CS162: Operating Systems and Systems Programming

Lecture 13: Linux Memory Management / Distributed Systems (intro)

13 July 2015

Charles Reiss

https://cs162.eecs.berkeley.edu/

Recall: Demand Paging

Key idea: **Illusion of infinite memory**

Process's memory lives **on secondary storage**

Memory is **just a cache**

- Block = page
- Fully associative
- Cache replacement = page fault

Recall: Demand Paging Policies

Ideal replacement policy: Belady's MIN

- Impossible to implement, but ideal (access time)
- Similar problems with SRTF

Possible replacement policy: Least Recently Used

- Still impossible to implement – too much overhead to track "uses"

Practical replacement policy: Clock algorithm

- "Not recently used"
- Use HW support of used bit, scan periodically

Practical replacement policy: Second chance list

- "Not recently used"
- Make inactive pages invalid to see if they're really unused

Recall: Reverse Page Mapping ("coremap")

Mapping from physical page frame # to **all its locations in memory**

Example uses:

- Scanning accessed/dirty bits of **all its PTEs**
- Marking page not present in **all its PTEs** when replacing with another

Linux implementation:

- Linked list of *memory regions* derived from an object (e.g. file)

Reality: Linux Memory Management

Three zones:

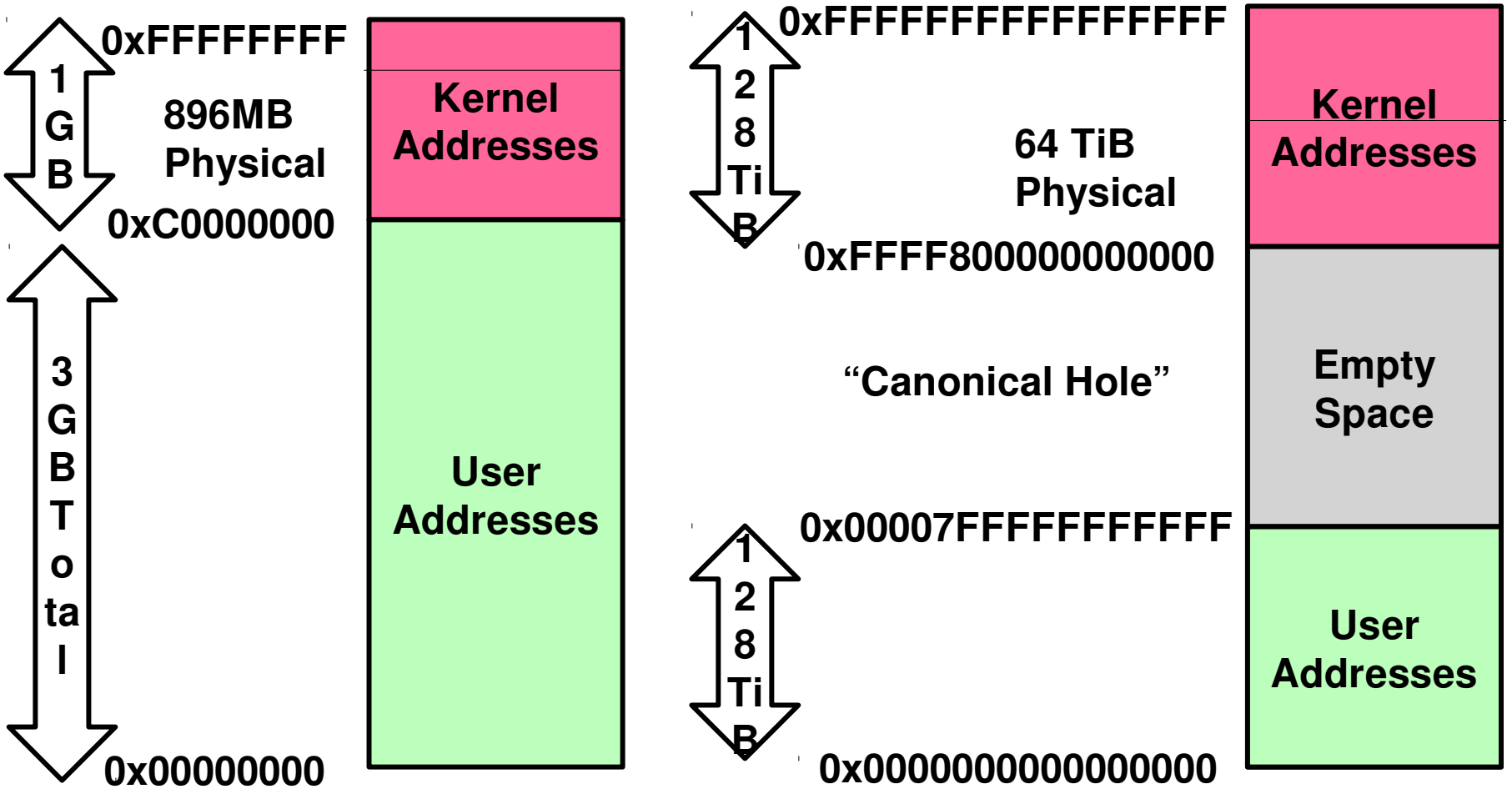
- ZONE_DMA: phys addr <16M, DMAable with ISA bus
- ZONE_NORMAL: 16M $\rightarrow$ 896MB
- ZONE_HIGHMEM: Everything else

Why?

- Hardware limitations
- On 32-bit, 0-896MB is always mapped in kernel space

Each zone has own freelist, LRU lists (active, inactive)

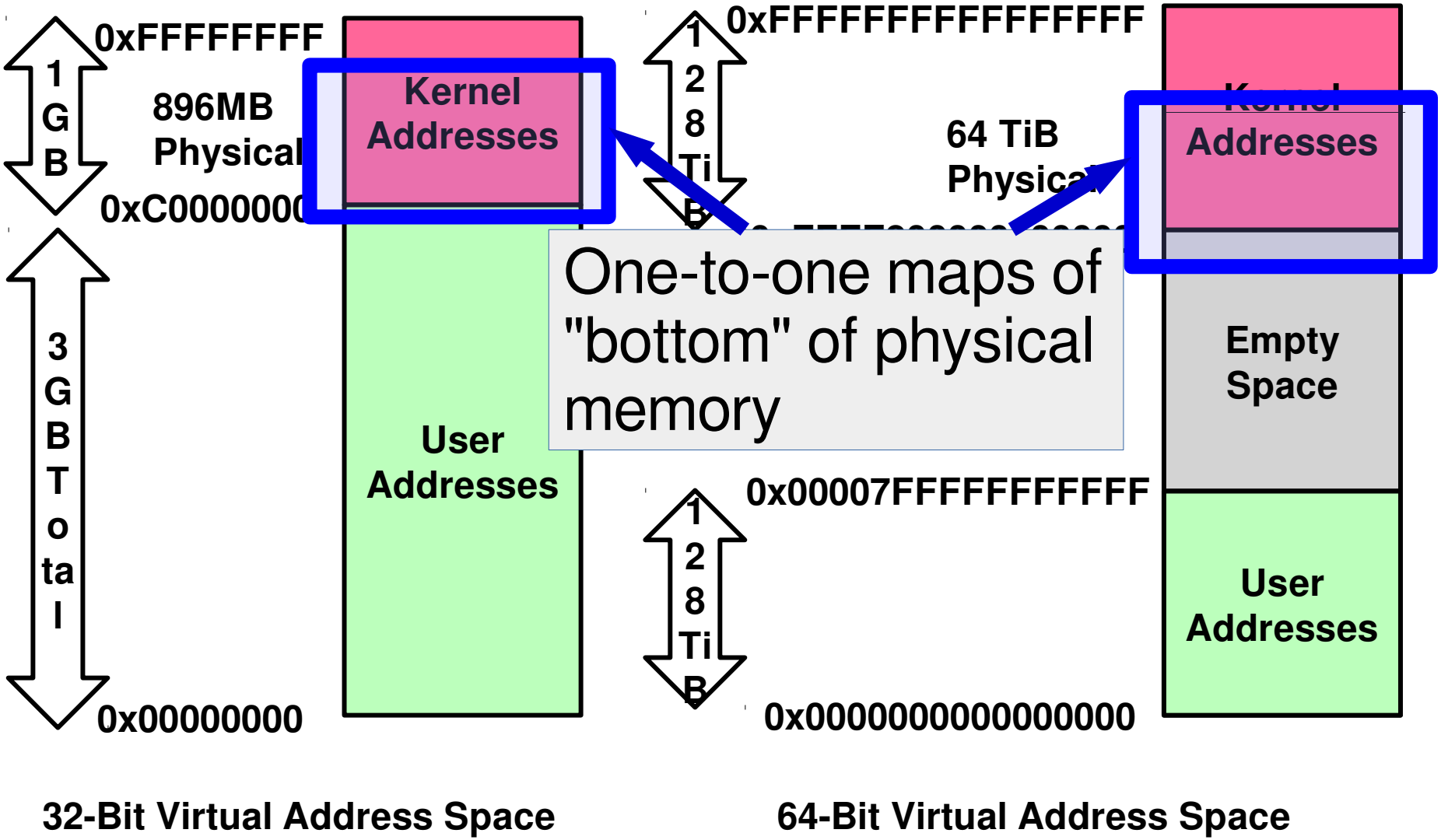
Linux Virtual Memory Map



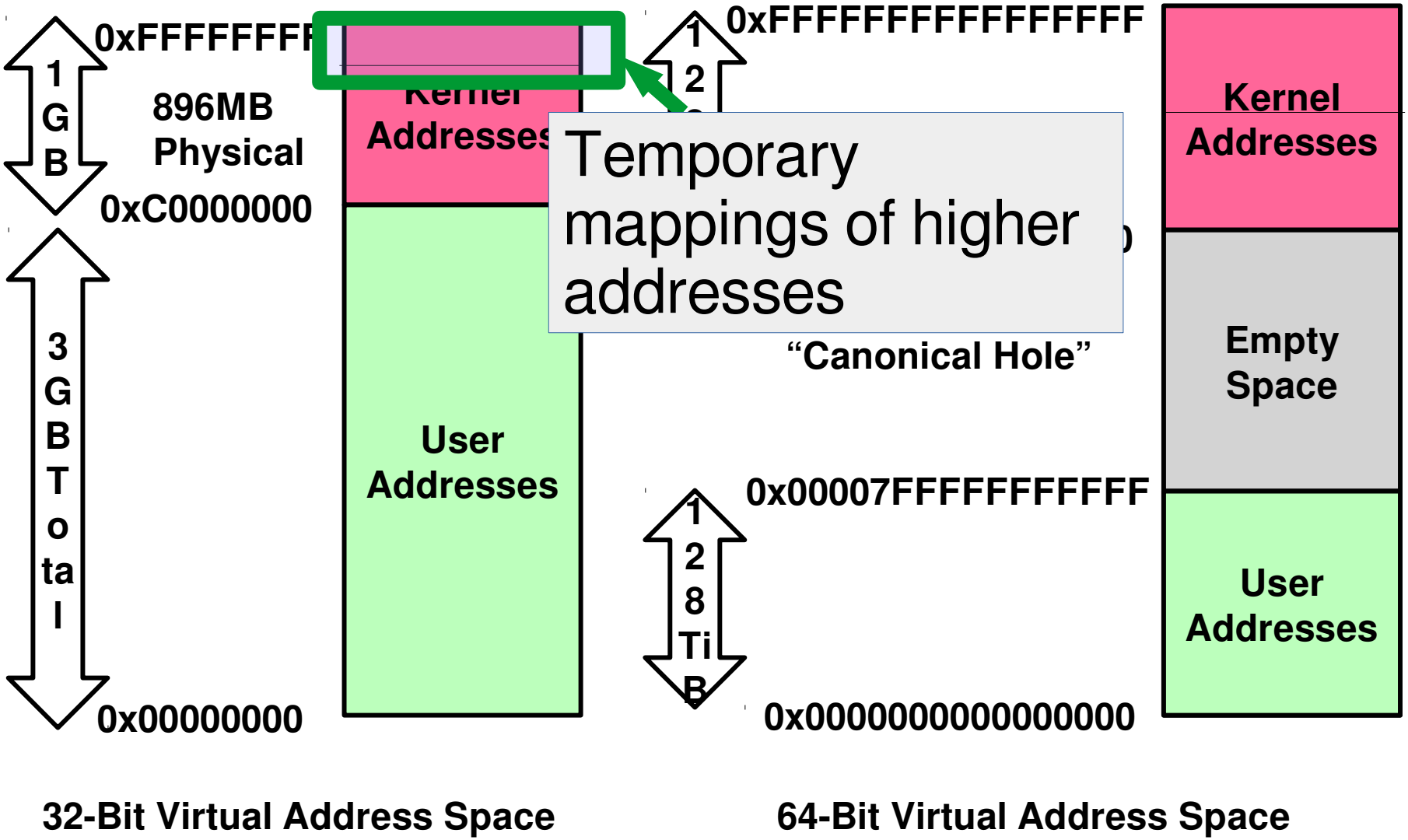
32-Bit Virtual Address Space

64-Bit Virtual Address Space

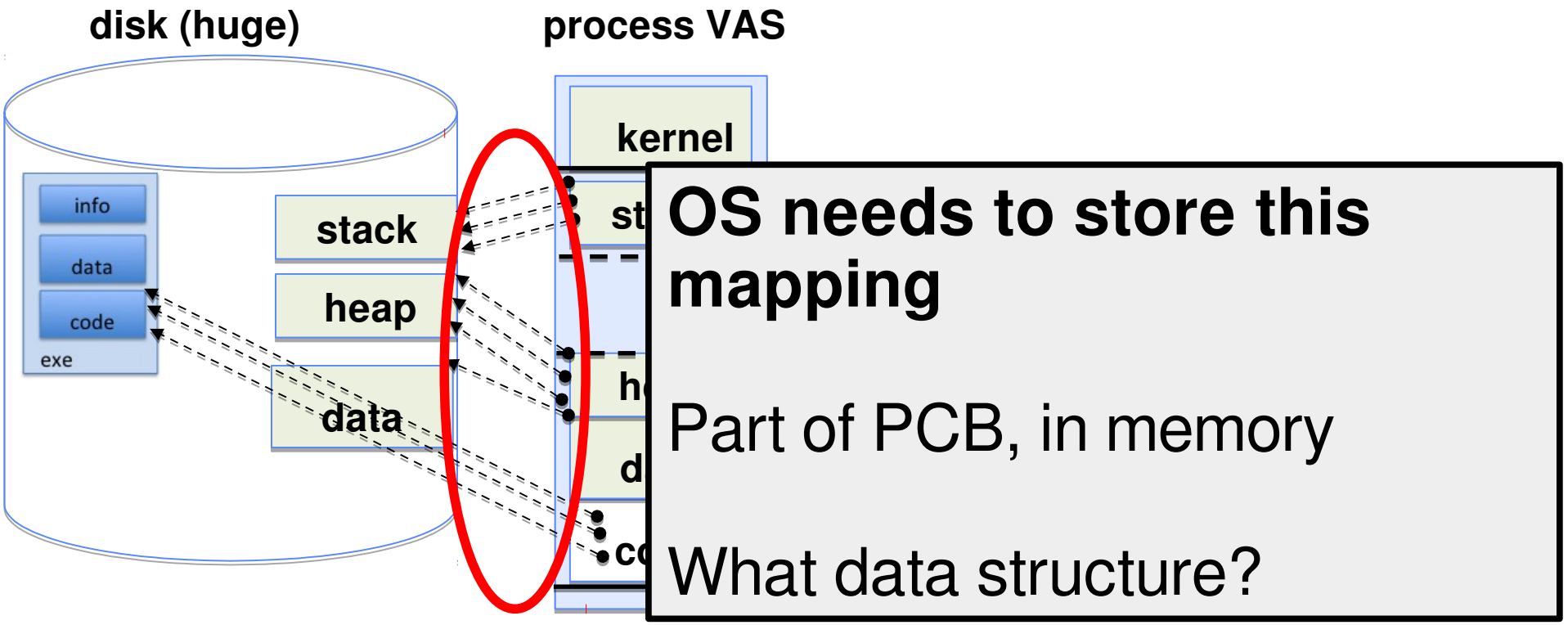
Linux Virtual Memory Map



Linux Virtual Memory Map



Recall: Create Address Space

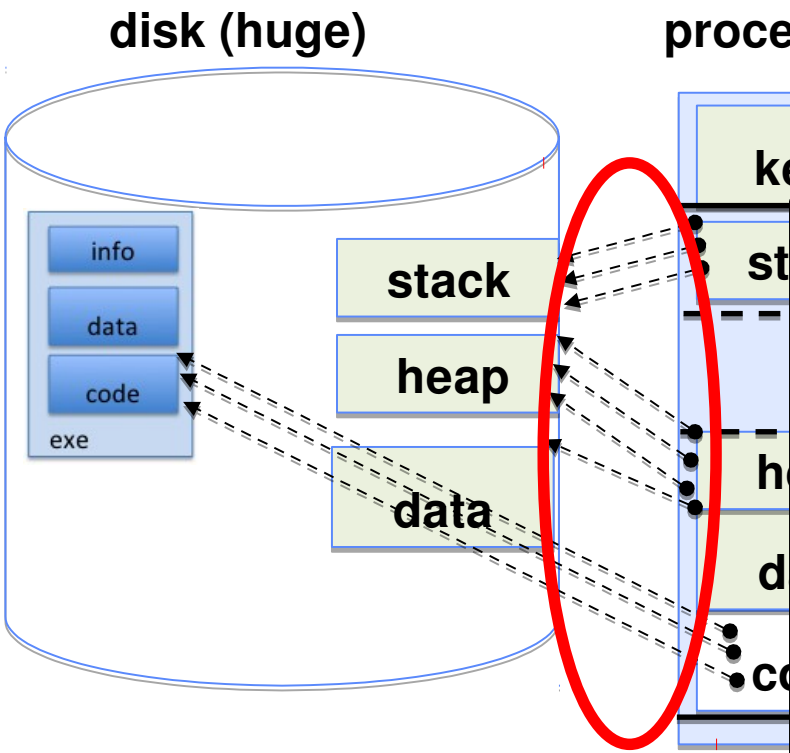


One method: Everything **backed by disk**

Just allocate space on disk

- Let page faults trigger it actually being read from disk

Linux Address Space



Linux: List of "memory regions" specifying backing object (file or anonymous memory region) and range from each

Linux Address Space Region

task_struct (PCB/TCB)

```
struct vm_area_struct {
  // start and end virtual address
  unsigned long vm_start;
  unsigned long vm_end;
  // …
  // permissions and policy
  pgprot_t vm_page_prot;
  unsigned long vm_flags; // MAP_PRIVATE == copy-on-write
  // …
  // backing store
  // how many pages into the backing store
  unsigned long vm_pgoff;
  // mapped file (or NULL)
  struct file * vm_file;
  // …
};
```

Page/Protection Fault Handler

Lookup struct vm_area_struct*.

Look at type of access versus area protection flags

Main cases:

- Allocate new page (copy from file or zero page)
- Copy-on-write page (allocate + copy)
- Swap in non-file backed page (trick: *store location on disk in unused bits of PTE*)

Recall: TLB and page table changes

Consider **copy-on-write**.

Process A calls fork().

OS marks its pages as read-only. **Tells MMU to clear those TLB entries.**

After returning from fork it tries to write to its stack.

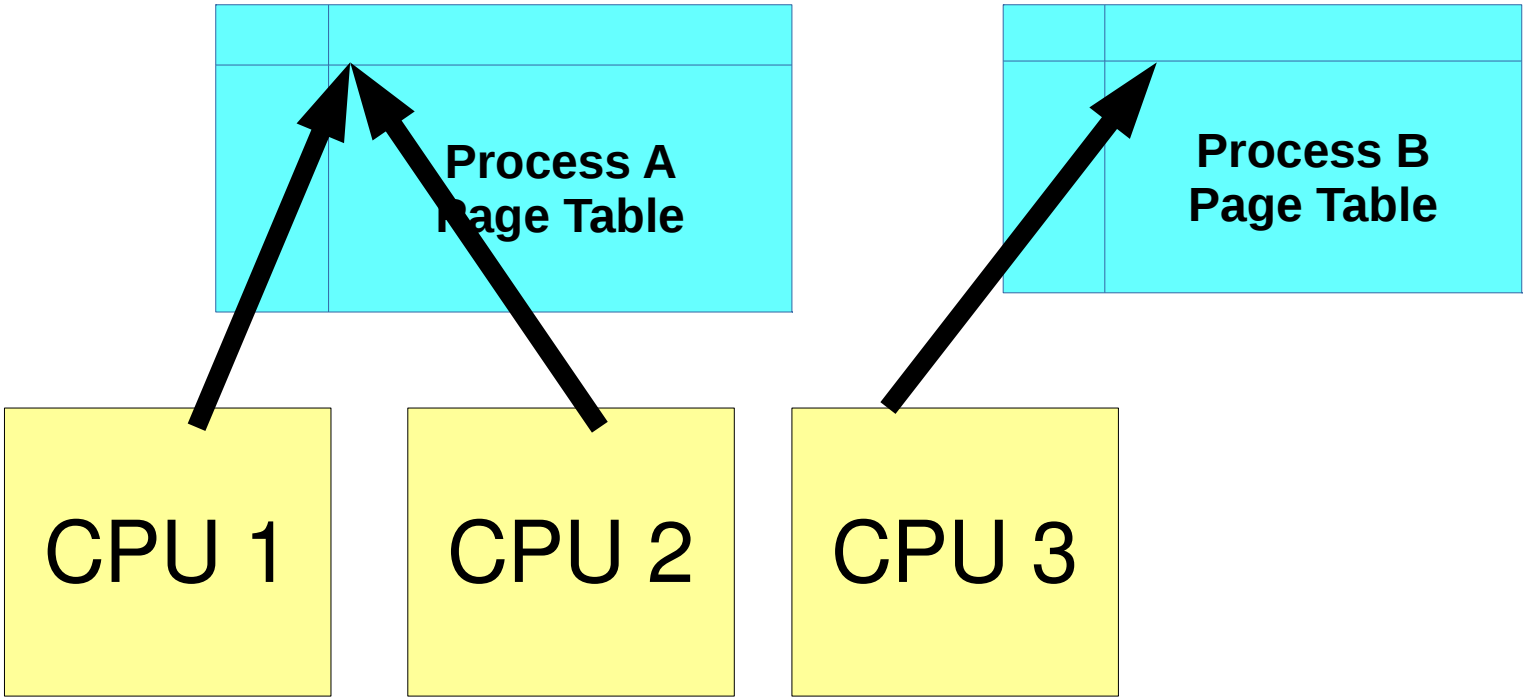
Triggers protection fault. OS makes a copy of the stack page, updates page table. **Tells MMU to clear that TLB entry.**

Restarts instruction.

What about multiple processors? (1)

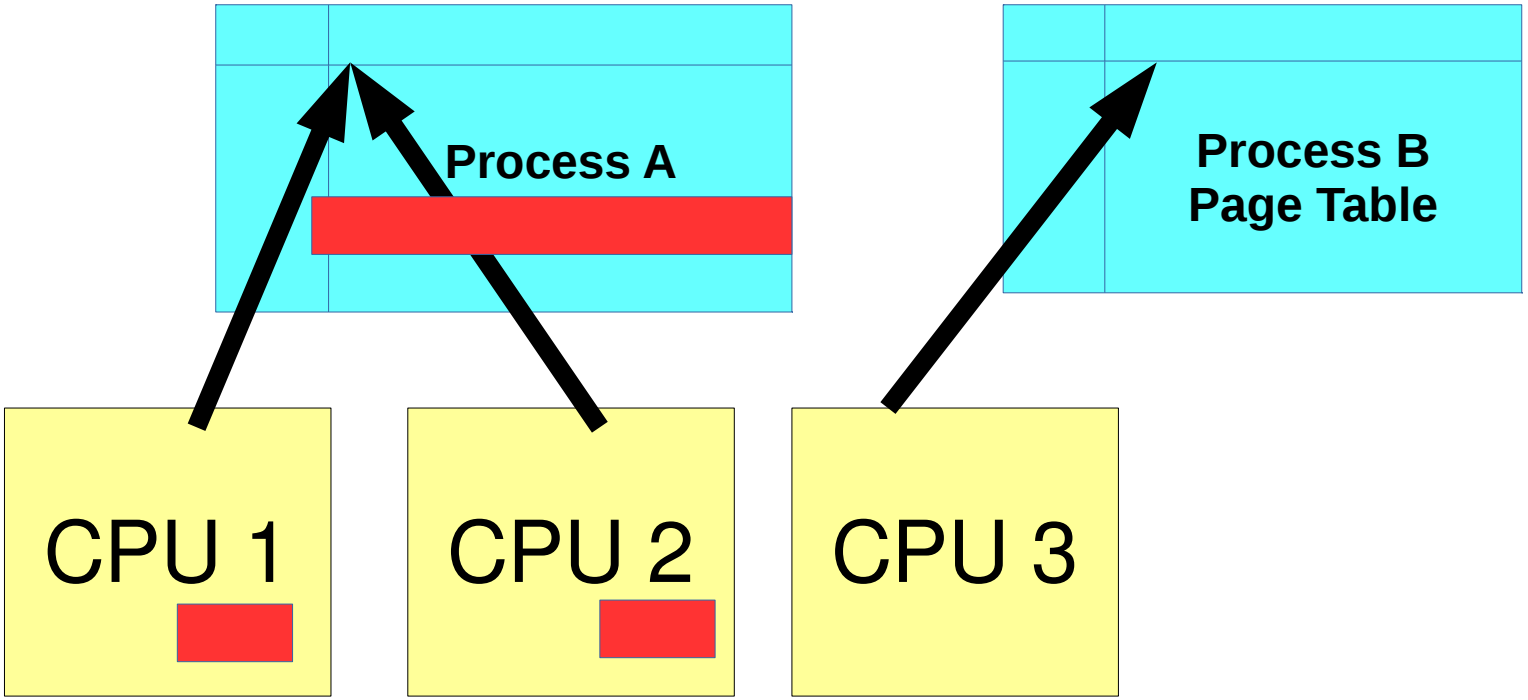
Process A running on CPU 1 + 2 (different threads)

Process B on CPU 3



What about multiple processors? (2)

Page fault on CPU 1 needs to invalidate TLB on CPU 1 *and* 2

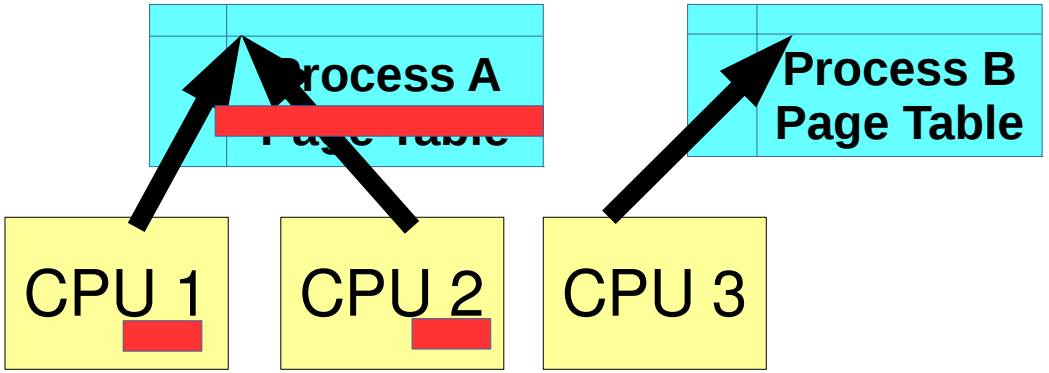


What about multiple processors? (3)

"TLB shootdown"

Interprocessor Interrupt

- Trigger an interrupt on another processor
- All of them – need synchronization to make sure another thread of process A doesn't start on CPU 3
- invlpg instruction or reset page table pointer (untagged TLB)



Linux Virtual Map Management

Kernel memory not visible to user

- Exception: shared memory for fast "system calls" (vDSO)

Forward mapping:

- PCB has list of vm_area_structs
- Also binary search tree (fast lookups by virtual address)

`struct page` describes physical page

- pointers to all its memory regions (**coremap**!)
- membership on all its "LRU" lists

Linux Page Allocation

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned order)
```

- gfp_mask specifies valid zones (among other options)
- Allocate 2^{order} bytes from **contiguous physical pages**

```
struct page * alloc_page(gfp_t gfp_mask)
```

- Allocate one page

Allocator uses "buddy" system – fast, avoids fragmentation (somewhat)

Why would you need more than one contiguous page?

- Some platforms have bigger page tables
- IO with some devices (later)

Page Frame Reclaiming Algorithm

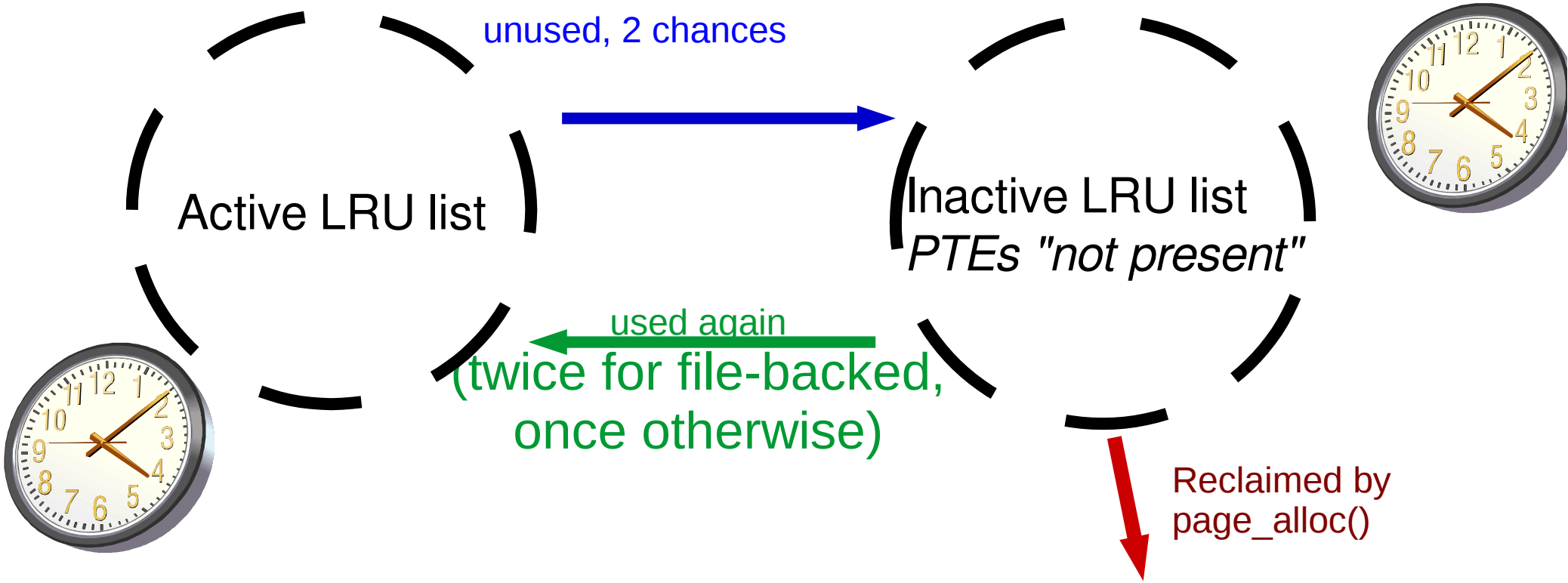
Implementation of "freelist" filling

Several triggers:

- Low on memory
- Hibernation (suspend-to-disk)
- **Periodic – background kernel thread**

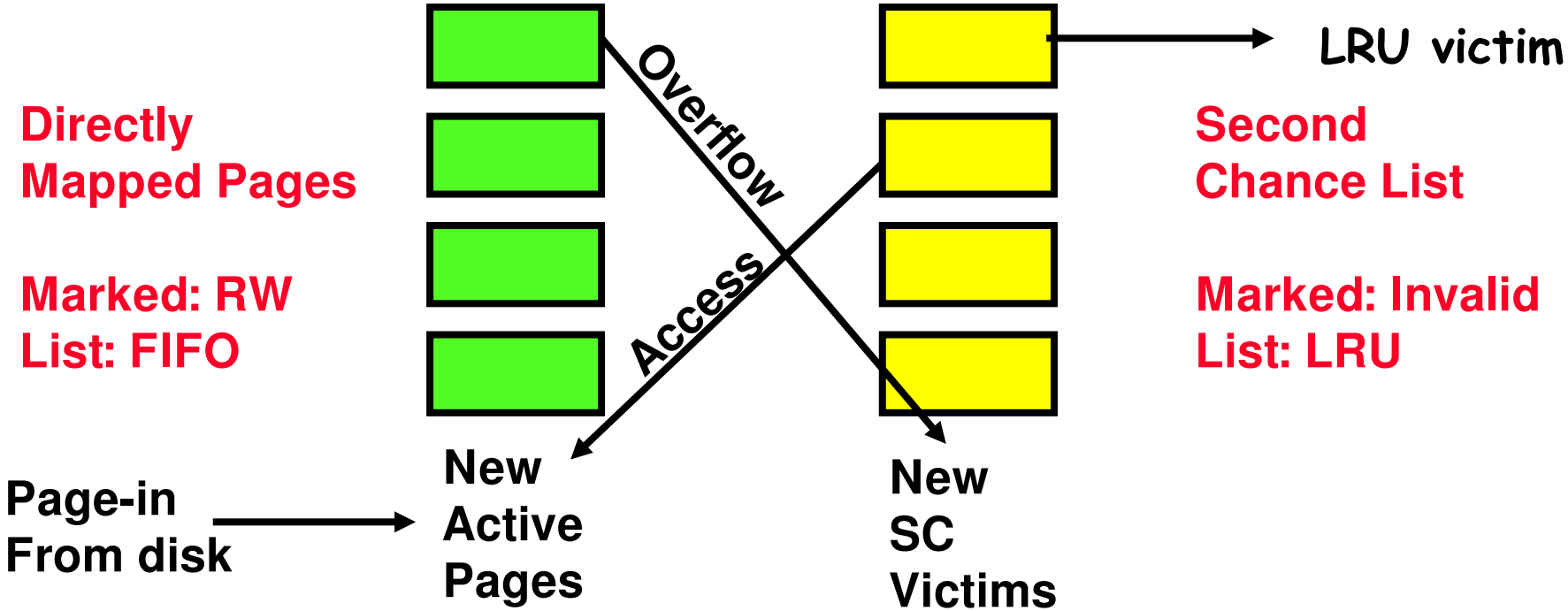
Linux LRU approximation

Each zone keeps two "LRU" lists: Active, Inactive



More aggressive behavior when tight on memory

Recall: Second-Chance List



Split memory in two: Active list (RW), SC list (**Invalid**)

Page Fault to access Second Chance List

- Move to front of access list on page fault

Victims from "front" of second chance list

Linux SLAB Allocator

Observation: object initialization time can be greater than allocation/deallocation time

Goal: **allocate once**, **reuse objects**

- Constructor run when first allocated only
- Not on subsequent free/allocation
- Handle actual page allocations (often >1 page at a time)

Freelist per object type (fast!)

(Objects of one type together → better locality??)

SLAB allocator: Cache use

```
task_struct_cachep =
   kmem_cache_create("task_struct",
             sizeof(struct task_struct),
        ARCH_MIN_TASKALIGN,
        SLAB_PANIC | SLAB_NOTRACK,
        NULL);
...
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
   return NULL;
...
kmem_free(task_struct_cachep,tsk);
```

SLAB allocator Details (1)

SLAB allocator: objects of one type together on a page

- Example type: open file
- Better locality?

Cache coloring:

- Offset the beginning of each page to avoid conflict misses
- Idea: index bits of more popular fields of different copies of the same kind of object are less likely to be the same

SLAB allocator Details (2)

Generic `kmalloc()/kfree()` implemented with "caches" for powers of two

Support **reclamation** of memory:

- Scan **global list of all caches** for pages with no allocated objects to free

Recall: Other Examples of Caching

Demand paging (later)

- Leave unused parts of process memory on disk/SSD

File systems

- Contents of files
- Directory structure

Networking

- Hostname to IP address translations
- Web pages

Recall: Other Examples of Caching

Demand paging

- Leave unused parts

File systems

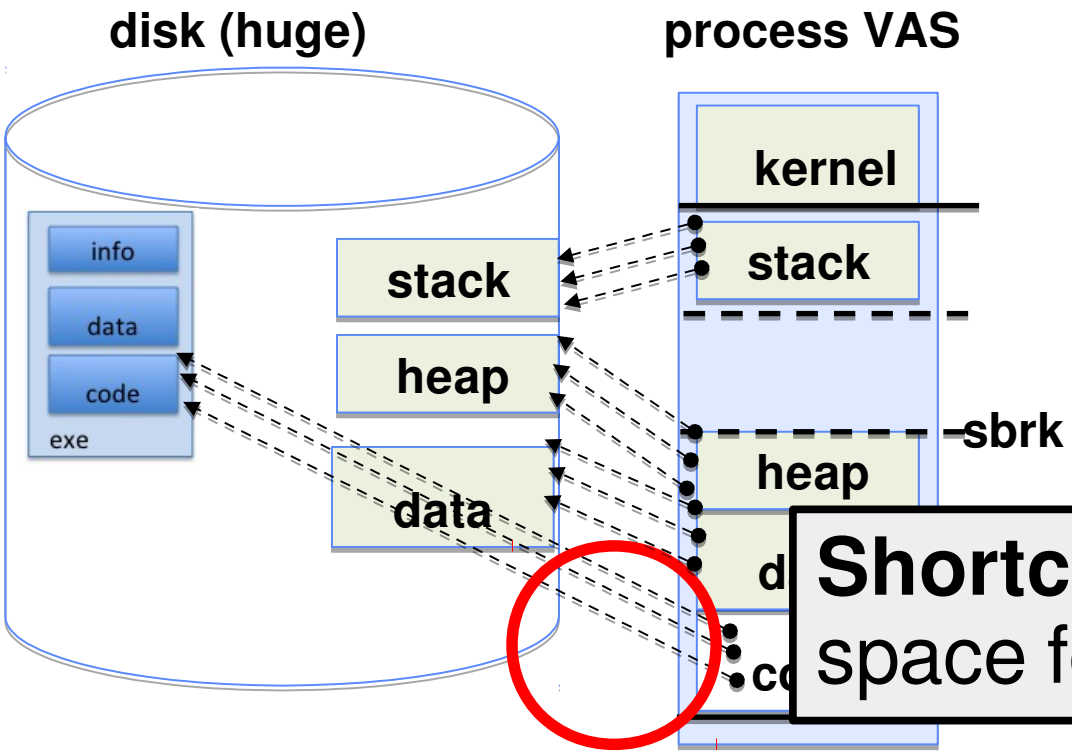
- **Contents of files**
- Directory structure

Networking

- Hostname to IP address translations
- Web pages

On Linux, these are the same cache!

Recall: Create Address Space



Shortcut: Don't allocate extra space for files already on disk

One method: Everything **backed by disk**

Just allocate space on disk

- Let page faults trigger it being read from disk

How does read()/write() work?

(For **regular files** – terminal, network, devices, ... are different)

Load the current page of the file as one of the pages to handle with demand paging

- If it's already there, **cache hit!**

Copy to or from that page

Stays in memory until clock algorithm/etc. removes it

Writing Dirty Pages

Since this is used for write()...

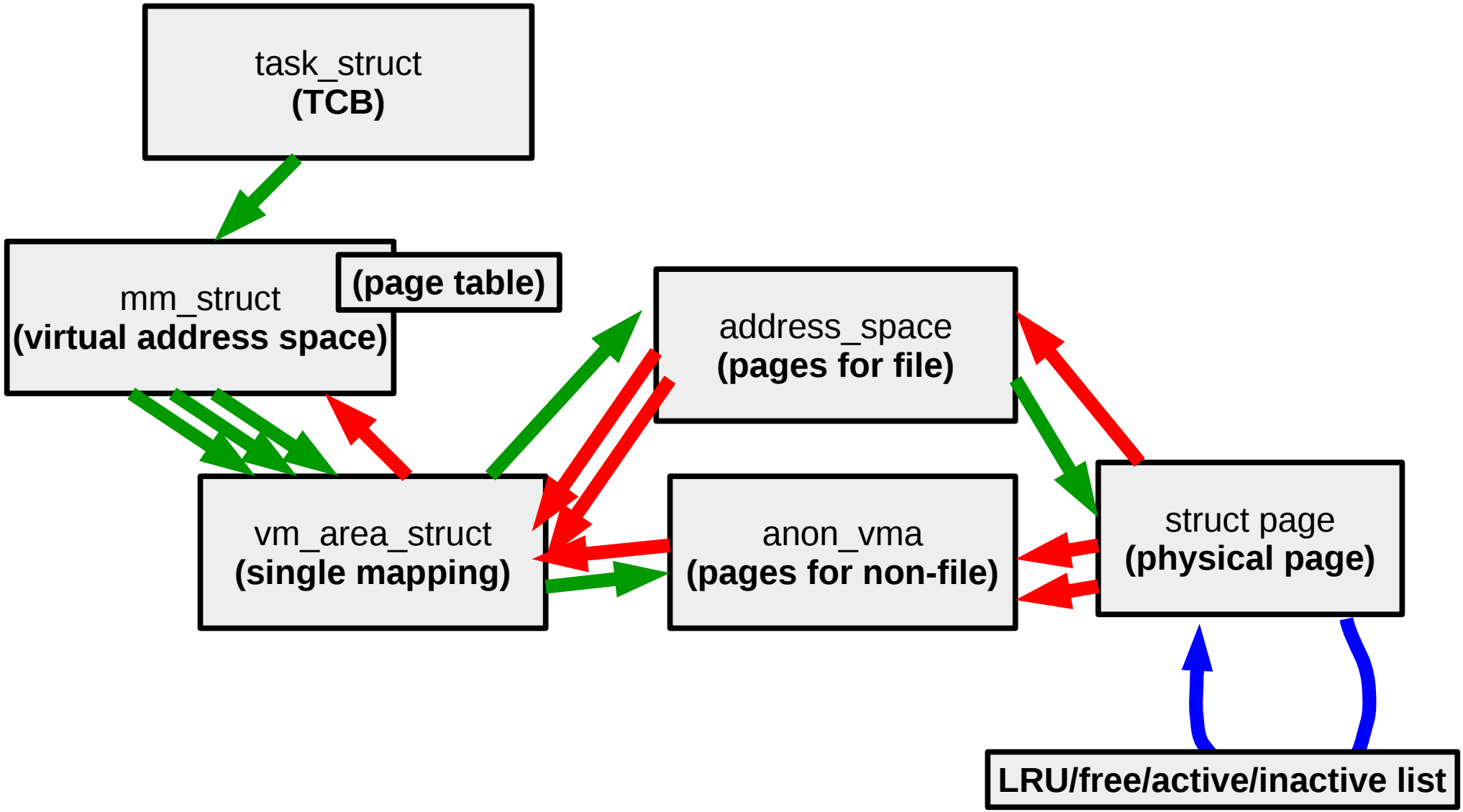
Not acceptable to wait for file-backed page to be ready to free to write it back

- Lose days of data?

Solution: background thread scans for dirty pages

- Write back data to files after no more than ~30 s (configurable timeout)
- Keep enough non-dirty memory to allow fast page allocation (configurable threshold)

Linux VM Data Structures



Logistics

Midterm Wednesday

Review session during class Tuesday

Project 2 – Design reviews **FRIDAY**

Recall: What is an operating system? (4)

Always:

- ***Memory management***
- I/O management
- ***CPU scheduling***
- Communication?

Sometimes:

- Filesystems
- ? Multimedia support
- ? User interface
- ? Internet browser

CPU management

CPU **scheduling** with conflicting goals:

- Response time – interactivity
- Throughput – CPU and I/O
- Fairness and avoiding starvation
- Simplicity of scheduler
- Limiting overhead

CPU abstractions

The **thread** – a virtual CPU

Represents **registers**

Has independent stack

- but in (potentially shared) address space

Synchronization

Multiple (virtual) CPUs → things can happen "at the same time"

Abstractions:

- **Monitors – Lock + Condition Variable**
- **Semaphores**
- **Locks**

Recall: Monitor Example: Queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();   // Get Lock
   queue.enqueue(item); // Add item
   dataready.signal();  // Signal a waiter, if any
   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);
}
```

Lock Building Blocks

Single processor: Interrupt disabling

- lowlevel_acquire() { disable interrupts; }
- lowlevel_release() { enable interrupts; }

Between processors: Atomic read-modify-write

- lowlevel_acquire() { while (test&set(&lock)); }
- lowlevel_release() { lock = 0; }

Deadlock

AllocateOrWait(1MB)

/* 1 MB free */

AllocateOrWait(1MB)

/* 0 MB free */

AllocateOrWait(1MB)

- **WAITS FOREVER**

AllocateOrWait(1MB)

- **WAITS FOREVER**

Recall: Four Requirements

Mutual exclusion

- One thread at a time can use a resource

Hold and wait

- Thread holding a resource waits to acquire another resource

No preemption

- Resources are released voluntarily – threads can't steal instead of waiting

Circular wait:

- There exists a set {T1, ..., Tn} of waiting threads
  - T1 is waiting for a resource that is held by T2
  - T2 is waiting for a resource that is held by T3
  - ...
  - Tn is waiting for a resource that is held by T1

Process

One or more threads + **address space**

Address Translation Comparison

	Advantages	**Disadvantages**
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation	Table size ~ # of pages in virtual memory, fast + easy allocation	Multiple memory references per page access
Two(+)-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex

System Calls

How do we get at all this functionality?

Controlled entry into kernel

- To **well-known location**
- Like every entry save registers
- Extract arguments out of registers + user memory

Recall: Dual Mode Operation

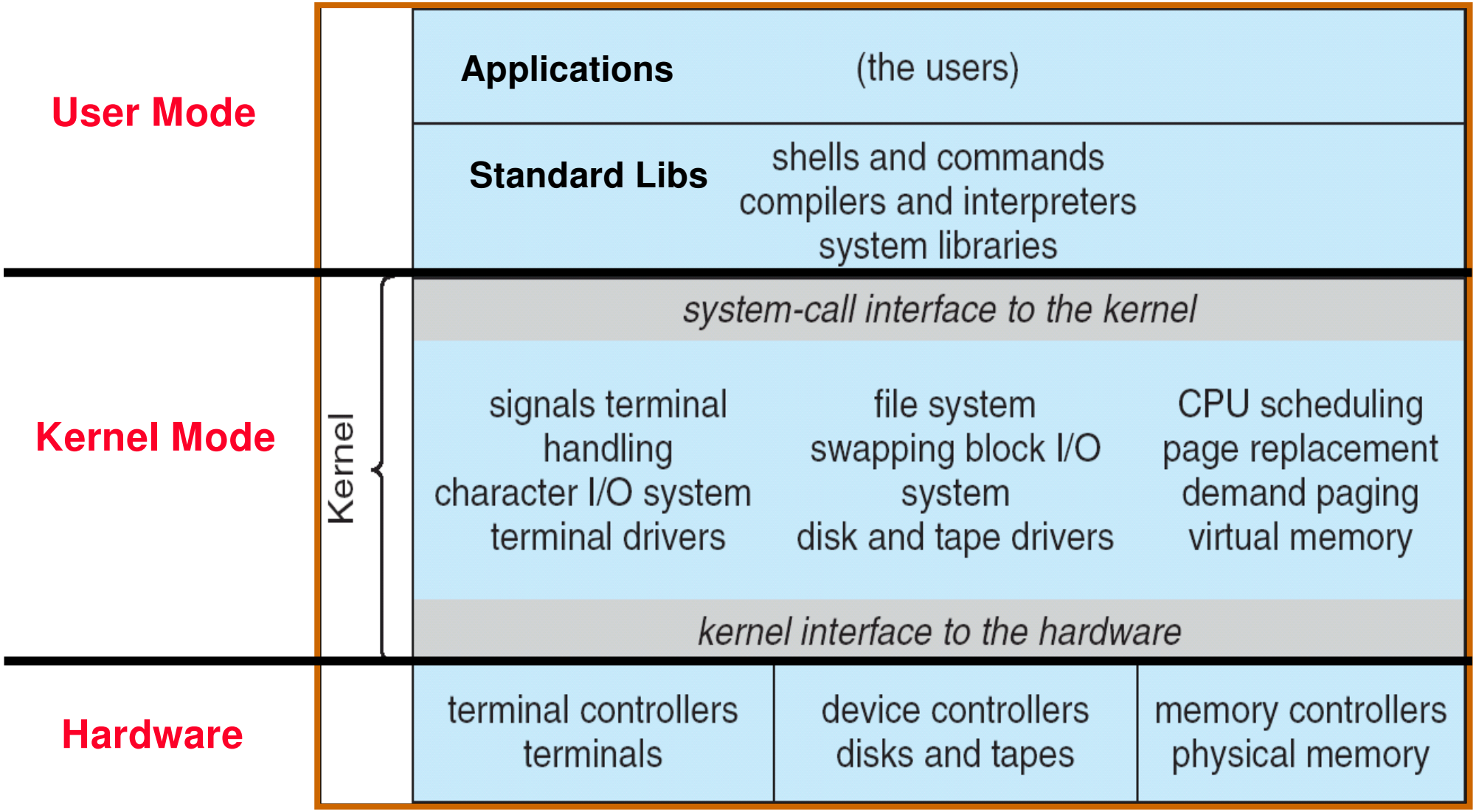
Hardware provides at least two modes:

- "**Kernel**" (or "supervisor" or "protected")
  - Unix "kernel" runs here (part of OS running all the time)
- "**User**" mode
  - Normal programs run here
  - Even if "administrator" or "superuser" or "sudo"

Some operations ***prohibited*** in user mode

- e.g. changing the page table pointer

UNIX System Structure



Process and Thread APIs

POSIX *process* APIs:

- fork – copy current into *new address space*
  - **copy-on-write**
- wait – wait for forked process
- exec – replace current with program on disk
  - completely new address space
  - inherit file descriptors

Pintos *thread* APIs:

- thread_create (kind-of like fork)
  - allocate new thread + stack for it
- thread_join (kind-of like wait)
  - wait for thread to finish

POSIX IO APIs

Low-level (buffered only in kernel space)

- open
- read
- write
- close

High-level (buffered in userspace, too)

- fopen,
- fread, fgetc, fgets
- fwrite, fputs, fprintf
- fclos

Recall: POSIX IO: Everything is a file

Uniform interface for

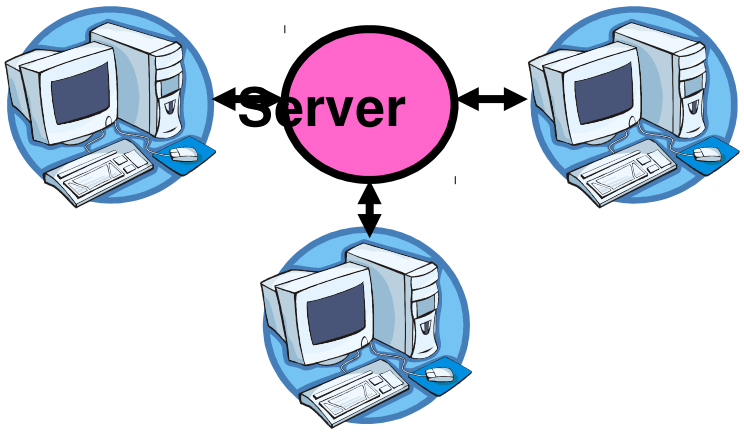
- Devices (terminals, printers, etc.)
- Regular files on disk
- Networking (sockets)
- Local interprocess communication (pipes, sockets)

Part of the process state

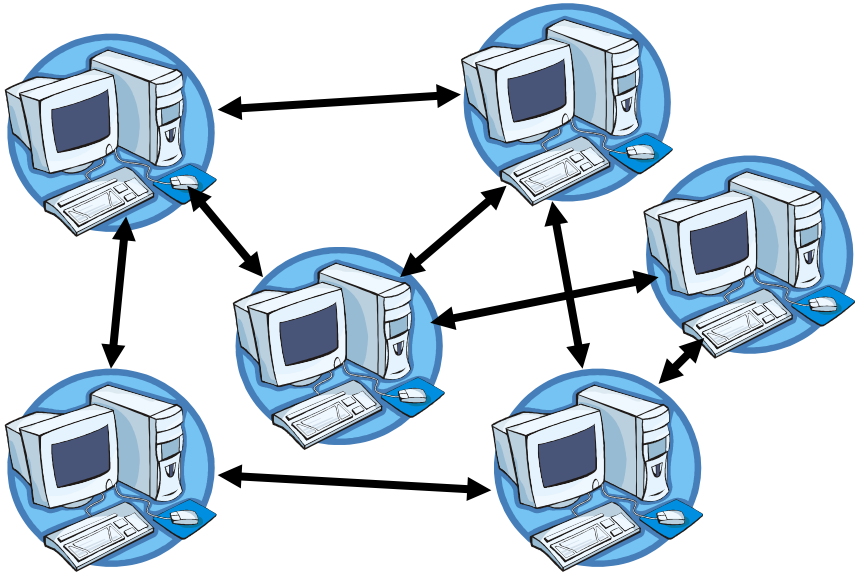
- accessed with file descriptors

Break

Centralized versus Distributed



Client/Server Model



Peer-to-Peer Model

Centralized system: System where major functions performed on one *physical* computer

Distributed system: Physically separate computers working together to perform a single task

Parallel versus Distributed

Distributed – different machines responsible for different parts of task

- Usually no centralized state
- Usually about different responsibilities or redundancy

Parallel – different parts of same task performed on different machines

- Usually about performance

Distributed: Why

Simple, **cheaper** components

Easy to **add** to **incrementally**

Let **multiple users cooperate** (maybe)

- Physical components owned by different users
- Enable **collaboration** between diverse users

Distributed: Promise

Availability

- One machine goes down, *system* up

Durability

- One machine loses data, *system* does not

Security

- Divide security problem into simpler pieces?

Distributed: Reality (sometimes)

Availability

- One machine takes them all down

Durability

- Any machine can lose your data

Security

- More places to break in

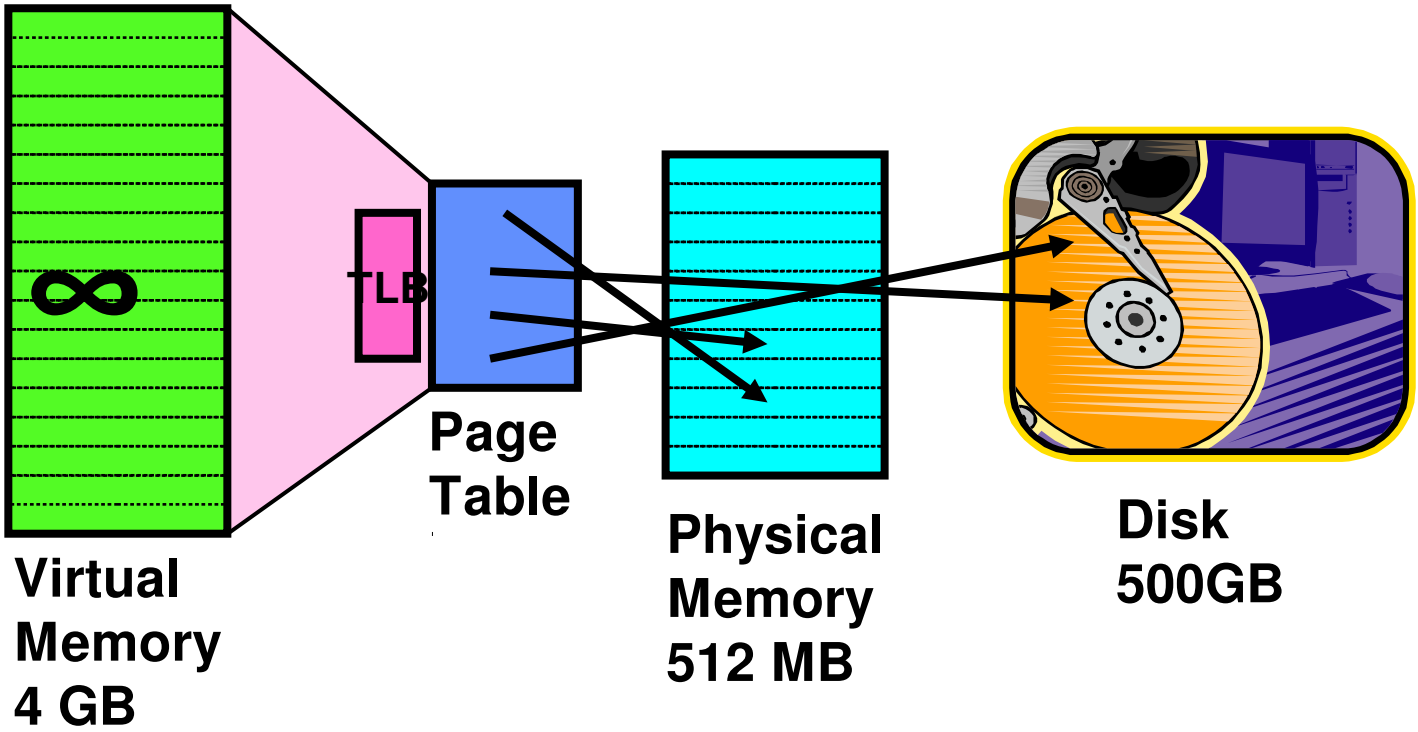
Distributed: Goal

Transparency: Hide distributedness, make system simpler

Kinds:

- *Location* – location of resources invisible/irrelevant
- *Migration* – location of resources changes invisibly
- *Replication* – invisible extra copies of resources (for reliability, performance)
- *Parallelism* – jobs split into pieces, look like on serial task
- *Fault tolerance* – components fail without users knowing

Recall: Illusion of "infinite" memory



How? **Transparent layer of indirection**

- the page table + page fault handlers

The Coordination Problem

Components communicate with network:

- Lets you send **messages** between machines

Need to use messages to **agree on state**

- Problem **does not exist** in centralized system

Protocols

Protocol is **an agreement on how to communicate**

- *Syntax*: "structure" – format, order of messages
- *Semantics:* "meaning" – actions taken when transmitting, receiving, or when timer expires

Formal description: ***state machine***

A distributed system is embodied by a protocol

Human Protocols: Telephone

1. (Pick up / open up the phone)
2. Listen for a dial tone / see that you have service
3. Dial
4. Should hear ringing …
5. Callee: “Hello?”
6. Caller: “Hi, it’s John….”
   Or: “Hi, it’s me” (←- what’s *that* about?)
7. Caller: “Hey, do you think … blah blah blah …” pause
8. Callee: “Yeah, blah blah blah …” pause
9. Caller: Bye
10. Callee: Bye
11. Hang up

Protocols With Humans: Asking a Question

1. Raise your hand

2. Wait to be called on

or

1. Wait for speaker to pause.

2. Speak.

Models for Organizing Distributed Systems

Client/server

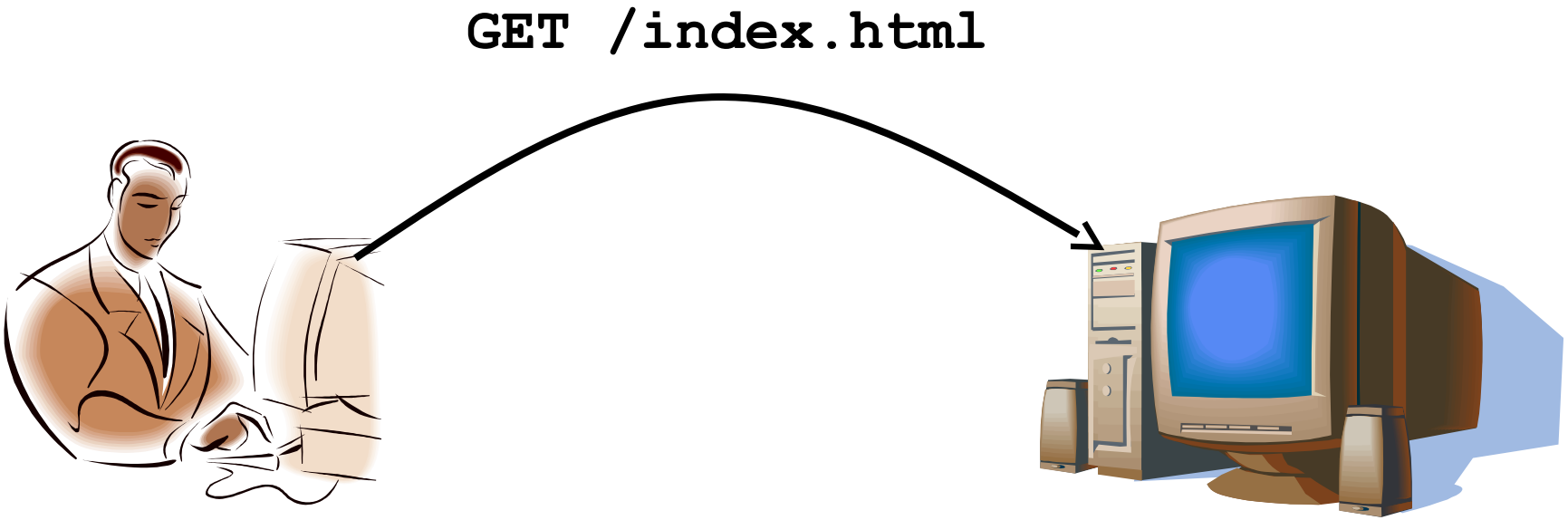
Peer-to-peer

???

Clients and Servers

Client program

- Running on end host
- Requests service
- E.g., Web browser



Client-Server Communication

Client “sometimes on”

- Initiates a request to the server when interested
- E.g., Web browser on your laptop or cell phone
- Doesn’t communicate directly with other clients
- Needs to know the server’s address

Server is “always on”

- Services requests from many client hosts
- E.g., Web server for the www.cnn.com Web site
- Doesn’t initiate contact with the clients
- Needs a fixed, well-known address

Peer-to-Peer Communication

No always-on server at the center of it all

- Hosts can come and go, and change addresses
- Hosts may have a different address each time

Example: peer-to-peer file sharing (e.g., BitTorrent)

- Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
- Scalability by harnessing millions of peers
- Each peer acting as both a client and server

Summary:

Distributed – multiple machines performing different parts of task

- Contrast to parallel: multiple (usually similar) things at the same time

Goal: Transparent – doesn't look like distributed system

Communication with **messages**

- Contrast with threads: shared memory