The Memory Hierarchy...

- Brief Overview of Memory Design
  - What is the memory hierarchy?
  - Focus on Cache design
    - How does Cache work?
    - How are addresses mapped to Cache?
    - How to rewrite code to get better cache performance?
  - How do disks work?
  - Virtual memory – what is it?

A Computer

- The computer is composed of input devices, a central processing unit, a memory unit and output devices.

Computer Organization - Recap

- CPU has two components:
  - Arithmetic and Logic Unit (ALU)
    - Performs arithmetic operations
    - Performs logical operations
  - Control Unit
    - Controls the action of the other computer components so that instructions are executed in the correct sequence
- Memory
  - Contains instructions and data
  - CPU requests data, data fetched from memory
    - How long does it take to fetch from memory?
CPU + memory

- An ordered sequence of storage cells, each capable of holding a piece of data.
- Address space
  - Size of memory: N bit address space = 2^N memory locations
- Addressability
  - Size of each memory location – k bits
- Total memory size = k.2^N bits
- Assumption thus far: Processor/CPU gets data or instruction from some memory address (Inst fetch or Load/Store instruction)
  - But how is memory actually organized?
  - Can everything we need fit into a memory that is close to the CPU?

Where does run-time stack fit into this.

- We looked at how variables in C are allocated memory addresses
  - Each function has activation record
  - Compiler takes care of allocation of memory addresses to variables
- Question now is: where are these memory addresses and how long does it take to fetch the contents into the processor register
  - LD R0, A
    - We know the address of A, but how long will it take to go into memory and fetch into register R0?

Memory Unit

- Random access memory
  - Can read from any location by supplying address of data
- Memory Comes in Many Flavors
  - Main RAM memory Key features
    - RAM is packaged as a chip.
    - Basic storage unit is a cell (one bit per cell).
    - Multiple RAM chips form a memory.
    - SRAM (Static Random Access Memory) or DRAM (Dynamic Random Access Memory)
  - ROM, EPROM, EEPROM, Flash, etc.
    - Read only memories – store OS
  - “Secondary memory” Disks, Tapes, etc.
- Difference in speed, price and “size”
  - Fast is small and/or expensive
  - Large is slow and/or cheap

Memory Technology

- Random access memory
  - Can read from any location by supplying address of data
- Memory Comes in Many Flavors
  - Main RAM memory Key features
    - RAM is packaged as a chip.
    - Basic storage unit is a cell (one bit per cell).
    - Multiple RAM chips form a memory.
    - SRAM (Static Random Access Memory) or DRAM (Dynamic Random Access Memory)
  - ROM, EPROM, EEPROM, Flash, etc.
    - Read only memories – store OS
  - “Secondary memory” Disks, Tapes, etc.
- Difference in speed, price and “size”
  - Fast is small and/or expensive
  - Large is slow and/or cheap
**How is memory really organized?**

- Many types of memory with different speeds
- Processor speed and memory speed mismatched
  - Data transferred between memory and processor
  - Instructions or data
- What does processor do while waiting for data to be transferred?
  - Idle – processor is stalled leading to slowdown in speed and lower performance
- Why can’t we have memory as fast as processor?
  - Technology, cost, size
- What is the solution then?

**Memory Hierarchies**

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

---

**An Example Memory Hierarchy**

- **L0:** Register file
- **L1:** On-chip L1 cache (SRAM)
- **L2:** Off-chip L2 cache (SRAM)
- **L3:** Main memory (DRAM)
- **L4:** Local secondary storage (local disks)
- **L5:** Remote secondary storage (distributed file systems, Web servers)

- Smaller, faster, and costlier (per byte)
- Storage devices
- Larger, slower, and cheaper (per byte)
- Storage devices

---

**Items on a desktop (register) or in a drawer (cache) are more readily accessible than those in a file cabinet (main memory) or in a closet in another room.**

- Keep more frequently accessed items on desktop, next frequent in drawer, etc.
- And things you need a lot less often in the closet in another room!
Memory Hierarchies

- **Key Principles**
  - Locality – most programs do not access code or data uniformly
  - Smaller hardware is faster
- **Goal**
  - Design a memory hierarchy “with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level”
  - This implies that we be clever about keeping more likely used data as “close” to the CPU as possible
- **Levels provide subsets**
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory

Locality

- **Principle of Locality:**
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - **Temporal locality:** Recently referenced items are likely to be referenced in the near future.
  - **Spatial locality:** Items with nearby addresses tend to be referenced close together in time.

Locality Example:

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Memory Hierarchy: The Tradeoff

- **Memory:**
  - CPU
  - Virtual memory
  - Cache
  - Other memory levels

Locality

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```
We shall return to a detailed discussion of the various components of the memory hierarchy:
- What is a Cache memory?
- How does Memory access and Disk access work?
- How should we organize the memory hierarchy?
- How can we rewrite our code to improve performance?
  - Based on memory access, type of instructions, etc.

A Simple Model of Memory...
- Sequence of addresses
  - How many?
- CPU generates request for memory location – i.e., an address
  - How long does it take to get this data?
    - Depends where it is in the Memory Hierarchy
- Simplified Model for memory hierarchy:
  - Small amount of On-chip Cache memory
  - Larger amount of off-chip Main memory
  - Huge Disk

Memory Access times
- Memory access time
  - On-chip Cache takes 1 processor cycle
  - Main memory takes a number (10-50) processor cycles
  - Disk takes a huge amount
- Simple model we will use for now:
  - Memory = Cache + Main memory
  - Small size Cache ≠ not everything fits in it
- Simplified Cache organization:
  - Cache consists of a set of blocks each of some number of bytes
    - Only a block can be fetched into and out of cache
    - Eg: if block is 16 bytes, then load 16 bytes into cache
      - Cannot load a single byte
Memory Access times using Simplified Model

- If data is found in Cache then time = 1
  - Called a cache hit
- Else time is Main memory access time
  - Cache miss, means read from next level
- Note: need a ‘control unit’ to determine if location is in cache or not
  - Cache controller
- Why does concept of caching work?
  - Principle of Locality
    - Programs access data nearby, or data/instructions that were used recently

Summary: Memory Access time optimization

- If each access to memory leads to a cache hit then time to fetch from memory is one cycle
  - Program performance is good!
- If each access to memory leads to a cache miss then time to fetch from memory is much larger than 1 cycle
  - Program performance is bad!
- Design Goal:
  - How to arrange data/instructions so that we have as few cache misses as possible.

Some details of Memory Organization

Random-Access Memory (RAM)

- Static RAM (SRAM)
  - This is what we saw before in the example of a memory built from latches (transistors).
  - Like a register – once written it keeps its value till next write
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.
- Dynamic RAM (DRAM)
  - Each cell stores a bit with a capacitor and transistor.
  - Value must be refreshed every 10-100 ms else we lose data
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.
  - Many types of DRAMS
    - SDRAM, DDR-DRAM, Video DRAM, etc.
Nonvolatile Memories

- DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off.
  - Generic name is read-only memory (ROM).
  - Misleading because some ROMs can be read and modified.
- Types of ROMs
  - Programmable ROM (PROM)
  - Eraseable programmable ROM (EPROM)
  - Electrically erasable PROM (EEPROM)
  - Flash memory
- Firmware
  - Program stored in a ROM
    - Boot-time code, BIOS (basic input/output system)
    - Graphics cards, disk controllers.

SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th>Trans. per bit</th>
<th>Access time</th>
<th>Persist?</th>
<th>Sensitive?</th>
<th>Cost</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>6</td>
<td>1X</td>
<td>Yes</td>
<td>No</td>
<td>100x</td>
<td>cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>No</td>
<td>Yes</td>
<td>1X</td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>

Building the Memory

- Build large memory using several smaller memory chips
  - How to organize a large memory using several "modules"
- CPU generates an address request
  - The address can be in any module
  - Need to figure out which one
Typical Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.

CPU chip

register file

ALU

system bus

memory bus

bus interface

I/O bridge

main memory

Memory Read Transaction (1)

- CPU places address A on the memory bus.

Load operation: Load R0, A
or in Intel: movl A, R0

Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Load operation: Load R0, A
or in Intel: movl A, R0

Memory Read Transaction (3)

- CPU reads word x from the bus and copies it into register %eax.
Memory Write Transaction (1)

- CPU places address A on the bus. Main memory reads it and waits for the corresponding data word to arrive.

Memory Write Transaction (2)

- CPU places data word y on the bus.

Memory Write Transaction (3)

- Main memory read data word y from the bus and stores it at address A.
**Memory Access Time and Performance**

\[ \text{CPU time} = \text{Seconds} \times \text{Instructions} \times \text{Cycles} \times \text{Seconds} \]

\[ \text{CPU} = IC \times CPI \times Clk \]

**How does memory access time fit into CPU time equation?**

---

**The CPU-Memory Gap**

- The increasing gap between DRAM, disk, and CPU speeds.

---

**Performance**

- Simplified model:
  - Processor is (1) in execution or (2) waits for memory
  - "effective" (Real) CPI increases
  - execution time = (execution cycles + memory stall cycles) \( \times \) cycle time

- Improve performance = decrease stall cycles
  - Decrease time to access memory
  - How?

---

**Memory Hierarchies**

- Organize memory as a memory hierarchy.

- Key Principles
  - Locality - most programs do not access code or data uniformly
  - Smaller hardware is faster

- Goal
  - Design a memory hierarchy "with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level."
  - This implies that we be clever about keeping more likely used data as "close" to the CPU as possible.

- Levels provide subsets
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory.
Recap – Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality Example:
- Data
  - Reference array elements in succession: spatial locality
  - Reference same each iteration: temporal locality
- Instructions
  - Reference instructions in sequence: spatial locality
  - Cycle through loop repeatedly: temporal locality

Cache memory

- Why?
- How does it work?
- How is the memory organized?
Why Cache?

- Gap between main memory speed and processor speed
  - Reading data/inst from memory will take more than 1 processor cycle
  - Increases time to execute program
- Place a small but fast memory close to the processor
- Why does this work
  - Principle of Locality

Simple Model of Memory Hierarchy...

- Sequence of addresses
  - How many?
- CPU generates request for memory location – i.e., an address
  - How long does it take to get this data?
    - Depends where it is in the Memory hierarchy
- Simplified Model for memory hierarchy:
  - small amount of Fast On-chip Cache memory
  - Larger amount of off-chip Main memory
  - Huge Disk

How does Cache memory work?

- Address space = $2^N$ words each of some size K bits
  - N bit address
- Memory addresses go from 0 to $2^N$-1
  - These are the addresses that the processor requests in the Load or Store instructions, or when fetching a new instruction (value in PC)
- Some of these memory locations are placed in the cache
  - If you see it in the cache then don’t need to go all the way to memory to read them
    - Faster time to read/write inst/data!

Memory Access times

- memory access time
  - On-chip Cache takes 1 processor cycle
  - Main memory takes a number (10-50) processor cycles
  - Disk takes a huge amount
- Simple model we will use:
  - Memory = Cache + Main memory
  - Small size Cache = not everything fits in it
- Cache organization:
  - Cache consists of a set of blocks each of some number of bytes
    - Only a block can be fetched into and out of cache
    - Eg: if block is 16 bytes, then load 16 bytes into cache
    - Cannot load a single byte
Memory Access times using Simplified Model

- If data is found in Cache then time = 1
  - Called a cache hit
- Else time is Main memory access time
  - Cache miss, means read from next level
- Note: need a ‘control unit’ to determine if location is in cache or not
  - Cache controller
- Why does concept of caching work?
  - Principle of Locality
    - Programs access data nearby, or data/instructions that were used recently

Terminology Summary

- Hit: data appears in block in upper level (i.e. block X in cache)
  - Hit Rate: fraction of memory access found in upper level
  - Hit Time: time to access upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (i.e. block Y in memory)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Extra time to replace a block in the upper level + Time to deliver the block to the processor
  - Hit Time << Miss Penalty (500 instructions on Alpha 21264)

Memory Hierarchy—Performance

- Placing the fastest memory near the CPU can result in increases in performance
- Consider the number of cycles the CPU is stalled waiting for a memory access: memory stall cycles
  - CPU execution time = (CPU clk cycles + Memory stall cycles) * clk cycle time.
  - Memory stall cycles = number of misses * miss penalty
  - Fewer misses in cache = better performance!

Cache Performance – Simplified Models

- Memory stall cycles = number of misses * miss penalty
  - = IC*(memory accesses/instruction)*miss rate * miss penalty
- Hit rate/ratio r = number of requests that are hits/total number requests
  - Miss rate = 1 - (hit rate)
- Cost of memory access= rC_h + (1-r) C_m
  - C_h is cost/time from cache, C_m is cost/time when miss – fetch from memory
  - Extend to multiple levels of cache
  - Hit ratios for level 1, level 2, etc.
    - Access times for level 1, level 2, etc.
    - \( r_1 C_h + r_2 C_m + (1-r_1) C_m \)
So how well does your memory perform?

- What is the average time to access memory?
- An application program has a miss rate
- The memory design gives us hit time and miss penalty

So what is the average memory access time (AMAT)?
- Hit time: if found in cache
- Hit time + miss penalty: if not in cache

Average Memory Access Time

\[ \text{AMAT} = \text{HitTime} + (1 - h) \times \text{MissPenalty} \]

- Hit time: basic time of every access.
- Hit rate \( h \): fraction of access that hit
  - usually substituted by miss rate \( m = (1-h) \)
- Miss penalty: extra time to fetch a block from lower level, including time to replace in CPU

Example 1.

- System = Processor, Cache, Main Memory
  - Cache time = 1 processor cycle
  - Memory access time = 50 processor cycles
- Suppose out of 1000 memory accesses (due to Load/Store and Inst fetch)
  - 40 misses in the cache
  - Miss ratio = 40/1000 = 4%
- Average memory access time with and without cache?

Example 2.

- Average memory access time with and without cache?
- AMAT-cache = \[ 1 + \text{miss ratio} \times \text{miss penalty} \]
  - AMAT-cache = 1 + 0.04\times50 = 3
- AMAT-without-cache = 50
- What happens if miss ratio increases?
Example 2

- In reality we have an application with a instruction mix
  - CPU time = IC*CPI*Clock
  - Effective CPI = CPI + Average Stalls per instruction
- How many memory access per instruction?
  - When is memory accessed?
- Example: suppose your program has 20% Load/Store operations

Example 2 – complete example

- Assume “ideal” CPI (without memory stalls) is 1.5
- What is the CPU time now?
- CPU = IC * CPI * Clock
  - IC*CPI is the number of CPU cycles
  - CPI is number of cycles for CPU execution per instruction
  - Each instruction stalls for some number of cycles
- IC * (CPI + Avg Stall cycles)*Clock
  - IC*(1.5+2.4)*clock = IC*3.9 clock cycles

Cache Memory Hardware Design

- Main memory has 2^N locations
- Cache has 2^k locations
  - Smaller than main memory
  - How to “organize” these cache locations?
- Processor generates N bit address
- How do we look at this N bit address and decide (a) if it is in cache and (b) where to place it in cache?
Cache Design—Questions

- **Q1:** Where can a block be placed in the upper level?
  - block placement
- **Q2:** How is a block found if it is in the upper level?
  - block identification
  - Use the address bits to identify the block
- **Q3:** Which block should be replaced on a miss?
  - block replacement
- **Q4:** What happens on a write?
  - Write strategy

Unified or Separate I-Cache and D-Cache

- **Two types of accesses:**
  - Instruction fetch
  - Data fetch (load/store instructions)
- **Unified Cache**
  - One large cache for both instructions and data
    - Pros: simpler to manage, less hardware complexity
    - Cons: how to divide cache between data and instructions? Confuses the standard harvard architecture model; optimizations difficult
- **Separate Instruction and Data cache**
  - Instruction fetch goes to I-Cache
  - Data access from Load/Stores goes to D-cache
    - Pros: easier to optimize each
    - Cons: more expensive; how to decide on sizes of each

Definitions

- Cache has a total size – number of bytes in cache
- Transfers take place in **blocks**
  - A whole block is transferred between memory and cache
- Locating a block requires two attributes:
  - Size of block
  - Organization of blocks within the cache
- **Block size** (also referred to as **line size**)
  - Granularity at which cache operates
  - Each block is contiguous series of bytes in memory and begins on a naturally aligned boundary
  - Each cache with 16 byte blocks
    - Each contains 16 bytes
    - Finishes aligned to 16 byte boundaries in address space
    - Last word of a block of 16 bytes must be aligned
  - Smallest usable block size is the natural word size of the processor
    - Else would require splitting an access across blocks and slow down translation

Memory viewed as blocks

- If cache block size = \( K \) bytes, then memory can be viewed as contiguous set of blocks each of size \( K \)
Where can a block be placed in a cache?

- **Cache Organization**
  - If each block has only one place it can appear in the cache, it is said to be "direct mapped" and the mapping is usually \((\text{Block address}) \mod (\text{Number of blocks in the cache})\).
  - If a block can be placed anywhere in the cache, it is said to be **fully associative**.
  - If a block can be placed in a restrictive set of places in the cache, the cache is **set associative**. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. If there are \(n\) blocks in a set, the cache is called **\(n\)-way set associative**.

**Cache Organizations**

- **Direct Mapped vs Fully Associate**
  - Direct mapped is not flexible enough:
    - If \(X \mod K = Y \mod K\), then \(X\) and \(Y\) cannot both be located in cache.
    - Fully associative allows any mapping, implies all locations must be searched to find the right one — expensive hardware.

- **Set Associative**
  - Compromise between direct mapped and fully associative.
  - Allow many-to-few mappings.
  - On lookup, subset of address bits used to generate an index.
  - BUT index now corresponds to a set of entries which can be searched in parallel — more efficient hardware implementation than fully associative, but due to flexible mapping behaves more like fully associative.

**Cache Design: How is data found in Cache?**

- Processor generates address request — some \(N\) bit address.
- Two issues:
  - How do we know if a data item is in the cache?
  - If it is, how do we find it? In which cache block do we look?
- "direct mapped"

For each item of data at the lower level, there is exactly one location in the cache where it might be.

- e.g., lots of items at the lower level share locations in the upper level.
Addressing.

- Memory address of N bits
  - This is what is requested by processor
- We need mapping from Addresses to Cache
- How to break up these N bits into fields so that the cache controller can easily determine if the address requested is already stored in cache?
- Focus on Direct mapped:
  - How to implement Mod.P where P=2^K
  - How to determine which cache block to map the address -- index
  - How to determine which of the words is the one in memory -- tag
  - Since we are working with blocks, how to find the specific word within the block -- offset
  - N bit address is broken into these 3 fields!

Addressing -- Example

- 8 bit address
  - Byte addressability
- 4 byte cache blocks
  - Each cache block has 4 bytes
- Total size of cache = 16 bytes

Which cache block do we place address 01101101?

How do we know which block from memory is in the cache block?
- Is it 01101101 or 11001101
Addressing – Example

- 8 bit address
  - Byte addressability
- 4 byte cache blocks
  - Each cache block has 4 bytes = need 2 bits to specify which byte within the block
- Total size of cache = 16 bytes
  - 4 blocks in cache = apply Modulo 4 mapping
- Which cache block do we place 01101101
  - Last two LSB are offset within a block
  - Next 2 LSB are modulo 4 and specify cache block
- How do we know which block from memory is in the cache block?
  - Is it 01101101 or 11001101

So how complex is the hardware required?

- Given N bit address, the cache controller needs to determine if the requested word is in cache or not
  - If not, then send the address to the memory bus
- This process has to be built into hardware
  - Complexity of the circuit determines time taken to determine if word is in cache
  - Hit time is function of complexity of circuit
  - For direct mapped, can you think of a hardware design?

Direct Mapped Caches

Cache Block Replacement?

- When we run out of space on the cache and need to bring in new block into cache
  - Replace a block that is currently in cache to create space
- Which block to replace?
  - Easy in Direct mapped cache – no choice!
  - More complex in Associative Caches since we have a choice
What happens on write?

- **Write through** –
  - Propagate each write through cache and on to next level written to both the block in the cache and to the block in lower level memory.

- **Write back** –
  - The information is written only to the block in the cache, memory is written only when block is replaced.
  - A dirty bit is kept in the block description area.

- Tradeoffs??

Performance

- **Simplified model:**
  \[ \text{execution time} = (\text{execution cycles} + \text{stall cycles}) \times \text{cycle time} \]
  \[ \text{stall cycles} = \# \text{ of instructions} \times \text{miss ratio} \times \text{miss penalty} \]

- **Two ways of improving performance:**
  - Decreasing the miss ratio
  - Decreasing the miss penalty

What happens if we increase block size?

- Increasing the block size tends to decrease miss rate:

<table>
<thead>
<tr>
<th>Program</th>
<th>Miss rate</th>
<th>Effective combined miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1.6%</td>
<td>2.1%</td>
</tr>
<tr>
<td>spice</td>
<td>1.2%</td>
<td>1.3%</td>
</tr>
</tbody>
</table>

- Use split caches because more spatial locality in code:

Decreasing miss penalty with multilevel caches

- **Add a second level cache:**
  - Often primary cache is on the same chip as the processor.
  - Use SRAMs to add another cache above primary memory (DRAM).
  - Miss penalty goes down if data is in 2nd level cache.

- **Using multilevel caches:**
  - Try and optimize the hit time on the 1st level cache.
  - Try and optimize the miss rate on the 2nd level cache.
Summary: Memory Access time optimization

- If each access to memory leads to a cache hit then time to fetch from memory is one cycle
  - Program performance is good!
- If each access to memory leads to a cache miss then time to fetch from memory is much larger than 1 cycle
  - Program performance is bad!
- Design Goal:
  How to arrange data/instructions so that we have as few cache misses as possible.

Secondary Memory

- CPU fetches from memory, memory is a sequence of $2^n$ locations
- What happens if main memory (chip capacity) is less than $2^n$
- Data and programs may be stored in a non-volatile component
  - MS-Word executable is on disk
  - Your application data
- What happens if more than one process is running on the system
  - Multiprogramming
  - What is the address space available to each user?
- Need to use Disks!

What is multiprogramming and Why?

- Processor overlaps execution of two processes/programs
  - When one is waiting for I/O, the other is “swapped” in to the processor
    - Save the “state” of the process being swapped out
- Processes need to share memory
  - Each has its own address space
- Leads to better throughput and utilization
The Complete memory hierarchy

- Processor has a set of registers
  - Processor instructions operate on contents in the registers
- Small, fast cache memory placed near the processor
- Main memory sitting outside the chip
  - If data is not in the cache then fetch from main memory
  - Takes longer to access main memory than cache
- Disk sitting outside the motherboard
  - If data/program not in main memory then fetch/load from disk
  - Takes much longer to access disk than main memory

Why does memory hierarchy work?

- Principle of locality!
  - Exploit at each level.

How do disks work?

Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.
**Disk Geometry (Multiple-Platter View)**

- Aligned tracks form a cylinder.

![Diagram of disk geometry](image)

**Disk Capacity**

- **Capacity**: maximum number of bits that can be stored.
  - Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9.
- Capacity is determined by these technology factors:
  - Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - Areal density (bits/in²): product of recording and track density.
- Modern disks partition tracks into disjoint subsets called recording zones:
  - Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  - Each zone has a different number of sectors/track.

**Disk Operation (Single-Platter View)**

- The disk surface spins at a fixed rotational rate.
- The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air.
- By moving radially, the arm can position the read/write head over any track.
**Disk Access Time**

- Average time to access some target sector approximated by:
  - \( T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \)
- **Seek time** (Tavg seek)
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek = 9 ms
- **Rotational latency** (Tavg rotation)
  - Time waiting for first bit of target sector to pass under r/w head.
  - \( T_{avg\ rotation} = \frac{1}{2} \times \frac{1}{RPMs} \times 60\ \text{sec/1\ min} \)
- **Transfer time** (Tavg transfer)
  - Time to read the bits in the target sector.
  - \( T_{avg\ transfer} = \frac{1}{RPMs} \times \frac{1}{(\text{avg # sectors/track})} \times 60\ \text{sec/1\ min} \)

**Logical Disk Blocks**

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2,...)
  - Mapping between logical blocks and actual (physical) sectors
    - Maintained by hardware/firmware device called \textit{disk controller}.
    - Converts requests for logical blocks into (surface, track, sector) triples.
  - Allows controller to set aside spare cylinders for each zone.
    - Accounts for the difference in “formatted capacity” and “maximum capacity”.

---

**I/O Bus**

- CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

**Reading a Disk Sector (1)**

- Expansion slots for other devices such as network adapters.
Reading a Disk Sector (2)

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special “interrupt” pin on the CPU).

Storage Trends

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM $/MB</td>
<td>19.20</td>
<td>2.900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>access (ns)</td>
<td>300</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>SRAM $/MB</td>
<td>8.000</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>8,000</td>
</tr>
<tr>
<td>access (ns)</td>
<td>375</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>typical size(MB)</td>
<td>0.064</td>
<td>0.256</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>1,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk $/MB</td>
<td>500</td>
<td>100</td>
<td>8</td>
<td>0.30</td>
<td>0.05</td>
<td>10,000</td>
</tr>
<tr>
<td>access (ns)</td>
<td>87</td>
<td>75</td>
<td>28</td>
<td>10</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>typical size(MB)</td>
<td>1</td>
<td>10</td>
<td>160</td>
<td>1,000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

CPU Clock Rates

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 4301</td>
<td>1</td>
<td>6</td>
<td>20</td>
<td>150</td>
</tr>
<tr>
<td>Pentium</td>
<td>150</td>
<td>750</td>
<td>750</td>
<td>750</td>
</tr>
<tr>
<td>Pentium III</td>
<td>750</td>
<td>750</td>
<td>750</td>
<td>750</td>
</tr>
<tr>
<td>cycle time(ns)</td>
<td>1,000</td>
<td>166</td>
<td>50</td>
<td>6</td>
</tr>
</tbody>
</table>

(Culled from back issues of Byte and PC Magazine)
Last step: Virtual Memory

- N bit address space...
- If word is not in main memory then it is on disk
  - need a controller to manage this...analogous to cache controller
- Virtual memory management system
  - Note: transfers take place from disk in “blocks” of M bytes (sector) – page of data
- Memory consists of a number of pages
  - Determine if the page is in main memory or not
- N bit address broken into fields which determine page number, and whether page is in the memory or disk
- If page size is 1024 bits...how to organize the N bit address ??

Main memory can act as a cache for the secondary storage (disk)

Advantages:
- illusion of having more physical memory
- program relocation
- protection

Pages: virtual memory blocks

- Page faults: the data is not in memory, retrieve it from disk
  - huge miss penalty, thus pages should be fairly large (e.g., 4KB)
  - reducing page faults is important (LRU is worth the price)
  - can handle the faults in software instead of hardware
  - using write-through is too expensive so we use writeback

Page Tables
Page Tables

More on Virtual memory

- This is part of the Operating system
  - CS 156 focus is the OS utilities