CIS 310 Operating Systems

Week 2: What is a Computer?

Dr. Brian C. Ladd

Thursday 4th November, 2021

Outline

- Computer Architecture
  - The main parts
- Hardware Support for an OS
  - Privilege Bit
  - Interrupts
  - System Calls

Computer Architecture

Definition

Your machine is a *digital*, *binary*, *general-purpose* device.

Computer Architecture

Definition

Your machine is a *digital*, *binary*, *general-purpose* device.

digital
: composed of discrete units (digits).

Computer Architecture

Definition

Your machine is a *digital*, *binary*, *general-purpose* device.

digital composed of discrete units (digits).

binary using base 2, the set of $\{0, 1\}$.

Computer Architecture

Definition

Your machine is a *digital*, *binary*, *general-purpose* device.

digital composed of discrete units (digits).

binary using base 2, the set of $\{0, 1\}$.

general purpose capable of interpreting *encoded* data. Given enough bits, anything can be encoded.

Computer Architecture

Definition

Your machine is a *digital*, *binary*, *general-purpose* device.

- digital: composed of discrete units (digits).
- binary: using base 2, the set of $\{0, 1\}$.
- general purpose: capable of interpreting *encoded* data. Given enough bits, anything can be encoded.

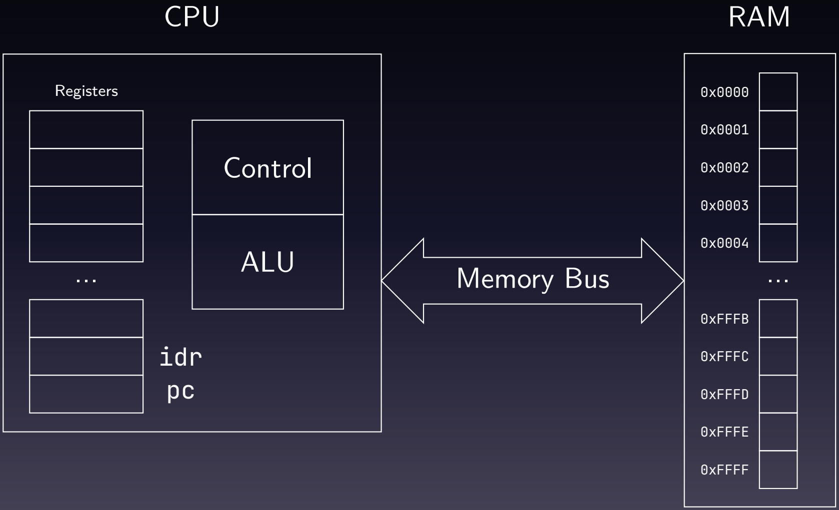
Definition

Our *stored-program* computers can store and interpret **instructions** on how to interpret bits in memory.

von Neumann Architecture

- A single, unified *memory*.
- A single* processor.
- A bidirectional connection between them.

Parts


CPU
RAM
Registers
...
Control
ALU
idr
pc
Memory Bus
0x0000
0x0001
0x0002
0x0003
0x0004
...
0xFFFB
0xFFFC
0xFFFD
0xFFFE
0xFFFF

The Cycle

```
while (not halted):
  fetch instruction from RAM[pc] into idr
  decode idr
  execute
```

What Only Hardware Can Do

- Interrupts

What Only Hardware Can Do

- Interrupts
- Privilege Bit

What Only Hardware Can Do

- Interrupts
- Privilege Bit
- System Calls

What Only Hardware Can Do

- Interrupts
- Privilege Bit
- System Calls
- Address Translation

What Only Hardware Can Do

- Interrupts
- Privilege Bit
- System Calls
- Address Translation
- Atomic Instructions

Kernel/User Mode

- CPU *status register* has a *privilege bit*
  `0` ⇒ **user** mode
  1 ⇒ **kernel** (system) mode

User Mode

- All addresses are *translated* by the hardware according to policy set by the operating system.
- Direct interaction with certain parts of memory is *forbidden*, *e.g.* memory-mapped device ports, OS data structures, interrupt service vector, OS code, hardware timer settings.
- Certain CPU instructions are *forbidden* and will generate an *interrupt* if they are attempted.
- **Limited Direct Execution**

Kernel Mode

- Permit all the forbidden stuff.
- **Direct Execution**

Switching?

How can your program write something to a file if it lives in *user* mode and only the *system* mode can interact with devices that have files on them?

Switching?

How can your program write something to a file if it lives in *user* mode and only the *system* mode can interact with devices that have files on them?

Good Question

We will answer it in a minute.

Next Instruction

How does the CPU select the *next* instruction to execute?

Next Instruction

How does the CPU select the *next* instruction to execute?

The value in the `pc` register

```
fetch

decode

execute
```

Next Instruction

How does the CPU select the *next* instruction to execute?
The value in the `pc` register

`fetch`

`decode`

`execute`

What if some *hardware* or even some *software* needs immediate service?

Next Instruction

How does the CPU select the *next* instruction to execute?
The value in the `pc` register

```
fetch
decode
execute
```

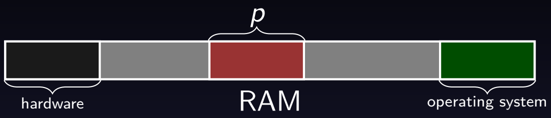
What if some *hardware* or even some *software* needs immediate service?
An **Interrupt**

Taking an Interrupt

- Before `fetch`: check for pending interrupt.
- With highest priority pending interrupt
  - Set *privilege bit*
  - Save *context* of user program on **k-stack**
  - Use *interrupt number* is index into `ISV`: jump to the address.
  - Run *interrupt service routine* using normal `fetch-decode-execute` cycle (but with privilege).

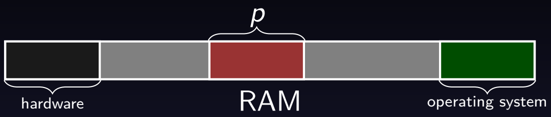
A Process in RAM

Machine RAM with a process, p in it.

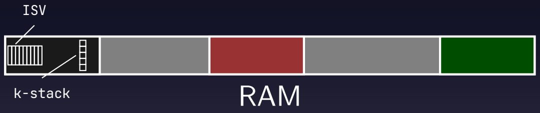


A Process in RAM

Machine RAM with a process, p in it.

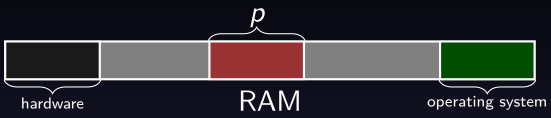


Looking more closely at hardware handled memory.

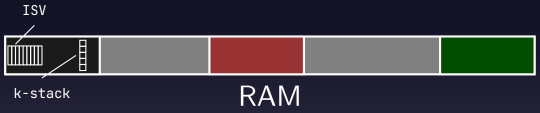


A Process in RAM

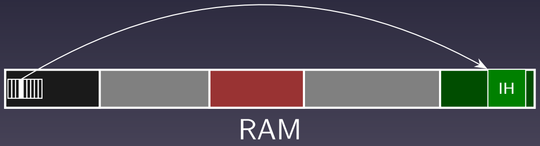
Machine RAM with a process, p in it.



Looking more closely at hardware handled memory.



Taking an interrupt.



Returning from an Interrupt

  - Ensure the context of appropriate user *process* is on `k-stack`.
  - Unset *privilege bit*
  - Restore context from `k-stack`.
- Restart instruction pointed to by `pc` for user code.

Big Idea

The idea of a **jump table** (or *service vector*) is very useful in building a flexible interface.

- Outside world provides a number indicating an event or request.
- Inside, we have an array of pointers at code. Indexing by event gives us the code to run.
- **Interface** because we publish the list of numbers we expect for different events (or accommodate such a list provided by a hardware manufacturer).
- **Flexible** because we can change how we handle any given event just by providing code and pointing at it.

System Call

- Can we avoid building a *new* system for user code to do system things?

System Call

- Can we avoid building a *new* system for user code to do system things?
- **Interrupt** system already promotes from *user* $\Rightarrow$ *system* mode.

System Call

- Can we avoid building a *new* system for user code to do system things?
- **Interrupt** system already promotes from *user* $\Rightarrow$ *system* mode.
- Still do not want to have user code executing in system mode (**Never** trust users. **Never!**)

System Call

- Can we avoid building a *new* system for user code to do system things?
- **Interrupt** system already promotes from *user* $\Rightarrow$ *system* mode.
- Still do not want to have user code executing in system mode (**Never** trust users. **Never!**)
- Can the OS leverage this to permit *user* mode software to *request* privileged services from the OS.

System Call

- Can we avoid building a *new* system for user code to do system things?
- **Interrupt** system already promotes from *user* $\Rightarrow$ *system* mode.
- Still do not want to have user code executing in system mode (**Never** trust users. **Never!**)
- Can the OS leverage this to permit *user* mode software to *request* privileged services from the OS.
- **System Call** or a *software interrupt*.

Handling a System Call

- OS knows what kind of *hardware* interrupt happened by the *interrupt number*.

Handling a System Call

- OS knows what kind of *hardware* interrupt happened by the *interrupt number*.
- User code has to ask for many **different** actions from the OS.

Handling a System Call

- OS knows what kind of *hardware* interrupt happened by the *interrupt number*.
- User code has to ask for many **different** actions from the OS.
- Yet `syscall` always generates the same interrupt.
  How can the *user* communicate the function they wish to request?

System Call Function Numbers

The OS can define that certain registers, in addition to being saved during a `syscall`, will also be read or written by the OS. The registers provide the function number and *parameters* for the `syscall`.

How does the OS call the right code for a given function?

Program Writes a String

```
mov   rax,  1   ; system call number for write
mov   rdi,  1   ; param 1: file descriptor (FD) number.
                ; FD 1 is stdout
mov   rsi, msg  ; param 2: address of string
mov   rdx, len  ; param 3: length of string
syscall         ; trap into OS kernel
```

By Linux convention

`rax` is the function number and for write

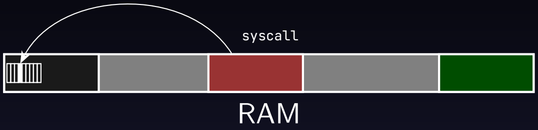
`rdi` is a file descriptor

rsi points to an array of char

rdx is length of array of char

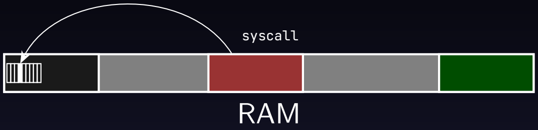
Requesting Privileged Access

What happens when our program makes a system call?

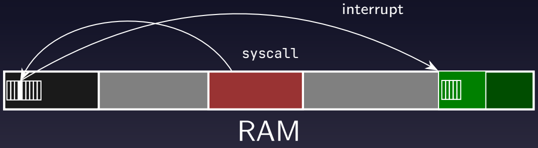


Requesting Privileged Access

What happens when our program makes a system call?

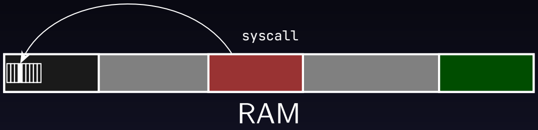


`syscall` generates an interrupt. Control passes through the ISV.

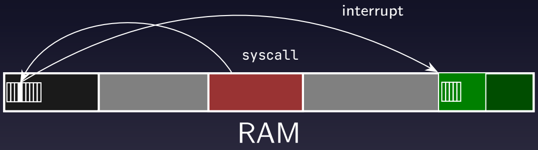


Requesting Privileged Access

What happens when our program makes a system call?



`syscall` generates an interrupt. Control passes through the ISV.



To the `syscall` handler with its own jump table.