CSE 506: Operating Systems

Memory Management

Review

- We've seen how paging works on x86
  - Maps virtual addresses to physical pages
  - These are the low-level hardware tools
- This lecture: build up to higher-level abstractions
- Namely, the process address space

Managing Physical Pages

- During boot, discover all physical pages
  - Based on e820 map
    - e820 information sometimes mangled
      - Real OSes sanitize / remove overlaps
    - Some OSes do extra "checks" on memory
      - Basic test to confirm memory works as expected
- Create data structures to keep track of pages
  - Usually maintained as lists of pages
    - Zeroed, Free (page colored), Cache, Wired, …
- Should kernel use physical pages?
  - Virtual is more convenient, but management is harder
  - Support both (Linux discourages V, FreeBSD discourages P)

Practical Considerations

- Must balance “free” queues
  - Zero free pages when idle
  - Launder pages (write dirty pages to disk and mark clean)
- TLB supports larger pages for better performance
  - Many names: Super Pages, Large Pages, Huge pages
  - Structures for maintaining multiple sizes are hard
    - Promoting/Demoting/Reserving/Fragmenting/…
    - Typically use 4KB, play games on alloc to get contiguous regions
- Physical page lists take space (and time to initialize)
  - Especially considering physical frames (pages) are 4KB
  - Beneficial (but hard) to do lazy initialization

Definitions (can vary)

- Process is a virtual address space
  - 1+ threads of execution work within this address space
- A process is composed of:
  - Memory-mapped files
    - Includes program binary
  - Anonymous pages: no file backing
    - When the process exits, their contents go away
- OSes don't usually allocate physical pages
  - Until application touches them
  - As needed, OS takes from free list (***_get_free_page()***)
    - What if contiguous physical pages needed? (***_get_free_pages()***)

Address Space Layout

- Determined (mostly) by the application
- Determined at compile time
  - Link directives can influence this
    - There is a default (internal) linker script in the system
      - ENTRY(_start), . = 0x400000, etc…
- OS usually reserves part of address space to itself
  - Usually “high” in space (0xfff…)
- Application can dynamically request new mappings
  - malloc(), mmap(), or stack

In practice

- See (part of) the requested layout using ldd:

```
$ ldd /usr/bin/git
 linux-vdso.so.1 =>  (0x00007fff197be000)
 libz.so.1 => /lib/libz.so.1 (0x00007f31b9d4e000)
 libpthread.so.0 => /lib/libpthread.so.0
                                (0x00007f31b9b31000)
 libc.so.6 => /lib/libc.so.6 (0x00007f31b97ac000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f31b9f86000)
```

Problem 1: How to represent in the kernel?

- How to represent the components of a process?
  - Common question: what is mapped at address x?
    - Needed on page faults, new memory mappings, etc…
- Hint: a 64-bit address space is huge
- Hint: programs (like databases) can map tons of data
  - Others map very little
- No one size fits all

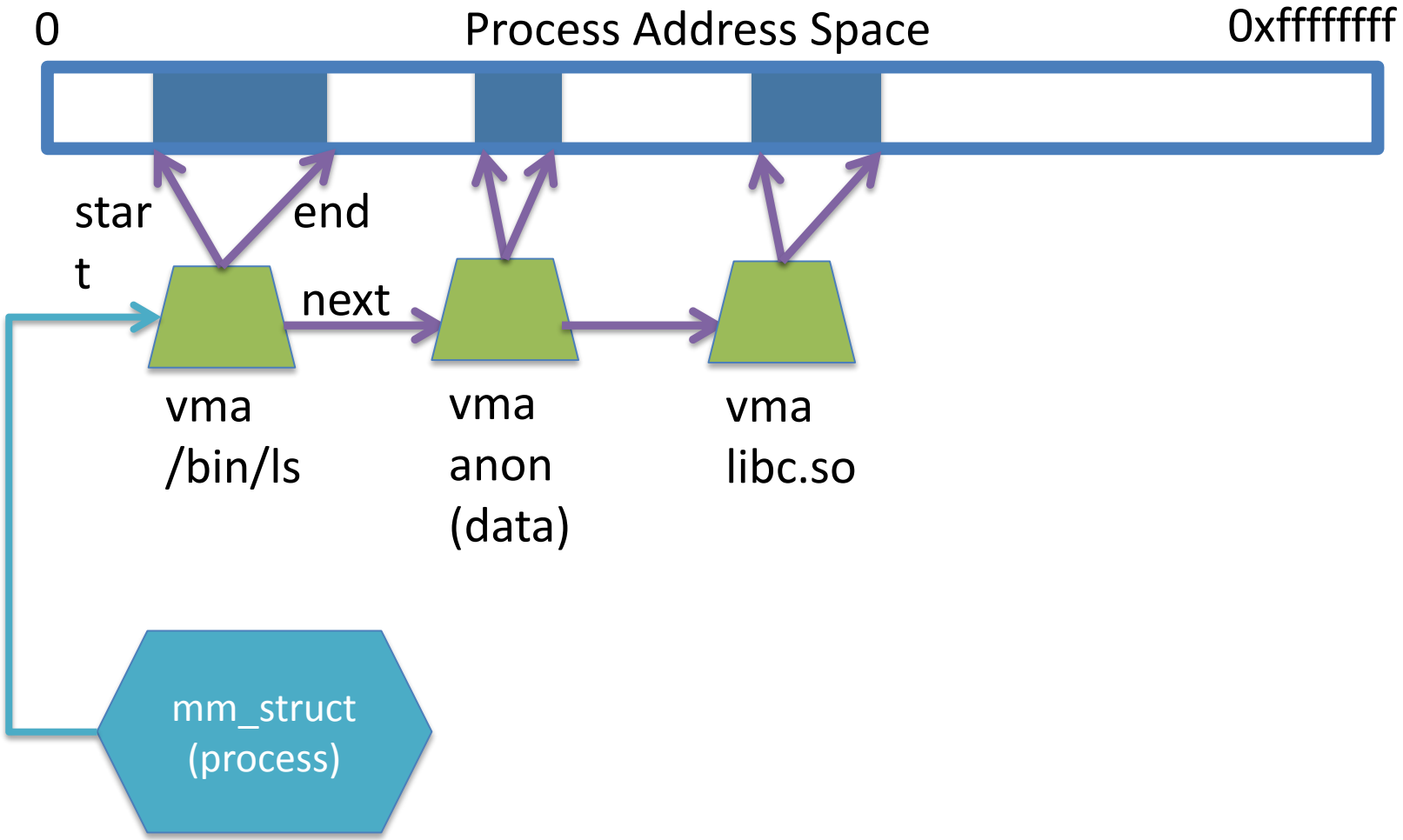
Sparse representation

- Naïve approach might make a big array of pages
  - Mark empty space as unused
  - But this wastes OS memory
- Better idea: allocate structure for mapped memory
  - Proportional to complexity of address space

Linux: vm_area_struct

- Linux represents portions of a process with a vm_area_struct (vma)
- Includes:
  - Start address (virtual)
  - End address (first address after vma) – why?
    - Memory regions are page aligned
  - Protection (read, write, execute, etc…) – implication?
    - Different page protections means new vma
  - Pointer to file (if one)
  - Other bookkeeping

Simple list representation



Simple list

- Linear traversal – O(n)
  - Shouldn't we use a data structure with the smallest O?
- Practical system building question:
  - What is the common case?
  - Is it past the asymptotic crossover point?
- Tree is O(log n), but adds bookkeeping overhead
  - 10 vmas: log 10 =~ 3; 10/2 = 5; Comparable either way
  - 100 vmas: log 100 starts making sense

Common cases

- Many programs are simple
  - Only load a few libraries
  - Small amount of data
- Some programs are large and complicated
  - Databases
- Linux uses both a list and a red-black tree

Red-black trees

- (Roughly) balanced tree
- Popular in real systems
  - Asymptotic == worst case behavior
    - Insertion, deletion, search: O(log n)
    - Traversal: O(n)

Optimizations

- Using a RB-tree gets logarithmic search time
- Other suggestions?
- If region x accessed, likely to be accessed again
  - Cache pointer in each process to the last vma looked up
  - Source code (mm/mmap.c) claims 35% hit rate

Memory mapping recap

- VM Area structure tracks regions that are mapped
  - Efficiently represent a sparse address space
  - On both a list and an RB-tree
    - Fast linear traversal
    - Efficient lookup in a large address space
  - Cache last lookup to exploit temporal locality

Linux APIs

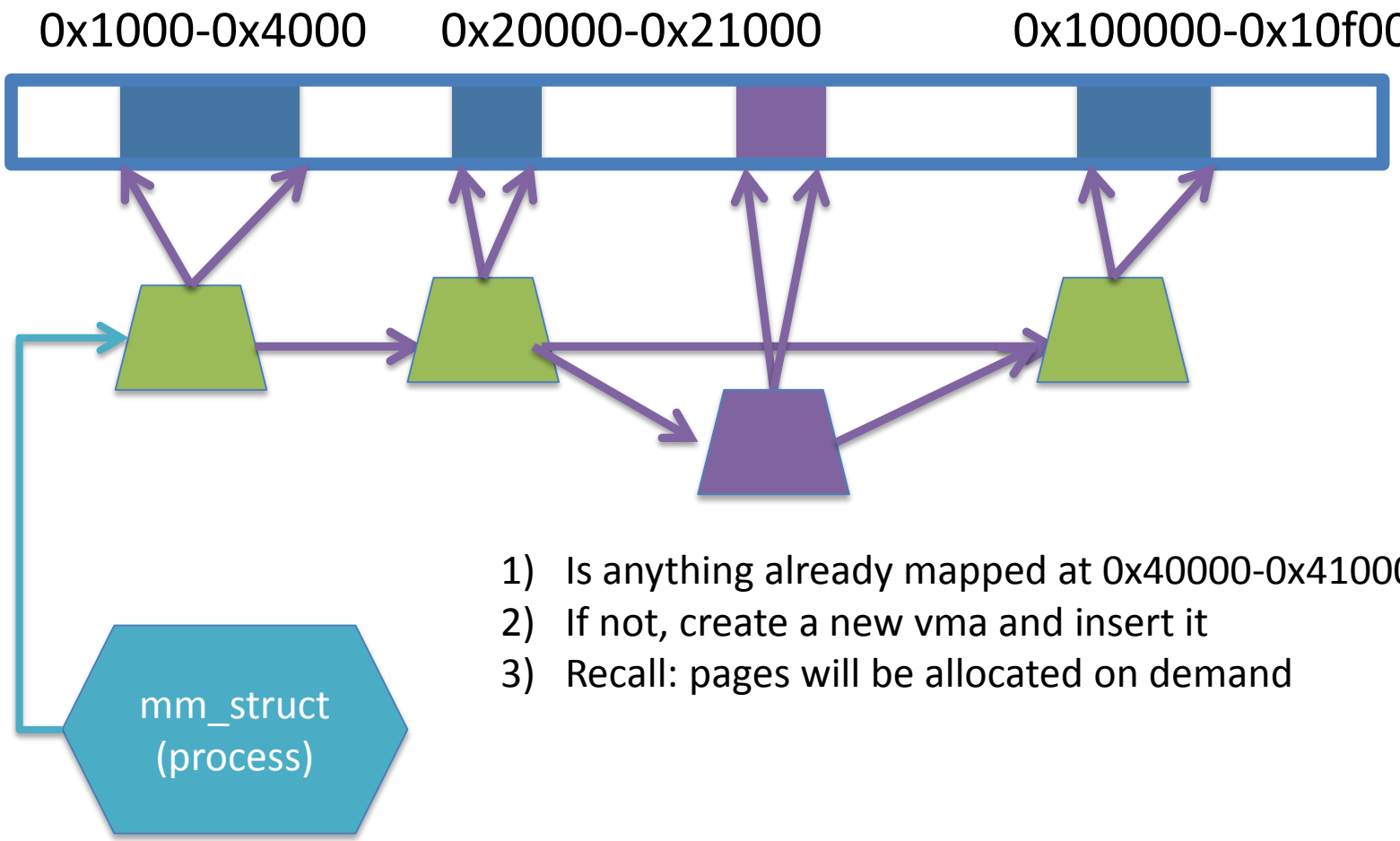
```
mmap(void *addr, size_t length, int prot, int flags,
        int fd, off_t offset);
munmap(void *addr, size_t length);
```

- How to create an anonymous mapping?
- What if you don't care where a memory region goes (as long as it doesn't clobber something else)?

Example 1:

- Map 4k anon region for rw data at 0x40000
- mmap(0x40000, 4096, PROT_READ|PROT_WRITE,
  MAP_ANONYMOUS, -1, 0);
  - Why wouldn't we want exec permission?

Insert at 0x40000

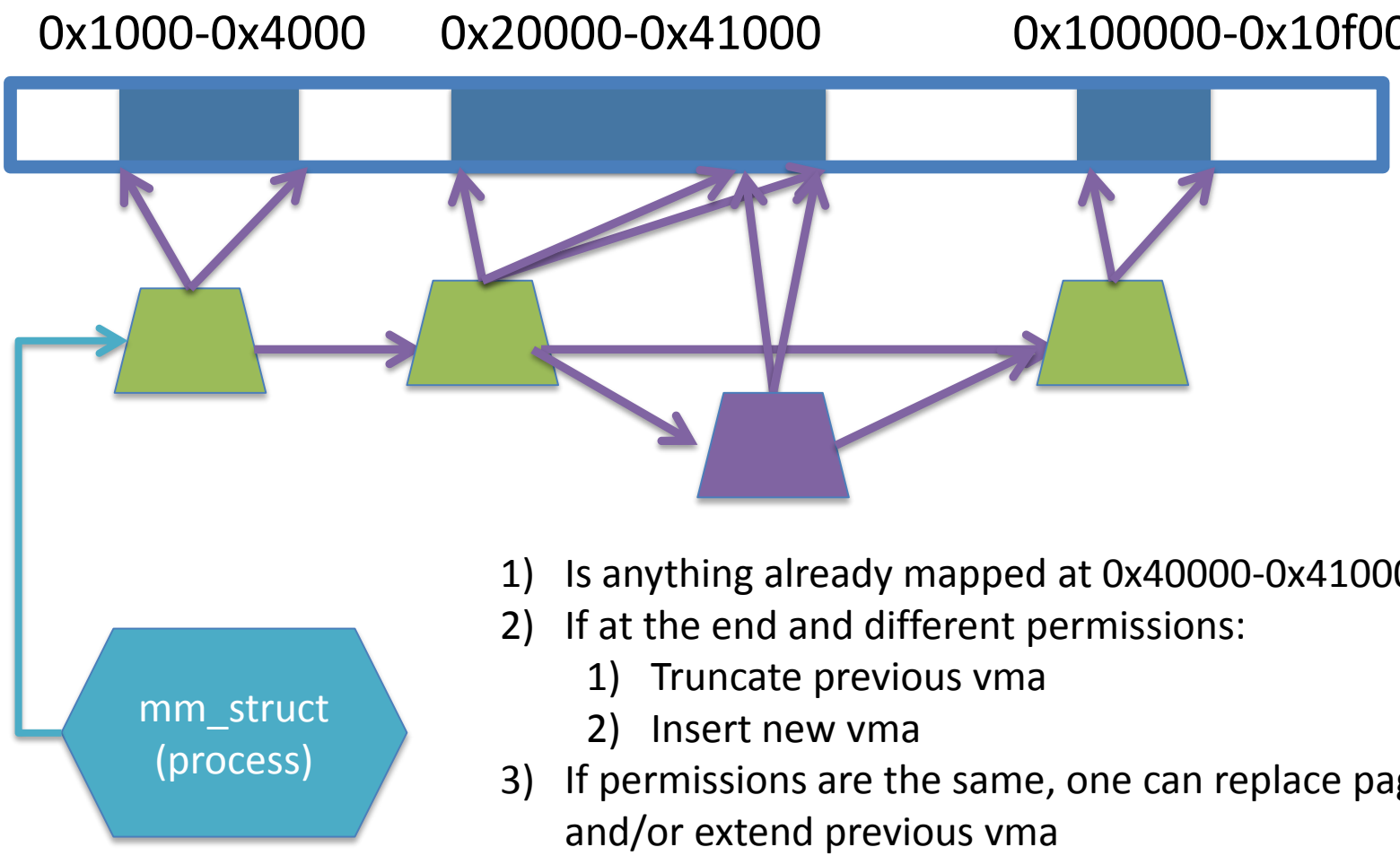


1) Is anything already mapped at 0x40000-0x41000?
2) If not, create a new vma and insert it
3) Recall: pages will be allocated on demand

Scenario 2

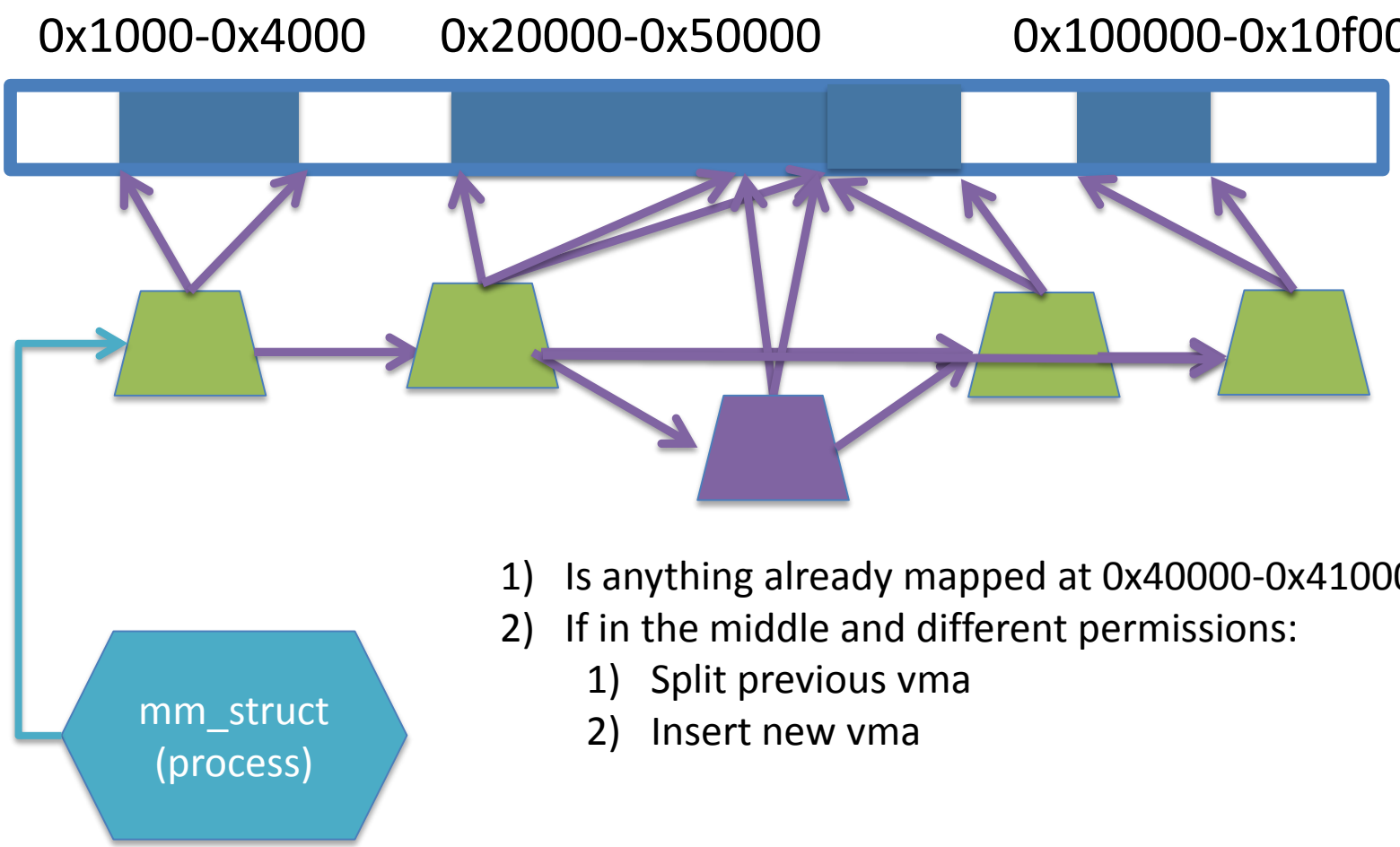
- What if there is something already mapped there with read-only permission?
  - Case 1: Last page overlaps
  - Case 2: First page overlaps
  - Case 3: Our target is in the middle

Case 1: Insert at 0x40000



1) Is anything already mapped at 0x40000-0x41000?
2) If at the end and different permissions:
   1) Truncate previous vma
   2) Insert new vma
3) If permissions are the same, one can replace pages and/or extend previous vma

Case 3: Insert at 0x40000



1) Is anything already mapped at 0x40000-0x41000?
2) If in the middle and different permissions:
   1) Split previous vma
   2) Insert new vma

Demand paging

- Memory mapping (vma) doesn't allocate
  - No need to set up physical memory or page table entries
- It pays to be lazy!
  - A program may never touch the memory it maps
    - Program may not use all code in a library
  - Save work compared to traversing up front
  - Hidden costs? Optimizations?
    - Page faults are expensive; heuristics could help performance

Copy-On-Write (COW)

- ***fork()*** duplicated the address space
  - Naïve approach would copy each page
- Most processes immediately exec()
  - Would need to immediately free all pages
  - Again, lazy is better!

How does COW work?

- Memory regions
  - New copies of each vma are allocated for child during fork
  - Traverse all pages in page table
    - Mark all pages read-only and COW
      - Can use one of the bits “unsued” by hardware in x86 for COW
- Pages in memory
  - Make a new, writeable copy on a write fault
    - In either parent or in child

Overarching Allocator Issues

- Fragmentation
- Cache behavior
  - Alignment (cache and word)
  - Coloring
- Implementation complexity
- Allocation and free latency
  - Synchronization/Concurrency

Fragmentation

- Undergrad review: What is it? Why does it happen?
- What is
  - Internal fragmentation?
    - Wasted space when you round an allocation up
  - External fragmentation?
    - Small chunks of free memory that are too small to be useful

Allocation Alignment

- Word
- Cacheline

Alignment (words)

```
struct foo {
        bit x;
        int y;
};
```

- Naïve layout: 1 bit for x, followed by 32 bits for y
- CPUs only do aligned operations
  - 32-bit ops start at addresses divisible by 32

Word alignment, cont.

- If fields of a data type are not aligned
  - Compiler must read low and high bits separately
    - Then put them together with << and |
  - No one wants to do this
- Compiler generally pads out structures
  - Waste 31 bits after x
  - Save a ton of code reinventing simple arithmetic
    - Code takes space in memory too!

Memory allocator + alignment

- Compilers allocate structures on ***word*** boundary
  - Otherwise, we have same problem as before
  - Code breaks if not aligned
- This often dictates some fragmentation

Cacheline Alignment

- Different issue, similar name
- Cache lines are bigger than words
  - ***Word***: 32-bits or 64-bits
  - ***Cache line (or block)***: 64-128 bytes on most CPUs
- Lines are the basic unit at which memory is cached
- Entire cache lines are ***coherent***
  - If one core updates an address, other cores see it
  - Internally, it means cache line has to move core to core

False sharing

Cache line

- These objects have nothing to do with each other
  - At program level, private to separate threads
- At cache level, CPUs are fighting for block

False sharing is BAD

- Leads to pathological performance problems
  - Super-linear slowdown in some cases
- Rule of thumb: any performance trend that is more than linear in the number of CPUs is probably caused by cache behavior

Strawman

- Round everything up to the size of a cache line
- Thoughts?
  - Wastes too much memory (fragmentation)
  - A bit extreme

Allocation Coloring

- "Color" allocations
- Allocations of the same color conflict in some way
  - Belong to same cache line
  - Maps to same location in cache
  - etc...
- Make consecutive allocations go to different colors
  - Reduces the probability of conflict
- Usually a good idea
  - Until it isn't - can back fire if not handled carefully

Kernel Allocators

- Three types of dynamic allocators available
  - Big objects (entire pages or page ranges)
    - Talked about before (***_get_free_page()***)
      - Just take pages off of the appropriate free list
  - Small kernel objects (e.g., VMAs)
    - Uses page allocator to get memory from system
    - Gives out small pieces
  - Small arbitrary-size chunks of memory
    - Looks very much like a user-space allocator
    - Uses page allocator to get memory from system

Memory Pools (kmem_caches)

- Each ***pool*** is an array of objects
  - To allocate, take element out of pool
  - Can use bitmap or list to indicate free/used
    - List is easier, but can't pre-initialize objects
- System creates pools for common objects at boot
  - If more objects are needed, have two options
    - Fail (out of resource – reconfigure kernel for more)
    - Allocate another page to expand pool

Problems with Complex Allocators

- Used to allocate arbitrary chunks of memory
  - Just like in user space
  - "SLAB" allocator in Linux
- Introduces a lot of complexity to the kernel
  - Lots of book-keeping of used/free
  - Avoid fragmentation, allow for fast allocation
- 2 groups upset
  - Users of very small systems
  - Users of large multi-processor systems

Small systems

- 4MB of RAM on a small device/phone/etc…
- Bookkeeping overheads become huge
  - Large percent of total system memory for book-keeping
- How bad is fragmentation really going to be?
  - Not that much memory to fragment anyway

Large systems

- Very large (thousands of CPU) systems
- Book-keeping has to work in multi-proc environment
- Tons of synchronized state
  - Slow to allocate/free
  - Hard to move memory around

SLOB Allocator

- Simple List Of Blocks
  - Just keep a free list of each available chunk and its size
- Grab the first one big enough to work
  - Split block if leftover bytes
- No internal fragmentation, obviously
- External fragmentation? Yes.
  - Traded for low overheads
  - Worst-case scenario?
    - Allocate fails, phone crashes (don't use in pacemaker)

SLUB Allocator

- The Unqueued Slab Allocator
- All objects of same size from same slab
  - Simple free list per slab
  - No cross-CPU complexity
- Some internal fragmentation
- Effectively the same as pool allocator
  - Round up requested size to next largest pool

SLUB Status

- Does better than SLAB in many cases
- Still has some performance pathologies
  - Not universally accepted
- General-purpose allocation is tricky business

User-Space Memory Allocation

- Can be a lot more complex than kernel allocation
  - And usually is
- Many flavors around
  - glibc in Linux, jemaloc in FreeBSD libc
  - Boehm (garbage collected), Hoard (highest performing), …
  - Buddy allocators (usually implemented in undergrad)
- Differ in many aspects (no size fits all)
  - Fragmentation, performance, multi-proc
- Will go over two
  - Simplest (bump) and realistic (Hoard)

Bump allocator

- malloc (12)
- malloc(20)
- malloc (5)

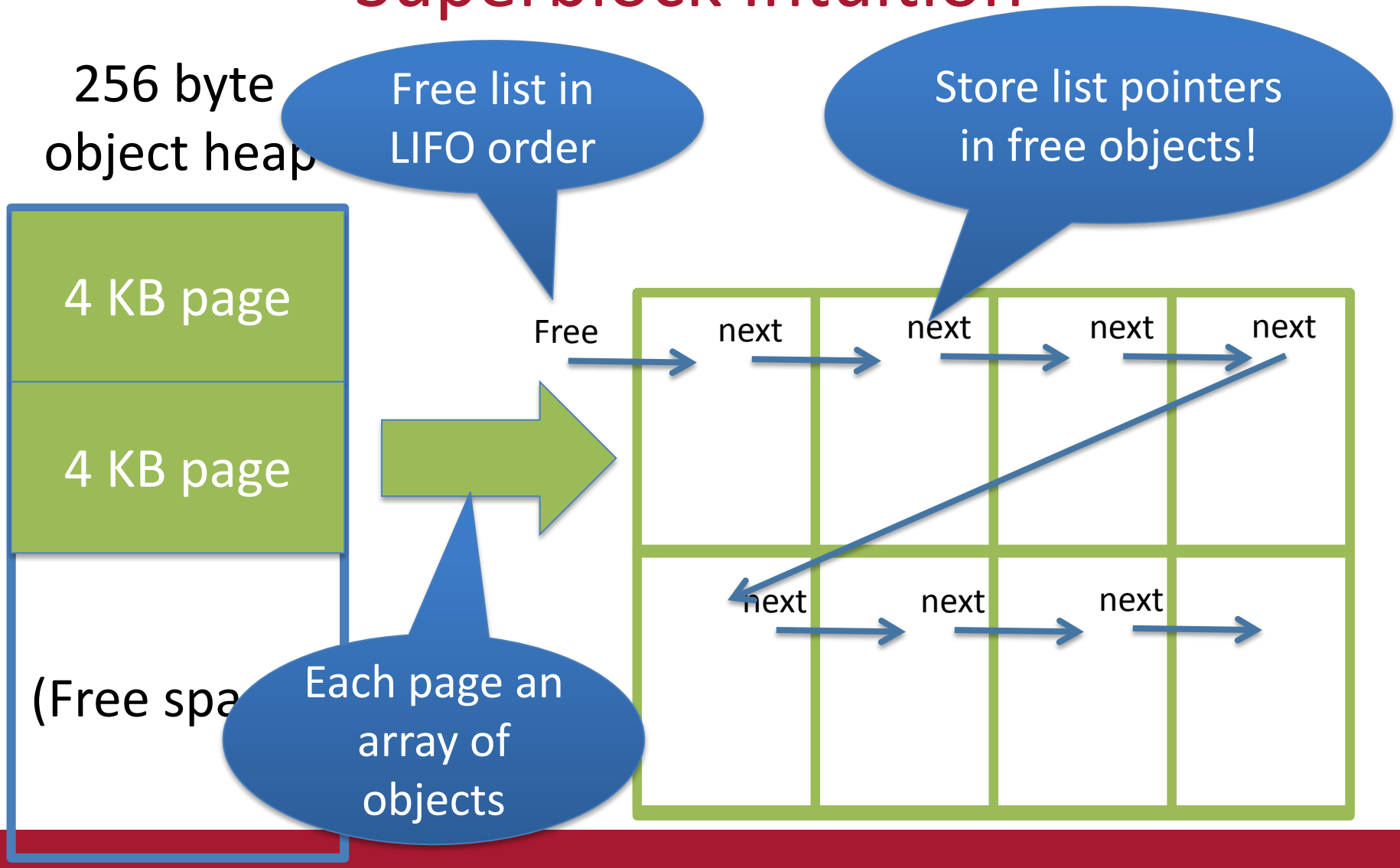
Bump allocator

- Simply “bumps” up the free pointer
- How does free() work? It doesn’t
  - Could try to recycle cells, needs complicated bookkeeping
- Controversial observation: ideal for simple programs
  - Only care about free() if you need the memory

Hoard: Superblocks

- At a high level, allocator operates on superblocks
  - Chunk of (virtually) contiguous pages
  - All objects in a superblock are the same size
- A given superblock is an array of same-sized objects
  - They generalize to “powers of b > 1”;
  - In usual practice, b == 2

Superblock Intuition



Superblock Intuition

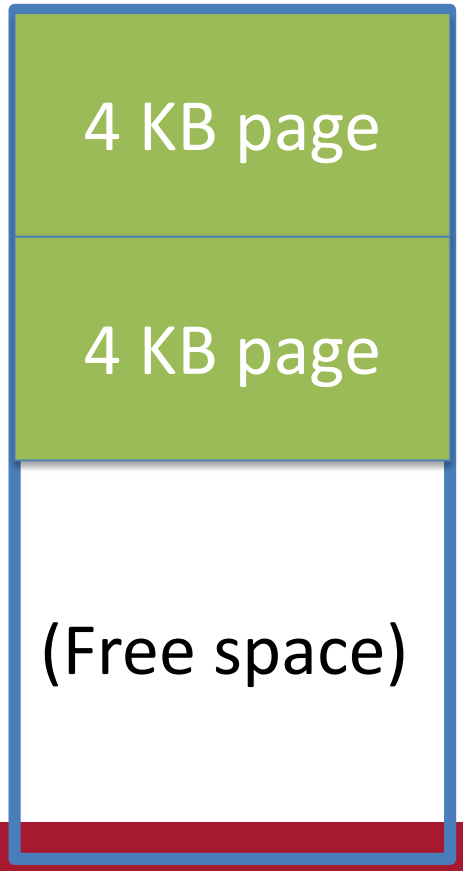
```
malloc (8);
```

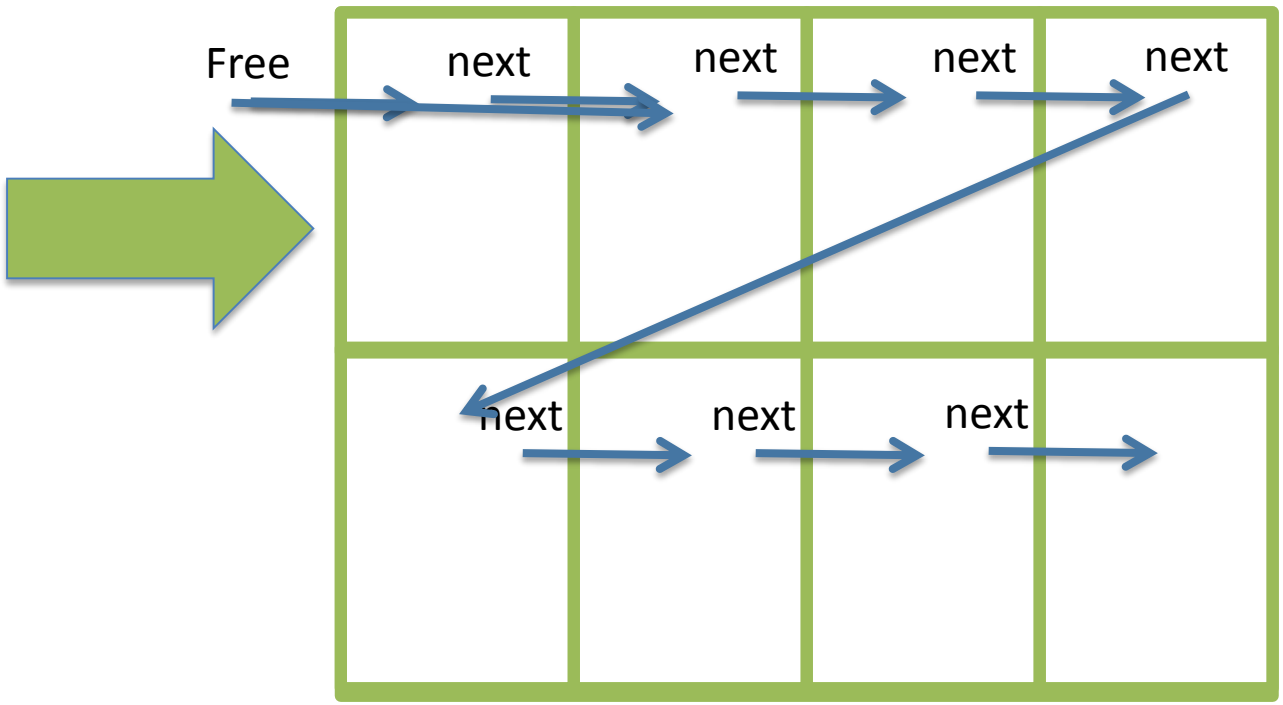
1) Find the nearest power of 2 heap (8)
2) Find free object in superblock
3) Add a superblock if needed. Goto 2.

malloc (200)

256 byte object heap

Pick first free object





Superblock Example

- Suppose program allocates objects of sizes:
  - 4, 5, 7, 34, and 40 bytes.
- How many superblocks do I need (if b ==2)?
  - 3 – (4, 8, and 64 byte chunks)
- Bounded internal fragmentation
  - Can't be more than 50%
  - Gives up some space to bound worst case and complexity

Memory Free

- Simple most-recently-used list for a superblock
- How do you tell which superblock an object is from?
  - Keep a pointer to superblock in allocation's "header"
  - Also indicates if allocation was mmap()ed
    - In that case, munmap() instead of returning to superblock

Big objects

- If alloc is bigger than ½ superblock, just mmap() it
  - A superblock is on the order of pages already
- What about fragmentation?
  - Example: 4097 byte object (1 page + 1 byte)
  - More trouble than it is worth
    - Extra bookkeeping
    - Potential contention
    - Potential bad cache behavior

LIFO

- Why objects re-allocated most-recently used first?
  - Aren't all good OS heuristics FIFO?
  - More likely to be already in cache (hot)
    - CPU caches are faster than memory
  - Easy to manage list

High-level strategy

- Allocate a heap for each CPU, and one shared heap
  - Note: not threads, but CPUs
  - Can only use as many heaps as CPUs at once
  - Requires some way to figure out current processor
- Try per-CPU heap first
- If no free blocks of right size, then try global heap
- If that fails, get another superblock for per-CPU heap

Hoard Simplicity

- Alloc and free is straightforward
  - Esp. when compared to other allocators
  - Overall: Need a simple array of (# CPUs + 1) heaps
- Per heap: 1 list of superblocks per object size
- Per superblock:
  - Need to know which/how many objects are free
    - LIFO list of free blocks

Hoard Locking

- On alloc and free, superblock and local heap locked
- Why?
  - An object can be freed from another CPU than allocated

Locking performance

- Acquiring and releasing a lock generally requires an atomic instruction
  - Tens to a few hundred cycles vs. a few cycles
- Waiting for a lock can take thousands
  - Depends on how good the lock implementation is at managing contention (spinning)
  - Blocking locks require many hundreds of cycles to context switch

Performance argument

- Common case: allocations and frees are from per-CPU heap
- Yes, grabbing a lock adds overheads
  - But better than the fragmented or complex alternatives
  - And locking hurts scalability only under contention
- Uncommon case: all CPUs contend to access one heap
  - Had to all come from that heap (only frees cross heaps)
  - Bizarre workload, probably won't scale anyway

Hoard strategy (pragmatic)

- Rounding up to powers of 2 helps
  - Once your objects are bigger than a cache line
- Locality observation: things tend to be used on the CPU where they were allocated
- For small objects, always return free to the original heap
  - Remember idea about extra bookkeeping to avoid synchronization: some allocators do this
    - Save locking, but introduce false sharing!

Hoard strategy (2)

- Thread A can allocate 2 small objects from the same line
- “Hand off” 1 to another thread to use; keep using 2nd
- This will cause false sharing
- Question: is this really the allocator’s job to prevent this?

Where to draw the line?

- Encapsulation should match programmer intuitions
  - (my opinion)
- In the hand-off example:
  - Hard for allocator to fix
  - Programmer would have reasonable intuitions (after 506)
- If allocator just gives parts of same lines to different threads
  - Hard for programmer to debug performance

Memory Management Wrapup

- Hoard is a really nice piece of work
  - Establishes nice balance among concerns
  - Good performance results
- For CSE506, need a user-level allocator
  - Will go into libc/ (e.g., libc/malloc.c)
  - Can use an existing one (e.g., Hoard)
    - But make sure license is compatible
    - Must support all syscalls needed by third-party allocator
  - Can implement own allocator
    - No need to worry about syscall interface (create your own)
    - Must take care to not leak memory on free()