Scheduling Real-Time Tasks in Multiprocessor and Distributed Systems
Real-Time Systems Design (CS 6414)

Sumanta Pyne

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

February 26, 2021

Overview

- Multiprocessor & distributed systems widely for real-time applications
- Recent price drop of these systems
- Dual processor architectures available at 50-60K INR even cheaper
- Distributed platform - networked PCs are common
- Better for faster response times and fault tolerance
- Distributed processing suitable for geographically distributed places
- Automated petroleum refinery - plant spread over geographic area
- Scheduling tasks more difficult than on a uniprocessor
- Uniprocessor - optimal schedule independent tasks in polynomial time
- Multiprocessor & Distributed - finding optimal schedule is NP-Hard

Multiprocessors & Distributed Systems

- Multiprocessors tightly coupled - shared physical memory
- Distributed Systems loosely coupled - no shared memory
- Interprocess communication (IPC) expensive in tightly-coupled syst.
- IPC time ignored compared to task execution times
- Inter-task communication - read/write on shared memory
- Distributed system IPC times comparable to task execution times
- Multiprocessor uses centralized scheduler/dispatcher
- Centralized scheduler maintain task state in centralized data structure
- High communicational oveheads - update data structure state change

Scheduling on Multiprocessors & Distributed Systems

- Two subproblems
  - task allocation to processors
  - scheduling tasks on individual processors
- Task assignment problem
  - how to partition a set of tasks
  - then how to assign these to processors
  - Static - task allocation to nodes is permanent
  - Dynamic - tasks are assigned to nodes as they arise
    - different task instance allocated to different nodes
- Scheduling tasks on individual processors
  - Scheduling problem reduced to uniprocessor scheduling

Task allocation in Multiprocessors and Distributed Systems

- An NP-Hard Problem
- Exponential time required to determine an optimal schedule
- Usage of heuristic algorithms
- Task allocation algorithms - static and dynamic
- Static algorithms
  - all tasks partitioned into subsystems
  - each subsystem is assigned a seperate processor
- Dynamic algorithms
  - tasks ready for execution placed in a common priority queue
  - dispatched to processors for execution as processors become available
  - different periodic task instances execute on different processors
- Most hard real-time systems till date are static in nature
- Dynamic real-time system make more efficient resource utilization

Multiprocessor Static Task Allocation

- Not applicable for distributed systems
- IPC time is same as memory access time due to shared memory
- Utilization Balancing algorithm
  - Maintains tasks in a queue in increasing order of utilizations
  - Removes tasks one by one from head
  - Allocates them to least utilized processor each time
  - To balance utilization of different processors
  - Perfectly balanced systems utilization ui per processor equals overall
  - For task set STi assigned to processor Pi, ui=$\sum$utj, j$\in$STi
  - utj utilization due to Tj
  - PR set of all processors, then total utilization is $\sum$ui, Pi$\in$PR
  - Difficult to achieve perfect balancing of utilizations, i.e. ui=u' for Pi
  - Simple heuristic gives suboptimal results
  - Objective of good utilization balancing - Minimize $\sum$ |u'-ui|, 1$\leq$i$\leq$n
  - n #processors, u' average utilization of processors, ui utilization of Pi
  - Suitable when #processors is fixed
  - Used when tasks at individual processors are scheduling using EDF

Next-Fit Algorithm for RMA

- A task set is partitioned, each scheduled on a uniprocessor using RMA
- Attempts to use as few processors as possible
- Classifies different tasks into few classes based on task utilization
- One or more processors assigned to each class of tasks
- Task with similar utilization values scheduled on same processor
- Policy for utilization based task classification
  - If tasks divided to m classes, Ti$\in$ class j, $0 \leq j \leq m$
- Partitioning tasks of a system into four classes
  - Class 1: $(2^{\frac{1}{2}} - 1) < C_1 \leq (2^{\frac{1}{1}} - 1)$
  - Class 2: $(2^{\frac{1}{3}} - 1) < C_2 \leq (2^{\frac{1}{2}} - 1)$
  - Class 3: $(2^{\frac{1}{4}} - 1) < C_3 \leq (2^{\frac{1}{3}} - 1)$
  - Class 4: $0 < C_4 \leq (2^{\frac{1}{4}} - 1)$
- Utilization grid for different classes
  - class 1:(0.41,1), class 2:(0.26,0.41), class 3:(0.19,0.26), class 4:(0,0.19)
- Higher task utilization values are coarser compared to lower
- Grid size: class 1 tasks is 1-0.41=0.59, class 3 tasks is 0.7
- Simulation - next-fit algorithm atmost 2.34$\times$ optimum #processors

Example 1

The following table shows the execution times (in msec) and periods in (msec) of a set of 10 periodic real-time tasks. Assume that the tasks need to run on a multiprocessor with four processors. Allocate the tasks to processors using the next fit algorithm. Assume that the individual processors are to be scheduled using RMA algorithm.

Task Set

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
e_i	5	7	3	1	10	16	1	3	9	17
p_i	10	21	22	24	30	40	50	55	70	100

Solution

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
e_i	5	7	3	1	10	16	1	3	9	17
p_i	10	21	22	24	30	40	50	55	70	100
u_i	0.5	0.33	0.14	0.04	0.33	0.4	0.02	0.05	0.13	0.17
Class	1	2	4	4	2	2	4	4	4	~~3~~ 4

Bin Packing for EDF

- Allocates tasks to processors
- Tasks on individual processors scheduled using EDF
- Tasks assigned to Pi such that ui$\leq$1
- Formulate task allocation as bin packing problem
  - Given n periodic real-time tasks
  - Individual processors scheduled using EDF
  - #bins necessary = $\sum$ui, 1$\leq$i$\leq$n
  - Bin packing is NP-complete
- First-fit random bin packing algorithm
  - Tasks selected randomly
  - Assigned processors arbitrarily as long as ui$\leq$1
  - Atmost 1.7$\times$ optimum #processors required
- First-fit decreasing bin packing algorithm
  - Tasks sorted in non-increasing order of CPU utilization in ordered list
  - Task selected one by one from ordered list
  - Assigned to bin(processor Pi) to which it fits first (i.e. ui$\leq$1)
  - Simulations show #processors required is 1.22$\times$ optimal #processors

Dynamic Allocation of Tasks

- Applications where tasks arrive sporadically at different nodes
- Dynamic algorithms needed to handle such tasks
- Assume any task can be executed on any processor
- Dynamic solutions naturally distributed
- No central allocation policy running on some processor
- No preallocation of tasks to processors
- Assign tasks when they arise
- Task allocation made on instantaneous load positions of nodes
- Achievable schedulable utilization better than static approaches
- High run time overhead - allocator component running at each node
- Static allocation - task assigned permanently during initialization
- No runtime overhead for static
- Dynamic ineffective if task bound to single or subset of processors

Focussed Addressing and Bidding

- Every processor maintain status table and system load table
- Status table contains
  - information of tasks committed to run
  - execute time of tasks
  - periods of tasks
- System load table contains latest load information other processors
- Determine surplus computing capacity available
- Time axis divided into windows - intervals of fixed duration
- At end of each window each processor broadcasts to all other
- Fraction of computing power currently free for next window
- Fraction of next window with no committed tasks
- On receiving this every processor updates system load table
- When task arises, node checks whether process it locally
- If possible, it updates its status table
- Otherwise, looks out for a process can offload task

Focussed Addressing and Bidding

- Processor finds suitable processor cosulting system load table
- Determines least loaded processors to accommodate the task
- It then sends request for bids (RFBs) to these processors
- While looking for processor an overloaded processor
- Checks surplus information and selects a *focussed* processor
- information in system load table may be out of date
- Obsolete information problem
- Solution - send RBFs only if it determines that task complete in time
- High communication overhead to maintain system load tables
- Larger window size lower overhead

Buddy Algorithm

- Aims to overcome high communication overhead
- Processor states - underloaded and overloaded
- Pi underloaded if ui$<$Th
- Pi overloaded if ui$\geq$Th
- Processor broadcasts - change in state
- Broadcasts to limited subset *buddy* set

Fault-Tolerant Scheduling of Tasks

- Task scheduling can be used for fault-tolerance in real-time systems
- An effective technique
- Requires very little redundant hardware resource
- Achieved by scheduling additional ghost copies with primary task copy
- Ghost copies (clones)
  - may not be identical to primary copy
  - copies stripped down versions
  - can be executed in shorter durations than primary
- Ghost copies of different tasks can be overloaded on same slot
- Success execution of primary, deallocates corresponding backup

Clocks in Distributed Real-Time Systems

- Clocks used for determining timeouts and time stamping
- Timeouts
  - determine failure of tasks due to deadline miss
  - indicate transmission faults or delays, or non-existent receivers
- Time stamping
  - used in message communication among tasks
  - sender includes current time along a message
  - gives age of message
  - used for ordering purposes
  - relies on good real-time clock services
- Distributed system has one clock at each node
- Different clocks diverge
- Two clocks of run exactly at same speed is impossible

Clocks in Distributed Real-Time Systems

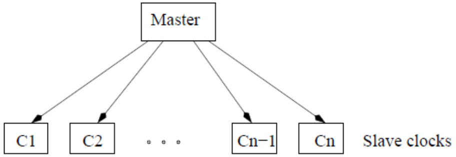
- Lack of synchrony among clocks expressed as *clock slew*
- *clock slew* determines attendant drift of clocks with time
- Lack of synchrony and drift - meaningless time stamping and timeout
- Sychronized clocks for meaningful time stamping and timeout
- Synchronization very important in distributed real-time systems

Clock Synchronization

- Makes all clocks in network agree on their time values
- Different clocks agree on some time, differ from world time standard
- World time standard called Univeral coordinated time (UTC)
- UTC based on international atomic time (TAI)
- TAI maintained at Paris by averaging atomic clocks around the world
- UTC signals used through GPS receivers and specialized radio stations
- Internal clock synchronization - all clocks synchronized wrt one clock
- External clock sych - a set of clocks synchronized with external clock
- Two main approaches for internal clock synchronization
  - centralized clock synchronization
  - distributed clock synchronization

Centralized Clock Synchronization

- One of the clocks is designated as master clock or *time server*
- Other clocks of the system are slaves
- Slaves are kept in synchronization with master clock
- Slave clocks C1,. . . ,Cn are to synchronized with master clock



- Server broadcasts its time to all clocks after every ΔT time interval
- On receiving a broadcast, slaves set clock as per time at master clock
- ΔT should be chosen carefully
- If ΔT too small
  - frequent broadcasts from master
  - good synchronization between slaves and master
  - high communication overhead
- If ΔT too large, clocks may drift too much apart

Centralized Clock Synchronization

- Assume maximum rate of drift between to individual clocks is ρ
- Clock manufactures provides ρ as a specification parameter of a clock
- ρ is unit less, it measures drift (time) per unit time
- Suppose clocks are resynchronized after every ΔT interval
- Ignoring communication time required for broadcast
- Once broadcast received, clocks set to received time instantly
- Drift of any clock from master clock is bounded by $\rho\Delta T$
- Maximum drift between any two clocks is limited to $2\rho\Delta T$
- In reality it takes a finite amount pf time to set a clock
- Suitable communication time and clock setting time required
- Otherwise, synchronized time become slower wrt external clock
- Slaves still remain synchronized within a specified bound
- Very difficult to compensate these in practical systems

Example 2

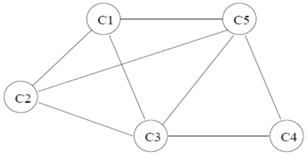
Assume that the drift rate between any two clocks is restricted to $\rho = 5 \times 10^{-6}$. Suppose we want to implement a synchronized set of six distributed clocks using the central sychronization scheme so that the maximum drift between any two clocks is restricted to $\epsilon = 1$ msec at any time, determine the period with which the clocks need to be resynchronized.

- Solution. The maximum drift rate between any two arbitrary clocks when the clocks are synchronized using a central time server with a resynchronization interval of ΔT is given by $2\rho\Delta T < \epsilon$. Therefore, the required resynchronization interval ΔT can be expressed as:
  $\Delta T < \frac{1\times10^{-3}}{5\times10^{-6}\times2}$ sec $= \frac{10^{-3}}{10^{-5}}$ sec $= \frac{1}{10^{-2}}$ sec $= 100$ sec.
  Therefore, resynchronization period must be less than 100 sec.

Distributed Clock Synchronization

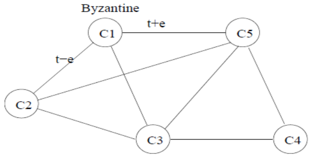
- Centralized clock synchronization susceptible to single point failure
- Any failure of master clock causes breakdown of synchronization
- Distributed clock synchronization - no master clock wrt all slaves
- All clocks periodically exchange clock readings among themselves
- Based on received time, clocks computes and sets synchronized time
- Possible that some clocks are bad or become bad during operation
- Bad clocks exhibit larger drifts than specified tolerance
- Bad clocks may even stop keeping time

Distributed Clock Synchronization



- Bad clocks can be identified and taken care of during synchronization
- By rejecting time values of any clock larger than specified bound
- Usage of Byzantine clocks
- A Byzantine clock is a two face clock
- Transmits different values to different clocks at same time

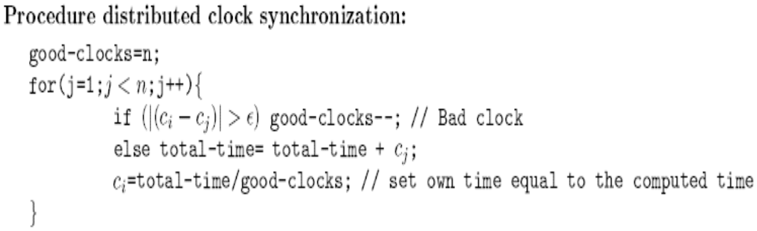
Byzantine Clock



- C_1 is a Byzantine clock
  - sending time value $t{+}e$ to clock C_5
  - t-e to clock C_3
  - at same time instant
- If $< \frac{1}{3}$ clocks bad(Byzantine), good clocks approx. synchronized
- Synchronization scheme of clocks
  - Let there be n clocks in a system
  - Each clock periodically broadcasts time value at end of certain interval
  - Assume clocks to be synchronized within ϵ time unita of each other
  - If a clock receives broadcast time differs from its own time by -> ϵ
  - then sending clock must be bad, safely ignores received time value
  - Each clock averages all good time values received
  - sets its time with average

Pseudo Code for Distributed Clock Synchronization

- Each clock Ci carries out following operations



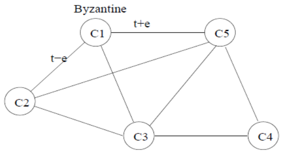
Procedure distributed clock synchronization:

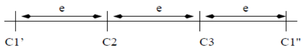
```
good-clocks=n;
for(j=1;$j < n$;j++){
        if $(|(c_i - c_j)| > \epsilon)$ good-clocks--; // Bad clock
        else total-time= total-time + $c_j$;
        $c_i$=total-time/good-clocks; // set own time equal to the computed time
}
```

- Each clock carries same set of steps
- If all n clocks carry these steps, then atmost m are bad
- $n > 3m$
- Good clocks will be synchronized within $3\epsilon m/n$ bound

Theorem 1

In a distributed system with n clocks, a single Byzantine clock can make two arbitrary clocks in a system to differ by $3\epsilon/n$ in time value, where ϵ represents the maximum permissible drift between two clocks.





Drift Between Two Clocks in Presence of Byzantine Clocks

- Proof. Consider three clocks C1, C2 and C3 in a distributed system.
  C1 is a Byzantine clock.
  C2 and C3 are good clocks - differ by $\not> \epsilon$.
  C3 being Byzantine shows two different values to C1 and C2.
  Effect of Byzantine clock in total time calculation makes two good clocks differ by atmost 3ϵ. Effect of a single Byzantine clock make two arbitrary clocks differ by $3\epsilon/n$. For m Byzantine clocks two good clocks differ by atmost $3\epsilon m$ in average. Individual clocks synchronized within $3\epsilon m/n$.

Time required for clocks drift to ϵ

- Let time required for two clocks to drift from $3\epsilon m/n$ to ϵ be ΔT.
  or, $2\Delta T\rho \leq (n\epsilon\text{-}3\epsilon m)/n$
  or, $\Delta T \leq (n\epsilon\text{-}3\epsilon m)/(n\times 2\rho)$
  ΔT is time required for two good clocks to drift from $3\epsilon m/n$ to ϵ
  $\Delta T \leq [(3m+1)\epsilon\text{-}3\epsilon m]/(n\times 2\rho)$
  $\Delta T \leq \epsilon/2n\rho$

Example 3

Let a distributed real-system have 10 clocks, and it is required to restrict their maximum drift to $\epsilon = 1$ msec. Let the maximum drift of the clocks per unit time (ρ) be 5×10^{-6}. Determine the required synchronization interval.

- Solution. $\Delta T = \frac{10^{-3}}{2 \times 10 \times 5 \times 10^{-6}}$
  $\Delta T = 10$ sec
  Required synchronization interval is 10 sec.

Thank you