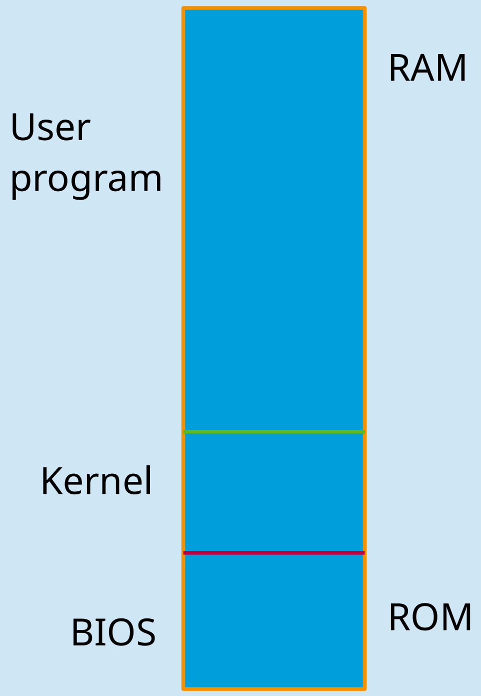
# INF-2201
# 02 and 04 – System calls, processes and protection

**John Markus Bjørndalen**

**2025**

## Status now

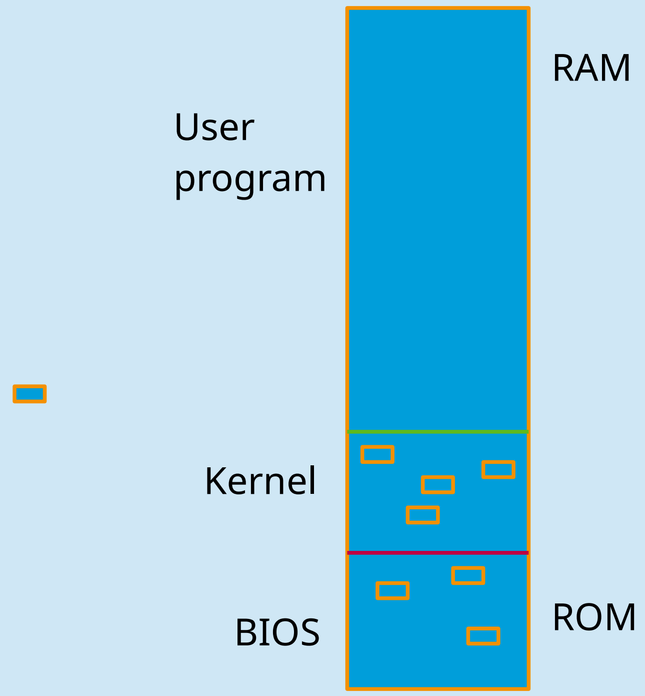
- Very vague definition of a kernel
  - Just some useful common functionality that may update more frequently than the functionality we have in the Monitor ROM.
  - Provides a more abstract computer on top of the real computer
- First example of a real operating system (CP/M) that uses a BIOS instead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



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

System calls, but how do we implement them?

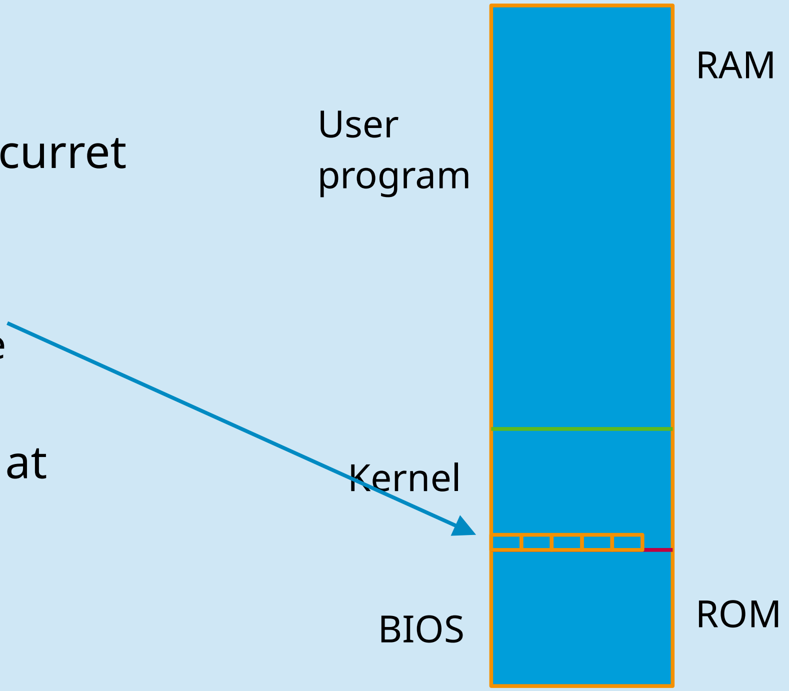
- Pointers to functions?
  - How will programs know about them?
  - Solution1 : functions at predefined locations in kernel
  - Issue: new version of kernel – may need to recompile user programs with new function addresses?



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

System calls, but how do we implement them?

- Jump table?
    - Table (at predefined location) with pointers to curret location of functions
        - Jump to (or "call") entry point in table.
        - The entry in the table is a jump instruction + the target address
    - Advantage: these jump tables can be updated at run time
- Similar mechanisms used in early operating systems
- Sufficient until we start protecting the kernel.
-



## 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

## Problem: computer can only run one program simultaneously

- Computers used to be very expensive
- First thought: Sometimes the CPU is just waiting for I/O  (user input, disks, paper tapes, somebody to load the tapes in the reader, …)
  - What if you could have some other program running simultaneously that could do things in the background?
- Second thought: if you can run multiple programs at the same time, then multiple users can share the same computer

# Observation

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

Basic information (for the current simple computer):

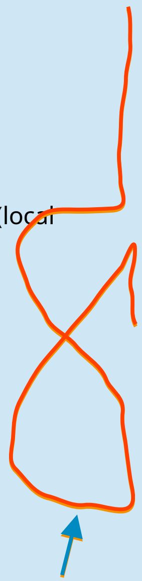
- Instruction pointer (IP/EIP)
- Stack pointer
- Registers

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 (ex: 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 (ret, iret, … )
  - Control returns to the calling function

```
; 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
0d07  call 0f00      ; call foo
0d0a  pop c          ; return here
0d0b  pop b
…...

; "function foo"
0f00 inc a           ; do something
…                    ; do something
0f40  mov a, 90      ; set return value
0f42  ret            ; return / pop ip
```



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

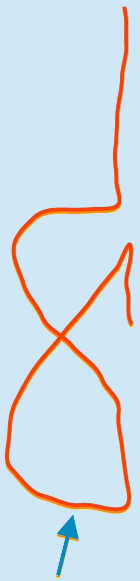
# Observation

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 swiches to a different program?
  - All registers
  - SP
  - IP



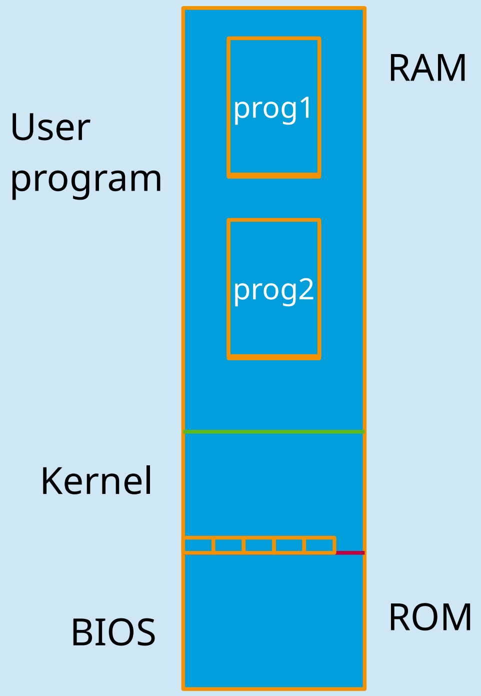
```
; Super simple program in z80-ish asm
…
0d00  push b        ; store registers
0d01  push c        ; store registers
0d02  mov a, 42     ; set parameter
0d04  push 0d0a     ; return addr/ip
0d07  call 0f00     ; call foo
0d0a  pop c         ; return here
0d0b  pop b
…. . .

; "function foo"
0f00 inc a          ; do something
…                   ; do something
0f40  mov a, 90     ; set return value
0f42  ret           ; return / pop ip
```

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

# Running multiple programs

- Switch to another program (**yield**):
  - store/snapshot current state (all registers, including IP to "return here" and SP)
  - call function that selects which program gets to run next (let's call that the scheduler)
  - Restore registers (start with stack pointer)
  - "Return" to selected/scheduled program using "ret"
- NB: order of store and restore is important. One method:
  - Store IP of where to return to
  - Store registers
  - Store SP "somewhere"
- When restoring state
  - Restore SP (to pick the right stack)
  - Restore registers
  - "ret" to return to the indicated return position (instruction after the call to yield)



# Processes

We currently have

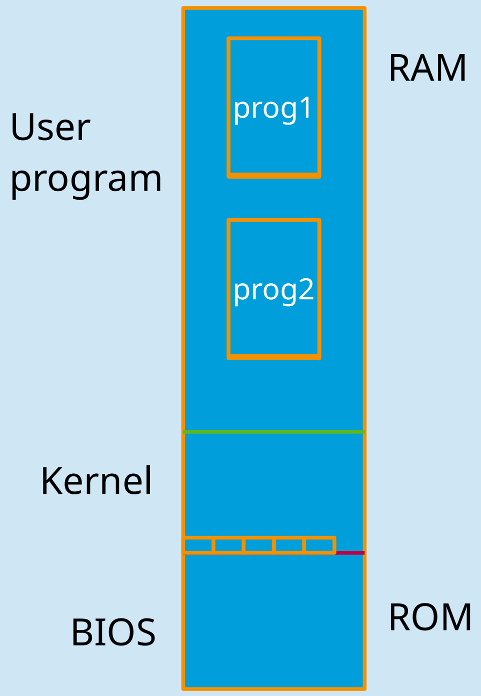
- Method for yielding execution to let somebody else take over: yield
- Method for selecting the next to run: scheduler

This is the first step to what we call cooperative multitasking

- Multiple programs that run concurrently, yielding execution time to each other

But how do we pick the next to run (scheduler), and where do we store the state of the programs?

- Introduce processes:
  - A process is a running program
    - The program binary
    - Data used by the program
    - Execution state (registers and stack)
    - Any other state that the operating system needs to keep track of the running process



# Keeping track of processes

- Need information
  - Current execution state
    - When it is running: in registers
    - When it is paused: kernels typically use a PCB (process control block) to save state necessary to restore the execution
  - Other state necessary for the kernel and processes
    - Identifier (ex: process id)
    - Permissions/right (we will look into this later)
- What is stored in a PCB?  (Assume registers are stored on stack when yielding)
  - Process ID
  - Maybe memory location where the process is stored
  - Stack pointer (to let us restore/find stack and restore registers from there)
  - Some implementations might store a small kernel stack in the PCB to use for kernel functionality

## Kernel needs to keep track of multiple processes

- If each process has a PCB that uniquely describes that process and state, use that PCB to also keep track of which processes that are currently waiting to run and which process is currently running
  - current_process → PCB of currently running process
  - ready_queue → queue of PCBs / processes waiting to run
- Process switching: when the scheduler is called:
  - Move the current_process PCB to the end of the ready queue
  - Move the first PCB from ready_queue to current_process
  - Restore process state of current_process
- This is called round robin scheduling

## How do we start a new process?

- We currently have mechanisms for yielding (storing state), selecting new state (schedule) and restore state.
- Instead of creating yet another mechanism, simply use the kernel's scheduler:
  - Create a restore state pointing to where you want the process to start  ("fake" a process that was interrupted before running its first instruction)
  - Fill in the PCB
  - Submit the process to the kernel scheduler
  - Let the scheduler run, pick and activate a process

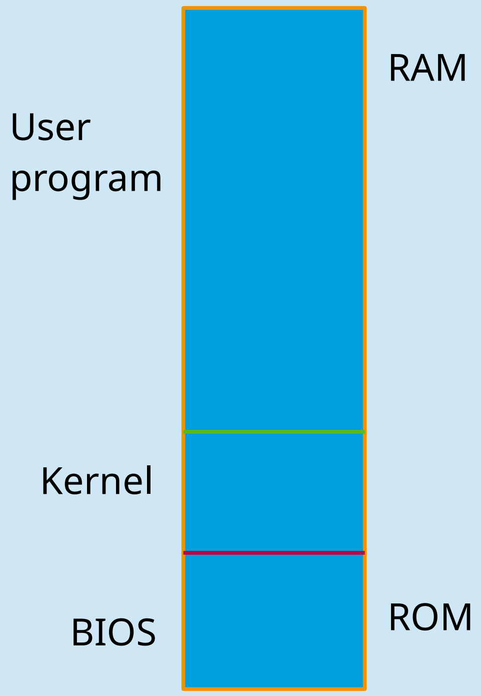
## Note

- The ideas presented so far will need to be extended and modified as the OS gets more advanced
- The core concepts are still similar

## 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
  - Some of this protection can use the priviledge rings shown during the P1 presentation
  - Ring 0: full access to memory, I/O and priviledged instructions
  - Ring 1-2: not used much now, but intended for device drivers etc
  - Ring 3: user mode for user programs

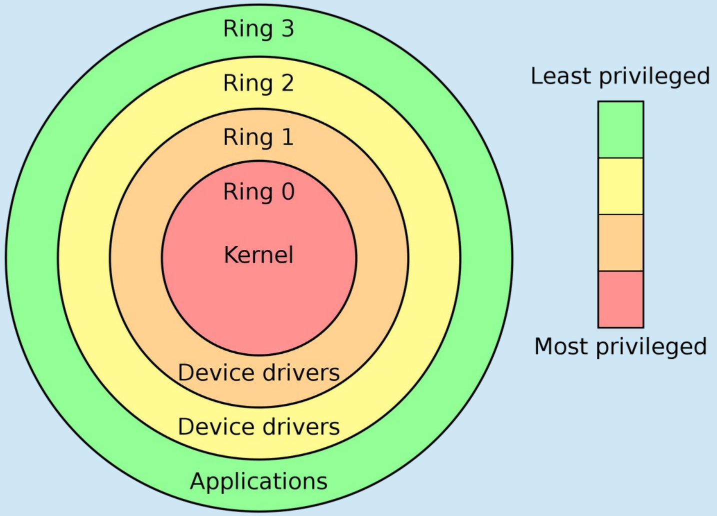
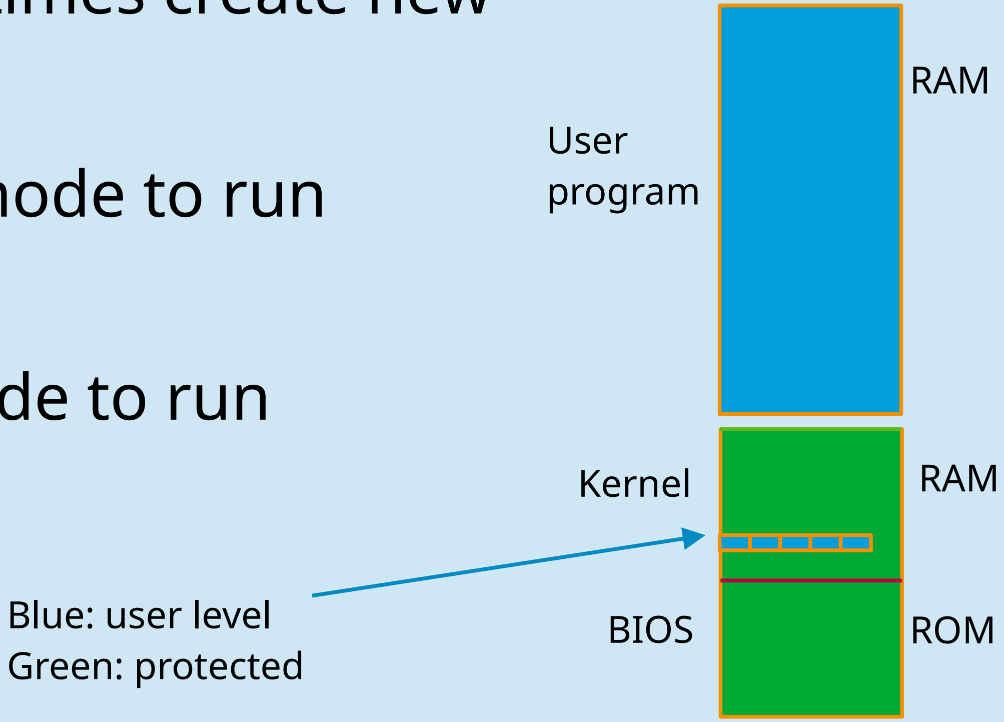


Image: https://en.wikipedia.org/wiki/Protection_ring

## Ooops

Notice a pattern: solutions sometimes create new problems.

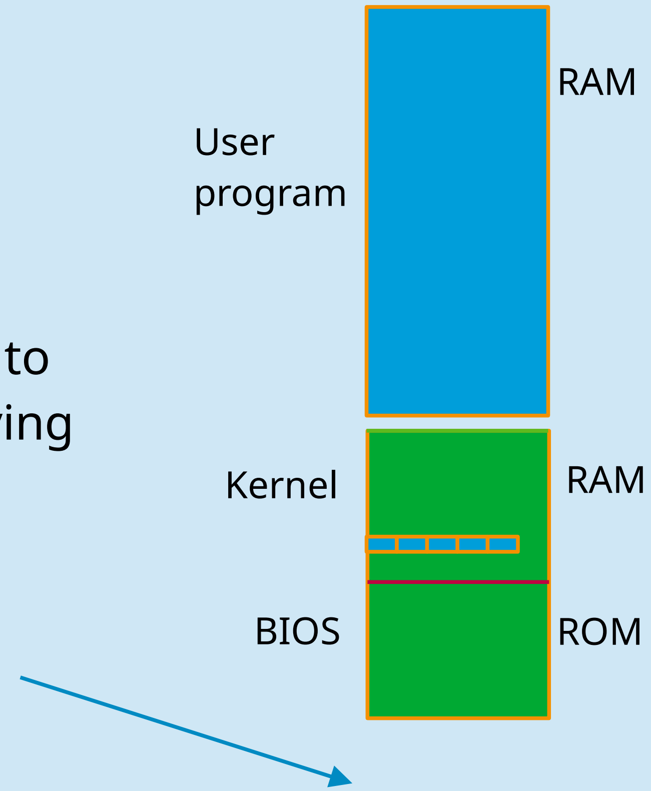
- Need to be in ring 0 / kernel mode to run priviledge instructions
- Need to be in ring 3 / user mode to run unpriviledge (user code)
- How do we switch?



# Handling protection modes

How do we switch between user and kernel mode?

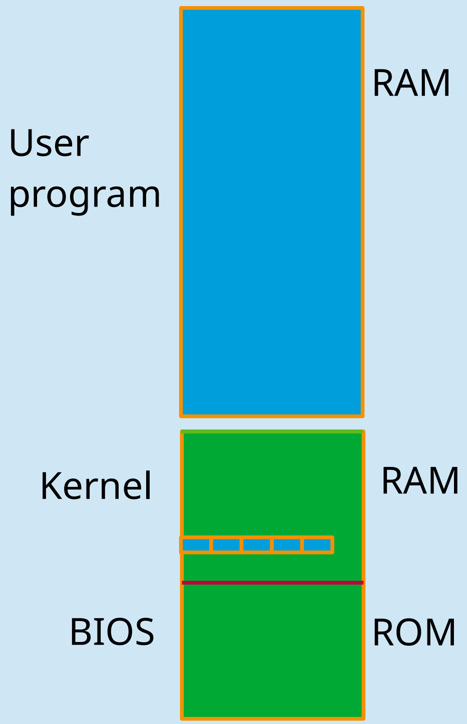
- Computer boots in ring 0 / priviledged
- Instructions switch to user mode.
  Roughly (on intel):
  - First, need to set up a GDT (Global Descriptor Table) to deal with memory protection. Again: we're handwaving memory protection
  - Interrupt table (we'll explain in a bit)
  - Set up user level stack (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?**

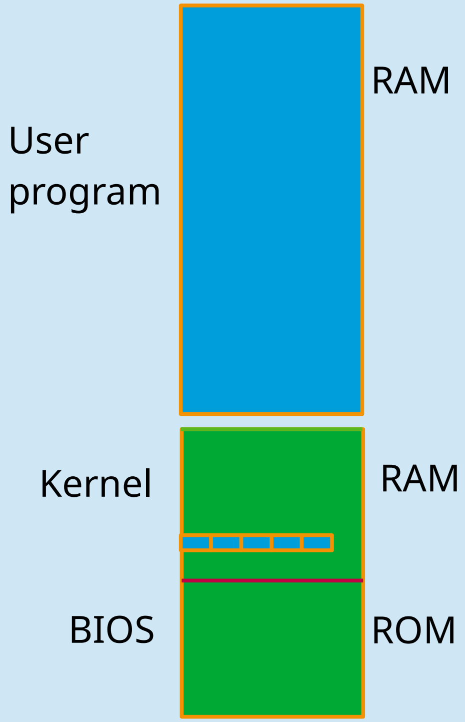
- Ring 0 is priviledged, so user level cannot snoop into kernel or just jump back in.
- Need a mechanism to enter the kernel again, but controlled by the kernel
- System call:
  - User level sets up what it wants to do in the kernel (function ID, data/parameters etc)
  - Traps to the kernel --- somehow
  - 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
  -


User program
RAM
Kernel
RAM
BIOS
ROM