Real-time response & Real-time operating systems (RTOS)

Luis Gonzalez, PhD
Computer Science Department
Tecnologico de Monterrey, Campus Guadalajara

Outline

- Introduction
- Concurrency
- Real-time planning algorithms
- Reentrant code
- Design patterns

Outline

- **Introduction**
- Concurrency
- Real-time planning algorithms
- Reentrant code
- Design patterns

Introduction

- It is time to consider the issues involved in the accurate measurement of time
- These issues are important cause many embedded systems must satisfy real-time constraints
- Real-time applications imply deterministic processing, i.e., guaranteeing that a particular activity/task will always be completed in a well-defined time interval
- Most embedded SW is multitasking in nature, supporting multiple tasks running concurrently on the same CPU
- But one CPU can only run one instruction at a time
- But multiple tasks need to run simultaneously in an embedded application

Introduction

- Example : Temperature controller
  - Components
    - Temperature sensor
    - Control panel with keypad and LCD
  - Specs
    - The controller must maintain a room within a temperature range
    - If the temperature goes above that range, a red LED is turned on
    - It the temperature goes below that range, a green LED is turned on
    - Both LEDs are off if the temperature is within the temperature range
    - The current temperature is always displayed on the LCD
    - Users can reset the upper and lower limits of the temperature range at any time using the keypad

Introduction

- Specs suggest that the software must run at least two tasks concurrently

Task 1

```
while (1)
{
      sample ADC with sensor;
      rest temp;
      if (temp > up_T) turn on Red;
      else if (temp < low_T) turn on Green;
      else turn off both
      wait (100 ms);
}
```

Task 2

```
while (1)
{
      switch(key)
      {
            Up: Set up_T;
            break;
            Low: Set low_T
            break;
}
```

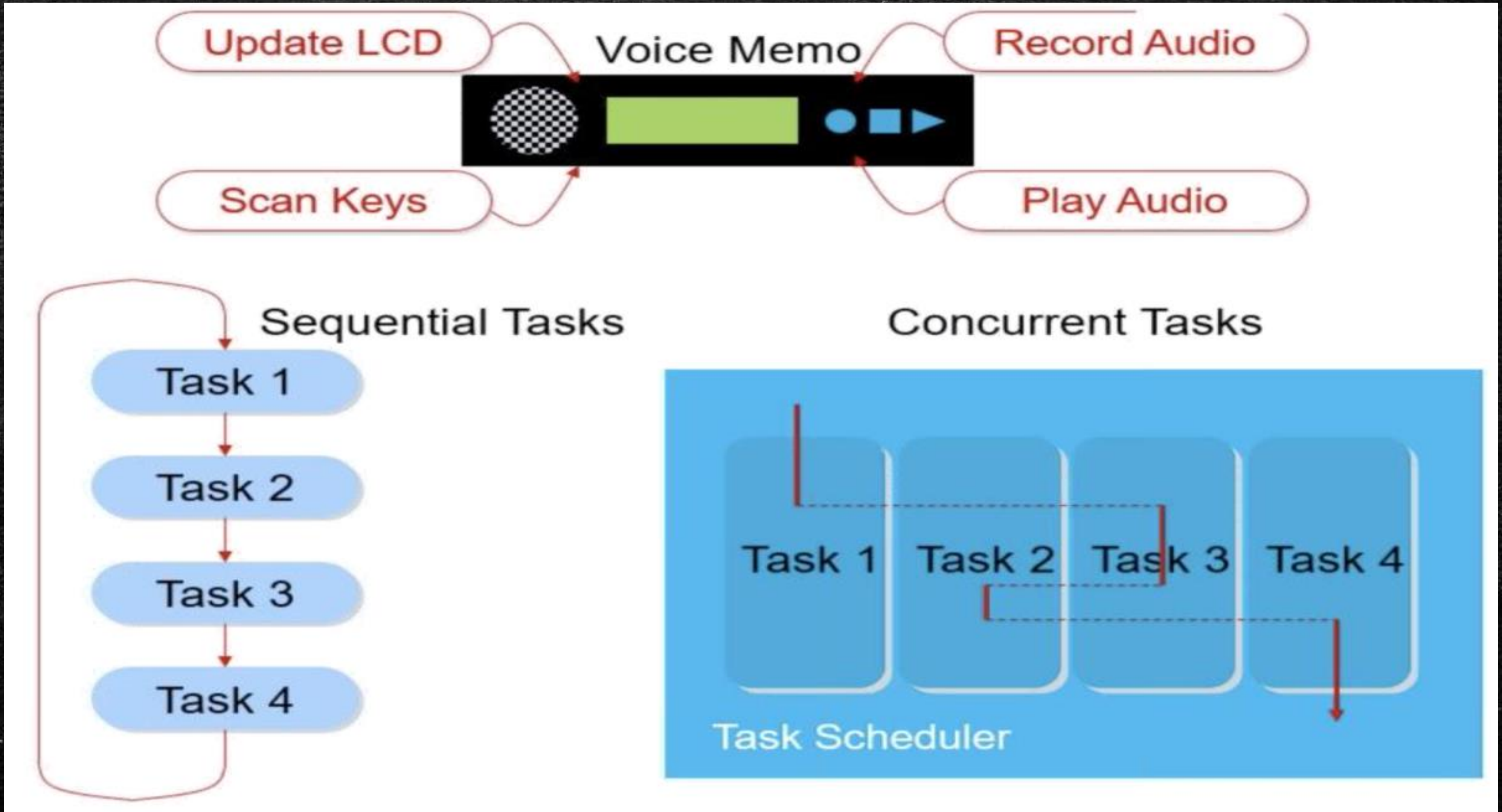
Introduction

- Task 1
  - monitors the current temperature and takes action when the temperature goes beyond the temperature range
  - Samples the temperature sensor every 100 ms
- Task 2
  - Take user inputs from the keypad to reset the temperature upper and lower limits

This very simple embedded system deals with real-time and multiple tasks. Two options can be identified to implement this system

- Real-Time operating system (RTOS)
- Using the interrupt mechanism supported by all microprocessors/microcontrollers

Why multitasking?



OS, RTOS or Baremetal?

- An OS provides a set of system and platform execution services for the app developer, specially around the management and use of target system resources, such as
  - Memory
  - Concurrency units (processes, tasks or threads)
  - Event queues
  - Interrupts
  - Hardware
  - Application programs

OS, RTOS or Baremetal?

- Most OSs do not provide any guarantee about timeliness
- Desktop OSs may invoke unpredictable delays due to internal processing, memory management or garbage collection at unforeseeable times
- This unpredictability makes them unsuitable for real-time and embedded environments that are constrained in time and resources

OS, RTOS or Baremetal?

- A real-time operating system (RTOS) is a multitasking operating system intended for real-time and embedded applications
- It is a piece of software that allows multiple tasks to appear to run simulatenously
- It also allows resources to be managed and shared
- They are written to provide services with good efficiency and performance
- Usually the predictability of the performance is more important than the maximum thoughput

OS, RTOS or Baremetal?

- RTOSs are smaller and often less capable than desktop OSs
- RTOSs do not guarantee real-time performance but they provide an application environment so that appropriate developed applications can achieve real-time performance
- RTOSs run applications and tasks using one of three basic schemes
  - Even-driven systems
    - They handle events as they arise and schedule tasks to handle the processing
    - They use task priority as a quantitative means to determine which task will run if multiple tasks are ready to run
    - Task priorities are typically static, specified at design time
    - Task priorities can also be dynamic, varying the task priorities to account for the current operating conditions

OS, RTOS or Baremetal?

- RTOSs run applications and tasks using one of three basic schemes
  - Periodic runtime tasks (time-base schemes)
  - Cyclical task execution (sequence-based schemes)

- When the embedded systema has too few resources to support an RTOS, the alternative is baremetal
- The application code must cope with managing services and functionality
- The application may simply be a set of interrupt handlers that communicate via a shared resource scheme such as queueing of shared memory

Outline

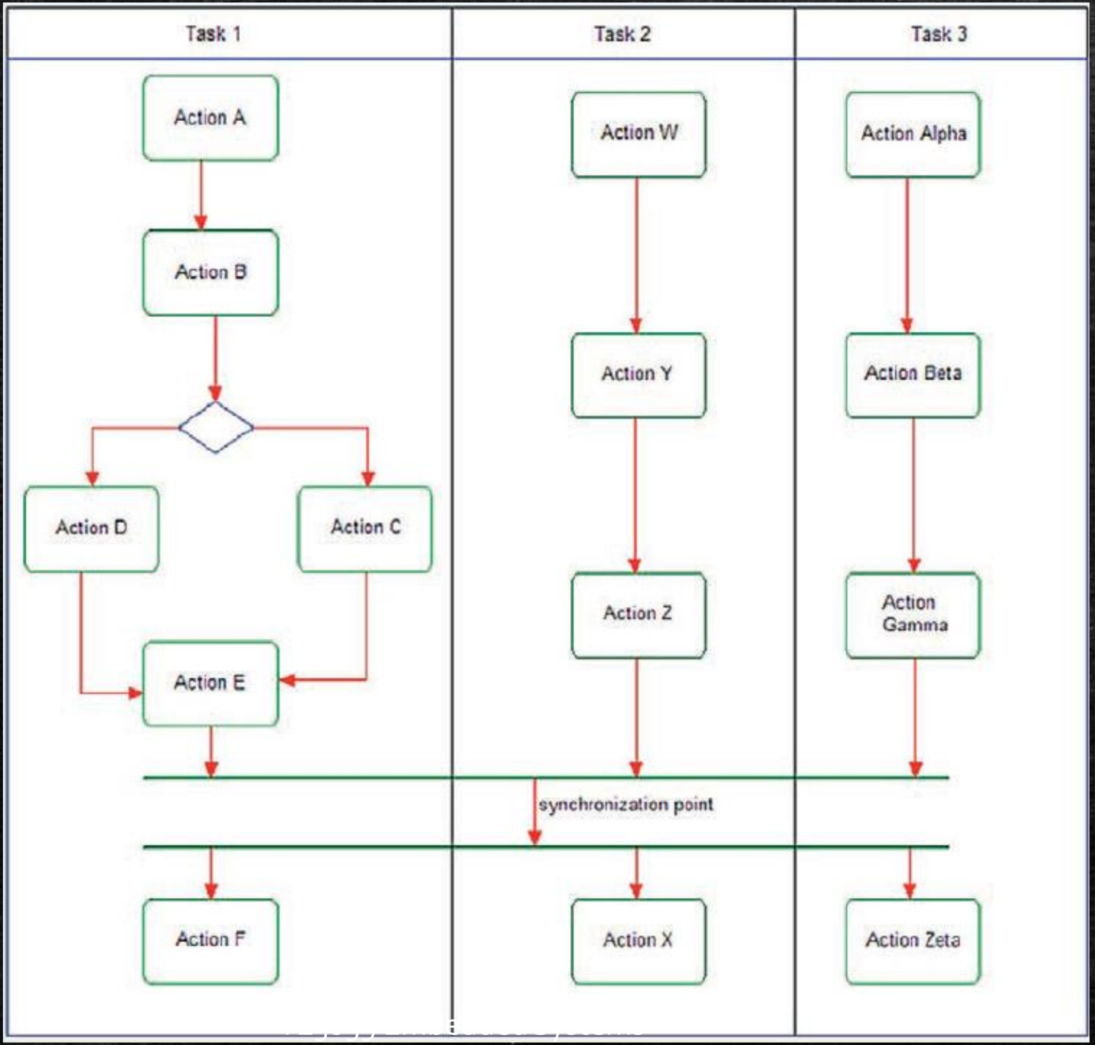
- **Introduction**
- Concurrency
- Real-time planning algorithms
- Reentrant code
- Design patterns

Concurrency

- Most embedded systems need to perform several activities simultaneously
- The key to achieve this is through the definition, understanding and management of the concurrency model of the system
- According to Wikipedia

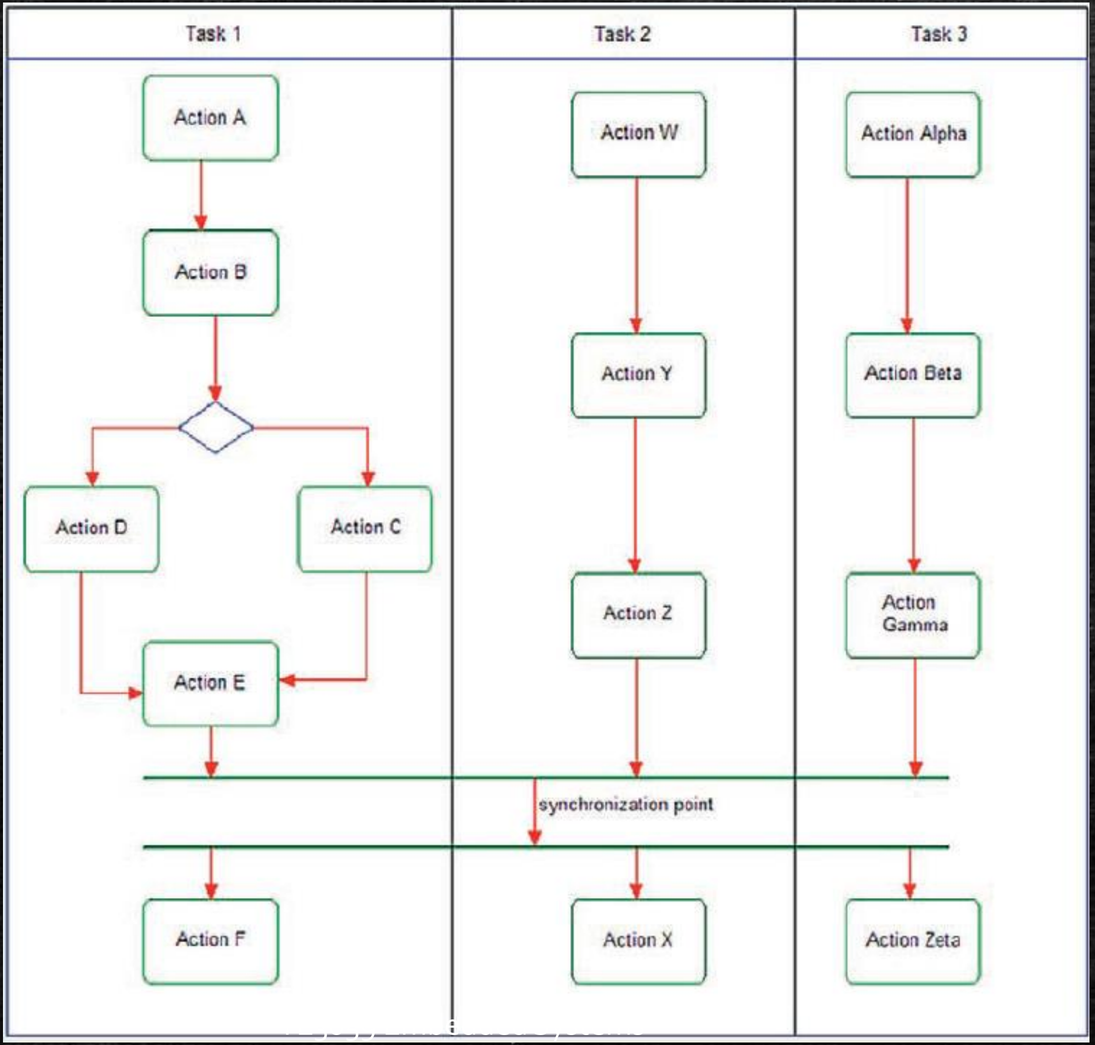
  "Concurrency is the ability of different parts or units of a program, algorithm or problem to be executed out-of-order or in partial order, without affecting the final outcome. This allows for parallel execution of the concurrent units, which can significantly improve the overall speed of the execution in multi-processor and multi-core systems."

Concurrency: Actions, action sequences and tasks



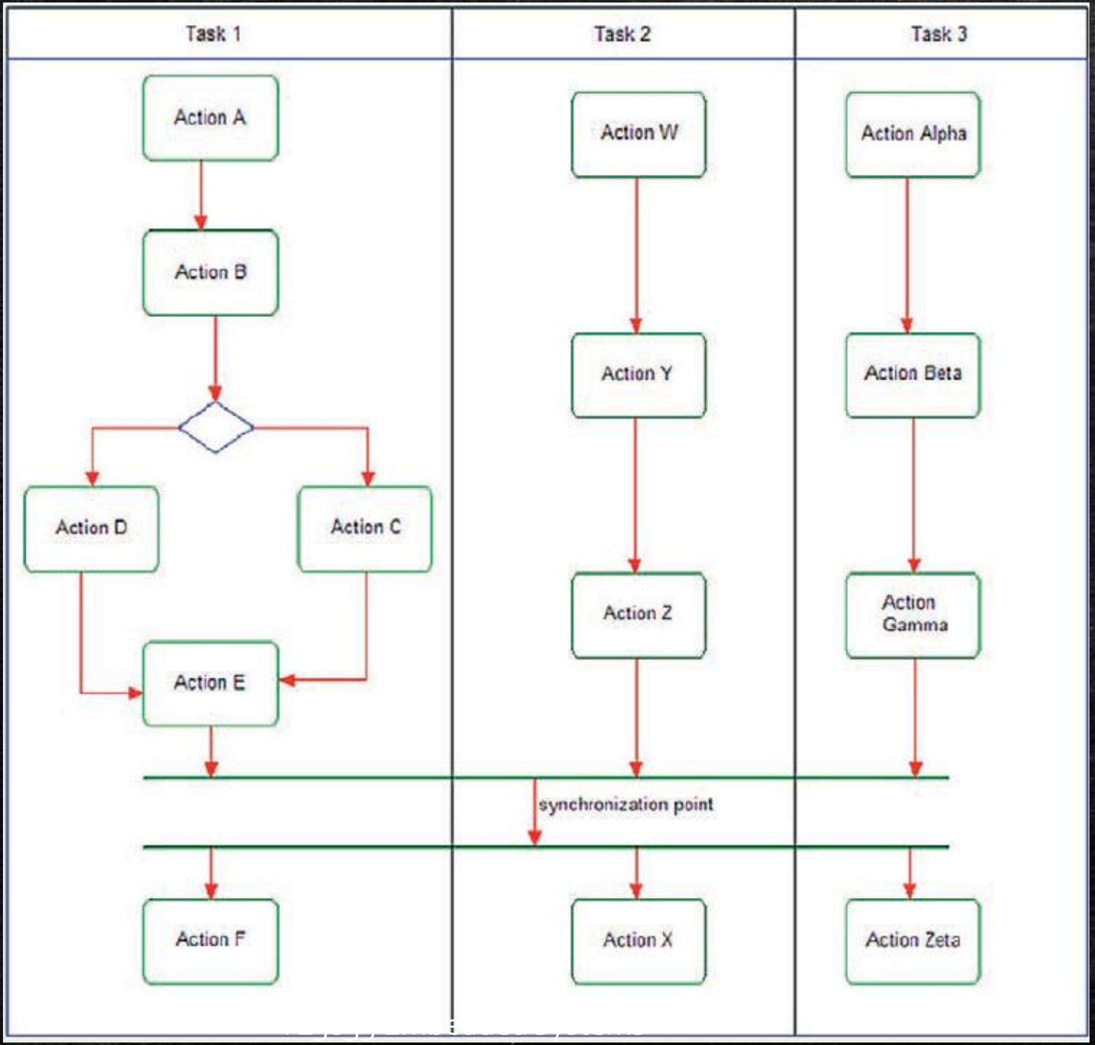
- A task (aka concurrency unit) consists of a set of actions performed in a specific sequence
- These action sequences are executed independently from the others
- Within a task, action sequences are fully known
  - Action A -> action B -> action C or action D -> …

Concurrency: Actions, action sequences and tasks



- What is not known is the order of action execution between tasks
- Which action executes first? Action A? Action Z? action Beta?
  - Answer: We don't know
  - And we don't care! ☺
  - If we cared, the concurrency model was not done well
- There are points in the execution where we might care about the order of execution between tasks. These points are called synchronization points

Concurrency: Actions, action sequences and tasks



- The first bar in the synchronization point is called a join since it joins a set of actions from different tasks into a single thread
- The second bar is called a fork as it forks from the single thread into multiple
- A synchronization point indicates that the flow cannot proceed past the synchronization point until ALL predecessor actions have completed
- Once the three actions have completed, control goes on to subsequent actions
- Synchronization points are places in the code in which tasks will share data or ensure preconditions are met before proceeding

Concurrency: Communication between tasks

- Tasks can communicate in two fundamental ways
  - Synchronous communications
  - Asynchronous communications

- Synchronous communications are like phone calls
  - Both parties are engaged at the same time and invoke services and exchange data immediately
  - The sender sends and waits for a response
  - Synchronous communication is implemented via function calls

Concurrency: Communication between tasks

- Tasks can communicate in two fundamental ways
  - Synchronous communications
  - Asynchronous communications

- Asynchronous communication is like a postcard
  - The sender "sends and forgets"
  - At some later time, the receiver gets and processes the message or data
  - The message processing is less timely but the sender need not wait until the receiver is ready

Concurrency

- For tasks to execute concurrently, they must run on different CPUs or cores (multi-core CPU)
- Within a given CPU core, multiple tasks are run using pseudo-concurrency
  - Only one task executes at any time instant
  - The processor gives the appearance of concurrency by switching among the ready tasks
  - There are many techniques that can be used to select the task to run
  - These are known as scheduling algorithms

Outline

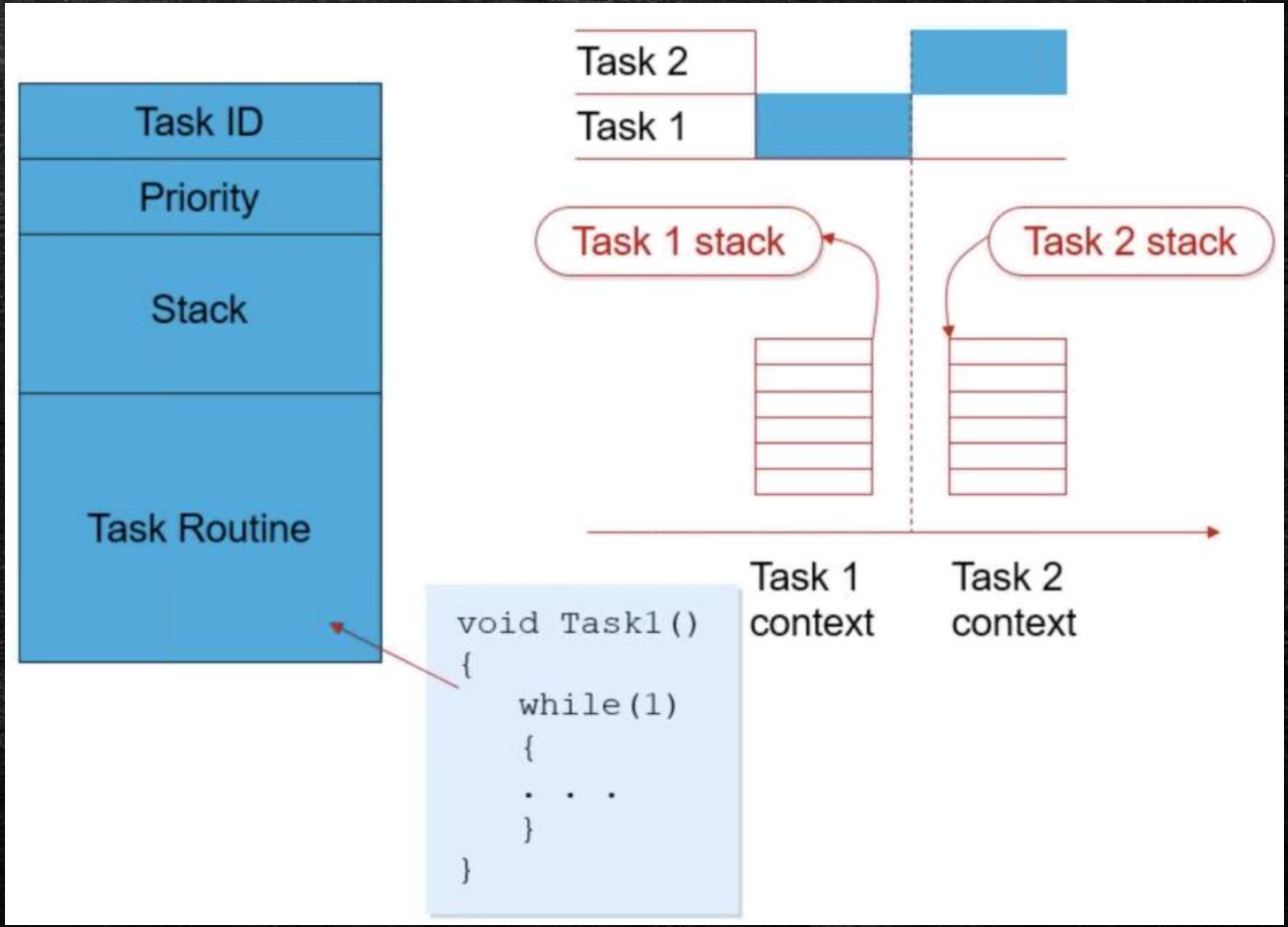
- **Introduction**
- Concurrency
- Real-time planning algorithms
- Reentrant code
- Design patterns

Scheduling

- Scheduling refers to the way the execution of concurrent tasks is performed
- Different techniques exist and the way they schedule tasks have pros and cons
- However, the main goal of any scheduling algorithm is to guarantee that a waiting task is given CPU time when an external event occurs
- With a single CPU, one task runs in the CPU while other tasks are waiting
- The act of reassigning a CPU from one task to another one is called context switch

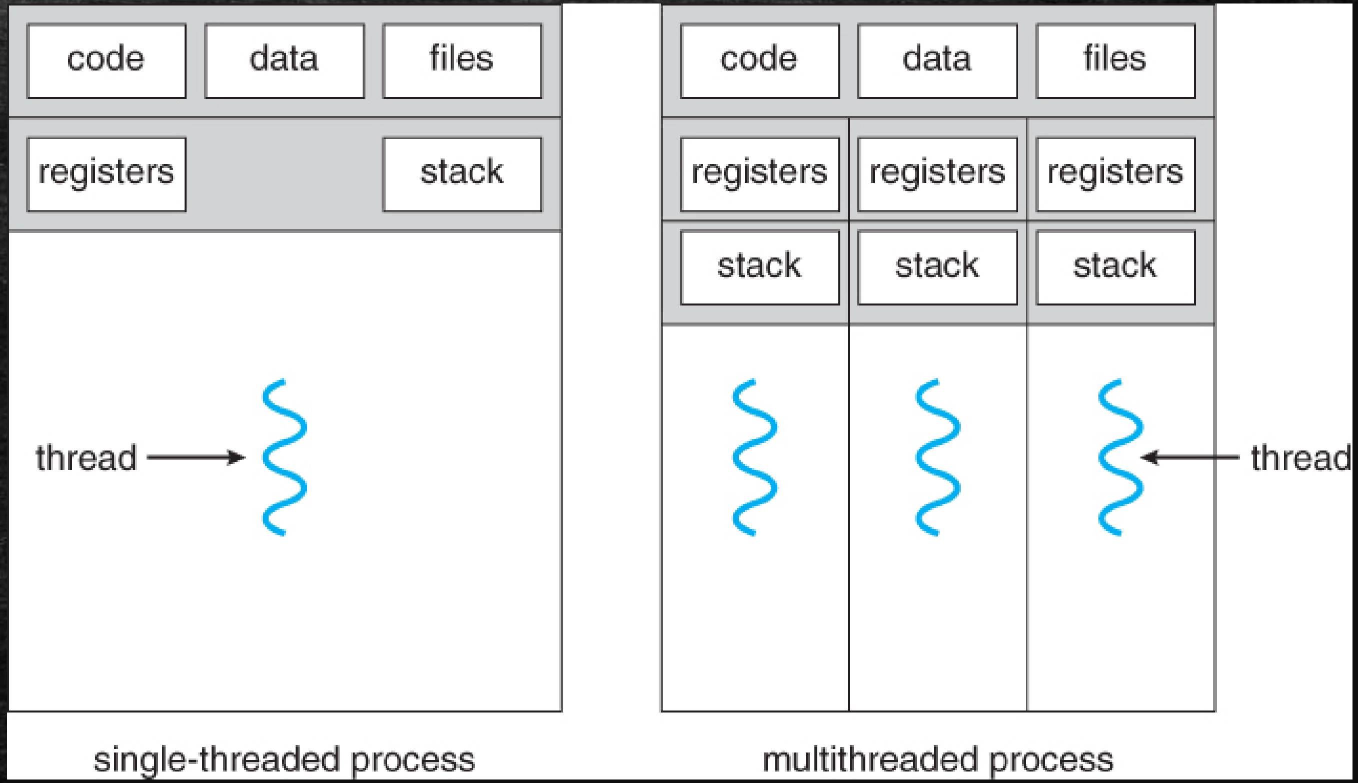
Scheduling

- Example of task switching



Scheduling

- A task represents an activity such as a process or a thread
- Threads are lightweight processes which share an entire memory space
- Threads are processes that run in the same memory context and may share the same data while executing
- A task can have multiple threads

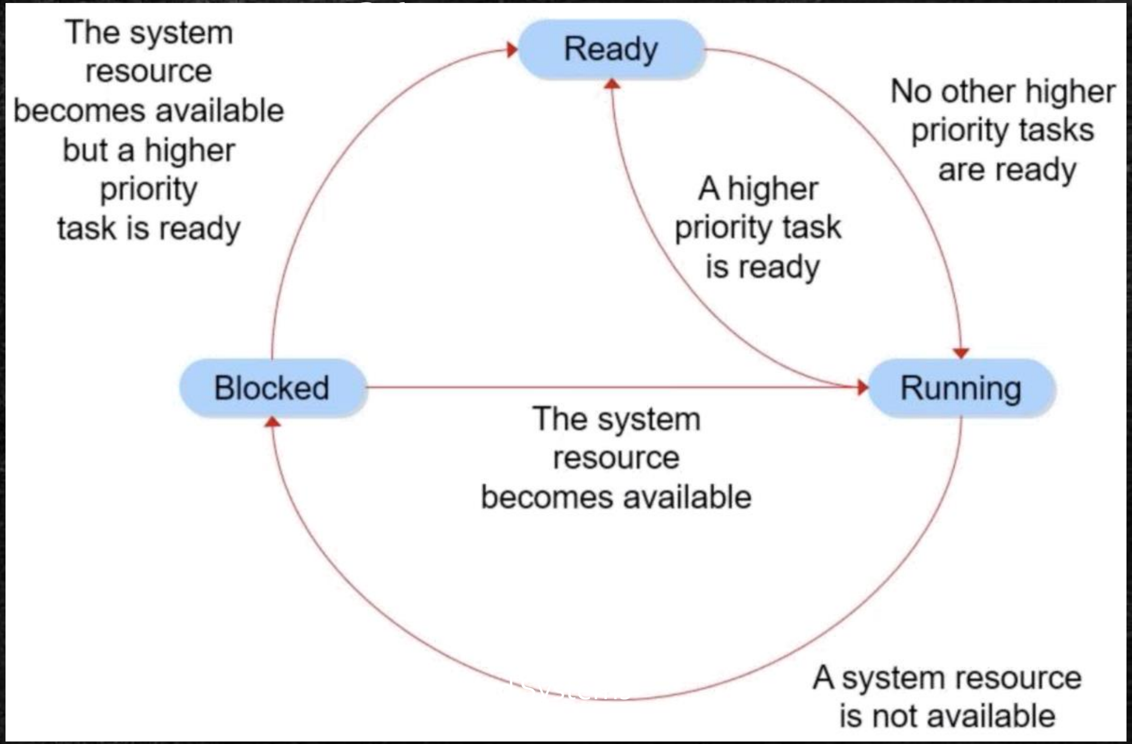


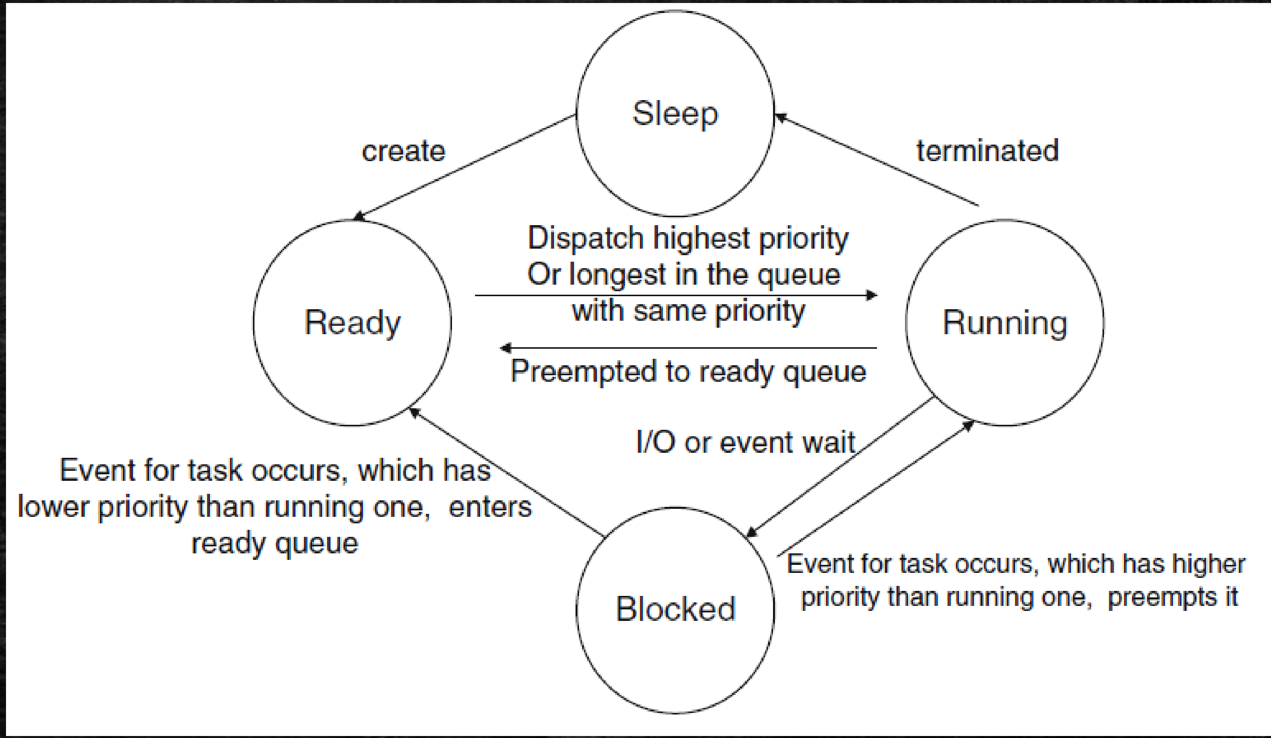
Scheduling – Timing requirement

- Each task must keep track of its own concerns: CPU register status, program count, memory space and stack
- This way, the CPU can switch back and forth between these tasks
- When an event request occurs in a reactive embedded system, the target task must respond to the event within a preset time
- A parameter of a task called Worst-Case Execution Time (WCET) indicates the maximum length of time a task could take to execute
- A task often shows a certaing variation of execution times depending on the input data or the conditions of the embedded system
- The actual response time may be shorter that the WCET

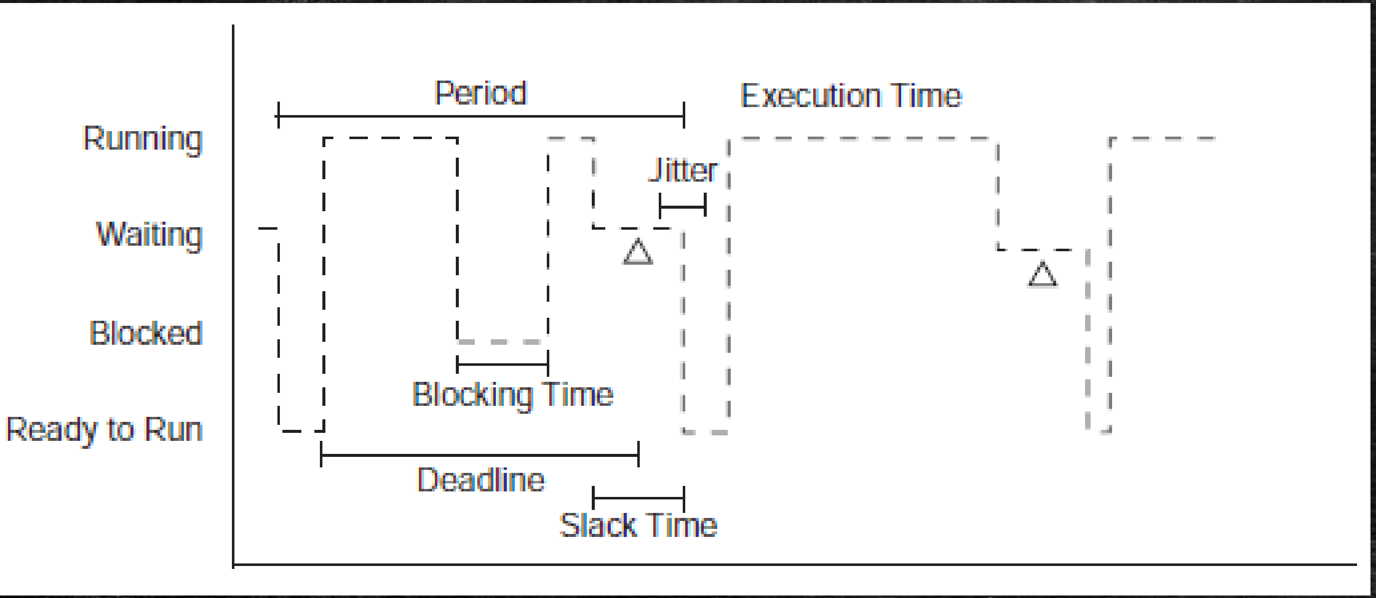
Scheduling – Task life cycle

- Every task in an embedded system has a life cycle
- Life cycle is important to determine time requirements of the task
- Depending on the scheduling algorithm/RTOS used, task life cycle





Scheduling – Time related terminology



The life cycle of a task as a function of time

- Assume a periodic task (it executes at a more or less fixed rate)
- Terminology
  - Period: duration of the time interval between task initiations
  - Jitter: variation in the period
  - Execution time: time required to complete the activities of the task
  - Deadline: period of time between when the task becomes ready to run and when if must complete

Scheduler

- The task selection for CPU (running state) is determined by a scheduler
- A scheduler has scheduling rules to select the running task
- Example of scheduling rules (RTX_51)
  - The task with the highest priority of all tasks in the READY state is executed first
  - If several tasks of the same priority are in the READY state, the task that has been ready the longest will be the next to execute
  - Exception: round-robin scheduling
- Scheduling techniques come next…

Scheduling techniques – a brief intro

- Cyclic executive scheduling
  - The scheduler consists of an infinite loop that executes the tasks one after the other
  - It is not flexible and can have poor response to incoming events since an event processed by a task must wait until that task is run within the cycle
  - Task execution must be short if other tasks are going to run in a timely fashion

```
void main(void) {
    /*  global static and stack data  */
    static int nTasks  =  3;
    int currentTask;

    /*  initialization code  */
    currentTask  =  0;

    if (POST()) { /*  Power On Self Test succeeds  */
        /*  scheduling executive  */
        while (TRUE) {
            task1();
            task2();
            task3();
        }; /*  end cyclic processing loop  */
    }
}
```

Scheduling technique – a brief intro

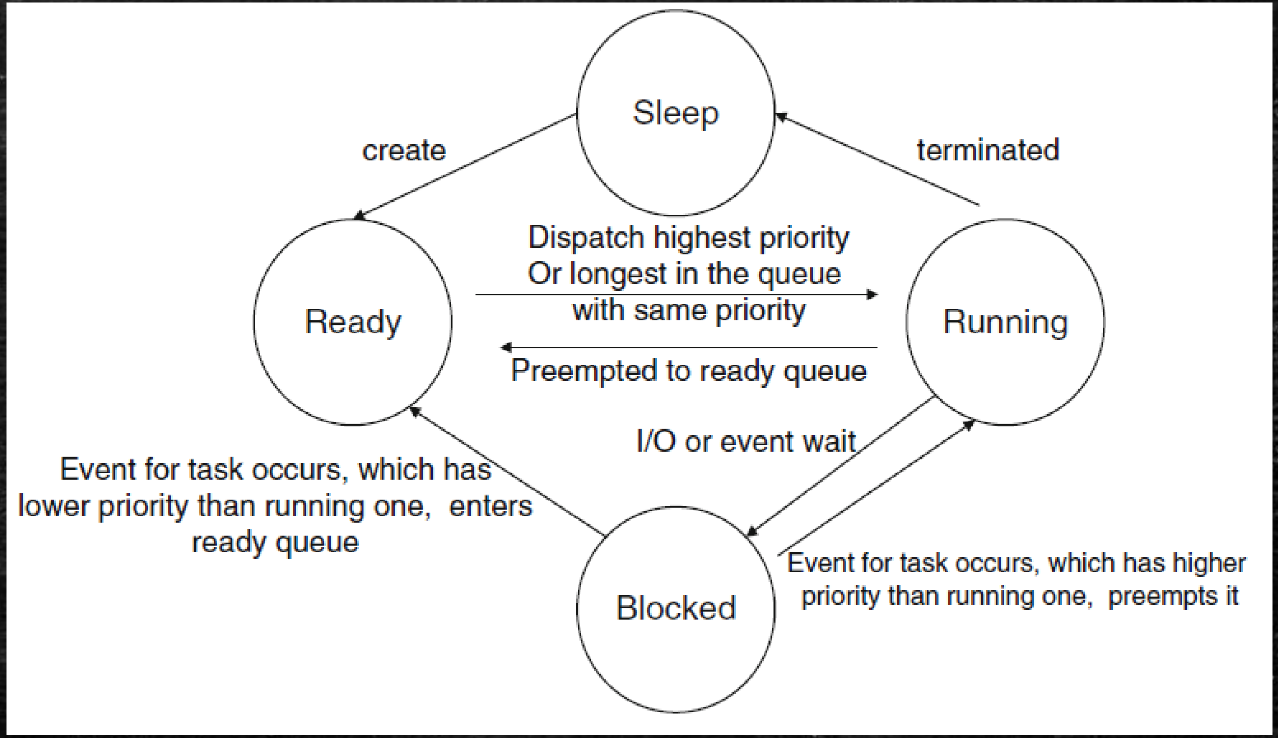
- Time-triggered cyclic executive
  - The cycle starts at a time boundary
  - If the schedule completes the cycle before the next time boundary, it sleeps until the start of the next epoch
- Cooperative round-robin
  - it is a cyclic executive in which the taks don't run to completion but instead run to specific points at which they release control back to the scheduling loop
  - This allows long tasks to run while still permitting other tasks to make progress in their processing

Scheduling technique – a brief intro

- Cyclic and round-robin schedulers present a number of problems
- They are not responsive to urgent events; the task that processes the urgent event simply runs in its sequence
- They are not flexible since the scheduler order is determined at design time
- They don't scale well to large number of tasks
- A single misbehaving task stops the entire system from execution
- These schedulers are best suited to very simple systems

Scheduling technique – a brief intro

- The other primary kind of scheduling is preemptive scheduling
- The scheduler stops the currently executing task when it decides to do so and runs the next task
- The criterion fro preemting a task and the selection of the next task in sequence depends on the particular scheduling rules
- Task life cycle presented in slide 27 are examples of preemptive scheduling



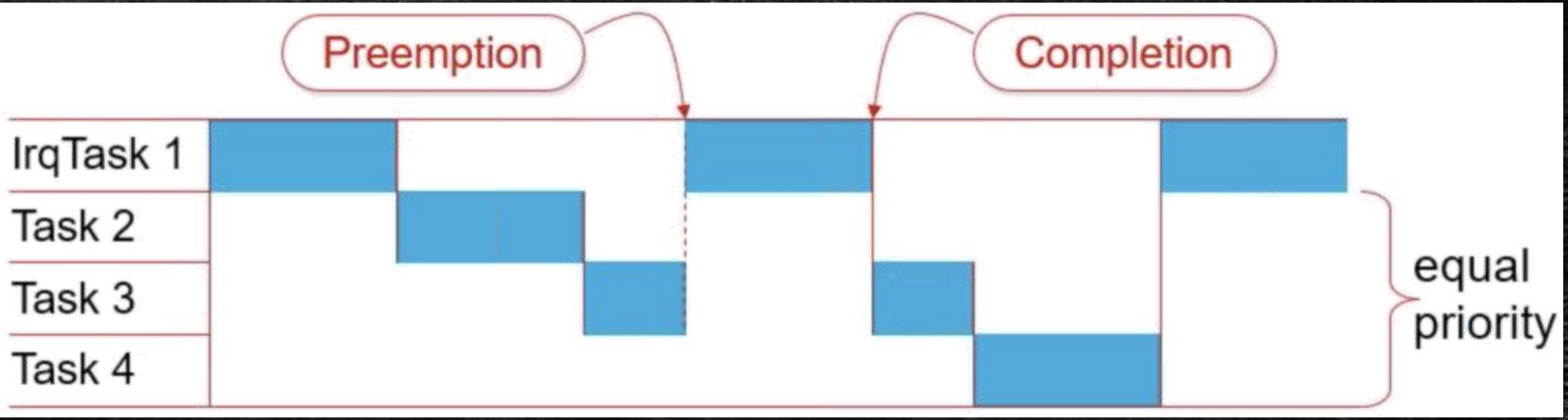
Scheduling technique – a brief intro

- The most common scheduling technique is priority based scheduling
- Each task is assigned a priority, a scalar value that will be used to select the task to execute
- The task runs in an infite loop and will be interrupted by the scheduler when it is appropriate for other tasks to run

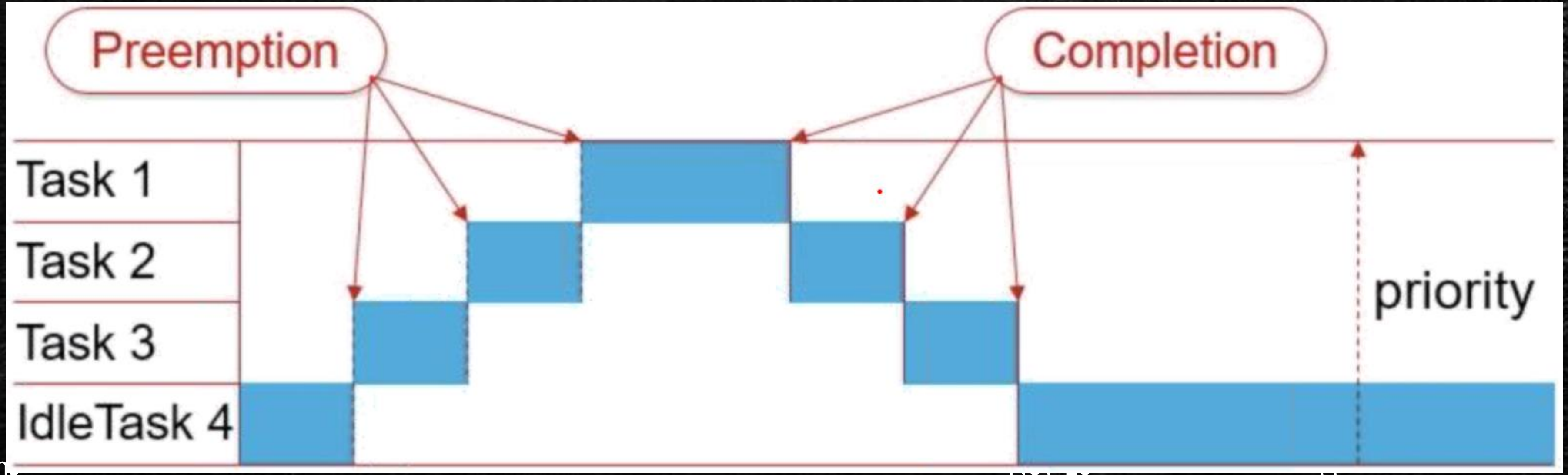
```
/*
     here is where private static data goes
*/
static int taskAInvocationCounter = 0;

void taskA(void) {
     /*  more private task data  */
     int stufftoDo;

     /*  initialization code  */
     stuffToDo  =  1;

     while (stuffToDo) {
          signal  =  waitOnSignal();
          switch (signal) {
               case signal1:
                    /*  signal 1 processing here  */
                    break;
               case signal2:
                    /*  signal 2 processing here  */
               case signal3:
                    /*  signal 3 processing here  */
          };
     }; /*  end infinite while loop  */
}; /*  end task  */
```

Scheduling Techniques – a brief intro

Round Robin



Priority-based



Non-Preemptive Scheduling

- Simple task scheduling method for periodic time requirement systems
- Systems belonging to this type of scheduling are periodic systems or systems where “cooperative” multitasking is implemented
- In cooperative multitasking the process that is running must offer control to other processes/tasks after a given amount of time
- Another example is cyclic scheduling where tasks can be scheduled in a fixed static table

Non-Preemptive Scheduling

- Non-preemptive scheduling is typically used in embedded systems that have a set of tasks and all of them have a fixed period.
- Aperiodic events can be estimated by their worst case interval gap between two consecutive task events
- All tasks are independent and WCET is known in advance, so that the deadline is at the end of the WCET
- The advantage of non-preemptive scheduling is zero-overhead for context switches between process/tasks
- The disadvantage is the inflexibility

Non-Preemptive Scheduling

- It there are n tasks and the WCET of the i^{th} task is c_i, then the period of any task Ti must satisfy

$$T_i \geq \sum c_j \quad (j = 1,2, \dots, n)$$

- Otherwise some tasks may miss its deadline
- As a result, software design should make each task as shorter as possible. For long tasks, they should be broken into many small tasks

Non-Preemptive Scheduling

- Cyclic schedule example

Task	Execution time	Period
Task 1	20 ms	50 ms
Task 2	25 ms	100 ms

- Period of T1 (50ms) $\geq$ c1 (20ms) + c2(25ms)
- Period of T2 (100ms) $\geq$ c1 (20ms) + c2(25ms)
- The tasks are schedulable, they wont miss their deadline

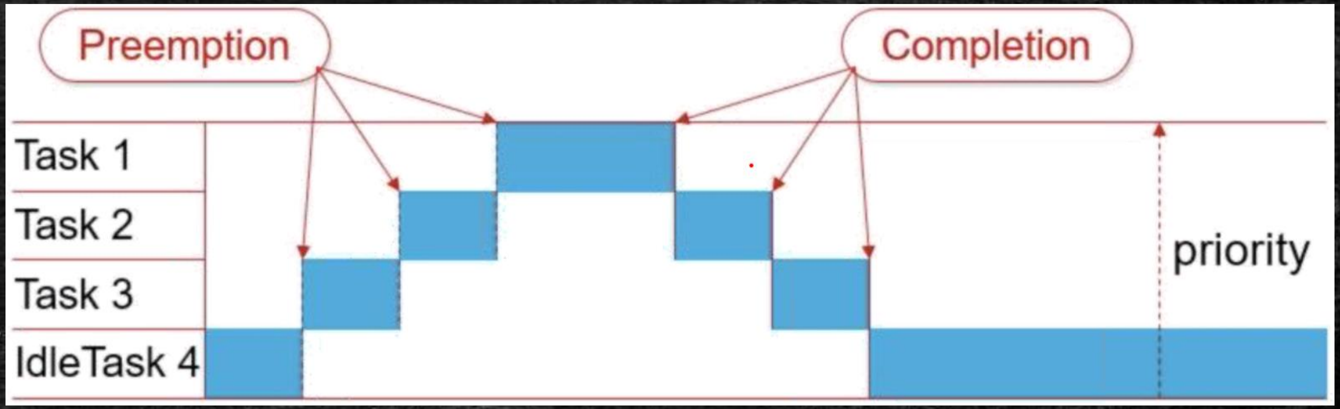
Non-Preemptive Scheduling

- Cyclic schedule example
  - Assumptions: there is a timer set on the period of 50 ms

```
while(1)
{
    Wait_for_50ms_timer_interrupt;
    do task1;
    Wait_for_50ms_timer_interrupt;
    do task1;
    do task2;
}
```

Pre-emptive Scheduling

- Priority-driven scheduling
- Reactive embedded systems must respond and handle external or internal events with different urgency
- Some events have hard real-time constraints while the others may only have soft real-time constraints
- That is, some tasks should be assigned higher priority than other tasks



Pre-emptive Scheduling

- The CPU always goes to the highest priority process that is ready
- Pre-emptive/priority-based scheduling has several algorithms
  - Static priority-based
    - Static timing scheduling
    - Round-robin scheduling
    - Rate Monotonic Scheduling (RMS) – based on the deadline based analysis called Rate Monotonic Analysis (RMA)
  - Dynamic priority-based
    - Easrliest Deadline First (EDF): it assigns priorityies at runtime based on upcoming execution deadlines
    - Priority is time varying rather than fixed

Rate Monotonic Scheduling

- Priority-based static scheduling method for preemptive real-time systems
- It can guarantee the time requirement and maximize the schedulability as long as the CPU utilization is below 0.7
- RMA (Rate Monotonic Analysis) has proven to be optimal among all static priority scheduling algorithms

Rate Monotonic Scheduling

- It assigns shorter period/deadline processes higher priority at design time, assuming that the processes have the following properties
  - No resource sharing
  - Deterministic deadlines are exactly equal to periods
  - Context switch times are free and have no impact on the model
- Once the priority of a task is assigned, it will remain constant for the lifetime of the task

Rate Monotonic Scheduling

- Example 1

Task	Execution Time	Period (Deadline)	Priority	Utilization
Task 1	20 ms	50 ms	2 (high)	40%
Task 2	45 ms	100 ms	1 (low)	45%

- Task 1 must meet its deadline of 50, 100, 150, …
- Task 2 must meet its deadline at 100, 200, 300, …
- Task 1 is assigned higher priority cause it has shorter period

Rate Monotonic Scheduling

- Example

Task	Execution Time	Period (Deadline)	Priority	Utilization
Task 1	20 ms	50 ms	2 (high)	40%
Task 2	45 ms	100 ms	1 (low)	45%

- Schedule:

T1(20)	**T2(30)**	**T1(20)**	**T2(25)**	**(5)**	**T1(20)**	**T2(30)**	**Continue the same pattern**

50 100 150

Rate Monotonic Scheduling

- Schedule:

T1(20)	**T2(30)**	**T1(20)**	**T2(15)**	**(15)**	**T1(20)**	**T2(30)**	**Continue the same pattern**

50 100 150

- At time 50, task 2 is preempted by task 1 cause it is ready every 50ms and has higher priority
- This schedule guarantees all tasks meet their deadline

Rate Monotonic Scheduling

- If task 2 gets higher priority over task 1, then task 1 will miss its deadline at time 50

T2(45)	**T1(5)**	**T1(45)**

50

Rate Monotonic Scheduling

- Example 2

Task	Execution time	Period (Deadline)	Priority	Utilization
Task 1	4 ms	10 ms	3 (high)	40%
Task 2	5 ms	15 ms	2 (medium)	33%
Task 3	6 ms	25 ms	1 (low)	24%

- Periods of tasks obey $T(1) < T(2) < T(3)$
- Therefore $P(1) > P(2) > P(3)$
- CPU utilization is 97%

Rate Monotonic Scheduling

- Schedule:

T1(4)	T2(5)	T3(1)	T1(4)	T3(1)	T2(5)	T1(4)	T3(1)	-3

10 15 20 25

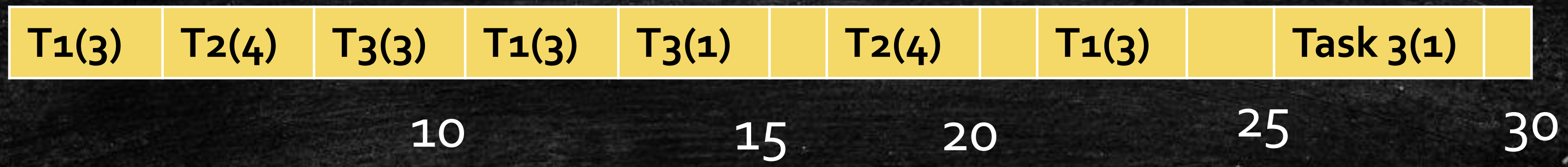
- At time 25, task 3 misses its dadline by 3 ms
- This is due to the fact that total utilization is 97% which is beyond the schedulable bound of 70%

Rate Monotonic Scheduling

- Example 2: Let's change the utilization rate

Task	Execution time	Period (Deadline)	Priority	Utilization
Task 1	3 ms	10 ms	3 (high)	30%
Task 2	4 ms	15 ms	2 (medium)	26%
Task 3	4 ms	25 ms	1 (low)	16%

- CPU utilization is 72%



It is not always possible to fully utilize the CPU

Rate Monotonic Scheduling

- According to Rate Monotonic Analysis, fixed-priority scheduling has a worst case schedulable bound of

$$U_n = \sum_{i=1}^{n} \frac{c_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

n is the number of tasks in the system

- As the number of tasks increases, the schedulable bound decreases, eventually approaching its limit of $ln2 = 69.3\%$
- It has been shown that for a set of n periodic tasks with unique periods, a feasible schedule that will meet deadlines exist if CPU utilization is < Un

Rate Monotonic Scheduling

- The other 30% of CPU utilization can be dedicated to lower-priority non real-time tasks
- The context switch cost for the RMS is very high
- In order to fully make use of CPU time and also to meet all deadlines, we need to use a dynamic priority scheduling algorithm

Dynamic Scheduling with EDF

- In a dynamic priority-based scheduling the priority of a process changes in order to increase the CPU utilization and to allow all tasks meet their deadline
- In Earliest Deadline First (EDF), higher priority is assigned to those tasks that are closer to their dealine at runtime
- EDF can be applied to both periodic and aperiodic tasks if deadlines are known in advance
- EDF is not very practical and is nor as popular as RMS

Dynamic Scheduling with EDF

- In a dynamic priority-based scheduling the priority of a process changes in order to increase the CPU utilization and to allow all tasks meet their deadline
- In Earliest Deadline First (EDF), higher priority is assigned to those tasks that are closer to their dealine at runtime
- EDF can be applied to both periodic and aperiodic tasks if deadlines are known in advance
- EDF is not very practical and is nor as popular as RMS
- EDF must recalculate the priority of each process at every context switch time (preemption time)

RTOS revisited