Theorem (Liu & Layland, 1973)

- A sufficient condition for a set of \( n \) tasks to be rate-monotonically schedulable on 1 processor, regardless of phasing is:
  \[
  \text{total utilization} \leq n^{*}(2^{1/n} - 1)
  \]
- where total utilization is defined as
  \[
  \text{sum}(C_i/P_i, i = 1 \text{ to } n)
  \]
- The condition is sufficient, but not necessary.

Numeric Values for Liu & Layland Rule

- bound on total utilization
  \[
  \begin{array}{|c|c|}
  \hline
  \text{tasks} & \text{bound on total utilization} \\
  \hline
  1 & 1 \\
  2 & 0.828427 \\
  3 & 0.779763 \\
  4 & 0.756828 \\
  8 & 0.724062 \\
  16 & 0.707472 \\
  32 & 0.700709 \\
  64 & 0.696914 \\
  \cdots & \ln(2) \approx 0.693147 \\
  \infty & \ln(2) \\
  \hline
  \end{array}
  \]

Rationale behind Liu & Layland

- Assume all phases are 0.
- Case of 1 task, \( T_1 \), period \( P_1 \), time \( C_1 \).
  - Obviously this will be schedulable iff \( C_1 \leq P_1 \),
    which is the same as
    \[
    \text{sum}(C_i/P_i, i = 1 \text{ to } 1) = 1.
    \]

RMA Rationale

- Case of \( n \) tasks, \( T_1, \ldots, T_n \), periods \( P_1 \leq \ldots \leq P_n \), times \( C_1, \ldots, C_n \).
  - Consider an interval of time \([0, t]\).
  - During the interval each \( T_i \) must execute \( \lfloor t/P_i \rfloor \) times.
  - The total time used for \( T_i \) during the interval will be \( C_i \lfloor t/P_i \rfloor \).
  - In order for each \( T_i \) to execute the appropriate number of times in the interval, we need \((\forall t' < t) \ t' > \text{sum}(C_i \lfloor t/P_i \rfloor)\).
  - If we can find a \( t < \text{lcm}(P_1, \ldots, P_n) \) having this property, then all tasks can be scheduled.

Observation

- \((\forall t' < t) \ t' \geq \text{sum}(C_i \lfloor t/P_i \rfloor)\)
  iff
- \((\forall t' < t) \ t' \text{ is a time corresponding to the end of some task's period} \Rightarrow t' \geq \text{sum}(C_i \lfloor t/P_i \rfloor)\).

Example

- \( P_1 = 100, \ P_2 = 150, \ C_1 = 20, \ C_2 = 30. \)
- Consider an interval of time \([0, 300]\), during which \( T_1 \) must complete \( \lfloor 300/100 \rfloor = 3 \) times and \( T_2 \) must complete 2 times.
- The total time required is \( 3 \times 20 + 2 \times 30 = 120 \leq 300. \)
- What schedule realizes the requirements?
  \[
  \begin{array}{|c|c|c|c|c|c|}
  \hline
  \text{task} & T_1 & T_2 & T_1 & T_2 & T_1 \\
  \text{start} & 0 & 20 & 100 & 150 & 200 \\
  \text{end} & 20 & 50 & 120 & 180 & 220 \\
  \hline
  \end{array}
  \]
Example (L&L rule not necessary)

- Consider adding a third task:
  \[ P_1 = 100, \ P_2 = 150, \ P_3 = 210,\]
  \[ C_1 = 20, \ C_2 = 30, \ C_3 = 80. \]

- The Liu and Layland rule computes total utilization:
  \[ 20/100 + 30/150 + 80/210 = 0.780952 \] which is not realizable for 3 tasks (0.7796).

- Since the Liu and Layland rule is sufficient, but not necessary, there still might be a schedule.

- Can you construct one?

Example

- \[ P_1 = 100, \ P_2 = 150, \ P_3 = 210,\]
- \[ C_1 = 20, \ C_2 = 30, \ C_3 = 80. \]

- Consider an interval of time \([0, 2100]\), \(2100\) is the lcm of 100, 150, and 210) during which \(T_1\) must complete 21 times, \(T_2\) 14 times, and \(T_3\) 10 times.

- The total time required is \(21 \times 20 + 14 \times 30 + 10 \times 80 = 420+720+800 = 1940 < 2100\). So it is at least plausible that there is a schedule.

- On the next page, we construct a schedule.

A Rate-Monotone Schedule

<table>
<thead>
<tr>
<th>(T_1)</th>
<th>(T_2)</th>
<th>(T_3)</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C = 20, P = 100)</td>
<td>(C = 30, P = 150)</td>
<td>(C = 80, P = 210)</td>
<td></td>
</tr>
</tbody>
</table>

We didn't have to compute that whole thing:

Lehoczky, Sha, and Ding Theorem (1987)

- Using RM scheduling, if each task meets its first deadline, then all deadlines will be met.

Example

- Consider adding a fourth task:
  \[ P_1 = 100, P_2 = 150, P_3 = 210, P_4 = 400,\]
  \[ C_1 = 20, \ C_2 = 30, \ C_3 = 80, \ C_4 = 100. \]

- Total utilization: \[ 20/100 + 30/150 + 80/210 + 100/400 = 1.03095 > 1, \] so this set is not realizable.

2nd Theorem of Liu & Layland

- If a set of tasks is schedulable under any fixed priority scheme, then it is schedulable using rate-monotonic scheduling.

- In other words, rate-monotonic is optimal among fixed priority schemes.
How release time of a higher-priority task can affect the response of a task in RM scheduling

Liu & Layland Idea

- A critical instant of a task is a time at which the task becomes available and all higher priority tasks also become available.
- If task scheduling can be determined to be feasible at critical instants, then it is feasible in general.

Priorities Revisited

- RMA is an example of a fixed priority scheme: priority is determined in order opposite periods.
- Dynamic priority schemes are possible.

Dynamic Priority

- Recall Horn’s rule: Preemptive EDF (Earliest Deadline First)
- It works for arbitrary arrivals.
- Therefore it will work for periodic tasks as well.

Dynamic Priority

- EDF is dynamic because the relative priority will depend on what is being executed when a task arrives. This could be:
  - A task with a longer period but nearer deadline.
  - A task with a shorter period but more distant deadline.

A Dynamic-Priority Schedule

(each box = 10 time units)
Exercise

- Devise a system of two periodic tasks such that:
  - There is no rate-monotonic schedule
  - There is an EDF schedule

Hint

- Make the longer-period task have a relatively high utilization, so that deferring it will make it miss its deadline.

Hint

- $P_1 = 5, C_1 = 2$
- $P_2 = 7, C_2 = 4$

Example

- $P_1 = 5, C_1 = 2$
- $P_2 = 7, C_2 = 4$

Utilization Bounds for Dynamic Priority Assignment

- For dynamic priority assignment, any system with a total utilization $\leq 1$ can be scheduled.
- EDF is adequate for constructing such a schedule.

Generalization of Periodic Tasks

- So far, deadline of a periodic task = end of period
- Generalization: periodic tasks with fixed deadlines relative to start of period (deadlines possibly sooner than end of period).
Deadline-Monotonic (DM) Scheduling

- Assume relative deadlines are constant.
- Assign priorities in order of nearest relative deadline.
- Since the relative deadlines are constant, this is a static priority assignment.
- DM has been shown optimal for this more general case (Leung and Whitehead, 1982).

If relative deadlines are $\leq$ period and not constant

- EDF again works
- Schedulability can be checked using “processor demand” approach:
  - Processor demand in an interval $[0, L]$ is the computation time required in order for all tasks to complete by their deadlines in that interval.
  - Check that processor demand $\leq$ length of interval.
  - Need only check at release times between 0 and lcm(periods).

Summary of Applicability

<table>
<thead>
<tr>
<th>Static Priority</th>
<th>Dynamic Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline $=$ Period</td>
<td>Deadline $&lt; $ Period</td>
</tr>
<tr>
<td>Rate Monotonic</td>
<td>Deadline Monotonic</td>
</tr>
<tr>
<td>EDF</td>
<td>EDF</td>
</tr>
</tbody>
</table>

References


Resource Access

- Semaphores can be used to control access to resources in a real-time system.
- Some interesting issues associated with priority arise.

Priority Inversion

- Consider two tasks, one high priority, one low, that share a resource protected by a critical section.
- If the high-priority task becomes schedulable during the time the low-priority one is in its critical section:
  - The high-priority task is blocked by the low priority one.
Push-Through Blocking

- The low-priority task, with the resource locked, could be preempted by a medium priority task that doesn’t necessarily use the resource.
- This task could go on indefinitely, in-effect blocking the higher-priority task through the lower-priority one.

Possible Resolutions of Priority Inversion

- Abort the low-priority task:
  - Messy, since this could leave the system in an inconsistent state.
- Priority Inheritance Protocol
- Priority Ceiling Protocol

Priority Inheritance Protocol (PIP)

- During the time the low-priority task is in its critical section, and while the high-priority task is blocked because of this, the low-priority task inherits the priority of the high-priority task.

Possible Resolution of Blocking

- Priority Inheritance Protocol:
  - More precisely: A task locking a shared resource inherits the priority of the highest-priority task blocked because of locking.
  - The locking task’s priority is recomputed whenever:
    - Other tasks request or release the shared resource, or
    - the locking task leaves the critical section, in which case the highest-priority blocked task is awakened.

Transitivity

- Priority Inheritance must be made Transitive:
  - If T₁ blocks T₂, and T₂ blocks T₃, then T₁ inherits the priority of T₃.
  - Note that transitive inheritance occurs only with nested critical sections (different semaphores). If there were only one semaphore, then T₁ would be blocking T₃ directly.

PIP Example

```
<table>
<thead>
<tr>
<th>higher base priority</th>
<th>T₃:0</th>
<th>T₁:1</th>
<th>T₂:2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

- T₃:0
- T₁:1
- T₂:2

S₁
S₂
S₃

requesting semaphores S₁, ...

holding semaphores S₁, ...

ordinary execution

P = preemption
A = acquisiton
R = request
X = release (of semaphore)

S₁
S₂
S₃

using semaphores S₁, ...

```
### Implementation Note

- Implementing priority inheritance in semaphores requires added sophistication beyond the basic semaphore mechanism.

---

### Computing Blocking Time

- If the computation time (without block) within critical sections is known, then the blocking time due to priority inheritance can be computed.
- This can be used in determining schedulability of a system of tasks with shared resources.

---

### Example: Solaris Operating System

- Solaris kernel schedules based on LWP's (Light-Weight Processes).
- Regard these as schedulable units of processor time.
- A new LWP can be created as an optional aspect of creating a new thread.

---

### Example: Solaris Operating System

- Each LWP has a priority, in one of three coarse classes:
  - RT (real-time) is highest.
  - System class is middle (not used by user processes).
  - TS (time-share) is lowest.

---

### Example: Solaris Operating System

- For the time-sharing class, the dispatch priority is calculated from
  - amount of CPU used since last I/O (less ⇒ higher-priority)
  - its nice level (set by the user)
- For the real-time class,
  - the highest-priority LWP runs until it blocks, terminates, reaches end of time-slice, or is preempted.
- When a process is created, its LWP gets the scheduling class and priority of the parent process.
- A thread can be either:
  - bound to a specific LWP
  - unbound (multiplexed among various LWP's)
- All unbound threads in a process have the same class and priority.
- Bound threads have the class and priority of the LWP to which they are bound.
Example: Solaris Operating System
Priority Inheritance

- Solaris implements basic priority inheritance protocol:
  - When a higher-priority thread is blocked, its priority is given to the lower-priority thread blocking it.
  - When the lower-priority thread ceases to block a higher priority one, its priority is set back to the original priority.

Priority Ceiling Protocol (PCP)

- A more involved form of priority inheritance.
- The PIP does not prevent a high priority task from being blocked multiple times by lower priority ones.
- The PCP does: Once a task gains entry to a critical section, it cannot be blocked by a lower priority task.

Priority Ceiling Protocol (PCP)

- Each semaphore has a (static) priority ceiling equal to the priority of the highest-priority task that could possibly lock it.
- A task is allowed to enter a critical section only if its priority is higher than the priority ceilings of all semaphores currently locked by other tasks.
- If a task is blocked on a semaphore, the priority of the blocked task is inherited by the task locking that semaphore.

PCP Example

Comparison: PIP vs. PCP

- PCP is more efficient at run-time, in that a high priority task cannot be blocked as many times.
- PIP is less demanding, in that it does not require a thorough analysis of a task’s behavior (in terms of which semaphores in might request).

References