

# 15-213: Final Exam Review

Your TAs

# Final Exam Logistics

- **May 1st from 1:00 pm - 4:00 pm**
- Rooms in Hall of Arts and Posner
- **ON GRADESCOPE, IN PERSON**
- Physical Cheat sheets - 2 double sided 8.5 x 11 in.
- Show ID and cheatsheet to TA
- You should have received an email with more detailed logistics

# Final Exam Topics

- Low-level C (structs, alignment)
- Bits, Bytes, Ints (datalab)
- Assembly (bomblab)
- Stacks (attacklab)
- Caches (cachelab)
- Malloc and Dynamic Memory Allocation (malloclab)
- Virtual Memory
- Processes, Signals, IO (tshlab)
- Proxy, Threads, Synchronization (proxylab)

# Assembly

# Assembly

- Typical questions asked
  - Given a function, look at assembly to fill in missing portions
  - Given assembly of a function, intuit the behavior of the program
  - (More rare) Compare different chunks of assembly, which one implements the function given?
- Important things to remember/put on your cheat sheet:
  - Memory Access formula:  $D(Rb, Ri, S)$
  - Distinguish between mov/lea instructions
  - Callee/Caller save regs
  - Condition codes and corresponding eflags

# Assembly

Consider the following x86-64 code (Recall that `%c1` is the low-order byte of `%rcx`):

```
# On entry:
#   %rdi = x
#   %rsi = y
#   %rdx = z

4004f0 <mysterious>:
 4004f0:  mov    $0x0,%eax
 4004f5:  lea   -0x1(%rsi),%r9d
 4004f9:  jmp   400510 <mysterious+0x20>
 4004fb:  lea   0x2(%rdx),%r8d
 4004ff:  mov   %esi,%ecx
 400501:  shl   %c1,%r8d
 400504:  mov   %r9d,%ecx
 400507:  sar   %c1,%r8d
 40050a:  add   %r8d,%eax
 40050d:  add   $0x1,%edx
 400510:  cmp   %edx,%edi
 400512:  ja    4004fb <mysterious+0xb>
 400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = ; ;  ){
        e = i + 2;
        e = ;
        e = ;
        d = ;
    }
    return ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

e = %r8d



# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = ; ;  ){
        e = i + 2;
        e = ;
        e = ;
        d = ;
    }
    return ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

Loop end: add 1, compare, iterate

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = ; ;  ){
        e = i + 2;
        e = ;
        e = ;
        d = ;
    }
    return ;
}
```

`cmp %edx, %edi` => `(edi - edx > 0)`, same as `x > i`

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

We know that  $e = \%r8d...$

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

Where did %cl come from?

<b>%ecx</b>	<b>%cx</b>	<b>%ch</b>	<b>%cl</b>
-------------	------------	------------	------------

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

Again, e = %r8d...

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja    4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

What's left?

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```

int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = z ; x > i ; i++){
        e = i + 2;
        e = e << y ;
        e = e >> (y - 1) ;
        d = e + d ;
    }
    return ;
}

```

```

# On entry:
# %rdi = x
# %rsi = y
# %rdx = z

4004f0 <mysterious>:
4004f0:  mov  $0x0,%eax
4004f5:  lea  -0x1(%rsi),%r9d
4004f9:  jmp  400510 <mysterious+0x20>
4004fb:  lea  0x2(%rdx),%r8d
4004ff:  mov  %esi,%ecx
400501:  shl  %cl,%r8d
400504:  mov  %r9d,%ecx
400507:  sar  %cl,%r8d
40050a:  add  %r8d,%eax
40050d:  add  $0x1,%edx
400510:  cmp  %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq

```



# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

# Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i =  ;  ;  ){
        e = i + 2;
        e =  ;
        e =  ;
        d =  ;
    }
    return  ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z
```

```
4004f0 <mysterious>:
4004f0:  mov    $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %r9d,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja   4004fb <mysterious+0xb>
400514:  retq
```

C stuff

# Structs

## Padding and Alignment Rules:

### Primitives:

- Char: 1-byte aligned (doesn't matter)
- Short: 2-byte aligned
- Int: 4-byte aligned
- Word/Pointer/Long: 8-byte aligned

# Structs

Padding and Alignment Rules:

Struct within a struct:

Uses the alignment of the biggest primitive within the struct. So if the struct has a pointer, it'll itself have 8 byte alignment in other structs.



# Structs

```
struct foo {  
    int *p;  
    char b;  
    char c;  
    int x;  
    short y;  
    char[4] buf;  
};
```

How would this be represented?

# Structs

```
struct foo {  
    int *p;  
    char b;  
    char c;  
    int x;  
    short y;  
    char[4] buf;  
};
```

<b>p</b>	<b>p</b>	<b>p</b>	<b>p</b>	<b>p</b>	<b>p</b>	<b>p</b>	<b>p</b>
<b>b</b>	<b>c</b>	-	-	<b>x</b>	<b>x</b>	<b>x</b>	<b>x</b>
<b>y</b>	<b>y</b>	<b>buf</b>	<b>buf</b>	<b>buf</b>	<b>buf</b>	-	-



# Structs

```
struct foo {  
    int *p;  
    char b;  
    char c;  
    int x;  
    short y;  
    char[4] buf;  
};
```

```
struct bar {  
    char a;  
    int b;  
    struct foo c;  
};
```

Now how do we represent bar?

# Structs

```
struct foo {  
    int *p;
```

```
    char b;
```

```
    char c;
```

```
    int x;
```

```
    short y;
```

```
    char[4] buf;
```

```
};
```

<b>a</b>	<b>-</b>	<b>-</b>	<b>-</b>	<b>b</b>	<b>b</b>	<b>b</b>	<b>b</b>
<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>
<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>
<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>	<b>c</b>

```
struct bar {  
    char a;  
    int b;  
    struct foo c;
```

```
};
```





# Arrays

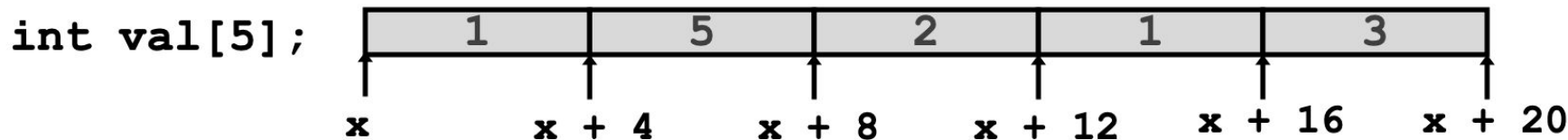
## IMPORTANT POINTS + TIPS:

- *Remember your indexing rules! They'll take you 95% of the way there.*
- Be careful about addressing (&) vs. dereferencing (\*)
- *You may be asked to look at assembly!*



# Arrays

## Good toy examples:



- A can be used as the pointer to the first array element: `A[0]`

Type

Value

`val`

`val[2]`

`*(val + 2)`

`&val[2]`

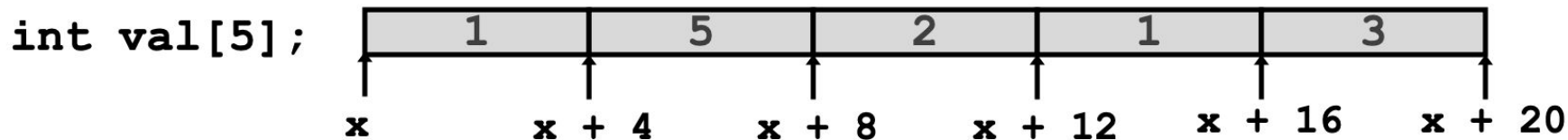
`val + 2`

`val + i`



# Arrays

## Good toy examples:



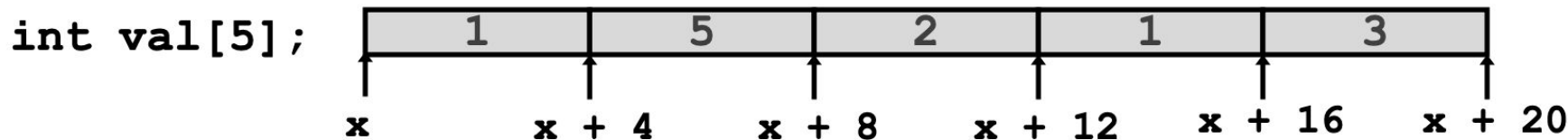
- A can be used as the pointer to the first array element: `A[0]`

	<u>Type</u>	<u>Value</u>
<code>val</code>	<code>int *</code>	<code>x</code>
<code>val[2]</code>	<code>int</code>	<code>2</code>
<code>*(val + 2)</code>	<code>int</code>	<code>2</code>
<code>&amp;val[2]</code>	<code>int *</code>	<code>x + 8</code>
<code>val + 2</code>	<code>int *</code>	<code>x + 8</code>
<code>val + i</code>	<code>int *</code>	<code>x + (4 * i)</code>



# Arrays

## Good toy examples:



- A can be used as the pointer to the first array element: `A[0]`

	<u>Type</u>	<u>Value</u>
<code>val</code>	<code>int *</code>	<code>x</code>
<code>val[2]</code>	<code>int</code>	2
<code>*(val + 2)</code>	<code>int</code>	2
<code>&amp;val[2]</code>	<code>int *</code>	<code>x + 8</code>
<code>val + 2</code>	<code>int *</code>	<code>x + 8</code>
<code>val + i</code>	<code>int *</code>	<code>x + (4 * i)</code>

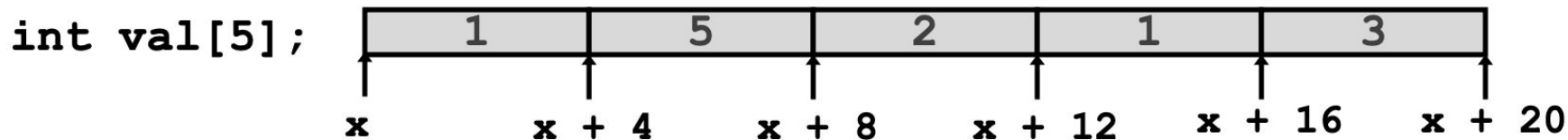
Accessing methods:

- `val[index]`
- `*(val + index)`



# Arrays

## Good toy examples:



- A can be used as the pointer to the first array element: `A[0]`

	<u>Type</u>	<u>Value</u>
<code>val</code>	<code>int *</code>	<code>x</code>
<code>val[2]</code>	<code>int</code>	<code>2</code>
<code>*(val + 2)</code>	<code>int</code>	<code>2</code>
<code>&amp;val[2]</code>	<code>int *</code>	<code>x + 8</code>
<code>val + 2</code>	<code>int *</code>	<code>x + 8</code>
<code>val + i</code>	<code>int *</code>	<code>x + (4 * i)</code>

Accessing methods:

- `val[index]`
- `*(val + index)`

Addressing methods:

- `&val[index]`
- `val + index`



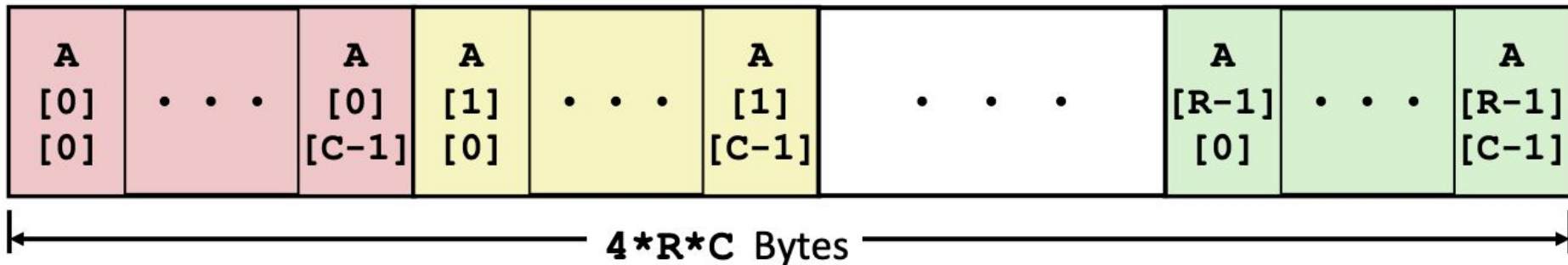


# Arrays

## Nested indexing rules

- Declared: `T A[R][C]`
- Contiguous chunk of space (think of multiple arrays lined up next to each other)

```
int A[R][C];
```



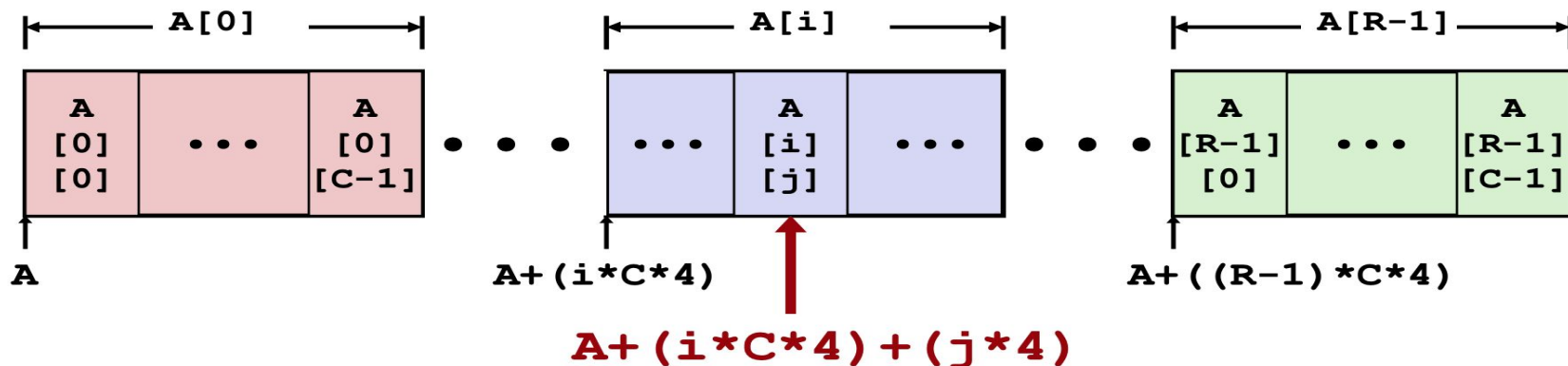


# Arrays

## Nested indexing rules:

- Arranged in ROW-MAJOR ORDER - think of row vectors
- $A[i]$  is an array of  $C$  elements (“columns”) of type  $T$

```
int A[R][C];
```





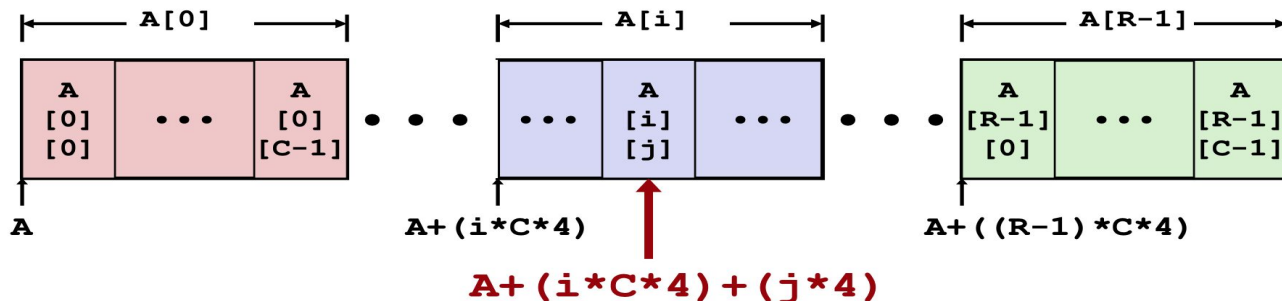
# Arrays

## Nested indexing rules:

$\mathbf{A}[i][j]$  is element of type  $T$ , which requires  $K$  bytes

$$\begin{aligned} \text{Address } \mathbf{A} + i * (\mathbf{C} * \mathbf{K}) + j * \mathbf{K} \\ = \mathbf{A} + (i * \mathbf{C} + j) * \mathbf{K} \end{aligned}$$

```
int A[R][C];
```





# Arrays

Consider accessing elements of **A**....

	<u>Compiles</u>	<u>Bad Deref?</u>	<u>Size (bytes)</u>
<code>int A1[3][5]</code>			
<code>int *A2[3][5]</code>			
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



# Arrays

Consider accessing elements of **A**....

	<u>Compiles</u>	<u>Bad Deref?</u>	<u>Size (bytes)</u>
<code>int A1[3][5]</code>	Y	N	$3 * 5 * (4) = 60$
<code>int *A2[3][5]</code>			
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



# Arrays

Consider accessing elements of **A**....

	<u>Compiles</u>	<u>Bad Deref?</u>	<u>Size (bytes)</u>
<code>int A1[3][5]</code>	Y	N	$3 * 5 * (4) = 60$
<code>int *A2[3][5]</code>	Y	N	$3 * 5 * (8) = 120$
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



# Arrays

Consider accessing elements of **A**....

	<u>Compiles</u>	<u>Bad Deref?</u>	<u>Size (bytes)</u>
<code>int A1[3][5]</code>	Y	N	$3 * 5 * (4) = 60$
<code>int *A2[3][5]</code>	Y	N	$3 * 5 * (8) = 120$
<code>int (*A3)[3][5]</code>	Y	N	$1 * 8 = 8$
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



# Arrays

Consider accessing elements of **A**....

	<u>Compiles</u>	<u>Bad Deref?</u>	<u>Size (bytes)</u>
<code>int A1[3][5]</code>	Y	N	$3 * 5 * (4) = 60$
<code>int *A2[3][5]</code>	Y	N	$3 * 5 * (8) = 120$
<code>int (*A3)[3][5]</code>	Y	N	$1 * 8 = 8$
<code>int *(A4[3][5])</code>	Y	N	$3 * 5 * (8) = 120$
<code>int (*A5[3])[5]</code>			





# Arrays

Consider accessing elements of **A**....

	<u>Compiles</u>	<u>Bad Deref?</u>	<u>Size (bytes)</u>
<code>int A1[3][5]</code>	Y	N	$3 * 5 * (4) = 60$
<code>int *A2[3][5]</code>	Y	N	$3 * 5 * (8) = 120$
<code>int (*A3)[3][5]</code>	Y	N	$1 * 8 = 8$
<code>int *(A4[3][5])</code>	Y	N	$3 * 5 * (8) = 120$
<code>int (*A5[3])[5]</code>	Y	N	$3 * 8 = 24$



# Arrays

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3][5]</code>	Y	N	60	Y	N	20	Y	N	4
<code>int *A2[3][5]</code>	Y	N	120	Y	N	40	Y	N	8
<code>int (*A3)[3][5]</code>	Y	N	8	Y	Y	60	Y	Y	20
<code>int *(A4[3][5])</code>	Y	N	120	Y	N	40	Y	N	8
<code>int (*A5[3])[5]</code>	Y	N	24	Y	N	8	Y	Y	20

ex.,     A3:        pointer to a 3x5 int array  
           \*A3:       BAD, 3x5 int array (3 \* 5 elements \* each 4 bytes = 60)  
           \*\*A3:      BAD, but means stepping inside one of 3 “rows” c



# Arrays

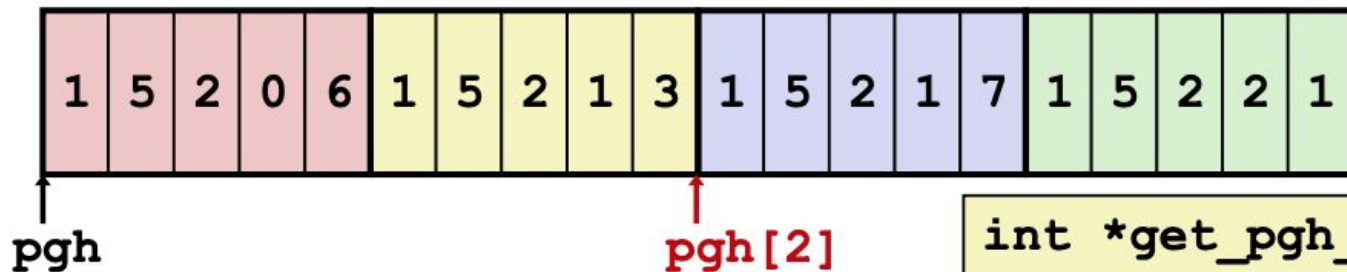
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3][5]</code>	Y	N	60	Y	N	20	Y	N	4
<code>int *A2[3][5]</code>	Y	N	120	Y	N	40	Y	N	8
<code>int (*A3)[3][5]</code>	Y	N	8	Y	Y	60	Y	Y	20
<code>int *(A4[3][5])</code>	Y	N	120	Y	N	40	Y	N	8
<code>int (*A5[3])[5]</code>	Y	N	24	Y	N	8	Y	Y	20

ex., A5: array of 3 (int \*) pointers  
 \*A5: 1 (int \*) pointer, points to an array of 5 ints  
 \*\*A5: BAD, means accessing 5 individual ints of the pointer  
 (stepping inside “row”)



# Arrays

## Sample assembly-type questions



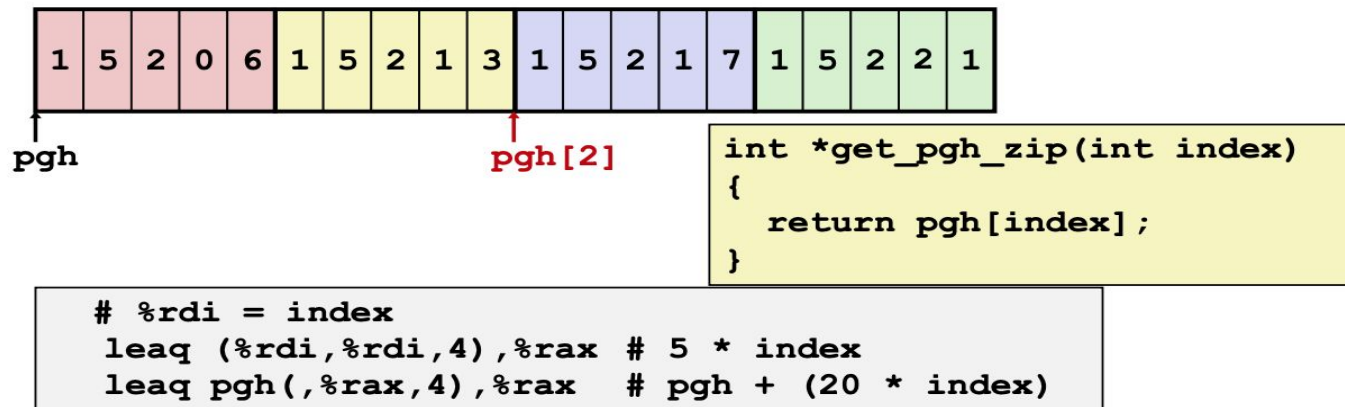
```
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq pgh(,%rax,4),%rax # pgh + (20 * index)
```



# Arrays

## Nested Array Row Access Code



### ■ Row Vector

- `pgh[index]` is array of 5 `int`'s
- Starting address `pgh+20*index`

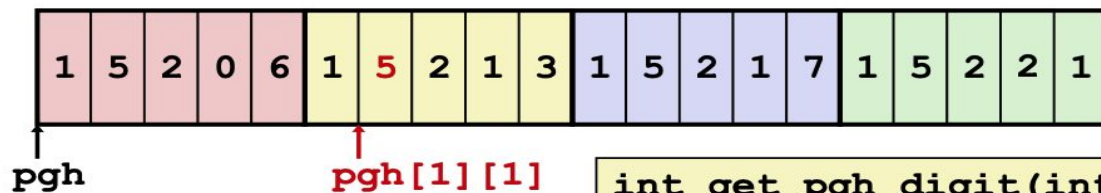
### ■ Machine Code

- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`



# Arrays

## Nested Array Element Access Code



```
int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```
leaq  (%rdi,%rdi,4), %rax    # 5*index
addl  %rax, %rsi            # 5*index+dig
movl  pgh(,%rsi,4), %eax    # M[pgh + 4*(5*index+dig)]
```

### ■ Array Elements

- `pgh[index][dig]` is `int`
- Address: `pgh + 20*index + 4*dig`  

$$= \text{pgh} + 4*(5*\text{index} + \text{dig})$$

# Stacks

# Stack

- Important things to remember:
  - Stack grows towards lower addresses
  - `%rsp` = stack pointer, always point to “top” of stack
  - Push and pop, call and ret
  - Stack frames: how they are allocated and freed
  - Which registers used for arguments? Return values?
  - Little endianness
- ALWAYS helpful to draw a stack diagram!!
- Stack questions are like Assembly questions on steroids



# Stack

- `popq D` instruction =
  - `mov (%rsp), D`
  - `add $0x8, %rsp`
- `pushq S` instruction =
  - `sub $0x8, %rsp`
  - `mov S, (%rsp)`
- `ret` instruction =
  - `pop %rip`
  - `jmp %rip`
- `callq <func>` instruction =
  - `push %rip`
  - `jmp func`

# Stack

Consider the following code:

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret

.section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "midtermexam"
```

Hints:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating `'\0'` character) to address `dst`.
- Keep endianness in mind!
- Table of hex values of characters in `"midtermexam"`

Assumptions:

- `%rsp = 0x800100` just before `caller()` calls `foo()`
- `.LC0` is at address `0x400300`

# Stack

Consider the following code:

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

Hints:

- strcpy(char \*dst, char \*src) copies the string at address src (including the terminating '\0' character) to address dst.
- Keep endianness in mind!
- Table of hex values of characters in "midtermexam"

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret

    .section      .rodata.str1.1,"aMS",@progbits,1
.LC0:= 0x400300
    .string "midtermexam"
```

→ %rsp = 0x800100

Assumptions:

- %rsp = 0x800100 just before caller() calls foo()
- .LC0 is at address 0x400300

# Stack

Question 1: What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()` ?

```

void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}

```

Hints:

- Step through the program instruction by instruction from start to end
- Draw a stack diagram!!!
- Keep track of registers too

```

foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    End call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    Start call  foo           %rsp = 0x800100
    addq   $8, %rsp
    ret

        .section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"

```

# Stack

Arrow is instruction that will execute NEXT

Question 1: What is the hex value of `%rsp` just before `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```


```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x800100</code>
<code>%rdi</code>	<code>.LC0</code>
<code>%rsi</code>	<code>0x15213</code>

<code>0x800100</code>	
<code>0x8000f8</code>	
<code>0x8000f0</code>	
<code>0x8000e8</code>	
<code>0x8000e0</code>	
<code>0x8000d8</code>	
<code>0x8000d0</code>	
<code>0x8000c8</code>	
<code>0x8000c0</code>	
<code>0x8000b8</code>	

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp    .L1
```

```
.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    End call  strcpy
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
     call  foo
    addq   $8, %rsp
    ret

    .section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
    .string "midtermexam"
```

`%rsp = 0x800100`

# Stack


Question 1: What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x8000f8</code>
<code>%rdi</code>	<code>.LC0</code>
<code>%rsi</code>	<code>0x15213</code>

<code>0x800100</code>	?
<b><code>0x8000f8</code></b>	<b>ret address for <code>foo()</code></b>
<code>0x8000f0</code>	
<code>0x8000e8</code>	
<code>0x8000e0</code>	
<code>0x8000d8</code>	
<code>0x8000d0</code>	
<code>0x8000c8</code>	
<code>0x8000c0</code>	
<code>0x8000b8</code>	

```
foo:
     subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call   foo
    jmp    .L1
```

```
.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    End call   strcpy
```

```
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call   foo
    addq   $8, %rsp
    ret
```

```
.section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
.string "midtermexam"
```

# Stack

Hint: \$24 in decimal = 0x18


Question 1: What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	0x8000e0
<code>%rdi</code>	.LC0
<code>%rsi</code>	0x15213

0x800100	?
0x8000f8	ret address for <code>foo()</code>
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	
0x8000d0	
0x8000c8	
0x8000c0	
0x8000b8	

```
foo:
    subq    $24, %rsp
     cml     $0xdeadbeef, %esi
    je      .L2
    movl   $0xdeadbeef, %esi
    call   foo
    jmp    .L1
```

```
.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    End call   strcpy
```

```
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call   foo
    addq   $8, %rsp
    ret
```

```
.section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
.string "midtermexam"
```

# Stack


Question 1: What is the hex value of `%rsp` just before `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x8000e0</code>
<code>%rdi</code>	<code>.LC0</code>
<code>%rsi</code>	<code>0xdeadbeef</code>

<code>0x800100</code>	?
<code>0x8000f8</code>	ret address for <code>foo()</code>
<code>0x8000f0</code>	?
<code>0x8000e8</code>	?
<code>0x8000e0</code>	?
<code>0x8000d8</code>	
<code>0x8000d0</code>	
<code>0x8000c8</code>	
<code>0x8000c0</code>	
<code>0x8000b8</code>	

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
     call    foo
    jmp    .L1
```

```
.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    End call    strcpy
```

```
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call   foo
    addq   $8, %rsp
    ret
```

```
.section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
.string "midtermexam"
```



# Stack


Question 1: What is the hex value of `%rsp` just before `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x8000d8</code>
<code>%rdi</code>	<code>.LC0</code>
<code>%rsi</code>	<code>0xdeadbeef</code>

<code>0x800100</code>	?
<code>0x8000f8</code>	ret address for <code>foo()</code>
<code>0x8000f0</code>	?
<code>0x8000e8</code>	?
<code>0x8000e0</code>	?
<code>0x8000d8</code>	ret address for <code>foo()</code>
<code>0x8000d0</code>	
<code>0x8000c8</code>	
<code>0x8000c0</code>	
<code>0x8000b8</code>	

```
foo:
     subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call   foo
    jmp    .L1
```

```
.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    End call   strcpy
```

```
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call   foo
    addq   $8, %rsp
    ret

        .section      .rodata.str1.1, "aMS", @progbits, 1
.LC0: = 0x400300
        .string "midtermexam"
```

# Stack


Question 1: What is the hex value of `%rsp` just before `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>.LC0</code>
<code>%rsi</code>	<code>0xdeadbeef</code>

<code>0x800100</code>	?
<code>0x8000f8</code>	ret address for <code>foo()</code>
<code>0x8000f0</code>	?
<code>0x8000e8</code>	?
<code>0x8000e0</code>	?
<code>0x8000d8</code>	ret address for <code>foo()</code>
<code>0x8000d0</code>	?
<code>0x8000c8</code>	?
<code>0x8000c0</code>	?
<code>0x8000b8</code>	?

```
foo:
    subq    $24, %rsp
     cmpq    $0xdeadbeef, %esi
    je      .L2
    movq    $0xdeadbeef, %esi
    call   foo
    jmp     .L1
```

```
.L2:
    movq    %rdi, %rsi
    movq    %rsp, %rdi
    End call   strcpy
.L1:
    addq    $24, %rsp
    ret
```

```
caller:
    subq    $8, %rsp
    movl    $86547, %esi
    movl    $.LC0, %edi
    call   foo
    addq    $8, %rsp
    ret

        .section      .rodata.str1.1, "aMS", @progbits, 1
.LC0: = 0x400300
        .string "midtermexam"
```

# Stack

Question 1: What is the hex value of `%rsp` just before `strcpy()` is called for the first time in `foo()` ?


```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>.LC0</code>
<code>%rsi</code>	<code>0xdeadbeef</code>

<code>0x800100</code>	?
<code>0x8000f8</code>	ret address for <code>foo()</code>
<code>0x8000f0</code>	?
<code>0x8000e8</code>	?
<code>0x8000e0</code>	?
<code>0x8000d8</code>	ret address for <code>foo()</code>
<code>0x8000d0</code>	?
<code>0x8000c8</code>	?
<code>0x8000c0</code>	?
<code>0x8000b8</code>	?

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp    .L1
```

```
.L2:
     movq    %rsi, %rdi
    movq    %rsp, %rdi
    End call    strcpy
.L1:
    addq    $24, %rsp
    ret
```

```
caller:
    subq    $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq    $8, %rsp
    ret

        .section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

# Stack

Question 1: What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()` ?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

**Answer!**

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>0x8000c0</code>
<code>%rsi</code>	<code>.LCO</code>

0x800100	?
0x8000f8	ret address for <code>foo()</code>
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	ret address for <code>foo()</code>
0x8000d0	?
0x8000c8	?
<b>0x8000c0</b>	?
0x8000b8	

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rsi, %rdi
    movq   %rdi, %rsp
    call  strcpy
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LCO, %edi
    call  foo
    addq   $8, %rsp
    ret

.section .rodata.str1.1,"aMS",@progbits,1
.LCO: = 0x400300
.string "midtermexam"
```

→ End

# Stack

**Question 2:** What is the hex value of `buf[0]` when `strcpy()` returns?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>0x8000c0</code>
<code>%rsi</code>	<code>.LC0</code>

<code>0x800100</code>	?
<code>0x8000f8</code>	ret address for <code>foo()</code>
<code>0x8000f0</code>	?
<code>0x8000e8</code>	?
<code>0x8000e0</code>	?
<code>0x8000d8</code>	ret address for <code>foo()</code>
<code>0x8000d0</code>	?
<code>0x8000c8</code>	?
<code>0x8000c0</code>	?
<code>0x8000b8</code>	?

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp    .L1
```

```
.L2:
    movq   %rsi, %rdi
    movq   %rsp, %rdi
    call  strcpy
```

```
.L1:
    addq   $24, %rsp
    ret
```

```
caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret
```

```
.section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
.string "midtermexam"
```

# Stack

**Question 2:** What is the hex value of `buf[0]` when `strcpy()` returns?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret

        .section      .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>0x8000c0</code>
<code>%rsi</code>	<code>\$.LC0</code>

0x800100	?								
0x8000f8	ret address for foo()								
0x8000f0	?								
0x8000e8	?								
0x8000e0	?								
0x8000d8	ret address for foo()								
0x8000d0	?								
0x8000c8									
0x8000c0									
0x8000b8	c7						c2	c1	c0

# Stack

**Question 2:** What is the hex value of `buf[0]` when `strcpy()` returns?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rsi, %rdi
    movq   %rsp, %rdi
    call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret

        .section      .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>0x8000c0</code>
<code>%rsi</code>	<code>.LC0</code>

0x800100	?							
0x8000f8	ret address for foo()							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for foo()							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8	c7		c2			c1	c0	

# Stack

Question 2: What is the hex value of `buf[0]` when `strcpy()` returns?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rsi, %rdi
    movq   %rsp, %rdi
    call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret

        .section      .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>0x8000c0</code>
<code>%rsi</code>	<code>.LC0</code>

0x800100	?							
0x8000f8	ret address for foo()							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for foo()							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8	c3 buf[0] c0							



# Stack

buf[0] = 

't'	'd'	'i'	'm'
-----	-----	-----	-----

= 

74	64	69	6d
----	----	----	----

(as int) = **0x7464696d**

Char	Hex	Char	Hex
a	61	m	6d
d	64	r	72
e	65	t	74
i	69	x	78

0x800100	?							
0x8000f8	ret address for foo()							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for foo()							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
<b>0x8000c0</b>	'e'	'm'	'r'	'e'	<b>'t'</b>	<b>'d'</b>	<b>'i'</b>	<b>'m'</b>
0x8000b8	<b>buf[0]</b>							

# Stack

**Question 3:** What is the hex value of `buf[1]` when `strcpy()` returns?

```

void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}

```

```

foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    call  foo
    jmp   .L1

.L2:
    movq   %rsi, %rdi
    movq   %rdi, %rsp
    call  strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call  foo
    addq   $8, %rsp
    ret

        .section      .rodata.s
.LC0: = 0x400300
        .string "midtermexam"

```

<code>%rsp</code>	<code>0x8000c0</code>
<code>%rdi</code>	<code>0x8000c0</code>
<code>%rsi</code>	<code>.LC0</code>

0x800100	?							
0x8000f8	ret address for foo()							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for foo()							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8	buf[1]				buf[0]			

# Stack

buf[1] = 

'e'	'm'	'r'	'e'
-----	-----	-----	-----

= 

65	6d	72	65
----	----	----	----


(as int) = **0x656d7265**

Char	Hex	Char	Hex
a	61	m	6d
d	64	r	72
e	65	t	74
i	69	x	78

0x800100	?							
0x8000f8	ret address for foo()							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for foo()							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
<b>0x8000c0</b>	<b>'e'</b>	<b>'m'</b>	<b>'r'</b>	<b>'e'</b>	't'	'd'	'i'	'm'
0x8000b8	<b>buf[1]</b>							


# Stack

**Question 4:** What is the hex value of `%rdi` at the point where `foo()` is called recursively in the successful arm of the `if` statement?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
         foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

This is before the recursive call to `foo()`

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
     call    foo
    jmp    .L1

.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    call   strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi
    call   foo
    addq   $8, %rsp
    ret

        .section      .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

# Stack

**Question 4:** What is the hex value of `%rdi` at the point where `foo()` is called recursively in the successful arm of the `if` statement?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        → foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

- This is before the recursive call to `foo()`
- Going backwards, `%rdi` was loaded in `caller()`
- `%rdi = $.LC0 = 0x400300` (based on hint)

```
foo:
    subq    $24, %rsp
    cmpl   $0xdeadbeef, %esi
    je     .L2
    movl   $0xdeadbeef, %esi
    → call  foo
    jmp    .L1

.L2:
    movq   %rdi, %rsi
    movq   %rsp, %rdi
    call   strcpy

.L1:
    addq   $24, %rsp
    ret

caller:
    subq   $8, %rsp
    movl   $86547, %esi
    movl   $.LC0, %edi ← loaded %rdi
    call   foo
    addq   $8, %rsp
    ret

.section .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
.string "midtermexam"
```

# Stack

**Question 5:** What part(s) of the stack will be corrupted by invoking `caller()`?  
Check all that apply.

- return address from `foo()` to `caller()`
- return address from the recursive call to `foo()`
- `strcpy()`'s return address
- there will be no corruption

# Stack

**Question 5:** What part(s) of the stack will be corrupted by invoking `caller()`?  
Check all that apply.

- return address from `foo()` to `caller()`
- return address from the recursive call to `foo()`
- `strcpy()`'s return address
- there will be no corruption

The `strcpy` didn't overwrite any return addresses, so there was no corruption!

0x800100	?							
0x8000f8	ret address for <code>foo()</code>							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for <code>foo()</code>							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8								

# Caches



# Cache Concepts

- Key Points

- Direct mapped vs. n-way associative vs. fully associative
- Tag/Set/Block offset bits, how do they map depending on cache size?
- LRU policies

(Spend time)

# Bonus! Another Cache problem (will update)

- Consider you have the following cache:
  - 64-byte capacity
  - Directly mapped
  - You have an 8-bit address space

# Bonus! (will update)

A. How many tag bits are there in the cache?

■ Do we know how many set bits there are? What about offset bits?  
 $2^6 = 64$

■ If we have a 64-byte **direct-mapped** cache, we know the number of  $s + b$  bits there are total!

■ Then  $t + s + b = 8 \rightarrow t = 8 - (s + b)$

■ Thus, we have 2 tag bits!

# Problem 1: Cache

- A. Assume you have a cache of the following structure:
  - a. 32-byte blocks
  - b. 2 sets
  - c. Direct-mapped
  - d. 8-bit address space
  - e. The cache is cold prior to access
  
- B. What does the address decomposition look like?

0 0 0 0 0 0 0 0

# Problem 1: Cache

- A. Assume you have a cache of the following structure:
  - a. 32-byte blocks
  - b. 2 sets
  - c. Direct-mapped
  - d. 8-bit address space
  - e. The cache is cold prior to access
  
- B. What does the address decomposition look like?

0 0 0 0 0 0 0 0

# Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0x56				
0x6D				
0x49				
0x3A				

# Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110				
0110 1101				
0100 1001				
0011 1010				

# Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101				
0100 1001				
0011 1010				



# Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101	1	01	M	N
0100 1001				
0011 1010				

# Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101	1	01	M	N
0100 1001	0	01	H	N
0011 1010				

# Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101	1	01	M	N
0100 1001	0	01	H	N
0011 1010	1	00	M	Y

# Problem 1: Cache

- A. Assume you have a cache of the following structure:
  - a. 2-way associative
  - b. 4 sets, 64-byte blocks
  
- B. What does the address decomposition look like?

... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

# Problem 1: Cache

- A. Assume you have a cache of the following structure:
  - a. 2-way associative
  - b. 4 sets, 64-byte blocks
  
- B. What does the address decomposition look like?

... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

# Problem 1: Cache

**B. Assume A and B are 128 ints and cache-aligned.**

- What is the miss rate of pass 1?**
- What is the miss rate of pass 2?**

```
int get prod and copy(int *A, int *B) {  
    int length = 64;  
    int prod = 1;  
    // pass 1  
    for (int i = 0; i < length; i+=4) {  
        prod*=A[i];  
    }  
    // pass 2  
    for (int j = length-1; j > 0; j-=4) {  
        A[j] = B[j];  
    }  
    return prod;  
}
```

# Problem 1: Cache

**B. Pass 1: Only going through 64 ints with step size 4. Each miss loads 16 ints into a cache line, giving us 3 more hits before loading into a new line.**

```
int get prod and copy(int *A, int *B) {  
    int length = 64;  
    int prod = 1;  
    // pass 1  
    for (int i = 0; i < length; i+=4) {  
        prod*=A[i];  
    }  
    // pass 2  
    for (int j = length-1; j > 0; j-=4) {  
        A[j] = B[j];  
    }  
    return prod;  
}
```

# Problem 1: Cache

## B. Pass 1: 25% miss

```
int get prod and copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```



# Problem 1: Cache

**B. Pass 2: Our cache is the same size as our working set! Due to cache alignment, we won't evict anything from A, but still get a 1:3 miss:hit ratio for B.**

```
int get prod and copy(int *A, int *B) {  
    int length = 64;  
    int prod = 1;  
    // pass 1  
    for (int i = 0; i < length; i+=4) {  
        prod*=A[i];  
    }  
    // pass 2  
    for (int j = length-1; j > 0; j-=4) {  
        A[j] = B[j];  
    }  
    return prod;  
}
```

# Problem 1: Cache

**B. Pass 2: For every 4 loop iterations, we get all hits for accessing A and 1 miss for accessing B, which gives us  $\frac{1}{8}$  miss.**

```
int get prod and copy(int *A, int *B) {  
    int length = 64;  
    int prod = 1;  
    // pass 1  
    for (int i = 0; i < length; i+=4) {  
        prod*=A[i];  
    }  
    // pass 2  
    for (int j = length-1; j > 0; j-=4) {  
        A[j] = B[j];  
    }  
    return prod;  
}
```

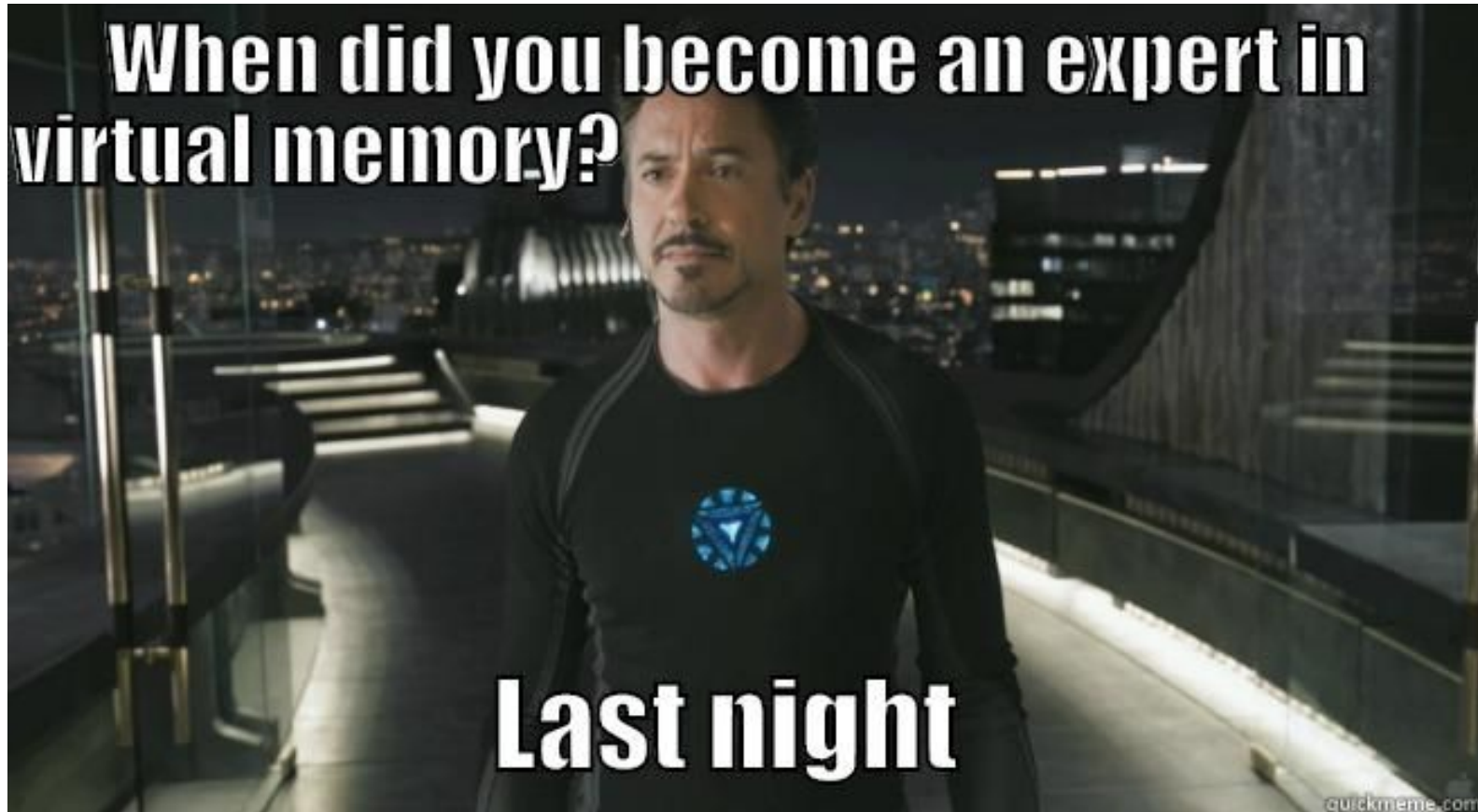
# Problem 1: Cache

## B. Pass 2: 12.5% miss

```
int get prod and copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```

# Virtual Memory

# Virtual Memory



# Virtual Memory

Virtual Address - 18 Bits

Physical Address - 12 Bits

Page Size - 512 Bytes

TLB is 8-way set associative

Cache is 2-way set associative

Page Table						TLB			
VPN	PPN	Valid	VPN	PPN	Valid	Index	Tag	PPN	Valid
000	7	0	010	1	0	0	55	6	0
001	5	0	011	3	0		48	F	1
002	1	1	012	3	0		00	A	0
003	5	0	013	0	0		32	9	1
004	0	0	014	6	1		6A	3	1
005	5	0	015	5	0		56	1	0
006	2	0	016	7	0		60	4	1
007	4	1	017	2	1		78	9	0
008	7	0	018	0	0	1	71	5	1
009	2	0	019	2	0		31	A	1
00A	3	0	01A	1	0		53	F	0
00B	0	0	01B	3	0		87	8	0
00C	0	0	01C	2	0		51	D	0
00D	3	0	01D	7	0		39	E	1
00E	4	0	01E	5	1		43	B	0
00F	7	1	01F	0	0		73	2	1

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

[Final S-02 \(#5\)](#)

[Lecture 17: VM - Systems](#)











# Virtual Memory

Label the following:

- (A) *VPO*: Virtual Page Offset
- (B) *VPN*: Virtual Page Number
- (C) *TLBI*: TLB Index - Location in the TLB Cache  
2 Indices  $\rightarrow$  1 Bit

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

TLBI

# Virtual Memory

Label the following:

- (A) *VPO*: Virtual Page Offset
- (B) *VPN*: Virtual Page Number
- (C) *TLBI*: TLB Index
- (D) *TLBT*: TLB Tag - Everything Else

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

TLBT

TLBI











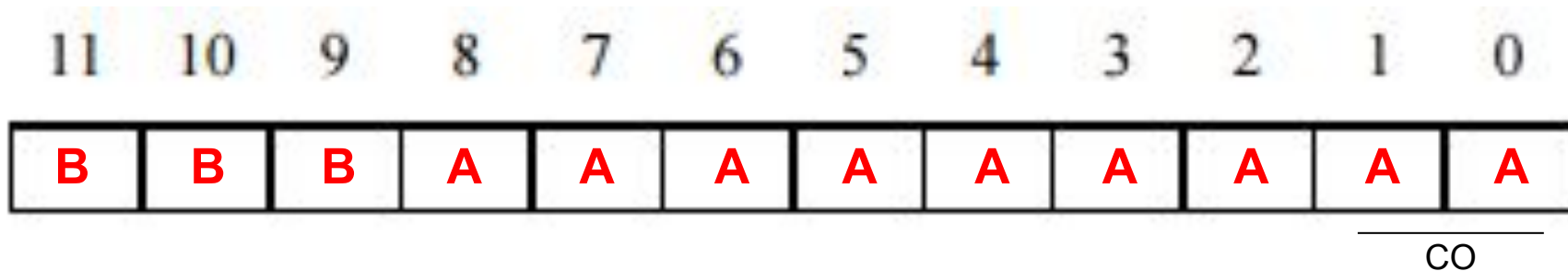


# Virtual Memory

Label the following:

- (A) *PPO*: Physical Page Offset - Same as VPO
- (B) *PPN*: Physical Page Number - Everything Else
- (C) *CO*: Cache Offset - Offset in Block

4 Byte Blocks  $\rightarrow$  2 Bits



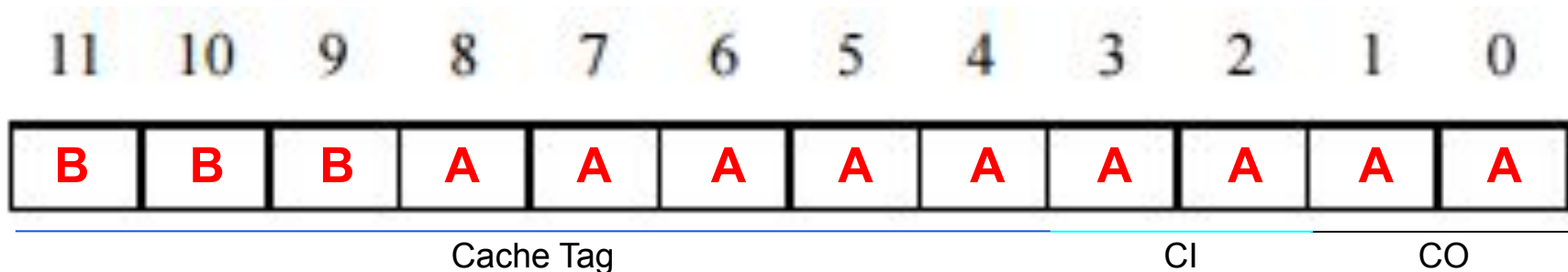




# Virtual Memory

Label the following:

- (A) *PPO*: Physical Page Offset - Same as VPO
- (B) *PPN*: Physical Page Number - Everything Else
- (C) *CO*: Cache Offset - Offset in Block
- (D) *CI*: Cache Index
- (E) *CT*: Cache Tag - Everything Else





# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation

1 = 0001    A = 1010    9 = 1001    F = 1111    4 = 0100

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0

# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0x??      TLBI: 0x??      TLBT: 0x??  
TLB Hit: Y/N?    Page Fault: Y/N?    PPN: 0x??

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0



# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4      TLBI: 0x??      TLBT: 0x??  
TLB Hit: Y/N?    Page Fault: Y/N?    PPN: 0x??

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0

---

# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4      TLBI: 0x00      TLBT: 0x??  
 TLB Hit: Y/N?   Page Fault: Y/N?   PPN: 0x??

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0

# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4

TLBI: 0x00

TLBT: 0x6A

TLB Hit: Y/N?

Page Fault: Y/N?

PPN: 0x??

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
1	71	5	1
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4      TLBI: 0x00      TLBT: 0x6A  
 TLB Hit: Y!   Page Fault: Y/N?   PPN: 0x??

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
1	56	1	0
	60	4	1
	78	9	0
	71	5	1
1	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Now to the actual question!

Q) Translate the following address: **0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4      TLBI: 0x00      TLBT: 0x6A  
 TLB Hit: Y!   Page Fault: N!   PPN: 0x??

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
	1	71	5
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4      TLBI: 0x00      TLBT: 0x6A  
 TLB Hit: Y!   Page Fault: N!   PPN: 0x3

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
	1	71	5
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

17   16   15   14   13   12   11   10   9   8   7   6   5   4   3   2   1   0

0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

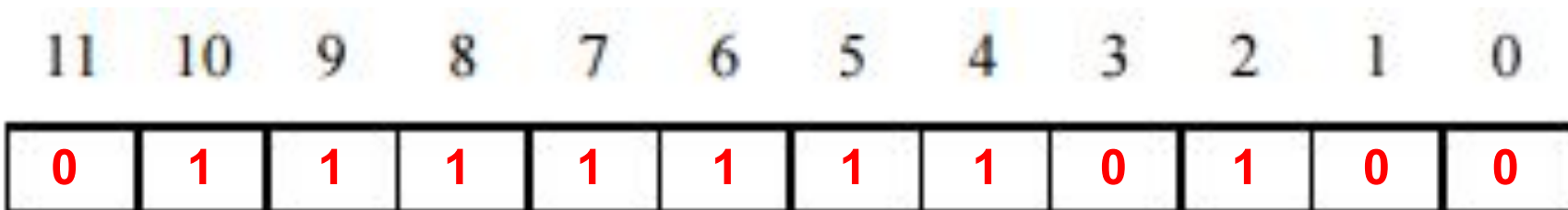


# Virtual Memory

Now to the actual question!

**Q) Translate the following address: 0x1A9F4**

1. Write down bit representation
2. Extract Information
3. Put it all together: PPN: 0x3, PPO = VPO = 0x1F4





# Virtual Memory

**Q) What is the value of the address?**

CO: 0x??    CI: 0x??    CT: 0x??    Cache Hit: Y/N?    Value:0x??

11   10   9   8   7   6   5   4   3   2   1   0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Q) **What is the value of the address?**

1. Extract more information

CO: 0x00    CI: 0x??    CT: 0x??    Cache Hit: Y/N?    Value:0x??

11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	1	0	0

---

# Virtual Memory

**Q) What is the value of the address?**

1. Extract more information

CO: 0x00    CI: 0x01    CT: 0x??    Cache Hit: Y/N?    Value:0x??

11   10   9   8   7   6   5   4   3   2   1   0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Q) What is the value of the address?

1. Extract more information
2. Go to Cache Table

CO: 0x00    CI: 0x01    CT: 0x7F    Cache Hit: Y/N?    Value:0x??

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

11    10    9    8    7    6    5    4    3    2    1    0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Q) What is the value of the address?

1. Extract more information
2. Go to Cache Table

CO: 0x00    CI: 0x01    CT: 0x7F    Cache Hit: Y    Value:0x??

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

11    10    9    8    7    6    5    4    3    2    1    0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

# Virtual Memory

Q) What is the value of the address?

1. Extract more information
2. Go to Cache Table

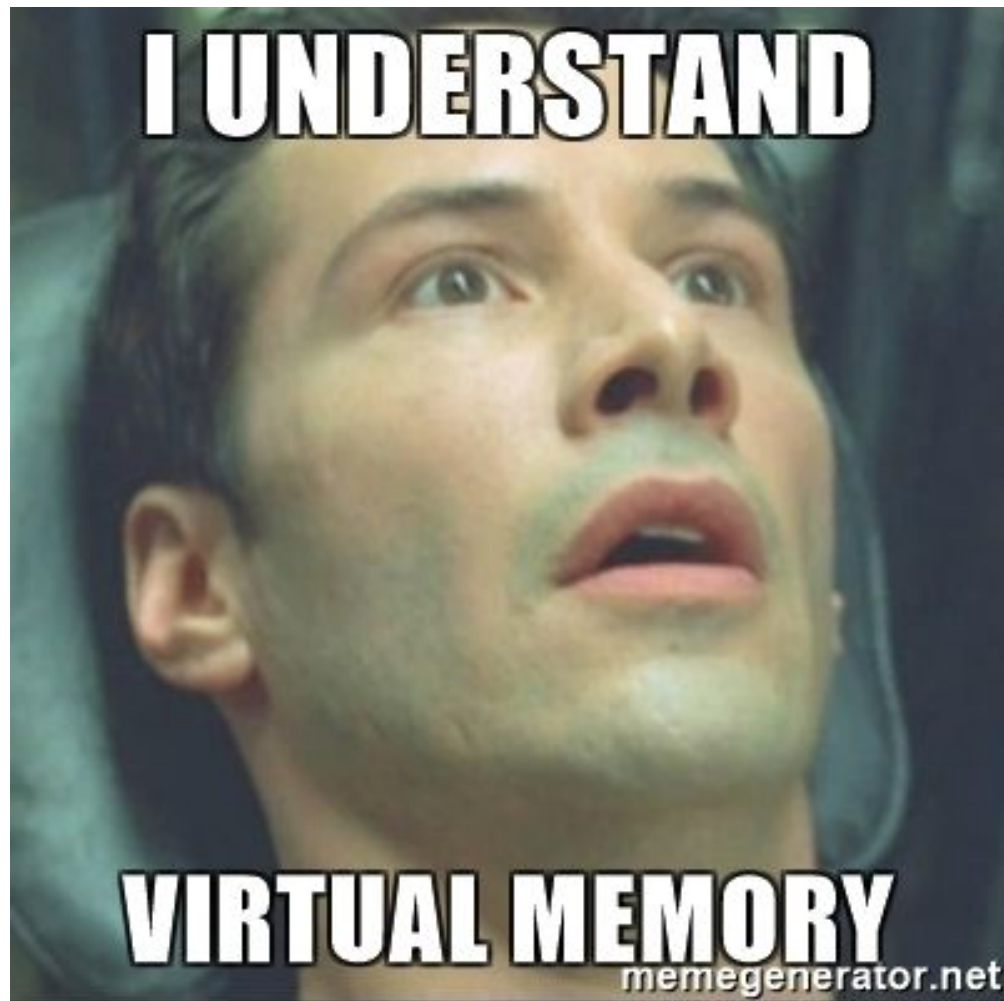
CO: 0x00    CI: 0x01    CT: 0x7F    Cache Hit: Y    Value:0xFF

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

11    10    9    8    7    6    5    4    3    2    1    0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

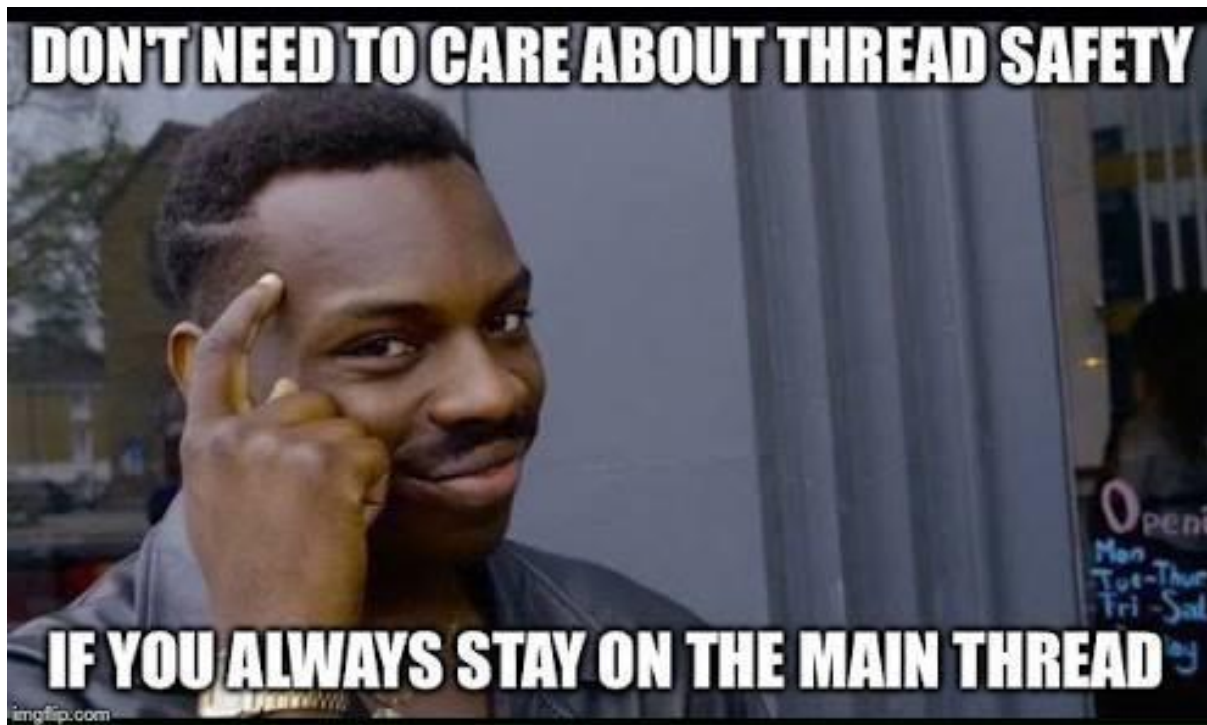
# Virtual Memory



# Threads



# Threads



# Threads

Given this code, what variables do you think are shared?

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 2
int balance = 10;

int main() {
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_create(&tid[0], NULL, threadA, (void*)0);
    pthread_create(&tid[1], NULL, threadB, (void*)0);
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], NULL);
    }
    printf("balance: %d\n", balance); // What is balance?
    return 0;
}

void *threadA(void *vargp) {
    long instance = (long)vargp;
    static int cnt = 0;
    deposit(4);
    withdraw(11);
    return NULL;
}

void *threadB() {
    withdraw(6);
    deposit(3);
    withdraw(7);
    return NULL;
}
```

## Threads (Contd.)

Which variables can be shared by multiple threads simultaneously in this program?

- (A) i
- (B) balance
- (C) instance
- (D) cnt
- (E) None of the above

## Threads (Contd.)

Which variables can be shared by multiple threads simultaneously in this program?

- (A) i
- (B) balance
- (C) instance
- (D) cnt
- (E) None of the above

Answer: B

## Threads (Contd.)

- (A) `i` is a local variable so it isn't shared.
- (A) `balance` is a global variable so it's shared.
- (A) `instance` is local to `threadA()` so it isn't shared.
- (A) `cnt` is a static variable, so it retains its value even outside the scope in which it was defined, so it isn't shared.

# Threads (Contd.)

Given the withdraw() and deposit() functions, what are the possible outputs? (balance = 10 initially)

```
int withdraw(int amt) {
    if (balance >= amt) {
        balance = balance - amt;
        return 0;
    } else {
        return -1;
    }
}

int deposit(int amt) {
    balance = balance + amt;
    sleep(2);
    return 0;
}
```

```
void *threadA(void *vargp) {
    long instance = (long)vargp;
    static int cnt = 0;
    deposit(4);
    withdraw(11);
    return NULL;
}

void *threadB() {
    withdraw(6);
    deposit(3);
    withdraw(7);
    return NULL;
}
```

# Threads (Contd.)

What can be the value of balance?

- (A) balance: 0
- (B) balance: -3
- (C) balance: 14
- (D) balance: 6
- (E) balance: 17
- (F) balance: 4

# Threads (Contd.)

What can be printed at the indicated line?

- (A) balance: 0
- (B) balance: -3
- (C) balance: 14
- (D) balance: 6
- (E) balance: 17
- (F) balance: 4

Answer: ABDF



## Threads (Contd.)

The following is one interleaving that leads to output 0:

- Thread A executes `deposit(4)`, balance = 14
- Thread B executes `withdraw(6)`, balance = 8
- Thread B executes `deposit(3)`, balance = 11
- Thread A executes `withdraw(11)`, balance = 0
- Thread B executes `withdraw(7)`, balance = 0

## Threads (Contd.)

The following is one interleaving that leads to output -3:

- Thread A executes `deposit(4)`, `balance = 14`
- Thread A starts to execute `withdraw(11)` and enters the if condition
- Thread B executes `withdraw(6)`, `balance = 8`
- Thread A computes RHS for `withdraw(11) = -3`
- Thread B executes `deposit(3)`, `balance = 11`
- Thread A completes `withdraw(11)`, `balance = -3`
- Thread B executes `withdraw(7)`, `balance = -3`

## Threads (Contd.)

The following is one interleaving that leads to output 6:

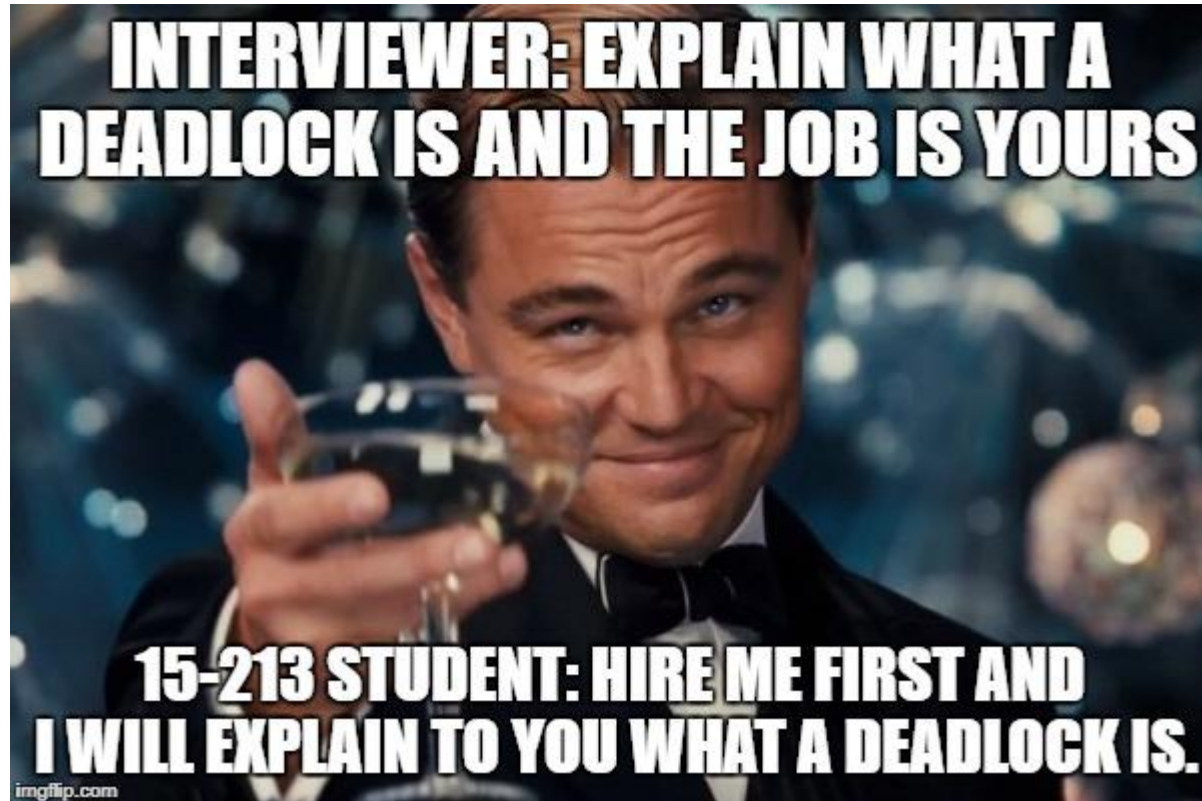
- Thread A executes `deposit(4)`, balance = 14
- Thread A executes `withdraw(11)`, balance = 3
- Thread B executes `withdraw(6)`, balance = 3
- Thread B executes `deposit(3)`, balance = 6
- Thread B executes `withdraw(7)`, balance = 6

## Threads (Contd.)

The following is one interleaving that leads to output 4:

- Thread B executes `withdraw(6)`, balance = 4
- Thread A executes `deposit(4)`, balance = 8
- Thread A executes `withdraw(11)`, balance = 8
- Thread B executes `deposit(3)`, balance = 11
- Thread B executes `withdraw(7)`, balance = 4

# Synchronization



# Thread Synchronization

How many potential deadlock situations are present?

```
void *thread1(void *vargp) {
    V(&add_sem);
    V(&rem_sem);

    remove();

    P(&add_sem);
    P(&rem_sem);

    add();

    V(&add_sem);
    V(&rem_sem);

    remove();
    add();
}

sem_t add_sem;
sem_t rem_sem;

void *thread2(void *vargp) {
    P(&rem_sem);
    P(&add_sem);

    add();
    remove();
}

int main() {
    pthread_t tid1, tid2;

    sem_init(&add_sem,0,0);
    sem_init(&rem_sem,0,0);

    pthread_create(&tid1, NULL, thread1, NULL);
    pthread_create(&tid2, NULL, thread2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

# Thread Synchronization (Contd.)

Situation 1:

tid1 executes  $V(&add\_sem)$  and  $V(&rem\_sem)$ . Then, tid2 executes  $P(&rem\_sem)$  and  $P(&add\_sem)$ . In this situation, tid1 can never execute  $P(&add\_sem)$  since the value of  $add\_sem = 0$ . As a result, this is a deadlock, since after the execution of thread 2, thread 1 can't resume. Thus, there's a deadlock.

# Thread Synchronization (Contd.)

Situation 2:

tid1 executes  $V(&add\_sem)$  and  $V(&rem\_sem)$ . Then, tid2 executes  $P(&rem\_sem)$ . Next, tid1 executes  $P(&add\_sem)$ . Thread 2 wants to execute  $P(&add\_sem)$  but it can't since  $add\_sem$  has value 0. Thread 1 wants to execute  $P(&rem\_sem)$  but it can't since  $rem\_sem$  has value 0. Thus, there's a deadlock.



# Thread Synchronization (Contd.)

For lengths 0-6, indicate the number of outcomes of that length that can be produced.

```

sem_t add_sem;
sem_t rem_sem;

void add() {
    printf("A");
}

void remove() {
    printf("R");
}

void *thread2(void *vargp)
    P(&rem_sem);
    P(&add_sem);

    add();
    remove();
}

void *thread1(void *vargp) {
    V(&add_sem);
    V(&rem_sem);

    remove();

    P(&add_sem);
    P(&rem_sem);

    add();

    V(&add_sem);
    V(&rem_sem);

    remove();
    add();
}

int main() {
    pthread_t tid1, tid2;

    sem_init(&add_sem,0,0);
    sem_init(&rem_sem,0,0);

    pthread_create(&tid1, NULL, thread1, NULL);
    pthread_create(&tid2, NULL, thread2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}

```

## Thread Synchronization (Contd.)

Response length 0: None

This is because at least 'R' must get printed due to the call to `remove()` in `thread1()`. Even if there is a deadlock, at least that statement gets executed by `tid1` before any sort of deadlock from the above situations.

# Thread Synchronization (Contd.)

Response length 1: 1 (R)

In the deadlock scenario 2, where thread 1 executes `P(&add_sem)` and thread 2 executes `P(&rem_sem)`, neither of the threads can proceed past that. Thus, no print statements are executed in either thread after that point. The only print statement that gets executed is due to the call to `remove()` before the calls to `P()` in thread 1.

# Thread Synchronization (Contd.)

Response length 2: None

We noticed that 'R' due to the call to `remove()` in `thread1()` gets printed no matter what. From the code, we notice that it's not possible for only one other print statement to get executed.

# Thread Synchronization (Contd.)

Response length 3: 2 (RAR, ARR)

This happens due to deadlock scenario 1 above, where thread2() executes completely but thread1() can't execute P(&add\_sem) and the statements after that.

- RAR: Thread 1 executes remove(), followed by thread 2 executing add() and remove().
- ARR: Thread 2 executes add(), followed by any ordering of the 2 calls to remove() by threads 1 and 2.

## Thread Synchronization (Contd.)

Response length 4: None

For any length greater than 3, it means that there was no deadlock, since thread 2 could run to completion and thread 1 could get past the calls to  $P()$ , which means it would run to completion as well. Thus, no responses of length greater than 3 and less than 6 are possible.

## Thread Synchronization (Contd.)

Response length 5: None

For any length greater than 3, it means that there was no deadlock, since thread 2 could run to completion and thread 1 could get past the calls to  $P()$ , which means it would run to completion as well. Thus, no responses of length greater than 3 and less than 6 are possible.

## Thread Synchronization (Contd.)

Response length 6: 4

(RARAAR, RARARA, RAARRA, RAARAR)

Since there are no deadlocks, it means that the initial calls to V() and P() get executed by thread 1. Thus, 'R' and 'A' definitely get printed. After this, the calls to V() get executed by thread 1 and then, thread 2 can execute its calls to P(). After this, based on the interleavings between the threads, there are 4 possible outputs.



## Thread Synchronization (Contd.)

- RARAAR: Thread 1 executes `remove()`, threads 1 and 2 execute the `add()` statements in any order, and then thread 2 executes `remove()`.
- RARARA: Thread 1 executes `remove()`, thread 2 executes `add()` and `remove()`, then thread 1 executes `add()`.

## Thread Synchronization (Contd.)

- RAARRA: Thread 2 executes `add()`, threads 1 and 2 execute the `remove()` statements in any order, and then thread 1 executes `add()`.
- RAARAR: Thread 2 executes `add()`, thread 1 executes `remove()` and `add()`, then thread 2 executes `remove()`.

# Processes

1. Logical control flow
2. Private address space

## Important system calls

1. Fork
2. Execve
3. Wait
4. Waitpid



# Processes

Draw a Process Graph!!!

(it does not have to be like mine)

# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What is printed?

Assume printf is atomic,  
and all system calls  
succeed.

# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

How many processes?

# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

How many processes?

Parent: forks child

Parent and child: each fork  
another child

Total: 4 processes

# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

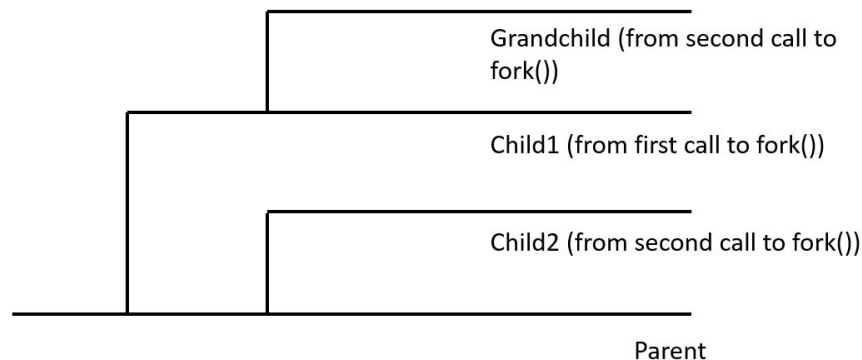
What does the process diagram look like?



# Processes

```
int main() {  
    int count = 1;  
    int pid1 = fork();  
    int pid2 = fork();  
  
    if(pid1 == 0)  
        count++;  
    else{  
        if(pid2 == 0)  
            count--;  
        else  
            count += 2;  
    }  
    printf("%d", count);  
}
```

What does the process diagram look like?



# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What does count look like?

Parent: pid1 != 0 and pid2 != 0

Child1: pid1 == 0 and pid2 != 0

Child2: pid1 != 0 and pid2 == 0

Grandchild: pid1 == 0 and pid2 == 0

# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What does count look like?

Parent: pid1 != 0 and pid2 != 0

- count = 3

Child1: pid1 == 0 and pid2 != 0

- count = 2

Child2: pid1 != 0 and pid2 == 0

- count = 0

Grandchild: pid1 == 0 and pid2 == 0

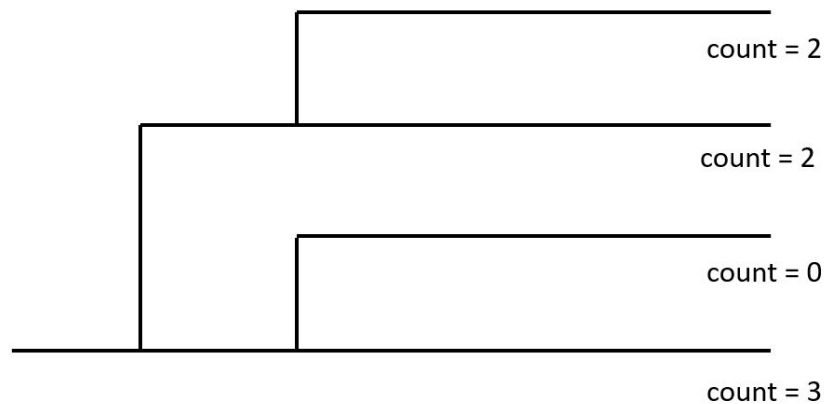
- count = 2

# Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

Given the process diagram, what are the different permutations that can be printed out?



# Processes

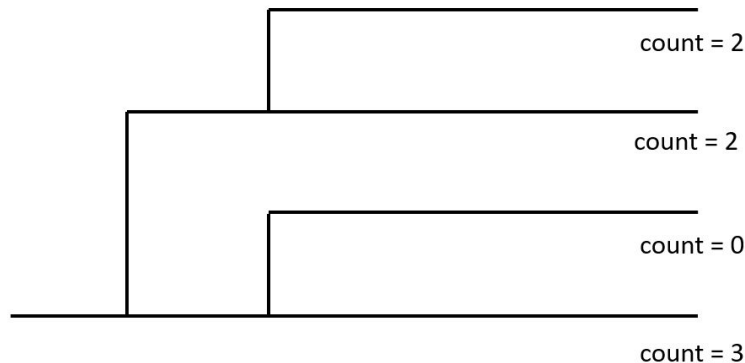
```

int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}

```

Given the process diagram, what are the different permutations that can be printed out?



Math!

$4! / 2 = 12$  different possible outcomes

# Processes



Remember:

- Processes can occur in any order
- Watch out for a `wait` or a `waitpid`!
  - What if I included a `wait(NULL)` before I printed out count?
- Good luck!

# Signals

## who would win?

several hundred lines of  
tshlab code

```
7 void
6 exec_cmdline(char *cmdline, char **argv, sigset_t *set,
5             int bg, int fd_in, int fd_out)
4 {
3     pid_t pid = 0;
2     if ((pid = Fork()) == 0) {
1         // Child process; restore mask and execute job.
314 |         Sigprocmask(SIG_SETMASK, set, NULL);
1
```

one asynchronous boi



SIGCHLD

# Signals

- Child calls `kill(parent, SIGUSR{1,2})` between 2-4 times.

What sequence of kills may print 1?

Can you guarantee printing 2?

What is the range of values printed?

```
int counter = 0;
void handler (int sig) {
    atomically {counter++;}
}
int main(int argc, char** argv) {
    signal(SIGUSR1, handler);
    signal(SIGUSR2, handler);
    int parent = getpid();    int child = fork();
    if (child == 0) {
        /* insert code here */
        exit(0);
    }
    sleep(1);    waitpid(child, NULL, 0);
    printf("Received %d USR{1,2} signals\n", counter);
}
```



## Signals (Contd.)

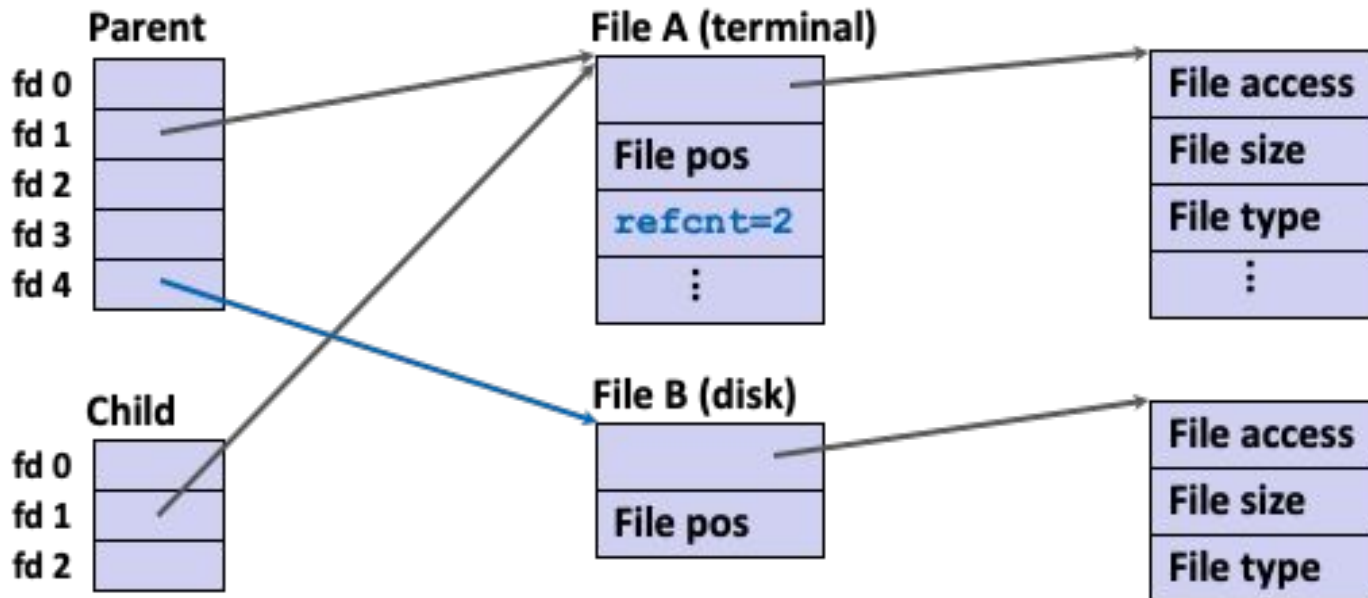
- Sending the same signal to the parent in all the calls to `kill()` may print 1 since there would be no queuing of signals.
- We can guarantee printing 2 if we send precisely one `SIGUSR1` and one `SIGUSR2`.
- We can print 1-4 depending on the manner in which signals are sent and received.

# File IO



# How the Unix Kernel Represents Open Files

**Descriptor table** [one table per process]      **Open file table** [shared by all processes]      **v-node table** [shared by all processes]



# File IO

```
foo.txt: abcdefgh...xyz
int main() {
    int fd1, fd2, fd3;
    char c;
    pid_t pid;
    fd1 = open("foo.txt", O_RDONLY);
    fd2 = open("foo.txt", O_RDONLY);
    fd3 = open("foo.txt", O_RDONLY);
    read(fd1, &c, sizeof(c)); // c = ?
    read(fd2, &c, sizeof(c)); // c = ?
    dup2(fd2, fd3);
    read(fd3, &c, sizeof(c)); // c = ?
    read(fd2, &c, sizeof(c)); // c = ?
}
```

## Main ideas:

- How does read offset?
- How does dup2 work?
  - What is the order of arguments?
  - Does fd3 share offset with fd2?

# File IO

```
foo.txt: abcdefgh...xyz
int main() {
    int fd1, fd2, fd3;
    char c;
    pid_t pid;
    fd1 = open("foo.txt", O_RDONLY);
    fd2 = open("foo.txt", O_RDONLY);
    fd3 = open("foo.txt", O_RDONLY);
    read(fd1, &c, sizeof(c)); // c = a
    read(fd2, &c, sizeof(c)); // c = a
    dup2(fd2, fd3);
    read(fd3, &c, sizeof(c)); // c = b
    read(fd2, &c, sizeof(c)); // c = c
}
```

- How does read offset?
  - Incremented by number of bytes read
- How does dup2 work?
  - Any read/write from fd3 now happen from fd2
  - All file offsets are shared

# File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

## Main ideas:

- How are fd shared between processes?
- How does dup2 work from parent to child?
- How are file offsets shared between processes?

# File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

What would this program print?

Just ignore the possible outcomes due to interleaving ... try two simple cases :

1. First child executes to the end
2. First parent executes to the end.

# File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

## Possible output 1:

```
c = b // in child
c = d // in child
c = c // in child
c = d // in child
c = e // in parent
c = e // in parent
```



# File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

## Possible output 2:

```
c = d // in parent
c = b // in parent
c = c // in child from fd1
c = e // in child from fd3
c = d // in child
c = e // in child
```

# File IO

```
.  
.br/>pid = fork();  
if (pid==0) {  
    read(fd1, &c, sizeof(c));  
    printf("c = %c\n", c);  
    dup2(fd1, fd2);  
    read(fd3, &c, sizeof(c));  
    printf("c = %c\n", c);  
}  
if (pid!=0) waitpid(-1, NULL, 0);  
read(fd2, &c, sizeof(c));  
printf("c = %c\n", c);  
read(fd1, &c, sizeof(c));  
printf("c = %c\n", c);  
return 0;  
}
```

What are the possible outputs now?

# File IO

```
.  
.pid = fork();  
if (pid==0) {  
    read(fd1, &c, sizeof(c));  
    printf("c = %c\n", c);  
    dup2(fd1, fd2);  
    read(fd3, &c, sizeof(c));  
    printf("c = %c\n", c);  
}  
if (pid!=0) waitpid(-1, NULL, 0);  
read(fd1, &c, sizeof(c));  
printf("c = %c\n", c);  
read(fd2, &c, sizeof(c));  
printf("c = %c\n", c);  
return 0;  
}
```

Possible output:

c = b // in child

c = d // in child

c = c // in child

c = d // in child

c = e // in parent

c = e // in parent

# File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
} else {
    close(fd1)
}
if (pid!=0) waitpid(-1, NULL, 0);
read(fd2, &c, sizeof(c));
read(fd1, &c, sizeof(c));
```

- Child creates a copy of the parent fd table
  - dup2/open/close in parent affect the child
  - dup2/open/close in child do NOT affect the parent
- File descriptors across process share the same file offset.

# Malloc



# Malloc

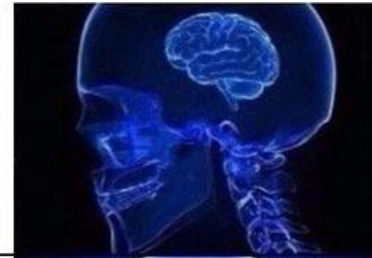
- Fit algorithms - first/next/best/good
- Fragmentation
  - Internal - inside blocks
  - External - between blocks
- Organization
  - Implicit
  - Explicit
  - Segregated

GOOD FIT

BEST FIT

FIRST FIT

EXTEND  
HEAP



# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)						
b = malloc(16)						
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)						
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						



# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)						
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)						
free(b)						

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)						



# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal
  - $(48-16) + (80-48) = 64$
  - external?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

# Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal
  - $(48-16) + (80-48) = 64$
  - external
  - 32

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)						
b = malloc(16)						
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)						
f = malloc(48)						
free(b)						

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)						
free(b)						



# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	64a		
free(b)						

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	64a		
free(b)	80f [0]		32a	64a		

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	64a		
free(b)	80f [0]		32a	64a		

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal
  - $(32-16) + (64-48) = 32$
  - external?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	64a		
free(b)	80f [0]		32a	64a		

# Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal
  - $(32-16) + (64-48) = 32$
  - external
  - 80

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	64a		
free(b)	80f [0]		32a	64a		

Good luck!

