15-213 Final
Review Session

Josh, Parth, Jerry

Sunday, December 8th

Final Exam Logistics

- Thursday December 12, 8:30-11:30AM
- **Location**
  - DH 2210, DH 2315, DH 2302, DH 2105, DH 2122
- Physical Cheat Sheets - 2 pages double sided
  - No previous exam questions
- Bring your IDs to the exam!

Overview of 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)

Overview of 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)**

Structs/Alignment

Alignment Rules

- Primitive Types
  - **`char`**: 1-byte aligned
  - **`short`**: 2-byte aligned
  - **`int`**: 4-byte aligned
  - **`long/pointer-type`**: 8-byte aligned
- Structs
  - Uses the alignment of the largest primitive within the struct.

Example: Struct

- How would the following struct be represented in memory?

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

Example: Struct

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

a1, **a2** are **int**s - 4 bytes each

a1	a1	a1	a1	a2	a2	a2	a2

Example: Struct

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

b,c are 1 btye each and have no alignment requirements

a1	a1	a1	a1	a2	a2	a2	a2
b	c						

Example: Struct

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

d is 4 bytes and must be 4 byte aligned. What is our current alignment status?

- 8+1+1 = 10 => Need padding!

a1	a1	a1	a1	a2	a2	a2	a2
b	c	-	-	d	d	d	d

Example: Struct

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

e is 2 bytes and must be 2 byte aligned. What is our current alignment status?

- 10+1+1+4 = 16 => Already satisfied!

a1	a1	a1	a1	a2	a2	a2	a2
b	c	-	-	d	d	d	d
e	e						

Example: Struct

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

Now we have a constant length array - what is the alignment policy?

- Takes alignment of primitive type!

a1	a1	a1	a1	a2	a2	a2	a2
b	c	-	-	d	d	d	d
e	e	buf	buf	buf	buf	-	-

Example: Nested Struct

- How would the following struct (**`final_nested`**) be represented in memory?

```
struct final {
    int a1;
    int a2;
    char b;
    char c;
    int d;
    short e;
    char[4] buf;
}
```

```
struct final_nested {
    int x;
    struct final;
    long y;
}
```

Example: Nested Struct

- Remember: Structs take the highest alignment requirement of its fields!
- What is the alignment of **`struct final`**?

```
struct final_nested {
    int x;
    struct final;
    long y;
}
```

- Alignment of **`struct final`** is 4
  - **`int`** is the largest type

x	x	x	x	a1	a1	a1	a1
a2	a2	a2	a2	b	c	-	-
d	d	d	d	e	e	buf	buf
buf	buf						

Example: Nested Struct

```
struct final_nested {
    int x;
    struct final;
    long y;
}
```

- Finally, we have a **`long`**, which has alignment of 8 bytes

x	x	x	x	a1	a1	a1	a1
a2	a2	a2	a2	b	c	-	-
d	d	d	d	e	e	buf	buf
buf	buf	-	-	-	-	-	-
y	y	y	y	y	y	y	y

Caches

Caches - Quick Review

- Direct Mapped vs. N-way associative vs. fully associative
  - What do these mean and how might they have an advantage over the other?
- Eviction Policy
  - The main one we covered was LRU (least recently used)

Cache

- Suppose you have a 2-way associative cache with 4 sets and 64 byte blocks.
- What would the address decomposition look like?

… 0 0 0 0 0 0 0 0 0 0 0 0

Cache

- Suppose you have a 2-way associative cache with 4 sets and 64 byte blocks.
- What would the address decomposition look like?
  - 4 sets = 2^2 sets => 2 set bits
  - 64 byte blocks => 2^6 byte blocks => 6 block offset bits
  - Remainder is tag!

… 0 0 0 0 0 0 0 0 0 0 0 0

Cache

- Suppose you have a 2-way associative cache with 4 sets and 64 byte blocks. Assume A and B are cache-aligned.
  - What is the miss rate of pass 1 and pass 2?

```
#define N 128
int get_prod_and_copy(int[N] A, int[N] 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;
}
```

Cache - Pass 1

- We have 64 byte blocks, indicating a cache line holds 16 **`int`**s
- We iterate through 64 elements with stride 4
    - 16 iterations total
- How many iterations access the same cache line?
    - 4 iterations covers 16 elements = one block

```
#define N 128
int get_prod_and_copy(int[N] A, int[N] 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;
}
```

Cache - Pass 1

- Then what is our miss rate?
- 4 iterations cover one cache line, meaning the first is a cold miss, then the next 3 are hits!
- This pattern repeats across all batches of iterations, giving us a miss rate of 1/4

Cache - Pass 2

- Once again we iterate through 64 elements with stride 4
  - 16 iterations total
- Remember our cache does not reset before pass 1 and pass 2.

  What is the state of our cache before pass 2?

```
#define N 128
int get_prod_and_copy(int[N] A, int[N] 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;
}
```

Cache 2 - Pass 2

- We had 4 cache line accesses from the 4 batches of iterations from pass 1. Remember each set has 2 lines and we have 4 sets.

`Set 0`	`A[0-15]`	`-`
`Set 1`	`A[16-31]`	`-`
`Set 2`	`A[32-47]`	`-`
`Set 3`	`A[48-63]`	`-`

- Do we need to evict from the cache during pass 2?

Cache 2 - Pass 2

- **No**, we do not need to evict!
  - We access 4 memory blocks of **B** in pass 2, and since there are 2 lines per set, we do not need to evict

Set 0	A[0-15]	B[0-15]
Set 1	A[16-31]	B[16-31]
Set 2	A[32-47]	B[32-47]
Set 3	A[48-63]	B[48-63]

- Yay! Our cache was the same size as our working set.

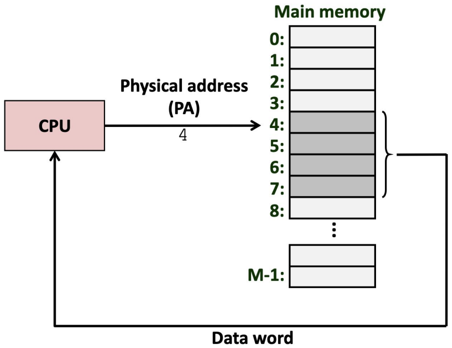
Cache 2 - Pass 2

- Now what is our miss rate?
- Per batch of iterations, we have 4 hits to **`A`**, 1 cold miss to **`B`**, and 3 following hits to **`B`**.
- This yields a miss rate of 1/8

Virtual Memory

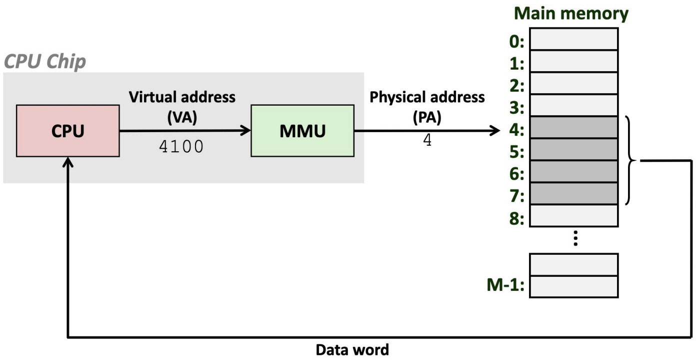
Virtual Memory - Review

Physical Addressing



Memory address refers to an exact location in memory—only used in simple systems

Virtual Addressing



Memory address refers to a process-specific address, mapped to physical memory via the hardware memory management unit.

One of the Great Ideas Of Computer Science™

Virtual Memory - Review

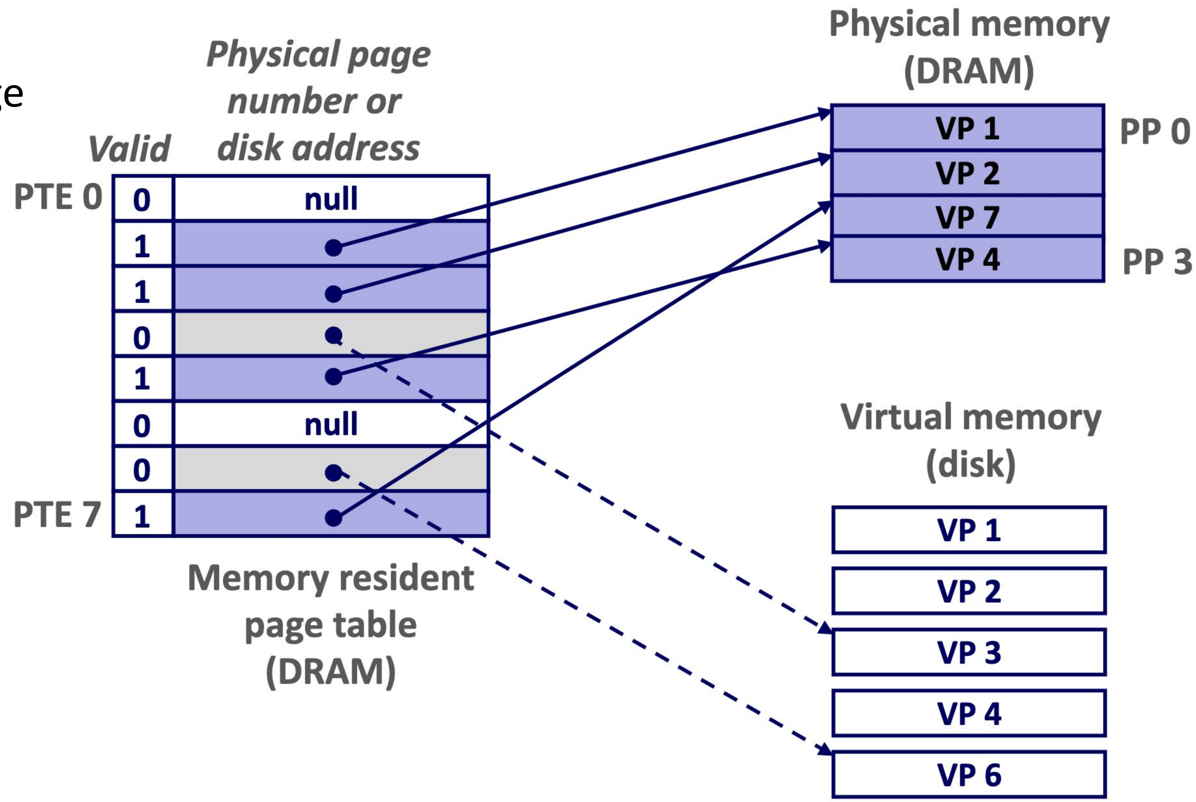
- Now that we've done **`tshlab`**, let's ask: is VM really that helpful?
- It definitely is! Not only does VM give us a way to access the disk, but it also gives us address space isolation!

Virtual Memory - Page Table

Virtual addresses are mapped to physical addresses in the page table. Each entry is called a page table entry.

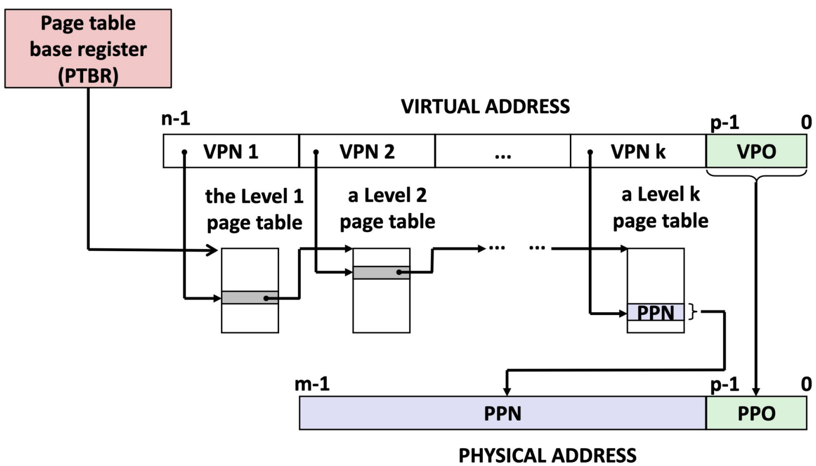
Pages are in memory, like a cache. If they are not available in memory, we have a page miss.

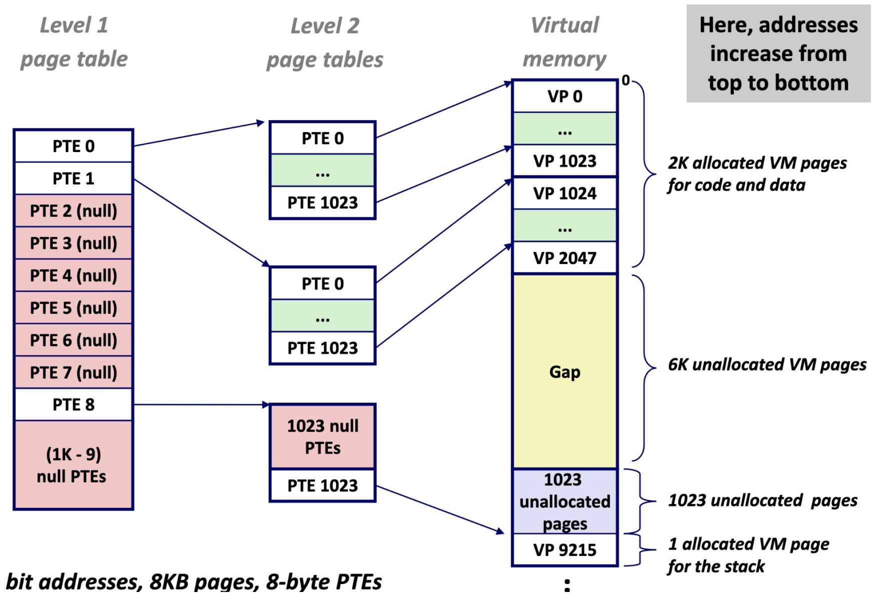
A page miss causes a page fault, which causes the OS to fetch the page from disk and evict a page from DRAM.



Virtual Memory - Multi-Level Page Tables

- The size of a page table quickly gets out of control when we have to address large addresses space.
- The solution is to nest page tables. The VPO/PPO acts as the pseudo-”block offset”





64 bit addresses, 8KB pages, 8-byte PTEs

Example - Multi-Level Page Table

- Consider a system with 32 bit virtual address space and a 24 bit physical address space. Page Size is 4KB. Assume the size of entries in the Page Table is 4 bytes.
- Question of interest : How would we map the virtual address space? Is a single-level page table enough? Do we need more levels? Let’s dive into it….

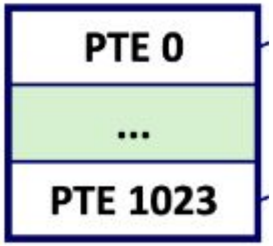
Example (Address Decomp.)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 1: How many bits in the virtual/physical address for page offset?
- VPO = PPO = $\log_2$(page size) = 12 bits

20 bits	12 bits
to be discussed in later slides	offset (VPO = PPO)

Example (PTEs in Pages)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 2: How many PTEs (page table entries) fit inside a single page?
- # of PTEs in a page = size of a page / size of a PTE
  - 4KB/4B = 2^12/2^2 = 2^10 = 1024



Example (Mapping PTEs to VA)

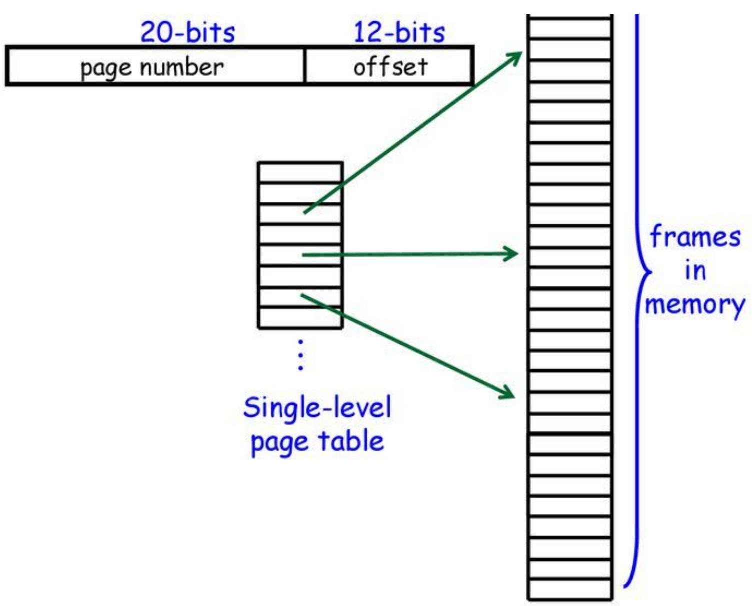
- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 3: How many PTEs are required to map the entire VA space?
- # of PTEs for VA space = size of VA space/size of a page
    - 2^32/2^12 = 2^20 PTEs

Example (Multi-Level Storage)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- So far, we've discussed preliminary values that tell us how to map onto the entire VA space.
  - General/"Single-Level" Ideas
- Now let's talk about how we can extend this to a multi-level page table

Example (Multi-Level Storage)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 4: How many pages do we need to cover the single level page table?
- # of pages for VA space = # of PTEs to map VA space/# of PTEs in a page
  - 2^20/2^10 = 2^10 pages

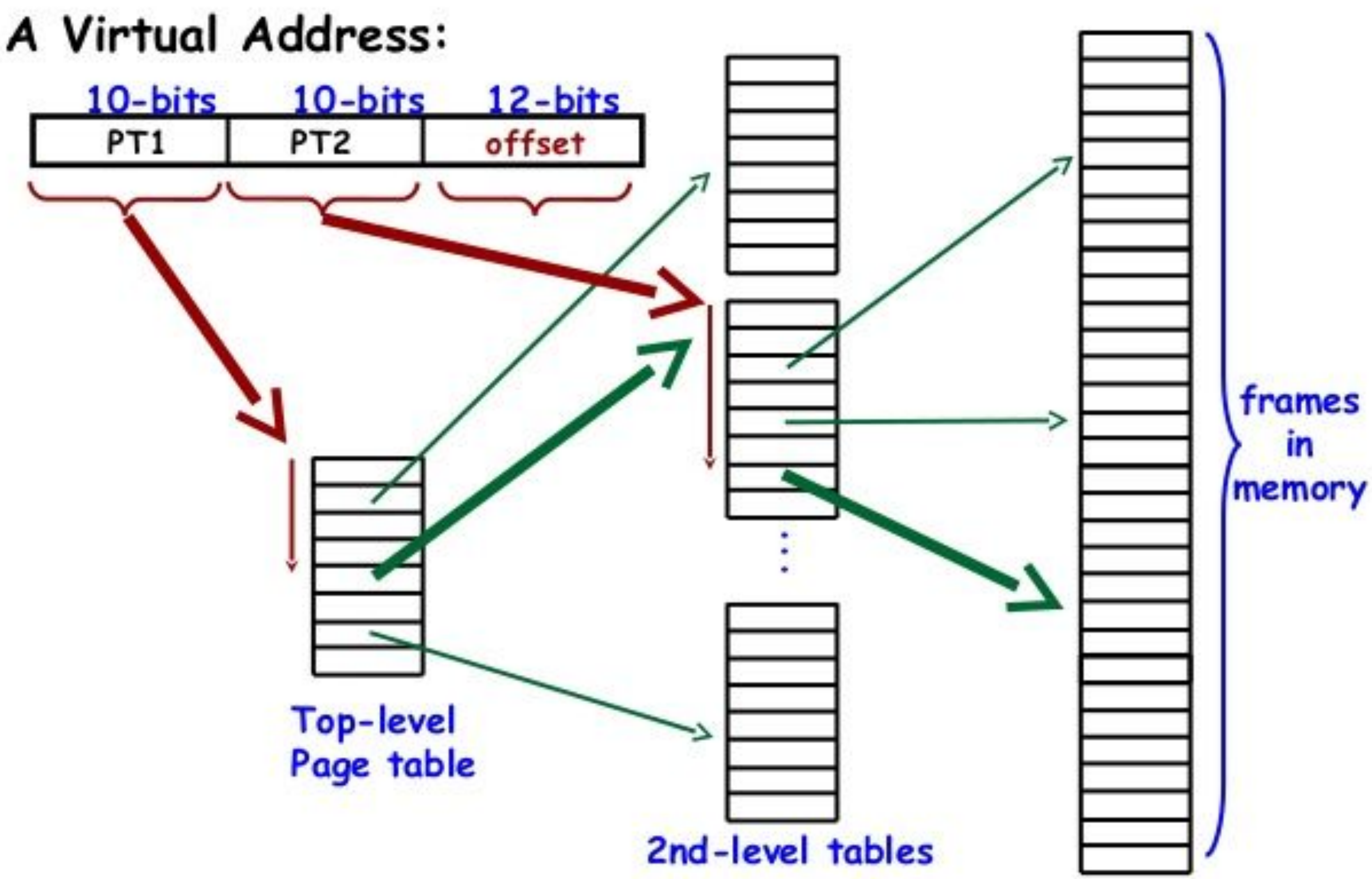


Example (Multi-Level Storage)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 5: How many pages do we need to represent the outer level page table?
- # of pages for outer level = # of pages for VA space / # PTEs in a page
  - 2^10/2^10 = 1 page

Example (Multi-Level Storage)

- This is what our final multi-level page table would look like



Example (Multi-Level Storage)

- Great, now we've setup a 2-level page table, let's talk about the benefits we get.
- Without the outer level, we would have to store the entirety of the single-level page table.
  - Oops that's (2^20 PTEs x 4 bytes) = 2^22 bytes = 4096 KB
  - Also can think of as (2^10 Pages x 4 KB)

Example (Multi-Level Storage)

- Now we have two-levels. Suppose we have a single memory access (assuming the page table was empty at first). How many pages would be required?
- Entire outer level (there is only one page)
- 1 PTE needed from outer level => 1 page in inner level
- Total 2 pages! We saved a huge chunk of space.
  - 2 pages = 8 KB <<<<<< 4096 KB

Processes/Signals

Processes

- Goal: figure out what are possible outcomes printed from executing this program.

```
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);
}
```

Processes

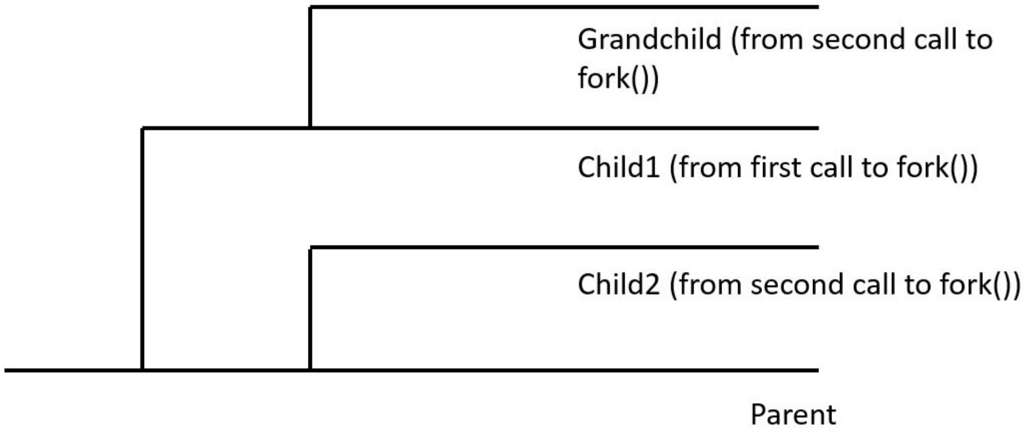
- Parent calls fork twice and forks two children.
- Child with **`pid = pid1`** forks another child.
- In total: 4 processes

```
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);
}
```

Processes

- Now a very important step, draw the process diagram.

```
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);
}
```



Processes

- Parent:
  - `pid1 != 0`
  - `pid2 != 0`
- Child1:
  - `pid1 == 0`
  - `pid2 != 0`
- Child2:
  - `pid1 != 0`
  - `pid2 == 0`
- Grandchild:
  - `pid1 == 0`
  - `pid2 == 0`

```
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);
}
```

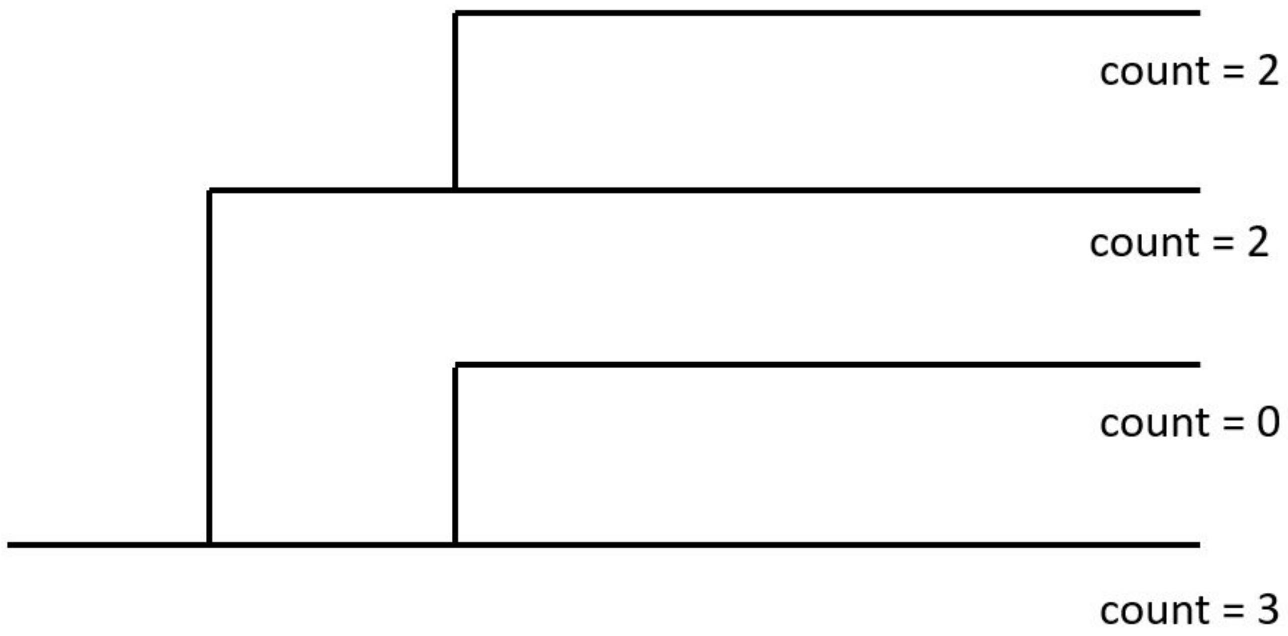
Processes

- Remember: Each process has its own memory space! - Let's figure out the outcomes now
- Parent: **`count = 3`**
- Child1: **`count = 2`**
- Child2: **`count = 0`**
- Grandchild: **`count = 2`**

```
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);
}
```

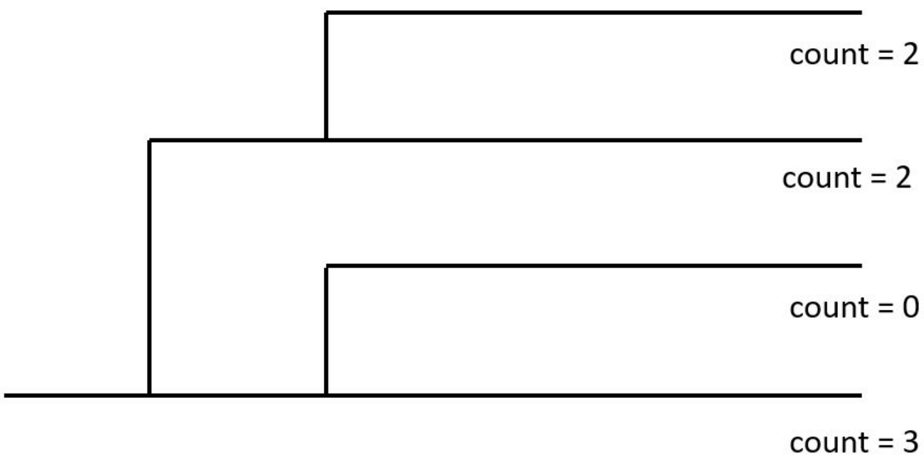
Processes

- Use the process diagram to figure out possible outcomes.
- 4 print branches, 2 repeated values
  - 4! / 2 = 12 different possible outcomes.



Processes

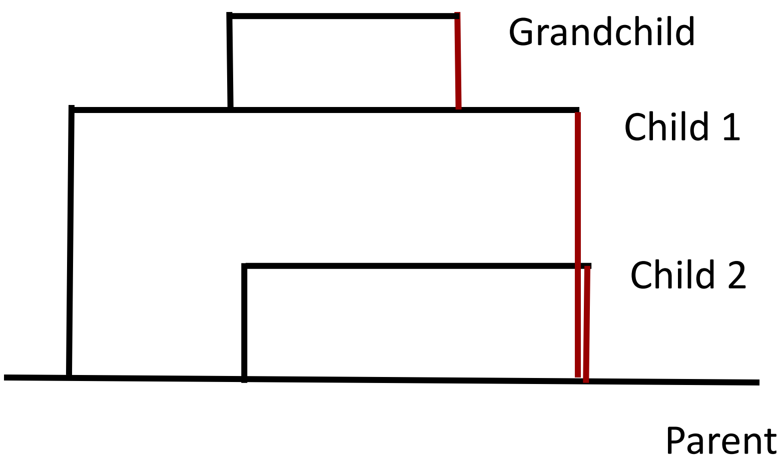
- How does the inclusion of **`wait(NULL)`** change our possible outcomes?




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

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

Processes

- How does the inclusion of **`wait(NULL)`** change our possible outcomes?




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

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

Signals

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

  What sequence of kills may print 1? How can you guarantee printing 2? What is the range of values printed?

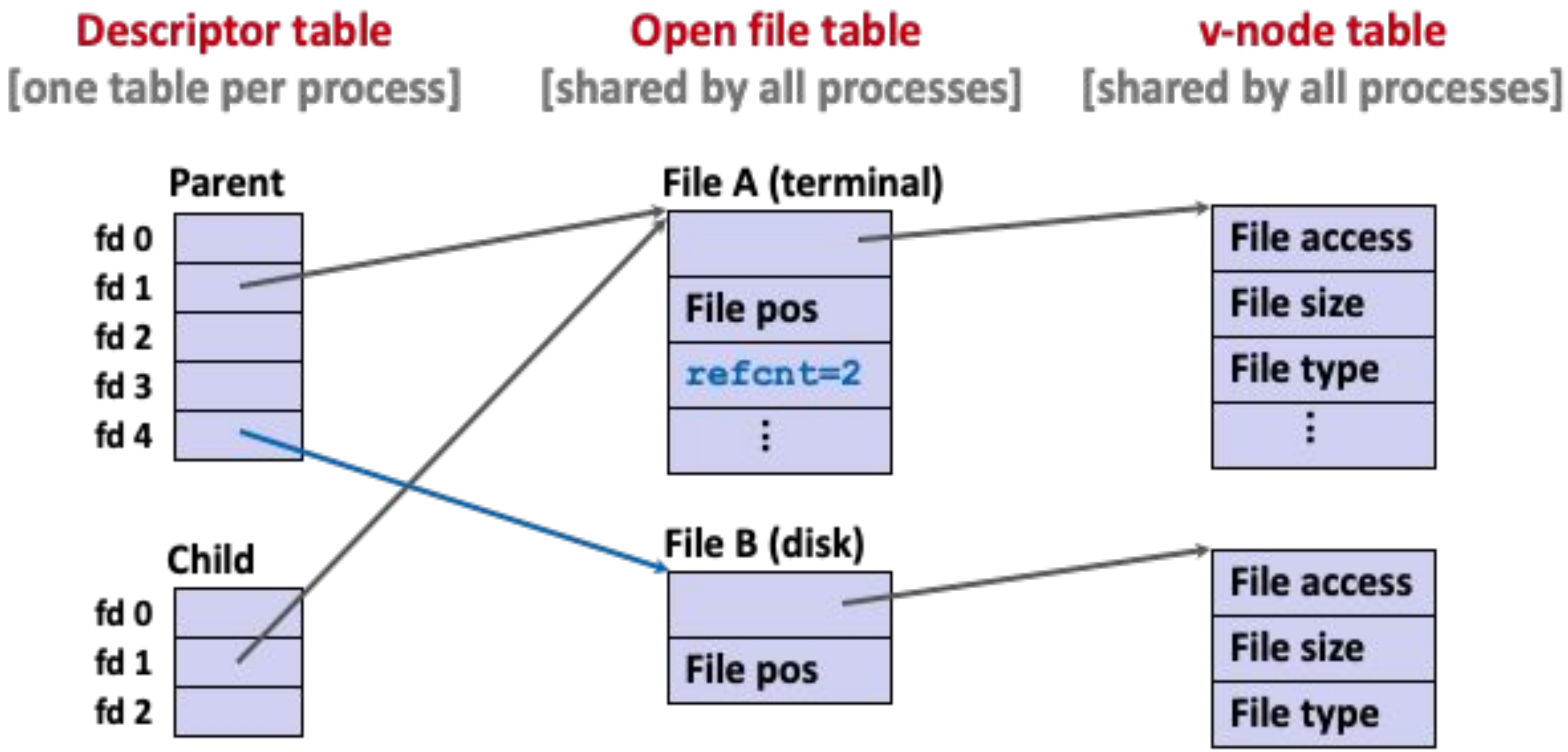
```
int counter = 0;
void handler (int sig) {
  atomically {counter++;}
}
int main(int argc, char** argv) {
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);
  int parent = getpid();   int child = fork();
  if (child == 0) {
    /* insert code here */
    exit(0);
  }
  sleep(1);   waitpid(child,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

Signals - Solution

- Sending the same signal to the parent in all the calls to kill() may print 1 since there would be no queuing of signals.
  - All the signals can coalesce and get handled at once
- We can guarantee printing 2 if we send precisely one SIGUSR1 and one SIGUSR2.
  - Different signals do not coalesce!
- We can print 1-4 depending on the manner in which signals are sent and received.

File I/O

Open files structures



File I/O

- How does **read** offset the current position?
  - Incremented by number of bytes read
- How does **dup2** work?
  - **dup2(old, new)**
  - points new to old
- Does fd3 share offset with fd2? (after **dup2**)
  - Yes
- What about before **dup2**?
  - No

```
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 = ?
}
```

File I/O

- How are file descriptors and open file tables shared between parent and children?
  - Descriptor table is copied, open file tables and v-node tables are shared

```
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);
```

File I/O

- Child creates a copy of the parent fd table
  - **`dup2/open/close`** in child do NOT affect the parent and vice versa
- File descriptors across processes share the same file offset.
- Many possible outputs!

```
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);
```

File I/O

- Parent then child, no interleaving case:
  - 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

```
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);
```

File I/O

- Child then parent, no interleaving case:
  - c = b // in child
  - c = d // in child
  - c = c // in child
  - c = d // in child
  - c = e // in parent
  - c = e // in parent

```
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);
```

File I/O

- What does adding a **`waitpid`** here do?

```
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);
}
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);
```

Threading/Synchronization

Classical Problems in Threading

- **Deadlock**
  - Two or more threads are unable to proceed because each is waiting for a resource that the other holds.
- **Livelock**
  - Two or more threads continuously change their state in response to each other - but with no further progress.
- **Starvation**
  - One of more threads continuously denied access to resources because other threads holds them.

Threads

- What variables might be shared in this code?

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

#define NUM_THREADS 2
int balance = 10;
int fail_count = 0;

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?
    printf("fail_count: %d\n", fail_count); // What is fail_count?
    return 0;
}
```

Threads

- What are some possible execution orders given these functions?

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

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

```
void *threadA(void *vargp) {
    deposit(4);
    withdraw(11);
    return NULL;
}

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

Threads

- Simple case where each thread fully executes their function calls to deposit and withdraw.

Thread A deposit(4)			Thread A withdraw(11)	
	Thread B withdraw(6)	Thread B deposit(3)		Thread B withdraw(7)
balance: 14 fail_count: 0	balance: 8 fail_count: 0	balance: 11 fail_count: 0	balance: 0 fail_count: 0	balance: 0 fail_count: 1

Threads

- Are we guaranteed each thread finishes their calls to deposit and withdraw?
- **No**, interleaving can take place within these functions!
- Even loading and storing variables are multi-step operations that can be interleaved.

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

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

Threads

- Assume Thread A just completed deposit(4) and balance = 14.

Thread A enters withdraw(11)	**Computes balance - amt = 3**		**Sets balance = 3**	

Thread B enters withdraw(6)		**Computes balance - amt = 8**		**Sets Balance = 8**

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

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

Threads

- How can we make this thread safe with one lock?

```
int withdraw(int amt) {
    pthread_mutex_lock(&lock);
    if (balance >= amt) {
        balance = balance - amt;
        pthread_mutex_unlock(&lock);
        return 0;
    } else {
        fail_count++;
        pthread_mutex_unlock(&lock);
        return -1;
    }
}
```

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

- Can we do better?

Threads

- What are our critical resources?
  - The two global variables!
  - Note: They do not need to be protected against each other; only within accesses to the same global
- Let's use two locks instead!

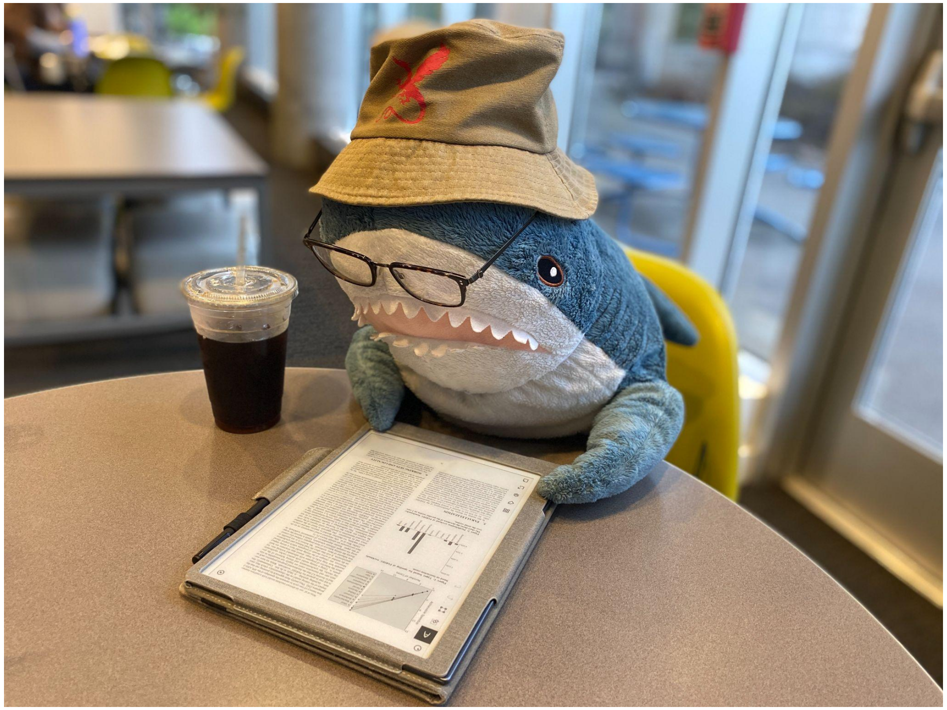
Threads

```
int withdraw(int amt) {
    pthread_mutex_lock(&balance_lock);
    if (balance >= amt) {
        balance = balance - amt;
        pthread_mutex_unlock(&balance_lock);
        return 0;
    } else {
        pthread_mutex_unlock(&balance_lock);
        pthread_mutex_lock(&fail_lock);
        fail_count++;
        pthread_mutex_unlock(&fail_lock);
        return -1;
    }
}
```

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

- Marginal benefit in this case as we perform trivial tasks in each case, but will lead to large gains if functions are more complex.

GOOD LUCK!!



[Requin is studying with you guys too :)]

Q/A

Other Practice Questions
(if time remains/for self-reference)

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 `%cl` 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    %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 ]; [        ]; [        ]){
    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 = [ z ]; [ ]; [ ]){
    e = i + 2;
    e = [ ];
    e = [ ];
    d = [ ];
  }
  return [ ];
}
```

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 = ________ ; ________ ; i++ ){
    e = i + 2;
    e = z ________ ;
    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 = [ z ] ; [ x > i ] ; [ 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
```

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

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 = [        ];
    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 = [z]; [x > i]; [i++]){
    e = i + 2;
    e = [e << y];
    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?

%ecx	%cx	%ch	%cl

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 = [        ];
    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.

```c
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 = [          ] ;
  }
  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 = [ z ]; [ x > i ]; [ i++ ]){
    e = i + 2;
    e = [ e << y ];
    e = [ e >> (y - 1) ];
    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 = [ 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 = [ z ]; [ x > i ]; [ i++ ]){
    e = i + 2;
    e = [ e << y ];
    e = [ e >> (y - 1) ];
    d = [ e + d ];
  }
  return [ d ];
}
```

```
# 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 [ d ] ;
}
```

```
# 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
```

Arrays

Arrays

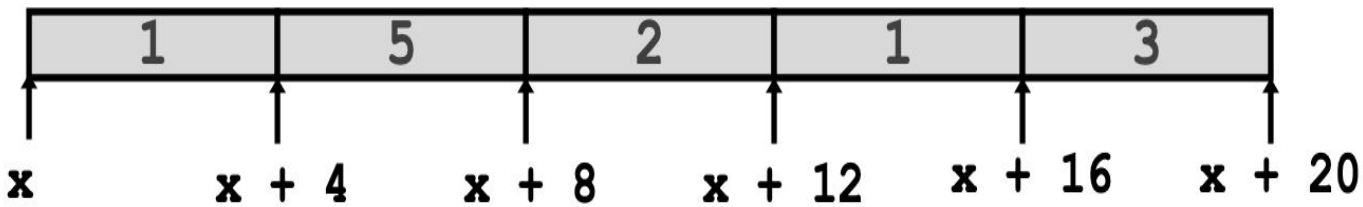
IMPORTANT POINTS + TIPS:

- ***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:

```
int val[5];
```



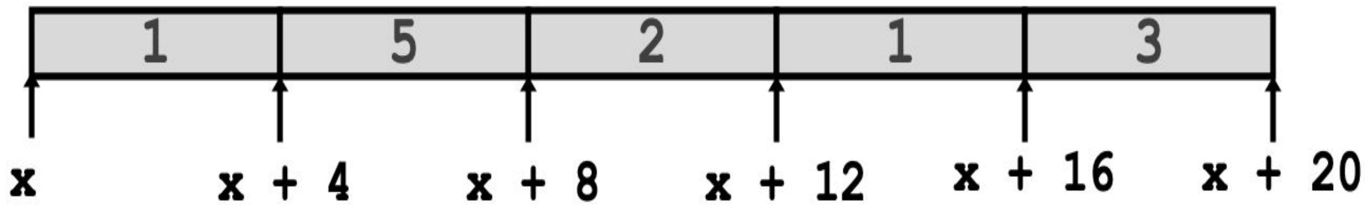
- **A** can be used as the pointer to the first array element: `A[0]`

	Type	Value
`val`		
`val[2]`		
`*(val + 2)`		
`&val[2]`		
`val + 2`		
`val + i`		

Arrays

Good toy examples:

`int val[5];`



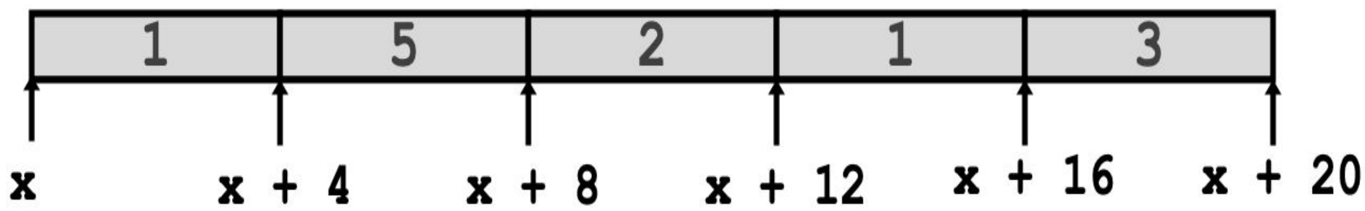
- `A` can be used as the pointer to the first array element: `A[0]`

	Type	Value
`val`	`int *`	`x`
`val[2]`	`int`	`2`
`*(val + 2)`	`int`	`2`
`&val[2]`	`int *`	`x + 8`
`val + 2`	`int *`	`x + 8`
`val + i`	`int *`	`x + (4 * i)`

Arrays

Good toy examples:

```
int val[5];
```



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

	Type	Value
`val`	`int *`	`x`
`val[2]`	`int`	`2`
`*(val + 2)`	`int`	`2`
`&val[2]`	`int *`	`x + 8`
`val + 2`	`int *`	`x + 8`
`val + i`	`int *`	`x + (4 * i)`

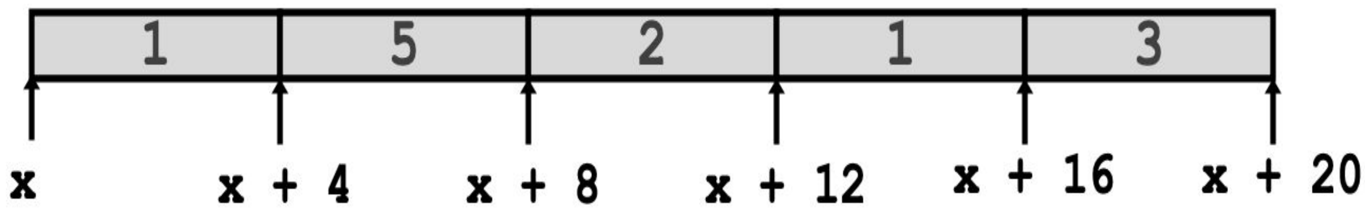
Accessing methods:

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

Arrays

Good toy examples:

```
int val[5];
```



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

	Type	Value
`val`	`int *`	`x`
`val[2]`	`int`	`2`
`*(val + 2)`	`int`	`2`
`&val[2]`	`int *`	`x + 8`
`val + 2`	`int *`	`x + 8`
`val + i`	`int *`	`x + (4 * i)`

Accessing methods:
- *val[index]*
- **(val + index)*

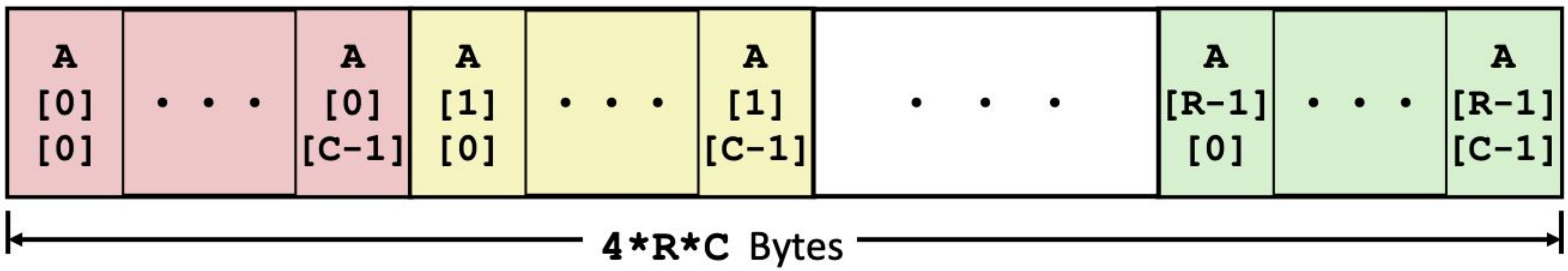
Addressing methods:
- *&val[index]*
- *val + index*

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];`

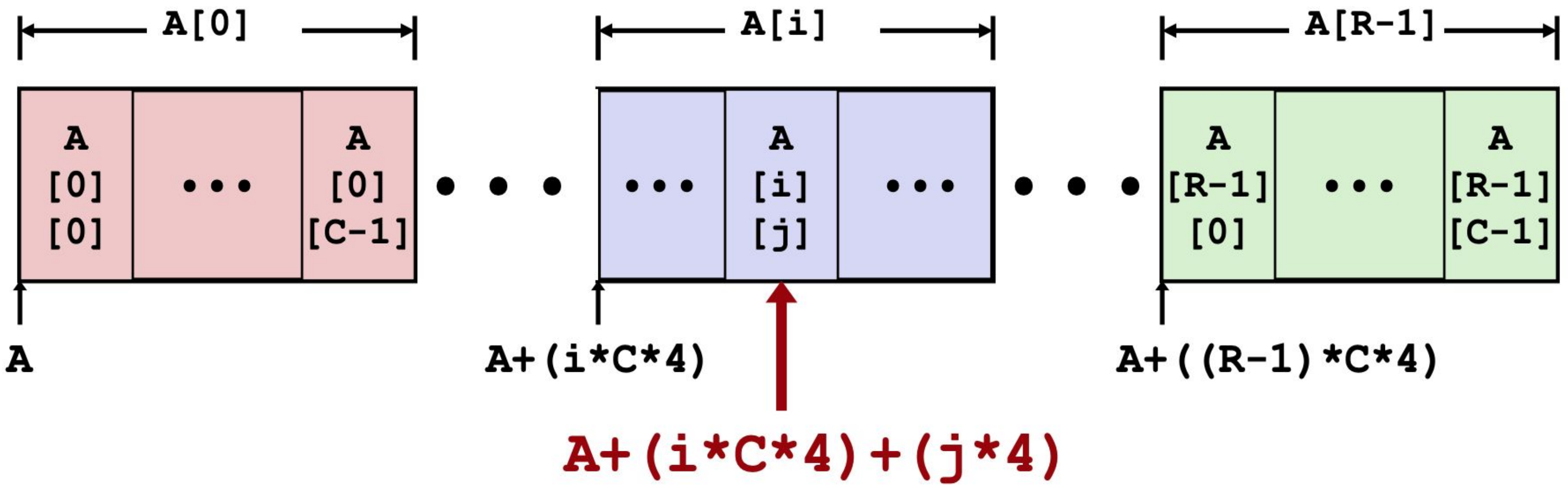


Arrays

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

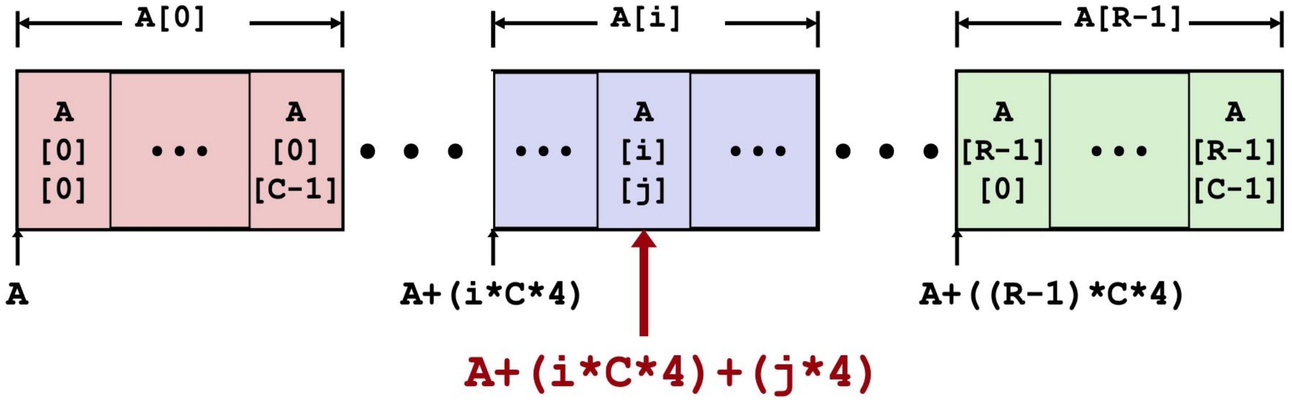
Nested indexing rules:

A[i][j] is element of type *T*, which requires *K* bytes

Address A + i * (C * K) + j * K

= A + (i * C + j) * K

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



Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	`3*5*(4) = 60`
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Arrays

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

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`	Y	N	1*8 = 8
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`	Y	N	1*8 = 8
`int *(A4[3][5])`	Y	N	3*5*(8) = 120
`int (*A5[3])[5]`			

Arrays

Consider accessing elements of A....

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

Arrays

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

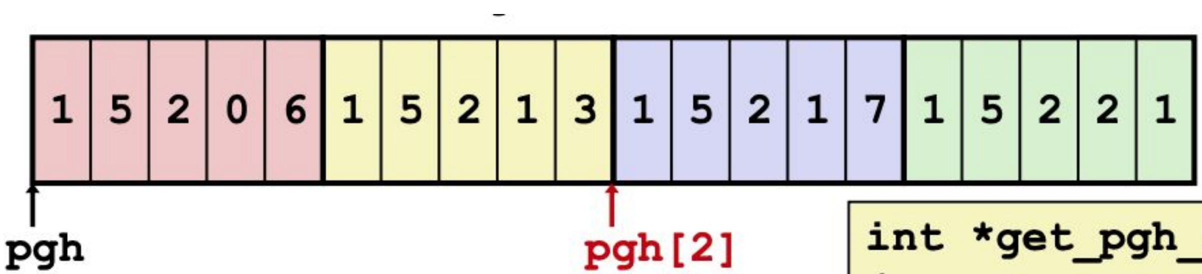
Arrays

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

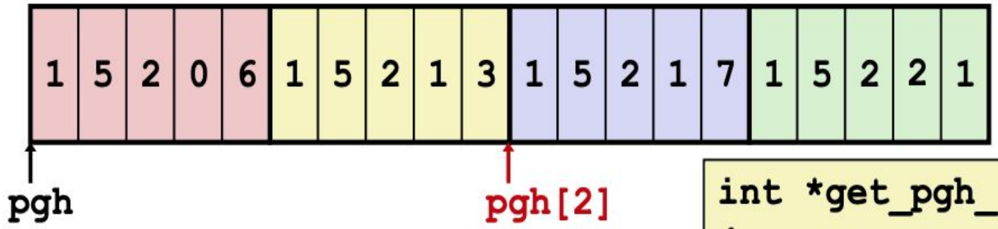



```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

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

Arrays

Nested Array Row Access Code



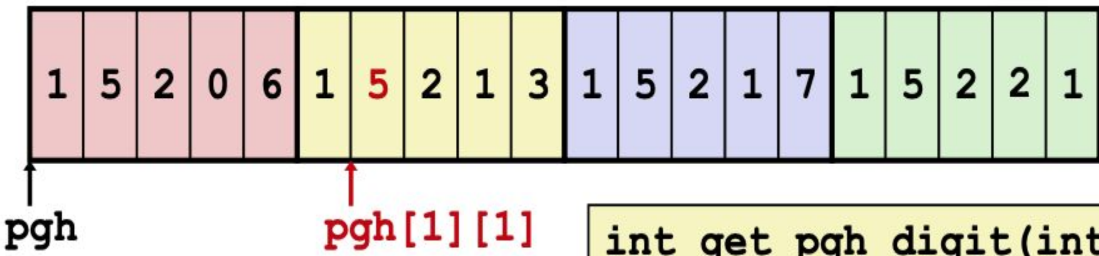

```
int *get_pgh_zip(int index)
{
   return pgh[index];
}
```

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

- **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`
    `= pgh + 4*(5*index + dig)`

Malloc

Virtual Memory - Tracing

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

Final S-02 (#5)

Lecture 17: VM - Systems

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	7	0	010	1	0
001	5	0	011	3	0
002	1	1	012	3	0
003	5	0	013	0	0
004	0	0	014	6	1
005	5	0	015	5	0
006	2	0	016	7	0
007	4	1	017	2	1
008	7	0	018	0	0
009	2	0	019	2	0
00A	3	0	01A	1	0
00B	0	0	01B	3	0
00C	0	0	01C	2	0
00D	3	0	01D	7	0
00E	4	0	01E	5	1
00F	7	1	01F	0	0

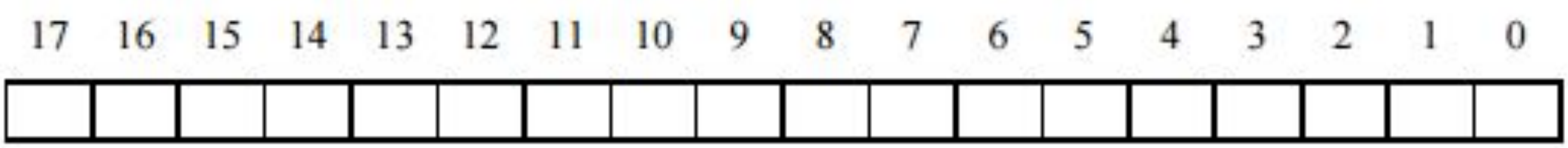
TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
1	71	5	1
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

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

Virtual Memory

Label the following:

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



Virtual Memory

Label the following:

(A) *VPO:* Virtual Page Offset - Location in the page

Page Size = 512 Bytes = $2^9 \rightarrow$ Need 9 bits

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

Virtual Memory

Label the following:

(A) *VPO:* Virtual Page Offset

(B) *VPN:* Virtual Page Number - 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

Virtual Memory

Label the following:

(A) *VPO:* Virtual Page Offset

(B) *VPN:* Virtual Page Number

(C) *TLBI:* TLB Index - Location in the TLB Cache

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

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 → 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

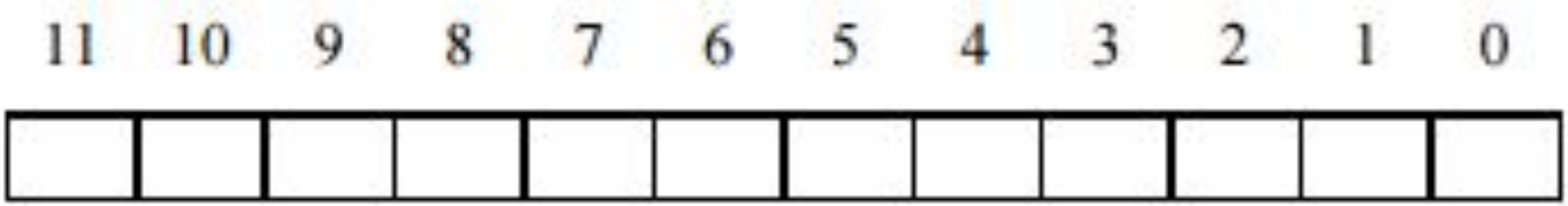
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A	A

TLBT TLBI

Virtual Memory

Label the following:

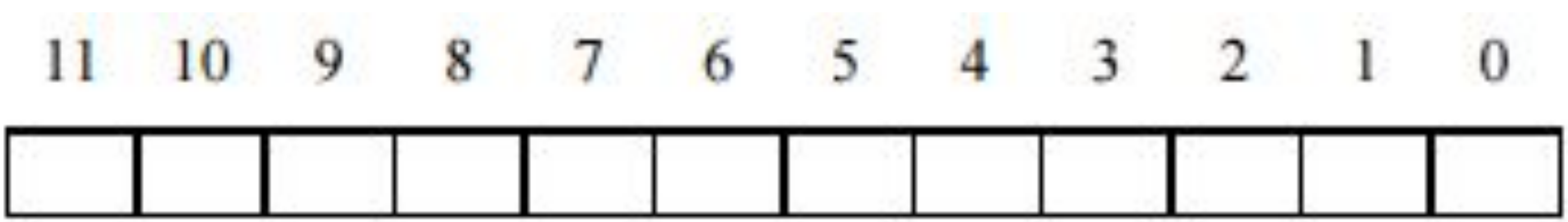
(A) *PPO:* Physical Page Offset
(B) *PPN:* Physical Page Number
(C) *CO:* Cache Offset
(D) *CI:* Cache Index
(E) *CT:* Cache Tag



Virtual Memory

Label the following:

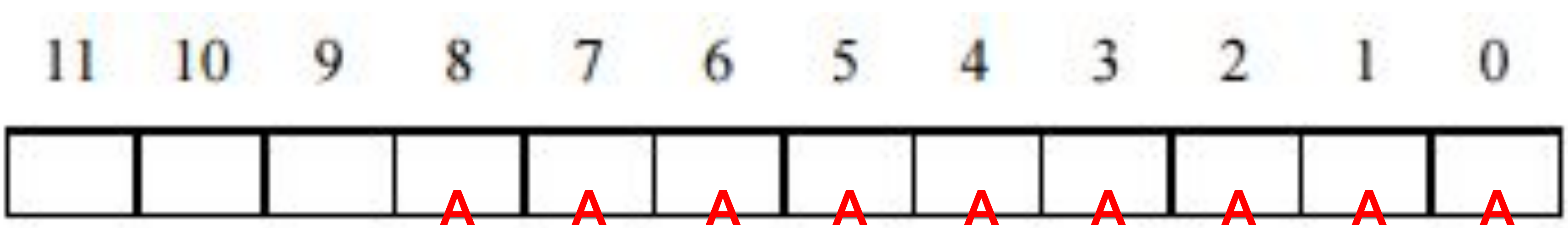
(A) *PPO:* Physical Page Offset



Virtual Memory

Label the following:

(A) *PPO:* Physical Page Offset - Same as VPO

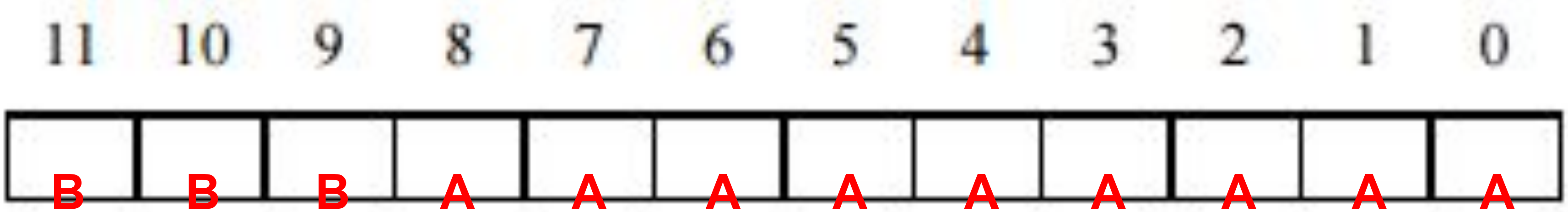


Virtual Memory

Label the following:

(A) *PPO:* Physical Page Offset - Same as VPO

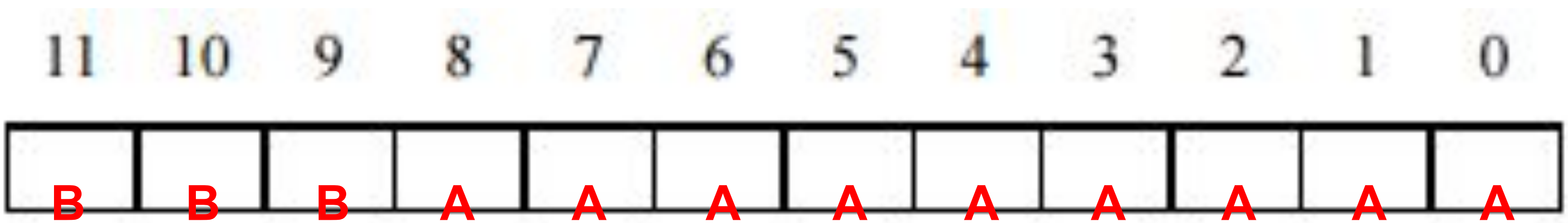
(B) *PPN:* Physical Page Number - Everything Else



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

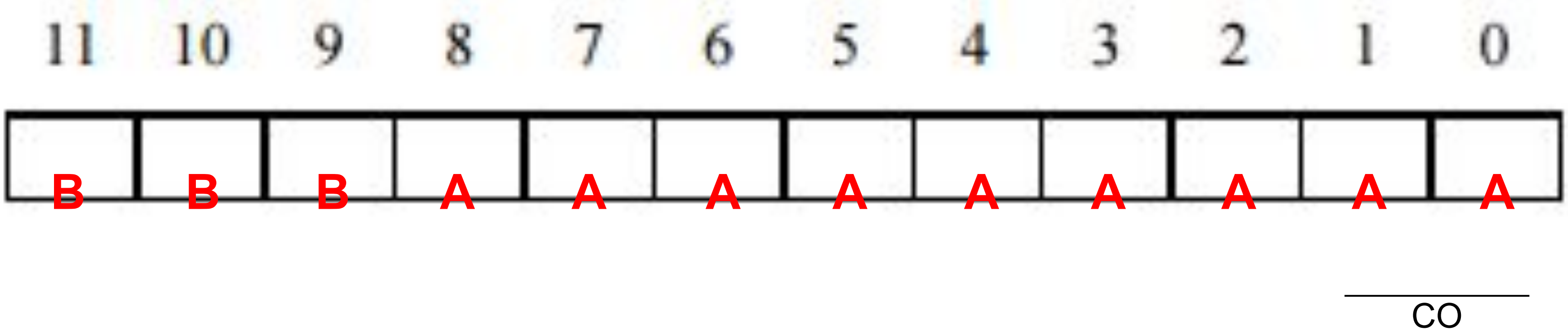


Virtual Memory

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 → 2 Bits



Virtual Memory

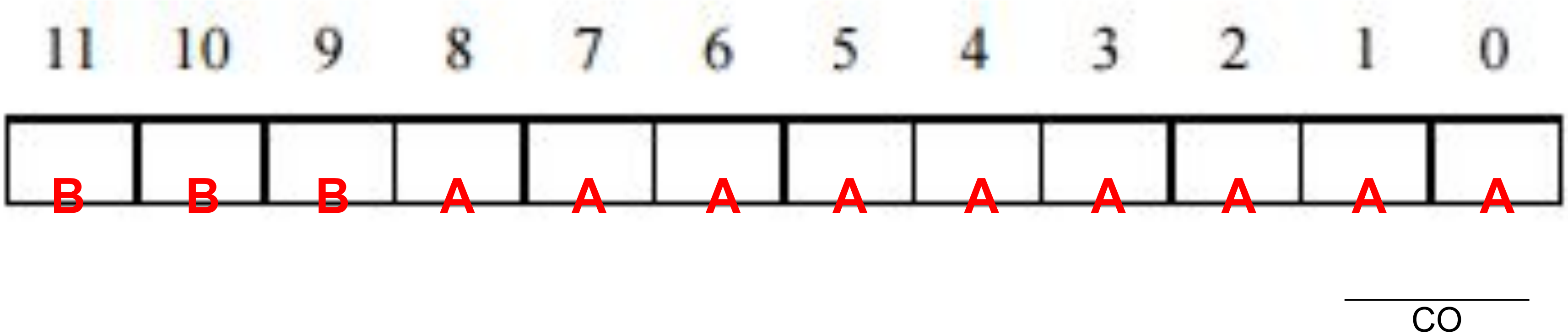
Label the following:

(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



Virtual Memory

Label the following:

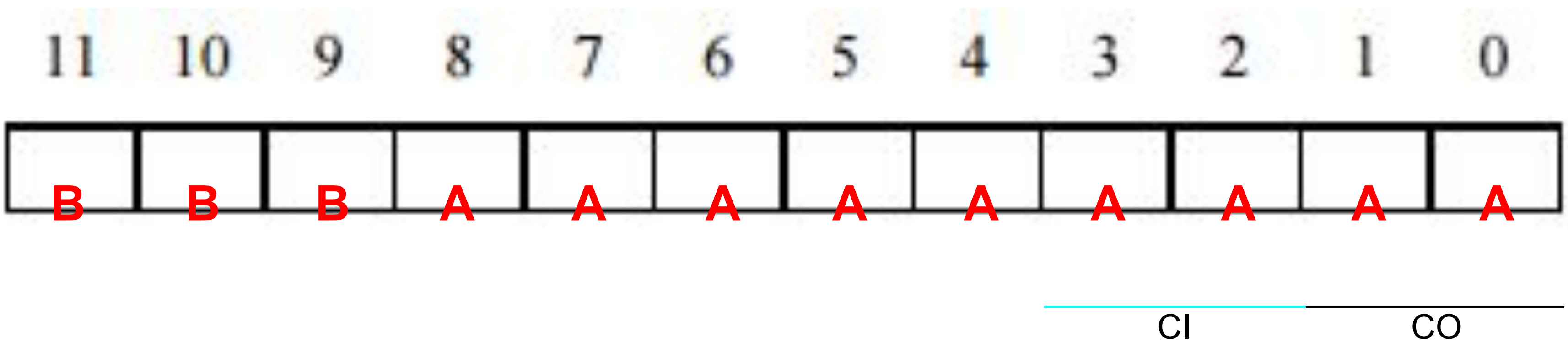
(A) *PPO:* Physical Page Offset - Same as VPO

(B) *PPN:* Physical Page Number - Everything Else

(C) *CO:* Cache Offset - Offset in Block

(D) *CI:* Cache Index

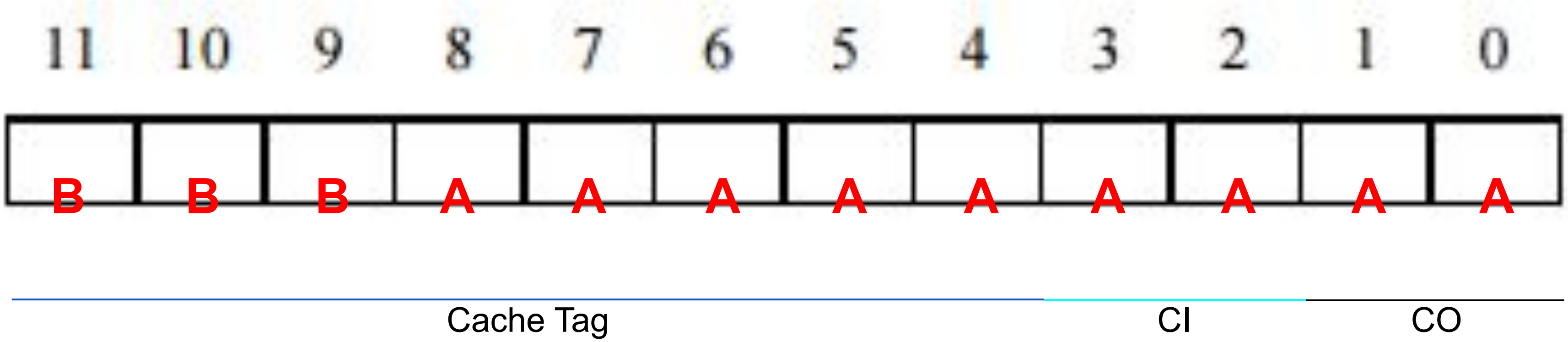
4 Indices → 2 Bits



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**

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

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

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

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??

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	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: 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	1
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

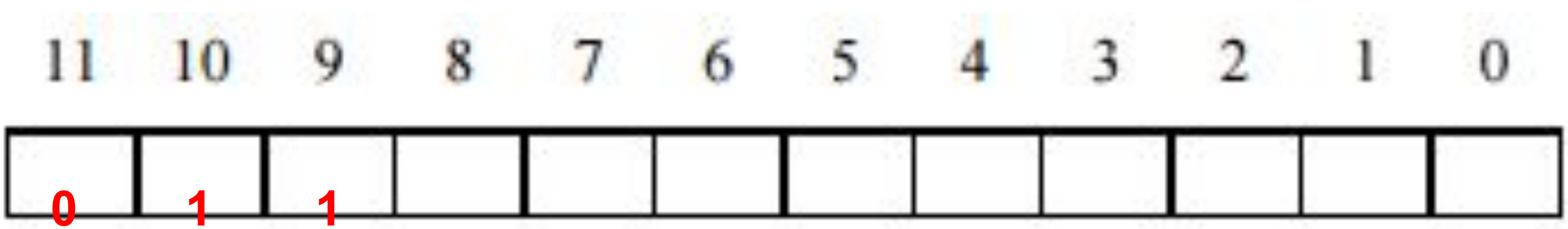
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0

Virtual Memory

Now to the actual question!

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

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

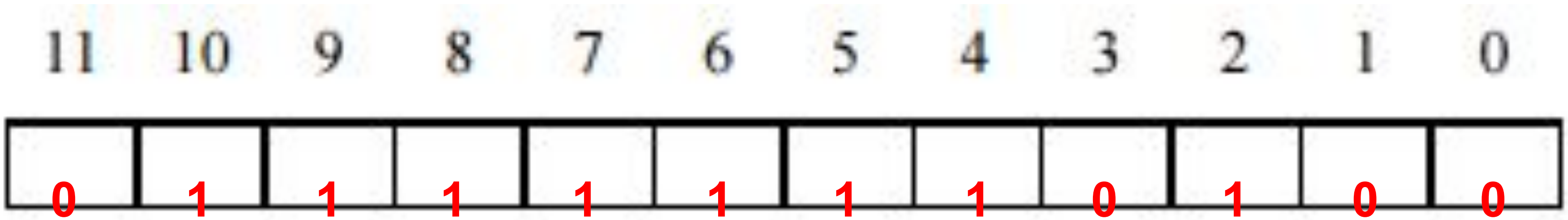


Virtual Memory

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