CSE150
Operating Systems
Lecture 24&25

Course Review

Final exam

- **Credit: 31%**
- In-person,
- Close book
- 3 hours.
- Chapter 1–7, 8-9, 11-13 and 17
- Lecture slides 1- 25.
- No peeking or talking – **Automatic 0 and being reported!!!**

2^32=???   4294967296

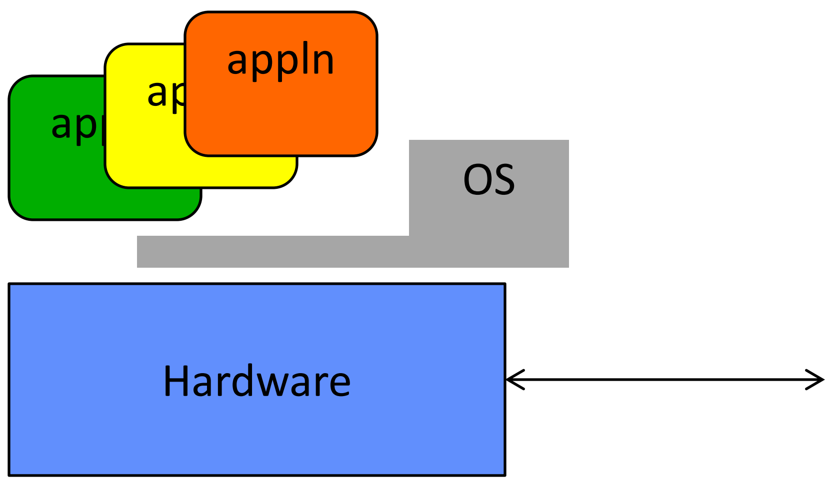
No need!!!

Purpose of this review

- Not to cover all knowledge points
- Form what we have learnt into a story.
  - Why people did it?
  - How people did it?

What is an Operating System?

- Special layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security and authentication
  - Communication amongst logical entities



Four Fundamental OS Concepts

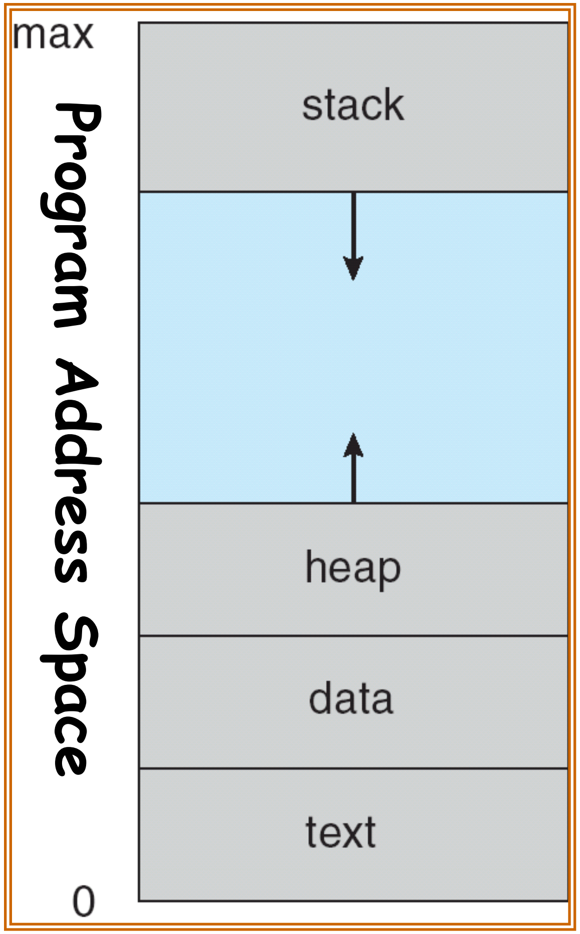
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Address space (with translation)
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Thread
  - Single unique execution context: fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- Dual mode operation / Protection
  - Only the “system” has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

First OS Concept: Process

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Why processes?
  - Protected from each other!
  - OS Protected from them
  - Processes provides memory protection
  - Threads more efficient than processes (later)
- Fundamental tradeoff between protection and efficiency
  - Communication easier *within* a process
  - Communication harder *between* processes
- Application instance consists of one or more processes

Second OS Concept: Program's Address Space

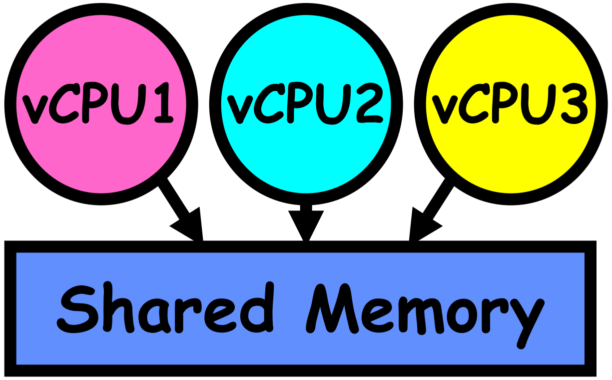
- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses

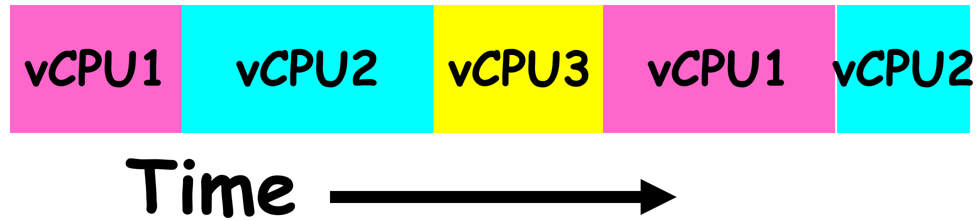


Second OS Concept: Thread of Control

- Thread: Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- A thread is executing on a processor when it is resident in the processor registers.
- PC register holds the address of executing instruction in the thread
- Certain registers hold the *context* of thread
  - Stack pointer holds the address of the top of stack
    - » Other conventions: Frame pointer, Heap pointer, Data
  - May be defined by the instruction set architecture or by compiler conventions
- Registers hold the root state of the thread.
  - The rest is "in memory"

How can we give the illusion of multiple processors?





- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

How to protect threads from one another?

- Need three important things:
  1. Protection of memory
     - » Every task does not have access to all memory
  2. Protection of I/O devices
     - » Every task does not have access to every device
  3. Protection of Access to Processor:
     Preemptive switching from task to task
     - » Use of timer
     - » Must not be possible to disable timer from usercode

Why Processes & Threads?

Goals:

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don't want a bad application to crash system!

Solution:

Process: unit of execution and allocation

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

Challenge:

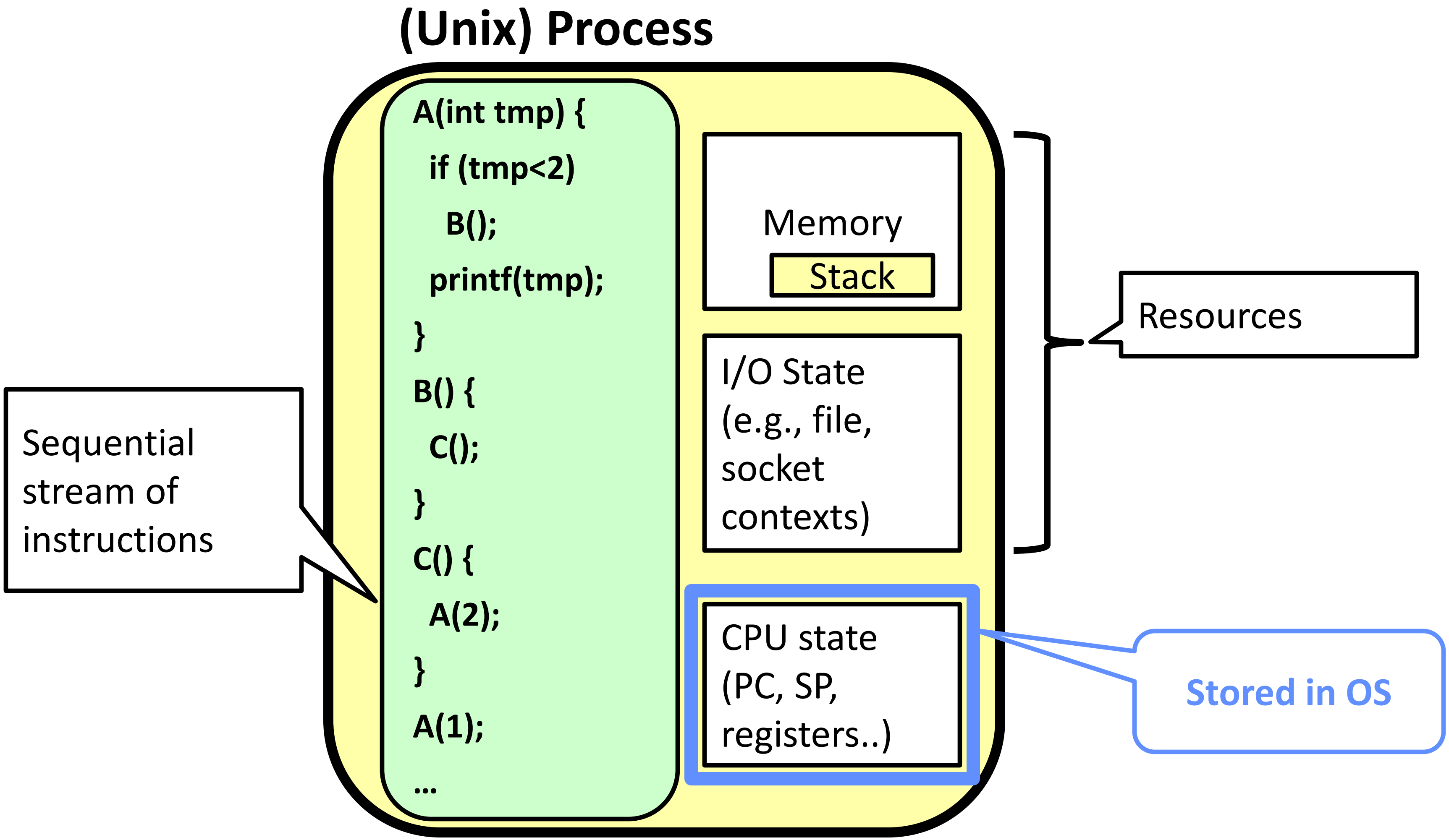
- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

Solution:

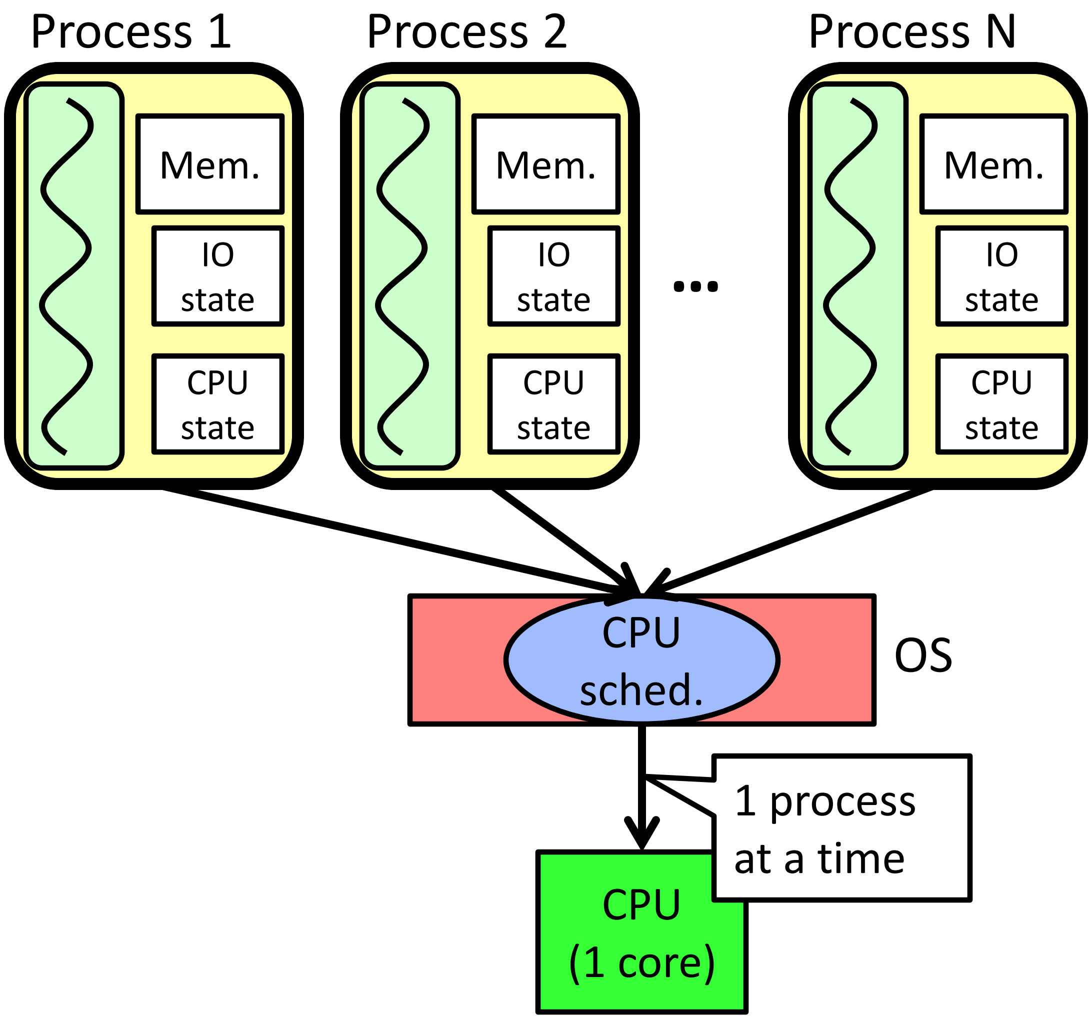
Thread: Decouple allocation and execution

- Run multiple threads within same process

Putting it together: Process

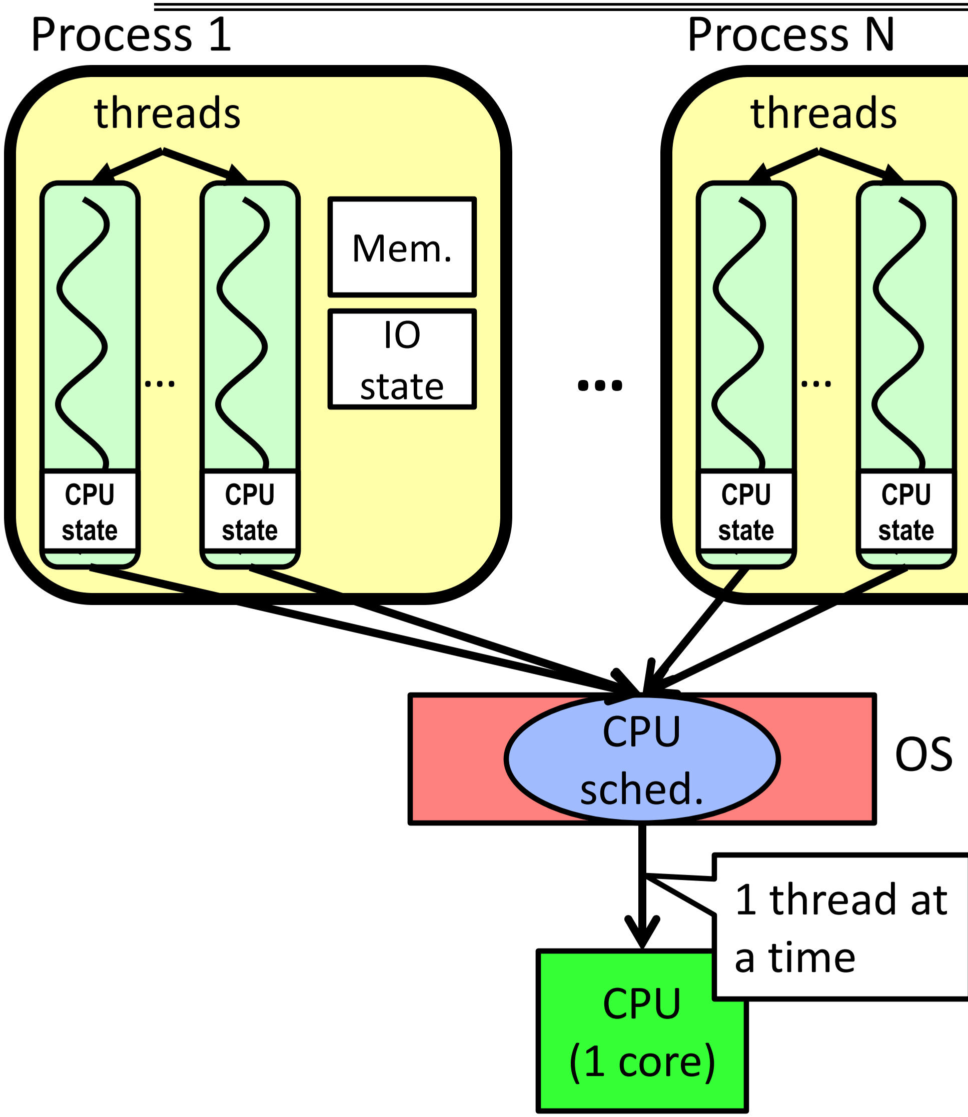


Putting it together: Processes



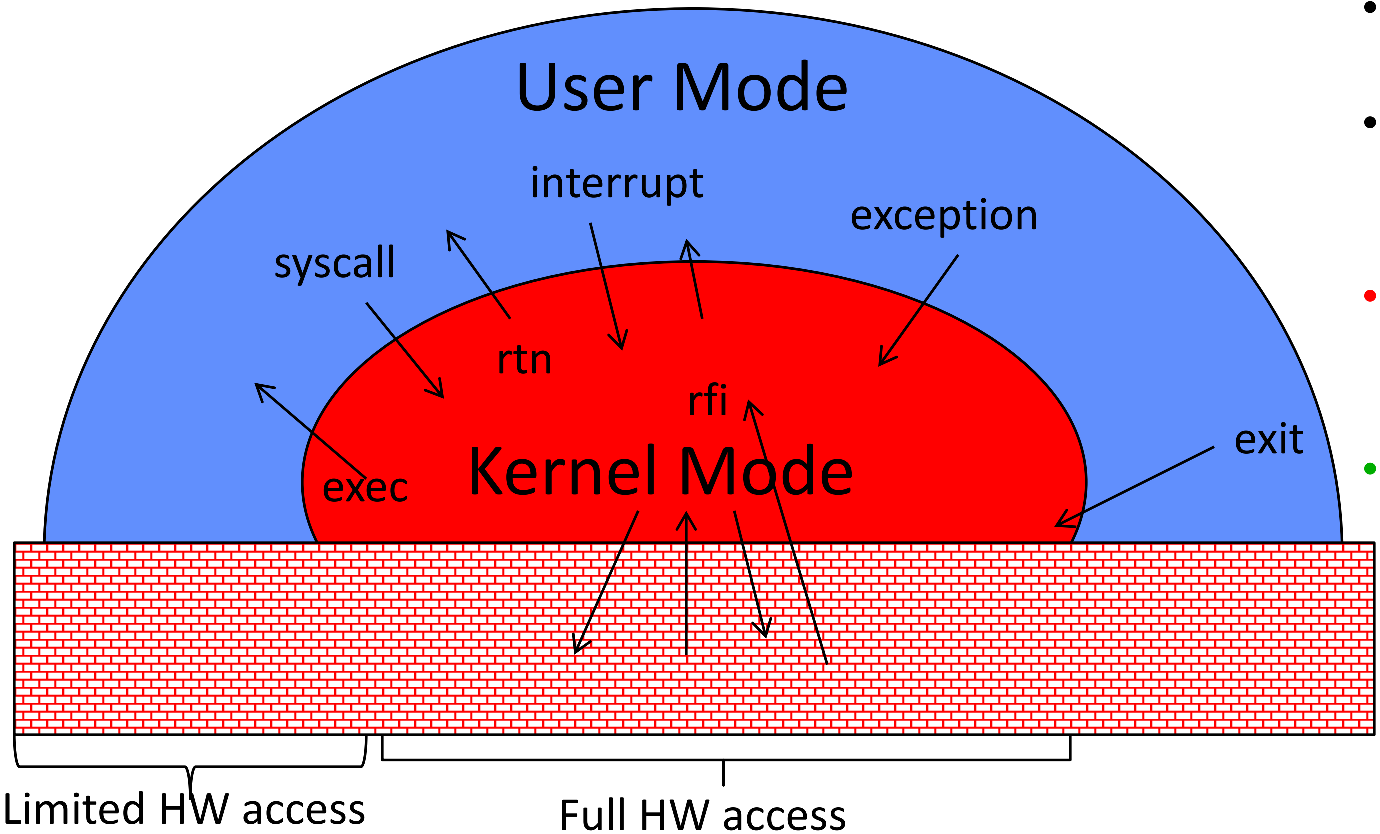
- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high (involves at least a context switch)

Putting it together: Threads



- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Fourth OS Concept: Dual Mode Operation



- A user/system mode bit
- Certain operations only permitted in system/kernel mode
- User→Kernel: *sets* system mode AND saves the user PC
- Kernel→User: clears system mode AND restores appropriate user PC

- 3 types of mode transfer (UNPROGRAMMED CONTROL TRANSFER)
  - Syscall: Process requests a system service (e.g., exit)
  - Interrupt: External asynchronous event triggers context switch (e.g., Timer, I/O device)
  - Trap or Exception: Internal synchronous event in process triggers context switch, (e.g., segmentation fault, Divide by zero)

Concurrency and Multithreading

Thread State

- State shared by all threads in process/address space
  – Content of memory (global variables, heap)
  – I/O state (file system, network connections, etc)

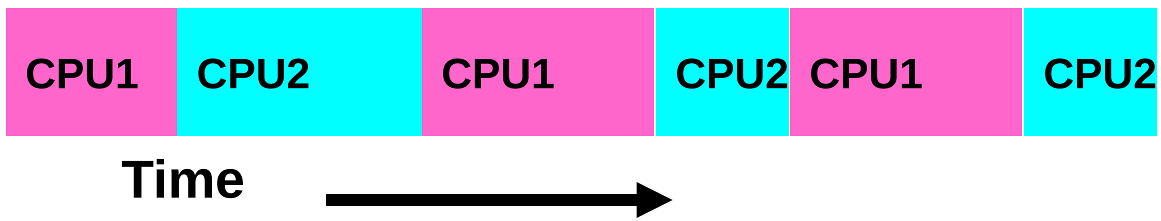
- State “private” to each thread
  – Kept in TCB ≡ Thread Control Block
  – Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  – Scheduling info: state (more later), priority, CPU time
  – Various Pointers (for implementing scheduling queues)
  – Pointer to enclosing process (PCB)
  – Etcetera (add stuff as you find a need)

Use of Threads

- A program with Threads:

```
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- Why use threads here?
- What does `CreateThread` do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does

Running a thread

- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

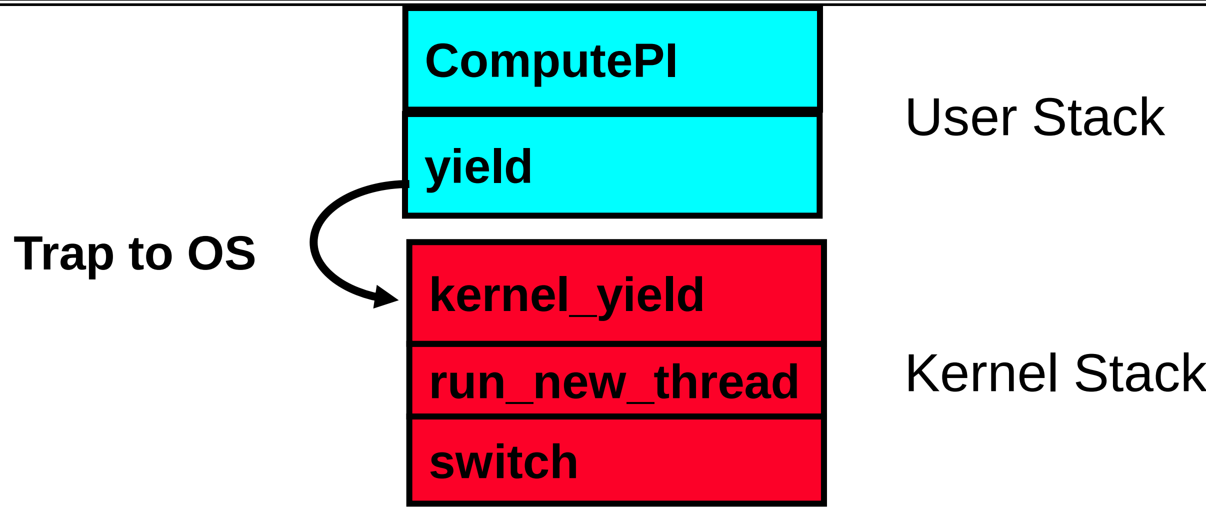
Yielding through Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU (e.g., printf)
- Waiting on a "signal" from other thread (e.g., join)
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

  - Note that `yield()` must be called by programmer frequently enough!

Stack for Yielding a Thread



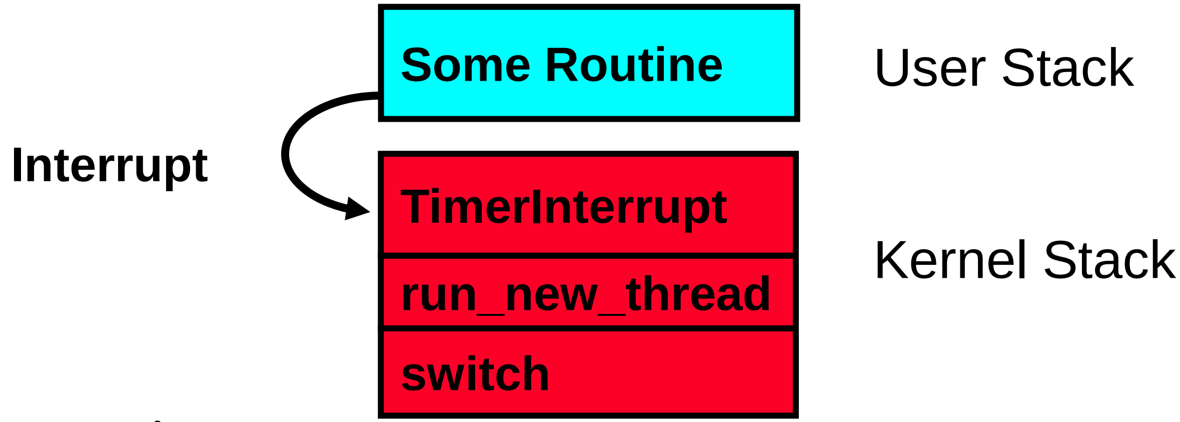
- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* deallocates finished
                             threads */
}
```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, SP
  - Maintain isolation for each thread

Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- This is often called preemptive multithreading, since threads are preempted for better scheduling
  – Solves problem of user who doesn't insert `yield();`

Synchronization

Synchronization problem with Threads

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=13

Preemption can occur at any time!

Synchronization problem with Threads

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
	y = 2;
	y = y*2;
x = 1;	
x = y+1;	

x=5

Preemption can occur at any time!

Synchronization problem with Threads

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
	y = 2;
x = 1;	
x = y+1;	
	y= y*2;

x=3

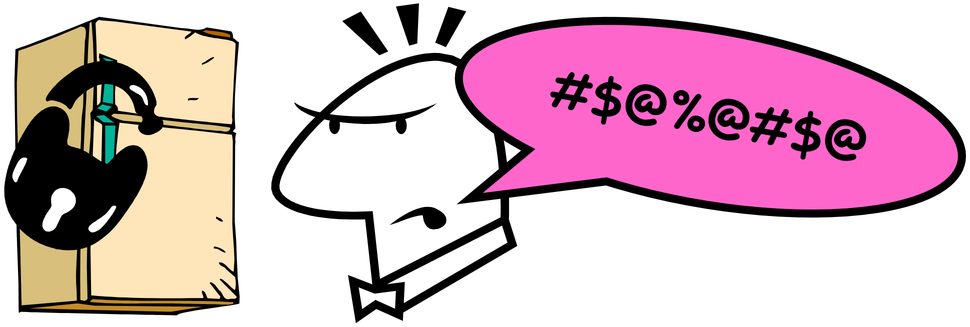
Preemption can occur at any time!

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We will show that it is hard to build anything useful with only atomic reads and writes
- Critical Section: piece of code that only one thread can execute at once.
- Mutual Exclusion: ensuring that only one thread executes a critical section
  - One thread *excludes* the other while doing its task
  - Critical section and mutual exclusion are two ways of describing the same thing.

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting



Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Motivation: "Too much milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Too Much Milk: Correctness Properties

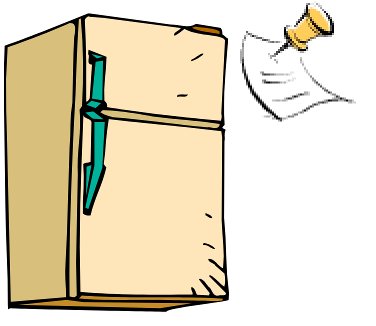
- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down ***desired*** behavior first
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the "Too much milk" problem?
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)

- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove note;
    }
}
```



- Result?

Too Much Milk: Solution #1

- Still too much milk but only occasionally!

```
   Thread A                      Thread B
 if (noMilk)
   if (noNote) {
                              if (noMilk)
                                if (noNote) {
     leave Note;
     buy milk;
     remove note;
   }
 }
                                  leave Note;
                                  buy milk;
                                  …
```

- Thread can get context switched after checking milk and note but before leaving note!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the thread dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
   }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

<u>Thread A</u>

```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

<u>Thread B</u>

```
leave note B;
if (noNote A) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work?

Too Much Milk Solution #2

- Possible for neither thread to buy milk!

Thread A	Thread B
`leave note A;`	

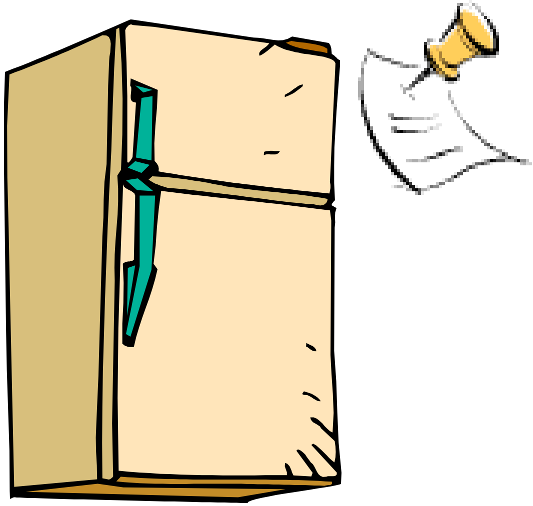
	`leave note B;`
	`if (noNote A) {`
	`    if (noMilk) {`
	`        buy Milk;`
	`    }`
	`}`

`if (noNote B) {`
`    if (noMilk) {`
`        buy Milk;`
`        …`	

	`remove note B;`

- Really insidious:
  - Unlikely that this would happen, but will at worse possible time

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  – It is safe to buy, or
  – Other will buy, ok to quit
- At X:
  – if no note B, safe for A to buy,
  – otherwise wait to find out what will happen
- At Y:
  – if no note A, safe for B to buy
  – Otherwise, A is either buying or waiting for B to quit

Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's a better way

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock

- Then, our milk problem is easy:

```
milklock.Acquire();
if (noMilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"

Locks by Disabling Interrupts

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep();
    // Enable interrupts?
  } else {
    value = 1;
  }
  enable interrupts;
}
```

```
Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

s using test&set

```
test&set (&address) {
  result = M[address];
  M[address] = 1;
  return result;
}
```

```
int value = 0;
Acquire() {
  while(test&set(value));
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
  value = 0;
}
```

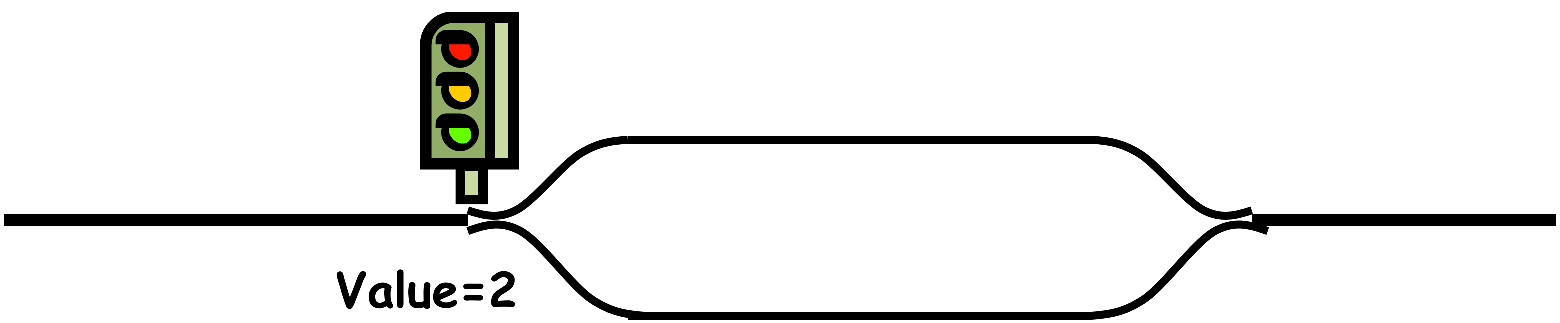
```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  guard = 0;
}
```

Threads waiting to enter critical section busy-wait

Semaphores

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Similar to unix wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » Similar to unix signal() operation
  - Only time we can set integer value directly is during initialization
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Full Solution to Bounded Buffer

```
Semaphore fullSlots = 0;  // Initially, no coke
Semaphore emptySlots = bufSize;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptySlots.P();       // Wait until space
    mutex.P();            // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();        // Tell consumers there is
                          // more coke
}

Consumer() {
    fullSlots.P();        // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();       // tell producer need more
    return item;
}
```

Monitor

- Semaphores are confusing because dual purpose:
  - Both mutual exclusion and scheduling constraints
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
- Lock: provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Condition Variables

- Condition variable: a variable `x` that implements:
  - `x.wait()`: Wait on a condition (go to sleep?)
  - `x.signal()`: Wake up one waiter, if any
  - Many threads can call `x.wait()`, they will be queued up waiting for a call to `x.signal()`. That call will start the first waiting thread.

- To support sleeping while waiting inside critical section, we add
  - `x.wait(&lock)`: Atomically release the lock and go to sleep. Re-acquire lock later, before returning.
- Some systems also implement:
  - `broadcast()` to wake up all waiting threads

- Rule: Must hold lock when doing condition variable operations

Monitor Example (with condition variable)

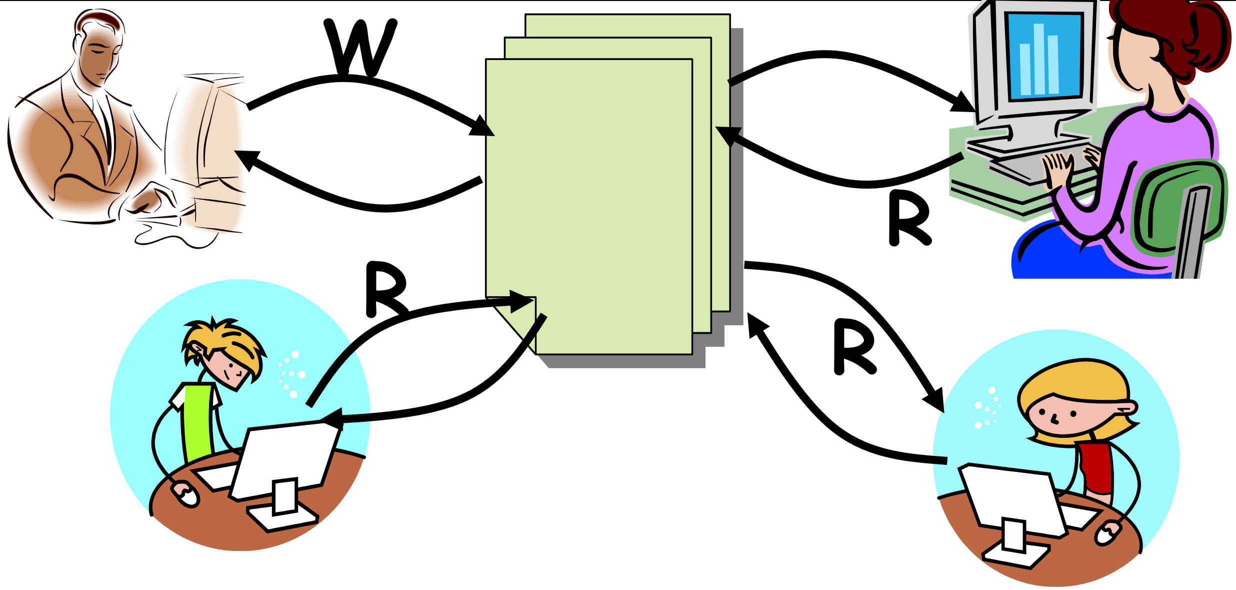
- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;                    // shared data


AddToQueue(item) {
   lock.Acquire();              // Get Lock
   queue.enqueue(item);         // Add item
   dataready.signal();          // Signal any waiters
   lock.Release();              // Release Lock
}


RemoveFromQueue() {
   lock.Acquire();              // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();   // Get next item
   lock.Release();              // Release Lock
   return(item);
}
```

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - `Reader()`
    ```
    Wait until no writers
    Access database
    Check out – wake up a waiting writer
    ```
  - `Writer()`
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called “lock”):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    okToWrite.signal();   // Wake up one writer
  lock.Release();
}
```

Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    okToWrite.signal();   // Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    okToWrite.signal();   // Wake up one writer
  lock.Release();
}
```

Deadlock

Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

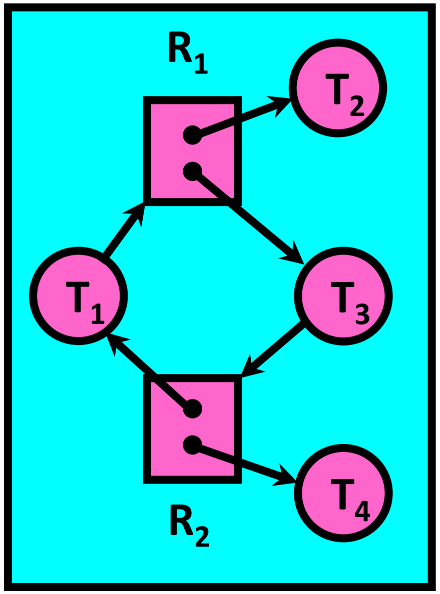
Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

    [FreeResources]: Current free resources each type
    [Request$_X$]: Current requests from thread X
    [Alloc$_X$]: Current resources held by thread X

  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

Banker's Algorithm for Preventing Deadlock

- Banker's algorithm:
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
      ($[Max_{node}]$-$[Alloc_{node}] \leq [Avail]$) for
      ($[Request_{node}] \leq [Avail]$)
      Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Scheduling

selecting a process from the ready queue and allocating the CPU to it.

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
- Maximize Throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

Scheduling

- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)

Scheduling (cont'd)

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair

- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

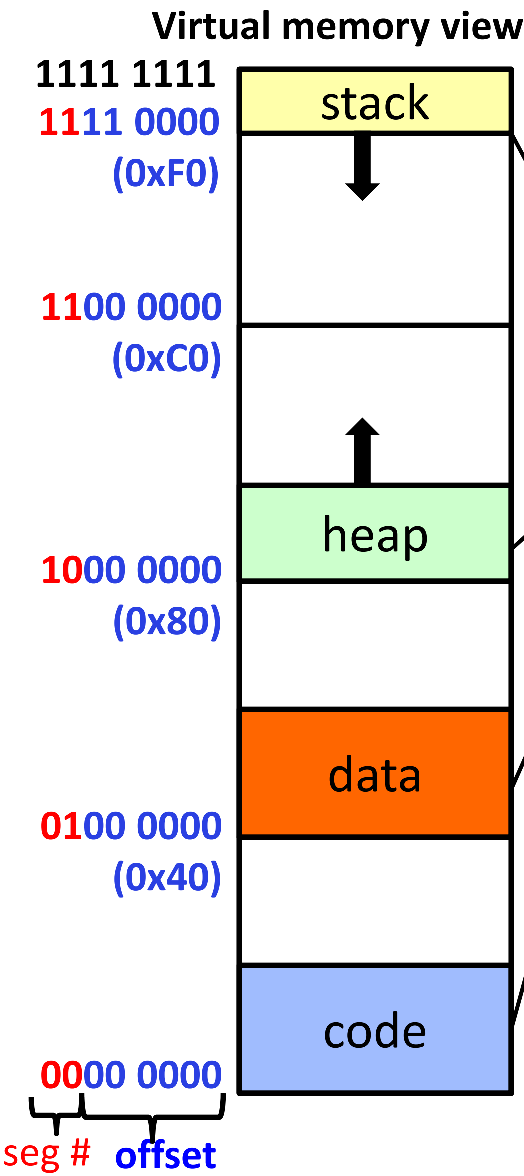
- Lottery Scheduling:
  - Give each thread a number of tokens (short tasks $\Rightarrow$ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness

Protection: Address Spaces and Translation

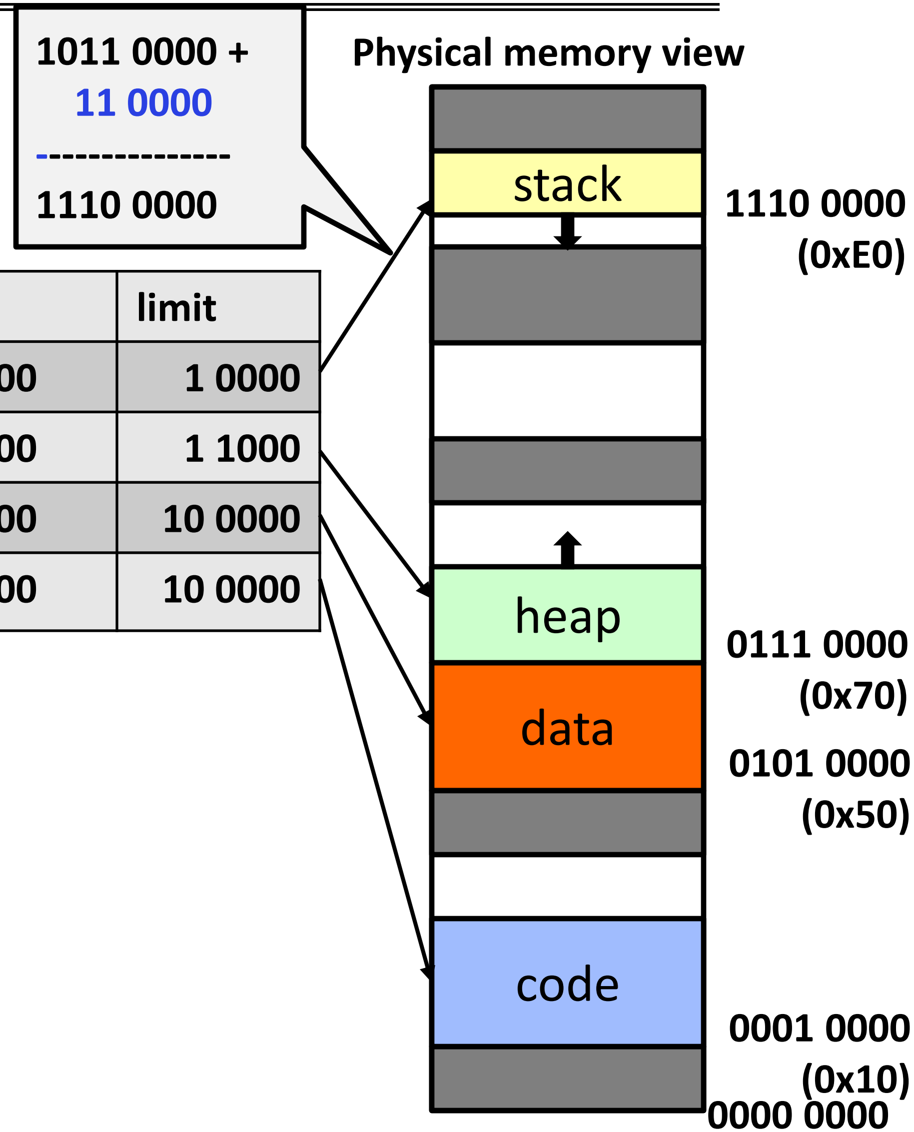
Memory Multiplexing

- Controlled overlap:
  - Processes should not collide in physical memory
  - Conversely, would like the ability to share memory when desired (for communication)
- Protection:
  - Prevent access to private memory of other processes
    - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc)
    - » Kernel data protected from User programs
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, process uses virtual addresses, physical memory uses physical addresses
  - Side Effects:
    - » Uniform view of memory to programs
    - » Avoid overlap

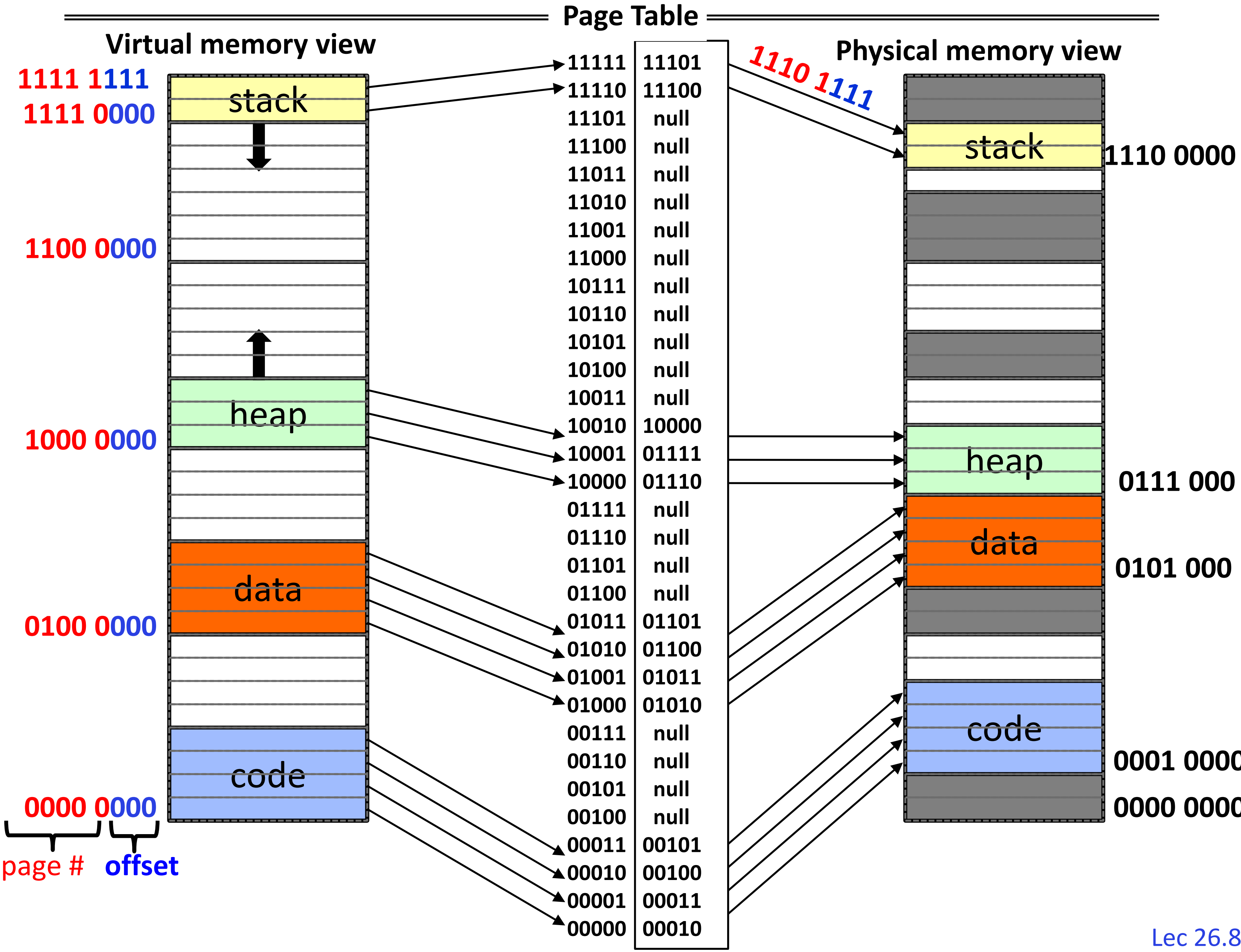
Address Segmentation



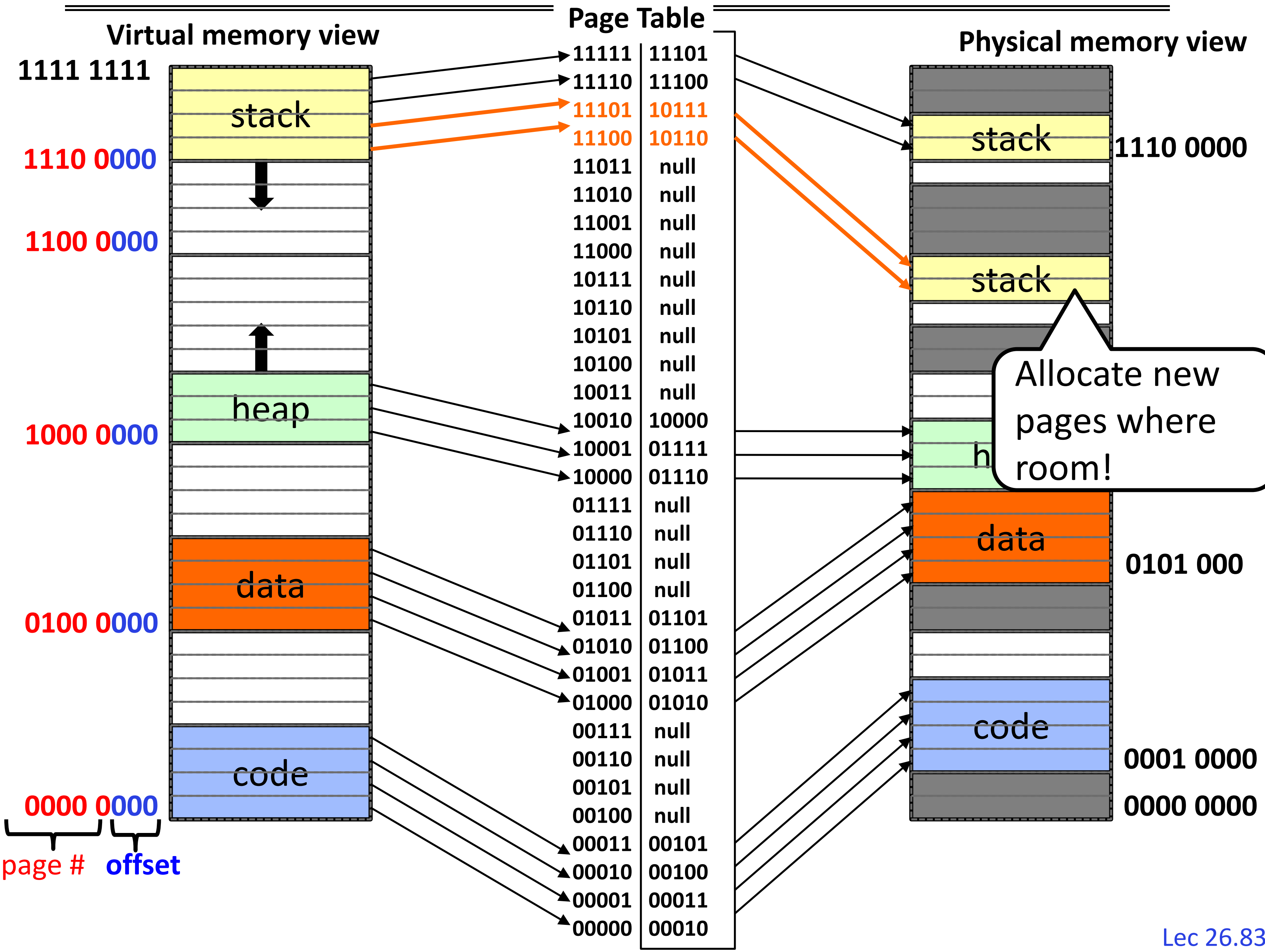
Seg #	base	limit
11	1011 0000	1 0000
10	0111 0000	1 1000
01	0101 0000	10 0000
00	0001 0000	10 0000



Paging


Page Table
Virtual memory view
Physical memory view
1111 1111
1111 0000
1100 0000
1000 0000
0100 0000
0000 0000
page #
offset
stack
heap
data
code
11111 11101
11110 11100
11101 null
11100 null
11011 null
11010 null
11001 null
11000 null
10111 null
10110 null
10101 null
10100 null
10011 null
10010 10000
10001 01111
10000 01110
01111 null
01110 null
01101 null
01100 null
01011 01101
01010 01100
01001 01011
01000 01010
00111 null
00110 null
00101 null
00100 null
00011 00101
00010 00100
00001 00011
00000 00010
1110 1111
stack
1110 0000
heap
0111 000
data
0101 000
code
0001 0000
0000 0000

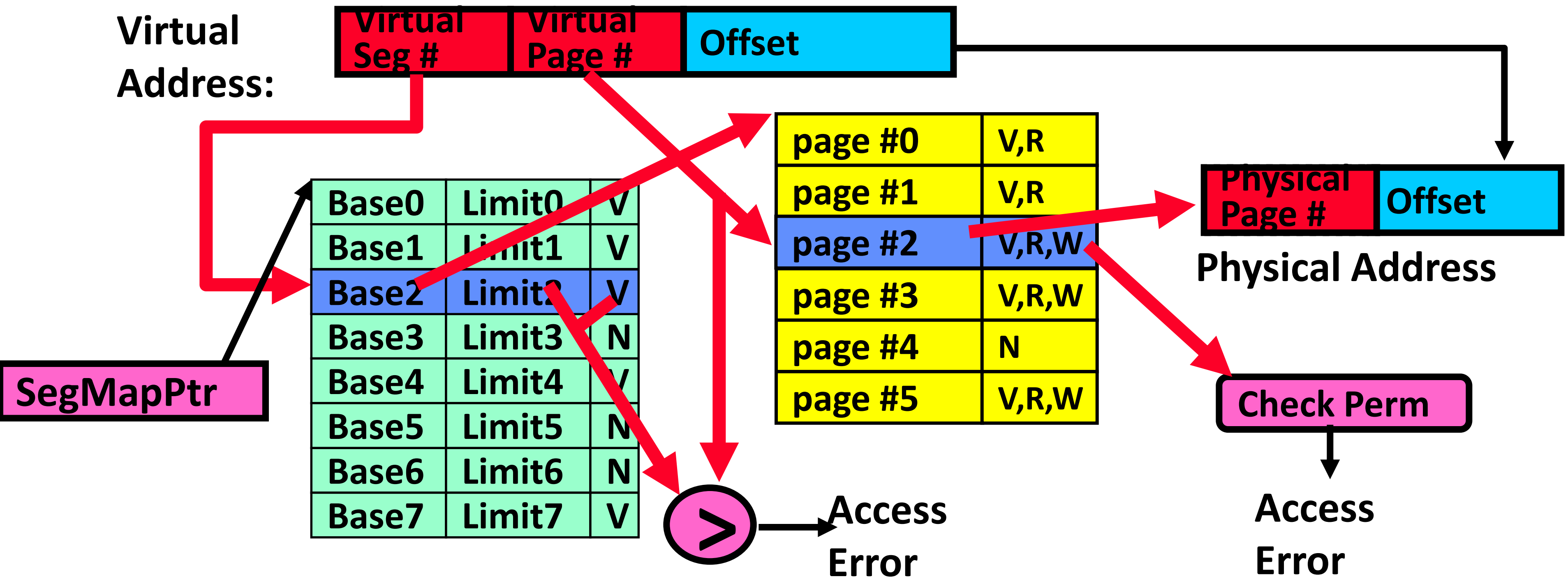
Paging



Page Table
Virtual memory view
Physical memory view
1111 1111
stack
1110 0000
1100 0000
heap
1000 0000
data
0100 0000
code
0000 0000
page #
offset
11111 11101
11110 11100
11101 10111
11100 10110
11011 null
11010 null
11001 null
11000 null
10111 null
10110 null
10101 null
10100 null
10011 null
10010 10000
10001 01111
10000 01110
01111 null
01110 null
01101 null
01100 null
01011 01101
01010 01100
01001 01011
01000 01010
00111 null
00110 null
00101 null
00100 null
00011 00101
00010 00100
00001 00011
00000 00010
stack
1110 0000
stack
Allocate new pages where room!
h
data
0101 000
code
0001 0000
0000 0000

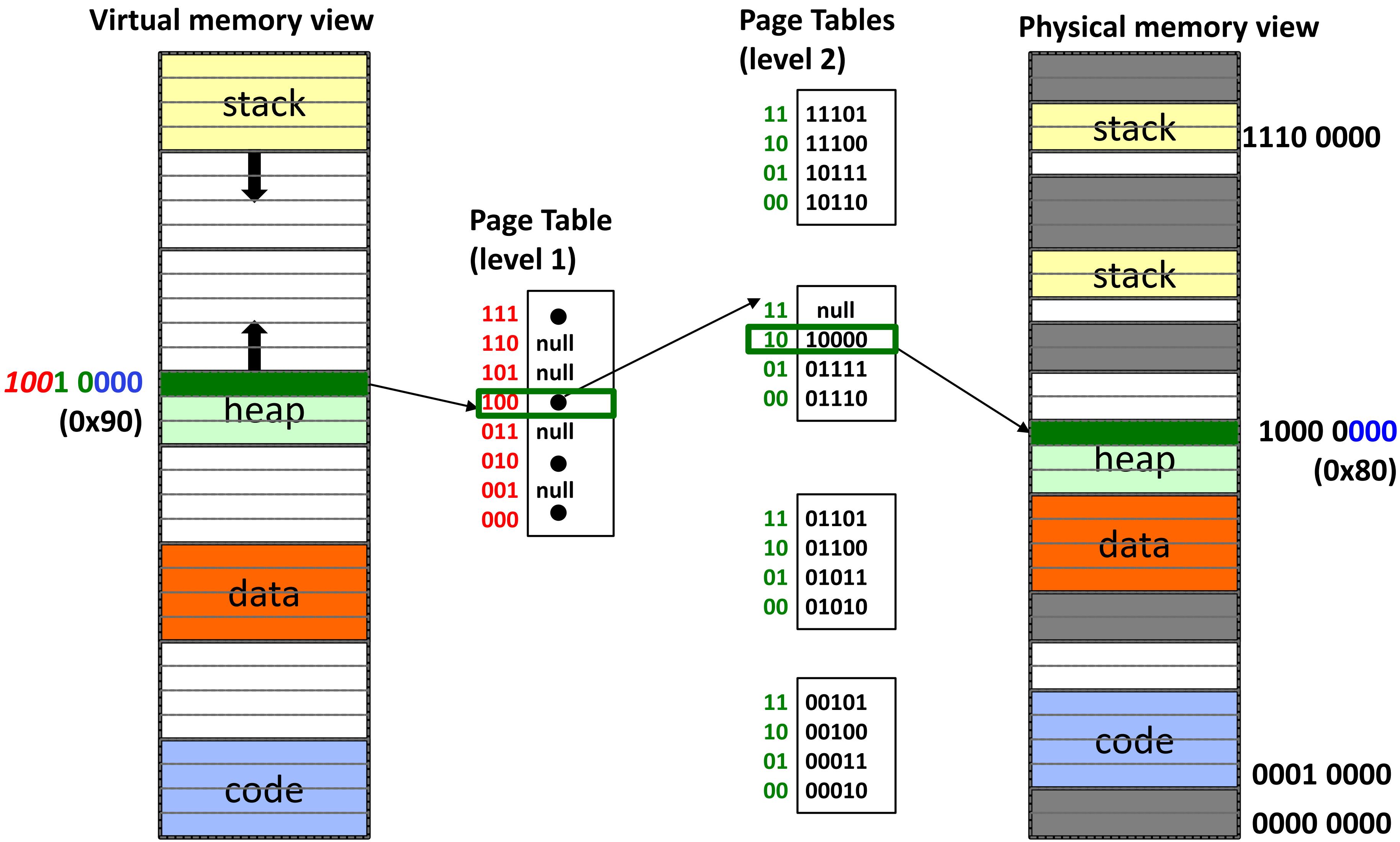
Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

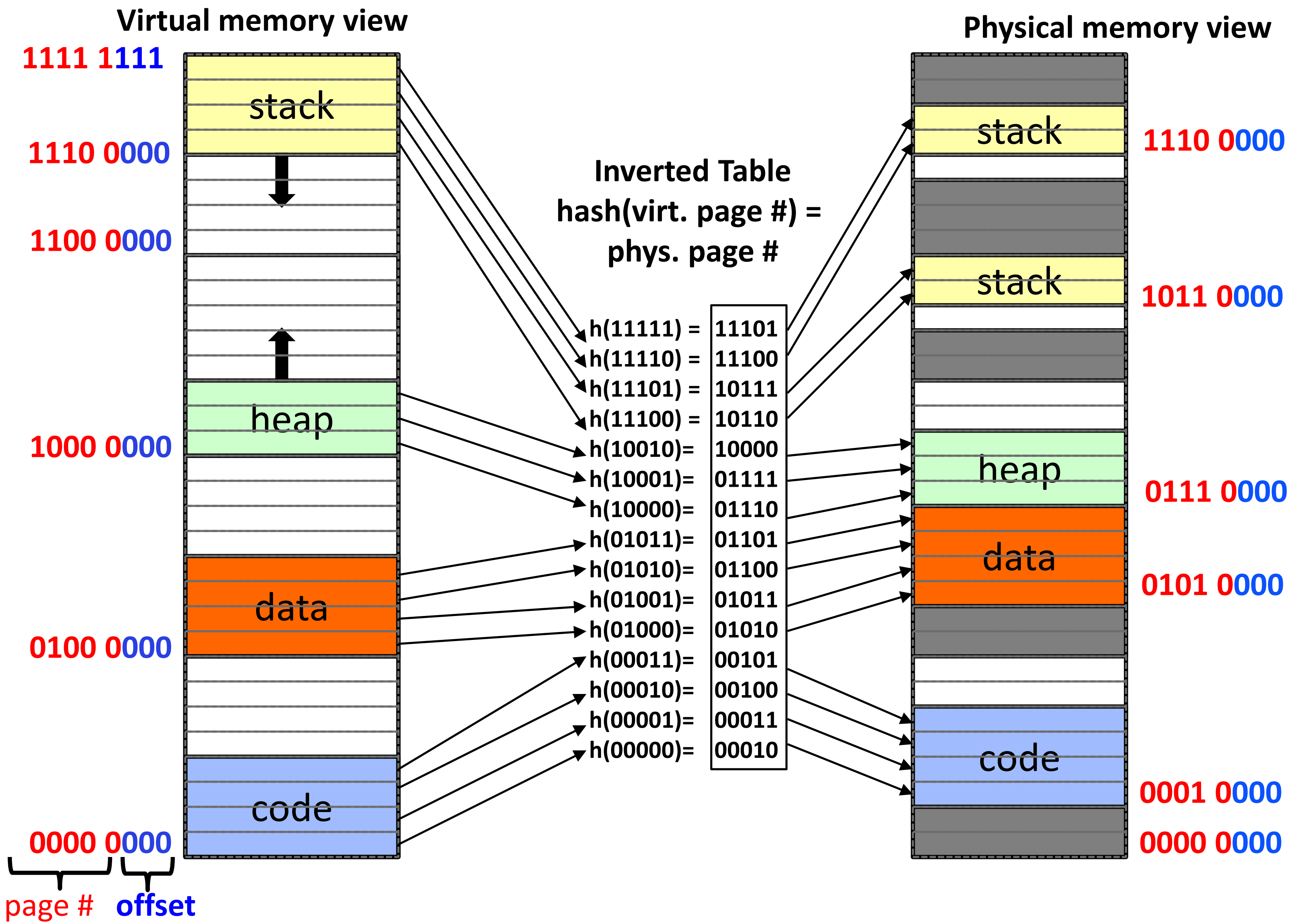


- What must be saved/restored on context switch?
  - Segment map pointer register (for this example)
  - Top-level page table pointer register (2-level page tables)

Two-Level Paging



Inverted Table



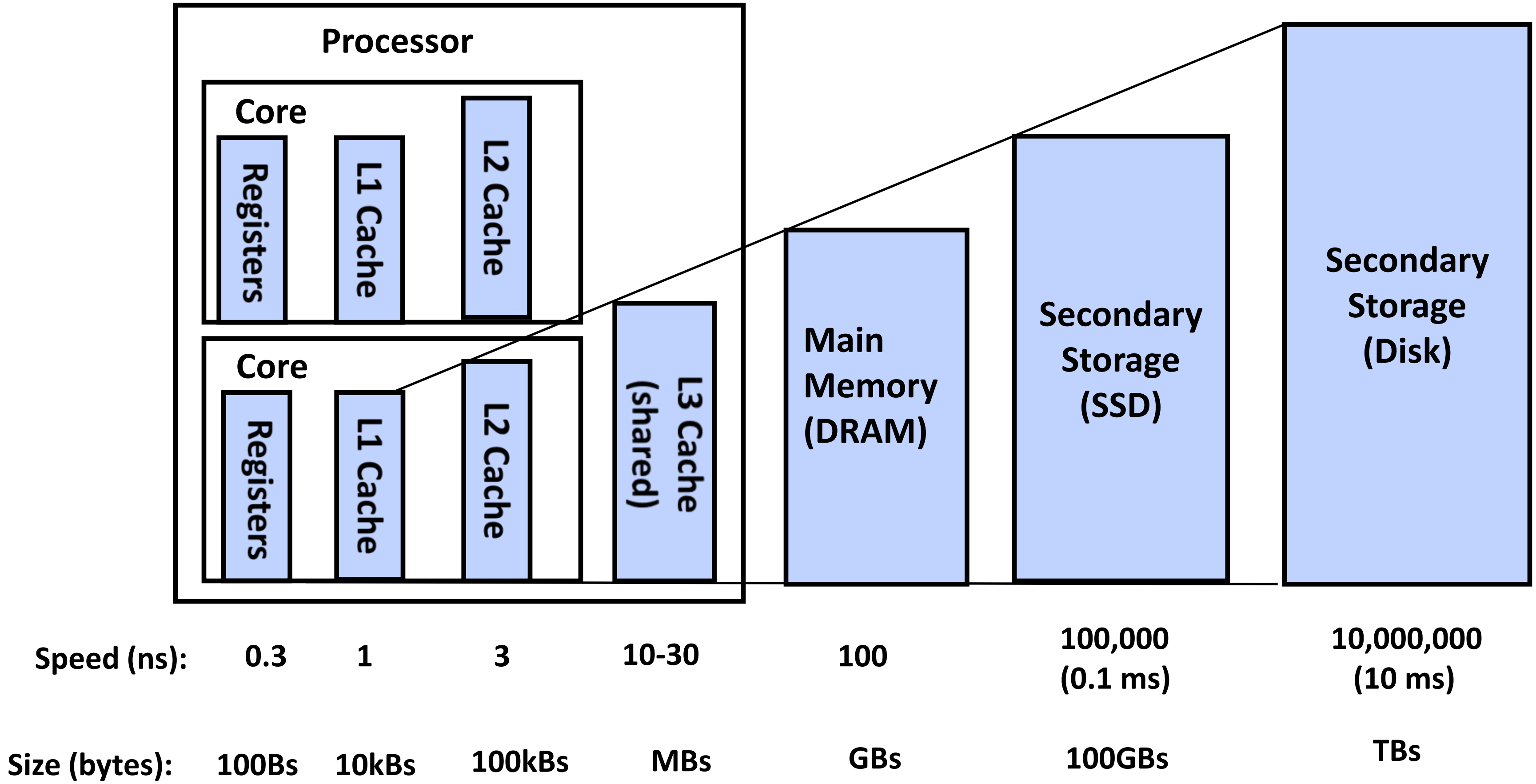
Summary

<table>
<tr><th></th><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>Segmentation</td><td>Fast context switching: Segment mapping maintained by CPU</td><td>External fragmentation</td></tr>
<tr><td>Paging (single-level page)</td><td>No external fragmentation, fast easy allocation</td><td>Large table size ~ virtual memory</td></tr>
<tr><td>Paged segmentation</td><td rowspan="2">Table size ~ # of pages in virtual memory, fast easy allocation</td><td rowspan="2">Multiple memory references per page access</td></tr>
<tr><td>Two-level pages</td></tr>
<tr><td>Inverted Table</td><td>Table size ~ # of pages in physical memory</td><td>Hash function more complex</td></tr>
</table>

Caching and TLBs

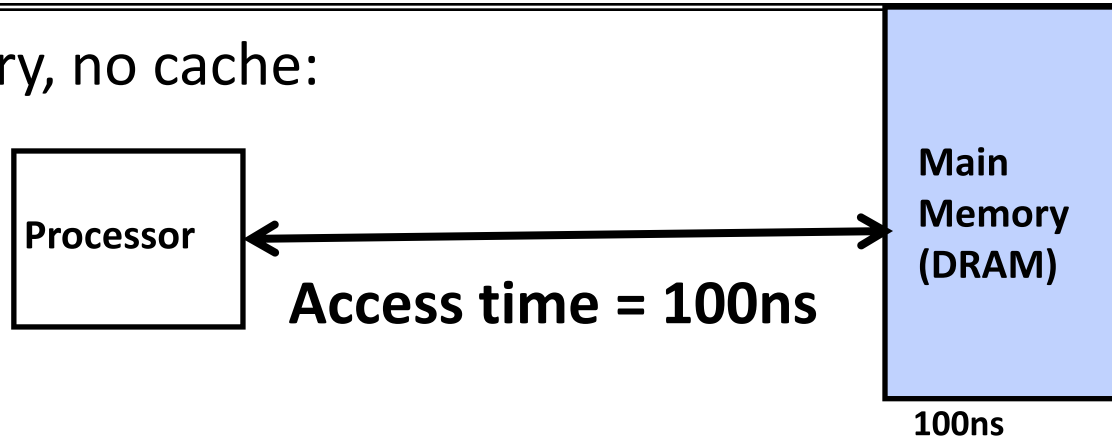
Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology

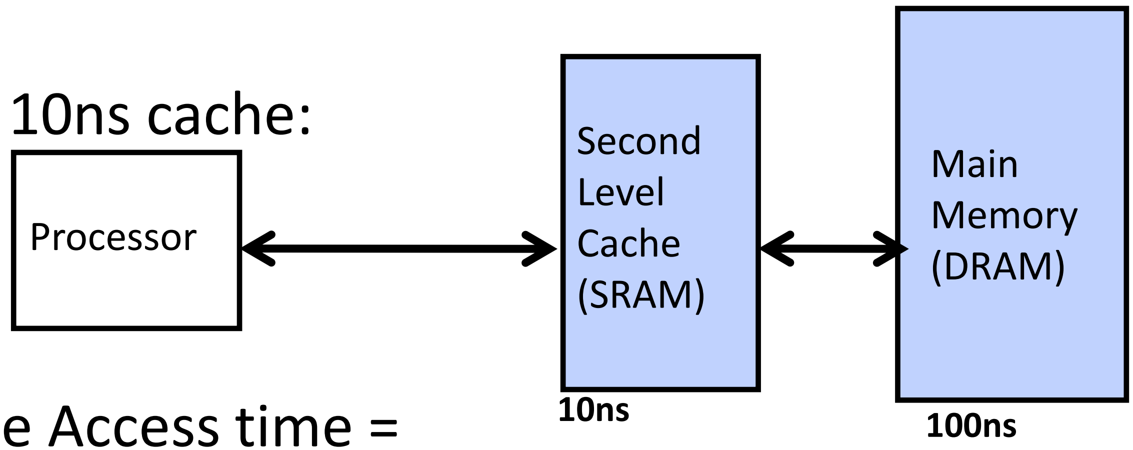


Example

- Data in memory, no cache:



- Data in memory, 10ns cache:

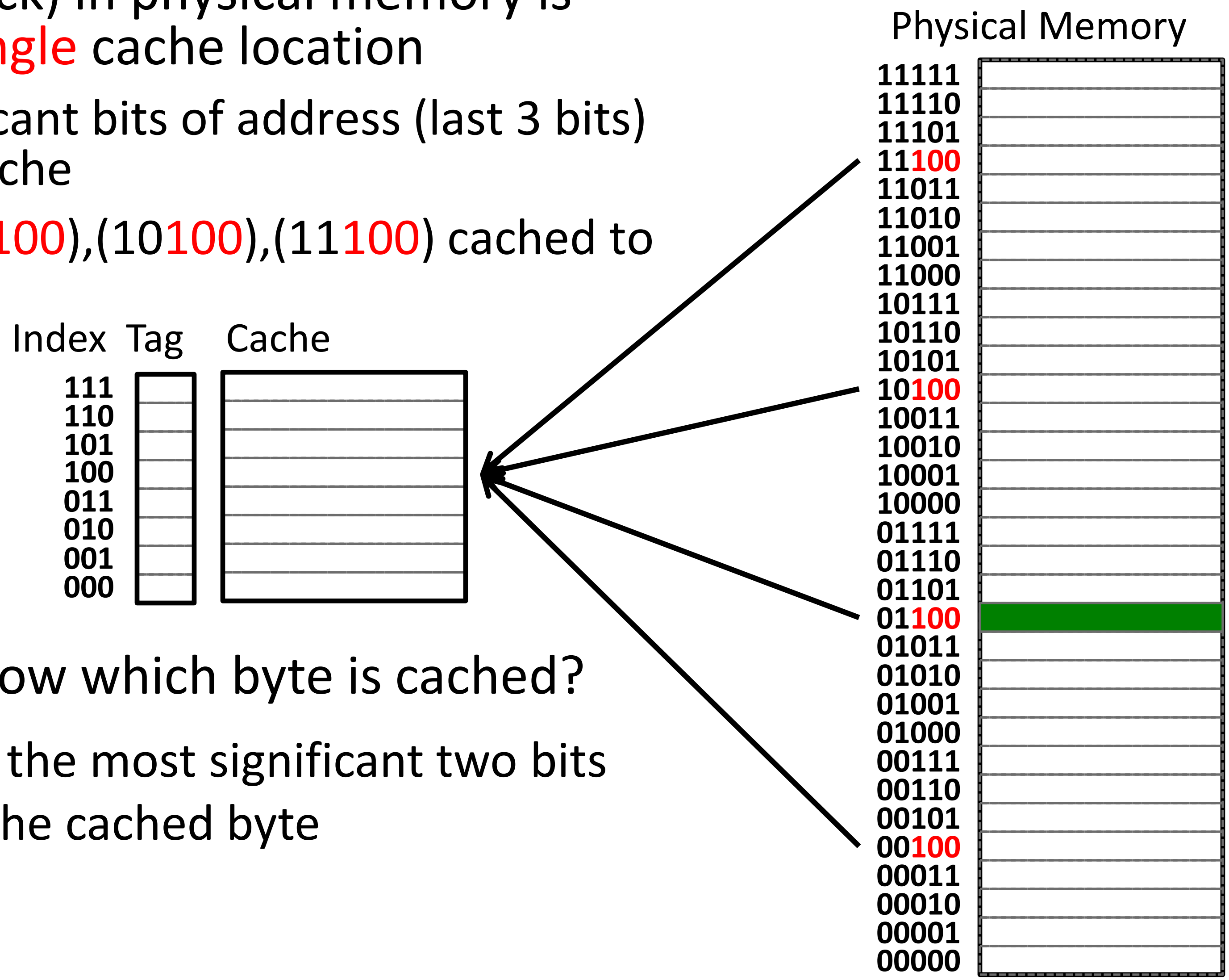


Average Access time =
(Hit Rate x HitTime) + (Miss Rate x MissTime)

- HitRate + MissRate = 1

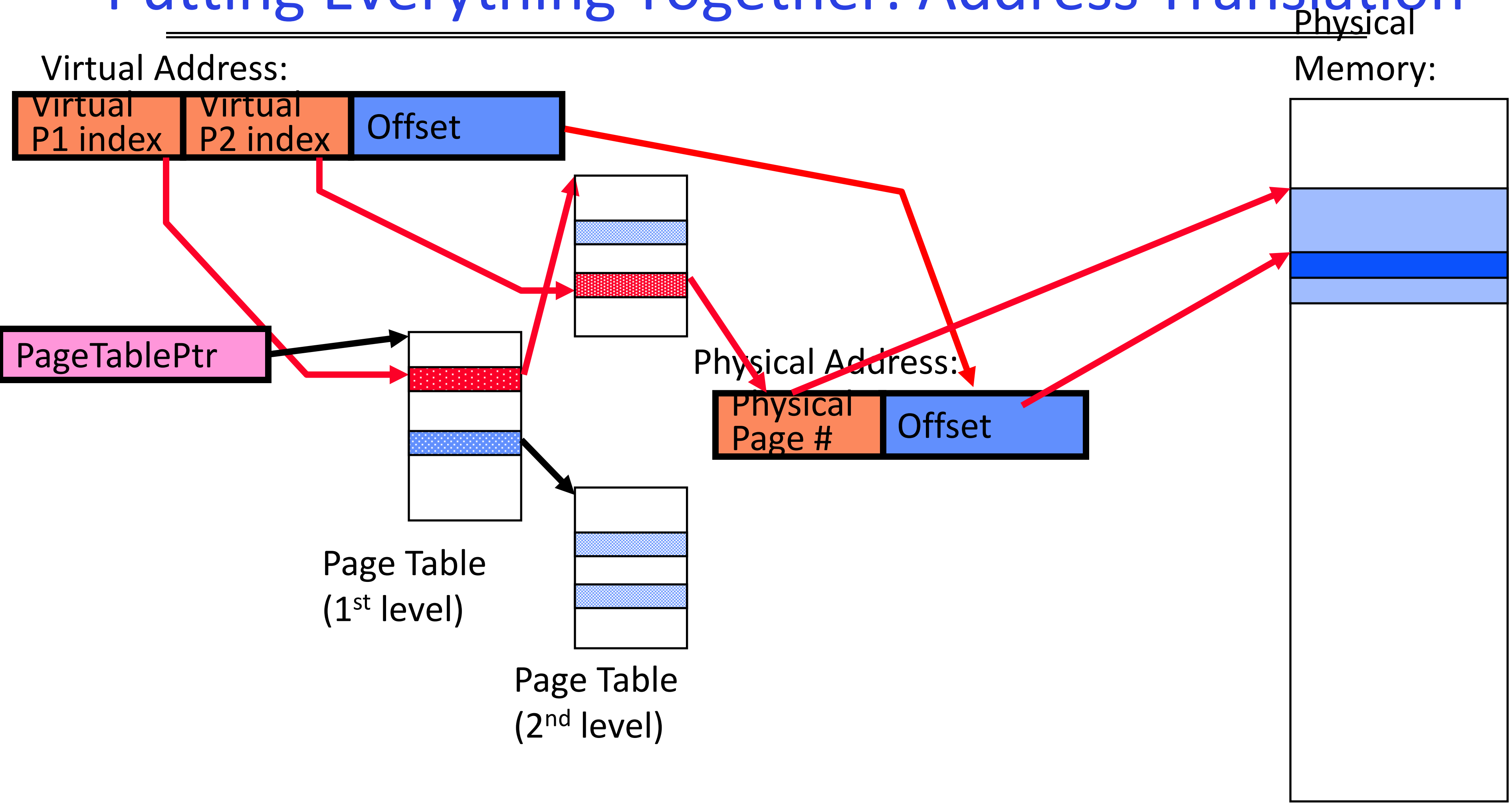
Simple Example: Direct Mapped Cache

- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100

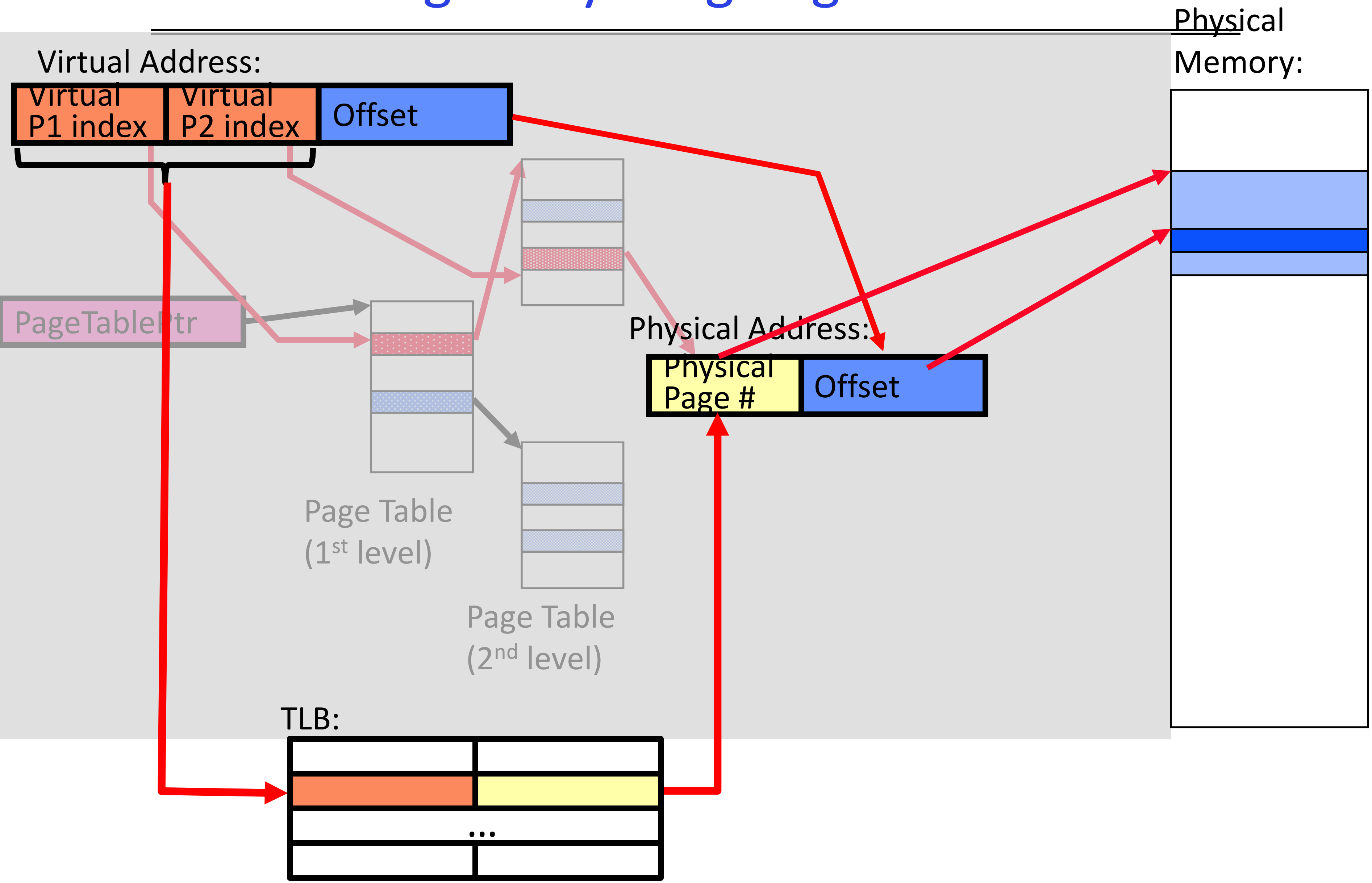


- How do you know which byte is cached?
  - Cache stores the most significant two bits (i.e., **tag**) of the cached byte

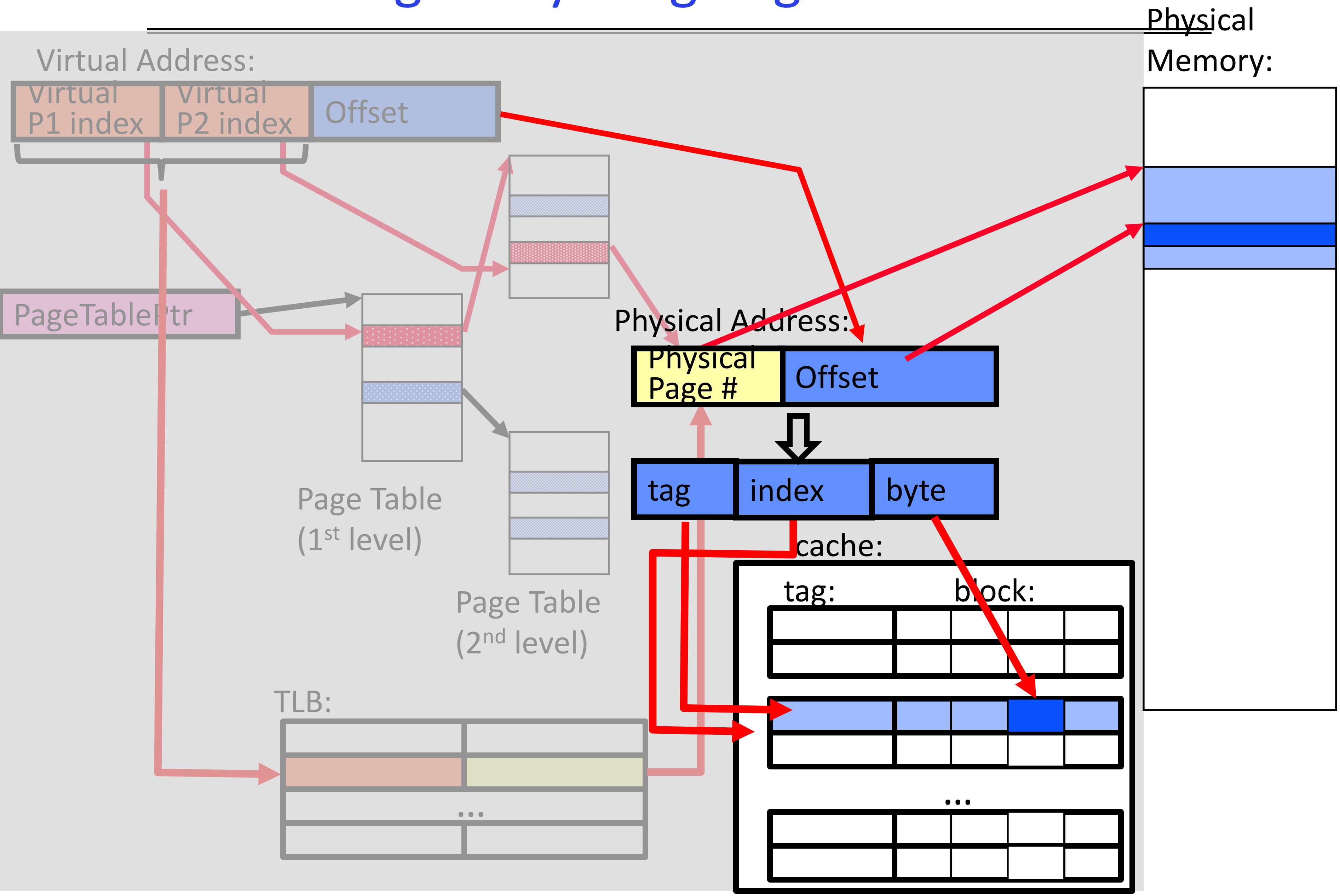
Putting Everything Together: Address Translation



Putting Everything Together: TLB



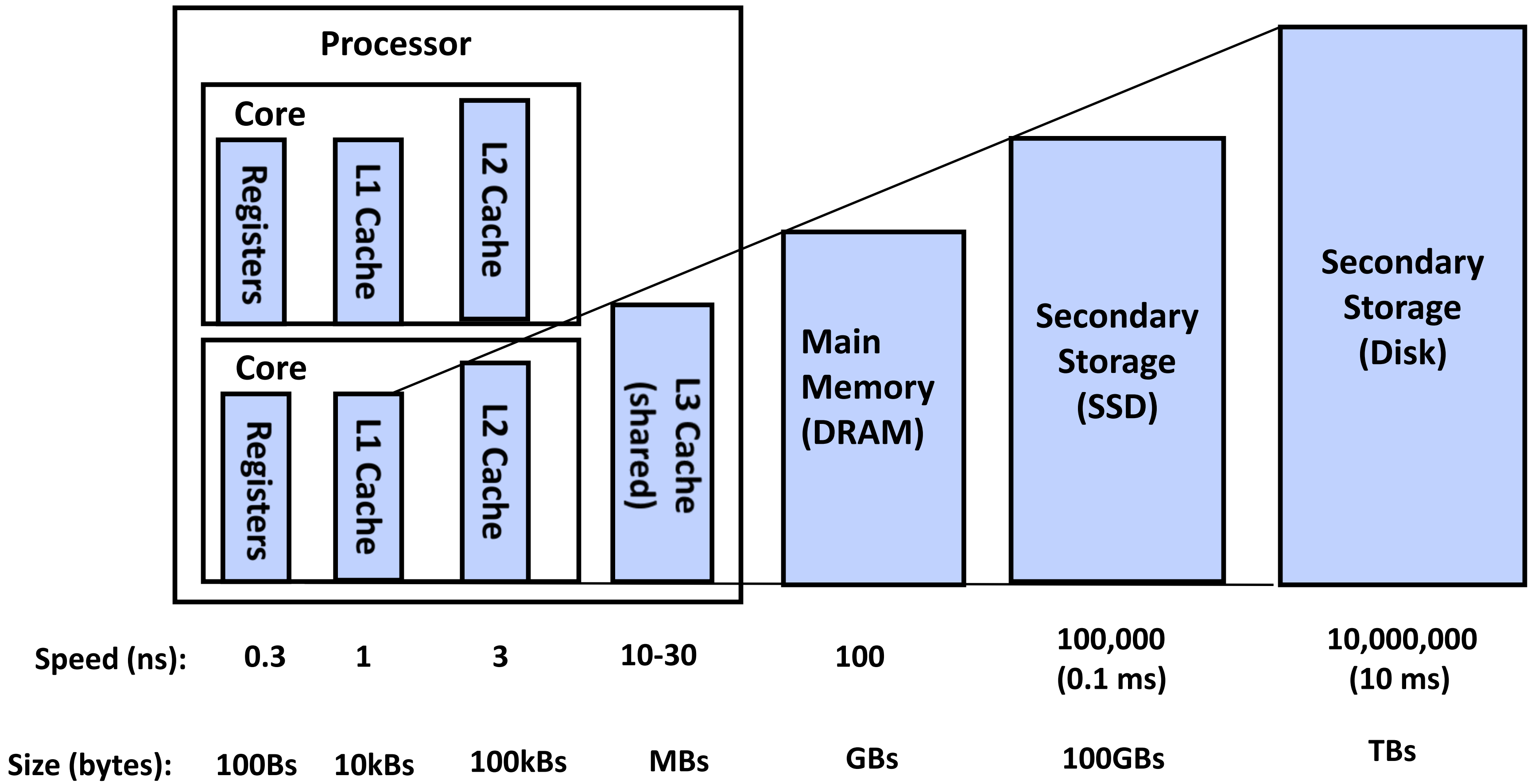
Putting Everything Together: Cache



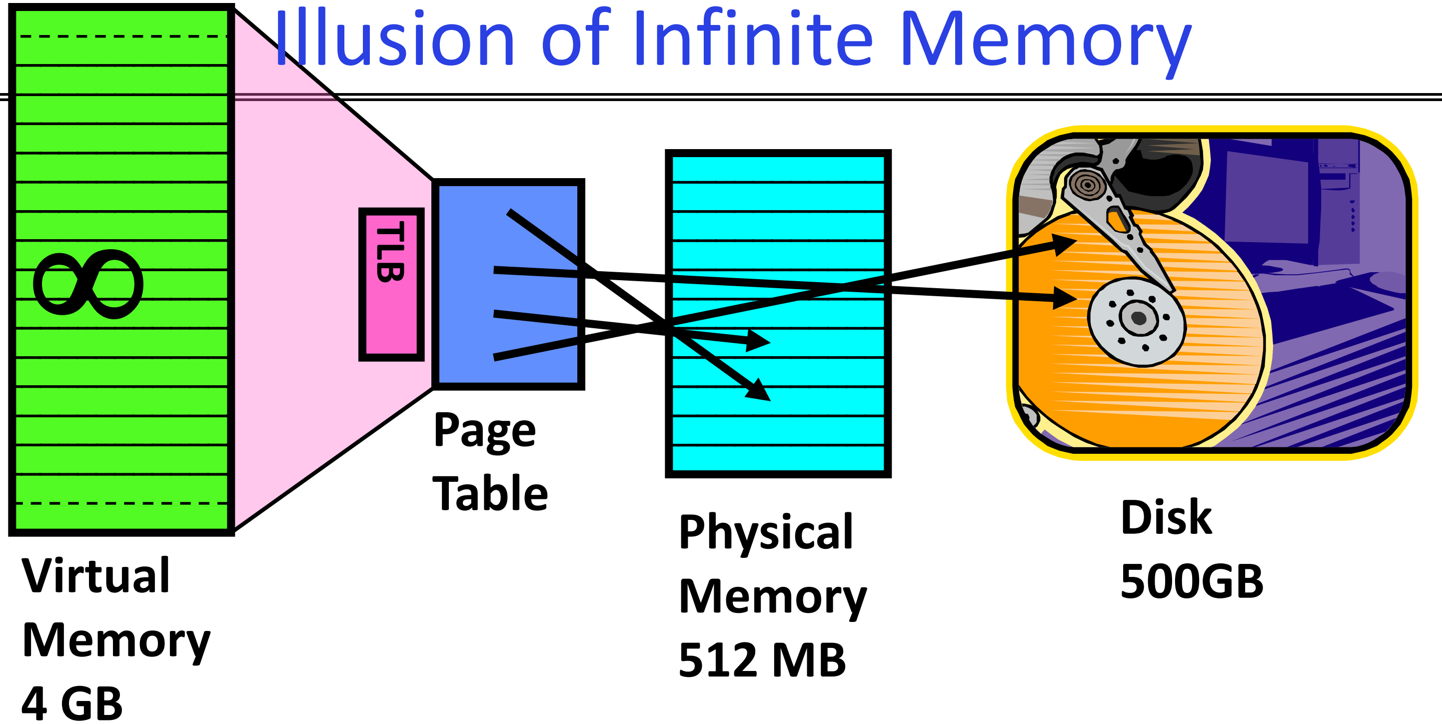
Demand Paging

Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Illusion of Infinite Memory



- Disk is larger than physical memory $\Rightarrow$
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE to be invalid
    - » Load new page into memory from disk
    - » Update page table entry
    - » Continue thread from original faulting location
  - While pulling pages off disk for one process, OS runs another process from ready queue

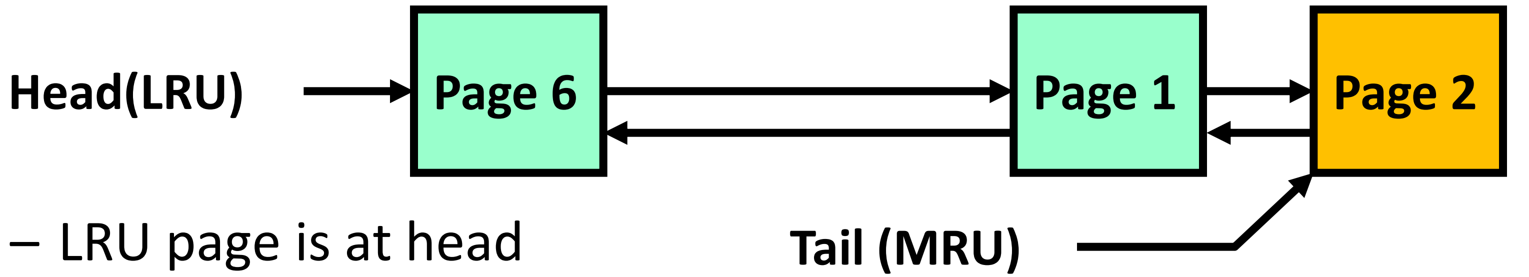
Cache

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Unpredictable – makes it hard to make real-time guarantees

Review: Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- Different if we access a page that is already loaded:



  - LRU page is at head
  - When a page is used again, **remove from list**, add it to tail.
  - Eject head if list longer than capacity
- Problems with this scheme for paging? Too Expensive
  - Updates are happening on page **use**, not just swapping
  - List structure requires extra pointers compared to FIFO, more updates

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	**A**			**A**		**D**				**C**	
2		**B**					**A**				
3			**C**						**B**		

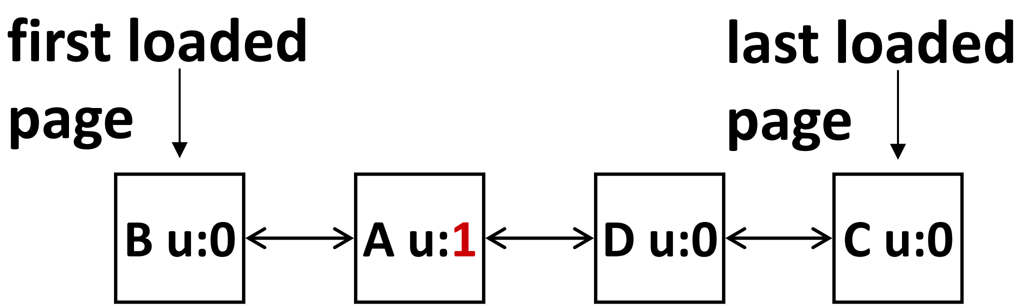
  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Implementing LRU & Second Chance

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality
- Second Chance Algorithm:
  - Approximate LRU (approx. to approx. to MIN)
    - » Replace an old page, not the oldest page
  - FIFO with “use” bit
- Details
  - A “use” bit per physical page
    - » Set when page accessed
    - » If not set, not referenced since last time use bit was cleared
  - On page fault check page at head of queue
    - » If use bit=1 → clear bit, and move page to tail (give the page second chance!)
    - » If use bit=0 → replace page
  - Moving pages to tail still complex

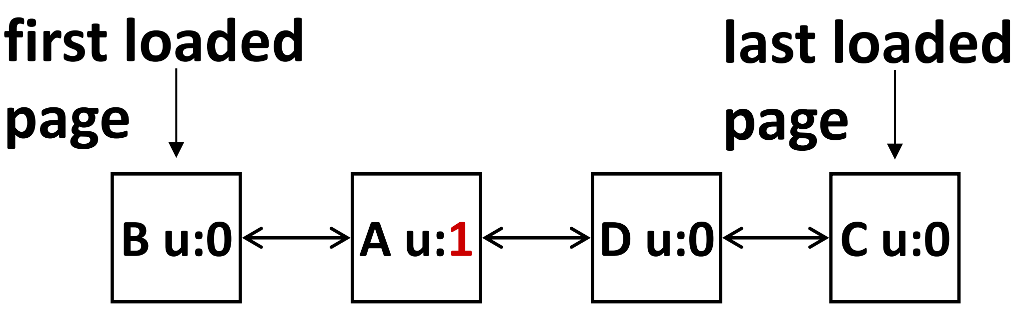
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives



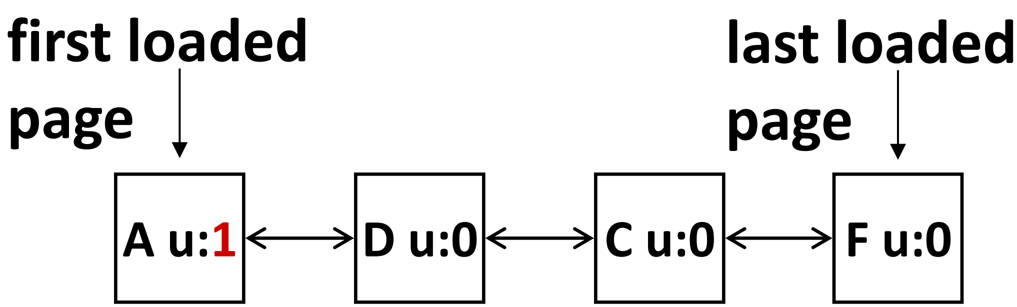
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



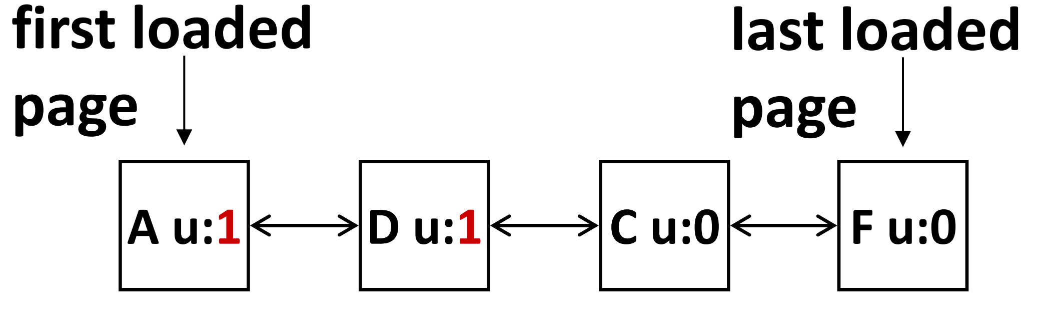
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



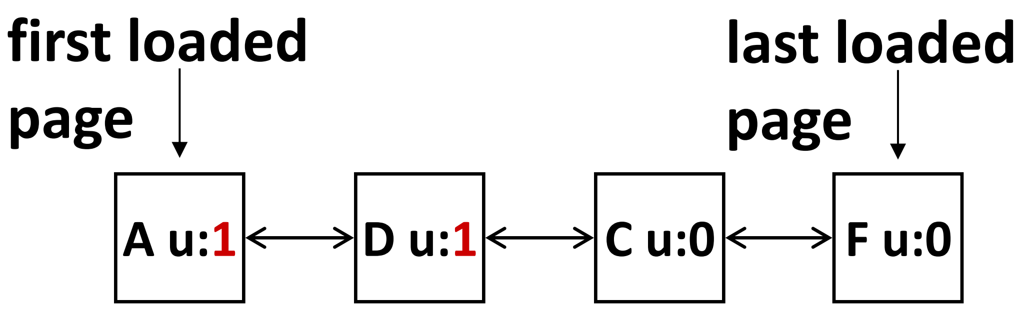
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D



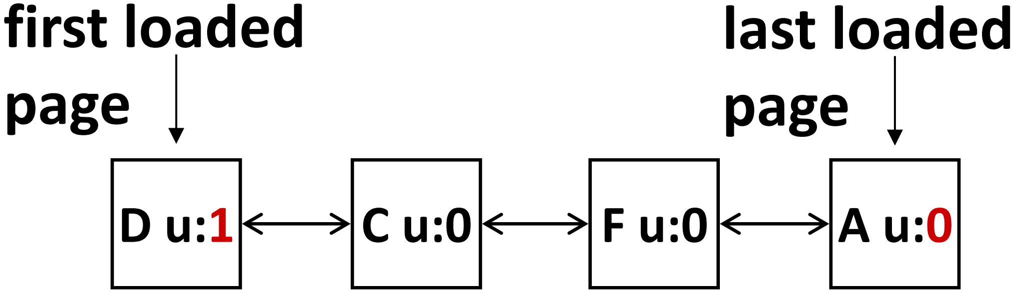
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



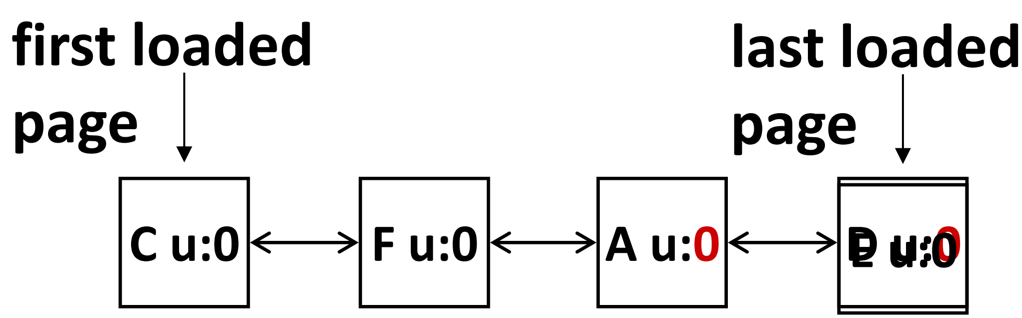
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Clock Algorithm

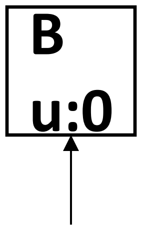
- Clock Algorithm: more efficient implementation of second chance algorithm
  - Arrange physical pages in circle with single clock hand
- Details:
  - On page fault:
    - » Check use bit: 1→used recently; clear and leave it alone
      0→selected candidate for replacement
    - » Advance clock hand (not real time)
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around (FIFO)

Clock Replacement Illustration

- Max page table size 4

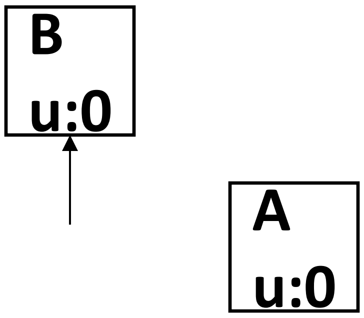
- Invariant: point at oldest page

  – Page B arrives



Clock Replacement Illustration

- Max page frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A

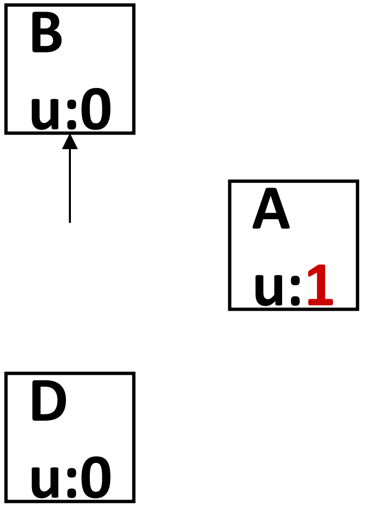


Clock Replacement Illustration

- Max page frames = 4

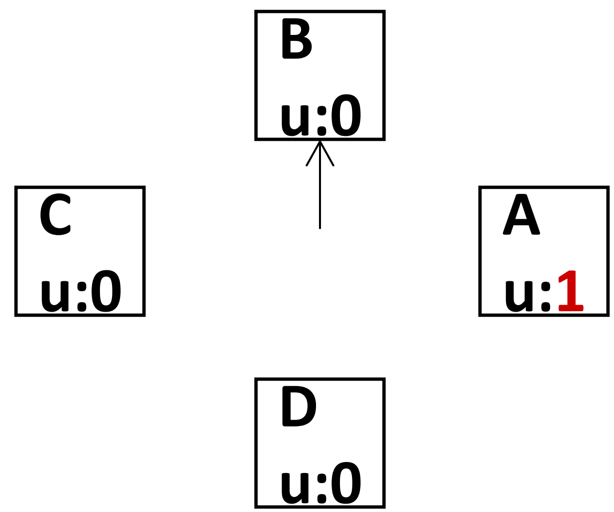
- Invariant: point at oldest page

  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives



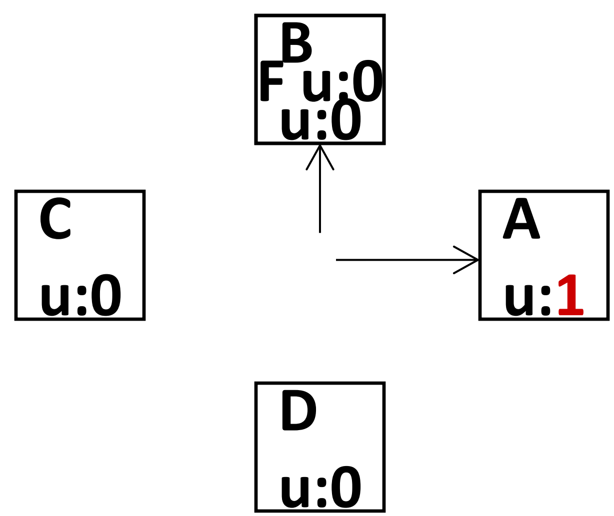
Clock Replacement Illustration

- Max page frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives



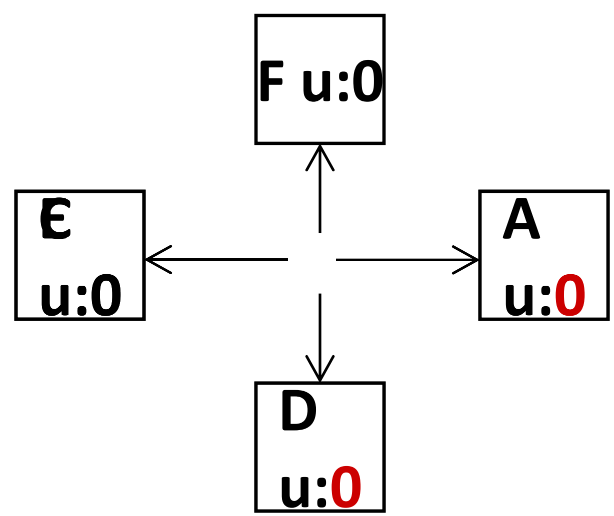
Clock Replacement Illustration

- Max page table frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D



Clock Replacement Illustration

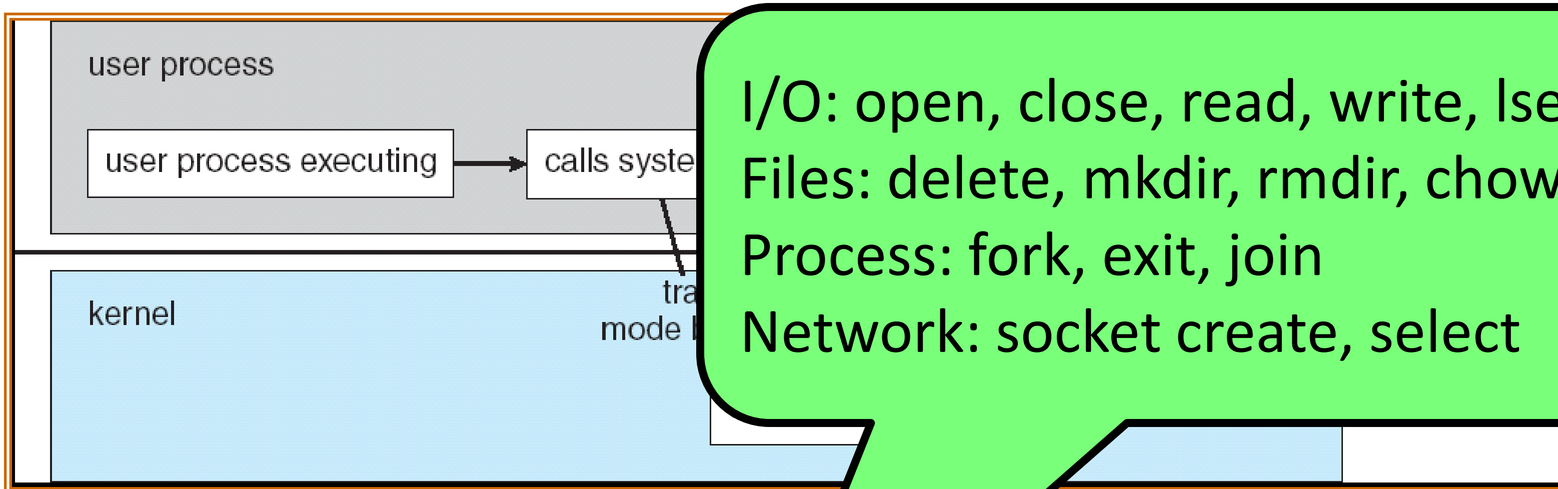
- Max page frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Dual Mode Operation

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  – Would defeat purpose of protection!
  – So, how does the user program get back into kernel?



I/O: open, close, read, write, lseek
Files: delete, mkdir, rmdir, chown
Process: fork, exit, join
Network: socket create, select

- System call: Voluntary procedur… into kernel
  – Hardware for controlled User… rnel transition
  – Can any kernel routine be ca… ed?
    » No! Only specific ones

User→Kernel (Exceptions: Traps and Interrupts)

- System call instr. causes a synchronous exception (or “trap”)
  - In fact, often called a software “trap” instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- SUMMARY – On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.

I/O Systems

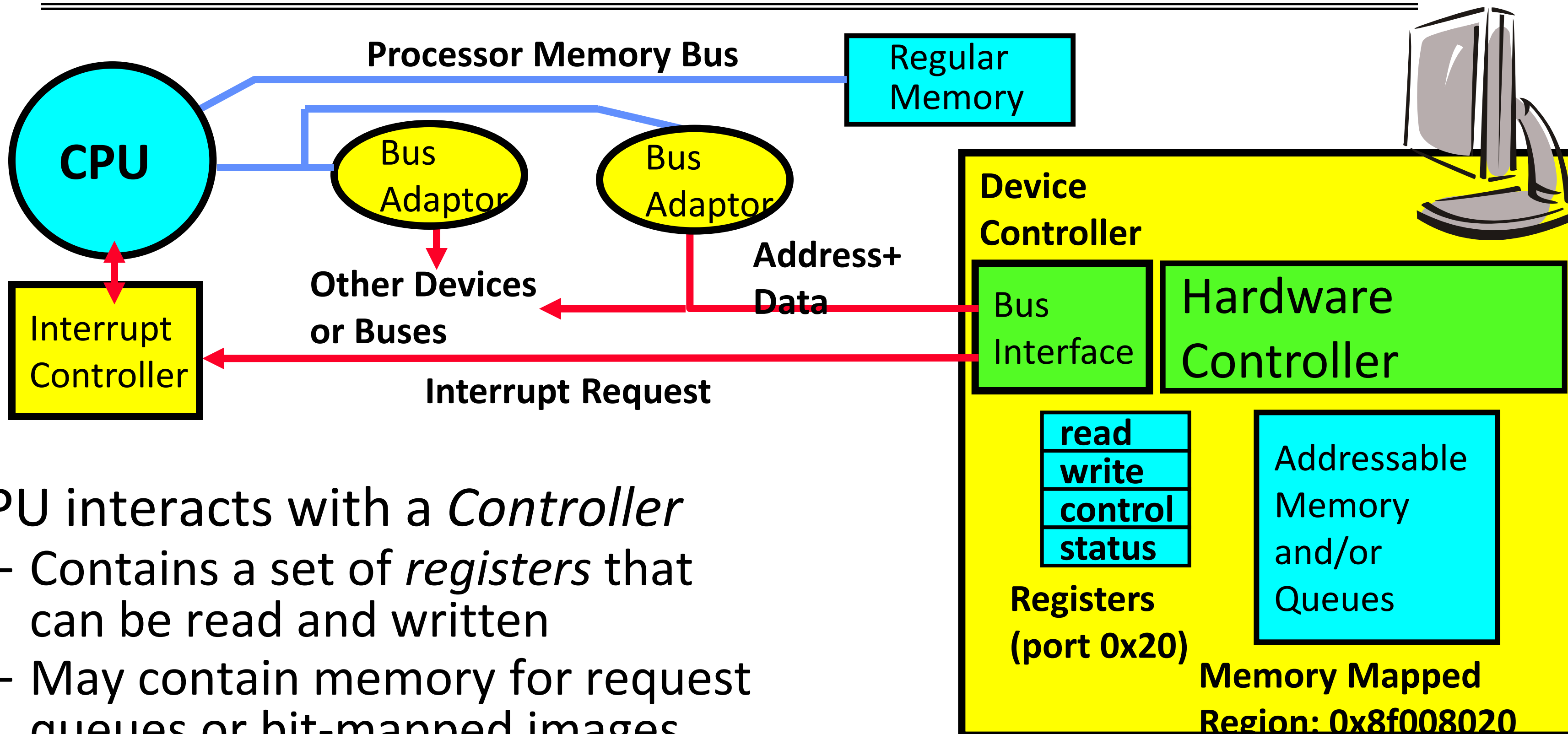
The Goal of the I/O Subsystem

- Provide uniform interfaces, despite wide range of different devices
  - This code works on many different devices:

```
FILE fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd, "Count %d\n", i);
}
close(fd);
```

  - How? Code that controls devices ("device driver") implements standard interface

How Does the Processor Talk to Devices?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways (IA):
  - I/O instructions: in/out instructions (e.g., Intel's 0x21,AL)
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

SSD

- Pros (vs. hard disk drives):
  - Low latency, high throughput (eliminate seek/rotational delay)
  - No moving parts:
    - » Very light weight, low power (0.3x disk), silent, very shock insensitive
  - Read at memory speeds (limited by controller and I/O bus)

- Cons
  - Smaller storage (0.5x disk), expensive (7~10x disk)
    - » Hybrid alternative: combine small SSD with large HDD
  - Cannot update a single page in a block.
  - Asymmetric block write performance: read pg/erase/write pg
    - » Controller garbage collection (GC) algorithms have major effect on performance

Filesystems

Building a File System

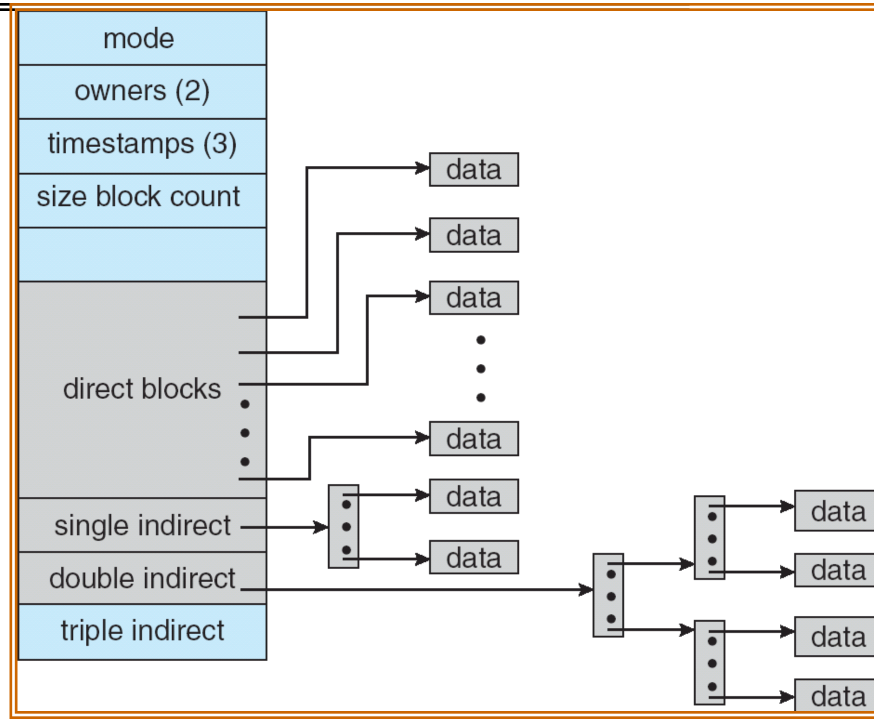
- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

- File System Components
  - Disk Management: organizing disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc.

- File System Goals
  - Maximize sequential performance
  - Efficient random access to file
  - Easy management of files (growth, truncation, etc.)

Multilevel Indexed Files (UNIX 4.1)

- Multilevel Indexed Files:
  (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files



- It is possible to have more than one single-indirect pointers.
- Assume 1024-byte block and 32-bit block pointer
  - What is the maximum size of the disk? 2^32*1024 Bytes
  - How many data blocks one single-indirect pointer can point? 1024/4=256
  - What is the maximum size of the above file system, assuming 10 direct blocks? (10+256+256*256)*1024 Bytes

Naming

- Naming (name resolution): process by which a system translates from user-visible names to system resources

- In the case of files, need to translate from strings (textual names) or icons to inumbers/inodes

Where are inodes stored?

- How many disk accesses to resolve "`./my/book/count.txt`"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data block for root
    - » Table of (filename, inumber/index) pairs. Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data block for "my"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"

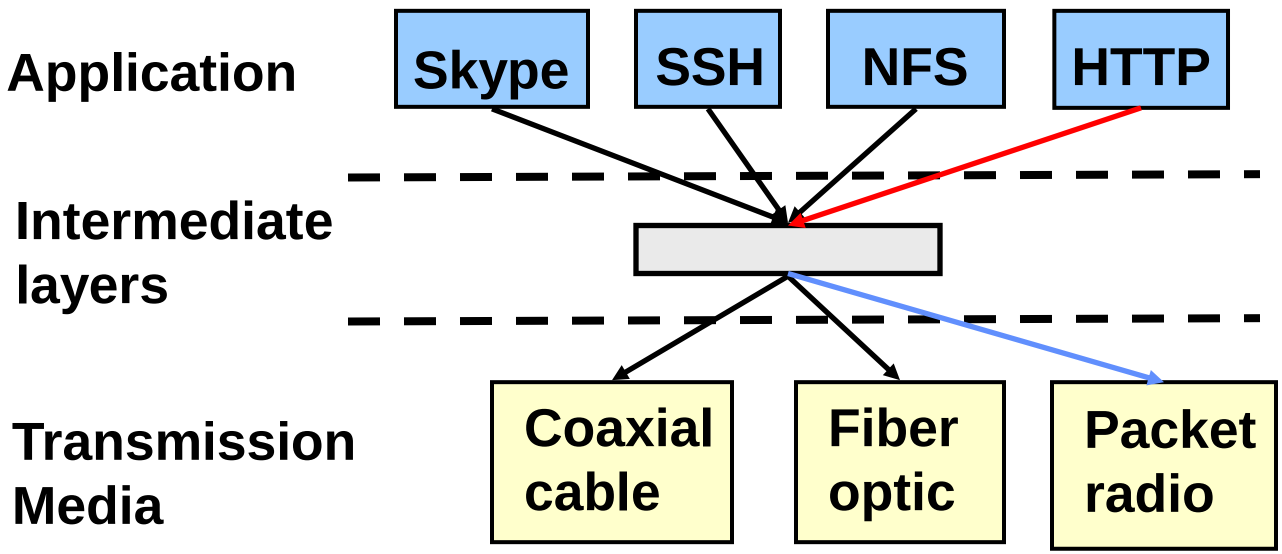
File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain “dirty” blocks (blocks not yet on disk)

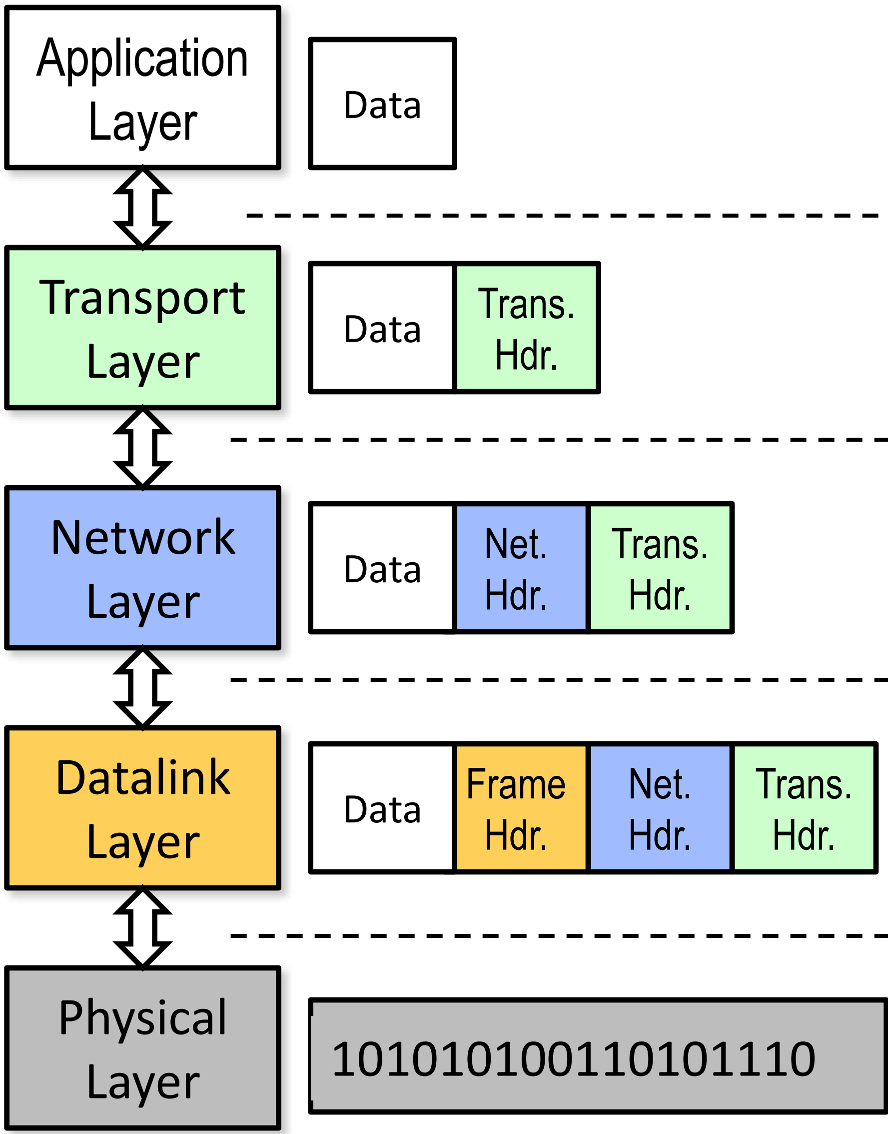
Distributed Systems and Networking

Intermediate Layers

- Introduce intermediate layers that provide set of abstractions for various network functionality & technologies
  - A new app/media implemented only once
  - Variation on "add another level of indirection"



Networking (Internet Layering)



Any distributed protocol (e.g., HTTP, Skype, p2p, KV protocol in your project)

Send *segments* to another *process* running on same or different node

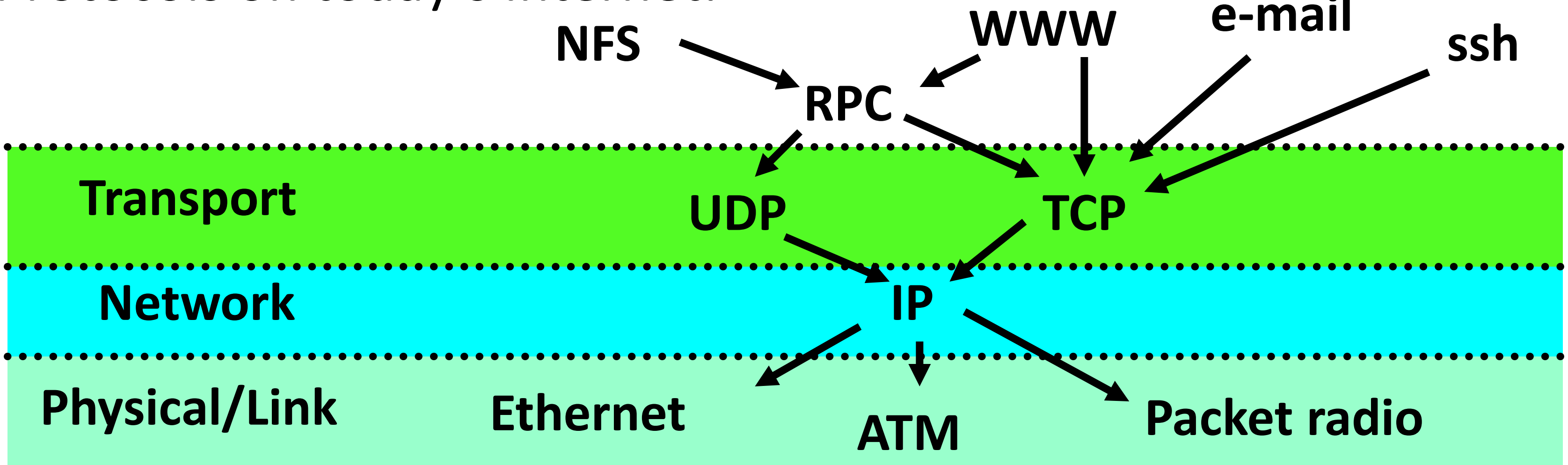
Send *packets* to another node possibly *located* in a different network

Send *frames* to other node directly connected to same physical network

Send *bits* to other node directly connected to same physical network

Network Protocols

- Protocol: Agreement between two parties as to how information is to be transmitted
  - Example: system calls are the protocol between the operating system and applications (maybe a stretch!)
  - Networking examples: many levels
    - » Physical level: mechanical and electrical network (e.g. how are 0 and 1 represented)
    - » Link level: packet formats/error control (for instance, the CSMA/CD protocol)
    - » Network level: network routing, addressing
    - » Transport Level: reliable message delivery
- Protocols on today's Internet:

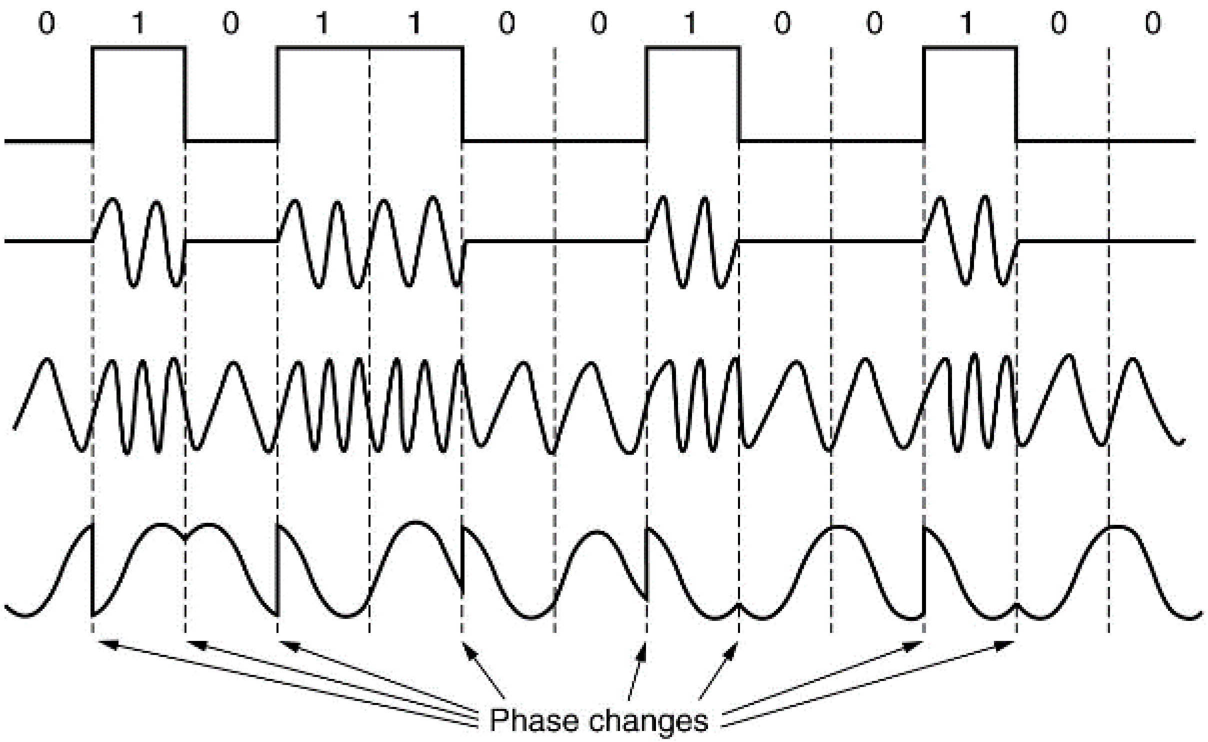


TCP

- TCP: Reliable Byte Stream
  - Open connection (3-way handshaking)
  - Close connection: no perfect solution; no way for two parties to agree in the presence of arbitrary message losses (General's Paradox)

- Reliable transmission
  - Sliding window not efficient for links with large capacity (bandwidth) delay product
  - Sliding window more efficient but more complex

Physical layer - Modulation



Binary signal

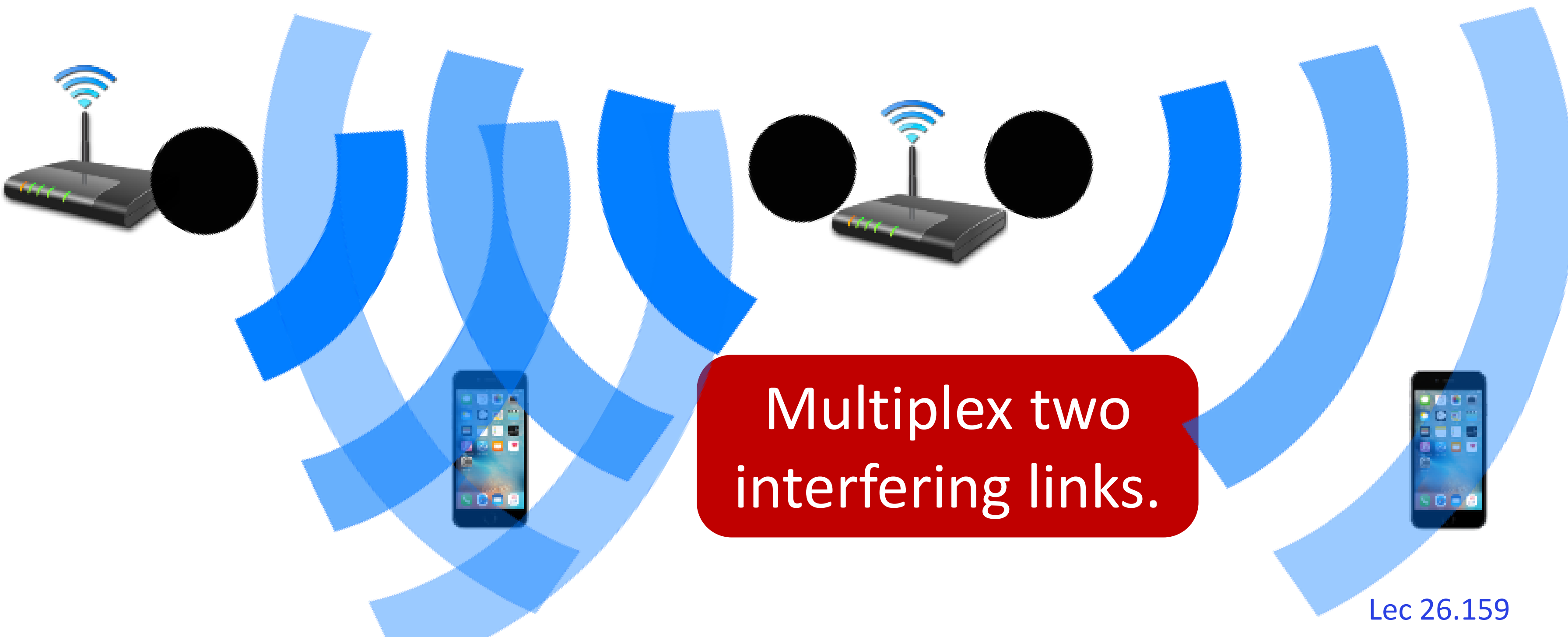
Amplitude modulation

Frequency modulation

Phase modulation

Medium Access Control (MAC)

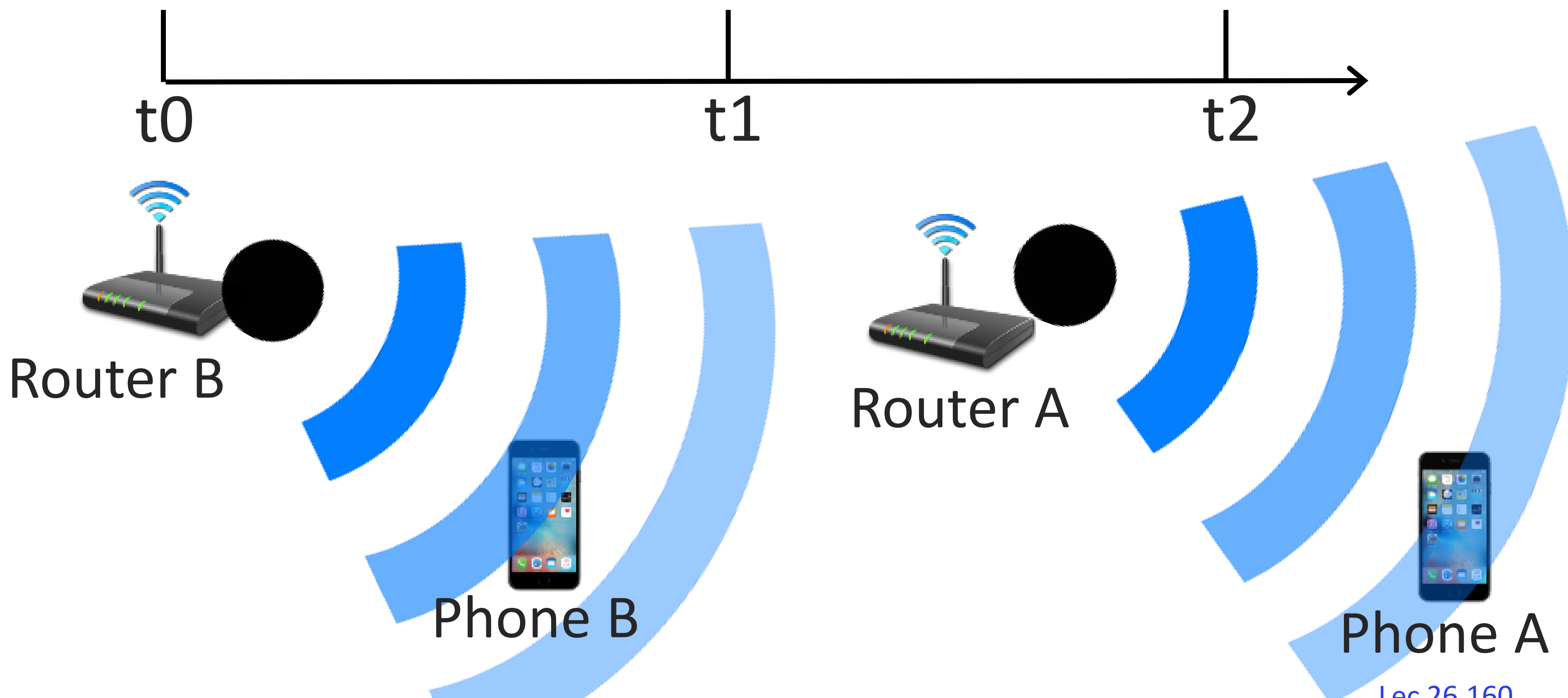
- Contention-free protocols
  - TDMA (Time Division Multiple Access)
  - FDMA (Frequency Division Multiple Access)
- Contention-based protocols
  - ALOHA (random access)
  - CSMA (Carrier Sense Multiple Access)



Carrier Sense Multiple Access (CSMA)

Sense by measuring the received signal strength.

If the channel is busy, retry after a random backoff.



Thank you!!