



CSE 451: Operating Systems, 26au

L4: Syscalls, Virtual Memory Intro

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

Sagar Saby

Andrew Wood

Adapted from slides by Tom Anderson

Aadminisstriviaaa

- Naomi's office hours!
 - Friday 2-3p (priority to CSE369 students)
 - Friday 3-4p (priority to ***you***, dear hacker)
 - By appointment (send me an email)



A slide about lab strategy

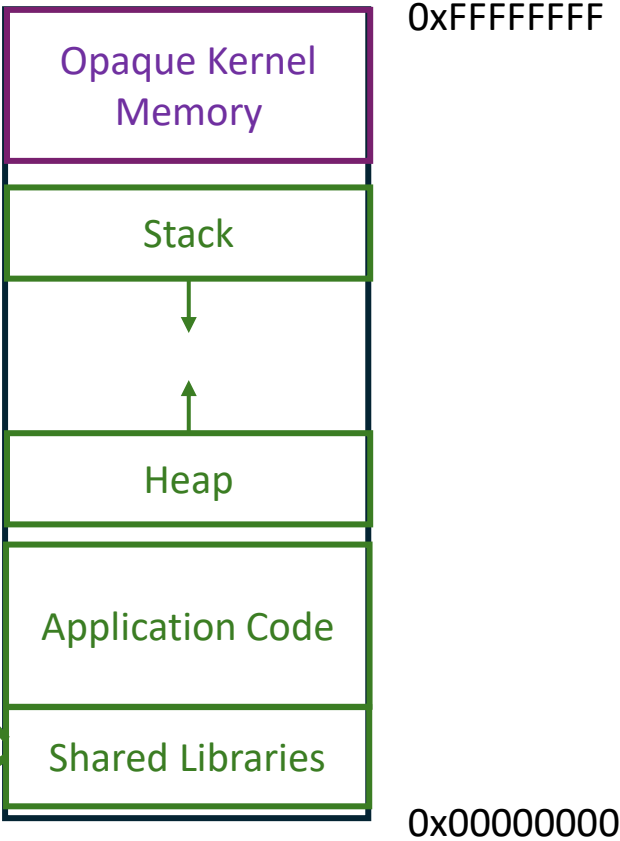
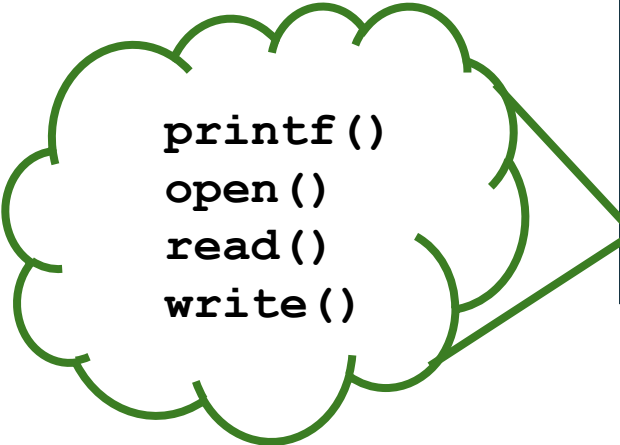
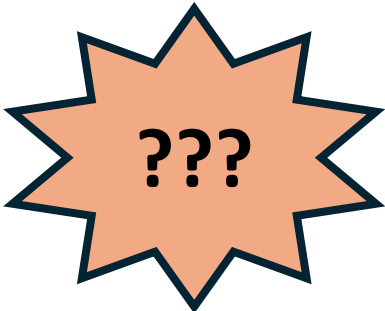
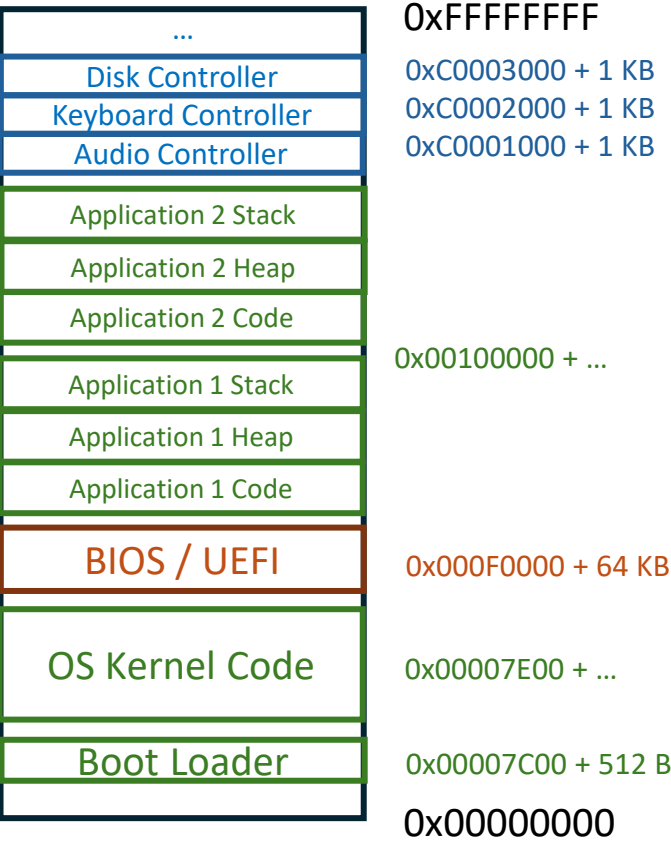


A second slide about lab strategy

1. Plant your “printf” flag
2. Read and understand all of the pre-existing data structures
3. Write an obviously wrong implementation (<3)
4. Think about your new data structures
5. Write half of a proper implementation
6. Think about your new data structures again, change them
7. Write the rest of your implementation

Physical Address Space

Process Memory (“user space”)



Syscall Implementation

(eg, lab 1)

System Calls

- Functions provided by the kernel to untrusted application code
 - Ex: file open, file read, file write, file close, pipe
- Limited set of entry points into the kernel
 - User code can't just call any kernel procedure

→ Protected procedure calls

→ API is system-specific (UNIX != Windows)

User Program

```
main() {  
    open("cats/danni.jpg", "r");  
}
```

User Stub

```
open:  
    push #SYSCALL_OPEN  
    trap  
    return
```

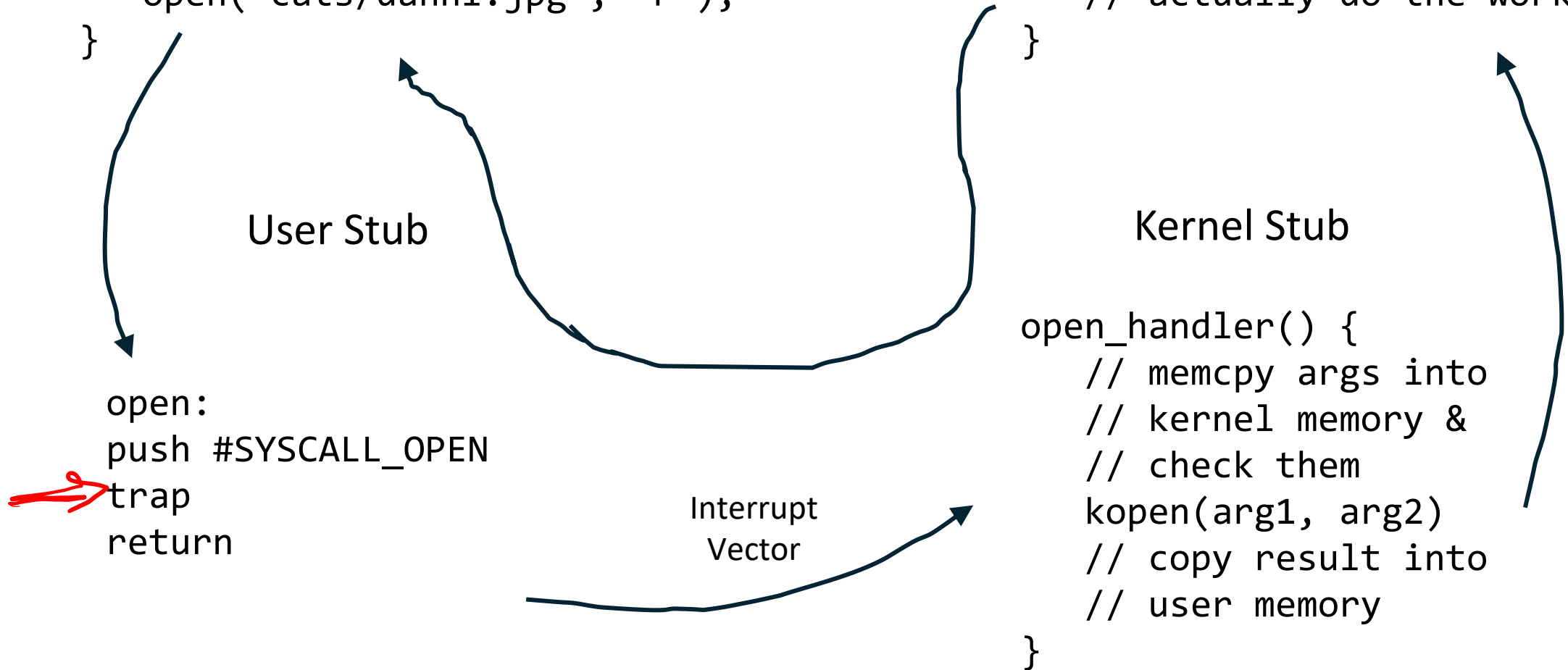
Kernel

```
kopen() {  
    // actually do the work  
}
```

Kernel Stub

```
open_handler() {  
    // memcpy args into  
    // kernel memory &  
    // check them  
    kopen(arg1, arg2)  
    // copy result into  
    // user memory  
}
```

Interrupt
Vector



Kernel System Call Handlers

- Locate arguments

- In registers or on user stack

 • *Translate* user addresses into kernel addresses

 • Copy arguments

- From user memory into kernel memory
 - Protect kernel from malicious code evading checks

- Validate arguments

- Protect kernel from errors in user code

- Copy results back into user memory

- *Translate* kernel addresses into user addresses

xk syscall deep dive

- We must get nittier and grittier

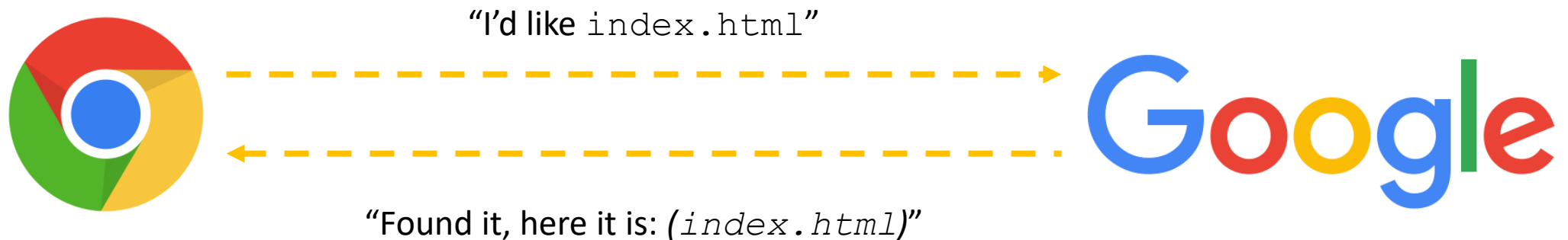


Questions

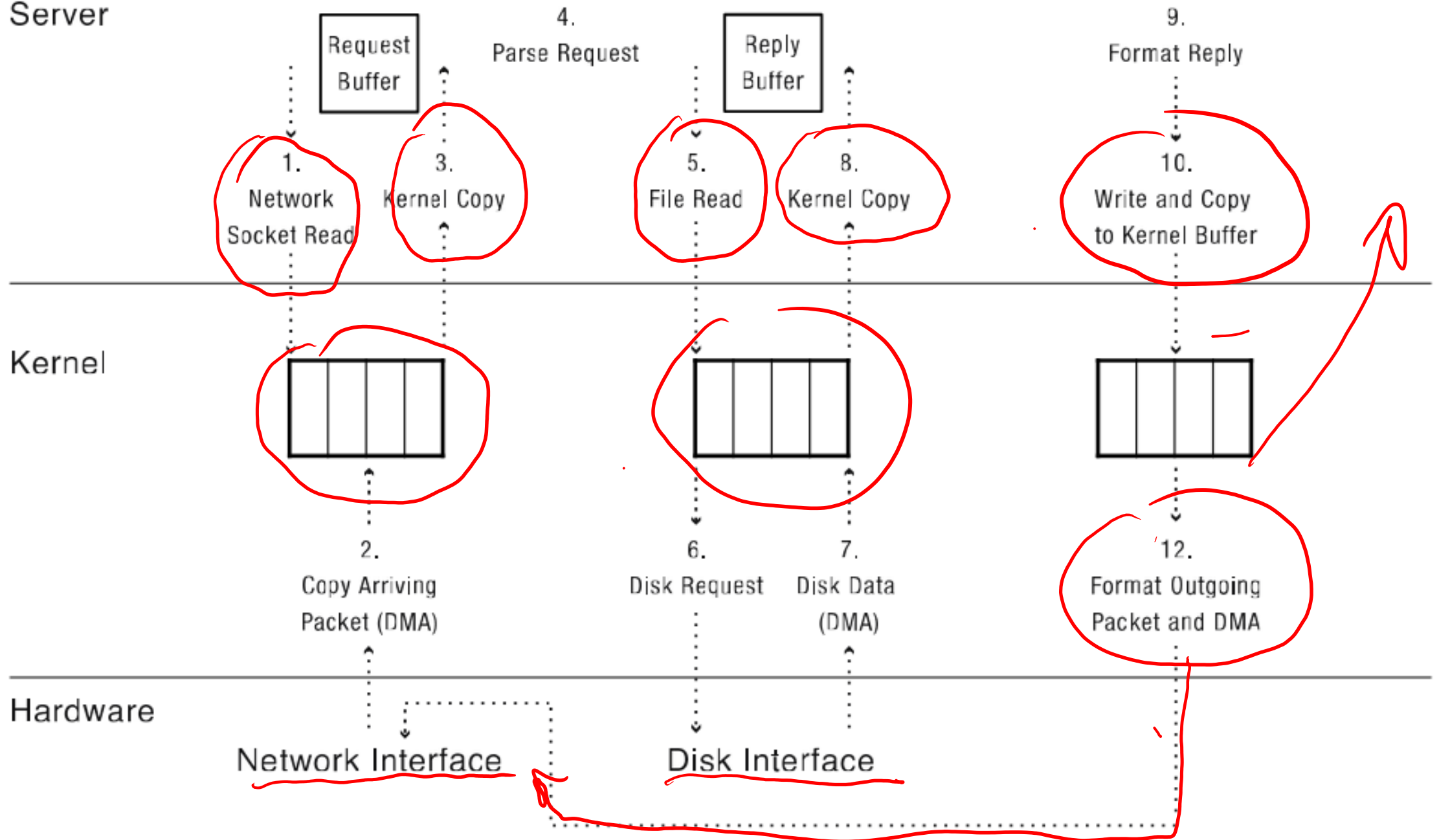
- Suppose the OS over-writes a value in the trapframe.
 - What happens when the handler returns?
 - Why might the OS want to do this?

Exercise

- How many user-kernel transitions are needed for a static web server to read an incoming HTTP request and reply with the file data?



Server



Upcalls

Upcall: User-level event delivery

- Notify user process of some event that needs to be handled right away
 - Time expiration
 - Real-time user interface
 - Time-slice for user-level thread manager
 - Interrupt delivery for VM player
 - Asynchronous I/O completion (async/await)

→ AKA UNIX signal

Upcalls vs Interrupts

- Signal handlers = interrupt vector
- Signal stack* = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

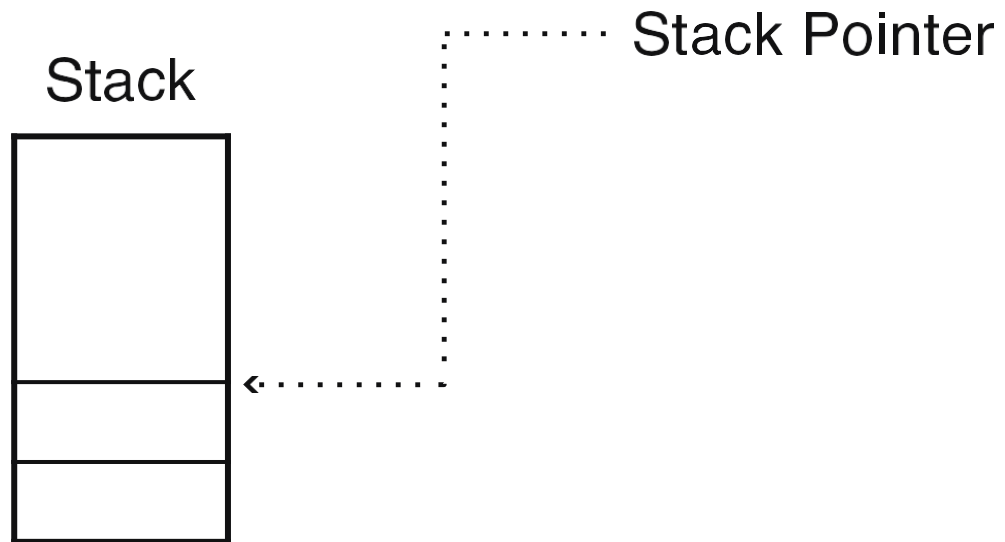
* still in user-space
memory

—

Upcall: Before

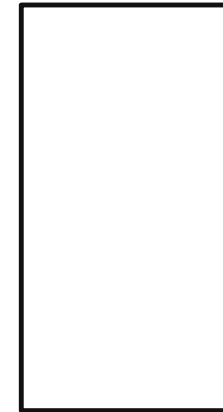
```
...  
x = y + z; <.....  
...
```

..... Program Counter

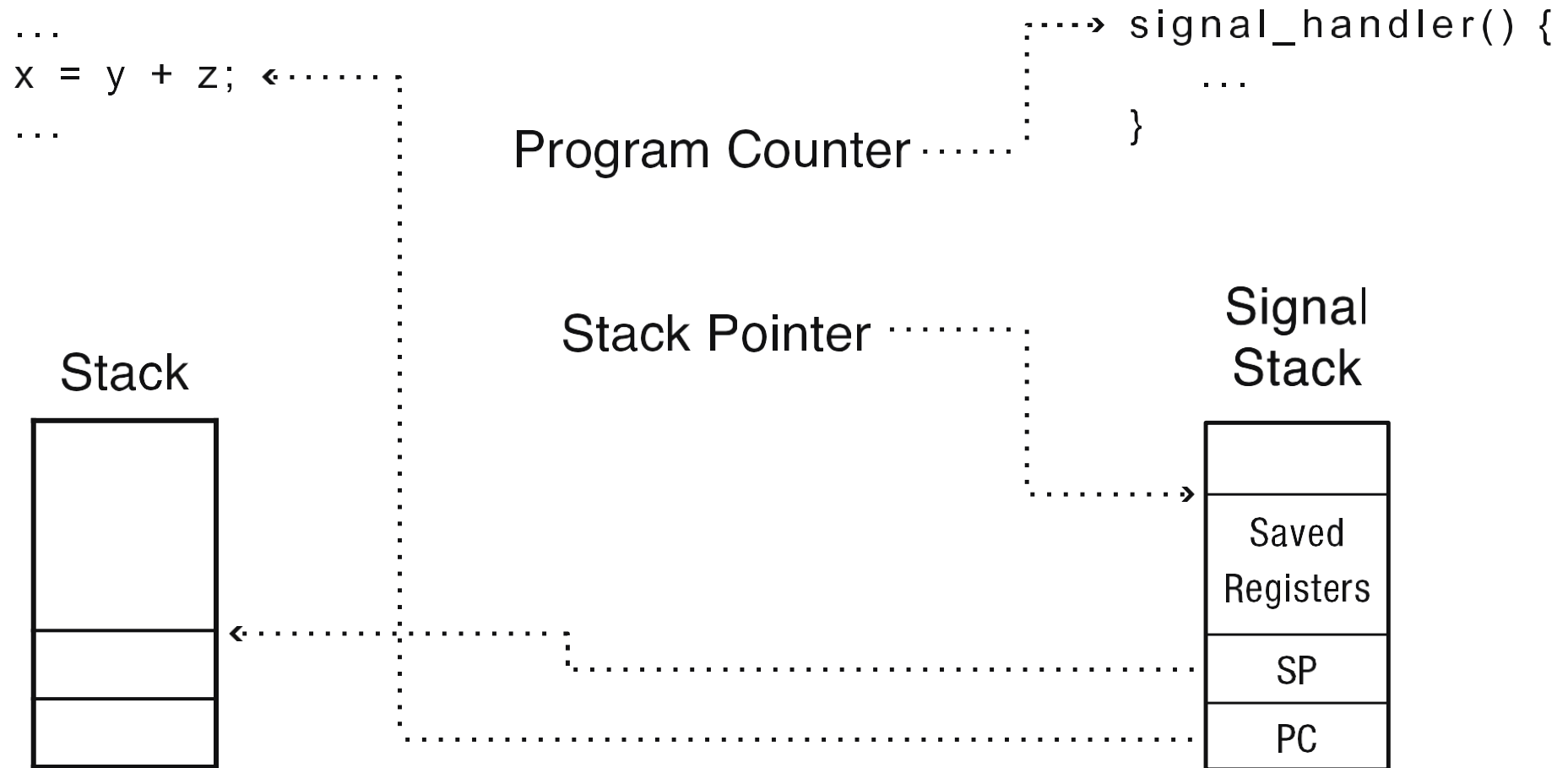


```
signal_handler() {  
    ...  
}
```

Signal
Stack



Upcall: During



But...h...how?

- In xk?
 - Not this quarter lol
- In Linux?
 - See the man page for `signal(7)`!
<https://man7.org/linux/man-pages/man7/signal.7.html>

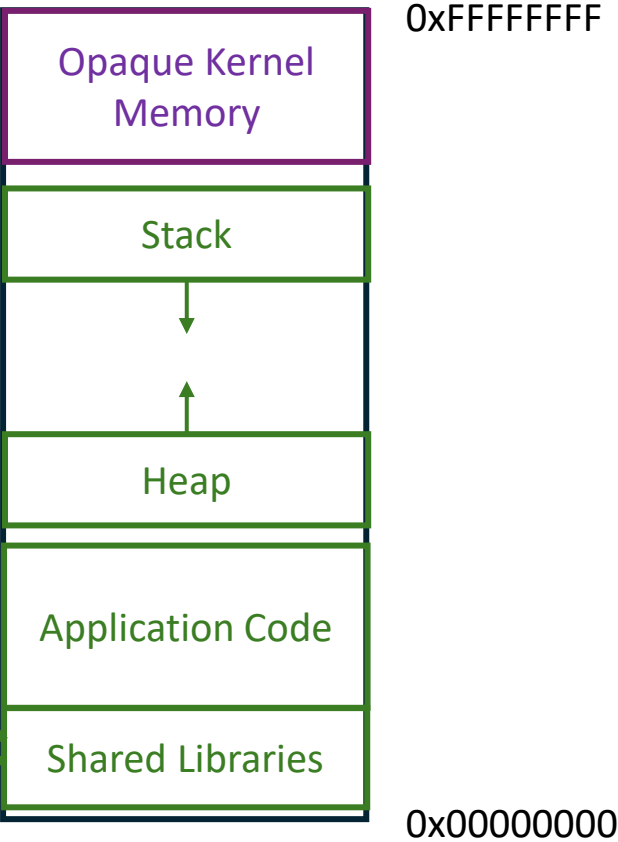
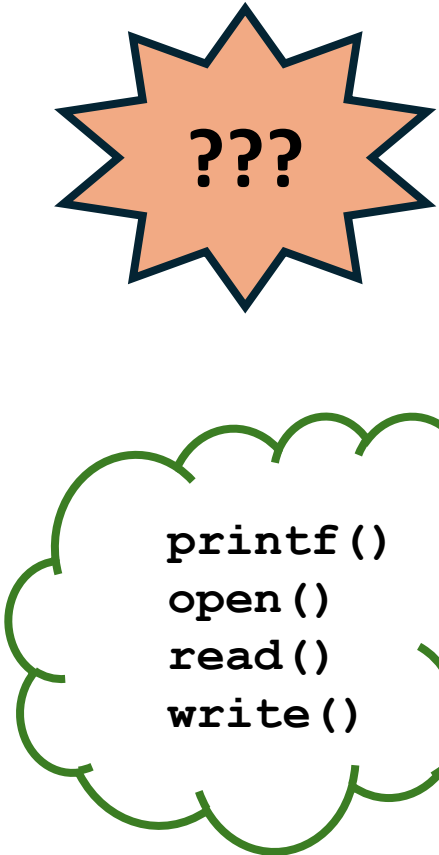
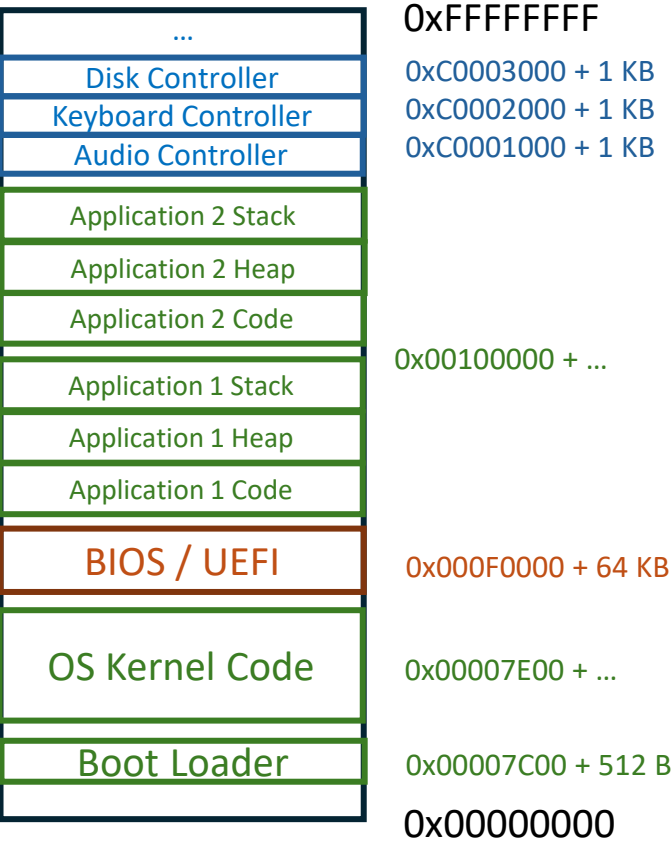
Class concluded here on 10/7
and will resume on 10/9



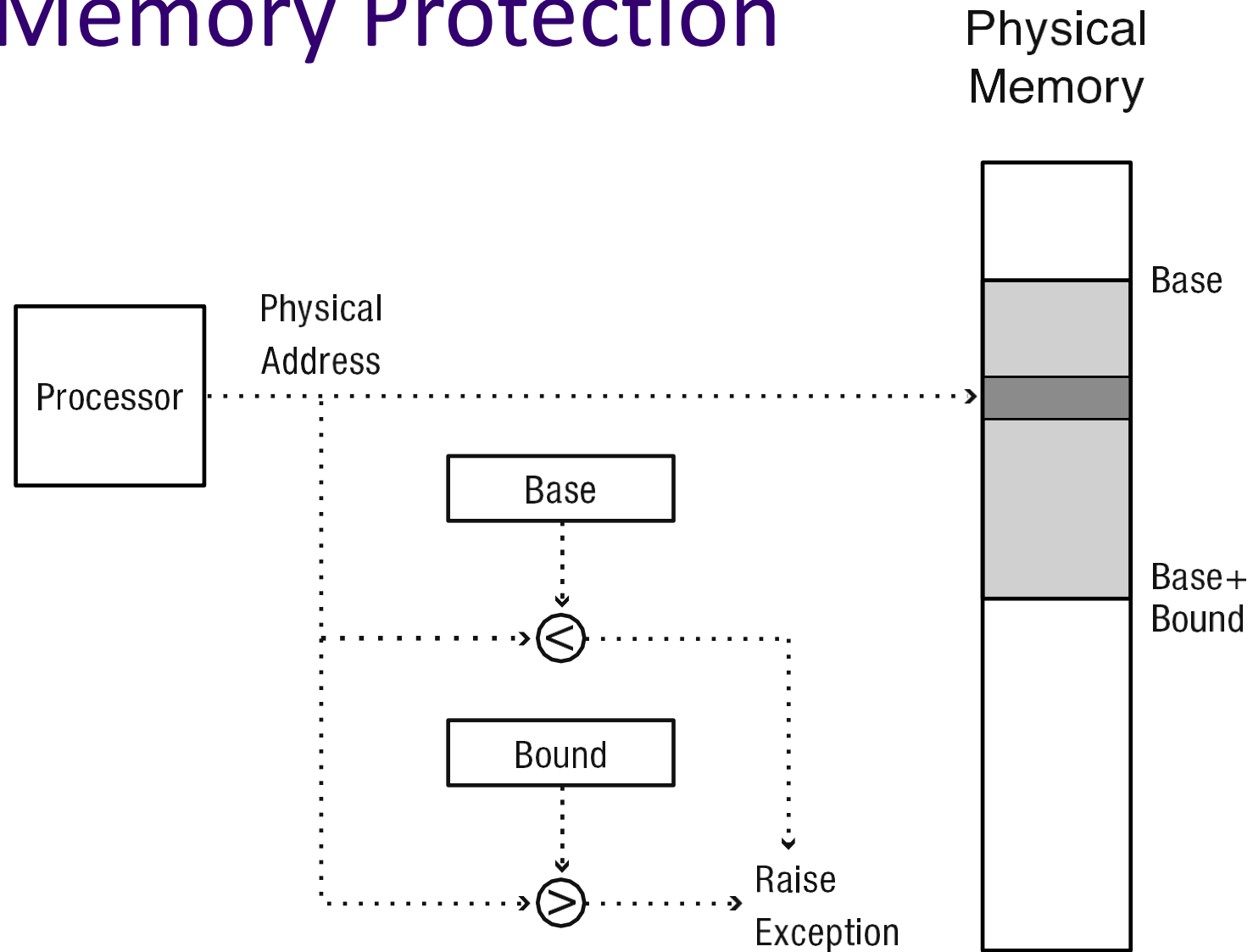
Sneak Preview: Virtual Memory

Physical Address Space

Process Memory (“user space”)



Simple Memory Protection

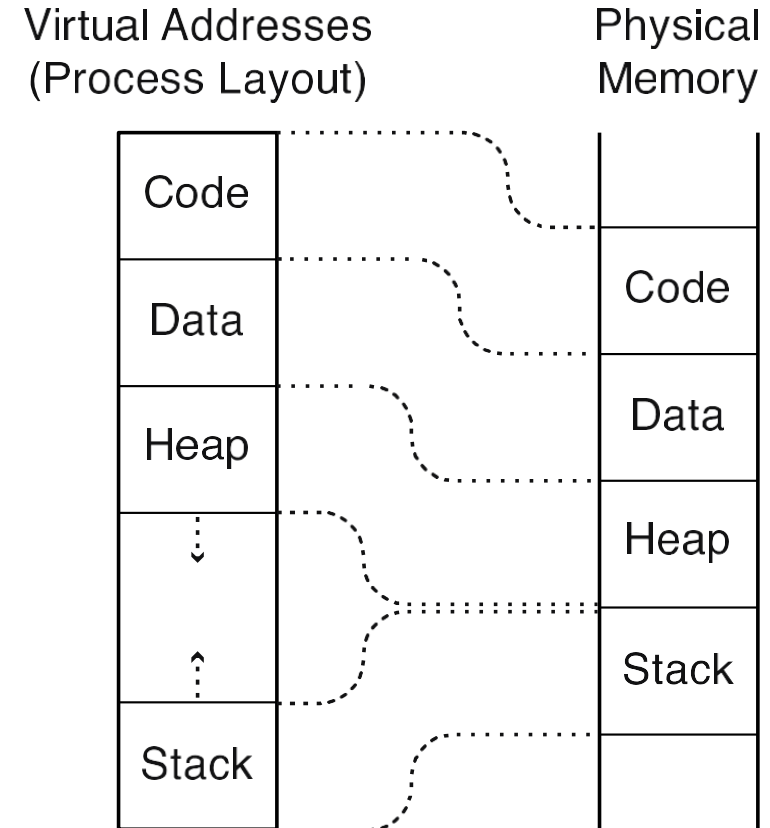


Benefits of Virtual Memory Translation

- Dynamically sized stack and heap
- Unmapped regions between valid ones to catch bugs
 - What if many threads? Need many stacks with many spaces between them
- Dynamically linked libraries
 - (Shared code regions between processes)
- “Zero copy” shared data between processes
- Shared data memory between chips/servers in the cloud
 - Remote DMA (RDMA)
- Memory mapped files

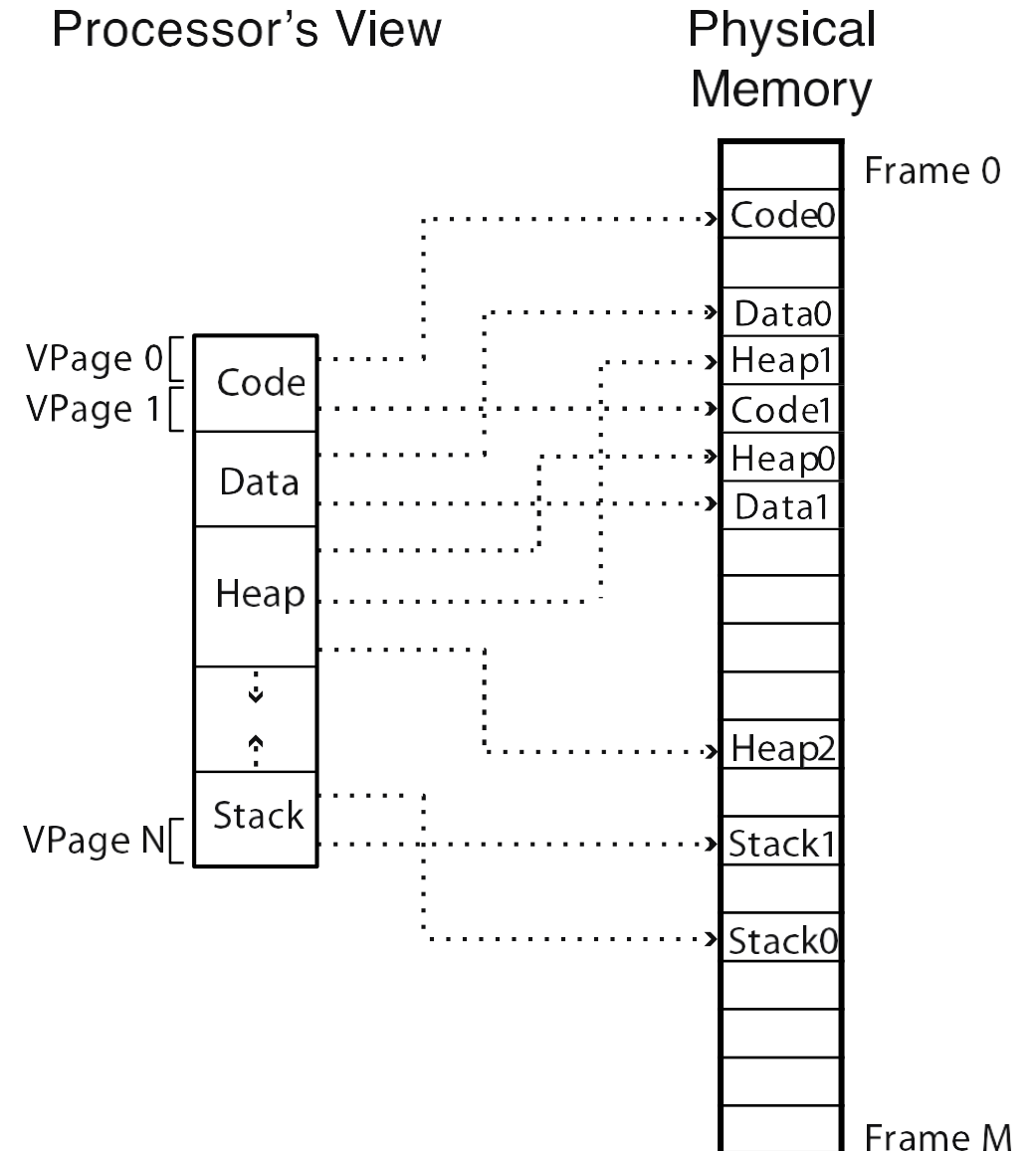
Virtual Addresses

- Translation done in hardware, using a table
 - On every instruction!
- Table set up by OS kernel
 - Eg, a page table points to fixed size page frames
 - Or a multi-level page table to handle very large or sparse virtual address space



Virtual Addresses

- Translation done in hardware, using a table
 - On every instruction!
- Table set up by OS kernel
 - Eg, a page table points to fixed size page frames
 - Or a multi-level page table to handle very large or sparse virtual address space



Experiment to see if our code lives in The Matrix

```
int staticVar = 0;          // a static variable
void main() {
    int localVar = 0; // a procedure local variable

    staticVar += 1;
    localVar += 1;
    sleep(10);           // sleep for x seconds
    printf ("static address: 0x%08x, value: %d\n",
            &staticVar, staticVar);
    printf ("local address: 0x%08x, value: %d\n",
            &localVar,  localVar);
}
```

Virtual Address != Physical Address

- The **same virtual address** in two different processes can refer to **different physical addresses...why?**
- The **same virtual address** in two different processes can refer to the **same physical address...why?**
- **Different virtual addresses** can refer to the **same physical address...why?**