The Kernel

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

What is the Kernel

- The OS is **software**, and it has many components:
  - User interface (graphical, terminal)
  - File system
  - Device drivers (code that knows how to “speak” to all kinds of external devices)
  - System utilities to manage the system (think the “control panel”)
  - Libraries (to make software development easier)
  - **The Kernel**
- There is some debate about what’s “in the OS” and what’s not
- But everybody agrees about the kernel
- The kernel is the core component of the OS in charge of resource virtualization and allocation
- It does all the special/dangerous things that we don’t want user programs to be able to do

Who Writes the Kernel?

- Kernel Developers :)
- Initially, kernels were written in assembly only (yikes!)
- Since 1960s: written in high-level languages (MS-DOS being an exception)
- Usually with a language in the C-language family
  - C-like languages are “close” to the hardware and make it easy for developers to play “tricks” to make the code space- and time-efficient
  - Compilers for these languages are really good at making fast executables for our CPUs
- Windows, Linux, iOS, MacOS kernels have been written mostly in C/C++
  - With parts still in assembly (e.g., for calling specific CPU instructions)
- In late 2022, Rust has become an official language for Linux Kernel development, in addition to C, and Rust kernel code is being developed (e.g., device drivers)

Kernel Development

- OS kernels are among the most impressive/challenging software development endeavors
  - Good news: a lot of very smart people have already written the critical parts of kernels
- As a kernel developer a constant concern is to not use too much memory so as to reduce memory footprint
  - Hence the need to write lean and mean code and data structures
  - Hence the struggle about whether to add new features
- Another constant concern is speed
- You cannot use standard libraries
  - Since you're writing the kernel, which sits below the libraries
- Nobody is watching over you, and bugs lead to crashes
- Let's look at some examples from the Linux kernel code...
  - You're not in ICS212 anymore!

Non-portable intrinsics

Faster conditional with a gcc directive

```
if (__builtin_expect(n == 0, 0)) {
  return NULL;
}
```

- In kernel code you often see things like the above
- The `__builtin_expect` keyword is a gcc directive where you get to indicate whether the condition is typically true or false
  - In the example above, the 0 second argument means “typically false”
- This is useful because then the compiler can generate faster code (by 1 or 2 cycles)
  - This has to do with pipelining and branch prediction (see a Computer Architecture course)

Bitwise operations and macros

Bitwise operations galore, often macroed

```
#define MODIFY_BITS(port, mask, dir)       \
        if (mask) {                        \
                val = sa1111_readl(port);  \
                val &= ~(mask);            \
                val |= (dir) & (mask);     \
                sa1111_writel(val, port);  \
        }

MODIFY_BITS(gpio + SA1111_GPIO_PADDR, bits & 15, dir);
MODIFY_BITS(gpio + SA1111_GPIO_PBDDR, (bits >> 8) & 255, dir >> 8);
MODIFY_BITS(gpio + SA1111_GPIO_PCDDR, (bits >> 16) & 255, dir >> 16);
```

- Bitwise operations are super fast/useful, and used a lot in Kernel code (due to having to encode information in as few bits as possible)

Macros, macros, ...

Bitwise operations galore, often macroed

```
#define DIV_ROUND_CLOSEST(x, divisor)({                    \
        typeof(x) __x = x;                                 \
        typeof(divisor) __d = divisor;                     \
        (((typeof(x))-1) > 0 ||                            \
         ((typeof(divisor))-1) > 0 || (__x) > 0) ?         \
                (((__x) + ((__d) / 2)) / (__d)) :          \
                (((__x) - ((__d) / 2)) / (__d)); })

#define container_of(ptr, type, member) ({                               \
        void *__mptr = (void *)(ptr);                                    \
        BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&    \
                         !__same_type(*(ptr), void),                     \
                         "pointer type mismatch in container_of()");     \
        ((type *)(__mptr - offsetof(type, member))); })
```

- Due in part to C's limitations, kernel developers typical define many macros

In-line Assembly

Code fragment with in-line assembly

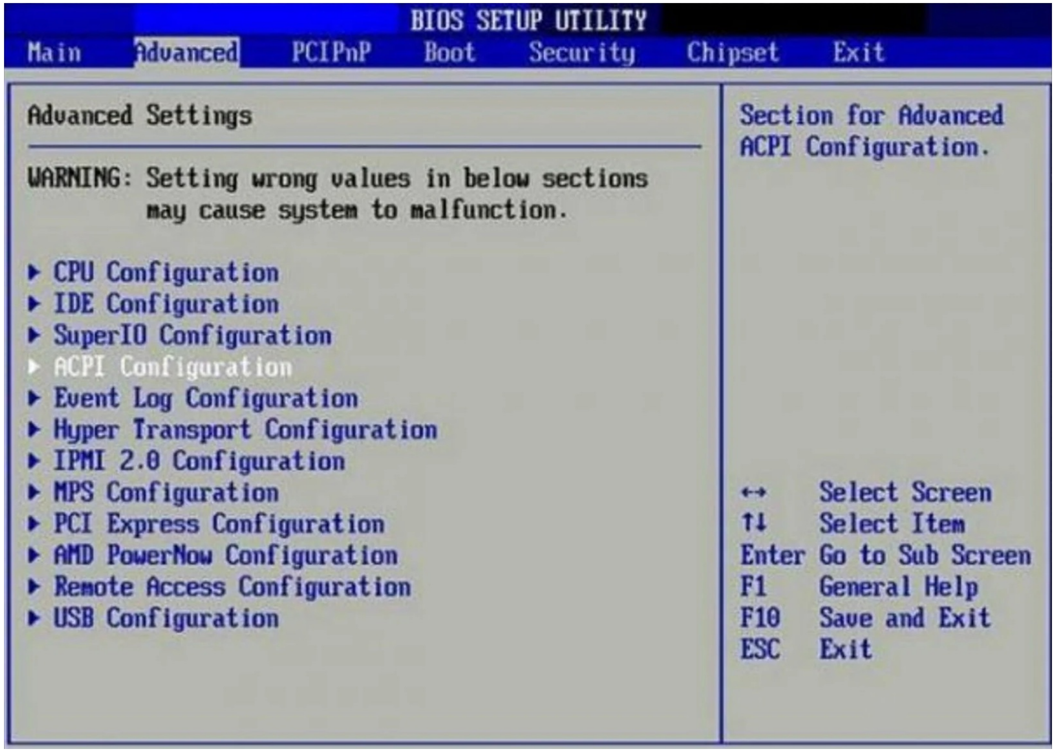
```
while (size >= 32) {
                asm("movq    (%0), %%r8\n" "movq   8(%0), %%r9\n"
                    "movq  16(%0), %%r10\n" "movq  24(%0), %%r11\n"
                    "movnti  %%r8,    (%1)\n" "movnti  %%r9,   8(%1)\n"
                    "movnti %%r10, 16(%1)\n" "movnti %%r11, 24(%1)\n"
                    :: "r" (source), "r" (dest)
                    : "memory", "r8", "r9", "r10", "r11");
                dest += 32;
                source += 32;
                size -= 32;
        }
```

- At many points in the kernel code there is inline assembly
- These are lines of assembly code that are spliced into the C code
- For speed or for doing things that would be difficult / impossible in C
  - The syntax above is x86 ATT syntax

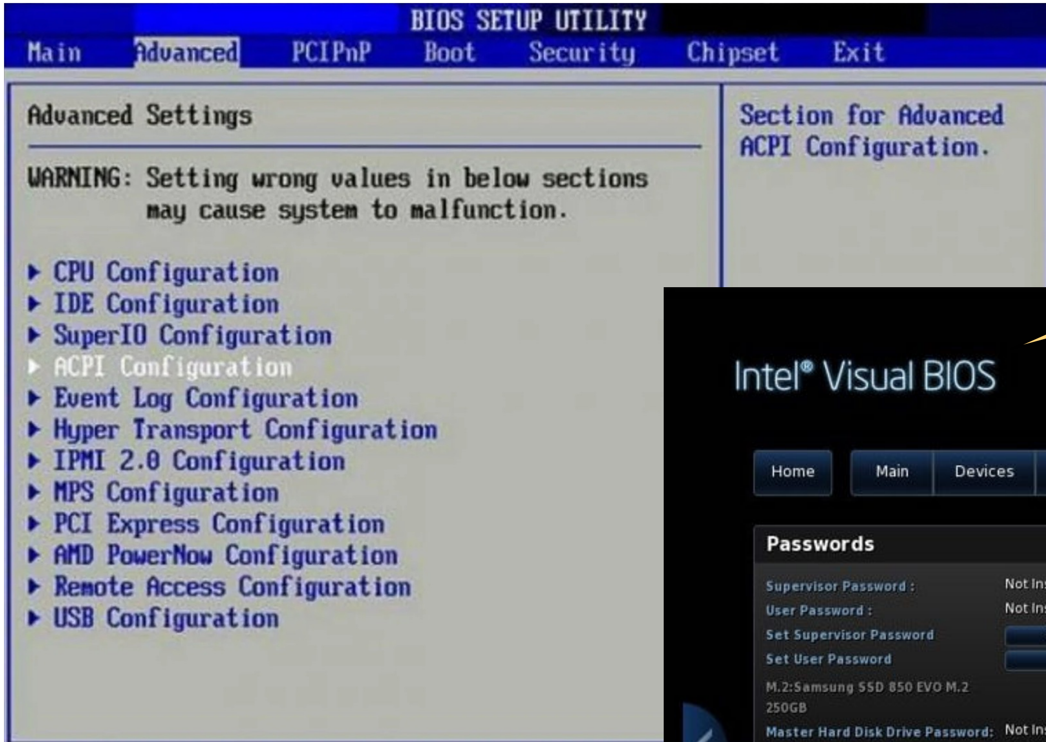
Who puts the Kernel in RAM?

- This happens during boot
  - Putting the kernel in RAM is the primary objective
- When you turn on your computer, POST (Power-On Self-Tests) are performed by the BIOS (Basic Input Output System)
  - Checks that RAM, disks, keyboard, etc. are all ok
  - Performs all kinds of initializations of registers and device controllers
- The BIOS is your computer's firmware: stored in non-volatile memory (doesn't need to be powered on to hold data)
- It used to be stored in a ROM chip (Read Only Memory), which means that a "firmware upgrade" would involved replacing the chip
- Nowadays it's stored in EEPROM / flash memory, which can be rewritten to do a firmware upgrade
  - Which opens security issues, and the possibility of a bug in the BIOS, which could turn your machine into the proverbial "brick"
- People still say "BIOS" but there have been some changes….

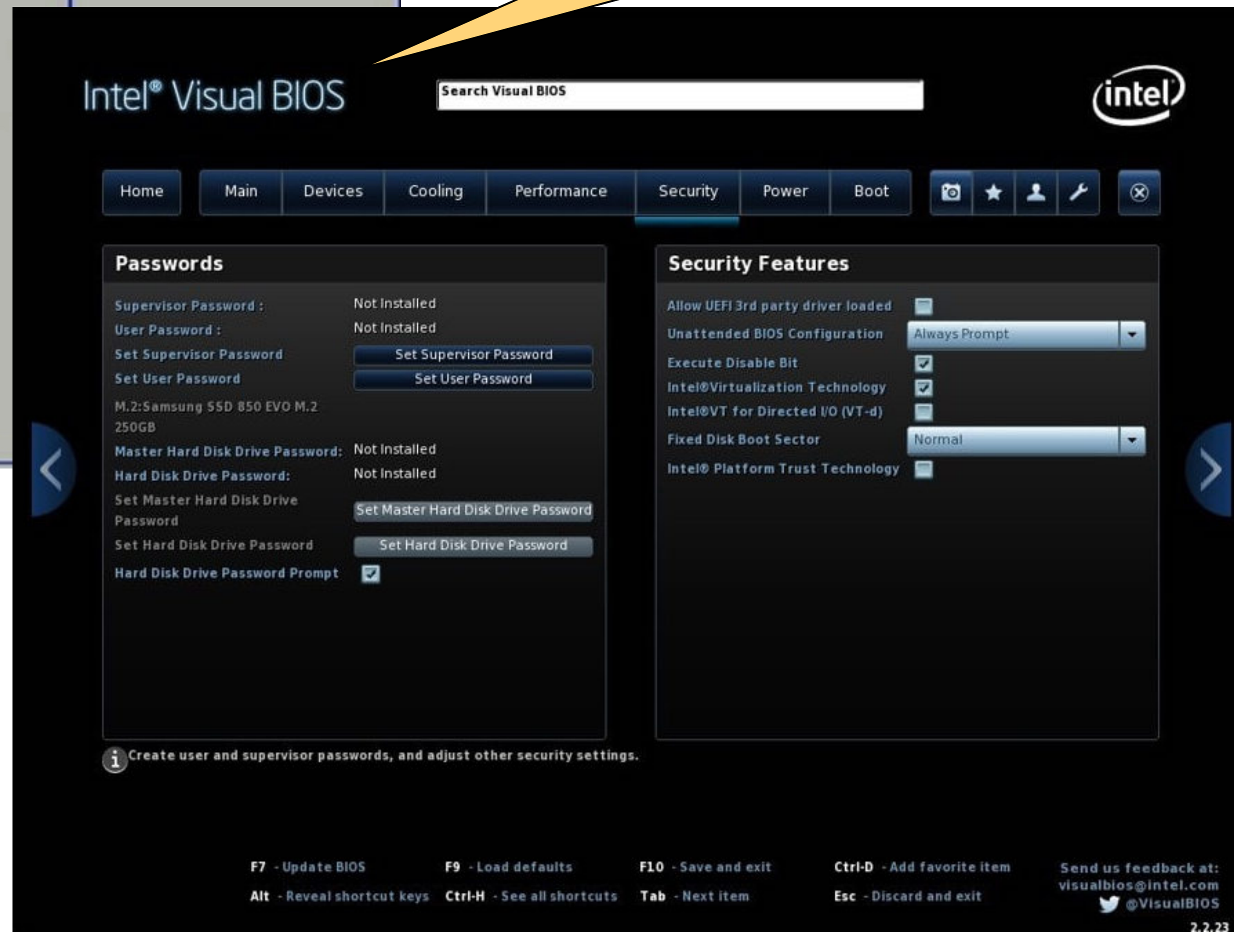
Basic Input Output System (BIOS)



Basic Input Output System (BIOS)



Still says BIOS



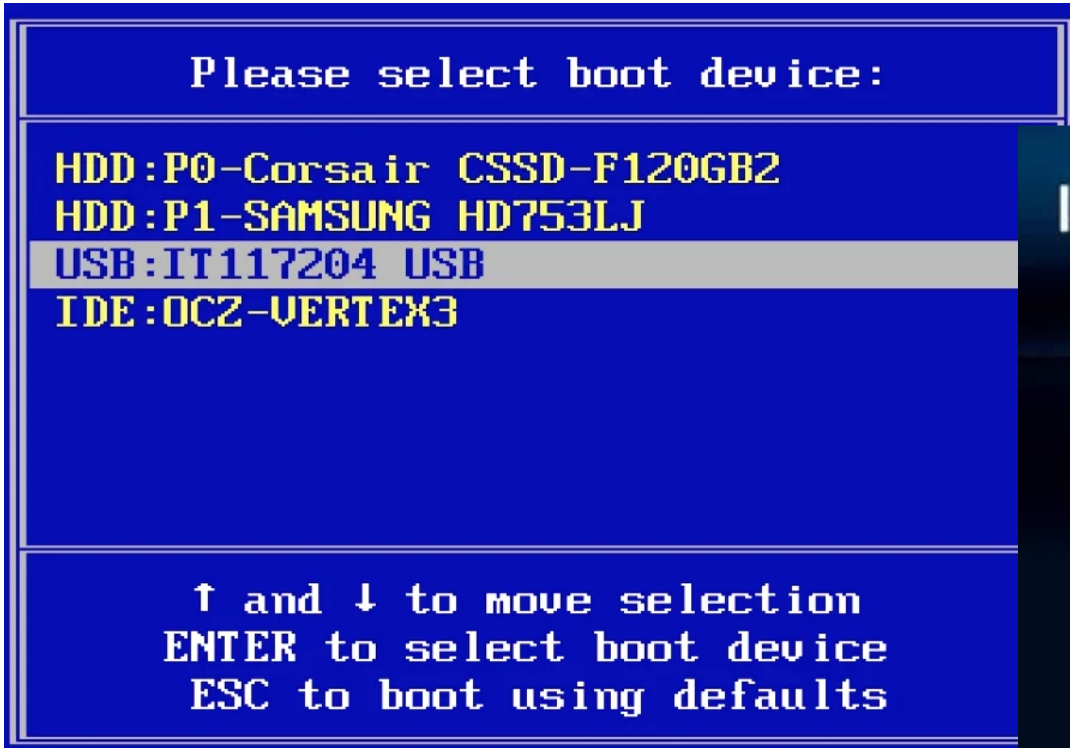
- Can do a lot more than the old BIOS
- Introduced in 2007
- Most computers today ship with UEFI instead of the old BIOS

Unified Extensible Firmware Interface (UEFI)

Finding a Bootable Device

- Configured in the BIOS is an ordered list of storage devices (disks, USB disks, CD-Rom, etc.)
  - This list is configurable in the BIOS
  - You may wonder how that works because the BIOS is stored in ROM!
    - Turns out, the list of bootable devices is stored in a small battery-powered CMOS memory, so that it keeps data even when the computer is powered off
    - And so the user can modify that list!
- The BIOS then goes through each device in order and determine whether it is bootable
  - It finds out whether the device contains a boot loader program
    - This is a program that knows how to load the kernel!
  - This is done in different ways (Master Boot Record, GUID Partition Table) and tons of technical details are available online
    - On my Mac: **`/System/Library/CoreServices/boot.efi`**

Selecting a bootable device



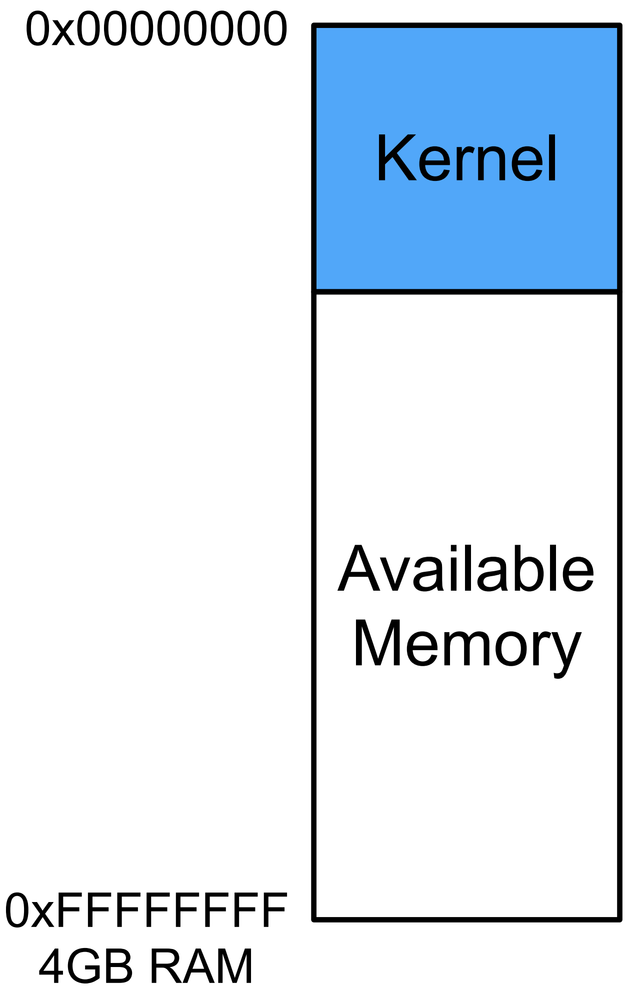
The Boot Loader Program

- The BIOS loads the boot loader program into RAM and hands over control to it (i.e., starts the fetch-execute-cycle from the boot loader program's first instruction)
- The boot loader program is the first program that runs on the machine
  - Linux: GRUB, LILO, etc.
  - Windows: WINLOAD. EXE
- The boot loader program does:
  - Perform some initializations to make sure the machine is ready for the kernel
  - Locates the kernel (code) on the bootable device and loads it into RAM and sets up data structures that the kernel will use
  - Then it hands off control to the bootstrap program…

The Bootstrap Program

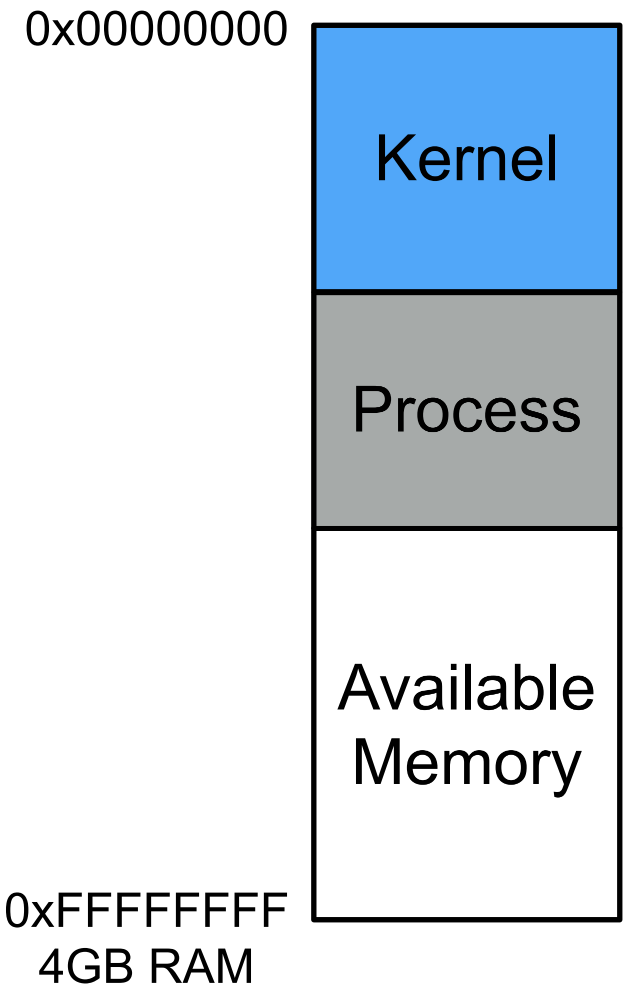
- The Bootstrap program is a program in Kernel code that
  - Does all “kernel initializations” (interrupt handles, timer, memory unit, etc.)
  - Configures and load all device drivers necessary for the detected attached devices
  - Starts system services (processes) that should be running
    - For instance, on Linux, the “init" process
  - Launches whatever application necessary for a user to start interacting with the OS
- Often this is done in a chain of loading/executing programs, each of them doing part of the work because loading/executing the next one

The Booted OS



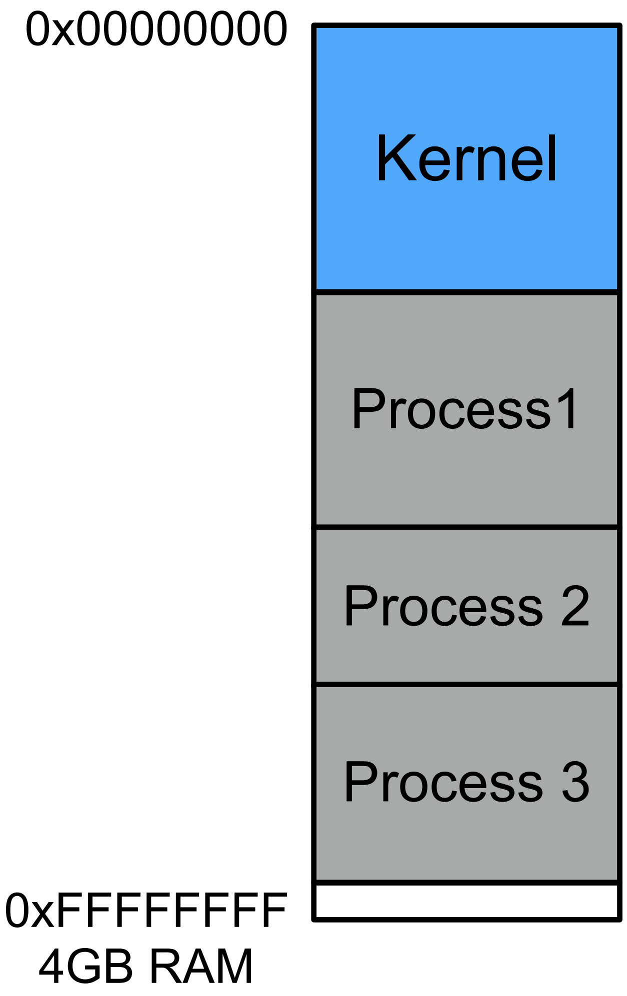
- The kernel code and data reside in memory at a specified address, as loaded by the bootstrap program(s)
- This picture is not to scale
- The kernel's memory footprint has to be small
  - This is memory the user cannot use

The Booted OS



- Each running program's code and data is then loaded into RAM
- A running program is called a process
- In RAM we thus have 2 kinds of code/data:
  - User code/data
  - Kernel code/data
- A process can run kernel code via system calls
  - Show of hands: who has heard that term before?

The Booted OS



- This figure shows 3 processes, occupying almost the full RAM
- Remember the OS illusion: each process thinks its alone, and processes never step on each other's toes in RAM (this is called memory protection)
- This figure makes drastic simplifications, and we'll see that the real picture is very different
  - But we can keep this simple picture in mind for a while
- If you want to know the list of processes running in your UNIX-ish machine: ps aux

The Kernel: An Event-Handler

- The Kernel is nothing but an event handler
  - After boot nothing happens until an event occurs!
- Once the system is booted, all entries into the kernel code occur as the result of an event
- The kernel defines a handler for each event type
- When an event occurs, the CPU stops what it was doing (i.e., going through the fetch-decode-execute cycle of some program), and instead starts running Kernel code
  - Just set the Instruction Counter register to the address of the first instruction in the appropriate event handler
- There are two kinds of events…

Interrupts and Traps

- **Interrupts**: Asynchronous events
  - Typically some device controller saying “something happened”
    - e.g., “incoming data on keyboard”
    - The kernel could then do: “great, I’ll write it somewhere in RAM and I’ll let some running program know about it”
  - ”Asynchronous” because generated in real time from the “outside world”
- **Traps**: Synchronous events (also called exceptions or faults)
  - Caused by an instruction executed by a running program
    - e.g.,“the running program tried to divide by 0”
    - The kernel could then do: “terminate the running program and print some error message to the terminal”
  - ”Synchronous" because generated as part of the fetch-decode-execute cycle from the “inside world”
- The two terms are often confused, even in textbooks…

The Kernel's (unrealistic) pseudo-code

Event handling code

```
class Kernel {
  method waitEvent() {
    while (doNotShutdown) {
      event = sleepTillEventHappens();
      processEvent(event);
    }
  }
  method processEvent(Event event) {
    switch (event.type) {

    case MOUSE_CLICK:
      Kernel.MouseManager.handleClick(event.mouse_position); break;

    case NETWORK_COMMUNICATION:
      Kernel.NetworkManager.handleConnection(event.network_interface); break;

    case DIVISION_BY_ZERO:
      Kernel.ProcessManager.terminateProgram("Can't divide by zero"); break;

  }
  return;
}
```

System Call: A Very Special Trap

- When a user program wants to do some "OS stuff", we've said it places a system call
  - e.g., to open a file, to allocate some memory, to get input from the keyboard, etc.
  - Essentially, to do anything that's not just "compute"
- A system call is really just a call to the kernel code
  - "Please kernel, run some of your code for me"
- We'll see how they work later
- But for now we can just think of it as just another case in our pseudo-code...

The Kernel’s (unrealistic) pseudo-code

Event handling code

```
class Kernel {
  method waitEvent() {
    while (doNotShutdown) {
      event = sleepTillEventHappens();
      processEvent(event);
    }
  }
  method processEvent(Event event) {
    switch (event.type) {

    case MOUSE_CLICK:
      Kernel.MouseManager.handleClick(event.mouse_position); break;

    case NETWORK_COMMUNICATION:
      Kernel.NetworkManager.handleConnection(event.network_interface); break;

    case DIVISION_BY_ZERO:
      Kernel.ProcessManager.terminateProgram(“Can't divide by zero"); break;

    case SYSTEM_CALL:
      Kernel.doSystemCall(event); break;
  }
  return;
}
```

Conclusion

- The kernel is code and data that always resides in RAM
- Booting is the process by which the machine goes from "turned on" to "the kernel has been loaded"
- The kernel is not a running program but really just an event handler
  - When some event occurs, some kernel code runs
- There are two kinds of events: asynchronous interrupts and synchronous traps
- An important kind of trap are system calls, by which user programs ask the kernel to do some work on their behalf

- Onward to Operating System interfaces…