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: Simple Segmentation (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

Segmentation with SegID = 0 and SegID = 1

Physical Address Space

Space for Other Apps

Shared with Other Apps

Might be shared
Recall: 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
Recall: Simple 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 2 million page table entries!
  - Con: What if table really big?
    » Not all pages used all the time ⇒ 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
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 (grown 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 = "Gee's own prototype";

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

program break
break
start_brk
brk

end_data

http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png
Fix for sparse address space: The two-level page table

- 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
What is in a Page Table Entry?

- **What is in a Page Table Entry (or PTE)?**
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only

- **Example: Intel x86 architecture PTE:**
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

<table>
<thead>
<tr>
<th>Page Frame Number</th>
<th>Free (OS)</th>
<th>P</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Physical Page Number)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- P: Present (same as “valid” bit in other architectures)
- W: Writeable
- U: User accessible
- PWT: Page write transparent: external cache write-through
- PCD: Page cache disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty (PTE only): page has been modified recently
- L: $L=1 \Rightarrow$ 4MB page (directory only).
- Bottom 22 bits of virtual address serve as offset
Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    » Region of address space is actually invalid or
    » Page/directory is just somewhere else than memory
  - Validity checked first
    » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives copy of parent address space to child
    » Address spaces disconnected after child created
  - How to do this cheaply?
    » Make copy of parent’s page tables (point at same memory)
    » Mark entries in both sets of page tables as read-only
    » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background
Summary: Two-Level Paging

Virtual memory view:
- stack
- heap
- data
- code

Page Table (level 1):
- null
- null

Page Tables (level 2):
- 11101
- 11100
- 11111
- 10110
- null
- 10000
- 01111
- 01110
- 01101
- 01100
- 01011
- 01010
- 00101
- 00100
- 00011
- 00010

Physical memory view:
- stack
- heap
- data
- code
Summary: Two-Level Paging

Virtual memory view

- stack
- heap
- data
- code

Page Table (level 1)

- 1001 0000 (0x90)
- 111
- 110
- 101
- 100
- 011
- 010
- 001
- 000

Page Tables (level 2)

- 11
- 10
- 01
- 00

Physical memory view

- stack
- heap
- data
- code

0000 0000
0001 0000
1000 0000 (0x80)
1110 0000
0001 0000
0000 0000
What about a tree of tables?
- Lowest level page table \(\Rightarrow\) memory still allocated with bitmap
- Higher levels often segmented

Could have any number of levels. Example (top segment):

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

Process A

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

Shared Segment

<table>
<thead>
<tr>
<th>page #0</th>
<th>V,R</th>
</tr>
</thead>
<tbody>
<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>

Process B

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</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>
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!
Making it real:
X86 Memory model with segmentation (16/32-bit)
X86_64: Four-level page table!

48-bit Virtual Address: 9 bits 9 bits 9 bits 9 bits 12 bits

Virtual P1 index  Virtual P2 index  Virtual P3 index  Virtual P4 index  Offset

4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

Physical Address: (40-50 bits)

PageTablePtr

8 bytes

Physical Page # 12bit Offset
### IA64: 64bit addresses: Six-level page table?!!?

<table>
<thead>
<tr>
<th>64bit Virtual Address:</th>
<th>7 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual P1 index</td>
<td>Virtual P2 index</td>
<td>Virtual P3 index</td>
<td>Virtual P4 index</td>
<td>Virtual P5 index</td>
<td>Virtual P6 index</td>
<td>Offset</td>
<td></td>
</tr>
</tbody>
</table>

- **No!**
- Too slow
- Too many almost-empty tables
Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    » Much of process space may be out on disk or not in use

- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces

- Cons: Complexity of managing hash changes
  - Often in hardware!
IPT address translation

- Need an associative map from VM page to IPT address:
  - Use a hash map

<table>
<thead>
<tr>
<th>PID</th>
<th>VM Page</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VMpage0</td>
<td>0x0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0x1</td>
</tr>
<tr>
<td>0</td>
<td>VMpage1</td>
<td>0x2</td>
</tr>
<tr>
<td>0</td>
<td>VMpage2</td>
<td>0x3</td>
</tr>
<tr>
<td></td>
<td>free</td>
<td>0x4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0x5</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0x6</td>
</tr>
<tr>
<td>0</td>
<td>VMpage3</td>
<td>0x7</td>
</tr>
</tbody>
</table>

Process 0 virtual address

Physical address

Hash VM page #

Inverse Page Table

 VMpage0, pid 0
 VMpage2, pid 0
 VMpage1, pid 0
 VMpage3, pid 0
**Summary: Inverted Table**

<table>
<thead>
<tr>
<th>Virtual memory view</th>
<th>Physical memory view</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111</td>
<td>stack</td>
</tr>
<tr>
<td>1110 0000</td>
<td>stack</td>
</tr>
<tr>
<td>1100 0000</td>
<td>stack</td>
</tr>
<tr>
<td>1000 0000</td>
<td>stack</td>
</tr>
<tr>
<td>0100 0000</td>
<td>stack</td>
</tr>
<tr>
<td>0000 0000</td>
<td>stack</td>
</tr>
</tbody>
</table>

**Inverted Table**

Hash(procID & virt. page #) = phys. page #

- h(11111) = 11101
- h(11110) = 11100
- h(11101) = 10111
- h(11100) = 10110
- h(10010) = 10000
- h(10001) = 01111
- h(10000) = 01110
- h(01111) = 01101
- h(01110) = 01100
- h(01101) = 01011
- h(01100) = 01010
- h(00101) = 00101
- h(00100) = 00100
- h(00011) = 00011
- h(00010) = 00010
- h(00001) = 00001
- h(00000) = 00000
<table>
<thead>
<tr>
<th>Address Translation Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td><strong>Simple Segmentation</strong></td>
</tr>
<tr>
<td><strong>Paging (single-level page)</strong></td>
</tr>
<tr>
<td><strong>Paged segmentation</strong></td>
</tr>
<tr>
<td><strong>Two-level pages</strong></td>
</tr>
<tr>
<td><strong>Inverted Table</strong></td>
</tr>
</tbody>
</table>
How is the translation accomplished?

• What, exactly happens inside MMU?
• One possibility: Hardware Tree Traversal
  - For each virtual address, takes page table base pointer and
    traverses the page table in hardware
  - Generates a “Page Fault” if it encounters invalid PTE
    » Fault handler will decide what to do
    » More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
• Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!
• In fact, need way to cache translations for either case!
Recall: Dual-Mode Operation

- Can a process modify its own translation tables?
  - NO!
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow

- Recall: To Assist with Protection, **Hardware** provides at least two modes (Dual-Mode Operation):
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode

- Certain operations restricted to Kernel mode:
  - Including modifying the page table (CR3 in x86), and segment registers
  - Have to transition into Kernel mode before you can change them
How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    » Point at code in memory so program can execute
    » Possibly point at statically initialized data
  - Run Program:
    » Set machine registers
    » Set hardware pointer to translation table
    » Set processor status word for user mode
    » Jump to start of program

- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)
Recall: User $\rightarrow$ Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?

**System call**: Voluntary procedure call into kernel
- Hardware for controlled User $\rightarrow$ Kernel transition
- Can any kernel routine be called?
  - No! Only specific ones.
- System call ID encoded into system call instruction
  - Index forces well-defined interface with kernel
User → Kernel (Exceptions: Traps and Interrupts)

• A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
• Other sources of Synchronous Exceptions ("Trap"):
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
• Interrupts are Asynchronous Exceptions
  - Examples: timer, disk ready, network, etc....
  - Interrupts can be disabled, traps cannot!
• On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
• Actual handler typically saves registers, other CPU state, and switches to kernel stack
Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)

- Protection via Strong Typing
  - Restrict programming language so that you can’t express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java

- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can’t step out of bounds
    » Compiler puts in checks for every “dangerous” operation (loads, stores, etc). Again, need special loader.
    » Alternative, compiler generates “proof” that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)
Caching Concept

• **Cache**: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant

• Caching underlies many of the techniques that are used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc...

• Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive

• Important measure: Average Access time =
  \[(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})\]
Why Bother with Caching?

Processor–DRAM Memory Gap (latency)

“Moore's Law”
(really Joy's Law)

“Less' Law?”

Processor-Memory Performance Gap:
(grows 50% / year)

μProc
60%/yr.
(2X/1.5yr)

DRAM
9%/yr.
(2X/10 yrs)
Another Major Reason to Deal with Caching

Virtual Address:

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

Virtual Seg # | Virtual Page # | Offset

| page #0 | V, R |
| page #1 | V, R |
| page #2 | V, R, W |
| page #3 | V, R, W |
| page #4 | N |
| page #5 | V, R, W |

Physical Address

Check Perm

Access Error

Access Error

- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access???
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")
Why Does Caching Help? Locality!

- **Temporal Locality** (Locality in Time):
  - Keep recently accessed data items closer to processor
- **Spatial Locality** (Locality in Space):
  - Move contiguous blocks to the upper levels

![Diagram showing probability of reference and address space with temporal and spatial locality concepts.](image-url)
Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology
A Summary on Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant

- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size

- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- **Coherence** (Invalidation): other process (e.g., I/O) updates memory
How is a Block found in a Cache?

- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don’t have data select field
Review: Direct Mapped Cache

- **Direct Mapped** $2^N$ byte cache:
  - The uppermost $(32 - N)$ bits are always the **Cache Tag**
  - The lowest $M$ bits are the **Byte Select** (Block Size = $2^M$)

- **Example**: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

```
<table>
<thead>
<tr>
<th>Byte 32</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache Tag</td>
<td>Cache Data</td>
</tr>
<tr>
<td>0x50</td>
<td>Byte 31</td>
</tr>
<tr>
<td></td>
<td>Byte 63</td>
</tr>
<tr>
<td></td>
<td>Byte 1</td>
</tr>
<tr>
<td></td>
<td>Byte 33</td>
</tr>
<tr>
<td></td>
<td>Byte 0</td>
</tr>
<tr>
<td></td>
<td>Byte 1023</td>
</tr>
<tr>
<td></td>
<td>Byte 992</td>
</tr>
</tbody>
</table>
```

**Diagram:**
- **Valid Bit**
- **Cache Index**: Ex: 0x01
- **Byte Select**: Ex: 0x00
- **Cache Tag**: Ex: 0x50
Summary (1/2)

• 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
• Inverted page table
  - Size of page table related to physical memory size
• PTE: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)
Summary (2/2)

• The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    » Temporal Locality: Locality in Time
    » Spatial Locality: Locality in Space

• Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices