R

بسم الله الرحمن الرحیم

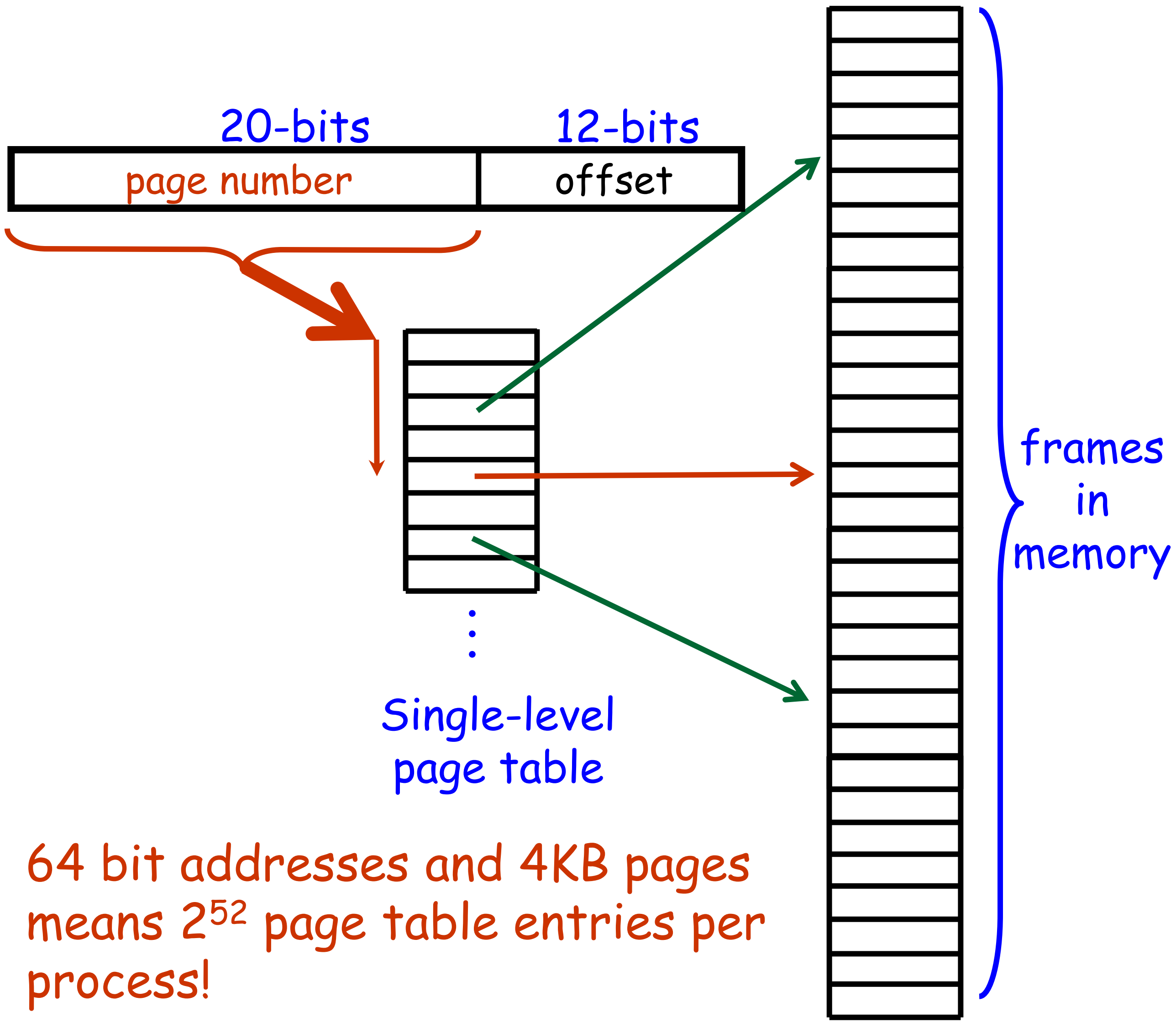
سیستم عامل

جلسه پانزدهم – حافظه‌ی مجازی

جلسه‌ی گذشته

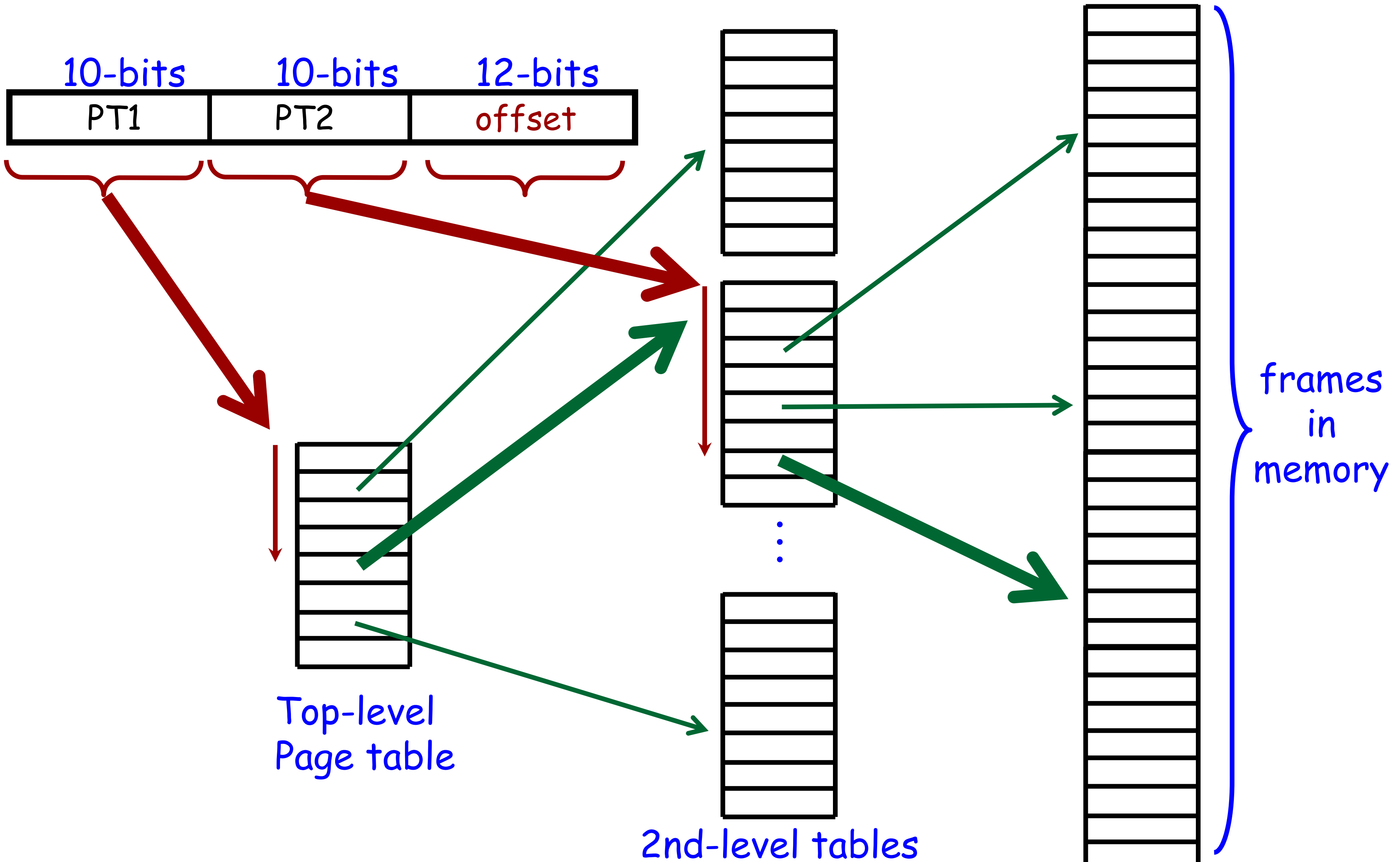
مدیریت حافظه – طراحی جدول صفحه

Single-Level Page Tables

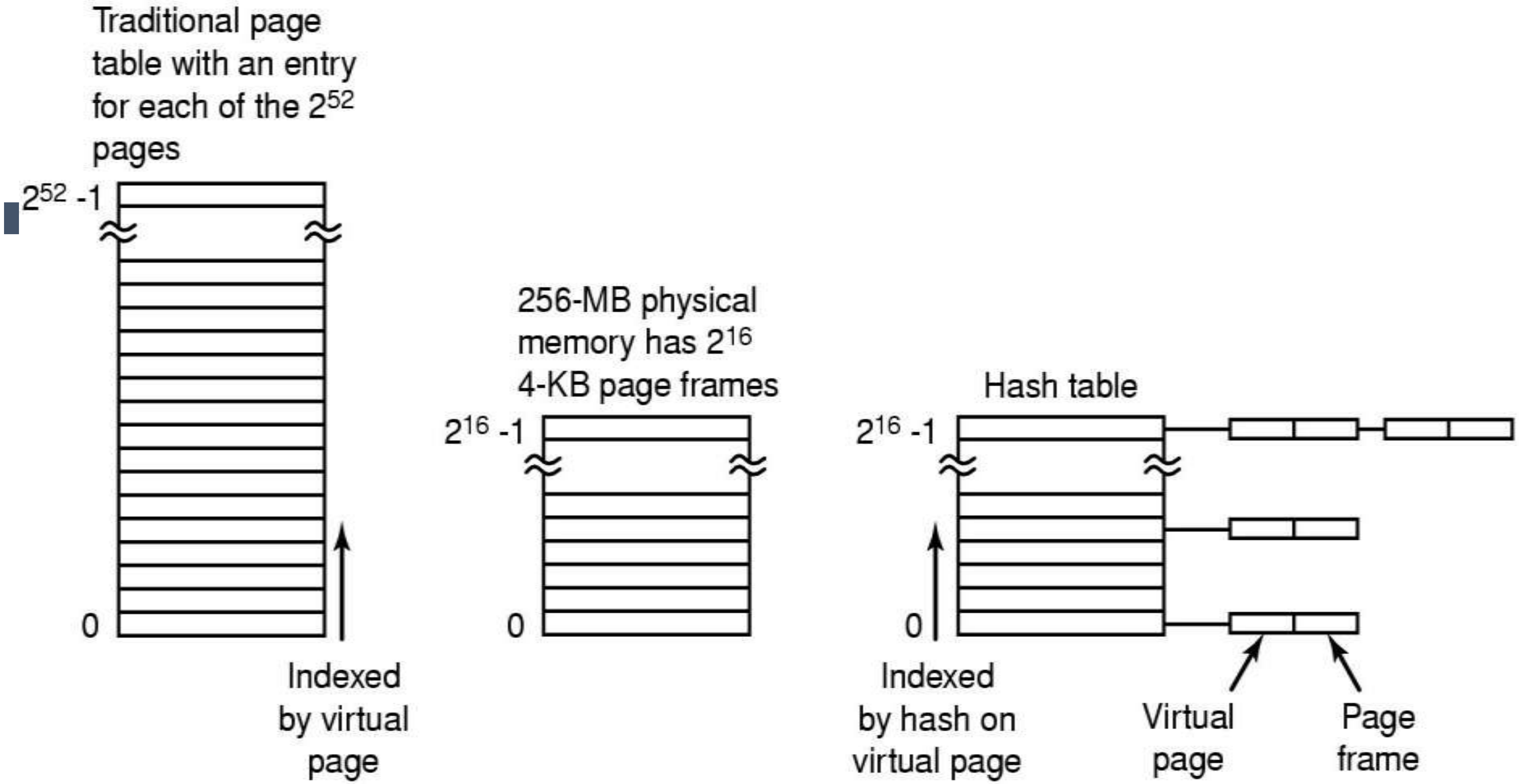


64 bit addresses and 4KB pages means 2^{52} page table entries per process!

Multi-Level Page Tables



Inverted page table



Memory Protection

- Protection though addressability
  - *If address translation only allows a process to access its own pages, it is implementing memory protection*
- But what if you want a process to be able to read and execute some pages but not write them?
  - *eg. the text segment*
- Or read and write them but not execute them?
  - *eg. the stack*
- Can we implement protection based on access type?

Protection Lookaside Buffer

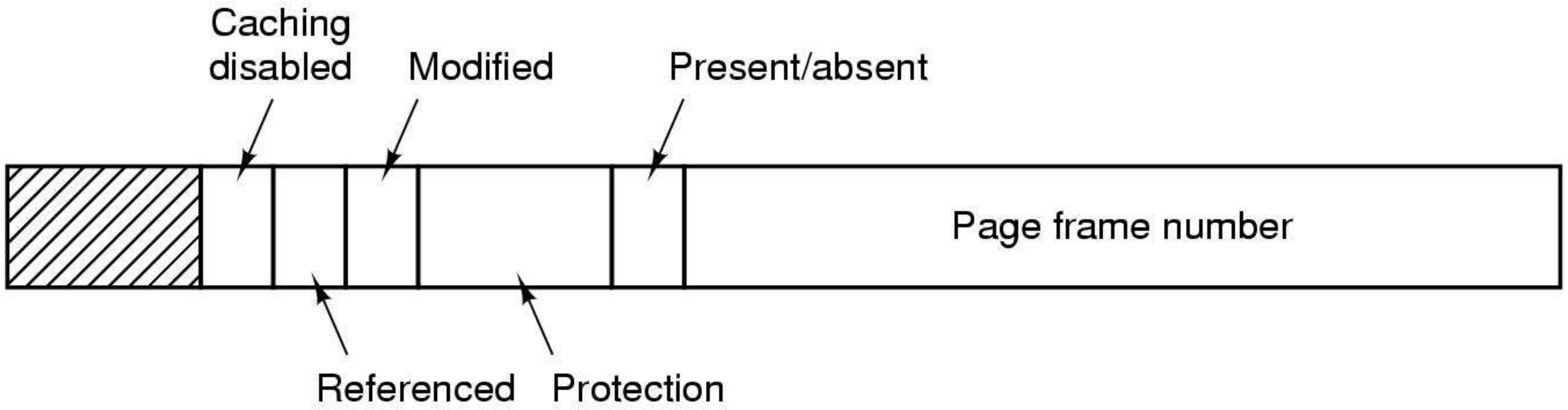
- A TLB is often used for more than just “translation”
- Memory accesses need to be checked for validity
  - *Does the address refer to an allocated segment of the address space?*
    - *If not: segmentation fault!*
  - *Is this process allowed to access this memory segment?*
    - *If not: segmentation/protection fault!*
  - *Is the type of access valid for this segment?*
    - Read, write, execute …?
    - *If not: protection fault!*

Protection Checking With a TLB

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

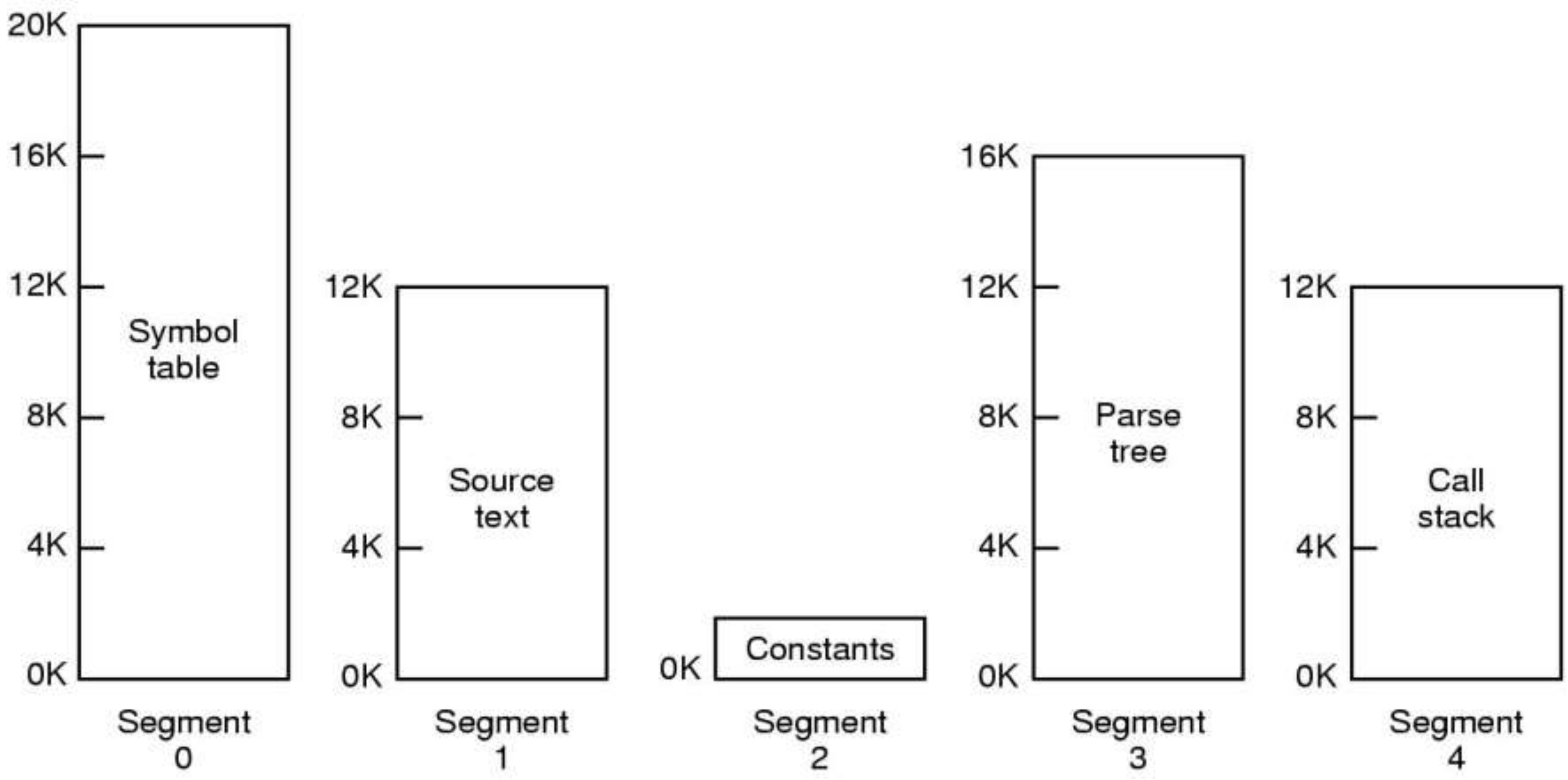
Page Grain Protection

A typical page table entry with support for page grain protection



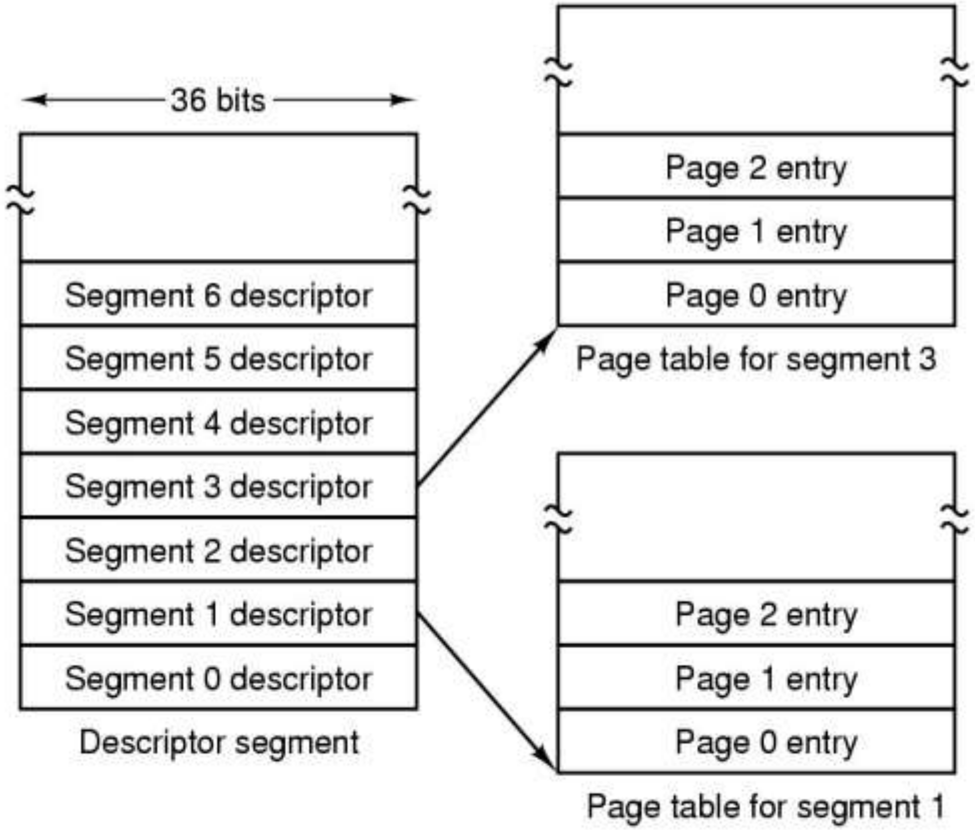
Segmented Memory

- Each space grows, shrinks independently!



Paged Segments in MULTICS

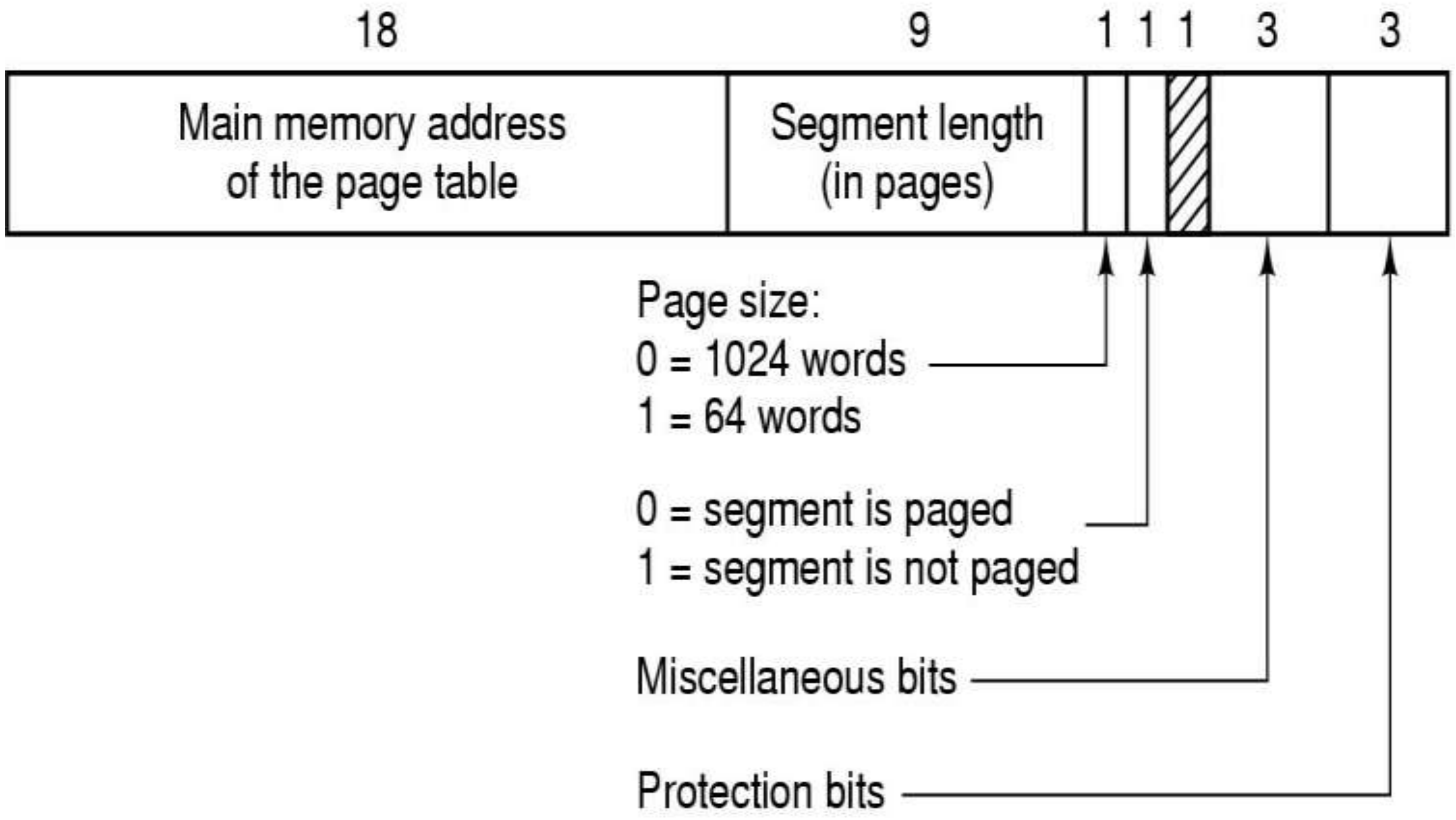
- Each segment is divided up into pages.
- Each segment descriptor points to a page table.



Paged Segments in MULTICS

(Multiplexed Information and Computing Service)

- Each entry in segment table



The MULTICS TLB

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

Comparison field					
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

Memory Management in Linux

- 64-bit addressable space, with only 48 bits currently used.
- No segmentation, with protection handled at the page level.
- Separate user and kernel spaces through negative addressing.
- Support for variable page sizes (4 KB, 2 MB, 1 GB).
- A 4-level page table hierarchy.

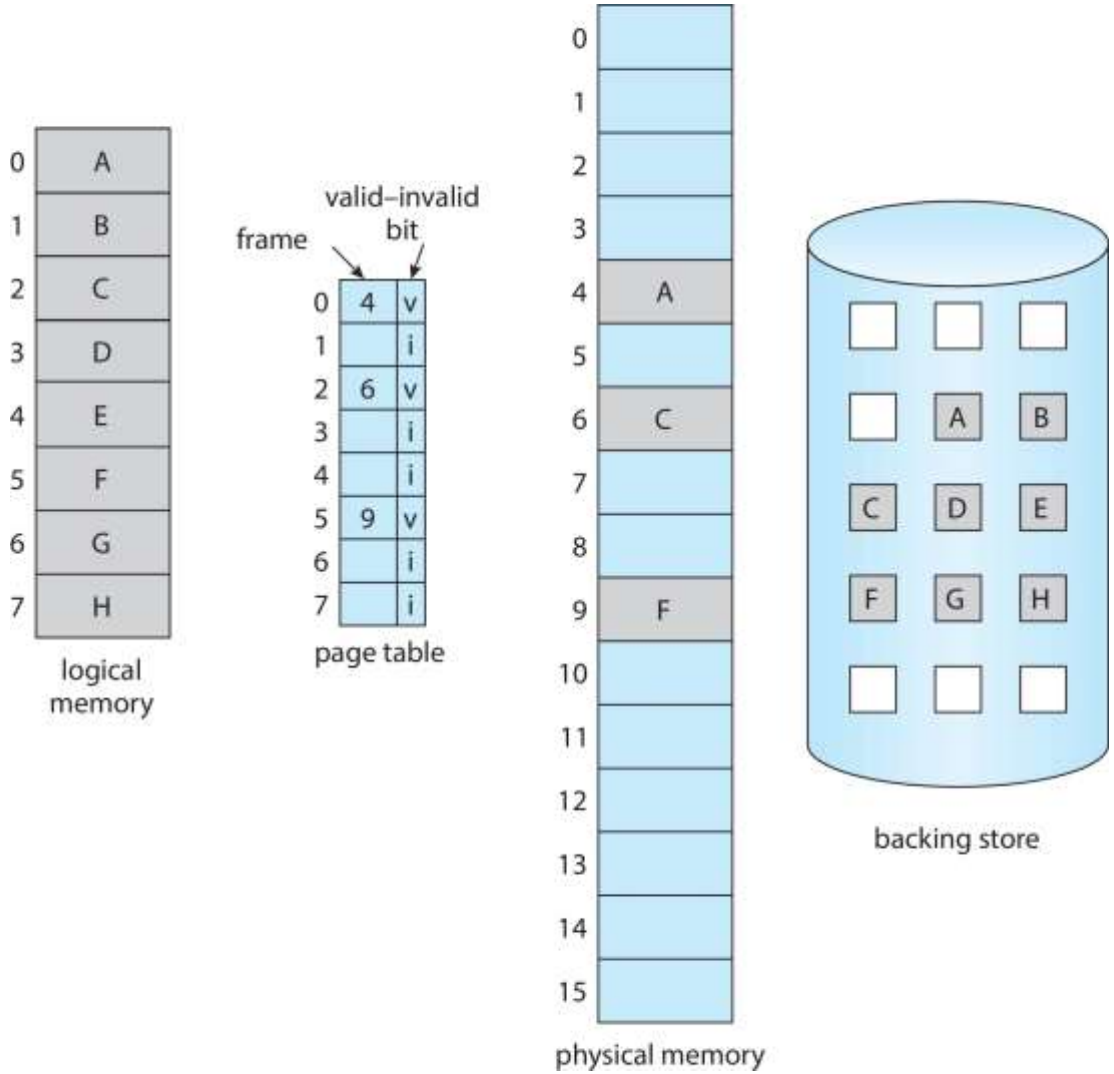
جلسه‌ی جدید

حافظه‌ی مجازی

Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory
  - *Only part of the program needs to be in memory for execution*
  - *Logical address space can therefore be much larger than physical address space*
  - *Allows address spaces to be shared by several processes*
  - *Allows for more efficient process creation*
  - *More programs running concurrently*
  - *Less I/O needed to load or swap processes*

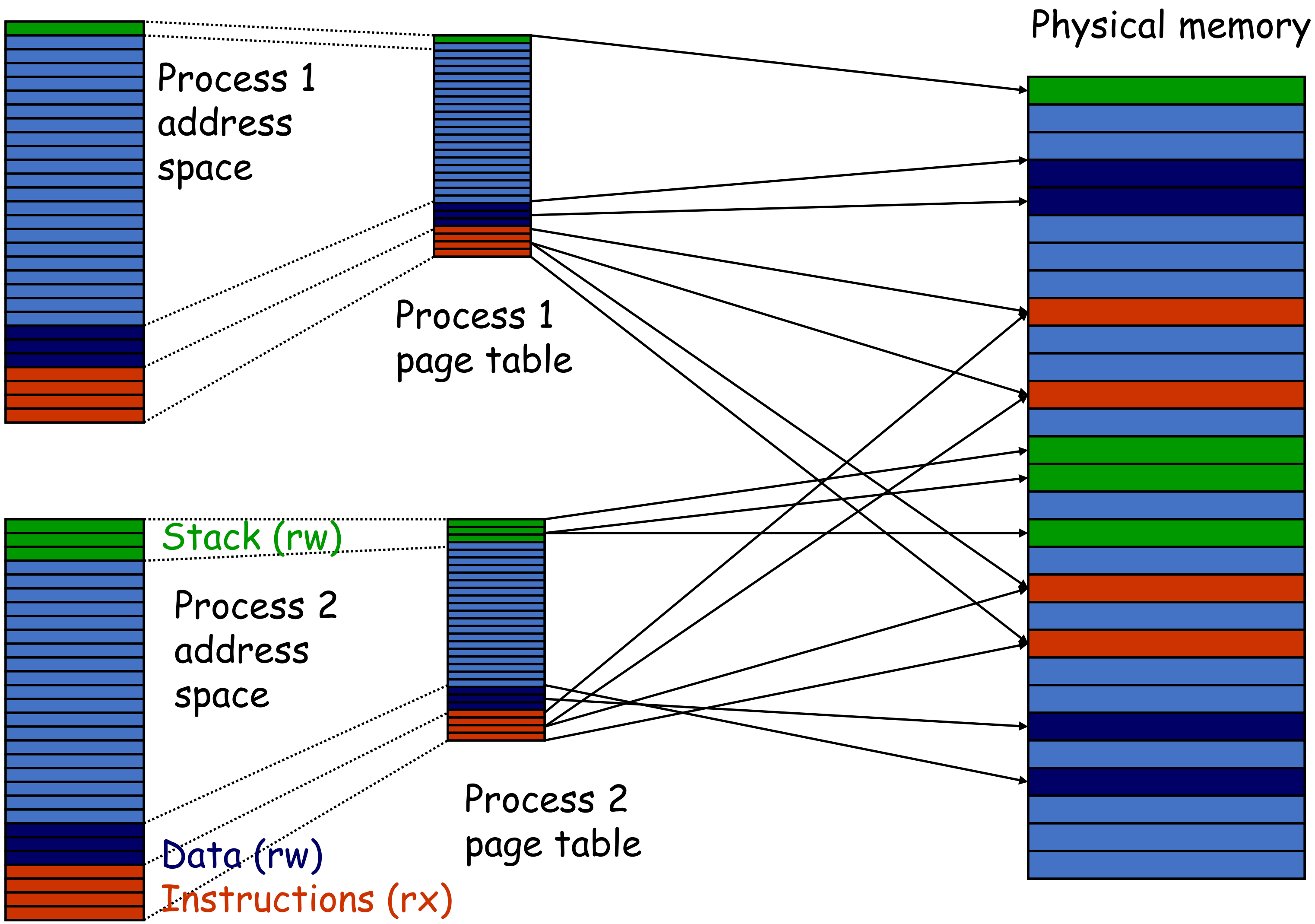
Virtual Memory That is Larger Than Physical Memory



اشتراک‌گذاری صفحه

Page Sharing

- In a large multiprogramming system some users run the same program at the same time
  - *Why have more than one copy of each page in memory?*
- *Goal:*
  - *Share pages among "processes" (not just threads!)*
    - Cannot share writable pages
    - If writable pages were shared processes would notice each other's effects
    - Text segment can be shared



Physical memory
Process 1
address
space
Process 1
page table
Stack (rw)
Process 2
address
space
Process 2
page table
Data (rw)
Instructions (rx)

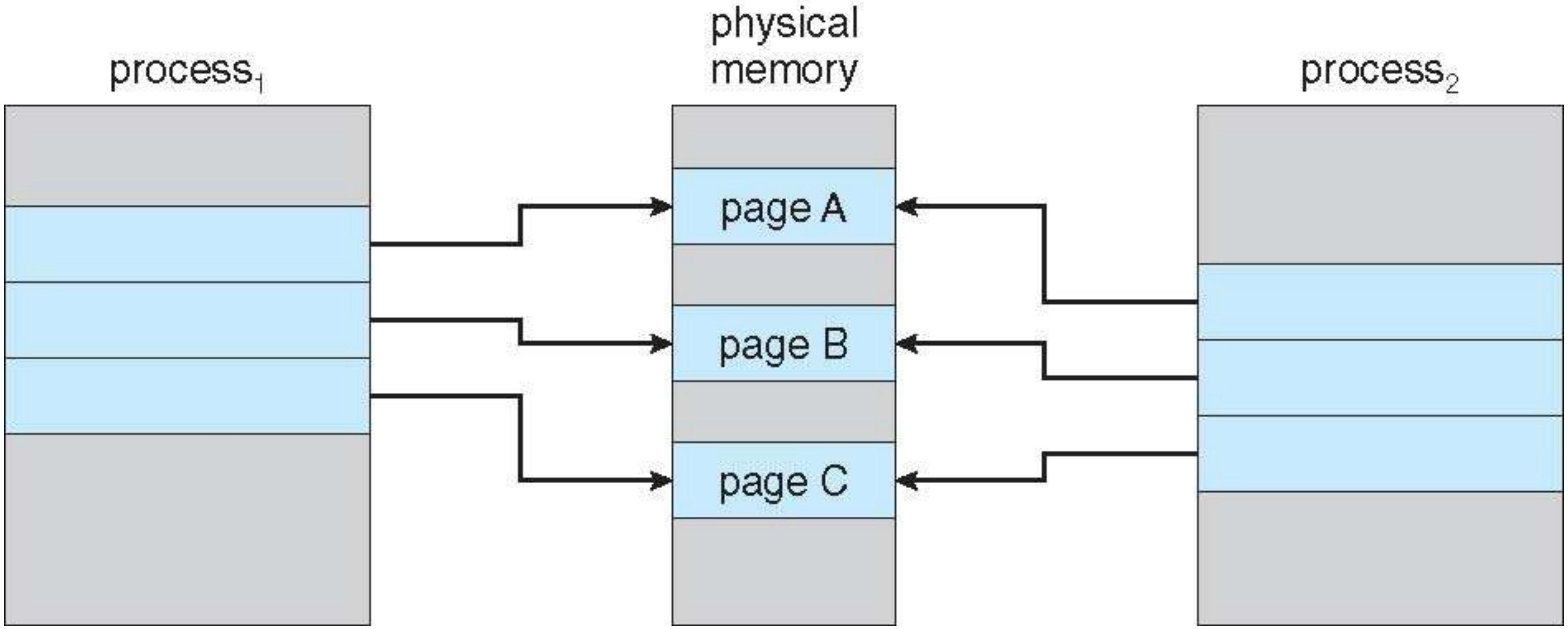
Page Sharing in Fork System Call

- Normal usage: copy the parent's virtual address space and immediately do an "Exec" system call
  - *Exec overwrites the calling address space with the contents of an executable file (ie a new program)*
- Desired Semantics:
  - *Pages are copied, not shared*
- Observations
  - *Copying every page in an address space is expensive!*
  - *Processes can't notice the difference between copying and sharing unless pages are modified!*

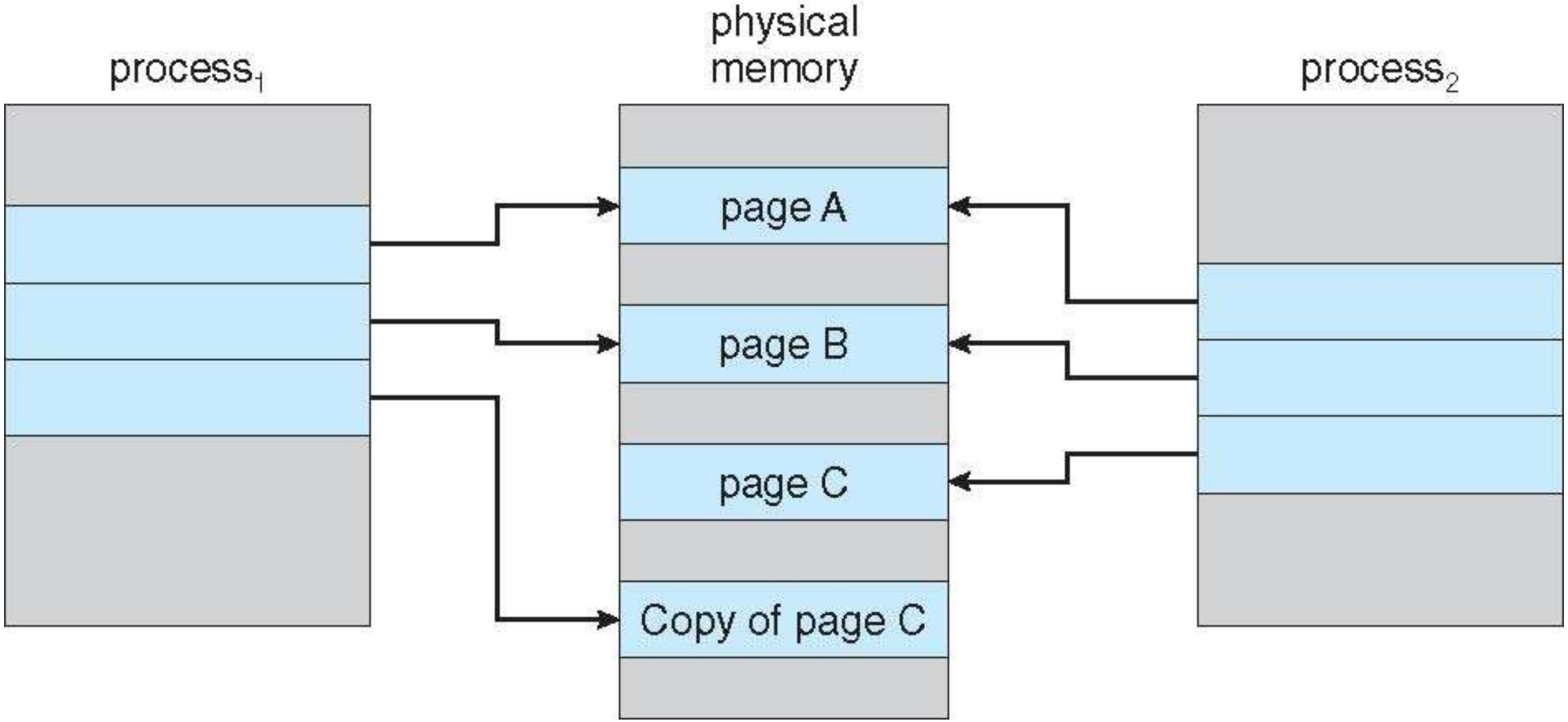
Copy-on-Write Page Sharing

- Initialize new page table, but point entries to existing page frames of parent, i.e. share pages
- Temporarily mark all pages “read-only”
- Continue to share all pages until a protection fault occurs
- Protection fault (copy-on-write fault):
  - *Is this page really read only or is it writable but temporarily protected for copy-on-write?*
  - *If it is writable, copy the page, mark both copies writable, resume execution as if no fault occurred*
- This is an interesting new use of protection faults!

Before Process 1 Modifies Page C



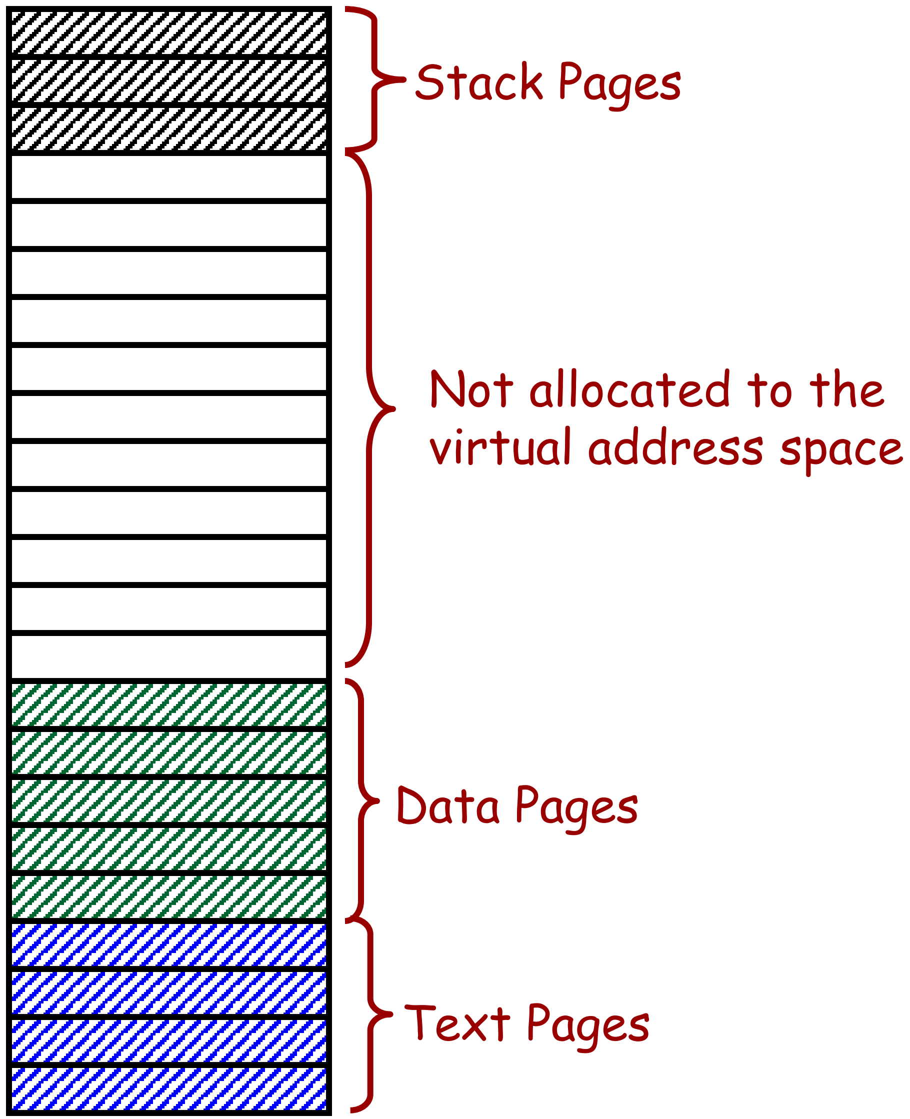
After Process 1 Modifies Page C



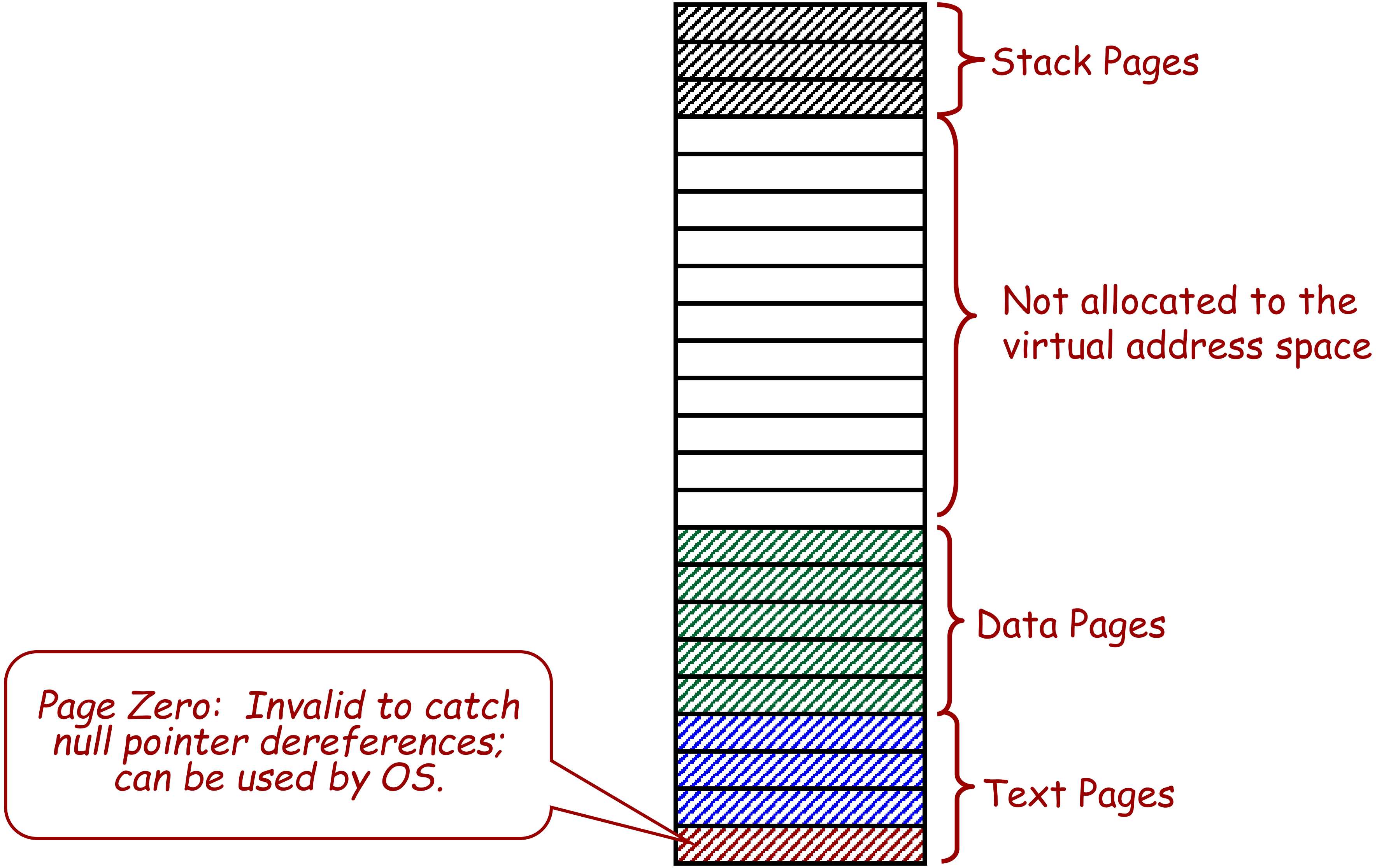
Page Management System Calls

- Goal: Allow some processes more control over paging!
- System calls added to the kernel
  - *A process can request a page before it is needed*
    - Allows processes to grow (heap, stack etc)
  - *Processes can share pages*
    - Allows fast communication of data between processes
    - Similar to how threads share memory
      - *… so what is the difference?*

Unix Processes



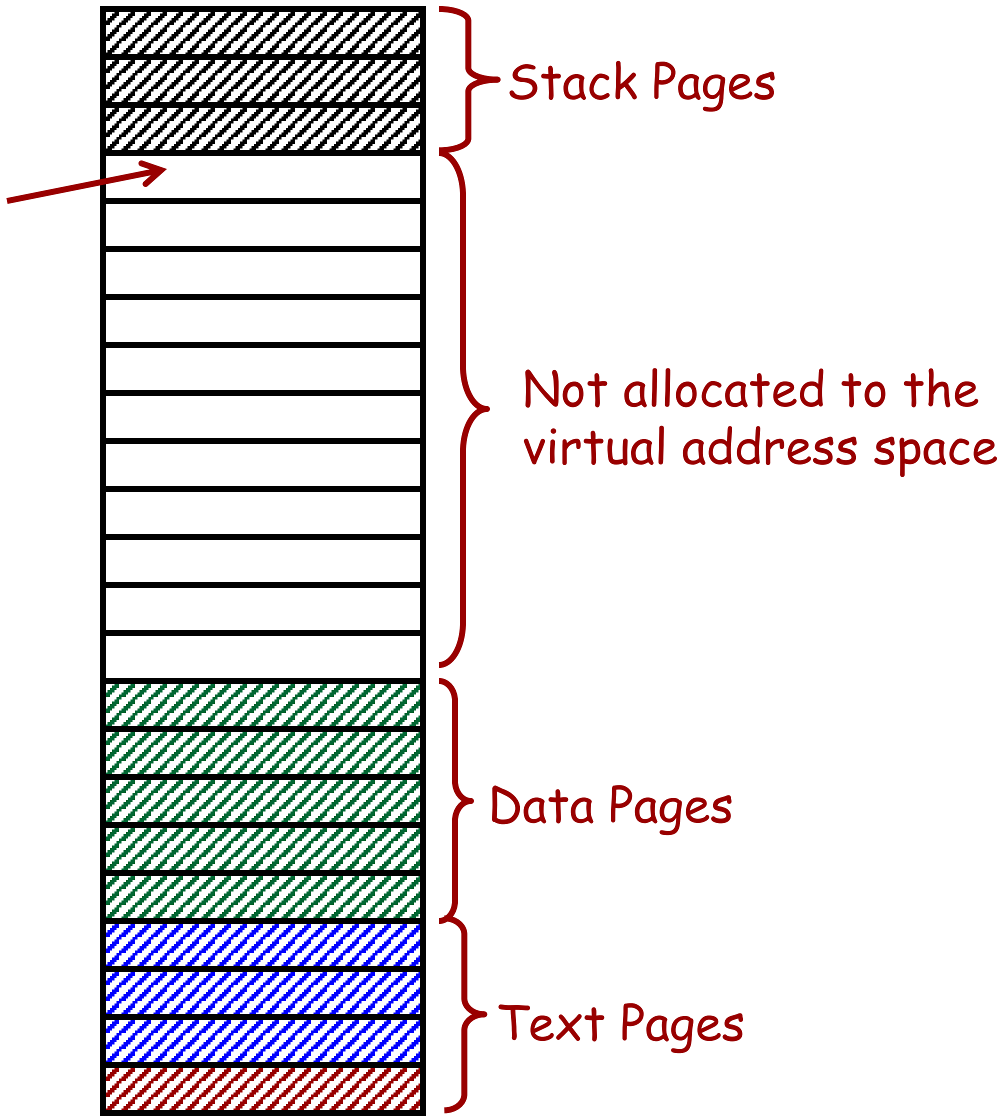
Unix Processes



Unix Processes

The stack grows;

Page requested here



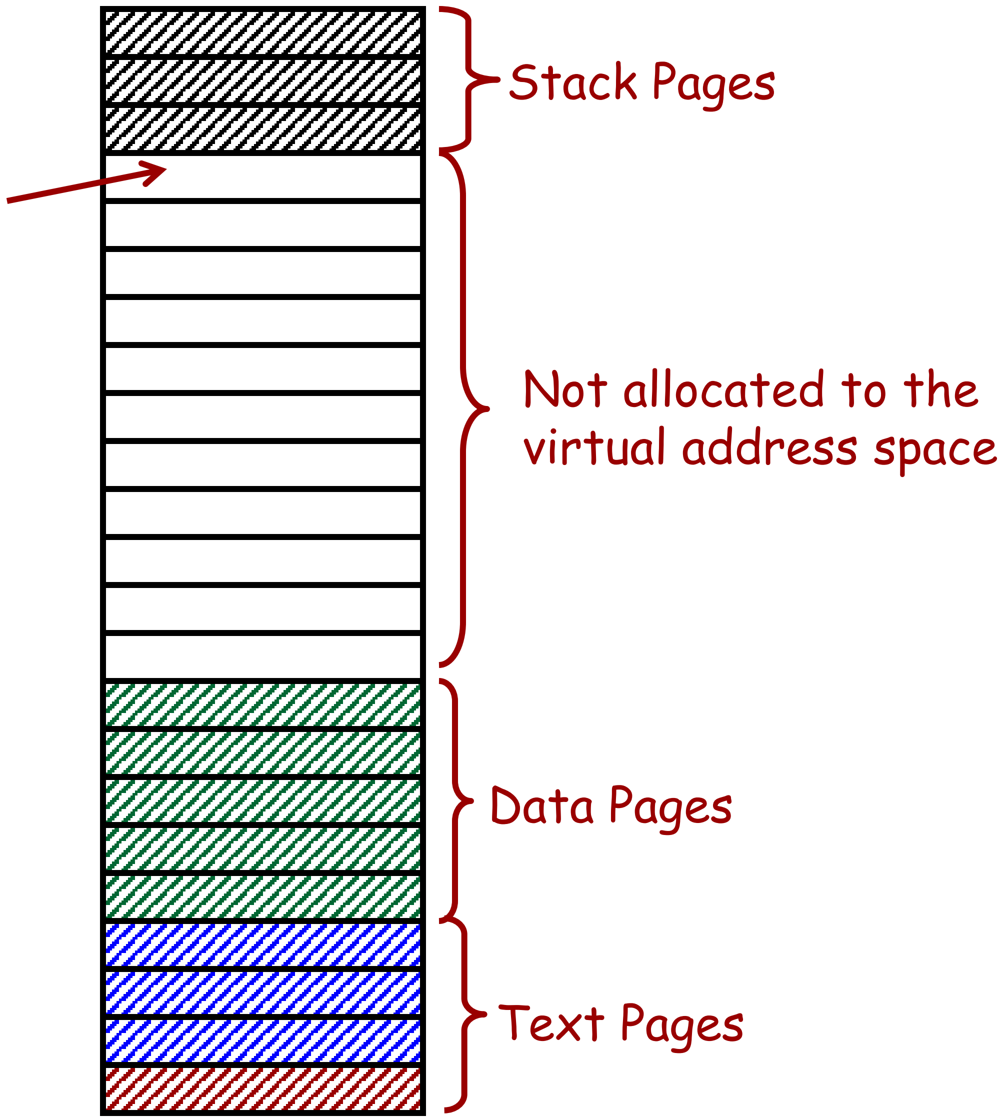
Unix Processes

The stack grows;

Page requested here

A new page is allocated

and process continues



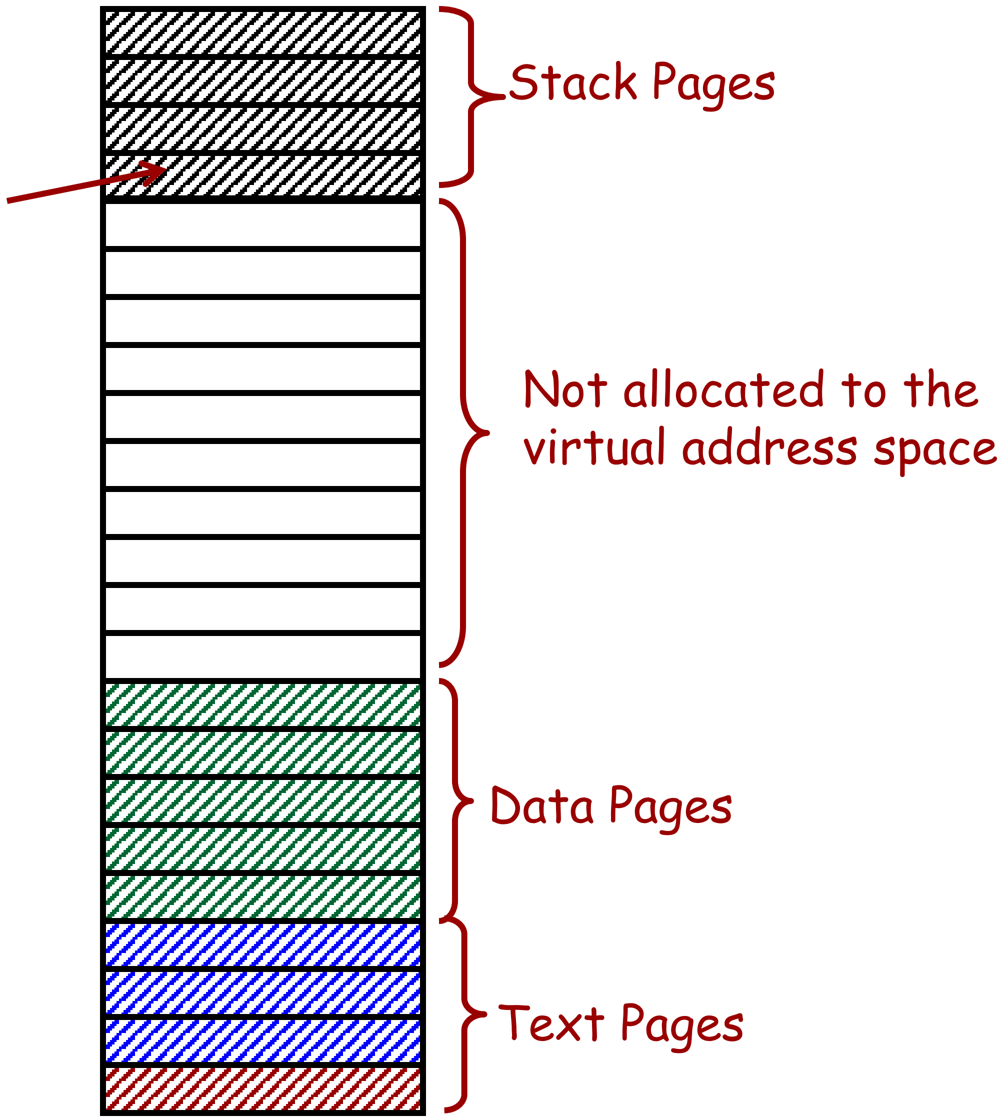
Unix Processes

The stack grows;

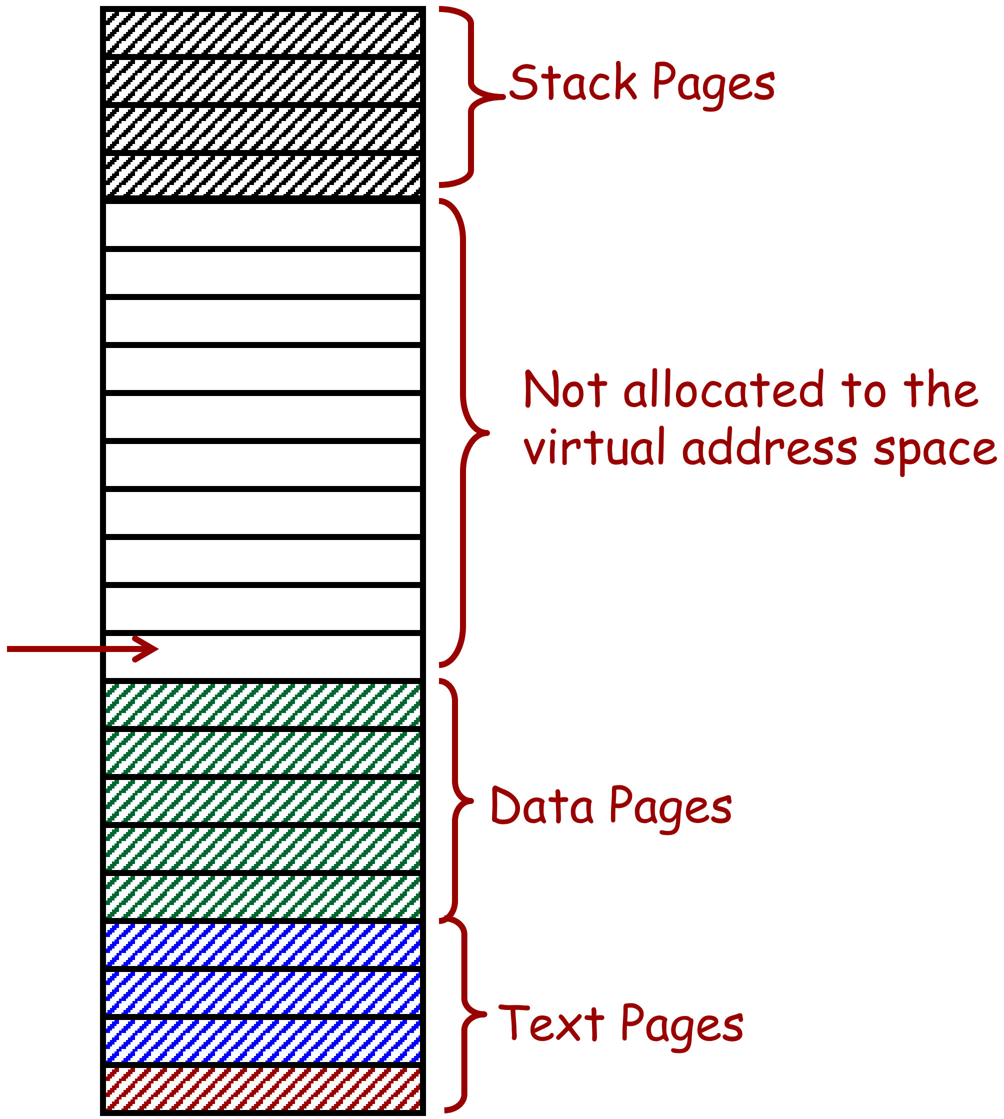
Page requested here

A new page is allocated

and process continues



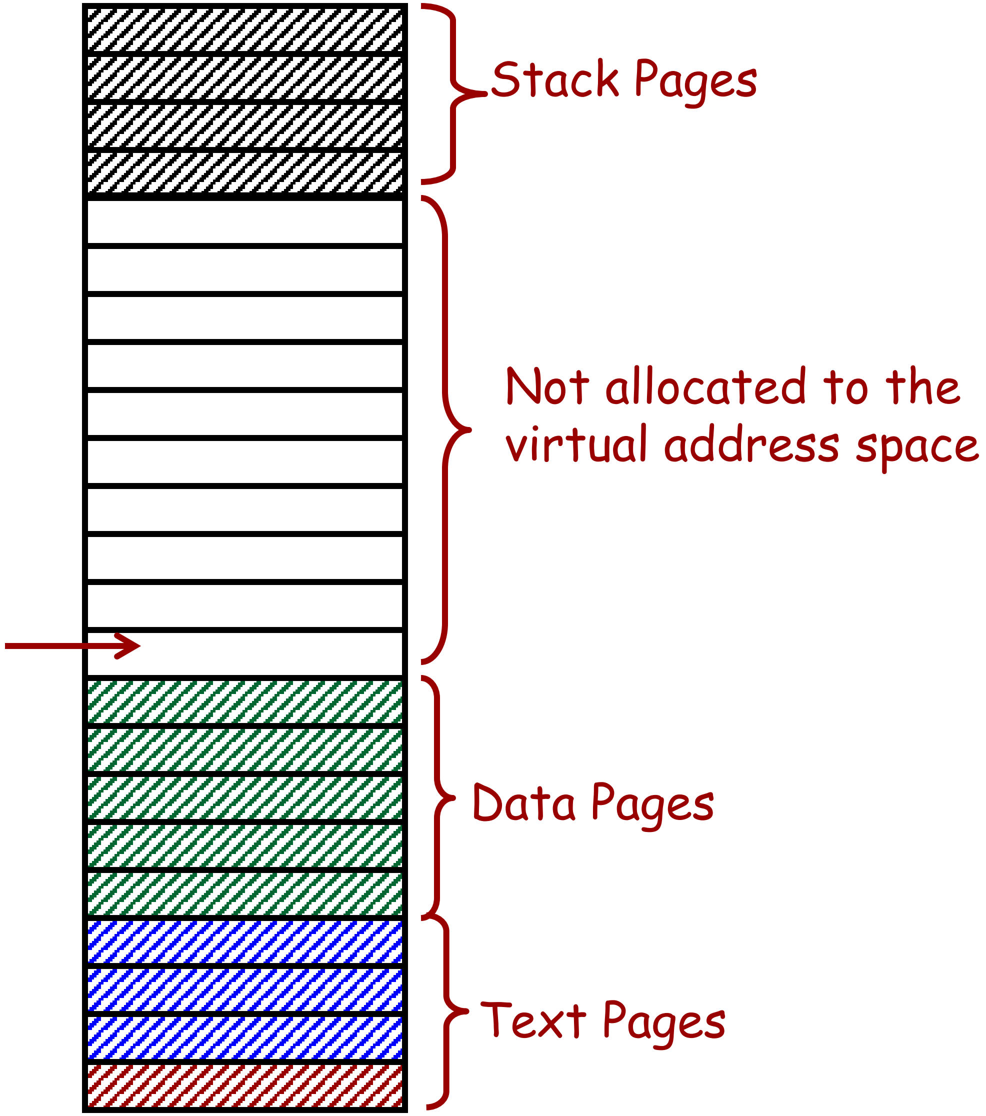
Unix Processes



The heap grows;

Page requested here

Unix Processes



The heap grows;

Page requested here

A new page is allocated

and process continues

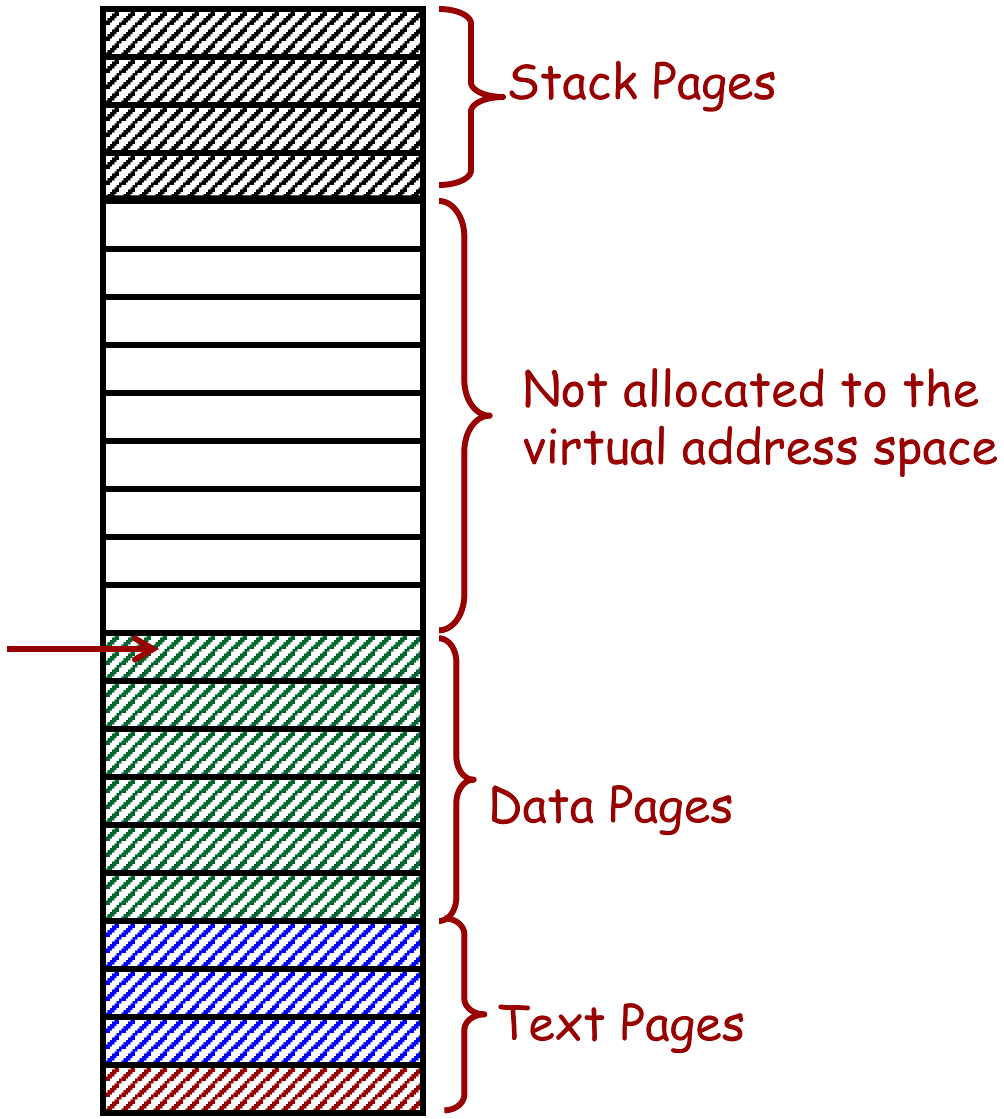
Unix Processes

The heap grows;

Page requested here

A new page is allocated

and process continues



Page Sharing with mmap syscall

- mmap
  - *Maps a file or memory region into the process's virtual memory*

نقش هسته در حافظه‌ی مجازی

Virtual Memory Implementation

- When is the kernel involved?
  - *Process creation*
  - *Process is scheduled to run*
  - *A fault occurs*
  - *Process termination*

Virtual Memory Implementation

- Process creation
  - *Determine the process size*
  - *Create new page table*

Virtual Memory Implementation

- Process is scheduled to run
  - *MMU is initialized to point to new page table*
  - *TLB is flushed (unless it’s a tagged TLB)*

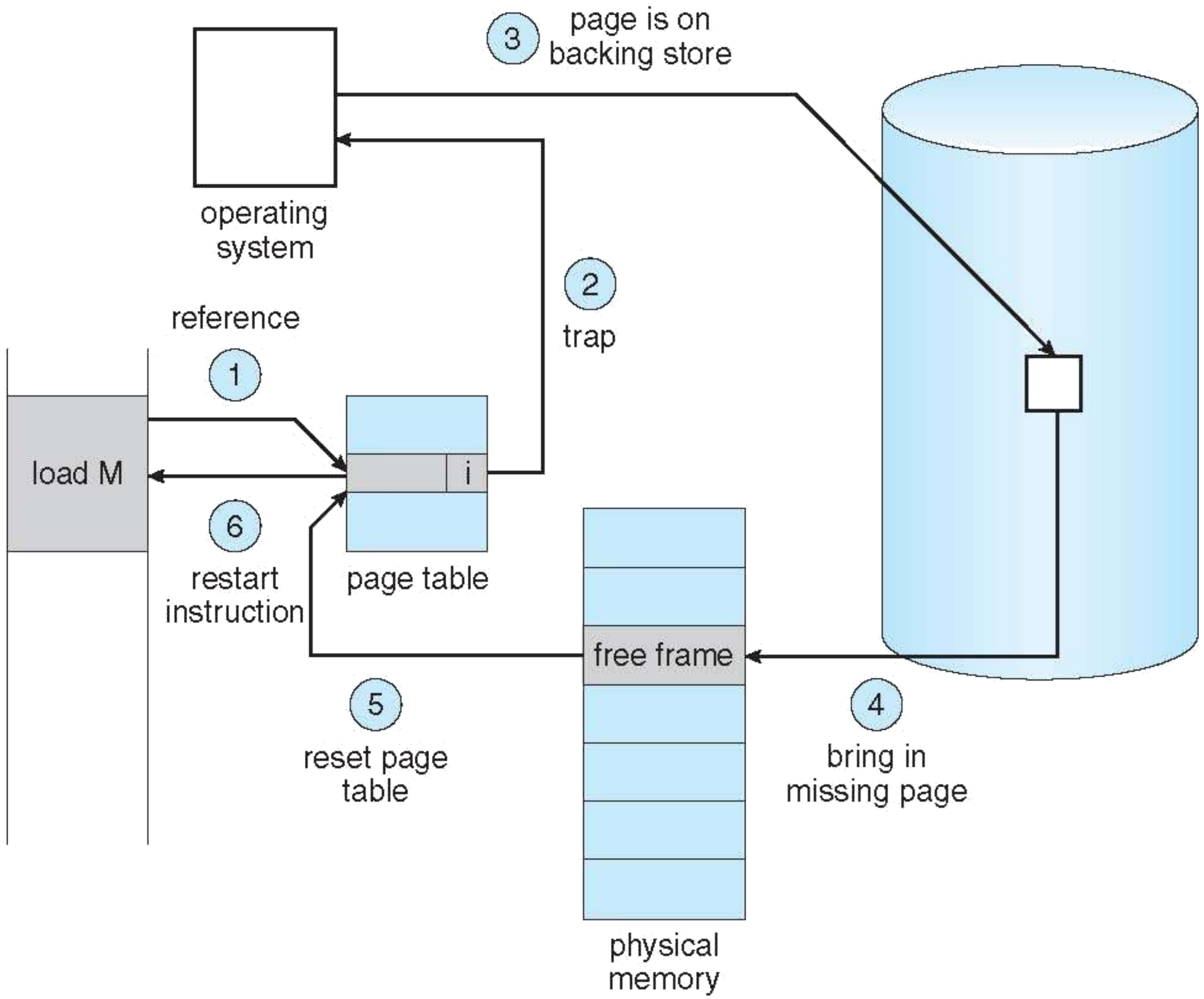
Virtual Memory Implementation

- A fault occurs
  - *Could be a TLB-miss fault, segmentation fault, protection fault, copy-on-write fault …*
  - *Determine the virtual address causing the problem*
  - *Determine whether access is allowed, if not terminate the process*
  - *Refill TLB (TLB-miss fault)*
  - *Copy page and reset protections (copy-on-write fault)*
  - *Swap an evicted page out & read in the desired page (page fault)*

Virtual Memory Implementation

- Process termination
  - *Release / free all frames (if reference count is zero)*
  - *Release / free the page table*

Handling a Page Fault



Handling a Page Fault (More Detail)

1. Hardware traps to kernel (PC and SR are saved on stack)
2. Save the other registers
3. Determine the virtual address causing the problem
4. Check validity of the address
   - *Determine which page is needed*
   - *May need to kill the process if address is invalid*
5. Find the frame to use (page replacement algorithm)
6. Is the page in the target frame dirty?
   - *If so, write it out (& schedule other processes)*
7. Read in the desired frame from swapping file
8. Update the page tables
9. *(continued)*

Handling a Page Fault (More Detail)

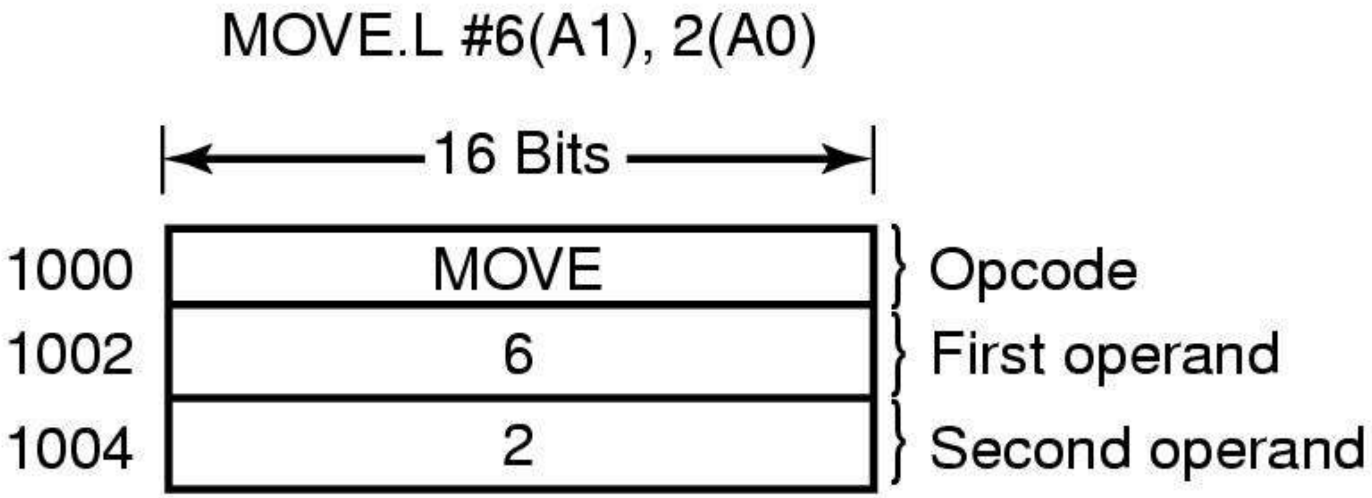
9. Back up the current instruction (ie., the *faulting instruction*)
10. Schedule the faulting process to run again
11. Return to scheduler
12. ...
13. Reload registers
14. Resume execution

پیچیدگی‌ها

- بعضی دستورالعمل‌ها از چند word تشکیل شدند، چطوری PC را برگردونیم؟
- وسط page fault، چون کار با دیسک داریم، context swtich رخ بده و ....

Problem: Backing up the PC

- 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?



Solution: Precise Interrupts

- Lots of architecture-specific code in the kernel

- Hardware support (precise interrupts)
  - *Dump internal CPU state into special registers*
  - *Make special registers accessible to kernel*

Complications

- What if you swapped out the page containing the first operand in order to bring in the second one?

Locking Pages in Memory

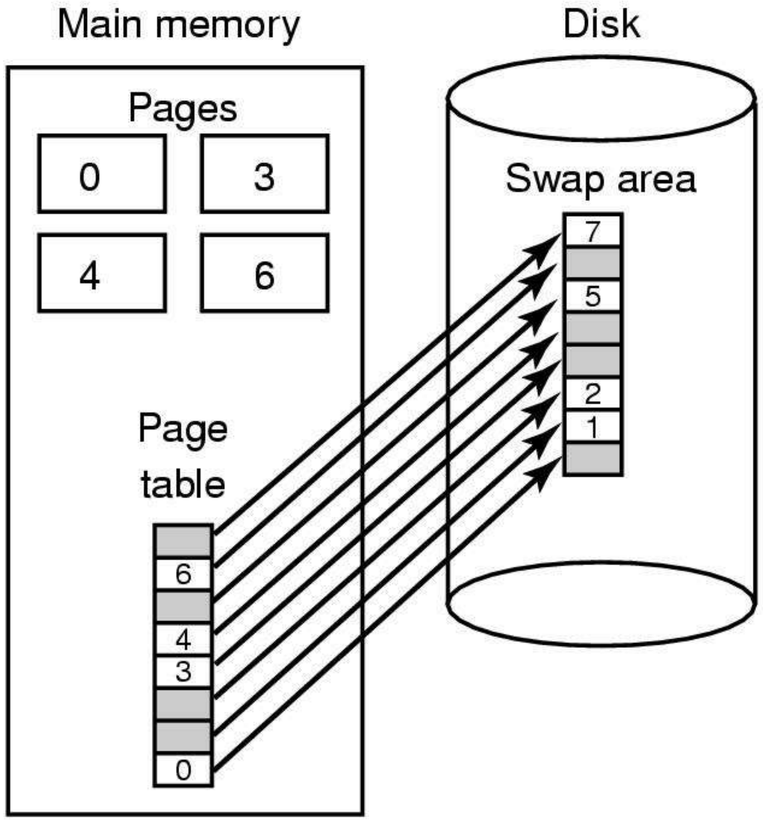
- Virtual memory and I/O interact, requiring *pinning* of pages
- *Example:*
  - *One process does a read system call*
    - This process suspends during I/O
  - *Another process runs*
    - It has a page fault
    - Some page is selected for eviction
    - The frame selected contains the page involved in the read
- *Solution:*
  - *Each frame has a flag: "Do not evict me".*
  - *Must always remember to un-pin the page!*

SWAP AREA

Managing the 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



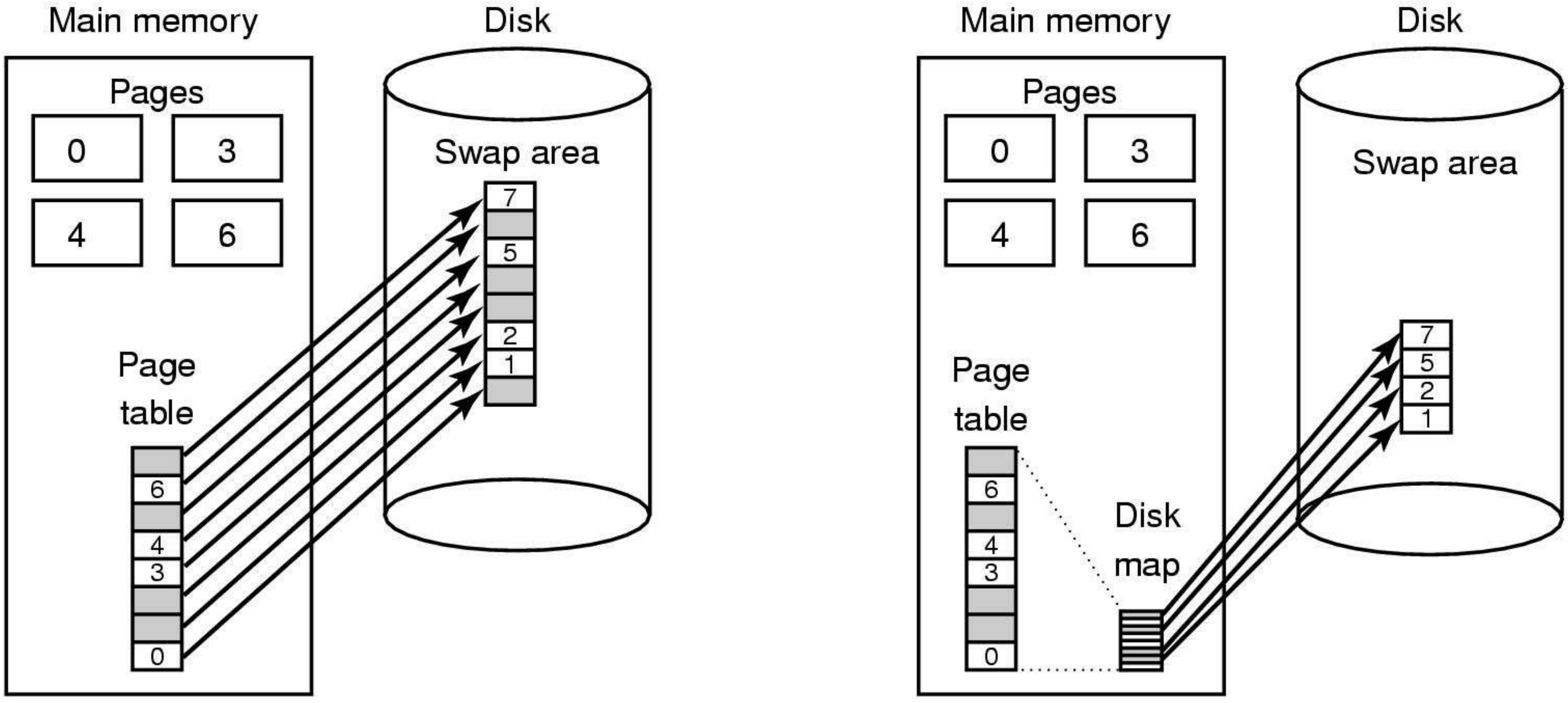
Approach #1

- What if more pages are allocated and the virtual address space grows during execution?

Approach #2

- ***Approach #2***
- Store the pages in the swap space 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



(a)

(b)

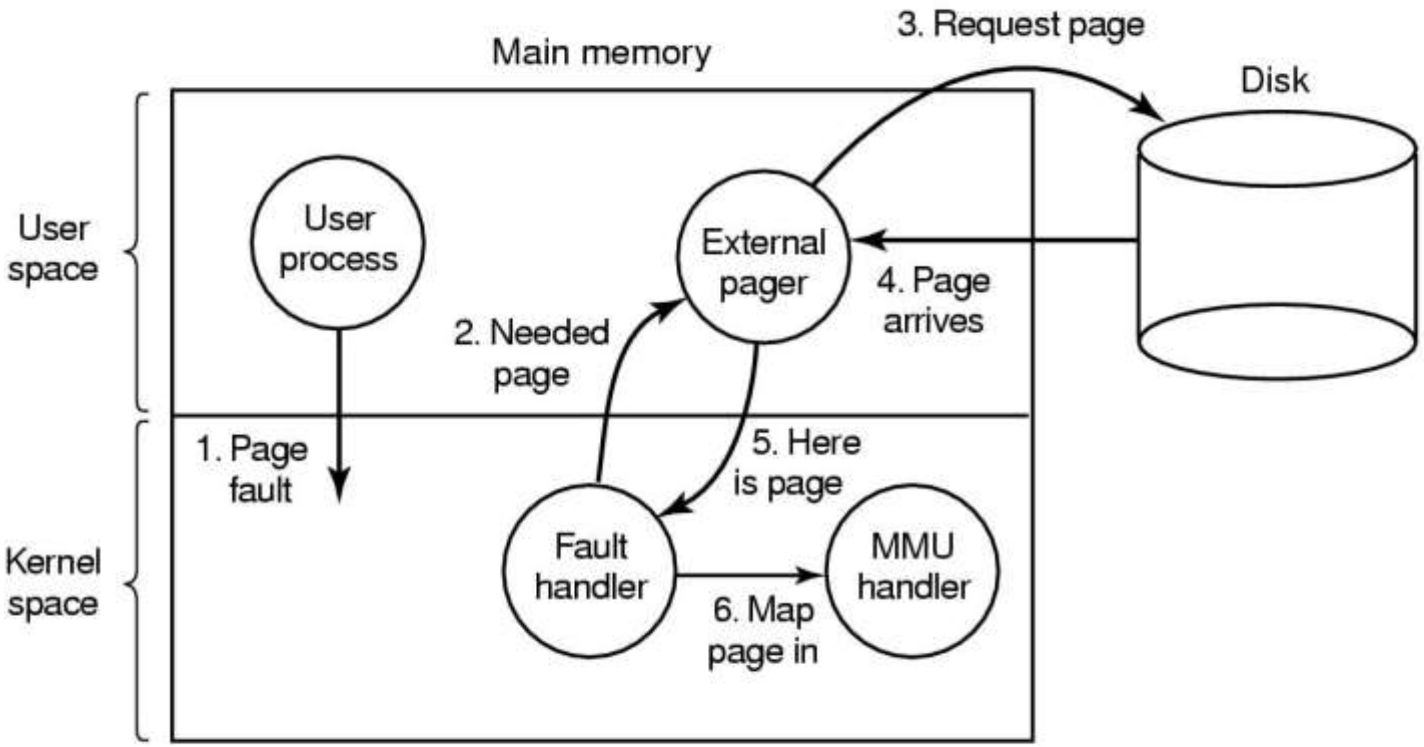
Approach #3

- Swap to a file
  - *Each process has its own swap file*
  - *File system manages disk layout of files*

Approach #4

- Swap to an external pager process (object)
- A user-level external pager determines policy
  - *Which page to evict*
  - *When to perform disk I/O*
  - *How to manage the swap file*
- When the OS needs to read in or write out a page it sends a message to the external pager
  - *Which may even reside on a different machine*

Approach #4



Main memory
3. Request page
Disk
User space
User process
External pager
4. Page arrives
2. Needed page
Kernel space
1. Page fault
5. Here is page
Fault handler
MMU handler
6. Map page in

Mechanism vs Policy

- Kernel contains
  - *Code to interact with the MMU*
    - This code tends to be *machine dependent*
    - *Mechanism*
  - *Code to handle page faults*
    - This code tends to be *machine independent* and may embody generic operating system policies
    - Policy

کارایی صفحەبندی

Paging Performance

- Paging works best if there are plenty of free frames
- If all pages are full of dirty pages…
  - *we must perform 2 disk operations for each page fault*
  - *This doubles page fault latency*
- It can be a good idea to periodically write out dirty pages in order to speed up page fault handling delay

Paging Daemon

- Paging daemon
  - *A kernel process*
  - *Wakes up periodically*
  - *Counts the number of free page frames*
  - *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 then this page is evicted*

جلسه‌ی بعد

Page replacement algorithms