Page Tables

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
(Typical) Linear Page Table

Virtual address space

- code
- data
- heap
- heap
- stack

Physical memory

Huge waste!
The Problem

• Space overhead of page tables
  – A 32-bit address space with 4KB pages (4 bytes/PTE):
    \(2^{20} \times 4 = 4\text{MB} \) (per process)
  – A 64-bit address space with 8KB pages (8 bytes/PTE):
    \(2^{51} \times 8 = 2^{54} = 32\text{PB} \) (per process)

• How can we reduce this overhead?
  – Observation: many invalid PTEs
  – Only need to map the portion of the address space actually being used which is a tiny fraction of entire address space
Paging with Segmentation (1)

• A segment represents a region of valid address space
  – Segmentation:
    • Divide virtual address space into segments
    • Each segment can have variable length
  – Paging:
    • Divide each segment into fixed-sized pages
    • Each segment has a page table
    • Each segment tracks base (physical address) and limit of the page table for that segment

• Virtual address divided into three portions
Paging with Segmentation (2)

- Example: Multics address translation
Paging with Segmentation (3)

• Pros
  – Can decrease the size of page tables
  – Segments can grow without any reshuffling
  – Can run process when some pages are swapped to disk
  – Increases flexibility of sharing: share either single page or entire segment

• Cons
  – Page tables potentially can be large
    • e.g. A large but sparse-used heaps will have a lot of waste
  – External fragmentation due to page tables
    • Each page table should be allocated contiguously
Multi-level Page Tables (1)

Linear Page Table

<table>
<thead>
<tr>
<th>valid</th>
<th>prot</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rx</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>rx</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>rw</td>
<td>100</td>
</tr>
</tbody>
</table>

Multi-level Page Table

<table>
<thead>
<tr>
<th>valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>201</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>204</td>
</tr>
</tbody>
</table>

The Page Directory

<table>
<thead>
<tr>
<th>valid</th>
<th>prot</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rx</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>rx</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>rw</td>
<td>100</td>
</tr>
</tbody>
</table>

[Page 1 of PT: Not Allocated]

[Page 2 of PT: Not Allocated]
Multi-level Page Tables (2)

- Allow each page table to be allocated non-contiguously
- Virtual addresses have 3 parts

<table>
<thead>
<tr>
<th>Outer page #</th>
<th>Secondary page #</th>
<th>Page offset</th>
</tr>
</thead>
</table>

- Outer page table: outer page number $\rightarrow$ secondary page table
- Secondary page table: secondary page # $\rightarrow$ page frame #
Multi-level Page Tables (3)

- Example
  - 32-bit address space, 4KB pages, 4 bytes/PTE
  - Want every page table fit into a page

![Diagram of multi-level page tables]

- Outer page table
- Secondary page table
- Physical memory
- Physical address
Multi-level Page Tables (4)

• Address translation in Alpha AXP architecture
  – Three-level page tables
  – 64-bit address space divided into 3 segments (coded in bits 63/62)
    • seg0 (0x): user code
    • seg1 (11): user stack
    • kseg (10): kernel
  – Alpha 21064
    • Page size: 8KB
    • Virtual address: 43 bits
    • Each page table is one page long
Multi-level Page Tables (5)

- Address translation in Intel 64 architecture
  - 48-bit “linear” address $\rightarrow$ 52-bit physical address (4KB page)

![Diagram of multi-level page tables]

- Linear Address
  - PML4
  - Directory Pointer
  - Directory
  - Table
  - Offset
  - Page-Directory-Pointer Table
  - PDPE
  - PDE with PS=0
  - Page-Directory
  - Page Table
  - Physical Addr
  - 4-KByte Page
  - CR3
Multi-level Page Tables (6)

• Pros
  – Compact while supporting sparse-address space
    • Page-table space in proportion to the amount of address space used
  – Easier to manage physical memory
    • Each page table usually fits within a page
  – Easier for hardware to walk though page tables
  – No external fragmentation

• Cons
  – More memory accesses on a TLB miss
  – More complex than a simple linear page-table lookup
Inverted Page Tables (1)

- Reverse mapping from PFN → <VPN, PID>
  - One entry for each page frame in physical memory
  - Entry consists of the virtual page number with information about the process that owns that page
  - Need to search through the table to find match
  - Use hashing to limit the search to one, or at most a few, page-table entries

- Pros & Cons
  - Decrease memory needed to store page tables: No need to have per-process page tables
  - Increase time needed to search the table on a TLB miss
Inverted Page Tables (2)
Paging Page Tables

• Store page tables in virtual address space
  – Cold (unused) page table pages can be paged out to disk
  – But, now addressing page tables requires translation
  – Outer page table is usually pinned into physical memory
  – Outer page table points to the virtual addresses (in the kernel address space) of secondary page tables
  – Need to handle nested page faults

• What if we page kernel code and data too?
(cf.) IA-32 VM Architecture

[Diagram of IA-32 VM Architecture]

- Logical Address (or Far Pointer)
  - Segment Selector
  - Offset
- Global Descriptor Table (GDT)
  - Segment Descriptor
  - Segment Base Address
- Linear Address Space
  - Segment
  - Lin. Addr.
  - Page Directory
    - Entry
  - Page Table
    - Entry
  - Page
- Physical Address Space
  - Phy. Addr.
- Segmentation
- Paging

(optional)
xv6: Page Table

![Page Table Diagram]

- **Virtual address**
  - Dir, Table, Offset

- **Physical Address**
  - PPN, Offset

- **Page Directory**
  - PPN, Flags
  - CR3

- **Page Table**
  - PPN, Flags

**Physical Page Number**

- AVL
- DA
- CW
- DT
- UWP

Page table and page directory entries are identical except for the D bit.

- **Legend:**
  - P - Present
  - W - Writable
  - U - User
  - WT - 1=Write-through, 0=Write-back
  - CD - Cache Disabled
  - A - Accessed
  - D - Dirty (0 in page directory)
  - AVL - Available for system use
xv6: Virtual Address Space
```c
2034 pde_t*
2035 copyuvm(pde_t *pgdir, uint sz)
2036 {
2037   pde_t *d;
2038   pte_t *pte;
2039   uint pa, i, flags;
2040   char *mem;
2041
2042   if((d = setupkvm()) == 0)
2043       return 0;
2044   for(i = 0; i < sz; i += PGSIZE){
2045     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
2046       panic("copyuvm: pte should exist");
2047     if(!(pte & PTE_P))
2048       panic("copyuvm: page not present");
2049     pa = PTE_ADDR(*pte);
2050   flags = PTE_FLAGS(*pte);
2051   if((mem = kalloc()) == 0)
2052       goto bad;
2053   memmove(mem, (char*)P2V(pa), PGSIZE);
2054   if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
2055     kfree(mem);
2056     goto bad;
2057   }
2058 }
2059   return d;
2060
2061 bad:
2062   freevm(d);
2063   return 0;
2064 }
```
1734 static pte_t *
1735 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737   pde_t *pde;
1738   pte_t *pgtab;
1739   pde = &pgdir[PDX(va)];
1740   if(*pde & PTE_P){
1741     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1742   } else {
1743     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1744       return 0;
1745     // Make sure all those PTE_P bits are zero.
1746     memset(pgtab, 0, PGSIZE);
1747     // The permissions here are overly generous, but they can
1748     // be further restricted by the permissions in the page table
1749     // entries, if necessary.
1750     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1751   }
1752   return &pgtab[PTX(va)];
1753 }
1754
static int mapprocess(pde_t *pgdir, void *va, uint size, uint pa, int perm) {
    char *a, *last;
    pte_t *pte;

    a = (char*)PGROUNDOWN((uint)va);
    last = (char*)PGROUNDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgd, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
xv6: Page Table Setup (cont’d)

• Question
  – Find and explain the following macros
    • PTE_ADDR()
    • PTE_FLAGS()
    • PDX()
    • PTX()
    • P2V()
    • V2P()
    • PTE_P, PTE_W, PTE_U