MEMORY: PAGING AND TLBS

Shivaram Venkataraman
CS 537, Spring 2019
- Project 1b is due Friday
- Project 1a grades are out
- Project 2a going out tomorrow

- 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

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

<table>
<thead>
<tr>
<th>0KB</th>
<th>1KB</th>
<th>2KB</th>
<th>16KB</th>
<th>15KB</th>
<th>320KB</th>
<th>384KB</th>
<th>448KB</th>
<th>512KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(free)</td>
<td>(free)</td>
<td>(free)</td>
<td>(free)</td>
<td>(free)</td>
<td>Process A (code, data, etc.)</td>
<td>(free)</td>
<td>(free)</td>
<td>(free)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Process B (code, data, etc.)</td>
<td>(free)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Process C (code, data, etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Operating System (code, data, etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The above diagram illustrates the allocation of address space in a computer system.
SEGMENTATION IMPLEMENTATION

MMU contains Segment Table (per process)
- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments;

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bounds</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x2000</td>
<td>0x6ff</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xffff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

How many bits for segment?
How many bits for offset?

Remember: 1 hex digit à 4 bits
**Quiz: Address Translations with Segmentation**

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bounds</th>
<th>R W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x2000</td>
<td>0x6ff</td>
<td>1 0</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>1 1</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xfff</td>
<td>1 1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x000</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Remember:
1 hex digit → 4 bits

Translate logical (in hex) to physical:

- 0x0240:
- 0x1108:
- 0x265c:
- 0x3002:
**REVIEW: MEMORY ACCESSES**

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

0x0010: \texttt{movl \ 0x1100, \%edi}

0x0013: \texttt{addl \ $0x3, \%edi}

0x0019: \texttt{movl \ \%edi, \ 0x1100}

%rip: 0x0010

1. Fetch instruction at logical addr 0x0010
   - Physical addr: 0x4010
2. Exec, load from logical addr 0x1100
   - Physical addr: 0x5900
3. Fetch instruction at logical addr 0x0013
   - Physical addr: 0x4013
4. Exec, no load
5. Fetch instruction at logical addr 0x0019
   - Physical addr: 0x4019
6. Exec, store to logical addr 0x1100
   - Physical addr: 0x5900
ADVANTAGES OF SEGMENTATION

Enables sparse allocation of address space
Stack and heap can grow independently

• Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
• Stack: OS recognizes reference outside legal segment, extends stack implicitly

Different protection for different segments
• Enables sharing of selected segments
• Read-only status for code

Supports dynamic relocation of each segment
Disadvantages of Segmentation

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation
<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
PAGING
FRAGMENTATION

Definition: Free memory that can’t be usefully allocated

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Status</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0KB</td>
<td>Not Compacted</td>
<td></td>
</tr>
<tr>
<td>8KB</td>
<td>Operating System</td>
<td></td>
</tr>
<tr>
<td>16KB</td>
<td>(not in use)</td>
<td></td>
</tr>
<tr>
<td>24KB</td>
<td>Allocated</td>
<td></td>
</tr>
<tr>
<td>32KB</td>
<td>(not in use)</td>
<td></td>
</tr>
<tr>
<td>40KB</td>
<td>Allocated</td>
<td></td>
</tr>
<tr>
<td>48KB</td>
<td>(not in use)</td>
<td></td>
</tr>
<tr>
<td>56KB</td>
<td>Allocated</td>
<td></td>
</tr>
<tr>
<td>64KB</td>
<td>Allocated</td>
<td></td>
</tr>
</tbody>
</table>
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…
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>
## ADDRESS FORMAT

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 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>
### ADDRESS FORMAT

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 PAGE TABLE

Virt Mem

Phys Mem

P1

P2

P3
FILL IN PAGETABLE

Page Tables:

Virt Mem

Phys Mem

0 1 2 3 4 5 6 7 8 9 10 11
QUIZ: HOW BIG IS A PAGETABLE?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries
WHERE ARE PAGETABLES 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
OTHER PAGETABLE INFO

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

Fetch instruction at logical addr 0x0010

- Access page table to get ppn for vpn 0
- Mem ref 1:
  - Learn vpn 0 is at ppn 2
  - 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 0
  - Movl from ______ into reg (Mem ref 4)
SUMMARY: PAGING

Page table:

3 7 5 2

Page frame 0 of physical memory

Page frame 1

Page 3 of AS

Page frame 2

Page 0 of AS

Page frame 3

Page frame 4

Page 2 of AS

Page frame 5

Page frame 6

Page frame 7
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 \(\rightarrow\) 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?
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 expensive step will we avoid next?
### 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

Diagram showing the connection between CPU and RAM through a translation cache and memory interconnect.
TLB: TRANSLATION LOOKASIDE BUFFER
### TLB Organization

**TLB Entry**

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

- Any given translation can be anywhere in the TLB.
- Hardware will search the entire TLB in parallel.

**Fully associative**
```c
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 ACCSESSES: SEQUENTIAL EXAMPLE

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>

PTBR

PT page table

0 KB
PT
4 KB
PT
8 KB
P1
12 KB
P2
16 KB
P2
20 KB
P1
24 KB
P1
28 KB
P2
TLB ACCESSES: SEQUENTIAL EXAMPLE

**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>

**PTBR**

**PI page table**

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

**Virt**

- load 0x1000
- load 0x1004
- load 0x1008
- load 0x100c
- load 0x2000
- load 0x2004

**Phys**

- load 0x0004
- load 0x5000 (TLB hit)
- load 0x5004 (TLB hit)
- load 0x5008 (TLB hit)
- load 0x500C
- load 0x0008 (TLB hit)
- load 0x4000 (TLB hit)
- load 0x4004
Performance of TLB?

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

Miss rate of TLB: \#TLB misses / \#TLB lookups

\#TLB lookups? number of accesses to a = 2048

\#TLB misses?
    = number of unique pages accessed
    = 2048 / (elements of ‘a’ per 4K page)
    = 2K / (4K / sizeof(int)) = 2K / 1K
    = 2

Miss rate? = 2/2048 = 0.1%

Hit rate? (1 – miss rate) = 99.9%

Would hit rate get better or worse with smaller pages?
TLB PERFORMANCE

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
Sequential array accesses almost always hit in TLB
   – Very fast!
What access pattern will be slow?
   – Highly random, with no repeat accesses
Workload Access Patterns

**Workload A**

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

**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?
NEXT STEPS

Project 1b: Due tomorrow!
Project 2a: Out tomorrow

Discussion today: Process API, Project 2a

Next class: More TLBs and better pagetables!