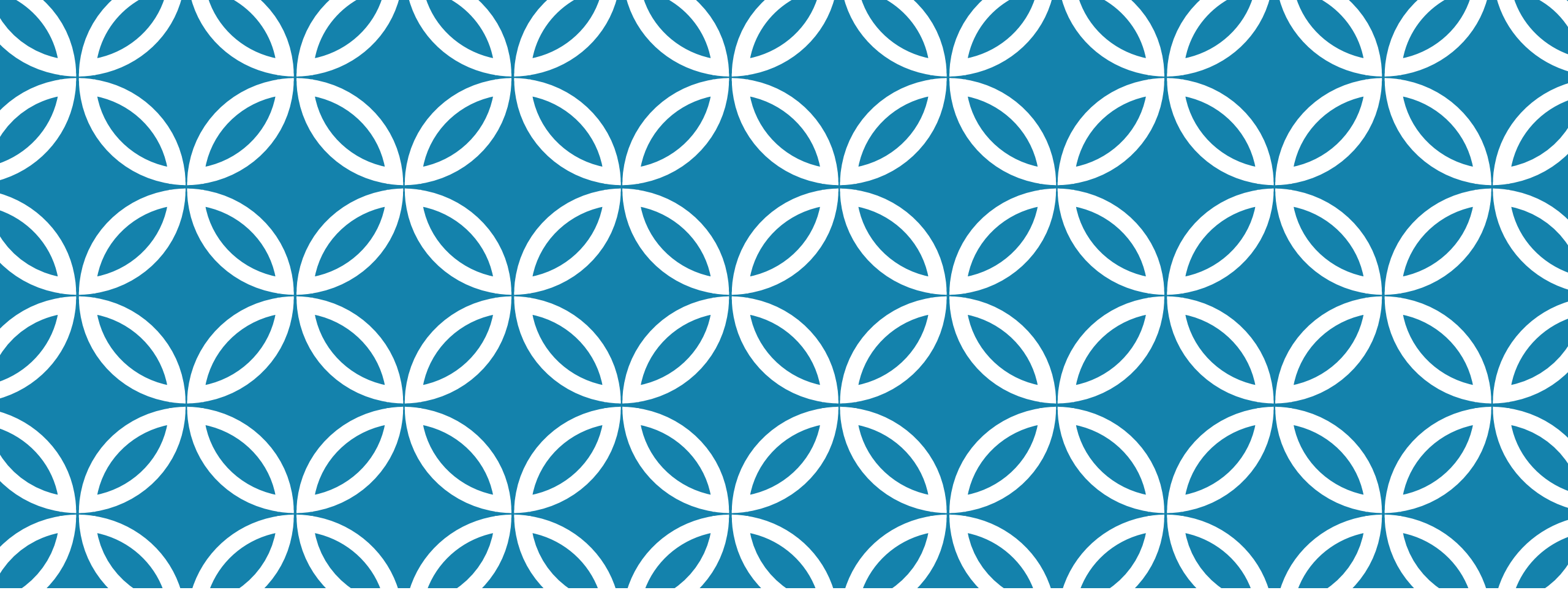


ABSTRACTION ↔ PERFORMANCE: AN ENDURING BATTLE

Professor Ken Birman

CS4414 Lecture 6

IDEA MAP FOR TODAY

A thing should be as simple as possible. This argues for elegant abstractions

Yet some things are just not simple, like NUMA hardware! This argues for powerful APIs that expose performance-critical controls

Complex tensions: Simplicity/expressiveness.
Performance/elegance
Correctness and Security/ convenience

Virtualization arises in many forms in Linux: virtual memory, the process abstraction, full virtual machines, container virtualization. These offer good examples of those tensions

To illustrate this idea we will look at the file system abstraction

EARLY COMPUTER SYSTEMS DIDN'T HAVE AN OPERATING SYSTEM!

You wrote your program out on paper

Then "toggled it into memory"

Put the start execution address into the PC register by hand.
Pressed "run"

CODE REUSE EMERGED AS AN IMPORTANT GOAL

Once you have a working solution such as code to print to the lineprinter, why reimplement it?

At first you had to attach a punch-card deck to your punch-card deck, with a copy of the code you wanted to reuse.

This led to the idea that a computer should have built-in functionality to manage the hardware and make programming easier. The idea of an operating system was born!

THE O/S AND THE KERNEL

Some parts of the operating system run the computer hardware, but other parts do other tasks, such as copying files.

Over time, we began to refer to the resident portion of the O/S as the kernel, and the rest as "additional O/S components"

… it quickly turned out that protecting the kernel is important!

THE DAWN OF HACKING (IBM 360, 1978)

The dawn of whacking. 1.8M BC

The IBM 360 kernel managed a list of user names and passwords.

Of course, passwords shouldn't be in plain text. So the passwords were encrypted by a messy function that did a lot of shift and rotate and multiplication operations and ended up with a 16-bit number.

On the IBM 360, the kernel memory was "visible" to any process

GUESS WHAT?

With the help of a friendly math student, it was possible to invert this function.

Helpful math wizard

… for each user account, we ended up with a whole list of passwords!

The bad guy went that-a-way

Late one night, the "IT police" rushed to the computer center… my list of passwords was right there, but they didn't recognize them. So, it didn't occur to them that I was the criminal!

IN FACT THERE WERE MANY ISSUES

Theft of passwords was a big issue, but even without passwords, any user could literally see the memory in use by other users.

Personal information was completely visible, such as healthcare records, payroll records, etc.

Even the code of the kernel itself was visible. Other companies could dissemble it as a shortcut to competing with IBM!

IBM 370 KERNEL PROTECTIONS

… So, it was not a surprise when the IBM 370 introduced a wide range of additional kernel protections!

- User mode / kernel mode: The kernel had a long list of special instructions that it could execute, that were illegal for users.
- Each user process was given its own page table, and limited to seeing its own memory (the kernel, in contrast, "sees all").
- Traps and exceptions switched from user mode to kernel mode

Take a security course to learn more, including how some traps returned to the user… in kernel mode!

PROTECTION OF THE FILE SYSTEM

The file system is just a data structure built and managed by the kernel, on the storage unit (disk or USB drive or whatever).

In Linux, you could literally "see" that disk in two ways:

- You can chdir into the folders and open files and access them.
- If you knew the name of the device, such as /dev/usb2, you could access that device in "raw" mode and see the actual blocks holding the directories and files.

WHY WOULD THIS MATTER?

Imagine that you and your friends are sharing a top-secret plan for the big surprise party for Rachel.

Rachel has access to your computer, so once you print the plan you delete all the copies – including any backup copies.

But Rachel has a friend who is a “disk forensics specialist”…

HOW DO FILE SYSTEMS REALLY WORK?

Think about malloc from our previous lecture.

Linux is managing a form of tree, on the disk, by reading blocks (fixed size, default is currently 4192 bytes), modifying data within them, then writing them back.

It has a free list of blocks: new space needs are satisfied from it, and any freed blocks (deleted files) are added to this list.

LINUX FILE SYSTEM DATA STRUCTURE

Could span multiple storage devices, but we will focus on one

Can reach over the network to a file system server (but we won't worry about that case)

Supports various "kinds" of files: small, medium, large, immense. File names can also be short or long or very, very long.

LINKS

In Linux, we distinguish the inode (the "file data structure") from the name. A name is said to be a "link" to the inode.

A single file (or even directory) can have multiple names.

- Original form: "link". A and B can be two names referencing the same inode number. Only works in a single file system, *not across mount points*.
- New form: "symbolic link". File B contains the actual name for file A. Works in a similar way… yet not identical.

LINKS

In l … n the
nar…

A s…

- …me
- …*ints.*

Thought puzzle 1:

Suppose file "B" is a link to file "A".

You delete A. What will happen to B?

LINKS

In Li… the nam…

A si…

- C… e i… nts.
- N… V…

Thought puzzle 2:

Suppose file "B" is a link to file "A".

You edit A. What will happen to B?

SYMBOLIC LINKS

These were added later because with mounted file systems, it was a problem that links didn't work across mount points.

With a symbolic link, a file can contain a pathname.

If you access it (or search through it), Linux "switches" to the symbolic link pathname.

SYMBOLIC LINKS

it

Thought puzzle 3:

Suppose file "B" is a symbolic link to file "A".

You delete A. What will happen to B?

SYMBOLIC LINKS

it

Thought puzzle 4:

Suppose file "B" is a symbolic link to file "A".

You edit A. What will happen to B?

SYMBOLIC LINKS

Thought puzzle 5:

Suppose file "B" is a symbolic link to file "A".

You delete B. What will happen to A?

SYMBOLIC LINKS

it

Thought puzzle 6:

A is a symbolic link to B. B is a symbolic link to A.

What happens if you try to open A?

POSIX FILE SYSTEM API

You access files via the "POSIX" API.

A process first must open the file:

```
int fd = open("filename", O_RDWR);
```

Now the file descriptor, fd, can be used to access the bytes. Linux considers all data files to just be buckets of bytes.

POSIX FILE SYSTEM API

Many Linux system calls take extra arguments, often as bit masks, where each set bit requests some feature.
O_RDWR is a mask that means "open for reading and writing…

A process first must open the file:

```
int fd = open("filename", O_RDWR);
```

Now the file descriptor, fd, can be used to access the bytes. Linux considers all data files to just be buckets of bytes.

POSIX FILE SYSTEM API

You can also create a file with “open”:

```
fd = open(“file”, O_CREAT, S_IRWXU|S_IRGRP|S_IROTH);
```

For O_CREAT you specify “permissions” on the new file, for yourself (as owner), your “group” (team members) and “others

MORE POSIX OPERATIONS

```
lseek(fd, location, SEEK_SET);  // Move file "pointer"
nb = read(fd, buffer, nbytes); // nb will be "bytes actually read"
write(fd, buffer, nbytes);      // Write nbytes at the current pointer
close(fd);                      // Releases resources
```

… many of these have also have options.

C++ FILE WRAPPERS, MMAP

In C++, wrappers are sometimes used to treat files like objects.

- Use **std::iostream** if you plan to just scan a text file.
- Use **std::FILE** if your file is a series of fixed-sized records.

There is also a way to "map" a file into memory: **mmap**.

- A mapped file looks like a vector of bytes (in C++, of type "char*")
- You read or write the data by simply indexing into this vector.
- Must call fsync() or close() to force the writes back out to disk.

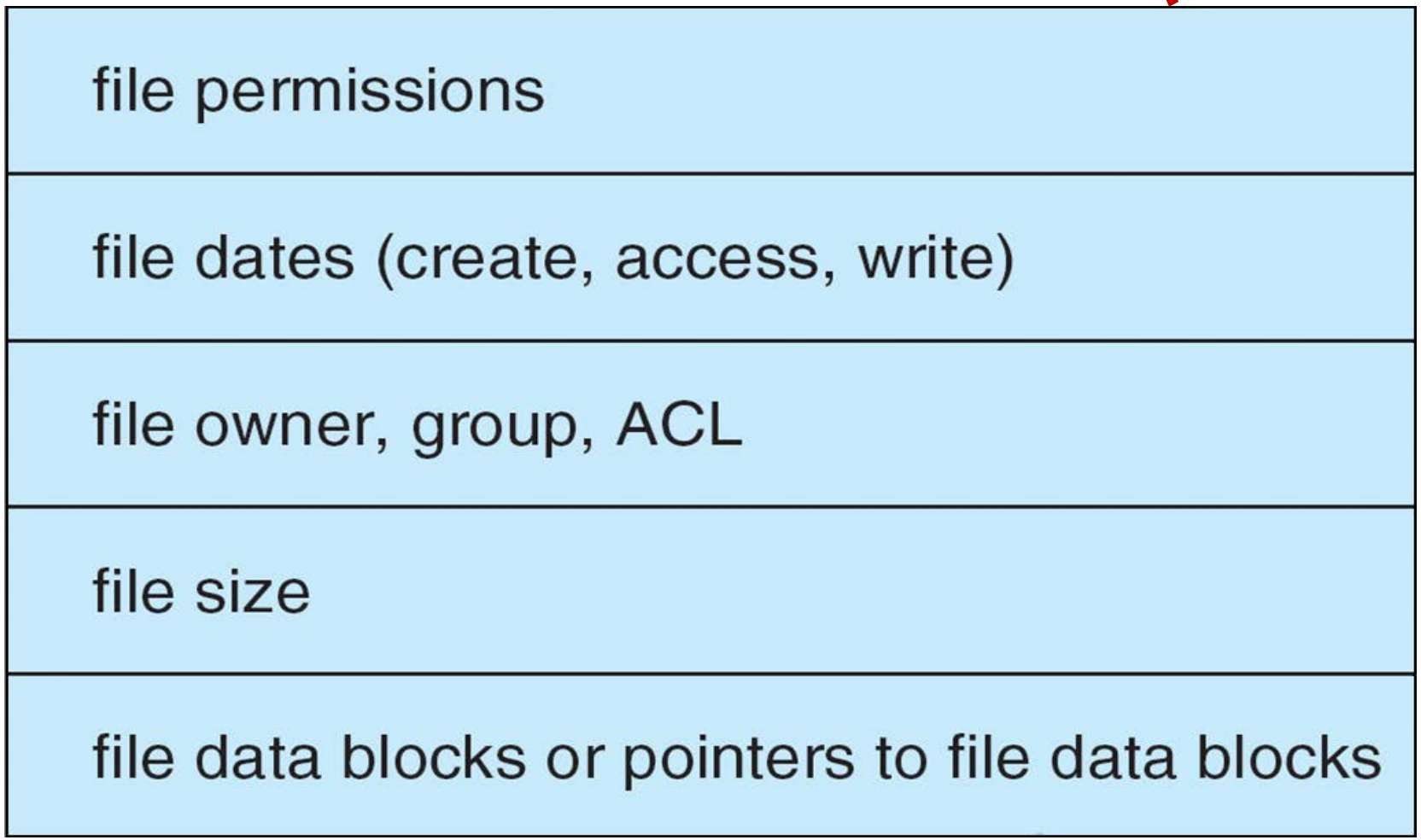
STORED DATA RESIDES IN BLOCKS

Linux stores data in fixed-sized blocks. Each file has a length, in bytes. The last block might be partially filled.

A process can't read beyond the end of the file (EOF): read indicates this by returning **nb < nbytes**. nb can be zero

You can create a *gap* by seeking beyond the end of a file and then writing. Linux returns 0's if an application reads the gap.

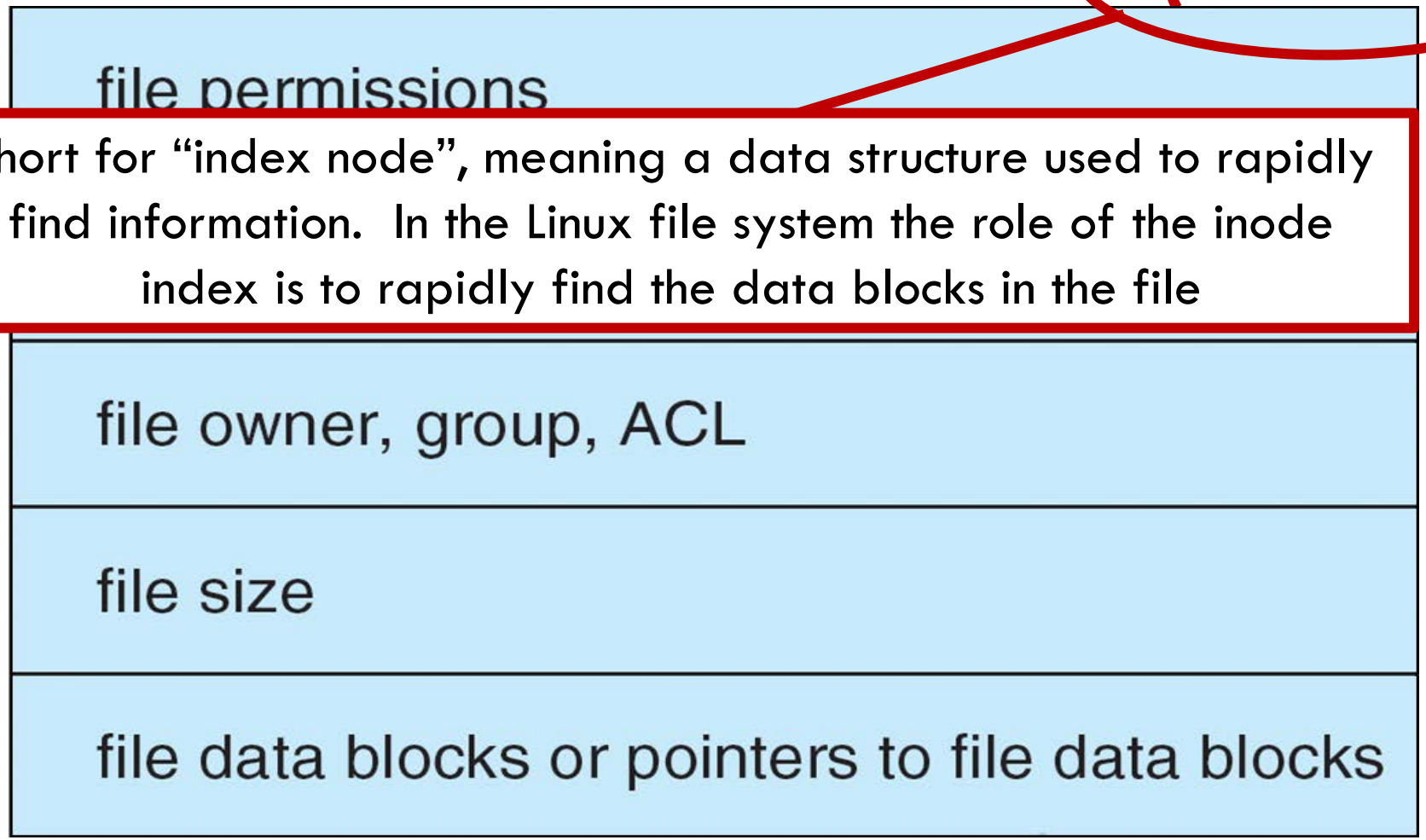
ON THE DISK AND INSIDE THE KERNEL, A FILE HAS A FILE CONTROL BLOCK (INODE)



For historical reasons, Linux calls this an *inode* structure

Each inode has a unique id number.

ON THE DISK AND INSIDE THE KERNEL, A FILE HAS A FILE CONTROL BLOCK (INODE)

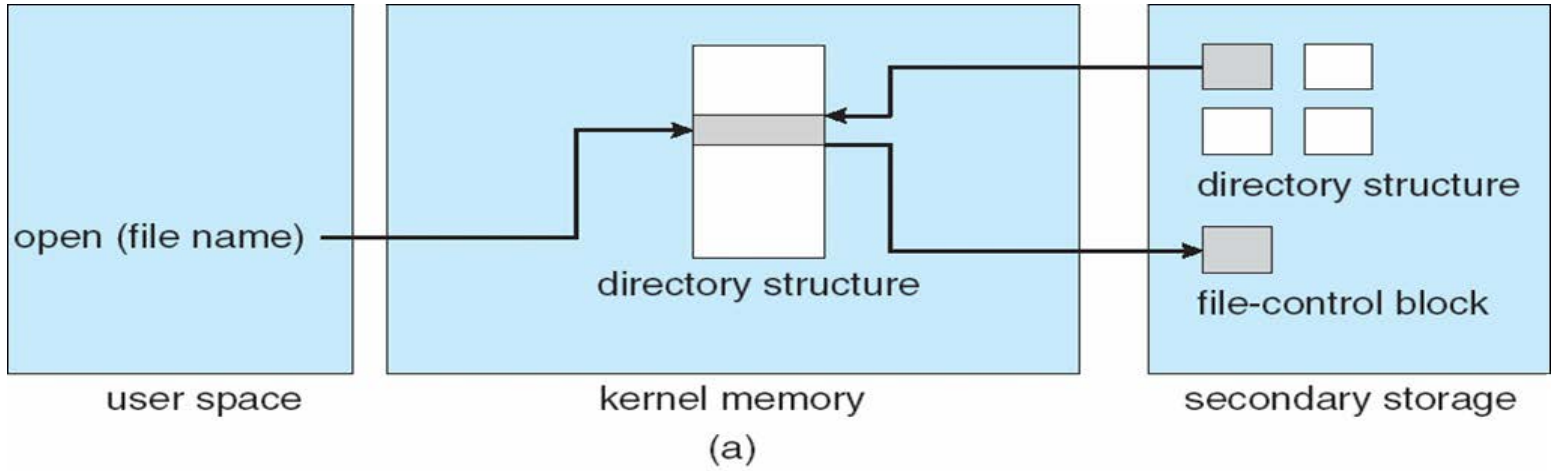


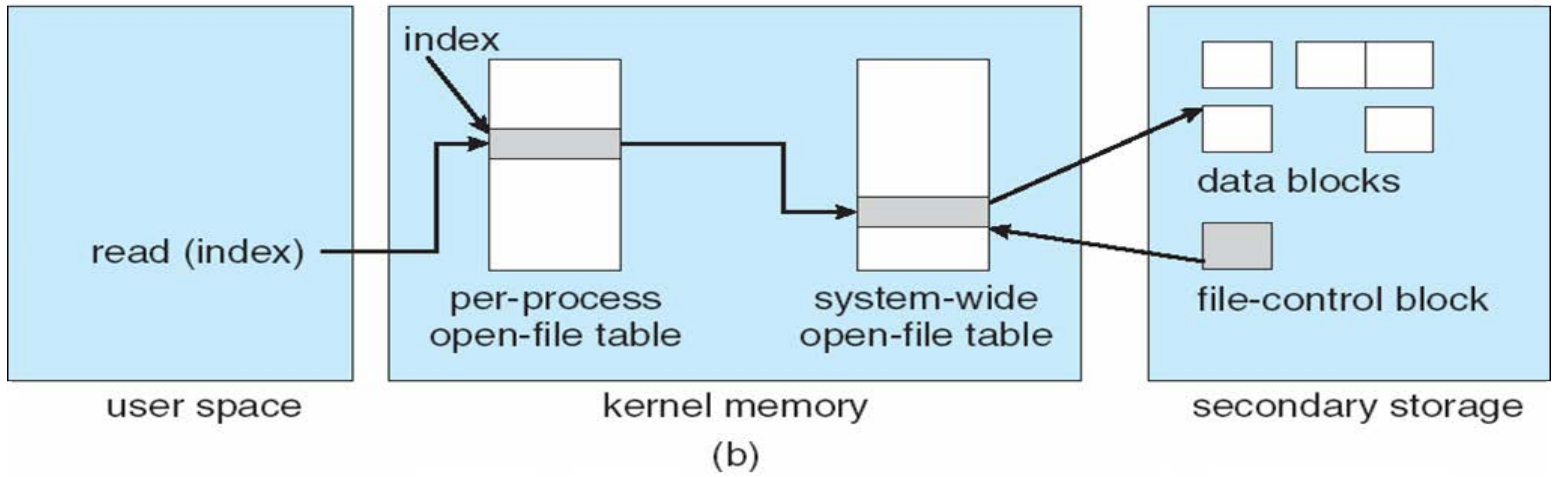
Short for "index node", meaning a data structure used to rapidly find information. In the Linux file system the role of the inode index is to rapidly find the data blocks in the file

For historical reasons, Linux calls this an *inode* structure

Each inode has a unique id number.

IN-MEMORY FILE SYSTEM STRUCTURES



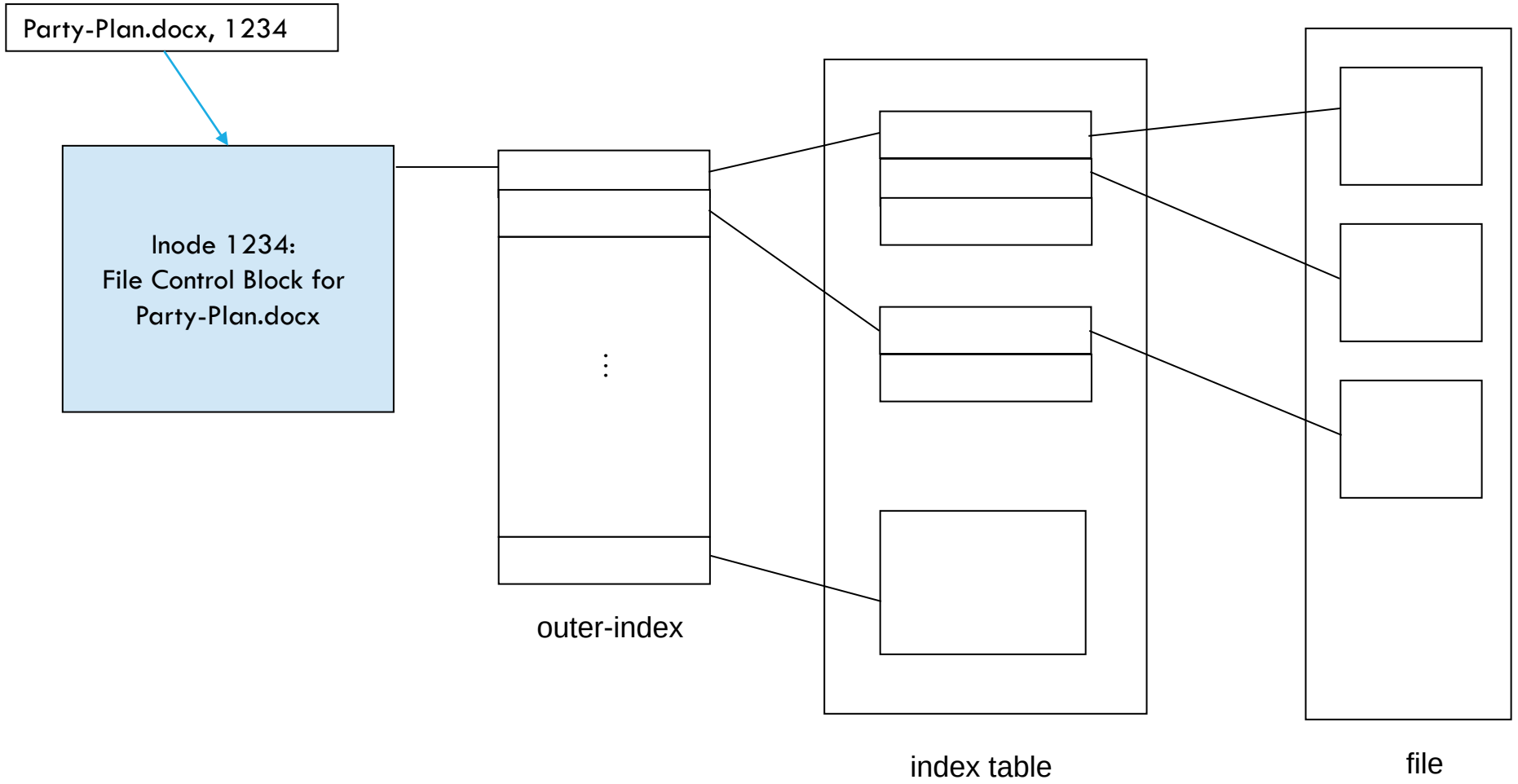


One file or directory can have multiple "alias" names (links)

A name in a directory lets us determine the inode number. This gets us to the data structure with file content information

Once a file is open, the inode is also listed in the kernel's "open files" table. Again, we can use this to get to the inode data structure containing information about the file permissions and contents.

FILE DATA IS STORED IN BLOCKS

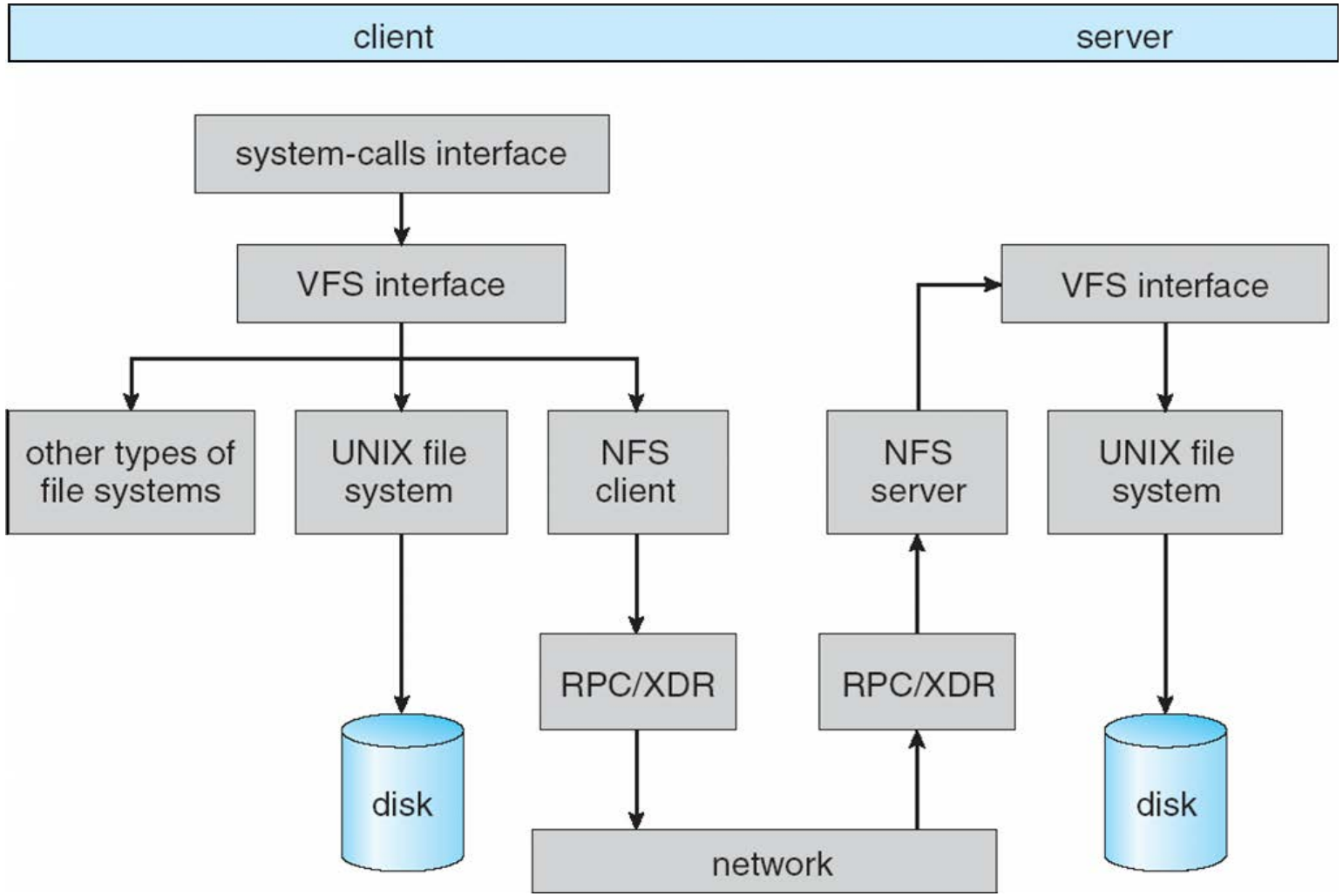


To keep the inode structure size small, the inode itself only can list a small number of blocks. If the file is small, these hold data.

For a large file, each of these blocks themselves hold lists of blocks: a hierarchy of index blocks, with the actual data blocks as the "leaves" of the structure.

Huge files have additional layers of index blocks

WITH A NETWORK FILE SYSTEM, SAME IDEA…



…. SO, WHEN YOU DELETED "PARTY-PLAN.DOCX"

In fact the actual disk I/O that occurred was this:

- Linux accessed the block containing the directory, zeroed the inode number next to the file name, rewrote the block.
- Linux accessed the tree node for the file (called an "inode") and changed its state from allocated to free. It put the inode on a freelist
- Linux walked down the list of blocks in the file, and put them on the freelist for disk blocks, and wrote that back to the disk.

What did Linux *not* do?

…. SO, WHEN YOU DELETED "PARTY-PLAN.DOCX"

Linux never zeroed the actual contents of the inode, it only was put on the inode freelist. It never overwrote the file name – it just changed the inode number in the directory to 0.

And it never zeroed the contents of the file, either. It simply put the blocks on the freelist for blocks.

Thus, if you can "find" the inode, you can still reconstruct the whole file, until those blocks are actually reused for some other purpose!

WHY WAS THE KERNEL SO "LAZY"?

Linux is optimized for speed.

The designers thought about it this way: if you really wanted to ensure that Party-Plan.docx was destroyed, you could have used a tool to rewrite the actual bytes with gibberish, before deleting it.

You didn't do this, hence you must care mostly about speed, and so you want Linux to be as fast as possible for file deletes.

IS IT POSSIBLE TO "REALLY" DELETE A FILE?

In some situations, yes. There are Linux tools to help you do this. Spies use them… most users don't even know about them.

They overwrite the file bits with random garbage dozens of times.

But for most mortals, the real answer is: *Maybe not*. Any file you create, or download – including a web page – may linger on your machine! (And web pages can even have hidden content)

RACHEL'S FORENSIC TOOLKIT

Her friend gave her copies of programs, easily downloaded from the network (in fact Linux even has standard ones you can install using apt-get) that

- Open the disk as a raw block device
- Scan the inode freelist looking for inodes that were freed yet where haven't actually be reused for some other purpose yet
- Access the corresponding blocks, copying their data

This allows them to generate "recovered files" without the proper name, but with some or even all of the data that they had when deleted!

THESE TOOLS BYPASS FILE SYSTEM SECURITY

In the file control block is a record of the file owner, permissions and last access time.

But when accessed as a raw disk block, the kernel ignores the existence of that file system data structure and treats the whole disk as a huge block device. The blocks are just byte arrays!

So normal file system permissions aren't checked.

… WHICH IS WHY LINUX NORMALLY ALLOWS THEIR USE ONLY BY THE SUPERUSER

You can't just open /dev/usb2 as a raw device without permission to access /dev/usb2 in the first place!

However… those permissions are fairly soft. If I found a backup of your disk, and plugged that into a Linux computer *without telling it that the USB contains a file system*, I could connect to it in this raw mode, and then could access the contents.

… HOW WOULD RACHEL EVEN FIGURE OUT THE FILE NAMES?

Some versions of the Linux kernel wipe clean the names of deleted files.

But the file system supports very long file names, and it can be costly to zero all of those bytes.

So many versions of Linux delete files by just zeroing the inode number in the directory, leaving the file name itself untouched.

… OF COURSE IT MAY NOT BE TRIVIAL TO MATCH THE NAME TO THE FILE

But many forensic tools make a good guess of which names match which inodes, with a good chance of finding

- Its old name ("Possibly PartyPlan.docx")
- The correct list of blocks
- Most or all of the data it used to contain

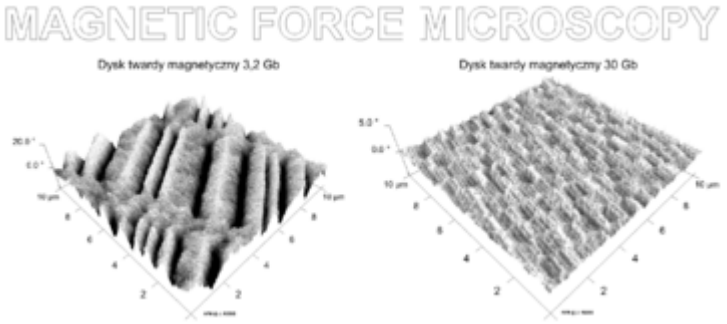
Rachel realizes her mom is not coming. ☹

A FILE SYSTEM MAY LOOK IMMACULATE…

… yet to a forensic specialist, the disk is like a genome, full of junk (non-coding) DNA that still reveals a lot about the past!

And this doesn't even consider forensic tools that try to use fancy hardware features to "recover" bytes that may have been written once, but then overwritten recently with zeros, or with random garbage.

THE FIRST-WRITE ISSUE



With most forms of storage, the first bits written in a block leave a ghostly impression.

With cutting edge forensic tools (magnetic force microscopy), those ghostly images can be pulled off the disk. Then you can use tools (like gdb) to examine the data.

EVEN USB STORAGE DEVICES CAN BE DANGEROUS!

Hackers have attacked via “poisoned” file systems.

How this worked: they damaged the file system data structures, and when mounted, this allowed them to take full control!

The damage could be found and repaired by checking integrity of the file system structure before mounting it. But with a large file system this is quite slow.

2008 cyberattack on United State

en.wikipedia.org/wiki/2008_cyberattack_on_United_States

Apps BT | WIFI Google News Sign In Imported From Edge New Tab

Not logged in Talk Contributions Create account Lo

WIKIPEDIA
The Free Encyclopedia

Article Talk Read Edit View history Search Wikipedia

2008 cyberattack on United States

From Wikipedia, the free encyclopedia

The **2008 cyberattack on the United States** was a malicious attack done by foreign actors on the United States Department of defense. Described as the "worst breach of U.S. military computers in history", the defense against the attack was named "Operation Buckshot Yankee". It led to the creation of the United States Cyber Command.[1][2][3]

Contents [hide]

1 History
  1.1 Operation Buckshot Yankee
2 References
3 Further reading

History [edit]

It started when a USB flash drive infected by a foreign intelligence agency was left in the parking lot of a Department of Defense facility [citation needed] at a base in the Middle East. It contained malicious code and was pu into a USB port from a laptop computer that was attached to United States Central Command. From there it spread undetected to other systems, both classified and unclassified.[1][2]

Operation Buckshot Yankee [edit]

The Pentagon spent nearly 14 months cleaning the worm, named agent.btz, from military networks. Agent.btz, a variant of the SillyFDC worm,[4] has the ability "to scan computers for data, open backdoors, and send through those backdoors to a remote command and control server."[5] It was suspected that Russian hackers were behind it because they had used the same code that made up agent.btz before in previous attacks. In order to try to stop the spread of the worm, the Pentagon banned USB drives, and disabled Windows autorun feature.[5]

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page

THIS IS A STORY ABOUT FILE SYSTEM SECURITY BUT ALSO ABOUT ABSTRACTIONS

We want to think about a storage system in terms of elegant, high-level concepts like files and directories and links.

Yet there is an underlying reality here: *data lives on a device*. And there are situations where we need direct access to devices, such as when importing files from a different operating system.

The tension is fundamental: abstraction ↔ reality, performance, …

DEFINITION: ABSTRACTION

Abstraction is **the process of filtering out – ignoring - the characteristics of patterns** that we don't need in order to concentrate on those that we do.

Abstractions arise we filter out details to focus on key ideas.

WE (WANT TO) TRUST THE KERNEL!

The O/S kernel is creating an illusion of private files, that we can safely delete or copy or modify.

But this illusion is only as good as the protection features that surround the O/S and those raw hardware devices. As historical operating systems got more complex, more and more issues surfaced!

When Unix was introduced, and then "reimplemented" as Linux, it was in part a response to a demand for simplicity

YET NO KERNEL IS ALL-POWERFUL

If someone can get physical access to a storage device, for example, those forensic tools still work. (Modern Linux does include the option of encrypting the contents of every file)

The idea, though, it to create a secure, performant, efficient environment for running programs and managing data.

LINUX THEMES

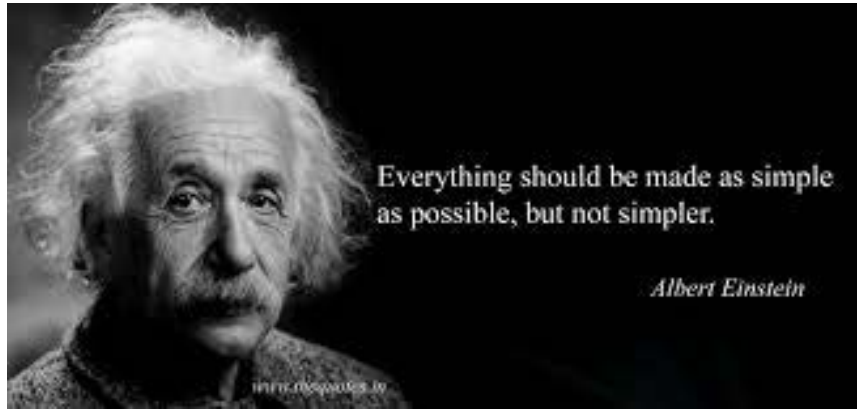
Minimalism: If something doesn't have to be in the kernel, leave it out! Instead of rarely used complex features, limit the kernel to simple, general features that can be combined to solve all needs!

Performance: The kernel should be blazingly fast and reliable.

Correctness and security: The kernel should behave as advertised, always, and defend itself against attacks.

MODERN HARDWARE IS NOT AT ALL SIMPLE…

… so this creates a dilemma. In fact, modern Linux is not remotely as simple as the earliest Unix.



Yet this tension between features and simplicity is still evident!

The modern concern is mostly centered on trust: If the kernel is small we have a better chance to verify its security!

LINUX IS MODULAR AND COMPOSITIONAL

We saw how Linux encourages combining a few programs to accomplish a general task.

The kernel itself has some basic modules, that combine to offer all sorts of broader functionality. These have simple interfaces.

In the CS4410 course, O/S, you'll learn all about how these work.

ROLES OF THE MODERN LINUX KERNEL

Manage the hardware and devices and file system. Be minimal, performant, correct, secure. Use energy efficiently.

Offer a process abstraction, segmented memory, threads.

Many computers are shared. Guarantee protection so that user A can't be attacked by user B (includes theft of data, disruption, viruses, crashes…). Create a "virtual private computer" for each.

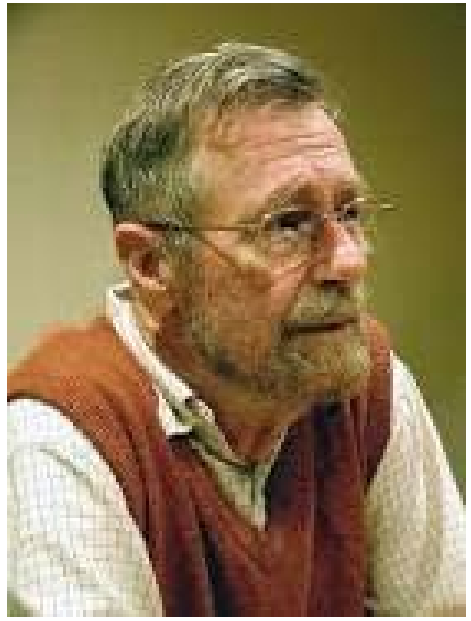
ROLES OF THE MODERN LINUX KERNEL

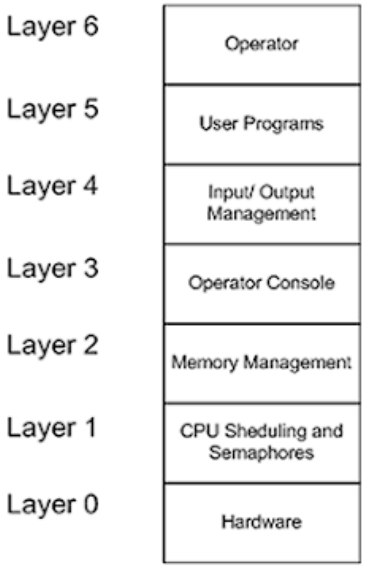
Be easy to manage ("administer"), and make it easy to diagnose problems when they occur.

Be portable: the same kernel should run on many kinds of computers, and it should be easy to move a user and her processes from machine to machine.

THEME: ABSTRACTIONS

Idea dates to Edsger Dijkstra and others.





… think of a complex system as a layered structure, in which each layer transforms layers below it into a higher-level abstraction

- At every layer we have data types, and abstract operations
- Each hides its implementation and introduces properties and guarantees

WIKIPEDIA

Consider some concrete computing task.

Now remove “details” until only the pattern remains.

This process allows you to identify the underlying abstraction. A mathematician would say we are “modelling” the task.

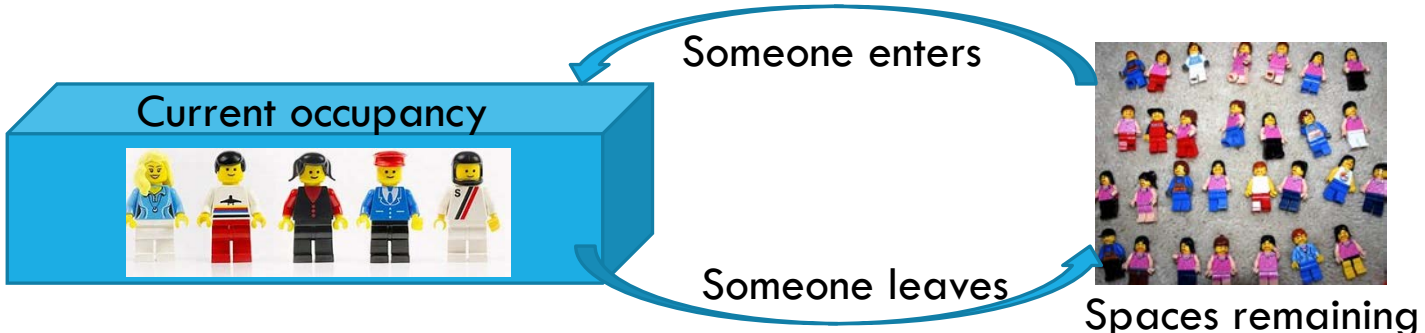
https://en.m.wikipedia.org/wiki/Abstraction_(computer_science);

EXAMPLE: GATEKEEPER AT BOOK SALE

Friend of the Library is limited to 50 customers at a time.

But as people come in and out, it was hard to count.

They adopted a "people counting" scheme



… THIS IDEA IS ADAPTABLE!

Two or more doors? Just partition the 50 tokens!

Load balance by shifting tokens (but you must keep them in the same status, of course)

Even works for ATMs with connectivity issues!

I have $150 of Ken's money available

$150 of Ken's money is blocked

THE "TOKEN ABSTRACTION"

Here we are using some form of token as an abstraction

The invariant differs: Maximum of 50 people inside. But they can enter or exit via any door

Account cannot be overdrawn

Yet the token abstraction covers all of these cases

PROS AND CONS OF ABSTRACTIONS

Dijkstra: *The real power of layered abstractions centers on specifications and proofs.* But proofs are just one aspect.

If we can fully describe the interfaces to our systems (APIs), and their behaviors, we can rigorously verify implementations.

This type of verification has been done for some versions of Linux and even for some C compilers (not C++ 11, however)

CONCRETE ABSTRACTIONS VERSUS CONCEPTUAL ONES

In low-level computing courses most abstractions map directly to some sort of object class, like a binary tree or a list. These are **concrete** abstractions.

But when we create systems, abstractions can be "ideas". We refer to these as **conceptual** abstractions.

We will give one example today and more in future lectures.

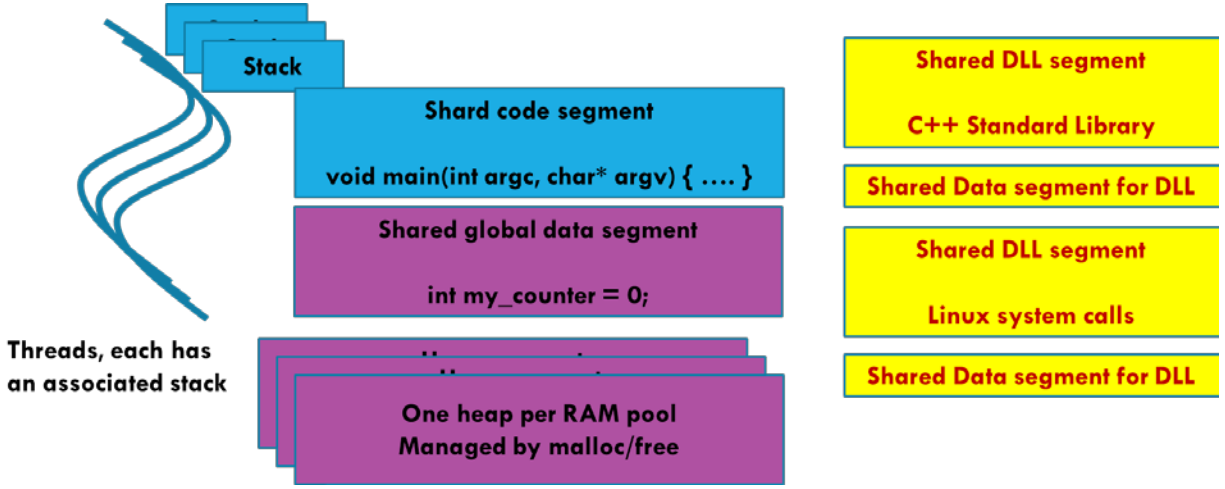
… A THREE-WAY TUG OF WAR!

We want simple, elegant abstractions. But abstractions can hide costs, as we saw with Python & Java!

We want to leverage high-performance hardware. But most hardware is very complex to use directly. Abstractions help!

We want security and protection… without sacrificing speed.

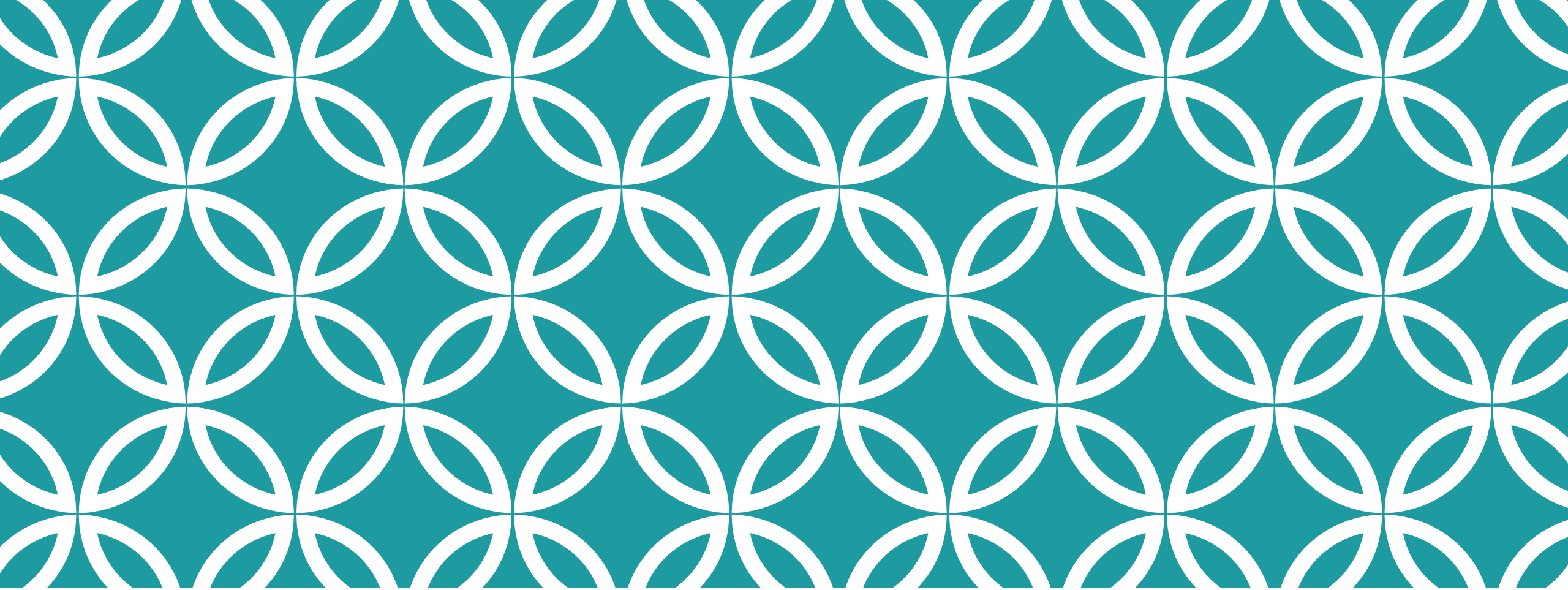
EXAMPLE: VIRTUAL MEMORY



In lecture 5, we learned about process address spaces (segments and paging), and used the term *virtual memory*.

The hardware doesn't "have" process address spaces. The abstraction is an idea that the kernel implements using hardware features.

If you see the term "virtual" it almost always means "an abstraction layered over a more general underlying feature."



VIRTUAL MACHINES AND CONTAINERS

Covered if time permits

VIRTUALIZATION CAN APPLY TO THE ENTIRE MACHINE

Dijkstra is suggesting that the entire computing experience is virtual. He sees layers and layers of abstractions. At the time this was a revolutionary concept.

Today, the entire computer actually can be "virtualized":

- We create a form of snapshot of the machine + your processes
- This virtual machine image can be moved easily, then restarted!

VIRTUALIZATION IS A VALUABLE TOOL!

It is extremely useful to be able to “move” programs or entire servers from place to place.

A virtual machine image is like a “program” that simulates your entire computer (even the file system, background jobs, etc).

You can even virtualize a legacy system and run it on a modern computer, if the modern system is compatible with the virtual image.

PROS AND CONS OF VIRTUALIZATION

But…

- Some programs depend on things they access over the network, such as default printers, remote file systems, databases
- These might not work if you move the VM to a setting where those other systems aren't available, or even if they have different host names

Virtualization also imposes some overheads, and many programs slow down (especially if several VMs actually share one file system: even if the VM is unaware, all the file open requests would need to be translated)

CONTAINER VIRTUALIZATION

There are actually two major kinds of virtualization

- With "true" virtualization, we literally virtualize the entire computer and run the original operating system as a process in some other computer.
- But as noted, this can be costly.

Container virtualization is a response (the "Docker" approach)

- In this approach a process or a set of processes are given the illusion that they run on a private computer, but in fact others can use it too.

CONTAINER VIRTUALIZATION

Each distinct user ends up with a seemingly private file system (in reality these are folders in a master file system, but they can't see the higher levels of it)

Each thinks they have private control over configurations, chron jobs and daemons, etc.

Sudo seems to work, but really works only within the scope of the user's own jobs. For example, a "su" user can't kill processes someone else actually owns (and can't see them, either).

CONTAINERS OFFER A PATH TO THE CLOUD

The approach centers on changes to Linux and to the libraries, and the famous packaging of this is Kubenetes+Docker.

Linux and the libraries work jointly to hide the processes one user is running from the processes other users created.

For example, the “ps” command, which lists active processes, will only show the ones I am running.

CONTAINERS OFFER A PATH TO THE CLOUD

A major trend is to develop code on your own computers but eventually run the production versions "in the cloud".

Here the idea is that a big company (such as Amazon, Microsoft or Google… but there are many more) operates a huge data center, maintaining the machines and managing them carefully.

They offer a virtualized "private cloud" to each customer.

ABSTRACTIONS AND SECURITY

Abstractions simply and allow us to work with a layered computing model.

Yet mundane tasks (like needing access to raw disks in order to import data from other operating systems, or to fix damage after a nasty crash) often involve tools that can "break" the abstraction boundary. Doing so exposes security risks!

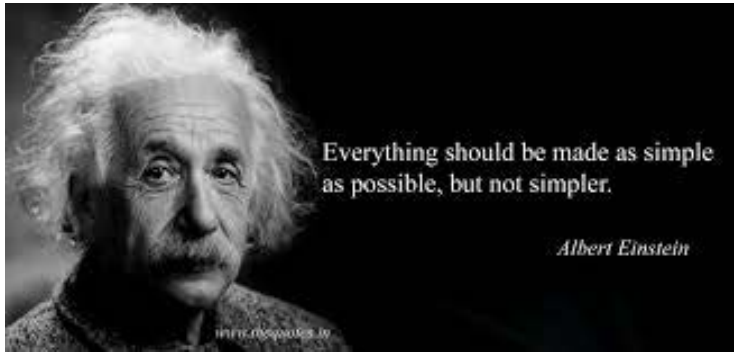
CONCLUSION

We started by looking at the evolution of the kernel, leading to Dijkstra's idea of "layered abstractions" and the KISS approach

Some layers we touched upon:

- The file system, the network, remote file servers.
  A USB drive is not "a file system" yet Linux lets us treat it that way.
- The process abstraction, the runtime environment…

CONCLUSION



Abstraction is a powerful tool for improving specifications, verifying systems and introducing protective boundaries.

- A file system abstracts storage: The device just hold bytes
- We can abstract a system as a VM/container, use this to move applications to a new environment like a cloud.

Yet excessive simplicity through an abstraction that hides performance-critical aspects can harm performance and even create security issues!