Address Translation

March 4th, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: Starvation vs Deadlock

- **Starvation vs. Deadlock**
  - **Starvation**: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - **Deadlock**: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
    - Thread B owns Res 2 and is waiting for Res 1

  \[\text{Deadlock} \Rightarrow 	ext{Starvation but not vice versa}\]

- **Deadlock** can’t end without external intervention
- **Starvation** can end (but doesn’t have to)
Recall: Four requirements for Deadlock

- **Mutual exclusion**
  - Only one thread at a time can use a resource.

- **Hold and wait**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads.

- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it.

- **Circular wait**
  - There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads:
    - \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    - \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    - ...
    - \( T_n \) is waiting for a resource that is held by \( T_1 \)
Recall: Address translation

- **Address Space:**
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- **Consequently, two views of memory:**
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views
- **Translation essential to implementing protection**
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- **With translation, every program can be linked/loaded into same region of user address space**
Recall: General Address Translation

Virtual Address Space 1

Translation Map 1

Physical Address Space

Translation Map 2

Virtual Address Space 2

Program 1

Program 2

OS heap & Stacks

OS code

OS data
Simple Base and Bounds (CRAY-1)

- Could use base/limit for *dynamic address translation* - translation happens at execution:
  - Alter address of every load/store by adding “base”
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

\[
\text{CPU} \quad \longrightarrow \quad \text{Virtual Address} \quad \downarrow \quad \text{Base} \quad \longrightarrow \quad \text{Physical Address} \quad \leftarrow \quad \text{Limit} \quad \downarrow \quad \text{No: Error!} \quad \longrightarrow \quad \text{DRAM}
\]
Issues with Simple B&B Method

- **Fragmentation problem**
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- **Missing support for sparse address space**
  - Would like to have multiple chunks/program
  - E.g.: Code, Data, Stack
- **Hard to do inter-process sharing**
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process
More Flexible Segmentation

- **Logical View**: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc

- **Each segment is given region of contiguous memory**
  - Has a base and limit
  - Can reside anywhere in physical memory
Implementation of Multi-Segment Model

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range

- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: mov [es:bx],ax.

- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well
Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Segment ID 0
- Base: 0x0000
- Limit: 0x4000

Segment ID 1
- Base: 0x4000
- Limit: 0x5C00

Segment ID 2
- Base: 0x4000
- Limit: 0xF000

Segment ID 3
- Base: 0xF000
- Limit: 0x1000

Physical Address Space

Might be shared

Space for Other Apps

Shared with Other Apps
Example of segment translation

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #: 0; Offset: 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4→PC

2. Fetch 0x244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x360. Translated to Physical=0x4360. Get “li $v0,0”
   Move 0x0000 → $v0, Move PC+4→PC

4. Fetch 0x364. Translated to Physical=0x4364. Get “lb $t0,($a0)”
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #: 1; Offset: 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC
Observations about Segmentation

• Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    » If it does, trap to kernel and dump core
• When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
• Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
• What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called “swapping”)

What if more segments than will fit into memory?

• Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
• Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory
Problems with Segmentation

• Must fit variable-sized chunks into physical memory

• May move processes multiple times to fit everything

• Limited options for swapping to disk

• **Fragmentation**: wasted space
  - **External**: free gaps between allocated chunks
  - **Internal**: don’t need all memory within allocated chunks
Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation:
      00110001110001101 ... 110010
    » Each bit represents page of physical memory
      1⇒allocated, 0⇒free

- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment
How to Implement Paging?

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    » Permissions include: Valid bits, Read, Write, etc

- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions
Simple Page Table Example

Example (4 byte pages)

Virtual Memory

```
0x00 0000 0000
0x04 0000 0100
0x06 0000 1000
0x08 0000 1000
0x09 0000 1000
```

Page Table

```
0001 0000
0000 1100
0000 0100
```

Physical Memory

```
0x00
0x04
0x08
0x0C
0x10
```

Virtual Memory

```
0000 0110
0000 1001
```

0x05!

0x0E!
What about Sharing?

Virtual Address (Process A):

Virtual Address (Process B):

Offset

PageTablePtrA

PageTablePtrB

Virtual Page #

Shared Page

This physical page appears in address space of both processes
Memory Layout for Linux 32-bit

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0xc0000000 == TASK_SIZE
- Random stack offset
- RLIMIT_STACK (e.g., 8MB)
- Random mmap offset

program break
brk
- Random brk offset

end_data
start_data
end_code

0x08048000
Summary: Simple Page Table

Virtual memory view

- **stack**
- **heap**
- **data**
- **code**

Page Table

- Page #
- Offset

Virtual memory view with page offsets:

- 1111 1111
- 1111 0000
- 1100 0000
- 1000 0000
- 0100 0000
- 0000 0000

Physical memory view with page offsets:

- Stack
- Heap
- Data
- Code

[Diagram of page table and memory views]
Summary: Simple Page Table

Virtual memory view

- stack
- heap
- data
- code

Page Table

- 1111 1111
- 1110 0000
- 1000 0000
- 0100 0000
- 0000 0000

Physical memory view

- stack
- heap
- data
- code

What happens if stack grows to 1110 0000?
Summary: Simple Page Table

Virtual memory view

0000 0000 00000000
0001 0000 00000000
0100 0000 01000000
1100 0000 11000000
1110 0000 11100000
1111 0000 11110000

Page Table

0000 0000
0001 0000
0100 0000
0101 0000
0110 0000
1110 0000

Allocate new pages where room!

Physical memory view

0000 0000 00000000
0001 0000 00000000
0100 0000 01000000
1100 0000 11000000
1110 0000 11100000
1111 0000 11110000

Allocate new pages where room!

Stack

Heap

Data

Code

Page # offset
Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- Analysis
  - Pros
    » Simple memory allocation
    » Easy to Share
  - Con: What if address space is sparse?
    » E.g. on UNIX, code starts at 0, stack starts at \((2^{31} - 1)\).
    » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    » Not all pages used all the time \(\Rightarrow\) would be nice to have working set of page table in memory

- How about combining paging and segmentation?
  - Segments with pages inside them?
  - Need some sort of multi-level translation
Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table → memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Virtual</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sea #</td>
<td>Page #</td>
<td></td>
</tr>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical</th>
<th>Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>V, R</td>
</tr>
<tr>
<td>#1</td>
<td>V, R</td>
</tr>
<tr>
<td>#2</td>
<td>V, R, W</td>
</tr>
<tr>
<td>#3</td>
<td>V, R, W</td>
</tr>
<tr>
<td>#4</td>
<td>N</td>
</tr>
<tr>
<td>#5</td>
<td>V, R, W</td>
</tr>
</tbody>
</table>

  Access Error

• What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)
What about Sharing (Complete Segment)?

<table>
<thead>
<tr>
<th>Process A</th>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

| Base0     | Limit0        | V              |        |
| Base1     | Limit1        | V              |        |
| Base2     | Limit2        | V              |        |
| Base3     | Limit3        | N              |        |
| Base4     | Limit4        | V              |        |
| Base5     | Limit5        | N              |        |
| Base6     | Limit6        | N              |        |
| Base7     | Limit7        | V              |        |

<table>
<thead>
<tr>
<th>Process B</th>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td>page #1</td>
<td>V,R</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>
Fix for sparse address space: The two-level page table

**Virtual Address:**
- 10 bits (Virtual P1 index)
- 10 bits (Virtual P2 index)
- 12 bits (Offset)

**Physical Address:**
- Physical Page #
- Offset

---

**Tree of Page Tables**
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register

**Valid bits on Page Table Entries**
- Don't need every 2nd-level table
- Even when exist, 2nd-level tables can reside on disk if not in use

---

PageTablePtr

4 bytes

---

4KB
Summary: Two-Level Paging

Virtual memory view

- stack
- heap
- data
- code

Page Tables (level 1)

- null
- null

Page Tables (level 2)

- stack
- heap
- data
- code

Physical memory view
Summary: Two-Level Paging

Virtual memory view

- stack
- heap
- data
- code

Page Table (level 1)

- 111
- 110
- 101
- 000
- 011
- 010
- 001

Page Tables (level 2)

- 11
- 10
- 01
- 00

Physical memory view

- stack
- heap
- data
- code

Page Tables (level 2)

- 11101
- 11100
- 11000
- 01111
- 01110
- 01011
- 01010
- 00111
- 00110
- 00011
- 00010
- 00000
- 1110000
- 1000000
- 0001000
- 0000000

1001 0000
(0x90)

110 0000
(0x80)
Multi-level Translation Analysis

• Pros:
  - Only need to allocate as many page table entries as we need for application
    » In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    » Share at segment or page level

• Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    » Seems very expensive!
Summary

• Segment Mapping
  - Segment registers within processor
  - Segment ID associated with each access
    » Often comes from portion of virtual address
    » Can come from bits in instruction instead (x86)
  - Each segment contains base and limit information
    » Offset (rest of address) adjusted by adding base

• Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory

• Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space