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
<table>
<thead>
<tr>
<th>Compilation</th>
<th>Assembly</th>
<th>Linking</th>
<th>Loading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog P</td>
<td>foo()</td>
<td>foo()</td>
<td>foo()</td>
</tr>
<tr>
<td></td>
<td>push ...</td>
<td>push ...</td>
<td>push ...</td>
</tr>
<tr>
<td></td>
<td>jmp _foo</td>
<td>jmp 75</td>
<td>jmp 175</td>
</tr>
<tr>
<td></td>
<td>foo: ...</td>
<td>foo: ...</td>
<td>foo: ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>75</td>
<td>175</td>
</tr>
<tr>
<td>P:</td>
<td></td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>P:</td>
<td></td>
<td>100</td>
<td>1100</td>
</tr>
<tr>
<td>P:</td>
<td></td>
<td>1100</td>
<td>1175</td>
</tr>
<tr>
<td>Library</td>
<td>Library</td>
<td>Library</td>
<td>Library</td>
</tr>
<tr>
<td>Routines</td>
<td>Routines</td>
<td>Routines</td>
<td>Routines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>push ...</td>
<td>push ...</td>
<td>push ...</td>
</tr>
<tr>
<td></td>
<td>jmp 1175</td>
<td>jmp 1175</td>
<td>jmp 1175</td>
</tr>
<tr>
<td></td>
<td>foo: ...</td>
<td>foo: ...</td>
<td>foo: ...</td>
</tr>
<tr>
<td></td>
<td>1175</td>
<td>1175</td>
<td>1175</td>
</tr>
</tbody>
</table>
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!
Compile Time Address Binding

Load Time Address Binding

Execution Time Address Binding

Base register

1000

Library Routines

P:
  
  push ...
  jmp 175

foo: ...

175

100

0

Library Routines

P:
  
  push ...
  jmp 1175

1175

1100

1000

Library Routines

P:
  
  push ...
  jmp 1175

foo: ...

1175

1000

Library Routines

P:
  
  push ...
  jmp 175

foo: ...

175

100

0

Library Routines

P:
  
  push ...
  jmp 1175

foo: ...

1175

1100

1000

Library Routines

P:
  
  push ...
  jmp 175

foo: ...

175

100

0

Library Routines

P:
  
  push ...
  jmp 1175

foo: ...

1175

1100

1000

Library Routines

P:
  
  push ...
  jmp 175

foo: ...

175

100

0

Library Routines

P:
  
  push ...
  jmp 1175

foo: ...

1175

1100

1000

Library Routines

P:
  
  push ...
  jmp 175

foo: ...

175

100

0

Library Routines

P:
  
  push ...
  jmp 1175

foo: ...

1175

1100

1000

Library Routines

P:
  
  push ...
  jmp 175

foo: ...

175

100

0

Library Routines

P:
  
  push ...
  jmp 1175

foo: ...

1175

1100

1000
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**

Relocation register for process $i$

Program generated address

[Diagram showing MMU, physical memory, max address, max mem, process $i$, and operating system with arrows and values]

- $1000$ as an example of a base register value
- Physical memory address
- Max Mem
- Max addr
- Operating system
Protection using base & limit registers

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

![Diagram of memory protection using base & limit registers]

- Logical address
- Base register
- Limit register
- Physical address
- Memory
- Addressing error
- Yes
- No
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
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
896K

576K

320K

P1

128K

128K
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
Actually in use
Room for growth
Actually in use
Operating system

(b) 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 = in use
    - 0 = 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 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

```
0  8  16  24  32
A   B   C   D   E
```

```
P 0  5 ----> H 5  3 ----> P 8  6 ----> P 14  4

H 18  2 ----> P 20  6 ----> P 26  3 ----> H 29  3 X
```

- Hole starts at 18
- Length 2
- Process
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 [X] B
Merging holes

Before $X$ terminates

\[
\begin{array}{|c|c|c|}
\hline
A & X & B \\
\hline
\end{array}
\]

becomes

\[
\begin{array}{|c|c|c|}
\hline
A & X & \text{ } \\
\hline
\end{array}
\]

becomes

\[
\begin{array}{|c|c|c|}
\hline
A & \text{ } & B \\
\hline
\end{array}
\]

After $X$ terminates
# Merging holes

<table>
<thead>
<tr>
<th>Before X terminates</th>
<th>becomes</th>
<th>After X terminates</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>B</td>
</tr>
</tbody>
</table>
Merging holes

Before X terminates  |  After X terminates

A  X  B  |  A  B

A  X  |  A

X  B  |  B

X  |  X
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
    - Not as good as first fit
  - Best Fit
    - Find the smallest hole that will work
    - Tends to create lots of little 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 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
  - 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 = $2^{12} = 4$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 = $2^{12} = 4KB$
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.

Diagram:
- CPU
- Memory Management Unit
- Memory
- Disk Controller
- Bus
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 and physical address spaces](image)
Virtual and physical address spaces

- Some frames are used for other processes
Virtual address spaces

- Address mappings say which frame has which page
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?
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**)
    - ... and knows how to deal with TLB misses
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:**

```
     23 13 0
  11 bits      13 bits
  page number  offset
```
The BLITZ architecture

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

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

![Address Diagram]

- 31
- 19 bits
- 13 bits
- 0

- **Frame number**
- **Offset**
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

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>...</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dirty bit</td>
<td>referenced bit</td>
<td>writable bit</td>
<td>valid bit</td>
</tr>
</tbody>
</table>
### The BLITZ page table

Indexed by the page number

<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>page table base register</th>
<th>virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1312</td>
</tr>
<tr>
<td>0</td>
<td>frame number</td>
</tr>
<tr>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>1</td>
<td>frame number</td>
</tr>
<tr>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>2</td>
<td>frame number</td>
</tr>
<tr>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>2K</td>
<td>frame number</td>
</tr>
<tr>
<td></td>
<td>unused</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>1312</td>
</tr>
<tr>
<td>0</td>
<td>0</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**: 23 1312 0
- **Page Number**
- **Offset**

**Physical Address**

- **Virtual Address**: 31 0
The BLITZ page table

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

Page table base register

Virtual address

Physical address
The BLITZ page table

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2K</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>

physical address

virtual address

page table base register
The BLITZ page table

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2K</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
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?