

UPE Socials

# UPE Tutoring: CS 33 Final Review

Sign-in <https://tinyurl.com/cs33finals22signin>

Slides link available upon sign-in



# Midterm Content:

<https://tinyurl.com/cs33midtermssp22>



# Floating Point Numbers



# Floating Point Numbers

## IEEE Floating Point

[s][ e ][ f ]

- s = sign bit, e = exponent, f = mantissa
- IEEE-754 32-bit 'Single'
  - 1 sign bit, 8 exponent bits, 23 fraction bits
- IEEE-754 64-bit 'Double'
  - 1 sign bit, 11 exponent bits, 52 fraction bits



# Floating Point Numbers

## IEEE Floating Point

[s][ e ][ f ]

- First separate the sign, exponent and the fraction bits **according to the standard**
  - Maybe the exam uses 1 sign, 4 exponent, 10 fraction bits. Use that then!
- Then, interpret the bits in [e] part as a positive integer.
  - That's the value of **e** for now.



# Floating Point Numbers

IEEE-754 32-bit 'Single'  
1 sign, 8 exponent, 23 fraction bits

## Normalized Numbers

When exponent in range  $1 \leq e \leq 254$  (bits are of the form 0...01 to 1...10)

$$\text{result} = (\text{sign}) * 2^{(e - 127)} * 1.f$$

## Denormalized “Tiny” Numbers

When  $e == 0$

$$\text{result} = (\text{sign}) * 2^{(e - 127)} * 0.f$$



# Floating Point Numbers

IEEE-754 32-bit 'Single'  
1 sign, 8 exponent, 23 fraction bits

## Why do we subtract exponent bias from the value of $e$ ?

- Linearly displace so that roughly half of the  $e$  represent negative exponent

## Why exactly 127?

- About the mid of the possible values of  $e$ , when  $e$  is 8 bits

$$\text{Bias} = \frac{2^{|e|}}{2} - 1$$

$$127 = \frac{2^8}{2} - 1 = 128 - 1$$



# Floating Point Numbers

IEEE-754

Applies to any bit distribution

What's 1.f or 0.f?

If f is  $1\ 1\ 1\ 0\ 0\ 0\ 0\dots$

Then 1.f is  $1.1\ 1\ 1\ 0\ 0\ 0\ 0\dots$   
=  $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + 0 + 0 + 0 + \dots$   
=  $1 + 0.5 + 0.25 + 0.125$   
=  $1.875$





# Floating Point Numbers

IEEE-754 32-bit 'Single'  
1 sign, 8 exponent, 23 fraction bits

- Infinity
  - $e = 255, f = 0$
- Zero
  - $s = 0 \text{ or } 1, e = 0, f = 0$
- NaN
  - $e = 255, f \neq 0$
  - Most operations involving NaN will return a NaN (e.g. addition)
  - Comparing NaN to any value will return false/0
    - `(NaN == 4)` // equals 0
    - `(NaN == NaN)` // also equals 0



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

0 10001101 100000000000000000000000



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$

Exponent:  $10001101_2 = 141_{10} = E$



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$

Exponent:  $10001101_2 = 141_{10} = E$

$F = 100000000000000000000000_2 = 2^{-1}$



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$

Exponent:  $10001101_2 = 141_{10} = E$

$F = 100000000000000000000000_2 = 2^{-1} = 0.5 = F$



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$

Exponent:  $10001101_2 = 141_{10} = E$

$F = 100000000000000000000000_2 = 2^{-1} = 0.5 = F$  or  $1.F = 1.5$

$$-1^0 * 2^{141 - 127} * 1.5$$





# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$

Exponent:  $10001101_2 = 141_{10} = E$

$F = 100000000000000000000000_2 = 2^{-1} = 0.5 = F$  or  $1.F = 1.5$

$$-1^0 * 2^{141 - 127} * 1.5 = 16384 * 1.5$$



# Floating Point Numbers

Convert the following single precision 32-bit floating point number to the corresponding decimal value.

01000110110000000000000000000000

$$-1^S * 2^{(e - 127)} * 1.F$$

Signed bit: 0,  $S = 0$

Exponent:  $10001101_2 = 141_{10} = E$

$F = 100000000000000000000000_2 = 2^{-1} = 0.5 = F$

$$-1^0 * 2^{141 - 127} * 1.5 = 16384 * 1.5 = \mathbf{24576 :)}$$



# Floating Point Numbers

- Problems
  - Overflow and underflow
  - Invalid operations (e.g., divide by zero, square root of negative)
  - Precision issues
    - $(1 + 23) \times \log_{10}(2) \approx 7.22$  digits of precision for float
    - $(1 + 52) \times \log_{10}(2) \approx 15.95$  digits of precision for double
- Tips
  - About half of all numbers between -1 and 1 in floating point
  - Write calculations to return results in that range
  - `printf("%.17f\n", value)`
    - Usually the best (lossless) printing of double value



# Floating Point Numbers

What is the first positive integer that can't be represented exactly by a float?




# Floating Point Numbers

What is the first positive integer that can't be represented exactly by a float?

$$2^{(1 + 23)} + 1 = 16,777,217$$

What happens if you try to add 1 to 16,777,216?

$$\begin{array}{l} 0 \quad 10010111 \quad 000000000000000000000000 \\ 1 \times \quad 2^{24} \quad \times 1.000000000000000000000000_2 = 10000000000000000000000000.0_2 \end{array}$$


After moving the decimal point to the right 24 times, we see that the one's place is not represented by f



# Practice Question

Consider floating point with 4 fractional bits and 3 exponent bits (bias = 3).  
What is the smallest normal positive number representable?



# Answer

Consider floating point with 4 fractional bits and 3 exponent bits (bias = 3)

What is the smallest normal positive number representable?

Without denormalized numbers, smallest nonzero number:

$$0 \ 001 \ 0000 = V1 = 2^{1-3} = 2^{-2}$$

next smallest number:

$$0 \ 001 \ 0001 = V2 = 2^{-2} + 2^{-6}$$



# Practice Question

In floating point arithmetic, what is the purpose of having denormalized numbers?





# Answer

In floating point arithmetic, what is the purpose of having denormalized numbers?

Represent even smaller numbers



# Practice Question

What's the point of having a varying range between numbers?



# Answer

What's the point of having a varying range between numbers?

- As numbers increase, significance of smaller values decrease
  - Value of 1 relative to 10 is quite significant
  - Value of 1 relative to  $10^9$  is insignificant



# Answer

What's the point of having a varying range between numbers?

- As numbers increase, significance of smaller values decrease
  - Value of 1 relative to 10 is quite significant
  - Value of 1 relative to  $10^9$  is insignificant
- Floating point allows representation of very large numbers by losing precision
  - Accomplished by each bit contributing a value that increases along with the exponent of the number



# Program Optimization



# Program Optimization

## Measuring Performance

In a shell, use the 'time' command

```
$ time ...
```

Ex: `$ time sort large_file.txt`

In C, use the function

```
clock_gettime(CLOCK_REALTIME, &ts)
```

```
// ts = 'timespec' struct containing:
```

```
//      time_t s (seconds since 1970/1/1)
```

```
//      long ns  (nanoseconds since start of that second)
```



# Program Optimization

**Hoisting / code motion:** doing more computation out of a loop

```
ListObject my_list;  
for (i = 0; i < my_list.size(); ++i) {  
    //do something with my_list that doesn't change its size  
}
```

Compiler generally can't know `my_list.size()` will stay the same



# Program Optimization

**Hoisting / code motion:** doing more computation out of a loop

Compared to:

```
ListObject my_list;  
int size = my_list.size();  
for (i = 0; i < size; ++i) {  
    //do something with my_list that doesn't change its size  
}
```

Faster, because no repeated calls to `my_list.size()`!





# Program Optimization

**Loop Unrolling:** doing more in the loop to reduce overhead

```
for (i = 0; i < 100; ++i) {  
    // Do something with i  
}
```

vs

```
for (i = 0; i < 100; i += 2) {  
    // Do something with i  
    // Do something with i + 1  
}
```

**Loop tiling/blocking:** if doing memory accesses in loop, do unrolling to use cache effectively



# Instruction-level Parallelism

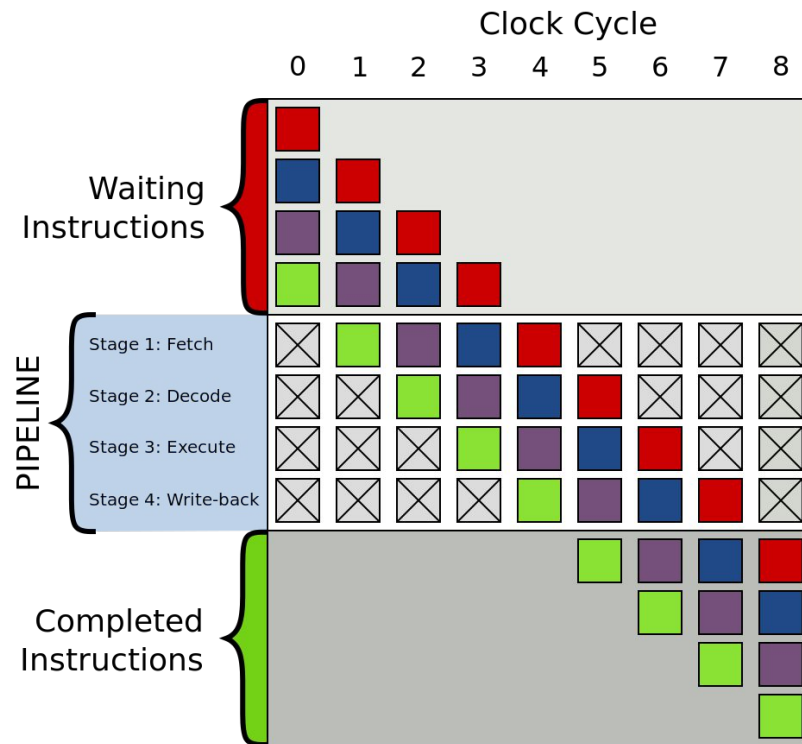
Breaking x86 instructions into lower instructions (**μops**) and executing these in parallel



# Instruction-level Parallelism

## Pipelining / Superscalar Execution

- Hardware translator turns a x86 instruction (like `addl reg1, reg2`) into the low level  $\mu$ ops needed to perform this instruction
- $\mu$ ops are then “pipelined” to be done in parallel
- Problem?
  - Killed by conditional jumps
  - How can you know where you’ll jump next, i.e. which instruction will be done next?
  - Guess! Go with what usually happens, keep track, and don’t commit results if guessed incorrectly



# Instruction-level Parallelism

## Out of Order Execution



# Instruction-level Parallelism

## Out of Order Execution

### In-Order Processor

1. Fetch the instruction
2. If inputs available, dispatch instruction to functional unit. If not, stall until available.
3. Instruction executed by functional unit
4. Functional unit writes results back to register file



# Instruction-level Parallelism

## Out of Order Execution

### In-Order Processor

1. Fetch the instruction
2. If inputs available, dispatch instruction to functional unit. If not, stall until available.
3. Instruction executed by functional unit
4. Functional unit writes results back to register file

### Out-of-Order Processor

1. Fetch the instruction
2. Dispatch instruction to instruction queue
3. Instruction waits in queue until inputs available. Allowed to leave queue before older instructions
4. When inputs available, instruction sent to functional unit and executed
5. Results queued
6. After older instructions have written their results, this instruction get to write its results



# Instruction-level Parallelism

## Out of Order Execution

- Instruction Control Unit
  - Talks to Instruction Cache
  - Fetches instructions, gives to Execution Unit
- Execution Unit
  - Is given  $\mu$ ops to perform
  - Gives  $\mu$ ops to hardware that can do it
    - Instruction either retired (commit its effect) or flushed (discarded)
  - Register renaming: using register other than named one to allow parallelism



# Practice Question

Under what circumstances will a compiler perform optimization?





# Answer

Under what circumstances will a compiler perform optimization?

The compiler will only perform optimizations when the result will behave identically to the unoptimized code.



# Practice Question

Under what circumstances will a compiler perform optimization?

The compiler will only perform optimizations when the result will behave identically to the unoptimized code.

Is this

```
*xp += *yp;
```

```
*xp += *yp;
```

the same as this?

```
*xp += 2 * (*yp);
```



# Answer

Under what circumstances will a compiler perform optimization?

The compiler will only perform optimizations when the result will behave identically to the unoptimized code.

Is this

```
*xp += *yp;
```

```
*xp += *yp;
```

the same as this?

```
*xp += 2 * (*yp);
```

Not if `xp == yp`

Then the result is `*xp += 3 * (*xp)`



# Practice Question

How can you tell if an optimization is useful in the long run?



# Answer

How can you tell if an optimization is useful in the long run?

Use Amdahl's Law!

$a$  = what fraction of your program is the hog

$k$  = hog speedup factor

$$\text{Speedup} = \frac{1}{(1 - a) + \frac{a}{k}}$$



# Answer

How can you tell if an optimization is useful in the long run?

Example:

Speedup half of your program ( $a = 0.5$ ) to be twice as fast ( $k = 2$ )

Total speedup is only 1.33!

$$\frac{1}{(1 - a) + \frac{a}{k}}$$



# Memory



# The Memory Hierarchy

- The general principle behind **caching** is that for each level  $k$ , a smaller and faster device at  $k$  serves as a cache for a larger slower device at  $k+1$ .
  - For example, DRAM is a cache to the disk, and the L1 – L3 caches are caches to DRAM.
- At the L1 cache, we often see a distinction between the L1 data and L1 instruction caches. The reasoning for this is twofold: first, as explored in ILP, we can fetch both the instruction and data in parallel. Secondly, it allows us to take advantage of ... locality!





# The Memory Hierarchy

- Cache blocks are generally stored in some format *like*:  
`[set][valid][dirty][tag][-- data --]`
  - The set and tag fields indicate the block of data being cached
  - We cache at a coarser granularity than single bytes
  - The distribution of bits to set and tag can heavily influence the likeliness that we're storing the right things in the cache. *(Although, this goes down a discussion about associativity that isn't too important for this class!)*



# The Memory Hierarchy

- Cache blocks are generally stored in some format *like*:  
[set][valid][dirty][tag][-- data --]
  - The set and tag fields indicate the block of data being cached
  - We cache at a coarser granularity than single bytes
  - The distribution of bits to set and tag can heavily influence the likeliness that we're storing the right things in the cache. *(Although, this goes down a discussion about associativity that isn't too important for this class!)*



# The Memory Hierarchy

- When we write programs, we want to take advantage of locality. This is usually expressed in two forms, spatial and temporal.
  - **Spatial locality:** accessing some data in some memory address probably means that you're going to be accessing data in nearby memory. Concretely, stride-1 reference patterns are good.
  - **Temporal locality:** you're probably going to be accessing data more than just once. So, repeated references to variables is great.



# Cache Memories

L1 Cache (2-4 cycles)

Between CPU Register File and Main Memory

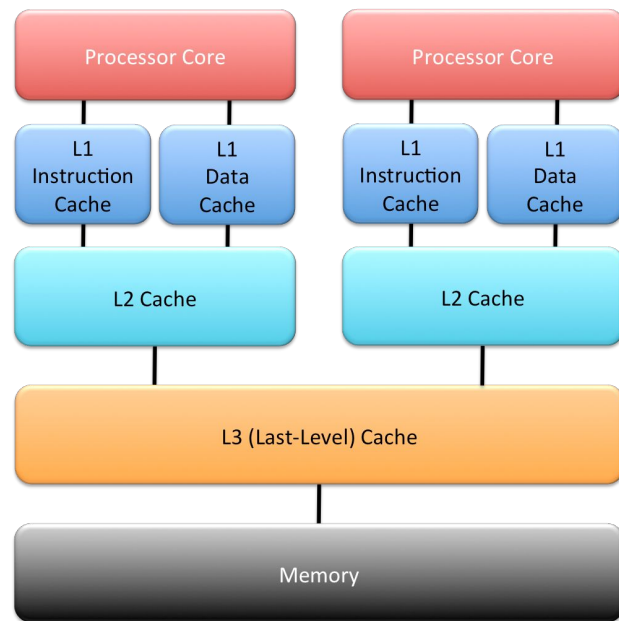
L2 Cache (10 cycles)

Between L1 Cache and Main Memory

L3 Cache (30-40 cycles)

Between L2 Cache and Main Memory

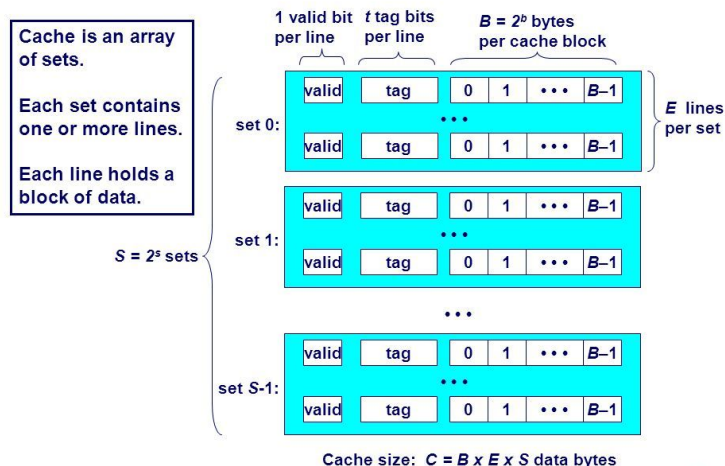
SRAM is faster than DRAM!



# Cache Memories

- General Caching Organization
  - $m$  bits for addressing:  $M = 2^m$  **addresses**
  - A cache has  $S = 2^s$  **cache sets**
  - Each set has  $E$  **cache lines**
  - Each line has 1 valid bit,  $t$  tag bits,  $B = 2^b$  bytes per **cache block**
  - Cache Size:  $C = B \times E \times S$

## General Org of a Cache Memory



# Cache Memories

- Types of Caches
  - Direct-mapped Cache
    - $E = 1$ , only one line per set
  - Set-associative Cache
    - $1 < E < C/B$ , more than one line per set
  - Fully Associative Cache
    - $E = C/B$  **AND**  $S = 1$ , only one set with  $C/B$  lines

$C$  = Cache Size

$S$  = # of Sets

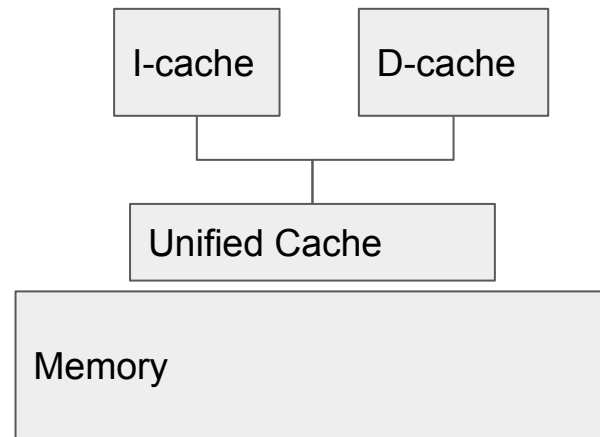
$E$  = # of Lines

$B$  = Block Size



# CPU Caches

- We can even split caches up even more!
  - i-cache
    - Only stores instructions
  - d-cache
    - Only stores data
  - Unified cache
    - Stores both instructions and data!
- This allows the processor to read instructions and data at the same time

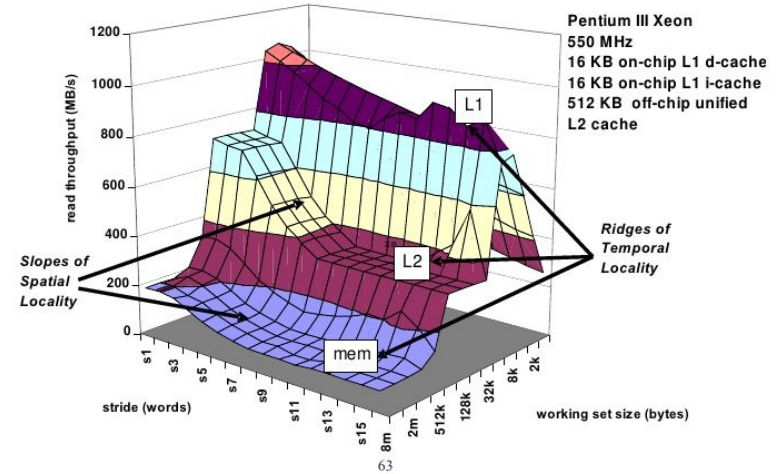


A **possible** cache hierarchy



# Cache Memories

- Performance of Caches
  - Miss Rate
    - $\# \text{misses} / \# \text{references}$
  - Hit Rate
    - $1 - \text{Miss Rate}$
  - Hit Time
    - Time to deliver word from cache to CPU
  - Miss Penalty
    - Any additional time due to a cache miss





# Cache Memories

- Write Hit Policies:
  - Write Back
    - Write to cache, and set the **Dirty Bit**
    - When data is about to be overwritten by cache eviction, send dirty data to lower level
  - Write Through
    - When write to a cache, write through to the lower level
- Write Miss Policies:
  - Write Allocate
    - Load corresponding block from lower level to cache, then write
    - Takes advantage of space locality, but misses lead to a load
  - No Write Allocate
    - Skip the cache and write directly to the lower level



# Practice Question

Discuss the tradeoffs between *write-through* and *write-back* write hit policies.



# Answer

- Under the **write-through** policy, for each write hit, write to the cache and also to the backing memory.
  - Memory and cache are always synchronized and consistent
  - However, we incur the penalty of writing to memory each time we write to cache, which negates the benefit of the cache for writes.



# Answer

- Under the **write-back** policy, for each write hit, write only to the cache but mark the cache block as dirty. We write back to memory upon eviction.
  - We benefit immensely from temporally localized programs!
  - Although we're guaranteed that we perform less or equally many writes as with the write-through policy, we run into the issue of unpredictability in our writes. For example, if we're developing a program for a nuclear plant, we might want to be sure of how long a write will take.
  - In the case that we have a program that performs many random reads, we may perform just as many writes but occupy extra space for the dirty bit.



# Concurrency and Parallelism



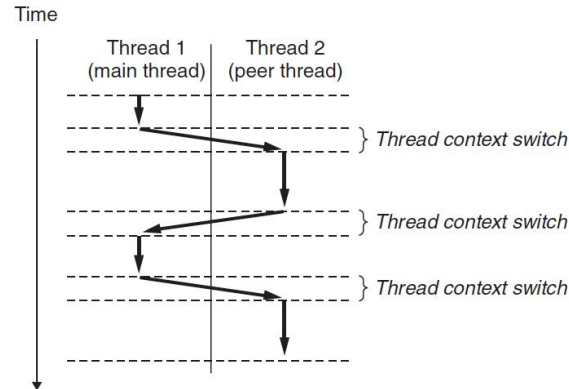
# Concurrency

- We use concurrency when we need to do multiple things at once
- We have different methods to implement concurrency
  - Processes / Threads
    - Can run one after the other or in parallel
  - I/O Multiplexing (Events)
    - Decide which event gets handled when
- Note: Processes each have their own address space, but threads share address space with other threads within a process



# Concurrency

- Threads
  - Logical flow within a process
  - Can run concurrently
    - Context switch between threads (works similarly for processes!)



# Thread Level Parallelism

- What's the difference between a process and a thread? (*This is a fairly common interview problem.*)
  - Threads are more lightweight
  - Threads share a memory space, while process have their own memory spaces.  
That said, threads maintain their own registers.
- We can use TLP to split a program, ideally an [embarrassingly parallel program](#), into independent tasks.
- Be careful about race conditions!





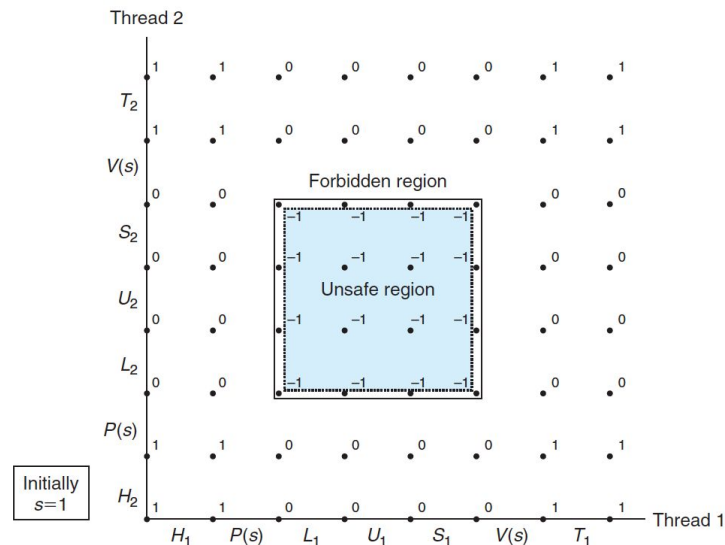
# Concurrency

- Shared Memory
  - Threads within the same process can share memory
  - Problems occur when processes / threads access the same memory
  - How can we fix this???
  - Semaphores
    - Only allow certain number of processes / threads access to shared resource at a time
  - Mutexes
    - Special version of a Semaphore that allows only **1** at a time



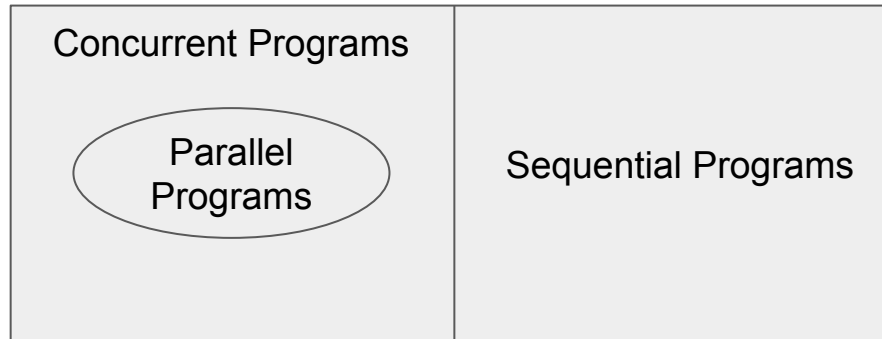
# Concurrency

- Semaphores
  - We can use semaphores (and Mutexes) as a way to maintain order
  - Essentially, they act like **locks** on the shared resource
  - This allows for **mutual exclusion**, which is good!

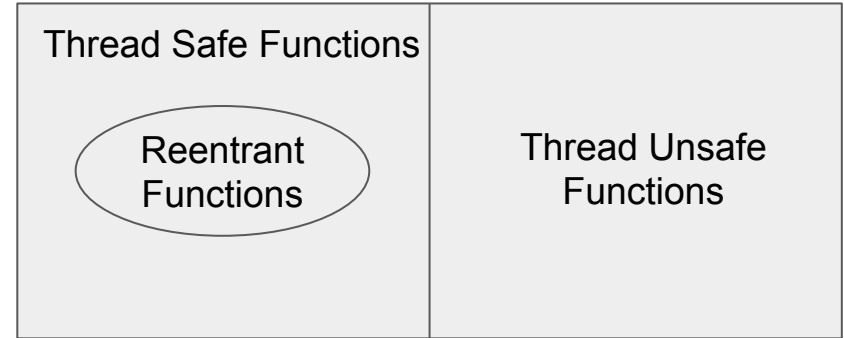


# Concurrency

- Parallelism
  - Allowing multiple processes / threads to run at the same time
  - But how is this different than concurrency??
    - It is a special type of concurrency



# Concurrency

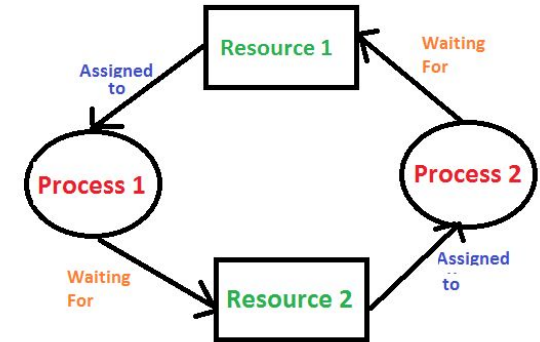
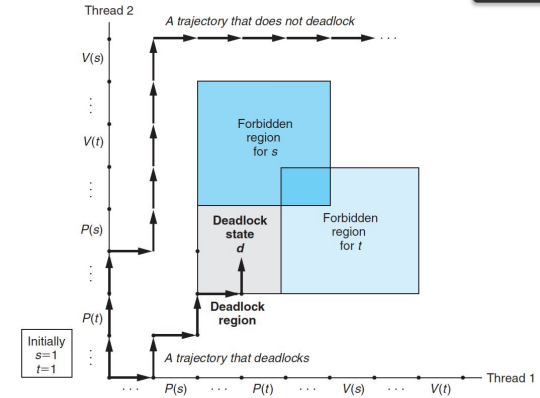


- Thread Safety
  - Types of Thread Unsafe Functions
    - Functions that don't protect shared variables
    - Functions that keep state over multiple invocations
    - Functions that return a pointer to a static variable
    - Functions that call Thread Unsafe Functions
  - Reentrancy
    - When a function does not access any shared memory



# Concurrency

- Problems with Concurrency
  - Race Conditions
    - Correct output depends on order of execution of threads
    - We can prevent these by using semaphores + mutexes!!
  - Deadlock
    - Multiple processes / threads are waiting for conditions that will never actually become true



# Deadlock

There are four conditions necessary for deadlock

1. **Mutual Exclusive Access** - only one thread can access resource
2. **Incremental Allocation** - Threads are able to grab locks one at a time and hold them while waiting for other locks.
3. **No Preemption** - No stealing locks from other threads
4. **Circular Waiting** - There is a cycle in resource allocation graph.



# Synchronization

- Semaphores allow us to enforce the number of threads that can enter a section of code. *(They're essentially glorified counters!)*
  - When the counter is nonzero, the door is open and the thread can pass.
  - When the counter is zero, the door is closed.
- We rely on three primitives to control the value of the counter: `sem_init`, `sem_wait`, and `sem_post`.



# Synchronization

- One way to look at the semaphore is to liken it to a doorman. The function that initializes the semaphore is the host, while the threads attempting to enter the critical section are the guests.
  - We begin with `sem_init` informing the the semaphore how many guests should be allowed;
  - The guests subsequently `sem_wait` until there is available capacity, which allows them to check in and enter the critical section; and
  - The guests check out using `sem_post`.





# Synchronization

The primitives are declared as follows:

```
int sem_init(sem_t *s, int pshared, unsigned int value);  
int sem_wait(sem_t *s); // Waits until  $s > 0$ . When this happens,  $s$   
                        // is decremented and 0 is returned.  
int sem_post(sem_t *s); // Increments  $s$  by one.
```



# Practice Question

Suppose you want a thread T1 to wait until thread T2 finishes, and that T2 is detached. Explain how to arrange for this reliably, assuming the threads cooperate by executing code that you specify. Your code may invoke any of the `pthread_*` or `sem_*` primitives discussed in the book or in class. Explain any assumptions you make and any race conditions that you couldn't fix.



# Hint

Suppose you want a thread T1 to wait until thread T2 finishes, and that T2 is detached. Explain how to arrange for this reliably, assuming the threads cooperate by executing code that you specify. Your code may invoke any of the `pthread_*` or `sem_*` primitives discussed in the book or in class. Explain any assumptions you make and any race conditions that you couldn't fix.

## Can't we just use `pthread_join`?

No, because T2 is detached! Instead, we need rely on semaphores (or more specifically, mutexes) to solve this problem. More specifically, we need to implement `pthread_join` using semaphores.



# Answer

Here's an idea: what if we enforce an order on the execution of critical sections?

```
void t1 {  
    sem_wait(&mutex);  
    // Do work here ...  
    sem_post(&mutex);  
}
```



# Answer

Here's an idea: what if we enforce an order on the execution of critical sections?

```
void t2 {  
    // Do work here ...  
    sem_post(&mutex);  
}
```



# Answer

```
sem_t mutex;  
sem_init(&mutex,  
        0 /* pshared */,  
        0 /* value */);  
  
void t1 {  
    sem_wait(&mutex);  
    // Do work here ...  
    sem_post(&mutex);  
}
```

```
void t2 {  
    // Do work here ...  
    sem_post(&mutex);  
}
```

In this solution, we take initialize mutex to zero such that t1's critical section is only "unlocked" after t2 completes its work. The race condition that we potentially encounter is that t1 can begin doing its work before t2 entirely terminates.



# Linking

- As the name suggests, this is the idea of bringing different parts of a program into one coherent executable.
- There are two types of linking to be aware of:
  - **Static Linking** - The linking is done at compile time through the use of static libraries (archive files - .a extension)
  - **Dynamic Linking/Loading** - The linking is done at run time through the use of shared libraries (shared object files - .so extension)



# Static Linking

- During the pre-processing stage the code not defined in the modules will be placed as symbol stubs, that the linker will resolve during compile time.
- The assembler creates a **relocatable object file** to help associate all the data for the creation of the final executable. Many formats exist - most popular is ELF.
- **Symbol Resolution** - Associate the stub with the function/variable
- **Relocation** - Find the address of symbol in the actual library and replace stub with it.
- Remember there are three types of symbols - based on how the variables/functions are defined (global, local and static):
  - Globals used to be marked as *weak* and *strong* if they had the same name, based on the data type.
  - **Common Storage Allocation** - Just make all symbols strong and reference them by their modules.





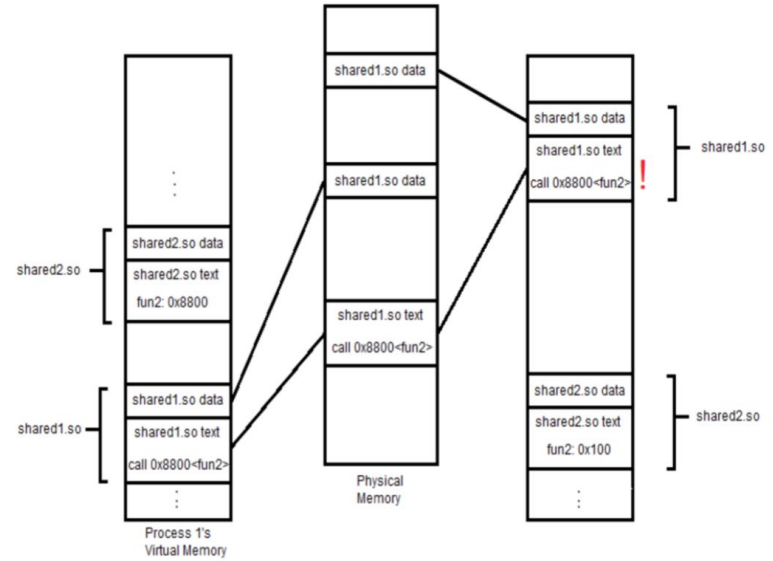
# Static Linking

- With Relocation we have four things to consider:
  - Offset into the module
  - Type of relocation required.
  - Index of this stub within the symbol table
  - Addend - the actual address offset within the module. This is of two types:
    - **Program Counter Relative Relocation** - Based on position of %rip from the next instruction.
    - **Absolute Relocation** - Relocation to the address itself.
- Static linking produces large executables since they contain all the data, but this is more portable. If a library is updated, the entire executable has to be re-linked.



# Dynamic Linking

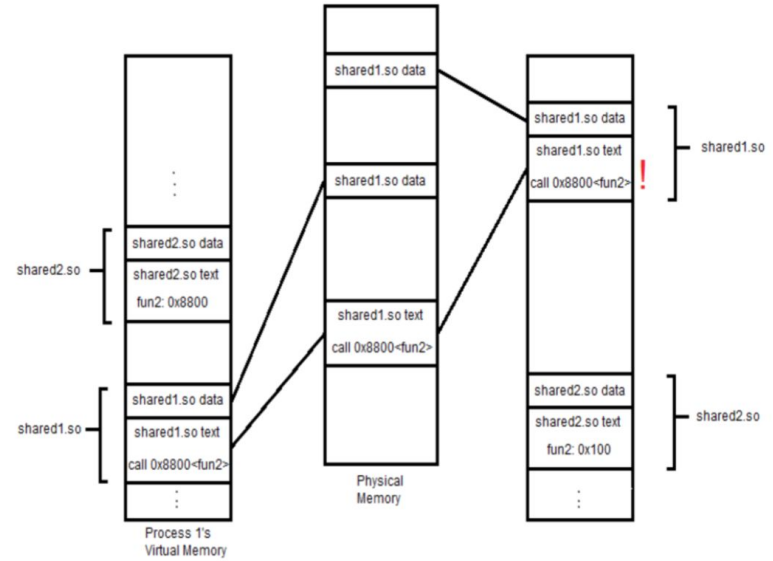
- Use one shared object file (.so) that has placeholders all over made by the linker
  - while running or loading, fill in placeholders with the necessary addresses through symbol resolution and relocation.
  - this **Shared Object** will be shared between different processes and so is a lot more effective.



# Dynamic Linking

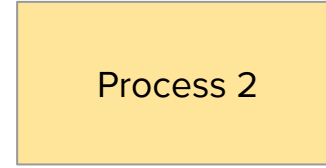
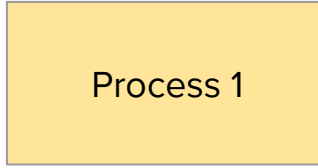
## Advantages and Disadvantages

- Compile time is sped up but run time is slower
- One break could cause more damage than with a static library.



# Memory Virtualization





- We have  $> 1$  process, but only 1 memory
- How do we divide this up?



Memory Part 1

Memory Part 2

Process 1

Process 2

- We could try to split up the memory beforehand.



Memory Part 1

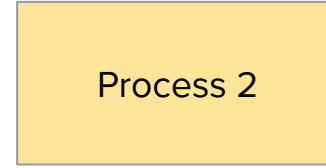
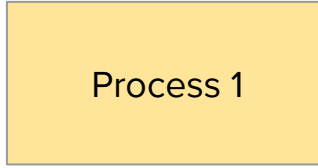
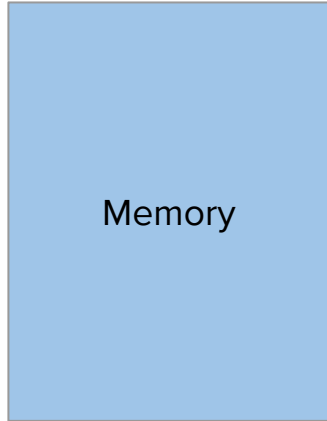
Memory Part 2

Process 1

Process 2

- But this can cause some problems.
- What if we add another process?
- How do we do memory access via pointers if we don't know which addresses belong to us until we start the program?

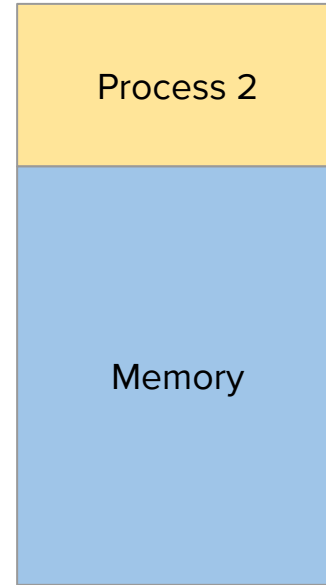
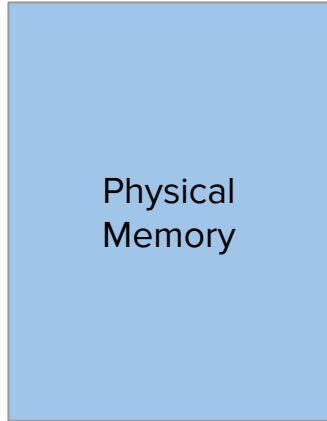


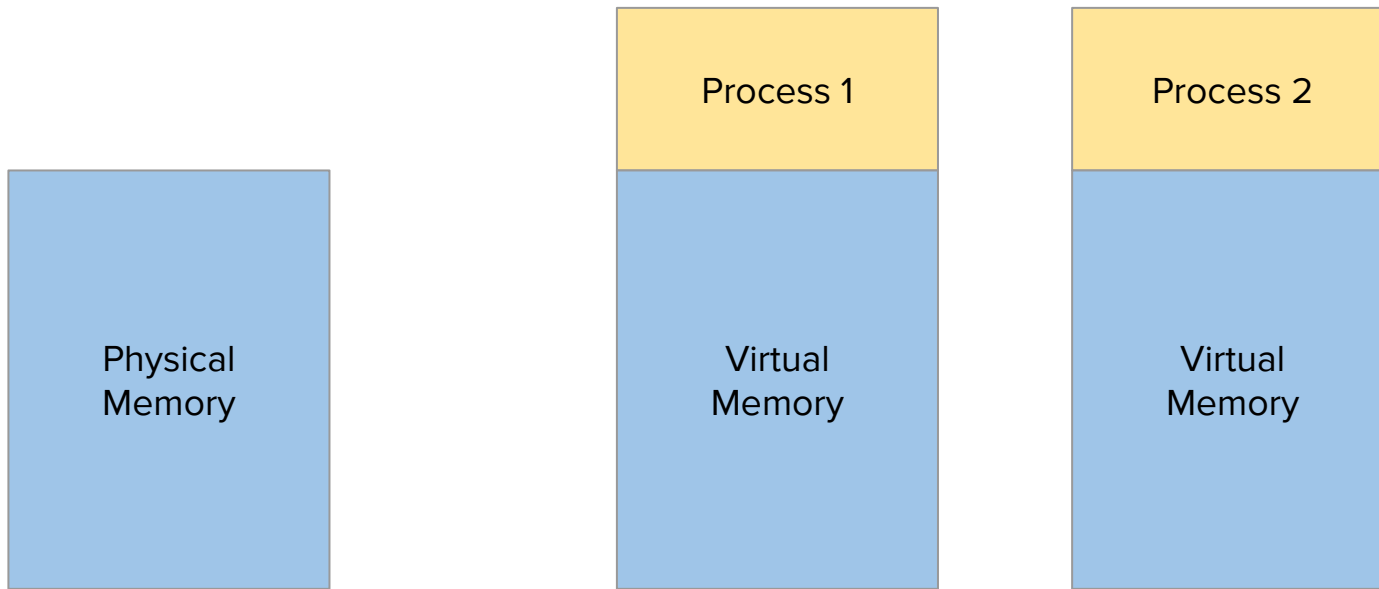


- Instead we generally try to “trick” each process into thinking it has ALL the memory.
- So let’s pretend each process has its own copy of all the memory



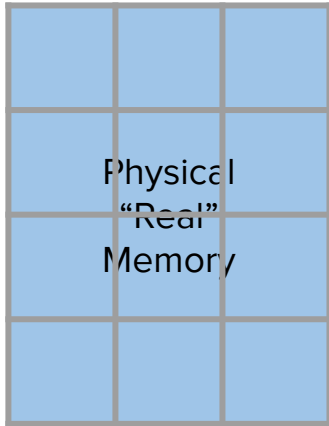




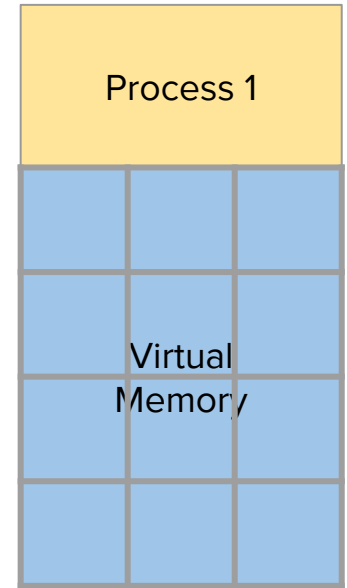


- We'll call this virtual memory



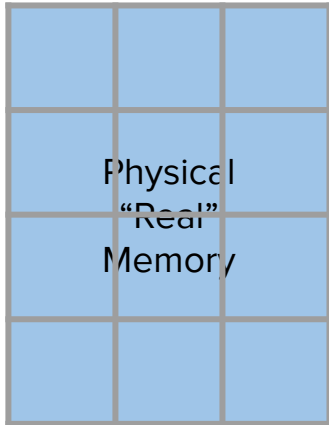


- Let's divide physical memory into **frames**

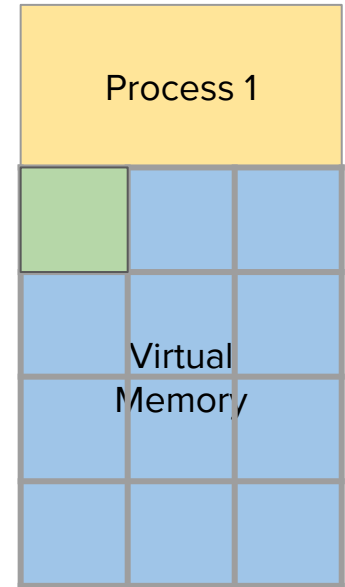


- Let's divide virtual memory into **pages** (chunks of memory)





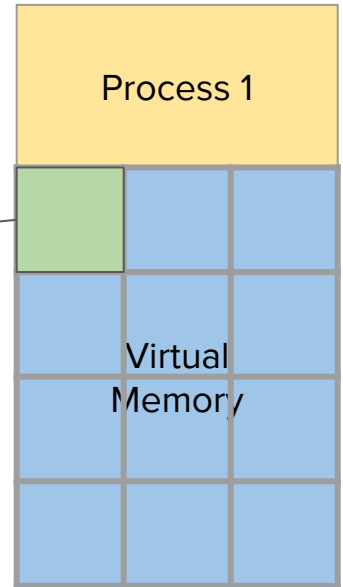
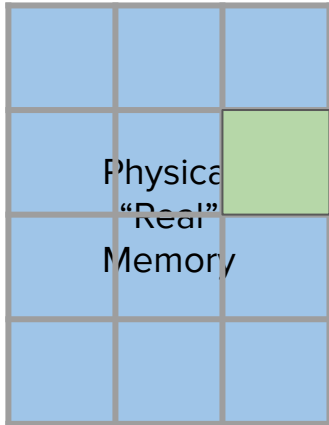
- Let's say this page is needed.



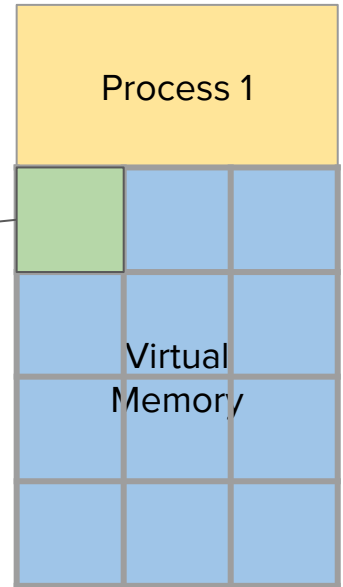
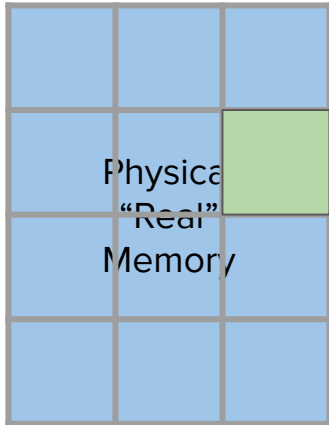
- Each frame/page will have an address



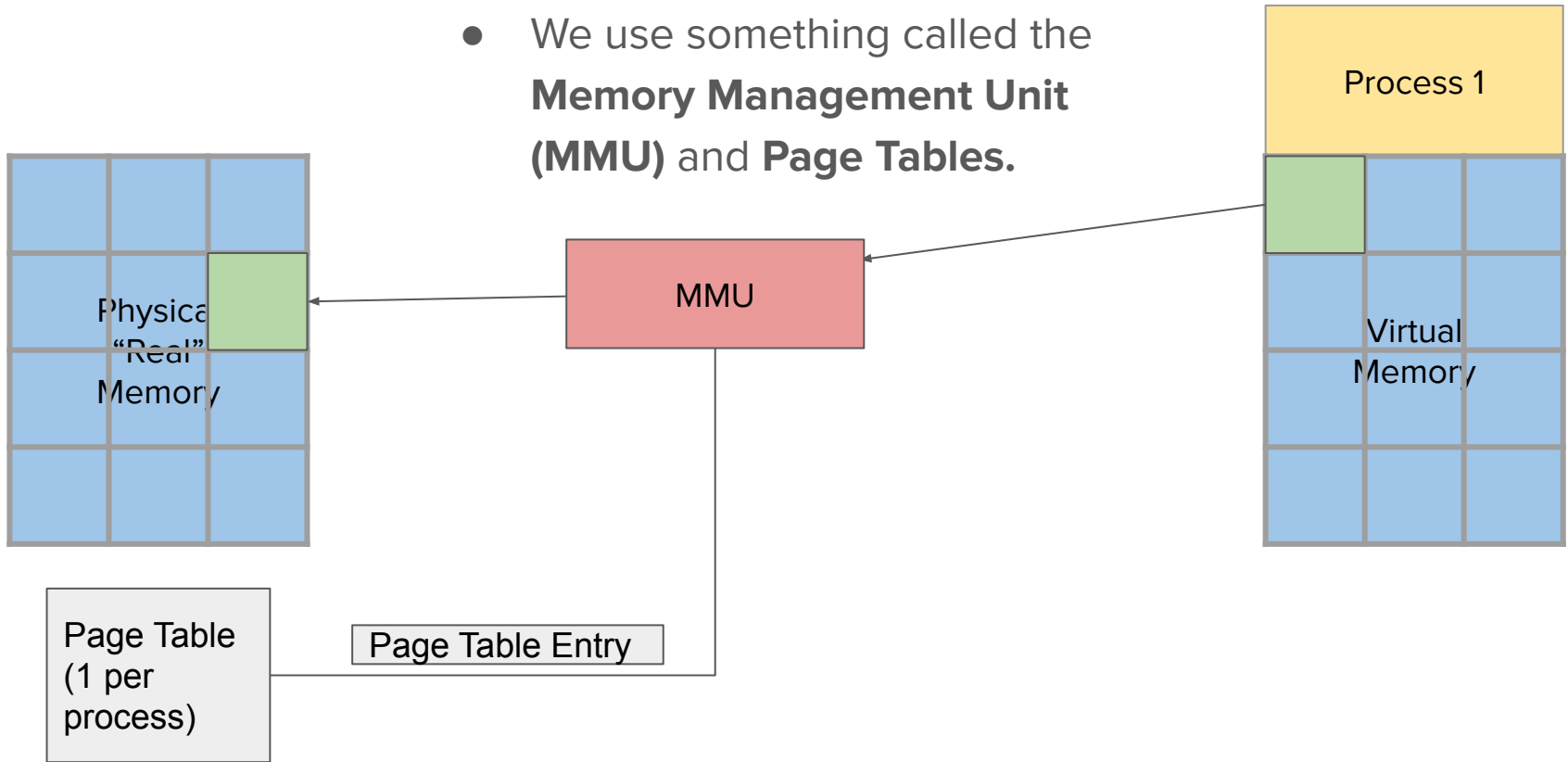
- We can then write this page into any frame.



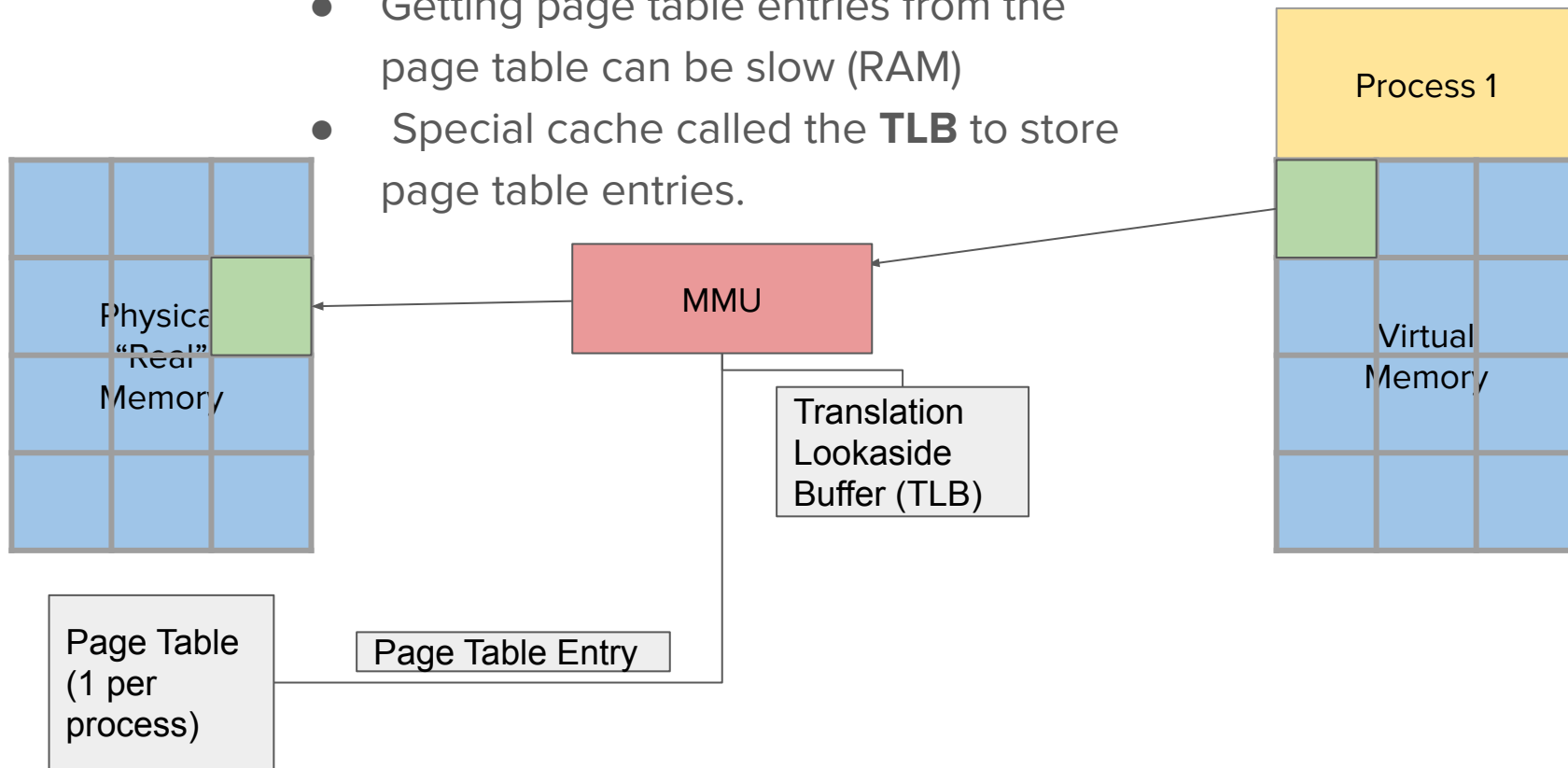
- But how does this translation process work?



- We use something called the **Memory Management Unit (MMU)** and **Page Tables**.



- Getting page table entries from the page table can be slow (RAM)
- Special cache called the **TLB** to store page table entries.





# Page Tables

- What is in a page table?
  - a. Index = Virtual Page Number
  - b. Entry = Page Table Entry
  - c. Page Table entry contains Physical Page Number + extra information (valid, dirty bits, etc.)
- Hierarchical Page Tables
  - a. Instead of giving Physical Page Number, gives address to another page table (allows us to store more page table entries).



# Memory Virtualization Terminology

- Page Hit - Page in physical memory
- Page Fault - Page not in physical memory. Raises Exception to be handled by OS.
- TLB Miss - PTE not in TLB, must consult Page Table
- Physical Page/Frame Number (PPN/PFN) - upper bits of physical address
- Virtual Page Number (VPN) - upper bits of virtual address
- Physical Page Offset = Virtual Page Offset = lower bits of both physical and virtual addresses. Same length for both.

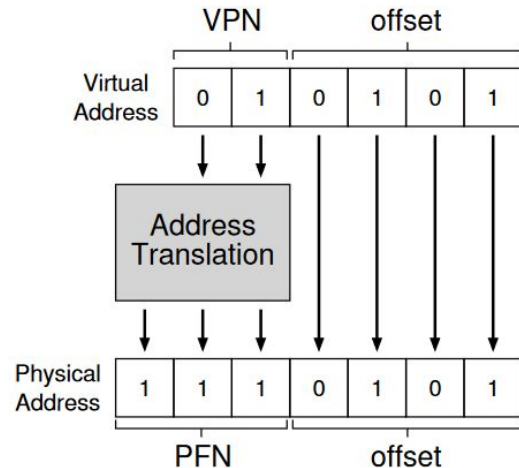


Figure 18.3: The Address Translation Process

Source: Operating Systems: 3 Easy Principles Chapter 18



# Memory Virtualization

- Turn your itty-bitty memory into an infinite space to work with, through the use of *virtual addresses*, *virtual page numbers*, *physical page numbers*, *pages* and the **Translation Lookaside Buffer (TLB)**.
- Based on the size of the system (64 or 32 bit systems) the address we usually work with is broken into a VPN (Offset into the Per Process Page Table) and VPO (size of each Page).
- A Page is just a fixed chunk of memory we can take. So essentially everything on our system is broken up into a bunch of pages. Each process has a page table that maps each of the pages to physical memory.



# Memory Virtualization

- The Page table is indexed through something known as the PTE (Page Table Entry) and the addresses in this table are broken into a PPO (same offset as VPO into the page in memory) and PPN (the reference to the page in physical memory).
- The Page table is actually stored within memory itself, so accessing the Page Table each time over is not super efficient. This is why the TLB exists.
- This is part of the MMU in the CPU itself
- Based on the scheme used (most likely LRU) the TLB contains the most recently accessed PTEs for fast access into Physical memory.
- **TLB Miss** - Go find the PTE based on the VPN in memory and update the cache and try again.
- **TLB Hit** - Yay go to that memory location.
- Every page also contains other information such as whether they are valid (allocated yet) or not. Accessing an invalid page will raise an exception into page fault handler.
- Pages are also managed into disk, but this is a more involved process.



# Exceptions

- When something unexpected happens, stop control flow and try to deal with the error.
- Update ERRNO in UNIX systems to give more information (it is thread local).
- Upon the operation that *raises* the exception, the PC (along with some other stuff) is saved and the mode changes to kernel mode to execute a trap/Interrupt handler.
- The handler will see what kind of error was set and decide how to proceed. It may choose to abort (everything is lost!) or halt (recoverable) execution.
- Exceptions are hardware based, we have something similar in software called Signals (where power is given to the user to define a handler).



# Types of Exceptions

- Asynchronous Exceptions

- Ex . I/O Interrupt

To be totally honest, this terminology is not super consistent.

- Synchronous Exceptions

- Traps

- Basically, means for application (software) to transfer control to OS
- Stuff like System Calls, Breakpoints

- Faults

- Unintentional, but recoverable
- Page Faults, Protection Faults, Floating point exceptions

- Abort

- Unintentional. Cannot be recovered.



# Good luck!

Sign-in <https://tinyurl.com/cs33finals22signin>

Slides <https://tinyurl.com/cs33finals22>

Feedback <https://tinyurl.com/uclaupetutoringfeedback>

Practice <https://github.com/uclaupe-tutoring/practice-problems/wiki>

## Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: <https://upe.seas.ucla.edu/tutoring/>
- You can also post on the Facebook event page.



**Only briefly covered in lecture**





# I/O

- This is to do with communication with devices, wherein the code to do so is defined within device drivers.
- These devices are connected through the BUS - Northbridge in Intel connects CPU to RAM and southbridge connects it to different peripherals.
- Sharing a BUS between devices increases efficiency and transparency through bus snooping but **bus arbitration** can cause contention between multiple devices, which would require some scheduling mechanism to resolve.
- Commands are defined in device drivers that moves down to the lower level and is implemented as either **memory mapped I/O** or **special assembly**.



# I/O

- Once an I/O operation has been requested the process can do one of two things:
  - a. **Polling** - Keep asking device if its done yet.
  - b. **Asynchronous Interrupts** - Send an interrupt when the device is done.
- To reduce overhead between switches from CPU to memory to device we have **Direct Memory Access** where the device is given access to the memory itself and operates independently to the CPU.

