Memory Management: paging

Dr. Yingwu Zhu

What do we know about page table?

- One page table per process
- Stores corresponding page frame # for a page #
- And stores page permissions:
  - Read-only
  - No-execute
  - Dirty (modified)
  - Referenced
  - Others (e.g., secure or privileged mode access)
- Page table is selected by setting a page table base register with the address of the table
- Memory protection
  - Isolation of address spaces
  - Access control defined in PTE

TLB to improve look-up performance

- Cache frequently-accessed pages
  - Translation lookaside buffer (TLB)
  - Associative memory: key (page #) and value (frame #)
- TLB is on-chip & fast … but small (64 – 1,024 entries)
- TLB miss: result not in the TLB
  - Need to do page table lookup in memory
- Hit ratio = % of lookups that come from the TLB
- Address Space Identifier (ASID): share TLB among address spaces

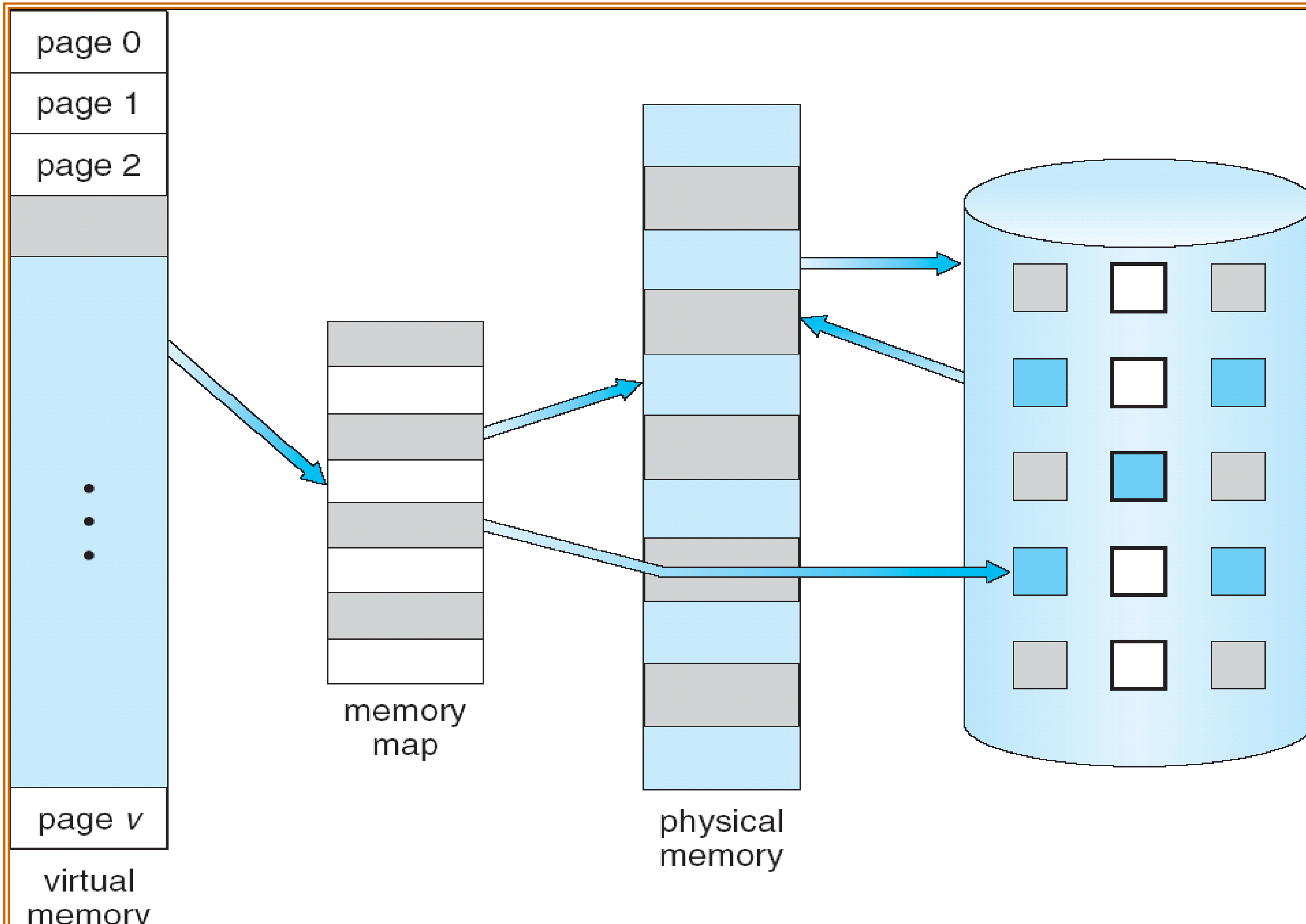
Our first cut on memory management

- Assumption:
  - Physical memory > process size
- MMU + page-table
  - Give the illusion of contiguous allocation
- Memory Protection
  - Each process' address space is separate from others
  - MMU allows pages to be protected:
    - Writing, execution, kernel vs. user access

Second cut: back to reality

- **Physical memory < process size!**
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows physical address spaces to be shared by processes.
  - Allows for more efficient process creation. Why?
- Virtual memory can be implemented via:
  - Demand paging (our focus)
  - Demand segmentation

Virtual Memory that is Larger than Physical Memory



Access memory

- Process makes *virtual address references for all memory access*
- MMU converts to physical address via a per-process page table
  - – Page number → Page frame number
  - – Basic info stored in a PTE (page table entry):
    - Valid? Is the page mapped?
    - Page frame number
    - Permissions (read-only, read-write, execute-only, …)
    - Modified?
- – Page fault if not a valid reference
- Most CPUs support:
  - – Virtual addressing mode and Physical addressing mode
    - CPU starts in physical mode … someone has to set up page tables
  - – Divide address space into user & kernel spaces

Kernel's view

- Each process sees a flat linear address space
  - Accessing regions of memory mapped to the kernel causes a page fault
- Kernel's view:
  - Address space is split into two parts
    - User part: changes with context switches
    - Kernel part: remains constant
  - Split is configurable:
    - 32-bit x86: PAGE_OFFSET: 3GB for process + 1 GB kernel

Page allocator

- With VM, processes can use non-contiguous pages
- Sometimes you need contiguous allocation
  - E.g., DMA logic ignores paging (bypass CPU & MMU)
  - If we rely on DMA, we need contiguous pages

Page allocator

- Linux kernel support for contiguous buffers
  - free_area: keep track of lists of free pages
  - 1st element: free single pages
  - 2nd element: free blocks of 2 contiguous pages
  - 3rd element: free blocks of 4 contiguous pages
  - …
  - 10th element: free blocks of 512 contiguous pages

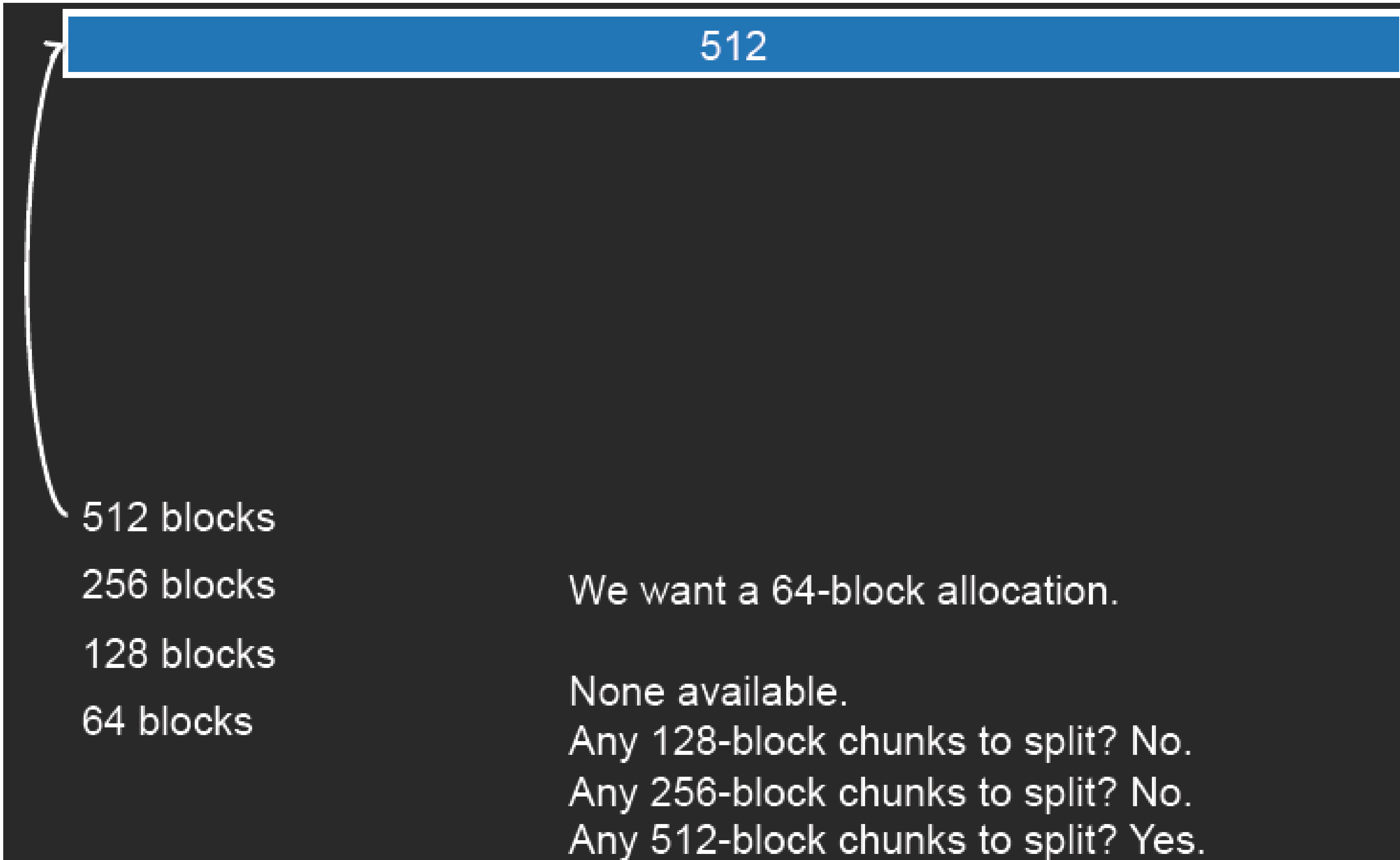
Buddy system

- Try to get the best usable allocation unit
- If not available, get the next biggest one & split
- Coalesce upon free
- Example
  - We want 8 contiguous pages
  - Do we have a block of 8? Suppose no.
  - Do we have a block of 16? Suppose no.
  - Do we have a block of 32? Suppose yes.
    - Split the 32 block into two blocks of 16. Back up.
  - Do we have a block of 16? Yes!
    - Split one of the 16 blocks into two blocks of eight. Back up.
  - Do we have a block of 8? Yes!

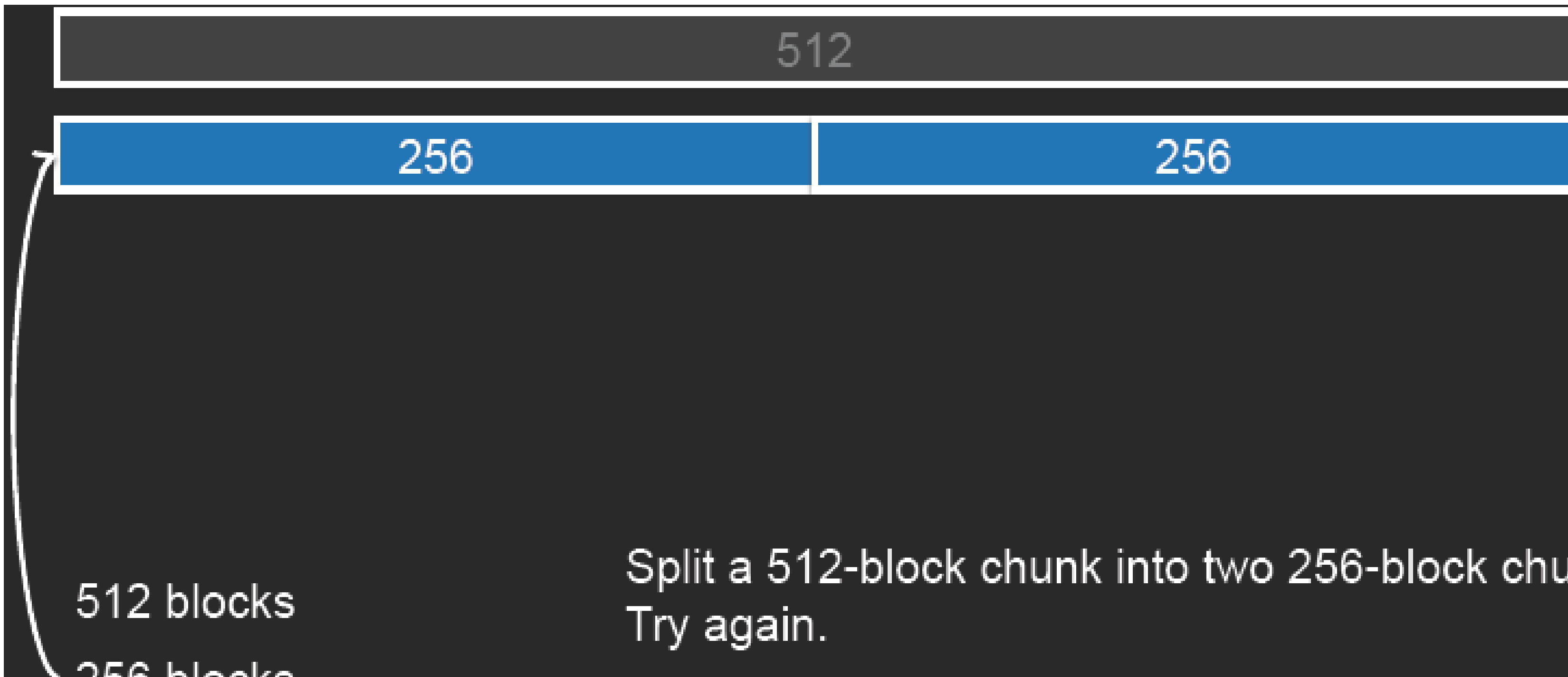
Buddy System: Coalescence

- When a block is freed, see if we can merge buddies
- Two blocks are buddies if:
  - They are the same size, *b*
  - They are contiguous
  - The address of the first page of the lower # block is a multiple of $2b \times page\_size$
- If two blocks are buddies, they are merged
- Repeat the process

Buddy system example



Buddy system example



512 blocks

256 blocks

128 blocks

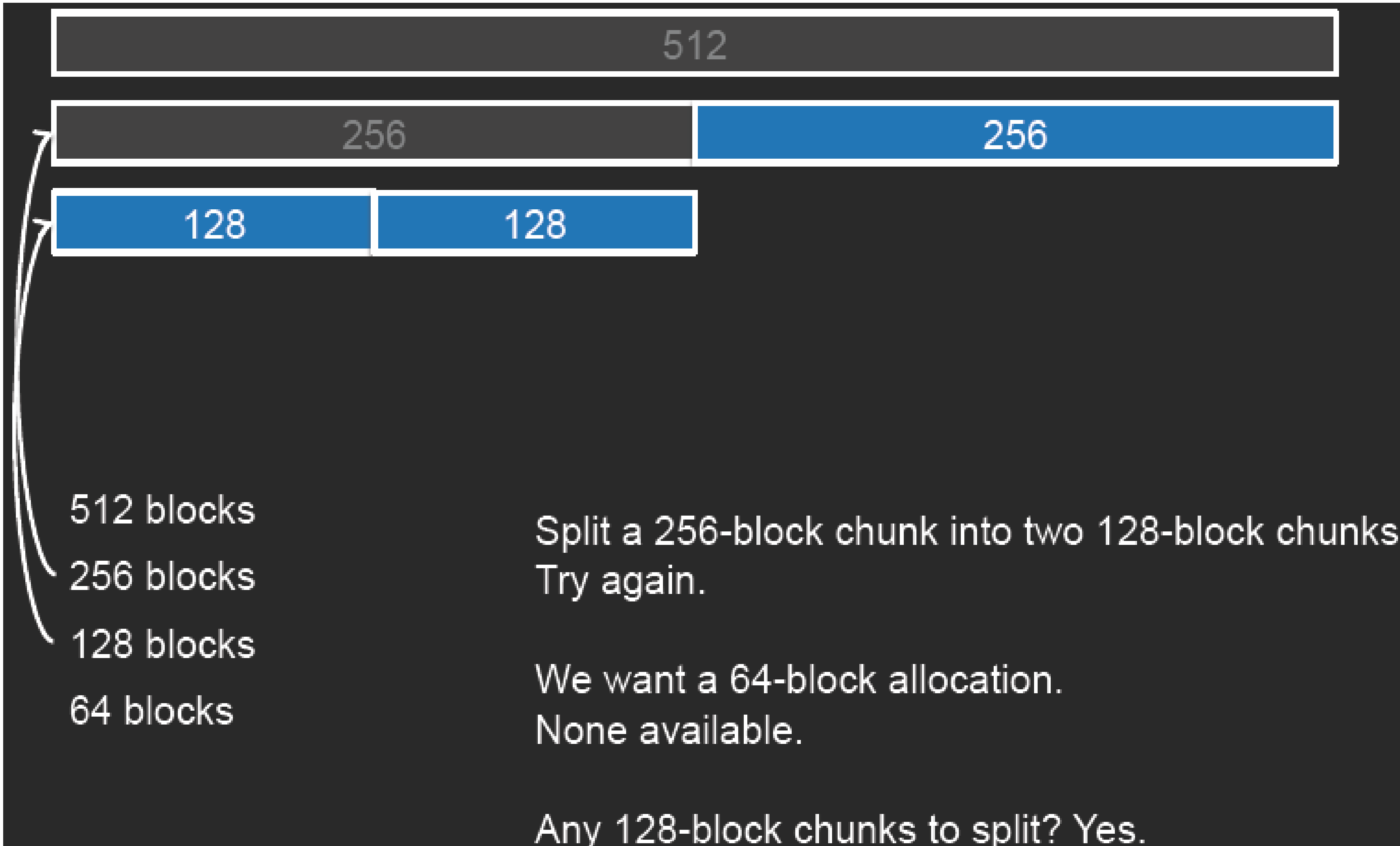
64 blocks

Split a 512-block chunk into two 256-block chunks.
Try again.

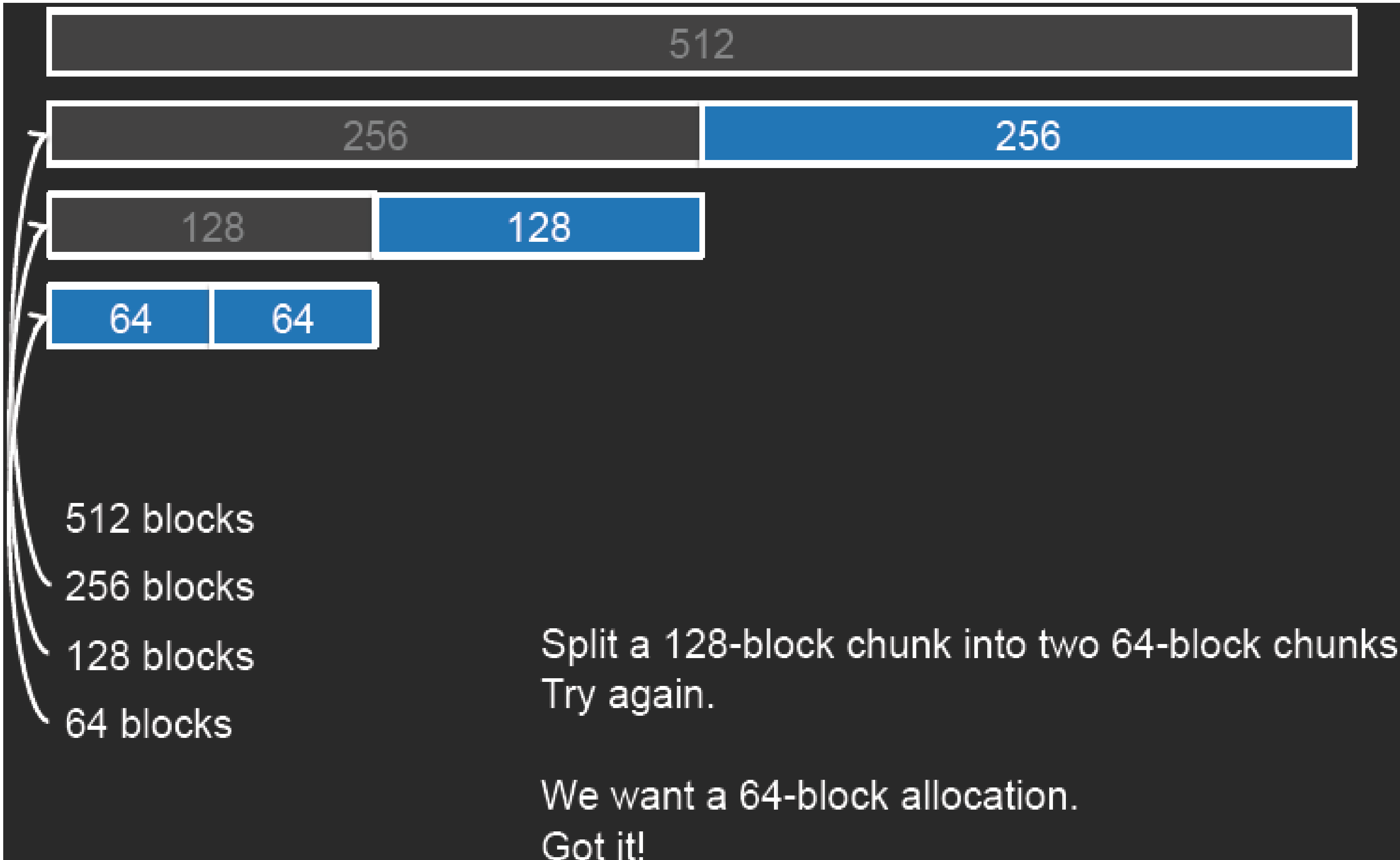
We want a 64-block allocation.
None available.

Any 128-block chunks to split? No.
Any 256-block chunks to split? Yes.

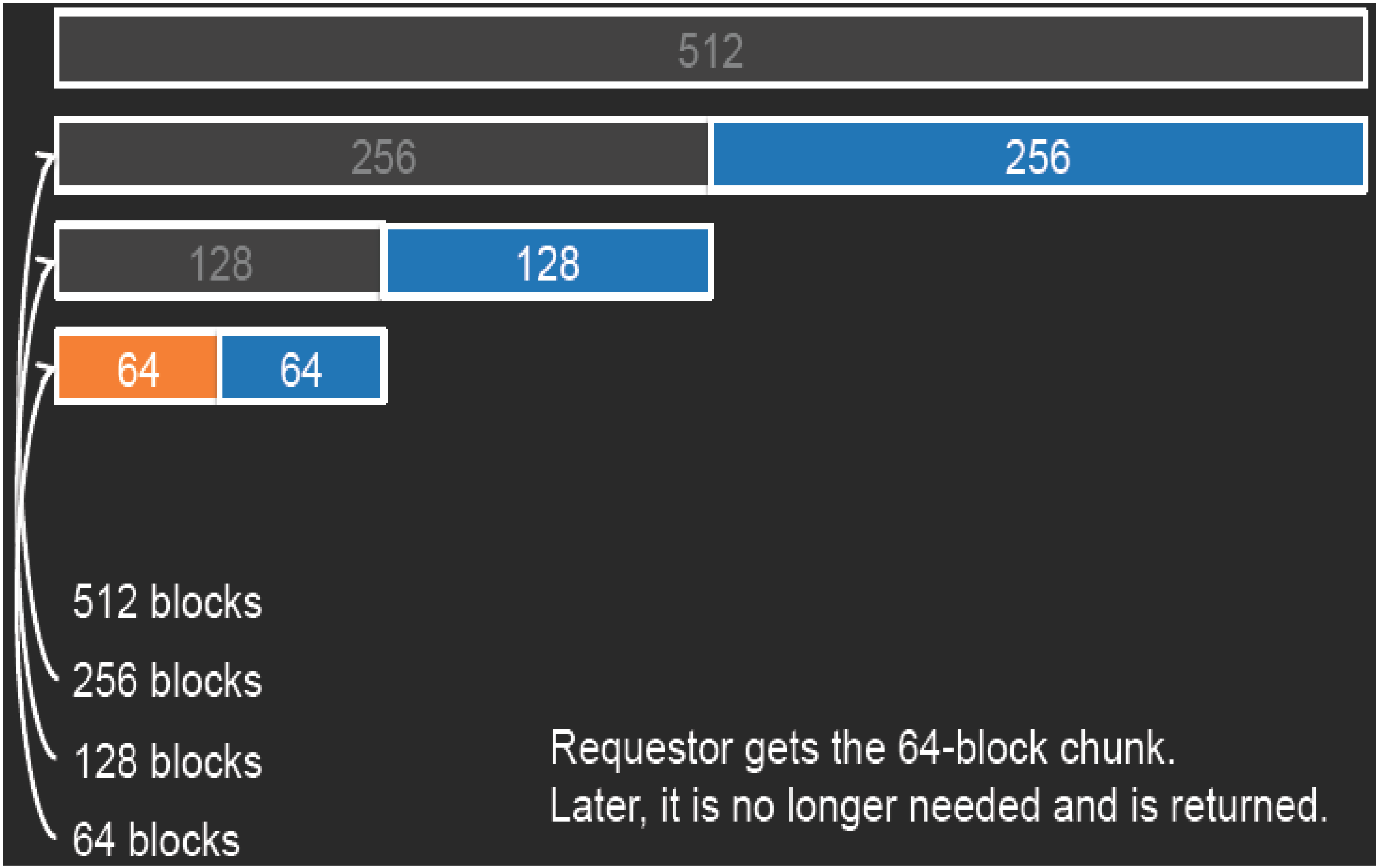
Buddy system example



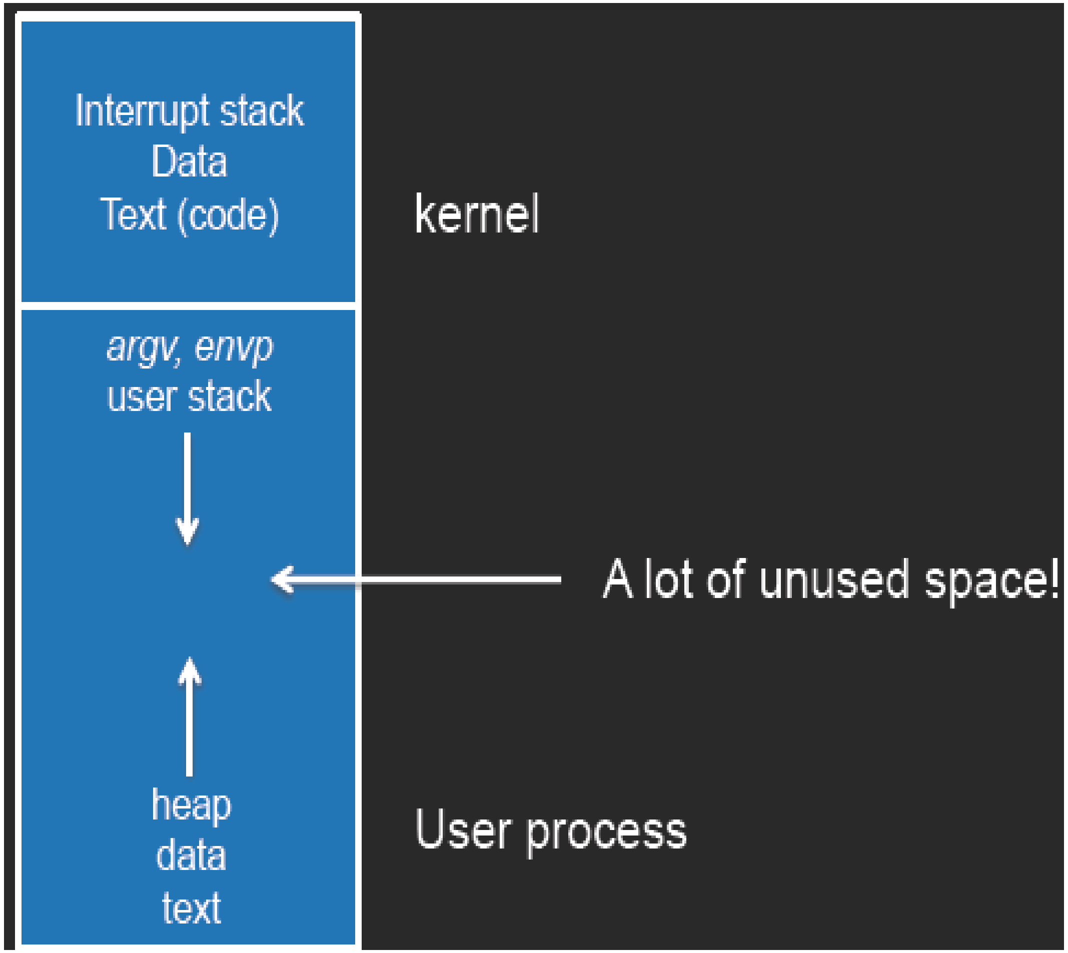
Buddy system example



Buddy system example



Sample memory map per process

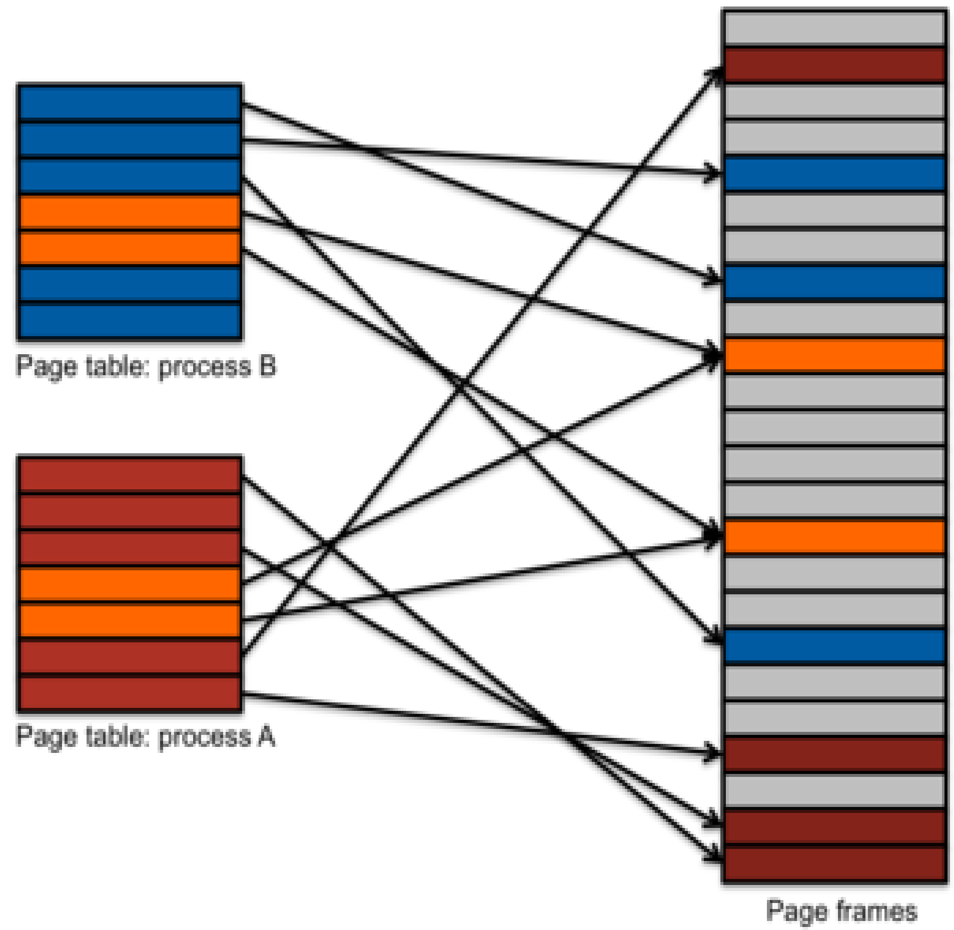


Multilevel (Hierarchical) page tables

- Most processes use only a small part of their address space
- Keeping an entire page table is wasteful
- E.g., 32-bit system with 4KB pages: 20-bit page table
  - 1,048,576 entries in a page table

Virtual memory makes memory sharing easy

- Sharing is by page granularity.
- Shared library or shared memory
- Keep reference counts!



Copy on write

- Share until a page gets modified
- Example: fork()
  - Set all pages to read-only
  - Trap on write
  - If legitimate write
    - Allocate a new page and copy contents

Demand Paging

Executing a Program

- Allocate memory + stack and load the entire program into memory (including linked libraries)
- Then execute it

Executing a Program

- Allocate memory + stack and load the entire program into memory (including linked libraries)
- Then execute it

We do not have to do this!

Demand Paging

- Load pages into memory only as needed
  - On first access
  - Pages that are never used never get loaded
- Use valid/invalid bit in page table entry
  - Valid: the page is in memory (“valid” mapping)
  - Invalid: out of bounds access or page is not in memory
    - Have to check the process’ memory map in the PCB to find out
- Invalid memory access generates a *page fault*

Demand Paging: At Process Start

- Open executable file
- Set up memory map (stack & text/data/bss)
  - But don't load anything!
- Load first page & allocate initial stack page
- Run it!

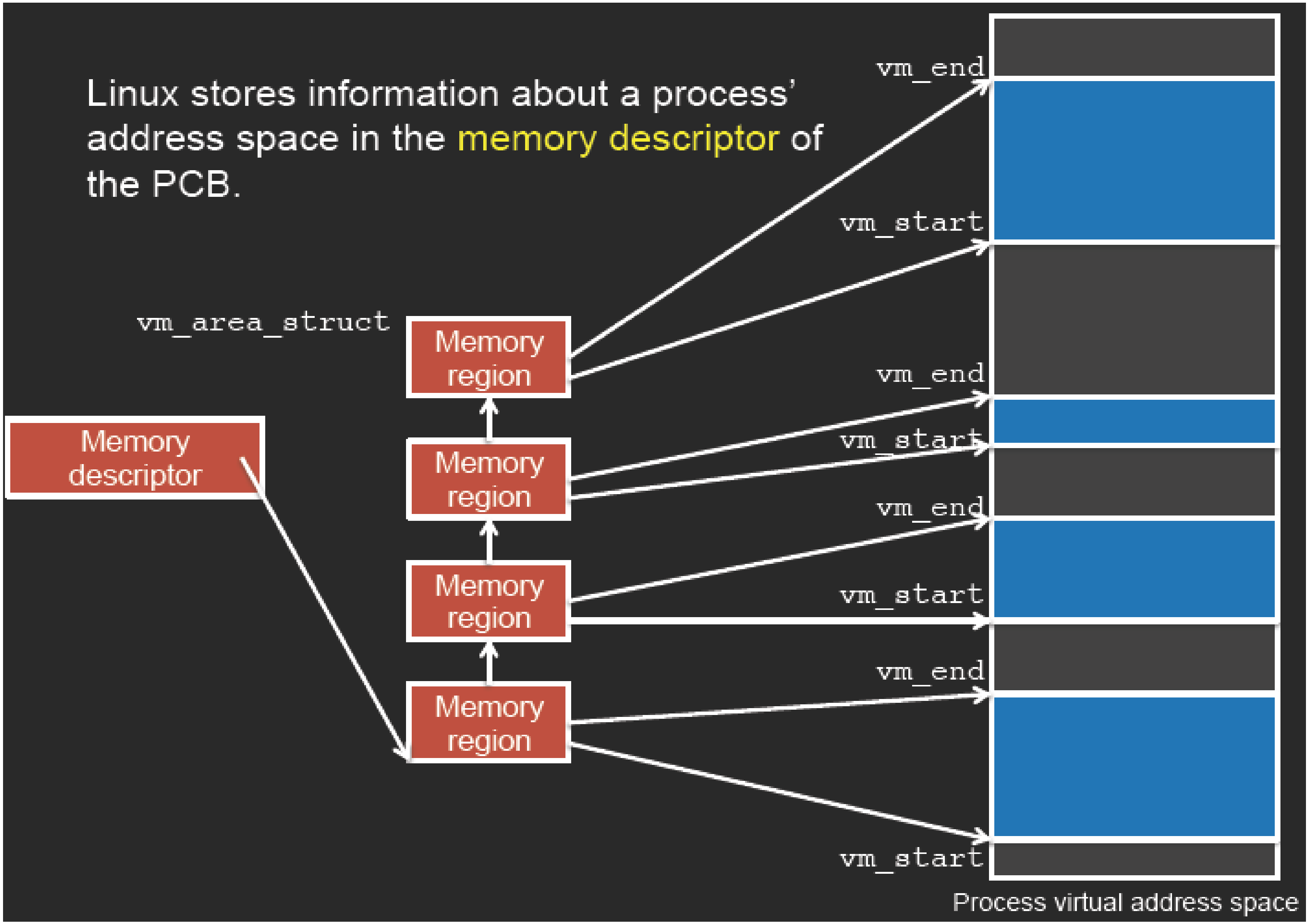
Memory Mapping

- Executable files & libraries must be brought into a process' virtual address space
  - File is *mapped into the process' memory*
  - As pages are referenced, page frames are allocated & pages are loaded into them
- If we ever run out of memory, we may need to save some modified pages into a swap file and load those in later on demand.
- vm_area_struct !
  - Defines regions of virtual memory
  - Used in setting page table entries
  - Start of VM region, end of region, access rights
- Several of these are created for each mapped image
  - Executable code, initialized data, uninitialized data

Demand Paging: Page Fault Handling

- Soon the process will access an address without a valid page
  - OS gets a page fault from the MMU
- What happens?
  - Kernel searches a tree structure of memory allocations for the process to see if the faulting address is valid
    - If not valid, send a SEGV (segmentation violation) signal to the process
  - Is the type of access valid for the page?
    - Send a signal if not
  - We have a valid page but it's not in memory

Keeping track of a processes' memory region



Demand Paging: Getting a Page

- The page we need is either in the a mapped file (executable or library) or in a swap file
  - If PTE is not valid but page # is present
    - The page we want has been saved to a swap file
    - Page # in the PTE tells us the location in the file
  - If the PTE is not valid and no page #
    - Load the page from the program file from the disk
- Read page into physical memory
  - Find a free page frame (evict one if necessary)
  - Read the page: This takes time: context switch & block
  - Update page table for the process
  - Restart the process at the instruction that faulted

Page Replacement

- A process can run without having all of its memory allocated
  - It's allocated on demand
- • If the {address space used by all processes + OS}
  ≤ physical memory
  then we're ok
- Otherwise:
  - Make room: discard or store a page onto the disk
  - If the page came from a file & was not modified
    - Discard … we can always get it
  - If the page is dirty, it must be saved in a swap file
  - Swap file: a file (or disk partition) that holds excess pages

Cost

- Handle page fault exception: ~ 400 usec (microseconds)
- Disk seek & read: ~ 10 msec
- Memory access: ~ 100 ns
- Page fault degrades performance by around 100,000!!
- Avoid page faults!
  - If we want < 10% degradation of performance, we can have just one page fault per 1,000,000 memory accesses

Page Replacement

- We need a good replacement policy for good performance

FIFO Replacement

- First In, First Out
- Good
  - May get rid of initialization code or other code that's no longer used
- Bad
  - May get rid of a page holding frequently used global variables

Least Recently Used (LRU)

- Timestamp a page when it is accessed
- When we need to remove a page, search for the one with the oldest timestamp
- Nice algorithm but…
  - Timestamping is a pain – we can’t do it with the MMU!

Not Frequently Used Replacement

- Approximate LRU
- Each PTE has a reference bit
- Keep a counter for each page frame
- At each clock interrupt:
  - Add the reference bit of each frame to its counter
  - Clear reference bit
- To evict a page, choose the frame with the lowest counter
- Problem
  - No sense of time: a page that was used a lot a long time ago may still have a high count
  - Updating counters is expensive

Clock (Second Chance)

- Arrange physical pages in a logical circle (circular queue)
  - Clock hand points to first frame
- Paging hardware keeps 1 *reference bit per frame*
  - Set *reference bit on memory reference*
  - If it's not set then the frame hasn't been used for a while
- On page fault:
  - Advance clock hand
  - Check *reference bit*
    - If 1, it's been used recently – clear & advance
    - If 0, evict this page

Enhanced Clock (Second Chance)

- Use the *reference and modify bits of the page*
- Choices for replacement – (reference, modify):
  - (0, 0): not referenced recently or modified
    - Good candidate for replacement
  - (0, 1): not referenced recently but modified.
    - The page will have to be saved before replacement
  - (1, 0): recently used.
    - Less ideal – will probably be used again
  - (1, 1): recently used and modified
    - Least ideal – will probably be used again AND we'll have to save it to a swap file if we replace it.
- Algorithm: like clock but replace the first page in the lowest non-empty class

Nth Chance Replacement

- Similar to Second Chance
- Maintain a counter along with a reference bit
- On page fault:
  - Advance clock hand
  - Check reference bit
    - If 1, clear and set counter to 0
    - If 0, increment counter. If counter < N, go on. Else evict
- Better approximation of LRU

Kernel Swap Daemon

- *kswapd on Linux*
- Anticipate problems
- Timer periodically triggered
- Decides whether to shrink caches if page count (# of free page frames) is low
  - Page cache, buffer cache
  - Evict pages from page frames

Demand paging summary

- Allocate page table
  - Map kernel memory
  - Initialize stack
  - Memory-map text & data from executable program (& libraries)
    - But don’t load!
- Load pages on demand (first access)
  - When we get a page fault

Summary: If we run out of free page frames

- Free some page frames
  - Discard pages that are mapped to a file

  or

  - Move some pages to a swap file
- Clock algorithm
- Anticipate need for free page frames
  - *kswapd – kernel swap dæmon*

Multitasking Considerations

Supporting multitasking

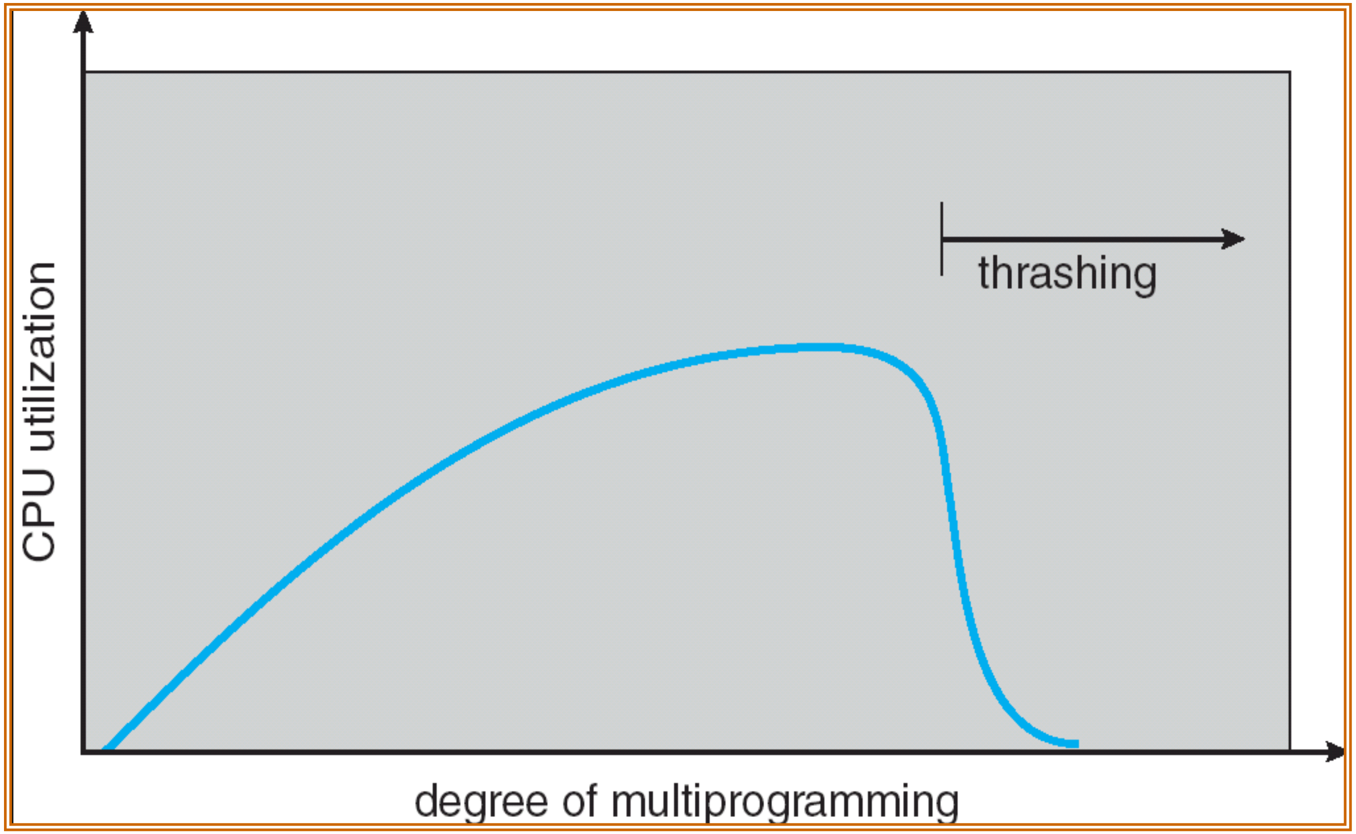
- Multiple address spaces can be loaded in memory
- A CPU register points to the current page table
- OS changes the register set when context switching
- Performance increased with Address Space ID in TLB

Working Set

- Keep active pages in memory
- A process needs its working set in memory to perform well
  - Working set = set of pages that have been referenced in the last window of time
  - Spatial locality
  - Size of working set varies during execution
- More processes in a system:
  - Good: increase throughput; chance that some process is available to run
  - Bad: thrashing: processes do not have enough page frames available to run without paging

Thrashing

- Locality
  - Process migrates from one locality (working set) to another
- Thrashing
  - Occurs when sum of all working sets > total memory



Working Set Model

- Approximates locality of a program
- Δ: *working set window*:
  - Amount of elapsed time while the process was actually executing (e.g., count of memory references)
- WSS_i : working set size of process P_i
  - WSS_i = set of pages in most recent Δ page references
- System-wide demand for frames

  $D = \sum WSS_i$
- If D > *total memory size*, then we get thrashing