CS162
Operating Systems and Systems Programming

Midterm Review

March 5, 2012

http://inst.eecs.berkeley.edu/~cs162

Synchronization, Critical section

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Locks: using interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

Better locks: using test&set

- ```
  test&set (&address) {   /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

```
int guard = 0;
int value = FREE;
```

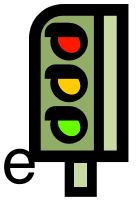
```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
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 = FREE;
   }
   guard = 0;
```
```

- Does this busy wait? If so, how long does it wait?
- Why is this better than the interrupt version?

- Miscellaneous: Remember that a thread can be context-switched while holding a lock.

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
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - **You cannot read the integer value!** The only methods are P() and V().
  - Unlike lock, there is no owner (thus no priority donation). The thread that calls V() may never call P(). Example: Producer/consumer

Buffered Producer/Consumer

- Correctness Constraints:
  - Consumer must wait for producer to fill slots, if empty (scheduling constraint)
  - Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)

- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullSlots; // consumer's constraint`
  - `Semaphore emptySlots;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

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 slot 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;
}
```

Condition Variables

- 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

- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. **Re-acquire** lock later, **before returning**.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters

- **Rule: Must hold lock when doing condition variable ops!**

Complete Monitor Example (with condition variable)

- Here is an (infinite

Is this Hoare or Mesa monitor?

```
Lock lock;
Condition da
Queue queue;


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);
}
```

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

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

  - Why didn't we do this?

```
if (queue.isEmpty()) {
  dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

- Answer: depends on the type of scheduling
  - Hoare-style (most textbooks):
    - » Signaler gives lock, CPU to waiter; **waiter runs immediately**
    - » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  - Mesa-style (most real operating systems):
    - » Signaler keeps lock and processor
    - » Waiter placed on ready queue with no special priority
    - » Practically, need to check condition again after wait

Reader/Writer

- Two types of lock: read lock and write lock (i.e. shared lock and exclusive lock)
- Shared locks is an example where you might want to do broadcast/wakeAll on a condition variable. If you do it in other situations, correctness is generally preserved, but it is inefficient because one thread gets in and then the rest of the woken threads go back to sleep.

Read/Writer

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only
    AccessDbase(

    // check out
    lock.Acquire(
    AR--;
    if (AR == 0 && WW > 0)
      okToWrite.signal();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okToWrite.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okToWrite.signal();
    } else if (WR > 0) {
      okToRead.broadcast();
    }
    lock.Release();
  }
```

What if we remove this line?

Read/Writer Revisited

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
       WR++;
       okToRead.wait(&lock);
       WR--;
    }
    AR++;
    lock.release();

    // read-only
    AccessDbase(

    // check out
    lock.Acquire
    AR--;
    if (AR == 0 && WW > 0)
       okToWrite.broadcast();
    lock.Release();
  }
```

What if we turn signal to broadcast?

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
       WW++;
       okToWrite.wait(&lock);
       WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
       okToWrite.signal();
    } else if (WR > 0) {
       okToRead.broadcast();
    }
    lock.Release();
}
```

Read/Writer Revisited

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
       WR++;
       okContinue.wait(&lock);
       WR--;
    }
    AR++;
    lock.release();


    // read-only access
    AccessDbase(ReadOnly);


    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
       okContinue.signal();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
       WW++;
       okContinue.wait(&lock);
       WW--;
    }
    AW++;
    lock.release();


    // read/write access
    AccessDbase(ReadWrite);


    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
       okToWrite.signal();
    } else if (WR > 0) {
       okContinue.broadcast();
    }
    lock.Release();
}
```

What if we turn okToWrite and okToRead into okContinue?

Read/Writer Revisited

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
       WR++;
       okContinue.wait(&lock);
       WR--;
    }
    AR++;
    lock.release();


    // read-only access
    AccessDbase(ReadOnly);


    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
       okContinue.signal();
    lock.Release();
}
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
       WW++;
       okContinue.wait(&lock);
       WW--;
    }
    AW++;
    lock.release();


    // read/write access
    AccessDbase(ReadWrite);


    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
       okToWrite.signal();
    } else if (WR > 0) {
       okContinue.broadcast();
    }
    lock.Release();
}
```

- R1 arrives
- W1, R2 arrive while R1 reads
- R1 signals R2

Read/Writer Revisited

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
       WR++;
       okContinue.wait(&lock);
       WR--;
    }
    AR++;
    lock.release();


    // read-only access
    AccessDbase(ReadOnly);


    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
       okContinue.broadcast();
    lock.Release();
  }
```

```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
       WW++;
       okContinue.wait(&lock);
       WW--;
    }
    AW++;
    lock.release();


    // read/write access
    AccessDbase(ReadWrite);


    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
       okToWrite.signal();
    } else if (WR > 0) {
       okContinue.broadcast();
    }
    lock.Release();
}
```

Need to change to broadcast!
Why?

Deadlock

Four requirements for Deadlock

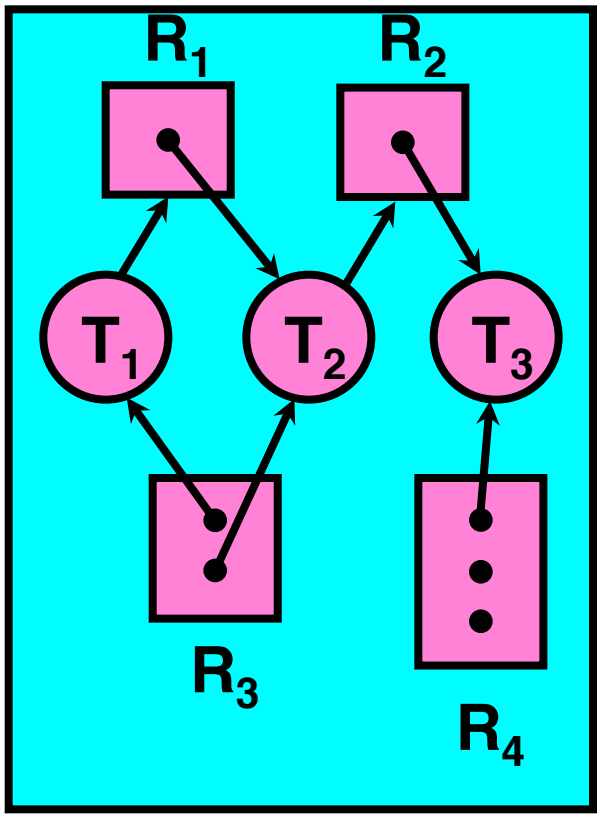
- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **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

- Circular wait: Sp11, #3

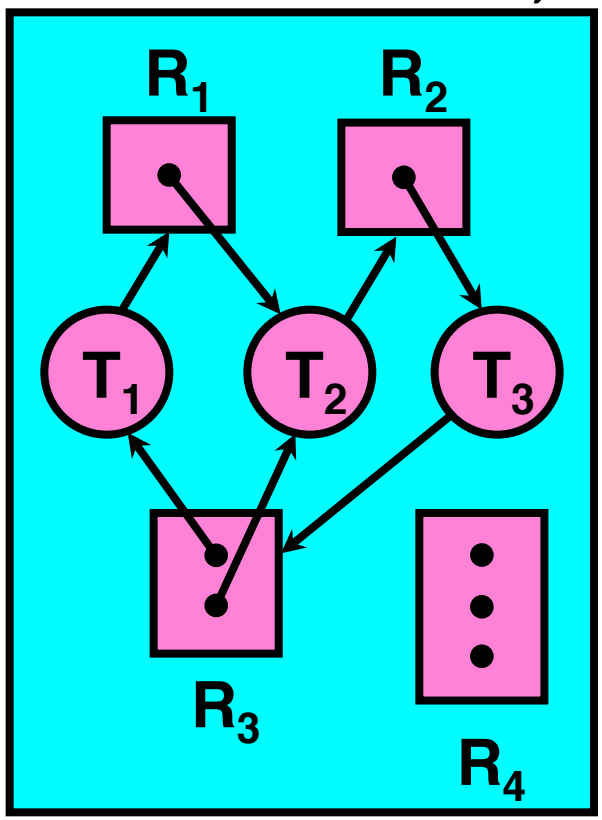
- Livelock: State constantly changing, but still no progress made. E.g. A system that deadlocks, then detects the deadlock, so it restarts everyone. And then they all deadlock again, over and over.
- Livelock and deadlock both cause starvation.

Resource Allocation Graph Examples

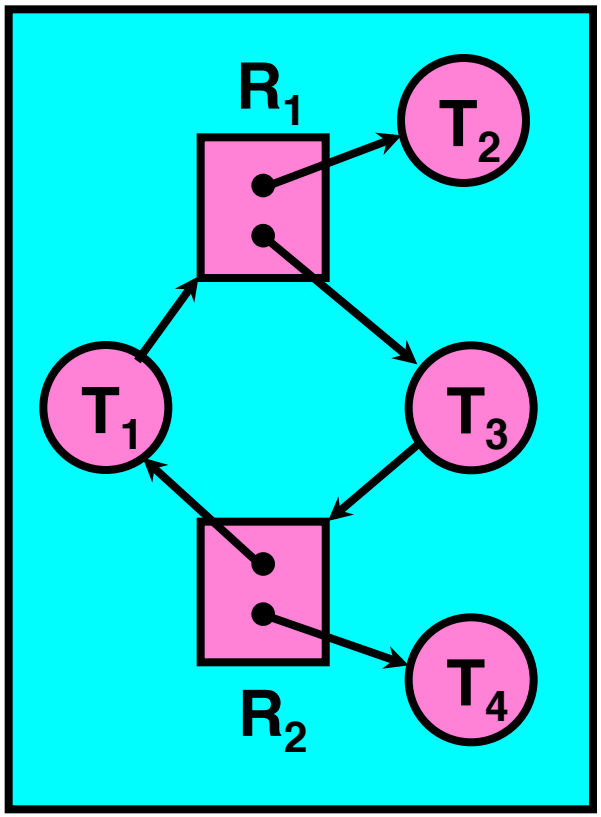
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

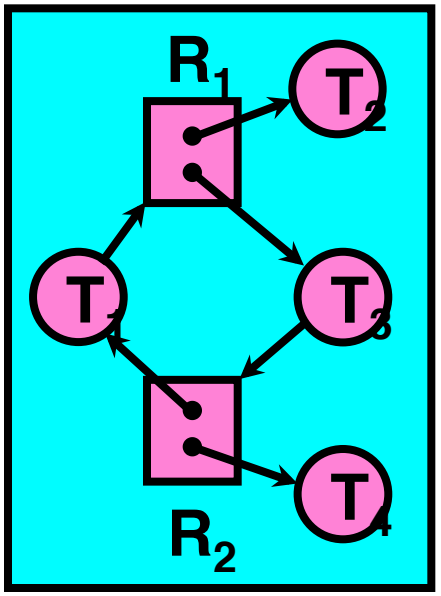
Deadlock Detection Algorithm

- 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

- Example: Sp11, #2

Banker's algorithm

- Method of *deadlock avoidance*.
- Process specifies max resources it will ever request. Process can initially request less than max and later ask for more. Before a request is granted, the Banker's algorithm checks that the system is "safe" if the requested resources are granted.
- Roughly: Some process could request it's specified Max and eventually finish; when it finishes, that process frees up its resources. A system is safe if there exists some sequence by which processes can request their max, terminate, and free up their resources so that all other processes can finish (in the same way).
- Example: Sp04 midterm, problem 4

Banker's Algorithm for *Deadlock Avoidance*

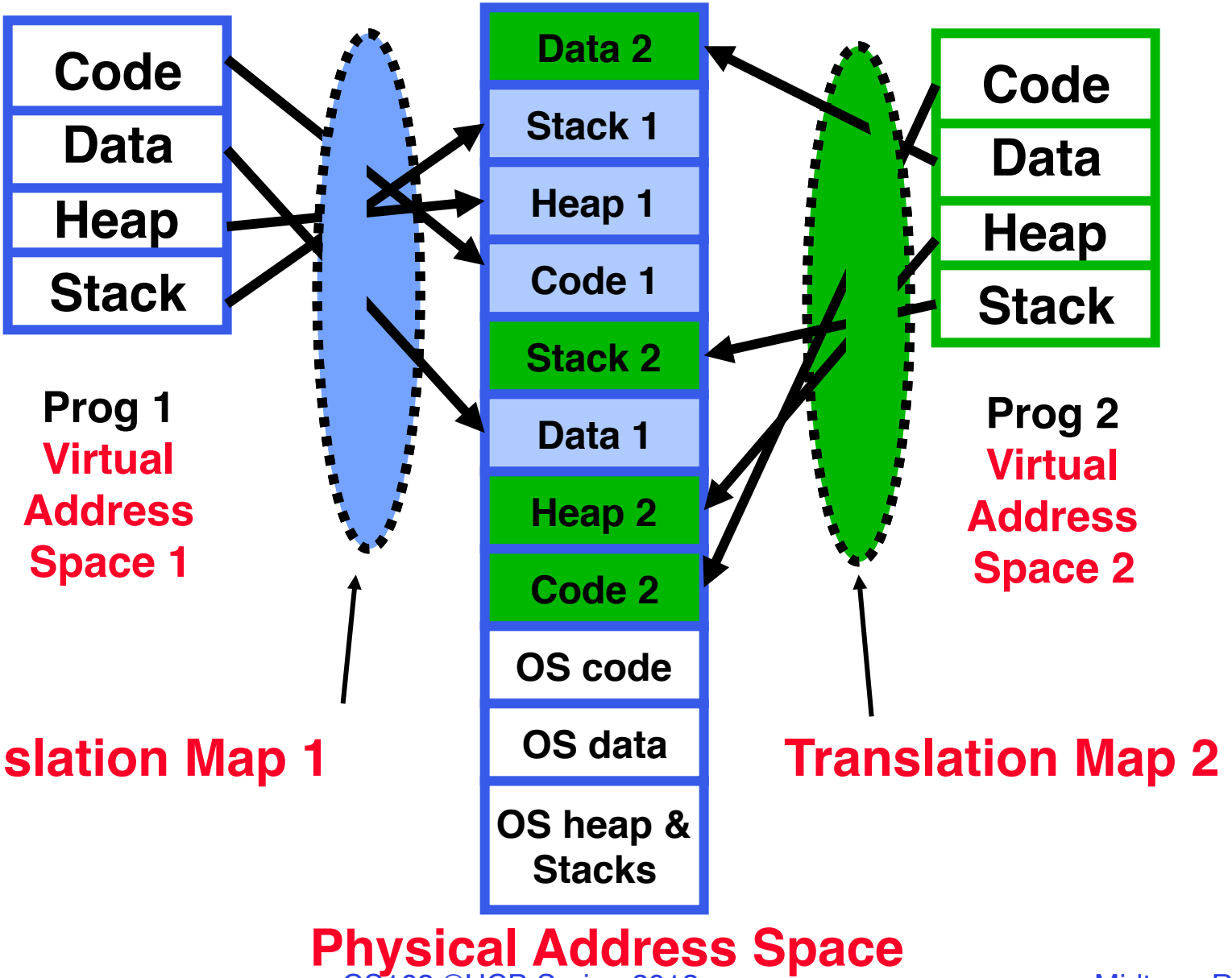
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}] \leq [Avail]$) 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

Memory Multiplexing, Address Translation

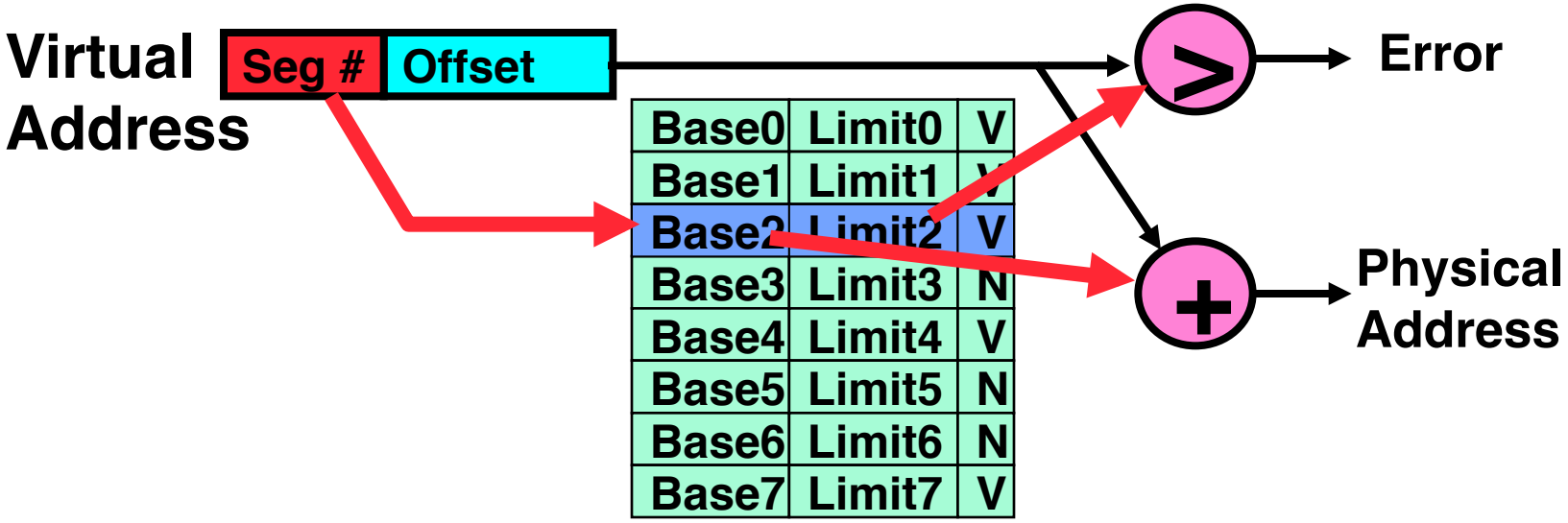
Important Aspects of Memory Multiplexing

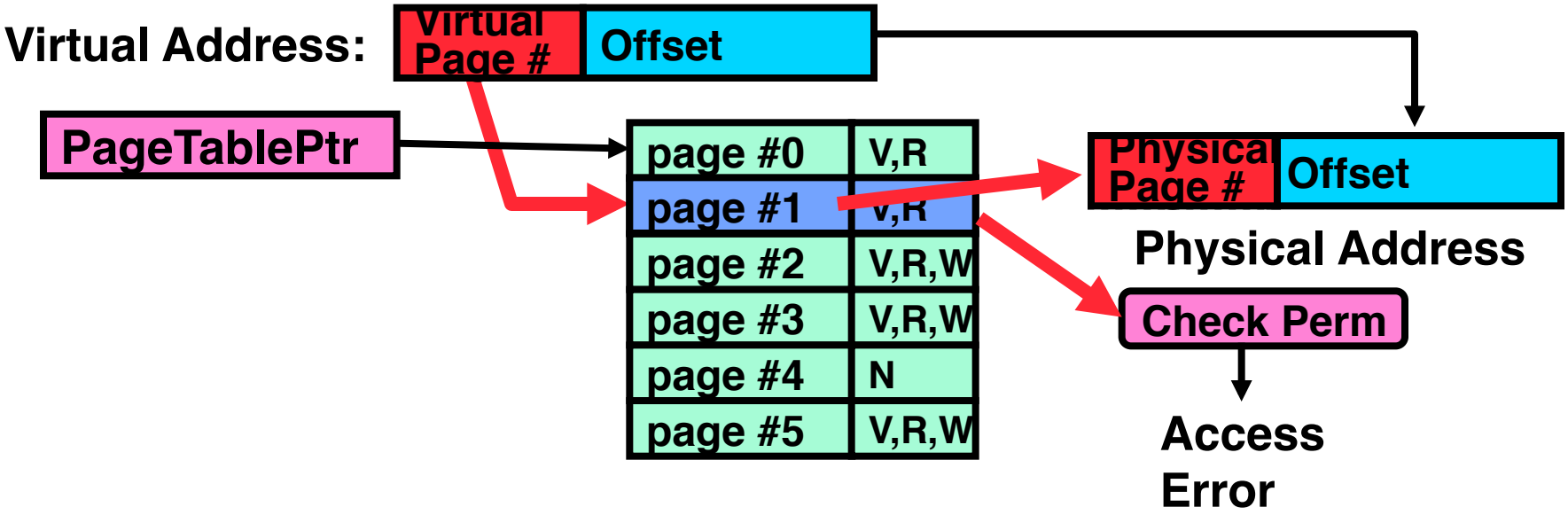
- Controlled overlap:
  - Processes should not collide in physical memory
  - Conversely, would like the ability to share memory when desired (for communication)
- Protection:
  - Prevent access to private memory of other processes
    - » Different pages of memory can be given special behavior (Read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

Why Address Translation?

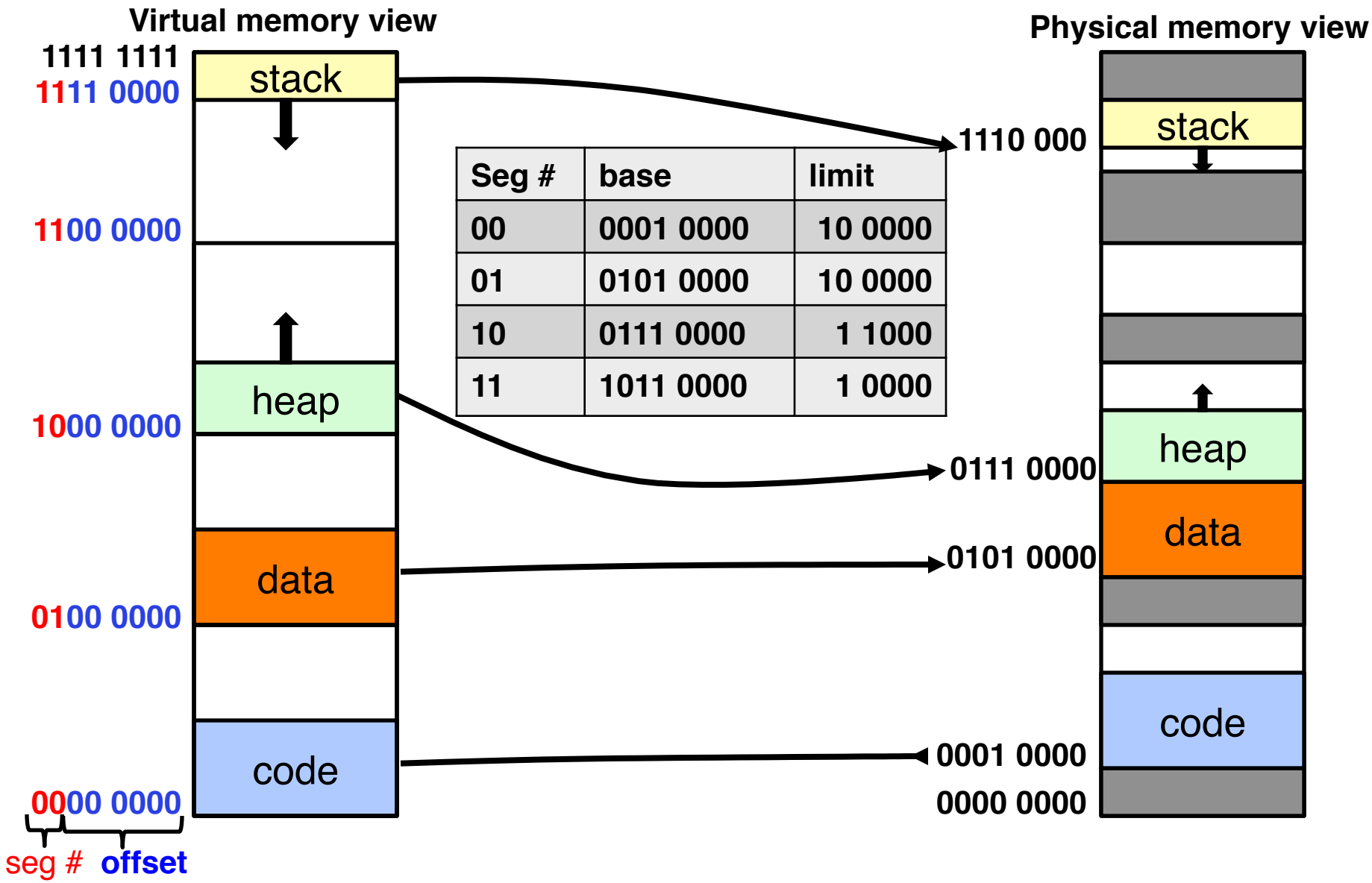


Addr. Translation: Segmentation vs. Paging



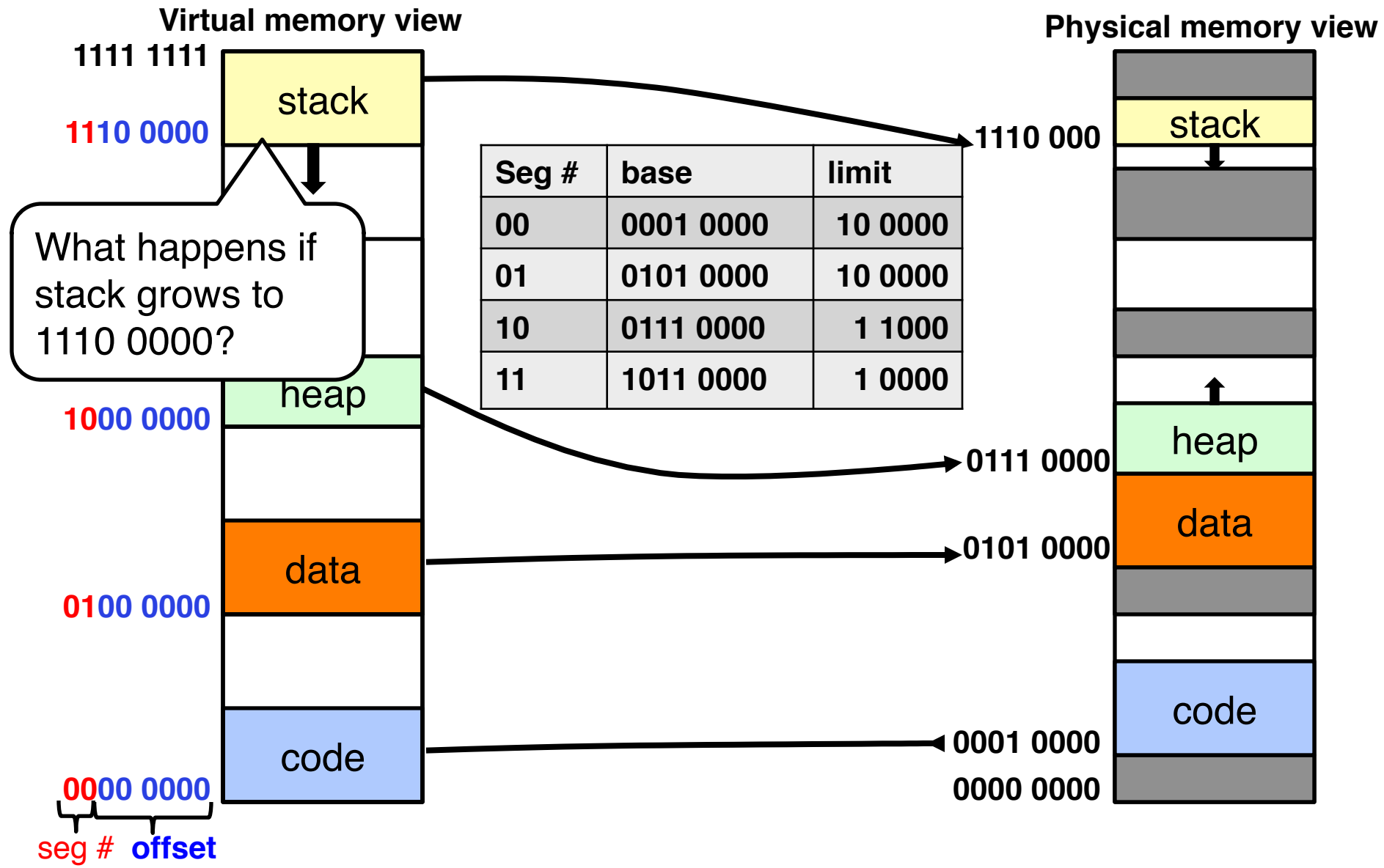


Review: Address Segmentation

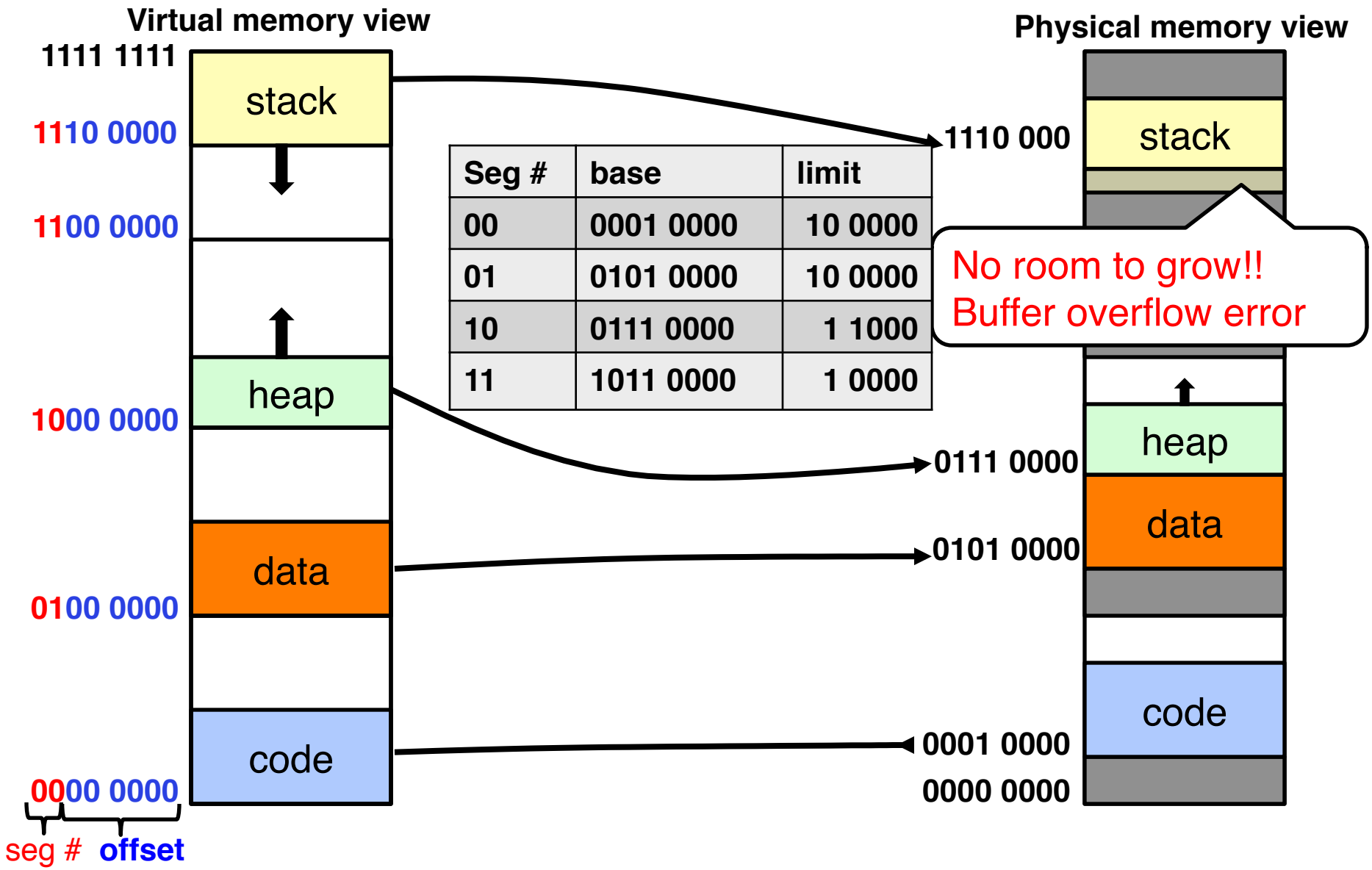


Seg #	base	limit
00	0001 0000	10 0000
01	0101 0000	10 0000
10	0111 0000	1 1000
11	1011 0000	1 0000

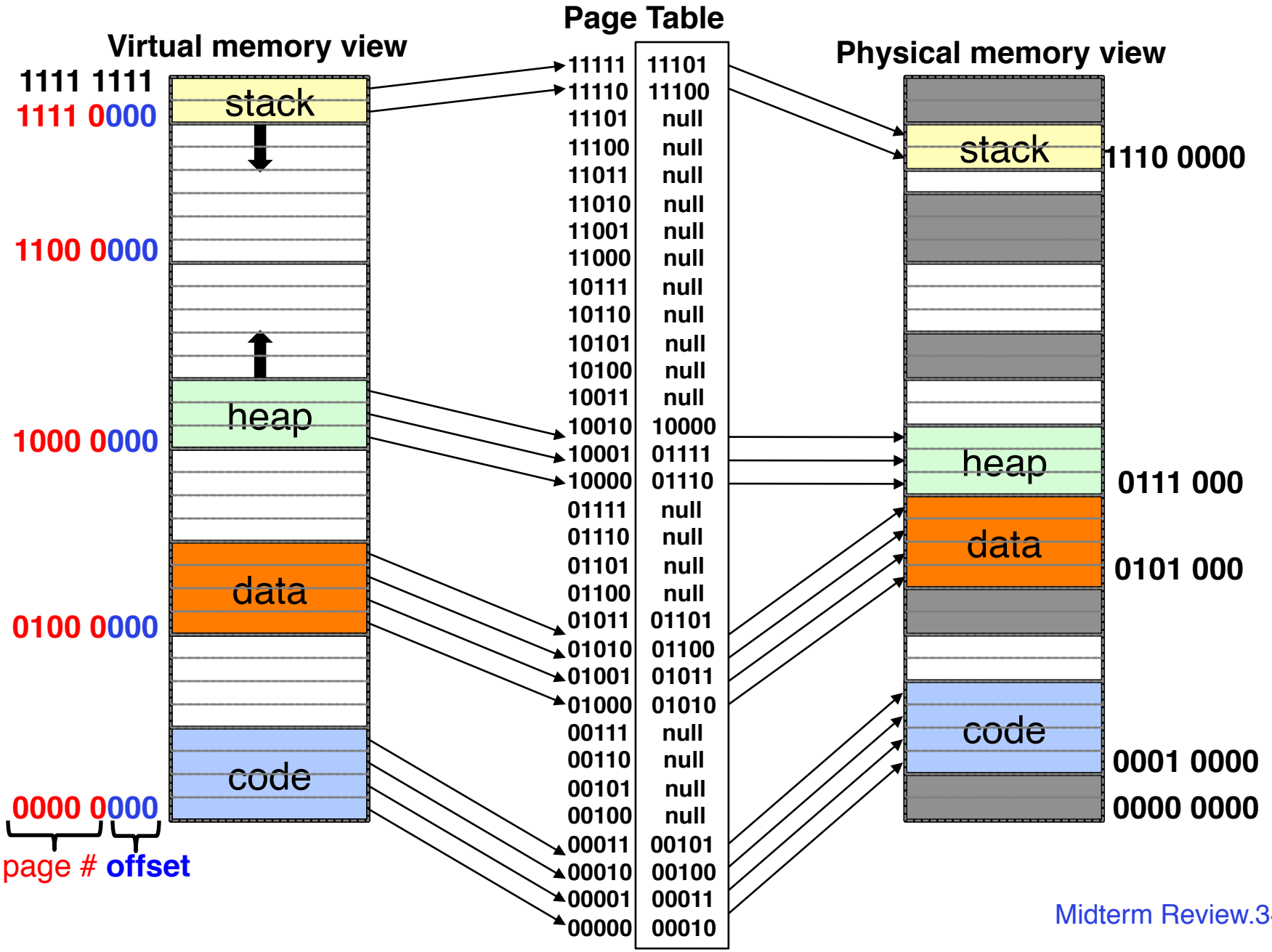
Review: Address Segmentation



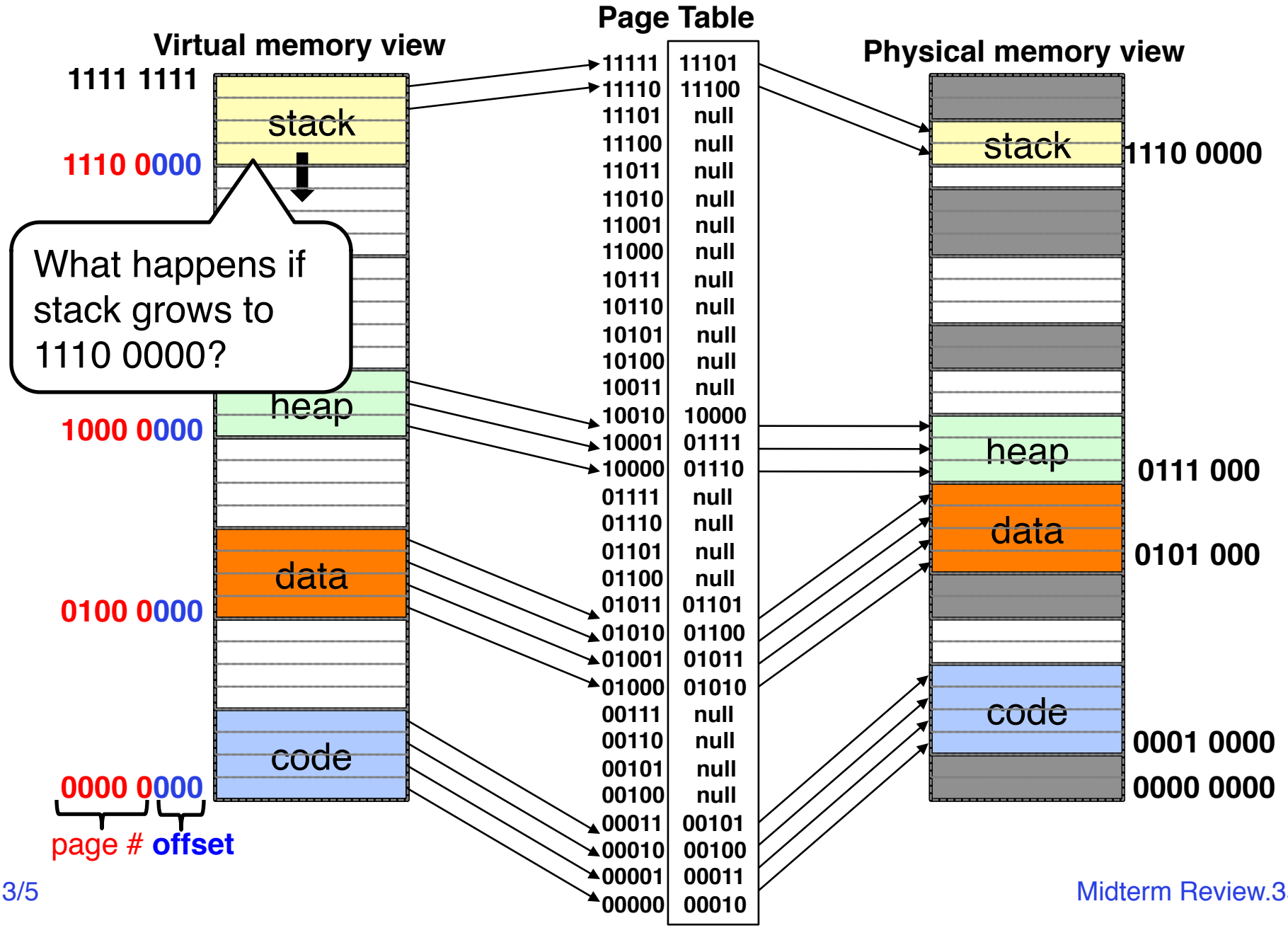
Review: Address Segmentation



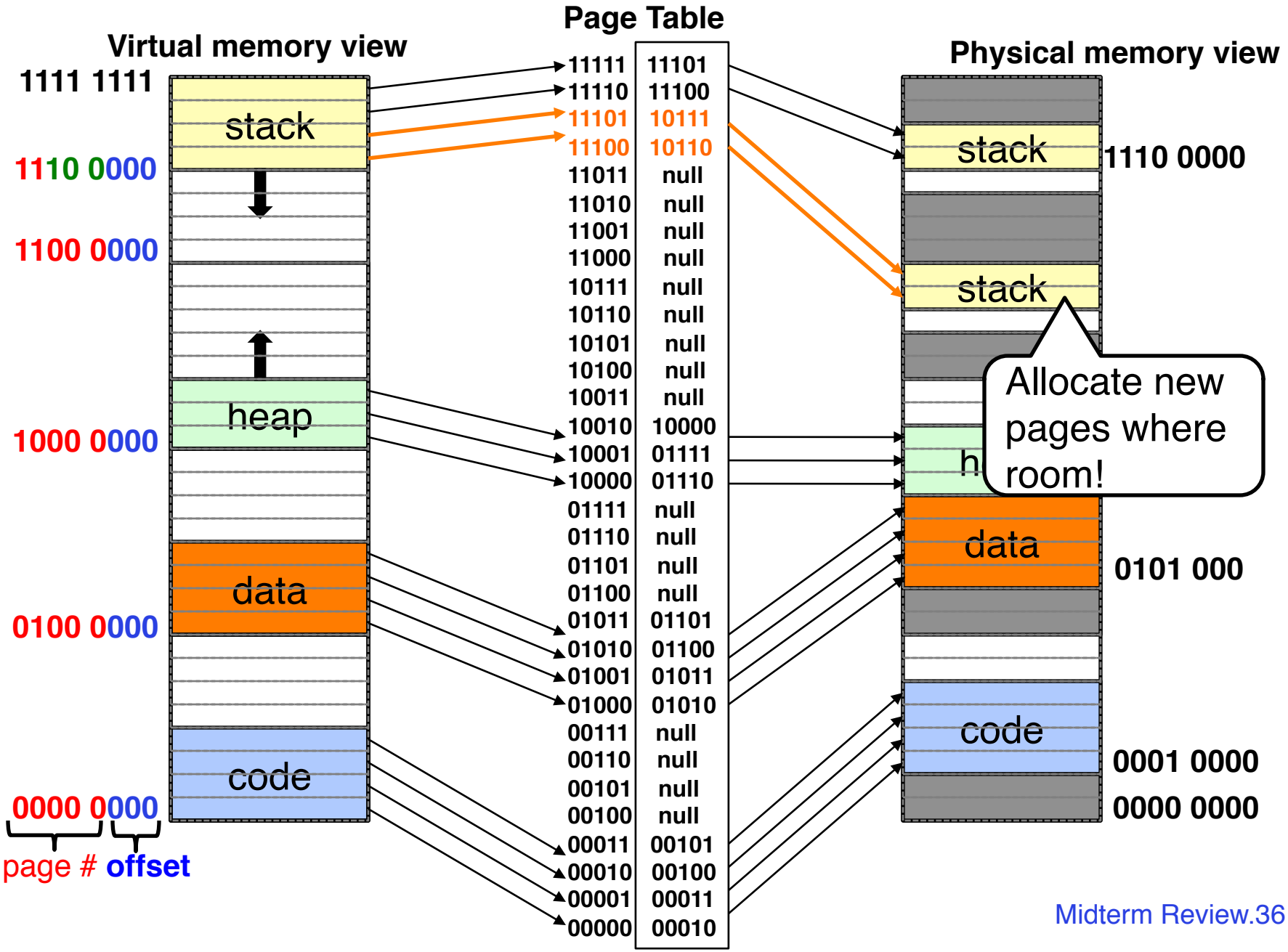
Review: Paging



Review: Paging



Review: Paging



Design a Virtual Memory system

Would you choose
Segmentation or Paging?

Design a Virtual Memory system

How would you choose your Page Size?

Design a Virtual Memory system

How would you choose your Page Size?

- Page Table Size
- TLB Size
- Internal Fragmentation
- Disk Access

Design a Virtual Memory system

You have a 32 bit virtual and physical memory and 2 KB pages.

How many bits would you use for the index and how many for the offset?

Design a Virtual Memory system

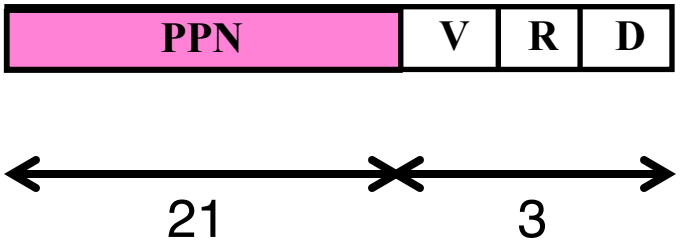
You have a 32 bit virtual memory and 2 KB pages.

Design a Virtual Memory system

How many bytes are required for a page table entry?

Design a Virtual Memory system

How many bytes are required for a page table entry?

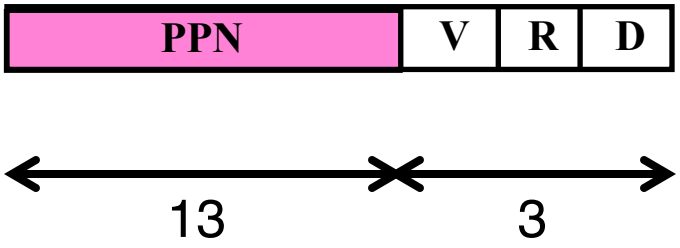


Design a Virtual Memory system

If you only had 16 MB of physical memory, how many bytes are required for a page table entry?

Design a Virtual Memory system

If you only had 16 MB of physical memory, how many bytes are required for a page table entry?

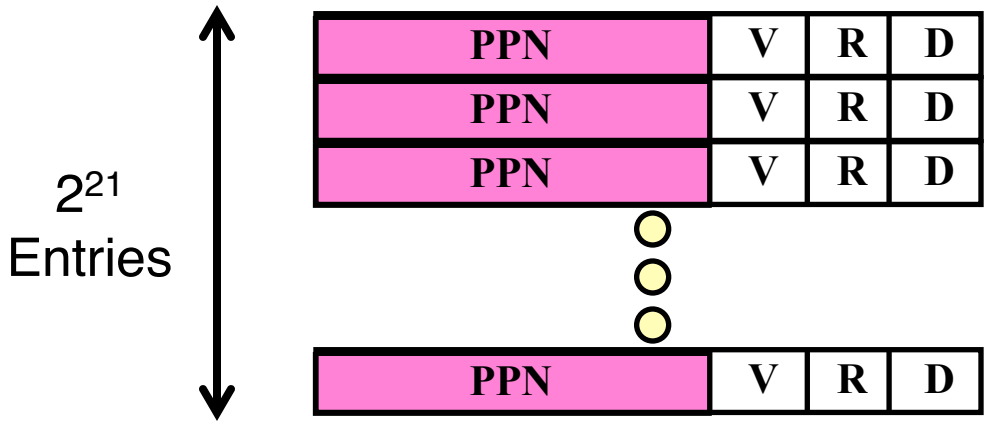


Design a Virtual Memory system

How large would the page table be?

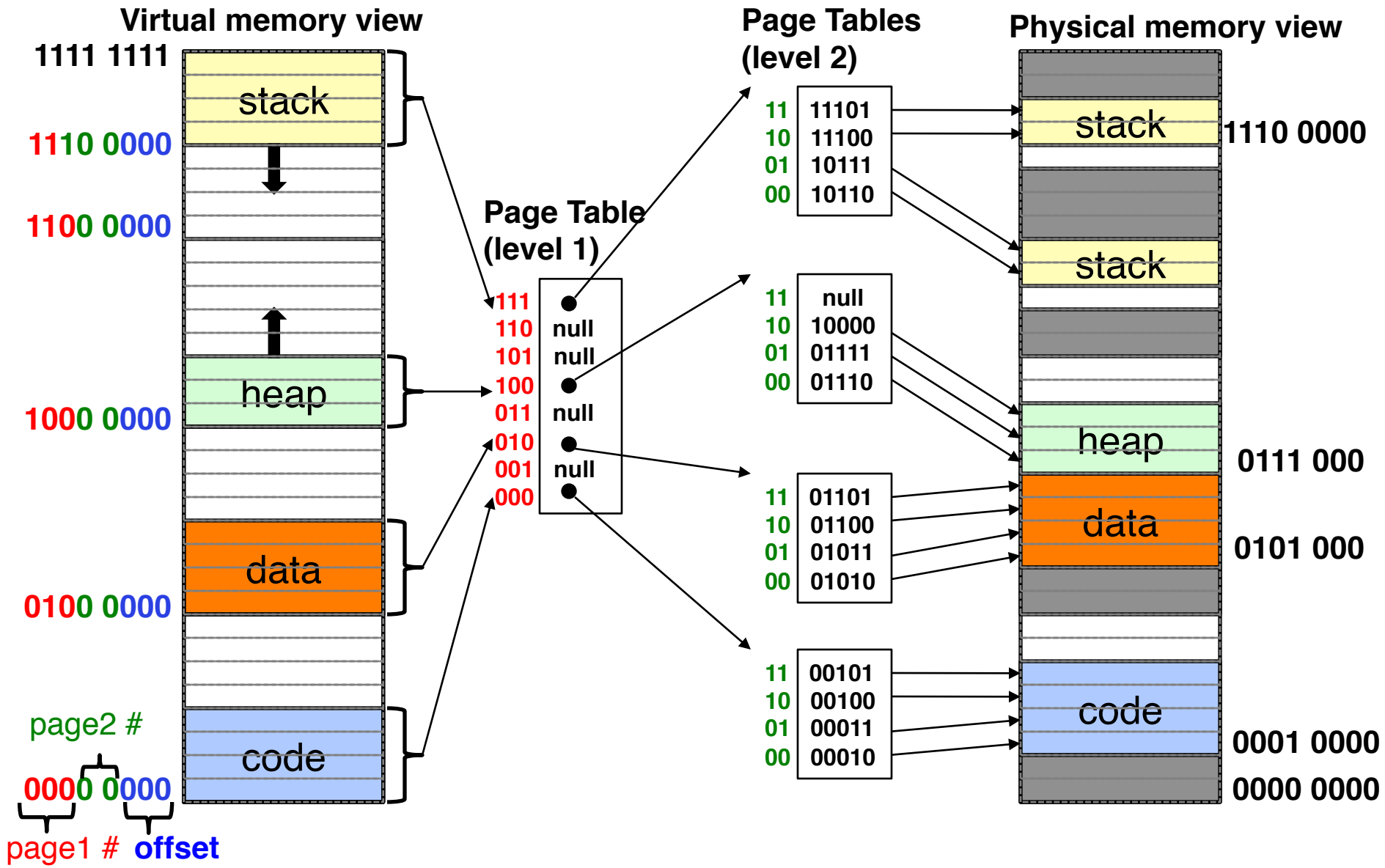
Design a Virtual Memory system

How large would the page table be?

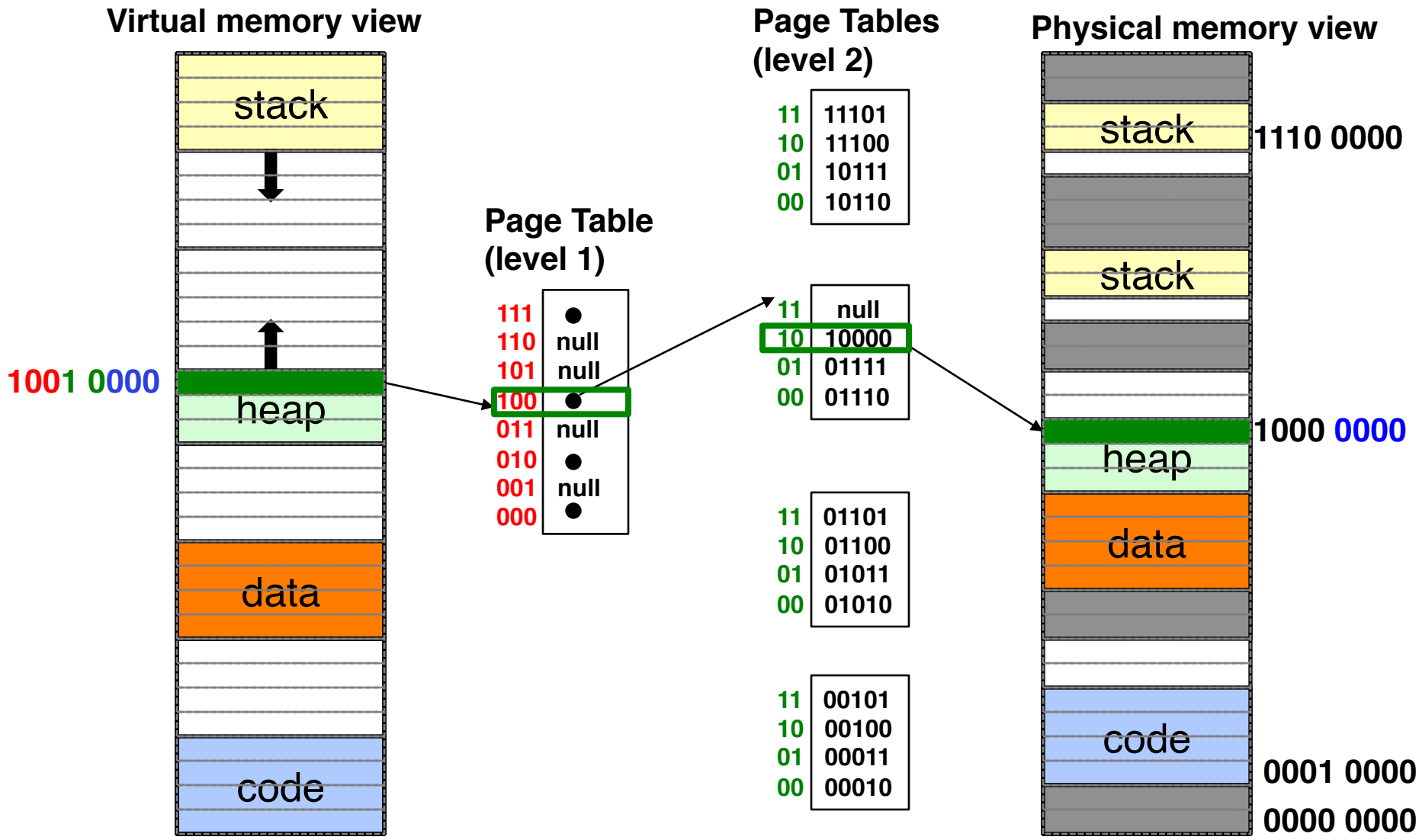


Total Size: 2^{21} x 2 B = 4MB

Review: Two-Level Paging



Review: Two-Level Paging

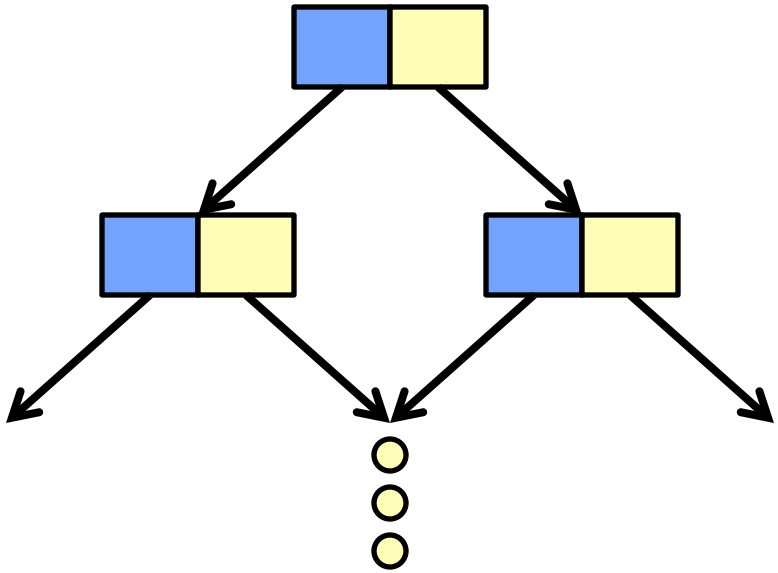


Design a Virtual Memory system

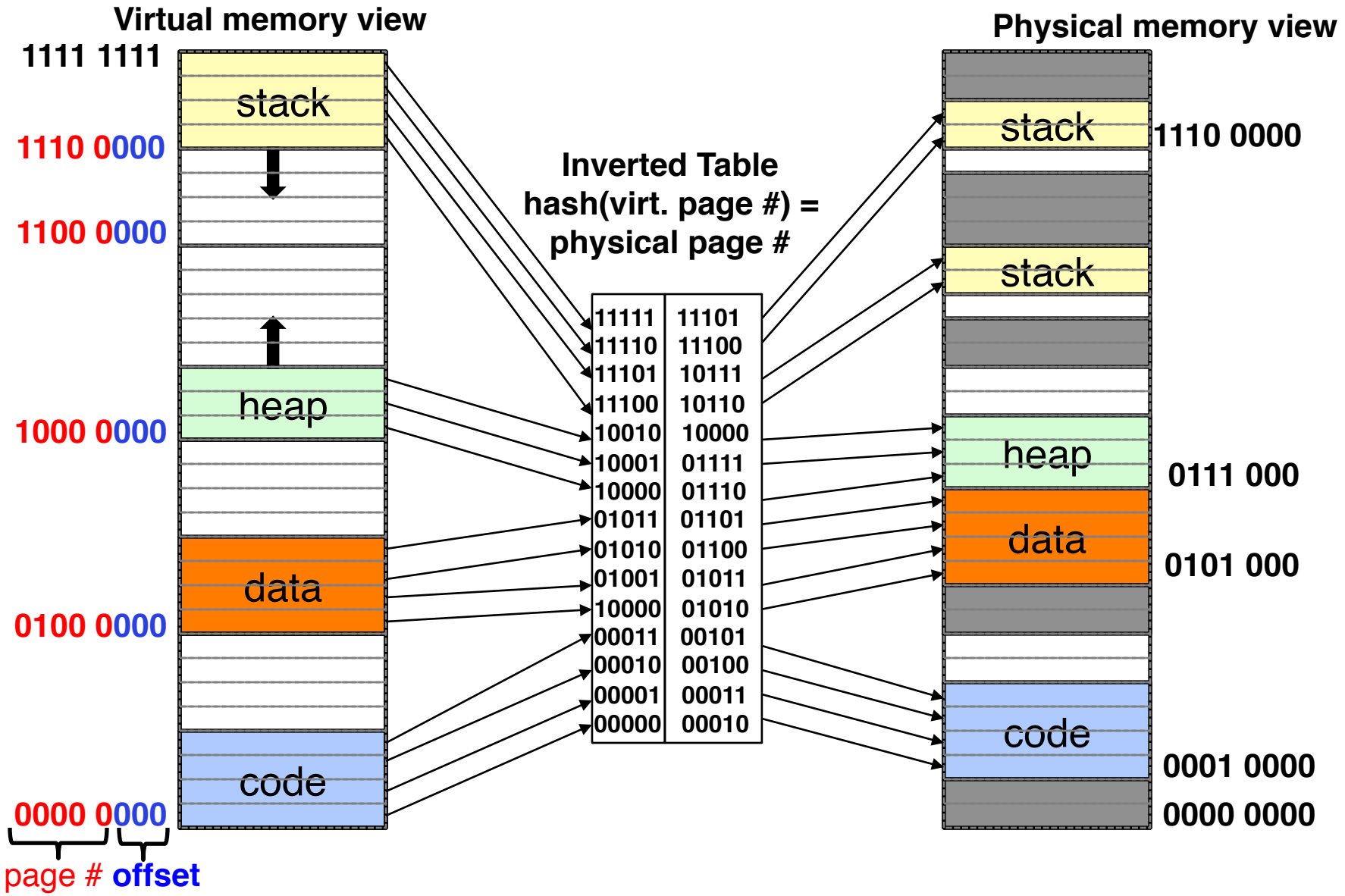
You have the same 32 bit virtual address space with 2 KB pages.

If you had a full 21-level page table. How much memory would be consumed in total?

Design a Virtual Memory system



Review: Inverted Table



Address Translation Comparison

	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation	•Large size: Table size ~ virtual memory •Internal fragmentation
Paged segmentation	•No external fragmentation •Table size ~ memory used by program	•Multiple memory references per page access •Internal fragmentation
Two-level pages		
Inverted Table		Hash function more complex

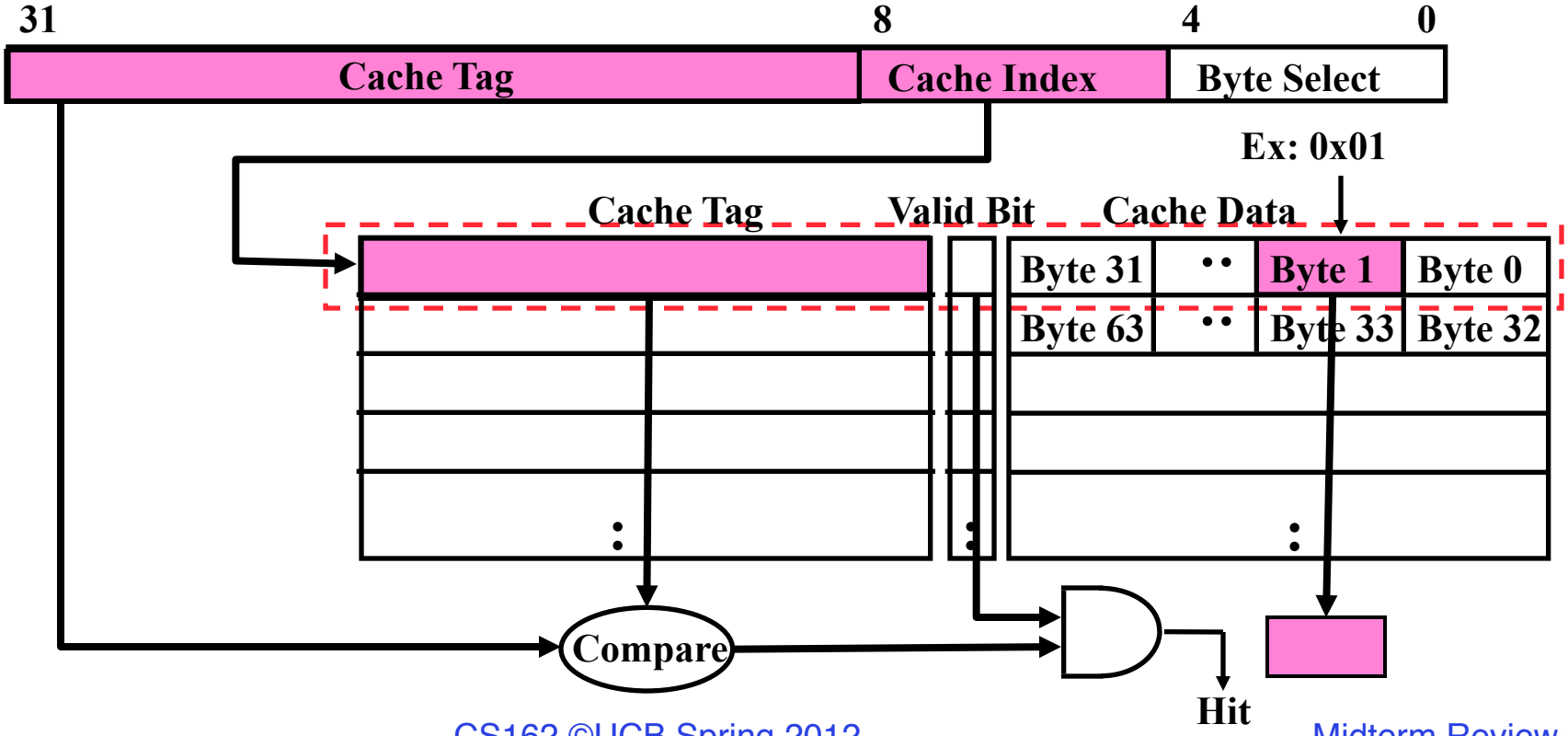
Caches, TLBs

Review: Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

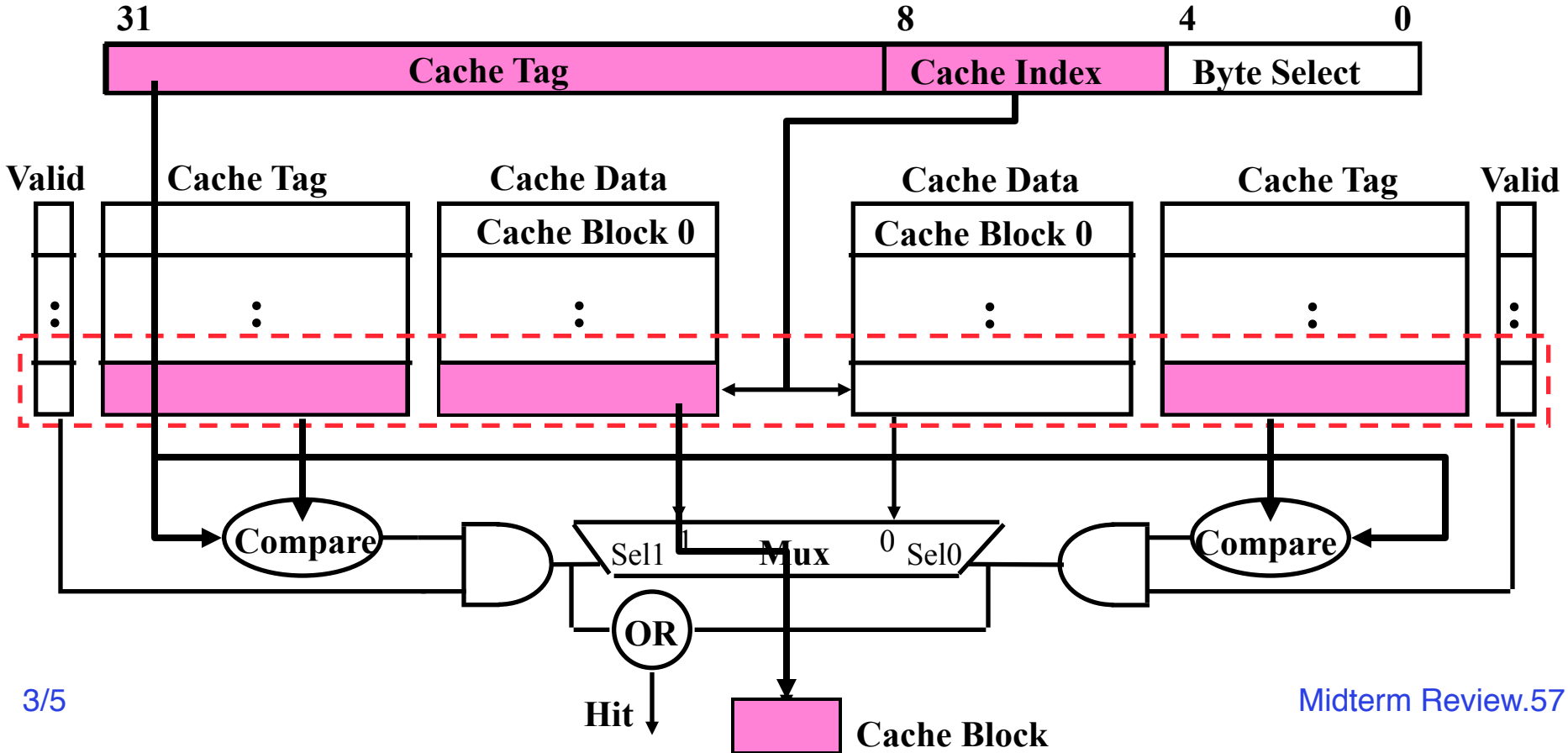
Direct Mapped Cache

- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same "cache tag" shares the same cache entry
  - Conflict misses



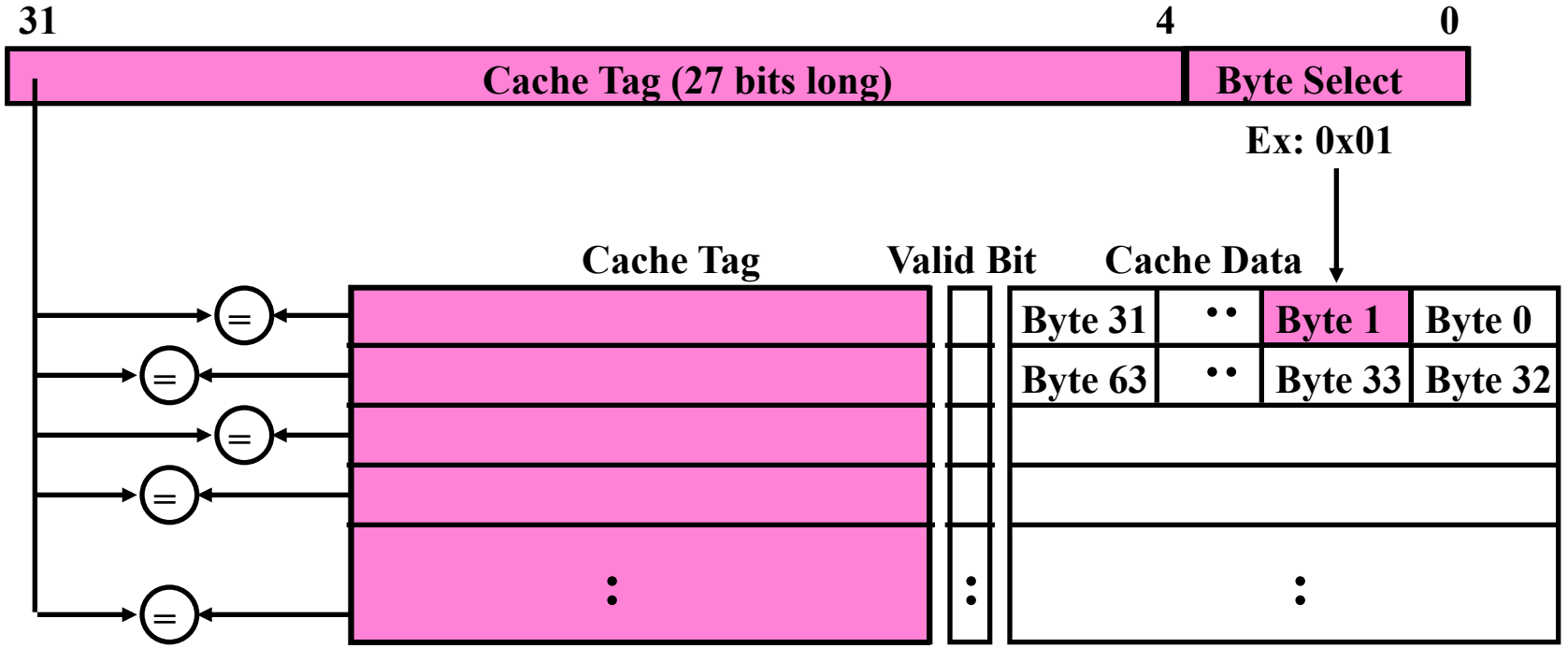
Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



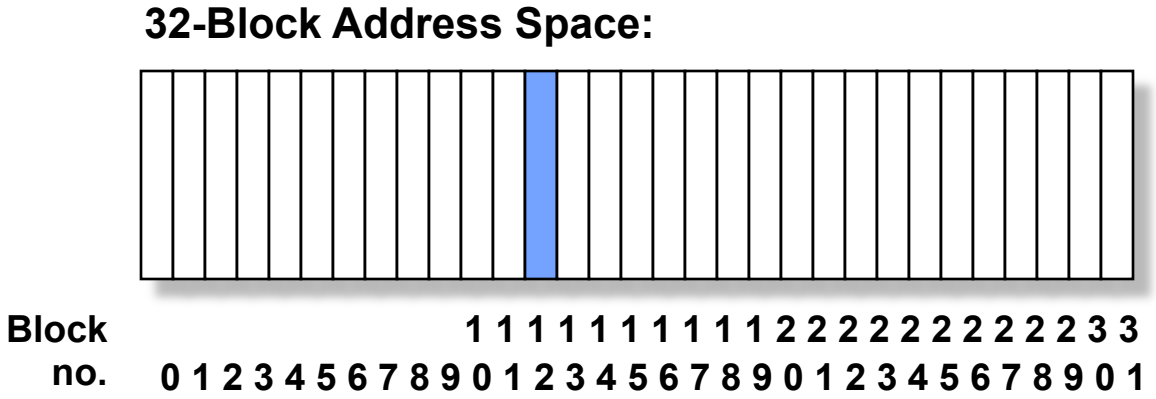
Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block

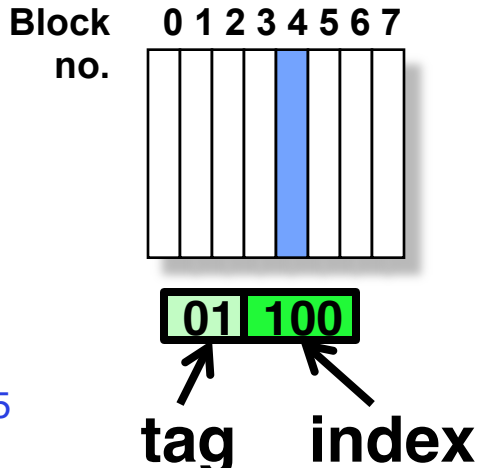


Where does a Block Get Placed in a Cache?

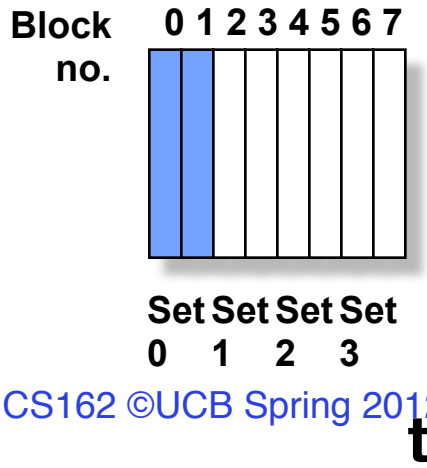
- Example: Block 12 placed in 8 block cache



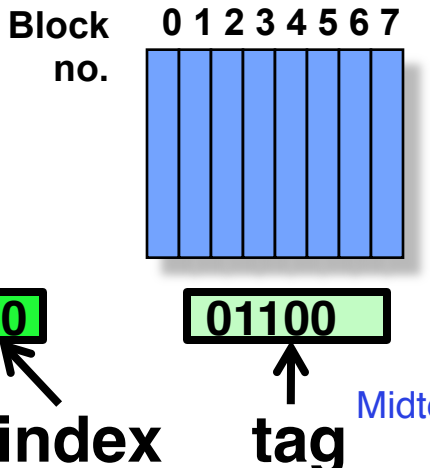
Direct mapped:
block 12 (01100) can go only into block 4 (12 mod 8)



Set associative:
block 12 can go anywhere in set 0 (12 mod 4)

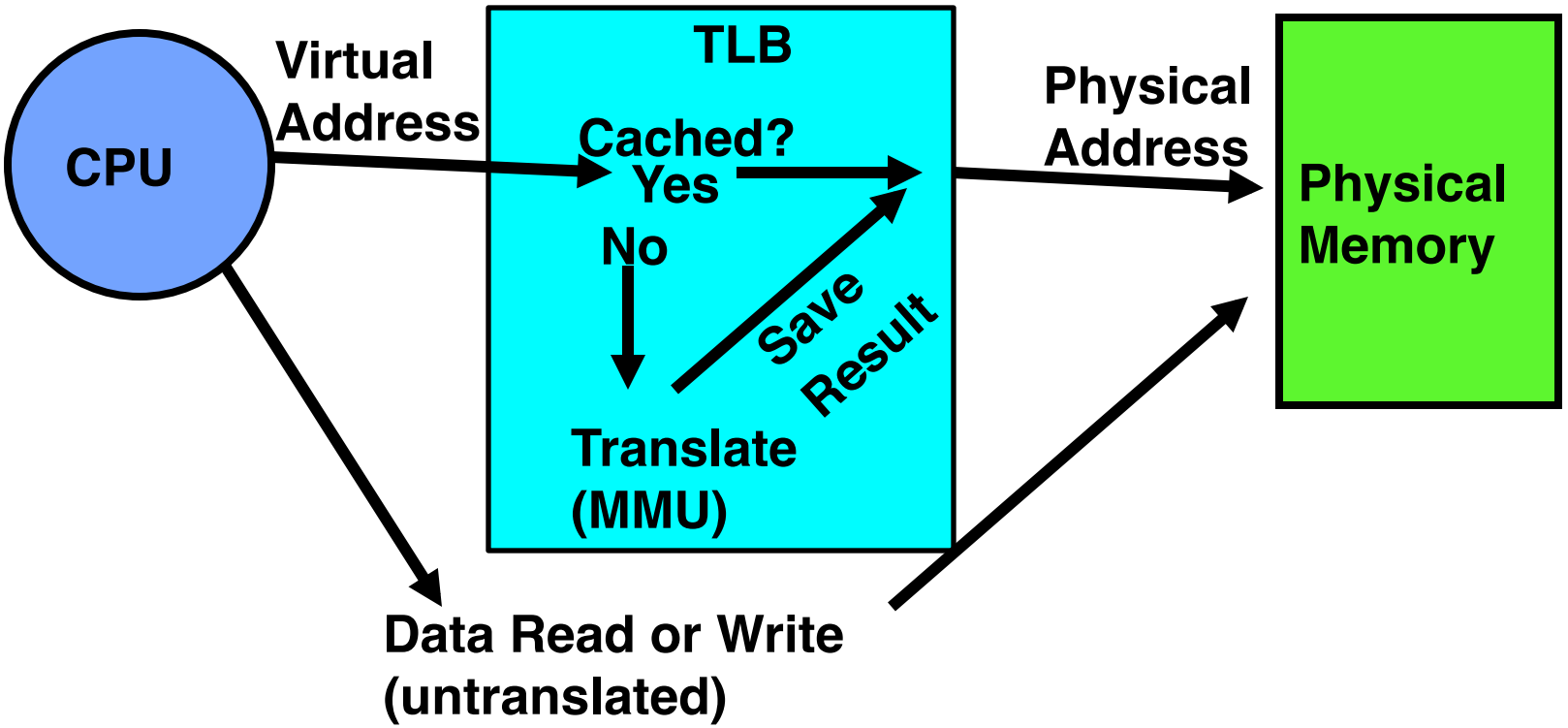


Fully associative:
block 12 can go anywhere



Review: Caching Applied to Address Translation

- Problem: address translation expensive (especially multi-level)
- Solution: cache address translation (TLB)
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…

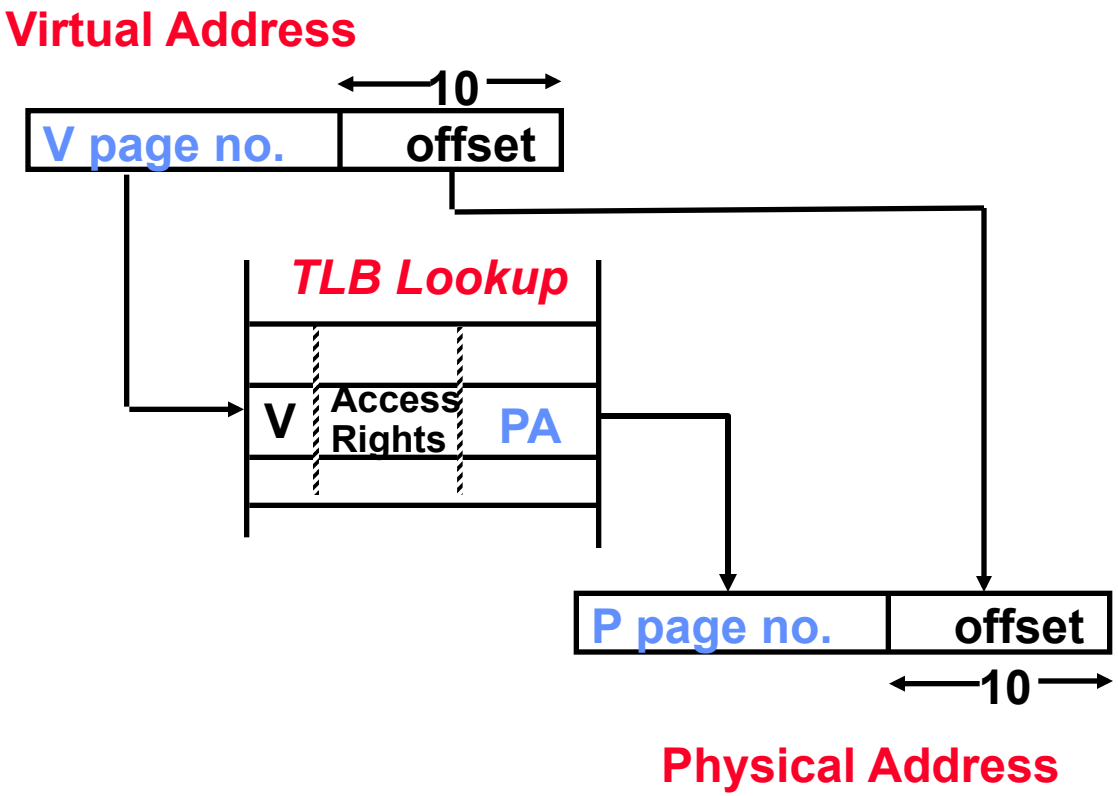


TLB organization

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a “TLB Slice”
- When does TLB lookup occur?
  - Before cache lookup?
  - In parallel with cache lookup?

Reducing translation time further

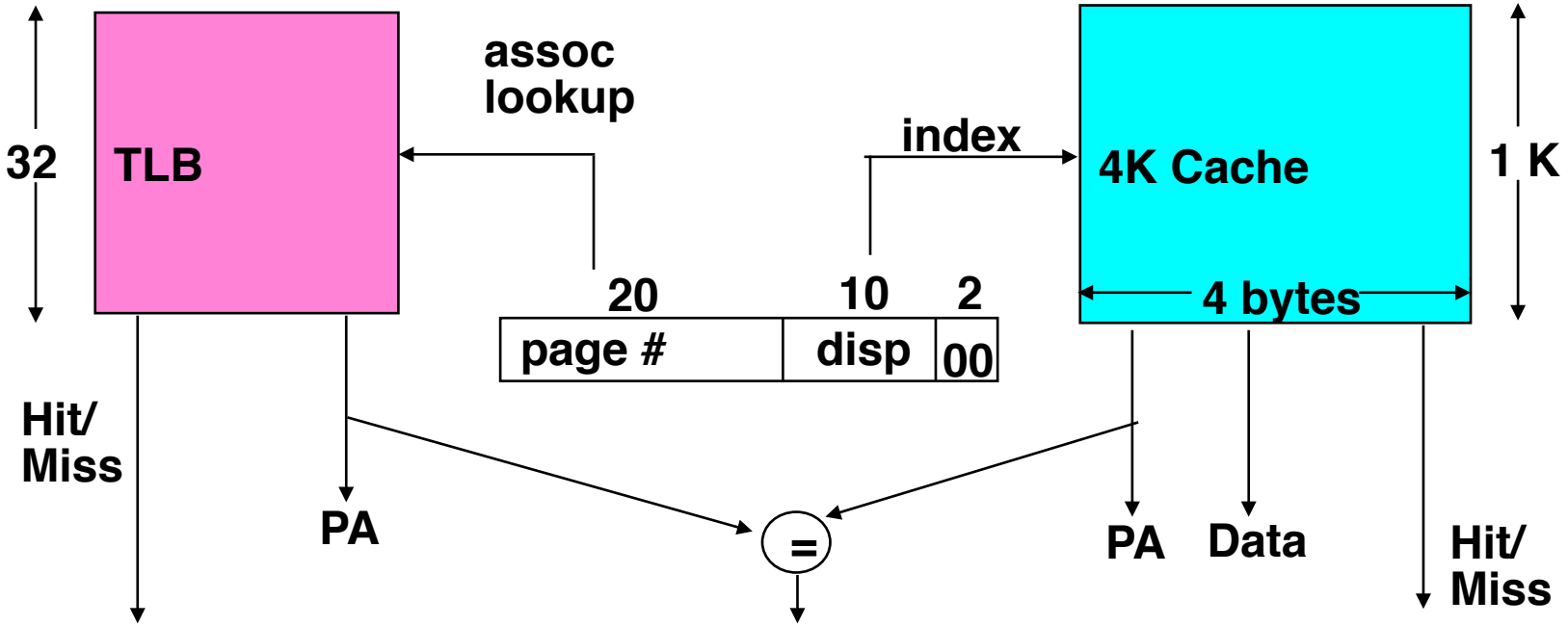
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

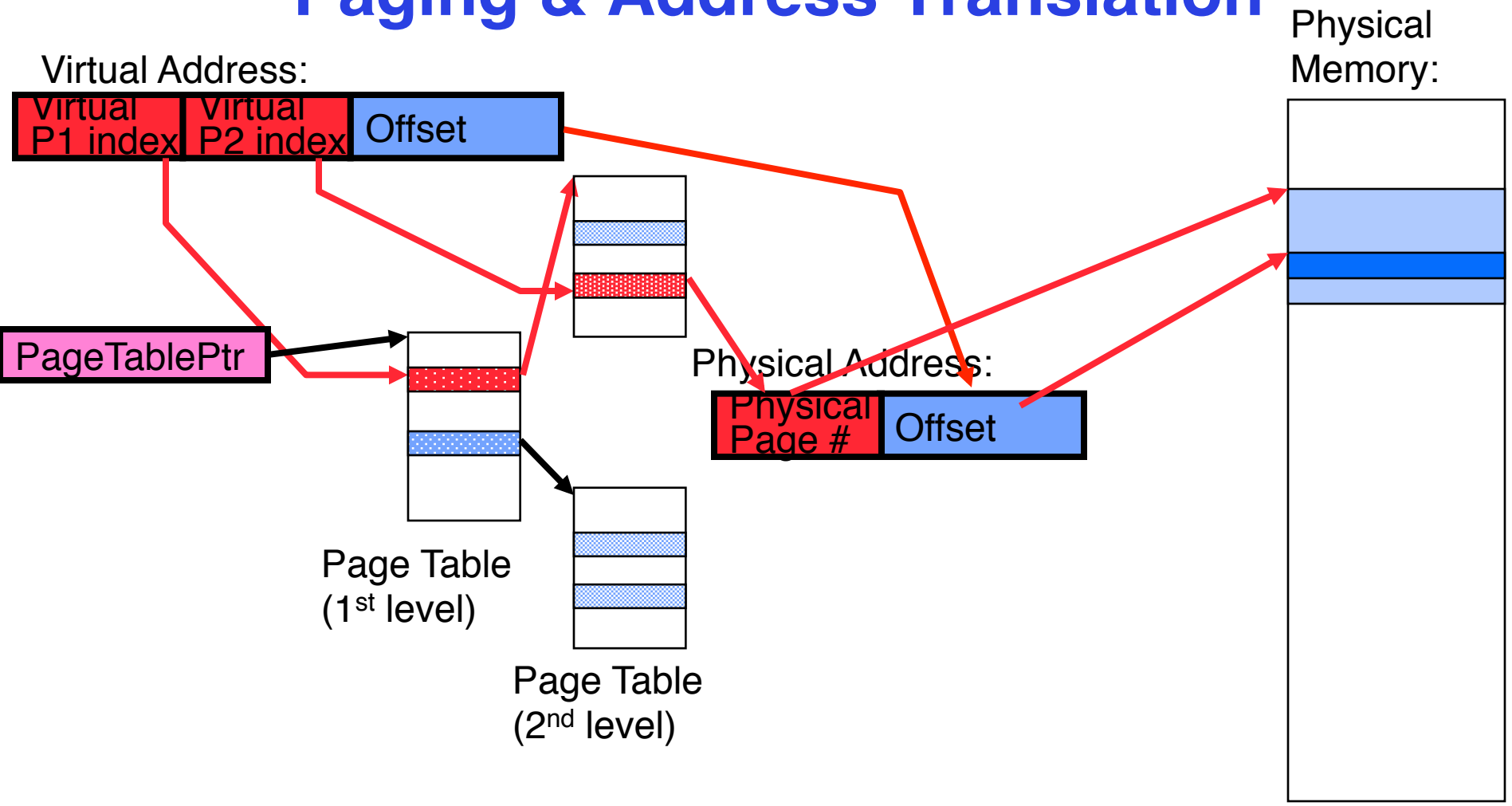
- Here is how this might work with a 4K cache:



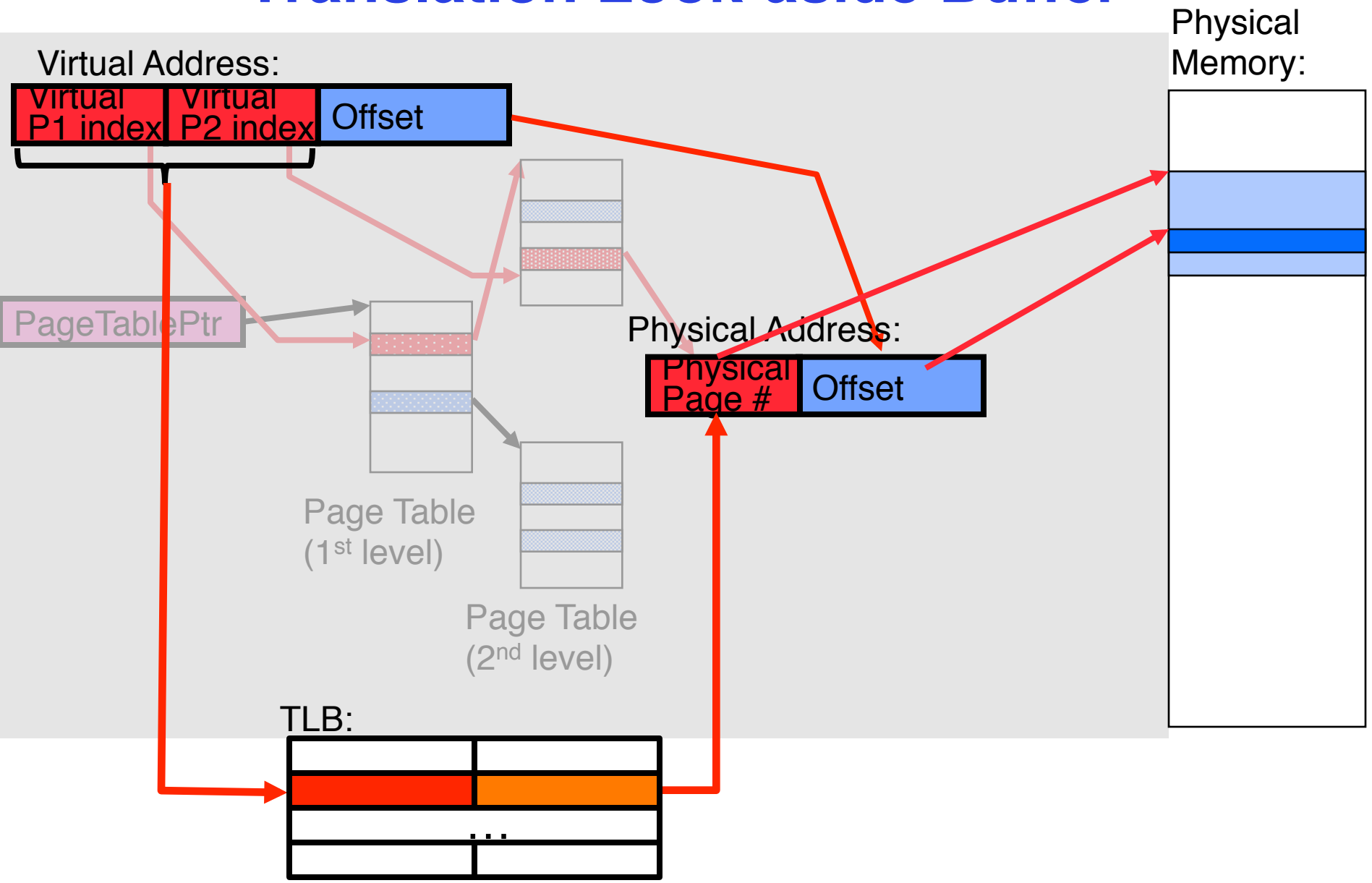
- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

Putting Everything Together

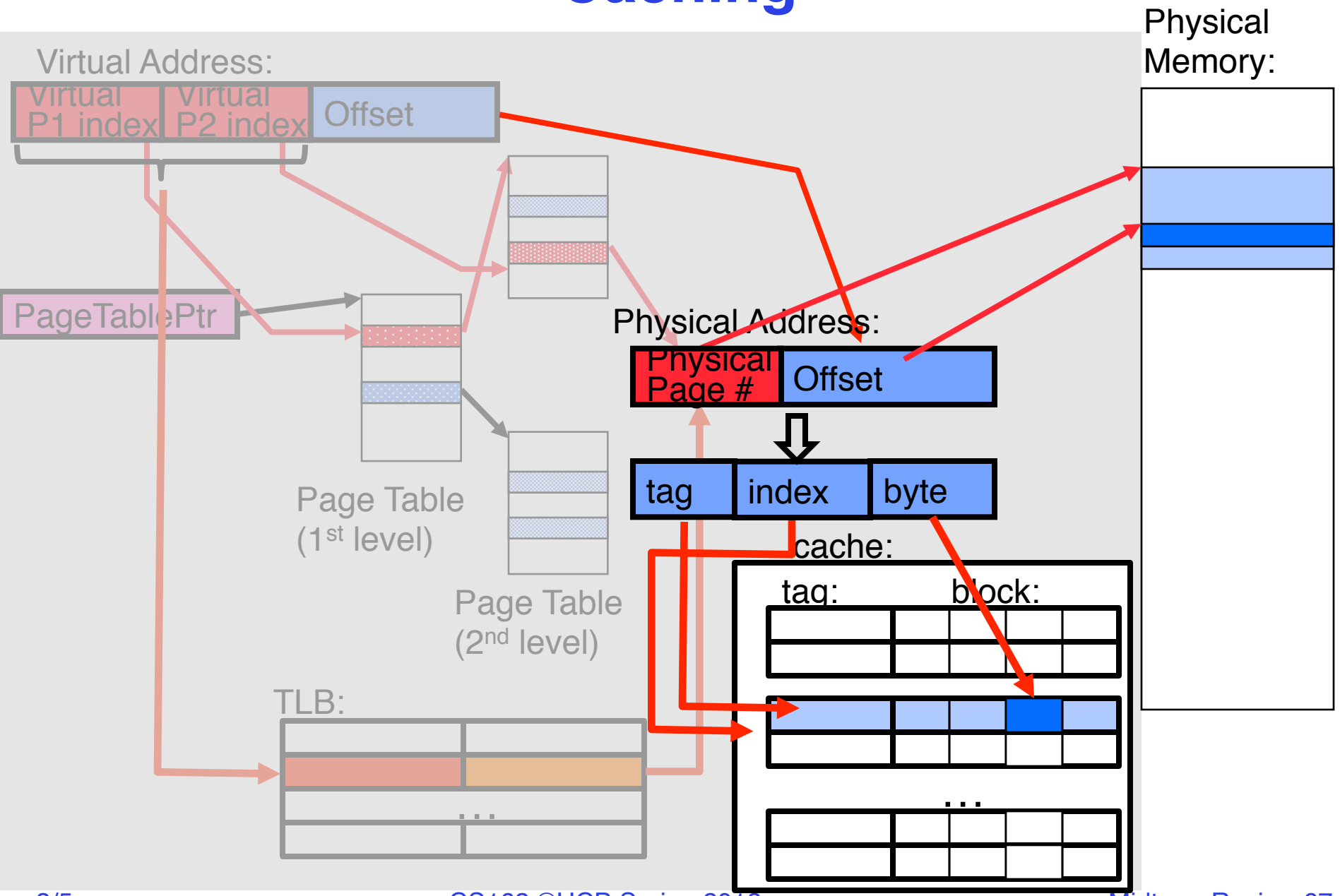
Paging & Address Translation



Translation Look-aside Buffer



Caching


Physical Memory:
Virtual Address:
Virtual P1 index
Virtual P2 index
Offset
PageTablePtr
Page Table (1st level)
Page Table (2nd level)
Physical Address:
Physical Page #
Offset
tag
index
byte
cache:
tag:
block:
TLB: