NTNU
Department of Engineering Cybernetics

# TTK4147 Real-time Systems

04 Scheduling

11.09.2018

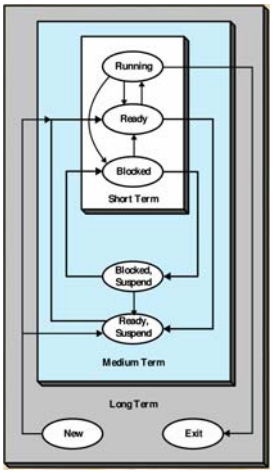


## Today

- Scheduling
  - Different Time-Scales
  - Scheduling Criteria
  - Priority Based Scheduling
  - Different Scheduling Policies
- Real-time Scheduling
  - Cyclic Executive
  - Fixed Priority Scheduling
  - Rate Monotonic
  - Is Scheduling Possible?
  - Response-time Analysis

## What is Scheduling?

- A strategy for deciding which process that are allowed to run on a processor over time.
  - Could also be scheduling of other resources, e.g. I/O resources.
- Common to break it down into three separate timescales:
  - Long-term scheduling is the decision of which processes that are allowed to be added to the pool of processes to be handled.
  - Medium-term scheduling is to decide which processes that are fully or partially swapped in to main memory.
  - Short-term scheduling is to decide which of the currently ready processes that the CPU should execute.



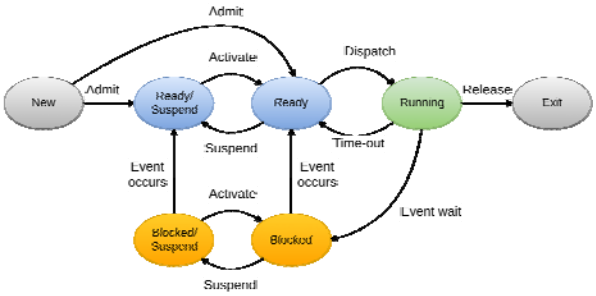

3 NTNU

## Short-Term Scheduling

- Main focus on this lecture.
- The short-term scheduler is also called *dispatcher.*
- Main objective is to allocate processor time to optimize certain aspects of system behavior

4 NTNU

## Dispatcher




- The dispatcher can be called due to:
  - A clock interrupt which decides that the allowable time slice for the process has been used.
  - I/O interrupt means that the OS has to reevaluate which process should run.
  - A trap (error) will exit the current process.
  - The process perform an action that will cause it to wait/be blocked.



## Scheduling Criteria, User Oriented

- Performance related / quantitative
  - Turnaround time
    This is the interval of time between the submission of a process and its completion.
  - Response time
    The time from the submission of a request until the start of response.
  - Deadlines
    When process completion deadlines can be specified, as many deadlines as possible should be met.
- Non-performance related / qualitative
  - Predictability
    A given job should run in about the same amount of time and at about the same cost regardless of the load on the system.

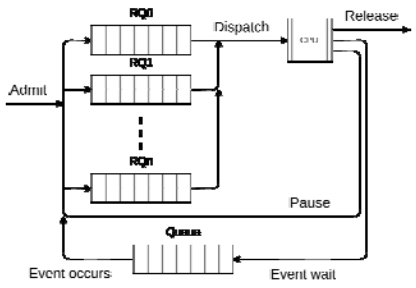
## Scheduling Criteria, System Oriented

- Performance related / quantitative
  - Throughput
    Maximize the number of processes completed per unit of time. This is a measure of how much work is being performed.
  - Processor utilization
    This is the percentage of time that the processor is busy.
- Non-performance related / qualitative
  - Fairness
    Processes should be treated the same, and no process should suffer starvation.
  - Enforcing priorities
    When processes are assigned priorities, the scheduling policy should favor higher-priority processes.
  - Balancing resources
    Balanced use of system resources. If a resource is overused, then processes that do not need it should be favored.



## Priority Based Scheduling

- Assign priorities to each process.
  - Dynamic or static.
- Each priority gets its own queue, and the dispatcher will check higher priority queues first.
- Lower priority tasks might suffer from "starvation".

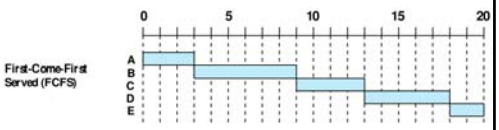


# Different Scheduling Policies



## First-Come-First-Served
**(First-In-First-Out)**

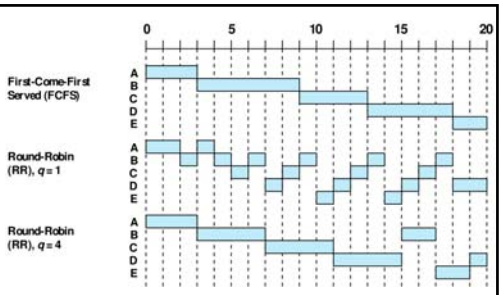
- Dispatcher chooses the process that has been in the ready queue the longest.
- Favor long processes.
- Favor processes that mostly use CPU over processes that mostly use I/O.
- Often used in combination with priorities.
- Non-preemptive, run-to-completion

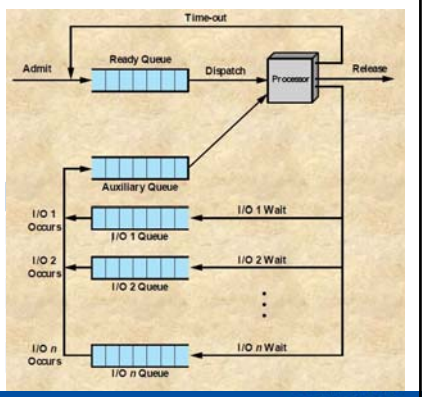



| Process | Arrival Time | Service Time |
|---|---|---|
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

## Round Robin

- Each process is given a time slice.
  - Regular clock interrupts (Pre-emptive).
  - Place running process in ready queue
  - Process that has been the longest in the ready queue is executed next.
- Favor short processes.
- Favor processes that mostly use CPU over processes that mostly use I/O.
- Variations can mitigate this problem.
  - Virtual Round-robin.

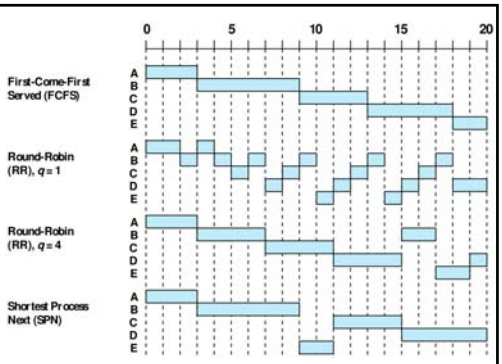





## Shortest Process Next

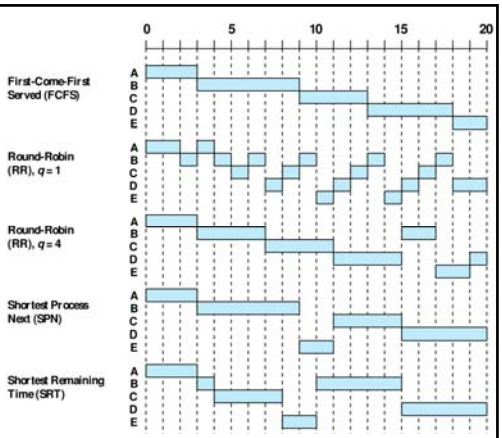
- Dispatcher chooses the process in the queue with the shortest executing time.
- Non-preemptive
- Requires knowledge of executing time.
- Favor short processes.
- Long processes might never get a chance to run (starvation).




| Process | Arrival Time | Service Time |
|---|---|---|
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

## Shortest Remaining Time

- Preemptive version of SPN.
- Requires knowledge of executing time.
- Favor short processes.
- Long processes might never get a chance to run (starvation).




| Process | Arrival Time | Service Time |
|---|---|---|
| A | 0 | 3 |
| B | 2 | 6 |
| C | 4 | 4 |
| D | 6 | 5 |
| E | 8 | 2 |

13 NTNU

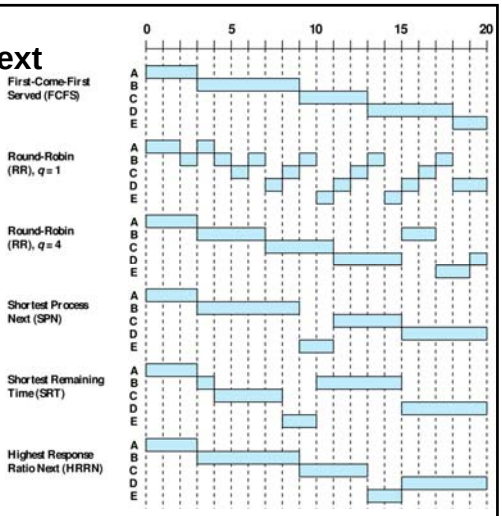
## Highest Response Ratio Next

- Dispatcher choose process from execution time and time waited in queue.
  - Ratio calculated for each process.

$$R = \frac{w + s}{s}$$

R = response ratio
w = time waited in queue
s = expected service time

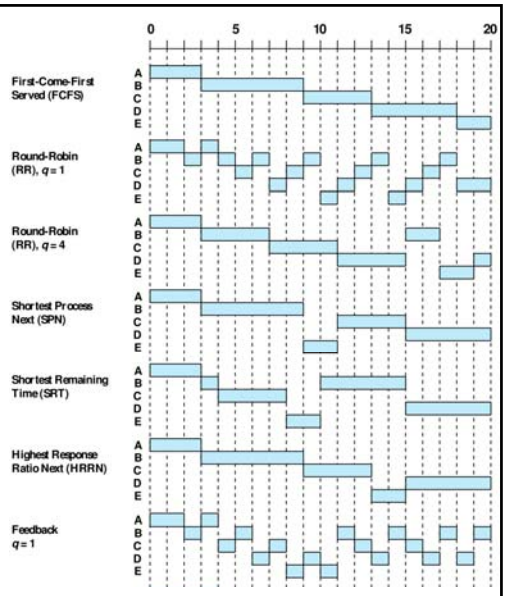
  - Highest R get dispatched.
- Requires knowledge of executing time.
- Good balance.
- Non-preemptive




14 NTNU

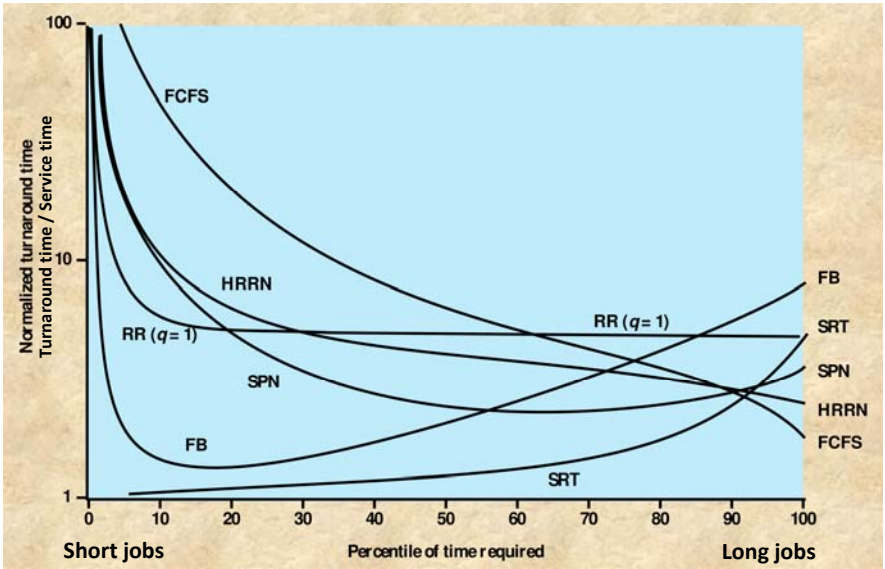
## Feedback

- Decrease priority each time process is preempted. (Preemptive)
- Newer and shorter processes favored over old and long.
- Does not require knowledge of execution time.
- Could cause starvation of long processes.
- Starvation can be mitigated by allowing longer time slices for lower-priority processes.






## Simulation Results



## Real-time Scheduling

- In real-time scheduling, its all about the deadlines...
- Can your system be scheduled so all tasks can be performed without missing any deadlines??
- Back to some of the concept described in the introduction lecture...




17 NTNU

## Simple Task Model

- We start with a simple task model:
  - The application is assumed to consist of a fixed set of tasks
  - All tasks are periodic, with known periods
  - The tasks are completely independent of each other
  - All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost)
  - All tasks have a deadline equal to their period (that is, each task must complete before it is next released)
  - All tasks have a fixed worst-case execution time

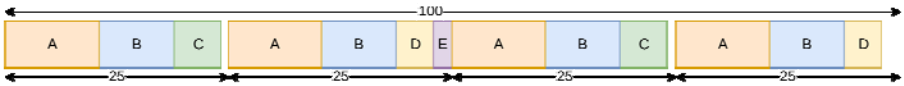
| Task | Period (T) | Time (C) |
|---|---|---|
| A | 25 | 10 |
| B | 25 | 8 |
| C | 50 | 5 |
| D | 50 | 4 |
| E | 100 | 2 |

18 NTNU

## Cyclic Executive

- Requires a fixed set of periodic tasks.
- Pre-plan and "hardcode" when each of the task should run
  - Static scheduling
- A major cycle is divided into several minor cycles
  - Each task has to be placed within these cycles.
  - No Processes/threads, is just a sequence of functions/procedure calls.
  - Easy to share data, concurrent access / race conditions not possible.

| Task | Period (T) | Time (C) |
|---|---|---|
| A | 25 | 10 |
| B | 25 | 8 |
| C | 50 | 5 |
| D | 50 | 4 |
| E | 100 | 2 |

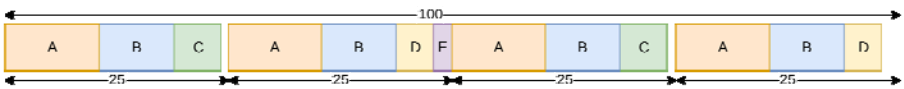



19 NTNU

## Cyclic Executive Problems

- Difficult to have sporadic tasks.
- Difficult to have long tasks.
- Difficult to "construct" the cyclic executive.
- In larger systems, it is likely that there will be very large major cycles, with very many minor cycles.

| Task | Period (T) | Time (C) |
|---|---|---|
| A | 25 | 10 |
| B | 25 | 8 |
| C | 50 | 5 |
| D | 50 | 4 |
| E | 100 | 2 |




20 NTNU

## Fixed-Priority Scheduling (FPS)

- This is the most widely used real-time scheduling approach.
  - If preemptive (which it probably should be), then the highest priority available task will always run.
- The priority of each task is found before run-time.
  - Priorities are not changed (static).
- Priorities should be assigned based on what is best for the whole system, not on the relative importance of the task.
- Optimal priority assignment for periodic, independent tasks is ***rate monotonic***.
  - The shorter the period, the higher priority.

| Task | Period (T) | Time (C) | Priority (P) | |
|---|---|---|---|---|
| A | 50 | 12 | 1 | |
| B | 40 | 10 | 2 | |
| C | 30 | 10 | 3 | (Highest) |



## Is Scheduling Possible?

- Scheduling tests.
  - A sufficient test means that if the test is passed, the tasks are certain to meet their deadlines.
  - A necessary test means that if the test is failed, the tasks will miss their deadlines.
  - An exact test is both necessary and sufficient.
- *Utilization-based scheduability test* is a simple test of fixed priority scheduling with rate monotonic.
  - Sufficient, but not necessary.
  - Require simple task model.

- U = Utilization
- N = Number of tasks
- $T_i$ = Period of task i
- $C_i$ = Computational time of task i

Utilization criteria

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\left(2^{1/N} - 1\right)$$

## Utilization limits

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

| Number of Tasks | Utilization Limit |
|---|---|
| 1 | 100,0% |
| 2 | 82,8% |
| 3 | 78,0% |
| 4 | 75,7% |
| 5 | 74,3% |
| 10 | 71,8% |



## Example of Utilization (A)

- Use Utilization on previous example:

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) = C/T |
|---|---|---|---|---|
| A | 50 | 12 | 1 | 24% |
| B | 40 | 10 | 2 | 25% |
| C | 30 | 10 | 3 | 33% |
| | | | Sum | 82% |

- 82% is NOT below the limit for 3 tasks (78%)
  - In this case the utilization test does not really tell us anything.
- Time line can be used to draw the tasks, and evaluate if they are schedulable.
  - If all tasks start at time 0, it is sufficient if all tasks are able to make their first deadline.

## Example of Utilization (B)

- Use Utilization on another example:

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) = C/T |
|---|---|---|---|---|
| A | 80 | 32 | 1 | 40% |
| B | 40 | 5 | 2 | 12,5% |
| C | 16 | 4 | 3 | 25% |
| | | | Sum | 77,5% |

- 77,5% is below the limit for 3 tasks (78%)
  - The utilization test guarantees that these tasks are scheduable.

| Number of Tasks | Utilization Limit |
|---|---|
| 1 | 100,0% |
| 2 | 82,8% |
| 3 | 78,0% |

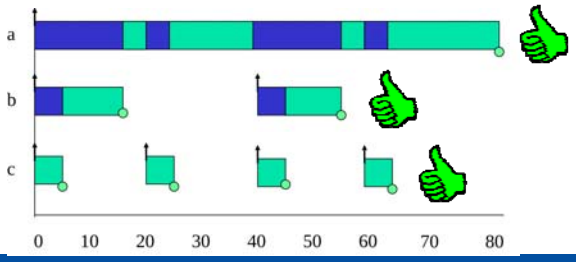


## Example of Utilization (C)

- Use Utilization on previous example:

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) |
|---|---|---|---|---|
| A | 80 | 40 | 1 | 50% |
| B | 40 | 10 | 2 | 25% |
| C | 20 | 5 | 3 | 25% |
| | | | Sum | 100% |

- 100% is NOT below the limit for 3 tasks (78%)
  - In this case the utilization test does not really tell us anything.
- However….



## Alternatives (improvements) for the Utilization Test 1 of 2

1. Tasks that have periods that are multiples of each other are considered to be in the same *family*. Counting *families of tasks* as tasks.
   - In a task set of 3 tasks, where 2 tasks are in the same family. Then the 2 task limit (82,8%) instead of the 3 task limit (78,0%) is used.

Example (task set C)

- All tasks are in family (20, 40, 80)
- Not necessary to use 3 task limit (78.0%), instead the 1 task limit (100%).

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) |
|---|---|---|---|---|
| A | 80 | 40 | 1 | 50% |
| B | 40 | 10 | 2 | 25% |
| C | 20 | 5 | 3 | 25% |
| | | | Sum | 100% |

| Number of Tasks | Utilization Limit |
|---|---|
| 1 | 100,0% |
| 2 | 82,8% |
| 3 | 78,0% |

27 NTNU

## Alternatives (improvements) for the Utilization Test 2 of 2

2. Alternative Formula:

$$\prod_{i=1}^{N}\left(\frac{C_i}{T_i}+1\right)\leq 2$$

Example:

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) = C/T |
|---|---|---|---|---|
| A | 76 | 32 | 1 | 42,1% |
| B | 40 | 5 | 2 | 12,5% |
| C | 16 | 4 | 3 | 25% |
| | | | Sum | 79,6% |

1,421 * 1,125 * 1,25 = 1.998 < 2

| Number of Tasks | Utilization Limit |
|---|---|
| 1 | 100,0% |
| 2 | 82,8% |
| 3 | 78,0% |

28 NTNU

## Response-time Analysis 1 of 2

- Is a method for calculating the worst case execution time of a task, when there are higher priority tasks that interfere.
  - $R_i$ : Response time of task i
  - $D_i$ : Deadline of task i
  - $C_i$ : The computational time of task i
  - $I_i$ : Interference from higher priority tasks

$$R_i \leq D_i$$
$$R_i = C_i + I_i$$

- If no interference, i.e. $task_i$ has highest priority / is the only task:

$$R_i = C_i$$

29 NTNU

## Response-time Analysis 2 of 2

- How can we calculate $R_i$??
  - $C_i$ we already know
  - $I_i$ is a bit more difficult
- This following test is **Exact** (sufficient and necessary)

30 NTNU

## Calculating $R_i$

- To calculate $R_i$, we need to get $I_i$.
  - We have to find out the maximum number of times EACH of the higher priority tasks can be released during $R_i$.
  - We come up with this formula, where $T_j$ is the period of task j (which has higher priority than task i)

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$ ( = Number of times task i is interrupted)

- A release of task j will interfere with the computation of task i.
  - The computation time of task j determines the length of this interference.
  - The maximum interference from task j on task will then be:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j$$ ( = The time task j takes from task i)

31 NTNU

## Formula for Response Time Analysis

- The total interference from all higher priority tasks will then be:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$ ( = The time all tasks with higher priority takes from task i)

- We then have a formula for the response time of task i ($R_i$):

$$R_i = C_i + I_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Oh great... you need $R_i$ to calculate $R_i$...

32 NTNU

## How to use the Formula

- The response time has to be calculated using a recurrence relationship.

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- Suitable to solve with an algorithm:

```
for i in 1..N loop -- for each process in turn
  n := 0
  w_i^n := C_i
loop
    calculate new w_i^(n+1)
    if w_i^(n+1) = w_i^n then
      R_i = w_i^n
      exit value found
    end if
    if w_i^(n+1) > T_i then
      exit value not found
    end if
    n := n + 1
  end loop
end loop
```



## Response Time Analysis on Paper

- You should not be surprised if you have to do this on the exam.
  - You should probably be surprised if you don't.

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) |
|---|---|---|---|---|
| A | 7 | 3 | 3 | 42,9% |
| B | 12 | 3 | 2 | 25,0% |
| C | 20 | 5 | 1 | 25,0% |
| | | | Sum | 92,9% |

- Utilization analysis can not say whether this is scheduable or not.

| Number of Tasks | Utilization Limit |
|---|---|
| 1 | 100,0% |
| 2 | 82,8% |
| 3 | 78,0% |

## Response Time Analysis on Paper

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

| | T | C | P |
|---|---|---|---|
| A | 7 | 3 | 3 |
| B | 12 | 3 | 2 |
| C | 20 | 5 | 1 |

- For Task A
  - Highest priority task, nothing can interfere.

$$R_a = 3 \quad <= 7$$

- For Task B
  - Start with the assumption that there is no interference.
  - Set $w^0$ into equation
  - Set $w^1$ into equation
  - No change in w, must be correct....

$$w_b^0 = 3$$

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6$$

$$w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6$$

$$R_b = 6 \quad <= 12$$



## Response Time Analysis on Paper

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

| | T | C | P |
|---|---|---|---|
| A | 7 | 3 | 3 |
| B | 12 | 3 | 2 |
| C | 20 | 5 | 1 |

- For Task C
  - Start with the assumption that there is no interference.
  - Set $w^0$ into equation
  - Set $w^1$ into equation
  - Set $w^2$ into equation
  - Set $w^3$ into equation
  - Set $w^4$ into equation
  - No change in w, must be correct....

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

$$R_c = 20 \quad <= 20$$

## REMEMBER!

| | Forelesning | | | Øvinger |
|---|---|---|---|---|
| Uke | Tirsdag 8:15 - 10:00 F6 Gamle fysikk | Fredag 12:00 - 15:00 U33 Handelshøyskolen | | |
| 34 | 01 Introduction | | | 0 - Basic tools (voluntary) |
| 35 | 02 Hardware and operation systems | | | 1 - Memory, errors and time |
| 36 | 03 Multiprogramming and Memory Management | Exercise lecture (Ex2+Ex3) | | 2 - Concurrency |
| 37 | 04 Scheduling | 05 Deadlocks, mixed criticallity and freertos | | 3 -AVR UC3 |
| 38 | 06 Priority Inversion | Exercise lecture (Ex4+Ex5) | | 4 -AVR FreeRTOS |
| 39 | 07 Linux | | | 5 - BRTT and Linux |
| 40 | | Exercise lecture (Ex6+Ex7) | | 6 - BRTT and Xenomai |
| 41 | * | * | | 7 - Xenomai |
| 42 | | Exercise lecture (Ex8 + Miniproject | | 8 - Cross platform with NGW100 |
| 43 | 08 More Scheduling | | | Miniproject |
| 44 | 09 QNX | | | Miniproject |
| 45 | 10 Windows | | | Theory Exercise |
| 46 | 11 VxWorks and Android | Exercise lecture (Ex9+Ex10) | | 9 - QNX IPC |
| 47 | 12 Repedition | | | 10 QNX resource manager |



NTNU
Department of Engineering Cybernetics

# TTK4147 Real-time Systems

05 Deadlocks, Mixed Criticality, and FreeRTOS

14.09.2018



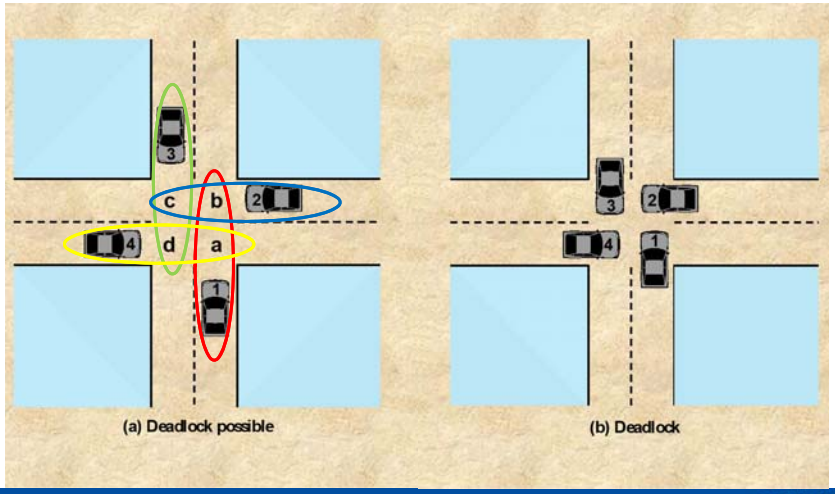
## Today

- Deadlocks
- Mixed Criticality
- FreeRTOS
  - Task Model
  - Scheduling
  - Inter-process Communication in FreeRTOS

## Deadlock and Starvation

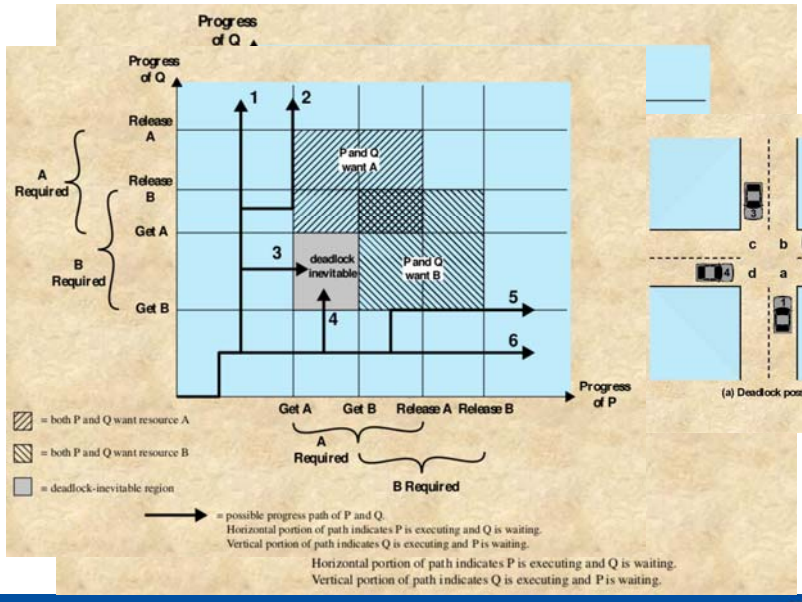
What is «Deadlock»?

- Deadlock is when multiple processes are blocking each other.
  - Most cases this will be due to shared resources that several of the processes want to access.




3 NTNU

## Example of Deadlock







4 NTNU

## Resource Types

- Reusable resources.
  - Variable in memory.
  - File in filesystem.
  - Available memory

| Process 1 | Process 2 |
|---|---|
| Request 80 kB | Request 70 kB |
| ... | ... |
| Request 60 kB | Request 80 kB |
| ... | ... |
| Free 60 kB | Free 80 kB |
| ... | ... |
| Free 80 kB | Free 70 kB |

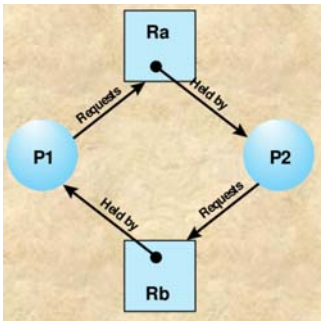
- Consumable resources.
  - Produced then consumed, typically by different processes.

| Process 1 | Process 2 |
|---|---|
| Receive from P2 | Receive from P1 |
| ... | ... |
| Send to P2 | Send to P1 |

5 NTNU

## Conditions for Deadlock

- Three conditions have to be present for a deadlock to be possible.
  1. Mutual Exclusion
     - Only one processes can use a resource at a time (i.e. a printer).
  2. Hold and wait
     - A process can hold one resource (A) while waiting for another resource (B), while another process holds one resource (B) while waiting for another (A).
  3. No preemption
     - No resource can (forcibly) be removed / released from the processes holding it.




- For a deadlock to **actually** take place, a circular wait has to happen.
  - Circular wait is not a requirement for a deadlock, but a possible consequence when all three previous conditions are met.

6 NTNU

## Deadlock Prevention

Prevent deadlock by removing the necessary conditions.

- Indirect, by removing one of the three conditions that make deadlocks possible.
  - No mutual exclusive
    - In most cases it is not practical to remove this condition.
  - No hold and wait
    - Can be removed by forcing a thread to request all resources at the same time.
      - Unnecessary locking of resources.
      - Not always possible to know which resources are needed in the future.
  - Allow preemption
    - Allow the OS to force a thread to release its resource.
      - Not always possible, resource can be left in an unpredictable state.
- Direct, by removing the possibility of circular wait.
  - Define an order in which the resources must be requested.
    If all processes request resources in the same order, a circular wait is impossible.
    - Unnecessary locking of resources.



## Deadlock Detection

- The deadlock detection method has no limitations for how processes should access resources.
  - The system will instead periodically check if any deadlock has occurred.
- If a deadlock is detected, the system has to be recovered.
- Recovery strategies.

  Process related:

  a) Abort all deadlocked processes.

  b) Return each deadlocked process to a previous checkpoint.
     - Will not always be possible.
     - Also possible that the deadlock will recur.

  c) Abort deadlock threads one by one until the deadlock no longer exist. Order of aborting can be decided by what is considered the smallest cost.

  Resource related:

  a) Preempt resources until the deadlock no longer exist.

## Deadlock Avoidance

More attractive than deadlock prevention, and it does not allow for deadlock to occur as deadlock detection does.

a) Process initiation rejection.
   - Prevent a process to start if it is likely to cause a deadlock.
   - Check that the maximum use of resources from all existing processes and the process that wants to start are less that the available resources.
   - Not optimal, as it assumes the worst.

b) Resource allocation rejection.
   - Only allow a process to request a resource if that does not lead to an *unsafe state* that could cause deadlock.
   - Not optimal, as it assumes the worst.



## Worst-case Execution Time and Mixed Criticality

- Two topics were I will go a little bit beyond what it covered in Burns and Wellings chapter 11.
- I do not have any "textbook" sources on this.
  - The information in these slides is the syllabus.
  - The slides are based on references given below, but these are NOT required reading.

    (Easy to search for if you want to.)

References:

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., ... & Mueller, F. (2008). "The worst-case execution-time problem—overview of methods and survey of tools." *ACM Transactions on Embedded Computing Systems*

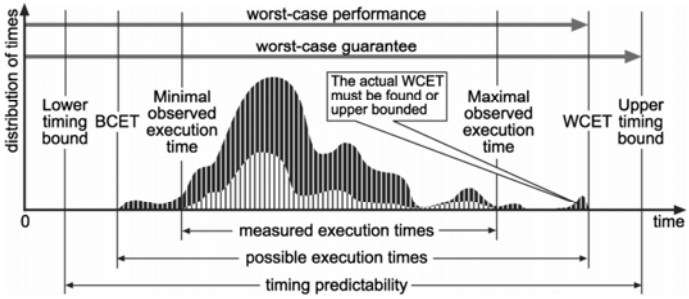
Vestal, S. (2007). "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance." *In Real-Time Systems Symposium*

Burns, Alan, and Rob Davis. (2013) "Mixed criticality systems-a review." *Department of Computer Science, University of York*, Tech. Rep

## Different Levels of Worst-case Execution Time

- For real-time tasks we are concerned with the worst-case execution time (WCET).
  - How do we know this for sure?
  - Is this one a fixed «value» or a fixed gap?

- Different WCET levels from most pessimistic to most optimistic:
  - Highest theoretical, provable execution time, found with static analysis.
    The actual value, but unfortunately difficult/impossible to get.
  - The worst execution time found when testing x number of times.






## Mixed Criticality

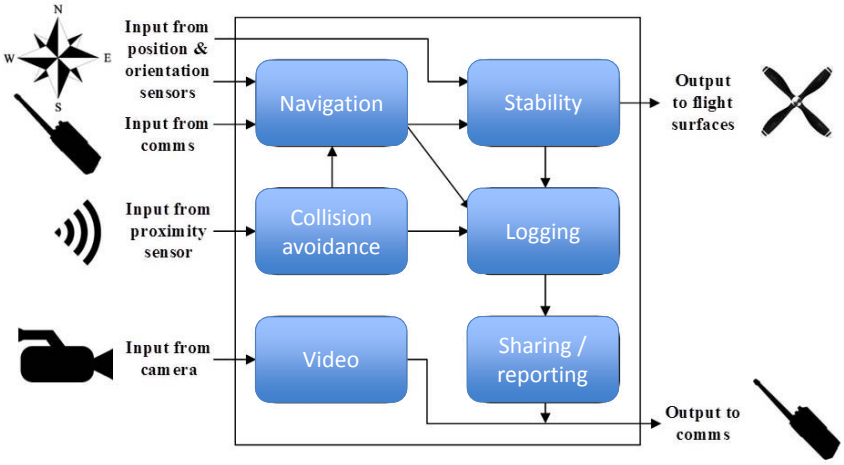
(Similar to "hard and soft task" as described in Burns and Wellings chapter 11.)

- Criticality is typically used as a description of the importance that a task is performed within a deadline.
  - In the simplest form binary; Critical or non-critical.
    However, possibly more than two levels (although dual-criticallity is often used).

- Example of three level mixed criticality:
  - Safety critical
  - Mission critical
  - Non critical

## Mixed Criticalty Unmanned Aerial Vehicle (UAV) Example:

Analyze the specific case, what is Safety critical, Mission critical, Non critical




13 NTNU

## Dual Criticality UAV Example

| Criticality | Max Observed WCET | Max Theoretical WCET |
|---|---|---|
| Safety | 60% | 90% |
| Mission | 40% | - |
| Total | 100% | 90% |

**Response time analysis:**

- You will always be able to do the safety critical task
- If safety critical task takes more than the max observed load, then there might not be enough time for mission critical tasks.

**Verification / approval of the code:**

- The safety critical tasks might be subject to 3rd party approval (which may not find max observed WCET = 60% acceptable).
- The mission critical tasks will typically be approved internally.
  - Acceptable to have a *lower standard*, since your product will not be illegal if these tasks fail.

14 NTNU

## Tripartite mixed Criticality UAV Example

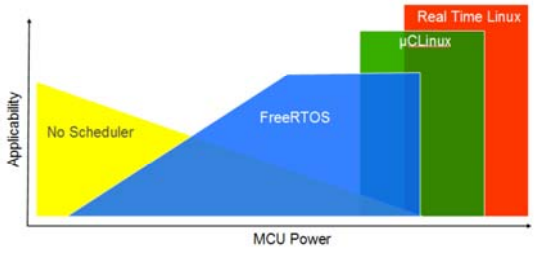
| Criticality | Mean Execution Time | Max Observed WCET | Max Theoretical WCET |
|---|---|---|---|
| Safety | 50% | 60% | 90% |
| Mission | 30% | 40% | - |
| Non | 20% | - | - |
| Total | 100% | 100% | 90% |

- If all tasks perform at mean execution time, you will even be able to do the non critical tasks.
- Otherwise as before



## Small OS: FreeRTOS

- Very small operating system
  - Consists of three main code files.
  - Plus architecture specific code.
- Usable for small micro-controllers.
  - Ported to more than 30 embedded system architectures.
  - From a few KB ROM.
  - ~200 bytes RAM and upwards.
- Very different from a "typical" operating system.



## FreeRTOS Licensing

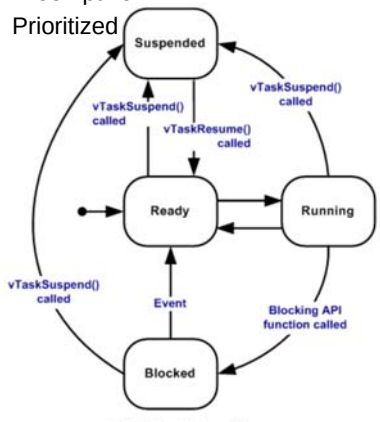
- Open and free
  - Modified GPL
  - Changes in the kernel must be made public available
  - Must document that FreeRTOS is used.
- OpenRTOS: Same code-base, commercial license.
- SafeRTOS: Safety-certified version, commercial license



## FreeRTOS Tasks

**Tasks**

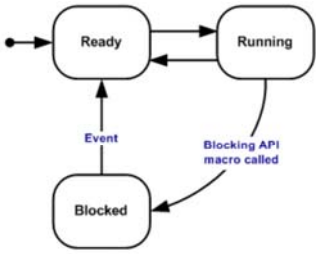
- Regular Process.
- Have its own stack.
- Preemptive.
- Prioritized




Valid task state transitions

**Co-Routines**

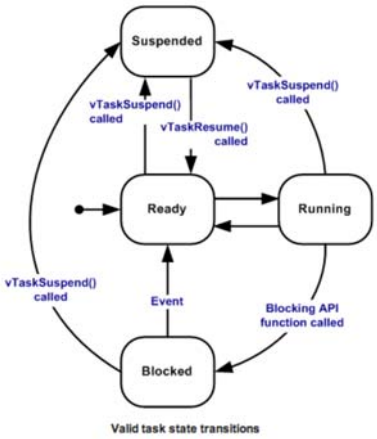
- Intended for use on small processors that have severe RAM constraints.
- Share a single stack.




Valid co-routine state transitions

## FreeRTOS Task Model

- Running:
  Actively executing and using the processor.
- Ready:
  Able to execute, but not because a task of equal or higher priority is in the Running state.
- Blocked
  Waiting for a temporal or external event. E.g., queue and semaphore events, or calling vTaskDelay() to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- Suspended
  Only enter via vTaskSuspend(), depart via xTaskResume() API calls.



Valid task state transitions



## The Idle Task

- The idle task is created automatically when the scheduler is started.
- It frees memory allocated by the RTOS to tasks that have since been deleted.
  - Will only run when nothing else is needed, thus this can take some time.
- There is an idle task hook, which can do some work at each idle interval

## FreeRTOS Scheduling

- Can be either co-operative or preemptive.
- Co-operative means non-preemptive
  → Each task runs until:
  - Its finished
  - Its blocked
  - It voluntarily “yields” its running status, allowing the scheduler to pick the highest priority task.
- The scheduler is activated by a timer interrupt every "clock tick".
  - If preemptive, the scheduler will check for any ready tasks that should be run, and starts a context switch.
    (If non-preemptive, the timer interrupt routine only increments the tick count.)
- The scheduler will select one of the ready tasks with the highest priority.
  - Equal priority tasks are scheduled with Round-Robin.



## Inter-process communication in FreeRTOS

- FreeRTOS have implemented common inter-process communication methods.
  - Semaphores (both binary and counting).
  - Mutexes (priority inheritance and recursive).
  - Queues (same as message passing).
- FreeRTOS also have their "own" *task notification* concept.
  - Supposed to be 45% faster and use less RAM than binary semaphore.
  - Can be used as light-weight alternatives to other inter-process communication methods, with some limitations.

## Queue Overview

- Queues are the primary form of inter-task communications.
- They can send messages between tasks as well as between interrupts and tasks.
- Supports appending data to the back of a queue, or sending data to the head of a queue.
- Queues can hold arbitrary items of a fixed size.
- The size of each item and the capacity of the queue are defined when the queue is created.
- Items are enqueued by copy, not reference.



## Queues and Blocking

- Access to queues is either blocking or non-blocking. (Read / Write)
- The scheduler blocks tasks when they attempt to read from or write to a queue that is either empty of full, respectively.
- If the xTicksToWait variable is zero and the queue is empty (full), the task does not block. Otherwise, the task will block for xTicksToWait scheduler ticks or until an event on the queue frees up the resource.
- This includes attempts to obtain semaphores, since they are special cases of queues.

NTNU
Department of Engineering Cybernetics

# TTK4147 Real-time Systems

06 Priority Inversion

18.09.2018



## Priority Inversion

- Priority inversion is when a high priority task has to wait for a lower-priority task.
- Typical example of this is:

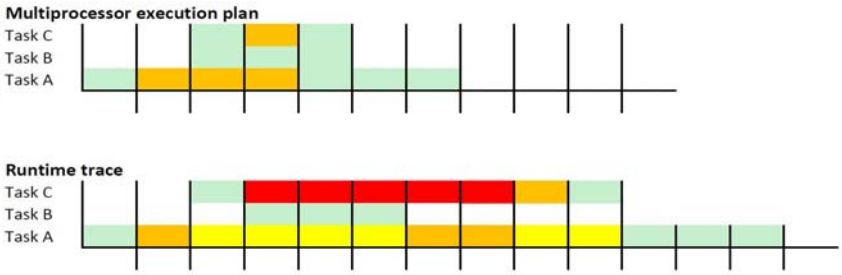


  - A low-priority task locks a mutex, which will remain locked while it access a shared resource.
  - A high-priority task need the same shared resource, and waits for the mutex.

**This is actually OK**

## Priority Inversion

- But, a more subtle effect is that the high-priority task, in practice, get the low-priority.



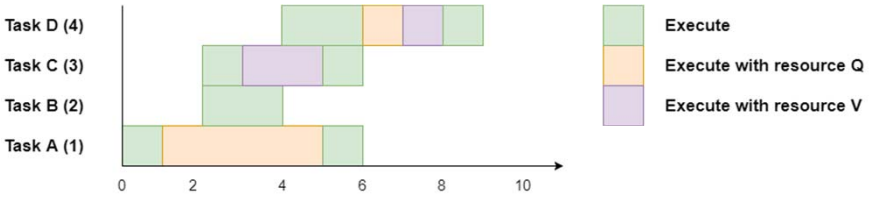

- If you have a *medium-priority* task, then it can prevent the low-priority task from completing its work and returning the mutex.
- This will prevent the high-priority task from running.

**This is NOT OK**

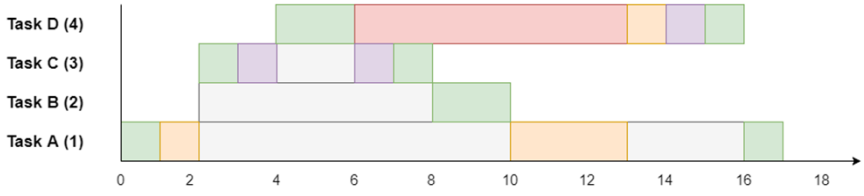
3 NTNU

## Priority Inversion

Multiprocessor execution plan:




No Inheritance

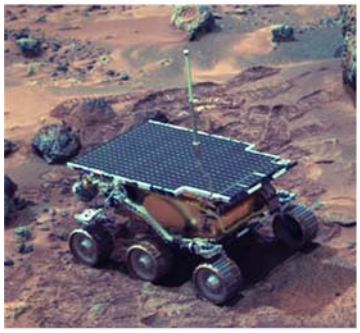



4 NTNU

## A much publicized example: The Mars Path-finder Incident

- A problem due to priority inversion nearly caused the lose of the Mars Path-finder mission.
- As a shared bus got heavily loaded critical data was not been transferred.
- Time-out on this data was used as an indication of failure and lead to re-boot (watchdog).
- Solution was a patch that turned on *priority inheritance*, this solved the problem.

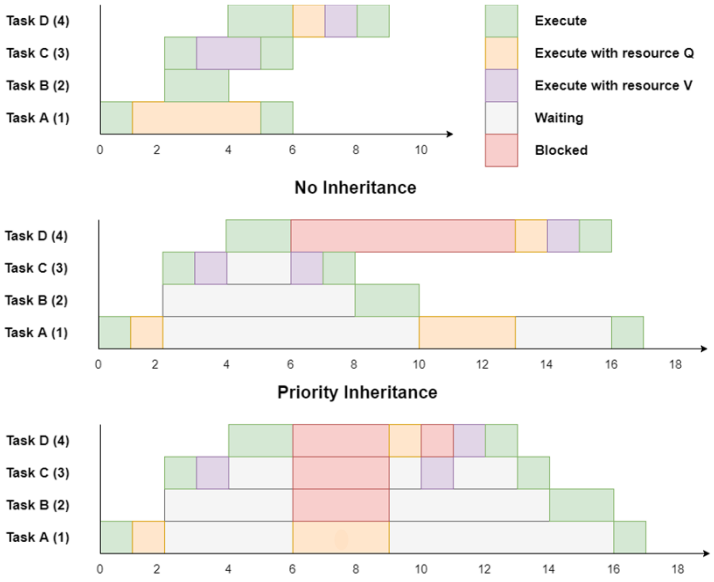
(What is priority inheritance?)



Fun Facts:
Price: 25 mill USD
Weight 11,5 kg
Speed: 0,024 km/h
Mission: 85 days

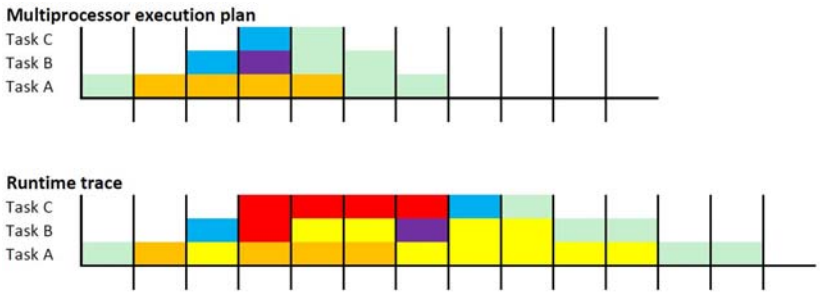
5 NTNU

## Priority Inheritance




6 NTNU

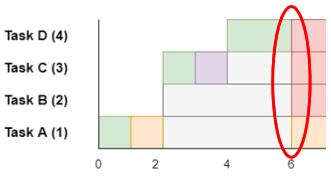
## Multi step Priority Inheritance






## Priority Ceiling Protocols

- Priority inheritance reduce the effect of priority inversion of high-priority tasks.
  - Is it possible to reduce it further?
  - What about deadlocks?
- A low priority task that access a shared resource does not get a priority inheritance effect before a high-priority task tries to access the same resource.
- Would it be better to immediately increase the priority so it will complete its use of the shared resource as fast as possible?
- Will cover two priority ceiling protocols:
  - Original ceiling priority protocol.
  - Immediate ceiling priority protocol.

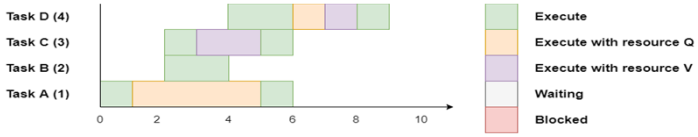


## Immediate Ceiling Priority Protocol (ICPP)

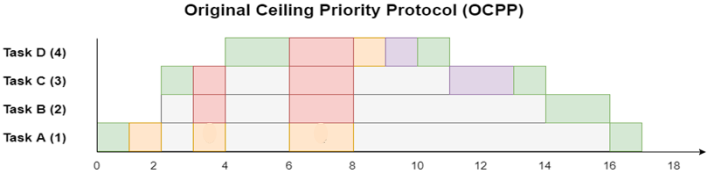

- Each shared resource is assigned the priority of the highest priority task that use it.
  - When a task is allowed to access a shared resource, it will inherit its priority.
  - This means that when a task starts using a resource, it is guaranteed to be uninterrupted until it is done with the resource.
  - Minimize the time a resource is *locked*.
- A task got the priority max of its own static and the ceiling priority of any resources it has locked


9 NTNU

## Original Ceiling Priority Protocol (OCPP)

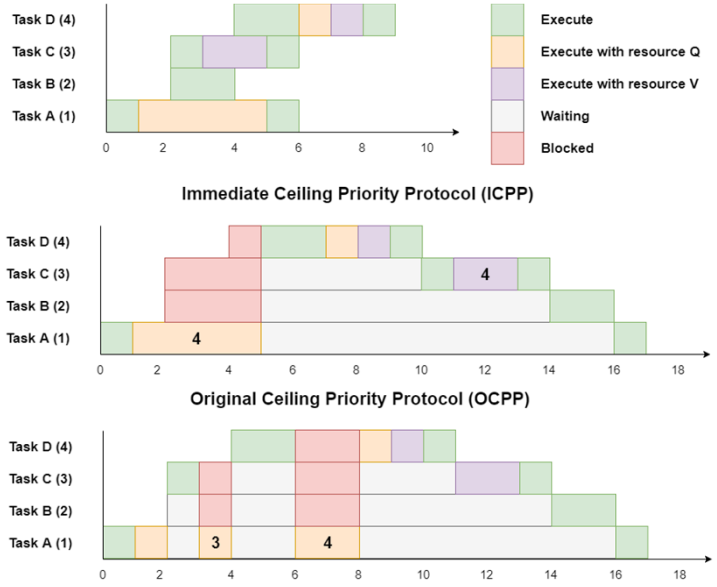



- Each task has a static default priority assigned.
- Each resource has a static ceiling value defined equal to the maximum priority of the tasks that use it.
- A task has a dynamic priority which is the maximum of its own static priority and that of any higher priority tasks it blocks.
- A task can only lock a resource if the resource's dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself).




10 NTNU

## OCPP vs ICPP



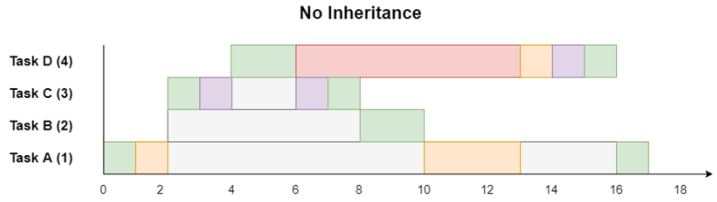



## OCPP vs ICPP

- ICCP is easier to implement than the original (OCPP), which has to keep track of what is blocking what.
- ICPP leads to less context switches as blocking is prior to first execution
- ICPP requires more priority movements as this happens with all resource usage
- OCPP changes priority only if an actual block has occurred

## Priority Inversion Blocking Time

- The general message is that priority inversion could cause higher priority tasks to be delayed by lower priority ones.
  - Priority inheritance and ceiling can reduce this problem.
- But in real-time computing, you of course know that there is one thing that is more important than speed...

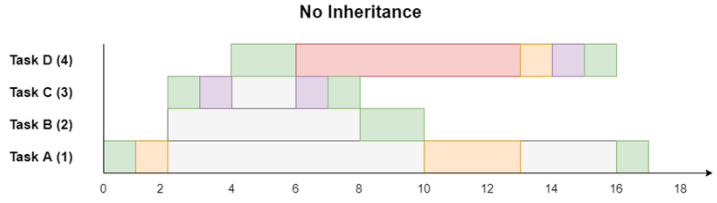



13 NTNU

## Priority Inversion Blocking Time

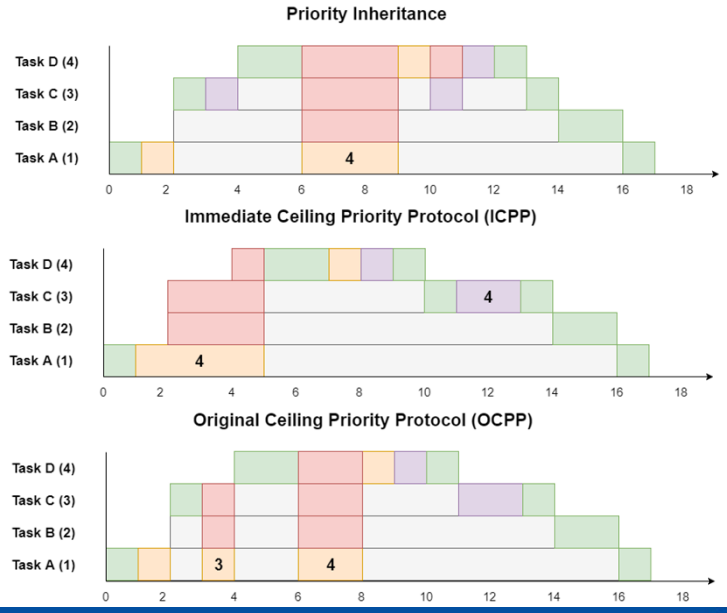
- **Predictability!!!**
  - The problem with the example below is not that task D is blocked.
  - Its that the worst case blocking time is really bad.
- With priority inheritance, it is easier to calculate the worst case blocking time.

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$




14 NTNU

## Priority Inheritance, OCPP, ICPP





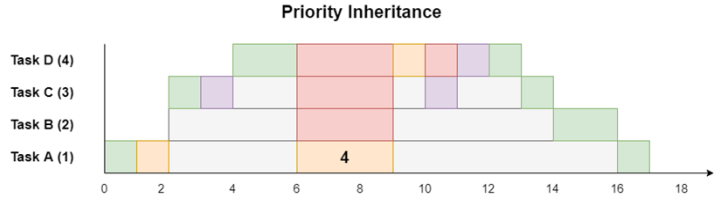

## Worst Case Blocking Time with Inheritance

- A task can only be blocked once due to each shared resource.
  - In example task D is first blocked due to Q then due to V.
- How long will a higher priority task be blocked due to a shared resource?
  - Block cause low priority task to get high priority.
  - Max block time is the max time the low priority task use the resource.

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

usage(k, i) = 1 if resource k is used by at least one task with a priority < Pi, and at least one task with a priority >= Pi

C(k) is the WCET of the k critical section



## Adding Block Time to Response (Turnaround) Time Analysis

- The *usage* function:
  - 1 for shared resources that are used by lower and higher (than i) priority tasks.
  - 0 for others.
- The worst case blocking time can be added to the response (turnaround) time expression.
- Can then be used in response (turnaround) time analysis.

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

NB Different

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

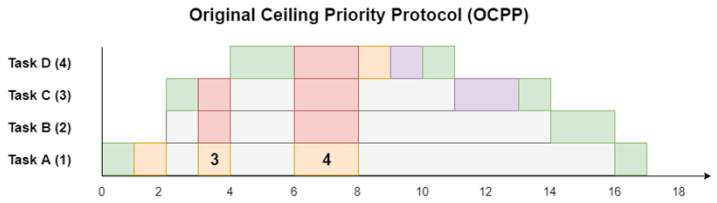


## Worst Case Blocking Time with Ceiling

- When using priority ceiling, a high-priority task can only be blocked ONCE by a lower priority task.
  - Does not matter how many different shared resources or lower-priority tasks that exist.
  - The worst case blocking time is the same for OCPP and ICPP.
  - Compared to priority inheritance, more tasks will be blocked, but max one time each.

$$B_i = \max_{k=1}^{k} usage(k,i)C(k)$$

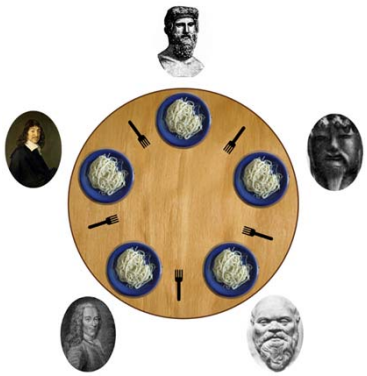
(To be used in $R_i = C_i + B_i + I_i$ )



## Priority Ceiling and Deadlocks

- Priority ceiling effectively prevent deadlocks from "dining philosopher" problems.

- With OCCP:
  - A philosopher is only allowed to take a fork, if no other forks are taken.
  - When he eventually is able to take the fork, no other philosopher will be allowed to take the other one.



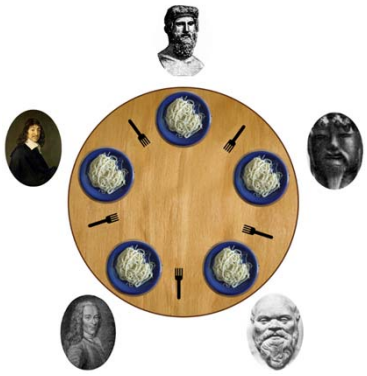
Because:

- A task has a dynamic priority which is the maximum of its own static priority and that of any higher priority tasks it blocks.
- A task can only lock a resource if the resource's dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself).



## Priority Ceiling and Deadlocks

- With ICPP:
  - When a philosopher takes one fork, he will get the priority of the forks.
  - Nothing can then prevent him from taking the other fork and start eating.
  - When he returns the forks, the next one can do the same.



Because:

- Each shared resource is assigned the priority of the highest priority task that use it.
  - When a task is allowed to access a shared resource, it will inherit its priority.
  - This means that when a task starts using a resource, it is guaranteed to be uninterrupted until it is done with the resource.
- A task got the priority max of its own static and the ceiling priority of any resources it has locked

NTNU
Department of Engineering Cybernetics

# TTK4147 Real-time Systems

08 More Scheduling

23.10.2018



## Today

- Deadline Monotonic Priority Ordering
- Sporadic and Aperiodic Tasks
  - How to Schedule
  - Polling Server
  - Deferable Server
  - Sporadic Server
- Extended Task Model
- Earliest Deadline First
- Multiprocessor Scheduling
  - Multicore computer
  - Effect on real-time

## Response Time Analysis Repetition, Fixed Priority Scheduling

(Calculating worst-case response time for each task, check that all WCRT < periods)

$$R_i = C_i + I_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$ ( = Number of times task i is interrupted)

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j$$ ( = The time task j takes from task i)

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) | Response time (R) |
|---|---|---|---|---|---|
| 1 | 7 | 3 | 3 | 42.9% | |
| 2 | 12 | 2 | 2 | 16.7% | |
| 3 | 20 | 5 | 1 | 25.0% | |
| | | | Sum | 84.6% | |

$$w_1^0 = C1 = 3 \implies r_1 = 3$$



## Response Time Analysis Repetition

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) | Response time (R) |
|---|---|---|---|---|---|
| 1 | 7 | 3 | 3 | 42.9% | 3 |
| 2 | 12 | 2 | 2 | 16.7% | |
| 3 | 20 | 5 | 1 | 25.0% | |
| | | | Sum | 84.6% | |

$$w_2^0 = C2 = 2$$

$$w_2^1 = C2 + \left\lceil \frac{w_2^0}{T1} \right\rceil C1 = 2 + \left\lceil \frac{2}{7} \right\rceil 3 = 5$$

$$w_2^2 = C2 + \left\lceil \frac{w_2^1}{T1} \right\rceil C1 = 2 + \left\lceil \frac{5}{7} \right\rceil 3 = 5 \implies r_2 = 5$$

## Response Time Analysis Repetition

| Task | Period (T) | Time (C) | Priority (P) | Utilization (U) | Response time (R) |
|---|---|---|---|---|---|
| 1 | 7 | 3 | 3 | 42.9% | 3 |
| 2 | 12 | 2 | 2 | 16.7% | 5 |
| 3 | 20 | 5 | 1 | 25.0% | |
| | | | Sum | 84.6% | |

$$w_3^0 = C3 = 5$$

$$w_3^1 = C3 + \left\lceil \frac{w_3^0}{T1} \right\rceil C1 + \left\lceil \frac{w_3^0}{T2} \right\rceil C2 = 5 + \lceil \tfrac{5}{7} \rceil 3 + \lceil \tfrac{5}{12} \rceil 2 = 10$$

$$w_3^2 = 5 + \lceil \tfrac{10}{7} \rceil 3 + \lceil \tfrac{10}{12} \rceil 2 = 13$$

$$w_3^3 = 5 + \lceil \tfrac{13}{7} \rceil 3 + \lceil \tfrac{13}{12} \rceil 2 = 15$$

$$w_3^4 = 5 + \lceil \tfrac{15}{7} \rceil 3 + \lceil \tfrac{15}{12} \rceil 2 = 18$$

$$w_3^5 = 5 + \lceil \tfrac{18}{7} \rceil 3 + \lceil \tfrac{18}{12} \rceil 2 = 18 \implies r_3 = 18$$



## Deadline Monotonic Priority Ordering

- Rate monotonic priority is not suitable when deadline is shorter than period (D < T).

| Task | Period (T) | Time (C) | Priority (P) |
|---|---|---|---|
| 1 | 7 | 3 | 3 |
| 2 | 12 | 2 | 2 |
| 3 | 20 | 5 | 1 |

- Deadline monotonic priorities on deadline (D) instead of period (T).
  - NOT THE SAME as Earliest deadline first, which will be described later today.
  - If D = T, then deadline monotonic and rate monotonic are the same.

| Task | Period (T) | Deadline (D) | Time (C) | Priority (P) | Response time (R) |
|---|---|---|---|---|---|
| A | 20 | 5 | 3 | 4 | 3 |
| B | 15 | 7 | 3 | 3 | 6 |
| C | 10 | 10 | 4 | 2 | 10 |
| D | 20 | 20 | 3 | 1 | 20 |

## Deadline Monotonic Example, worst-case time analysis

| Task | Period (T) | Deadline (D) | Time (C) | Priority (P) |
|---|---|---|---|---|
| A | 20 | 5 | 3 | 4 |
| B | 15 | 7 | 3 | 3 |
| C | 10 | 10 | 4 | 2 |
| D | 20 | 20 | 3 | 1 |

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

$$\omega_A^0 = C_A = 3$$

$$\omega_B^0 = C_B = 3$$
$$\omega_B^1 = C_B + \left\lceil \frac{\omega_B^0}{T_A} \right\rceil C_A = 3 + \left\lceil \frac{3}{20} \right\rceil 3 = 6$$
$$\omega_B^2 = C_B + \left\lceil \frac{\omega_B^1}{T_A} \right\rceil C_A = 3 + \left\lceil \frac{6}{20} \right\rceil 3 = 6$$

$$\omega_C^0 = C_C = 4$$
$$\omega_C^1 = C_C + \left\lceil \frac{\omega_C^0}{T_A} \right\rceil C_A + \left\lceil \frac{\omega_C^0}{T_B} \right\rceil C_B$$
$$\omega_C^1 = 4 + \left\lceil \frac{4}{20} \right\rceil 3 + \left\lceil \frac{4}{15} \right\rceil 3 = 10$$
$$\omega_C^2 = C_C + \left\lceil \frac{\omega_C^1}{T_A} \right\rceil C_A + \left\lceil \frac{\omega_C^1}{T_B} \right\rceil C_B$$
$$\omega_C^2 = 4 + \left\lceil \frac{10}{20} \right\rceil 3 + \left\lceil \frac{10}{15} \right\rceil 3 = 10$$



## Deadline Monotonic Example

| Task | Period (T) | Deadline (D) | Time (C) | Priority (P) | Response time (R) |
|---|---|---|---|---|---|
| A | 20 | 5 | 3 | 4 | 3 |
| B | 15 | 7 | 3 | 3 | 6 |
| C | 10 | 10 | 4 | 2 | 10 |
| D | 20 | 20 | 3 | 1 | 20 |

$$\omega_D^0 = C_D = 3$$
$$\omega_D^1 = C_D + \left\lceil \frac{\omega_D^0}{T_A} \right\rceil C_A + \left\lceil \frac{\omega_D^0}{T_B} \right\rceil C_B + \left\lceil \frac{\omega_D^0}{T_C} \right\rceil C_C$$
$$\omega_D^1 = 3 + \left\lceil \frac{3}{20} \right\rceil 3 + \left\lceil \frac{3}{15} \right\rceil 3 + \left\lceil \frac{3}{10} \right\rceil 4 = 13$$
$$\omega_D^2 = C_D + \left\lceil \frac{\omega_D^1}{T_A} \right\rceil C_A + \left\lceil \frac{\omega_D^1}{T_B} \right\rceil C_B + \left\lceil \frac{\omega_D^1}{T_C} \right\rceil C_C$$
$$\omega_D^2 = 3 + \left\lceil \frac{13}{20} \right\rceil 3 + \left\lceil \frac{13}{15} \right\rceil 3 + \left\lceil \frac{13}{10} \right\rceil 4 = 17$$

$$\omega_D^3 = C_D + \left\lceil \frac{\omega_D^2}{T_A} \right\rceil C_A + \left\lceil \frac{\omega_D^2}{T_B} \right\rceil C_B + \left\lceil \frac{\omega_D^2}{T_C} \right\rceil C_C$$
$$\omega_D^3 = 3 + \left\lceil \frac{17}{20} \right\rceil 3 + \left\lceil \frac{17}{15} \right\rceil 3 + \left\lceil \frac{17}{10} \right\rceil 4 = 20$$
$$\omega_D^4 = C_D + \left\lceil \frac{\omega_D^3}{T_A} \right\rceil C_A + \left\lceil \frac{\omega_D^3}{T_B} \right\rceil C_B + \left\lceil \frac{\omega_D^3}{T_C} \right\rceil C_C$$
$$\omega_D^4 = 3 + \left\lceil \frac{20}{20} \right\rceil 3 + \left\lceil \frac{20}{15} \right\rceil 3 + \left\lceil \frac{20}{10} \right\rceil 4 = 20$$

## Sporadic and Aperiodic Tasks

- Aperiodic tasks can be released at any time.
  - Period (T) can be represented as the average release interval.
- Sporadic tasks are aperiodic tasks that have a minimum interval between releases.
  - Period (T) can be represented as the minimum *or* average release interval.
- The simple task model assumes that the deadline of the task should complete before its next release (D = T).
  - This is unreasonable for sporadic and aperiodic tasks.
  - Occur infrequently, but often urgent when they do. (faults, alarms, …)
- Can still use response time analysis.

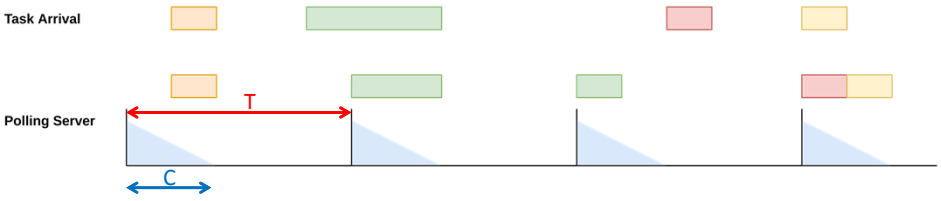


## How to Schedule Aperiodic Tasks

- For sporadic tasks, deadline monotonic is often suitable.
  - Sporadic task is modeled as a task with period equal to the minimum release time.
  - Deadline is the sporadic tasks actual deadline.
  - Not suitable for aperiodic, since we do not have a minimum release time.
- In a rate/deadline monotonic scheme, one could give aperiodic tasks the lowest priority (?).
  - It will only run when the system otherwise is idle.
  - Can suffer from starvation.
  - What about alarms, faults, …?
- To provide better responses for aperiodic tasks, we can use *Execution-time Servers*.
  - Runs as one task / process in the system.
  - May have a high priority
  - Maintain a budget for *clients*, typically aperiodic tasks.

## Periodic/Polling Server (PS)

- Runs periodically, with a time budget.
  - Period = T, Budget = C
  - Can be scheduled by rate-monotonic

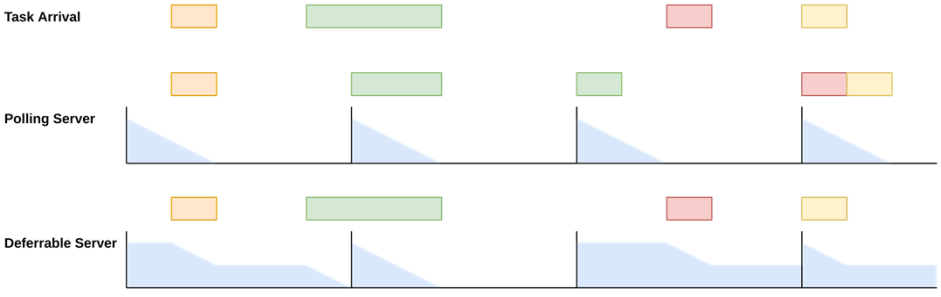



- Each period, there the computation time budget is available for any pending clients.
  - If a client require more work than available within a budget, then it will have to continue next period.
  - If more than one clients are pending, they have to be scheduled somehow within the server.
- The budget has to be used when the server is running or it is wasted.



## Deferrable Server (DS)

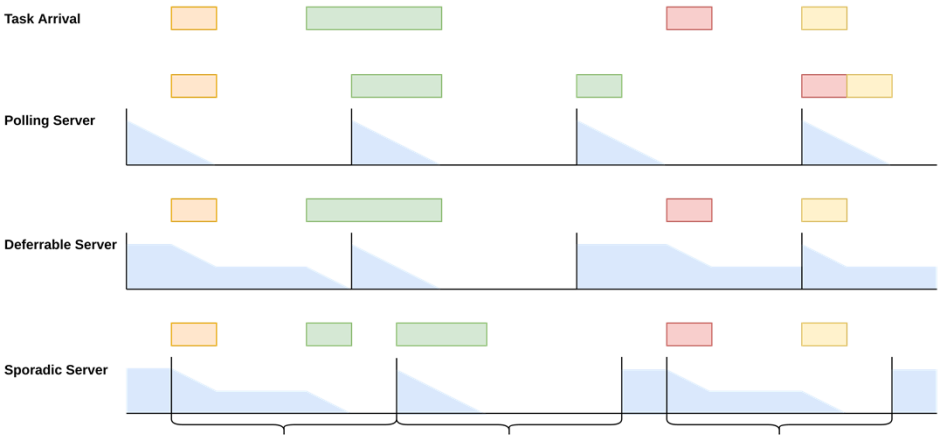
- Differs from polling server in situations where budget and CPU is available but no client.
  - Deferrable server only lose budget when used.




- Deferrable server have better response time for aperiodic tasks.
- Back-to-back, where one budget is used just after another.

## Sporadic Server (SS)

- Differs from deferred server on its replenish policy.
  - Budget will be replenished one time period after it has started to be used.






## Complete Model

- In order to have an accurate model, you can add expressions for:
  - Context switches between the tasks.
  - Interrupt handling when an aperiodic or sporadic task arrives (handling is high priority even if task is not).
  - Cost of scheduling itself.

## An Extendible Task Model

**Starting point: Simple task model:**

- Fixed set of tasks
- Independent tasks
- Deadlines equal to periods
- No internal delays or blocking requests
- Periodic tasks, known periods
- No system overheads
- Fixed worst-case execution times
- All tasks on a single processor

**So far:**

- Deadlines can be less than the period (D<T)
- Sporadic and aperiodic tasks, as well as periodic tasks, can be supported
- Task interactions are possible, with the resulting blocking being factored into the response time equations

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$



## Further model extensions

- Release Jitter
- Arbitrary Deadlines
- Cooperative Scheduling
- Fault Tolerance
- Offsets
- Optimal Priority Assignment
- Execution-time Servers

(and combinations)

## Release Jitter

The maximum variation in a task's release is termed its release jitter, J.

Example:
Release of a task from the end of another task with varying response time

J influence on the time a task can be effected by other tasks and thus, on the response time.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$$



## Arbitrary Deadlines

What if $D_i$ (and hence potentially $R_i$) is > $T_i$?

In the following we assume that the release of a task will be delayed until any previous releases of the same task have completed.

More (q) succeeding releases may have R > T

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

The worst-case response time is then the maximum value found for each q:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q)$$

## Cooperative Scheduling

- True preemptive behavior is not always acceptable for safety-critical systems
- Cooperative scheduling or deferred preemption splits tasks into slots
- Mutual exclusion is via non-preemption
- The use of deferred preemption has two important advantages
  - It increases the schedulability of the system, and it can lead to lower values of C
  - With deferred preemption, no interference can occur during the last slot of execution

- Let the execution time of the final block be $F_i$

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

$$R_i = w_i^n + F_i$$



## Fault Tolerance

- Fault tolerance via either forward or backward error recovery always results in extra computation
- In a real-time fault tolerant system, deadlines should still be met even when a certain level of faults occur
- If F is the number of faults allowed during one execution

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} FC_k^f$$

$C_i^f$ is the extra computation time that results from an error in task i

- If there is a minimum arrival interval $T_f$ for faults

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left( \left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right)$$

## Earliest Deadline First (EDF)

- Not as important or in use as FPS, thus it will not be covered as much.
- But, you should know what it is, and how it can be analyzed.
- Firstly, it is NOT the same as deadline monotonic, so do not confuse the two.



## Earliest Deadline First (EDF) Scheduling

- The task that has the shortest time to deadline will always be running.
  - The priority of each task will change dynamically based on the current deadlines.
- Schedulability can also be tested with utilization, but with another formula than for FPS.
  - Just as utilization for FPS, it also require simple task model.

- Fixed set of tasks
- Independent tasks
- Deadlines equal to periods
- No internal delays or blocking requests
- Periodic tasks, known periods
- No system overheads
- Fixed worst-case execution times
- All tasks on a single processor

## Earliest Deadline First (EDF)

- The task that has the shortest time to deadline will always be running.
  - The priority of each task will change dynamically based on the current deadlines.
- Schedulability can also be tested with utilization, but with another formula than for FPS.
  - Just as utilization for FPS, it also require simple task model.

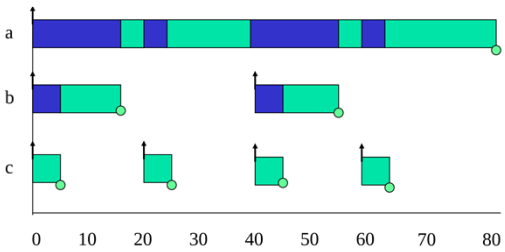
$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

- This means that EDF is able to schedule any task set as long as it takes less time to execute than what is available.
  - In this sense we can say that EDF is optimal.



## EDF Without Simple Task Model

- How can we analyze EDF when the task model is not simple (which basically is all actual systems)? (D <> T)
  - For FPS, we can use response time analysis, which can incorporate delays from shared resources, task switching delays etc.
  - Response time analysis can be used for EDF, but will be much more complicated.
  - Why?



## How to judge schedulability at EDF, D <> T?

**1. Processor Demand Criteria (PDA)**

- Assume that system starts at t=0.
  - All tasks are released at this point.
  - All tasks will arrive at their maximum frequency at any future time.

| Task | T | D | C | C/T |
|---|---|---|---|---|
| a | 4 | 4 | 1 | 0,25 |
| b | 15 | 10 | 3 | 0,20 |
| c | 17 | 14 | 8 | 0,47 |
| | | | | **0,92** |

- It is then possible to calculate all future deadlines.
  - Then evaluate each deadline to check that they are met.

$$h(t) = \sum_{i=1}^{N} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i$$

(The load of the system at *t*)

Requirement:

$$\forall t > 0, h(t) \leq t$$

for t in each deadline



## How to judge schedulability, PDA continue

Problem is to know when to stop.

- In rate and deadline monotonic, we can release all tasks at t=0, and know that this is the worst possible starting point.
  For EDF, we do not know this.
- Necessary to calculate a value L, and if we reach t = L without any deadline misses, then the system is scheduable.
- For larger task sets, L can be very large → many deadlines to check.

## How to judge schedulability at EDF, D <> T?

**2. Quick Processor Demand Test (QPD)**
is a faster way to do this analysis, as number of deadlines to check is reduced.:

- Calculate L as in PDC
- Starts at t = L and works backwards.
- Only checks some critical deadlines.
- Can have as little as 1% the effort as PDA, but equally correct.



## Fixed Priority vs Earliest Deadline First

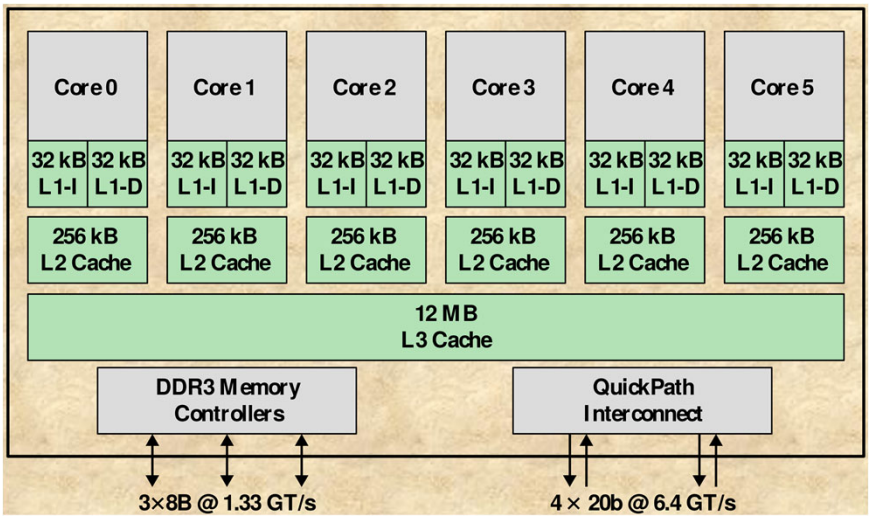
- EDF is guaranteed to work for higher utilization.
- EDF is more difficult to implement, and will have more overhead.
- It is more difficult to incorporate tasks without deadlines in EDF.
- FPS is more predictable if the system gets overloaded.
  - You know that it is the lowest priority task that will fail its deadlines.
- In EDF you can get a cascade effect, where the fact that one task fail its deadline will cause multiple others to do the same.
- Systems scheduable in FPS is also scheduable in EDF.
  Not necessary the other way around

## Multiprocessor Scheduling

- How to schedule on modern CPUs that have multiple cores?
- Probably material for a course by its own, will only cover some basic concept and how this affect real-time
- Are only consider multicore computers.
  - Multiple CPU cores on the same physical CPU chip.
  - Share main memory.
  - Usually share operating system.
- Are not considering three other types of multiprocessor systems mentioned in the book.
  - Computers with multiple CPUs (although it will in practice be the same)
  - Clusters of computers
  - I/O processors



## Multicore CPU



## Assignment of Processes to Cores

- Static assignment of processes to a core.
  - Each process is permanently assigned to a core.
  - One queue of processes in for each core.
  - Minimal scheduling overhead.
  - This can result in one core being idle, while another is very busy.
- A global queue for static assignment of processes to a core.
  - Each process i added to a global queue of processes.
  - When a processor core is idle, it will select the next process and start running it.
  - No core is idle if there as long as there is available work to do.
  - Preempted or blocked processes are likely to run on a different core next, which makes caching less effective.
- Dynamic load balancing.
  - Each process is assigned to a core.
  - A process can be moved between cores depending on load.



## How to Assign Processes

- Master/slave architecture
  - Kernel scheduling functions (part of OS) always run on the same *master* core.
  - *Slave* cores sends requests for services to the master (with the OS)
  - Simple and requires little changes from uniprocessor scheduling.
  - Master can become a bottleneck and a single point of failure.
- Peer architecture
  - Kernel scheduling functions can run on any core.
  - Each core schedule itself with available processes from a pool.
  - Must make sure that not two cores run the same process.

Also variations between these two extremes.

## Multiprogramming on Individual Processes

- Seems like a stupid question, why shouldn't you?

Not only one answer, but usually:

- If you have many cores, it might not be important that all are utilized fully.
- If a process can get full access of a processor core of its own, many issues become simpler.
  - Basically what VxWorks provide with CPU reservation.
  - Can be used in real-time, by giving important processes its own core.



## Real-time on Multicore

- Real-time and multicore is difficult, since you introduce another potential uncertainty.
  - To move a thread from one core to another will cause a longer (and more unpredictable) delay than normal scheduling.
  - Should at least be able to know that a thread runs on a specific core.
- Some industries, e.g. aviation, might not allow multicore computers in safety critical systems.
- But it can have beneficial applications:
  - Run (all or some) hard real-time tasks on a dedicated core.
  - Run all non real-time tasks on a separate core.