CS510 Operating System Foundations
Jonathan Walpole
Memory Management
Memory Management

Memory – a linear array of bits, bytes, words, pages ...
- Each byte is named by a unique memory address
- Holds instructions and data for OS and user processes

Each process has an *address space* containing its instructions, data, heap and stack regions

When processes execute, they use addresses to refer to things in their memory (instructions, variables etc)

... But how do they know which addresses to use?
Addressing Memory

Cannot know ahead of time where in memory instructions and data will be loaded!
- so we can’t hard code the addresses in the program code

Compiler produces code containing names for things, but these names can’t be physical memory addresses

Linker combines pieces of the program from different files, and must resolve names, but still can’t encode addresses

We need to bind the compiler/linker generated names to the actual memory locations before, or during execution
Binding Example

Prog P
::
foo()
::
End P

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

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

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

0 75 175

0 100 175

Library Routines

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

1000 1100 1175

Library Routines

Compilation Assembly Linking Loading
Relocatable Addresses

How can we execute the same processes in different locations in memory without changing memory addresses?

How can we move processes around in memory during execution without breaking their addresses?
Simple Idea: Base/Limit Registers

Simple runtime relocation scheme
- Use 2 registers to describe a processes memory partition
- Do memory addressing indirectly via these registers

For every address, before going to memory ...
- Add to the base register to give physical memory address
- Compare result to the limit register (& abort if larger)
Dynamic Relocation via Base Register

Memory Management Unit (MMU)
- Dynamically converts re-locatable logical addresses to physical addresses

Relocation register for process \(i\)

Program generated address

\[ + \]

MMU

Max Mem

Physical memory address

Operating system

Max addr

0

0
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

![Diagram showing partitions and base/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 why keep it in memory?
- Could swap it out to disk to make room for other processes

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
- How much stack memory do we need?
- How much heap memory do we need?

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?
- … but that is just a different kind of fragmentation
Allocating Extra Space Within

(a) Room for growth
Actually in use
Room for growth
Actually in use
Operating system

(b) B-Stack
B-Data
B-Program
Room for growth
A-Stack
A-Data
A-Program
Room for growth
Operating system
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
What Causes These Problems?

Contiguous allocation per process leads to fragmentation, or high compaction costs.

Contiguous allocation is necessary if we use a single base register - ... because it applies the same offset to all memory addresses.
Non-Contiguous Allocation

Why not allocate memory in non-contiguous fixed size pages?
- Benefit: no external fragmentation!
- Internal fragmentation < 1 page per process region

How big should the pages be?
- The smaller the better for internal fragmentation
- The larger the better for management overhead (i.e. bitmap size required to keep track of free pages)

The key challenge for this approach
How can we do secure dynamic address translation?
I.e., how do we keep track of where things are?
Paged Virtual Memory

Memory divided into fixed size page frames
- Page frame size $= 2^n$ bytes
- $n$ low-order bits of address specify byte offset in a page
- remaining bits specify the page number

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 physical page frame?

Solution – per-process page table for 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 = $2^{12} = 4$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 offsets for multiple pages, i.e., a page table
- Like a base register except each entries value is substituted for the page number rather than added to it
- Why don’t we need a limit register for each page?
- Typically called a translation look-aside buffer (TLB)
MMU / TLB

The CPU sends virtual addresses to the MMU

The MMU sends physical addresses to the memory

CPU package

Memory management unit

Memory

Disk controller

Bus
Virtual Address Spaces

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

Lowest address

Highest address

Virtual Addr Space
Virtual Address Spaces

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

Virtual Addr Space

Page 0

Page 1

A Page

Page N
Virtual Address Spaces

In reality, only some of the pages are used

Virtual Addr Space

Unused

N
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.

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
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?
Address Mappings

Address mappings are stored in a page table in memory
  - one page table for each process because each process has its own independent address space

Address translation is done by hardware (ie the TLB ... translation-look-aside buffer)

How does the TLB get the address mappings?
  - Either the TLB 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 TLB holds a portion of the page table and knows how to deal with TLB misses
    - the TLB caches page table entries
Two Types of 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, perhaps replacing an existing entry

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 loads current process page table there on context switch!
The BLITZ Memory 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 Memory 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 12 0
```

11 bits 13 bits

page number offset
The BLITZ Memory Architecture

Physical addresses

32 bits --> 4 Gbyte installed memory (max)

2^{19} Frames --> 19 bits for frame number
The BLITZ Memory Architecture

Physical addresses

32 bits $\rightarrow$ 4 Gbyte installed memory (max)

$2^{19}$ Frames $\rightarrow$ 19 bits for frame number
The BLITZ Memory Architecture

The page table mapping:
Page --> Frame

Virtual Address: 23 13 12
11 bits

Physical Address: 31 13 12
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
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>

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>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>

- **Page Table Base Register**: The page table base register is used to store the base address of the page table in memory.
- **Virtual Address**: The virtual address is the address used by the CPU to reference memory.
- **Physical Address**: The physical address is the actual memory location where the data is stored.
The BLITZ Page Table

![Diagram of the BLITZ Page Table]

- Page table base register
- Virtual address
- Physical address

The diagram illustrates the structure of the BLITZ Page Table, showing how page numbers and offsets are used to map virtual addresses to physical addresses. Each entry in the table contains a frame number, an unused flag, and permissions (D, R, W, V) for direct, read, write, and validity.
The BLITZ Page Table

<table>
<thead>
<tr>
<th>Page Table Base Register</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td>offset</td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>frame number</td>
<td>unused</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>

<table>
<thead>
<tr>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
</tr>
</tbody>
</table>

- 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>0</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>2K</td>
<td>23</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>

Virtual Address

Physical Address

Page Table Base Register
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?
Spare Slides
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

Diagram (a) shows a bit map with labeled segments A through E, each containing a binary sequence. Diagram (b) illustrates a process flow starting at 18, with a length of 2, indicated by an arrow pointing to another process. Diagram (c) connects these processes, showing the flow from one to another.
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

(a) 

(b) 

(c) 

Hole Starts Length Processes
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