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
  - Hold O.S. and programs (processes)
  - Each memory cell 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

Prog P
::
foo()
::
End P

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

0

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

75

100

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

175

0

Library Routines

1000

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

1100

Library Routines

1175
Address binding

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

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

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

- **Execution time**
  - 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

Relocation register for process $i$

Program generated address

$1000$

+$\quad$ MMU

Physical memory address

$0$

Max Mem

Max addr

Operating system

$0$

process $i$
Protection using base & limit registers

- Memory protection
  - Base register gives starting address for process
  - Limit register limits the offset accessible from the relocation register
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
128K O.S. 896K
128K O.S. 576K 320K
128K O.S. 352K 224K
128K O.S. 288K 320K
128K O.S. 224K 224K
128K O.S. 64K
Dealing with fragmentation

- **Compaction** - from time to time shift processes around to collect all free space into one contiguous block

- Placement algorithms: First-fit, best-fit, worst-fit
Influence of allocation policy

**FIRST-FIT**

1. Scan
2. Compact

**BEST-FIT**
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 modification of base and limit regs
  - Why not make the partitions a little larger than necessary to accommodate “some” growth?

- Fragmentation:
  - **External fragmentation** = unused space between partitions
  - **Internal fragmentation** = unused space within partitions
Allocating extra space within partitions
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
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  \[\text{shaded}\]  B
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></td>
</tr>
<tr>
<td>A</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Merging holes

Before X terminates

A  X  B
becomes

A  X  
becomes

X  B  
becomes

After X terminates

A  
becomes

A

X

B
Merging holes

<table>
<thead>
<tr>
<th>Before X terminates</th>
<th>After X terminates</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>B</td>
</tr>
<tr>
<td>X</td>
<td>becomes</td>
</tr>
</tbody>
</table>
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
  - This is external fragmentation
- If we allow free space within a partition we have internal fragmentation
Solution to fragmentation?

- Allocate memory in equal fixed size units?
  - No external fragmentation problems
  - But what about wasted space inside a unit due to internal fragmentation?

- How big should the units be?
  - The smaller the better for internal fragmentation
  - The larger the better for management overhead

- Can we use a unit size smaller than the memory needed by a process?
  - Allocate non-contiguous units to the same process?
  - ... but how would the base and limit registers work?
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 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

32 bit virtual address
Page size $= 2^{12} = 4$KB
Address space size $= 2^{32}$ bytes $= 4$GB
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 page frame number

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

- Hardware maps page numbers to page frame numbers
- Memory management unit (MMU) has a register for each page
  - 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 SPANK, 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 page frames are used to hold the pages of this process
Virtual and physical address spaces

- Some page frames are used for other processes

```
0 1 2 3 4 5 6 7
N

Virtual Addr Space

Used by other processes

Physical memory
```
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?
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 the MMU holds a portion of the page table
    - **MMU caches** of page table entries
    - 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 memory location for the appropriate entry in the page table
    - The hardware architecture defines where page tables must be stored in memory
The SPANK architecture

- Page size
  - 8 Kbytes

- Virtual addresses ("logical addresses")
  - 24 bits → 16 Mbyte virtual address space
  - 2K Pages → 11 bits
The SPANK architecture

- Page size
  - 8 Kbytes

- Virtual addresses ("logical addresses")
  - 24 bits → 16 Mbyte virtual address space
  - 2K Pages → 11 bits

- An address:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
   | 11 bits |
   | 13 bits |
   |   0     |
```

- Page number
- Offset
The SPANK architecture

- **Physical addresses**
  - 32 bits --> 4 Gbyte installed memory (max)
  - 512K Frames --> 19 bits
The SPANK architecture

- Physical addresses
  - 32 bits --> 4 Gbyte installed memory (max)
  - 512K Frames --> 19 bits
The SPANK architecture

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

- Virtual Address:

- Physical Address:
The SPANK page table

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

- 2K 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)
Example from the textbook
The SPANK page table

- **Two registers in the CPU**
  - Page Table Base Register
  - Page Table Length Register

- **These define the “current” page table.**
  - Must be saved and restored on context switch

- **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 SPANK page table

- 31
- 13 12
- 0

frame number

unused
D R W V

19 bits

dirty bit
referenced bit
writable bit
valid bit
The SPANK page table

Indexed by the page 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></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 SPANK 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>

- **Frame number**
- **D**: Deny
- **R**: Read
- **W**: Write
- **V**: Valid

**Page Table Base Register**

<table>
<thead>
<tr>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2K</td>
</tr>
</tbody>
</table>

**Virtual Address**
The SPANK 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>

<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>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Page Table Base Register**

**Virtual Address**

**Physical Address**
The SPANK page table

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>012</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>31</td>
<td>1312</td>
</tr>
<tr>
<td>0</td>
<td>2K</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
The SPANK page table

---

**page table base register**

**virtual address**

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

---

**physical address**

---

23 13 12 0

---

**page number**

---

offset
The SPANK 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></td>
<td>physical address</td>
</tr>
<tr>
<td>23</td>
<td>1312</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>page number</td>
<td>unused</td>
</tr>
<tr>
<td>0</td>
<td>D R W V</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>1</td>
<td>D R W V</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>2</td>
<td>D R W V</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>2K</td>
<td>D R W V</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>frame number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>offset</td>
</tr>
<tr>
<td>1312</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Diagram shows the mapping between virtual and physical addresses through the page 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?