Memory Management
Big Picture

- We want to avoid a fixed one to one mapping from program memory addresses to physical memory addresses.
  - Logical vs. physical addresses.
- First we will look at some historical mechanisms. Then modern approaches.
- Memory management involves two kinds of “tricks” one handled in chapter 8, one in chapter 9.
- Hardware details will be important.
Reminders

- Program must be brought from disk to memory and placed within a process for it to run.
- Main memory and registers are only storage CPU can access directly.
- Register access in one CPU clock (or less).
- Main memory can take many cycles
- Cache sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.
Address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile time**
  - When might this make sense?

- **Load time**
  - At load time, directly edit every memory location in the program.
  - Any problems?

- **Execution time**
  - Handle binding on the fly.
Simple Memory Protection: Base and Limit Registers

- Every memory access is checked against base and limit registers.
- OS loads these registers when a process is dispatched.
Simple Address Translation: Relocation Register
Linking/Loading/Compiling
Dynamic Loading

• Routine is not loaded until it is called.
• Better memory-space utilization; unused routine is never loaded.
• Useful when large amounts of code are needed to handle infrequently occurring cases.
• Not used much, we'll see better mechanisms later.
Dynamic Linking

- Linking postponed until execution time.
- Small piece of code, stub, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Dynamic linking is particularly useful for libraries.
- System also known as shared libraries.
- Do you see any challenges?
- More on sharing when we talk about paging.
Supporting Multiprogramming: Swapping

• A process can be swapped temporarily out of memory and then brought back into memory for continued execution.

• Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

• Modified versions of swapping are found on some systems.
  – Remember medium term schedulers?
Mapping Processes to Memory: Contiguous Allocation

• Partition main memory into two regions:
  – Resident operating system, usually held in low memory with interrupt vector.
  – User processes held in high memory.

• Each process is granted exclusive access to a contiguous region of memory.

• Base and limit registers used for dynamic memory mapping and memory protection.
Contiguous Allocation

- OS needs to track free regions of memory: holes.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
Dynamic Storage Allocation Problem

• Where to put new processes?
  – **First-fit**: Allocate the first hole that is big enough.
  – **Best-fit**: Allocate the smallest hole that is big enough.
    • Produces the smallest leftover hole.
  – **Worst-fit**: Allocate the largest hole.
    • Produces the largest leftover hole.

• First-fit and best-fit generally better than worst-fit in terms of fragmentation...
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory.

- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.

- I/O problem
  - Latch job in memory while it is involved in I/O.
  - Do I/O only into OS buffers.
Paging

- Address space of a process can be noncontiguous.
  - Process is allocated physical memory wherever it is available.
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes).
- Divide logical memory into blocks of same size called pages.
- To run a program of size $n$ pages, need to find $n$ free frames.
- Set up a page table to translate logical to physical addresses.
Logical vs. Physical Memory

<table>
<thead>
<tr>
<th>Logical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
</tr>
<tr>
<td>page 1</td>
</tr>
<tr>
<td>page 2</td>
</tr>
<tr>
<td>page 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

page table:
- 0: 1
- 1: 4
- 2: 3
- 3: 7

<table>
<thead>
<tr>
<th>physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
</tr>
<tr>
<td>page 1</td>
</tr>
<tr>
<td>page 2</td>
</tr>
<tr>
<td>page 3</td>
</tr>
</tbody>
</table>
Address Translation

● Address generated by CPU is divided into:
  
  - **Page number** \((p)\) – used as an index into a page table which contains base address of each page in physical memory.
  
  - **Page offset** \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit.

\[
\begin{array}{c|c}
\text{page number} & \text{page offset} \\
p & d \\
\hline
m - n & n
\end{array}
\]

- For given logical address space \(2^m\) and page size \(2^n\).
Tiny Page Table Example

• 4 byte pages.
• 32 bytes total.
• # bits in page number?
• # bits in offset?
Page Table Implementation

- Page table is kept in main memory.
- **Page-table base register (PTBR)** points to the page table.
- **Page-table length register (PRLR)** indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
Page Table Hardware

- This picture is a bit deceptive...

- How can we avoid doubling # of memory accesses?
Translation Look-Aside Buffer

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers (TLB's)**.
- Some TLB's store **address-space identifiers (ASID's)** in each TLB entry.
  - Why?
**Effective Access Time**

- Hit rate for TLB can have a large impact on memory performance.
- Associative Lookup time \(= \epsilon\).
- Assume memory cycle time \(= \mu\).
- Hit ratio – percentage of times that a page number is found in the associative registers.
- Hit ratio \(= \alpha\).
- Effective Access Time:

\[
(\mu + \epsilon)\alpha + (2\mu + \epsilon)(1 - \alpha) = 2\mu + \epsilon - \mu \alpha
\]
Memory Protection

- Memory protection implemented by associating protection bit(s) with each frame in the page table.
  - Read/write, read only, executable...

- Valid-invalid bit attached to each entry in the page table:
  - valid - the page is in the process’ logical address space.
  - invalid - the page is not in the process’ logical address space.
Shared Pages

- Page tables provide a simple mechanism for implementing shared memory.
- Page table entries for different processes refer to the same physical frames.
Problem With Basic Page Table Scheme...

- Assume page size of 4KB.
  - (page offset of 12 bits.)
- In a 32 bit address space we will have $2^{32}/2^{12} = 2^{20}$ entries in the page table.
- Assuming each entry is 4 bytes, this is about 4MB.
Solution 1: Multi-Level Page Tables

- Page the page table.
- Here is a two level table.
- Also possible to have three or more levels.
- What is the advantage?
**Two Level Paging Example**

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
• This is all great. What will happen when we try to tackle a 64 bit address?
3-Level Page Table?

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$d$</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Hashed Page Table

- The virtual page number is hashed into a page table.
- Page table contains chains of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Hashed Page Table
Inverted Page Table

- Fundamental inefficiency:
  - Many processes, each with a logical address space potentially larger than the physical address space.
  - Many big page tables are required to manage all of those logical address spaces.

- Why not just have one page table that maps from physical frames to processes and logical addresses?
Inverted Page Tables

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- TLB will help...
- Use hash table to limit the search to one, or at most a few, page-table entries.
- Any implications for sharing?
Inverted Page Table
Segmentation

- Wait. Take a step back.
- We have been assuming that all processes want to live their lives in single, linear, logical address space.
- It might be nice to have lots of address spaces:
  - main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays.
- Could potentially make compilation linking and loading simpler.
User's View of Program
Incidentally, what problem again rears its ugly head?
Segmentation Architecture

• Logical address consists of a two tuple:
  – <segment-number, offset>

• Segment table – each table entry has:
  – base – contains the starting physical address.
  – limit – specifies the length of the segment.

• Segment-table base register (STBR) points to the segment table’s location in memory.

• Segment-table length register (STLR) indicates number of segments used by a program.
Segmentation Architecture

- Protection:
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
  - Protection bits associated with segments; code sharing occurs at segment level.
  - It's all a bit like paging with variable sized pages.
Segmentation Hardware
Segmentation Example
Example: Intel Pentium

- Supports segmentation AND paging.
- Big picture:
  - Logical address generated by CPU consists of
    - 16 bit segment selector,
    - and 32 bit segment offset.
  - Segmentation unit produces 32 bit linear address.
  - Linear address is given to paging unit.
  - Paging unit produces physical address.
- Does that mean the Pentium has a 48 bit address space?
Pentium Segmentation Unit

- Selector:
  - 13 bits for segment id.
  - 1 bit indicating whether it is a local or global segment.
  - 2 protection bits.
Pentium Paging

- Supports 4KB or 4MB pages.
How Is That All Used By the OS?

- We may hear something about that later.
- Most OS's mostly ignore the fancy segment features.
- Just stick everything in a single segment.
Acknowledgments

• Portions of these slides are taken from PowerPoint presentations made available along with:

• Original versions of those presentations can be found at: