BITS ZG553 - Real Time System – Lecture 1

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Course Organization

1. Fundamentals of Real-Time Systems
2. A Reference Model of Real-Time Systems
3. Commonly Used Approaches to Real-Time Scheduling
4. Clock Driven Scheduling
5. Priority-Driven Scheduling of Periodic Tasks
6. Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems
7. Resource and Resource Access Control
8. Hardware Considerations in Real-Time Systems
9. Real-Time Operating Systems
10. Software Programming Languages in Real-time Systems
11. Software Design Approaches
12. Software Requirements
13. Validation and verification techniques
14. Redundancy and Reliability

Books

Text Books

1. Liu, Jane W.S., Real Time Systems, Pearson Education, 2000
2. Laplante, Phillip A., Real-Time Systems Design and Analysis, WSE, 4th Ed., 2012

Reference Books

3. Li Quing, Real-Time Concepts for Embedded Systems, CMP books, paperback 2003 (Available on Books 24x7)
4. Burns Allen and Wellings Andy, Concurrent and Real-Time Programming in ADA, Cambridge University Press, paperback 2007 (Available on Books 24x7)

Evaluation Criteria

EC1- Online Quizes & Lab Assignments

EC2 - Mid semester Test – Closed Book

EC3 - Comprehensive Examination – Open Book

Fundamentals of Real-Time Systems

BITS Pilani
Pilani Campus

Definitions

1. A real-time system is a computer system that must satisfy bounded response-time constraints or risk severe consequences, including failure.

2. A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period

A System

(Source: P. A. Laplante & S. J. Ovaska, Real-Time Systems Design and Analysis: Tools for the Practitioner, Wiley, 4th edition)

- A system is an assembly of components connected together in an organized way
- A system is fundamentally altered if a component joins or leaves it
- It has a purpose
- It has a degree of permanence
- It has been defined as being of particular interest

Example: A Real-Time Control System

- Inputs are *excitations* and outputs are corresponding *responses*
- Inputs and outputs may be digital or analog
- Inputs are associated with sensors, cameras, etc.
- Outputs with actuators, displays, etc.

***Source:** P. A. Laplante & S. J. Ovaska, Real-Time Systems Design and Analysis: Tools for the Practitioner, Wiley, 4th edition*

Sampled Data System

- **Plant**: Controlled system, e.g. an engine, a brake etc
- **Control-law computation**: Real time system, which computes a control o/p based on the sensor output y(t) and the reference input r(t).
- **Controller**: A PID (Proportional, Integral and Derivative controller)

Sample Data System

A/D: Samples data at interval 'T'

Data samples: $r_k, y_k, u_k, k = 0, 1, 2, 3, \ldots$

$$e_k = r_k - y_k$$

Controller o/p

$$u_k = u_{k-2} + \alpha e_k + \beta e_{k-1} + \gamma e_{k-2}$$

Constants and the sampling period is chosen at the design time

Sample Data System

Feedback Control Loop pseudocode:

```
set timer to interrupt periodically with period 'T'
at each timer interrupt do
{
    do analogue-to-digital conversion to get 'y'
    compute control output 'u'
    do digital-to-analogue conversion
}
```

Sample Data System

Determining the Sampling Rate

Sampling time interval = T

In general, the faster a system can and must respond to changes, the faster the input to the actuator varies, hence the shorter the sampling period should be.

We can measure the responsiveness of the system by its *rise time* **R**.

R is the time it takes to get close to its final state after the input changes.

Typically the ratio R/T of rise time to sampling period should be between 10 and 20 in order to give a smooth response.

One can also consider this in terms of bandwidth ω which is approximately 1/2R Hz.

The Nyquist sampling theorem says that any time-continuous signal of bandwidth ω can be reproduced faithfully from its sampled values only if the sampling rate is at least 2 ω.

Recommended sampling rate is **20 to 40 times the bandwidth ω.**

Sample Data System

Multirate System

- A system typically has state defined by multiple state variables. e.g. rotation speed, temperature, fuel consumption, etc. of an engine.
- The state is monitored by multiple sensors and controlled by multiple actuators.
- Hence there are multiple sampling rates, one for each state variable.
- A system with multiple sampling rates is called a multirate system.

Determining Sampling Rate for a Multirate System

- Fastest sample rate is the choice ?
- Usually longer sampling period is an integer multiple of every shorter sampling period
- Design is done in iterative way
  - Select the fastest sampling rate controller assuming it is independent of other controller. Integrate it with the plant
  - Select the next fastest one and integrate. Continue till the slowest sampling rate controller is integrated
  - Correct the error, aroused due to the assumption that the first one is independent of others

A More Complex Control Law Computation

```
Set timer to interrupt periodically with period 'T'
At each timer interrupt do
{
    Sample and digitize sensor readings;
    Compute control output from measured and state-variable values;
    Convert control output to analogue form;
    Estimate and update system parameters;
    Compute and update state variables;
}
```

Controller Hierarchy

Controllers in a complex systems are typically organised hierarchically.

Example – Air Traffic Control System

- The air-traffic control system is at the highest level.
  - Regulates the flow of flights to the destination
  - Does by assigning each aircraft an arrival time at each metering fix (a geographical point) en route to the destination
  - Assigned arrival time to the next metering fix is the reference for the on board flight management system
- The on-board flight management system chooses the flight paths etc. and sets parameters for the lower level controllers.
  - It has to optimize the flight path based on certain constraints
  - Constrains include the ones imposed by the flight characteristics such as maximum and minimum allowed cruise speeds and decent/accent rates.
  - Another constraint is the cost i.e. Fuel consumption. A most desirable path is the most fuel efficient path.
  - When the flight is late, it should try to bring the aircraft to the next metering fix in the shortest time.
- The flight controller at the lowest level handles cruise speed, turn radius, ascend/descend rates etc.

Controller Hierarchy

Air Traffic Control System

- Real Time Command and Control Operation
  - ATC gathers information on the state of each aircraft via one or more active radars.
  - Such radars interrogates the aircrafts periodically and the aircraft responds with its state variables such as identifiers, position, altitude, heading etc.
  - ATC stores the information received from the aircrafts in a database.
  - This information is processed by the processors.
  - At the same time the surveillance system continuously analyzes the scenario and alerts the operators whenever there is a possibility of collision.

Real Time Application – Real Time Database

- Refers to diverse spectrum from stock price information system to track record databases
- Real-time data is perishable (in contrast with non-real time data such as pay-roll database)
- The age of an object measures how up-to-date the information is. It is the length of the time since the instant of its last update.
- The age of an object whose value is computed from other objects is equal to that of the oldest of those objects.
- **Absolute Temporal Consistency**
  - A set of data objects is said to be absolutely temporally consistent if the maximum age in the set is no greater than a certain threshold.

  Examples:

  - A fighter jet and the targets it tracks move at supersonic speeds, so the information on where they are must be is less than 50 ms.
  - An air traffic control system monitors commercial aircrafts at subsonic speed, hence the absolute temporal consistency threshold for it is much larger
- **Relative Temporal Consistency**
  - A set of data objects is said to be relatively temporally consistent if the maximum difference in ages of the objects in the set is no greater than the relative consistency threshold used by the application.

Real Time Application – Multimedia Systems

- **MPEG Compression / Decompression**
  - Motion Estimation
  - Discrete Cosine Transform / Inverse Discrete Cosine Transform
  - Huffman Encoding / Decoding
- **Real Time Characteristics**
  - SD Video requires 30 frames per sec, HD video requires 60 frames per sec
  - Lower frame-rates are fine for applications like video conferencing
  - DCT/IDCT, Motion Estimation and Compensation and Huffman Encoding / Decoding are computation intensive.
  - Computation need to be done per frame basis.
  - Lip-sync between Audio and Video also need to be performed.
  - Muxing / Demuxing also add to the computation.

Types of Real Time Systems

- **Purely cyclic**
  - Every task executes periodically. Even I/O operations are polled.
  - Demands on resources do not vary significantly from period to period.
  - Example: Real-time monitors

- **Mostly cyclic**
  - Most tasks execute periodically.
  - The system can also respond to some external events asynchronously.
  - Example: Process control systems

- **Asynchronous and somewhat predictable**
  - Most tasks are not periodic.
  - Duration between executions of a task may vary considerably or the resource requirements may vary. Variations have bounded ranges or known statistics.
  - Example: Multimedia systems, Radar signal processing

- **Asynchronous and unpredictable**
  - Systems which react to asynchronous events and have tasks with high run-time complexity.
  - Example: Intelligent real-time control system

Soft & Hard Real Time Systems

Soft Real Time System

- A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints.
- User requires only a demonstration that the system always meet some *statistical constraints*.
- Best effort service
- Example: Multimedia system

Hard Real Time System

- A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure
- User requires validation that the system always meet the *timing constraint*. (validation means demonstration by a provably correct and efficient procedure)
- Guaranteed service
- Example: Avionics weapons delivery system

Firm Real Time System

- A firm real-time system is one in which failure to meet a single deadline will not lead to total system failure, but missing more than few may lead to complete and catastrophic system failure
- Example: Navigation controller for an autonomous weed-killer robot

(In most of the texts, Firm Realtime Systems are not mentioned)

Jobs and Tasks

Job: Each unit of work that is scheduled and executed by the system is a job.

Task: A set of related jobs which jointly provide some system function is a task.

Example:

Computation of FFT of sensor data at a particular time instance, sending a data packet are examples of job.

Whereas the function of sending data packets is an example of a task.

Processors

The jobs are executed by operating systems.

Each job requires some resource to execute such as a CPU, a Network, a Disk etc.

In Queuing Theory, these are called Servers.

In Operating Systems literature, these are called Active Resources.

Here we will call them Processors.

Release Time and Deadline

Release time of a job is the instance of time at which the job becomes available for execution.

Deadline of a job is the instant of time by which its execution is required to be completed.

It is also called Absolute Deadline.

Relative Deadline = *Absolute Deadline – Release time*.

Example

A system which monitors and controls several furnaces

At the beginning, the system is initialized and starts execution (time 0 ms).
The system samples and reads each temperature sensor every 100 ms.
The system computes the control law of each furnace every 100 ms.

Each of the control law computation is a job, say J_k, $k = 0, 1, 2, \ldots$.
The function of control law computation all the time is a task.

The first control law computation started at 20 ms.

So, release time of first job, $r_0 = 20$ *ms*

Subsequent release times: *120 ms, 220 ms, …*

So release time of *k*th job $r_k = (20 + 100 * k)$ *ms*

If each job needs to be completed within *70 ms,* then the absolute deadlines of the jobs: *90 ms, 190 ms, 290 ms, …*

Relative deadline $D_k = 70$ *ms*

Response Time

The time between the presentation of a set of inputs to a system and the realization of the required behavior, including the availability of all associated outputs, is called the response time of the system

In other words, **Response time** is the length of the time from the release time of the job and the time instance when it completes.

So Relative Deadline is the maximum allowable Response Time.

Real-Time Punctuality

Real Time Punctuality means every response time has an average value t_R with upper and lower bounds of $(t_R + \varepsilon_U)$ and $(t_R - \varepsilon_L)$ with $\varepsilon_U, \varepsilon_L \rightarrow 0^+$.

In all practical situations, $\varepsilon_U, \varepsilon_L \neq 0$.

It is due to the propagation delays and cumulative latencies in both hardware and software components.

So the response time will have jitter within the interval $[-\varepsilon_L , +\varepsilon_U]$.

Response Time - Example

For the door reopening operation of an elevator system, individual response times of the component involved are following.

Components	Min Response Time	Max Response Time
Sensor	5 ms	15 ms
Hardware	1 µs	2 µs
System Software	16 µs	48 µs
Application Software	0.5 µs	0.5 µs
Door drive	300 ms	500 ms
Total	**305 ms**	**515 ms**

Please note that overall response time is dominated by the response time of the door drive.

Some Other Definitions

Hard and Soft Deadlines

- A timing constraint or deadline is hard, if the failure to meet it is considered to be a fatal fault.
- A late completion of a job that has a soft deadline is undesirable. However, few misses of soft deadline don't cause serious harm.

Tardiness

- The tardiness of a job measures how late it completes respective to its deadline.
- Tardiness is 0, if it completes at or before the deadline
- Tardiness is positive and is equal to the difference between its completion time and the deadline, if the job completes after the deadline.

Usefulness

- Usefulness of a hard real-time job falls off abruptly and becomes negative when its tardiness becomes positive.
- Usefulness of a soft real-time job decreases as the tardiness increases.
- The deadline of a job is softer if its usefulness decreases at slower rate as its tardiness increases.

Usual Misconceptions

- Real-time systems are synonymous with “fast” systems
  - Many (but not all) hard real-time systems deal with deadlines in the tens of milliseconds
- There are universal, widely accepted methodologies for real-time systems specification and design
  - There is still no methodology available that answers all of the challenges of real-time specification and design all the time and for all applications
- There is no more a need to build a real-time operating system, because many commercial products exist
  - Commercial solutions have certainly their place, but choosing when to use an off-the-shelf solution and choosing the right one are continuing challenges
- Rate-monotonic analysis has solved “the real-time problem”
  - While rate-monotonic systems provide guidance in the design of real-time systems, and while there is theory surrounding them, they are not a panacea
- The study of real-time systems is mostly about scheduling theory
  - While it is scholarly to study scheduling theory, most published results require impractical simplifications and clairvoyance in order to make the theory work

Rich Variety of "Real-Time" Disciplines

***Source:** P. A. Laplante & S. J. Ovaska, Real-Time Systems Design and Analysis: Tools for the Practitioner, Wiley, 4th edition*

BITS ZG553 - Real Time System – Lecture 2

BITS Pilani
Pilani Campus

Brajabandhu Mishra

A Reference Model of Real-Time Systems

BITS Pilani
Pilani Campus

Three Elements of a Model

Each system is characterised by three elements

- **Workload model:**
  - Describes applications supported by the system
- **Resource model**
  - Resources available to the system
- **Algorithms**
  - Algorithm that defines how the system utilizes the resources at all times

Processors and Resources

All system resources are divided into two categories:

Processors

- Active resources e.g. Computers, Transmission links, Disks etc
- Denoted by P

Resources

- Passive resources e.g. Memory, Sequence numbers, Mutexes, Database locks etc
- Denoted by R

Assumptions

- Many parameters of hard real-time jobs are known at all time
  - For example, number of tasks in a hard-real time system is known at the beginning
- Rate of progress of a job depends upon the speed of the processor on which it executes
- Rate of progress of a job doesn't depend upon the resources it use during execution
- Plentiful resources (resources for which no job is prevented from execution) are omitted from the consideration
- Memory is considered as a plentiful resource, hence omitted from the consideration

Parameters associated with a Job

Job: Work done by a system e.g. computation of a control law, FFT computation etc

Each job J_i is characterized by its

- Temporal Parameters
- Functional Parameters
- Resource Parameters
- Interconnection Parameters

Timing Specifications

- **Release Time** *(r_i)*: The instance of time at which the job becomes available for execution
- **Deadline** : Deadline of a job is the instant of time by which its execution is required to be completed
- **Execution time** *(e_i)*: The amount of time required to complete the execution of the job. The actual amount of time required by a job may vary due to various reasons like conditional branches, performance enhancing features like caches and pipelines etc.
- **Response Time**: The length of time from the release time of the job to the instance when it completes
- **Relative Deadline** *(D_i)* : Maximum allowable response time
- **Absolute Deadline** *(d_i)* = Release time + Relative deadline
- **Timing Constraint** is specified in terms of its release time and relative or absolute deadlines
- **Feasible interval**: The time between the release time and the absolute deadline i.e. *(r_i, d_i]*

*Release Time, Relative Deadline and Absolute Deadline are the **Temporal Parameters** of Job J_i*

Example

A system which monitors and controls several furnaces

At the beginning, the system is initialized and starts execution (time 0 ms).
The system samples and reads each temperature sensor every 100 ms.
The system computes the control law of each furnace every 100 ms.

Each of the control law computation is a job, say J_k, $k = 0, 1, 2, \ldots$

The first control law computation started at 20 ms.

So, release time of first job, $r_0 = 20$ *ms*

Subsequent release times: *120 ms, 220 ms, …*

So release time of *k*th job $r_k = 20 + 100 * k$ *ms*

Period $p_k = 100$ *ms*

If each job needs to be completed within *70 ms,* then the absolute deadlines of the jobs: *90 ms, 190 ms, 290 ms, …*

Relative deadline $D_k = 70$ *ms*

Periodic Task Model

Well-known deterministic workload model

Each computation or data transmission is executed repeatedly at regular or semi regular intervals.

Period p_i of a periodic task T_i is the minimum length of all time intervals between release time of all jobs in it.

Execution time e_i is the maximum amount of time required to complete the execution of a task T_i when it executes alone and has all the resources it requires.

The accuracy of the periodic model decreases with increasing jitter in release time and variations in execution times.

Tasks in the system are denoted as: $T_1, T_2, T_3, \ldots, T_n$

Jobs in Task T_i are denoted as: $J_{i,1}, J_{i,2}, J_{i,3}, \ldots, J_{i,k}$

The **release times** of the jobs in Task T_i are denoted as: $r_{i,1}, r_{i,2}, r_{i,3}, \ldots, r_{i,k}$

Timing Specifications – Periodic Task

Periodic Task Model

The release time of the first job $J_{i,1}$ of Task T_i is called **Phase** Φ_i of the Task T_i i.e.

$\Phi_i = r_{i,1}$

Two tasks are said to be in phase, when they have same phase.

Hyperperiod H of a periodic task T_i is LCM of all periods of that task i.e.

$H = LCM(p_1, p_2, \ldots, p_k)$

Relative deadline (or simply **deadline**) D_i of a task T_i is the maximum allowable time for each job of the task to complete i.e. maximum response time for each job of the task.

We will often assume that for every task, a job is released and become ready at the beginning of each period and must complete by the end of the period. **Hence D_i will become equal to p_i** . However, in general D_i can have an arbitrary value less than p_i.

Absolute deadline of the job $J_{i,j}$, $d_{i,j} = r_{i,j} + D_i$

Feasible Interval is $(r_{i,j}, d_{i,j}]$

Periodic Task Model

Tasks	Φ_i	p_i	e_i	D_i	u_i
T_1	1	3	1	3	0.33
T_2	2	4	1	4	0.25
T_3	1	5	2	5	0.4

Periodic Task Model

Utilization of task T_i

$$u_i = \frac{e_i}{p_i}$$

Total Utilization

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

CPU Utilization Zones

Utilization %	Zone Type	Typical Application
< 26	Unnecessarily safe	Various
26 – 50	Very safe	Various
51 – 68	Safe	Various
69	**Theoretical limit (RM Scheduling)**	**Embedded systems**
70 – 82	Questionable	Embedded systems
83 – 99	Dangerous	Embedded systems
100	Critical	Marginally stressed system
> 100	Overloaded	Stressed system

Parameters Summary

Precedence constraints: specifies if any task(s) needs to precede other tasks.

Release or arrival time $r_{i,j}$: the release time of the jth instance of task .

Phase: Φ_i the release time of the first instant of task .

Response time: Time span between the task activation and its completion.

Absolute deadline d_i: is the instant of time by which the task must complete.

Relative deadline D_i: is the maximum allowable response time of the task.

Laxity type: Notion of urgency or leeway in a task's execution.

Period p_i: is the minimum length of intervals between the release times of consecutive tasks.

Execution time e_i: is the (maximum) amount of time required to complete the execution of a task *i* when it executes alone and has all the resources it requires.

Aperiodic and Sporadic jobs

- We call a task to be aperiodic if the jobs in it have either soft deadlines or no deadline.

  Example: Task to adjust radar's sensitivity

- We call a task to be sporadic when the jobs are released at random time instants, but have hard deadlines.

  Example: A command and control system (of airplane control) must process sporadic data messages in order to continuous voice and video traffic.

Timing Specifications – Aperiodic & Sporadic Task

Precedence Constraints, Predecessors and Successors

- Jobs are said to have precedence constraints if they are constrained to execute in some order.
- If jobs can execute in any order, they are called independent.
- A job J_i is a predecessor of another job J_k if J_k can't begin execution until the execution of J_i completes. It is denoted by

  $J_i < J_k$

  Here J_k is a successor of J_i .

  Let us consider the relation $J_i < J_j < J_k$.

  J_i is a immediate predecessor of J_j and predecessors of J_j and J_k

  J_k is a immediate successor of J_j and successor of J_i and J_j

- A job with predecessors is ready for execution when the time is at or after its release time and all of its predecessors are completed.

Precedence Graph

Precedence Constraints are represented through a graph called ‘Precedence Graph’.

How to draw a precedence graph :

- Each vertex in the graph represents a job.
- Above each vertex, the feasible interval for the job is mentioned in brackets.
- Draw a directed edge (arrowhead) from vertex J_i to J_k if J_i is a immediate predecessor of J_k.

Precedence Graph - Example

- J_1, J_2, J_3 and J_4 have feasible intervals (0, 6], (5, 8], (2, 8] and (3, 5] respectively
- J_1 and J_4 are predecessors of J_2
- J_3 is neither a predecessor, not a successor of any other jobs

Task Graph

Task Graph is like a Precedence Graph.

Each vertex represents a job, which is indicated by circles and squares.

Above each vertex, the feasible interval for the job is mentioned in brackets.

A sub-graph becomes a chain, when the jobs in the sub-graph need to be serially executed i.e. either $J_i < J_k$ or $J_i > J_k$.

Unlike precedence graph, the edges are of different types representing different types of dependencies.

The types of an edge connecting two vertices and other parameters of the edge are interconnection parameters of the jobs represented by the vertices.

Data Dependency

- In a producer-consumer environment, producer puts the data in a shared memory and the consumer consumes the data from the shared memory.
- The precedence graph will show that the producer and consumer are independent, since they are not explicitly constrained for execution.
- In task graphs, these two jobs will be connected via data-dependency edges.

Temporal Dependency

- Difference in completion time of two jobs is called temporal distance.

  Example: To have lip synchronization, the time between the display of each frame and the generation of corresponding audio must be now more than 160 ms.

- In task graph, temporal distance is represented by temporal dependency edges.
- There is a temporal dependency edge from vertex J_i to vertex J_k, if J_k has to be completed after a certain time after J_i completes. The value of this parameter is infinite if there is no temporal dependency between these jobs.

AND / OR Precedence Constraints

- An AND job is the one which can begin execution at or after its release time provided all of its predecessors has been completed.
- AND jobs are represented by unfilled circles in the task graph.
- An OR job is the one which can begin execution at or after its release time provided one or some of its immediate predecessors has been completed.
- OR jobs are represented by unfilled squares in the task graph. The fraction of the number of predecessor to be completed is indicated below the vertex e.g. if 2 out of 3 jobs are to be completed, then put 2/3 there.

Conditional Branches

- The job from which the conditional branch arises is called a branch job.
- The job at which the conditional branches join is called a join job.
- Both these jobs are represented as filled circles in the task graph.
- The sub-graph that begins from the branch job and ends at the join job is called conditional block.

Pipeline Relationship

- Pipelined relationship is used when between a pair of producer-consumer jobs, data can be piped.
- In the task graph, the vertices are the granules of the producer and the consumer.
- Each granule of the consumer can begin execution when the previous granule of the job and the corresponding granule of the consumer job is completed.
- This relationship is represented by a dotted arrow in the task graph.

Task Graph - Example

Functional Parameters

- Preemptivity
- Criticality
- Optional Interval
- Laxity type

Preemptivity of Jobs

- A job is preemptable if its execution can be suspended at any point of time to allow execution of other jobs and later its execution can be resumed from the point of suspension.
- A job is nonpreemptable if it must be executed from start to end without any interruption.

Example: Interrupt Handling is non-preemptable.

Context Switching

- During preemption, the system must first save the state of the preempted job, so that it can resume the execution of it later. The state of a job typically includes
  - Program Counter Register
  - Stack Pointer Register
  - Other General Purpose Registers
  - …
- After doing this, the system must load new processor status register, initialize the program counter, clear the processor pipeline etc.
- These actions are called context switching.
- Time taken for context switching is called context switch time.

Criticality of Jobs

- The criticality or importance of a job, is a positive number, which indicates how critical the execution of the job is, for the system.
- Priority and Weights are often used to refer to the criticality.
- During overload, when it is not possible to schedule all the jobs to meet their deadline, less critical jobs are sacrificed.

Optional Execution

- A job or a portion of a job can be called optional, if it completes late or not executed at all, the system performance may degrade but nevertheless the function is satisfactory.
- During overload, when it is not possible to complete all the jobs on time, we may chose to discard the optional jobs (i.e. leave them unexecuted or partially executed).
- The jobs and portion of the jobs which are not optional, are mandatory.

Laxity Type

- Laxity type of a job indicates whether the time constraints are hard or soft.
- It is represented by a 'usefulness function', which gives the usefulness of the result produced by the job as the function of its tardiness.
- If the usefulness of a job becomes zero or negative as soon as the job becomes tardy, then it is better not to execute that job rather than completing it late.

Resource Parameter

Preemptivity of Resources

- A resource is nonpreemptible if each unit of the resource is constrained to be used serially. It means if a nonpreemptible resource is allocated to a job, other jobs needing it must wait until the current jobs complete its use.
- If jobs can use every unit of the resource in an interleaved manner, then the resource is preemptible.

Feasibility, Optimality and Performance Measures

- A valid schedule is feasible schedule if every job completes by its deadline.
- A set of job is schedulable according to a scheduling algorithm if when using the algorithm the scheduler always produces a feasible schedule.
- A hard real-time scheduling algorithm is said to be optimal if using this algorithm the scheduler always produces a feasible schedule if the given set of jobs has feasible schedule.
- Other than the above other performance measures used are: Maximum and average
  - Tardiness,
  - Lateness,
  - Response time,
  - Miss, Loss and Invalid rates.

Feasibility, Optimality and Performance Measures (contd.)

- Lateness of a job is the difference between its completion time and the deadline (i.e. completion time – deadline). It can become negative if the job completes early and positive it completes late.
- For a set of jobs in the system, the response time of the job which completes last, is called makespan of the schedule.
- **Makespan is used to compare scheduling algorithm performance. The algorithm which produces a schedule with a shorter makespan is better.**
- **Most frequently used performance measure for jobs that have soft deadlines is their average response time**.
- For some soft real-time systems, it is acceptable to complete some jobs late or to discard late jobs. For those systems, a suitable performance measure include the miss rate (% of jobs those complete late) or loss rate (% of jobs those are discarded)
- Invalid rate = Loss rate + Miss rate : It should be minimized

BITS ZG553 - Real Time System – Lecture 3

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Commonly Used Approaches to Real-Time Systems

BITS Pilani
Pilani Campus

Three Approaches

- Clock-driven
- Round-robin / Weighted round-robin
- Priority-driven

Clock-driven Approach

- Scheduling decisions are made at specific time instants, which are chosen a priori before the system begins execution
- Typically this type of scheduling is suitable for hard real-time systems, where the parameters are fixed and known.
- Scheduling decisions are computed off-line and stored for use at run-time, thus scheduling overhead is minimal.
- Generally a hardware time is set to expire periodically.
- After the system gets initialized, the scheduler selects and schedules jobs which execute till the next scheduling decision is made. Then the scheduler blocks itself waiting for the expiration of the timer.
- When the timer expires, the scheduler wakes up, does necessary scheduling and sleeps again. This process repeats.

Round-robin Approach

- Also known as time-sharing
- Every job joins a FIFO (First-in-first-out) queue when it becomes ready for execution
- The entire time period is divided into several time-slices
- The job at the head of the queue executes for one time-slice.
- If the job doesn't complete at the end of the time-slice, it gets pre-empted and placed at the end of the queue to waits for its next turn.
- If there are '*n*' jobs ready for execution, each job gets *1/n*th share of the processor.

Round-robin Approach - Example

(Round-robin execution of two tasks on a single processor)

(Round-robin execution of two tasks on two processors)

Weighted round-robin Approach

- This approach is a round robin approach with different weights assigned to different jobs.
- If a job has weight *'wt'*, then it will get '*wt*' time slices every round for execution.

Priority Driven Approach

- Also known as greedy scheduling, list scheduling and work-conserving scheduling
- Priorities are assigned to the jobs based on their criticality
- Jobs ready for execution are placed in one or more queues ordered by priorities of the jobs.
- At any scheduling decision time, the jobs with the highest priorities are scheduled and executed on the available processors.
- Most scheduling algorithms used in non-real-time systems are priority driven.
  - FIFO (First In First Out)
  - LIFO (Last In First Out)
  - SETF (Shortest Execution Time First)
  - LETF (Longest Execution Time First)
- For jobs of same priority, round-robin scheduling is used

Priority-driven Scheduling Example

Rules:

- each process has a fixed priority (1 highest);
- highest-priority ready process gets CPU;
- process continues until done.

Processes

- J1: priority 1, release time 15, execution time 10
- J2: priority 2, release time 0, execution time 30
- J3: priority 3, release time 18, execution time 20

Priority-driven Scheduling Example

Effective Release Time and Deadline

- It may happen that a job has release time later than that of its successor and deadline earlier than that of its predecessor.
- Hence it may not be useful to work with individual jobs release time and deadlines.

Effective Release Time

- Effective Release Time of a job without any predecessor is its given release time.
- Effective Release Time of a job with predecessors is maximum of its given release time and the effective release times of its predecessors.

Effective Deadline

- Effective Deadline of a job without any successor is its given deadline.
- Effective Deadline of a job with successors is minimum of its given deadline and the effective deadlines of its successors.

By release time and deadline, we will mean effective release time and effective deadlines respectively.

Off-line vs On-line scheduling

Off-line scheduling: When the schedule is pre-computed and kept, it is called off-line scheduling.

- Example: Clock-driven scheduling
- It is possible only when the system parameters are known a priori
- Advantages: Deterministic timing behaviour, Lesser complexity, Very less scheduling overhead

On-line scheduling: When the scheduler makes each scheduling decision without knowledge about the jobs that will be released in the future

- Example: Priority driven scheduling
- It is the only option when future workload is unpredictable
- The price of the flexibility and adaptability is a reduced ability for the scheduler to come up with an optimal schedule making the best use of the system resources

Priority Driven vs Clock Driven approaches

- Priority driven approaches have many advantages compared to clock driven approach:
  - They don't have to have the information on the release time, execution time etc (in contrast with clock driven approach, where these parameters are required to be known a priori)
  - It is best suited for applications with varying time and resource requirements
  - Many well-known priority –driven algorithms use very simple priority assignments reducing the overhead of maintaining multiple queues.
- Despite all these advantages, Clock-driven approaches are used for hard real-time systems, especially in safety-critical systems.
  - **The major reason is that the timing behaviour of a priority-driven system is nondeterministic when job parameters vary.**
  - Consequently it is difficult to validate the deadlines of all the jobs in a priority-driven approach that they meet the deadline, when job parameters vary.

Overload condition

- A system is said to be overloaded when the job offered to the scheduler can't be feasibly scheduled by a clairvoyant scheduler.
- When the system is not overloaded, an optimal on-line scheduling algorithm is one that always produces a feasible schedule of all offered job.
- No optimal on-line scheduling algorithm exists when some jobs are non-preemptable.
- During an overload, some jobs must be discarded in order to allow other jobs to complete on time.

EDF (Earliest Deadline First) Algorithm

- Job with earliest (absolute) deadline has highest priority (Process closest to its deadline has highest priority)
- Priorities is assigned dynamically, since deadlines of the jobs varied. So it is a *dynamic-priority algorithm.*

<u>Example</u>

(Execution times are mentioned before the feasible intervals)

- t = 0: J1 is released, no other job in the system, so gets scheduled
- t = 2: J3 is released.
  - Deadline of J1 = 6,
  - Deadline of J3 = 8.
  - So J1 has higher priority, hence it continues
- t = 3: J1 completes, J3 starts.
- t = 5: J2 is released
  - Deadline of J2 = 8
  - Deadline of J3 = 8
  - So both have same priority, let J3 continue.
- t = 6: J3 is done, J2 gets scheduled
- t = 8: J2 is done

Optimality of EDF Algorithm

Theorem:

When preemption is allowed and jobs don't contend for resources, the EDF algorithm can produce a feasible schedule if a set $\mathbf{J}$ of jobs with arbitrary release times and deadlines on a processor if and only if $\mathbf{J}$ has feasible schedules.

LRT (Latest Release Time) Algorithm

- There is no advantage in finishing early in a hard-real time system.
- LRT Algorithm takes the advantage of this fact.
- It treats release time as deadline and deadline as release time and schedule the jobs backward starting from latest deadline in 'priority-driven' manner.

Example

- Latest deadline is 8. So start scheduling backward from time 8.
- J2 should start at time 6, so that it completes at time 8 (=6 + execution time 2)
- At time 7, J3 is ready to be scheduled (its deadline is 7). So it gets scheduled at time 6 (after J2).
- J3 should start at time 4, so that it completes at time 6 (=4 + execution time 2)
- At time 6, J1 is ready to be scheduled. So it gets scheduled at time 4 after J3.
- J1 should start at time 1, so that it completes at time 4 (=1 + execution time 3)

LST (Least Slack Time First) Algorithm

- Also known as **Minimum-Laxity-First (MLF) Algorithm**
- **Slack (or laxity**) of a job at time 't' = 'd – t – time required to complete the remaining portion of the job', where 'd' is the deadline
- Smaller the slack, higher the priority.
- As long as a job is executing the slack at time 't' = d – t – (r + e – t) = d – r - e,

  So slack is same as long as the job is executing. It changes after preemption of the job

<u>Example</u>

(Execution times are mentioned before the feasible intervals)

- t = 0: J1 is released, no other job in the system, so gets scheduled
- t = 2: J3 is released.
  - Slack of J1 = 6 – 2 – (3 – 2) = 3,
  - Slack of J3 = 8 – 2 – 3 = 3.
  - So both J1 and J3 are of same priority. Let J1 continue
- t = 3: J1 completes, J3 starts.
- t = 5: J2 is released
  - Slack of J2 = 8 – 5 – 2 = 1
  - Slack of J3 = 8 – 5 – (3 – 2) = 2
  - So J2 will have higher priority, hence J2 will preempt J3.
- t = 7: J2 is done, J3 gets scheduled
  - Slack of J3 = 8 – 7 – (3 – 2) = 0
- t = 8: J3 is done

Exercise

The feasible interval of each job in the precedence graph is given next to its name. The execution time of all jobs are equal to 1.

a) Find out the effective release times and deadlines of the jobs

b) Find an EDF schedule of the jobs

Exercise – Answer (a)

Effective Release Times

- J1, J4 and J5 has no predecessors. So effective release times for these jobs are same as their own release times.
- Then traverse the diagram from left.
- Effective release time of other jobs = Max (own release time, effective release times of the predecessors)
  - Effective Rel Time (J2) = Max (0, 1, 1) = 1
  - Effective Rel Time (J7) = Max (1, 1) = 1
  - Effective Rel Time (J3) = Max (0, 1, 3) = 3
  - Effective Rel Time (J6) = Max (2, 3, 3) = 3
  - Effective Rel Time (J8) = Max (1, 3) = 3
  - Effective Rel Time (J9) = Max (1, 3) = 3

Jobs	Effective Release Times
J1	0
J2	1
J3	3
J4	1
J5	3
J6	3
J7	1
J8	3
J9	3

Exercise – Answer (a)

Effective Deadlines

- J3, J6 and J9 has no successors. So effective deadlines for these jobs are same as their own deadlines.
- Then traverse the diagram from right.
- Effective deadline of other jobs = Min (own deadline, effective deadlines of the successors)
  - Effective Deadline(J8) = Min (12, 12) = 12
  - Effective Deadline(J2) = Min (4, 5, 10) = 4
  - Effective Deadline(J5) = Min (9, 5, 10, 12) = 5
  - Effective Deadline(J7) = Min (12, 12) = 12
  - Effective Deadline(J4) = Min (6, 4, 12) = 4
  - Effective Deadline(J1) = Min (10, 4) = 4

Jobs	Effective Deadlines
J1	4
J2	4
J3	5
J4	4
J5	5
J6	10
J7	12
J8	12
J9	12

Exercise – Answer (b)

EDF Schedule

- Time 0: J1 is released and scheduled.
- Time 1: J2, J4, J7 gets released. J2 and J4 have same deadlines, which is earlier than that of J7. But J4 is predecessor of J2. Hence we have to schedule J4.
- Time 2: Out of J2 and J7, J2 has earlier deadline i.e. 4, so gets scheduled.
- Time 3: All other jobs gets released. J3 and J5 have earlier deadlines i.e. 5. But J5 is predecessor of J3. So we have to schedule J5.
- Time 4: J3 gets scheduled, since it has the earliest deadline i.e. 5.
- Time 5: J6 gets scheduled, since it has the earliest deadline i.e. 10.
- Time 6, 7, 8: J7, J8, J9 have same deadlines i.e. 12, so same priority. But J7 is predecessor of J8 and J8 is predecessor of J9. So they have to be scheduled in the order J7, J8 and J9.

Jobs	Effective Release Times	Effective Deadlines
J1	0	4
J2	1	4
J3	3	5
J4	1	4
J5	3	5
J6	3	10
J7	1	12
J8	3	12
J9	3	12

BITS ZG553 - Real Time System – Lecture 4

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Clock-Driven Scheduling

BITS Pilani
Pilani Campus

Assumptions

- There are '*n*' periodic tasks in the system and '*n*' is fixed
- The parameters of all periodic tasks are known a priori.
- Variation in the inter-release time is negligibly small. For all practical purpose each job in T_i is released p_i units of time after the previous job in T_i .
- Each job $J_{i,k}$ is ready for execution at release time $r_{i,k}$

A periodic task T_i is characterised by the **4-tuple (Φ_i, p_i, e_i, D_i).**

Example: (1, 10, 3, 6) periodic task

- 1st job is released at time 1. It must completes by time 7 (=1 + 6). It executes for 3 units of time.
- 2nd job is released at time 11 (= 1+ 10). It must complete by time 17 (=11 + 6). It executes for 3 units of time.
- ....

- Tuples having default values are omitted.
- By default, the phase of each task is 0 and relative deadline is equal to the period.

Example: (10, 3, 6) and (10, 3) have zero phases and relative deadlines of 6 and 10 respectively.

Example

- Consider 4 tasks: T1 = (4,1);T2 = (5, 2); T3 = (20, 1); T4 = (20, 1)
- Hyperperiod = LCM (4, 5, 20, 20) = 20 -> Schedules can repeat every 20 time units
- Max no of jobs in each hyperperiod (No of entries in the schedule) = (20 / 4) + (20 / 5) + (20 /20) + (20 /20) = 5 + 4 + 1 + 1 = 11

T1				T1				T1				T1				T1				T1
T2	T2				T2	T2				T2	T2				T2	T2				T2
T3																				T3
T4																				T4
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

T1	T3	T2	T2	T1	T2	T2	T4	T1		T2	T2	T1				T1	T2	T2		T1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

(An arbitrary static schedule)

A Clock-Driven Scheduler

- Whenever the parameters of the jobs are known a priori (As in case of Hard Real-time Systems), it is better to construct a static schedule offline.
- We call a periodic static schedule a ***cyclic schedule*** as well.
- Store the pre-computed schedule in a table
- Each entry $(t_k, T(t_k))$ in the table gives a time t_k, and the task T_k or I to be scheduled at that time.
- T_k is a periodic job and I is idle interval.
- There should be a timer, which expires at the decision times t_k and raises an interrupt as it expires.
- This interrupt wakes up the scheduler, which schedules the appropriate job from the off-line schedule.
- If the job to be scheduled is I, then it is the idle time (no periodic job is available to be scheduled). Hence the job at the head of the aperiodic queue is scheduled.

General Structure of Cyclic Schedules

- A cyclic scheduler repeats a pre-computed schedule.
- The pre-computed schedule needs to be stored only for one major cycle.
- Usually the major cycle is the Hyperperiod.
- Each task in the task set to be scheduled repeats identically in every major cycle.
- The major cycle is divided into one or more frames
- The scheduling points of a cyclic scheduler occur at frame boundaries. This means that a task can start executing only at the beginning of a frame.
- The frame boundaries are defined through the interrupts generated by a periodic timer. Each task is assigned to run in one or more frames.

Tasks	Frame
T_1	f_1
T_3	f_2
T_5	f_3
T_4	f_4
I	f_5

A Clock-Driven Scheduler

H = Length of the hyperperiod
N = Number of entries in the schedule of each hyperperiod
I = Idle Interval
Input: Stored schedule $(t_k, T(t_k))$, for $k = 0, 1, \ldots, N-1$

```
Task: SCHEDULER:
   set the next decision point i and table entry k to 0;
   set the timer to expire at t_k
   do {
   accept the timer interrupt;
   if(an aperiodic job is executing) preempt it;
   current task T = T(t_k)
   i = i++;
   compute the next table entry k = i mod(N)
   set the timer to expire at  ⌊i/N⌋  H + t_k
   if(the current task T == I)
   schedule the job at the head of the aperiodic queue
   else
   schedule the task T
   }
End SCHEDULER
```

General Structure of Cyclic Schedules

Frames

- Scheduling time is divided into multiple frames of size *'f'*.
- Scheduling is done at starting of each frame, not at any other time.
- Hence there is no preemption within a frame.
- The phase of each periodic task is a nonnegative integer multiple of a frame. In other words, the first job of every task is released at the beginning of some frame.
- Apart from scheduling decision, schedule carries out monitoring and enforcement actions at the beginning of each frame.

Floor and Ceiling Functions

<u>Floor</u>

floor(x) = $\lfloor x \rfloor$ is the largest integer not greater than x

<u>Ceiling</u>

ceiling(x) = $\lceil x \rceil$ is the smallest integer not less than x

Example:

x	$\lfloor x \rfloor$	$\lceil x \rceil$
2.4	2	3
5.5	5	6
-2.1	-3	-2
-2	-2	-2

Size of a Frame

A selected frame size should satisfy the following three constraints.

1. **Minimum Context Switching**
   - To avoid unnecessary context switches, the selected frame size should be larger than the execution time of each task, so that when a task starts at a frame boundary it should be able to complete within the same frame.
   - It means, the frame size should be larger than maximum execution time.
2. **Minimization of Table Size**
   - This constraint requires that the number of entries in the schedule table should be minimum
   - Unless a frame divides the major cycle, the number of entries in a major cycle will not be sufficient for the schedule.
   - Hence the frame should divide the major cycle i.e. Hyper period of the tasks.

Size of a Frame

3. **Satisfaction of Task Deadline**
   - At least one frame should be there between the time job is released and its deadline.
   - Otherwise it will miss the deadline, just because it gets scheduled late in the next frame boundary (as shown in the diagram)

These three conditions are formulated into three equations in the next slides.

Frame Size Constraints

Minimum Context Switching

- Frame size should be larger than maximum execution time, i.e.

$$f \geq \max_{1 \leq i \leq n}(e_i) \quad \text{... (1)}$$

Minimization of Table Size

- Length of a major cycle is equal to the Hyperperiod H.
- The frame size should be as short as possible so that there are integer number of frames in an Hyperperiod *H*. For this it needs to divide the Hyperperiod, which in turn implies that it should divide the period p_i of at least one task T_i. i.e.

$$\lfloor p_i / f \rfloor - p_i / f = 0 \quad \text{... (2)}$$

- Let *F* = No of frames in an Hyperperiod (should be an integer, as per the previous equation)
- The Hyperperiod, which begins at the beginning of *(kF + 1)*st frame, is called a '*major cycle*', for *k = 0, 1, …*

Satisfaction of Task Deadline

- Hence frame size should be small enough so that between the release time and deadline of a job, there should be at least one frame.

Frame Size Constraints

t = Beginning of the frame in which a job in T_i is released

t' = Release time of the job

If(t' > t), we need frame k+1 to be in t' and t'+D_i.

For this to happen,

$t + 2f \leq t' + D_i$

$\Rightarrow 2f - (t' - t) \leq D_i$

t' is an integer multiple of period p_i *(say* lp_i*)* and *t* is an integer multiple of frame *f (say kf).*

So, $t' - t = lp_i - kf$, where *k, l* are positive integers.

$kf - lp_i$ will be at least be equal to GCD of *f* and p_i.

Hence $t' - t \geq gcd(f, p_i)$

Then the above inequality becomes

$$2f - gcd(f, p_i) \leq D_i \quad ...(3)$$

Frame Size Constraints

For $t = t'$, we need $f \leq D_i$ for a frame to be included between t' and D_i

If eq. (3) is satisfied, then above condition will automatically be satisfied, since $gcd(f, p_i) < f$.

So eq. (3) is sufficient for both the conditions $t = t'$ and $t < t'$.

Equations (1), (2) and (3) are referred as 'frame size constraints'.

Multiple frame sizes satisfying the Frame size constraints

- For a given task set it is possible that more than one frame size satisfies all the three constraints.
- In such cases, it is better to choose the shortest frame size.
- This is because of the fact that the schedulability of a task set increases as more frames become available over a major cycle.

Frame Size Constraints - Example

Exercise 5.1 (a)

Find out an appropriate frame size for the following system of periodic tasks to be scheduled executed according to a cyclic schedule.

{(6,1), (10,2) and (18,2)}.

Solution:

Max execution time = 2.

So frame size f ≥ 2 ... (1)

Frame size should divide at least one period.

So, f = 2,3,5,6,10,18 ... (2)

2f – gcd(f, p_i)≤ Di

So for T1, 2f – gcd(f, 6) ≤ 6

⇒f = 1, 2, 3, 4, 6

For T2, 2f – gcd(f, 10) ≤ 10

⇒f = 1, 2, 3, 4, 5, 6, 10

For T3, 2f – gcd(f, 18) ≤ 18

⇒f = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12

Hence values of f, which will satisfy 2f – gcd(f, p_i)≤ Di

f = 1,2,3,4,6 ... (3)

The values of 'f', which will satisfy eq (1), (2) and (3) are: f = 2,3,6

Hence, possible values for frame size are 2, 3 and 6.

So choose 2 (since it is smallest) as the frame size.

Tasks	Period / Deadline p_i / D_i	Execution time e_i
T1	6	1
T2	10	2
T3	18	2

Job Slices

Example

Find out an appropriate frame size for the following system c
periodic tasks to be scheduled executed according to a cycli
schedule.

{(4,1), (5,2,7) and (20,5)}

Solution:

Max execution time = 5.

So frame size f ≥ *5* ... (1)

Frame size should divide at least one period.

So, f = 1,2,4,5,10,20 ... (2)

2f – gcd(f, p_i)≤ Di

So for T1, 2f – gcd(f, 4) ≤ 4

⇒f = 1, 2, 4

For T2, 2f – gcd(f, 5) ≤ 7

⇒f = 1, 2, 3, 4, 5

For T3, 2f – gcd(f, 20) ≤ 20

⇒f = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20

Hence values of f, which will satisfy 2f – gcd(f, p_i)≤ Di

f = 1,2,4 ... (3)

Satisfying eq (1) and (3) is not possible.

Tasks	Period p_i	Deadline D_i	Execution time e_i
T1	4	4	1
T2	5	7	2
T3	20	20	5

So it is better, if we can slice the job with larger execution time T3 to 3 subtasks (a job of computation can be broken into multiple procedures, a message transmission job can be broken into sending multiple smaller messages etc) so that all frame size constraints gets satisfied.

In this case, if we slice T3 as, T3 = {(20,1), (20, 3), (20, 1)} , then eq (1) will become f ≥ 3. So frame size constrains will be satisfied e.g. f = 4.

This process is called ***job slicing***.

Process of Constructing Cyclic Schedules

Process

- Choosing frame size
- Partitioning the jobs into slices
- Placing the slices in the frames

Tasks	Period p_i	Deadline D_i	Execution time e_i
T1	4	4	1
T2	5	7	2
T3.1	20	20	1
T3.2	20	20	3
T3.3	20	20	1

For the previous example, we have

- Partitioned T3 into 3 job slices.
- Fixed the frame size f = 4
- Now we have to create the schedules by putting the jobs and job slices into fames.

Cyclic Executive

- "Cyclic Executive" refers to a scheduler that deterministically interleaves and sequentializes the execution of periodic tasks on a CPU according to a given cyclic schedule.
- Each job slice is a 'procedure'
- A cyclic executive executes in a single do loop, beginning of each frame and executes the slices scheduled within a frame.
- The stored table that gives the precomputed cyclic schedule has *F* entries, where *F* is the number of frames per major cycle (please note that the schedule will repeat in every major cycle, hence storing the schedule for one major cycle is enough)
- Each entry is called a 'scheduling block'. A scheduling block for *kth* frame is denoted by *L(k)* and it stores the job slices to be scheduled in *kth* frame.
- Usually a 'periodic task server' executes the job slices of a frame.
- Upon completion of executing the periodic job slices scheduled in the current frame, if there is time remaining in the current frame, the cyclic executive schedules aperiodic jobs.

Cyclic Executive

F = Number of frames in a major cycle

L(k) = Names of the job slices that are scheduled to be executed in frame *k*, called *'scheduling block'*

```
Input: Stored schedule: L(k), k = 0, 1, …, F-1
     Aperiodic job queue
Task: SCHEDULER:
   current time t = 0;
   current frame k = 0;
   do {
   accept the timer interrupt at time t_f;
   current block = L(k);
   t = t + 1;
   k = t mod(F);
   if last job is not completed take appropriate action;
   if any of the job slices are not released, take appropriate action;
   execute the slices in the current block by waking of the periodic task server;
   sleep until the periodic task server completes;
   while (aperiodic job queue is not empty) {
   wake up the job at the head of the aperiodic job queue and schedule it;
   sleep until the aperiodic job completes;
   remove the aperiodic job from the queue;
   }
   sleep until next timer interrupt arrives;
   }
End SCHEDULER
```

Scheduling Aperiodic Jobs

- Response time of aperiodic jobs is an important criterion
- Sooner the aperidic job completes, the more responsive the system is.
- Hence improving average response time of all aperiodic jobs is a design goal
- One of the way is '*Slack Stealing*'

Slack Stealing

- There is no advantage completing a periodic task much before the deadline.
- Hence execute the aperiodic jobs ahead of periodic jobs whenever possible
- Every periodic job slice is scheduled in a frame that ends no later than its deadline.
- If the total time allocated to the job slices in a frame (kth frame) is x_k , the slack (time) available in the frame is equal to $(f- x_k)$.
- If the aperiodic job queue is not empty, then aperiodic jobs can be scheduled at the beginning of the frame for the slack i.e. $(f- x_k)$.
- If the aperiodic job queue is empty, then job slices from the next frame can be executed in the current block.

Example – Slack Stealing

- Above is the major cycle of a cyclic executive schedule.
- Three aperiodic jobs A1, A2, A3 arrived as shown in the table.

Aperiodic jobs	Release time	Execution time
A1	4	1.5
A2	9.5	0.5
A3	10.5	2

Example – Slack Stealing (contd.)

Cyclic Executive Schedule without Slack Stealing

- Time 4: A1 is released
- Time 7: Slot is available, so A1 is scheduled
- Time 8: Next frame started, so A1 is preempted
- Time 9.5: A2 is released
- Time 10: A1 is scheduled
- Time 10.5: A1 completes, so A2 is scheduled
- Time 10.5: A3 is released
- Time 11: A2 completes, so A3 is scheduled
- Time 12: Next frame started, so A3 is preempted
- Time 15: A3 is schedules
- Time 16: A3 completes.

Response time of A1 = 10.5 – 4 = 6.5

Response time of A2 = 11 – 9.5 = 1.5

Response time of A3 = 16 – 10.5 = 5.5 Average response time = (6.5 + 1.5 + 5.5) / 3 = 4.5

Aperiodic jobs	Release time	Execution time
A1	4	1.5
A2	9.5	0.5
A3	10.5	2

Example – Slack Stealing (contd.)

Cyclic Executive Schedule with Slack Stealing

- Time 4: A1 is released, scheduler finds 1 slot slack in frame 2, so schedules A1
- Time 5: A1 is preempted and periodic job slices are scheduled
- Time 8: Next frame started, scheduler finds 2 slots of slack, so schedules A1
- Time 8.5: A1 is completed, so periodic job slices of frame 3 are scheduled
- Time 9.5: A2 is released, the scheduler calculates remaining slack 1.5, so schedules A2.
- Time 10: A2 is completed, so remaining job slices of frame 3 are scheduled
- Time 10.5: A3 is released
- Time 11: Job slices for frame 3 are completed, the scheduler calculates the remaining slack as 1 time slot, so A3 is scheduled
- Time 12: Next frame started, the scheduler calculates the slack as 1 time slot, so A3 continues to execute
- Time 13: Job slices of frame 4 should be scheduled. A3 also completes at this time

Aperiodic jobs	Release time	Execution time
A1	4	1.5
A2	9.5	0.5
A3	10.5	2

Response time of A1 = 8.5 – 4 = 4.5

Response time of A2 = 10 – 9.5 = 0.5

Response time of A3 = 13 – 10.5 = 2.5 Average response time = (4.5+0.5+2.5) / 3 = 2.5

Scheduling Sporadic Jobs

- Scheduler performs an acceptance test for the sporadic jobs before accepting them.
- Acceptance test is done at the frame boundaries.
- During *Acceptance Test*, the scheduler checks if the newly released sporadic job can be feasibly scheduled with all the jobs in the system (all the jobs means either the periodic jobs for which time has been allocated or sporadic jobs scheduled but not yet completed).
- If there is sufficient amount of time in the frames so that the newly released sporadic job can be scheduled and completes before its deadline without causing all the jobs in the system miss their deadlines, then the scheduler accepts the job. Otherwise it rejects it.
- We assume that maximum execution time of the sporadic jobs are known upon their releases.
- We also assume that all sporadic jobs are preempteble. Therefore a sporadic job can execute in more than one frames if no frame has sufficient time to accommodate the entire job.

Acceptance Test of Sporadic Jobs

- Let us suppose that at the beginning of frame 't', an acceptance test is done on a sporadic job 'S' with deadline 'd' and execution time 'e'.
- Suppose that the deadline 'd' of S is in the frame '$l+1$', and $l > t$.
- Then the job must be scheduled in the lth or earlier frames.
- Hence the job can complete on time only if the current (total) amount of slack time in frames $t, t+1, \ldots, l$

$$\sigma_c(t, l) \geq e$$

- The scheduler accepts the job if above condition is satisfied.

EDF Scheduling of Accepted Jobs

- EDF algorithm is good way to schedule sporadic jobs.
- Scheduler maintains a queue of accepted sporadic jobs in non decreasing order of their deadlines.
- Inserts newly accepted jobs into this queue in this order.
- Whenever all slices of the periodic tasks scheduled in each frame is completed, the scheduler schedules the sporadic job from head of the queue
- The scheduler allows aperiodic jobs to execute only if the accepted sporadic job queue is empty.

EDF Scheduling of Sporadic Jobs - Example

Sporadic jobs	Release time	Deadline	Execution time
S1	3	17	4.5
S2	5	29	4
S3	11	22	1.5

EDF Scheduling of Sporadic Jobs – Example (Contd)

- Time 3: S1 is released. Its deadline falls in frame 5. Acceptance test is done at starting of frame 2, since it is released at time 3, which falls inside frame 1.Total slack time between frame 2 and frame 4 is 4. Hence this job was rejected, since its execution time is 4.5, which is greater than total available slack.
- Time 5: S2 was released. Acceptance test is done at start of frame 3. Frames 3 through 7 ends before its deadline, having total slack time 5.5. So S2 was accepted.
- Time 10: S2 was scheduled, since there was a slack time.
- Time 11: S3 was released. Acceptance test is done at start of frame 3. Frame 3 to 5 ends before its deadline, having total slack time 2. So the scheduler accepts S3.
- Time 12: S2 is prempted because new frame starts and job slices gets scheduled. Since S3 has earlier deadline then S2, S2 is put after S3 in the sporadic job queue.
- Time 15: S3 is scheduled, since job slices of frame 4 are done.
- Time 16: S3 is prempted because next frame started and its job slices are scheduled.
- Time 19: S3 is scheduled, since job slices of frame 4 are done.
- Time 19.5: S3 is completed, so S2 is scheduled.
- Time 20: S2 is prempted because next frame started and its job slices are scheduled.
- Time 23.5: S2 is scheduled, since job slices of frame 6 are done.
- Time 24: S2 is prempted because next frame started and its job slices are scheduled.
- Time 27: S2 is scheduled, since job slices of frame 7 are done.
- Time 28: S2 is completed

Practical Considerations

- Handling Frame Overruns
  - Unexpected reasons like a transient h/w or a s/w bug may cause a job to execute longer than expected and cross the frame boundary.
  - A way to handle it is to simply abort the overrun job at the beginning of the next frame. Such a fault can be handled by a recovery mechanism.
  - Another way (which is used most frequently)
    - To preempt the overrun job immediately if it is not in critical section or once it exists from the critical section.
    - The unfinished job is scheduled later as an aperiodic job.

Practical Considerations

- Mode Changes
  - During mode change, the system is reconfigured, some of the old tasks may be deleted, some new tasks may come.
  - Assume that the timing parameters are known a priori for the tasks in the new mode as well.
  - Mode change can be accomplished in two ways:
  - Consider it as an Aperiodic job
    - Mode change task is executed at highest priority among all aperiodic jobs.
    - If there are other aperiodic jobs, their execution is delayed after the mode change.
    - If there are sporadic jobs, delay the switching over to new schedule till the sporadic jobs are completed.
    - Acceptance test is suspended and the mode changer is scheduled during the time allocated for the deleted periodic tasks.
    - Once the mode changer task is executed, the acceptance test resumes at the start of the next frame.
  - Consider it as an Sporadic job
    - Mode change task is executed as any other Sporadic job.

- General Workloads and Multiprocessor Scheduling
  - Clock driven approach can be used to do scheduling among multiple processors in a multi-processor system, whenever the workload parameters are known a priori.

Pros & Cons of Clock-Driven Scheduling

Advantages

- Conceptual Simplicity: Since the schedule is statically determined,
  - Deadlocks can be avoided during scheduling
  - There will be no unpredictable delays
  - No need of concurrency control
  - No need for any synchronization mechanism
- Suitable for hard real-time systems, where all the parameters are known a priori

Disadvantages

- Release times of all jobs must be fixed
- All combinations of periodic tasks that might execute at the same time must be known a priori, so a schedule for the combination can be computed.
- Not suitable for systems which are combination of both hard real-time and soft real-time systems.

Real Time System – Lecture 5

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Priority-Driven Scheduling of Periodic Tasks

BITS Pilani
Pilani Campus

Assumptions

- Tasks are independent
- There are no aperiodic and sporadic tasks
- Every job is ready for execution as soon as released
- A job can be preempted any time
- A job never suspends itself
- Scheduling decisions are made immediately upon the job releases and completions
- Context switch overhead is negligibly small compared with the execution times of the tasks
- Number of priority levels are unlimited
- Number of periodic tasks are fixed

When an application creates a new task,

- The application first requests the scheduler to add a new task by providing the scheduler with relevant parameters of the task including period, execution time and relative deadline.
- Then the scheduler does acceptance test to check if the new task can be feasibly scheduled with all other existing tasks. If it is not feasible, the scheduler rejects the task.

Fixed Priority vs Dynamic Priority Algorithms

- A priority driven scheduler is an on-line scheduler. It doesn't pre-compute the schedule.
- It assigns priorities to jobs after they are released, and places them in appropriate queues.

- Fixed priority algorithm: A fixed priority algorithm assigns the same priority to all the jobs in each task.
- Dynamic priority algorithm: A dynamic priority algorithm assigns different priorities to the individual jobs in each task. So priorities of a task can change as the jobs of that task are released.

Generally the real time algorithms of practical interest are fixed priority algorithms.

Rate Monotonic (RM) Algorithm

- Fixed priority algorithm
- **Shorter the period, higher the priority**
- Rate is inverse of the period. Hence higher the rate, higher the priority. – so the name *'rate monotonic'*

Example: 3 tasks T1 = (4,1), T2 = (5, 2), T3 = (20, 5) to be scheduled based on RM algorithm.

- T1 has shortest period (i.e. 4), so should have higher priority, followed by T2 and T3.
- T1 get scheduled at after 4 time units i.e at times 0, 4, 8, 12, 16, 20, ...
- T2 gets scheduled at time units 1, 5, 11 for 2 time slots. When T2 gets released at time 15, it is given this slot, preempting T3. But at time 16, T1 gets released. It gets scheduled preempting T2. Once T1 is done, T2 again gts scheduled at time 18.
- T3 gets scheduled in the remaining slots 3, 7. It gets time slot 9, since T1 is done and T2 is not released that time. But it gets preempted in the next slot because T2 gets released. Again it gets scheduled in slot 13 and 14. With this T3 completes for one period 20.

T1	T2	T2	T3	T1	T2	T2	T3	T1	T3	T2	T2	T1	T3	T3	T2	T1	T2			T1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	29	20

Deadline Monotonic (DM) Algorithm

- Fixed priority algorithm
- **Shorter the relative deadline, higher the priority**
- When the relative deadlines of every task is proportional to their period, the schedule produced by RM and DM algorithms are identical.
- But when the relative deadlines are arbitrary, DM may produce a feasible schedule when RM fails.

Example:

T1 = (50, 50, 25, 100), T2 = (0, 60, 10, 20), T3 = (0, 125, 25, 50)

According to DM algorithm, T2 has highest priority because its relative deadline is 20.

Similarly T1 has lowest priority and T3 has priority in between.

EDF (Earliest Deadline First) Algorithm

- Dynamic priority algorithm
- Job with earliest (absolute) deadline has highest priority

Example: Consider two tasks T1 = (2, 0.9) and T2 = (5, 2.3)

LST (Least-Slack-Time-First) Algorithm

- Task level dynamic algorithm, and job level dynamic algorithm
- Job with smallest slack has highest priority
- At time *t*, the slack of a job whose remaining execution time is *x* and whose deadline is *d*

  $= d - t - x$

Example: Consider two tasks T1 = (2, 0.9) and T2 = (5, 2.3)

Schedulable Utilization

A scheduling algorithm can feasibly schedule any set of periodic task on a processor if the total utilization of the tasks is equal to or less than the schedulable utilization of the algorithm.

No algorithm can schedule a set of tasks with a total utilization greater than 1

Schedulable Utilization

Example for U > 1:

Consider EDF schedule for tasks: T1 = (2,1), T2 =(5,3)

Total utilization U = 1/2+ 3/5 = 0.5 + 0.6 = 1.1

Relative Merits

- Criterion to measure performance: Schedulable Utilization
- So higher the schedulable utilization, better is the algorithm.
- An algorithm whose schedulable utilization is 1, is an optimal algorithm.
- Optimal dynamic-priority algorithms outperforms fixed-priority algorithms in terms of schedulable utilization.
- But advantage of fixed-priority algorithms is predictability.

Schedulable Utilization of EDF Algorithm

Theorem:

A system T of independent, preemptable tasks with relative deadlines equal to their respective periods can be feasibly scheduled on one processor if and only if its total utilization is equal to or less than 1.

Schedulable utilization of the LST algorithm is also 1.

Schedulablity Test of EDF Algorithm

Schedulablity condition for EDF algorithm:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq 1$$

If above condition is satisfied, the system is schedulable according to EDF algorithm.

Optimality of RM & DM Algorithms

- Since these algorithms assign fixed priorities, they can't be optimal.
- While RM algorithm is not optimal for arbitrary periods, **it is optimal in the special case when the periodic tasks in the system are simply periodic.**
- A system of periodic tasks is **simply periodic** if for every pair of tasks T_i and T_k in the system and $p_i < p_k$, p_k is an integer multiple of p_i.
- In other words, for simply periodic tasks, the period of all tasks are integer multiple of the shortest period.

Optimality of RM Algorithm

Theorem:

A system of simply periodic, independent, preemptable tasks whose relative deadlines are equal to or larger than their periods is schedulable on one processor according to the RM algorithm if and only if it total utilization is equal to or less than 1.

RM Algorithm is optimal among all fixed-priority algorithms whenever the relative deadlines of the tasks are proportional to their periods.

Optimality of DM Algorithm

Theorem:

A system T of periodic, independent, preemptable tasks that are in phase and have relative deadlines are equal to or less than their periods can be feasibly scheduled on one processor according to the DM algorithm whenever it can be feasibly scheduled according to any fixed-priority algorithms.

Sufficient schedulability condition for RM algorithm

Theorem:

A system of 'm' independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to RM algorithm if its total utilization 'U' is less than or equal to

$$U_{RM} = m(2^{\frac{1}{m}} - 1)$$

As number of tasks approaches infinity, schedulable bound

$$U_{RM} = \lim_{m->\infty} m(2^{\frac{1}{m}} - 1) = \ln 2 = 0.69 = 69\%$$

$U(n) \leq U_{RM}(n)$ is not a necessary condition, a system of tasks may nevertheless be scheduled even when its total utilization exceeds the schedulable bound $U_{RM}(n)$.

Lehoczky's Schedulability Test for Fixed-Priority Algorithms

Used when Sufficient Schedulability Condition fails

Theorem

A set of periodic real-time tasks is RMA schedulable under any task phasing, if all the tasks meet their respective first deadlines under zero phasing (i.e. when all tasks have phases equal to 0).

Explanation

Let there are two tasks T1 = (30, 10) and T2 = (120, 60) scheduled as per RM Algorithm.

Then T1 has higher priority then T2.

.

(T1 and T2 have 0 phases – First job of T2 finishes at 90)

(T1 has phase 20 and T2 has phase 0 – First job of T2 finishes at 80)

From this example, it is obvious that worst case response time occurs for a lower priority task, the phase of this task and the phases of all other higher priority task are 0.

Lehoczky's Schedulability Test (Contd.)

As seen in the example, within deadline of the first job of T2 i.e. 120, higher priority task T1 can be scheduled 120 / 30 = 4 times.

So T2 has to wait for 4 X 10 (execution time of each job of T1) time slots during execution of its first job.

Hence in the worst case, the amount of time a low priority task T_i has to wait due to the higher priority tasks ($T_1, T_2, \ldots, T_{i-1}$) in the system is

$$\sum_{k=1}^{i-1} \left\lceil \frac{D_i}{p_k} \right\rceil e_k$$

So in worst case, T_i will be in the system for

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{D_i}{p_k} \right\rceil e_k$$

Then for all the tasks to be feasibly scheduled, this time should be less than or equal to the respective deadline i.e.

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{D_i}{p_k} \right\rceil e_k \leq D_i$$

This is Lehochky's Scheculability Test.

Floor and Ceiling Functions

<u>Floor</u>

floor(x) = $\lfloor x \rfloor$ is the largest integer not greater than x

<u>Ceiling</u>

ceiling(x) = $\lceil x \rceil$ is the smallest integer not less than x

Example:

x	$\lfloor x \rfloor$	$\lceil x \rceil$
2.4	2	3
5.5	5	6
-2.1	-3	-2
-2	-2	-2

Example

Question:

Please check if following sets of tasks can be scheduled by EDF and RM Algorithms.

T1 = (8,3), T2 = (9, 3), T3 = (15, 3)

Answer:

Utilization U = (3 / 8) + (3 / 9) + (3 / 15) = 0.375 + 0.333 + 0.2 = 0.9083

U < 1, so these tasks are schedulable by EDF algorithm.

Let us calculated the sufficient condition for RM schedulablity.

$$U_{RM} = m(2^{\frac{1}{m}} - 1) = 3(2^{\frac{1}{3}} - 1) = 3X(1.26 - 1) = 0.78$$

So U > U_{RM}, hence fails this test. But this doesn't mean that these tasks can't be schedulable by RM algorithm.

Let us perform Lehoczky's test.

Example (contd.)

As per RMA, the priorities of these tasks are T1 > T2 > T3.

For T1, the execution time 3 is less than its deadline 8, so it is schedulable.

For T2, the time it will be in the system in worst case scenario =

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{D_i}{p_k} \right\rceil e_k = 3 + \left\lceil \frac{9}{8} \right\rceil X3 = 3 + 2X3 = 9$$

T2 is schedulable since this time is equal to its deadline.

For T3, the time it will be in the system in worst case scenario =

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{D_i}{p_k} \right\rceil e_k = 3 + \left\lceil \frac{15}{8} \right\rceil X3 + \left\lceil \frac{15}{9} \right\rceil X3 = 3 + 2X3 + 2X3 = 15$$

T3 is schedulable since this time is equal to its deadline.

Hence all the 3 tasks pass Lehoczky's schedulability test.

Therefore all the 3 tasks are schedulable by RM Algorithm.

Practical Factors

Nonpreemptability

- So jobs are by nature nonpreemptable e.g. disk scheduling.
- When a low priority job is scheduled and it happens to be nonpreemptable, if a high priority job arrives later (either from blocked state or it gets released), then it has to wait.
- The high priority job has to wait until the nonpreemtable low priority job completes.
- This will increase the response time of the high priority job.
- Hence while considering whether the high priority jobs can meet their deadline or not, we also need to consider the effect of low priority nonpreemptable jobs on them.

Practical Factors

Self-suspension

- A job may suspend itself during execution due to various reasons like waiting for an I/O or remote procedure call etc.
- As a result O/S removes it from the ready queue and puts it in the suspended queue.
- The time spent during self suspension should also be considered during timing analysis of the jobs.

Context Switches

- Context switch is a usual phenomenon in a priority driven system.
- Hence context switch time should also be taken into consideration during the timing analysis.

Practical Factors

Limited-Priority Levels

- In practical systems number of priority levels are limited (e.g. in a token ring network, there are 8 priority levels, in RTOSes, usually there are 256 levels)
- Hence tasks(jobs) have non-distinct priorities, which need to be considered during the analysis.

Tick Scheduling

- During our analysis till now, we have assumed that scheduler does the schedulability tests as and when the jobs arrives (i.e. scheduler is event-driven).
- But practically, there will be a timer running and the scheduler will be waking up at each timer tick.
- So even if a job is ready, the scheduler may not notice it until the next timer interrupt. This introduce a certain delay in completion of the job.
- Also the ready job which yet to be noticed and to be put in a ready queue, should be placed in some other queue.
- These factors should also be considered during the analysis.

Practical Factors

Varying Priority in Fixed-Priority Systems

- In order to tackle priority inversion problem, sometimes the priorities of the lower priority jobs are raised. Such an operation will have effect on the analysis.

Hierarchical Scheduling

- This scheduling is done, when there are multiple tasks/jobs of same priorities.
- The tasks/jobs having same priorities are put into a cluster/subsystem.
- Two common type of scheduling approaches are used.
- Priority-driven/round-robin system: Here The clusters are scheduled in priority driven manner and the tasks/jobs in a clusters are scheduled in a round-robin manner.
- Fixed-time partitioning scheme: The clusters/subsystems are scheduled according to a cyclic schedules and the tasks/jobs in the subsystem are scheduled as per the scheduling algorithm chosen for the sybsystem.

Real Time System – Lecture 7

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Schedulability Test for Fixed-Priority Tasks with Short Response Times

BITS Pilani
Pilani Campus

Assumptions

We will confine our attention to the case where

- Priorities are fixed (e.g. RM Algorithm)
- Response times of the jobs are smaller than or equal to their respective periods.

Critical Instants

Critical Instant of a task T_i is a time instant which is such that

- The job in T_i released at the instant has the maximum response time of all jobs in T_i, if the response time of every job in T_i is equal or less than the relative deadline D_i of T_i

And

- The response time of the job released at that instant is greater than D_i if the response time of some jobs in T_i exceeds D_i.

The response time of a job in Ti released at **the critical instance** has **maximum response time**.

Theorem

In a fixed-priority system, where every job completes before the next job in the same task is released, a critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job in every higher-priority task, that is $r_{i,c} = r_{k,l_k}$ for some l_k for every $k = 1,2,\ldots,i-1$.

Time Demand Function

Suppose the release time t_0 is the job is a critical instant of task Ti.

Then at time $t_0 + t$, $t > 0$, the total processor time demand $w_i(t)$ of this job and all the higher-priority jobs released in $[t_0, t]$ is given by

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, 0 < t \le p_i$$

$w_i(t)$ is called the **time demand function** of task Ti.

If $w_i(t) > t$ for all $0 < t \le D_i$, then the job can not complete by its deadline.

The **maximum possible response time** W_i of all jobs in T_i is equal to the **smallest value of *'t'*** (because the job will try to complete earliest) that satisfies the equation

$$t = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, 0 < t \le p_i$$

Time Demand Function - Example

4 Tasks: T1 = (ϕ_1, 3, 1), T2 = (ϕ_2, 5, 1.5), T3 = (ϕ_3, 7, 1.25), T4 = (ϕ_4, 9, 0.5) are scheduled based on RM algorithm.

So priorities of these tasks are as following.

T1 > T2 > T3 > T4.

$w_1(t) = e_1 = 1, 0 < t \leq 3$

$$w_2(t) = e_2 + \left\lceil \frac{t}{p_1} \right\rceil e_1, 0 < t \leq p_2$$

$$\Rightarrow w_2(t) = 1.5 + \left\lceil \frac{t}{3} \right\rceil 1, 0 < t \leq 5$$

$$\Rightarrow w_2(t) = \begin{cases} 2.5, 0 < t \leq 3 \\ 3.5, 3 < t \leq 5 \end{cases}$$

$$w_3(t) = e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2, 0 < t \leq p_3$$

$$\Rightarrow w_3(t) = 1.25 + \left\lceil \frac{t}{3} \right\rceil \times 1 + \left\lceil \frac{t}{5} \right\rceil \times 1.5, 0 < t \leq 7$$

$$\Rightarrow w_3(t) = \begin{cases} 3.75, 0 < t \leq 3 \\ 4.75, 3 < t \leq 5 \\ 6.25, 5 < t \leq 6 \\ 7.25, 6 < t \leq 7 \end{cases}$$

Time Demand Function – Example (contd.)

$$w_4(t) = e_4 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2 + \left\lceil \frac{t}{p_3} \right\rceil e_3, 0 < t \le p_4$$

$$\Rightarrow w_4(t) = 0.5 + \left\lceil \frac{t}{3} \right\rceil \times 1 + \left\lceil \frac{t}{5} \right\rceil \times 1.5 + \left\lceil \frac{t}{7} \right\rceil \times 1.25, 0 < t \le 9$$

$$\Rightarrow w_4(t) = \begin{cases} 4.25, 0 < t \le 3 \\ 5.25, 3 < t \le 5 \\ 6.75, 5 < t \le 6 \\ 7.75, 6 < t \le 7 \\ 9.0, 7 < t \le 9 \end{cases}$$

Time Demand Function

Please note that, for T2, the y=t line crosses 2.5 and 3.5. The max response time is 2.5 (the smallest value of t satisfying the equation. 3.5 can't be considered as the max response time, because in one period 5, a job of T2 can execute at earliest possible instance, which is 2.5.

Time Demand Function

4 Tasks: T1 = (3, 1), T2 = (5, 1.5), T3 = (7, 1.25), T4 = (9, 0.5) are scheduled based on RM algorithm.

So priorities of these tasks are: T1 > T2 > T3 > T4.

Response times are shown as blue arrows in the diagram.

Time Demand Analysis

- Time demand function is a staircase function, with steps at integer multiple of the periods of high priority tasks.
- $w_i(t)$ is the demand of time and t is the supply of time.
- The task is schedulable if at any point of time during the inter-release time of two adjacent jobs of the task, the demand is less than or equal to the supply.
- It happens if $w_i(t) \leq t$ at any point of time during the inter-release time of two adjacent jobs of the task
- It means that $w_i(t)$ must intersect the straight line $y(t) = t$.
- It can only happen if $w_i(t) \leq t$ for some t = the integer multiple the period of any of the high priority tasks or that of the current task
- Intersection of $w_i(t)$ and $y(t)=t$, indicates the maximum response time, since this is the time instance where demand = supply.

Time Demand Analysis

Time Demand Analysis method proposed by Lehoczky

For each task T_i,

1. Compute the time demand function $w_i(t)$.

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, 0 < t \le p_i$$

2. Check whether the inequality

$$w_i(t) \le t$$

is satisfied for the values of t that are equal to

$$t = jp_k,$$

$$k = 1, 2, 3, \ldots, i;$$

$$j = 1, 2, \ldots, \lfloor \min(p_i, D_i) / p_k \rfloor$$

If this inequality is satisfied at **any of these instants**, T_i is schedulable.

Time Demand Analysis - Example

Let us take the old example i.e. T1 = (φ_1, 3, 1), (φ_2, 5, 1.5), (φ_3, 7, 1.25), (φ_4, 9, 0.5).

The time demand functions for all 4 tasks have been calculated.

$$w_1(t) = 1.0, 0 < t \leq 3$$

$$w_2(t) = \begin{cases} 2.5, 0 < t \leq 3 \\ 3.5, 3 < t \leq 5 \end{cases}$$

$$w_3(t) = \begin{cases} 3.75, 0 < t \leq 3 \\ 4.75, 3 < t \leq 5 \\ 6.25, 5 < t \leq 6 \\ 7.25, 6 < t \leq 7 \end{cases}$$

$$w_4(t) = \begin{cases} 4.25, 0 < t \leq 3 \\ 5.25, 3 < t \leq 5 \\ 6.75, 5 < t \leq 6 \\ 7.75, 6 < t \leq 7 \\ 9.0, 7 < t \leq 9 \end{cases}$$

Time Demand Analysis – Example (contd.)

For T1:

$$w_1(t) = 1.0, 0 < t \le 3$$

i = 1, so k and j doesn't exist.

Hence T1 is schedulable.

For T2:

$$w_2(t) = \begin{cases} 2.5, 0 < t \le 3 \\ 3.5, 3 < t \le 5 \end{cases}$$

i = 2, so k = 1, 2

For k=1:

j = 1,2,..., floor(min(5,5)/3)) = 1

t = jp_k = 1 X 3 = 3, w_2 (3) = 2.5 < 3.

For k=2:

j = 1,2,..., floor(min(5,5)/5)) = 1

t = jp_k = 1 X 5 = 5, w_2 (5) = 3.5 < 5.

Hence T2 is schedulable, since the inequality is satisfied for all the cases.

For T3:

$$w_3(t) = \begin{cases} 3.75, 0 < t \le 3 \\ 4.75, 3 < t \le 5 \\ 6.25, 5 < t \le 6 \\ 7.25, 6 < t \le 7 \end{cases}$$

i = 3, so k = 1, 2, 3

For k = 1:

j = 1,2,..., floor(min(7,7)/3)) = 1, 2

For j=1, t = jp_k = 1 X 3 = 3, w_3 (3) = 3.75 > 3 .

For j=2, t = jp_k = 2 X 3 = 6, w_3 (6) = 6.25 > 6.

For k = 2:

j = 1,2,..., floor(min(7,7)/5)) = 1

For j=1, t = jp_k = 1 X 5 = 5, w_3 (5) = 4.75 < 5.

For k = 3.

j = 1,2,..., floor(min(9,9)/7)) = 1

For j=1, t = jp_k = 1 X 7 = 7, w_3 (7) = 7.25 > 7.

Hence T3 is schedulable, since the inequality is satisfied for t = 5.

Time Demand Analysis – Example (contd.)

For T4:

i = 4, so k = 1, 2, 3, 4

For k = 1:

j = 1,2,..., floor(min(9,9)/3)) = 1, 2, 3

For j=1, t = jp_k = 1 X 3 = 3, w_4 (3) = 4.25 > 3.

For j=2, t = jp_k = 2 X 3 = 6, w_4 (6) = 6.25 > 6.

For j=3, t = jp_k = 3 X 3 = 9, w_4 (9) = 9.

For k = 2:

j = 1,2,..., floor(min(9,9)/5)) = 1

For j=1, t = jp_k = 1 X 5 = 5, w_4 (5) = 5.25 > 5.

For k = 3:

j = 1,2,..., floor(min(9,9)/7)) = 1

For j=1, t = jp_k = 1 X 7 = 7, w_4 (7) = 7.75 > 7.

For k = 4:

j = 1,2,..., floor(min(9,9)/9)) = 1

For j=1, t = jp_k = 1 X 9 = 7, w_4 (9) = 9 .

$$w_4(t) = \begin{cases} 4.25, 0 < t \le 3 \\ 5.25, 3 < t \le 5 \\ 6.75, 5 < t \le 6 \\ 7.75, 6 < t \le 7 \\ 9.0, 7 < t \le 9 \end{cases}$$

T4 is schedulable, since the inequality is satisfied for t = 9.

Busy Intervals

The time interval $(t_0, t]$ is called level $\prod_i$ busy interval, if in this interval, the processor is busy all the time executing jobs with priorities $\prod_i$ or higher, all the jobs executed in the busy interval are released in the interval, and at the end of the interval there is no backlog of jobs to be executed afterwards.

BITS ZG553 - Real Time System – Lecture 7

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems

BITS Pilani
Pilani Campus

What is a sporadic task?

- Tasks containing jobs that are released at random time instants and have hard deadlines are *sporadic tasks*.
- We treat sporadic tasks as hard real-time tasks.
- Our primary concern is to ensure that their deadlines are met.
- Minimizing their response times is of secondary importance.

What is a aperiodic task?

- We say a task is *aperiodic* if the jobs in it have either soft deadlines or no deadlines.
- We therefore want to optimize the responsiveness of the system for the aperiodic jobs, but never at the expense of hard real-time tasks whose deadlines must be met at all times.

Assumptions

It is impossible for some sporadic jobs to meet their deadlines no matter what algorithm is used.

So there are two alternatives:

a) Reject the sporadic jobs that can't complete on time
b) Accept all sporadic jobs and allow some of them to complete late.

Option (a) will be the focus here.

Objective

Two problems for the Sporadic and Aperiodic job scheduling algorithms:

1. If the scheduler accepts the sporadic job, it should schedule the job so that the job completes in time without causing any periodic tasks and previously accepted sporadic jobs to miss their deadlines. The problems are:
   a) How to do acceptance test
   b) How to schedule the accepted sporadic job

2. How to complete each aperiodic job as soon as possible without causing periodic tasks and accepted sporadic jobs to miss their deadlines

System Model

Optimality

1. An aperiodic job scheduling algorithm is optimal if it minimizes either the response time of the aperiodic job at the head of the aperiodic job queue or the average response time of all the aperiodic jobs for the given queuing discipline.

2. A sporadic job scheduling algorithm is optimal if it accepts each sporadic job newly offered to the system and schedule the job to complete in time if and only if the new job can be correctly scheduled to complete in time by some means.

Scheduling of Aperiodic tasks - Background Approach

Aperiodic jobs are scheduled and executed only when there is no periodic or sporadic job ready for execution.

Advantages:

- This method always produces correct schedule
- It is simple to implement.

Disadvantage(s):

- Execution of aperiodic jobs are delayed and their response times prolongs unnecessarily.

Scheduling of Aperiodic tasks – Background Approach

Example: T1 = (3,1), T2 = (10,4) scheduled as per RM algorithm

Aperiodic job A with execution time 0.8 arrives at time 0.1.

After $J_{1,3}$ is completed, there is free time slot, so A can be scheduled. A completes at 7.8

Response time of A = 7.8 – 0.1 = 7.7

Scheduling of Aperiodic tasks – Interrupt Driven Approach

Whenever an aperiodic job arrives, the execution of periodic tasks are interrupted and the aperiodic job is executed.

Advantage:

Aperiodic job will always meet its deadline and will have shortest possible response time.

Disadvantage:

Periodic tasks may miss some deadline.

Scheduling of Aperiodic tasks – Interrupt Driven Approach

Example: T1 = (3,1), T2 = (10,4) scheduled as per RM algorithm

Aperiodic job A with execution time 0.8 arrives at time 0.1. So it gets scheduled at time 0.1. Response time of A = 0.8. But the periodic jobs get delayed by 0.8 times.

Scheduling of Aperiodic tasks – Interrupt Driven with Slack Stealing

Whenever an aperiodic job arrives, the scheduler checks if there is any slack available. If available, it postpones the periodic job as per the slack availability and executes the aperiodic job during this time. This way periodic job doesn't miss its deadline. At the same time aperidic job executes as soon as possible.

Scheduling of Aperiodic tasks – Interrupt Driven with Slack Stealing

Example: T1 = (3,1), T2 = (10,4) scheduled as per RM algorithm

Aperiodic job A with execution time 0.8 arrives at time 0.1.

$J_{2,1}$ and $J_{1,1}$ have slack times, so can be postponed. During which A can be scheduled.

Polled Execution

- A poller or polling server (p_s, e_s) is a periodic task: p_s is its period and e_s is its execution time.
- When executed, it executes an aperiodic job, if the aperiodic job queue is non-empty.
- Poller suspends execution or is suspended by the scheduler either
  - when it has executed for e_s, *or*
  - when the aperiodic job queue becomes empty

Polled Execution

- Periodic Server: A periodic task which is created to execute aperiodic jobs.
- A periodic server (p_s, e_s) is defined by its period p_s and execution time e_s
- The term is e_s called execution budget (or simply budget)
- The ratio $u_s = e_s / p_s$ is called the size of the server.
- At the beginning of the period, the budget of the poller is set to e_s . We say that the budget is replenished.
- The time instant when the budget is replenished is called replenishment time.
- We say that the periodic server is backlogged whenever the aperiodic job queue is nonempty.
- The server is eligible (i.e. ready) for execution only when it is backlogged and has budget.
- When the server executes, it consumes its budget at one per unit time.
- The server budget becomes exhausted when the budget becomes zero.
- The budget of a poller becomes exhausted instantaneously whenever the poller finds the aperiodic queue empty i.e. itself idle.

Polling Server Example

Take the old example : **T1 = (3,1), T2 = (10,4), Aperiodic job A of execution time 0.8 arrives at time 0.1**.

- The poller (2.5, 0.5) is considered as the highest priority periodic task.
- At time 0, it wakes up, finds no aperiodic job, so gets suspended.
- At time 0.1, the aperiodic job arrives, but doesn't get scheduled, because poller is not running.
- At time 2.5 the poller gets scheduled again. It finds aperiodic task A, so executes it for 0.5 time units. Then it gets suspended.
- At time 5, the poller gets scheduled again and executes remaining 0.3 time units of the aperiodic job A and then gets suspended.
- Therefore, the job A completes at time 5.3 and its response time is 5.3 – 0.1 = 5.2.

Bandwidth Preserving Servers

- A bandwidth-preserving server is a periodic server
- Compared to polling server bandwidth preserving servers try to preserve their budget when they are not executed.
- Hence there are additional rules for consumption and replenishment for these servers:
  - A backlogged bandwidth-preserving server is ready for execution when it has budget. Scheduler keeps track of the consumption of the server budget. If budget is exhausted server becomes idle. Scheduler moves server back to ready queue, when budget is replenished and server is backlogged
  - If a new aperiodic job arrives, an idle server becomes backlogged and is put into the ready queue if it has budget

Deferrable Servers

- A deferrable server is the simplest bandwidth-preserving server.
- Unlike poller, it preserves its budget when there is no aperiodic tasks to execute.
- Consumption rule: The execution budget of the server is consumed at the rate of one unit per time whenever the server executes
- Replenishment rule: The execution budget of the server is set to e_s *at* kp_k ,$k = 0, 1, 2, \ldots$
- Server is not allowed to cumulate budget from period to period

Deferrable Server Example - 1

Take the old example.

The deferrable server is defined by T_{DS} = (2.5, 0.5) is considered as the highest priority periodic task.

At time 0, it wakes up, finds no aperiodic job, so gets suspended.

At time 0.1, the aperiodic job arrives, it wakes up and scheduled the aperiodic job for 0.5 time units.

At time 2.5 the deferrable server gets scheduled again. It executes remaining 0.3 time units of the aperiodic job.

Therefore, the job A completes at time 2.8 and its response time is 2.8 – 0.1 = 2.7.

Deferrable Server Example - 2

The deferrable server is defined by T_{DS} = (3, 1) is considered as the highest priority periodic task.

Periodic Tasks T1 = (2, 3.5, 1.5, 3.5) and T2 = (6.5, 0.5) are scheduled rate monotonically.

An aperiodic job A with execution time 1.7 arrives at time 2.8.

At time 0 , the server budget = 1. No aperiodic task arrives, so server is not executed.

At time 1, the server budget = 1. No aperiodic task arrives, so server is not executed.

At time 2, the server budget = 1. No aperiodic task arrives, so server is not executed.

At time 2.8, the server budget = 1. The aperiodic task arrives, so server is executed.

At time 3, the server budget = 0.8. It gets replenished to 1. So it continues execution till time 4.

At time 4, the server budget = 0, so the server gets suspended.

At time 6, server budget is replenished to 1, so server gets scheduled. This executes the aperidoic task till completion i.e. time 6.5.

At time 6.5, server has budget 0.5 remaining. Since there is no aperiodic tasks in the queue, it suspends itself.

Schedulability of Sporadic Jobs

- Sporadic jobs are scheduled as done in case of clock-driven scheduling.
- Acceptance tests are performed on sporadic jobs in EDF order.
- Once accepted, Sporadic jobs are ordered among themselves in EDF order.
- In a fixed-priority system , they are executed in a bandwidth preserving server.
- In a deadline-driven system, they are scheduled with periodic jobs on EDF basis.

Acceptance Test for Sporadic Jobs in Deadline-Driven Systems

For a sporadic job J_i, that has release time r_i, maximum execution time e_i and absolute deadline d_i,

$$\textbf{Density} = e_i / (d_i - r_i)$$

A sporadic job is said to be active in its feasible interval $(r_i, d_i]$.

<u>Theorem</u>

A system of independent, preemptable sporadic jobs is schedulable according to EDF algorithm if the total density of all active jobs in the system is no greater than 1 at all times.

Acceptance Test for Sporadic Jobs in Deadline-Driven Systems

Let us say, Δ = Total density of all the periodic tasks.

The all accepted sporadic jobs can meet their deadlines as long as the total density of all the active sporadic jobs is no greater than 1 - Δ.

<u>Acceptance Test Procedure:</u>

1. Calculate the total density of the periodic jobs Δ .
2. When the a Sporadic job *S(t, d, e)* arrives, divide the time interval into two: time interval I_1 before *d* and time interval I_2 after *d*.
3. Let the total densities of active Sporadic jobs in these two intervals are $\Delta_{s,1}$ and $\Delta_{s,2}$ respectively.
4. Accept the sporadic job S, if

$$\frac{e}{d-t} + \Delta_{s,1} \leq 1 - \Delta$$

Acceptance Test for Sporadic Jobs in Deadline-Driven Systems

Generalising the steps,

Let there are n_s sporadic jobs in the system, then the deadlines of these sporadic jobs will partition the time into $n_s + 1$ disjoint intervals: I_1 , I_2 , ..., I_{n_s+1}

Let I_l be the time interval during which a new Sporadic job $S(t, d, e)$ arrives.

Accept the sporadic job S, if

$$\frac{e}{d-t} + \Delta_{s,k} \leq 1 - \Delta$$

$$k = 1,2,3,\ldots,l$$

Example – Scheduling Sporadic Jobs (contd.)

Two periodic tasks T1 = (4,1) and T2 = (6, 1.5) are in the system, scheduled based on EDF algorithm.

Four sporadic jobs S1 = (0, 8, 2), S2 = (2, 7, 0.5), S3 = (4, 14, 1) and S4 = (9, 13, 2) are arrived and are to be scheduled.

Solution:

$\Delta = 0.25 + 0.25 = 0.5$.

So $1 - \Delta = 1 - 0.5 = 0.5$.

Time 0: $J_{1,1}$ and $J_{2,1}$ are released. Also sporadic job S_1 is released.

Density of S1 = 2 / (8-0) = 0.25 < 0.5.

So S1 is accepted. All the 3 jobs are scheduled based on EDF.

S1 divides the time into two intervals: (0, 8], (8, ∞).

Scheduler updates $\Delta_{s,1} = 0.25$, $\Delta_{s,2} = 0$.

$J_{1,1}$ has earlier deadline, so scheduled. Then $J_{2,1}$ has earlier deadline, so scheduled at time 1.

Example – Scheduling Spradic Jobs (contd.)

Example – Scheduling Sporadic Jobs (contd.)

Time 2: Sporadic job S_2 is released.

Density of S2 = 0.5 / (7-2) = 0.1

Total density of sporadic jobs in this interval = 0.25 + 0.1 = 0.35 < 0.5

So S2 is accepted. All the 2 jobs are scheduled based on EDF.

S2 divides the first time into two intervals: (0, 7], (7, 8].

Scheduler updates $\Delta_{s,1} = 0.35$, $\Delta_{s,2} = 0.25$ and $\Delta_{s,3} = 0$.

S2 has earlier deadline, so scheduled first, and then S1 is scheduled.

Example – Scheduling Sporadic Jobs (contd.)

Time 4: $J_{1,2}$ are released, but it has the deadline same as that of S1, so S1 can continue to execute.

Only sporadic job in the system now is S1.

S3 is also released.

Density of S3 = 1 / (14-4) = 0.1

$\Delta_{s,1}$ = 0.25. So total density of sporadic job in this interval = 0.25 + 0.1 = 0.35 < 0.5

So S3 is accepted. All the 3 jobs are scheduled based on EDF. S1 having earliest deadline, it continued to execute.

S3 divides the time into three intervals: (0, 8], (8, 14], (14, ∞).

Scheduler updates $\Delta_{s,1}$ = 0.35, $\Delta_{s,2}$ = 0.1 and $\Delta_{s,3}$ = 0

Time 5: S1 is done and $J_{1,2}$ is scheduled.

Time 6: $J_{2,2}$ is released. It has earlier deadline than S3, so gets scheduled.

Time 7.5: S3 gets scheduled.

Time 8: $J_{1,3}$ gets released. It has earlier deadline than S3, so S3 gets pre-empted and $J_{1,3}$ gets scheduled.

Example – Scheduling Spradic Jobs (contd.)

Example – Scheduling Sporadic Jobs (contd.)

Time 9: $J_{1,3}$ is completed. S4 gets released.

S4 divides the time into three intervals: (9, 14], (14, ∞)

In the interval (9,14], there are two active sporadic jobs S3 and S4 now.

Density of existing sporadic jobs = 0.1

Density of S4 = 2 / (13-9) = 0.5.

Total Density of sporadic jobs = 0.1 + 0.5 = 0.6 > 0.5.

Hence S4 is rejected. So S3 gets scheduled at time 9.

BITS ZG553 - Real Time System – Lecture 8

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Resources and Resource Access Control

BITS Pilani
Pilani Campus

Assumptions

- System contains only one processor
- System contains *ρ types of serially reusable* resources $R_1, R_2, \ldots, R_\rho$
- There are υ_i *indistinguishable* units of a resource of type R_i
  - Plentiful resources are ignored
  - Binary semaphore has one unit
  - Counting semaphore has *n units*
  - Only one unit of *write-lock*
  - A system containing 5 printers have 5 units of printer resources
- Serially reusable resources are allocated to jobs on a *non-preemptive basis and used in a mutually exclusive manner.* It means when a unit of resource is granted to a job, this unit is no longer available to other jobs until the job frees the unit.
- If a resource can be used by more than one jobs at the same time, this is modelled as a resource with several units, each used in a mutual exclusive manner. Example: A file which can be read by v users at a time is modelled as v mutually exclusive units of file resources

Mutual Exclusive Resource Access

- A lock-based concurrency control mechanism assumed to be used to enforce mutual exclusive access to resources
- When a job wants to use η_i *units of a* resource R_i, *it executes a lock* $L(R_i, \eta_i)$ (η =“eta”) to request them
- When the job no longer needs the resources, it releases them be executing an *unlock* $U(R_i, \eta_i)$
- When a lock request fails, the requesting job is blocked and loses the processor
- It stays blocked until the scheduler grants the resources the job is waiting for
- If a resource has only 1 unit the simpler notations $L(R_i)$ and $U(R_i)$ are used for lock and unlock

Critical Section

- A segment of a job that begins with a lock and ends at a matching unlock is called a ***critical section***
- Resources are released in last-in-first-out order
- A critical section that is not included in other critical sections is called an *outermost critical section*
- Critical sections are denoted by [*R, η; e*], where
  - *R* gives the *name* and
  - *η* the *number of* units of a resource and
  - *e* the *(maximum) execution time* of the critical section
- If there is only one unit of a resource the simpler notation [*R; e*] *is used*

Nested Critical Sections

- Nested critical sections are denoted by nested square brackets
  - *[R1; 7 [R2; 3]]* indicates that the critical section beginning *L(R1)* from includes another critical section that begins with *L(R2)*
  - Locks are locked and unlocked in the following order*: L(R1), L(R2), U(R2), U(R1)*

Resource Conflicts

- Two jobs conflict with each other, if some of the resources they require are of the same type
- They contend for a resource when one job requests a resource that the other job already has
- These terms are used interchangeably

Resource Conflicts - Example

- Jobs *J1, J2*, and *J3* with feasible intervals *(6,12], (2,14], (0,15]*
- Critical sections: *[R; 2], [R; 2], [R; 4]* for jobs 1, 2, 3
- Execution times of these jobs are 5, 4 and 6 times slots respectively.
- Resource requests by the three jobs are after 2, 2 and 1 time units respectively.
- EDF Schedule:

Priority Inversion

- Priority inversion occurs, when a low-priority job executes while a ready higher-priority job waits

Timing Anomalies

- Timing Anomalies can occur due to priority inversion
- Assume that critical section of job 3 is reduced to *[R; 2.5]* instead of *[R; 4]*
  - Jobs *J1, J2,* and *J3* with feasible intervals *(6,12], (2,14], (0,15]*
  - Critical sections: *[R; 2], [R; 2], [R; 2.5]* for jobs 1, 2, 3
  - Execution times of these jobs are 5, 4 and 6 times slots respectively.
  - Resource requests by the three jobs are after 2, 2 and 1 time units respectively.

Priority Inversion - Uncontrolled

- In this example J1 is blocked by J3
- But J3 is preempted by J2
- Thus J1 has to wait for the lower-priority job J2 that does not even require the resource that J1 needs.
- In this case priority inversion is considered *uncontrolled.*
- There can be arbitrary number of jobs with priorities lower than J1 and higher than J3 released in the meantime. They can further lengthen the duration of [priority inversion.
- In fact when priority inversion is uncontrolled, then a job can be blocked for an infinitely long time.

Deadlock

Deadlock can occur, if jobs block each other from execution

Example:

Job 1: *L(R1), L(R2), U(R2), U(R1)*

Job 2: *L(R2), L(R1), U(R1), U(R2)*

Wait-for-graph

- Wait-for-Graphs are used to describe the dynamic-blocking relationship among jobs
- Vertexes represent jobs (J_i) and resources
- Resources are represented by R_i, k; where R_i is the resource id and k is the number of units of resources available.
- If a job has acquired x units of resources, then it is represented by an edge (line with arrowhead) from the resource to the job with x labelled on it.
- If a job requested y units of resources, but denied earlier, then it is represented by an edge (line with arrowhead) from the job to the resource with y labelled on it.
- **A circular wait-for-graph indicates deadlock.**

- *J_3 has one unit of resource R*
- *J_2 is waiting for one unit of resource R*
- *J_1 is not waiting for resource R*

Wait-for-graph : Deadlock condition

- **A circular wait-for-graph indicates deadlock.**

- J_2 has one unit of resource R_1
- J_1 is waiting for one unit of resource R_1
- J_1 has one unit of resource R_2
- J_2 is waiting for one unit of resource R_2

So it is a deadlock.

Additional notations

- A periodic task is represented by the tuple (Φ_i, p_i, e_i, D_i, $[R, x; y]$), where [R, x; y] represents the critical section requiring x units of resource R for y units of time

- When a job requires more then one resources and has more than one critical sections, all of the critical sections are listed in the tuple.

- When the resources are not important, it is represented by the tuple (Φ_i, p_i, e_i, D_i, c_i), where c_i is the maximum execution time of the longest critical section of the job. (Please note that c_i is included in e_i)

- In general
  - A critical section here means outermost critical section (when inner critical sections are not specified)
  - A critical section of a periodic task means critical section of each job of that periodic task

Resource requirement graph

- In a resource requirement graph, there is a vertex for each job and resource.
- The integer next to each resource vertex indicates number of units of the resource.
- If a job requires a resource, there will be a edge drawn from the job to the resource
- The edge will be labelled by the 2 tuple *(x, y)*, where the job requires *x* units of the resources for *y* units of time in the critical section.
- If there is one unit of resource, then first parameter of the tuple is omitted i.e. Only the time units required for the critical section is mentioned

- Each job of T_1 requires 2 units of R_1 for 3 units of time and 1 unit of $R_2$for 1 unit of time
- R_1 has 5 units of resources and R_2 has 1 unit of resource.
- T_2 doesn't require any resources
- Each job of T_3 requires R1 and R2 as specified by the critical section (rather than indicating over the edges)

Blocking time

- In a fixed priority system, let tasks are indexed in the decreasing order of the priority i.e. In a system of tasks $T_1, T_2, T_3, \ldots, T_i, T_{i+1}, \ldots T_n$; T_i has higher priority than T_{i+1} .
- A job in task T_i is blocked if a the resource required by T_i is taken by any task of priority lower than T_i i.e any task between T_{i+1} to T_n.
- If c_k is the maximum critical section time of task T_k, then blocking time of Task T_i is

$$b_i(rc) = \max_{i+1 \le k \le n} (c_k)$$

$$where$$

$$i = 0,1,\ldots,n$$

$$k = 1,2,\ldots,n$$

$$n \ge 1$$

Nonpreemptive Critical Section (NPCS) Protocol

- **When a job holds a resource, it executes at a priority higher than the priorities of all jobs.**
- Because no job is ever preempted when it holds a resource, Deadlock can never occur.
- Uncontrolled priority inversion can't occur

Previous Example

- Priorities of J1 and J3 are highest and lowest, J2 has priority lower than J1, but higher than J3
- When J3 is released, it acquires the resource and continues till the end of the critical section
- When J1 is released at time 2, it waits till time 5, when J3 releases the resource.
- At time 5, J1 Gets scheduled. At the same time J2 is released, but it doesn't get scheduled, since J1 has higher priority.
- At time 6, J1 acquires the resource and holds it till time 8.
- J1 completes at time 9. At this time J2 is scheduled.
- J2 continues till time time 14. Then J3 is scheduled.
- J3 completes at time 15.

Nonpreemptive Critical Section (NPCS) Protocol

- Advantages
  - Simple to implement
  - Uncontrolled priority inversion cannot occur
  - Good protocol when critical sections are short

- Disadvantages
  - Every job can be blocked by every lower-priority job even if there is no resource conflict between them

Priority Inheritance Protocol

- Like NPCS, it doesn't require any prior knowledge on the resource requirements of the jobs.
- Uncontrolled priority inversion cannot occur
- It does not avoid deadlock. External mechanisms needed to avoid deadlock

Priority Inheritance Protocol - Rule

Scheduling Rule

- Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities.
- At its release time t, the current priority $\pi(t)$ of every job J is equal to its assigned priority
- The job remains at this priority except under the condition stated in the priority-inheritance rule

Allocation Rule

- When a job J requests a resource R at time t,
  - if R is free, R is allocated to J until J releases the resource, and
  - if R is not free, the request is denied and J is blocked

Priority-Inheritance Rule

- When the requesting job J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J.
- The job J_l executes at its inherited priority $\pi(t)$ until it releases R
- At that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquired the resource R

Priority Inheritance Protocol - Example

Job	r_i	e_i	Π_i	Critical sections
J1	7	3	1	[Light blue; 1]
J2	5	3	2	[Orange; 1]
J3	4	2	3	
J4	2	6	4	[Light blue; 2.5[Orange; 1.5] (Light blue: 2 slots, Orange: 1.5 slots, Light blue: 0.5 slots)
J5	0	6	5	[Orange; 4]

The jobs require resources as mentioned 1 time slot after start of their execution

Priority Inheritance Protocol - Example

At time 0, J5 is released. Since there were no other jobs, it was scheduled.

At time 1, it was granted resource ‘orange’

At time 2, CJ4 was released, so J5 was preempted and J4 was executed

At time 3, J4 required light blue resource and was granted

At time 4, J3 was released, so J4 was preempted. J3 didn’t require any resource.

At time 5, J2 was released, which preempted J3.

At time 6, J2 required orange resource. But it was hold by J5. As per priority inheritance rule, the priority of J5 was made that of J2, hence it was scheduled and J2 was preempted.

At time 7, J1 is released, which is of highest priority. So J5 gets preempted and J1 is scheduled

At time 8, J1 requires light blue resource, which is already acquired by J4. Hence as per priority inheritance rule, priority of J4 is made equal to that of J1. So J1 gets preempted and J4 gets scheduled.

At time 9, J4 requests for resource Orange, which is acquired by J5. Hence J5 inherits the priority of J4 (in turn of J1) and executes.

At time 11, J5 releases the Orange resource, its priority is back to original, so gets preempted. J4 gets scheduled and acquires the orange resource

At time 12.5, J4 completes using orange resource. Then it uses the blue resource and releases it at time 13.

At time 13, J4 releases light blue resource, so its priority become the original one. So it gets preempted and J1 executes acquiring the light blue resource

At time 14, J1 releases the light blue resource. But it is the highest priority task hence continues till its completion at time 15.

At time 15, next high priority task J2 executes , acquires the orange resource.

At time 16, J2 releases the orange resource, but being the highest priority task in the system continues till its completion at time 17.

At time 17 J3 gets scheduled and continues till its completion at time 18.

At time 18, J4 gets scheduled and it completes at time 19.

At time 19, J5 gets scheduled and completes at time 20.

Priority Ceiling Protocol

- Priority Ceiling Protocol extends Priority Inheritance Protocol to prevent deadlock.
- Two key assumptions:
  - Assigned priorities of the jobs are fixed.
  - Resource required by all jobs are known a priori before the execution of any job begins.
- Priority Ceiling of any resource Ri is the highest priority of all the jobs that require the resource Ri, denoted by Π(Ri).
- At any time t, the current priority ceiling (or simply ceiling) Π(t) of the system is equal to the highest priority ceiling of the resources that are in use at the time
- If all resources are free, Π(t) is equal to Ω, a nonexistent priority level that is lower than the lowest priority level of all jobs

Priority Ceiling Protocol - Rule

Scheduling Rule

- Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities.
- At its release time t, the current priority π(t) of every job J is equal to its assigned priority
- The job remains at this priority except under the condition stated in the priority-inheritance rule

Priority Ceiling Protocol - Rule

Allocation Rule

When a job J requests a resource R at time t,

- if R is not free, the request is denied and J is blocked
- If R is free,
  - If J's priority $\pi(t)$ is higher than the current priority ceiling **$\Pi(t)$**, R is allocated to J.
  - If J's priority $\pi(t)$ is not higher than the current priority ceiling $\Pi(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling is equal to $\Pi(t)$, otherwise J's request is denied and J becomes blocked.

Priority-Inheritance Rule

- When the requesting job J becomes blocked, the job J_l which blocks J inherits the current priority $\pi(t)$ of J.
- The job J_l executes at its inherited priority $\pi(t)$ until it releases every resource whose priority ceiling is equal to or higher than $\pi(t)$ of J.
- At that time, the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquired the resource R

Priority Ceiling Protocol - Rule

Please note:

- The Allocation rule assumes that only one job can holds all the resources with priority ceiling equal to $\Pi(t)$.
- The Priority inheritance rule assumes that only one job is responsible for other jobs' request being denied, because it holds either the requested resource or a resource with priority ceiling $\Pi(t)$.

Priority Ceiling Protocol - Example

Job	r_i	e_i	Π_i	Critical sections
J1	7	3	1	[Light blue; 1]
J2	5	3	2	[Orange; 1]
J3	4	2	3	
J4	2	6	4	[Light blue; 2.5[Orange; 1.5]

Priority Ceiling Protocol - Example

At time 0, J5 is released. Since there were no other jobs, it was scheduled.

At time 1, it was granted resource ‘orange’. At this time the ceiling $\Pi(t) = 2$, the priority ceiling of resource ‘orange’.

At time 2, J4 was released, so J5 was preempted and J4 was executed

At time 3, J4 required light blue resource. But J4’s priority is 4, which is less than the ceiling 2. So J4 is blocked. J5 is the job which blocked J4, since it holds the ‘orange’ resource which determines the ceiling. So J5 inherits J4’s priority i.e. 4 and executes.

At time 4, J3 was released, so J5 was preempted, since j3 has higher priority than the J5’s inherited priority. J3 didn’t require any resource.

At time 5, J2 was released, which preempted J3.

At time 6, J2 required orange resource. But it was hold by J5. As per priority inheritance rule, the priority of J5 was made that of J2, hence it was scheduled and J2 was preempted.

At time 7, J1 is released, which is of highest priority. So J5 gets preempted and J1 is scheduled

At time 8, J1 requires light blue resource. J1 has higher priority than the ceiling. So it is granted the resource. The ceiling $\Pi(t) = 1$. It releases the resource at time 9. Then it continues execution (being the highest priority job) till its completion i.e. Time 10.

At time 10: J5 is scheduled, since it has the next highest priority because of priority inheritance (i.e. 2).

At time 11, J5 releases the Orange resource, its priority is back to original i.e. 5, so gets preempted. The ceiling of the system becomes Ω, since no jobs hold any resource. So highest priority job J2 get scheduled, which acquires the ‘orange’ resource’. So the ceiling of the system again becomes 2.

At time 12: J2 releases the ‘orange’ resource. Being the highest priority job in the system, it continues till completion i.e. Time 13.

At time 13: J3 gets scheduled being the highest priority job in the system. It continues till completion i.e. Time 14.

At time 14: J4 gets scheduled. It uses the ‘light blue’ resource for 2 time slots, then also uses ‘orange’ resource for 2 time slots and then uses the ‘blue’ resource again for 0.5 time slots till time 18. After then it executes for one more slot and gets completed.

At time 19, J5 gets scheduled and completes at time 20.

BITS ZG553 - Real Time System – Lectures 9

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Hardware for Real-Time Systems

BITS Pilani
Pilani Campus

Motivation

- Many real-time systems are embedded systems
- Embedded systems interact closely with specialize input and output devices including sensors and actuators
- Hardware structure can affect timing and require special software design techniques
- Real-time systems engineers must have some familiarity with hardware and do some “low level” programming

Processor Architecture

BITS Pilani
Pilani Campus

Processor Architectures

- Von Neumann Architecture
- Harvard Architecture

Von Neumann Architecture

- Described in 1945 by the mathematician and physicist John von Neumann and others in the First Draft of a Report on the EDVAC.
- Also known as Princeton Architecture.
- CPU is connected to Memory and I/O through the System Bus.
- In a Von Neumann Architecture, I/O is Memory Mapped.

System Bus constitutes:

- Unidirectional Address Bus
- Bidirectional Data Bus
- Control Bus, which is a heterogeneous collection of independent lines such as control, status and power lines.

Harvard Architecture

- The name Harvard Architecture comes from the Harvard Mark I relay-based computer.
- Separate data and address bus
- It is possible to access program memory and data memory simultaneously.
- Typically, code (or program) memory is read-only and data memory is read-write. Therefore, it is impossible for program contents to be modified by the program itself.
- But the price paid is increase in chip area (therefore power consumption)
- Many modern CPUs comprises both Harvard and von Neumann characteristics: the separate on-chip instruction and data caches have a Harvard type interface, while the common off-chip cache memory is interfaced through a single system bus.

CPU Internal Architecture

- Instruction processing takes place inside the CPU
- It has 3 parts: Control Unit, Internal Bus and Data Path

- Control Unit
  - Interfaces to the system bus through Program Counter Register (PCR)
  - PC points to the address from which the next instruction to be fetched.
  - Instruction fetched is stored in the Instruction Register (IR)
  - Instruction is then decoded, desired operands are fetched from memory and appropriate command is given to the ALU of the Data Path
- Data Path
  - ALU (Arithmetic and Logic Unit), performs the desired operation.
  - The output of ALU is either stored in Registers of Memory depending upon the Instructions
  - Also various status flags such as Carry/Borrorw, Zero etc are updated in the status registers depending upon the state of the CPU. These flags are used for implementing conditional instructions.

Instruction Processing

- Instruction processing requires multiple consecutive stages.
- Typical instruction processing stages constitute: Fetch, Decode and Execute.
- But these stages can be further divided into multiple sub-stages. For example, we can have 5 stages for instruction processing: Fetch, Decode, Load, Execute and Store.
- The duration of each instruction may vary, because each of these stages may take different amount of clock cycles.

Instruction Processing

- Each Instruction consists of an Opcode and zero, one or more Operands.

  Example: `MOV AX, BX`

  Here, `MOV` is the Opcode and `AX` and `BX` are the Operands
- Addressing modes are the way CPU accesses the Operands

  Immediate: `ADD AX, 5 ; AX = AX + 5`

  Register Direct: `SUB AX, BX ; AX = AX - BX`

  Register Indirect: `ADD AX, [BX]; AX = AX + *BX`

  … … … … …
- Two principal techniques for implementing the control unit
  - Microprogramming
    - Every instruction consists of a sequence of primitive hardware commands, microinstructions for activating he data path.
    - Suitable when the number of instructions is large
    - Instruction processing is slower
  - Hardwired Logic
    - Each instruction is mapped to some combinational and sequential logic hardware.
    - Suitable when the number of instructions is small
    - Results in fast instruction processing
- In order to conserve power, modern processors implements 'slowdown modes'.
  - Specific instructions enable such modes, where the voltage and/or frequency are lowered, reducing the power consumption.

Pipeline

Example of a 3-stage pipeline:

- Fetch, Decode, Execute
- 3 cycle latency

Pipeline Hazards

- **Structural Hazards**
  - These occur when a single piece of hardware is used in more than one stage of the pipeline, so it's possible for two instructions to need it at the same time.
  - Example: One memory unit being shared by subsequent instructions)

- **Data Hazards**
  - This is when reads and writes of data occur in a different order in the pipeline than in the program code.
  - RAW (Read After Write): Occurs when, one instruction reads a location after an earlier instruction writes new data to it, but in the pipeline the write occurs after the read.
  - WAR (Write After Read): A write occurs after a read, but the pipeline causes write to happen first.
  - WAW (Write After Write): Two writes occur out of order.

  In order to resolve these hazards, pipeline stall happens.

- **Control Hazards**
  - This is when a decision needs to be made, but the information needed to make the decision is not available yet.
  - Example: Branch instruction in a pipeline.

  In some RISC architectures, the instruction following the branch is executed whether or not the branch is taken. It is called delayed branch.

Where do Processors Spend Time ?

Instruction Type	Dynamic Usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Others	1%

Source: Steve Furber, *ARM System-on-chip Architecture*, Second Edition, Pearson, 2007

CISC Processor Architecture

- CISC – **Complex Instruction Set Computer**
- Instructions are of variable format and length
- Large number of instructions
- Practically any instruction can reference memory
- Single set of work registers
- Micro-programmed instruction decode logic. Complexity of instruction is handled at this level
- Great variety of addressing modes
- No instruction pipelines
- Programmer's coding effort is simplified
- Example: x86 Architecture

RISC Processor Architecture

- RISC – **Reduced Instruction Set Computer**
- Fixed 32-bit instruction size
- Load-store architecture – Instructions operating on data operates only on registers.
- Large register bank of thirty-two 32-bit registers
- Hard-wired instruction decode logic (CISC processors use large microcode ROMs)
- Pipelined execution
- Single-cycle execution (most of the instructions)
- Performance is enhanced
- Example: ARM Architecture

Multi-core Processor

Motivation

- CPU architecture is evolving, with higher frequencies, longer pipelines.
- As clock frequencies increases, power consumption increases.
- Cooling such high speed processors is an costly affair.
- Then why not have processors with multiple cores, operating in parallel ?

Multi-core processors

- Have individual cores with private cache memory, separate instruction and data caches.
- These small on-chip caches are interfaced to larger on-chip common cache memories

Multi-core Processor System

- From HW Perspective:
  - Homogenous: If all CPU cores are of same architecture
  - Heterogeneous: If all CPU cores are of different architecture
- From SW Perspective:
  - AMP: Asymmetric Multi-processing
  - SMP: Symmetric Multi-processing

Multi-core Processor System - AMP

- CPU cores can be of homogenous or heterogeneous architectures
- Each core has own address space
- Used when different architectures are optimal for specific activities, such as: one is for DSP and other one is for MCU
- There is an opportunity to deploy different O/Ses on different cores

Multi-core Processor System - SMP

- CPU cores are of homogenous architecture
- CPU cores share the memory space
- A single O/S runs on all the cores
- Some kind of communication facility between the CPUs is provided this is normally through shared memory, but accessed through the API of the OS
- Typically, SMP is used when an embedded application simply needs more CPU power to manage its workload

ARM11MPCore Architecture

- **SMP** Architecture
- Shared L1 cache is managed by Snoop Control Unit:
  - Each CPU's snooping unit looks at writes from other processors
  - If a write modifies a location in this CPU's level 1 cache, the snoop unit modifies the locally cached value
- Interrupt Distributer
  - Masks and prioritizes the interrupts as in standard interrupt system
  - Identifies the set of CPUs that can handle the interrupts
  - Sends each CPU its highest priority pending interrupt

Source: http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php

Common DSP Processor features

- Harvard architecture
  - Needed for accessing multiple memory locations simultaneously
- Dedicated single-cycle Multiply-Accumulate (MAC) instruction (hardware MAC units)
  - Required for processing arithmetic operations like

$$Y = \sum_{n=1}^{N} a_n * x_n$$

- Single-Instruction Multiple Data (SIMD)
- Very Large Instruction Word (VLIW) architecture
- Pipelining
- Saturation arithmetic
- Special Execution Units with
  - Minimum overhead looping
  - Low interrupt and branch latency
  - Fast context switching
  - Register banking / stacking
- Cache
- DMA

SIMD – Single Instruction Multiple Data

One instruction operates on multiple data simultaneously.

Example: Addition of two row vectors:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Same ADD operation on 4 separate data in parallel

VLIW Architecture

- **VLIW - Very Long Instruction Word**
- Exploits instruction level parallelism
- Has **multiple execution units** executing in parallel
- Hence has **multiple register files** to help parallel execution of instructions
- Instruction word is a combination of multiple instructions, hence it is very long
- **Compiler reorders the instructions** in order to achieve maximum parallel execution possible. Hence VLIW compiler is very complex.

VLIW Architecture

VLIW Architecture

Superscalar Architecture

Difference between VLIW and Superscalar Architectures

VLIW	Superscalar
• Single and very long instruction word containing multiple instructions	• Multiple instructions are issued separately
• Compiler does instruction reordering (Static instruction scheduling)	• Hardware does instruction reordering (Dynamic instruction scheduling)

Memory Technology

BITS Pilani
Pilani Campus

Motivation

- Memory access has major contribution in latency.
- It becomes important when the utilization factor is in dangerous zone (83% - 99%) (Refer to lecture 2)
- In such systems hierarchical memory architecture may cause aperidic jobs to miss deadlines with considerable delays.
- So it is necessary to understand memory technologies, so that appropriate design decision can be taken

Different Classes of Memory

- Traditional Distinction between Memories:
  - ROM (Read Only Memory) – Non-volatile
  - RAM (Random Access Memory) – Volatile
- ROMS used to get programmed once at the beginning using a special programming units.
- But at present, EEPROM and Flash memories can be rewritten without any special programming units.
- At present following types of Memories are widely used
  - Non Volatile
    - EEPROM
    - Flash
  - Volatile
    - SRAM
    - DRAM

Non-Volatile Memory

- Erasing is slower than RAMs
- Each device can be rewritten 1000,000 – 1,000,000 times only.
- EEPROMs are usually used for storing nonvolatile programs and parameters.
- Flash memory is usually used for storing both application programs (including operating system) and large data records.
- These are slower to read compared to RAM
- So it is a common practice in many real-time systems to load the application-program (including the operating system) from a flash memory to the RAM and then run it from the RAM.
- In many cases the flash memory used is a movable one (e.g. a SD Card or a USB Memory stick)

SRAM and DRAM

- SRAM-type memory cell requires typically 6 transistors, while DRAM cell requires a single transistor and a capacitor only.
- So SRAM takes mores chip area compared to DRAM.
- Hence SRAMs are costlier than DRAMs.
- But SRAMs are faster than DRAMs.
- DRAM has the issue of leakage in their storage capacitors. So DRAMs need to be refreshed periodically (typical refresh cycle is 3-4 ms).

While designing a RAM subsystem of a real-time system, the basic rule of thumb is that if you need a large amount of memory, then use DRAM, otherwise go for SRAM.

Evolution of DRAM

- First page mode (FPM) DRAM
- Extended data output (EDO) DRAM
- Synchronous DRAM (SDRAM)
- Direct Rambus DRAM (DRDRAM)
- Double data rate 3 synchronous DRAM (DDR3 DRAM)

Memory Hierarchy

- CPU Clock Rates are increasing at a faster rate compared to DRAM access times.
- Programmers always desire faster and infinite memory.
- Solution is Memory Hierarchy.
- In order of fastest to slowest, memory should be assigned, considering cost as follows:
  - internal CPU memory
  - registers
  - cache
  - main memory
  - memory on board external devices

Cache

- Cache memories rely on the 'locality of reference' principle.
- Locality of reference
  - Refers to the address distance between memory between consecutive code and data access.
  - If the code or data fetched tends to reside close in memory, then the locality of reference is low.

- A cache is a relatively small storage of fast memory where frequently used instructions and data are kept.
- Basic operations of Cache
  - Suppose CPU requests the content of a DRAM.
  - First the cache controller checks the address to see if the particular location is in the cache.
  - If present, the data is immediately retrieved from the cache, which is significantly faster.
  - If the address is not in the cache, then a cache miss occurs. Subsequently the cache content gets written back to the main memory and the new block is loaded into the cache.
- Cache Hierarchy
  - L1 Cache: Nearest cache to CPU
  - L2 Cache: Next level cache between L1 cache and Main Memory

Cache Access Time

t_{main} – access time for main memory

t_{L1} – access time for primary cache

h_1 – hit ratio of primary cache

t_{L2} – access time for secondary cache

h_2 – hit ratio of secondary cache but not primary

If system has only L1 cache

Average access time, $t_{av} = h_1 t_{L1} + (1-h_1) t_{main}$

If system has L1 and L2 cache

Average access time, $t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1-h_1 -h_2) t_{main}$

Cache Access Time - Example

Problem:

Assume that the system has a two-level cache. The level-1 cache has a hit rate of 90% and the level-2 cache has a hit rate of 97%. The level 1 cache access time is 4 ns, the level 2 cache access time is 15 ns and access time of main memory is 80 ns. What is the average memory access time?

Solution:

$h_1 = 0.9$,

$h_2 = (1-0.9) \times 0.97 = 0.1 \times 0.97 = 0.097$,

$t_{L1} = 4$ ns, $t_{L2} = 15$ ns, $t_{main} = 80$ ns.

$t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1-h_1-h_2) t_{main}$

$= 0.9 \times 4 \text{ ns} + 0.097 \times 15 \text{ ns} + (1 - 0.9 - 0.097) \times 80 \text{ ns}$

$= 0.9 \times 4 \text{ ns} + 0.097 \times 15 \text{ ns} + 0.003 \times 80 \text{ ns}$

$= 3.6 \text{ ns} + 1.455 \text{ ns} + 0.24 \text{ ns}$

$= 5.295 \text{ ns}$

Peripheral Interfacing

BITS Pilani
Pilani Campus

Interrupts

- An hardware interrupt is an external hardware signal that initiates an event.
- Each interrupt is associated with an interrupt vector address, which contains the interrupt handler routine.
- Interrupt handler routine is a program for the desired action to be taken when the interrupt occurs.
- There are software interrupts as well, which are generated internally.
- A typical interrupt service process is as follows
  - The interrupt request line is activated
  - The interrupt request is latched by the CPU hardware
  - The processing of ongoing instruction is completed
  - The content of program counter register (PCR) is pushed to the stack
  - The content of status register (SR) is pushed to the stack
  - The PCR is loaded with the interrupt handler's address.
  - The interrupt handler is executed.
  - The original content of the SR is popped back from the stack.
  - The original content of the PCR is popped back from the stack.

Programmable Interrupt Controller (PIC)

- When number of interrupt sources are more than the available external interrupt inputs on a microprocessor/microcontroller, then PIC chips are used.
- Priorities can be assigned among the interrupts.
- Example: 8059 for x86

Direct Memory Access (DMA)

- Access to the computer's memory is given to other devices in the system without CPU intervention.
- Information is transferred directly into main memory by the external device.
- DMA controller is required unless the DMA circuitry is integrated into the CPU.
- Because CPU participation is not required, data transfer is fast.
- DMA is the most preferred method for data transfer in real-time systems.

DMA Transfer Process

- I/O Device requests DMA transfer by activating DMA-request signal (D_REQ)
- This cause the DMA Controller issue a bus-request signal (B_REQ) to the CPU.
- Then the CPU finishes its present bus cycle and activates a bus-acknowledgement signal (B_ACK).
- After recognizing B_ACK signal, DMA Controller activates a DMA-acknowledgement signal (D_ACK) to the I/O device.
- Then I/O device performs the data transfer to the memory.
- One the transfer is done, the DMA controller deactivates the B_REQ signal, giving the bus back to the CPU.

BITS ZG553 - Real Time System – Lecture 11

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Real Time Operating Systems - Memory Management

BITS Pilani
Pilani Campus

Memory Management in Real-Time Systems

- Memory is limited, hence must be used carefully
- System must meet real-time constraints, memory management overhead should be minimal

In general frequent memory allocation and deallocation dynamically is discouraged.

Memory Fragmentation

- Gaps between allocated contiguous memory - total memory space exists to satisfy a request, but it is not contiguous
- Allocating and deallocating memory of different sizes will cause Fragmentation of memory.

Fragmentation hamper efficient memory usage and degrade real-time performance due to associated overhead.

Heap state after series of malloc and free requests

Memory Fragmentation

- External Fragmentation
  - Gaps between allocated contiguous memory - total memory space exists to satisfy a request, but it is not contiguous
  - Allocating and deallocating memory of different sizes will cause External Fragmentation of memory.

- Internal Fragmentation
  - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Both internal and external fragmentation hamper efficient memory usage and degrade real-time performance due to associated overhead.

Memory Management - Swapping

- A simple scheme that allows the operating system to allocate memory to two processes “simultaneously” is swapping.
- In this case, the operating system is always memory resident, and one process can co-reside in the memory space not required by the operating system.
- When a second process needs to run, the first process is suspended and then swapped, along with its context, to a secondary storage device, usually a disk.
- The second process, along with its context, is then loaded into the user space and initiated by the dispatcher.

The access time to the secondary store is the principal contributor to the context switch overhead and real-time response delays.

Memory Management - Overlays

- A simple technique that allows a single program to be larger than the allowable memory.
- Here the program is broken up into dependent code and data sections called overlays, which can fit into available memory.
- Special program code must be included that permits new overlays to be swapped into memory as needed (over the existing overlays), and care must be exercised in the design of such systems.

It has negative real-time implications because the moving the memory between primary and secondary storages consumes time.

But overlaying can be used to extend the available address space.

Memory Management – Fixed Partition

- A more elegant scheme allows process to be memory-resident simultaneously by dividing the user memory space into a number of fixed-size partitions.
- These partitions are allocated to the processes when processes request for memory.
- This scheme is useful in systems where the number of tasks to be executed is fixed, as in many embedded applications.
- Partition swapping to disk can occur when a task is preempted.
- Tasks must reside in continuous partitions, and the dynamic allocation and deallocation of memory can cause problems.
- When a process requires less memory than the partition size, Internal fragmentation occurs.

Memory Management – Variable Partition

- In some commercial real-time executives, memory is divided into regions in which each region contains a collection of different-sized, fixed-sized partitions.
- For example, one region of memory might consist of 10 blocks of size 16Mb, while another regions might contain 5 blocks of 32M, etc.
- The operating system tries to satisfy a memory request so that the smallest available partitions are used. This tends to reduce internal fragmentation.
- Memory is allocated in amounts that are determined by the requirements of the process to be loaded into memory.
- This technique is more appropriate when the number of real-time tasks is unknown or varies.
- In addition, memory utilization is better for this technique than for fixed block schemes because little or no internal fragmentation can occur, as the memory is allocated in the amount needed for each process.
- External fragmentation can still occur because of the dynamic nature of memory allocation and deallocation, and because memory must still be allocated to a process contiguously.

Memory Allocation

- C library function malloc() allocates memory from heap
- Some embedded systems may have individual heaps per process
- Some may have a single heap for all processes
- Memory allocation function returns the virtual address of the memory, which is valid for the process making the call.
- The memory allocator algorithm attempts to maximize the size of contiguous blocks within a heap over time to ensure that the heap doesn't get fragmented (or fragmentation is minimized).
- In order to achieve it, following scheme is used
  - Heap is partitioned into a number of fixed-size pools and a single large variable size pool.
  - The pools are often referred as power of two pools (32 bytes, 64 bytes, 128 bytes etc)
  - When an application makes a request for number of bytes, say 64 bytes, the allocator makes an attempt to allocate from the 64 byte pool.
  - If it fails it goes to the next size pool and so on.
  - If the allocation fails for all sized pools, the allocator allocates from the variable size heap.
  - The number of pools in the heap are configurable and should be tuned to your embedded system.

Virtual Memory and Paging

- Process level memory addresses are virtual address (what CPU can access)
- But the actual memory content may be in a different physical memory(ies).
- Entire memory is divided into fixed-size pages.
- Memory Management Unit (MMU) maintains a Page Table, which maps the virtual address to physical pages.
- In demand page systems, program segments are permitted to be loaded in noncontiguous memory as they are requested in fixed-size pages.
- Program code that is not held in main memory is “swapped” to secondary storage.
- When a memory reference is made to a location within a page not loaded in main memory, a page fault exception is raised.
- The interrupt handler for this exception checks for a free page slot in memory. If none is found, a page frame must be selected and swapped to disk (if altered)—a process called page stealing.
- This scheme is called Demand Paging
- Demand Paging is allows nonconsecutive references to pages via the page table, which helps to eliminate fragmentation.

Virtual Memory and Paging

Real Time Operating Systems - Storage and File Systems

BITS Pilani
Pilani Campus

Storage File Systems

- File Systems are predominantly used to provide an abstract mechanism to open, close, read and write to the storage device.
- Example:
  - NTFS and FAT : for Windows
  - EXT3/4: For Linux
  - JFFS2, TFAT, CRAMFS, RAMFS, INTFS etc: For embedded systems

File System Types

- Two Types: Journaling and Non-journaling
- Journaling File Systems
  - Maintains the deltas of the updates to the existing state
  - These deltas are called metadata, and update of these metadata is called transactions
  - A transaction is atomic.
  - Hence at any point of time the file system can be recovered to a consistent state
  - The journal updates are held on the storage device.
  - When the file system is mounted, these updates are applied to the filesystem baseline
  - It is also done periodically during the operation of the file system
  - Example: EXT3
- Non-journaling File Systems
  - Performs updates directly to the file system (i.e. File Allocation Table)
  - Hence it is very resilient and ensures consistency
  - But it can loose transactions based on scenarios
  - Example: FAT32 (used in Windows, and USB Pen Drives)

Flash Based File Systems

- In many embedded systems, Flash memory is used as the secondary storage, hence Flash-based file systems are used
- Two types of Flash devices:
  - NAND Flash
  - NOR Flash
  - NOR devices are more reliable, but costly compared to NAND devices.
- CRAMFS
  - Simple read-only compressed file system
  - Uses zlib compression
  - Can hold root file system of the embedded system
- JFFS2 (Journaling Flash File system) or YAFFS2 (Yet Another Flash File System)
  - Journaling file system
  - Continuously writes at the end for wear-leveling purpose
  - Fast booting from media
  - Error correction and robust handling during power failure
  - Small RAM foot-print
- Device Wear and Tear
  - Frequent access to the storage devices (either magnetic or solid state) will reduce the life of the device
  - Wear-leveling is used to increase the life of solid state devices. Still there is a finite number of cycles supported.
  - It is advices to use RAM based file system or ensure that the application only writes to the storage device periodically.

Real Time Operating Systems – Device Drivers

BITS Pilani
Pilani Campus

Device Drivers

Device Driver are meant for providing the services for the underlying hardware, which middlewares and applications can use.

A General Classifications of Device Drivers:

- Serial port drivers:
  - Drivers for UART interfaces
  - Used for debugging and communication between host and the target
- Network Device Drivers:
  - Required for the network interfaces
- Serial ATA drivers:
  - Disk drives are attached via Serial ATA (SATA)
  - It provides block services to the file system driver
- Flash drivers:
  - Flash drivers are special class of block device drivers
  - Meant for flash memory devices
- Bus drivers:
  - Required for busses like PCIe, AMBA etc
  - The bus driver for each bus type scans the bus for devices and allocate resources to them.
- Interrupt controller

Character vs Block Devices

A Character ('c') Device

- Is one with which the Driver communicates by sending and receiving single characters (bytes, octets).
- Examples: serial ports, parallel ports, sounds cards.

A Block ('b') Device

- Is one with which the Driver communicates by sending entire blocks of data.
- Examples: hard disks.

Handling of Memory

A Flat Memory System

- Buffers are always resident in memory.
- So, it is very simple for the device driver to access any memory.
- But biggest problem is robustness and security.

Separate Address Space for User Applications and Kernel

- Separate memory for User Applications and Kernel
- User space memory is subjected to paging, whereas, kernel space is not.
- Since drivers run at kernel space, kernel space buffers are directly accessible to the kernel.
- User space buffers are physically non-contiguous. MMU maps a large virtually contiguous buffers to a number of physical non-contiguous pages.
- For accessing user space memory from the driver, will require buffer copy between kernel and user space memories. Example: Linux function `copy_from_user`
- There are techniques of mapping user space buffers to kernel space (Example: NMAP)
- Kernel buffers themselves have a maximum contiguous size.

Real Time Operating Systems – Boot Sequence

BITS Pilani
Pilani Campus

Steps of Boot Sequence

- External Power is applied to the system
- Power settles down (called power sequencing)
- Processor reset-line is de-asserted after power sequencing
- Two types of resets:
  - Hard reset: Reset caused by power-on events
  - Soft reset: Rest caused by asserting the reset line.
- Processor fetches instruction from the reset memory location (Called reset vector)
- Then the processor start to boot in an architectural defined state.
- Thus the boot sequence starts
- 3 phases of boot sequence:
  - Early Initialization: CPU and Memory Controller Initialization
  - Advanced Init: Platform and peripheral related initialization
  - Handoff: Identifying the image and transferring the control to it.

Early Initialization

- CPU Initialization (includes initialization of Cache as well)
- Device Initialization (Devices connected to the bus)
- Memory Configuration (including initialization of Memory controller)
- Post-memory Setup
  - Memory test
  - ECC (Error Correction Codes) related initialization (if present)
  - Zeroing the memory
- Shadowing
  - From the reset vector, the execution starts from the NVRAM (Non volatile flash storage)
  - NVRAM is slow. Hence sometimes, one of the early firmware will copy the code from the NVRAM to RAM and then the execution will jump to the RAM location.
  - This is called Shadowing. It is done for reducing the boot time.
  - In Shadowing, we will eventually have same content in both NVRAM and RAM.

Advanced Initialization

Focused on device specific initializations

- GPIO
- Interrupt Controller
- Timers
- Serial ports
- PCI bus initialization
- USB
- SATA
- Network controller
- …

Boot Loaders

- Initial program had to be as small as possible, it would have minimal capabilities. It is called 1st Stage Boot Loader
- This boot loader loads another boot loader, called a 2nd Stage Boot Loader, which is more sophisticated.
- This second stage boot loader would have error checking, among possibly other features, such as giving the user a choice of operating systems to boot, the ability to load diagnostic software, or enabling diagnostic modes in the operating system.
- The boot loader will often perform some core initialization of the system hardware and will then load the operating system.
- Once the operating system is loaded, the boot loader transfers control to it and is no longer needed.
- The operating system will initialize itself, configure the system hardware (e.g., set up memory management, set timers, set interrupts), and load device drivers, if needed.

Legacy BIOS based boot sequence in Intel based systems

- BIOS : Basic Input/Output System
- When the CPU is reset at startup, the computer starts execution at memory location `0xffff0`
- Upon start-up, the BIOS goes through the following sequence:
  - Power-on self-test (POST)
  - Detect the video card's (chip's) BIOS and execute its code to initialize the video hardware
  - Detect any other device BIOSes and invoke their initialize functions
  - Display the BIOS start-up screen
  - Perform a brief memory test (identify how much memory is in the system)
  - Set memory and drive parameters
  - Configure Plug & Play devices (traditionally PCI bus devices)
  - Assign resources (DMA channels & IRQs)
  - Identify the boot device
- When the BIOS identifies the boot device (typically one of several disks that has been tagged as the bootable disk), it reads block 0 from that device into memory location 0x7c00 and jumps there.
- This first disk block, block 0, is called the Master Boot Record (MBR) and contains the first stage boot loader.
- Then MBR code scans through its partition table and loads the Volume Boot Record (VBR) for that partition.
- The VBR is a sequence of consecutive blocks starting at the first disk block of the designated partition. The first block of the VBR identifies the partition type and size and contains an Initial Program Loader (IPL), which is code that will load the additional blocks that comprise the second stage boot loader (example: NTLDR of Windows-NT)
- The second stage boot loader then loads the full fledged operating system.

Source: https://www.cs.rutgers.edu/~pxk/416/notes/02-boot.html

UEFI based boot sequence in Intel based systems

- With 64-bit platforms, UEFI (Unified Extensible Firmware Interface) replaced the legacy BIOS
- With UEFI, there is no longer a need for the Master Boot Record to store a stage 1 boot loader; UEFI has the smarts to parse a file system and load a file on its own, even if that file does not occupy contiguous disk blocks.
- UEFI reads the GUID (Globally Unique IDentifier) Partition Table (GPT), which is located in blocks immediately after block 0 (which is where the MBR still sits for legacy reasons).
- The GPT describes the layout of the partition table on a disk.
- From this, the EFI boot loader identifies the EFI System Partition.
- This system partition contains boot loaders for all operating systems that are installed on other partitions on the device.

Source: https://www.cs.rutgers.edu/~pxk/416/notes/02-boot.html

Booting in Embedded Systems

- The method vary depending upon the requirements.
- Many embedded devices will not load an operating system but have one already stored in non-volatile memory (such as flash or ROM).
- Those that load an OS, such as ARM-based Android phones, for instance, will execute code in read-only memory
- The considerations also should be given to factors such as fail-safe upgrades of images (which is not important for general purpose systems)

Selecting An RTOS

BITS Pilani
Pilani Campus

Build vs Buy

Should a commercial real-time solution be used or should one be built from scratch?

- Commercial kernels are frequently chosen, because
  - They generally provide robust services
  - They are easy to use
  - They may be portable.
  - They are wide-ranging in features and performance
  - They can support many standard devices and network protocols.
  - Often these systems come equipped with useful development and debugging tools
  - They can run on a variety of hardware and environments.
- Commercial real-time systems are best used when they can satisfy response requirements at a reasonable price, and if the system must run on a variety of platforms.
- Drawbacks of Commercial kernels
  - They are usually slower than using the hard-wired interrupt driven model because of the overhead incurred in implementing the task-control block model
  - They can include many unneeded features, increasing run-time memory requirements
  - Manufacturers may be tempted to make misleading claims, or give best case performance figures only, not revealing the worst case figures
- When the per-unit charge for commercial products is too high, or when desired features are unavailable, or when the overhead is too high – build a real-time kernel.
- Otherwise, it is better to go for Commercial RTOS always.

Selection Criteria for Commercial RTOS

Following 13 parameters can be used as an “apples-to-apples” technique for comparing RTOS.

Criterion	Description	Weight, w_i	Ratings, m_i
m_1	Minimum Interrupt Latency		
m_2	Maximum number of tasks		
m_3	Total memory required		
m_4	Scheduling mechanism		
m_5	Communication / Synchronization mechanisms		
m_6	After-sale support		
m_7	Application availability		
m_8	CPUs supported		
m_9	Source code availability		
m_{10}	Time taken for context switch		
m_{11}	Cost		
m_{12}	Compatibility with development platforms		
m_{13}	Supported networks and protocols		

$$Mean, \overline{M} = \frac{1}{13}\sum_{i=1}^{13} w_i m_i$$

Example

RTOS 'A'

Criterion	Description	Ratings, m_i	Comments
m_1	Minimum Interrupt Latency	*	CPU Dependent
m_2	Maximum number of tasks	0.5	32 task-priority levels
m_3	Total memory required	0.7	ROM: 60 K bytes
m_4	Scheduling mechanism	0.25	Preemptive only
m_5	Communication / Synchronization mechanisms	0.5	Direct message passing
m_6	After-sale support	0.5	Paid phone support
m_7	Application availability	1	Various
m_8	CPUs supported	0.8	Various
m_9	Source code availability	1	Available
m_{10}	Time taken for context switch	*	Unknown
m_{11}	Cost	0.5	$2500 + royalty fee
m_{12}	Compatibility with development platforms	*	Unknown
m_{13}	Supported networks and protocols	1	Various

Example

Selecting between 5 RTOSs A, B C, D, E

Criterion	Description	Weight	A	B	C	D	E
m_1	Minimum Interrupt Latency	1	0.5	0.8	1	0.5	1
m_2	Maximum number of tasks	0.1	0.5	0.5	0.5	1	1
m_3	Total memory required	1	0.7	0.2	0.5	1	0.9
m_4	Scheduling mechanism	0.5	0.25	0.5	0.25	1	0.25
m_5	Communication / Synchronization mechanisms	1	0.5	1	0.5	1	1
m_6	After-sale support	1	0.5	0.5	1	0.8	1
m_7	Application availability	1	1	0.75	1	1	0.5
m_8	CPUs supported	0.1	0.8	0.5	0.2	1	0.2
m_9	Source code availability	0.5	1	1	0	0.4	1
m_{10}	Time taken for context switch	1	0.5	0.5	0.5	1	0.5
m_{11}	Cost	0.4	0.5	0.5	0.1	0.1	0.7
m_{12}	Compatibility with development platforms	0.5	0.5	0.5	0.5	0.5	0.5
m_{13}	Supported networks and protocols	0.5	1	1	1	1	0.6
Mean			0.405	0.417	0.419	0.527	0.489
SD			0.269	0.295	0.384	0.382	0.364

Extra Selection Criteria

Two new criteria for selecting RTOS in a present day systems:

- Multi-core support
- Energy aware support

BITS ZG553 - Real Time System – Lecture 12

BITS Pilani
Pilani Campus

Brajabandhu Mishra

POSIX Threads

BITS Pilani
Pilani Campus

POSIX Thread

- Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process.
- The purpose of using the POSIX thread library in your software is to execute software faster.
- Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id
- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno

POSIX Thread Creation

Source: https://computing.llnl.gov/tutorials/pthreads/

- pthread_create(), creates a new thread.
- pthread_create arguments:
  - thread: An opaque, unique identifier for the new thread returned by the subroutine.
  - attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  - start_routine: the C routine that the thread will execute once it is created.
  - arg: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Example:

```
void *PrintHello(void *threadid) {
   …
}
int main (int argc, char *argv[]) {
   pthread_t threads;
   int rc;
   printf("In main: creating thread \n");
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *)0);
   if (rc){
   printf("ERROR; return code from pthread_create() is %d\n", rc);
   exit(-1);
   }
}
```

POSIX Thread Creation – Thread Attributes

Source: https://computing.llnl.gov/tutorials/pthreads/

- By default, a thread is created with certain attributes.
- Some of these attributes can be changed by the programmer via the thread attribute object.
- pthread_attr_init and pthread_attr_destroy are used to initialize/destroy the thread attribute object.
- Other routines are then used to query/set specific attributes in the thread attribute object.
- Attributes include:
  - Detached or joinable state
  - Scheduling inheritance
  - Scheduling policy
  - Scheduling parameters
  - Scheduling contention scope
  - Stack size
  - Stack address
  - Stack guard (overflow) size

POSIX Thread – Terminating Threads

Source: https://computing.llnl.gov/tutorials/pthreads/

Several ways in which a thread may be terminated:

- The thread returns normally from its starting routine. Its work is done.
- The thread makes a call to the pthread_exit subroutine
- The thread is canceled by another thread via the pthread_cancel routine.
- The entire process is terminated due to making a call to either the exec() or exit()
- If main() finishes first, without calling pthread_exit explicitly itself.

Example:

```
void *PrintHello(void *threadid) {
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
int main (int argc, char *argv[]) {
   pthread_t threads;
   int rc;
   printf("In main: creating thread \n");
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *)0);
   if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
   }
   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

POSIX Thread – Joining and Detaching Threads

- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() subroutine blocks the calling thread until the specified thread id thread terminates.

Source: https://computing.llnl.gov/tutorials/pthreads/

POSIX Thread – Joining and Detaching Threads

- When a thread is created, one of its attributes defines whether it is joinable or detached.
- Only threads that are created as joinable can be joined.
- If a thread is created as detached, it can never be joined.
- To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used. The typical 4 step process is:
  - Declare a pthread attribute variable of the pthread_attr_t data type
  - Initialize the attribute variable with pthread_attr_init()
  - Set the attribute detached status with pthread_attr_setdetachstate()
  - When done, free library resources used by the attribute with pthread_attr_destroy()

<u>Recommendations</u>

- If a thread requires joining, consider explicitly creating it as joinable.
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state. Some system resources may be able to be freed.
- There is a definite problem if main() finishes before the threads it spawned if you don't call pthread_exit() explicitly.
  - All of the threads it created will terminate because main() is done and no longer exists to support the threads.
  - So main() (or the master thread) should wait for the child threads to complete. It can be achieved by 'joining'

POSIX Thread Example – Hello World program

pthread_helloworld.c

Pthread Mutex

- Mutexes are used for syncronizing pthreads.
- The pthread_mutex_init() routine is called to initialize a mutex.
- The pthread_mutex_lock() routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- pthread_mutex_unlock() will unlock a mutex if called by the owning thread. An error will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread

pthread mutex - Example

pthread_mutex_ex.c

Makefile

POSIX Semaphores

- Declare the semaphore

  Example: sem_t sem_name;

- Initalize the semaphore

  int sem_init(sem_t *sem, int pshared, unsigned int value)

  Example: sem_init(&sem_name, 0, 10);

  Value will be incremented in a 'post' and decremented in a 'wait' operation.

- Wait on a semaphore

  sem_wait: int sem_wait(sem_t *sem);

  Example of use: sem_wait(&sem_name);

If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls

- Increment the value of a semaphore, use

  sem_post: int sem_post(sem_t *sem);

  Example of use: sem_post(&sem_name);

It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

- Destroy a semaphore, use

  int sem_destroy(sem_t *sem);

  Example: sem_destroy(&sem_name);

POSIX Semaphores - Example

posix_semaphore.c

BITS ZG553 - Real Time System – Lecture 12

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Hands-on exercise using Keil RL-RTX RTOS

BITS Pilani
Pilani Campus

Keil RL-RTX

- RTOS from ARM, designed for ARM based microcontrollers.
- Links for Keil RL-RTX:
  - http://www.keil.com/support/man/docs/rlarm/rlarm_ar_artxarm.htm
  - http://www.keil.com/product/brochures/rl-arm_gs.pdf
- Download Keil uVision 4 (IDE to be used for programming using Keil RL-RTX) from: https://www.keil.com/demo/eval/armv4.htm

Starting to Write an Application Using Keil RL-RTX

- Download and Install Keil uVision 4.
- Open Keil uVision 4.
- Create a Keil uVision4 Project, with desired CPU (CPU used here is LPC2378).
- Click 'Yes' for the 'Create Startup file'. It will add the file meant for required initilization code.
- Configure the target options as following.

Options for Target 'Target 1'

Target Option:

Xtal (MHz) : You may keep the default value (12 MHz)

Code Generation: ARM Mode

Check 'Use MicroLib'

Operating System: RTX Kernel

Use memory layout as default

Linker Option:

Use Memory Layout from Target Dialog

Debug Option:

Use Simulator

Starting to Write an Application Using Keil RL-RTX (contd.)

- Copy RTX config file from the C:\Keil\ARM\RL\RTX\Cpnfig\ to the project directory. For LPC2378, the RTX Config file name is RTX_RL_LPC23xx.c. Right Click 'Source Group' and select this file.
- Open the startup file (here it is LPC2300.s). Comment the SWI_Handler and import the same like the following.

```
IMPORT  SWI_Handler      <-- Add this line
Undef_Handler   B      Undef_Handler
;SWI_Handler     B      SWI_Handler<--- Comment this line
PAbt_Handler    B      PAbt_Handler
DAbt_Handler    B      DAbt_Handler
IRQ_Handler     B      IRQ_Handler
FIQ_Handler     B      FIQ_Handler
```

- Create your source code as given below.

  File->New to create a new text file

  Write the code on the Text File

  Save the file as a file with .c extension

  Right Click 'Source Group' and Add the .s file to the Source Group

Example

- Please refer to ‘RL-RTX-Example-Priority.rar’
- It is a small program using RL-RTX demonstrating the priority driven scheduling.

BITS ZG553 - Real Time System – Lecture 10

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Real Time Operating Systems (RTOS)

BITS Pilani
Pilani Campus

Operating Systems

The operating system controls resources:

- who gets the CPU;
- when I/O takes place;
- how much memory is allocated.

The most important resource is the CPU itself.

- CPU access controlled by the scheduler.

Real Time Operating System

- Real-time operating systems must provide three specific functions with respect to tasks:
  - scheduling
  - dispatching
  - intercommunication and synchronization

- A scheduler determines which task will run next in a multitasking system
- A dispatcher performs the necessary bookkeeping to start that task.
- Inter-task communication and synchronization assures that the tasks cooperate.

Layers of Operating Systems

Operating System	User Interface, Security Features, File Management etc
Executive	Private Memory Blocks, I/O Services, Supporting Features etc
Kernel	Inter-task Communication and Synchronization
Microkernel	Task Scheduling and Despatching

Pseudokernels

- Real-time multitasking in rudimentary form can be achieved without operating systems and interrupts.
- These systems are known as pseudo-kernels.
- In this approach, the resultant systems are often highly predictable.
- Hence these are sometime preferred in low-end embedded systems.

Pseudokernels

System without O/S and Interrupt:

Polled Loop:

```
while (1)
{ /* do forever */
   if (packet_here) /* check flag */
   {
   process_data(); /* process data */
   packet_here=0; /* reset flag */
   }
}
```

Pros:

- Fast response

Cons:

- Can fail due to burst of events
- Generally not sufficient to handle complex systems
- Waste of CPU time, especially when event being polled occurs infrequently

Pseudokernels

Polled Loop with Delay:

- Used to treat a problematic events exhibiting Contact Bounce behaviour.
- Any electromechanical switch can't change its state instantaneously. It needs some time to settle down the oscillations. It is called Contact Bounce behaviour.
- So a delay equal to the time little more than the contact bounce behaviour disappears, is introduced in the polled loop

In the following example, contact bounce behaviour disappears after 20 ms. So delay of 20 ms + 1 ms (to be on the safer side) = 21 ms is introduced.

```
while(1)
{ /* do forever */
   if(flag) /* check flag */
   {
   process_event(); /* process event */
   pause(21); /* wait 21 ms */
   flag=0; /* reset flag */
   }
}
```

- The `flag` is set by an external device.
- Delay (`pause` functionality) is generated by a timer.

Pros:

- Excellent for high speed data channels, especially when the events occurs at widely dispersed intervals and CPU is dedicated to handle the data channel

Cons:

- Can fail due to burst of events
- Generally not sufficient to handle complex systems
- Waste of CPU time, especially when event being polled occurs infrequently

Pseudokernels

Cyclic Code Structure:

```
while (1)
{ /* do forever */
   Process_1();
   Process_2();
   ...
   Process_N();
}
```

- Tasks are executed in round-robin fashion
- Higher priority can be given to a task by repeating it.
- The task list can be managed by a 'pseudo-operating system', as the tasks are created and completed.
- Inter-process communication is achieved through global variables
- Suitable for simplest real time systems, but not suitable for complex systems, since there are difficulties in uniformly dividing the tasks and the response times are lengthy.

Pseudokernels

Code driven by FSMs (Finite State Machines)

```
void process_a(void)
{
   for(;;)
   {
      switch(state_a)
      {
      case 1: phase_a1();
         break;
      case 2: phase_a2();
         break;
      case 3: phase_a3();
         break;
      case 4: phase_a4();
         break;
      case 5: phase_a5();
         break;
      }
   }
}
```

Pseudokernels - Coroutines

- Tasks are separated into multiple phases based on their states in the FSM.
- After executing a phase, the task is suspended and dispatcher is invoked.
- Dispatcher holds the list of the tasks to be executed in a round-robin fashion. It picks up the next task from the list and executes it.
- The states are stored as global variables.
- Inter-task communication and synchronization are managed by global variables.
- Programming languages like Ada have efficient built-in constructs for implementing coroutines.

Pros

- Are easiest type for fairness scheduling.
- Easy to determine the response time, since calls to the dispatched is given at known intervals.

Cons

- Fairness scheduling is not suitable for real-time system, since the tasks scheduling should be made depending upon importance and urgency.
- Final number of tasks need to be known beforehand.
- Tasks can’t be always decomposed uniformly.

Pseudokernels - Coroutines

```
void process_a(void)
{
   for(;;)
   {
      switch(state_a)
      {
      case 1: phase_a1();
         break;
      case 2: phase_a2();
         break;
      case 3: phase_a3();
         break;
      case 4: phase_a4();
         break;
      case 5: phase_a5();
         break;
      }
   }
}
```

```
void process_b(void)
{
   for(;;)
   {
      switch(state_b)
      {
      case 1: phase_b1();
         break;
      case 2: phase_b2();
         break;
      case 3: phase_b3();
         break;
      case 4: phase_b4();
         break;
      case 5: phase_b5();
         break;
      }
   }
}
```

- Dispatcher is running in the background, which is scheduling `process_a` and `process_b` alternatively
- Dispatcher maintains '`state_a`' and '`state_b`', which are global variables
- Inter-task communication and synchronization are managed by global variables.

Pseudokernels – 'Interrupt-only Systems

```
void main(void)
{
   init(); /*initialize system, load interrupt handlers */
   while(TRUE);   /* infinite wait loop */
}
void intl (void)  /* interrupt handler 1 */
{
   save(context);     /* save context on stack */
   task1();   /* execute task 1 */
   restore(context);/* restore context from stack */
}
void int2(void)   /* interrupt handler 2 */
{
   save(context);     /* save context on stack */
   task2();   /* execute task 2 */
   restore(context);/* restore context from stack */
}
```

- Main program is a jump-to-self instruction (infinite loop)
- Tasks in the systems are scheduled via either H/W or S/W interrupts
- Access to resources is usually controlled by disabling interrupts around any code that reads or writes to the shared resource.
- Snapshot of the current system state (context) for the task must be preserved while switching to other task. It is called context switching.

Interrupt Service Routine

- Steps inside an ISR (Interrupt Service Routine)
  - Disables all interrupts
  - Runs code to service the event
  - Clears the interrupt flag that got it called
  - Re-enables interrupts
  - Exits so the processor can go back to its running task
- Interrupt handlers should not take a lot of time to execute, otherwise it will be keeping the external device waiting for longer period of time.
- An ISR (Interrupt Service Routine) is said to be reentrant if, while the ISR is handling an interrupt, the same interrupt can occur again and the ISR can process the second occurrence of the interrupt before it has finished processing the first.
- Reentrant code is not allowed to
  - Use any data in non-atomic way except when they are saved on the stack
  - Call any other code that is not re-entrant.
  - Use any h/w resources in a non-atomic way.
  - Change its own code.
- An atomic operation refers to a group of sub-operations that can be combined to appear as a single non-interruptable operation.
- Normally inside the interrupt handlers, interrupts are disabled during context switching period.

Context Switching

- Context switching is the process of saving and restoring sufficient information for a real-time task so that it can be resumed after being interrupted.
- Context is ordinarily saved to a stack data structure.
- Context-switching time is a major contributor to response time.
- The rule for saving context is simple: save the minimum amount of information necessary to safely restore any process after it has been interrupted.
- The context data typically includes
  - Contents of general registers
  - Contents of the program counter
  - Contents of coprocessor registers (if present)
  - Contents of memory page register
  - Images of memory-mapped I/O locations (mirror images)

Preemptive Priority Systems

- A higher-priority task is said to preempt a lower-priority task if it interrupts the lower-priority task.

- The priorities assigned to each interrupt are based on the urgency of the task associated with the interrupt.

- Prioritized interrupts can be either fixed priority or dynamic priority.
  - Fixed priority systems are less flexible, since the task priorities cannot be changed.
  - Dynamic-priority systems can allow the priority of tasks to be adjusted at runtime to meet changing process demands.

- Preemptive-priority schemes can suffer from resource hogging by higher priority tasks. The lower-priority tasks are said to be facing a problem called starvation.

- A special class of fixed-rate preemptive-priority interrupt-driven systems, called rate-monotonic systems, comprises those real-time systems where the priorities are assigned so that the higher the execution frequency, the higher the priority.

  Example: In the aircraft navigation system, the task that gathers accelerometer data every 10 milliseconds has the highest priority. The task that collect gyro data, and compensates these data and the accelerometer data every 40 milliseconds, has the second highest priority. Finally, the task that updates the pilot's display every second has lowest priority.

Hybrid Systems

- Hybrid systems include interrupts that occur at both fixed rates and sporadically.
- The sporadic interrupts can be used to handle a critical error that requires immediate attention, and thus have highest priority.
- Another type f hybrid system consists of a combination of round0robin and preemptive priority systems. – generally found in commercial operating systems.

Foreground / Background Systems

- Disadvantages of Interrupt driven systems
  - Time is wasted in jump to self loop
  - Difficulty in providing advance services
- Forground/background systems are 'Interrupt Driven Systems with small variation':
  - Jump-to self routine is replaced by a low priority code that does useful processing.
- It Constitutes
  - a set of interrupt-driven or real-time processes called the foreground
  - a collection of non-interrupt-driven processes called the background
- Background task is fully pre-emptable by any foreground task (hence is the lowest priority task)
- Foreground tasks run in round-robin, pre-emptive priority, or hybrid fashion.
- Background task generally does
  - Initialization
  - House keeping,
  - Low priority tasks such as parameter entry through keypad, low priority display updates etc
- Initialization includes
  - Disable interrupts
  - Setup interrupt vectors and tables
  - Initialize peripherals and other hardware
  - Perform self-tests
  - Perform necessary software initializations
  - Enable interrupts

Task Control Block (TCB) Model

- Each task is associated with a data structure – a task control block (TCB).
- TCB contains context. The system stores TCBs in one or more data structures, such as a linked list.
- TCB model can be used in round-robin, preemptive priority or combination systems, although it is generally associated with round-robin systems with a single clock.
- Is the most popular method for implementing commercial, full-featured, real-time operating systems.
- The operating system manages TCBs by keeping track of the status of each task.
- A task can typically be in any one of the following states:
  - executing
  - ready
  - suspended (or blocked)
  - dormant (or sleeping)
- Every hardware interrupt and every system level call (such as a request on a resource) invokes the real-time operating system.
- The operating system is responsible for maintaining a linked list containing the TCBs of all the ready tasks, and a second linked list of those in the suspended state.
- It also keeps a table of resources and a table of resource requests.

The difference between TCB model and the Interrupt-handler model is that the resources are managed by the operating system in the TCB model, whereas in interrupt-handler model, tasks track their own resources.

Task Control Block (TCB) Model

TCB

Task Identifier
Priority
Status
Work Registers
Program Counter
Status Register(s)
Stack Pointer
Pointer to Next TCB

Process and Thread

Process

- A process is an abstraction of a running program and is the logical unit of work scheduled by the operating system.
- Typically represented by a data structure that contains at least a state of execution, an identity (real-time), attributes (e.g., execution time), and the resources associated with it.

Thread

- A thread is a lightweight process that shares resources with other processes or threads.
- Each thread must "reside" within some process and make use of the resources of that process.

Process States

Scheduling Policy defines how processes are selected for promotion from the ready state to the running state

Timer and Clock Services

- Timing services are essential for Real Time Software
- Example: a diagnostic task checks the health of the system periodically.
- A system call '`delay`' is commonly used to suspend a task.
- The parameter passed to this function is an integer specifying the number of '`ticks`'.
- A timer circuit is configured to interrupt the CPU at a fixed rate and the internal system time is incremented at each timer interrupt. The interval of time for which the timer interrupt is programmed is called '`tick`' or time resolution.

Linux Kernel Scheduler

- Linux kernel scheduler supports
  - Nonpreemptible scheduler (initial version of Linux kernel supported this policy only)
  - Preemptible scheduler (introduced from version 2.6) – required for real-time tasks/threads
  - Completely fare scheduler (CFS) (introduced in the version 2.6.23) – Primarily for desktops
- Following function controls the behaviour of process scheduling discipline:

```
int sched_setscheduler (pid_t pid, int policy,
                        const struct sched_param *param);
```

Non-real time policies supported:

SCHED_OTHER: Standard round-robin policy

SCHED_BATCH: Batch style execution – it doesn't preempt processes often

SCHED_IDLE: Runs the task at very low priority background task

Real time policies supported:

SCHED_FIFO: First in first out

(when the process becomes runnable, it will be inserted at the end of the list for its priority)

SCHED_RR: Round robin

Please note that this function is called per process basis. That means different processes can be scheduled based on different scheduling policies

BITS ZG553 - Real Time System – Lecture 11

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Inter-Process Communication (IPC) and Synchronization

BITS Pilani
Pilani Campus

Communication/Synchronization between Processes

- Semaphores and Mutexes
- Spin Lock
- Mailboxes
- Message Queues
- Pipes
- Sockets
- Buffers

Critical Regions

- Multitasking systems are concerned with resource sharing.
- When resources can only be used by one task at a time, and use of the resource cannot be interrupted they are said to be “serially reusable”
- Serial reusable devices include certain peripherals, shared memory, and the CPU.
- The CPU protects itself against simultaneous use.
- Code that interacts with the other serially reusable resources is called a “critical region.”
- If two tasks enter the same critical region simultaneously, a catastrophic error can occur.
- Simultaneous use of a serial reusable resource results in a “collision.”
- The concern is to provide a mechanism for preventing collisions.

Critical Regions

- Consider two C programs, Task_A and Task_B, in a round-robin system.
- Task_B outputs the message "I am task_B" and Task_A outputs the message "I am Task_A."
- In the midst of printing, Task_B is interrupted by Task_A, which begins printing. The result is the incorrect output: I am I am Task_A Task_B
- More serious complications could arise if both tasks were controlling devices in an embedded system.

Source: P. A. Laplante & S. J. Ovaska, Real-Time Systems Design and Analysis: Tools for the Practitioner, Wiley, 4th edition

Critical Regions

- A segment of a job that begins with a lock and ends at a matching unlock is called a ***critical section***
- Resources are released in last-in-first-out order
- A critical section that is not included in other critical sections is called an *outermost critical section*
- Critical sections are denoted by [*R, η; e*], where
  - *R* gives the *name* and
  - *η* the *number of* units of a resource and
  - *e* the *(maximum) execution time* of the critical section
- If there is only one unit of a resource the simpler notation [*R; e*] *is used*

Semaphores

- The most common method for protecting critical regions involves a special variable called a semaphore.
- A semaphore *S* is a memory location that acts as a lock to protect critical regions.
- Two operations, wait and signal are used either to acquire or to release the semaphore respectively.
- Traditionally, one denotes the wait operation as P(S) and the signal operations V(S).

<u>Semaphore vs Mutex</u>

The only difference between a "mutex" and a semaphore is that the mutex has to be unlocked by the same thread that locked it.

Can we use a mutex in ISRs ???

Semaphores

Task 1	Task 2
.	.
.	.
.	.
P(S)	P(S)
critical region 1	critical region 2
V(S)	V(S)
.	.
.	.
.	.

Semaphores

- Consider a preemptive priority system with separate measurement channels for acceleration and temperature and a single A/D converter to be used by two tasks, each measuring these two quantities separately periodically.
- A binary semaphore ‘s’ is used to protect the serially usable resource A/D converter.

<u>Task 1</u>

```
…
wait(&s);
select_channel(acceleration);
a_data = ad_conversion();
signal(&s);
…
```

<u>Task 2</u>

```
…
wait(&s)
select_channel(temperature);
a_data = ad_conversion();
signal(&s);
…
```

Semaphore Implementation

```
P(S)
{
   temp=FALSE;
   while (temp==FALSE)
   {
       disable interrupts
       temp=S;
       if (temp==TRUE)
       {
           S=FALSE;
           enable interrupts;
           return
       }
       enable interrupts;
   }
}
```

```
V(S)
{
    S=TRUE;
}
```

The code between 'disable interrupt' and 'enable interrupt' is ***atomic***. (why ?)

Binary & Counting Semaphores

- **Binary Semaphore:**
  - Can take only two values: 0 and 1
  - Can protect only one resource
- **Counting Semaphore:**
  - Can take multiple values
  - Can protect a pool of resources

Counting Semaphore Logic

```
void MP(int S);
{
    S=S-1;
    while (S < 0);
}

void MV(int S)
{
    S=S+1
}
```

Common mistakes while using Semaphores

Common mistakes committed by programmers while using semaphores (Simon, 1999)

- The use of a specific semaphore is forgotten
- A wrong semaphore is used in error
- The semaphore is held for an overly long time
- The semaphore used is not released at all

Deadlock

task 1

P(s)

critical region 1

.

.

.

P(r)

critical region 2

.

.

V(r)

.

V(s)

task 2

.

P(r)

critical region 2

.

.

.

P (s)

critical region 1

.

V(s)

.

V(r)

Four necessary conditions for deadlock

Four conditions are necessary for deadlock:

- Mutual exclusion:
  - At least one resource must be held in a non-sharable mode i.e. one process can use the resource at a time.
- Hold and wait:
  - There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are held by other processes
- No preemption:
  - Resources can't be pre-empted i.e. resources can be released only voluntarily by the processes holding it.
- Circular wait:
  - There must exist a set of processes {P0, P1, …. Pn} of waiting processes such at P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held be P2, and so on.

Eliminating any one of the four necessary conditions will prevent deadlock from occurring.

Four necessary conditions for deadlock

Condition	Solution	Possible Adverse Consequences
mutual exclusion	allow sharing, use spoolers	starvation
circular wait	device ordering	starvation
hold and wait	pre-allocate resources, no hold and wait	resources not always known *a priori*, poor resource utilization
no preemption	allow preemption	starvation

Deadlock Avoidance

- An alternative method is to require additional information about how resources are to be requested.
- With the knowledge of the complete sequence of request, we can decide at run time whether to allocate the resource to the requesting process or not – so that deadlock can't occur
- A dead-lock avoidance algorithm dynamically examines the resource-allocation to ensure that deadlock doesn't occur

Deadlock Avoidance: Safe and Unsafe States

- A stafe is safe if the system can allocate resources to each process (up to maximum) in some order and still avoid deadlock.
- A sequence of processes $<P_1, P_2, \ldots, P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $i < j$.
- A system is in a *safe state* only if there exists a *safe sequence*.

Example

Consider a system with 12 magnetic tape drives and 3 processes P_0, P_1 and P_2. These processes require maximum of 10, 4 and 9 tape drives respectively.

Suppose at time t_0, P_0, P_1 and P_2 hold 5, 2 and 2 tape drives respectively (Thus 3 tape drives are remaining). If P1 asks its maximum needed tape drives, additional 2 can be allocated to it. Then it can release them and can be allocated to P0, since 5 tape drives will be available that time. Once P0 releases them, 10 tape drives will be available and then they can be allocated to P2, which require 7 additional tape drives at maximum. The system is in safe condition.

Processes	Max Requirement	t0 (safe)	t1 (unsafe)
P0	10	5	5
P1	4	2	2
P2	9	2	3
Total Available = 12			

If at t1, P2 requests for one more tape drive, with all the processes holding the existing tape drives, then the system will get into unsafe state. Because, at this point total 2 tape drives are free and can satisfy P1's maximum need. But once P1 returns them, total free tape drives will be 4 in the system and will not be able to full-fill the maximum request of P0.

Deadlock Avoidance : Banker's Algorithm

Table For One Resource

Initial State

Processes	Max Requirement	Allocated	Needed
A	6	0	6
B	5	0	5
C	7	0	7
Total Available			10

Safe State

(because remaining requirement of A or B can be satisfied with the available resources, once they release, the requirement of C can be satisfied))

Processes	Max Requirement	Allocated	Needed
A	6	2	4
B	5	3	2
C	7	1	6
Total Available			4

Processes	Max Requirement	Allocated	Needed
A	6	4	2
B	5	3	2
C	7	2	5
Total Available			1

Unsafe State

(because remaining requirement of none of the processes can be satisfied with the available resources)

The same can be extended for multiple resources

Banker's Algorithm

- It put excessive overheard, so not suitable for real-time systems.
- Also it requires the resource requirements to be known a priori, which may not be possible always.

Deadlock – Best way is to avoid it

- Best way to deal with deadlock is to avoid it.
- If semaphores protecting critical resources are implemented by mailboxes with time-outs, then deadlocking cannot occur. But starvation of one or more tasks is possible.
- Then the following approach is recommended can help avoid deadlock.
  - Minimize the number of critical regions as well as minimizing their size.
  - All processes must release any semaphore before returning to the calling function.
  - Do not suspend any task while it controls a critical region
  - All critical regions must be error free.
  - Do not lock devices in interrupt handlers.
  - Always perform validity checks on pointers used within critical regions. Pointer errors are common in certain languages, like C, and can lead to serious problems within the critical regions.

Lock and Unlock Functions

- A lock() function disables the OS Scheduler.
- So, the current task is not preempted and no other new task is started after lock().
- The OS Scheduler is enabled by calling unlock() function.
- Example: Spin Lock
  - ‘Spin Lock’ when executed, sets the lock flag in the O/S.
  - The current task continues execution while lock flag is set, till ‘spin unlock’ function executes.
  - Polling is used in stead of putting a process to sleep here

Mutex vs Spin Lock

- Mutex
  - When a thread tries to lock a mutex and it doesn't succeed, it goes to sleep allowing other threads to run.
  - Sleep overhead is substantial, due to context switching.
  - Hence it is not suitable for frequent lock and unlock operations (in other ward smaller critical section)
- Spin Lock
  - When a thread tries to lock a spin lock, it continuously tries it in a polling mode until it is successful, not allowing other threads to run until the CPU quantum of time allotted to it expires.
  - So overhead is less, since no context switching is involved during polling.
  - On the other hand it wastes CPU time.
  - Hence it is not suitable for longer critical sections.
- Hence Spin Locks are more suitable for multi-processor / multi-core environments, where critical sections are smaller, since different portion of the code are executed in different processors / cores.
- Mutexes are more suitable in case of single-core / single processor systems.

Mail Boxes

- A mutually agreed upon memory location that one or more tasks can use to pass data, or more generally for synchronization.
- A mailbox message may have a header to identify the message-type specification.
- Tasks rely on the kernel to allow them to write to the location via a post operation or to read from it via a pend operation.
- The difference between the pend operation and polling is that the pending task is suspended while waiting for data to appear -- eliminates the busy waiting condition.
- If the mail box has size one message, then the mailbox become full after post and empty after pend.
- Mailboxes are available in most commercial RTOS.

Message Queues

- A message queue is like a mailbox with capacity of holding multiple messages in a queue.
- Either one process or set of process can use the queue.
- A write into the queue is a message post operation.
- A read from the queue is a message pend operation.
- Some RTOS don’t differentiate between mailboxes and message queues.

Pipes

- A message pipe is a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks.
- One task uses a function to write to pipe back-end of the pipe.
- While other tasks uses a function to read from the front-end of the pipe.

Sockets

- Pipes are useful for communicating between processes in the same machine.
- If communication is required between the processes in two different machines, then sockets are used.
- Each socket has a source address and a port number.
- Use TCP/IP framework for communication

Buffers (FIFOs)

- Buffers are used to pass data between two processes, where one process is the producer and other one is consumer
- Practically production and consumption rates are unequal
- Global variables are simple and fast, but have collision potential.
- Example, one task may produce data at a constant 100 units per second, whereas another may consume these data at a rate less than 100 units per second.
  - Assuming that the production interval is finite (and relatively short), the slower consumption rate can be accommodated if the producer fills a storage buffer with the data.
  - The buffer holds the excess data until the consumer task can catch up.
  - The buffer can be a queue or other data structure, including an unorganized mass of variables.
  - If consumer task consumes this information faster than it can be produced, or if the consumer cannot keep up with the producer, problems occur.
- Selection of the appropriate size buffer and synchronization mechanisms is critical in reducing or eliminating these problems.

Buffer Structure

Double Buffering Schemes

Used when time-relative (correlated) data need to be transferred between cycles of different rates, or when a full set of data is needed by one process but can only be supplied slowly by another process.

<u>Example: Displaying Video Frames</u>:

- Decoder fills one frame buffer
- Then the buffer is switched, and the decoded fills up the new one
- At the same time display module displays the already filled buffer, since it can only display a fully filled buffer.
- Ping-pong continues

Double buffering configuration. Two identical buffers are filled and emptied by alternating tasks.

Ring Buffers

- A circular queue or ring buffer can be used to solve the problem of synchronizing multiple reader and writer tasks.
- Simultaneous input and output to the list are achieved by keeping head and tail indices.
- **Data are loaded at the tail and read from the head**.
- Can be used in conjunction with a counting or binary semaphore to control multiple requests for a single resource such as memory blocks, modems, and printers.

Ring Buffers

- Can be easily implemented using linked list in software
- In Hardware, it is implemented as FIFOs
- Write at the tail and read from the head
- Accesses to the queue should be protected by semaphores
- Overflow and Underflows are indicated by interrupts / events
- It is a good design practice to indicate Overflows and Underflows before they actually occurs

BITS ZG553 - Real Time System – Lecture 14

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Software Requirements

BITS Pilani
Pilani Campus

Introduction

- Requirements engineering is the subdiscipline of software engineering
- Concerned with determining the goals, functions, and constraints of software systems.
- The goal is to create a requirements specification that is complete, correct, and understandable to both customers and developers.
- While the development of the solution is considered increasingly as commoditized activities (if requirements are made clear), hence can be outsourced, requirement engineering is a crucial activity and therefore should be conducted by the development organisation – together with an appropriate group of customer representatives.

Requirement Engineering Process

Adapted from Sommerville (2000)

Types of Requirements

As per IEEE, a general software requirement for a real-time software are of following types:

- C1. Functional : *Fundamental actions of features*
- C2. External Interfaces: *Inputs and Outputs*
- C3. Performance : *Static and Dynamic numerical requirements*
- C4. Logical database: *Logical requirements related to database*
- C5. Design constraints: *Restrictions*
- C6. Software system attributes: *Various quantifiable attributes*

Types C2-C6 are categorized under 'Non-Functional' Requirements.

Functional Requirements

- All system inputs
- Exact sequence of operations and responses (outputs) to normal and abnormal situations for every input possibility
- May use case-by-case description or other general form of description (e.g. using universal quantification, use cases, user stories)

External Interface Requirements

- name of item
- description of purpose
- source of input or destination of output
- valid range, accuracy, and/or tolerance
- units of measure
- timing
- relationships to other inputs/outputs
- screen formats/organization
- window formats/organization
- data formats
- command formats

Performance Requirements

- Static and dynamic requirements placed on the software or on human interaction with the software as a whole .
- Might include:
  - the number of simultaneous users to be supported
  - the numbers of transactions and tasks the amount of data to be processed within certain time periods for both normal and peak workload conditions

…

…

Logical database requirements

- Types of information used by various functions such as
  - Frequency of use
  - Accessing capabilities
  - Data entities and their relationships
  - Integrity constraints
  - Data retention requirements

Design constraint requirements

Related to :

- Standards compliance
- Hardware limitations

Software system attribute requirements

- Reliability
- Availability
- Security
- Maintainability
- Portability
- Any other miscellaneous requirements you can think of …

Requirements specification for real-time systems- Approaches

There is no one way to do it. Various ways include:

- Top-down process decomposition or structured analysis
- Object-oriented approaches
- Program description languages (PDL) or pseudo-code
- High-level functional specifications that are not further decomposed
- Ad hoc techniques, including simply natural language and mathematical description, and are always included in virtually every system specification

3 Types of Specification Techniques

- Formal: Involve rigorous mathematical or logical modeling
- Informal: Can't be translated into rigorous mathematical notations and associated rules (e.g. specifications involving flowcharts where the decision blocks involve natural languages)
- Semiformal: Methods which don't fall into the above two categories e.g. models involving UML.

Boundaries between these are sometime unclear.

Formal Methods in System Specification

- Contribute significantly to requirements formulation and validation by use of extensive mathematical techniques
- By nature, real-time systems usually contain some formalism of mathematical expression of interaction with the operating environment.
- Three typical use of formal methods
  - Consistency checking: System behavioral requirements are described using a mathematically-based notation.
  - Model checking: Uses state machines to verify whether a given property is satisfied under all conditions.
  - Theorem proving: Axioms of system behavior are used to derive a proof that a system will behave in a given way.

Finite State Machine (FSM)

A finite state machine can be described mathematically by a five tuple, $M = \{ S, i, T, \sum, \delta \}$

- S is a finite, non-empty set of states
- i is the initial state (i is a member of S),
- T is the set of terminal states,
- $\sum$ is an alphabet of symbols or events used to mark transitions,
- δ is a transition function that describes the next state of the machine given the current state, and a symbol from the alphabet (an event).

Finite State Machine - Example

S={**calibration**, **diagnostic**, **operational**}, *i*=**calibration**, *T*=*S* and

∑ = {op_op, op_cal, error}. The transition function *δ* can be described by a set of triples of the form (state, signal, next_state).

Source: P. A. Laplante, Real-Time Systems Design and Analysis, Wiley, 3rd edition

State Charts

Statechart = FSM + Depth + Orthogonality + Broadcast Communication

- FSM is a finite state machine,
- Depth represents levels of detail
- Orthogonality represents the existence of separate tasks
- Broadcast communication is a method for allowing different orthogonal processes to react to the same event.

The statechart resembles a finite state machine where each state can contain its own FSM that describes its behaviour.

Statechart is part of UML.

State Charts

Various components of the statechart:

- The FSM is represented in the usual way, with capital letters or descriptive phrases used to label the states.
- Depth is represented by the insideness of states.
- Broadcast communications are represented by labelled arrows, in the same way as FSMs.
- Orthogonality is represented by dashed lines separating states.
- Symbols *a,b, . . . , z* represent events that trigger transitions, in the same way that transitions are represented in FSMs.
- Small letters within parentheses represent conditions that must be true for the transitions to occur.

State Chart - Example

Source: P. A. Laplante & S. J. Ovaska, Real-Time Systems Design and Analysis: Tools for the Practitioner, Wiley, 4th edition

Petri Nets

- First introduced by Carl Adam Petri in 1962.
- A diagrammatic tool to model concurrency and synchronization in distributed systems.
- Based on strong mathematical foundation.
- Consists of three parts:
  - Circles: Called Places (means States)
  - Rectangles: Transitions (or events)
  - Arcs (Arrows): Connects Places to Transitions
- Each Arc is labeled with a number. If there is no label on an arrow, it is considered as '1'.
- Tokens are represented as black dots, which move from one place to other at each transition.
- Change of transition at an event is called firing of event.
- Markings (m_0, m_1, …) represents initial data count (represented by tokens) in all the places
- At each transition, tokens move from one place to other based on the direction of the arrow.

Petri Nets

Firing Table					
	P_1	P_2	P_3	P_4	P_5
m_0	1	1	0	0	0
m_1	0	0	1	0	0
m_2	0	0	0	1	1

m_0

Petri Nets

m_1

m_2

Petri Nets – Example – Vending Machine

- Simple Vending m/c, accepting Rs 5/- and Rs 10/-.
- All snacks are of Rs 5/- or Rs 10/-

Semi-Formal Methods

Two wide-spread approaches:

- Structured Analysis and Structured Design (SA/SD)
  - Closely associated with procedural programming languages like C
- Unified Modeling Language (UML)
  - Closely associated with object oriented programming languages like C++ and Java

UML is covered extensively in the subject like 'Software for Embedded Systems', 'Object Oriented Analysis and Design' etc.

Structured Analysis and Structured Design (SA/SD)

Three complementary models to describe a real-time system.

- M1. Environmental Model
  - Context Diagram
  - Event List
  - Natural Language
- M2. Behavioural Model
  - Data Flow Diagram
  - Control Flow Diagram
  - Entity Relationship Diagram
  - Data Dictionary
  - Process Specification
  - Control Specification
  - State Transition Diagram
  - Natural Language
- M2. Implementation Model
  - Structure Charts
  - Pseudocode
  - Temporal Logic
  - Natural Language

Structured Analysis and Structured Design (SA/SD)

- The purpose of the environmental model is to model the system at high level of abstraction.
- Behavioral model embodies the design aspect of SA/SD as a series of data flow diagrams, control flow diagrams, entity-relationship diagrams, process and control specifications, state transition diagrams and data dictionary.
- At the lowest level of abstraction, implementation models like pseudocode come into picture.

The Requirement Document

SRS (Software Requirement Specification) can be viewed as a binding contract among the designers, developers, testers and customers.

Recommended table of contents for the SRS as per IEEE Std 830-1998:

1. Introduction
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Definitions and Acronyms
    - 1.4 References
    - 1.5 Overview
2. Overall Description
    - 2.1 Product Perspective
    - 2.2 Product Functions
    - 2.3 User Characteristics
    - 2.4 Constraints
    - 2.5 Assumptions and Dependencies
3. Specific Requirements

Appendices

Index

Requirement Validation

The purpose of requirement validation is to ask this question

– “Am I building the right software ?”

Requirement Validation involves checking the followings:

- Validity - Does the system provide the functions that best support the customer’s needs?
- Consistency -Are there any requirements conflicts?
- Completeness -Are all functions required by the customer included?
- Realism -Can the requirements be implemented given available budget and technology?
- Verifiability -Can the requirements be checked?

Requirement Validation

Many ways of checking the software requirements specification for conformance to IEEE standard's best practices:

1. Automated consistency analysis
2. Checking the consistency of a structured requirements description.
3. Comparing the requirements to those for a similar system
4. Developing tests for requirements to check testability.
5. Prototyping.
6. Requirements reviews.
7. Systematic manual analysis of the requirements.
8. Test-case generation.
9. Using an executable model of the system to check requirements.

Please go through the Case study at the end of the Chapter as Appendix

BITS ZG553 - Real Time System – Lecture 15

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Real Time Performance Analysis

BITS Pilani
Pilani Campus

Introduction

- Performance analysis activities occurs in all phases of development.
- In testing phase, it is practical to measure performance in a real working environment.
- But there are needs to analyze the performance at design phase itself.
- Such performance measures are usually predicted and estimated.
- It requires theoretical performance analysis techniques.

Execution Time Estimation

- Estimation of the execution time e_i is the most important task in modeling.
- The most accurate method is using 'Logic Analyzer'. But it can be done, when physical system is ready i.e. at the time of testing and characterization.
- When the whole system is not ready, the execution time can be estimated by examining the compiler output, either manually or through a tool.
- Another possible method is to use system timer
  - Note the system time before start of the execution.
  - Note the system time after end of the execution.
  - The difference in the time is the execution time.
  - Do this exercise for all possible code execution path and take the worst case time.

Analysis of round-robin systems

Assume

Number of tasks in the system = n

Time slice (time quantum) = q

Maximum execution time of each task $T_i = c_i$

Maximum execution time of all tasks $c = max(c_1, c_2, \ldots, c_n)$

Then amount of time each task has to wait in one round = $(n-1)q$

Maximum number of time slices each task require to complete = $\lceil c/q \rceil$

Then amount of time (worst case) each task has to wait in total = $(n-1)q\lceil c/q \rceil$

The worst case time for a task to complete from readiness to completion

$$T = c + (n-1)q\lceil c/q \rceil$$

If there is a context-switching overhead of 'o', then for switching 'n' tasks in a cycle, the context switching overhead will be '$n.o$'.

In that case, the worst case time for a task to complete from readiness to completion

$$T = c + [(n-1)q + n.o]\lceil c/q \rceil$$

Round Robin System - Example

In a round-robin system, there are 6 equally important tasks, each with a maximum execution time of 600 ms and the time quantum is 40 ms, and every context switch costs 2 ms, then find out the response time of each task.

Solution:

Here, n = 6, q = 40 ms, c = 600 ms, o = 2 ms.

So,

$$
\begin{aligned}
T &= c + [(n-1)q + n.o]\lceil c/q \rceil \\
&= 600ms + [(6-1)\times 40 + 6\times 2]\lceil 600/40 \rceil ms \\
&= 600ms + (200+12)\times 15ms \\
&= (600 + 212\times 15)ms \\
&= (600 + 3180)ms \\
&= 3780ms
\end{aligned}
$$

Fixed Priority System

From time-demand analysis, the worst case response time is given by

$$t = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, 0 < t \leq p_i$$

But due to the ceiling function, this equation can't be solved in a straight forward method. So let us do it in iterative method. We can rewrite this equation as,

$$t_{n+1} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t_n}{p_k} \right\rceil e_k, 0 < t \leq p_i$$

where 'n' is the number of iteration.

Start the recursion with $t_0 = e_i$. The recursion stops, when $t_{n+1} = t_n$

If the recursion doesn't converge, then the processor is overloaded.

Fixed Priority System - Example

Find out the worst case response times of the tasks T1 = (9, 3), T2= (12, 4) and T3 = (18, 2), when they are scheduled rate-monotonically.

Solution:

As per RM algorithm, the priorities of the tasks are: T1 > T2 > T3.

First check the total utilization.

Total utilization U = 3/9 + 4/12 + 2/18 = 0.72 <1.

Hence processor is not overloaded.

Since T1 is the highest priority task, its response time is equal to its execution time i.e. 3.

Fixed Priority System - Example

For T2,

$$t_{n+1} = 4 + \left\lceil \frac{t_n}{9} \right\rceil \times 3, 0 < t_n \leq 12$$

$$t_0 = 4 : t_1 = 4 + \left\lceil \frac{4}{9} \right\rceil \times 3 = 4 + 3 = 7$$

$$t_1 = 7 : t_2 = 4 + \left\lceil \frac{7}{9} \right\rceil \times 3 = 4 + 3 = 7$$

So for T2, worst case response time is 7.

For T3,

$$t_{n+1} = 2 + \left\lceil \frac{t_n}{9} \right\rceil \times 3 + \left\lceil \frac{t_n}{12} \right\rceil \times 4, 0 < t_n \leq 18$$

$$t_0 = 2 : t_1 = 2 + \left\lceil \frac{2}{9} \right\rceil \times 3 + \left\lceil \frac{2}{12} \right\rceil \times 4 = 2 + 3 + 4 = 9$$

$$t_1 = 9 : t_2 = 2 + \left\lceil \frac{9}{9} \right\rceil \times 3 + \left\lceil \frac{9}{12} \right\rceil \times 4 = 2 + 3 + 4 = 9$$

So for T3, worst case response time is 9.

Analysis of Non-periodic Systems

- A real-time system having one or more aperiodic or sporadic job can be modeled (as covered in the previous lectures)
- But practically, they can be modeled rate monotonically as having periods equal to worst case inter-arrival times.
- Such rough approximation leads to unacceptably high utilization.
- So some heuristic approach should be used in stead.
- Queuing theory could also be helpful.

Sources of indeterministic behaviour

- Interrupt Latency
- DMA
- Caches
- Instruction Pipelines
- Floating point instructions

All of the above are to improve the CPU performance, but they destroys the determinism. This leads to go for a probabilistic performance model.

Single Server Queue Model

- Known as M/M/1 Queue
  - First M: Probability distribution of the time arrival of consumers : Here interrupt requests
  - Second M: Probability distribution of the time needed to service each customer: Here interrupt servicing
  - Third 1: Number of servers
  - The probability distribution of arrival is Poisson Distribution (inter-arrival times are exponentially distributed)
  - The service or processing time is also exponentially distributed.

Single Server Queue Model

Let

1/λ = Mean inter-arrival time of the customers (interrupt requests)

1/μ = Mean service (or interrupt processing) time

1/λ > 1/μ

N = Number of customers (interrupt requests) in the queue

Then, the expected number (mean) of customers (interrupt requests or jobs) in the queue,

$$\overline{N} = \frac{\rho}{1-\rho}, \text{ where } \rho = \frac{\lambda}{\mu}$$

The corresponding variance,

$$\sigma_N^2 = \frac{\rho}{(1-\rho)^2}$$

Single Server Queue Model

The mean time a customer (job) spends in the system,

$$\overline{T} = \frac{1/\mu}{1-\rho}$$

If *Y* is a random variable for the time spent in the system, which has an exponential probability distribution,

$$s(y) = \mu(1-\rho)e^{-\mu(1-\rho)y}$$

Probability that at least k customers (jobs) are in the queue simultaneously,

$$\Pr[\geq k] = \rho^k$$

When there are more than one customer waiting in the system, it is **a time-overloaded condition**. From this equation, it is obvious that the probability of exceeding a certain number of customers (jobs) in the system decreases geometrically (since ρ≤1). So it is pointless to determine the probability when there are more then two customer in the systems, since it will be negligible. Hence for determining the probability of time overloading, it is sufficient to consider the probability of two customers (jobs) in the system i.e. **Pr[≥2]**.

Example 1

In an interrupt driven system, the mean processing time of the interrupt is 5 ms. What should be the mean inter-arrival time of the interrupts, so that there is a 98% confident that CPU will not be overloaded?

Solution:

Mean interrupt processing time, 1/μ = 5 ms.

Probability of overload = 100% - 98% = 2%

$$\Rightarrow \Pr[\geq 2] = \rho^2 \leq 0.02$$

$$\Rightarrow \rho \leq \sqrt{0.02} = 0.141$$

$$\Rightarrow \frac{\lambda}{\mu} \leq 0.141$$

$$\Rightarrow \frac{\mu}{\lambda} \geq \frac{1}{0.141}$$

$$\Rightarrow \frac{1}{\lambda} \geq \frac{1}{0.141} \times \frac{1}{\mu} = \frac{5}{0.141} ms = 35.46 ms$$

Hence mean interrupt inter-arrival time should be more than 35.46 ms.

Example 2

In an interrupt driven system, the mean inter-arrival time of the interrupt is 10 ms. What should be the mean processing time of each interrupt, so that there is a 98% confident that CPU will not be overloaded?

Solution:

Mean inter-arrival time, 1/λ = 10 ms.

Probability of overload = 100% - 98% = 2%

$$\Rightarrow \Pr[\geq 2] = \rho^2 \leq 0.02$$

$$\Rightarrow \rho \leq \sqrt{0.02} = 0.141$$

$$\Rightarrow \frac{\lambda}{\mu} \leq 0.141$$

$$\Rightarrow \frac{1}{\mu} \leq \frac{0.141}{\lambda} ms = 0.141 \times 10 ms = 1.41 ms$$

Hence mean interrupt processing time should be less than 1.41 ms.

Example 3

A producer task is known to process data at rate that is exponentially distributed with average service time of 3 ms per task. What is the maximum allowable average data rate if the probability of collision is to be no more than 1%. Assume that the data arrive at intervals that are exponentially distributed.

Solution:

Mean task processing time, 1/μ = 3 ms.

Probability of collision(overload) = 1%

$$\Rightarrow \Pr[\geq 2] = \rho^2 \leq 0.01$$

$$\Rightarrow \rho \leq \sqrt{0.01} = 0.1$$

$$\Rightarrow \frac{\lambda}{\mu} \leq 0.1$$

$$\Rightarrow \frac{\mu}{\lambda} \geq \frac{1}{0.1}$$

$$\Rightarrow \frac{1}{\lambda} \geq \frac{1}{0.1} \times \frac{1}{\mu} = \frac{3}{0.1} ms = 30ms$$

Hence maximum allowable data rate = 1 / (30 ms) = 1000 / 30 per sec = 33.33 per sec.

Example 4

In an interrupt driven system, the mean inter-arrival time of the interrupt is 5 ms and mean interrupt processing time is 3 ms. What should be the mean response time of each interrupt ? What should be the mean and variance of the buffer required to store the incoming interrupt requests ?

Solution:

Mean inter-arrival time, 1/λ = 5 ms.

Mean interrupt processing time, 1/μ = 3 ms.

Average response time,

$$\overline{T} = \frac{1/\mu}{1-\rho} = \frac{1/\mu}{1-\lambda/\mu} = \frac{3}{1-3/5} ms = 3/0.4ms = 7.5ms$$

Average length of the buffer required to store the interrupt requests,

$$\overline{N} = \frac{\rho}{1-\rho} = \frac{\lambda/\mu}{1-\lambda/\mu} = \frac{3/5}{1-3/5} = 0.6/0.4 = 1.5$$

Variance of the buffer length,

$$\sigma_N = \sqrt{\frac{\rho}{(1-\rho)^2}} = \sqrt{\frac{\lambda/\mu}{(1-\lambda/\mu)^2}} = \sqrt{\frac{3/5}{(1-3/5)^2}} = \sqrt{0.6/(0.4)^2} = \sqrt{0.6/0.16} = \sqrt{3.75} = 1.936$$

Analysis of Memory Requirements

A typical memory map constitutes

- Program
- Stack
- Data
- Parameters

Total Memory Utilization,

$$M_T = M_{PG}.P_{PG} + M_{ST}.P_{ST} + M_{DT}.P_{DT} + M_{PM}.P_{PM}$$

$$M_A = \frac{U_A}{T_A}, A \in \{T, PG, ST, DT, PM\}$$

Here M_{PG}, M_{ST}, M_{DT}, M_{PM} represents **memory utilization** of program memory, stack memory, data memory and parameter memory respectively. Similarly P_{PG}, $P_{ST,}$ P_{DT}, P_{PM} represents the **fraction of the total memory used** for these types of memories respectively.

Example - Memory Requirements

A system has 64 M bytes of program memory, 16 M bytes of data memory and 8 M bytes of stack area. A program requires 75% of program memory, 25% of data memory and 50% of stack area, which are the worst case scenario. The system doesn't have separate parameter memory area. Calculate the memory utilization of the program.

Solution:

Total Memory = 64 + 16 + 8 M bytes = 88 M bytes

Total memory utilization,

$$M_T = 0.75 \times \frac{64}{88} + 0.25 \times \frac{16}{88} + 0.50 \times \frac{8}{88}$$
$$= 0.5454 + 0.0454 + 0.0454 = 63.63\%$$

BITS ZG553 - Real Time System – Lecture 15

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Verification and Validation Techniques

BITS Pilani
Pilani Campus

The role of testing

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
- A good test case is one that has a high probability of finding an error.
- A successful test is one that uncovers an error.
- The purpose of testing is also to address various aspects of qualities like
  - Errors
  - Performance
  - Reliability etc

Testing real-time software

Testing principles

- All tests should be traceable to requirements
- Tests should be planned long before testing begins
- Testing should begin “in the small” and progress toward testing “in the large”
- Test to also improve quality, not just demonstrate correctness

Quality Assurance Activity Sequence

Code Review

- The purpose of code review is to
  - Find bugs
  - Suggest code restructuring for better maintainability, readability etc
  - Improve performance
  - ….
- It is performed before Integration / Testing
- More eyes may catch more issues
- Input to Code Review
  - Coding Guidelines
  - Willingness to participate in code review
  - Incorporating time for code review in the plan

Unit level testing

- Several methods can be used to test individual modules or units
- Include black and white box techniques
- Black box techniques include
  - exhaustive testing
  - boundary value testing (boundary value analysis)
  - random test generation
  - worst case testing
- White box techniques include
  - judicious test selection
  - formal program proving
  - code inspections

System-level testing

- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Development and QA Activities should be in parallel

Continuous integration

BITS ZG553 - Real Time System – Lecture 16

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Uncertainties in Real-Time Systems

BITS Pilani
Pilani Campus

Introduction

- Focus of real-time system design is shifting from meeting mere performance goals to *designing for uncertainty*
- Uncertainty exists in embedded systems along 3 dimensions – Time, Space and Behaviour.
- Time:
  - Main of the challenge in design of embedded systems is unpredictability of response time.
  - RM Algorithm produces a deterministic schedule only when sporadic tasks, aperiodic tasks, mutual exclusion and resource contention are excluded.
- Space:
  - Space dimension deals with the limitation posed by physical resources like memory.
- Behaviour
  - Component behaviour, environmental behaviour (such as inputs, outputs), state, uncertainty of requirements, uncertainty in testing etc.

Introduction

Heisenberg Uncertainty Principle:

The precise position and momentum of a particle can't be known precisely; trying to be more certain about one comes at the expense of increased uncertainty of others.

Similarly, in case of real-time systems, trying to bring more certainty in once dimension, bring uncertainty to other dimensions.

Sources of Uncertainties

Kind of Uncertainties	Signs	Possible Causes	Possible Solutions
System under control	Bizarre inputs; Bizarre outputs;	Nature of application; Faulty hardware	Use aggressive fault tolerant design
Operating environment	Bizarre inputs; Bizarre outputs;	SEUs, sand, salt, shock, etc.	Use aggressive fault tolerant design
Requirements	Sparse requirements; Numerous “TBDs”	Incomplete requirements; Inconsistent requirements	Goals based requirements analysis; formal consistency checking
Testing	System that passes testing fails in the field	Poor testing regimen; Incomplete coverage	Improve testing process
Input	Strange behavior; Explicating comments	Unstable input sources; Defective hardware	Kalman filters, data fusion, median filters or averaging, roll back and recovery blocks.
Output	Strange behavior; Explicating comments	Defective hardware; Corrupted inputs from SUC	Kalman filters, data fusion, median filters or averaging, roll back and recovery blocks.
State	Strange behavior; Explicating comments	Program code jumping; Pointer errors	Model checking; Software black boxes

Sources of Uncertainties

Kind of Uncertainties	Signs	Possible Causes	Possible Solutions
Timing and schedulability	Dubious constraints; Missed deadlines; Explicating comments	Delays as loops; Speculative generality; Failed off-the-shelf components	Use OS provided timer facilities; Model checking; Fault injection
Programming Language & Compiler	Explicating comments;	Compiler induced errors	Test the compiler; Improve coding techniques
Off –the-shelf components	Missed deadlines; Inexplicable failure;	Badly tested software or hardware; Falsely advertised claims	Fault injection; Software black boxes
Execution Time	Missed deadlines	Gradual build up of various truncation and round-off errors	Rejuvenation (stopping and restarting the system regularly – should be used cautiously

BITS ZG553 - Real Time System – Lecture 16

BITS Pilani
Pilani Campus

Brajabandhu Mishra

Design for Fault Tolerance

BITS Pilani
Pilani Campus

Faults and Their Sources

Fault is an erroneous state of software or hardware resulting from failures of its components

Fault Sources

- Mechanical --"wears out"
  - Deterioration: wear, fatigue, corrosion
  - Shock: fractures, overload, etc.
- Electronic Hardware --"bad fabrication; wears out"
  - Latent manufacturing defects
  - Operating environment: noise, heat, ESD, electro-migration
  - Design defects (Pentium F-DIV bug)
- Software --"bad design"
  - Design defects
  - "Code rot" --accumulated run-time faults
- System Misuse – "people"

Faults Classifications

- Failure:
  - Component does not provide service
- Fault:
  - A defect within a system
- Error:
  - A deviation from the required operation of the system or subsystem

Reliability

- A measure of whether a user can depend on the system.
  - The system “stands the test of time.”
  - There is an absence of known catastrophic errors; that is, errors that render the system useless.
  - The system recovers “gracefully” from errors.
  - The software is robust.
- For real-time systems, other informal characterizations of reliability might include
  - Downtime is below a certain threshold.
  - The accuracy of the system is within a certain tolerance.
  - Real-time performance requirements are met consistently.

Traditionally, MTFF (Mean Time To First Failure) and MTBF (Mean Time Between Failures) are used as a measure of Software Reliability.

Reliability – Statistical Perspective

Let S = The system

T = Time instance of fault

Then Reliability of S at any time instance t, is defined by

$R(t) = P(T > t)$ i.e. Probability of not having any fault till time t.

R(t) = 1 is the ideal case. Practically, r(t) is less than 1.

<u>Example:</u>

In the monitoring system of a nuclear power plant, the failure probability is no more than 10^{-9} per hour.

So, $R(t) = (1 - 10^{-9})^t$, t is in hours.

So, as t->∞, r(t) -> 0.

Component Reliability Model

Software Reliability – Statistical Perspective

Failure function tends to become a *bathtub curve* i.e. failures reduces initially at an exponential rate, but increases towards the end.

Source: P. A. Laplante, Real-Time Systems Design and Analysis, Wiley, 3rd edition

Software Reliability – Statistical Perspective

<u>Reasons for Bathtub curve for failures:</u>

- Hardware can wear and tear. So after some point of time, the number of failures increases.
- But software can't wear and tear. Then why bathtub curve for the software ?
  - During patch releases, making quick correction without designing them properly.
  - During patch releases, ignoring the effect of the new bug fix on other modules (thereby fixing one issue and introducing more new issues)
  - Inadequate testing of the software during patch releases.
  - Wear and tear of underlying hardware.

Serial and Parallel System Reliability

Serial Component Probability

Let there are n serially connected components, with reliability of $R_k(t)$ for kth component

Then total system reliability,

$$R(t) = 1 - Q(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$$

Parallel Component Probability

If there n parallel connected components , with reliability of $R_k(t)$ for kth component

Then total failure probability,

$$R(t) = 1 - Q(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$$

So total reliability,

$$R(t) = 1 - Q(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$$

So, reliability increases with parallel components

What is Fault Tolerance ?

Fault Tolerance in real-time system is the tendency to continue functioning in the presence of hardware or software failures.

Two Types:

- Spatial Fault Tolerance: Includes methods involving redundant hardware and/or software solutions
- Temporal Fault Tolerance: Involves miscellaneous techniques that allow for tolerating missed deadlines

Methods to increase fault tolerance

- Voting schemes
- Software Check Points
- Software Black Boxes
- Built-in-Test Software
- CRC check
- Hamming code error detection and correction
- Handling Spurious and Missed Interrupts

Voting Schemes

- Combine input from two or more process blocks (hardware or software) and combine in some manner
- Voting schemes use majority rule or average of block outputs to obtain final values

- *P1, P2 and P3 processors execute different versions of the code for the same application (usually developed by different teams).*
- *Voter compares the results and forward the majority vote of results (two out of three).*

Software Check Points

- Intermediate results are dumped at fixed locations for diagnostic purpose
- Can be used during verification or regular operation
- If used during testing only then this additional code is called a test probe

Recovery Blocks

- Fault tolerance is increased by using checkpoints with predetermined reset points in the software.
- These reset points mark recovery blocks
- At the end of each recovery block, the corresponding check point is tested.
- If the results are not reasonable, then the processing resumes with a prior recovery block.

N-Version Programming

- Used to increase software fault tolerance
- Generally redundant processors are used.
- Same software developed by different teams using different algorithms runs on these redundant processors.
- In case one such software locks-up, then the system can ignore this software and go ahead with the other software which is functioning correctly on different processors.
- This is called N-Version programming.
- Generally this method is used mission-critical systems like the space shuttle's general purpose computer.

Built-in-Test Software (BITS)

Can increase fault-tolerance in presence of hardware errors

- CPU Tests
- Memory Tests (ROM and RAM)
- Other Devices

BITS – CPU test

- Perform carefully constructed set of tests
- Must be performed with interrupts disabled
- Too slow to perform in foreground

BITS – Memory test

- All kind of memory can be corrupted via electrostatic discharge, power surging, vibration or other physical means
- Corruption of memory devices by randomly encountered charge particles is a particular problem in space.

ROM check – using CRC

- CRC – Cyclic Redundancy Code
- CRC value of each memory is stored and checked when data from the memory is used
- CRC Calculation
  - Treats memory as a long binary polynomial
  - Divide polynomial by generator polynomial
    - $X^{16}+X^{12}+X^{5}+1$ (CCITT)
    - $X^{16}+X^{15}+X^{2}+1$ (CRC-16)
  - Remainder is CRC

3-bits CRC Example

Generator polynomial: x^3+x+1

$= 1x^3+0x^2+1x+1$

So generator polynomial is represented as 1011

Let the input polynomial is 11010011101100.

Pad the input polynomial by 3 bits (3 bits CRC) : 11010011101100000

Divide 11010011101100000 by 1011:

Essentially x-or the 4 bits of input polynomial with 4 bits of generator polynomial starting from left bit. Then move the generator polynomial to the next 1 of the result.

3-bits CRC Example (contd.)

```
11010011101100 000 <--- input right padded by 3 bits
1011 <--- divisor
01100011101100 000 <--- result (note the first four bits are the XOR with the divisor beneath, the rest of the bits are unchanged)
 1011 <--- divisor ...
00111011101100 000
   1011
00010111101100 000
     1011
00000001101100 000 <--- note that the divisor moves over to align with the next 1 in the dividend (since quotient for that step was zero)
            1011      (in other words, it doesn't necessarily move one bit per iteration)
00000000110100 000
              1011
00000000011000 000
              1011
00000000001110 000
                1011
00000000000101 000
                  101 1
-----------------
00000000000000 100 <--- remainder (3 bits). Division algorithm stops here as quotient is equal to zero.

So, CRC is 100
```

RAM Check – using EDC

- CRC is not suitable for RAM, since RAM content is volatile
- A Hardware circuit implementing some Hamming code error detection and correction (EDC) is used
- During normal data access, the data must pass through the EDC hardware before into or out of the memory
- The hardware compares the data against the check bits and makes necessary corrections if necessary.

Spurious and Missed Interrupts

Spurious Interrupts

- Spurious (or *phantom*) Interrupts are the extraneous and unwanted interrupts.
- Caused by noisy hardware, power surges, electrostatic discharges etc.
- They can destroy algorithmic integrity, stack overflows, compromise the deadlines etc.
- Such interrupt can be tolerated by
  - Having redundant interrupt hardware
  - The device issuing the interrupt can issue a redundant check like
    - Using DMA to send a confirming flag.
    - Then the device receiving the interrupt can check if this flag upon receiving it.

Missed Interrupts

- Missed interrupts are more difficult to deal with.
- They can be tolerated by robust algorithms

Watchdog Timer

- *A **watchdog timer** is used to automatically detect software anomalies and reset the processor if any such anomaly occur.*
- It is a timing device such that it is set for a preset time interval and an event must occur during that interval else the device will generate the timeout signal on failure to get that event in the watched time interval
- Watchdog timer can be
  - A Hardware
  - A Software task
- A simple watchdog timer example:
  - Watchdog timer is based on a counter that counts down from some initial value to zero.
  - The embedded software selects the counter's initial value and periodically restarts it.
  - If the counter ever reaches zero before the software restarts it, the software is presumed to be malfunctioning and the processor's reset signal is asserted.
- Another Example:
  - Assume that we anticipate that a set of tasks must finish in 100 ms interval.
  - The watchdog timer is disabled and stopped by the program instruction in case the tasks finish within 100 ms interval.
  - In case task does not finish (not disabled by the program instruction), watchdog timer generates interrupts after 100 ms and executes a routine, which is programmed to run because there is failure of finishing the task in anticipated interval.