CSCI 2021: Memory Systems

Chris Kauffman

Last Updated:
Fri Apr 2 09:44:55 AM CDT 2021

Logistics

Reading Bryant/O'Hallaron

- Ch 4: Finish / Skim
- Ch 6: Memory

Goals

- Finish Arch
- 2D arrays
- Timing issues
- Memory efficient programs
- Permanent Storage

Assignments

- Lab 11: `clock()` function
- HW 11: Memory Optimization
- Both deal with memory layout affecting performance

Project 4

Later today

Architecture Performance

```
// LOOP 1
for(i=0; i<iters; i++){
  retA += delA;
  retB += delB;
}
*start = retA+retB;

// LOOP 2
for(i=0; i<iters; i++){
  retA += delA;
  retA += delB;
}
*start = retA;
```

- LOOP1 or LOOP2 faster?
- Why?

Exercise: 2D Arrays

- Several ways to construct "2D" arrays in C
- All must *embed* a 2D construct into 1-dimensional memory
- Consider the 2 styles below: how will the picture of memory look different?

```
// REPEATED MALLOC
// allocate
int rows=100, cols=30;
int **mat =
   malloc(rows * sizeof(int*));


for(int i=0; i<rows; i++){
  mat[i] = malloc(cols*sizeof(int));
}

// do work
mat[i][j] = ...

// free memory
for(int i=0; i<rows; i++){
  free(mat[i]);
}
free(mat);
```

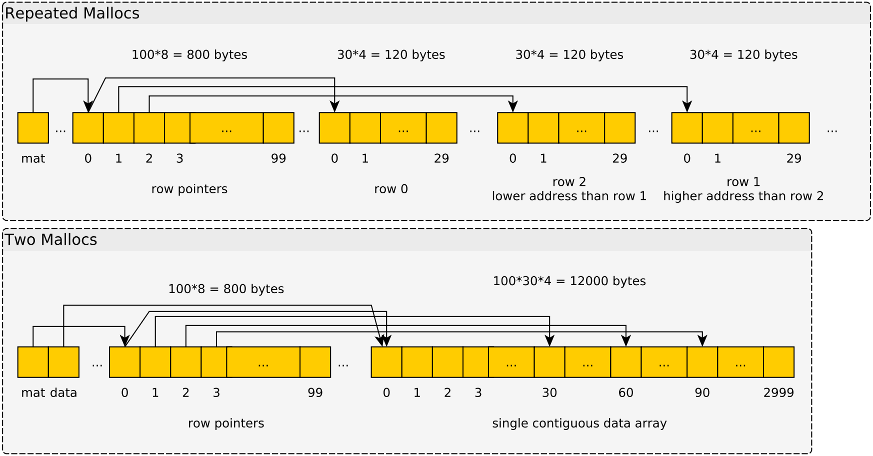
```
// TWO MALLOCs
// allocate
int rows=100, cols=30;
int **mat =
   malloc(rows * sizeof(int*));
int *data =
   malloc(rows*cols*sizeof(int));
for(int i=0; i<rows; i++){
  mat[i] = data+i*cols;
}

// do work
mat[i][j] = ...

// free memory
free(data);


free(mat);
```

Answer: 2D Arrays



Single Malloc Matrices

Somewhat common to use a 1D array as a 2D matrix as in

```
int *matrix =
   malloc(rows*cols*sizeof(int));

int i=5, j=20;
int elem_ij = matrix[ i*cols + j ]; // retrieve element i,j
```

P4 will use this technique along with some structs and macros to make it more readable:

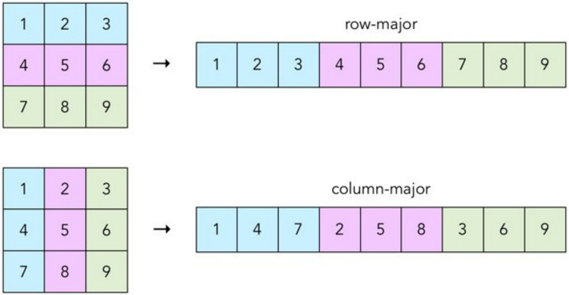
```
matrix_t mat;
matrix_init(&mat, rows, cols);

int elij = MGET(mat,i,j);
//  elij = mat.data[ mat.cols*i + j]

MSET(mat,i,j, 55);
// mat.data[ mat.cols*i + j ] = 55;
```

Aside: Row-Major vs Col-Major Layout

- Many languages use **Row-Major** order for 2D arrays/lists
  - C, Java, Python, Ocaml,...
  - `mat[i]` is a contiguous row, `mat[i][j]` is an element
- Numerically-oriented languages use **Column-Major** order
  - Fortran, Matlab/Octave, R, Ocaml (?)...
  - `mat[j]` is a contiguous **column**, `mat[i][j]` is an element
- Being aware of language convention can increase efficiency



Source: The Craft of Coding

Exercise: Matrix Summing

- How are the two codes below different?
- Are they doing the same number of operations?
- Which will run faster?

```
int sumR = 0;
for(int i=0; i<rows; i++){
  for(int j=0; j<cols; j++){
    sumR += mat[i][j];
  }
}
```

```
int sumC = 0;
for(int j=0; j<cols; j++){
  for(int i=0; i<rows; i++){
    sumC += mat[i][j];
  }
}
```

Answer: Matrix Summing

- Show timing in `matrix_timing.c`
- sumR faster the sumC: caching effects
- Discuss timing functions used to determine duration of runs

```
> gcc -Og matrix_timing.c
> a.out 50000 10000
sumR: 1711656320 row-wise CPU time: 0.265 sec, Wall time: 0.265
sumC: 1711656320 col-wise CPU time: 1.307 sec, Wall time: 1.307
```

- `sumR` runs about 6 times faster than `sumC`
- Understanding why requires knowledge of the memory hierarchy and cache behavior

Measuring Time in Code

- Measure CPU time with the standard `clock()` function; measure time difference and convert to seconds
- Measure Wall (real) time with `gettimeofday()` or related functions; fills struct with info on time of day (duh)

CPU Time

```
#include <time.h>

clock_t begin, end;
begin = clock(); // current cpu moment

do_something();

end = clock();   // later moment

double cpu_time =
  ((double) (end-begin)) / CLOCKS_PER_SEC;
```

Real (Wall) Time

```
#include <sys/time.h>

struct timeval tv1, tv2;
gettimeofday(&tv1, NULL); // early time

do_something();

gettimeofday(&tv2, NULL); // later time

double wall_time =
  ((tv2.tv_sec-tv1.tv_sec)) +
  ((tv2.tv_usec-tv1.tv_usec) / 1000000.0);
```

Tools to Measure Performance: `perf`

- The Linux `perf` tool is useful to measure performance of an entire program
- Shows variety of statistics tracked by the kernel about things like memory performance
- **Examine** examples involving the `matrix_timing` program: `sumR` vs `sumC`
- **Determine** statistics that explain the performance gap between these two?

Exercise: `perf` stats for sumR vs sumC, what's striking?

```
> perf stat $perfopts ./matrix_timing 8000 4000 row    ## RUN sumR ROW SUMMING
sumR: 1227611136 row-wise CPU time: 0.019 sec, Wall time: 0.019
Performance counter stats for './matrix_timing 8000 4000 row':      %SAMPLED
135,161,407  cycles:u                                               (45.27%)
417,889,646  instructions:u        #  3.09  insn per cycle          (56.22%)
 56,413,529  L1-dcache-loads:u                                      (55.96%)
  3,843,602  L1-dcache-load-misses:u #  6.81% of all L1-dcache hits  (50.41%)
 28,153,429  L1-dcache-stores:u                                     (47.42%)
        125  L1-icache-load-misses:u                                (44.77%)
  3,473,211  cache-references:u    # last level of cache            (56.22%)
  1,161,006  cache-misses:u        # 33.427 % of all cache refs     (56.22%)

> perf stat $perfopts ./matrix_timing 8000 4000 col   # RUN sumC COLUMN SUMMING
sumC: 1227611136 col-wise CPU time: 0.086 sec, Wall time: 0.086
Performance counter stats for './matrix_timing 8000 4000 col':       %SAMPLED
372,203,024   cycles:u                                               (40.60%)
404,821,793   instructions:u        #  1.09  insn per cycle          (57.23%)
 61,990,626   L1-dcache-loads:u                                      (60.21%)
 39,281,370   L1-dcache-load-misses:u # 63.37% of all L1-dcache hits  (45.66%)
 23,886,332   L1-dcache-stores:u                                     (43.24%)
      2,486   L1-icache-load-misses:u                                (40.82%)
 32,582,656   cache-references:u    # last level of cache            (59.38%)
  1,894,514   cache-misses:u        #  5.814 % of all cache refs     (60.38%)
```

Answers: `perf` stats for sumR vs sumC, what's striking?

Observations

- Similar number of instructions between row/col versions
- `#cycles` lower for row version → higher `insn per cycle`
- **L1-dcache-misses**: marked difference between row/col version
- **Last Level Cache Refs** : many, many more in col version
- Col version: much time spent waiting for memory system to feed in data to the processor

Notes

- The right-side percentages like (50.41%) indicate how much of how much of the time this feature is measured; some items can't be monitored all the time.
- Specific `perf` invocation is in `10-memory-systems-code/measure-cache.sh`

Exercise: Time and Throughput

Consider the following simple loop to sum elements of an array from `stride_throughput.c`

```
int *data = ...;   // global array
int sum_simple(int len, int stride){
  int sum = 0;
  for(int i=0; i<len; i+=stride)
  {
    sum += data[i];
  }
  return sum;
}
```

- Param `stride` controls step size through loop
- Interested in two features of the `sum_simple()` function:

  1. Total Time to complete
  2. **Throughput**:

  $$Throughput = \frac{\#Additions}{Second}$$

- How would one measure and calculate these two in a program?
- As stride increases, predict how Total Time and Throughput change

Answers: Time and Throughput

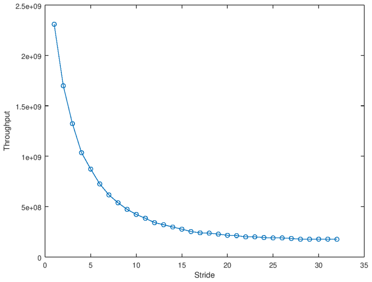
Measuring Time/Throughput

Most interested in CPU time so

```
begin = clock();
sum_simple(length,stride);
end = clock();
cpu_time = ((double) (end-begin))
            / CLOCKS_PER_SEC;

throughput = ((double) length) /
             stride /
             cpu_time;
```

Plot of Stride vs Throughput



Time vs Throughput

As `stride` increases...

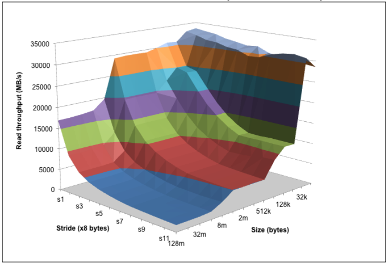
- Time decreases: doing fewer additions (duh)
- Throughput **decreases**

- Stride = 1: consecutive memory accesses
- Stride = 16: jumps through memory, more time

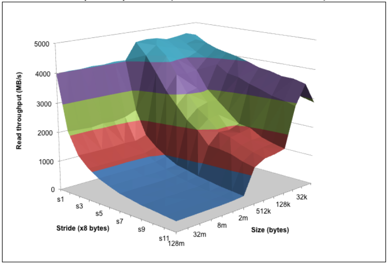
Memory Mountains from Bryant/O'Hallaron

- Varying `stride` for a fixed `length` leads to decreasing performance, 2D plot
- Can also vary `length` for size of array to get a 3D plot
- Illustrates features of CPU/memory on a system
- The "Memory Mountain" on the cover of our textbook
- What **interesting structure** do you see?

CS:APP3e: Core i5-4440 (2013 Haswell)



Raspberry Pi 3B (2016 ARM Cortex-A53)



Increasing Efficiency

- Can increase the efficiency of loop summing with tricks
- B/O'H use multiple *accumulators*: multiple variables for summing
- Facilitates pipelining / superscalar processor
- Code is significantly faster BUT much trickier and less readable
- May be compiler options which enable this but not with defaults in `gcc -O3` (try searching optimization options, ~67 pages)

```
// From Bryant/O'Hallaron
int sum_add4(int elems, int stride){
  int i,
    sx1 = stride*1, sx2 = stride*2,
    sx3 = stride*3, sx4 = stride*4,
    acc0 = 0, acc1 = 0,
    acc2 = 0, acc3 = 0;
  int length = elems;
  int limit = length - sx4;

  /* Combine 4 elements at a time */
  for (i = 0; i < limit; i += sx4) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+sx1];
    acc2 = acc2 + data[i+sx2];
    acc3 = acc3 + data[i+sx3];
  }

  /* Finish any remaining elements */
  for (; i < length; i += stride) {
    acc0 = acc0 + data[i];
  }
  return acc0+acc1+acc2+acc3;
}
```

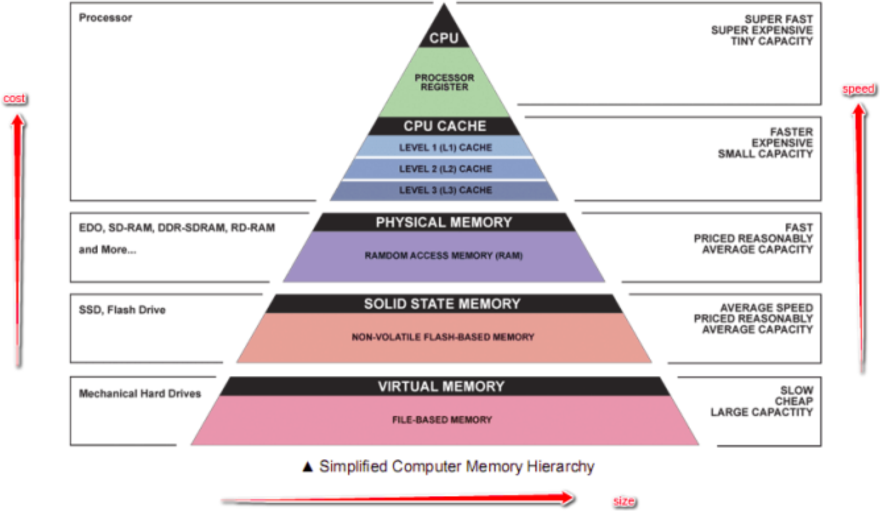
Temporal and Spatial Locality

- In the beginning, there was only CPU and Memory
- Both ran at about the same speed (same clock frequency)
- CPUs were easier to make faster, began outpacing speed of memory
- Hardware folks noticed programmers often write loops like

```
for(int i=0; i<0; i++){
  sum += array[i];
}
```

- Led to development of faster memories exploit Locality
- **Temporal Locality**: memory recently used likely to be used again soon
- **Spatial Locality**: memory near to recently used memory likely to be used
- Register file and Cache were developed to exploit this: faster memory that is automatically managed

The Memory Pyramid



▲ Simplified Computer Memory Hierarchy

Source

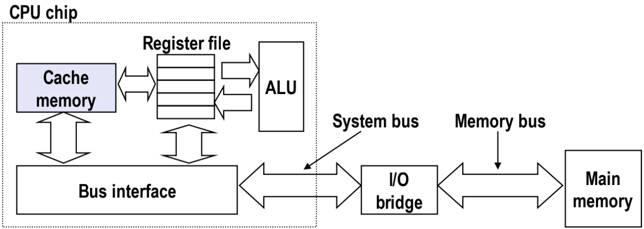
Numbers Everyone Should Know

Edited Excerpt of Jeff Dean's talk on data centers.

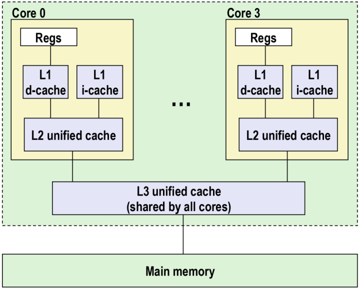
Reference	Time	Analogy
Register	-	Your brain
L1 cache reference	0.5 ns	Your desk
L2 cache reference	7 ns	Neighbor's Desk
Main memory reference	100 ns	This Room
Disk seek	10,000,000 ns	Salt Lake City

Big-O Analysis does NOT capture these; proficient programmers do

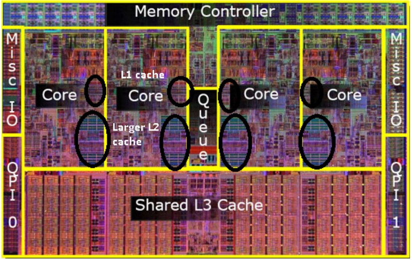
Diagrams of Memory Interface and Cache Levels



Source: Bryant/O'Hallaron CS:APP 3rd Ed.



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Source: SO "Where exactly L1, L2 and L3 Caches located in computer?"

Why isn't Everything Cache?

Metric	1985	1990	1995	2000	2005	2010	2015	2015/1985
SRAM $/MB	2,900	320	256	100	75	60	25	116
SRAM access (ns)	150	35	15	3	2	1.5	1.3	115
DRAM $/MB	880	100	30	1	0.1	0.06	0.02	44,000
DRAM access (ns)	200	100	70	60	50	40	20	10

Source: Bryant/O'Hallaron CS:APP 3rd Ed., Fig 6.15, pg 603

1 bit SRAM = 6 transistors

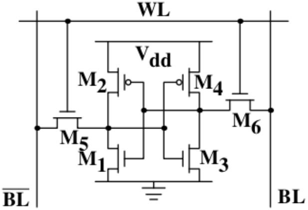


Figure 2.4: 6-T Static RAM

1 bit DRAM = 1 transistor + 1 capacitor

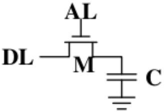


Figure 2.5: 1-T Dynamic RAM

"What Every Programmer Should Know About Memory" by Ulrich Drepper, Red Hat, Inc.

Cache Principles: Hits and Misses

CPU-Memory is a Client-Server

- CPU makes requests
- Memory system services request as fast as possible

Cache Hit

- CPU requests memory at address `0xFFFF1234` be loaded into register `%rax`
- **Finds** valid data for `0xFFFF1234` in L1 Cache: **L1 Hit**
- Loads into register fast

Cache Miss

- CPU requests memory at address `0xFFFF7890` be loaded into register `%rax`
- `0xFFFF7890` **not in** L1 Cache: **L1 Miss**
- Search L2: if found move into L1, then `%rax`
- Search L3: if found move into L2, L1, `%rax`
- Search main memory: if found, move into caches, if not...

Wait, how could `0xFFFF7890` not be in main memory... ?

Types of Cache Misses

Compulsory "Cold" Miss: Getting Started

- First time accessing an element in a program
- After the cache "warms up" hopefully doesn't happen much

Capacity Miss: Too Big to Fit

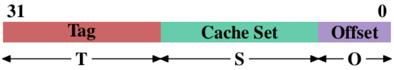
- **Working set** is set of memory being frequently accessed in a phase of a program
- Large working set may exceed the size of a cache

Conflict Miss: This Stall Occupied

- Internal **placement policies** of cache lead to a conflict
- Two pieces of memory in working set both want to reside at the same location but cannot
- Makes more sense after discussing placement policies

Memory Address Determines Location in a Cache

- Cache specified by
  - # of **Sets** S
  - # of **Lines** per Set E
  - # of bytes in a **Block** B
- Each Line in a Set has a **Tag**
- Combination of (tag+set) uniquely identifies a Block in memory
- To determine whether memory address A is in cache, check each line in associated Set for its Tag
- Specific byte will be at an **Offset** in cache block



Address Bits to Cache Location

- Bits from address determine location for memory in cache
- Direct-Mapped cache, 4 sets and 16 byte blocks/lines
- Load address 0x28

```
       0  2  8
0x28 = 00 10 1000
       |  |  |
       |  |  +-> Offset: 4 bits
       |  +-> Set: 2 bits
       +-> Tag: Remaining bits
```

- 0x20 in the same line, will also be loaded int set #2

Exercises: Anatomy of a Simple CPU Cache

```
MAIN MEMORY
| Addr | Addr Bits        | Value | Tag | Set | Off |
|------+------------------+-------+-----+-----+-----|
| 00   | 00  00  0000     |   331 |  00 |   0 |   0 |
| 08   | 00  00  1000     |   332 |  00 |   0 |   8 |
| 10   | 00  01  0000     |   333 |  00 |   1 |   0 |
| 18   | 00  01  1000     |   334 |  00 |   1 |   8 |
| 20   | 00  10  0000     |   335 |  00 |   2 |   0 |
| 28   | 00  10  1000     |   336 |  00 |   2 |   8 |
| 30   | 00  11  0000     |   337 |  00 |   3 |   0 |
| 38   | 00  11  1000     |   338 |  00 |   3 |   8 |
|      | ...              |       |     |     |     |
| C0   | 11  00  0000     |   551 |  11 |   0 |   0 |
| C8   | 11  00  1000     |   552 |  11 |   0 |   8 |
| D0   | 11  01  0000     |   553 |  11 |   1 |   0 |
| D8   | 11  01  1000     |   554 |  11 |   1 |   8 |
| E0   | 11  10  0000     |   555 |  11 |   2 |   0 |
| E8   | 11  10  1000     |   556 |  11 |   2 |   8 |
| F0   | 11  11  0000     |   557 |  11 |   3 |   0 |
| F8   | 11  11  1000     |   558 |  11 |   3 |   8 |
|------+------------------+-------+-----+-----+-----|
|      | Tag Set Offset   |       |     |     |     |
```

```
CACHE
|     |   |     | Blocks/Line |
| Set | V | Tag | 0-7   8-15  |
|-----+---+-----+-------------|
| 00  | 0 |   - | -           |
| 01  | 1 |  00 | 333   334   |
| 10  | 1 |  11 | 555   556   |
| 11  | 1 |  00 | 337   338   |
|-----+---+-----+-------------|
|     |   |     | 0-7   8-15  |
DIRECT-MAPPED Cache
- Direct-mapped: 1 line per set
- 16-byte lines = 4-bit offset
- 4 Sets = 2-bit index
- 8-bit Address = 2-bit tag
- Total Cache Size = 64 bytes
    4 sets * 16 bytes

HITS OR MISSES? Show effects
1. Load 0x08
2. Load 0xF0
3. Load 0x18
```

Answers: Anatomy of a Simple CPU Cache

```
MAIN MEMORY
| Addr | Addr Bits        | Value | Tag | Set | Off |
|------+------------------+-------+-----+-----+-----|
| 00   | 00  00  0000     |   331 | 00  |   0 |   0 |
| 08   | 00  00  1000     |   332 | 00  |   0 |   8 |
| 10   | 00  01  0000     |   333 | 00  |   1 |   0 |
| 18   | 00  01  1000     |   334 | 00  |   1 |   8 |
| 20   | 00  10  0000     |   335 | 00  |   2 |   0 |
| 28   | 00  10  1000     |   336 | 00  |   2 |   8 |
| 30   | 00  11  0000     |   337 | 00  |   3 |   0 |
| 38   | 00  11  1000     |   338 | 00  |   3 |   8 |
|      | .. .             |       |     |     |     |
| C0   | 11  00  0000     |   551 | 11  |   0 |   0 |
| C8   | 11  00  1000     |   552 | 11  |   0 |   8 |
| D0   | 11  01  0000     |   553 | 11  |   1 |   0 |
| D8   | 11  01  1000     |   554 | 11  |   1 |   8 |
| E0   | 11  10  0000     |   555 | 11  |   2 |   0 |
| E8   | 11  10  1000     |   556 | 11  |   2 |   8 |
| F0   | 11  11  0000     |   557 | 11  |   3 |   0 |
| F8   | 11  11  1000     |   558 | 11  |   3 |   8 |
|------+------------------+-------+-----+-----+-----|
|      | Tag Set Offset   |       |     |     |     |
```

```
CACHE
|     |   |     | Blocks/Line |
| Set | V | Tag | 0-7   8-15  |
|-----+---+-----+-------------|
| 00  | 1 | *00 | 331   332   |
| 01  | 1 |  00 | 333   334   |
| 10  | 1 |  11 | 555   556   |
| 11  | 1 | *11 | 557   558   |
|-----+---+-----+-------------|
|     |   |     | 0-7   8-15  |
DIRECT-MAPPED Cache
- Direct-mapped: 1 line per set
- 16-byte lines = 4-bit offset
- 4 Sets = 2-bit index
- 8-bit Address = 2-bit tag
- Total Cache Size = 64 bytes
    4 sets * 16 bytes

HITS OR MISSES? Show effects
1. Load 0x08: MISS to set 00
2. Load 0xF0: MISS overwrite
              set 11
3. Load 0x18: HIT  in set 01
              no change
```

Direct vs Associative Caches

Direct Mapped

One line per set

```
|     |   |     | Blocks/Line |
| Set | V | Tag | 0-7   8-15  |
|-----+---+-----+-------------|
| 00  | 0 |   - | -           |
| 01  | 1 |  00 | 333   334   |
| 10  | 1 |  11 | 555   556   |
| 11  | 1 |  00 | 337   338   |
|-----+---+-----+-------------|
|     |   |     | 0-7   8-15  |
```

- Simple circuitry
- **Conflict misses** may result: 1 slot for many possible tags
- **Thrashing:** need memory with overlapping tags

```
           vv
0x10 = 00 01 0000 : in cache
0xD8 = 11 01 1000 : conflict
           ^^
```

N-Way Associative Cache

Ex: 2-way = 2 lines per set

```
|     |   |     | Blocks      |
| Set | V | Tag | 0-7   8-15  |
|-----+---+-----+-------------|
| 00  | 0 |   - | -           | Line1
|     | 1 |  11 | 551   552   | Line2
|-----+---+-----+-------------|
| 01  | 1 |  00 | 333   334   | Line1
|     | 1 |  11 | 553   554   | Line2
|-----+---+-----+-------------|
| 10  | 1 |  11 | 555   556   | Line1
|     | 0 |   - | -           | Line2
|-----+---+-----+-------------|
| 11  | 1 |  00 | 337   338   | Line1
|     | 1 |  11 | 557   558   | Line2
|-----+---+-----+-------------|
|     |   |     | 0-7   8-15  |
```

- Complex circuitry → $$
- Requires an **eviction policy**, usually least recently used

How big is your cache? Check Linux System special Files

`lscpu` Utility

Handy Linux program that summarizes info on CPU(s)

```
> lscpu
Architecture:   x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order:     Little Endian
Address sizes:  36 bits physical,
                48 bits virtual
CPU(s):         4
Vendor ID:      GenuineIntel
CPU family:     6
Model:          58
Model name:     Intel(R) Core(TM)
                i7-3667U CPU @ 2.00GHz
...
L1d cache:      64 KiB
L1i cache:      64 KiB
L2 cache:       512 KiB
L3 cache:       4 MiB
Vulnerability Meltdown: Mitigation; ...
Vulnerability Spectre v1: Mitigation ...
...
```

Detailed Hardware Info

Files under `/sys/devices/...` show hardware info (caches)

```
> cd /sys/devices/system/cpu/cpu0/cache/
> ls
index0  index1  index2  index3 ...

> ls index0/
number_of_sets  type  level  size
ways_of_associativity ...

> cd index0
> cat level type number_* ways_* size
1 Data 64 8 32K

> cd ../index1
> cat level type number_* ways_* size
1 Instruction 64 8 32K

> cd ../index3
> cat level type number_* ways_* size
3 Unified 8192 20 10240K
```

Disks: Persistent Block Storage

- Have discussed a variety of fast memories which are **small**
- At the bottom of the pyramid are **disks**: slow but **large** memories
- These are **persistent**: when powered off, they retain information

Using Disk as Main Memory

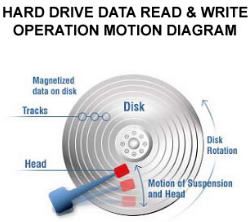
- Operating Systems can create the illusion that main memory is larger than it is in reality
- Ex: 2 GB DRAM + 6 GB of disk space = 8 GB Main Memory
- Disk file is called **swap** or a **swap file**
- Naturally much slower than RAM so OS will try to limit its use
- A **Virtual Memory** system manages RAM/Disk as main memory, will discuss later in the course

Flavors of Permanent Storage

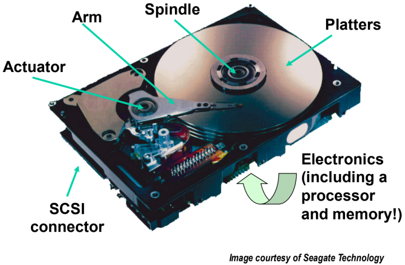
- Permanent storage often referred to as a "drive"
- Comes in many variants but these 3 are worth knowing about in the modern era
  1. Rotating Disk Drive
  2. Solid State Drive
  3. Magnetic Tape Drive
- Surveyed in the slides that follow

Ye Olde Rotating Disk

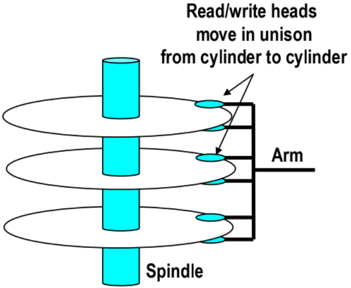
- Store bits “permanently” as magnetized areas on special platters
- Magnetic disks: moving parts $\rightarrow$ slow
- Cheap per GB of space



Source: Realtechs.net



Source: CS:APP Slides



Rotating Disk Drive Features of Interest

Measures of Quality

- Capacity: bigger is usually better
- Seek Time: delay before a head assembly reaches an arbitrary track of the disk that contains data
- Rotational Latency: time for disk to spin around to correct position; faster rotation $\rightarrow$ lower Latency
- Transfer Rate: once correct read/write position is found, how fast data moves between disk and RAM

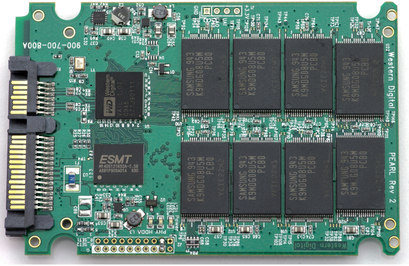
Sequential vs Random Access

Due to the rotational nature of Magnetic Disks...

- Sequential reads/writes comparatively FAST
- Random reads/writes comparatively very SLOW

Solid State Drives

- No moving parts → speed
- Most use "flash" memory, non-volatile circuitry
- Major drawback: limited number of **writes**, disk wears out eventually



- Reads faster than writes
- Sequential somewhat faster than random access
- **Expensive:**

  A 1TB internal 2.5-inch hard drive costs between $40 and $50, but as of this writing, an SSD of the same capacity and form factor starts at $250. That translates into

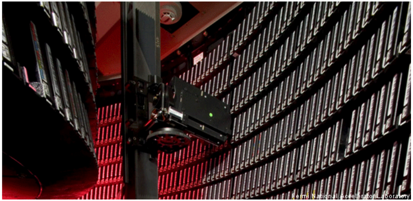
  – 4 to 5 cents/GB for HDD

  – 25 cents/GB for the SSD.

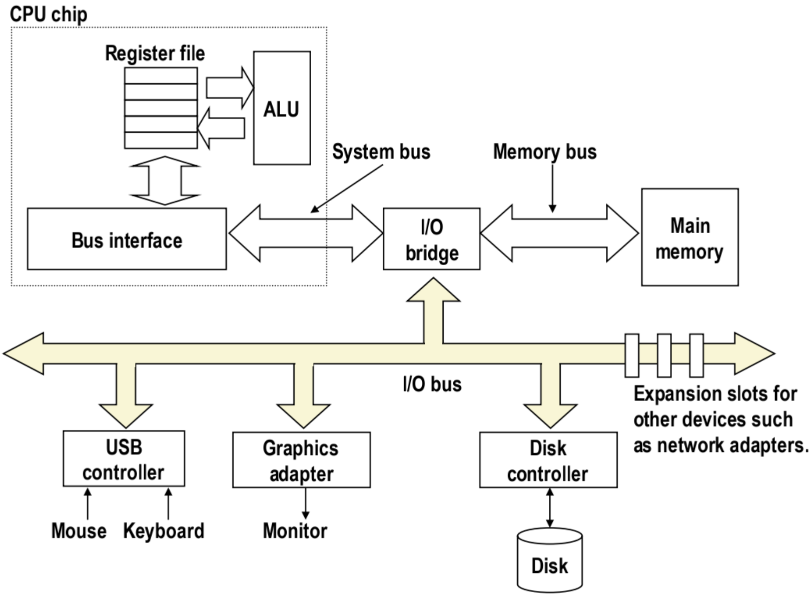
  PC Magazine, "SSD vs HDD" by Tom Brant and Joel Santo Domingo March 26, 2018

Tape Drives

- Slowest yet: store bits as magnetic field on a piece of “tape” a la 1980’s cassette tape / video recorder
- Extremely cheap per GB so mostly used in backup systems
- Ex: CSELabs does nightly backups of home directories, recoverable from tape at request to Operator



The I/O System Connects CPU and Peripherals



Terminology

Bus A collection of wires which allow communication between parts of the computer. May be serial (single wire) or parallel (several wires), must have a communication protocol over it.

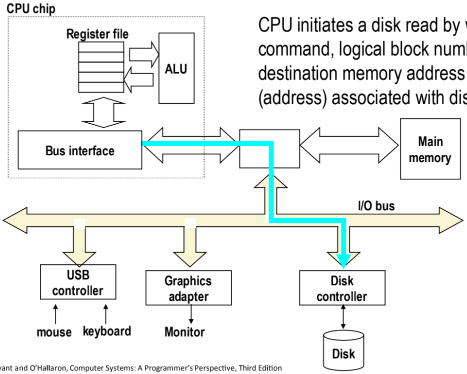
Bus Speed Frequency of the clock signal on a particular bus, usually different between components/buses requiring interface chips
CPU Frequency $>$ Memory Bus $>$ I/O Bus

Interface/Bridge Computing chips that manage communications across the bus possibly routing signals to correct part of the computer and adapting to differing speeds of components

Motherboard A printed circuit board connects to connect CPU to RAM chips and peripherals. Has buses present on it to allow communication between parts. *Form factor* dictates which components can be handled.

Memory Mapped I/O

- Modern systems are a collection of devices and microprocessors
- CPU usually uses **memory mapped I/O**: read/write certain memory addresses translated to communication with devices on I/O bus

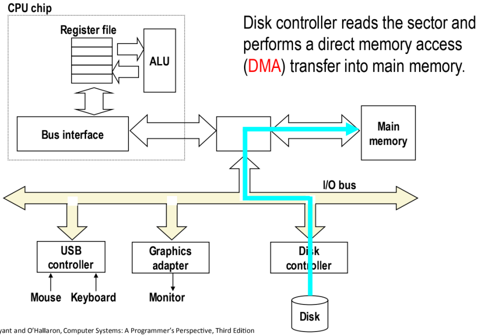


CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Direct Memory Access

- Communication received by *other* microprocessors like a Disk Controller or Memory Management Unit (MMU)
- Other controllers may talk: Disk Controller loads data directly into Main Memory via **direct memory access**



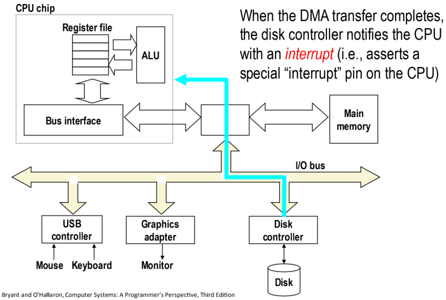
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Interrupts and I/O

Recall access times

Place	Time
L1 cache	0.5 ns
RAM	100 ns
Disk	10,000,000 ns

- While running Program X, CPU reads an `int` from disk into `%rax`
- Communicates to disk controller to read from file
- Rather than wait, OS puts Program X to "sleep", starts running program Y



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

- When disk controller completes read, signals the CPU via an **interrupt**, electrical signals indicating an event
- OS handles interrupt, schedules Program X as "ready to run"

Interrupts from Outside and Inside

- Examples of events that generate interrupts
  - Integer divide by 0
  - I/O Operation complete
  - Memory address not in RAM (Page Fault)
  - User generated: x86 instruction `int 80`
- Interrupts are mainly the business of the Operating System
- Usually cause generating program to immediately transfer control to the OS for handling
- When building your own OS, must write "interrupt handlers" to deal with above situations
  - Divide by 0: **signal** program usually terminating it
  - I/O Complete: schedule requesting program to run
  - Page Fault: sleep program until page loaded
  - User generated: perform system call
- User-level programs will sometimes get a little access to interrupts via **signals**, a topic for CSCI 4061