CMSC216: Memory Systems

Chris Kauffman

Last Updated:
Tue Nov 14 03:02:14 PM EST 2023

Logistics

Assignments

- Lab/HW 11: Timing Memory Strides
- P4 Up: Shellac, video forthcoming

Goals

- Overview of Memory System / Hierarchy
- Writing Cache-friendly code for speed
- Physical Devices for Memory
- Virtual Memory and Address translation

Reading Bryant/O'Hallaron

Ch	Read?	Topic
Ch 6		**The Memory Hierarchy**
Ch 6.1	skim	Storage Technologies
Ch 6.2	READ	Locality
Ch 6.3	READ	The Memory Hierarchy
Ch 6.4	opt	Cache Memories
Ch 6.5	READ	Writing Cache Friendly Code
Ch 6.6	skim	Impacts of Cache on Performance
Ch 9		**Virtual Memory**
Ch 9.1-6	skim	VM Overview, Address Translation
Ch 9.7	opt	Case Study
Ch 9.8	READ	Memory mapping and `mmap()`
Ch 9.9	READ	Dynamic Memory Allocation
Ch 9.10	opt	Garbage Collection
Ch 9.11	skim	Memory Bugs in C Programs

Announcements

None

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);
```

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;
}

int main(){
  ...;
  int x1 = sum_simple(n,1);
  int x2 = sum_simple(n,2);
  int x3 = sum_simple(n,3);
  // total time for each stride?
  // throughput for each stride?
}
```

- 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

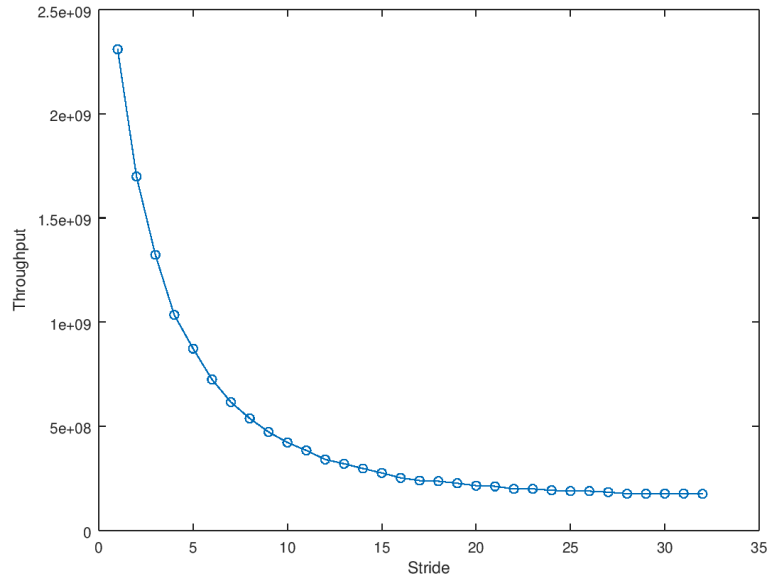
```
begin = clock();
sum_simple(length,stride);
end = clock();
cpu_time = ((double) (end-begin))
            / CLOCKS_PER_SEC;

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

Time vs Throughput

As `stride` increases...

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

Plot of Stride vs Throughput

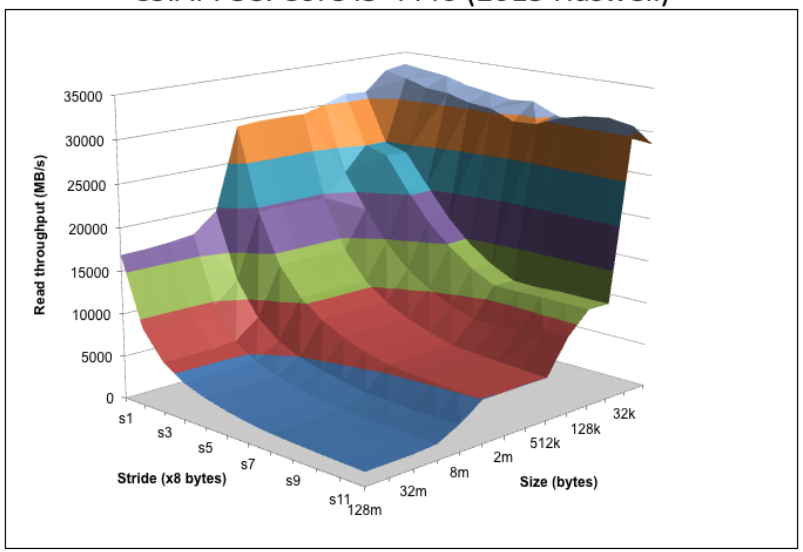


- Stride = 1: consecutive memory accesses
- Stride = 16: jumps through memory, more time

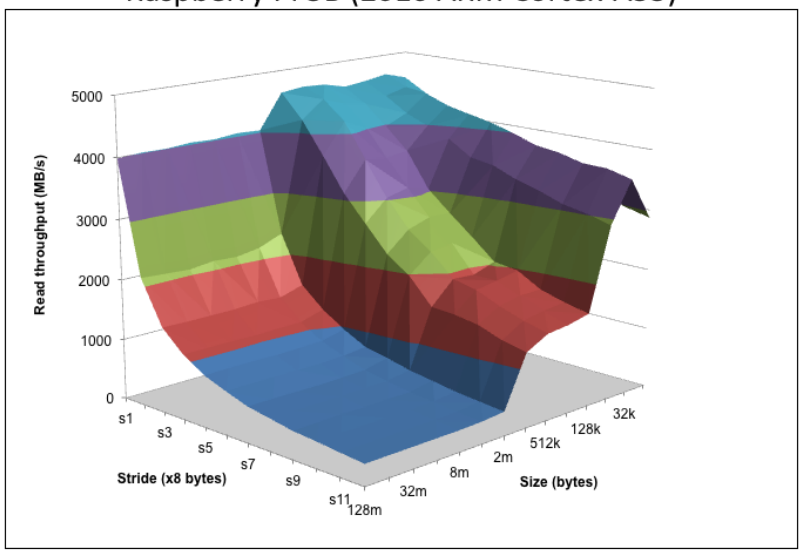
Memory Mountains from Bryant/O'Hallaron

- 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)



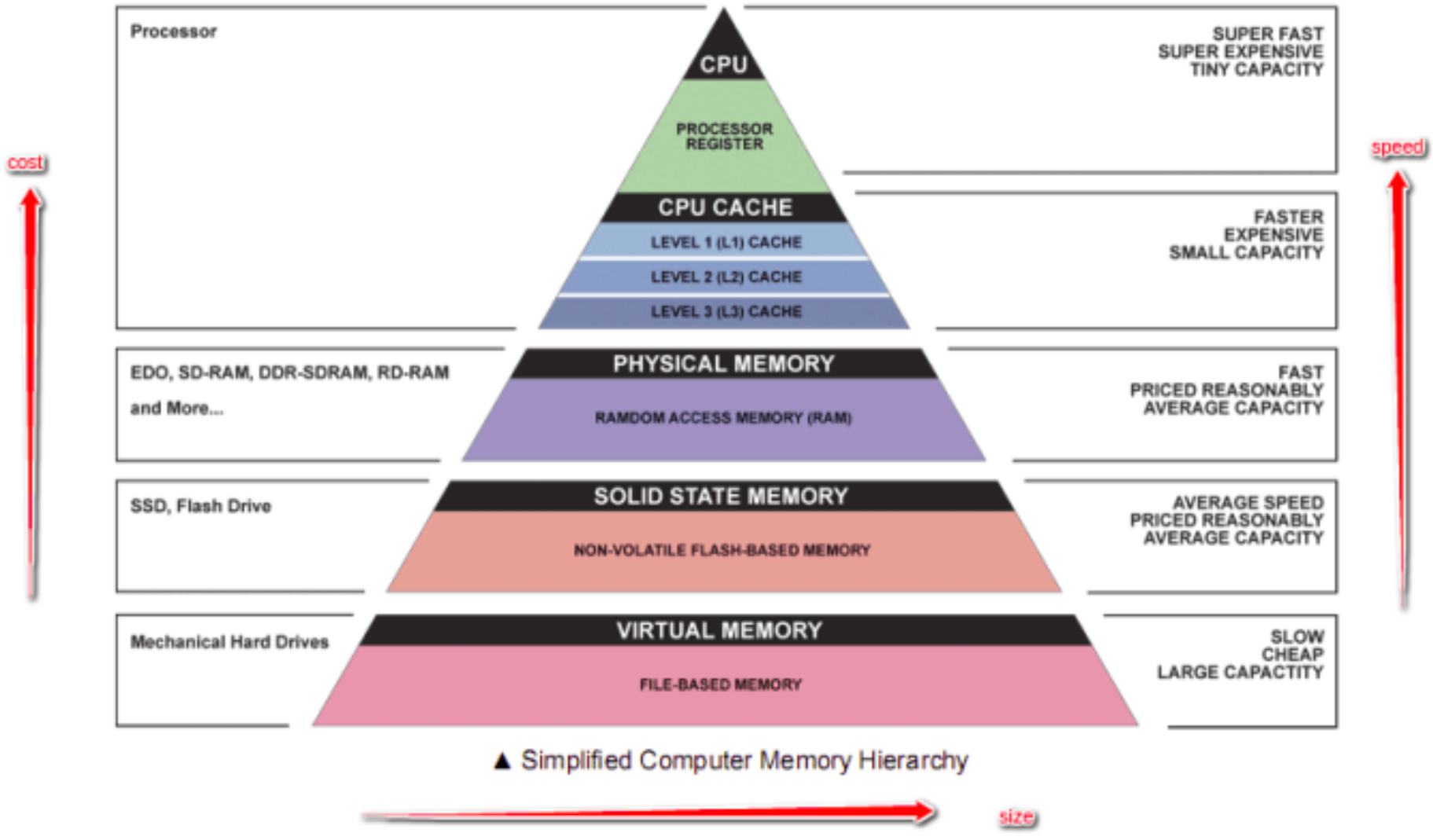
Cache Favors Temporal and Spatial Locality

- 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<len; i++){
  sum += array[i];
}
```

  which exhibits two Memory Locality features
  - **Temporal Locality**: memory recently used likely to be used again soon (like `sum` and `i` used in every loop iteration)
  - **Spatial Locality**: memory near to recently used memory likely to be used (like `arr[0]` first then `arr[1]`,`arr[2]`)
- Register file and Cache were developed to exploit locality

The Memory Pyramid



▲ Simplified Computer Memory Hierarchy

Source

Numbers Everyone Should Know

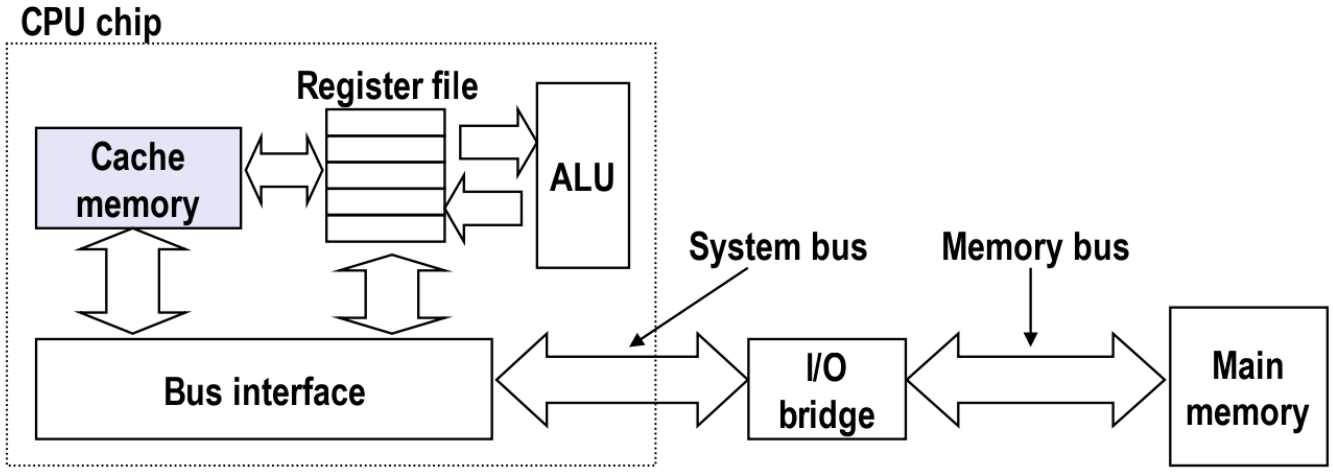
- "Main Memory" is comprised of many different physical devices that work together and have differing sizes/speeds
- Accessing memory at #4096 may involve some or all of...
  - Several Levels of Cache Memory on CPU (SRAM)
  - DRAM memory on separate chips
  - Permanent storage (SSDs and HDDs)

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

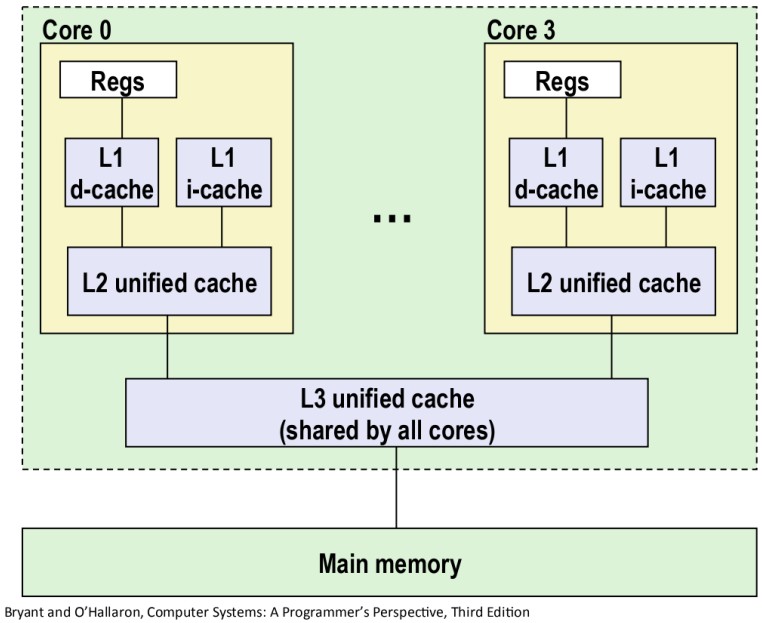
Reference	Time	Analogy
Register	-	Your brain
L1 cache reference	0.5 ns	Your desk
L2 cache reference	7 ns	Neighbor's Desk
DRAM memory reference	100 ns	This Room
Disk seek	10,000,000 ns	Salt Lake City

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

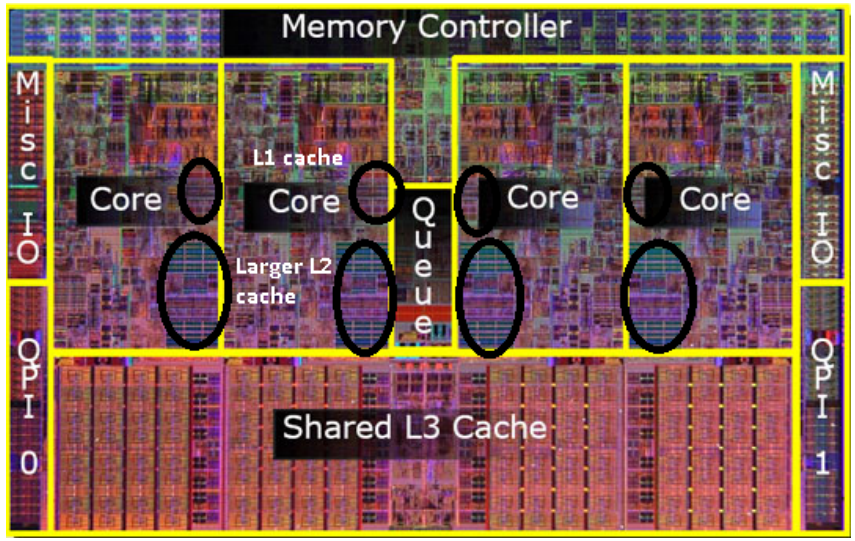
Diagrams of Memory Interface and Cache Levels



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

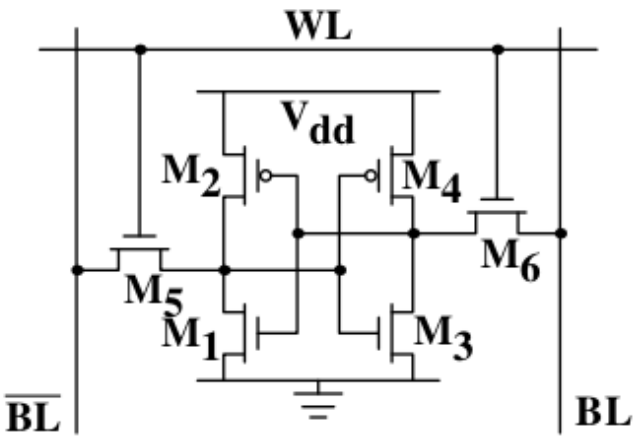


Figure 2.4: 6-T Static RAM

1 bit DRAM = 1 transistor + 1 capacitor

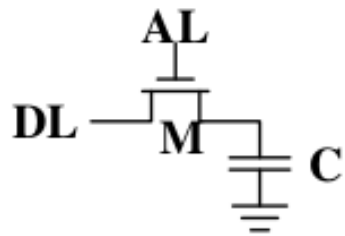
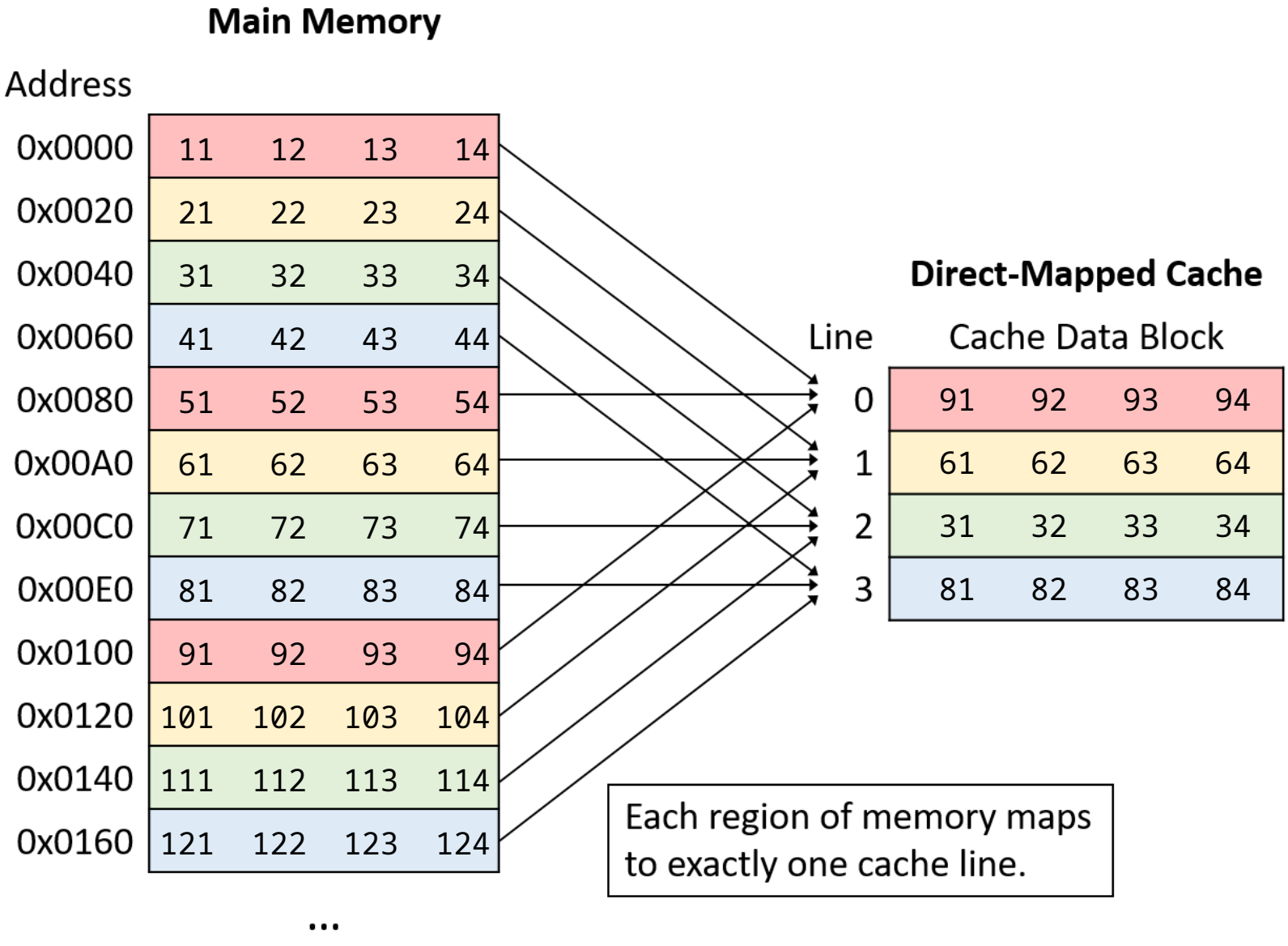


Figure 2.5: 1-T Dynamic RAM

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

Diagram of Direct Mapped Cache



Source: Dive into Systems dot org, with modifications

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
```

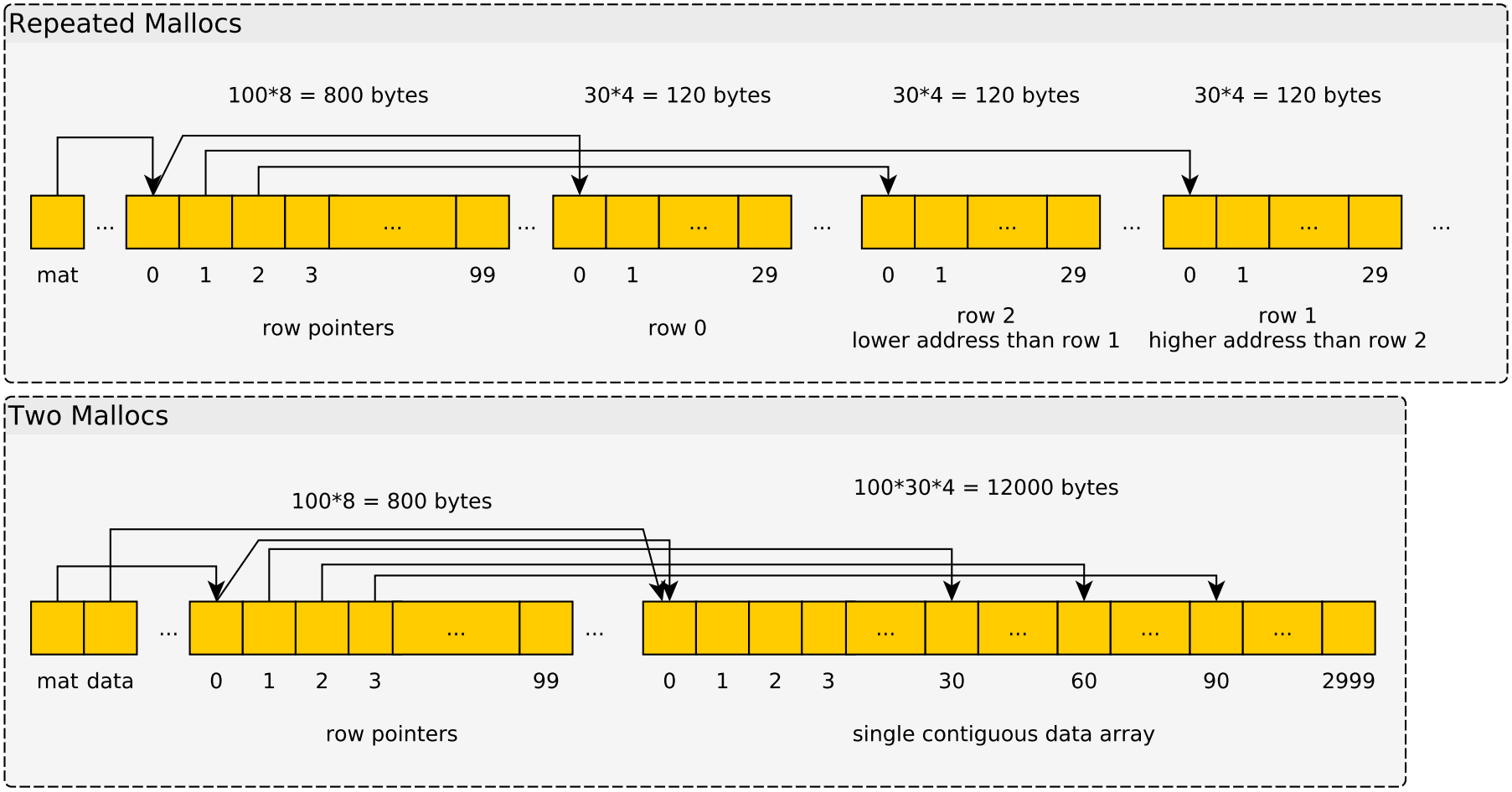
Exercise: 2D Arrays

- 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

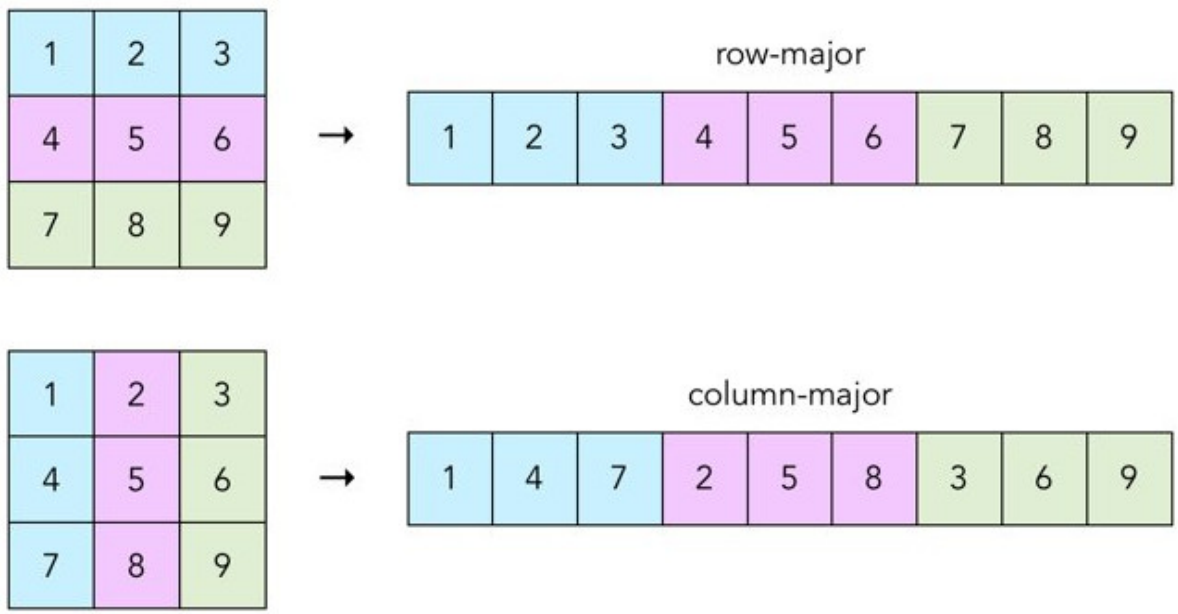
```
int *matrix =
   malloc(rows*cols*sizeof(int));

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

HWs / Labs / 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

(Optional) 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?

(Optional 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

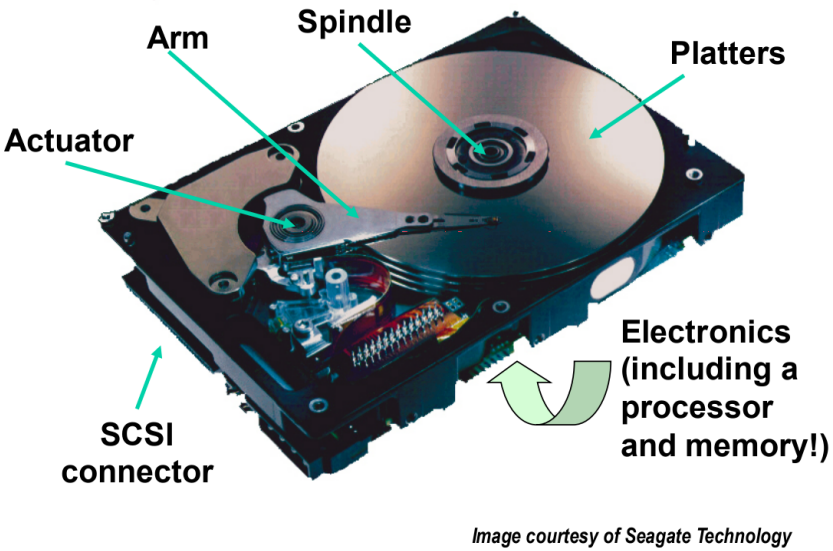
- The right-side percentages like (50.41%) indicate how much of the time this feature is measured; some items can't be monitored all the time.
- Specific `perf` invocation is in `10-memory-systems-code/measure-cache.sh`

Flavors of Permanent Storage

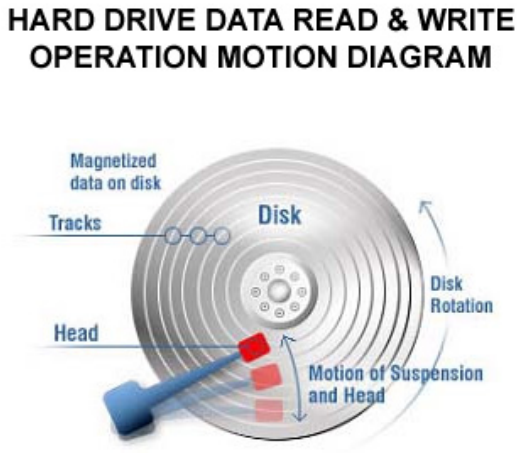
- Have discussed a variety of fast memories which are **small**
- At the bottom of the pyramid are **disks**: slow but **large** memories, may contain copies of what is in higher parts of memory pyramid
- These are **persistent**: when powered off, they retain information
- Permanent storage often referred 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

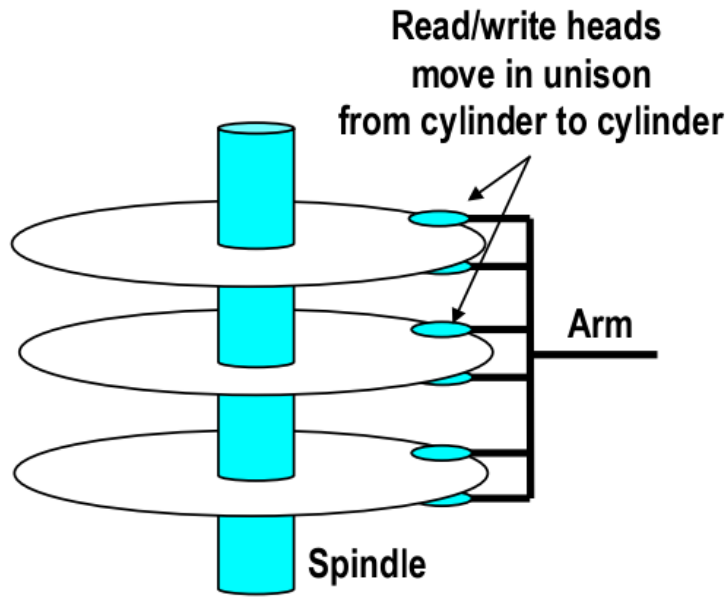
- Store bits "permanently" as magnetized areas on special platters
- Magnetic disks: moving parts $\rightarrow$ slow
- Cheap per GB of space



Source: CS:APP Slides



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

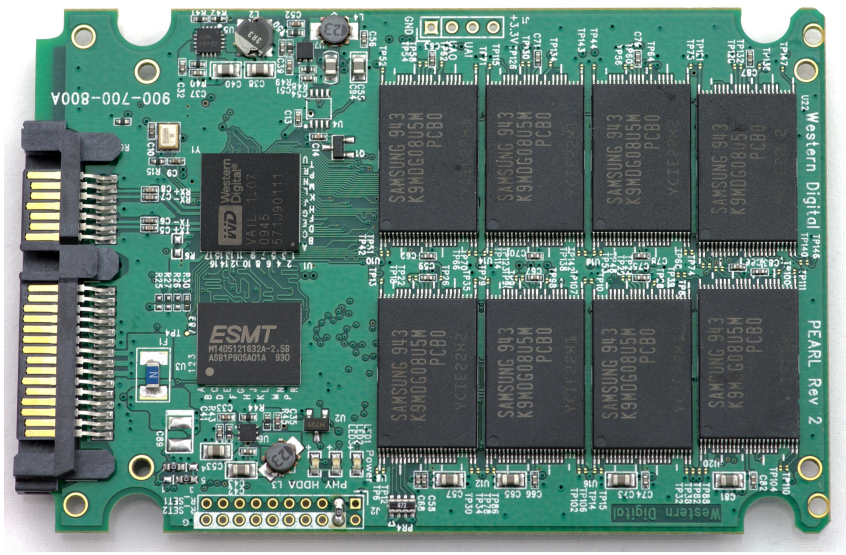
Sequential vs Random Access

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 $\rightarrow$ 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

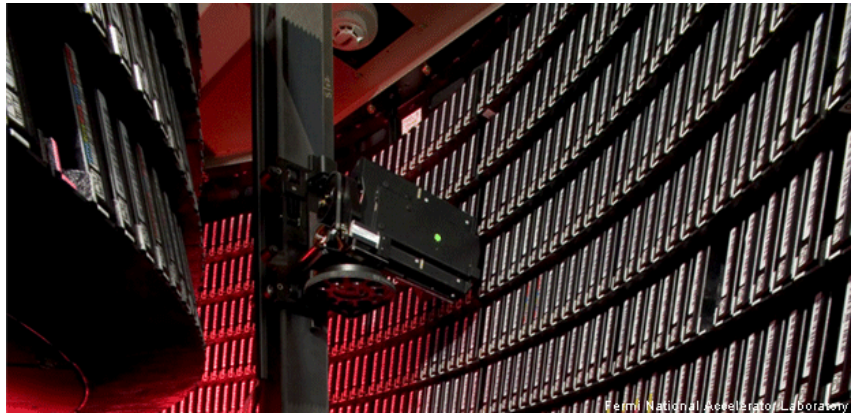
  – 4 to 5 cents/GB for HDD

  – 25 cents/GB for the SSD.

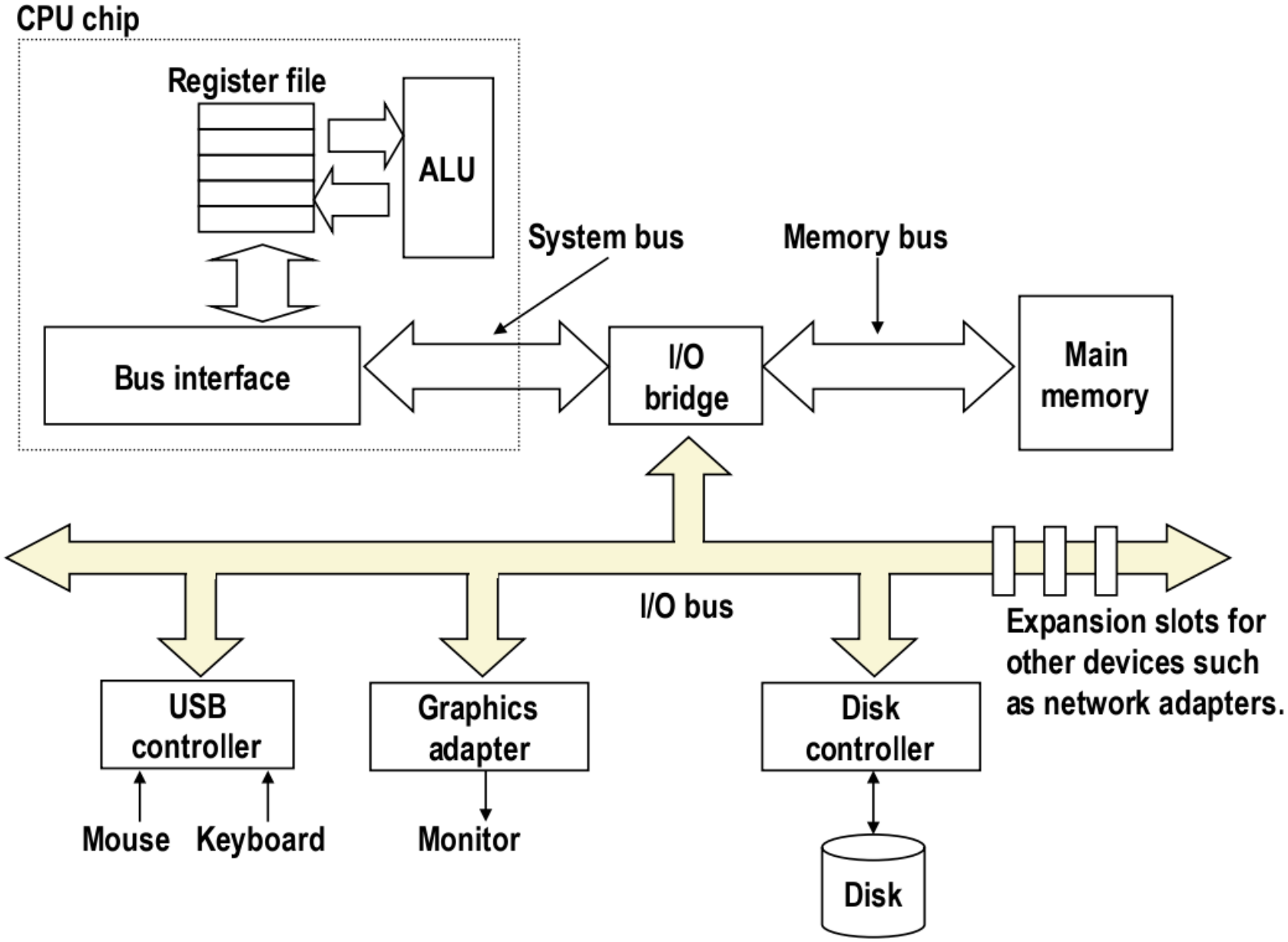
  PC Magazine, "SSD vs HDD" by Tom Brant and Joel Santo Domingo March 26, 2018

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

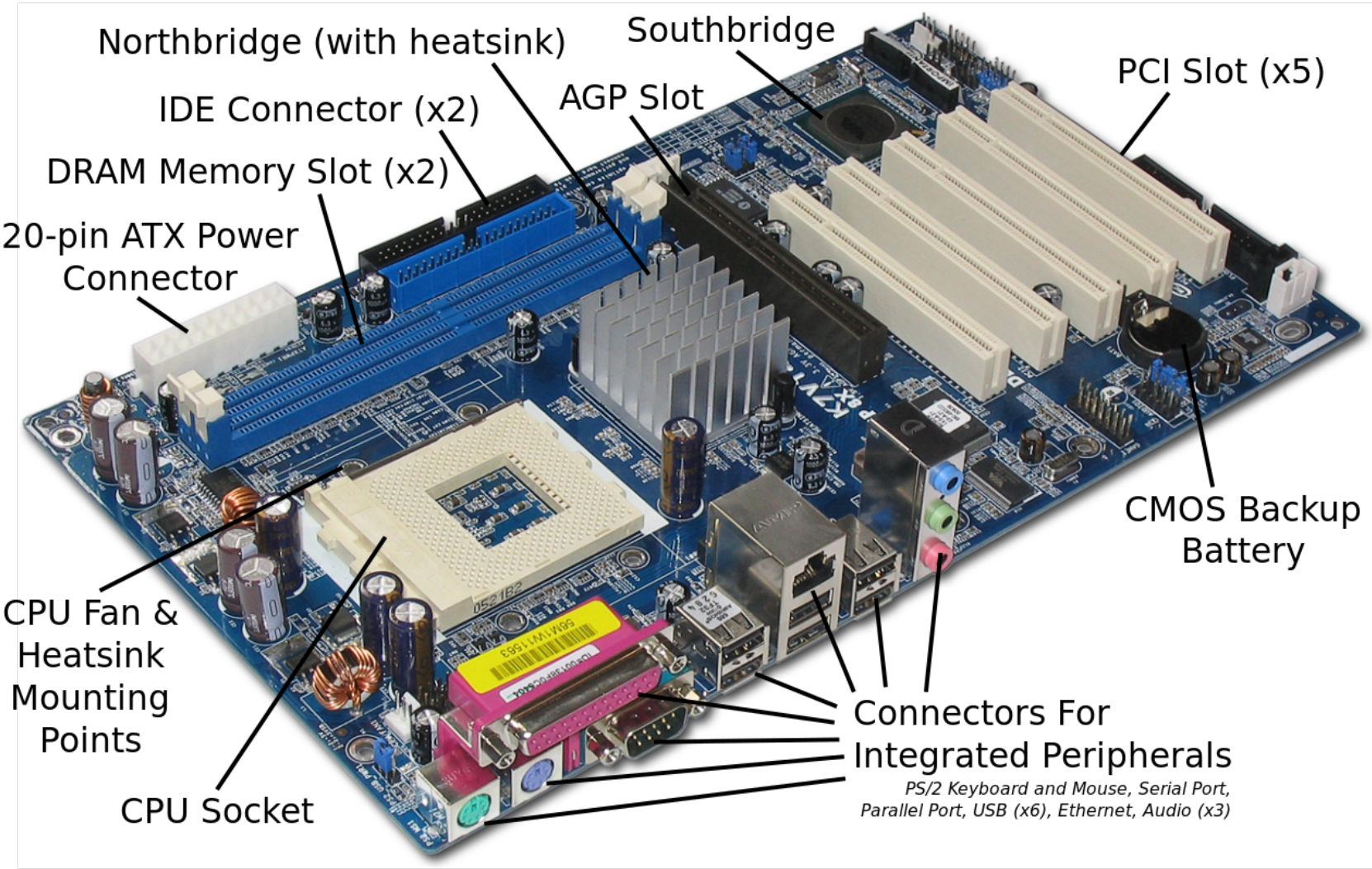
Bus A collection of wires which allow communication between parts of the computer. May be serial (single wire) or parallel (several wires), must have a communication protocol over it.

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.

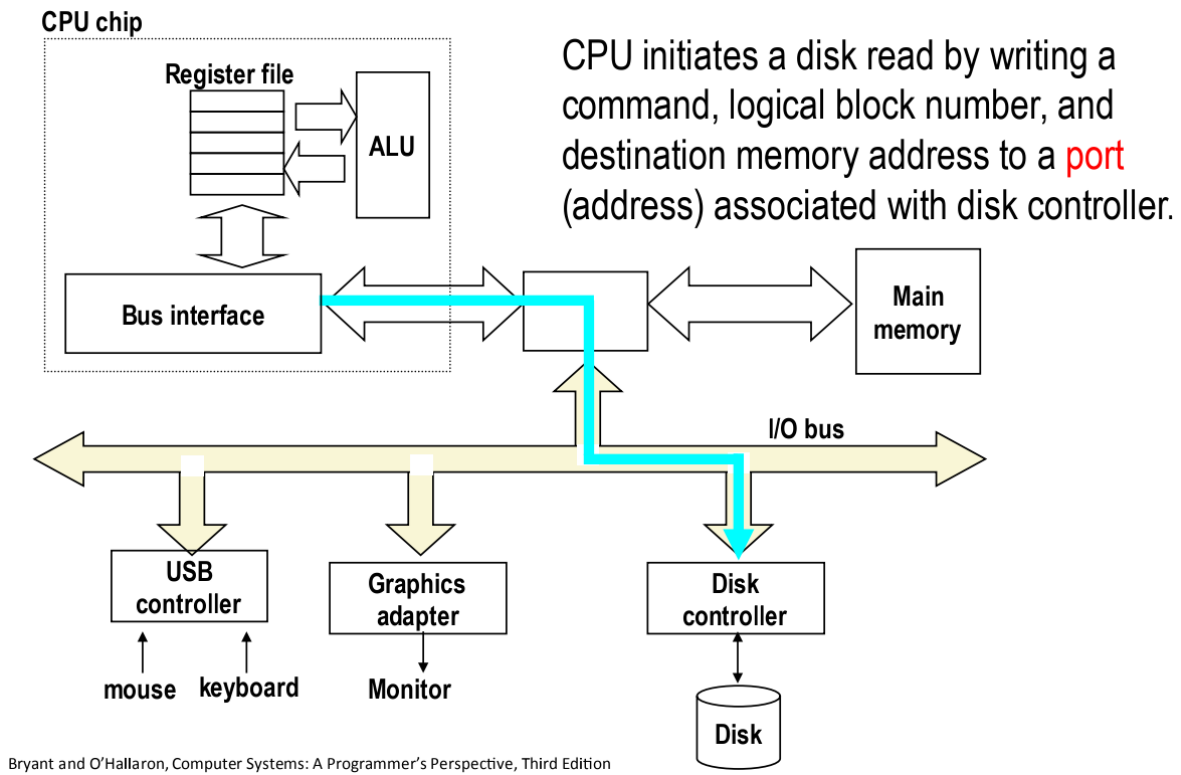
The Motherboard



Picture Source: Wikipedia
Live Props Courtesy of Free Geek Minneapolis

Memory Mapped I/O

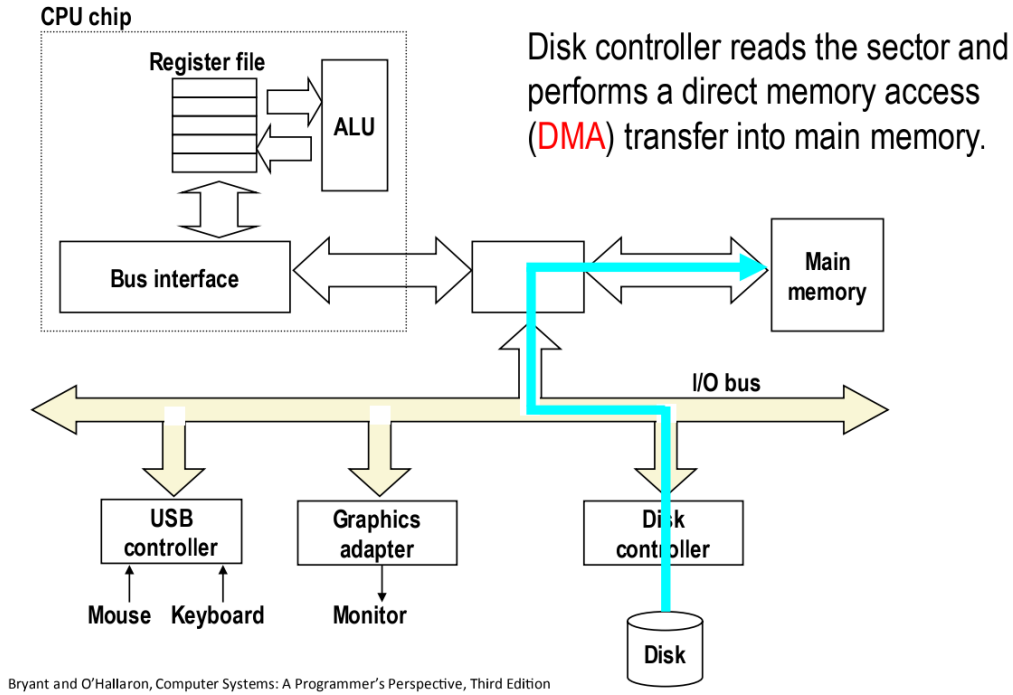
- Modern systems are a collection of devices and microprocessors
- CPU usually uses **memory mapped I/O**: read/write certain memory addresses translated to communication with devices on I/O bus



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**

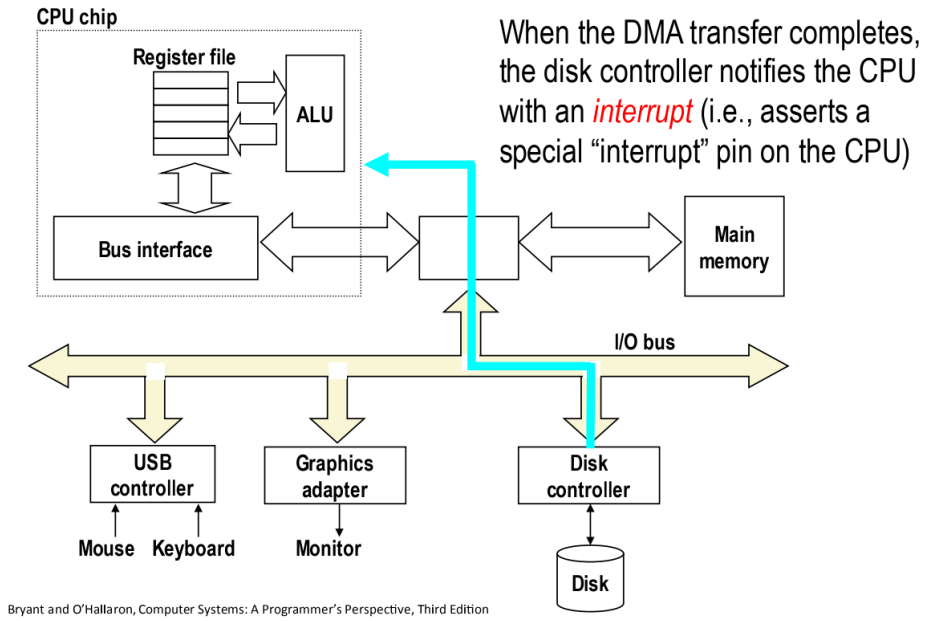


Interrupts and I/O

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



- 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

Exercise: Potential Conflicts in Memory

- Running multiple programs gets interesting particularly if they both reference the *same memory location*, e.g. address 8192

```
PROGRAM 1                          PROGRAM 2
...                                ...
## load global from #8192          ## add to global at #8192
movq 8192, %rax                    addl %esi, 8192
...                                ...
```

- What conflict exists between these programs?
- What are possible solutions to this conflict?

Answers: Potential Conflicts in Memory

- Both programs use address #8192, behavior depends on order that instructions are interleaved between them

```
ORDER A: Program 1 loads first
------------------------------
PROGRAM 1          PROGRAM 2
movq 8192, %rax    ...
...                addl %esi, 8192
```

```
ORDER B: Program 2 adds first
-----------------------------------
PROGRAM 1          PROGRAM 2
...                addl %esi, 8192
movq 8192, %rax    ...
```

- **Solution 1:** Never let Programs 1 and 2 run together (bleck!)
- **Solution 2:** Translate every memory address/access in every program while it runs

As wild as it sounds, most modern systems use memory address translation schemes called **Virtual Memory** (Solution 2) due to its many powerful features

Paged Memory

- Physical devices divide memory into chunks called **pages**
- Common page size supported by many OS's (Linux) and hardware is 4KB = 4096 bytes, can be larger with OS config
- CPU models use some # of bits for **Virtual Addresses**

```
> cat /proc/cpuinfo
vendor_id     : GenuineIntel
cpu family    : 6
model         : 79
model name    : Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50GHz
...
address sizes : 46 bits physical, 48 bits virtual
                                  ^^^^^^^
```

- Example of address with page number and offset labelled

```
   xxxxPagenumbrOff    : 48 bits used
 0x00007ffa0997a428    : 64 bit address
   |    |         |
   |    |         +-> Offset 0x428 within page, 12 bits
   |    +-> Page number 0x7ffa0997a, 36 bits
   +-> Constant bits, not used by processor
```

Translation happens at the Page Level

- Within a page, addresses are sequential
- Between pages, may be non-sequential

Page Table:

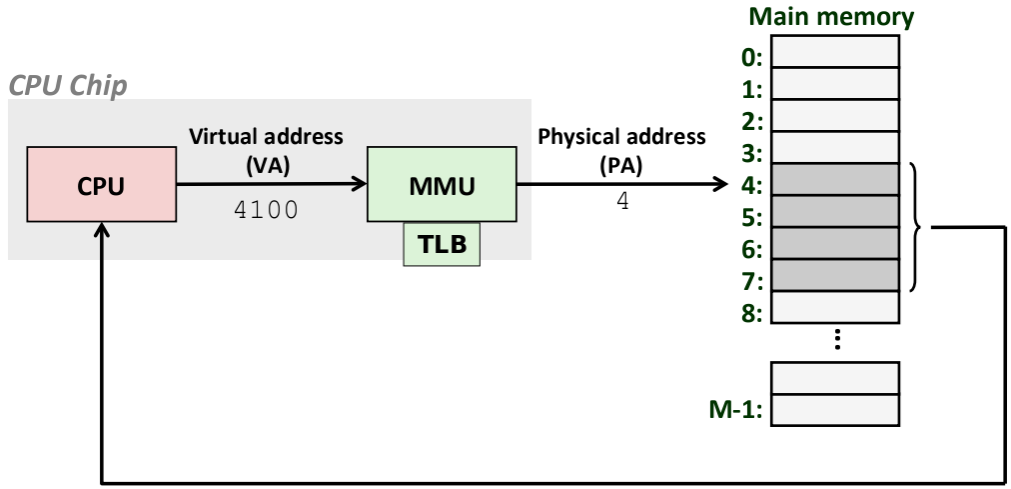
Virtual Page Num	Size	Physical Page Num
00007ffa0997a000	4K	RAM: 0000564955aa1000
00007ffa0997b000	4K	RAM: 0000321e46937000
...		...

Address Space From Page Table:

Virtual Address	Page Offset	Physical Address
00007ffa0997a000	0	0000564955aa1000
00007ffa0997a001	1	0000564955aa1001
00007ffa0997a002	2	0000564955aa1002
...		...
00007ffa0997afff	4095	0000564955aa1fff
00007ffa0997b000	0	0000321e46937000
00007ffa0997b001	1	0000321e46937001
...		...

Addresses Translation Hardware

- Translation must be **FAST** so usually involves hardware
- **MMU (Memory Manager Unit)** is a hardware element specifically designed for address translation
- Usually contains a special cache, **TLB (Translation Lookaside Buffer)**, which stores recently translated addresses

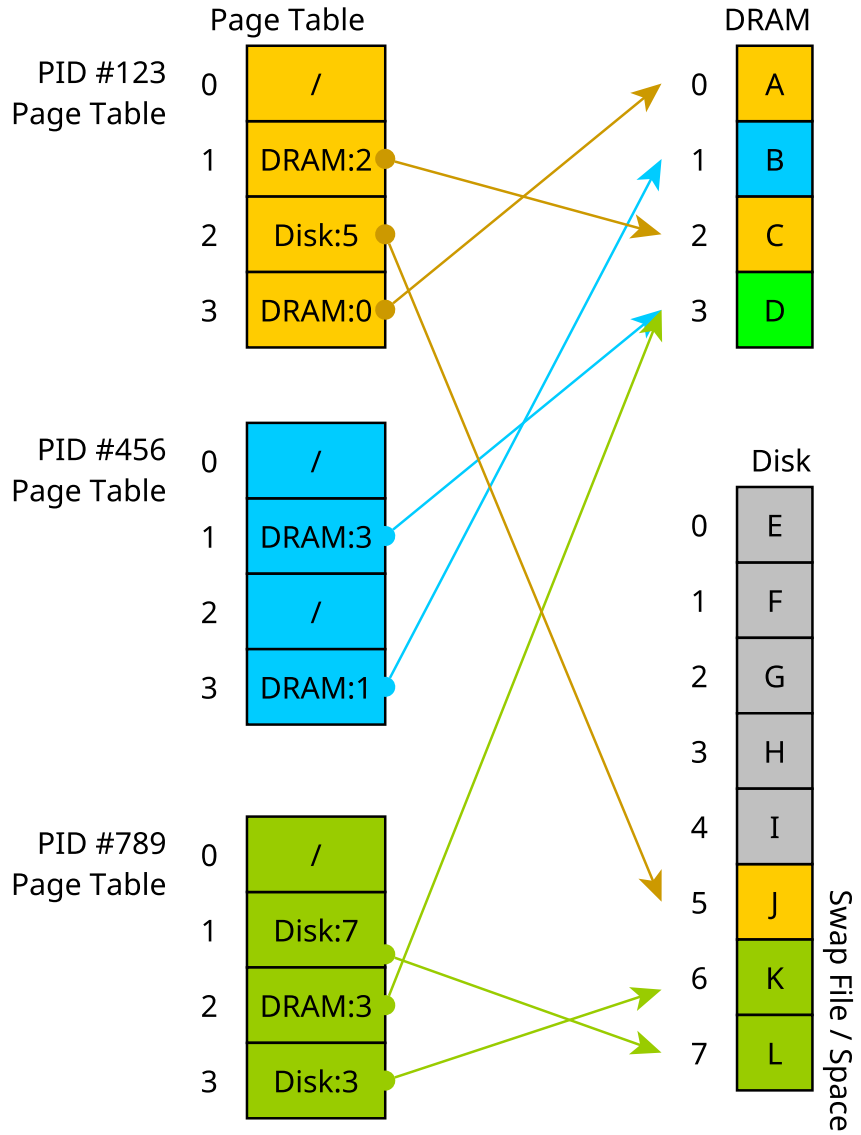


- OS Kernel interacts with MMU
- Provides location of the **Page Table**, data structure relating Virtual/Physical Addresses
- **Page Fault** : MMU couldn't map Virtual to Physical page, runs a Kernel routine to handle the fault

Exercise: Translating Virtual Addresses

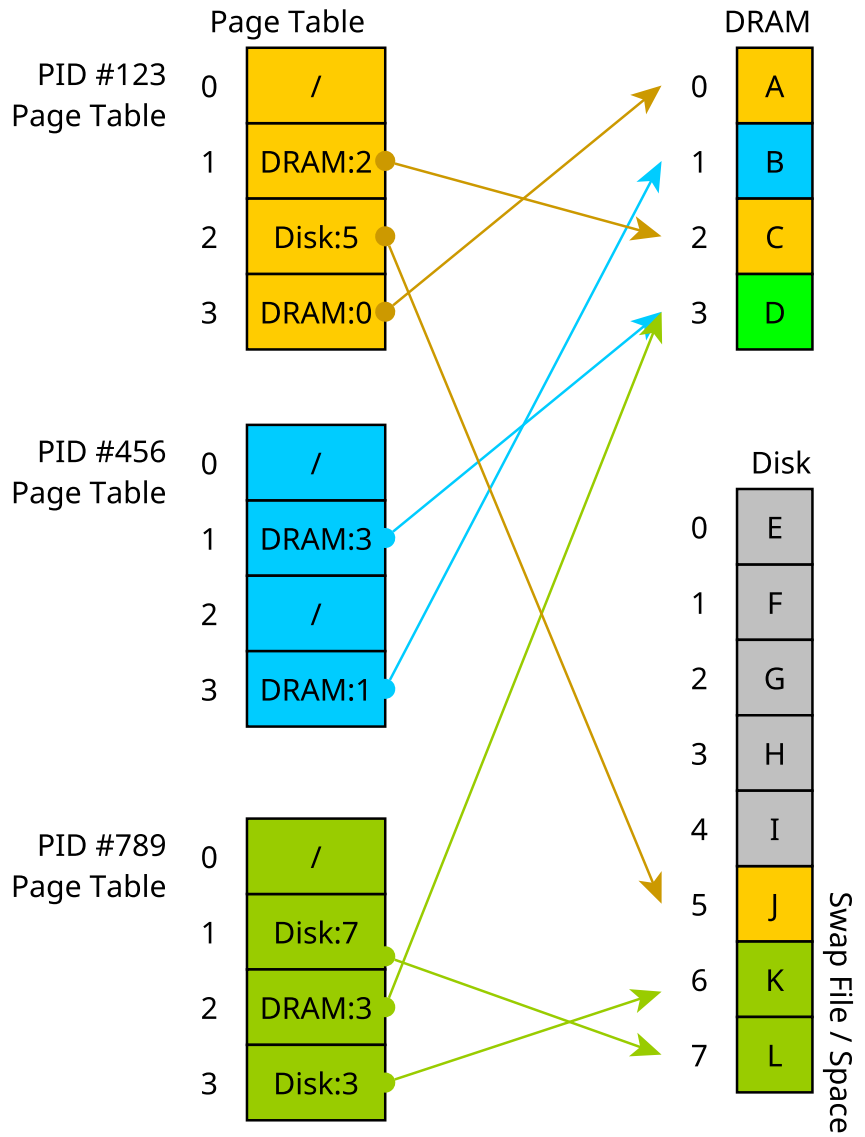
Nearby diagram illustrates relation of Virtual Pages to Physical Pages

1. How many page tables are there?
2. Where can a page table entry refer to?
3. Count the number of Virtual pages, compare to the number of physical pages - which his larger?
4. What happens if PID #123 accesses its Virtual Page #2
5. What happens if PID #456 accesses its Virtual Page #2



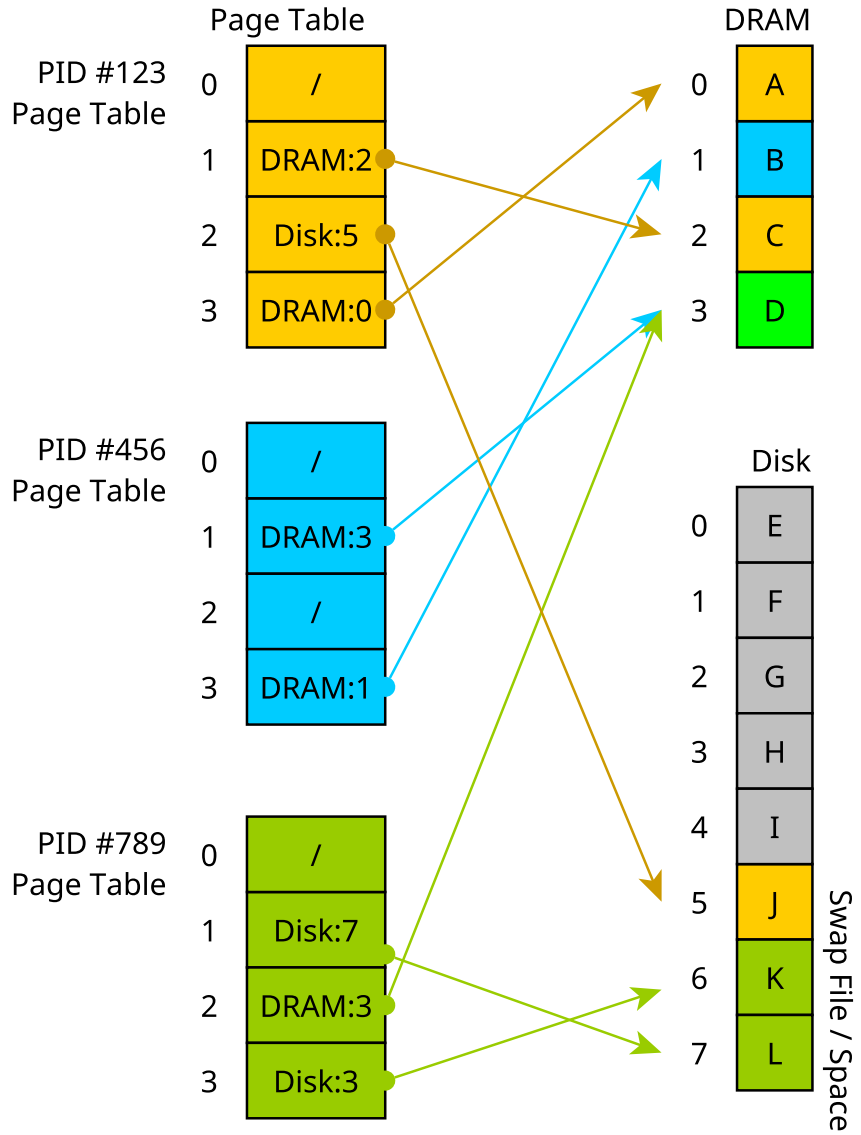
Translating Virtual Addresses 1/2

- On using a Virtual Memory address, MMU will search TLB for physical DRAM address,
- If found in TLB, Hit, use physical DRAM address
- If not found, MMU will search Page Table, if found and in DRAM, cache in TLB
- Else Miss = **Page fault**, OS decides..
  1. Page is swapped to Disk, move to DRAM, potentially evicting another page
  2. Page not in page table = Segmentation Fault



Translating Virtual Addresses 2/2

- Each process has its own page table, OS maintains mapping of Virtual to Physical addresses
- Processes "compete" for RAM
- OS gives each process impression it owns all of RAM
- OS may not have enough memory to back up all or even 1 process
- Disk used to supplement ram as **Swap Space**
- **Thrashing** may occur when too many processes want too much RAM, "constantly swapping"



Trade-offs of Address Translation

Wins of Virtual Memory

1. Avoids memory Conflicts where separate programs each use the same memory address
2. Programs can be compiled to assume they will have all memory to themselves
3. OS can make decisions about DRAM use and set policies for security and efficiency (next slide)

Losses of Virtual Memory

1. Address translation is not constant O(1), has an impact on performance of real algorithms*
2. Requires special hardware to make translation fast enough: MMU/TLB
3. Not needed if only a single program is running on a machine

Wins outweigh Losses in most systems so Virtual Memory is used widely, a *great idea* in CS

*See On a Model of Virtual Address Translation (2015)

The Many Other Advantages of Virtual Memory

1. Swap Space: System can project larger total memory than available DRAM by using Disk Space, DRAM is a "cache" for larger disk space, Swap program memory between DRAM+Disk as it is used
2. Security: Translation allows OS to check memory addresses for validity, segfault on out-of bounds access
3. Debugging: Valgrind checks addresses for validity
4. Sharing Data: Processes can share data with one another; request OS to map virtual addresses to same physical addresses
5. **Sharing Libraries**: Can share same program text between programs by mapping address space to same shared library
6. **Convenient I/O**: Map internal OS data structures for files to virtual addresses to make working with files free of read()/write()

Virtual Memory and `mmap()`

- Normally programs interact indirectly with Virtual Memory system
  - Stack/Heap/Globals/Text are mapped automatically to regions in Virtual Memory System
  - Maps are adjusted as Stack/Heap Grow/Shrink
- `mmap()` / `munmap()` directly manipulate page tables
  - `mmap()` creates new entries in page table
  - `munmap()` deletes entries in the page table
  - Can map arbitrary or specific addresses into memory
- `mmap()` is used to initially set up Stack / Heap / Globals / Text when a program is loaded by the program loader
- While a program is running can also use `mmap()` to interact with virtual memory
- A convenient way to do File I/O via **Memory Mapped Files**

Exercise: Printing Contents of file

Examine the two programs below which print the contents of a file

- Identify differences between them
- Which has a higher memory requirement?

```c
// print_file.c
int main(int argc, char *argv[]){
  FILE *fin = fopen(argv[1], "r");
  char inbuf[256];
  while(1){
    int nread =
      fread(inbuf, sizeof(char),
            256, fin);
    if(nread == 0){
      break;
    }
    for(int i=0; i<nread; i++){
      printf("%c",inbuf[i]);
    }
  }

  fclose(fin);
  return 0;
}
```

```c
// mmap_print_file.c
int main(int argc, char *argv[]){
  int fd = open(argv[1], O_RDONLY);

  struct stat stat_buf;
  fstat(fd, &stat_buf);
  int size = stat_buf.st_size;

  char *file_chars =
    mmap(NULL, size,
         PROT_READ, MAP_SHARED,
         fd, 0);

  for(int i=0; i<size; i++){
    printf("%c",file_chars[i]);
  }
  printf("\n");

  munmap(file_chars, size);
  close(fd);
  return 0;
}
```

Answers: Printing Contents of file

1. Write a simple program to print all characters in a file. What are key features of this program?
   - Open file
   - Read up to 256 characters into memory using `fread()/fscanf()`
   - Print those characters with `printf()`
   - Read more characters and print
   - Stop when end of file is reached
   - Close file
2. Examine `mmap_print_file.c`: does it contain all of these key features? Which ones are missing?
   - Missing the `fread()/fscanf()` portion
   - Uses `mmap()` to get **direct access** to the bytes of the file
   - Treat bytes as an array of characters and print them directly

mmap(): Mapping Addresses is Amazing

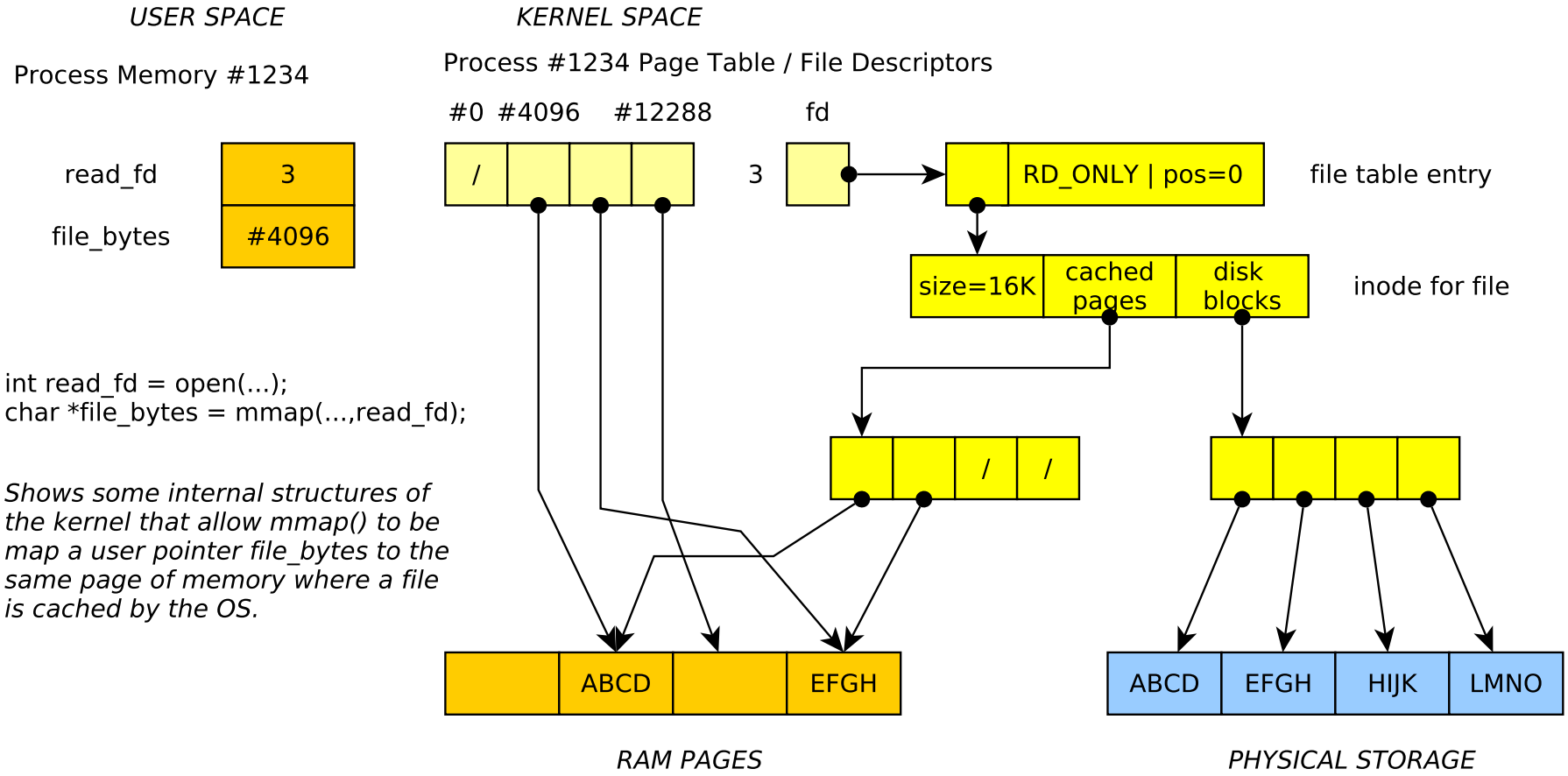
- ptr = mmap(NULL, size,...,fd,0) arranges backing entity of fd to be mapped to be mapped to ptr
- fd often a file opened with open() system call

```
int fd = open("gettysburg.txt", O_RDONLY);
// open file to get file descriptor

char *file_chars = mmap(NULL, size, PROT_READ, MAP_SHARED,
                        fd, 0);
// call mmap to get a direct pointer to the bytes in file associated
// with fd; NULL indicates don't care what address is returned;
// specify file size, read only, allow sharing, offset 0

printf("%c",file_chars[0]);                    // print 0th file char
printf("%c",file_chars[5]);                    // print 5th file char
```

OS usually Caches Files in RAM

- For efficiency, part of files are stored in RAM by the OS
- OS manages internal data structures to track which parts of a file are in RAM, whether they need to be written to disk
- `mmap()` alters a process Page Table to translate addresses to the cached file page
- OS tracks whether page is changed, either by file write or `mmap()` manipulation
- Automatically writes back to disk when needed
- Changes by one process to cached file page will be seen by other processes
- **See diagram on next slide**

Diagram of Kernel Structures for `mmap()`



Changing Files

- `mmap()` exposes several capabilities from the OS

```
char *file_chars =
  mmap(NULL, size,
       PROT_READ | PROT_WRITE, // map allowing read + write
       MAP_SHARED,             // share changes with original file
       fd, 0);                 // file to map + offset from start
```

- Assign new value to memory, OS writes changes into the file
- **Example**: `mmap_tr.c` to transform one character to another

Mapping things that aren't characters

`mmap()` just gives a pointer: can assert type of what it points at

- Example `int *`: treat file as array of binary ints
- Notice changing array will write to file

```
// mmap_increment.c: demonstrate working with mmap()'d binary data

int fd = open("binary_nums.dat", O_RDWR);
// open file descriptor, like a FILE *

int *file_ints = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
// get pointer to file bytes through mmap,
// treat as array of binary ints

int len = size / sizeof(int);
// how many ints in file

for(int i=0; i<len; i++){
  printf("%d\n",file_ints[i]);  // print all ints
}

for(int i=0; i<len; i++){
  file_ints[i] += 1;  // increment each file int, writes back to disk
}
```

`mmap()` Compared to Traditional `fread()`/`fwrite()` I/O

Advantages of `mmap()`

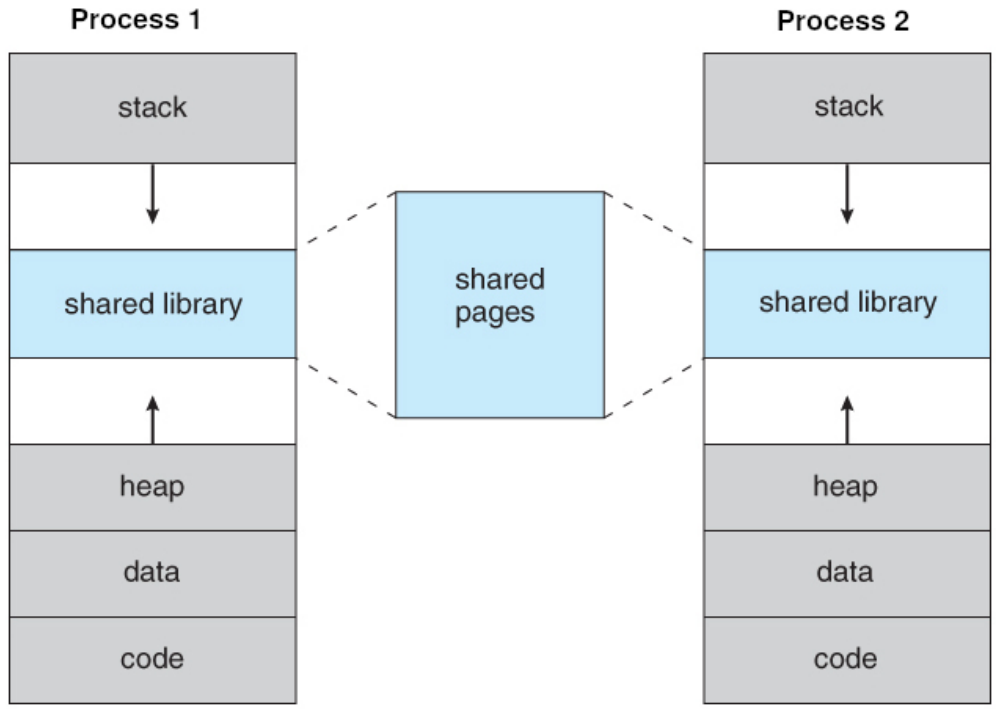
- Avoid following cycle
  - `fread()`/`fscanf()` file contents into memory
  - Analyze/Change data
  - `fwrite()`/`fscanf()` write memory back into file
- Saves memory and time
- Many Linux mechanisms backed by `mmap()` like processes sharing memory

Drawbacks of `mmap()`

- Always maps **pages** of memory: multiple of 4096b (4K)
- For small maps, lots of wasted space
- Cannot change size of files with `mmap()`: must used `fwrite()` to extend or other calls to shrink
- No bounds checking, just like everything else in C

Virtual Memory Enables Shared Libraries: `*.so` Files

- Many programs need to use `malloc()`, `printf()`, `fopen()`, `etc.`
- Rather than each program having its own copy, modern systems use **Shared Objects** and **Shared Libraries**



Source: John T. Bell Operating Systems Course Notes

- Example: `libc.so` is the C Library which contains Code/Text for `malloc()`, `printf()`, `fopen()`, `etc.`, 1-2MB of code
- One copy of `libc.so` exists in DRAM
- Many programs "share it" via Page Table mappings in Virtual Memory, reduces overall memory required

pmap: show virtual address space of running process

```
> ./memory_parts
0x5575555a71e9 : main()
0x5575555aa0c0 : global_arr
0x557555b482a0 : heap_arr
0x600000000000 : mmap'd block1
0x600000001000 : mmap'd block2
0x7f2244dc4000 : mmap'd file
0x7ffff0133b70 : stack_arr
my pid is 496605
press any key to continue
```

- Determine **process id** of running program
- pmap reports its virtual address space
- Reports features of each mapped page range such as size, permissions, possibly logical area

```
> pmap 496605
496605:   ./memory_parts
00005575555a6000      4K r---- memory_parts
00005575555a7000      4K r-x-- memory_parts TEXT
00005575555a8000      4K r---- memory_parts
00005575555a9000      4K r---- memory_parts
00005575555aa000      4K rw--- memory_parts GLOBALS
00005575555ab000      4K rw---   [ anon ]
0000557555b48000    132K rw---   [ anon ]   HEAP
000060000000000       8K rw---   [ anon ]
00007f2244bca000      8K rw---   [ anon ]
00007f2244bcc000    152K r---- libc-2.32.so
00007f2244bf2000   1332K r-x-- libc-2.32.so
00007f2244d3f000    304K r---- libc-2.32.so
00007f2244d8e000     12K rw--- libc-2.32.so
00007f2244d91000     24K rw---   [ anon ]
00007f2244dc4000      4K r---- gettysburg.txt
00007f2244dc5000      8K r---- ld-2.32.so
00007f2244dc7000    132K r-x-- ld-2.32.so
00007f2244de8000     36K r---- ld-2.32.so
00007f2244df2000      8K rw--- ld-2.32.so
00007ffff0114000    132K rw---   [ stack ]  STACK
00007ffff014d000     12K r----   [ anon ]
 total              2352K
```

Memory Protection

- Output of `pmap` indicates another feature of virtual memory: **protection**
- OS marks pages of memory with Read/Write/Execute/Share permissions like files
- Attempt to violate these and get segmentation violations (segfault)
- Ex: Executable page (instructions) usually marked as `r-x`: no write permission.
- Ensures program don't accidentally write over their instructions and change them
- Ex: By default, pages are not shared (no `'s'` permission) but can make it so with the right calls

Physical Locations of Pages

- UMN Kernel Object Student group members put together a `vpmap` program to print virtual to physical page locations on Linux
- Requires Administrator rights to use as physical locations are OS business
- https://github.com/UMN-Kernel-Object/virtmem

vpmap Sample Output

```
################################################################################
## vpmap shows Virtual Page Number (vpn) followed by Page Frame Number (pfn)
$> sudo ./vpmap 64814
[sudo] password for sudo:
Process 64814
55d11d5c7000-55d11d5c8000 r--p 00000000 fe:01 5119082       /virtmem/memory_parts
| vpn: 55d11d5c7  present  pfn: 2a9314  dirty: 1  exclu: 1  wprot: 0  isfile: 1

55d11d5c8000-55d11d5c9000 r-xp 00001000 fe:01 5119082       /virtmem/memory_parts
| vpn: 55d11d5c8  present  pfn: 1fddc6  dirty: 1  exclu: 1  wprot: 0  isfile: 1
...

55d11e7f0000-55d11e811000 rw-p 00000000 00:00 0             [heap]
| vpn: 55d11e7f0  present  pfn: 440dc0  dirty: 1  exclu: 1  wprot: 0  isfile: 0
| vpn: 55d11e7f1
| vpn: 55d11e7f2
| vpn: 55d11e7f3  ## unmapped pages (promised but not delivered)
...

7fc074a41000-7fc074a63000 r--p 00000000 fe:01 19139877      /usr/lib/libc.so.6
| vpn: 7fc074a41  present  pfn: 22b275  dirty: 1  exclu: 0  wprot: 0  isfile: 1
| vpn: 7fc074a42  present  pfn: 3b677d  dirty: 1  exclu: 0  wprot: 0  isfile: 1
...

7fc074a63000-7fc074bbd000 r-xp 00022000 fe:01 19139877      /usr/lib/libc.so.6
| vpn: 7fc074a63  present  pfn: 3ac617  dirty: 1  exclu: 0  wprot: 0  isfile: 1
...
| vpn: 7fc074a6b  present  pfn: 3ac61f  dirty: 1  exclu: 0  wprot: 0  isfile: 1
| vpn: 7fc074a6c  present  pfn: 22b200  dirty: 1  exclu: 0  wprot: 0  isfile: 1
| vpn: 7fc074a6d  present  pfn: 22b201  dirty: 1  exclu: 0  wprot: 0  isfile: 1

7ffd46c53000-7ffd46c74000 rw-p 00000000 00:00 0             [stack]
...               ## Highest addresses in stack in use but no physical pages
| vpn: 7ffd46c6f  ## yet assigned to lower pages
| vpn: 7ffd46c70
| vpn: 7ffd46c71  present  pfn: 403934  dirty: 1  exclu: 1  wprot: 0  isfile: 0
| vpn: 7ffd46c72  present  pfn: 21b607  dirty: 1  exclu: 1  wprot: 0  isfile: 0
| vpn: 7ffd46c73  present  pfn: 18ef8e  dirty: 1  exclu: 1  wprot: 0  isfile: 0
...
```

Exercise: Quick Review

1. While running a program, memory address #1024 always refers to a physical location in DRAM (True/False: why?)
2. Two programs which both use the address #1024 cannot be simultaneously run (True/False: why?)
3. What do MMU and TLB stand for and what do they do?
4. What is a memory page? How big is it usually?
5. What is a Page Table and what is it good for?

Answers: Quick Review

1. While running a program, memory address #1024 always refers to a physical location in DRAM (True/False: why?)
   - False: #1024 is usually a **virtual address** which is translated by the OS/Hardware to a physical location which *may* be in DRAM but may instead be paged out to disk
2. Two programs which both use the address #1024 cannot be simultaneously run (True/False: why?)
   - False: The OS/Hardware will likely translate these identical virtual addresses to **different physical locations** so that the programs doe not clobber each other's data
3. What do MMU and TLB stand for and what do they do?
   - Memory Management Unit: a piece of hardware involved in translating Virtual Addresses to Physical Addresses/Locations
   - Translation Lookaside Buffer: a special cache used by the MMU to make address translation **fast**
4. What is a memory page? How big is it usually?
   - A discrete hunk of memory usually 4Kb (4096 bytes) big
5. What is a Page Table and what is it good for?
   - A table maintained by the operating system that is used to map Virtual Addresses to Physical addresses for each page

Additional Review Questions

- What OS data structure facilitates the Virtual Memory system? What kind of data structure is it?
- What does `pmap` do?
- What does the `mmap()` system call do that enables easier I/O? How does this look in a C program?
- Describe at least 3 benefits a Virtual Memory system provides to a computing system