CSE 237A
Timing and scheduling

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

Hardware components

Hardware

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
  - instantaneous events are partially ordered
- 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 <= 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 clocks are consistent but not necessarily strongly consistent.
Scalar time evolution

- Lamport’s logical clock
Vector time

- For each process $p_i$, vector maintains logical time of process and $p_i$’s latest knowledge of every other $p_j$
- Tracks casual dependencies exactly
- Used in distributed debugging, global breakpointing, checkpoint consistency for recovery etc.
Vector time example
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 scheduling algorithms

- Real-time scheduling
  - Hard deadlines
    - Periodic
      - Preemptive
        - Static
      - Non-preemptive
        - Dynamic
    - Aperiodic
      - Preemptive
        - Static
      - Non-preemptive
        - Dynamic
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:
- Dispatcher
- Timer
- Real-time scheduling
- Hard deadlines
- Soft deadlines
- Periodic
- Aperiodic
- Preemptive
- Non-preemptive
- Static
- Dynamic
Centralized and distributed scheduling

- **Centralized and distributed scheduling:**
  - multiprocessor scheduling either locally on one or on several processors.

- **Mono- and multi-processor scheduling:**
  - Simple algorithms for single or complex algorithms for multiple processors (e.g. CPU with HW accelerators).

- **Online & offline scheduling:**
  - Online is at run-time, based on the information about the tasks arrived so far.
  - Offline assumes prior knowledge about arrival times, execution times, and deadlines.
Cost functions

- **Cost function:** Different scheduling algorithms aim at minimizing different functions.
  - Minimize max lateness, min power while meeting deadlines etc

- **Def.: Maximum lateness =**
  \[
  \text{max}_{\text{all tasks}} (\text{completion time} - \text{deadline})
  \]
  Is <0 if all tasks complete before deadline.

```
T1
\[\text{-------}\]
T2
\[\text{-------}\]
```

Max. lateness
\[
\text{-------}\]
\[t\]
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.
- For HW need to synthesize first
- For SW requires complex program analysis
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 \( l_i = d_i - c_i \)

Availability of Task \( i \):

\[
\begin{align*}
\text{Availability of Task } i & \quad \text{d} \quad \text{i} \\
\text{i} & \quad \text{c} \quad \text{i} \quad \text{t} \\
\text{l} & \quad \text{i} \quad \text{d} \quad \text{i} \quad \text{t}
\end{align*}
\]
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]
  - EDD requires all tasks to be sorted by their deadlines.
  - Complexity is $O(n \log(n))$. 

![Diagram of task scheduling](image)
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

Later deadline

Earlier deadline

preemption

no preemption
Least laxity (LL), Least Slack Time First (LST)

<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>

Priorities are dynamically changing and in decreasing function of slack.
Preemptive, detects missed deadlines early.
LL is also an optimal scheduling 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.
Scheduling with precedence constraints

- Task graph with a possible schedule

Schedule can be stored in table.
Asynchronous Arrival Times: Modified EDF Algorithm

- Transform a set of dependent tasks into a set of independent tasks with different timing parameters.
- Optimal for mono-processor systems.
- Heuristics available when no preemption.
Best solution is to design an algorithm which will always find a schedule if one exists.

- Optimal scheduler will find a schedule if one exists.
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:
    \[
    \mu = \sum_{i=1}^{n} \frac{C_i}{P_i}
    \]
    Necessary condition for schedulability (with \( m=\text{number of processors} \)):
    \[
    \mu \leq m
    \]
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!
    - $\lim_{n \to \infty} n \mu \approx 0.7$
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:

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

\[
\lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2)
\]
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!
Properties of RM scheduling

- RM scheduling is based on **static** priorities.
  - can be used in 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:

EDF:

T2 not preempted, due to its earlier deadline.
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.
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.
Resource access protocols

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

- $P(S)$ 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).
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 $T_2$. 
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 T1 blocks T0 and T2 blocks T1, then T2 inherits the priority of T0.

T3 inherits the priority of T1 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].

Tajana Simunic Rosing
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 (briefly)

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

- Nikil Dutt @ UCI
- Mani Srivastava @ UCLA