Where Are We Now?

◆ Where we’ve been:
  • Distributed systems
  • Embedded communications: protocols & performance

◆ Where we’re going today:
  • Real Time Scheduling in distributed systems
  • This adds on to what you saw in 18-348/18-349
    – There is overlap, especially since grad students may not have seen this material

◆ Where we’re going next:
  • Humans as a system component
  • Mid-semester presentations next week – Mon/Wed/Fri
  • How to make sure you build systems right …
    … and how you can actually know that you built them right
Preview

◆ Basic real time review

◆ Scheduling – does it all fit?
  • Schedulability
  • Scheduling algorithms, including
    – Distributed system adaptations
    – How they degrade

◆ Complications
  • Aperiodic tasks
  • Task dependencies
Review: Real Time Review

- **Reactive:** computations occur in response to external events
  - Periodic events (e.g., rotating machinery and control loops)
  - Aperiodic events (e.g., button closures)
  - Real time means that correctness of result depends on both functional correctness and time that the result is delivered

- **Soft real time**
  - Utility degrades with distance from deadline

- **Hard real time**
  - System fails if deadline window is missed

- **Firm real time**
  - Result has no utility outside deadline window, but system can withstand a few missed results
Types of Real-Time Scheduling

- **Dynamic vs. Static**
  - Dynamic schedule computed at run-time based on tasks really executing
  - Static schedule done at compile time for all possible tasks

- **Preemptive permits one task to preempt another one of lower priority**
  - Also, centralized or distributed implementation?

---

Figure 11.1: Taxonomy of real-time scheduling algorithms.

[Kopetz]
Schedulability

- NP-hard if there are any resource dependencies at all
  - So, the trick is to put cheaply computed bounds/heuristics in place
    - Prove it definitely can’t be scheduled
    - Find a schedule if it is easy to do so
    - Punt if you’re in the middle somewhere

Figure 11.2: Necessary and sufficient schedulability test.
Periodic Messages and Tasks

− “Time-triggered” (periodic) tasks are common in embedded systems
  • Often via control loops or rotating machinery

− Components to periodic tasks
  • Period (e.g., 50 msec)
  • Offset past period (e.g., 3 msec offset/50 msec period -> 53, 103, 153, 203)
  • Jitter is random “noise” in release time (not oscillator drift)
  • Release time is when message submitted to transmit queue
  • Release time_n = (n*period) + offset + jitter ; assuming perfect time precision
Scheduling Parameters

- **Set of tasks** \{T_i\}
  - Periods p_i
  - Deadline d_i (completion deadline after task is queued)
  - Execution time c_i (amount of CPU time to complete)
  - Worst case latency to complete execution W_i
    - This is something we solve for, it’s not a given

- **Handy values:**
  - Laxity l_i = d_i - c_i (amount of slack time before Ti must begin execution)
  - Utilization factor \( \mu_i = \frac{c_i}{p_i} \) (portion of CPU used)
Simple Schedulability

\[ \mu = \sum \mu_i = \sum \frac{c_i}{p_i} \leq N \]

- **Necessary:**
  
  "You can’t use more that 100% of available CPU power!"

  \[ \mu_i = \frac{c_i}{p_i} \leq 1 \quad \text{and } 0 \leq i \leq N \]

- **Trivially Sufficient:**
  
  "One CPU per task always works, if each task fits on a single CPU"

- **Of course, the hard part is putting tighter sufficiency bounds on things..."
Distributed Static Schedule

- **Co-schedule CPUs and Network:**
  - Assign specific network transmission time to each message using a spreadsheet
  - Assign dedicated CPU time to each CPU to compute/transmit each message
  - Assign dedicated CPU time to receive/process applicable incoming messages
  - Iterate until the schedule contains no double-booked resources
Distributed Static Schedule Tradeoffs

- In a nutshell, this is time-triggered system design taken to extremes

- **Pro:**
  - Relaxes some of the scheduling assumptions discussed in next slide
  - If it works once, it will always work
    - Assuming that compute time never varies, ignoring message losses, etc.
    - Can adapt by putting in slack space for message retries
  - You can guarantee it works while using 100% of all resources
    - (Assuming that it is statically schedulable)
    - This makes it attractive for safety critical design – easy to know it will really work

- **Con:**
  - Might have to reschedule the whole thing for every change!
    - (build a tool to do this)
  - Probably have a different schedule for each operating mode
    - (build a tool to do this)
  - Might need a different set of schedules for each different model of the design
    - (build a tool to do this)
Major Assumptions

- Five assumptions are the starting point for this area:
  1. Tasks \( \{T_i\} \) are periodic, with hard deadlines and no jitter
     - Period is \( P_i \)
  2. Tasks are completely independent
     - \( B=0; \) Zero blocking time; no use of a mutex; interrupts never masked
  3. Deadline = period
     - \( P_i = D_i \)
  4. Worst case computation time is known and used for calculations
     - \( C_i \) worst case is always the same for each execution of the task
  5. Context switching is free (zero cost)
     - Executive takes zero overhead, and task switching has zero latency

- These assumptions are often not realistic
  - But sometimes they are close enough in practice
  - We’re going to show you the common special cases that are “easy” to use
    - And the starting points for dealing with situations in which the rules are bent
EDF: Earliest Deadline First

◆ Assume a preemptive system with dynamic priorities, and
  { same 5 assumptions}

◆ Scheduling policy:
  • Always execute the task with the nearest deadline
    – Priority changes on the fly!
    – Results in more complex run-time scheduler logic

◆ Performance
  • Optimal for uniprocessor (supports up to 100% of CPU usage in all situations)
    – If it can be scheduled – but no guarantee that can happen!
    – Special case where it works is very similar to case where Rate Monotonic can be used:
      » Each task period must equal task deadline
      » But, still pay run-time overhead for dynamic priorities
  
  • If you’re overloaded, ensures that a lot of tasks don’t complete
    – Gives everyone a chance to fail at the expense of the later tasks
Least Laxity

- Assume a *preemptive* system with *dynamic priorities*, and
  \{ same 5 assumptions \}

- **Scheduling policy:**
  - Always execute the task with the
    smallest laxity \( l_i = d_i - c_i \)

- **Performance:**
  - Optimal for *uniprocessor* (supports up to 100% of CPU usage in all situations)
    - Similar in properties to EDF
    - If it can be scheduled – but no guarantee that can happen!
  - A little more general than EDF for multiprocessors
    - Takes into account that slack time is more meaningful than deadline for tasks of
      mixed computing sizes
  - Probably more graceful degradations
    - Laxity measure permits dumping tasks that are hopeless causes
Distributed EDF/Least Laxity

- Requires using deadline information as priority (use CAN as example)
  - Each node does EDF CPU scheduling according to an end-to-end deadline
  - Each node locally prioritizes outgoing messages according to EDF or laxity
  - Each receiving node prioritizes tasks sparked by received messages EDF
  - Usually not globally optimal – not every CPU kept busy all the time

```
CAN ID FIELD

0 | LAXITY (x 100 usec) | NODE # | ID | ...

1 | NON-REAL-TIME MESSAGES
```

Global laxity priority for a network ([Livani98] has a more sophisticated scheme)
EDF/Least Laxity Tradeoffs

◆ **Pro:**
  • If it works, it can get 100% efficiency (on a uniprocessor)
  • Does not restrict task periods
  • Special case works if, for each task, Period = Deadline

◆ **Con:**
  • It is not always feasible to prove that it will work in all cases
    – And having it work for a while doesn’t mean it will always work
  • Requires dynamic prioritization
  • EDF has bad behavior for overload situations (LL is better)
  • The laxity time hack for global priority has limits
    – May take too many bits to achieve fine-grain temporal ordering
    – May take too many bits to achieve a long enough time horizon

◆ **Recommendation:**
  • Avoid EDF/LL if possible
    – Because you don’t know if it will really work in the general case!
    – And the special case doesn’t buy you much, but comes at expense of dynamic priorities
Rate Monotonic Scheduling

Problems with previous approaches

- Static scheduling – can be difficult to find a schedule that works
- EDF & LL – run-time overhead of dynamic priorities

Wanted:
- Easy rule for scheduling
- Static priorities
- Guaranteed schedulability

Rate Monotonic Scheduling

- { same 5 assumptions }

1. Sort tasks by period (i.e., by “rate”)
2. Highest priority goes to task with shortest period (fastest rate)
   - Tie breaking can be done by shortest execution time at same period
3. Use prioritized preemptive scheduler
   - Of all ready to run tasks, task with fastest rate gets to run
Rate Monotonic Characteristics

◆ **Static priority**
  • Priorities are assigned to tasks at design time; priorities don’t change at run time

◆ **Preemptive**
  • When a high priority task becomes ready to run, it preempts lower priority tasks
  • This means that ISRs have to be so short and infrequent that they don’t matter
    – (If they are non-negligible, see Blocking Time discussion later)

◆ **Guarantees** schedulability if you don’t overload CPU (see next slide)
  • All you have to do is follow the rules for task prioritization
  • (And meet the 5 assumptions)

◆ **Variation: Deadline Monotonic**
  • Use min(period, deadline) to assign priority rather than just period
  • Works the same way, but handles tasks with deadlines shorter than their period
Example of a Missed Deadline at 79% CPU Load

- **Task 4 misses deadline**
  - This is the worst case launch time scenario

- **Missed deadlines can be difficult to find in system testing**
  - 5 time units per task is worst case
    - Average case is often a bit lighter load
  - Tasks only launch all at same time once every 224,808 time units
    
    \[
    \text{LCM}(19, 24, 29, 34) = 224,808
    \]

    \[
    (\text{LCM} = \text{Least Common Multiple})
    \]
Harmonic RMS or DMS

- For arbitrary periods, only works for \(~70\%\) CPU loading

$$\mu = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(\sqrt{2} - 1); \quad \lim_{n \to \infty} (\mu) \leq 69.3\%$$

  - Most systems don’t want to pay 30\% tribute to the gods of schedulability… so…

- Make all periods “harmonic”
  - \(P_i\) is evenly divisible by all shorter \(P_j\)
  - This period set is harmonic: \(\{5, 10, 50, 100\}\)
    - \(10 = 5*2; \quad 50 = 10*5; \quad 100 = 50*2; \quad 100 = 10*5*2\)
  - This period set is \(\text{not}\) harmonic: \(\{3, 5, 7, 11, 13\}\)
    - \(5 = 3 * 1.67\) (\(\text{non-integer}\)), etc.

- If all periods are harmonic, works for \(\text{CPU load of } 100\%\)
  - Harmonic periods can’t drift in and out of phase – avoids worst case situation

$$\mu = \sum_{i} \frac{C_i}{P_i} \leq 1 \; ; \; \forall_{p_j < p_k} \{p_j \text{ evenly divides } p_k\}$$
Example Deadline Monotonic Schedule

<table>
<thead>
<tr>
<th>Task #</th>
<th>Period ((P_i))</th>
<th>Deadline ((D_i))</th>
<th>Compute ((C_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>16</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>T3</td>
<td>30</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>T4</td>
<td>60</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>T5</td>
<td>60</td>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task #</th>
<th>Priority</th>
<th>(\mu)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>1/5 = 0.200</td>
</tr>
<tr>
<td>T2</td>
<td>3</td>
<td>2/16 = 0.125</td>
</tr>
<tr>
<td>T3</td>
<td>2</td>
<td>2/6 = 0.333</td>
</tr>
<tr>
<td>T4</td>
<td>5</td>
<td>3/60 = .05</td>
</tr>
<tr>
<td>T5</td>
<td>4</td>
<td>4/30 = 0.133</td>
</tr>
</tbody>
</table>

TOTAL: \(\mu = 0.841\)

\[
\mu = \sum \frac{C_i}{P_i} \leq N(\sqrt{2} - 1) \quad ; \quad N = 5
\]

\[
\mu = 0.841 \quad (not \leq) \quad 0.743
\]

\(Not Schedulable!\)

(Might be OK with exact schedulability math...
... but then you have to use fancy math!)
**Example Harmonic Deadline Monotonic Schedule**

<table>
<thead>
<tr>
<th>Task #</th>
<th>Period (P&lt;sub&gt;i&lt;/sub&gt;)</th>
<th>Deadline (D&lt;sub&gt;i&lt;/sub&gt;)</th>
<th>Compute (C&lt;sub&gt;i&lt;/sub&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>15</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>T3</td>
<td>30</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>T4</td>
<td>60</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>T5</td>
<td>60</td>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task #</th>
<th>Priority</th>
<th>( \mu )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>( \frac{1}{5} = 0.200 )</td>
</tr>
<tr>
<td>T3</td>
<td>2</td>
<td>( \frac{2}{5} = 0.400 )</td>
</tr>
<tr>
<td>T2</td>
<td>3</td>
<td>( \frac{2}{15} = 0.133 )</td>
</tr>
<tr>
<td>T5</td>
<td>4</td>
<td>( \frac{4}{30} = 0.133 )</td>
</tr>
<tr>
<td>T4</td>
<td>5</td>
<td>( \frac{3}{60} = 0.05 )</td>
</tr>
</tbody>
</table>

\[ \mu = \sum \frac{C_i}{P_i} \leq 1 \quad ; \text{Harmonic periods \{5, 15, 30, 60\}} \]

\[ \mu = 0.916 \leq 1 \]

*Schedulable, even though usage is higher!*
Distributed Rate/Deadline Monotonic

- **Schedule network using Deadline Monotonic assignment**
  - Implement by assigning CAN priorities according to period length
    - This is what is done in CAN most of the time anyway
  - Network is non-preemptable, but assume it’s close enough because each message (=task) is short compared to deadlines
    - Add longest message as blocking time
    - Look up the blocking time math in an RMS/DMS paper (it’s a bit complex)

- **Schedule each node using Deadline Monotonic assignment**
  - Static priorities and pre-emptive prioritized scheduler

- **Is that enough?**
  - Should work for piecewise compute+transmit+compute deadlines
  - But for each “hop” you might lose out on one local period extra latency
Dealing With Background Tasks

◆ “Other” tasks need to be executed without deadlines

◆ Several possible approaches:
  • Dedicate a fixed number of CPUs to routine tasks
  • Assign all routine tasks lowest priority, and execute round-robin
    – Effectively equivalent to an “other task server” but also uses any leftover time from other tasks that run short, are blocked, or aren’t in execution
  • Assign an “other task server” for routine tasks
    – Each “other task” is executed from the server’s budget
    – Has the advantage of giving consistent CPU proportion for system validation

◆ Distributed version: do the same thing with network bandwidth
What If:

1. Tasks \( \{T_i\} \) are NOT periodic
   - Use Sporadic techniques (stay tuned)

2. Tasks are NOT completely independent
   - Worry about dependencies (stay tuned)

3. Deadline NOT = period
   - Use Deadline monotonic

4. Computation time \( c_i \) isn’t known and constant
   - Use worst case computation time (WCET), if known
   - Can be tricky to compute – for example what if number of times through a loop is data dependent? (stay tuned)

5. Context switching is free (zero cost)
   - If it isn’t free add this to blocking time (see assumption 2 above)
Aperiodic Tasks

- **Asynchronous tasks**
  - External events with no limits on inter-arrival rates
  - Often Poisson processes (exponential inter-arrival times)

- **How can we schedule these?**
  - Mean inter-arrival rate? (only useful over long time periods)
  - Minimum inter-arrival time with “filtering” (limit rate to equal deadline)
    - Artificial limit on inter-arrival rate to avoid swamping system
    - May miss arrivals if multiple arrivals occur within the filtering window length
Dealing With Sporadic Tasks

- “Sporadic” means there is a limit on maximum repetition time
- “Aperiodic” means all bets are off – none of the theories handle this case

◆ **Approach #1: pretend sporadic tasks are periodic**
  - Schedule time for a sporadic task at maximum possible needed execution rate
  - Simplest approach if you have capacity
  - But, this can be wasteful, because reserves CPU for tasks that seldom arrive

◆ **Approach #2: Use a **sporadic server**  (this is a simplified description)**
  - Schedule a periodic task that is itself a scheduler for sporadic tasks
    - For example, might serve sporadic tasks in FIFO or round robin order
    - But, sporadic server limits itself to a maximum $C_i$ and runs once every $P_i$
    - This might look like a preemptive mini-tasker living as a single RTOS task
  - Use sporadic server time for any sporadic task that is available
  - Decouples timing analysis for sporadic server from other tasks
  - Can also handle aperiodic tasks without disrupting other main tasks
    - But, no magic – still can’t make guarantees for those aperiodic tasks
    - Need some specialty math to manage and size the sporadic server task
Special Case For Mixed Safety/Non-Safety Systems

- **Two-phase schedule to ensure safety critical task service times**
  1. Critical: Round robin schedule with maximum times per task
     - Non-preemptive tasking with deterministic timing and fixed ordering
  2. Non-Critical: Prioritized task segment with maximum *total* time
     - Basically a sporadic server, for example first-in/first-out ordering within time slice
     - **Terminates or suspends tasks at end of its designated slice**

- For example, the FlexRay automotive network protocol does this
  - Except it applies it to scheduling network messages, not CPU tasks
Blocking Time: Mutex + Priorities Leads To Problems

- Scenario: Higher priority task waits for release of shared resource
  - Task L (low prio) acquires resource X via mutex
  - Task H (high prio) wants mutex for resource X and waits for it

- Simplistic outcome with no remedies to problems (don’t do this!)
  - Task H hogs CPU in an infinite test-and-set loop waiting for resource X
  - Task L never gets CPU time, and never releases resource X

  - Strictly speaking, this is “starvation” rather than “deadlock”
Bounded Priority Inversion

- An possible approach (BUT, this has problems…)
  - Task H returns to scheduler every time mutex for resource X is busy
  - Somehow, scheduler knows to run Task L instead
    - If it is a round-robin preemptive scheduler, this will help
    - In prioritized scheduler, task H will have to reschedule itself for later
      » Can get fancy with mutex release re-activating waiting tasks, whatever ….
  - Priority inversion is bounded – Task L will eventually release Mutex
    - And, if we keep critical regions short, this blocking time B won’t be too bad
**Unbounded Priority Inversion**

- But, simply having Task H relinquish the CPU isn’t enough
  - Task L acquires mutex X
  - Task H sees mutex X is busy, and goes to sleep for a while; Task L resumes
  - Task M preempts task L, and runs for a long time
  - Now task H is waiting for task M ➔ Priority Inversion
    - Task H is *effectively* running at the priority of task L because of this inversion

![Figure 2: Unbounded priority inversion](Renwick04)
Solution: Priority Inheritance

- **When task H finds a lock occupied:**
  - It elevates task L to at least as high a priority as task H
  - Task L runs until it releases the lock, but with priority of at least H
  - Task L is demoted back to its normal priority
  - Task H gets its lock as fast as possible; lock release by L ran at prio H

- **Idea: since mutex is delaying task H, free mutex as fast as you can**
  - Without suspending tasks having higher priority than H!
  - For previous slide picture, L would execute with higher prio than M

![Figure 5: Simple priority inheritance](Renwick04)
Priority Inheritance Pro/Con

◆ Pro: it avoids many deadlocks and starvation scenarios!
  • Only elevates priority when needed (only when high prio task wants mutex)
  • (An alternative is “priority ceiling” which is a similar idea)

◆ Run-time scheduling cost is perhaps neutral
  • Task H burns up extra CPU time to run Task L at its priority
  • Blocking time B costs per the scheduling math are:
    – L runs at prio H, which effectively increases H’s CPU usage
    – But, H would be “charged” with blocking time B regardless, so no big loss

◆ Con: complexity can be high
  • Almost-static priorities, not fully static
    – But, only changes when mutex encountered, not on every scheduling cycle
  • Nested priority elevations can be tricky to unwind as tasks complete
  • Multi-resource implementations are even trickier

◆ If you can avoid need for a mutex, that helps a lot
  • But sometimes you need a mutex; then you need priority inheritance too!
Mars Pathfinder Incident (Sojourner Rover)

◆ July 4, 1997 – Pathfinder lands on Mars
  • First US Mars landing since Vikings in 1976
  • First rover to land (in one piece) on Mars
  • Uses VxWorks RTOS

◆ But, a few days later…
  • Multiple system resets occur
    – Watchdog timer saves the day!
    – System reset to safe state instead of unrecoverable crash
  • Reproduced on ground; patch uploaded to fix it
    – Developers didn’t have Priority Inheritance turned on!
    – Scenario pretty much identical to H/M/L picture a couple slides back
    – Rough cause: “The data bus task executes very frequently and is time-critical -- we shouldn't spend the extra time in it to perform priority inheritance” [Jones07]
Applied Deadline Monotonic Analysis With Blocking

- **Blocking time** $B_i$ is worst case time that Task $i$ can be blocked
  - Combination of blocking from semaphores, bounded length priority inversion, etc.

- **For each task**, ensure that task plus its blocking time uses less than 100% of CPU:
  - $\mu_1 = \left( \frac{c_1}{p_1} \right) + \frac{B_1}{p_1} \leq 1$
  - $\mu_2 = \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \frac{B_2}{p_2} \leq 1$
  - $\mu_3 = \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \left( \frac{c_3}{p_3} \right) + \frac{B_3}{p_3} \leq 1$

  $\forall k; \mu_k = \sum_{i\leq k} \mu_i = \sum_{i\leq k} \left( \frac{c_i}{p_i} \right) + \frac{B_k}{p_k} \leq 1$; for harmonic periods

- **Pessimistic bound** – penalize all tasks with worst case blocking time:
  - $\mu = \left( \sum_i \frac{c_i}{p_i} \right) + \frac{\max(B_j)}{p_j} \leq 1$; for harmonic periods
Determinacy & Predictability (the “C” term)

◆ **Determinacy** = *same performance every time*
  - Low determinacy can cause control loop instabilities
    - If it’s non-deterministic, how do you know you certified/tested the worst case?
  - System-level mechanisms can cause non-determinism:
    - Cache memory; speculative execution; virtual memory; disk drive seek times
    - Context switching overhead; interrupts
    - Prioritized network/task interactions (depends on situation; this is controversial)
  - Determinacy can be improved
    - Insert waits to ensure results are always delivered with worst case delay
    - Avoid/turn off non-deterministic hardware & software features
    - Ensure conditional paths through software are the same length
    - Use only static iteration counts for loops
    - Extreme case – end-to-end cyclic static schedule for everything

◆ **Predictability** = *designer can readily predict performance*
  - High end processors are nearly impossible to understand clock-by-clock
    - Some have ways to make things predictable & deterministic (e.g. Power PC 603e)
How Hard Is It To Predict Performance?

- Computing worst-case “C” is difficult for high performance CPUs
  - Data from an 80486 (cache, but no speculative execution)
Watch Out For Network Problems!

◆ Corrupted network messages
  • Do you re-transmit?
    – Introduces jitter for that message
    – Delays all subsequent messages
    – Need to reserve extra space to avoid later messages missing deadlines
  • Do you ignore?
    – Use stale data or introduce large jitter for one variable

◆ Network blackout
  • What if entire network is disrupted for 100+ msec?
    – (What if the cable gets cut?)

◆ Alternate strategies for dealing with network noise
  • Maintain freshness counters for all network data
  • Send every message twice
    – Or, run control loops faster than necessary (including message traffic)
  • Forward error correction codes (but won’t help with blackout)
Review

◆ Scheduling – does it all fit?
  • Schedulability – necessary vs. sufficient
  • Scheduling algorithms – static, EDF, LL, RM
    – Distributed versions as well as single-CPU versions

◆ Complications
  • Aperiodic tasks
  • Inter-task dependencies
  • Worst case execution time

◆ Assumptions… (next slide)
Review – Assumptions

- Assume non-preemptive system with 5 Restrictions:
  1. Tasks \( \{T_i\} \) are periodic & predictable, with hard deadlines and no jitter
     - Various hacks to make things look periodic
     - Various hacks to increase determinacy
  2. Tasks are completely independent
     - Pretend a string of tasks is really one task for scheduling
  3. Deadline = period \( p_i = d_i \)
     - Use worse case of deadline or period for scheduling
  4. Worst case computation time \( c_i \) is known and used for calculations
     - For a pessimistic approximation, turn off caches to take measurements
  5. Context switching is free (zero cost) INCLUDING network messages to send context to another CPU(!)
     - It’s not free, but as CPUs get faster it gets cheaper compared to real time

- Don’t forget that the theory does not account for dropped messages!