Lecture 12: An Overview of Scheduling Theory

[RTCS Ch 8]

- Introduction
- Execution Time Estimation
- Basic Scheduling Approaches
  - Static Cyclic Scheduling
  - Fixed Priority Scheduling
    - Rate Monotonic Analysis
  - Earliest Deadline Scheduling

Goal

Question to be answered:

- *How can we guarantee that a set of tasks meet their deadlines?*

Problem Formulation

Events

- events occur that require computations (interrupts)
- aperiodic (sporadic) events and periodic events

Worst-Case Execution Time

- a task executes a piece of code in response to an event
- an upper bound on the CPU time it takes to execute the task without any interfering tasks (alone on the CPU)

Deadline

- Maximum allowed time when the task should be completed

Scheduling

- the choice of which event to process at a given time, i.e. which task to execute

Schedulability Analysis

- For hard real-time systems the deadlines must always be met
- Off-line guarantee test (before the system is started) required to check so that there are no circumstances that could lead to missed deadlines
- A system is *unschedulable* if the scheduler will not find a way to switch between the tasks such that the deadlines are met
- The test is *sufficient* if, when it answers "Yes", all deadlines will be met
- The test is *necessary* if, when it answers "No", there really is a situation where deadlines could be missed
- The test is *exact* if it is both sufficient and necessary
- A sufficient test is an absolute requirement and we like it to be as close to necessary as possible

- Introduction
- **Execution Time Estimation**
- Basic Scheduling Approaches
  - Static Cyclic Scheduling
  - Fixed Priority Scheduling
    - Rate Monotonic Analysis
  - Earliest Deadline Scheduling

Execution Time Estimation

Basic Question:

- "How much CPU time does this piece of code need?"

Two major approaches:

1. Measuring execution times
2. Analyzing execution times

Measuring Execution Times

- the code is compiled and run with measuring devices (e.g. logical analyzer) connected, or ..
- ..., the OS provide execution time measurements
- a large set of test input data is used
- longest time required = longest time measured (+ safety margin)

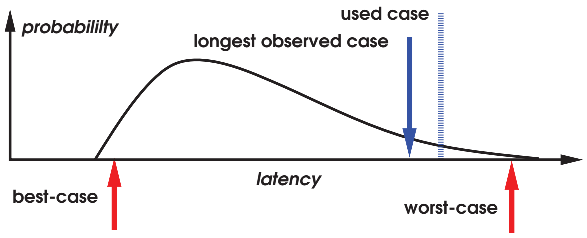
General problem:

- No guarantees that we really have encountered the longest execution time

Problems:

- execution times are data dependent (e.g. a sensor reading)
- caching
  - memories have different speeds
  - a memory reference causing a cache miss takes much longer time than a reference inside the cache
- pipelining & speculative execution
- memory accesses for multiprocessor systems
- testing a real-time problem is difficult and time consuming
- garbage collection in e.g., Java (may occur at any time)

Main Problem: No guarantees



Analyzing Execution Times

Aim:

- a tool that takes the source code and automatically and formally correct decides the longest execution time
- research area for the last 10-15 years

Problems:

- compiler dependent
  - different compilers generate different code
  - Remedy: work with the machine code

Approach:

- use the instruction time tables from the CPU manufacturer
- add up the instruction times of the individual statements

Problem:

- branching statements (IF, CASE)
  - how should we know which code that is executed

```
IF X > 5 THEN
  X := X + 1;
ELSE
  X := X * 3;
ENDIF;
```

```
MOVE (_X),D0   ]
CMP D0,#5      ] B1
BGT L1         ]
MUL D0,#3      ]
MOVE D0        ] B2
JMP L2         ]
L1: ADD D0,#1  ] B3
L2: ...
```

```
IF B1 THEN
  B3
ELSE
  B2
ENDIF
```

Longest execution time = time(B1) + max(time(B2),time(B3))

Execution times of the basic blocks:

Operation	Number of CPU cycles
MOVE	8
CMP	4
BGT	4
MUL D0,#3	16 + 2 times # '1's
JMP	4
ADD	4

time(B1) = $8 + 4 + 4 = 16$ cycles

time(B2) = $(16 + 2 * 16) + 8 + 4 = 60$ cycles (word length = 16 bits)

time(B3) = 4 cycles

$\Rightarrow$ time(if-statement) = 76 cycles
8MHz clock frequency $\Rightarrow$ 1 cycle takes 125ns
$\Rightarrow$ time(if-statement) = $76 * 125$ns= $9.5\mu s$

Extended to more complex statements

```
IF X = 0 THEN
  IF X > 5 THEN
    X := X + 1;
  ELSE
    X := X * 3;
  ENDIF;
ELSE
  X := 1;
ENDIF;
```


Problems:

- Loops (WHILE, ..)
  - How should we know how many times the code will loop?

```
WHILE X > 5 DO
  X := X - 1;
END;
```

  - Remedy: the programmer must annotate the source code with the maximum number of times the loop executes
- Recursion
  - difficult to know beforehand how deep the recursive call can get
  - Remedy: recursion not allowed
- allocation of dynamic memory
  - the time for the memory management often unknown
  - difficult for an analysis tool to handle

- `goto` statements
  - the data flow in a piece of code is difficult to work out
- caches and multi-threaded applications
  - caches with single-threaded applications can be handled reasonably well
  - caches with multi-threaded applications extremely pessimistic
    - each context switch may cause a cache miss
- Main problem: *pessimism*
  - the actual longest execution time may be substantially smaller than what the analyzer says
  - however, if we want formal guarantees the analytical approach is the only choice

WCET Analysis Tools

Three phases:

1. Flow Analysis
   - calculates all possible execution paths in the program
   - in order to limit the number of times the instructions can be executed
2. Low-level Analysis
   - calculates the execution time of the different instructions on the given hardware
3. WCET Calculation
   - Combine step 1 and 2

For the uni-processor case with simple cache structures and without complex pipelines the obtained results is typically only 10-15 % larger than the true WCET for single-threaded applications.

However, for multi-threaded applications either on a uniprocessor or a multicore platform the pessimism is much larger.

Static Cyclic Scheduling

- off-line approach
- configuration algorithm generates an execution table or calendar
- many different algorithms (optimization)
- the table repeats cyclically ⇒ static cyclic scheduling
- works for both non-preemptive and preemptive scheduling
- the run-time dispatcher simply follows the table
  - sets up an hardware interrupt at the time when a context switch should be performed (preemptive)
  - starts the first task in the calendar
  - when the hardware interrupt arrives the first task is preempted and next task is run,
  - ...

Analysis:

- trivial, run through the table and check that all timing requirements are met

Limitations:

- can only handle periodic tasks
  - aperiodic tasks are made periodic through polling
- the calendar cannot be too large
  - shortest repeating cycle = the hyperperiod = the *least common multiple, LCM* of the task periods
  - periods 5,10,20 ms gives cycle of 20 ms
  - periods 7,13,23 ms gives cycle of 2093 ms
  - periods are made shorter than they need to be to reduce the calendar

Advantages:

- A number of different task constraints can be handled
  - Exclusion constraints can be handled
  - Precedence constraints can be handled
  - Constraint programming can be used to find a schedule

Disadvantages:

- Inflexible
  - static design
- building a schedule is NP-Hard
  - we cannot expect an algorithm to always find a schedule even if one exists
  - good heuristic algorithms exist that can mostly find an solution if one exists

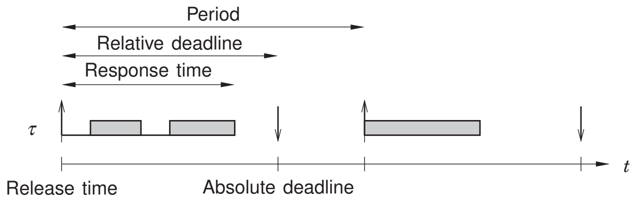
Notation

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:

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

Notation

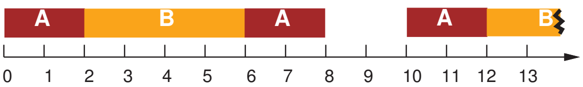


Example

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

Utilization: $2/5 + 4/10 = 0.8$

Schedule length: LCM(5,10) = 10



Worst case response time for task A, $R_A = 3 < D_A$
Worst case response time for task B, $R_B = 6 < D_B$

Implementation

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

Problem: Assume it only takes 2 time units to execute task B. Then task A will start before it should do.

Better implementation

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


- Introduction
- Execution Time Estimation
- Basic Scheduling Approaches
  - Static Cyclic Scheduling
  - **Fixed Priority Scheduling**
    - **Rate Monotonic Analysis**
  - Earliest Deadline Scheduling

Fixed Priority Scheduling

- each task has a fixed priority
- the dispatcher selects the task with the highest priority
- preemptive
- used in most r-t kernels and RTOS

The Critical Instant

It can be shown that, in the uni-processor case, the worst situation, from a schedulability perspective, occurs when all tasks want to start their execution at the same time instant.

This is known as the *critical instant.*

If we can show that 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, then the task set will also be schedulable if the actual execution times are shorter.

Hence, all uni-processor scheduling analysis only need to check for this case.

Rate Monotonic Priority Assignment

- a scheme for assigning priorities to processes
- priorities are set monotonically with rate (period)
- 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

Rate Monotonic Analysis

Assumptions needed = model

Model:

- periodic tasks
- $D_i = T_i$
- tasks are not allowed to be blocked or suspend themselves
- priorities are unique
- task execution times bounded by C_i
- task utilization $U_i = C_i/T_i$
- interrupts and context switches take zero time

Result:

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

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

Sufficient condition (if the utilization is larger than the bound the task set may still be schedulable)

As $n \to \infty$, the utilization bound $\to 0.693 (= \ln 2)$

"If the CPU utilization is less than 69%, then all deadlines are met"

Alternative tighter test (Hyperbolic Bound):

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

Response Time 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

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

Model:

- $D_i \le T_i$

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

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 $\ge x$.

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 $R_i^0 = 0$

Example

Task set:

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

Original (approximative) analysis:

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

$$3(2^{1/3} - 1) = 0.7798$$

Not schedulable

Hyperbolic bound:

$$\prod_{i=1}^{i=3} (\frac{C_i}{T_i} + 1) = 2.0508$$

Not schedulable

Exact analysis:

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

Task name	T	D	C	Priority	R
A	52	52	12	low	52
B	40	40	10	medium	20
C	30	30	10	high	10

$R_i \le D_i \Rightarrow$ schedulable

Derivation of exact formulae

Task C has highest priority → 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., $R_A^5 = 52$.

Limitation of the Exact Formula

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.

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.

Deadline Monotonic Scheduling

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.

Deadline Monotonic Scheduling - Sufficient Condition

For a system with n tasks, all tasks will meet their deadlines if the total utilization of the system is below a certain bound.

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

Deadline Monotonic Scheduling - 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.

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

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

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

Overrun Behaviour - Fixed Priorities

Overrun = exceeding the worst-case execution time

Will only affect the current task and lower priority tasks

These will miss deadlines or, in the worst case, not get any execution time at all

Higher priority tasks will be unaffected.

- Introduction
- Execution Time Estimation
- Basic Scheduling Approaches
  - Static Cyclic Scheduling
  - Fixed Priority Scheduling
    - Rate Monotonic Analysis
  - **Earliest Deadline Scheduling**

Earliest Deadline First (EDF) Scheduling

- dynamic approach: all scheduling decisions are made on-line by the dispatcher
- the task with the smallest absolute deadline runs
- preemptive
- ready-queue sorted in deadline order
- "dynamic priorities"
- more intuitive to assign deadlines to tasks than to assign priorities
  - requires only local knowledge

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
  - $D_i = T_i$
  - independent task execution
  - ideal kernel

Result:

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

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

Necessary and sufficient condition

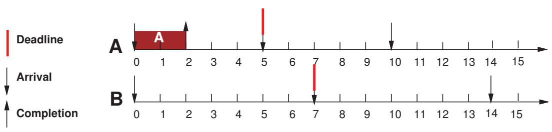
Advantage: Processor can be fully used.

Less restrictive assumptions make the analysis harder (see RTCS for the analysis in the case $D_i \leq T_i$.)

Example

Task name	T	D	C
A	5	5	2
B	7	7	4

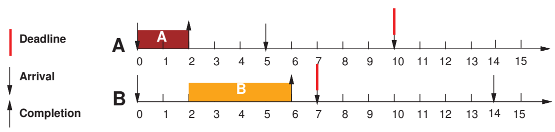
Utilization: $2/5 + 4/7 = 0.971$



Example

Task name	T	D	C
A	5	5	2
B	7	7	4

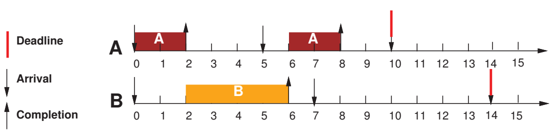
Utilization: $2/5 + 4/7 = 0.971$



Example

Task name	T	D	C
A	5	5	2
B	7	7	4

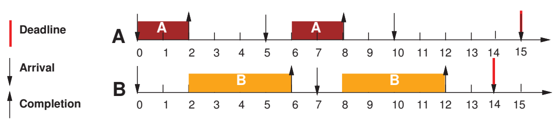
Utilization: $2/5 + 4/7 = 0.971$



Example

Task name	T	D	C
A	5	5	2
B	7	7	4

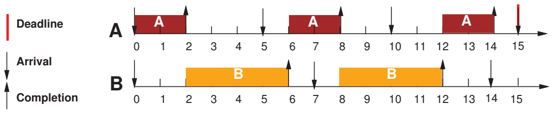
Utilization: $2/5 + 4/7 = 0.971$



Example

Task name	T	D	C
A	5	5	2
B	7	7	4

Utilization: $2/5 + 4/7 = 0.971$



Overrun Behaviour

In the case of overrun all tasks will be affected, i.e., all tasks may miss deadlines.

The "Domino effect"

However, in general EDF is more fair than priority-based scheduling

- the available resources will be distributed among all the tasks

EDF: Summary

Also for EDF there exists a very well-developed schedulability theory

Resource access protocols similar to priority inheritance and ceiling.