Università
della
Svizzera
italiana

Facoltà
di
scienze
informatiche

Edge Computing in the IoT

Embedded Operating Systems

Alberto Ferrante (alberto.ferrante@usi.ch)

Software on Bare Metal

- Bare metal refers to a computer executing instructions directly on logic hardware without an intervening operating system
- No resources taken by the operating system
- Better suited for systems with very limited resources
- Better suited for simple applications

Software on Bare Metal

- The absence of an operating system leaves everything in the hands of the programmer:
  - Memory management
  - Device management
  - Scheduling of resources
  - …

Software on Bare Metal

- Suited for single (or limited) tasks that are part of the same program and run one at a time
  - It gets difficult when we need to manage complex devices that are asynchronous with the application, such as network devices
- System libraries are used to help managing the system and the devices
  - Contain all the low-level instructions required for accessing certain (family of) device(s) or file systems
    - e.g., a library for managing a specific display will provide to the programmer functions for visualizing contents on the display. These functions will contain the low level instructions required
  - A system library may provide help in managing the memory and the system, by defining some system-specific variables

Software on Bare Metal – An Example

- On “classic” Arduino boards (e.g., Arduino Uno), software runs on bare metal
- The Arduino library (Arduino.h) provides basic functions defines system-wide constants
- Other libraries are provided to help managing specific devices
  - e.g., the LiquidCrystal library provides the functions for managing an LCD
- Other devices must be managed manually

```
/*********** mid level commands, for sending ata/cmds */

inline void LiquidCrystal::command(uint8_t value) {
  send(value, LOW);
}

inline size_t LiquidCrystal::write(uint8_t value) {
  send(value, HIGH);
  return 1; // assume sucess
}

/************ low level data pushing commands **********/

// write either command or data, with automatic 4/8-bit
selection
void LiquidCrystal::send(uint8_t value, uint8_t mode) {
  digitalWrite(_rs_pin, mode);

  // if there is a RW pin indicated, set it low to Write
  if (_rw_pin != 255) {
    digitalWrite(_rw_pin, LOW);
  }

  if (_displayfunction & LCD_8BITMODE) {
    write8bits(value);
  } else {
    write4bits(value>>4);
    write4bits(value);
  }
}
```

```
lcd.begin(16, 2);

// Print a message to the LCD.

lcd.print("hello, world!");
```

Asynchronous I/O

- E.g., An mp3 player plays music at a predefined rate; buttons pressed by the user are asynchronous with music playing
  - The buttons will be depressed at a much lower rate than the processing rate of audio and asynchronous with that
  - The buttons will not be synchronous with system clock
- Keeping up with the input and output data while checking on the button can introduce some very complex control code into the program
  - Sampling the button's state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently can cause the machine to incorrectly play music
  - One solution is to introduce a counter into the main music playing, so that a subroutine to check the input button is called once every *n* times the loop is executed
    - This solution does not work when either the music playing loop or the button-handling routine has highly variable execution times. If the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost
  - We need to be able to keep track of these two different tasks separately, applying different timing requirements to each

Asynchronous I/O

- **When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution becomes very complex very quickly**
  - Such complex control is usually quite difficult to verify for either functional or timing properties

Application flow control - Polling

- Polling, active wait, busy-waiting, busy-looping, or spinning
- The application keeps querying the device / checking a condition
- System resources are dedicated to checking a condition continuously and the system stays busy
  - It works well for devices that need a fast response and are often ready
  - Not efficient in all the other cases

Polling

```
void loop()
{
  if (digitalRead(pinToPoll)) {
    // react
  }
}
```

Application flow control - Interrupts

- A signal to the processor indicating an event that needs immediate attention
  - Hardware interrupts
    - **Asynchronous** with the clock
    - Used by devices
  - Software interrupts
    - **Synchronous** with the clock
    - Caused either by an exceptional condition in the processor (trap, exception) itself, or a special instruction in the instruction set
      - e.g., a divide by zero
      - e.g., a page fault

Interrupts

- Timeline of a HW interrupt
  - Interrupt request (IRQ) by the device
  - The CPU receives the IRQ and suspends its current activities, saving its state
    - Asynchronous: it might arrive when the processor is executing an instruction
  - The CPU executes a function called *interrupt handler* (or an *interrupt service routine, ISR*)
  - After the interrupt handler finishes, the processor resumes normal activities

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin,
INPUT_PULLUP);

attachInterrupt(digitalPinToInterru
pt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

Interrupt Request

- CPUs have dedicated IRQ lines
  - Number is limited (e.g., 4)
  - Multiple devices may be connected to the same IRQ
    - After the interrupt is received, devices should be queried to discover which one made the request
    - External Interrupt management devices may provide multiplexing to allow more IRQ signals in a clean way
  - IRQ lines may have different priorities
- *Maskable interrupts*: hardware interrupts that may be ignored by setting a bit in an interrupt mask register's (IMR)
- *Non-maskable interrupts* (NMI): hardware interrupts that can never be ignored

Multiple Interrupt requests

- What happens if an interrupt request is received while serving a preceding request?
  - E.g. An interrupt from device A is being served while device B makes another interrupt request
- First method: disable interrupts while interrupt service routines are being executed
  - Interrupts are enabled again after the completion of the ISR and other outstanding requests are checked and served
  - E.g., while interrupt from device A is being served, the interrupt from device B is ignored
  - Enforces sequential execution of ISRs and, thus, disregards interrupt priorities
- Second method: higher-priority interrupts are allowed to interrupt lower-priority ones
  - E.g., while the interrupt from device A is being served, the interrupt from device B (at higher priority) is allowed. Thus, the second interrupt is served; then, the execution of the first ISR is resumed and completed

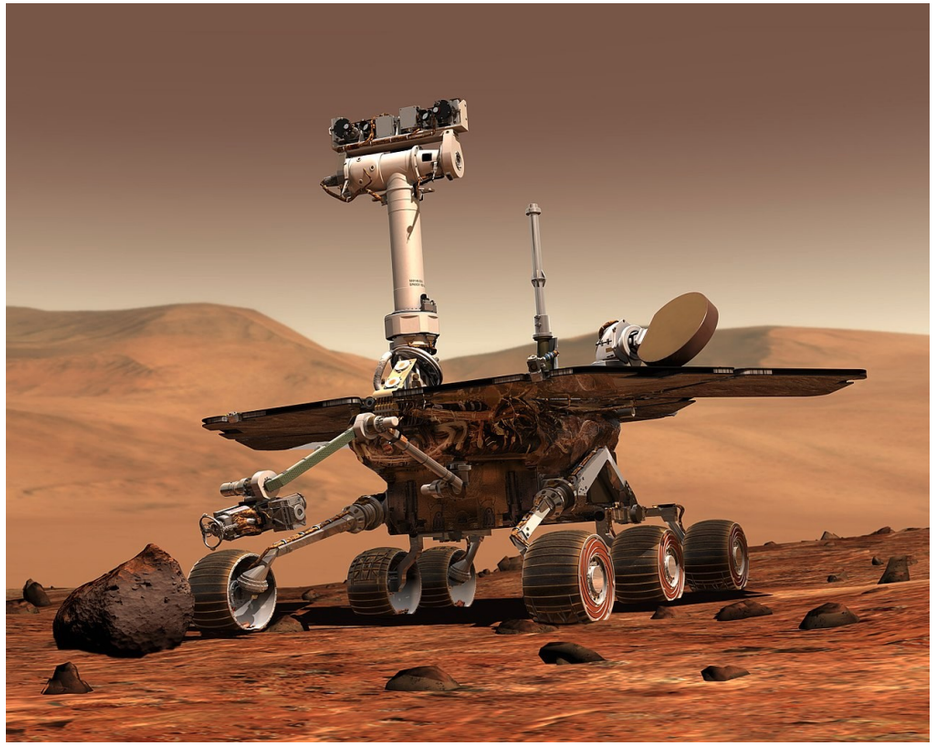
The Watchdog Timer

- An electronic timer that is used to detect and recover from computer malfunctions
  - E.g., crashes of the operating system or hangs in the application
- Commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner

https://en.wikipedia.org/wiki/Watchdog_timer

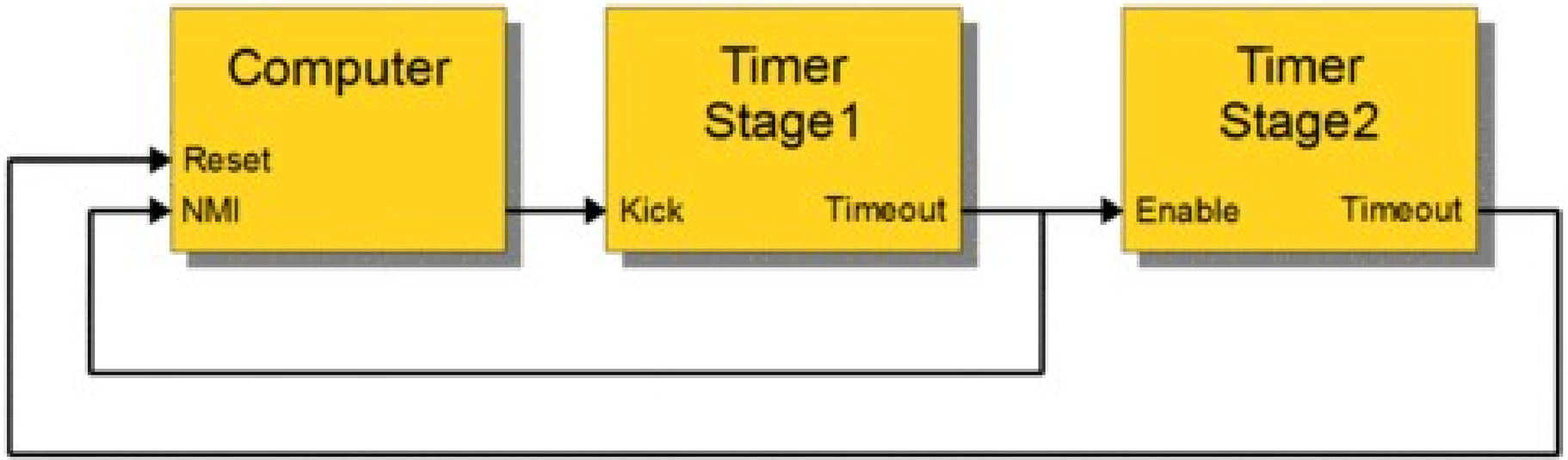
The Watchdog Timer

- During normal operation, the application regularly resets the watchdog timer to prevent it from timing out
  - In computers that are running operating systems, watchdog resets are usually invoked through a device driver
  - E.g., in Linux, a user space program kicks the watchdog by interacting with the watchdog device driver, by writing a zero character to /dev/watchdog
- If the timer expires, it will generate a timeout signal:
  - maskable interrupt
  - non-maskable interrupt
  - processor reset / power cycling
  - fail-safe state activation
  - ...



https://en.wikipedia.org/wiki/Watchdog_timer

Two-stage Watchdog



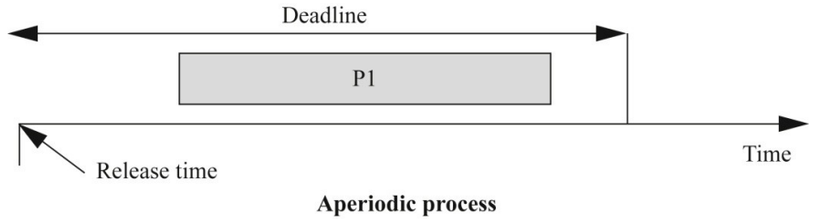
Embedded Operating Systems

- Many applications are complex enough that cannot be handled by one program
  - Variable data rate
    - The need to receive or send data at different rates
  - Asynchronous I/O
    - The system needs to perform some actions at a certain rate while handling I/O that is asynchronous
  - Different tasks: a set of programs that may communicate each other
    - Tasks are implemented by means of processes or threads

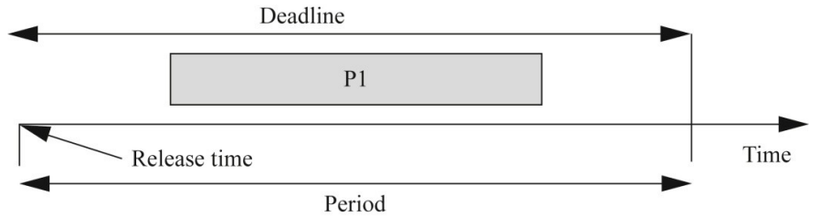
Embedded/IoT operating systems

- Unlike general purpose systems, embedded OSes are highly heterogeneous
  - From full fledged OSes requiring GB of RAM to lightweight OSes requiring a few kB of RAM
  - Very different user interfaces, depending on the application
  - May or may not support real time
  - May or may not support virtual memory
- In most of the cases, embedded OSes are
  - Simpler and relatively lightweight
  - Low-power/energy oriented

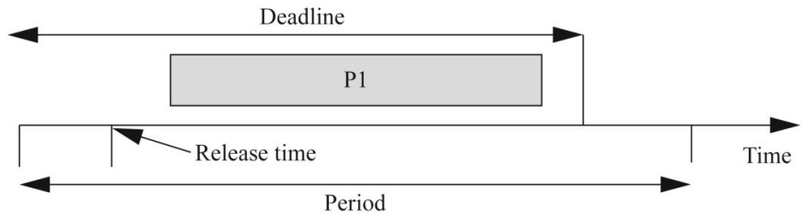
Timing Requirements on Processes



Aperiodic process



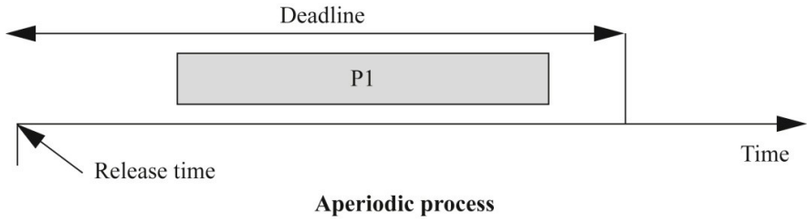
Periodic process initiated at start of period



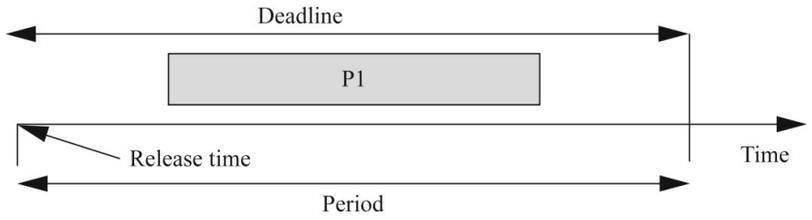
Periodic process released by event

- **Initiation time** (or **release time**): the time at which the process goes from the *waiting* to the *ready* state
  - The process will start execution at a certain time after the release time
  - An aperiodic process is by definition initiated by an event
  - For periodic processes
    - At the beginning of the period
    - At the arrival time of certain data after the start of the period
- **Period of a process**: the time between successive executions
  - e.g., the period of a digital filter is defined by the time interval between successive input samples
  - **Process rate**: inverse of the period

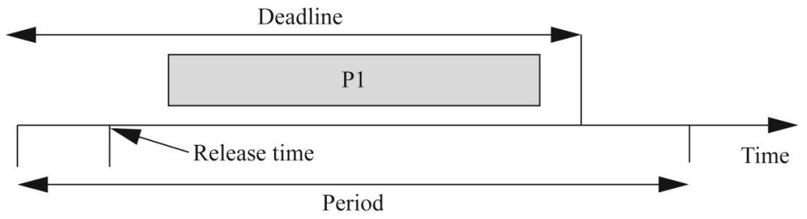
Timing Requirements on Processes



Aperiodic process



Periodic process initiated at start of period



Periodic process released by event

- A **deadline** specifies when a computation must be finished
  - For an aperiodic process, it is generally measured from the initiation time
  - For a periodic process
    - May occur at some time other than the end of the period
    - Some scheduling policies make the simplifying assumption that the deadline occurs at the end of the period

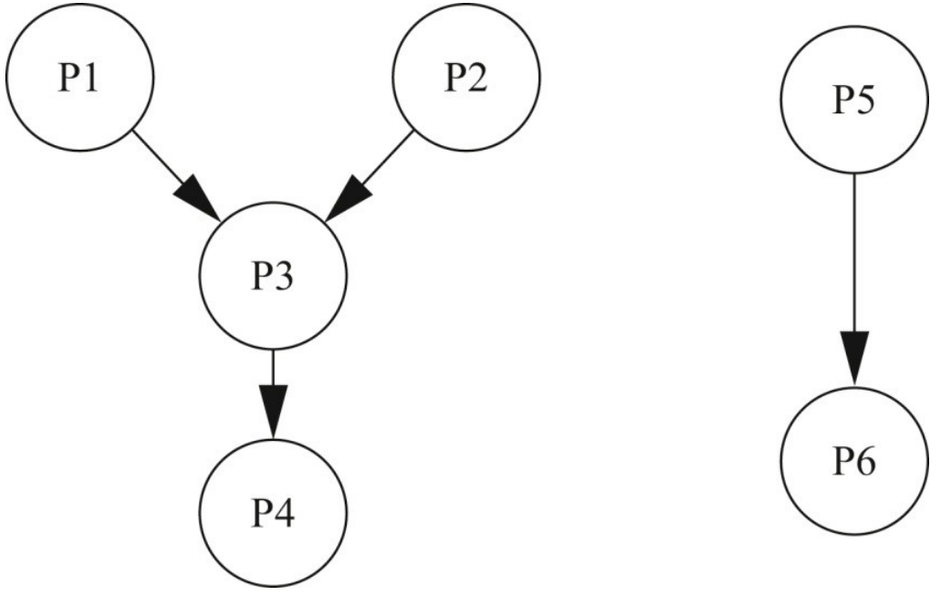
Timing Requirements on Processes

- The most common case for periodic processes is for the initiation interval to be equal to the period
- **Hyperperiod**: the least common multiple (LCM) of the periods of multiple processes
- **Response time**: the time at which the process finishes
  - If the schedule meets the requirements, the response time will be before the end of the process's period
  - The response time depends only in part on computation time of the process and how long it takes to execute
    - In a multitasking system, the process may be interrupted to let other processes run

Timing Requirements on Processes

- **Jitter**: the allowable variation in the completion of the task
  - Can be important in a variety of applications: in the playback of multimedia data to avoid audio gaps or jerky images; in the control of machines to ensure that the control signal is applied at the right time
- When a process misses a deadline the effects depend on the application
  - May be catastrophic
  - The system can be designed to take a variety of actions when a deadline is missed
    - Safety-critical systems may try to take compensatory measures such as approximating data or switching into a special safety mode
    - Systems for which safety is not as important may take simple measures to avoid propagating bad data, such as inserting silence in a phone line, or may completely ignore the failure

Timing Requirements on Processes



- Timing of processes may be constrained when the processes pass data among each other
  - Before a process can become ready, all the processes on which it depends must complete and send their data to it.
  - The data dependencies define a partial ordering on process execution
  - All processes must finish before the end of the period
  - The data dependencies must form a directed acyclic graph (DAG)
    - Cycle in the data dependencies is difficult to interpret in a periodically executed system
  - Communication among processes that run at different rates cannot be represented by data dependencies
    - Nevertheless, communication among processes of different rates is very common

What's *real time*?

Real Time

- Quite often, embedded applications require *real time processing*
  - There is an upper bound to time allowed for execution of the application or – more in general – of specific segments of the application
  - It does not necessarily means “quickly”, just that time is bounded

Real Time

- Real time may involve
  - **Throughput** (i.e., frequency at which the computation must be performed on a continued sequence of data samples)
    - Movie player: latency may be loosely defined, but the processor needs to accept/process/provide frames at strictly defined frequency – constraint is on throughput
    - ➔ Constraints are set on task periodicity
  - **Latency** (i.e., time elapsing between data input and results availability)
    - Emergency button: constraints refer to latency
    - ➔ Constraints are set on the response time to an event

Hard Real Time

- ***Hard*** real-time: an upper bound on time parameter **must be *always* respected**
- Required to complete a critical task within a guaranteed amount of time

Abraham Silbertchatz, *Operating Systems Concepts*

Hard Real Time

- Generally, a process is submitted along with a statement of the amount of time within which it needs to complete or perform I/O
- The scheduler either admits the process, guaranteeing that it will complete on time, or rejects the request as impossible
  - Guarantee made under resource reservation:
    - The scheduler needs to know how long it takes to perform each type of operating-system function (maximum amount of time for each operation): impossible in a system with secondary storage or virtual memory
    - Real time systems are often composed of special-purpose software running on hardware dedicated to their critical process and lack the full functionality of modern computers and operating systems

Abraham Silbertchatz, *Operating Systems Concepts*

Soft Real Time

- ***Soft*** real-time: time constraint is less strict and some violations may be accepted, provided they are “few”
- Soft real-time computing is less restrictive: it requires that critical processes receive priority over less fortunate ones
- Adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and may result in longer delays or even starvation for some processes

Abraham Silbertchatz, *Operating Systems Concepts*

Real Time Operating Systems (RTOS)

- Operating systems that support real-time processing
- Either:
  - Event-driven: switch tasks only when an event of higher priority needs servicing; called preemptive priority, or priority scheduling
  - Time-sharing: switch tasks on a regular clocked interrupt, and on events
- Scheduling of tasks is performed by means of specific scheduling algorithms and priority settings
- The most reliable way to meet timing requirements accurately is to use
  - Preemption to allow the OS switching among different processes
  - Priorities to control what process runs at any given time

Scheduling

- Scheduling is a fundamental operating-system function
  - Almost all computer resources are scheduled
  - CPU is certainly the scarcest resource
- In an uniprocessor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled

Abraham Silbertchatz, *Operating Systems Concepts*

Scheduling

- The objective of multiprogramming is to have one of the processes running at all times to maximize CPU utilization
  - Several processes are kept in memory
  - When one process has to wait for resources, the operating system takes the CPU away from that process and gives the CPU to another process
- The *dispatcher* is the module that gives control of the CPU to the process selected by the short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

Abraham Silbertchatz, *Operating Systems Concepts*

CPU Scheduling

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed
  - The selection process is carried out by the *short-term scheduler* (or *CPU scheduler*)
- The scheduler selects among the processes in memory that are ready to execute and allocates the CPU to one of them
  - Processes ready to execute: those processes that have all the required resources allocated, they just miss the CPU
  - Queue of ready processes: not necessarily a FIFO
    - It depends on the scheduling algorithm

Abraham Silbertchatz, *Operating Systems Concepts*

CPU Scheduling

- **Nonpreemptive scheduling**: scheduling is performed only when a running process terminates or it goes from the running to the waiting state
  - Once the CPU has been allocated to a process, the process keeps the CPU until it terminates or it switches to the waiting state
- **Preemptive scheduling**: executing processes can be preempted; scheduling is performed
  - When a process goes from a running to a ready state (e.g., when an interrupt occurs)
  - When a process switches from waiting to ready state

Abraham Silbertchatz, *Operating Systems Concepts*

Scheduling Criteria

- Many criteria can be used to compare different scheduling algorithms and to decide which algorithm to use
- Criteria
  - CPU utilization: keep the CPU as busy as possible
  - Throughput: maximize the number of processes completed per time unit
  - Turnaround time: minimize the time from submission of a process and its completion
  - Waiting time: minimize the time spent by processes waiting in the ready queue
  - Response time: minimize the time from submission of a request to the process and the production of the first result
- In general we tend to optimize the average value of the selected criterion
  - In some cases we might need to optimize maximum and/or minimum or variance

Abraham Silbertchatz, *Operating Systems Concepts*

First-Come, First-Served Scheduling

- The process that requests the CPU first is allocated the CPU first
- The implementation of the FCFS policy is easily managed with a FIFO queue:
  - When a process enters the ready queue, it is placed at the tail of the queue
  - When the CPU is free, it is allocated to the process at the head of the queue
    - The process is removed from the ready queue

Abraham Silbertchatz, *Operating Systems Concepts*

First-Come, First-Served Scheduling

- The average waiting is in general not minimal and may vary substantially if the process CPU-burst times vary greatly
  - The following set of processes arrive at time 0, with the length of the CPU-burst time given in ms: P1 (100), P2 (20), P3 (10)
    - The processes arrive in the order P1, P2, P3 → the average waiting time is (0+100+20)/3=40
    - The processes arrive in the order P2, P3, P1 → the average waiting time is (0+20+10)/3=10
- It may result in lower CPU and I/O utilization if processes are not homogeneous
  - e.g., one process CPU bound and multiple processes I/O bound

Abraham Silbertchatz, *Operating Systems Concepts*

Shortest-Job-First Scheduling

- Each process is associated to the length of its next CPU burst
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used
- Scheduling does not depend on process total length in time, just on the next burst
- Priority algorithm where the priority is the inverse of the next CPU burst
- The SJF scheduling algorithm is provably optimal: it gives the minimum average waiting time for a given set of processes
  - Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process

Abraham Silbertchatz, *Operating Systems Concepts*

Shortest-Job-First Scheduling

- The real difficulty with the SJF algorithm is knowing the length of the next CPU request
  - Impossible to implement in an exact way as short term scheduler
  - We may predict the length of the next burst based on the previous one(s)
- The SJF algorithm may be either preemptive or nonpreemptive
  - The choice arises when a new process arrives at the ready queue while a previous process is running
    - The next CPU burst of the new arriving process may be shorter than what is left of the currently executing process
  - A preemptive SJF algorithm will preempt the currently executing process
  - Preemptive SJF scheduling is also called **shortest-remaining-time-first scheduling**

Abraham Silbertchatz, *Operating Systems Concepts*

Round Robin

- A small unit of time, called a *time quantum* or *time slice*, is defined
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum
  - The ready queue is treated as a circular queue
  - No process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process)
  - If a process's CPU burst exceeds 1 time quantum, that process is preempted and put back in the ready queue
  - If there are *n* processes in the ready queue and the time quantum is *q*
    - Each process gets *1/n* of the CPU time in chunks of at most *q* time units
    - Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum

Abraham Silbertchatz, *Operating Systems Concepts*

Round Robin

- Performance of the RR algorithm depends heavily on the size of the time quantum
  - If the time quantum is extremely large, the RR policy becomes similar to the FCFS policy
  - If the time quantum is extremely small the RR approach is called processor sharing
    - It creates the appearance that each of the n processes has its own processor running at 1/n the speed of the real processor
  - We need also to consider the effect of context switching on the performance of RR scheduling
    - We want the time quantum to be large with respect to the context switching time

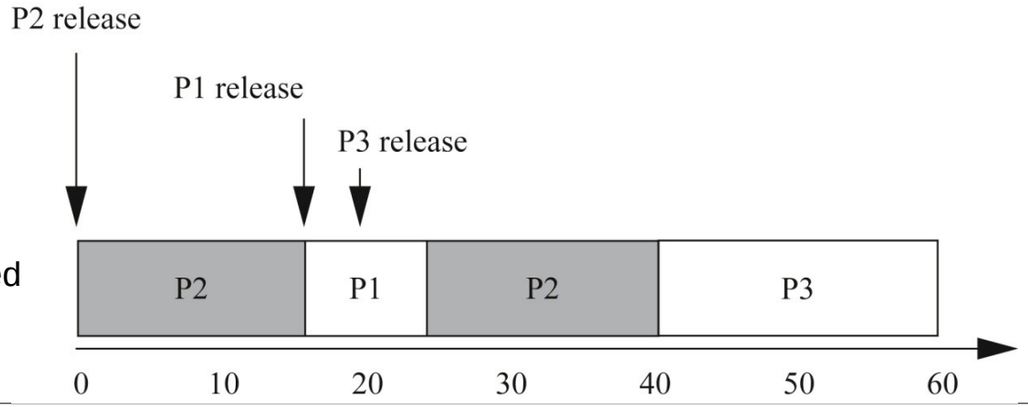
Abraham Silbertchatz, *Operating Systems Concepts*

Priority Scheduling

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority
  - Equal-priority processes are scheduled in FCFS order
- Priority scheduling may be either preemptive or nonpreemptive
  - The choice arises when a new process arrives at the ready queue while a previous process is still running
  - A preemptive priority scheduler will preempt the currently executing process if the new arriving process has an higher priority

Process	Priority	Execution time
P1	1	10
P2	2	30
P3	3	20

- P2 is ready to run when the system is started
- P1 becomes ready at time 15
- P3 becomes ready at time 18



Abraham Silbertchatz, *Operating Systems Concepts*
Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design

Priority Scheduling

- A major problem with priority scheduling is indefinite blocking, or starvation
  - In heavily loaded computer systems, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU
    - A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- A solution to the problem of indefinite blockage of low-priority processes is **aging**
  - Gradually increasing the priority of processes that wait in the system for a long time

Abraham Silbertchatz, *Operating Systems Concepts*

Rate Monotonic Scheduling (RMS)

- One of the first scheduling policies developed for real-time systems and is still widely used
- The theory underlying RMS is known as rate-monotonic analysis (RMA), that uses the following model of the system
  - All processes run periodically on a single CPU
  - Context switching time is ignored
  - There are no data dependencies between processes
  - The execution time for a process is constant
  - All deadlines are at the end of the periods
  - The highest-priority ready process is always selected for execution
    - **With preemption**

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/rate-monotonic-scheduling

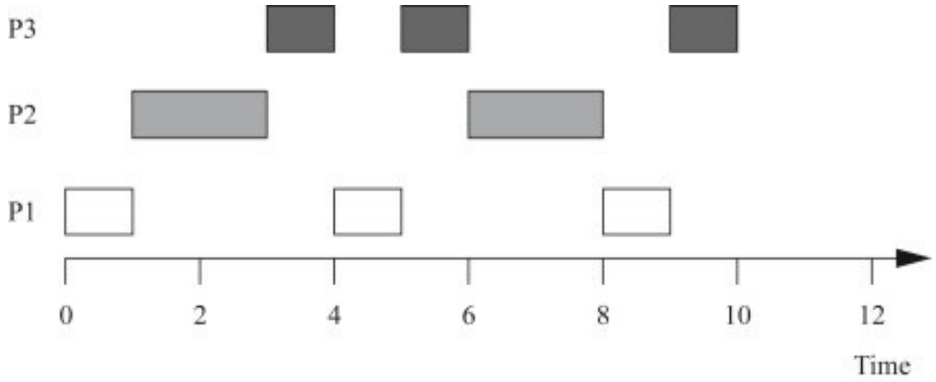
Rate Monotonic Scheduling (RMS)

- It is a static scheduling policy
  - It assigns fixed priorities to processes
  - The process with the shortest period is assigned the highest priority
    - The optimum assignment of **static priorities** to processes
- It does not provide 100% CPU utilization

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/rate-monotonic-scheduling

Rate Monotonic Scheduling - Example

Process	Exec. time	Period	Priority
P1	1	4	1
P2	2	6	2
P3	3	12	3



- To understand all the interactions between the periods, we need to construct a timeline equal in length to the least-common multiple of the process periods (**hyperperiod**)
  - 12 in this case

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/rate-monotonic-scheduling

Rate Monotonic Scheduling - Example

Process	Exec. time	Period	Priority
P1	2	4	1
P2	3	6	2
P3	3	12	3

- There is no feasible assignment of priorities that guarantees scheduling
  - During one hyperperiod (12 time-unit interval) we would be supposed to execute
    - P1 three times → 6 units of CPU time
    - P2 twice → 6 units of CPU time
    - P3 one time → 3 units of CPU time
    - The total of 6 + 6 + 3 = 15 units of CPU time is more than the 12 time units available
  - Even though each process alone has an execution time significantly lower than its period, combinations of processes require more than 100% of the available CPU cycles

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/rate-monotonic-scheduling

Rate Monotonic Scheduling – Feasibility of Scheduling

- RMS always provides a feasible schedule if such a schedule exists
- **CPU utilization: Sum, for each process, of execution_time/period**
- When no. of processes ≥ 10, a scheduling is feasible if total CPU utilization is less than 70%
  - CPU utilization tends to *ln 2=0.69* when the number of processes tends to infinity
- For smaller number of processes or when CPU utilization is close to 70%, the actual schedule should be considered

Earliest Deadline First (EDF)

- Dynamic priority scheme
  - Process priorities are changed during execution
- Priorities
  - Are assigned in order of deadline
    - The highest-priority process is the one whose deadline is nearest in time, and the lowest-priority process is the one whose deadline is farthest away
  - Are updated at every time quantum
  - The highest-priority ready process is run
- With preemption

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/earliest-deadline-first

Earliest Deadline First (EDF)

- EDF can achieve 100% CPU utilization
- **A feasible schedule exists if CPU utilization ≤ 1**
- The implementation of EDF is more complex than RMS
  - The major problem is keeping the processes sorted by time to deadline
    - Because the times to deadlines for the processes change during execution, we cannot presort the processes into an array, as we could for RMS

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/earliest-deadline-first

EDF Example

Process	Exec. time	Period
P1	1	3
P2	1	4
P3	2	5

- The least-common multiple of the periods is 60
- The utilization is 1/3 + 1/4 + 2/5 = 0.9833333
  - Scheduling is feasible!
- One time slot is left at the end of this unrolled schedule, which is consistent with our calculation of the CPU utilization

Time	Running process	P1 next deadline	P2 next deadline	P3 next deadline
		3	4	5
0	P1	6	4	5
1	P2	6	8	5
2	P3	6	8	5
3	P3	6	8	10
4	P1	9	8	10
5	P2	9	12	10
6	P1	12	12	10
7	P3	12	12	10
8	P3	12	12	15
9	P1	15	12	15
10	P2	15	16	15
11	P3	15	16	15
12	P3	15	16	20
13	P1	18	16	20
14	P2	18	20	20
15	P1	21	20	20

Marilyn Wolf, Computers as Components - Principles of Embedded Computing System Design
https://www.sciencedirect.com/topics/computer-science/earliest-deadline-first

Real Time Scheduling

- Hard Real Time
  - Often managed with exclusive use of hardware (e.g., dedicated microcontroller) or at least overprovisioning and hardware resource reservation
  - Periodic tasks: EDF or RMS
  - Response to events:
    - Assign high priority to connected interrupts
    - Use a preemptive OS
    - Check requirements against system resources
    - No algorithms to guarantee that the deadline is respected: we are dealing with asynchronous events
- Soft Real Time
  - Allocation of priorities to processes and resource overprovisioning
  - Response to events:
    - Assign high priority to connected interrupts
    - Use a preemptive OS
    - Check requirements against system resources

Scheduling for low power

- Considering CPU utilization in RMS and EDF, we can decide to lower the CPU clock period
  - Decreases power
  - Respects deadlines

Shared Resources

- A ***race condition*** or *race hazard* is the condition where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events
- Example: consider a device with a flag in memory that must be tested and modified by a process
  - Case 1:
    - Task 1 reads 0 in the flag memory location
    - Task 1 sets the flag to 1 and starts using the device writing to the device data register
    - Task 2 reads 1 in the flag memory location
    - Task 2 cannot use the device until Task 1 releases it (sets flag to 0)
  - Case 2
    - Task 1 reads 0 in the flag memory location
    - Task 2 reads 0 in the flag memory location
    - Task 1 sets the flag to 1 and writes data to the device’s data register
    - Task 2 also sets the flag to 1 and writes its own data to the device data register

Shared Resources – Critical Section

- We need to control the order in which some operations occur
  - e.g., we need to be sure that a task finishes an I/O operation before allowing another task to starts its own operation on the same I/O device
- Sensitive sections of code are enclosed in a *critical section*
  - A section of code that executes without interruption

Shared Resources – Semaphores

- Semaphores are primitives provided by the OS
- Used to guard resources
- Mutex/Binary semaphores
  - Critical sections start by calling a semaphore function that does not return until the resource is available
  - When we are done with the resource we use another semaphore function to release it

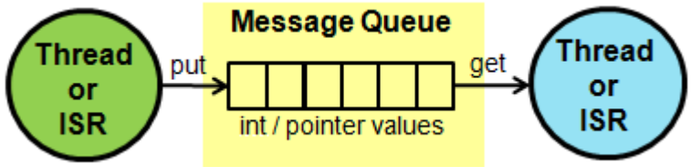
```
/* some nonprotected operations here */
P(); /* wait for semaphore */
/* do protected work here */
V(); /* release semaphore */
```

Shared Resources – Semaphores

- To implement P() and V() , the microprocessor bus must support atomic read/write operations
- These types of instructions first read a memory location and then set it to a specified value, returning the result of the test
  - If the location was already set, then the additional set has no effect but the instruction returns a false result
  - If the location was not set, the instruction returns true and the location is in fact set
- Non-binary semaphores can be used to manage pool of resources
  - A counter is used to count the number of free resources in the pool

Communication Among Processes - Queue

- A data structure used for communication between processes
- Producer threads insert elements in the queue
- Consumer threads consume elements



https://os.mbed.com/docs/mbed-os/v6.3/apis/queue.html

Shared Resources and RTOSes

- Processes may compete not only for the CPU, but also for I/O devices
- Race conditions are managed in the classic way:
  - Critical sections implemented with semaphores
- Critical sections are problematic in real-time systems
  - The interrupt system is shut off during the critical section
    - The timer cannot interrupt and other processes cannot start to execute
  - The kernel may also have its own critical sections that keep interrupts from being serviced and other processes from executing

Priority Inversion and RTOSes

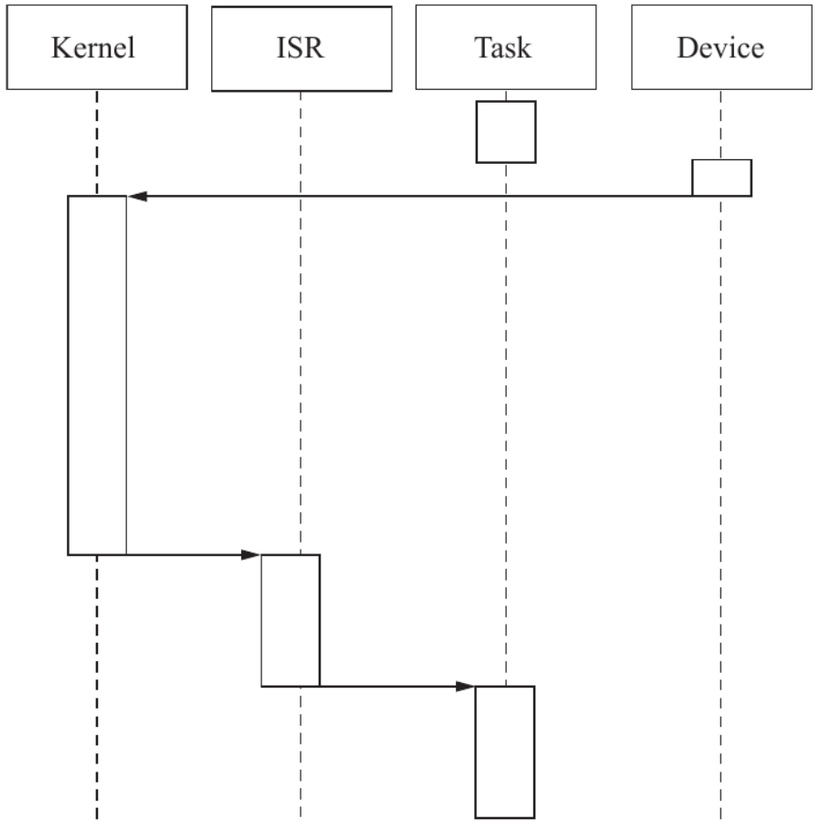
- **Priority inversion**: a low priority process blocks the execution of a higher-priority process by keeping hold of a resource
  - e.g., three processes
    - P1 (highest priority), P2 (mid priority), and P3 (lowest priority)
    - P1 and P3 share the same resource
    - P3 becomes ready and enters its critical section, acquiring the shared resource
    - P2 becomes ready and preempts P3
    - P1 becomes ready: it preempts P2, but as soon as it reaches its critical section, it stops as it cannot acquire the shared resource (held by P3)
    - P2 is allowed to finish
    - P3 will finish and release the shared resource
    - P1 can finally access the critical section and complete

Priority Inversion and RTOSes

- **Priority inheritance**: the priority of a process holding a resource is increased so that it inherits the highest priority among all the processes that might access the resource
  - When the resource is released, the process is demoted to its normal priority
  - In our previous example, while holding the shared resource, P3 will inherit the priority of P1
    - When P2 arrives, P3 is allowed to complete, having a higher priority
    - P1 is not forced to wait for P2 and then P3 to complete

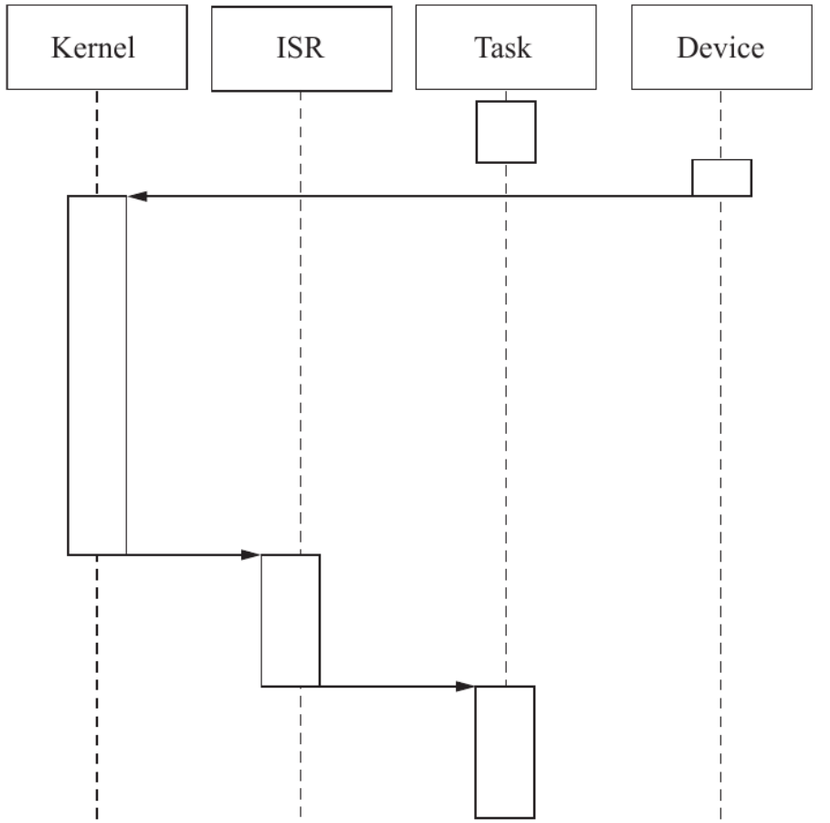
Interrupts and RTOSes

- The latency from when an interrupt is requested to when the device's service is complete is a critical parameter of real-time performance
- Interrupt latency: the duration of time from the assertion of a device interrupt to the completion of the requested operation
  - Critical because data may be lost when an interrupt is not serviced in a timely fashion



Interrupts and RTOSes

- Interrupt latency is influenced by
  - The processor interrupt latency
  - The execution time of the interrupt handler
  - Delays due to RTOS scheduling



Interrupts and RTOS Scheduling

- Critical sections in the kernel will prevent the RTOS from taking interrupts
  - Very long critical sections are a problem
- A lower-priority interrupt may delay a higher-priority one
  - Priorities are determined by the hardware not by the RTOS
- Mitigation: divide the interrupt handler into
  - The Interrupt Service Handler (system mode)
    - Must be as short as possible
  - The Interrupt Service Routine (user-mode thread)
    - Called by the handler, it follows the priorities decided by the OS

Embedded OSes – Examples – Linux

- There exist a number of Linux distributions specifically designed for embedded devices
  - Used in: consumer electronics (i.e. set-top boxes, smart TVs, personal video recorders, in-vehicle infotainment, networking equipment (routers, switches, wireless access points or wireless routers, ...), machine control, industrial automation, navigation equipment, spacecraft flight software, and medical instruments
  - Motivations
    - The Linux kernel is available for most of the ISAs on the market
      - Provided that the system in hand has enough resources
    - Linux is highly customizable
    - Open source
- Many embedded OSes are based on Linux

Embedded OSes – Examples – Real-Time Linux

- Based on the PREEMPT_RT patch for the Linux kernel
- Implements several real-time and non real-time scheduling policies. Real-time policies:
  - SCHED_FIFO: Tasks have a priority between 1 (low) and 99 (high). A task running under this policy is scheduled until it finishes or a higher prioritized task preempts it
  - SCHED_RR: This policy is derived from SCHED_FIFO
    - A task runs for the duration of a defined time slice (if it is not preempted by a higher prioritized task). It can be interrupted by a task with the same priority once the time slice is used up
  - SCHED_DEADLINE: This policy implements the Global Earliest Deadline First (GEDF) algorithm
    - GEDF is a version of EDF for multiprocessors
    - Tasks scheduled under this policy can preempt any task scheduled with SCHED_FIFO or SCHED_RR

https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/start

FEDERICO REGHENZANI, GIUSEPPE MASSARI, and WILLIAM FORNACIARI, *The Real-Time Linux Kernel: A Survey on PREEMPT_RT*,

Embedded OSes – Examples – Real-Time Linux

- Preemption models:
  - Fully Preemptible Kernel (RT): All kernel code is preemptible except for a few selected critical sections
- Threaded interrupt handlers are forced
- Supports priority inheritance
- RT throttling is a mechanism that limits the execution time of real-time tasks per period
  - When the real-time application has the highest possible priority and is scheduled with SCHED_FIFO policy, no other task can preempt it. If the real-time task fails, it may lead to the system blocking all other tasks and scheduling this loop with a CPU load of 100 percent

https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/start

FEDERICO REGHENZANI, GIUSEPPE MASSARI, and WILLIAM FORNACIARI, *The Real-Time Linux Kernel: A Survey on PREEMPT_RT*,

Embedded OSes – Examples – Android

- Open source
  - Based on a modified version of the Linux kernel and other open source software
- Originally developed for smartphones
  - Different versions for other devices
    - Android Wear
    - Android Things

Embedded OSes – Examples – mBed OS

- Free, open-source embedded operating system
- Designed specifically for the "things" in the Internet of Things
- Designed to develop connected products based on an Arm Cortex-M microcontrollers
- Supports real time

In Short ...

- Software on bare metal
- Polling and interrupts
- Real time
- CPU scheduling
- Communication among processes