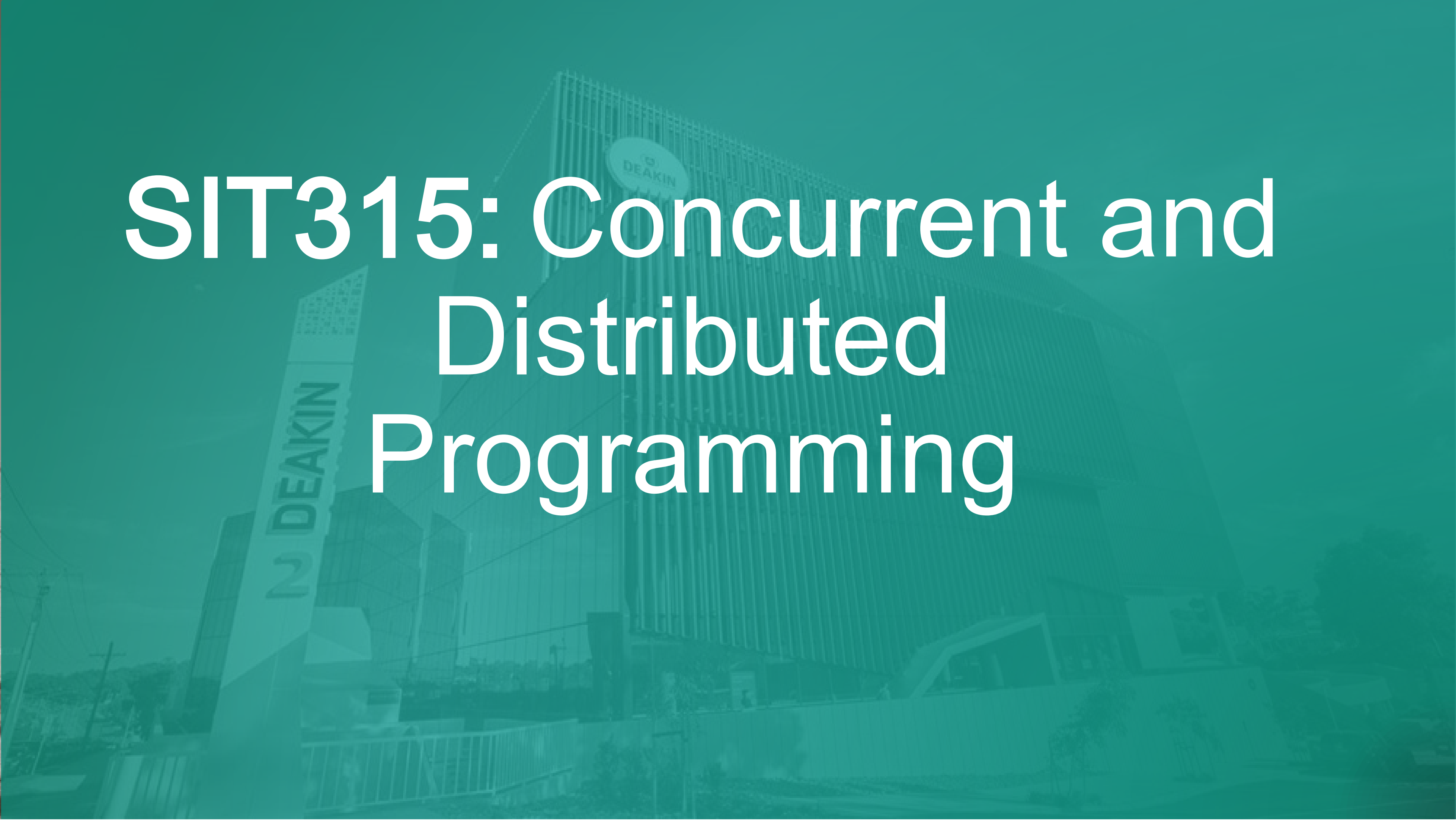


SIT315: Concurrent and Distributed Programming

SIT315 ULOs

ULO1: Assess the suitability of different programming paradigms for a range of problem types to facilitate effective language selection and program design

ULO2: Design and implement programs in languages encompassing different programming paradigms to demonstrate effective and efficient computational problem solving

ULO3: Apply the theoretical aspects of different programming paradigms to analyse and critique the design of programs

SIT315 Unit Structure

This unit has five modules (programming paradigms):

1. **M1:** Real-time systems (W1 & W2): Interrupts, timers, scheduling

2. **M2:** Concurrent systems (W3, W4, W5): Multi-threading

3. **M3:** Distributed systems (W6, W7, W8): Multi-core, distributed memory & Hybrid computing systems

4. **M4:** Parallel patterns (W9, W10, W11): Parallel programming patterns e.g. MapReduce and ProducerConsumer, CAP Theorem, Blockchain, Docker and container orchestration,..

SIT315 Assessment

This is a portfolio-unit, all tasks/assessment are OnTrack. Each module has assessment tasks - see OnTrack, E.g M1.T1P is a pass task number 1 in module 1.

A **Pass** in this unit means a student can design, implement, assess and critic **simple** programs in different programming paradigms.

A **Credit** in this unit means a student can design, implement, assess and critic **Intermediate** programs in different programming paradigms.

A **Distinction and High Distinction** in this unit means **Credit** plus a project demonstrating skills beyond what was covered in the module tasks. The grade will depend on the complexity and quality of the project.

SIT315 Example projects

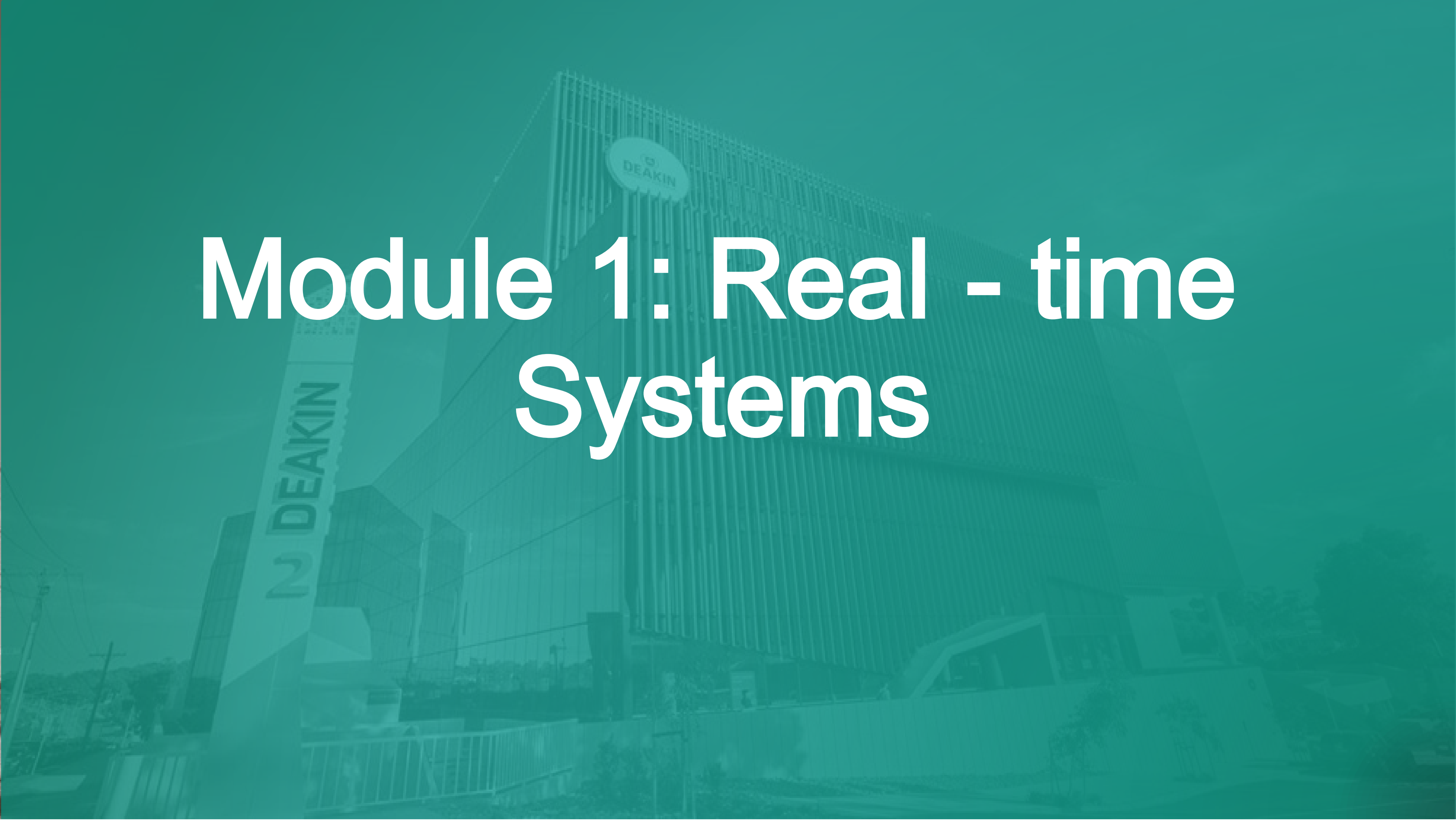
Distinction and High Distinction Projects are either:

- **Module++ Project:** Demonstrate advanced concepts in one of modules

- **Multi-module Project:** Demonstrate advanced concepts across multiple modules

E.g:

- Traffic Light Systems [Real-time++]
- Multi-robot Localisation [Real-time + Distributed]
- Cluster computing platform



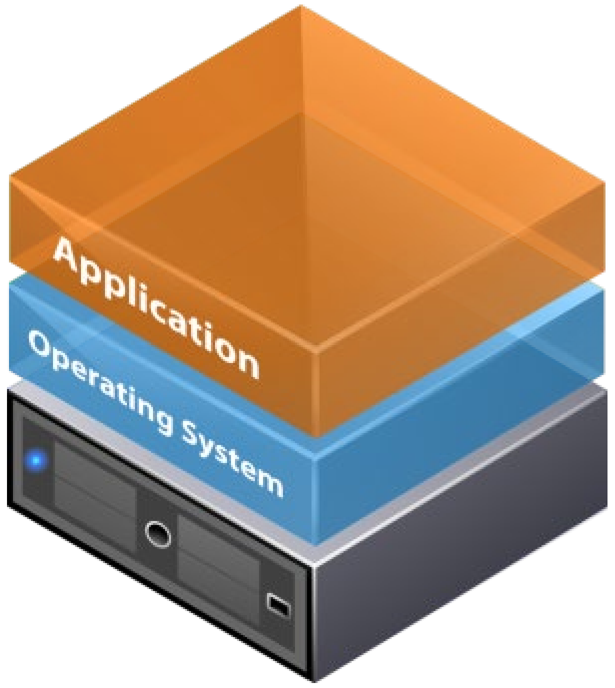
Module 1: Real - time Systems

Getting Started

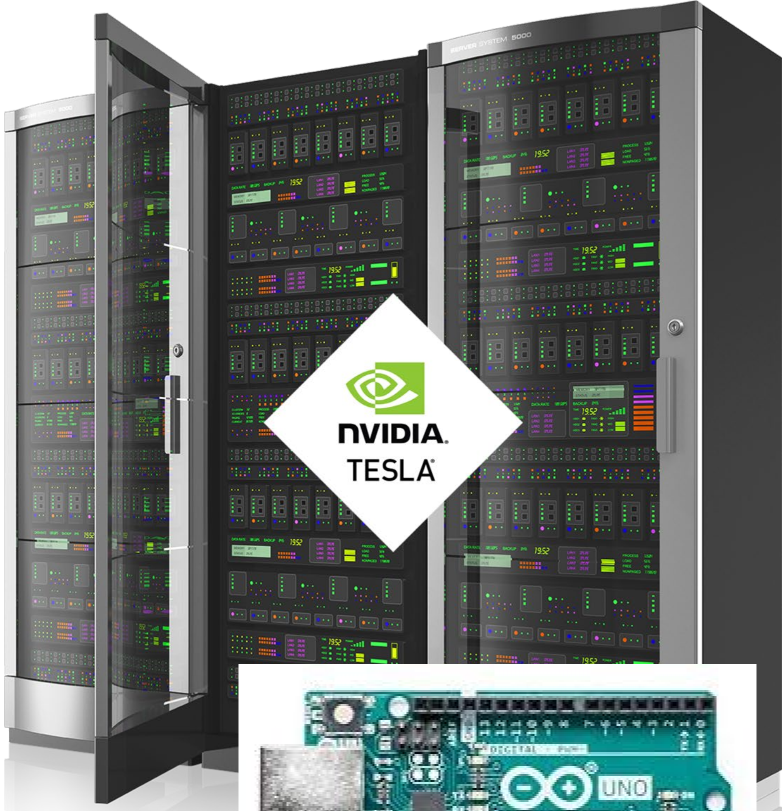
As programmers (Computer Scientist, Software Engineer) building apps that run on different platforms, we need to understand what happens under the hood

- Hardware
- Operating Systems
- Applications

In this lecture, we will review the relevant details of these three layers

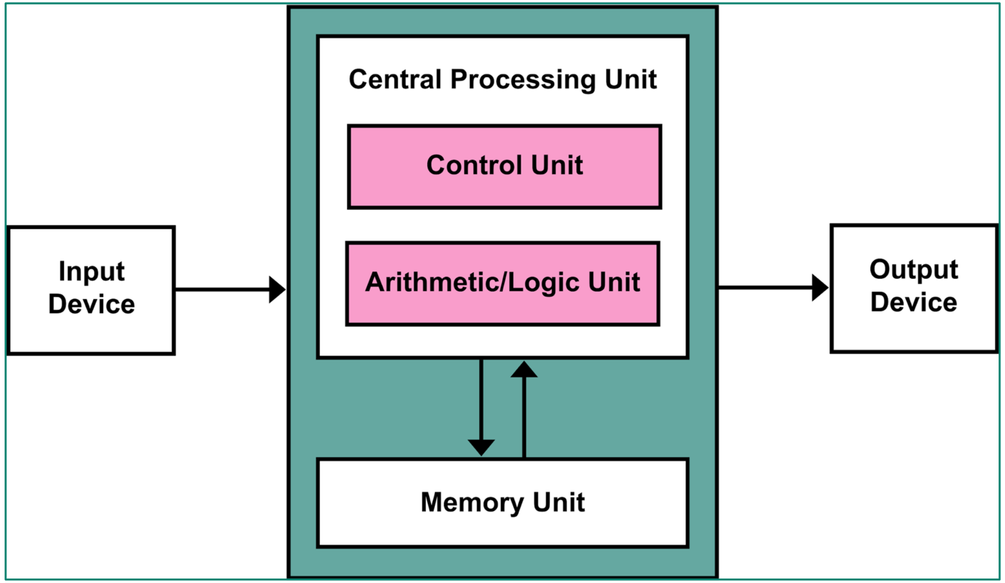


Hardware



Computer Architecture

Most of the normal PCs and laptops

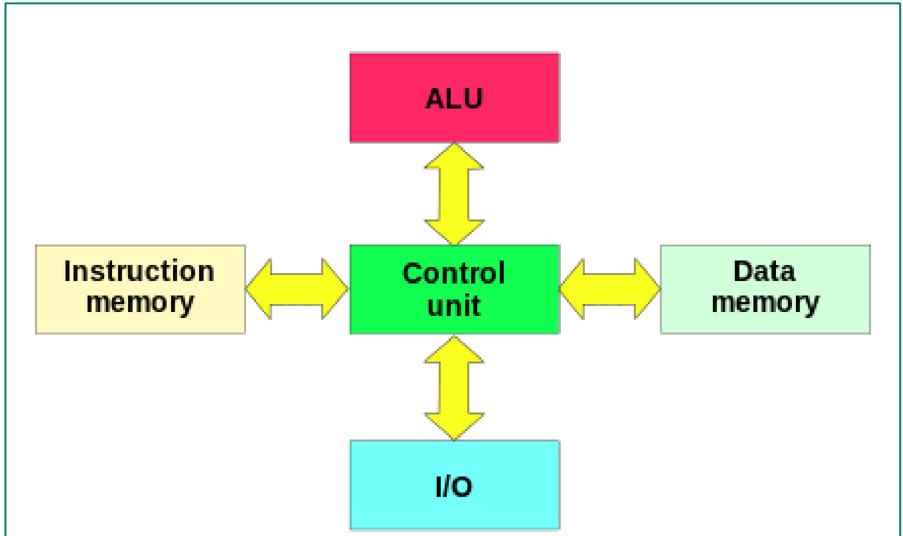


Von Neumann architecture

Same physical memory address is used for instructions and data.

There is common bus for data and instruction transfer.

Arduino

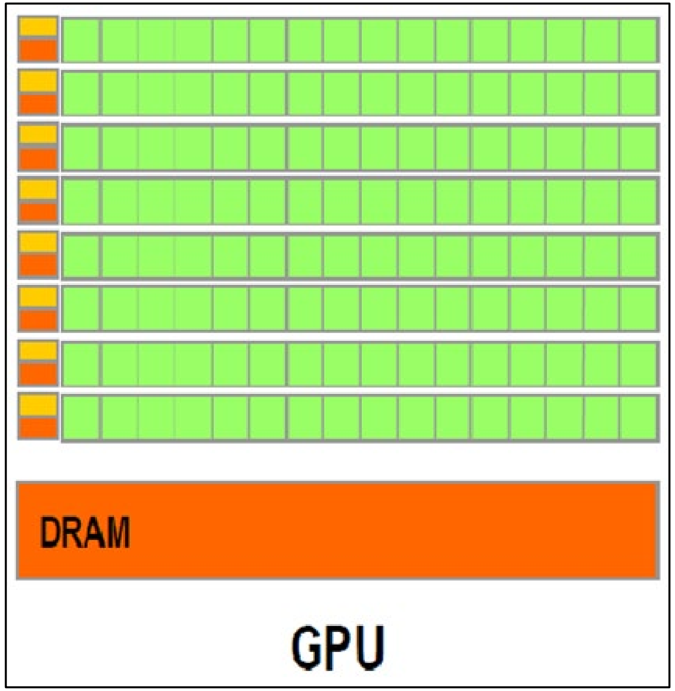


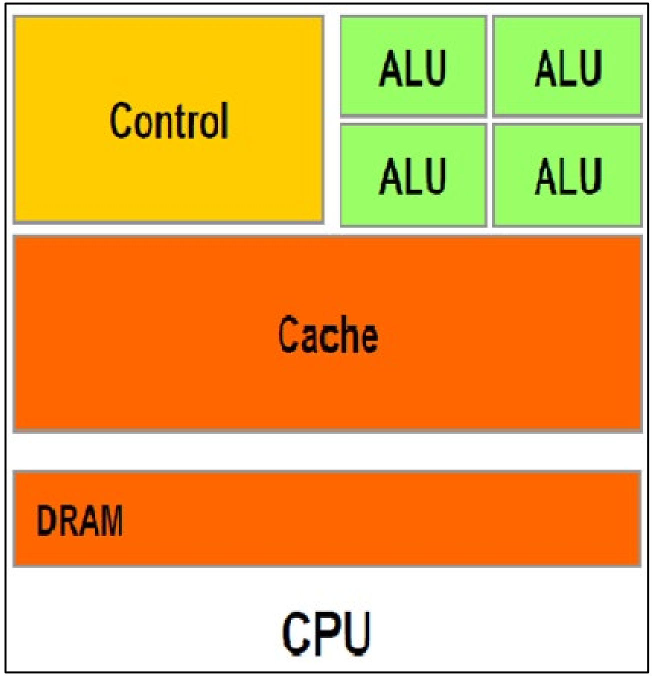
Harvard architecture

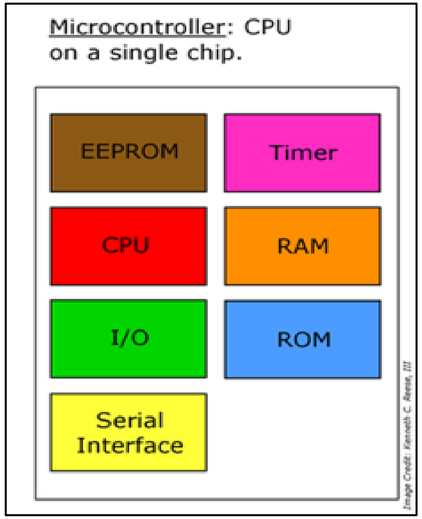
Separate physical memory address is used for instructions and data.

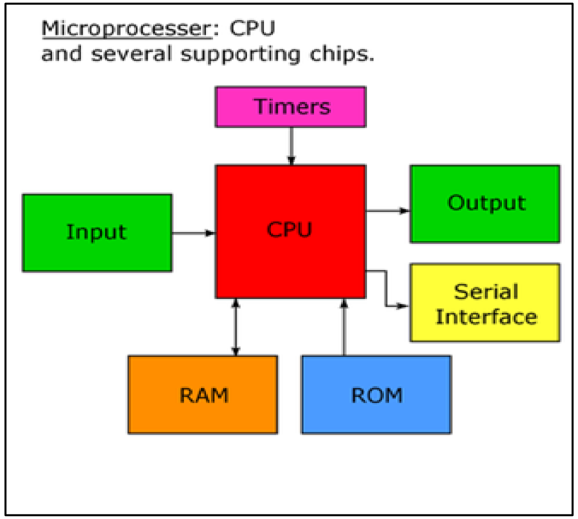
Separate buses are used for transferring data and instruction.

CPUs/Microprocessors






Microcontroller: CPU on a single chip.
EEPROM
Timer
CPU
RAM
I/O
ROM
Serial Interface
Image Credit: Kenneth C. Reese, III



CPUs/Microprocessors

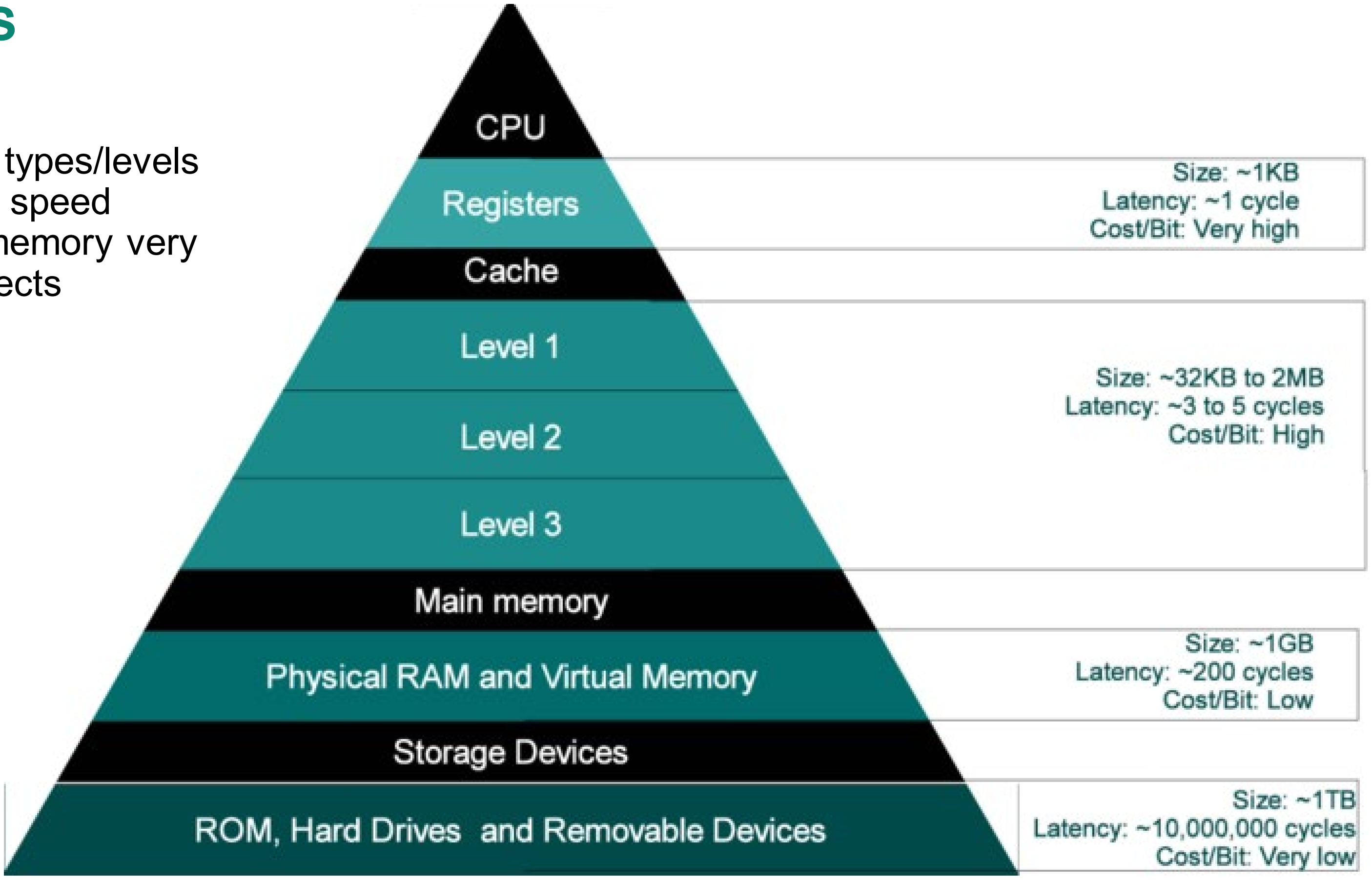
A CPU/Microprocessor has Arithmetic Logic Unit (ALU) and Control Logic Unit (CLU) that manages the ALU. Special purpose designs:

- A digital signal processor (DSP) is specialized for signal processing.
- Graphics processing units (GPUs) are processors designed primarily for realtime rendering of 3D images. GPUs are evolving into increasingly general-purpose stream processors
- Microcontrollers integrate a microprocessor with peripheral devices in embedded systems - e.g. in Arduino
- Systems on chip (SoCs) often integrate one or more microprocessor or microcontroller cores - e.g. mobile phones, Raspberry PI,..

Memory Levels

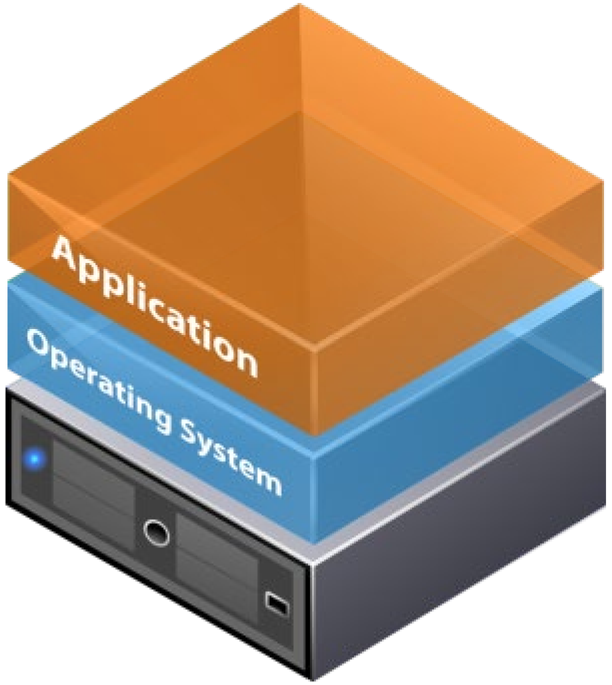
- Different memory types/levels
- Vary in size, cost, speed
- We need to use memory very efficiently as it affects performance!

Fastest, smallest, expensive



Slowest, biggest, cheapest

Operating System



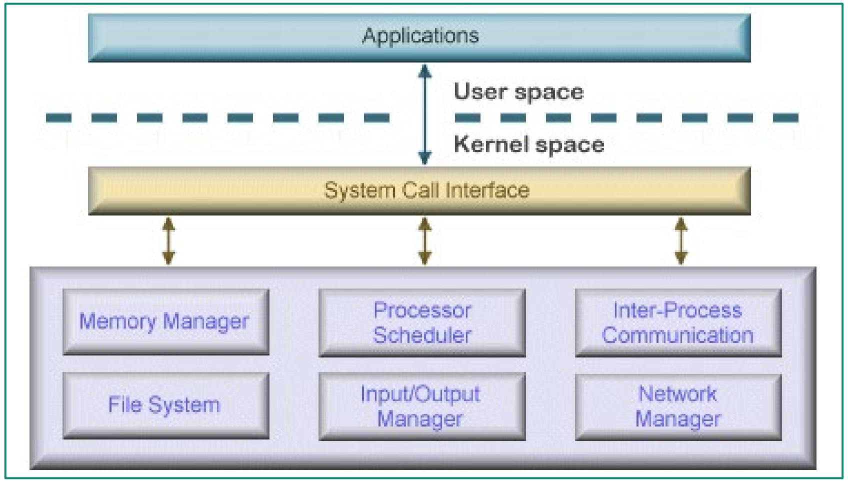
Operating Systems

Software that manages computer hardware and software resources and provides common services for your programs. Operating Systems should provide functionalities including:

- Memory Management
- Processor Scheduler
- Interprocess Communication
- File System
- IO Manager
- Network Manager

Types of OS:

- Single/multi-tasking OS
- Distributed OS
- Embedded OS
- Real-time OS



OS Processes

A program in execution is called a process. OS has a set of processes including system processes and user processes.

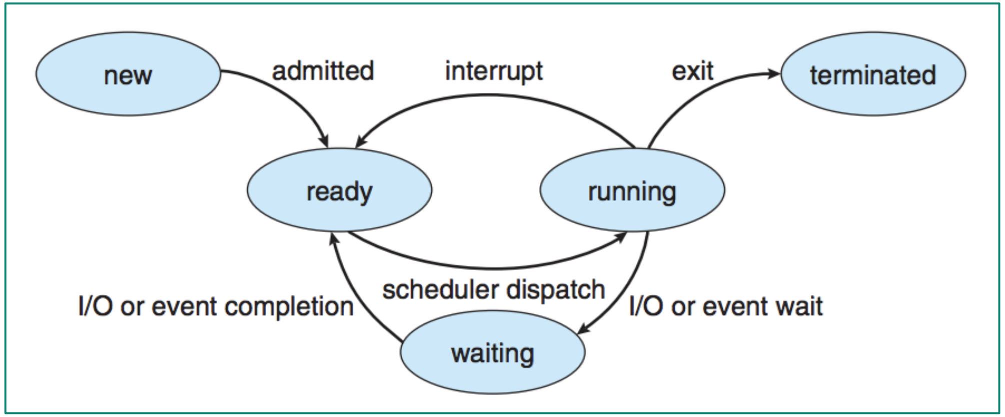
A process needs resources to accomplish its task including:

- CPU time,
- Memory,
- System files,
- Inter-process communication
  (Shared memory or message passing), and
- I/O devices.

Process in memory

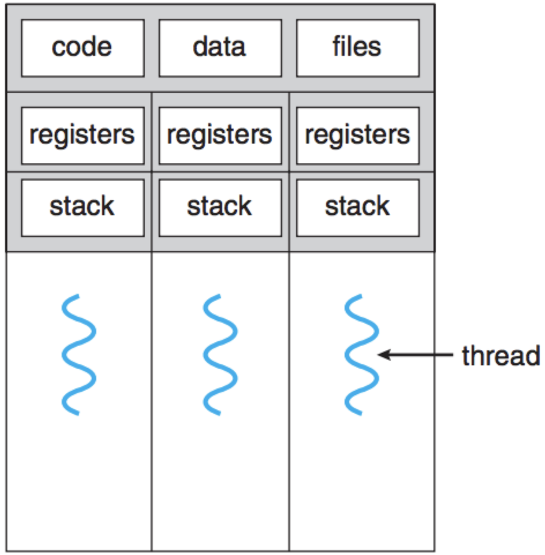
Stack (tmp vars)
Heap (dynamically allocated data)
Data (global vars)
Text (program code)

Process States



Threads

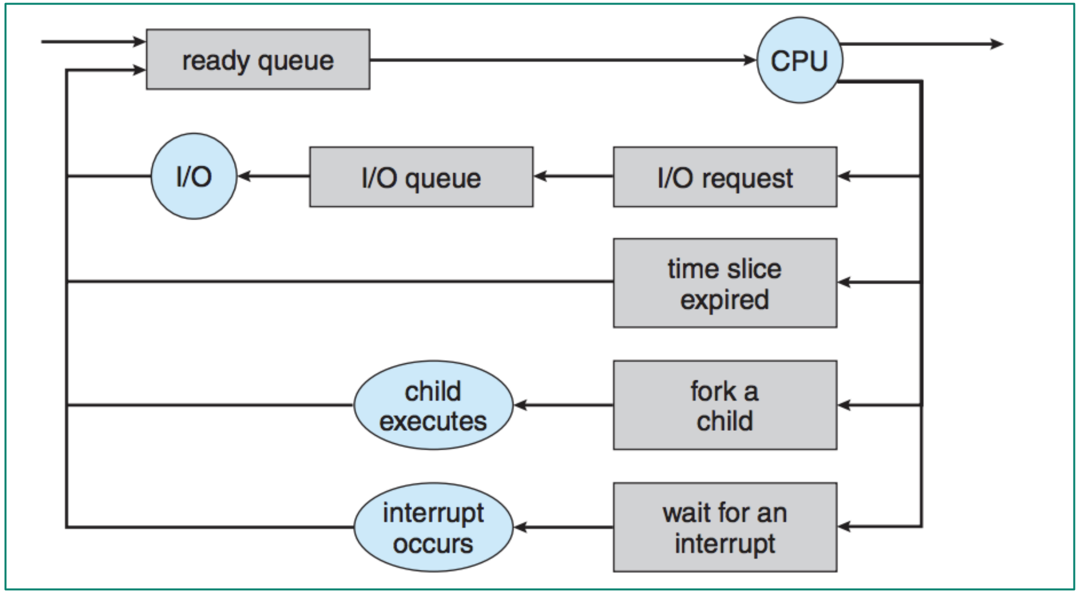
- A thread is the unit of execution within a process.
- A process can have multiple threads of execution to allow a process to perform more than one task at a time.
- Threads share process memory and resources, while each thread has its own stack and registers.
- Threads are the primary units of the proce scheduled for execution.

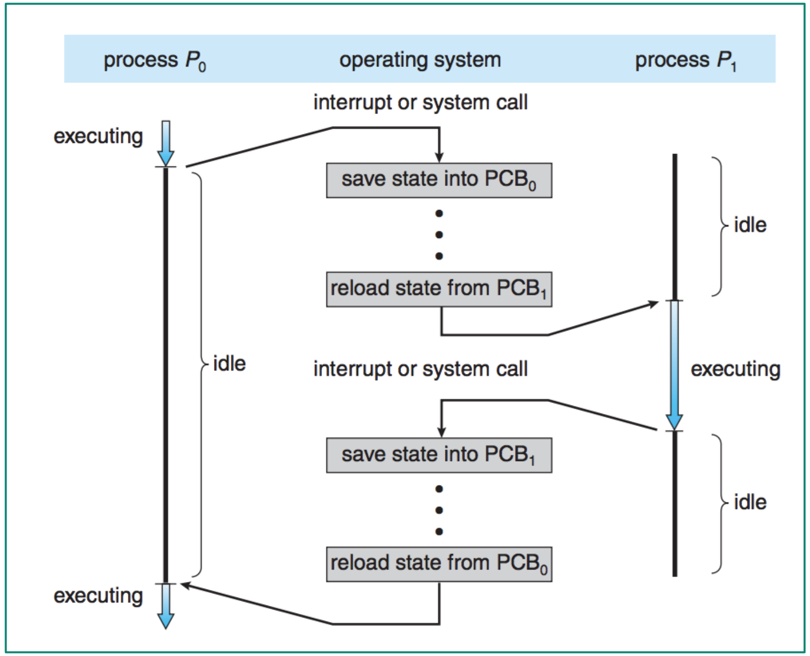


multithreaded process

OS CPU Scheduling

Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. During I/O time, the CPU will be idle! Hence we can run another process...





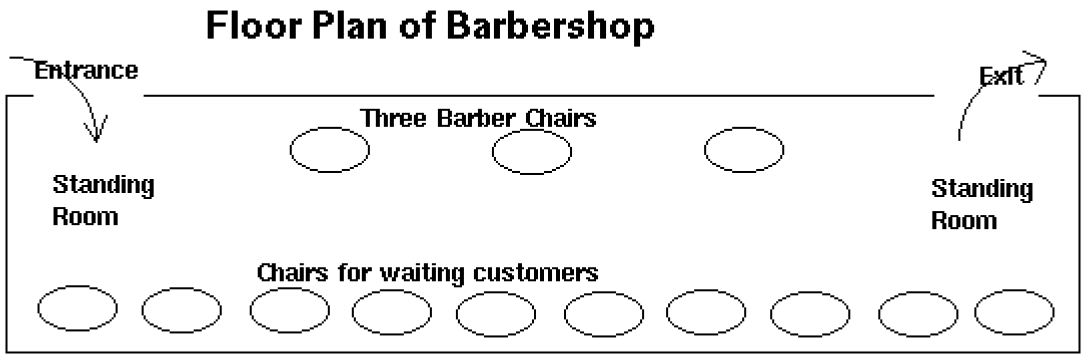
* PCB is a Process Control Block struct used in the OS kernel to track (linked list) running processes

OS CPU Scheduling

When an operating system allows more than one process to be loaded and share the CPU, we need to have a scheduler to decide which process should use the CPU at any given point in time.

Key terms:

- Arrival time
- Completion time
- Waiting time
- Burst time



Scheduling algorithms

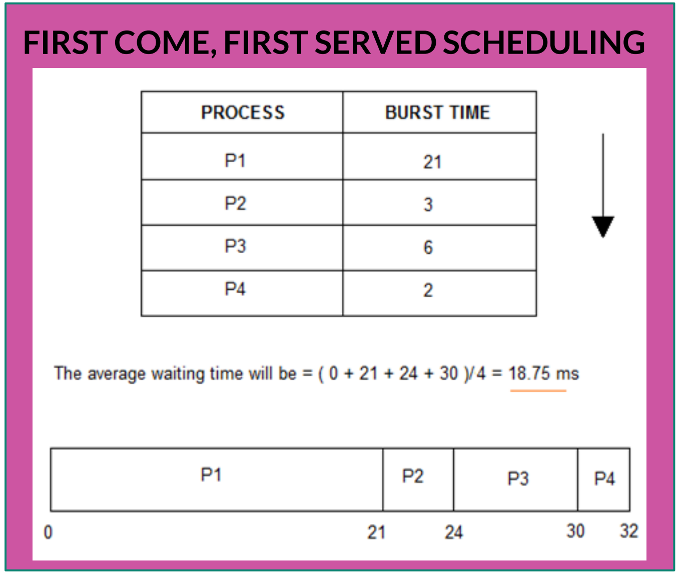
There are many scheduling algorithms in the literature:

- Non-preemptive - once allocated to CPU, it stays until completed or waiting for IO.
- Preemptive - Interrupted/swapped every X time units.

Example scheduling algorithms (few below):

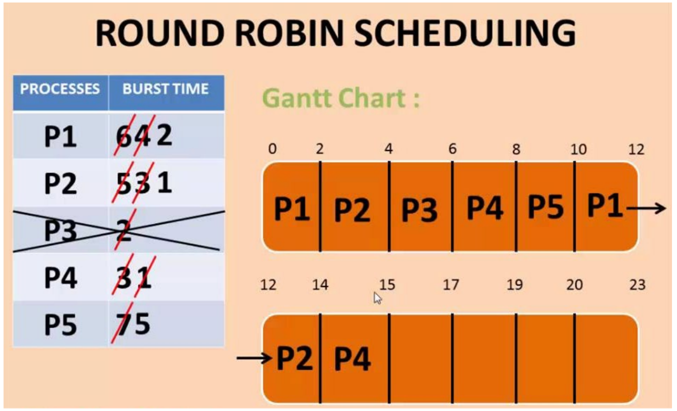
- First come, first served -- FCFS [it only depends on the arrival time]
- Round-Robin - all tasks have equal time sharing on the CPU
- Priority-based scheduling [highest priority first]

Pros & Cons? When to use each of these algorithms?


FIRST COME, FIRST SERVED SCHEDULING

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

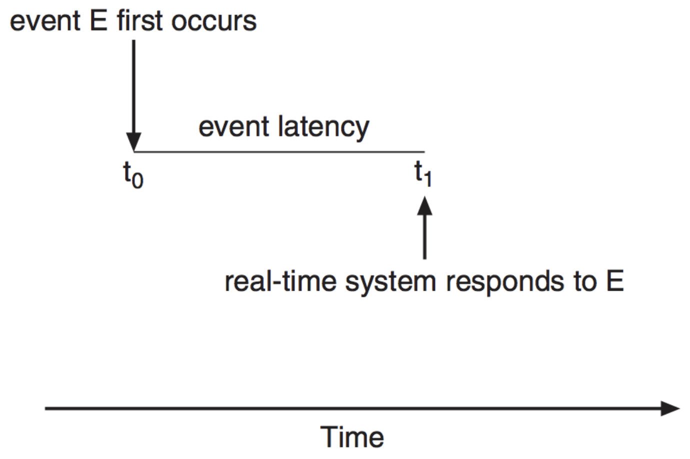
The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms

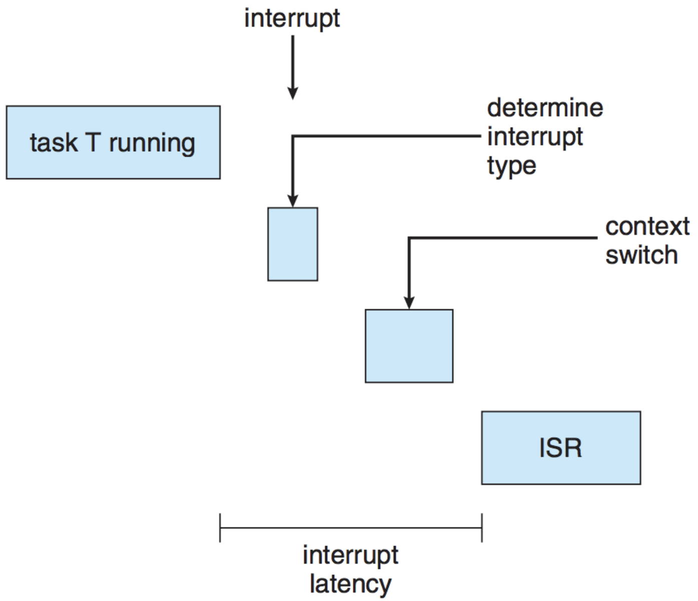

ROUND ROBIN SCHEDULING

PROCESSES	BURST TIME
P1	~~6~~~~4~~2
P2	~~5~~~~3~~1
~~P3~~	~~2~~
P4	~~3~~~~1~~
P5	~~7~~5

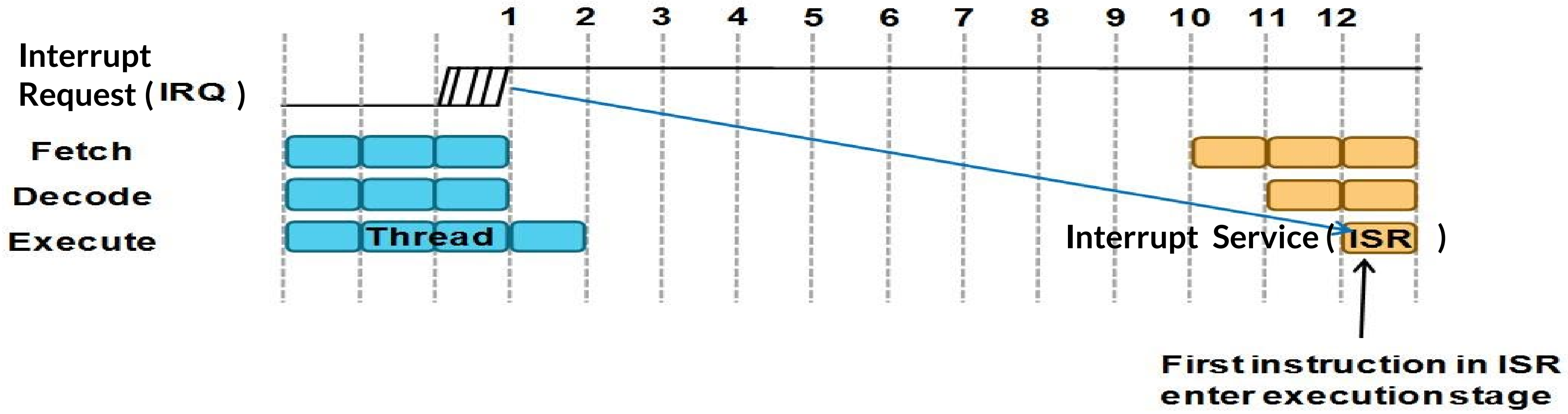
Polling and Interrupts

In **Polling**, Processor is continuously monitoring the devices and when a request is ready, it handles the requested service. In **interrupt**, hardware can signal to the processor when they have something to do -- Which one is more efficient?





Interrupt latency


1 2 3 4 5 6 7 8 9 10 11 12
Interrupt Request (IRQ)
Fetch
Decode
Execute
Thread
Interrupt Service (ISR)
First instruction in ISR enter execution stage

* Each IRQ line has a numeric number (Id) to differentiate between devices. Once an interrupt is received, the processor has to pause, save context, and start processing the interrupt (ISR)

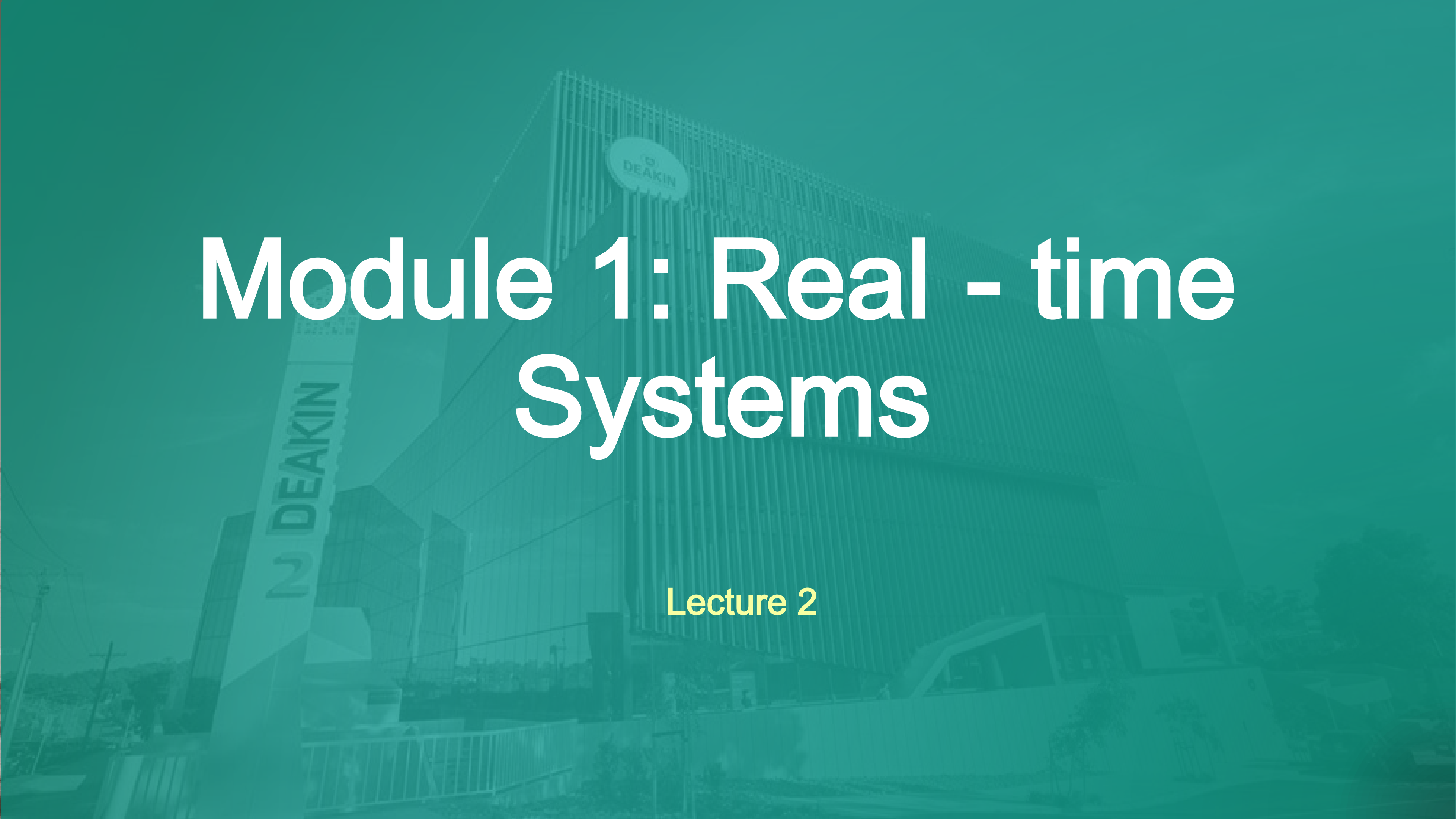
* ISR is the function/routine run by the OS to handle the interrupt -- usually part of the device driver.

Context switching

- Switching the CPU to another process requires performing a state save of the current process and a state restore of the new process. This task is known as a context switch.

- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

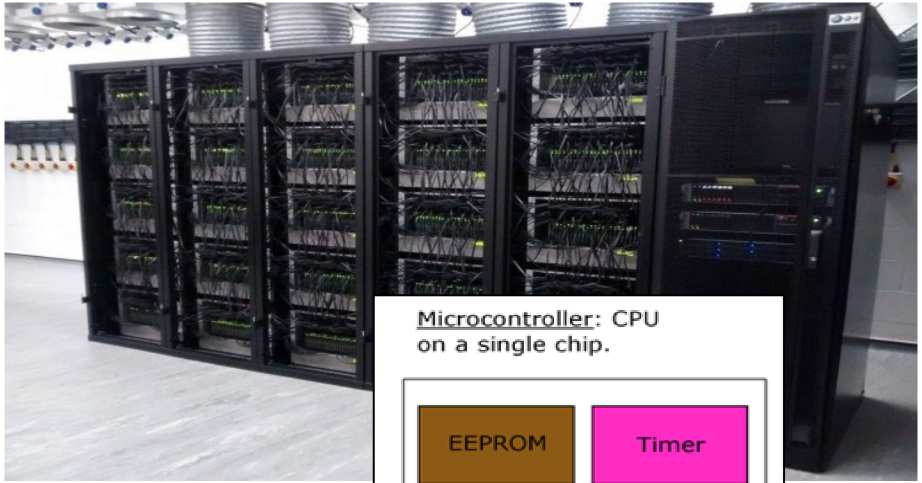
- Context-switch time is pure overhead.

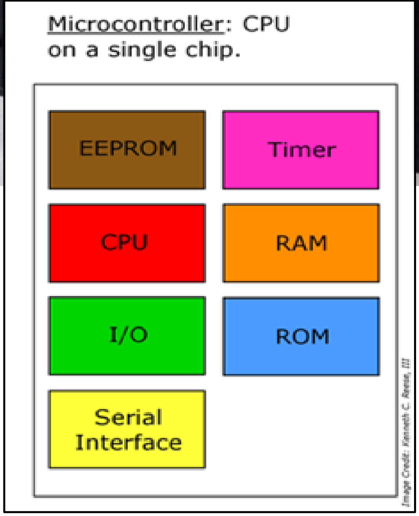


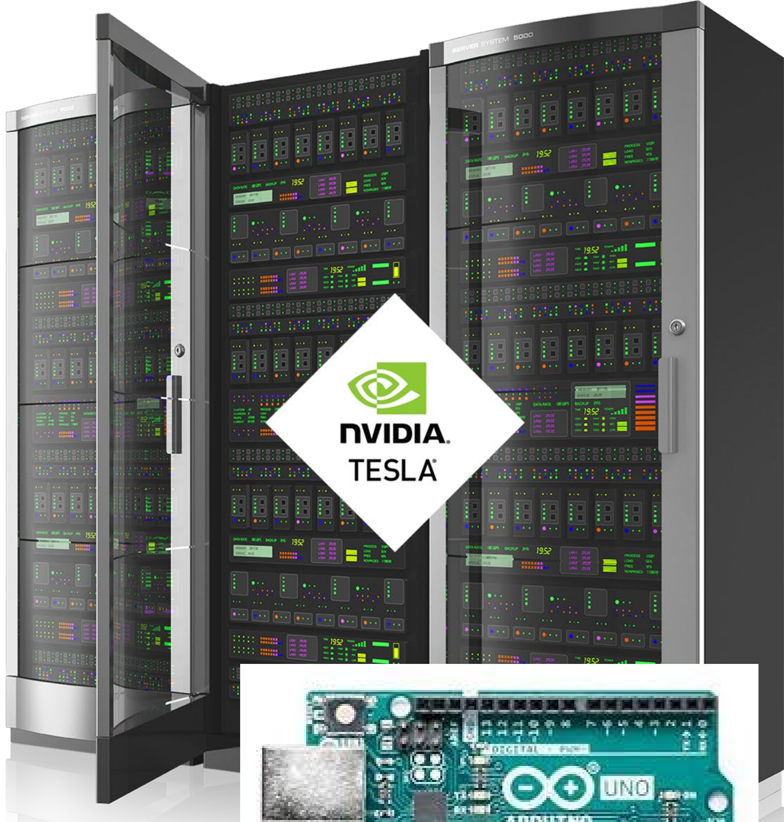
Module 1: Real - time Systems

Lecture 2

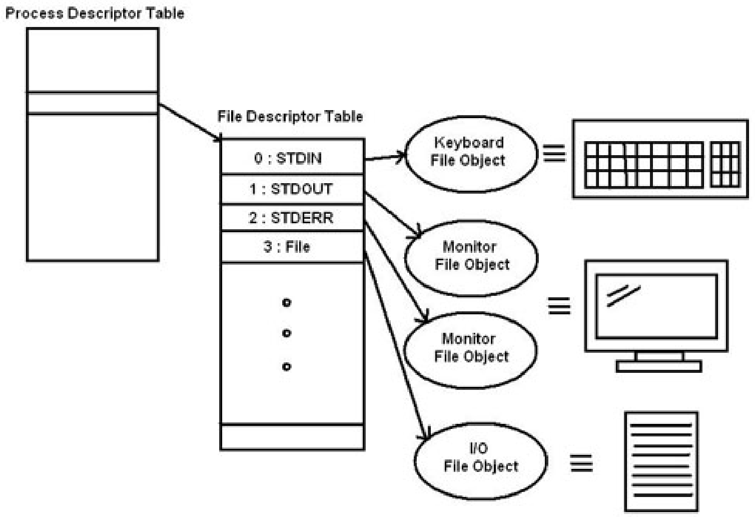
Recap - Hardware



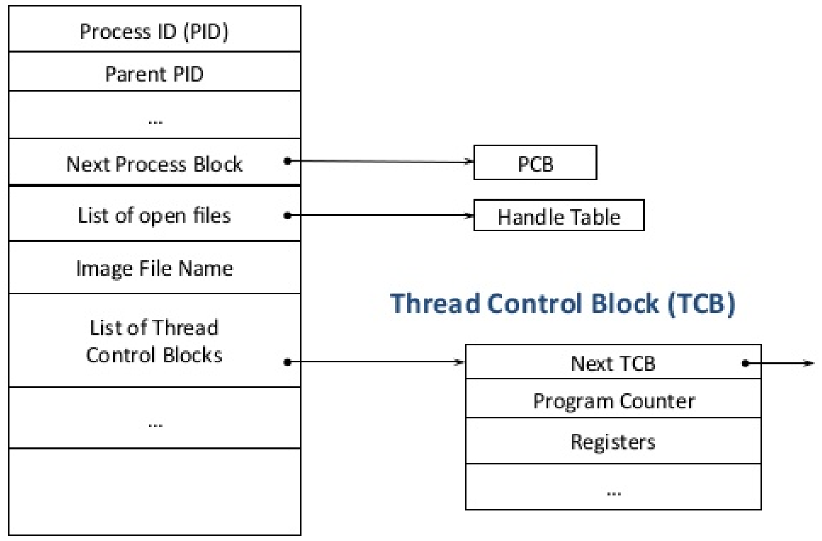

Microcontroller: CPU on a single chip.
EEPROM
Timer
CPU
RAM
I/O
ROM
Serial Interface
Image Credit: Kenneth C. Reese, III



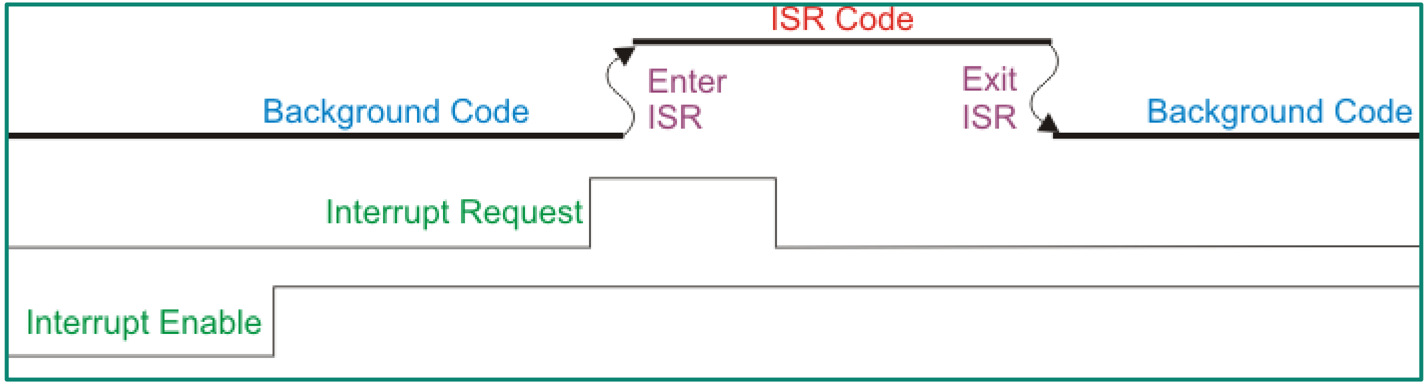
Recap - Processes and Threads

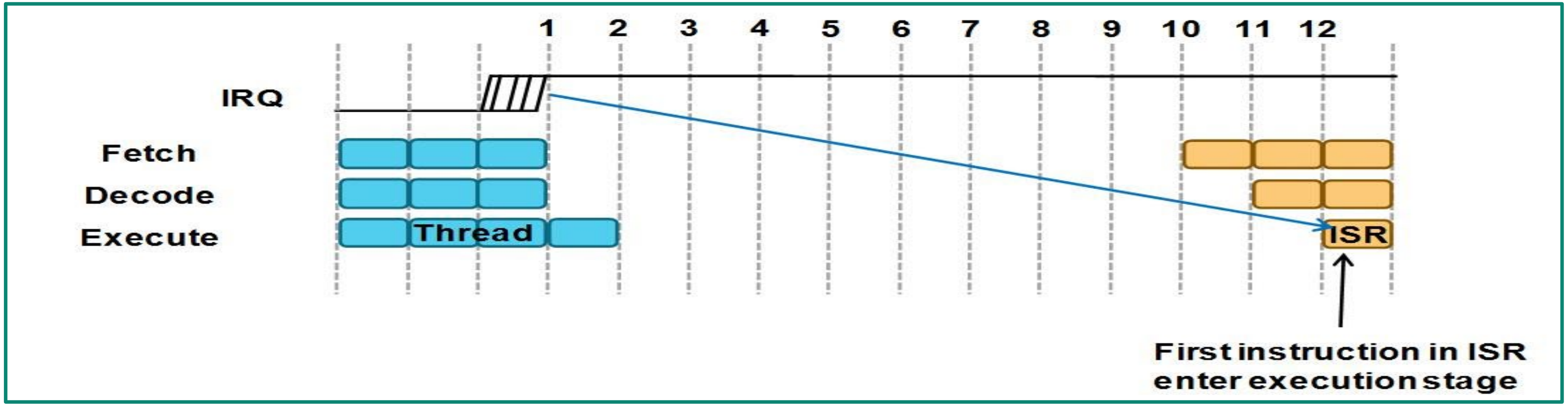


Control Blocks



Recap - Interrupts





Programming With/without interrupts

Keep working until interrupted vs keep checking until pressed

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
volatile int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
  // Attach an interrupt to the ISR vector
  attachInterrupt(0, pin_ISR, CHANGE);
}

void loop() {
  // Nothing here!
}

void pin_ISR() {
  buttonState = digitalRead(buttonPin);
  digitalWrite(ledPin, buttonState);
}
```

Interrupt - driven programming - what could go wrong?

ISRs may involve passing data from/to background code causing corruption of the data, for example:

- Background code loads data1
- Interrupt comes along, updates data1 and data2
- Background code loads data2
- Background code calculates data1 Operation data2

To avoid this problem, we could use "atomic access", implying that the access is indivisible. Any section of code that must have indivisible access to data (or any other resource, for that matter) is called a "critical section." This means we could disable interrupts that access the same data until the atomic access is complete.

In Arduino, we declare shared variables **volatile** so the compiler will trigger re-load anytime we try to use the variables.

Arduino IDE and TinkerCAD

What happens from writing Sketch files until your Arduino Program is up and running?

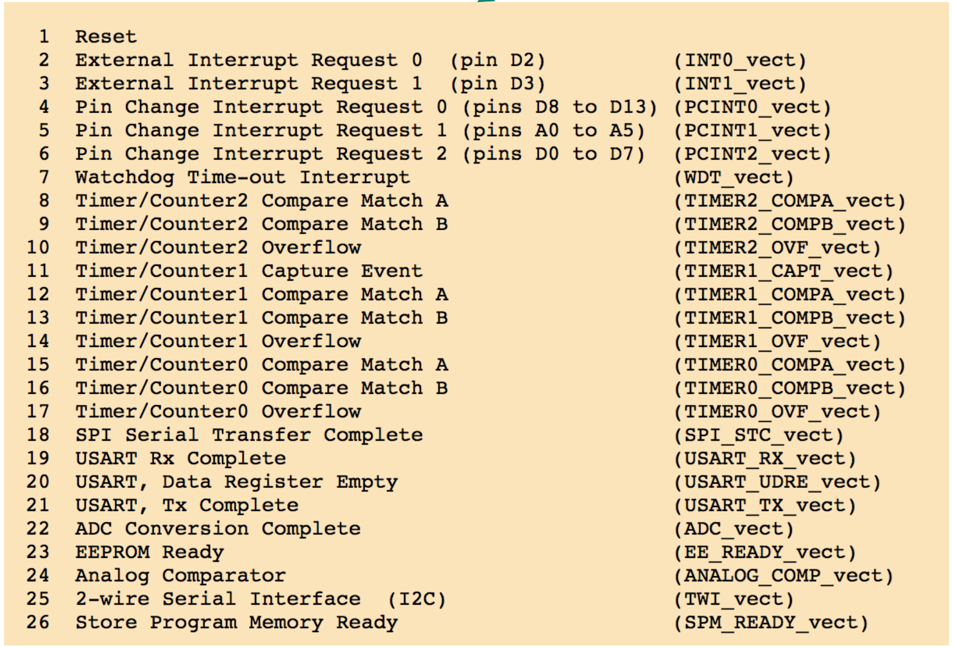
- You write your code in Arduino IDE in Sketch
- Code gets pre-processed into C++
- Code then gets compiled
- Output gets linked to libs
- Hex output file sent/uploaded flashed to the board - avrdude tool
- Program gets stored in the non-volatile Flash memory
- While your program is running, it uses volatile RAM to store the variables you have in the program
- Any program settings you want to persist (non-volatile), you can put/read from EEPROM

See more here: https://github.com/arduino/Arduino/wiki/Build-Process and here: https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5-3rd-party-Hardware-specification

Possible Arduino Interrupts

Void function, no params

attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);

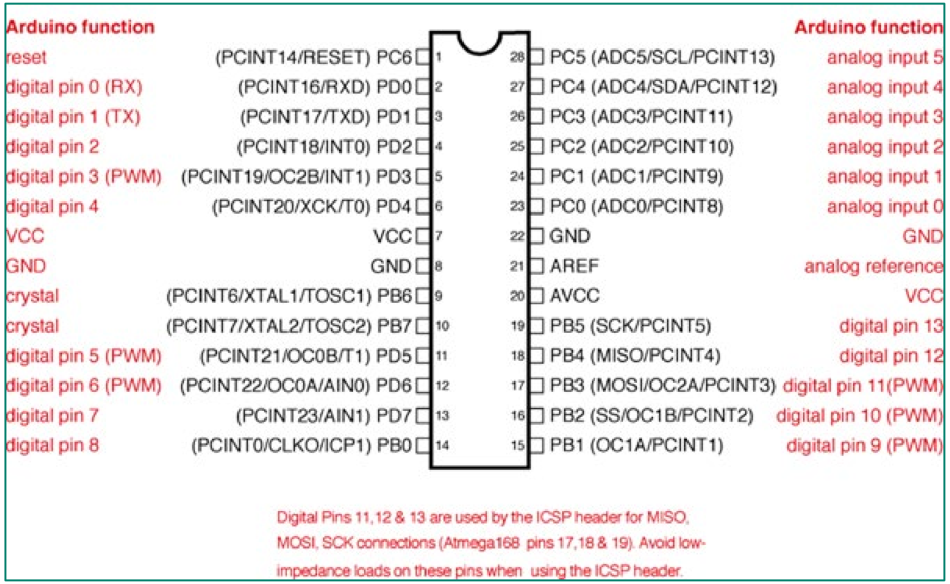


1	Reset	
2	External Interrupt Request 0 (pin D2)	(INT0_vect)
3	External Interrupt Request 1 (pin D3)	(INT1_vect)
4	Pin Change Interrupt Request 0 (pins D8 to D13)	(PCINT0_vect)
5	Pin Change Interrupt Request 1 (pins A0 to A5)	(PCINT1_vect)
6	Pin Change Interrupt Request 2 (pins D0 to D7)	(PCINT2_vect)
7	Watchdog Time-out Interrupt	(WDT_vect)
8	Timer/Counter2 Compare Match A	(TIMER2_COMPA_vect)
9	Timer/Counter2 Compare Match B	(TIMER2_COMPB_vect)
10	Timer/Counter2 Overflow	(TIMER2_OVF_vect)
11	Timer/Counter1 Capture Event	(TIMER1_CAPT_vect)
12	Timer/Counter1 Compare Match A	(TIMER1_COMPA_vect)
13	Timer/Counter1 Compare Match B	(TIMER1_COMPB_vect)
14	Timer/Counter1 Overflow	(TIMER1_OVF_vect)
15	Timer/Counter0 Compare Match A	(TIMER0_COMPA_vect)
16	Timer/Counter0 Compare Match B	(TIMER0_COMPB_vect)
17	Timer/Counter0 Overflow	(TIMER0_OVF_vect)
18	SPI Serial Transfer Complete	(SPI_STC_vect)
19	USART Rx Complete	(USART_RX_vect)
20	USART, Data Register Empty	(USART_UDRE_vect)
21	USART, Tx Complete	(USART_TX_vect)
22	ADC Conversion Complete	(ADC_vect)
23	EEPROM Ready	(EE_READY_vect)
24	Analog Comparator	(ANALOG_COMP_vect)
25	2-wire Serial Interface (I2C)	(TWI_vect)
26	Store Program Memory Ready	(SPM_READY_vect)

- LOW to trigger the interrupt whenever the pin is low,
- CHANGE to trigger the interrupt whenever the pin changes value
- RISING to trigger when the pin goes from low to high,
- FALLING for when the pin goes from high to low.

Arduino Pin Mapping and registers

Arduino-Uno Pin mapping


Arduino function
reset (PCINT14/RESET) PC6 1
digital pin 0 (RX) (PCINT16/RXD) PD0 2
digital pin 1 (TX) (PCINT17/TXD) PD1 3
digital pin 2 (PCINT18/INT0) PD2 4
digital pin 3 (PWM) (PCINT19/OC2B/INT1) PD3 5
digital pin 4 (PCINT20/XCK/T0) PD4 6
VCC VCC 7
GND GND 8
crystal (PCINT6/XTAL1/TOSC1) PB6 9
crystal (PCINT7/XTAL2/TOSC2) PB7 10
digital pin 5 (PWM) (PCINT21/OC0B/T1) PD5 11
digital pin 6 (PWM) (PCINT22/OC0A/AIN0) PD6 12
digital pin 7 (PCINT23/AIN1) PD7 13
digital pin 8 (PCINT0/CLKO/ICP1) PB0 14
Arduino function
28 PC5 (ADC5/SCL/PCINT13) analog input 5
27 PC4 (ADC4/SDA/PCINT12) analog input 4
26 PC3 (ADC3/PCINT11) analog input 3
25 PC2 (ADC2/PCINT10) analog input 2
24 PC1 (ADC1/PCINT9) analog input 1
23 PC0 (ADC0/PCINT8) analog input 0
22 GND GND
21 AREF analog reference
20 AVCC VCC
19 PB5 (SCK/PCINT5) digital pin 13
18 PB4 (MISO/PCINT4) digital pin 12
17 PB3 (MOSI/OC2A/PCINT3) digital pin 11(PWM)
16 PB2 (SS/OC1B/PCINT2) digital pin 10 (PWM)
15 PB1 (OC1A/PCINT1) digital pin 9 (PWM)

Digital Pins 11,12 & 13 are used by the ICSP header for MISO, MOSI, SCK connections (Atmega168 pins 17,18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

Arduino-Uno Pin-Register mapping

PORT B ; pins 8 -- 13	PORT D ; pins 0 -- 7
	PORT C (A0 -- A5)

DDR -- pinMode (INPUT or OUTPUT)
PORT -- digitalRead/digitialWrite (LOW or HIGH)
PIN -- digitalRead of input pins

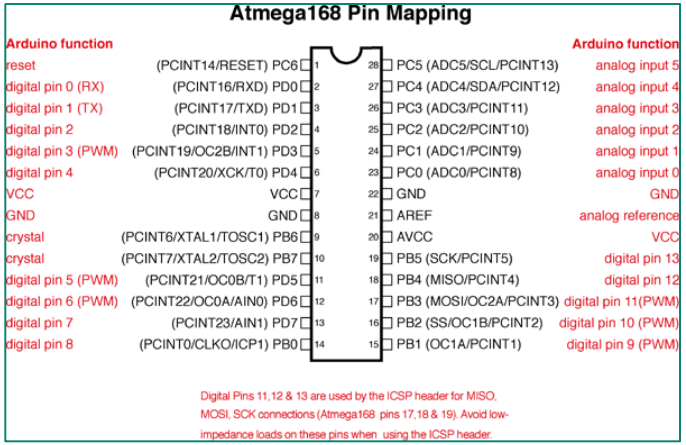
Arduino Interrupts

SREG -- Process status register -- BIT7 like: (SREG |= (ONE << 7); or (cli(); sei();)

PCICR : a register where the three least significant bits enable or disable pin change interrupts on a range of pins, i.e. {0,0,0,0,0,PCMSK2,PCMSK1,PCMSK0}

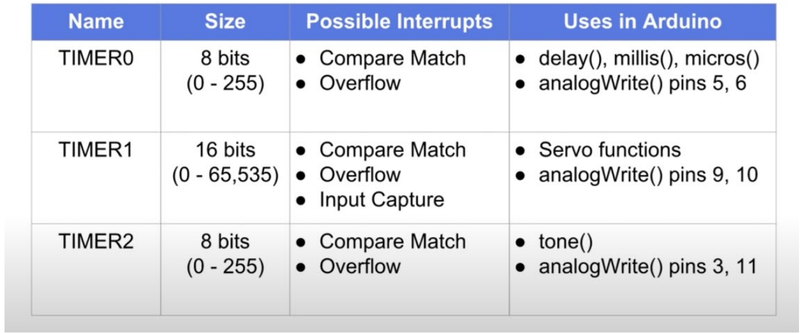
PCMSK0 = {PCINT7, PCINT6, PCINT5, PCINT4, PCINT3, PCINT2, PCINT1, PCINT0}

```
Step1: Turn interrupts on the port you want to use
PCICR |= 0b00000001;    // turn on port b
PCICR |= 0b00000010;    // turn on port c
PCICR |= 0b00000100;    // turn on port d
PCICR |= 0b00000111;    // turn on all ports

Step2: Enable the pin(s) you will use for interrupt
PCMSK0 |= 0b00000001;  // pin PB0, PCINT0, physical pin 14
PCMSK1 |= 0b00010000;  // pin PC4, which is PCINT12, physical pin 27
PCMSK2 |= 0b10000001;  // turn on pins PD0 & PD7, PCINT16 & PCINT23


Step3: Implement your interrupt function
ISR(PCINT0_vect){}    // Port B, PCINT0 - PCINT7
ISR(PCINT1_vect){}    // Port C, PCINT8 - PCINT14
ISR(PCINT2_vect){}    // Port D, PCINT16 - PCINT23

Step4: You might need to check which pin was switched on/off inside your ISR
Use PINB or PIND to read the pin value
```



DDR -- pinMode (INPUT or OUTPUT)
PORT -- digitalRead/digitialWrite (LOW or HIGH)
PIN -- digitalRead of input pins

Arduino Timers



Name	Size	Possible Interrupts	Uses in Arduino
TIMER0	8 bits (0 - 255)	• Compare Match • Overflow	• delay(), millis(), micros() • analogWrite() pins 5, 6
TIMER1	16 bits (0 - 65,535)	• Compare Match • Overflow • Input Capture	• Servo functions • analogWrite() pins 9, 10
TIMER2	8 bits (0 - 255)	• Compare Match • Overflow	• tone() • analogWrite() pins 3, 11

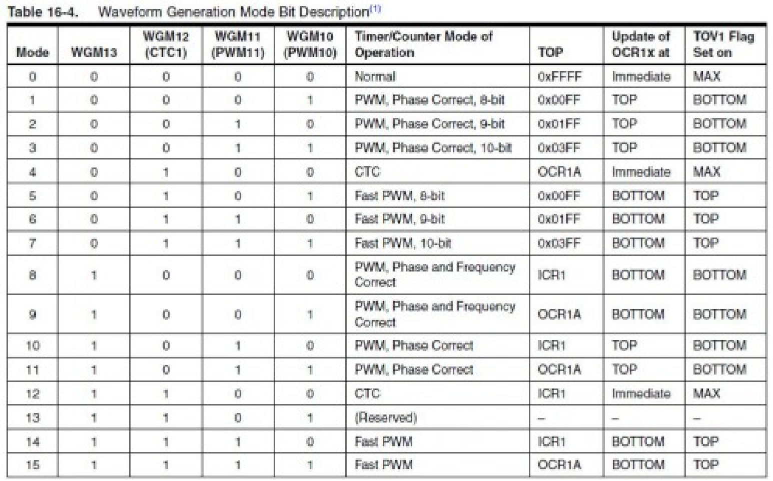


Table 16-4. Waveform Generation Mode Bit Description[1]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

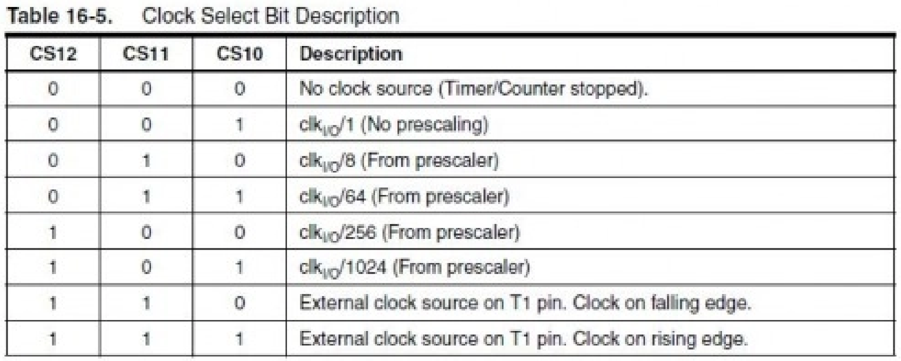


Table 16-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

TCCRx - Timer/Counter Control Register. The prescaler can be configured here

TCNTx - Timer/Counter Register. The actual timer value is stored here.

OCRx - Output Compare Register

ICRx - Input Capture Register (only for 16bit timer)

TIMSKx - Timer/Counter Interrupt Mask Register. To enable/disable timer interrupts.

TIFRx - Timer/Counter Interrupt Flag Register. Indicates a pending timer interrupt.

https://www.robotshop.com/community/forum/t/arduino-101-timers-and-interrupts/13072 and https://diyi0t.com/arduino-interrupts-and-timed-events/#:~:text=CTC%20timer%20interrupts%20are%20events,zero%20and%20restarts%20the%20counting.

Arduino Timers

```
void setup() {
  pinMode(ledPin, OUTPUT);
  noInterrupts();
  // Clear registers
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 0;

  //  we want turn led 13 to blink every 4 seconds = 0.25 Hz
  // 0.25Hz = (16000000/((62499+1)*1024))

  //Set timer compare
  OCR1A = 62499;

 // Prescaler 1024
  TCCR1B |= (1 << CS12) | (1 << CS10);

 // Output Compare Match A Interrupt Enable
  TIMSK1 |= (1 << OCIE1A);

  // CTC
  TCCR1B |= (1 << WGM12);



  interrupts();
}

void loop() {
}


ISR(TIMER1_COMPA_vect) {
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
}
```

```
Microcontroller clock frequency = 16,000,000
What is your timer frequency? How many ticks per second? 0.25? 0.5? 1? 2?
What should be the prescale? 1, 2, ...256, 65K --divider?
What should be the timer compare value?

         Your timer frequency = (16000000/((TCCRX)*prescale))
```

TCCRx - Timer/Counter Control Register. The prescaler can be configured here

TCNTx - Timer/Counter Register. The actual timer value is stored here.

OCRx - Output Compare Register

ICRx - Input Capture Register (only for 16bit timer)

TIMSKx - Timer/Counter Interrupt Mask Register. To enable/disable timer interrupts.

TIFRx - Timer/Counter Interrupt Flag Register. Indicates a pending timer interrupt.

Real- time System

A real time system is a system that **<u>must satisfy explicit (bounded) response-time constraints or risk severe consequences</u>**, including failure. A real time system is a system whose logical correctness is based on both the correctness of the outputs and their timeliness.

- Vending machine
- Gas pump
- Irrigation system controller
- Multicopter
- Mars Rover
- Automobile cruise control
- Alarm / security system
- Heating / air conditioning thermostat
- Microwave oven
- Anti-skid braking controller
- Traffic light controller

Real- time operating system - RTOS

A predictable and reliable operating system that shows consistent behavior under each and every possible load scenarios.

- Hard real time systems – These systems behave in a deterministic and time constrained manner
- Soft real time systems – These systems can be best described with our existing computer/laptop based operating systems.

Can you think of example hard/soft real-time systems?

There is many RTOS out there, https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

Choosing the right RTOS

When choosing RTOS, there are many factors involved, mainly:

- Memory footprints -- how much memory a program uses when running
- Interrupt latency -- the number of clock cycles* required for a processor to respond to an interrupt
- Microprocessor supported - ARM, MIPS, X86, RISC
- Power consumption - IoT device?
- Multicore support -- more in the coming weeks
- Virtualisation support -- can we run it in virtual environment
- Development tools & IDE

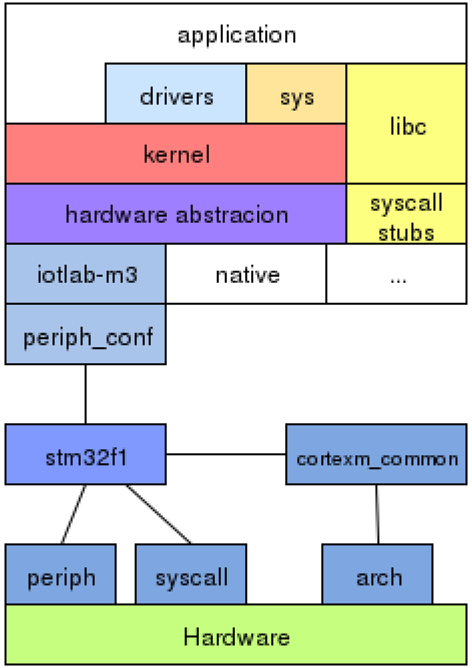
*** 4GHz processor performs 4*10^9 clock cycles per second**

RIOT OS

A real-time, microkernel-based, multi-threading operating system that explicitly considers devices with minimal resources typically found in the Internet of Things.

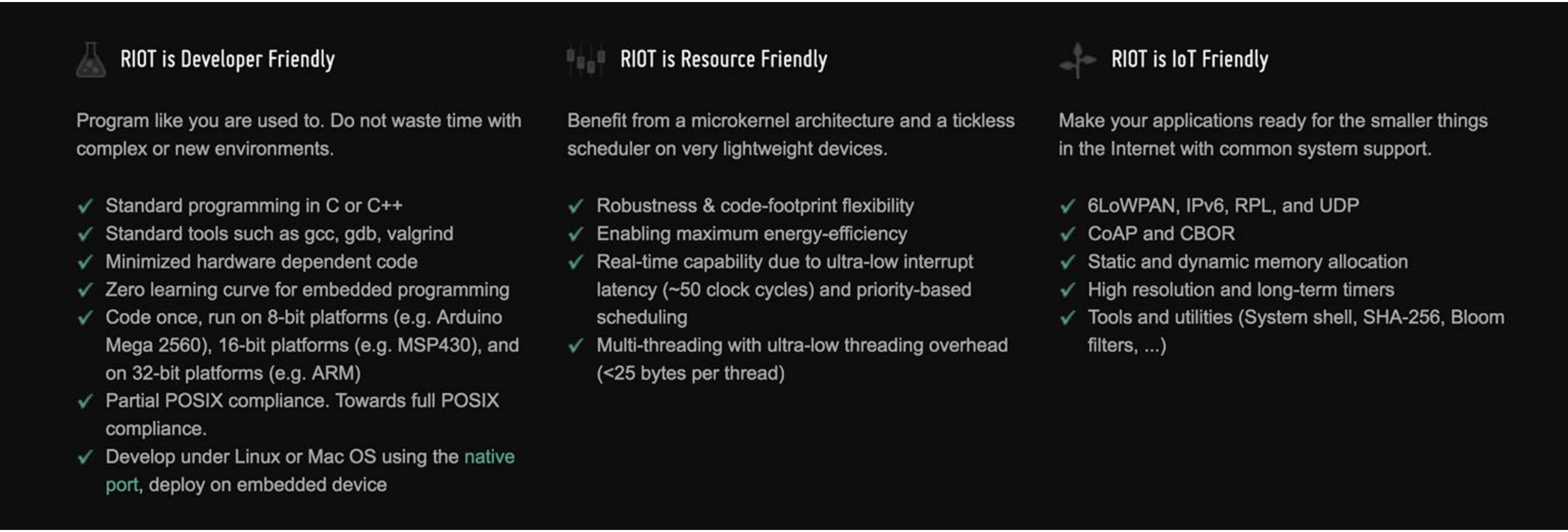
RIOT is based on design objectives including energy-efficiency, reliability, real-time capabilities, small memory footprint, modularity, and uniform API access, independent of the underlying hardware. Full IPv6 network protocol stack including the latest standards for connecting constrained systems to the Internet (6LoWPAN, IPv6, RPL, TCP and UDP).

RIOT can run IoT devices, testbed hardware (e.g. IoT-lab), and also as a process on your machine (native port).



https://www.silecs.net/wp-content/uploads/2018/04/2018-04FIT-IoT-Lab-Grid5000-school-Baccelli.pdf

RIOT Features



RIOT is Developer Friendly

Program like you are used to. Do not waste time with complex or new environments.

- ✓ Standard programming in C or C++
- ✓ Standard tools such as gcc, gdb, valgrind
- ✓ Minimized hardware dependent code
- ✓ Zero learning curve for embedded programming
- ✓ Code once, run on 8-bit platforms (e.g. Arduino Mega 2560), 16-bit platforms (e.g. MSP430), and on 32-bit platforms (e.g. ARM)
- ✓ Partial POSIX compliance. Towards full POSIX compliance.
- ✓ Develop under Linux or Mac OS using the native port, deploy on embedded device

RIOT is Resource Friendly

Benefit from a microkernel architecture and a tickless scheduler on very lightweight devices.

- ✓ Robustness & code-footprint flexibility
- ✓ Enabling maximum energy-efficiency
- ✓ Real-time capability due to ultra-low interrupt latency (~50 clock cycles) and priority-based scheduling
- ✓ Multi-threading with ultra-low threading overhead (<25 bytes per thread)

RIOT is IoT Friendly

Make your applications ready for the smaller things in the Internet with common system support.

- ✓ 6LoWPAN, IPv6, RPL, and UDP
- ✓ CoAP and CBOR
- ✓ Static and dynamic memory allocation
- ✓ High resolution and long-term timers
- ✓ Tools and utilities (System shell, SHA-256, Bloom filters, ...)

https://www.youtube.com/watch?v=TOdPmiU-cXA

Create an application in RIOT OS

Tutorials:

https://materials.dagstuhl.de/files/16/16353/16353.OliverHahm.Slides1.pdf

http://doc.riot-os.org/creating-an-application.html

RIOT Task, No Arduino Board?

If you do not have arduino board, you will need to compile your code in RIOT, then get your **ELF** file and upload it to an IoT cloud testbed. I have two options:

- https://www.iot-lab.info/testbed/dashboard

It is free and simple to use. However, it only has Arduino-Zero or M3 boards firmware.

- Send me your **source code** and **makefile** to deploy it on **my Arduino-Uno board to** test it for you.