1. Too much to cover in one lecture. So, will skip through some slides and you can take a look at them after the class.
What we need to know how to do. . . (what the compiler must be able to implement)

1. call
2. return
3. AR and local variables
4. return value
5. parameters
pushl *

does \%esp <- \%esp - 4
movl *, (\%esp)

popl *

does movl (\%esp), *
\%esp <- \%esp + 4

THE STACK
1. **call**
   - remember the return address
   - go to fcn

   This is such a common operation that the x86 architecture supports it with a single instruction:

   ```
   call fcn
   ```

   Does the equivalent of:

   ```
   push %eip
   jmp fcn
   ```
2. return

use the return address pushed onto the stack

```
ret
does the equivalent of
popl %eip
```
3. incorporate AR

For example, assume we need AR space for 3 ints.
gcc on x86 allocates AR space in multiples of 16 bytes.
prologue code

```
pushl  %ebp
movl  %esp, %ebp
subl  $16, %esp
```

Before fcn starts, but after the call instruction

After fcn "prologue"
epilogue code

leave  does  movl %ebp, %esp
       popl %ebp

ret  does  popl %eip
4. return value
On x86, return value goes in %eax (by convention)

```c
int b() {
    b:
        c();
        return 4;
}
```
```
call c
movl $4, %eax
leave
ret
```
5. parameters
No room in registers on the x86, so parameters go onto the stack.
Caller allocates space and places copies (for call by value). Child retrieves and uses copies.

```c
main () {
    a( 1, 2, 3 );
}
```

```assembly
main:    pushl %ebp
         movl %esp, %ebp
         subl $12, %esp
         movl $1, (%esp)
         movl $2, 4(%esp)
         movl $3, 8(%esp)
         call a
         leave
         ret
```
A view of the stack taken from the CSAPP textbook

- **Frame pointer**: %ebp
- **Stack pointer**: %esp
- **Argument build area**
- **Saved registers, local variables, and temporaries**
- **Argument n**
- **Argument 1**
- **Return address**
- **Saved %ebp**
1. The following slides step through the assembly instructions for the program simplefunctions1.c from Lecture 16 and show how the stack changes.

2. Keep the files simplefunctions1.c and simplefunctions1.objdump open while going over the following slides that show the stack layout.
### Prologue: After executing Instruction: 0x80483be: push `%ebp`

<table>
<thead>
<tr>
<th><code>%esp</code></th>
<th><code>%ebp</code> of main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prologue: After executing Instruction: 0x80483bf: mov %esp,%ebp

<table>
<thead>
<tr>
<th>%esp</th>
<th>%ebp main’s caller</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Addresses
Lower at bottom
Higher at top
Prologue: After executing Instruction: 0x80483c1: sub $0x18,%esp
Allocating Space for local variables: a, b, c and parameters to func1
(gcc allocates in multiples of 16 bytes)

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b</td>
<td></td>
</tr>
<tr>
<td>int a</td>
<td></td>
</tr>
</tbody>
</table>

%esp ➔

Addresses
Lower at bottom
Higher at top
After executing Instruction: 0x80483c4: movl $0xc,-0xc(%ebp)

Initializing local variable a;

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b</td>
<td></td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
<td></td>
</tr>
</tbody>
</table>

%esp →

Addresses
Lower at bottom
Higher at top
### After executing Instruction : `0x80483cb: movl $0x18,-0x8(%ebp)`

Initializing local variable b;

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b : 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
<td></td>
</tr>
</tbody>
</table>

### Addresses
- Lower at bottom
- Higher at top

- `%ebp` is the base pointer for the caller's stack frame.
- `%esp` points to the top of the caller's stack.

- Variables `c`, `b`, and `a` are initialized with the values 12, 24, and 24, respectively.

- The diagram illustrates the layout of the stack frame, with `%ebp` pointing to the caller's stack and `%esp` pointing to the top of the stack.
After executing Instruction: 0x80483d2: `mov -0x8(%ebp),%eax`

Fetch b in %eax;

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
<td></td>
</tr>
</tbody>
</table>

%esp ↗

Addresses
Lower at bottom
Higher at top
After executing Instruction: `0x80483d5: mov %eax,0x4(%esp)`

Set up parameter `b`;

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp →</td>
<td></td>
</tr>
</tbody>
</table>

Addresses
Lower at bottom
Higher at top
After executing Instruction: 0x80483d9: mov -0xc(%ebp),%eax
Fetch a into %eax;

%ebp main’s caller ← %ebp

- int c
- int b: 0x18 == 24
- int a : 0xc == 12

%esp →

Addresses
Lower at bottom, Higher at top
After executing Instruction: \texttt{0x80483dc: mov %eax, (%esp)}

Set up parameter a:

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

Addresses

Lower at bottom

Higher at top
After executing Instruction: 0x80483df: call 8048394 <func1>
Call function func1: which pushes return address on stack and jumps to func1;

%ebp main’s caller ← %ebp

- int c
- int b: 0x18 == 24
- int a: 0xc == 12

%esp → Return address: 0x80483e4

Addresses
Lower at bottom
Higher at top
Prologue: After executing Instruction: 0x8048394: push %ebp
Push %ebp of main into stack

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>Return address: 0x80483e4</td>
<td></td>
</tr>
</tbody>
</table>

Addresses:
Lower at bottom
Higher at top
Prologue: After executing Instruction: 0x8048395: mov %esp,%ebp
Setup frame for func1

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>Return address: 0x80483e4</td>
</tr>
</tbody>
</table>

%esp → %ebp of main ← %ebp
Prologue: After executing Instruction: 0x8048397: sub $0x10,%esp
Allocate space for local variables: diff, sum (gcc allocates in multiples of 16 bytes)

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
</tr>
</tbody>
</table>

Return address: 0x80483e4

<table>
<thead>
<tr>
<th>%ebp of main</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sum</td>
</tr>
<tr>
<td>int diff</td>
</tr>
</tbody>
</table>

%esp ➔ %ebp ➔ %ebp
After executing Instruction: 0x804839a: mov 0xc(%ebp),%eax

Fetch second parameter into %eax

General rule is: parameter i is at offset (4+4*i) from %ebp

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>int c</td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
</tr>
</tbody>
</table>

Return address: 0x80483e4

<table>
<thead>
<tr>
<th>%ebp of main</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>int sum</td>
</tr>
<tr>
<td>int diff</td>
</tr>
</tbody>
</table>

Lower at bottom
Higher at top

Addresses

%ebp of main ← %ebp

%esp →
After executing Instruction: 0x804839d: mov 0x8(%ebp),%edx
Fetch first parameter into %edx
General rule is: parameter i is at offset (4+4*i) from %ebp

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
</tr>
</tbody>
</table>

b

a

Return address: 0x80483e4

%ebp of main

<table>
<thead>
<tr>
<th>%ebp of main</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sum</td>
</tr>
<tr>
<td>int diff</td>
</tr>
</tbody>
</table>

%esp →
After executing Instructions:

0x80483a0: mov %edx,%ecx
0x80483a0: mov %edx,%ecx
0x80483a2: sub %eax,%ecx
0x80483a4: mov %ecx,%eax

These instructions calculate x-y and store it in %eax
After executing Instruction: 0x80483a6: mov %eax,-0x8(%ebp)

Store result in diff

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
</tr>
</tbody>
</table>

a

b

Return address: 0x80483e4

%ebp of main ← %ebp

<table>
<thead>
<tr>
<th>int sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>int diff = x-y</td>
</tr>
</tbody>
</table>

%esp ↗

Addresses
Lower at bottom
Higher at top
After executing Instructions:

0x80483a9: mov 0xc(%ebp),%eax
0x80483ac: mov 0x8(%ebp),%edx
0x80483af: lea (%edx,%eax,1),%eax

These instruction fetch parameters x, y into temporary registers, calculate x+y into register %eax
### After executing Instruction:

0x80483b2: mov %eax,-0x4(%ebp)

Store result in sum

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>Return address: 0x80483e4</td>
<td>← %ebp</td>
</tr>
</tbody>
</table>

| %ebp of main                        | ← %ebp                    |
| int sum = x+y                        |                           |
| int diff = x-y                       |                           |

| %esp →                              |                           |
After executing Instructions:
0x80483b5: mov -0x4(%ebp),%eax
0x80483b8: imul -0x8(%ebp),%eax

These instructions fetch sum into %eax, and then calculate product of sum and diff into register %eax

Since by x86 conventions, the result of a function is left in %eax, we do not need to anything further.
After executing First part of Instruction: 0x80483bc: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

- %ebp main’s caller
  - int c
  - int b: 0x18 == 24
  - int a: 0xc == 12

- Return address: 0x80483e4

- %esp → %ebp of main
  - int sum
  - int diff = x-y
After executing Second part of Instruction: 0x80483bc: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>%esp → Return address: 0x80483e4</td>
<td></td>
</tr>
<tr>
<td>%ebp of main</td>
<td></td>
</tr>
<tr>
<td>int sum</td>
<td></td>
</tr>
<tr>
<td>int diff = x-y</td>
<td></td>
</tr>
</tbody>
</table>
After executing Instruction: 0x80483bd: ret
Return to main by popping into %eip

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
<td></td>
</tr>
</tbody>
</table>

Return address: 0x80483e4

%ebp of main

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int sum</td>
<td></td>
</tr>
<tr>
<td>int diff = x-y</td>
<td></td>
</tr>
</tbody>
</table>
After executing Instruction: \texttt{0x80483e4: mov \%eax,-0x4(\%ebp)}

Store result into local variable \texttt{c}

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c = return value of func1()</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a: 0xc == 12</td>
<td></td>
</tr>
</tbody>
</table>

Return address: 0x80483e4

%ebp of main

int sum

int diff = x-y

Addresses

Lower at bottom

Higher at top

Return address: 0x80483e4

%ebp of main

int sum

int diff = x-y
After executing Instruction: `0x80483e7: mov $0x0,%eax`

Store result of main (value 0) into %eax by x86 convention

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c = return value of func1()</td>
<td></td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
<td></td>
</tr>
</tbody>
</table>

%esp →

b

a

Return address: 0x80483e4

%ebp of main

int sum

int diff = x-y
After executing Part 1 of Instruction: 0x80483ec: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

<table>
<thead>
<tr>
<th>%esp →</th>
<th>%ebp main’s caller</th>
<th>← %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int c = return value of func1()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int b: 0x18 == 24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int a : 0xc == 12</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return address: 0x80483e4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp of main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int diff = x-y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After executing Part 2 of Instruction: 0x80483ec: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

<table>
<thead>
<tr>
<th>%ebp main’s caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>int c = return value of func1()</td>
</tr>
<tr>
<td>int b: 0x18 == 24</td>
</tr>
<tr>
<td>int a : 0xc == 12</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>

Return address: 0x80483e4

%ebp of main

| int sum |
| int diff = x-y |
IA 32 convention

**caller save**

%eax  %edx  %ecx

**callee save**

%ebx  %esi  %edi

Which is %ebp ?
Stack Smashing

- Caused by **Buffer overflow**
- Attacker can store arbitrary code in the stack and execute it leading to attacks that can comprise privacy of your data or even destroy it.
- As a programmer, avoid using versions of library functions that can cause buffer overflow issues. E.g use `strncpy()` instead of `strcpy()`
- Also, check for return values of library functions and handle appropriately.
Stack Smashing

- **Stack Randomization**: Location of the Stack in the memory layout of the program varies between executions of a program. (Other segments can also be relocated: Address Space Layout)

- **Reduce executable code locations**: mark data, stack, heap as not executable.

- **Canary value** is inserted by compiler during function call and checked just before return to detect buffer overflow attacks.
<table>
<thead>
<tr>
<th>Type</th>
<th>What cached</th>
<th>Where cached</th>
<th>Latency (cycles)</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU registers</td>
<td>4-byte or 8-byte word</td>
<td>On-chip CPU registers</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-chip TLB</td>
<td>0</td>
<td>Hardware MMU</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte block</td>
<td>On-chip L1 cache</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte block</td>
<td>On/off-chip L2 cache</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>L3 cache</td>
<td>64-byte block</td>
<td>On/off-chip L3 cache</td>
<td>30</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Controller firmware</td>
</tr>
<tr>
<td>Network cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
The patterns come from the fetch-execute cycle:

1. Fetch instruction
2. Update PC
3. Decode
4. Get operands
5. Do operation
6. Put result(s) away

They exhibit locality.
temporal locality

Recently referenced memory locations are likely to be referenced again (soon!)

```
loop:  instr 1  @ A1  
      instr 2  @ A2  
      instr 3  @ A3  
      jmp/b loop  @ A4  
```

Instruction stream references:

```
A1 A2 A3 A4 A1 A2 A3 A4 A1 A2 A3 ...
```

Note that the same memory location is repeatedly read (for the fetch).
spatial locality

Memory locations near to referenced locations are likely to also be referenced.

Code must do something to each element of the array.

Must load each element.
The fetch of the code exhibits a high degree of spatial locality.

I2 is next to I1.

If these instructions are not branches, then we fetch

I1
I2
I3
etc.
Needed terminology:

\[
\text{miss ratio} = \frac{\# \text{ of misses}}{\text{total \# of accesses}}
\]

\[
\text{hit ratio} = \frac{\# \text{ of hits}}{\text{total \# of accesses}}
\]

or \[1 - \text{miss ratio}\]

You already assumed that \(\text{total \# of accesses} = \# \text{ of misses} + \# \text{ of hits}\)
When a memory access causes a miss, place that location's bytes and its neighbors (spatial locality) into the cache. Keep the block of bytes there for as long as possible (temporal locality).

A statistic to measure how well this works:

$$\text{Average Memory Access Time} = T_c + (\text{ratio}) (T_m)$$
Quick example:

\[ T_c = 1 \text{ nsec} \]
\[ T_m = 20 \text{ nsec} \]

hit ratio is \( .98 \)
for measured program

\[ \text{AMAT} = 1 + (0.02)(20) \]
\[ = 1.4 \text{ nsec} \]

Note: individual memory access takes either
1 nsec (hit)
or 21 nsec (miss).
Generic Cache Organization

Set 0:
- Valid: 1 valid bit per line
- Tag: \( t \) tag bits per line
- \( B = 2^b \) bytes per cache block

\[
\begin{array}{cccc}
\text{Valid} & \text{Tag} & 0 & 1 & \cdots & B-1 \\
\vdots
\end{array}
\]

Set 1:
\[
\begin{array}{cccc}
\text{Valid} & \text{Tag} & 0 & 1 & \cdots & B-1 \\
\vdots
\end{array}
\]

Set \( S - 1 \):
\[
\begin{array}{cccc}
\text{Valid} & \text{Tag} & 0 & 1 & \cdots & B-1 \\
\vdots
\end{array}
\]

\( S = 2^s \) sets

Cache size: \( C = B \times E \times S \) data bytes

\( B \) bytes per cache block

\( E \) lines per set

\( S = 2^s \) sets
Looking up a memory address in Direct Mapped Cache

Set 0:

Set 1:

Set S-1:

Looking up a memory address in Direct Mapped Cache
This cache is called
direct mapped
or
1-way set associative
or
set associative, with a set size of 1

Each index # maps to exactly 1 block frame
Lookup algorithm:
(cache receives address)

[use index to identify frame

if frame is valid

  if frame’s tag matches address’ tag
    HIT
  else
    MISS

else
  MISS
Looking up a memory address in Direct Mapped Cache

(1) The valid bit must be set

(2) The tag bits in the cache line must match the tag bits in the address

(3) If (1) and (2), then cache hit, and block offset selects starting byte.

Selected set (i):

\[
\begin{array}{cccccc}
1 & 0110 & \_ & \_ & w_0 & w_1 \ w_2 \ w_3 \\
\end{array}
\]
On a miss

- Send the memory request to main memory.
- Memory returns the entire block containing the needed byte/word.
- Place the block into the frame.
  - Set the tag bits
  - Mark the frame valid.

And, while doing this, extract the byte/word and return it to the processor, completing the memory access.
Types of Misses

- **Compulsory or cold misses:** Cache is empty to start with and will miss.

- **Conflict misses:** Cache has space but because objects map to the same cache block they keep missing.

- **Capacity misses:** Cache does not have space because size of the working set exceeds the size of the cache.
To reduce conflict misses
increase set associativity

2-way set associative
2 blocks per set (line)

4-way set associative
Set Associative Cache Organization

Set 0:
- Valid
- Tag
- Cache block

Set 1:
- Valid
- Tag
- Cache block

Set \( S - 1 \):
- Valid
- Tag
- Cache block

\( E = 2 \) lines per set
larger set size tends to lead to higher hit ratio (due to fewer conflict misses)

amount of circuitry goes up, leading to increase in $T_c$
Looking up a memory address in Set Associative Cache

- **Valid**
- **Tag**
- **Cache block**

```
set 0:
Valid  Tag  Cache block
Valid  Tag  Cache block

set 1:
Valid  Tag  Cache block
Valid  Tag  Cache block

set S -1:
Valid  Tag  Cache block
Valid  Tag  Cache block
```

```
\[
t \text{bits} \quad s \text{ bits} \quad b \text{ bits}
\]
\[
\begin{array}{c}
0 \quad 0 \quad 0 \quad 1
\end{array}
\]
```

Selected set

```
\[
\text{m-1} \quad \text{Tag} \quad \text{Set index} \quad \text{Block offset}
\]
```

```m-1``` bits

```s``` bits

```b``` bits

```0``` bits
Looking up a memory address in Set Associative Cache

(1) The valid bit must be set

(2) The tag bits in one of the cache lines must match the tag bits in the address

(3) If (1) and (2), then cache hit, and block offset selects starting byte
Fully Associative Cache Organization

More circuitry and hence more expensive than Direct mapped and Set Associative Caches

Set 0:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Cache block</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ E = \frac{C}{B} \] lines in the one and only set
Looking up a memory address in Fully Associative Cache

The entire cache is one set, so by default set 0 is always selected.

\[ \begin{array}{ccc}
    & t \text{ bits} & b \text{ bits} \\
\end{array} \]

\[ \begin{array}{cc}
    \text{Tag} & \text{Block offset} \\
\end{array} \]

Set 0:

- Valid
- Tag
- Cache block

- Valid
- Tag
- Cache block

- Valid
- Tag
- Cache block

- \ldots
Looking up a memory address in Fully Associative Cache

=1 ? (1) The valid bit must be set

Entire cache

(2) The tag bits in one of the cache lines must match the tag bits in the address

= ?

(3) If (1) and (2), then cache hit, and block offset selects starting byte

0110 100

m-1

Tag

Block offset

w_0  w_1  w_2  w_3
Implementing writes

1. write through
   change data in the cache, and send the write to main memory

   slow 😞 , but very little circuitry 😊
2. **write back**
   - at first, change data in the cache
   - write to memory only when necessary

dirty bit is set on a write, to identify blocks to be written back to memory

when a program completes, all dirty blocks must be written to memory...
2 write back (continued)

- faster 🌈
  multiple stores to the same location result in only 1 main memory access

- more circuitry 😞
  - must maintain the dirty bit
  - dirty miss: a miss caused by a read or write to a block not in the cache, but the required block frame has its dirty bit set. So, there is a write of the dirty block, followed by a read of the requested block.
Writing during cache miss: (Two approaches)

- **Write Alloc**: Load block in cache and update word (often used along with Write back)
- **Write No-Alloc (a.k.a. Write around)**: Just update memory (often used along with Write through)
How about 2 separate caches?

**I-cache**
- for instructions only
- can be rather small, and still have excellent performance.

**D-cache**
- for data only
- needs to be fairly large
We can send memory accesses to the 2 caches independently...

😊 (increased parallelism)
Strided Access Patterns

int i, j, sum = 0;
for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)
        sum += a[i][j]

What if: sum += a[j][i]?
Writing Cache Friendly Code

1. Focus on the inner loops where bulk of computation and memory accesses occur

2. Maximize spatial locality by reading data objects sequentially with stride 1

3. Maximize temporal locality by reading a data object as often as possible once it has been read from memory.
**Hard Disk Overview**

**Platters:** Consists of two sides or surfaces coated with magnetic recording material

**Spindle:** Spins the platter at fixed rotational rate (5400 to 15K revolutions per minute - RPM)
**Track**: A ring on the magnetic surface made up of sectors.

**Sectors**: Each track is partitioned into a collection of Sectors. Gaps in between sectors store formatting bits that identify sectors.

**Cylinder**: Collection of tracks on all surfaces that are equidistant from the center of the spindle.
Access Time for a Sector

\[ T_{I/O} = T_{seek} + T_{rot} + T_{xfr} \]

Seek Time: Time taken to position the head over the track that contains the target sector.

Rotational Latency: Time taken for the first bit of the target sector to pass under the head.

Transfer time: Time taken to read or write the contents once the first bit of the target sector is under the head.
Accessing Disks

Memory mapped I/O: A block of addresses in the address space is reserved for communicating with the I/O devices.

Each of these addresses is called an I/O Port.

Every device attached to a bus will have a corresponding I/O Port.

Small Communication with the I/O devices happens through the I/O port.
Suppose magnetic disk is mapped to I/O port 0xa0

CPU initiates disk read by 3 store instructions:

1) Send a command word indicating to perform a read along with other parameters like interrupt on completion of DMA.

2) Send the logical block number that must be read.

3) Indicate the main memory address where the contents of the disk sector should be stored.
Direct Memory Access

After issuing the request, CPU executes other instructions. Note: A 3GHz processor with a 0.33 ns clock cycle can execute 9 million instructions in 3ms.

After receiving the read command from the CPU, the disk controller fetches data from the right sector and transfers it directly to the memory without the involvement of the CPU.

After DMA is complete, disk controller notifies by sending interrupt signal to the CPU.
Disk Controller reads the data and DMA transfer to memory
Disk Controller notifies CPU with an interrupt after DMA completes
Solid State Disks

Made up of flash memory chips that store data instead of the magnetic surfaces in a conventional disk.

A Flash Translation Layer translates logical block addresses to accesses to the right block and page within the flash memory chip.
SSD vs. Magnetic Disks

Advantages: SSDs have no moving parts and hence are more quiet during operation, faster, need less power and are more rugged.

Disadvantages:

1. Possibility of wear out after several program-erase cycles. This is mitigated by the flash translation layer.
2. More expensive than magnetic disks.
Storage Trends

DRAM and disk performance are lagging behind CPU performance. Though SRAM performance also lags, it is roughly keeping up.
Unix I/O

Unix file is a sequence of bytes.

All I/O devices including networks, terminals, disks are modeled as files. E.g. /dev/sda for disk, /dev/tty for terminal

This way, programmers use a simple low-level interface called the Unix I/O to interact with all I/O devices

Some examples of operations on files:

• Opening and Closing files
• Reading and writing files
• Changing the current file offset
• Read metadata about a file
Unix I/O APIs

```c
int open(char *filename, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, const void *buf, size_t n);
off_t lseek(int fd, off_t offset, int whence);
int fstat(int fd, struct stat *buf);
```

Each process created by a Unix shell begins life with three open files: standard input (descriptor 0), standard output (descriptor 1), and standard error (descriptor 2)
Sharing Files
(Descriptions from CSAPP Text)

**Descriptor table:** Each process has its own separate descriptor table whose entries are indexed by the process’s open file descriptors. Each open descriptor entry points to an entry in the file table.

**File table:** The set of open files is represented by a file table that is shared by all processes. Each file table entry consists of (for our purposes) the current file position, a reference count of the number of descriptor entries that currently point to it, and a pointer to an entry in the v-node table. Closing a descriptor decrements the reference count in the associated file table entry. The kernel will not delete the file table entry until its reference count is zero.

**v-node table:** Like the file table, the v-node table is shared by all processes. Each entry contains most of the information in the stat structure, including the st_mode and st_size members. v-node table and the related VFS(Virtual File System) interface is the separation between specific file system implementations and the generic file system operations.
Typical OS kernel data structures for open files.

- **Descriptor table** (one table per process)
  - stdin: fd 0
  - stdout: fd 1
  - stderr: fd 2, fd 3, fd 4

- **Open file table** (shared by all processes)
  - File A
    - File pos
    - refcnt=1
    - ...
  - File B
    - File pos
    - refcnt=1
    - ...

- **v-node table** (shared by all processes)
  - File access
  - File size
  - File type
    - ...
  - File access
  - File size
  - File type
    - ...

The diagram illustrates the relationship between the descriptor table, open file table, and v-node table, showing how files are referenced and managed by processes in the OS kernel.
C Standard library (libc) provides:

- `fopen`, `fclose`
- `fread`, `fwrite`
- `fgets`, `fputs`
- `fscanf`, `fprintf`

Standard I/O models files as streams and buffers I/O
<table>
<thead>
<tr>
<th>Unix I/O vs Standard I/O vs RIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix I/O is the lowest form of interface and provides basis for both Standard I/O and RIO.</td>
</tr>
<tr>
<td>Unix I/O can be used in signal handlers.</td>
</tr>
<tr>
<td>Unix I/O does not deal with short counts like Standard I/O or RIO.</td>
</tr>
<tr>
<td>Unix I/O does not have buffering and hence is inefficient.</td>
</tr>
<tr>
<td>Standard I/O solves both short counts and buffering issues of Unix I/O but cannot be used for networking applications due to poorly documented restrictions hence the need for RIO.</td>
</tr>
</tbody>
</table>
Different I/O interfaces in perspective

C application program

Standard I/O functions

Unix I/O functions (accessed via system calls)

RIO functions

fopen  fdopen
fread  fwrite
fscanf  fprintf
sscanf  sprintf
fgets  fputs
fflush  fseek
fclose

open  read
write  lseek
stat  close

rio_readn
rio_writen
rio_readinitb
rio_readlineb
rio_readnb
memory mapped

Keyboard

Display

addresses 0xfffff0008 0xfffff000c

Data
Status

addresses 0xfffff0010 0xfffff0014

Data
Status

P

M
Now, driver code uses a **spin wait loop**
(to implement blocking I/O)

```assembly
kb_spin:
    testl Keyboard_Status, Keyboard_Status
    jz kb.spin
    movl Keyboard_Data, %eax
    ret from syscall

disp-spin:
    testl Display_Status, Display_Status
    jz disp-spin
    movl %eax, Display_Data
    ret from syscall
```
Byte transfers are OK,  
But, what about faster devices that like to transfer more than a byte?

the solution: DMA  
Direct Memory Access
Issue for spin wait loop implementations:

One byte *only* in `_Data` has the potential for an incorrect result.

For example, if the user types 2 characters on the keyboard before `getchar()` is called.

The needed fix introduces a kernel-maintained *queue* for each device. Then, the kernel *polls* to check status bits and handle any ready devices.
Because polling is so inefficient,

instead of

OS

ask

device 1 ready?  device 2 ready?  ...  device n ready?
Turn the situation upside down

device 1  device 2  ...  device n
Anatomy of an exception

An exception is an abrupt change in control flow.

Examples: div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C
# Classes of Exceptions

<table>
<thead>
<tr>
<th></th>
<th>Classes</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Interrupts (Asynchronous)</strong></td>
<td>Always return to next instruction.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Traps &amp; System Calls (Synchronous)</strong></td>
<td>Always return to next instruction.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Faults (Synchronous)</strong></td>
<td>Might return to next instruction.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Aborts (Synchronous)</strong></td>
<td>Never returns</td>
</tr>
</tbody>
</table>
Interrupts

Examples of Interrupts:

- Timer interrupt
- Arrival of a packet from a network
- When a key is pressed on the keyboard
- When the mouse is moved
Traps

Traps are intentionally issued by executing an instruction.

Example: System calls

1. Application makes a system call
2. Control passes to handler
3. Trap handler runs
4. Handler returns to instruction following the syscall
Faults

Faults result from error conditions that might be correctable.

Examples: Page fault, Divide error

(1) Current instruction causes a fault

(2) Control passes to handler

(3) Fault handler runs

(4) Handler either reexecutes current instruction or aborts.
Aborts

Aborts result from unrecoverable fatal errors.

Example: parity errors due to DRAM bit corruption
Exception Table

Exception table

<table>
<thead>
<tr>
<th>Exception table</th>
<th>Code for exception handler 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Code for exception handler 1</td>
</tr>
<tr>
<td>1</td>
<td>Code for exception handler 2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td>Code for exception handler n-1</td>
</tr>
</tbody>
</table>
Exception Table lookup

Exception is similar to procedure calls except for some important differences:

- Return address is not the next instruction always
- Push EFLAGS register also onto kernel stack
- Run exception handler in kernel mode

![Diagram showing Exception Table lookup]
# IA32 Exception Table

From CSAPP text book:

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-127</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>System call</td>
<td>Trap</td>
</tr>
<tr>
<td>129-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>
Rather important, but not covered in textbook:

If running a handler, and a new interrupt request arrives, what should happen?

* Continue on, complete handling of current interrupt, then, when done, deal with new request? (probably) non reentrant

* Interrupt the handling of this interrupt? reentrant
Consider the x86 instruction:

\texttt{cli} \hspace{5mm} \text{clear IF}

\textit{What happens if an application includes this \texttt{cli} instruction?}

Irrelevant (to this discussion) x86 instruction:

\texttt{sti} \hspace{5mm} \text{set IF}
Reentrant timeline

dev 2 interrupts are higher priority
Processes

1. **Process**: an instance of a program in execution
2. Give the illusion that our program is the only one currently running in the system
3. **Process Context** consists of state including: Virtual Memory Layout, CPU registers, file descriptors, environment variables etc.
4. Key abstraction provided by a process:
   a. Independent logical control flow
   b. Private address space
Logical Control Flow

The single physical control flow of CPU is partitioned into logical control flows of several processes.

![Diagram showing control flows of Process A, Process B, and Process C over time.](image-url)
Concurrent and Parallel Flows

1. A logical flow whose execution overlaps in time with another flow is called a **concurrent flow**.

2. E.g. A & B are concurrent in previous slide, A & C are also concurrent while B & C are not concurrent.

3. **Parallel flows**: A subset of concurrent flows where the individual flows run on multiple cores or machines in parallel.
Private Address Space

1. A process provides each program the illusion that it has exclusive use of the system’s address space through virtual memory.

2. This concept of private address space per process makes writing programs much easier rather than dealing with physical memory addresses. (e.g. frees the programmer from managing the physical memory resources)
Privileged Mode

1. **User Mode**: Cannot execute privileged instructions like one that halts the CPU. Also cannot access kernel area of address space.

2. **Kernel Mode (Privileged/Supervisor Mode)**: Can execute any instruction and access any memory location.

Process runs application code in user mode and switches to kernel mode only via an exception like interrupt, system call etc.
Signals

Unix Signal is a higher level software form of exceptional control flow.

A Signal is a small message that notifies a process that an event of some type has occurred in the system.

Processes and the Operating System can interrupt other processes using Signals.
Signal Handling

Control flow while handling signals.

1. Signal received by process
2. Control passes to signal handler
3. Signal handler runs
4. Signal handler returns to next instruction
Signals

Low level exception we discuss in last lecture are handled by the Operating System’s exception handlers and are not visible to user level processes.

Signals provide a mechanism for exposing these low level exceptions to user processes.

E.g. - If a process executes an illegal instruction, then OS kernel sends a SIGILL signal.

- If a process divides by zero, then the OS kernel sends the process a SIGFPE signal.
Signals

List of Linux Signals in “man 7 signal”

The transfer of a signal occurs in two distinct steps:

1) Sending a signal
2) Receiving a signal
Sending Signals

OS Kernel sends/delivers a signal to a destination process by updating the process context.

A signal can be sent in two ways:
1) Kernel has detected an event like divide-by-zero or termination of a child process
2) A process has invoked the kill function to explicitly request the kernel to send a signal to the destination process.
Four ways of Sending Signals

1. With `/bin/kill` program:
   a. “/bin/kill -9 pid” sends signal 9 (SIGKILL) to process 15213
   b. “/bin/kill -9 -pid” sends signal 9 to all processes in process group 15213

2. Sending signals from the keyboard:
   a. Typing Ctrl-C on shell sends SIGINT signal to every process in the foreground process group.
   b. Typing Ctrl-Z sends SIGTSTP to every foreground process and the result is to suspend them.
Four ways of Sending Signals

3. Sending signals with the kill function:
   ```c
   int kill(pid_t pid, int sig);
   ```
   • positive pid sends signal to that process
   • negative pid sends signal to every process in process group abs(pid)

4. Sending signals with the alarm function:
   ```c
   unsigned int alarm(unsigned int secs)
   ```
   • A process can send SIGALRM signals to itself by calling the alarm function.
Receiving Signals

Before kernel returns control to a process after executing a exception handler, it checks the set of unblocked pending signals.

- If the set is empty (the usual case), then control goes to the next instruction.

- If the set is not empty, then OS kernel chooses one of the pending signals and forces the process to receive the signal.
Receiving Signals

Each signal has a predefined default action which is one of:

1) The process terminates
2) The process terminates and dumps core
3) The process stops until restarted by a SIGCONT signal
4) The process ignores the signal
Receiving Signals

However, a process can choose to install its own modified default action for all signal except SIGSTOP and SIGKILL using:

```c
sighandler_t signal(int signum, sighandler_t handler);
```

Signal handlers are yet another example of concurrency.
Receiving Signals

The signal function can change the action associated with a signal in one of three ways:

1) If handler is SIG_IGN, then signals of type signum are ignored.
2) If handler is SIG_DFL, then the action for signals of type signum reverts to the default action.
3) Otherwise, handler is the address of a user defined function called signal handler that will be invoked whenever the process receives a signal of type signum.

Example program for user defined signal handler function.
Signal Handing Issues

- Pending signals are blocked: Unix signal handlers block pending signals of the type currently being processed by the handler.

- Pending signals are not queued: There can be atmost one pending signal of any particular type.

- System calls can be interrupted: In some systems, interrupted system calls will return immediately to user with an error condition.