CS399 New Beginnings

Jonathan Walpole
Memory Management
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

Prog P
: foo():
: End P

P:
: push ...
: jmp _foo
: foo: ...

P: 75
: push ...
: jmp 75
: foo: ...

P: 175
: push ...
: jmp 175
: foo: ...

P: 1175
: push ...
: jmp 1175
: foo: ...

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!
Base and 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

Memory Management Unit (MMU)
- Dynamically converts logical to physical address
- Contains base address register for running process

Diagram:
- Program generated address
- Relocation register for process $i$
- MMU
- Physical memory address
- Max Mem
- Max addr

- Process $i$
- Operating system
Protection

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

<table>
<thead>
<tr>
<th>Logical address</th>
<th>Base register</th>
<th>Limit register</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

addressing error
Multiprogramming

Multiprogramming: a separate partition per process
What happens on a context switch?
Store process base and limit register values
Load new values into base and limit registers
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
Swapping

Benefits of swapping:
Allows multiple programs to be run concurrently
... more than will fit in memory at once
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

(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
Management Data Structures

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
Management 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
Management With Bit Maps

(a) Diagram showing bit maps with labels and arrows indicating different processes and holes.

(b) Close-up view of specific bit maps with labels indicating hole starts at 18 and length 2.

(c) Continuation of processes and holes with label 29.
Management 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
Management With Linked Lists
Management 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 Revisited

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 fragmentation
  External fragmentation = unused space between partitions
  Internal fragmentation = unused space within partitions
Solutions 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?
Non-Contiguous Allocation (Pages)

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 = $2^{12} = 4KB$
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.
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 & Physical Address Spaces

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

![Diagram showing virtual and physical address spaces with some frames marked as used for the process.]

- **Virtual Addr Space**
- **Physical memory**

These frames are used for this process.
Virtual & Physical Address Spaces

Some frames are used for other processes

Virtual Addr Space

Physical memory

Used by other processes
Virtual & Physical Address Spaces

Address mappings say which frame has which page

Virtual Addr Space  Physical memory
Page Tables

Address mappings are stored in a *page table* in memory
1 entry/page: is page in memory? If so, which frame is it in?

<table>
<thead>
<tr>
<th>Virtual Addr Space</th>
<th>Physical memory</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>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
Address Mappings

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

Address translation is done by hardware (ie 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 and knows how to deal with TLB misses
  - MMU caches page table entries
  - Cache is called a translation look-aside buffer (TLB)
Address Mappings & TLB

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: 
\[ \begin{array}{c}
23 & 13 & 12 & 0 \\
\end{array} \]

- 11 bits for page number
- 13 bits for 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 --> 4 Gbyte installed memory (max)
$2^{19}$ Frames --> 19 bits for frame number
The BLITZ Architecture

The page table mapping:
Page --> Frame

Virtual Address: 23 13 12 0
11 bits

Physical Address: 31 13 12 0
19 bits
The BLITZ Page Table

An array of “page table entries”
    Kept in memory

$2^{11}$ pages in a virtual address space?
    ---> 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
- Unused
- Dirty bit (D)
- Referenced bit (R)
-Writable bit (W)
- Valid bit (V)

19 bits
## The BLITZ Page Table

The table is indexed by the page number and contains the following fields:

- **frame number**
- **used**
- **D**
- **R**
- **W**
- **V**

The table is structured as follows:

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Used</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>
<td></td>
</tr>
<tr>
<td>1</td>
<td></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>
<td></td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indexed by the page number.
The BLITZ Page Table

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>1312</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
# The BLITZ Page 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>

## Virtual Address

**Frame 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>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
</tbody>
</table>

## Physical Address

<table>
<thead>
<tr>
<th>physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
The BLITZ Page Table

- Page number
- Offset
- Frame number
- Unused
- DRWV

Virtual address

Physical address

Page table base register
# The BLITZ Page Table

<table>
<thead>
<tr>
<th>Page Table Base Register</th>
<th>Virtual Address</th>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>frame number</td>
<td>unused</td>
<td>D R W V</td>
</tr>
<tr>
<td>1</td>
<td>frame number</td>
<td>unused</td>
<td>D R W V</td>
</tr>
<tr>
<td>2</td>
<td>frame number</td>
<td>unused</td>
<td>D R W V</td>
</tr>
<tr>
<td>2K</td>
<td>frame number</td>
<td>unused</td>
<td>D R W V</td>
</tr>
</tbody>
</table>

Physical Address

```plaintext
31 1312 0
```
The BLITZ Page Table

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

<table>
<thead>
<tr>
<th>physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 frame number</td>
</tr>
<tr>
<td>1312 offset</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
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?