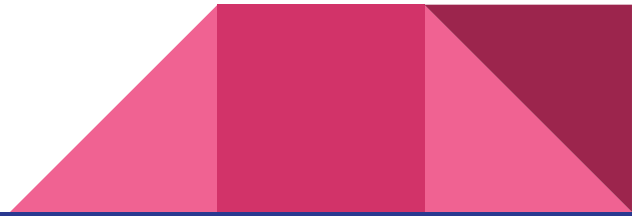
CS 33: Computer Organization

Dis 1B: Week 6 Discussion

Agenda

- Lecture Recap
  - Optimization
  - Cache
- Stack Exploits
  - This will help a lot for the buffer lab
- We are going to cover a lot of material today
  - So bear with me



Compiler Optimization

- Compilers are pretty smart
- It optimizes your code so that your code can run faster
  - Reducing frequency of computation
  - Replacing expensive instructions with cheaper instructions
    - ex) Replacing multiplication with left shift or addition
    - Multiplication is about 4 times more expensive than left shift or addition
    - ex) x = 10 * n

Code Motion

```
void setRow(int *a, int *b, int n) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        a[n * i + j] = b[j];
                }
        }
}
```

- How can a compiler optimize this code?
- What are some code blocks that can be optimized, without changing its functionality?

Compiler Optimization

- Compilers are awesome and let people write codes in very dumb ways
  - Not true!
- However, compilers are not *that* smart
- It can only optimize your code only if the optimized program is guaranteed to have the same behavior as the unoptimized version for all possible cases
- **All possible cases?!**

Memory Aliasing

```
void func1(int* a, int* b) {
    *a = *a + *b;
    *a = *a + *b;
}
```

```
void func2(int* a, int* b) {
    *a = *a + 2 * *b;
}
```

Memory Aliasing

- func1 and func2 seem like they would return the same output
- func1 reads memory 6 times whereas func2 only reads memory three times
- Can we replace func1() with func2()?

Function Calls

```
int f();

int func1() {
        return f() + f() + f() + f();
}

int func2() {
        return 4 * f();
}
```

- func1 and func2 seem like they would return the same output
- func1 calls f() 4 times whereas func2 only calls f() once
- Can we replace func1() with func2()?

Function Calls

```
int f();

int func1() {

        return f() + f() + f() + f();

}

int func2() {

        return 4 * f();

}
```

```
int counter = 0;

int f() {

        return counter++;

}
```

Compiler Optimization

- You have to be smarter than your compiler!
- Compilers do not optimize function calls because it is very hard to determine if a function is free of side effects

Optimizing Function Calls

```
void encryptString(char* s) {
    for (int i = 0; i < strlen(s); i++) {
        s[i] += 5;
    }
}
```

- What is the time complexity of this function?
- Hmm, since we have a loop that iterates through the entire string, it must be o(N) where N is the size of the input string!
- Is that true?

Optimizing Function Calls

```
void encryptString(char* s) {
    for (int i = 0; i < strlen(s); i++) {
        s[i] += 5;
    }
}
```

- o(N^2) time complexity!
- Why?
- How do we optimize this code to o(N)?

Optimizing Function Calls

```
void encryptString(char* s) {
	int size = strlen(s);
	for (int i = 0; i < size; i++) {
		s[i] += 5;
	}
}
```

- o(N) time complexity!

Loop Unrolling

```
void add10000Elements(int* a, int* b, int* c) {
    for (int i = 0; i < 10000; i++) {
        a[i] = b[i] + c[i];
    }
}
```

- How many times are we checking if i is less than 10000?
- Can we do better?

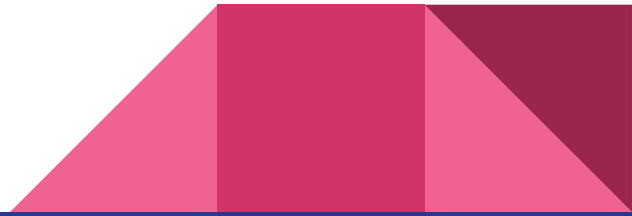
Loop Unrolling

```
void add10000Elements(int* a, int* b, int* c) {
    for (int i = 0; i < 10000; i += 5) {
        a[i] = b[i] + c[i];
        a[i + 1] = b[i + 1] + c[i + 1];
        a[i + 2] = b[i + 2] + c[i + 2];
        a[i + 3] = b[i + 3] + c[i + 3];
        a[i + 4] = b[i + 4] + c[i + 4];
    }
}
```

- How many times are we checking if i is less than 10000?
- You guys will learn why / when this is faster and more efficient in detail in CS M151B
- For now, just be aware that there is a technique called loop unrolling which does this
- k = 5
  - loop unrolling factor

Locality

- Temporal Locality: Recently referenced items are likely to be referenced again in the near future
- Spatial Locality: Items with nearby addresses tend to be referenced close together in time
- Your program is much faster if your code has good locality



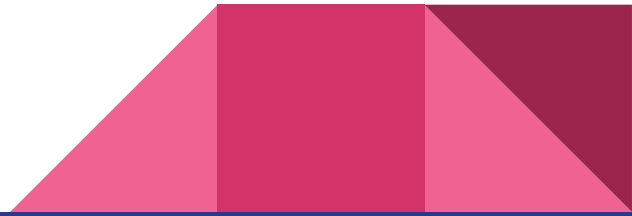
Locality

```
int sum2DArray(int arr[][], int n) {
        sum = 0;
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; i++) {
                        sum += a[j][i];
                }
        }
        return sum;
}
```

```
int sum2DArray(int arr[][], int n) {
        sum = 0;
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; i++) {
                        sum += a[i][j];
                }
        }
        return sum;
}
```

Optimization

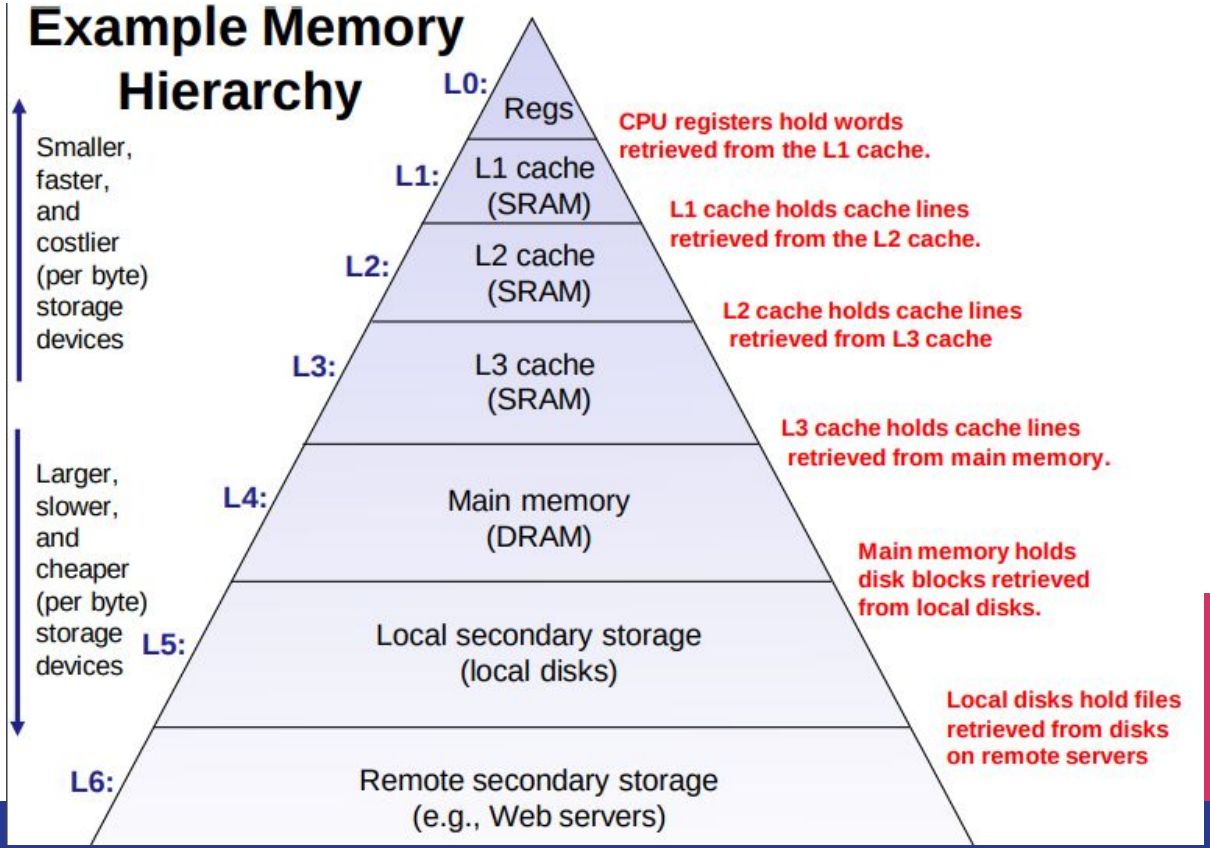
- Optimization will be back next week!
  - Concurrent Programming



Memory Hierarchy

- There are many kinds of memory
  - Registers
  - SRAM
  - DRAM
  - Disk
- Fast memory are smaller because they are more expensive

Memory Hierarchy

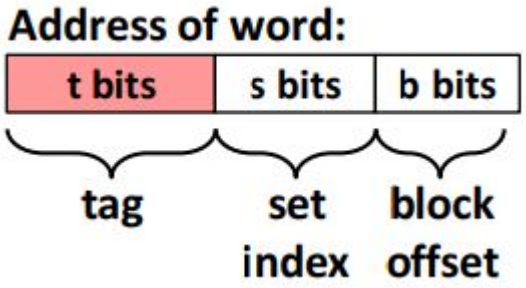


Cache Basic Terms

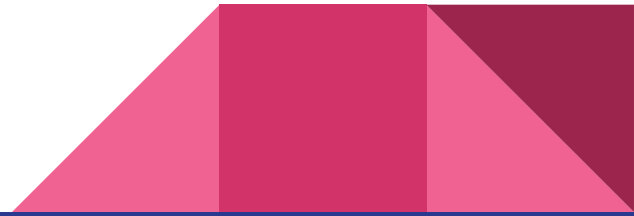
- Read hit
  - If the data that we want is in the cache, simply fetch the data from cache
- Read miss
  - If the data that we want is not in the cache, ask memory to fetch the data
  - Store the data fetched from memory in cache (temporal locality!)

Cache Basic Terms

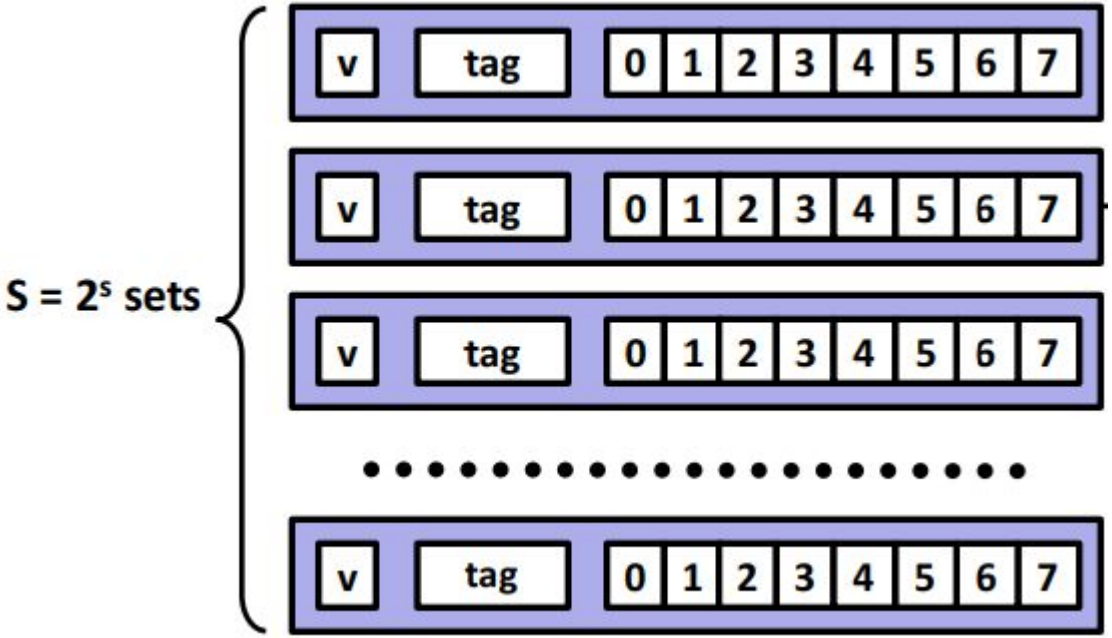
- Word
  - The unit that a machine uses when working with memory
  - In x86-64, word size is 64 bits (8 bytes)



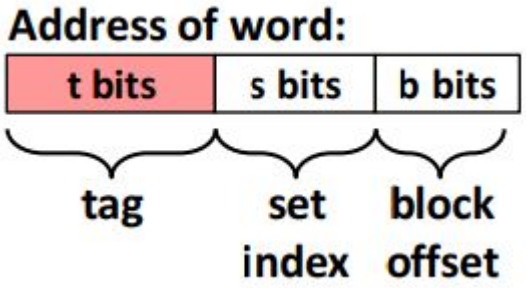
How does the data get fetched?



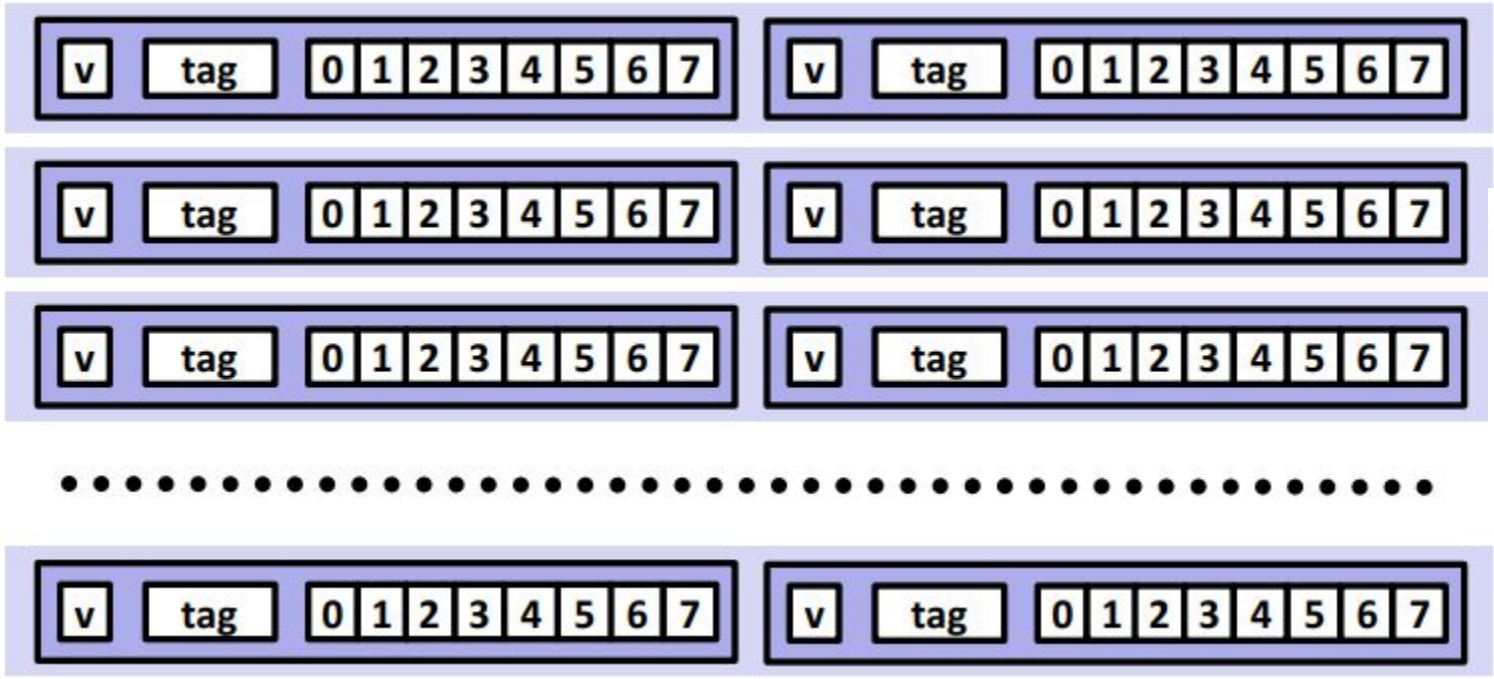
Direct Mapped Cache

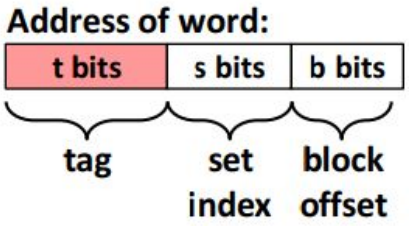


- Why do we need a valid bit?



Associated Mapped Cache (W = 2)

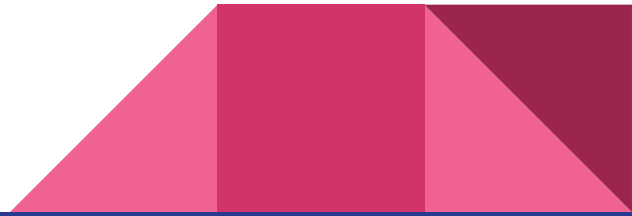




Stack Exploits

- Consider the following function

```
int foo() {
        long a = 0x7766554433221100;
        char c[16];
        gets(c);
}
```



Stack Exploits

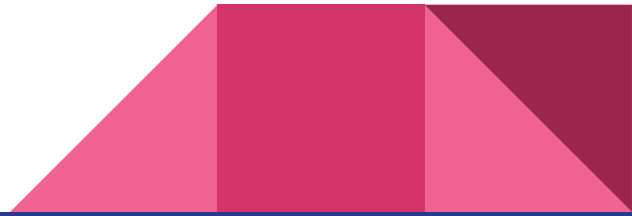
- Disassembled code looks like this:

```
0x400528 <+0>:     push   %rbp
0x400529 <+1>:     mov    %rsp,%rbp
0x40052c <+4>:     sub    $0x20,%rsp
0x400530 <+8>:     movabs $0x7766554433221100,%rax
0x40053a <+18>:    mov    %rax,-0x8(%rbp)
0x40053e <+22>:    lea    -0x20(%rbp),%rax
0x400542 <+26>:    mov    %rax,%rdi
0x400545 <+29>:    callq  0x4003b0 <gets@plt>
0x40054a <+34>:    leaveq
0x40054b <+35>:    retq
```

Adapted from CS 33 Discussion Slides by Uen-Tao Wang

Stack Exploits

- “gets” function takes a character pointer as an argument
- Then, it asks the user to input a character string where it then copies the string into the specified character pointer
- Let’s draw the stack frame of this function



Stack Exploits

- If you typed “DJ is my TA” (11 characters), 11 bytes will be occupied from c to c + 11
- What would happen if you type “DJ is my super awesome TA!” (26 characters)?
- In the C code, we only specified character array of size 16
- “gets” doesn’t care

Stack Exploits

- How can we change the value of a?
- How can we change the return address?
- What happens when we change the return address?
- How can we inject “malicious” codes?