CS 31: Intro to Systems C Programming
L16: Storage

Vasanta Chaganti & Kevin Webb
Swarthmore College
Nov 7, 2023
Announcements

• HW 4 is out!
Reading Quiz
Data Alignment: Why?

• Simplify hardware
  – e.g., only read ints from multiples of 4
  – Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware

• Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)

• Simplify OS:
  – Prevents data from spanning virtual pages
  – Atomicity issues with load/store across boundary
Structs

• Laid out contiguously by field
  – In order of field declaration.
  – May require some padding, for alignment.

```c
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```
Structs

struct student{
    char name[11];
    short age;
    int id;
};
How much space do we need to store one of these structures? Why?

```c
struct student{
    char name[11];
    short age;
    int id;
};
```

A. 17 bytes
B. 18 bytes
C. 20 bytes
D. 22 bytes
E. 24 bytes
Use sizeof() when allocating structs with malloc()!
struct student{
  char name[11];
  short age;
  int id;
};
Alternative Layout

```c
struct student{
    char name[11];
    short age;
    int id;
};
```

size of data: 17 bytes
size of struct: 17 bytes

In general, this isn’t a big deal on a day-to-day basis. Don’t go out and rearrange all your struct declarations.
struct student {
    int id;
    short age;
    char name[11];
};

struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, "Alice");

Struct is declared on the stack. (NOT a pointer)
struct student {
    int id;
    short age;
    char name[11];
};
struct student *s = malloc(sizeof(struct student));

How do we get to the id and age?
Struct field syntax...

```
struct student {
    int id;
    short age;
    char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Option 1: Works but ugly

```
(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, "Alice");
```

Option 2: Use struct pointer dereference!

```
s->id = 406432;
s->age = 20;
strcpy(s->name, "Alice");
```
Memory alignment applies elsewhere too!

```
int x;         vs.       double y;
char ch[5];    int x;
short s;       short s;
double y;      char ch[5];
```

In nearly all cases, you shouldn't stress about this. The compiler will figure out where to put things.

Exceptions: networking, OS
Unions

• Declared like a struct, but only contains one field, rather than all of them.

• Struct: field 1 and field 2 and field 3 ...
• Union: field 1 or field 2 or field 3 ...

Intuition: you know you only need to store one of N things, don’t waste space.
Unions

struct my_struct {
    char ch[2];
    int i;
    short s;
}

union my_union {
    char ch[2];
    int i;
    short s;
}

my_struct in memory

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;
```

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;

u.s = 2;
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;
u.s = 2;
u.ch[0] = 'a';

u.i = 5;
```

Reading i or s here would be bad!

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!
Unions

```c
my_union u;

u.i = 7;
u.s = 2;
u.ch[0] = 'a';

u.i = 5;

union my_union {
    char ch[2];
    int i;
    short s;
}

Same memory used for all fields!
```

Reading i or s here would be bad!
Unions

- You probably won’t use these often.
- Use when you need mutually exclusive types.
- Can save memory.

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!
Strings

• Strings are *character arrays*

• Layout is the same as:
  – char name[10];

• Often accessed as (char *)
String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

char name[10];
strcpy(name, “CS 31”);
String Functions

• C library has many built-in functions that operate on char *’s:
  – `strcpy`, `strdup`, `strlen`, `strcat`, `strcmp`, `strstr`

```c
char name[10];
strcpy(name, "CS 31");
```

• Null terminator (\0) ends string.
  – We don’t know/care what comes after
String Functions

• C library has many built-in functions that operate on char *’s:
  – strcpy, strdup, strlen, strcat, strcmp, strstr

• Seems simple on the surface.
  – That null terminator is tricky, strings error-prone.
  – Strings used everywhere!

• You will implement use these functions in a future lab.
Up next…

• New topic: Storage and the Memory Hierarchy
Transition

• First half of course: hardware focus
  – How the hardware is constructed
  – How the hardware works
  – How to interact with hardware / ISA

• Up next: performance and software systems
  – Memory performance
  – Operating systems
  – Standard libraries (strings, threads, etc.)
Efficiency

• How to Efficiently Run Programs

• Good algorithm is critical...

• Many systems concerns to account for too!
  – The memory hierarchy and its effect on program performance
  – OS abstractions for running programs efficiently
  – Support for parallel programming
Efficiency

• How to Efficiently Run Programs

• Good algorithm is critical...

• Many systems concerns to account for too!
  – The memory hierarchy and its effect on program performance
  – OS abstractions for running programs efficiently
  – Support for parallel programming
Suppose you’re designing a new computer architecture. Which type of memory would you use? Why?

A. low-capacity (~1 MB), fast, expensive

B. medium-capacity (a few GB), medium-speed, moderate cost

C. high-capacity (100’s of GB), slow, cheap

D. something else (it must exist)

trade-off between capacity and speed
Abstraction Goal

• Reality: There is no one type of memory to rule them all!

• Abstraction: hide the complex/undesirable details of reality.

• Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.
Classifying Memory

• Broadly, two types of memory:
  1. Primary storage: CPU instructions can access any location at any time (assuming OS permission)
  2. Secondary storage: CPU can’t access this directly
Random Access Memory (RAM)

• Any location can be accessed directly by CPU
  – Volatile Storage: lose power \(\rightarrow\) lose contents

• Static RAM (SRAM)
  – Latch-Based Memory (e.g. RS latch), 1 bit per latch
  – Faster and more expensive than DRAM
    • “On chip”: Registers, Caches

• Dynamic RAM (DRAM)
  – Capacitor-Based Memory, 1 bit per capacitor
    • “Main memory”: Not part of CPU
Memory Technologies

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

- Dynamic RAM (DRAM)
  - 50ns – 100ns, $20 – $75 per GB
  (Main memory, “RAM”)

We’ve talked a lot about registers (SRAM) and we’ll cover caches (SRAM) soon. Let’s look at main memory (DRAM) now.
Dynamic Random Access Memory (DRAM)

Capacitor based:
- cheaper and slower than SRAM
- capacitors are leaky (lose charge over time)
  - **Dynamic**: value needs to be refreshed (every 10-100ms)

Example: DIMM
(Dual In-line Memory Module):
Connecting CPU and Memory

- Components are connected by a **bus**:
  - A bus is a collection of parallel wires that carry address, data, and control signals.
  - Buses are typically shared by multiple devices.
How A Memory Read Works

(1) CPU places address A on the memory bus.

Load operation: \texttt{mov (Address A), \%rax}
(2) Main Memory reads address A from memory, fetches value at that address and puts it on the bus

Sending the value back to the CPU
(3) CPU reads value from the bus, and copies it into register rax. A copy also goes into the on-chip cache memory.
How a Memory Write Works

1. CPU writes A to bus, memory reads it
2. CPU writes value to bus, memory reads it
3. Memory stores value at address A
Secondary Storage

• Disk, Tape Drives, Flash Solid State Drives, ...

• Non-volatile: retains data without a charge

• Instructions **CANNOT** directly access data on secondary storage
  – No way to specify a disk location in an instruction
  – Operating System moves data to/from memory
Secondary Storage

CPU
- Register
- Register
- Register
- Register
- ALU
- CPU Cache

Memory Bus

Memory Module Slots

I/O Bus (e.g., PCI)
- SATA Controller
- USB Controller
- IDE Controller
- ...

I/O Controller

Secondary Storage Devices

path is much longer
What’s Inside A Disk Drive?

- Spindle
- Arm
- Actuator
- Platters
- Controller Electronics (includes processor & memory)
- Bus connector
- R/W head
- Data Encoded as points of magnetism on Platter surfaces

Device Driver (part of OS code) interacts with Controller to R/W to disk

Image from Seagate Technology
Reading and Writing to Disk

Data blocks located in some Sector of some Track on some Surface

1. Disk Arm moves to correct track (seek time)
2. Wait for sector spins under R/W head (rotational latency)
3. As sector spins under head, data are Read or Written (transfer time)
Memory Technology

• **Static RAM (SRAM)**
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

• **Dynamic RAM (DRAM)**
  - 50ns – 100ns, $20 – $75 per GB

  Solid-state disks (flash): 100 us – 1 ms, $2 - $10 per GB

• **Magnetic disk**
  - 5ms – 15ms, $0.20 – $2 per GB

  Like walking:
  - Down the hall

  Across campus
  - (to Cleveland / Indianapolis)

  To Seattle

1 ms == 1,000,000 ns
The Memory Hierarchy

- **Smaller**
  - Faster
  - Costlier per byte

- **Larger**
  - Slower
  - Cheaper per byte

---

- **Register**
  - 1 cycle to access

- **Cache(s) (SRAM)**
  - ~10’s of cycles to access

- **Main memory (DRAM)**
  - ~100 cycles to access

- **Local secondary storage (disk)**
  - ~100 M cycles to access

---

- **CPU instructions can directly access**
- **On-Chip Storage**

---

**Example:**
- CPU can directly access registers.
- Registers have the fastest access time.
- Main memory is faster than local secondary storage.
- Local secondary storage is the slowest and cheapest per byte.
Where does accessing the network belong?

- Larger
  - Slower
  - Cheaper per byte

- Local secondary storage (disk)
  - ~100 M cycles to access

- On Chip Storage
  - ~100 cycles to access
  - Cache(s) (SRAM)
    - ~10’s of cycles to access
  - Main memory (DRAM)
    - ~100 cycles to access
  - Registers
    - 1 cycle to access

A: Here
B: Here
C: Somewhere else
The Memory Hierarchy

- **Local secondary storage (disk)**: Larger, Slower, Cheaper per byte
- **Remote secondary storage**: Tapes, Web servers / Internet
- **On Chip Storage**: Smaller, Faster, Costlier per byte
- **Main memory (DRAM)**: ~100 cycles to access
- **Cache(s) (SRAM)**: ~10’s of cycles to access
- **Registers**: 1 cycle to access

CPU instructions can directly access:
- Registers
- Cache(s)
- Main memory

Access times:
- Local secondary storage: Slower than local disk
- Remote secondary storage: Slower than local disk
Abstraction Goal

• Reality: There is no one type of memory to rule them all!

• Abstraction: hide the complex/undesirable details of reality.

• Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.
Motivating Story / Analogy

• You work at a video rental store (remember Blockbuster?)

• You have a huge warehouse of movies
  – 10-15 minutes to find movie, bring to customer
  – Customers don’t like waiting...

• You have a small office in the front with shelves, you choose what goes on shelves
  – < 30 seconds to find movie on front shelf
The Video Store Hierarchy

**Goal:** strategically put movies on office shelf to reduce trips to warehouse.

- **Large Warehouse:** ~10 minutes to find movie
- **On Shelf Storage:** ~30 seconds to find movie
- **Front Office Shelves:** ~30 seconds to find movie
Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind

B. The Godfather

C. Pulp Fiction

D. Rocky V

E. There’s no way for us to know.
Problem: Prediction

• We can’t know the future...

• So... are we out of luck?
  What might we look at to help us decide?

• The past is often a pretty good predictor...
Repeat Customer: Bob

- Has rented “Eternal Sunshine of the Spotless Mind” ten times in the last two weeks.

- You talk to him:
  - He just broke up with his girlfriend
  - Swears it will be the last time he rents the movie (he’s said this the last six times)
Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind
B. The Godfather
C. Pulp Fiction
D. Rocky V
E. There’s no way for us to know.
Repeat Customer: Alice

• Alice rented Rocky a month ago

• You talk to her:
  – She’s really likes Sylvester Stalone

• Over the next few weeks she rented:
  – Rocky II, Rocky III, Rocky IV
Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind

B. The Godfather

C. Pulp Fiction

D. Rocky V

E. There’s no way for us to know.
Critical Concept: Locality

• **Locality**: we tend to repeatedly access recently accessed items, or those that are nearby.

• **Temporal locality**: An item that has been accessed recently is likely to be accessed again soon. (Bob)

• **Spatial locality**: We’re likely to access an item that’s nearby others we just accessed. (Alice)
In the following code, how many examples are there of temporal / spatial locality? Where are they?

```c
int i;
int num = read_int_from_user();
int *array = create_random_array(num);
for (i = 0; i < num; i++) {
    printf("At index %d, value: %d", i, array[i]);
}
```

A. 1 temporal, 1 spatial
B. 1 temporal, 2 spatial
C. 2 temporal, 1 spatial
D. 2 temporal, 2 spatial
E. Some other number
Big Picture

For memory exhibiting locality (stuff we’re using / likely to use):

Work hard to keep them up here!

Bulk storage down here.

Move this up on demand.
**Big Picture**

- **Registers**
  - Faster than cache.
  - Holds a VERY small amount (subset of cache).

- **Cache(s)** (SRAM)
  - Faster than memory. (On-chip hardware)
  - Holds a subset of memory.

- **Main memory** (DRAM)
  - Faster than disk.
  - Holds a subset of disk.

- **Flash SSD / Local network**

- **Local secondary storage (disk)**

- **Remote secondary storage (tapes, Web servers / Internet)**
Cache

• In general: a storage location that holds a subset of a larger memory, faster to access

• CPU cache: an SRAM on-chip storage location that holds a subset of DRAM main memory (10-50x faster to access)

• Goal: choose the right subset, based on past locality, to achieve our abstraction

When we say “cache”, assume we’re referring to CPU cache from now on, unless we say otherwise.
Cache Basics

• CPU real estate dedicated to cache

• Usually two (or more) levels:
  – L1: smallest, fastest
  – L2: larger, slower

• Same rules apply:
  – L1 subset of L2
Cache Basics

• CPU real estate dedicated to cache

• Usually two levels:
  – L1: smallest, fastest
  – L2: larger, slower

• We’ll assume one cache (same principles)

Cache is a subset of main memory.
(Not to scale, memory much bigger!)
Cache Basics: Read from memory

• In parallel:
  – Issue read to memory
  – Check cache
Cache Basics: Read from memory

- In parallel:
  - Issue read to memory
  - Check cache

- Data in cache (hit):
  - Good, send to register
  - Cancel/ignore memory
Cache Basics: Read from memory

- **In parallel:**
  - Issue read to memory
  - Check cache

- **Data in cache (hit):**
  - Good, send to register
  - Cancel/ignore memory

- **Data not in cache (miss):**
  1. Load cache from memory (might need to evict data)
  2. Send to register
Cache Basics: Write to memory

- Assume data already cached
  - Otherwise, bring it in like read

1. Update cached copy.

2. Update memory?
When should we copy the written data from cache to memory? Why?

A. Immediately update the data in memory when we update the cache.

B. Update the data in memory when we remove ("evict") the data from the cache.

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. (When?)
When should we copy the written data from cache to memory? Why?

A. **Write-through**: Immediately update the data in memory when we update the cache.

B. **Write-back**: Update the data in memory when we remove ("evict") the data from the cache.

C. Update the data in memory if the data is needed elsewhere (e.g., another core).

D. Update the data in memory at some other time. (When?)
Cache Basics: Write to memory

• Both options (write-through, write-back) viable

• write-through: write to memory immediately
  + simpler
  - accesses memory more often (slower)

• write-back: only write to memory on eviction
  + potentially reduces memory accesses (faster)
  - complex (cache inconsistent with memory)
Cache Coherence

- Keeping multiple cores’ memory consistent
Cache Coherence

- Keeping multiple cores’ memory consistent

- If one core updates data
  - Copy data directly from one cache to the other.
  - Avoid (slower) memory

- Lots of HW complexity here. We might discuss towards end of semester.
The Memory Hierarchy

- **Local secondary storage (disk)**: Larger, Slower, Cheaper per byte.
- **Remote secondary storage (tapes, Web servers / Internet)**: Slower than local disk to access.
- **Main memory (DRAM)**: ~100 cycles to access.
- **Cache(s) (SRAM)**: ~10’s of cycles to access.
- **Registers**: 1 cycle to access.

CPU instructions can directly access on-chip storage.
Data Access Time over Years

Over time, gap widens between DRAM, disk, and CPU speeds.

Really want to avoid going to disk for data.

Want to avoid going to Main Memory for data.

multicore
Recall

• A cache is a smaller, faster memory, that holds a subset of a larger (slower) memory

• We take advantage of **locality** to keep data in cache as often as we can!

• When accessing memory, we check cache to see if it has the data we’re looking for.
Why we miss…

• **Compulsory (cold-start) miss:**
  – First time we use data, load it into cache.

• **Capacity miss:**
  – Cache is too small to store all the data we’re using.

• **Conflict miss:**
  – To bring in new data to the cache, we evicted other data that we’re still using.
Cache Design

- Lot’s of characteristics to consider:
  - Where should data be stored in the cache?

![Diagram showing cache and main memory connections]
Cache Design

• Lot’s of characteristics to consider:
  – Where should data be stored in the cache?
  – What size data chunks should we store? (block size)
Cache Design

• Lot’s of characteristics to consider:
  – Where should data be stored in the cache?
  – What size data chunks should we store? (block size)

• Goals:
  – Maximize hit rate
  – Maximize (temporal & spatial) locality benefits
  – Reduce cost/complexity of design
Suppose the CPU asks for data, it’s not in cache. We need to move it into cache from memory. Where in the cache should it be allowed to go?

A. In exactly one place.

B. In a few places.

C. In most places, but not all.

D. Anywhere in the cache.
A. In exactly one place. (“Direct-mapped”)  
   – Every location in memory is directly mapped to one place in the cache. Easy to find data.

B. In a few places. (“Set associative”)  
   – A memory location can be mapped to (2, 4, 8) locations in the cache. Middle ground.

C. In most places, but not all.

D. Anywhere in the cache. (“Fully associative”)  
   – No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.
A larger block size (caching memory in larger chunks) is likely to exhibit...

A. Better temporal locality

B. Better spatial locality

C. Fewer misses (better hit rate)

D. More misses (worse hit rate)

E. More than one of the above. (Which?)
Block Size Implications

• Small blocks
  – Room for more blocks
  – Fewer conflict misses

• Large blocks
  – Fewer trips to memory
  – Longer transfer time
  – Fewer cold-start misses
Trade-offs

• There is no single best design for all purposes!

• Common systems question: which point in the design space should we choose?

• Given a particular scenario:
  – Analyze needs
  – Choose design that fits the bill
Real CPUs

• Goals: general purpose processing
  – balance needs of many use cases
  – middle of the road: jack of all trades, master of none

• Some associativity, medium size blocks:
  – 8-way associative (memory in one of eight places)
  – 16 or 32 or 64-byte blocks
What should we use to determine whether or not data is in the cache?

A. The memory address of the data.
B. The value of the data.
C. The size of the data.
D. Some other aspect of the data.
What should we use to determine whether or not data is in the cache?

A. The memory address of the data.
   – Memory address is how we identify the data.

B. The value of the data.
   – If we knew this, we wouldn’t be looking for it!

C. The size of the data.

D. Some other aspect of the data.