CS362
Operating Systems
Simple Segmentation: Base and Limit

- Can use base/limit for dynamic address translation (Simple form of “segmentation”):
  - Alter every address 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
Base and Limit segmentation discussion

• Provides level of indirection
  - OS Can move bits around behind program’s back
  - Can be used to correct if program needs to grow beyond its bounds or coalesce fragments of memory

• Only OS gets to change the base and limit!
  - Would defeat protection

• What gets saved/restored on a context switch?
  - Everything from before + base/limit values
  - Or: How about complete contents of memory (out to disk)?
    » Called “Swapping”

• Hardware cost
  - 2 registers/Adder/Comparator
  - Slows down hardware because need to take time to do add/compare on every access

• Base and Limit Pros: Simple, relatively fast
Cons for Simple Segmentation Method

- **Fragmentation problem (complex memory allocation)**
  - Not every process is the same size
  - Over time, memory space becomes fragmented
  - Really bad if want space to grow dynamically (e.g. heap)

- **Other problems for process maintenance**
  - Doesn’t allow heap and stack to grow independently
  - Want to put these as far apart in virtual memory space as possible so that they can grow as needed

- **Hard to do inter-process sharing**
  - Want to share code segments when possible
  - Want to share memory between processes
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”?  
  - Can mark segments as invalid; requires check as well
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:

- 0x0000
- 0x4000
- 0x8000
- 0xC000

Physical Address Space:

- 0x0000
- 0x4000
- 0x4800
- 0x5C00
- 0xF000

Might be shared:

Space for Other Apps

Shared with Other Apps
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”

2. Fetch 0x244. Translated to Physical=0x4244. Get “jal strlen”

3. Fetch 0x360. Translated to Physical=0x4360. Get “li $v0, 0 ;count

4. Fetch 0x364. Translated to Physical=0x4364. Get “lb $t0, ($a0)”

5. Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
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”)
10
11/1/2006

Schematic View of Swapping

• 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
Paging: Physical Memory in Fixed Size Chunks

- **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
- **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

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

```
Virtual Address: Page Table # Offset
Page Table Ptr
Page Table Size
Access Error
```

```
<table>
<thead>
<tr>
<th>Physical Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
Physical Address
Check Perm
Access Error
```

11/1/2006
What about Sharing?

Virtual Address (Process A):

<table>
<thead>
<tr>
<th>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>

PageTablePtrA

Virtual Address: Process B

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

PageTablePtrB

Shared Page

This physical page appears in address space of both processes
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?
Multi-level Translation

- 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):

  - 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)?

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 #</th>
<th>Permissions</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>

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>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>
Another common example: two-level page table

Physical Address: 10 bits 10 bits 12 bits
Offset

Virtual Address: Virtual P1 index P2 index Offset

PageTablePtr

- 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 2\textsuperscript{nd}-level table
  - Even when exist, 2\textsuperscript{nd}-level tables can reside on disk if not in use
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 (need additional reference counting)

• 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!
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!
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

• Inverted page table
  - Size of page table related to physical memory size