Real-Time Task Scheduling
Real-Time Systems Design (CS 6414)

Sumanta Pyne

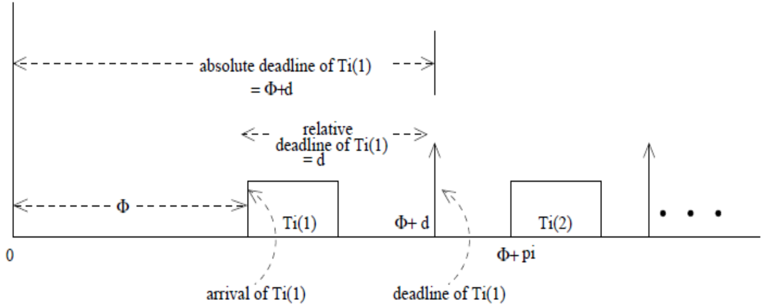
Assistant Professor
Computer Science and Engineering Department
National Institute of Technology Rourkela
pynes@nitrkl.ac.in

January 26, 2021

Important Concepts and Terminologies

- Task Instance
    - A task is generated when some specific event occurs
    - Real-time tasks recur large amount of times at different instants
    - Each time a task recurs - instance of task
    - First occurrence of a task - first instance
    - Next occurrences - second and so on . . .
    - *j*th instance of task *Ti* - *Ti(j)*
    - Each task instance need to complete and produce result in deadline
    - Task instance also referred a process
    - Temperature sensing in chemical plant
        - recur indefinitely
        - with a certain period
        - temperature sampled periodically
    - Task handling a device - interupt might recur at random instants

Relative Deadline vs. Absolute Deadline



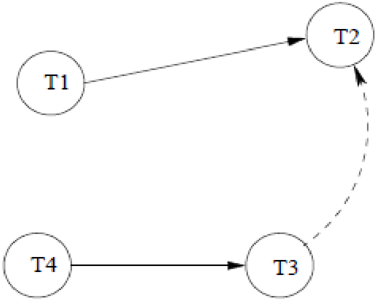
- Absolute deadline
  - absolute time value counted from 0 by which expect results
  - interval between time 0 and actual instant of deadline
  - measured by physical clock
- Relative deadline
  - time interval between start of task and instant of deadline
  - time interval between arrival of task and its deadline

Response Time

- Time taken by a task to produce its results
- Measured from task arrival time
- Duration occurrence of event generating the task and get its result
- Internal event - clock interrupt, external event - robot obstacle
- Not important for hard real-time tasks as long as deadlines are met
- Important performance metric for soft real-time task scheduler
- Soft real-time task scheduler minimize average response time of tasks

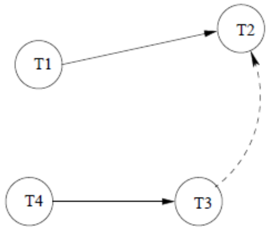
Task Precedence

- Task T1 precedes task T2 if T1 must complete before T2 can start
- When a task Ti precedes another task Tj
  - Each instance of Ti precedes corresponding instance of Tj
- If T1 precedes T2, then
  - T1(1) precedes T2(1), T1(2) precedes T2(2) and so on
- A precedence order defines a partial order among tasks
- Task precedence relation for task scheduling algorithms



Data Sharing

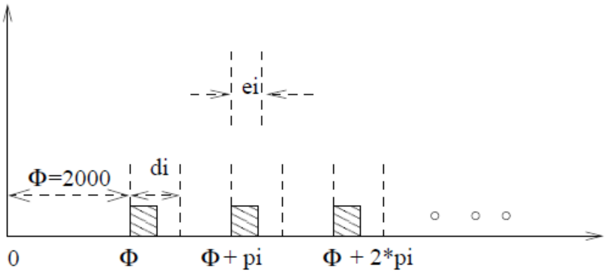
- Tasks share results among each other
- When one task needs to share result produced by another
  - Second task must precede first task
- Precedence relation implies data sharing between two tasks
- A task may precede another even no need of data sharing
- In a chemical plant
  - Reaction chamber filled with water before chemicals introduced
  - Water filling task complete before task introducing chemicals
- Data sharing among concurrent and overlapping tasks
- Data sharing among two tasks is represented by dashed arrow



Types of Real-Time Tasks

- Periodic Tasks
  - Repeats after a certain fixed time interval
  - Time instants demarcated by clock interrupts
  - Also referred as clock-driven tasks
  - Period - Fixed time interval after which task repeats
- If Ti is a periodic task then
  - from time 0 till occurrence of first instance Ti(1) denoted by phi_i
  - second instance occurs at phi_i+p_i
  - third instance occurs at phi_i+2*p_i and so on
- Formally a periodic task Ti can be represented by
  - a four tuple (phi_i,p_i,e_i,d_i)
  - p_i period of task
  - e_i worst case execution time
  - d_i relative deadline

An Example - Track Correction Task of a Rocket



Track Correction Task ($2000mSec, p_i, e_i, d_i$) of a Rocket

- Track correction task starts 2000 msec after launch
- Periodically recurs every 50 msec
- Each instance requires 8 msec processing time
- Relative deadline 50 msec, Phase of task 2000 msec
- Task represented as Ti = (2000 msec, 50 msec, 8 msec, 50 msec)
- Ti = (2000 msec,50 msec,8 msec) when p_i = d_i
- Ti = (20 msec,100 msec) indicates
- phi_i = 0, p_i = 100 msec, e_i = 20 msec, d_i = 100 msec

Periodic Tasks

- Majority of real-time tasks are periodic
- Monitoring certain conditions
- Polling information from sensors at regular intervals
- In a chemical plant
  - temperature, pressure and chemical concentration monitors
  - periodically sample and communicate to plant controller
  - instances of monitoring tasks generated by periodic timer interrupt
  - corrective actions taken by actuators to maintain safe reaction rate
- Dynamic period tasks
  - Computation in air traffic monitors
  - Once flight detected, radar exits radar signal zone

Sporadic Task

- Recurs at random instants
- Represented as Ti = (ei,gi,di)
  - ei - worst case execution time of an task instance
  - gi - minimum separation between two consecutive task instances
  - Once an instance occurs, next instance can't occur before gi time units
  - gi restricts rate at which sporadic tasks can arise
  - di - deadline
  - Instances - 1st Ti(1), 2nd Ti(2), 3rd Ti(3) and so on
- Sporadic tasks like emergency message arrival highly critical
- In robot a task generated to handle sudden obstacles
- In factory a task handling fire conditions
- Time of occurence cannot be predicted
- Varies from highly critical to moderately critical
- I/O device and DMA interrupts are moderately critical
- Task handling reporting of fire conditions is highly critical

Aperiodic Task

- Many ways similar to sporadic task
- Arise at random instants
- Minimum separation time gi between two tasks can be 0
- Deadline is expressed as an average value or statistically
- Are generally soft real-time tasks
- Can recur in quick succession
- Difficult to meet deadline of all task instances
- When several aperiodic tasks occur in quick succession
  - there is bunching of task instances
  - it might lead to few deadline misses
  - Signals sampled at a prespecified frame rate
- Logging task in a distributed system
  - logging requests from different tasks may arrive at same time, or
  - requests may be spaced out in time
- Operator requests, keyboard presses, mouse movements
- All interactive commands handled by aperiodic tasks

Task Scheduling

- Determines order of tasks executed by operating system
- Operating system relies one or more task schedulers
- Task scheduler characterized by scheduling algorithm
- Large number of algorithms for real-time task scheduling
- Real-time task scheduling on uniprocessor developed since 1970s
- Classify existing scheduling algorithms into few broad classes
- Study characteristics of few important classes

Basic Concepts and Terminologies

- Valid Schedule
  - set of tasks where at most one task assigned processor at a time
  - no task is scheduled before its arrival time
  - precedence and resource contraints of all tasks are satisfied
- Feasible Schedule
  - a valid schedule is feasible if all tasks meet time constraints
- Proficient Scheduler
  - Task scheduler sch1 more proficient than task scheduler sch2
  - if sch1 can feasibly schedule all task sets sch2 can feasibly schedule,
  - but not vice versa
  - Equally proficient schedulers
    - if sch1 can feasibly schedule all task sets sch2 can feasibly schedule,
    - and vice versa
- Optimal Scheduler
  - If it can feasibly schedule any task set than any other feasible scheduler
  - Most proficient scheduling algorithm
  - If can't schedule some task set, then no other should be able to d o it

Basic Concepts and Terminologies

- Scheduling Points
  - Points on time line at which scheduler decides task to run next
  - Scheduler activated by operating system at scheduling points
  - Defined at time instants marked by interrupts in clock-driven scheduler
  - Determined by occurrence of certain events in event-driven scheduler
- Preemptive Scheduler
  - Higher priority task arrives, suspends execution of lower priority task
  - Takes up higher priority task for execution
  - No higher priority task ready and waiting, lower priority task executing
  - Lower priority task resumes when no higher priority task is ready
- Utilization
  - Average time for which a task executes per unit time interval
  - For a periodic task Ti, utilization ui=ei/pi
  - ei execution time, pi period of Ti
  - For a set of periodic tasks {Ti}, total utilization due to all tasks U=e1/p1+e2/p2+...+en/pn
  - Good scheduling algorithm targets very high utilization i.e. $U \rightarrow 1$
  - $U>1$ is not possible on a uniprocessor

Basic Concepts and Terminologies

- Jitter
  - Deviation of a periodic task from its strict periodic behaviour
  - Arrival time jitter deviation of arriving task from periodic arrival time
  - Arrival jitter caused by imprecise clocks or network congestion
  - Completion jitter deviation of task completion from periodic points
  - Completion jitter caused by increase in load at an instant
  - Jitters are undesirable for some applications

Classification of Real-Time Scheduling Algorithms

- Clock-driven
  - Scheduling points determined by interrupts received from a clock
  - Members – Table-driven, Cyclic
  - Satisfactorily handle periodic tasks
  - Simple and efficient
  - Used in embedded applications
- Event-driven
  - Scheduling points are defined by events precluding clock interrupts
  - Members – Simple priority-based, Rate Monotonic Analysis (RMA), Earliest Deadline First (EDF)
  - More sophisticated, proficient, flexible than clock-driven
  - Proficient - Can feasibly schedule task sets which clock-driven cannot
  - Flexible - Feasibly schedule periodic, sporadic and aperiodic tasks
- Hybrid
  - Clock driven interrupts and event occurrences to define schedule points
  - Round-robin

Classification of scheduling based on task acceptance test

- Planning-based
  - Task arrives scheduler determines can it meet deadline
  - If not rejected
  - If task meets deadline without causing others miss deadline,
  - then accept and schedule it, otherwise reject it
- Best-effort
  - No acceptance test is applied
  - All arriving tasks are scheduled
  - Best effort is made to meet deadlines
  - No guarantee for meeting a task's deadline

Classification of scheduling based on target platform

- Uniprocessor
  - Simplest
- Multiprocessor
  - Scheduler first decides which task needs to run on which processor
  - Shared memory and up-to-date state information
- Distributed
  - Scheduler first decides which task needs to run on which processor
  - No shared memory and up-to-date state information
  - Assumes central state information of all tasks and processors
  - Tasks communicate using message passing
  - Minimize communication overhead

Clock-Driven Scheduling

- Decides on which task to run next at clock interrupt points
- Scheduling points are determined by timer interrupts
- Also called off-line schedulers
- Fix schedule before system starts to run
- Predetermine which task will run when
- Incur very little run time overhead
- Cannot handle aperiodic and sporadic tasks
- Also known as static scheduler

Table-Driven Scheduling

- Precompute which task would run when
- Store schedule in a table when system is designed or configured
- Application programmer can select and store his own schedule
- At run time scheduler finds schedule from schedule table with phi_i=0

Task	Start Time in milli Seconds
T_1	0
T_2	3
T_3	10
T_4	12
T_5	17

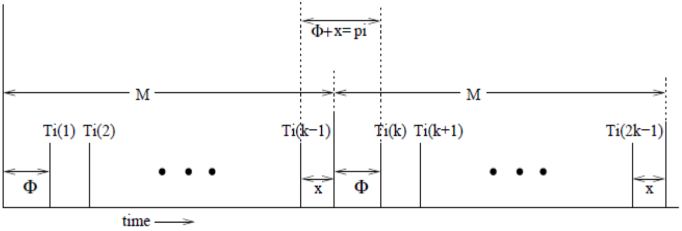
- T1 executes at 0, T2 starts 3 msec afterwards, so on
- What would be size of schedule table ?
- Set of tasks ST={Ti:i∈{1,2,...,n}}, pi is period of Ti
- Table entries replicate after LCM(p1,p2,...,pn) time units
- For example: (e1=5,p1=20), (e2=20,p2=100), (e3=30,p3=250)
- Schedule repeats after every 500=LCM(20,100,250) time units

Major Cycle

- Sufficient to store LCM(p1,p2,. . . ,pn) duration in table
- LCM(p1,p2,. . . ,pn) is major cycle for ST
- A major cycle of set of tasks
  - is an interval of time on time line such that
  - in each major cycle different tasks recur identically

Theorem 1

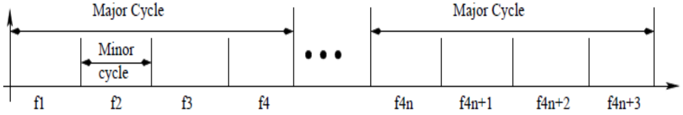
- The major cycle of a set of tasks ST={T1,T2,...,Tn} is LCM(p1,p2,...,pn) even when tasks have arbitrary phasings.



- Proof
  - k-1 occurrences of Ti in a major cycle
  - Ti(1) starts phi time units from start of major cycle
  - Major cycle ends x time units after Ti(k-1), same for each major cycle
  - Size of each major cycle M = (k-1)pi + phi + x (1)
  - Ti to have identical occurrence times in each major cycle, pi=phi + x
  - Substituting pi in (1), M=(k-1)*pi+pi=k*pi contains multiple of pi
  - Therefore, M=LCM(p1,p2,...,pn)

Cyclic Schedulers

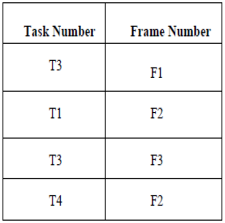
- Popular, simple, efficient, easy to program
- Manufacture of small embedded system applications
- Temperature controller
    - Sample temperature of a room maintained at present value
    - Embedded in computer-controlled air conditioners
- Repeat a precomputed schedule stored for one major cycle
- Scheduled task in task set repeat indentically in every major cycle
- Major cycle divided into one or more minor cycles or frames



- Scheduling points occur at frame boundaries
- Task can start execution only at beginning of frame
- Frame boundaries defined by a periodic timer

Cyclic Schedulers

- Each task assigned to run in one or more frames
- Assignment of tasks to frames stored in a *schedule table*
- Size of frame is an important design parameter, needs careful choice



Task Number	Frame Number
T3	F1
T1	F2
T3	F3
T4	F2

- A selected frame size should satisfy three constraints
  1. Minimum Context Switching
  2. Minimization of Table Size
  3. Satisfaction of Task Deadline

Minimum Context Swiching

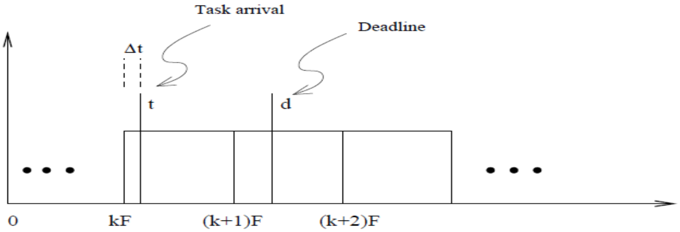
- Minimize number of context switches occurring during task execution
- Task must complete running within its assigned frame
- Context switch involve processing overheads - suspend and restart
- Avoidance - selected frame size larger than task execution time
- This allows a task to start and complete within same frame
- Formally, $\max(\{ei\}) \leq F$
    - ei - execution time of Ti, F - frame size
- Constraint imposes a lower-bound on frame size
- F must not be smaller than $\max(\{ei\})$

Minimization of Table Size

- Minimum number of entries to minimize storage requirement
- Cyclic schedulers used in small embedded applications
- This constraint important for commercial success of production
- Achieved when minor cycles squarely divides major cycle
- Results integral number of non-fractional minor cycles
- Formally, floor(M/F)=M/F
- Unless satisfied, insufficient storing schedule for one major cycle
- Non-repeatance of schedule in major cycle result larger size table

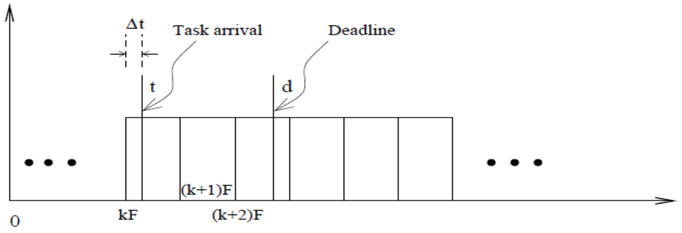
Satisfaction of Task Deadline

- At least one full frame exist between task arrival and its deadline
- Necessary, task should not miss its deadline
- By the time task is taken for scheduling, deadline is imminent
- Consider a task taken up for scheduling at start of a frame



- If no frame exists between task arrival and its completion
- Task arrives a little later after kth frame started
- Scheduler will consider it for (k+1)th frame
- Might result in missing of deadline d
- Need a full frame to exist task arrival and deadline

Satisfaction of Task Deadline



- Task arises after delta_t time units have passed since last frame
- Assuming a single frame is sufficient to complete the task
- Task can complete before deadline if
  - 2F - delta_t $\leq$ di or, 2F $\leq$ di + delta_t
- delta_t may vary from one instance of task to another
- Worst case occurs for task instance have minimum delta_t (> 0)
  - task has to wait longest before execution starts

Satisfaction of Task Deadline

- If a task arrives just after a frame starts then waits full frame duration
- Worst case - task meets deadline for min. separation from frame start
- Minimum separation value min(delta_t) determines feasible frame size
- min(delta_t)=GCD(F,pi)
- Constraint: for every Ti, 2F - GCD(F,pi) $\leq$ di
- This defines upper-bound on frame size for task Ti
- If frame size larger than upper-bound then tasks might miss deadlines
- 2F - GCD(F,pi) $\leq$ di defines frame size for only one task
- For all tasks, F $<$ max(GCD(F,pi)+di)/2

Theorem 2

- The minimum separation of task arrival from corresponding frame start time (min(delta_t)) considering all instances of a task Ti is equal GCD(F,pi)
- Proof
  - Let g = GCD(F,pi), g squarely divide each F and pi
  - Let Ti be zero phasing
  - Assume this theorem violate for integers m and n
  - Ti(n) occurs in mth frame
  - Difference start time mth frame and arrival time nth task less than g
  - $0 < (m*F - n*pi) < g \implies 0 < (m*F/g - n*pi/g) < 1$ (2)
  - F/g and pi/g are integers since g = GCD(pi,F)
  - Let F/g=I1 and Pi/g=I2 for some integral values I1 and I2
  - Substituting I1 and I2 in (2) gives $0 <$ m*I1 - n*I2 < 1
  - m*I1 and n*I2 are integers, $0 <$ m*I1 - n*I2 < 1 is not true
  - Therefore min(delta_t) of Ti can not be less than GCD(F,pi)

Satisfaction of Task Deadline

- For a task set more than one frame size satisy all three constraints
- Better to choose shortest frame size
- Schedulability of tasks $\propto$ number of frames available in a major cycle
- Suitable frame size does not mean feasible schedule
- Less number of frames available in major cycle for all task instances

Example 1

- A cyclic scheduler is to be used to run the following set of periodic tasks on a uniprocessor: T1=(e1=1,p1=4), T2=(e2=1,p2=5), T3=(e3=1,p3=20), T4=(e4=2, p4=20). Select an appropriate frame size.
- Solution:
  Constraint 1: Let F be an appropriate frame size, then max{ei} $\leq$ F. Here, F $\geq$ 2.
  Constraint 2: Major cycle for given task set M = LCM(4,5,20)=20
  M *mod* F = 0, F $\in$ {2, 4, 5, 10, 20 }, F $\neq$ 1 by constraint 1
  Constraint 3: F should satisfy: 2F - GCD(F,pi)$\leq$ di for each pi
  For F=2 and T1: 2*2-GCD(2,4)$\leq$ 4 $\implies$ 4 - 2 $\leq$ 4, satisfied for p1
  For F=2 and T2: 2*2-GCD(2,5)$\leq$ 5 $\implies$ 4 - 1 $\leq$ 5, satisfied for p2
  For F=2 and T3: 2*2-GCD(2,20)$\leq$ 20 $\implies$ 4 - 2 $\leq$ 20, satisfied for p3
  For F=2 and T4: 2*2-GCD(2,20)$\leq$ 4 $\implies$ 4 - 2 $\leq$ 20, satisfied for p4
  F=2 satisfies all constrains and is feasible

Example 1 continued

- For F=4 and T1: 2*-GCD(4,4)$\leq$ 4 $\implies$ 8 - 4 $\leq$ 4, satisfied for p1
  For F=4 and T2: 2*4-GCD(4,5)$\leq$ 5 $\implies$ 8 - 1 $\leq$ 5, not satisfied
  F=4 is not suitable frame size
- For F=5 and T1: 2*5-GCD(5,4)$\leq$ 4 $\implies$ 10 - 1 $\leq$ 4, not satisfied
  F=5 is not suitable frame size
- For F=10 and T1: 2*10 - GCD(10,4)$\leq$4 $\implies$ 20-2$\leq$4, not satisfied
  F=10 is not suitable frame size
- For F=20 and T1: 2*20 - GCD(20,4)$\leq$4 $\implies$ 40-4$\leq$4, not satisfied
  F=20 is not suitable frame size
- Only frame size 2 is suitable for scheduling.

Example 2

- Consider the following set of periodic real-time tasks to be scheduled by a cyclic scheduler: T1=(e1=1,p1=4), T2=(e2=2,p2=5), T3=(e3=5, p3=20). Determine a suitable frame size for the task set.
- Solution:
  Using first constraint $F \geq 5$.
  Using second constraint, major cycle M = LCM(4,5,20) = 20.
  Permissible values of F are 5, 10 and 20.
  No value of F satisfy third criterion.
  Split the task making task set unschedulable.
  T3 has largest execution time, constraint1 makes large feasible frames.
  Split T3 into three tasks,
  T3.1=(20,1,20),T3.2=(20,2,20),T3.3=(20,2,20).
  F=2 and F=4 become feasible frame sizes.

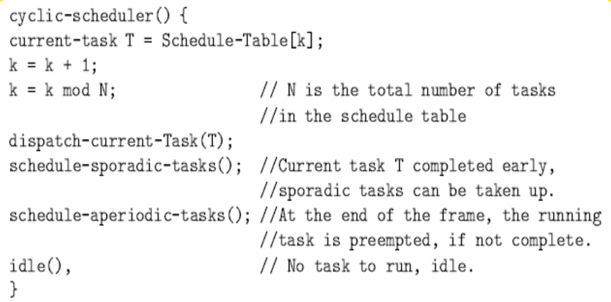
A Generalized Task Scheduler

- Cyclic schedulers restricted to periodic real-time tasks
- Practical Applications - mixture of periodic, aperiodic, sporadic tasks
- Are aperiodic and sporadic tasks accommodable by cyclic schedulers?
- Arrival times of aperiodic and sporadic tasks expressed statistically
- Assign aperiodic and sporadic tasks to frames lowering utilization
- In generalized scheduler, initial schedule for periodic tasks only
- Aperiodic and sporadic tasks scheduled in slack times in frames
- Slack time - time left after a periodic task in frame completes exec.
- Non-zero slack time in frame exists if execution time $<$ frame size
- Sporadic task scheduled if slack time allow completion before deadline
- Sporadic tasks have strict deadlines
- Acceptance test on arrival - checks sporadic task meet deadline
- Rejects if not meet - corresponding recovery routines for task are run

A Generalized Task Scheduler

- Aperiodic tasks do not have strict deadlines
- Aperiodic tasks scheduled without acceptance test
- Best effort made to schedule them in available slack times
- No guarantee given for a task's completion time
- Best effort made to complete task at its earliest
- An efficient implementation
  - store slack times in a table
  - during acceptance test use table to check schedulibilty of arriving tasks
- Another alternative popular
  - aperiodic and sporadic tasks accepted without any acceptance test
  - best effort made to meet their respective deadlines

Pseudo-code for a Generalized Scheduler



```
cyclic-scheduler() {
current-task T = Schedule-Table[k];
k = k + 1;
k = k mod N;                   // N is the total number of tasks
                               //in the schedule table
dispatch-current-Task(T);
schedule-sporadic-tasks();  //Current task T completed early,
                               //sporadic tasks can be taken up.
schedule-aperiodic-tasks(); //At the end of the frame, the running
                               //task is preempted, if not complete.
idle(),                        // No task to run, idle.
}
```

- Schedules periodic, aperiodic and sporadic tasks
- Precomputed schedule for periodic tasks stored in schedule table
- Acceptibilty test if required, schedule tasks passed test
- cyclic-scheduler() activated at end every frame by periodic timer
- If task not completed in a frame, then
  - task suspended, and
  - task to run in frame is dispatched by cyclic-scheduler()
- If task completes early in a frame, execute sporadic or periodic task

Comparison of Cyclic with Table-Driven Scheduling

- Both are important clock-driven schedulers
- Cyclic scheduler needs to set periodic timer once at initialization
- Timer continues interrupt at every frame boundary
- Table-driven scheduling - timer is set every time a task starts to run
- Overhead of call to timer every few msec degrades peformance
- Cyclic scheduler more efficient than table-driven scheduler
- Cyclic scheduler used in embedded applications
- If overhead ignored, table-driven schedule more proficient
- Because $F \geq \max\{ei\}$
- Leads to inefficiency, wastes processor time for $ei < F$

Hybrid Schedulers

- Scheduling points - clock interrupts and event occurrences
- Time-sliced round-robin scheduling - a popular hybrid scheduler
- Commonly used in traditional operating system
- Time-sliced round-robin scheduling is a preemptive scheduling method
- Round-robin scheduling - ready tasks are in held a circular queue
- Tasks are taken up one after another in a sequence from queue
- Time-sliced round-robin is less proficient than table-driven or cyclic
- Treats all tasks equally, assigns identical time slices irrespective of
    - priority, criticality or closeness of deadline

Hybrid Schedulers

- Tasks with short deadlines might fail to complete in time
- Task priorities through a minor extension to basic round-robin scheme
- Assign larger time slices to higher priority tasks
- Time-sliced round-robin not satisfactory for real-time task scheduling
- Higher priority tasks are made to complete as early as possible
- Proficient real-time schedulers maximize #tasks to meet deadline
- Completing higher priority tasks in shortest time is not important

Event-Driven Scheduling

- Cyclic scheduler are very efficient
- #tasks increase, complex to find suitable frame size, feasible schedule
- Processing time is wasted in almost every frame, F $>$ ei for all i
- Even-driven schedulers overcome these shortcomings
- Handle aperiodic and sporadic tasks more proficiently
- Less efficient as they deploy more complex scheduling algorithms
- Less suitable for embedded applications - lesser size, cost & power
- Event-driven schedulers used in moderate and large-sized applications
- Scheduling points defined by task completion and task arrival events
- Are normally preemptive
- Higher priority tasks ready preempts lower priority task running
- Three important event-driven schedulers
  1. Foreground-Background Scheduler
  2. Earliest Deadline First (EDF) Scheduler
  3. Rate Monotonic Algorithm (RMA)

Foreground-Background Scheduler

- Simplest priority-driven scheduler
- Periodic real-time tasks in application are run as foreground tasks
- Sporadic, aperiodic and non-real time tasks run as background tasks
- Foreground tasks - at every scheduling point takes highest priority task
- A background task runs when none of foreground task is ready
- Background tasks run at lowest priority
- Let T1,T2,. . . ,Tn be n foreground tasks, all periodic
- TB be only background task, eB be its processing time
- Completion time for background task, ctB=eB/(1-(e1/p1+e2/p2+. . . +en/pn))
- When any foreground task is executing, background task waits
- Average CPU utilization due to foreground task Ti is ei/pi
- CPU utilization of all foreground tasks, e1/p1+e2/p2+. . . +en/pn
- Average time available for execution of background tasks in every unit of time is 1-(e1/p1+e2/p2+. . . +en/pn)

Earliest Deadline First (EDF) Scheduler

- At every scheduling point task having shortest deadline is taken
- A task is schedulable iff total processor utilization due to task set < 1
- For set of periodic real-time tasks $\{T1, T2, \ldots, Tn\}$ with period same as deadline, EDF schedulability is
  $e1/p1 + e2/p2 + \ldots + en/pn = u1 + u2 + \ldots + un \leq 1$
- Necessary and sufficient condition
- Optimal uniprocessor scheduling algorithm
- If task set unschedulable under EDF, none can feasibly schedule it
- If pi $>$ di, each task needs ei computing time every min(pi,di) duration
- Schedulability test for period of task different from deadline,
  $$e1/\min(p1,d1) + e2/\min(p2,d2) + \ldots + en/\min(pn,dn) \leq 1 \qquad (3)$$
- If pi $<$ di, task set EDF schedulable even when task fails to meet (3)
- (3) is a sufficient condition for given task set to be EDF schedulable

Minimum Laxity First (MLF) Scheduling

- A variant of EDF scheduling
- At every schedule point, a laxity value computed for every task
- Task having minimum laxity is executed first
- Laxity - amount of time remains if task is taken up for execution next
- Laxity - measure of flexibilty available for scheduling a task
- Unlike EDF, MLF considers execution time of a task

Example 3

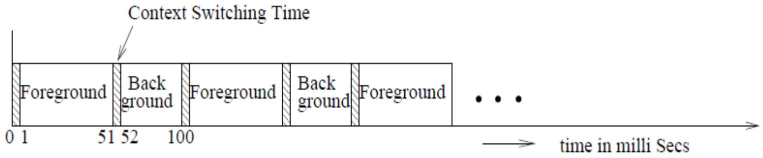
- Consider a real-time system in which tasks are scheduled using foreground-background scheduling. There is only one periodic foreground task Tf=(phif=0,pf=50 msec,ef=100 msec,df=100 msec) and the background task TB = (eB=1000msec). Compute the completion time for background task.
- Solution.
  $\text{ctB} = \frac{1000}{1-\frac{50}{100}}$ msec $= 2000$ msec
  The background task TB would take 2000 msec to complete.

Example 4

- In a simple priority-driven preemptive scheduler, two periodic tasks T1 and T2 and a background task are scheduled. The periodic task T1 has the highest priority and executes once every 20 msec and requires 10 msec of execution time. T2 requires 20 msec of processing every 50 msec. T3 is a background task and requires 100 msec to complete. Assuming that all tasks start at time 0, determine the time at which T3 will complete.
- Solution.
  Total CPU utilization due to foreground tasks $= \frac{10}{20} + \frac{20}{50} = \frac{90}{100}$
  Remaining fraction of time to execute background task, $1 - \frac{90}{100} = \frac{10}{100}$
  Background task gets 1 msec every 10 msec.
  The background task would take $100 \times (\frac{10}{1})$ msec to complete.

Example 5

- Consider a real-time system in which tasks are scheduled using foreground-background scheduling. There is only one periodic foreground task Tf=(phif=0,ef=50 msec,pf=100 msec,df=100 msec) and the background task TB = (eB=1000msec). An overhead of 1 msec on account of every context switch is to be taken into account. Compute the completion time for background task.
- Solution.



Execution time of foreground task increased by two context switches. Execution time of foreground task become 52 msec from 50 msec. ctB $= \frac{1000}{1-\frac{52}{100}}$ msec $=$ 2083.4 msec.

Example 6

- Check whether the set of following periodic real-time tasks is schedulable under EDF on a uniprocessor:
  T1=(e1=10,p1=20), T2=(e2=5,p2=50), T3=(e3=10,p3=35).
- Solution.
  Total CPU utilization achieved $= \frac{10}{20} + \frac{5}{50} + \frac{10}{35} = 0.89$
  $0.89 < 1 \implies$ the task set is EDF schedulable.

Is EDF Really a Dynamic Priority Scheduling Algorithm ?

- If dynamic then able to determine task priority value at any time
- Also able to show how it changes with time
- EDF don't require any priority value computed for any task any time
- EDF has no notion of a priority value for a task
- Tasks are scheduled on proximity to their deadline
- Longer a task waits in queue, higher is probability of being scheduled
- A virtual priority of a task keeps increasing with time until scheduled
- EDF tasks have no priority value
- No priority computations by scheduler at run time or compile time

Naive Implementation of EDF

- Maintain all tasks ready for execution in a queue
- Any freshly arriving task inserted at end of queue
- Every node in queue contain absolute task deadline
- At every preemption point, scan queue to find shortest deadline task
- Very inefficient
- For n tasks
    - Task insertion - O(1) time
    - Task selection/deletion - O(n) time

Better Implementation of EDF - Using Heap

- Maintain all ready tasks in priority queue using a heap
- Tasks are kept sorted according proximity of their deadline
- When task arrives, record inserted into heap in $O(\log_2 n)$ time
- At every scheduling point, next task to run is found at heap root
- Task removal from priority queue takes $O(1)$ time

Best Implementation of EDF - use relative deadline queues

- Restrict number of distinct deadlines for tasks in an application
- When task arrives,
  - compute absolute deadline from release time and relative deadline
- A FIFO queue maintained for each distinct relative deadline of tasks
- Scheduler inserts newly arrived task at end of corresponding queue
- Tasks in each queue are ordered according to absolute deadlines
- To find earliest absolute scheduler search threads of all FIFO queues
- If Q is #priority queues - both search and insert times O(1)

Shortcomings of EDF

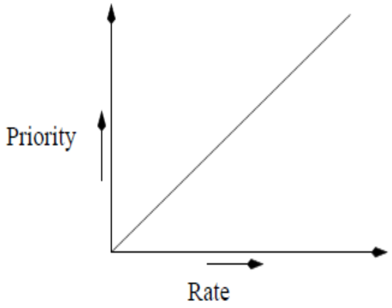
- Transient Overload Problem
  - Transient overload - system overload for very short time
  - Occur when some tasks take more time to complete than original time
  - A task may take longer to complete due to several reasons
  - Enter infinite loop, encounter unusual condition
  - Enter rearly used branch due to abnormal input values
  - Task overshooting its completion time causes other tasks miss deadline
  - Difficult to predict during design
  - Possible prediction
    - Tasks run immediately after task causing transient overload
    - get delayed and might miss deadlines
  - At different times a task might be follwed by different tasks
  - This does not help find which task might miss its deadline
  - Most critical task might miss deadline
    - due to low priority task overshooting its planned completion time
  - EDA does not guarantees most critical task not to miss deadline
    - under transient overload

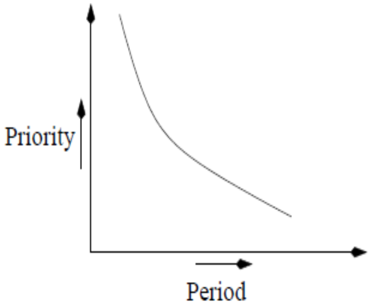
Shortcomings of EDF

- Resource Sharing Problem
  - EDF may incur very high overheads, resource sharing among tasks
    - without making tasks miss their deadlines
- Efficient Implementation Problem
  - Efficient implementation achieves O(1) overhead
  - Assumes number of limited deadlines restricted
  - May be unacceptable in some situations
  - More flexible algorithm keep tasks ordered by deadline in priority queue
  - Arriving task inserted into priority queue in $O(\log_2 n)$ time
  - High run time overhead since most real-time tasks
    - are periodic with small periods and strict deadlines

Rate Monotonic Algorithm (RMA)

- An optimal static priority algorithm for practical applications
- Assigns priority to tasks based on their rates of occurrence
- Lower occurrence rate of a task, lower priority assigned to it
- Highest occurrence rate (lowest period) task accorded highest priority
- Task priority $\propto$ rate, or priority $=$ k/pi, pi period of Ti, k constant

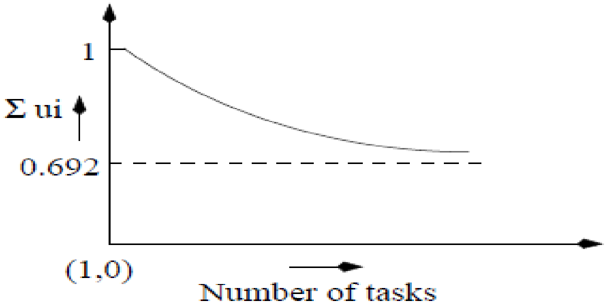




Schedulability Test for RMA

- Check a set of periodic tasks feasibly scheduled on uniprocessor
- Determined from worst-case execution times and periods of tasks
- Worst-case execution times determined by experiment or simulation
- Necessary Condition
  - For n tasks, e1/p1+e2/p2+...en/pn = u1+u2+...+un $\leq$ 1
  - For task Ti, ei worst-case execution time, pi period, ui CPU utilization
  - expresses total CPU utilization $<$ 1
- Sufficient Condition
  - By Liu and Layland in 1973
  - For n tasks, u1+u2+...+un $\leq$ n[2$\hat{(}$1/n)-1]
  - Single task system achieves CPU utilization is 1
  - As number of tasks increases, CPU utilization falls
  - As n$\rightarrow \infty$ utilization stablizes at loge2 $\approx$ 0.692
  - If test passed, a task set guaranteed to be RMA schedulable
  - If failed, a task set may be RMA schedulable

Achievable Utilization for RMA



- If Liu & Layland test passed, RMA scheduling guaranteed for task set
- If Liu & Layland test fails, task set still may be RMA schedulable

Example 7

- Check whether the set of following periodic real-time tasks is schedulable under RMA on a uniprocessor:
  T1=(e1=20,p1=100), T2=(e2=30,p2=150), T3=(e3=60,p3=200).
- Solution.
  Total CPU utilization achieved $= \frac{20}{100} + \frac{30}{150} + \frac{60}{200} = 0.7$
  $0.7 < 1 \implies$ necessary condition for schedulability is satisfied.
  Liu and Layland's test for checking sufficiency condition:
  Maximum achievable utilization $= 3(2^{\frac{1}{3}} - 1) = 0.78$
  $0.7 < 0.78 \implies$ sufficient condition of RMA schedulability is satisfied.

Example 8

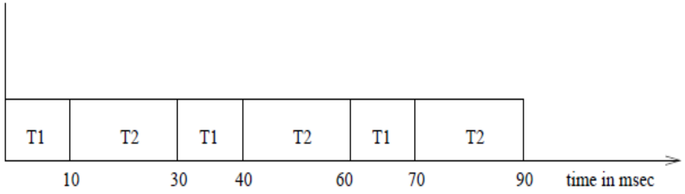
- Check whether the set of following periodic real-time tasks is schedulable under RMA on a uniprocessor:
  T1=(e1=20,p1=100), T2=(e2=30,p2=150), T3=(e3=90,p3=200).
- Solution.
  Total CPU utilization achieved $= \frac{20}{100} + \frac{30}{150} + \frac{90}{200} = 0.85$
  $0.85 < 1 \implies$ necessary condition for schedulability is satisfied.
  Liu and Layland's test for checking sufficiency condition:
  Maximum achievable utilization $= 3(2^{\frac{1}{3}} - 1) = 0.78$
  $0.85 \nleq 0.78 \implies$ the task set is not RMA schedulable.

Theorem 3 - Lehockzy's test

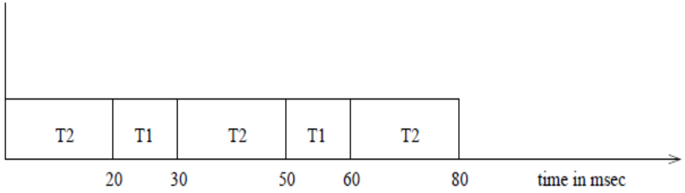
- A set of periodic real-time task is RMA schedulable under any task phasing, if all tasks meet their respective first deadlines under zero phasing.
- If Liu and Layland's test fails then go for Lehockzy's test
- Worst case response time occurs when in phase with higher priority
- In RMA when higher priority task ready, lower priority tasks wait
- When higher priority in phase with lower priority task
- More higher priority task instances during execution of lower

An Example

- Higher priority T1=(10,30) in phase with lower priority T2=(60,120)
- Response time of T2 is 90 msec
- When T1 has a 20 msec phase, response time is 80 msec
- If a task meets deadline under zero phasing, it meet all deadlines



(a) T1 is in phase with T2

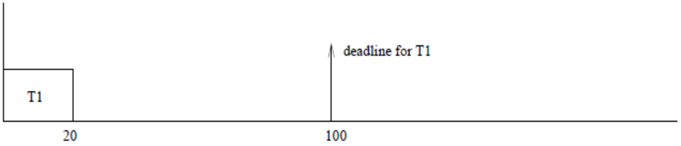


(b) T1 has a 20 msec phase with respect to T2

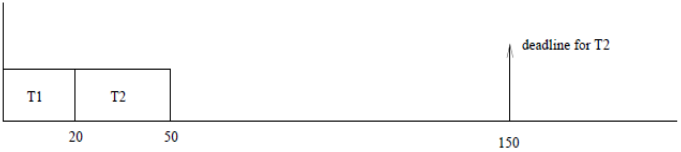
Example 9

- Check whether the set of following periodic real-time tasks is schedulable under RMA.
  T1=(e1=20,p1=100), T2=(e2=30,p2=150), T3=(e3=90,p3=200).
- Solution.
  Apply Lehockzy's test
  For T1: e1 $<$ p1 holds since 20msec$<$30msec. Meets first deadline.
  For T2: T1 higher priority with 0 phasing. Occur before T2 deadline.
  20+30=50msec$<$150msec $\Longrightarrow$ (e1+e2)$<$p2. T2 meets first deadline.
  For T3: $(2 \times 20 + 2 \times 30 + 90) = 190 < 200 \Longrightarrow$ (2e1+2e2+e3)$<$ p3
  T1 and T2 occur twice within first deadline of T3.
  T3 meets deadline.
  Task set is RMA schedulable.

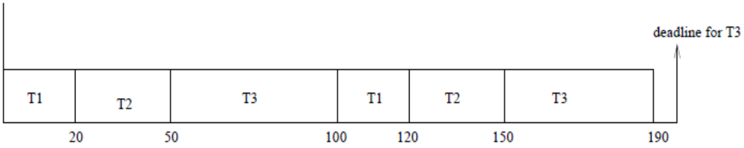
Example 9 - Checking Lehockzy's Criterion



(a) T1 meets its first deadline



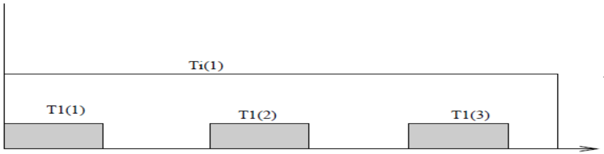
(b) T2 meets its first deadline



(c) T3 meets its first deadline

Formal Expression for Lehockzy's criterion

- Let {T1,T2,...,Ti} be task sets to be scheduled
- Tasks in descending order of priority, pri(T1)>pri(T2)> $\cdots$ >pri(Ti)



- Ti arrives at time instant 0
- During first instance of Ti, three instances of T1 occured
- Each time T1 occurs, Ti waits since pri(T1)>pri(Ti)
- Number of times T1 occurs within a single instance of Ti is ceil(pi/p2)
- Total execution time due to T1 before deadline of Ti is ceil(pi/p2)*e1
- Total execution time of Ti's higher priority tasks for which Ti waits $\sum$ceil(pi/pk)*ek, for $1 \leq$ k $\leq$ i-1

Formal Expression for Lehockzy's criterion

- Ti meets its deadline if
  ei $+ \sum$ceil(pi/pk)*ek$\leq$pi, for $1 \leq$ k $\leq$ i-1 and pi $=$ di
- If pi$<$di, then ei $+ \sum$ceil(di/pk)*ek$\leq$di, for $1 \leq$ k $\leq$ i-1
- Even if above not satisfied, some possibility of task set schedulable
- Above expression considers zero phasing - worst case
- Consider a task set where Ti fails to meet the condition
- This makes task set unschedulable
- Task misses deadline when in phase with all higher priority task
- When the task have non-zero with some higher priority tasks
  - the task might meet first deadline contrary to condition failure

Example 10

Consider the following set of three periodic real-time tasks: T1 = (10,20), T2 = (15,60), T3 = (20,120) to be run on a uniprocessor. Determine whether the task set is schedulable under RMA.

- Solution. Test sufficiency (Liu & Layland test) for RMA schedulability
  Task set RMA schedulable if $\sum$ui≤ 0.78, $\sum$ui$= \frac{10}{20} + \frac{15}{60} + \frac{20}{120} = 0.91$
  $0.91 > 0.78 \implies$ given task set fails Liu & Layland test
  Lehockzy's test. T1, T2, T3 ordered in decreasing order of priorities.
  For T1: e1=10<d1=20 $\implies$ T1 meet first deadline.
  For T2: $15 + \left\lceil \frac{60}{20} \right\rceil \times 10 \leq 60 \implies 45 \leq 60$. T2 meet first deadline.
  For T3: $20 + \left\lceil \frac{120}{20} \right\rceil \times 10 + \left\lceil \frac{120}{60} \right\rceil \times 15 = 110 \leq 120$.
  T3 meet first deadline.
  All tasks meet first deadline.
  Task set is RMA schedulable according to Lehockzy's results.

Example 11

RMA is used to schedule a set of periodic hard real-time tasks in a system. Is it possible in this system that a higher priority task misses its deadline, whereas a lower priority task meets its deadlines ? Give an example involving three tasks scheduled using RMA where the lower priority task meets all its deadlines whereas the higher priority task misses its deadline.

- Solution. Yes. It is possible under RMA deadline higher priority task misses deadline while lower priority task meets deadline.
  Consider a task set: T1=(e1=15msec,p1=20msec), T2=(e2=6msec,p2=35msec), T3=(e3=3msec,p3=100msec).
  pri(T1)>pri(T2)>pri(T3). T1,T2,T3 in decreasing order of priorities.
  According to Lehockzy's test T3 meets deadline since,
  e3+(ceil(p3/p2)*e2)+(ceil(p3/p1)*e1)=3+3*6+5*15=96$\leq$100.
  T2 does not meet deadline since,
  e2+(ceil(p2/p1)*e1)=6+2*15=36>35 deadline of T2.

Achieving higher CPU Utilization

- Liu & Layland's criterion restricts CPU utilization under RMA to 0.69
- Experimentally maximum utilization independent tasks $\approx$ 88%
- Higher maximum utilization for harmonic tasks
- If task periods are harmonically related, 100% utilization
- Task periods in task set are harmonically related if
    - for any Ti and Tk when pi>pk
    - pi expressed as n×pk for some n>1
    - Example: T1=(5ms,30ms),T2=(8ms,120ms),T3=(12ms,60ms)

Theorem 4

For a set of harmonically related tasks HS=$\{$Ti$\}$, the RMA schedulability criterion is given by $\sum$ui$\leq$1, 1$\leq$i$\leq$n.

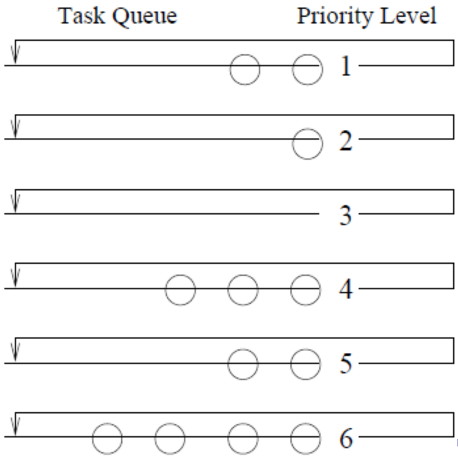
- Proof: T1,T2,. . . ,Tn are tasks in a given task set.
  Tasks T1,T2,. . . ,Tn arranged in increasing order of periods.
  For any i and j, pi<pj when i<j.
  Ti meets deadline if ei+$\sum$ceil(pi/pk)*ek$\leq$pi, 1$\leq$k$\leq$i-1.
  For harmonically related task set, pi=m*pk for some m.
  Using ceil(pi/pk)=pi/pk, ei+$\sum$(pi/pk)*ek$\leq$pi, 1$\leq$k$\leq$i-1.
  For Ti=Tn, en+$\sum$(pn/pk)*ek$\leq$pn, 1$\leq$k$\leq$n-1.
  Dividing both sides by pn,
  $\sum$ek/pk$\leq$1, 1$\leq$k$\leq$n $\implies$ $\sum$ui$\leq$1, 1$\leq$i$\leq$n.

Advantages and Disadvantages of RMA

- RMA is simple, efficient and optimal
- Unlike EDF, RMA requires very few data structures
- RMA possesses good transient overload handling capability
    - When a lower priority task does not complete in planned time
    - Cannot make any higher priority task to miss deadline
    - Preempt any executing lower priority task
    - RMA stable under transient overload
- Difficult to schedule aperiodic and sporadic tasks
- RMA not optimal when task periods and deadlines differ

Multi-Level Feedback Queue

- Most real-time operating systems support static priority level tasks
- Tasks having real-time priority levels - multilevel priority queue
- Schedule tasks in single level - time slicing, round robin or FIFO
- Equal priority tasks - resource sharing protocols



Deadline Monotonic Algorithm (DMA)

- RMA not optimal for periodic tasks when di$\neq$pi for some tasks
- DMA more proficient that RMA for di$\neq$pi
- DMA a variant of RMA
- Assigns priorities to tasks based on deadlines (RMA priority periods)
- Higher priorities to tasks with shorter deadlines
- Relative deadlines proportional to period $\implies$ RMA & DMA identical
- Arbitrary relative deadlines $\implies$ DMA more proficient
- RMA always fails when DMA fails

Example 12

Is the following task set schedulable by DMA? Also, check whether it is schedulable using RMA. T1=(e1=10msec,p1=50msec,d1=35msec), T2=(e2=15msec,p2=100msec,d2=20msec), T3=(e3=70msec,p3=200msec,d3=200msec).

- Solution. Check RMA schedulability by Lehockzy's criterion.
  Tasks ordered in descending order of priorities, T1, T2, T3.
  For T1: 10msec<35msec $\implies$ T1 meet first deadline.
  For T2: $(10+15) \nleq 20 \implies$ T2 miss first deadline.
  Given task set cannot be feasibly scheduled under RMA.
  Check schedulability usng DMA.
  Priority ordering of tasks: Pr(T2)>Pr(T1)>Pr(T3).
  For T2: 15msec<20msec $\implies$ T2 meet first deadline.
  For T1: $(10+15)<35 \implies$ T1 meet first deadline.
  For T3: $(70+30+40)<200 \implies$ T3 meet first deadline.
  Given task set can be feasibly scheduled under DMA.

Context Switching Overhead

- Each task incurs atmost two context switches under RMA
- One, when it runs preempting currently running task
- Other, when it completes
- One context switching overhead, if no tasks preempted
- If task arrives when CPU idle, or running higher priority task
- Worst-case context switching overload considers two context switches
- For simplicity context switching time is constant, c milliseconds
- Effect - for Ti, ei is at most ei+2*c
- Schedulability computations need to replace ei by ei+2c for each Ti

Example 13

Check whether the following set of periodic real-time tasks schedulable under RMA on a uniprocessor: T1=(e1=20msec,p1=100msec), T2=(e2=30msec,p2=150msec), T3=(e3=90msec,p3=200msec). Assume that context switching overhead does not exceed 1msec and is to be taken into account in schedulability computations.

- Solution. Task execution time increased by two context switches.
  Utilization = u1+u2+u3=$\frac{22}{100} + \frac{32}{150} + \frac{92}{200} = 0.893$
  $0.893 > 0.78 \implies$ task set not RMA schedulable by Liu & Layland test.
  For T1: 22msec<100msec $\implies$ T1 meets first deadline.
  For T2: 22*2+32<150msec $\implies$ T2 meets first deadline.
  For T3: 22*2+32*2+92<200msec $\implies$ T3 meets first deadline.
  Given task set can be feasibly scheduled under RMA considering context switching overhead.

Self Suspension

- A task might cause its self suspension when
    - it performs input/output operations, or
    - it waits for some events/conditions to occur
- When a task self suspends itself
    - operating system removes it from ready queue
    - places it in the blocked queue
    - takes up eligible task for scheduling
- Self suspension introduces an additional scheduling point
- In event-driven scheduling, scheduling points defined by
    - task completion
    - task arrival
    - self-suspension events

Effect of Self Suspension on Schedulability

- Consider a set of periodic real-time tasks {T1,T2,...,Tn}
- Arranged in increasing (decreasing) order of priorities (periods)
- bi - worst case self suspension time of Ti
- bti - delay Ti incurs due to self suspension (own & all higher priority)
- bti = bi + $\sum$ min(ek,bk), 1≤k≤i-1
- Self suspension of higher priority Tk affect
  - response time lower priority Ti
  - ek execution time of Tk
  - if ek<bk
- Worst case delay might occur when
  - higher priority task after self suspension starts execution
  - exactly at time instant lower priority task would have o.w. executed
- After self suspension, execution higher priority overlaps with lower
- Which would o.w. not have overlapped
- If ek>bk, self suspension of a higher priority task
  - delay a lower priority task by at most bk
  - max. overlap period of execution of higher priority restricted to bk

Effect of Self Suspension on Non-preemptable Tasks

- More severe
- Every time a processor self suspends itself, it loses processor
- Blocked by non-preemptive low priority after self suspension completes
- In a non-preemptable scenario a task incurs delays due to
  - self-suspension of itself and higher priority tasks, and
  - delay caused due to non-preemptable lower priority tasks
- Task can't delayed due self suspen. of low priority non-preempt. task
- In Liu & Layland schedulabilty criterion substitute ei by (ei+bti)
- Lehockzy criterion for task completion time
  $ei+bti+\sum ceil(pi/pk)*(ek \leq pi)$, $1 \leq k \leq i-1$

Example 14

Consider the following set of three periodic real-time tasks:
T1 = (e1=10 msec,p1= 50 msec), T2 = (e2=25 msec,p2=150 msec), T3 = (e3=50 msec,p3=200 msec). Assume that the self suspension times of T1, T2 and T3 are 3 msec, 3 msec, and 5 msec, respectively. Determine whether the tasks would meet their respective deadlines, if scheduled using RMA.

- Solution. Lehockzy's test. T1, T2, T3 ordered in decreasing order of priorities. For T1: $(10+3)<50 \implies$ T1 meets first deadline.
  For T2: (25+6+10*3)<150. T2 meets first deadline.
  For T3: (50+11+(10*4+25*2))<200. T3 meets first deadline.
  All tasks meet first deadline.
  Task set is RMA schedulable according to Lehockzy's results.

Self Suspension with Context Switching Overhead

- In a fixed priority preemptable system
  - each task preempts atmost once on other if no self suspension
- Each task atmost two context switches
  - when it starts
  - when it completes
- Self suspension causes atmost two additional context switches
- Task allowed two self suspensions, four additional context switches
- Maximum context switch time c
- Increases worst case execution time of Ti from ei to ei+4*c
- Execution time of Ti with single self suspension is ei+4*c
- Extended fot two, three, or more self suspensions

RMA in Practical Situations

- Handling Critical Tasks with Long Periods
  - Applications where task criticalities differs from task priorities
  - Situation a very critical task has higher period than lower critical tasks
    - miss deadline due to transient overload low critical-higher priority task
  - If critical task priority raised then
    - RMA schedulability would not hold, and
    - determing schedulability extremely difficult
  - A solution is period transformation technique by Sha and Raj Kumar
    - a critical task logically divided into small subtasks
    - Critical task Ti split into k subtasks
    - Each subtasks have period pi/k, deadline di/k, execution time ei/k
    - Net effect of splitting Ti into {Ti1,. . . ,Tik} to raise priority of Ti
    - Each subtask of Ti is Tij=<ei/k,pi/k,di/k>
    - Splitting into k parts done virtually at conceptual level
    - Period transformation raises priority, doesn't make RMA analysis invalid
    - Due to raising priority RMA results do not hold accurately
    - Utilization due to all higher priority tasks k*(ei/k)/(pi/k)=k*ei/pi
    - Actual utilization ei/pi
    - Splitted tasks appear unschedulable by RMA may be schedulable

Handling Aperiodic and Sporadic Tasks

- Under RMA difficult to assign high priority values to sporadic tasks
- Burst of sporadic tasks arrive overload system, tasks miss deadlines
- RMA schedulability not applicable for period sporadic high priority
- Aperiodic and sporadic usually assigned very low priority values
- Practical situation tasks handling emergency conditions
    - are time bound and critical
    - may require higher priority value for sporadic task
    - aperiodic server technique is used

Aperiodic Server

- Handles aperiodic and periodic tasks
- Selects them at appropriate times and passes them to RMA scheduler
- Difficult to analyze schedulability of aperiodic sporadic tasks
- Server deposits a “ticket”
- “ticket” replenished at expiration of replenishment period
- When aperiodic event occurs server checks ticket availability
- If available system immediately passes arriving tasks to scheduler
- Creates another ticket on basis of its policy
- Makes aperiodic tasks more predictable, suitable for RMA analysis
- Based on ticket creation policies, two types of aperiodic servers
    - deferrable and sporadic
- Sporadic server results higher schedulable utilization
- More easy analysis, more complex to implement

Defferable Server

- Deferrable server tickets are replenished at regular intervals
- Completely independent of actual ticket usage
- Aperiodic task arrives immediate system process if enough tickets
- Wait till tickets replenished if not available
- Bursts tasks to scheduler, tickets accumulated no task over duration
- Simpler to implement
- Deviates from RMA strict periodic execution model
- Leads to conservative system design and low processor utilization

Sporadic Server

- Replenishment time depends on exact ticket usage time
- As soon as ticket used, system sets timer to replace used tickets
- Guarantees minimum separation between two sporadic task instances
- Consider sporadic periodic task as periodic for schedulability analysis
- Assigns a suitable priority to it
- In some periods aperiodic or sporadic tasks may not rise
- Leads to unavoidable CPU idling
- Aperiodic server
  - Overcomes non-determinism of aperiodic and sporadic tasks
  - Helps in fast approximation as periodic tasks
    - with period equal to token generation time

Coping with Limited Priority Levels

- #real-time tasks exceeds #priority levels supported by OS
- Occurs in small embedded systems, limited memory size
- RTOS normally do not support too many priority levels
- Few out of all priority levels are marked as real-time levels
- In commercial OS have 8 to 256 priority levels
- #priority level $\propto$ #feedback queues of OS
- Increases memory requirement
- Increases processing overhead at every scheduling point
- At every scheduling point
  - OS selects highest priority task by scanning all priority queues
  - Scanning incur considerable computational overhead
- Restricted #priority levels
  - reduces OS overload
  - improves response time

Coping with Limited Priority Levels

- When more real-times tasks than #priority levels from OS
- More than one task share a single priority value
- Results lower processor utilization than predicted by RMA expression
- Let SP(Ti) be set of all tasks sharing single priority value with Ti
- For a set of tasks to be schedulable
  - $A+B+C+D \leq pi$
  - $A=ei$, $B=bit$, $C=\sum ek$, where $Tk \in SP(Ti)$
  - $D=\sum ceil(pi/pk) * ek$, where $1 \leq k \leq i-1$
- Term C needed as FCFS scheduling policy among equal priority tasks
- Tk blocked by an equal priority task only once
- Unlike higher priority tasks block Tk at their arrival
- An equal priority task unlike higher priority blocks a task only once

Why a task blocked by an equal priority task atmost once?

- Let Ti and Tj share same priority level
- Tj has a much shorter period compared to Ti
- A task instance Ti blocked by Tj only once for duration of Tj
- Even Tj has a shorter period than Ti
- Because once Tj completes , Ti gets to execute
- Ti complete execution since arriving Tj instance can't preempt Ti
- Instance of Ti blocked by equal priority Tj only once

Priority Assignment to Tasks

- More real-time tasks with distinct deadlines than #priority levels
- Some tasks have to share same priority value
- Priority assignment method affect achievable processor utilization
- Assigning priority to tasks
  - Uniform Scheme
  - Arithmetic Scheme
  - Geometric Scheme
  - Logarithmic Scheme

Uniform Scheme

- All tasks are uniformly divided among available priority levels
- Achieved when #priority levels divides #tasks to be scheduled
- If divison not possible, more tasks share lower priority levels
- Higher priority values are shared by lesser number of tasks
- For better schedulability
- For N number of tasks and n priority levels
    - number of tasks are assigned to each level, and
    - rest are distributed among lower priority levels

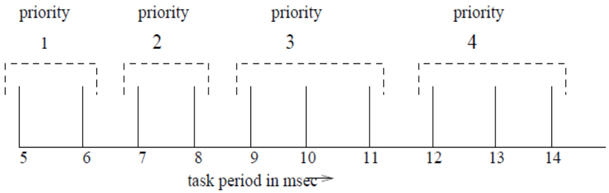
Example 15

In a certain applicable being developed, there are 10 periodic real-time tasks T1, T2,. . . ,T10 whose periods are: 5, 6,. . . ,14 msec, respectively. These tasks are to be scheduled using RMA. However, only four priorities levels are supported by the underlying operating system. In this operating system, the lower the priority value, the higher is the priority of the task. Assign suitable priority levels to tasks using the uniform assignment scheme for scheduling the tasks using RMA.

- Solution. Number of priority does not divide number of tasks.
  Some priority values assigned more tasks than others.
  Tasks in ascending order of periods (descending order of priorities).
  First, uniformly divide tasks among priority levels.
  Each level is assigned two tasks each.
  Rest are distributed among lower priority levels.
  A possibility, two lower priority levels assigned three tasks each.

Example 15 (continued)

- A suitable task assignment scheme
  - Priority Level 1: T1, T2
  - Priority Level 2: T3, T4
  - Priority Level 3: T5, T6, T7
  - Priority Level 4: T8, T9, T10



Arithmetic Scheme

- Number of tasks assigned different priority levels form an
  - Arithmetic Progression
- r tasks having shortest periods assigned to highest priority level
- 2r tasks are assigned next highest priority level, and so on
- N be total number of tasks
  - Then, $N = r + 2r + 3r + 4r + \ldots + nr$
  - n is total number of priority levels

Arithmetic Scheme

- Number of tasks assigned different priority levels form a
  - Geometric Progression
- r tasks having shortest periods assigned to highest priority level
- kr^2 tasks are assigned immediately lower priority, and so on
- N be total number of tasks
  - Then, $N = r + kr^2 + kr^3 + kr^4 + \ldots + k$(r pow n)
  - n is total number of priority levels

Logarithmic Scheme

- Also known as *logarithmic grid assignment scheme*
- Shorter period (higher priority) tasks alloted distinct priority levels
  - as much as possible
- Many lower priority tasks clubbed together at same priority levels
  - without causing any problem to schedulability of higher priority tasks
- To achieve logarithmic grid assignment
  - task first arranged increasing order of priorities
- For priority allocation
  - range of tasks divided into sequence of logarithmic intervals
- Tasks assigned priority levels based on logarithmic interval they belong
- If pmax is max period and pmin is min period among tasks then
  - r = (pmax/pmin) pow (1/n)
  - n is total number of priority levels

Logarithmic Scheme

- Tasks with periods upto r assigned to highest priority
- Tasks with periods in range r to r^2 assigned to next highest priority
- Tasks with periods in range r^2 to r^3 assigned to next highest priority
- and so on
- Simulation experiments show logarithmic priority scheme works
  - very well for practical problems
- Works well only when task periods uniformly distributed over interval
- If most task periods clustered over small part of interval, and
- other tasks are sparsely distributed in rest of interval then
- logarithmic scheme may yield poor results

Example 16

Consider an operating system supporting only four priority levels. An application with 10 periodic real-time tasks are to be scheduled on this operating system using RMA. It is also known that of the given tasks, the largest period is 10,000 msec and the shortest period is 1 msec. Other task periods are distributed uniformly over this interval. Assign the tasks to priority levels using the logarithmic grid assignment scheme.

- Solution. r $= \left(\frac{10000}{1}\right)^{\frac{1}{4}} = 10$
  Tasks with periods in range 1 msec to 10000 msec assigned to highest priority level. Tasks with periods in range 11 to 100 msec assigned to next lower priorities level, and so on.

Dealing with Task Jitter

- Jitter is magnitude of variation in arrival or completion times
- Arrival time jitter = latest arrival time - earliest arrival time
- Completion time jitter = latest comple. time - earliest comple. time
- Presence of small amount of jitter is unavoidable
- All physical clocks show small amount of skew
- Completion jitter by RMA - schedules tasks at earliest opportunity
- Task response time depends on #higher priority tasks arrive or wait
- Small jitters do not cause any problem
- As long as arrival and completion times tasks are in tolerance limits
- Applications might require to minimize jitter as much as possible

Techniques to handle tasks with tight completion jitter

- Method 1
  - If only one or two tasks have tight jitter requirements
  - Then these tasks are assigned very high priority
  - Works well for very small number of tasks
  - Used in applications in which tasks are barely schedulable
  - May result in some tasks missing their respective deadlines
- Method 2
  - If jitter minimized for an applicable that is barely schedulable
  - Each tasks needs to be split into two
  - One computes output but does not pass it on,
  - one which passes output on
  - Involves setting second tasks's priority to very high values and
  - its period to be same as that of first
  - Action scheduled will run on cycle behind schedule
  - But all tasks will have tight completion time jitter

Thank you