Operating Systems

Memory Management II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

What is the problem?

- Not enough memory
  - Having enough memory is not possible
    - How do you determine “enough”?
    - Programs keep growing in size.
- Processor does not execute anything that is not in the memory.

Have you ever thought …

- Why the *text* segment (Do you remember heap, stack, text, and data?) of any process starts at the same address?
- Why don't you run out of memory even if the sum of memory requirement of each program you are running at the same time exceeds the amount of memory you have in your machine?
- The address is 64-bit, in 64-bit machines. It accesses memory from 0 to 2^{64} - 1 which is way more than the amount of memory you have on your machine?

How come?

But We Can See That …

- All memory references are logical(virtual) addresses that are dynamically translated into physical addresses at run time
- A process may be broken up into several pieces that don't need to be contiguously located in main memory during execution.

So:

It is not necessary that all the pieces of a process be in main memory during execution.

Definition of Virtual Memory

Mapping from logical (virtual) address space to physical address space

Implementation of Virtual Memory Using Paging

The Story

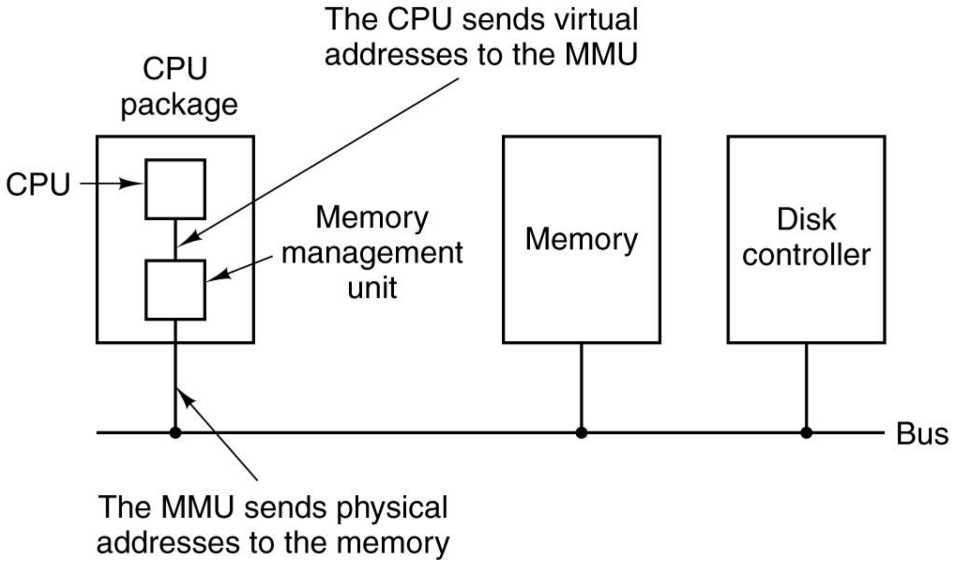
1. Operating system brings into main memory a few pieces of the program.
2. An interrupt is generated when an address is needed that is not in main memory.
3. Operating system places the process in a blocking state.
4. Operating system issues a disk I/O Read request.
5. Another process is dispatched to run while the disk I/O takes place.
6. An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
7. Piece of process that contains the logical address is brought into main memory.

The Story

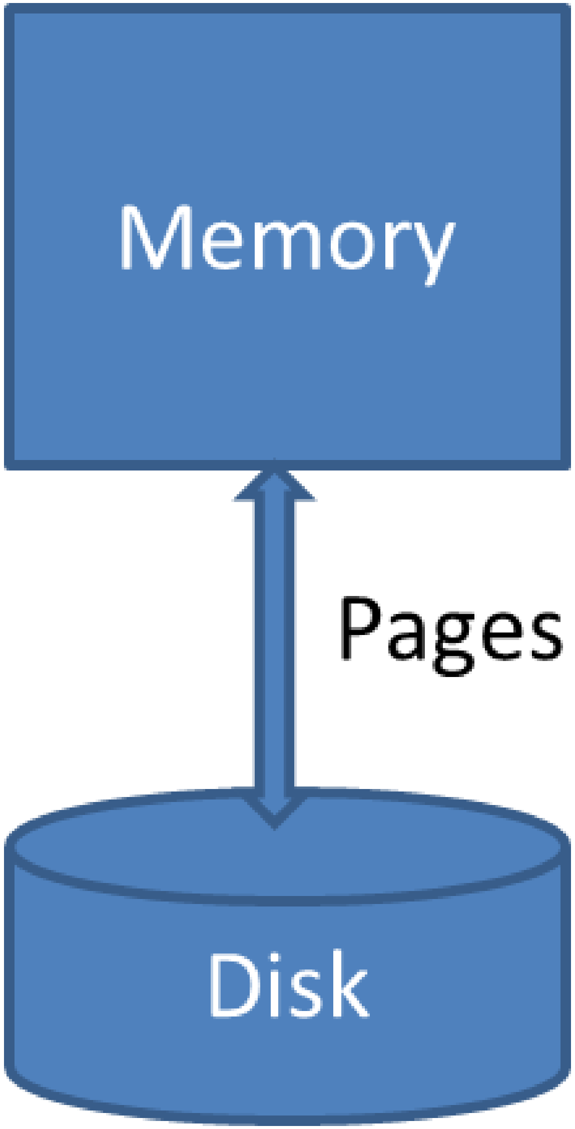
1. Operating system brings into main memory a few pieces of the program. What do you mean by “pieces”?
2. An interrupt is generated when an address is needed that is not in main memory. How do you know it isn’t in memory?
3. Operating system places the process in a blocking state.
4. Operating system issues a disk I/O Read request. Why?
5. Another process is dispatched to run while the disk I/O takes place.
6. An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
7. Piece of process that contains the logical address is brought into main memory. What if memory is full?

Virtual Memory

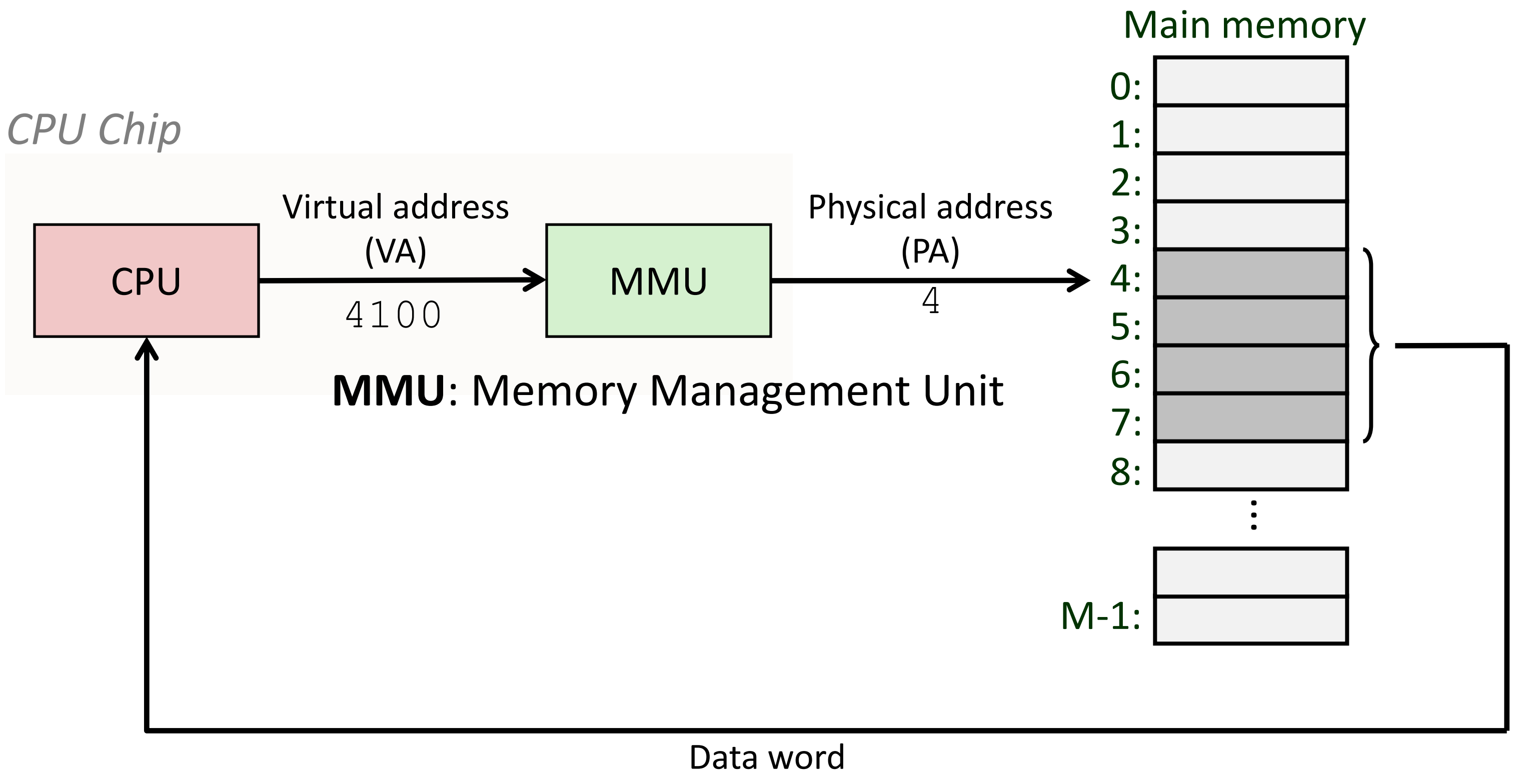
- Each program has its own address space
- This address space is divided into pages
- Pages are mapped into physical memory


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Use part of the disk as extension to the memory!



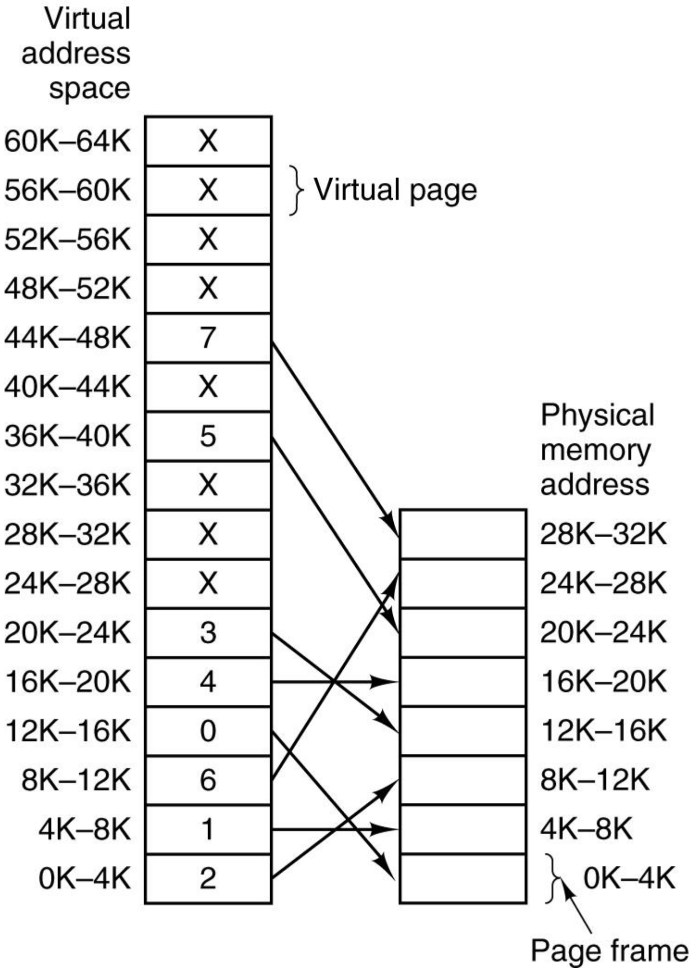
Virtual Addressing



MMU: Memory Management Unit

- Used in all modern machines.

Virtual Memory

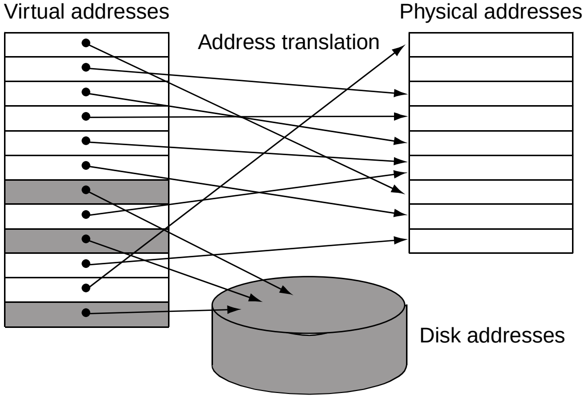


Virtual Address Space

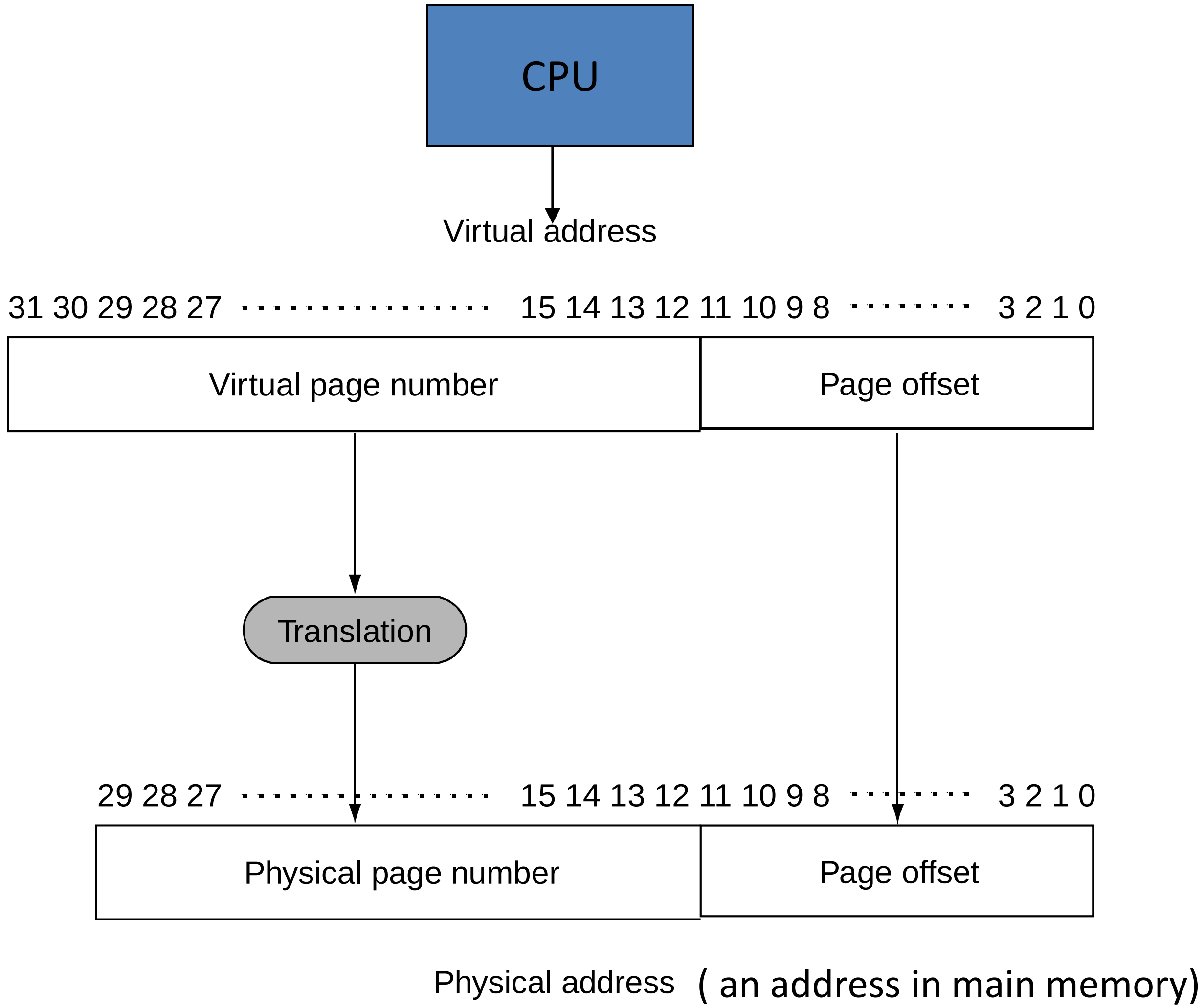
Physical Memory

Virtual Memory

- We can think of memory as acting as a cache to the secondary storage (disk)



- Advantages:
  - illusion of having more physical memory
  - program relocation
  - protection



CPU

Virtual address

31 30 29 28 27 · · · · · · · · · · · · · · · · 15 14 13 12 11 10 9 8 · · · · · · · · · 3 2 1 0

Virtual page number

Page offset

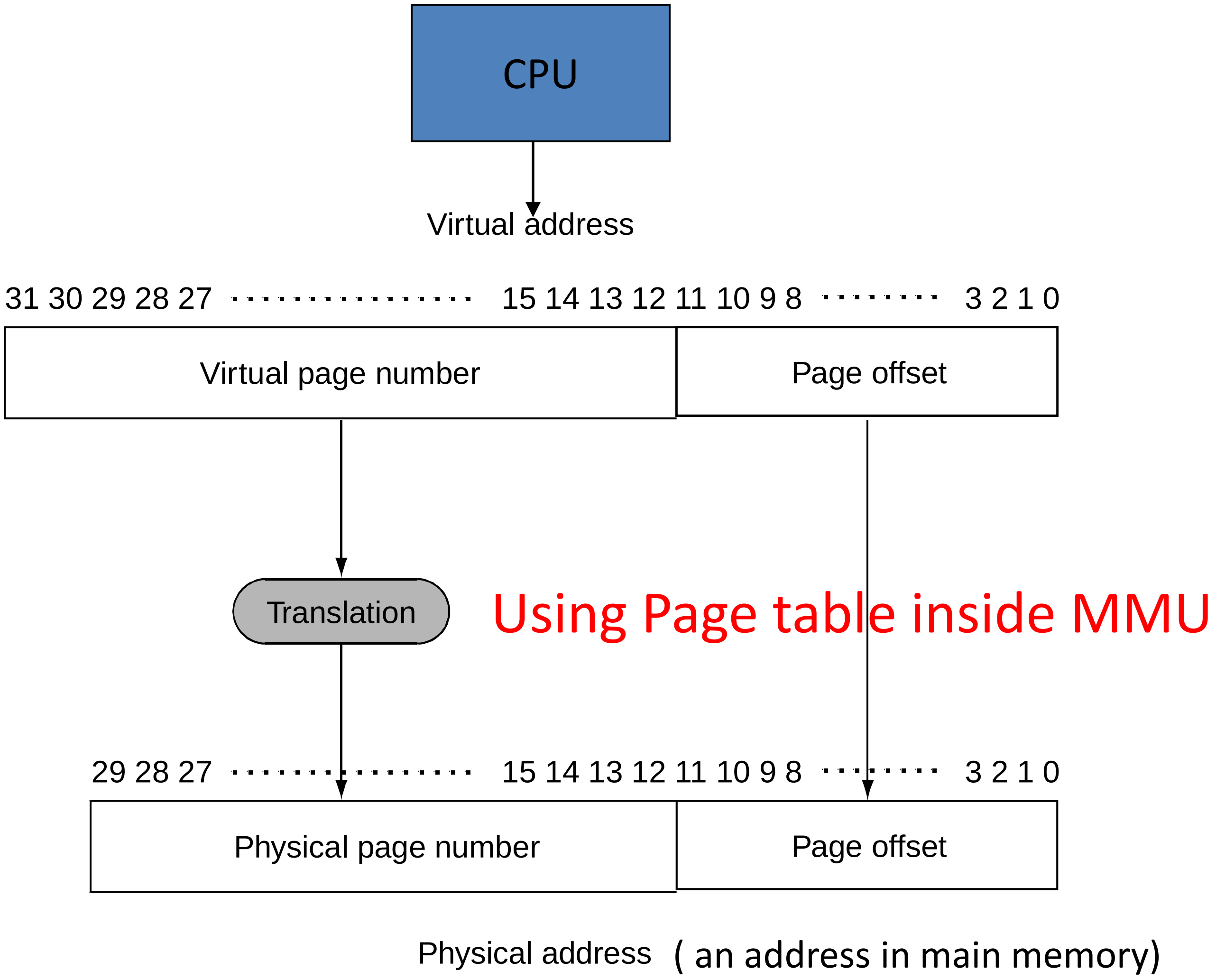
Translation

29 28 27 · · · · · · · · · · · · · · · · 15 14 13 12 11 10 9 8 · · · · · · · · · 3 2 1 0

Physical page number

Page offset

Physical address (an address in main memory)



CPU

Virtual address

31 30 29 28 27 ················ 15 14 13 12 11 10 9 8 ········ 3 2 1 0

Virtual page number

Page offset

Translation

Using Page table inside MMU

29 28 27 ················ 15 14 13 12 11 10 9 8 ········ 3 2 1 0

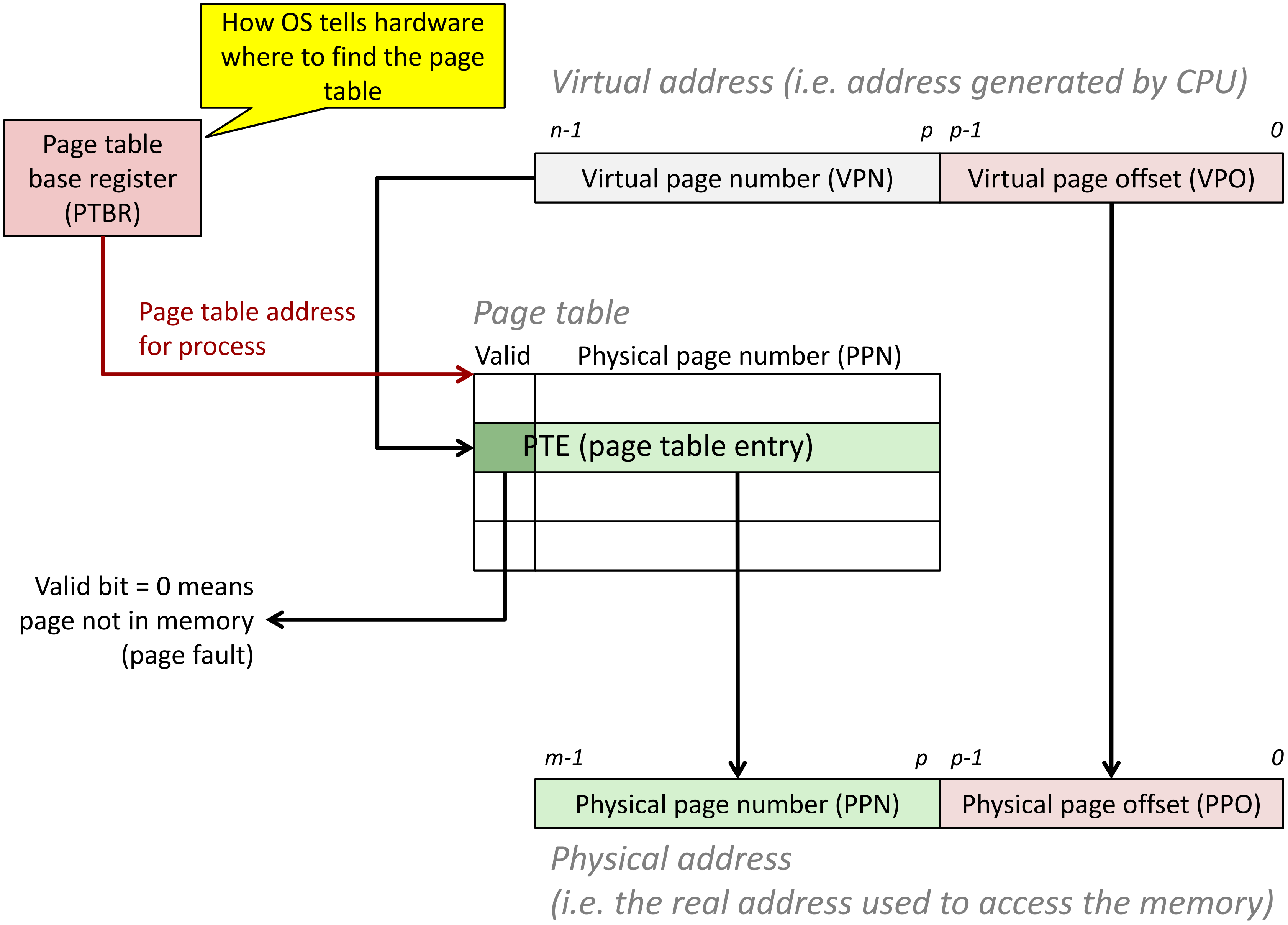
Physical page number

Page offset

Physical address (an address in main memory)

MMU = Memory Management Unit

Address Translation w/ a Page Table



How OS tells hardware where to find the page table

Page table base register (PTBR)

Virtual address (i.e. address generated by CPU)

n-1 *p* *p-1* *0*

Virtual page number (VPN)

Virtual page offset (VPO)

Page table address for process

Page table

Valid

Physical page number (PPN)

PTE (page table entry)

Valid bit = 0 means page not in memory (page fault)

m-1 *p* *p-1* *0*

Physical page number (PPN)

Physical page offset (PPO)

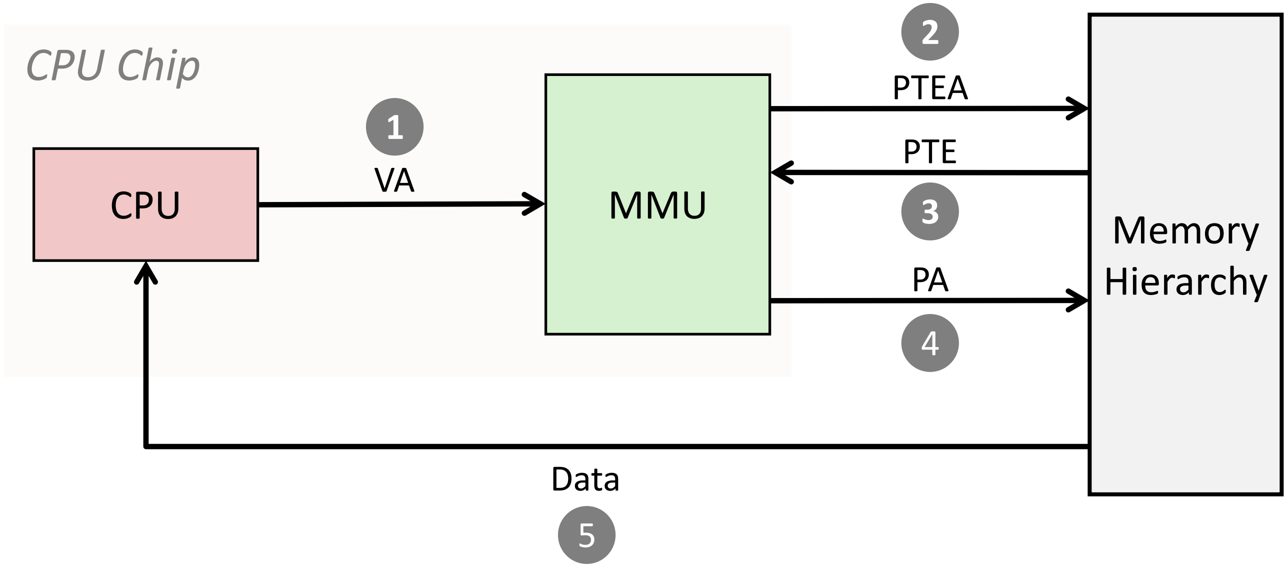
Physical address

(i.e. the real address used to access the memory)

Page Table Base Register (PTBR)

- The operating system maintains information about each process in a process control block.
- The page table base address for the process is stored there.
- The operating system loads this address into the PTBR whenever a process is scheduled for execution.
- Only the kernel (i.e. the OS) can access PTBR

Address Translation: Page Hit



VA: Virtual Address
PA: Physical Address
PTE: Page Table Entry
PTEA: PTE Address

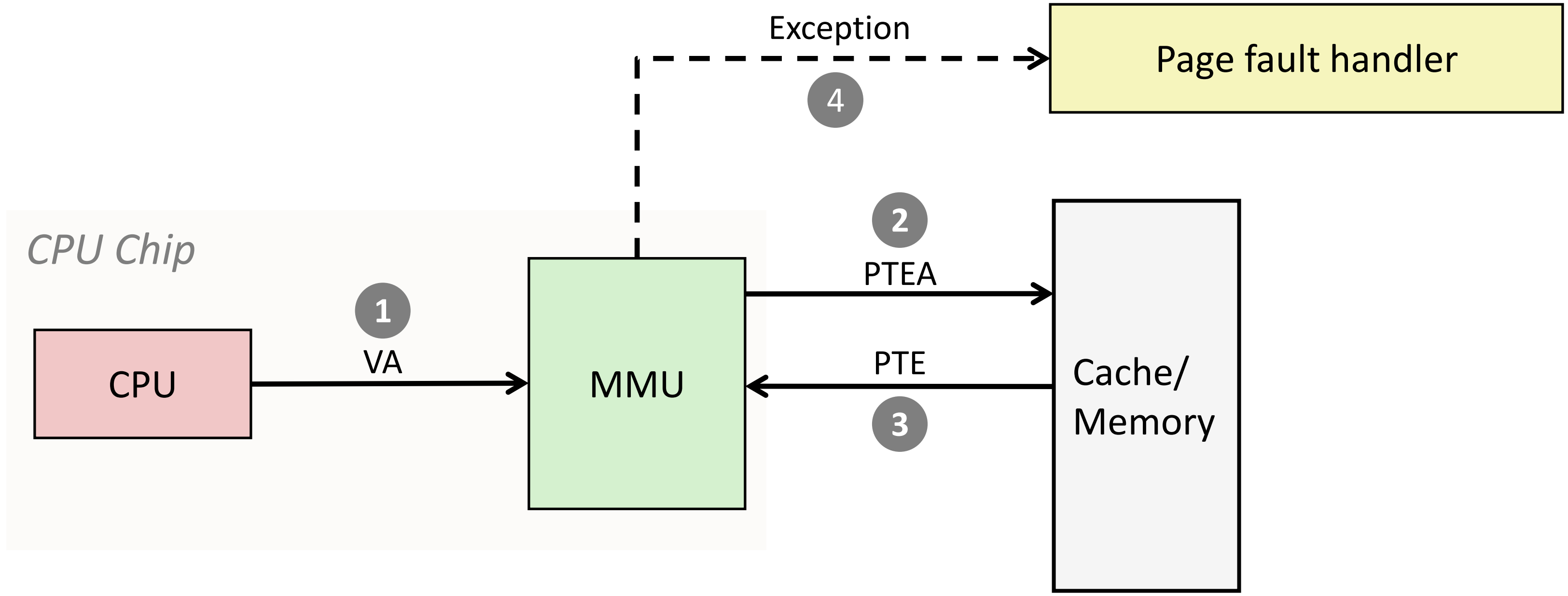
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception in kernel

If VA is invalid, then kill process (SIGSEGV)

If VA has been paged out to disk, then swaps in faulted page, update page table, resume faulted process

There are two challenges

- **Speed**: VA to PA translation means we need to access the memory twice for each CPU memory request!

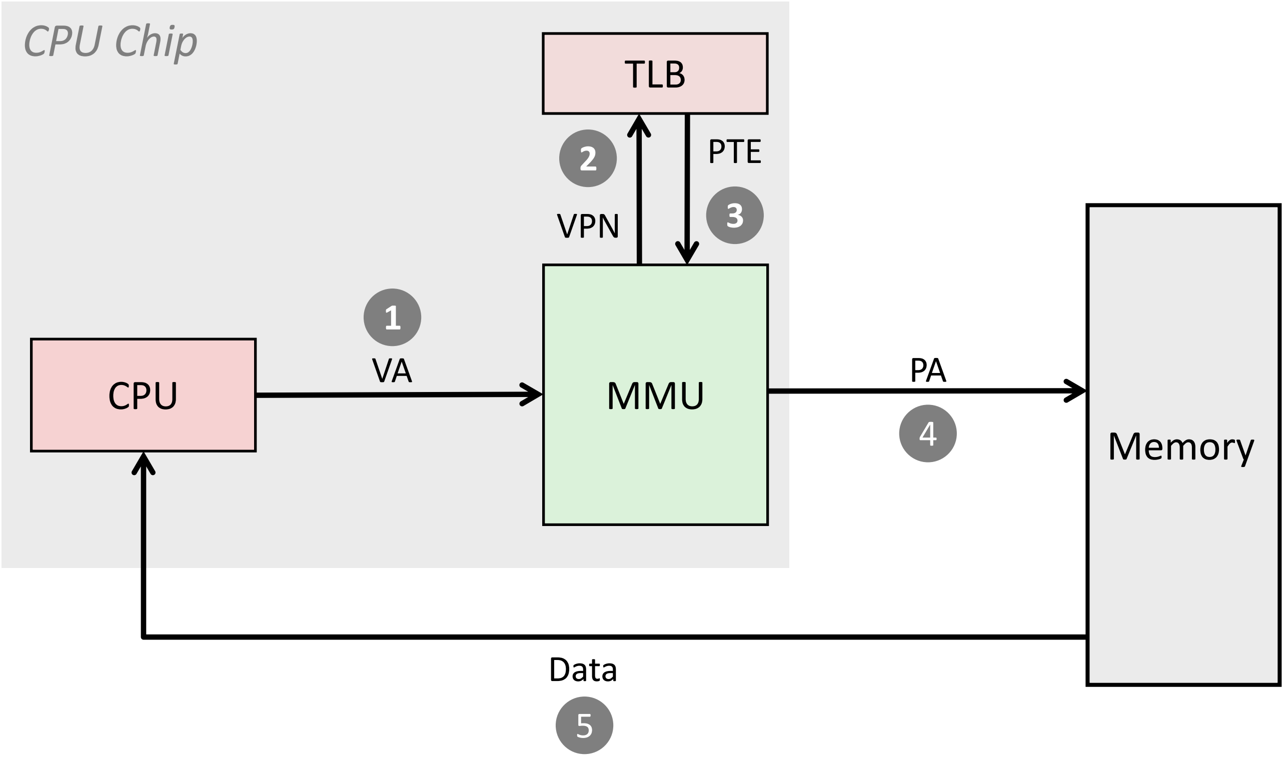
- **Size**: page table can be huge. And we have a page table per process.

Speeding Up Virtual Memory: Translation Lookaside Buffer (TLB)

TLB

- **Observation**: most programs tend to make a large number of references to a small number of pages -> only fraction of the page table is heavily used
- **Solution:** *Translation Lookaside Buffer* (TLB)
  - Small hardware cache inside the MMU (i.e. on chip)
  - Maps virtual page numbers to physical page numbers address without going to the page table
  - Contains complete page table entries for small number of pages

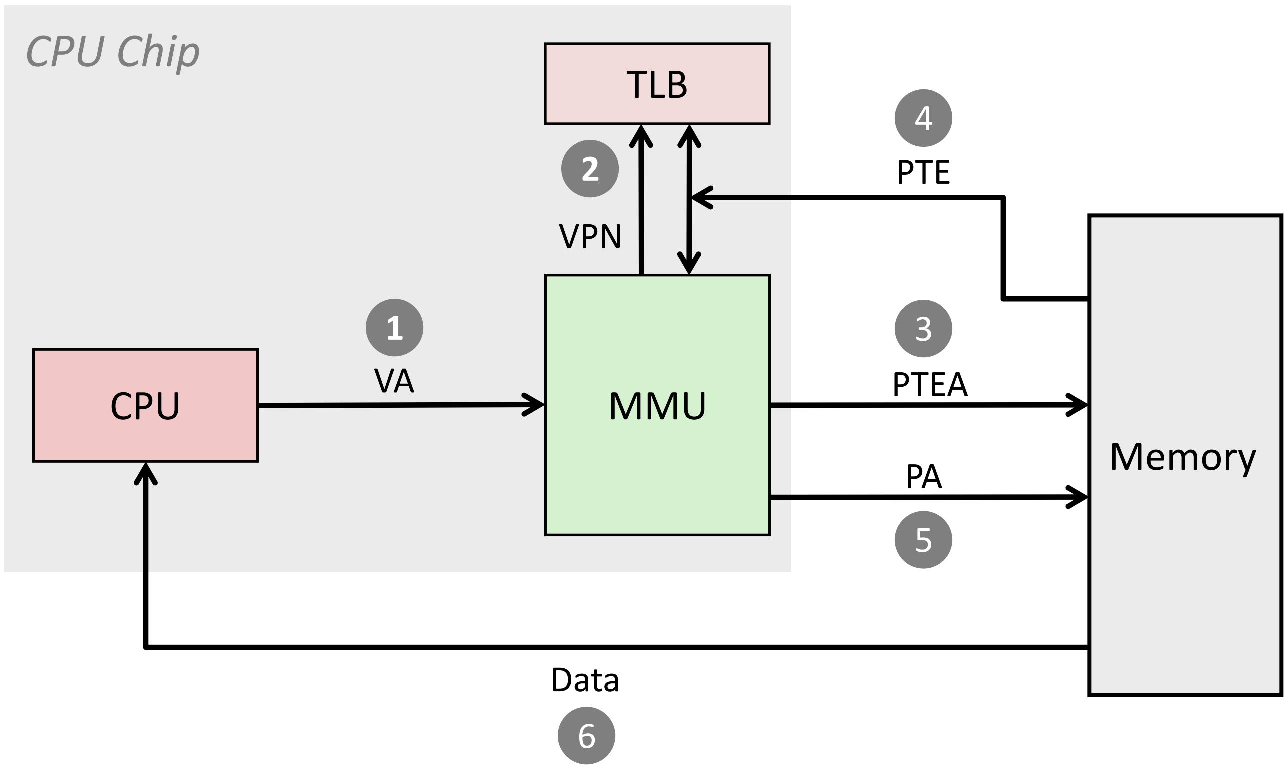
TLB Hit



A TLB hit eliminates a memory access

VA: Virtual Address **PA:** Physical Address **PTE:** Page Table Entry

TLB Miss



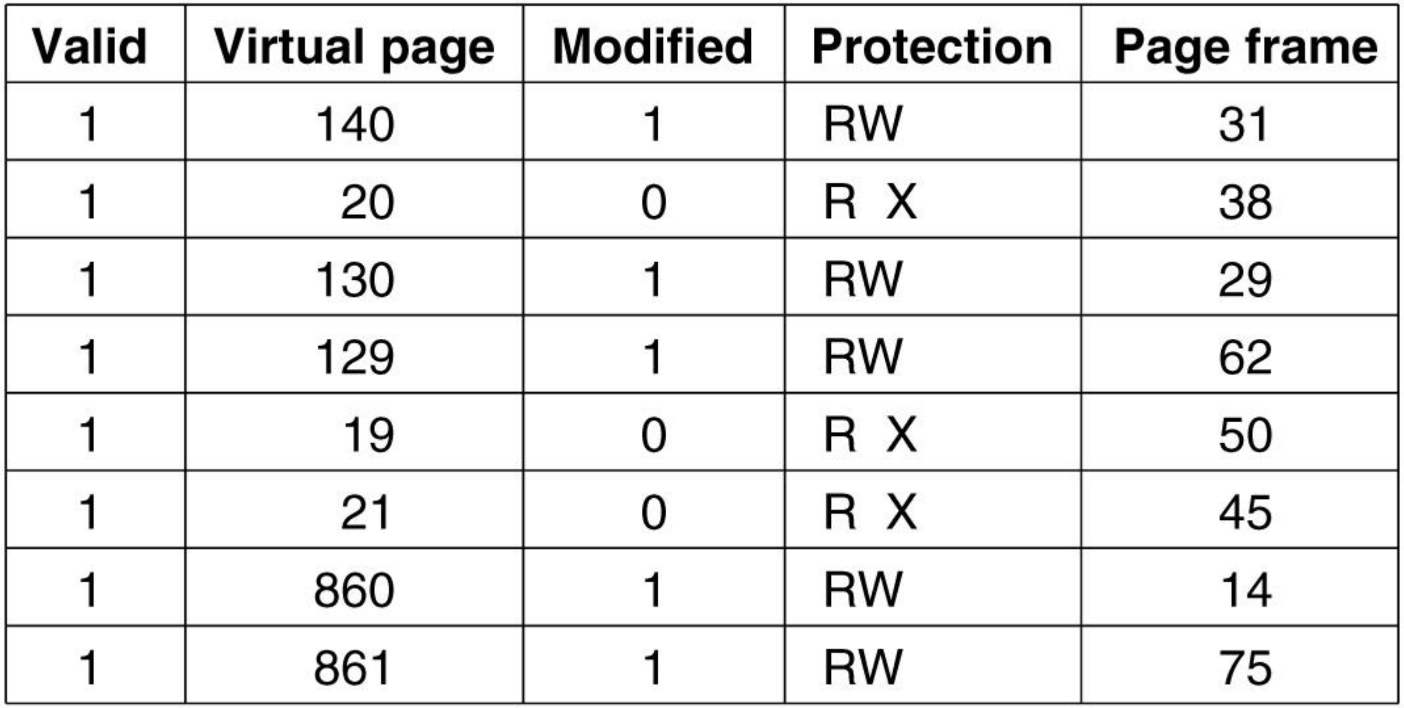
A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare.

TLB

- In case of TLB miss (did not find the address translation you want) -> MMU accesses page table
- TLB misses occur more frequently than page faults

Example of contents of a TLB



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

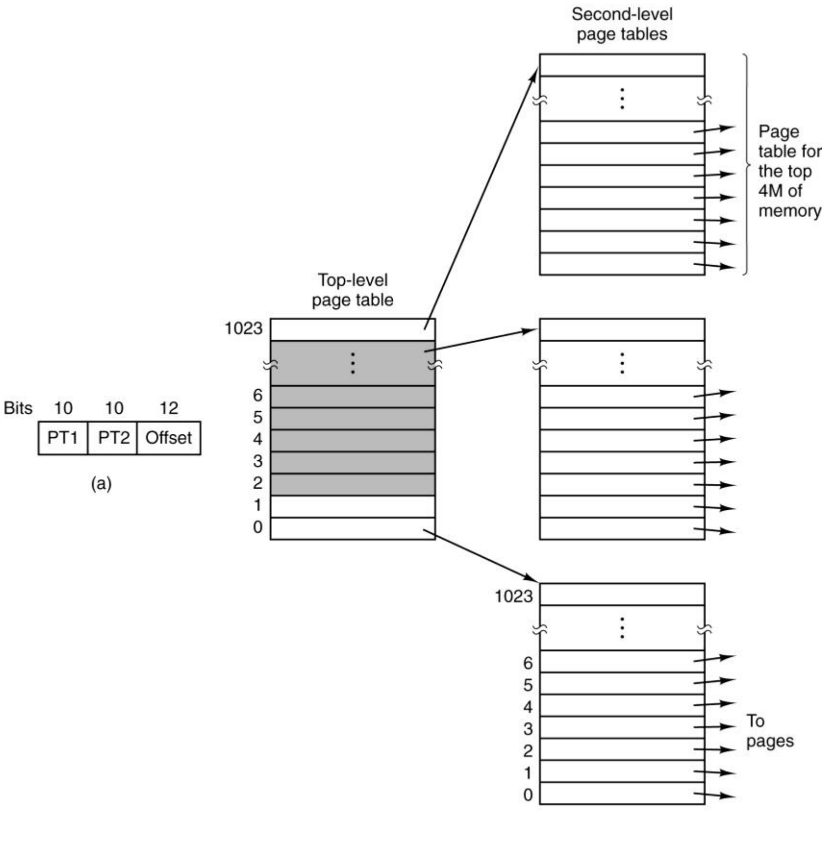
TLB In Case of Context Switch

- TLB contains translations from the page table of a process.
- If a new process is schedule for execution, the page table of that new process is used.
- **Solution 1:** TLB must be flushed.
- **Solution 2:** TLB augmented with process ID.

Tackling The Page Table Size Problem

Multi-Level Page Table

- To reduce storage overhead in case of large memories



(a)

(b)

Why Two-level Page Table Reduces Memory Requirement?

- If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist.
- Only the level 1 table needs to be in main memory at all times.
- The level 2 page tables can be created and paged in and out by the VM system as they are needed.

Inverted Page Table

- Page table has one entry per page frame not one entry per page.
  - For each page slot (frame) it shows which virtual page it has.

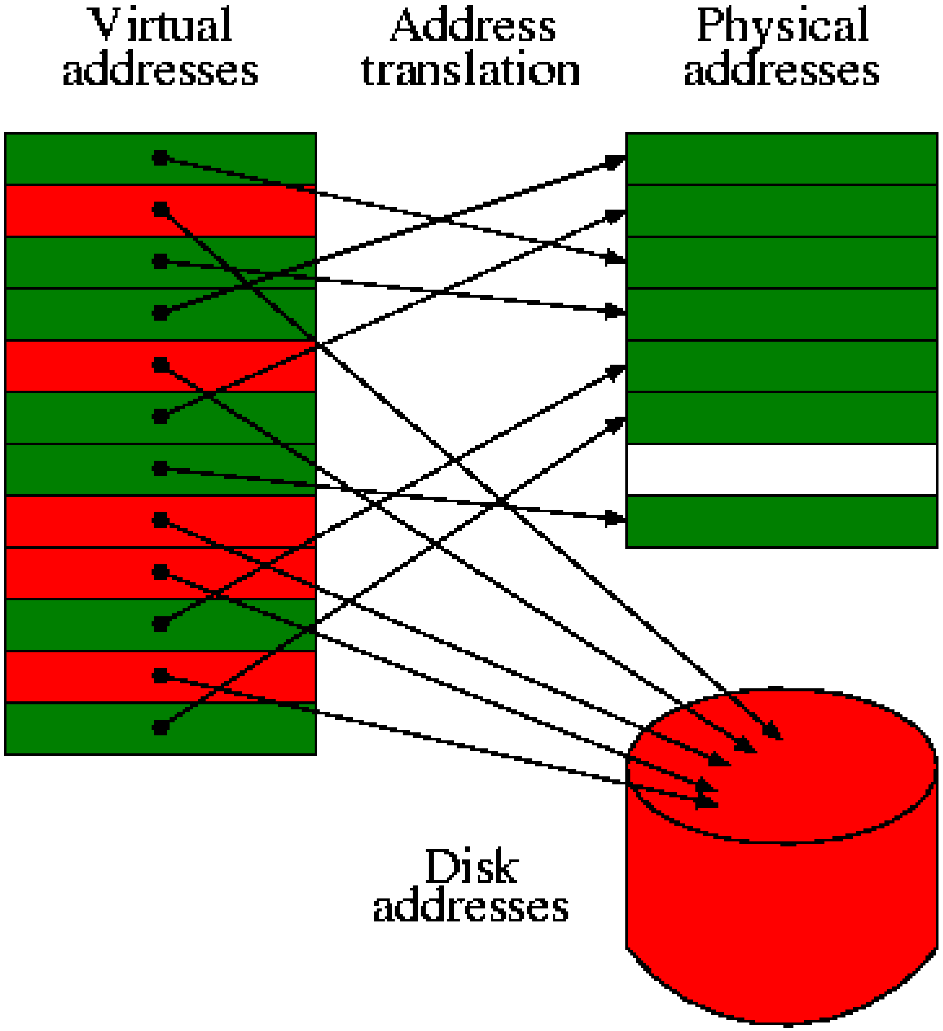
+ Save vast amount of storage

- virtual-to-physical translation much harder

What About Page Faults?

Definition

- A Page fault is when the page table is consulted, and we found that the requested page is not in the memory but in the disk.
  - That is, the page has been swapped out because the memory was full.
- Note: In reality, the OS does not wait till memory is full to start making page replacement because the OS wants to have some spare memory free just in case.

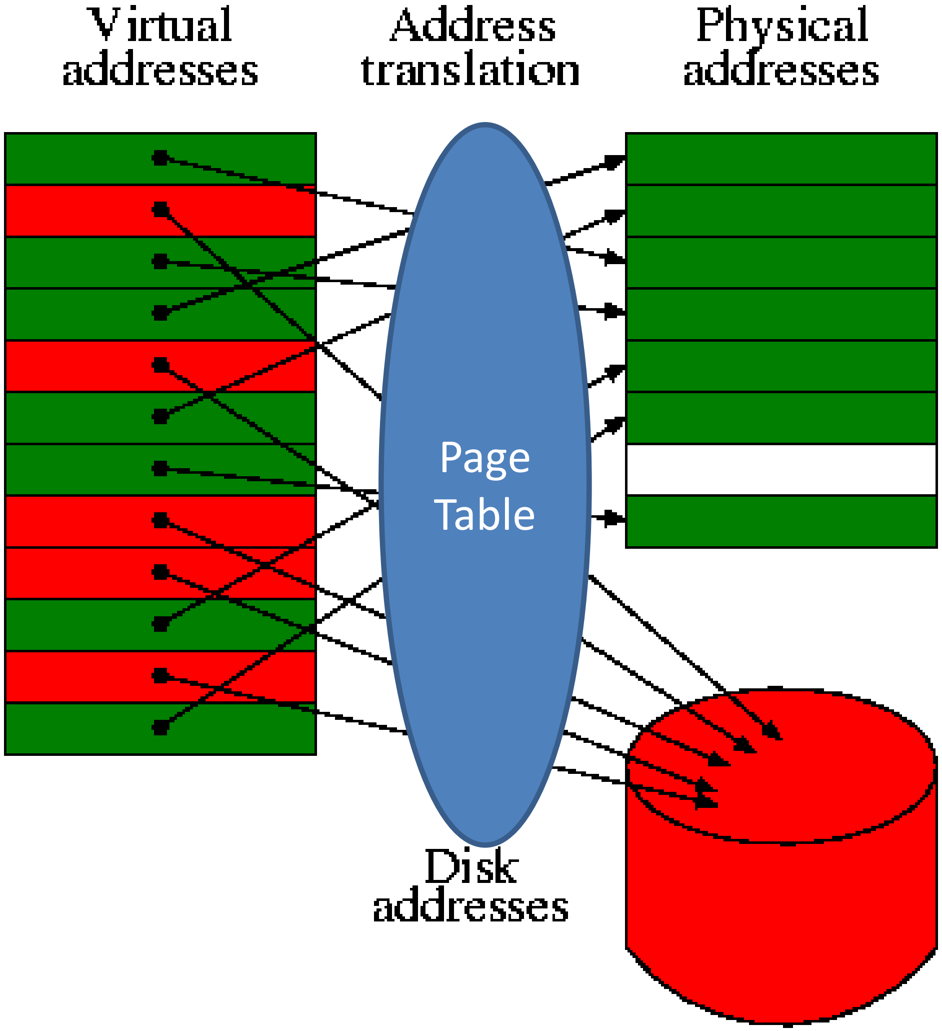


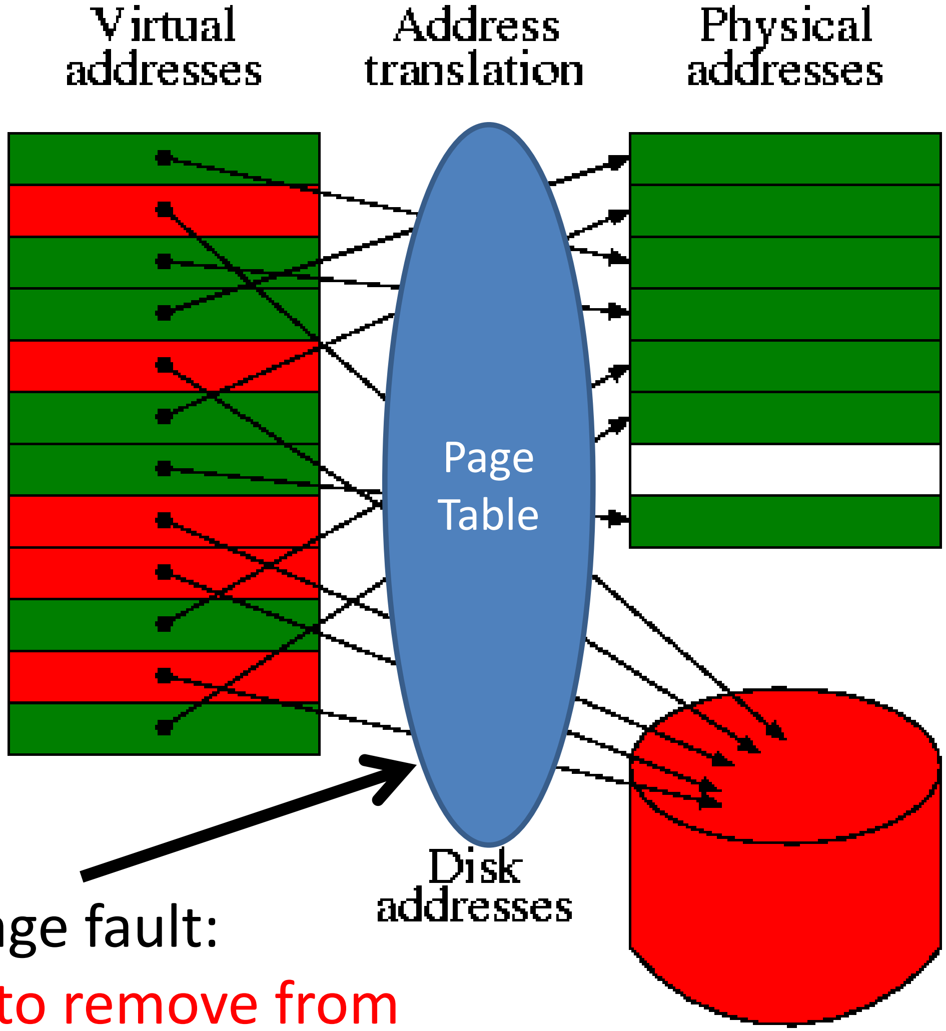
Virtual addresses

Address translation

Physical addresses

Disk addresses


Virtual addresses
Address translation
Physical addresses
Page Table
Disk addresses



Virtual addresses

Address translation

Physical addresses

Page Table

Disk addresses

In case of page fault:

which page to remove from Memory if the memory is full?

Replacement Policies

- Used in many contexts when storage is not enough
  - caches
  - web servers
  - pages
- Things to consider when designing a replacement policy
  - measure of success
  - cost

Optimal Page Replacement Algorithm

- Each page labeled with the number of instructions that will be executed before this page is referenced
- Page with the highest label should be removed
- Impossible to implement because we do not know the future.

The Not Recently Used (NRU) Replacement Algorithm

- Two status bits with each page
  - R: Set whenever the page is referenced (used)
  - M: Set when the page is written
- R and M bits are available in most computers implementing virtual memory
- Those bits are updated with each memory reference
  - Must be updated by hardware
  - Reset only by the OS
- Periodically (e.g. on each clock interrupt) the R bit is cleared
  - To distinguish pages that have been referenced recently

The Not Recently Used (NRU) Replacement Algorithm

	R	M
Class 0:	0	0
Class 1:	0	1
Class 2:	1	0
Class 3:	1	1

NRU algorithm removes a page at random from the lowest numbered nonempty class

The FIFO Replacement Algorithm

- OS maintains a list of the pages currently in memory
- The most recent arrival at the tail
- On a page fault, the page at the head is removed

The Second-Chance Page Replacement Algorithm

- Modification to FIFO
- Inspect the R bit of the oldest page
  - If R=0 page is old and unused -> replace
  - If R=1 then
    - bit is cleared
    - page is put at the end of the list
    - the search continues
- If all pages have R=1, the algorithm degenerates to FIFO

The Least Recently Used (LRU) Page Replacement Algorithm

- Good approximation to optimal
- When page fault occurs, replace the page that has been unused for the longest time
- Realizable but not cheap

LRU
Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enough to include the value of the counter
- After each memory reference to a page, the counter is written in the entry's counter field.
- At page fault, the page with lowest value is discarded

LRU
Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enou unter
- Afte age, the coun ter field.
- At page fault, the page with lowest value is discarded

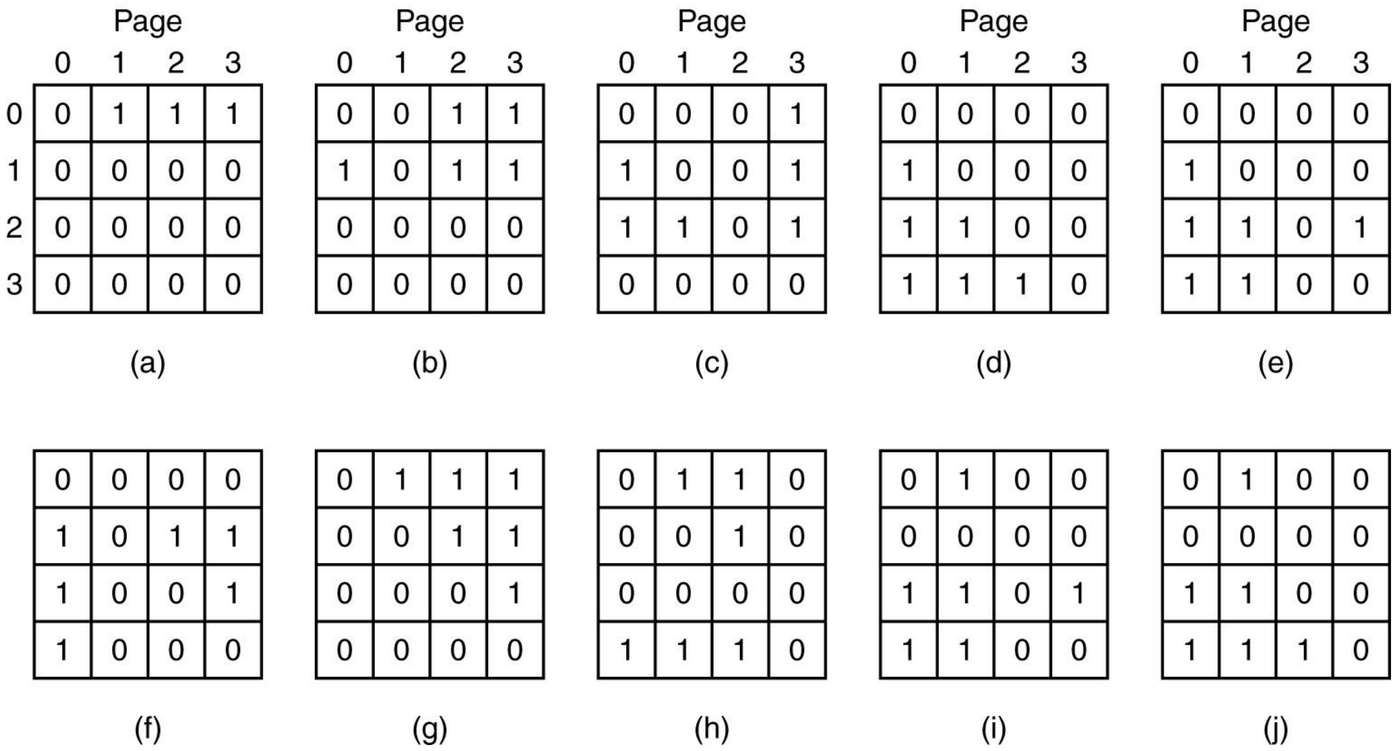
Too expensive!

Too Slow!

LRU:
Hardware Implementation 2

- Machine with n page frames
- Hardware maintains a matrix of nxn bits
- Matrix initialized to all 0s
- Whenever page frame k is referenced
  - Set all bits of row k to 1
  - Set all bits of column k to 0
- The row with lowest value is the LRU

LRU: Hardware Implementation 2



Page

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

Page

0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

(b)

Page

0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

(c)

Page

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

(d)

Page

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	1
1	1	0	0

(e)

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

(f)

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

(g)

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

(h)

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

(i)

0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

(j)

Pages referenced: 0 1 2 3 2 1 0 3 2 3

LRU Implementation

- Slow
- Few machines have required hardware

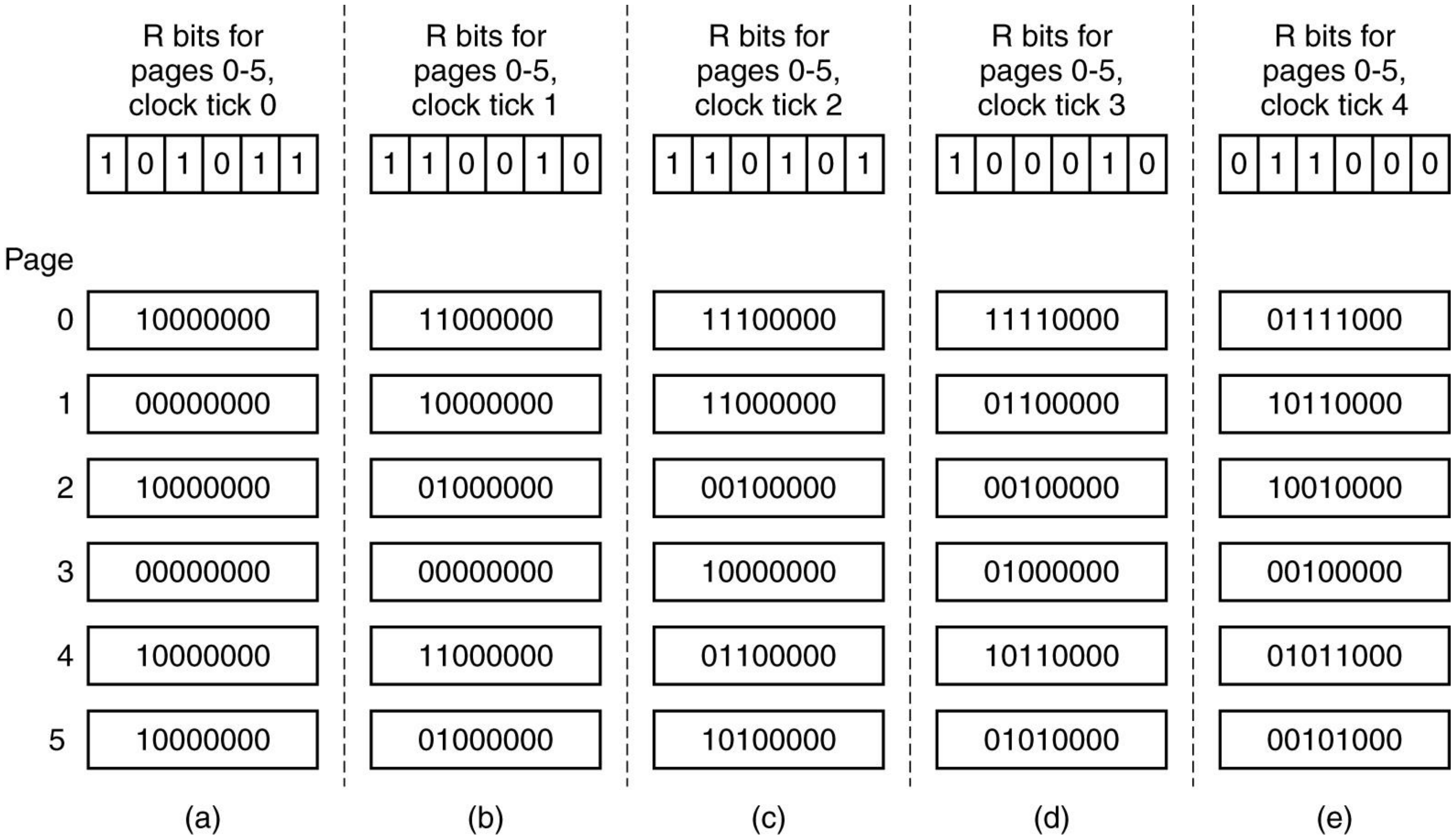
Simulating LRU in Software

- Not Frequently Used (NFU) algorithm
- Software counter associated with each page, initially zero
- At each clock interrupt, the OS scans all pages and add the R bit of each page to the counter of that page.
- At page fault: the page with lowest counter is replaced

Enhancing NRU

- NRU never forgets anything -> high inertia
- Modifications:
  - shift counter right 1 bit before adding R
  - R is added to the leftmost
- This modified algorithm is called aging
- The page whose counter is lowest is replaced at page fault

Aging Algorithm



Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

The Working Set Model

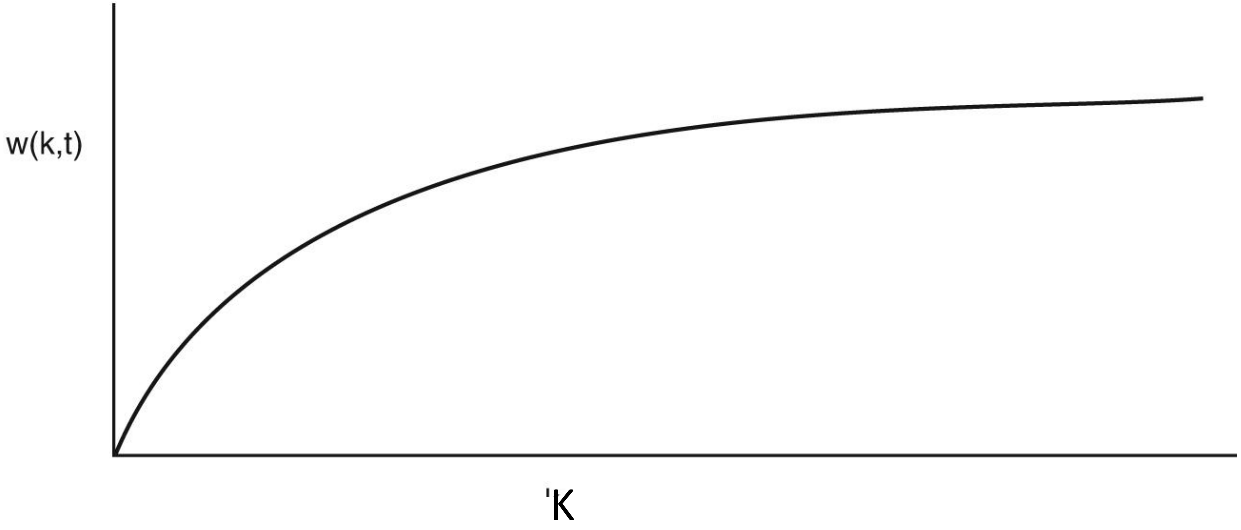
- Working set: the set of pages that a process is currently using
- Thrashing: a program causing page faults every few instructions

An important question:

In multiprogramming systems, processes are sometimes swapped to disk (i.e. all their pages are removed from memory). When they are brought back, which pages to bring?

The Working Set Model

- Try to keep track of each process' working set and make sure it is in memory before letting the process run.



w(k,t): the set of pages accessed in the last k references at instant t

The Working Set Model

- OS must keep track of which pages are in the working set
- Replacement algorithm: evict pages not in the working set
- Possible implementation (but expensive):
  - working set = set of pages accessed in the last k memory references
- Approximations
  - working set = pages used in the last 100 msec

Some Design Issues

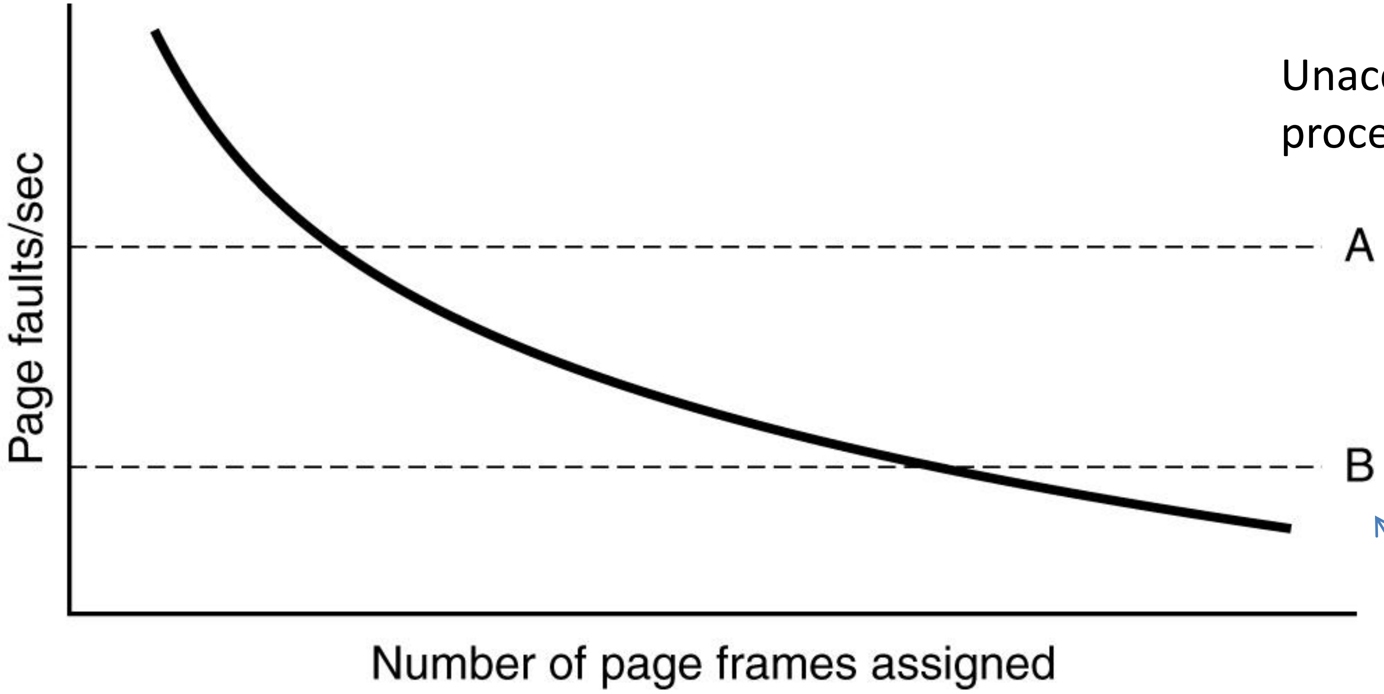
Design Issues for Paging: Local vs Global Allocation

- How memory should be allocated among the competing runnable processes?
- Local algorithms: allocating every process a fixed fraction of the memory
- Global algorithms: dynamically allocate page frames
- Global algorithms work better
  - If local algorithm used and working set grows → thrashing will result
  - If working set shrinks → local algorithms waste memory

Global Allocation

- **Method 1:** Periodically determine the number of running processes and allocate each process an equal share
- **Method 2** (better): Pages allocated in proportion to each process total size (in terms of number of pages).
- **Page Fault Frequency (PFF) algorithm**: tells when to increase/decrease page allocation for a process but says nothing about which page to replace.

Global Allocation: PFF



Unacceptable high
process given more pages

so low, process
has too much memory
page frames may be taken away

Design Issues: Load Control

- What if PFF indicates that some processes need more memory but none need less?
- Swap some processes to disk and free up all the pages they are holding.
- Which process(es) to swap?
  - Strive to make CPU busy (I/O bound vs CPU bound processes)
  - Process size

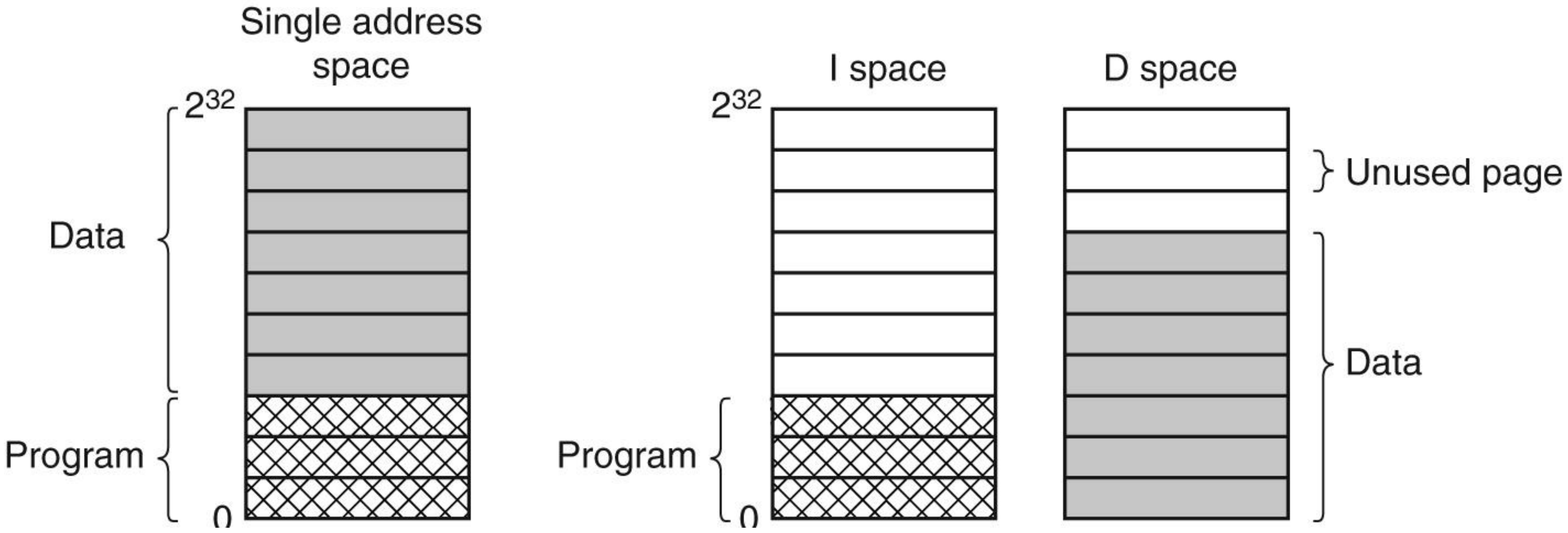
Design Issues: Page Size

- Large page size → internal fragmentation
- Small page size →
  - larger page table
  - More overhead transferring from disk

Design Issues: Page Size

- Assume:
  - s = process size
  - p = page size
  - e = size of each page table entry
- So:
  - number of pages needed = s/p
  - occupying: se/p bytes of page table space
  - wasted memory due to fragmentation: $p/2$
  - overhead = $se/p + p/2$
- We want to minimize the overhead:
  - Take derivative of overhead and equate to 0:
  - $-se/p^2 + \frac{1}{2} = 0$ → $p = \sqrt{2se}$

Design Issues:
Separate Instruction (I) and Data (D) Spaces



- The linker must know about it
- Paging can be used in each separately

Design Issues: Shared Pages

- To save space, when same program is running by several users for example
- If separate I and D spaces: process table has pointers to Instruction page table and Data page table
- In case of common I and D spaces:
  - Special data structure is needed to keep track of shared pages
  - Copy on write for data pages

Design Issues:
Shared Libraries

- Dynamically linked
- Loaded when program is loaded or when functions in them are called for the first time
- Compilers must not produce instructions using absolute addresses in libraries → position-independent code

Design Issues: Cleaning Policy

- Paging daemon
- Sleeps most of the time
- Awakened periodically to inspect state of the memory
- If too few page slots are free -> daemon begins selecting pages to evict

Conclusions

- Virtual memory is very widely used
- Many design issues for paging systems:
    - Page replacement algorithm
    - Page size
    - Local vs Global Allocation
        - Global algorithms work better
    - Load control
    - Dealing with shared pages