# CS162
# Operating Systems and Systems Programming
# Lecture 15

## Deadlock (cont'd) + Memory Management (cont'd)

October 22nd, 2024

Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

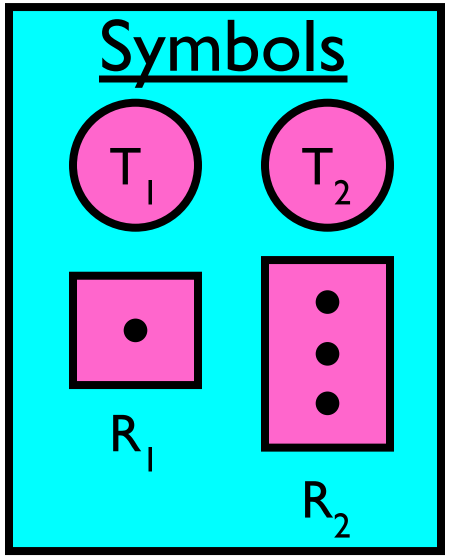
# Four requirements for occurrence of Deadlock (cont'd)

- 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 $T_1$

# Detecting Deadlock: Resource-Allocation Graph

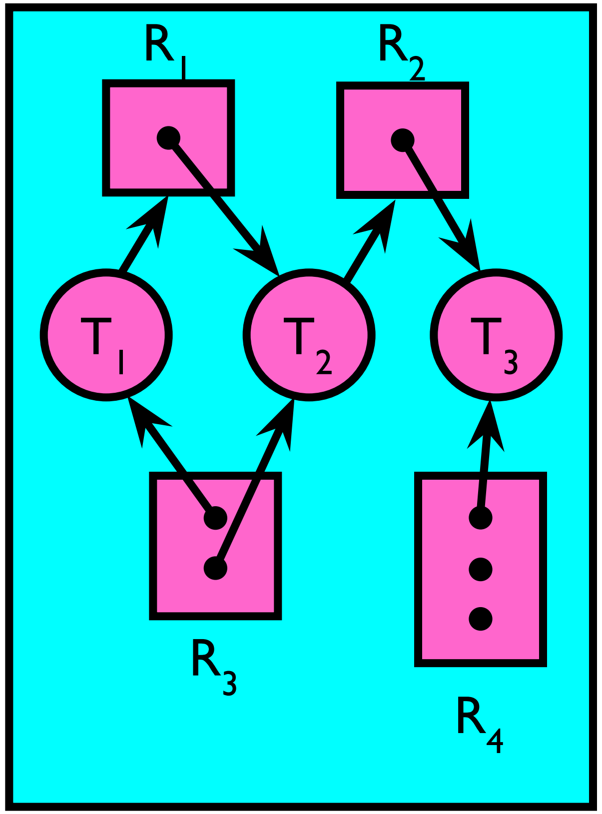
- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

    *CPU cycles, memory space, I/O devices*
  - Each resource type $R_i$ has $W_i$ instances
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

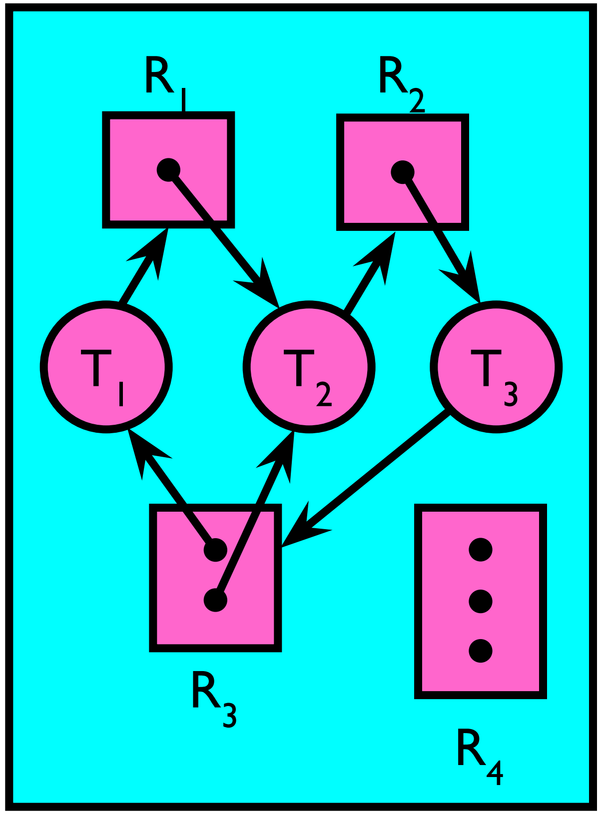


# Resource-Allocation Graph Examples

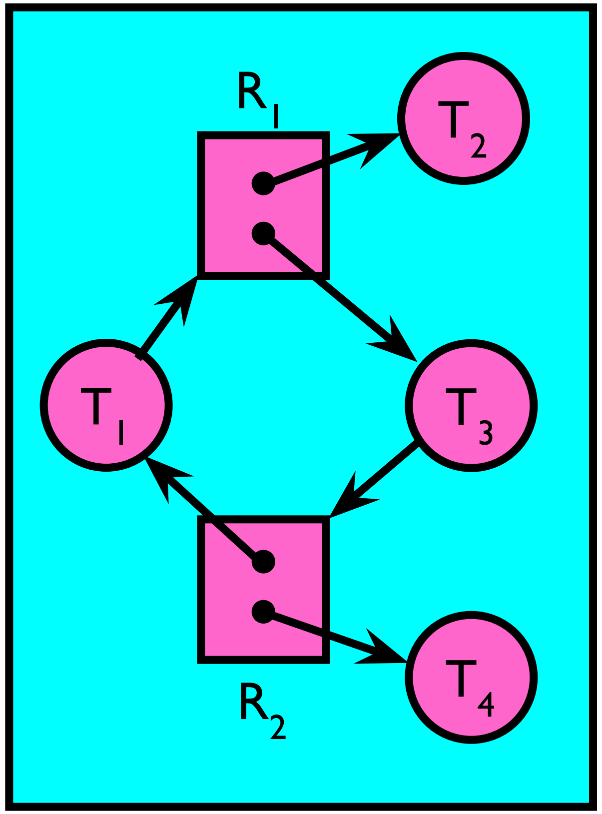
- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$




Simple Resource Allocation Graph




Allocation Graph With Deadlock




Allocation Graph With Cycle, but No Deadlock

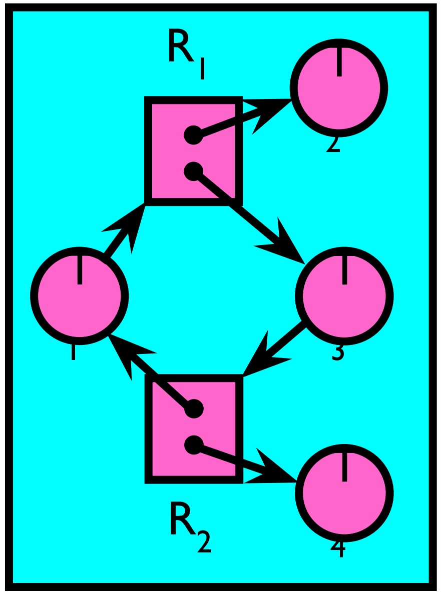
# Deadlock Detection Algorithm

- 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

# How should a system deal with deadlock?

- Four different approaches:

1. <u>Deadlock prevention</u>: write your code in a way that it isn't prone to deadlock
2. <u>Deadlock recovery</u>: let deadlock happen, and then figure out how to recover from it
3. <u>Deadlock avoidance</u>: dynamically delay resource requests so deadlock doesn't happen
4. <u>Deadlock denial</u>: ignore the possibility of deadlock

- Modern operating systems:
  - Make sure the *system* isn't involved in any deadlock
  - Ignore deadlock in applications
    - » "Ostrich Algorithm"

# Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't actually have to be infinite, just large…
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call Mom in Toledo, works way through phone network, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - » Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to retry!

# (Virtually) Infinite Resources

```
Thread A                 Thread B
AllocateOrWait(1 MB)     AllocateOrWait(1 MB)
AllocateOrWait(1 MB)     AllocateOrWait(1 MB)
Free(1 MB)               Free(1 MB)
Free(1 MB)               Free(1 MB)
```

- With virtual memory we have "infinite" space so everything will just succeed, thus above example won't deadlock
  - Of course, it isn't actually infinite, but certainly larger than 2MB!

# Techniques for Preventing Deadlock

- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (`x.Acquire()`, `y.Acquire()`, `z.Acquire()`,…)
    - » Make tasks request disk, then memory, then…
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

# Request Resources Atomically (1)

**Rather than:**

**Thread A:**

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

**Thread B:**

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

**Consider instead:**

**Thread A:**

```
Acquire_both(x, y);
…
y.Release();
x.Release();
```

**Thread B:**

```
Acquire_both(y, x);
…
x.Release();
y.Release();
```

# Request Resources Atomically (2)

**Or consider this:**

| Thread A | Thread B |
| --- | --- |
| `z.Acquire();` | `z.Acquire();` |
| `x.Acquire();` | `y.Acquire();` |
| `y.Acquire();` | `x.Acquire();` |
| `z.Release();` | `z.Release();` |
| `…` | `…` |
| `y.Release();` | `x.Release();` |
| `x.Release();` | `y.Release();` |

# Acquire Resources in Consistent Order

**Rather than:**

**Thread A:**

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

**Thread B:**

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

**Consider instead:**

**Thread A:**

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

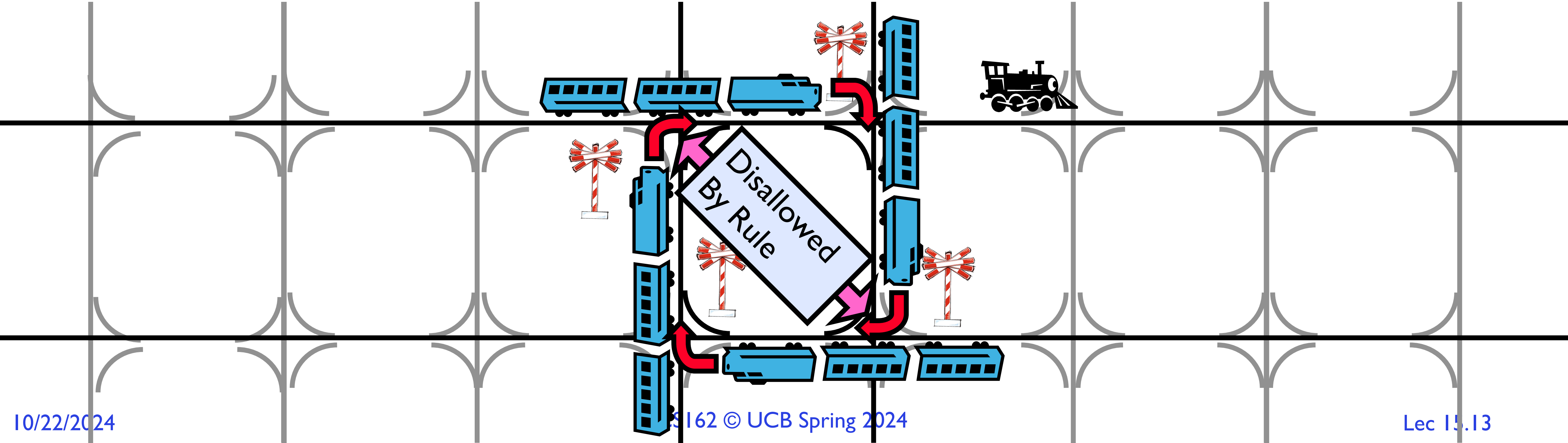
**Thread B:**

```
x.Acquire();
y.Acquire();
…
x.Release();
y.Release();
```

**Does it matter in which order the locks are released?**

# Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right, but is blocked by other trains
- Similar problem to multiprocessor networks
  - Wormhole-Routed Network: Messages trail through network like a "worm"
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



# Techniques for Recovering from Deadlock

- Terminate thread, force it to give up resources
  – In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  – Hold dining lawyer in contempt and take away in handcuffs
  – But not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  – Take away resources from thread temporarily
  – Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
  – For bridge example, make one car roll backwards (may require others behind him)
  – Common technique in databases (transactions)
  – Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

# Another view of virtual memory: Pre-empting Resources

| **Thread A:** | **Thread B:** |
| --- | --- |
| `AllocateOrWait(1 MB)` | `AllocateOrWait(1 MB)` |
| `AllocateOrWait(1 MB)` | `AllocateOrWait(1 MB)` |
| `Free(1 MB)` | `Free(1 MB)` |
| `Free(1 MB)` | `Free(1 MB)` |

- Before: With virtual memory we have "infinite" space so everything will just succeed, thus above example won't deadlock
  – Of course, it isn't actually infinite, but certainly larger than 2MB!

- Alternative view: we are "pre-empting" memory when paging out to disk, and giving it back when paging back in
  – This works because thread can't use memory when paged out

# Techniques for Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in deadlock
  – If not, it grants the resource right away
  – If so, it waits for other threads to release resources

**THIS DOES NOT WORK!!!!**

- Example:

**Thread A:**

```
x.Acquire();
y.Acquire();   Blocks…
…
y.Release();
x.Release();
```

**Thread B:**

```
y.Acquire();
x.Acquire();   Wait?
…              But it's already too late…
x.Release();
y.Release();
```

# Deadlock Avoidance: Three States

- Safe state
  – System can delay resource acquisition to prevent deadlock
- Unsafe state
  – No deadlock yet…
  – But threads can request resources in a pattern that ***unavoidably*** leads to deadlock

**Deadlock avoidance: prevent system from reaching an *unsafe* state**

- Deadlocked state
  – There exists a deadlock in the system
  – Also considered "unsafe"

# Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in ~~deadlock~~ an unsafe state
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

- Example:

**<u>Thread A</u>:**

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

**<u>Thread B</u>:**

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

**Wait until Thread A releases mutex X**

# Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

- 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:

      ($[Max_{node}]$-$[Alloc_{node}]$ <= [Avail]) for ($[Request_{node}]$ <= [Avail])

      Grant request if result is deadlock free (conservative!)

# Banker's Algorithm for Avoiding Deadlock

```
[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)
```

» 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:

$([Max_{node}]-[Alloc_{node}] <= [Avail])$ for $([Request_{node}] <= [Avail])$

Grant request if result is deadlock free (conservative!)

# Banker's Algorithm for Avoiding Deadlock

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

» 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:

  ($[Max_{node}]-[Alloc_{node}]$ <= [Avail]) for ($[Request_{node}]$ <= [Avail])

  Grant request if result is deadlock free (conservative!)

# Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

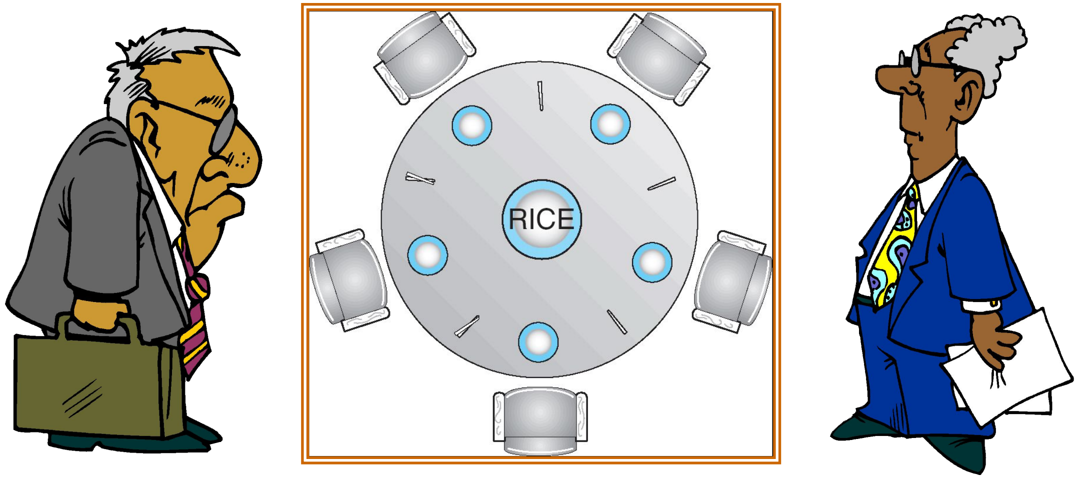
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

      ([$\text{Max}_{\text{node}}$]-[$\text{Alloc}_{\text{node}}$] <= [Avail]) for ([$\text{Request}_{\text{node}}$] <= [Avail])

      Grant request if result is deadlock free (conservative!)
  - Keeps system in a "SAFE" state: 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..

# Banker's Algorithm Example

- Banker's algorithm with dining lawyers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards

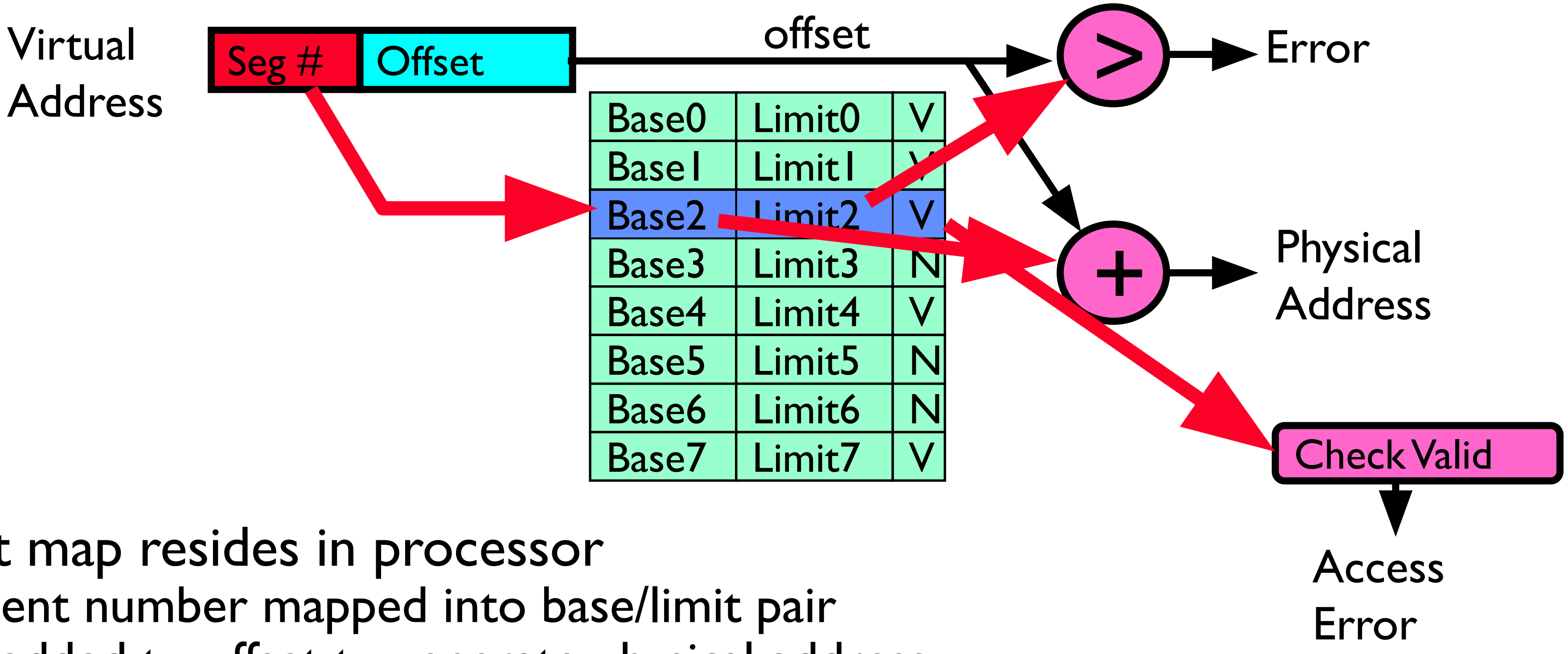



  - What if k-handed lawyers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

# Administrivia

- Midterm 2: Tuesday 11/05 from 7-9PM
  - Two weeks from today
  - Also includes the Midterm 1 material
  - Closed book: with two double-sided handwritten sheets of notes
- Project 2 design document due this Friday!
- Reminder: Ion's Office Hour
  - Thursday 11:00AM—12:00PM

# Recall: Implementation of Multi-Segment Model




- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: mov [es:bx],ax.
- What is "V/N" (valid / not valid)?
  - Can mark segments as invalid; requires check as well

# Recall: Segmentation Summary Pros

Minimal hardware requirements & efficient translation

Segmentation can better support sparse address spaces

Avoids internal fragmentation.

Minimises memory waste between logical segments of the address space

# Recall: Limitations of Segmentation

1) No expandable memory

Static memory allocation

2) No memory Sharing

Cannot share memory between processes

3) Non-Relative Memory Addresses

Location of code & data determined at runtime

4) External Fragmentation

Cannot relocate/move programs: leads to fragmentation

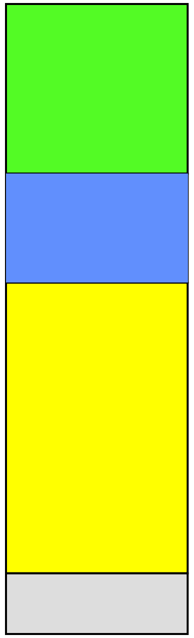
5) Internal Fragmentation

Address Space must be contiguous

# Recall: Segmentation Summary Cons

External fragmentation still a problem
Must fit variable-sized chunks into physical memory.

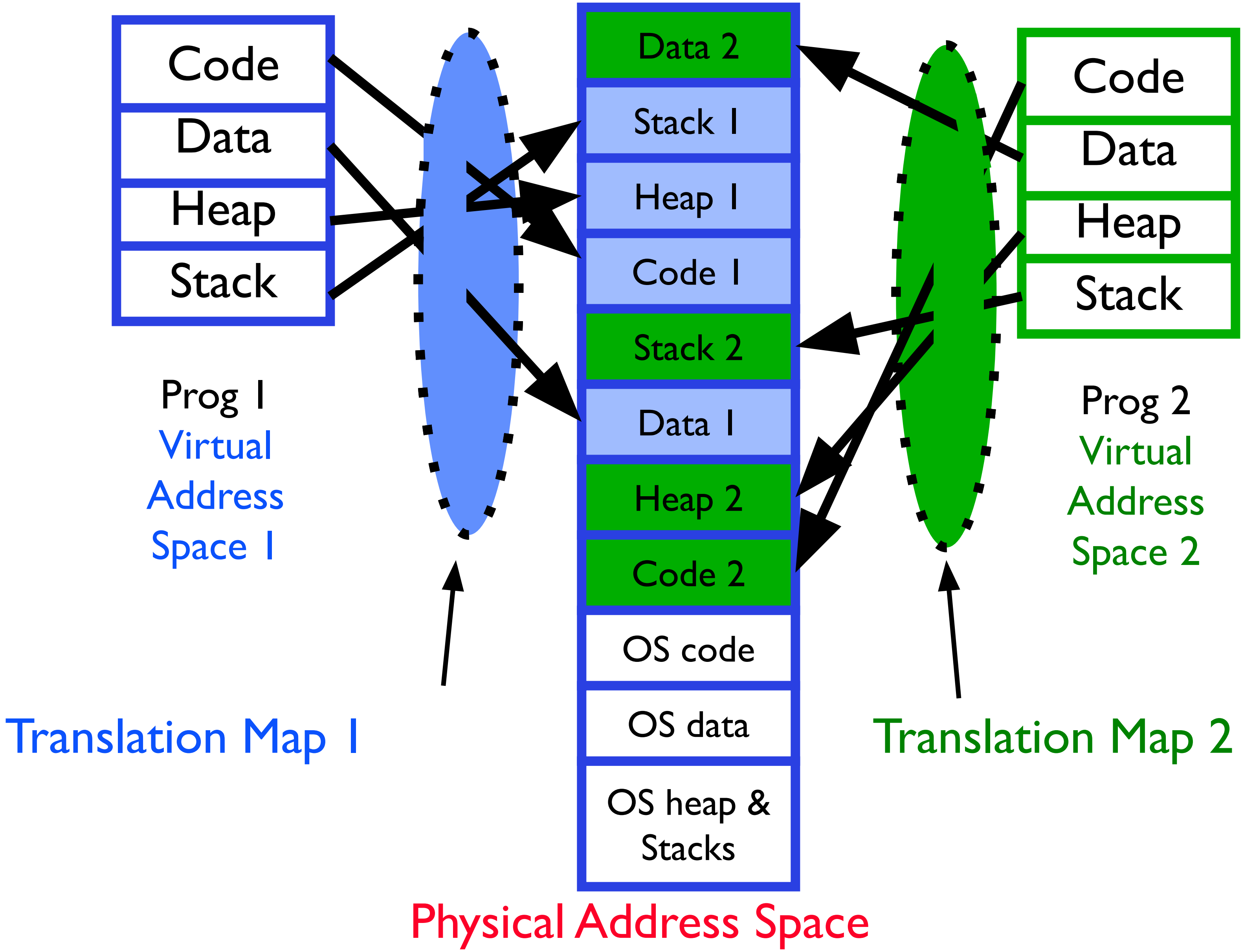
May move processes multiple times to fit everything



# Recall: Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks

# Recall: General Address Translation

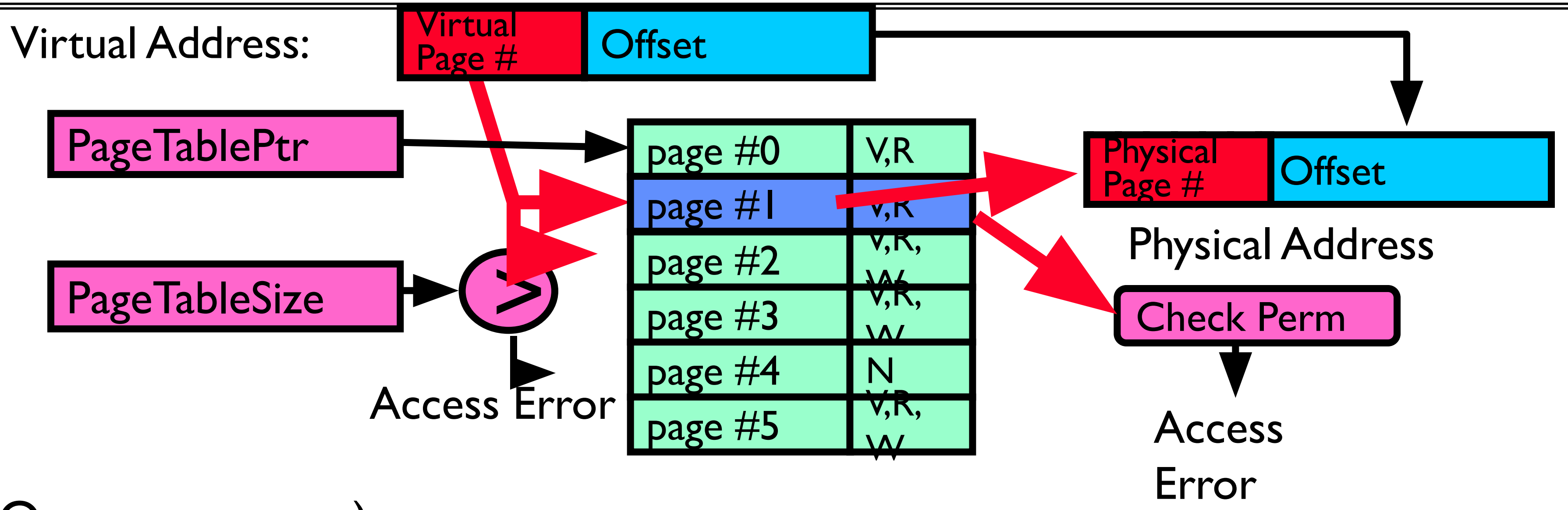


# Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      `00110001110001101 … 110010`
    - » Each bit represents page of physical memory
      `1` ⇒ allocated, `0` ⇒ free

- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

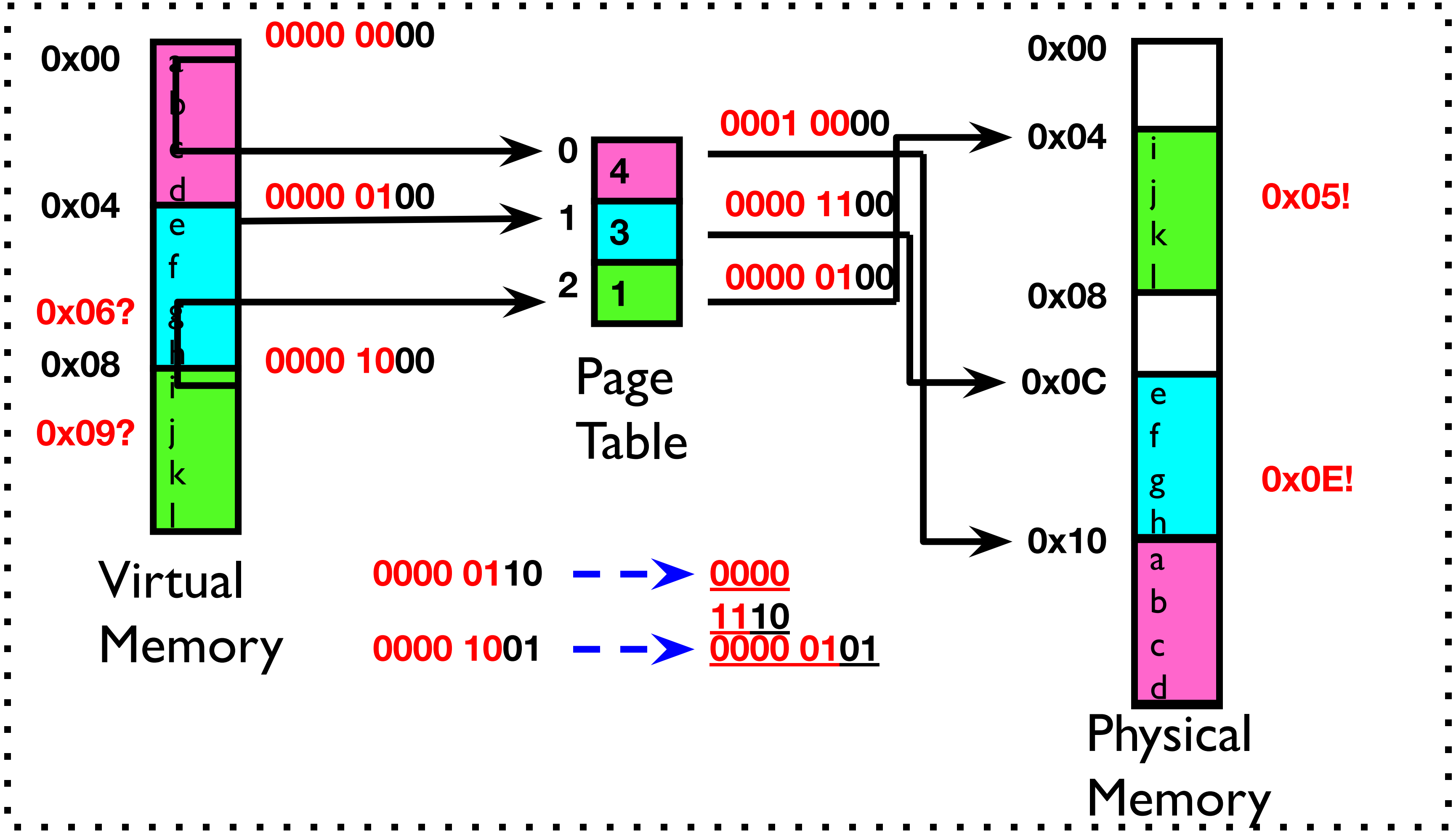
# How to Implement Simple Paging?



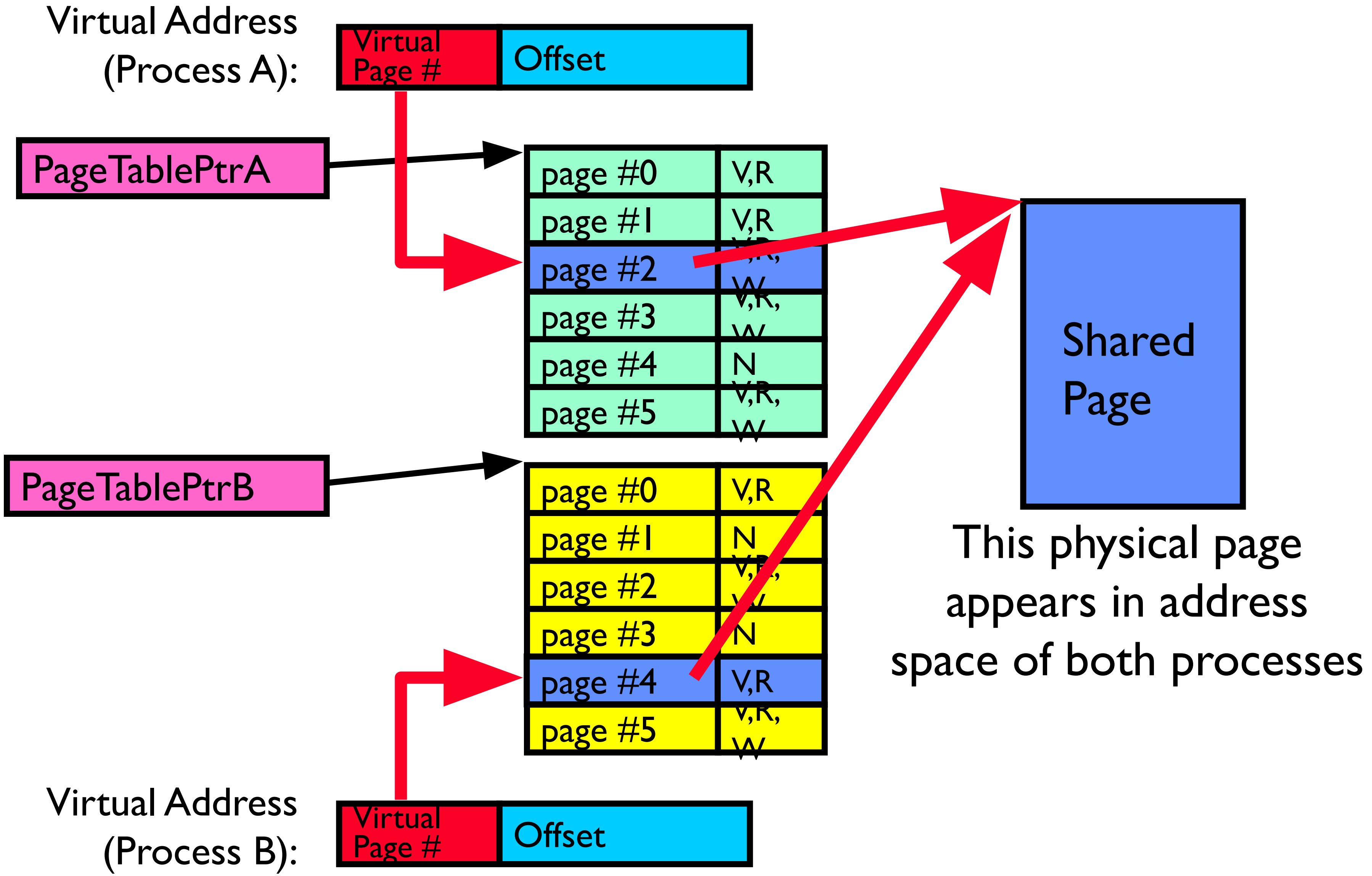

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

# Simple Page Table Example

Example (4 byte pages)



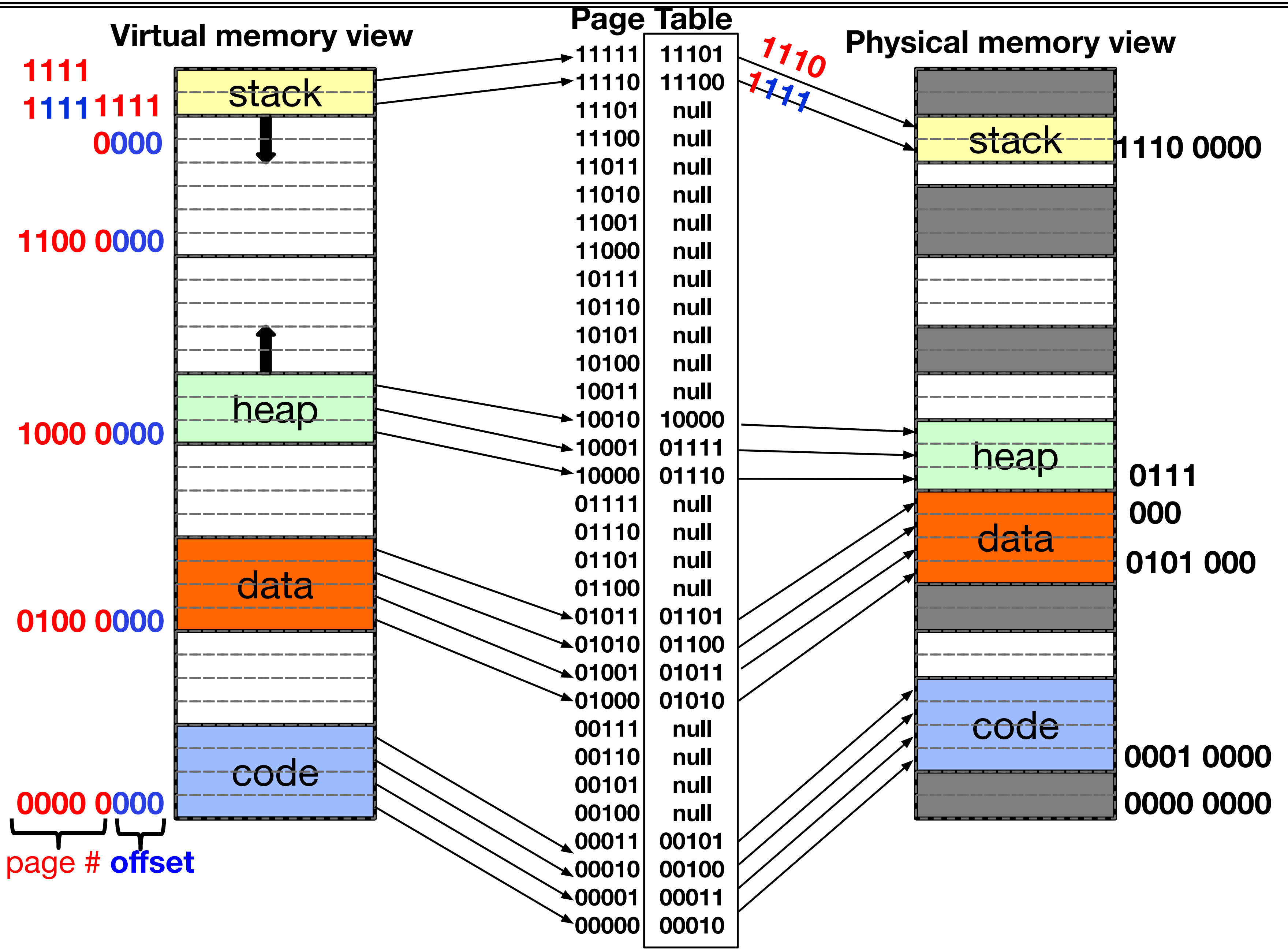
# What about Sharing?



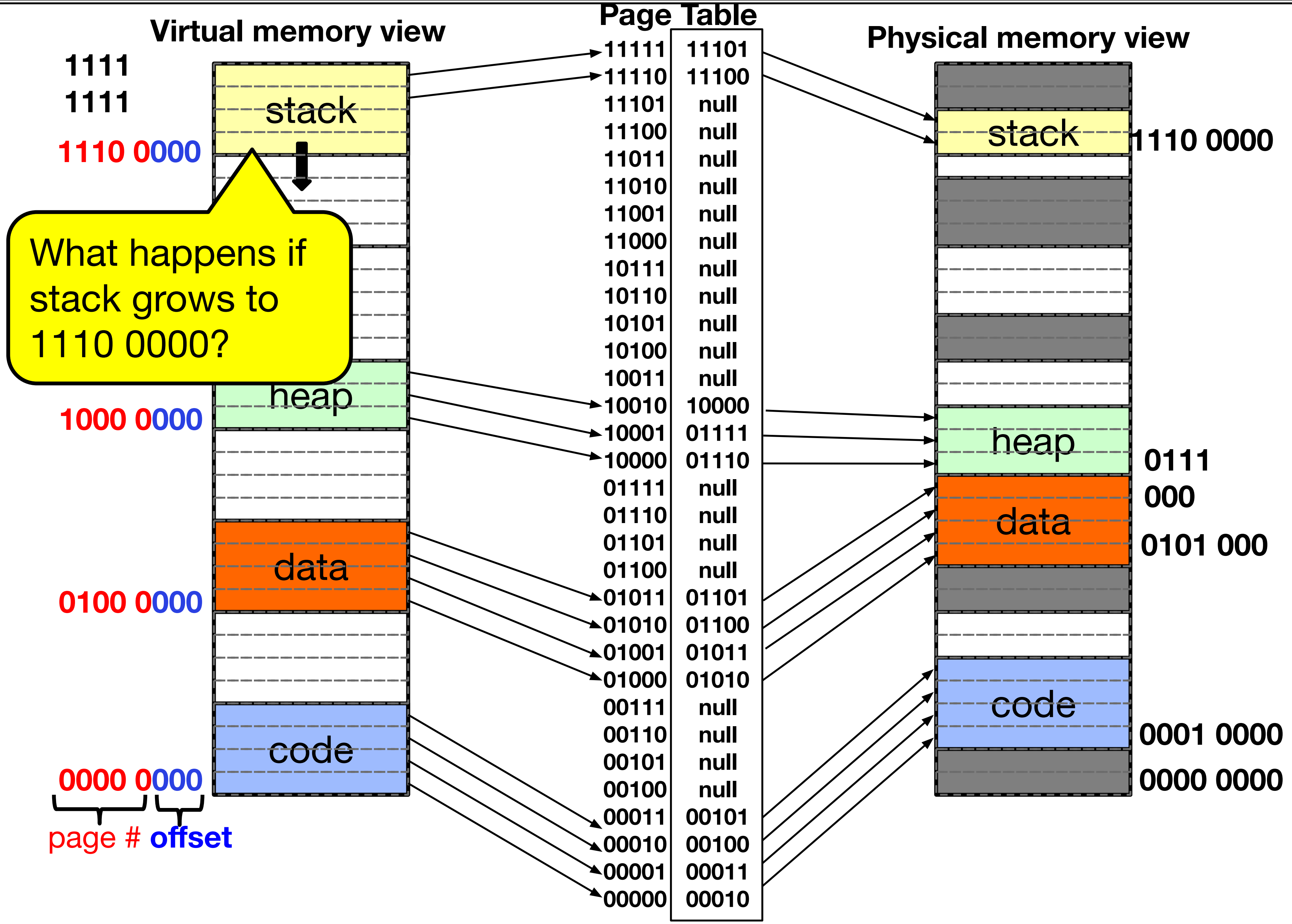
# Where is page sharing used ?

- The "kernel region" of every process has the same page table entries
  – The process cannot access it at user level
  – But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?
- Different processes running same binary!
  – Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  – Can actually share objects directly between processes
    » Must map page into same place in address space!
  – This is a limited form of the sharing that threads have within a single process

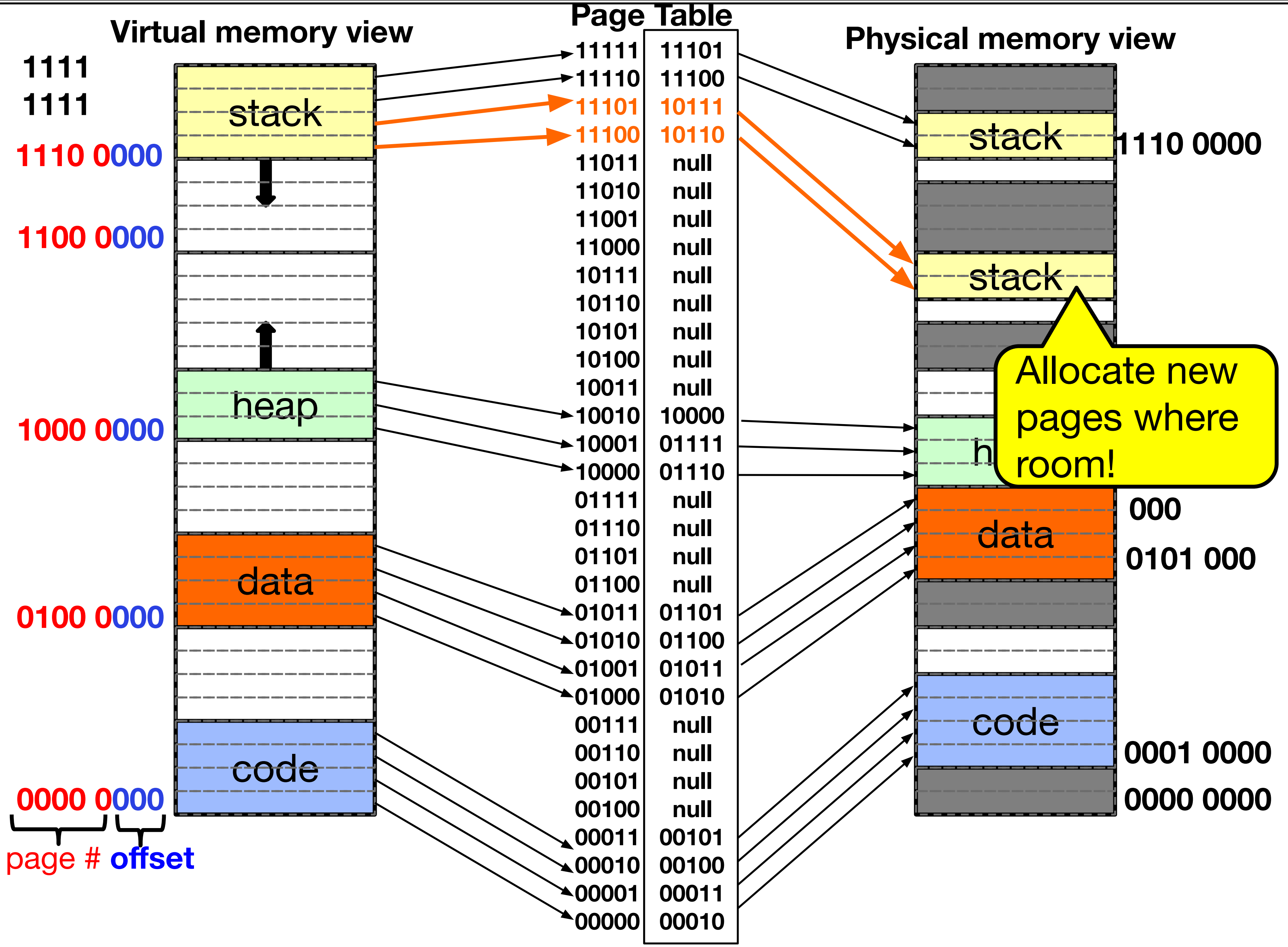
# Summary: Paging



# Summary: Paging



# Summary: Paging



# How big do things get?

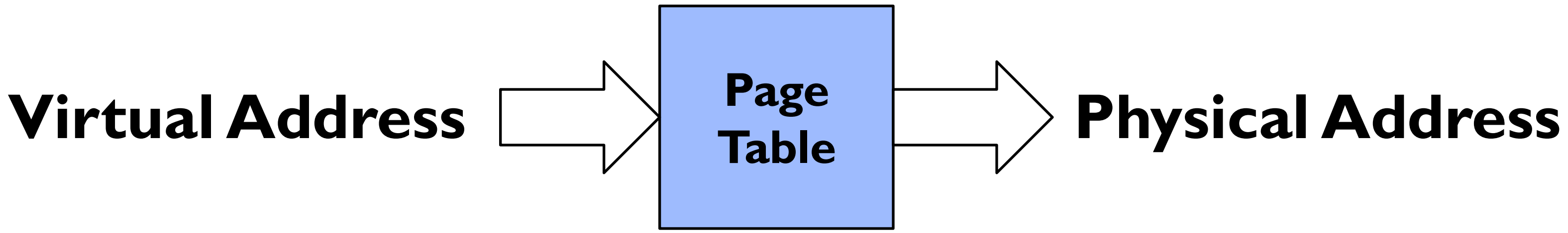
- 32-bit address space => $2^{32}$ bytes (4 GB)
  - Note: "b" = bit, and "B" = byte
  - And *for memory*:
    - » "K"(kilo) = $2^{10}$ = 1024 ≈ $10^3$ (But not quite!): Sometimes called "Ki" (Kibi)
    - » "M"(mega) = $2^{20}$ = $(1024)^2$ = 1,048,576 ≈ $10^6$ (But not quite!): Sometimes called "Mi" (Mibi)
    - » "G"(giga) = $2^{30}$ = $(1024)^3$ = 1,073,741,824 ≈ $10^9$ (But not quite!): Sometimes called "Gi" (Gibi)
- Typical page size: 4 KB
  - how many bits of the address is that ? (remember $2^{10}$ = 1024)
  - Ans – 4KB = $4 \times 2^{10} = 2^{12} \Rightarrow$ 12 bits of the address
- So how big is the simple page table for *each* process?
  - $2^{32}/2^{12} = 2^{20}$ (that's about a million entries) x 4 bytes each => 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory
- How big is a simple page table on a 64-bit processor (x86_64)?
  - $2^{64}/2^{12} = 2^{52}$(that's $4.5 \times 10^{15}$ or 4.5 exa-entries)×8 bytes each = $36 \times 10^{15}$ bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
  - This is really a lot of space – for only the page table!!!
- The address space is *sparse*, i.e. has holes that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing

# Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) *and* dual-mode!
  - Can't let process alter its own page table!
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to share
  - Con: What if address space is sparse?
    - » E.g., on UNIX, code starts at 0, stack starts at $(2^{31}-1)$
    - » With 1K pages, need 2 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation

# How to Structure a Page Table

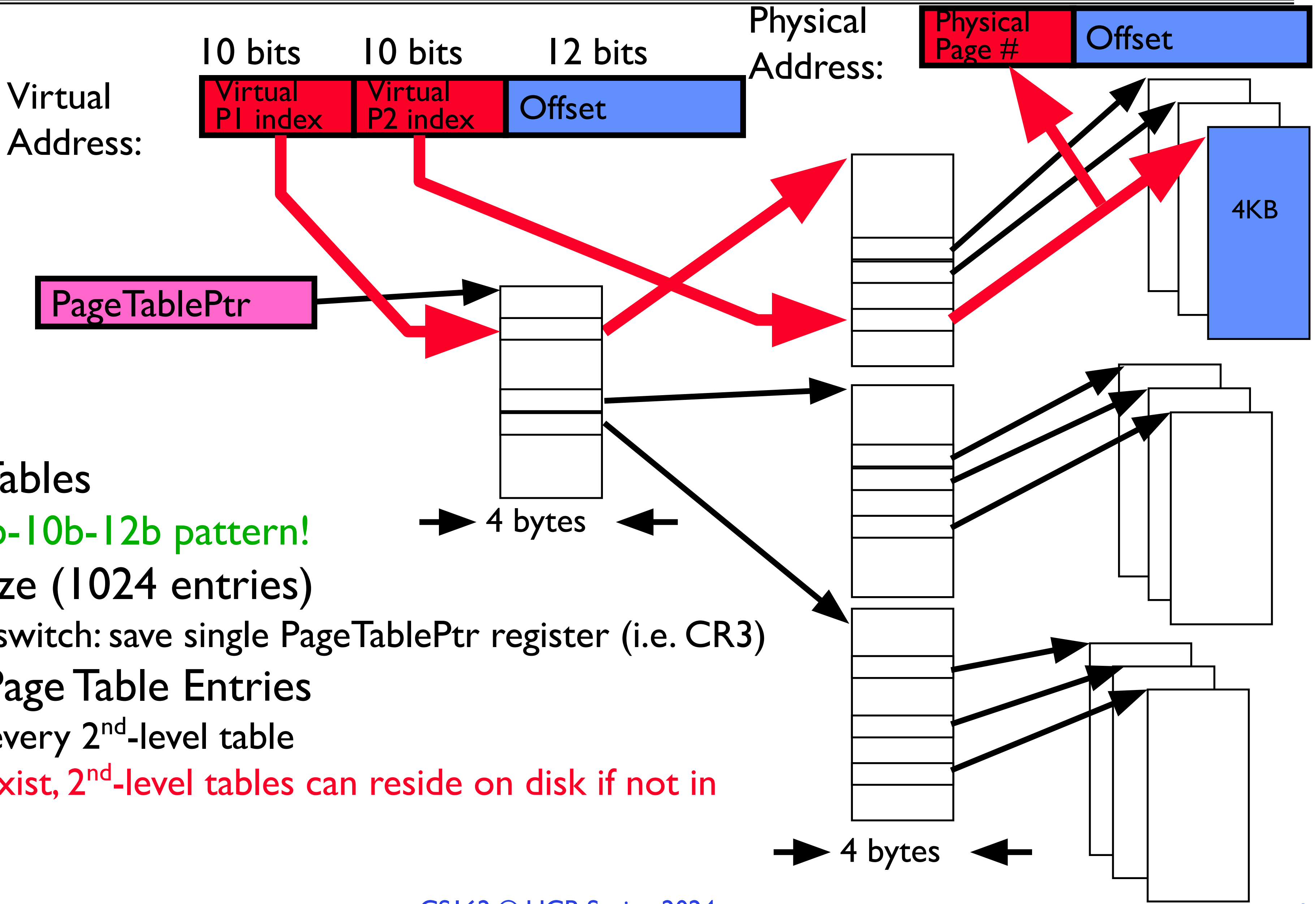
- Page Table is a *map* (function) from VPN to PPN




- Simple page table corresponds to a *very large* lookup table
  - VPN is index into table, each entry contains PPN

- What other map structures can you think of?
  - Trees?
  - Hash Tables?

# Fix for sparse address space: The two-level page table




- Tree of Page Tables
  - "Magic" 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

# What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

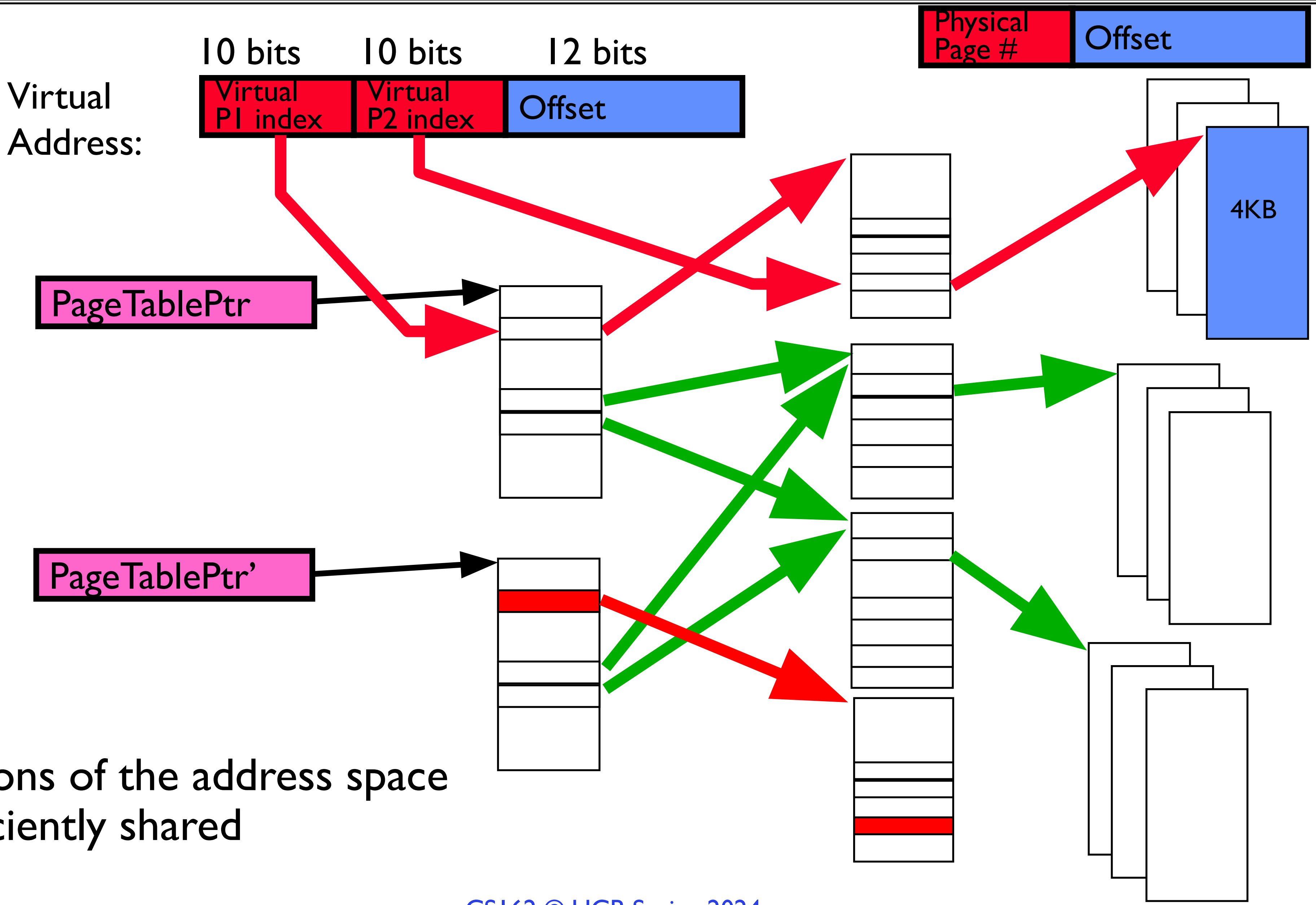
| Page Frame Number (Physical Page Number) | Free (OS) | 0 | PS | D | A | PCD | PWT | U | W | P |
|---|---|---|---|---|---|---|---|---|---|---|
| 31-12 | 11-9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
PS: Page Size: PS=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

# Examples of how to use a PTE

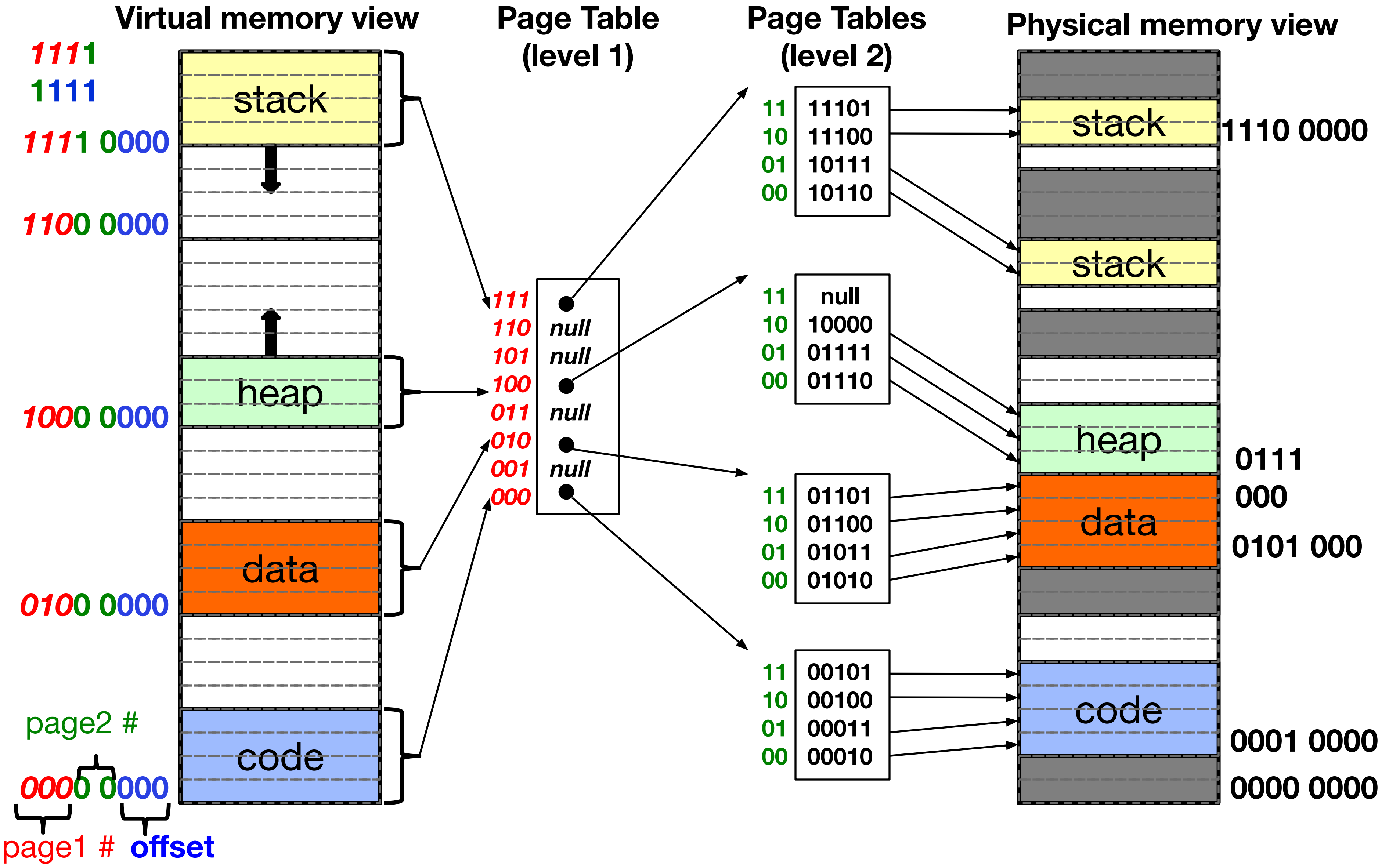
- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

# Sharing with multilevel page tables

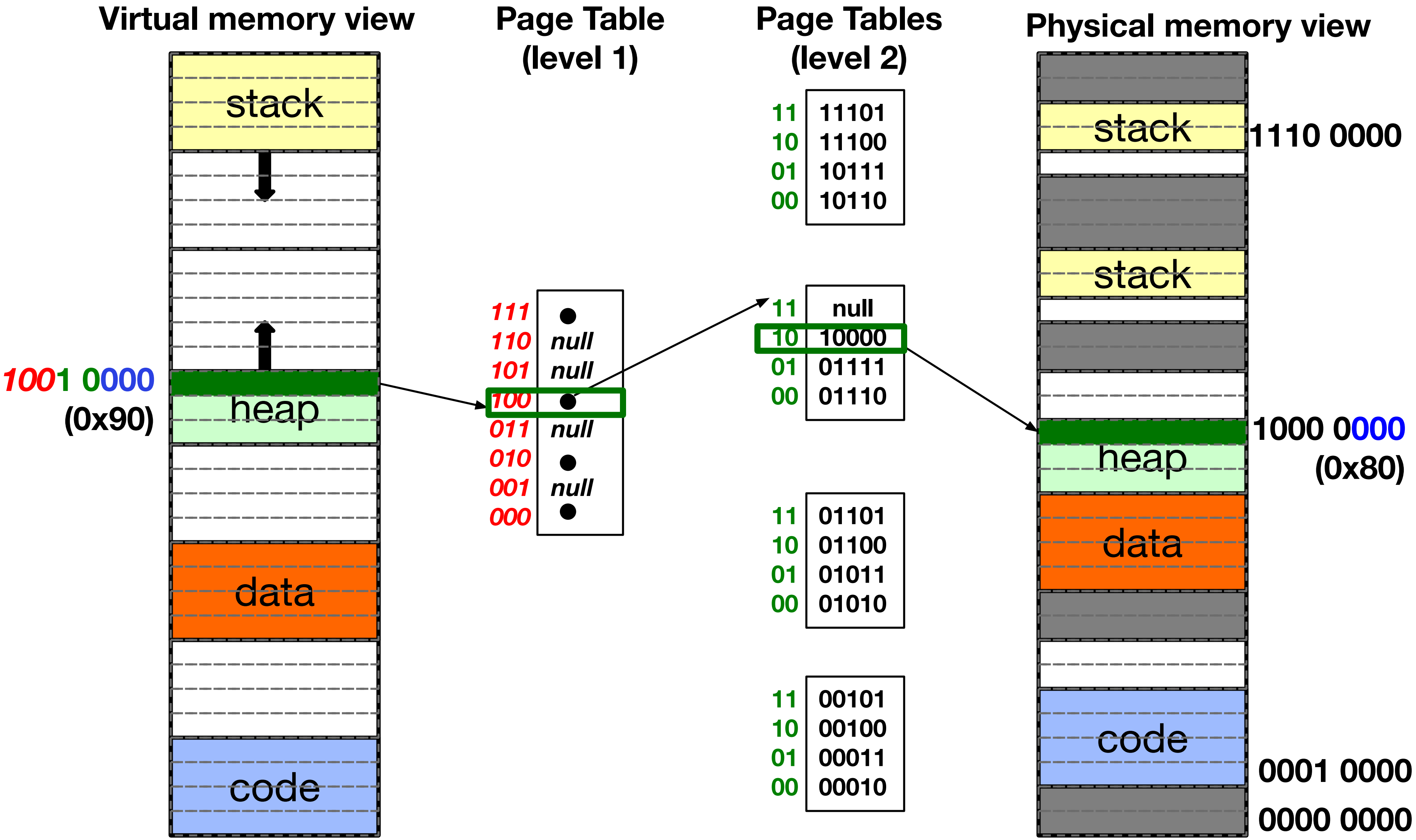



- Entire regions of the address space can be efficiently shared

# Summary: Two-Level Paging

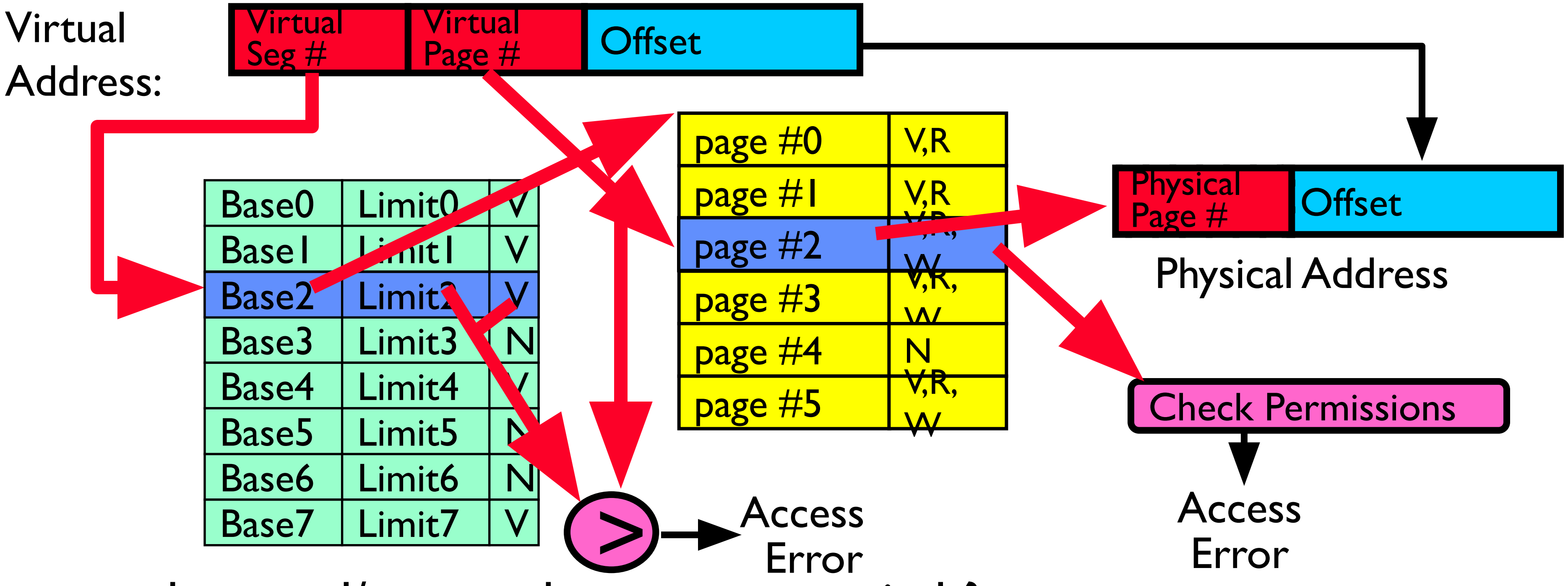


# Summary: Two-Level Paging



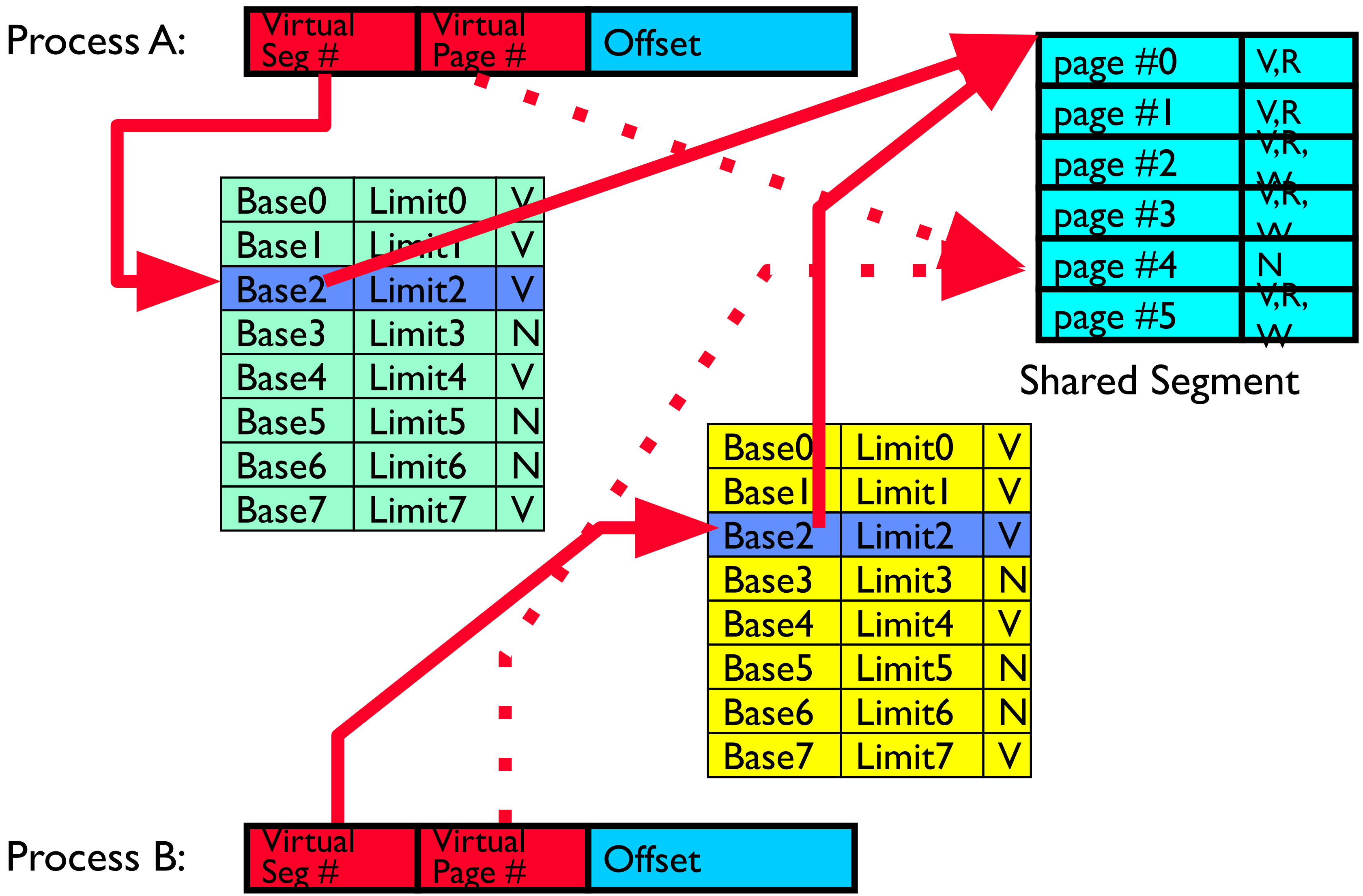
# Multi-level Translation: Segments + Pages

- 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?
  – Contents of top-level segment registers (for this example)
  – Pointer to top-level table (page table)

# What about Sharing (Complete Segment)?



# Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, the 10b-10b-12b configuration keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

# Recall: Dual-Mode Operation

- Can a process modify its own translation tables? NO!
  - If it could, could get access to all of physical memory (no protection!)
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bit(s) in control register only accessible in Kernel mode
  - Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode (more in moment)
- Note that x86 model actually has more modes:
  - Traditionally, four "rings" representing priority; most OSes use only two:
    - » Ring 0 $\Rightarrow$ Kernel mode, Ring 3 $\Rightarrow$ User mode
    - » Called "Current Privilege Level" or CPL
  - Newer processors have additional mode for hypervisor ("Ring -1")
- Certain operations restricted to Kernel mode:
  - Modifying page table base (CR3 in x86), and segment descriptor tables
    - » Have to transition into Kernel mode before you can change them!
  - Also, all page-table pages must be mapped only in kernel mode

# Conclusion

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Techniques for addressing Deadlock
  - <u>Deadlock prevention</u>:
    - » write your code in a way that it isn't prone to deadlock
  - <u>Deadlock recovery</u>:
    - » let deadlock happen, and then figure out how to recover from it
  - <u>Deadlock avoidance</u>:
    - » dynamically delay resource requests so deadlock doesn't happen
    - » Banker's Algorithm provides on algorithmic way to do this
  - <u>Deadlock denial</u>:
    - » ignore the possibility of deadlock