Virtual Memory

Kevin Webb
Swarthmore College
March 8, 2018
Today’s Goals

- Describe the mechanisms behind address translation.

- Analyze the performance of address translation alternatives.

- Explore page replacement policies for disk swapping.
Address Translation: Wish List

- Map virtual addresses to physical addresses.
- Allow multiple processes to be in memory at once, but isolate them from each other.
- Determine which subset of data to keep in memory / move to disk.
- Allow the same physical memory to be mapped in multiple process VASes.
- Make it easier to perform placement in a way that reduces fragmentation.
- Map addresses quickly with a little HW help.

Combination of hardware and OS, working together.

In hardware, MMU: Memory Management Unit
Simple (Unrealistic) Translation Example

- Process $P_2$’s virtual addresses don’t align with physical memory’s addresses.

- Determine offset from physical address 0 to start of $P_2$, store in $base$. 
Generalizing

• Problem: process may not fit in one contiguous region
Generalizing

• Problem: process may not fit in one contiguous region

• Solution: keep a table (one for each process)
  • Keep details for each region in a row
  • Store additional metadata (ex. permissions)

• Interesting questions:
  • How many regions should there be (and what size)?
  • How to determine which row we should use?
Defining Regions - Two Approaches

• Segmentation:
  • Partition address space and memory into segments
  • Segments have varying sizes

• Paging:
  • Partition address space and memory into pages
  • Pages are a constant, fixed size
Fragmentation

**Internal**
- Process asks for memory, doesn’t use it all.
- Possible reasons:
  - Process was wrong about needs
  - OS gave it more than it asked for
- *internal*: within an allocation

**External**
- Over time, we end up with these small gaps that become more difficult to use (eventually, wasted).
- *external*: unused memory between allocations
Which scheme is better for reducing internal and external fragmentation.

A. Segmentation is better than paging for both forms of fragmentation.

B. Segmentation is better for internal fragmentation, and paging is better for external fragmentation.

C. Paging is better for internal fragmentation, and segmentation is better for external fragmentation.

D. Paging is better than segmentation for both forms of fragmentation.
Which would you use? Why? Pros/Cons?

A. Segmentation:
   - Partition address space and memory into segments
   - Segments have varying sizes

B. Paging:
   - Partition address space and memory into pages
   - Pages are a constant, fixed size

C. Something else (what?)
Segmentation vs. Paging

• A segment is good *logical* unit of information
  • Can be sized to fit any contents
  • Easy to share large regions (e.g., code, data)
  • Protection requirements correspond to logical data segment

• A page is good *physical* unit of information
  • Simple physical memory placement
  • No external fragmentation
  • Constant sizes make it easier for hardware to help
Generalizing

• Problem: process may not fit in one contiguous region

• Solution: keep a table (one for each process)
  • Keep details for each region in a row
  • Store additional metadata (ex. permissions)

• Interesting questions:
  • How many regions should there be (and what size)?
  • How to determine which row we should use?
For both segmentation and paging...

- Each process gets a table to track memory address translations.

- When a process attempts to read/write to memory:
  - use high order bits of virtual address to determine which row to look at in the table
  - use low order bits of virtual address to determine an offset within the physical region
Address Translation

Virtual Address

Upper bits

Lower bits

Table

Meta | Phy Loc | Perm | ...  

Physical Address

Physical Memory
Performance Implications

Without VM:
Go directly to address in memory.

With VM:
Do a lookup in memory to determine which address to use.

Concept: level of indirection
Defining Regions - Two Approaches

- **Segmentation:**
  - Partition address space and memory into segments
  - Segments have varying sizes

- **Paging:**
  - Partition address space and memory into pages
  - Pages are a constant, fixed size
Segment Table

• One table per process

• Where is the *table* located in memory?
  • Segment table base register (STBR)
  • Segment table size register (STSR)

• Table entry elements
  • V: valid bit (does it contain a mapping?)
  • Base: segment location in physical memory
  • Bound: segment size in physical memory
  • Permissions
Address Translation

- Physical address = base of $s + i$
- First, do a series of checks...

Virtual Address

Segment $s$  Offset $i$

V | Base | Bound | Perm | ...

Physical Address
Check if Segment $s$ is within Range

Virtual Address

Segment $s$  Offset $i$

STBR  STSR

$s < \text{STSR}$

V | Base | Bound | Perm | ...

Physical Address
Check if Segment Entry $s$ is Valid

Virtual Address

Segment $s$  Offset $i$

STBR
STSR

$V = 1$

$V$, Base, Bound, Perm, ...

Physical Address
Check if Offset $i$ is within Bounds

Virtual Address

Segment $s$  Offset $i$

STBR  STSR

$V$  Base  Bound  Perm  ...

$i < \text{Bound}$

Physical Address
Check if Operation is Permitted
Translate Address

Virtual Address

Segment $s$  Offset $i$

STBR  STSR

$V$  Base  Bound  Perm  ...

Physical Address

$+_

Translate Address
Sizing the Segment Table

Virtual Address

<table>
<thead>
<tr>
<th>Segment s</th>
<th>Offset i</th>
</tr>
</thead>
</table>

Number of bits $n$ specifies max size of table, where number of entries $= 2^n$

Number of bits $n$ specifies max size of segment

Number of bits needed to address physical memory

Number of bits needed to specify max segment size

Helpful reminder:

$2^{10} \Rightarrow$ Kilobyte

$2^{20} \Rightarrow$ Megabyte

$2^{30} \Rightarrow$ Gigabyte
Example of Sizing the Segment Table

- Given 32-bit virtual address space, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset
5 bit segment address, 32 bit logical address, 1 GB Physical memory.
How many entries (rows) will we have in our segment table?

A. 32: The logical address size is 32 bits

B. 32: The segment address is five bits

C. 30: We need to address 1 GB of physical memory

D. 27: We need to address up to the maximum offset
Example of Sizing the Segment Table

- Given 32-bit virtual address space, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset
How many bits do we need for the base?

A. 30 bits, to address 1 GB of physical memory.
B. 5 bits, because we have 32 rows in the segment table.
C. 27 bits, to address any potential offset value.
How many bits do we need for the base?

A. 30 bits, to address 1 GB of physical memory.
B. 5 bits, because we have 32 rows in the segment table.
C. 27 bits, to address any potential offset value.
Example of Sizing the Segment Table

- Given 32-bit virtual address space, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset
How many bits do we need for the bound?

A. 5 bits: the size of the segment portion of the virtual address.
B. 27 bits: the size of the offset portion of the virtual address.
C. 32 bits: the size of the virtual address.
How many bits do we need for the bound?

A. 5 bits: the size of the segment portion of the virtual address.
B. 27 bits: the size of the offset portion of the virtual address.
C. 32 bits: the size of the virtual address.
Example of Sizing the Segment Table

- Given 32 bit logical, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset

```
Segment s: 5 bits
Offset i: 27 bits
```

- 5 bits to address $2^5 = 32$ entries
- 30 bits needed to address 1 GB
- 27 bits needed to size up to 128 MB
Example of Sizing the Segment Table

<table>
<thead>
<tr>
<th>Segment s: 5 bits</th>
<th>Offset i: 27 bits</th>
</tr>
</thead>
</table>

- 5 bits to address $2^5 = 32$ entries
- 30 bits needed to address 1 GB
- 8 bytes needed to contain 61 (1+30+27+3+...) bits
- 27 bits needed to size up to 128 MB
- Total table size?

- Given 32 bit logical, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset
Example of Sizing the Segment Table

- Given 32 bit logical, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset

Segment $s$: 5 bits
Offset $i$: 27 bits

5 bits to address $2^5 = 32$ entries

30 bits needed to address 1 GB

27 bits needed to size up to 128 MB

8 bytes needed to contain 61 (1+30+27+3+...) bits

Table size = $32 \times 8 = 256$ bytes
Pros and Cons of Segmentation

• Pro: Each segment can be
  • located independently
  • separately protected
  • grown/shrunken independently

• Pro: Small segment table size

• Con: Variable-size allocation
  • Difficult to find holes in physical memory
  • External fragmentation
Defining Regions - Two Approaches

• Segmentation:
  • Partition address space and memory into segments
  • Segments have varying sizes

• Paging:
  • Partition address space and memory into pages
  • Pages are a constant, fixed size
Paging Vocabulary

• For each process, the *virtual* address space is divided into fixed-size pages.

• For the system, the *physical* memory is divided into fixed-size frames.

• The size of a page is equal to that of a frame.
  • Often 4 KB in practice.
  • Some CPUs allow for small and large pages at the same time.
**Page Table**

- **One table per process**
- **Table parameters in memory**
  - Page table base register
  - Page table size register
- **Table entry elements**
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
  - Frame: location in phy mem
  - Perm: access permissions

<table>
<thead>
<tr>
<th>PTBR</th>
<th>PTSR</th>
<th>V</th>
<th>R</th>
<th>D</th>
<th>Frame</th>
<th>Perm</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Address Translation

- Physical address = frame of $p + \text{offset } i$
- First, do a series of checks...

<table>
<thead>
<tr>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page $p$</td>
</tr>
</tbody>
</table>

- Physical Address = frame of $p + \text{offset } i$
- First, do a series of checks...
Check if Page $p$ is Within Range

Virtual Address

<table>
<thead>
<tr>
<th>Page $p$</th>
<th>Offset $i$</th>
</tr>
</thead>
</table>

- **PTBR**
- **PTSR**

$p < \text{PTSR}$

Physical Address
Check if Page Table Entry $p$ is Valid

Virtual Address

<table>
<thead>
<tr>
<th>Page $p$</th>
<th>Offset $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>V R D</td>
<td>Frame Perm</td>
</tr>
</tbody>
</table>

V == 1

PTBR
PTSR

Physical Address
Check if Operation is Permitted

Virtual Address

Page \( p \)  Offset \( i \)

PTBR
PTSR

Perm (op)

V | R | D | Frame | Perm | ...

Physical Address
Translate Address

Virtual Address

Page $p$ | Offset $i$
---|---
PTBR | PTSR

$V\ R\ D\ |\ Frame\ |\ Perm\ ...$

Physical Address

concat
Frames are all the same size. Only need to store the frame number in the table, not exact address!
Sizing the Page Table

Virtual Address

Page \( p \)

Offset \( i \)

Number of bits \( n \)
specifies max size of table, where number of entries = \( 2^n \)

Number of bits needed to address physical memory *in units of frames*

Number of bits specifies page size
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
• Given: 32 bit virtual addresses, 1 GB physical memory
  • Address partition: 20 bit page number, 12 bit offset
How many entries (rows) will there be in this page table?

A. \(2^{12}\), because that’s how many the offset field can address

B. \(2^{20}\), because that’s how many the page field can address

C. \(2^{30}\), because that’s how many we need to address 1 GB

D. \(2^{32}\), because that’s the size of the entire address space

• Given: 32 bit virtual addresses, 1 GB physical memory
  • Address partition: 20 bit page number, 12 bit offset
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
What will be the frame size, in bytes?

A. $2^{12}$, because that’s how many bytes the offset field can address

B. $2^{20}$, because that’s how many bytes the page field can address

C. $2^{30}$, because that’s how many bytes we need to address 1 GB

D. $2^{32}$, because that’s the size of the entire address space

• Given: 32 bit virtual addresses, 1 GB physical memory
  • Address partition: 20 bit page number, 12 bit offset
Example of Sizing the Page Table

Given: 32 bit virtual addresses, 1 GB physical memory
   – Address partition: 20 bit page number, 12 bit offset
How many bits do we need to store the frame number?

• Given: 32 bit virtual addresses, 1 GB physical memory
  • Address partition: 20 bit page number, 12 bit offset

• A: 12    B: 18    C: 20    D: 30    E: 32
Example of Sizing the Page Table

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
Example of Sizing the Page Table

• Given: 32 bit virtual addresses, 1 GB physical memory
  • Address partition: 20 bit page number, 12 bit offset
Example of Sizing the Page Table

- 20 bits to address $2^{20} = 1$ M entries
- 18 bits to address $2^{30}/2^{12}$ frames
- 4 bytes needed to contain 24 (1+1+1+18+3+...) bits
- Page size = frame size = $2^{12} = 4096$ bytes
- Table size = $1$ M x 4 = 4 MB

- 4 MB of bookkeeping for *every process*?
  - 200 processes -> 800 MB just to store page tables...
Pros and Cons of Paging

• Pro: Fixed-size pages and frames
  • No external fragmentation
  • No difficult placement decisions

• Con: large table size

• Con: *maybe* internal fragmentation
x86: Hybrid Approach

• Design:
  • Multiple lookups: first in segment table, which points to a page table.
  • Extra level of indirection.

• Reality:
  • All segments are max physical memory size
  • Segments effectively unused, available for “legacy” reasons.
  • (Mostly) disappeared in x86-64
Outstanding Problems

• Mostly considering paging from here on.

1. Page tables are way too big. Most processes don’t need that many pages, can’t justify a huge table.

Outstanding Problems

• Mostly considering paging from here on.

1. Page tables are way too big. Most processes don’t need that many pages, can’t justify a huge table.


Solution: MORE indirection!
Multi-Level Page Tables

**Virtual Address**

1st-level Page $d$  |  2nd-level Page $p$  |  Offset $i$

Points to (base) frame containing 2nd-level page table

V | R | M | Frame | ...

concat

Insight: VAS is typically sparsely populated.

Idea: every process gets a page directory (1st-level table)

Only allocate 2nd-level tables when the process is using that VAS region!
Multi-Level Page Tables

Virtual Address

1st-level Page $d$  2nd-level Page $p$  Offset $i$

Points to (base) frame containing 2nd-level page table

Physical Address

Physical Address concatenation

OS
Text
Data
Heap
Stack
Multi-Level Page Tables

• With only a single level, the page table must be large enough for the largest processes.

• Multi-level table => extra level of indirection:
  • WORSE performance – more memory accesses
  • Much better memory efficiency – process’s page table is proportional to how much of the VAS it’s using.

• Small process -> low page table storage
• Large process -> high page table storage, needed it anyway
Translation Cost

• Each application memory access now requires multiple accesses!

• Suppose memory takes 100 ns to access.
  • one-level paging: 200 ns
  • two-level paging: 300 ns

• Solution: Add hardware, take advantage of locality...
  • Most references are to a small number of pages
  • Keep translations of these in high-speed memory
Memory Management Unit

- When a process tries to use memory, send the address to MMU.

- MMU will do as much work as it can. If it knows the answer, great!

- If it doesn’t, trigger exception (OS gets control), consult software table.
Translation Look-aside Buffer (TLB)

- Fast memory mapping cache inside MMU keeps most recent translations
  - If key matches, get frame number quickly
  - Otherwise, wait for normal translation (in parallel)

<table>
<thead>
<tr>
<th>Page p or or [page d, page p] or [segment s, page p]</th>
<th>Offset i</th>
</tr>
</thead>
</table>

```
<table>
<thead>
<tr>
<th>“key”</th>
<th>frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Frame f</th>
<th>Offset i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Recall: Context Switching Performance

• Even though it’s fast, context switching is expensive:
  1. time spent is 100% overhead
  2. must invalidate other processes’ resources (caches, memory mappings)
  3. kernel must execute – it must be accessible in memory

• Also recall: Advantage of threads
  • Threads all share one process VAS
Translation Cost with TLB

• Cost is determined by
  • Speed of memory: ~ 100 nsec
  • Speed of TLB: ~ 10 nsec
  • Hit ratio: fraction of memory references satisfied by TLB, ~95%

• Speed to access memory with no address translation: 100 nsec

• Speed to access memory with address translation (2-level paging):
  • TLB miss: 300 nsec (200% slowdown)
  • TLB hit: 110 nsec (10% slowdown)
  • Average: $110 \times 0.95 + 300 \times 0.05 = 119.5 \text{ nsec}$
TLB Design Issues

• The larger the TLB...
  • the higher the hit rate
  • the slower the response
  • the greater the expense
  • the larger the space (in MMU, on chip)

• TLB has a major effect on performance!
  • Must be flushed on context switches
  • Alternative: tagging entries with PIDs
Summary

• Many options for translation mechanism: segmentation, paging, hybrid, multi-level paging. All of them: level(s) of *indirection*.

• Simplicity of paging makes it most common today.

• Multi-level page tables improve memory efficiency – page table bookkeeping scales with process VAS usage.

• TLB in hardware MMU exploits locality to improve performance