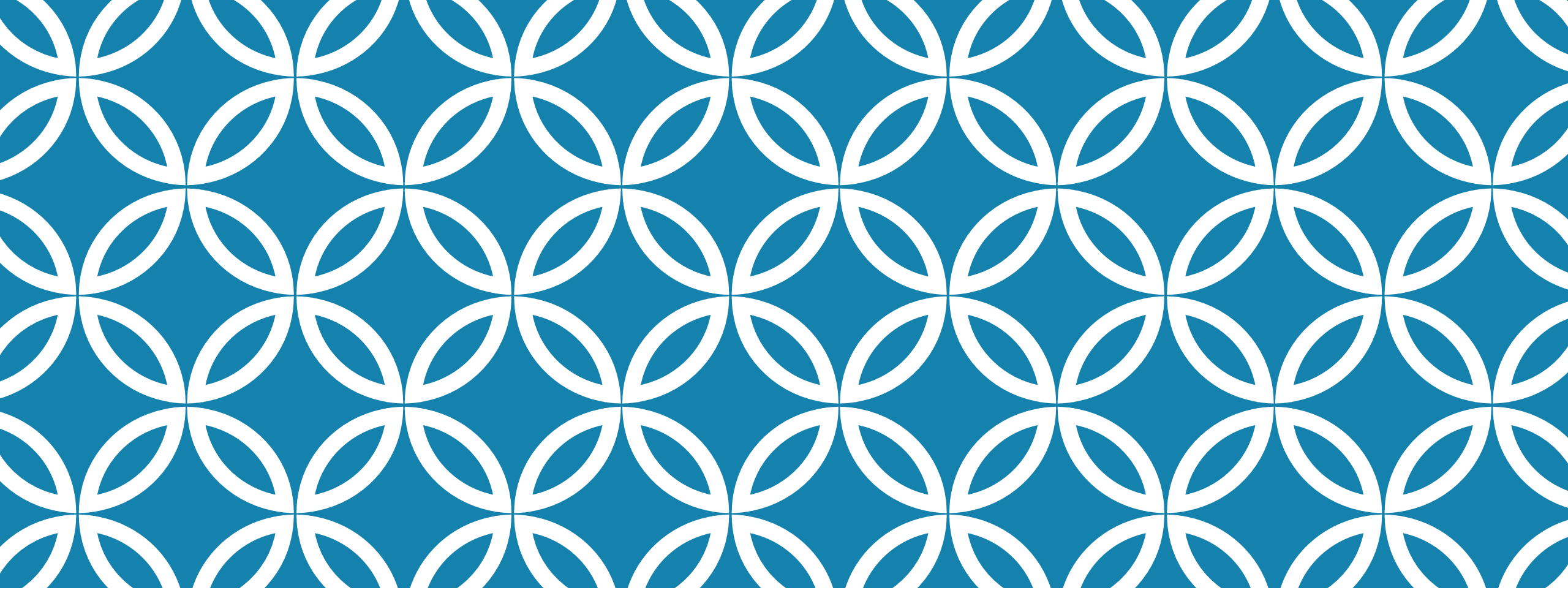


ABSTRACTION VERSUS PERFORMANCE

Professor Ken Birman

CS4414/5416 Lecture 4

TODAY'S LECTURE

Linux files and segmented address spaces are a form of "abstraction". This concept is strongly associated with object-oriented programming, yet in the O/S we see it in a different form.

Today we'll look at the importance of abstractions in systems programming: why they are useful but also how in systems settings we sometimes peek behind the curtain to gain big efficiencies.

Our two examples will be the Linux file system abstraction (POSIX) and Linux virtualization of the address space (or even the computer). lecture 5 will examine segment roles and the reasons the virtual address space is segmented.

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 and at virtualization (docker containers use this concept)

BACK TO THE FUTURE

Back to word-count again… it came down to:

- Doing things **efficiently** (like not creating extra objects)
- Doing things **concurrently** (like using multiple counting threads)
- Doing things **elegantly** (not wasting effort where it doesn't help)

Often, efficiency seemed to require a **implicit** control over things actually happening inside the O/S (file system)

… IN WORD COUNT

In any language, the fastest solutions need to use:

- Multiple threads to do the counting, but also a separate thread to "pre-open" the files. The permissions checking runs concurrently with word counting on previously opened files.
- Sequential reads of the files (or loading the entire file into memory in one shot). Avoids delays while scanning because the data is already in memory when a word counting thread starts to count.

THIS FILE SYSTEM STUFF RAISES A WEIRD PUZZLE

Toto pulls back the curtain

… shouldn't we think of the operating system as a black box? The implementation of a black box is supposed to be hidden!

Is it right to be peeking behind the curtains by coding our word count as if it was sort of a "partner" of the file system?

- And if so, shouldn't the file system abstraction explicitly expose the features we are trying to control and "assist"?

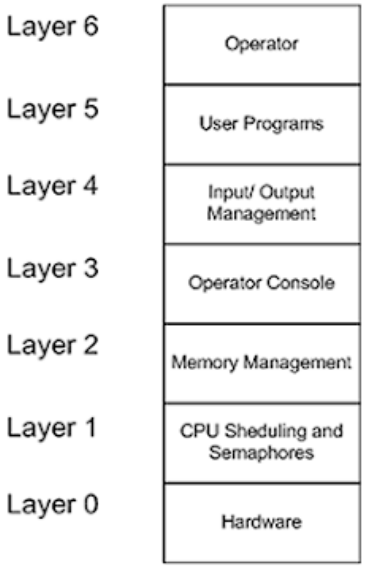
DEFINITION: A SYSTEM ABSTRACTION

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.

In systems, abstractions arise we filter out details to focus on key ideas. The O/S kernel API offers many abstractions.

O/S ABSTRACTIONS

Dates to Edsger Dijkstra and others. Before this, abstractions were mostly used in programming.



… 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

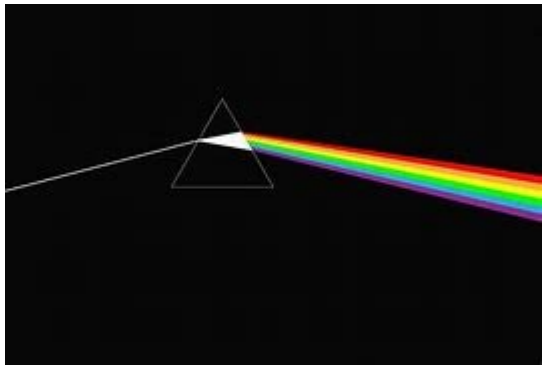
ABSTRACTION BECAME REALLY POPULAR!

Once people started to view O/S functionality as a set of abstraction interfaces, doing so became a huge success

- Instead of a disk being an array of "disk blocks", think of files
- Files can have "types." A file could hold source code, input data, or even a compiled program ready to execute.
- Each process on the computer could have its own "address space" composed of "segments"

But even in the early days, abstraction brought a dark side

THE DARK SIDE



Abstractions can be deceptive, too

- The way you use an API can have cost impacts, yet the API often won't warn you or provide any feedback
- An API can pretend something is true, yet hackers might know it really is an illusion and work around that API
- Some O/S abstractions oversimplify and deny control that you need in order to build the most effective solution

FAST FORWARD TO THE MODERN ERA: POSIX FILE SYSTEM API

In Linux, we 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

TYPICAL POSIX REQUEST SEQUENCE

```
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.

THESE METHODS ARE "SYSTEM CALLS"

You can access them from your code (in any language!) via subroutine calls.

But those subroutines use a special form of exception to "trap" into the O/S, which is where the functionality is really implemented.

The O/S has its own address space and stronger permissions. So it can do things your process would not have been allowed to do.

EXAMPLE: THE POSIX API (1)

Operation	Description
open()	Opens a file and returns a file descriptor.
close()	Closes an open file descriptor.
read()	Reads data from a file descriptor into a buffer.
write()	Writes data from a buffer to a file descriptor.
lseek()	Repositions the file offset of an open file descriptor.
stat() / fstat() / lstat()	Retrieves file metadata (size, permissions, timestamps, etc.).
unlink()	Deletes a file.
rename()	Renames or moves a file or directory.
mkdir()	Creates a new directory.
rmdir()	Removes an empty directory.

EXAMPLE: THE POSIX API (2)

Operation	Description
opendir()	Opens a directory stream for reading entries.
readdir()	Reads the next entry in a directory stream.
closedir()	Closes a directory stream.
chmod()	Changes the permissions of a file or directory.
chown()	Changes the owner and group of a file or directory.
access()	Checks a file's accessibility (e.g., readable, writable, executable).
truncate()	Shrinks or extends a file to a specified size.
fsync()	Flushes all buffered data of a file to disk.
mmap()	"Memory maps" the file into a new segment in memory, returns the base address

HOW LINUX IMPLEMENTS THE ABSTRACTION

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.

THE FILE "DESCRIPTION" RESIDES IN STRUCTURES CALLED "INODES"

Each storage volume has a table of inodes on it. Each inode has an integer number: an index into the table. The O/S keeps track of which volume a given inode lives on, to avoid ambiguity

One inode represents one file. Just like for blocks, inodes can be free or in use and there is a free list of inodes available for use to create new files. (If we run out, the volume is full)

INODES HOLD VARIOUS INFORMATION ABOUT THE FILE

We refer to this as the file "meta-data"

- Count of how many "true" names there are for the file (also called "true links", as on the next slides)
- Access permissions, owner id, "group" id
- Size of the file
- Access time, update time, initial creation time

The inode also has a structure telling us the list of blocks in the file. It looks like a tree and has several formats depending on the size of the file

FILE NAMES AND "LINKS": A DIFFERENT FORM OF META-DATA

We consider **naming** to be separate from other information.

- The file name is used to find a directory in which the name is paired with ("linked with") an inode number.
- The pair (storage-volume-id, inode number) is unique and used to fetch the file meta-data structure.
- In fact, one file can be named with multiple names (aliases, in effect). There are two styles of these extra links

TRUE LINKS, SYMBOLIC LINKS

From the outset, it was possible to just create additional (name, inode-number) references to the identical inode. "True links".

- Example: in any directory, "." is a name that links to the directory itself. This is how ./myFile.cpp works
- And ".." is a link to the parent directory

SYMBOLIC LINKS

But true links were not flexible enough: they only work within a single storage volume (like one SSD), because the inode number implicitly means "in the current storage volume". So, Linux added "symbolic links"

- These are normal files that contain a pathname
- If you search /a/b/c… and "b" is a symbolic link containing a path like /x/y, then Linux jumps to /x/y to search for c.
- In effect, Linux appends the rest of the search path to the symbolic link and restarts its search for the file using this new path.

LINUX FILE SYSTEM – RANDOM ADDITIONAL THINGS TO KNOW

In fact, a file system directory tree can span multiple storage devices, but we will focus on one for the rest of today's lecture.

- Those other devices can be on networked servers
- We talk about the "local" file system and the "global" one.

In the docker containers we provided to you, there is a local file system with various applications, pre-installed, like the C++ compiler and Visual Studio Code editor, gprof and gdb, etc.

LINUX FILE SYSTEM – RANDOM ADDITIONAL THINGS TO KNOW

For this lecture, we also need to know one thing about local disks in Linux: they are almost never very full.

Normally, a disk has LOTs of free blocks and free inodes.

Physical media can suffer from wear and tear. To avoid disk failures, Linux tries to balance its use of blocks – a recently freed inode or block won't be reused for a while.

WHY WOULD THIS MATTER?

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

Rachel has access to his computer, so once he prints the plan he deletes all the copies – including any backup copies. Rachel is able to log in using Chandler's login and password. But the party plan is gone!

PROTECTION OF THE FILE SYSTEM

Rachel herself is not really a computer hacker.

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

…. WHEN HER FRIENDS 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?

…. WHEN HER FRIENDS DELETED "PARTY-PLAN.DOCX"

In

- t

What did Linux not do?

Linux never zeroed the actual <u>contents</u> 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.

And it tends to assume that the developers and users know this, and even know how it really works.

In some sense, Linux assumes that you realize that its abstractions were just for convenience.

RACHEL'S FORENSIC TOOLKIT

Her friend's program opens the disk as a **raw block device**

"RAW BLOCK DEVICE"?

For Linux, any device is initially considered to be a "raw device". A storage device is a "raw block device".

… a list of storage blocks (raw mode also exposes various special control options like rewinding a tape drive).

The raw device is in the Linux folder hierarchy, with a special name. The inode itself has the information about which volume it refers to.

RACHEL'S FORENSIC TOOLKIT

To open a raw device, Rachel normally would need administrative (superuser) privileges.

Rachel only knows Chandler's login and password. The administrative account uses a different login: "root".

Same account as is used for "sudo" in bash.

A COMMON MISTAKE!

Many people use the same password for their normal login and for the special "superuser" login. This is to make the "sudo" command easier to use when in a hurry.

So if Rachel has Chandler's normal login, that same password would often work as a sudo password, too!

RACHEL'S FORENSIC TOOLKIT

Now that the forensic tool can open the raw device, it

- Scans the inode freelist looking for inodes that were freed but haven't been reused for some other purpose yet
- Accesses the corresponding blocks, copying their data
- Because the disk has so many blocks and inodes, they are probably exactly as they were when the file was deleted.

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

IS IT POSSIBLE TO "REALLY" DELETE A FILE?

There are special Linux tools to help you do this. Banks use them… but 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)

WHAT IF YOU DON'T HAVE THE PASSWORD?

Another way to bypass passwords is to remove the SSD disk (which is easy) and plug it into some other computer as an extra disk. Most computers can support extra disks.

The second computer will consider it to be a raw disk… and the relevant root password will be the one on the second computer.

At computer repair shops, they do this all the time!

HACKER TOOLS BYPASS FILE SYSTEM SECURITY

Rachel would not have had permission to do this.

But these tools **aren't following the normal pathway**. They tell you about the file, but don't respect those permissions.

Hacker's often use non-standard paths to take control. They could arise from bugs in Linux or administrative errors by the user

… HOW WOULD RACHEL FIGURE OUT THE FILE NAMES?

The recovered files do have inodes and we probably can see when they were last edited. The tools sort in that order.

Moreover the directory that had the list of file names and inode numbers is also a form of Linux file, with an inode number. We can see when *it* was last modified.

Just lining the times up will probably work!

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

But in this case, Rachel sees a recovered file with:

- A best guess of 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 to her baby shower. ☹

KEY INSIGHT?

The file system abstraction is incredibly valuable.

We depend on it – and on file protection – all the time.

Yet this abstraction is only “skin deep”. The same Linux system can be accessed in other ways that pull back that curtain.

ABSTRACTIONS AREN'T JUST ABOUT FILES

There are many other storage units that can be abstracted as if they were file systems.

And the O/S offers many abstractions that aren't simply about pretending that a storage system is a file system.

Let's consider one more: "virtualization"

VIRTUALIZATION

The term is used for situations where we pretend that some abstraction is the true situation – the abstraction "is reality".

For example, each process has a Linux-managed address space.

This "virtual address space" can be larger than physical memory (you learned how the hardware supports this in CS3410).

VIRTUALIZATION

Dijkstra urged us to accept that the **entire computing experience** is virtual. He sees layers and layers of abstractions.

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

- Docker creates a snapshot of some computer + a set of 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

Suppose a process has a huge virtual address space…

What if it is far larger than physical memory?

- This can happen due to a lack of physical memory
- Linux also supports a command, **limit**, that can limit per-process resources

… the process will begin **thrashing**. The program runs "correctly" yet very slowly. The O/S spends all its time paging in and out.

PROS AND CONS OF VIRTUALIZATION

Moreover…

- 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
- Even if they work perfectly well, there can be overheads

CONTAINER VIRTUALIZATION IS USED IN OUR COURSE

When we give you a docker container, multiple distinct users could run it on the same machine. Yet each sees a 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, files, 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

In cloud settings, virtualization and containers are widely used

You pay much more if you need raw access to your machines.

But this means you probably are sharing your cloud servers with other people: "multi-tenancy". This drives costs down… but cloud operators are constantly worried about hackers!

WHAT WE LEARNED? FIRST, ABOUT ABSTRACTIONS AND SECURITY

Abstractions simplify and allow us to work with a layered computing model. The OS was one of the earliest **large** abstractions, at a time when language features like objects were still emerging.

In programming languages we try to avoid violating abstraction boundaries, but with the OS we view the abstractions as porous.

We often need to think about how they are really implement and work.

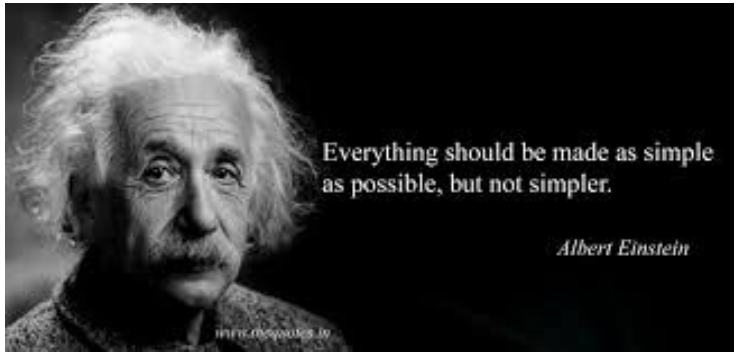
MORE INSIGHTS

We also saw that there can be legitimate reasons to **break** an abstraction boundary.

With a file system, we try to "work with it" to encourage efficiency, prefetching and to optimize caching.

Even file forensic tools are useful for recovering data from a crashed disk.

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!

DID YOU UNDERSTAND TODAY'S LECTURE?

List abstractions you are familiar with on your Windows or Macbook system, focusing on the display.

Why could an abstraction have performance costs you can avoid by understanding how it was implemented? In what situations would this be important?

Why are systems programmers so interested in this duality (abstractions, but also the mechanisms that implement them)?

FILE SYSTEMS AND… VIRTUAL MEMORY?

Word count speed depends on knowing how the file system works. Suppose your LLM or LRM uses a really big pretrained ML model. How might virtualization impact query performance?

What if we use an LLM for document (knowledge) retrieval? In what ways do <u>both</u> kinds of costs matter?

MORE SELF-TESTING QUESTIONS

Suppose that you want to edit /a/b/c and /a/b is a symbolic link containing /x/y.

Will the link reference count in the inode for c include this as an extra reference?

Hint: We didn't answer this in the lecture. But you can easily research it and learn the answer. And knowing it is useful.

MORE SELF-TESTING QUESTIONS

Suppose that you want to edit /a/b/c and /a/b is a symbolic link containing /x/y.

Suppose that someone renames c, and its new name is /a/b/d.

Is it still possible to open the file using the old path, /a/b/c?

MORE SELF-TESTING QUESTIONS

Suppose that you want to edit /a/b/c and it has a **true** link to it from /x/y/c, which is on the same storage volume

Suppose that someone renames /a/b/c, and its new name is /a/b/d. They do not rename the secondary link /x/y/c

Is it still possible to open the file using the old path, /x/y/c?

MORE SELF-TESTING QUESTIONS

Suppose that we replace /a/b/c with a totally new file, d, by removing the old /a/b/c and renaming d as /a/b/c.

*Now suppose that /x/y/c was a **true** link created before this replace operation occurred. We open /x/y/c. Do we see the old content from before the file replace, or the new content from after d was renamed /a/b/c?*

MORE SELF-TESTING QUESTIONS

Suppose that we replace /a/b/c with a totally new file, d, by removing the old /a/b/c and renaming d as /a/b/c.

*Now suppose that /x/y/c was a **symbolic** link created before this replace operation occurred. We open /x/y/c. Do we see the old content from before the file replace, or the new content from after d was renamed /a/b/c?*

MORE SELF-TESTING QUESTIONS

Recall from slide 7 that the reference count field of the inode meta-data doesn't have any count for the number of symbolic links to an inode… it only counts true links.

In a nutshell, they had no choice because tracking symbolic links is hard: they often are on other storage volumes and in a cloud, might even be on different file system servers.

A symbolic link to a deleted file is a kind of dangling pointer. Should this bother us? Could it ever cause confusing behavior?

MORE SELF-TESTING QUESTIONS

Give an example of a situation where the name /user/ken/a definitely exists and can be seen in folder /user/ken, but when you try to open "a" you get the error "File does not exist".

If /user/ken/a is a symbolic link to /user/alicia/b, would it be safe to replace "a" (delete it and then create another file named "a")? What could go wrong? How would you fix this?

OFFLINE DEBATE TOPIC

Names are an abstraction mechanism. But Linux has evolved to have two side by side link semantics under a single namespace abstraction. As a result we need to be very aware of links and symbolic links.

Should large-scale ML storage like the databases used by RAG MLs guarantee a single naming abstraction and a single semantics? What costs would such a decision bring? *Hint: We haven't talked about distributed computing yet, but think about it anyhow!*

ML-SELF TESTING QUESTION

Linux enables the "abstraction" that all memory used by a process is in its address space.

- On NUMA systems we use malloc/free to manage memory in pools, one pool per core. By default memory a thread allocates is in a local pool.
- The GPU programming environment, cuda, has an additional memory pool. cuda malloc allocates from this pool.

Pointers work the same way no matter which pool an object is in. Are there important location-based considerations that might matter in your code?

ML-PERFORMANCE SELF TESTING QUESTION

Many ML systems center on linear algebra. Often matrix multiply is the most important computing step.

Yet pure compute might not be the limiting factor on the critical path.

What **other** factors have we learned about that could shape the performance of matrix multiply?

WE MENTIONED THAT LLMS AND LRMS OFTEN FAVOR A KEY-VALUE STORAGE MODEL

Suppose that a system has vast numbers of K/V objects

Would this put an old-style Linux system under stress, if each key was viewed as a file name?

How might a K/V storage system reduce the number of system calls for accessing huge numbers of objects?

ONE LAST SELF-TEST DEBATE QUESTION

Think about performance. If we optimize word-count for a particular platform our solution won't be equally efficient on different hardware.

Violating abstraction boundaries makes programs less portable. If everything is evolving incredibly quickly, portably is important.

When should we violate abstraction boundaries even so?