5118020-03 Operating System

Paging: Smaller Tables

OSTEP Chapter 20

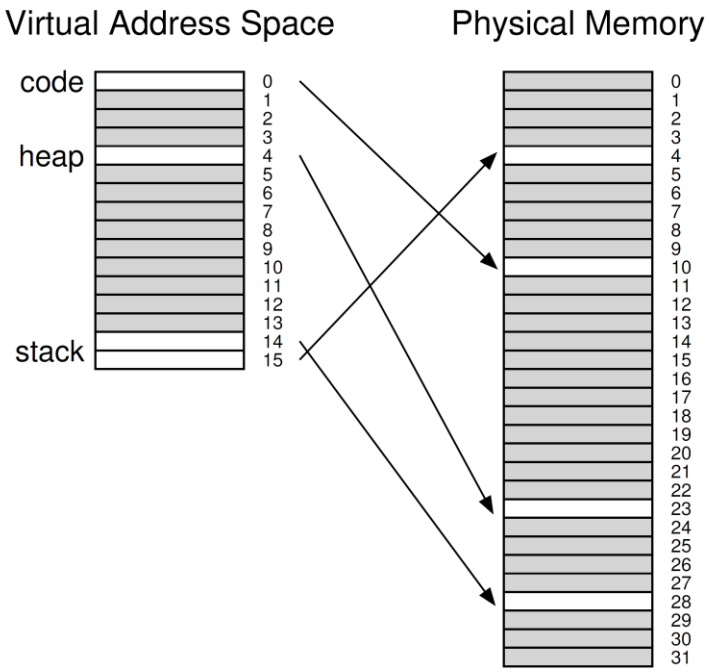
Shin Hong

Memory-efficient Page Table Structures

- Array-based page tables take up too much memory
  even though most page entries remain unused (i.e., invalid)
  - e.g., for example of a 32-bit address space with 4 KB pages,
    a per-process page table takes 4 MB

- Approaches
  1. use bigger pages
  2. per-segment page tables
  3. multi-level page tables
  4. inverted page table

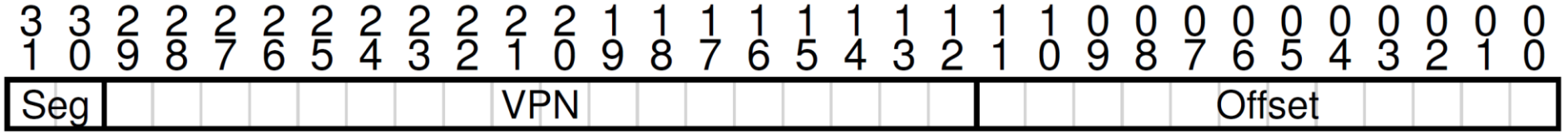
Per-segment Page Table



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	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

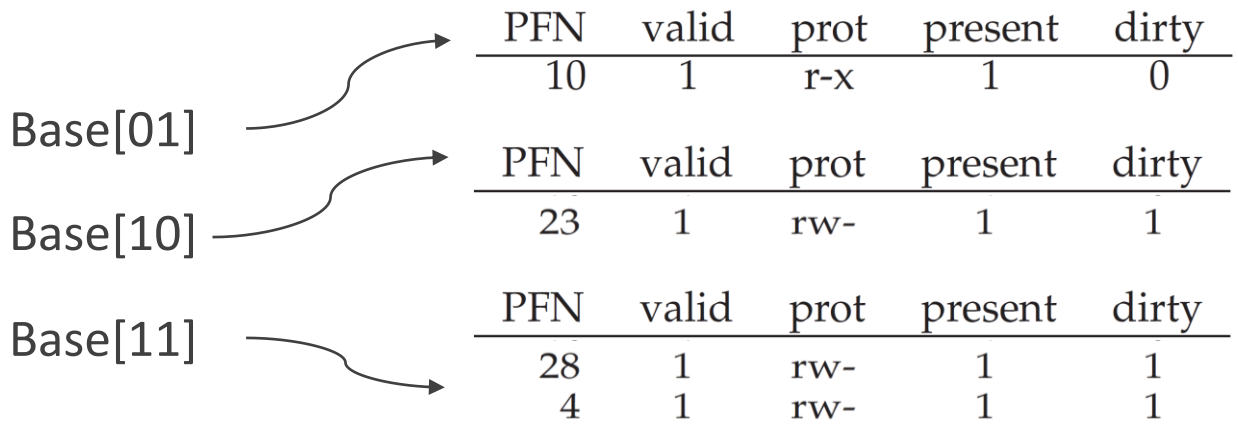
- Motivation: it is likely that only first few pages of each segment is used
- Allocate a variable-length a page table for each segment
  - re-use base-bound register pairs
  - a base register points to the beginning of a page table, and a bound register represents the number of allocated pages in the segment

Example



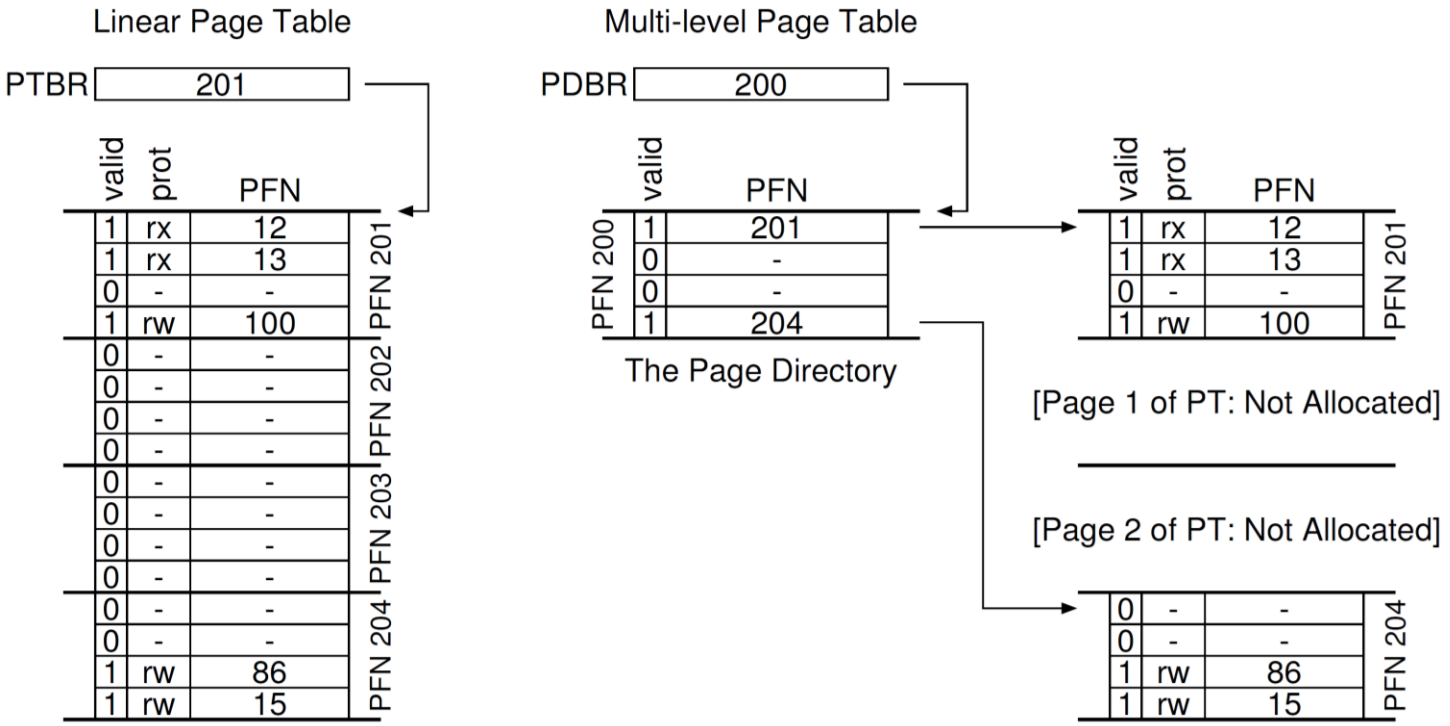
- use the two bits to represent a segment identifier
  - 01 for code, 10 for the heap, 11 for the stack
- access a segment page table at a TLB miss

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



Multi-level Page Tables

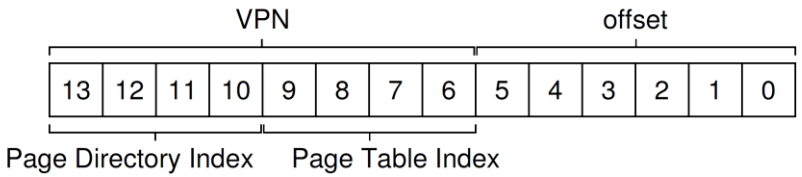
- Divide a page table into page-size units
  - A page table spans over multiple pages
  - Do not allocate a page if the corresponding piece of a page table has no valid entry
- Create a page directory as an index of the allocated pages for a page table
  - E.g.



Example

- Address size of 16 KB (2^{14}) with 64-byte pages
  - 256 VPNs
  - if a PTE is in 4 bytes, a page table takes 1 KB which is for 16 frames
- Suppose that only six VPNs, 0, 1, 4, 5, 254 and 255 are used, and the rest are unused

0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack



PDEAddr = PageDirBase + PDIndex * sizeof(PDE)

PTEAddr = (PDEAddr->PFN << SHIFT)
+ PTIndex * sizeof(PTE)

Page Directory

PFN	valid?
100	1
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
101	1

Page of PT (@PFN:100)

PFN	valid	prot
10	1	r-x
23	1	r-x
—	0	—
—	0	—
80	1	rw-
59	1	rw-
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—

Page of PT (@PFN:101)

PFN	valid	prot
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
55	1	rw-
45	1	rw-

Two-level Page Table 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)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

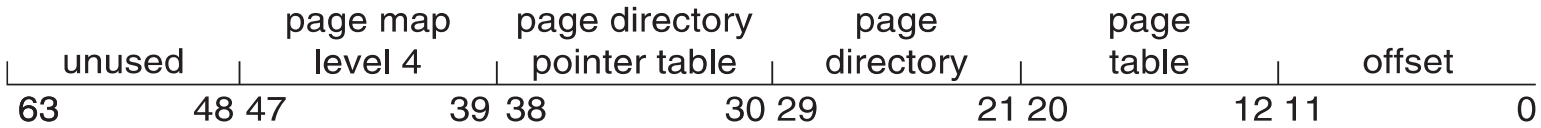
Inverted Page Table

- Keep a single page table for the entire system
  - each frame is mapped to a pair of a process ID and a VPN
  - each frame has a single entry

- Searching the entry for a VPN of a process in an inverted page table takes much more time than in a per-process page table
  - linear search, hashing, etc.

Intel x86-64

- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



ARM Architecture

- Modern, energy efficient, 32-bit CPU for mobile platform
- 4 KB and 16 KB pages
- 1 MB and 16 MB sections (large-size pages)
- two-level paging for pages &
  One-level paging for sections

- Two levels of TLBs
  - Inner is single main TLB
  - Outer level has two micro TLBs (one data, one instruction)
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU