INF1060:
Introduction to Operating Systems and Data Communication

Operating Systems: Memory

Pål Halvorsen

Wednesday, September 26, 2012

Overview

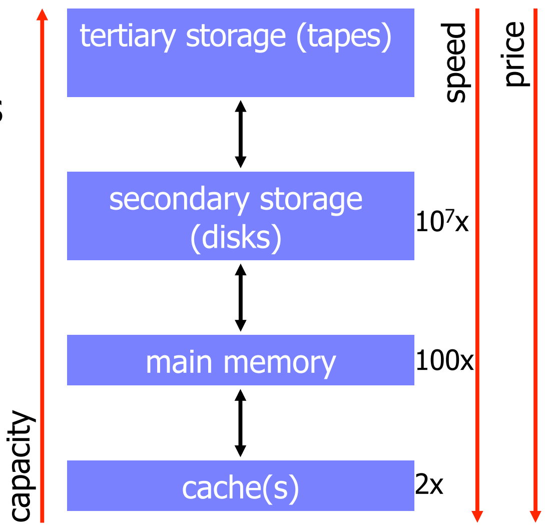
- Memory management
- Hierarchies
- Multiprogramming and memory management
- Addressing
- A process' memory
- Partitioning
- Paging and Segmentation
- Virtual memory
- Page replacement algorithms

Memory Management

- Memory management is concerned with managing the systems’ memory resources
  - different levels of memory in a hierarchy
  - providing a virtual view of memory giving the impression of having more than the amount of available bytes
  - allocating space to processes
  - protecting the memory regions

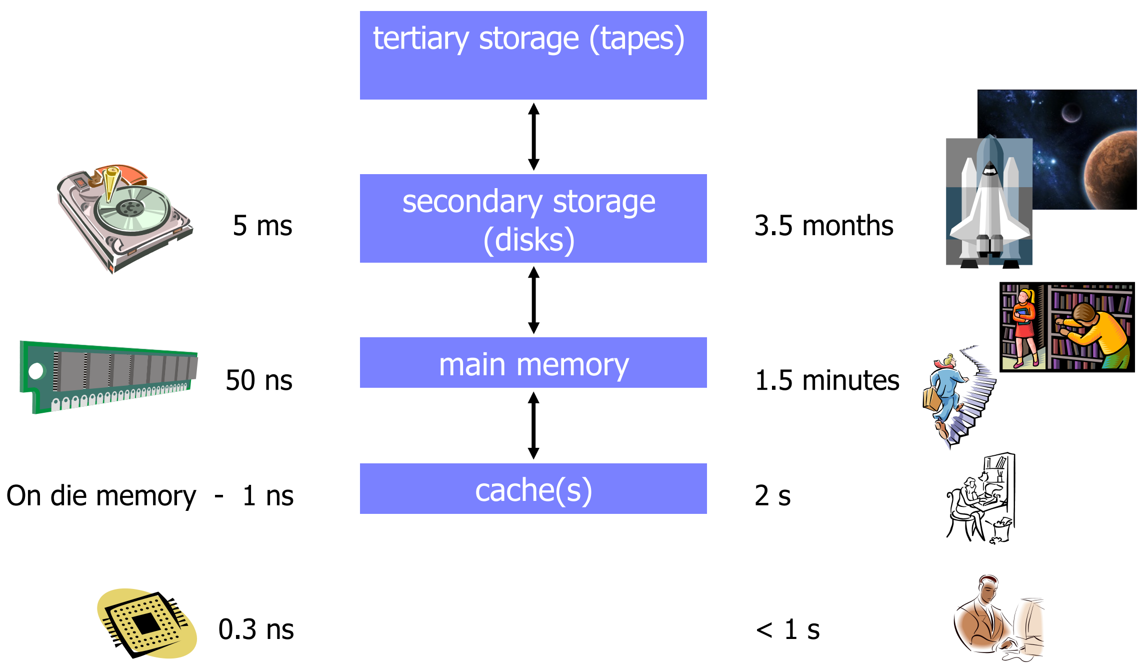
Memory Hierarchies

- We can’t access the disk each time we need data
- Typical computer systems therefore have several different components where data may be stored
  - different capacities
  - different speeds
  - less capacity gives faster access and higher cost per byte
- Lower levels have a copy of data in higher levels
- A typical memory hierarchy:



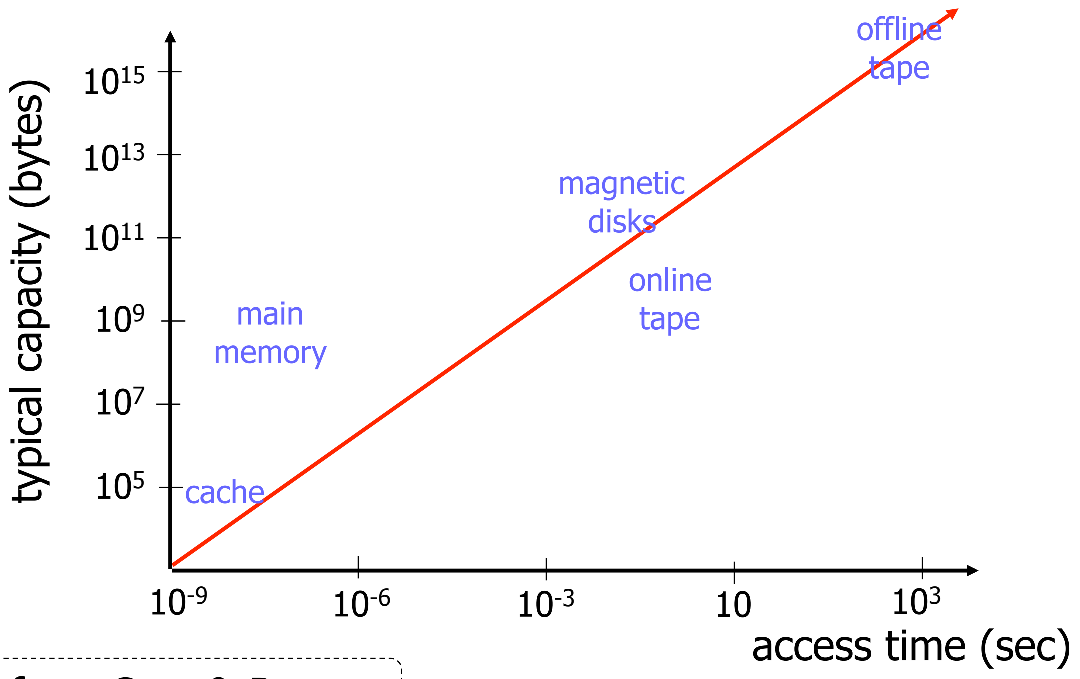
Memory Hierarchies

For handouts



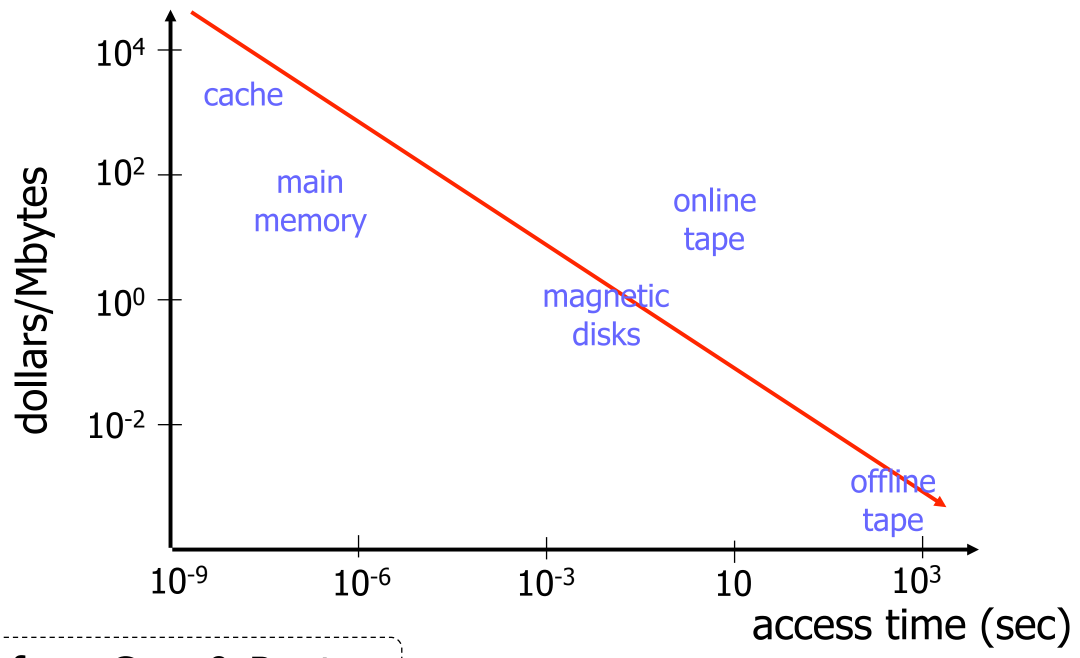
Memory Hierarchies

Storage Costs: Access Time vs Capacity



from Gray & Reuter

Storage Costs: Access Time vs Price



from Gray & Reuter

The Challenge of Multiprogramming

- Many “tasks” require memory
  - several programs concurrently loaded into memory
  - memory is needed for different tasks within a process
  - process memory demand may change over time

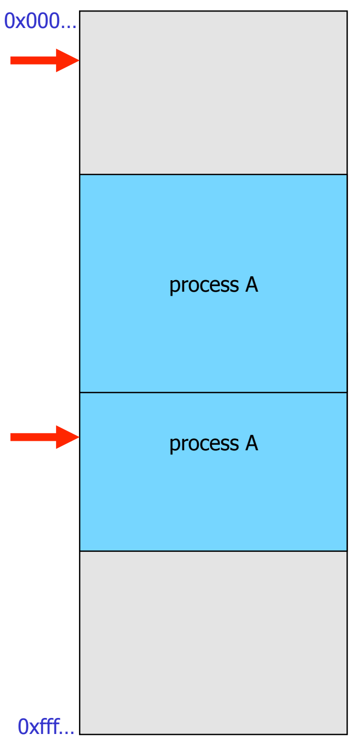
  ➥ **OS must arrange memory sharing**

Memory Management for Multiprogramming

- Use of secondary storage
  - keeping all programs and their data in memory may be impossible
  - move (parts of) a processes from memory
- Swapping: remove a process from memory
  - with all of its state and data
  - store it on a secondary medium
    (disk, flash RAM, other slow RAM, historically also tape)
- Overlays: manually replace parts of code and data
  - programmer’s rather than OS’s work
  - only for very old and memory-scarce systems
- Segmentation/paging: remove part of a process from memory
  - store it on a secondary medium
  - sizes of such parts are fixed

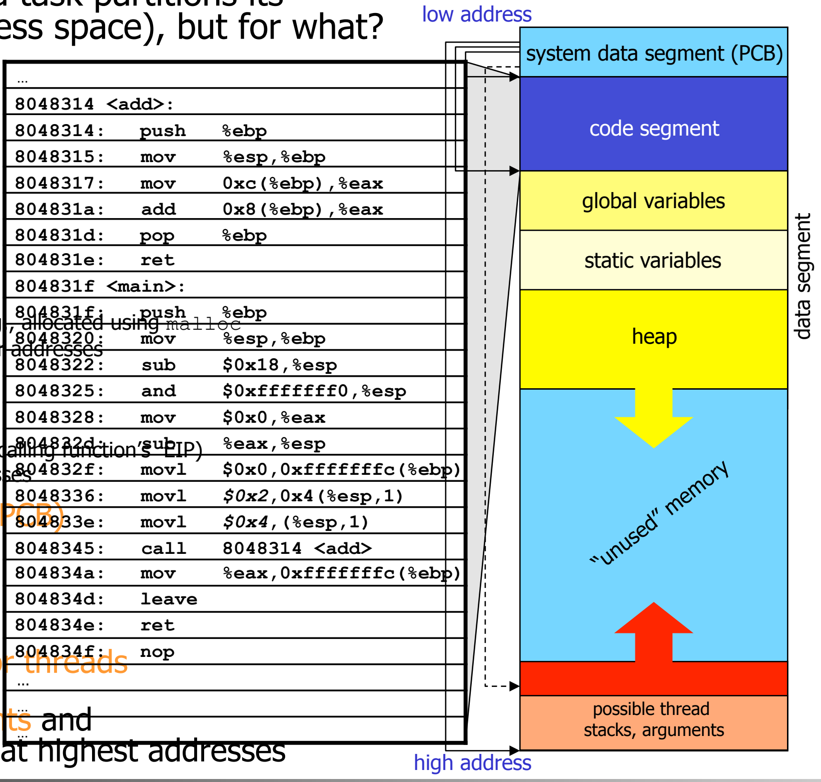
Absolute and Relative Addressing

- Hardware often uses absolute addressing
  - reserved memory regions
  - reading data by referencing the byte numbers in memory
  - read absolute byte 0x000000ff
  - *fast!!!*
- What about software?
  - read absolute byte 0x000fffff (process A)
  - ⇨ result dependent of physical process location
  - absolute addressing not convenient
  - but, addressing within a process is determined during programming!!??
- ⇨ Relative addressing
  - independent of process position in memory
  - address is expressed *relative to some base location*
  - *dynamic address translation* – find absolute address during run-time adding relative and base addresses



Processes' Memory

- On most architectures, a task partitions its available memory (address space), but for what?
  - a text (code) segment
    - read from program file for example by `exec`
    - usually read-only
    - can be shared
  - a data segment
    - global variables
    - static variables
    - heap
      - dynamic memory, e.g., allocated using `malloc`
      - grows against **higher** addresses
  - a stack segment
    - variables in a function
    - stored register states (e.g., calling function's EIP)
    - grows against **lower** addresses
  - system data segment (PCB)
    - segment pointers
    - pid
    - program and stack pointers
    - …
  - possibly more stacks for threads
  - command line arguments and environment variables at highest addresses

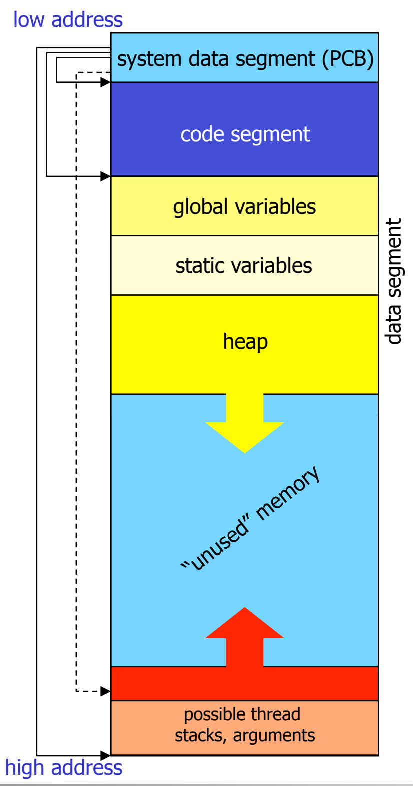


```
…
8048314 <add>:
8048314:    push    %ebp
8048315:    mov     %esp,%ebp
8048317:    mov     0xc(%ebp),%eax
804831a:    add     0x8(%ebp),%eax
804831d:    pop     %ebp
804831e:    ret
804831f <main>:
804831f:    push    %ebp
8048320:    mov     %esp,%ebp
8048322:    sub     $0x18,%esp
8048325:    and     $0xfffffff0,%esp
8048328:    mov     $0x0,%eax
804832d:    sub     %eax,%esp
804832f:    movl    $0x0,0xfffffffc(%ebp)
8048336:    movl    $0x2,0x4(%esp,1)
804833e:    movl    $0x4,(%esp,1)
8048345:    call    8048314 <add>
804834a:    mov     %eax,0xfffffffc(%ebp)
804834d:    leave
804834e:    ret
804834f:    nop
…
…
…
```

Processes' Memory

For handouts

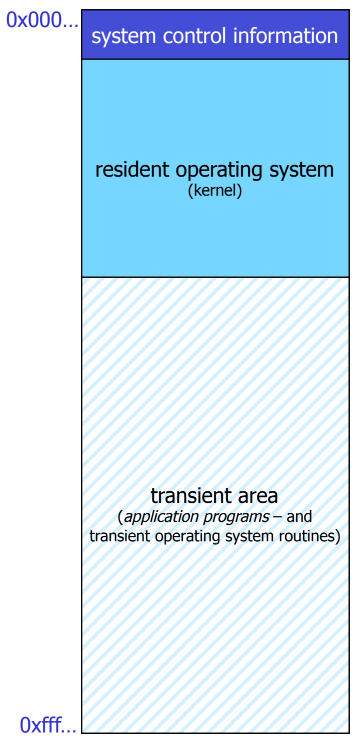
- On most architectures, a task partitions its available memory (address space), but for what?
  - a text (code) segment
    - read from program file for example by `exec`
    - usually read-only
    - can be shared
  - a data segment
    - global variables
    - static variables
    - heap
      - dynamic memory, e.g., allocated using `malloc`
      - grows against **higher** addresses
  - a stack segment
    - variables in a function
    - stored register states (e.g., calling function's EIP)
    - grows against **lower** addresses
  - system data segment (PCB)
    - segment pointers
    - pid
    - program and stack pointers
    - …
  - possibly more stacks for threads
  - command line arguments and environment variables at highest addresses



Memory Layout

- Memory is usually divided into regions
  - operating system occupies low memory
    - system control
    - resident routines
  - the remaining area is used for transient operating system routines and application programs
- How to assign memory to concurrent processes?

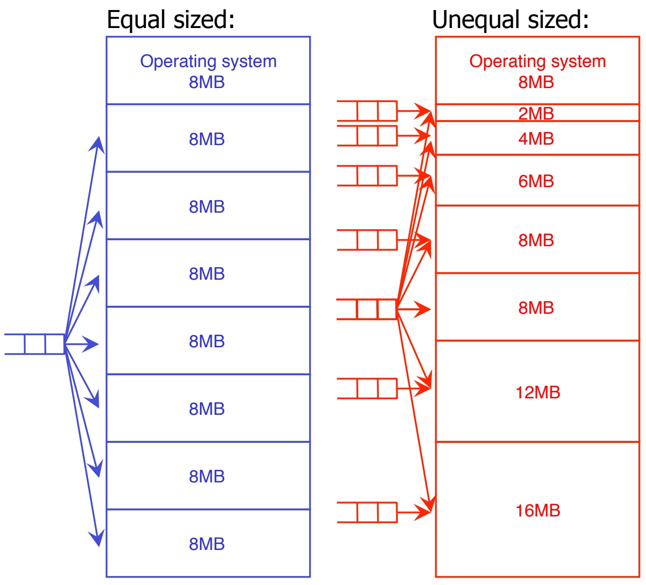
⇨ Memory partitioning
  - Fixed partitioning
  - Dynamic partitioning
  - Simple segmentation
  - Simple paging
  - Virtual memory with segmentation
  - Virtual memory with paging



0x000…
system control information
resident operating system
(kernel)
transient area
(application programs – and transient operating system routines)
0xfff…

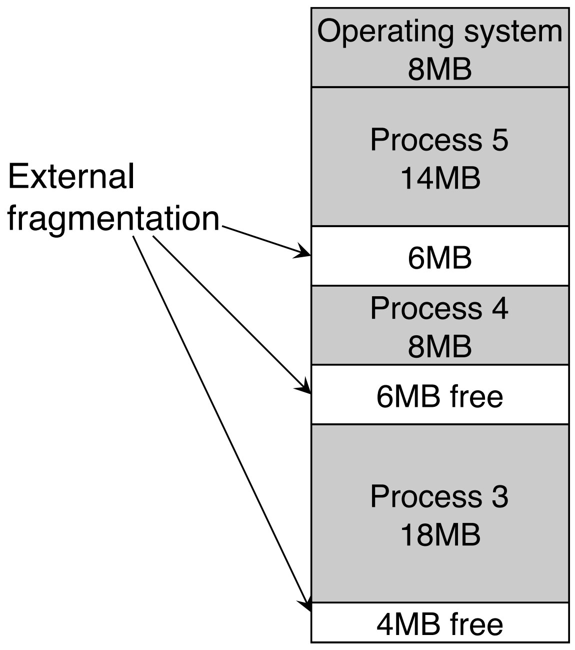
Fixed Partitioning

- Divide memory into static partitions at system initialization time (boot or earlier)
- Advantages
  - very easy to implement
  - can support swapping process
- Two fixed partitioning schemes
  - equal-size partitions
    - large programs cannot be executed (unless program parts are loaded from disk)
    - small programs don't use entire partition (problem called "internal fragmentation")
  - unequal-size partitions
    - large programs can be loaded at once
    - less internal fragmentation
    - require assignment of jobs to partitions
    - one queue or one queue per partition
    - …but, what if only small or large processes?



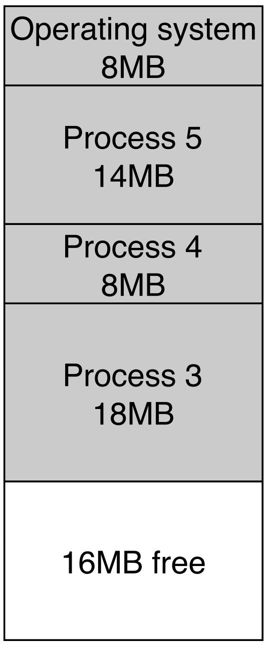
Dynamic Partitioning

- Divide memory in run-time
  - partitions are created dynamically
  - removed after jobs are finished
- External fragmentation increases with system running time



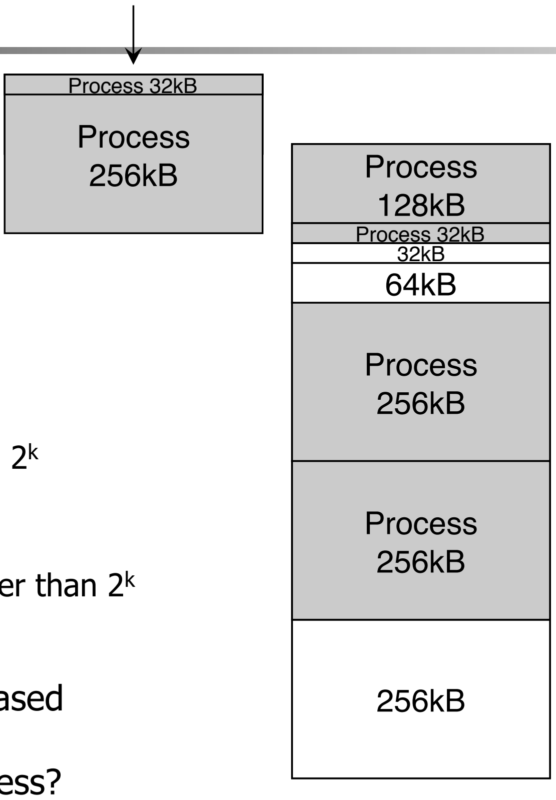
Dynamic Partitioning

- Divide memory in run-time
  - partitions are created dynamically
  - removed after jobs are finished
- External fragmentation increases with system running time
- ***Compaction*** removes fragments by moving data in memory
  - takes time
  - consumes processing resources
- Proper placement algorithm might reduce need for compaction
  - first fit – simplest, fastest, typically the best
  - next fit – problems with large segments
  - best fit – slowest, lots of small fragments, therefore worst



Buddy System

- Mix of fixed and dynamic partitioning
  - partitions have sizes 2^k, $L \leq k \leq U$
- Maintain a list of holes with sizes
- Assigning memory to a process:
  - find smallest k so that process fits into 2^k
  - find a hole of size 2^k
  - if not available, split smallest hole larger than 2^k recursively into halves
- Merge partitions if possible when released
- … but what if I now got a 600kB process?
  … do we really need the process in continuous memory?



Segmentation

- Requiring that a process is placed in contiguous memory gives much fragmentation (and memory compaction is expensive)

- Segmentation
  - different lengths
  - determined by programmer
  - memory frames

- Programmer (or compiler tool-chain) organizes program in parts
  - move control
  - needs awareness of possible segment size limits

- Pros and Cons
  - 👍 principle as in dynamic partitioning – can have different sizes
  - 👍 no internal fragmentation
  - 👍 less external fragmentation because on average smaller segments

  - 👎 adds a step to address translation

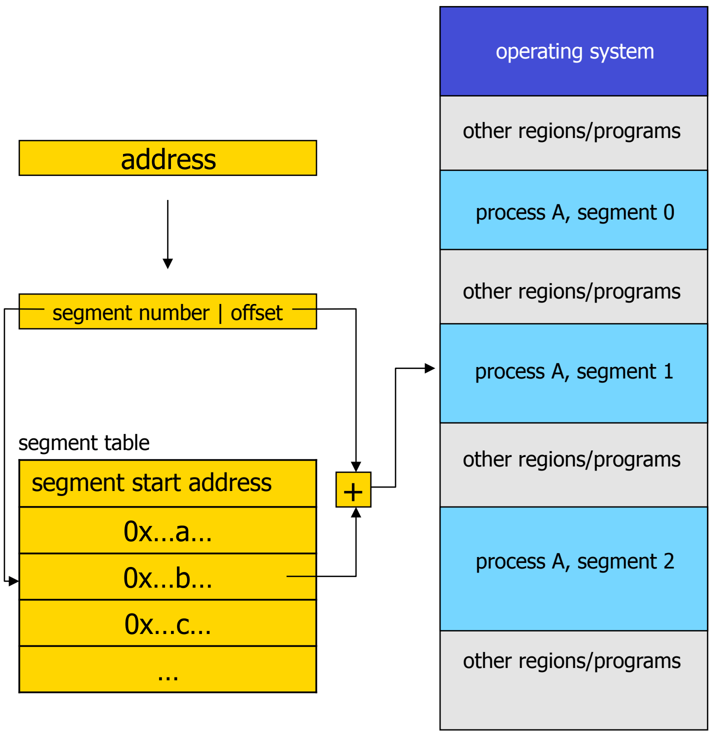
Segmentation

1. find segment table in register

2. extract segment number from address

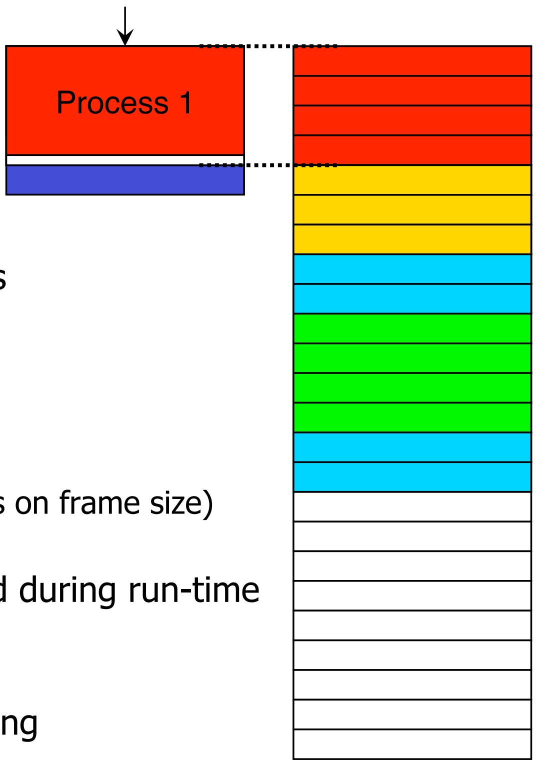
3. find segment address using segment number as index to segment table

4. find absolute address within segment using relative address



Paging

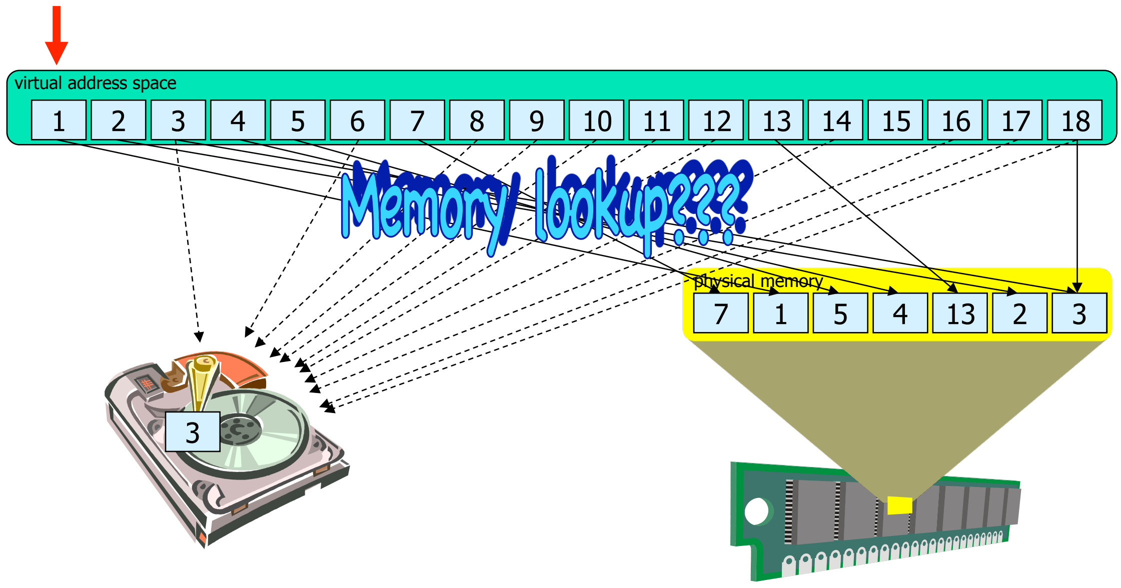
- Paging
  - equal lengths determined by processor
  - one page moved into one page (memory) frame
- Process is loaded into several frames (not necessarily consecutive)
- Fragmentation
  - no external fragmentation
  - little internal fragmentation (depends on frame size)
- Addresses are dynamically translated during run-time (similar to segmentation)
- Can *combine* segmentation and paging



Virtual Memory

- The described partitioning schemes may be used in applications, but the modern OS also uses virtual memory:
  - early attempt to give a programmer more memory than physically available
    - older computers had relatively little main memory
    - but, all instructions does not have to be in memory before execution starts
    - break program into smaller independent parts
    - load currently active parts
    - when a program is finished with one part a new can be loaded
  - memory is divided into equal-sized frames often called *pages*
  - some pages reside in physical memory, others are stored on disk and retrieved if needed
  - virtual addresses are translated to physical (in MMU) using a *page table*
  - both Linux and Windows implements a flat linear 32-bit (4 GB) memory model on IA-32
    - **Windows**: 2 GB (high addresses) kernel, 2 GB (low addresses) user mode threads
    - **Linux**: 1 GB (high addresses) kernel, 3 GB (low addresses) user mode threads

Virtual Memory



Memory Lookup

Page table

		present bit
15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

Outgoing physical address

1	1	0													

(0x6004, 24580)

Example:
- 4 KB pages (12-bit offsets within page)
- 16 bit virtual address space → 16 pages (4-bit index)
- 8 physical pages (3-bit index)

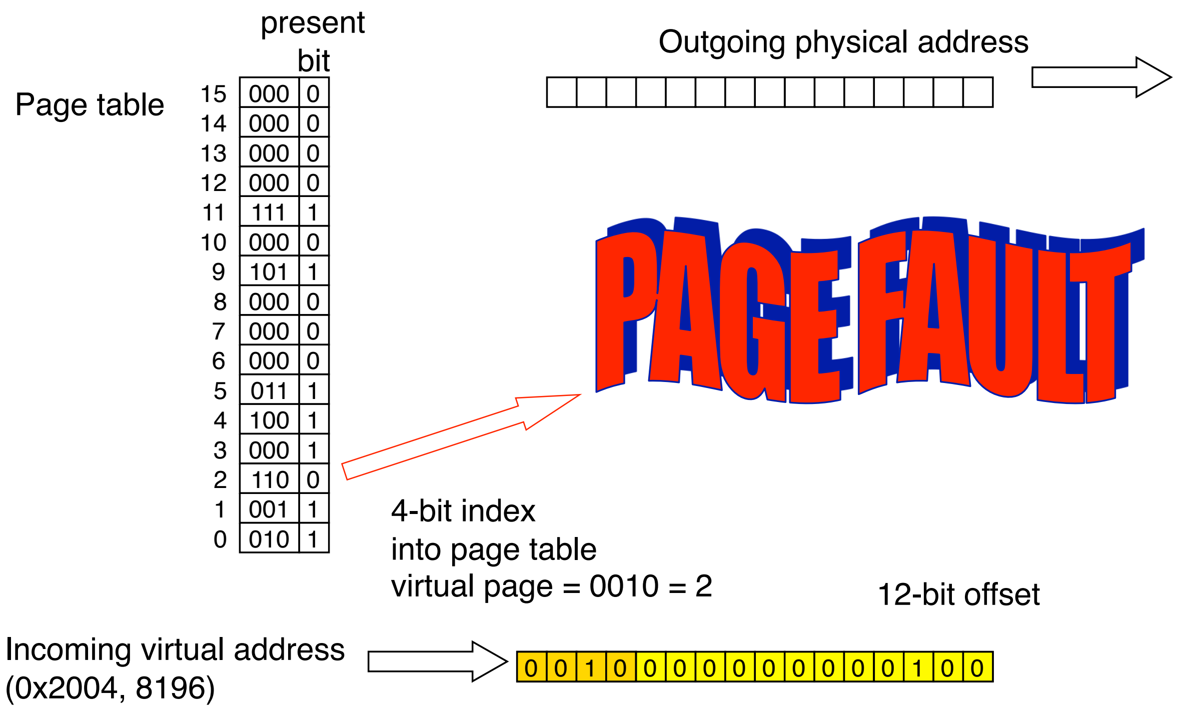
4-bit index
into page table
virtual page = 0010 = 2

12-bit offset

Incoming virtual address
(0x2004, 8196)

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

Memory Lookup



Page Fault Handling

1. Hardware traps to the kernel saving program counter and process state information
2. Save general registers and other volatile information
3. OS discovers the page fault and determines which virtual page is requested
4. OS checks if the virtual page is valid and if protection is consistent with access
5. Select a page to be replaced
6. Check if selected page frame is ”dirty”, i.e., updated. If so, write back to disk, otherwise, just overwrite
7. When selected page frame is ready, the OS finds the disk address where the needed data is located and schedules a disk operation to bring in into memory
8. A disk interrupt is executed indicating that the disk I/O operation is finished, the page tables are updated, and the page frame is marked ”normal state”
9. Faulting instruction is backed up and the program counter is reset
10. Faulting process is scheduled, and OS returns to the routine that made the trap to the kernel
11. The registers and other volatile information are restored, and control is returned to user space to continue execution as no page fault had occured

Page Replacement Algorithms

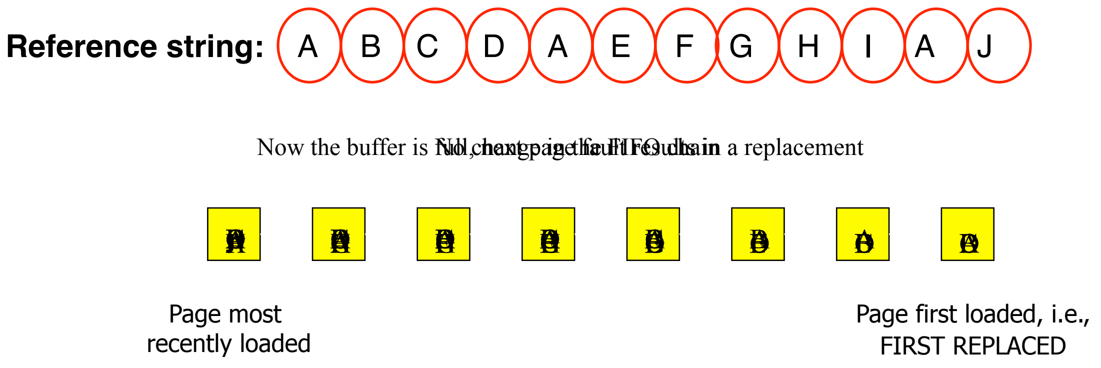
- Page fault → OS has to select a page for replacement
- How do we decide which page to replace?

  → determined by the ***page replacement algorithm***
  → several algorithms exist:

  - Random
  - Other algorithms take into acount usage, age, etc.
    (e.g., FIFO, not recently used, least recently used, second chance, clock, …)
  - which is best???

First In First Out (FIFO)

- All pages in memory are maintained in a list sorted by age
- FIFO replaces the oldest page, i.e., the first in the list


Reference string:
A
B
C
D
A
E
F
G
H
I
A
J
Now the buffer is full, next page fault results in a replacement
No change in the FIFO chain
Page most recently loaded
Page first loaded, i.e., FIRST REPLACED

- Low overhead
- Non-optimal replacement (and disc accesses are EXPENSIVE)

➥ FIFO is rarely used in its pure form

Second Chance

- Modification of FIFO
- ***R*** bit: when a page is referenced again, the R bit is set, and the page will be treated as a newly loaded page

Reference string: A B C D A E F G H I

Page I will be inserted, find a page to page out by looking at the first page loaded:

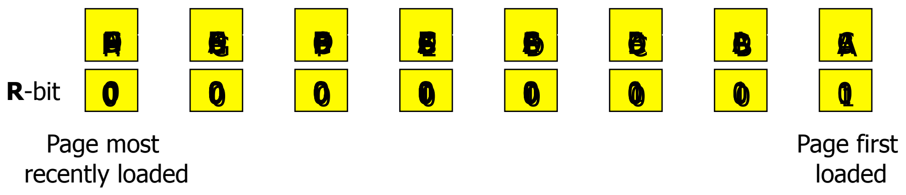
-if R-bit = 0 → replace

Page A's R-bit = 1 → move last in chain and clear R-bit, look at new first page (B)

Page B's R-bit = 0 → page out, shift chain left, and insert I last in the chain

Now the buffer is full, next page fault results in a replacement

-if R-bit = 1 → clear R-bit, move page last, and finally look at the new first page

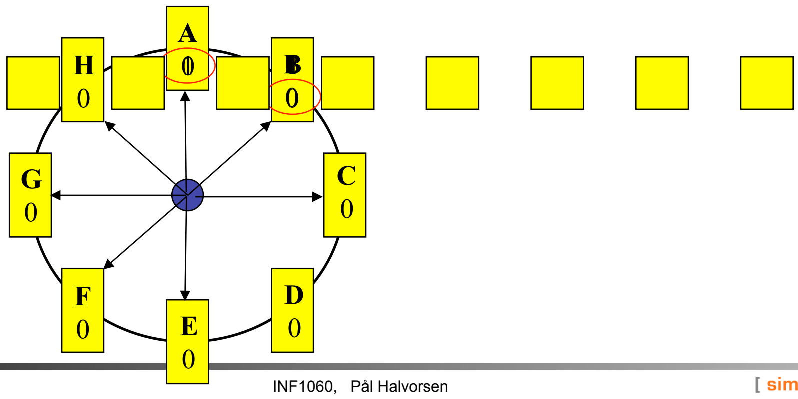


- *Second chance* is a reasonable algorithm, but inefficient because it is moving pages around the list

Clock

- More efficient implemention ***Second Chance***
- Circular list in form of a clock
- Pointer to the oldest page:
  - R-bit = 0 → replace and advance pointer
  - R-bit = 1 → set R-bit to 0, advance pointer until R-bit = 0, replace and advance pointer

Reference string: A B C D A E F G H I



Least Recently Used (LRU)

- Replace the page that has the longest time since last reference

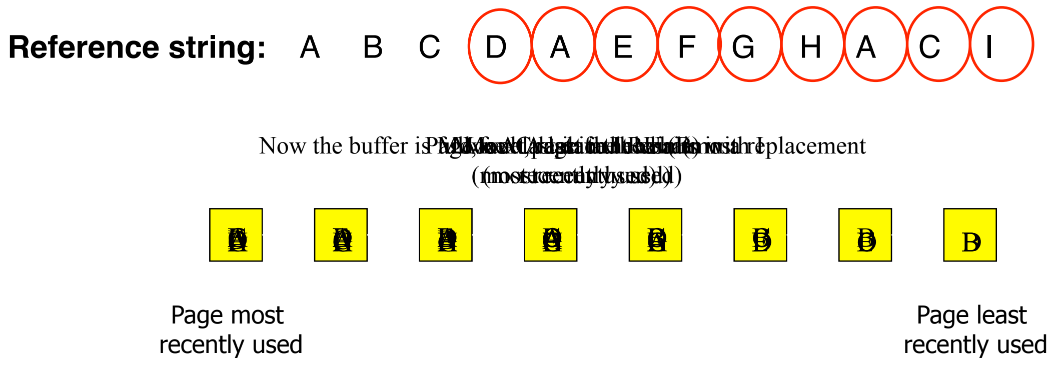
- Based on the observation that

  pages that was heavily used in the last few instructions will probably be used again in the next few instructions

- Several ways to implement this algorithm

Least Recently Used (LRU)

- LRU as a linked list:


Reference string: A B C D A E F G H A C I
Now the buffer is [illegible] replacement
[illegible]
Page most recently used
Page least recently used

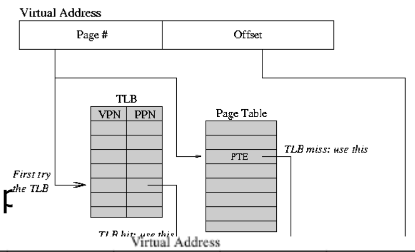
- Saves (usually) a lot of disk accesses
- **Expensive** - maintaining an ordered list of all pages in memory:
  - most recently used at front, least at rear
  - update this list every memory reference !!

- Many other approaches: using aging and counters

Speeding up paging...

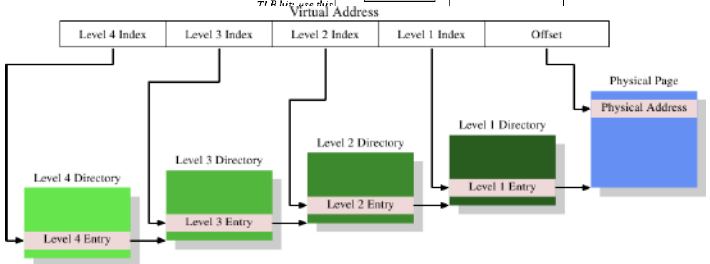
- Every memory reference needs a virtual-to-physical mapping
- Each process has its own virtual address space (an own page table)
- Large tables:
  - 32-bit addresses, 4 KB pages → 1.048.576 entries
  - 64-bit addresses, 4 KB pages → 4.503.599.627.370.496 entries

➥ Transaction lookaside buffers (aka associative memory)
- hardware "cache" for the page table
- a fixed number of slots containing the last page table entries



➥ Page size:
larger page sizes reduce number of p

➥ Multi-level page tables

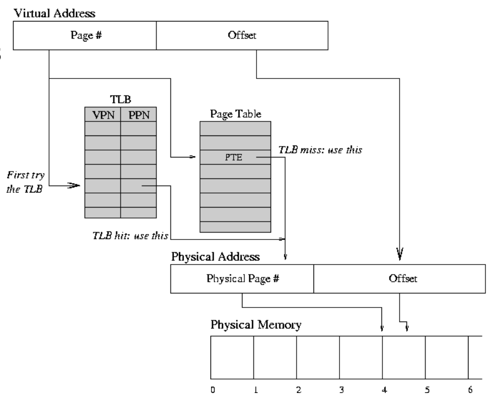


Speeding up paging...

For handouts

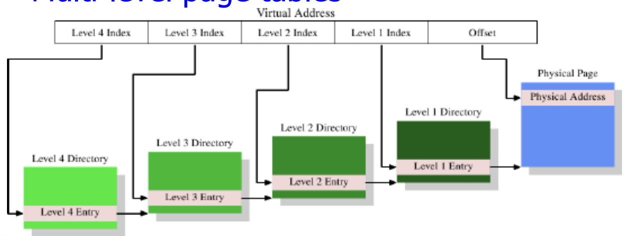
- Every memory reference needs a virtual-to-physical mapping
- Each process has its own virtual address space (an own table)
- Large tables:
  - 32-bit addresses, 4 KB pages → 1.048.576 entries
  - 64-bit addresses, 4 KB pages → 4.503.599.627.370.496 entries

➥ Transaction lookaside buffers (aka associative memory)
- hardware "cache" for the page table
- a fixed number of slots containing the last page table entries



➥ Page size:
larger page sizes reduce number of pages

➥ Multi-level page tables

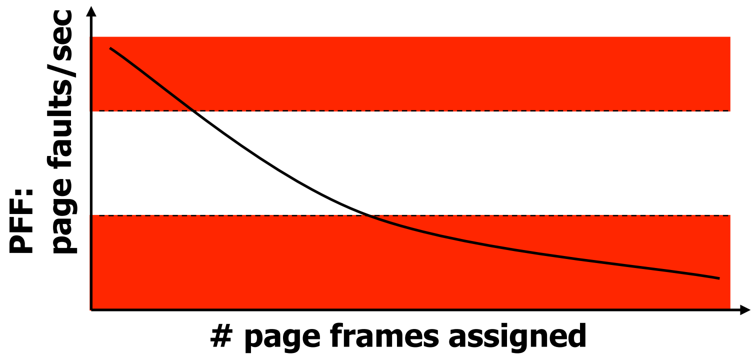


Many Other Design Issues

- Page size
- Reference locality in time and space
- Demand paging vs. pre-paging
- Allocation policies: equal share vs. proportional share
- Replacement policies: local vs. global
- …

Allocation Policies

- Page fault frequency (PFF):
  Usually, more page frames → fewer page faults



PFF is unacceptable high
→ process needs more memory

PFF might be too low
→ process may have too much memory!!!??????

If the system experience too many page faults, what should we do?
Reduce number of processes competing for memory
- reassign a page frame
- swap one or more to disk, divide up pages they held
- reconsider degree of multiprogramming

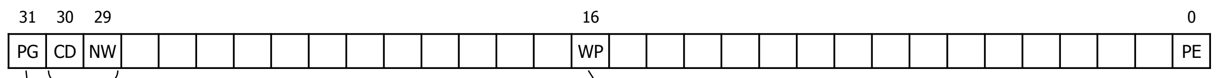
Example:
Paging on Pentium

Paging on Pentium

- The executing process has a 4 GB address space (2^{32}) – viewed as 1 M (2^{20}) 4 KB pages
  - The 4 GB address space is divided into 1 K page groups (pointed to by the 1 level table – page directory)
  - Each page group has 1 K 4 KB pages (pointed to by the 2 level tables – page tables)
- Mass storage space is also divided into 4 KB blocks of information
- Uses control registers for paging information

Control Registers used for Paging on Pentium

- Control register 0 (CR0):



Not-Write-Through and Cache Disable:
used to control internal cache

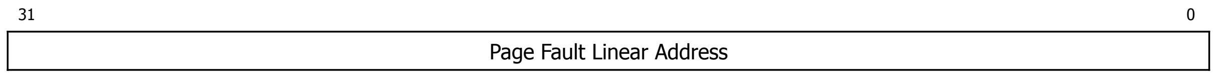
Paging Enable:
OS enables paging by setting CR0[PG] = 1

Write-Protect: If CR0[WP] = 1,
only OS may write to read-only pages

Protected Mode Enable: If CR0[PE] = 1,
the processor is in protected mode

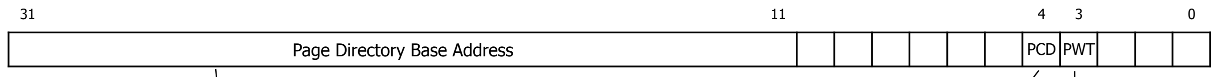
- Control register 1 (CR1) – does not exist, returns only zero

- Control register 2 (CR2)
  - only used if CR0[PG]=1 & CR0[PE]=1



Control Registers used for Paging on Pentium

- Control register 3 (CR3) – page directory base address:
  - only used if CR0[PG]=1 & CR0[PE]=1

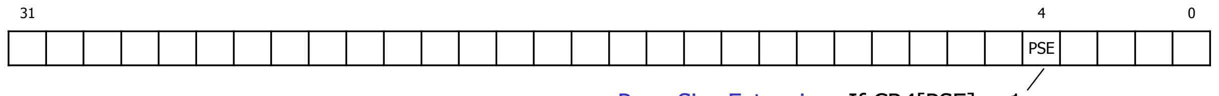


A 4KB-aligned physical base
address of the page directory

Page Cache Disable:
If CR3[PCD] = 1, caching is turned off

Page Write-Through:
If CR3[PWT] = 1, use write-through updates

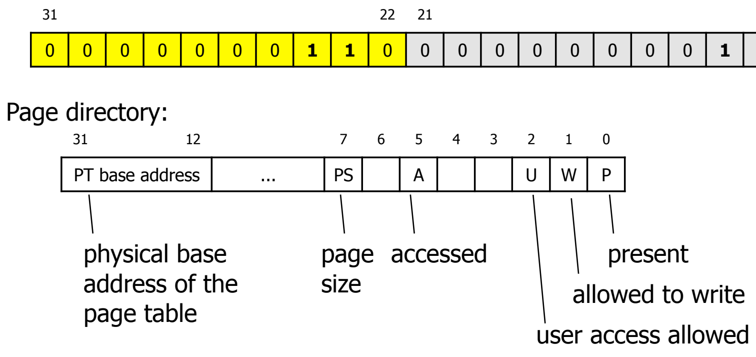
- Control register 4 (CR4):



Page Size Extension: If CR4[PSE] = 1,
the OS designer may designate some pages as 4 **MB**

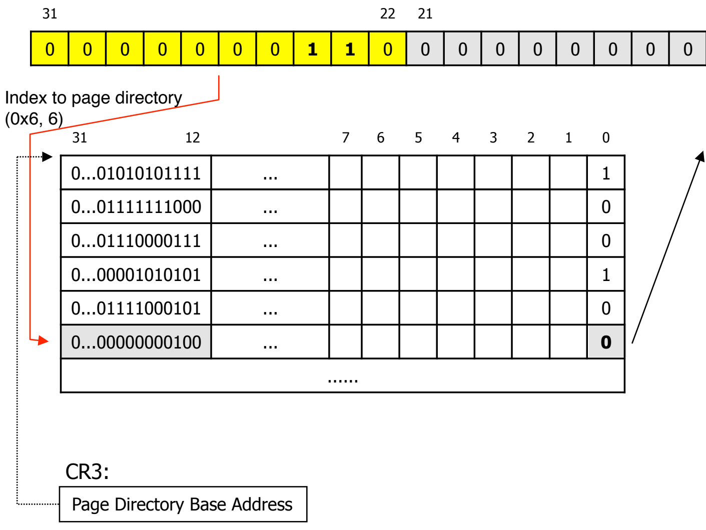
Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)



Pentium Memory Lookup

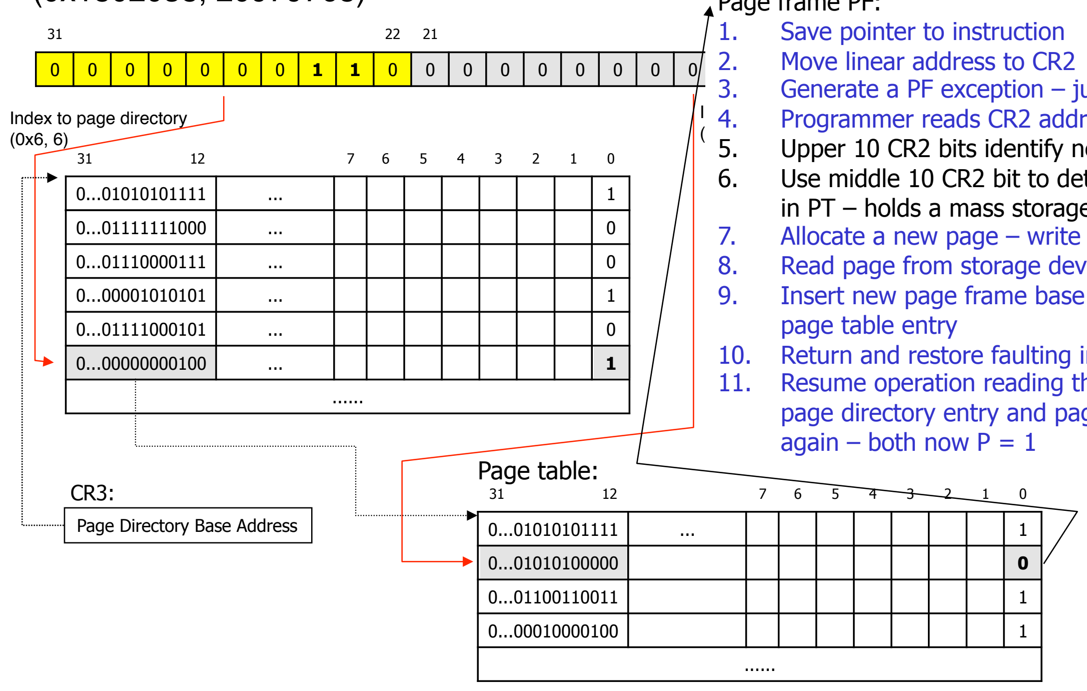
Incoming virtual address (CR2)
(0x1802038, 20979768)



Page table PF:
1. Save pointer to instruction
2. Move linear address to CR2
3. Generate a PF exception – jump to handler
4. Programmer reads CR2 address
5. Upper 10 CR2 bits identify needed PT
6. Page directory entry is really a mass storage address
7. Allocate a new page – write back if dirty
8. Read page from storage device
9. Insert new PT base address into page directory entry
10. Return and restore faulting instruction
11. Resume operation reading the same page directory entry again – now P = 1

Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)



31	22	21															
0	0	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0

31 12		7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01111111000	...								0
0...01110000111	...								0
0...00001010101	...								1
0...01111000101	...								0
0...00000000100	...								**1**

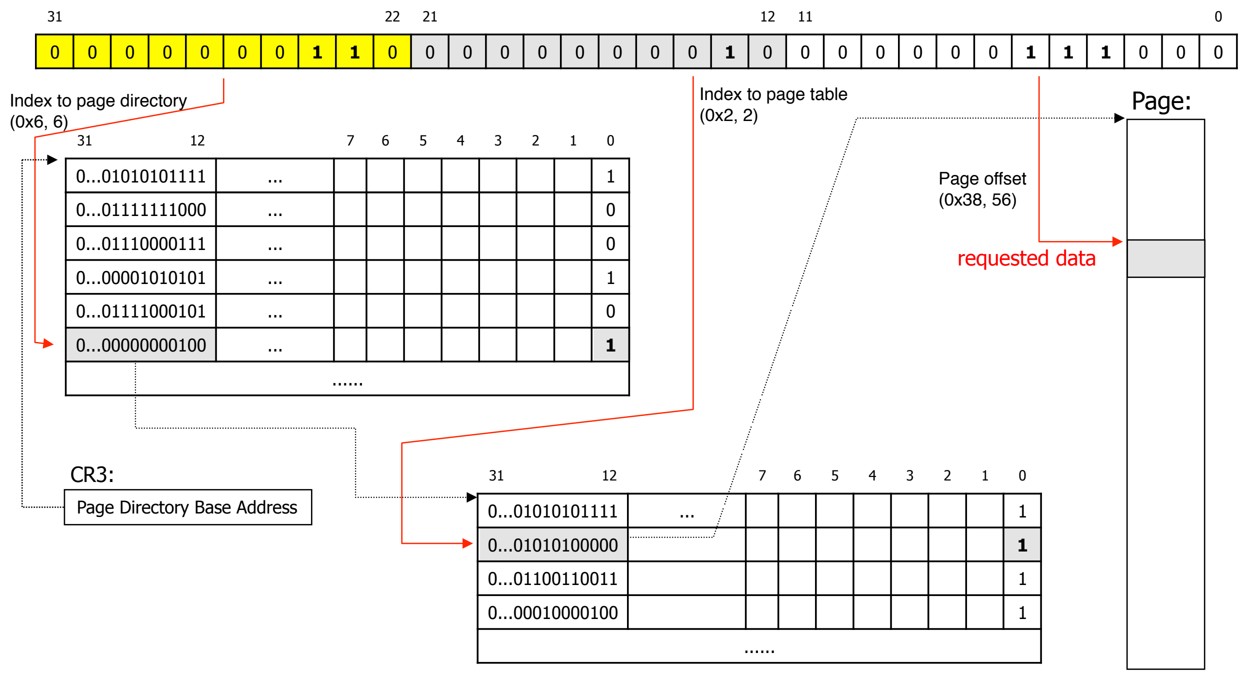
31 12		7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01010100000									**0**
0...01100110011									1
0...00010000100									1

Page frame PF:

1. Save pointer to instruction
2. Move linear address to CR2
3. Generate a PF exception – jump to handler
4. Programmer reads CR2 address
5. Upper 10 CR2 bits identify needed PT
6. Use middle 10 CR2 bit to determine entry in PT – holds a mass storage address
7. Allocate a new page – write back if dirty
8. Read page from storage device
9. Insert new page frame base address into page table entry
10. Return and restore faulting instruction
11. Resume operation reading the same page directory entry and page table entry again – both now P = 1

Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)



31								22	21									12	11											0
0	0	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	**1**	**1**	**1**	0	0	0

31 12		7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01111111000	...								0
0...01110000111	...								0
0...00001010101	...								1
0...01111000101	...								0
0...00000000100	...								**1**

31 12		7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01010100000									**1**
0...01100110011									1
0...00010000100									1

Pentium Page Fault Causes

- Page directory entry's P-bit = 0:
  page group's directory (page table) not in memory
- Page table entry's P-bit = 0:
  requested page not in memory
- Attempt to write to a read-only page
- Insufficient page-level privilege to access page table or frame
- One of the reserved bits are set in the page directory or table entry

Summary

- Memory management is concerned with managing the systems' memory resources
  - allocating space to processes
  - protecting the memory regions
  - in the real world
    - programs are loaded dynamically
    - physical addresses it will get are not known to program – dynamic address translation
    - program size at run-time is not known to kernel
- Each process usually has text, data and stack segments
- Systems like Windows and Unix use virtual memory with paging
- Many issues when designing a memory component