

OS Structure: Unix

Emmett Witchel

CS380L

Faux quiz (any 2, 5 min)

UNIX

1. What is a “capability”?
2. What does setuid do? Why is it necessary?
3. What’s the difference between a process and an image?
4. What’s the difference between soft and hard links? Pros/cons?
5. Why does Unix FS opt for **strict** hierarchy? What would get harder if this were relaxed?
6. List some pros/cons for encapsulating devices with a file abstraction
7. Re: “Locks are neither necessary nor sufficient in our environment...”: why did the UNIX authors say this?
8. What is the relationship between unlink and delete in a UNIX file system?

Unix!

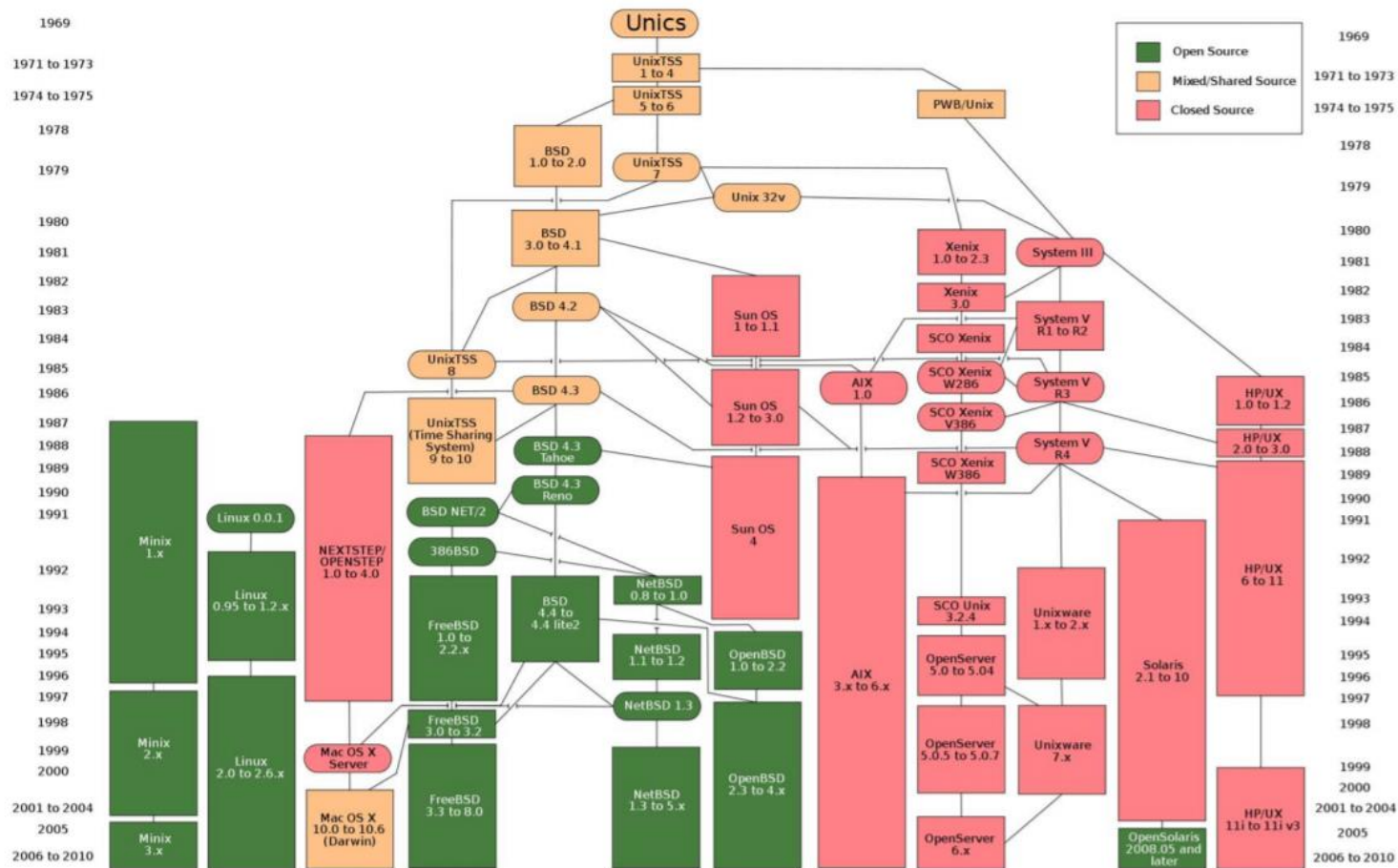
- Minimum functionality and implementation, yet...
- Powerful, and...
- The pieces fit together seamlessly
- Feels obvious doesn't it?
 - This paper outlines most of the basics of modern OSes

Unix across the ages

...demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as \$40,000, and less than two man years were spent on the main system software...

Feature	Unix 1973	Linux 2005	Linux 2016
Min cost system	\$40,000	\$100	\$5 (Raspberry Pi Zero)
Applications	C compiler assembler debugger YACC form letter generator (!?)	C compiler assembler debugger YACC	C compiler assembler debugger YACC
Memory	144KB	2GB	32GB
Memory (min)	50KB	500KB (4MB embedded systems, e.g., ttylinux)	~1MB for microYocto 32-bit, 1.3MB for 64-bit, IoT
Disk	1MB swap, 2.5MB, 40MB	500GB, swap on partition or files	4TB disk or 128GB SSD, swap on partition or files
File names	Up to 14 characters	Up to 255 characters	Up to 255 characters
mkdir	setuid program	user program (mkdir syscall)	user program (mkdir syscall)
File creation	create syscall	O_CREAT flag to open syscall	O_CREAT flag to open syscall
File block size	512B	4KB	4KB
Max file size	1MB	4TB (NTFS is 2TB)	16TB (ext4)
Static lines of code	10K	4.2M	20.6M (4.3 2015-11-01) [12M (Sloccount on 3.16.1)]
Intellectual property	AT&T proprietary	Open source	Open source
Person-years	2	4,528 http://www.dwheeler.com/essays/linux-kernel-cost.html	8,000 (\$1 Trillion)

UNIX family tree



Unix: what were the core new ideas?

- New (at the time) file I/O paradigm
 - Unification of I/O + FS → ***everything*** is a file, unstructured data
 - Hierarchy, relative paths, links, mounting special files, inodes
 - Setuid, fsck
- Process management
- Fork/exec/wait/exit
- Pipes, filters, STDIO
- Process hierarchy/shell

File I/O

Hierarchical name space

- strict hierarchy across directories
- Disallowing multiple links to directories →
 - * Easier search
 - * Easier garbage collection – no cycles
- Engineering “taste”
 - give up a tiny bit of generality for a big savings in complexity
- Eventually augmented with soft links:
 - * soft links don’t increment link count, so dangling is an issue on delete

What are some alternatives to a hierarchical FS?

File IO+Storage: lots of alternatives!

- *Media/interfaces*

- Disk interfaces
 - Word serial
 - CTL-I, SCSI, Parallel ATA
 - Bit serial
 - SDI, Fiber Channel, SATA
- Tape
- NVM: byte-addressable
- All: R/W blocks at offset
- *Foreshadowing: multi-layers of APIs in a real FS*

- *Abstractions*

- Database
- File system
 - Flat
 - Hierarchical
- KV: Get/put
- ...

- *Implementations*

- File system:
 - FAT
 - Log-structured
 - ext
- KV store
- Multi-level FS (e.g. GFS)
- ...

File IO+Storage: lots of alternatives!

- *Media/interfaces*

- Disk interfaces
 - Word serial
 - CTL-I, SCSI, Parallel ATA
 - Bit serial
 - SDI, Fiber Channel, SATA
- Tape
- NVM: byte-addressable
- All: R/W blocks at offset
- *Foreshadowing: multi-layers of APIs in a real FS*

- *Abstractions*

- Database
- File system
 - Flat
 - Hierarchical
- KV: Get/put
- ...

- *Implementations*

- File system:
 - FAT
 - Log-structured
 - ext
- KV store
- Multi-level FS (e.g. GFS)
- ...

Discussion

- Why are there duplicates (e.g. FS)?
- How is a DB different from FS?
- Unix paper mostly about abstraction, less about impl

File I/O

- Untyped data (byte oriented)
 - *“...structure of files controlled by programs which use them, not by the system.”*
- Memory also “untyped”:
 - *“Another important aspect of programming convenience is that there are not “control blocks” with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program’s address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space. (p. 374)”*

File I/O

- Untyped data (byte oriented)
 - *“...structure of files controlled by programs which use them, not by the system.”*
- Memory also “untyped”:
 - *“Another important aspect of programming convenience is that there are not “control blocks” with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program’s address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space. (p. 374)”*

Discussion

- Early Macs had typed FS data
- IBM storage layers have typed data
- **Memory+FS subsystems tightly coupled**
- Others...

File creation/typing: IBM system 360

```
//PDS CRTJ1 JOB SIMOTIME,ACCOUNT,CLASS=1,MSGCLASS=0,NOTIFY=CSIP1,
//          COND=(0,LT)
//* *****
/**          This program is provided by:          *
/**          SimoTime Enterprises, LLC              *
/**          (C) Copyright 1987-2005 All Rights Reserved *
/**          Web Site URL:   http://www.simotime.com   *
/**          e-mail:   helpdesk@simotime.com           *
//* *****
/**
/** Subject: Define a PDS using the IEFBR14 with a DD Statement
/** Author:   SimoTime Enterprises
/** Date:    January 1,1998
/**
/** Technically speaking, IEFBR14 is not a utility program because it
/** does nothing. The name is derived from the fact that it contains
/** two assembler language instruction. The first instruction clears
/** register 15 (which sets the return code to zero) and the second
/** instruction is a BR 14 which performs an immediate return to the
/** operating system.
/**
/** IEFBR14's only purpose is to help meet the requirements that a
/** job must have at least one EXEC statement. The real purpose is to
/** allow the disposition of the DD statement to occur.
/**
/** For example, the following DISP=(NEW,CATLG) will cause the
/** specified DSN (i.e. PDS) to be allocated.
/** Note: a PDS may also be referred to as a library.
/** *****
/**
/**IEFBR14 EXEC PGM=IEFBR14
/**TEMPLIB1 DD DISP=(NEW,CATLG),DSN=SIMOTIME.DEMO.TEMPLIB1,
/**          STORCLAS=MFI,
/**          SPACE=(TRK,(45,15,50)),
/**          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
/**
```

File creation/typing: IBM system 360

```
//PDS CRTJ1 JOB SIMOTIME,ACCOUNT,CLASS=1,MSGCLASS=0,NOTIFY=CSIP1,
//          COND=(0,LT)
//* *****
/**          This program is provided by:          *
/**          SimoTime Enterprises, LLC              *
/**          (C) Copyright 1987-2005 All Rights Reserved *
/**          Web Site URL:   http://www.simotime.com   *
/**          e-mail:   helpdesk@simotime.com           *
//* *****
/** Subject: Define a PDS using the IEFBR14 with a DD Statement
/** Author:   SimoTime Enterprises
/** Date:     January 1,1998
/**
/** Technically speaking, IEFBR14 is not a utility program because it
/** does nothing. The name is derived from the fact that it contains
/** two assembler language instruction. The first instruction clears
/** register 15 (which sets the return code to zero) and the second
/** instruction is a BR 14 which performs an immediate return to the
/** operating system.
/**
/** IEFBR14's only purpose is to help meet the requirements that a
/** job must have at least one EXEC statement. The real purpose is to
/** allow the disposition of the DD statement to occur.
/**
/** For example, the following DISP=(NEW,CATLG) will cause the
/** specified DSN (i.e. PDS) to be allocated.
/** Note: a PDS may also be referred to as a library.
/** *****
/**
/** IEFBR14 EXEC PGM=IEFBR14
/** TEMPLIB1 DD DISP=(NEW,CATLG),DSN=SIMOTIME.DEMO.TEMPLIB1,
/**          STORCLAS=MFI,
/**          SPACE=(TRK,(45,15,50)),
/**          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800,DSORG=PO)
/** *****
```

- Top line: scheduling class, how error messages are reported
- EXEC line: program, IEFBR14 is a label
- TEMPLIB1: new dataset to create
- NEW,CATLG: new dataset should persist after the job.
- DSN: dataset name. (OS360 → 3 level “hierarchy.”)
- STORCLAS: symbolic name for unit+disk-vol for data+metadata (like RECFM, LRECL, SMS).
- SPACE: size of dataset in tracks (blocks, cylinders ok)
 - This case: 45 tracks initially, increment by 15 on grow
 - Disks in IBM land have 512 byte blocks, a device-dependent number of sectors per track (17, 35, 75), a device dependent number of tracks (up to 1024), a device dependent number of heads, and a cylinder which contains as many tracks as there are heads.
- DCB: data control block
- RECFM record format. FB is fixed block records (variable length, undefined length, other options).
- LRECL: logical record length, here 80 characters
- BLKSIZE: size of the data control block (i.e., “inode”).
- * DSORG

File creation: Unix

```
#> echo > /tmp/foo
```

Pros?

Cons?

File I/O

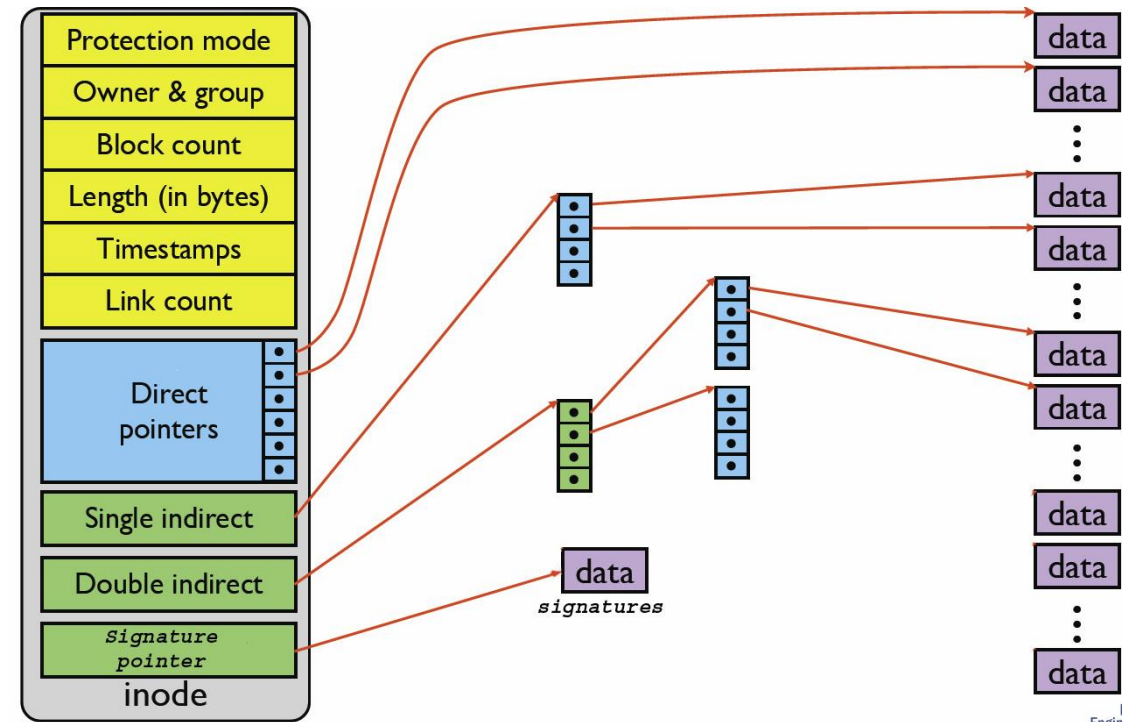
- * Directories are files
 - does this really help anything?

File I/O

- * Directories are files
 - does this really help anything?
- * What is in an inode?

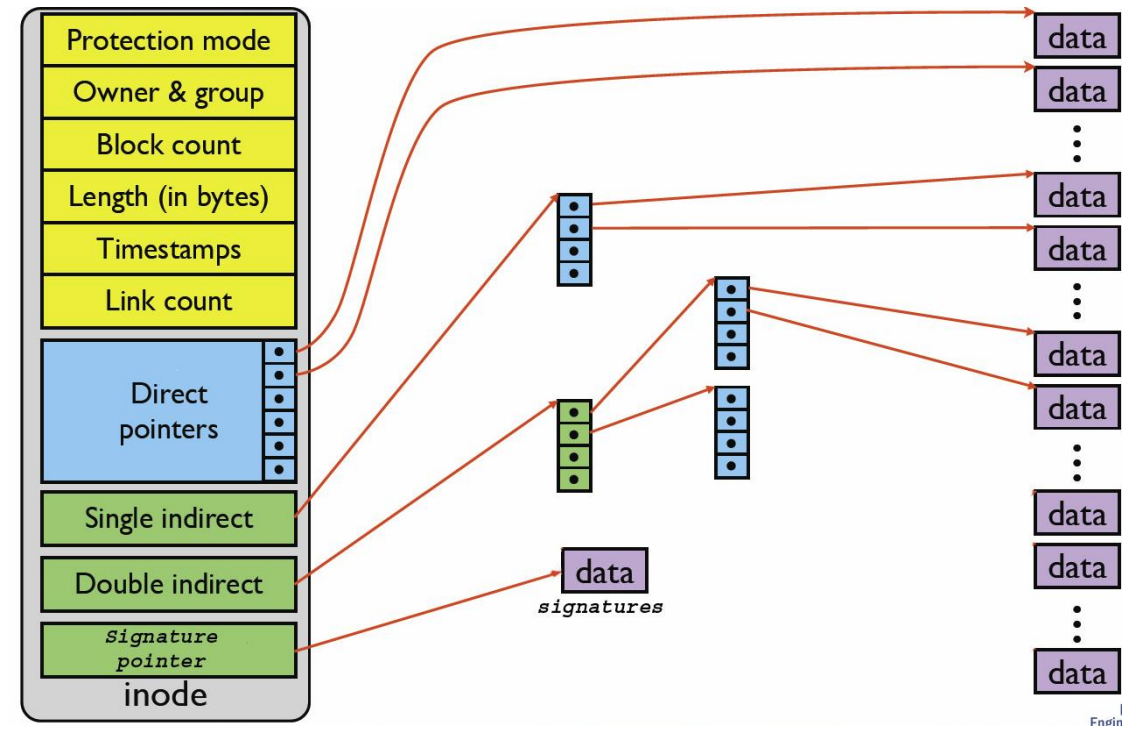
File I/O

- * Directories are files
 - does this really help anything?
- * What is in an inode?



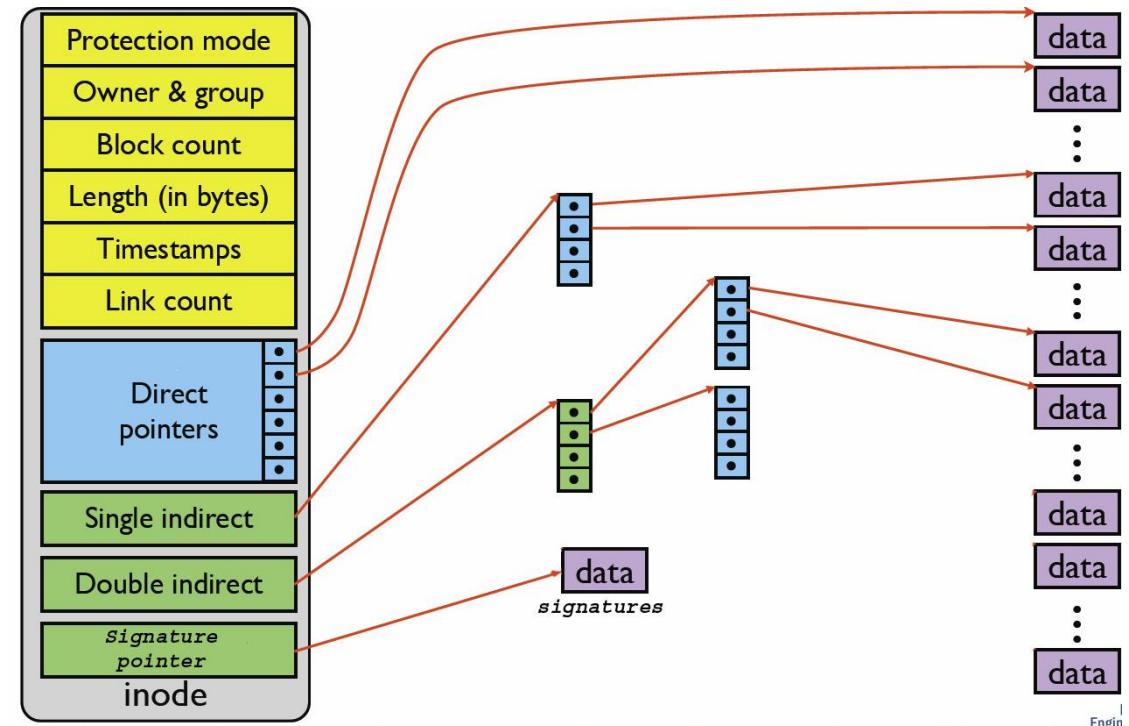
File I/O

- * Directories are files
 - does this really help anything?
- * What is in an inode?
- * What is in a directory?



File I/O

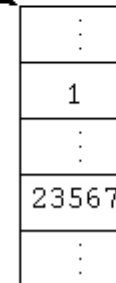
- * Directories are files
 - does this really help anything?
- * What is in an inode?
- * What is in a directory?



directory entry in /dirA

inode	name
12345	name1

inode 12345

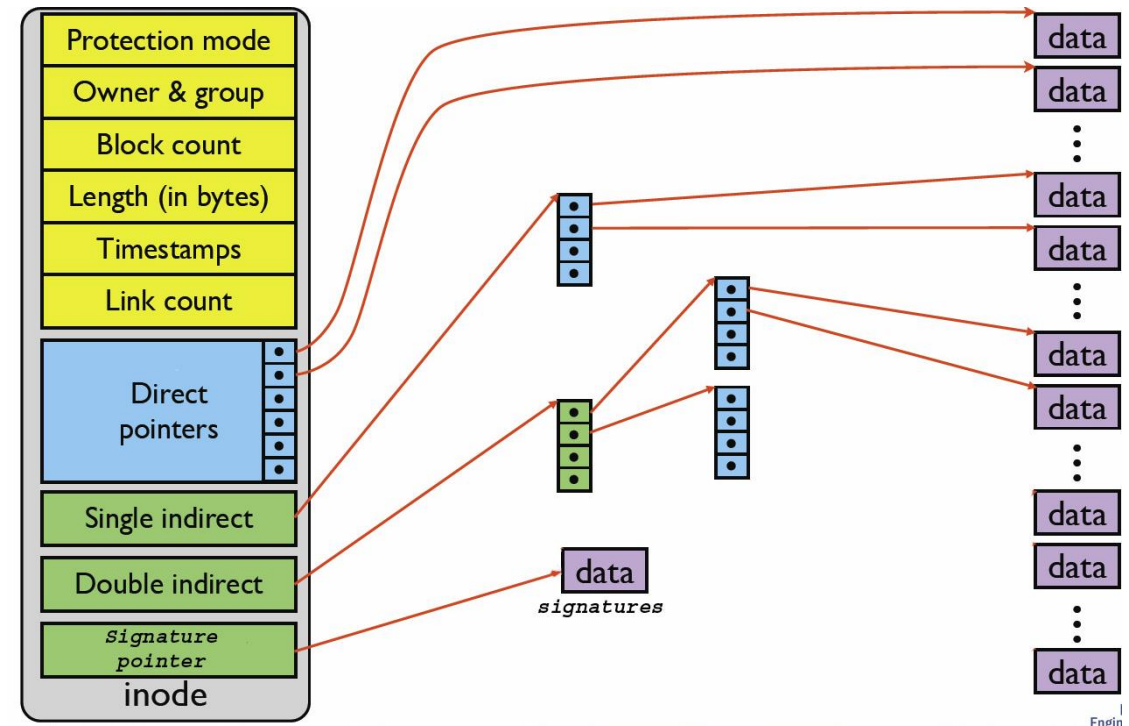


block 23567

"This is the
text in the
file."

File I/O

- * Directories are files
 - does this really help anything?
- * What is in an inode?
- * What is in a directory?
- * How do I find the inumber for file /foo/bar?



directory entry in /dirA

inode	name
12345	name1

inode 12345

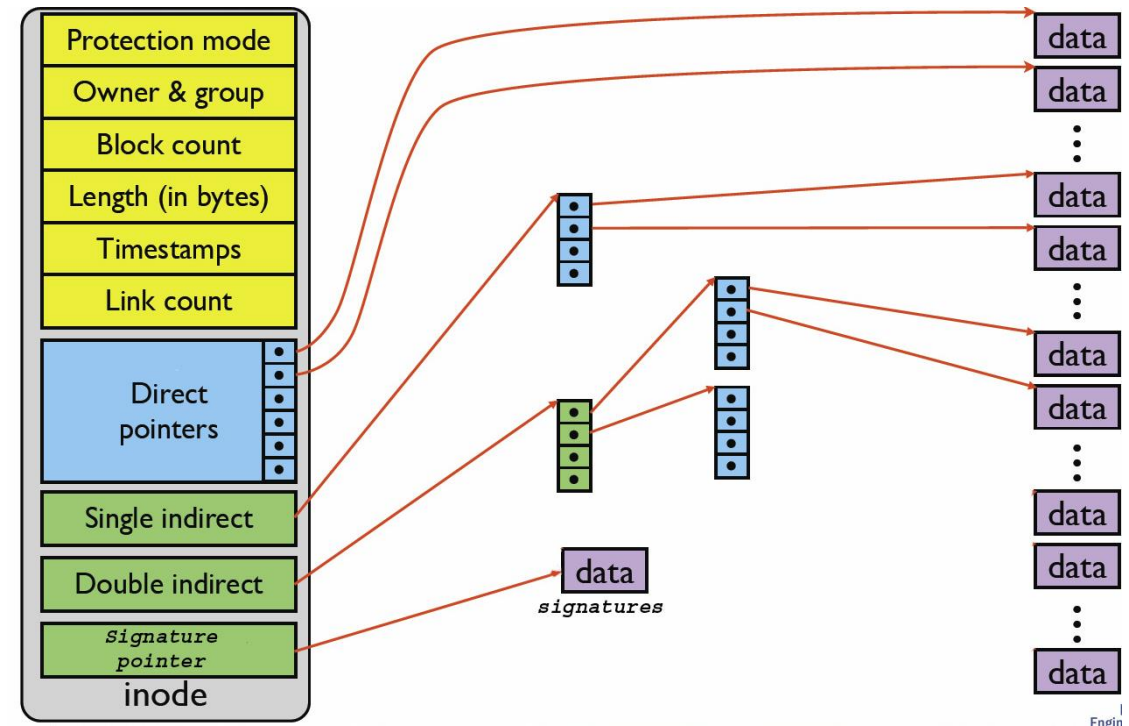
...
1
...
23567
...

block 23567

"This is the
text in the
file."

File I/O

- * Directories are files
 - does this really help anything?
- * What is in an inode?
- * What is in a directory?
- * How do I find the inumber for file /foo/bar?
- * How do I find the inode for inumber 49824?



directory entry in /dirA

inode	name
12345	name1

inode 12345

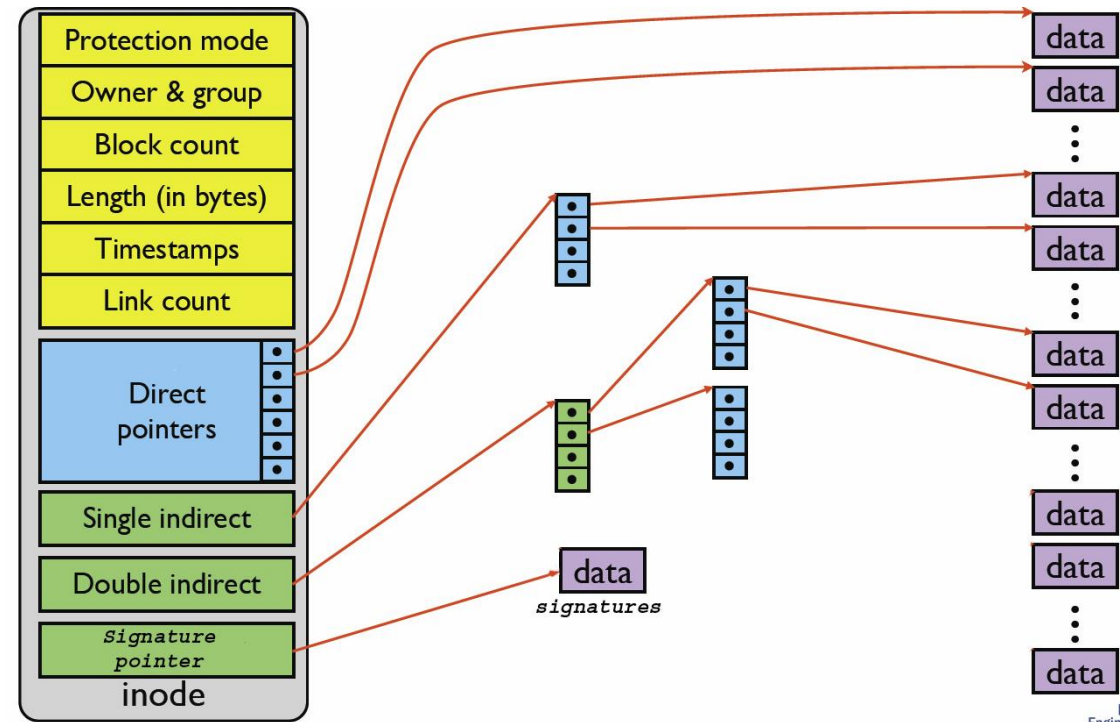
...
1
...
23567
...

block 23567

"This is the
text in the
file."

File I/O

- * Directories are files
 - does this really help anything?
- * What is in an inode?
- * What is in a directory?
- * How do I find the inumber for file /foo/bar?
- * How do I find the inode for inumber 49824?
- * How do I read the third block of file /foo/bar?



directory entry in /dirA

inode	name
12345	name1

inode 12345

...
1
...
23567
...

block 23567

"This is the
text in the
file."

Files in UNIX

- Permissions checks done at open
 - Changes to permissions do not affect open files
- In order to be able to list, read or write a file, you need execute permission on the directories leading to that file (e.g., on a, b, and c for /a/b/c/execute me.py).
- File owner can always chmod, does not need write or execute permission in enclosing directory.
- Return value of read/write. Short reads, short writes, EWOULDBLOCK.
- What are sparse files? Why are they needed?

Files in UNIX

- Files exist independently from directories
 - Open a file in a process (thereby increasing its link count)
 - Unlink file from file system
 - Why do this?
- No guarantee on large (>4KB writes) concurrent writes
 - Reads can see partial writes to a file, even if done with 1 **write** system call

Files in UNIX

- Files exist independently from directories
 - Open a file in a process (thereby increasing its link count)
 - Unlink file from file system
 - Why do this?
 - Because now when process dies for any reason, file disappears
 - NFS maintains this behavior by moving open files to .nfsXXXXXX
- No guarantee on large (>4KB writes) concurrent writes
 - Reads can see partial writes to a file, even if done with 1 **write** system call

Device-independent I/O

Advantages for treating I/O devices as files:

Device-independent I/O

Advantages for treating I/O devices as files:

- 1) file and device I/O are as similar as possible
- 2) file and device names have same syntax/meaning
- 3) special files subject to the same protection mechanisms

Device-independent I/O

Advantages for treating I/O devices as files:

- 1) file and device I/O are as similar as possible
- 2) file and device names have same syntax/meaning
- 3) special files subject to the same protection mechanisms

Do we agree that these advantages are compelling?

Device-independent I/O

Advantages for treating I/O devices as files:

- 1) file and device I/O are as similar as possible
- 2) file and device names have same syntax/meaning
- 3) special files subject to the same protection mechanisms

Do we agree that these advantages are compelling?

- Simple owner/group/other permissions remarkably flexible and useful.

“Pipes are not a completely general mechanism since the pipe must be set up by a common ancestor of the process.”

- Now, named pipes in the file system.
- Though sockets are more general than pipes.

(udev has replaced devfs, which replaced /dev--a “deep” change that took a while to settle down).

Device-independent I/O

Advantages for treating I/O devices as files:

- 1) file and device I/O are as similar as possible
- 2) file and device names have same syntax/meaning
- 3) special files subject to the same protection mechanisms

Do we agree that these advantages are compelling?

- Simple owner/group/other permissions remarkably flexible and useful.

“Pipes are not a completely general mechanism since the pipe must be set up by a common ancestor of the process.”

- Now, named pipes in the file system.
- Though sockets are more general than pipes.

(udev has replaced devfs, which replaced /dev--a “deep” change that took a while to settle down).

Device-independent I/O

Advantages for treating I/O devices as files:

- 1) file and device I/O are as similar as possible
- 2) file and device names have same syntax/meaning
- 3) special files subject to the same protection mechanisms

Do we agree that these advantages are compelling?

- Simple owner/group/other permissions remarkably flexible and useful.

“Pipes are not a completely general mechanism since the pipe must be set up by a common ancestor of the process.”

- Now, named pipes in the file system.
- Though sockets are more general than pipes.

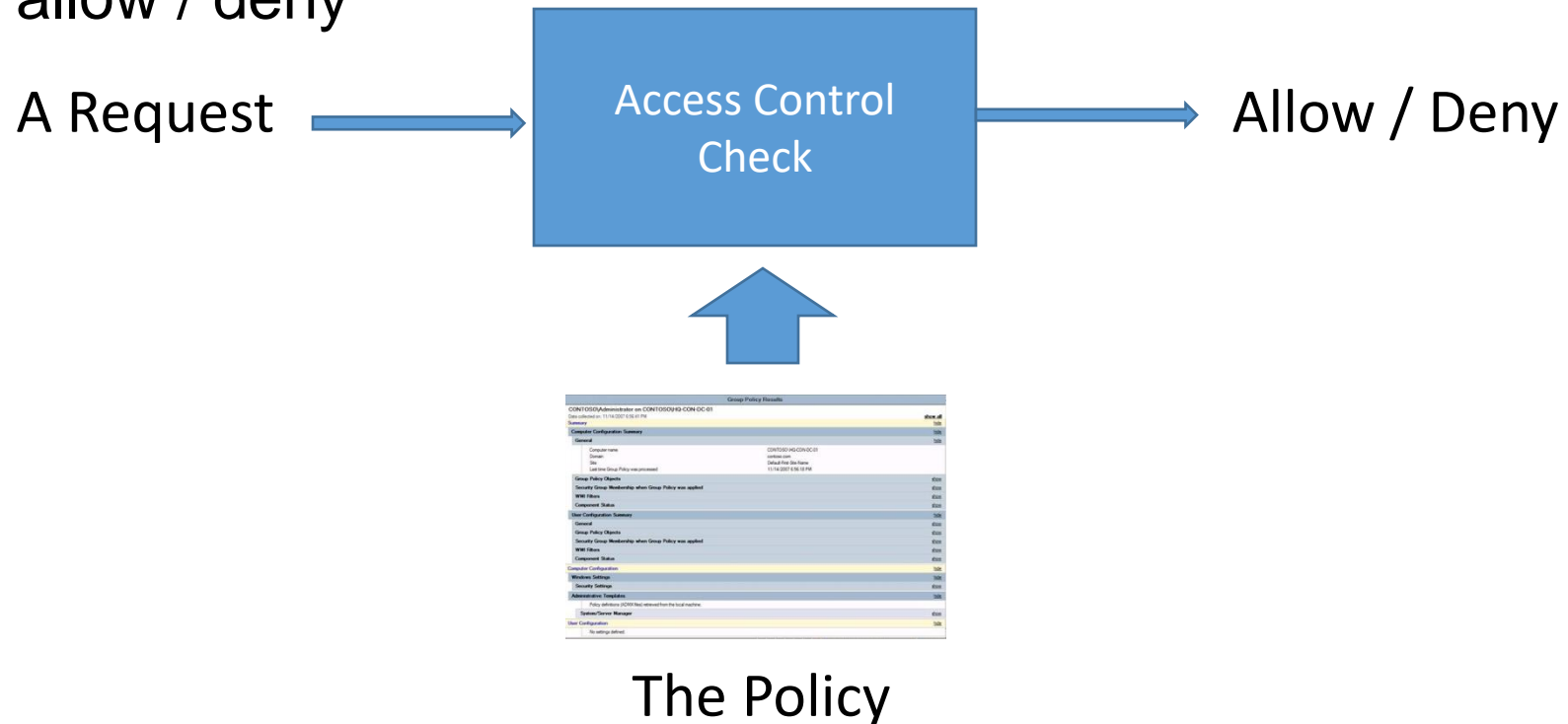
(udev has replaced devfs, which replaced /dev--a “deep” change that took a while to settle down).

File descriptors

- Filter programs do not know the name of input or output files.
- capability: “Handle with access rights”
 - *Wait...what's a capability?*

Access Control Check

- Input: access request, policy
- Output: access control decision based on policy
 - allow / deny



Access Control Matrix

- Representation/definition of permissible operations in a system

	Objects				
Subjects	user ₁	user ₂	user ₃	file ₁	file ₂
user ₁		Send msg		RW	R
user ₂	Send msg				RW
user ₃	Set passwd	Set passwd	Set passwd		R

- **Subjects**: users, processes, groups, etc.
- **Objects**: other users/processes, files, memory objects, etc.
- **Privileges/rights**: depends on object
 - for file: read, write, execute, etc.

Access Control Matrix

- Representation/definition of permissible operations in a system

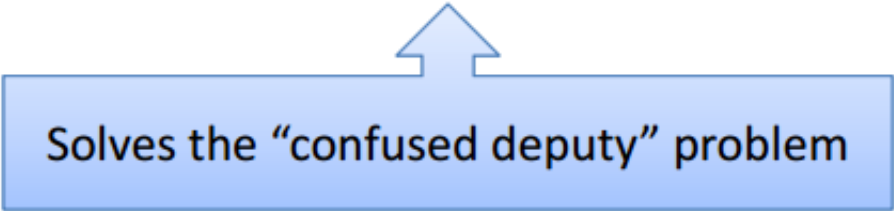
	Objects				
Subjects	user ₁	user ₂	user ₃	file ₁	file ₂
user ₁		Send msg		RW	R
user ₂	Send msg				
user ₃	Set passwd	Set passwd	Set passwd		

- **Subjects**: users, processes, groups, etc.
- **Objects**: other users/processes, files, memory o
- **Privileges/rights**: depends on object
 - for file: read, write, execute, etc.

- Dynamic data, frequent changes
- Very sparse, repeated entries
- Impractical to store explicitly
- Most common mechanisms:
 - Access control list: stores a column (who can access this)
 - Capabilities: store a row (what this can access)

Capabilities

- Main advantage of capabilities is **fine-grained access control**
 - ⇒ Easy to provide access to specific subjects
 - ⇒ Easy to delegate permissions to others
- A cap presents prima facie evidence of **right to access**
 - Think of it as a key
 - Any representation must protect capabilities against forgery
- Consists of **object identifier** and a set of **access rights**
 - Implies *object naming*



Solves the “confused deputy” problem

File descriptors: brilliant

- Filter programs do not know the name of input or output files.
- capability: “Handle with access rights”
- How many file descriptors can you open?
- File descriptors are just integers, why can't a user program forge one?

setuid: rights amplification

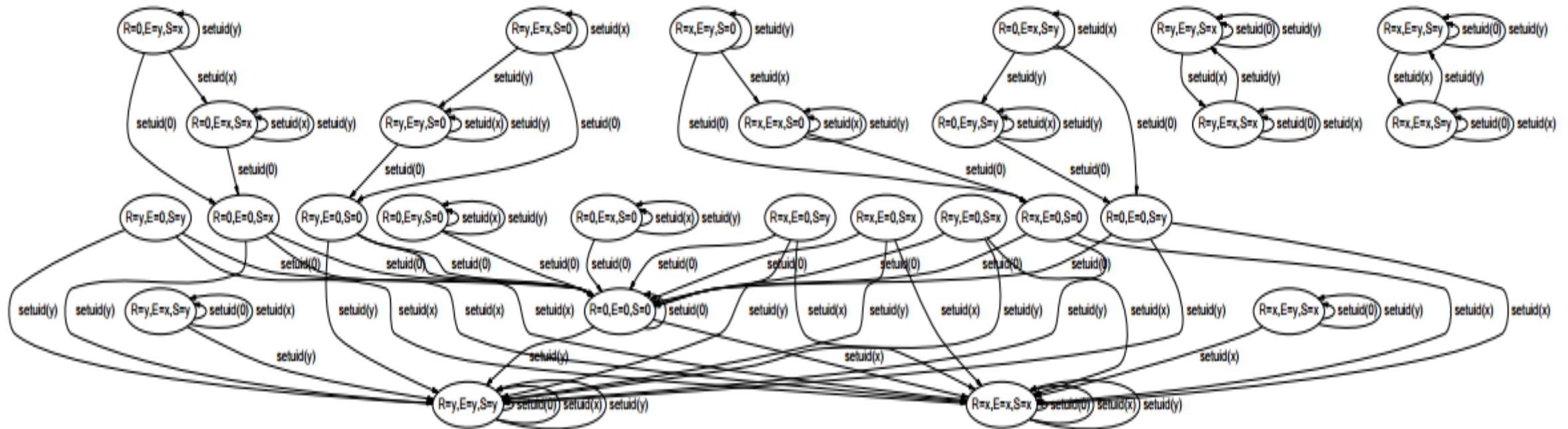
Coarse-grained sharing – “execute a program as someone else”

- v. Multics rings – fine grained sharing – “execute a procedure as someone else”
- Minimalist: Need to have process == principal
- add setuid to that basic mechanism rather than invent orthogonal model
- Limits to how fine-grained we can (correctly/conveniently) divide programs?

Also “Make common case fast. Make uncommon case correct.” Common case is procedure call to same code base. How much extra mechanism do you want (complexity, cost, speed penalty in common case) for uncommon case?

setuid: a cautionary tale?

Each process has three user IDs: the real user ID (real uid, or ruid), the effective user ID (effective uid, or euid), and the saved user ID (saved uid, or suid). The real uid identifies the owner of the process, the effective uid is used in most access control decisions, and the saved uid stores a previous user ID so that it can be restored later. Similarly, a process has three group IDs: the real group ID, the effective group ID, and the saved group ID.



(a) An FSA describing *setuid* in Linux 2.4.18

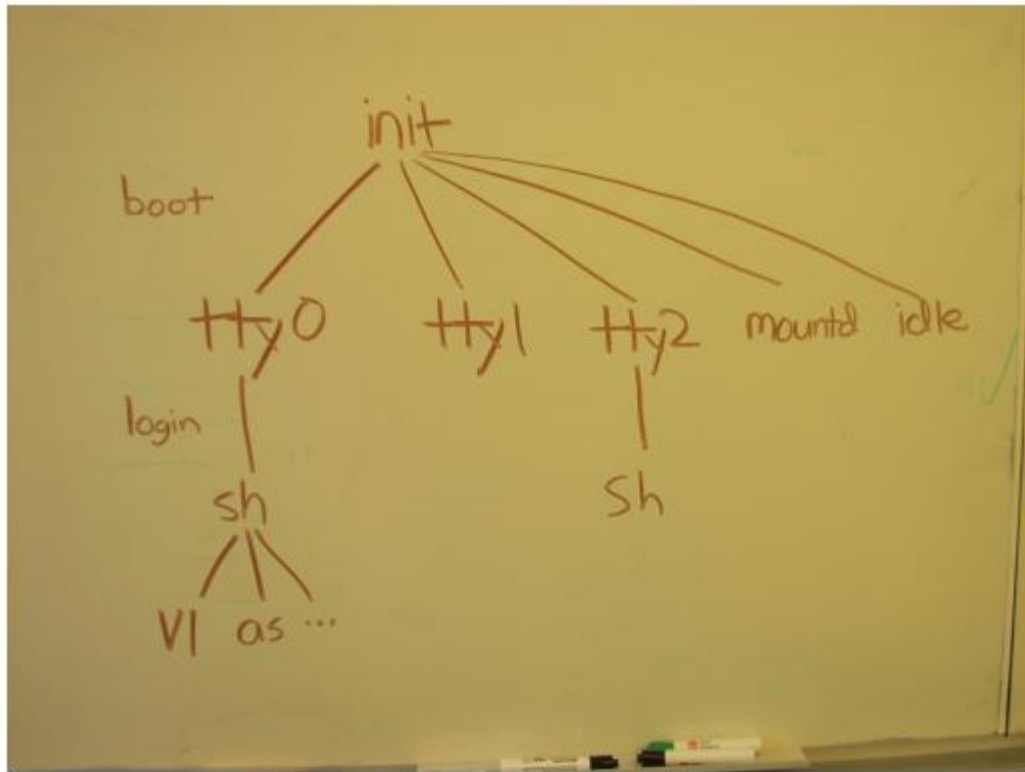
Mount

- Removable storage; expand storage;
- Engineering simplification: No cross-volume links allowed

Process management: “primitives not solutions”

- Process is an executing program (or image). Code, heap and stack.
- Building blocks
 - Fork, exec, wait
 - File I/O structure
 - Fork'd child shares parent's open files
 - → pipe, std I/O, redirection, filters
 - Coarse grained sharing of programs: `cat foo | grep "bar" | sort | tail -10`
 - Shell, background execution
 - Stdin, stdout: enables redirection and pipelines.
 - Shell: a good programming language?
 - Elegant process structure enables communication
 - Signals as another form of inter-process communication.

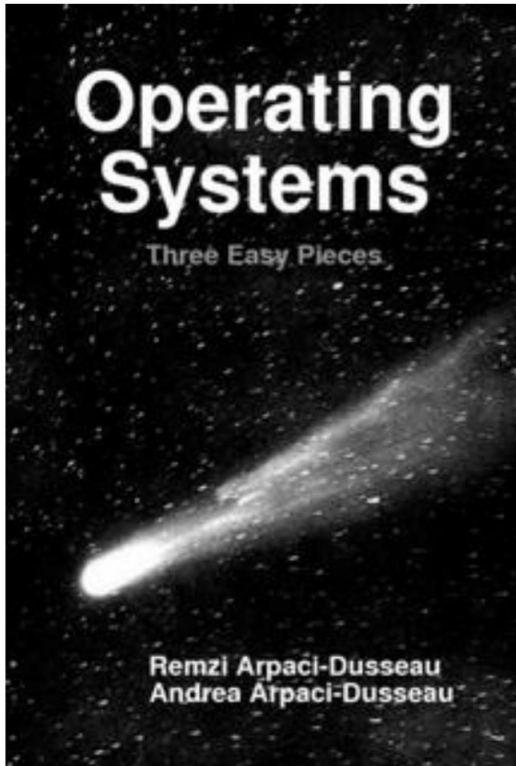
Process Hierarchy



```
shell(){
    while(got = read(STDIN, buffer, ...)){
        command, args, redirection, bg = parse(buffer);
        if(pid = fork()){
            /* I am the child */
            if(redirection){
                close stdin and/or stdout and open specified files
            }
            exec(command, args);
            /* Only reached if error on exec */
            exit(-1);
        }
        /* I am the parent */
        if(!bg && donePid != pid){
            donePid = wait();
        }
    }
}
```

Fork: Not Without Controversy

Fork: Not Without Controversy



Interlude: Process API

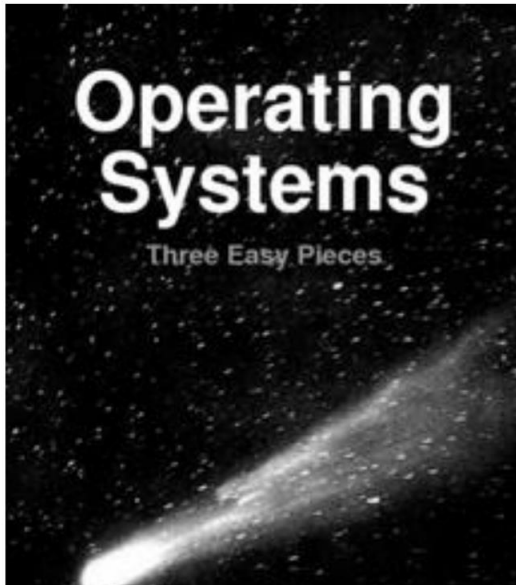
5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "Get it right. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful. Here, the UNIX designers simply got it right. And because Lampson so often "got it right", we name the law in his honor.

Fork: Not Without Controversy



5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

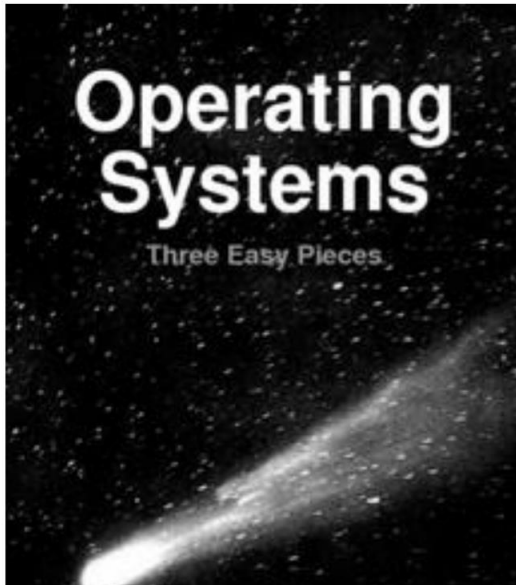
TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "Get it right. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots on; however, the combination immensely powerful. Here, the because Lampson so often "got

Why do people like fork?

- Simple: no parameters!
 - cf. Win32 `CreateProcess`
- Elegant: fork is orthogonal to exec
- System calls that a process uses on itself also initialise a child
 - e.g. shell modifies FDs prior to exec
- It eased concurrency
 - Especially in the days before threads and async I/O

Fork: Not Without Controversy



5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the ab
of interesting features t

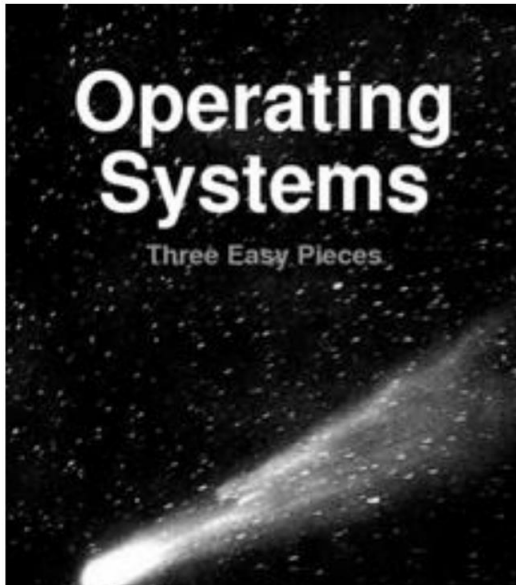
TIP: GE
As Lampson states in
Design" [L83], "Get it
stitute for getting it rig
and when you do, it

Why do people like fork?

- Simple: no parameters!
 - cf. Win32 `CreateProcess`
- Elegant: fork is orthogonal to `exec`
- System calls that a process uses on itself also initi
 - e.g. shell modifies FDs prior to `exec`
- It eased concurrency
 - Especially in the days before threads and async I/O

```
BOOL CreateProcess(  
    LPCSTR                lpApplicationName,  
    LPSTR                 lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL                  bInheritHandles,  
    DWORD                 dwCreationFlags,  
    LPVOID                lpEnvironment,  
    LPCSTR                 lpCurrentDirectory,  
    LPSTARTUPINFOA         lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```


Fork: Not Without Controversy



5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

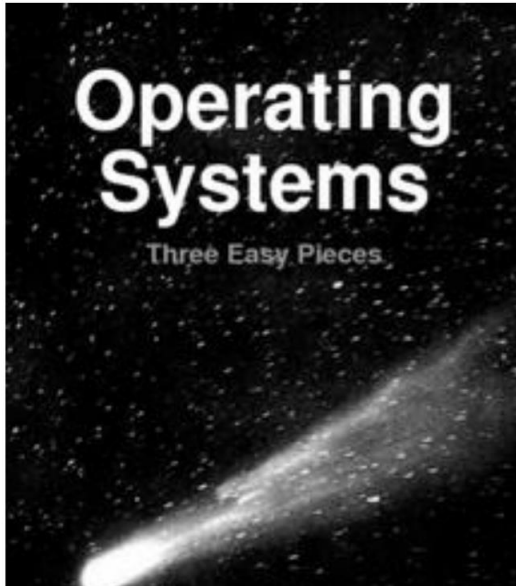
TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "Get it right. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots on; however, the combination immensely powerful. Here, the because Lampson so often "got

Why do people like fork?

- Simple: no parameters!
 - cf. Win32 `CreateProcess`
- Elegant: fork is orthogonal to exec
- System calls that a process uses on itself also initialise a child
 - e.g. shell modifies FDs prior to exec
- It eased concurrency
 - Especially in the days before threads and async I/O

Fork: Not Without Controversy



5.4 Why? Motivation

Of course, or such an odd interprocess? Well, a essential in building the call to fork environment of of interesting features

As Lampson states in "Design" [L83], "substitute for getting and when you

Fork: actual motivation:

- For implementation expedience [Ritchie, 1979]
- fork was 27 lines of PDP-7 assembly
- One process resident at a time
- Copy parent's memory out to swap
- Continue running child
- exec didn't exist – it was part of the shell
- Would have been more work to combine them

Fork was a hack!

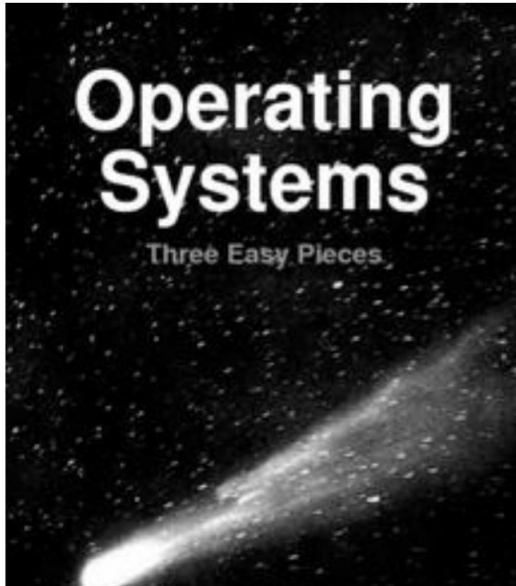
- Fork is not an inspired design, but an accident of history
- Only Unix implemented it this way
- We may be stuck with fork for a long time to come
- But, let's not pretend that it's still a good idea today!

Why do people like fork?

- Simple: no parameters!
 - cf. Win32 CreateProcess
- Elegant: fork is orthogonal to exec
- System calls that a process uses on itself also initialise a child
 - e.g. shell modifies FDs prior to exec
- It eased concurrency
 - Especially in the days before threads and async I/O

because Lampson so often "got

Fork: Not Without Controversy



5.4 Why? Motivation

Of course, or such an odd inter process? Well, a essential in building the call to fork environment of of interesting features

As Lampson states in "Design" [L83], "substitute for getting and when you

Fork: actual motivation:

- For implementation expedience [Ritchie, 1979]
- fork was 27 lines of PDP-7 assembly
- One process resident at a time
- Copy parent's memory out to swap
- Continue running child
- exec didn't exist – it was part of the shell
- Would have been more work to combine them

Fork was a hack!

- Fork is not an inspired design, but an accident of history
- Only Unix implemented it this way
- We may be stuck with fork for a long time to come
- But, let's not pretend that it's still a good idea today!

Why do people like fork?

- Simple: no parameters!
 - cf. Win32 CreateProcess
- Elegant: fork is orthogonal to exec
- System calls that a process uses on itself also initialise a child
 - e.g. shell modifies FDs prior to exec
- It eased concurrency
 - Especially in the days before threads and async I/O

because Lampson so often "got

Thanks to Andrew Baumann et al. "a fork() in the road '19"