MEMORY: PAGING AND TLBS

Shivaram Venkataraman
CS 537, Spring 2020
Project 2a is out!

Reminder: Midterm 1 makeup

Discussion section: Process API, Project 2a
AGENDA / LEARNING OUTCOMES

Memory virtualization

What is paging and how does it work?
What are some of the challenges in implementing paging?
RECAP
MEMORY VIRTUALIZATION

Transparency: Process is unaware of sharing

Protection: Cannot corrupt OS or other process memory

Efficiency: Do not waste memory or slow down processes

Sharing: Enable sharing between cooperating processes
ABSTRACTION: ADDRESS SPACE

- **Stack Segment**: Contains local variables, arguments to routines, return values, etc.
- **Heap Segment**: Contains malloc'd data, dynamic data structures (it grows downward)
- **Program Code Segment**: Where instructions live

The diagram illustrates the allocation of memory in an address space, with sections allocated to the operating system, processes, and free memory.
Review: Segmentation

0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi

%rip: 0x0010

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000</td>
<td>0xffff</td>
</tr>
<tr>
<td>1</td>
<td>0x5800</td>
<td>0xffff</td>
</tr>
<tr>
<td>2</td>
<td>0x6800</td>
<td>0x7ff</td>
</tr>
</tbody>
</table>

1. Fetch instruction at logical addr 0x0010
   Physical addr:
2. Exec, load from logical addr 0x1100
   Physical addr:
3. Fetch instruction at logical addr 0x0013
   Physical addr:
4. Exec, no load
FRAGMENTATION

Types of fragmentation
- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester

Definition: Free memory that can’t be usefully allocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Not Compacted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0KB</td>
<td>Operating System</td>
</tr>
<tr>
<td>8KB</td>
<td>(not in use)</td>
</tr>
<tr>
<td>16KB</td>
<td>(not in use)</td>
</tr>
<tr>
<td>24KB</td>
<td>Allocated</td>
</tr>
<tr>
<td>32KB</td>
<td>(not in use)</td>
</tr>
<tr>
<td>40KB</td>
<td>Allocated</td>
</tr>
<tr>
<td>48KB</td>
<td>(not in use)</td>
</tr>
<tr>
<td>56KB</td>
<td>Allocated</td>
</tr>
<tr>
<td>64KB</td>
<td>Allocated</td>
</tr>
</tbody>
</table>
PAGING
PAGING

Goal: Eliminate requirement that address space is contiguous
   Eliminate external fragmentation
   Grow segments as needed

Idea:
Divide address spaces and physical memory into fixed-sized pages

Size: $2^n$, Example: 4KB
How to translate logical address to physical address?

- High-order bits of address designate page number
- Low-order bits of address designate offset within page

No addition needed; just append bits correctly!
## ADDRESS FORMAT

Given known page size, how many bits are needed in address to specify offset in page?

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td></td>
</tr>
<tr>
<td>1 KB</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td></td>
</tr>
</tbody>
</table>
Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Total Bits</th>
<th>High Bits (vpn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
Given number of bits for vpn, how many virtual pages can there be in an address space?

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
<th>Virt Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
How should OS translate VPN to PPN?

Number of bits in virtual address need not equal number of bits in physical address.
What is a good data structure?

Simple solution: Linear page table aka *array*
PER-PROCESS PAGETABLE

Virt Mem

Phys Mem

P1

P2

P3
Virt Mem

Phys Mem

Page Tables:

P1

P2

P3
<table>
<thead>
<tr>
<th>Description</th>
<th>Name of approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. one process uses RAM at a time</td>
<td></td>
</tr>
<tr>
<td>2. rewrite code and addresses before running</td>
<td></td>
</tr>
<tr>
<td>3. add per-process starting location to virt addr to obtain phys addr</td>
<td></td>
</tr>
<tr>
<td>4. dynamic approach that verifies address is in valid range</td>
<td></td>
</tr>
<tr>
<td>5. several base+bound pairs per process</td>
<td></td>
</tr>
</tbody>
</table>

Candidates: Segmentation, Static Relocation, Base, Base+Bounds, Time Sharing

https://tinyurl.com/cs537-sp20-quiz9
Consider a 32-bit address space with 4 KB pages. Assume each PTE is 4 bytes.

How many bits do we need to represent the offset within a page?

How many virtual pages will we have in this case?

What will be the overall size of the page table?
WHERE ARE PAGE TABLES STORED?

Implication: Store each page table in memory

   Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

   Change contents of page table base register to newly scheduled process
   Save old page table base register in PCB of descheduled process
What other info is in pagetable entries besides translation?
  – valid bit
  – protection bits
  – present bit (needed later)
  – reference bit (needed later)
  – dirty bit (needed later)

Pagetable entries are just bits stored in memory
  – Agreement between HW and OS about interpretation
MEMORY ACCESSES WITH PAGING

14 bit addresses

0x0010: movl 0x1100, %edi

Assume PT is at phys addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view of page table

| 2 | 0 | 80 | 99 |

Fetch instruction at logical addr 0x0010
Access page table to get ppn for vpn 0
Mem ref 1:
Learn vpn 0 is at ppn ___
Fetch instruction at ______ (Mem ref 2)

Exec, load from logical addr 0x1100
Access page table to get ppn for vpn 1
Mem ref 3:
Learn vpn 1 is at ppn ___
Movl from _____ into reg (Mem ref 4)
MEMORY ACCESSES WITH PAGING

14 bit addresses

0x0010: movl 0x1100, %edi

Assume PT is at phys addr 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view of page table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>80</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fetch instruction at logical addr 0x0010

Access page table to get ppn for vpn 0
Mem ref 1: ___0x5000___
Learn vpn 0 is at ppn 2
Fetch instruction at ___0x2010___ (Mem ref 2)

Exec, load from logical addr 0x1100

Access page table to get ppn for vpn 1
Mem ref 3: ___0x5004___
Learn vpn 1 is at ppn 0
Movl from ___0x0100___ into reg (Mem ref 4)
ADVANTAGES OF PAGING

No external fragmentation
- Any page can be placed in any frame in physical memory

Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn’t have to coalesce with adjacent free space

Simple to swap-out portions of memory to disk (later lecture)
- Page size matches disk block size
- Can run process when some pages are on disk
- Add “present” bit to PTE
Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process
- Wasted memory grows with larger pages
- Tension?

Additional memory reference to page table → Very inefficient
- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial
- Simple page table: Requires PTE for all pages in address space
  Entry needed even if page not allocated?
SUMMARY: PAGE TRANSLATION STEPS

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory into register

Which steps are expensive?
Example: Array Iterator

```c
int sum = 0;
for (i=0; i<N; i++) {
    sum += a[i];
}
```

What virtual addresses?
- load 0x3000
- load 0x3004
- load 0x3008
- load 0x300C

What physical addresses?
- load 0x100C
- load 0x7000
- load 0x100C
- load 0x7004
- load 0x100C
- load 0x7008
- load 0x100C
- load 0x700C

Assume ‘a’ starts at 0x3000
Ignore instruction fetches and access to ‘i’
STRATEGY: CACHE PAGE TRANSLATIONS

- CPU
  - Translation Cache

- RAM
  - PT

Memory interconnect
TLB: TRANSLATION LOOKASIDE BUFFER
## TLB Organization

<table>
<thead>
<tr>
<th>Tag (virtual page number)</th>
<th>Physical page number (page table entry)</th>
</tr>
</thead>
</table>

### TLB Entry

- **Fully associative**
- Any given translation can be anywhere in the TLB
- Hardware will search the entire TLB in parallel
int sum = 0;
for (i = 0; i < 2048; i++){
    sum += a[i];
}

Assume following virtual address stream:
load 0x1000
load 0x1004
load 0x1008
load 0x100C
...

Assume ‘a’ starts at 0x1000
Ignore instruction fetches
and access to ‘i’

What will TLB behavior look like?
TLB ACCESSES: SEQUENTIAL EXAMPLE

Virt

0x1000
0x1004
0x1008
0x100c
...
0x2000
0x2004
**TLB ACCESSES: SEQUENTIAL EXAMPLE**

<table>
<thead>
<tr>
<th>0 KB</th>
<th>PT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 KB</td>
<td>PT</td>
</tr>
<tr>
<td>8 KB</td>
<td>P1</td>
</tr>
<tr>
<td>12 KB</td>
<td>P2</td>
</tr>
<tr>
<td>16 KB</td>
<td>P2</td>
</tr>
<tr>
<td>20 KB</td>
<td>P1</td>
</tr>
<tr>
<td>24 KB</td>
<td>P1</td>
</tr>
<tr>
<td>28 KB</td>
<td>P2</td>
</tr>
</tbody>
</table>

**PTBR**

**P1 pagetable**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**CPU's TLB**

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**Virt**

- load 0x1000
- load 0x1004
- load 0x1008
- load 0x100c
- ... (TLB hit)
- load 0x2000
- load 0x2004

**Phys**

- load 0x0004
- load 0x5000 (TLB hit)
- load 0x5004 (TLB hit)
- load 0x5008 (TLB hit)
- load 0x500C
- ... (TLB hit)
- load 0x0008 (TLB hit)
- load 0x4000
- load 0x4004
Consider a processor with 16-bit address space and 4kB page size. Assume Page Table is at 0x2000 and each PTE is of 4 bytes.

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Virtual Addresses
- 0x3000: load 0x5320, %eax
- 0x3004: load 0x4004, %ebx
- 0x3008: mul %ecx, %eax, %ebx
- 0x300C: store %ebx, 0x5324
- 0x3010: load 0x5328, %ebx

Total number of memory accesses
### Simplified view of the PT

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Memory accesses

- Virtual Addresses
  - 0x3000: load 0x5320, %eax
  - 0x3004: load 0x4004, %ebx
  - 0x3008: mul %ecx, %eax, %ebx
  - 0x300C: store %ebx, 0x5324
  - 0x3010: load 0x5328, %ebx

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>89</td>
</tr>
</tbody>
</table>
Performance of TLB?

Miss rate of TLB: \(#\text{TLB misses} / \#\text{TLB lookups}\)

\(#\text{TLB lookups}\)? number of accesses to \(a\) = 2048

\(#\text{TLB misses}\)?

\[\text{number of unique pages accessed} = \frac{2048}{(\text{elements of } a \text{ per 4K page})} = \frac{2K}{(4K/\text{sizeof} (\text{int}))} = \frac{2K}{1K} = 2\]

Miss rate? \(= \frac{2}{2048} = 0.1\%\)

Hit rate? \((1 - \text{miss rate}) = 99.9\%\)

Would hit rate get better or worse with smaller pages?
How can system improve hit rate given fixed number of TLB entries?

Increase page size:
  Fewer unique page translations needed to access same amount of memory

TLB Reach: Number of TLB entries * Page Size
Workload Access Patterns

**Workload A**

```c
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Sequential array accesses almost always hit in TLB!

**Workload B**

```c
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```
WORKLOAD ACCESS PATTERNS

Spatial Locality
Sequential Accesses

Temporal Locality
Repeated Random Accesses
WORKLOAD LOCALITY

**Spatial Locality**: future access will be to nearby addresses

**Temporal Locality**: future access will be repeats to the same data

What TLB characteristics are best for each type?

**Spatial**:
- Access same page repeatedly; need same vpn $\rightarrow$ ppn translation
- Same TLB entry re-used

**Temporal**:
- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?
OTHER TLB CHALLENGES

How to replace TLB entries? LRU? Random?

TLB on context switches? HW or OS?
Project 2a is out!

Discussion today: Process API, Project 2a

Next class: More TLBs and better pagetables!