EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical &
Computer Engineering

Boston University

Slides from Christopher Kruegel, UCSB

Memory Abstractions

- The set of addresses a program can refer to is called its *address space*
- Trouble awaits if all programs have the same address space
  - No protection from each other – errors in one program can cause damage to others
  - Programs must be written cooperatively, knowing about where others are located in memory
- We would like to create an abstraction, so that each process has a *private address space*: make 0x1234 in Program A different from 0x1234 in Program B

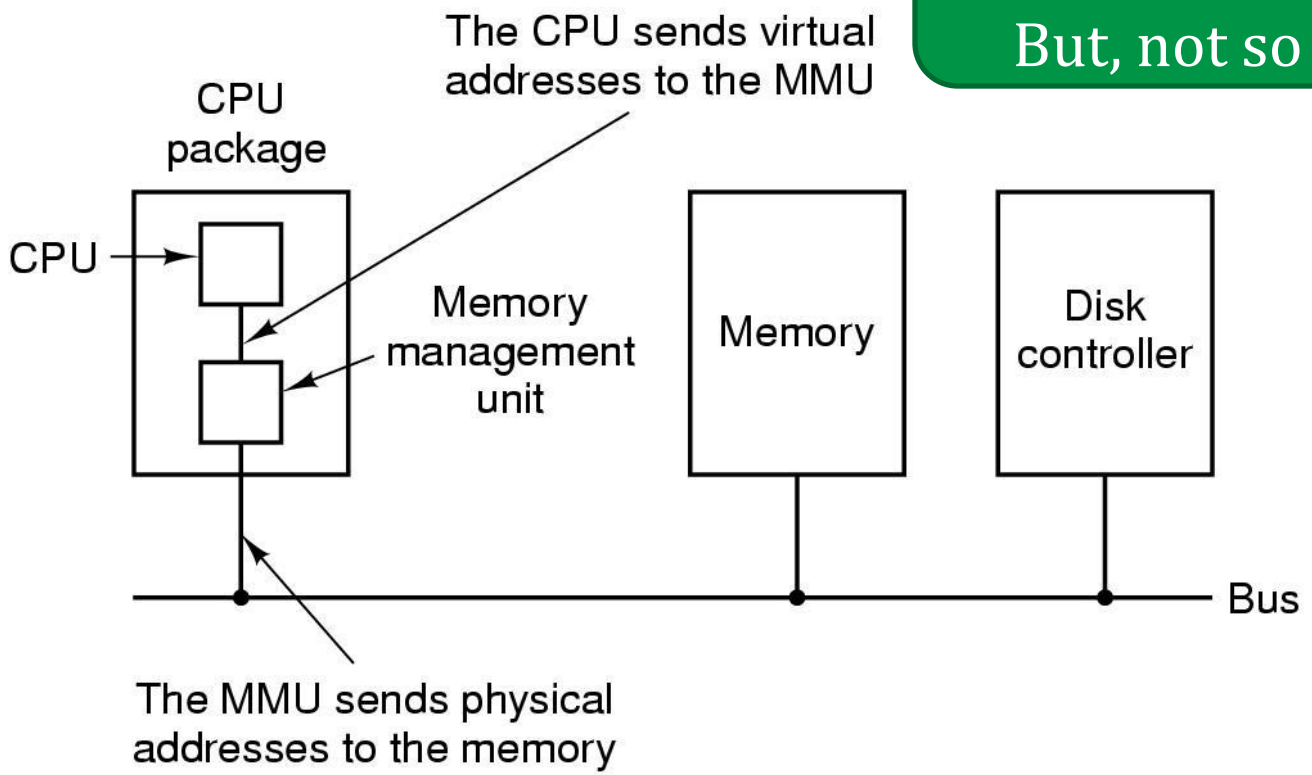
Virtual Memory

- Recall from last time – segmentation is no longer used to separate processes' memory from on another
- Instead, *virtual addressing* is used
- Virtual memory is used to map contiguous virtual addresses to a discontiguous set of physical pages
- OS (with help from the hardware) manages the mapping between pages and page frames
- Each memory access no longer refers directly to physical memory, but instead is *mapped* to some actual physical address

Memory Management Unit (MMU)

Automatically performs the mapping from virtual addresses to physical addresses

Done!
See you next time.
But, not so quick!

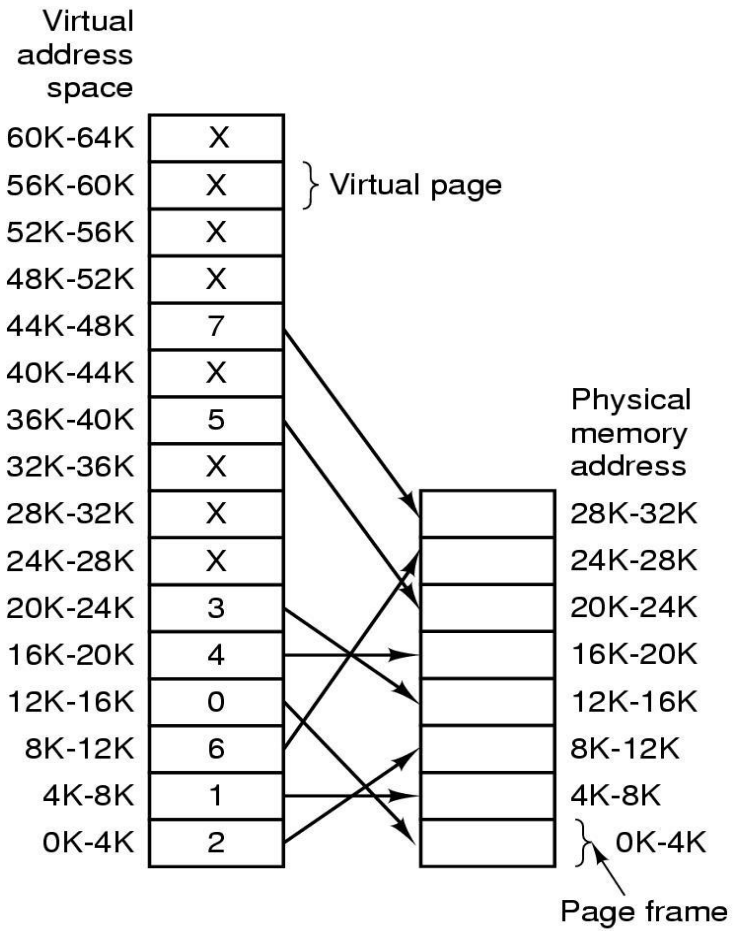


Paging

- Virtual memory is broken up into fixed-sized units (commonly, $0x1000_{16}$ == $4{,}096_{10}$ bytes) called *pages*
- Page sizes can vary though:
  - 32-bit x86 supports 4KB and 4MB pages
  - 64-bit x86 supports 4KB, 2MB, and 1GB pages
- The underlying physical pages of memory are called *page frames*

Mapping Pages to Page Frames

- Virtual memory: 64KB
- Physical memory: 32KB
- Page size: 4KB
- # Virtual memory pages: 16
- # Page Frames: 8



Page Faults

- What happens if we try to access a page that is not mapped?
- The MMU notices, and we raise a CPU exception called a *page fault*
- Control is passed to the OS (via interrupt) to decide what to do
  - Kill the process (segmentation fault)
  - Find some physical page to map to it

Programs Bigger than Memory

- Note that this gives us a way to have programs that don't all fit into memory at once
- We can just map in the parts of the program we're using right now
- If we hit code or data that isn't mapped, we can *swap* some other page to disk and update the mappings

Types of Page Faults

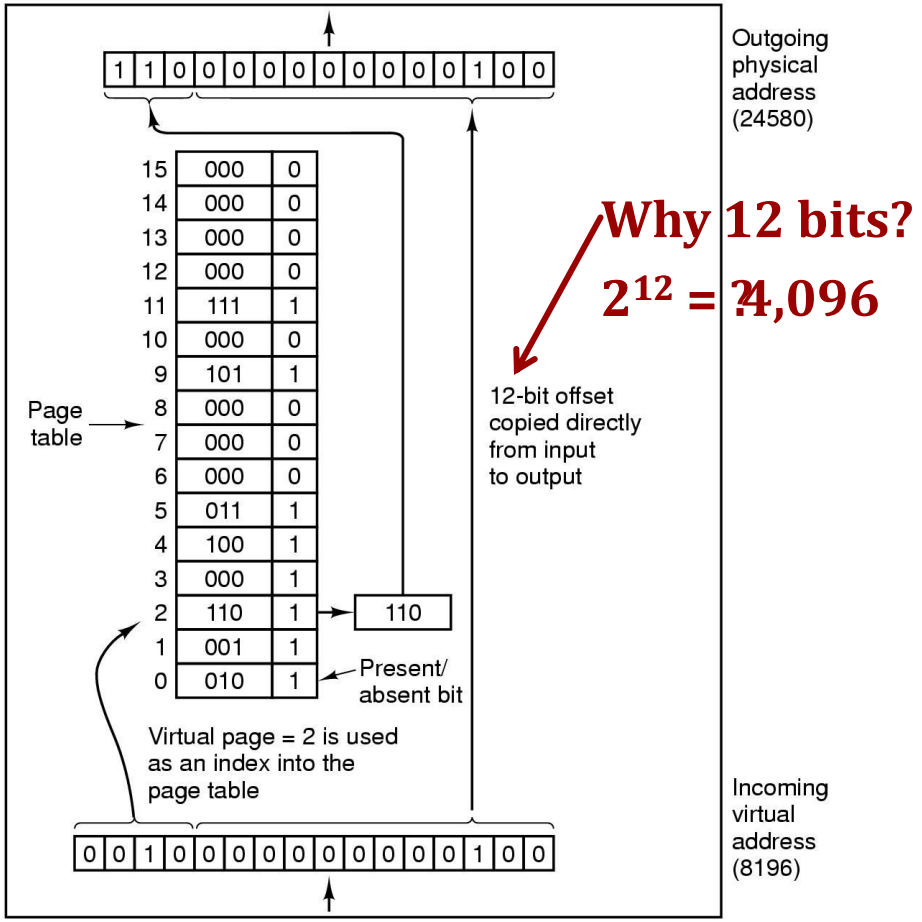
- **Minor page fault** – can be serviced by just creating the right mapping
- **Major page fault** – must load in a page from disk to service
- **Segmentation fault** – invalid address accessed; can't service so we usually just kill the program

Page Tables

- The MMU has to maintain information about the virtual -> physical mapping
- In the simplest case, this could be a simple array that stores the physical page number for each virtual page number
- The virtual address would then be split into two parts: an index into the mapping table, and then the offset within the page

Memory Management Unit

- Addresses are split into a page number and an offset
- Page numbers are used to look up a table in the MMU with as many entries as the number of virtual pages
- Each entry in the table contains a bit that states if the virtual page is actually mapped to a physical one
- If it is so, the entry contains the number of physical page used
- If not, a *page fault* is generated and the OS has to deal with it

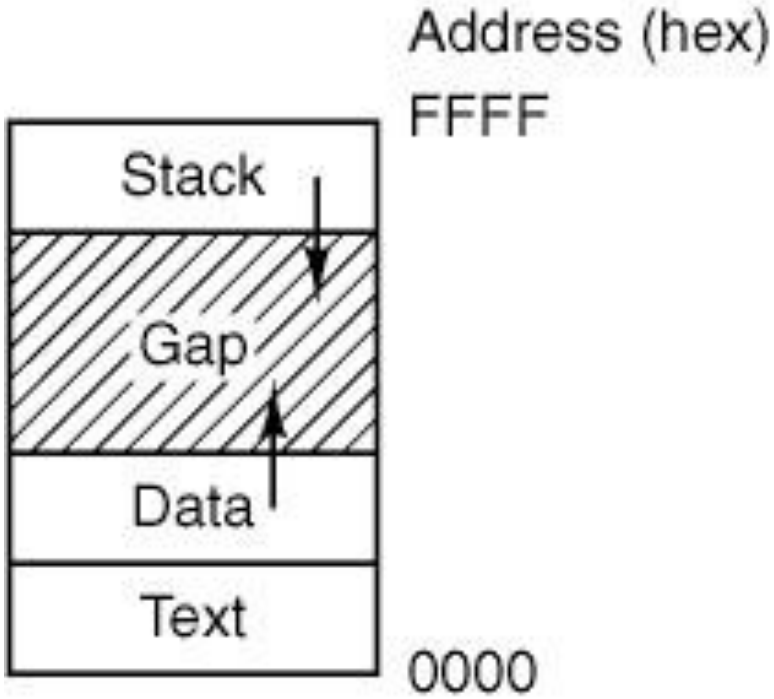


Page Tables

- Page tables contain one entry for each virtual page
- If virtual memory is big (e.g., 32 bit and 64 bit addresses) the table can become of unmanageable size
- Solution: instead of keeping them in the MMU move them to main memory
- Problem:
  - Page tables are used each time an access to memory is performed. Adding a level of indirection, may kill performance
  - What if memory is only sparsely used (e.g., used-mem << address space == 4GB)

Single Level Page Tables

- Contains one entry for each page frame
- Virtual address space is designed to be far larger than ever required (32 bits ;-))

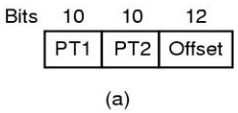


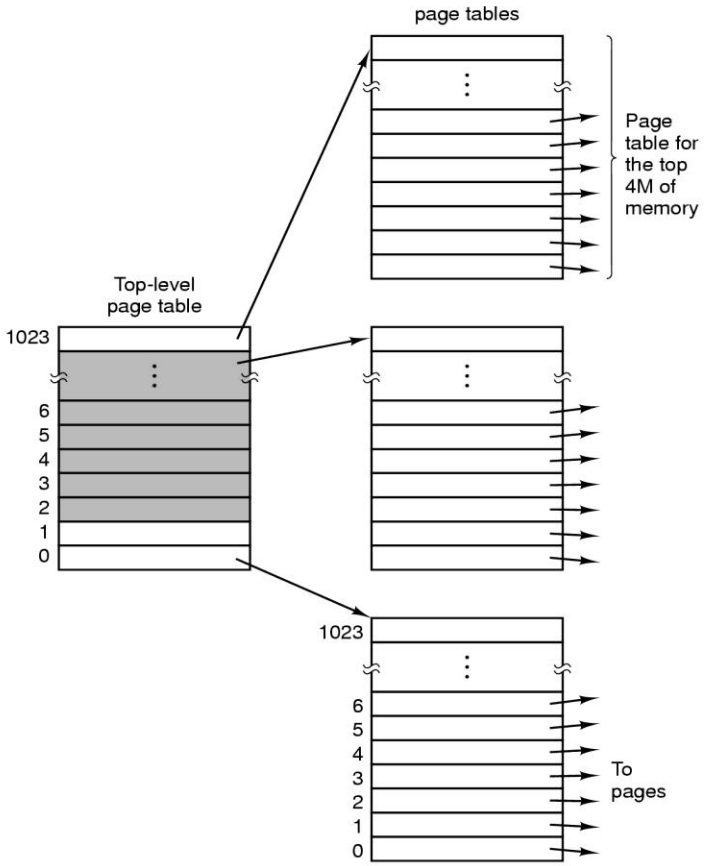
Real Page Tables

- Simple scheme is very inefficient if the virtual address space is expected to be sparse (i.e., not many mapped pages)
- Instead, multi-level page tables are used
  - The virtual address now has multiple indices
  - This allows us to only allocate tables for portions of the space that are used
- Page tables themselves are stored in memory!

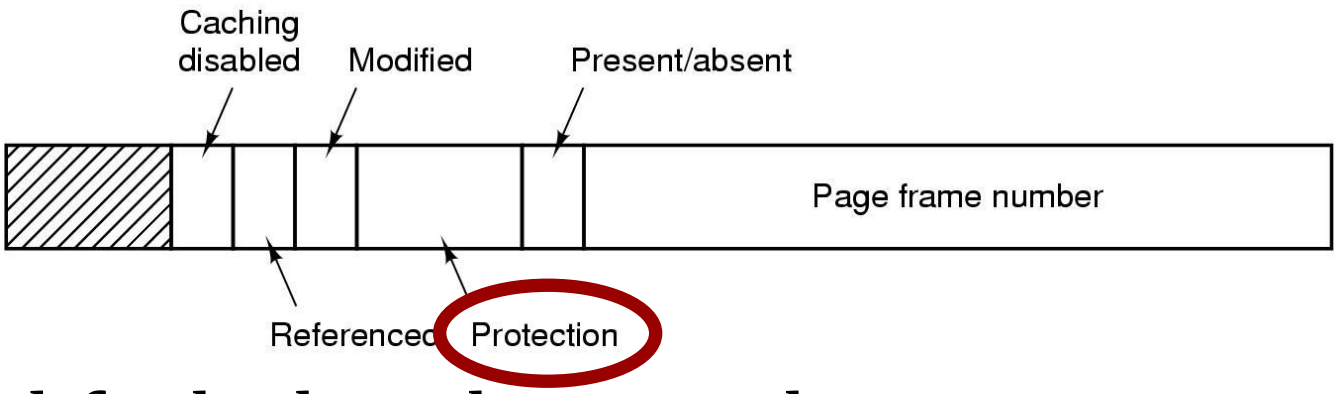
Multilevel Page Tables

- 32 bit virtual address
- PT1: Top-level index, 10 bits
- PT2: Second-level index, 10 bits
- Offset: 12 bits
- Page size: 4KB
- Second-level maps 4MB (1024 entries of 4KB)
- Top-level maps 4GB (1024 entries of 4MB)





Page Table Entries (PTEs)



- Modified – has this page been written to?
  - If so, we will need to write to disk before evicting
- Referenced – has anyone used this page?
- Caching disabled – used if physical page is used for device I/O

Protection

- Because the OS can give processes different virtual address spaces, we have already solved the problem of *isolation*
- But we may want to protect process from themselves in some cases:
  - Detect programmer errors before they do damage
  - Prevent attacks that exploit software vulnerabilities

Protection

Simplest protection is to mark pages as read-only or read/write

- Now, if someone attempts to modify read-only code or data, a page fault will occur

Some processors (in x86-land, starting with the AMD64 in 2003) have a bit to prevent code from being executed on a certain page

- This has been called variously the NX bit, the XD bit, Data Execution Prevention (DEP)
- The idea is to prevent buffer overflows from being exploitable – the attacker won't be able to run his own code because it will be in a data region

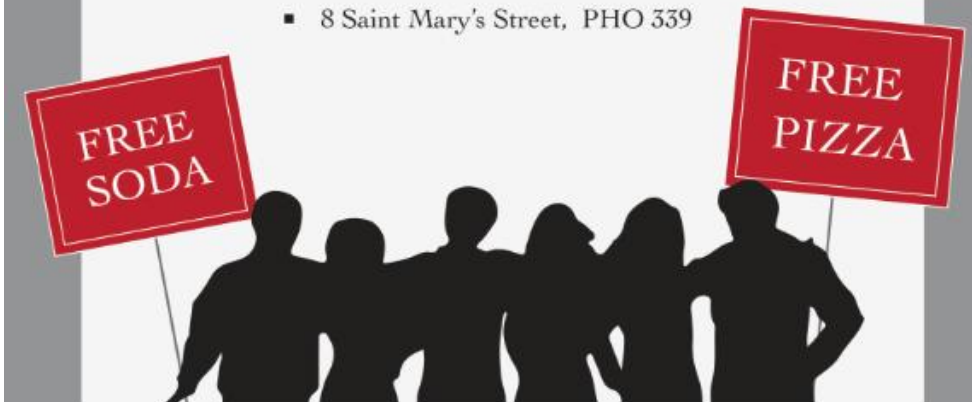
BU Department of Electrical & Computer Engineering

Voice Your Opinion on ECE Undergraduate Programs

ECE TOWN HALL

The **ECE Undergraduate Town Hall Meeting** is a chance for students to candidly discuss academics with department administration. The conversation will be led by ECE Associate Chair of Undergraduate Programs Tali Moreshet.

- Wednesday, November 2, 2016
- 6 pm - 7:30 pm
- 8 Saint Mary's Street, PHO 339



Midterm Results

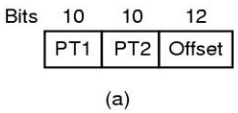
Name,	Points
terrier002,	21.5
terrier003,	39
terrier004,	41
terrier005,	23.5
terrier006,	24
terrier007,	32
terrier008,	31
terrier009,	14
terrier013,	34
terrier014,	39.5
terrier016,	29
terrier018,	24

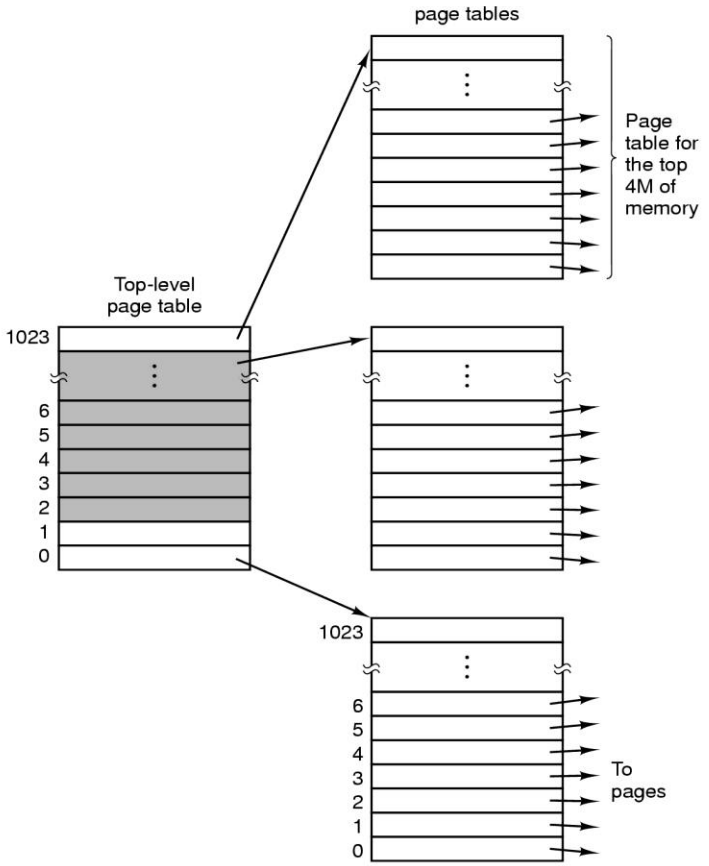
Name,	Points
terrier019,	30
terrier020,	30.5
terrier021,	28
terrier022,	37.5
terrier024,	27
terrier026,	29
terrier027,	25
terrier028,	30.5
terrier029,	28.5
terrier030,	36
terrier031,	25
terrier032,	36.5
terrier033,	44.5
terrier034,	39

Name,	Points
terrier036,	38.5
terrier037,	17.5
terrier038,	35.5
terrier039,	27.5
terrier041,	23
terrier042,	28
terrier044,	29.5
terrier045,	22.5
terrier046,	25.5
terrier047,	35
terrier050,	15.5
terrier051,	42
terrier052,	18

Multilevel Page Tables

- 32 bit virtual address
- PT1: Top-level index, 10 bits
- PT2: Second-level index, 10 bits
- Offset: 12 bits
- Page size: 4KB
- Second-level maps 4MB (1024 entries of 4KB)
- Top-level maps 4GB (1024 entries of 4MB)





Locality of Reference

- Multilevel tables can hold many pages but they still require multiple index lookups for each memory access
  - If we have to do 2 table lookups for every memory access, we've just made memory 3x slower
- Most programs use a subset of their memory pages (loops, sequential executions, updates to the same data structures, etc.) and the set changes slowly
  - *Locality of reference*
  - *Working set*
- The CPU keeps a small set of mappings that it can translate directly without consulting the page tables

Translation Look-aside Buffers

- Translation Look-aside Buffer (TLB)
  - hardware device that allows fast access without using the page table
- Small number of entries (e.g., 8) accessible as an associative memory
- Checked before doing a page table walk
- If lookup succeeds (hit), the page is accessed directly
- If TLB lookup fails (miss), the page table is used and the corresponding entry in TLB is added
- When an entry is taken out of the TLB, the modified bit is updated in the corresponding entry of the page table
- TLB management can be done both in hardware (MMU) or in software (by the OS)

Translation Look-aside Buffers

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 on x86

- Can look this up on Linux with the `x86info` command:

```
x86info –c
```

TLB on x86

- Can look this up on Linux with the `x86info` command:

```
Cache info
TLB info
 Instruction TLB: 4K pages, 4-way associative, 64 entries.
 Data TLB: 4KB or 4MB pages, fully associative, 32 entries.
 Data TLB: 4KB pages, 4-way associative, 64 entries
 Data TLB: 4K pages, 4-way associative, 512 entries.
```

- Note that the x86 keeps multiple separate TLBs for code vs. data as well as different page sizes

Software TLB Management

- In some architectures (e.g., SPARC, MIPS), the TLB is managed by software
- TLB entries are explicitly loaded by the OS
- If we look up an address and it's not in the TLB (a *TLB miss*) we raise a TLB fault
- The OS then has to fill in the missing TLB entry

Software TLB Management

- Why manage the TLB explicitly instead of letting hardware do it?
  - The MMU can be much simpler, which saves space on the CPU that can be used for other things
  - Flexibility – the OS can choose its own algorithms for which TLB entries to evict and add
- But: this is generally slower than hardware-managed TLBs

It's a tradeoff!

TLBs and Context Switching

- TLBs map a virtual page to a physical page frame
- But, once we change to a new process, these mappings are no longer valid, and a *TLB flush* occurs
- This makes context switching more expensive – the first few memory accesses a process makes will have to be serviced by walking the page tables (i.e., 3x slower memory)

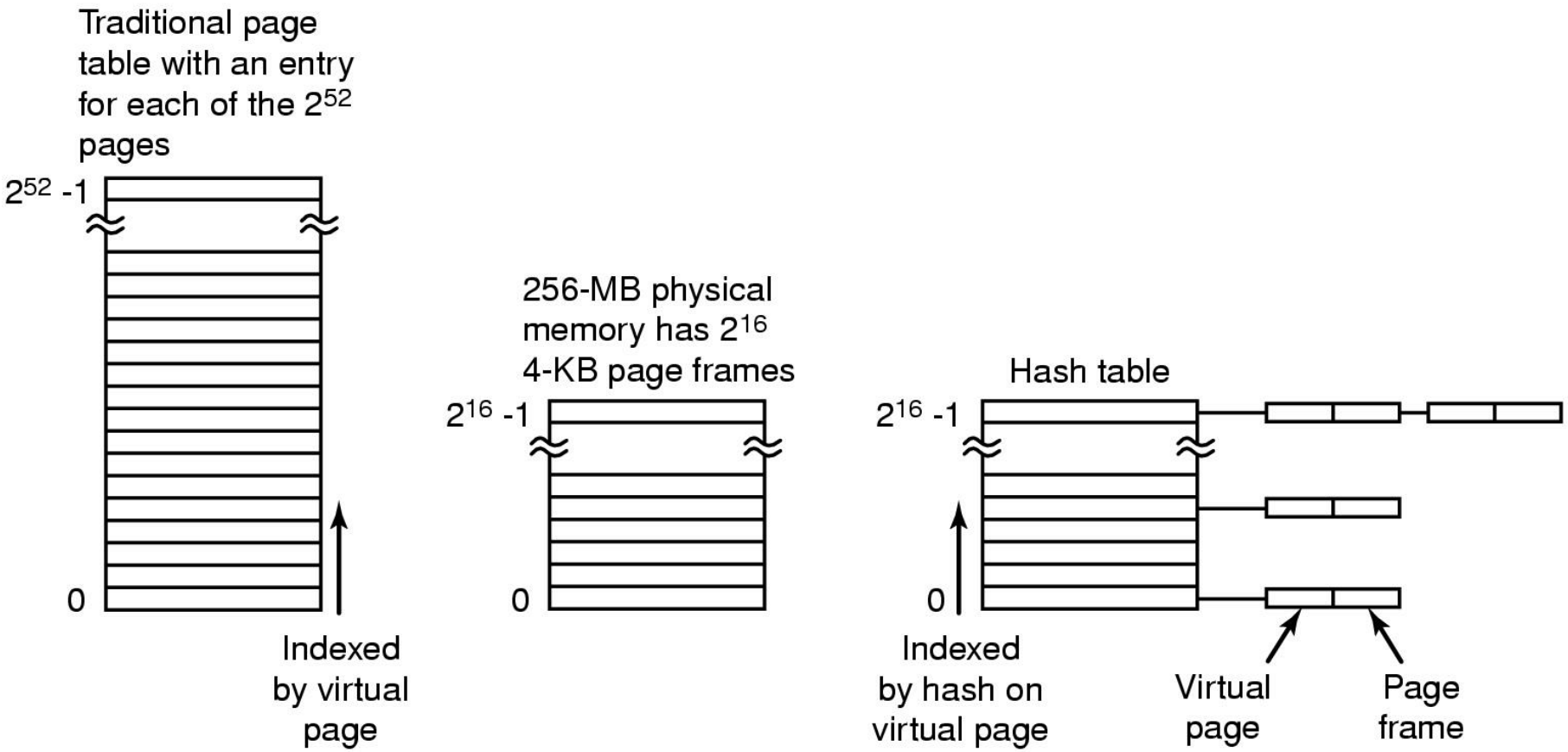
Tagged TLBs

- On some architectures (e.g., ARM), each TLB entry can be associated with a *tag* that says what address space it belongs to
- Now we don't have to flush the TLB when switching address spaces
- This can help make context switching faster – some TLB entries might still be valid when we switch back to a process

Inverted Page Table

- When virtual pages are too many, maintaining a page table is not feasible
- Solution: Inverted Page Table
  - One entry per physical page frame
  - Each entry contains a pair <process, virtual page>
- Address cannot be resolved simply by looking for an index in a table
- When process n accesses page p, the table must be scanned for an entry <n, p>
- Solution
  - TLB should catch most of the accesses
  - Table hashed on virtual address to resolve the mapping

Inverted Page Table



Page Replacement Algorithms

***Page fault* forces choice**

- Which page should be removed to make room for an incoming page?

Criteria:

- Modified page must first be saved
- Unmodified just overwritten
- Better not to choose an often used page that will probably need to be brought back in soon

Thrashing:

- Constantly swapping in and out memory pages
- Large negative impact on system performance
- i.e., more time is spent on moving data from memory to disk (and back) than making progress in the program

R and M Bits

- Each page has a Reference (R) bit and a Modified (M) bit

- R and M bits can be provided by the hardware or can be managed in software
  - Initial process with no pages in memory
  - When page is loaded
    - R bit is set
    - READONLY mode is set
  - When modification is attempted a fault is generated
    - M bit is set
    - READ/WRITE mode is set

Optimal Page Replacement Algorithm

- Determine how far in the execution of the program a page will be hit
- Replace page needed at the farthest point in the future
  - Optimal but unrealizable
- Useful to compare with other algorithms
  - Log page use on first execution of the program
  - Develop a scheduling for following executions (with same inputs)

"*Prediction is very difficult, especially about the future*"
Niels Bohr

Not Recently Used (NRU) Page Replacement Algorithm

- When process starts, R and M bit are set to 0
- Periodically R bit is cleared to take into account for pages that have not been referenced recently
- Pages are classified according to R and M
  - Class 0: not referenced, not modified
  - Class 1: not referenced, modified (R bit has been cleared!)
  - Class 2: referenced, not modified
  - Class 3: referenced, modified
- NRU removes page at random from lowest numbered non empty class
- Easy to implement but not terribly effective

EC 440 – Introduction to Operating Systems

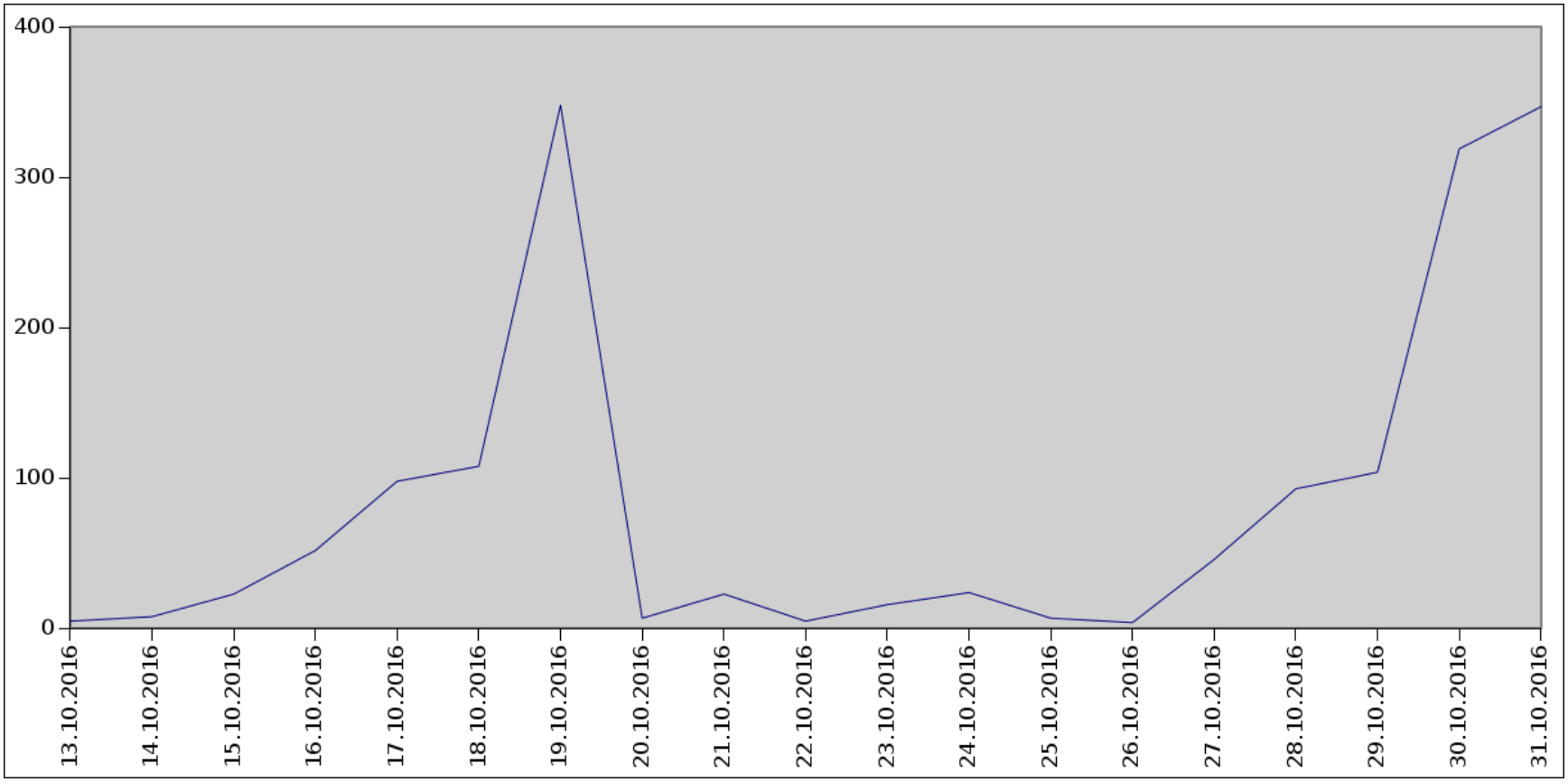
Manuel Egele

Department of Electrical &
Computer Engineering

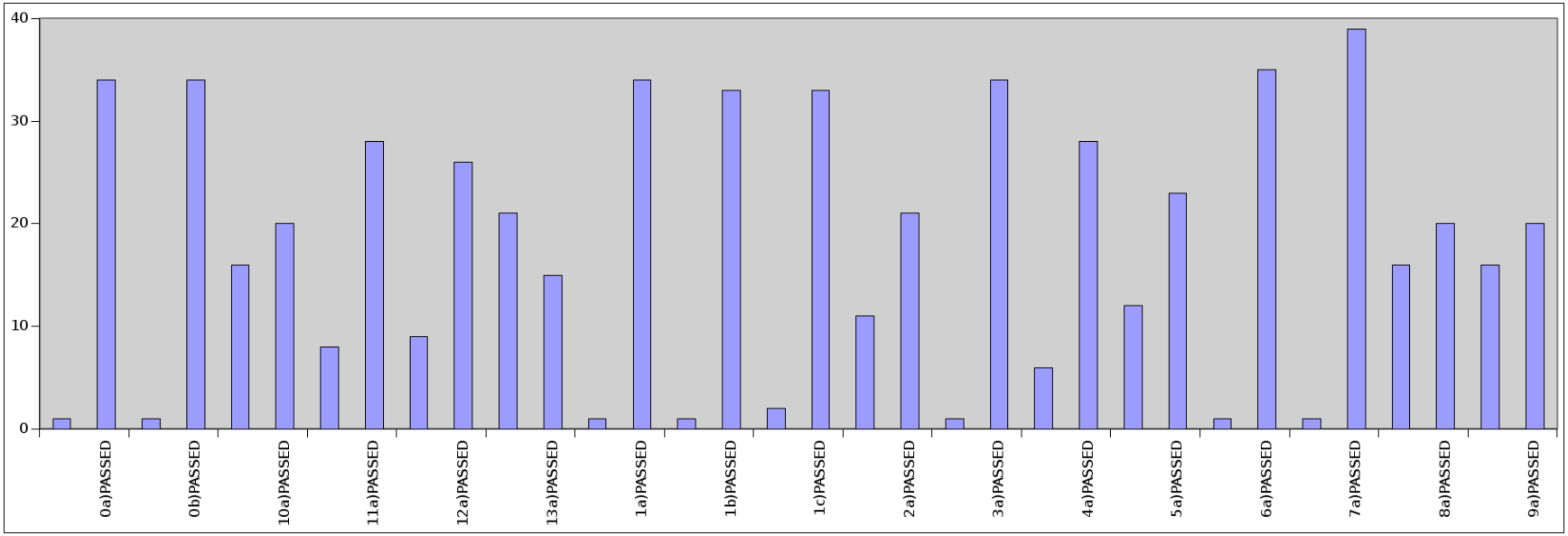
Boston University

Slides from Christopher Kruegel, UCSB

Project 3 – Stats – # of Submissions



Project 3 – Stats – Tests



Most failed tests: 13, 8, 9

Project 4

- Released Today
- Deadline: Nov. 17th 18:00 EST
- Thread local storage (TLS)
- Concepts you need:
  - Memory protection (R,W bits)
  - Copy on write
  - Signals (esp. segfault)
  - Working knowledge of threading library
    Either use your own or use libpthread

Project 4

- Tests made by the submission bot are not authoritative
- They merely exist for your convenience
- ***Do not*** program against the tests! Instead, implement the specification.
  - i.e., if your solution passes the test but does not implement the specification, it will not be counted as correct

First-in First-out (FIFO) Page Replacement Algorithm

- Maintain a linked list of all pages
- List is ordered by loading time
- The page at the beginning is replaced
  - Oldest one

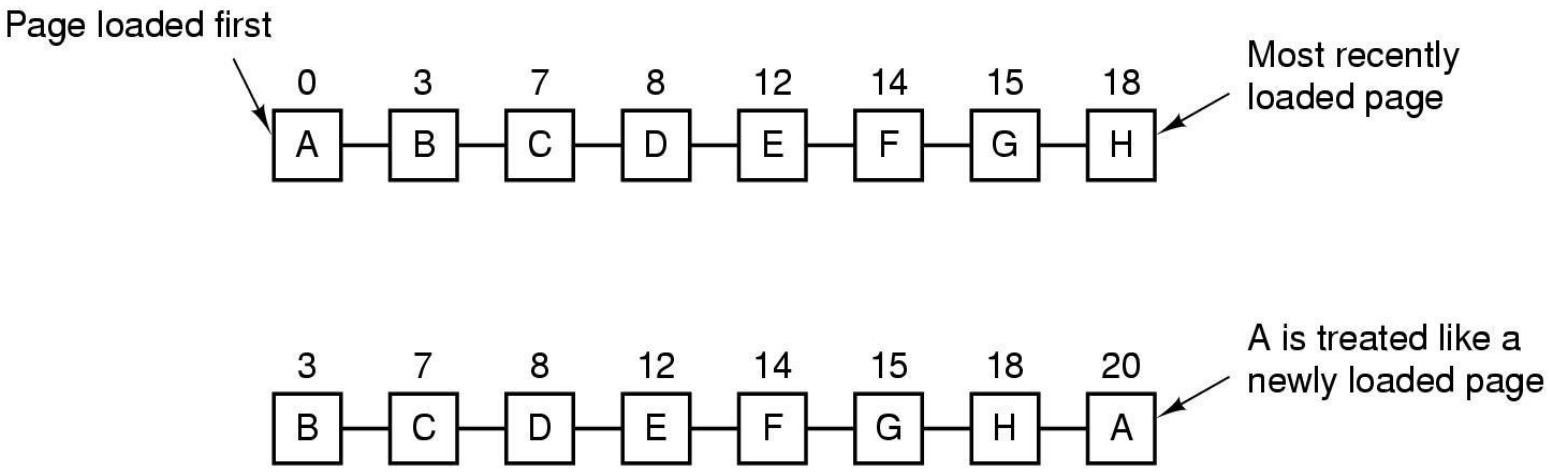
- Advantage
  - Easy to manage
- Disadvantage
  - Page in memory the longest may be often used

Second Chance Page Replacement Algorithm

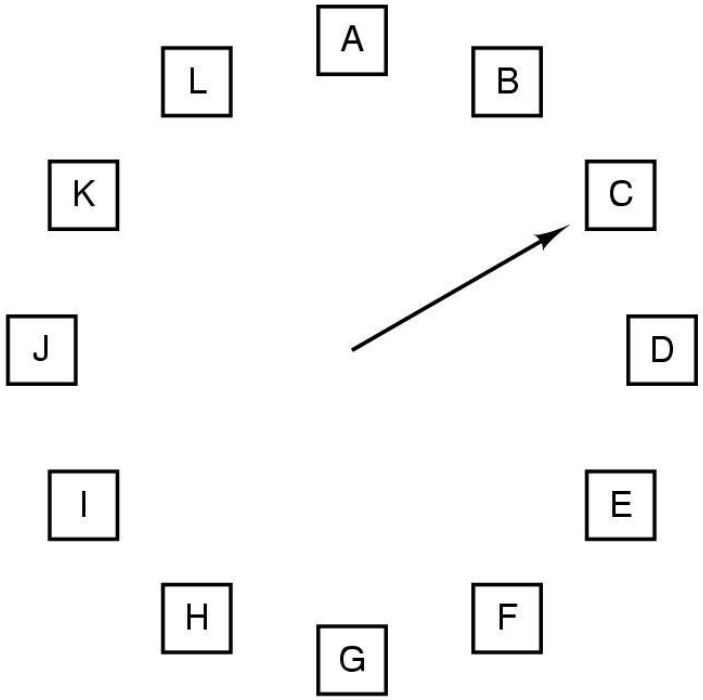
- Pages in list are sorted in FIFO order
- R bits are cleared regularly
- If the R bit of the oldest page is set it is put at the end of the list
- If all the pages in the list have been referenced the page that was “recycled” will reappear with the R bit cleared and will be thrown away

Second Chance Page Replacement Algorithm

Page list if fault occurs at time 20, A has R bit set



Clock
Page Replacement Algorithm



- Same concept as Second Chance, different implementation
- Hand points to oldest page
- When a page fault occurs
  - If R=0: evict the page
  - If R=1: clear R and advance hand until page with R=0 is encountered

Least Recently Used (LRU) Page Replacement Algorithm

- Assumption: pages used recently will be used again soon
  - Throw out page that has been unused for longest time

Problem?

- Very expensive: Must keep a linked list of pages
  - Most recently used at front, least at rear
  - Update this list every memory reference !

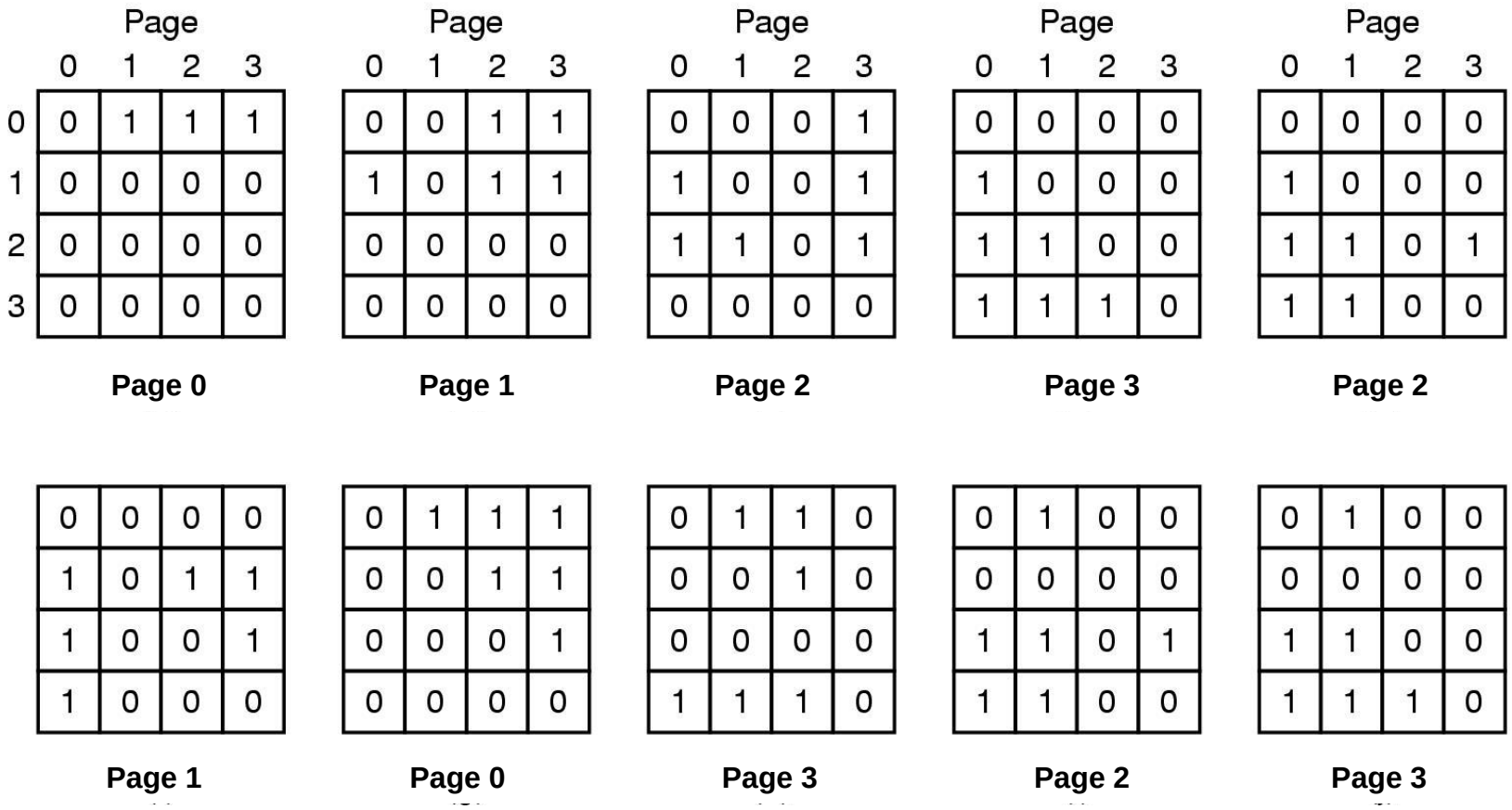
- Alternative
  - Maintain global counter that is incremented at each instruction execution
  - Maintain similar counter in each page table entry
  - If page is referenced copy global counter in page counter
  - Choose page with lowest value counter

Least Recently Used (LRU) Page Replacement Algorithm

Another alternative:

- Maintain matrix *M* with *n* x *n* bits, where *n* is the number of page frames
- If page *j* is accessed
  - set to 1 all the bits in the corresponding row ($M_{j,i} = 1, i = 1 \ldots n$)
  - set to 0 all the bits in the corresponding column($M_{i,j} = 0, i = 1 \ldots n$)
- At any moment the page with the row containing the lowest value is the least recently used

Least Recently Used (LRU) Page Replacement Algorithm

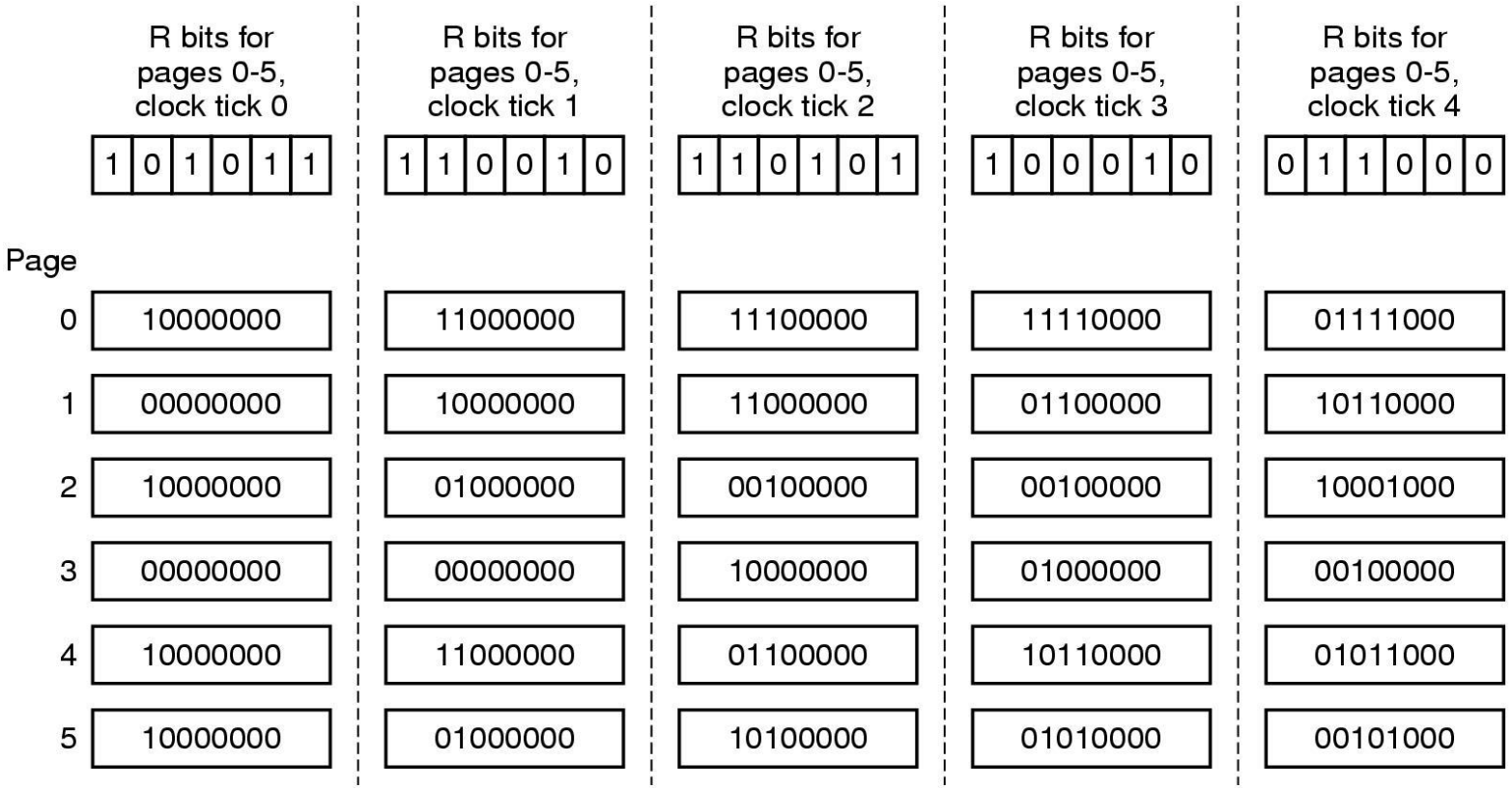


Not Frequently Used (NFU) Page Replacement Algorithm

- A counter is associated with each page
- At each clock interval, the counter is incremented if the page has been referenced (R=1)
- The page with the lowest counter is removed
- Problem:

  pages that have been heavily used in the past will always maintain high counter values
- Need for an aging mechanism
  - First shift the counter
  - Then set bit in most significant position if referenced

Not Frequently Used (NFU) Page Replacement Algorithm

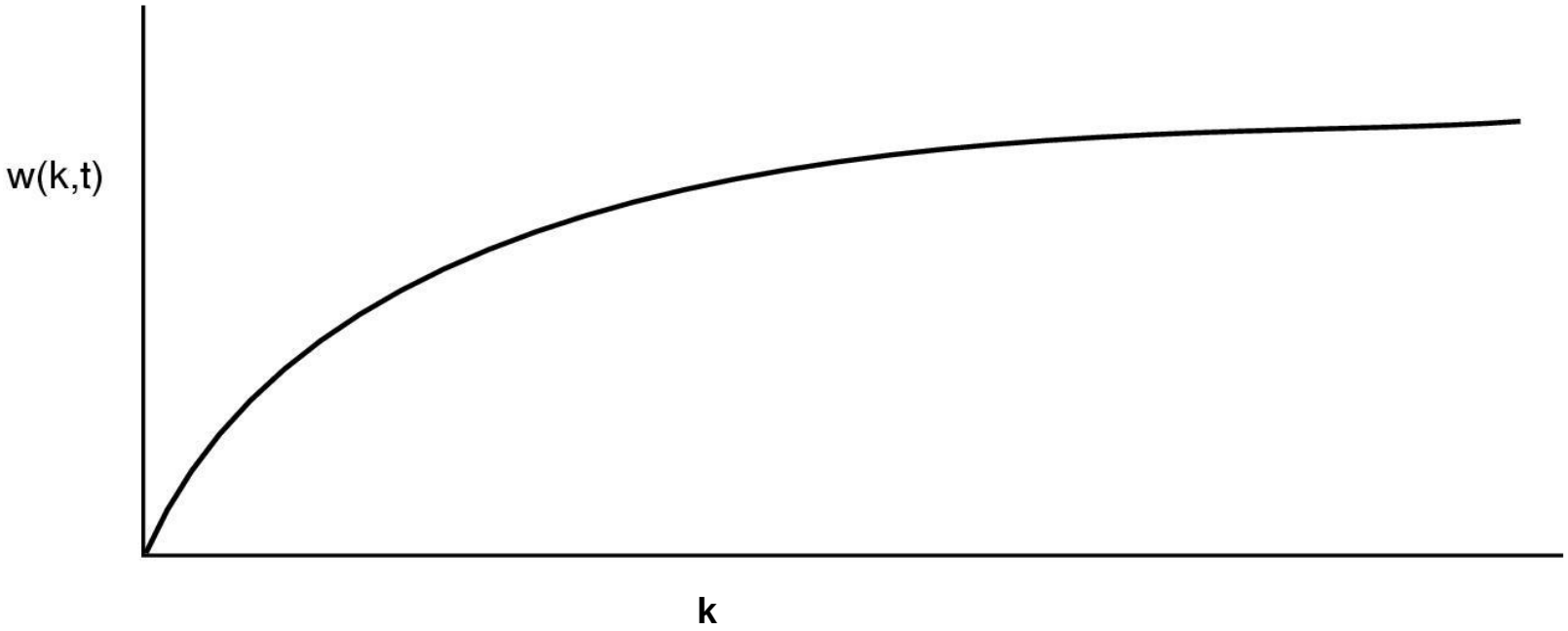


Working Set Page Replacement Algorithm

- Locality of reference: Most programs use a subset of their memory pages (loops, sequential executions, updates to the same data structures, etc.) and the set changes slowly
- The working set is the set of pages used by the k most recent memory references
- For a reasonable value of k, the number of page faults is reduced and the process does not *thrash*
- If the working set can be determined it can be preloaded at context switch to minimize the initial demand of pages

Working Set
Page Replacement Algorithm

w(k,t): the size of the working set at time, t



Working Set Page Replacement Algorithm

- Algorithm:

  when a page has to be evicted, find one that is not in the working set

- Use a shift register of size k

- At every reference
  - Right-shift the register
  - Insert page in left most position

- At replacement time
  - Remove duplicates and obtain working set
  - Remove page not in working set

- Problem: too heavy to maintain

Working Set Page Replacement Algorithm

- Use execution time instead of references
- Working set composed of pages referenced in the last t msec of execution
- Each entry contains
  - The time the page was last used
  - The reference bit, R

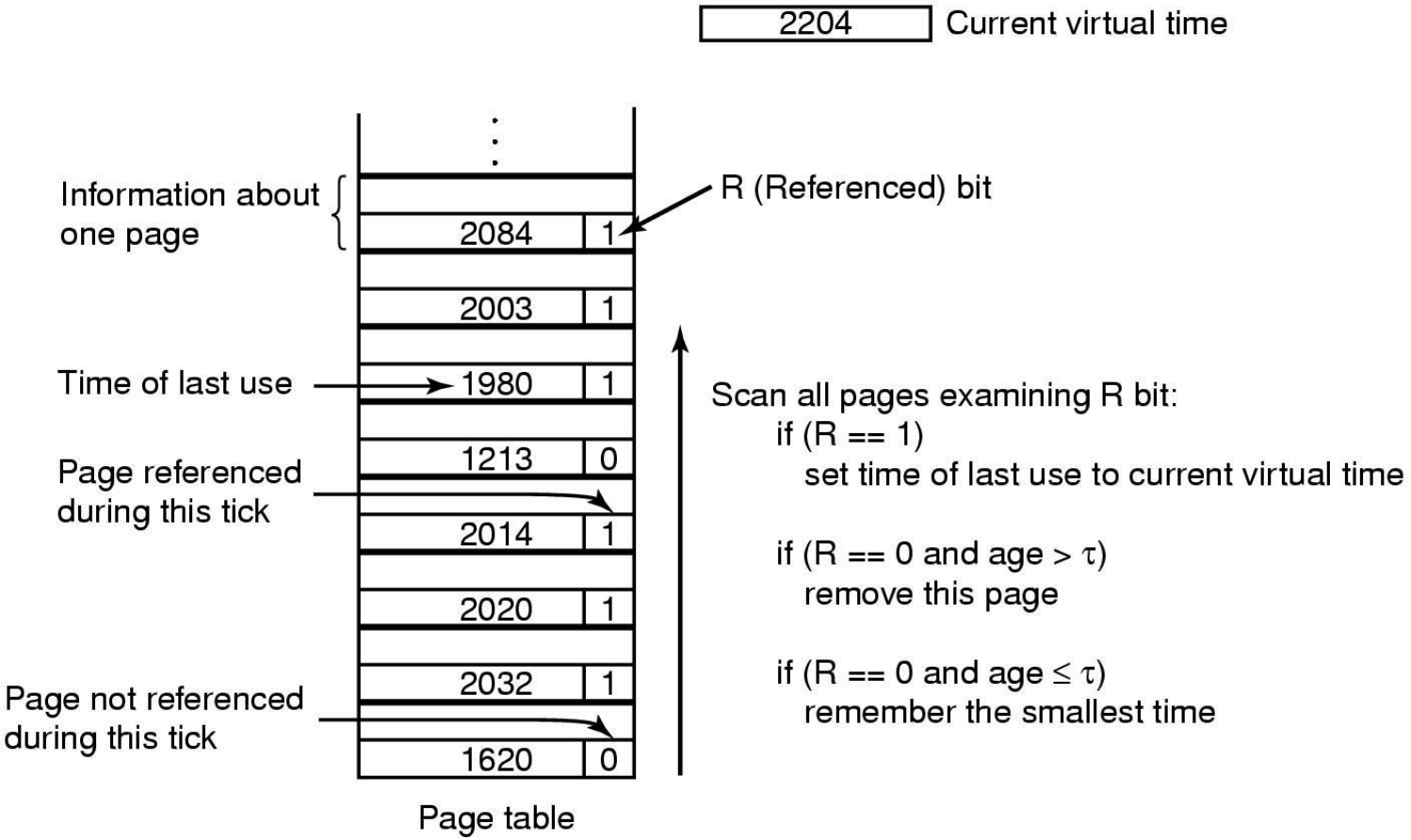
Working Set Page Replacement Algorithm

- At every page fault
  - If R=1 the current time is written in the page entry
  - If R=0
    - If the “age” (current time - time of last reference) is smaller than t, the page is spared (but the page with the highest age/smallest time of last usage in the working set is recorded)
    - If the “age” is greater than t, the page is a candidate
  - If there is one or more candidates, the candidate with highest age is evicted
  - If there are no candidates the oldest page in the working set is evicted

- Problem:

  whole page table must be scanned

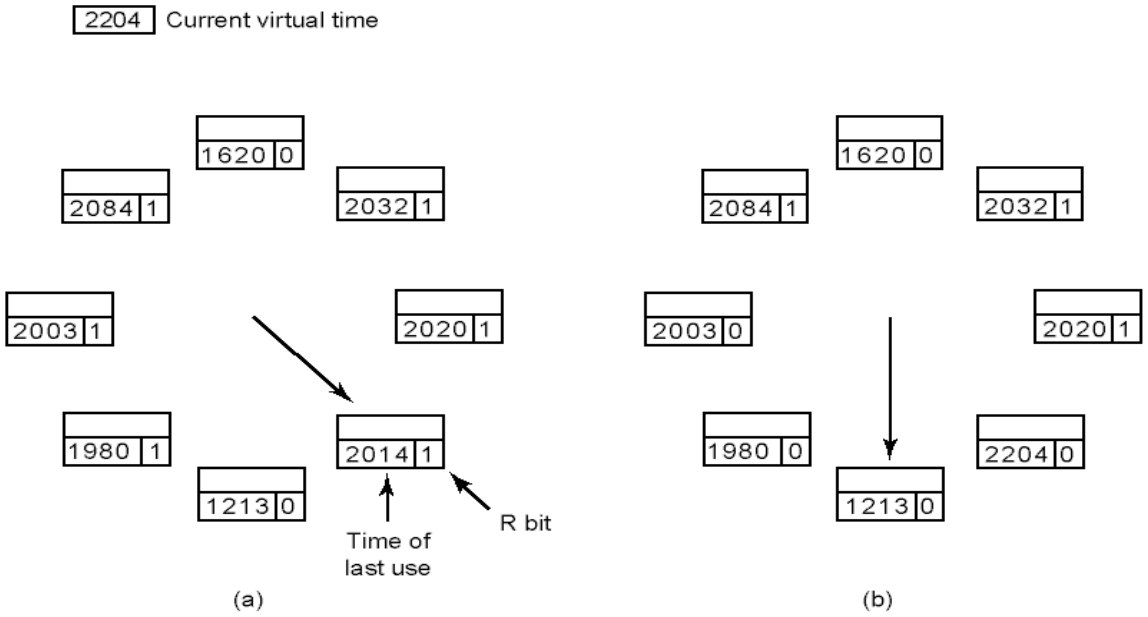
Working Set
Page Replacement Algorithm



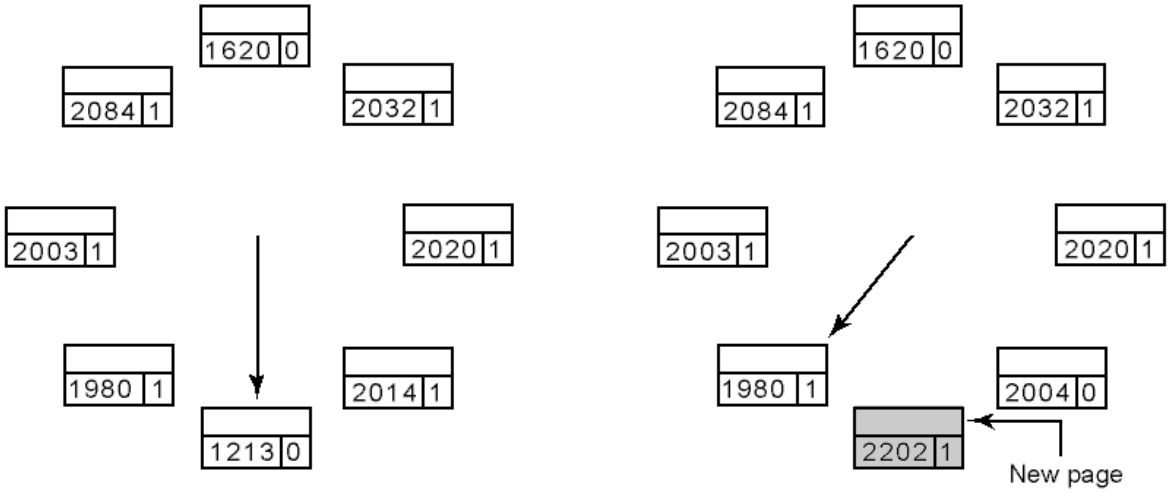
WSClock
Page Replacement Algorithm

- Every time a page is loaded is added to a circular list
- Each page is marked with the time of last use
- At page fault:
  - Examine page pointed by hand
    - If R=1: R is cleared, time is updated and hand advanced
    - If R=0:
      - If age is greater than t
        - » If page is clean (M=0) then evict
        - » If page is dirty (M=1) page is scheduled for writing to disk and hand advanced
      - If age is less than t the hand is advanced
  - If hand returns to initial position
    - If no writes are scheduled choose a random clean page
    - If writes have been scheduled continue until a clean, old page is found

WSClock
Page Replacement Algorithm



WSClock
Page Replacement Algorithm



Review of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Modeling Page Replacement Algorithms

- Program memory access is characterized as a string of referenced page numbers, called *reference string*
- Memory has n virtual pages and $m < n$ page frames (what if $m \geq n$?)
- Memory is modeled as an array M divided in two portions:
  - Top m rows are the actual mapping
  - Bottom n - m rows represent swapped pages
- As the reference string is examined
  - the top portion is checked to see if the reference page is present
    - If not, a page fault is generated
  - In any case the chosen algorithm is used to determine the configuration of the next column

Example with LRU

- State of memory array, M, after each item in reference string is processed (n =8, m=4)

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	

EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical &
Computer Engineering

Boston University

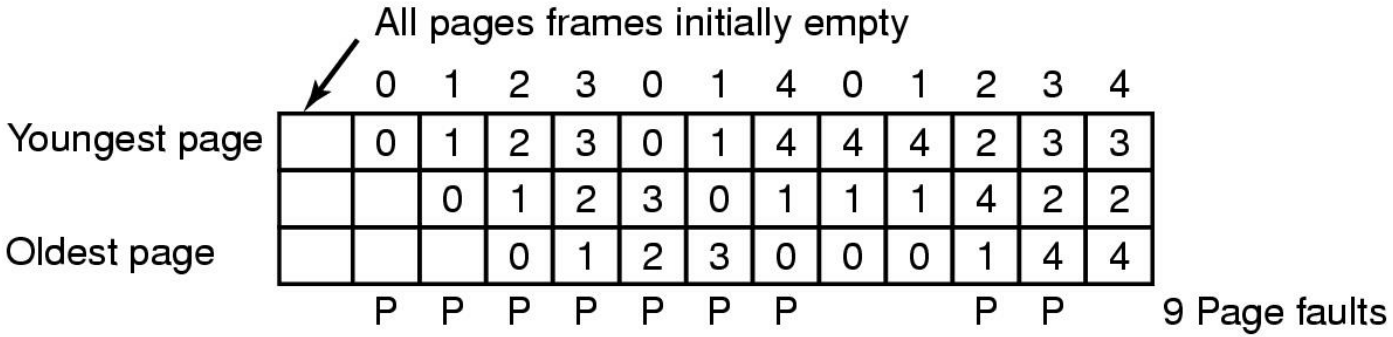
Slides from Christopher Kruegel, UCSB

Belady's Anomaly

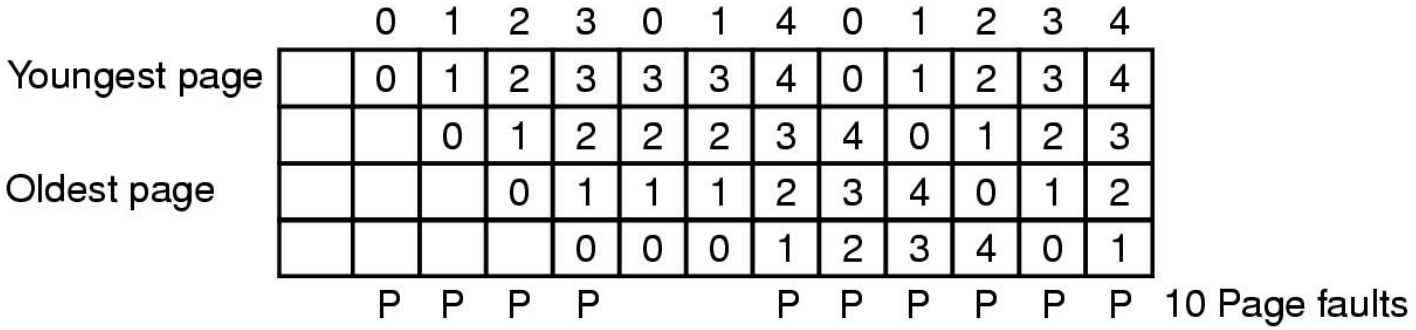
If you have more page frames
(i.e., more physical memory)
…
you'll have fewer page faults, right?!

Belady's Anomaly (e.g., in FIFO)

FIFO with 3 page frames = 9 page faults!



FIFO with 4 page frames = 10 page faults!



Stack Algorithms

- Algorithms that satisfy

  M(m,r) is in M(m+1,r)

  are called stack algorithms

- Stack algorithms do not suffer from the Belady's anomaly

- This means: if the same reference string is run on two memories with frame pages *m* and *m+1* respectively, the set of pages loaded in memory in corresponding points in the reference string are one a subset of the other

- Violated at the seventh reference in previous example

Design Issues

- Local vs. global allocation policies
- Load control
- Page size and internal fragmentation
- Sharing pages
- Locking pages
- Separating policy and mechanism

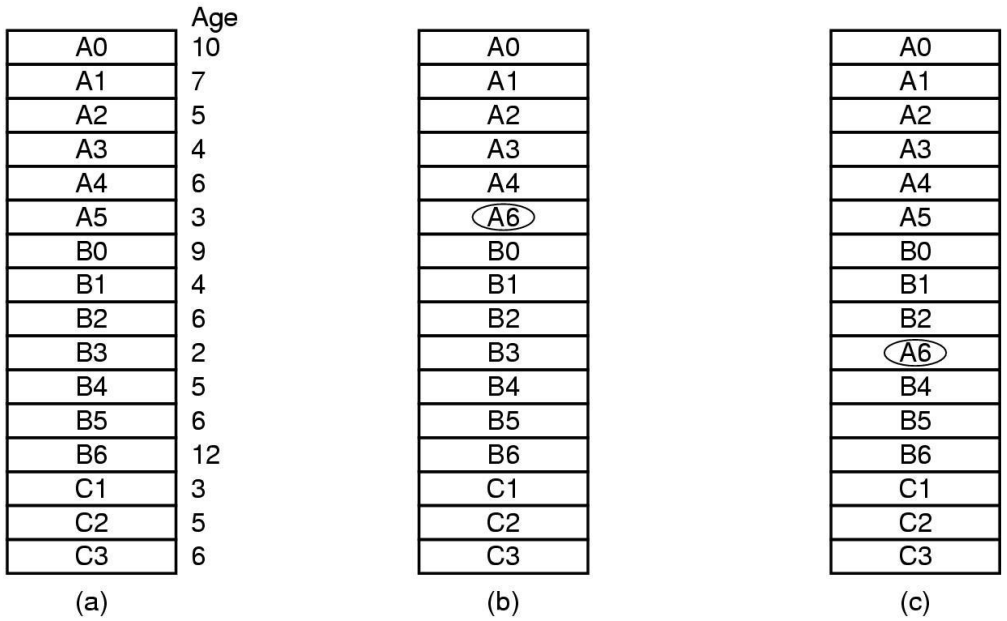
Local versus Global Allocation Policies

(a) Original configuration

(b) Local page replacement

(c) Global page replacement

Scenario: Bring in page 6 for process A (i.e., A6)



Load Control

- Despite good designs, system may still thrash
- The Page Fault Frequency algorithm uses the frequency of page faults to determine
  - Which processes need more memory
  - Which processes need less
- Reduce number of processes competing for memory
  - Swap one or more to disk, divide up pages they held
  - Reconsider degree of multiprogramming

Page Size

Small page size

- Advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
  - Less unused program in memory

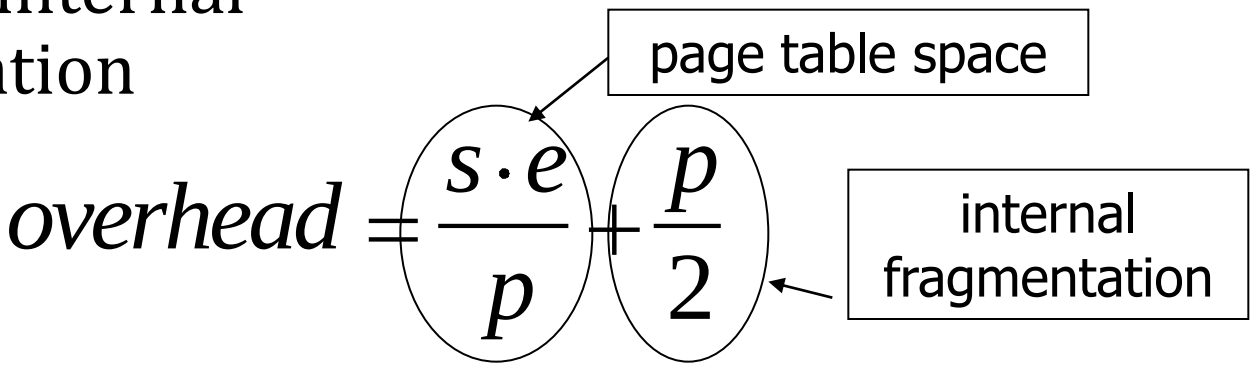
- Disadvantages
  - Programs need many pages, larger page tables

Page Table Overhead Calculations

- Worst case: single-level page table, 64-bit virtual address space, 4KB pages, 4byte PTEs
- $2^{64}/4096 * 4$ bytes = $2^{52} * 2^2 = 2^{54}$ bytes = 16 petabytes of memory used for pages
- And that's just for one process!

Page Size

- Overhead due to page table and internal fragmentation



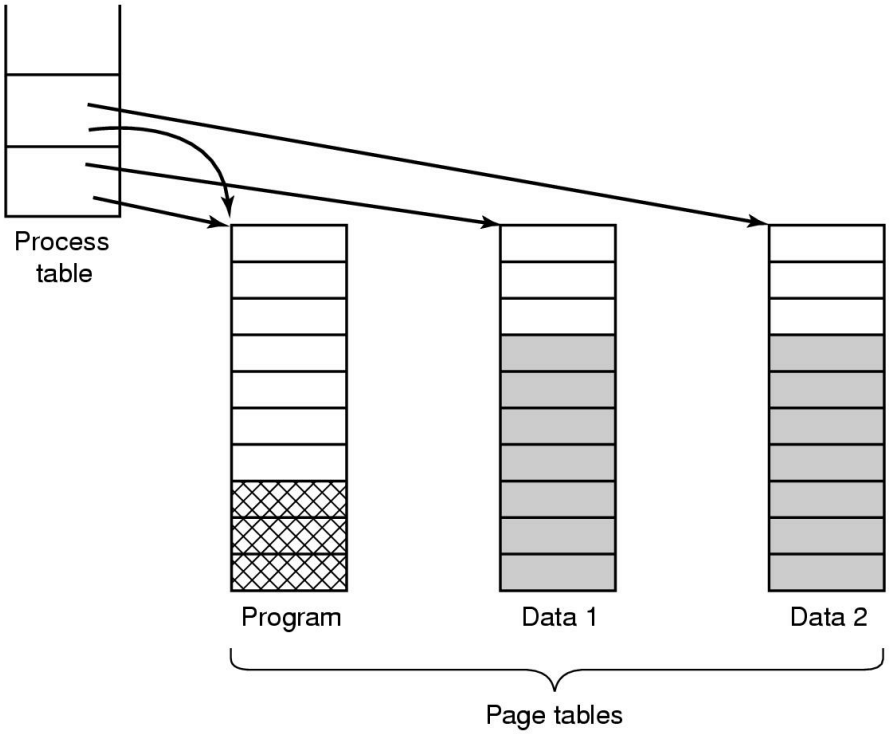
- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry

Optimized when

$$p = \sqrt{2se}$$

Shared Pages

Two processes sharing same program, sharing its page table

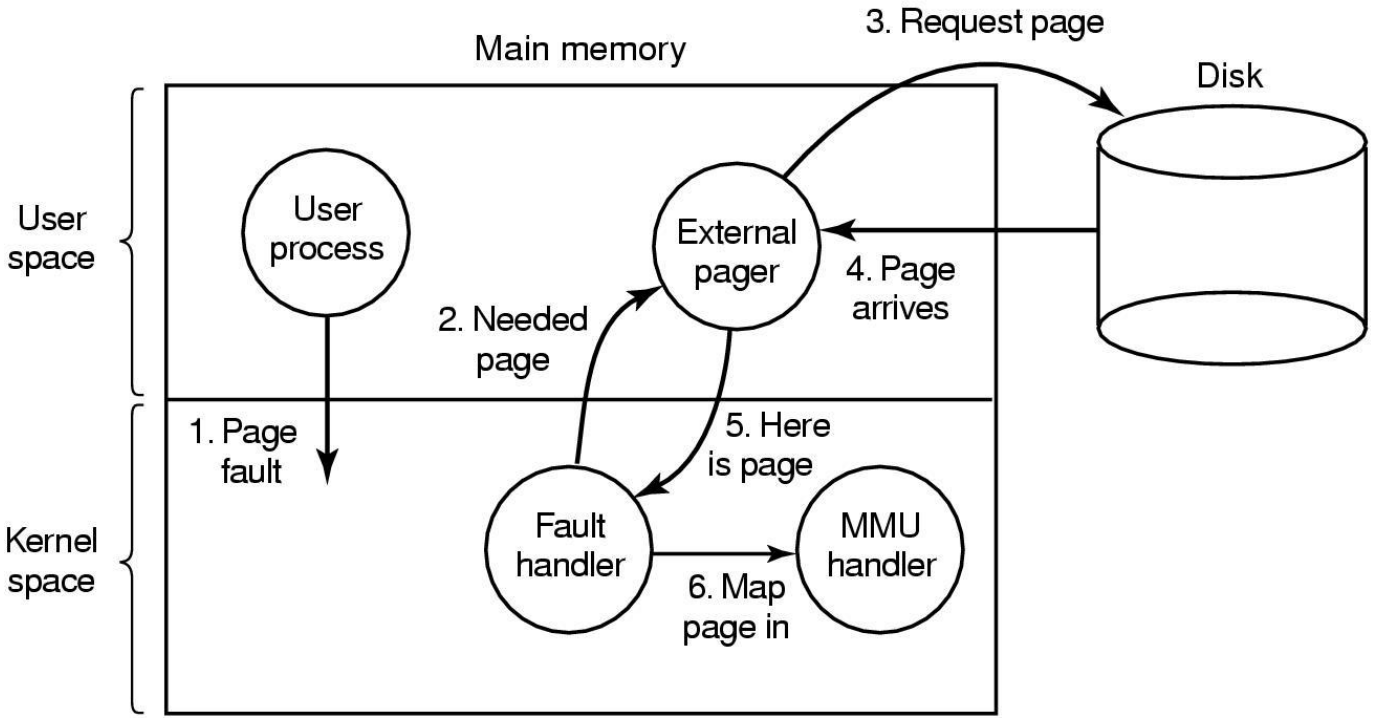


Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Process issues call for read from device into buffer
  - While waiting for I/O, another processes starts up
  - New process has a page fault
  - Buffer for the first process may be chosen to be paged out
- Need to specify some pages locked
  - Exempted from being target pages

Separation of Policy and Mechanism

Page fault handling with an external pager



Operating System Involvement

- Process creation
  - Determine program size
  - Create page table
  - Allocate swap
- Process execution
  - MMU reset for new process
  - TLB flushed
  - Make page table current
- Page fault time
  - Determine virtual address causing fault
  - Swap target page out, needed page in
- Process termination time
  - Release page table, pages

Virtual Memory in the Real World

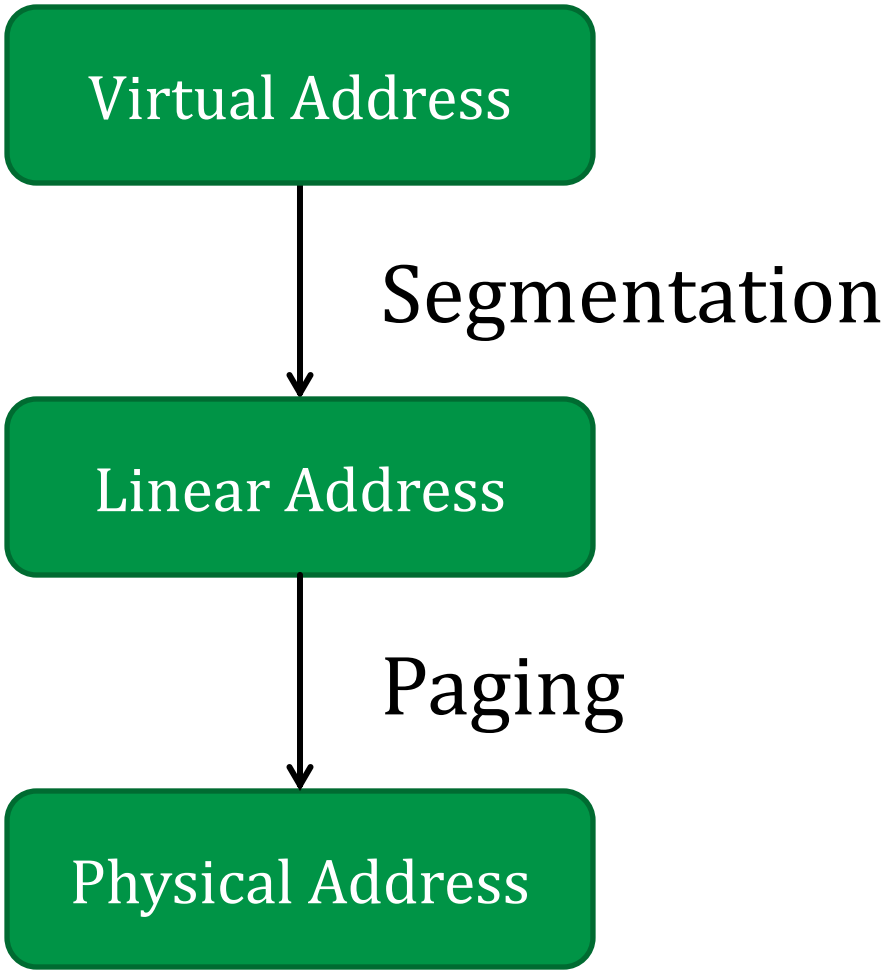
Virtual Address Translation in x86

- We will, for now, consider only:
  - 32-bit x86
  - 4KB and 4MB (“super”) pages
- So not:
  - 32-bit PAE (allows using more than 4GB of RAM on a 32bit machine)
  - 64-bit mode

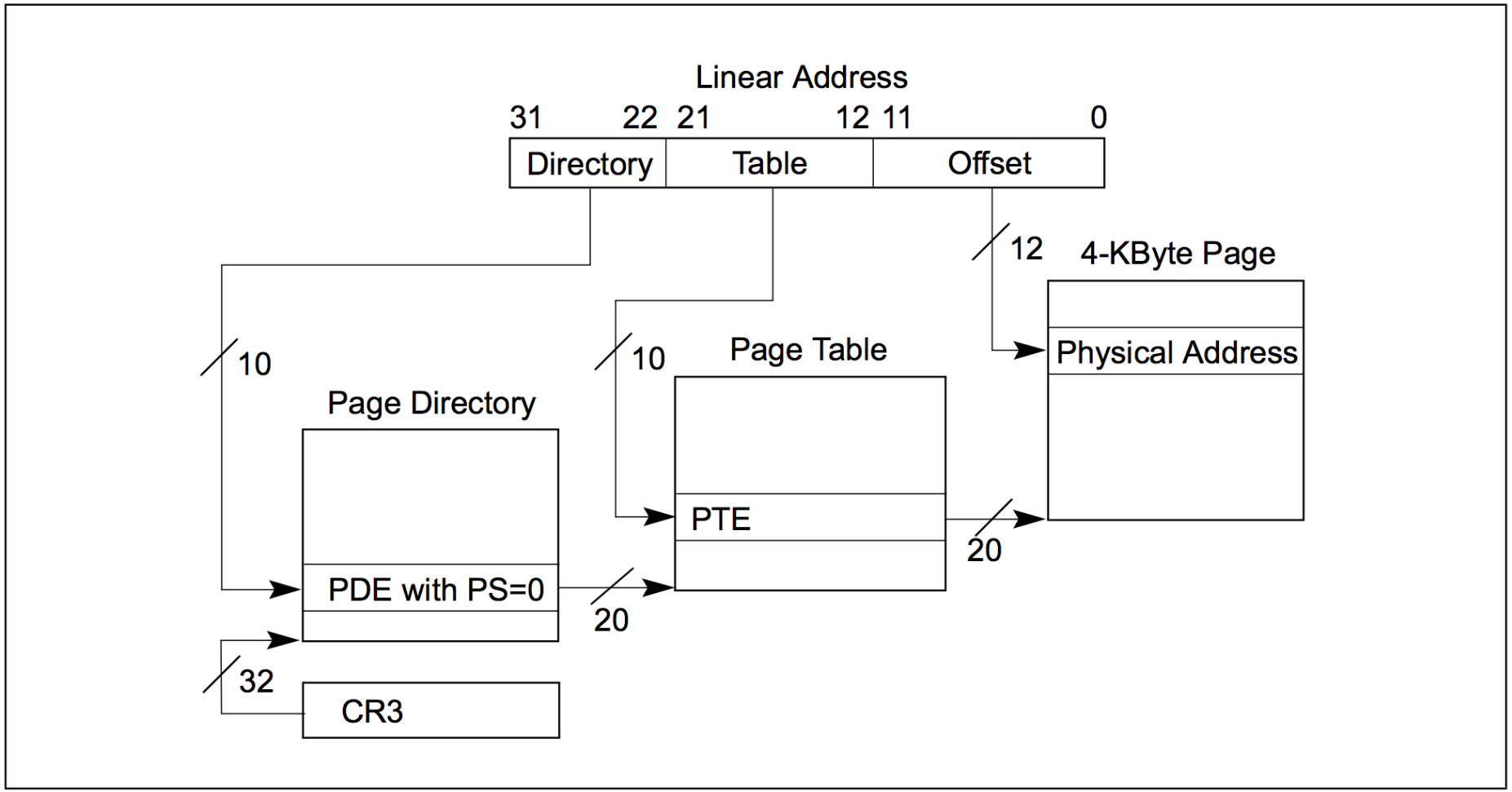
x86 Paging Basics

- x86 virtual address translation uses a two-level page table:
  - A top-level *page directory* stores pointers to the *page tables*
  - *Page tables* contain the actual *page table entries* (PTEs) referring to physical page frames
- The current mappings in use are determined by the value of the CR3 CPU register, which stores the *physical address* of the page directory
  - Each process has a different page directory
  - CR3 … control register 3 or page table base register

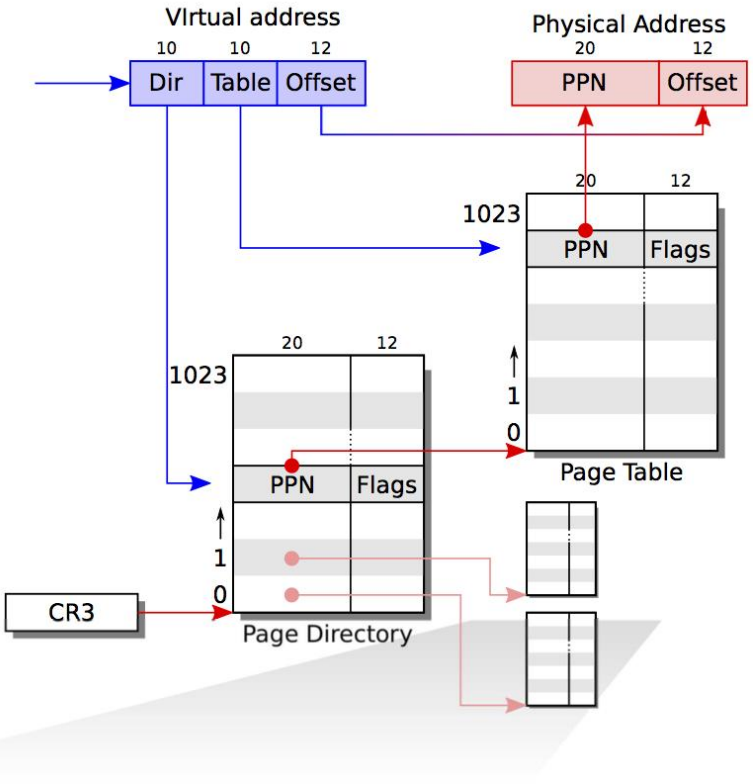
x86 Types of Addresses

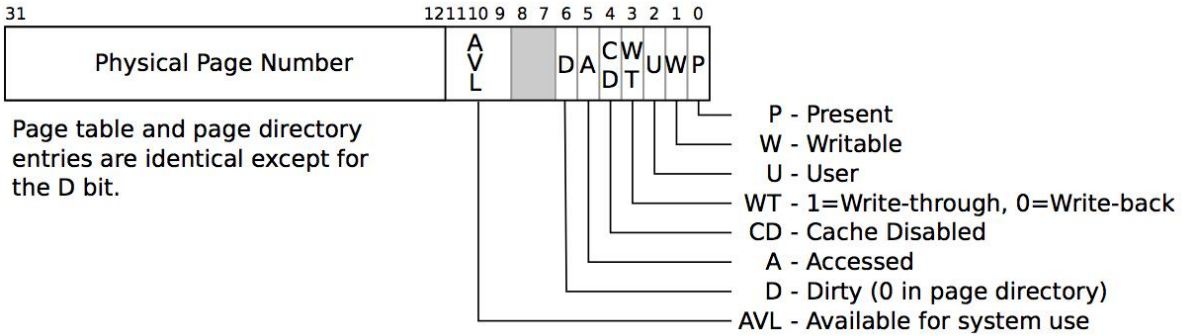


x86 Paging


Linear Address
31
22
21
12
11
0
Directory
Table
Offset
12
4-KByte Page
Physical Address
10
Page Table
10
Page Directory
PTE
20
PDE with PS=0
20
32
CR3

PDE/PTE


VIrtual address
10
10
12
Dir
Table
Offset
Physical Address
20
12
PPN
Offset
20
12
1023
PPN
Flags
1
0
Page Table
20
12
1023
PPN
Flags
1
0
CR3
Page Directory


31
12 11 10 9 8 7 6 5 4 3 2 1 0
Physical Page Number
AVL
D A CD WT U W P
Page table and page directory entries are identical except for the D bit.
P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

x86 PDEs and PTEs

- The entries in a page directory and a page table have an almost identical format
- Each 32-bit
- The basic structure:
  - Physical address of a page table (for PDEs) or page frame (for PTEs)
  - Protection, caching, etc. flags
  - A bit to indicate present/not present

x86 PDEs and PTEs

31–12	11–9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]	Ignored					PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame; Reserved (must be 0); Bits 39:32 of address[2]; PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table	Ignored		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored										0	PDE: not present
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

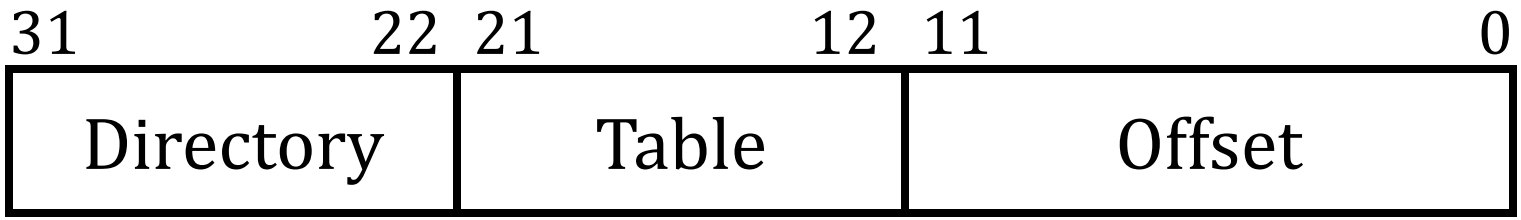
Working Example

- Let's take an arbitrary address from running a program: 0x8010c608
- In binary:

  `10000000000100001100011000001000`

- CR3 = 0x003ff000

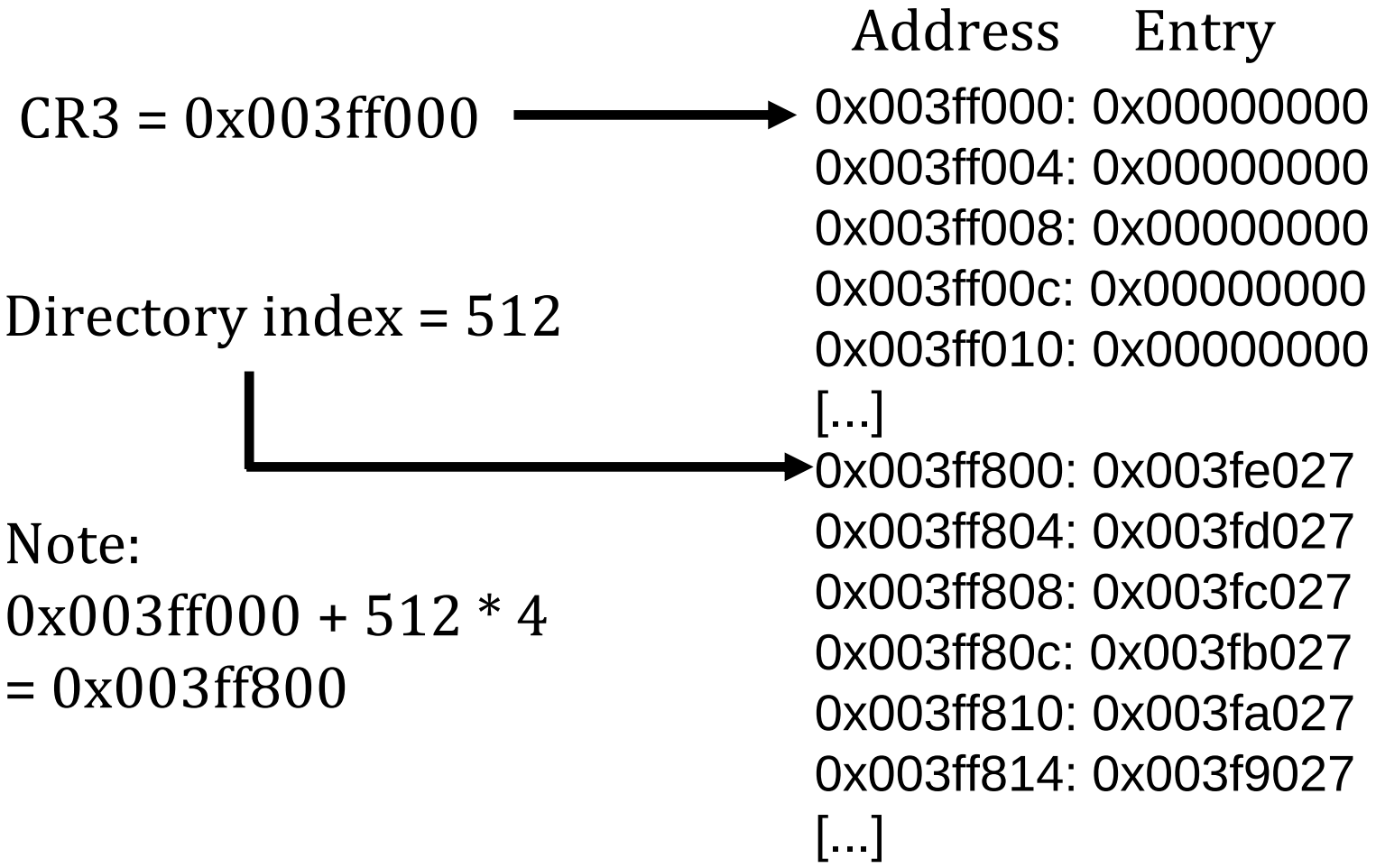
Working Example



`1000000000 0100001100 011000001000`

- Directory index = 512
- Table index = 268
- Offset = 1544

Page Directory



Page Directory Entry

0x003fe027

00000000001111111100000001001111

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table

Address of page table: 0x3fe000

(A) Accessed: Yes

(PCD) Cache disabled: No

(PWT) Write through caching: No

(U/S) User-accessible: Yes

(R/W) Read/Write: Yes

(P) Present: Yes

Page Table

Address of page table:
0x3fe0000

Table index = 268

Note:
0x003fe000 + 268 * 4
= 0x003fe430

Address	Entry
0x003fe000:	0x00000063
0x003fe004:	0x00001003
0x003fe008:	0x00002003
0x003fe00c:	0x00003003
0x003fe010:	0x00004003
[…]	
0x003fe430:	0x0010c063
0x003fe434:	0x0010d063
0x003fe438:	0x0010e063
0x003fe43c:	0x0010f063
0x003fe440:	0x00110063
0x003fe440:	0x00111063
[…]	

Page Table Entry

0x0010c063

00000000000100001100000011000011

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page

Address of physical page frame: 0x10c000

(G) Global: No

(PAT) PAT page: No

(D) Dirty: Yes

(A) Accessed: Yes

(PCD) Page Cache Disable: No

(PWT) Page Write Through: No

(U/S) User-accessible: No

(R/W) Read/Write: Yes

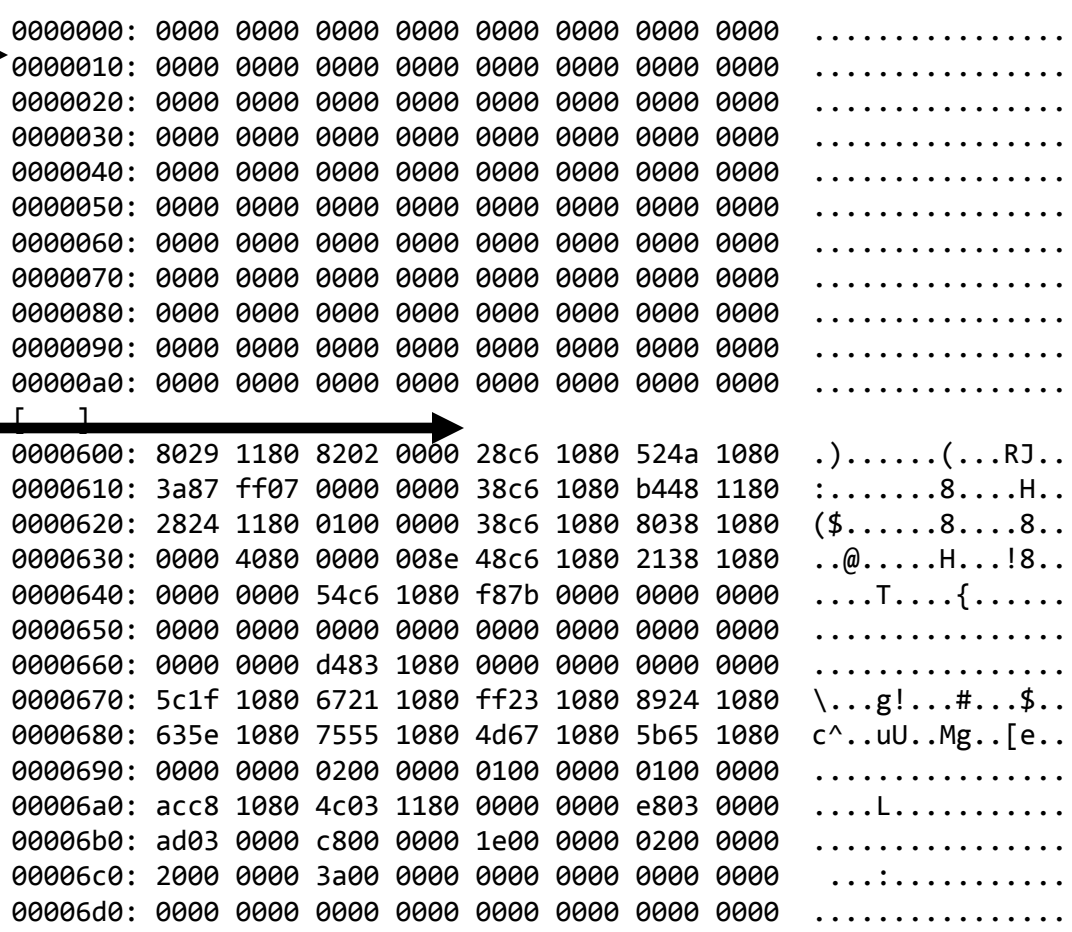
(P) Present: Yes

Physical Page

Address of page frame: 0x10c000

Offset = 1544 = 0x608

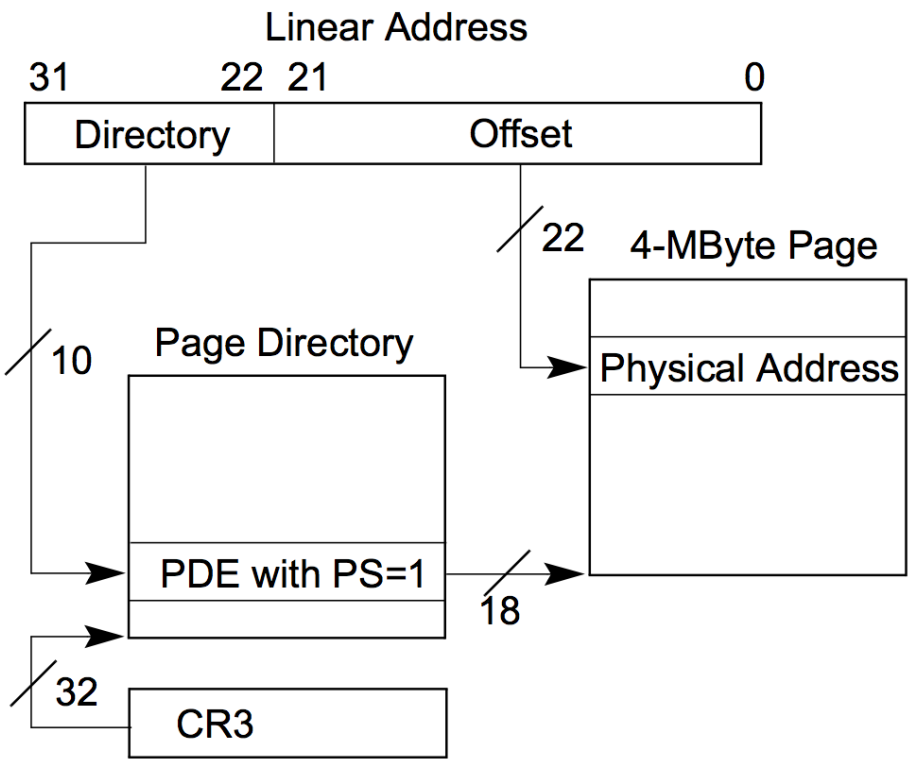
Data: 0x8010c628



Super Pages

- If the Page Size Extension (PSE) bit is set in CR4, we can optionally have entries in the page directory point directly to 4MB page frames
- Why would this be a good idea?
- This can be beneficial because it reduces paging overhead (only one level of lookup, more data mapped)

Translation with Super Pages



x86 PDE/PTE Super Pages

31–22	21–13	12	11–9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]			Ignored					PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame	Reserved (must be 0)	Bits 39:32 of address[2]	PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table			Ignored		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored												0	PDE: not present
Address of 4KB page frame			Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored												0	PTE: not present

Running the Numbers

- Basics: super pages use 22 bits for offset because 2^{22} = 4MB
- How many entries in a 4KB page table?
  - 1024 entries = 4MB addressed by one page table
  - So each page directory addresses 4MB of memory whether it points to a page table or a super page
- How many page directory entries?
  - We want to address 4GB of memory => 4GB/4MB = 1024
- That's why the available sizes are 4KB and 4MB pages

Paging Overhead in x86

- Suppose we have mapped 512MB worth of virtual address space using 4KB pages
- Each page table covers 4MB of memory
- So space required is … ?

```
sizeof(1 page directory) +
(512MB/4MB) * sizeof(1 page table) =
4KB + 128 * 4KB =
4KB + .5MB =
~0.503MB
```

Max Overhead

- 4KB + 1024 page tables * 4KB / table = ~4MB
- What if we just used a one-level page table with 4KB pages?
  - (4GB/4KB) entires * 4 bytes / entry = 4MB
- What if we used 4MB pages everywhere?
  - (4GB/4MB) entries * 4 bytes = 4096 bytes = 4KB

Other Features of x86 Paging

When a page directory / page table entry is non-present, it's format is unspecified by Intel

31–12	11–9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]	Ignored					PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame; Reserved (must be 0); Bits 39:32 of address[2]; PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table	Ignored		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored										0	PDE: not present
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

Non-Present PDEs/PTEs

- Since the CPU/MMU ignore these parts of the PDE/PTE, we can store stuff in them
- Common to use that space to store metadata about the non-present page
  - Example: if the page is available on disk (i.e., swapped out), give info how to retrieve it

Windows Kernel x86 Invalid PTEs

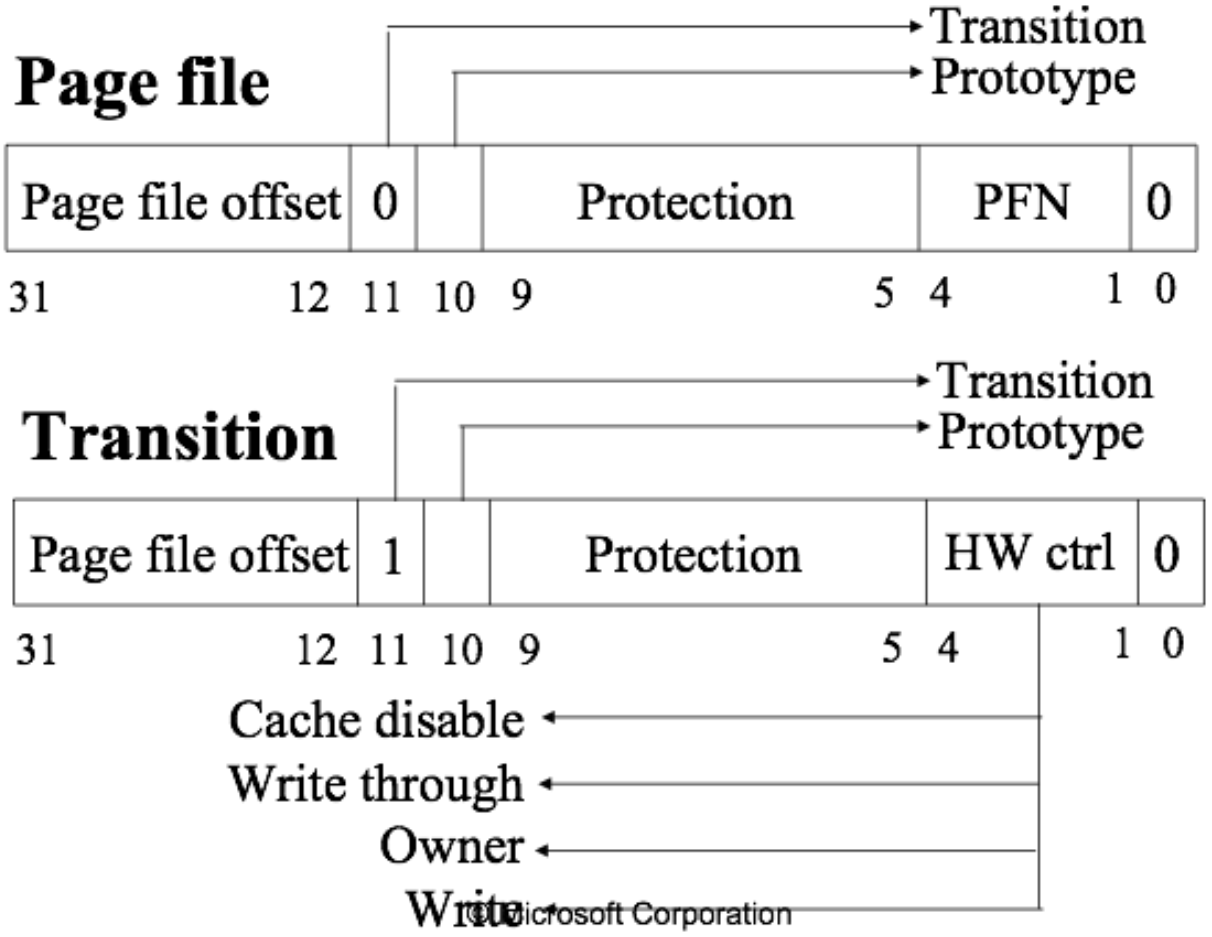


Diagram from David B. Probert / Microsoft Corp.

Windows Kernel

x86 Invalid PTEs

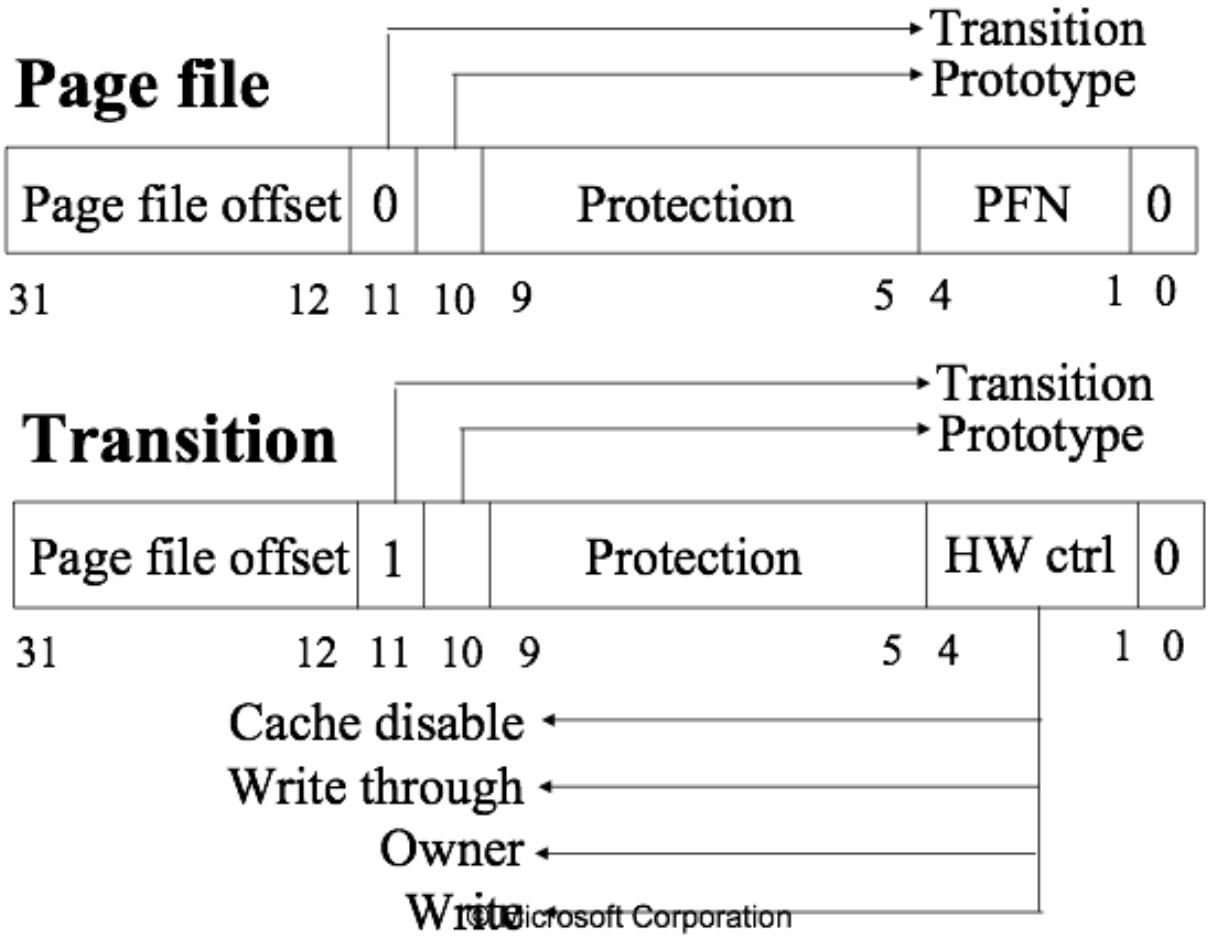


Diagram from David B. Probert / Microsoft Corp.