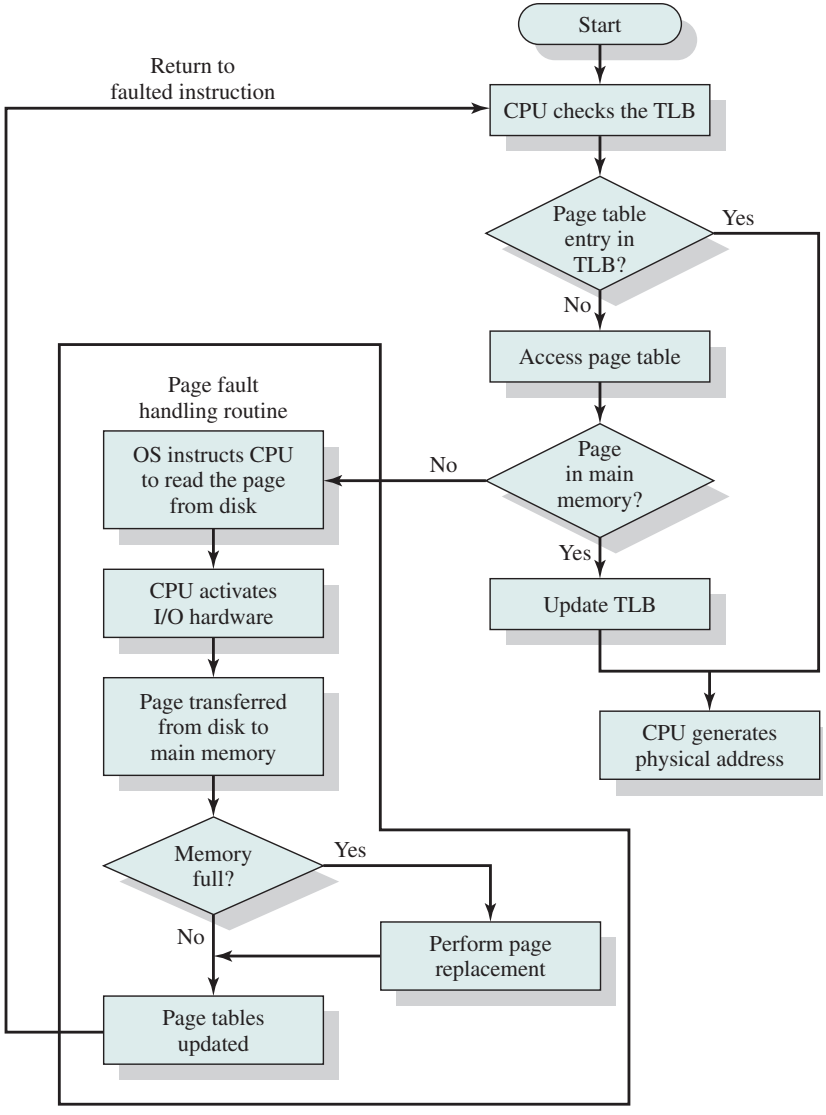


CSL 301

OPERATING SYSTEMS

Lecture 13

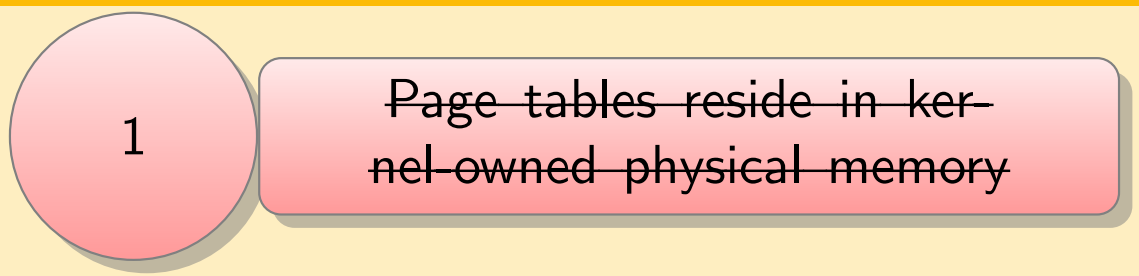
Beyond Physical Memory

Instructor
Dr. Dhiman Saha

Issues?

- Page tables are smaller
- But they still reside in main memory
- Recall our assumption

Relaxing Assumption



Attempt #8: Swapping Pages to Disk

Going Beyond the Physical Memory
Page Faulting Mechanism

Going Beyond Physical Memory

- What we have assumed that we intend to relax
  - Every address space of every running process fits into memory
- New scenario
  - Support many concurrently-running large address space

Crux — **How to go beyond physical memory?**

How can the OS make use of a **larger**, **slower** device to transparently provide the **illusion** of a large virtual address space?

- What does one mean by **larger** and **slower** device?
  - For now, just **assume** we have a big and relatively-slow device which we can use to help us build the illusion of a very large virtual memory, **even bigger than physical memory itself**.

Going Beyond Physical Memory

Why support using more memory than is physically available?

- Historical Significance: Multiprogramming
  - Idea of **memory-overlay**s
  - Manually move pieces of code or data in and out of memory as they were needed
- Convenience and ease of use and a better abstraction
  - Programmer does not have to worry about if there is room enough in memory for your program's data structures
  - Write code and leave rest on OS.

Swap-space

- Need to reserve some space on the disk for moving pages back and forth
- Swap pages out of memory to it and swap pages into memory from it.
- Assume that the OS can read from and write to the swap space, in page-sized units
- Note: OS will need to remember the **disk address** of a given page.

Question

What does the size of the swap space determine?

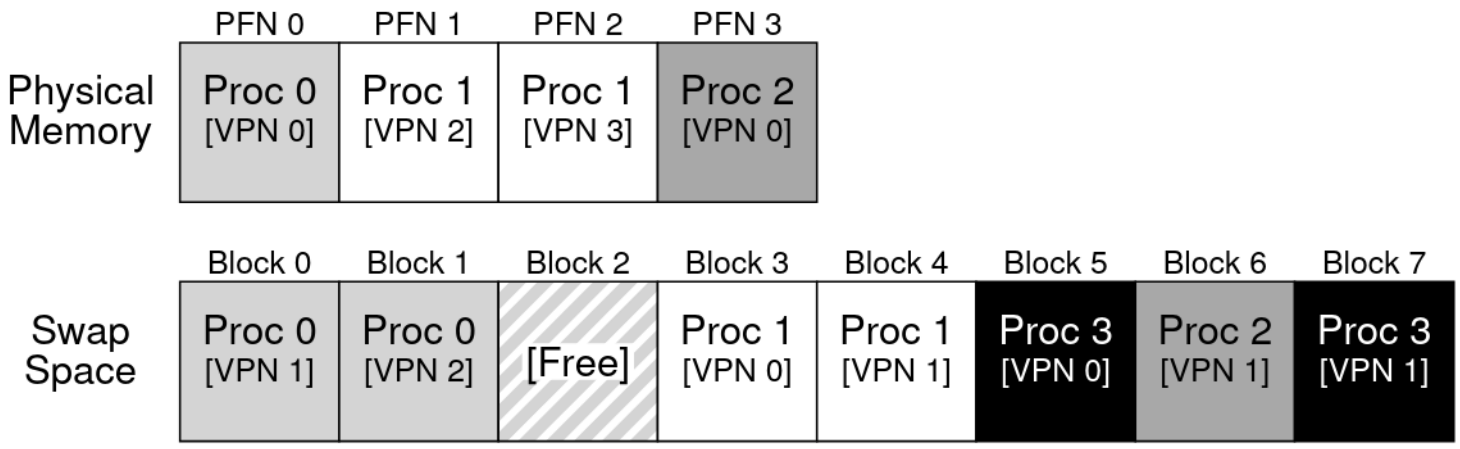
Swap-space Size?

- This can vary.
- For now assume it is very large.

HW

Determine the size of the swap-space your OS is currently using?

Example

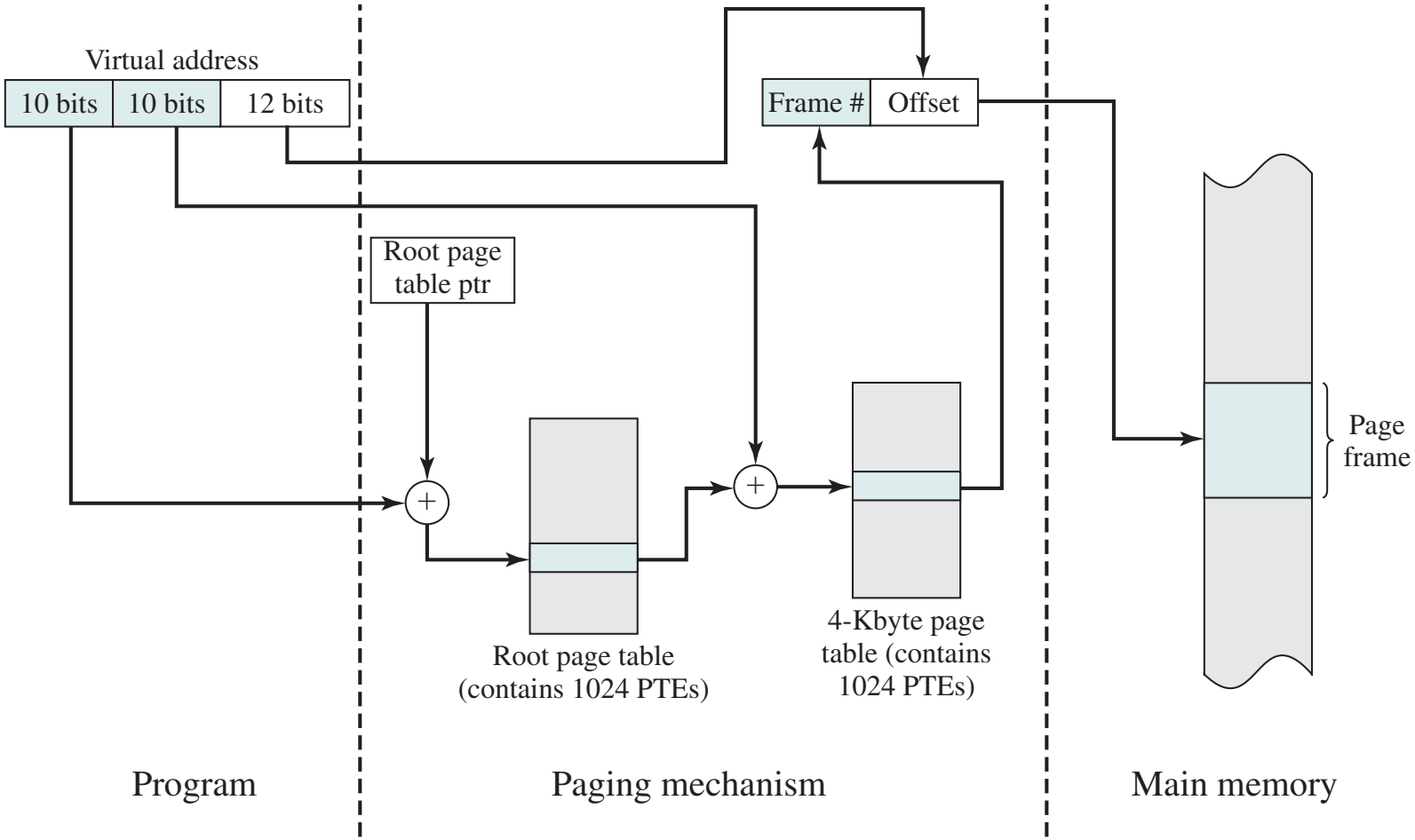


Point to ponder

Is swap-space the only on-disk location for swapping traffic?

- What happens when you run `ls`? or
- Your own compiled `main` program
- Hint:
  - Think about the `code` pages.

Recall: Address Translation (TLB Miss) Multi-level PT



Note Something changes with swap-space

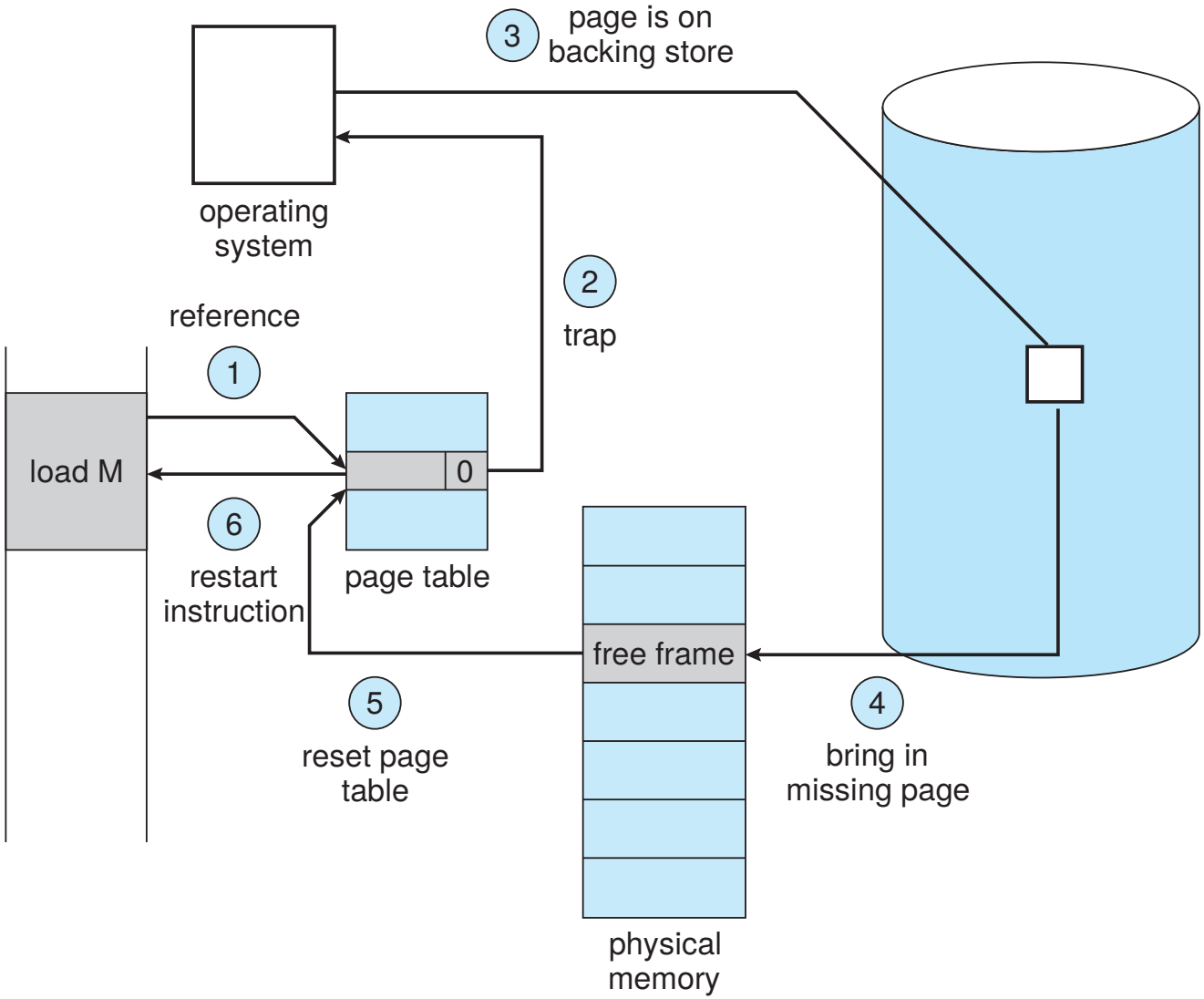
The page requested may **not** be in the memory at all! Then what?

If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is *not present* in physical memory. The way the hardware (or the OS, in a software-managed TLB approach) determines this is through a new piece of information in each page-table entry, known as the **present bit**. If the present bit is set to one, it means the page is present in physical memory and everything proceeds as above; if it is set to zero, the page is *not* in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

Who handles the page fault?

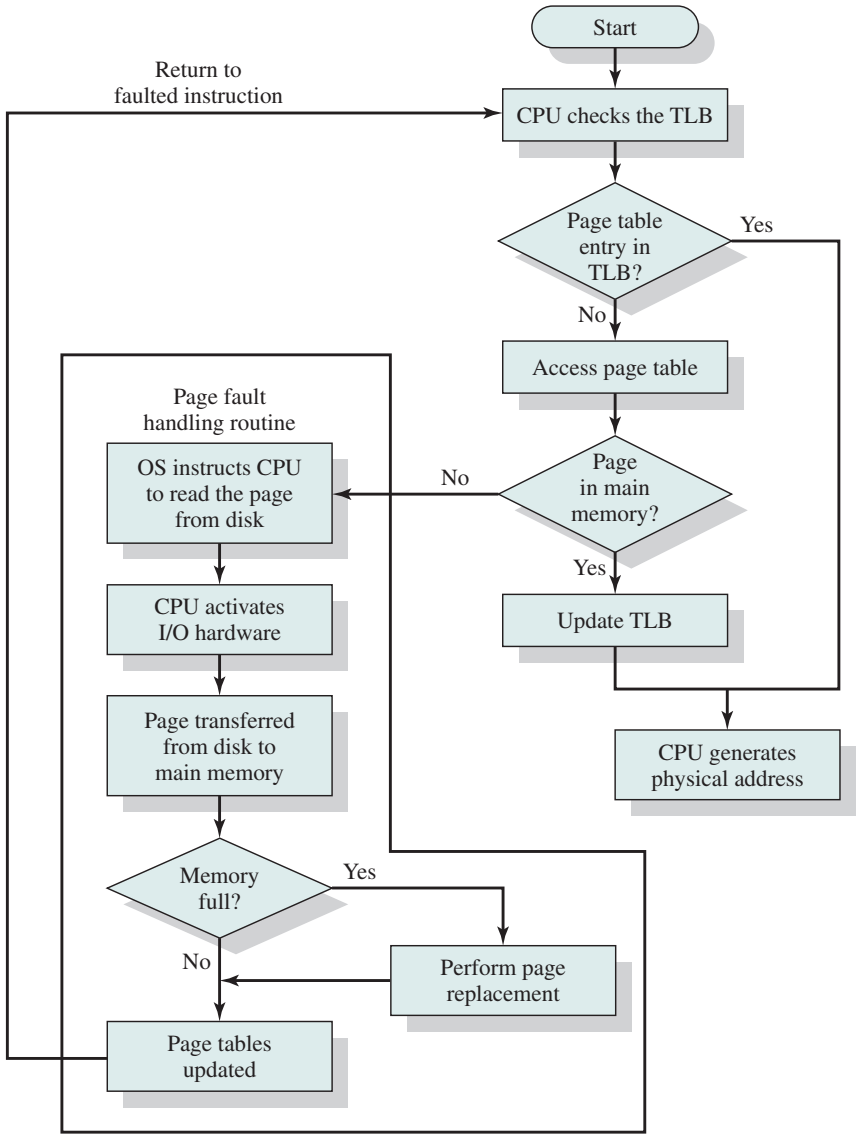
- The mechanism is same as exception handling.
- Using **page-fault handler**

Steps In Handling A Page Fault



Note that this figure does not include the TLB.

Page Fault Control Flow with TLB



Page-Fault Control Flow Algorithm

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)   // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGE_FAULT)
```

Page-Fault Control Flow Algorithm

```
1   PFN = FindFreePhysicalPage()
2   if (PFN == -1)                  // no free page found
3       PFN = EvictPage()           // run replacement algorithm
4   DiskRead(PTE.DiskAddr, PFN)     // sleep (waiting for I/O)
5   PTE.present = True              // update page table with present
6   PTE.PFN     = PFN               // bit and translation (PFN)
7   RetryInstruction()              // retry instruction
```

Where is the Page?

How will the OS know where to find the desired page?

- The page table is a natural place to store such information.
- OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address.
- When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory

What happens when page is being copied?

- Note that while the I/O is in flight, the process will be in the **blocked** state.
- OS will be free to run other ready processes while th epage fault is being serviced.
- Recall: I/O is expensive
- This **overlap** of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most **effective** use of its hardware.

When Page Replacements Really Occur?

Does the OS really wait until memory is entirely full?

- That sounds unrealistic.

- Idea: Keep small amounts of memory always free **pro-actively**
- Notion of **high watermark (HW)** and **low watermark (LW)**

How does this work?

If `#FreePages < LW`

- OS invokes the **swap daemon/page daemon**
- This daemon is a **background** thread that is responsible for freeing memory
- The thread evicts pages until `#FreePages > HW` pages

Summary

The notion of accessing **more memory than is physically present** within a system.

- Idea of **swap-space** / The Present bit / Page faults

Summary

The notion of accessing **more memory than is physically present** within a system.

- Idea of **swap-space** / The Present bit / Page faults

Point-to-ponder

- All actions happen **transparently** to the process.
- Process still has the **illusion** (not without a cost) of accessing own private, contiguous virtual memory.
- Behind the scenes pages are placed in
  - Arbitrary (non-contiguous) locations in physical memory
  - sometimes not even present in memory
  - Hence have to be fetched from disk