

INF-2201

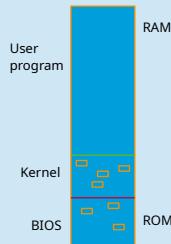
03 – Protection and system calls

John Markus Bjørndalen
2026

Problem: how do you call functions in the monitor or the kernel?

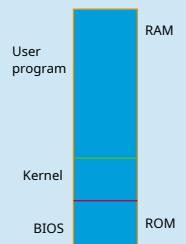
System calls, but how do we implement them?

- Pointers to functions?
 - How will programs know about them?
 - Solution1 : functions at predefined locations in kernel
 - Issue: new version of kernel – may need to recompile user programs with new function addresses?



Status now

- Very vague definition of a kernel
 - Just some useful common functionality that may update more frequently than the functionality we have in the Monitor ROM.
 - Provides a more abstract computer on top of the real computer
- First example of a real operating system (CP/M) that uses a BIOS instead of a Monitor
- Some of the boot process (no POST yet, ...)



NOP

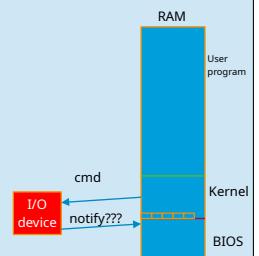
Status now

- There is now a separation between user level and kernel level
- The main function of the kernel (and BIOS) is to provide abstraction layers above the physical computer
 - Porting and software development easier and quicker
 - Programs may be moved between different computers without recompiling
- Kernel functions reached through a system call mechanism (jump tables)
- The separation is not enforced – mainly a convention

NOP

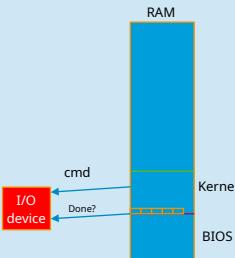
How do we handle I/O devices from software?

- Principle
 - Operating system or program uses I/O ports or memory mapped I/O to send commands and data to the I/O device
 - I/O device notifies software side that commands have completed or input is available
 - How?



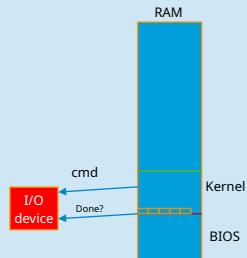
How do we handle I/O devices from software?

- I/O device notifying software 1: polling
 - Write commands
 - Periodically check if I/O device is done or has result
 - Read back result when I/O device states that it is ready
 - Same for incoming data: poll periodically to check if I/O device has received anything



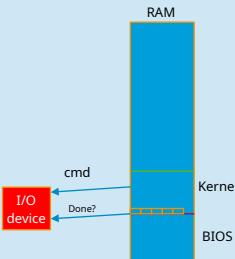
How do we handle I/O devices from software?

- I/O device notifying software 1: polling
 - Simple way of doing I/O, but there are a couple of harder points
 - How often do you poll?
 - How do you write programs that can alternate between processing and polling?
 - How many devices to poll?
 - Cmd
 - Poll
 - Poll
 - Poll – Are we there yet?
 - Poll – No. Go away



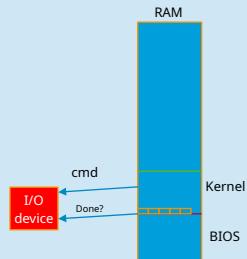
How do we handle I/O devices from software?

- I/O device notifying software 1: polling
 - Simple way of doing I/O, but there are a couple of harder points
 - How often do you poll?
 - How do you write programs that can alternate between processing and polling?
 - How many devices to poll?
 - Cmd
 - Poll
 - Poll
 - Poll – Are we there yet?
 - Poll – No. Go away



How do we handle I/O devices from software?

- I/O device notifying software 1: interrupt
 - Write commands
 - I/O device has a "bell" it can ring when it needs to notify software
 - Software is interrupted
 - pauses what it was doing
 - CPU handles source of interrupt in an interrupt handler
 - Interrupt handler or kernel restores state of running program and jumps back to program



Problem: How do we store and restore the state of a program during an interrupt?

- Rough overview
 - Need to know what defines the state of a running program
 - Need to pause and save this state somewhere
 - After processing interrupt, need to restore this state
 -

Observation

What does the program use and keep track of for a running program?

Basic information (for the current simple computer):

- Instruction pointer (IP/EIP)
- Stack pointer
- Registers

How they are used

- Variables are typically stored on heap (global variables or allocated variables) or stack/registers (local variables). NB: assuming a C-like language.
- Progress of the program is tracked using the IP
- Function calls:
 - Store state you need to keep (esp registers) on the stack
 - Store parameters to function on stack or in registers (here: register A)
 - Store return pointer on the stack (the next instruction / address after the "call")
 - Jump/Call to the function
 - Function looks at parameters and does its things
 - Return value from function may be stored in a register (can also use stack, but slightly more tricky)
 - When returning back to caller: pop the stored instruction pointer into current IP (rec, iret, ...)
- Control returns to the calling function

Super simple program in z80-ish asm

```

; Super simple program in z80-ish asm
0d00 push b      ; store registers
0d01 push c      ; store registers
0d02 mov a, 42   ; set parameter
0d04 push 0d0a   ; return addr/ip
0d05 call 0f00   ; call foo
0d06a push c     ; return here
0d06b pop b      ; return here
.....
; "function foo"
0f00 inc a      ; do something
0f01 mov a, 90   ; do something
0f02 ret         ; set return value
0f03 mov a, 90   ; return / pop ip
0f04 ret         ; return here

```

Tracing the instructions executed draws a "thread" of execution through the program

Observation

Basic information (for the current simple computer):

- Instruction pointer (IP/EIP)
- Stack pointer
- Registers

How they are used

- Sufficient to keep track of the current state of a running program in the current system we have!
- Store much of them when calling a function
- What if we can store everything we need before calling a "super function" that switches to an interrupt handler?
 - All registers
 - SP
 - IP

Super simple program in z80-ish asm

```

; Super simple program in z80-ish asm
0d00 push b      ; store registers
0d01 push c      ; store registers
0d02 mov a, 42   ; set parameter
0d04 push 0d0a   ; return addr/ip
0d05 call 0f00   ; call foo
0d06a push c     ; return here
0d06b pop b      ; return here
.....
; "function foo"
0f00 inc a      ; do something
0f01 mov a, 90   ; do something
0f02 ret         ; set return value
0f03 mov a, 90   ; return / pop ip
0f04 ret         ; return here

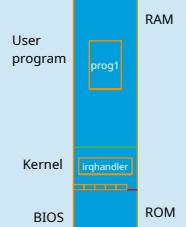
```

Tracing the instructions executed draws a "thread" of execution through the program

Switching to interrupt handler

Switch to interrupt handler:

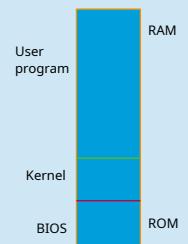
- CPU receives interrupt
 - Looks at interrupt ID/number to determine interrupt handler to run (similar to looking up system calls earlier)
 - In our OS, the handler will be in the kernel
 - Jumps to interrupt handler
- Interrupt handler
 - store/snapshot current state (all registers)
 - Talk to hardware to receive data, update state information (cmd done) etc
 - Restore state of interrupted program (restore registers)
 - Return to program using iret



NOP

Problem: protect OS against bugs

- Problem: bugs in user programs may overwrite parts of the operating system in memory
 - User programs can crash more than themselves
 - Computer could crash in strange ways or worse: proceed *almost* correctly (damaging files etc)
 - Software that intentionally tries to harm a system

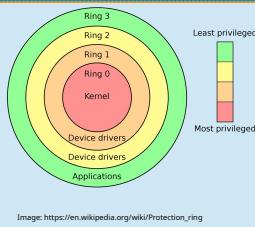


Solution: lock down the kernel

Lock down computer so user code only has access to itself and resources that it is permitted to use.

How?

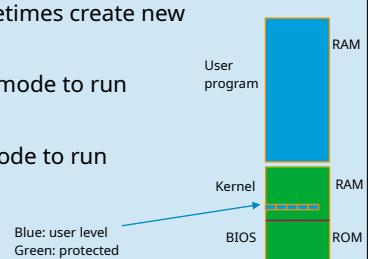
- Assume memory space can be partially locked down to protect kernel
 - For now, ignore *how* we do this (we will look at it later in the course)
- Need to lock down access to I/O, and to change things the kernel needs. One mechanism: protection rings.
 - Ring 0: full access to memory, I/O and privileged instructions
 - Ring 1-2: not used much now, but intended for device drivers etc
 - Ring 3: user mode for user programs



Ooops

Notice a pattern: solutions sometimes create new problems.

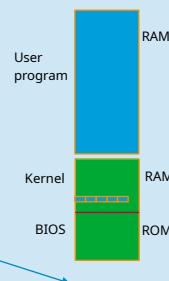
- Need to be in ring 0 / kernel mode to run privileged instructions
- Need to be in ring 3 / user mode to run unprivileged (user code)
- How do we switch?



Handling protection modes

How do we switch between user and kernel mode?

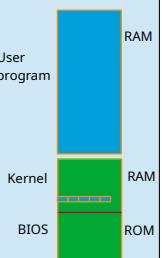
- Computer boots in ring 0 / privileged
- Instructions switch to user mode.
Roughly (on intel):
 - First, need to set up a GDT (Global Descriptor Table) to deal with memory protection. Again: we're handwaving memory protection
 - Interrupt table (we'll explain in a bit)
 - Set up user level stack (and some other things)
 - "Return" to user level using iret or sysexit



Handling protection modes

Great, the kernel set everything up and switched to user level. The code is running and needs I/O. **How do we get back in?**

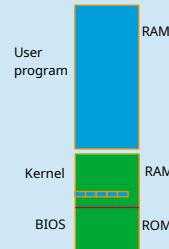
- Ring 0 is privileged, so user level cannot snoop into kernel or just jump back in.
- Need a mechanism to enter the kernel again, but controlled by the kernel
- System call:
 - User level sets up what it wants to do in the kernel (function ID, data/parameters etc)
 - Traps to the kernel --- somehow
 - CPU switches to kernel mode and runs as a kernel mode at a predefined entry point
 - Kernel
 - saves necessary state from the user process
 - Inspects the description of what needs to be done and decides if the user is allowed
 - Runs the necessary bits and sets up return values
 - Returns back to user mode (similar to previously described)



Handling protection modes

There are now mechanisms in place for

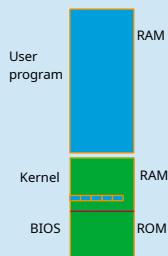
- Separating kernel from user level
- Protecting the kernel from user level code
- Switching to user level code
- Making system calls
 - Kernel can inspect the described call before taking action and decide what to do
- Resources (mostly for P3):
 - https://wiki.osdev.org/System_Calls
 - https://wiki.osdev.org/Getting_to_Ring_3
 - <https://wiki.osdev.org/GDT>



NOP

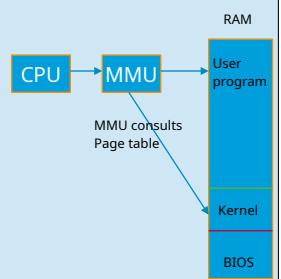
Protection part 2 : memory protection

- We have protected against execution of privileged instructions.
- How do we protect memory?
 - Need to flag which memory addresses belong to the kernel (and should be protected)
 - Which memory addresses belong to user program(s)
- Need something that enforces the memory protection policy that the operating system sets up
 - Hard to do only in software



Protection part 2 : memory protection

- Enter MMU (Memory Management Unit)
 - Inspects addresses of memory requests from CPU to memory
 - Compares address with a "list"/data structure of address ranges and permissions
 - Organized to address "pages" of memory (for your OS: 4KiB per page)
 - Can also be used to translate / re-map addresses. We will get back to that later in the course (Virtual Memory). For now, permission is the key idea
 - Can be integrated in CPU or separate chip



NOP