Operating Systems
ECE344

Virtual Memory Hardware

Ashvin Goel
ECE
University of Toronto
Outline

- Introduction to virtual memory
- Paging MMU
- Page tables
Problems with Simple Memory Management

- A simple memory management scheme allocates contiguous physical memory to programs

- Problems
  - Growing a program requires copying entire program
  - Wastes memory due to internal and external fragmentation
  - Running a program requires loading entire program in memory
  - Maximum program size is limited by memory size

- These problems occur because programs are allocated contiguous memory
Non-Contiguous Memory Allocation

- Can we use non-contiguous memory for programs?
  - It is hard to program with non-contiguous memory (why?)
  - Challenge is to hide non-contiguous memory from programs
Use Memory Management H/W

- Insight: Take advantage of MMU h/w to translate CPU virtual addresses to physical memory addresses
- Program can use contiguous virtual addresses
- Physical memory is allocated non-contiguously

1. Programs use **virtual** memory addresses
2. CPU sends these virtual addresses to MMU
3. MMU translates virtual address to **physical** memory address
4. MMU accesses memory using physical addresses
Example: MMU with 2 Base Registers

- Consider an MMU with **two** base and limit registers

- **Glossary**
  - $V =$ virtual address
  - $P =$ physical address
  - $B_1 =$ base 1, $L_1 =$ limit 1
  - $B_2 =$ base 2, $L_2 =$ limit 2

- **Virtual address space**
  - $0 \leq V < (L_1 + L_2)$

- **Address translation**
  - $P = V + B_1$, when $0 \leq V < L_1$
  - $P = (V - L_1) + B_2$, when $L_1 \leq V < (L_1 + L_2)$
Paging MMU Idea

- Problems with two base-register MMU
  - Limited: supports only two physical memory regions
  - Slow: with N registers, address translation needs $O(N)$ time

- A paging MMU supports a very large number of non-contiguous physical memory regions efficiently

- Key paging MMU ideas
  - Use fixed size virtual memory regions called pages
  - Map page to physical memory efficiently by keeping the mapping information in an array or tree structure
Paging MMU

- Virtual address space consists of contiguous, fixed size chunks called **pages**
  - Page size $= 2^n$ bytes, fixed by paging MMU, e.g., 4KB

- Paging MMU maps each page to a physical memory region called a **frame** that is the **same size** as a page
  - frame = map(page)  // map function is an array/tree lookup
Virtual and Physical Address

- Virtual address = (page number, offset)
  - On 32 bit machine, virtual address space size = $2^{32}$ bytes = 4GB
  - page size = $2^{12} = 4$KB, nr. of pages = $2^{20}$

- Physical address = (frame number, offset)
  - Say, physical memory size = $2^{30}$ bytes = 1GB
  - frame size = $2^{12} = 4$KB, nr. of frames = $2^{18}$
On each memory reference:
- CPU produces \( \text{vaddr} = (\text{page nr}, \text{offset}) \)
- Memory sees \( \text{paddr} = (\text{frame nr}, \text{offset}) \)
- MMU maps \( \text{vaddr} \) to \( \text{paddr} \), i.e., \( \text{frame} = \text{map}(\text{page}) \)
  - Note that offset remains the same
Paging Example

Virtual address space

MMU translates virtual to physical address

Physical address space

Page 2^20 - 1
Unallocated pages

Frame 2^18 - 1
Unused frame
Frame for another address space
Frame 1
Frame 0

MMU translates virtual to physical address
## Benefits of Paging

<table>
<thead>
<tr>
<th>Contiguous Memory Allocation</th>
<th>Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growing a program requires copying entire program</td>
<td>Growing a program requires allocating a page at a time</td>
</tr>
<tr>
<td>Wastes memory due to internal and external fragmentation</td>
<td>No external fragmentation, internal fragmentation is $\frac{1}{2}$ page per region</td>
</tr>
<tr>
<td>Running a program requires loading entire program in memory</td>
<td>As program runs, pages can be loaded in memory (we will see this later)</td>
</tr>
<tr>
<td>Maximum program size is limited by memory size</td>
<td>Maximum program size is limited by disk size (we will see this later)</td>
</tr>
</tbody>
</table>
Page Tables

- Paging MMU maintains its mapping information in a page table

- A page table contains page table entries (PTE) that map a page to a frame
  - Typically, each entry is one word
    - E.g, 32 bits on 32-bit architecture, 64 bits on 64-bit architecture
  - Each entry contains frame number and various bits such as valid/invalid, writeable, dirty, etc.
    - Valid bit says whether mapping is valid or not
    - We will discuss how these bits are used later
Linear Page Table

32 bit virtual address: 0x1005a6f

frames in memory

Example architecture
- Virtual address size: 32 bits
- Page size: $2^{12}$ B (4KB)

Example vaddr->paddr translation:

- vaddr = 0x01005a6f
- offset = vaddr & 0xffff = 0xa6f
- page = vaddr >> 12 = 0x1005
  // pte is page_table[page]
- fr = page_table[page].frame = 0xe
  // (fr <<12) is frame phys address
- paddr = (fr << 12) | offset = 0xea6f
Storing Page Table

- Page table is large and stored in memory

- Problem
  - MMU needs to access page table on every instruction
  - Each memory access requires two memory accesses
    - First one for page table entry in physical memory
      - Where is that shown in the example translation in previous slide?
    - Second one for the physical memory location being accessed

- Solution
  - Cache PTE in MMU, handle misses from physical memory
  - Cache is called translation lookaside buffer (TLB)
    - Discussed later
Accessing Page Tables

- MMU needs to know the location of page table in physical memory
- Paging MMU has a page table register (PTR) that stores the location of the start of the page table
  - Similar to base register
- OS associates a separate page table for each process
  - Thus, each process has its own address space
- Implementing context switch
  - context switch = thread switch + address space switch
  - OS implements address space switching by changing PTR to the start of the appropriate page table for each process
Page Table Size

- Page table size
  - \#PTE = \#pages = 
    - virtual address space size / page size = \(2^{32} / 2^{12} = 2^{20}\)
  - Typically, each PTE is word sized (4B)
  - Page table size = \#PTE \times PTE size = 2^{20} \times 4B = 4 \text{ MB}
  - Each process needs 4MB for its own page table!

- Effect of page size on performance
  - Smaller size => lower internal fragmentation
  - Larger size => fewer PTE
    - lower memory overhead for PTE, but more internal fragmentation
    - better performance (we will see this later)
Reducing Page Table Size

- Consider a small process with one page of text, data and stack each
  - It requires 3 pages of memory (12KB)
  - It requires 4MB of memory for page tables!

- With the linear page table design, we need a PTE even when the corresponding page is not in use
  - Notice that most of the page table entries has invalid
    - 3 pages are valid, \((2^{20} – 3)\) entries are invalid
    - i.e., the page table array is sparsely populated
  - Need a more space efficient data structure that avoids storing invalid entries
    - Use a tree data structure, instead of an array
    - A parent does not need to store a child sub-tree if all elements of the sub-tree are invalid
Multi-Level Page Table (in this case with 2 levels)

32 bit virtual address: 0x1005a6f

| 10 bits | 10 bits | 12 bits |

PT1  PT2  offset

PT1

Top-level page table

PT2

Second level page tables

frames in memory

 PTR

0xa000

0xe000

0xa6f

0x0

0xa

0xe v
# Example Translation

<table>
<thead>
<tr>
<th>vaddr = 0x1005a6f</th>
<th>0001 0000 0000 0101 a6f</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset = vaddr &amp; 0xffff = 0xa6f</td>
<td>get 12 low bits of addr</td>
</tr>
<tr>
<td>pg2 = (vaddr &gt;&gt; 12) &amp; 0x3ff = 0x5</td>
<td>get next 10 bits of addr</td>
</tr>
<tr>
<td>pg1 = vaddr &gt;&gt; (12+10) = 0x4</td>
<td>get top 10 bits of addr</td>
</tr>
<tr>
<td>pt_1 = page_table_register</td>
<td>base of top page table</td>
</tr>
<tr>
<td>pt_2 = pt_1[pg1].frame &lt;&lt; 12 = 0xa000</td>
<td>look up top page table to find physical memory location of second level page table. pt_2 is stored page aligned</td>
</tr>
<tr>
<td>fr = pt_2[pg2].frame = 0xe</td>
<td>look up second page table</td>
</tr>
<tr>
<td>paddr = (fr &lt;&lt; 12)</td>
<td>offset = 0xea6f</td>
</tr>
</tbody>
</table>
Comparing Single & Two-Level Page Table

- Is address translation faster with a single-level page table or a two-level page table?

- How does two-level page table save space compared to the single-level page table?
  - Not all pages within an address space are allocated
  - E.g., consider the region between heap and stack
    - This region does not need any frames
    - So there is no need to maintain mapping information
  - When a 2^{nd} level page table is entirely invalid, it is not allocated, and corresponding entry in top-level page table is marked invalid
Summary

- Contiguous memory allocation makes it hard to grow programs and causes external fragmentation
- It is easier to allocate memory non-contiguously, but it is easier to program a contiguous address space
- We can solve this dilemma by using MMU hardware to virtualize the address space of a process
  - Process sees contiguous (virtual) memory addresses
  - MMU translates these addresses to physical memory addresses that may be non-contiguous
Summary

- A paging MMU breaks the virtual address space of a process into fixed size pages and maps each page to a physical memory frame
  - Enables growing programs at page granularity
  - No external fragmentation
  - Internal fragment per memory region is roughly half page size

- Page mapping information is stored in memory
  - Single level page table stores mappings in an array
    - Inefficient because address space is generally sparse
  - Multi-level page table stores mappings in a tree structure
    - Reduces memory overhead significantly, but
    - Requires more time for translating virtual to physical address
Think Time

- What is the purpose of a page table?
- Where is the page table located?
- How does the h/w locate the page table?
- How many address bits are used for page offset when page size is 2KB?
- Discuss advantages and disadvantages of linear and multi-level page tables
What is the purpose of a page table?
- page table stores virtual to physical page mappings

Where is the page table located?
- page table is stored in physical memory

How does the h/w locate the page table?
- OS programs a register called the page table register (PTR) to point to the start of the page table. The address stored in the PTR is a physical address. The MMU looks up the PTR to find the start of the page table in physical memory. Later, we will discuss software-managed TLB, in which case, there is no PTR and h/w does not directly lookup the page table.
Think Time Answers

- How many address bits are used for page offset when page size is 2KB?
  - 2KB = $2^{11}$, so 11 bits for page offset

- Discuss advantages and disadvantages of linear and multi-level page tables
  - Linear page table
    - Pros: fast, since there is only one memory access to map a virtual address to a physical address
    - Cons: large memory overhead
  - Two-level page table
    - Cons: slower, since there are two memory accesses to map a virtual address to a physical address
    - Pros: lower memory overhead compared to linear page table