Chapter 7

Memory

Users want large and fast memories!

Static Random Access Memory
access times are 2 - 25ns at cost of $100 to $250 per Mbyte.

Dynamic Random Access Memory
access times are 60-120ns at cost of $5 to $10 per Mbyte.

Disk
access times are 10 to 20 million ns at cost of $.10 to $.20 per Mbyte.

Memory Hierarchy
Registers
L1 Cache
L2 cache
Main Memory
Disk cache
Disk

Levels in the memory hierarchy

Increasing distance from the CPU in access time

Size of the memory at each level
Our initial focus: two levels (upper, lower) block:
  minimum unit of data

hit:
  data requested is in the upper level

miss:
  data requested is not in the upper level

Memory management - Basics
- swapping
- contiguous allocation
- paging
- segmentation

Programs must be in memory to run

Limited physical memory

User does not know a priori where the program will reside in memory

Assumes it resides contiguously starting at 0.

Logical address - generated by user programs relative to location 0 in memory
  0 - L addresses .. L ⇒ P

Physical address - actual addresses used to fetch and store data in memory
  N - N+L
Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.

- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

- 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 many systems, i.e., UNIX and Microsoft Windows.
Contiguous Allocation

- Multiple-partition allocation
  - **Hole**
    - block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)
Hardware for address mapping
- base ex 10000
- limit 2800

If Logical address < limit
then (Base register + Logical address) = Physical address
Else
trap; addressing error

Fixed partitioning
Equal size $\rightarrow$ placement algorithm easy - load into available partition
Unequal size $\rightarrow$
placement
  1. Assign each process to smallest - one process queue per partition
  2. Single process queue.

OS maintains info about allocated partitions and free partitions
Simple to implement
Inefficient use of memory due to internal fragmentation

* First-fit: Allocate the first hole that is big enough.
* Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
* Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

ex
 memory partitions
  100K
  500K
  200K
  300K
  600K
process
  212K
  417K
  112K
  426K
First fit – 500K, 600K, 288K, wait
Best fit - 300K, 500K, 200K, 600K
Dynamic partitioning
Partitions are created dynamically
Each process loaded into a partition exactly the same size as that process
   No internal fragmentation
   Inefficient use of processor due to the need for compaction to counter external fragmentation

Allocation
- Single queue for all partitions
- Multiple queues for all partitions

Ex. Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>600K</td>
<td>10</td>
</tr>
<tr>
<td>P₂</td>
<td>1000K</td>
<td>5</td>
</tr>
<tr>
<td>P₃</td>
<td>300K</td>
<td>20</td>
</tr>
<tr>
<td>P₄</td>
<td>700K</td>
<td>8</td>
</tr>
<tr>
<td>P₅</td>
<td>500K</td>
<td>15</td>
</tr>
</tbody>
</table>

memory available
2160K

P₁, P₂, P₃, P₁, P₃, P₄, P₁, P₃, P₄, P₃, P₄, P₅

Job scheduling
- with/without skip process - if first process cannot fit in a free partition, select next in the queue
- each queue separately
- one queue for all jobs
- keep memory regions from being idle
- allow swapping -
   to reduce context switching
   overlap swapping and program execution
   minimize the amount of memory of a process to be swapped
   increase the speed of the disk used for swapped out process
- high priority jobs first
To manage memory
Partition the program into sections - ex. Program into Code and data
   Need multiple base registers
      - code
      - data
easier to find a fit
code sharing

Paging
- Frame
   Fixed size block of physical memory - memory divided into equal size partitions
- Page
   divide program into partition of size of frame
   Keep track of all the frames

To run the program of Size N partitions -- Find N free frames
   Need not be contiguous

Page table contains - base address of each frame and corresponding page number

Ex.

<table>
<thead>
<tr>
<th>Program</th>
<th>memory</th>
<th>page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 A</td>
<td>page</td>
<td>0 frame</td>
</tr>
<tr>
<td>1 B</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2 C</td>
<td>2 A</td>
<td>1 5</td>
</tr>
<tr>
<td></td>
<td>3 C</td>
<td>2 3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 B</td>
<td></td>
</tr>
</tbody>
</table>

Ex.
Paging hardware

Logical address - divided into two parts \( p \ d \)  ------
- \( p \) - left hand bits  \( d \) - right hand bits
- \( p \) - page number - represents starting address of the frame  say \( f \)

Then \( PA = f + D \)
- \( D \) - displacement or offset

Ex.
Hardware for 16 bit machine with page size 512 bytes
- \( LA = 16 \) bits
- Page size 512 bytes \( \Rightarrow 9 \) bits \( -d \)
  \( \Rightarrow \) first 7 bits represent the page number - \( p \)
Ex.

Relative address 1502 \(\Rightarrow\) 000001 | 0111011110
Page size 1K
\(\Rightarrow\) page 1 offset 478

If page 1 in page table represents 000110 \(\Rightarrow\) 16 bit PA 000110 | 0111011110

ex. – Logical address space
8 pages of 1024 words

- Physical memory \(\Rightarrow\) 32 frames

- How many bits are in logical address/physical address

8 pages of 1024 words \(\quad\) (13 bits)

Physical memory \(\Rightarrow\) 32 frames \(\quad\) (15 bits)
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>P1</td>
<td>P2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>D1</td>
<td>12</td>
</tr>
</tbody>
</table>

where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table.

Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture
Segmentation

- Segment
  Divide memory into variable size partitions
  Can be used for procedures, functions, local variables, global variables, stack etc..

Addressing
Segment #, offset \( S \ D \)

Hardware
Need a segment table
If \( D < \) limit then \( D + \) base represented at segment# = PA
Else trap

Ex.
Logical address \( 0001 | 000111011110 \)
4 bit segment #
segment # 1 gives base \( 0010000000100000 \)
then 16 bit PA \( 0010001100010000 \)

Page + Segment systems
Page the segments
\( S | D = S | P | D \)

\( S \) - displacement in the segment table
\( P \) - displacement in the page table \( D \) - offset

ex

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>219</td>
<td>600</td>
</tr>
<tr>
<td>1</td>
<td>2300</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1327</td>
<td>580</td>
</tr>
<tr>
<td>4</td>
<td>1952</td>
<td>96</td>
</tr>
</tbody>
</table>

LA
\( 0, 430 \)  \( 219 + 43 \)
\( 1, 10 \)  \( 2300 + 10 \)
\( 2, 500 \)  error
Four important concerns with Caches

1. Memory blocks can be placed in the cache
   One place - direct map
   Any place - Full associative
   Few places - set associative

2. A block in the cache can be found
   Indexing - in direct map
   Limited search - set-associative
   Full search - associative

3. Block replacement
   Least recently used
   FIFO
   Random

4. Writes
   Write in the cache and memory at the same time called write through
   Write-back - first write in the cache then in the memory when the block replaced
Locality

temporal locality:
If an item is referenced, it will tend to be referenced again soon

spatial locality:
If an item is referenced, nearby items will tend to be referenced soon.

Temporal locality
Low
Data
Accessing variables only once
Code
No loops
No reuse of instructions

High
Data
Accessing variables over and over again and again
Code
Loops
Reuse of instructions

Spatial locality
Low
Data
scattered
No arrays
Code
Lots of jumps to far away places

High
Data
Arrays
Code
No branches/jumps
Direct Mapping
Each main memory block has one specific location in the cache

Block j of the main memory maps onto block \((J \mod \text{cache size})\)

Example
Cache size 128
Block 0, 128, 256, ... mapped one at a time to cache block 0.
**Direct Mapped Cache**

Memory address divided into 3 fields
- Tag,
- block
- word

Tag represents the higher order bits of the memory address
  Match the tags in the memory address and cache address to determine whether the block is in cache

Block field determines the cache position
Word field determines the number of words in a block

Example
- Word field 4 → 16 words in a block
- Word field 2 → 4 words in a block

![Diagram of Direct Mapped Cache](image-url)
Direct Mapped Cache

• Taking advantage of spatial locality:
Hits vs. Misses

- Read hits
  - this is what we want!

- Read misses
  - stall the CPU, fetch block from memory, deliver to cache, restart

- Write hits:
  - can replace data in cache and memory (write-through)
  - write the data only into the cache (write-back the cache later)

- Write misses:
  - read the entire block into the cache, then write the word
Associative mapping
More flexible mapping
Main memory block can be mapped onto any cache block position

Two fields
Tag and word

Compare the tag bits of an address received from the CPU with the tag bits of each block of the cache

High search cost
Need to search all tag patterns in the cache

Set-associative mapping
Combination of direct and associative mapping

Few choices for block placement
Decrease the size of associative search

Have three fields
Tag
Set
Word

Set field determines number of sets in the cache

Set field also determines which set of the cache might contain the desired block

Example
128 block cache
7 bit tag

7 bit tag implies 128 different sets of blocks
each cache set with one block (128/128) ➔ direct mapping

Example
128 block cache
6 bit tag

6 bit tag implies 64 different sets of blocks
each cache set with two blocks (128/64) ➔ two-set associative

5 bit tag implies 32 different blocks
each cache set with four blocks (128/32) ➔ four set associative

0 bit tag implies different block
each cache set with 128 blocks (128/1) ➔ full associative
Cache size is 8 blocks.

Where does word 12 from memory go?

**Fully associative:**
Block 12 can go anywhere

**Direct mapped:**
Block no. = (Block address) mod (No. of blocks in cache)
Block 12 can go only into block 4 (12 mod 8 = 4)

=> Access block using lower 3 bits

**2-way set associative:**
Set no. = (Block address) mod (No. of sets in cache)
Block 12 can go anywhere in set 0 (12 mod 4 = 0)

=> Access set using lower 2 bits
Decreasing miss ratio with associativity

Compared to direct mapped, give a series of references that:
  – results in a lower miss ratio using a 2-way set associative cache
  – results in a higher miss ratio using a 2-way set associative cache

assuming we use the “least recently used” replacement strategy