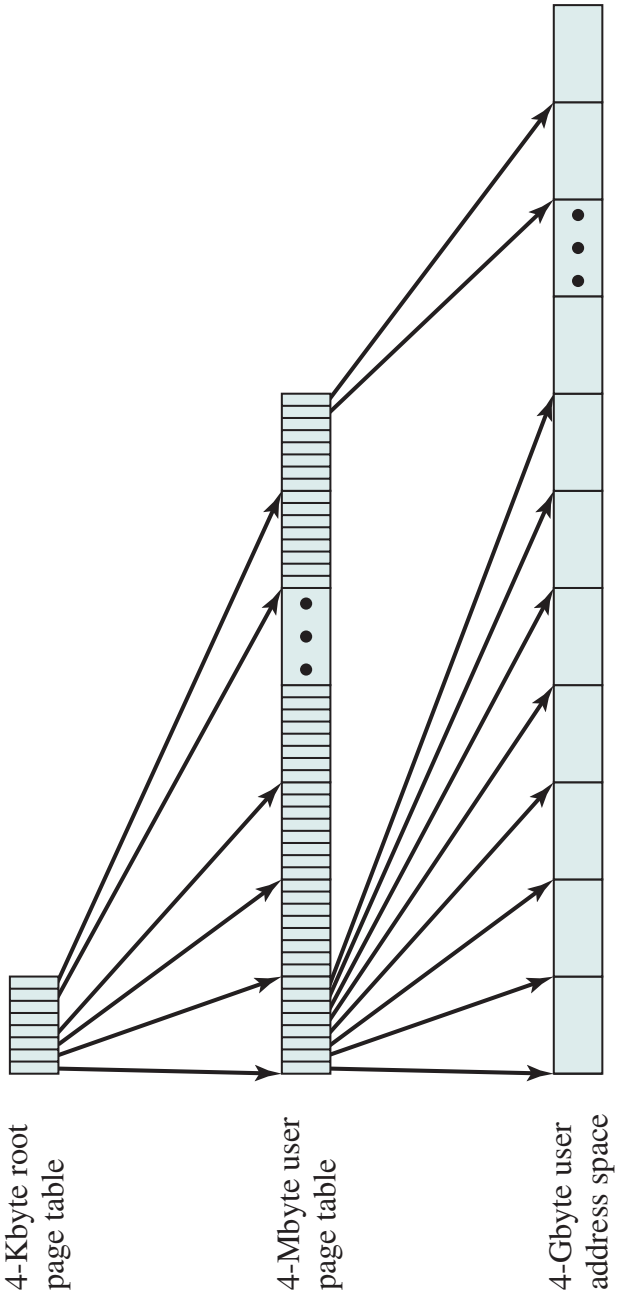


CS 250
OPERATING SYSTEMS

Lecture 10
Smaller Page Tables

Instructor
Dr. Dhiman Saha

Space overheads — Issue 1

Storing PT in memory wastes valuable memory space.

> Factor 2 slow down — Issue 2

For every memory reference, paging requires us to perform one extra memory reference in order to first fetch the translation from the page table

Space overheads — Issue 1

Storing PT in memory wastes valuable memory space.

> Factor 2 slow down — Issue 2

For every memory reference, paging requires us to perform one extra memory reference in order to first fetch the translation from the page table

Solves one issue with Paging

- Effective access time is decreased

However

- Issue of TLB Coverage

Does not solve

- Issue of storing large PT in memory

Issues?

Solves one issue with Paging

- Effective access time is decreased

However

- Issue of TLB Coverage

Does not solve

- Issue of storing large PT in memory

Solves one issue with Paging

- Effective access time is decreased

However

- Issue of TLB Coverage

Does not solve

- Issue of storing large PT in memory

Attempt #7: Smaller Page Tables

Do we have any assumption?

Assumption

1. Page tables reside in kernel-owned physical memory

Linear Page Tables

Simple array-based page tables are too big, taking up far too much memory on typical systems.

- Recall the space consumption for 32-bit address space with 4KB pages

- How can we make page tables smaller?
- What are the key ideas?
- What inefficiencies arise as a result of these new data structures?

Idea #1 Bigger Pages

Lets do the math

32-bit address space with **16KB** pages

- Whats the reduction?
- And whats the problem?

Problem

- Waste within each page
- **Internal Fragmentation**
- The waste **internal** to the unit of allocation

- Common page-sizes: 4KB (as in x86) or 8KB (as in SPARCv9)

Idea #1 Bigger Pages

Lets do the math

32-bit address space with **16KB** pages

- Whats the reduction?
- And whats the problem?

Problem

- Waste within each page
- **Internal Fragmentation**
- The waste **internal** to the unit of allocation

- Common page-sizes: 4KB (as in x86) or 8KB (as in SPARCv9)

Idea #1 Bigger Pages

Lets do the math

32-bit address space with **16KB** pages

- Whats the reduction?
- And whats the problem?

Problem

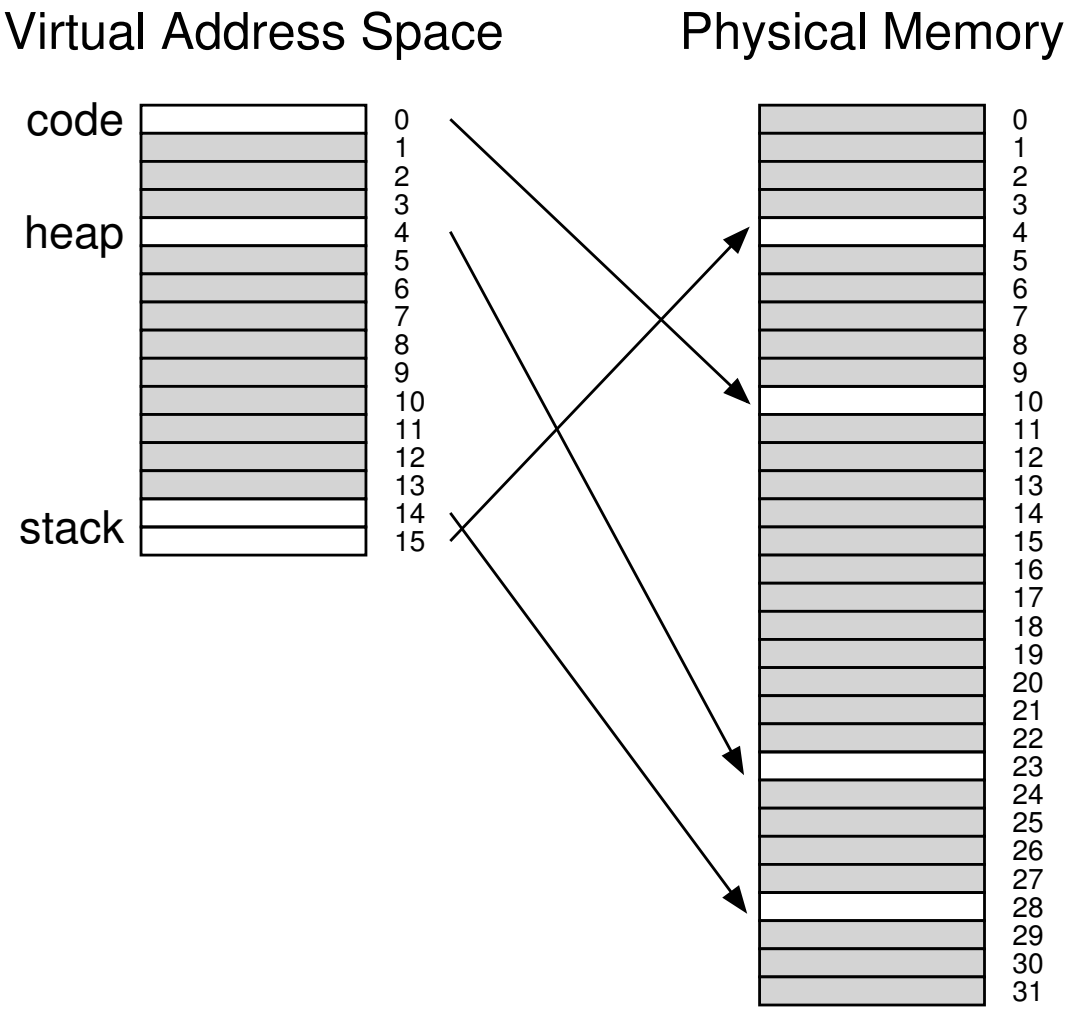
- Waste within each page
- **Internal Fragmentation**
- The waste **internal** to the unit of allocation

- Common page-sizes: 4KB (as in x86) or 8KB (as in SPARCv9)

Idea #2 Hybrid Approach: Paging and Segments

Lets get the intuition using an example

A 16KB Address Space With 1KB Pages



A Page Table For 16KB Address Space

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Note

Most of the page table is unused, full of **invalid** entries.

Why **single** page table for the entire address space of the process?

Combining paging and segmentation

- Different segments
- Different page table for each segment

- What hardware support is needed?
- What OS needs to do at context switch?

Why **single** page table for the entire address space of the process?

Combining paging and segmentation

- Different segments
- Different page table for each segment

- What hardware support is needed?
- What OS needs to do at context switch?

Why **single** page table for the entire address space of the process?

Combining paging and segmentation

- Different segments
- Different page table for each segment

- What hardware support is needed?
- What OS needs to do at context switch?

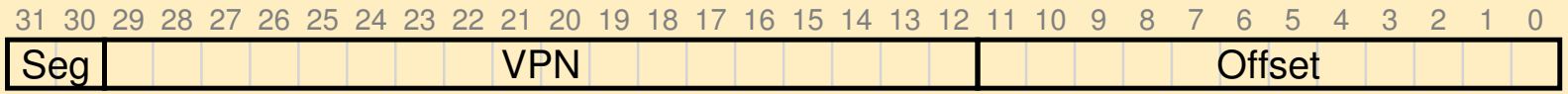
Why **single** page table for the entire address space of the process?

Combining paging and segmentation

- Different segments
- Different page table for each segment

- What hardware support is needed?
- What OS needs to do at context switch?

Address Translation



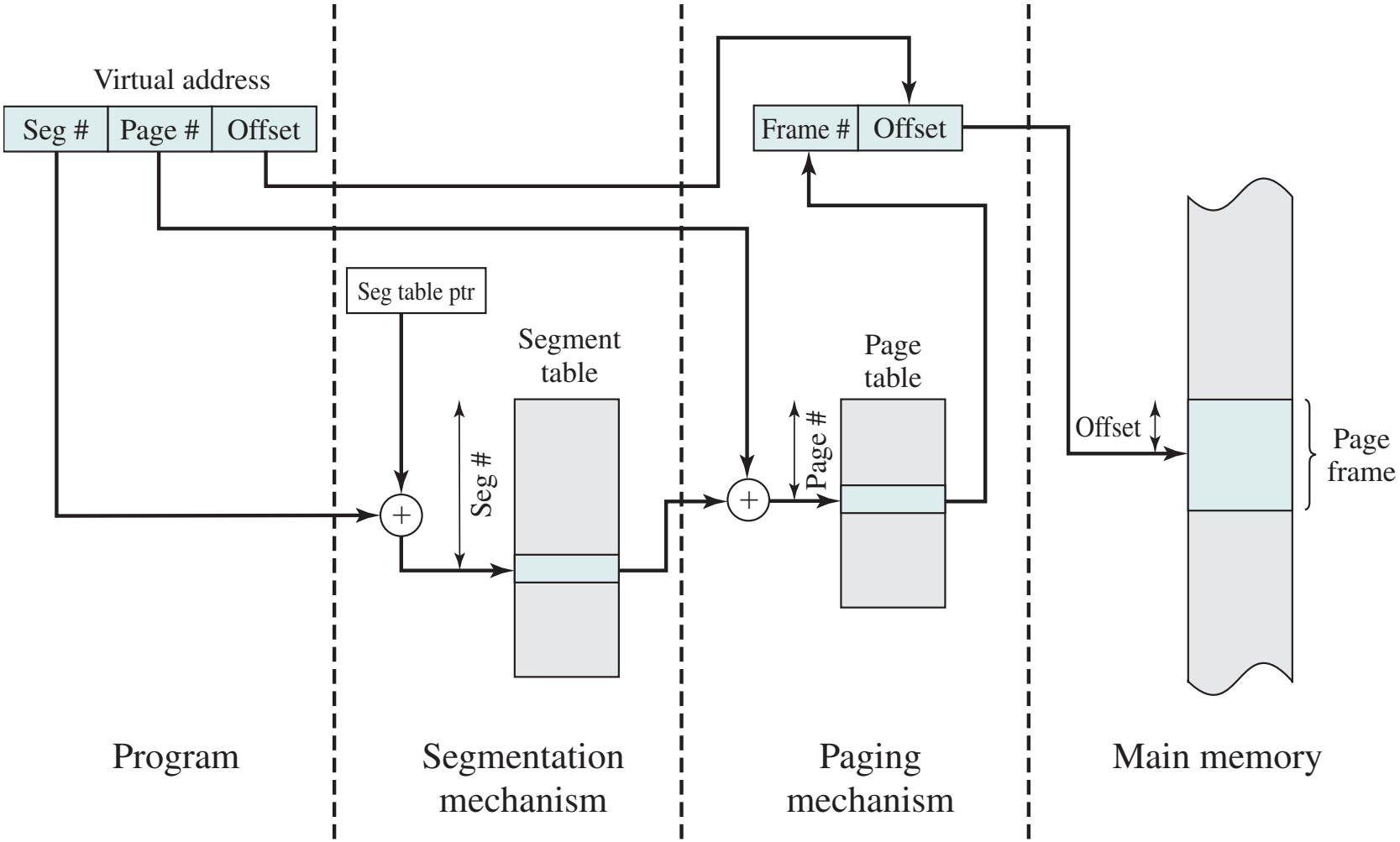

```
SN            = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN           = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Note

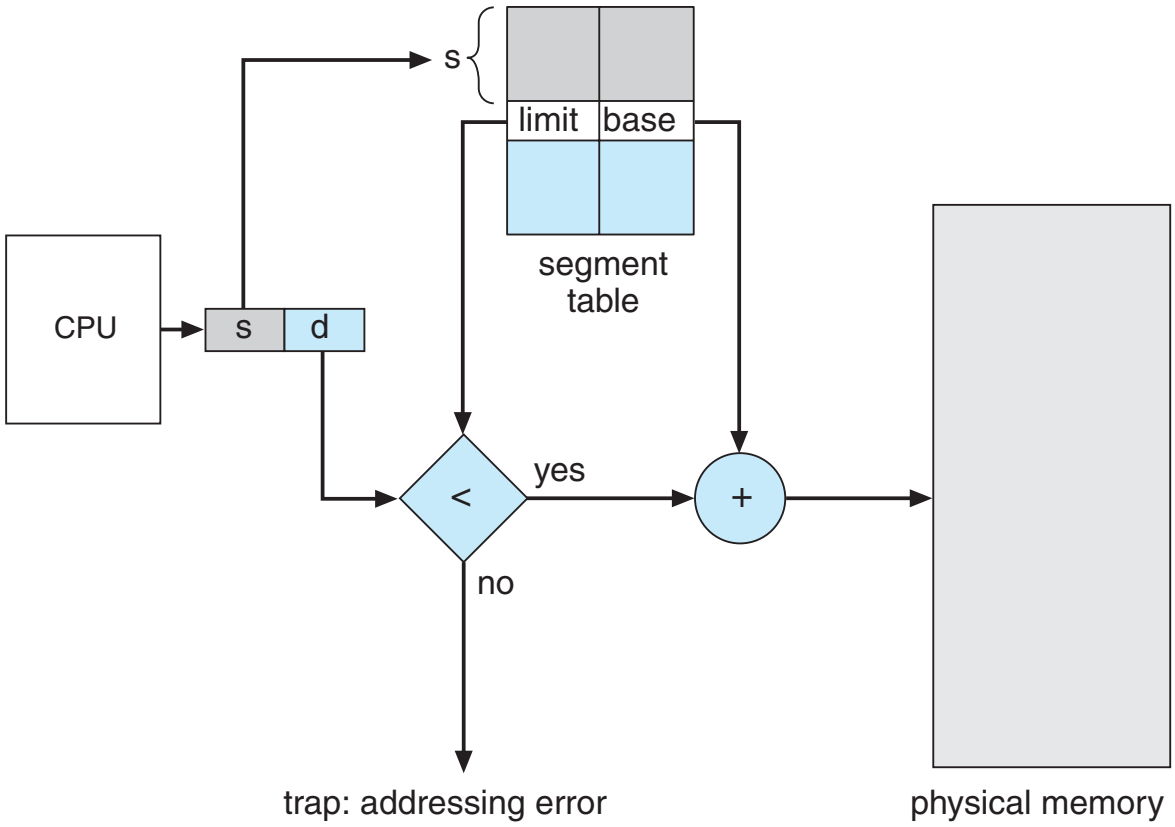
The use of one of three segment base registers instead of the single page table base register.

On a TLB Miss

Address Translation



Recall Bounds Register (Segmentation)



Will happen in hybrid approach too

Memory accesses beyond the end of the segment will generate an exception

Recall the issues with segmentation

- Address space usage
- External Fragmentation

Idea #3 Multi-level Page Tables

Aim

How to get rid of all invalid regions in the page table instead of keeping them all in memory?

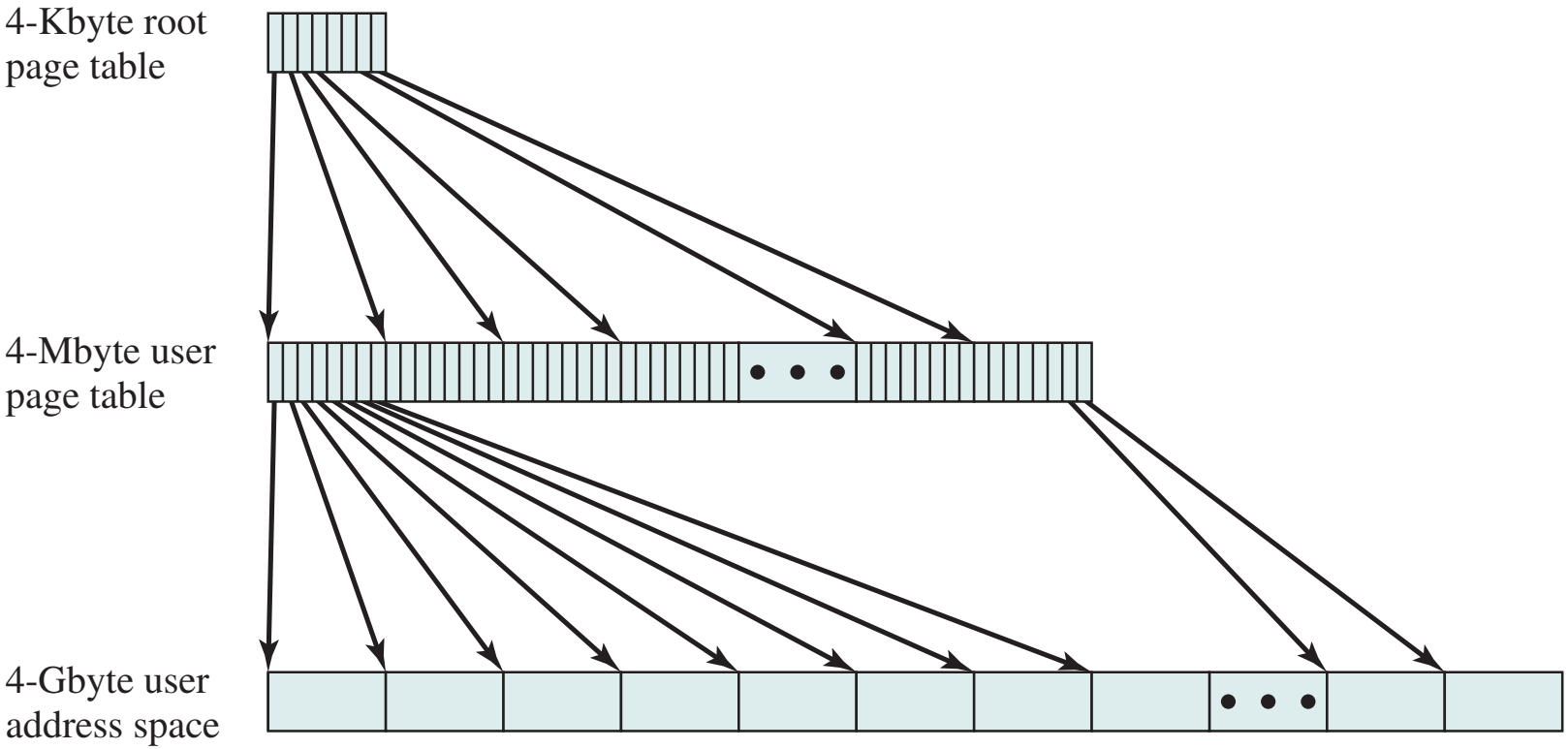
Use a hierarchical approach — Idea

- First, chop up the page table into page-sized units
- If an entire page of page-table entries (PTEs) is invalid, dont allocate that page of the page table at all

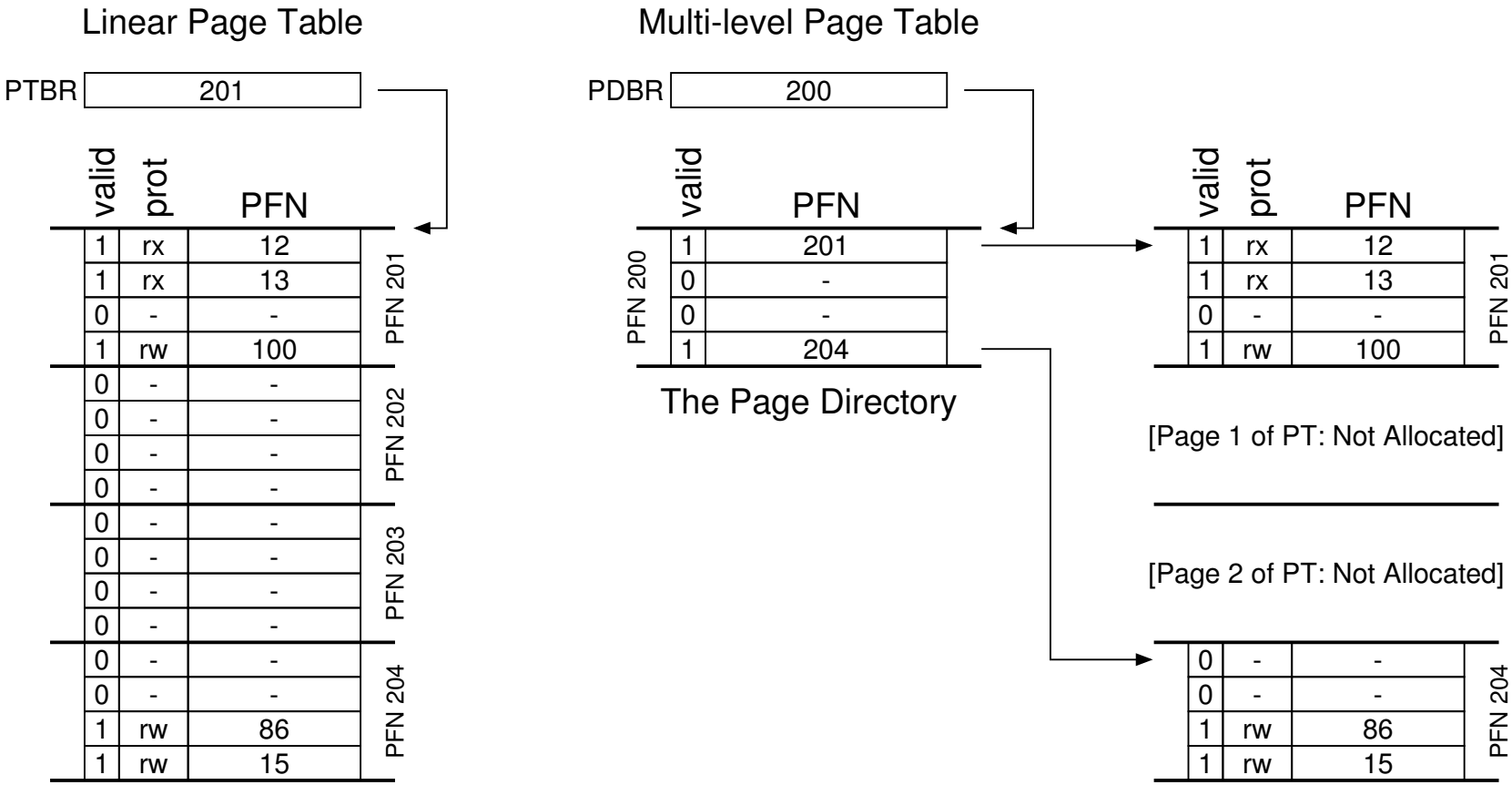
New structure — The page directory.

To track whether a page of the page table is valid (and if valid, where it is in memory)

Hierarchical Approach



Linear PT Vs Multi-level PT



Note the role of valid bit in PD

Advantages

- Incorporates address apace usage
- Easier memory allocation

Compare linear PT allocation

- Additional level of indirection through PD
- Allows placing **page-table pages** at different locations in physical memory.

Advantages

- Incorporates address apace usage
- Easier memory allocation

Compare linear PT allocation

- Additional level of indirection through PD
- Allows placing **page-table pages** at different locations in physical memory.

Advantages

- Incorporates address apace usage
- Easier memory allocation

Compare linear PT allocation

- Additional level of indirection through PD
- Allows placing **page-table pages** at different locations in physical memory.

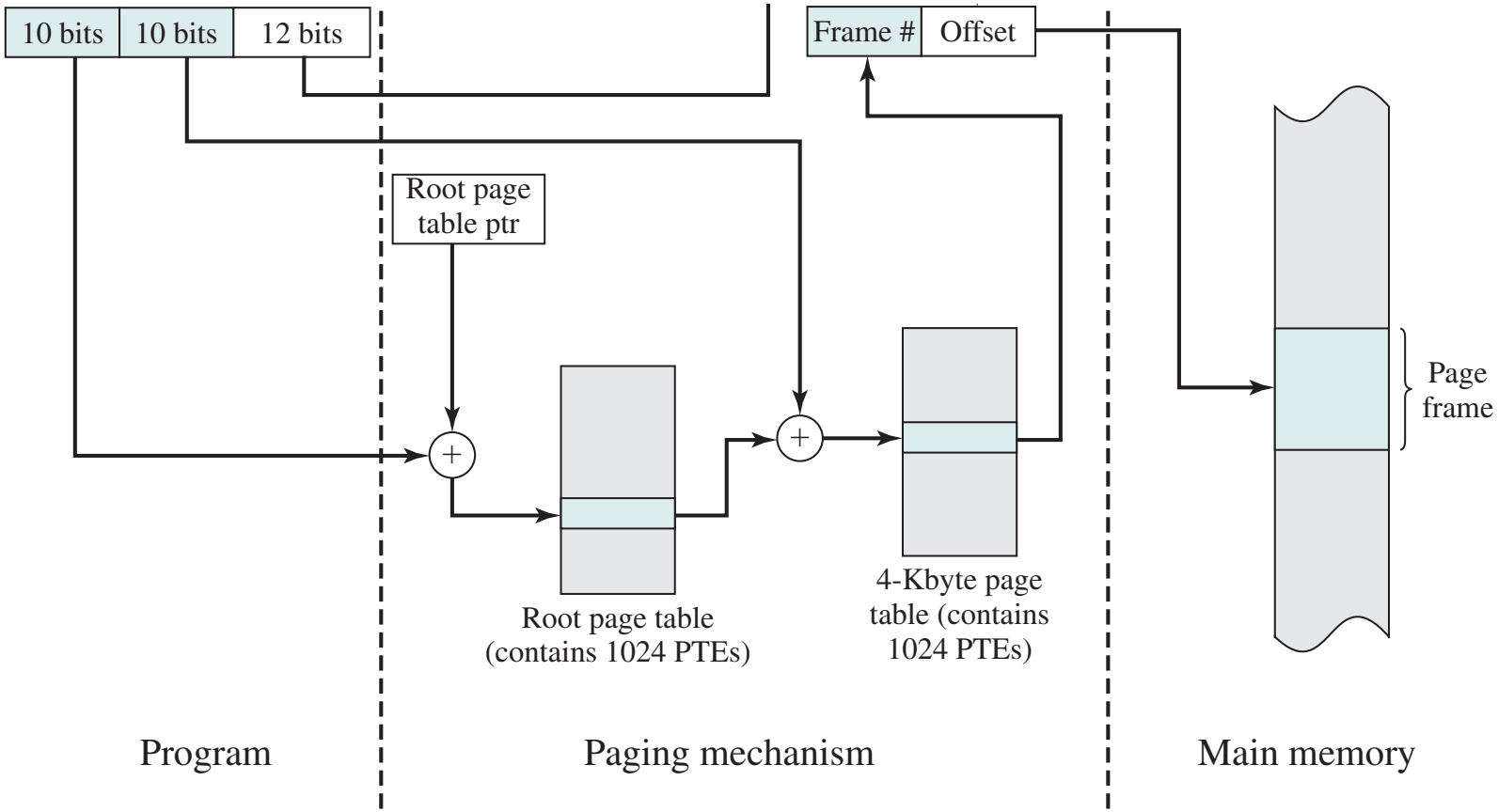
Advantages

- Incorporates address apace usage
- Easier memory allocation

Compare linear PT allocation

- Additional level of indirection through PD
- Allows placing **page-table pages** at different locations in physical memory.

Address Translation (TLB Miss)



Practice

Example from OSTEP

Note

The control flow in address translation on TLB Miss

- How many memory accesses we need to make?
- **Space-Time Trade Off**

- Complexity w.r.t. linear PT

H/W-managed TLB — Multi-level PT Control Flow

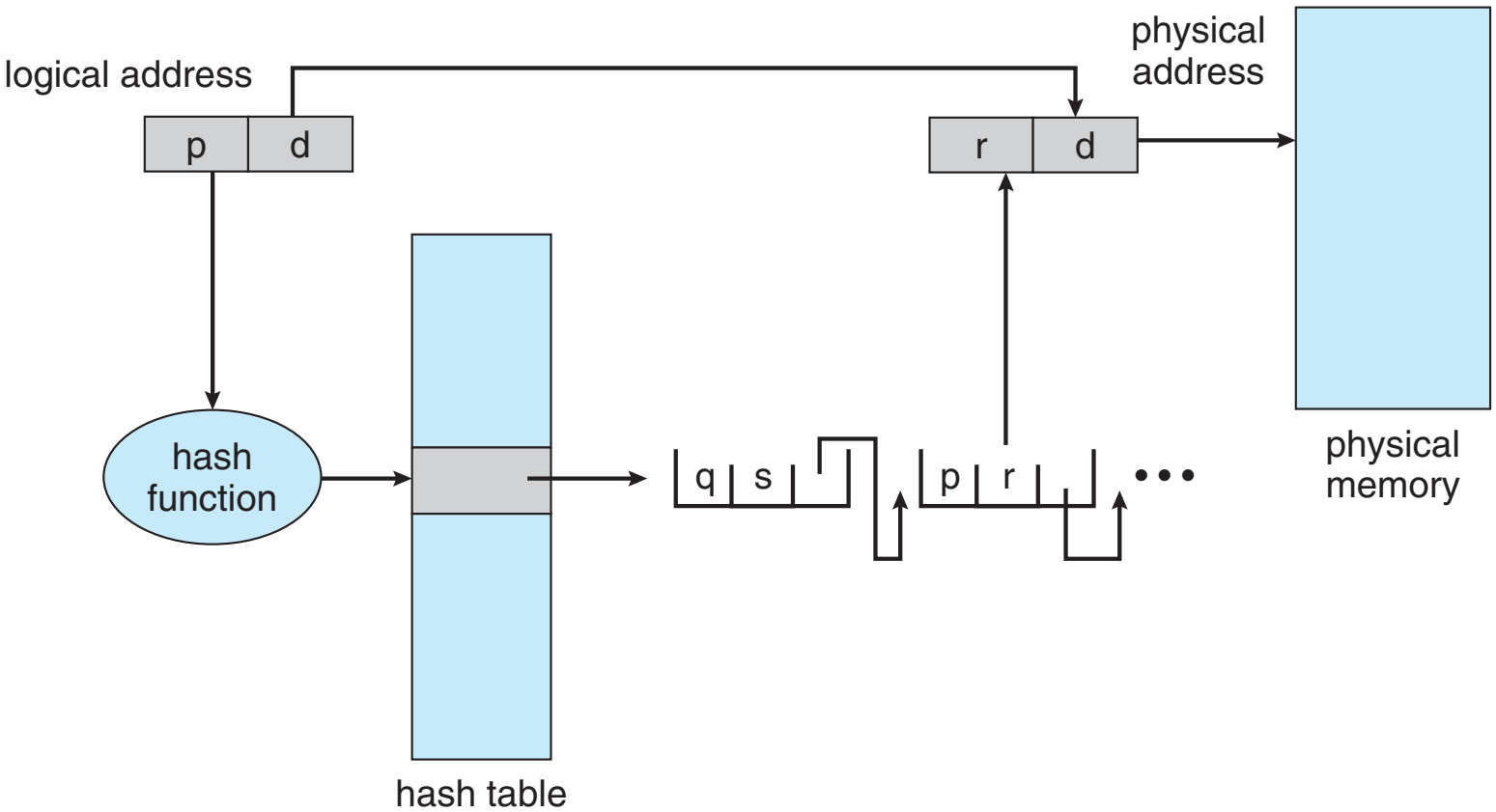
```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      // first, get page directory entry
12      PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13      PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14      PDE     = AccessMemory(PDEAddr)
15      if (PDE.Valid == False)
16          RaiseException(SEGMENTATION_FAULT)
17      else
18          // PDE is valid: now fetch PTE from page table
19          PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20          PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21          PTE     = AccessMemory(PTEAddr)
22          if (PTE.Valid == False)
23              RaiseException(SEGMENTATION_FAULT)
24          else if (CanAccess(PTE.ProtectBits) == False)
25              RaiseException(PROTECTION_FAULT)
```

What about more levels?

How does the complexity increase?

Idea #4

Hashed Page Tables



Idea #5 Inverted Page Tables

Idea

A **single page table** that has an entry for each physical page of the system.

- The entry tells us which process is using this page, and
- Which virtual page of that process maps to this physical page.

Point to Ponder

- How would you now find an entry?
- Will the PT technique work?
- Is there a speed issue?
- What is the scalability advantage of such a setting?

Idea #5 Inverted Page Tables

Idea

A **single page table** that has an entry for each physical page of the system.

- The entry tells us which process is using this page, and
- Which virtual page of that process maps to this physical page.

Point to Ponder

- How would you now find an entry?
- Will the PT technique work?
- Is there a speed issue?
- What is the scalability advantage of such a setting?

Idea #5 Inverted Page Tables

Idea

A **single page table** that has an entry for each physical page of the system.

- The entry tells us which process is using this page, and
- Which virtual page of that process maps to this physical page.

Point to Ponder

- How would you now find an entry?
- Will the PT technique work?
- Is there a speed issue?
- What is the scalability advantage of such a setting?

Idea #5 Inverted Page Tables

Idea

A **single page table** that has an entry for each physical page of the system.

- The entry tells us which process is using this page, and
- Which virtual page of that process maps to this physical page.

Point to Ponder

- How would you now find an entry?
- Will the PT technique work?
- Is there a speed issue?
- What is the scalability advantage of such a setting?

Idea #5 Inverted Page Tables

Idea

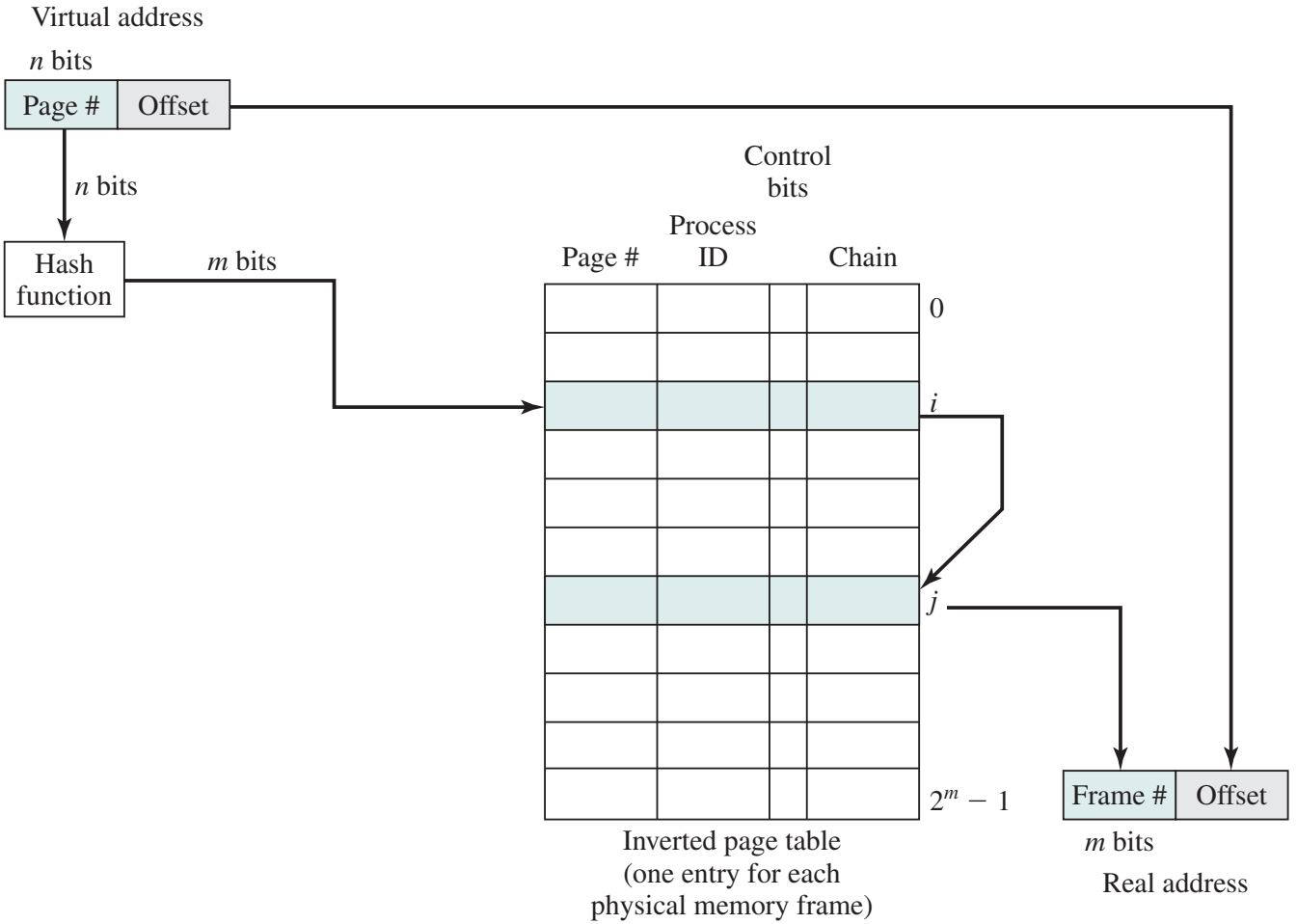
A **single page table** that has an entry for each physical page of the system.

- The entry tells us which process is using this page, and
- Which virtual page of that process maps to this physical page.

Point to Ponder

- How would you now find an entry?
- Will the PT technique work?
- Is there a speed issue?
- What is the scalability advantage of such a setting?

Inverted Page Tables



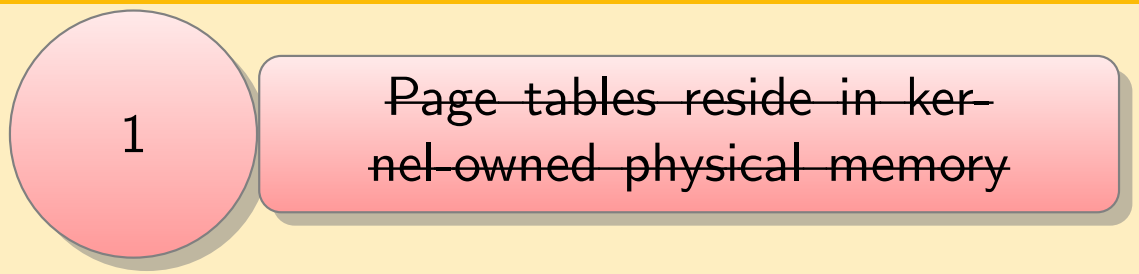
HW-4/5

How address translation takes place here?

Issues?

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

Relaxing Assumption



Attempt #8: Swapping the Page Tables to Disk

Page Faulting Mechanism

Page Fault Control Flow