Memory

CS 241
February 1, 2012

Slides adapted in part from material by Matt Welsh, Harvard U.
Announcements

- MP2 released
- Brighten’s office hours this week
  - Wednesday 3-4
  - Thursday 3-4
- Talk today: Nick Feamster, Georgia Tech

“The Battle for Control of Online Communications”

4:00 p.m.
2405 Siebel Center
Recap: Virtual Addresses

- A **virtual address** is a memory address that a process uses to access its own memory
  - Virtual address ≠ actual physical RAM address
  - When a process accesses a virtual address, the MMU hardware **translates** the virtual address into a physical address
  - The OS determines the mapping from virtual address to physical address

- **Benefit: Isolation**
  - Virtual addresses in one process refer to **different** physical memory than virtual addresses in another
  - Exception: shared memory regions between processes (discussed later)

- **Benefit: Illusion of larger memory space**
  - Can store unused parts of virtual memory on disk temporarily

- **Benefit: Relocation**
  - A program does not need to know which physical addresses it will use when it’s run
  - Can even change physical location while program is running
Mapping virtual to physical addresses

(Reserved for OS)

Stack

Heap

Uninitialized vars (BSS segment)

Initialized vars (data segment)

Code (text segment)

Physical RAM

How does this thing work??
**MMU and TLB**

- **Memory Management Unit (MMU)**
  - Hardware that translates a virtual address to a physical address
  - Each memory reference is passed through the MMU
  - Translate a virtual address to a physical address
    - Lots of ways of doing this!

- **Translation Lookaside Buffer (TLB)**
  - Cache for MMU virtual-to-physical address translations
  - Just an optimization – but an important one!

![Diagram of MMU and TLB](image.png)
Translating virtual to physical

- Can do it almost any way we like
- But, some ways are better than others…

- Strawman solution from last time: base and bound
Base and bound

```java
if (virt addr > bound)
  trap to kernel
else
  phys addr = virt addr + base
```

- Process has the illusion of running on its own dedicated machine with memory [0,bound)
- Provides protection from other processes also currently in memory
Base and bound

CPU Address

Memory Address MA

Logical Address LA

Fault

<

Base Address BA

Base Register

Bounds Register

Base Address

MA+BA Memory

Bound Address

Physical Address PA

Fault

Base: start of the process’s memory partition
Bound: length of the process’s memory partition

Copyright ©: University of Illinois CS 241 Staff
Base and bounds

- Problem: Process needs more memory over time
  - Stack grows as functions are called
  - Heap grows upon request (malloc)
  - Processes start and end

- How does the kernel handle the address space growing?
  - You are the OS designer
  - Design strategy for allowing processes to grow
But wait, didn’t we solve this?

Problem: wasted space

- And must have virtual mem $\leq$ phys mem
Another attempt: segmentation

- Segment
  - Region of contiguous memory

- Segmentation
  - Generalized base and bounds with support for multiple segments at once
Segmentation

<table>
<thead>
<tr>
<th>Seg #</th>
<th>Base</th>
<th>Bound</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4000</td>
<td>700</td>
<td>Code segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>Data segment</td>
</tr>
<tr>
<td>2</td>
<td>Unused</td>
<td></td>
<td>Stack segment</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>1000</td>
<td>Stack segment</td>
</tr>
</tbody>
</table>
Segmentation

- Segments are specified in many different ways
- Advantages over base and bounds?
- Protection
  - Different segments can have different protections
- Flexibility
  - Can separately grow both a stack and heap
  - Enables sharing of code and other segments if needed
Segmentation

- Segments are specified many different ways
- What are the advantages over base and bounds?
- What must be changed on context switch?
  - Contents of your segmentation table
  - A pointer to the table, expose caching semantics to the software (what x86 does)
Recap: mapping virtual memory

- **Base & bounds**
  - Problem: growth is inflexible
  - Problem: external fragmentation
    - As jobs run and complete, holes left in physical memory

- **Segments**
  - Resize pieces based on process needs
  - Problem: external fragmentation
  - Note: x86 used to support segmentation, now effectively deprecated with x86-64

- **Modern approach: Paging**
Paging
Paging

- Solve the external fragmentation problem by using **fixed-size chunks** of virtual and physical memory
  - Virtual memory unit called a **page**
  - Physical memory unit called a **frame** (or sometimes **page frame**)

![Diagram of paging with virtual memory and physical memory]

- virtual memory (for one process)
  - page 0
  - page 1
  - page 2
  - page 3
  - page X

- physical memory
  - frame 0
  - frame 1
  - frame 2
  - frame Y
Application Perspective

- Application believes it has a single, contiguous address space ranging from 0 to $2^P - 1$ bytes
  - Where $P$ is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under its control!
Translation process

- Virtual-to-physical address translation performed by MMU
  - Virtual address is broken into a virtual page number and an offset
  - Mapping from virtual page to physical frame provided by a page table (which is stored in memory)

0xdeadbeef = 0xdeadb 0xeef

<table>
<thead>
<tr>
<th>virtual address</th>
<th>virtual page #</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xdeadb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>page frame 0</td>
</tr>
<tr>
<td>page frame 1</td>
</tr>
<tr>
<td>page frame 2</td>
</tr>
<tr>
<td>page frame 3</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>page frame Y</td>
</tr>
</tbody>
</table>

Page table entry
Translation process

if (virtual page is invalid or non-resident or protected)
    trap to OS fault handler
else
    physical frame # = pageTable[virtpage#].physPageNum

- Each virtual page can be in physical memory or swapped out to disk (called “paged out” or just “paged”)
- What must change on a context switch?
  - Could copy entire contents of table, but this will be slow
  - Instead use an extra layer of indirection: Keep pointer to current page table and just change pointer
Where is the page table?

- Page Tables store the virtual-to-physical address mappings.
- Where are they located? *In memory!*
- OK, then. How does the MMU access them?
  - The MMU has a special register called the *page table base pointer*.
  - This points to the *physical memory address* of the top of the page table for the currently-running process.
Page Faults

- What happens when a program accesses a virtual page that is not mapped into any physical page?
  - Hardware triggers a page fault

- Page fault handler
  - Find any available free physical page
  - If none, evict some resident page to disk
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table
Advantages of Paging

- Simplifies physical memory management
  - OS maintains a free list of physical page frames
  - To allocate a physical page, just remove an entry from this list

- No external fragmentation!
  - Virtual pages from different processes can be interspersed in physical memory
  - No need to allocate pages in a contiguous fashion

- Allocation of memory can be performed at a (relatively) fine granularity
  - Only allocate physical memory to those parts of the address space that require it
  - Can swap unused pages out to disk when physical memory is running low
  - Idle programs won't use up a lot of memory (even if their address space is huge!)
Paging Example

Request Address within Virtual Memory Page 3

Virtual Memory Stored on Disk

Page Table
VM Frame
1 3
2 1
3 2
4 3

Real Memory

Disk

Copyright ©: University of Illinois CS 241 Staff
Paging Example

Request Address within Virtual Memory: Page 1

Cache

Virtual Memory Stored on Disk

Page Table
VM Frame

Real Memory

Disk

Copyright ©: University of Illinois CS 241 Staff
Paging Example

Request Address within Virtual Memory Page 6

Virtual Memory Stored on Disk

Page Table VM Frame

Real Memory

Disk
Paging Example

Request Address within Virtual Memory

Page 2

Cache

1 2 3 4

Virtual Memory Stored on Disk

1 2 3 4 5 6 7 8

Page Table

VM Frame

3 1
1 2
6 3
2 4

Real Memory

1 2 3 4

Disk

Copyright ©: University of Illinois CS 241 Staff
What happens when there is no more space in the cache?
Paging Example

Store Virtual Memory Page 1 to disk

Virtual Memory Stored on Disk

Cache

Page Table
VM Frame

Real Memory

Disk
Paging Example

Process request for Address within Virtual Memory Page 8

Cache

Virtual Memory Stored on Disk

Page Table
VM Frame

Real Memory

Disk
Paging Example

Load Virtual Memory
Page 8 to cache

Virtual Memory Stored on Disk

Disk

Cache

Page Table
VM Frame

Real Memory

Copyright ©: University of Illinois CS 241 Staff
Is paging enough?

How do we allocate memory in here?

- Stack
- Heap
- Uninitialized vars (BSS segment)
- Initialized vars (data segment)
- Code (text segment)

MMU

Physical RAM
Memory allocation w/in a process

- What happens when you declare a variable?
  - Allocating a page for every variable wouldn’t be efficient
  - Allocations within a process are much smaller
  - Need to allocate on a finer granularity

- Solution (stack): stack data structure (duh)
  - Function calls follow LIFO semantics
  - So we can use a stack data structure to represent the process’s stack – no fragmentation!

- Solution (heap): `malloc`
  - This is a much harder problem
  - Need to deal with fragmentation
MP2: malloc

- Introduction by Wade