Lecture 1 1/4/2016

Red Star 3.0 (from fedora)
- automatic watermarking (steganography)
- tamper-resistant OS modification

http://web.cs.ucla.edu/classes/winter16/cs111

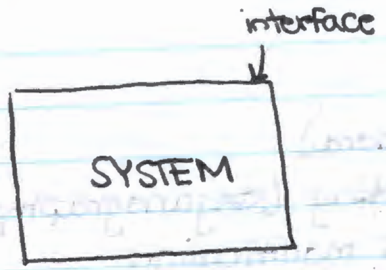
Course organization and grading

17 lectures
1/9 1 midterm (during lecture) 100 minutes
2/9 1 final exam 180 minutes
1/3 4 labs (teams of 2) 1/12 each
shell, kernel hacking, file system, networking
[redacted] miscellaneous
2/15 individual minilabs 1/15 each
scheduling, virtual memory with partner
1/12 1 design problem (lab extension), requires written report
1/15 1 2-3 page paper on an operating system topic
1/20 scribe notes (groups of up to 4), webpage for lecture
Due week after lecture

Open book/notes/assignments on the exams :)

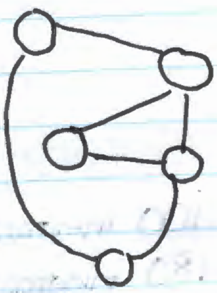
<u>What's a System?</u>
OED original (1928)
I. An organized or connected group of objects
II. A set of principles, etc., a scheme, method
from Greek σύστημα' - organized whole, government, constitution, a body of people or animals, musical interval

roots:
- "set up with"



ENVIRONMENT

Book's definition:
- A set of interconnected components that has a specified behavior observed at the interface of its environment.



Example of subsystems, not seen from environment

Operating System – American Heritage Dict. (2000)

software designed to control hardware of a specific data processing system in order to allow users and application programs to make use of it.

Encarta (2007)

master control program in a computer

Wikipedia v698216816 (2016-01-04)

collection of smaller programs and software that is used to control and operate the computer system

....

Some major missing issues...

- resource management
- reliability + error handling
- security

circa 2008 hardware

$ ls -l big $\sim 10^{19}$

-rw-rw-r-- 1 eggert faculty 9223372036854775000
-Oct 6 11:31 big ← ZFS
Zeta File System

$ grep x big 10^{21} bytes/second
$ time grep x big
real 0m0.009s 10^{-2} s 10^{22} bits/second

http://what-if.xkcd.com/31/
Internet bandwidth ~~167~~ 167 Tb/s $\sim 10^{11}$ bits/second

This means it's possible to physically move data faster than it is to send it through the Internet, given a sufficient size

$1.2 million/gallon → all freight trucks in the U.S.
~ 0.52 ~~bytes~~ bits/second

ZFS has intensional files - the data is not stored on the disk, rather, it is a description or a program that represents the data.

- It also has extensional files, which are interpreted directly.

The file above was created with...

$ truncate 922..........000 f

So how does grep work so quickly?

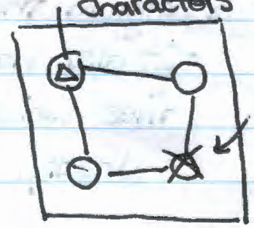
If it is an intensional file, then it just reads that.
Hence, in this case grep quickly realizes that there is no x and returns almost immediately.

Aside....
This is useful for virus scanners to exploit. It wants to skip parts that cannot possibly be viruses. That way, it catches them faster.

<u>Problem Areas in OS Design</u>

- Incommensurate scaling
  not everything scales at the same rate
  cost/unit ↓ economies of scale (Adam Smith, pin-factory)
  cost/unit ↑ diseconomies of scale (star network)
  Things break as you grow with diseconomies.
  Economies can cause waste.
- Emergent properties
  larger systems have properties that smaller ones didn't
  • (Tacoma Narrows bridge) ← resonance frequency
  • UCLA campus network → Napster
- Propagation of effects
  pathways for propagation are "more effective"
  in a digital system, one component change can be very profound

international characters 天 ← \309\257



file system ← C:\Windows\foo.txt should become 天.txt

Turns out, the coding was shift-JIS where first byte is all 1's, and the second could be anything.

Change Δ breaks part X.

However: the 2nd byte could be the ASCII '\', and the file system will not work even though it wasn't changed.

SOLUTION: use UTF-8.

- Trade Offs

Waterbed Effect - if we push one part of the waterbed we must send the water somewhere else.

$\boxed{\text{ex}}$ if we used the SOLUTION, we would need to represent Japanese characters with ~~base~~ 3 characters.

$\boxed{\text{ex}}$ Too easy to break into UCLA registrar

passwords too easy to guess

SOLUTION: keycards.

Tradeoff: extra work for maintenance

Benefit: Security.

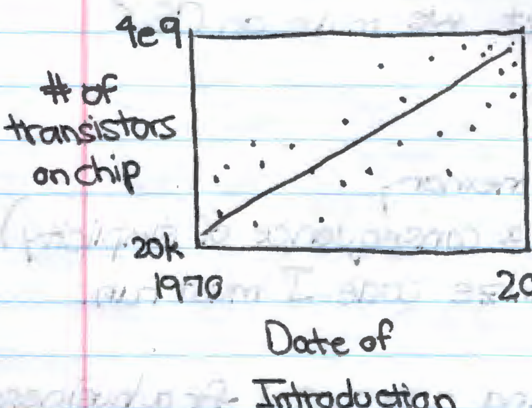
$\boxed{\text{Lecture 2}}$ 1/6/2016

Today's topics:

- a bit more philosophy
- how not to make an OS

Complexity

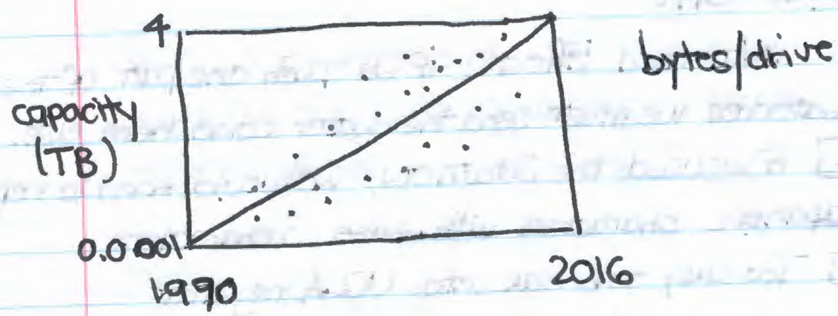
Moore's Law - # of transistors doubles (that are economically viable) approx every yea



Log scale! bits/chip

However, in recent years the exponential growth has begun to trail off.

Kryder's Law secondary storage capacity



However, complexity is increasing at a higher rate

Why Exponential Growth?

use to design the UNIVAC II

UNIVAC I (slow, safe, hand designed)

UNIVAC II

$$\frac{d(\text{technology})}{dt} = K * \text{technology}$$ which comes out to be exponential.

How Not to Make an OS

First off, why shouldn't we make an OS?

- simplicity
- performance
  - speed - memory
- Reliability (as a consequence of simplicity)
- security (minimize code I must run.

Consider that I am submitting a proposal for a business on Friday and I want to make my own system (I'm paranoid)

no OS
Application – count of words in an ASCII text file.
(bytes '\001'–'\177')
'1'–'127'

Interface
① Power switch

39716

~10 year old desktop
Core i3-4160 (3 MiB cache,
4 GiB dual channel DDR3
1 TB hard drive, SATA, 7200rpm
Intel 4400 graphics

BIO's – where is program, and where is text file?
file is here → in adjacent sectors

Traditional disk drive, 512 bytes each sector

Cycling power clears CPU, cache, and RAM

Bootstrapping Problem – if we know ..., then....., but where do we start? We need to have something to work with in order to get started

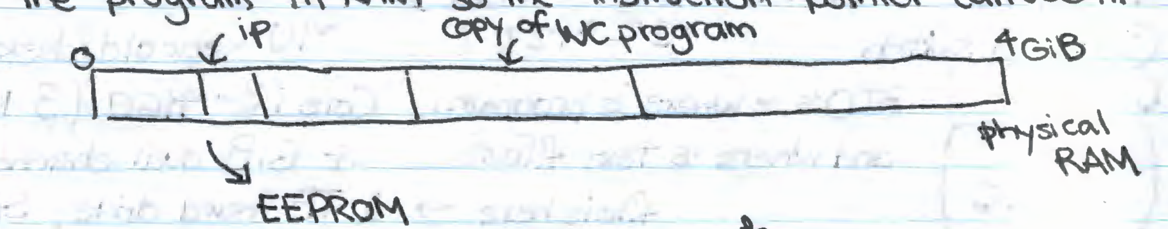
built into hardware
instruction pointer – 0xffff0 $2^{20} - 16$

access a special region, 0xfff0

ROM region ← hardwired by manufacturer to do what we want it to do.
-or-
PROM (programmable read only memory) – we program, if we got the manufacturer to produce it for us. Most paranoid approach.
-or-
EEPROM (electronically erasable PROM) – called "nonvolatile memory"

Downsides – might render computer unbootable if bad erasure.
Erasure takes time and effort

We want the word count program to be on disk, and use the EEPROM to get to it somehow. The problem is, we need the program in RAM so the instruction pointer can use it.

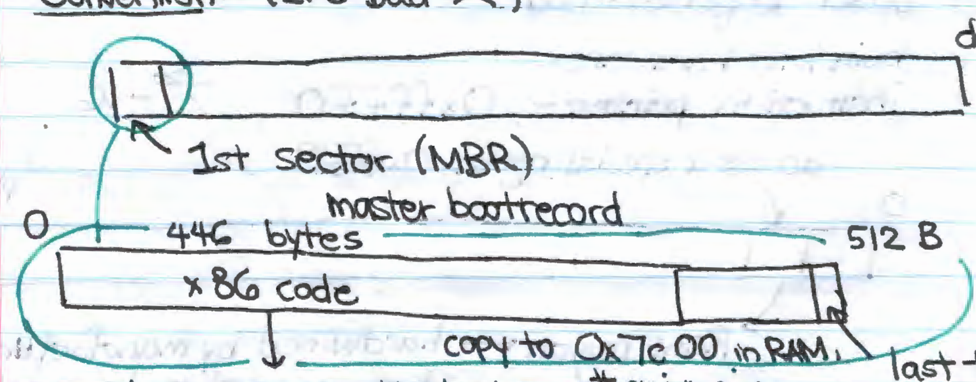


location of wc program, size of wc * + loading program

Problem: * still program dependent. We don't want the EEPROM to have to change.

and so the solution is to have the size and location to be the same for every program in EEPROM.

<u>Convention:</u> (It's bad :( )



<u>convention:</u> last two bytes eventually have signature

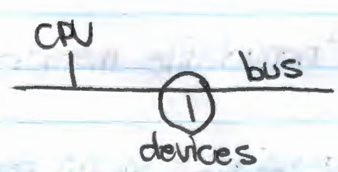
0x55 0xAA

little endian

0xAA55

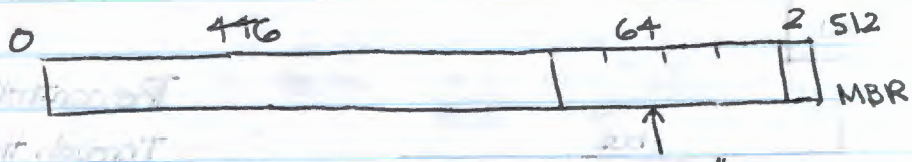
- hardware sanity checks
- some CPU checks - checks RAM
- checks for devices in order

finds 1st device with an MBR



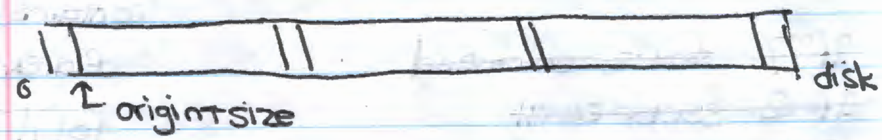
Firmware (EEPROM) → MBR (first sector on disk) →

do whatever to code so long as it is in first 446 bytes



Each entry will tell you a bit about the rest of the disk, with sector counts

array of 4 16 byte items, called the partition table, which tells how the disk is divided.



32 bit sector counts for offset ~~partitions~~ size of partitions

other info: type byte, partition types, isBootable?

Chain Loading - the firmware boots by reading the MBR, which reads the first bootable ~~partit~~ partition (called a volume boot record) and so on.

firmware → MBR → VBR → kernel → programs, more programs, even more programs

MBR: OS-agnostic; VBR: OS-specific; kernel: (many sectors)

We need a subroutine to read data from the disk.

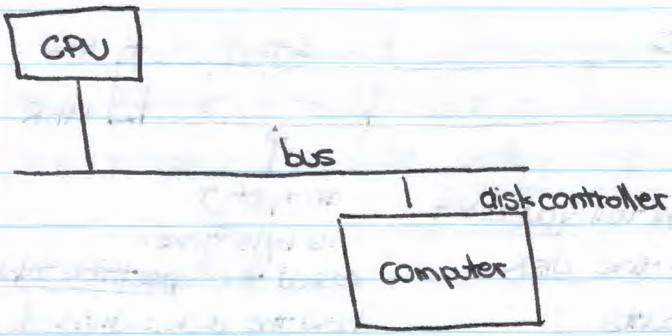
needed by: programs, MBR, and the firmware

V1 cleaner in v2, pg 11.

Snippet 0

sector # → s; memory address → a; ~~int~~ char*

```
void read_ide_sector(int s, char* a)
{
*  while ((inb(0x1f7) & 0xc0) != 0x40)    goes to disk controller to get data from disk controller, is slow
     continue;                            retrieve data from the bus address
   outb(0x1f2, 1);  ~~outb(0x1f~~
   outb(0x1f3, s & 0xff);   //low order 8 bits
   outb(0x1f4, s >> 8);   → outb(0x1f6, s >> 24);   0x20 is read
   outb(0x1f5, s >> 16);    outb(0x1f7, 0x20)
```



The controller communicates through the buses via their registers

0x1f7 → status register 1B contains info about state of disk's controller

$\boxed{0 \mid 1 \mid \quad\quad\quad}$

controller is available!

~~1f7 status, command~~
~~1f2 sector count~~

1f0 read data
1f2 sector count
1f3 — low order byte
1f4
1f5
1f6 — high order byte
1f7 status, cmd

Sector # — 1f3 … 1f6

```
*
While ((inb(0x1f7) & 0xc0 != 0x40
        continue;
ins l (0x1f0, a, 128);    ← 512 / sizeof(int)
}
```

✓ We can have a copy in firmware at some location

-or-

✗ it can be in the master boot record

-or-

✗ it can be in the word count program

Isn't this wasteful? Let's just use one copy

BIOS = basic input/output system

downside: this must be built into the system

V_2

Final ①

sector #, memory address

```
void read_ide_sector (int s,  char* a)
                      4byte   4byte
{
   while((inb(0x1f7)& 0xc0) != 0x40)
        continue;
   // While not free, continue;
   outb(0x1f2, 1);        // Tells that sector count is just 1
   outb(0x1f3, s);        // Write low 8 bits from s.
   outb(0x1f4, s>>8);     // Write next 8 bits from proper spot in s.
   outb(0x1f5, s>>16);    // Write next 8 bits from proper spot in s.
   outb(0x1f6, s>>24);    // Write next 8 bits from proper spot in s.
   outb(0x1f7, 0x20);     // change status to reading the sector
   while((inb(0x1f7) & 0xc0) != 0x40)
        continue;
   insl(0x1f0, a, 128);   // Read 128 longs worth of memory into RAM
                          // starting from a, the mem address
                 512/sizeof(int)
}
```

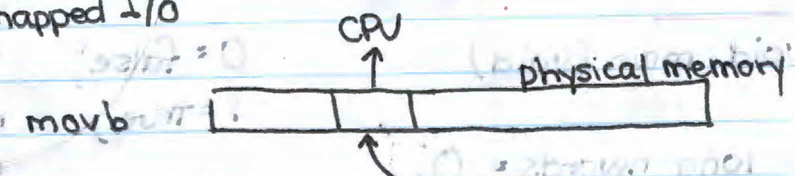
(status & 11000000)
!= 01000000

01 . . . .
sector is free

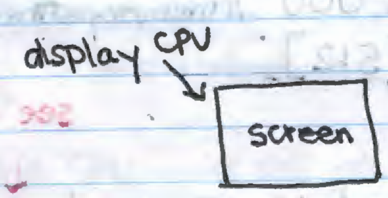
Lecture 3

1/11/16

To output to screen, not programmed I/O (PIO) but memory mapped I/O

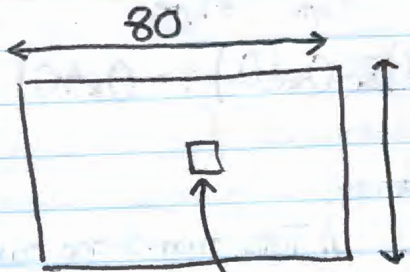


not RAM in ordinary sense, but is actually displayed.



Standard:

Model screen as 80×25 grid of characters



0x B8000

4000 bytes = 80 · 25 · 2

16 bits to represent
low order 8 bits is ASCII
high order bits = graphical appearance

V1 cleaner in v2 pg

64 bit integer

figure out where onscreen to start

Snippet 1

void output_to_screen (long n)
{
short ~~long~~ *p = (long*) ~~0xB8000~~ 0xB8000 + 80·25/2 − 80/2;
}

~~*p = x;~~ // Won't work!. Output will correspond to early bytes on screen.

otherwise blank screen if 0.

do ~~while (n != 0)~~
{ *--p ~~*p++~~ = (7 << 8) | (n % 10 + '0');
graphic setting 0x700
reversed order!
n /= 10;
} while (n!=0);

V1

Snippet 2

void main (void)
{
long nwords = 0;
bool inword = 0;
int s = 1,000 //memory other stuff isn't using
char buf [512];
for (;;)
{

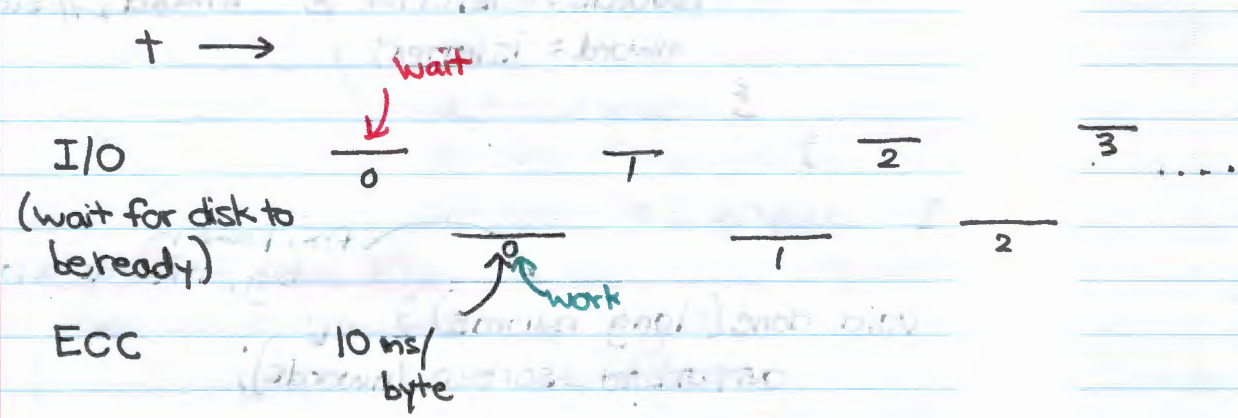
0 = false; 1 = true;

'\0' = 0 means EOF, for our system

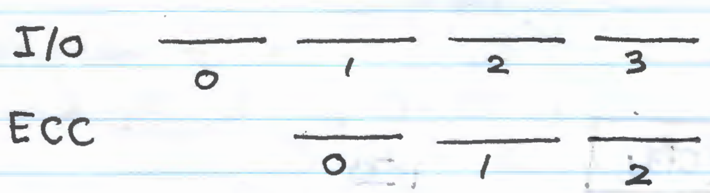
see Final Snippet 0, pg 11

This adds complexity!. Furthermore, since we're looking at the data in RAM anyway, it is sent to the CPU anyway, so DMA isn't as useful.

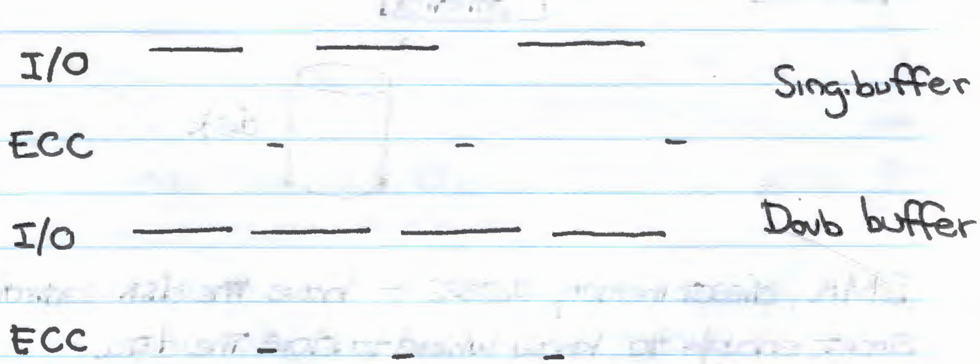
However; consider a crypto app; very CPU intensive



How do we improve this? Double buffering, where we load one buffer while working on the other one.

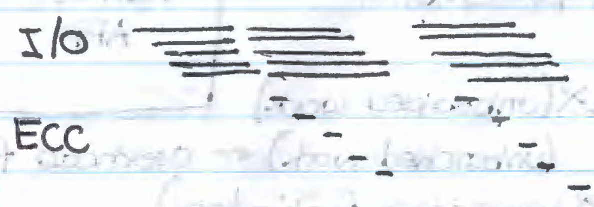


However, if I/O and computation is too high in discrepancy using double buffering is not as worthwhile.



Triple buffering is good when we load, process 1, process 2, etc.!

Multitasking (with multiple processors)



Load multiple from I/O then process one at a time while loading next set, if processing is faster by a significant margin.

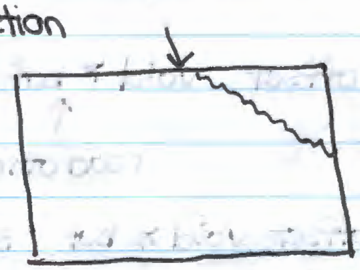
How to scale up these programs.

- Fancier performance tricks
  without rewriting applications
- Multitasking
  without rewriting applications

Modularity

cost of maintaining a module of N lines of code is $O(N^2)$

Abstraction



Finding natural divisions in the programs, that makes the program is easier to manage.

How do you measure the quality of modularity + abstraction?

<u>Simplicity</u> (ease of use)
(ease of learning)

<u>Robustness</u> (tolerance of errors, large inputs) harsh conds

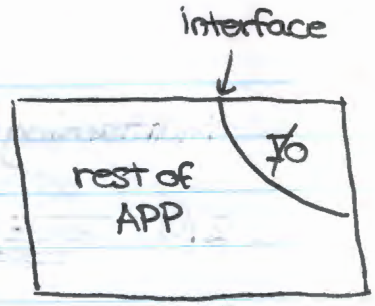
<u>Performance</u> modularity costs this, minor costs unavoidable, avoid major costs as often as possible.

<u>Flexibility / Lack of Assumptions / Neutrality</u>

Consider....

char * readline (FILE *f);

BAD DESIGN for O.S.



Why?

1. Performance × (unbounded work)
   (unbatched work) ← overhead for small lines
2. Robustness × (apps crash for big lines)
3. Neutrality × (forces particular line ending convention on the app)
4. Simplicity ✓ :)

Let's now examine read_ide_sector...

```
void read_ide_sector (int s, char *buf)
int read_ide_sector (int s, char *buf)
int read (int byte_offset, char *buf, int bufsize)
```

improve robustness by reporting potential error msgs. → `int read_ide_sector`

↑ more general, no need to worry about exact sector size

```
int read (int byte_offset, void *buf, int bufsize)
```

↑ read other things besides chars

```
int read (off_t byte_offset, void *buf, size_t bufsize)
```

~~int~~ off_t // long long on x86-64

↑ increase max bufsize, make this portable
// size_t is unsigned long on x86-64

```
ssize_t read (off_t byte_offset, void *buf, size_t bufsize)
```

↑ # bytes read or -1 if error

Other improvements....
have it return error code? Not the convention, sadly.

which file? (opaque file handle) where did the byte-offset go?

- ssize_t read (int fd, void * buf, size_t bufsize);

BIG IDEA OF UNIX

everything outside the program is a file

(disk, flash drive | network, mouse, keyboard, display)

random access device | stream device

doesn't need/use the offset, hence its removal

but we lost random access! How do we get it back?

- size_st lseek (int fd, off_t ~~value~~ where, int flag);

Corollary, the OS records current file positions
starts at 0, after reading n we add n.

- pread ...
  add complexity to get performance

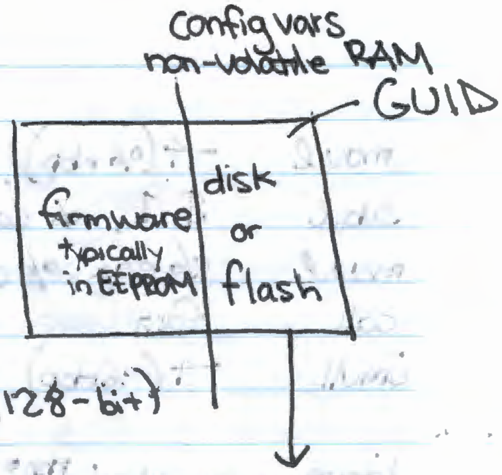
<u>Mechanisms for Modularity</u>

- Function calls
- + simple and well understood
  + reasonably fast
  − things can go wrong
- Client-server (works, but.... slow)
- Virtualization

Lecture 4:
OS Organization

1/13/16

Booting UEFI



GUID: globally unique ID for disk partitions (128-bit)

standardized — GUID partition table (GPT)

Standard layout in each partition:

FAT 12
FAT 16
FAT 32

kernel with known name

Firmware is in charge until booting. This is enforced organization.

How to enforce modularity after booting
- function calls (terrible way?)

```
char buf[2000];
read(3, buf, 1000);
```

```
int fact (int n)
{
    if (!n)
        return 1;
    return n * fact(n-1);
}
```

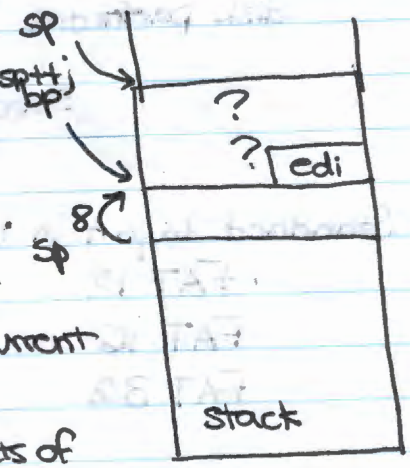
unoptimized:

```
fact:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    cmpl    $0, -4(%rbp)
    jne     .L2
    movl    $1, %eax
    jmp     .L3
```

push

Recall, %edi is parameter
%rax is return value

```
.L2
   movl    -4(%rbp), %eax
   subl    $1, %eax
   movl    %eax, %edi
   call    fact
   imull   -4(%rbp), %eax
.L3
pop  leave   rsp=rbp; rbp=*rsp++;
     ret     rip = *rsp++;
```



From start to finish:

- Decrement by 8
- Set %rbp to %rsp's current location
- Move up 16 bytes, 2 sets of junk inside.
- low order 4 bytes of %rdi is set to where %rbp is
- Perform comparison, if %edi is 0 go to L3, store 1 in %eax (low 4 bytes of %rax), then return
- Otherwise we jump to L2, first two commands get n-1 into %eax.
- movl %eax, %edi moves n-1 %edi
- Then we call factorial
- We then multiply into our %eax register which will contain the answer.

Things that can go wrong:

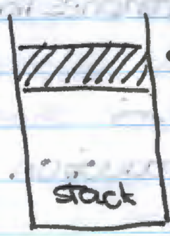
fact (INT_MIN) ← overflow, attempt fact (INT_MAX)

fact (50) ← will only give us the bottom 32 bits (4 bytes) of the correct answer

However, each time we call it we will put up 12 bytes, eventually obliterating the rest of the data structures (or dump core, if we're lucky).

We can catch this but they have some downsides...

- Adding checks will greatly slow it down
- Guard page,



← if it reaches here we will trap

Suppose in the earlier machine code we call fact2 instead of fact. What can fact2 do to mess up fact?

- Modify stored values in fact (activation record)
  - make it return to the wrong spot
  - modify the return value
  - Cause an infinite loop.
  - Jump elsewhere in the program

Can fact mess up fact2?

- Put a random value into %edi
- movl $0, %esp ← will have fact2 enter forbidden zone as soon as it tries to push to the stack
  jump fact 2

We have soft modularity.
- it doesn't scale to large applications

We want hard modularity, where a failure in a module won't tank the rest of them.
- virtualization — Unidirectional hard modularity
- client-server — Hard modularity in both directions

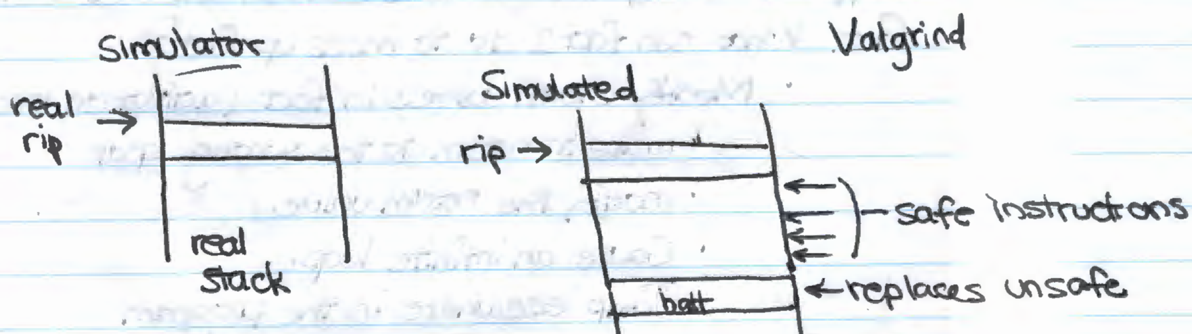
Simplest way to get virtualization

write a simulator for the machine the app runs on
carefully checks instructions
- halt → return
- outofrangeaddr → return

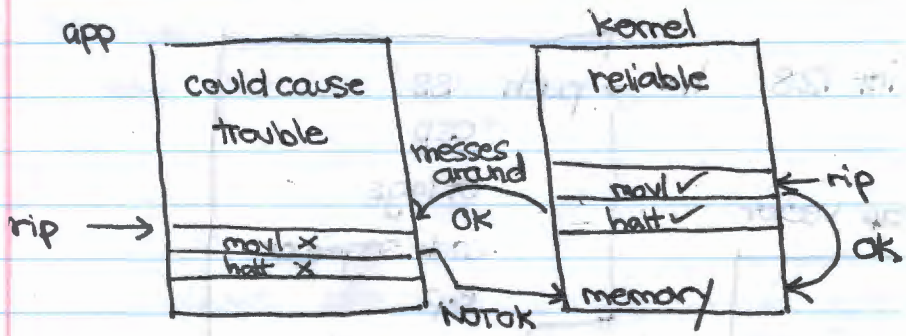
count instructions
- when it reaches limit → return

Unfortunately, this method is slow. :(
In fact, it's usually too slow for production

We need a virtualizable machine, a virtualizable processor



There's one big problem left... it's still too slow!!!



Protected Transfer of Control

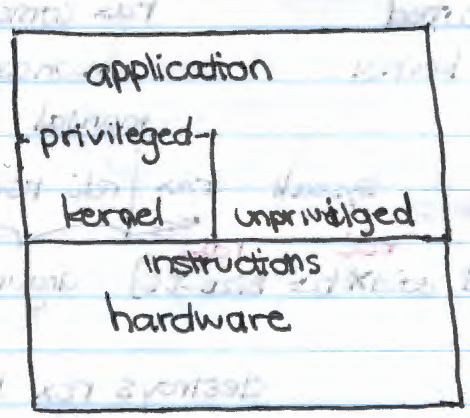
should make these rare for this model to work

~~dangerous~~ privileged instructions in the application do not execute.

rip becomes in kernel

We have a privileged bit, which is set to 0 in the app. When it encounters a privileged instruction, it passes to the kernel, sets the privileged bit to 1, and does what it needs to do.

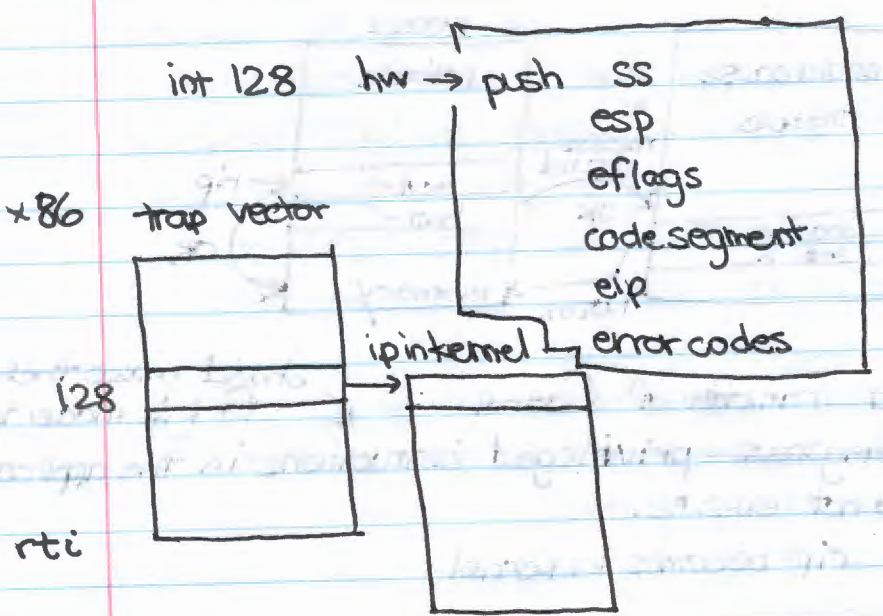
Layered architecture



Classic way to enter the kernel:

execute a privileged instruction with a standard convention

this is → int 128



rti : int as ret : call
expensive cheap

Nowadays, for the x86, x86-64, we have a special syscall <u>machine instruction</u>:

- When executed
  - enters kernel
  -

rax contains syscall #'s, which can be found in a manual

syscall rax | rdi rsi rdx r10 r8, r9 arguments

```
                rdi     rsi        rdx
ssize_t read(int fd, void* buf, size_t s)
{
    asm;
    deal with errno;
}
```

destroys rcx, r11
result into rax
-4095 -1 means failure -errno

So if rax is wrong then we send result to errno and return -1.

Mechanism works

Now, what does the user see?

app dev

Model: processes: programs running in a virtual machine atop an operating system

Creation pid_t fork(void); Clones current process
returns 0 in child
returns child's pid in parent
returns -1 on failure

Destruction void _Noreturn _exit(int);

never returns — bypass normal cleanup

Calling process immediately stops running, but not gone!

Process object

exit_status	pid
fd_table	

So this isn't the true way to destroy a process

status — flags — 0, NOHANG

pid_t ~~int~~ waitpid (pid_t , int* , int)