# Scheduling

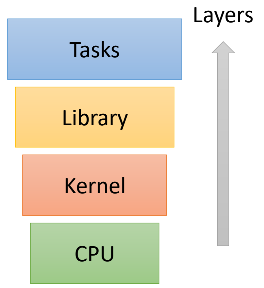
Fabrice Muller
Polytech Nice Sophia
Fabrice.Muller@univ-cotedazur.fr



# Introduction

# Scheduling without a RTOS

- Simple solution with standalone application
- Use of datas from polling or interrupt solutions
- Libraries of deterministic functions
- Scheduling of tasks by hand

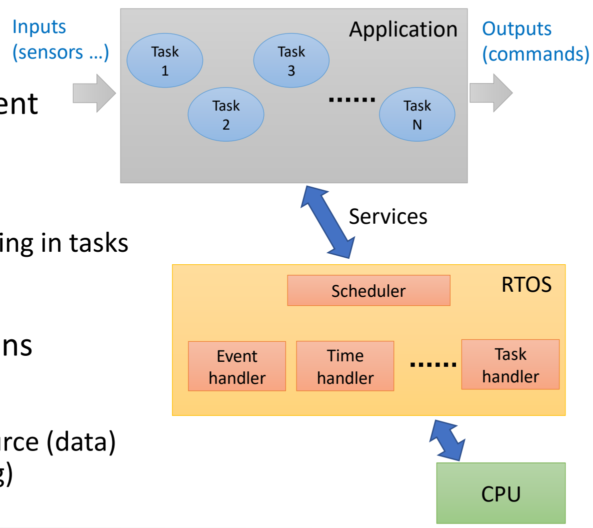






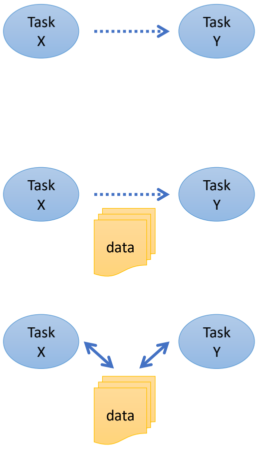
# Scheduling with a RTOS

- Need of services
- Input/output management
  - Input/output handler
  - Interrupt handler
- Task scheduling
  - Organization of functioning in tasks
  - Scheduling policy
  - Time handler
- Inter task communications
  - Synchronization (event)
  - Communication (data)
  - Access to a shared resource (data)
  - Time (counter, watchdog)



# Inter task communications

- Synchronization
  - Event
  - Semaphore (boolean)
  - Rendezvous
- Communication
  - Mail box
  - Semaphore with counter
- Shared critical resources
  - Semaphore with mutual exclusion

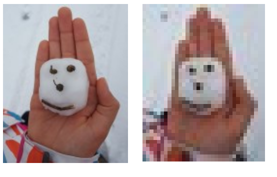






# Quality of Service (QoS)

- Subjective notion
- Depend on systems and services
  - Response time (interactive or critical applications)
  - Bit rate (video broadcast)
  - Availability (access to shared services)
  - Packet loss rate (voice or video perception)
  - Signal-to-noise ratio (communication)
  - …



- For application layer services for telephony and streaming video, QoS is the acceptable cumulative effect on subscriber satisfaction of all imperfections affecting the service.
- High QoS is often confused with a high level of performance (high bit rate, low latency and low bit error rate …)

# Criteria for real-time computing

- Hard real time (QoS = 100%)
  - Missing a deadline = total system failure
  - To be predictable, deterministic and reliable
  - Use of mathematical techniques
- Firm real time (X% ≤ QoS ≤ 100%)
  - Infrequent deadline misses = tolerable (X%)
  - May degrade the system's QoS
  - The usefulness of a result is <u>wrong after its deadline</u>
  - Minimize the probability of missing a deadline several times consecutively
- Soft real time (QoS ≤ 100%)
  - The usefulness of a result degrades after its deadline but maybe acceptable



# Hard real time - Definitions

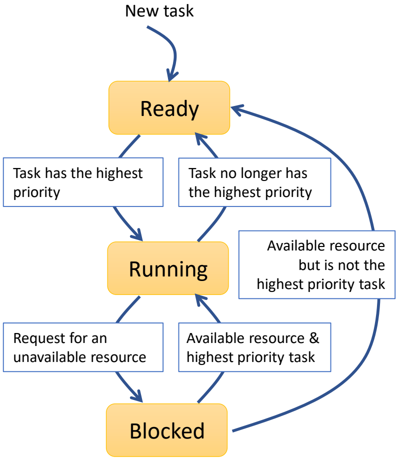
- Predictability
  - The performance of the application must be defined in all possible cases so as to ensure respect for time constraints
  - Use of the <u>worst case</u>
- Determinism
  - No uncertainty about the behavior of the system
  - Behavior is always the same for a given context
- Reliability
  - Ability of a system to achieve and maintain functionality under normal conditions of use
  - Respect of real-time constraints
- Fault Tolerance
  - Fault-tolerance is the ability of a system to maintain its functionality even in the presence of faults
  - Like Reliability even if certain failures have appeared

# What is a Task ?



# Typical state of a task

- Ready state
  - The task is ready to run but cannot because a higher priority task is executing
- Running state
  - The task is the highest priority task and is running
- Blocked state
  - The task has requested a resource that is not available
  - The task has requested to wait until some event occurs
  - The task has delayed itself for some duration
- Some kernels (VxWorks, FreeRTOS …) define more granular states such as suspended, pended, delayed, blocked …



# Priority

- The priority level allows you to define who has access to
  - CPU
  - Shared resources
- The kernel (or tick OS) has the highest priority
- Idle task has the lowest priority (normally zero)

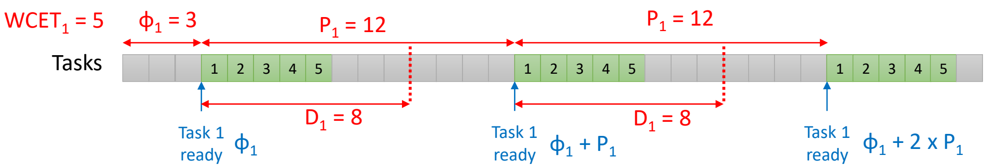
- FreeRTOS Example
  - Each task is assigned a priority from 0 to configMAX_PRIORITIES-1
  - Low priority numbers denote low priority tasks



# Periodic Tasks

- Task $T_i$ repeats after a certain fixed time interval (period)
- Characteristics of $T_i$
  - $\phi_i$ : phase (from 0s till the first execution of the task i)
  - $P_i$ : period
  - $WCET_i$ : worst case execution time
  - $D_i$ : relative deadline

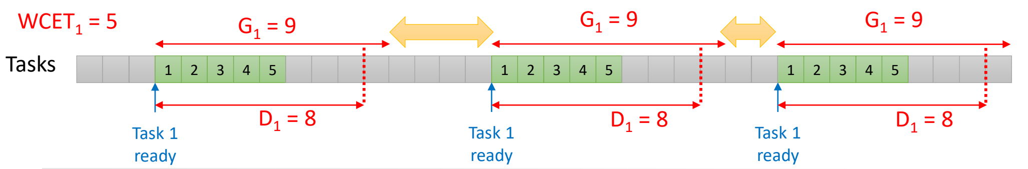
Example of periodic task $T_3$



# Aperiodic / Sporadic Tasks

- Sporadic
  - Task $T_i$ recurs at random instants with a minimum separation between 2 consecutive instances of task $T_i$
  - Characteristics of $T_i$
    - $G_i$ : next instance cannot occur before $G_i$
    - $WCET_i$ : worst case execution time
    - $D_i$ : relative deadline
- Aperiodic
  - Task $T_i$ can arise at random instants ($G_i$ =0s)
  - Two or more instances might occur at the same time
  - Might lead to a few deadline misses (used for firm/soft real-time)

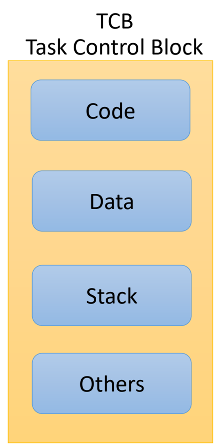
Example of sporadic task $T_3$






# Task descriptor / Task Control Block

- **Data structure describing a task = TCB**
- Scheduler uses TCB for the management of multitask environment
- Code
  - Instructions, constants
- Data
  - Shared with the tasks of the same process
- Stack
  - Contains temporary information
  - Local variables, context, registers
  - Program counter of subroutine
- Others
  - Identifier
  - State
  - Task priority
  - Expected events (rendezvous)
  - …



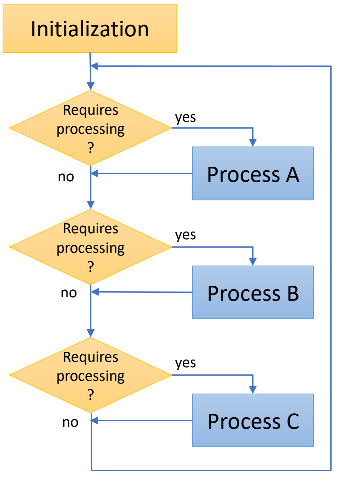
# How to manage tasks ?

The task management is done through a scheduler which defines the order of the tasks with their priority.



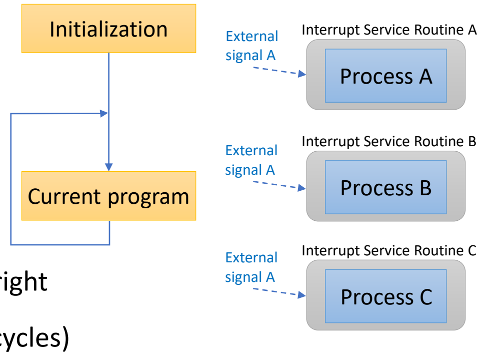

# Polling mode

- Polling mode repeatedly checks whether a device needs servicing
- Polling cycle is the time in which each element is monitored once
- Disadvantages
  - if there are too many devices to check, you risk missing a state change of devices.
  - Increases CPU rate (wastes lots of CPU cycles)



# Interrupt mode

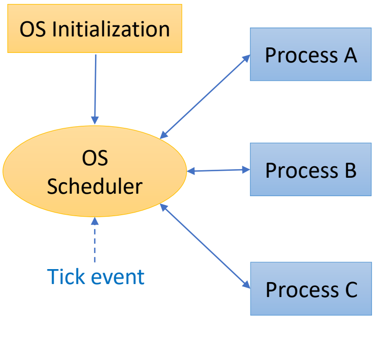
- An interrupt is a signal from a device or from a program
  - Stops the current program
  - Saves registers
  - Runs a interrupt routine
  - Restores registers
- For each interrupt
  - Priority assignment
  - Masking to disable interrupt
- Advantages
  - Each process is triggered at the right time
  - Decreases CPU rate (saves CPU cycles)






# RTOS mode

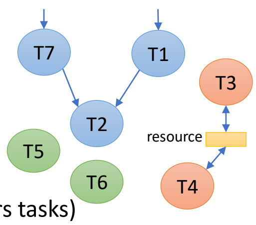
- Scheduler
  - Manages process/tasks
  - Can be seen as an interrupt routine
  - Manages context switch (save/restore context in TCB)
- Tick event
  - Wake up the scheduler periodically
  - Comes from a timer
  - Tick period is configurable



# Scheduler



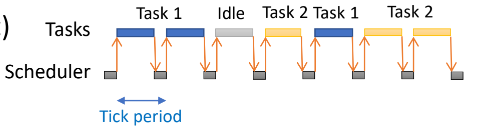

# Some definitions




- Tasks
  - Independent tasks share only the processor
  - Dependent tasks
    - share other resources
    - Have precedence constraints (waiting for data of others tasks)
- Scheduling
  - Offline scheduling is pre-calculated before execution
  - Online scheduling decides dynamically of the execution of tasks
- Cooperative/Preemptive
  - A cooperative scheduler cannot interrupt a task
  - A preemptive scheduler can interrupt a task (and save this context) to execute a higher priority task (restore this context)

# Goals of scheduler

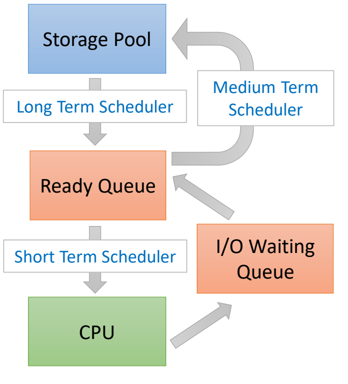
- Scheduler manages process/tasks
- Scheduler handles
  - the selection of a process/tasks for the processor on the basis of a scheduling algorithm
  - the removal of a process/tasks from the processor
- Ensure properties
  - Equity (all processes must run)
  - Load balancing (multi-core context)
  - …
- Objectives are often opposed
  - Minimize response time but …
  - Maximize CPU ratio






# Types of Schedulers

- Types of Processes
  - **I/O-bound process** - it spends his time doing I/O operations; a lot of little CPU bursts
  - **CPU-bound process** – it spends its time doing calculation; small number of very large CPU bursts
- Long-term scheduler / job scheduler
  - Select in the storage pool the processes must be add in the ready queue
  - It must select a careful mixture of I/O bound and CPU bound processes to yield optimum system throughput
  - Not invoked very often (seconds, minutes)
- Short-term scheduler / CPU scheduler
  - Select in the ready queue which process/task should be implemented soon and reserve the CPU
  - Invoked very frequently (in milliseconds), quick response time
- Medium Term Scheduler
  - Remove the processes from the main memory
  - In-charge of handling the swapped out (roll out)-processes
  - Suspended process is moved from ready queue to the storage pool



# Scheduling Policy

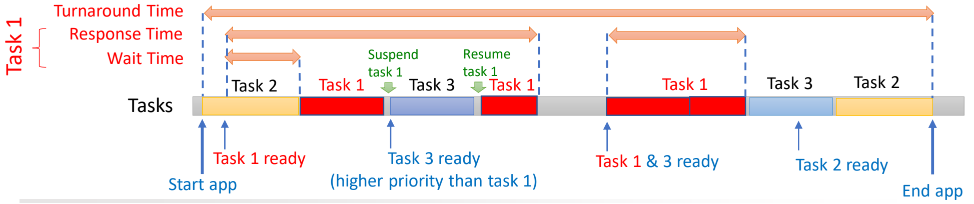


# Objectives

- Improving efficiency of the system
  - Reducing delays and wait times, response times to the system
  - Managing CPU resources better
- Fairness
  - Scheduler shouldn't give unfair advantage to a process/task
  - Important to balance long-running tasks and ensure that the lighter tasks can be run quickly
- Heuristic algorithms for scheduling
  - Trade-off between decision time and optimality
  - Heuristic does not always reach optimality
  - Multi-criteria algorithms
- Lot of scheduling policies exist !

# Criteria for Scheduling

- Reduce the strain on the **CPU Utilization**
  - Manage the percentage of time the CPU is busy
- Optimize the **Throughput**
  - Increase the number of processes completed in a given time frame
- Reduce the **(Average) Wait Time**
  - Waiting time is amount of time a process has been waiting in the ready queue
- Reduce the **Response Time**
  - The response time of a task/thread is defined as the time elapsed between the dispatch (time when task is ready to execute) to the time when it finishes its job (one dispatch)
- Respect the **Turnaround Time**
  - Total time a process takes to run, from start to finish (includes all waiting time)






# Main Execution Times of a task

- Depend on
  - Target architecture (processor, memory accesses, …)
  - Algorithms (conditions, loops)
- **Worst-Case Execution-Times** (WCET)
  - Maximum length of time the task could take to execute on a specific hardware platform
  - Assess resource needs for real-time systems
  - Ensure meeting deadlines
  - Perform schedulability analysis
- **Best-Case Execution-Times** (BCET)
  - Assess code quality
  - Assess resource needs for non/soft real-time systems
  - Ensure meeting livelines (new starting points)
  - Benchmark hardware
- **Average-Case Execution-Times** (ACET)
  - Assess behavior of the real-time systems from simulations

Algorithm

val : range 0 to 255

BCET WCET

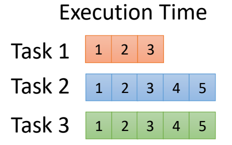
```
void f1() {
  int val = read(PortA);
  if (val = 0) {
    write(PortB, -1);
  }
  else {
    for (int i=0; i<val, i++)
      write(PortB, var[i]);
  }
}
```

val (max)=255

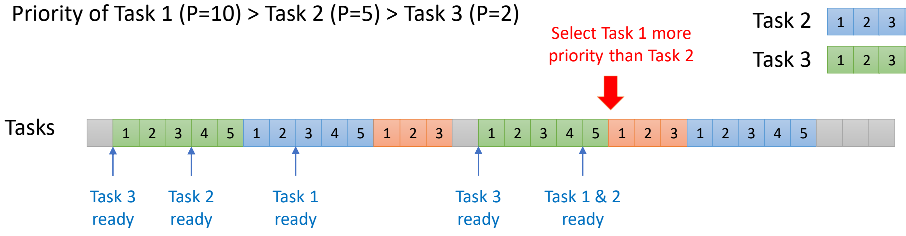
# Non-Preemptive Scheduling

- **Task runs on CPU till it gets terminated or it reaches a blocked state**
- Scheduler does not interrupt a task in running state in middle of the execution
- Cooperative scheduler
- Scheduling: Shortest Job First (SJF), Priority

Example of Priority scheduling




Priority of Task 1 (P=10) > Task 2 (P=5) > Task 3 (P=2)



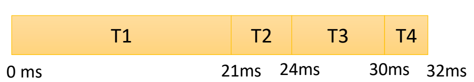



# First Come First Serve (FCFS)
## Non-Preemptive version

- First Come First Serve is just like FIFO (First in First out) Queue
- **Selects task from the head of the queue and new task enters through the tail of the queue**
- Average Waiting Time (AWT)
  - Crucial parameter to judge it's performance
  - *Lower the Average Waiting Time, better the scheduling algorithm*
- Disadvantages
  - Non Preemptive algorithm which means the process priority doesn't matter
  - Not optimal Average Waiting Time
  - Resources utilization in parallel is not possible

| Task | Execution Time (ms) |
|---|---|
| T1 | 21 |
| T2 | 3 |
| T3 | 6 |
| T4 | 2 |

AWT = (0 + 21 + 24 + 30) / 4 = 18,75 ms




Waiting Time

WT(T1) = 0 ms

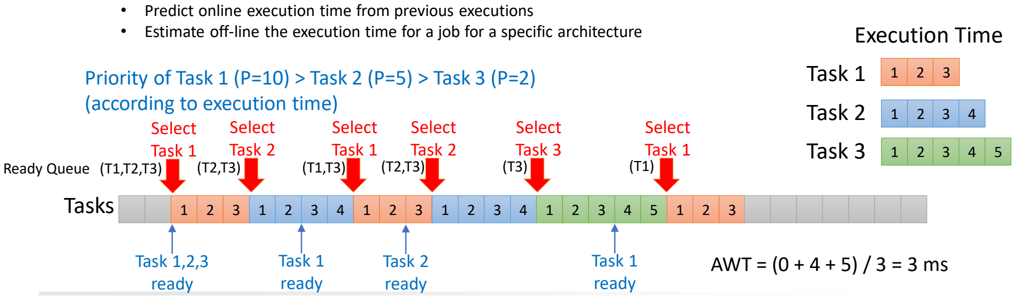
WT(T2) = 21 ms

WT(T3) = 24 ms

WT(T4) = 30 ms

# Shortest Job First (SJF)
## Non-Preemptive version

- **Select the ready task with the smallest execution time to execute next**
- Also known as Shortest job next (SJN)
- Greedy Algorithm
- SJF has the advantage of having minimum average waiting time among all scheduling algorithms
- May cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging
- Practically infeasible as Operating System may not know execution time
  - Predict online execution time from previous executions
  - Estimate off-line the execution time for a job for a specific architecture

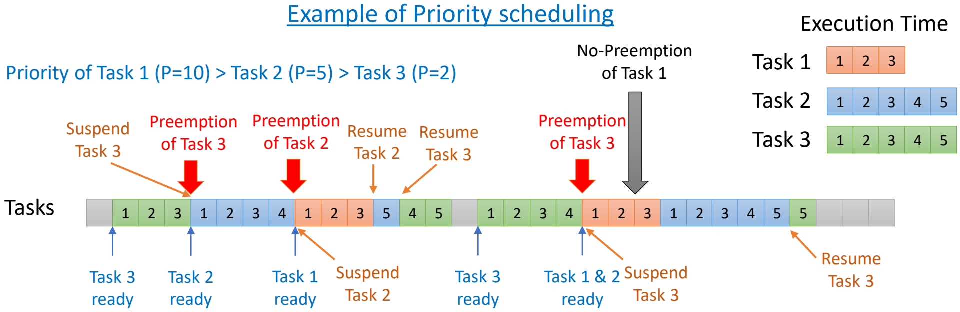





# Preemptive Scheduling

- **Scheduler can interrupt a task in running state in middle of the execution**
- Task switches
  - from running state to ready state
  - from blocked state to ready state (but not in running state due to a higher priority task)
- Scheduling: Round Robin (RR), Priority or Rate-Monotonic (RM) , Earliest Deadline First (EDF), Shortest Remaining Time First (SRTF)

Example of Priority scheduling



# Rate-Monotonic scheduling Schedulability

- Priority assignment algorithm
- Properties of tasks
  - No resource sharing = independent tasks
  - Deadlines are exactly equal to periods
  - Static priorities: **tasks with shorter periods/deadlines are given higher priorities**
  - Task with the highest static priority that is runnable immediately preempts all other tasks
  - Context switch times and other thread operations are free and have no impact on the model
- Schedulability
  - Liu & Layland (1973) proved a feasible schedule that will always meet deadlines exists if the CPU utilization is below a specific bound

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

n : number of task
Ci : Computation time or Execution time
Ti : Release period

  - If n tends towards infinity

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

  - RM Scheduling can meet all of the deadlines if CPU utilization is less than 69.32%.
  - 30.7% of the CPU can be used to lower-priority non real-time tasks.



# Shortest Remaining Time First (SRTF) scheduling

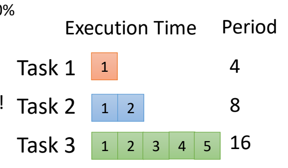
- Preemptive version of Shortest Job First
- **Task with the smallest amount of time remaining until completion is selected to execute**
- Decisions are made when
  - Task finished
  - New task added
- Advantages
  - Short tasks are handled very quickly
  - Little overhead through decision steps

# Earliest Deadline First (EDF) scheduling

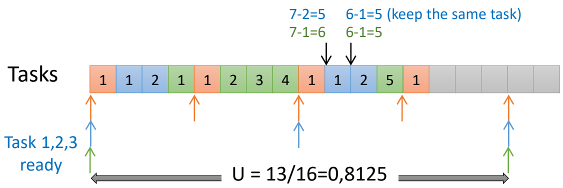
- Dynamic priority scheduling algorithm and preemptive scheduling
- **Search for the task closest to its deadline whenever a scheduling event occurs** (task finishes, new task released …)
- Schedulability
  - If the deadlines equal to their periods, EDF scheduling has a utilization bound of 100%

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \le 1,$$

n : number of task
Ci : Computation times or Execution times
Ti : Release periods (equal to relative deadlines)




- EDF scheduling can guarantee all the deadlines in the system at higher loading !
- Difficult to implement because the relative deadline is not precise to compute
- EDF scheduling is not commonly use in industrial real-time computer systems




Schedulability :

U = (1/4+2/8+5/16)

U = (4/**16**+4/**16**+5/**16**)

U = 0,8125 < 1

Least common multiple of the period : **16**

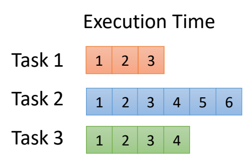


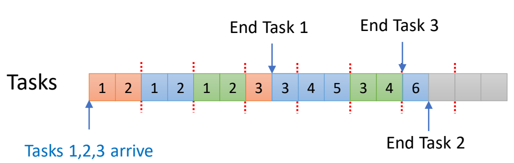
# Time Sharing - Round Robin Scheduling

- CPU scheduling algorithm and preemptive scheduling
- **Each task is assigned a fixed time slot in a cyclic way**
- Time slot or Quantum
  - Short quantum : overhead of context switching
  - Long quantum : long response time (infinite quantum = FIFO algorithm)
  - Set a quantum (statistic behavior) when 80% of tasks finish their CPU usage before the end of the quantum

Arrival time = 0
Order : Task 1, Task 2, Task 3

Time slot = 2 (or quantum)



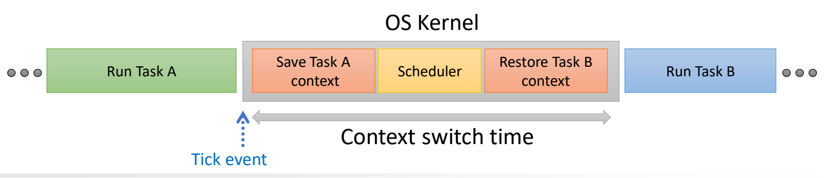



# Context Switch



## Context Switch

- A task must be switch out of the CPU to perform another tasks
- Context switch time depends on the architecture
- Save and restore the context
  - On Stack or/and Task Control Block (TCB)
  - Some processors can internally backup and restore the process context

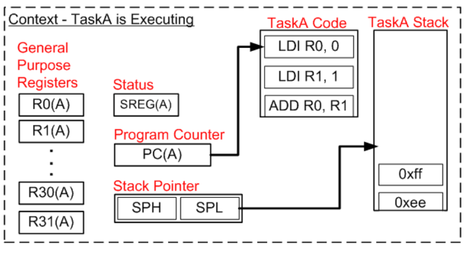


# Context Switch – FreeRTOS Example

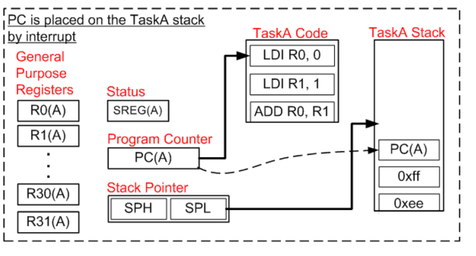
## Steps 1 & 2

- Task A is running
- Assume that Task B has previously been suspended (context has already been stored on the Task B stack)
- When the interrupt occurs, the CPU automatically places the current program counter onto the stack before jumping to the start of the RTOS tick ISR.

Step 1




Step 2







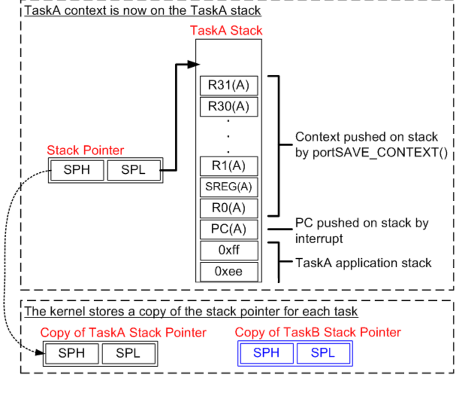
# Context Switch – FreeRTOS Example

## Step 3

- IG_OUTPUT_COMPARE1A() and vPortYieldFromTick() are naked functions
- portSAVE_CONTEXT() pushes the entire CPU execution context onto the stack of Task A and store a copy of the stack pointer

```
/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A(void) {
    vPortYieldFromTick();
    asm volatile ("reti");
}

void vPortYieldFromTick(void) {
    portSAVE_CONTEXT();

    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();

    asm volatile ("ret");
}
```

Step 3

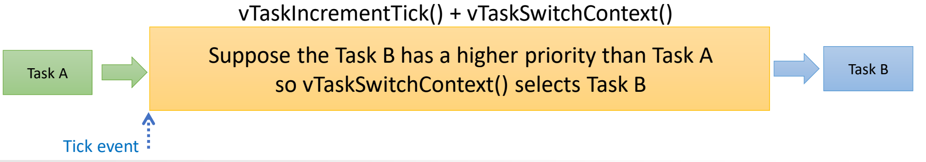


# Context Switch – FreeRTOS Example
## Step 4

- vTaskIncrementTick()
  - Takes care of all aspects related to tick timer
  - Updating the current time
  - Checking whether some task timeouts have expired …
- vTaskSwitchContext()
  - Looks at which tasks are in the ready state
  - Selects the higher priority tasks depending on scheduling policy

```
void vPortYieldFromTick(void) {
    portSAVE_CONTEXT();

    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();

    asm volatile ("ret");
}
```

Step 4





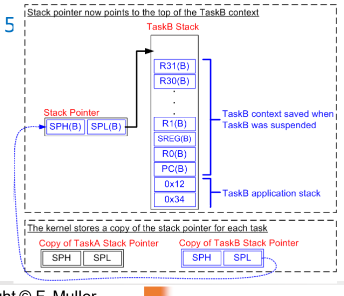


# Context Switch – FreeRTOS Example
## Steps 5 & 6

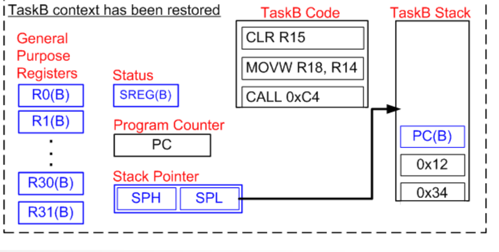
- The Task B context must be restored
- Retrieve the Task B stack pointer from the copy taken when Task B was suspended
- Task B stack pointer is loaded into the processor stack pointer
- CPU stack points to the top of the Task B context
- Restoring the Task B context from its stack into the appropriate processor registers (only the program counter remains on the stack.).

```
void vPortYieldFromTick(void) {
    portSAVE_CONTEXT();

    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();

    asm volatile ("ret");
}
```

Step 5



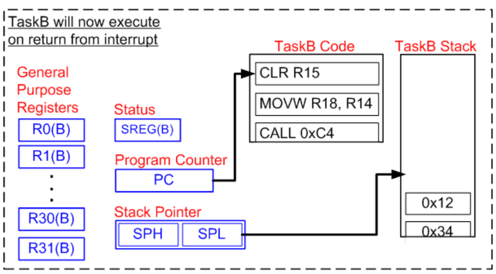

Step 6



# Context Switch – FreeRTOS Example
## Step 7

- vPortYieldFromTick() returns to SIG_OUTPUT_COMPARE1A()
- *RETI* instruction assumes the next value on the stack is a return address placed onto the stack when the interrupt occurred
- Task B is running !

Step 7




```
/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A(void) {
    vPortYieldFromTick();
    asm volatile ("reti");
}

void vPortYieldFromTick(void) {
    portSAVE_CONTEXT();

    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();

    asm volatile ("ret");
}
```



# Critical Section & Shared Resources

# What is a Critical Section ?

- Also called *critical region*
- Useful for concurrent programming
- Protected accesses to a shared resource
  - Data structures
  - Peripheral devices
  - Network connections
- Critical section may be protected by different mechanisms
  - Semaphore
  - Mutex (mutual exclusion)
  - Dedicated functions generally by interrupt masking



# Critical Section – P()/V() operations

- Operate on a mutex-semaphore variable
- P operation
  - Dutch word **P**roberen (to attempt)
  - For *wait operation*
  - Require a resource and if not available waits for it
- V operation
  - Dutch word **V**erhogen (to increase)
  - For *signal passing operation*
  - Pass to the OS that the resource is now free to other tasks
- How implemented ?
  - Test & Set instruction
  - Fetch & Add instruction
- Use of standard POSIX 1003.1b, IEEE standard

# Critical Section – P()/V() operations
# Test & Set instruction

- Implement read/write/test on a shared variable
- Used for boolean semaphore
- Test & Set instruction
  - Used to both test and (conditionally) write to a memory location as part of a single atomic (i.e. non-interruptible) operation
- No other process may begin another test-and-set until the first process's test-and-set is finished
- CPU itself may offer a test-and-set instruction
- http://en.wikipedia.org/wiki/Test-and-set

Example

```
boolean lock = false;

void P(void) {
    /* Active wait */
    while (TestAndSet(lock));
}

void V(void) {
    lock = false;
}
```



# Critical Section – P()/V() operations
# Fetch & Add instruction

- Implement read/write/test on a shared resource
- Used for mutex and counter semaphore
- Fetch & Set instruction
  - atomically (i.e. non-interruptible) modifies the contents of a memory location by a specified value
  - increment the value at address ADDR by VALUE and return the original VALUE at ADDR
- When this instruction is executed by one process in a concurrent system, no other process will ever see an intermediate result
- CPU itself may offer a fetch & Set instruction
- http://en.wikipedia.org/wiki/Fetch-and-add

```
int FetchAndAdd(int* addr) {
    int value = *addr;          } Let suppose that
    *addr = value + 1;          } it is an atomic operation
    return value;
}
struct PVType {
    int number;
    int turn;
}
```

Example of Mutex

```
void PVInit(PVType* pv) {
    pv->number = 0;
    pv->turn = 0;
}
void P(PVType* pv) {
    int turn = FetchAndAdd(&pv.number);
    /* Active wait */
    while (pv.turn != turn);
}
void V((PVType* pv) {
    FetchAndAdd(&pv.turn);
}
```

# Shared Resources - Peterson's algorithm

- Formulated by Gary L. Peterson in 1981
- Concurrent programming algorithm for mutual exclusion
- Based on active wait (polling mode)
- Share a single-use resource without conflict from two or more processes

Example for 2 processes

Process N°0

```
enter[0] = true;
turn = 1;
while (enter[1] && turn == 1) {};
// critical section

data[...] = ...

// end of critical section
enter[0] = false;
```

Shared resource

int data[]

Process N°1

```
enter[1] = true;
turn = 0;
while (enter[0] && turn == 0) {};
// critical section

data[...] = ...

// end of critical section
enter[1] = false;
```



# Shared Resources - Priority Inversion

- **A high priority task is indirectly preempted by a lower priority task**
- Inverting the relative priorities of the two tasks !
- Does not take care of priority inversion can have disastrous effects
  - Response to emergency situations
  - System can be blocked ...
- Limitation of priority inversion
  - Allow access to critical sections only to tasks with the same priority
  - Use specific semaphores : priority inheritance semaphores or ceiling priority semaphores.

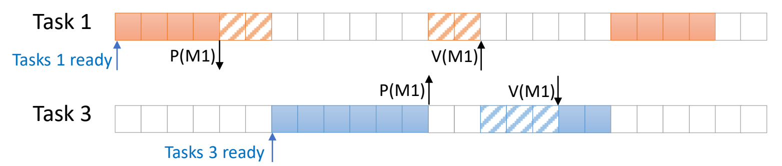
# Shared Resources - Priority Inversion Two tasks, no choice !

- Task 1 acquires the mutex M1
- An event wakes Task 3 which preempts Task 1
- Task 3 tries to get the mutex; since it is already acquired by Task 1, Task 3 is suspended.
- Task 1 runs and releases the mutex 1
- Task 3 preempts Task 1

Task 1 was performed first instead of task 3 because of the shared resource !

P(Sx) : Start of the critical section of mutex x   V(Sx) : End of the critical section of mutex x   Resource used by task i

Priority(Task 3) > Priority (Task 1)





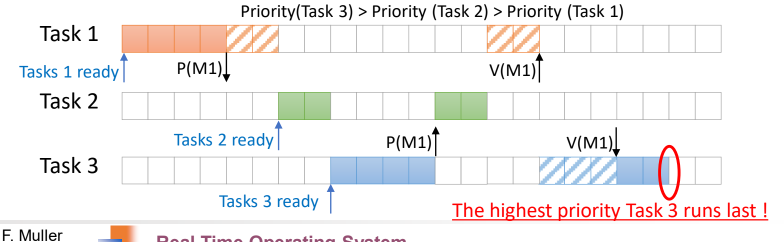


# Shared Resources - Priority Inversion Three tasks, problem !

- Task 1 acquires the mutex M1
- An event wakes Task 2 which preempts Task 1
- Task 3 becomes in ready stage and preempts Task 2
- Task 3 tries to get the mutex. Since it is already acquired by task 1, Task 3 is suspended and Task 2 runs till ended
- Task 1 runs and releases the mutex 1
- Task 3 can run now till ended

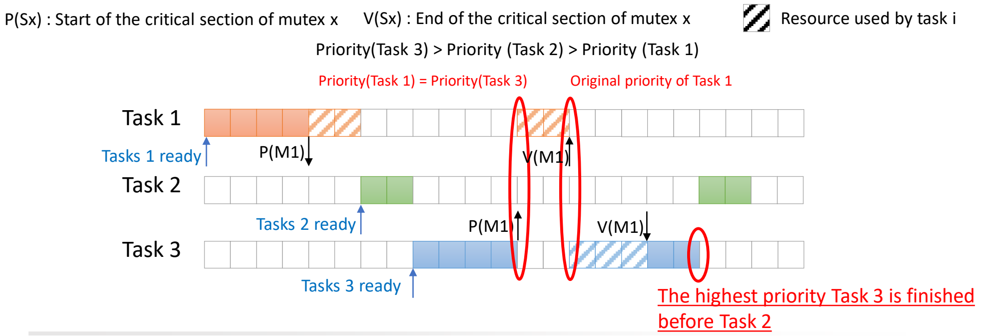
P(Sx) : Start of the critical section of mutex x   V(Sx) : End of the critical section of mutex x   Resource used by task i

Priority(Task 3) > Priority (Task 2) > Priority (Task 1)



# Shared Resources - Priority Inversion Three tasks, Basic Priority inheritance

- Priority inheritance
  - Priority inheritance raises the priority of the blocking task to the blocked one
  - Once the semaphore is released, the blocked task returns to its original priority






# Shared Resources - Priority Inversion Basic Priority inheritance

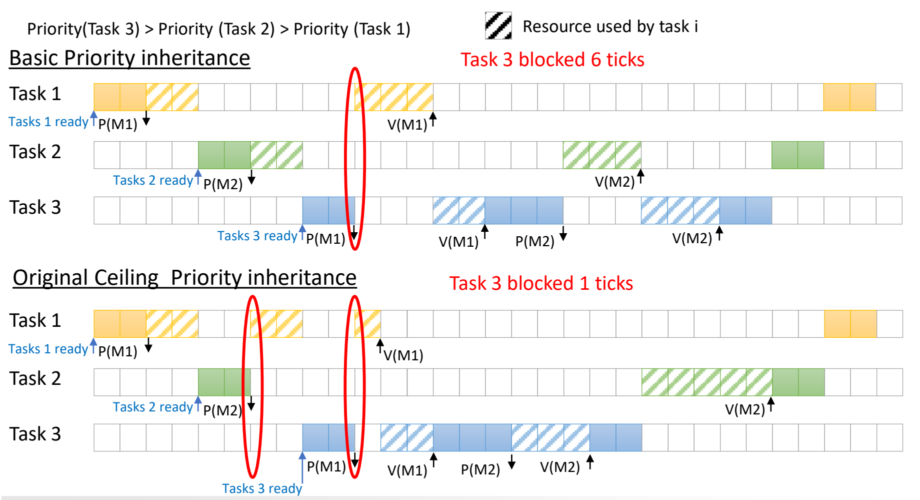
- Advantage
  - Bounded Priority inversion
  - Reasonable Run-time performance
- Disadvantage
  - Potential deadlocks
  - Chain-blocking – many preemptions

# Shared Resources - Priority Inversion
# OCPP - Original Ceiling Priority Protocol

- **A Task 1's priority is raised when a higher-priority Task 2 tries to acquire a resource that Task 1 has locked**
- The task's priority is then raised to the priority ceiling of the resource, ensuring that Task 1 quickly finishes its critical section, unlocking the resource
- A task is only allowed to lock a resource if its dynamic priority is higher than the priority ceilings of all resources locked by other tasks. Otherwise the task becomes blocked, waiting for the resource
- OCPP changes priority only if an actual block has occurred



# Shared Resources - Priority Inversion
# Original Ceiling Priority Protocol



# Shared Resources - Priority Inversion Immediate Ceiling Priority Protocol

- Also called Highest Locker's priority Protocol (HLP)
- **A task's priority is immediately raised when it locks a resource**
- The task's priority is set to the priority ceiling of the resource, thus no task that may lock the resource is able to get scheduled.
- A task can only lock a resource if its dynamic priority is higher than the priority ceilings of all resources locked by other tasks
- ICPP changes immediately priority




# Shared Resources - Priority Inversion Immediate Ceiling Priority Protocol