Reminder: Thread Synch project due on Monday

Case study guidelines posted

Case study random draw for topics

Questions?
Outline

- Segmentation
- Combining paging and segmentation
- Chapter 9 - Dynamic, partial, non-contiguous storage organization
- Demand paging
  - Address translation
Last time looked at storage organization that includes providing support for:

- **complete** vs. **partial** allocation
- **fixed-size** vs. **variable-size** allocation
- contiguous vs. **fragmented** allocation
- **static** vs. dynamic allocation of partitions

Now look at effect of variable-size allocation
Segmentation

- Paging makes a very clear distinction between the user's logical view of memory and the actual physical memory. But generally, users tend to think of memory in segments related to the purpose or part of a program rather than as a linear array of bytes. E.g., objects, functions, global data, etc.

- Users do not care where in memory each segment is in relation to each other, and the segments are variable in length.
Segmentation

- This idea can be used as a storage organization and management scheme called *segmentation*. Logical address space is a *collection of segments*. Each segment has a "name" and a length.

- Logical addresses are of the form <s, d>, where s is the segment's name (usually a segment number) and d is the displacement.

- The MMU consists of a segment table (ST) of <base address, limit> pairs.
Address Translation

- **CPU**
  - log. addr.
  - s d
  - base limit
  - b l

- **d < l**
  - yes
  - trap invalid address

- **main memory**
  - b
  - d

- **main memory**
  - b
  - d
Segmentation Example

<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
<th>Segment Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Func</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>main</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>symbol table</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base Addr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4700</td>
</tr>
</tbody>
</table>

- Translate the following addresses:
  - \(< 2, 53 >\)
  - \(< 3, 852 >\)
  - \(< 0, 1222 >\)
Segmentation

- Implementation of segment table is similar to a page table. If small, stored directly in hardware. If large, stored in memory. Use TLB to cache mappings.

- Also need a *segment table length* register to check if segment number is valid.
Combining Paging and Segmentation

- Often systems combine paging and segmentation. The most common way is to divide (fairly large) segments into pages.

- Example: Multics - 36-bit word, 34-bit addresses divided into 18-bit segment number and 16-bit displacement. Maximum size segment is 64KB, rather large. Divide the logical address displacement into pages:

```
| s | dp | dd |
```
Combining Paging and Segmentation

- The pages are 1KB, so dp is 6 bits, and dd is 10 bits.
- The ST entries contain a pointer to the segment's PT rather than a base address.
- 18 bits of segment number is large, too, so the ST is paged as well.

```
| sp | sd | dp | dd |
```
Combining Paging and Segmentation

- Address translation is now
  - sp indexes ST page table to get appropriate page of the ST
  - sd indexes ST page to get the PT of segment s
  - dp indexes the PT of segment s to get the appropriate page of physical memory
  - dd is the displacement into the page
Combining Paging and Segmentation

- Example: Intel Pentium - supports pure segmentation and segmentation with paging
- CPU generates 48-bit \(<s, d>\) logical addresses, where \(s\) is 16 bits and \(d\) is 32 bits, that are translated into 32-bit linear addresses.

- \(s\) is used to index the ST to get the base address of a segment, then the 32 bit displacement is added to create a 32-bit linear address. These are then paged in either 4KB or 4MB pages.
Combining Paging and Segmentation

- Example: Linux on Pentium - Linux is designed to be multi-platform, so cannot depend on any particular hardware support. On a Pentium, Linux had 6 segments:
  - Kernel code
  - Kernel data
  - User code - shared by all user processes
  - User data - shared by all user processes
  - Task-state segment (TSS) - stores contexts for switches
  - Default local descriptor table (LDT) segment - generally unused
Combining Paging and Segmentation

- Effectively, each segment is the physical memory being managed for its category.
- Linux is designed for 3 levels of paging, since it can run on 64-bit architectures. The number of bits for each level depends on the architecture. Since a Pentium is a 32-bit architecture, only needed two levels, so the middle level has 0 bits, effectively bypassing it.
Partial allocation goes hand in hand with dynamic allocation, so we look at them together:

- complete vs. **partial** allocation
- **fixed-size** vs. variable-size allocation
- contiguous vs. **fragmented** allocation
- static vs. **dynamic** allocation of partitions

Start looking at effects on storage organization and management. This part of the design space is **virtual memory**.
Partial Allocation

- So far, all schemes considered have static, complete allocation. When a process is admitted, its entire logical address space is loaded into physical memory.

- What happens if we allow partial allocation? Partial allocation necessarily implies dynamic allocation (during run-time), so there must be a mechanism for identifying the logical addresses that are not yet present in physical memory and loading them from the backing store.
Partial Allocation

- Why would partial allocation be a good idea?

- Given that now some accesses will cause a disk read, why should we expect this to work at all?
Virtual memory (VM) commonly is implemented using a **demand paging** technique. The idea is fairly simple, only bring a logical page of a process into memory when it is used.

As with complete allocation organizations, a process's logical address space is loaded onto a **backing store** (also called **swap space**) in a contiguous manner to make loading into memory easier. Backing store usually is a disk.
Demand Paging

- The PT is modified to have a **valid/invalid bit** in each entry to indicate whether the page is in memory.
- If the entry is **valid**, it contains the physical frame number as usual.
- If the entry is **invalid**, it contains the disk address on the backing store that holds the page.
Address Translation

- Address translation must now handle the case where the logical page is not in memory:
  - Page number $p$ is obtained from logical address
  - If TLB hit, access memory
  - If TLB miss, access PT
    - If PT entry is valid, access memory
    - If PT entry is invalid, trap to OS and process goes to the Wait Queue. This is called a *page fault*. 
Address Translation

- Page fault processing consists of
  - Issuing a disk transfer request to the backing store
  - Loading the requested page into a free frame in physical memory
  - Updating the page table entry with valid bit and frame number
- OS issues an Event completion interrupt and process goes to the Ready Queue to wait for CPU. Then it attempts the same access that caused the page fault.
Address Translation

CPU → log. addr. → TLB → phys. addr. → main memory

- TLB hit: valid PT entry
- TLB miss: OS trap
  - page fault
  - update page table
  - backing store
  - transfer page from disk to memory
  - I/O completion interrupt
Effective Memory Access Time

- What is the effect of partial allocation on performance? Could be very bad.
- Start next class looking at effective memory access time.