Chapter 3

Memory Management

Part 3

Outline of Chapter 3

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation

} in this file

Local vs. Global Page Replacement

Assume several processes: A, B, C, ...
Some process gets a page fault.
(say, process A)
Choose a page to replace.

Local Page Replacement
Only choose one of A's pages

Global Page Replacement
Choose any page

Local vs. Global Page Replacement

Example: Process has a page fault...

Original	Age	Local	Global
A0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	A6	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	A6
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3

Original Local Global

Local vs. Global Page Replacement

Assume we have
- **5,000 frames in memory**
- **10 processes**

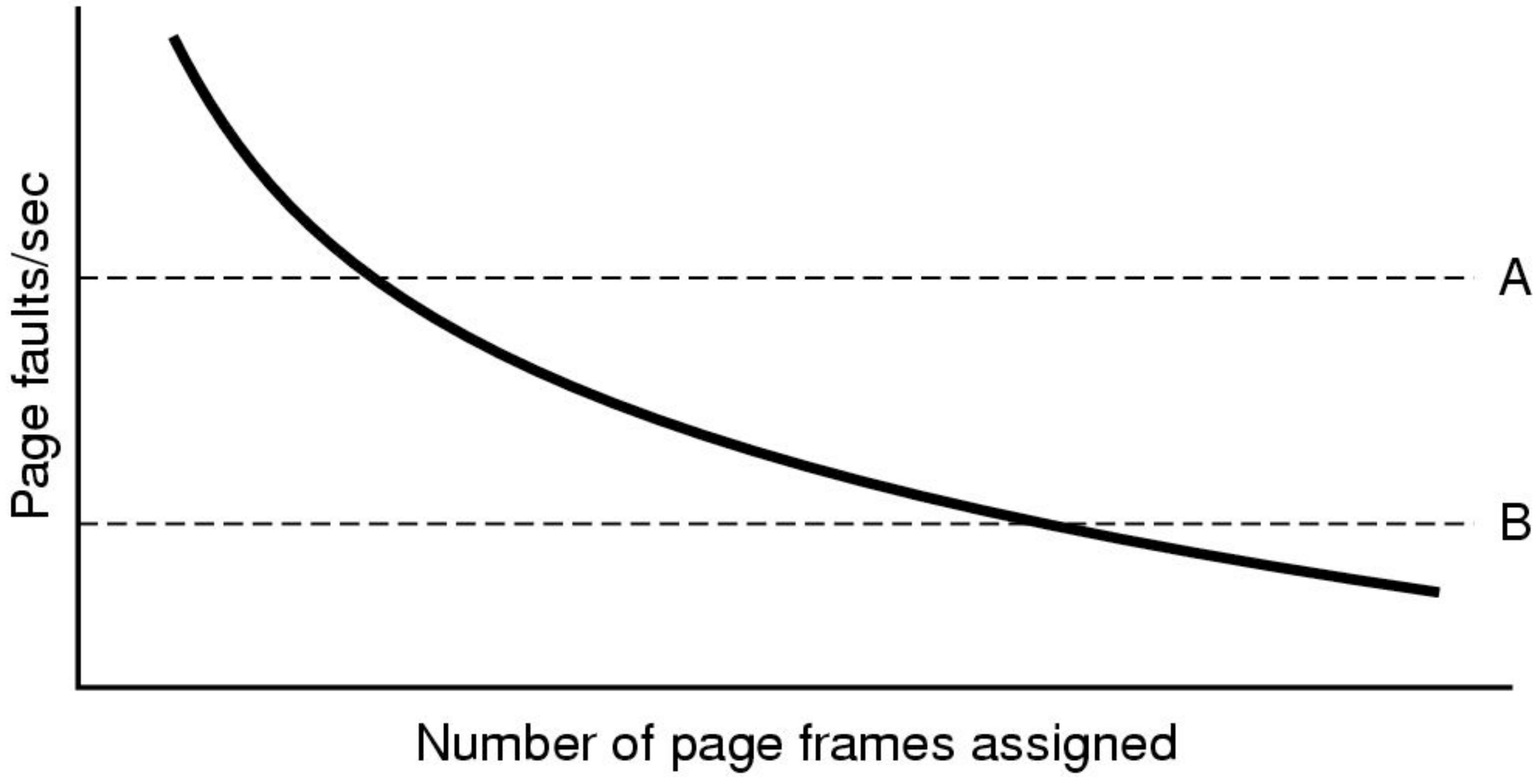
Idea: Give each process 500 frames

Fairness?
- **Small processes: do not need all those pages**
- **Large processes: may benefit from even more frames**

Idea:
- **Look at the size of each process**
- **Give them a pro-rated number of frames**
- **With a minimum of (say) 10 frames per process**

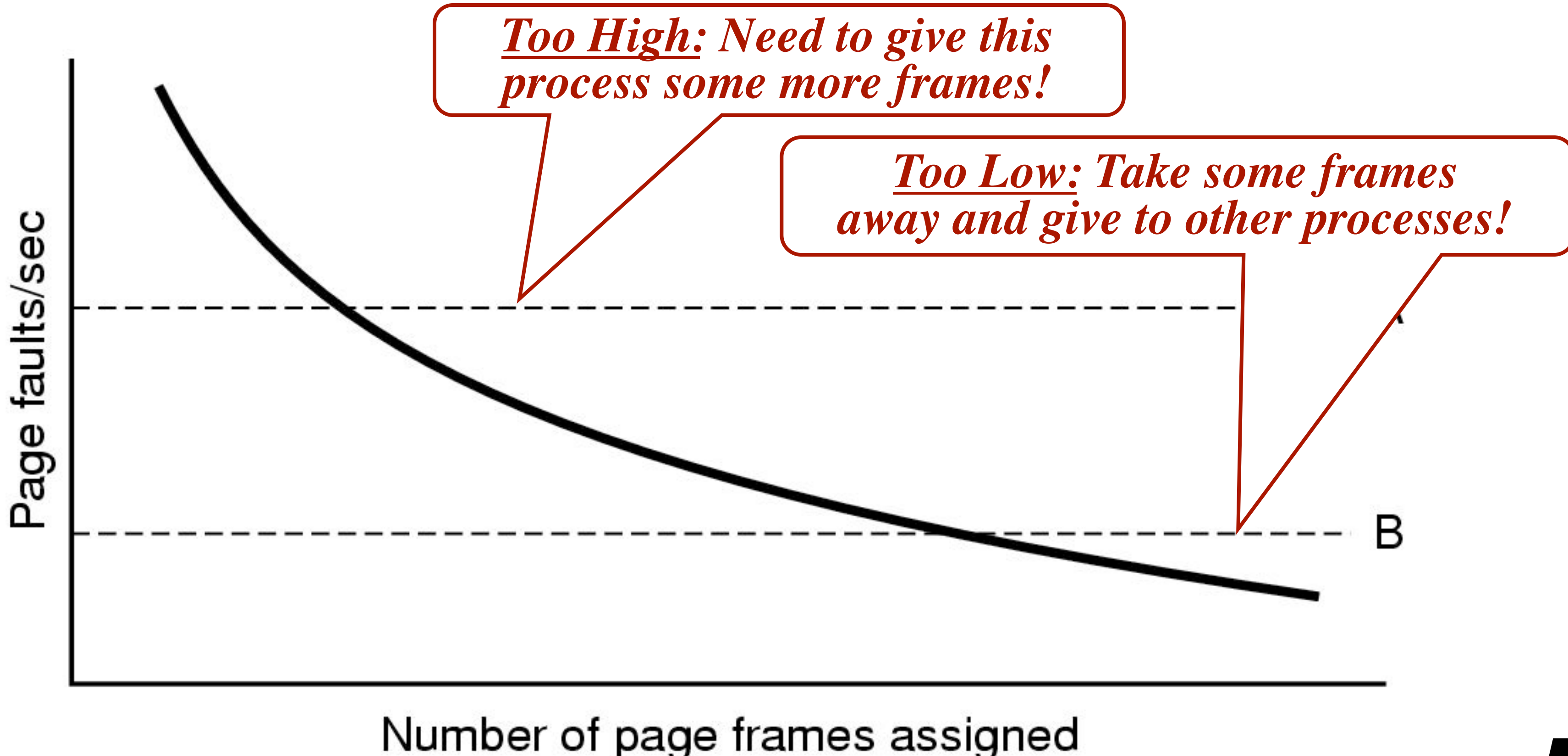
Page Fault Frequency

"If you give a process more pages,
its page fault frequency will decline."



Page Fault Frequency

"If you give a process more pages,
its page fault frequency will decline."



Page Fault Frequency

Measure the page fault frequency of each process.
Count the number of faults every second.

May want to consider the past few seconds as well.

Page Fault Frequency

Measure the page fault frequency of each process.
Count the number of faults every second.

May want to consider the past few seconds as well.

Aging:

- **Keep a running value.**
- **Every second**
  - **Count number of page faults**
  - **Divide running value by 2**
  - **Add in the count for this second**

Load Control

Assume:

- **The best page replacement algorithm**
- **Optimal global allocation of page frames**

Load Control

Assume:

- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!

Load Control

Assume:

- **The best page replacement algorithm**
- **Optimal global allocation of page frames**

Thrashing is still possible!

- **Too many page faults!**
- **No useful work is getting done!**
- **Demand for frames is too great!**

Load Control

Assume:

- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!

- Too many page faults!
- No useful work is getting done!
- Demand for frames is too great!

Solution:

- Get rid of some processes (temporarily).
- Swap them out.
- "Two-level scheduling"

Which Page Size is Best?

Smaller Page Sizes...

Advantages

- **Less internal fragmentation**
  On average: half of the last page is wasted
- **Working set takes less memory**
  Less unused program in memory

Disadvantages

- **Page tables are larger**
- **Disk-seek time dominates transfer time**
  (It takes same time to read large page as small page)

Which Page Size is Best?

Let

s = size of average process
e = bytes required for each page table entry
p = size of page, in bytes

s/p = Number of pages per process
es/p = Size of page table
p/2 = space wasted due to internal fragmentation

overhead = se/p + p/2

Which Page Size is Best?

Let

s = size of average process

e = bytes required for each page table entry

p = size of page, in bytes

overhead = se/p + p/2

Want to choose p to minimize overhead.

Take derivative w.r.t. p and set to zero

$-se/p^2 + 1/2 = 0$

Solving for p...

p = sqrt (2se)

Which Page Size is Best?

Let

s = size of average process = 1MB

e = bytes required for each page table entry = 8 bytes

p = size of page, in bytes

Solving for p...

p = sqrt (2se)

Example:

Which Page Size is Best?

Let

s = size of average process = 1MB

e = bytes required for each page table entry = 8 bytes

p = size of page, in bytes

Solving for p...

p = sqrt (2se)

Example:

p = sqrt (2 * 1MB * 8) = 4K

Which Page Size is Best?

Let

- **s = size of average process = 8MB**
- **e = bytes required for each page table entry = 4 bytes**
- **p = size of page, in bytes**

Solving for p...

- **p = sqrt (2se)**

Example:

- **p = sqrt (2 * 8MB * 4) = 8K**

Sharing Pages

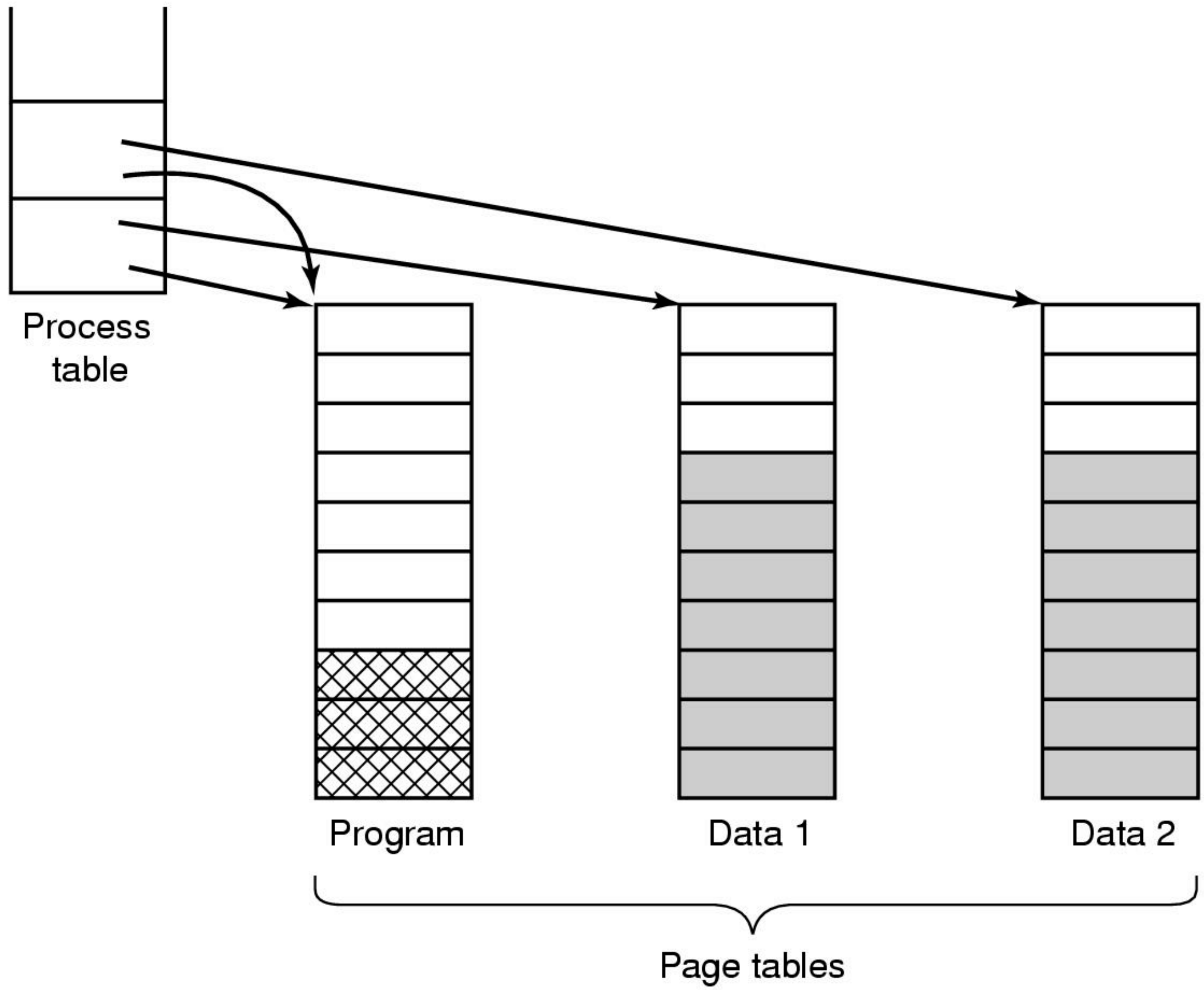
In a large multiprogramming system...

- **Many users**
- **Some running the same program at the same time**

Goal:

- **Share pages**
  - **Can only share read-only pages (text segment)**

Sharing Pages



Sharing Pages

In Unix:

A "Fork" syscall

Copy the parent's virtual address space

... and immediately do an "Exec" syscall

Desired Semantics:

"Data and text segments are copied"

Sharing Pages

In Unix:

A "Fork" syscall

Copy the parent's virtual address space

... and immediately do an "Exec" syscall

Desired Semantics:

"Data and text segments are copied"

Idea: Copy-On-Write

- Share all pages
- Mark all pages "read-only"
- Page Fault:
  - Is this a "data" page?
  - Copy the page
  - Mark both copies "writable"
  - Resume execution

Paging Daemon

Paging works best if there are plenty of free frames.

- If all pages are full of dirty pages...
  - Must perform 2 disk operations for each page fault

Paging Daemon

Paging works best if there are plenty of free frames.
If all pages are full of dirty pages...
Must perform 2 disk operations for each page fault

Page Daemon

- **A kernel process**
- **Wakes up periodically**
- **Counts the number of free pages**
- **If too few, run the page replacement algorithm...**
  - **Select a page & write it to disk**
  - **Mark the page as clean**

If this page is needed later... then it is still there.
If an empty frame is needed later... this page is evicted.

New System Calls for Page Management

Goal:

Allow some processes more control over paging!

System calls added to the kernel

Example: A process can request a page before it is needed

Processes can share pages

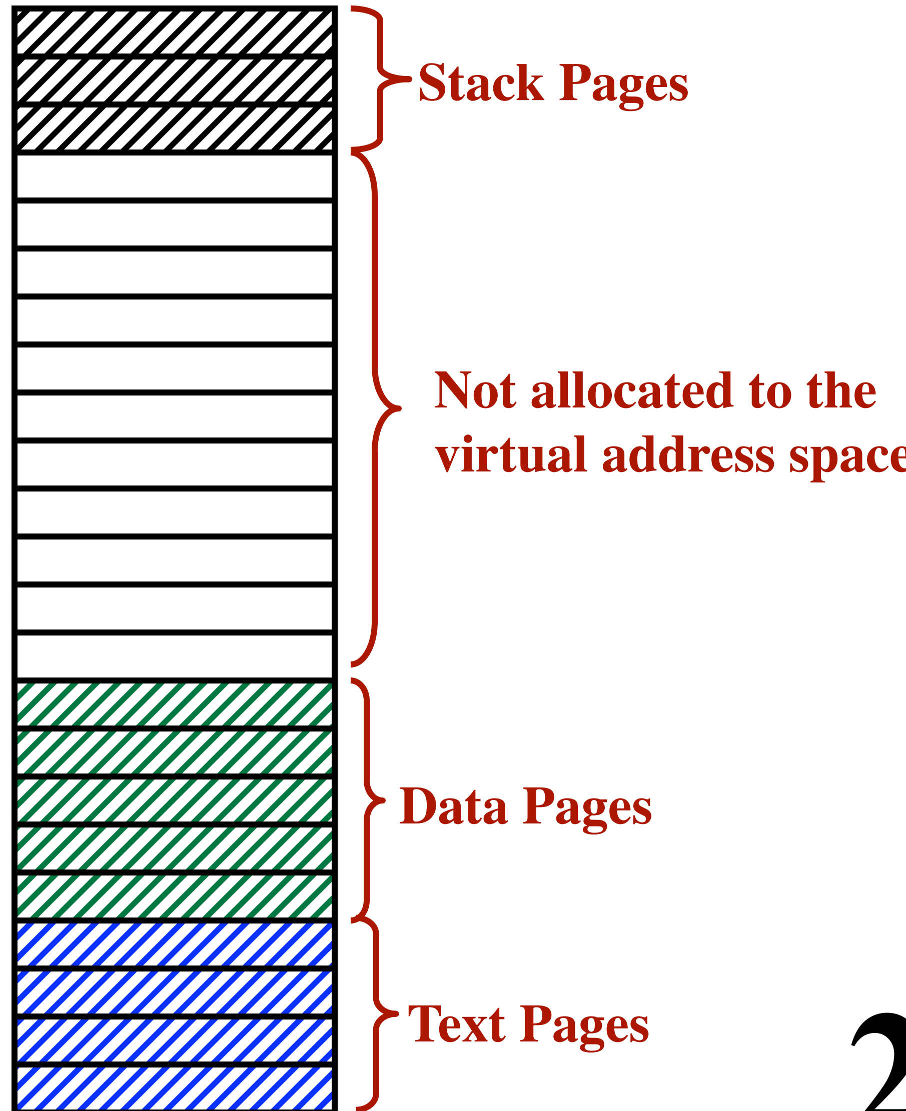
Allows fast movement of data between processes

Processes can grow

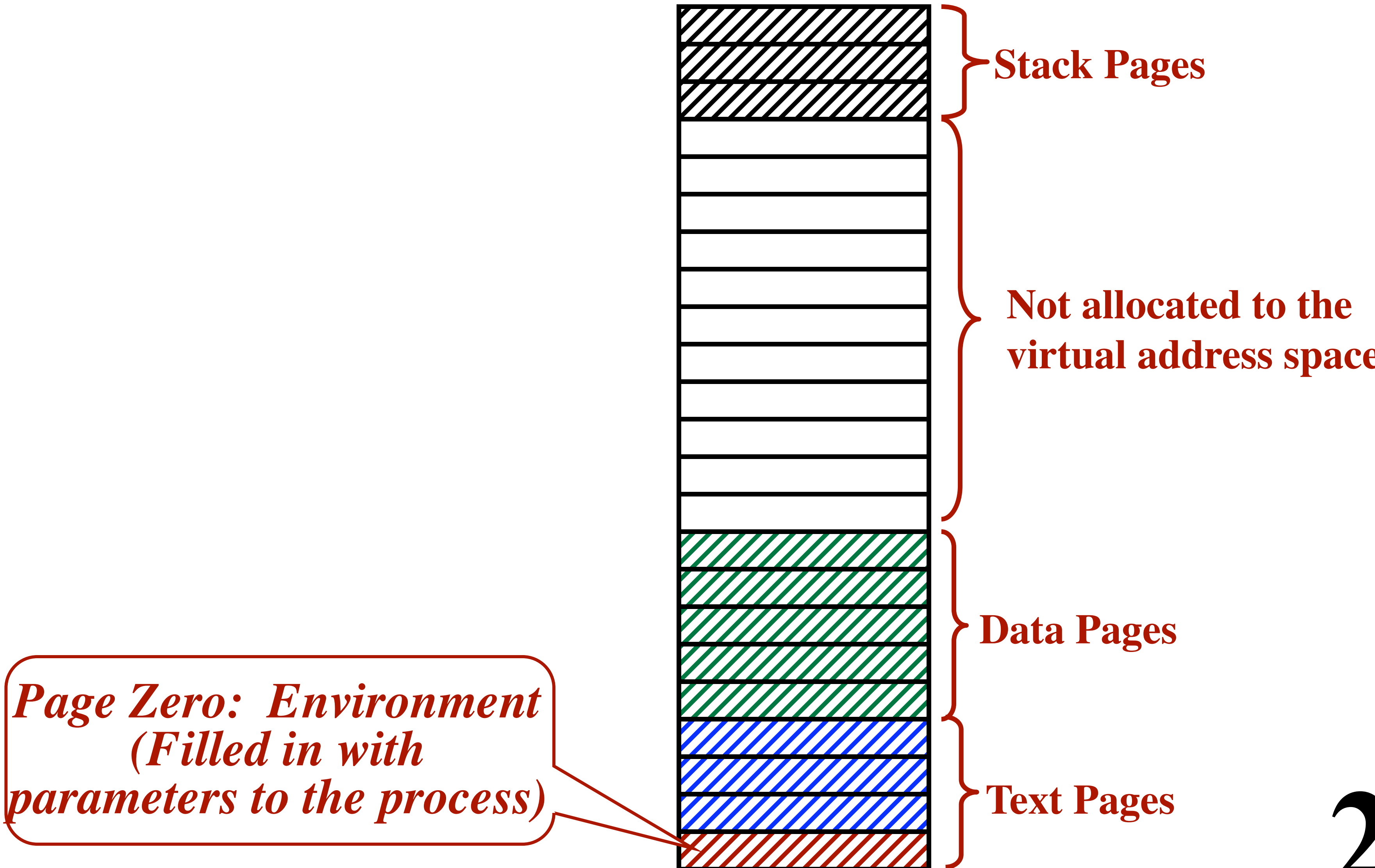
Heap manager

- **User-level code**
- **May request more memory, as needed**

Unix Processes

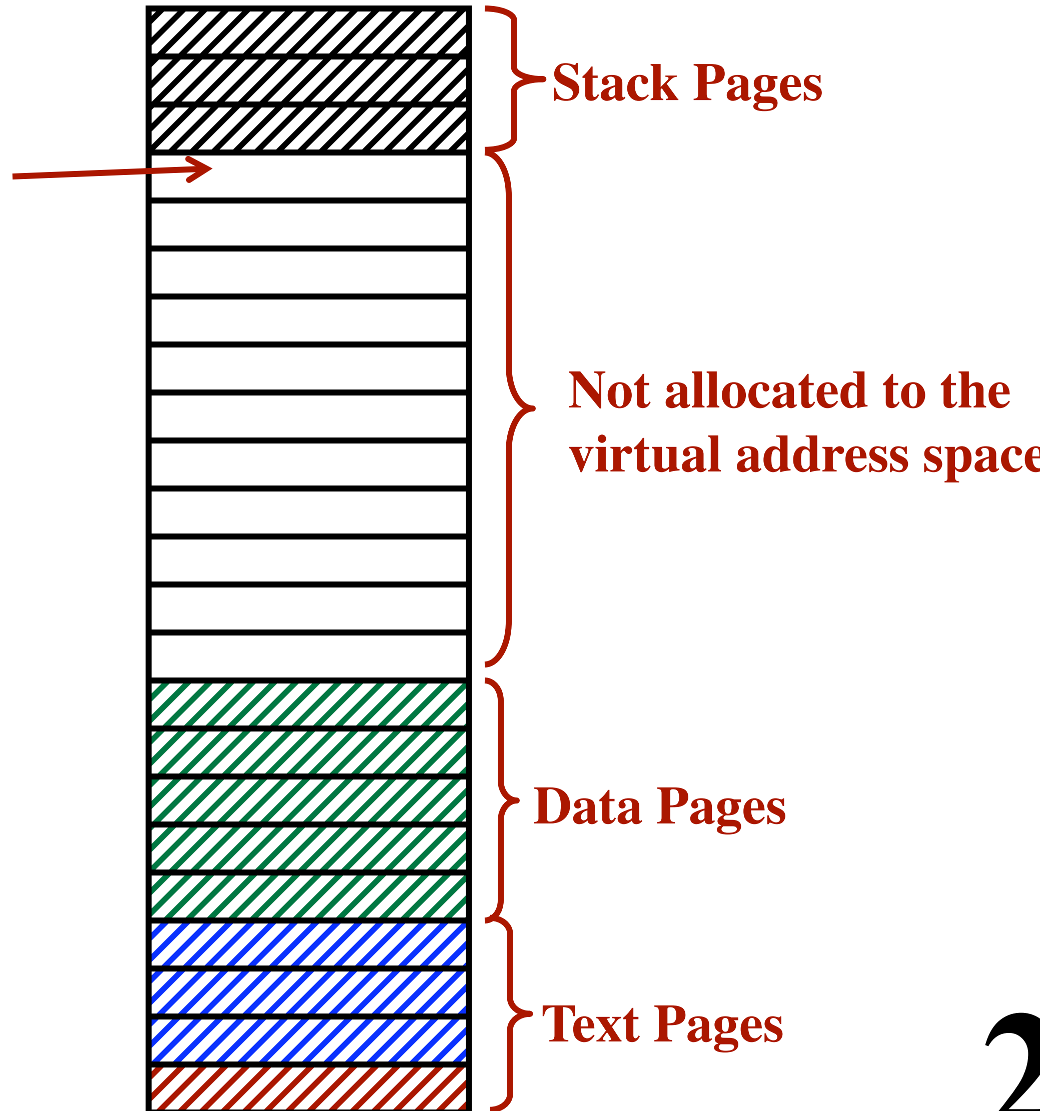


Unix Processes



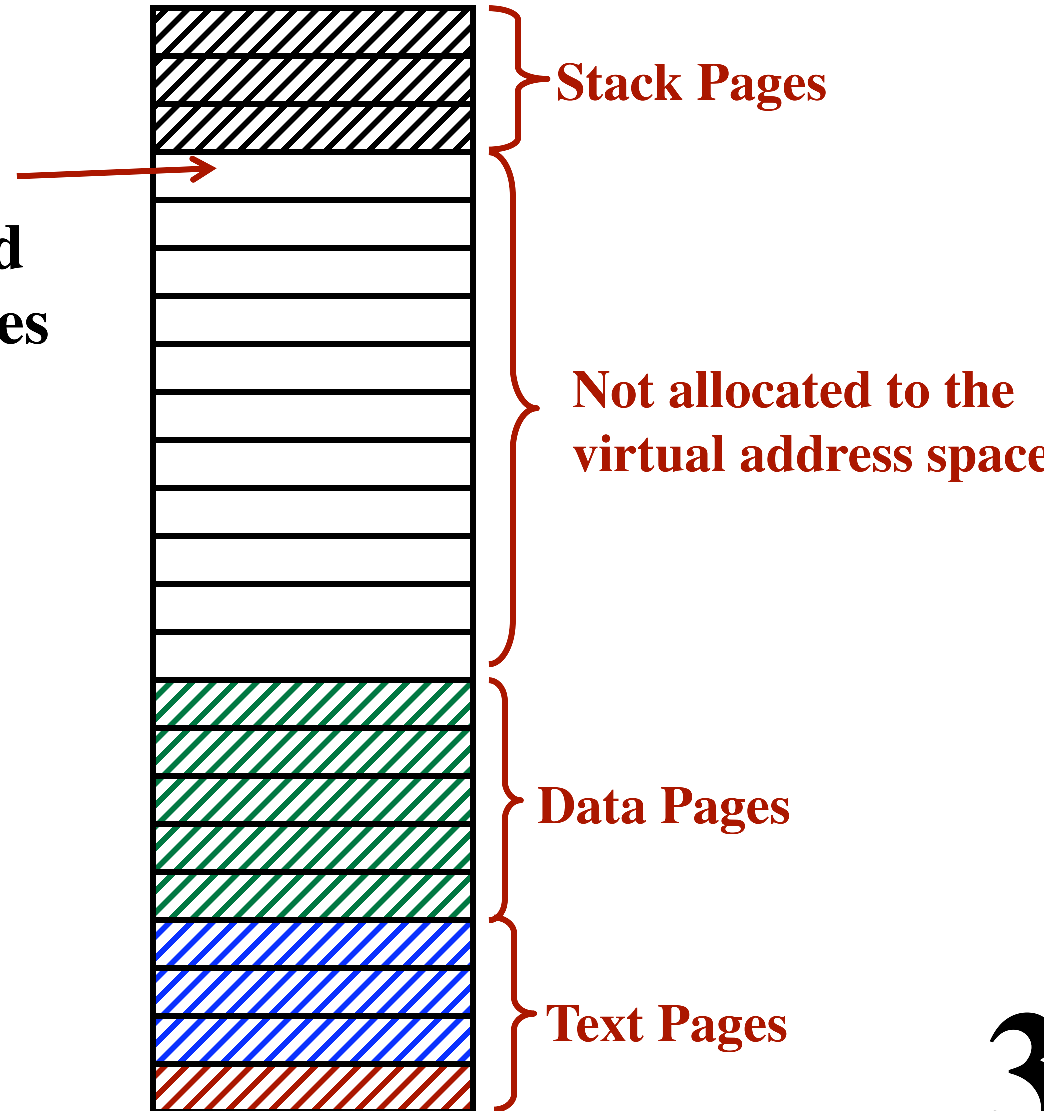
Unix Processes

The stack grows;
Page fault occurs here



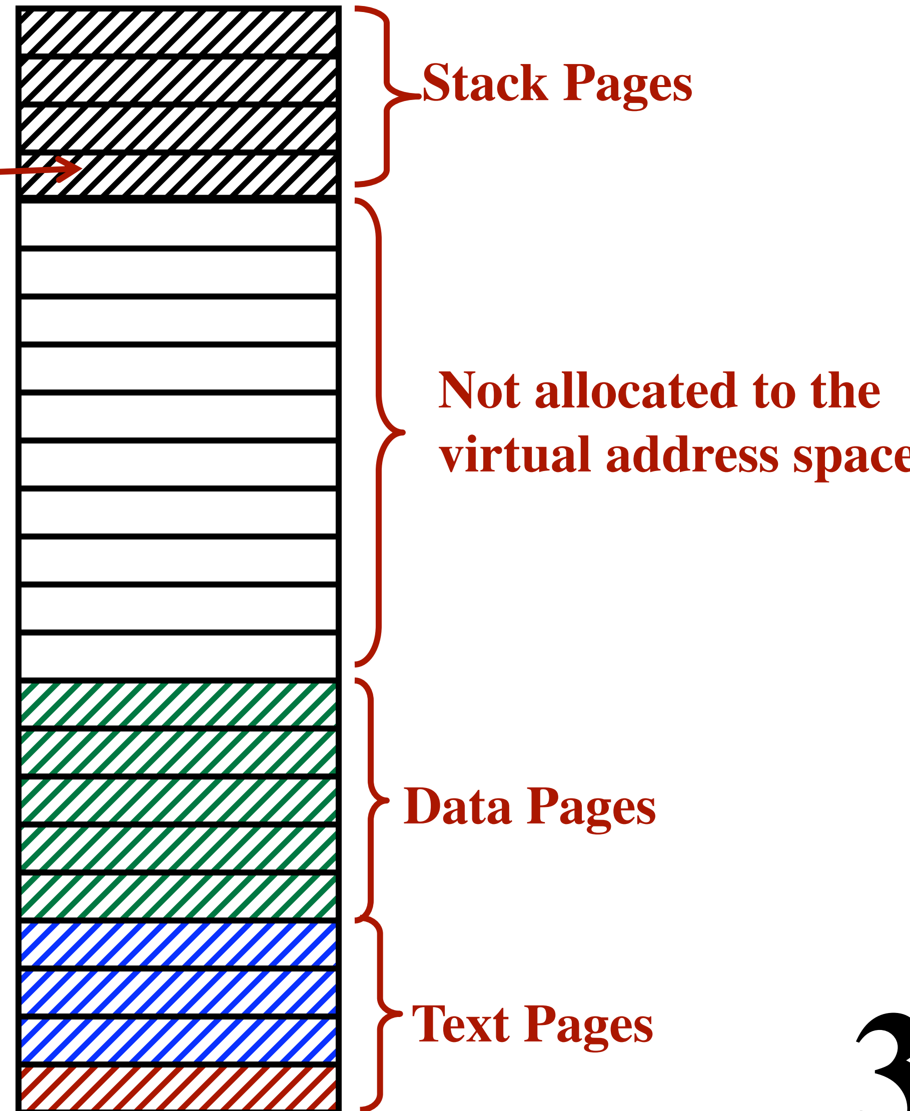
Unix Processes

The stack grows;
Page fault occurs here
A new page is allocated
and process continues



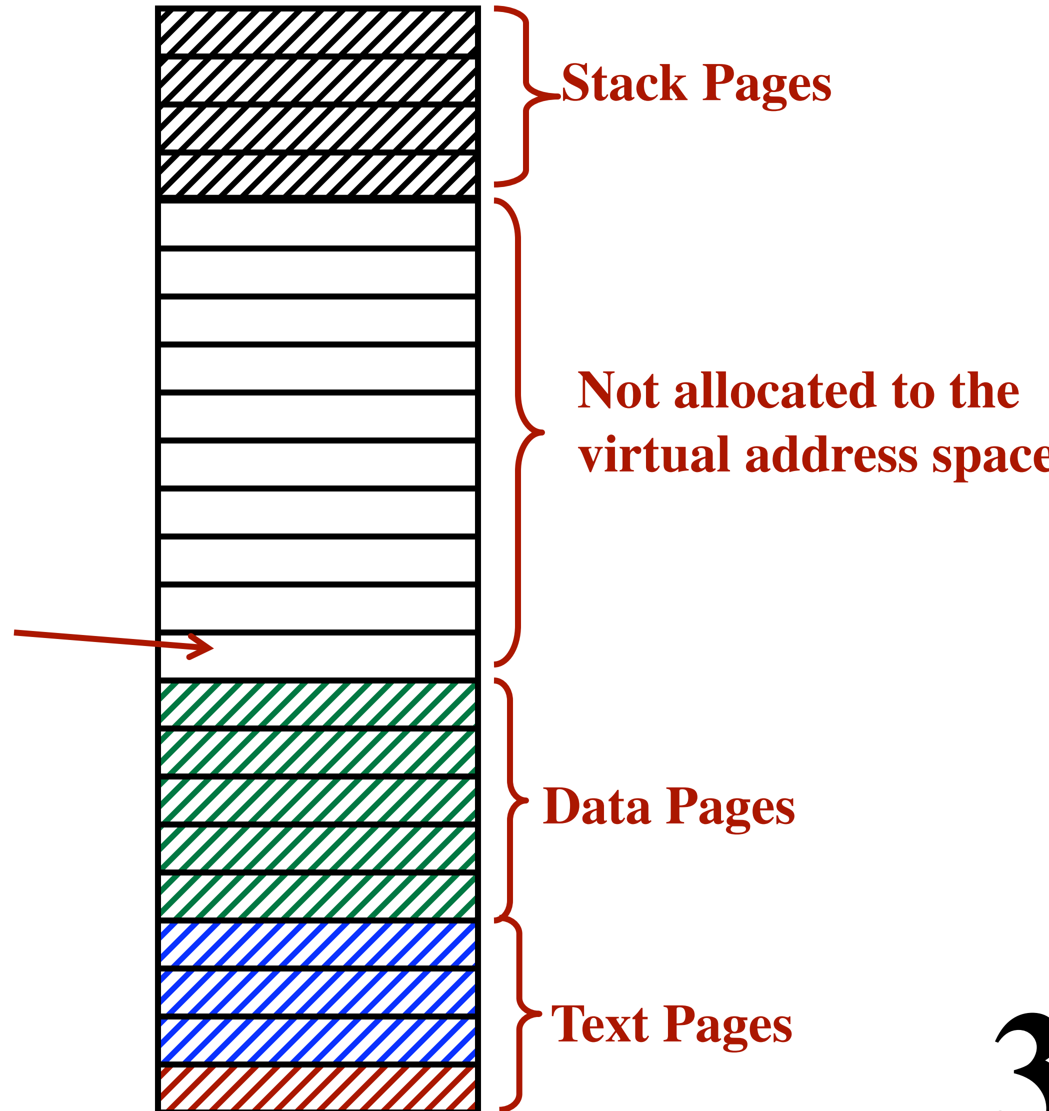
Unix Processes

The stack grows;
Page fault occurs here
A new page is allocated
and process continues



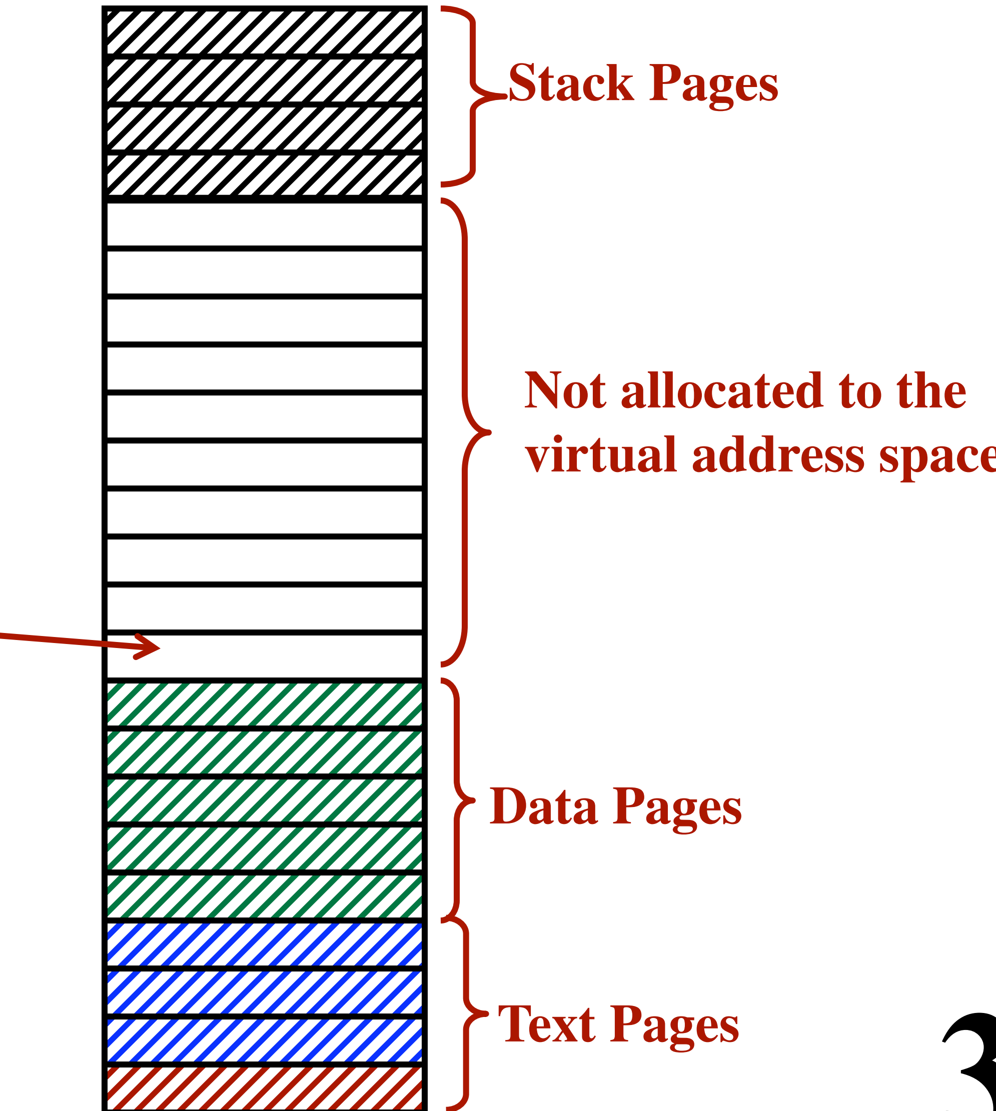
Unix Processes

The heap grows;
Page fault occurs here



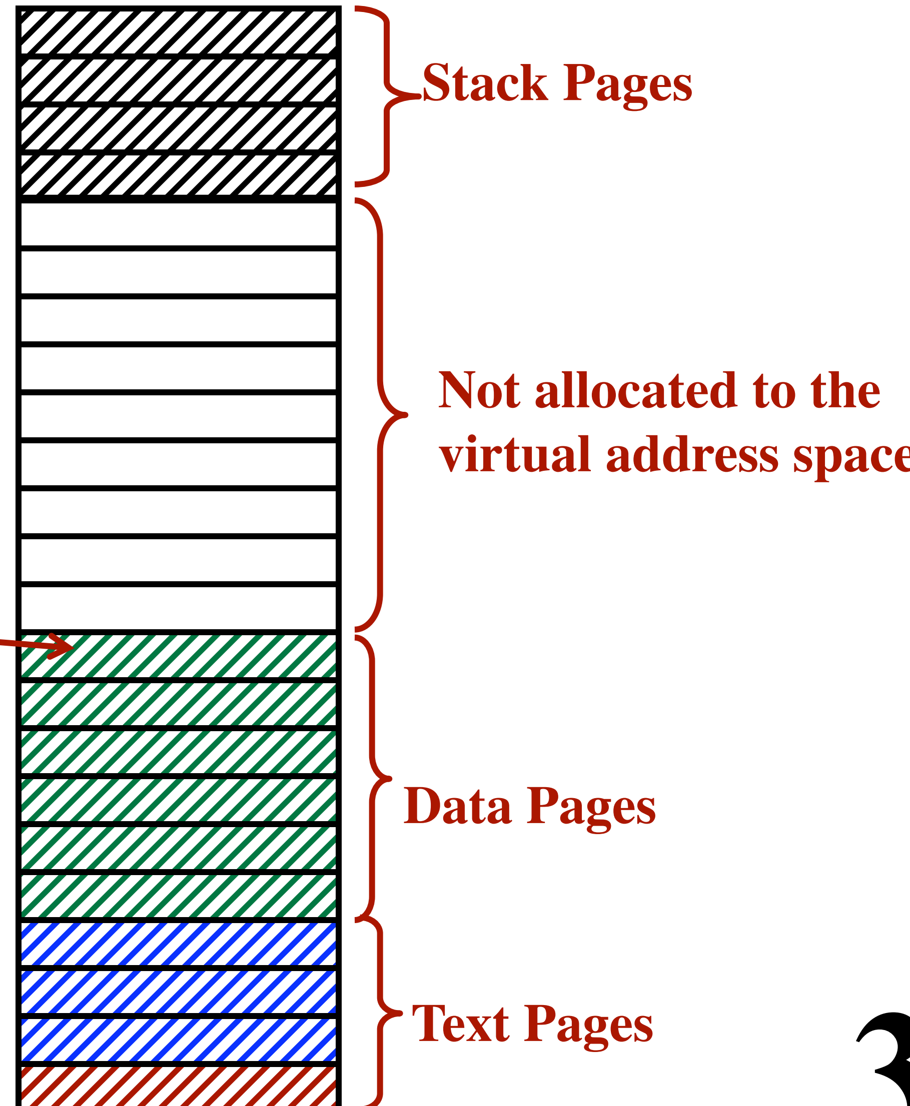
Unix Processes

The heap grows;
Page fault occurs here
A new page is allocated
and process continues



Unix Processes

The heap grows;
Page fault occurs here
A new page is allocated
and process continues



Virtual Memory Implementation

When is the kernel involved?

Virtual Memory Implementation

When is the kernel involved?

- ***Process Creation***
- ***Process is scheduled to run***
- ***Page Fault Occurs***
- ***Process Termination***

Virtual Memory Implementation

When is the kernel involved?

- ***Process Creation***

  Determine the process size

  Create page table

- ***Process is scheduled to run***

- ***Page Fault Occurs***

- ***Process Termination***

Virtual Memory Implementation

When is the kernel involved?

- ***Process Creation***

  Determine the process size

  Create page table

- ***Process is scheduled to run***

  MMU is initialized to point to new page table

  TLB is flushed

- ***Page Fault Occurs***

- ***Process Termination***

Virtual Memory Implementation

When is the kernel involved?

- ***Process Creation***

  Determine the process size

  Create page table
- ***Process is scheduled to run***

  MMU is initialized to point to new page table

  TLB is flushed
- ***Page Fault Occurs***

  Determine the virtual address causing the problem

  Swap the evicted page out & read in the desired page
- ***Process Termination***

Virtual Memory Implementation

When is the kernel involved?

- ***Process Creation***

  Determine the process size

  Create page table

- ***Process is scheduled to run***

  MMU is initialized to point to new page table

  TLB is flushed

- ***Page Fault Occurs***

  Determine the virtual address causing the problem

  Swap the evicted page out & read in the desired page

- ***Process Termination***

  Release / free all frames

  Release / free the page table

Handling a Page Fault

Hardware traps to kernel
- **PC and SR are saved on stack**

Save rest of registers

Determine the virtual address causing the problem

Check validity of the address; determine which page needed
- **May need to just kill the process**

Find the frame to use (page replacement algorithm)

Is the target frame dirty? Write it out.
- **(& schedule other processes)**

Read in the desired frame from swapping file.

Update the page tables

(continued)

Handling a Page Fault

Back up the current instruction

The "faulting instruction"

Schedule the faulting process to run again

Return to scheduler

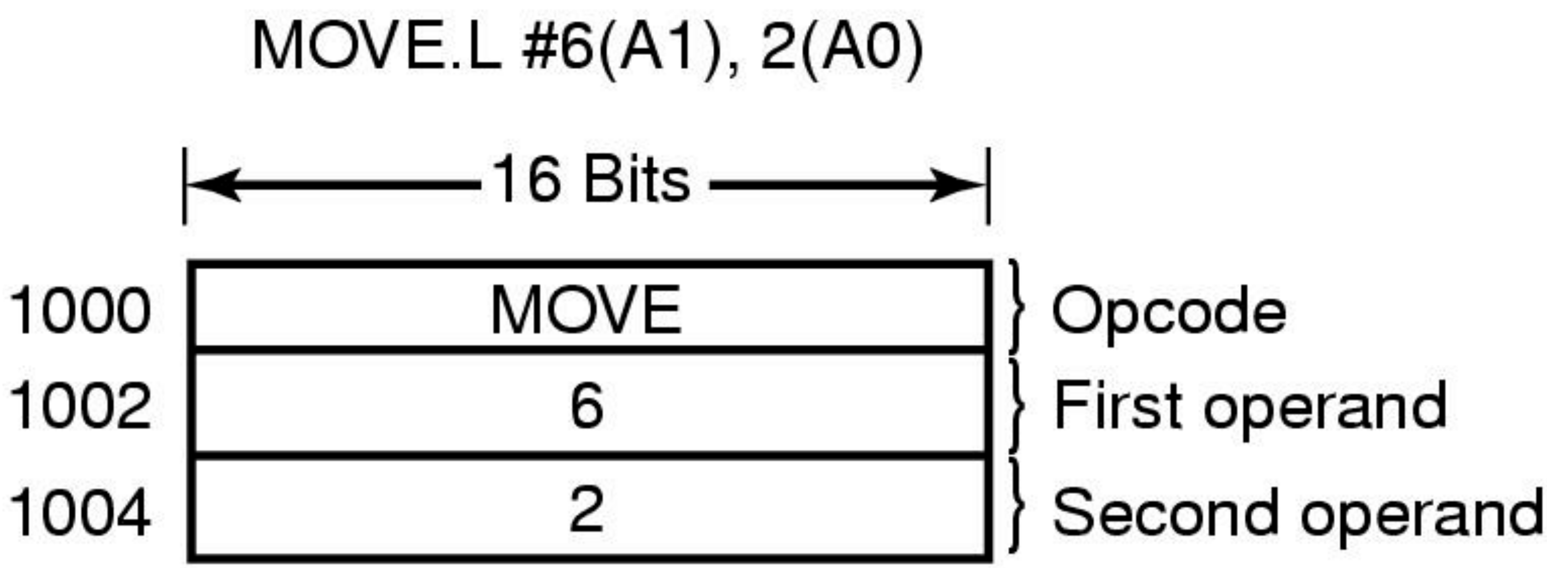
...

Reload registers

Resume execution

Backing the PC Up to Restart an Instruction

Consider a multi-word instruction.
The instruction makes several memory accesses.
One of them faults.
The value of the PC depends on when the fault occurred.
How can you know what instruction was executing???



Solutions

- Lot’s of clever code in the kernel

- Hardware support
  Dump internal CPU state into special registers
  Make “hidden” registers accessible to kernel

- Better ISA design

Locking Pages in Memory

"Pinning" the Pages

Virtual Memory and I/O interact

Example:

One process does a Sys_Read

(This process suspends during I/O)

Another process runs

It has a page fault

Some pages is selected for eviction

The frame selected contains the page involved above!!!

Solution:

Each frame has a flag: "Do not evict me".

Must always remember to un-pin the page!

Swap Area on Disk

Approach #1:

A process starts up

Assume it has N pages in its virtual address space

A region of the swap area is set aside for the pages

There are N pages in the swap region

The pages are kept in order

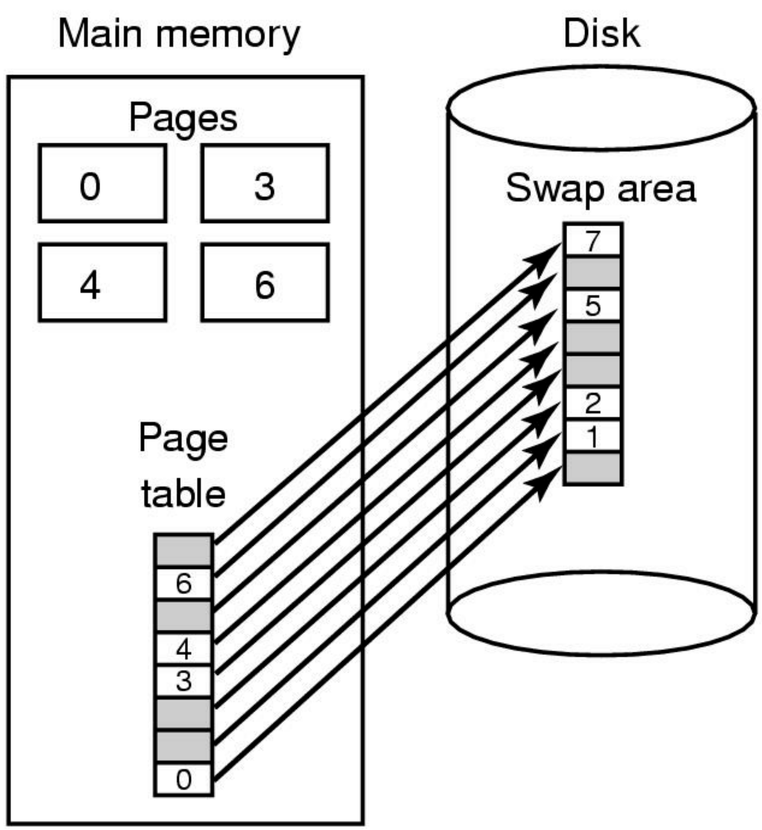
For each process, we need to know:

- Disk address of page 0
- Number of pages in address space

Each page is either...

- In a memory frame
- Stored on disk

Approach #1



Problem

What if the virtual address space grows during execution?

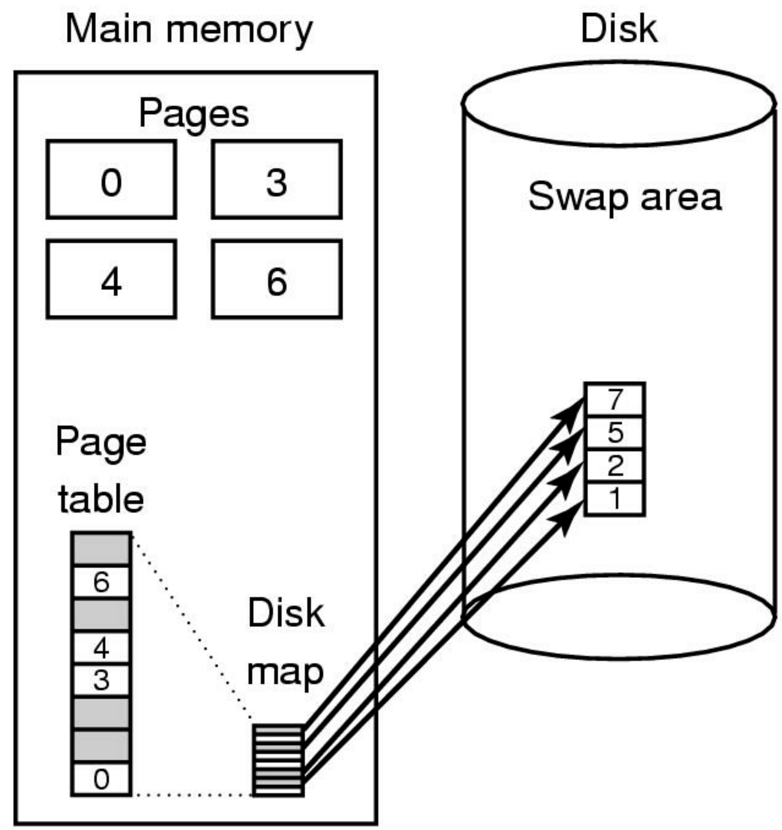
Approach #2

Store the pages in the swap in a random order.
View the swap file as a collection of free "swap frames".
Need to evict a frame from memory?
Find a free "swap frame".
Write the page to this place on the disk.
Make a note of where the page is.
Use the page table entry.
Just make sure the valid bit is still zero!
Next time the page is swapped out,
it may be written somewhere else.

Approach #2

This picture uses a
separate data structure
to tell where pages are.

But perhaps you can use
the page table entries.



Separation of Policy and Mechanism

Kernel contains

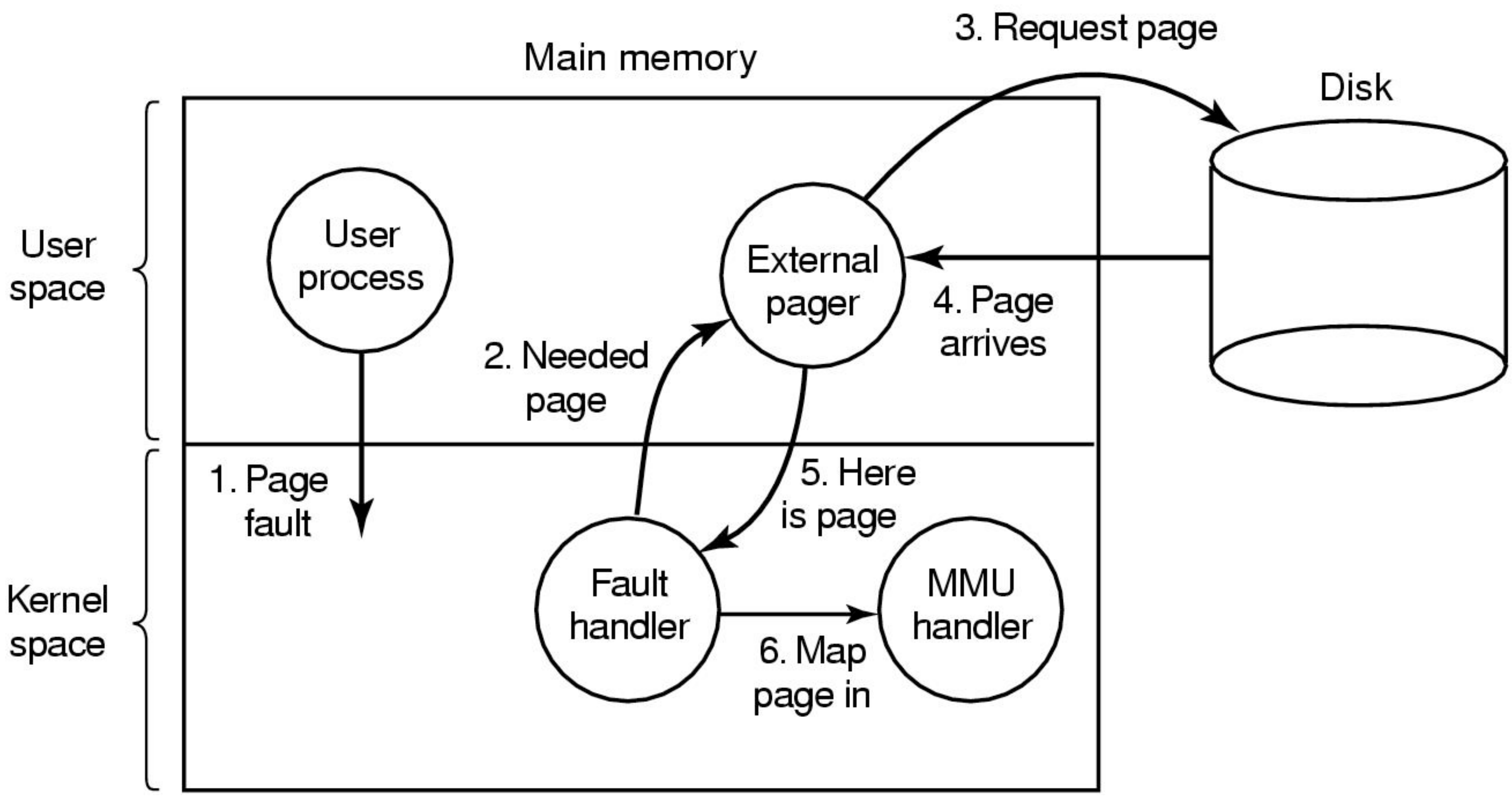
- **Code to manipulate the MMU**
  Machine dependent
- **Code to handle page faults**
  Machine independent

User-level Process

- **"External Pager"**
  Determines policy
    - **Which page to evict**
    - **When to perform disk I/O**
    - **How to manage the swap file**

Examples: Mach, Minix

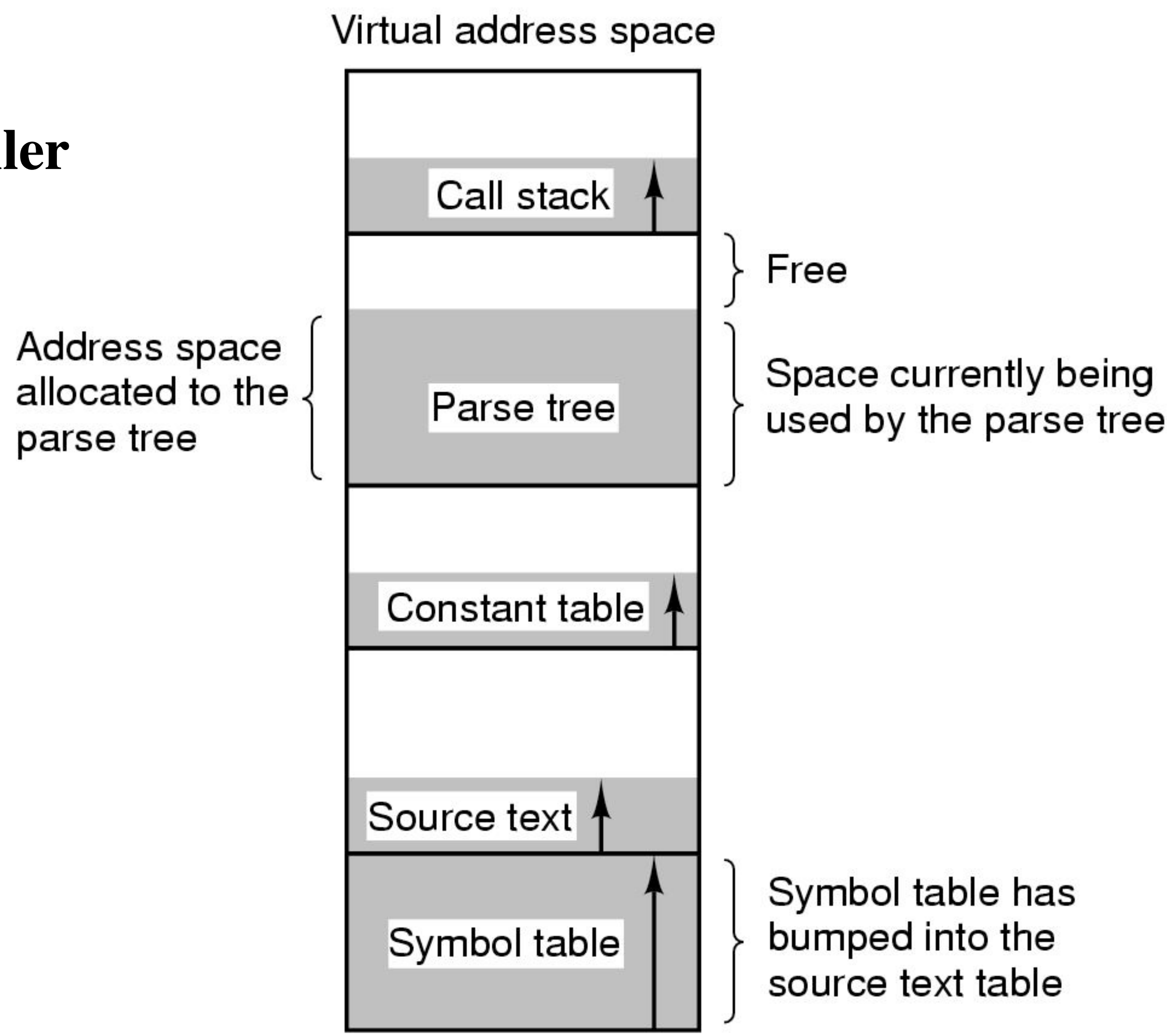
Separation of Policy and Mechanism



Problem with a Flat Address Space

Example:

A compiler



Segmentation

Traditional Virtual Address Space

“flat” address space (1 dimensional)

Segmented Address Space

Program made of several “pieces”

Each segment is like a mini-address space

Addresses within a segment start at zero

The program must always say which segment it means

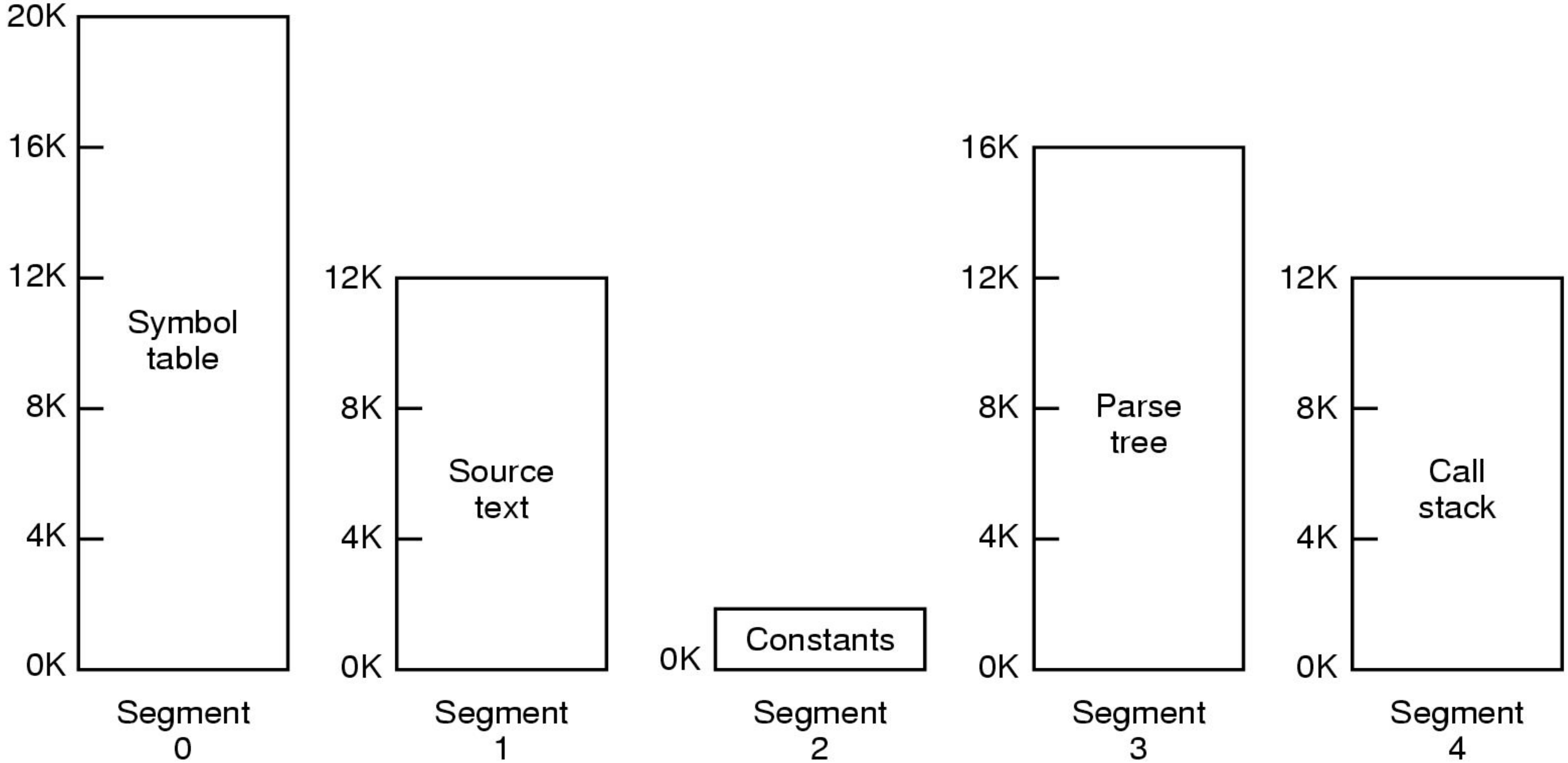
Addresses:

Segment + Offset

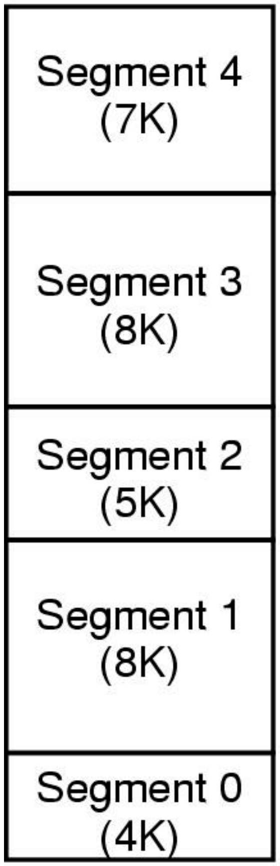
Each segment can grow independently of others

Segmented Memory

Each space grows, shrinks independently!



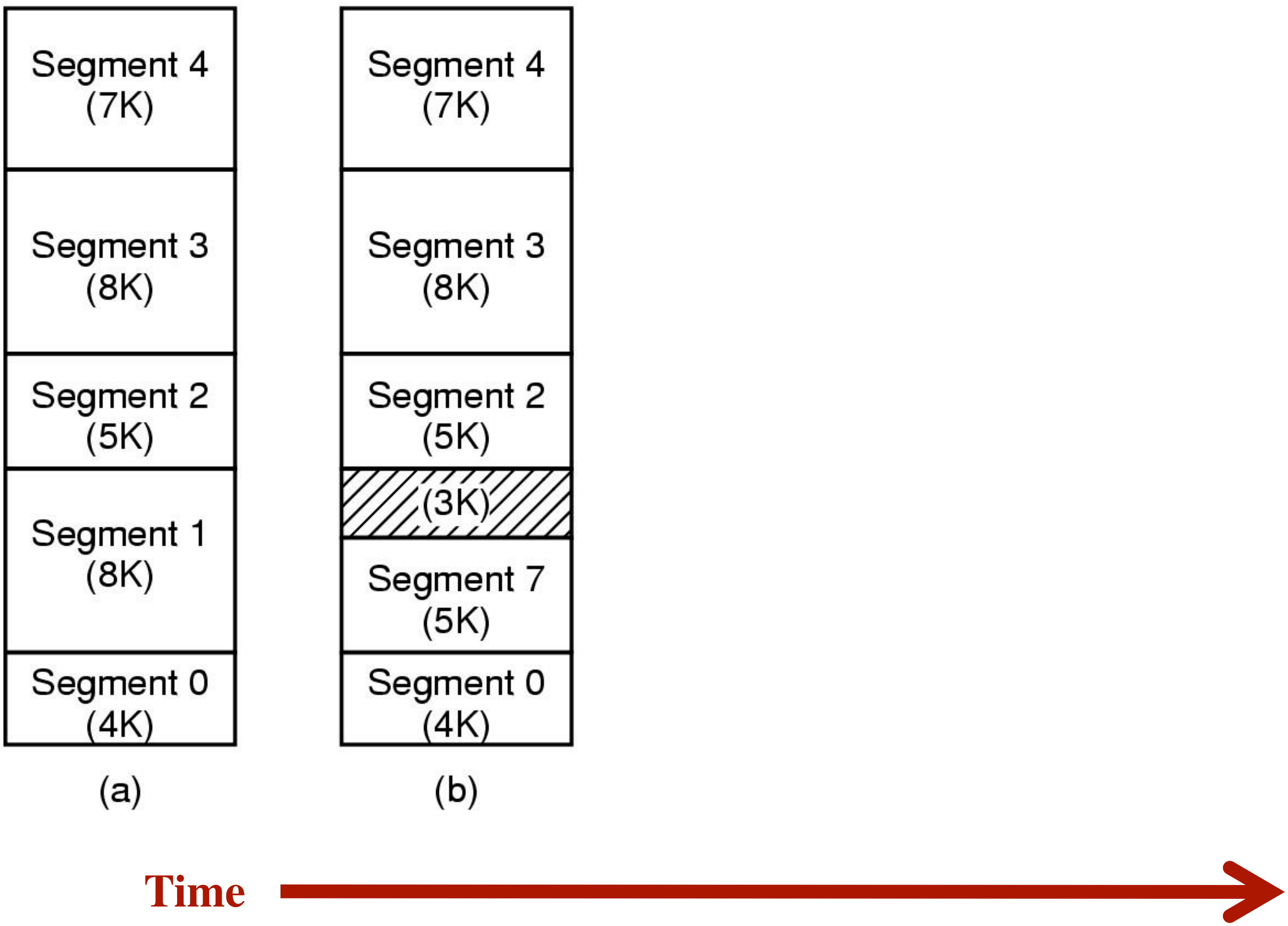
Implementation of Pure Segmentation



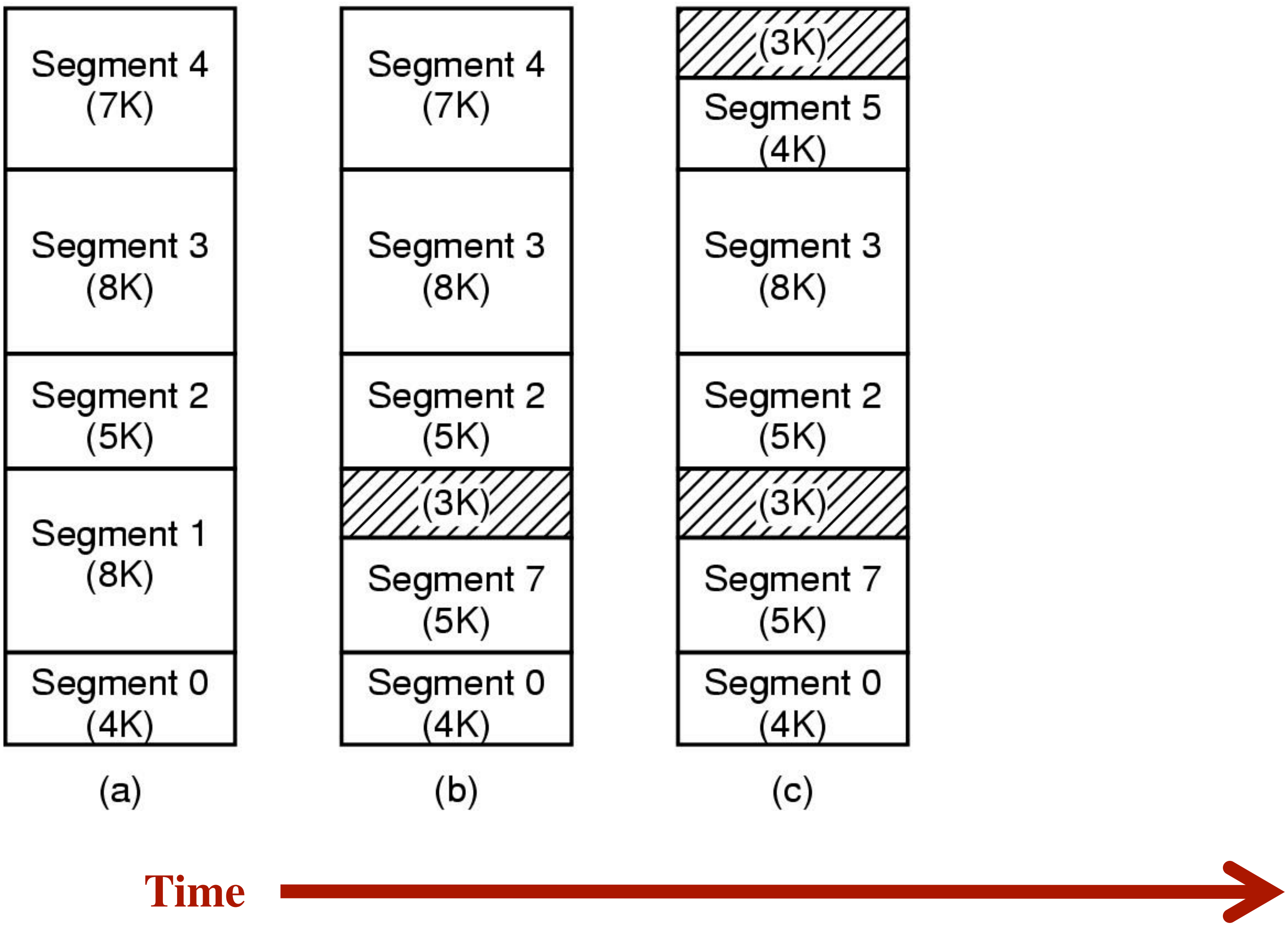
(a)

Time

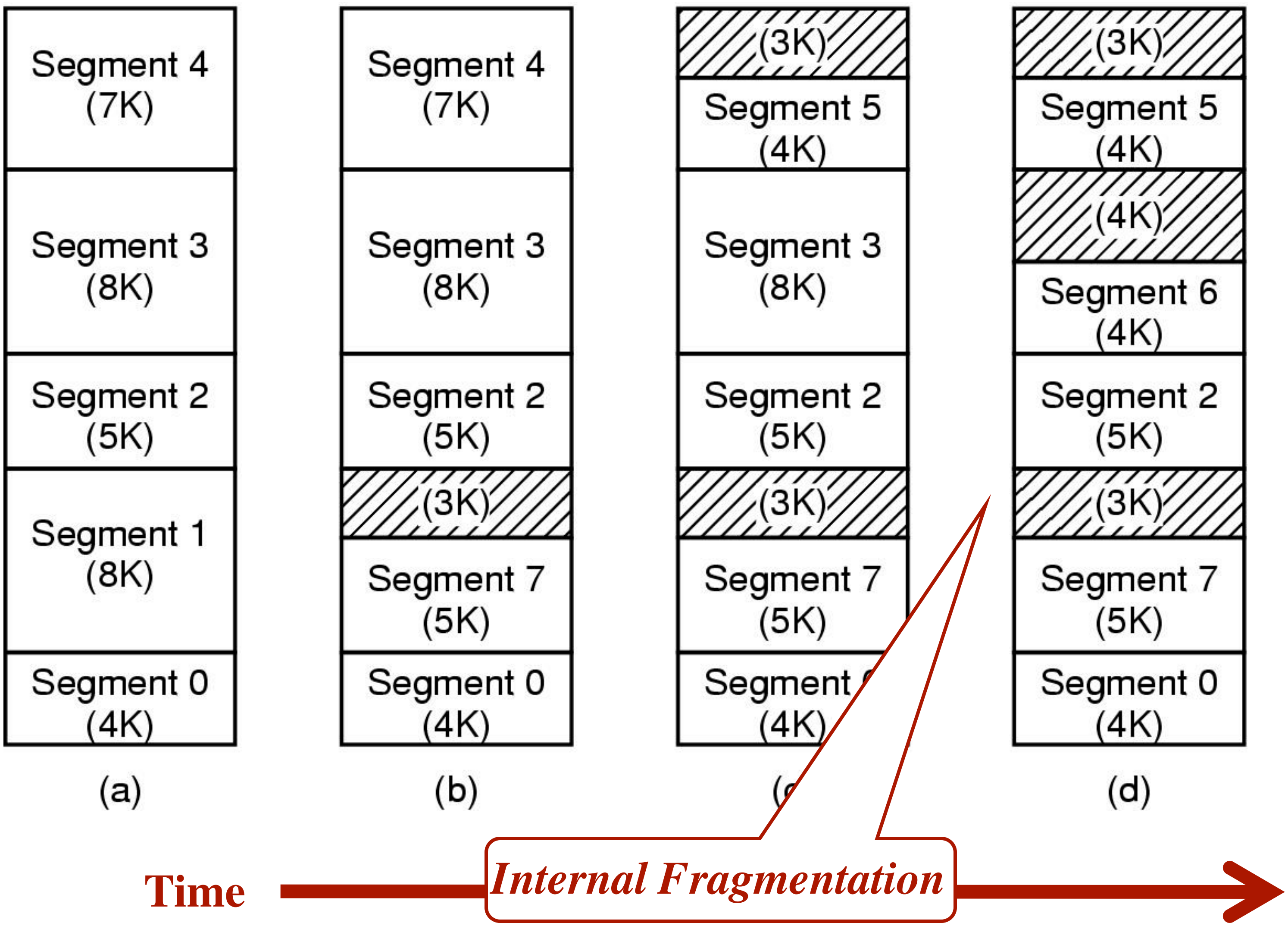
Implementation of Pure Segmentation



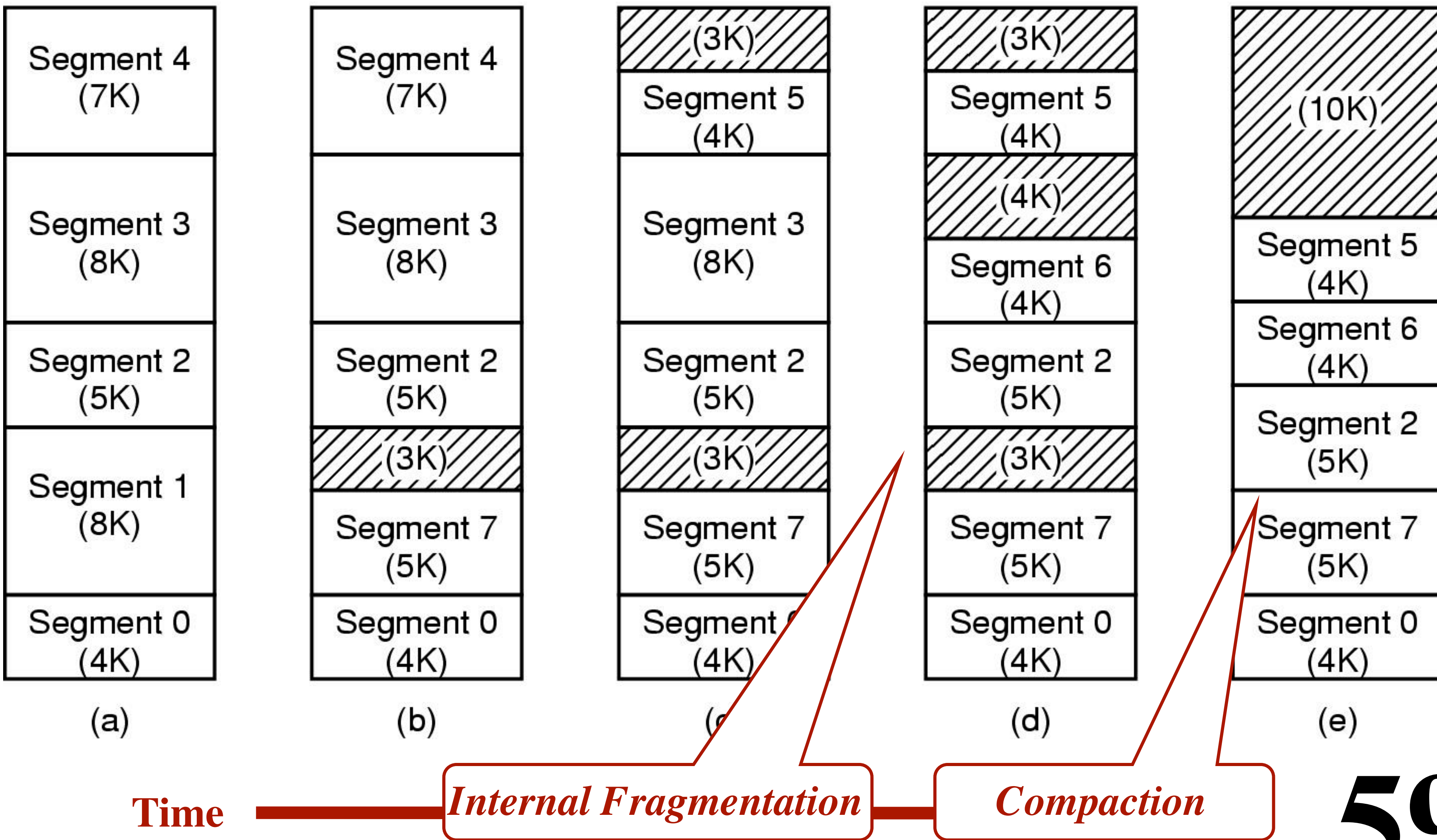
Implementation of Pure Segmentation



Implementation of Pure Segmentation

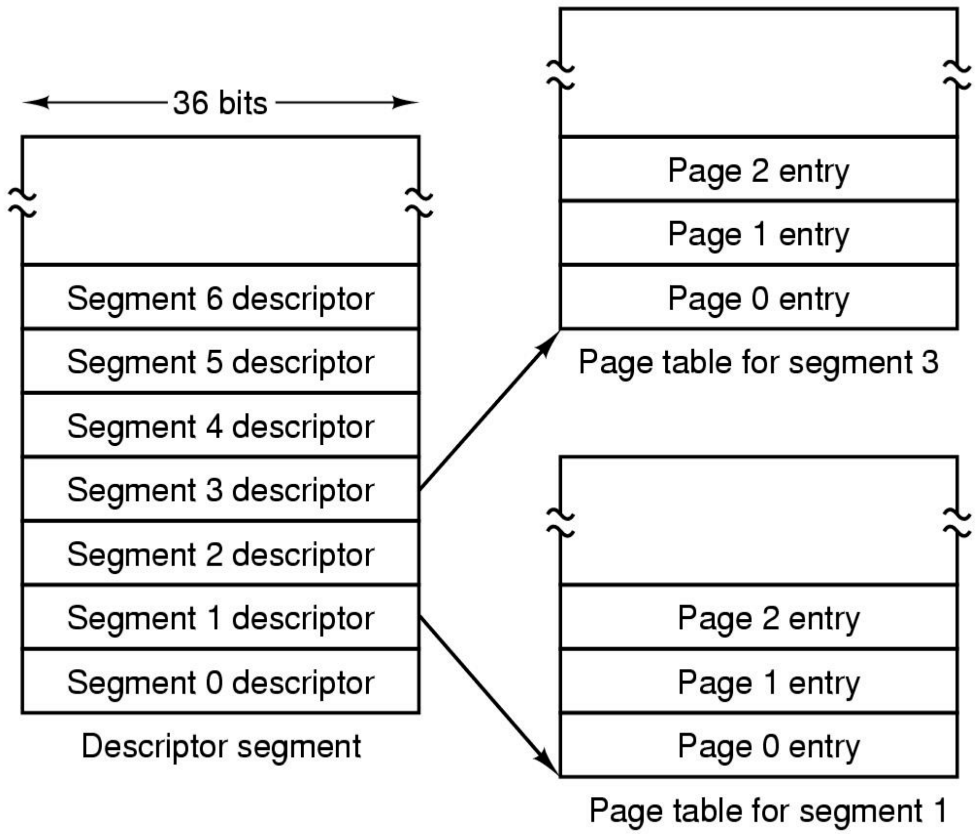


Implementation of Pure Segmentation



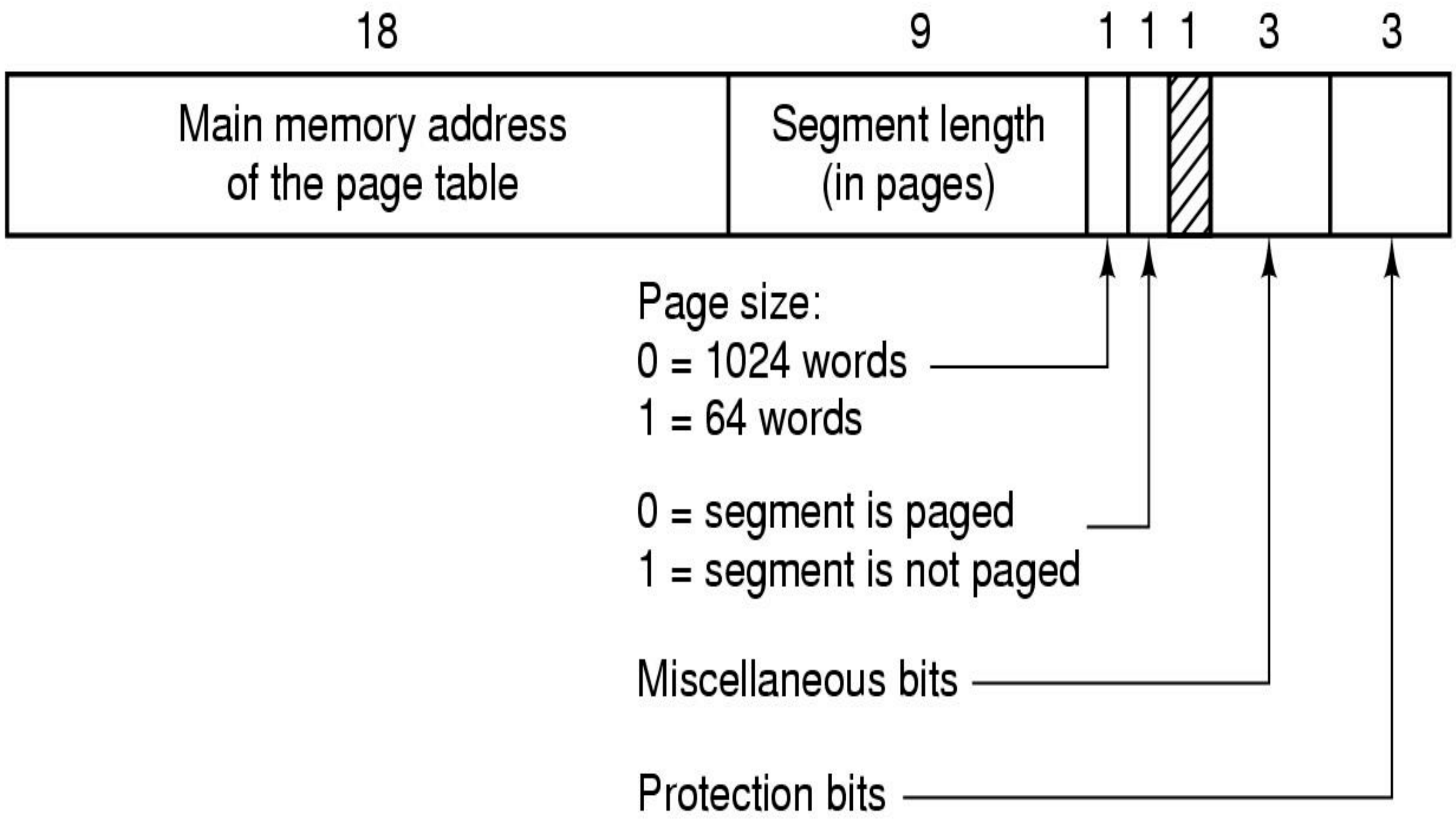
Segmenting with Paging (MULTICS)

Each segment is divided up into a pages.
A segment consists of several pages.
Each segment descriptor points to a page table.



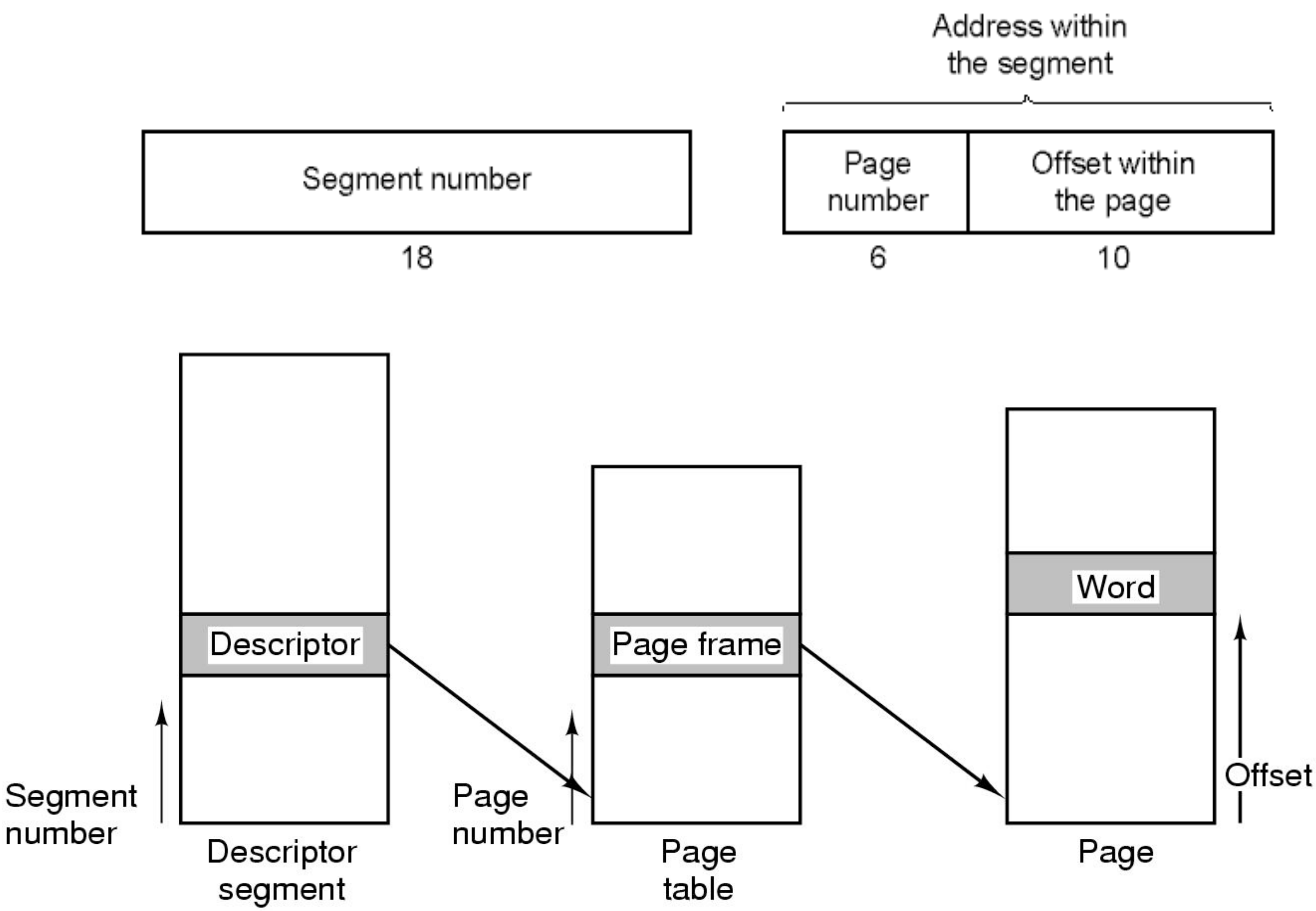
Segmenting with Paging (MULTICS)

Each entry in segment table...



Segmenting with Paging (MULTICS)

Each address is a 34-bit number.



Comparison

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