CS 333
Introduction to Operating Systems

Class 9 - Memory Management

Jonathan Walpole
Computer Science
Portland State University
Memory management

- **Memory** - a linear array of bytes
  - Holds O.S. and programs (processes)
  - Each cell (byte) is named by a unique memory address
- **Recall**, processes are defined by an *address space*, consisting of text, data, and stack regions
- **Process execution**
  - CPU fetches instructions from the text region according to the value of the program counter (PC)
  - Each instruction may request additional operands from the data or stack region
Addressing memory

- Cannot know ahead of time where in memory a program will be loaded!

- Compiler produces code containing embedded addresses
  - these addresses can’t be absolute (physical addresses)

- Linker combines pieces of the program
  - Assumes the program will be loaded at address 0

- We need to **bind** the compiler/linker generated addresses to the actual memory locations
Relocatable address generation

Compilation
Assembly
Linking
Loading
Address binding

- **Address binding**
  - fixing a physical address to the logical address of a process’ address space

- **Compile time binding**
  - if program location is fixed and known ahead of time

- **Load time binding**
  - if program location in memory is unknown until run-time AND location is fixed

- **Execution time binding**
  - if processes can be moved in memory during execution
  - Requires hardware support!
Runtime binding – base & limit registers

- Simple runtime relocation scheme
  - Use 2 registers to describe a partition

- For every address generated, at runtime…
  - Compare to the limit register (& abort if larger)
  - Add to the base register to give physical memory address
Dynamic relocation with a base register

- Memory Management Unit (MMU) - dynamically converts logical addresses into physical address
- MMU contains base address register for running process

![Diagram showing dynamic relocation with a base register](image-url)
Protection using base & limit registers

- Memory protection
  - Base register gives starting address for process
  - Limit register limits the offset accessible from the relocation register

```
+-----------------+-------------------+------+
| logical address | limit register     | base |
|                 | yes               | register |
|                 | < yes             | > no   |
| no              | < < no            | yes   |
| addressing error| no yes            | < yes |
```

Multiprogramming with base and limit registers

- **Multiprogramming**: a separate partition per process

- **What happens on a context switch?**
  - Store process A’s **base** and **limit** register values
  - Load new values into **base** and **limit** registers for process B

```
OS
Partition A
Partition B
Partition C
Partition D
Partition E
```

- limit
- base
Swapping

- **When a program is running...**
  - The entire program must be in memory
  - Each program is put into a single partition

- **When the program is not running...**
  - May remain resident in memory
  - May get “swapped” out to disk

- **Over time...**
  - Programs come into memory when they get swapped in
  - Programs leave memory when they get swapped out
Basics - swapping

- Benefits of swapping:
  - Allows multiple programs to be run concurrently
  - ... more than will fit in memory at once
Swapping can lead to fragmentation
Dealing with fragmentation

- **Compaction** - from time to time shift processes around to collect all free space into one contiguous block
  - Memory to memory copying overhead
  - memory to disk to memory for compaction via swapping
How big should partitions be?

- Programs may want to grow during execution
  - More room for stack, heap allocation, etc

- Problem:
  - If the partition is too small programs must be moved
  - Requires copying overhead
  - Why not make the partitions a little larger than necessary to accommodate “some” cheap growth?
Allocating extra space within partitions

(a) Room for growth
(b) Room for growth

Operating system

B

Actually in use

Room for growth

A

Actually in use

Room for growth

B-Stack

B-Data

B-Program

A-Stack

A-Data

A-Program

Operating system
Managing memory

- Each chunk of memory is either
  - Used by some process or unused ("free")

- Operations
  - Allocate a chunk of unused memory big enough to hold a new process
  - Free a chunk of memory by returning it to the free pool after a process terminates or is swapped out
Managing memory with bit maps

- Problem - how to keep track of used and unused memory?
- Technique 1 - Bit Maps
  - A long bit string
  - One bit for every chunk of memory
    - $1 = \text{in use}$
    - $0 = \text{free}$
  - Size of allocation unit influences space required
    - Example: unit size = 32 bits
      - overhead for bit map: $1/33 = 3\%$
    - Example: unit size = 4Kbytes
      - overhead for bit map: $1/32,769$
Managing memory with bit maps

Diagram showing memory allocation with bit maps.
Managing memory with linked lists

- Technique 2 - Linked List
  - Keep a list of elements
  - Each element describes one unit of memory
    - Free / in-use Bit ("P=process, H=hole")
    - Starting address
    - Length
    - Pointer to next element
Managing memory with linked lists
Merging holes

- Whenever a unit of memory is freed we want to merge adjacent holes!
Merging holes

Before X terminates

A X B

becomes

After X terminates

A .......... B
Merging holes

Before X terminates

A X B

A X

becomes

becomes

After X terminates

A

A

B
Merging holes

Before X terminates

A  X  B

becomes

A  X  B

becomes

X  B

becomes

A  B

After X terminates
Merging holes

Before X terminates

A  X  B

A  X

X  B

X

becomes

becomes

becomes

becomes

After X terminates

A  [X]  B

A

[ ]

[ ]

[ ]

[ ]
Managing memory with linked lists

- Searching the list for space for a new process
  - First Fit
  - Next Fit
    - Start from current location in the list
  - Best Fit
    - Find the smallest hole that will work
    - Tends to create lots of really small holes
  - Worst Fit
    - Find the largest hole
    - Remainder will be big
  - Quick Fit
    - Keep separate lists for common sizes
Fragmentation

- Memory is divided into partitions
- Each partition has a different size
- Processes are allocated space and later freed
- After a while memory will be full of small holes!
  - No free space large enough for a new process even though there is enough free memory in total
- If we allow free space within a partition we have internal fragmentation
- Fragmentation:
  - External fragmentation = unused space between partitions
  - Internal fragmentation = unused space within partitions
Solution to fragmentation?

- Compaction requires high copying overhead
- Why not allocate memory in non-contiguous equal fixed size units?
  - no external fragmentation!
  - internal fragmentation < 1 unit per process
- How big should the units be?
  - The smaller the better for internal fragmentation
  - The larger the better for management overhead
- The key challenge for this approach:
  “How can we do secure dynamic address translation?”
Using pages for non-contiguous allocation

- Memory divided into fixed size page frames
  - Page frame size = $2^n$ bytes
  - Lowest $n$ bits of an address specify byte offset in a page

- But how do we associate page frames with processes?
  - And how do we map memory addresses within a process to the correct memory byte in a page frame?

- Solution – address translation
  - Processes use virtual addresses
  - CPU uses physical addresses
  - Hardware support for virtual to physical address translation
Virtual addresses

- Virtual memory addresses (what the process uses)
  - Page number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page number

Example: 32 bit virtual address
Page size = $2^{12} = 4\text{KB}$
Address space size = $2^{32}$ bytes = 4GB
Physical addresses

- Physical memory addresses (what the CPU uses)
  - Page “frame” number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the frame number

Example: 24 bit physical address
Frame size = $2^{12} = 4$KB
Max physical memory size = $2^{24}$ bytes = 16MB
Address translation

- Hardware maps page numbers to frame numbers
- Memory management unit (MMU) has multiple registers for multiple pages
  - Like a base register except its value is substituted for the page number rather than added to it
  - Why don't we need a limit register for each page?
Memory Management Unit (MMU)

The CPU sends virtual addresses to the MMU

The MMU sends physical addresses to the memory
Virtual address spaces

- Here is the virtual address space
  - (as seen by the process)
Virtual address spaces

- The address space is divided into “pages”
  - In BLITZ, the page size is 8K
Virtual address spaces

- In reality, only some of the pages are used
Physical memory

- Physical memory is divided into "page frames"
  - (Page size = frame size)
Virtual and physical address spaces

- Some frames are used to hold the pages of this process

![Diagram showing virtual address space and physical memory with some frames highlighted for this process.](image-url)
Virtual and physical address spaces

- Some frames are used for other processes

![Diagram of virtual and physical address spaces]

- Virtual Addr Space
- Physical memory

- Used by other processes
Virtual address spaces

- Address mappings say which frame has which page
Page tables

- Address mappings are stored in a page table in memory
- One page table entry per page...
  - Is this page in memory? If so, which frame is it in?

Virtual Addr Space

Physical memory
Address mappings and translation

- **Address mappings** are stored in a **page table** in memory
  - Typically one page table for each process

- **Address translation** is done by **hardware** (i.e., the MMU)

- How does the MMU get the address mappings?
  - Either the MMU holds the entire page table (too expensive)
    - or it knows where it is in physical memory and goes there for every translation (too slow)
  - Or the MMU holds a portion of the page table
    - **MMU caches** page table entries
    - Cache is called a translation look-aside buffer (**TLB**)
Address mappings and translation

- What if the TLB needs a mapping it doesn’t have?

  - **Software managed TLB**
    - it generates a **TLB-miss fault** which is handled by the operating system (like interrupt or trap handling)
    - The operating system looks in the page tables, gets the mapping from the right entry, and puts it in the TLB

  - **Hardware managed TLB**
    - it looks in a pre-specified physical memory location for the appropriate entry in the page table
    - The hardware architecture defines where page tables must be stored in physical memory
      - **OS must load current process page table there on context switch!**
The BLITZ architecture

- **Page size**
  - 8 Kbytes

- **Virtual addresses ("logical addresses")**
  - 24 bits  -->  16 Mbyte virtual address space
  - $2^{11}$ Pages  -->  11 bits for page number
The BLITZ architecture

- **Page size**
  - 8 Kbytes

- **Virtual addresses ("logical addresses")**
  - 24 bits → 16 Mbyte virtual address space
  - $2^{11}$ Pages → 11 bits for page number

- **An address:**

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>23</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11 bits</td>
<td>13 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

page number  offset
The BLITZ architecture

- **Physical addresses**
  - 32 bits $\rightarrow$ 4 Gbyte installed memory (max)
  - $2^{19}$ Frames $\rightarrow$ 19 bits for frame number
The BLITZ architecture

- Physical addresses
  - 32 bits --> 4 Gbyte installed memory (max)
  - \(2^{19}\) Frames --> 19 bits for frame number
The BLITZ architecture

- The page table mapping:
  - Page --> Frame

- Virtual Address:

- Physical Address:
The BLITZ page table

- An array of "page table entries"
  - Kept in memory

- $2^{11}$ pages in a virtual address space?
  - $\rightarrow$ 2K entries in the table

- Each entry is 4 bytes long
  - 19 bits The Frame Number
  - 1 bit Valid Bit
  - 1 bit Writable Bit
  - 1 bit Dirty Bit
  - 1 bit Referenced Bit
  - 9 bits Unused (and available for OS algorithms)
The BLITZ page table

- Two page table related registers in the CPU
  - Page Table Base Register
  - Page Table Length Register

- These define the “current” page table
  - This is how the CPU knows which page table to use
  - Must be saved and restored on context switch
  - They are essentially the Blitz MMU

- Bits in the CPU “status register”
  - “System Mode”
  - “Interrupts Enabled”
  - “Paging Enabled”
    - 1 = Perform page table translation for every memory access
    - 0 = Do not do translation
The BLITZ page table

- Frame number
- 19 bits
- Unused
- Dirty bit
- Referenced bit
- Writable bit
- Valid bit
The BLITZ page table

Indexed by the page number

<table>
<thead>
<tr>
<th>31</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>1</td>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>2</td>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>2K</td>
<td>frame number</td>
<td>unused</td>
</tr>
</tbody>
</table>
The BLITZ page table

<table>
<thead>
<tr>
<th>frame number</th>
<th>unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

page table base register

virtual address
The BLITZ page table

<table>
<thead>
<tr>
<th>23</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frame number</th>
<th>unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

page table base register

virtual address

physical address
The BLITZ page table

![Diagram of the BLITZ page table]

- **Page Table Base Register**: 31 bits
- **Virtual Address**: 31 bits (31:0)
  - **Page Number** (23:0)
  - **Offset** (13:12)
- **Physical Address**: 31 bits (31:0)
  - **Offset** (13:0)

---

<table>
<thead>
<tr>
<th>Frame Number</th>
<th>Unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The BLITZ page table

<table>
<thead>
<tr>
<th>frame number</th>
<th>unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page table base register

Virtual address

Physical address

Offset
The BLITZ page table

- Page number
- Offset
- Frame number
- Unused
- D R W V

Page table base register

Virtual address

Physical address
Quiz

- What is the difference between a virtual and a physical address?
- What is address binding?
- Why are programs not usually written using physical addresses?
- Why is hardware support required for dynamic address translation?
- What is a page table used for?
- What is a TLB used for?
- How many address bits are used for the page offset in a system with 2KB page size?