# Chapter 4

# Memory Management

4.1 Basic memory management

4.2 Swapping

4.3 Virtual memory

4.4 Page replacement algorithms

4.5 Modeling page replacement algorithms

4.6 Design issues for paging systems

4.7 Implementation issues

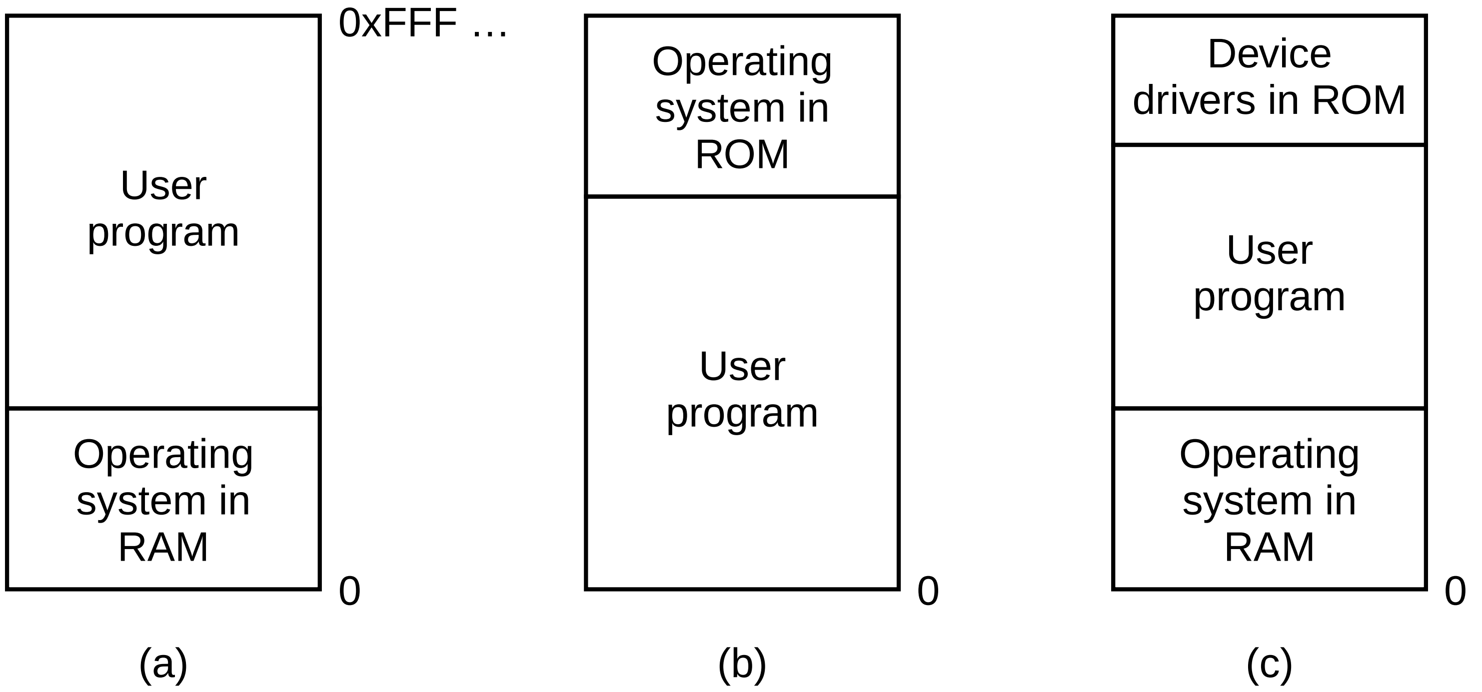
4.8 Segmentation

# Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast,expensive memory - cache
  - some medium-speed,medium price main memory
  - gigabytes of slow,cheap disk storage
- Memory manager handles the memory hierarchy

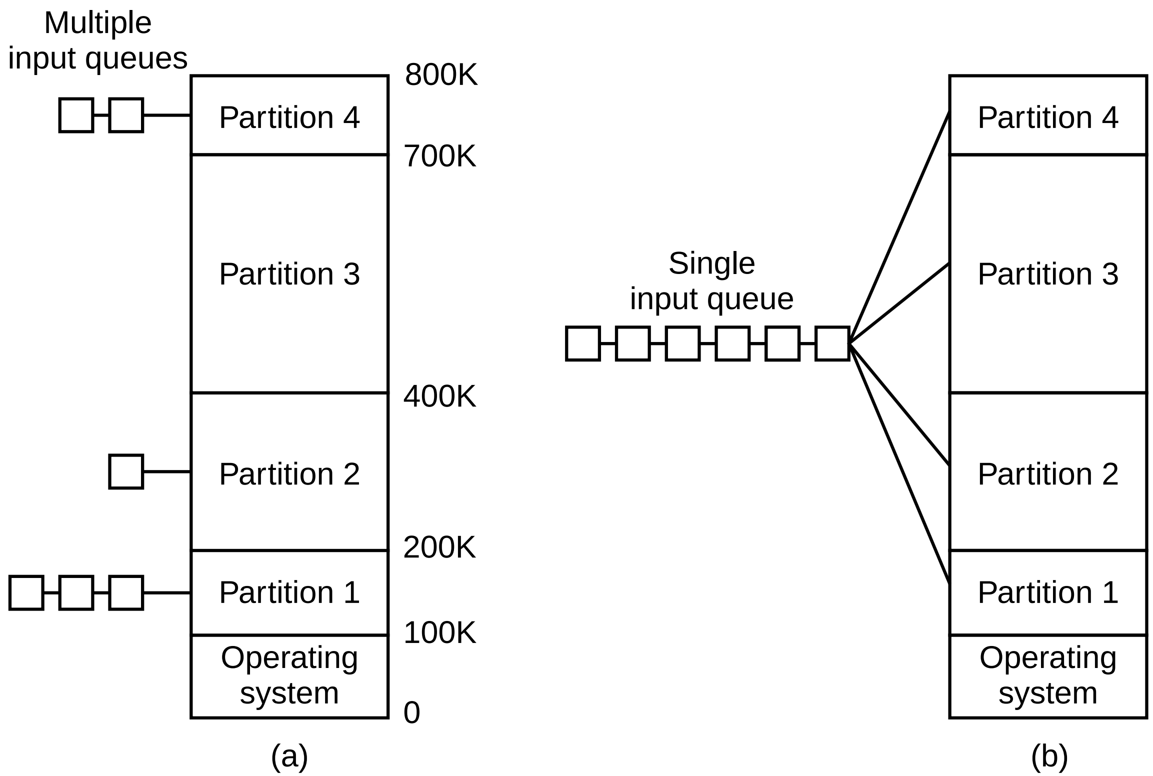
# Basic Memory Management

## Monoprogramming without Swapping or Paging




- Three simple ways of organizing memory
  - an operating system with one user process

# Multiprogramming with Fixed Partitions

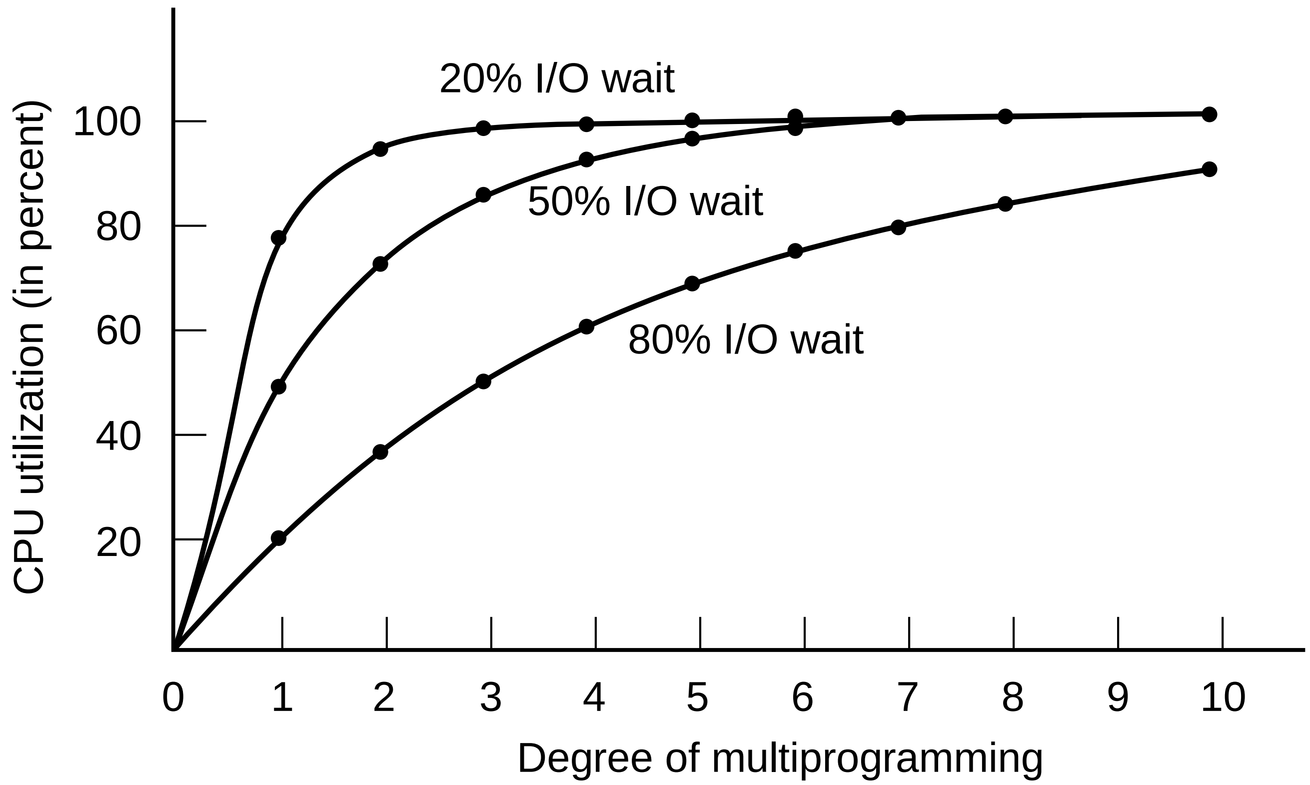



- Fixed memory partitions

  **(a)** separate input queues for each partition

  **(b)** single input queue

# Modeling Multiprogramming




CPU utilization as a function of number of processes in memory

$CPUutil = 1 - p^n$, where $n$: # Processes, $p$: I/O Wait Prob.

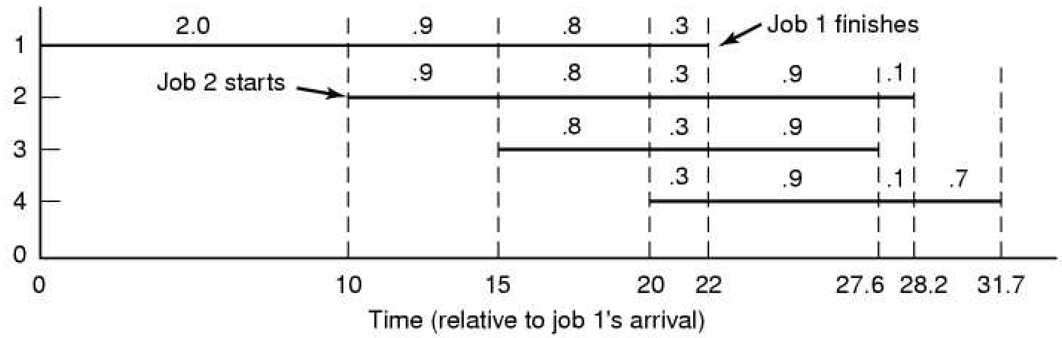
# Analysis of Multiprogramming System Performance

| Job | Arrival time | CPU minutes needed |
|---|---|---|
| 1 | 10:00 | 4 |
| 2 | 10:10 | 3 |
| 3 | 10:15 | 2 |
| 4 | 10:20 | 2 |

(a)

| | # Processes 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| CPU idle | .80 | .64 | .51 | .41 |
| CPU busy | .20 | .36 | .49 | .59 |
| CPU/process | .20 | .18 | .16 | .15 |

(b)




(c)

(a) Arrival and work requirements of 4 jobs

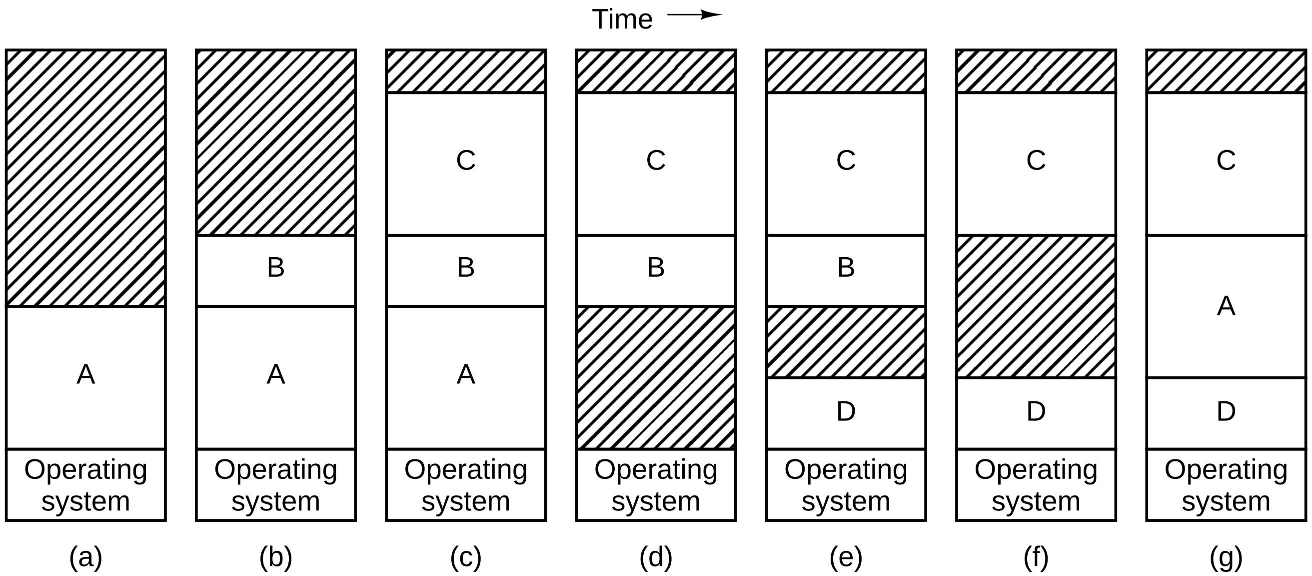
(b) CPU utilization for 4 jobs with 80% I/O wait

(c) Sequence of events as jobs arrive and finish
(numbers show amount of CPU time jobs get in each interval)

# Relocation and Protection

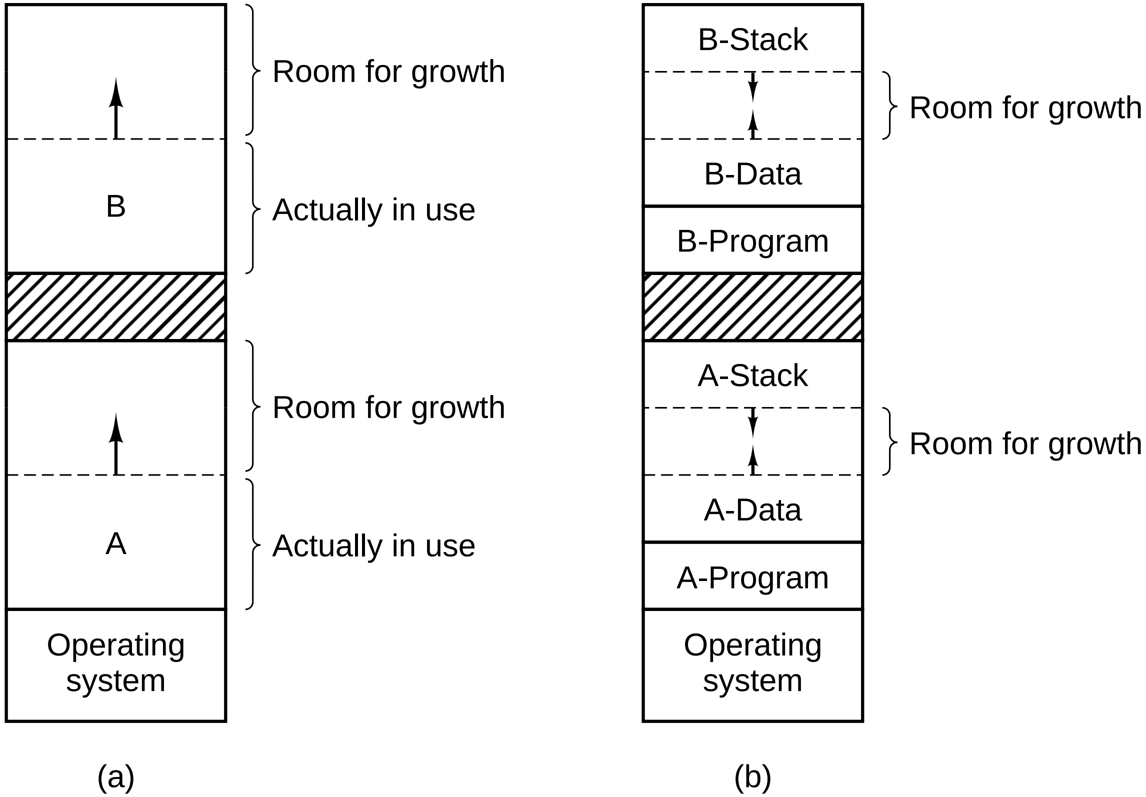
- Cannot be sure where program will be loaded in memory
  - address locations of variables,code routines cannot be absolute
  - must keep a program out of other processes' partitions
- Use base and limit values
  - address locations added to base value to map to physical addr
  - address locations larger than limit value is an error

# Swapping (1)



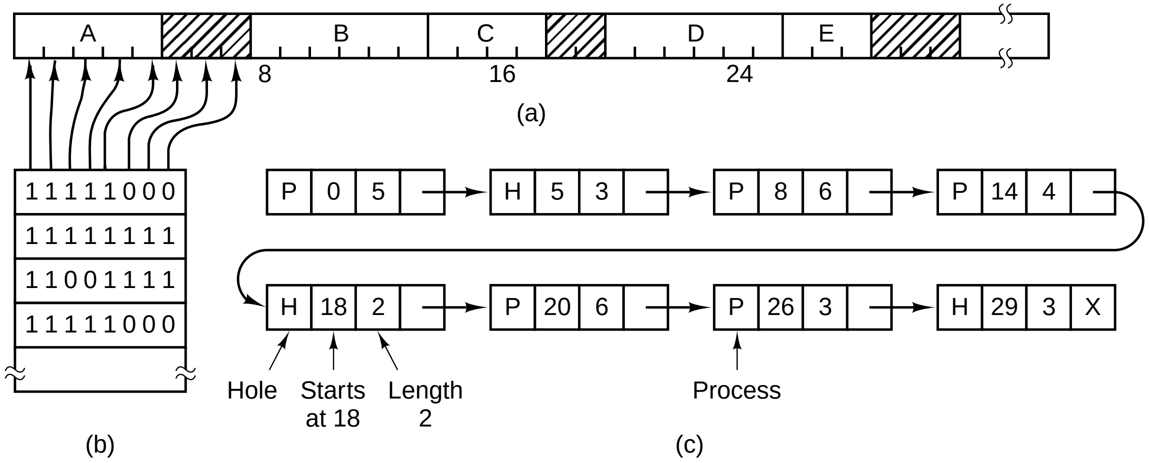

- Memory allocation changes as
  - processes come into memory
  - leave memory
- Shaded regions are unused memory

# Swapping (2)




- Allocating space for growing data segment
- Allocating space for growing stack & data segment

# Memory Management with Bit Maps



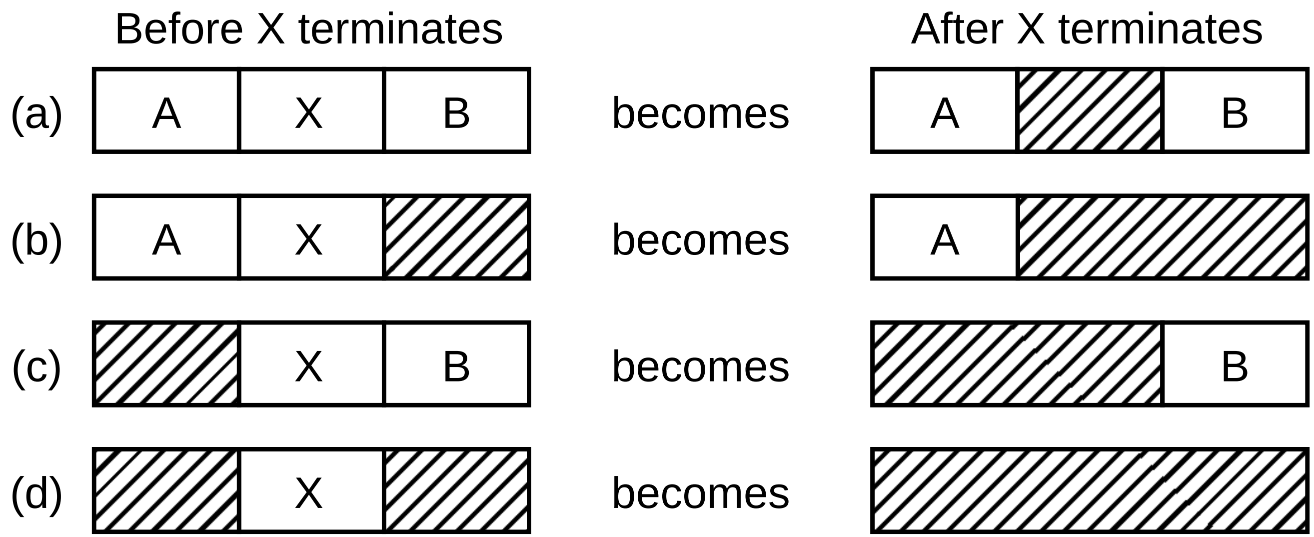

(a) Part of memory with 5 processes,3 holes

- tick marks show allocation units
- shaded regions are free

(b) Corresponding bit map

(c) Same information as a list

# Memory Management with Linked Lists



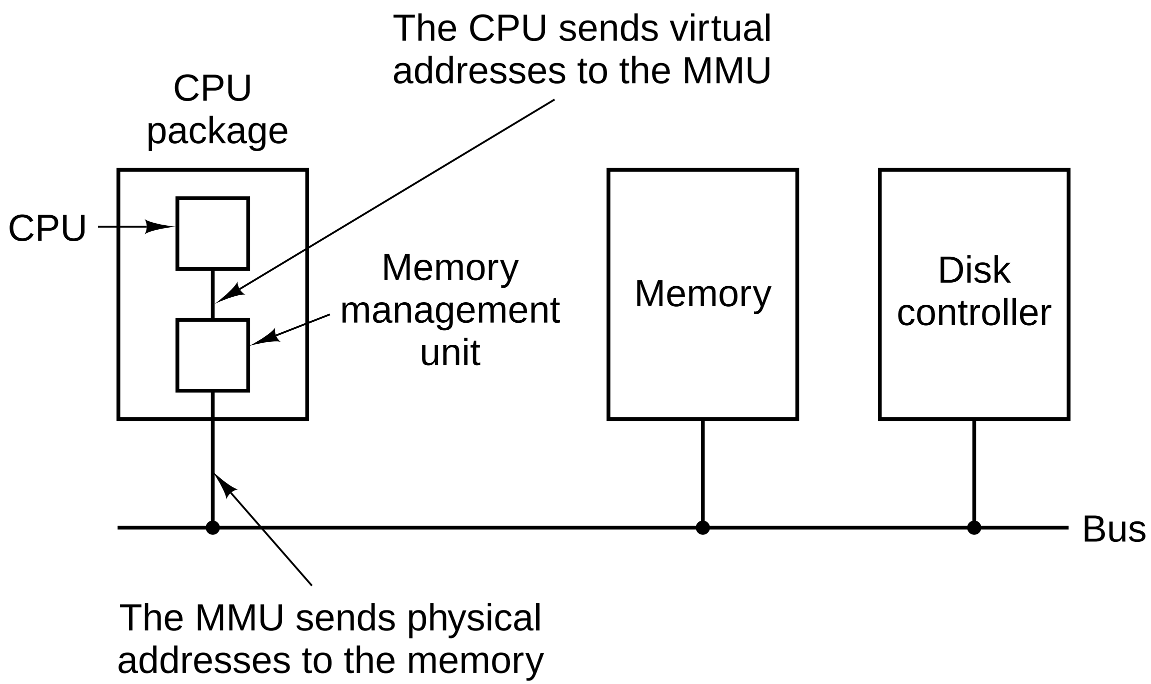

Four neighbor combinations for the terminating process X

## Basic Allocation Algorithms

- Best Fit
- First Fit
- Next Fit
- Worst Fit
- Quick Fit

# Virtual Memory

## Paging (1)




The position and function of the MMU

Keywords: Virtual and physical addresses (spaces), paging, overlay

# Paging (2)

The relation between virtual addresses and physical memory addresses given by page table

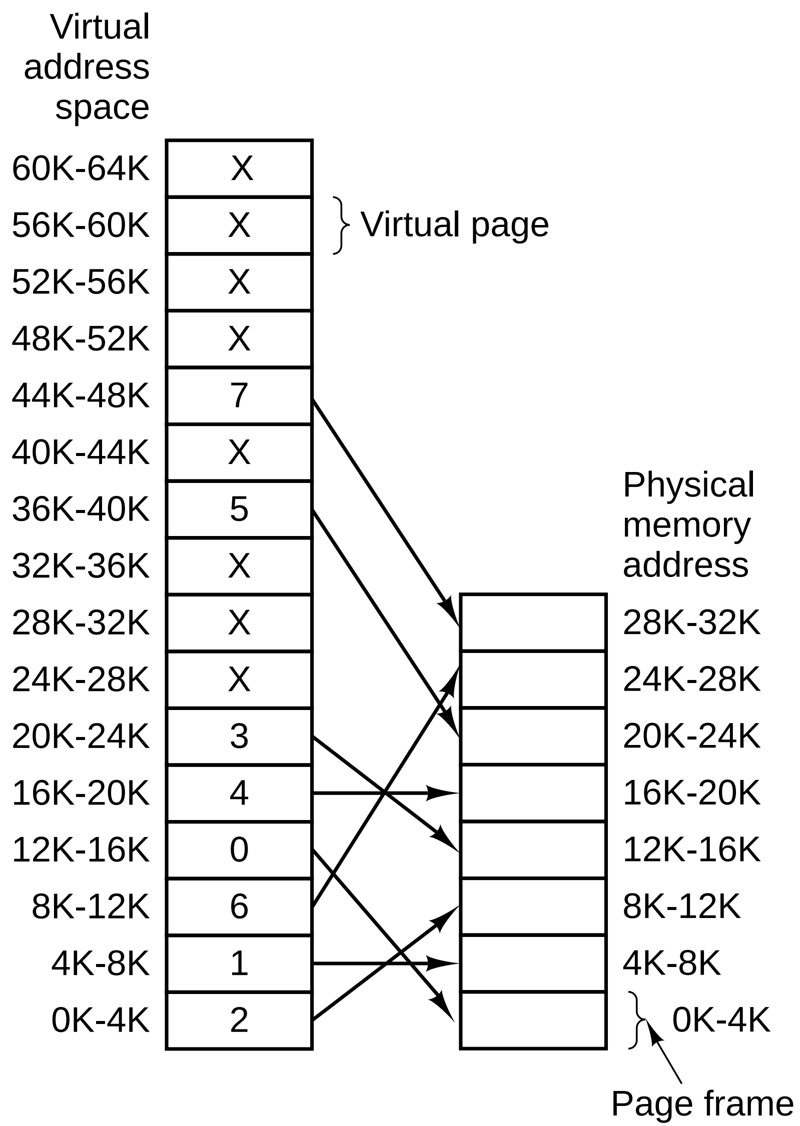
64KB Virtual Address

32KB Physical Memory

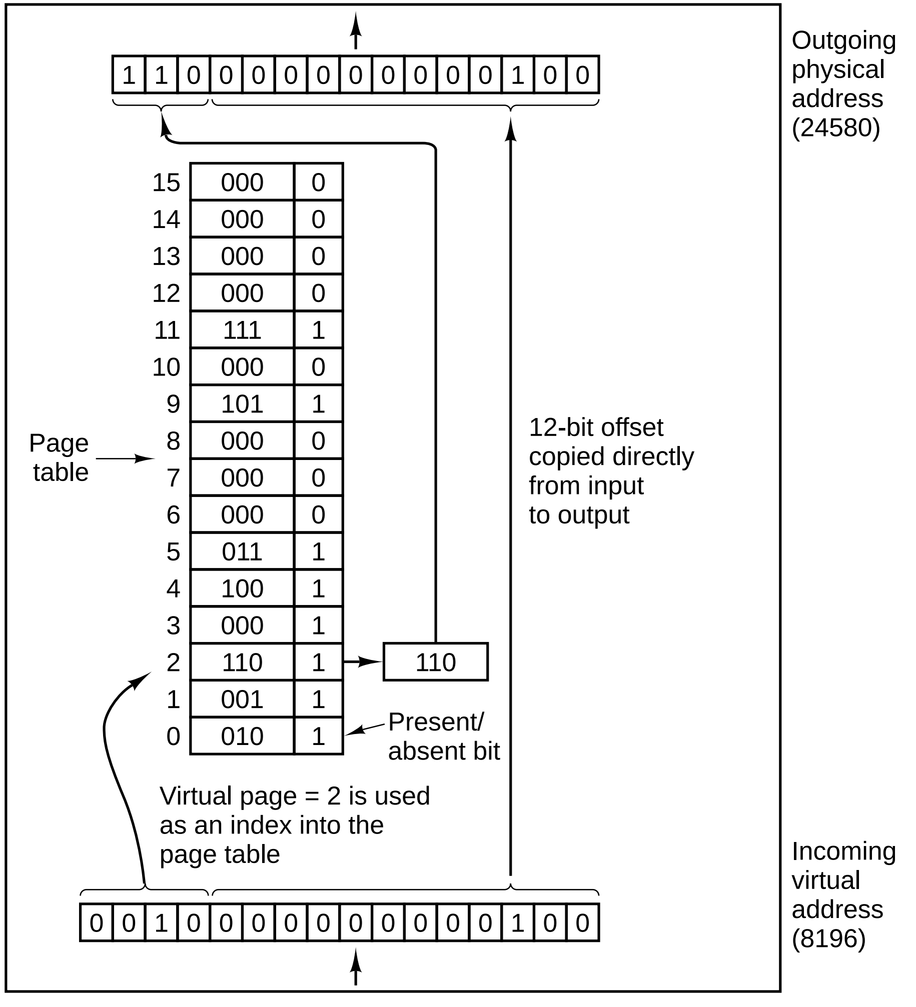
4KB Page Size

Mapping Example:

```
V: MOV R0, 8192
P: MOV R0, 24576
```

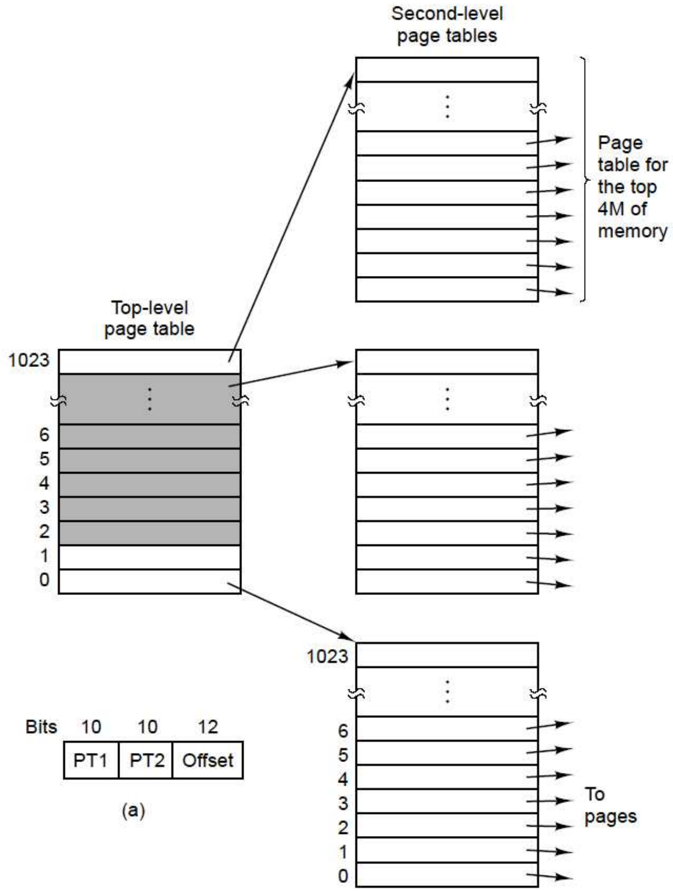


# Page Tables (1)




Internal operation of MMU with 16 pages

# Page Tables (2)




32-bit address space

4KB page size

Page Table Overhead

Assumption:

(Only 3K pages are used)

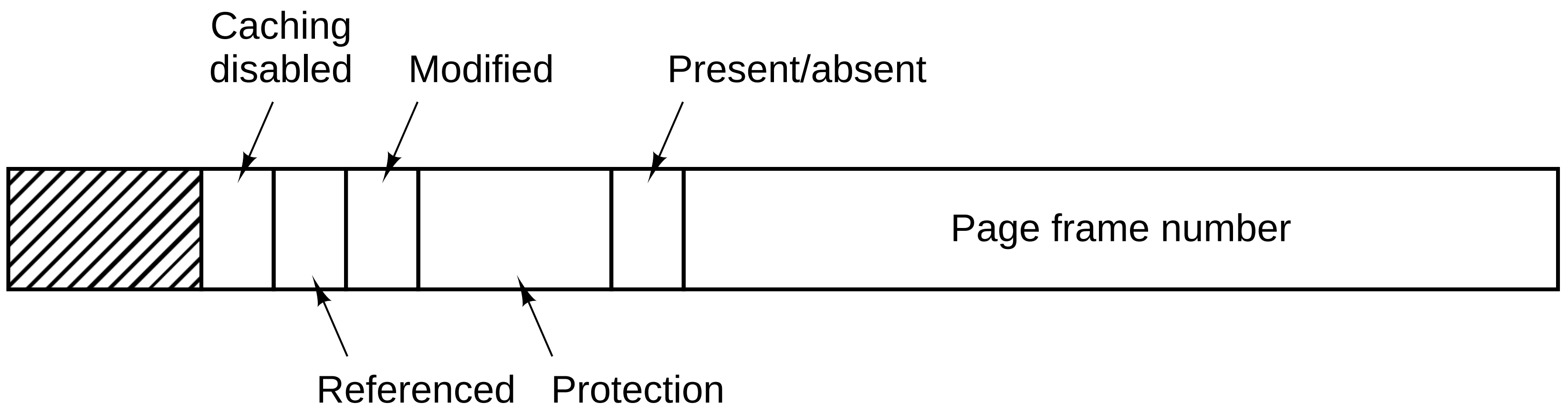
**1-Level:** $2^{20} = 1M$ entries

**2-Levels:** $2^{10}$ (Level 1) +

$3 \times 2^{10}$ (Level 2)

$=$ 4K entries

# Page Tables (3)




Typical page table entry

# TLBs - Translation Lookaside Buffers

**Where to store the page table ?**

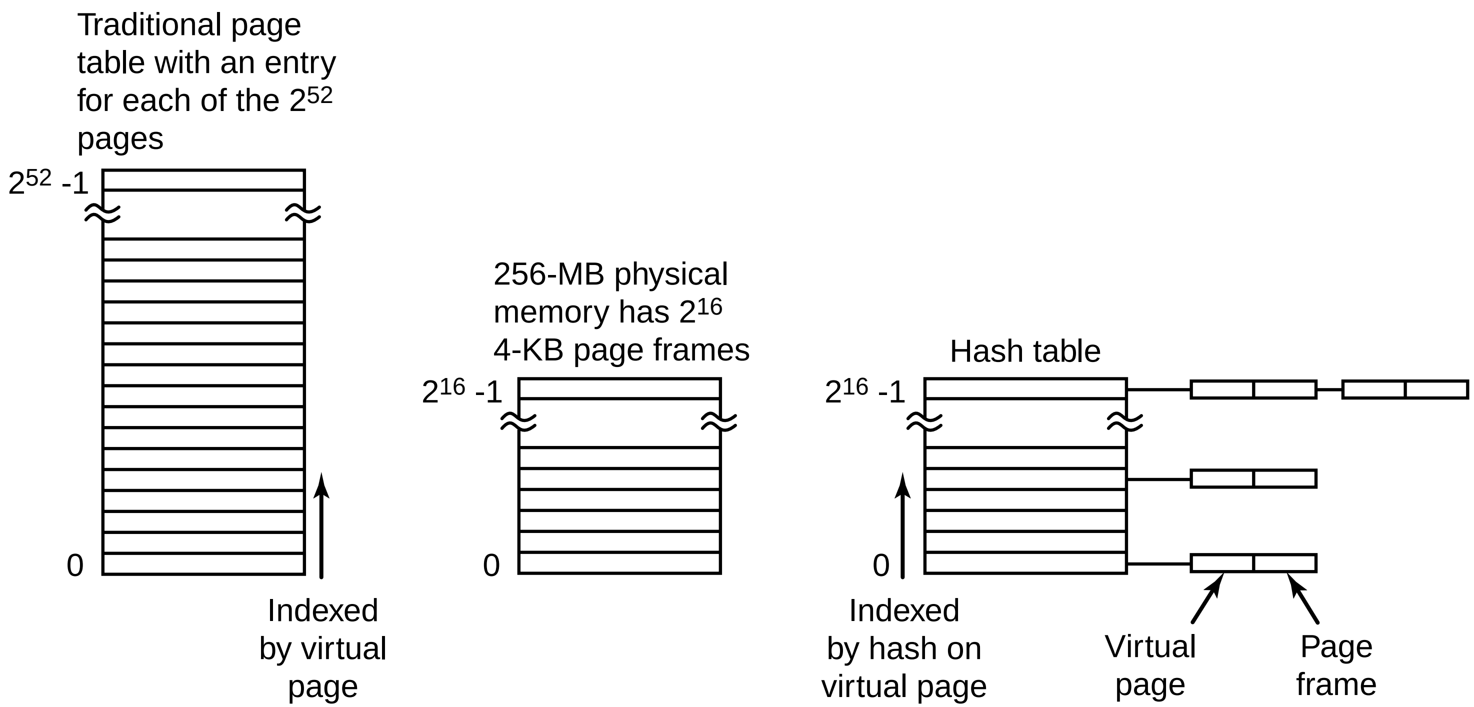
**Special hardware registers:** too large to store entire page table

**Main memory:** doubles the number of memory access
(also the page table itself may be swapped out).

| 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 by access locality

# Inverted Page Tables




Comparison of a traditional page table ($V \rightarrow P$) with an inverted page table ($V \rightarrow P$). Hash table to accelerate the search.

# Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified pages must be written back to the disk
  - unmodified pages are just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

# Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by ...
  - logging page use on previous runs of process
  - although this is impractical

# Not Recently Used Page Replacement Algorithm

- Each page has Reference bit,Modified bit
  - bits are set when page is referenced,modified
- Pages are classified in four classes

**0** not referenced, not modified

**1** not referenced, modified (previously Class 3)

**2** referenced, not modified

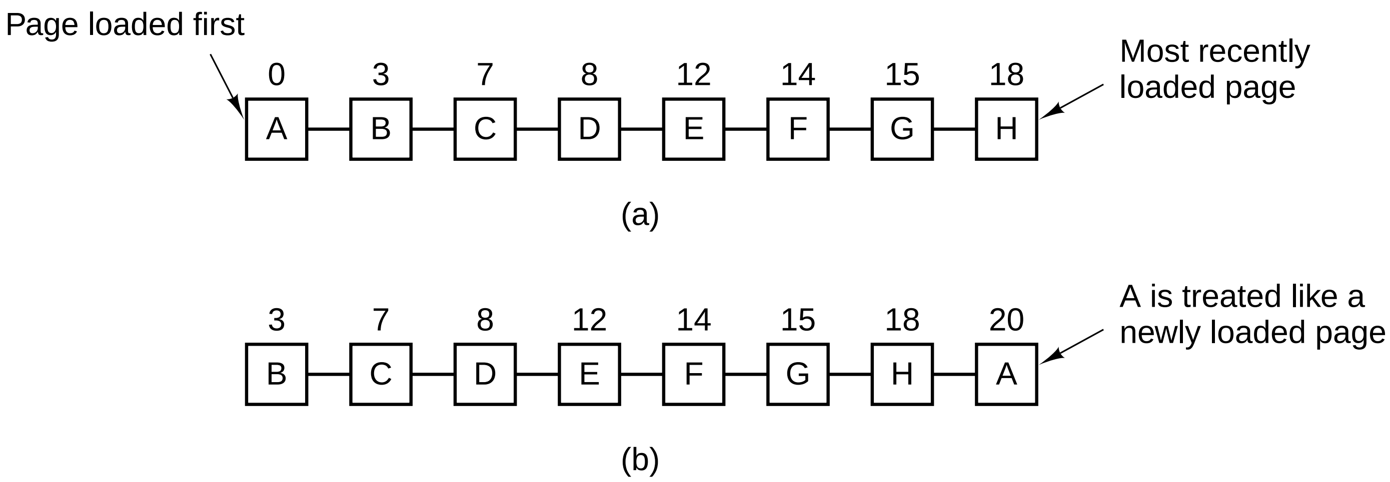
**3** referenced, modified

- NRU removes page at random
  - from lowest numbered non empty class
  - prefers not referenced pages to not modified

# FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - page in memory the longest may be often used
    (the first access determines the position of the page in the list).

# Second Chance Page Replacement Algorithm



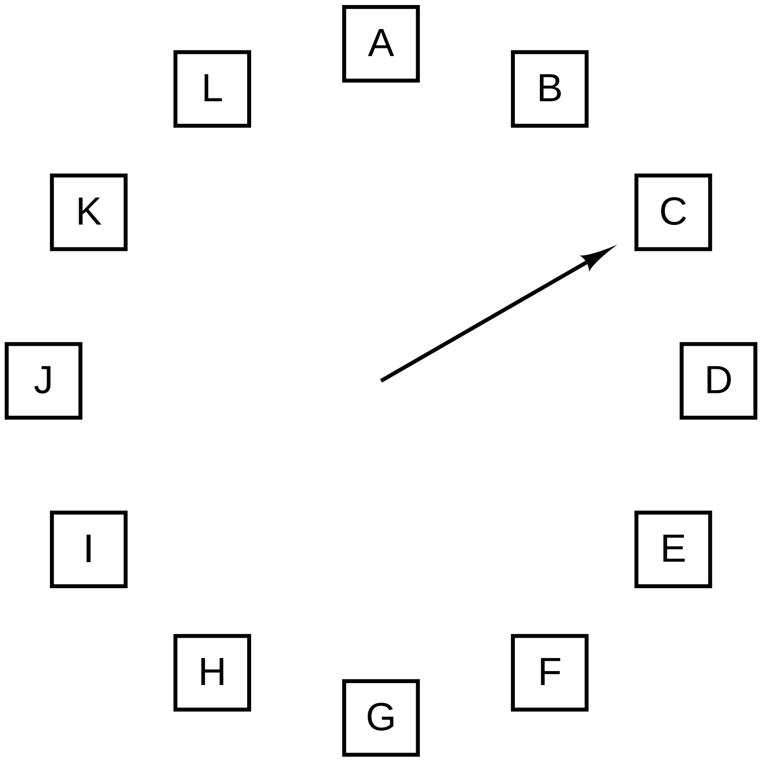

- Operation of a second chance: same as FIFO but skips the pages with $R = 1$ for the first round (R-bits are cleared and loading times are updated when skipped) .

  **(a)** Pages sorted in FIFO order

  **(b)** Page list if fault occurs at time 20, $\underline{A}$ has $R$ bit set (numbers above pages are loading times)

# The Clock Page Replacement Algorithm

Eliminate the cost of list manipulation in the Second Chance 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

# Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time
- Must keep a linked list of pages
  - most recently used at front,least at rear
  - update this list every memory reference !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter (must find it !)
  - periodically zero the counter

# Simulating LRU in Software (1)

Page

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |

(a)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

(b)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(c)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(d)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(e)

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 |

(f)

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(g)

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(h)

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(i)

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(j)

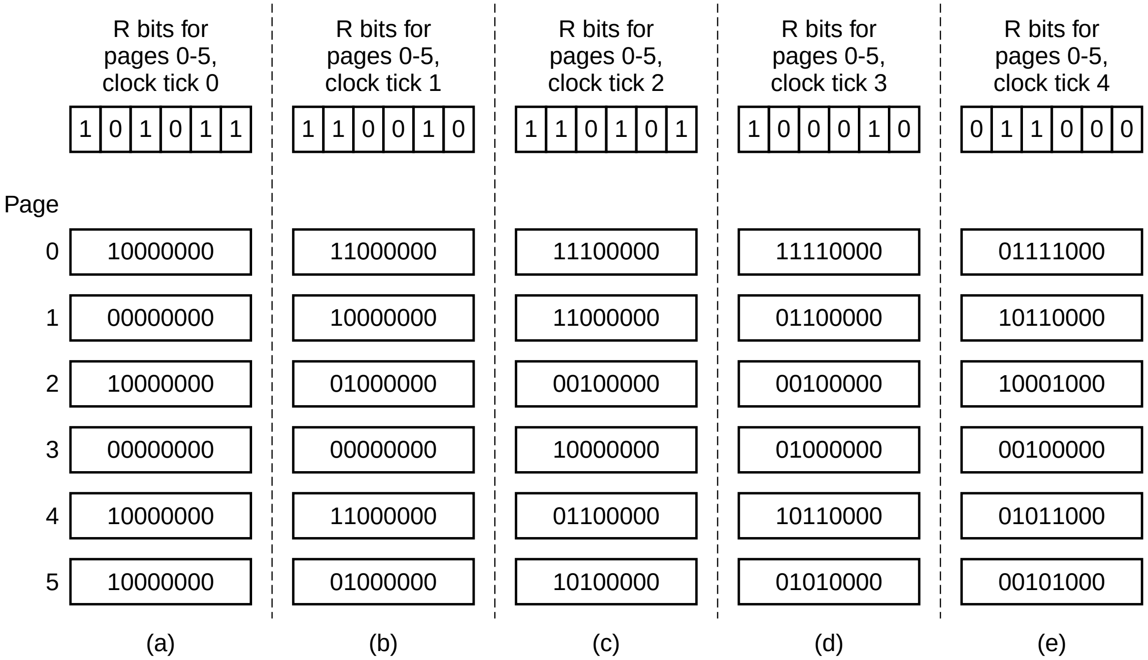
LRU using a matrix

For an access to page $i$, set $i$-th column and clear $i$-th row

Choose the least numbered page (take the row as a binary number)

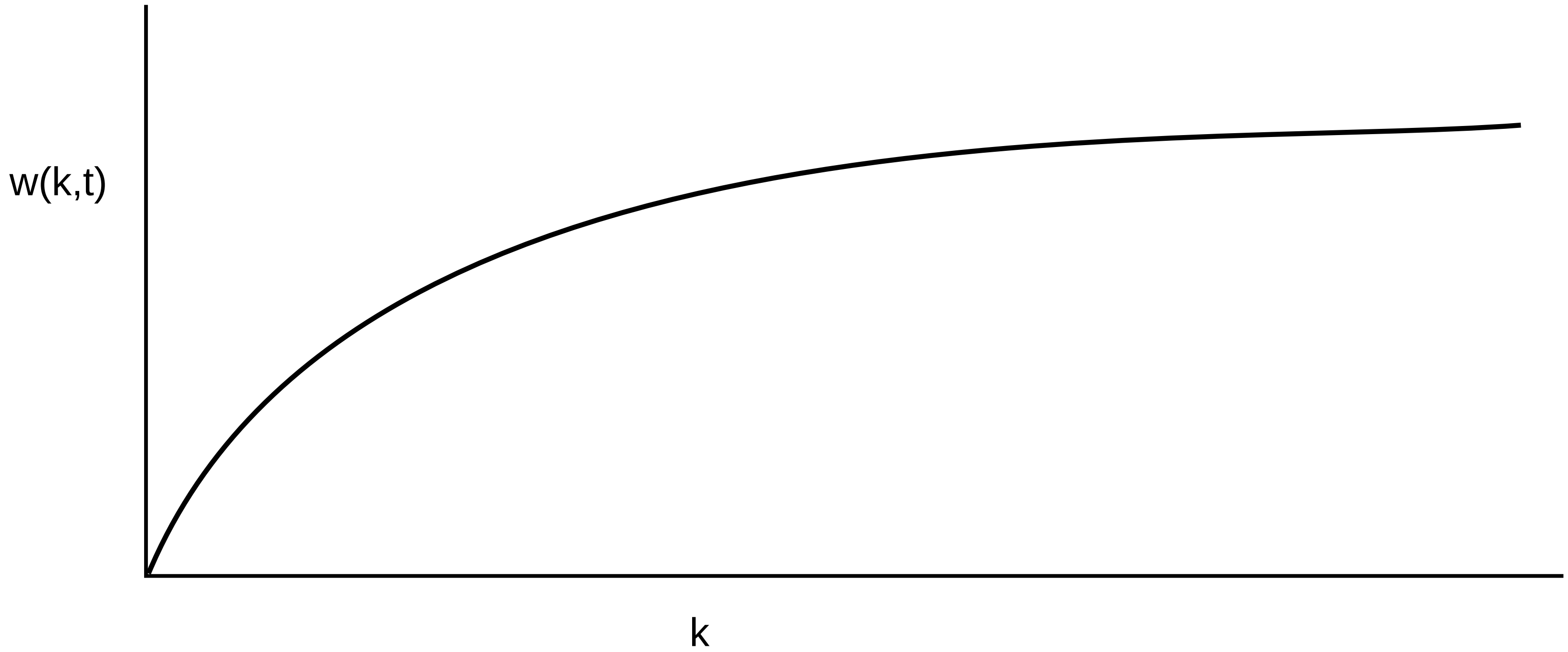
Example: pages referenced in order 0,1,2,3,2,1,0,3,2,3

# Simulating LRU in Software (2)



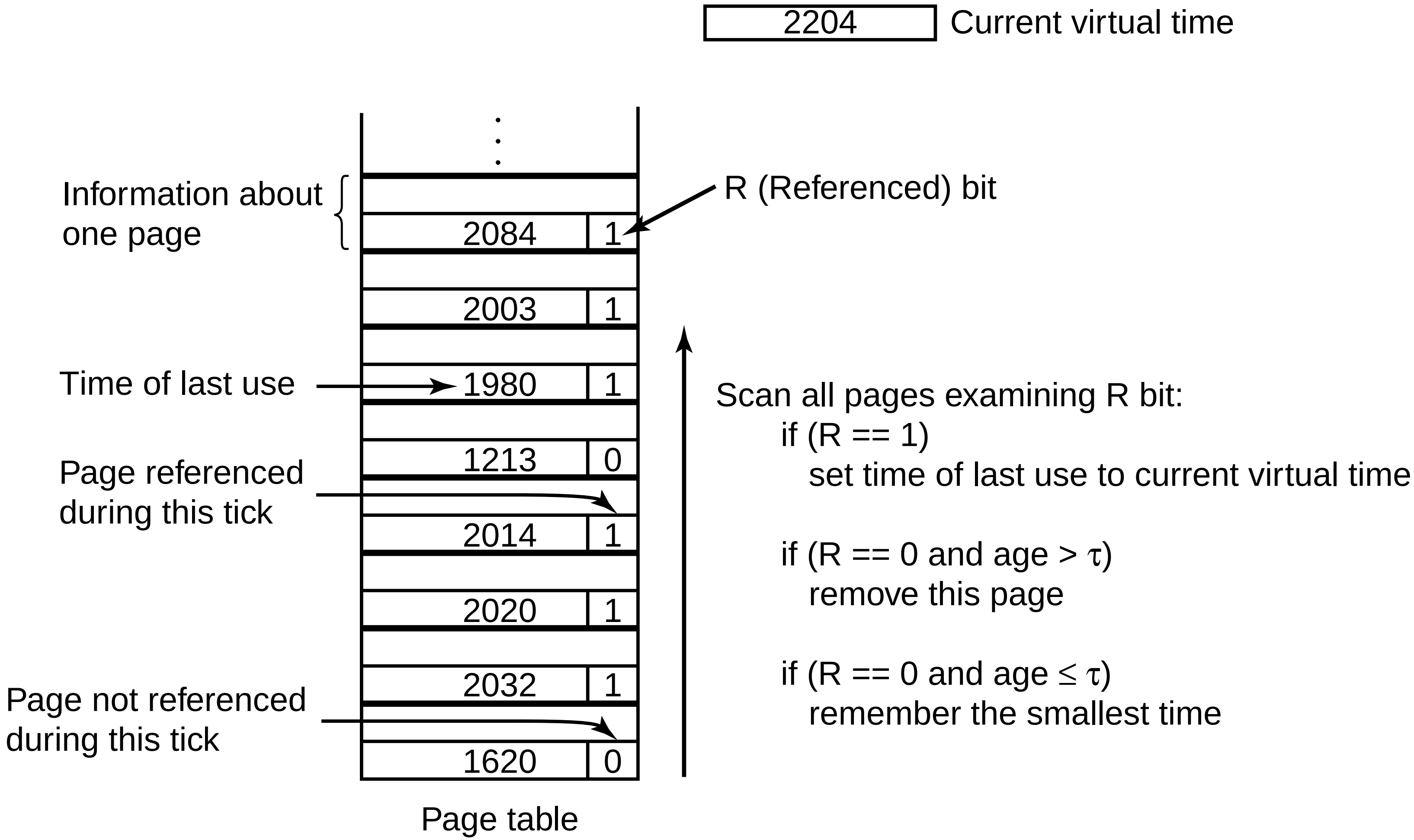

- The aging algorithm simulates LRU in software
  Shift-in the R-bits from left at every clock tick.
- Example: 6 pages for 5 clock ticks, (a) - (e)
- Not Real LRU: access order within a clock tick is lost and counters have a limited length.

# The Working Set Page Replacement Algorithm (1)



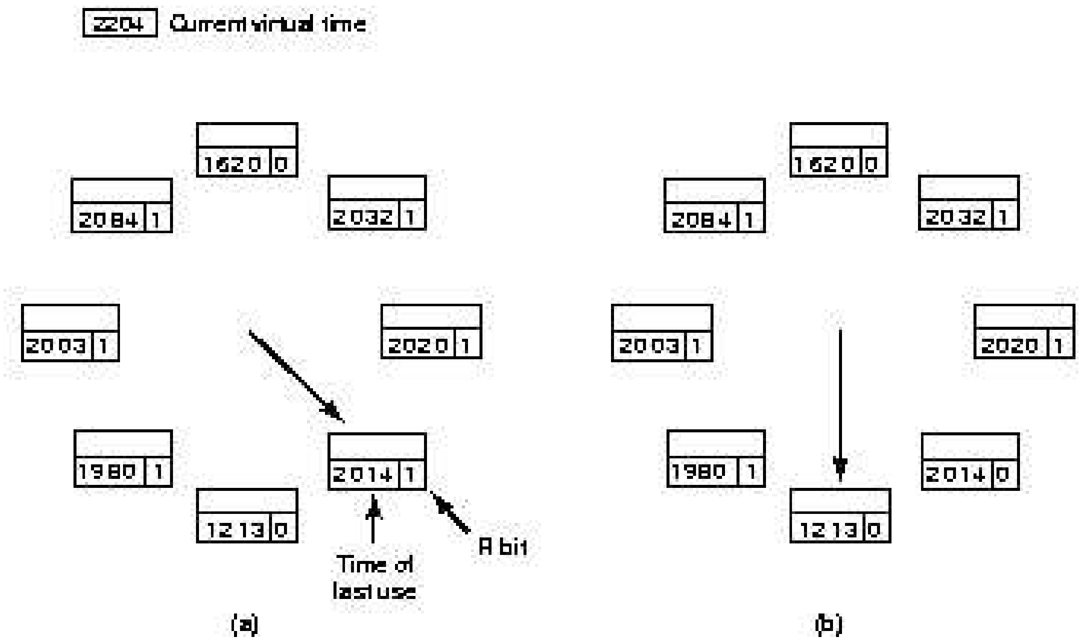

- The working set is the set of pages used by the $k$ most recent memory references (locality of access)
- w(k,t)is the size of the working set at time, $t$
- Evict a page not in the working set on a page fault

# The Working Set Page Replacement Algorithm (2)




The working set algorithm. Pages accessed within $\tau$ virtual time are considered to be in the working set.

# The WS Clock Page Replacement Algorithm (1)

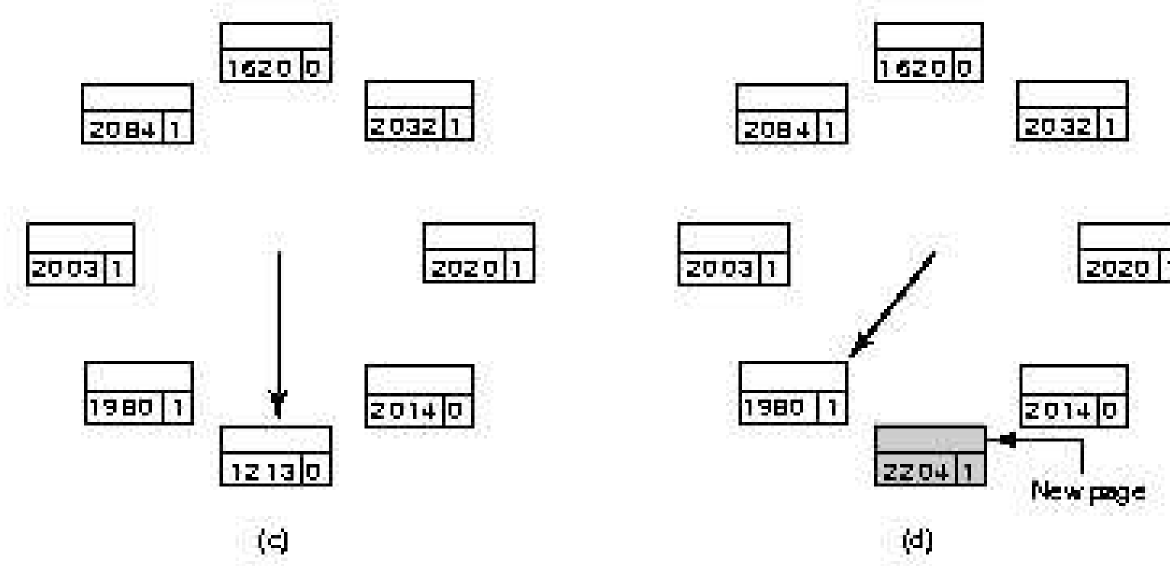



Operation of the WS clock algorithm

(a) Not replaced since R = 1

(b) R-bit cleared and the hand is advanced

# The WS Clock Page Replacement Algorithm (2)




Operation of the WS clock algorithm

(c) Page selected for replacement since R = 0

(d) R-bit is set and access time is updated and the hand is advanced

# 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 Belady's Anomaly

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
- 32 $P$'s show which page references show page faults

# Stack Algorithms

| 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 |

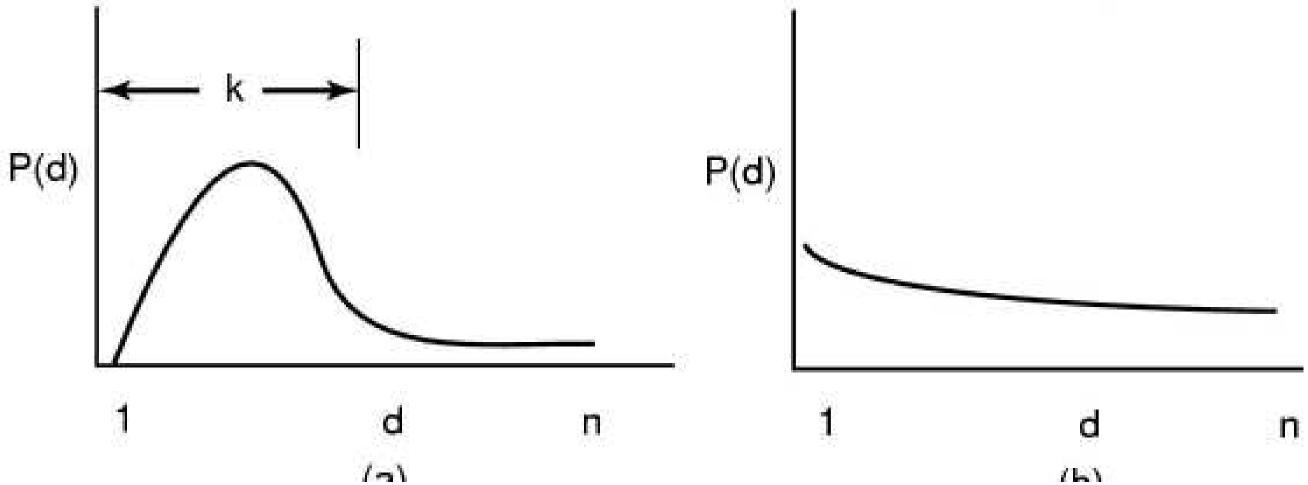
State of memory array, $M$ ,after each item in reference string is processed

$M(m, r) \subseteq M(m{+}1, r)$

m: number of frames

r: index into reference string

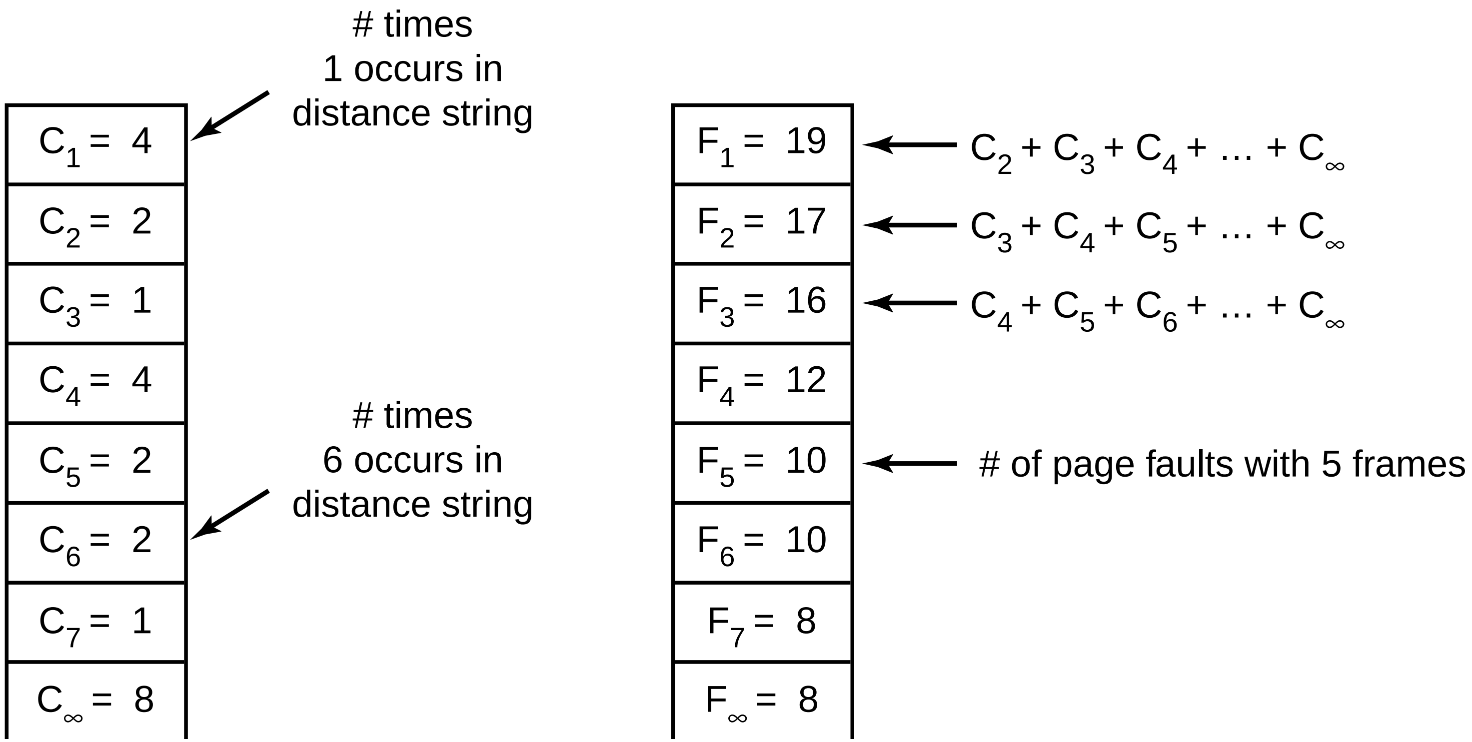
# The Distance String




Probability density functions for two hypothetical distance strings

d: distance of accessed page from top of stack

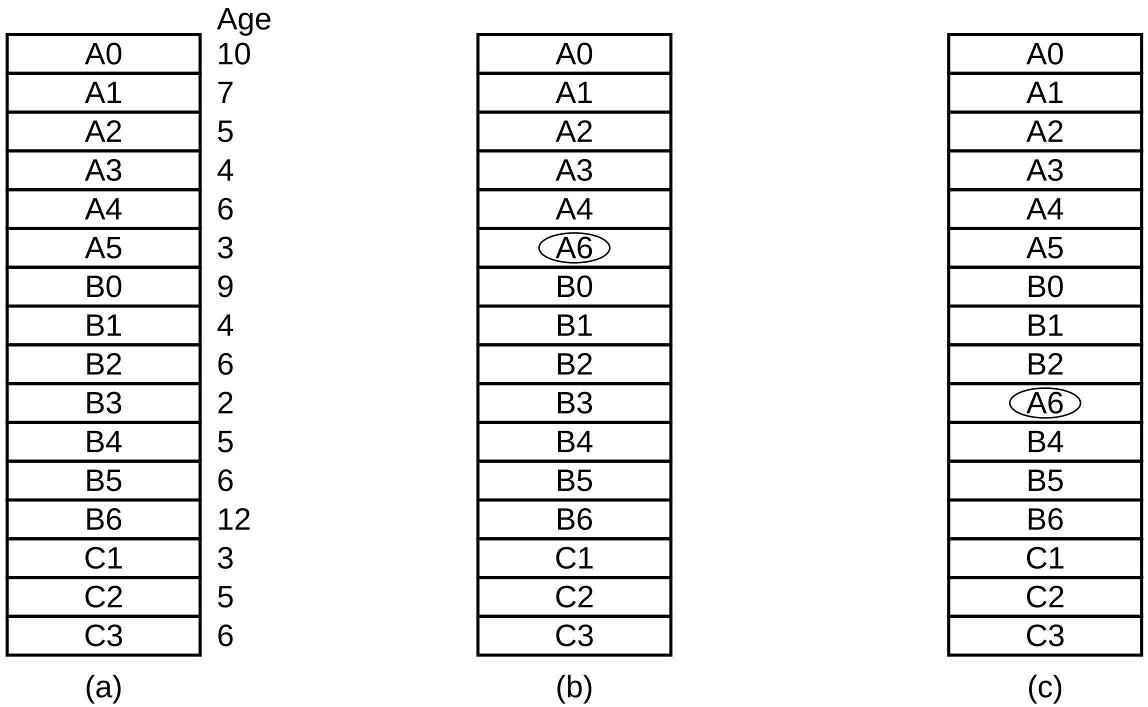
# The Distance String




- Computation of page fault rate from distance string
  - the $C$ vector
  - the $F$ vector

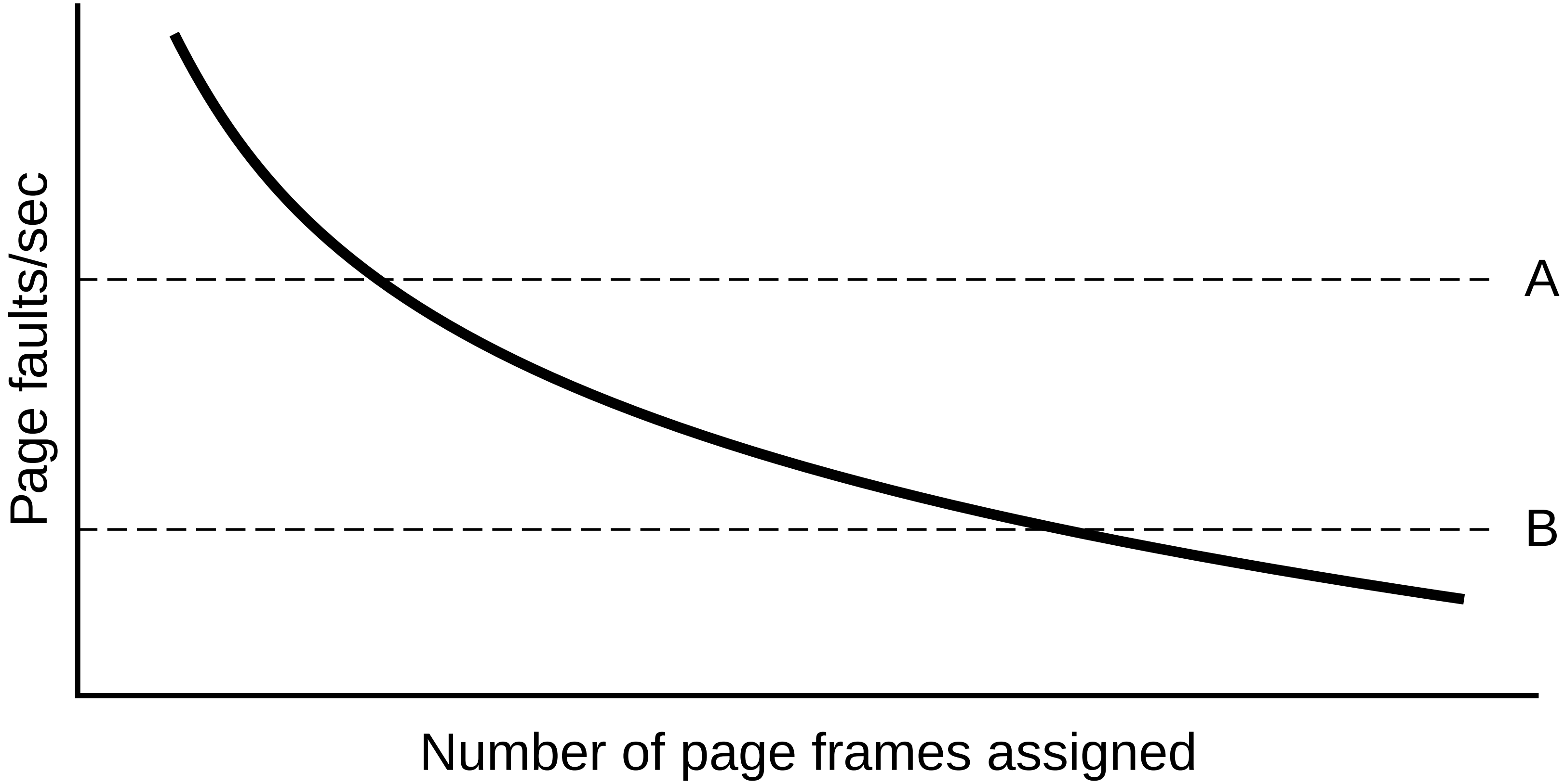
# Design Issues for Paging Systems

## Local versus Global Allocation Policies (1)




- Original configuration
- Local page replacement
- Global page replacement

# Local versus Global Allocation Policies (2)




Page fault rate as a function of the number of page frames assigned

A: pagefault rate too high

B: too much memory allocated

# Load Control

- Despite good designs,system may still thrash
- When PFF(Page Fault Frequency) algorithm indicates
  - some processes need more memory
  - but no processes need less
- Solution :
  Reduce number of processes competing for memory
  - swap one or more to disk,divide up pages they held
  - reconsider degree of multiprogramming

# Page Size (1)

## 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 Size (2)

- Overhead due to page table and internal fragmentation

$$overhead = \frac{s * e}{p} + \frac{p}{2}$$

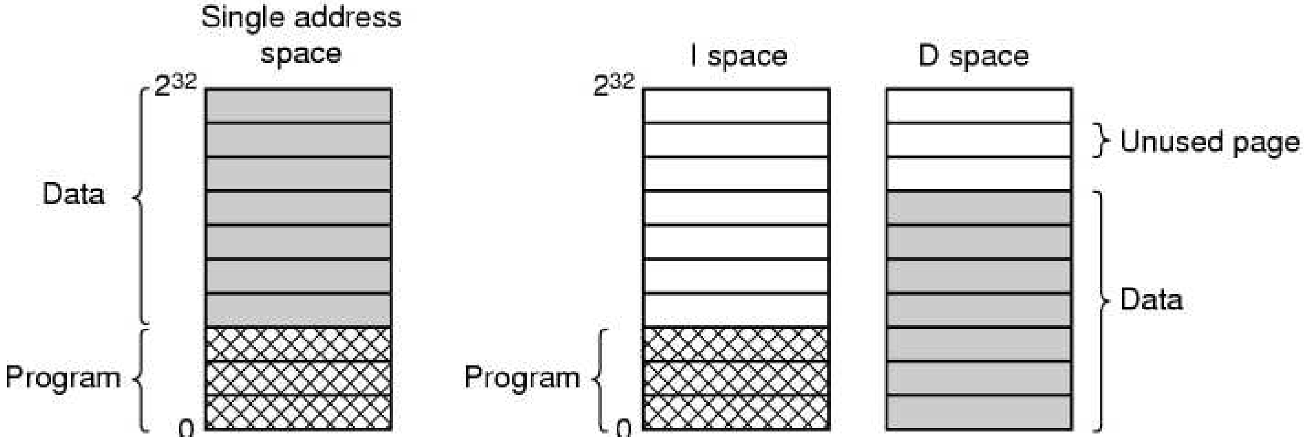
$\frac{s * e}{p}$: page table pace, $\frac{p}{2}$: internal fragmentation

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

Optimized when : $p = \sqrt{2se}$

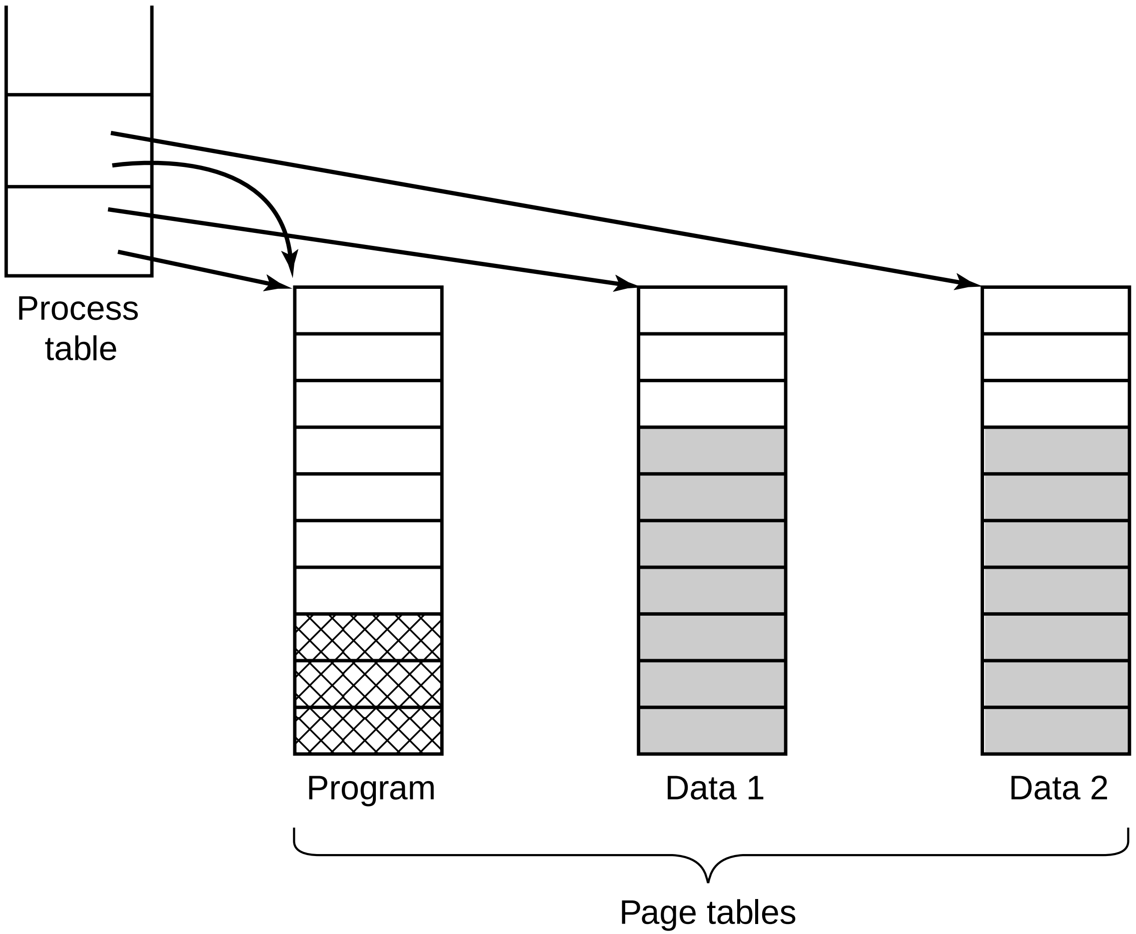
$\frac{d}{dp} overhead = 0$

# Separate Instruction and Data Spaces




- One address space
- Separate I and D spaces

# Shared Pages




Two processes sharing same program sharing its page table

# Cleaning Policy

- Need for a background process,paging daemon
  - periodically inspects state of memory
- When too few frames are free
  - selects pages to evict using a replacement algorithm
- It can use same circular list (clock)
  - as regular page replacement algorithm but with diff ptr
- one for eviction
  - flush if it points to a dirty page
- another for replacement

## Implementation Issues

## Operating System Involvement with Paging

Four times when OS involved with paging

1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages

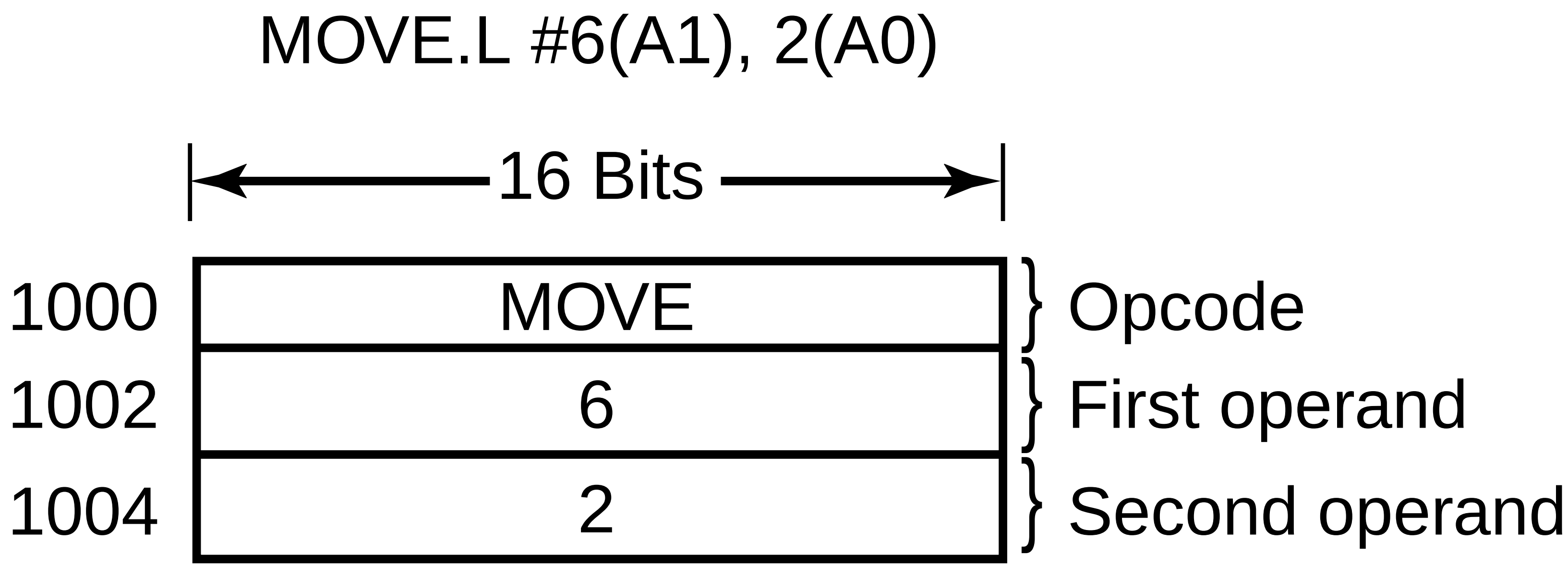
# Page Fault Handling (1)

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address,seeks page frame
5. If selected frame is dirty,write it to disk

# Page Fault Handling (2)

- OS brings new page in from disk
- Page tables updated
- Faulting instruction backed up to when it began
- Faulting process scheduled
- Registers restored
- Program continues

# Instruction Backup




An instruction causing a page fault

→ auto increment –R, R++

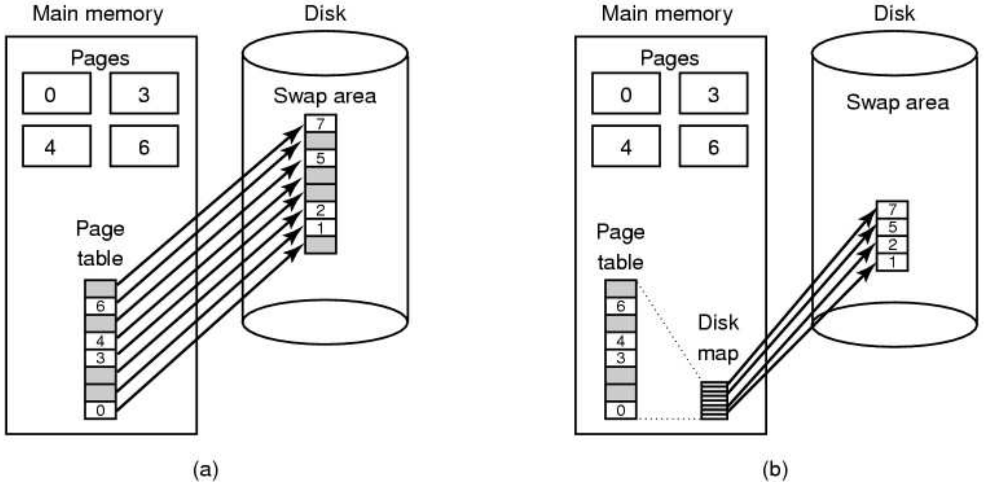
A index register(s) save the instruction that caused pagefault

(which register have already been inc/dec)

# Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Proc issues call for read from device into buffer
  - while waiting for I/O,another processes starts up
  - has a page fault
  - buffer for I/O DMA for the first proc may be chosen to be paged out
- Need to specify some pages locked
  - exempted from being target pages

# Backing Store

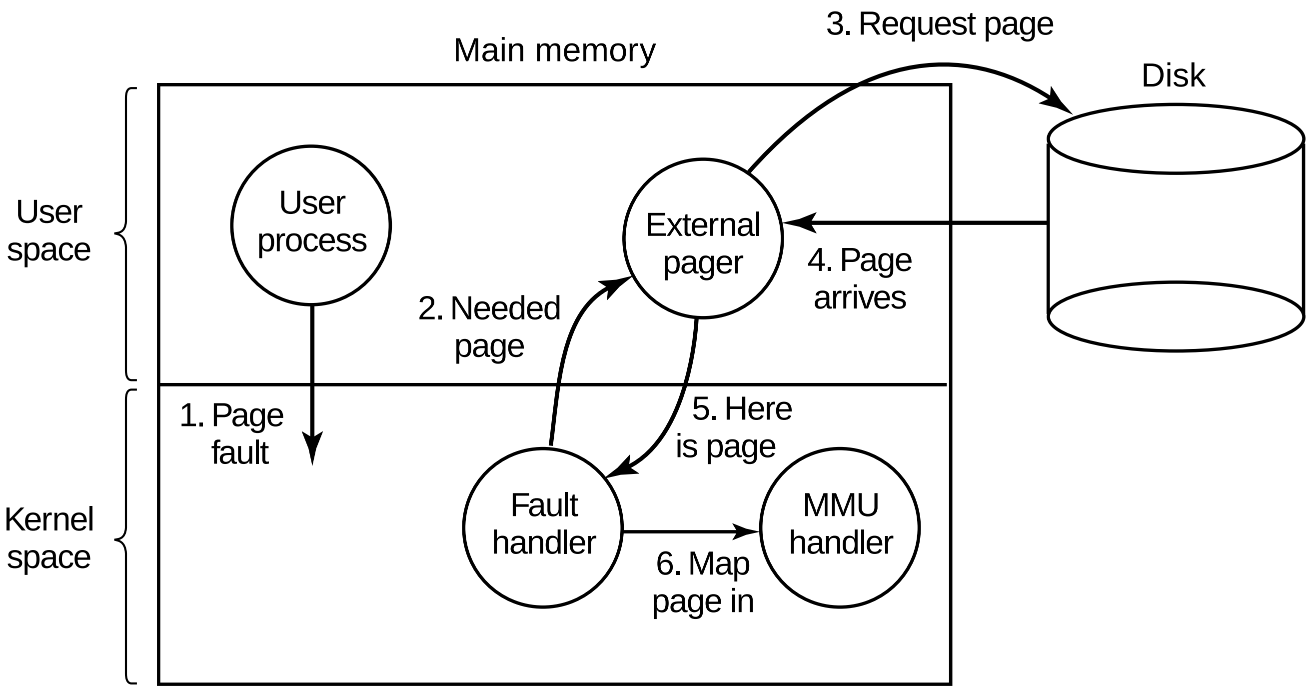



(a)Paging to static swap area (only vpn is needed)

(b)Backing up pages dynamically

need map (vpn → disk block)

# Separation of Policy and Mechanism

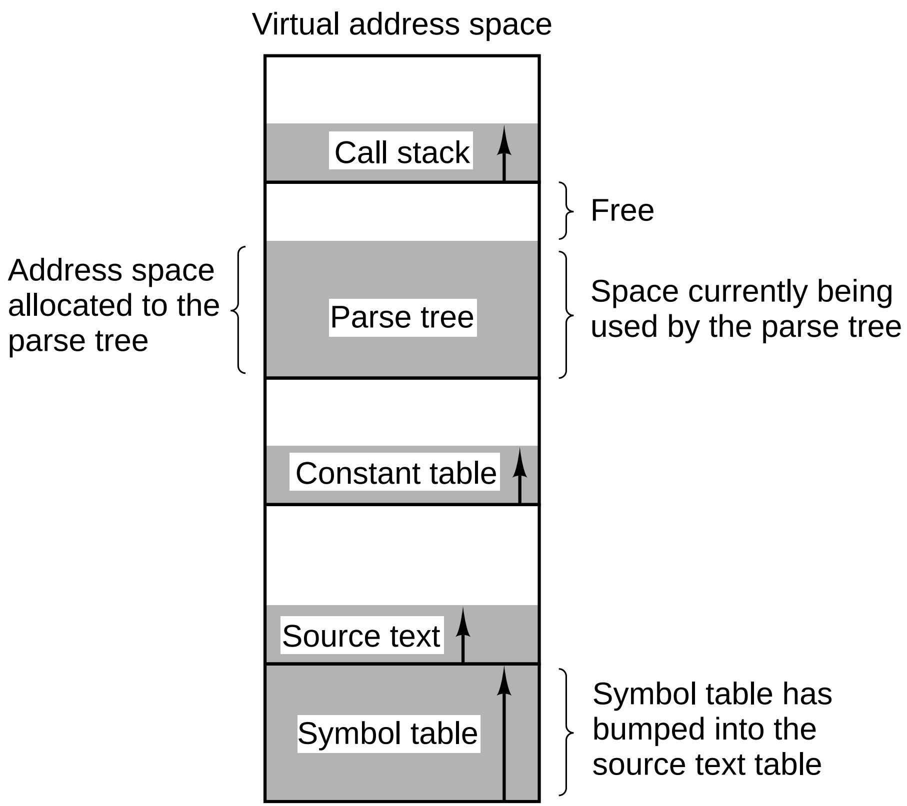



Page fault handling with an external pager

→ No access to modify and accessed bits

- Adv - more modular design
- Disadv - message overhead

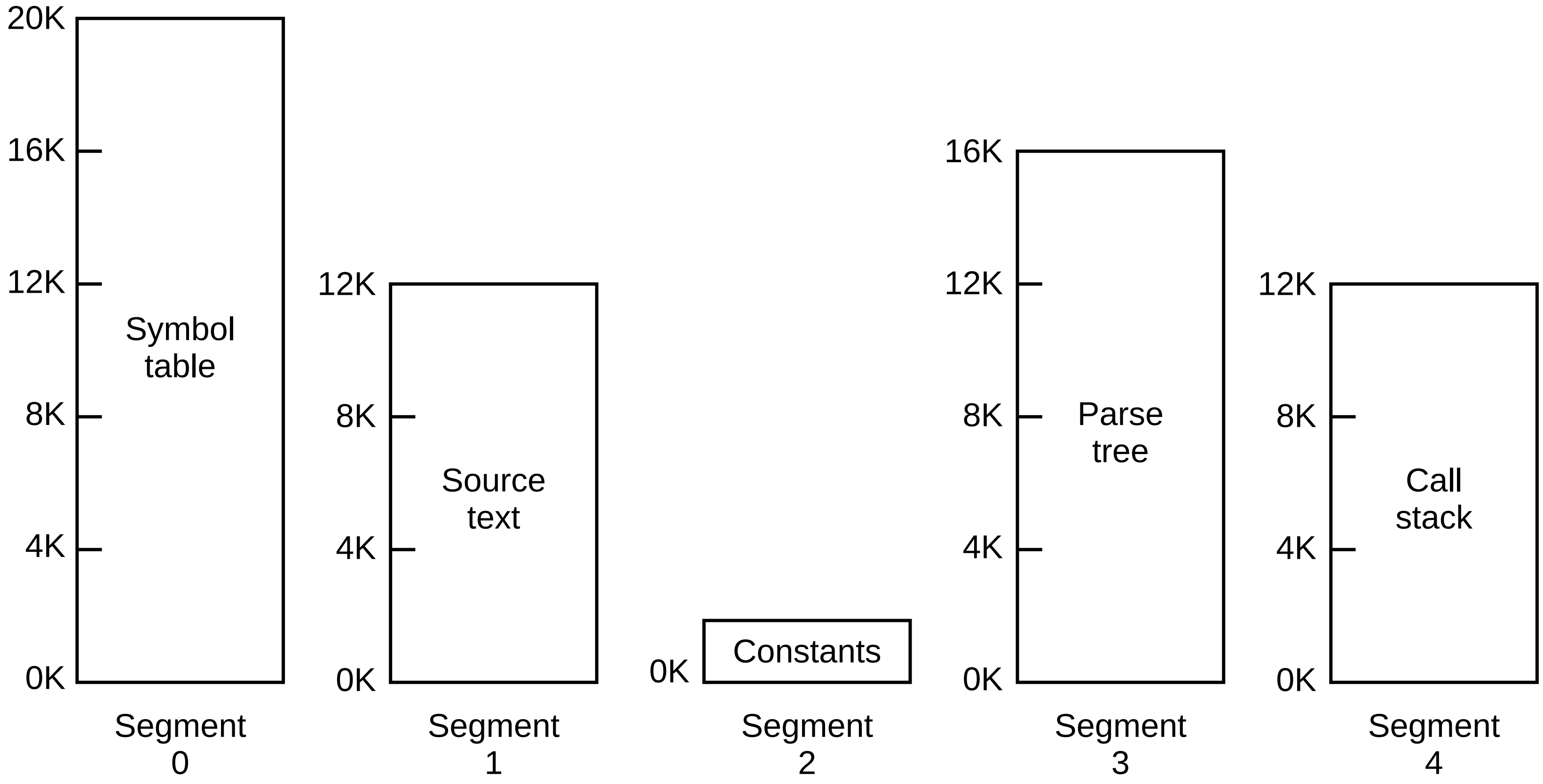
# Segmentation (1)




- One-dimensional address space with growing tables
- One table may bump into another

Example - Compiler program

# Segmentation (2)



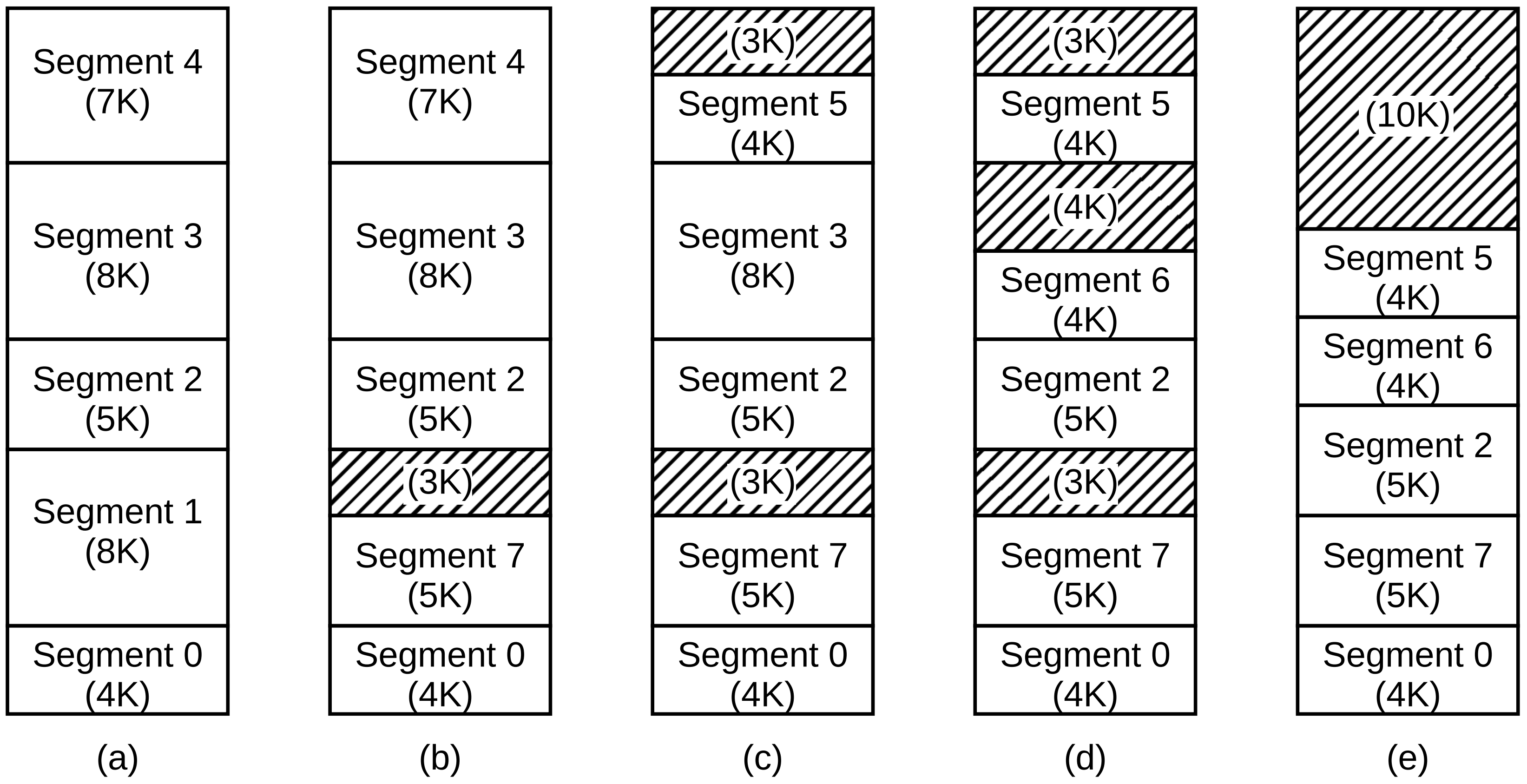

Allows each table to grow or shrink,independently

# 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

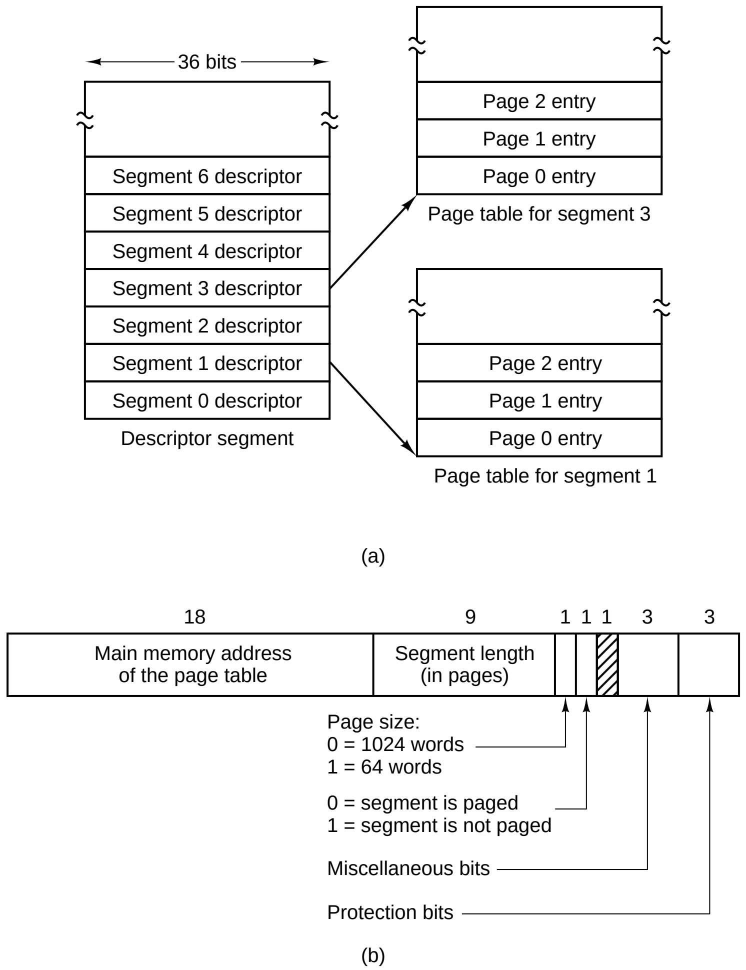
# Implementation of Pure Segmentation




(a)-(d)Development of checker boarding

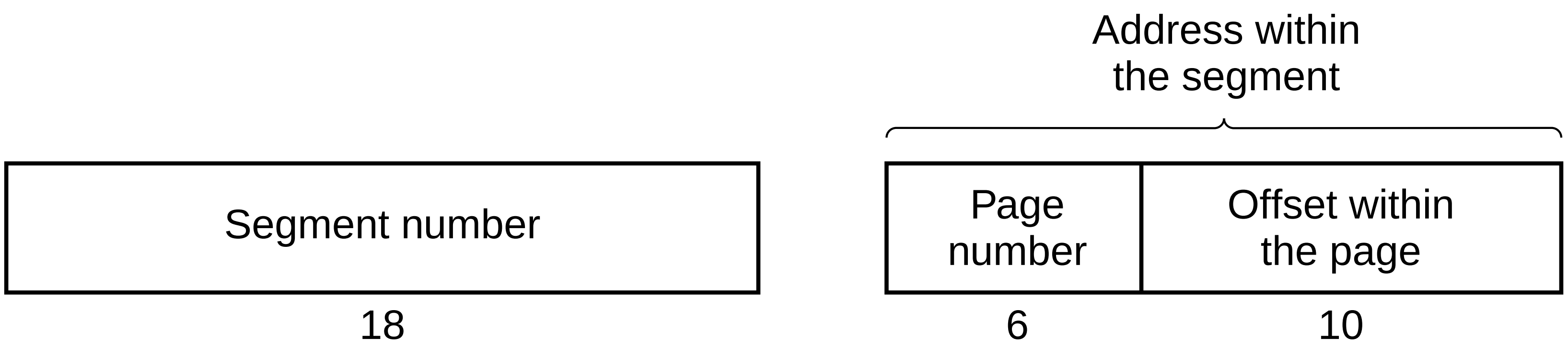
(e)Removal of the checker boarding by compaction

# Segmentation with Paging:MULTICS (1)



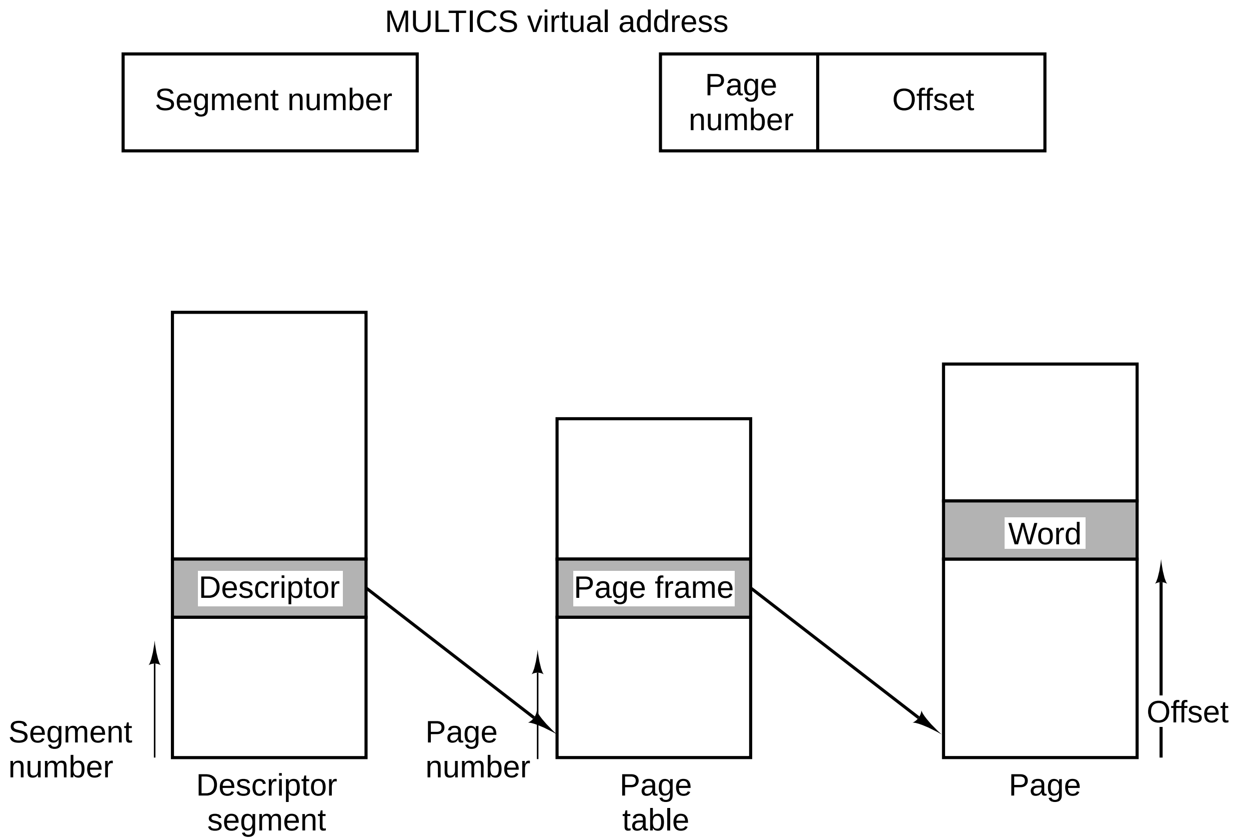

- Descriptor segment points to page tables
- Segment descriptor - numbers are field lengths

# Segmentation with Paging:MULTICS (2)




A 34-bit MULTICS virtual address

# Segmentation with Paging:MULTICS (3)



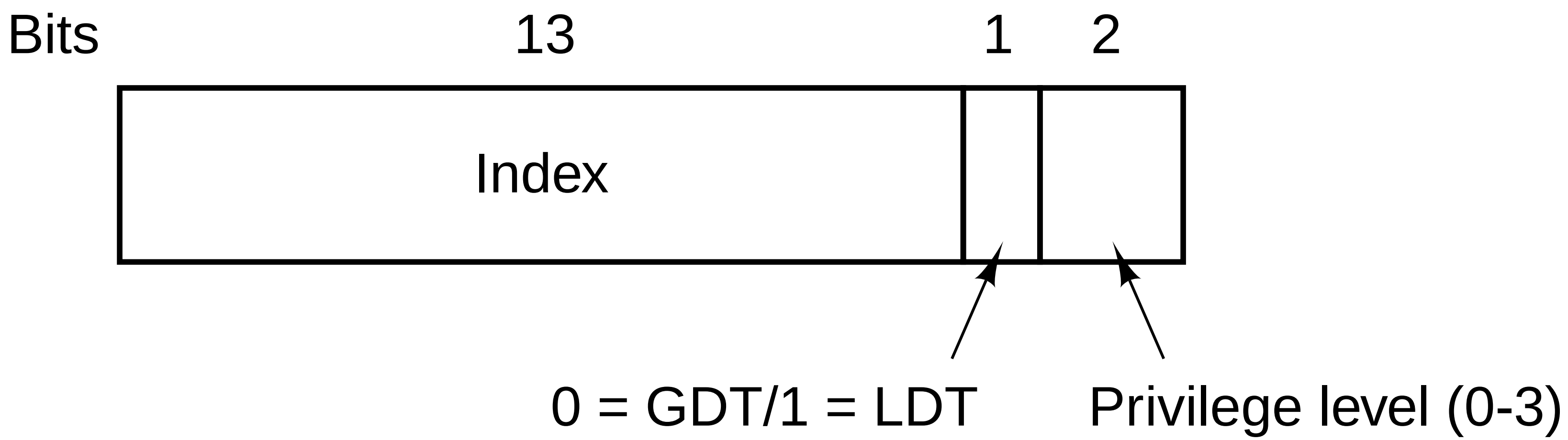

Conversion of a 2-part MULTI S address into a main memory address

# Segmentation with Paging:MULTICS (4)

| Segment number (Comparison field) | Virtual page (Comparison field) | 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 |
| | | | | | |

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

# Segmentation with Paging:Pentium (1)

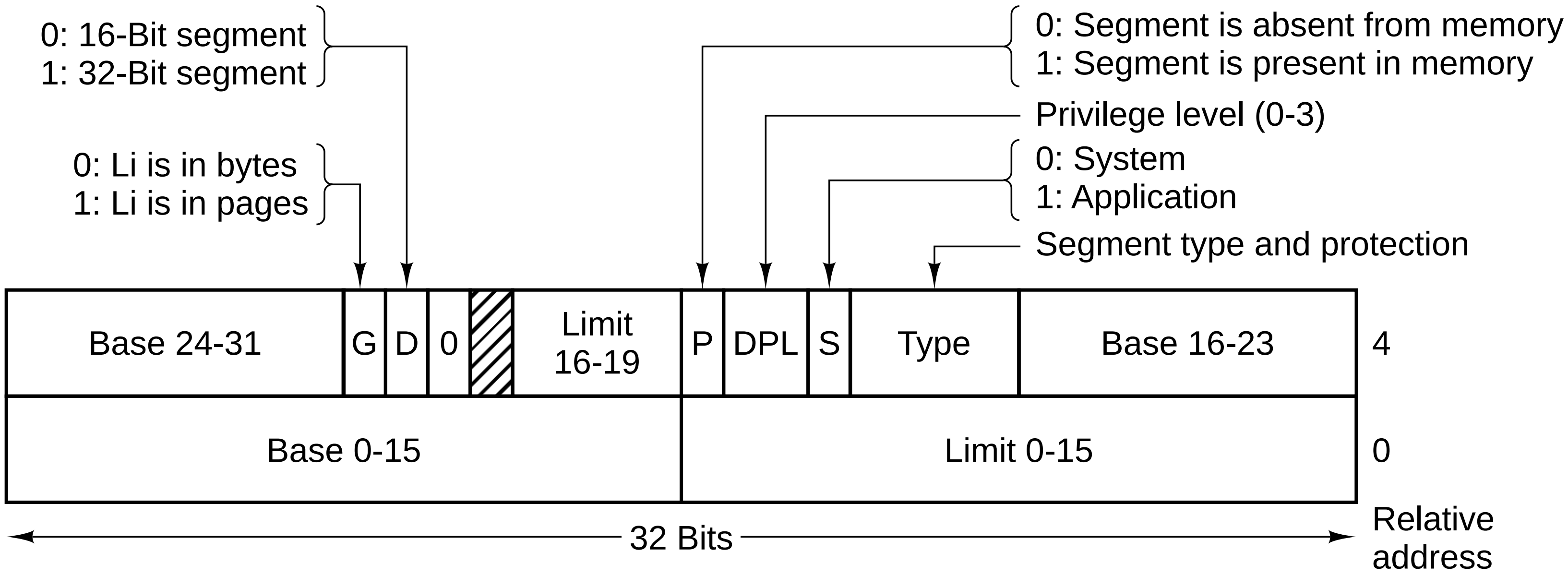



A Pentium selector

$\Downarrow$

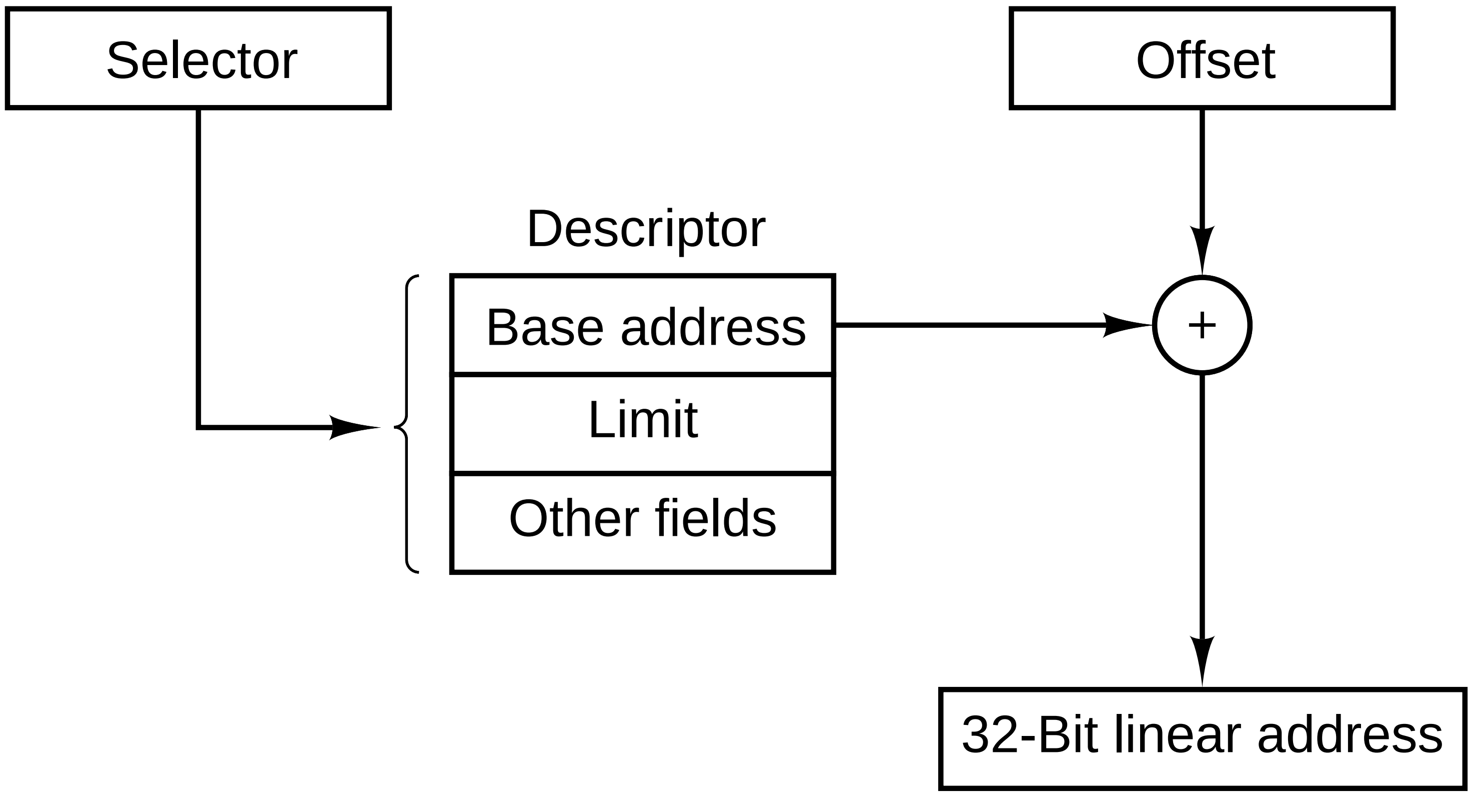
Loaded into one of segment registers (DS, CS, etc)

# Segmentation with Paging:Pentium (2)




- Pentium code segment descriptor
- Data segments differ slightly

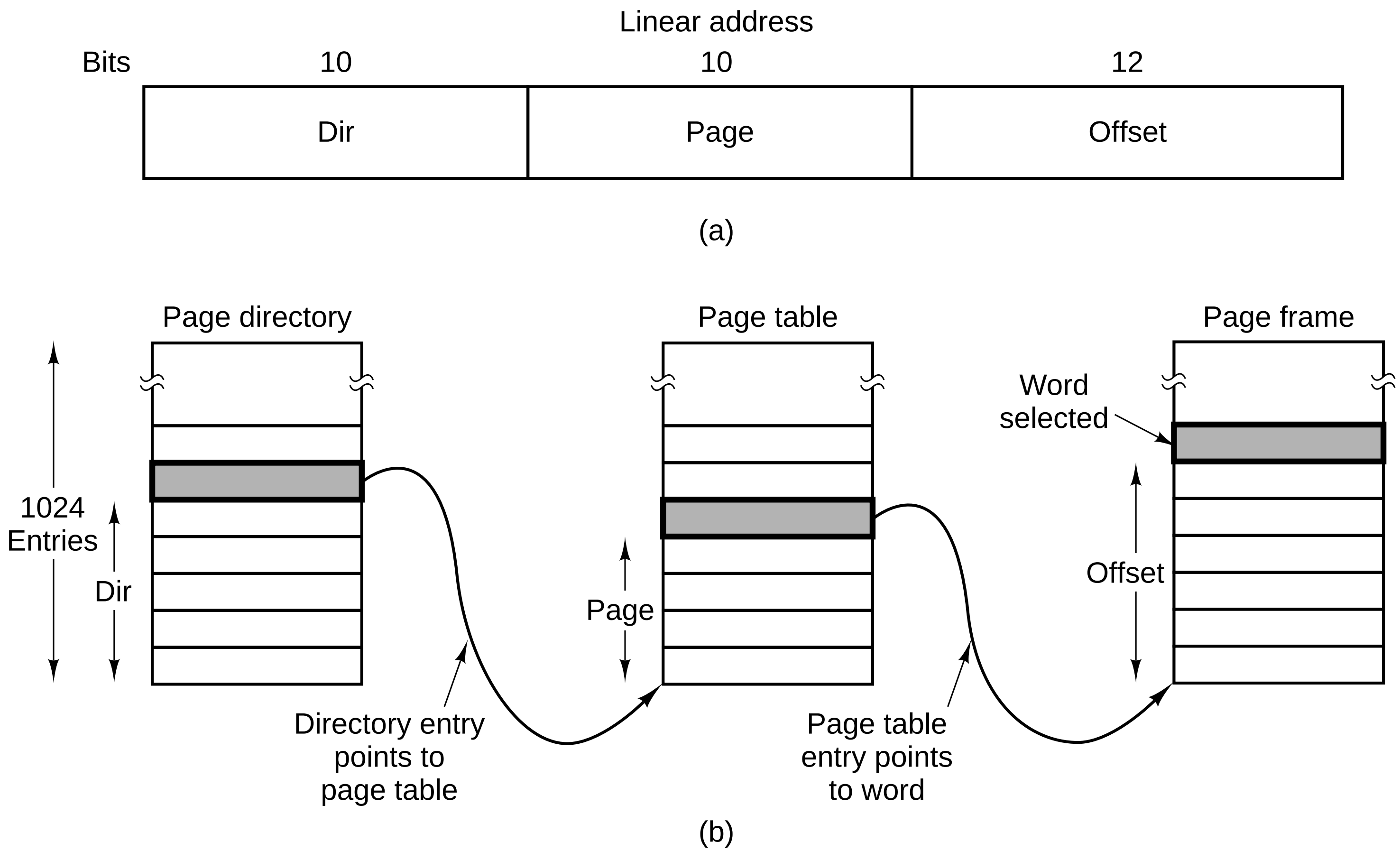
# Segmentation with Paging:Pentium (3)




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

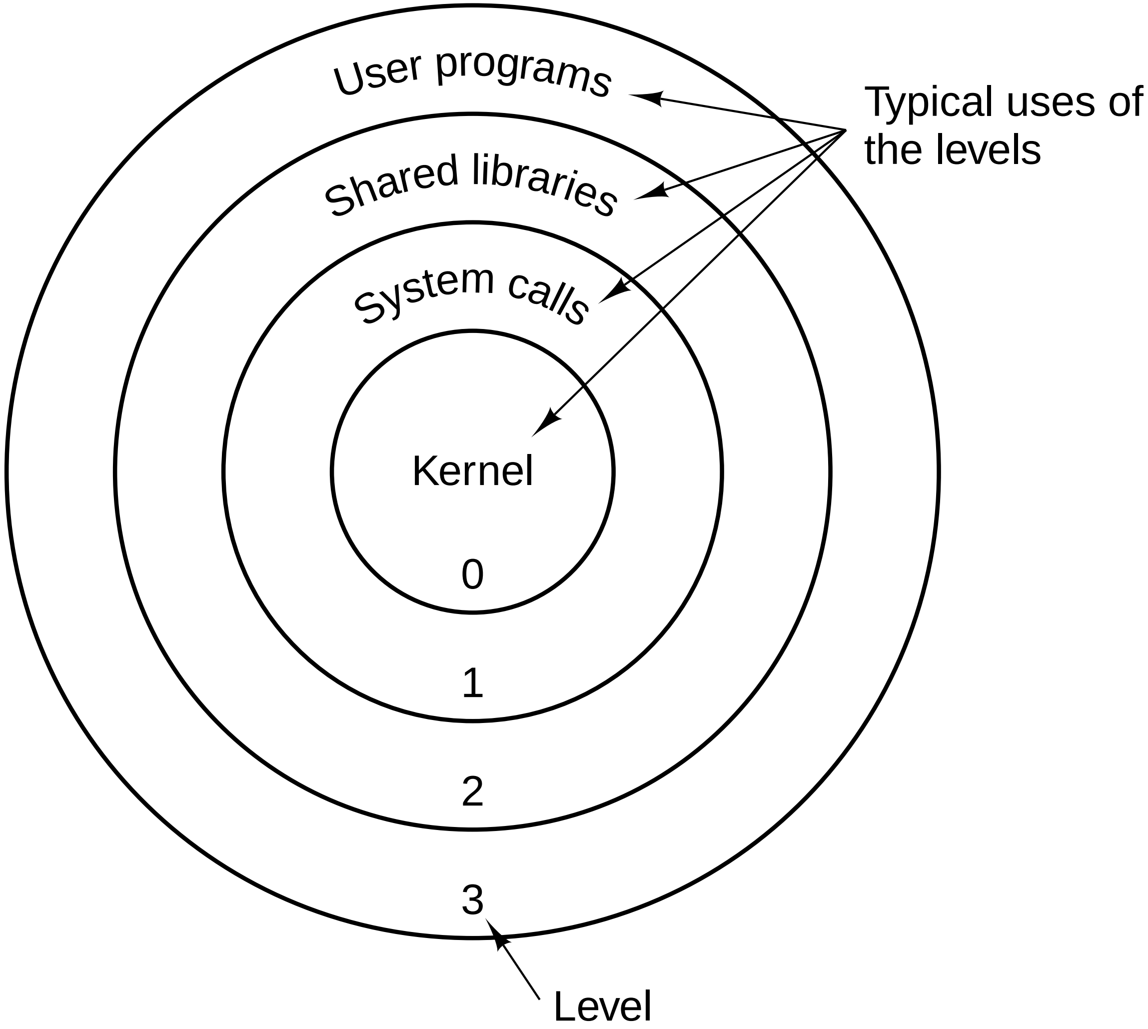
Limit $\geq$ Offset

# Segmentation with Paging:Pentium (4)




Mapping of a linear address onto a physical address

# Segmentation with Paging:Pentium (5)




Protection on the Pentium