Memory Management
Part II

- Virtual memory
- Paging and page replacement
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation

Virtual Memory (VM)

- Basic idea
- Paging
- Demand Paging and its performance
- Page Replacement and Page-Replacement Algorithms
- Allocation of Frames
- Thrashing problem
- Segmentation: pure segmentation, Segmentation with paging

Basic Idea of Virtual Memory

- Basic idea behind the virtual memory is that the combined size of programs, data, and stacks may easily exceed the amount of physical memory available.
- This is more true in multiprogramming environment.
- The operating system uses physical memory together with secondary storage to solve this problem.
- **Virtual Memory** seems to be a state of art universal method. Processes are allocated physical memory at a point they need based on the availability.

Paging Concept in Virtual Memory

- Logical memory is divided into blocks of same size called **pages**.
- Physical memory is divided into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Pages are needed to be mapped to frames during the execution.
- To run a program of size n pages, need to use *n* frames.
- OS need to manage the page-to-frame mapping dynamically.

Page Faulting

- During the execution, a reference to virtual address is first checked if it has already been mapped to a physical address.
- If not it is said ta cause a “page fault”.

Page Fault Algorithm-1

- If there is a reference to a page (every address reference is a reference to a page),
  - It will cause a trap to OS, which will in turn cause “page fault” processing.
- Page fault processing:
  - Find an empty frame
  - Swap page into that frame
  - Reset page’s validation bit.
  - Give control back to the process causing the trap (restart instruction)

Page Fault-2

- **If there is no free page, OS starts page replacement procedure:**
  - Page replacement: find some page in the memory, the one which is not really in use, to replace.
  - Conduct the swap operations.
  - For performance reasons the algorithm used should result in minimum number of page faults.
  - It is possible that the same page is brought into memory several times, during execution of a program…

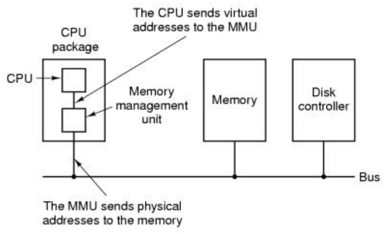
Address Translation-1

- The virtual address generated during the execution is composed of two parts:
  - *page number (p)*
    - used as an index into a *page table* which contains **base address** of the page in physical memory (**frame**).
  - *Page offset (d)*
    - combined with base address to define the physical memory address to be referenced.

Address Translation-2

- The base address is the address of the frame to which the page is mapped.
  - if the virtual page is already in the memory, the address mapping is straight forward and very efficient as it is done in firmware.
- If the page is not in the memory, first a **page fault** occurs, after which the address mapping is as before.

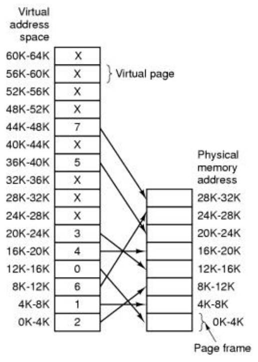
Virtual address and MMU-Memory Management Unit


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

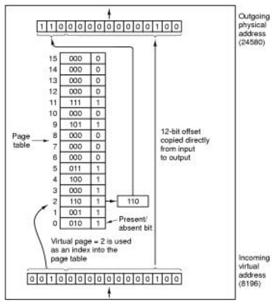
The position and function of the MMU

Page-frame mapping: Example

The relation between virtual addresses and physical memory address: page table and memory



MMU address mapping



Internal operation of MMU with 16 4 KB pages

Implementation of Page Table

- Page table is kept in main memory.
- *Page-table base register* (PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers (TLBs)*

Associative Register

- Associative registers – parallel search

Page #	Frame #

- Address translation (A´, A´´)
  - If A´ is in associative register, get frame #.
  - Otherwise get frame # from page table in the main memory

Associative Register

- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.

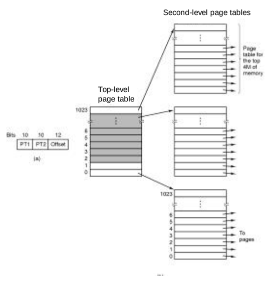
Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative memory = α
- Effective Access Time (EAT)

EAT = f(PageTableAccessTime,MemoryAccessTime)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Two-level page tables



- 32 bit address with 2 page table fields

Multilevel Paging and Performance

- Since each level is stored as a separate table in memory, mapping a logical address to a physical one may take four memory accesses.
- Even though time needed for one memory access is theoretically four times as much, caching permits performance to remain reasonable.

Multilevel Paging and Performance: Example

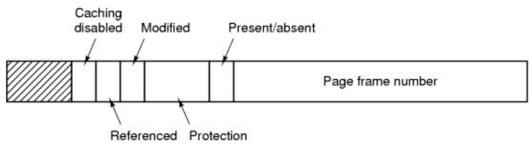
- Cache hit rate of 98 percent yields: mem.access=100 nsec, cache.access=20 nsec,

  effective access time = 0.98 x (100+20) + 0.02 x (400+20)

  = 126 nanoseconds.

  which is only a 26 percent slowdown in memory access time.

Page Tables Entry Format



Typical page table entry

TLBs – Translation Lookaside Buffers

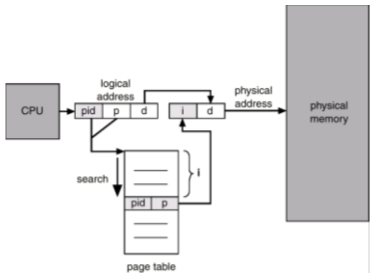
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

A TLB to speed up paging

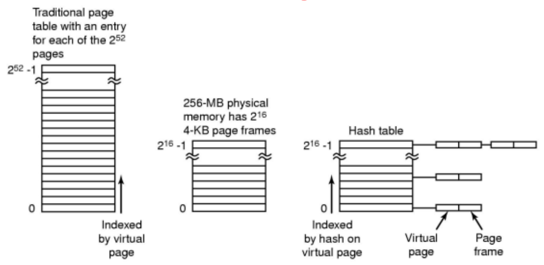
Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Inverted Page Table Architecture



Inverted Page Tables

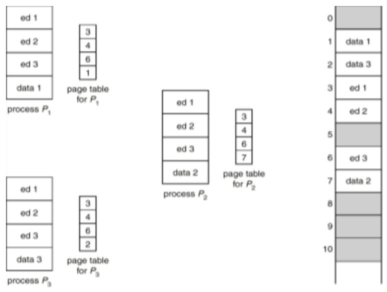

Traditional page table with an entry for each of the 2^52 pages
2^52 -1
0
Indexed by virtual page
256-MB physical memory has 2^16 4-KB page frames
2^16 -1
0
Hash table
2^16 -1
0
Indexed by hash on virtual page
Virtual page
Page frame

Comparison of a traditional page table with an inverted page table

Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example: Editor is shared



DEMAND PAGING

Demand Paging

- It means bringing a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- Page replacement – find some page in memory, but not really in use, swap it out:
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT):

  EAT = (1 – *p*) *memory access time

  + *p**(page fault overhead+ swap page out

  + swap page in + restart overhead time)

Demand Paging: Example

- Assumption: No associative memory
  - Memory access time = 1 microsecond
  - Page fault rate p=50%
  - q (=50) % of the time the page that is being replaced has been modified and therefore needs to be swapped out.
  - Swap Page Time = 10 msec = 10,000 microsec
- Computation of Effective Access Time:

  EAT = (1 – p) *2 + p (2+q*10000+(1-q)*20000))

  =(1-0.5)*2+0.5(2+0.5*(10000)+(1-0.5)*20000)

  =1+1+2500+5000

  =75002 micro sec

Page Replacement Algorithms

- Page fault process:
  - Find the page to be removed or
  - make room for incoming page
- Modified page must first be saved before being overwritten
  - If unmodified can be overwritten
- Better not to choose an often or recently used page.
  - It will probably need to be brought back in soon!

Page Replacement Algorithms

- The real algorithms in use are generally combination of several approaches. The discussion of few of the ones listted below is more an academic one:
  - Optimal
  - Not recently used
  - FIFO
  - Second Chance
  - The Clock
  - Least Recently Used
  - Simulating LRU

Optimal Page Replacement Algorithms

- Replace the page needed at the farthest point in the future
- Normally, given a proram, this is theoretically impossible to know. However it can be predicted by some means.
  - For example, logging page use on the previous runs of a process can be replayed for the future executions of the same program with same data…

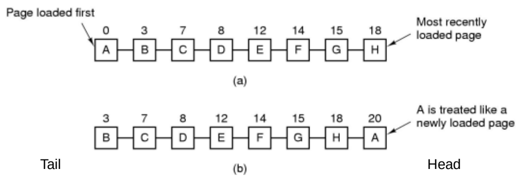
Not Recently Used Page Replacement Algorithm: Implementation

- Each page has Reference bit, Modified bit
  - R ad M bits are **set** when page is referenced and or modified
- They are classified according to the combination.
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random from lowest numbered non empty class

FIFO Page Replacement Algorithm: Implementation

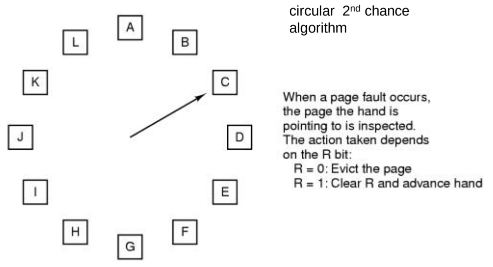
- Maintain a linked list of all pages
  - in order of their coming into the memory
- When required, the page at beginning of list is replaced.
- Disadvantage
  - page in the memory the longest time, may be the most often used one! Yet it will be replaced, causing extensive page fault…

Second Chance Page Replacement Algorithm: number indicate the last reference times



- Operation of a second chance
  - The candidate page to be replaced is the one at the tail of the list. The replacement depends on the reference bit.
  - The page referenced, has its R bit set to 1. If a tail page has it is R bit set, it is reset and moved to the head.
  - If the tail page has its R bit 0, it is replaced, R bit is set to 1 and moved to the head.
  - At page fault at time 20, *A* has *R* bit is reset: it is treated as if it has just been brought in (numbers above pages are reference times), with R bit cleared.

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

circular 2nd chance algorithm

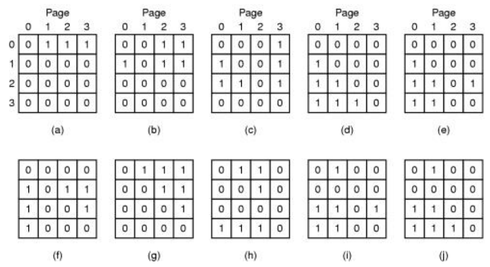
Least Recently Used (LRU)

- It is based on the assumption that the pages used recently will be used again soon
- Write out the page that has been unused for the longest time
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list at <u>every memory reference</u> !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

Least Recently Used (LRU):bit matrix

- LRU maintains an nxn bit matrix in hardware
- On page reference, the corresponding row is set to 1, column to 0
- At any instant the lowest value row is the LRU

n x n LRU in hardware

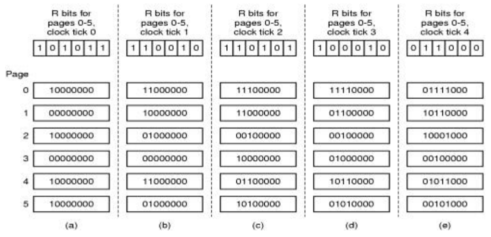

Page
0 1 2 3
(a) 0111 / 0000 / 0000 / 0000
(b) 0011 / 1011 / 0000 / 0000
(c) 0001 / 1001 / 1101 / 0000
(d) 0000 / 1000 / 1100 / 1110
(e) 0000 / 1000 / 1101 / 1100
(f) 0000 / 1011 / 1001 / 1000
(g) 0111 / 0011 / 0001 / 0000
(h) 0110 / 0010 / 0000 / 1110
(i) 0100 / 0000 / 1101 / 1100
(j) 0100 / 0000 / 1100 / 1110

LRU using a matrix – pages referenced in order 0,1,2,3,2,1,0,3,2,3

Simulating LRU in Software (1)

- Hardware may not exist for the architecture
- So a software solution is more practical
- Implementation of a counter that includes the effect of the aging
- Shift the counters right before the R bits are added on a reference
- This is repeated at every clock tick: each time reference bits are cleared, reference counters are shifted…

Simulating LRU in Software(2)

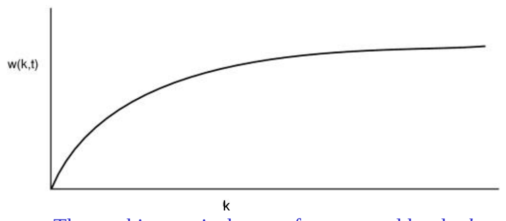


- The aging algorithm simulates LRU in software
- Note 6 pages for 5 clock ticks, (a) – (e)

The Working Set Page Replacement Algorithm (1)

- WS is the set of the pages that are currently in use
- WS page replacing algorithms make use of the locality of the reference
- If the entire WS of a process is in the memory, no page fault occurs until the process moves to a new locality.
- The pages will remain in a WS as long as their reference ® bit is 1.
- Otherwise, the difference between timestamp and the current time is used to decide which one to push out of WS.

The Working Set Page Replacement Algorithm (2)



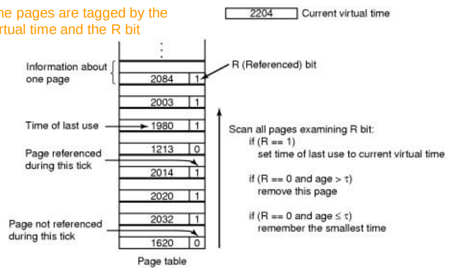
- The working set is the set of pages used by the *k* most recent memory references
- w(k,t) is the size of the working set at time, *t*

WS based PRA

- Which page to be excluded from a WS, in case of a page fault:
- Remember that R and M bits are automatically set
- R bits are cleared periodically
- Each entry in the page table has a time stamp field and an R bit together with other inf about the page…
- On page fault, if the R bit is set, the clock time is written to that entry, otherwise it is candidate for eviction, based on the difference between the time stamp and the current time…

The Working Set Page Replacement Algorithm (3)

The pages are tagged by the virtual time and the R bit


2204
Current virtual time
Information about one page
R (Referenced) bit
2084 1
2003 1
Time of last use
1980 1
1213 0
Page referenced during this tick
2014 1
2020 1
Page not referenced during this tick
2032 1
1620 0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

The working set algorithm

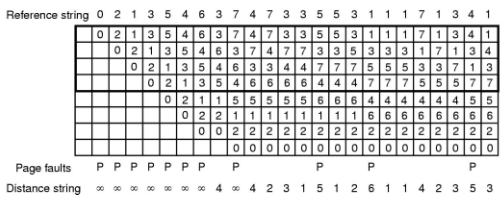
The WSClock Page Replacement Algorithm(1)

- Similar to clock replacement algorithm, the pages are arrange in a circular data structure with a pointer.
- The test is conducted from the pointer on ward, which may remove the page from the WS or keeps it as is, with R bit is set to 0,
- If it is referenced its time is updated.
- If R is 0 and it is not dirty and it is not within the time τ evict it. τ is the age of the page, relative to the current time (=currentTime-timestamp)…
- If it is dirty and outside the window it is put on hold for eviction…

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

Stack Algorithms: 4 frames memory



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	
Distance string		∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

State of memory array *M*, after each item in reference string is processed

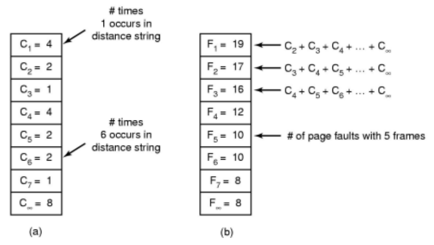
The Distance String in a stack algorithm

- Distance is the number of pages that the referenced page is far from the top of the column, including itself.
- The pages that are not in the memory yet are at a distance of infinity from the top of the table

Predicting the page fault rates

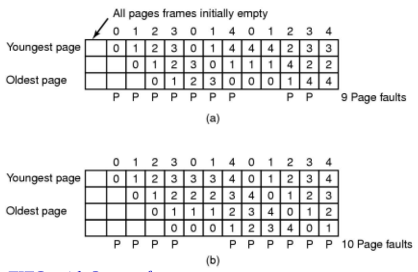
- Distance string can be used to predict the page fault rates.
- Given memory size in number of frames (m) and the distance string, one can compute the number of page faults (F).
- $F_m = \sum C_k + C_{\infty}$, k=m+1,..n
- Where m is memory size, n is the number of virtual pages, and F_m is the number of page faults, C_i is the frequency of number i in a distance string…

Predicting the page fault rates


# times 1 occurs in distance string
$C_1 = 4$
$C_2 = 2$
$C_3 = 1$
$C_4 = 4$
$C_5 = 2$
$C_6 = 2$
$C_7 = 1$
$C_\infty = 8$
# times 6 occurs in distance string
(a)
$F_1 = 19$
$F_2 = 17$
$F_3 = 16$
$F_4 = 12$
$F_5 = 10$
$F_6 = 10$
$F_7 = 8$
$F_\infty = 8$
$C_2 + C_3 + C_4 + \dots + C_\infty$
$C_3 + C_4 + C_5 + \dots + C_\infty$
$C_4 + C_5 + C_6 + \dots + C_\infty$
# of page faults with 5 frames
(b)

- Computation of page fault rate from distance string
  - the *C* vector for the memory size of 4.
  - the *F* vector for different memory sizes:1,2,3,4,5,6,7,8

Belady's Anomaly in FIFO PRAs



All pages frames initially empty

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Oldest page				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 Page faults

(a)

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4	
			0	1	2	2	2	3	4	0	1	2	3	
Oldest page				0	1	1	1	2	3	4	0	1	2	
					0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	10 Page faults

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- *P*'s show which page references show page faults

Modeling Page Replacement Algorithms

- Belady's Anomaly in FIFO algorithms:
  - More frames more faults!
- Stack algorithms: all other algorithms
  - More frames less faults

Design Issues for Paging Systems

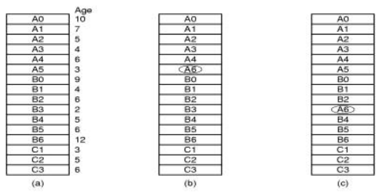
Local versus Global Allocation Policies

- Local page replacement: the replacement is local to the process, ie., one of the pages of the process causing page fault is replaced.
- Global page replacement: the oldest page in the system is replaced

Design Issues for Paging Systems

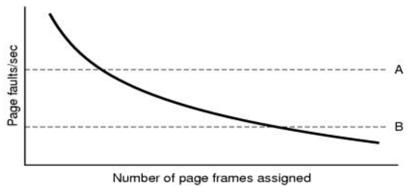
- Local vs Global allocation policies
- Thrashing is eliminated using load control, externally
- Page size can be optimized
- Separate instruction and data spaces
- Cleaning policy
- Sharing memory pages

Local versus Global Allocation Policies (1)



a) Original configuration
b) Local page replacement
c) Global page replacement

Local versus Global Allocation Policies (2)



- Page fault rate as a function of the number of page frames assigned
- A: high page fault rate, B: Low page fault rate

Load Control

- Despite good designs, system may still thrash
- Solution :
  Reduce number of processes competing for memory
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming

Advantages of small page size vs disadvantages

- 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

Minimization of overhead: Optimum Page Size

- Overhead due to page table and internal fragmentation

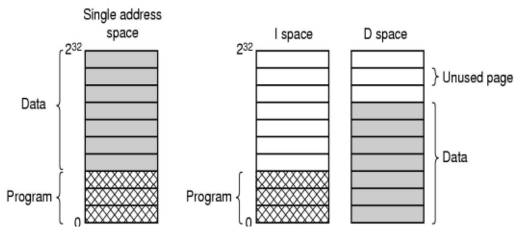
$$overhead = \underbrace{\frac{s \cdot e}{p}}_{\text{page table space}} + \underbrace{\frac{p}{2}}_{\text{internal fragmentation}}$$

- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = size of each entry in the PT in bytes

Optimized when

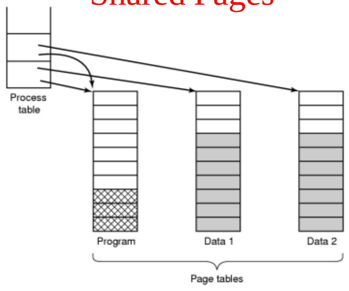
$$p = \sqrt{2se}$$

Separate Instruction and Data Spaces may be preferred



1. Program and data address spaces together: one address space
2. Program address space
3. Data address spaces

Shared Pages



- Two processes may share the same program related page table with separate data and related page tables

Cleaning Policy: maintenance of an acceptable level of free frames

- Need for a background process, paging daemon
  - periodically inspects state of memory to maintain an acceptable level of available pages
- When too few frames are free
  - selects pages to evict using a replacement algorithm
- Implementation example:
  - use same clock type page handling: two handed clock, front hand does cleaning, back hand does replacement..

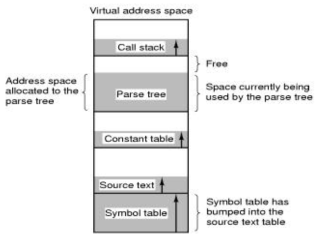
Further Reading

Segmentation

Segmentation vs. Linear mempory

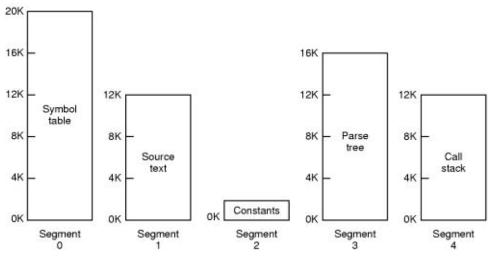
- So far virtual memory was linear, but frames were in ad-hoc positions
- For many problems two or more memory address spaces are more convenient.
- Compilers are good examples for this with many unrelated components such as text, symbol table, parse tree, etc.

Segmentation (1)


Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

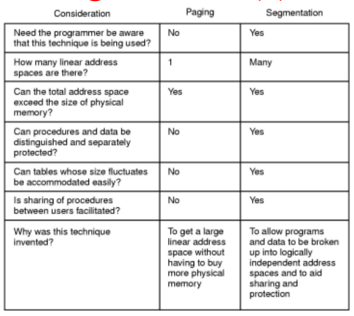
- With one-dimensional address space with growing tables, one table may bump into another

Segmentation (2)



Allows each table to grow or shrink, independently

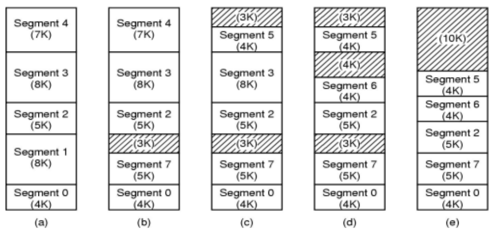
Segmentation (3)



Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

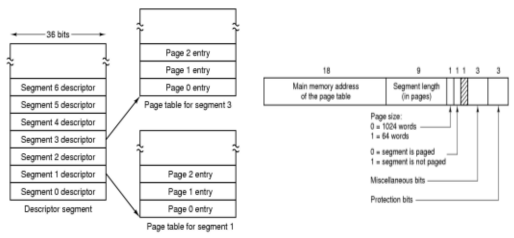
Comparison of paging and segmentation

Implementation of Pure Segmentation



(a)-(d) Development of checkerboarding
(e) Removal of the checkerboarding by compaction

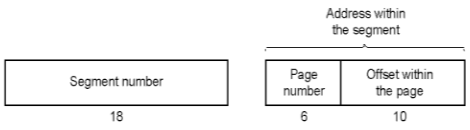
Segmentation with Paging: MULTICS (1)



- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths

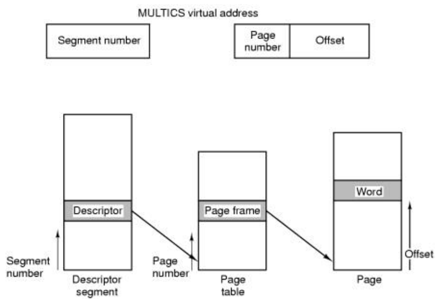
- Segment page table may not be in the memory
- Page referenced may not be in the memory
- TLB is used to keep the address of the most recently used pages

Segmentation with Paging: MULTICS (2)



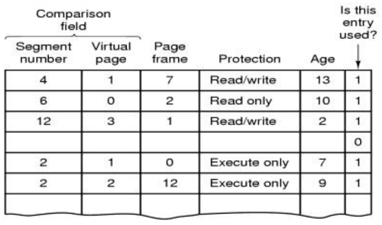
A 34-bit MULTICS virtual address

Segmentation with Paging: MULTICS (3)



Conversion of a 2-part MULTICS address into a main memory address

Segmentation with Paging: MULTICS (4)

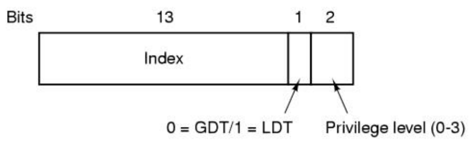


Segment number	Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Comparison field

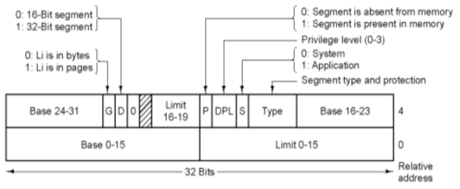
- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated

Segmentation with Paging: Pentium (1)



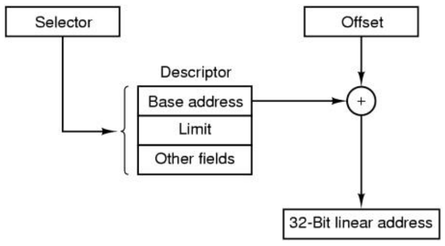
A Pentium selector

Segmentation with Paging: Pentium (2)



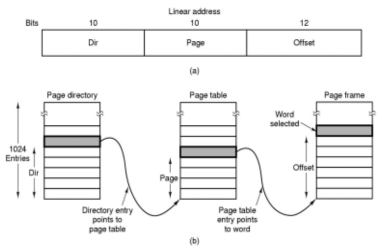
- Pentium code segment descriptor
- Data segments differ slightly

Segmentation with Paging: Pentium (3)



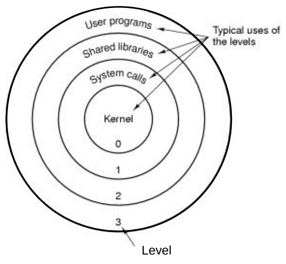
Conversion of a (selector, offset) pair to a linear address

Segmentation with Paging: Pentium (4)



Mapping of a linear address onto a physical address

Segmentation with Paging: Pentium (5)



Protection on the Pentium