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

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

The figure illustrates the address space layout of a computer system, showing the stack, heap, and program code segments. The diagram also indicates how different processes (A, B, C) are allocated space within the address space, with operating system code/data also present.

- Operating System (code, data, etc.): free
- Process C (code, data, etc.): free
- Process B (code, data, etc.): free
- Process A (code, data, etc.): free
- Free space (0KB to 512KB)
**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>RW</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>0xffff</td>
<td>1 1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x000</td>
<td>0 0</td>
</tr>
</tbody>
</table>
### Quiz: Address Translations with Segmentation

#### 14 bit = 3 hex bits are 12 bit

Top hex bit is 2 bits

<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>0xffff</td>
<td>1 1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x000</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Translate logical (in hex) to physical

- **0x0240:**
  \[ 0x2000 + 0x0240 = 0x2240 \]

- **0x1108:**
  \[ 0x3000 + 0x0108 = 0x3108 \]

- **0x265c:**
  \[ 0x3000 + 0x065c = 0x365c \]

- **0x3002:**
  \[ 0x0000 + 0x0002 = 0x0002 \]
  **FAIL**

#### 1 hex digit $\rightarrow$ 4 bits

- $0x1001$ $fff$
- $max$ $addr$ $0x3fff$
**REVIEW: MEMORY ACCSESSES**

0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100

%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>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>

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>Size</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0KB</td>
<td>Not Compacted</td>
</tr>
<tr>
<td>8KB</td>
<td>Operating System</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>

![Fragmentation Diagram]

*Note: Diagram shows a segment of memory with a 12KB free space and 4KB used space.*
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>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10 ((2^{10} = 1 \text{ KB}))</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
<tr>
<td>(\log_2(512)) bytes</td>
<td>9</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</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>6</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</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>$2^6 = 64$</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>$2^{10} = 1024$</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td>$2^{12} = 4096$</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>7</td>
<td>$2^7 = 128$</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
<td>$2^{20}$</td>
</tr>
</tbody>
</table>

$2^{12} = 4096$
How should OS translate VPN to PPN?

Number of bits in virtual address need not equal number of bits in physical address.

VPN

offset

0 1

0 1 0 1

Addr Mapper

PPN

offset

1 0 1 1

0 1 0 1

64 bit

2^64 Virtual address
What is a good data structure?

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

Virt Mem

Phys Mem

P1

P2

P3
Page Tables:
How big is a typical page table?
- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

\[
\text{Page table size} = \text{12 bit offset} = \text{Number of entries} \times \text{size entry}
\]

Number of entries = 20 bits of VPN

= 2\(^{20}\) virtual pages

= 1 MB of virtual page

= 4 bytes

= 4 MB
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 PAGENTABLE 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
MEMORY ACCESSES WITH PAGING

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

14 bit addresses

Fetch instruction at logical addr 0x0010
- Access page table to get ppn for vpn 0
- Mem ref 1: To the page table 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: 0x5000
- Learn vpn 1 is at ppn 0
- Movl from 0x0100 into reg (Mem ref 4)
<table>
<thead>
<tr>
<th>Page Frame</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Page frame 0 of physical memory</td>
</tr>
<tr>
<td>1</td>
<td>Page frame 1</td>
</tr>
<tr>
<td>2</td>
<td>Page frame 2</td>
</tr>
<tr>
<td>3</td>
<td>Page frame 3</td>
</tr>
<tr>
<td>4</td>
<td>Page frame 4</td>
</tr>
<tr>
<td>5</td>
<td>Page frame 5</td>
</tr>
<tr>
<td>6</td>
<td>Page frame 6</td>
</tr>
<tr>
<td>7</td>
<td>Page frame 7</td>
</tr>
</tbody>
</table>

**Page Table:**

- Page 0 of AS
- Page 3 of AS
- (Unused)
- Page 0 of AS
- Page 2 of AS
- (Unused)
- Page 1 of AS
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
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?
PAGING 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 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
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>

A B C D E L M N O P

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
...

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

PTBR

PI pagetable

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>
**TLB ACCESSES: SEQUENTIAL EXAMPLE**

**PTBR**

**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
- 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
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?
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 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
**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 → 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!