CS5460/6460: Operating Systems

Lecture 27: Recap

Anton Burtsev
April, 2014

File systems

The role of file systems

- Sharing
  - Sharing of data across users and applications
- Persistence
  - Data is available after reboot

Crash recovery

- File systems must support crash recovery
  - A power loss may interrupt a sequence of updates
  - Leave file system in inconsistent state
    - E.g. a block both marked free and used

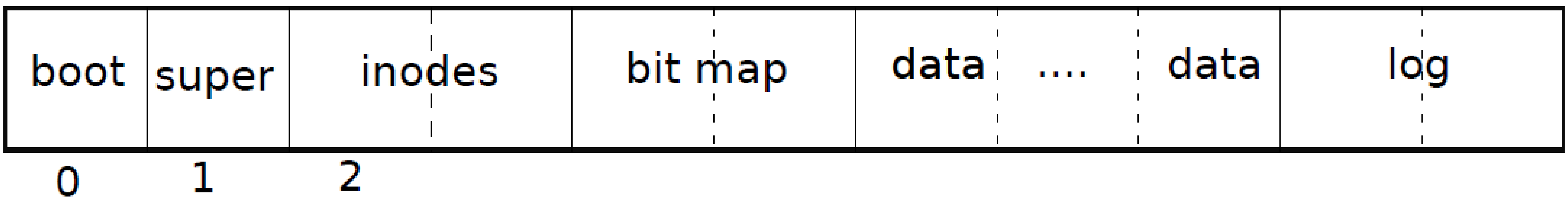
Speed

- Access to a block device is several orders of magnitude slower
  - Memory: 200 cycles
  - Disk: 20 000 000 cycles
- A file system must maintain a cache of disk blocks in memory

FS/Block Layer Stack

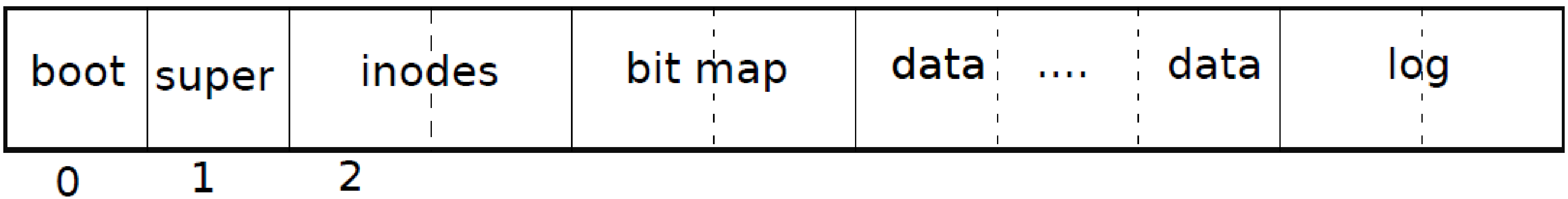
System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

File system layout on disk



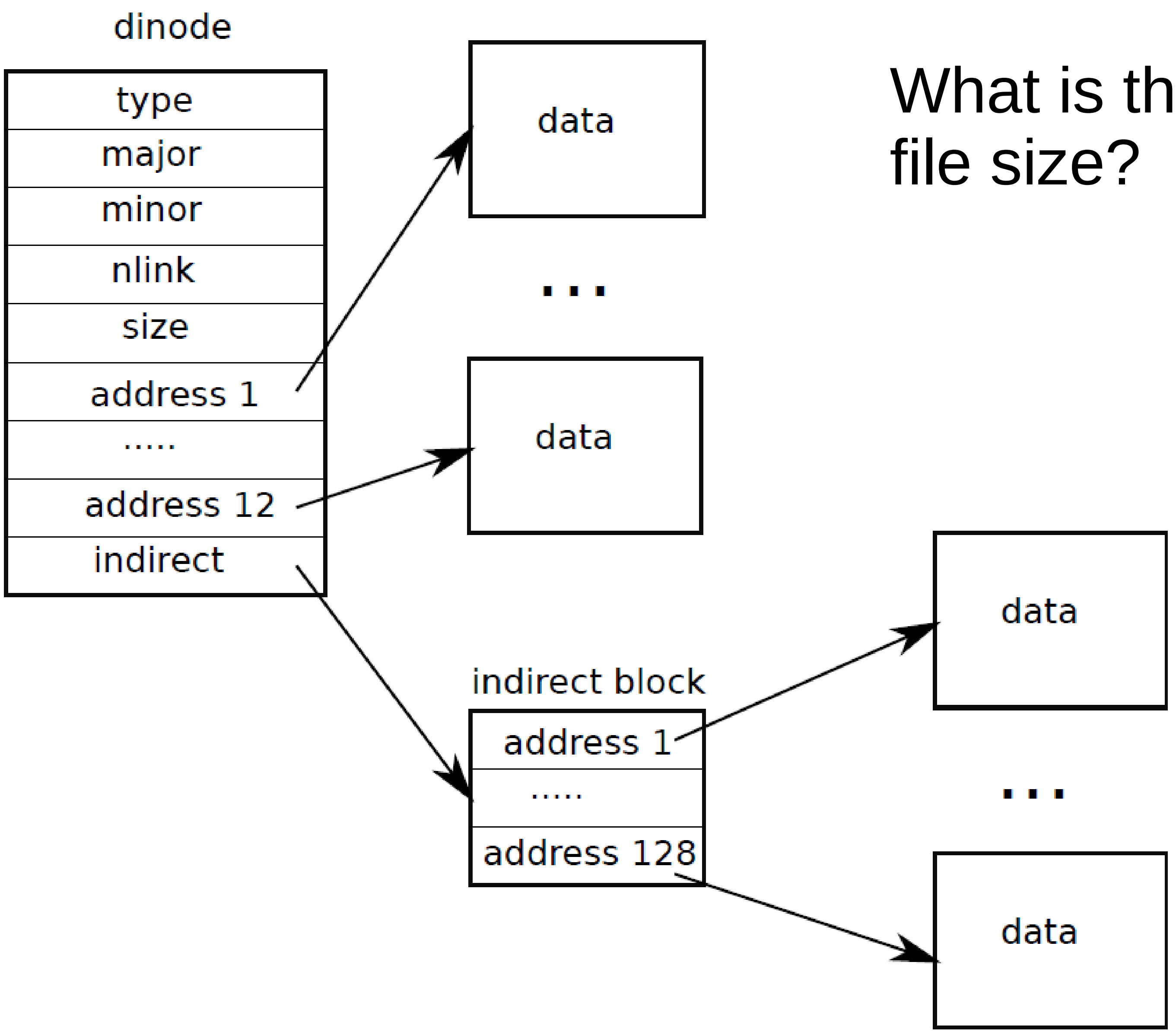
- Block #0: Boot code
- Block #1: Metadata about the file system
  - Size (number of blocks)
  - Number of data blocks
  - Number of inodes
  - Number of blocks in log

File system layout on disk



- Block #2 (inode area)
- Bit map area: track which blocks are in use
- Data area: actual file data
- Log area: maintaining consistency in case of a power outage or system crash

Representing files on disk



What is the max file size?

Logging layer

- Consistency
  - File system operations involve multiple writes to disk
  - During the crash, subset of writes might leave the file system in an inconsistent state
  - E.g. file delete can crash leaving:
    - Directory entry pointing to a free inode
    - Allocated but unlinked inode

Logging

- Writes don't directly go to disk
  - Instead they are logged in a journal
  - Once all writes are logged, the system writes a special commit record
    - Indicating that log contains a complete operation
- At this point file system copies writes to the on-disk data structures
  - After copy completes, log record is erased

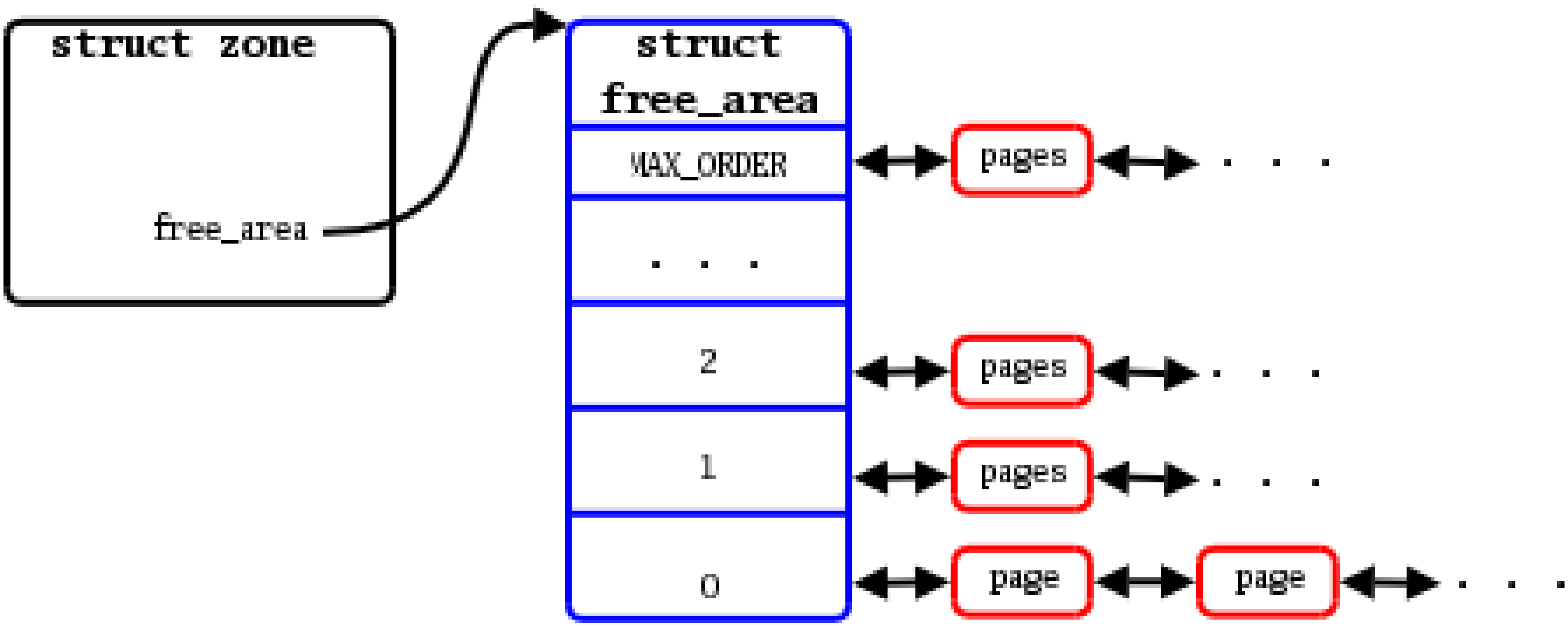
Recovery

- After reboot, copy the log
  - For operations marked as complete
    - Copy blocks to disk
  - For operations partially complete
    - Discard all writes
    - Information might be lost (output consistency, e.g. can launch the rocket twice)

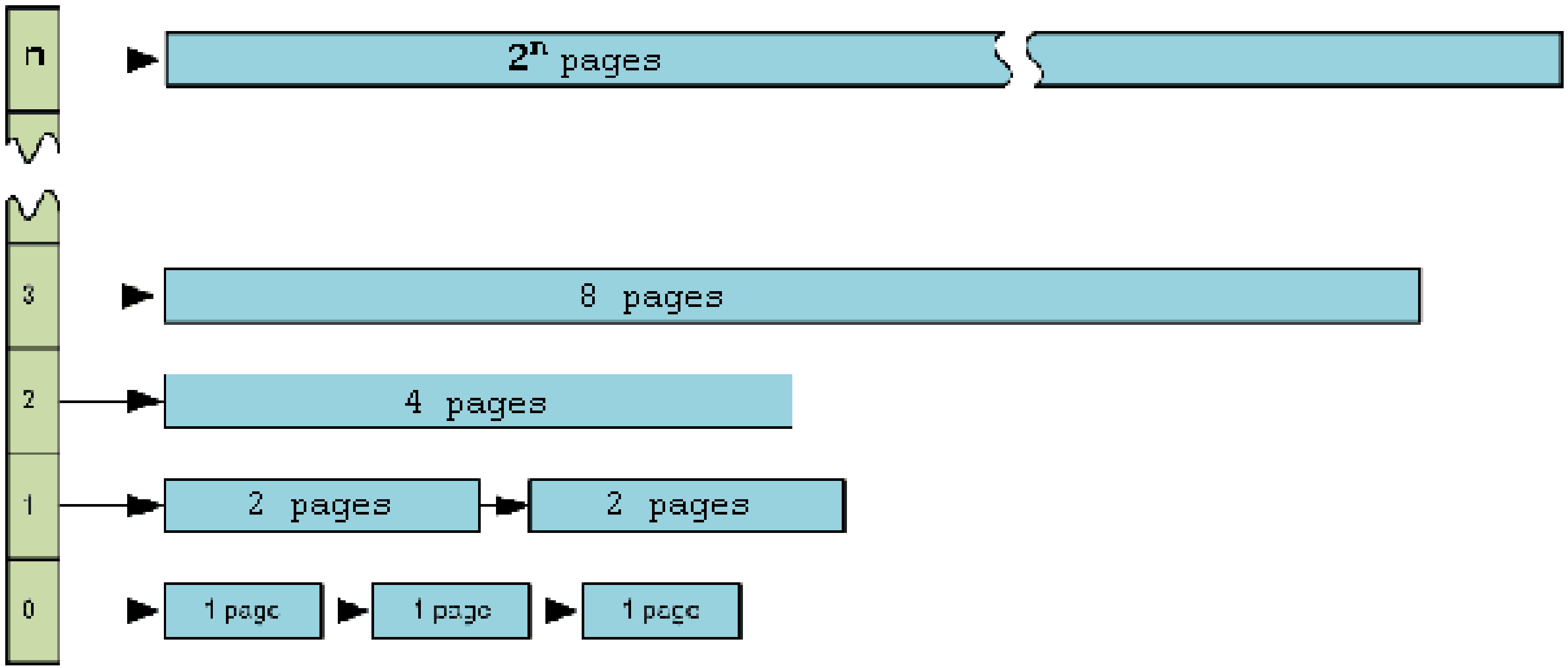
Memory management

Buddy memory allocator

- Each zone has a buddy allocator



Buddy allocator

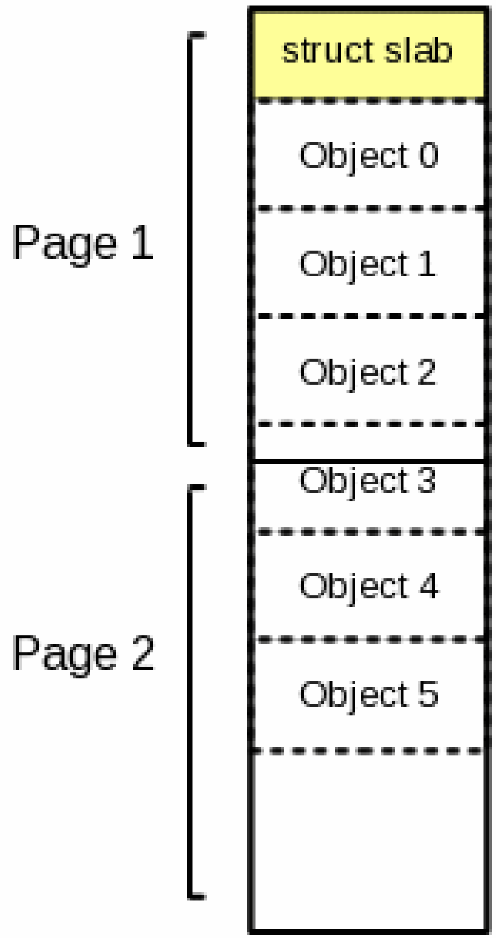


Slab allocator

- Buddy allocator is ok for large allocations
  - E.g. 1 page or more
- But what about small allocations?
  - Buddy uses the whole page for a 4 bytes allocation
    - Wasteful
  - Buddy is still slow for short-lived objects

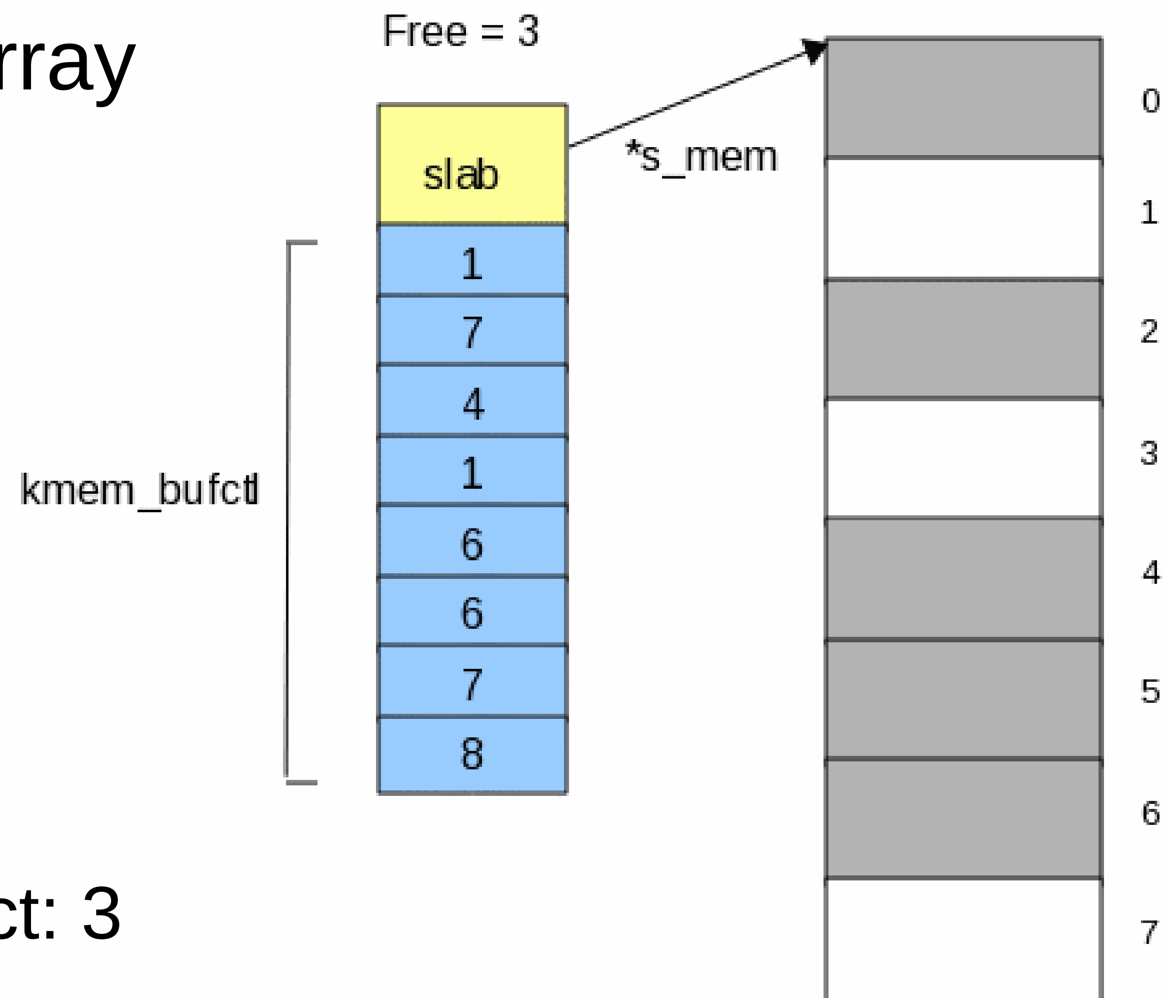
Slab

- A 2 page slab with 6 objects



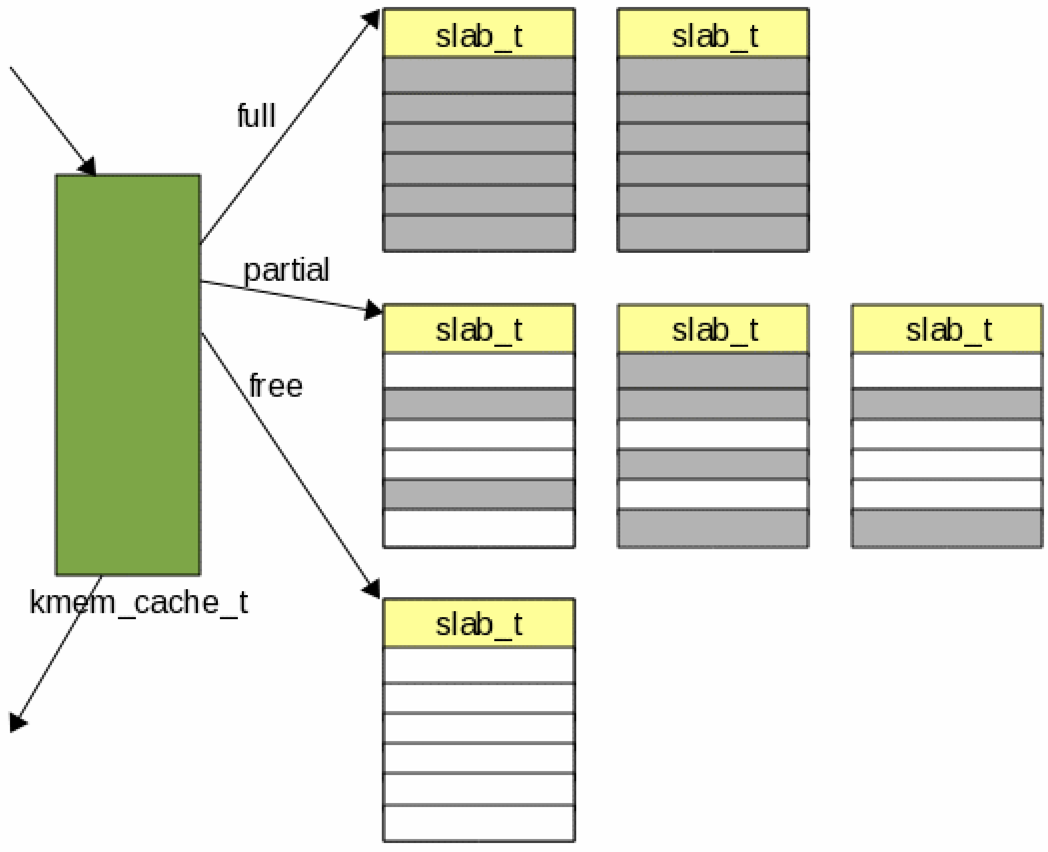
Keeping track of free objects

- kmem_bufctl array is effectively a linked list



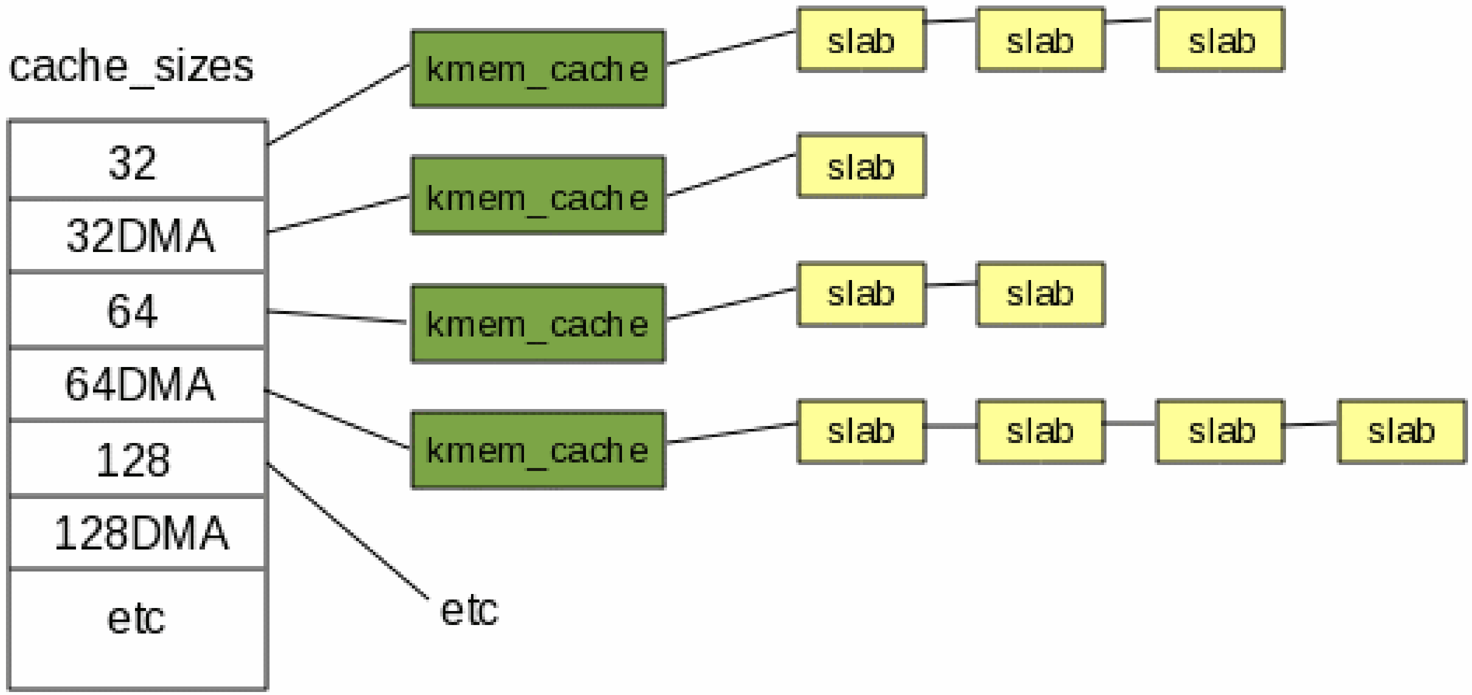
- First free object: 3
- Next free object: 1

A cache is formed out of slabs

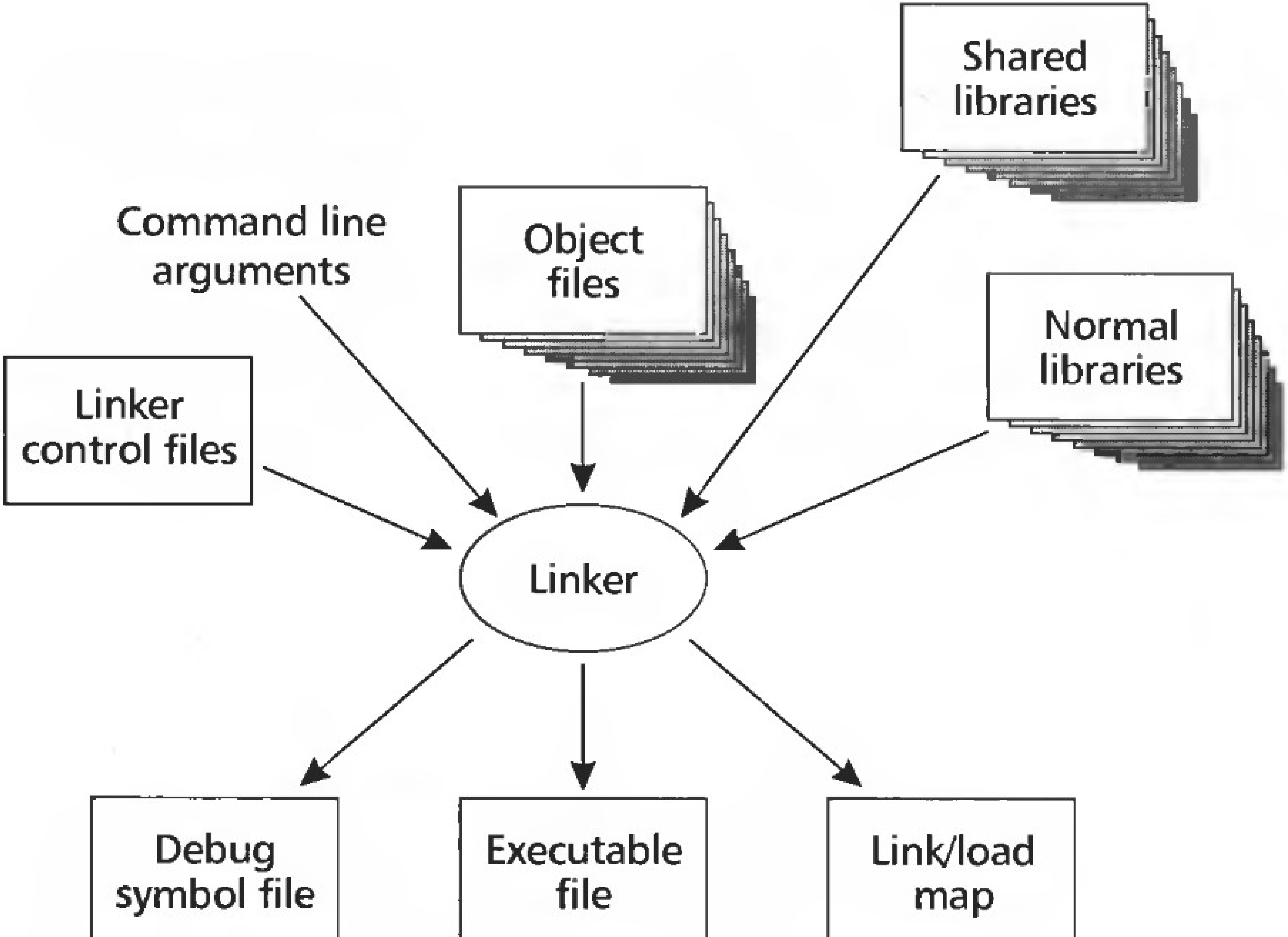


Kmalloc(): variable size objects

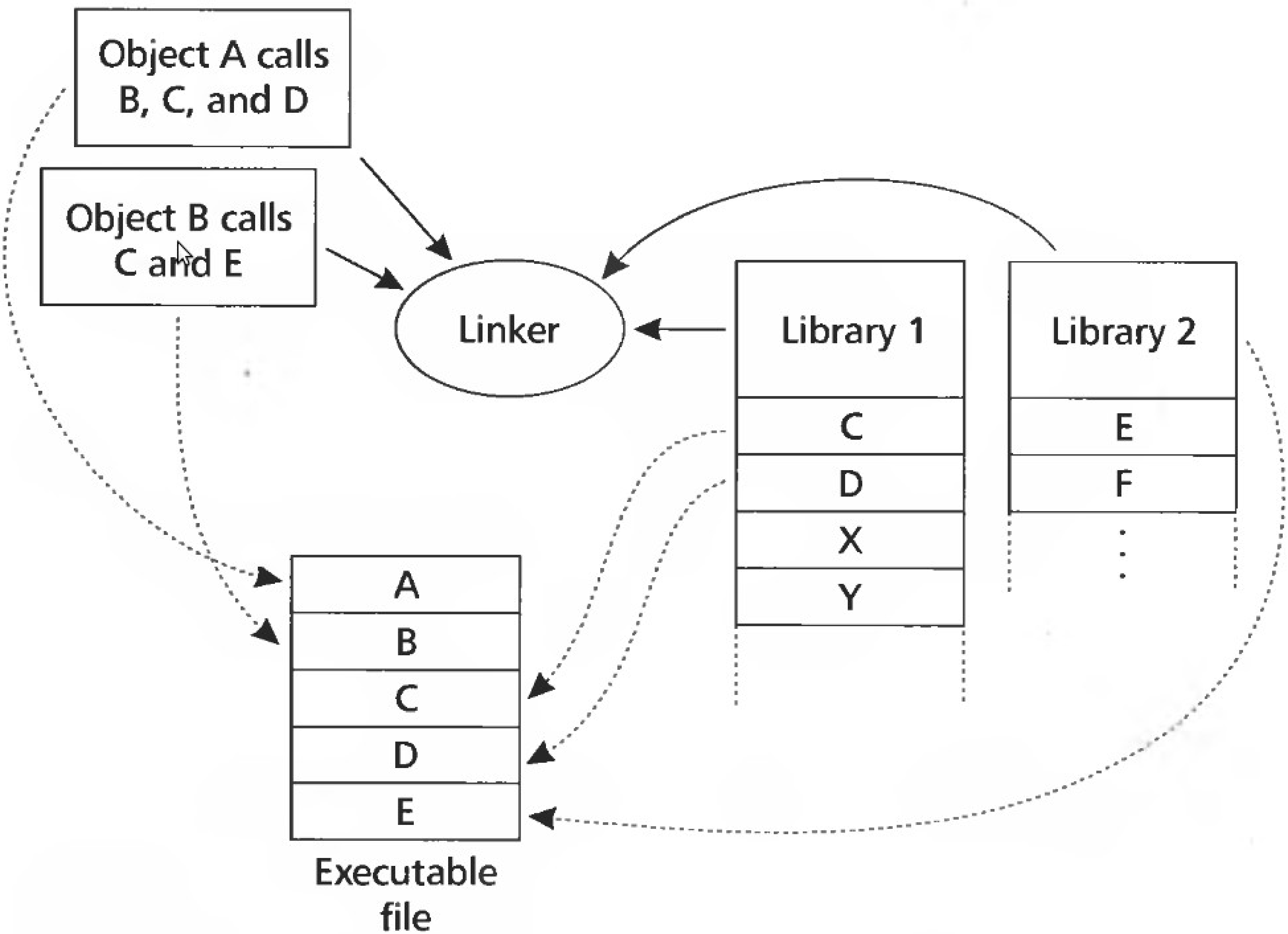
- A table of caches
  - Size: 32, 64, 128, etc.



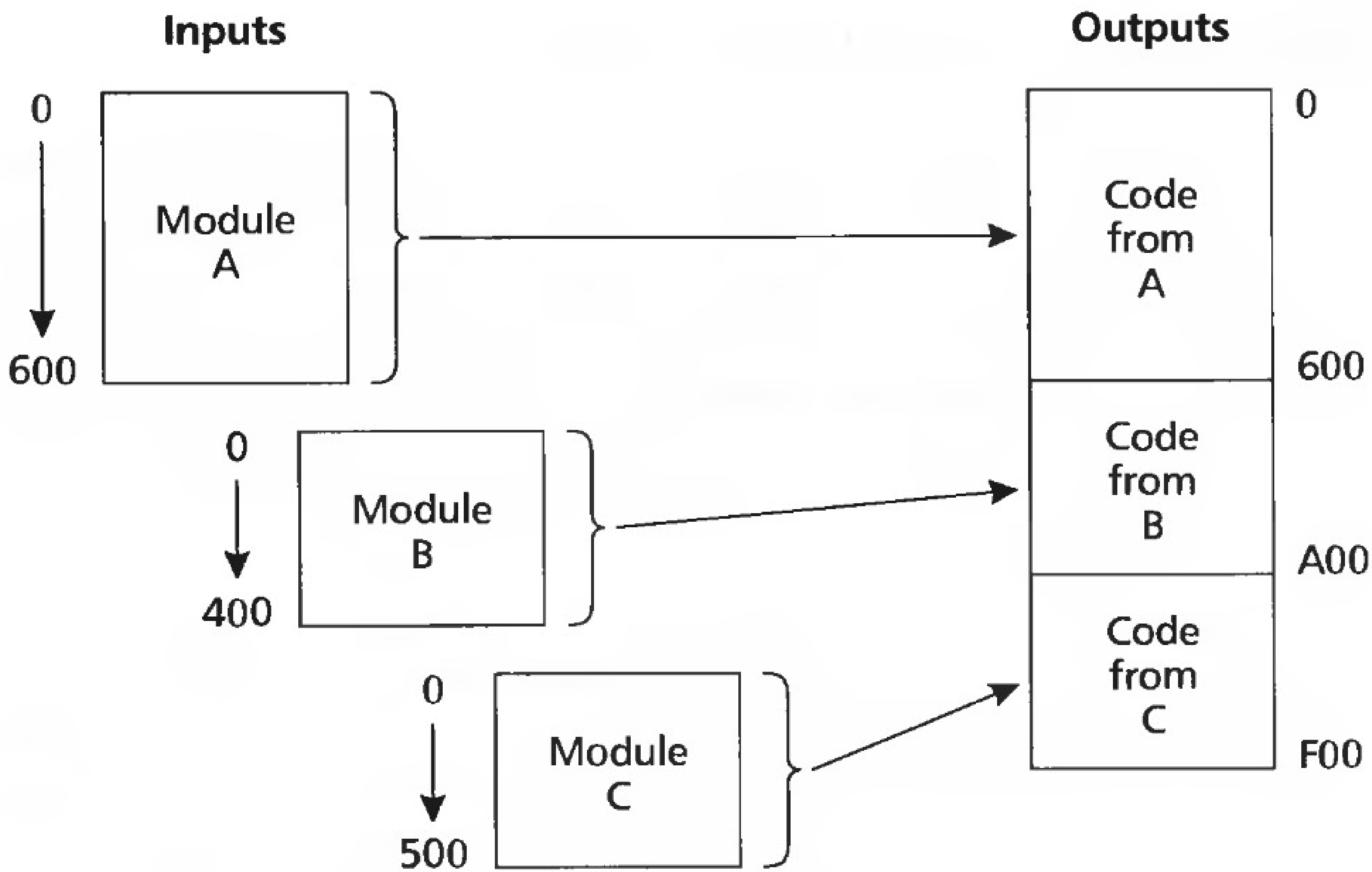
Linking and loading


Shared libraries
Command line arguments
Object files
Normal libraries
Linker control files
Linker
Debug symbol file
Executable file
Link/load map

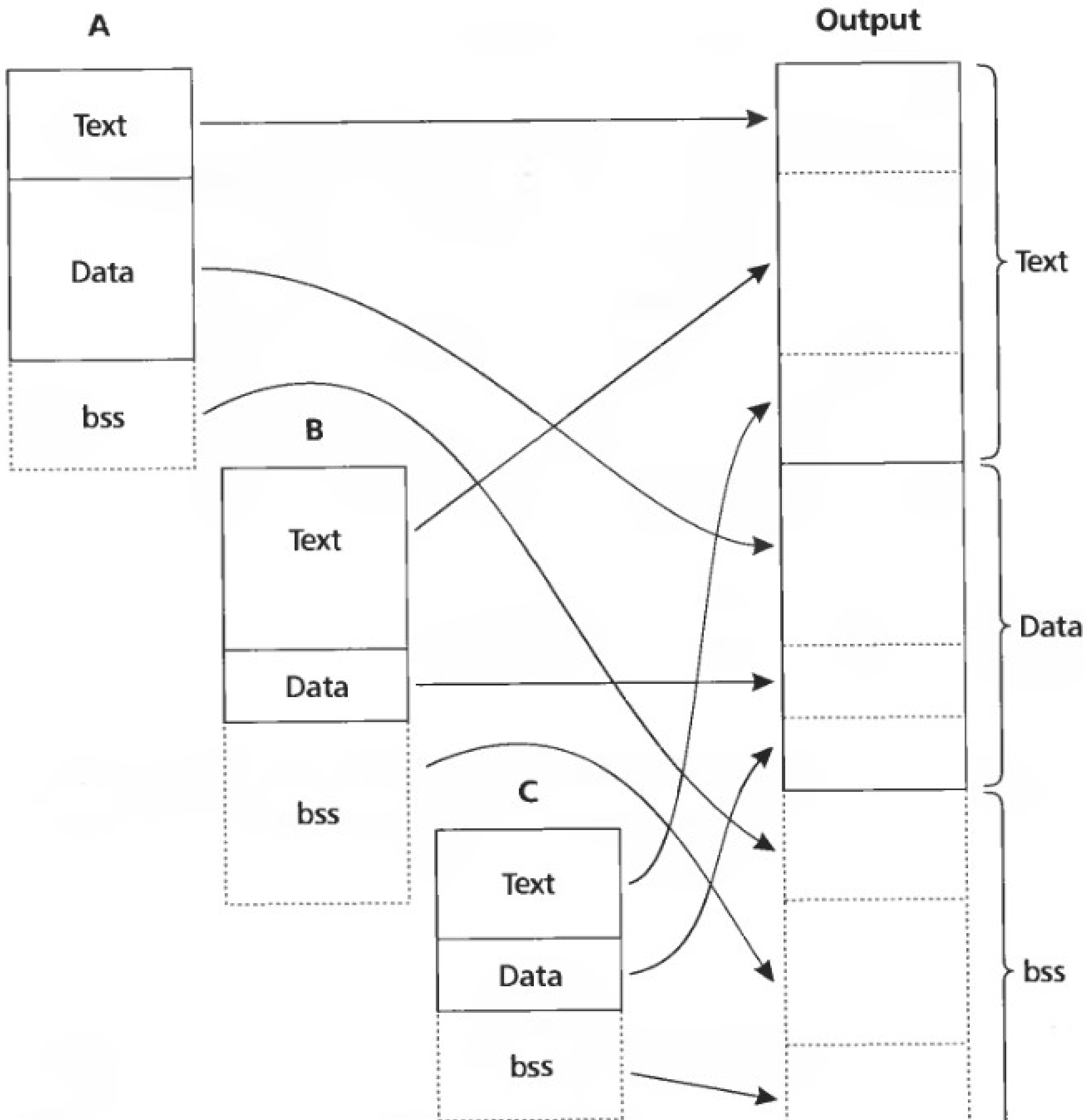
- Input: object files (code modules)
- Each object file contains
  - A set of segments
    - Code
    - Data
  - A symbol table
    - Imported & exported symbols
- Output: executable file, library, etc.


Object A calls
B, C, and D
Object B calls
C and E
Linker
Library 1
C
D
X
Y
Library 2
E
F
A
B
C
D
E
Executable
file

Multiple object files



Merging segments



Relocation, why?

- Each program gets its own private space, why relocate?
  - Linkers combine multiple libraries into a single executable
  - Each library assumes private address space
    - E.g., starts at 0x0
- Is it possible to go away with segments?
  - Each library gets a private segment (starts at 0x0)
  - All cross-library references are patched to use segment numbers
- Possible!
  - But slow.
  - Segment lookups are slow

Relocation

- Each relocatable object file contains a relocation table
  - List of places in each segment that need to be relocated
  - Example:
    - Pointer in the text segment points to offset 200 in the data segment
    - Input file: text starts at 0, data starts at 2000, stored pointer has value 2200
    - Output file: Data segment starts at 15000
      - Linker adds relocated base of the data segment 13000 (DR)
    - Output file: will have pointer value of 15200
  - All jumps are relative on x86
    - No need to relocate
    - Unless its a cross-segment jump, e.g. text segment to data segment

Types of object files

- Relocatable object files (.o)
- Static libraries (.a)
- Shared libraries (.so)
- Executable files

- We looked at A.OUT, but Unix has a general format capable to hold any of these files

ELF

Elf header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Segment header table

- Page size, virtual addresses memory segments (sections), segment sizes.

`.text` section

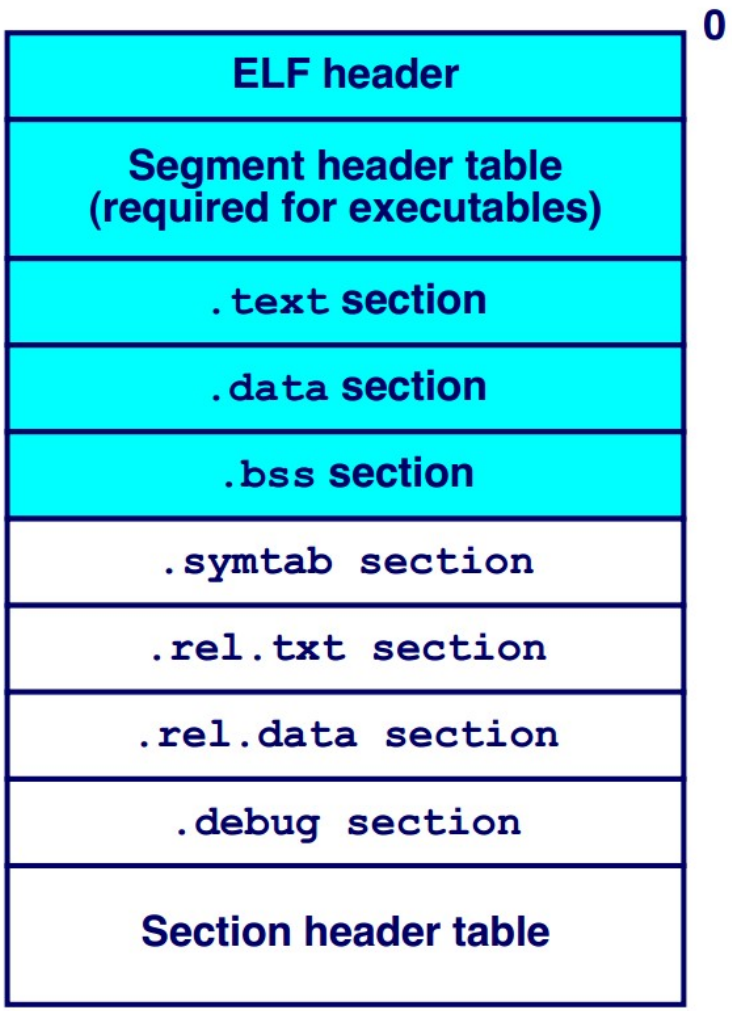
- Code

`.data` section

- Initialized global variables

`.bss` section

- Uninitialized global variables
- “Block Started by Symbol”
- “Better Save Space”
- Has section header but occupies no space



ELF (continued)

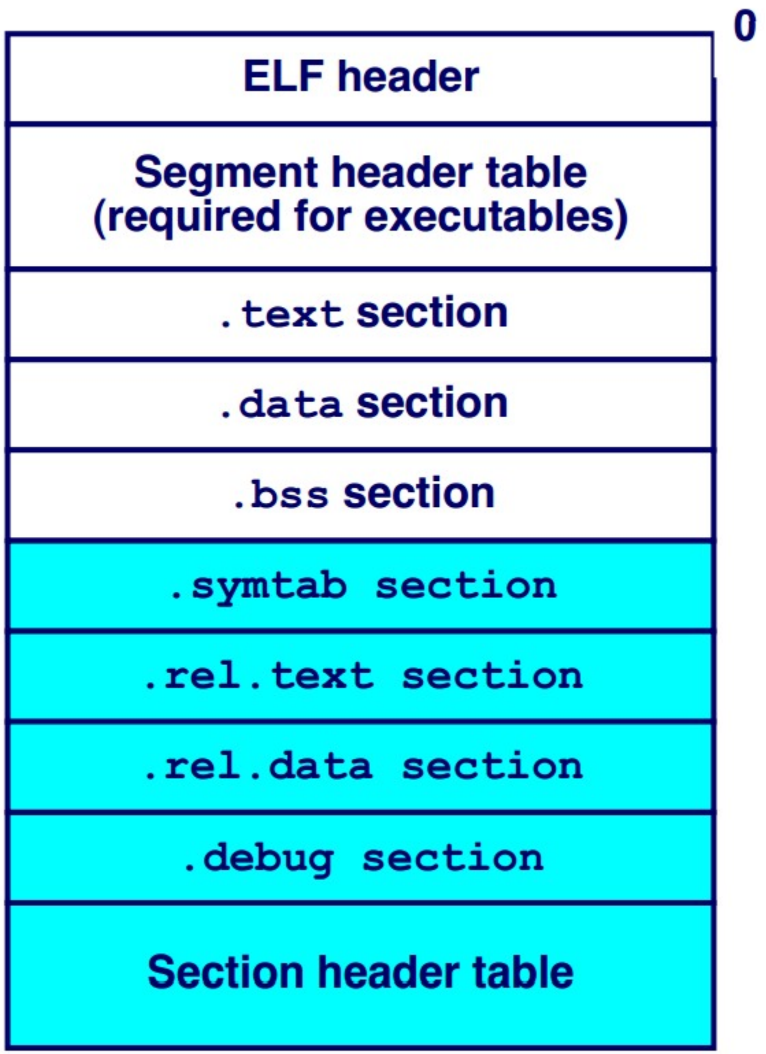
`.symtab` section
- Symbol table
- Procedure and static variable names
- Section names and locations

`.rel.text` section
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

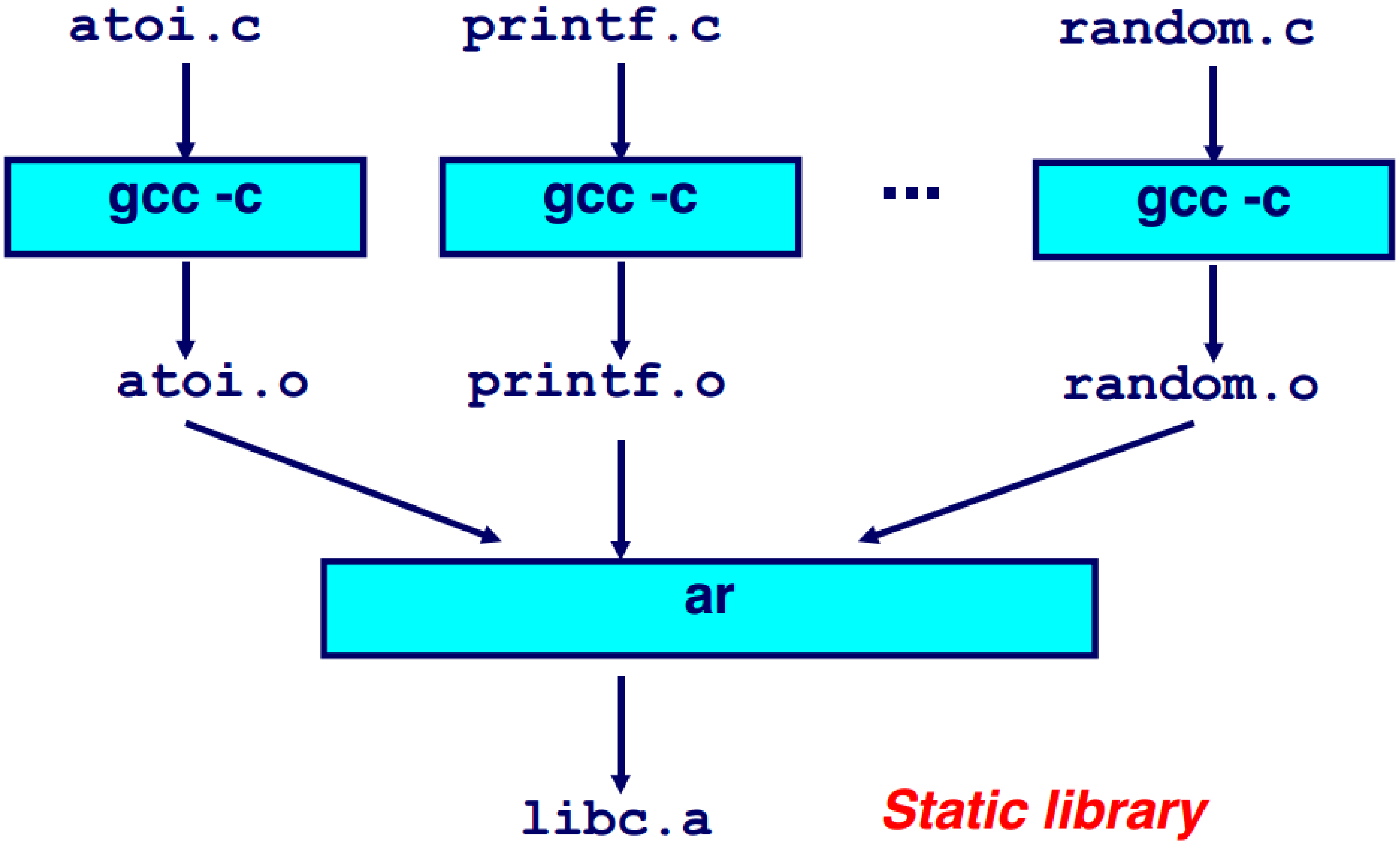
`.rel.data` section
- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` section
- Info for symbolic debugging (`gcc -g`)

Section header table
- Offsets and sizes of each section



Creating a static library



Searching libraries

- First linker path needs resolve symbol names into function locations
- To improve the search library formats add a directory
  - Map names to member positions

Shared libraries

Motivation

- 1000 programs in a typical UNIX system
- 1000 copies of printf

- **How big is printf actually?**

Motivation

- 1000 programs in a typical UNIX system
- 1000 copies of printf
  - **Printf is a large function**
  - **Handles conversion of multiple types to strings**
  - **5-10K**
- This means 5-10MB of disk is wasted on printf
- Runtime memory costs are
  - 10K x number of running programs

Position independent code

- Motivation
  - Share code of a library across all processes
    - E.g. libc is linked by all processes in the system
  - Code section should remain identical
    - To be shared read-only
  - What if library is loaded at different addresses?
    - Remember it needs to be relocated

Position independent code (PIC)

- Main idea:
  - Generate code in such a way that it can work no matter where it is located in the address space
  - Share code across all address spaces

What needs to be changed?

- Can stay untouched
  - Local jumps and calls are relative
  - Stack data is relative to the stack
- Needs to be modified
  - Global variables
  - Imported functions

Example

```
000010a4 <_main>:
    10a4: 55             pushl %ebp
    10a5: 89 e5          movl %esp,%ebp
    10a7: 68 10 00 00 00 pushl $0x10
      10a8: 32 .data
    10ac: e8 03 00 00 00 call 10b4 <_a>
  ...
000010b4 <_a>:
    10bc: e8 37 00 00 00 call 10f8 <_strlen>
      ...
    10c3: 6a 01 pushl $0x1
    10c5: e8 a2 00 00 00 call 116c <_write>
    ...
```

- Reference to a data section
  - Code and data sections can be moved around

Example

```
000010a4 <_main>:
    10a4: 55             pushl %ebp
    10a5: 89 e5          movl %esp,%ebp
    10a7: 68 10 00 00 00 pushl $0x10
      10a8: 32 .data
    10ac: e8 03 00 00 00 call 10b4 <_a>
  ...
000010b4 <_a>:
    10bc: e8 37 00 00 00 call 10f8 <_strlen>
      ...
    10c3: 6a 01 pushl $0x1
    10c5: e8 a2 00 00 00 call 116c <_write>
    ...
```

- Local function invocations use relative addresses
  - No need to relocate

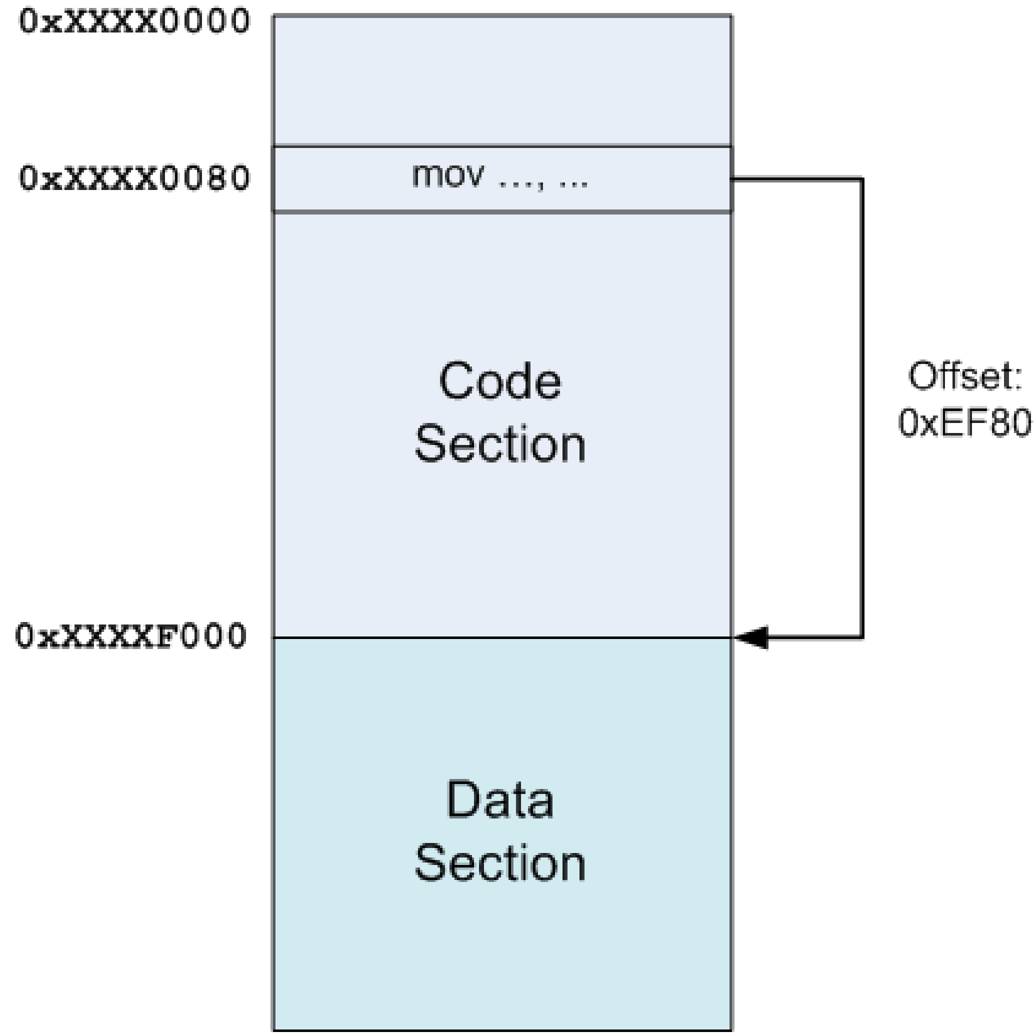
Position independent code

- How would you build it?

Position independent code

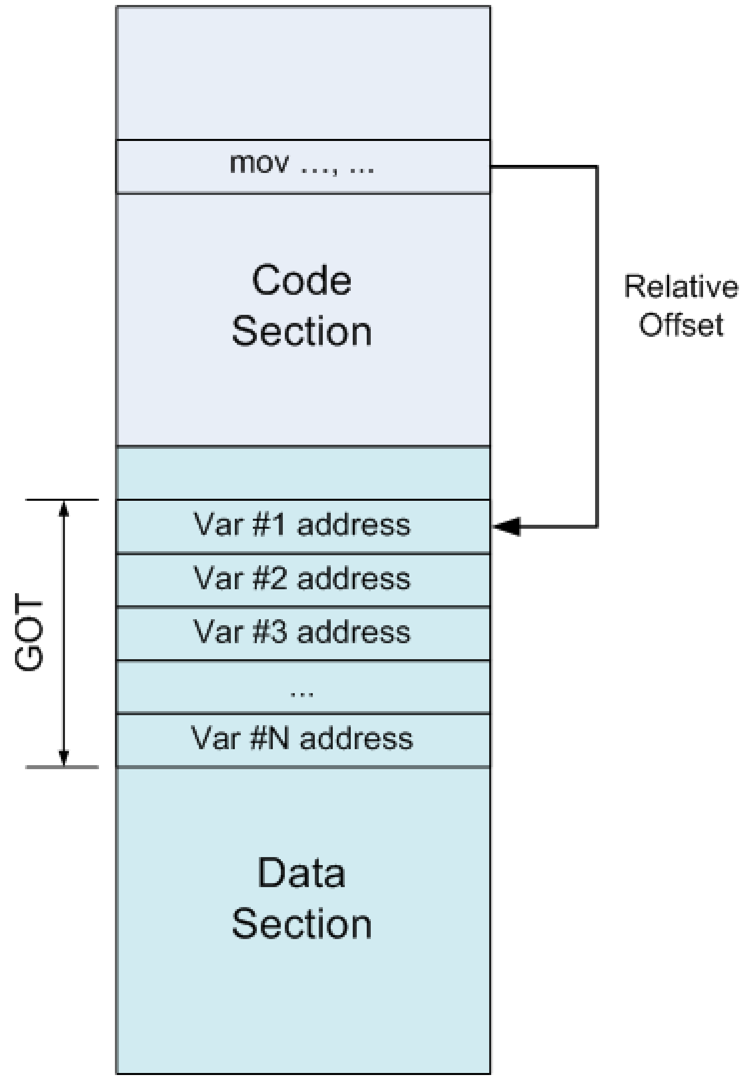
- Main insight
  - Code sections are followed by data sections
  - The distance between code and data **remains constant even if code is relocated**
    - Linker knows the distance
    - Even if it combines multiple code sections together

Insight 1: Constant offset between text and data sections



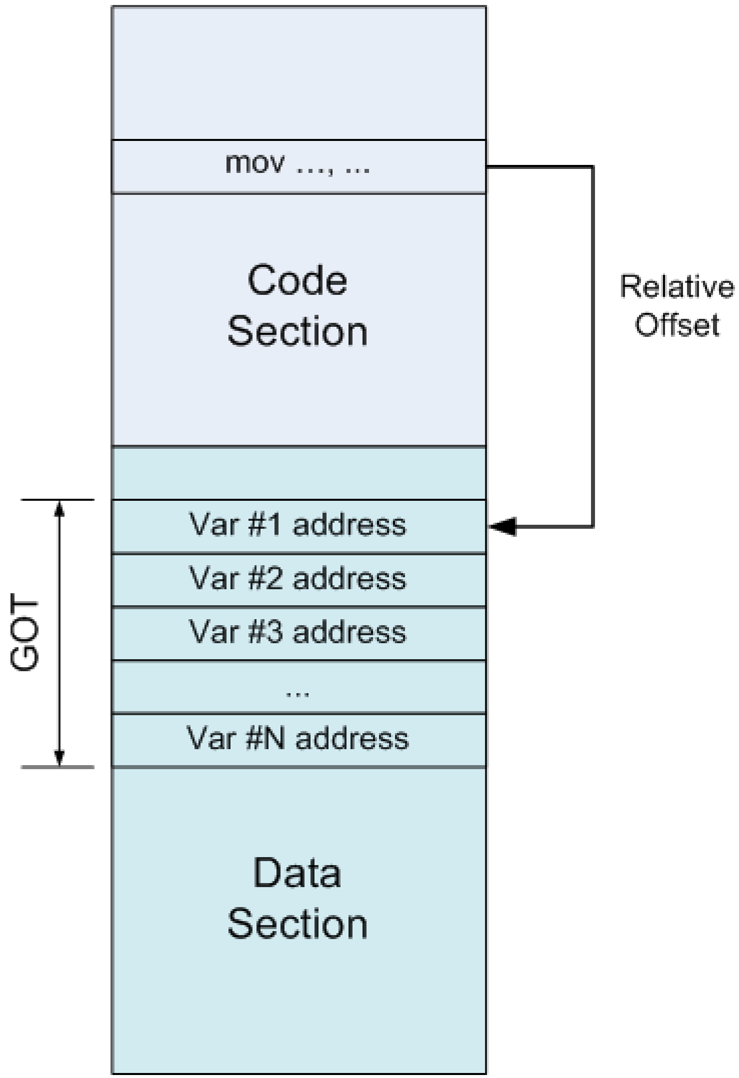
Global offset table (GOT)

- Insight #2:
  - Instead of referring to a variable by its absolute address
  - Refer through GOT



Global offset table (GOT)

- GOT
  - Table of addresses
  - Each entry contains absolute address of a variable
  - GOT is patched by the linker at relocation time



How to find position of the code in memory at run time?

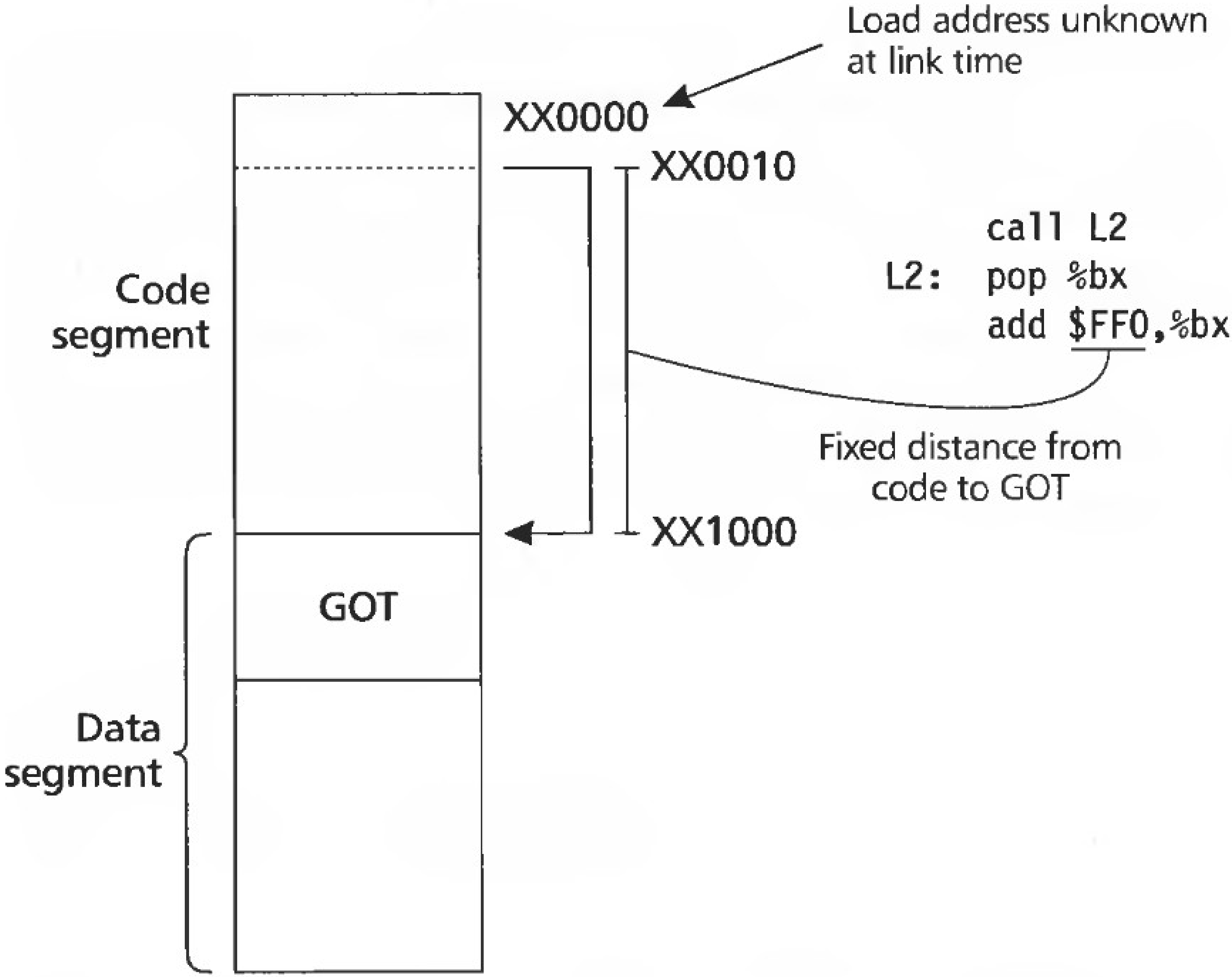
- Is there an x86 instruction that does this?
  - i.e., give me my current code address

How to find position of the code in memory at run time?

- Simple trick

```
    call L2
L2: popl %ebx
```

- Call next instruction
  - Saves EIP on the stack
  - EIP holds current position of the code
  - Use popl to fetch EIP into a register

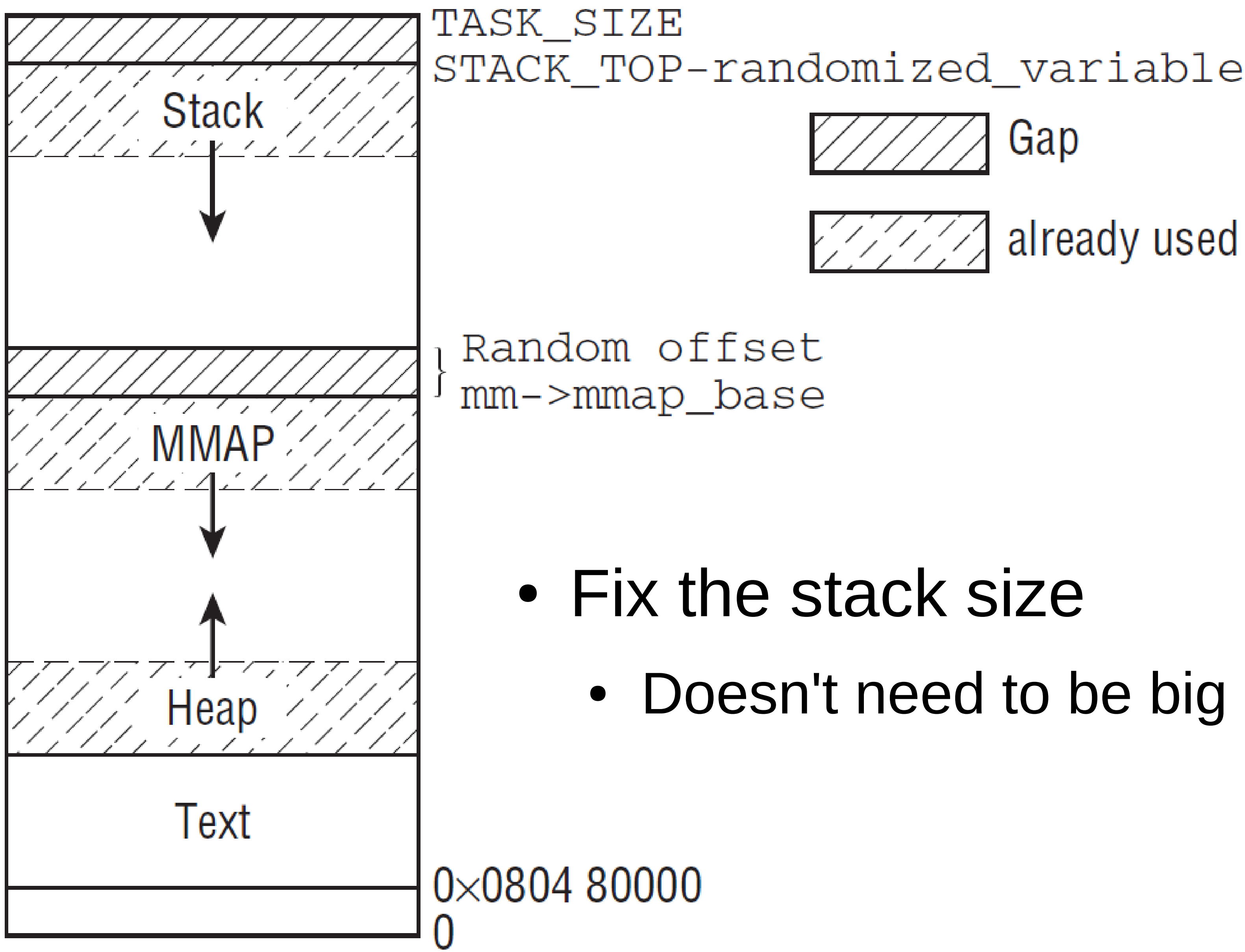

Load address unknown
at link time
XX0000
XX0010
call L2
L2: pop %bx
add $FF0,%bx
Fixed distance from
code to GOT
XX1000
Code
segment
GOT
Data
segment

PIC: Advantages and disadvantages

- Bad
  - Code gets slower
    - One register is wasted to keep GOT pointer
      - x86 has 6 registers, loosing one of them is bad
    - One more memory dereference
      - GOT can be large (lots of global variables)
      - Extra memory dereferences can have a high cost due to cache misses
    - One more call to find GOT
- Good
  - Share memory of common libraries
  - Address space randomization

Process virtual memory

Alternative address space layout



- Fix the stack size
  - Doesn't need to be big

Recap: known mappings

- Virtual to memory regions mapping
  - struct mm_struct (memory map)

Two kinds of memory regions

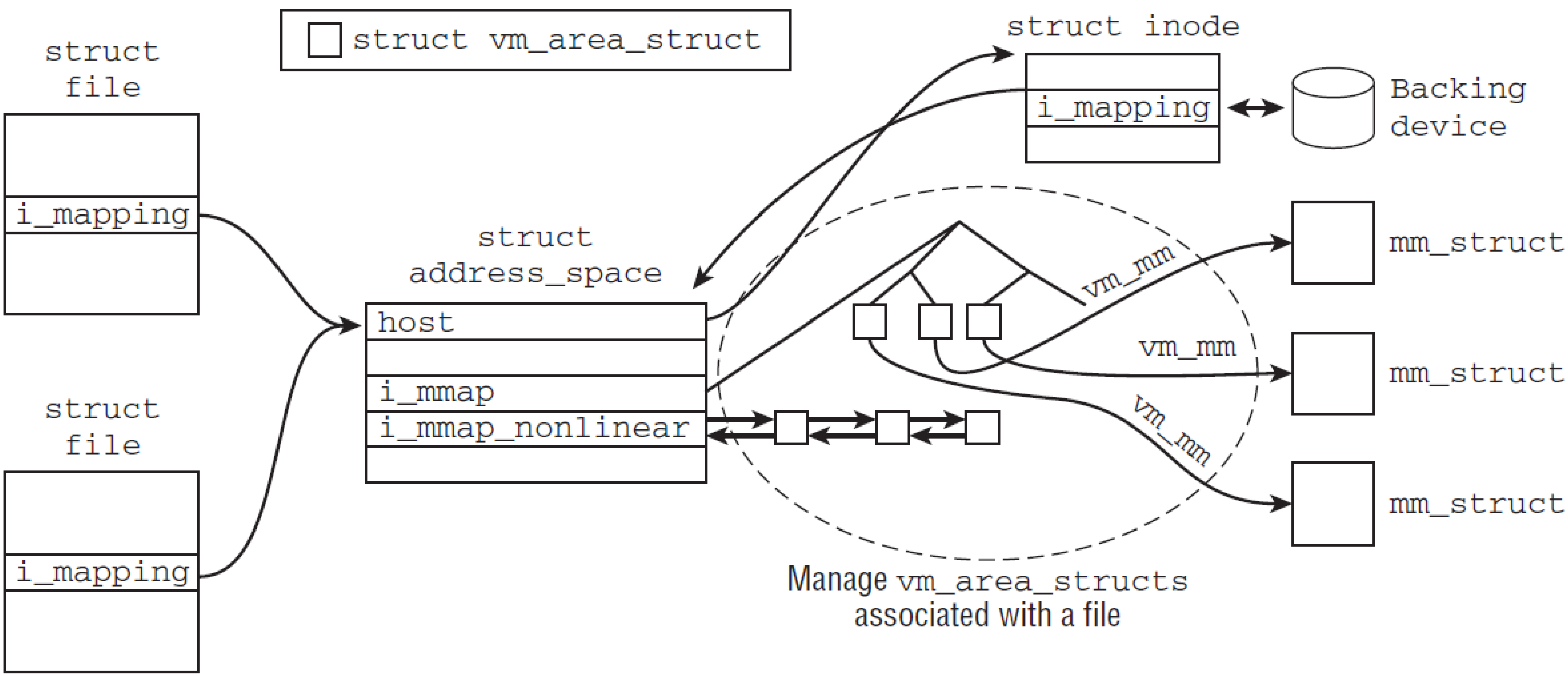
- Anonymous
  - Not backed or associated with any data source
    - Heap, BSS, stack
  - Often shared across multiple processes
    - E.g., after fork()
- Mapped
  - Backed by a file

Pagefault

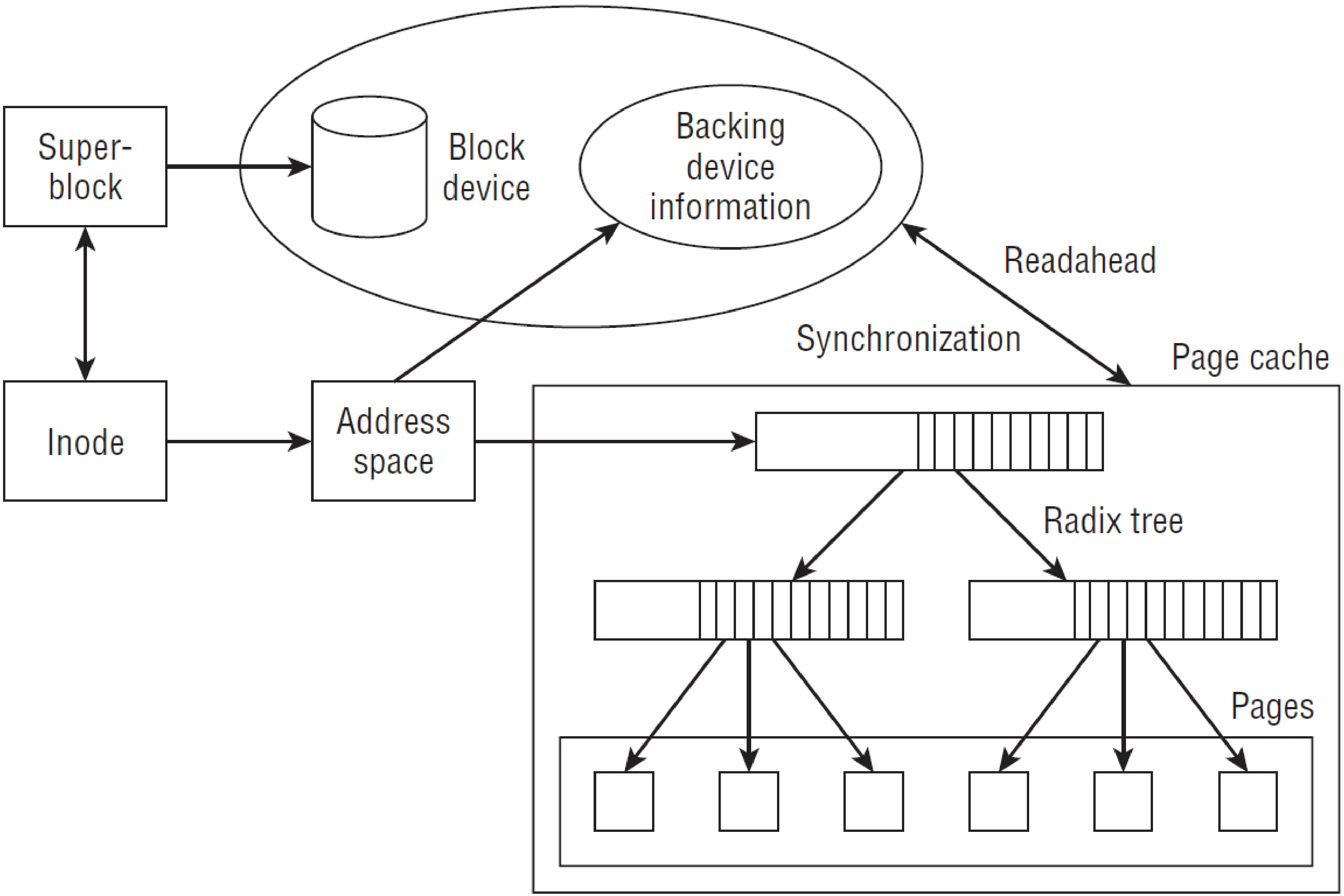
- For the current process
  - Represented with the `task_struct`
  - Walk the `mm->mmap_rb` to locate a `vm_area_struct` for the faulting virtual address

Pagefault (2)

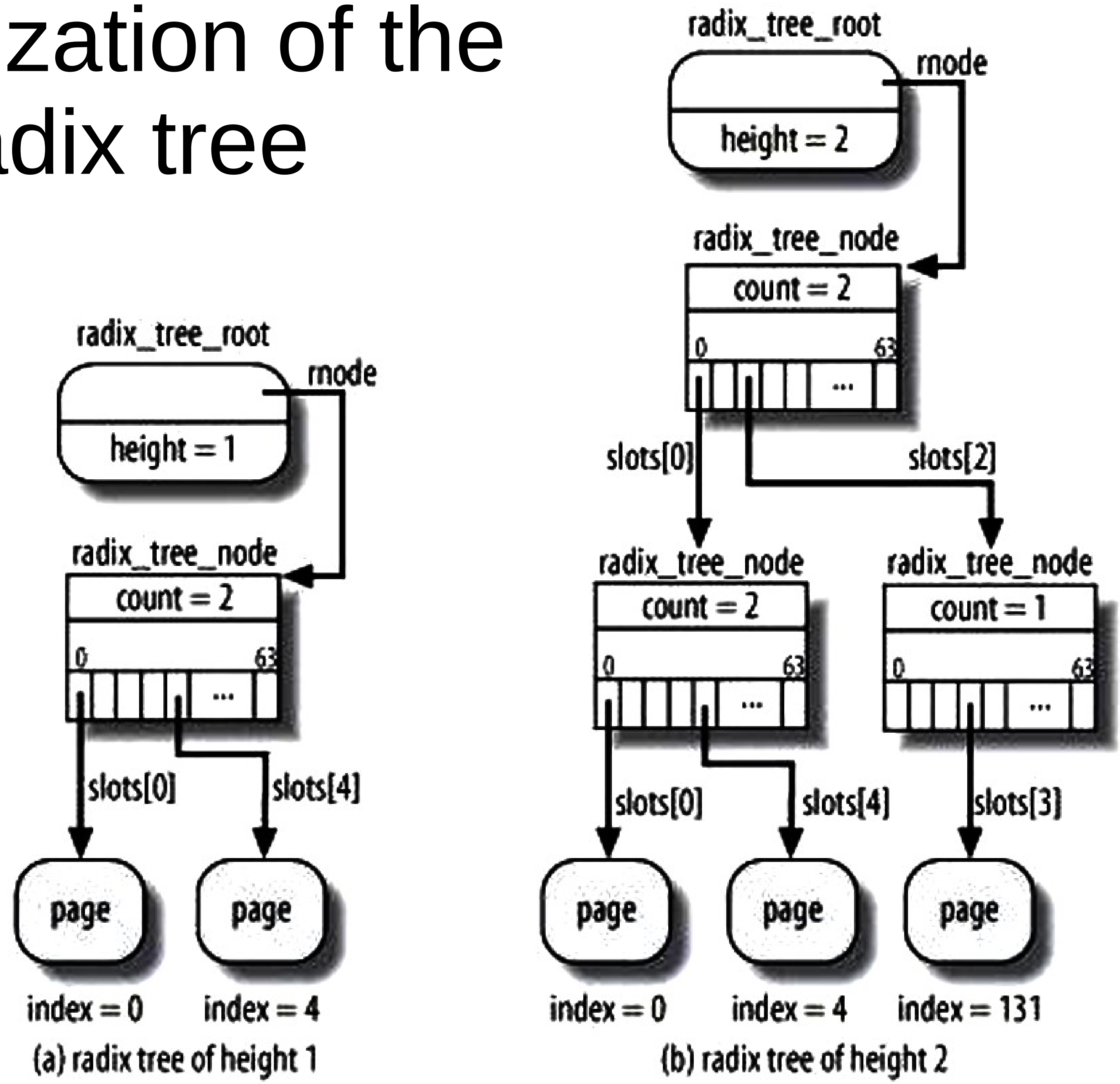
- Each `vm_area_struct` has a pointer to a `vm_file` backing this area



Page cache


Super-
block
Block
device
Backing
device
information
Readahead
Synchronization
Page cache
Inode
Address
space
Radix tree
Pages

Organization of the radix tree

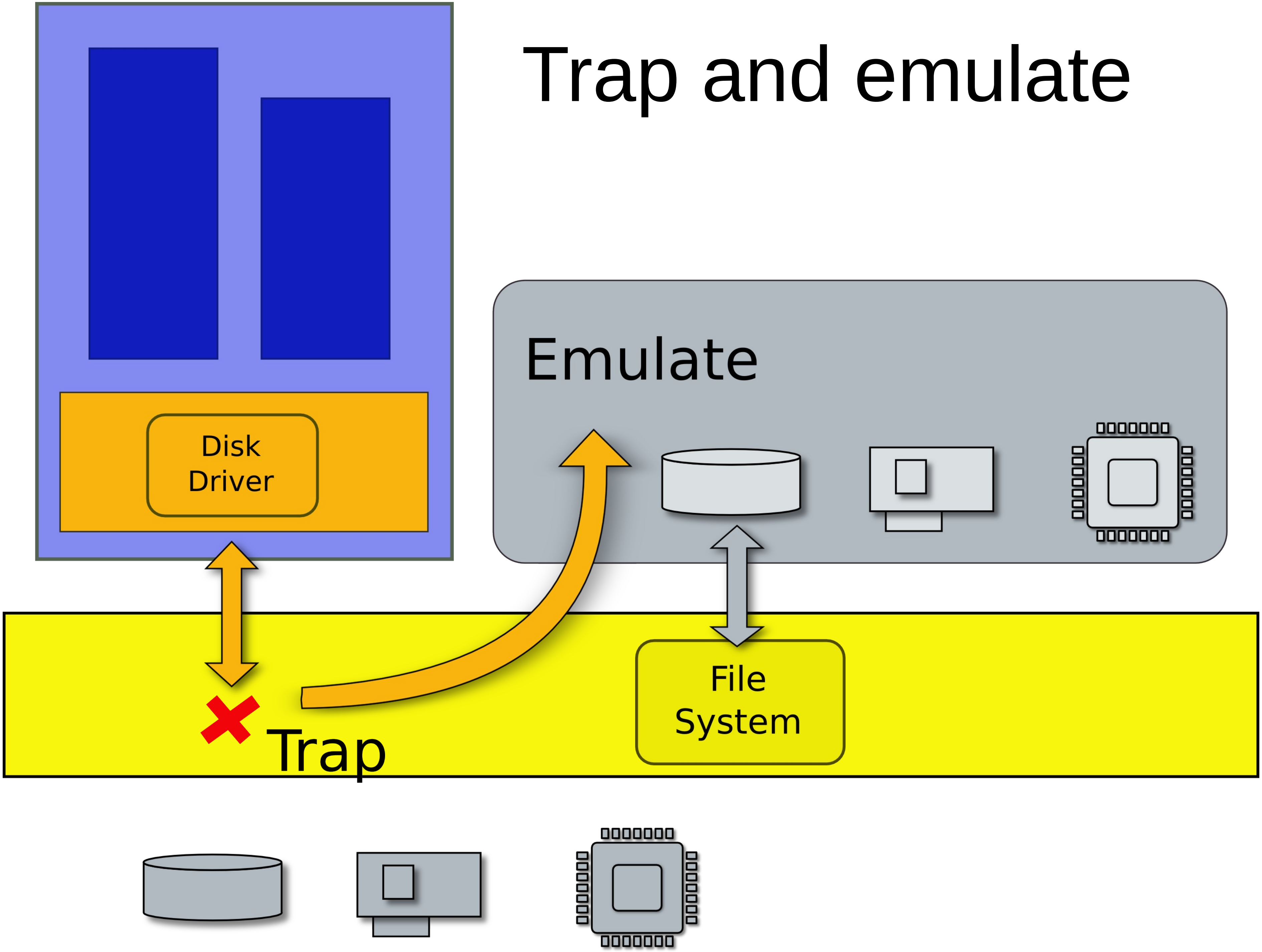


Variable height

- 1 – max index (64) – max file size 256 KB
- 2 – max index (4095) – max file size 16MB
- 3 – max index (262 143) – max file size 1GB
- 4 – … – max file size 64GB
- 5 – … – max file size 4TB
- 6 – … – max file size 16TB

Virtualization

Trap and emulate



x86 is not virtualizable

- Some instructions (*sensitive*) read or update the state of virtual machine and don't trap (*non-privileged*)
  - 17 sensitive, non-privileged instructions [Robin et al 2000]

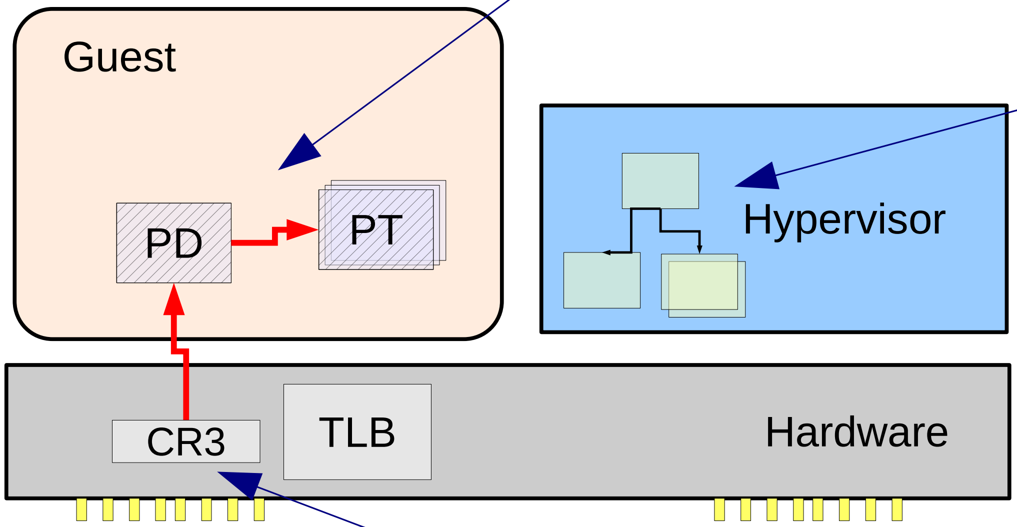
Solution space

- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
  - Binary translation (VMWare, QEMU)
- Change the operating system
  - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
  - Hardware supported virtualization (Xen, KVM, VMWare)
    - Intel VT-x, AMD SVM

Memory virtualization: brute force.

Write / read protected
page table area.

Every access results in VM-Exit and
passes control to hypervisor



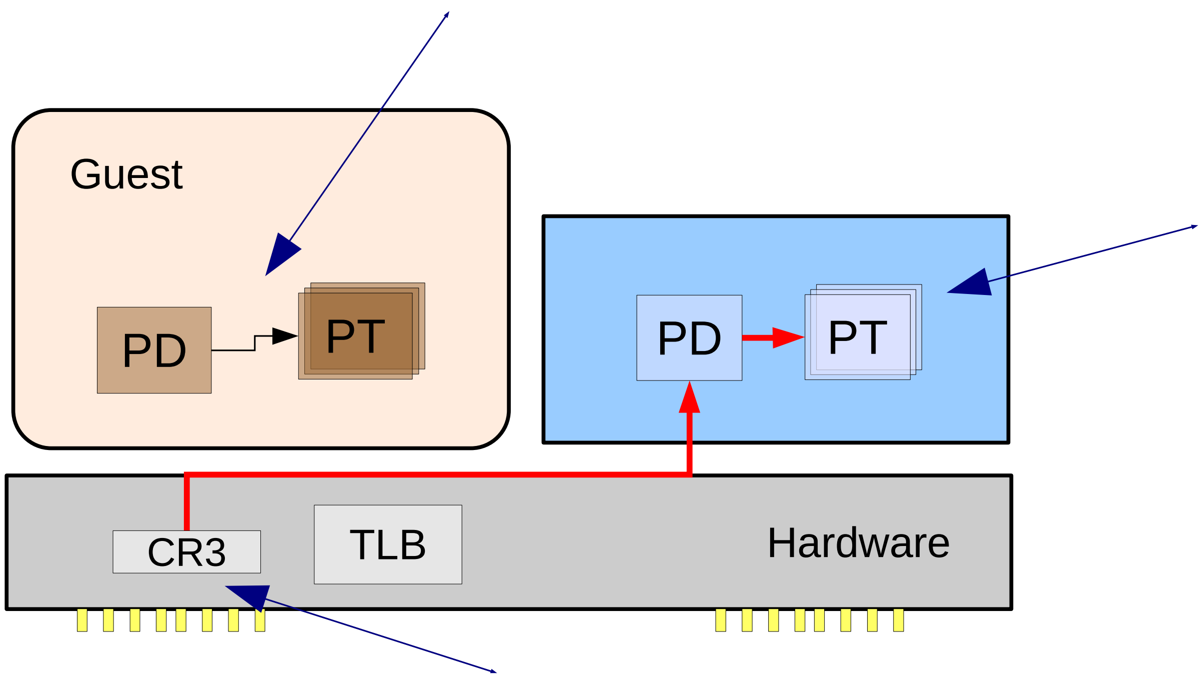
Helper structures describe
actual guest VM layout

Maintained for each guest. On
VM-Exit hypervisor adjusts guest
page accordingly.

CPU stores pointer on
guest page table directory

Memory virtualization: shadow page tables

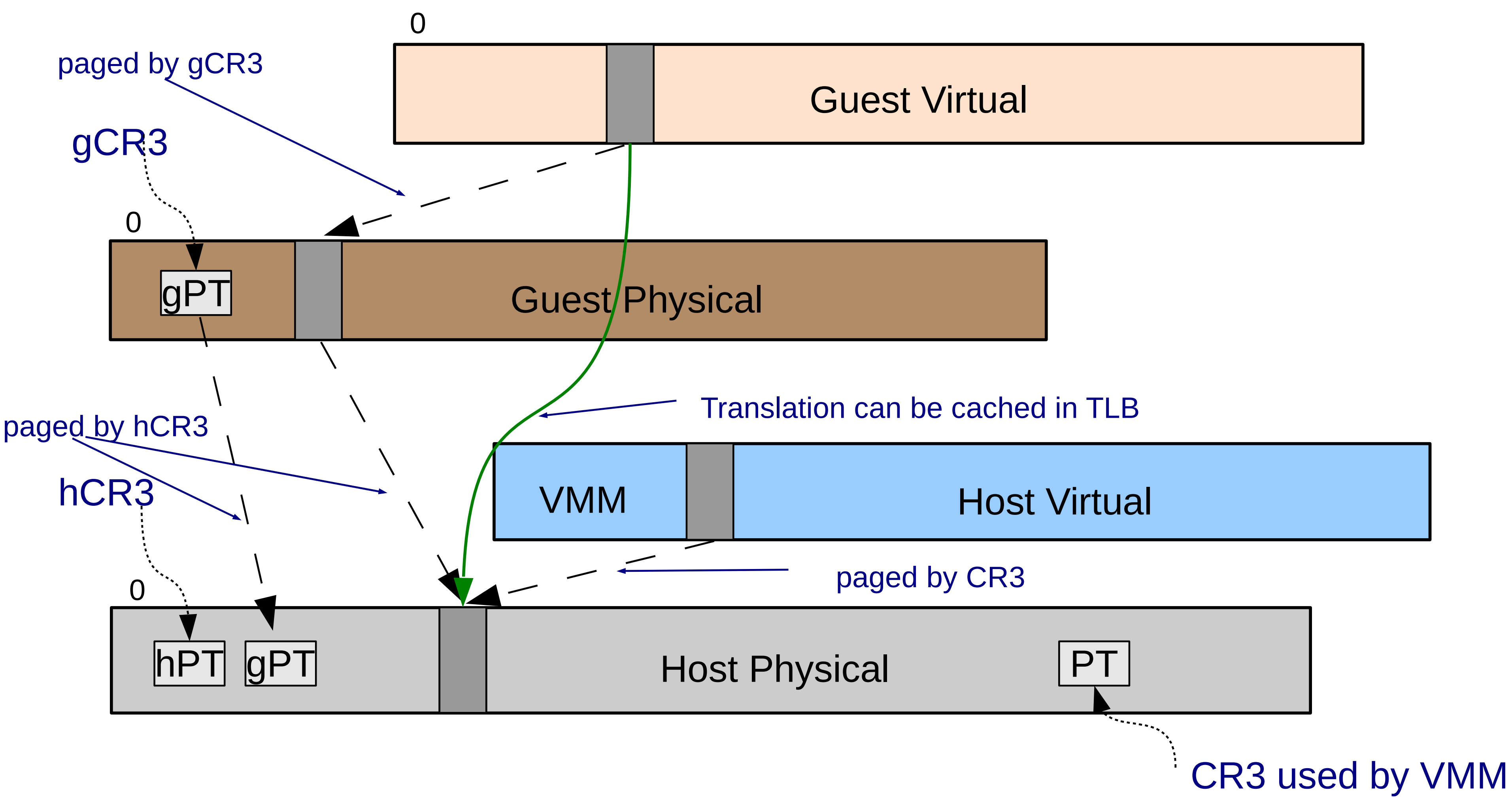
Guest page table hierarchy
It's writable, but can be inconsistent with active page table hierarchy stored by the hypervisor



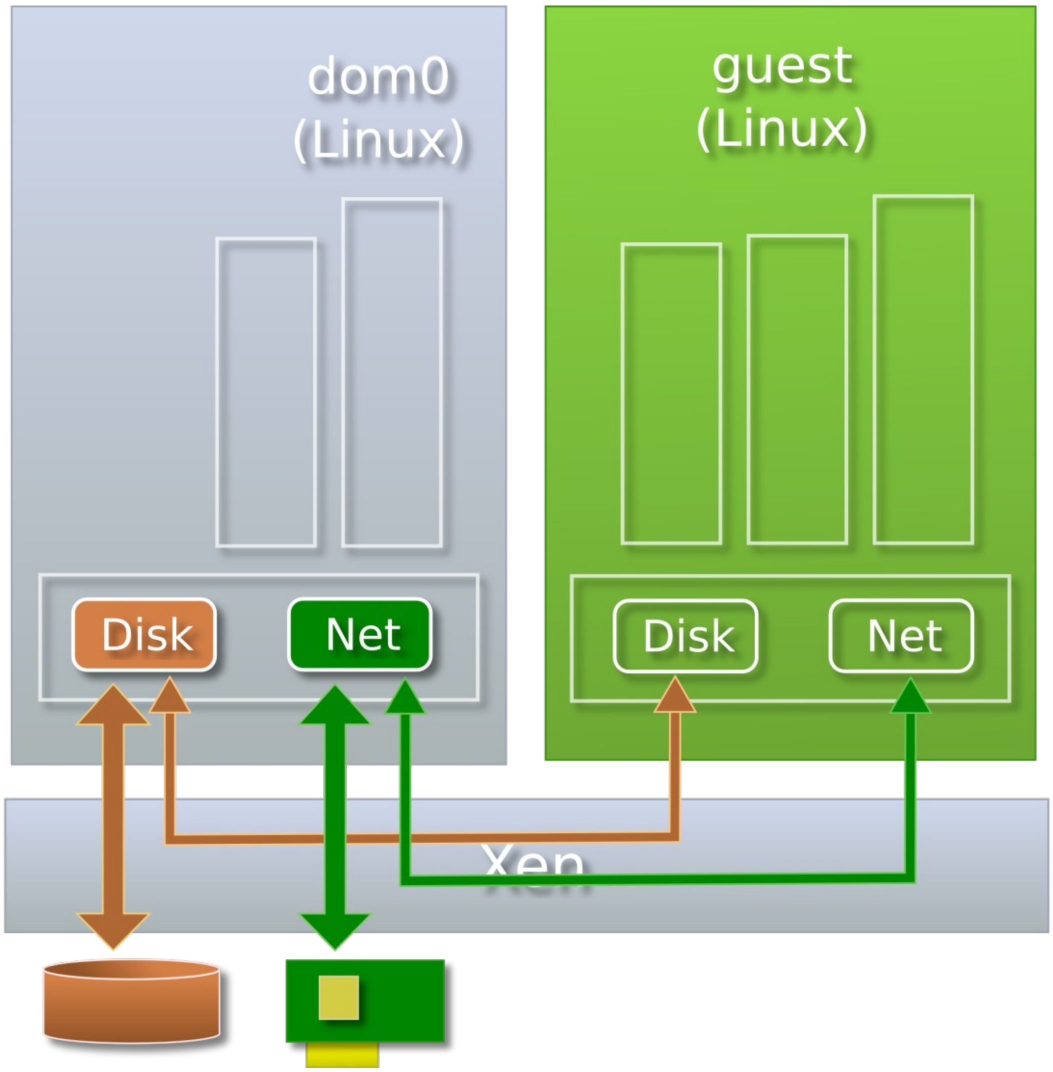
Active page table hierarchy
VMM maintains it for each VM that it supports

CPU stores pointer on active page table hierarchy.
On Intel CPUs TLB is always refilled from active page table directory

Nested page tables



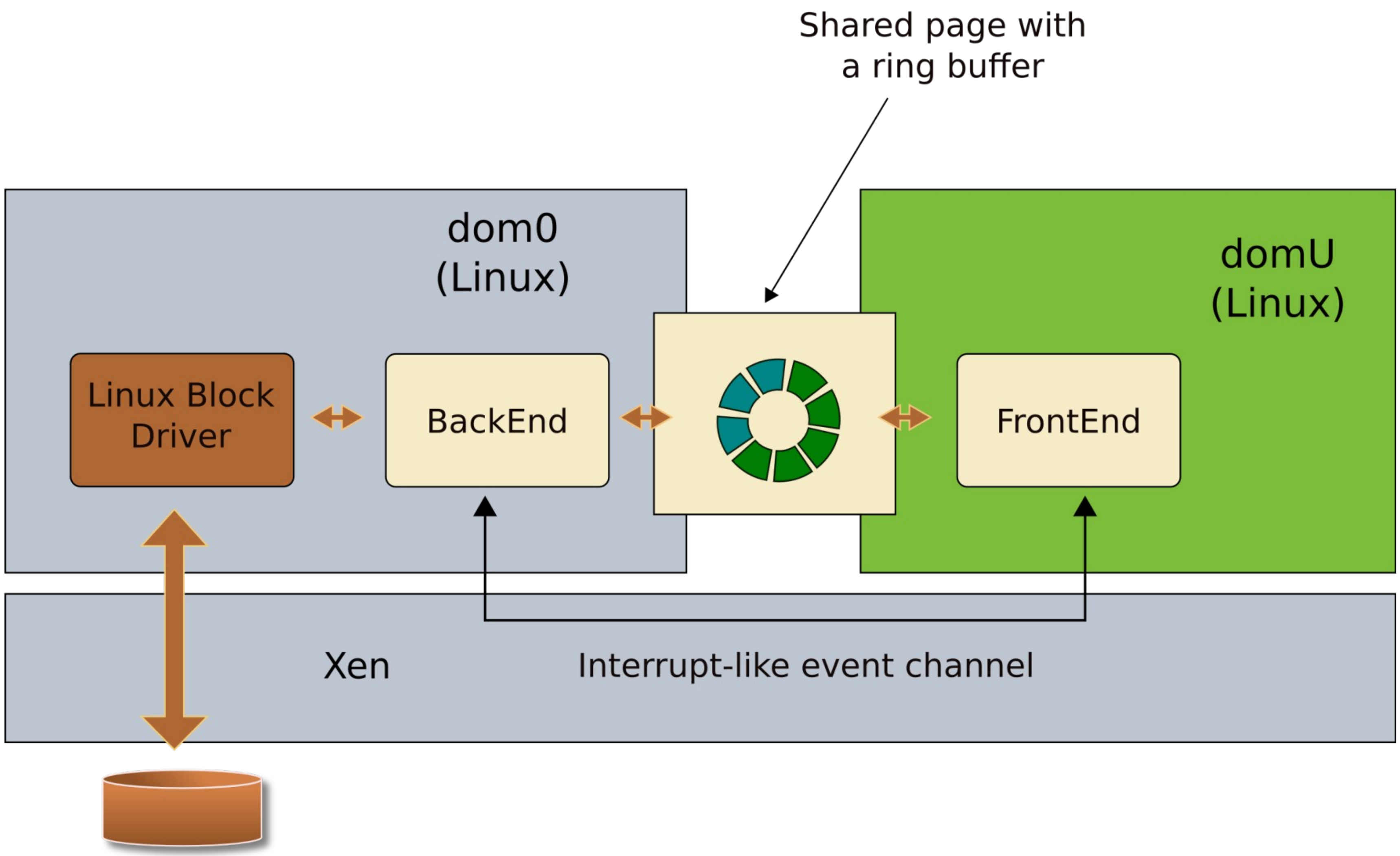
Virtual devices in Xen



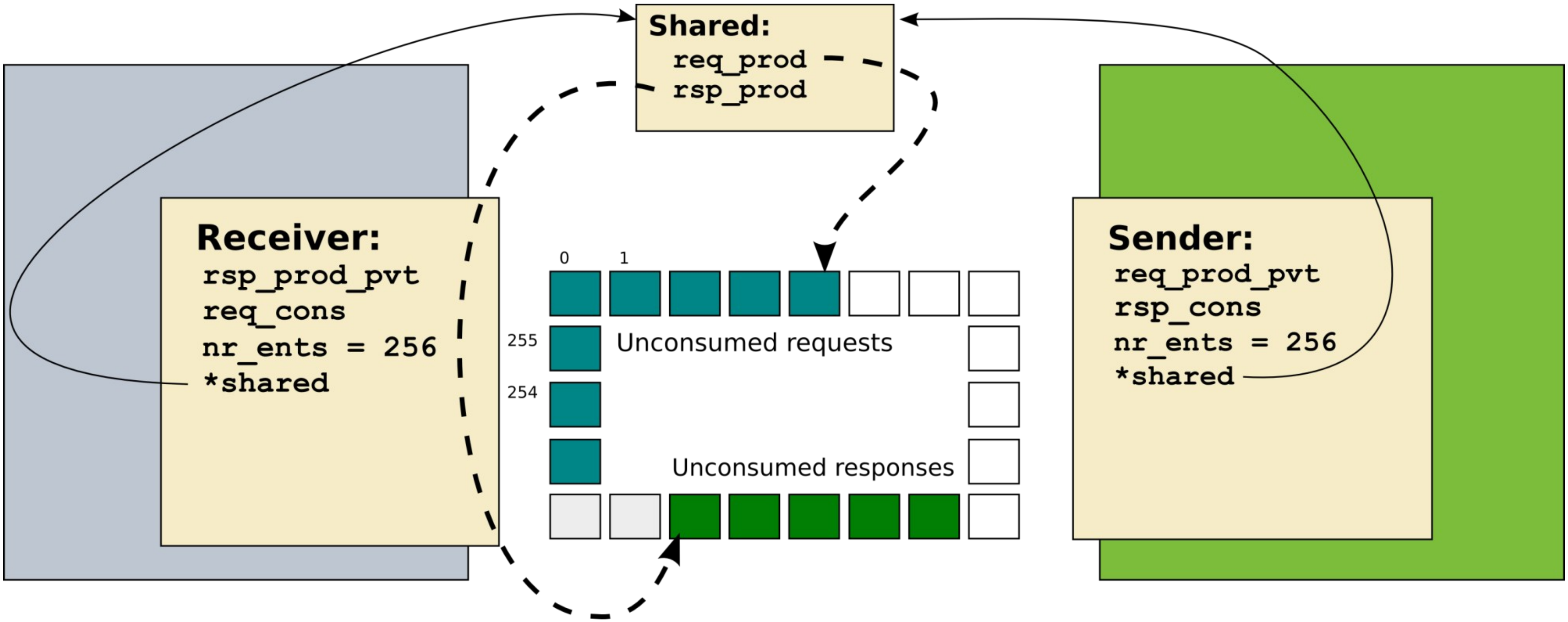
How to make the I/O fast?

- Take into account specifics of the device-driver communication
  - Bulk
    - Large packets (512B – 4K)
  - Session oriented
    - Connection is established once (during boot)
    - No short IPCs, like function calls
    - Costs of establishing an IPC channel are irrelevant
  - Throughput oriented
    - Devices have high delays anyway
  - Asynchronous
    - Again, no function calls, devices are already asynchronous

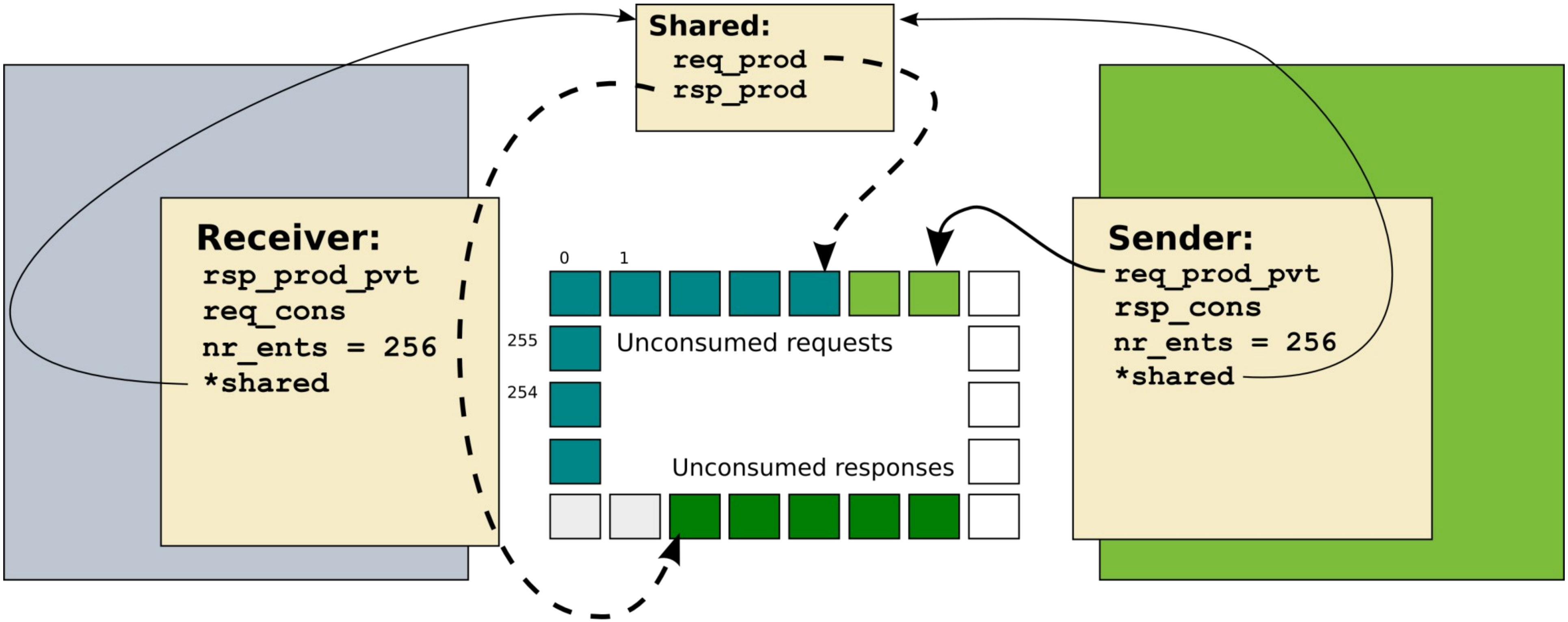
Shared rings and events



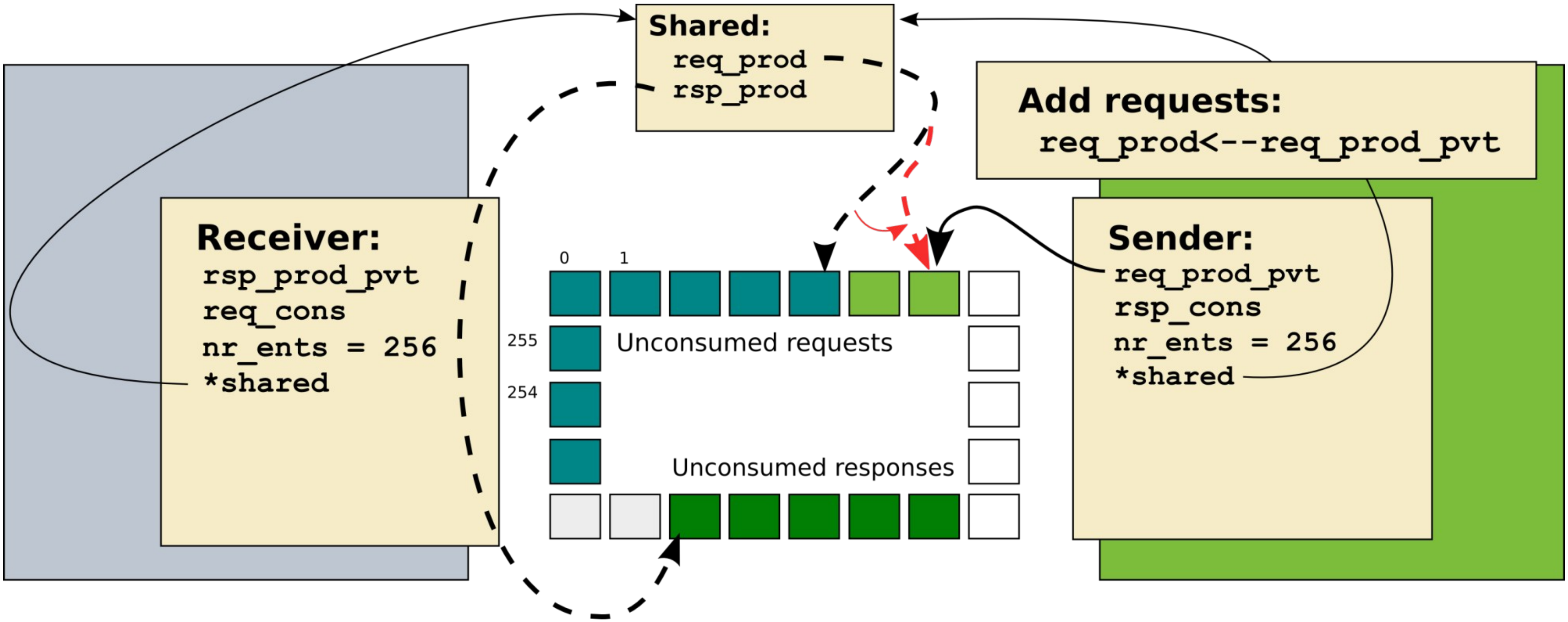
Shared rings



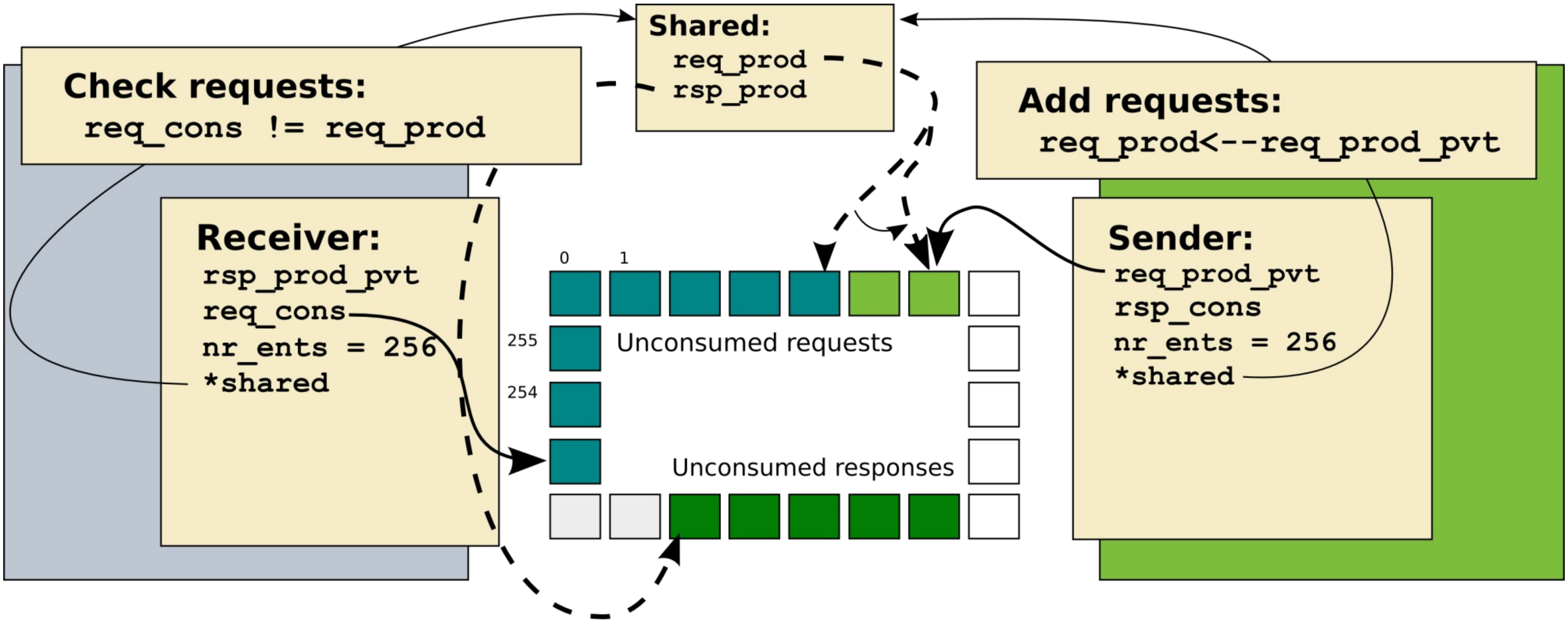
Shared rings



Shared rings



Shared rings



Thank you!