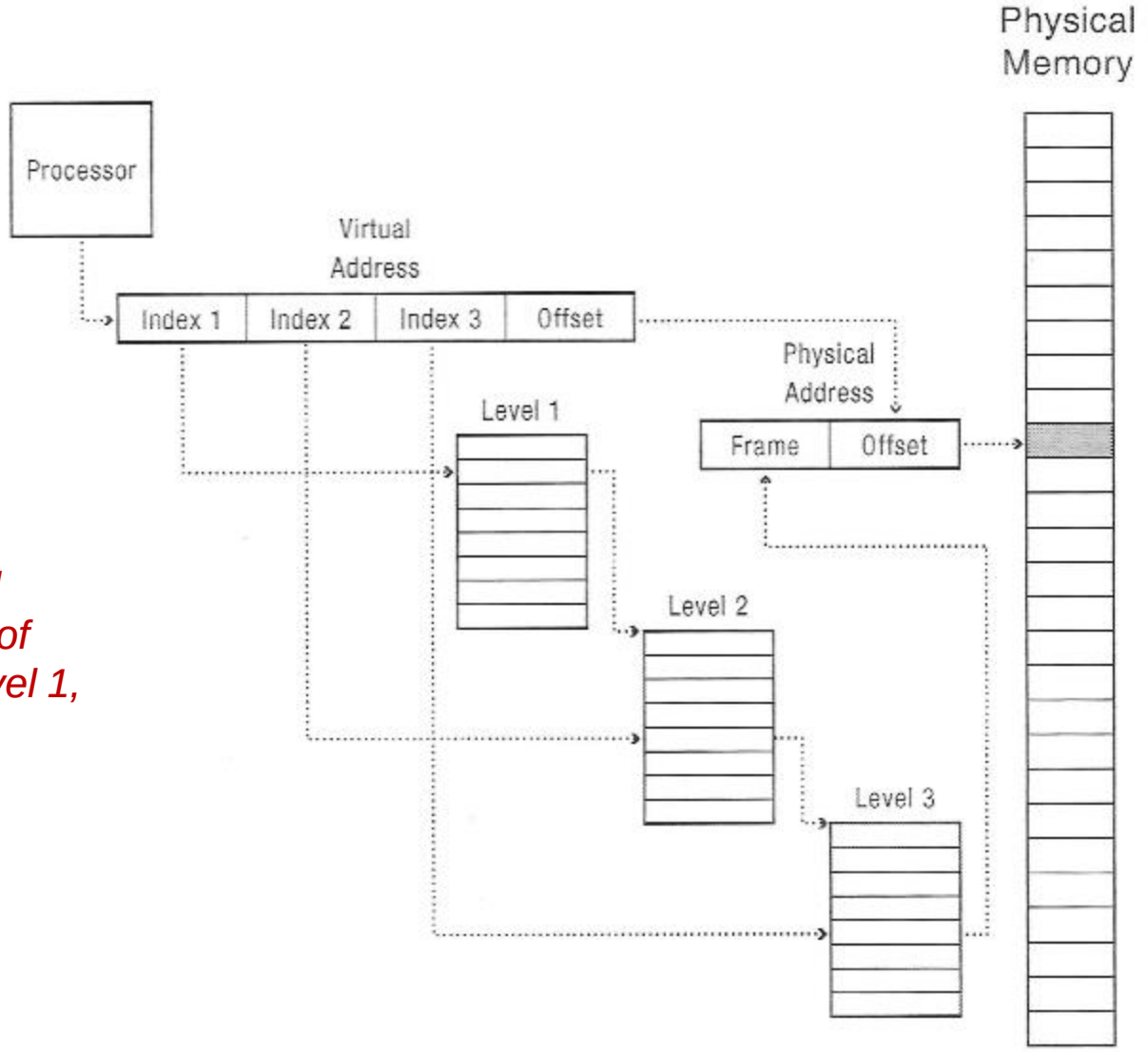


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multi-level Paging

What's not to like about this strategy?

Every memory access by a user application requires multiple table lookups.

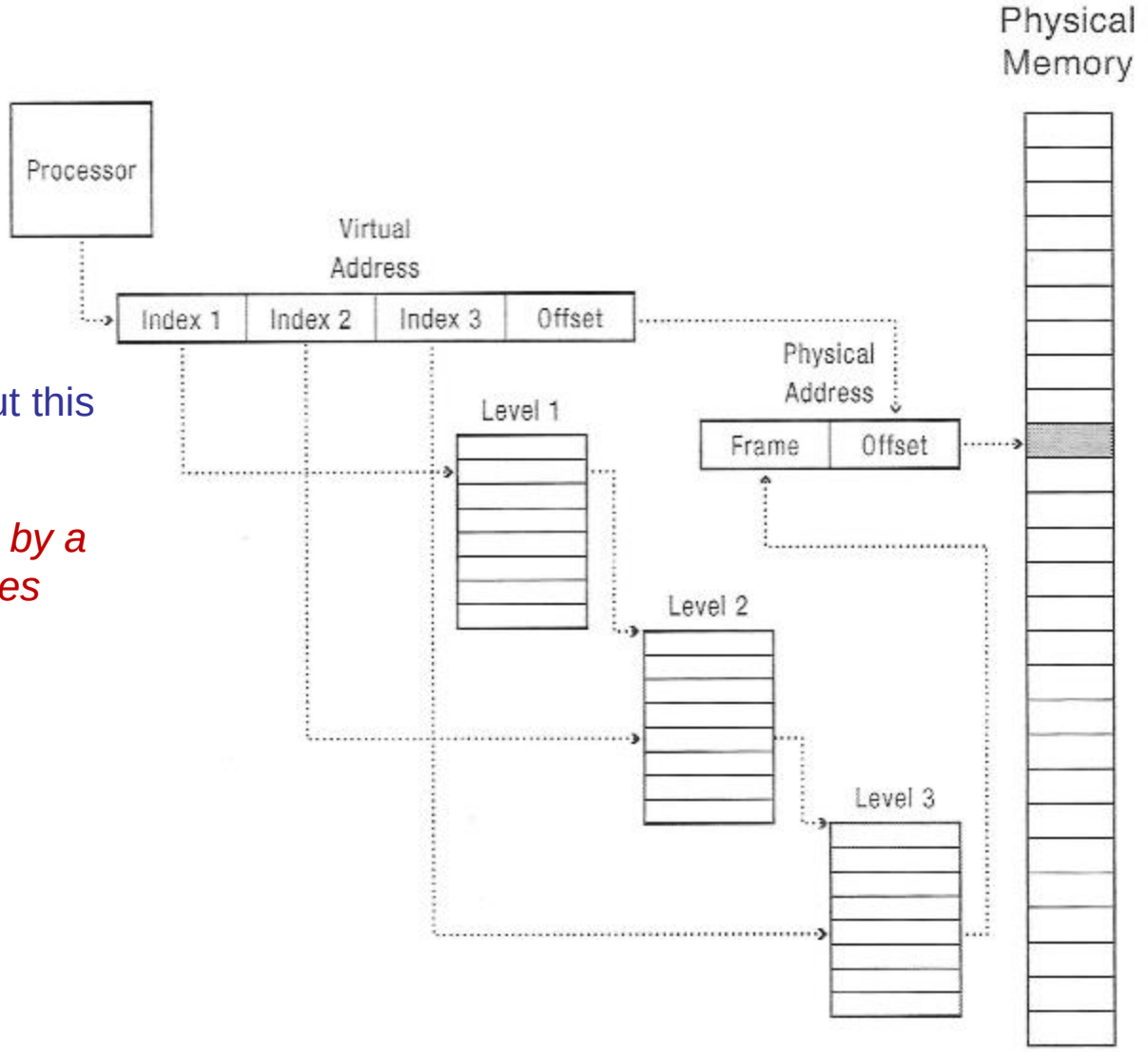


Image source: Anderson & Dahlin, *Operating Systems: Principles and Practice*, p. 398.

Multilevel paging

Pros

1. Simple memory allocation.
2. Flexible sharing.
3. Easy to grow address space.
4. Space-efficient representation of the page table.

Cons

1. Two or more extra lookups per memory reference.

What could be done to solve this?

We can cache the translations in hardware.

Agenda

1. Midterm.
2. Paging.
3. Multilevel paging.
4. Translation lookaside buffer.
5. Eviction.

TLB caches the virtual page number to PTE mapping.

A *cache hit* skips all the translation steps.

A *cache miss* requires searching the page table, updating the TLB, and restarting the instruction.

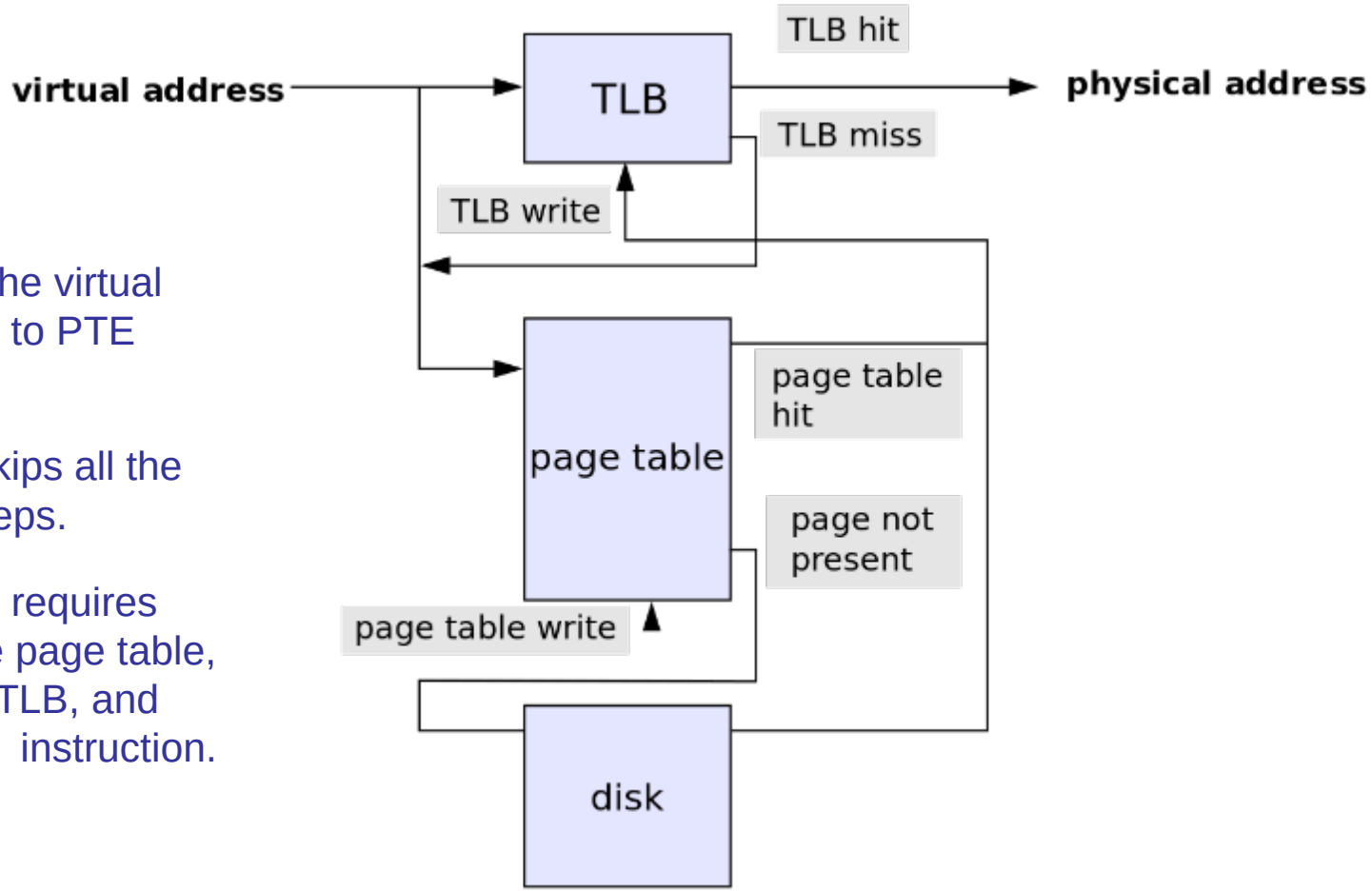


Image source: Wikipedia, *"Page table".*

TLB caches the virtual page number to PTE mapping.

A *cache hit* skips all the translation steps.

A *cache miss* requires searching the page table, updating the TLB, and restarting the instruction.

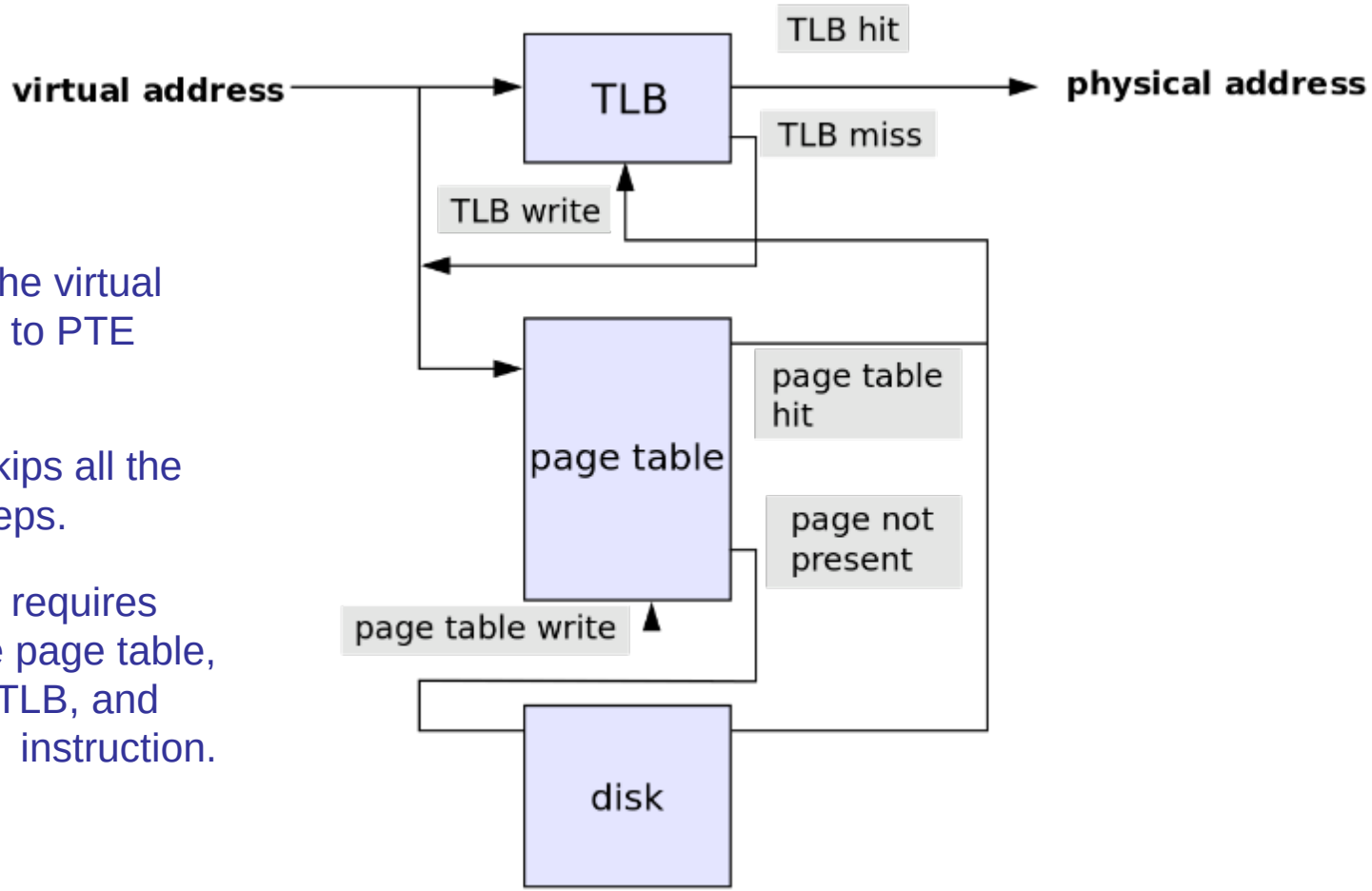


Image source: Wikipedia, *"Page table".*

The TLB is implemented in hardware as a content-addressable memory that acts like a map in C++.

Page table lookups are done in software in the operating system.

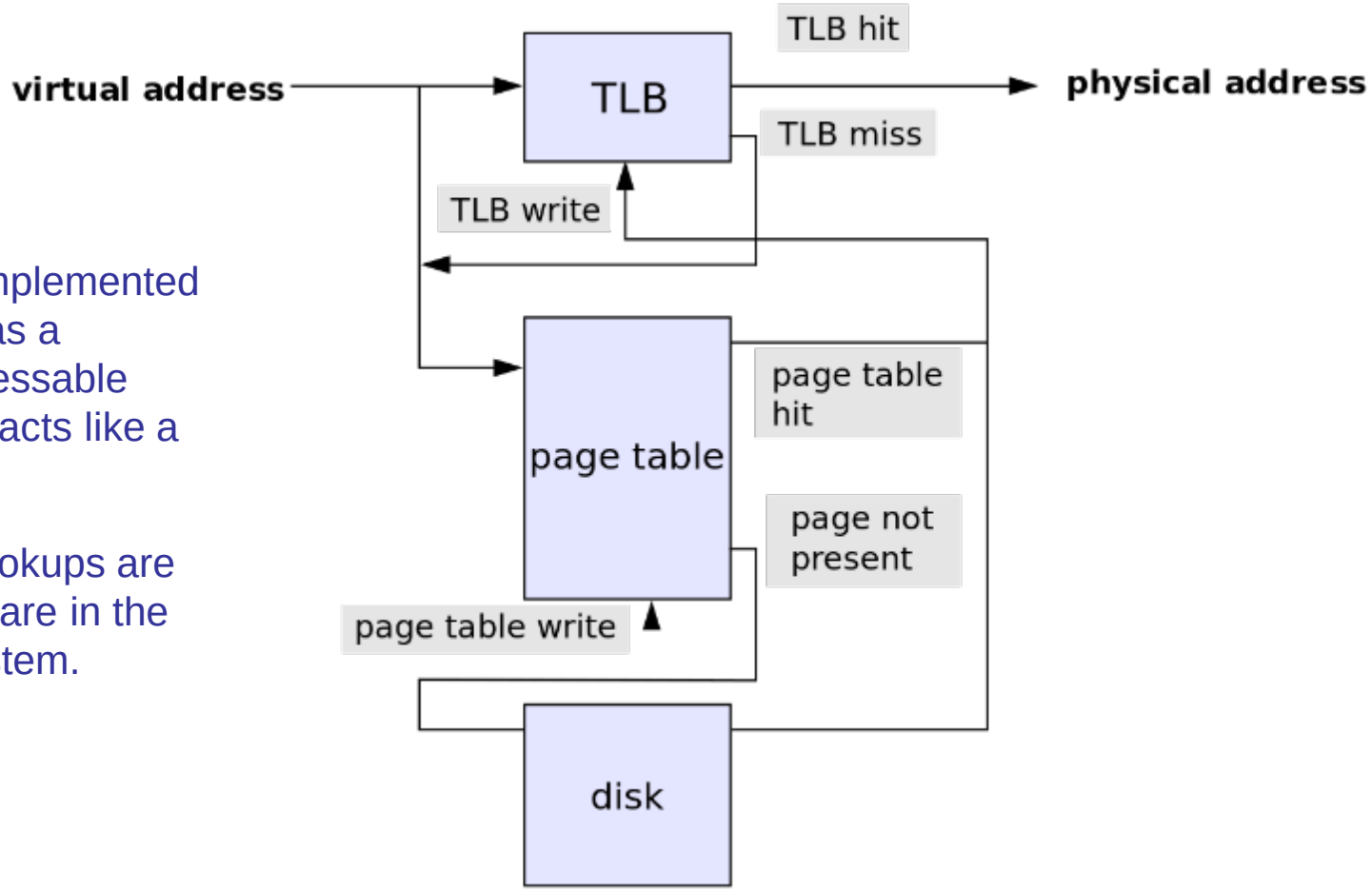


Image source: Wikipedia, *"Page table".*

What do you have to do in a context switch?

Reload the TLB.

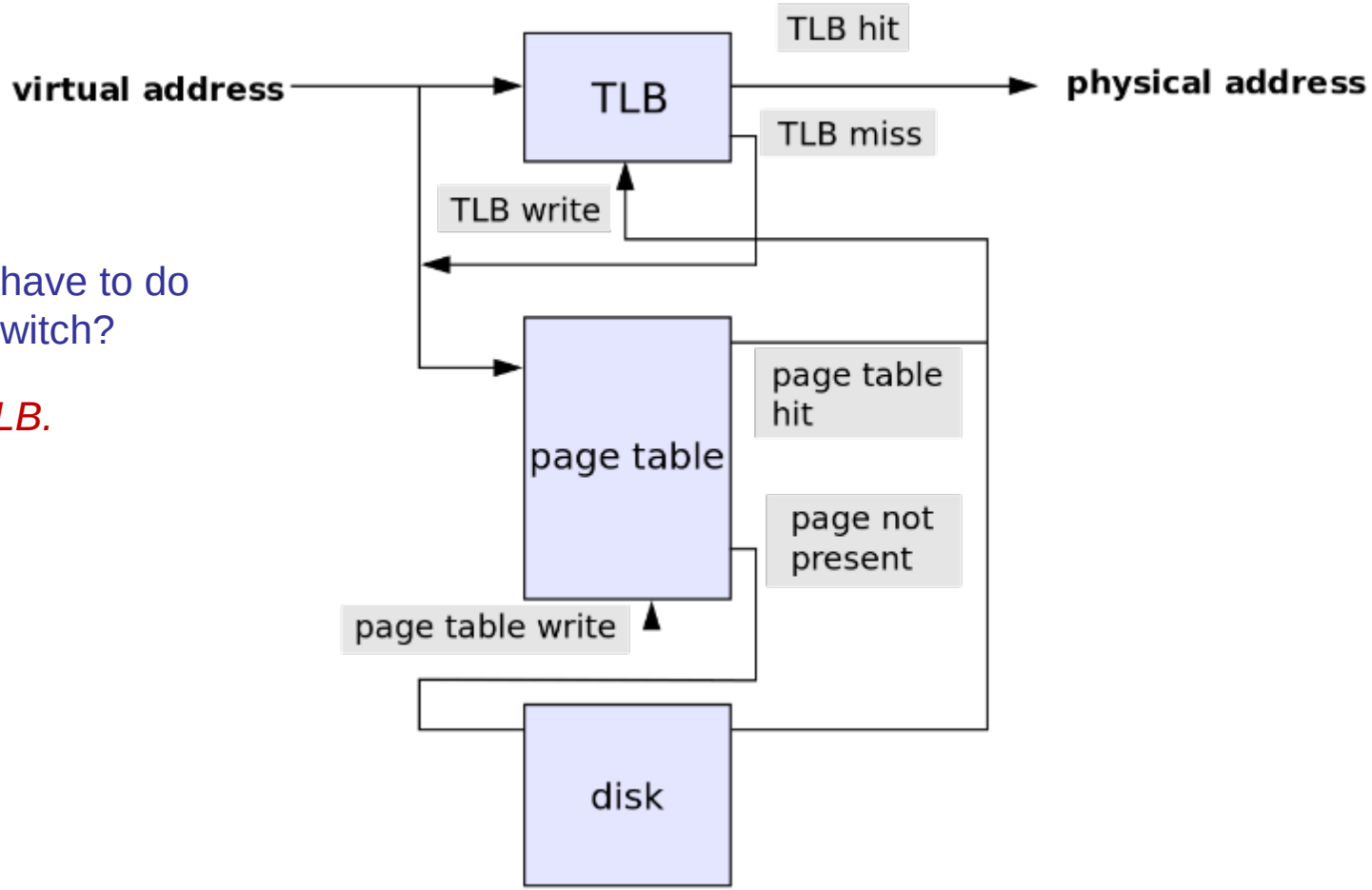


Image source: Wikipedia, *"Page table".*

Page replacement

Not at all valid pages can be in physical memory.

Must decide how to handle loads/stores to non-resident pages.

Sometimes, we will need to *evict* a page to make room for another.

Agenda

1. Midterm.
2. Paging.
3. Multilevel paging.
4. Translation lookaside buffer.
5. Eviction.

Page Replacement

Not all valid pages may fit in physical memory

Some pages must be paged out (written) to disk.

Disk is the “backing store”, physical mem acts as cache.

To read in a page from disk, some resident page may need to be paged out, *“evicted”,* first.

Need an algorithm to decide which page to evict to make space.

Goal: minimize page faults.

Replacement policies

Possibilities:

1. Random.
2. First in, first out (FIFO).
3. An imaginary optimum.
4. Least recently used (LRU).

FIFO

Under FIFO, we replace the oldest page brought into memory.

May replace pages that continue to be frequently used.

A	B	C	D	C	E	A	C	D	B	C
A	A	A	A	A	E	E	E	E	E	E
	B	B	B	B	B	A	A	A	A	A
		C	C	C	C	C	C	C	B	B
			D	D	D	D	D	D	D	C

Header row is sequence of accesses to 5 virtual pages A thru E that are paged in and out of 4 physical pages in memory, represented by the rows.

Each cell indicates what virtual page is in that physical page.

Optimal replacement

Under an optimal strategy, we would replace the page that won't be used again for the longest time, minimizing cache misses.

But that requires knowledge of the future!

A	B	C	D	C	E	A	C	D	B	C
A	A	A	A	A	A	A	A	A	A	A
	B	B	B	B	E	E	E	E	B	B
		C	C	C	C	C	C	C	C	C
			D	D	D	D	D	D	D	D

Header row is sequence of accesses to 5 virtual pages A thru E that are paged in and out of 4 physical pages in memory, represented by the rows.

Each cell indicates what virtual page is in that physical page.

LRU

Under LRU, we approximate the optimal solution by using past references. If a page hasn't been used for a while, we assume it probably won't be used again anytime soon.

A	B	C	D	C	E	A	C	D	B	C
A	A	A	A	A	A	A	A	A	A	A
	B	B	B	B	E	E	E	E	B	B
		C	C	C	C	C	C	C	C	C
			D	D	D	D	D	D	D	D

Header row is sequence of accesses to 5 virtual pages A thru E that are paged in and out of 4 physical pages in memory, represented by the rows.

Each cell indicates what virtual page is in that physical page.

Why would this work well?

LRU

Works well because of temporal locality: Recently accessed pages are often accessed again. Surprisingly good approximation of the optimum and hard to beat.

A	B	C	D	C	E	A	C	D	B	C
A	A	A	A	A	A	A	A	A	A	A
	B	B	B	B	E	E	E	E	B	B
		C	C	C	C	C	C	C	C	C
			D	D	D	D	D	D	D	D

Header row is sequence of accesses to 5 virtual pages A thru E that are paged in and out of 4 physical pages in memory, represented by the rows.

Each cell indicates what virtual page is in that physical page.

But could there be situations where doesn't work?

LRU

But depending on the access pattern, it can sometimes it can be very wrong!

A	B	C	D	E	A	B	C	D	E	A
A	A	A	A	E	E	E	E	D	D	D
	B	B	B	B	A	A	A	A	E	E
		C	C	C	C	B	B	B	B	A
			D	D	D	D	C	C	C	C

Header row is sequence of accesses to 5 virtual pages A thru E that are paged in and out of 4 physical pages in memory, represented by the rows.

Each cell indicates what virtual page is in that physical page.

Approximating LRU

Can’t afford to timestamp every access and maintain an actual queue, so we approximate with hardware support.

Most MMUs maintain a “referenced” bit for each resident page.

- Set by MMU when page is read or written.
- Can be cleared by OS.

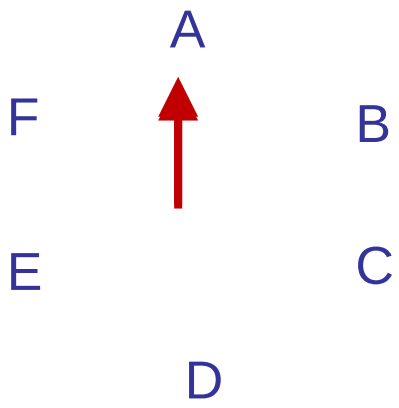
How to use reference bit to identify old pages?

- Clear reference bit for all pages.
- After some time, examine reference bits.
- Reference bit = 0 for pages not accessed recently.

Clock replacement algorithm

Arrange resident pages around a clock.

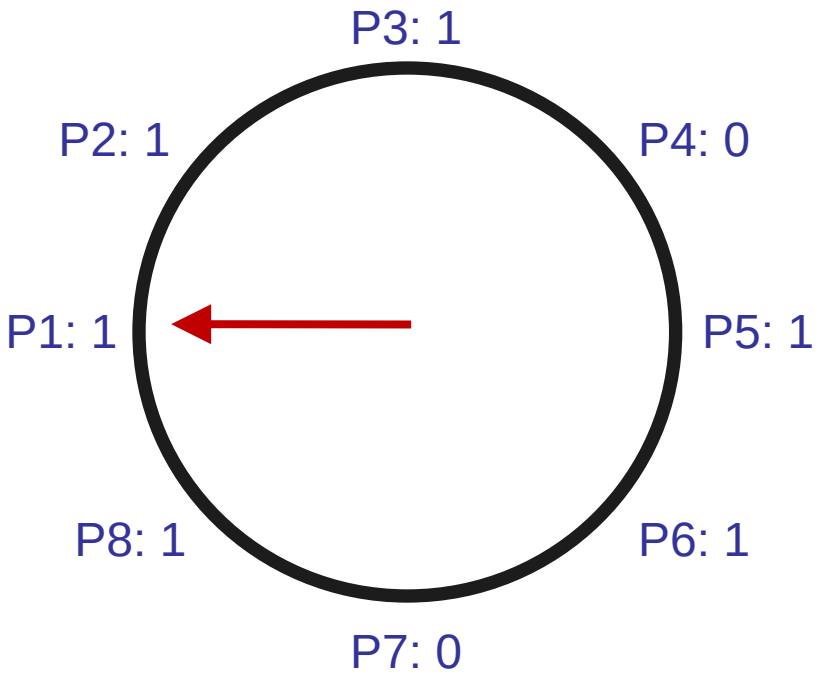
(You'll do this in project 3.)

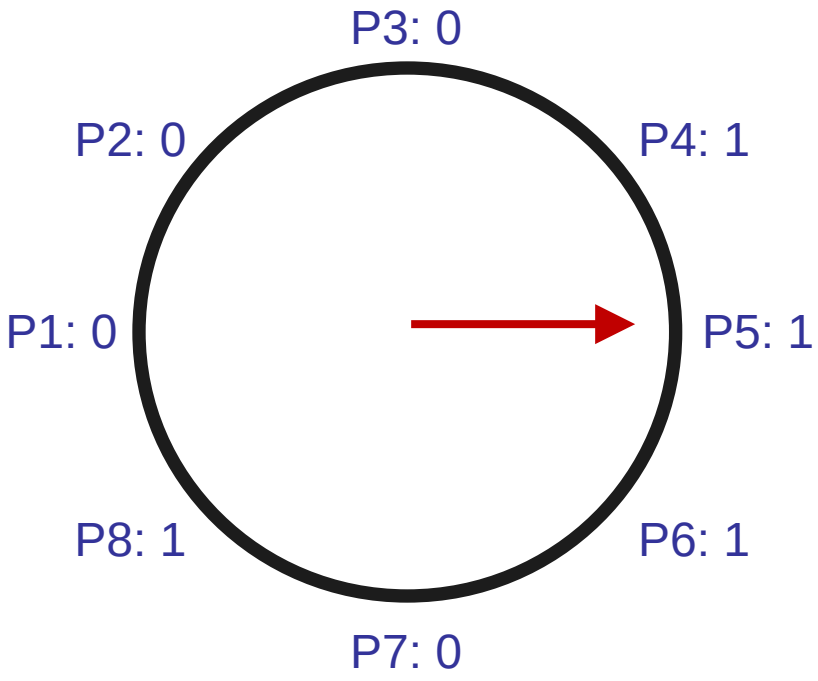


To select a page for eviction:

1. Consider page pointed to by clock hand.
2. If not referenced, evict page.
3. If referenced, clear reference bit.
4. Continue until a page is found that hasn't been referenced.

Clock example





What if all pages referenced since last sweep?

Page eviction

Some questions.

1. Where does the evicted page go?

   It will go to disk.

2. When do you not need to write page to disk?

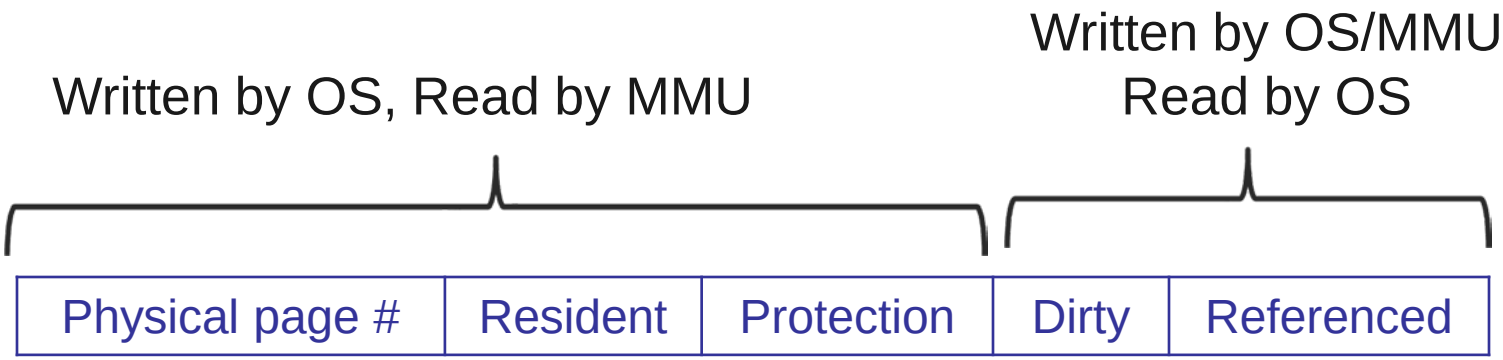
   When it's not been changed.
   Rely on hardware to maintain dirty bit in PTE.

3. Why not write to disk on every store?

   Would make every write as slow as the disk.
   Write to disk only when necessary.

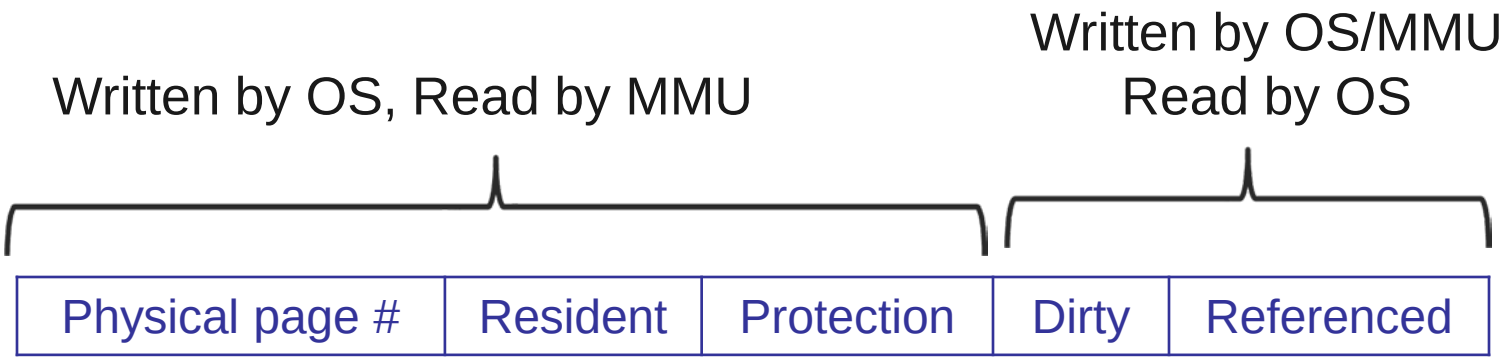
4. How can you optimize eviction?

Page table entry




```
MMU_translation( )
   {
   if ( virtual page is invalid or non-resident or protected )
      trap to OS fault handler;
   else
      {
      physical page # = pageTable[ virtual page # ].physPageNum;
      pageTable[ virtual page # ].referenced = true;
      if ( access is write )
         pageTable[ virtual page # ].dirty = true;
      physical address = concat( Physical page #, offset );
      }
   }
```

Page table entry



Why no valid bit in PTE?

All invalid virtual pages are non-resident.

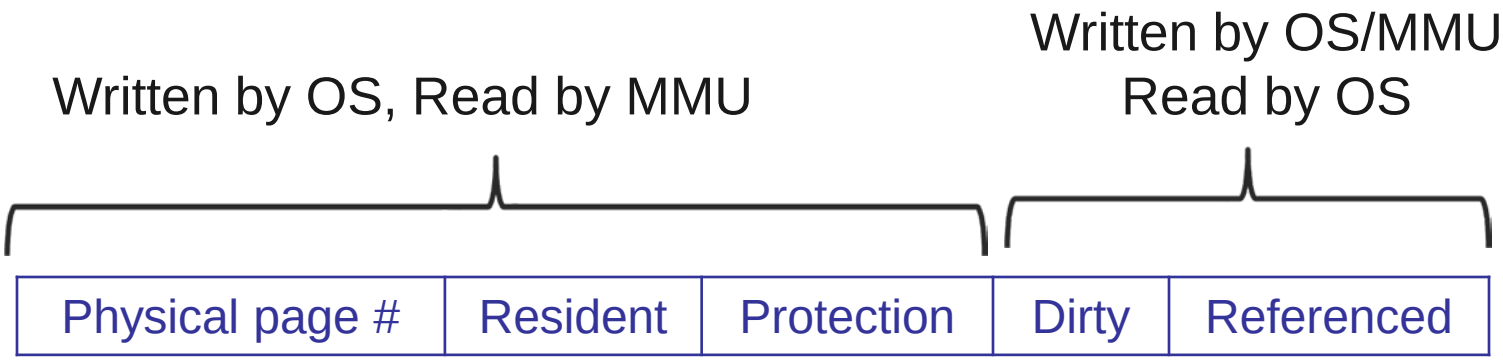
Valid non-resident pages: where's the disk block?

OS must maintain this, MMU simply traps to OS.

How can we make do without resident bit?

Clear protection bits when non-resident to cause hardware fault.

Page table entry



Can we make do without the dirty bit?

Have OS set the dirty bit itself.

Naive solution: Trap on every store & mark dirty.

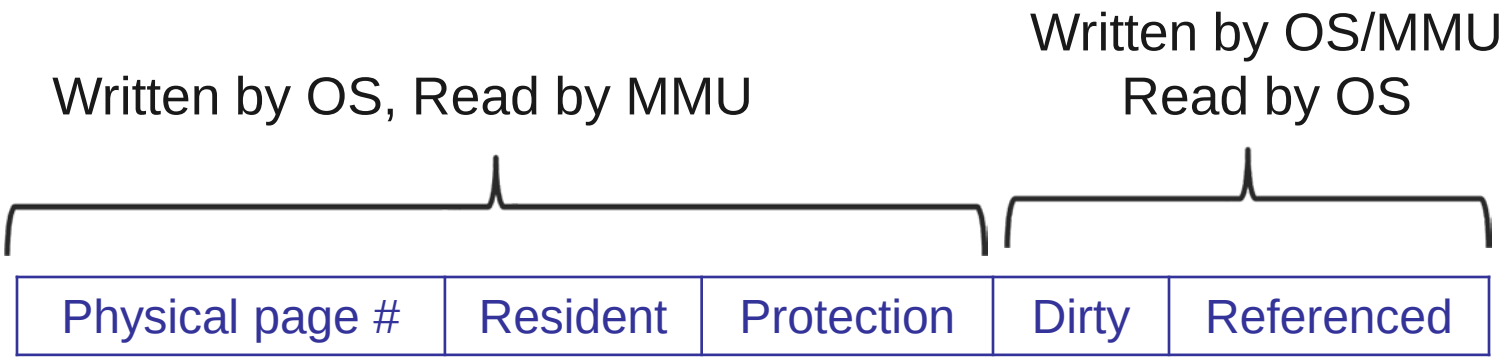
How to reduce # of page faults?

Only care about transition from clean to dirty.

Clean pages have 0 write protection bit.

Dirty pages have usual value.

Page table entry



Can we make do without the dirty bit?

Have OS set the dirty bit itself.

Naive solution: Trap on every store & mark dirty.

How to reduce # of page faults?

Only care about transition from clean to dirty.

Clean pages have 0 write protection bit.

Dirty pages have usual value.

Page table contents

Physical page #	Protection	Referenced

Can we make do without referenced bit?

Can use similar trick as that used for dirty bit

Insight: only care about unreferenced → referenced

MMU simpler but page fault handler more complex