Lecture 6 1/25/16

OS Organization Revisited

Processes and files

these are all "fakes"

process & file has

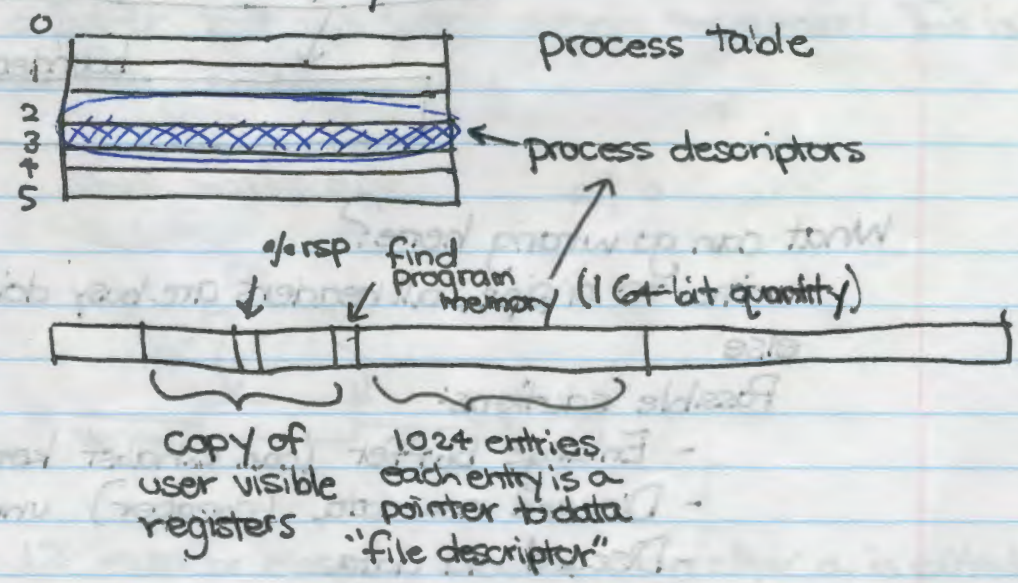
resources we need:

- ALU ⟹ ALU
- registers (%rsp, %rip, etc.) ← registers *
- memory (RAM) ← physical memory **
- I/O ← emulated/simulated

pretend — on a real

* when running
** when running, accessing

Kernel Memory



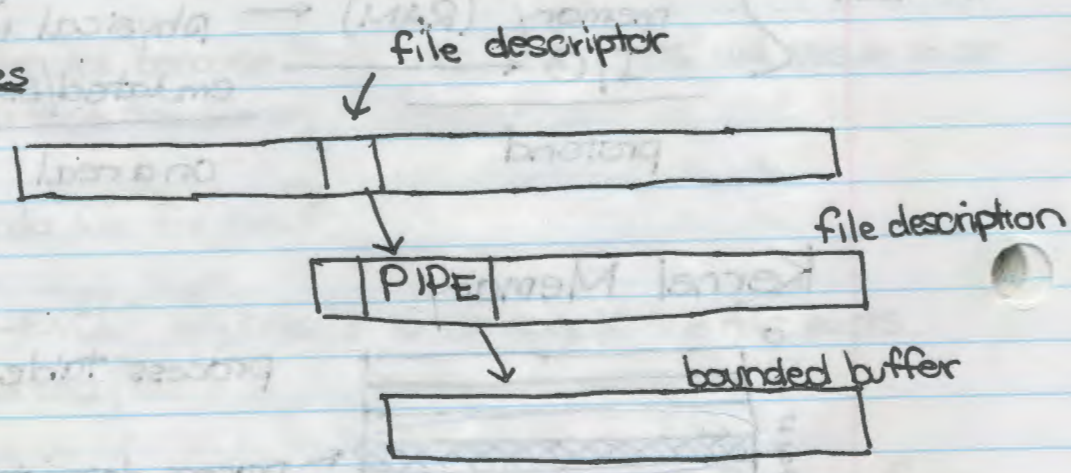
rti or sysexit → pull out these registers and put them in actual registers

ALU is not saved, because it doesn't have a state; it simply performs arithmetic. It's easier to throw it away and start from scratch than to try to remember it.

handles:
Linux/Unix 'int' is a handle for an open file
Other OS struct filedes *
'pointer'

int
+ greater ability to move things around
- won't be distinguished by regular integers

Pipes



What can go wrong here?
write to a pipe, all readers are busy doing something else
Possible solutions:
- Enlarge buffer (can exhaust kernel mem)
- Discard new data (cheaper) unreliable
- Discard old data
- Suspend the writer ✓ Linux standard

write to a pipe, but there are no readers
- suspend would cause writer to hang forever!
- kill the writer (SIGPIPE)
(optionally, ignore SIGPIPE): (writes fail w/ errno == ESPIPE)

read from empty pipe
→ hang
read from empty pipe, no writers!
read returns 0 ← EOF

Aside: named pipes

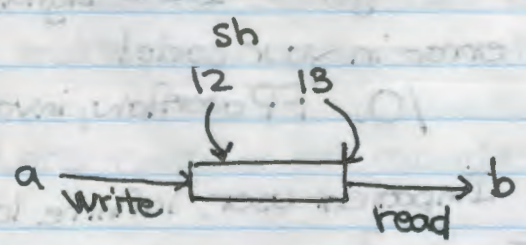
```
$mkfifo /tmp/pipe
```

Anyone can read from this pipe whenever they feel like it!

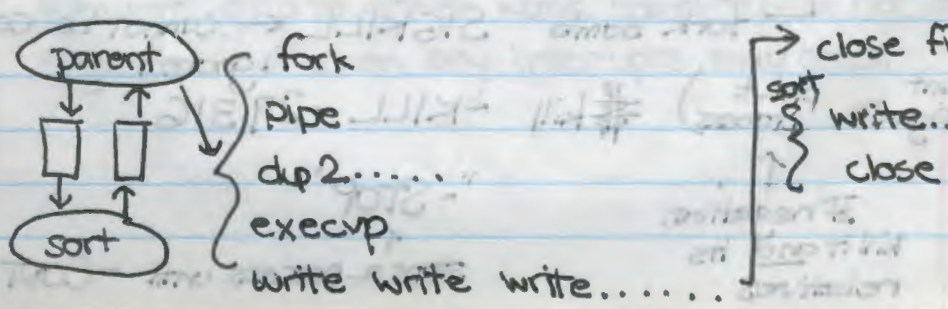
```
$cat /tmp/pipe > out &
$echo Hello > /tmp/pipe
```

What if we make a pipe that is never used? We have a buffer that will waste system resources! This is a <u>file descriptor leak</u>.

Shell makes pipe a | b



12 must be closed, because even after a is exited, 12 will remain and b will hang forever.



This pattern only works because sort only begins writing after it has read in all the data!

With sed, we could encounter double full pipes, which results in a double <u>deadlock</u> where both parent and sed hang!

Consider:

```
rm bigfile
grep interesting
 < bigfile;
```

This will work! ← data removed hear

fds across files at a lower level than the file names
pipe() already does this.

The file is only removed when all of the file descriptors looking at it are gone!

<u>Signals</u>:
Why?

- asynchronous I/O
  - aread()
    - get a SIGIO signal later.
- error in your code!
  - /0, FP overflow, invalid instr
- Impatient user, infinite loop SIGINT
- Impending power outage. (SIGPWR)
- send signal when a process is finished SIGCHILD
- User went away SIGHUP
- fork bomb SIGKILL ← cannot be caught or ignored

int kill (int signal#, pid_t process) #kill -KILL -29316

↑ If negative, kill it <u>and</u> its relatives

"-STOP"
↑ stops a process until -CONT is called for it

- runaway programs ← SIGKILL on previous page
- alarm(20); SIGALARM

Uses for Signals

- Break out of a loop
- Suspend or kill a process
- Timeouts
- important, unusual, unexpected events

Receiving Signals

sig_handler_t Signal(int , sighandler_t)

↑ old handler ↑ signal# ↑ new handler

sighandler_t

void (*sighandler_t)(int)

pointer to a function that takes an int and returns a void

```
#include <signal.h>
    SIGINT = some constant
```

SIG#

Notice that signals can be executed at <u>any</u> time, no matter how inconvenient it's.

Let's consider an example:

gzip.c

```
$gzip foo
foo.gz
```

We want gzip to either work completely or not at all, regardless of when any interrupts may occur.

```
fd = open("foo", O_RDONLY); ← signal(SIGINT, cleanup)
fo = open("foo.gz", O_RDONLY);
while (compress(fd,fo)) ← ~~If interrupted~~ If interrupted in
    continue;                here, then we will be stuck with
close(fo);                   a partially compressed result
unlink("foo");

static void cleanup(int sig)
{
    unlink("foo.gz")
    _exit(1);
}
```

Races: - remove a foo.gz that is already there
: Break the foo file, then run cleanup even though it had worked properly.

1/27/16
James W.

Lecture 7

gzip has
critical section

tell signal handler not to remove compressed remove uncompressed file

~~signal(SIGINT, SIG_DFL)~~ * ok_to_remove = false;
* unlink("foo.gz")

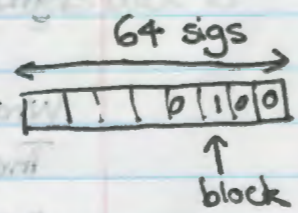
```
void handle_interrupt (int sig).
{
    if (ok_to_remove)
        unlink("foo.gz");
}
```

Blocking Signals

```
int pthread_sigmask (
    int how,
    const sigset_t *restrict set,
    sigset_t *restrict oset)
```

SIG_BLOCK
SIG_UNBLOCK
SIG_SETMASK

new set of signals to block




```
    sigset_t ss;
    sigemptyset(&ss);
    sigaddset(&ss, SIGINT);
*   pthread_sigmask (SIG_BLOCK, &ss, Ø);
    pthread_sigmask (SIG_UNBLOCK, &ss, Ø);
```

Threads and Signals

Threads

+ less CPU time
+ less overhead
+ less memory
+ performance

- less reliability
- dangerous

This is dangerous enough: but now we have signals!

multithreaded program:

```
signal(SIGINT, handle_interrupt);
```

affects all threads! Linux picks a thread at random

Important note:

pthread_sigmask : affects only current thread
sigprocmask: can affect the whole program

Why? We want to give individual threads the ability to ignore signals.

What goes into signal handlers must be very carefully placed. They must be asynchronous signal safe functions (otherwise we run the risk of signals running on functions called by the signal handler).

_exit ~~exit~~
write ~~printf~~
~~malloc~~

In fact, the only safe safe ways to go about it:

sig_atomic_t volatile glob //Set this in handlers

What's the problem? This requires us to change our code to pull a variable every time!

```
void handle_interrupt (int sig)
{
    if (pthread_self == stgmgr)
    {
        really handle interrupt();
    }
    else
    {
        pthread_kill(SIGINT, stgmgr);
        // Pass signal onwards to the manager
    }
}
```

will be initialized to be a thread, perhaps the main process

Remember that after this call returns the thread goes back to whatever it was doing.

A read can:

- Succeed
- fail
- be interrupted and return with an error EINTR

(CPU) <u>Scheduling</u> (has to be done right) (taken for granted) ← irl

Scale Issues : • lots of threads • lots of thread types
• lots of time

long term scheduler for an OS
- which processes are going to be admitted for the system?

fork() < 0 ← scheduler rejected it

medium term
- Which processes out of the current ones should we have reside in RAM?

short term
- Which threads get a CPU?

Some Scheduling Ideas

Different thread types — multilevel priority system
system jobs (password check)
interactive jobs
batch jobs
~~student jobs~~.....

Priorities
external (set by user)
internal (set by OS) ← system

system call:
int nice (int niceness);

~~Priority~~ niceness
1. Most important
2. Less important
3. Least important ← nicest :)

1 < 3?

priority vals generally -19 to 19
nice(10)
niceness begins at 0.
Nice guys finish last. :(

10 is default niceness
↑ when the program is called

nice.c

```
int main(int arg, char **argv)
{
    nice()
    execvp();
    //error message here if failed
}
```

Realtime schedulers (for apps where performance is part of the correctness criteria)

- Hard - deadlines are unchangable and must be met
  - we want predictability; it trumps performance
  - we are willing to give up performance so long as it meets the deadline
    - Thus, caching in this case is <u>bad</u>.
- Soft - has deadlines can be missed but at a quality cost

Possible Schedulers

- earliest deadline first
  - +simple
  - +efficient
  - - one stream can monopolize
- rate monotonic scheduling
  - Give each stream a % of CPU time

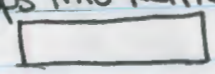
Scheduling =

set of policies (theory)	+	set of dispatch mechanisms (practical)

Dispatch Mechanisms
for CPU scheduling
for OS
system call

Cooperative Multitasking

close(29);
SYSENTER
↓
drops into kernel



SYSEXIT

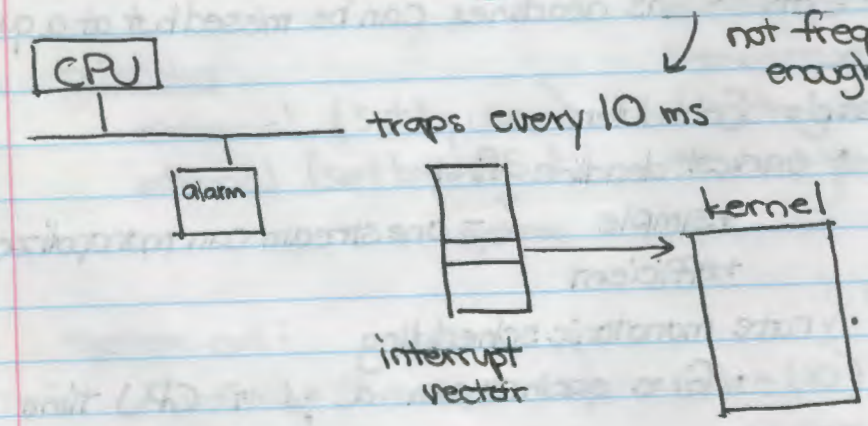
+ No need to write new code
− If we have code that does a lot w/o making system calls, we won't pass control.

The assumption with this mechanism is that every thread issues a syscall every so often

yield() or schedule() //Does nothing except maybe //cause transfer of //control

Preemptive Multitasking

too frequent → too much overhead
not frequent enough → too slow to respond to signals!



If we have
- busy waiting — while(!ready) continue;
- polling — ready must yield if not ready
- blocking — check if device is ready wait_until_ready

Midterm Review

1) Name some ways in which the processor can transition from user to kernel mode.

- external interrupt ← SIG_TIMR (clock that triggers every 10 ms) keyboard
- system calls (internal interrupt) ↑ maybe a read. or some other memory access.
- page fault
- privileged instructions ↑ int, divide by 0, etc.

2) Suppose INT doesn't trap when executing in user mode. How ~~w~~ can we implement system calls?

Need two things:

- switch from user to kernel mode.
- A way to indicate the interrupt code + arguments

Let's try a timer interrupt

```
int main ()
{
  while (true)
  {
    asm (jump •);
  }
```

jump to its current ip.

// = a and set up the arguments to try and run.

modify kernel code to recognize this situation as an attempted syscall (in a handler). make sure that this method has soft modularity.

Another approach: Reserve some kernel memory for system calls.

Load invalid memory ← in user mode

Kernel traps

Meanwhile, syscall handling is inside the kernel mem and ready to go.

3) In WeensyOS1 we implement getpid as a system call. What if we had done it as an ordinary function?

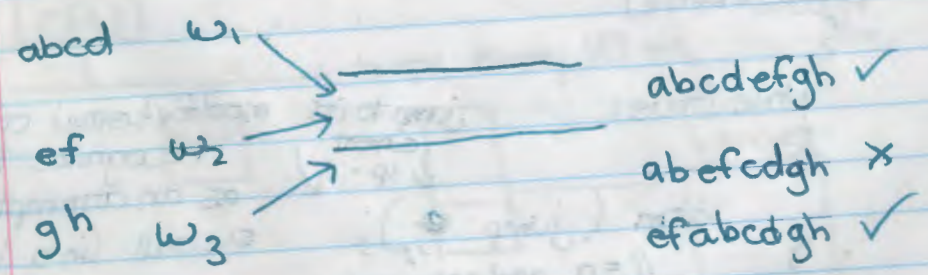
<u>Pros</u> + Fast

<u>Cons</u> When we create a ~~child~~ process, it's pid has to be in the process's address space, otherwise the process won't be able to see the pid value. But, the user can now <u>modify it</u>! ← mark this as read only through a page table entry.

4) POSIX requires that a write system call requesting a write of fewer than PIPE-BUF (4KB on Linux) to a pipe must be atomic. A write of more than PIPE-BUF bytes need not be atomic.

a) What's the point of the atomicity requirement?

- The data of that size will not be split apart (hence the term)



b) Suppose your OS supports PIPE-BUF ≈ 2^64 - 1 bytes what sort of problems do you anticipate with your OS?

<u>Starvation</u>

+ higher buffer for atomicity
- increase wait time and potentially cause starvation.

6) One way a process can notify another is via the "kill" system call. Another is via "write" through a pipe. Can you replace "kill" with calls to "pipe" followed by calls to write?

Three ways to communicate:

1) Pipe + write - processes cannot be in existence yet
2) Kill - processes can.
3) Socket-based communication.

In order for two processes to communicate via pipe, we must pipe first. If they allready exist, pipes cannot be done to communicate them.

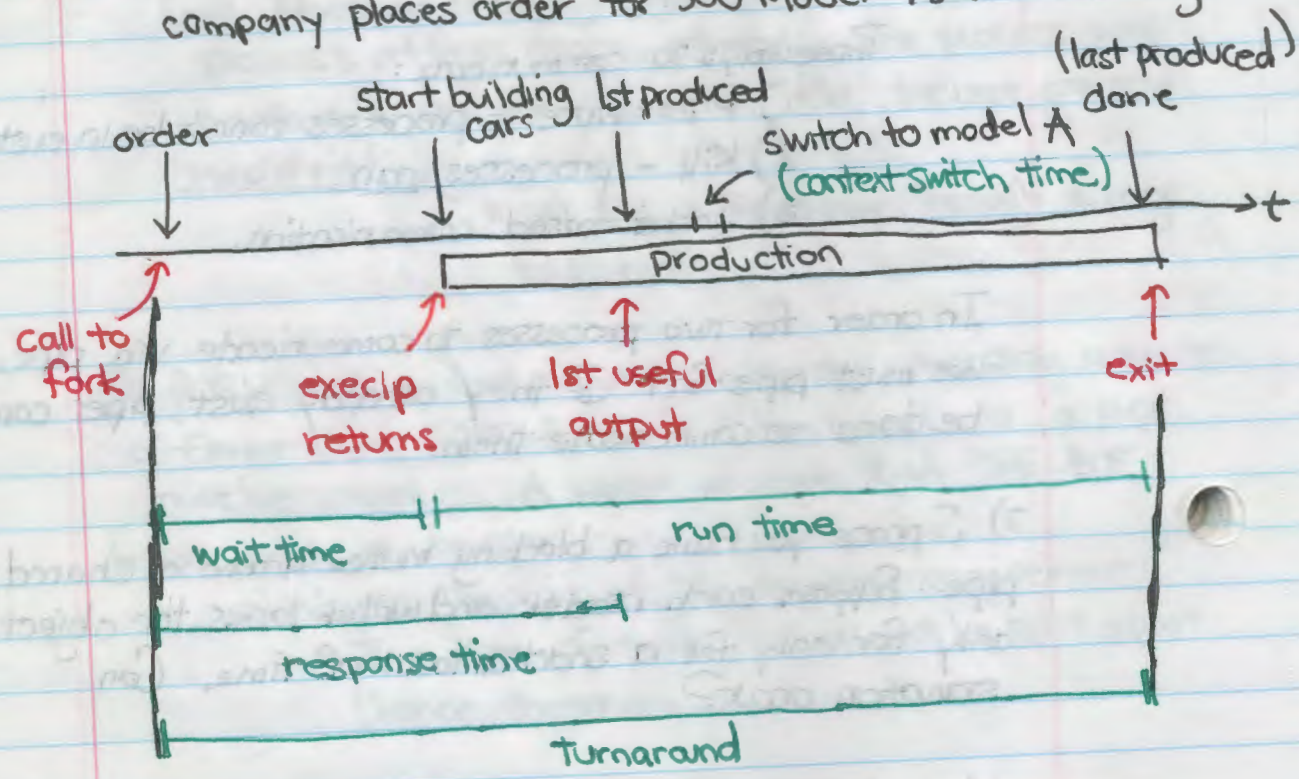
7) Suppose you have a blocking mutex around a shared pipe. Suppose each reader and writer locks the object only for only for a short amount of time. Can starvation occur?

Yes. A process that is never woken up can starve.

Lecture 8

Scheduling Metrics

Consider the Ford Model T:
company places order for 500 Model T's from Chicago



Think of the metrics as a set of raw materials to get some higher level properties:

1. Throughput (cars/month) - ("useful" instructions/sec)
2. Utilization - [0,1] Use as much of the factory as we can. useful/total instructions
3. Fairness - Each order should be given decent time. How likely will a process starve?

tend to fight each other :(

④ Latency ≈ turnaround time. Also competes with throughput.

Scheduling policies

1 First come, first served (FCFS)

Example problem:

process	arrival time (sec)	run time (sec)	1st output
A	0	5	2
B	1	2	1
C	2	9	4
D	3	4	1

δ = context switch time

0 → t

FCFS → AAAAA | BB | CCCCCCCCC | DDDD

5 | 5+δ | 7+δ | 7+2δ | 16+2δ | 16+3δ | 20+3δ

$$\text{Utilization} = \frac{20}{20+3\delta}$$:)

$$\text{Average wait time} = \frac{0+(4+\delta)+(5+2\delta)+(13+3\delta)}{4} = \cancel{10.5+1.5\delta}$$

:($= 5.5+1.5\delta$

How can we improve this?

2 Shortest Job First

0 → t

AAAAA | BB | DDDD | CCCCCCCCC

Utilization: The same

Average wait time: $\frac{0 + (4+\delta) + (4+2\delta) + (9+3\delta)}{4} = 4.25 + 1.5\delta$

This approach assumes that the job length is known. Reasonable in controlled, unreasonable in chaotic.

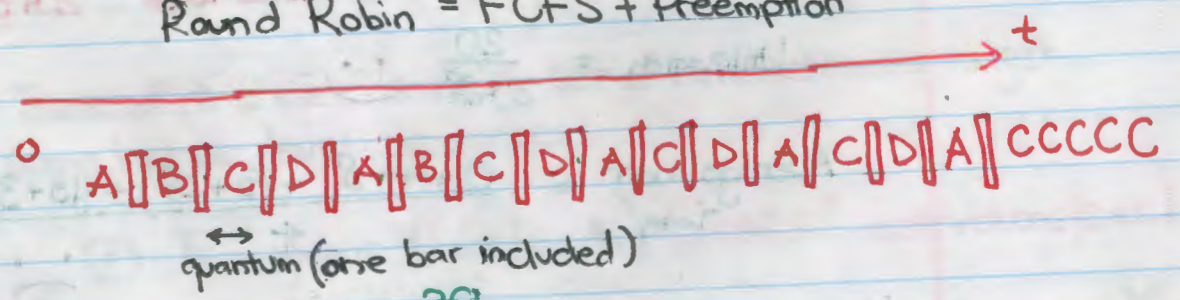
Another big downside: a job might wait "forever": It can be pushed back forever (so long as smaller jobs keep coming in).

For FCFS, once it arrives there's a finite amount of time before it is scheduled.

FCFS has better fairness.

$\boxed{3}$ Preemption

Round Robin = FCFS + Preemption



quantum (one bar included)

Utilization $\frac{20}{20+15\delta}$

Average wait time: $\frac{0 + \delta + 2\delta + 3\delta}{4} = 1.5\delta$

Very fair.

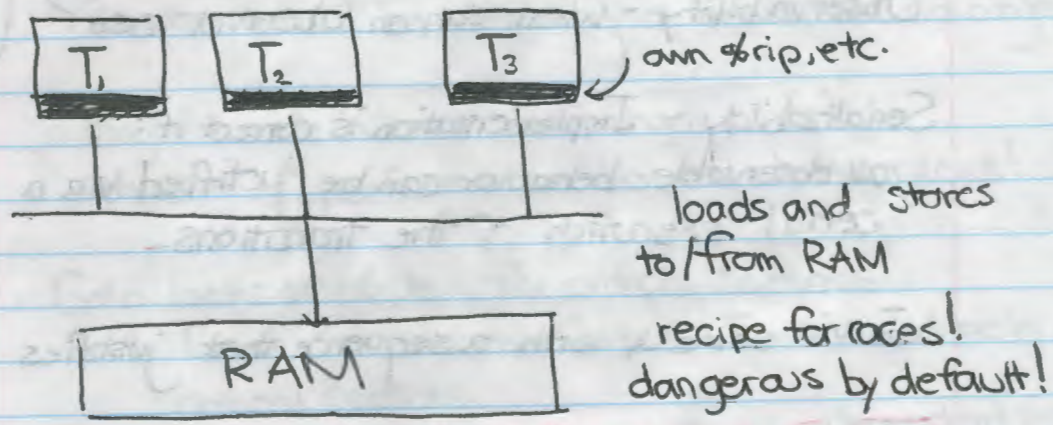
Note this approach applied to Shortest + Preemption it's basically the same; only if we're in the middle of a process and an even shorter process shows up.

Important: if we don't put new arrivals at the end, then we have potential starvation!

Note that the example above violates this convention!

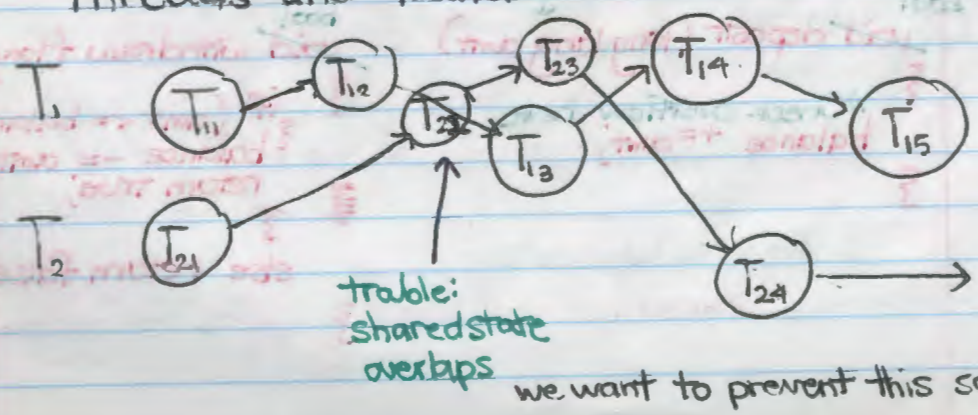
Synchronization

Coordinate actions in a shared address space.



- We want to be able to maintain data consistency
- Do so efficiently
- Do so clearly, simply, and flexibly

Threads and Trafalmidore



① $S_3; S_4$ ← <u>sequence coordination</u> done in each thread "x must occur before y"

② <u>Isolation</u> - make sure that it does not interact with any variables from another thread.

$S_5 \| S_6$

③ <u>Atomicity</u> - situation where threads must share information.
$\exists$ some ops that are atomic (either done or not done)

Observability - what can an observer see? (if invisible, its not a bug, nor is it correct! We don't care!)

Serializability - Implementation is correct if:
all observable behavior can be justified via a serial execution of the transitions.

It will come up with a sequence that justifies an observation.

$\boxed{\text{ex:}}$ Brothers and Bank Account
begin with $100

T_1 (me) T_2 (bro)
deposit $50 withdraw $20

long long ~~int~~ int balance = 10000 // in pennies

```
bool ~~void~~ deposit (long long amt)
{ //Check overflow here
  balance += amt;
}

bool ~~void~~ withdraw (long long amt)
{ if (amt <= balance)
  { balance -= amt;
    return true;
  }
  else return false;
}
```

check+

Race conditions!

Two withdrawals at the same time!

- Can go negative

Two deposits at the same time!

- ~~amt~~ balance += amt is <u>not</u> atomic.

How do we make it atomic?

<u>critical section</u>

section of code in which at most one program counter should be executing on behalf of an object.

Big = safe but ↓ performance :(

small = ↑ performance but still have races! :(

Goldilocks principle for critical sections

1. Look for shared writes. These should be in critical sections.

dependent reads

```
long long new_balance = balance + amt;
balance = new_balance
```

2. Expand to include dependent reads + intervening computation.

Lecture 9

Theory

How big should a critical section be?

Goldilocks: too small → races
too big → no llism

transfer (fromacct, toacct)

lock (from);
[delicate actions on from]
unlock (from) ← inconsistent state!!!
lock (to)
[delicate actions on t]
unlock (to)

We can lock both beforehand, but then what if one of them is used (locked). What we have here is a possibility of a deadlock.

audit (all objects) ← must lock everything!!! :-(
Take a snapshot()? still tricky! requires help from the system

Practicality

1) implementing critical sections on a uniprocessor
- disable interrupts during the critical section, only temporarily → ones that might affect it. especially the clock interrupt
use cooperative multitasking

p-thread sigmask

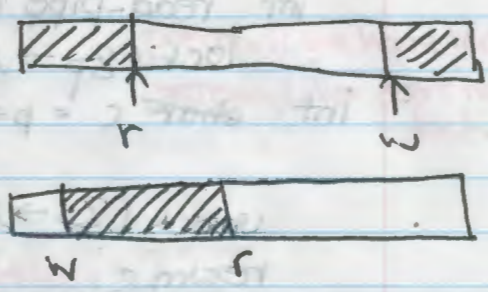
2) on a multiprocessor
sample program involving pipes of our own

```
struct pipe
{
    char buf[1024];
    unsigned r, w;
    lock_t lock;
}
```

go up to intmax

take these mod 1024

ask Tuan →



We can't tell if the pipe is empty or full!!!

expl: if we keep it this way then subtract r from w if full diff will be 1024 if empty it will be 0.

```
bool
~~void~~ writepipe (struct pipe *p, char c)
{   if (p->w - p->r == 1024) return false;
    p->buf[p->w++ % 1024] = c;
}   return true;
```

lock(&p->~~lock~~ l)

This is horrible: Lock at start, unlock before return

unlock(&p->l)

```
int
~~char~~ readpipe (struct pipe *p)
{   if (p->w == p->r) return CHAR_MIN - 1;
    return p->buf[p->r++ % 1024];
}
```

something that can't be a char!

Assume we have

lock(&l) ← spin lock: keep other threads spinning while it is used

unlock(&l) ←

```
char *ptr = ∅;
lock(&p->l);
bool full = p->w - p->r == 1024
if (!full)
ptr = &p->buf[p->w++ % 1024];
unlock(&p->l)
if (!full) *ptr = c  return !full;
```

won't work →

what if we have so many reads!!?

```
int read-pipe (struct pipe *p) {
    lock (&p->l)
int ~~char~~ c = p->r == p->w ? p->buf[p->r++%1024]
                              : CHAR_MIN-1;

    unlock (&p->l);
    return c;
}

bool writepipe(struct pipe *p, char c) {
 lock (&p->l);
 bool full = p->w - p->r == 1024
  if (!full)
        p->buff[p->w++ %1024] = c;
  unlock (&p->l)
  return !full;
}
```

To implement lock+unlock

A) as syscalls

+ easy

− way too slow

B) as plain machine instructions

```
typedef char lock_t;

void lock (lock_t *l)
{
    while (*l)
        continue
    *l = 1;
}
```

0 = unlocked

1 = locked

precondition

precond = we don't have

postcond = we do have lock

check

race here!

```
void unlock (lock_t *l)
{
    *l = 0;
}
```

precond: we have lock
postcond: we don't have lock

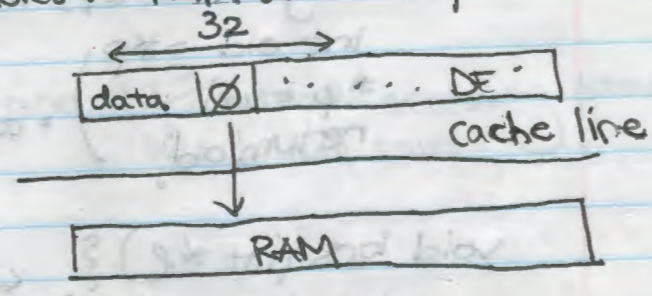
DOESN'T WORK, race condition

How does hardware access actually work?

Caching and cache lines, say 32 bytes

all loads + stores to RAM are 32 bytes at once

True, *l = 0;



Spin locks should not be too large

```
typedef __int128 lock_t;
```

registers only 64 bits

Would be split to two movq's, meaning it is not atomic!!!

spin locks should not be too small

```
struct pipe {
    unsigned char lock : 1; ← bitfield
}
```

Race between loading byte and storing byte in the bit field.

x86
x86-64 int works when aligned (on addrs of multiples of 4)

extra insns for atomic access for "int"

```
lock incl x
xchgl r, x
```

← some memory location.

```
int xchgl (int *p, int val)
    int old = *p;
    *p = val;
    return old;
```

= asm xchgl

```
void lock (int *l) {
    while (xchgl(l, 1)
        continue
}
```

← good.

```
int old = x;
int new = f(x);
while (xchgl(&x, new) != old)
    continue
```

NOT good enough, changes value even if it was wrong, then waits for it to be right again! :-

```
bool cas (int *p, int old, int new) {
    if (*p == old)
    {
        *p = new
        return true
    }
    else return false
```

```
while(true){
 int old = x
 ~~int new = f(x)~~
  if (cas(&x, old, f(x)))
    break
}
```

```
struct pipe {
   data      > 1000 bytes  ← large
   lock_t l      4 bytes  ← small
}
```

readpipe writepipe N-2 threads accesing same pipe
lock () N-1 will spin
work
unlock ()

If we made the lock global, it will still work, just make it slow (one lock for everything) <u>Coarse-grained</u>
+ easy to understand
- access to lock is a bottleneck

Lock per pipe is fine-grained locking
- complexity increase
+ fewer bottlenecks

What if I have two locks: r_l and w_l. does this work? No, they look at each other for this particular example.

6)

Suppose

many readers, few writers

reader:

```
while ((c = read_pipe) == CHARMIN_-1)
        yield() // let a more deserving thread go.
(do work)
```

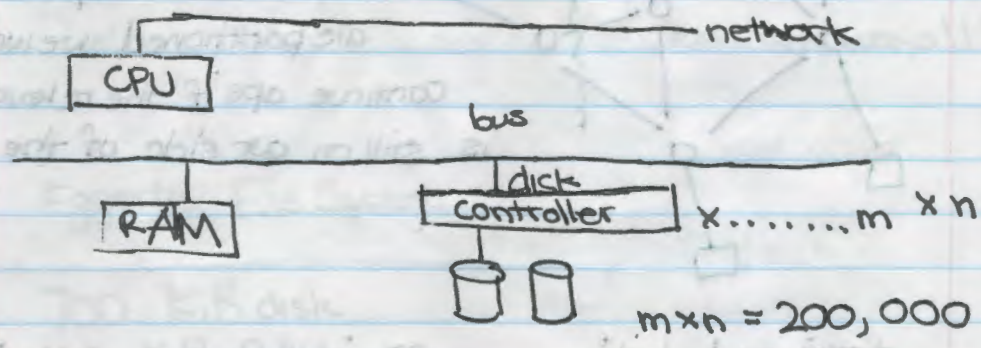
Lecture 11

File Systems

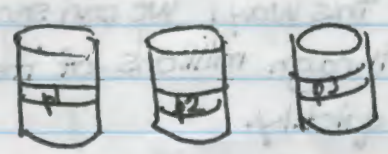
sample big file system

120 PB 200,000 drives (each ~600 GB)

GPFS General Parallel File System



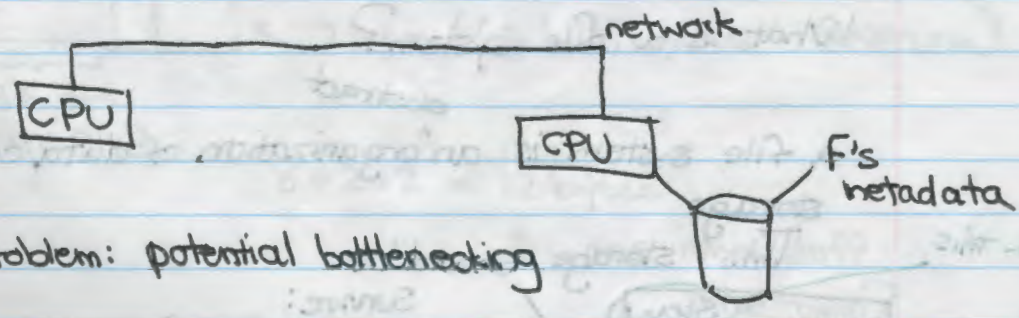
Striping - split files between disks



use ||ization for a faster throughput

distributed metadata (directories, timestamps)

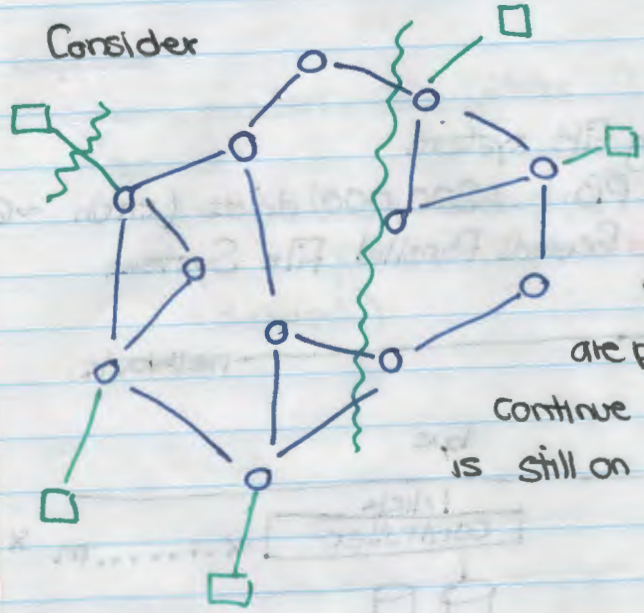
Hadoop, etc....



Problem: potential bottlenecking

partition awareness

Consider



When data/networks are partitioned, we want to continue ops if the relevant data is still on our side of the partition.

distributed locking

efficient directory indexing
↓
this way, we can search through millions of items very quickly.

file system stays live during maintenance
↓
while reorganizing/cleaning the system, users can still use the system.

What is a file system?

a file system is an abstract organization of data on secondary storage

This storage should be:

mask this if possible

It is: slow, big, persistent

Survive:
- hardware (?!)
- O.S.
- programs
- power

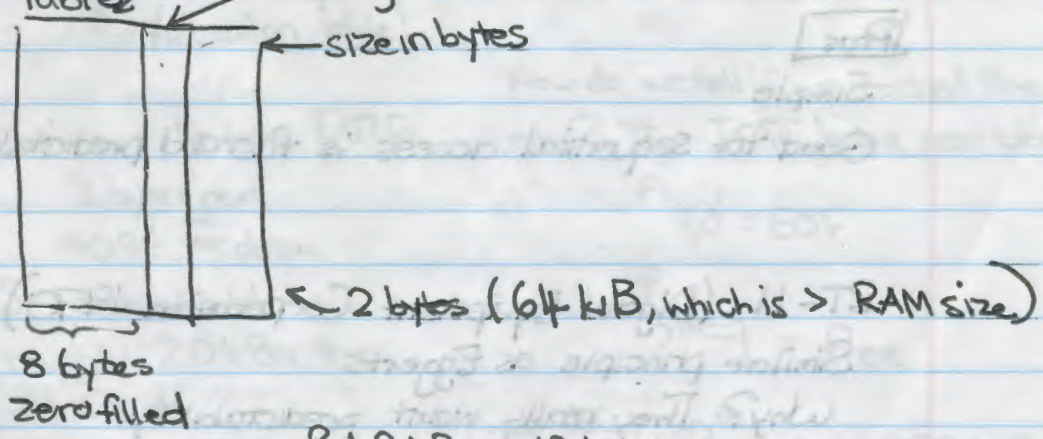
How abstract could it be?
- read data
- write data + rewrite
- copy data
- move data
- deleting data
- allocate space
- search data

↑ problem: how do we do this?
too many ways!!!

Eggert's File System

700 KiB disk
16 KiB RAM
16-bit

↑ table



8 + 2 + 2 = 12 bytes

4 bytes other stuff
timestamp?
16 bytes total
2^5 directory entries/sector
2^6 total files

If the name is all 0's then it is not in use.

Cons

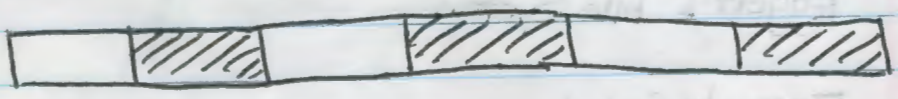
\# of files is limited by the code.

internal fragmentation up to 511 bytes – waste a sector regardless of the size of the file

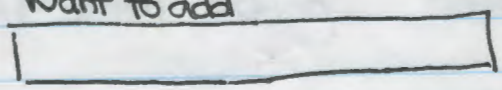
avg waste is 255.5 bytes

no permissions requires preallocation.

Fatal flaw:



Want to add



Now we have space that can't be used for this files!

This is called external fragmentation.

Pros

Simple

Good for sequential access, is fast and predictable

RT-11 (Digital Equipment Corporation 1970)

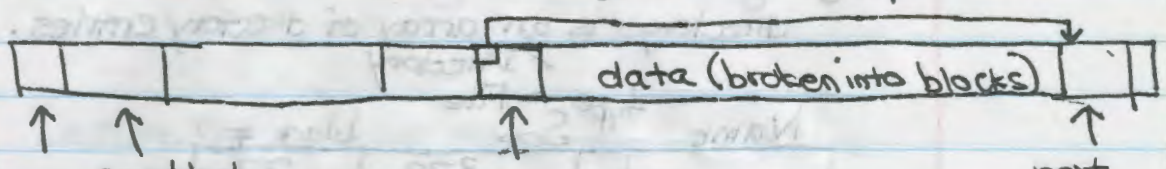
Similar principle as Eggert's

Why? They really want predictability.

Let's look at the FAT file system (1970's)

Seeks to avoid external fragmentation and preallocation

4 kiB blocks



boot sector

superblock
meta-inf about file system
- version
- size (blocks allocated)
- used count (blocks)

current block
do not insist on contiguous files, it could be anywhere!
Have a pointer to next spot!
Like a linked list

next block

Problem: Inconvenient data alignment.

next fields actually here, in the File Allocation Table!

Entries are 16 bits, FAT16

2 bytes next

4096 for data

So data should be roughly 2048x the next fields

How do we tell if we reached the end? The FAT has a next block

if $\emptyset$ = EOF

if $2^{16}-1$ = free

$(2^{16}-2) \cdot 2^{12}$ bytes = 256 MB

FAT 32 $\rightarrow (2^{32}-2) \cdot 2^{12}$ bytes = 16 TB

Big Idea: Directories are files!

directory is an array of directory entries.

Name	type (1 = File, 2 = Directory)	Size	block #
	1	372	27
		n bytes	index of 1st block

Directory is a pointer to a linked list of blocks.

11 bytes

8 3

foo bar.c ✓

c.foobar ✗

a.b.c ✗

+ No external fragmentation
+ No preallocation
+ # files not prelimited

– lose sequential access (it could be slow)
– defragmenting is slow and tricky
– lseek is O(N)

suppose we want to rename a FAT file

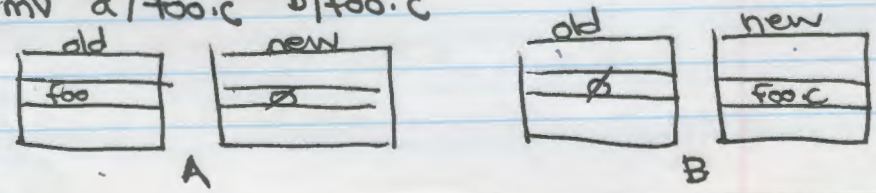
mv foo.c bar.c easy

foo.c	F	127	19

← make this change easily

what if

mv a/foo.c b/foo.c



Crash while writing b first → same location has two tags!
Crash while writing a first → nothing in start and end!

Lecture 12

File Systems continued
Hybrid data structure
- RAM → persistence
+ disk → performance

e.g. Directories (RT-11, EAT)

flat ← tree structure nestled

Linux ext3

directory entry
- 32-bit inode #
- 8-bit name length
- N-byte file name
- 8-bit file type
- 16-bit directory entry length

on disk

Inode - (in main memory metadata for file)
- fixed size (per file)

varying length entries

inode Table

1	2	3	4	.....

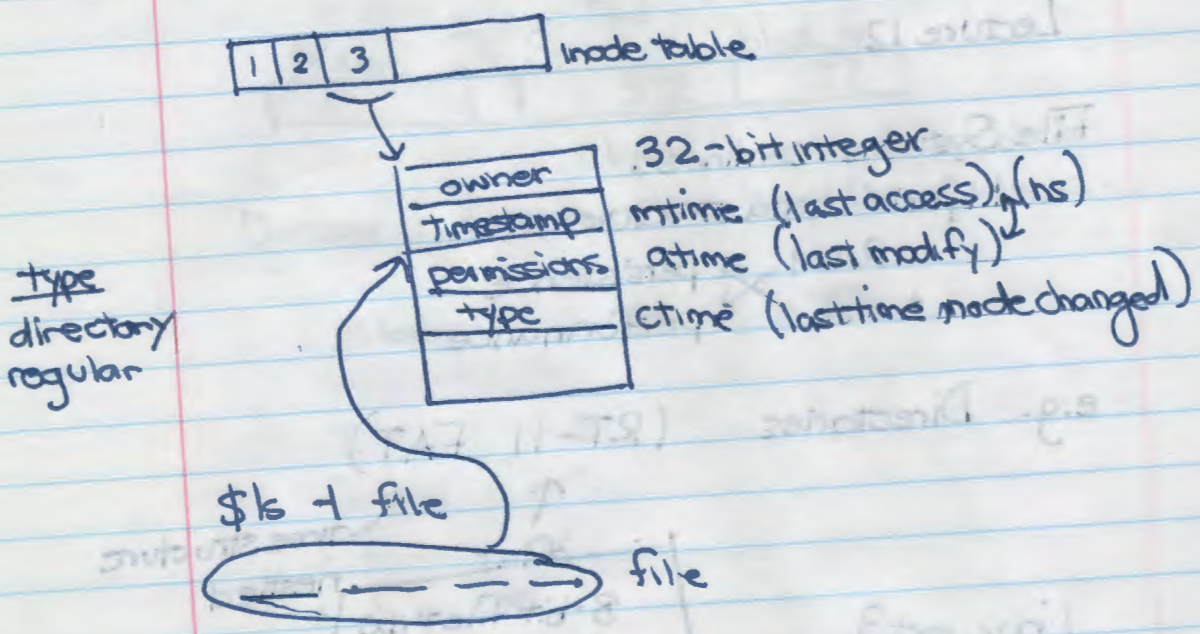
info about file 3

- have numbers and not names

$0 \le N \le 255$

Why do we leave so much unused space?
Why track the dir length?
- unlink ("foo.bar")

To unlink we look at the previous one and expand its unused space.



Inode table should tell us all the info about the file
The directory entry should store about the file name.

We can think of type as a cache for inodes.

find . type f
"Find every file in current directory, output time. Turns out, type is beneficial for performance reasons.

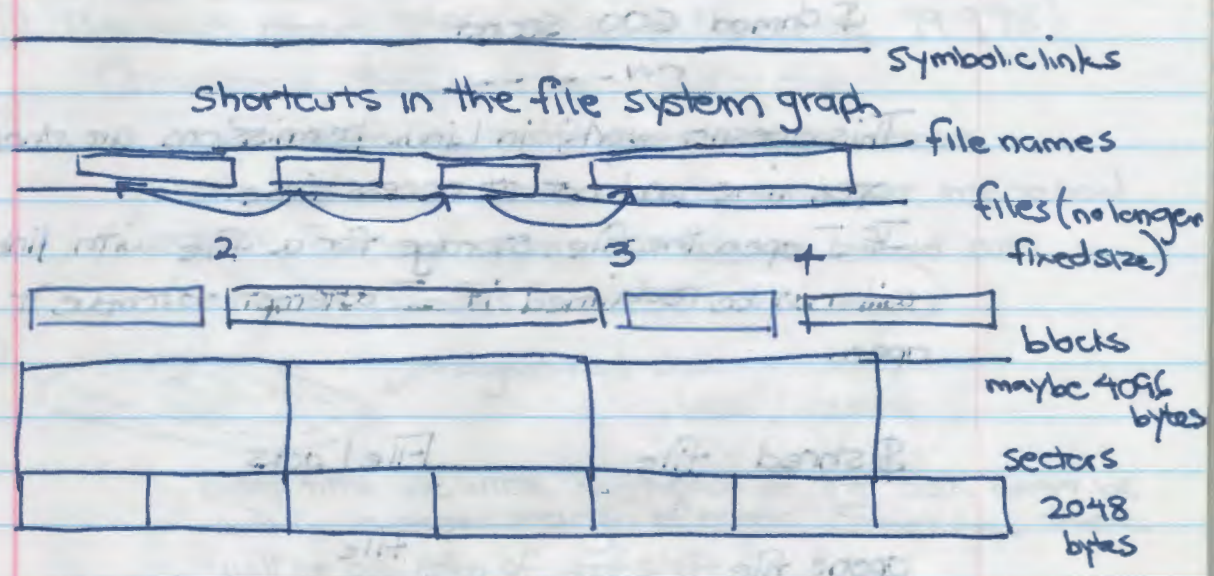
So why don't we cache other stuff?

find . ~~type~~ mtime +1
(slower)

Well, these things are subject to change. After files are created, don't underestimate the gobies anymore

The inode does not tell you its file name

(levels of abstraction in the file system)



What if we have a couple xtra sectors? Ignoring them is not costly, so we do that.

Suppose we want to ~~a~~ remove a file securely

Will unlink ("foo") work? No! All it does is remove the directory entry!

watch out! link count could still be non-zero!

assume link count is ~~0~~. 1

$ ls -l foo

- - - - - - - - - - -

$ rm foo safe? No!!!

If I ran ls -li it would give me an inode #. I could

look at.

What if we have race conditions we think that the link was removed but one was added in a race condition

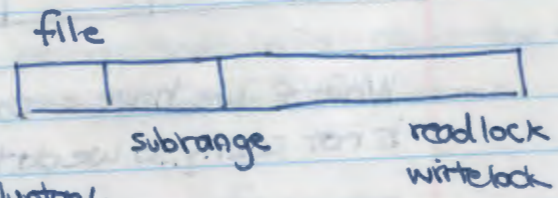
$ touch secret
$ chmod 600 secret
rw- --- ---

This doesn't work; in Linux permissions are checked at open time and not at access time.
If I open the file, storage for a file with link count 0 will not be reclaimed if I attempt to remove it while it is open.

$ shred file

opens file
stat to get size
overwrites with 0's.
closes the file
unlink file

File Locks



voluntary.
why!.
- Prevents deadlocks
- Lets user override processes that may be wrong about their locks.

How "root" can defeat shred.

/dev/rdsk/00 ← block special device that allows us to access very low level access (blocks or sectors)

shred's overwrite step may use new blocks that were free, so the sector still holds the old data

How do we fix this?

$ shred /dev/rdsk/00 → will shred every sector of the disk.
Surely, this must work.
But it DOESN'T :(

① Disk drives aren't 100% reliable (more like 99.97%)
Occassionally there will be a bad sector.
- Attempted read/write gives checksum errors
We have replacement sectors at the end that map to bad sectors. Thus bad sectors will not be zeroed out
(hardware pretends bad sectors don't exist)

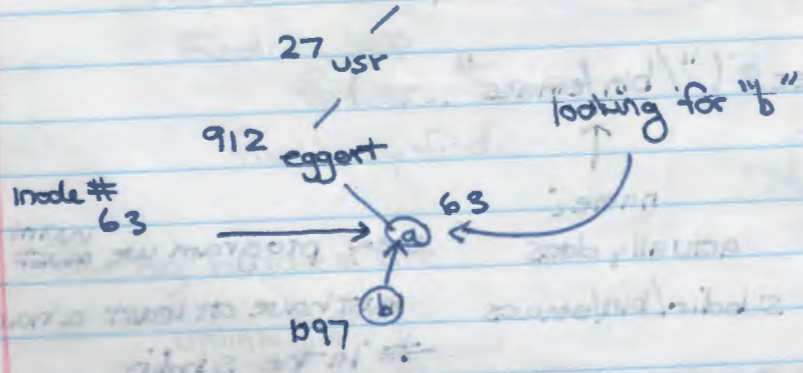
②
magnetized disk

Every time we write, the heads of the disk won't be within a proper margin of error. Sometimes there will be an echo of what is left.

How to fix? We overwrite w/ random data, 3 times.

How to interpret file names

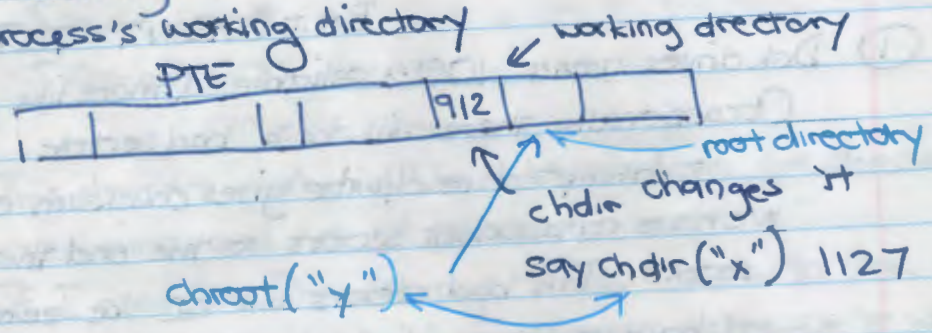
namei: "a/b/c" → 12963



However, something like cat a/b/c it interprets from the working directory, or do we?

The case is, use working directory if no slash.
If slash, start from the root.

The working directory is <u>not</u> a global property, it is a process's working directory




```
chroot ("subdir");
execl ("/bin/emacs", .....)
```

← Chrooted "jail"

abc/. ≡ abc
/a/b/.. ≡ /a *
/.. ≡ / parent of root is root, inode # will be same

If we do...

```
chroot("/..");
chdir(".");
```

← noop! can't get out of jail! >:D
← noop.

```
execl ("/bin/emacs" ....)
```

namei actually does subdir/bin/emacs

every program we ~~must~~ want to run must have at least a hardlink ~~to~~ in the subdir

How to Escape Chrooted Jail

```
ln /usr/bin/emacs subdir/bin/emacs
ln /dev/tty       subdir/dev/tty
ln /dev/dsk/00    subdir/dev/dsk/00
chroot("subdir")
execl("/bin/emacs"....)
```

Once we open /dev/disk/00 for rw
In some other file to accessible location

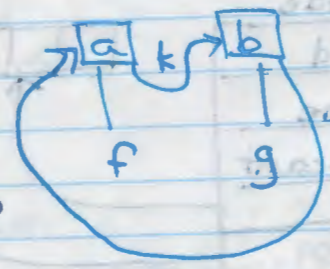
---

last component special

```
open("a/b/c/d", O_WRONLY | O_CREAT | .....)
                           | O_EXCL
```

special
we don't insist ~~that~~ this (d) exist!

link("a", "b") ← create cycles?



mkdir a b
touch a/f b/g
ln a b/l
ln b a/k

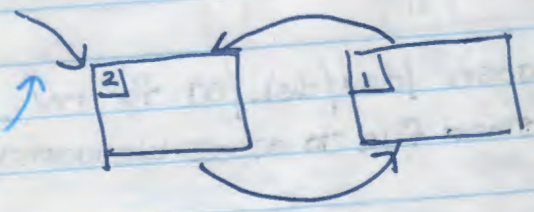
Why is this bad?
Find will loop forever. This is fixable by find.

If we name:
a/k/l/k/l/k/l/f
This is a perfectly valid filename!

The big boddies are
unlink("f")
rmdir("k") ← when we remove and its the last use, we want to give the space to the OS.
Cycles make it hard to figure out when storage is fo

ex: rm a/f
rmdir a/k
rmdir a

linkcount = count # of directory entries pointing at a node.



erase this will make these files inaccessible.

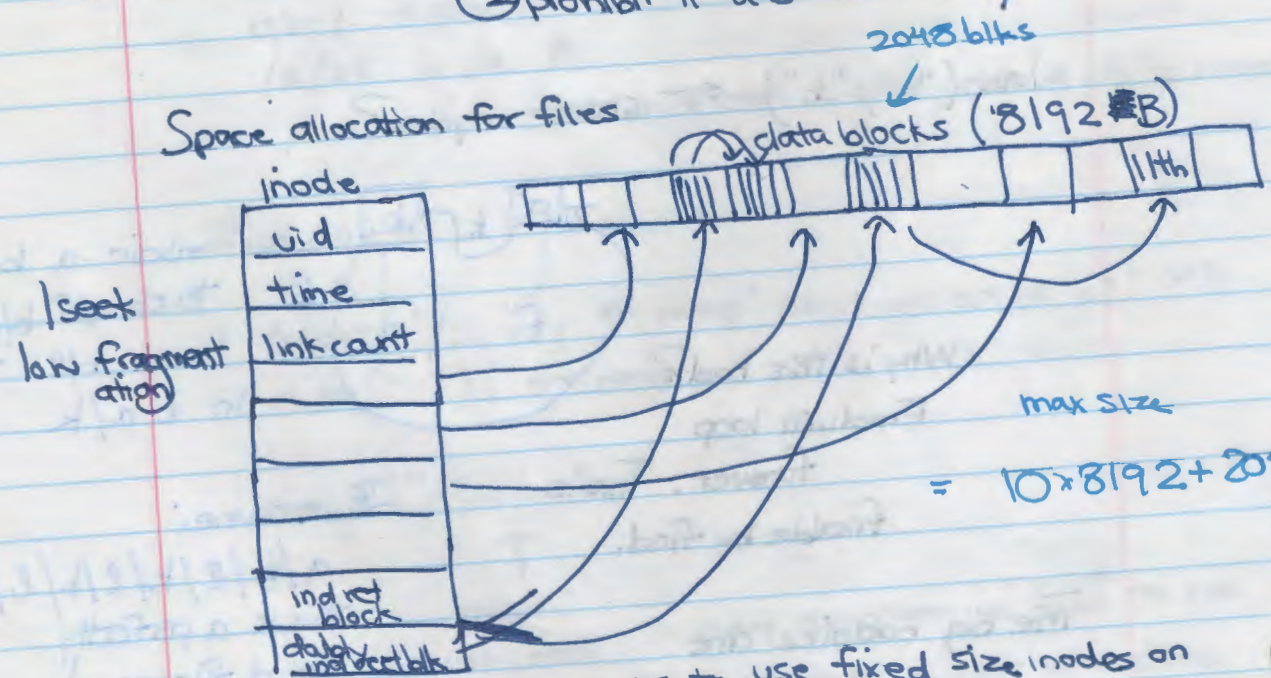
Just make ~~files inaccessible~~ cycles impossible.

How?

① check for cycle dynamically in link()
— expensive

② prohibit if a is a directory

Space allocation for files



max size

= 10×8192 + 2046×8192

allows us to use fixed size inodes on large files

Lecture 13

What's the worst that could happen?

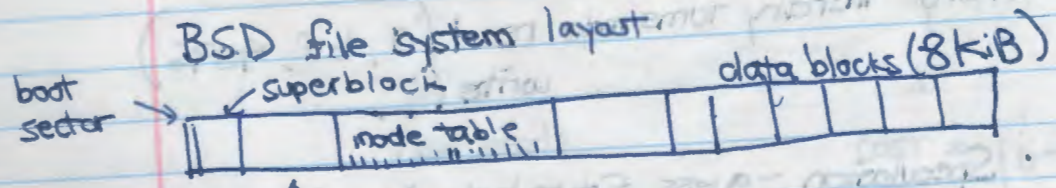
```
gzip foo        ifd = open("foo", O_RDONLY);   ← if (ifd<0) error();
                ofd = open("foo.gz", O_WRONLY | O_CREAT | O_EXC, 0x600);   ← if (ofd<0) error();
  ↓
foo.gz

        read(ifd, buf, bufsize)   ← check
        write(ofd, buf, )         ← for errors

if (close(ifd) != 0) error();
if (close(ofd) != 0) error();
if (unlink("foo") != 0) error();
exit(0);
```

Bug report: This is unreliable. In some instances gzip can cause you to lose all of your data.

But what's going wrong? We checked everything!

We'll come back to this later.

BSD file system layout



free block bitmap

1 bit per block of data, redundant but helps performance 65536:1

↑

plausibly cachable in RAM

Performance issues with file systems

- overhead to read a block from disk
  = seek time + rotational latency + transfer from disk + transfer to RAM
  10 ms + 8 ms + 0.1 ms

18 ms = 18,000,000 instructions

I/O

metrics

utilization - what % of time are we reading/writing data

throughput - rate of request completion (operations/sec)

competing objectives — latency - delay between a request and a response

How to Improve File System Performance

- Batching - coalesce small requests into larger ones

```
while ((c = getchar() != EOF)
    putchar(c);
```

← superslow

stdio library turns into read(0, buf, 4096)
write(1,

- Speculation - guess future behavior of apps and do that work preemptively
  - reading, PREFETCHING
  - writing, DALLYING

LOCALITY OF REFERENCE

<u>temporal</u> locality - access a location i now, likely to access it again in the future

at time t

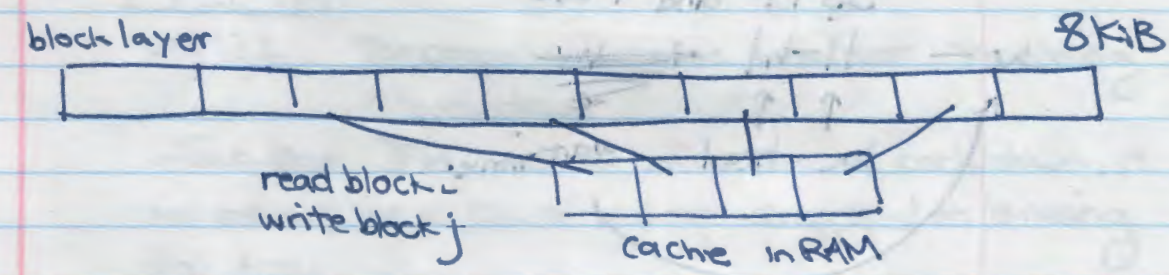
and again at time $t+\delta$

↑ small

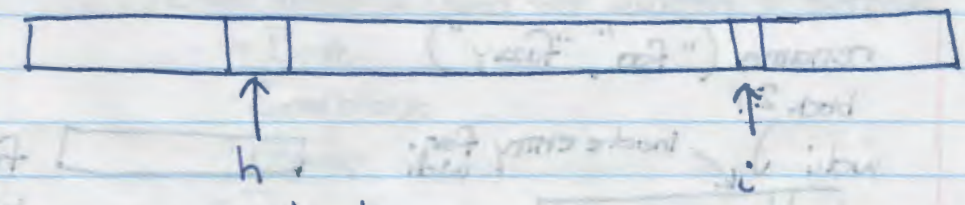
<u>spatial</u> locality access location i at time t

likely to access location $i+\epsilon$ at t

↑ nearly the same time



Simple model of disk access



cost $= |i-h|$

Avg cost of a read.

assume random requests

$0.25 \le \text{avg} \le 0.50$

$$\int_0^1 \frac{(h)^2}{2} + \frac{(1-h)^2}{2}\, dh$$

$$= \frac{1}{3}$$

To approximately maximize throughput, we will do shortest seek time first

This does not always produce the minimal approach. Worse, it might cause starvation!!!

Elevator algorithm - a compromise
popular, but.....
It's easier to wait at the center b/c we can encounter it going in either direction

~~Circle~~ Circular Elevator, always seek forward!

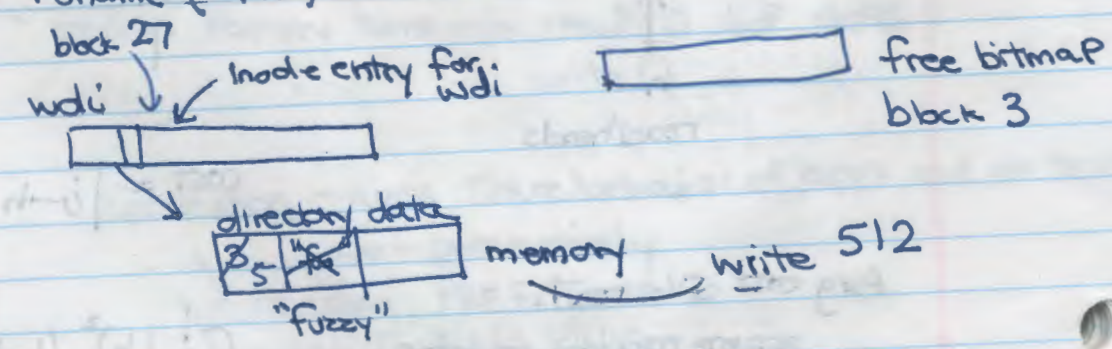
SSTF and FCFS

w — $|l-h|$

loc head wait time

Suppose we issue a system call that renames a file

rename ("foo", "fuzzy")



If the directory grows, we need another free block, say 1973

So we need to rewrite a total of 4 blocks to get this to work!

What order should we perform these writes?

Block 1973 – crash can't corrupt a free block

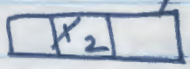
Block ~~27~~ 3 (~~inode~~) – memory leak: free bitmap

Block 27 (inode) ← crash here = 2 hard links to same file, not the end of the world.

Directory Data

However, ~~the~~ link count will not be updated! Disaster.

inode entry for "foo"



blk 132

Temporarily increase link count, then bring it back down

~~This cha~~ If it crashes before coming back down, it will merely be a space leakage as opposed to erasing vital data.

File System Invariants

1. Every block is used for exactly 1 purpose.
   - Boot
   - Superblock
   - Bitmap
   - Inode
   - Data
2. All referenced blocks are appropriately initialized.
3. All referenced data blocks are marked as used in the bitmap
4. All unreferenced data blocks are marked free in the bitmap.

Penalty for violation: total disaster. Data loss.... which is the whole point

Note that 4 doesn't cause data loss, we instead merely lose some storage.

stat ("foo", &st)

↑

st.st_ino → gives file's inode number

fsck → looks through file system for.....

- free blocks not marked free
- files that have nonzero link count but not in directory

↓

/usr/lostandfound/01396

Now let's return to the gzip problem at the start of the lecture.

ext 2.. ext 4 ordered mount option

↓

Tells file system code to not use the "queue", do everything in order user encountered. This lets code survive crashes.

$\boxed{\text{Fix}}$ significantly slower ☹

Suppose gzip crashes here...

block w → write (ofd) ← if (fsync (ofd) != 0) error();

if (unlink("foo") != 0) error();

crash → exit(0);

↓

block u

If u is written before w, we crash beforehand, and lose the uncompressed file before finishing the compressed files!

☹

Lecture 14

http://bugsignv.org/22768

```
gzip foo
fd = open("foo.gz", O_CREAT.....)
write(....)
close(fd)
unlink("foo");
```

The problem with this code is that it might crash and have executed the unlink code

```
dfd = open(" ", O_SEARCH)
    fd = openat(dfd, "foo.gz", O_CREAT|.....)
    write(.....)
    fsync(fd) ← saves 'write'
    datasync(dfd) ← saves file creation
    close(fd);
    unlink(dfd, "foo.gz");
```

implemented in — synchronous

The controversy: This is 700x slower!!!

man mount ext4

we want reliability 1st
performance 2nd

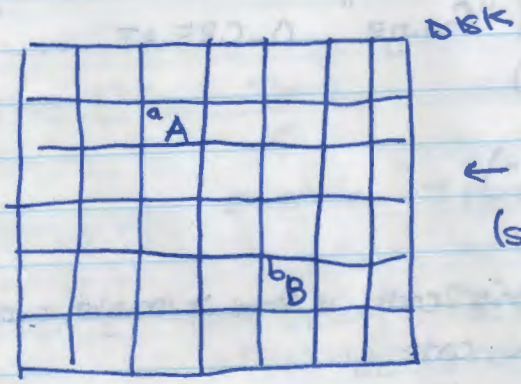
2 big ideas for reliable filesystems

1. Commit Record
   a) assume individual sectors can be written atomically
   b) collect several writes, $\underbrace{S_1, S_2 \ldots S_n}_{\text{tentative}} + \underset{\text{commit record}}{S_{n+1}}$

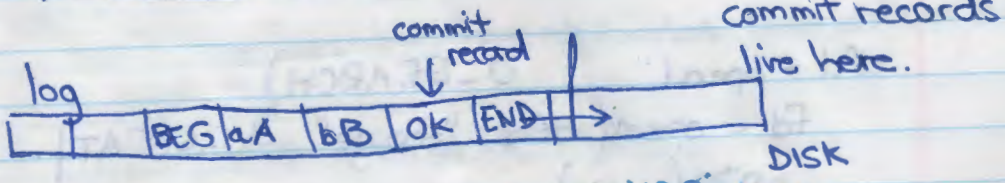
If crash happens in tentative, they are ignored.

② Journal

cell memory



← lots of blocks
(state of the filesystem)



commit records live here.

write ~~and~~ ahead log

BEGIN

good to abort if fail here

- log intended changes

upon reboot, if there is a begin, an OK, and no end, it will replay from the last begin

COMMIT (write commit record)

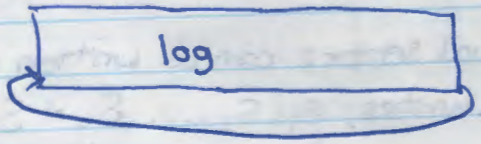
- copy changes from log to cell disk

if these fail, it is probably good to compensate

END

What happens if we run out of log? :(Just go back to the beginning.

All log, no cell data



+ periodic g.c.

efficient for lots of small writes.

fast recovery requires a scan backwards and a scan forwards. Is there a way to make it so we only need to scan one time?

Write behind log

BEGIN

log old values

install new values into cell data

COMMIT

END

+ faster recovery (we restore old vals from logs right to left in the event of a crash)
+ Only need one COMMIT/END (not both).
- Old values have to be written from disk.
- Wastes effort copying/reading old data from disk.

Main memory database

+ fast access

- If you lose power, main memory will be lost (solvable w/ log)

A more complicated approach.

Let users see "inside" pending results

① Cascading abort

caller should fail if callee doesn't commit.

COMMIT will become ABORT

High level issues aborts if a low level one also aborts

② compensating actions

caller fixes low level problem in some other way.

recovery should be robust in the presence of crashes
- make it idempotent:

$$recover(recover(x)) = recover(x)$$

Can we partition our data into "must be persistent" vs. "OK if we lose it"

Yes. We can use multiple file systems on the same machine.
But how do we support things like name_i?

Mount table

fs, inode # → fs		
dev_t	ino_t	dev_t

name_i

/usr/bin/sh

ext4 → ↑

(dev, inode #)

uses this device to identify which methods to use.

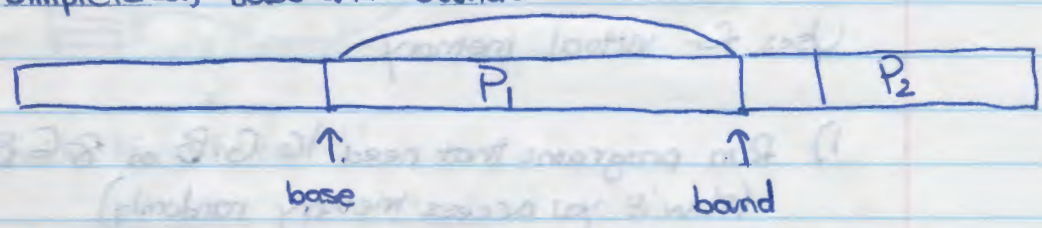
vfat → /home/eggert/catvideos

Next Problem:

unreliable programs ≡ invalid memory references

Solutions: 1) hire perfect programmers
2) Use Java, javascript, Python (software checking)
3) Have hardware guys fix it.

Simple idea, base and bounds



We want every mem access to be between these two

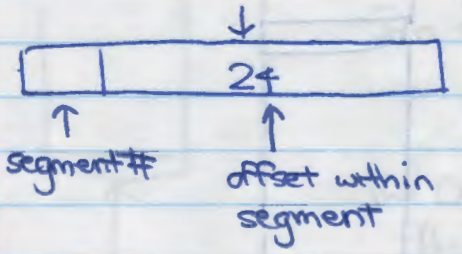
base, bound } live in PTE for context switches

privileged regs

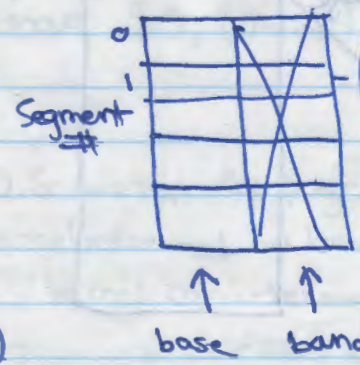
Problem: Fragmentation. We aren't really allowed to do this.

Segmentation

addresses



segment table

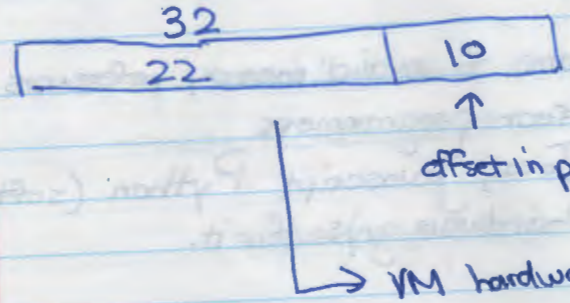


(Internel grow stack (move it, maybe adjust these values)

stack segment (for stack)
text segment (for code --)

still have problems of segments too big.

we will insist that every segment is the same, relatively small size



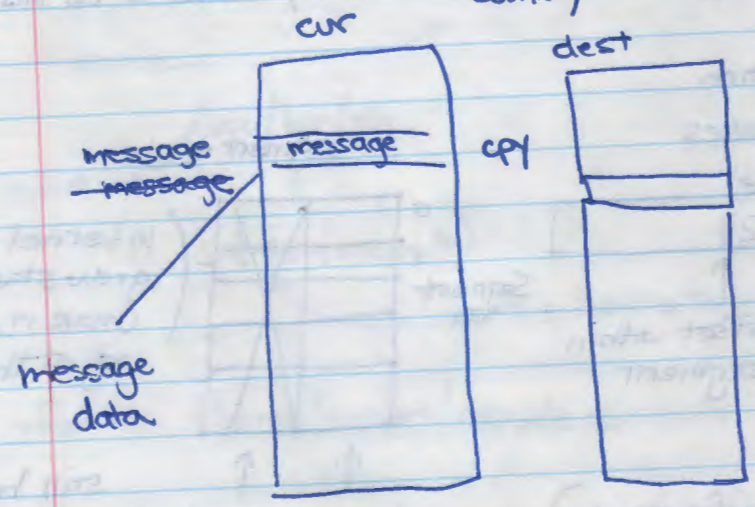
page-based memory
every page is 2^{10} bytes

Uses for virtual memory

1) Run programs that need 16 GiB on 8 GiB RAM computers!
(slow if you access memory randomly)
2) Programs can share memory safely.

sendmsg(destproc, message)

pointer to
4096-byte buffer aligned to a 4096-bit boundary.

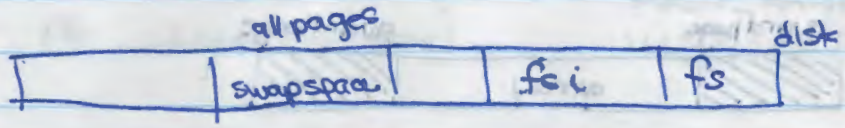


malloc (293617000), implemented how?

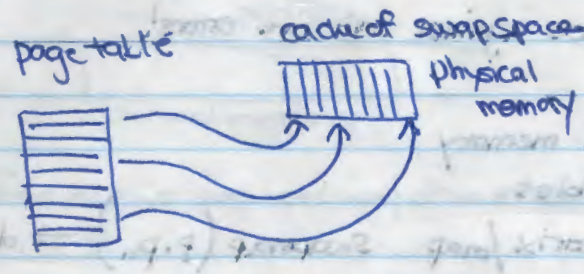
implement via a call to mmap (man mmap)

application asks kernel to modify its page table

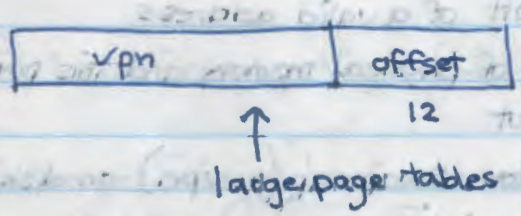
args: fd to mmap from (you ~~get~~ see a section of this file into main memory)



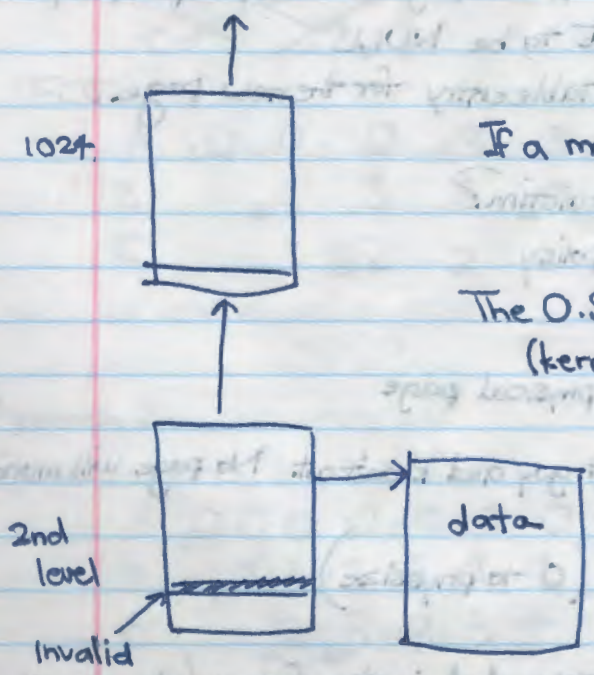
2TB



Lecture 15 VM cont'd

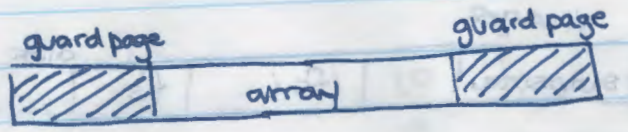


What if we split it into two pieces? 2-level page table



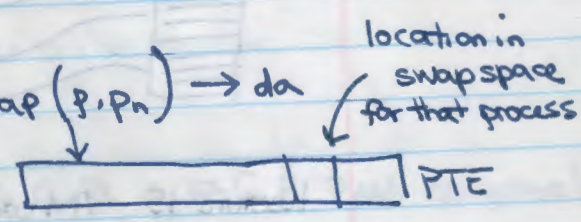
If a movq $0, (%rsp) looks at an invalid entry, it falls into the OS after trapping

The O.S will typically page fault
(kernel informed by hw of faulting address)
kernel can: - terminate the process
- send a signal, SIGSEGV
- arrange for the VM to become real memory, pull the page in and retry the instruction (slowest)



Hardware protects against software errors!

What's in kernel memory
- page tables
- swap matrix / map swapmap $(p, p_n) \rightarrow da$



on a page-fault of a valid address
- we ran out of physical memory and this particular item is sitting out
- read page from swapmap (p, vpn) on disk into....
  ??? (the victim page)

We need some used memory to get kicked back out onto disk!

chosen from a victim page (p', pn')

takes a while
- write from physical page to swapmap (p', pn')
- update old PTE to be NULL
- update page table entry for the new page.

How do we choose our next victim?

Page replacement policy

① i++ / victim++
↑ index of physical page

Just take the next guy and kick it out. No page will monopolize

② victim = random (0 to physsize)

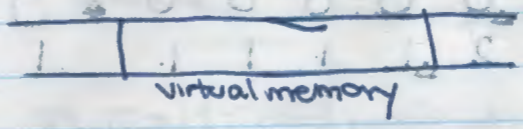
③ victim = page least needed in the future (the longest amount of time we go without it) ← ORACLE POLICY

"advice"

third arg = MADV_SEQUENTIAL
MADV_RANDOM
MADV_NONE ← free to victimize

④ madvise (start, size)

virtual address



Turns out this isn't used very often at all....

Why?
- It's secret.
- It doesn't always improve things

⑤ Least Recently Used (LRU)

Reference String - log of accessed pages

Ex: Say we have 5 processes, 5 virtual pgs, and 3 physical pgs

0 1 2 3 0 1 4 0 1 2 3 4

Approach 1

A	0	0	0			3					4
B		1	1	1			0	0			
C				2	1						

FCFS

A	0	0	0	3	3	3	4	4	4	4	4	4
B		1	1	1	0	0	0	0	0	2	2	2
C				2	2	2	1	1	1	1	1	3	3

9 Page faults

Approach 2

Oracle

A	0	0	0	0	0	0	0	0	0	2	2	2
B		1	1	1	1	1	1	1	1	1	3	3
C			2	3	3	3	4	4	4	4	4	4

7 page faults

Approach 3

LRU

A	0	0	0	3	3	3	4	4	4	2	2	2		10 pg
B		1	1	1	0	0	0	0	0	0	3	3		faults
C			2	2	2	1	1	1	1	1	1	4		

Throwing hardware at the problem makes the problem worse!
This is Bélády's Anomaly

How do we build LRU?

page table

pa	bits	clk

clock value of last access
updated at each access
kill the oldest

page table

keep track w/ one bit based on bits

Keep an array that the hw doesn't know about

Virtual Memory Optimizations

- demand paging A) read program into RAM, run it
  B) don't read the whole program, just page 0

+ say we only want to check the program's version. Simple uses for the program will avoid unnecessary work!

- If we have a program that needs most of its pages, we will be unable to batch it and thus will have more page faults!!!

• Dirty bit Page fault

chooses a victim

Skip if possible → writes out old page (can be redundant)

reads replacement page

SOLUTION 1: read before writing X (unless reads are significantly cheaper than writes, unlikely)

SOLUTION 2: checksum what is on disk X collisions causes data loss

That is, checksum may be same for two different pieces of data

Also, checksums take memory

SOLUTION 3: dirty bit

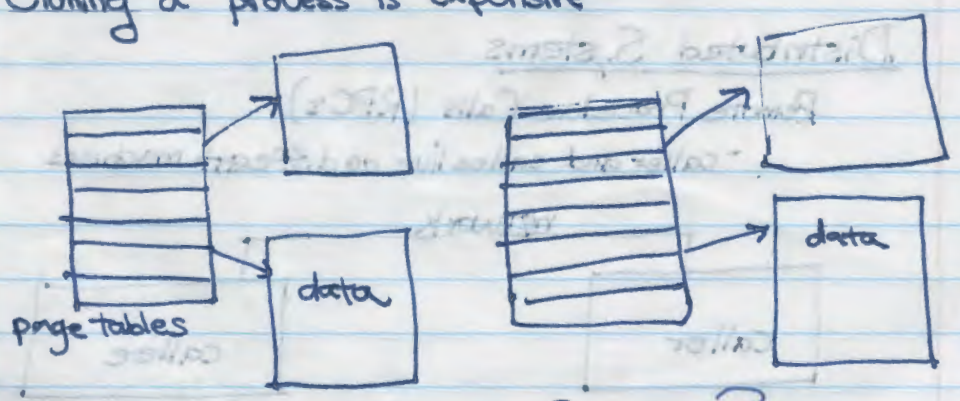
Whenever we access a page, if it is 0 we will set it to 1.

Note that it is likely to change to 1 and stay 1 for a long time or remain 0 for a long time (minimal change overhead)

How do we mark a dirty bit w/o having it in hardware?

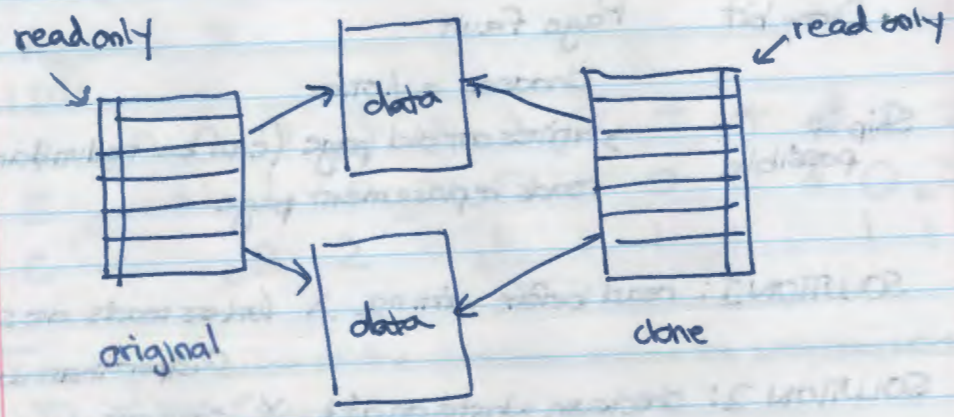
We have the page set to read only even if it is read/write after changing, and have a software track.

Cloning a process is expensive



copy of page. So expensive?

When we fork, we don't make a copy of the data



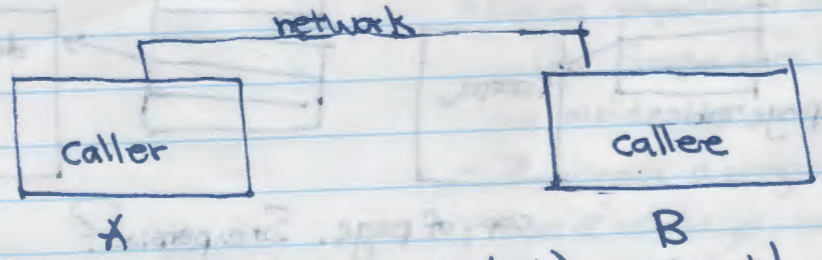
If we try a write there will be a page fault. It will show that it should be writable in the kernel (but we set it to read only before!) It will then copy ~~these pages~~ this page, make a copy for this page, and set the entry in the table for this page to be rw once again. This is the conventional modern fork.

Downside, we still have to copy the page table!!!

vfork = fork, except
1) Parent + child can share memory
2) Parent is frozen until the child execs or exits.

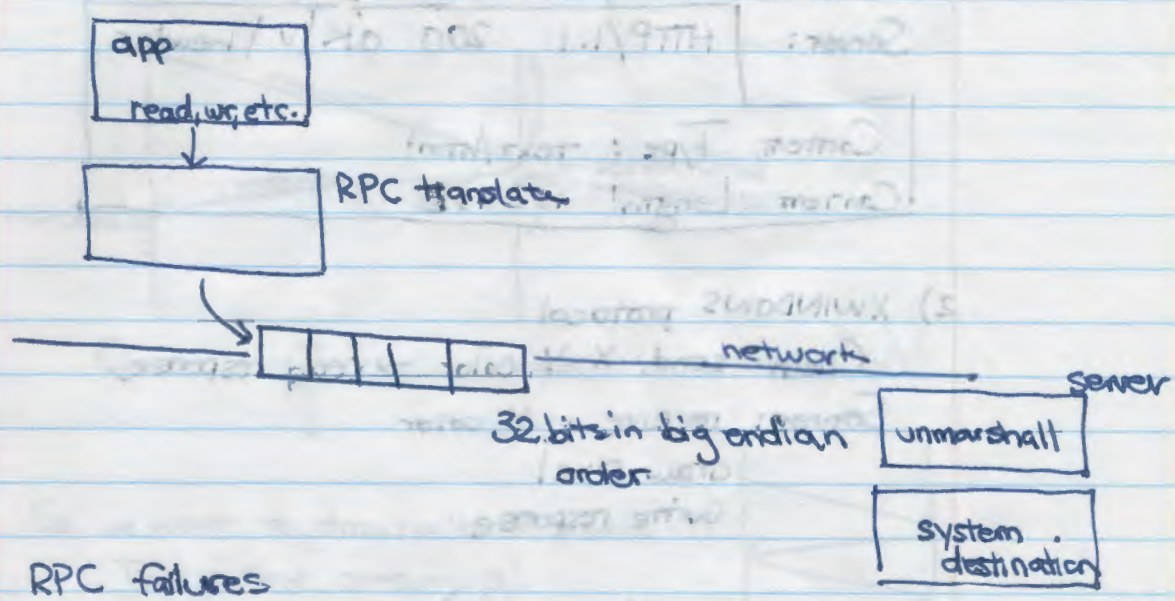
Distributed Systems

Remote Procedure Calls (RPC's)
- caller and callee live on diff machines



1) No shared memory strcmp(a,b) won't work!
+ Hard modularity - Pain to access data
2) No call by reference
3) Machines might not even be the same in terms of insns, data formats

Simulate RPC's as if they were ordinary calls



RPC failures

- messages can be lost
- messages can be corrupted (more often than in a central system)
- network might be down (or) up and slow

Lecture 15 RPC cont'd

1) Client: GET / HTTP/1.1 \r\n
Server: HTTP/1.1 200 OK \r\ header
Content Type: text/html
Content Length: 102438

2) XWINDOWS protocol
Client: send, X, Y, color → read response
Server: receive x, Y, color
draw pixel
write response

XDrawPixel (x, y, color)

new problems → messages can be corrupted, lost, or even duplicated.

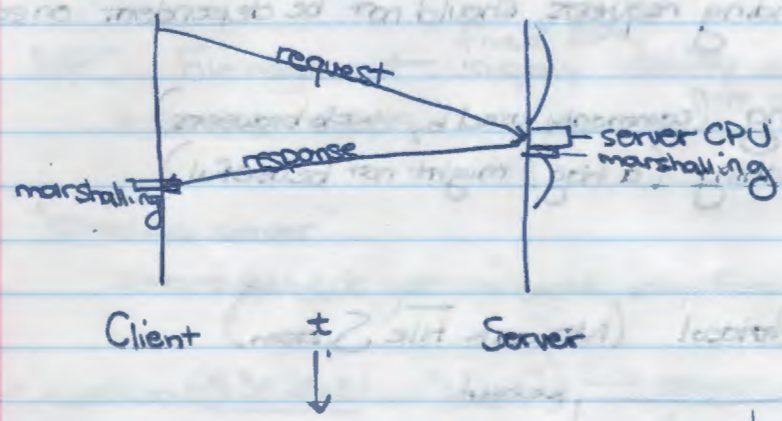
Solving them
duplication: - sequence number
- If action is idempotent, then it doesn't matter!
corruption: - checksums : 16 bit (simple, crypto, RS)
→ ask correspondent to resend if there's an error
lost: - wait for awhile, fail → at most once RTC
??? ← too short = not actually a failure
too long = wasted time!

Furthermore, what if the operation succeeded!

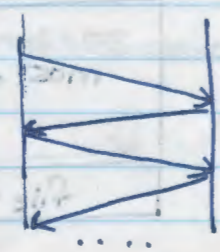
- wait for awhile, retry until it works
→ at least once RTC

- exactly once RTC :), but hard to implement (impossible)

Performance issues with RPC

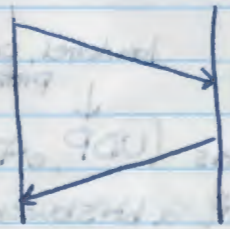


Say we want to draw a lot of pixels:
This is time consuming!



Approaches:

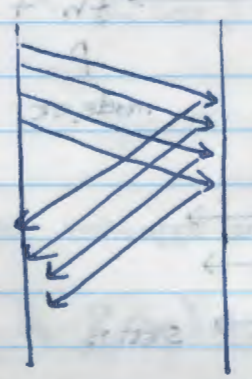
① Batching ex: XDrawImage(X, Y, 2Darray of colors)



Short, 1 time request/response
Takes longer than a single pixel but needs to be performed ~~for~~ less often.

Problem with batching:
- Affects client code!

②



Pipelining: Don't wait for the previous response before considering the next request!

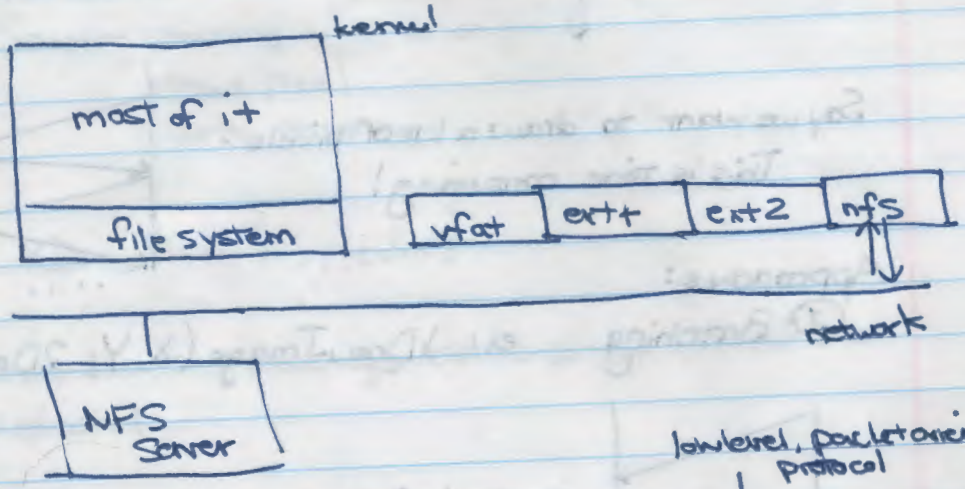
Good if each individual error doesn't have to be accounted for.

```
for(i=0; ......)
    putchar(c);
if (ferror(stdout));
```

A) Be prepared for asynchronous errors.
B) Outstanding requests should not be dependent on each other.

Caching (commonly used by web browsers)
Prefetching (a hog, might not be useful)

NFS Protocol (Network File System)



NFS v3 – useful in machine rooms (UDP, efficient, good for close by) — lowlevel, packet oriented protocol
v+ – useful in internet (TCP, a bit slower, good for wide area)

protocol
C request — inode # on server — w/other stuff — no "/" — "bash"
LOOKUP(dirfh, name) ⟶ S response: fh + attrs — like stat/statat
fh: inode, etc. — attrs: timestamp, ownership, etc.

CREATE(dirfd, name, attr) ⟶
MKDIR(" ") ⟶
REMOVE(dirfd, name) ⟶ status
READ(.....) ⟶
WRITE(.....) ⟶

~~CLOSE~~

NFS Filehandles ← fixed size, uniquely identify a file on the server, persist during server reboots
- survive reboots

(use inode # on server)

"Stateless server"

servers RAM is just a cache, only for performance

Lookup(9312631, "bash") → server
(part of NFS server is inside kernel)
("fdrive)

Advantages

C1: rename

C2: write(fh, ...)
write(fh, ...)

this works

(mv a b; & cat) < a

What if?

(rm a & cat) < a if one client removes and then another one reads from it the read will fail.
(errno == ESTALE)

Exception both apps are running on the same client

(rm a & cat) < a name not known to user?

kernels unlink("a") → rename("a", ".nfs937291")
When cat finishes unlink(".nfs937291")

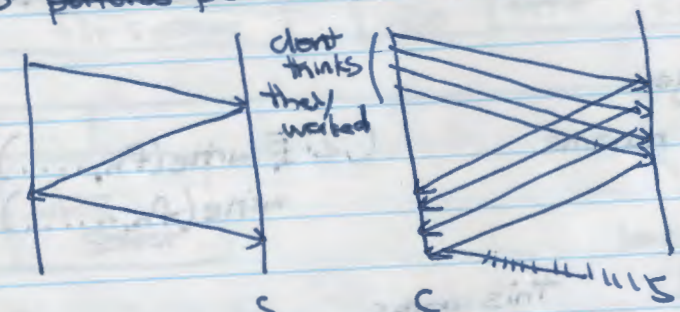
Suppose

1. (rm a & cat) < a
2. rm .nfs*
3. echo foo > bar
   gets inode 937291

Step 1 will start reading from the new file!

Really this number is partially a inode# and a serial #, so the chances of reusing the inode # is next to 0.

NFS ~~perforce~~ performance issues



close(fd) wait for all outstanding requests, report an error if they don't come at night

same file

C1	C2	
write (D1)	read( )	
write (D2)	read( )	might see D2 first (if D1 gets lost)
close ( )	recd( )	
open ( )		

We don't have read-after write consistency
We do have open-after consistency

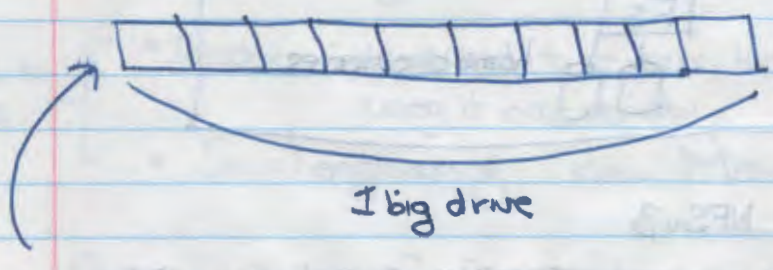
File System Robustness
1) Crashes (log-based filesystem)
2) Media Faults

Media Fault Handling

RAID

Redundant
Array
Independent
Disks

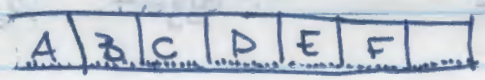
concatenate 10 cheap drives



1 big drive

RAID 0

mirroring RAID 1



RAID 4

6 datadrives

parity drive
A xor B xor C
xor D xor E xor F

If parity drive fails, remake it.
If B fails, take A^C^D^E^F^parity = B
We can always recompute.

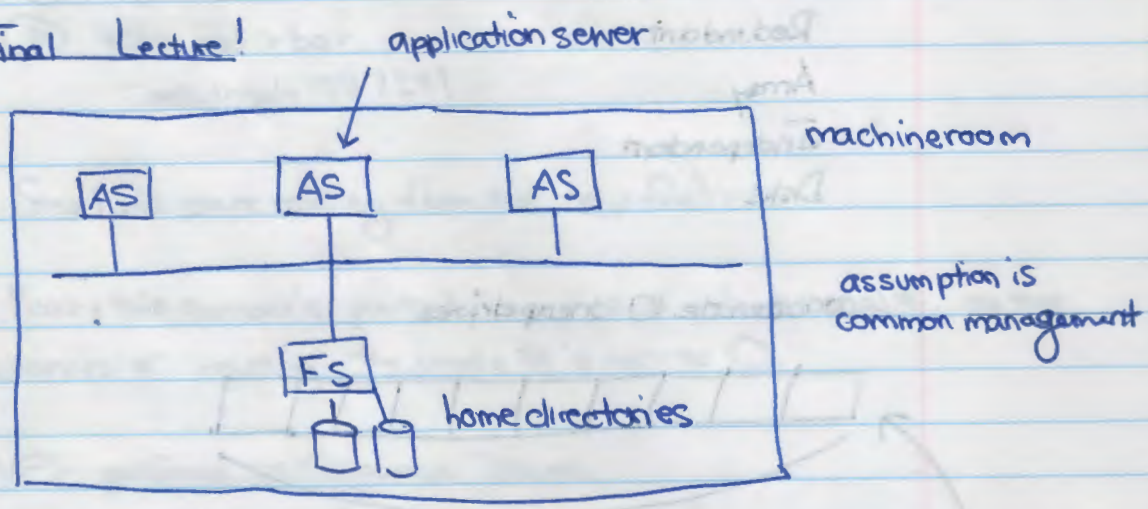
RAID 5

split parity to even out the hotspot.

Yet, raid 4 is more oftenly sold? Why?

Well, it is easy to add a drive. It's very cheap to do so because we need only 1 drive with 1 XOR operation!!! :)

Final Lecture!



through NFSv3

requests and responses contain user IDs

10932 (eggert)

/etc/password (Machine 1) (Rogue machine)

eggert: 10932 eggert: Ø ← superuser by convention

Security Issues:

1. Rogue application servers
2. Could be a diff machine that has a different userID (say physics and CS both have an "eggert")
3. Rogue may be able to snoop packets on a shared network
4. Packet insertion, send a packet that claims to have data that it doesn't really have (instead, it contains some other data

NFSv4 adds support for:

- data encryption ← (downside, costly)
- user names ←

OS Security

Some example issues

- Supreme Council of Virtual Space
  (>50% of world's websites are blocked in Iran)

- November 2014 attack on Sony
  (mostly a network attack, but fundamentally an OS bug)

Threat Modeling and Classification

Use logs to fix

- Insiders → supposed to be users, but they break it by using it inappropriately.
- Social engineering (Kevin Mitnick)

Can we use the logs for file systems and ~~res~~ reuse them for these security logs!?

- yes, if logs also contain security metadata
- no, garbage collection will eventually ~~se~~ erase the evidence.

- networking
  - virus / drive-by-download (DBD)
  - DoS: denial of service → keep bogus requesting to prevent server from accomplishing any useful tasks!!!
  - buffer ~~overflow~~ overruns
- devices attacks
  - USB that installs a bad boot block on a ~~USB~~ computer.

General Security Functions

- Authentication
  - passwords
- Integrity
  - checksums
- Authorization
  - access control list
- Auditing
  - Logs

Things to Consider:
- Efficiency
- Correctness

How to Authenticate:

Based on:

1) Who they are (Who the principal is) ← retinal scan, lol?
2) Something the principal has ← key card
3) Something the principal knows ← password.

Weakness: bootstrapping
How do we obtain this stuff in the first place???

Weakness: eavesdropping (mostly a problem for #3)

Two-factor authentication: do TWO of those things

Authentication tries to prevents masquerading
it can be pretty elaborate

external authentication vs internal authentication

can be expensive to help internal

want cheap/fast

give you a badge

check the badge

login : eggert
password: [ ] } ext
$cd foo } int
requires search permission

chdir("foo")

inode entry
[| perms] cheap and fast

Issues

privilege escalation
gaining more rights than you should
default permissions
generally 022 (if you're trusting)
revocation of permissions
if i'm in a spot and lose perms, it doesn't yet kick me out.

Authorization

Access Control
• Only some kinds of accesses are allowed.

Goals	Nongoals
Privacy	Denial of Service Attacks
Integrity	
Trust	
Sharing	

ACLs (Access Control Lists)

For each principal p
  for each object o
    for each access type
      is a(p,o) ok?

This is <u>super</u> costly

For seasnet

10,000 p
500,000 o
4+ a

} 2×10^{10} bits, 2 gigs

How do we shorten this a little?

- Look for clustering of perms and look for short representations → shorten to a few KB

- for each object we ~~didn't~~ maintain an ACL

P	list of a's (or bitmask)
1	
2	
3	
4	
5	
⋮	

$ getfacle → shows allowed access controls

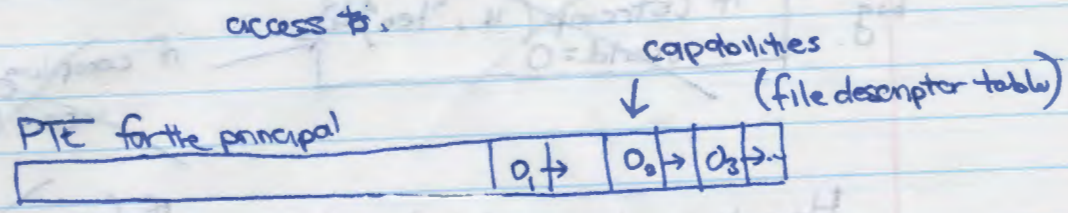
usr: rwx
group: r-x
other r-x

$ setfacl
-r -m -group: tas ; rwx
updates so ~~to~~ group also has access

any user can run this

ACL's record, for each object, which ~~principles~~ principals have access.

Capabilities – record, for each principal, which objects they have access to.



I look in the PTE for the principal.

Suppose we used capabilities on a network
Every file has a hard to guess (say 264 bit) number. You get access if you know the #
+ I can delegate access to another process b/c I know the number.

Trust
"Reflections on Trusting Trust (K. Thompson)
- Most programs are not trustworthy/trusted
- However, there must be some trusted programs
  login su sudo

How does this work? How do we differentiate?
We have a setUID bit. If this bit is set, when a program runs, it runs as if it were the owner of the file.
su, sudo, etc. are started by root.

TCB: trusted computing Base: we rely on these to work
✓ hardware ✓ su ✓ login
✓ Kernel ✓ sudo

untrusted
TCB

untrusted
- emacs
- firefox - gcc

Turns out, Thompson's paper shows that this doesn't work.

login.c

bug

```
if (strcmp(u, "ken")==0)
    uid=0
```

gcc.c

```
if compiling("login.c")
    insert bugs
```

```
if (compiling("gcc.c"))
    insert this bug
```

How do we know if this has happened already? ....we DON'T.