Writing a Kernel

or

Lies Operating Systems Tell Us

Memory Is All the Same?

```
Int[,] bigArray = new int[20000,20000];
for(int i = 0;i<20000;++i){
   for(int j = 0;j<20000;++j){
         bigArray[i][j] = i+j;
 }}
```

Is as fast as

```
for(int i = 0;i<20000;++i){
   for(int j = 0;j<20000;++j){
         bigArray[j][i] = i+j;
 }}
```

Right?

Dirty Lies, Memory Access isn't Equal

First example: 6.3 sec

Second example: 9.7 sec

30% Difference!

Why?

Cache Locality

Virtual Memory Implementation

Page Tables

Bears, Oh My

WTF is Virtual Memory?

Every process gets to pretend it's the only process running on the machine!

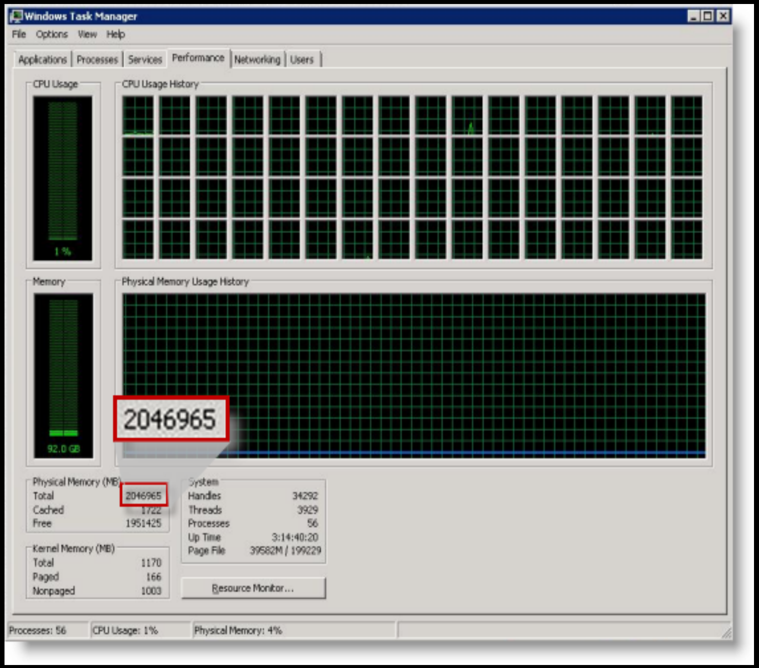
Kernel does lots of voodoo under the hood to keep up the illusion.

TLB

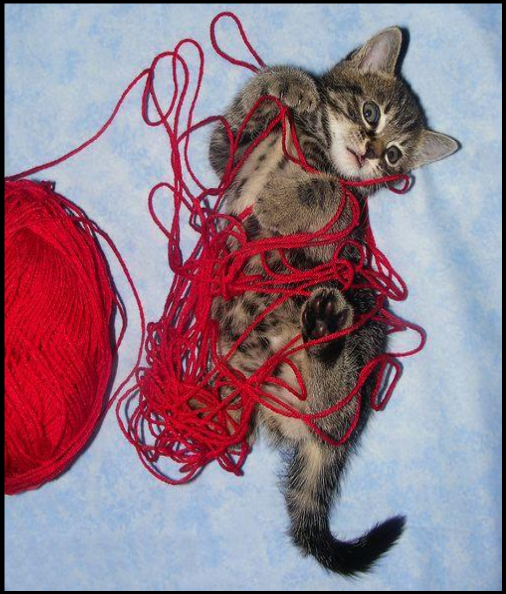
Paging

MultiLayered Memory

VM



Threads/Processes More Lies



What's a process?

At the base level, one or more threads worth of code executing in a distinct virtual memory context.

What's a thread?

At a base level, a piece of code operating in a shared virtual memory context.

Reality is more layered:

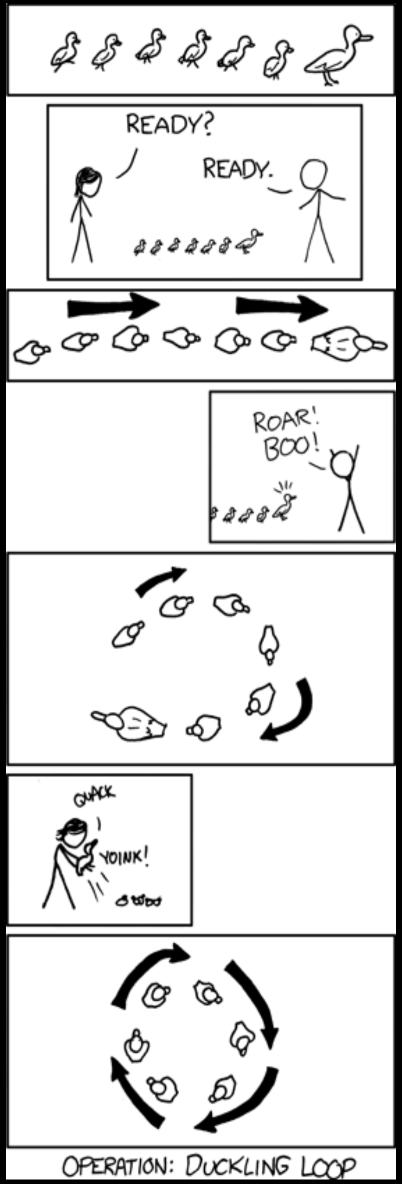
- User Threads Vs Kernel Threads
- VM Threads (thread pools, i/o threads,etc)
- Apartments, cross boundary threads,Fibers, etc

Illusion is Great, but Leads to Chaos

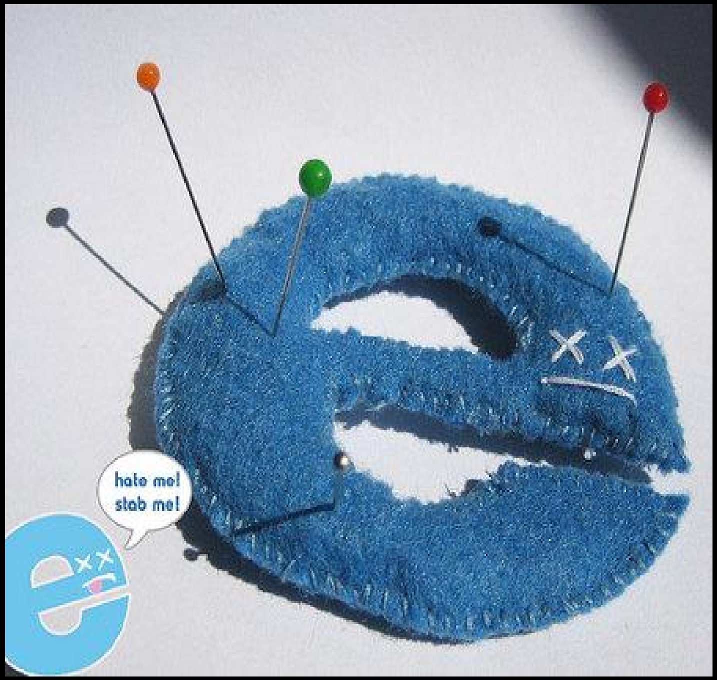
Race conditions

Live Lock

Dead Lock



So, what is the Kernel's Voodoo



Process/Thread Management

Managing creating, deleting processes and kernel threads

Managing Virtual Memory

Scheduling threads to run

Hardware Access and Performance

Disks, cards, etc.

Drivers

Network Stack, Virtual Machine Host?, Webserver?!

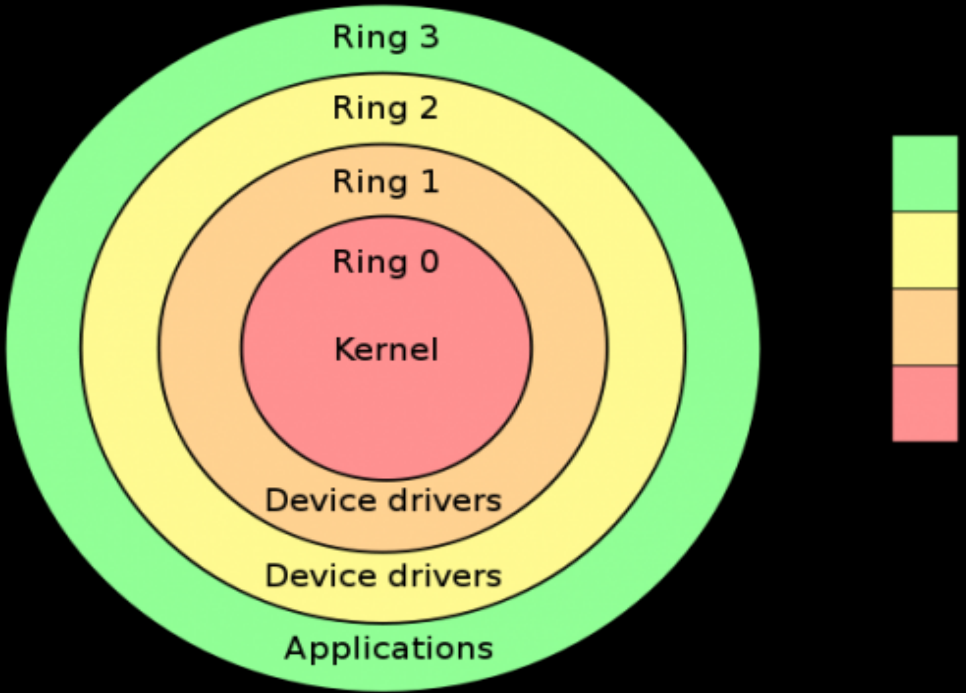
Protection Rings - Other Voodoo

No, more like:

Built into the processor itself.

Different Privileges

lower ring, bigger the tinfoil hat you wear when writing/installing



Tangent: Secure Boot

All operating system code would need to be signed by a certificate authority to boot, sort of like SSL

Gives some confidence that ring 0 hasn't been breached by malicious bootloader, or bad kernel image, etc

Major problem for running Linux, BSD, or any other experimental OS on hardware because the certificates have to ship with the hardware!

Writing Your Own

Mine was really simple, in the unix flavor.

Fork,

Thread_Fork

Yield

Exec

Video and Keyboard Drivers!!

Check Out That Sweet x86 Goodness

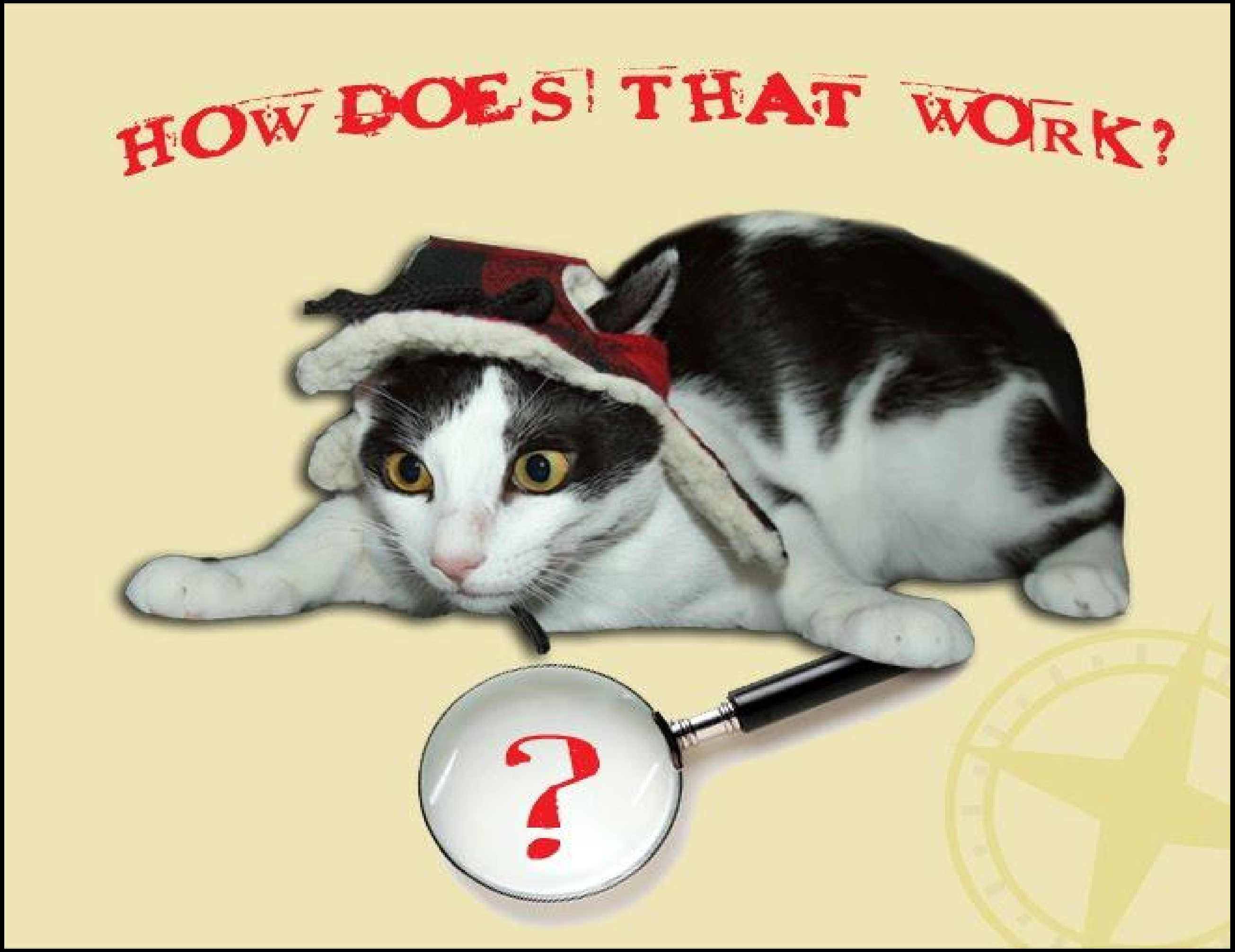
~ six+ weeks development for two people.

Very, very little sleep.

Many Weird and wooly bugs, especially once multiprocess/multithreading started working.

Still lots of bugs

So?



Note, *really* technical from here on

We're going to study some small bites

Exec is 650+ lines of C and would take multiple hours by itself. Plus I'd have to remember...

Warmup: How to build a mutex?

C function supplied us with two int *

```
.globl k_xchange

k_xchange:
        MOVL     8(%esp),%eax   /*get the int to exchange*/
        MOVL     4(%esp),%edx   /*Get the address of the pointer to exchange*/
        XCHG     (%edx), %eax   /*exchange the two */
        RET
```

Use magic processor instruction, atomically swap the contents.

The only tool needed to build a mutex. Now we have the basic component of thread safety. From mutexes come semaphores, reader/writer locks, etc

Fork it Baby

Save our current state so everything gets duplicated nicely

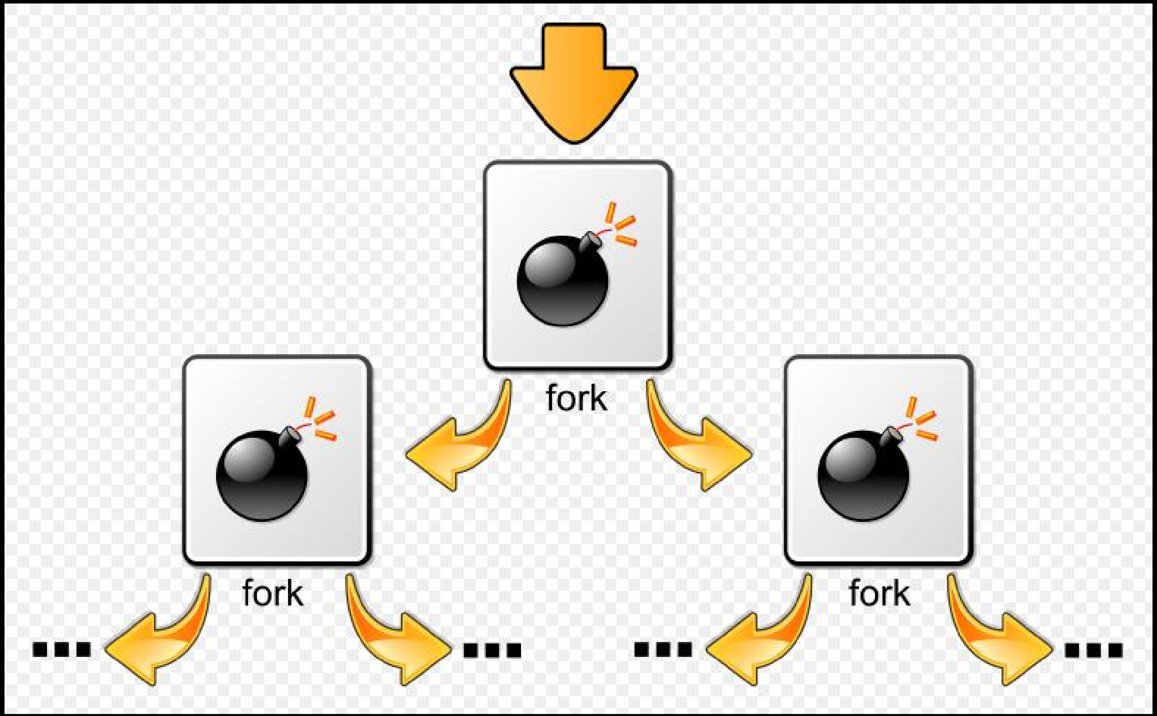
```
fork_handler:
    PUSHA                           #push all general purpose registers
    PUSHL    $-1                    #push a -1 to signify to context switch out just forked
    PUSHL    %esp                   #save the esp
    CALL    save_esp                #saves the previous esp
    POPL    %eax                    #remove the esp
    CALL    continue_forking        #continue the fork p
    POPL    %edi                    #We pushed -1
    MOVL    %eax, 28(%esp)          #Overwrite esp with the new tid
    POPA                            #Pop all registers
    IRET                            #return from syscall

thread_fork_handler:
    PUSHA                                   #save all general purpose registers
    PUSHL    $-1                             #push a -1 to signify just forked
    PUSHL   %esp                            #save the esp
    CALL    save_esp                        #saves the previous esp
    POPL    %eax                            #remove the esp we pushed
    CALL    continue_thread_forking #continue thread forking process
    POPL    %edi                            #remove the negative -1
    MOVL    %eax, 28(%esp)                  #insert the new tid into the eax position
    POPA                                    #restore the registers
    IRET                                    #return from interrupt handler
```

Go into C and do the actual copying and queue up the new thread.

Note how similar, main difference happens in what gets copied.

Remember later, one in, two out

Still With Me?



Excellent, we can now forkbomb the kernel.

Changing who is running

Remember earlier in fork we pushed -1 ?

```
.globl context_switch

context_switch:
    PUSHA                          /*Save the general puprose registers */
    PUSHL   $0                     /*This is a normal case, not the result of a fork */
    PUSHL   %esp                   /*put in the argument for save_esp */
    CALL    save_esp               /*Save the ESP into this processes TCB */
    POPL    %eax                   /*Remove the argument for save_esp*/
    CALL    context_switch_out     /*start the new context */

```

These 8 lines of assembly took multiple days of hard thinking to get right.

Lets make a thread run

```
void context_switch_out(){
    assert(cswitch_lock_var == 0);
    disable_interrupts();
    tcb_t * next = next_to_run();
    assert(next !=NULL);
    set_cr3((int)next->pcb_entry->page_table_directory);
    set_currently_running(next);
    set_esp0(next->esp0);
    asm_set_esp((int)(next->esp));
}
```

An interrupt here would be disasterous. Inevitable crash!

Tell virtual memory where to find this guy's stack

Note, we never actually return from this function! By changing the esp pointer

Not so bad...

Back to the future...

```
.globl asm_set_esp

asm_set_esp:
    MOVL    4(%esp),%eax    /*set esp to our new stack */
    MOVL    %eax, %esp      /*move in the new eax pointer*/
    STI                     /*enable interrupts*/
    POPL    %eax            /*get the restore flag*/
    CMPL    $0, %eax        /*if the flag isn't 0, jump to the IRET return*/
    JNE     .LABLE
    CALL    asm_restore

 .LABLE:
    CALL    asm_fork_restore /*second pathway out, with IRET*/

```

We disabled interrupts on the last slide

Jump ifnot equal

Figuring out the "fork" case made my brain hurt for days.

One easy case

```
/*This function restores the context of a
  thread paused by the task switcher */

.globl asm_restore

asm_restore:
    POPL    %eax
    POPA            /*Restore the general purpose registers from the kernel stack */
    RET             /*Return out of context_switch */
```

Pop my return address

Restore the registers

Again, not so bad

And then there's this....

```
/*This function handles a special case
  when we are context switching to a thread
  that has been created via fork.  This thread
  should instantly drop back into user mode
  as if the fork call had not happened
*/

#include <user_exec_info.h>

.globl asm_fork_restore

asm_fork_restore:

    #set segment selectors
    MOVL    $USER_DS_SEGSEL, %edi   /* segment setting needs to happen from a register */
    MOVW    %di, %ds                /*Set %ds*/
    MOVW    %di, %es                /*Set %es*/
    MOVW    %di, %fs                /*Set %fs*/
    MOVW    %di, %gs                /*Set %gs*/

    #set the esp value, popal and return
    POPL    %eax
    POPA                        /*We should POPA the general
    MOVL    $0 ,%eax            /*We are the child, our PID after fork is 0
    IRET                        /*IRET back to user mode, this process shouldn't be interrupted*/

```

Set up everything to run user code *black magic*

Our kernel stack is a duplicate of the parent thread

the child thread gets a 0 back from fork. it's like it never happened

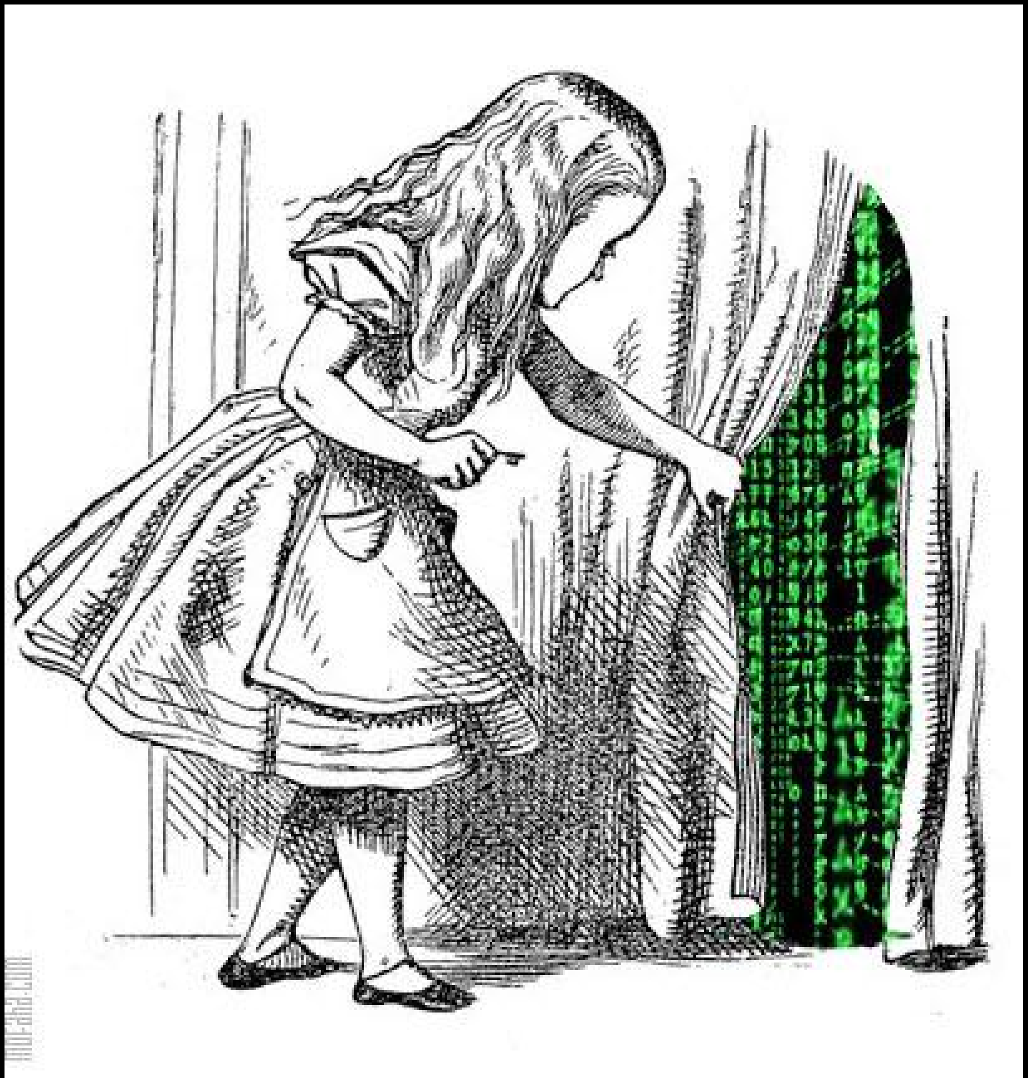
Oh the humanity, make it stop

Hey, that was easy!

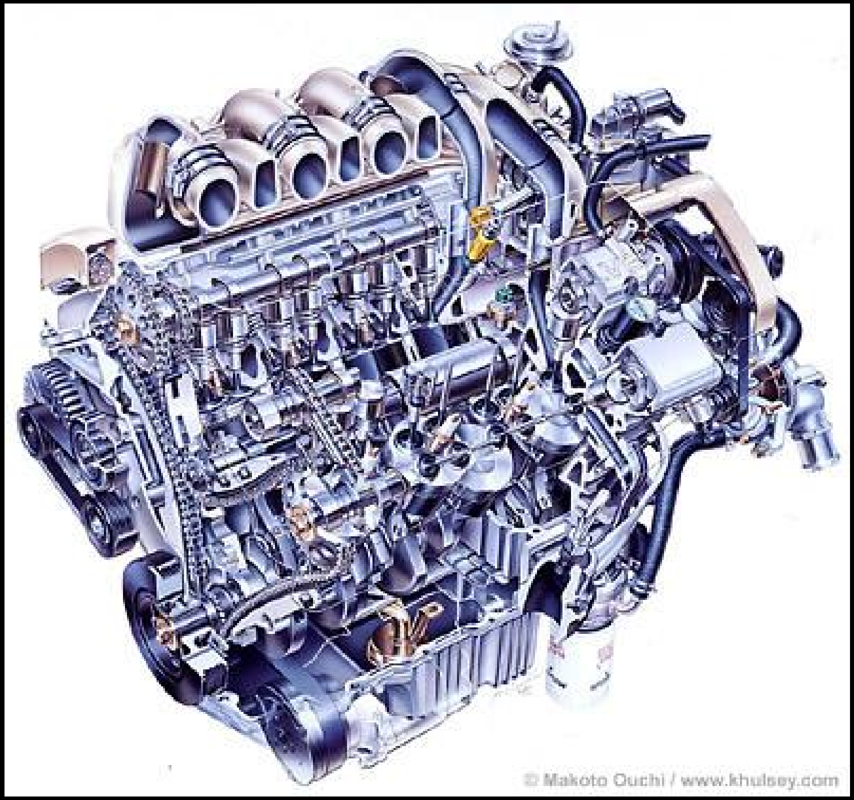
Except...

- All the datastructures tracking stacks, threads, processes, etc
- All the C code managing allocating and deallocating the memory pages
- Loading program out of memory and getting all the bits where they need to go
- Exec
- Drivers
- Bootstrapping the kernel itself
- the list goes on, some I wrote, some I didn't...

mmm... The rabbit hole goes deep.



Takeaways



1. There's a ton of complexity lurking under our happy illusions
2. Nothing in user code is safe, just more or less safe.
3. OS design can have a huge effect on performance
4. There are definitely bugs in the OS that will eventually cause a program to crash
5. Writing a kernel is awesome, every programmer should try it.

Parting Thoughts? Questions?