**Real-Time Computing**

**Definition of Real-Time Computing**
- Real-time computing is computing in which the time of the computation plays an essential role in the result.
- This includes:
  - Computations that measure time
  - Computations that must meet deadlines
  - Computations that synchronize other computations based on time

**Obligatory Inspiring Example**
- July 20, 1969, landing module 10,000 feet above the Moon:
  - Houston: “Eagle, you’re go for a landing.”
  - Houston: “One minute [of fuel left].”
  - Lander: 100 feet, 3 1/2 down, 9 forward.”
  - Houston: “30 Seconds.”
  - Lander: “OK, engine stop.”

**Example of Real-Time**
- “During the descent of the Eagle [landing module], an incorrect switch position caused the analog-to-digital conversion circuit of the rendezvous to send some bursts of high-priority requests to the computer.
- After 15 percent of the computer resources were tied up in responding to the spurious requests, jobs began to miss their deadlines.
- A hardware recovery mechanism detected the timing fault and restarted the computer.”

**Control Loops**

- Autonomous, or semi-autonomous vehicles or robots require control loops to govern their motion.
- Computing equipment may also require such loops (e.g. transfers to/from rotational media).
- Less-regular computational tasks may need to be handled by the same system.

**Example of Real-Time: Control Loop**
- A vehicle is governed by a control law, that takes input from sensors and produces output to actuators.
- From sensor input, an estimate of position, velocity, etc. is computed.
- Based on the computed estimate, appropriate adjustment is made to actuators.
- For example, the adjustment may be based on difference between estimated state and desired state.
- The control law is based on a sampling and update rate.
- All computations must complete in time for the next sample.
Example of Real-Time: Control Loop

Examples of Control Laws

- Open- vs. Closed-loop control
- Bang-Bang ("On-Off") controllers
- P controller (proportional controller)
- PI controller (proportional-integral controller)
- PID controller (proportional-integral-derivative controller)

Comparison of Control Laws

- Bang-Bang:
  - Measure error; Fully open the valve for a fixed time (or not)*
- P controller ("proportional")
  - Measure error; Open the valve in proportion to error*
- PI controller (proportional-integral)
  - Measure error; Open the valve in proportion to integral of error*
- PID controller (proportional-integral-derivative)
  - Similar to PI, with "kicker" based on derivative of output

P vs. PI Controller Response

Figures from:

PID Response

PID Controller
Sample PID Controller Code

typedef struct
{
    double dState;        // Last position input
    double iState;        // Integrator state
    double iMax, iMin;    // Maximum and minimum allowable integrator state
    double iGain, pGain, dGain;          // integral gain
                                    // proportional gain
                                    // derivative gain
} SPid;

double UpdatePID(SPid * pid, double error, double position)
{
    double pTerm, dTerm, iTerm;
    pTerm = pid->pGain * error;   // calculate the proportional term
    // calculate the integral state with appropriate limiting
    pid->iState += error;
    if (pid->iState > pid->iMax) pid->iState = pid->iMax;
    else if (pid->iState < pid->iMin) pid->iState = pid->iMin;
    iTerm = pid->iGain * iState;  // calculate the integral term
    dTerm = pid->dGain * (pid->dState - position);
    pid->dState = position;
    return pTerm + dTerm + iTerm;
}

High-Level Applications

Other Real-Time Computation Issues

- Database and networking access
- Image/video/speech processing
- Rendering
- Performance monitoring
- Fault monitoring
- Fault recovery
- Security checks, certification
- Integrity checking
- Logging
- Planning

Distinctions

- **Hard** real-time: Deadlines must be met in order for the system to be **correct**.
- **Soft** real-time: It is **desirable** for deadlines to be met, but if not, the system can still be **correct**.
- “Real-time” does not necessarily mean “fast”.  

High-Level Applications

High-Level Applications
Hard Real-Time Examples

- Vehicular fuel or rendezvous problems
  - Lunar lander
  - Train scheduling
  - Baggage handlers
  - Assembly lines
  - Production deadlines
    - Newspaper
    - Live TV show
    - Graduation

Hardness Expressed as a **Utility** Function

![Utility Function Diagram]

- Hard real-time
  - Utility as a function of time
  - Deadline
- Softer real-time
  - Utility as a function of fraction of the task completed
  - Completion time

Progressive Utility

- A generalization of the preceding concept regards Utility as a function of both time and fraction of the task completed.

![Progressive Utility Diagram]

- Utility\(U(t,f)\) is the utility of completing fraction \(f\) of the task by time \(t\).

Real-Time includes Communication

- Obviously communication, as well as computation, must be taken into account if the system consists of multiple components.

Common Aspects of Real-Time Systems

- Deadlines
- Scheduled task start times
- Timeouts
- Periodic & Aperiodic (Sporadic, Episodic) tasks
- Priorities among tasks
- Preemption/resumption of lower priority tasks (interrupts)

Real-Time Operating Systems

- Solaris, when generated with real-time features
- VxWorks (Wind River Systems, Inc.)
- QNX (POSIX.1 certified)
- RTMX (POSIX RT extensions to free BSD)
- Windows Embedded
- Many others, see [http://www.ipi.com/page2.html](http://www.ipi.com/page2.html)
Example Real-Time Task Considerations

Scheduling Algorithms

- Choices of performance metric to meet requirement:
  - Overall completion time among all tasks
  - Average response time over all tasks
  - Weighted sum of completion times
  - Maximum lateness
  - Number of late tasks
  - each of these to be minimized.

Scheduling to Meet Deadlines

- Assume a set of tasks \( \{T_i\} \)
- Associate with each task \( T_i \):
  - computation time \( C_i \)
  - deadline \( D_i \)

- Suppose we want to minimize maximum lateness

Proof that Jackson’s Rule works

(typical of reasoning used in proofs)

- Let \( S’ \) be a schedule that executes the tasks in some order other than by Jackson’s Rule.
- Let \( S \) be a schedule that executes a pair of tasks out-of-order in \( S’ \) in order of decreasing deadline.
- We want to show that the maximum lateness of \( S \) is no more than that of \( S’ \).

Proof that Jackson’s Rule works

- In \( S’ \) there are two tasks \( T_a \) and \( T_b \) such that \( D_a \leq D_b \) but \( T_b \) precedes \( T_a \).
- Let \( L_i \) be the lateness of task \( T_i \), defined as:
  \[ L_i = F_i - D_i \]
  - Finish time - Deadline
- In schedule \( S \), the combined lateness of \( T_a \) and \( T_b \) is:
  \[ \max(L_a, L_b) \]
Proof that Jackson’s Rule works

- We want to show that:
  \[ \max(L_a, L_b) \leq \max(L'_a, L'_b) \]  (*)
  where the ' designates S' vs. S.
- Consider two cases:
  - \( L_a < L_b \)
  - \( L_a \geq L_b \)
  - We will show that the inequality holds in either case.

Proof of Jackson’s Rule

- Case \( L_a < L_b \):
  \[ \max(L_a, L_b) = L_b \] def'n of max, since \( L_a < L_b \)
  \[ = F_b - D_b \] def'n of L
  \[ = F'_a - D_b \] a and b are flipped in S' vs S
  \[ \leq F'_a - D_a \] assumption \( D_a \leq D_b \)
  \[ = L'_a \] def'n of L
  \[ \leq \max(L'_a, L'_b) \] def'n of max

Jackson’s Rule with \( L_a < L_b \)

- Case \( L_a > L_b \):
  \[ \max(L_a, L_b) = L_a \] def'n of max
  \[ = F_a - D_a \] def'n of L
  \[ < F'_a - D_a \] assumption that \( T_b \) precedes \( T_a \) in S'
  \[ = L'_a \] def'n of L
  \[ \leq \max(L'_a, L'_b) \] def'n of max

Jackson’s Rule with \( L_a \geq L_b \)

Corollary to Jackson’s Rule

- Assume that tasks are numbered in order of increasing deadlines \( D_i \).
- Let \( C_i \) be the corresponding computation time.
- Then all tasks can be executed so as to meet their deadline provided that
  \( \forall i \)
  \[ \sum(C_k, k = 1 \text{ to } i) \leq D_i \]
Limitation of Jackson’s Rule

- The set of all tasks is not always presented in advance.
- New tasks may arrive at arbitrary times.
- To minimize maximum lateness in this setting, it may be necessary to preempt a task already being executed.
- This issue is addressed by **Horn’s rule**.

Horn’s Rule (1974)

- Arrange execution, using **preemption** if necessary, so that:
  - At every instant the task with the current earliest deadline is being executed.
  - Horn’s rule can be proved to minimize maximum lateness in a manner similar to the proof of Jackson’s rule.

Horn’s Rule (1974)

- Horn’s rule is based on preemptability of tasks.
- If **preemption is not allowed**, then Horn’s rule does not minimize maximum lateness.

Example:

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
<th>Deadline</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>T₂</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

- Horn’s rule says: start T₁ at time 0, which would make T₂ wait to time 4. The max lateness would be 1.
- The optimum way would be do nothing at time 0, start T₂ at time 1, then start T₁ at time 3. The maximum lateness would be 0.

Exercise

- What happens in the previous example if preemption is allowed?

Hard Problem

- Finding an **optimal non-preemptive** schedule when arrival times are arbitrary is NP-hard.
- An enumerative, branching, algorithm can be used.
Lawler's Algorithm (1973)

- Schedules a set of simultaneously arriving tasks on one process subject to precedence constraints.

- Minimizes maximum lateness for 1 processor, among all non-preemptive algorithms.

Example: Lawler's Algorithm

Assume all computation times are 1 and deadlines are as shown.

Stacking order (LDF, obeying precedence):

\[ T_6, T_5, T_3, T_4, T_2, T_1 \]

Execution order:

\[ T_1, T_2, T_4, T_3, T_5, T_6 \]

Completion:

\[ 1, 2, 3, 4, 5, 6 \]

Using EDF, the order would be: \[ T_1, T_3, T_2, T_4, T_5, T_6 \]

Completion:

\[ 1, 2, 3, 4, 5, 6 \]

One task would be late, so max. lateness = 1.
### Other Algorithms

- An optimal method for the **preemptive** arbitrary-arrival case is also known (run time $O(n^2)$).

- This and the previous methods may be found in reference:
  - G.C. Buttazo
  - *Hard Real-Time Computing Systems*

### Periodic Tasks

- Many realtime systems are based on **periodic** tasks, where each task $T_i$ has:
  - Computation time $C_i$
  - Period $P_i$
  - Phase $\phi_i$

- The meaning of “period” is that, for each $T_i$, it must execute once every $P_i$ units.

- The meaning of “phase” is that, for each $i$, the earliest time at which $T_i$ is available within the current period is at relative time $\phi_i$.

### “Release Time”

- “Phase” is typically not used to describe real-time schedulers, although it has an obvious engineering significance.

- Instead, “release time” is used to mean “the time at which a task is next available for scheduling”; the “release” is releasing the task into the system, not out of it.

- In less we indicate otherwise, the release time for a task will coincide with the beginning of the period for the task, i.e., a periodic task with duration 2 and period 20 will be “released” at times 0, 20, 40, ...

### Preemptability

- Preemptability is a common assumption in real-time systems:

  A lower priority task may need to be preempted to give the processor to a higher priority one.

- Typically it is assumed that preempted tasks can be **resumed**.

### Preemption Costs

- In general, there will be a cost (delay) associated with preempting a task.

- For now, we assume that this cost is negligible.

### Utilization

- The **utilization** in a system of periodic tasks by a task is defined as

  \[
  \text{utilization} = \frac{\text{compute time}}{\text{period}}
  \]

- A necessary condition for a set of period tasks to be schedulable is that the **sum** of their utilizations be $\leq 1$. 
Exercise

Three periodic tasks:

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>T₂</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>T₃</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

See if you can construct a (1 processor) schedule that schedules these tasks periodically.

Remember that tasks can be preempted.

EDF Revisited

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 priorities 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.

Example

Constructing An EDF Schedule

An EDF Schedule
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.

Rate-Monotonic Assumption (RMA)

- A task that is preempted by another is said to have lower priority.
- Obviously tasks with shorter periods should generally have higher priority, since lower priority tasks can be preempted, then resumed, to allow higher priority tasks to meet their periodic deadlines.
- Assuming that shorter period \( \Rightarrow \) higher rate is called the rate-monotonic assumption (RMA).

Note on RMA Assumption

- RMA is a mathematical notion.
- It should not be inferred that every set of user priorities will agree with RMA.
- However, if priorities don’t align this way, it may be best to revisit the required periodicities.

Exercise

- Revisit your schedule from the previous exercise. Is it rate-monotonic?

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>bound on total utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.828427</td>
</tr>
<tr>
<td>3</td>
<td>0.779763</td>
</tr>
<tr>
<td>4</td>
<td>0.756828</td>
</tr>
<tr>
<td>8</td>
<td>0.724062</td>
</tr>
<tr>
<td>16</td>
<td>0.707472</td>
</tr>
<tr>
<td>32</td>
<td>0.700709</td>
</tr>
<tr>
<td>64</td>
<td>0.696914</td>
</tr>
<tr>
<td>( \infty )</td>
<td>( \ln(2) = 0.693147 )</td>
</tr>
</tbody>
</table>
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 $\sum(C_i/P_i, i = 1 to 1) \leq 1$.

RMA Rationale

- Case of $n$ tasks, $T_1,..., T_n$, 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 $\lceil t/P_i \rceil$ times.
  - The total time used for $T_i$ during the interval will be $C_i \lceil t/P_i \rceil$.
  - In order for each $T_i$ to execute the appropriate number of times in the interval, we need $(\forall t' < t) t' \geq \sum(C_i \lceil t/P_i \rceil)$.
  - If we can find a $t \leq \text{lcm}(P_1, \ldots, P_n)$ having this property, then all tasks can be scheduled.

Observation

- $(\forall t' < t) t' \geq \sum(C_i \lceil t/P_i \rceil)$
  iff $(\forall t' < t) t'$ is a time corresponding to the end of some task's period $\Rightarrow t' \geq \sum(C_i \lceil t/P_i \rceil)$.

- So it suffices to consider only times that end the period of some task.

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 $\lceil 300/100 \rceil = 3$ 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?

<table>
<thead>
<tr>
<th>task</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>20</td>
<td>100</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>end</td>
<td>20</td>
<td>50</td>
<td>120</td>
<td>180</td>
<td>220</td>
</tr>
</tbody>
</table>

Example

- 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 $21 \times 20 + 14 \times 30 + 10 \times 80 = 420 + 720 + 800 = 1940 \leq 2100$. So it is at least plausible that there is a schedule.

- On the next page, we construct a schedule.
A Rate-Monotone Schedule

We didn't have to compute that whole thing: Lehoczky, Sha, and Ding Theorem (1987)

- Under 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.

Priorities Revisited

- RMA is an example of a fixed priority scheme: priority is determined in order opposite periods.
- EDF is an example of a dynamic priority scheme.
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.

P1 = 5, C1 = 2
P2 = 7, C2 = 4

Example

P1 = 5, C1 = 2
P2 = 7, C2 = 4

5 0
5 0
0 10
missed deadline

RM

EDF

34/35 = 0.97 utilization
lcm(5, 7) = 35

non-preemption due to deadline (7 < 10), violation of rate-monotonicity

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 \(\text{lcm}(\text{periods})\).

### Summary of Applicability

<table>
<thead>
<tr>
<th>Schedulability</th>
<th>Deadline = Period</th>
<th>Deadline (\leq) Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Priority Rate Monotonic</td>
<td>EDF</td>
<td>EDF</td>
</tr>
<tr>
<td>Dynamic Priority EDF</td>
<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 $T_1$ blocks $T_2$, and $T_2$ blocks $T_3$, then $T_1$ inherits the priority of $T_3$.
  - Note that transitive inheritance occurs only with nested critical sections (different semaphores). If there were only one semaphore, then $T_1$ would be blocking $T_3$ directly.

---

### PIP Example

<table>
<thead>
<tr>
<th>Higher base priority</th>
<th>$T_0:0$</th>
<th>$T_1:1$</th>
<th>$T_2:2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$ priority</td>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$T_2$ priority</td>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$T_3$ priority</td>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$S_1$</td>
</tr>
</tbody>
</table>

- **P = preemption**
- **A = acquisition**
- **R = request**
- **X = release (of semaphore)**

### 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 LWP's as schedulable units of processor time ("slots").
- A new LWP can be created as an optional aspect of creating a new thread.

Solaris LWP’s vs. Processes

- 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.

Example: Solaris Operating System

- 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.

**From VxWorks FAQ**

The priority inheritance protocol also accounts for the ownership of more than one mutual exclusion semaphore at a given time. A task in such a situation will execute at the priority of the highest priority task blocked on any of the owned resources. The task will return to its normal, or standard, priority only after relinquishing all of the mutual exclusion semaphores with the inversion safety option enabled.

If you use nested mutex semaphores with priority inheritance turned on then when a task inherits a high priority due to some inner semaphore it owns, it doesn’t lose that priority until it relinquishes all the semaphores it holds. This doesn’t quite follow the rules for priority inheritance (to the extent that there really are any rules) in that normally, a task’s inherited priority should decrease as it releases each nested semaphore to whatever the priority ceiling is for the semaphores it still holds. Getting this incremental priority reduction to work right in practice, though, is extremely difficult (some of the SUN papers on the Solaris real time scheduling indicate that this was one of the hardest things for SUN to get right in their OS upgrades).

**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, extending the blocking time indefinitely.
- The PCP does prevent this: Once a task gains entry to a critical section, it cannot subsequently 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. Otherwise the task is “blocked” on the semaphore (it hasn’t really locked it yet).
- If a task is blocked on a semaphore, the task locking that semaphore inherits the priority of the blocked task (as in PIP).

**Priority Ceiling Protocol (PCP)**

- When a task locking a semaphore unlocks it, its priority is set to that of the highest priority resource that it now holds (or to its base priority, if none).
- Priority inheritance is constrained to be transitive, as with the PIP.

**PIP Example**

<table>
<thead>
<tr>
<th>T₀:0</th>
<th>T₁:1</th>
<th>T₂:2</th>
<th>T₃:3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₀:0</td>
<td>T₁:1</td>
<td>T₂:2</td>
<td>T₃:3</td>
</tr>
<tr>
<td>P</td>
<td>A</td>
<td>A</td>
<td>PR</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- T₀:0 higher base priority
- T₁:1 using semaphores S₁, ...
- T₂:2 requesting semaphores S₁, ...
- T₃:3 holding semaphores S₁, ..., but don’t have processor

**Legend**

- P = preemption
- A = acquisition
- R = request
- X = release (of semaphore)
PCP Example

T0: higher base priority
S0: lower number = higher priority
T1: priority: 2

T0 blocked because S2 has ceiling of 0.
T2 inherits T0’s priority.

Were this PIP, T0 would not block here.

T1 blocked: S1 has ceiling of 1, but T1’s priority is 1, not higher.
T2 inherits T1’s priority.

T2 released
S1 released
S2 released

Comparison: PIP vs. PCP

- PCP is more efficient at run-time, in that a high priority task cannot be blocked more than once at the same level.
- 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).
- PCP automatically prevents deadlocks, since it induces an ordering on the way that resources can be requested.

References


Comparison: PIP vs. PCP

- PCP is more efficient at run-time, in that a high priority task cannot be blocked more than once at the same level.
- 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).
- PCP automatically prevents deadlocks, since it induces an ordering on the way that resources can be requested.

An Implementation of PCP (in MKS)

`pthread_mutex_setprioceiling()`

set the priority ceiling of a mutex.

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *mutex, int ceiling, int *oldceiling);
```

DESCRIPTION

The `pthread_mutex_setprioceiling()` function either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex, then it changes the mutex’s priority ceiling and releases the mutex. When the change is successful, the previous value of the priority ceiling is returned in oldceiling. The process of locking the mutex need not adhere to the priority ceiling protocol.

In the Commercial World
(e.g. Tri-Pacific Software PERTS simulation tool)

Included in the new PERTS 3.0 is support for BIP, Basic Inheritance Protocol, that allows users to analyze systems with single nodes and have dependencies between tasks that are not necessarily in an end-to-end or path dependency structure. BIP, a simpler version of PCP (Priority Ceiling Protocol), is easier to implement from an application standpoint and mirrors more closely the resource protocol in VxWorks. PCP allows for one blocking instance, at most, and no deadlock situations regarding resources. With BIP, on the other hand, a task can be blocked multiple times for each instance, as well as deal with resource contention deadlocks. PERTS 3.0 offers Rate Monotonic and Deadline Monotonic analysis with support for BIP, PCP, and SBP (Stack Based Protocol).

SRP (Stack Resource Protocol)

- All tasks share a single stack
- Greatly simplifies stack management
- However, a task cannot suspend itself (which would “block” the stack)
- Similar to PCP
  - PCP: Task is blocked when it wants to lock resource, vs.
  - SRP: Task is blocked when it attempts to preempt.