Scheduling Theory

Real-Time Systems, Lecture 12

Martina Maggio
23 February 2016
Lund University, Department of Automatic Control

Content

[Real-Time Control System: Chapter 8]

1. Introduction

2. Execution Time Estimation

3. Scheduling Approaches
3.1 Static Cyclic Scheduling
3.2 Fixed Priority Scheduling
3.3 Earliest Deadline Scheduling
3.4 Reservation Based Scheduling

Introduction

Scheduling Theory

Goal: guarantee that a set of tasks sharing resources (CPU) meet their deadlines.

Notation: *events* occurr (they require computations, like interrupts) and can be distinguished in aperiodic (sporadic) and periodic; a task executes a piece of code in response to an event, the *worst-case execution time (WCET)* is an upper bound to the amount of CPU time the task execution requires (when the task is alone in the system); a *deadline* is the maximum allowed time for the task completion.

Scheduling is the act of choosing which event to process at a given time (which task to execute at a given time).

Schedulability Analysis

For hard real-time systems the deadlines must always be met. An offline test (performed before the system is started) is required, to check that there are no circumstances when deadlines are missed. A system is **unschedulable** if the scheduler does not find a way to switch between the tasks and meet all the deadlines.

A test can be:

- **sufficient** if when the answer is yes all the deadlines are met;
- **necessary** if when the answer is no there really is a situation in which deadlines are missed;
- **exact** if it is both sufficient and necessary.

We require a **sufficient** test and we would like it to be as close as possible to necessary.

Execution Time Estimation

Execution Time Estimation

Basic question: how much CPU does this piece of code need?

Two major approaches:

- measuring execution time;
- analyzing execution time.

Measuring Execution Time

The code is compiled and run with measuring devices (for example a logical analyzer) connected or the operating system provides execution time measurements.

If the piece of code has input data, a large set of test input data must be used and the longest time required is the longest time masured plus some safety margin.

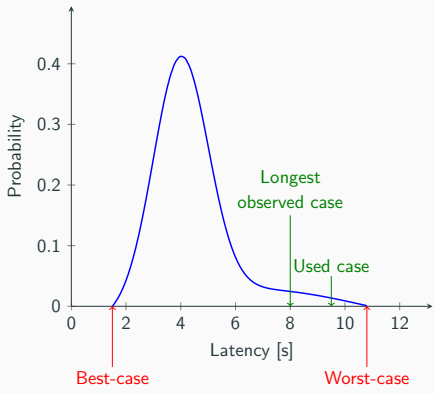
Measuring Execution Time

Problems:

- execution times are data dependent (for example sensor reading);
- caching (memories have different speed, a memory reference causing a cache miss takes much longer time that one that finds the data in the cache);
- pipelining and speculative execution;
- garbage collectors, etc.

The general problem of this approach is that we are **not guaranteed** that we have experienced the longest execution time.

Measuring Execution Time



Analyzing Execution Time

Aim: a tool that takes the source code and automatically decides the longest execution time with formal correctness (researched in the last 20 years).

Problems: compiler dependent (different compilers generate different code, the remedy is working with the machine code).

Approach: use the instruction tables from the CPU manufacturer and add up the instruction times of the individual statements.

Analyzing Execution Time

Problem: branching statements (if, case)
we don't know which code is executed before runtime.

```
if (x>5)
 x=x+1;
else
 x=x*3;
```

```
MOV(_X) D0
CMP D0 5
BGT L1
MUL D0 3
MOVE D0
JMP L2
L1: ADD D0 1
L2: ...
```

B1 | B2 | B3

```
if B1
 B3
else
 B2
```

Longest execution time:
`time(B1) + max(time(B2), time(B3))`

Analyzing Execution Time

Problem: branching statements (if, case)
we don't know which code is executed before runtime.

```
cycles
MOV: 8
CMP: 4
BGT: 4
MUL: 48
JMP: 4
ADD: 4
```

```
cycles(B1) = 8+4+4 = 16
cycles(B2) = 48+8+4 = 60
cycles(B3) = 4
cycles(if) = 16 + max(60,4) = 76
```

8MHz clock frequency, 1 cycle takes 125 ns
time(if) = 76 * 125 ns = 9.5 μs

Analyzing Execution Time

Extended to more complex statements like nested if statements.

```
if (x>0)
 if (x>5) x = x+1;
 else x = x*3;
else
 x = x+10;
```


Analyzing Execution Time

Problems:

- **Loops** (for, while): number of iterations unknown.
  Remedy: the programmer must annotate the source code with the maximum number of iterations the loop executes.
- **Recursion**: how deep can the recursive call get.
  Remedy: recursion not allowed.
- **Allocation of dynamic memory**: unknown time for memory management. Difficult to handle for an analysis tool.

Analyzing Execution Time

Problems:

- **goto**: data flow difficult to find out.
- **Caches and multi-threaded execution**: with single threaded applications caches can be handled well, with multi-threaded application pessimistic assumption that each context switch causes cache misses.

The general problem of the approach is that it is very pessimistc. The actual longest execution time might be substantially smaller than the analysis response. However, the analytical approach is the only choice to obtain formal guarantees.

WCET Analysis Tools

Three phases:

- Flow analysis: calculate all possible execution paths in the program, in order to limit the maximum number of times the different instruction types can be executed.
- Low-level analysis: calculates the execution time of the instructions on the given hardware.
- WCET calculation: combine the previous steps.

For the uniprocessor case with simple cache structures and single threaded applications, results are typically only 10-15% bigger than the true WCET. For multi-threaded applications or multicore platforms the pessimism increases.

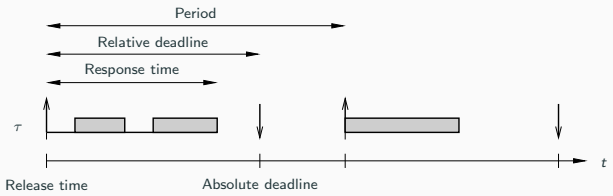
Scheduling Approaches

Notation

Notation	Description
C_i	Worst case execution time of task i
T_i	Period of task i
D_i	Relative deadline of task i

CPU Utilization: $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$

Notation



The critical instant

It can be shown that, in the uni-processor case, the worst situation from a schedulability perspective occurs when all the tasks want to start their execution at the same time instant. This is known as the *critical instant*. If the task set is schedulable in this situation it will also be schedulable in other situations. If we can show that the task set is schedulable for the worst-case execution times, the task set will be schedulable if execution times are shorter. **In uni-processor analysis, we just need to check for the worst case execution time and the critical instant**.

Scheduling Approaches

- Static Cyclic Scheduling
- Fixed Priority Scheduling
- Earliest Deadline Scheduling
- Reservation Based Scheduling

Static Cyclic Scheduling

- Offline approach.
- Configuration algorithm generates an execution table (or calendar) using many different algorithms (optimization).
- The table repeats cyclically. The runtime dispatcher simply follows the table (sets up an hardware interrupt when a context switch should be performed, starts the first task in calendar, when the hardware interrupt arrives the first task is preempted and the second is run).
- Both preemptive and non-preemptive scheduling.

Static Cyclic Scheduling: analysis

- **Analysis**: trivial, run through the table and check timing requirements.
- **Limitations**: can only handle periodic tasks, aperiodic are made periodic using polling; the task calendar cannot be too large (shortest repeating cycle is the hyperperiod – the least common multiple, LCM, of the task periods – example: periods 5, 10 and 20 gives a cycle of 20, periods 7, 13 and 23 gives a cycle of 2093 – to reduce the calendar the periods can be made shorter).

Static Cyclic Scheduling: analysis

- **Advantages**: a number of different constraints can be handled – for example exclusion constraints, precedence constraints. It is possible to use constraint programming to find a schedule.
- **Disadvantages**: inflexible (static design), building a schedule is NP-hard (potentially even if a schedule exists it can be difficult to find, good heuristic algorithms).

Static Cyclic Scheduling: example

Task Name	T	D	C
A	5	5	2
B	10	10	4

CPU Utilization: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{2}{5} + \frac{4}{10} = 0.8$
Schedule lenght: LCM(5, 10) = 10

Static Cyclic Scheduling: example

Schedule (repeat):

- from 0 to 2, task A;
- from 2 to 6, task B;
- from 6 to 8, task A;
- from 8 to 10, no task.

Worst case response time for task A: $3 \leq D_A$
Worst case response time for task B: $6 \leq D_B$

Static Cyclic Scheduling: implementation

```
CurrentTime(t);
LOOP
  A();
  B();
  A();
  IncTime(t, 10);
  WaitUntil(t);
END;
```

Problem: what if it only takes 2 time units to execute task B? Then A would start before it should.

Static Cyclic Scheduling: implementation

```
CurrentTime(t);
LOOP
  A();
  IncTime(t, 2);
  WaitUntil(t);
  B();
  IncTime(t, 4);
  WaitUntil(t);
  A();
  IncTime(t, 4);
  WaitUntil(t);
END;
```


Fixed Priority Scheduling

- Each task has a fixed priority.
- The dispatcher selects the task with the highest priority.
- Preemptive.
- Used in most real-time kernels and real-time operating systems.

Fixed Priority Scheduling: Rate Monotonic

- Rate Monotonic is a scheme for assigning priorities to tasks.
- Priorities are set monotonically with rate.
- A task with a shorter period is assigned a higher priority.
- Introduced in C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, JACM, Vol. 20, Number 1, 1973.

Fixed Priority Scheduling: Rate Monotonic analysis

Model:

- Periodic tasks,
- $D_i = T_i$,
- Tasks are not allowed to be blocked or to suspend themselves,
- Priorites are unique,
- Task execution times bounded by C_i,
- Task utilization $U_i = \frac{C_i}{T_i}$,
- Interrupts and context switches take zero time.

Fixed Priority Scheduling: Rate Monotonic analysis

Result:

- If the task set has a utilization below a utilization bound then all deadlines will be met.

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

Sufficient condition (if the utilization is larger than the bound, the task set may still be schedulable).
As $n \to \infty$ the utilization bound tends to 0.692 ($\log 2$). If the CPU utilization is less than 69% all the deadlines are met.
Alternative (tighter) test – Hyperbolic Bound:

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

Fixed Priority Scheduling: Rate Monotonic analysis

Since 1973 the models have become more flexible and the analysis better; M. Joseph and P. Pandaya, *Finding Response Times in a Real-Time System*, The Computer Journal, Vol. 29, No. 5, 1986.

Notation:

Notation	Description
C_i	Worst-case execution time of task i
T_i	Period of task i
D_i	Relative deadline of task i
R_i	Worst-case response time of task i

Fixed Priority Scheduling: Rate Monotonic analysis

Scheduling test: $R_i \leq D_i$ (necessary and sufficient)

Model:

$$D_i \leq T_i, \qquad R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of tasks of higher priority than task i. The function $\lceil x \rceil$ is the *ceiling function* that returns the smallest integer $\geq x$.
Recurrence relation, solved by iteration. The smallest solution is searched for.

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

Start with $R_i^0 = 0$.

Fixed Priority Scheduling: Rate Monotonic analysis

Limitations of the exact analysis: if the response time is larger than the period then the quantitative value cannot be trusted

- Reason: The analysis does not take interference from previous jobs of the same task into account.
- More advanced analysis exists.

However, one still knows that the deadline won't be met, which is normally what one is interested in.

Fixed Priority Scheduling: Rate Monotonic analysis

Best-case response time:
Under rate-monotonic priority assignment one can also calculate the best-case response time R_i^b of a task i.

$$R_i^b = C_i^{\min} + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^b - T_j}{T_j} \right\rceil_0 C_j^{\min}$$

where $C_i^{\min}$ is the best-case execution time of the task and $\lceil x \rceil_0 = \max(0, \lceil x \rceil)$.

Can be used to calculate the worst-case input-output latency of a control task.

Fixed Priority Scheduling: Rate Monotonic example

Task name	T	D	C	Priority
A	52	52	12	low
B	40	40	10	medium
C	30	30	10	hight

Original (approximative) analysis:

$\sum_{i=1}^{3} \frac{C_i}{T_i} = 0.814$.

$3(2^{1/3} - 1) = 0.7798$. Using this analysis we cannot say if the task set is schedulable or not.

Hyperbolic bound:

$\prod_{i=1}^{3}(\frac{C_i}{T_i} + 1) = 2.0508$ (not ≤ 2). Using this analysis we cannot say if the task set is schedulable or not.

Fixed Priority Scheduling: Rate Monotonic example

Task name	T	D	C	Priority
A	52	52	12	low
B	40	40	10	medium
C	30	30	10	hight

Exact analysis:

$$R_C^0 = 0, R_C^1 = C_C = 10, R_C^2 = C_C = 10$$
$$R_B^0 = 0, R_B^1 = C_B = 10,$$
$$R_B^2 = C_B + \left\lceil \frac{10}{T_C} \right\rceil C_C = 20,$$
$$R_B^3 = \ldots = 20$$
$$R_A^0 = 0, R_A^1 = C_A = 12,$$
$$R_A^2 = C_A + \left\lceil \frac{12}{T_B} \right\rceil C_B + \left\lceil \frac{12}{T_C} \right\rceil C_C = C_A + C_B + C_C = 32$$
$$R_A^3 = \ldots = 42, R_A^4 = \ldots = 52, R_A^5 = \ldots = 52$$

$R_i \leq D_i \Rightarrow$ schedulable

Fixed Priority Scheduling: Rate Monotonic example

Task C has highest priority $\rightarrow$ will not be interrupted and hence $R_C = C_C = 10 \quad (R_C^1)$

Task B has medium priority. The response time will be at least equal to $C_B = 10 \quad (R_B^1)$. During that time B will be interrupted once by C. Hence, the response time will be extended by the execution time of C, i.e. $R_B^2 = 10 + 10 = 20$. During this time B will only be interrupted once by C and that has already been accounted for, i.e. $R_B^3 = 20$.

Task A has lowest priority. The response will be at least equal to $C_A = 12 \quad (R_A^1)$. During that time A will be interrupted once by C and once by B, i.e., $R_A^2 = 12 + 10 + 10 = 32$. During this time A will be interrupted twice by C and once by B, i.e., $R_A^3 = 32 + 10 = 42$. During this time A will be interrupted twice by C and twice by B, i.e., $R_A^4 = 42 + 10 = 52$. During this time no more unaccounted for interrupts will occur, i.e., $R_A^5 = 52$.

Fixed Priority Scheduling: Deadline Monotonic

- The rate monotonic policy is not very good when $D \leq T$.
- An infrequent but urgent task would still be given a low priority.
- The *deadline monotonic* ordering policy works better.
- A task with a short relative deadline D gets a high priority.
- This policy has been proved optimal when $D \leq T$ (if the system is unschedulable with the deadline monotonic ordering then it is unschedulable with *all* other orderings).
- With $D \leq T$ we can control the jitter in control delay.

Fixed Priority Scheduling: Deadline Monotonic analysis

For a system with n tasks, all tasks will meet their deadlines if the sum over all tasks of the ratio between the worst-case execution time of the task and the deadline of the task is below a certain bound.

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

Fixed Priority Scheduling: Deadline Monotonic analysis

Exact Analysis:
The response time calculations from the rate monotonic theory is also applicable to deadline monotonic scheduling. Response time calculation does not make any assumptions on the priority assignment rule.

Fixed Priority Scheduling: Extension – the blocking problem

How should interprocess communication be handled.

The analysis up to now does not allow tasks to share data under mutual exclusion constraints (e.g. no semaphores or monitors)

Main problem:

- a task i might want to lock a semaphore, but the semaphore might be held by a lower priority task;
- task i is blocked.

Fixed Priority Scheduling: Extension – the blocking problem

The *blocking factor*, B_i is the longest time a task i can be delayed by the execution of lower priority tasks

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

Priority inversion may cause unbounded blocking time if ordinary locks are used.

Different locking schemes have different blocking times:

- ordinary priority inheritance;
- priority ceiling protocol;
- immediate inheritance protocol.

Fixed Priority Scheduling: Further extensions

- Release Jitter: the difference between the earliest and latest release of a task relative to the invocation of the task;
- Context Switch Overheads;
- Clock Interrupt Overheads;
- Distributed systems using CAN.

Fixed Priority Scheduling: Overrun behavior

- Overrun means exceeding the worst-case execution time.
- With fixed priority schemes, this will only affect the current task and lower priority ones. These tasks can miss deadlines and/or not get any execution time at all. Higher priority tasks are unaffected.

Earliest Deadline First (EDF) Scheduling

- Dynamic Approach: all scheduling decisions are made online.
- The task with the absolute smallest deadline runs.
- Preemptive.
- Ready-queue sorted in deadline order.
- Dynamic priorities. It is more intuitive to assign deadlines to tasks than priorities. Requires only local knowledge.

Earliest Deadline First (EDF) Scheduling: analysis

Simplest Model:

- Periodic Tasks,
- Each task i has a period T_i, a worst-case computation time requirement C_i and a relative deadline D_i,
- Tasks are independent,
- The kernel is ideal,
- $D_i = T_i$.

Earliest Deadline First (EDF) Scheduling: analysis

Result:

- If the utilization U of the system is not more than 100% then all the deadlines are met.

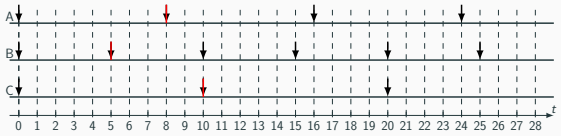
$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Necessary and sufficient condition. Advantage: the processor can be fully used. Less restrictive assumptions make the analysis harder (see RTCS for the analysis in the case $D_i \leq T_i$).

Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

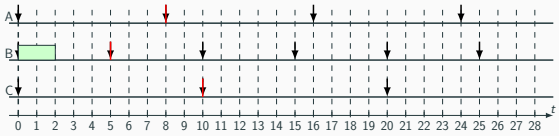
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

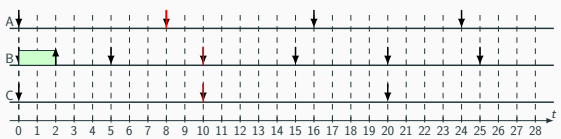
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

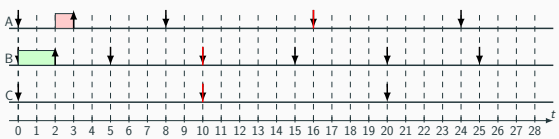
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

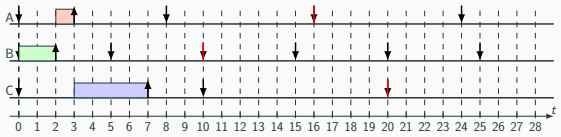
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

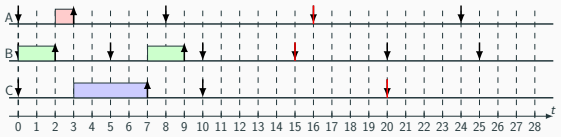
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

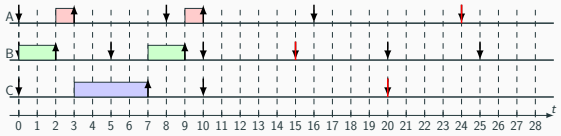
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

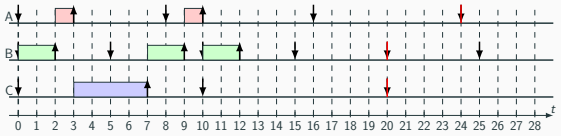
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

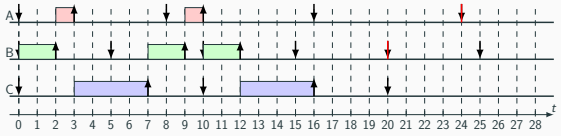
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

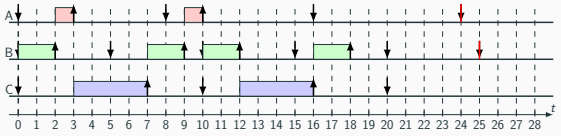
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

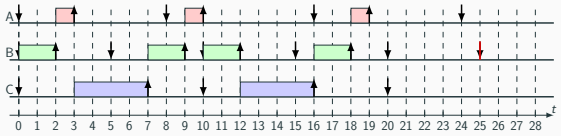
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

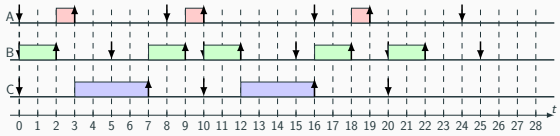
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

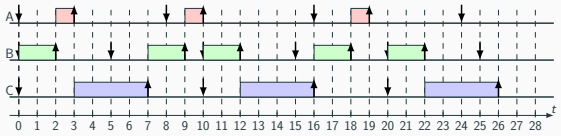
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

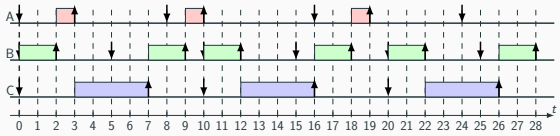
Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: example

Task name	T	D	C
A	8	8	1
B	5	5	2
C	10	10	4

Utilization: $1/8 + 2/5 + 4/10 = 0.925$



Earliest Deadline First (EDF) Scheduling: overrun behavior

In the case of overrun all tasks will be affected, so they may all miss deadlines (there is a "domino" effect). However, in general EDF is more fair than priority-based scheduling – the available resources are distributed among all the tasks.

Earliest Deadline First (EDF) Scheduling: summary

Also for EDF there exists a very well-developed schedulability theory. Resource access protocols similar to priority inheritance and ceiling.

Reservation Based Scheduling

If a tasks overruns (executes longer than anticipated) this will effect other tasks negatively.

- In priority-based systems the priority decides which tasks that will be effected.
- In deadline-based systems all tasks will be effected.

We want to provide temporal protection between tasks that guarantees that a certain task or group of tasks receives a certain amount of the CPU time.

Reservation Based Scheduling: static and dynamic

Use static cyclic scheduling for some tasks and let the other tasks be priority-based (event-based) which only may execute during the idle periods of the static schedule.

Used in Rubus from Arcticus:

- swedish RTOS used by Volvo,
- red threads - statically scheduled,
- blue threads - dynamically scheduled.

Reservation Based Scheduling: priority-based systems

How can, conceptually, a reservation-based scheduling system be implemented on top of ordinary priority-based scheduling. Each task or task set receives a certain percentage of the CPU. (50% + 30% + 20%). Can be viewed as if the tasks are executed on a correspondingly much slower CPU.

Two variants:

- Each task set gets exactly its share of the CPU;
- Each task set gets at least its share of the CPU.

Question: Over which time horizon does the CPU reservation hold?

Reservation Based Scheduling: priority-based systems

Each task set gets exactly its share of the CPU.

The scheduler can be viewed as consisting of as many ready-queues as there are reservation sets.

An external timer is set up to generate interrupts when it is time to switch which ready-queue that is active.

One idle process in each ready-queue.

Reservation Based Scheduling: priority-based systems

Each task set gets at least its share of the CPU.

One ready-queue.

Make sure that the tasks belonging to the currently serviced task set all have higher priority than the tasks in the tasks sets which are not serviced.

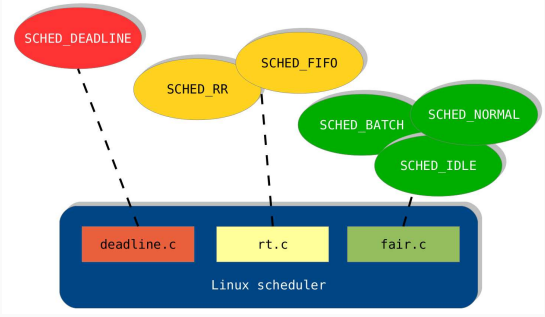
An external timer is set up to generate interrupts when it is time to switch between the tasks sets.

Lower the priorities of the the tasks that have been serviced and raise the priorities of the tasks that should be serviced.

A single idle task.

Reservation Based Scheduling: SCHED_DEADLINE

In Mainline Linux Kernel since 2 Feb 2014 17:12:22 (Linux 3.14.2).



Reservation Based Scheduling: Industrial practice

Beginning to emerge in commercial RTOS.

Integrity from Green Hills Software Inc.

More Information

A nice introduction and overview of the state-of-the-art in uni-processor scheduling of real-time systems can be found in:

"Real Time Systems by Fixed Priority Scheduling" by Ken Tindell and Hans Hansson, Dept. of Computer Systems, Uppsala University

http://www.docs.uu.se/~hansh/fpsnotes-9710.ps