CS 330 - Operating Systems

Paging

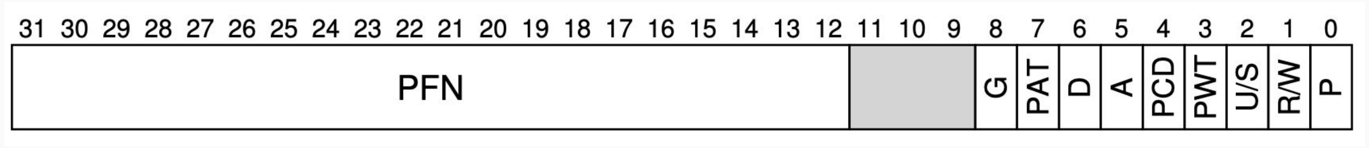
01-09-2025

Page Tables

- Stores virtual-to-physical address translations
- One page table per process
- Where are page tables stored?

Page Tables

- Stores virtual-to-physical address translations
- One page table per process
- Where are page tables stored?
  - Can get large
- Page table entry



Slowness of Paging

```
movl 21, %eax
```

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))

offset    = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

Summary of Paging

- Manage memory in fixed size units, or pages
- Each process has its own page table
    - Stored in physical memory
    - Hardware registers
        - pointer to page table start
        - page table length
- Cons
    - **Too slow**
    - Too much memory

Translation Lookaside Buffer (TLB)

- In memory management unit (MMU)
- Cache of certain VA to PA translations
  - Address translation cache

TLB Basic Algorithm

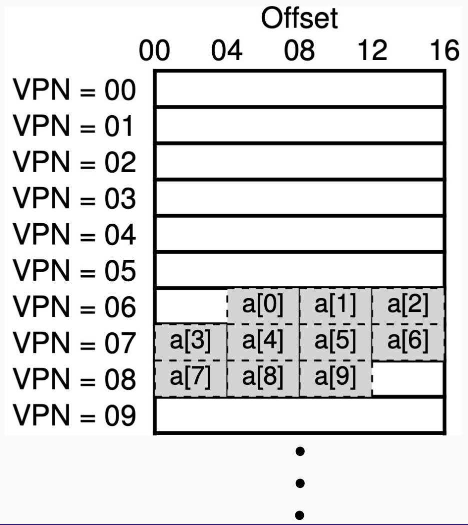
1. Extract VPN from VA
2. Check if translation for VPN is in TLB
3. If **TLB hit**
   a. Retrieve the translation entry
   b. Extract PFN from TLB entry
   c. Concatenate offset from VA to obtain PA
   d. (access checks should not fail)
4. If **TLB miss**
   a. Access the page table to retrieve translation (if valid)
   b. Update TLB with translation
   c. *Retry instruction (step 2)*

Example

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i]; }
```

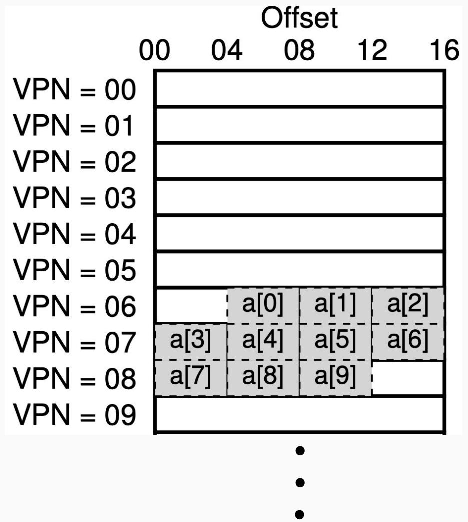
Example

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i]; }
```



Example

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i]; }
```



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
M	H	H	M	H	H	H	M	H	H

TLB Hit Rate = No. of hits/No. of accesses

Caching with TLB

- Spatial locality
  - Access to `a` means program may access `a+1`
  - Normally the next instruction is accessed
- Temporal locality
  - Re-access accessed data or instruction (loops)
- Why not big caches?

Memory Cycle Rate =

Hit rate * Hit cycles + Miss rate * (Miss cycles + Hit cycles)

TLB Contents

- Fully associative
  - Translation may be anywhere
  - TLB entry VPN | PFN | other bits
  - Other bits
    - Valid bit
    - Protection bit
    - Address-space identifier
    - Dirty bit
    - ...

Sharing TLB

Process 1

VPN	PFN
1	21
3	39

Sharing TLB

Process 1 and Process 2

VPN	PFN
1	21
3	39
1	49

Flushing TLB

Process 1 and Process 2

VPN	PFN	Valid bit
1	21	0
3	39	0
1	49	1

Address space identifiers

Process 1 and Process 2

VPN	PFN	Valid bit	ASID
1	21	1	1
3	39	1	1
1	49	1	2

TLB Entry Replacement

- Installing a new entry
  - Replace when no more space in TLB
- Cache replacement policies

Techniques So Far

- Base & Bounds
  - Pros: Very quick, 2 registers
  - Cons: Contiguous block of memory -> fragmentation ·
- Segmentation
  - Pros: Still relatively simple, 3 registers, lesser fragmentation
  - Cons: Still contiguous block of memory for segment
- Paging
  - Pros: Very low chances of segmentation
  - Cons: Slow, lots of memory accesses; overhead/process is huge!
- Paging + TLB
  - Pros: Improves address translation (spatial & temporal locality)
  - Cons: Limited in size, overhead/process still huge

Paging Memory Overheads

Assuming a

- 32-bit address space (2^{32} bytes),
- 4KB (2^{12} bytes) pages,
- and 4-byte PTE

what is the size of the page table?

Paging Memory Overheads

Assuming a

32-bit address space (2^{32} bytes),

4KB (2^{12} bytes) pages,

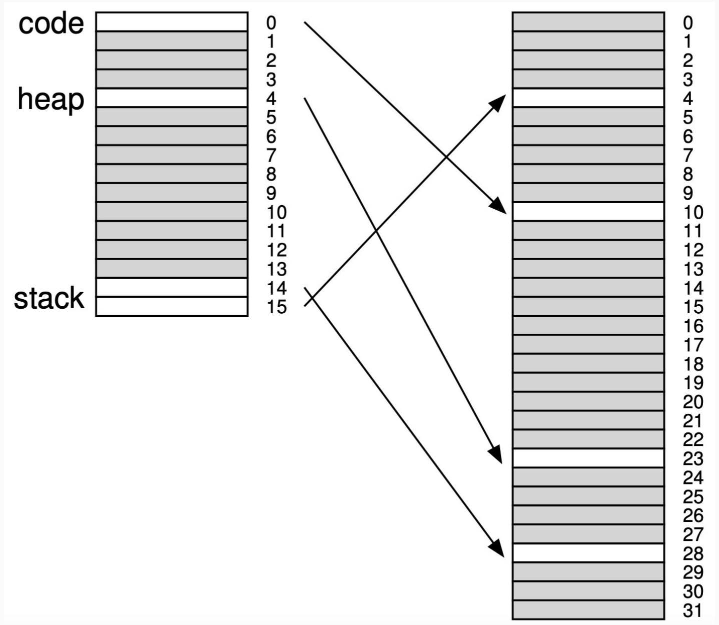
and 4-byte PTE

what is the size of the page table?

With 100 processes running simultaneously?

Combine Pages and Segments

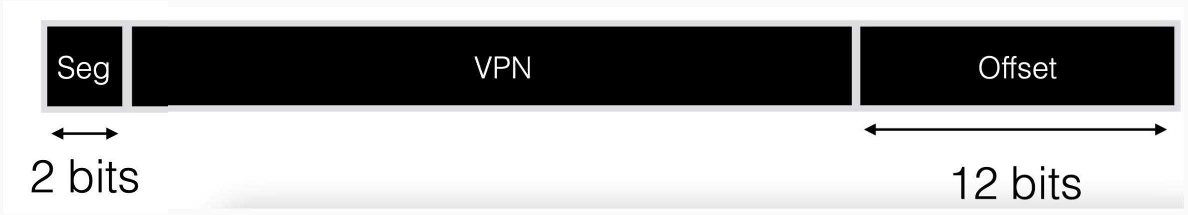
16 KB address spaces having 1 KB pages



PFN	Valid	...
10	1	..
-	0	
-	0	
-	0	
23	1	
-	0	
-	0	
-	0	
-	0	
-	0	
-	0	
-	0	
-	0	
-	0	
28	1	
4	1	

Combine Pages and Segments

- Per segment PT
- Different size
- Separate base & bounds per segment
- Base register
  - Stores PA of start of PT
- Bounds
  - Stores PA of end of PT (or number of valid entries in PT)



Paging with Segmentation

- Suppose we have the following virtual address design:

Segment No.	Page No.	Offset
2 bits	8 bits	8 bits

Virtual addresses are translated into 16-bit physical addresses of the following form:

Physical Frame No.	Offset
8 bits	8 bits

What is the page size?
What is the maximum size of the physical memory?

Paging with Segmentation

- Suppose we have the following virtual address design:

```
Segment No.       |       Page No.       |      Offset      |
-------------------------------------------------------------
  2 bits          |        6 bits        |      8 bits      |
```

0x4225?

Virtual addresses are translated into 16-bit physical addresses:

```
Physical Frame No.   |       Offset       |
--------------------------------------------
      8 bits         |       8 bits       |
```

Segment Table:

Segment	Base Addr	Limit
0x0	0x2100	0x1000
0x1	0x3500	0x2000
0x2	0x1500	0x0200

Page tables at:

0x2100	0x3500	0x1500
0x10	0x35	0x75
0x15	0x45	0x80
0x20	0x55	0x85
0x25	0x65	0x90

Combine Pages and Segments

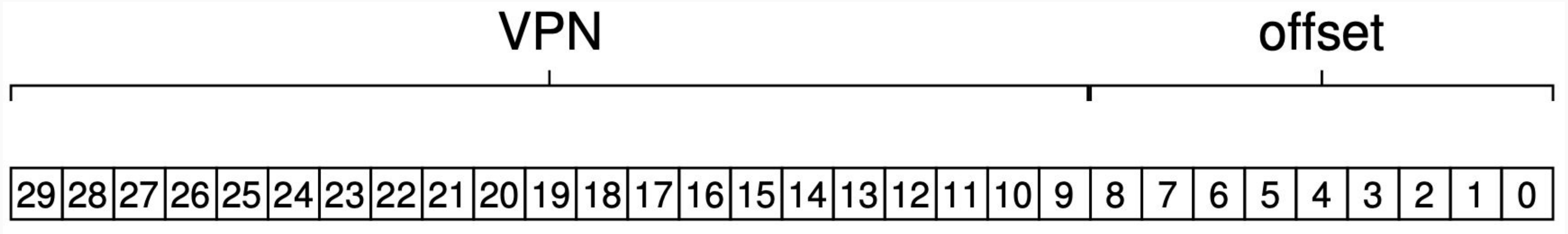
- Pros:
  - Leads to memory saving (Large gaps between segments)
- Cons:
  - Uses segmentation
    - Assumes certain usage pattern of address space
    - Sparsely used segments (sub-segment internal fragmentation) have same space waste issue
    - Variable size page tables can lead to fragmentation

Combine Pages and Pages

- Avoid segmentation with multi-level paging
- Linear page table → tree (hierarchical structure)
- Multi-level paging
    - Break PT into pages
    - Allocate page of PT if at least one entry in PT is valid
    - **Page directory**
        - Indicates where a page of the PT is (or) if invalid

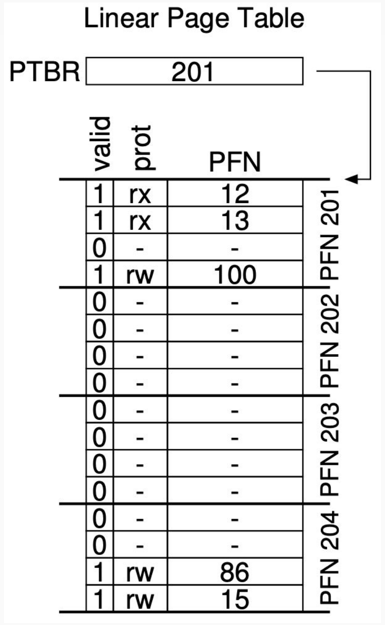
Multi-level Paging

- Suppose a 30-bit VA space and 512 byte page-size
  - VA has 21-bit VPN and 9-bit offset
  - With 4-bytes PTE, 128 PTEs forms one page (7 bits for PN)
  - Remaining 14 bits are for page directory
  - Spans 128 pages!



- TLB with multi-level paging
  - Multiple additional memory accesses to look up translation

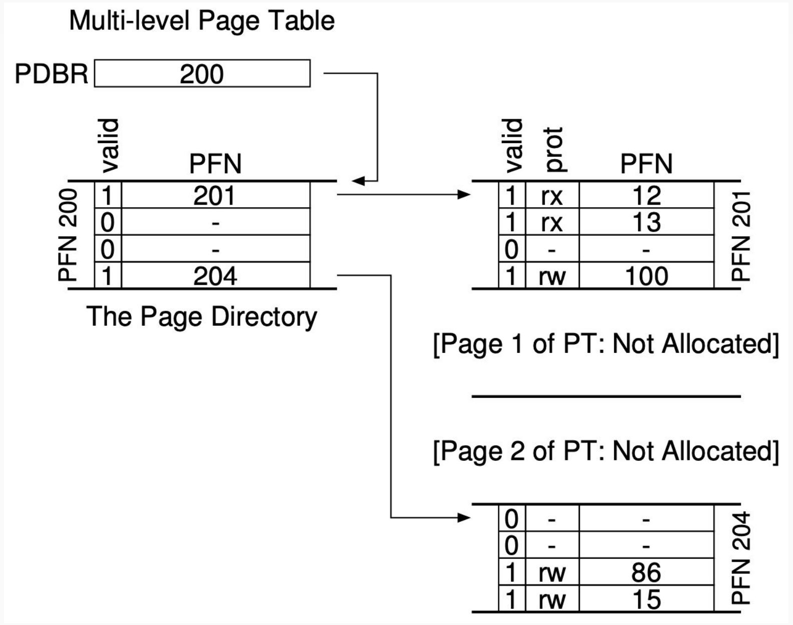
Multi-level Paging



Linear Page Table

PTBR 201

valid	prot	PFN	
1	rx	12	PFN 201
1	rx	13	
0	-	-	
1	rw	100	
0	-	-	PFN 202
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN 203
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN 204
0	-	-	
1	rw	86	
1	rw	15	



Multi-level Page Table

PDBR 200

	valid	PFN
PFN 200	1	201
	0	-
	0	-
	1	204

The Page Directory

valid	prot	PFN	
1	rx	12	PFN 201
1	rx	13	
0	-	-	
1	rw	100	

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]

valid	prot	PFN	
0	-	-	PFN 204
0	-	-	
1	rw	86	
1	rw	15	

Multi-level Paging (Example)

Address space	- 16 KB

What is the size of the virtual address in bits?

Page size	- 64 bytes
PTE	- 4 bytes

No. of bits for VPN and offset?

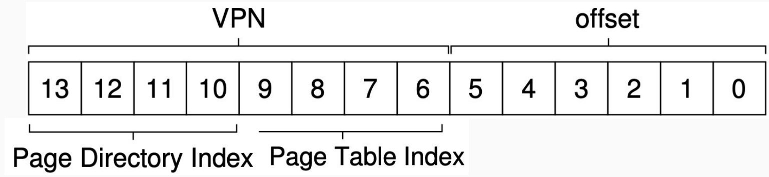
No. of entries in linear page table?

Size of the linear page table?

For 64-byte pages, with 1 KB table: 16 64-byte pages, i.e., each second level PT will hold 16 PTE

Multi-level Paging (Example)

Address space	- 16 KB (14-bit VA - 8 for VPN and 6 for offset)
Page size	- 64 bytes
PTE	- 4 bytes
Code	- pages 0 and 1
Heap	- pages 4 and 5
Stack	- pages 254 and 255



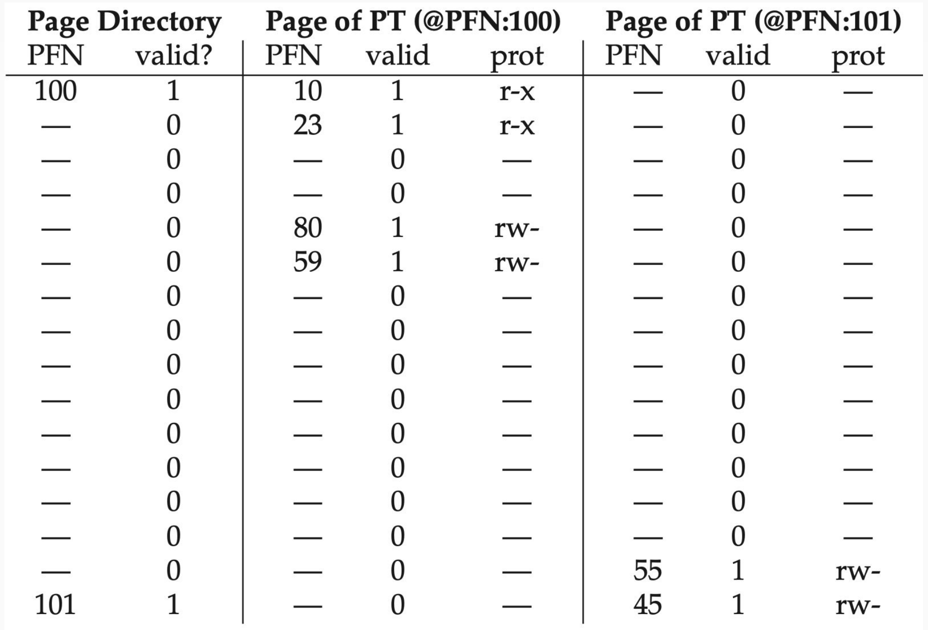
Multi-level Paging (Example)

- Assume that the page directory has two entries for two page tables stored at 100 and 101:

Code	- pages 0 and 1
Heap	- pages 4 and 5
Stack	- pages 254 and 255

0x3F80?

0x0030?



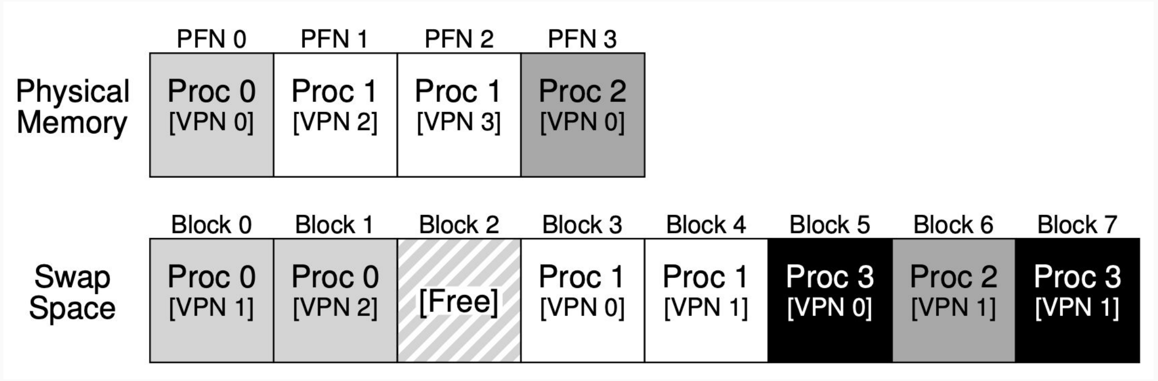
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Larger Virtual Memory

- Virtual memory may be larger than physical memory
- Stash address spaces from physical memory that are not in use
    - Normally, in a location that has more capacity
    - Hard disk
- Larger devices to provide illusion of larger virtual address space
    - Helps with multiprogramming
- Memory overlays in older systems

Demand Paging

- Disk space to move pages back and forth
  - Swap space
  - Remember disk address of a page



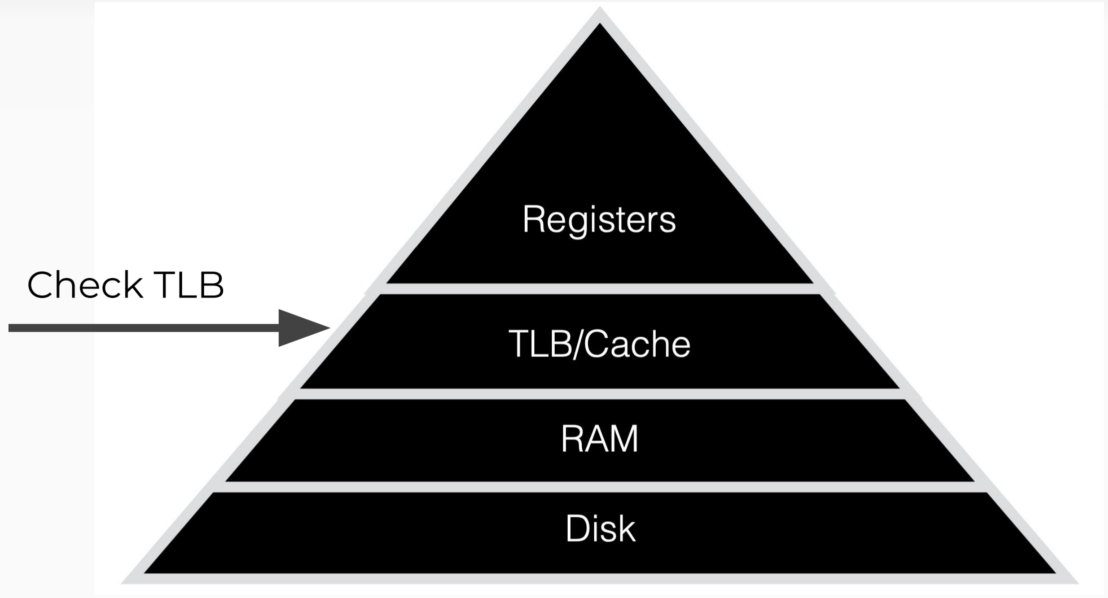
Cache Management

- Main memory caches some/all pages of a process
- Minimize number of misses when replacing pages
- Given the number of hits and misses, the **average memory access time** for a program is:

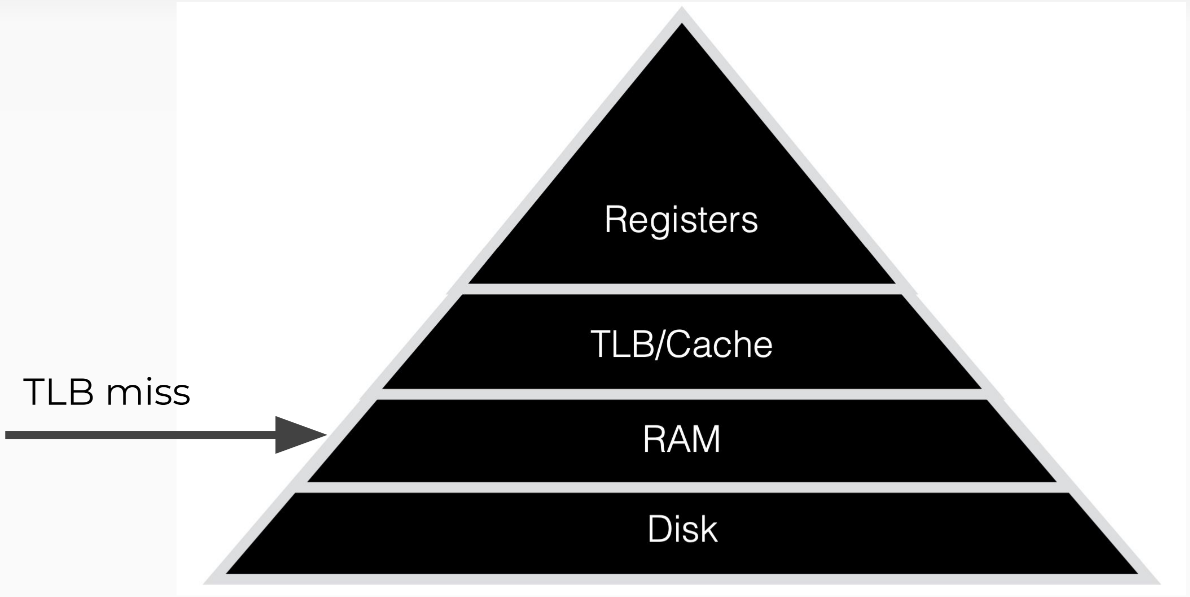
$$AMAT = TM + (PMiss * TD)$$

where TM represents the cost of accessing memory,
TD the cost of accessing disk, and
PMiss the probability of a miss

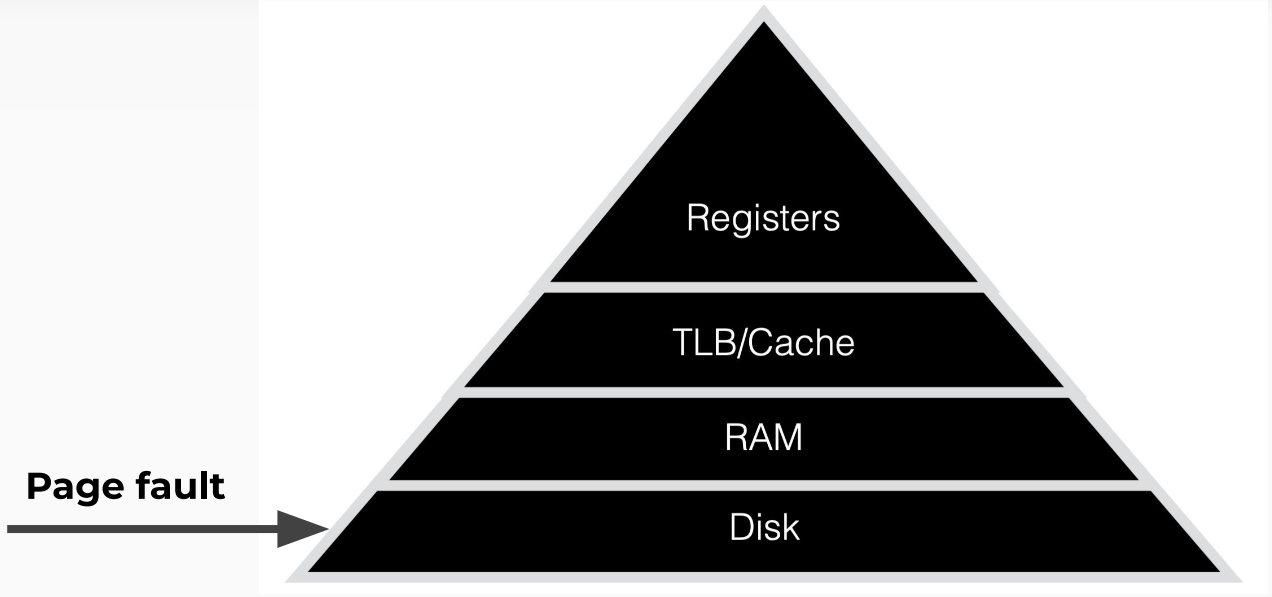
Page faults



Page faults



Page faults



Avg. Mem. Access Time = Time-Mem + (PMiss * Time-Disk)

Present Bit

- Program generates virtual memory address
- Translated into physical address
  - Extracts VPN from VA
  - Checks TLB (TLB hit or not)
  - TLB miss
    - Locates PT in memory
    - Looks up PTE and gets the page (if in memory)
  - Page not present in physical memory
  - Identify using Present bit
    - Present bit set to one means the page is in physical memory
    - If not, it's on the disk (**page fault**)

Page faults

- When page is not found, page-fault handler in the OS is triggered
    - Looks up PTE bits for obtaining this data
    - Fetch page from disk address in memory
- Swap in (Disk → Memory)
    - Mark page as present in PT
- Swap out (Memory → Disk)
- Expensive to handle page faults
    - Helps with multiprogramming

Memory Full?

- Memory may be full
    - *High watermark* and *low watermark*
    - Start evicting when fewer than LW pages
    - Stop when there are HW free pages available in memory
- Swap out (page out) one or more pages
- Pick a page to replace in memory
    - Page replacement policy
    - Need good page replacement policies
- Can swap a group of pages for efficiency

Optimal Page Replacement Strategy

- Example
  - Assume the following page accesses:

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- If we have 3 physical pages, how many page faults?
- With 4?

Optimal Page Replacement Strategy

- Optimal strategy requires knowing the future
    - Replace page that will not be used for longest period of time

FIFO Page Replacement Strategy

- Evict oldest fetched page in system
- Example
  - Assume the following page accesses:

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- If we have 3 physical pages, how many page faults?

FIFO Page Replacement Strategy

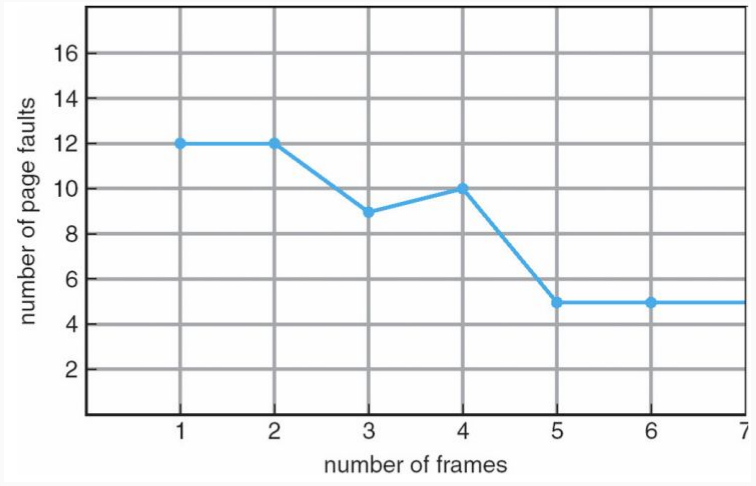
- Evict oldest fetched page in system
- Example
  - Assume the following page accesses:

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- If we have 3 physical pages,

  9 page faults
- With 4 physical pages?

Belady’s Anomaly

- More memory does not always mean lesser page faults



LRU Page Replacement Strategy

- Least recently used
  - Because past often predicts the future
  - Replace page that has least recently been used
- Example
  - Assume the following page accesses:

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- If we have 3 physical pages, how many page faults?
- With 4 pages?

Clock Algorithm

- Use accessed bit supported by most hardware
    - E.g., x86 will write 1 to A bit in PTE on first access
    - Software managed TLBs like MIPS can do the same
- Do FIFO but skip accessed pages
- Keep pages in circular FIFO list
- Scan:
    - page's A bit = 1, set to 0 & skip
    - else if A = 0, evict
- Second-chance replacement

Clock Algorithm

- Example
  - Assume the following page accesses:

    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- If we have 3 physical pages, how many page faults?
- With 4 pages?

Thrashing

- Too much paging