

Lecture 8: Scheduling Theory

[RTCS Chap. 8]

- Introduction
- The Scheduling Problem
- Execution Time Estimation
 - Measurements
 - Code Analysis
- Three Scheduling Approaches
 - Static Cyclic Scheduling
 - Earliest Deadline Scheduling
 - Fixed Priority Scheduling
 - * Rate Monotonic Analysis

1

Introduction

Questions to be answered:

- *How should we chose priorities?*
- *How can we guarantee that different tasks meet their deadlines?*

2

Background

Scheduling problems are studied in two research communities:

- Operations Research
 - job shop and flow-shop problems
 - scheduling of machines, orders, batches, projects, ..
 - resources: machines, factory cells, unit processes, ..
 - static (off-line) techniques
- Computer Science
 - schedule tasks in a uni- or multi-processor environment
 - dynamic techniques

Still, the problems have many things in common.

3

Problem Formulation

Events

- events occur that require computations
- interrupts
- a task executes a piece of code in response to an event
- aperiodic (sporadic) / periodic events

Required Computation Time

- the CPU time it takes to execute the piece of code associated with the event

Deadline

- an upper bound on the time taken to execute the piece of code associated with the event

Scheduling

- the choice of which event to process at a given time

4

- Introduction
- **The Scheduling Problem**
- Execution Time Estimation
 - Measurements
 - Code Analysis
- Three Scheduling Approaches
 - Static Cyclic Scheduling
 - Earliest Deadline Scheduling
 - Fixed Priority Scheduling
 - * Rate Monotonic Analysis

5

The Scheduling Problem

Three activities:

1. run-time dispatching
2. off-line configuration
3. analysis

6

Run-Time Dispatching

- the actual switching between computations for different events at run-time
- task-based system
- preemptive or non-preemptive
- uses knowledge about
 - the current state of the system (e.g. the time)
 - off-line information (e.g. task priorities)
- cf. Schedule in the kernel

7

Off-Line Configuration

- generates the static information for the run-time dispatcher
- very large activity in some approaches – scheduling table
- very small activity in other approaches – task priorities

8

Analysis

- in hard real-time systems the deadlines must always be met
- off-line analysis (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 analysis is *sufficient* if, when it answers "Yes", all deadlines will be met
- the analysis is *necessary* if, when it answers "No", there really is a situation where deadlines could be missed
- the analysis is *exact* if it is both sufficient and necessary
- a sufficient analysis is an absolute requirement and we like it to be as close to necessary as possible

9

- Introduction
- The Scheduling Problem
- **Execution Time Estimation**
 - Measurements
 - Code Analysis
- Three Scheduling Approaches
 - Static Cyclic Scheduling
 - Earliest Deadline Scheduling
 - Fixed Priority Scheduling
 - * Rate Monotonic Analysis

10

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

11

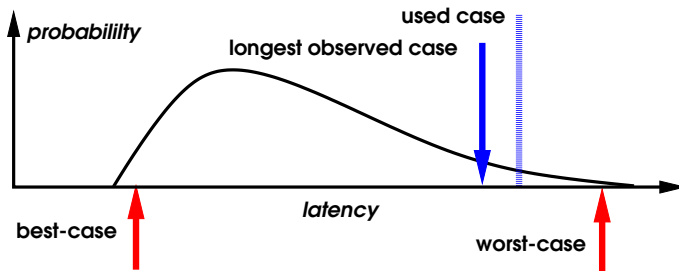
Measuring Execution Times

- the code is compiled and run with measuring devices (e.g. logical analyzer) connected
- a large set of test input data is used
- longest time required = longest time measured

General problem:

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

12



13

Problems:

- execution times are data dependent (e.g. a sensor value)
- caching (i.e. buffering of code locally to improve performance)
 - if the code is small enough to stay in the cache then the computation time is much faster
 - remedy: disable caching
- pipelining (i.e. an implementation technique where multiple instructions are overlapped in execution)
 - if the code does not execute any branches then it stays in the pipeline and executes much faster
 - remedy: disable pipelining
- memories with different speeds
 - the execution time can change depending on where the program is stored

14

- memory access for multiprocessor systems
 - several processors accessing the same memory over a bus
 - the execution time of a piece of code on one processor depends on how code on the other processors execute
- testing a real-time problem is difficult and time consuming
- automatic garbage collection in e.g., Java

15

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

16

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      MOVE (_X),D0
  X := X + 1;      CMP D0,#5
ELSE              BGT L1
  X := X * 3;      MUL D0,#3
ENDIF;            MOVE D0
                 JMP L2
                 L1: ADD D0,#1
                 L2: ...

```

} B1 IF B1 THEN
} B2 B3
} 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) = 48 + 8 + 4 = 60 cycles (word length = 16 bits)

time(B3) = 4 cycles

⇒ time(if-statement) = 76 cycles

8MHz clock frequency ⇒ 1 cycle takes 125ns

⇒ time(if-statement) = 76 * 125ns = 9.5µs

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 (IF, 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
- 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

21

- Introduction
- The Scheduling Problem
- Execution Time Estimation
 - Measurements
 - Code Analysis
- **Three Scheduling Approaches**
 - Static Cyclic Scheduling
 - Earliest Deadline Scheduling
 - Fixed Priority Scheduling
 - * Rate Monotonic Analysis

22

Static Cyclic Scheduling

- off-line approach
- configuration algorithm generates an execution table or calendar
- many different algorithms (optimization)
- the table repeats cyclically \Rightarrow 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,
 - ...

23

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
 - problems with short-deadline sporadic events
- the calendar cannot be too large
 - shortest repeating cycle = 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

24

- 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 a solution if one exists

25

Advantages:

- Exclusion constraints can be handled
 - tasks are allowed to share data
 - assume that tasks A and B share some data: the scheduling algorithm can ensure that we never switch from A to B unless A has completed all its execution and vice versa
 - can control the concurrency even when a task is not actually accessing shared data (cf. semaphores or monitors)
- Precedence constraints can be handled
 - requirements on a task X to finish before another task Y is allowed to run

26

Notation

Notation	Description
C_i	Worst-case computation time of task i
T_i	Period of task i
D_i	Deadline of task i

CPU utilization U :

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

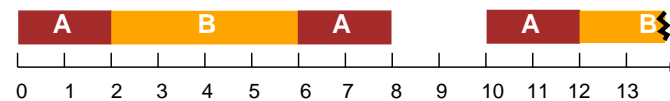
27

Example

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

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

Schedule length: $\text{LCM}(5,10) = 10$



Worst case response time for task A, $R_A = 3 < D_A$

Worst case response time for task B, $R_B = 6 < D_B$

28

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.

29

Better implementation

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

30

- Introduction
- The Scheduling Problem
- Execution Time Estimation
 - Measurements
 - Code Analysis
- Three Scheduling Approaches
 - Static Cyclic Scheduling
 - **Earliest Deadline Scheduling**
 - Fixed Priority Scheduling
 - * Rate Monotonic Analysis

31

Earliest Deadline First (EDF) Scheduling

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

32

Analysis:

- assumptions needed = model
- Simplest model:
 - periodic tasks
 - each task i has a period T_i ,
 - a worst-case computation time requirement C_i , and
 - a deadline D_i
 - $D_i = T_i$
 - * infrequent events with short deadlines does not fit
 - independent task execution
 - * strong and unrealistic assumption

33

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.

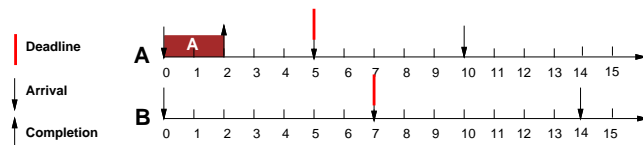
Less restrictive assumptions make the analysis harder

34

Example

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

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

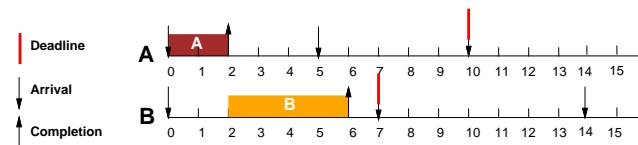


35

Example

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

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

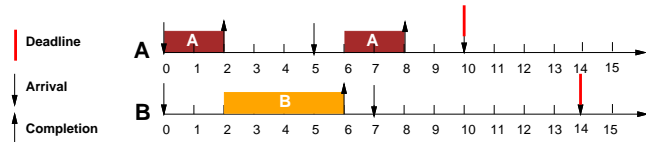


36

Example

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

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

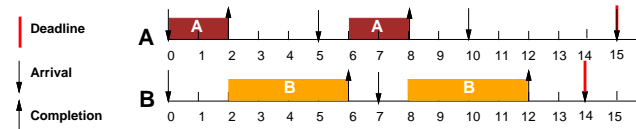


37

Example

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

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

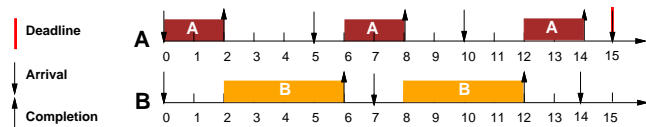


38

Example

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

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



39

- Introduction
- The Scheduling Problem
- Execution Time Estimation
 - Measurements
 - Code Analysis
- Three Scheduling Approaches
 - Static Cyclic Scheduling
 - Earliest Deadline Scheduling
 - **Fixed Priority Scheduling**
 - * Rate Monotonic Analysis

40

Fixed Priority Scheduling

- each task has a fixed priority
- the dispatcher selects the task with the highest priority
- preemptive
- cf. the real-time kernel and most RTOS

41

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

42

Rate Monotonic Analysis

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
- interrupts and context switches take zero time

43

Result:

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

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

Approximate analysis (if the utilization is larger than the bound the task set may still be schedulable)

As $n \rightarrow \infty$, the utilization bound $\rightarrow 0.693$

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

44

Generalized 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 computation time of task i
T_i	Period of task i
D_i	Deadline of task i
R_i	Worst-case response time of task i

Scheduling test: $R_i \leq D_i$

Model:

- $D_i \leq T_i$
- can handle sporadic events (long period but short deadline)

45

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

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 $R_i^0 = 0$

46

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

47

Exact analysis:

$$R_A^0 = 0, R_A^1 = C_A = 12,$$

$$R_A^2 = C_A + \left\lceil \frac{12}{T_B} \right\rceil C_B + \left\lceil \frac{12}{T_C} \right\rceil C_C = C_A + C_B + C_C = 32$$

$$R_A^3 = \dots = 42, R_A^4 = \dots = 52, R_A^5 = \dots = 52$$

$$R_B^0 = 0, R_B^1 = C_B = 10,$$

$$R_B^2 = C_B + \left\lceil \frac{10}{T_C} \right\rceil C_C = 20,$$

$$R_B^3 = \dots = 20$$

$$R_C^0 = 0, R_C^1 = C_C = 10, R_C^2 = C_C = 10$$

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

$R_i \leq D_i \Rightarrow$ schedulable

48

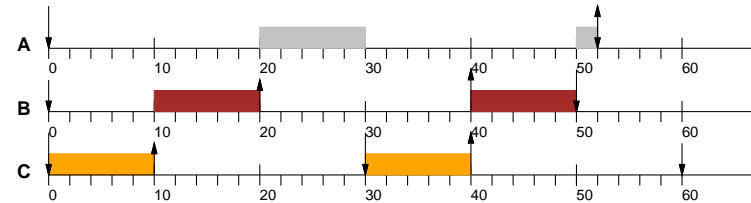
Derivation of exact formulae

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

Task B has medium priority. The response time will be at least equal to $C_B = 10$ (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 $C_A = 12$ (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., $R_A^5 = 52$.

49



50

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 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.

The response time calculations from the rate monotonic theory is also applicable on deadline monotonic scheduling.

51

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

52

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

Example Response Time Analysis in Priority Ceiling Protocol:

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

Calculation of B_i :

$$B_i = \max_{\forall k \in lp(i), \forall s \in uses(k) | ceil(s) \geq pri(i)} cs_{k,s}$$

We look at all the tasks of lower priority than task i , and then at all the semaphores that they can lock, and then from those select only the semaphores where the ceiling of the semaphore is higher than or the same the priority of task i . Then we look at the computation times that these semaphores are held for. The longest of these times is B_i .

53

Further Extensions: Jitter & Overheads

Release Jitter:

- the difference between the earliest and latest release of a task relative to the invocation of the task
- J_i is the release jitter of task i

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

$$R_i = J_i + \omega_i$$

Context Switch Overheads:

- a context switching takes time C_{sw}

$$\omega_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\omega_i + J_j}{T_j} \right\rceil (C_j + 2C_{sw})$$

$$R_i = J_i + \omega_i$$

54

Clock Interrupt Overheads:

- T_{tick} tick interval
- C_{tick} the time taken to handle the interrupt
- C_{queue} the time taken to move a task from the Time-queue to the Ready-queue

$$\omega_i = C_i + 2C_{sw} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{\omega_i + J_j + T_{tick}}{T_j} \right\rceil (C_j + 2C_{sw}) + \sum_{\forall j \in alltasks} \left\lceil \frac{\omega_i + J_j + T_{tick}}{T_j} \right\rceil C_{queue} + \left\lceil \frac{\omega_i}{T_{tick}} \right\rceil C_{tick}$$

$$R_i = J_i + T_{tick} + \omega_i$$

Distributed systems using CAN-bus for communication

55

Alternative Scheduling Models

The Multi-Frame Model:

- execution times and deadlines for tasks are allowed to vary from job to job according to a periodic pattern
- sufficient and necessary schedulability conditions

Sub-Task Scheduling:

- each task is divided into serially executed subtasks each characterized by an execution time, a priority, and a deadline
- in the recurrent task model is also possible to model branching in the execution of the subtasks

56

Offset-Based Scheduling:

- the periodic task model is based on the assumptions that all tasks may arrive simultaneously
- if the is can be avoided the schedulability of a task set increases
- using task offsets task can be shifted in time
- schedulability analysis (Gutierrez and Harbour)

57

Scheduling of Mixed Task Sets

Hard periodic tasks in combination with soft aperiodic tasks

Polling aperiodic tasks – increases processor utilization

Server techniques:

- Main idea: use a special high-priority server task for scheduling the pending aperiodic work
- the server has time tickes that can be used to schedule and execute the aperiodic tasks
- if there is aperiodic work pending and the server has unused tickes, the aperiodic tasks execute until they finish or the available tickes are exhausted.
- several servers proposed, e.g., priority exchange server, deferrable server, sporadic server, slack stealer, constant bandwidth server
- the main differences concern how the capacity of the server is replenished and the maximum capacity of the server
- schedulability results available

58

Summary

- scheduling is an active research area in real-time systems
- Generalized Rate Monotonic Scheduling Theory is relatively mature
 - e.g. adopted by IBM, European Space Agency, NASA, several ADA vendors, ...
 - not widely spread in Sweden
- research areas
 - aperiodic tasks
 - distributed scheduling

59