Timing and scheduling

Tajana Simunic Rosing
Department of Computer Science and Engineering
University of California, San Diego.
ES Design

Hardware components

Software Components

Verification and Validation
The scheduling problem

- Basic issue: can we meet deadlines?
  - Related problem: How much horsepower do we need to meet our deadlines?

- Why schedule?
  - CPU is shared among several processes.
    - Cost, Energy/power, Physical constraints.
  - Distribution of CPU time to processes.
    - Co-operation between processes.
    - RTOS.
Embedded vs. GP scheduling

- Priorities determine scheduling policy
  - CPU goes to highest priority process that is ready
  - Fixed priority vs. time-varying priorities.

- Workstations avoid starving processes of CPU
  - Fairness = access to CPU.

- Embedded systems must meet deadlines.
  - Low-priority processes may not run for a long time.
  - Real-time OS
    - Clear understanding of task & event timing
Timing and Clocks
Actions, Events, Order

- **Action** is a function or task that performed by a system.
- **Event** is an instance of an action.
  - Instances are commonly labeled using time stamps and action values.
- An **order** is a binary relation between two events.
- Two events are **temporally ordered** if the respective time instants are not identical on a directed timeline.
- Two events are **causally ordered** if one event is caused by the other (primary or causative) event.
  - Induced by order on respective actions.
  - Stronger condition than temporal ordering.
- **Delivery order** is defined by the communication system between system components.
Clocks

- Physical clock
  - a clock contains a *counter* and a physical mechanism that periodically generates an event to increase the counter
  - the periodic event is called a *microtick* of the clock
  - granularity = duration between two microticks

- Reference clock
  - defined by its adherence to a standard
  - for a clock with $10^{15}$ microticks per second the granularity of the clock is 1 femtosecond.
Clock Properties

- **Offset** between two clocks with the same granularity
  - the time difference between the two clocks

- **Precision** of a set of clocks is the maximum offset between any two clocks in the set
  - Local precision maintained through internal synchronization.

- **Accuracy** of a clock
  - Maximum drift with respect to the reference clock
  - Maintained through external synchronization.
Drift

- Drift of a physical clock is the frequency ratio between it and the ref. clock at any instance.
  - a good clock has a drift of close to 1
    - drift rate = drift - 1
    - Perfect clock has a drift rate of 0.
    - Typically drift rate is within $10^{-2}$ to $10^{-7}$ sec/sec.

- Example:
  - During the Gulf war on February 25, 1991, a Patriot missile defense system failed to intercept an incoming scud rocket.
    - The clock drift over a 100 hour period resulted in a tracking error of 678 meters.
    - The original requirement was resynchronization over 14 hour intervals (mission time).
Clock synchronization in distributed systems

- Distributed systems drift:
  - Relative to each other
  - Relative to a real world clock

- Two ways to solve the problem
  - State correction
    - Agree on a time and jump to it
      - discontinuities in time
  - Rate correction
    - Speed up/slow down to converge
    - Hard to implement, but less problems
    - E.g. GPS time is rate steered with accuracy 200ns to 1us
Clock synchronization in distributed systems

- **Network Time Protocol (NTP)**
  - Used for Internet time synch – within 10ms
  - Relies on GPS time servers
    - GPS within 200ns accuracy
    - Need clear sky view
    - Several min to setup time
    - Higher power requirements

- **802.11 broadcast synch**
  - Time Synch Function
    - 4ms max clock offset
    - If beacon’s timestamp is later than the station’s then the station sets its TSF timer to the beacon’s
Logical Time & Logical Clocks

- A system consists of a set of **processes**
  - process produces a sequence of **events**

- Logical time is where time progress is by events.
  - no event = no time progress
  - the events are causally related

- A system of logical clocks consists of a time domain, T, and a logical clock, C.
  - elements of T form a partially-ordered set over the relation “has happened before”
  - C is a function that maps an event, e, to an element of T
    - C(e) is called the time-stamp of event e.
Logical clocks

- Monotonically increasing counter
- No relation with real clock
- Each process keeps its own logical clock
  Cp used to timestamp events
Synchronizing logical clocks

- Understand the ordering of events
- “happens before” notion
  - Concurrency using timestamps
  - Not easy in distributed systems
    - No guarantees of synchronized clocks
    - Communication latency
Logical Clock Implementation

- Consists of:
  - data-structure local to every process for modeling clock(s)
    - a local logical clock that helps process measure its own progress
    - a global logical clock that represents process’s view of the global logical time
  - a protocol to update the clock-related data structures to ensure consistency:
    - R1: how does a process update its local logical clock?
    - R2: how does a process update its global logical clock?

- There are several implementations of logical clocks
  - Lamport’s Scalar Time.
  - Vector time
  - Matrix time – large overhead, good for distributed garbage collection
Scalar Time

- Allows determination of a total order of events in a distributed system.
- Time domain consists of a set of non-negative integers
  - Local and global logical clocks use a single integer variable $C$ per each process $P$
- Protocol rules are implemented as follows:
  - R1: before executing an event the process increments the clock:
    - $C \leq C + d$ where $d > 0$; typically, $d = 1$
  - R2: each message contains the clock value of its sender at sending time.
    - Receiving process sets its clock to the maximum of received clock value or its own clock, executes R1 and proceeds to deliver the message.
Scalar time evolution

- Lamport’s logical clock
Vector time

- For each process pi, vector maintains logical time of process and pi’s latest knowledge of every other pj
- Tracks casual dependencies exactly
- Used in distributed debugging, global breakpointing, checkpoint consistency for recovery etc.
Vector time example
Program execution time estimation
Execution time of a program

- **WCET**: worst case execution time
  - ensure deadlines are met – accuracy may be safety-critical, assess real-time system resource needs
- **BCET**: best case execution time
  - benchmark software & hardware, evaluate resource needs for non/soft real-time systems
- **ACET**: average case execution time
Worst case execution time

- **Worst case execution time** (WCET) is an **upper bound** on the execution times of tasks
  - in the general case computing WCET is undecidable.
  - HW needs to be synthesized first
  - SW requires complex program analysis

<table>
<thead>
<tr>
<th>t</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>feasible execution times</td>
<td>WCET’ (some tighter bound)</td>
</tr>
<tr>
<td></td>
<td>Average-case execution time (ACET)</td>
</tr>
<tr>
<td></td>
<td>Some tighter lower bound for best case</td>
</tr>
</tbody>
</table>
Estimating WCET & BCET

- Approaches for approximating WCET or BCET:
  - Measuring: Measure run time of program on target hardware
    - Call OS timers, use HW timers, use external HW, count emulator cycles,
    - Do high water marking: continuously record actual execution times & read at service intervals; this is standard in many RTOS implementations
  - Analysis: Compute an estimate of run time based on program analysis and model of target hardware -> complex and inexact

- Key challenges:
  - Program execution depends on inputs – carefully choose data sets
  - Program context affects execution – cache, pipeline etc.
Flow analysis: dynamic behavior of the program

- Loop iterations, recursion depth, input dependencies, infeasible paths, function instances
- Information from static analysis and manual annotations
- Analyzed at object and source code levels
Obtaining WCET: Low-level Analysis

- Determine execution time of program parts
  - Accounts for HW effects
  - Work on object code
  - Exact results are not possible

- Local: affect single instruction + neighbors
  - pipeline effects

- Global: reaches across entire program
  - e.g. cache, branch predictors, TLBs
Obtaining WCET: Calculation Step

- Find the path that gives the longest execution time
- Approaches:
  - Structure-based
  - Path-based
  - Constraint-based (Implicit path enumeration technique - IPET)

For more info see:
Scheduling
Scheduling

- A schedule reserves spatial and temporal resources for a given task set

- Scheduler decides the order of task execution, dispatcher starts task execution
Schedule properties

- **Feasible** if it fulfils all application constraints for a given set of tasks
- A set of tasks is **schedulable** if there is at least one feasible schedule
- **Optimal** if a feasible schedule is found whenever any other scheduling algorithm can do so
Classification of schedulers

- A time-constraint (deadline) is called **hard** if not meeting that constraint could result in a catastrophe [Kopetz, 1997].
- All other time constraints are called **soft**.
Periodic and aperiodic tasks

- Tasks which must be executed once every $p$ units of time are called **periodic** tasks. $p$ is called their period. Each execution of a periodic task is called a **job**.

- All other tasks are called **aperiodic**.
Preemptive and non-preemptive

- **Non-preemptive schedulers:**
  Tasks are executed until they are done so response time for external events may be quite long.

- **Preemptive schedulers:**
  - Use if some tasks have long execution times or the response time for external events needs to be short.
Static and dynamic scheduling

- **Dynamic scheduling**: done at run-time.
- **Static scheduling**: done at design-time.
  - Dispatcher allocates processor on timer interrupt
  - Timer controlled by a table generated at design time.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>start T1</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>send M5</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>stop T1</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>start T2</td>
<td>20</td>
</tr>
<tr>
<td>47</td>
<td>send M3</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing scheduling examples]
Classification of Schedulers with respect to task dependencies

- **Independent Tasks**
  - EDD, EDF, LL, RMS

- **Dependent Tasks**
  - **Resource constrained**
    - Single CPU
      - LDF
  - **Time constrained**
    - LS
  - **Unconstrained**
    - FDS
    - ASAP, ALAP
Aperiodic scheduling with no precedence constraints

- Let \( \{T_i\} \) be a set of tasks. Let:
  - \( c_i \) be the execution time of \( T_i \),
  - \( d_i \) be the **deadline interval**, that is, the time between \( T_i \) becoming available and the time until which \( T_i \) has to finish execution.
  - \( l_i \) be the **laxity** or **slack**, defined as \( l_i = d_i - c_i \)

![Diagram showing availability of Task i and the parameters c_i, d_i, l_i]
Uniprocessor with equal arrival times

- **Earliest Due Date (EDD) - Jackson's rule:**
  - Any algorithm that executes a set of $n$ independent tasks in order of increasing deadlines is optimal with respect to minimizing the maximum lateness. Proof: [Buttazzo, 2002]
  - Maximum lateness is $<0$ if all tasks complete on time
    \[
    \text{Max Lateness} = \max_{\text{all tasks}} (\text{completion time} - \text{deadline})
    \]
  - EDD requires all tasks to be sorted by their deadlines.
    - complexity is $O(n \log(n))$. 

\[\text{Diagram}]}
Earliest Deadline First (EDF)

- Different arrival times - preemption can reduce lateness.

**Theorem [Horn74]:**

- Any algorithm that at any instant executes a task with the earliest absolute deadline among all the ready tasks in set $n$ is optimal with respect to minimizing the maximum lateness.

**Earliest deadline first (EDF) algorithm:**

- Insert each new task into a queue of ready tasks, sorted by their deadlines.
- If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted.
- If sorted lists are used the complexity is $O(n^2)$
Earliest Deadline First (EDF)

<table>
<thead>
<tr>
<th></th>
<th>arrival</th>
<th>duration</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>0</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>T3</td>
<td>5</td>
<td>10</td>
<td>29</td>
</tr>
</tbody>
</table>

Task arrivals

- Earlier deadline: preemption
- Later deadline: no preemption
Earliest Deadline First (EDF)

<table>
<thead>
<tr>
<th>Task</th>
<th>Deadline</th>
<th>Period</th>
<th>Exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tajana Simunic Rosing
Least laxity (LL), Least Slack Time First (LST)

- Priorities are dynamically changing and are in decreasing function of slack.
- Preemptive, detects missed deadlines early.
- LL is also an optimal scheduler for mono-processor systems.
  - Uses dynamic priorities so it cannot be used with a fixed priority OS.
- LL scheduling requires the knowledge of the execution time.
  - Might not know this in advance.
Periodic Task Scheduling
Characterizing the Task Set

- Set on $n$ independent tasks $\tau_1, \tau_2, \ldots, \tau_n$
- Request periods are $T_1, T_2, \ldots, T_n$
  - Request rate of $\tau_i$ is $1/T_i$
- Run-times are $C_1, C_2, \ldots, C_n$
- Utilization:
  - Accumulated execution time divided by the period:
  - Necessary condition for schedulability (with $m=$number of processors):

\[ \mu \leq m \]
Rate monotonic (RM) scheduling

- Assumptions:
  - All tasks that have hard deadlines are periodic.
  - All tasks are independent.
  - \( d_i = p_i \), for all tasks.
  - \( c_i \) is constant and is known for all tasks.
  - The time required for context switching is negligible.
  - For a single processor and for \( n \) tasks, the following equation holds for the accumulated utilization \( \mu \):
    \[
    \mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)
    \]

- Establishes a condition for schedulability!
  - \( \lim_{n \to \infty} \mu \sim 0.7 \)
RM Scheduling

**RM policy**: The priority of a task is a monotonically decreasing function of its period.

*low period = high priority*

At any time, a highest priority task among all those that are ready for execution is allocated.

When all assumptions are met, schedule exists!

Maximum utilization as a function of the number of tasks:

\[ \mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1) \]

\[ \lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2) \]
Theorem: For a set of independent periodic tasks, if a task meets its first deadline when all the higher priority tasks are started at the same time, then it meets all its future deadlines with any other task start times.

Demand on the system at time $t$ is defined as a function of the number of times a task $i$ arrives to the system $\left\lfloor \frac{t}{p_i} \right\rfloor$

$$W_n(t) = \sum_{i=1}^{n} c_i \left\lfloor \frac{t}{p_i} \right\rfloor$$

Goal: Find the minimum $t$, where $W_i(t) = t$ as follows:

Set $t_0 \leftarrow \sum_{j=1}^{i} C_j$

$t_1 \leftarrow W_i(t_0)$
$t_2 \leftarrow W_i(t_1)$
$t_3 \leftarrow W_i(t_2)$

... 

$t_k \leftarrow W_i(t_{k-1})$

Stop when $(W_i(t_k) = t_k)$
Example of RM schedule

T1 preempts T2 and T3.

T2 and T3 do not preempt each other.

<table>
<thead>
<tr>
<th>Period</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: 2</td>
<td>0.5</td>
</tr>
<tr>
<td>T2: 6</td>
<td>2</td>
</tr>
<tr>
<td>T3: 6</td>
<td>1.75</td>
</tr>
</tbody>
</table>
Case of failing RM scheduling

Task 1: period 5, execution time 2
Task 2: period 7, execution time 4
\[ \mu = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} \approx 0.97 \]
\[ 2(2^{1/2} - 1) \approx 0.828 \]

\[ \mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1) \]

Not enough idle time!

Missed deadline

Missing computations scheduled in the next period
Properties of RM scheduling

- RM scheduling is based on **static** priorities.
  - can be used in a standard OS
  - many variations of RM scheduling exists.

- In the context of RM scheduling, many formal proofs exist.
  - Idle capacity is not needed if periods of all tasks are multiples of the period of the highest priority task
RM in Distributed/Networked Embedded Systems

- Task is scheduled on multiple resources in series
- Need to schedule communication messages
  - propagation delay & jitter
  - queuing delay & jitter
- Divide end-to-end deadline into subsystem deadlines
- Buffering to mitigate jitter problem as task may arrive too early
EDF for periodic scheduling

- Optimal for periodic scheduling
- EDF is able to schedule the example in which RMS failed.
- EDF requires dynamic priorities
  - cannot be used with operating system providing only static priorities.
- Sufficient and necessary condition for uniprocessor scheduling with EDF under assumptions:
  - All tasks are periodic, independent and with deadlines equal to periods

\[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \]
Comparison EDF/RMS

RMS:

T1

T2
J2,1

J2,2

EDF:

T1

T2

T2 not preempted, due to its earlier deadline.
Sporadic tasks

- If sporadic tasks were connected to interrupts, the execution time of other tasks would become very unpredictable.
  - Introduction of a sporadic task server, periodically checking for ready sporadic tasks;
  - Sporadic tasks are essentially turned into periodic tasks.
Dependent Task Scheduling
Classification of Schedulers

Independent Tasks
- EDD, EDF, LL, RMS

Dependent Tasks
- Resource constrained
  - ASAP, ALAP
- Time constrained
  - FDS
- Unconstrained
  - Single CPU
    - LDF
  - LS
  - LDF
Dependent tasks

The problem of deciding whether or not a schedule exists for a set of dependent tasks and a given deadline is NP-complete in general [Garey/Johnson].

Strategies:

1. Add resources, so that scheduling becomes easier
2. Split problem into static and dynamic part so that only a minimum of decisions need to be taken at run-time.
3. Use scheduling algorithms from high-level synthesis
Classes of task mapping algorithms

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for mono- and homogeneous multiprocessors

- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable

- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems, focus on resource assignment

- **Design space exploration using genetic algorithms**
  Heterogeneous systems, incl. communication modeling
Latest Deadline First (LDF) Algorithm

- Among the tasks with no successors insert the one with the latest deadline into a queue. Repeat this process, putting tasks whose successors have all been selected into the queue.
- At run-time, the tasks are executed in the generated total order.
- LDF is non-preemptive and is optimal for single processor systems.

If no local deadlines exist, LDF performs just a topological sort.
Asynchronous Arrival Times: Modified EDF Algorithm

- Transform a set of dependent tasks into a set of independent tasks with different timing parameters
- Optimal for single processor systems.
- Heuristics available when no preemption
As soon as possible (ASAP)

- ASAP: All tasks are scheduled as early as possible

- Loop over (integer) time steps:
  - Compute the set of unscheduled tasks for which all predecessors have finished their computation
  - Schedule these tasks to start at the current time step.
ASAP scheduling example
As-late-as-possible (ALAP)

ALAP: All tasks are scheduled as late as possible

Start at last time step*:

Schedule tasks with no successors and tasks for which all successors have already been scheduled.

* Generate a list, starting at its end
ALAP scheduling example

\[
\begin{align*}
\tau &= 0 \\
\tau &= 1 \\
\tau &= 2 \\
\tau &= 3 \\
\tau &= 4 \\
\tau &= 5
\end{align*}
\]
Resource Constrained: List Scheduling

List scheduling: extension of ALAP/ASAP

Preparation:

- Topological sort of task graph $G=(V,E)$
- Computation of priority of each task:

  Possible priorities $u$:
  - Number of successors
  - Longest path
  - Mobility $= \tau$ (ALAP schedule) - $\tau$ (ASAP schedule)
Mobility as a priority function

Mobility is not very precise
List Scheduling Algorithm

List($G(V,E)$, $B$, $u$){
  $i := 0$;
  repeat {
    Compute set of candidate tasks $A_i$;
    Compute set of not terminated tasks $G_i$;
    Select $S_i \subseteq A_i$ of maximum priority $r$ such that
    $|S_i| + |G_i| \leq B$ (*resource constraint*)
    foreach ($v_j \in S_i$): $\tau(v_j):=i$; (*set start time*)
    $i := i + 1$;
  }
  until (all nodes are scheduled);
  return ($\tau$);
}

Complexity: $O(|V|)$
List Scheduling Example

- Assuming $B = 2$, unit execution time and $u : \text{path length}$

- $u(a) = u(b) = 4$
- $u(c) = u(f) = 3$
- $u(d) = u(g) = u(h) = u(j) = 2$
- $u(e) = u(i) = u(k) = 1$
- $\forall i : G_i = 0$
Time constrained: Force-directed scheduling

- **Goal**: balanced utilization of resources
- **Assumes** time constraints are known
- **Originally** for high-level synthesis
- **Based on** spring model:

```
procedure forceDirectedScheduling;
  begin
    AsapScheduling;
    AlapScheduling;
    while not all tasks scheduled do
      begin
        select task $T$ with smallest total force;
        schedule task $T$ at time step minimizing forces;
        recompute forces;
      end;
  end;
```

May be repeated for different task/processor classes
Force-directed scheduling steps

1. Compute time frames $R(j)$
   
   $R(j) = \{\text{ASAP-control step} \ldots \text{ALAP-control step}\}$

2. Compute probability $P(j,i)$ of assignment $j \rightarrow i$

   $$P(j, i) = \begin{cases} 
   \frac{1}{|R(j)|} & \text{if } i \in R(j) \\
   0 & \text{otherwise}
   \end{cases}$$
Force-directed scheduling steps

3. Compute “distribution” $D(i)$ - # Operations in control step $i$)

$$D(i) = \sum_{j, \text{type}(j) \in H} P(j,i)$$

<table>
<thead>
<tr>
<th>Step</th>
<th>Operations</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2 5/6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2 2/6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>5/6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
4. Compute overall forces as a function of distribution and probabilities previously computed

- Total forces are a sum of direct and indirect forces

\[ F(j, i) = SF(j, i) + VF(j, i) + NF(j, i) \]

- Direct forces:

\[ SF(j, i) = \sum_{j' \in R(j)} D(i') \Delta P_i(j, i') \]

\[ \Delta P_i(j, i') = \begin{cases} 
1 - P(j, i) & \text{if } i = i' \\
-P(j, i') & \text{otherwise}
\end{cases} \]

- Indirect forces:

\[ VF(j, i) = \sum_{j' \in \text{predecessor of } j} \sum_{i' \in I} D(i') \Delta P_{j,i}(j', i') \]

\[ NF(j, i) = \sum_{j' \in \text{successor of } j} \sum_{i' \in I} D(i') \Delta P_{j,i}(j', i') \]

5. Schedule tasks to minimize forces
Dependent Task Schedulers

- Mostly heuristic
- Not using global knowledge about periods etc.
- Consider discrete time intervals
- Variable execution time available only as an extension
- Model heterogeneous systems
Scheduler Overview

- Scheduling of tasks with real-time constraints:

<table>
<thead>
<tr>
<th></th>
<th>Equal arrival times; non-preemptive</th>
<th>Arbitrary arrival times; preemptive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent tasks</td>
<td>EDD (Jackson), RM (periodic)</td>
<td>EDF (Horn)</td>
</tr>
<tr>
<td>Dependent tasks</td>
<td>LDF (Lawler), ASAP, ALAP, LS, FDS</td>
<td>EDF* (Chetto extensions)</td>
</tr>
</tbody>
</table>
Resource access management
Resource access protocols

- **Critical sections**: sections of code at which exclusive access to some resource must be guaranteed.
- Can be guaranteed with semaphores $S$.

Task 1

- $P(S)$ checks semaphore to see if resource is available
  - if yes, sets $S$ to "used".
  - if no, calling task has to wait.

Task 2

- $P(S)$ checks semaphore to see if resource is available
  - if yes, sets $S$ to "used".
  - if no, calling task has to wait.

- $V(S)$: sets $S$ to "unused" and starts sleeping task (if any).

Exclusive access to resource guarded by $S$
The MARS Pathfinder problem

- A few days into gathering meteorological data, the spacecraft began experiencing total system resets
- OS level preemptive priority scheduling of threads
- Problem:
  - Bus thread runs frequently; uses mutexes
  - Interrupt schedules a communication task for a short interval while the bus thread is blocked waiting for the data
  - Watchdog timer goes off if data bus task had not been executed for some time
    - initiates a total system reset

<table>
<thead>
<tr>
<th>High priority:</th>
<th>bus thread: retrieval of data from shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium priority:</td>
<td>communications task</td>
</tr>
<tr>
<td>Low priority:</td>
<td>thread collecting meteorological data</td>
</tr>
</tbody>
</table>
Priority inversion

- Priority $T_1 >$ priority of $T_2$.
- If $T_2$ requests exclusive access first (at $t_0$), $T_1$ has to wait until $T_2$ releases the resource (time $t_3$), thus inverting the priority:

 Duration of inversion bounded by length of critical section of $T_2$. 

![Diagram showing priority inversion](image)
Priority inversion with >2 tasks

- Duration of priority inversion can exceed the length of the critical section
- Priorities: T1 > T2 > T3
  - T2 preempts T3; T2 can prevent T3 from releasing the resource.
Priority inheritance example

- Schedule according to active task priorities.
  - Tasks inherit the highest priority of tasks blocked by it
  - Transitive: if $T_1$ blocks $T_0$ and $T_2$ blocks $T_1$, then $T_2$ inherits the priority of $T_0$.

T3 inherits the priority of $T_1$ and T3 resumes.
Priority inheritance on Mars

- Use a flag for the calls to mutex primitives
  - Set to on to allow priority inheritance
  - Default was “off”.

The problem on Mars was corrected by changing the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].
Lottery Scheduling

- Flexible proportional-share resource management
- Allocation of resource rights
  - determined by holding a lottery
  - allocates resources to competing clients in proportion to the number of tickets that they hold
- Scheduling by lottery is probabilistically fair
  - Binomial distribution of a number of lotteries won by a client
  - Geometric distribution of a number of lotteries required for a client’s first win
  - scheduling quantum is typically 10 ms (100 lotteries per second)
- Priority inversion solved by ticket transfer between clients
Real-time scheduling

- **Scheduling**
  - Rate monotonic scheduling
  - EDF
  - Dependent and sporadic tasks

- **Resource access**
  - Priority inversion
  - Priority inheritance
  - Lottery scheduling
Sources and References

- Peter Marwedel, “Embedded Systems Design”
- Nikil Dutt @ UCI
- Mani Srivastava @ UCLA
- Prof. Dr. Reinhard von Hanxleden @ CAU