Unit 1

Introduction

- Real time system can be defined as those system to which the correctness of the system depends not only on the logical computation but also on the time at which the results are produced. E.g. Avionics, defense system, etc.
- Real time system can be divided into three components:-
  - The controlling system
  - The controlled system
  - And environment
- Example: Avionics Computer (controller), controls aircraft system (controlled) to maintain desirable flight (environment) characteristics.

Digital ~~computer~~ controller

- Many real time system are ~~invaded~~ embedded in sensors and actuators and function as digital controller.

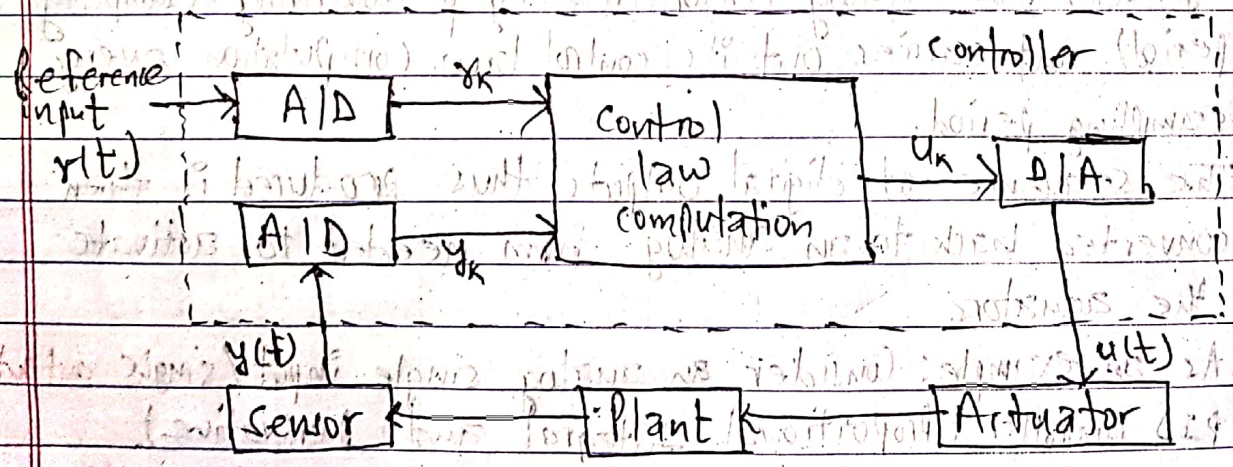


fig. A digital ~~computer~~ controller

- Plant refers to a controlled system. E.g.: engine, a brake, an aircraft, etc.
- The state of the plant is monitored by sensors and can be changed by actuators.
- The real time system (computing system) estimates from the sensor readings, the current state of the plant and computes a control output

based on the difference between the current state and the desired state (called reference input).

- This computation is called control law computation of the controller.
- The output generated activates the actuator which brings the plant closer to the desired state.

Sampled Data System

- A common approach to designing a digital controller is to start with an analog controller having the desired behavior and then transforming into a digital version. The resultant controller is the sampled data system.
- A sampled data system typically samples (i.e. reads and digitizes) the analog sensor reading periodically (sampling period) and carries out its control law computation every sampling period.
- The sequence of digital outputs thus produced is then converted back to an analog from needed to activate the actuator.
- As an example: Consider an analog single input/single output PID controller (Proportional Integral and Derivative).
- The analog sensor reading widely gives the measured state of the plant at time 't'.
- ~~$e(t) = r(t) - y(t)$ and the measured state $y(t)$ at time~~
- $e(t) = r(t) - y(t)$, i.e., the difference between the desired state $r(t)$ and the measured state $y(t)$ at time 't'.
- The output $u(t)$ of the controller consists of three terms:-
  - A term that is proportional to $e(t)$
  - A term that is proportional to the integral of $e(t)$
  - A term that is proportional to the derivative of $e(t)$

Scanned by CamScanner

High-level Controls

- Complex system controllers - organized hierarchically.
  Lowest level controllers - directly control physical plant
  Highest level controllers - interfare with operators
  Output of HLC is a reference input of one or more LLC

- E.g. of control hierarchy
- Patient Care System
  HLC (e.g. Expert System) interacts with operator (e.g. doctor, nurse)
  LLC - microprocessor based controllers that monitor and control patients BP, respiration, glucose, etc.
  E.g.: Air traffic and control light system (Book p.g.: 25)

Signal processing
- Short radio frequency pulse
- Only background noise - no object hit
- Stranger pulse - object hit
  (After hitting)

Tracking:
- Shorts out false returns
- Assigns each measured value to a trajectory
- Existing one - gives current position
  New:- New object

- Gating

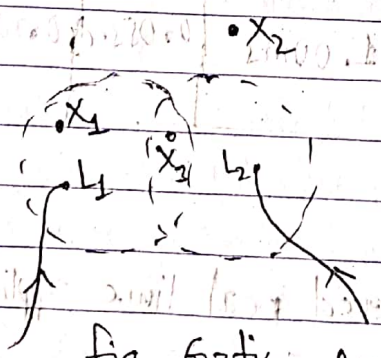


fig. Gating process

- Data Association - KNN (K-Nearest Neighbour)

Scanned by CamScanner

Other Real-Time Applications

- Real-time Database
- Multimedia Applications

The characteristics and requirements of two most common realtime applications are:-

(a) Real-time database
(b) Multimedia applications

(a) Real-time databases:

- The term real-time database system refers to a diverse spectrum of information systems, ranging from stock price quotation systems, to track records databases, to real-time file systems.
- Some of the examples are:

Application	Size	Average Resp. Time	Max Resp. Time	Abs. Cons.	Rel. Cons.	Permanence
Air traffic Control	20,000	0.50ms	5.00 ms	3.0sec	6.0sec	12 hours
Aircraft Mission	3,000	0.05ms	1.00ms	0.05sec	0.20sec	4 hour

(b) Multimedia applications:

- It is most frequently encountered real time applications multimedia.

Scanned by CamScanner

- A multimedia application may process, store, transmit, and display any number of video streams, audio streams, images, graphics, and text.

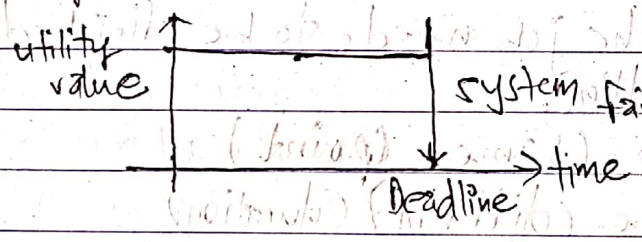
Unit:2

Types of RTS:

Depending upon the temporal behaviour of RTS, they can be characterized as hard, soft and firm RTS.

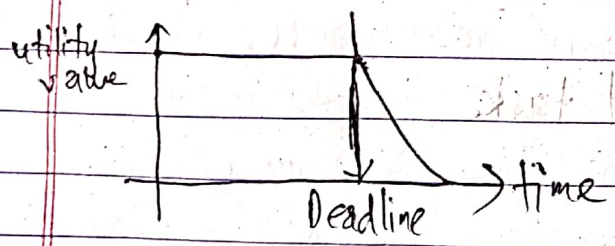
1) Hard Real Time System:

- A system crashes if the deadline is not met by the tasks.



E.g. An enboard computer of an aircraft

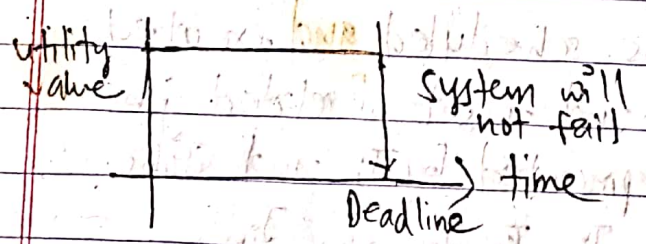
2) Soft RTS:



- Here utility value drops with time after deadline; performance decreases

E.g. Web Browsing (clicking of URL)

3) Firm RTS



- System doesn't fail but ignores the result in case of failure

E.g. video conferencing

Scanned by CamScanner

Date
Page

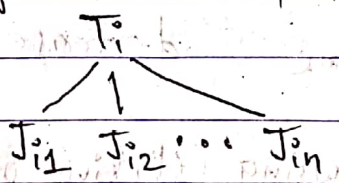
Key Terminologies:-

- Job : In RTS, the smallest unit of job
  E.g. : file retrieve
  In Gantt chart use J_n
- Task : E.g. Controller of digital
  : Collection of job that works for certain task
- Processor : resources that is used for job and task
  Eg to retrieve file we use memory which is resource
- Release Time : time after which the job is ready for execution.
- Deadline : time that the job needs to be finished within deadline.

Absolute ~~Relative~~ Deadline (same) (point)
Relative ~~Absolute~~ Deadline (different) (duration)

- Response time : The length of time from the release time of the job to the instant when it completes.

Collection of job is called task.



Job and Task:
Each unit of work/task is scheduled and executed by the system is called job and a set of related jobs is called a task. Task is represented by T_i and Jobs of task T_i are represented by $J_{i1}, J_{i2}, J_{i3}, \ldots, J_{in}$

Scanned by CamScanner

Periodic task : (P_i, e_i)
task period, i.e. the minimum time between job releases
worst case execution time

Resources are called processors. E.g. CPU, Memory, Data Link

Real Time Tasks

- Periodic:
  Periodic tasks repeat after a certain fixed time interval. It has fixed release time and hard (strict) deadline. E.g. Digital controller
- Sporadic:
  Sporadic tasks recurs at random instants. It has strict deadline. (E.g. Avionic machine)
- Aperiodic:
  Same as sporadic except that minimum separation between two instances can be 0. It has no fixed release time and deadline (soft). (E.g. window update)
  ↳ may or may not have deadline

r_{i1} : release time of first job of ith task
Period (P_i) : Period of ith task
Phase (ϕ_i) : release time of first job of T_i, $\phi_i = r_{i1}$

Release Time:
The release time of a job is the instant of time at which the job becomes available for execution. The job can be scheduled and executed at any time at or after its release time.
Release Time of Jobs ~~are~~ $J_{i1}, J_{i2}, \ldots, J_{in}$ are represented by $r_{i1}, r_{i2}, \ldots, r_{in}$

Deadline (Absolute Deadline): (d)
The deadline of a job is the instant of time by which its execution is required to be completed

Response time:
The length of time from the release time of the job to the instant when it completes.
Response time = Job completion time − Job release time

Scanned by CamScanner

Relative Deadline (D) : The maximum allowable response time of a job is called its relative deadline.

$D = d - r$

Timing Constraints

- A timing constraint is defined with respect to some event.
- An event
  - Can occur at an instant of time
  - May also have duration
  - Generated either by the system or its environment
  - There are two types of events:-
    - Stimulus Events
    - Response Events

Stimulus Event

- Generated by the environment
  - Act on the system
- Typically asynchronous in nature
  - Aperiodic
  - Can also be periodic

Example: • A user pressing a button on a telephone set
• Stimulus event acts on the telephone system
• Periodic sensing of temperature in a chemical plant is an example of stimulus event

Response Event

- Produced by the system:
  - In response to some stimulus events
- Example: • In a chemical plant as soon as the temperature exceeds 100°C, the system responds by switching off the heater.

Scanned by CamScanner

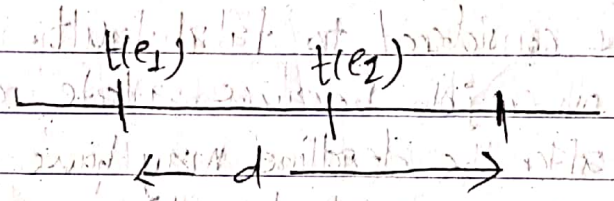
Date
Page

Classification of Timing Constraints

- A constraint imposed on the timing behaviour of a job is called timing constraint.
- Classification of the timing constraints in a system can help us to quickly identify these ~~problem~~ from a problem description.
- Timing constraints can broadly be classified into:
  - Performance constraints : imposed on the response of the system
  - Behavioural constraints : imposed on the stimuli generated by the environment
- Both performance and behavioural constraints can be classified into:
  - Delay constraints
  - Deadline constraints
  - Duration constraints

Delay constraints :

- Expresses minimum ~~delay~~ time delay allowed between the occurence of two arbitrary events e1 and e2
- $t(e2) - t(e1) \geq d$
- If e2 occurs earlier than d then a delay violation would occur.



Deadline Constraint:

- Expresses the maximum permissible separation between any two ~~abt~~ arbitrary events.
- $t(e2) - t(e1) \leq d$

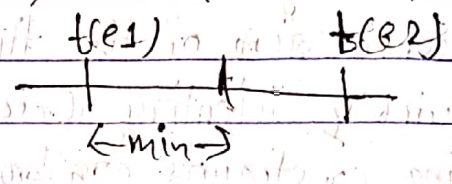
Duration Constraint

- A duration constraint on an event specifies the time period over which the event acts.
- A duration constraint can be ☐ minimum type ☐ Maximum type

Scanned by CamScanner

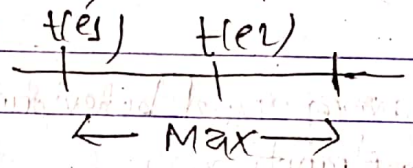
Minimum duration constraint:

- Once a duration event starts, it must not end before a certain minimum time.



Maximum duration constraint:

- Once a duration even starts, it must end before a certain maximum time.



Timing constraint can be broadly classified into:

Time constraint
- Hard timing constraint
- Soft timing constraint

Hard and Soft Timing Constraints:

A timing constraint or deadline is hard if the failure to meet it is considered to fatal fault. A hard deadline is imposed on a job because a late result produced by the job after the deadline may have disatenous consequences. E.g. missile launching, a late command to stop a train may cause collision

In contrast; the late completion of a job that has a soft deadline is undesirable, however a few misses of soft deadlines do not cause serious harm; only the system's overall performance becomes poorer and poorer.

E.g. late retrieval of file from server

Tardiness:

Tardiness = Completion time − Deadline

(A measure of how late a job completes with respect to deadline)

- Completion on or before deadline ⇒ Tardiness = 0
- Hard job → Tardiness = 0
- Soft Real time Job → Tardiness ≥ 0

But, when tardiness increases, performance decreases

Q. Consider a system which monitors and controls several furnaces. After it is initialized and starts execution (say at time 0), the system samples and reads each temperature sensor every 100msec and places the sampled readings in the memory. It also computes control law of each furnace every 100msec in order to process the temperature readings and determine flow rates of fuel, air and coolant. Suppose that the system begins ~~at~~ the first control law computation at time 20msec.

Ans: Control law is computed periodically. So, it can be stated in terms of release times of jobs $J_0, J_1, \ldots, J_k$

Release time of job $k = 20 + k \times 100$ msec, for $k = 0, 1, \ldots$

We say that jobs have no release time if all the jobs are released when the system begins execution.

Deadlines of subsequent jobs are 120ms, 220ms and so on.

Relative deadline of every job is 100msec.

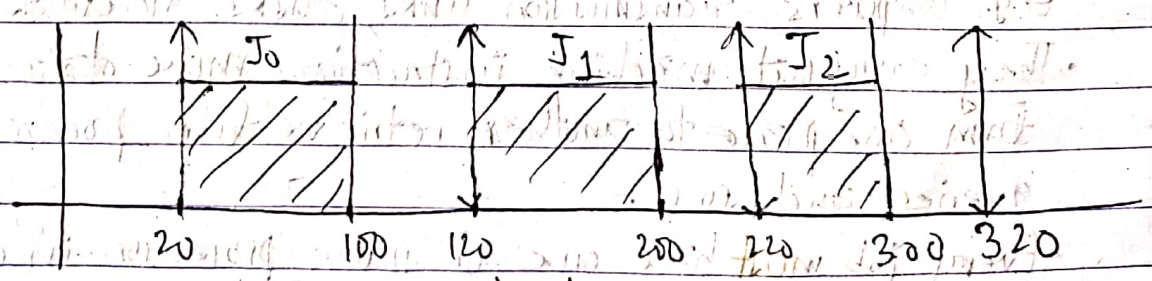


fig. Gantt Chart

Scanned by CamScanner

Date
Page

Temporal QOS Guarantees

- For a hard real-time job / Job with hard timing constraints
  - the user requires validation that the system always meets the timing constraint.
  - Guaranteed temporal QOS required.
- For soft real-time job / Job with soft timing constraints:
  - no validation required.
  - Best effort temporal QOS
  - user asks for the best QOS the system can provide, but allows the system to deliver quality below what is defined by the timing constraint.

Resource:

A resource is any physical or virtual component of limited availability within a computer system. Every device connected to a computer system is a resource. Virtual system resources include files (concrete file handles), network connections (concretely network sockets), and memory areas.

Examples: CPU, RAM, Hard disk, etc.

Types:

All the system resources can be divided into two major types: processors and resources.

* Processors:
  - Active resources
  - Eg. computers, transmission links, disks and databases
  - They carry out machine instructions, move data from one place to another, retrieve files, process queries, and so on.
  - Every job must have one or more processors in order

Scanned by CamScanner

to execute and make progress towards completion.

- Processors are denoted by the letter P.
- Main attribute is speed.
- Amount of time required for completion is a function of speed.

* Resource:

- Passive resources
- Examples: memory, sequence numbers, mutexes, and database locks
- It is not associated with speed.
- Jobs need or may need some resources in addition to processor to complete. or make progress.

E.g. Sliding Window Scheme
Job → Transmission of data
Processor → Data link
Resource → sequence number

Temporal Parameters of Real-time Work load

- For every job J_i
  - → Release time $= r_i$
  - → Absolute deadline $= d_i$
  - → Relative deadline $= D_i$
  - → Feasible interval $= (r_i, d_i]$
  - → Execution time $= e_i$
  - → Earliest Release Time $= r_i^-$
  - → Latest Release Time $= r_i^+$
  - → Release Time Jitter $= [r_i^-, r_i^+]$

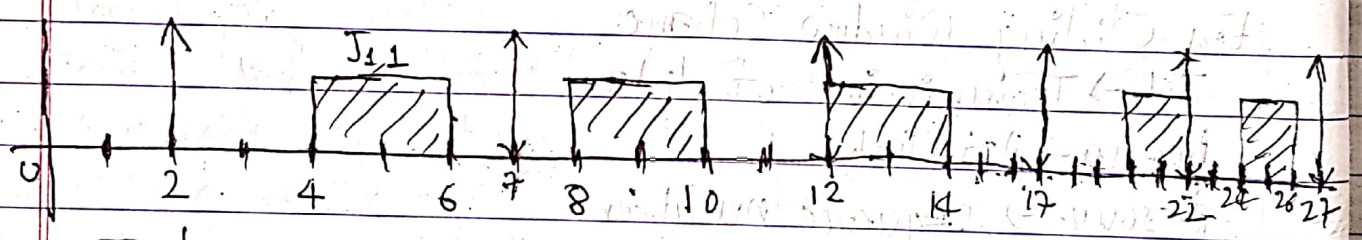
Scanned by CamScanner

Execution Time:

- Amount of time required to complete the execution of J_i when it executes alone and has all the resources it requires.
- Depends on: The complexity of job
  : The speed of processor
- Example:
  MPEG frame transmission
- Actual execution time not known until it is completed.
  - $[e_i^-, e_i^+]$

Periodic Task Model

E.g. A periodic task T_i with $r_i = 2$, $p_i = 5$, $e_i = 2$ and $D_i = 5$



T_i has jobs $J_{i,1}, J_{i,2}, J_{i,3}, \ldots, J_{i,k}$

$\phi_1 = 2$ Phase (ϕ_i) ⇒ Release time ($r_{i,1}$) of the 1st job ($J_{i,1}$)

Tasks in phase

Tasks out of phase

Hyperperiod (H) = LCM of p_i

Jobs in each hyperperiod $= \sum_{i=1}^{n} H/p_i$

Task utilization $(u_i) = \frac{e_i}{p_i}$

Total (System Utilization) $= \sum_{i=1}^{n} u_i$

Scanned by CamScanner

Date
Page

Assume, $D_i = P_i$ for all jobs if D_i is not given

A periodic task T_3 with:

$P_1 = 3$ $\quad P_2 = 4$ $\quad P_3 = 10$

$e_1 = 1$ $\quad e_2 = 1$ $\quad e_3 = 3$

Hyperperiod (H) = LCM of P_1, P_2, P_3

$= LCM(3, 4, 10)$

$= 60$

No. of Jobs in each hyperperiod (max) $= \sum_{i=1}^{n} \frac{H}{P_i}$

$= \frac{60}{3} + \frac{60}{4} + \frac{60}{10}$

$= 20 + 15 + 6$

$= 41$

Task utilization $(u_i) = \frac{e_i}{P_i}$

1st task $= 1/3 = 0.33$, 2nd task $= \frac{1}{4} = 0.25$, 3rd task $= \frac{3}{10} = 0.3$

Total (system) utilization $= \sum_{i=1}^{n} u_i = 0.33 + 0.25 + 0.3 = 0.88 = 88\%$

Task Dependencies

- Jobs are under precedence constraint if they are constrained to execute in some order.

Otherwise independent

$<$ ⇒ Precedence relation

Precedence Graph

$J_i < J_k$ ⇒ J_i is (immediate) predecessor of J_k and J_k is (immediate) successor of J_i

A directed graph, $G = (J, <)$

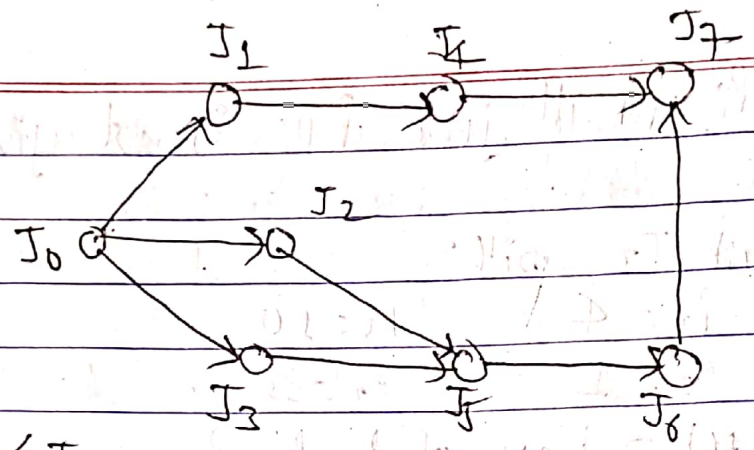
vertex → Jobs

Directed edge from J_i to J_k exists if $J_i < J_k$

Scanned by CamScanner

Date
Page

E.g.



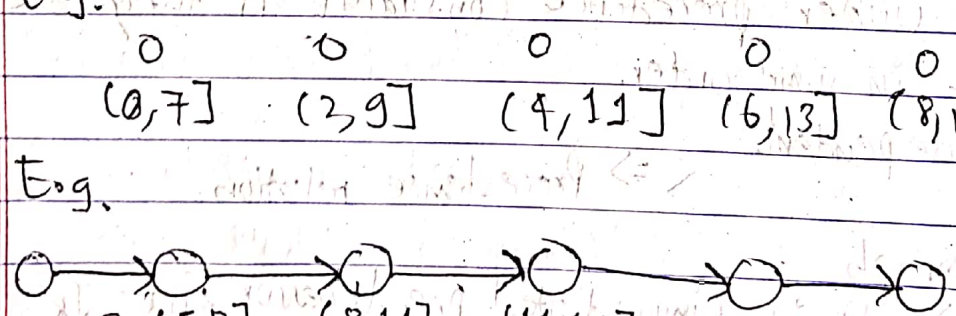
$J_0 < J_1$
$J_0 < J_2$
$J_0 < J_3$

~~We prefer task graph to precedence graph because~~
~~- shows order of feasible interval~~

We prefer task graph to precedence graph because
- shows order of execution because of feasible interval

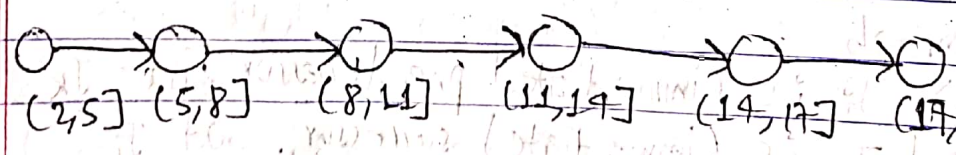
Task Graph
- Extended form of precedence graph
- Vertices represent Jobs
- Edges represent dependencies
- No. in bracket represent feasible interval

E.g.



Phase = 0
Period = 2
R. Deadline = 7

E.g.



Phase = 2
Period = 3
R. Deadline = 3

Scanned by CamScanner

Date
Page

Data Dependency:

- Task graph may contain different types of edges to represent different dependencies (Precedence graph fails to capture data dependency).
- An edge from vertex J_i to J_k represents data dependency if

(i) Job J_k consumes data generated by J_i or

(ii) Job J_i sends a message to Job J_k

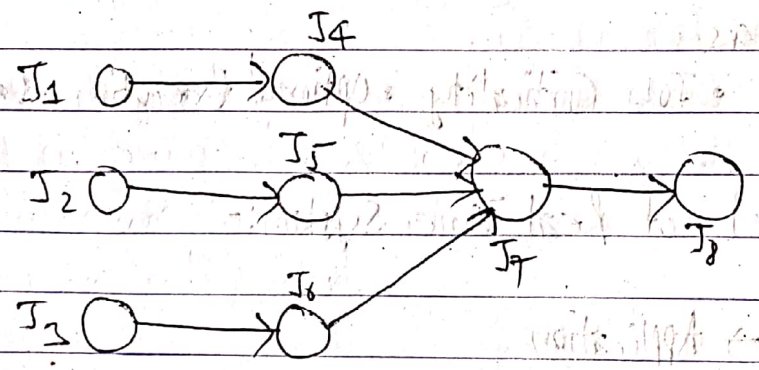
- A parameter of an edge from J_i to J_k is the volume of data from J_i to J_k.

Other Data Dependencies:

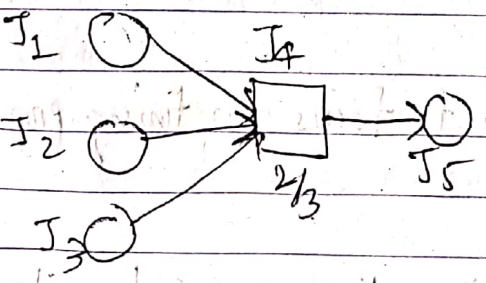
(i) Temporal Dependency:

- Temporal distance between 2 jobs = difference in completion time of two jobs
- Jobs have temporal distance constraint if their TD must be less than or equal to ($\leq$) some finite value.
- Represented in graph by temporal dependency edge

(ii) AND / OR Precedence Constraint:



AND jobs represented by hollow circle



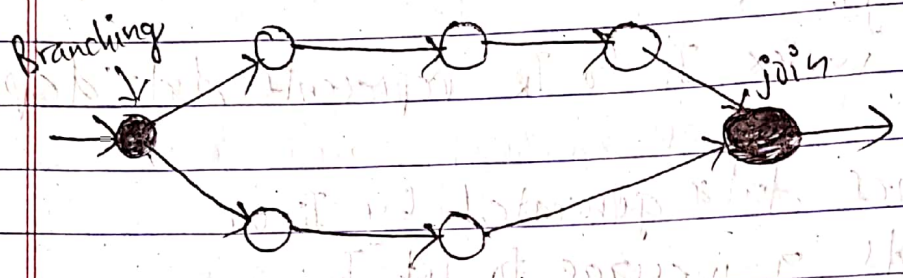
OR jobs represented by hollow squares

Job parameters:-
In-type
Out-type

Scanned by CamScanner

(iii) Conditional Branches:

- Branch Job – one of many immediate successor is allowed to be executed.

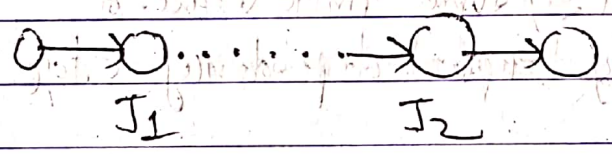


Conditional block

- Branch and Join jobs are shown by filled circle

(iv) Pipeline Relationship

- Represent producer-consumer relationship
- Pipeline edge is indicated by a dotted edge



Functional Parameters

- Job Preemptivity • Job Criticality • Optional Execution • Laxity Type

Reference Model of Real Time System:-

- Workload Model → Application
- → Resource Model → Resource
- → Algorithms

- Models describe the system to focus on timing properties and resource requirements.
- Composed of 3 components

(i) A workload model: describes the supported system.

Scanned by CamScanner

It has temporal parameters, precedence constraints and dependencies and functional parameters.

(ii) Resource Model: Describes the system resources available to an application (homogeneous / hetero)

(iii) Algorithms :- defines how the application system uses resources.

Tasks and Jobs

- Task (T_i): set of ~~jobs~~ related jobs jointly provide a function
- Job (J_{ij}): Unit of work, scheduled and executed by system.

A job is characterized by the following parameters

- Temporal parameters: timing constraints and behaviours
- Functional Parameters: Intrinsic properties of job
- Resource Parameters: Resource requirements
- Interconnection parameters: how it depends on other job and how other jobs depend on it.

Periodic Task Model:

- It is a well known deterministic workload model.
- Jobs repeated at a regular intervals called period.
- Task T_i is a series of periodic Jobs J_{ij}.

E.g. $T_1 = (3, 4)$

period ↑ execution time

$p_i = p_i, \phi_i = 0$

J_{11}	J_{12}	J_{13}	J_{14}	
↓	↓		↓	
0	0+3	6	9	release time

- p_i: Period: inter-release time of jobs
- e_i: execution time: maximum execution time of all jobs in it.
- r_{ij}: release time of jth job of ith task
- ϕ_i: phase: release time of first job of ith task.
- H (Hyperperiod): The LCM of p_i for $i = 1, 2, \ldots, n$

$H = LCM(p_i) \ \forall i$

Scanned by CamScanner

Utilization of Task(U_i): is equal to the fraction of a time truely periodic task with period P_i and execution time e_i keeps processor busy, i.e.

$$U_i = \frac{e_i}{P_i}$$

total utilization of system $= U = \sum_{i=1}^{n} U_i$

Functional Parameters

- Preemptivity:

The scheduler suspends the execution of a less urgent job and give the processor to a more urgent job. This interruption of job execution is called preemption.

A job is preemptable if its execution can be suspended at any time to allow the execution of other jobs and can later be resumed from the point of suspension.

A job is non-preemptable if it must be executed from start to completion without interruption.

Example: Computation jobs that execute on CPUs are examples of preemptable jobs.

Transmission of data frame in a token ring (or bus) is an example of non-preemptable jobs. Mutually exclusive variables are also examples of non-preemptable jobs.

Context switch:

The process of storing the state of the preempted job at the time of preemption so that it can ~~resume~~ be resumed, restored and execution from the same point later.

Scanned by CamScanner

priority or weight

- Criticality:

Jobs are not equally important. The importance (or criticality) of a job is a positive number that indicates how critical a job is with respect to other jobs. The more important the job, the higher the priority. [If not specified, 0 represents the highest priority.]

For example, in a flight control and management system the job that controls the flight of the aircraft is more critical than the navigation job that determines the current position relative to the chosen course. The navigation job is more critical than the job that adjusts the course and cruise speed in order to minimize fuel consumption. The cabin air flow and temperature control jobs are more critical than the job that runs the in-flight movies, and so on.

- Optional Execution:

An application may be structured in a way that some portions are optional while the rest is mandatory. Delay in completion or skipping of an optional job may degrade degree of performance, but still the system functions are satisfactory. Mandatory jobs must be executed to completion.

During overload; the mandatory portions must be run to completion while the optional portion may be skipped, completed or partially executed.

For example, in a collision avoidance system, we may consider the job that computes the correct evasive action and informs the operator of this action optional.

- Laxity Type and Laxity Function

The laxity type of a job indicates whether its timing constraints

are soft or hard. It can be used to indicate the relative importance of a timing constraint. It is supplemented with a utility function that gives usefulness of a result versus its degree of tardiness. The usefulness of the result becomes zero or negative as soon as the job is tardy in case of hard real time jobs. The usefulness of the result decreases gradually as the tardiness increases and may be acceptable or tolerable to some extent in case of soft real time jobs.

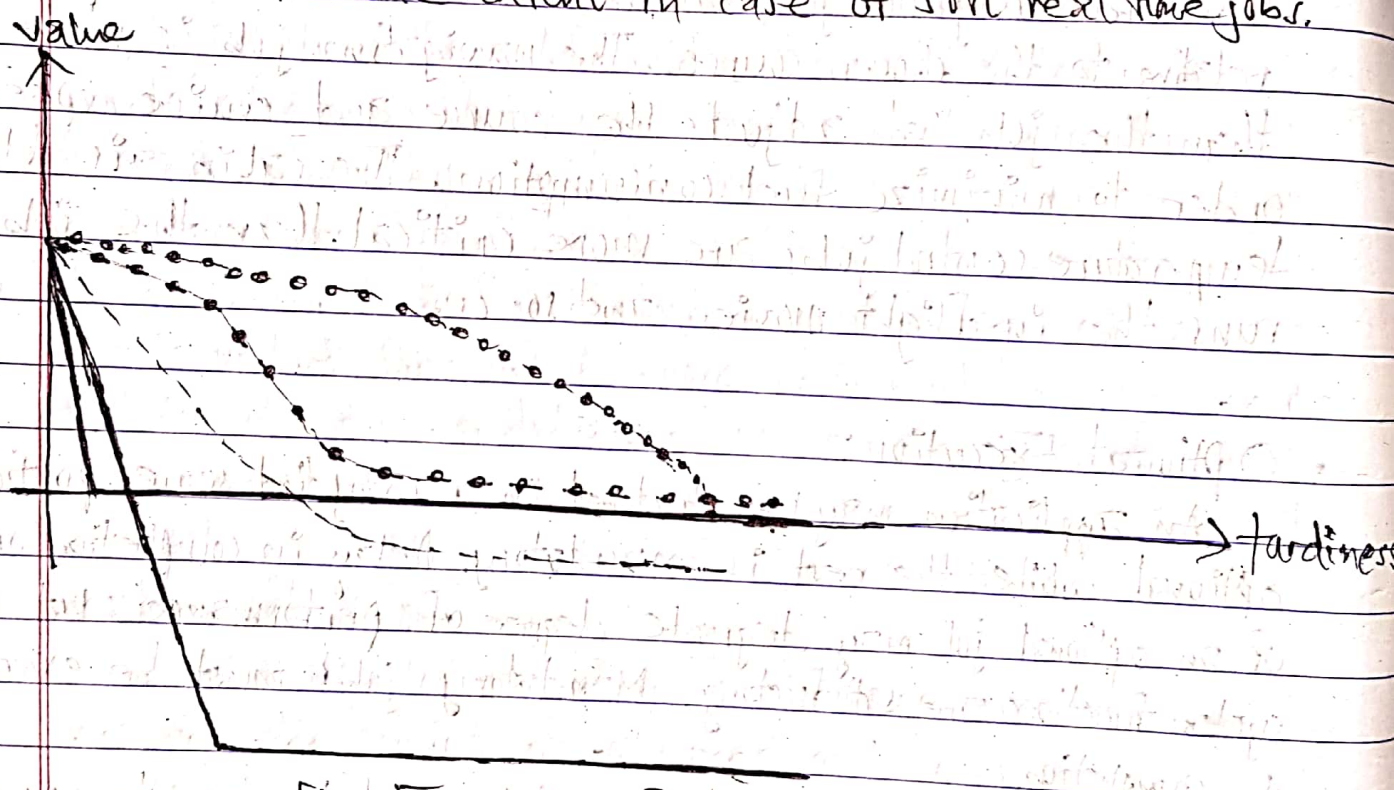


fig. Examples of usefulness functions

figure shows several usefulness functions as examples. Solid step functions are usually associated with hard-time jobs. The usefulness of the result becomes zero or negative as soon as the tardiness of the job increases. In the latter case (-ve usefulness), it is better not to execute the job rather than to execute it and complete it late. E.g. the transmission and execution of a command to release a bomb

on a target.

The dashed and dotted lines two other usefulness functions. In particular, the dotted ones may be that of a point-of-sales transaction, for example, one that executes to check whether you have enough credit for the current purchase. The usefulness of the result decreases gradually. Eventually, you are likely to give up and walk away. At that point, the usefulness of the result becomes zero.

The dashed line shows a function that decreases faster and becomes negative. An example of jobs that have this kind of usefulness function is a stock price update transaction. It may be tolerable if the update completes slightly late and the price x written into the database is somewhat old. However, if the transaction completes so late that the current price differs significantly from x, the result x can be misleading. By that time, the usefulness of this update becomes negative.

Usefulness functions

- used to describe qualitatively the real-time performance objectives of the system.
- They can guide the choice and implementation of scheduling strategies.

Resource Parameters of jobs and Parameters of Resources

The resource parameters of each job give us the type of processor and the units of each resource type required by the job and the time intervals during its execution when the units are required.

These parameters provide the information that is needed to support resource management decisions.

Describes processors and resources independently of application.

Scanned by CamScanner

Resource Parameters include-

(i) Resource Graph

* vertex represents resource (both active and passive)
* vertex (parameters) attributes → resource type
  → number

* Edges → is-a-part-of
  → accessibility

(ii) Preemptivity of resources:

~~It describe~~ A resource is non-preemptible if each unit of the resource is constrained to be used serially. In other words, once a unit of a non-preemptable resource is allocated to a job, other jobs needing the unit must wait until the job completes its use.

If the jobs can use every unit of a resource in an interleaved fashion, the resource is preemptable. The lock on a data object in a database is an example of non-preemptable jobs resource

Resource Graph:

- describe the configuration of the resources ~~using a~~
- gives the amount of resources available to execute the application, attribute of resources and rules governing their usage.
- In a resource graph there is a vertex R_i for every processor or resource R_i in the system. The attributes of the vertex are the parameters of the resource. The resource type of the resource gives whether the resource is a processor or (passive) resource, and its number gives the number of available units.
- Edges in a resource graph represent the relationship among

resources.

- There are two types of edges:
  - An edge from vertex R_i to R_{jk} means R_k is a component of R_i. This edge is called is-a-part-of edge. Subgraph containing all the is-a-part-of edges is a major component which contains sub-components represented by vertices in the tree.
  - The edge that represents connectivity between components are called accessibility edges. E.g. if there is a connection between two CPUs in two computers, then each CPU is accessible from the other computer and there is an accessibility ~~graph~~ edge from each component to CPU on other computer.

    Each accessibility edge may have several parameters whose value help to complete the description of interconnection of the resources. E.g. cost of sending a unit of data from a job executing on P_i to a job executing on P_k can be represented by a parameter of an accessibility edge from P_i to P_k.

8. Scheduling Hierarchy

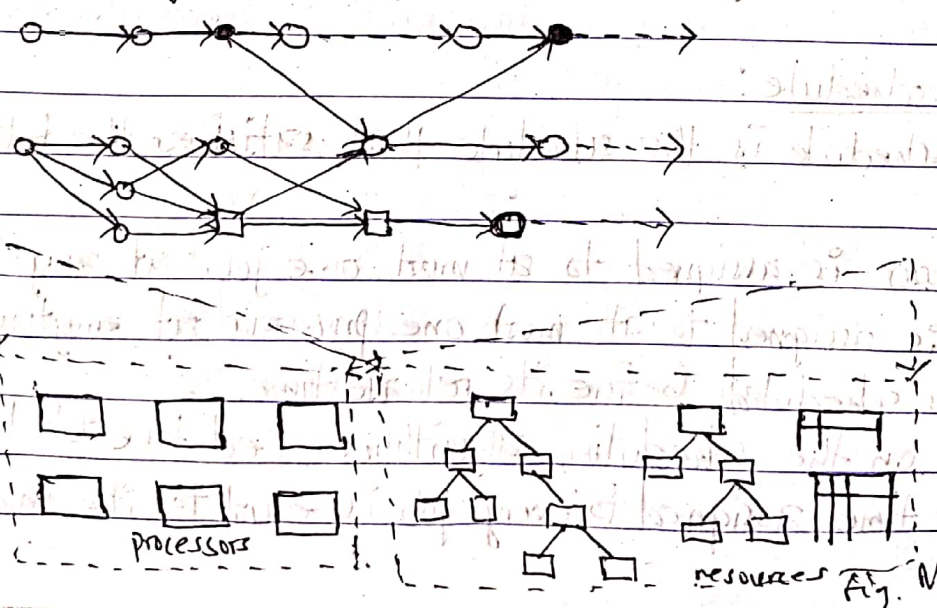


Fig. Model of a RTS

Scanned by CamScanner

- The figure 'model of real time system' shows the three elements of our model of real time systems.
- The application system is represented by
  → a task graph which gives the processor time and resource requirements of jobs, their timing constraints and dependencies.
  → A resource graph describing the resource available to execute the application system, their attributes and rules governing their use.
  → And between these graph are the scheduling and resource access-control algorithms used by the operating system.

Scheduler and Scheduler

- Jobs are scheduled and allocated resources according to a chosen set of scheduling algorithms and resource access-control protocols.
- The scheduler is a module that implements these algorithms.
- The scheduler assigns processors to jobs, or equivalently, assigns jobs to processors.
- A schedule is an assignment by the scheduler of all the jobs in the system on the available processors.

A valid schedule:

A valid schedule is the schedule that satisfies the following conditions:

(i) Every processor is assigned to at most one job at any time.
(ii) Every job is assigned to at most one processor at anytime.
(iii) No job is scheduled before its release time.
(iv) Depending on the scheduling algorithms used, the total amount of processor time assigned to every job is equal to its maximum

Scanned by CamScanner

or actual execution time.

(v) All the precedence and resource usage constraints are satisfied.

Feasibility:

- A valid schedule is a feasible schedule if every job completes by its deadline and in general meets its timing constraints.
- A set of jobs is schedulable according to a scheduling algorithm if when using the algorithm the scheduler always produces a feasible schedule.

Optimality:

- A hard real time scheduling algorithm is optimal if using the algorithm the scheduler always produces a feasible schedule if the given set of jobs has feasible schedules.
- If an optimal algorithm cannot find a feasible schedule, we can conclude that a given set of jobs cannot feasibly be scheduled by any algorithm.

Unit - 4

Approaches to Real-Time Scheduling

(1) Clock Driven Approach (Static Approach)

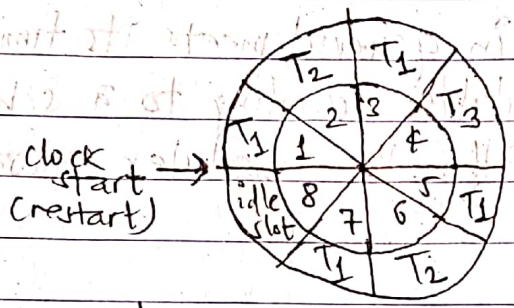
- Decisions on what job executes at what time are made at specific time instants (time driven)
  - Choosen apriori, before the system executes.
- Schedule computed offline
- Stored in a table for use at runtime.
- Scheduler schedules at regularly spaced time instants using a hardware timer (clock).
- Suitable for small and simple systems

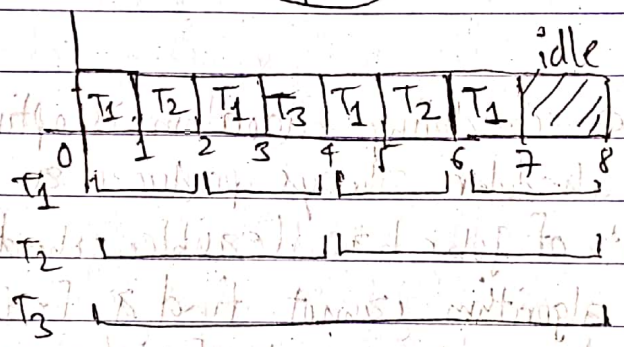
Scanned by CamScanner

Eg. Scheduling of 3 periodic tasks using CDSA where,

$T_i = (e_i, p_i)$

$T_1 = (1, 2)$ $T_2 = (1, 4)$ $T_3 (1, 8)$





Hyper period = 8

$T_1 = 8/2 = 4$

$T_2 = \frac{4}{2} = 2$

$T_1 = 8/8 = 1$.

2) Weighted Round Robin Approach

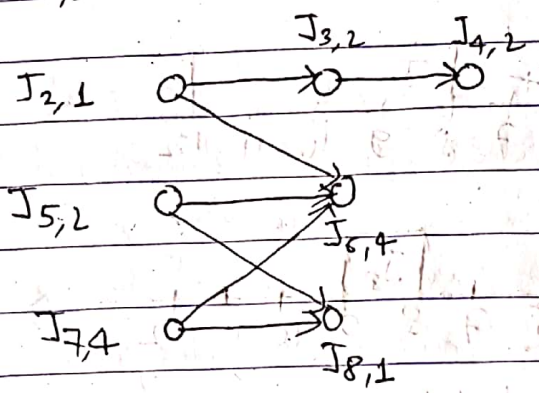
- Round Robin:
  - Jobs join a FIFO queue
  - Job at head of queue gets one time slice and executes
  - After execution, it is preempted and goes at the end of queue.
  - Job gets turn every n^{th} time slice.

- WRR:
  - Allocates different weights to different ready jobs
  - A job is weight 'w' gets 'w' time slices.
  - Round Trip Time (RTT) $= \sum_{i=1}^{n} w_i$
    (queue length)

Scanned by CamScanner

3) Priority Driven Approach:

E.g.



Highest priority = 1
lowest priority = 8

$J_{1,3}$
Job execution time

J_5 released at time 4
All other jobs released at time 0.
J_i has higher priority than J_k if $i<k$
Processors P_1 & P_2

(i) Preemptive scheduling
(ii) Non-Preemptive scheduling

- At time 0, jobs J_1, J_2 and J_7 are ready for execution. They are the only jobs in the common priority queue at this time. J_1 and J_2 have higher priorities than J_7 and hence are scheduled.
- At time 1, J_2 completes and hence, J_3 becomes ready. J_3 is placed in the priority queue ahead of J_7 and is scheduled on P_2, the processor freed by J_2.
- At time 3, both J_1 and J_3 complete. J_5 is still not released. J_4 and J_7 are scheduled.
- At time 4, J_5 is released. Now there are three ready jobs. J_7 has the lowest priority among them. Consequently, it is preempted. J_4 and J_5 have processors.
- At time 5, J_4 completes. J_7 resumes on processor P_1
- At time 6, J_5 completes. Because J_7 is not yet completed, both J_6 and J_8 are not ready for execution. Consequently, processor P_2 becomes idle.

Scanned by CamScanner

- J_7 finally completes at time 8. J_6 and J_8 can now be scheduled and they are,

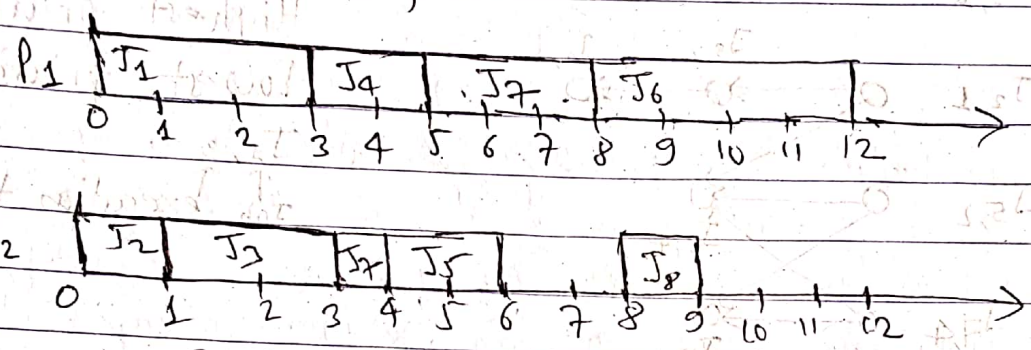


fig. Preemptive schedule

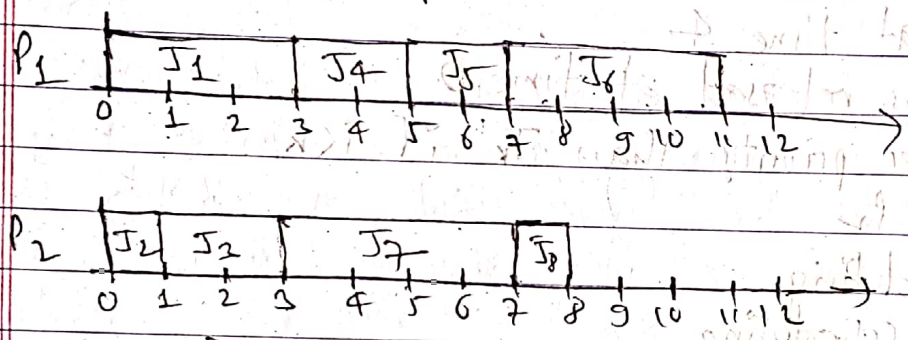


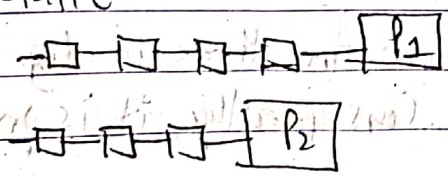
fig. Non-preemptive schedule

- In general, non-preemptive scheduling is not better than preemptive.

Priority scheduling algorithms are - event-driven
- other names: greedy scheduling, list scheduling and work-conserving scheduling

Dynamic v/s Static System

Static

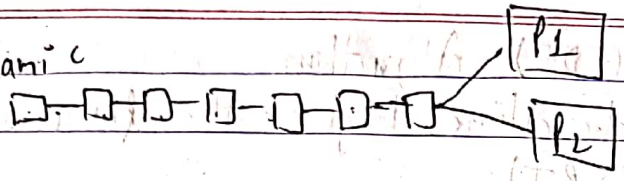


- Partition jobs in the system into subsystems and assign the subsystems statically to the processors.

Job migrates only if
- System is reconfigured
- Operation mode changes
- Some processor fails.

Eg. Put J_1, J_2, J_3, J_4 on P_1 and J_5, J_6, J_7 on P_2 satisfying the precedence constraints

Dynamic



- Ready jobs kept in priority queue common to all processors.
- When a processor is available, job at head of queue is dynamically dispatched

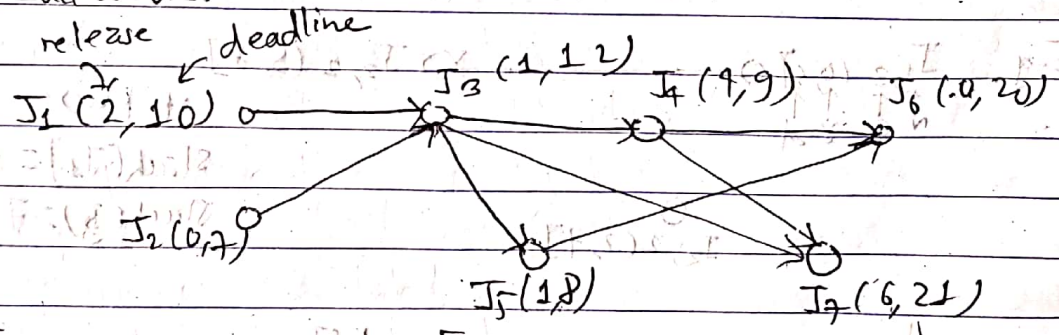
• Effective Release Time (ERT)

- For a job without predecessor, given release time is the effective release time.
- For a job with predecessor, the effective release time is equal to the maximum value among its given release time and the effective release times of all of its predecessors.

• Effective Deadline (ED)

- For a job without successor, effective deadline is equal to its given deadline.
- For a job with successor, effective deadline is equal to minimum value among its given deadline and effective deadlines of all its successors.

E.g.



ERT	ED
$J_1 = 2$, $J_6 = 4$	$J_1 = 8$, $J_6 = 20$
$J_2 = 0$, $J_7 = 6$	$J_2 = 7$, $J_7 = 21$
$J_3 = 2$	$J_3 = 8$
$J_4 = 4$	$J_4 = 9$
$J_5 = 2$	$J_5 = 8$

Scanned by CamScanner

Earliest Deadline First (EDF) Algorithm

- Earlier the absolute deadline, higher the priority

Latest Release Time (LRT)

- This algorithm treats release time as deadlines and deadlines as release time and schedules job backwards starting from latest deadline of all jobs.
- Reason: Completing jobs sooner than deadline has no advantage.

Least Slack Time First (LST)

- also called Minimum-Laxity-First (MLF)
- At any time t, the slack (or laxity) of a job with deadline at d is equal to d-t minus the time required to complete the remaining portion of the job.
- slack (or laxity) = $d - t - e$

  d → deadline; t → time at which we are calculating; e → remaining execution time

- Smaller the slack, higher the priority

E.g. $J_{1,3}(0,6]$ O ——— O $J_{2,2}(5,8]$

$J_{1,3}$ ↑ e, $(0$ ↑ r, $6]$ ↑ d

O

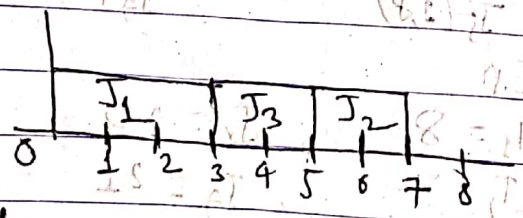
$J_{3,2}(2,7]$

At $t=0$

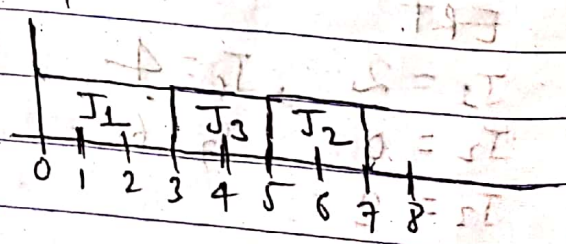
$slack(J_1) = 6-0-3 = 3$

$slack(J_3) = 7-0-2 = 5$

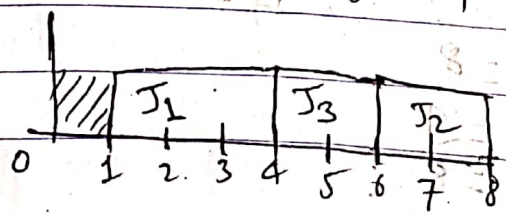
EDF



LST



LRT



Scanned by CamScanner

Date
Page

Precedence Constraints:

The jobs are said to have precedence constraints if they are constrained to execute in some order; otherwise they are said to be independent.

Examples:

1. In a radar surveillance system, the signal-processing task is the producer of track records, while the tracker task is the consumer. In particular, each tracker job processes the track records generated by a signal-processing job. The designer may choose to synchronize the tasks so that the execution of the kth tracker job doesn't begin until the kth signal-processing job completes.
2. We consider queries to an information server. Suppose that before each query is processed and the requested information retrieved, its authentication to access the requested information is first checked. The retrieval job cannot begin execution until the authentication job completes. The communication job that forwards the information to the requester cannot begin until the retrieval job completes.
3. In a communication system, the jobs that generate an acknowledgement of a message and transmit the acknowledgement message cannot begin until the job that receives and processes the message completes.

~~Imme~~ Precedence Graph:

We use a partial order relational operator, $<$, called precedence relation, over a set of jobs specify the precedence constraints among jobs.

- A job J_i is a predecessor of another job J_k (and J_k is the successor of J_i) if J_k cannot begin execution until the execution of J_i completes, represented as $J_i < J_k$.
- J_i is an immediate predecessor of J_k (and J_k is an immediate successor of J_i) if $J_i < J_k$ and there is no another job J_j such that $J_i < J_j < J_k$.

Scanned by CamScanner

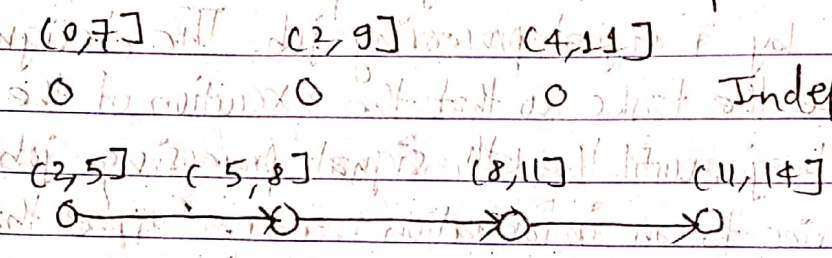
- Two jobs are said to be independent when neither $J_i < J_k$ nor $J_k < J_i$.

How to represent the precedence constraints among jobs?

- By directed graph $G = (J, <)$:

  vertex : Each vertex represents a job J_i

  edge : There is a directed edge from vertex J_i to J_k when J_i is immediate predecessor of J_k.



Date
Page

Optimality of the EDF and LST Algorithms

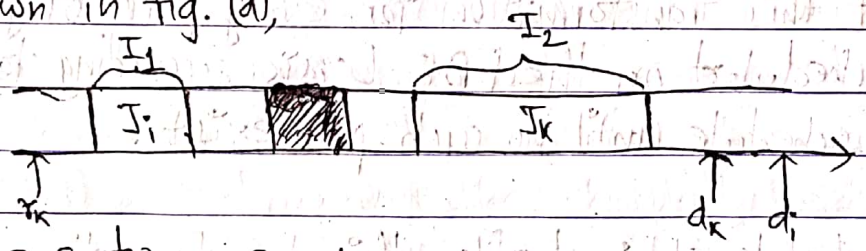
Theorem:

When preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set 'J' of jobs with arbitrary release times and deadlines on a processor if and only if 'J' has feasible schedules.

Proof: The proof is based on the fact: Any feasible schedule of J can be systematically transformed into an EDF schedule (i.e., a schedule produced by the EDF algorithm).

Suppose J has feasible schedules, then we have to prove there is an EDF schedule (produced by EDF algorithm).

Suppose that in a schedule, parts of J_i and J_k are scheduled in intervals I_1 and I_2, respectively. Furthermore, the deadline d_i of J_i is later than the deadline d_k of J_k but I_1 is earlier than I_2 as shown in fig. (a),



There are two cases:

Case I:

The release time of J_k may be later than the end of I_1. J_k cannot be scheduled in I_1; the two jobs are already scheduled on the EDF basis in these intervals. Hence, we need to consider only the second case. ~~where the release time r_k of J_k is~~

Case II:

The release time r_k of J_k is before the end of I_1; without loss of generality, we assume that r_k is no later than the beginning of I_1.

Scanned by CamScanner

To transform the given schedule, we swap J_i and J_k. Specifically, if the interval I_1 is shorter than I_2, as shown in fig (a), we move the portion of J_k that fits in I_1 forward to I_1 and move the entire portion of J_i scheduled in I_1 backward to I_2 and place it after J_k. This is shown in fig (b). Clearly, this swap is always possible.

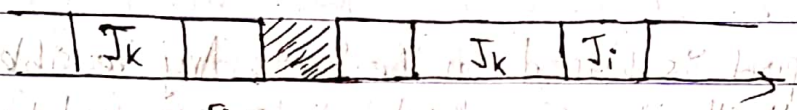


fig (b)

We can do a similar swap if the interval I_1 is longer than I_2: We move the entire portion of J_k scheduled in I_2 to I_1 and place it before J_i and move the portion of J_i that fits in I_2 to the interval. The result of this swap is that these two jobs are now scheduled on the EDF basis. We repeat this transformation for every pair of jobs that are not scheduled on the EDF basis according to the given non-EDF schedule until no such pair exists.

The schedule obtained after this transformation may still not be an EDF schedule if some interval is left idle while there are jobs ready for execution but are scheduled in ~~interval by moving one or more of these jobs~~ a later interval as shown in fig (b). We can eliminate such an idle interval ~~where the jobs were scheduled~~ by moving one or more of these jobs forward into the idle interval and leaving the interval where the jobs were scheduled idle. This is clearly always possible. We repeat this process if necessary until the processor never idles when there are jobs ready for execution as shown in fig (c).

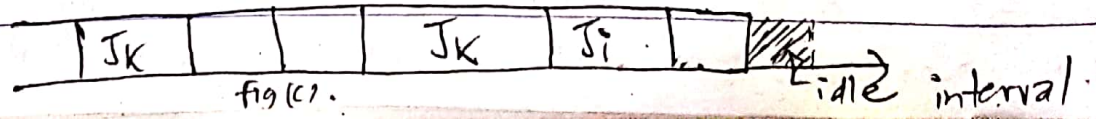


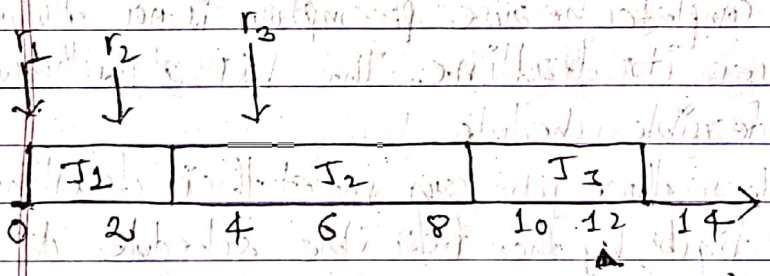
fig (c).

Converse proof:

The preemptive EDF algorithm can always produce a feasible schedule as long as feasible schedules exist, follows straightforwardly from the fact that every feasible schedule can be transformed into a preemptive EDF schedule. If the EDF algorithm fails to produce a feasible schedule, then no feasible schedule exists. (If a feasible schedule were to exist, it could be transformed into an EDF schedule, which contradicts the statement that the EDF algorithm fails to produce a feasible schedule)

Non-Optimality of the EDF and LST Algorithms

The EDF and the LST algorithms doesn't remain optimal if preemption is not allowed or there is more than one processor.



J_3 misses its deadline

fig. (a) EDF schedule

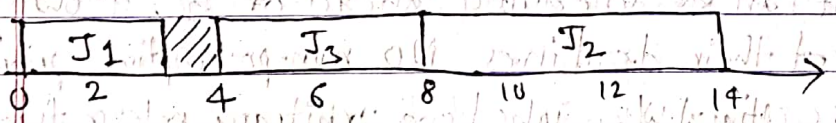


fig (b) A non-EDF schedule

fig. Example illustrating non-optimality of the non-preemptive EDF algorithm

The system shown in the above fig. has three independent, nonpreemptable jobs J_1, J_2 and J_3. Their release times are 0, 2 and 4 respectively, and are indicated by the arrows above the schedules.

Scanned by CamScanner

Their execution times are 3, 6 and 4 respectively; and their deadlines are 10, 14 and 12 respectively.

Job	Release Time	Execution Time	Deadline
J_1	0	3	10
J_2	2	6	14
J_3	4	4	12

fig(a) shows the schedule produced by the EDF algorithm. In particular, when J_1 completes at time 3, J_2 has already been released but not J_3. Hence, J_2 is scheduled. When J_3 is released at time 4, J_2 is executing. Even though J_3 has an earlier deadline and, hence, a higher priority, it must wait until J_2 completes because preemption is not allowed. As a result, J_3 misses its deadline. The LST algorithm would produce the same infeasible schedule.

The fact that these three jobs can meet their deadlines is demonstrated, in fig(b), by the feasible schedule. At time 3 when J_1 completes, the processor is left idle, even though J_2 is ready for execution. Consequently, when J_3 is released at 4, it can be scheduled ahead of J_2, allowing both jobs to meet their deadlines. No non-preemptive priority driven algorithm is optimal when jobs have arbitrary release times, execution times, and deadlines as fig (b) cannot be produced by any priority-driven scheduling algorithm.

The fig. below shows that the EDF algorithm is not optimal for scheduling preemptable jobs on more than one processor. The system in this fig also contains three jobs J_1, J_2 and J_3.

Scanned by CamScanner

Job	Release time	Execution Time	Deadline
J_1	0	1	1
J_2	0	1	2
J_3	0	5	5

The system has two processors. According to the EDF algorithm, J_1 and J_2 are scheduled on the processors at time 0 because they have higher priorities.

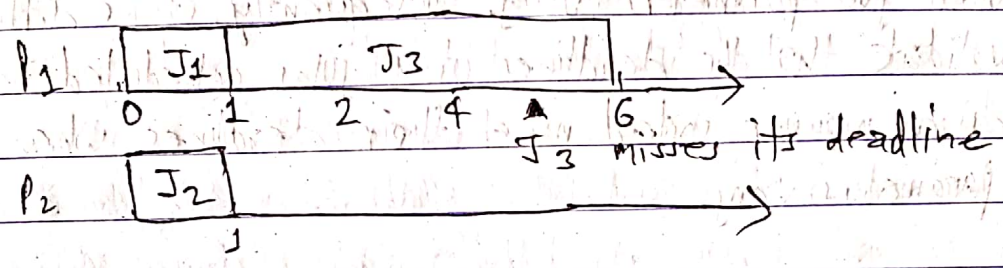


fig. (a) Non-optimality of EDF on multiple processor

An algorithm which assigns a higher priority to J_3 in this case can feasibly schedule the jobs. An example of such algorithms is the LST algorithm. The slacks of the J_1, J_2 and J_3 in the below fig. are 0, 1 and 0 respectively. Hence the algorithm would produce the feasible schedule in below fig. Unfortunately, LST is also not optimal for scheduling jobs on multiple processor.

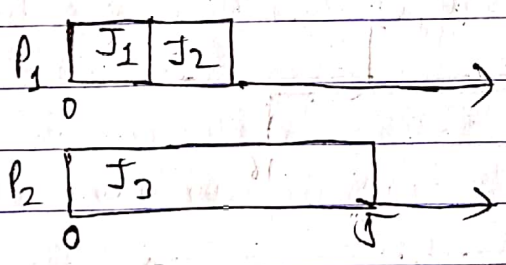


fig. (b) Feasible schedule

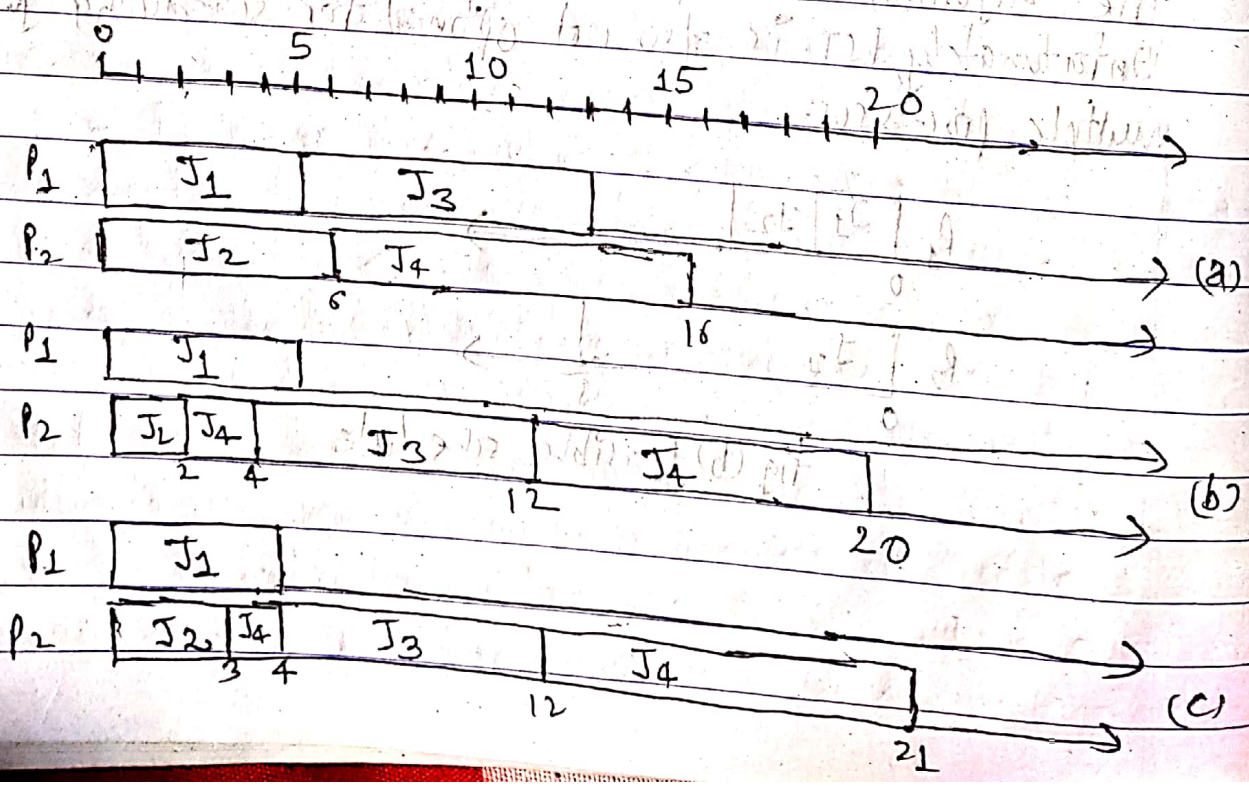
Challenges in Validating timing constraints in priority-driven systems

The validation problem can be stated as follows:
Given a set of jobs, the set of resources available to jobs, and the scheduling (and resource access-control) algorithm to allocate processors and resources to jobs, determine whether all the jobs meet their deadlines.

Timing behavior of a priority-driven system is non-deterministic when job parameters vary. Consequently, it is difficult to validate that the deadlines of all jobs scheduled in a priority-driven manner indeed meet their deadlines when the job parameters vary.

Anomalous Behavior of Priority-Driven Systems

	r_i	d_i	$[e_i^-, e_i^+]$
J_1	0	10	5
J_2	0	10	[2, 6]
J_3	4	15	8
J_4	0	20	10



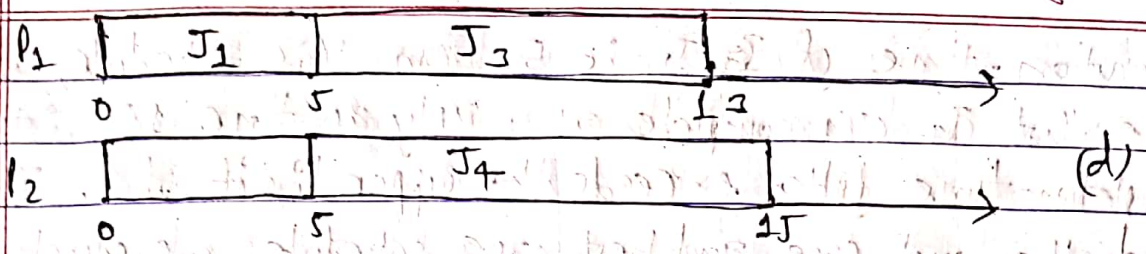


fig. Example illustrating scheduling anomalies

The above fig. gives an example illustrating why the validation problem is difficult when the scheduling algorithm is priority-driven and job parameters may vary. The simple system contains four independent jobs. The jobs are scheduled on two identical processors in a priority-driven manner. The processors maintain a common priority queue, and the priority order is J_1, J_2, J_3 and J_4 with J_1 having the highest priority. In other words, the system is dynamic. The jobs may be preempted but never migrated, meaning that once a job begins executing on a processor, it is constrained to execute on that processor until completion.

The release times, deadlines and execution times of the jobs are listed in the table. Suppose that we schedule the jobs according to their given priorities, assuming first that the execution time of J_2 has the maximum value 6 and then the minimum value 2. The resultant schedules are shown in fig (a) and (b) respectively. Looking at these schedules, we might conclude that all jobs meet their deadlines, and the completion time jitters are sufficiently small. This would be an incorrect conclusion, as demonstrated by the schedules in fig (c) and (d). As far as J_4 is concerned, the worst-case schedule is the one shown in fig (c); it occurs when the execution time of J_2 is 3. According to this schedule, the completion time of J_4 is 21; the job misses its deadline. The best case schedule for J_4 is shown in fig (d); it occurs when the

execution-time of J_2 is 5. From this schedule, we see that J_4 can complete as early as time 15; its completion-time jitter exceeds the upper limit of 4. To find the worst case and best case schedules, we must try all the possible values of e_2. The phenomenon illustrated by this example is known as a scheduling anomaly, an unexpected timing behavior of priority-driven systems.

Predictability of Executions

When the timing constraints are specified in terms of deadlines of jobs, the validation problem is the same as that of finding the worst-case (the largest) completion time of every job.

The schedule of J produced by the given scheduling algorithm when the execution time of every job has its maximum value is called the maximal schedule of J.

The schedule of J produced by the given scheduling algorithm when the execution time of every job has its minimum value is the minimal schedule.

When the execution time of every job has its actual value, the resultant schedule is the actual schedule of J.

Since the range of execution time of every job is known, the maximal and minimal schedules of J can easily be constructed when the release-time jitters are negligible. In contrast, its actual schedule is unknown because the actual values of the execution times are unknown.

Let $s(J_i)$ be the instant of time at which the execution of J_i begins according to the actual schedule of J. $s(J_i)$ is the (actual) start time of J_i. Let $s^+(J_i)$ and $s^-(J_i)$ be the start times of J_i according to the maximal and minimal

schedule of J, respectively. These start times can easily be found by constructing the maximal and ~~minimum~~ minimal schedules and observing when J_i starts according to these schedules. We say that J_i is start-time predictable if $s^-(J_i) \le s(J) \le s^+(J_i)$.

Similarly, let $f^+(J_i)$ and $f^-(J_i)$ be the completion time of J_i according to the maximal schedule and minimal schedule of J, respectively and $f(J_i)$ be the actual completion time (also called finishing time) of J_i according to the actual schedule of J. We say that J_i is completion-time predictable if $f^-(J_i) \le f(J_i) \le f^+(J_i)$. The execution time of J_i is predictable or simply J_i is predictable, if J_i is both start-time and completion-time predictable. The execution behavior of the entire set J is predictable if every job in J is predictable.

For example:

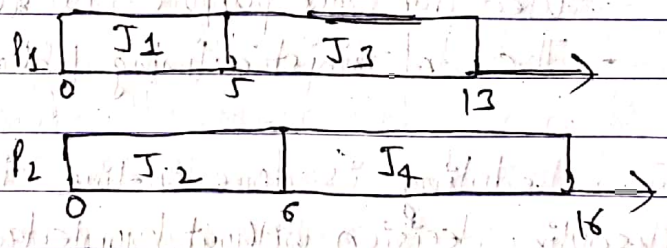
	r_i	d_i	e_i^-, e_i^+
J_1	0	10	5
J_2	0	10	[2, 6]
J_3	4	15	8
J_4	0	20	10

Here, $s^-(J_4) = 2$ and $s^+(J_4) = 6$

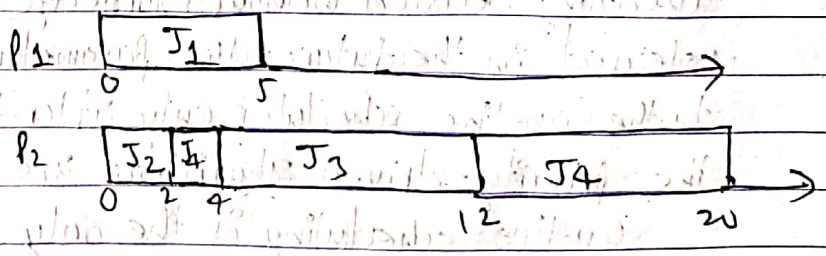
Actual start time is in the range [2,8]

$\therefore J_4$ is start-time predictable.

Maximal schedule:



Minimal schedule:



Also, $f^-(J_4) = 20$ and $f^+(J_4) = 16$, It is impossible for the inequality $20 \le f(J_4) \le 16$ to hold. Hence, J_4 is not completion time predictable, and the system is not predictable.

Scanned by CamScanner

Date
Page

Off-line versus On-line scheduling

A clock-driven scheduler typically makes use of a pre-computed schedule of all hard real-time jobs. This schedule is computed off-line before the system begins to execute, and the computation is based on the knowledge of the release times and processor-time/resource requirements of all the jobs for all times. When the operation mode of the system changes, the new schedule specifying when each job in the new mode executes is also precomputed and stored for use. Scheduling is (done) off-line, and the precomputed schedules are off-line schedules.

Disadvantages:

- Off-line scheduling is not flexible.
- It can be used only when the system is deterministic, meaning that the system provides some fixed set(s) of functions and that the release times and processor-time/resource demands of all its job are known and do not vary or vary only slightly.

Advantages (for a deterministic system):

- Because the computation of the schedules is done off-line, the complexity of the scheduling algorithm(s) used for this purpose is not important.
- The deterministic timing behaviour of the resultant system

The scheduling is done on-line if the scheduler makes each scheduling decision without knowledge about the jobs that will be released in the future; the parameters of each job become known to the on-line scheduler only after the job is released. The priority-driven algorithms are on-line algorithms.

On-line scheduling is the only option in a system whose future workload is unpredictable. An online scheduler can accomodate

Scanned by CamScanner

clairvoyant : able to forsee the future

dynamic variations in user demands and resource availability.

Advantages:
- Flexibility and Adaptability

Disadvantages:
- Reduced ability for the scheduler to make the best use of system resources.
- Without prior knowledge about future jobs, the scheduler cannot make optimal scheduling decisions.

Overloaded System

The system is said to be overloaded when the jobs offered to the scheduler cannot be feasibly scheduled even by a clairvoyant scheduler. During an overload, some jobs must be discarded in order to allow other jobs to complete in time.

Competitive factor

The value of a job is equal to its execution time if the job completes by its deadline according to a given schedule and is equal to zero if the job fails to complete in time according to the schedule.

The value of a schedule of a sequence of jobs is equal to the sum of the values of all the jobs in the sequence according to the schedule.

A scheduling algorithm is optimal if it always produces a schedule of the maximum possible value for every finite set of jobs.

An on-line algorithm has a competitive factor c if and only if the value of the schedule of any finite sequence of jobs produced by the algorithm is at least c times the value of the schedule of the jobs produce by an optimal clairvoyant algorithm.

Scanned by CamScanner

Unit-5

Clock-Driven Scheduling

Notations and Assumptions

The clock-driven approach to scheduling is applicable only when the system is deterministic, except for a few aperiod and sporadic jobs - to be accommodated in the deterministic framework.

Assumptions:

1. There are n periodic tasks in the system. As long as the system stays in an operation mode, n is fixed.
2. The parameters of all periodic tasks are known a priori. ↳ from before
   For all pratical purposes, each job in T_i is released p_i units of time after the previous job in T_i
3. Each job $J_{i,k}$ is ready for execution at its release time $r_{i,k}$.

Operating system maintains a queue for aperiodic jobs. Aperiodic jobs are released at unexpected times. On the arrival of aperiodic job, scheduler is not intervened. Aperiodic jobs are ordered suitably in the queue by OS and are scheduled whenever the processor is available.

Notations:

A periodic task T_i is represented by 4 Tuple;

$$T_i = (\phi_i, p_i, e_i, D_i)$$

It can be abbreviated to;

$T_i = (p_i, e_i, D_i)$ $\phi_i = 0$ (default)

$T_i = (p_i, e_i)$ $p_i = D_i$ (default)

Task utilization $(U_i) = \frac{e_i}{p_i}$

Where,

ϕ_i = phase of ith periodic task

p_i = Periodic of ith periodic task

e_i = Execution time

D_i = relative deadline

Scanned by CamScanner

For example: (1, 10, 3, 6) is a periodic task whose phase is 1, period is 10, execution time is 3, and relative deadline is 6. Therefore, the first job in this task is released and ready at time 1 and must be completed by time 7; the second job is ready at 11 and must be completed by 17, and so on. Each of these jobs executes for at most 3 units of time.

Static Timer-Driven Scheduler

It is assumed that the operating system maintains a queue for aperiodic jobs. When an aperiodic job is released, it is placed in the queue without the attention of the scheduler. Whenever the processor is available for aperiodic jobs, the job at the head of this queue executes.

Scheduler produces a static schedule of the jobs which is computed offline and stored in a table.

According to the schedule, the amount of processor time allocated to every job is equal to its maximum execution time, and every job completes by its deadline. During runtime, jobs are dispatched as per schedule. The processor idles nearly periodically to accomodate aperiodic jobs.

For example: We consider a system that contains four independent periodic tasks; $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ and $T_4 = (20, 2)$. Their utilizations are $\frac{1}{4} = 0.25$, 0.36, 0.05 and 0.1 respectively, and the total utilization is 0.76.

It suffices to construct a static schedule for the first hyperperiod of the tasks. The length of each hyperperiod = lcm (4, 5, 20, 20) = 20

The entire schedule consists of replicated segments of length 20.

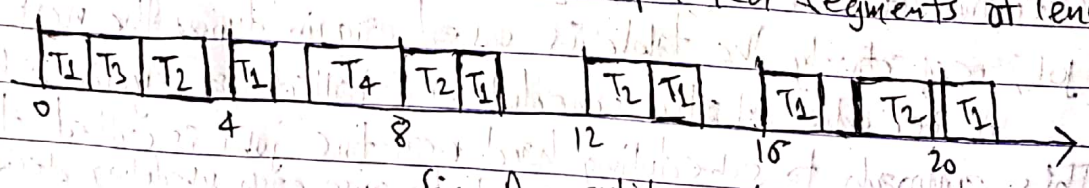


fig. An arbitrary static schedule

The fig. shows a schedule segment on one processor. T_1 starts execution at time 0, 4, 9.8, 13.8, and so on; T_2 starts execution at 2, 8, 12, 18, and so on. All task meet their deadlines.

Unused intervals like (3.8, 4), (5, 6) and (10.8, 12). can be used to execute aperiodic jobs.

A straightforward way to implement the scheduler is to store the precomputed schedule as a table. Each entry $(t_k, T(t_k))$ in this table gives a decision time t_k, which is an instant when a scheduling decision is made, and $T(t_k)$, which is either the name of the task whose job should start at t_k or I. The later indicates an idle interval during which no periodic task is scheduled.

The scheduler makes use of a timer. Immediately after all the tasks have been created and initialized and then at every scheduling decision time, the scheduler sets the timer so the timer will expire and request an interrupt at the next decision time. Upon receiving a timer interrupt at t_k, the scheduler sets the timer to expire at t_{k+1} and prepares the task $T(t_k)$ for execution. It then suspends itself, letting the task have processor and execute. When the timer expires again, the scheduler repeats this operation.

In the above example in fig., the stored table contains 17 entries. They are (0, T_1), (1, T_3), (2, T_2), (3.8, I), (4, T_1)... (19.8, I). Hence, the timer is set to expire at 0, 1, 2, 3.8, and so on. At these times, the scheduler schedules the execution of tasks T_1, T_3, T_2 and an aperiodic or background job, respectively. The table is used again during the next hyperperiod.

A periodic static schedule is called a cyclic schedule. This approach to scheduling hard real-time jobs is called the clock-driven or time-driven approach because each scheduling decision is made at specific time, independent of events.

Date
Page

General Structure of Cyclic Schedules

The cyclic schedule and the scheduler should have the following characteristics:

* Frames and Major Cycles
* Frame Size Constraints
* Job Slices

Frames and Major Cycles

- To keep the stored table small, the timeline is divided into frames.
- Scheduling decisions are made at (periodically) frame boundaries and not at arbitrary times t_k.
- Each frame has size 'f' called frame length.
- Preemption within frame is not allowed.
- Phase of each task is a non-negative integer multiple of frame size, i.e. the first job of every task is released at the beginning of some frame.

Frame Size Constraints

1) The frames needs to be sufficiently long so that every job can start and complete its execution within a frame. This can be achieved by making the frame size f larger than the execution time e_i of every task T_i, i.e.

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

2) To keep the length of the cyclic schedule as short as possible, the frame size f should be chosen so that it divides H, the length of the hyperperiod of the system. This condition is met when f divides the period p_i of at least one task T_i, i.e.

$$\left\lfloor \frac{p_i}{f} \right\rfloor - \frac{p_i}{f} = 0 \quad \text{for at least one } i.$$

Scanned by CamScanner

This condition ensures that there is an integer number of frames in each hyperperiod (H).

$$F = \frac{H}{f}$$ where H = major cycle = hyperperiod
F = minor cycle
f = frame size

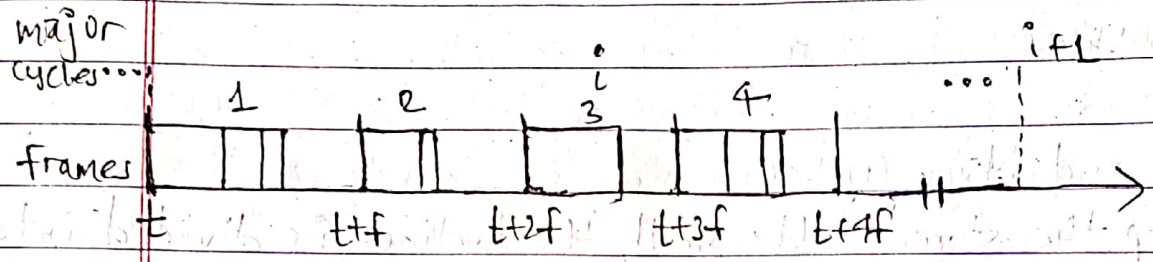


fig. General structure of a cyclic schedule

3) The frame size should be sufficiently small so that between the release time and deadline of every job, there is at least one frame, so as to make it possible for the scheduler to determine whether every job completes by its deadline.

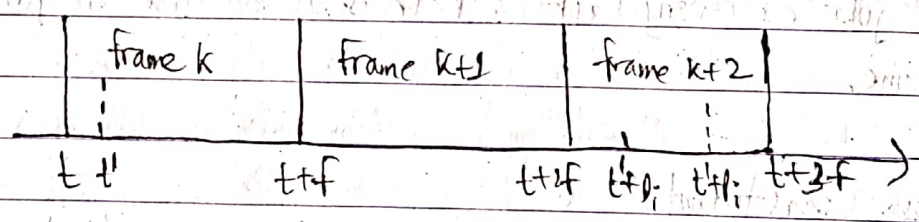


fig. A constraint on the value of frame size

fig. illustrates the suitable range of F for a task $T_i = (P_i, e_i, D_i)$. When f is in this range, there is at least one frame between the release time and deadline of every job in the task.

In the fig., t denotes the beginning of a frame (called the kth frame) in which a job in T_i is released.
t' denotes the release time of this job.
We consider two cases: (i) $t' > t$ and (ii) $t' = t$.

Scanned by CamScanner

Case I: $t' > t$

If t' is later than t, as shown in fig, we want the $(k+1)^{th}$ frame to be in the interval between the release time t' and the deadline $t'+D_i$ of this job. For this to be true, we must have $t+2f$ equal to or earlier than $t'+D_i$, i.e.

$$2f-(t'-t) \le D_i$$

$t'-t$ is at least equal to the $\gcd(P_i, f)$.

$$\therefore 2f - \gcd(P_i, f) \le D_i \text{ for all } i = 1, 2, \ldots, n.$$

Case II: $t' = t$

The condition $f \le D_i$ is satisfied for all values of f that satisfy $2f - \gcd(P_i, f) \le D_i$ and hence doesn't need to be considered separately. Thus, we choose a frame size that is equal to or less than D_i.

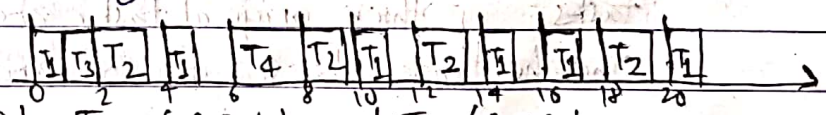
To summarize, the frame size constraints are:-

1) $f \ge \max_{1 \le i \le n}(e_i)$

2) $\lfloor P_i/f \rfloor - P_i/f = 0$

3) $2f - \gcd(P_i, f) \le D_i$

cyclic schedule with frame size 2



for example:

$T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ and $T_4 = (20, 2)$

$H = \text{lcm}(4, 5, 20, 20) = 20$

Hence, 2, 4, 5, 10 and 20 are possible frame sizes according to constraint (2) and constraint (1).

$f \ge 2$ and $\left\lfloor \frac{P_i}{f} \right\rfloor - \frac{P_i}{f} = \left\lfloor \frac{4}{2} \right\rfloor - \frac{4}{2} = 0$.

From constraint 3, $2f - \gcd(P_i, f) \le D_i$

$i=1$ $\quad 4 - \gcd(4, 2) \le 4$, i.e. $4 - 2 \le 4$ (true)

$i=2$ $\quad 4 - \gcd(5, 2) \le 5$ (True)

$i=3, 4$ $\quad 4 - \gcd(20, 2) \le 20$ (True)

Hence, framesize 2 satisfies all the constraints.

Scanned by CamScanner

Job Slices

Sometimes, the given parameters of some task systems cannot meet all three frame size constraints simultaneously.
For example: $T = \{(4,1), (5,2,7), (20,5)\}$. For first constraint to be true, $f \geq 5$ but to satisfy third constraint $f \leq 4$.

In this situation, we are forced to partition each job in a task that has a larger execution time into slices (i.e. subjobs) with smaller execution times.

For $T = \{(4,1), (5,2,7), (20,5)\}$, we can divide each job in $(20,5)$ into a chain of three slices with execution times 1, 3, and 1. The task $(20,5)$ now consists of three subtasks $(20,1)$, $(20,3)$ and $(20,1)$.

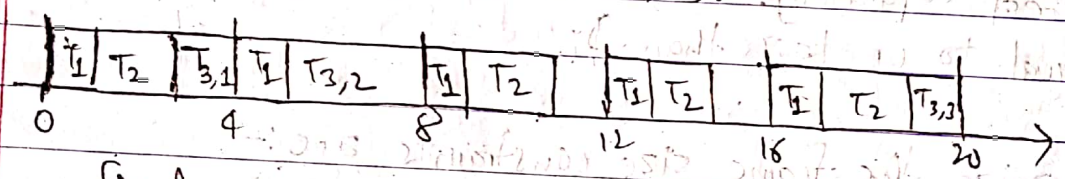


fig. A preemptive cyclic schedule of $T = \{(4,1), (5,2,7), (20,5)\}$

The resultant system has five tasks for which we can choose the frame size 4. The fig. shows a cyclic schedule for these tasks. The three original tasks are called T_1, T_2 and T_3, respectively and the three subtasks of T_3 are called $T_{3,1}$, $T_{3,2}$ and $T_{3,3}$.

Cyclic Executives

- It is a scheduler that deterministically interleaves and sequentializes the execution of periodic tasks on a CPU according to a given cyclic schedule.
- Starting from the beginning of each frame, it executes in turn the slices scheduled in the frame.

Scanned by CamScanner

Improving the average Response time of aperiodic jobs

There is no advantage to completing a job with a hard deadline early. Aperiodic job is released and executed by the system in response to an event. The sooner an aperiodic job completes, the more responsive the system is.

Slack Stealing

Executing the aperiodic jobs ahead of the periodic jobs such that periodic jobs do not miss their deadline (whenever possible) so as to improve the response times of aperiodic jobs is called slack stealing.

Let the total amount of time allocated to all the slices scheduled in the frame k be x_k. The slack (time) available in the frame is equal to $f - x_k$ at the beginning of the frame. If the aperiodic job queue is non-empty at this time, the cyclic executive can let aperiodic jobs execute for this amount of time without causing any job to miss its deadline.

After y units of slack time are consumed by aperiodic jobs, the available slack is reduced to $f - x_k - y$. The cyclic executive can let the aperiodic jobs execute in the frame k as long as there is slack, i.e. $f - x_k - y \geq 0$ in the frame k.

If the aperiodic job queue is empty, the cyclic executive lets the periodic task server execute the next slice in the current block. As long as there is slack, the cyclic executive returns to examine the aperiodic job queue after each slice completes.

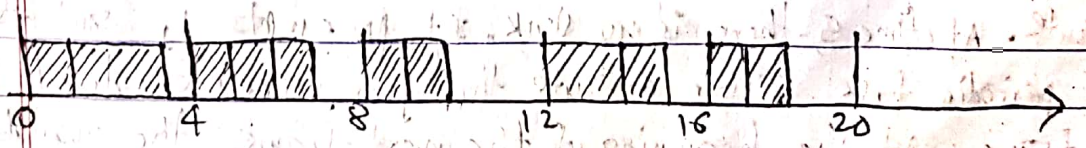


fig. (a)

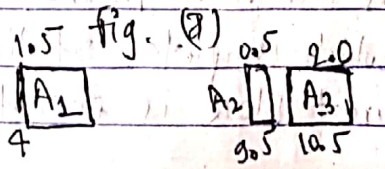


fig (b)

fig.(a) shows the first major cycle in the cyclic schedule of the periodic tasks. fig(b) shows three aperiodic jobs A_1, A_2, and A_3. Their release times are immediately before 4, 9.5 and 10.5, and their execution times are 1.5, 0.5 and 2 respectively.

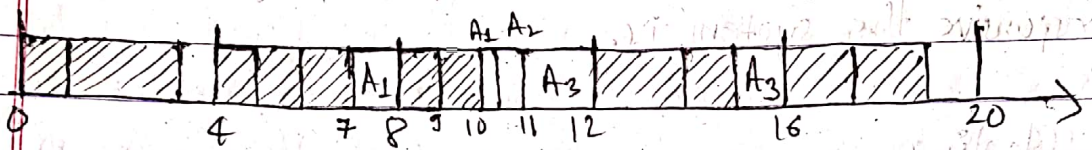


fig.(c) Without slack stealing

Response time of $A_1 = 10.5 - 4 = 6.5$
Response time of $A_2 = 11 - 9.5 = 1.5$
Response time of $A_3 = 16 - 10.5 = 5.5$

Average response time $= \frac{6.5 + 1.5 + 5.5}{3} = 4.5$

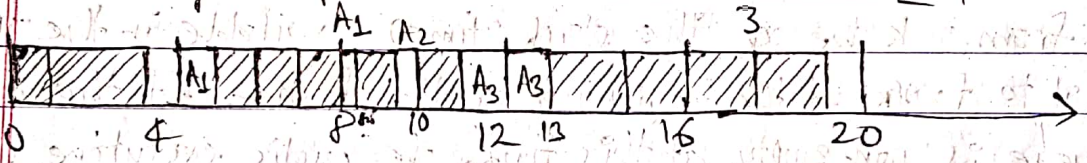


fig. (d) Example illustrating slack stealing

Response time of $A_1 = 8.5 - 4 = 4.5$
Response time of $A_2 = 10 - 9.5 = 0.5$
Response time of $A_3 = 13 - 10.5 = 2.5$
Average response time $= \frac{4.5 + 0.5 + 2.5}{3} = 2.5$

fig(d) shows what happens if the cyclic executive does slack stealing. At time 4, the cyclic executive finds A_1 in the aperiodic job queue, and there is 1 unit of slack. Hence, it lets A_1 execute. At time 5, there is no slack. It preempts A_1 and lets the periodic task server execute the job slices scheduled in the frame. At the beginning of the next frame, the available slack is 2. It resumes A_1, which completes at time 8.5. At the time, the first slice in the current block is executed, since the

aperiodic job queue is empty. Upon completion of the slice at time 9.5, the cyclic executive checks the aperiodic queue again, finds A_2 ready and lets A_2 execute. When the job completes at time 10, the cyclic executive finds the aperiodic job queue empty and lets the periodic task server execute the next job slice in the current block. At time 11, it finds A_3 and lets the job execute during the last unit of time in the ~~current block.~~ A frame, as well as in the beginning of the next frame. The job completes by 13.

Hence, the average response time of the aperiodic jobs is improved.

Scheduling Sporadic Jobs

- Like jobs in periodic tasks, sporadic jobs have hard deadlines.
- Their minimum release times and maximum execution times are unknown a priori.

Thus, it is impossible to guarantee a priori that all sporadic jobs can complete in time.

Acceptance Test

During an acceptance test, the scheduler checks whether the newly released sporadic job can be feasibly scheduled with all the jobs in the system at the time.

Acceptance: If according to the existing schedule, there is a sufficient amount of time in the frames before its deadline to complete the newly released sporadic job without causing any job in the system to complete too late, the scheduler accepts and schedules the job, rejects otherwise.

If the sporadic job is rejected, the scheduler gives the application system as much time as there is to take any necessary recovery action.

Let us suppose that at the begginning of frame t, an

acceptance test is done on a sporadic job $S(d,e)$, with deadline d and (maximum) execution time e. Suppose that the deadline d of S is in frame $l+1$ and $l \geq t$. Clearly, the job must be scheduled in the lth or earlier frames. The job can complete in time only if the current (total) amount of slack time $\sigma_c(t,l)$ in frames $t, t+1, \ldots, l$ is equal to or greater than its execution time e, i.e. $\sigma_c(t,l) \geq e$.

The scheduler should reject S if $e > \sigma_c(t,l)$.

The scheduler accepts the new job $S(d,e)$ only if $e \leq \sigma_c(t,l)$ and no sporadic jobs in the system are adversely affected.

EDF Scheduling of the Accepted Jobs

The scheduler maintains a queue of accepted sporadic jobs in non-decreasing order of their deadlines and inserts each newly accepted sporadic job into this queue in this order.

Whenever all the slices of periodic tasks scheduled in each frame are completed, the cyclic executive lets the jobs in the sporadic queue execute in the order they appear in the queue.

Example:

$f = 4$, $S_1 = (17, 4.5)$, $r_1 = 3$

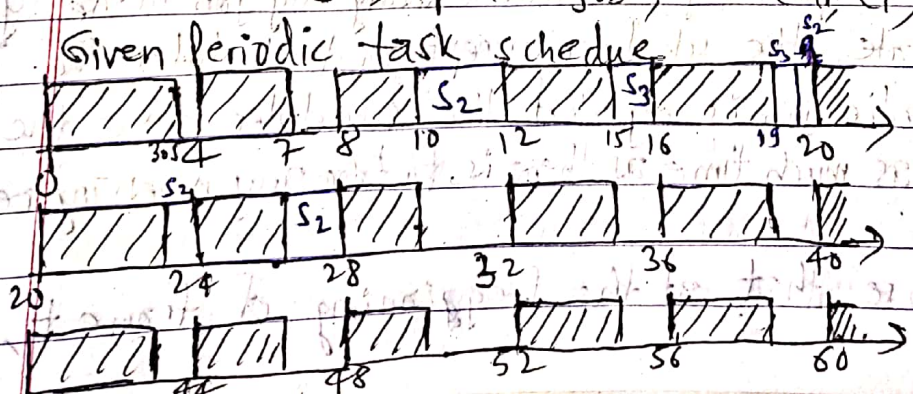
$S_2 = (29, 4)$, $r_2 = 5$

$S_3 = (22, 1.5)$, $r_3 = 11$

$S_4 = (44, 5)$, $r_4 = 14$

$S \rightarrow$ sporadic job ; $S = (d_i, e_i)$

Given periodic task schedule



Scanned by CamScanner

The shaded boxes show where periodic tasks are scheduled.

- Suppose that at time 3, a sporadic job $S_1(17, 4.5)$ with execution time 4.5 and deadline 17 is released. The acceptance test on this job is done at time 4, i.e. the beginning of frame 2. S_1 must be scheduled in frames 2, 3 and 4. In these frames, the total amount of slack time is only 4, which is smaller than the execution time of S_1. Consequently, the scheduler rejects the job.
- At time 5, $S_2(29, 4)$ is released. Frames 3 through 7 end before its deadline. During the acceptance test at 8, the scheduler finds the total amount of slack in these frames is 5.5. Hence, it accepts the job S_2. The first part of S_2 with execution time 2 executes in the current frame.
- At time 11, $S_3(22, 1.5)$ is released. At time 12, the scheduler finds 2 units of slack time in frames 4 and 5, where S_3 can be scheduled. Moreover, there still is enough slack to complete S_2 even though S_3 exeute ahead of S_2. Consequently, the scheduler accepts S_3. This job executes in frame 4.
- Suppose that at time 14, $S_4(44, 5)$ is released. At time 16 when the acceptance test is done, the scheduler finds only 4.5 units of time available in frames before the deadline of S_4, after it has accounted for the slack time that has already been committed to the remaining portion of S_2 and S_3. Therefore, it rejects S_4.

The Acceptance Test

An acceptance test is carried out at the beginning of frame t to decide whether a sporadic job $S(d, e)$ should be accepted or rejected. The acceptance test consists of the following two steps:

1. The scheduler first determines whether the current total amount of slack in the frames before the deadline of job S is at least equal to the execution time e of S. If the answer is no, it rejects S. If yes, the second step is carried out.

Scanned by CamScanner

2. In the second step, the scheduler determines whether any sporadic job in the system will complete late if it accepts S. If the acceptance of S will not cause any sporadic job in the system to complete too late, it accepts S; otherwise, it rejects S.

Practical Considerations and Generalizations

The practical problems are:-

- how to handle frame overruns
- how to do mode changes
- how to schedule tasks on multiprocessor systems.

Handling Frame Overruns

Causes:

* When the execution time of a job is input data dependent, it can become unexpectedly large for some rare combination of input values.
* A transient hardware fault or a software flaw undetected during debugging and testing may case some job to execute longer than expected.

Ways to handle:

* ~~Killing the job and starting an error recovery task.~~
* Abort the overrun job at the beginning of the next frame, log the premature termination of the job and start an error recovery task
* Preempting the job and scheduling the remainder as an aperiodic job.

Mode Changes

During a mode change, the system is reconfigured. Mode switching implies reconfiguring the scheduler and bringing in the

Scanned by CamScanner

code/data for new jobs.

- New mode may not have all old jobs and new jobs may be added.
- Reconfiguration of job can take a long time. Schedule the reconfiguration of job as an aperiodic or sporadic task to ensure other deadlines are met during mode changes.

Multiprocessor Scheduling

Tasks/Jobs can be scheduled on several processors whenever the workload parameters are known a priori and there is a global clock. It can be computed off-line but off-line scheduling table generation is more complex. Thus, searching a feasible multiprocessor schedule is considerably more complex than for a uniprocessor schedule.

Algorithm for Constructing Static Schedules

The general problem of choosing a minor frame length for a given set of periodic tasks, segmenting the tasks if necessary, and scheduling the tasks so that they meet all their deadlines is NP-hard. For the special case where the periodic tasks contains no non-preemptable section, there exists a polynomial time solution.

Scheduling Independent Preemptable Tasks

The system of independent, preemptable periodic tasks whose relative deadlines are equal to or greater than their periods is schedulable if ~~relative~~ and only if the total utilization of the tasks is no greater than 1.

The iterative algorithm (iterative-network-flow algorithm) or the INF algorithm in short, enables us to find a feasible cyclic schedule if one exists. The INF algorithm iteratively

Scanned by CamScanner

tries to find a feasible cyclic schedule of the system for a possible frame size at a time, starting from the largest possible frame size in order of decreasing frame size.

Network-Flow Graph

The jobs to be scheduled in a major cycle of F frames are $J_1, J_2, \ldots, J_N$. The constraints on when the jobs can be scheduled are represented by the network-flow graph of the system.

1. There is a job vertex J_i representing each job J_i, for $i = 1, 2, \ldots, N$
2. There is a frame vertex named j representing each frame j in the major cycle, for $j = 1, 2, \ldots, F$.
3. There are two special vertices named source and sink.
4. There is a directed edge (J_i, j) from a vertex J_i to a frame vertex j if the job J_i can be scheduled in the frame j, and the capacity of the edge is the frame size f.
   ↳ non-negative number associated with the edge.
5. There is a directed edge from the source vertex to every job vertex J_i, and the capacity of this edge is the execution time e_i of the job.
6. There is a directed edge from every frame vertex to the sink, and the capacity of this edge is f.

A flow of an edge is a non-negative number that satisfies the following constraints:

(1) It is no greater than the capacity of the edge
(2) with the exception of the source and sink, the sum of the flows of all the edges into every vertex is equal to the

sum of the flows of all the edges out the vertex.

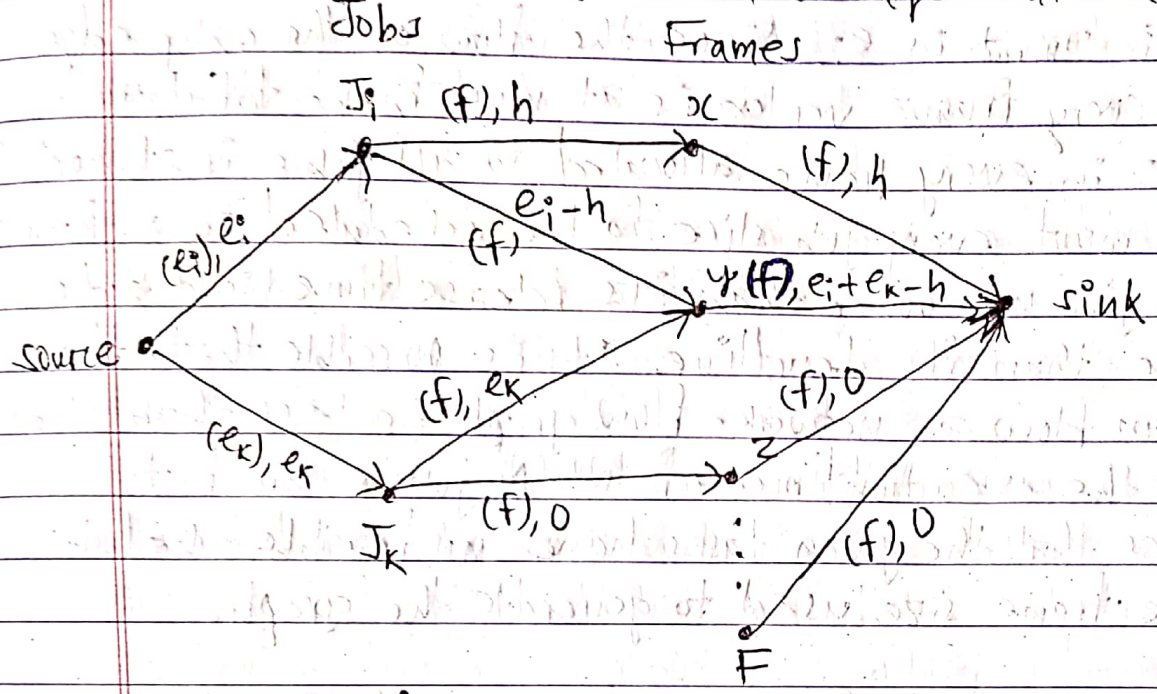


fig. Part of a network-flow graph

The label "(capacity), flow" of each edge gives its capacity and flow. This graph indicates that job J_i can be scheduled in frames x and y and the job J_k can be scheduled in frames y and z.

Maximum Flow and Feasible Preemptive Schedule

The maximum flow of a network-flow graph is at most equal to the sum of the execution times of all ~~vertices that gives~~ the jobs to be scheduled in a major cycle. The set of ~~jobs~~ flows of edges from job vertices to frame vertices that gives this maximum flow represents a feasible preemptive schedule of the jobs in the frames.

The flow of an edge (J_i, j) from a job vertex J_i to a frame vertex j gives the amount of time in frame j allocated to job J_i. The total amount of time allocated to a job J_i is represented by the total flow out of all the edges from the job

vertex J_i. Since this amount is equal to the flow into J_i, this amount is e_i. Since the flow of the only edge out of every frame vertex is at most f, the total amount of time in every frame allocated to all jobs is at most f.

We want every job slice to be scheduled in a frame which begins no sooner than its release time and ends no later than its deadline. It is possible that the maximum flow a network-flow graph is less than the sum of the execution times of all N jobs. This fact indicates that the given tasks have no feasible schedule with the frame size used to generate the graph.

Pros and Cons of Clock-Driven Scheduling

Advantages:

- Conceptual Simplicity:
  We can take into account complex dependencies, communication delays, and resource contention among jobs in the choice and construction of static schedule making sure there will be no deadlock and unpredictable jobs.

- Jobs can be scheduled according to different static schedules in different modes just by changing the table which represents static schedules.

- No need for concurrency control

- Precedence constraint and other dependency can be taken care of by the choice of the schedule. So, there is no need for synchronization.

Scanned by CamScanner

- When workload is mostly periodic and schedule is cyclic, timing constraints can be checked and enforced at frame boundary.

- Context switching and communication overhead can be kept low.

- Relatively easy to validate, test and certify.

- Possible to determine whether the system indeed meets all of its timing requirement by exhaustive simulation and testing.

Disadvantages:

- System based on this approach is brittle, i.e. relatively difficult to modify and maintain.
- Suited for system which are rarely modified.
- Release time of all jobs must be fixed.
- All combination of periodic task might execute at the same time must be known a priori so a schedule for the combination can be precomputed.
- Not suitable for system that contain both hard and soft real time application.

Scanned by CamScanner

Chapter 6

Priority-Driven Scheduling of Periodic Tasks

The simplifying assumptions are

* the task are independent
* there are no aperiodic and sporadic tasks

Restrictions

* Every job is ready for execution as soon as it is released, can be prempted at any time, and never suspends itself.
* The context switch overhead is negligibly small compared with execution times of the task.
* The no. of priority levels is unlimited.
* Scheduling decisions are made immediately upon job releases and completions.
* Fixed number of periodic tasks
* Scheduling on uniprocessor system is assumed.

Fixed Priority V/s Dynamic Priority Algorithms:

- Priority-driven scheduler is an online scheduler. Hence, it doesn't precompute a schedule of task.
- It assigns priorities to jobs after they are released and places the jobs in a ready job queue in priority order.
- When preemption is allowed at anytime, a scheduling decision is made whenever a job is released or completed.
- At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue.

* A fixed-priority algorithm assigns the same priority to all the jobs in each task. In other words, the priority of each periodic task is fixed relative to other tasks
* A dynamic-priority algorithm assigns different priorities to the individual jobs in each task. Hence, the priority of the task with respect to that of the other tasks changes as jobs are released and completed.

Scanned by CamScanner

* Most real-time scheduling algorithms of practical interest are fixed priority scheduling algorithms.

Categories of algorithms
- fixed priority algorithms
- task-level dynamic priority (and job level fixed-priority) algorithms
- job-level (and task-level) dynamic algorithms

Fixed Priority Algorithm:

1. Rate Monotonic Algorithm (RMA):

This algorithm assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. The rate (of job releases) of a task is the inverse of its period. Hence, the higher its rate, the higher the priority.

Example:

The system contains three tasks: $T_1 = (4,1)$, $T_2 = (5,2)$ and $T_3 = (20,5)$. The priority of T_1 is the highest because its rate is the highest. T_2 has the next highest priority.

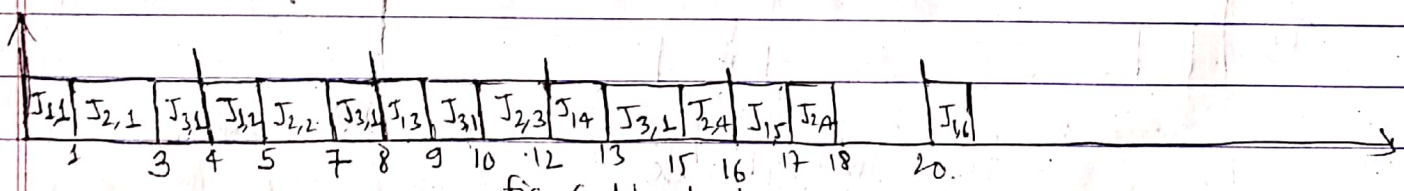
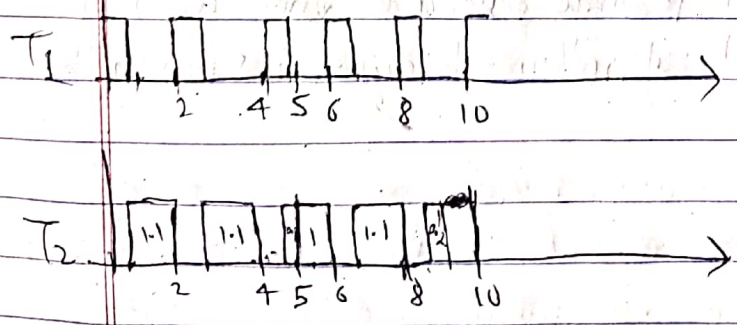


fig. Gantt chart

Example 2: $T_1 = (2, 0.9)$ and $T_2 = (5, 2.3)$

Represent in timeline:



Scanned by CamScanner

2. Deadline Monotonic Algorithm (DMA):

This algorithm assigns priorities to tasks according their relative deadlines: the shorter the relative deadline, the higher the priority.

For example: The system consists of three tasks: $T = (\phi, P_i, e_i, D_i)$

$T_1 = (50, 50, 25, 100)$, ~~and~~ $T_2 = (0, 62.5, 10, 20)$ and

$T_3 = (0, 125, 25, 50)$

Their utilizations are ~~$\frac{1}{50}, \frac{1}{62.5}$ and $\frac{1}{125}$, i.e. 0.2~~, $\frac{25}{50}, \frac{10}{62.5}, \frac{25}{125}$, i.e. 0.5, 0.16 and 0.2

According to DM, T_2 has the highest priority and T_1 has the lowest.

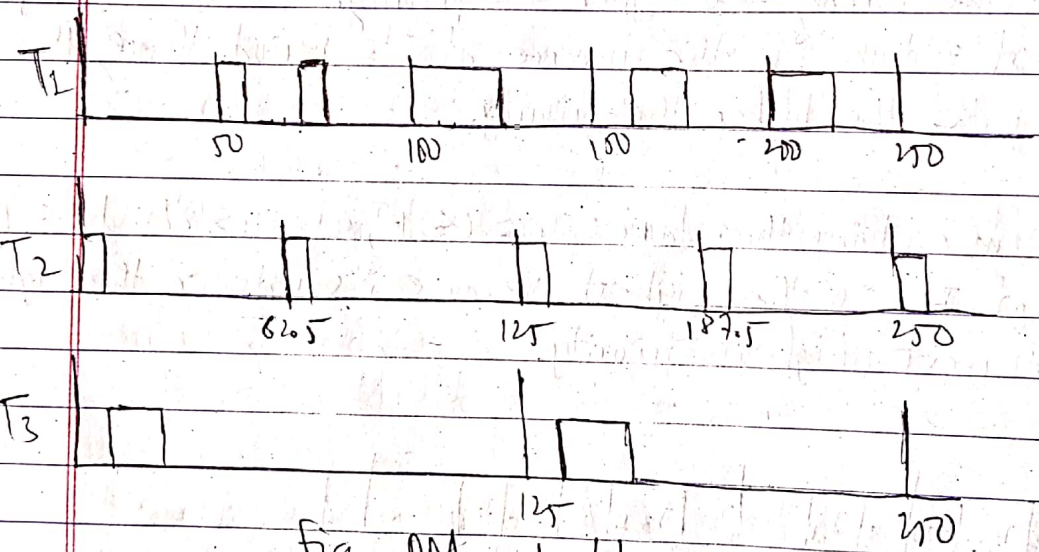


fig. DM schedule

When the relative deadline of every task is proportional to its period, the RM and DM algorithms are identical. When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a feasible schedule when the RM algorithm fails, while the RM algorithm always fails when the DM algorithm fails.

Scanned by CamScanner

Date
Page

Well-Known Dynamic Algorithms:

The EDF algorithm assigns priorities to individual jobs in the task according to their absolute deadlines; it is a dynamic-priority algorithm.

Example: $T_1 = (2, 0.9)$ and $T_2 = (5, 2, 3)$

- At time 0, the first jobs $J_{1,1}$ and $J_{2,1}$ of both tasks are ready. The deadline of $J_{1,1}$ is 2 while the deadline of $J_{2,1}$ is 5. Consequently, $J_{1,1}$ has a higher priority and executes. When $J_{1,1}$ completes, $J_{2,1}$ begins to execute.
- At time 2, $J_{1,2}$ is released, and its deadline is 4, earlier than the deadline of $J_{2,1}$. Hence, $J_{1,2}$ is placed ahead of $J_{2,1}$ in the ready job queue. $J_{1,2}$ preempts $J_{2,1}$ and executes.
- At time 2.9, $J_{1,2}$ completes. The processor then executes $J_{2,1}$.
- At time 4, $J_{1,3}$ is released; its deadline is 6, which is later than the deadline of $J_{2,1}$. Hence the processor continues to execute $J_{2,1}$.
- At time 4.1, $J_{2,1}$ completes, the processor starts to execute $J_{1,3}$ and so on.

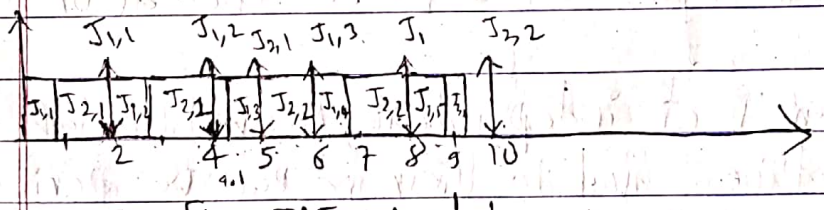


fig. EDF schedule

Other dynamic-priority algorithms are LST, FIFO, LIFO, etc.

A scheduling algorithm can feasibly schedule any set of periodic tasks 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 feasibly schedule a set of tasks with a total utilization greater than 1.

Scanned by CamScanner

An optimal algorithm has schedulable utilization equal to 1.
EDF and LST are optimal but RM and DM are not.
Advantage of fixed priority algorithm: The timing behaviour of a system scheduled according to a fixed priority algorithm is more predictable. It is possible to predict which tasks will miss their deadlines during an overload.
In EDF, a late job which has already missed its deadline has a higher priority than a job whose deadline is still in future. This causes many other jobs to be late.

Maximum Schedulable Utilization

A system is schedulable by an algorithm if the algorithm always produces a feasible schedule of the system.
A system is schedulable (and feasible) if it is schedulable by some algorithm, i.e. feasible schedules of the system exist.

Current period: At any time t, the current period of a task is the period that begins before t and ends at or after t.

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.

$P_i = D_i \Rightarrow$ The job in each period completes before the next period starts so there is no backlog of jobs.

The following facts follow straightforwardly from the above theorem.

1. A system of independent, preemptable tasks with relative deadlines longer than their periods can be feasibly scheduled on a processor

as long as the total utilization is equal to or less than 1.

2. The schedulable utilization $U_{EDF}(n)$ of the EDF algorithm for n independent, preemptable periodic tasks with relative deadlines equal to or larger than their periods is equal to 1.

Theorem 2: A system T of independent, preemptable tasks can be feasibly scheduled on one processor if its density is equal to or less than 1.

The condition given by this theorem is not necessary for a system to be feasible. A system may nevertheless be feasible when its density is greater than 1. E.g. A system consisting of tasks $T_1 = (2, 0.6, 1)$ and $(5, 2.3)$. Its density is

$$\sum \frac{e_i}{\min(p_i, D_i)} = \frac{0.6}{1} + \frac{2.3}{5} = 1.06 > 1$$

Schedulability Test for the EDF Algorithm: $\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq 1$

Sufficient but not necessary condition

$p \Rightarrow q$ means "q is necessary for p" because p cannot be true unless q is true. Similary, we say that "p is sufficient for q" because p being true always implies that q is true but p not being true doesn't always imply that q is not true.

A is necessary for B means the falsity of A guarantees the falsity of B. $\neg A \Rightarrow \neg B$ or $B \Rightarrow A$

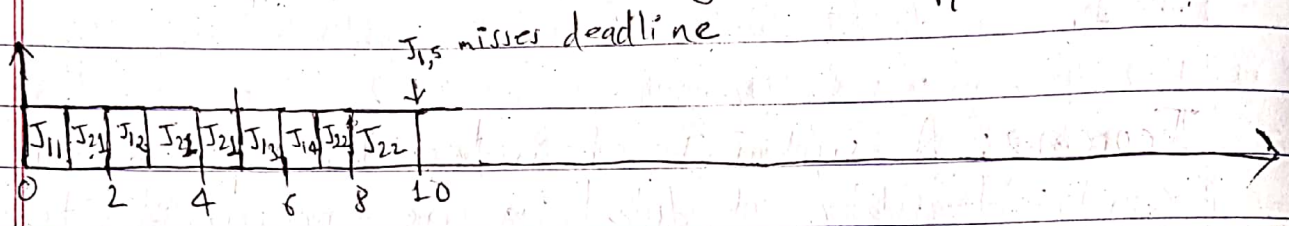
A is sufficient for B means the truth of A guarantees the truth of B. $A \Rightarrow B$ or $\neg B \Rightarrow \neg A$

Scanned by CamScanner

Example of infeasible EDF schedule

Consider a system ~~consti~~ consisting of tasks $T_1 = (2, 1)$ and $T_2 = (5, 3)$

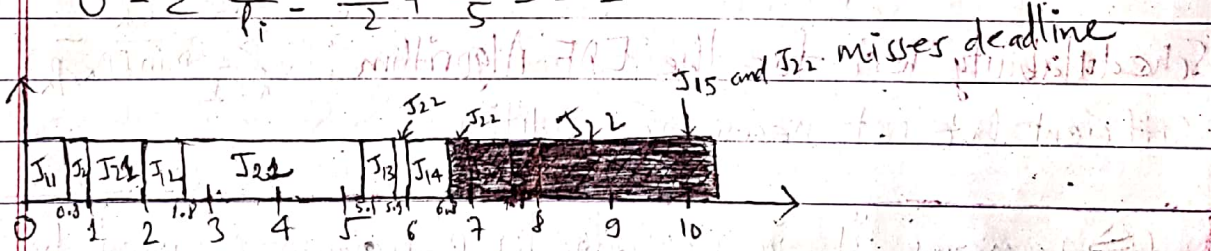
The total utilization of the system $= \sum \frac{e_i}{p_i} = \frac{1}{2} + \frac{3}{5} = 1.1 > 1$



According to EDF schedule as shown in fig, the job $J_{1,5}$ in $T_1 = (2,1)$ is not scheduled until 10 and misses its deadline at 10. Deadlines of all jobs in T_2 are met in the schedule segment.

Example 2: Consider a system consisting of tasks $T_1 = (2, 0.8)$ and $T_2 = (5, 3.5)$

$$U = \sum \frac{e_i}{p_i} = \frac{0.8}{2} + \frac{3.5}{5} = 1.1$$



According to EDF schedule, $J_{1,5}$ in $T_1 = (2, 0.8)$, as well as every job in T_2, cannot complete on time.

Optimality of the RM and DM algorithms

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.

Theorem 1: 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 its total utilization is equal to or less than 1.

Theorem 2:

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

In some systems, the relative deadlines of every task T_i is equal to δp_i for some constant $0<\delta$. For such systems, the DM and RM algorithms are the same.

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

Schedulability test for fixed-priority tasks with short response times

Assumptions:

* Response times of the jobs are smaller than or equal to their respective periods
* total utilization $U \leq 1$

Critical Instants:

A 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 to or less than the relative deadline D_i of T_i, and
* the response time of ~~a job in T_i~~ the job released at the instant is greater than D_i, if the response time of some jobs in T_i exceeds D_i

Scanned by CamScanner

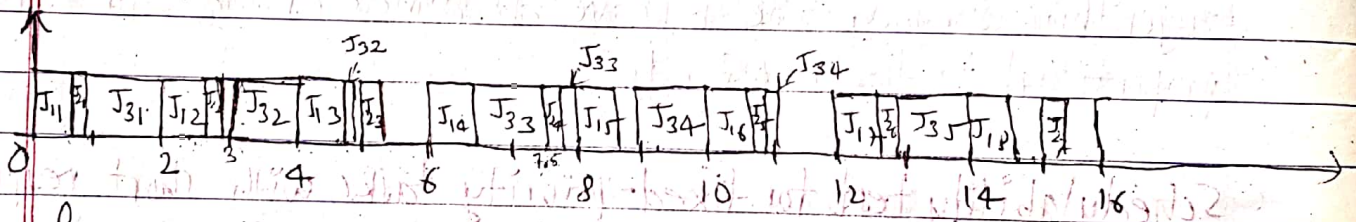
The response time of a job in T_i released at a critical instant the maximum (possible) response time of the task and is denoted by W_i

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, i.e., $r_{i,c} = r_{k,l_k}$ for some l_k for every $k = 1, 2, \ldots, i-1$.

Example:

$T_1 = (2, 0.6)$, $T_2 = (2.5, 0.2)$ and $T_3 = (3, 1.2)$

According to RM T_1 has the highest priority and T_3 has the lowest priority.



Response time of $J_{11} = 0.6 - 0 = 0.6$

Response time of $J_{12} = 2.6 - 2 = 0.6$

Response time of all jobs in $T_1 = 0.6$

Response time of $J_{21} = 0.8 - 0 = 0.8$

Response time of $J_{22} = 2.8 - 2.5 = 0.3$

Response time of $J_{23} = 5.2 - 5 = 0.2$

Response time of $J_{24} = 7.7 - 7.5 = 0.2$

Response time of $J_{25} = 10.8 - 10 = 0.8$

Response time of $J_{31} = 2 - 0 = 2$

Response time of $J_{32} = 4.8 - 3 = 0.8$

Response time of $J_{33} = 8 - 6 = 2$

Critical instant for T_1, T_2, $T_3 = 0$

Example 2: Phase of task with period 2.5 is one and the phases of other tasks are 0.

Date
Page

Schedulability test for fixed-priority tasks with arbitrary response times

General time demand analysis is used to determine the schedulability of tasks whose relative deadlines are larger than their respective periods. Since the response time of a task may be larger than its period, it may have more than one job ready for execution at any time. Ready jobs in the same task are usually scheduled on the FIFO basis.

Busy Intervals:

notation used in chapter 6 →

We index the tasks in decreasing order of their priorities except where stated otherwise, i.e. T_i has a higher priority than the task T_k if $i<k$. We refer to the priority of a task T_i as priority π_i, π's are positive integers $1, 2, \ldots, n$, 1 being the highest priority.
We denote the subset of tasks with equal or higher priority than T_i by $\mathbf{T}_i$ and its total utilization by $U_i = \sum_{k=1}^{i} u_k$

A level-π_i busy interval $(t_0, t]$ begins at an instant t_0 when
(1) all jobs in $\mathbf{T}_i$ released before the instant have completed and
(2) a job in $\mathbf{T}_i$ is released. ~~The i~~
The interval ends at the first instant t after t_0 when all the jobs in $\mathbf{T}_i$ released since t_0 are complete.

In other words, in the interval $(t_0, t]$, the processor is busy all the time executing jobs with priorities π_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.

While computing the response times of jobs in $\mathbf{T}_i$, we consider every level-π_i busy interval independently from other level-π_i busy intervals.

Scanned by CamScanner

A level-π_i busy interval is in phase if the first jobs all tasks that have priorities equal to or higher than the priority π_i and are executed in this interval have the same release time. Otherwise, the tasks have arbitrary phases in the interval.

Example:

The below fig. shows the schedule of three tasks $T_1 = (2, 1)$, $T_2 = (3, 1.25)$ and $T_3 = (5, 0.25)$ in the first hyperperiod. The filled rectangles depict where jobs in T_1 are scheduled.

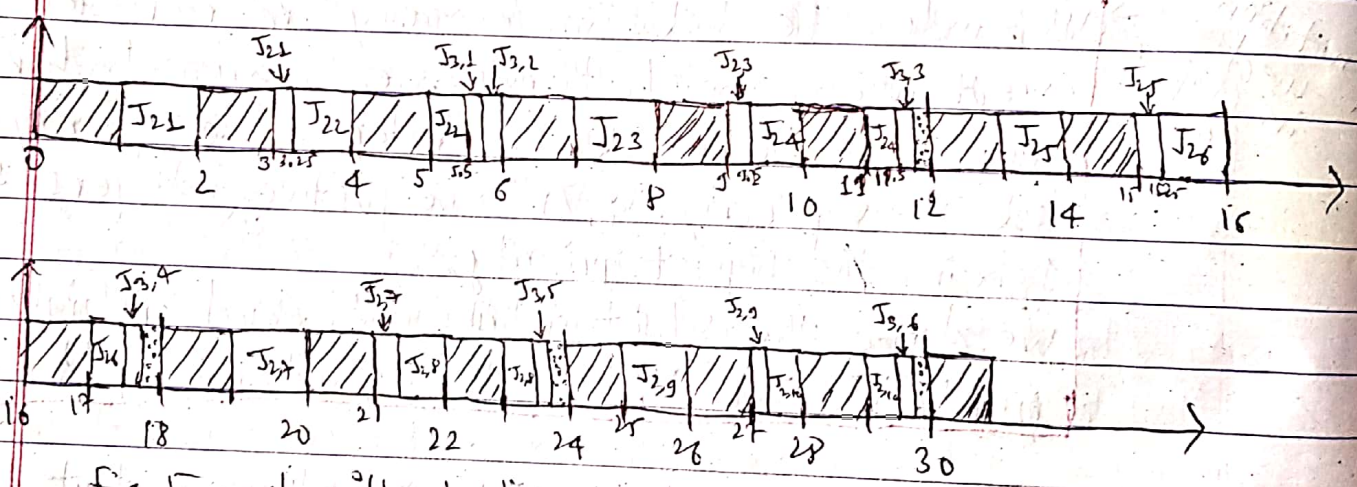


fig. Example illustrating busy intervals.

The first busy intervals of all levels are in phase. The priorities of the tasks are $\pi_1 = 1$, $\pi_2 = 2$ and $\pi_3 = 3$, with 1 being the highest priority and 3 being the lowest priority. As expected, every level-1 busy interval always ends 1 unit time after it begins.

In this system, all the level-2 busy intervals are in phase. They begin at times 0, 6 and so on, which are the least common multiples of the periods of tasks T_1 and T_2. The length of these intervals are equal to 5.5.

Before time 5.5, there is at least one job of priority 1 or 2

ready for execution, but immediately after 5.5, there are none. Hence at 5.5, the first job in T_3 is scheduled. When this job completes at 5.75, the second job in T_3 is scheduled. At time 6, all the jobs released before time 6 are completed; hence the first level-3 busy interval ends at this time.

The second level-3 busy interval begins at time 6. This level-3 busy interval is not in phase since the release times of the first higher-priority jobs in this interval are 6, but the first job of T_3 in this interval is not released until time 10. The length of this level-3 busy interval is only ~~5.75 - 10.25 = 6~~ $11.75 - 6 = 5.75$

Similarly, all the subsequent level-3 busy intervals in the hyperperiod have arbitrary phases.

General Schedulability Test

It relies on the fact that when determining the schedulability of a task T_i in a system in which the response times of jobs can be larger than their respective periods. However, the first job $J_{i,1}$ may no longer have the largest response time among all jobs in T_i. Consequently, we must examine all the jobs of T_i that are executed in the first level-π_i busy interval. (This busy interval is in phase when the tasks are in phase).

If the response times of all these jobs are no greater than the relative deadline of T_i, T_i is schedulable; otherwise, T_i may not be schedulable.

General Time-Demand Analysis Method

Test one task at a time starting from the highest priority task T_1 in order of decreasing priority. For the purpose of

Scanned by CamScanner

determining whether a task T_i is schedulable, assume that all the tasks are in phase and the first level-π_i busy interval begins at time 0.

While testing whether all the jobs in T_i can meet their deadlines (i.e., whether T_i is schedulable), consider the subset $\mathcal{T}_i$ of tasks with priorities π_i or higher.

(i) If the first job of every task in $\mathcal{T}_i$ completes by the end of the first period of the task, check whether the first job $J_{i,1}$ in T_i meets its deadline. T_i is schedulable if $J_{i,1}$ completes in time. Otherwise, T_i is not schedulable.

(ii) If the first job of some task in $\mathcal{T}_i$ does not complete by the end of the first period of the task, do the following:

(a) Compute the length of the in-phase level-π_i busy interval by solving the equation $t = \sum_{k=1}^{i} \left\lceil \frac{t}{p_k} \right\rceil e_k$ iteratively, starting from $t^{(1)} = \sum_{k=1}^{i} e_k$ until $t^{(l+1)} = t^{(l)}$ for some $l \geq 1$. The solution $t^{(l)}$ is the length of the level-π_i busy interval.

(b) Compute the maximum response time of all $\left\lceil \frac{t^{(l)}}{p_i} \right\rceil$ jobs of T_i in-phase level-π_i busy interval in the manner described below and determine whether they complete in time.

T_i is schedulable if all these jobs complete in time; otherwise T_i is not schedulable.

The maximum response time of the first job $J_{i,1}$ of T_i in the first in-phase level-π_i busy interval is given by

$$w_{i,1}(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k \quad \text{for } 0 < t \leq w_{i,1}(t)$$

To obtain $w_{i,1}$, we solve the equation iteratively and terminate the iteration only when we find $t^{(l+1)}$ equal to $t^{(l)}$.

The max^m possible response time $w_{i,1}$ of $J_{i,1}$ is equal to the smallest value of t that satisfies the equation $t = w_{i,1}(t)$.

Scanned by CamScanner

Date
Page

Lemma: The maximum response time $w_{i,j}$ of the jth job of T_i in an in-phase level-π_i busy period is equal to the smallest value of t that satisfies the equation,

$$t = w_{i,j}(t + (j-1)p_i) - (j-1)p_i$$

where $w_{i,j}(t)$ is given by

$$w_{i,j}(t) = je_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \quad \text{for } (j-1)p_i < t \le w_{i,j}(t)$$

For example:

$T_1 = (2, 1)$, $T_2 = (3, 1.25)$ and $T_3 = (5, 0.25)$

$\pi_1 = 1$, $\pi_2 = 2$, $\pi_3 = 3$

Length of first level-π_1 busy interval $= e_1 = 1$ [by default; no need to calculate]

<u>Length of first level-π_2 busy interval</u>

$$t^{(0)} = \sum_{k=1}^{2} \left\lceil \frac{t}{p_k} \right\rceil e_k = \left\lceil \frac{t}{2} \right\rceil e_1 + \left\lceil \frac{t}{3} \right\rceil e_2 = \left\lceil \frac{t}{2} \right\rceil \times 1 + \left\lceil \frac{t}{3} \right\rceil \times 1.25$$

$$t^{(1)} = \sum_{k=1}^{2} e_k = e_1 + e_2 = 1 + 1.25 = 2.25$$

$$t^{(2)} = \left\lceil \frac{2.25}{2} \right\rceil \times 1 + \left\lceil \frac{2.25}{3} \right\rceil \times 1.25$$

$$= \cancel{1(1+1.25) = 2.25} \; 2 + 1.25 = 3.25$$

$$t^{(3)} = \left\lceil \frac{3.25}{2} \right\rceil \times 1 + \left\lceil \frac{3.25}{3} \right\rceil \times 1.25$$

$$= 2 + 2 \times 1.25 = 4.5$$

$$t^{(4)} = \left\lceil \frac{4.5}{2} \right\rceil \times 1 + \left\lceil \frac{4.5}{3} \right\rceil \times 1.25 = 3 + 2 \times 1.25 = 5.5$$

$$t^{(5)} = \left\lceil \frac{5.5}{2} \right\rceil \times 1 + \left\lceil \frac{5.5}{3} \right\rceil \times 1.25 = 3 + 2 \times 1.25 = 5.5$$

$\therefore$ Length of first level-π_2 busy interval $= 5.5$

<u>For first level-π_3 busy interval</u>

$$t^{(0)} = \sum_{k=1}^{3} \left\lceil \frac{t}{p_k} \right\rceil e_k = \left\lceil \frac{t}{2} \right\rceil \times 1 + \left\lceil \frac{t}{3} \right\rceil \times 1.25 + \left\lceil \frac{t}{5} \right\rceil \times 0.25$$

Scanned by CamScanner

$$t^{(1)} = \sum_{k=1}^{3} e_k = 1 + 1.25 + 0.25 = 2.5$$

$$t^{(2)} = \left\lceil \frac{2.5}{2} \right\rceil \times 1 + \left\lceil \frac{2.5}{3} \right\rceil \times 1.25 + \left\lceil \frac{2.5}{5} \right\rceil \times 0.25$$

$$= 2 + 1.25 + 0.25 = 3.5$$

$$t^{(3)} = \left\lceil \frac{3.5}{2} \right\rceil \times 1 + \left\lceil \frac{3.5}{3} \right\rceil \times 1.25 + \left\lceil \frac{3.5}{5} \right\rceil \times 0.25$$

$$= 2 + 2 \times 1.25 + 1 \times 0.25 = 4.75$$

$$t^{(4)} = \left\lceil \frac{4.75}{2} \right\rceil \times 1 + \left\lceil \frac{4.75}{3} \right\rceil \times 1.25 + \left\lceil \frac{4.75}{5} \right\rceil \times 0.25$$

$$= 3 + 2 \times 1.25 + 1 \times 0.25 = 5.75$$

$$t^{(5)} = \left\lceil \frac{5.75}{2} \right\rceil + \left\lceil \frac{5.75}{3} \right\rceil \times 1.25 + \left\lceil \frac{5.75}{5} \right\rceil \times 0.25$$

$$= 3 + 2 \times 1.25 + 2 \times 0.25 = 6$$

$$t^{(6)} = \left\lceil \frac{6}{2} \right\rceil + \left\lceil \frac{6}{3} \right\rceil \times 1.25 + \left\lceil \frac{6}{5} \right\rceil \times 0.25$$

$$= 3 + 2 \times 1.25 + 2 \times 0.25 = 6$$

$\therefore$ length of first Π_3-level busy interval $= 6$

<u>For maximum response times,</u>

For job $J_{1,1}$:

$$t^{(1)} = e_1 = 1$$

$$t^{(2)} = e_1 + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k = e_1 = 1$$

For job $J_{2,1}$:

$$t^{(1)} = e_2 = 1.25$$

$$t^{(2)} = e_2 + \left\lceil \frac{1.25}{2} \right\rceil \times 1 = 1.25 + 1 = 2.25$$

$$t^{(3)} = 1.25 + \left\lceil \frac{2.25}{2} \right\rceil \times 1 = 1.25 + 2 = 3.25$$

$$t^{(4)} = 1.25 + \left\lceil \frac{3.25}{2} \right\rceil \times 1 = 1.25 + 2 \times 1 = 3.25$$

$$\therefore w_{2,1} = 3.25$$

Scanned by CamScanner

For job $J_{3,1}$:

$t^{(1)} = e_3 = 0.25$

$t^{(2)} = 0.25 + \left\lceil \frac{0.25}{2} \right\rceil \times 1 + \left\lceil \frac{0.25}{3} \right\rceil \times 1.25$

$= 0.25 + 1 + 1.25 = 2.5$

$t^{(3)} = 0.25 + \left\lceil \frac{2.5}{2} \right\rceil \times 1 + \left\lceil \frac{2.5}{3} \right\rceil \times 1.25$

$= 0.25 + 2 + 1.25 = 3.5$

$t^{(4)} = 0.25 + \left\lceil \frac{3.5}{2} \right\rceil \times 1 + \left\lceil \frac{3.5}{3} \right\rceil \times 1.25$

$= 0.25 + 2 + 2 \times 1.25 = 4.75$

$t^{(5)} = 0.25 + \left\lceil \frac{4.75}{2} \right\rceil \times 1 + \left\lceil \frac{4.75}{3} \right\rceil \times 1.25$

$= 0.25 + 3 \times 1 + 2 \times 1.25 = 5.75$

$t^{(6)} = 0.25 + \left\lceil \frac{5.75}{2} \right\rceil \times 1 + \left\lceil \frac{5.75}{3} \right\rceil \times 1.25$

$= 0.25 + 3 + 2 \times 1.25 = 5.75$

$\therefore w_{3,1} = 5.75$

For job $J_{3,2}$,

$t^{(1)} = e_2 = 1.25$

$\cancel{w_{2,2}(1.25) = 2 \times 1.25 + \left\lceil \frac{1.25}{2} \right\rceil \times 1 = 3.5}$

$\cancel{t^{(2)} = w_{2,2}(1.25 + 2) = w(3.25) = 2 \times 1.25 + \left\lceil \frac{3.25}{2} \right\rceil \times 1 = 4.5}$

$\cancel{t^{(2)} = 4.5 - 1 \times 2}$

$w_{2,2}(1.25 + 1 \times 3) = w_{2,2}(4.25) = 2 \times 1.25 + \left\lceil \frac{4.25}{2} \right\rceil \times 1 = 5.5$

$t^{(2)} = 5.5 - 1 \times 3 = 2.5$

$w_{2,2}(2.5 + 1 \times 3) = w(5.5) = 2 \times 1.25 + \left\lceil \frac{5.5}{2} \right\rceil \times 1 = 5.5$

$t^{(3)} = 5.5 - 1 \times 3 = 2.5$

$\therefore w_{2,2} = 2.5$

Scanned by CamScanner

Date
Page

For the job $J_{3,2}$:

$$t^{(1)} = e_3 = 0.25$$

$$(j-1)p_j = (2-1)\times 5 = 5$$

$$t = w_{3,2}(t+5) - 5$$

$$w_{3,2}(t) = 2\times e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2 = 2\times 0.25 + \left\lceil \frac{t}{2} \right\rceil \times 1 + \left\lceil \frac{t}{3} \right\rceil \times 1.25$$

$$\therefore w_{3,2}(t) = 0.5 + \left\lceil \frac{t}{2} \right\rceil + \left\lceil \frac{t}{3} \right\rceil \times 1.25$$

$$t^{(2)} = 0.5 + \left\lceil \frac{0.25+5}{2} \right\rceil + \left\lceil \frac{0.25+5}{3} \right\rceil \times 1.25 - 5 = 0.5 + 3 + 2.5 - 5 = 1$$

$$t^{(3)} = 0.5 + \left\lceil \frac{1+5}{2} \right\rceil + \left\lceil \frac{1+5}{3} \right\rceil \times 1.25 - 5 = 1$$

$$\therefore w_{3,2} = 1$$

Finally to determine whether the jth job in an in-phase level-π_i busy interval completes in time, we can check whether the inequality $w_{i,j}(t) \le t$ is ever satisfied for some instant t in the range from $(j-1)p_i$ to $(j-1)p_i + D_i$

For arbitrary phase ϕ_k: maximum response time = smallest value of t

$$t = je_i + \sum_{k=1}^{i-1} \left\lceil \frac{t+\phi_i-\phi_k}{p_k} \right\rceil e_k - \phi_i - (j-1)p_i$$

Scanned by CamScanner

Sufficient Schedulability Conditions for the RM and DM algorithms

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

$$U_{RM}(n) = n(2^{1/n} - 1)$$

Specifically, $U(n) \leq U_{RM}(n)$ is a sufficient schedulability condition for any system of n independent, preemptable tasks that have relative deadlines equal to their respective periods to be schedulable rate-monotonically.

E.g.

Tasks	Utilization
$T_1 = (1, 0.25)$	0.25
$T_2 = (1.25, 0.1)$	0.08
$T_3 = (1.5, 0.3)$	0.20
$T_4 = (1.75, 0.07)$	0.04
$T_5 = (2, 0.1)$	0.05
Total	0.62

Here, $n = 5$

$U_{RM}(n) = n(2^{1/n} - 1) = 0.743$

Total utilization $U(n) = 0.62$

Since $0.62 < 0.743$, these tasks can be feasibly scheduled according to the RM algorithm.

Scanned by CamScanner

Practical Factors

We have assumed –

- every job is preemptable at any time
- a job never suspends itself after its release
- a job is always ready for execution until it completes.
- the scheduler is event-driven and acts immediately upon event occurences.

These assumptions are often not valid in real life systems

Practical Factors

1. Non-preemptability
   - Causes Blocking and Priority Inversion
2. Self-Suspension
   - While executing a job may invoke some external operation, for eg., an I/O operation that is executed on another processor.
   - Self-suspension occurs when the job is suspended and waits until such an operation completes before its execution can continue.
3. Context Switches:
   - Process of storing and restoring the state of a job/task so that execution can be resumed from the same point at a later time.
4. Tick Scheduling:
   - In all scheduling tests we have seen, an underlying assumption is that the scheduler is event driven and it executes immediately upon the occurence of every scheduling event.
   - This assumption is sometimes not valid.
   - Sometimes the scheduler may be time-driven and scheduling decisions are made at time instants.
   - This method is called tick-scheduling.

Scanned by CamScanner

Chapter 7

Scheduling Aperiodic and Sporadic Jobs in Priority-Driven Systems

Assumptions and Approaches

- There is only one processor
- The period tasks in the system are independent.
- Aperiodic and sporadic jobs are also independent of each other and of the periodic tasks.
- Every job can be preempted at any time.
- Parameters of each sporadic job become known after it is released.
- Sporadic jobs may arrive at any instant, even immediately after each other.
- Their execution times may vary widely, and their deadlines are arbitrary.

Objectives

We assume that we are given the parameters $\{p_i\}$ and $\{e_i\}$ of all the periodic tasks and a priority-driven algorithm used to schedule them. Moreover, when the periodic tasks are scheduled according to the given algorithm and there are no aperiodic and sporadic jobs, the periodic tasks meet all their deadlines.

The operating system maintains three queues; ~~sporadic~~ sporadic jobs queue, aperiodic jobs queue and periodic job queue.

The aperiodic job and sporadic job scheduling algorithms are solutions to the following problems:

1. Based on the execution time and deadline of each newly arrived sporadic job, the scheduler decides whether to accept or reject the job. If it accepts the job, it schedules the job so that the job completes in time without causing periodic tasks and previously accepted sporadic jobs to miss their deadlines. The problems are how to do the acceptance test and how to schedule the accepted sporadic jobs.
2. The scheduler tries to complete each aperiodic job as soon as possible.

Scanned by CamScanner

The problem is how to do so without causing periodic tasks and accepted sporadic jobs to miss their deadlines.

Correctness

- An aperiodic job or sporadic job scheduling algorithm is correct if it produces only correct schedules of the system.
- A correct schedule means one according to which periodic and accepted sporadic jobs never miss their deadlines.

Optimality

- 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 queueing discipline.
- An algorithm for (accepting and) scheduling sporadic jobs is optimal if it accepts each sporadic job newly offered to the system and schedules the job to complete in time if and only if the new job can be correctly scheduled to complete in time by some means.

Commonly used methods/ approaches for scheduling aperiodic jobs

- Background Approach • Polled Approach • Interrupt driven executions

Background Approach

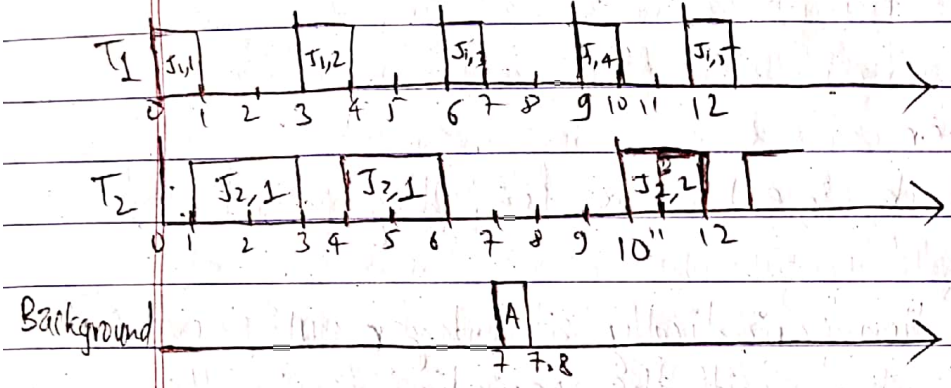
- Aperiodic jobs are scheduled and executed only at times when there is no periodic or sporadic job ready for execution.
- This method always produces correct schedules and is simple to implement.
- However, the execution of aperiodic jobs may be delayed and their response times prolonged unnecessarily.

Scanned by CamScanner

Date
Page

Example:

Consider the system of two periodic tasks $T_1 = (3,1)$ and $T_2 = (10,4)$. The tasks are scheduled rate monotically. Suppose that an aperiodic job A with execution time equal to 0.8 is released at time 0.1.



The execution of A begins at 7 after $J_{1,3}$ completes. Hence its response time = 7.8 − 0.1 = 7.7

Interrupt-Driven Execution

- A way to make the response time of aperiodic jobs as short as possible is to make their execution interrupt driven.
- Whenever an aperiodic job arrives, the execution of periodic tasks are interrupted, and the aperiodic job is executed.
- Problem: If aperiodic job always execute as soon as possible, periodic tasks may miss some deadlines.
- E.g. In the above example, if the execution time of A were equal to 2.1, both $J_{1,1}$ and $J_{2,1}$ would miss their deadlines.

Slack Stealing:

The solution to above problem is to postone the execution of periodic tasks only when it is safe to do so. In the example above, we see that the execution of $J_{1,1}$ and $J_{2,1}$ can be postponed by 2 units of time because they both have this much slack.

Scanned by CamScanner

- By postponing the execution of $J_{1,1}$ and $J_{2,1}$, the scheduler allows aperiodic job A to execute immediately after its arrival.
- The response time of A is equal to 0.8, which is as short as possible.
- The periodic job $J_{1,1}$ completes at time 1.8, and $J_{2,1}$ completes at 7.8

Poller/Polling server

- It is a periodic task (p_s, e_s), where p_s = polling period and e_s = execution time.
- It is ready for execution periodically at integer multiples of p_s and is scheduled together with the periodic tasks in the system according to the given priority-driven algorithm.
- When it executes, ~~the~~ it examines the aperiodic job queue.
- If the queue is non-empty, the poller executes the job at the head of the queue.
- The poller suspends its execution or is suspended by the scheduler either when it ~~or~~ has executed for e_s units of time in the period or when the aperiodic job queue becomes empty, whichever occurs sooner.
- It is ready for execution again at the beginning of the next polling period (budget is replenished).
- On the other hand, if at the beginning of a polling period the poller finds the aperiodic job queue empty, it suspends immediately. It will not be ready for execution and able to examine the queue again until the next polling period.
- Poller is given the highest priority among periodic tasks.
- The ratio $u_s = e_s/p_s$ is the size of the server.

Scanned by CamScanner

Periodic Server

- A task that behaves more or less like a periodic task and that is created for the purpose of executing aperiodic jobs is called a period server.
- A periodic server (p_s, e_s) is defined partially by its period p_s and execution time e_s (execution budget / budget).
- A poller is a kind of periodic server.
- We say that the periodic server is backlogged whenever the aperiodic job queue is non-empty and, hence, there is at least an aperiodic job to be executed by the server.
- The server is eligible (i.e. ready) for execution only when it it is backlogged and has budget (i.e. its budget is non-zero).

Shortcoming / Limitations of Poller

- An aperiodic job that arrives after the aperiodic job queue is examined and found empty must wait for the poller to return to examine the queue again a polling period later.
- It does not preserve bandwidth.

E.g. $T_1 = (3,1)$, $T_2 = (10,4)$, Poller $= (2.5, 0.5)$ and aperiodic job with execution time 0.8 arrived at time 0.1. Poller has the highest priority.

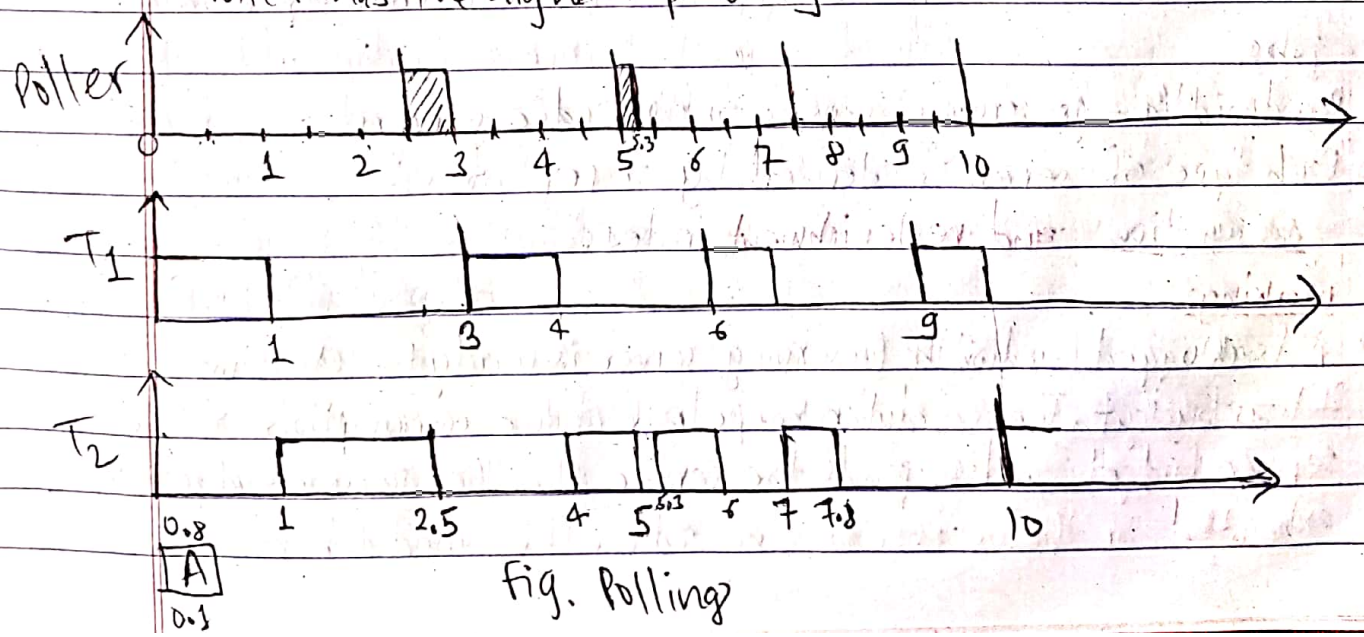


fig. Polling

Scanned by CamScanner

At the beginning of the first polling period, the poller's budget is replenished, but when it executes, it finds the aperiodic job queue empty. Its execution budget is consumed instantaneously, and its execution suspended immediately. The aperiodic job A arrives a short time later and must wait in the queue until the beginning of the second polling period when the poller's budget is replenished. The poller finds A at head of the queue at time 2.5 and executes the job until its execution budget is exhausted at time 3.0. Job A remains in the aperiodic job queue and is again executed when the execution budget of the poller is replenished at 5.0. The job completes at time 5.3, with a response time of 5.3 − 0.1 = 5.2. Since, the aperiodic job queue is empty at 5.3, the budget of the poller is exhausted and the poller suspends.

Bandwidth-Preserving Server

- The server that preserves the execution budget of the poller when it finds an empty queue and allows it to execute later in the period if any aperiodic job arrives, is called bandwidth preserving server.
- These servers shorten the response times of some aperiodic jobs.
- Bandwidth-preserving servers are periodic servers.
- Each type of server is defined by a set of consumption and replenishment rules.

Working:

- A backlogged bandwidth-preserving server is ready for execution when it has budget. The scheduler keeps track of the consumption of the server budget and suspends the server when the server budget is exhausted or the server becomes idle. The scheduler moves the

Scanned by CamScanner

server back to the ready queue once it replenishes the server budget if the server is still backlogged at the time.

- The server ~~suffe~~ suspends itself whenever it finds the aperiodic job queue empty, that is, when it becomes idle. When the server becomes backlogged again upon of arrival of an aperiodic job, the scheduler puts the server back to the ready queue if the server has budget at the time.

Examples: Deferrable Servers, Sporadic Servers. and, constant utilization/total bandwidth servers. and weighted fair-queueing servers.

Deferrable Servers

- A deferrable server is the simplest of bandwidth-preserving servers.
- Like a poller, the execution budget of a deferrable server with period P_s and execution budget e_s is replenished periodically with period P_s.
- Unlike a poller, however, when a deferrable server finds no aperiodic job ready for execution, it preserves its budget.

Operations of Deferrable Servers

The consumption and replenishment rules that define a deferrable server (P_s, e_s) are as follows:

- Consumption Rule:
  The execution budget of the server is consumed at the rate of one per unit time whenever the server executes.
- Replenishment Rule:
  The execution budget of the server is set to e_s at time instants kP_k, for $k = 0, 1, 2, \ldots$

The server is not allowed to accumulate its budget from period to period. Any budget held by the server immediately before each replenishment time is lost.

Scanned by CamScanner

Example:

Consider $T_1 = (2, 3.5, 1.5)$, $T_2 = (6.5, 0.5)$ and the

phase p_1 e_1

deferrable server $T_{DS} = (3, 1)$

(a) RM schedule

T_{DS} has the highest priority, and T_2 has the lowest priority

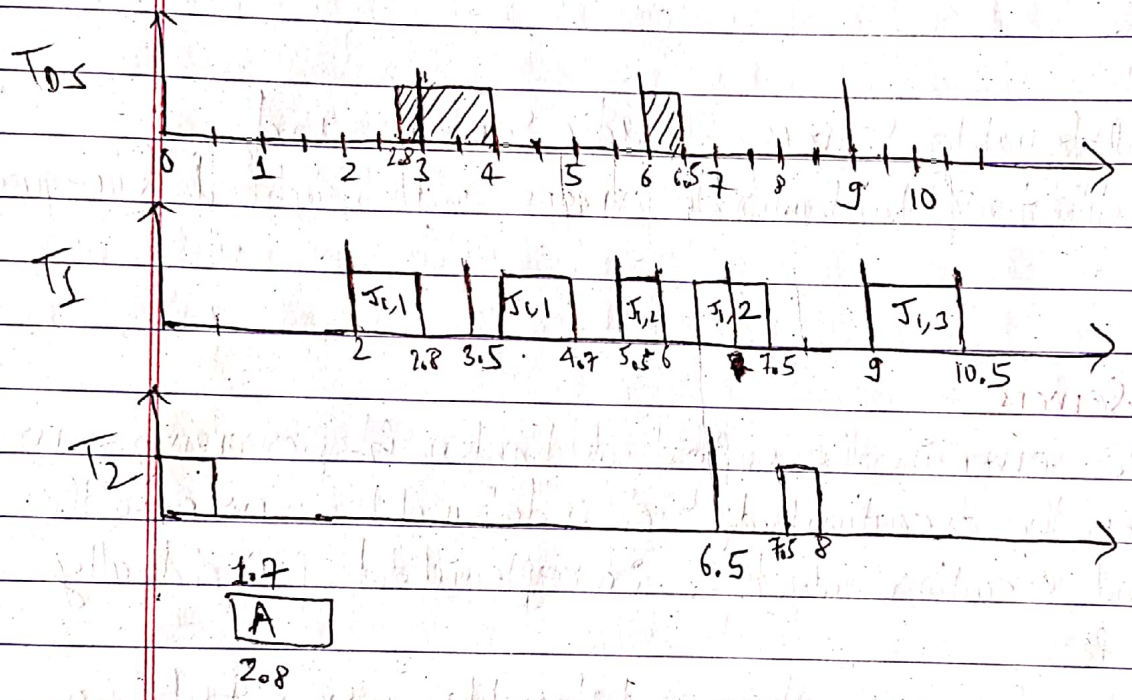


fig. Operation of a deferrable server using RM schedule

- At time 0, the server is given 1 unit of budget. The budget stays at 1 ~~unit time~~ until time 2.8. When A arrives, the deferrable server executes the job. Its budget decreases as it executes.
- Immediately before the replenishment time 3.0, its budget is equal to 0.8. This 0.8 unit is lost at time 3.0, but the server acquires a new unit of budget. Hence, the server continues to execute.
- At time 4.0, its budget is exhausted. The server is suspended and the aperiodic job A waits.

Scanned by CamScanner

- At time 6.0, its budget is replenished, the server resumes to execute A.
- At time 6.5, job A completes. The server still has 0.5 unit of budget. Since, no aperiodic job waits in the queue, the server suspends holding this budget.

(b) According to EDF

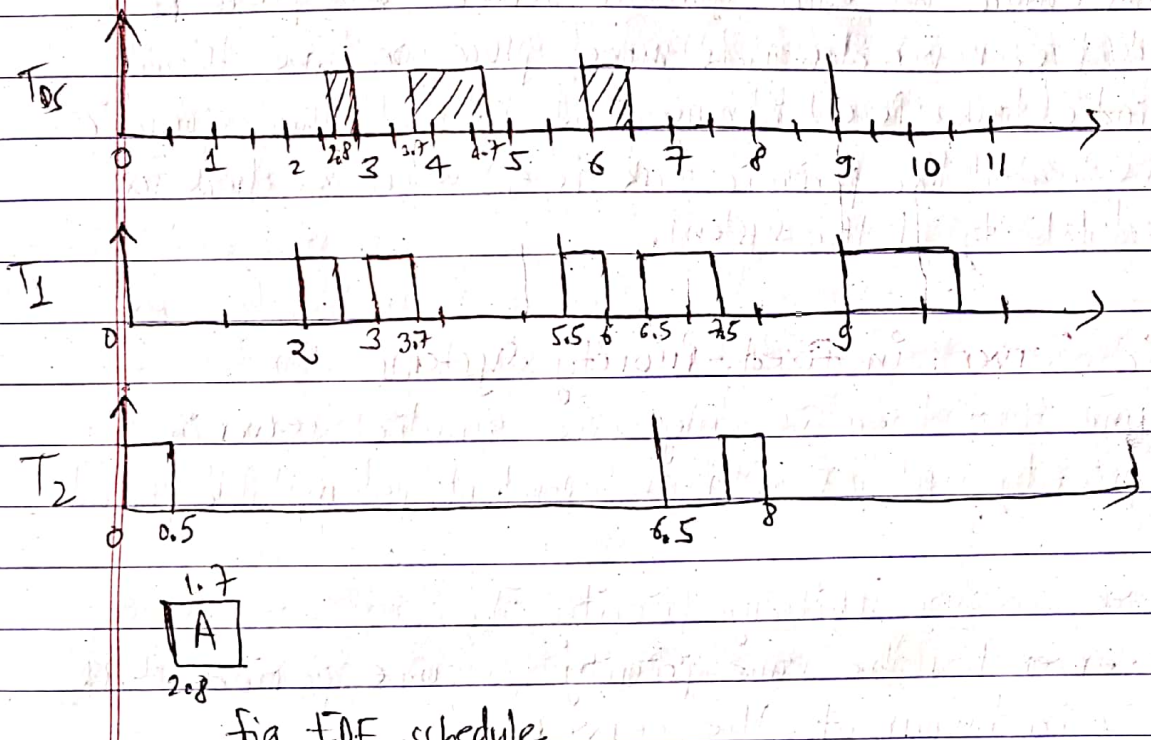


fig. EDF schedule

- At time 2.8, the deadline of the deferrable server is 3.0. Consequently, the deferrable server executes at the highest-priority beginning at this time.
- At time 3.0, when the budget of the deferrable server is replenished, its deadline for consuming this new unit of budget is 6. Since the deadline of $J_{1,1}$ is sooner, ~~then~~ this job has a higher priority. The deferrable server is preempted.
- At time 3.7, $J_{1,1}$ completes. The deferrable server executes until time 4.7 when its budget is exhausted.

Scanned by CamScanner

- At time 6 when the server's budget is replenished, its deadline is 9, which is the same as the deadline of the job $J_{1,2}$. Hence, $J_{1,2}$ would have the same priority as the server. The figure shows that the tie is broken in favour of the server.

Sporadic Servers

- The consumption and replenishment rules of sporadic server algorithms ensure that each sporadic server with period p_s and budget e_s never demands more processor time than the periodic task (p_s, e_s) in any time interval. Thus, sporadic server is treated like periodic task (p_s, e_s) when we check for the schedulability of the system.

Sporadic Server in Fixed-Priority Systems

- We assume that there is only one sporadic server in a fixed-priority system T of n independent, preemptable periodic tasks.
- The server has an arbitrary priority π_s.
- If the server has the same priority as some periodic task, the tie is in favour of the server.
- T_H denotes the subset of periodic tasks that have higher priorities than the server.
- We say the system T of periodic tasks (or the higher-priority subsystem T_H) idles when no job in T (or T_H) is ready for execution; T (or T_H) is busy otherwise.
- A server busy interval is a time interval which begins when an aperiodic job arrives at an empty aperiodic job queue and ends when the queue becomes empty again.

Scanned by CamScanner

t_f मा t_e calculate गरिन्छ

Date
Page

Notations used in Sporadic Server

- t_r denotes the latest (actual) replenishment time.
- t_f denotes the first instant after t_r at which the server begins to execute.
- t_e denotes the latest effective replenishment time.
- At any time t, BEGIN is the beginning instant of the earliest busy interval among the latest contiguous sequence of busy intervals of the higher-priority subsystem T_H that started before t.
- END is the end of the latest busy interval in the latest contiguous sequence of busy intervals of T_H if this interval ends before t and equal to infinity if the interval ends after t.
- The scheduler sets t_r to the current time each time it replenishes the server's execution budget.
- When the server begins to execute after a replenishment, the scheduler determines the latest effective replenishment time t_e and sets the next replenishment time to $t_e + P_s$.

Simple Sporadic Server

← information on BEGIN and END is maintained by monitoring busy intervals of T_H.

- The sporadic server is governed by the following consumption and replenishment rules.
- Consumption Rule for Fixed-Priority Sporadic Server

  At any time t after t_r, the server's execution budget is consumed at the rate of 1 per unit time until the budget is exhausted when either one of the following two conditions is true. When these conditions are not true, the server holds its budget

- C_1 : The server is executing.
- C_2 : The server has executed since t_r and END $< t$; At t server is suspended and the T_H is idle.

Scanned by CamScanner

Date
Page

Replenishment Rules of Simple Fixed-Priority Sporadic Server

R1: Initially, when the system begins execution and each time when the budget is replenished, the execution budget $= e_s$ and $t_r =$ current time.

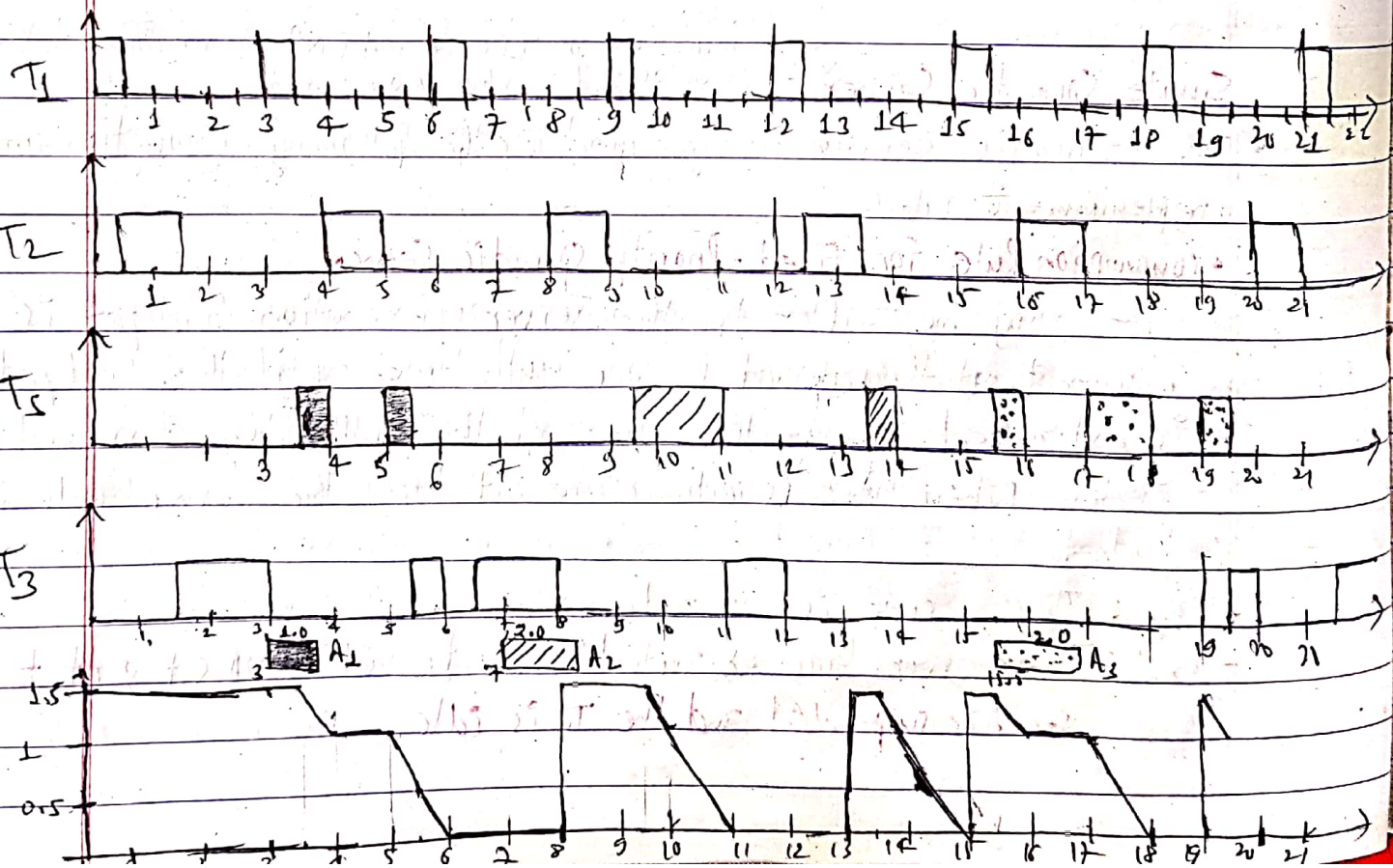
R2: At time t_f, if END $= t_f$, $t_e = \max(t_r, \text{BEGIN})$. If END $< t_f$, $t_e = t_f$. The next replenishment time is set at $t_e + p_s$.

R3: The next replenishment occurs at the next replenishment time except under the following conditions. Under these conditions, replenishment is done at times stated below:

(a) If the next replenishment time $t_e + p_s$ is earlier than t_f, the budget is replenished as soon as it is exhausted.

(b) If the system becomes idle before the next replenishment time $t_e + p_s$ and become busy again at t_b, the budget is replenished at $\min(t_e + p_s, t_b)$.

Example: $T_1 = (3, 0.5)$, $T_2 = (4, 1)$, $T_3 = (19, 4.5)$, $T_5 = (5, 1.5)$



Scanned by CamScanner

19मा Budget आऊनुको कारण (18, 19) idle र 19मा T3 release भयो

Date
Page

1. From time 0 to 3, the aperiodic job queue is empty and the server is suspended. Since, it has not executed, its budget stays at 1.5. At time 3, the aperiodic job A_1 with execution time 1.0 arriver; the server becomes ready. Since the higher-priority task (3, 0.5) has a job ready for execution, the server and the aperiodic job wait.
2. The server does not begin to execute until time 3.5; At the time, t_r is 0, BEGIN is equal to 3 and END is equal to 3.5. According to the rule R_2, the effective replenishment time t_e is equal to $\max(0, 3.0) = 3$, and the next replenishment is set as 8.
3. The server executes until time 4; while it executes, its budget decreases with time.
4. At time 4, the server is preempted by T_2. While it preempted, it holds on to its budget.
5. After the server resumes execution at 5, its budget is consumed until the exhaustion because it first executes (C_1) and then, when it is suspended again, T_1 and T_2 are idle (C_2).
6. When the aperiodic job A_2 arrives at time 7, the budget of the server is exhausted; the job waits in the queue.
7. At time 8, its budget is replenished (R_3), the server is ready for execution again.
8. At time 9.5, the server begins to execute for the first time since 8. t_e is equal to the latest replenishment time 8. Hence the next replenishment time is 13. The server executes until its budget is exhausted at 11; ~~the~~ it is suspended and waits for the next replenishment time. In meantime A_2 waits in the queue.
9. Its budget replenished at time 13, the server is again scheduled and begins to execute at time 13.5. This time, the next replenishment time is set at 18. However at 13.5, the periodic task system T becomes idle. Rather than 18, the budget is replenished at 15, when a new busy interval of T begins, according to the Rule R3b.

$$\min(t_e + t_s, t_b) = \min(18, 15) = 15$$

Scanned by CamScanner

10. The behavior of the later segment also obeys the above stated rules. In particular, rule R3b allows the server budget to be replenished at 19 i.e. $\min(t_e + P_s, t_b) = \min(20, 19) = 19$

Constant Utilization, Total Bandwidth, and Weighted Fair-queueing servers

The three bandwidth preserving server algorithms that offer a simple way to schedule aperiodic jobs in deadline driven system are

(i) Constant Utilization servers ~~algorithm~~

(ii) Total Bandwidth server

(iii) Weighted Fair-Queueing

- They more or less emulate GPS (Generalized Processor Sharing) algorithm.
- GPS, sometimes called fluid-flow processor sharing, is an idealized weighted round robin algorithm

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

Constant Utilization Server Algorithm

The server is defined by its size, which is its instantaneous utilization $\tilde{u}_s$; this ~~function~~ fraction of processor time is reserved for the execution of aperiodic jobs. The server is eligible and ready for execution only when its budget is non-zero. When the server is ready, it is scheduled with the period tasks on the EDF basis.

Scanned by CamScanner

Date
Page

Consumption Rules:

A server consumes its budget only when it executes.

Replenishment Rules for a Constant Utilization Server of size $\tilde{u}_s$:

The budget of a basic constant utilization server is replenished and its deadline set according to the following rules:

R1 Initially, $e_s = 0$ and $d = 0$.

R2 When an aperiodic job with execution time 'e' arrives at time t to an empty aperiodic job queue,

(a) if $t < d$, do nothing

(b) if $t \geq d$, $d = t + \frac{e}{\tilde{u}_s}$, and $e_s = e$

R3 At the deadline d of the server,

(a) if the server is backlogged, set the server deadline to $d + \frac{e}{\tilde{u}_s}$ and $e_s = e$;

(b) if the server is idle, do nothing

Where

$\tilde{u}_s$ = size of the server

e_s = budget of the server

d = deadline of the server

t = the current time

e = execution time of the job at the head of aperiodic job queue.

Scanned by CamScanner

Example: $T_1 = (3, 0.5)$, $T_2 = (4, 1.0)$, $T_3 = (19, 4.5)$

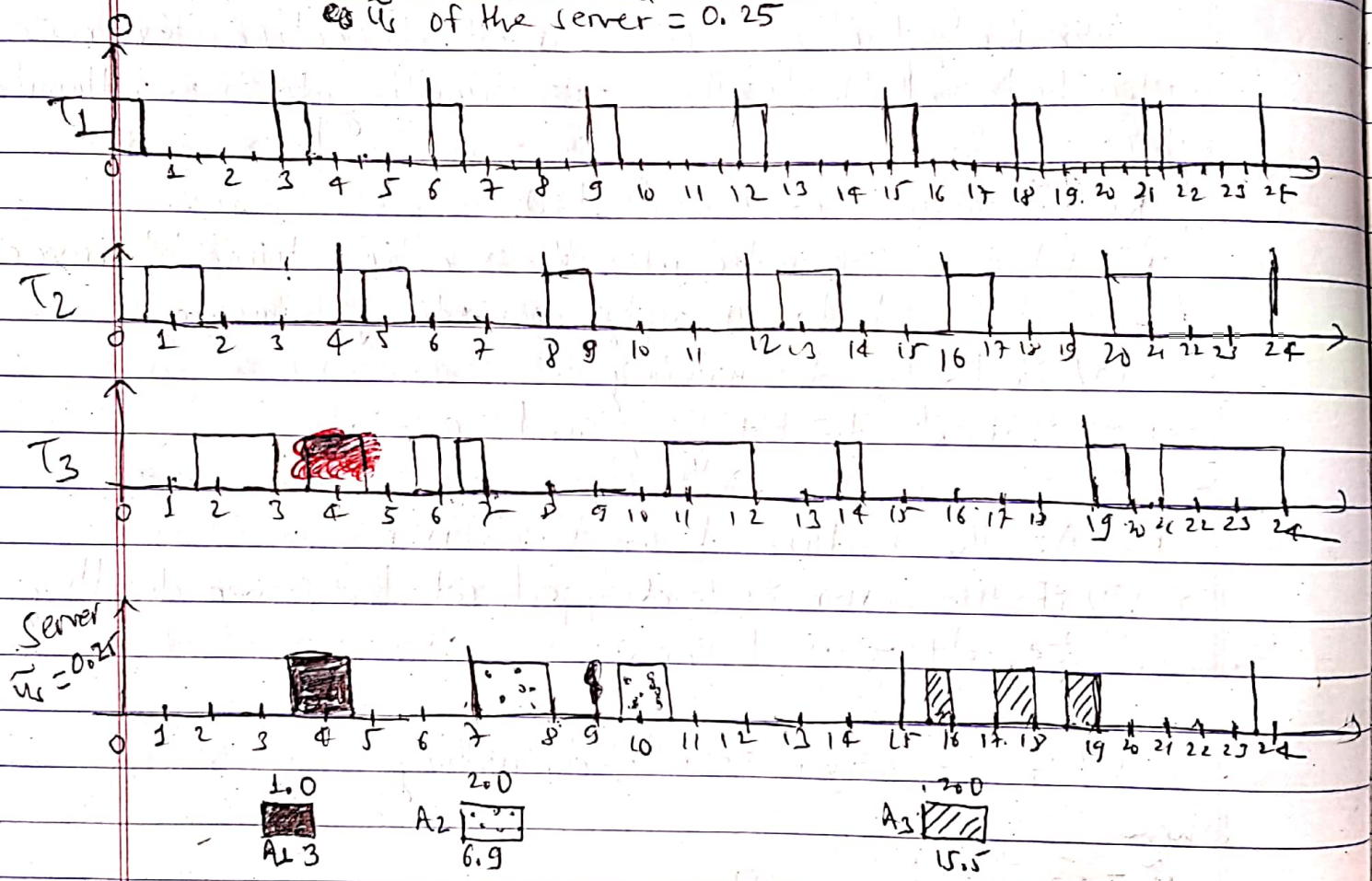
~~$A_1 = (1.0, 3)$, $A_2 = (2.0, 6.9)$~~

A_1 with execution time 1.0 arrives at 3

A_2 with execution time 2.0 arrives at 6.9

A_3 with execution time 2.0 arrives at 15.5

$\tilde{u}_s$ of the server = 0.25



1. Before time 3.0, the budget of the server is 0. Its deadline is 0.
2. At time 3, A_1 arrives. $e_s = e = 1.0$, $d = 3 + \frac{1.0}{0.25} = 7$
   Server is ready for execution. It completes A_1 at 4.5
3. When A_2 arrives at time 6.9, the deadline of the server is later than the current time.
4. At the next deadline of the server at 7, the aperiodic job queue is checked and A_2 is found waiting.
   $e_s = 2.0$, $d = 7 + \frac{2.0}{0.25} = 15$

Scanned by CamScanner

The server is scheduled and executes at time 7, is preempted by τ_2 at time 8, resumes execution at 9.5 and completes A_2 at time 10.5

5. At time 15, the aperiodic job queue is found empty.
6. At time 15.5, A_3 arrives. At the time, the deadline of the server is 15. Hence, $d = 15.5 + \frac{2.0}{0.25} = 23.5$ and $e_s = 2.0$

The server executes A_3 and completes at time 19.

Total Bandwidth Server Algorithm

The total Bandwidth server is defined by its size, which is its instantaneous utilization $\tilde{u}_s$; this fraction of processor time is reserved for the execution of aperiodic jobs.

The total bandwidth server algorithm improves the responsiveness of a constant utilization server by allowing the server to claim the background time not used by periodic tasks. This is done by having the scheduler replenish the server budget as soon as the budget is exhausted if the server is backlogged at the time or as soon as the server becomes backlogged.

Consumption Rule:

A server consumes its budget only when it executes.

Replenishment Rules:

R_1 Initially $e_s = 0$ and $d = 0$

R_2 When an aperiodic job with execution time e arrives at time t to an empty aperiodic job queue, set d to $\max(d, t) + \frac{e}{\tilde{u}_s}$ and $e_s = e$

R_3 When the server completes the current aperiodic job, the job is removed from its queue.

(a) If the server is backlogged, the server ~~idle~~ deadline is

set to $d + \frac{e}{u_s}$ and $e_s = e$.

(b) If the server is idle do nothing

Slack Stealing in Deadline Driven System

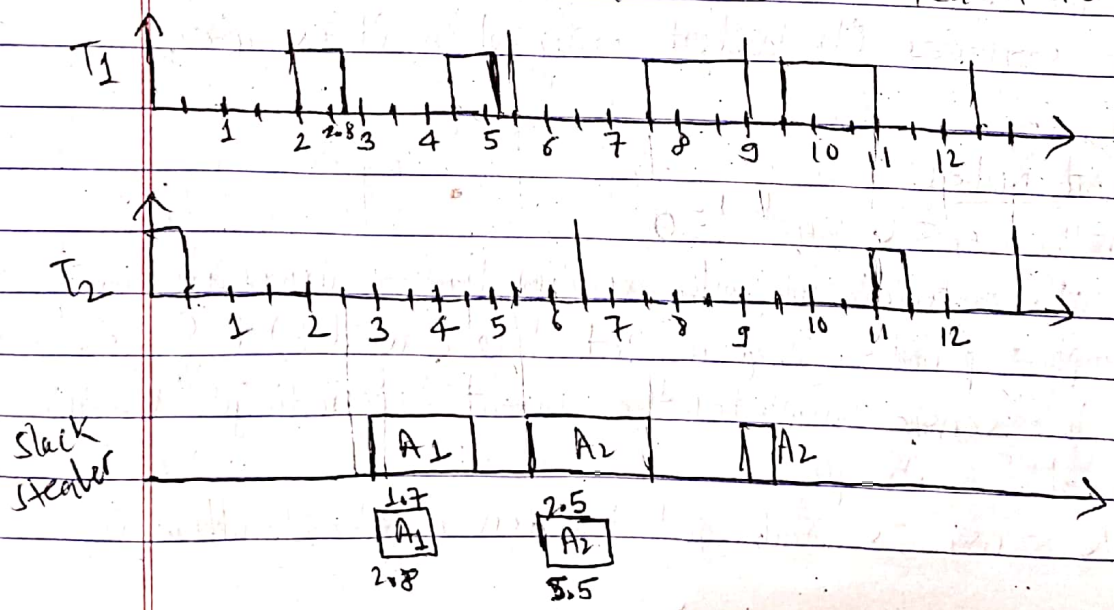
Aperiodic jobs are executed by a slack stealer. The slack stealer is ready for execution whenever the aperiodic job queue is nonempty and is suspended when the queue is empty. The scheduler monitors the periodic tasks in order to keep track of the amount of available slack. It gives the slack stealer the slack stealer the highest priority whenever there is slack and the lowest priority whenever there is no slack.

When the slack stealer executes, it executes the aperiodic job at the head of the aperiodic job queue. This kind of slack stealing algorithm is said to be greedy. The available slack is always used if there is an aperiodic job ready to be executed.

Example: $T_1 = (2.0, 3.5, 1.5)$ and $T_2 = (6.5, 0.5)$

A_1 with execution time 1.7 is released at 2.8

A_2 with execution time 2.5 is released at 5.5



1. Initially, the slack stealer is suspended because the aperiodic job queue is empty. When A_1 arrives at 2.8, the slack stealer resumes. Execution of last 0.7 units of $J_{1,1}$ can be postponed until time 4.8 (i.e. 5.5 − 0.7) and T_2 has no ready job at the time, the system has 2 units of slack. Slack stealer preempts $J_{1,1}$ and starts to execute A_1.
2. At time 4.5, A_1 completes. The slack stealer is suspended. The job $J_{1,1}$ resumes and executes to completion on time.
3. At time 5.5, A_2 arrives, and the slack stealer becomes ready again. At this time, the execution of the second job $J_{1,2}$ of T_1 can be postponed until time 7.5, and the second job $J_{2,2}$ of T_2 can be postponed until 12.5. Hence, the system as a whole has 2.0 units of slack. Slack stealer executes A_2.
4. At time 7.5, all the slack consumed, the slack stealer is given the lowest priority, $J_{1,2}$ preempts the slack stealer and executes.
5. At time 9, $J_{1,2}$ completes, and the system again has slack. The slack stealer now has the highest priority. It executes A_2.
6. When A_2 completes, the slack stealer is suspended again. For as long as there is no job in the aperiodic job queue, the periodic tasks execute on the EDF basis.

Slack Computation

A slack computation algorithm is correct if it never says that the system has slack when the system does not. An optimal slack computation algorithm gives the exact amount of slack the system has at the time of the computation. A non-optimal slack computation algorithm gives a lower bound to the available slack.

There are two approaches to slack computation: static and dynamic.

Scanned by CamScanner

Static approach:

According to this approach, the initial slacks of all periodic jobs are computed off-line based on given parameters of the periodic tasks. The scheduler only needs to update the slack information during run time to keep the information current, rather than having to generate the information from the scratch.

Dynamic approach:

The scheduler computes the amount of available slack during runtime. When the interrelease times of periodic jobs vary widely, dynamic-slack computation is the only choice.

Disadvantage: High run-time overhead

Slack stealing in Fixed Priority Systems

$T_1 = (3,1)$, $T_2 = (4,1)$, $T_3 = (6,1)$

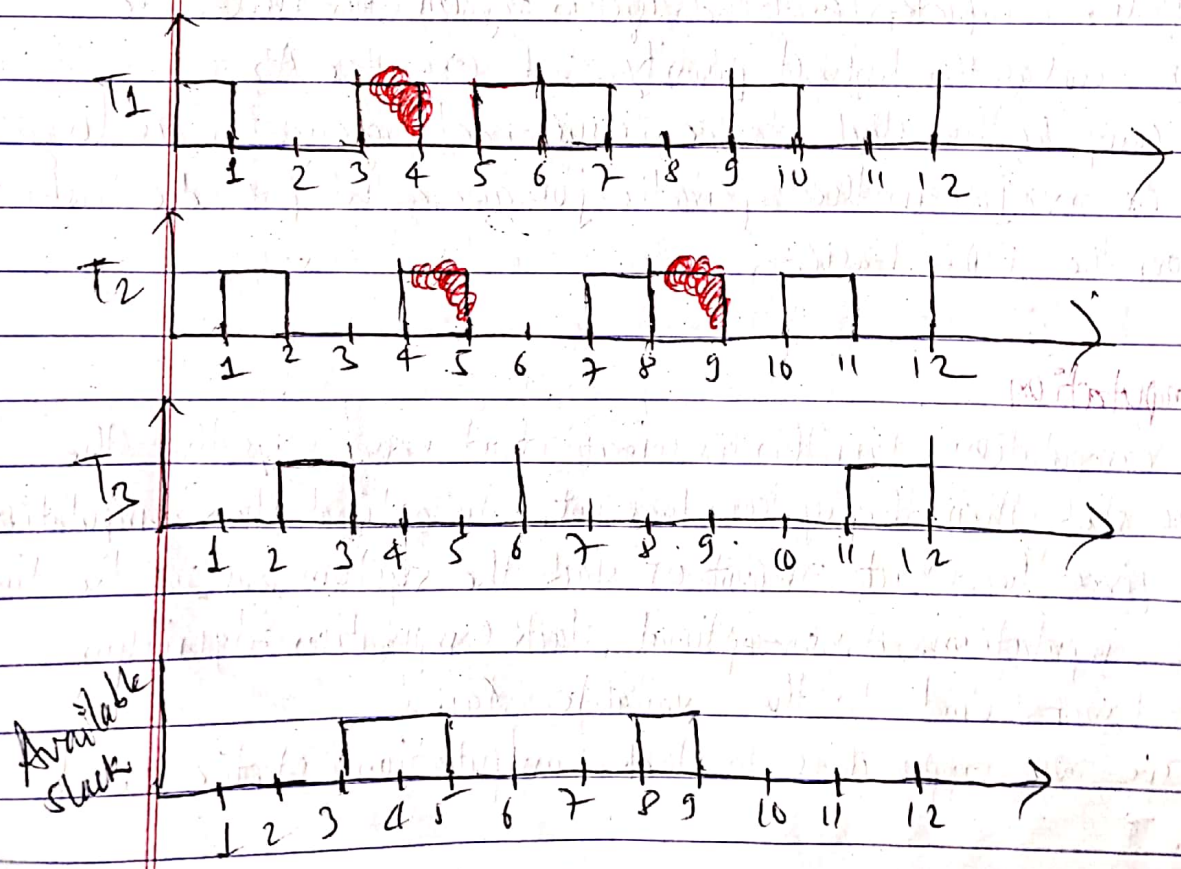
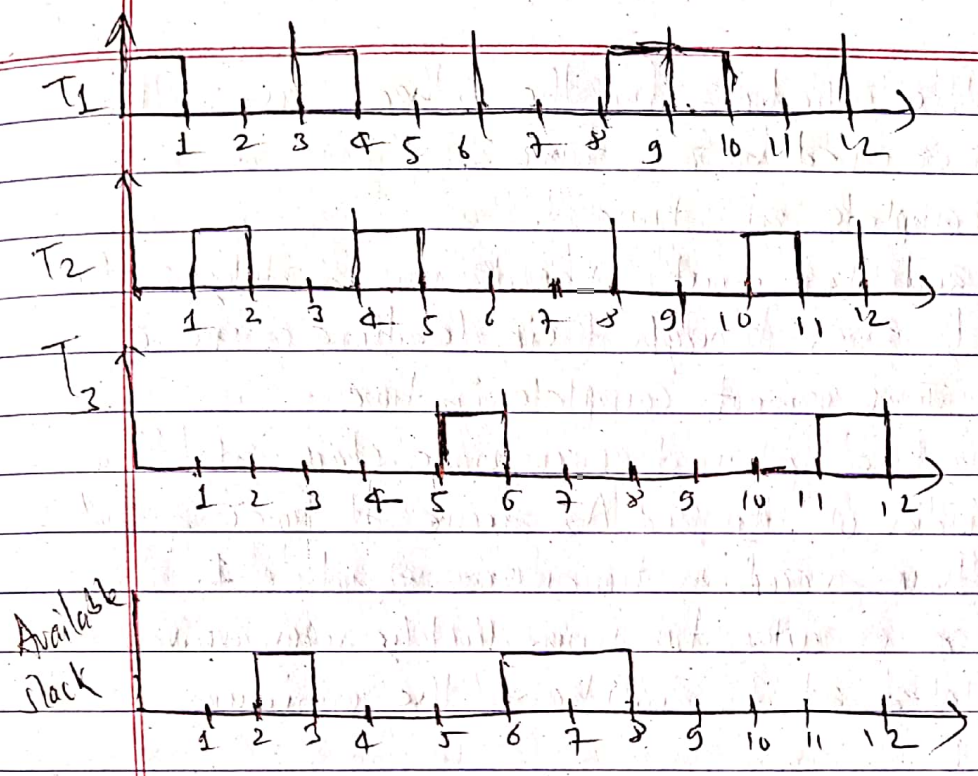


fig. (a)

Scanned by CamScanner



(b)

- The tasks are scheduled rate monotincally
- If the system were deadline-driven, it would have 2 units of slack in the interval (0, 3] but this system has only 1 unit.
- The reason is that once $J_{1,2}$ becomes ready $J_{2,1}$ must wait for it to complete.
- As a consequence, $J_{2,1}$ must complete by time 3, although its deadline is 4.

Optimality Criterion and Design Consideration

- Figure (a) shows the schedule for the case when the 1 unit of slack is not used before time 3.
- At time 3, $J_{3,1}$ has already completed
- $J_{1,2}$ and $J_{2,2}$ can start as late as time 5 and 7 respectively,
- Therefore, the system has two units of slack at time 3

Scanned by CamScanner

- Figure (b) shows the schedule for the other case: The 1 unit of slack is used before time 3.
- $J_{3,1}$ is not yet complete at time 3.
- Consequently, $J_{1,2}$ and $J_{2,2}$ must execute immediately after they are released, even though their deadlines are 6 and 8; otherwise $J_{3,1}$ cannot complete in time.
- Under this condition, the system has no more slack until time 6.
- Suppose that aperiodic jobs A_1 and A_2 arrive at times 2 and 3, respectively, and their execution times are equal to 1.
- If the slack stealer executes A_1 immediately upon arrival, the job can be completed at 3 and have the minimum response time of 1
- Since the system has no more slack until the time 6, A_2 cannot be completed until time 7 (i.e. its response time is 4)
- On the other hand, if the scheduler waits until time 3 and then schedules the ~~st~~ slack stealer, the aperiodic jobs are completed at times 4 and 5, respectively.
- A_2 now has the minimum response time of 2
- This reduction is achieved at the expense of A_1, whose response time is no longer ~~than~~ the minimum possible

- As a variation, suppose that A_1 is the only aperiodic job
- When its execution time is 1, its response time is minimum if the job is executed immediately upon its arrival.
- However, if the execution time of the job ~~where~~ were larger than 1 but no greater than 2, the scheduler would do better by waiting until time 3 to start its execution; the response time of the job would be 3 or less but would be more than 4 if the job were executed starting from time 2.
- If the execution time of A_1 were between 2 and 3, the scheduler

would again do better by letting the job be executed immediately upon its arrival; the job would be completed by time 8.

This example points out the following important facts:

1. No slack-stealing algorithm can minimize the response time of every aperiodic job in a fixed-priority system even when prior knowledge on the arrival times and execution times of aperiodic jobs is available.
2. The amount of slack a fixed-priority system has, in a time interval may depend on when the slack is used. To minimize the response time of an aperiodic job, the decision on when to schedule the job must take into account the execution time of the job.

Because of (1), we use here a weaker optimality criterion: A slack-stealing algorithm for fixed-priority systems is optimal in the weak sense if it is correct and it minimizes the response time of the job at the head of the aperiodic job queue.

Because of (2), an optimal slack-stealing algorithm for fixed priority systems does not use available slack greedily.

Scanned by CamScanner

Unit 8 Resource And Resource Access Control

Date
Page

Priority Inversion:

Priority Inversion is a scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.

Unbounded Priority Inversion:

- A low priority is holding a resource.
- A high priority task is waiting.
- An intermediate priority task which does not need resource preempts the lower priority task.

Chain Blocking:

A task needing to use a set of resources is said to undergo chain blocking:

- If each time it needs a resource, it undergoes priority inversion.

every processor has its own scheduler.

Chapter 9 Model of Multiprocessor and Distributed Systems

In MPCP (Multiprocessor Priority Ceiling Protocol)

Synchronization processor:

- The processor on which each resource resides
- By residing we mean, the scheduler of the processor controls the access to the resource R and when a job uses the resource its critical section executes on the processor.

Local processor:

- Each job is released and becomes ready for execution

Local resource: From the perspective of a job

A resource that also resides on the local processor of the job

Scanned by CamScanner

Date
Page

Remote resource
- A resource that resides on another processor

Global resource
- A resource required by jobs that have different local processors

chapter 7

Slack Computation in Fixed-Priority Systems

The effective deadline of every job in the highest priority periodic task is equal to its deadline.
For $i > 1$, the effective deadline $d^e_{i,j}$ of a job $J_{i,j}$ is equal to the deadline $d_{i,j}$ of the job if higher-priority periodic tasks have no ready jobs immediately before $d_{i,j}$. Otherwise, if $d_{i,j}$ is amid or at the end of a level π_{i-1} busy interval, $d^e_{i,j}$ is equal to the beginning of this busy interval.

The initial slack of $J_{i,j}$ is given by

$$\sigma_{i,j}(0) = \max\left(0, d^e_{i,j} - \sum_{k=1}^{i} e_k \left\lceil \frac{d^e_{i,j}}{p_k} \right\rceil\right)$$

The effective deadlines of all N periodic jobs in a hyperperiod can be determined by constructing a schedule of the periodic tasks for the hyperperiod when $\sigma_{i,j}(0)$'s are computed. The N effective deadlines and initial amounts of slack are stored for use at runtime.

Slack Functions: slack computation is done at t_c.

let t be a future time instant (i.e. $t > t_c$) and $J_{i,j}$ be the first job in T_i whose effective deadline is after t, i.e.
$d^e_{i,j-1} \le t < d^e_{i,j}$.

~~t_c~~ The amount of time, before effective deadline $d^e_{i,j}$ of $J_{i,j}$ if the job is not complete or before effective deadline $d^e_{i,j+1}$ of the job $J_{i,j+1}$ if $J_{i,j}$ is complete, needed by jobs in $H_i(t_c)$

The slack function of T_i is given by,

$$\sigma_i(t_c, t) = \begin{cases} \sigma_{i,j}(0) - I - ST - \Theta_i & \text{if } J_{i,j} \text{ is not completed at } t \\ \sigma_{i,j+1}(0) - I - ST - \Theta_i & \text{if } J_{i,j} \text{ is completed at } t \end{cases}$$

The minimum slack function of the periodic tasks is,

$$\sigma^*(t_c, t) = \min_{1 \le i \le n} \sigma_i(t_c, t)$$ gives the minimum amount of slack from t_c to $x_i(t)$.

I = total idle time

and ST = the total stolen time, i.e. the cumulative amount of time since t_0 during which the ~~processor idles~~ aperiodic job executes.

Θ_i, for $i = 1, 2, \ldots, n$ is the cumulative amount of time since t_0 used to execute periodic tasks with priorities lower than T_i

~~Σ_i, for $i = 1, 2, \ldots, n$ is the cumulative amount of processor time since t_0 used to execute T_i~~

Computation Procedure

The slack function $\sigma_i(t_c, t)$ of each task T_i is a staircase function and it has a step (increase) at the effective deadline of each job in T_i.

Let $y_1 < y_2 < \cdots < y_k$ be the values of t at which $\sigma^*(t_c, t)$ increases; y_k is the effective deadline of some periodic job.

According to the algorithm,

- Each slack computation finds the next step of the minimum slack function and the value of the function until the step.
- The initial step of y_0 occurs immediately after the beginning of the current hyperperiod, and the slack of each task T_i at the initial step is equal to $\sigma_i(0)$.
- Let y_{k-1} $(k > 0)$ denote the step of the minimum slack function

found during the initial or the previous slack computation.

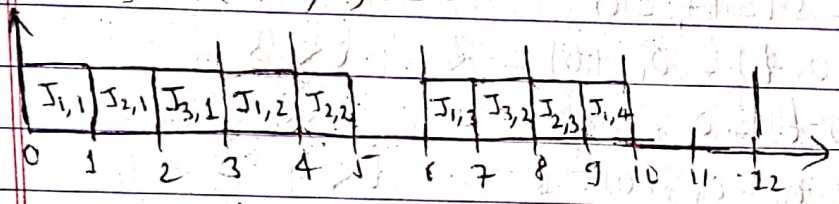
- z_i denotes the next step of $\sigma_i(t_c, t)$
- z_{min} denotes the step among z_i for $1 \le i \le n$ that has the least slack.

  If there is a tie, z_{min} is the latest among the z_i's that have the minimum slack.
- The next step y_k is equal to z_{min}.

Example:

$T_1 = (3,1)$, $T_2 = (4,1)$ and $T_3 = (6,1)$

L.C.M. $(3,4,6) = 12$



The ~~length~~ level-π_1 busy intervals are

$(0,1)$, $(3,4)$, $(6,7)$, $(9,10)$

The level-π_2 busy intervals are

$(0,2)$, $(4,5)$, $(8,9)$

The level-π_3 busy intervals are

$(0,3)$, $(6,8)$

The effective deadlines of jobs are

$d^e_{1,1} = 3$, $d^e_{1,2} = 6$, $d^e_{1,3} = 9$, $d^e_{1,4} = 12$

$d^e_{2,1} = 3$, $d^e_{2,2} = 8$, $d^e_{2,3} = 12$

$d^e_{3,1} = 6$, $d^e_{3,2} = 12$

The initial slack stack of jobs are

$\sigma_{1,1}(0) = \max(0, 3 - e_1 \times \lceil \frac{3}{p_1} \rceil) = \max(0, 3 - 1 \times \lceil \frac{3}{3} \rceil) = 2$

$\sigma_{1,2}(0) = \max(0, 6 - 1 \times \lceil \frac{6}{3} \rceil) = 4$

Scanned by CamScanner

$\sigma_{1,3}(0) = \max(0, 9 - 1\times\lceil\frac{9}{3}\rceil) = 6$

$\sigma_{1,4}(0) = \max(0, 12 - 1\times\lceil\frac{12}{3}\rceil) = 8$

$\sigma_{2,1}(0) = \max(0, 3 - 1\times\lceil\frac{3}{3}\rceil - 1\times\lceil\frac{3}{4}\rceil) = \max(0, 3-1-1) = 1$

$\sigma_{2,2}(0) = \max(0, 8 - 1\times\lceil\frac{8}{3}\rceil - 1\times\lceil\frac{8}{4}\rceil) = \max(0, 8-3-2) = 3$

$\sigma_{2,3}(0) = \max(0, 12 - 1\times\lceil\frac{12}{3}\rceil - 1\times\lceil\frac{12}{4}\rceil) = \max(0, 12-4-3) = 5$

$\sigma_{3,1}(0) = \max(0, 6 - 1\times\lceil\frac{6}{3}\rceil - 1\times\lceil\frac{6}{4}\rceil - 1\times\lceil\frac{6}{6}\rceil) = 1$

$\sigma_{3,2}(0) = \max(0, 12 - 1\times\lceil\frac{12}{3}\rceil - 1\times\lceil\frac{12}{4}\rceil - 1\times\lceil\frac{12}{6}\rceil) = 3$

For $t_1 = 0$, $I = ST = \Theta_1 = 0$

$\cancel{\sigma_1(0,3)} = \sigma_1(0,t) = \sigma_{1,1}(0) = 2 \quad t<3$

$\sigma_1(3,t) = \sigma_{1,1}(0) = 4 \quad t<6$

$\sigma_1(6,t) = \sigma_{1,3}(0) = 6 \quad t<9$

$\sigma_1(9,t) = \sigma_{1,4}(0) = 8 \quad t<12$

$\sigma_2(0,t) = \sigma_{2,1}(0) = 1 \quad t<3$

$\sigma_2(3,t) = \sigma_{2,2}(0) = 3 \quad t<8$

$\sigma_2(8,t) = \sigma_{2,3}(0) = 5 \quad t<12$

$\sigma_3(0,t) = \sigma_{3,1}(0) = 1 \quad t<6$

$\sigma_3(6,t) = \sigma_{3,2}(0) = 3 \quad t<12$

$\cancel{\sigma^*(0,2) = 1, \ \sigma^*(3,4) = 2}$

fig. slack functions

At $t_c = 3$, $I = 0$, $\Theta_1 = 1+1 = 2$, $\Theta_2 = 1$, $ST = 0$

$\sigma_1(3,t) = \sigma_{1,2}(0) - I - ST - \Theta_1 = 4 - 2 = 2 \quad t<6$

$\sigma_1(3,t) = \sigma_{1,3}(0) - 0 - \Theta_1 = 6 - 2 = 4, \quad t<9$

$\sigma_1(3,t) = \sigma_{1,4}(0) - 0 - 0 - 2 = 8 - 2 = 6, \quad t<12$

$\sigma_2(3,t) = \sigma_{2,2}(0) - \Theta_2 = 3 - 1 = 2, \quad t<8$

$\sigma_2(3,t) = \sigma_{2,3}(0) - \Theta_2 = 5 - 1 = 4, \quad t<12$

$\sigma_3(3,t) = \sigma_{3,2}(0) - \Theta_3 = 3 - 0 = 3 \quad t<6$

$\sigma_3(3,t) = \sigma_{3,2}(0) - 0 = 3 - 0 = 3 \quad t<12$

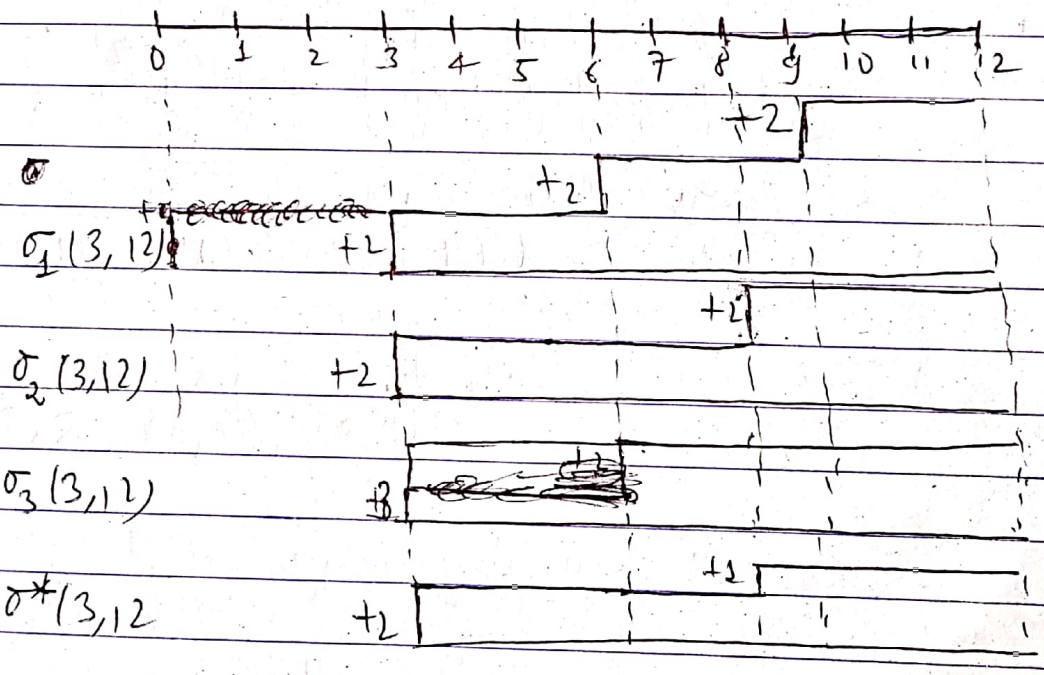


fig. Slack functions

The minimum slack function $\sigma^*(0,t)$ shows that the periodic tasks have 1 unit of slack before 6 and 2 more units after 6. The amount of slack available before 6 increases by 1 at time 3.

Suppose that the scheduler does a slack computation at time $t_c = 1$.

Date
Page

$\sigma_1(1,t) = \sigma_{1,1}(0) - 0 - 0 - 0 = 2 \quad t<3$

$\sigma_2(1,t) = \sigma_{2,1}(0) - 0 - 0 - 0 = 1 \quad t<3$

$\sigma_3(1,t) = \sigma_{3,1}(0) - 0 - 0 - 0 = 1 \quad t<6$

Here, $z_1 = 3,\ z_2 = 3,\ z_3 = 6$

and $z_{min} = 6$

$\therefore\ \sigma^*(t, y_1) = 1$ and the constraining job is $J_{3,1}$.

it has $d_i^p = z_{min}$

Scanned by CamScanner

Radar Range Equation

→ Power density by isotropic source $= \frac{P_t}{4\pi R^2}$

R = range

Signal Processing Applications:

response time → few milliseconds to a few seconds

E.g. digital filtering, video and voice compression/decompression and radar signal processing

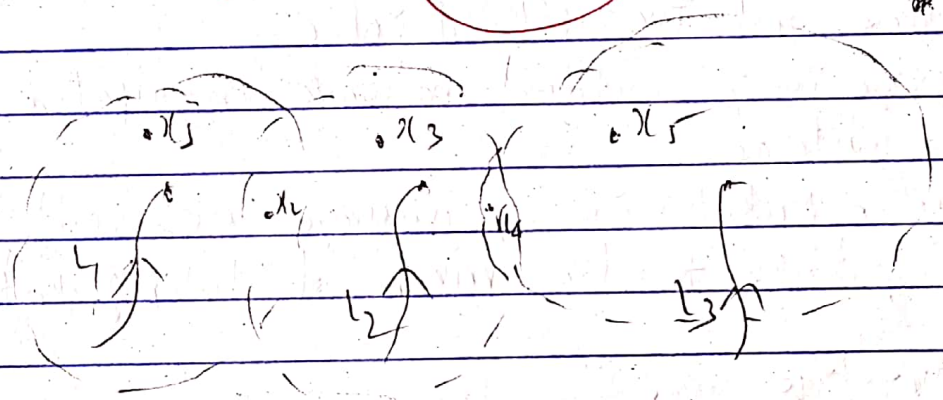
Radar System = Radar Signal Processing system + Tracking System

next cycle:

tentative assignments → trial

$x_1 \leftrightarrow L_1$

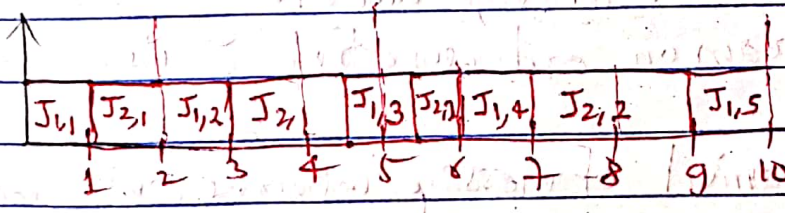
x_6 $x_3 \leftrightarrow L_1, L_2$



$x_1 \leftrightarrow L_1$, $x_2 \leftrightarrow L_1, L_2$, $x_3 \leftrightarrow L_2$, $x_4 \leftrightarrow L_2, L_3$, $x_5 \leftrightarrow L_3$

$x_1 \leftrightarrow L_1$, $x_5 \leftrightarrow L_3$, $x \leftrightarrow L_2$ $x_3 \leftrightarrow L_2$ $x_4 \leftrightarrow L_3$

$x_1 \leftrightarrow L_1$ $x_2 \leftrightarrow L_2$, $x_3 \not\leftrightarrow L_3$, $x_4 \leftrightarrow L_3$ $x_1 \not\leftrightarrow L_3$



Scanned by CamScanner

WFQ

Each connection has FIFO Queue

backlogged if has ready ~~jobs~~ packets or currently being transmitted packet

ready packet → packet at the head of the queue

WFQ computes f_n of ready packets only

Ready packets मध्य कसलाई transmit गर्ने

→ decided by SFQ queue's order

f_{n_i} is computed when ~~connection-i is idle and~~ connection-i packet arrives on idle connection-i

WRR

- not suitable for scheduling precedence-constraint jobs
- suitable for scheduling message transmission through switches, and for pipelined jobs
- message in a switched network are pipelined through the switches.

Earlier packets in a message are sent from switch without having to wait for arrivals of later packets

Advantage over time-driven

- It doesn't require globally synchronized clocks, often used for end-to-end scheduling in a network or distributed system

Advantage over priority driven scheme

- it doesn't require any sorted queue

* suitable for constant-bit rate traffic such as uncompressed voice. These messages fit periodic message model.

So, the message stream on each connection i is

$M_i = (p_i, e_i, D_i)$.

p_i = minimum interarrival of message instances on connection i

e_i = maximum number of packets in each message (instance)

D_i = end-to-end relative deadline for delivery of every message

Scanned by CamScanner

Greedy WRR

- During the connection establishment,
  - the scheduler at each switch, assigns a weight wt_i to the new connection i.
  - this means that the connection i is allocated wt_i slots in each round during which packets on all existing connections sharing the same output link are transmitted in turn.
  - Each slot has length 1, the time to transmit a maximum-size packet.

Throughput and Delay Guarantees

During each round, if more than wt_i packets on connection i are waiting, wt_i packets are transmitted.

If wt_i or fewer packets are waiting, all packets on the connection are transmitted.

After the scheduler completes the transmission of packets on connection i, it proceeds to ~~transfer~~ transmit packets on connection $i+1$ in the same manner, except, the maximum number of packets on connection $(i+1)$ transmitted per round is wt_{i+1}.

In this way, each connection i is guaranteed ~~to~~ wt_i slots in each round.

RL = round length (of the output link).

is the maximum no. of slots per round.

At all times, $\sum_{i=1}^{n} wt_i \leq RL$

Each connection i is guaranteed the throughput $\frac{wt_i}{RL}$

$RL < P_{min}$ where P_{min} is the minimum periods of messages on all existing connections.

Scanned by CamScanner

$$wt_i \geq \left\lceil \frac{e_i}{\lfloor p_i / RL \rfloor} \right\rceil$$

Connection Establishment

- Weights of all connections on an output link depend on round length used for the link. Change of RL requires changes of weights which would be costly.

Hence, round length is chosen apriori satisfying constraint $RL \leq p_{min}$

The request for each new connection provides the parameters p_i, e_i and D_i of the message stream M_i to be carried out by the new ~~messag~~ connection i.

The scheduler of each switch computes the weight wt_i required by the connection according to the expression $wt_i \geq \left\lceil \frac{e_i}{\lfloor \frac{p_i}{RL} \rfloor} \right\rceil$

If the sum of weights assigned to all existing connections is no more than $RL - wt_i$, the scheduler accepts the connection and allocates wt_i slots per round to the connection. Otherwise, the scheduler rejects the connection.

Scanned by CamScanner

Communication in multicomputer system

-) concerned with flow control and scheduling in switched networks that are used in multicomputer system

-) multihop networks adopt a simple routing and flow-control scheme called wormhole routing

Wormhole Networks

- Networks that use wormhole routing are called wormhole networks.
- In a wormhole network, messages are segmented into very small flow-control units, called flits.
- Each switch provides enough buffer space to hold one flit per input link.

The buffer is there in order to decouple the input link from the output links.

Routing and Transmission

- Only one flit can occupy a link at each time; while a message is using a link, another message that also needs the link must wait.
  - Therefore, the transmission of a message may be blocked from starting.
- When the header (i.e. the first flit) of a messages reaches a switch, the switch selects an output link for the message based on information provided in the header.
- If the output link is free, the header moves forward on that link to the next switch, leaving the input link it used to reach the current switch to the second flit in the message.

Similarly, the third flit ~~fto~~ follows, using the link freed by second flit and so on,

Scanned by CamScanner

- On the otherhand, if an output link chosen for a message is in use, the header is buffered and waits at the switch until the output link becomes free.
- In the meantime, subsequent flits that have reached upstream switches occupy the flit buffers there, one per switch.
- The associated links are not available to other messages during this time.
- This phase of message transmission is called the routing phase.
- It starts when header leaves the source processor and completes when it reaches the destination processor
- thereafter, the message has all the links ~~between~~ along the path between the source and destination
Each link is occupied by one message flit.
The flits shift downstream by one link in each time step without intervention of the switches.
The message "worms" its way non-preemptively through the network without being queued at any switch.
Transmission completes when last flit reaches the destination.

(6,14], (3,7], (0,18] → Conflict for Resource
↳ Priority Inversion

EDF basis

Single resource R

Scanned by CamScanner

A simple Acceptance Test in Deadline-Driven System

- EDF basis ~~ﾌﾟ~~ schedule
- The total density of periodic task $= \Delta$
- All the accepted sporadic jobs can meet their deadlines as long as the total density of all the active sporadic jobs is no greater than $1-\Delta$ at all times.

Acceptance Test Procedure:

- For the first sporadic job $s(t, d, e)$, the scheduler accepts s if its density $\frac{e}{d-t}$ is no greater than $1-\Delta$.

- If the scheduler accepts the job, the deadline d of s divides the time after t into two disjoint time intervals;
  * the Interval I_1 at and before d
  * and the interval I_2 after d

The job s is active in I_1, ~~and~~ but not in I_2.
consequently, the total densities $\Delta_{s,1}$ and $\Delta_{s,2}$ of the active sporadic jobs in these intervals are equal to $\frac{e}{d-t}$ and 0 respectively

For general case;
At time t when the scheduler does an acceptance test on $s(t, d, e)$, there are n_s active sporadic jobs in the system.
~~The deadlines of accept active these sporadic jobs are sorted~~
~~maintai~~
The scheduler maintains a non-decreasing list of deadlines of these sporadic jobs.
Those deadlines partition the time interval from t to infinity into n_s+1 disjoint intervals: $I_1, I_2, \ldots, I_{n_s+1}$.
~~I_1 begins at t and~~ For ~~each~~ $1 \leq k \leq n_s$, each subsequent interval I_{k+1} begins when the previous interval I_k ends and ends at the next deadline in the list or in the infinity in case of I_v

Scheduler keeps up-to-date the total density $\Delta_{s,k}$ of the sporadic jobs that are active during each of these intervals.

let I_ℓ be the time interval containing the deadline d of the new sporadic job $s(t,d,e)$.

The scheduler accepts the job s if

$$\frac{e}{d-t} + \Delta_{s,k} \leq 1 - \Delta \quad \text{for all } k = 1, 2, \ldots, \ell$$

otherwise rejets s.

Example: $T_1 = (4, 1)$ and $T_2 = (6, 1.5)$

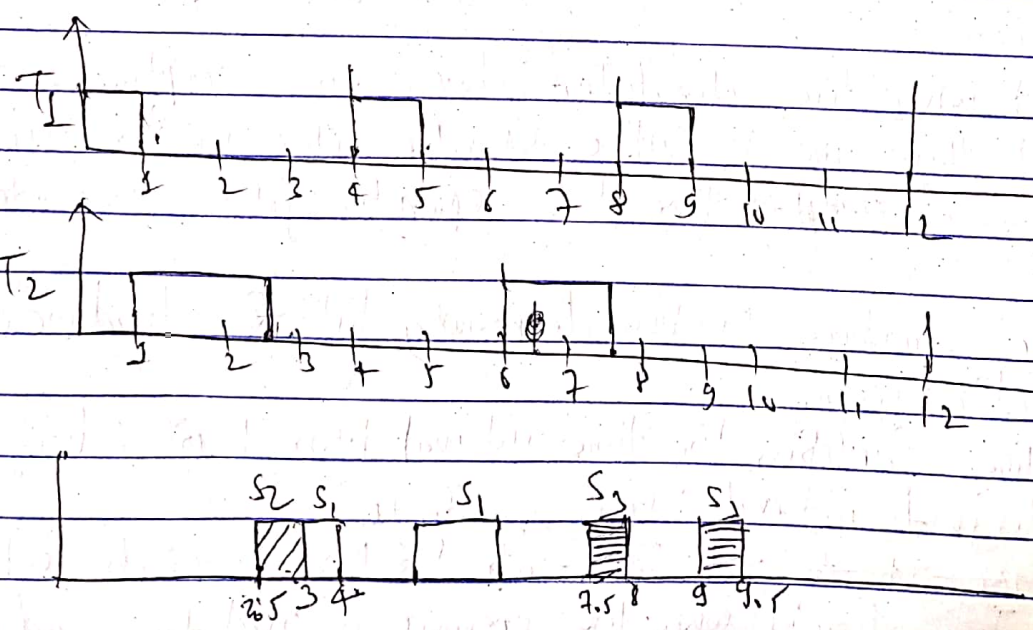
$$\Delta = \frac{1}{4} + \frac{1.5}{6} = 0.5$$

$$1 - \Delta = 0.5$$

$S_1(0^+, 8, 2)$, $S_2(2, 7, 0.5)$, $S_3(4, 14, 1)$, $S_4(9, 13, 2)$

$$\Delta_{s,1} = \frac{2}{8-0} = 0.25 \qquad \Delta_{s,3} = \frac{1}{14-4} = 0.1$$

$$\Delta_{s,2} = \frac{0.5}{7-2} = 0.1 \qquad \Delta_{s,4} = \frac{2}{13-9} = 0.5$$



Scanned by CamScanner