Today’s class

- Operating System Machine Level
Operating System Machine

Level 3: Operating system machine level
Level 2: Instruction set architecture level
Level 1: Microarchitecture level
Paging

- A mapping in which virtual addresses 4096 to 8191 are mapped onto main memory addresses 0 to 4095.
Implementation of Paging

- Virtual address space divided into a number of equal-sized pages
- Main memory is divided the same way
- This figure shows 4KB pages for the first 64KB of virtual memory

<table>
<thead>
<tr>
<th>Page</th>
<th>Virtual addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 – 4095</td>
</tr>
<tr>
<td>1</td>
<td>4096 – 8191</td>
</tr>
<tr>
<td>2</td>
<td>8192 – 12287</td>
</tr>
<tr>
<td>3</td>
<td>12288 – 16383</td>
</tr>
<tr>
<td>4</td>
<td>16384 – 20479</td>
</tr>
<tr>
<td>5</td>
<td>20480 – 24575</td>
</tr>
<tr>
<td>6</td>
<td>24576 – 28671</td>
</tr>
<tr>
<td>7</td>
<td>28672 – 32767</td>
</tr>
<tr>
<td>8</td>
<td>32768 – 36863</td>
</tr>
<tr>
<td>9</td>
<td>36864 – 40959</td>
</tr>
<tr>
<td>10</td>
<td>40960 – 45055</td>
</tr>
<tr>
<td>11</td>
<td>45056 – 49151</td>
</tr>
<tr>
<td>12</td>
<td>49152 – 53247</td>
</tr>
<tr>
<td>13</td>
<td>53248 – 57343</td>
</tr>
<tr>
<td>14</td>
<td>57344 – 61439</td>
</tr>
<tr>
<td>15</td>
<td>61440 – 65535</td>
</tr>
</tbody>
</table>
Implementation of Paging

- Physical memory is divided into page frames, each frame capable of holding one page of virtual memory.
- This shows 32KB of physical memory divided into 4KB page frames.

<table>
<thead>
<tr>
<th>Page frame</th>
<th>Bottom 32K of main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>28672 – 32767</td>
</tr>
<tr>
<td>6</td>
<td>24576 – 28671</td>
</tr>
<tr>
<td>5</td>
<td>20480 – 24575</td>
</tr>
<tr>
<td>4</td>
<td>16384 – 20479</td>
</tr>
<tr>
<td>3</td>
<td>12288 – 16383</td>
</tr>
<tr>
<td>2</td>
<td>8192 – 12287</td>
</tr>
<tr>
<td>1</td>
<td>4096 – 8191</td>
</tr>
<tr>
<td>0</td>
<td>0 – 4095</td>
</tr>
</tbody>
</table>
Implementation of Paging

- Memory Management Unit handles translation of virtual address to physical address
Demand Paging

- Not all virtual pages will be in main memory at the same time.
- When a reference is made to a virtual page that is not in main memory you generate a page fault.
- The operating system must retrieve the desired page from disk and place it in physical memory.
Working Set

- The set of pages that a program is actively and heavily using is called the working set
- Ideally want the working set kept in main memory to reduce page faults
Page Replacement Policy

- When a page fault is generated a page of virtual memory must be brought into main memory.
- If main memory is full a page from main memory must be written back to disk (if it was changed) or otherwise removed to make room for the new page.
- How the page to remove is chosen is known as the page replacement policy.
Least Recently Used (LRU)

- Remove the page from memory that was used the longest ago
- Works well if the working set fits in main memory
- If it doesn’t, bad things can happen, as shown here: the working set contains a loop that covers 9 pages of memory, but room for only 8 in physical memory
First-In, First-Out (FIFO)

- Removes the page loaded into memory longest ago, independent of when it was last referenced.
Segments

- Some software products generate many tables that grow and shrink in size during program execution
- Compilers are notorious for this
- Segments provide a way to provide completely independent address spaces
Segmented Memory

- Allows each table to grow or shrink independently of the other tables.
Comparison of Paging and Segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware of it?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear addresses spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can virtual address space exceed memory size?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can variable-sized tables be handled easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was the technique invented?</td>
<td>To simulate large memories</td>
<td>To provide multiple address spaces</td>
</tr>
</tbody>
</table>
External Fragmentation

(a) Segment 4 (7K)
   Segment 3 (8K)
   Segment 2 (5K)
   Segment 1 (8K)
   Segment 0 (4K)

(b) Segment 4 (7K)
   Segment 3 (8K)
   Segment 2 (5K)
   Segment 0 (4K)

(c) (3K)
   Segment 5 (4K)
   Segment 3 (8K)
   Segment 2 (5K)
   Segment 0 (4K)

(d) (3K)
   Segment 5 (4K)
   Segment 3 (8K)
   Segment 2 (5K)
   Segment 0 (4K)

(e) 10K
   Segment 5 (4K)
   Segment 6 (4K)
   Segment 2 (5K)
   Segment 0 (4K)
Segmentation with Paging

Two-part MULTICS address

18-Bit Segment number

Segment number

Descriptor segment

Descriptor

Page frame

Page table

Page

Word

6-Bit page number

10-Bit offset within the page

Offset

Friday, November 16, 2007

Computer Architecture I - Class 12
Virtual Memory on the Pentium 4

- Sophisticated!
- Supports:
  - Demand paging
  - Pure segmentation
  - Segmentation with paging
- Two tables:
  - LDT (Local Descriptor Table), one for each program
  - GDT (Global Descriptor Table), shared by all programs
Pentium 4 Selector

- Used for accessing a segment
- The 13-bit index specifies the LDT or GDT entry number, so each table can have at most 8 KB segment descriptors
Pentium 4 Code Segment Descriptor

- 8 bytes
- Includes segment’s base address, size, and other information

```
    BASE 0-15  32 Bits  LIMIT 0-15
    BASE 24-31 G D 0 LIMIT 16-19 P DPL TYPE BASE 16-23
```

- Segment type and protection
- Privilege level (0-3)
- 0: Segment is absent from memory
  1: Segment is present in memory

0: LIMIT is in bytes
1: LIMIT is in pages
0: 16-bit segment
1: 32-bit segment

Friday, November 16, 2007
Obtaining a Linear Address

- If paging is disabled, the linear address is interpreted as the physical address.
- If paging is enabled, the linear address is interpreted as a virtual address.
Protection on the Pentium 4

Diagram:
- User programs
- Shared libraries
- System calls
- Kernel
- Level 0
- Level 1
- Level 2
- Level 3

Possible uses of the levels

Friday, November 16, 2007
Computer Architecture I - Class 12
Virtual I/O Instructions

- I/O is one area where OSM and ISA levels differ considerably
- User does not want to deal with the complexities of device register interactions
- One way of organizing the virtual I/O is to use an abstraction called a file
Files

- To the operating system a file is normally just a sequence of bytes
  - However, some operating systems can organize a file into a sequence of logical records
- Any further structure is up to the application programs
- I/O is done by system calls for opening, reading, writing, and closing files
Disk Allocation Strategies

(a) A file in consecutive sectors.

(b) A file not in consecutive sectors.
## Keeping Track of Available Sectors

### Free list

<table>
<thead>
<tr>
<th>Track</th>
<th>Sector</th>
<th>Number of sectors in hole</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

### Bit map

<table>
<thead>
<tr>
<th>Track</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Directories

- Organize files on disk
- A user file directory and the contents of a typical entry in a file directory are shown at the right

<table>
<thead>
<tr>
<th>File 0</th>
<th>File name: Rubber-ducky</th>
</tr>
</thead>
<tbody>
<tr>
<td>File 1</td>
<td></td>
</tr>
<tr>
<td>File 2</td>
<td></td>
</tr>
<tr>
<td>File 3</td>
<td></td>
</tr>
<tr>
<td>File 4</td>
<td></td>
</tr>
<tr>
<td>File 5</td>
<td></td>
</tr>
<tr>
<td>File 6</td>
<td></td>
</tr>
<tr>
<td>File 7</td>
<td></td>
</tr>
<tr>
<td>File 8</td>
<td></td>
</tr>
<tr>
<td>File 9</td>
<td></td>
</tr>
<tr>
<td>File 10</td>
<td></td>
</tr>
<tr>
<td>Length:</td>
<td>1840</td>
</tr>
<tr>
<td>Type:</td>
<td>Anatidae dataram</td>
</tr>
<tr>
<td>Creation date:</td>
<td>March 16, 1066</td>
</tr>
<tr>
<td>Last access:</td>
<td>September 1, 1492</td>
</tr>
<tr>
<td>Last change:</td>
<td>July 4, 1776</td>
</tr>
<tr>
<td>Total accesses:</td>
<td>144</td>
</tr>
<tr>
<td>Block 0:</td>
<td>Track 4    Sector 6</td>
</tr>
<tr>
<td>Block 1:</td>
<td>Track 19   Sector 9</td>
</tr>
<tr>
<td>Block 2:</td>
<td>Track 11   Sector 2</td>
</tr>
<tr>
<td>Block 3:</td>
<td>Track 77   Sector 0</td>
</tr>
</tbody>
</table>
Parallel Processing

(a) True parallel processing with multiple CPUs

(b) Parallel processing simulated by switching one CPU among three processes
Processes

- A program runs as part of a process
- Processes can be created and terminated dynamically
- To achieve parallel processing a system call to create a process is needed
- Created process can:
  - Exist in a parent/child relationship with creating process
  - Run completely independently of creating process
Communication

- Parallel processes need to communicate and synchronize with each other to get their work done
- As an example, two processes can communicate via a shared memory buffer
- Consider process 1 the **producer**, which puts data values into the buffer
- Consider process 2 the **consumer**, which takes data values out of the buffer
Synchronization

- The two processes run in parallel at different rates
- If the producer discovers the buffer is full it suspends itself until a signal from the consumer tells it there is room in the buffer
- If the consumer discovers the buffer is empty it suspends itself until a signal from the producer that there is a number in the buffer
Circular Buffer
Race Condition

- Suppose only one number left in buffer at location 21 ($\text{in} = 22$ and $\text{out} = 21$)
- Now suppose the consumer takes this number out, changing $\text{out}$ to 22
- Suppose the consumer continues, and gets to the point where it fetches $\text{in}$ and $\text{out}$ getting ready to compare them to see if they are equal (indicating an empty buffer)
- After the fetch, but before the comparison, the producer puts the next number into the buffer and increments $\text{in}$ to 23
Race Condition

- It now recognizes there is only one number in the buffer so sends a signal to the consumer to wake up; the consumer is awake so the signal is lost.
- Now the consumer goes back to work and thinks the buffer is empty (\(in = out\)) and goes to sleep.
- Now the producer puts another number in the buffer, but since it sees two numbers in the buffer it (incorrectly) assumes the consumer is awake.
Race Condition

- Producer continues running and eventually fills the buffer; when it does so it will go to sleep
- Consumer is asleep and never gets a signal to wake up
- Both processes are sleeping and will remain so forever
- This came about because of a race condition, because it was a race to see who modified in or out first
Semaphores

- Non-negative integer variables that solve the communication problem between parallel processes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semaphore = 0</th>
<th>Semaphore &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>Semaphore = semaphore + 1; if the other process was halted attempting to</td>
<td>Semaphore = semaphore + 1</td>
</tr>
<tr>
<td></td>
<td>complete a down instruction on this semaphore, it may now complete the down</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and continue running</td>
<td></td>
</tr>
<tr>
<td>Down</td>
<td>Process halts until the other process ups this semaphore</td>
<td>Semaphore = semaphore − 1</td>
</tr>
</tbody>
</table>
Windows XP
Windows XP Virtual Memory

- Each process has its own virtual address space
- Virtual addresses are 32 bits long, so there’s a 4 GB virtual address space
- The lower 2 GB are available for the process’ code and data
- The upper 2 GB allow limited access to kernel memory
Windows XP Virtual I/O

- XP supports several file systems, but the one of most interest is NTFS (NT File System)
- Unicode is used throughout XP
- File names can be up to 255 characters
- A file is just a linear sequence of bytes, up to a maximum of $2^{64}-1$ bytes
## Principal Win32 API

### Functions for File I/O

<table>
<thead>
<tr>
<th>API function</th>
<th>UNIX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFile</td>
<td>open</td>
<td>Create a file or open an existing file; return a handle</td>
</tr>
<tr>
<td>DeleteFile</td>
<td>unlink</td>
<td>Destroy an existing file</td>
</tr>
<tr>
<td>CloseHandle</td>
<td>close</td>
<td>Close a file</td>
</tr>
<tr>
<td>ReadFile</td>
<td>read</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>WriteFile</td>
<td>write</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>SetFilePointer</td>
<td>lseek</td>
<td>Set the file pointer to a specific place in the file</td>
</tr>
<tr>
<td>GetFileAttributes</td>
<td>stat</td>
<td>Return the file properties</td>
</tr>
<tr>
<td>LockFile</td>
<td>fcntl</td>
<td>Lock a region of the file to provide mutual exclusion</td>
</tr>
<tr>
<td>UnlockFile</td>
<td>fcntl</td>
<td>Unlock a previously locked region of the file</td>
</tr>
</tbody>
</table>
## Principal Win32 API Functions for Directory Work

<table>
<thead>
<tr>
<th>API function</th>
<th>UNIX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateDirectory</td>
<td>mkdir</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>RemoveDirectory</td>
<td>rmdir</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>FindFirstFile</td>
<td>opendir</td>
<td>Initialize to start reading the entries in a directory</td>
</tr>
<tr>
<td>FindNextFile</td>
<td>readdir</td>
<td>Read the next directory entry</td>
</tr>
<tr>
<td>MoveFile</td>
<td></td>
<td>Move a file from one directory to another</td>
</tr>
<tr>
<td>SetCurrentDirectory</td>
<td>chdir</td>
<td>Change the current working directory</td>
</tr>
</tbody>
</table>
Windows XP Master File Table

MFT entry for one file

MFT header

Master file table

Standard information File name MS-DOS name Security Data
Windows XP Process Management

- XP supports multiple processes
- Processes can communicate and synchronize
- No enforcement of parent-child or other hierarchy – all processes are created equal