CS557: Foundation of CPS

Realtime Systems and Scheduling

XiangYang Li
Slides are developed based on slides by others.
Special thanks to S.-P. Ren
Embedded vs. Real Time Systems

• Embedded system:
  - is a computer system that performs a limited set of specific functions. It often interacts with its environment.

• RTS:
  - Correctness of the system depends not only on the logical results, but also on the time in which the results are produced.

Examples?
Examples

• **Real Time Embedded:**
  - Nuclear reactor control
  - Flight control
  - Basically any safety critical system
  - GPS
  - MP3 player
  - Mobile phone

• **Real Time, but not Embedded:**
  - Stock trading system
  - Skype
  - Pandora

• **Embedded, but not Real Time:**
  - Home temperature control
  - Sprinkler system
  - Washing machine, refrigerator, etc.
  - Blood pressure meter
Characteristics of RTS

• Event-driven, reactive.
• High cost of failure.
• Concurrency/multiprogramming.
• Stand-alone/continuous operation.
• Reliability/fault-tolerance requirements.
• Predictable behavior.
Reliability and Safety

- **Reliability**: probability that the system will provide the specified service for a given time period. (Also see Failure Rate or Mean Time To Failure: MTTF)
- **Safety**: reliability regarding critical failure modes
- **Fail-safe system**: if the system has a guaranteed safe state that can be reached in case of a critical failure. It is a property of the controlled object and not the computer system.
  - Watchdog: external device that gets periodic life sign from the computer system. If it does not get it, it forces the controlled object into a safe state.
- **Fail-operational system**: no such safe state exists, so the computer system must provide (limited) functionality in case of failures to avoid a catastrophic failure.
- **Alarm monitoring.**
  - Primary event
  - Secondary alarms. Temporal order is very important. Alarm shower
  - Rare events
1: INTRODUCTION TO TIME
Definitions

• **Hard real-time** — systems where it is absolutely imperative that responses occur within the required deadline. E.g. Flight control systems.

• **Soft real-time** — systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Data acquisition system.

• **Real real-time** — systems which are hard real-time and which the response times are very short. E.g. Missile guidance system.

• **Firm real-time** — systems which are soft real-time but in which there is no benefit from late delivery of service.

A single system may have all hard, soft and real real-time subsystems. In reality many systems will have a cost function associated with missing each deadline.
Time

digital clock

tick

granule

now

instant

past

future

event

duration

event

duration

duration

duration

duration

duration

duration

duration

Time line
Time in Real-Time Systems

- Actions, Events and Orders
- Clocks
- Time Standards
- Time Measurements
- Logical Clocks
Actions, Events & Order

• An action: a function or task that performed by a system
• An event: an instance of an action
  instances are commonly labeled using time stamps and action values.

• Timeline
  Directed, an infinite set \{T\} of instances with the following properties:
  1. \{T\} is an ordered set, i.e., if \(p\) and \(q\) are any two instances, then either \(p\) is simultaneous with \(q\), or \(p\) proceeds \(q\), or \(q\) proceeds \(p\), where these relations are mutually exclusive.
  2. \{T\} is a dense set. This means that there is at least one \(q\) between \(p\) and \(r\) iff \(p\) is not the same instance as \(r\), where \(p\), \(q\), and \(r\) are instances.
Temporal order
- A binary relation between two events
- Instantaneous events (i.e., events without a duration) are *partially ordered* with no order among simultaneous events

Causal order
- Causal order is temporal order
  - With additional condition of locally finite
  - For all \( x, z \in C \) we have \( \text{card}(\{ y \in C | x \preceq y \preceq z \}) < \infty \)
- Temporal order is not necessarily a causal order
Actions, Events & Order

- 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
- Delivery order defines a weaker relation between two events:
  - defined by the communication system between system components
Clocks

• A physical clock is a device for measuring real time based on some physical phenomenon (atomic oscillations, e.g.)
  ▶ a clock tick may consist of several microticks

• A reference clock is defined by its adherence to a standard
  ▶ for a clock with 1015 microticks per second the granularity of the clock is 1 femtosecond.
    • Can not measure events at any smaller intervals.
Clock Failures

Time of reference clock

Error in drift

Perfect Clock

A good clock within a bounded drift rate stays within the shaded area.

Error in Clock Value / counter

Time of local clock
Clock Properties

- **Offset** between two clocks (at a given microtick) is the time difference between the respective microticks of the two clocks, measured in the number of microticks of the reference clock.

- **Precision** of a set of clocks (at a given microtick) is the maximum offset between any two clocks in the set
  - clocks in a set are maintained within a certain precision through internal synchronization.

- **Accuracy** of a clock is maximum drift with respect to the reference clock
  - maintained through external synchronization.
Time Measurement

• A **global time** refers to approximate measure of time generated by internally synchronized local clocks
  ‣ a group of microticks of a local clock defines a tick of the global time

• **Global time** is considered reasonable if all local implementations of global time have granularity larger than the precision of the local clocks
  ‣ ensures that the synchronization error using this time is bounded to within one tick of the global time
    • that is, the time stamps of an event sampled by any two local clocks can differ at most by 1 tick of the global time
  ‣ durations are bounded by $2 \times$ granularity of global time.

  *Can you explain why?*
Time Standards

- **International Atomic Time (TAI):** TAI is a physical time standard that defines the second as the duration of 9 192 631 770 periods of the radiation of a specified transition of the cesium atom 133. TAI is a chronoscopic timescale, i.e., a timescale without any discontinuities. It defines the *epoch, the origin of time measurement*, as January 1, 1958 at 00:00:00 hours, and continuously increases the counter as time progresses.

- **Universal Time Coordinated (UTC):** UTC is an astronomical time standard that is the basis for the time on the “wall clock”. In 1972 it was internationally agreed that the duration of the second should conform to the TAI standard, but that the number of seconds in an hour will have to be occasionally modified by inserting a leap second into UTC to maintain synchrony between the wall clock time and the astronomical phenomena, like day and night.
Clock Synchronization

• Why do we need to agree on time?

• Why do we need clock synchronization?
  ‣ time initially set incorrectly
  ‣ clock drifts over the course of time

• Clock synchronization:
  ‣ internal vs external
  ‣ master-slave vs distributed
  ‣ hardware vs software

• Goals:
  ‣ high precision: small deviation between clocks
  ‣ high accuracy: small deviation to external time
  ‣ monotonic and “continuous” time
Why Clock Synchronization?

Computer on which compiler runs

2144  2145  2146  2147

- output.o created

Computer on which editor runs

2142  2143  2144  2145

- output.c created

Time according to local clock
Physical Clocks

- A high-frequency oscillator, counter, and holding register
  - Counter decrements on each oscillation, generates a “tick” and is reset from the holding register when it goes to 0.
- On a network, clocks will differ: “skew”.
Clock Drift

\[
\frac{dC}{dt} > 1 \quad \text{Fast clock}
\]

\[
\frac{dC}{dt} = 1 \quad \text{Perfect clock}
\]

\[
\frac{dC}{dt} < 1 \quad \text{Slow clock}
\]
Clock Drift

- The *drift* of a physical clock, $k$, between utick $i$ and utick $i+1$ is the frequency ratio between $k$ and the reference clock, $z$, at instant $i$.

\[
drift^k_i = \frac{z(\text{utick}^k_{i+1}) - z(\text{utick}^k_i)}{n^k}
\]

- difference in the duration of a granule of clock $k$ with the reference clock $z$ divided by the nominal no. of reference clock uticks in a granule

- a good clock has a drift close to 1
  - drift rate = $|\text{drift} - 1|$
  - typically drift rate is within $10^{-2}$ to $10^{-7}$ sec/sec.
Clock Skew

- Computer clocks are not generally in perfect agreement
- Skew: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to clock drift (they count time at different rates)
- Clock drift rate: the difference per unit of time from some ideal reference clock
- Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
- High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec
Clock Synchronization

• **Two problems:**
  ‣ How do we synchronize multiple clocks to the “real” time?
  ‣ How do we synchronize with each other?
Clock Resynchronization

- $\rho$: rate of deviation from external time
- $R$: time between synchronizations
- $\Delta s$: precision after synchronization
- Skew increase by $2\rho R$ between two synchronizations
- Choose $R$ s.t. $\Delta s + 2\rho R < \text{required precision}$
- Loosely coupled systems:
  - inaccurate, expensive clock readings: use high accuracy clocks, synchronize seldom
- Tightly coupled systems:
  - accurate clock readings: use low accuracy clocks and synchronize often
- Rate adjustment:
  - required adjustment: $\Delta$
  - time between adjustments: $R$
  - adjust with rate $\Delta/R$
Synchronization in Synchronous System

- A synchronous distributed system is one in which the following bounds are defined:
  - the time to **execute each step** of a process has known lower and upper bounds
  - each **message** transmitted over a channel is received within a known bounded time
  - each process has a local clock whose **drift rate** from real time has a known bound
Synchronization in Synchronous System

- **Internal synchronization in a synchronous system**
  - One process $p_1$ sends its local time $t$ to process $p_2$ in a message $m$,
  - $p_2$ could set its clock to $t + T_{\text{trans}}$ where $T_{\text{trans}}$ is the time to transmit $m$
  - $T_{\text{trans}}$ is unknown but $\min \leq T_{\text{trans}} \leq \max$
  - Uncertainty $u = \max - \min$. Set clock to $t + (\max - \min)/2$ then skew $\leq u/2$
Asynchronous Systems: Cristian’s Method

- A time server $S$ receives signals from a UTC source
  - Process $p$ requests time in $T_0$ and receives $t$ in $T_1$ from $S$
  - $p$ sets its clock to $t + T_{\text{round}}/2$
Berkeley Algorithm

a) The time daemon asks all the other machines for their clock values.
b) The machines answer and the time daemon computes the average.
c) The time daemon tells everyone how to adjust their clock.
Averaging Algorithms

- Every $R$ seconds, each machine broadcasts its current time.
- The local machine collects all other broadcast time samples during some time interval, $S$.
- The simple algorithm:
  - the new local time is set as the average of the value received from all other machines.
- Improved algorithm:
  - Correct each message by adding to the received time an estimate of the propagation time from the $i^{th}$ source (extra probe messages are needed to use this scheme).
- One of the most widely used algorithms in the Internet is the Network Time Protocol (NTP).
  - achieves worldwide accuracy in the range of 1-50 msec.
Network Time Protocol (NTP)

- Uses a network of time servers to synchronize all processors on a net.
- Time servers are connected by a synchronization subnet tree. The root is adjusted directly. Each node synchronizes its children nodes.
Network Time Protocol

- Contact a time server to ask the time.
  - What is the major issue here?
Fault-Tolerant Clock Synchronization

- **Central Master Algorithm:**
  - master clock sends periodically its time value to all other clocks (not fault-tolerant).

- **Average Algorithm:**
  - each clock gets the time from all others and average difference and a local correction factor (almost fault-tolerant).

- **Fault-Tolerant Average Algorithm:**
  - sort times and remove \( k \) smallest and \( k \) largest values, compute average difference from rest (tolerates \( k \) Byzantine clocks).
Very Common Now: GPS

- Satellite system launched by military in early 1990’s, became public and inexpensive
- 24 active satellites, with atomic clocks, at 20,000 km
- Each satellite continuously broadcasts its position and time
- Can think of satellites as broadcasting the time
- Small radio receiver picks up signals from three satellites and triangulates to determine position
- Same computation also yields extremely accurate (accurate to a few milliseconds)

- Put two GPS receivers (or more) on a network
- Periodically, receivers broadcast the “true time”
- Other machines only need to adjust their clocks to overcome propagation delays for clock sync messages through the network!
- Well matched to the a-posteriori clock synchronization approach
LOGICAL TIME/CLOCKS
Logical Time & Logical Clocks

• A system consists of a set of *processes*
• Execution of a process produces a sequence of *events*
• The easiest characterization of logical time is one where time progress is by events.
  ▸ That is, no event => no time progress
  ▸ the events are causally related
Logical Time & Logical Clocks

- 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 $e$. 
Implementation of Logical Clocks

• An implementation consists of two things:
  ‣ data-structure local to every process for modeling clock(s)
  ‣ a protocol (methods) to update the clock-related data structures (to ensure consistency property)

• Logically one can think of data-structure support in a process for
  ‣ 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
Implementation of Logical Clocks

• A protocol consists of following two rules:
  † 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.
Lamport Scalar Time

- Scalar time representation allows one to determine a total order of events in a distributed system.
- Time domain, T, consists of a set of non-negative integers
- Local and global logical clocks of each process is implemented using an integer variable, C (one per process)
Scalar Time

- Protocol rules are implemented as follows:
  - R1: before executing an event (send, receive message or an internal event) the process executes the following:
    - $C := C + d$ where $d > 0$
    - typically, $d = 1$
  - R2: each message contains the clock value of its sender at sending time. When the receiving process receives a message, it set the maximum of received clock value or its own clock value as its new clock value, executes R1 and proceeds to deliver the message.
- Scalar clocks are consistent (show it) but not necessarily strongly consistent.
Reading Material

- Friedemann Mattern, *Virtual Time and Global States of Distributed Systems*, 1989
A Reference Model of Real-Time Systems

2: REFERENCE MODEL
Reference Model

• Modeling the system to focus on timing properties and resource requirements. Composed of three elements:
  ▸ workload model - describes applications supported by system
    • functional parameters
    • temporal parameters
    • precedence constraints and dependencies
  ▸ resource model - describes system resources available to applications
    • modeling resources (Processors and Resources)
    • resource parameters
  ▸ algorithms - defines how application uses resources at all times.
    • scheduling hierarchy
Tasks and Jobs

- Task \((T_i)\):
  - set of related jobs jointly provide function.

- Job \((J_{ij})\):
  - unit of work, scheduled and executed by system, characterized by the following parameters:
    - temporal parameters: timing constraints and behavior
    - functional parameters: intrinsic properties of the job
    - resource parameters: resource requirements.
    - interconnection parameters: how it depends on other jobs and how other jobs depend on it
Temporal Parameters

Job $J_i$: when it is released (release time, $r_i$), how long does it take to execute (execution time, $e_i$), and when it has to be finished with respect to its release time (relative deadline, $d_i$), or absolute deadline ($D_i = r_i + d_i$)

- **Release Time ($r_i$)**
  - Jittered: release range $[r_i^-, r_i^+]$
  - Fixed: released at $r_i^-$, or $r_i^+$
  - Sporadic: release time is random
Job/Task Temporal Parameters

- **Hard real-time:**
  - number and parameters of tasks are known at all time.

- **For Job J_i:**
  - $r_i$ - release time, may know range $[r^-, r^+]$ (*jitter*). For aperiodic/sporadic release or inter-release time is a random variable.
  - $d_i$ - absolute deadline
  - $D_i$ - relative deadline
  - $(r_i, d_i]$ - feasible interval
  - $e_i$ (also: $c_i$) - execution time. May know range $[e^-, e^+]$. Most deterministic models use $e^+$. 
Temporal Parameters

• **Execution time**
  - $E_i$ is the amount of time required to complete the execution of $J_i$
  - Execute alone and with all the resource
  - Usually only know the execution range $[e_i^-, e_i^+]$, not the actual execution time
    - Branches
    - Performance enhancing features (catch memory, pipeline)
    - Inputs
    - etc
Temporal Parameters

Job $J_i$: when it is released (release time, $r_i$), how long does it take to execute (execution time, $e_i$), and when it has to be finished with respect to its release time (relative deadline, $d_i$), or absolute deadline ($D_i = r_i + d_i$)

**Periodic Task Model**

- Periods $p_i$ of periodic task $T_i$ is the minimum length of all time intervals between release times of consecutive jobs in $T_i$
- Execution time is the maximum execution time of all the jobs in it
- $r_{ij}$ - release time of the $j^{th}$ Job in Task $i$ ($J_{ij}$ in $T_i$).
- Phase of periodic task $T_i$ is the release time $r_{i,1}$ of the first job $J_{i,1}$ in each task $T_i$, also denoted as $\Phi_i$
- Hyperperiod of periodic tasks: $H = lcm(p_i)$,
- The maximal number of jobs in each hyperperiod is $\sum_{i=1}^{n} = H/ p_i$
- Utilization of a period task: $u_i = e_i/ p_i$
Worst-Case Execution Time (WCET)

• Need to
  ▶ analyze and instrument the task
  ▶ analyze the compiler
  ▶ analyze the operating system
  ▶ analyze the hardware

• Analytical Approach:
  ▶ all sub-problems are solved analytically.

• Pragmatic Approach:
  ▶ investigate and instrument the source program to generate test cases that are biased towards the maximum execution time. Execute the test cases on the target system.
Aperiodic and Sporadic Tasks

• Aperiodic or Sporadic task is a stream of aperiodic or sporadic jobs
• Jobs with a task have similar statistical behavior and timing requirements

• Aperiodic:
  ‣ jobs have soft or no deadlines. Want responsiveness
• Sporadic:
  ‣ jobs have hard deadlines
**Precedence Constraints**

- Jobs are either constrained to execute in some order or independent

- Precedence relation: partial ordering operator $<$
  - $J_i < J_k$: $J_i$ is predecessor of $J_k$, $J_k$ is successor of $J_i$
  - Task graph
    - A graphical representation of applications
    - directed graph $G = (J, <)$
  - Other task models
Task Graph

- A directed graph that shows the **dependencies** between a number of functions (jobs)
  - $G=(V,E)$
    - Nodes ($V$): each node has input/output data ports
    - Arces ($E$): connections between the output ports and input ports
  - Semantics
    - Fire when input data are ready
    - There may be many nodes that are ready to fire at a given time
  - Precedence Graph vs Task Graph
    - Precedence graph: edges only for precedence relation
    - Task graph: different type of edges for different types of dependences
Task Graph
Data Dependency Example

\[ x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]

\[ x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \]
Other task models

- Finite state machine
- Petri net
Resources

• Resources can be divided into passive and active:
  
  ▶ Active resources == \textit{Processors} \((P_i)\): they execute jobs
    • every job must have one or more processors
    • same type if functionally identical and used interchangeably
  
  ▶ Passive resource == \textit{Resource} \((R_i)\):
    • job may require resources in addition to processor
    • reusable resources are not consumed
Processors

- **Processors execute tasks**
  - Predictability is important for real-time systems
  - Fancy cache systems, instruction executions, virtual memory management, etc might not be desirable
  - One of the major power hungry devices in real-time systems

- **Homogeneous vs. heterogeneous**
  - Homogeneous:
    - all processors are of the same type
    - Execution times are the same
  - Heterogeneous:
    - Different types of processors
    - Execution times may be different
The Communication Network

- **Shared media**
  - Multiple processors share the same communication media, such as bus
  - one message at a time

- **Switched media (point-to-point)**
  - Allows communications go straightly from source to destination
  - Circuit switch/packet switch
  - Channel establishment/message routing
Functional Parameters

• **Preemptivity**
  ‣ preemption: suspend job then dispatch different job to processor, cost includes context switch overhead
  ‣ non-preemptable task - must be run from start to completion

• **Criticalness** - positive integer indicating the relative importance of a job (*useful during overload*)
  ‣ Priority, weight
  ‣ Importance

• **Optional Executions** - jobs or portions of jobs may be declared optional (*useful during overload*)

• **Laxity** - Laxity type => hard or soft timing constraints, supplemented by a usefulness function (*useful during overload*)
Functional Parameters

- Laxity - Laxity type => hard or soft timing constraints, supplemented by a usefulness function (useful during overload)
Resource Parameters of Jobs and Parameters of Resources

- **Preemptivity of resources**
  - Nonpreemptable if each unit of the resource is constrained to be used serially
  - Preemptable if each unit of the resource can be used in an interleaved fashion
  - Resource is nonpreemptable does not mean jobs using the resource is nonpreemptable on other resources.
Resource Parameters of Jobs and Model of Real-Time Systems

scheduling and resource-access control

processors

resources
Resource Parameters of Jobs and Scheduling Hierarchy

- **Scheduler**: assign jobs to processors, or processors to jobs
- **Schedule**: an assignment of all jobs produced by the scheduler
- **Valid schedule** (assume no jobs run in parallel on more than one processors)
  - Every processor is assigned to **at most one** job at any time
  - Every job is assigned **at most one** processor at any time
  - No job is scheduled before its release time
  - The total amount of processor time assigned to every job is equal to its maximum or actual execution time
  - All the precedence and resource usage constraints are satisfied
Resource Parameters of Jobs and Scheduling Hierarchy

• Feasibility
  - A valid schedule is a feasible schedule if all jobs complete within their timing constraints

• Optimality of scheduling algorithm
  - For a given set of jobs, if the scheduling algorithm (scheduler) always produces a feasible schedule assuming such feasible schedule exists, the scheduling algorithm (or the scheduler) is optimal.

• Performance measure
  - Tardiness
  - Lateness
  - Response time
  - Completion time jitter
  - Makespan
  - …
Meeting Application Requirements

• Scheduling
  ‣ Allocate computing resources (processor, link capacity, memory, storage space, etc.) and time to meet the requirements

• Two research problems
  ‣ Given a real-time system, develop a scheduling algorithm (policy) that can meet the requirement (and also optimize system resource usage)

  ‣ Given a scheduling algorithm (policy), what types of real-time systems can be satisfied by the policy?
Review: Process Scheduling

• Problem:
  ‣ single resource (CPU)
  ‣ many users (process)

• Common policies:
  ‣ First-In First-Out (FIFO)
    • runs to completion
  ‣ Round Robin (RR)
    • runs for a specified time quantum
  ‣ Time-Sharing (TS)
    • Multilevel feedback queues
Typical Scheduler Goals

- **Interactive**
  - shells, GUI
  - spend most of their time waiting for I/O
  - minimize perceived delay

- **Batch**
  - compiles, computations
  - optimize throughput

- **Real-time**
  - require predictable behavior
  - may require guarantees on throughput, latency or delay
3: REALTIME SCHEDULING
What’s Important in Real-Time

Metrics for real-time systems differ from that for time-sharing systems.

<table>
<thead>
<tr>
<th></th>
<th>Time-Sharing Systems</th>
<th>Real-Time Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Capacity</strong></td>
<td>High throughput</td>
<td>Schedulability</td>
</tr>
<tr>
<td><strong>Responsiveness</strong></td>
<td>Fast average response</td>
<td>Ensured worst-case response</td>
</tr>
<tr>
<td><strong>Overload</strong></td>
<td>Fairness</td>
<td>Stability</td>
</tr>
</tbody>
</table>

- **schedulability** is the ability of tasks to meet all hard deadlines
- **latency** is the worst-case system response time to events
- **stability** in overload means the system meets critical deadlines even if all deadlines cannot be met
Real-Time Scheduling

- Determines the order of real-time task executions
Scheduling

• **Scheduling analysis**
  - Study the properties of scheduling policies
    - Can a task set meet the timing requirement with certain given scheduling policies?

• **Scheduling synthesis**
  - For a given task set, what scheduling algorithm produce a feasible schedule?
Schedulability

- Property indicating whether a real-time system (a set of real-time tasks) can meet their deadlines
Feasible Schedule vs Optimal Schedule

- **Feasible schedule**
  - meet each task’s deadline without violating any constraints

- **Optimal schedule**
  - Optimality criteria: an optimality criterion assesses the relative merits of competing feasible schedules
  - Optimal schedule may not be unique
Scheduling Policy Taxonomy

- Uniprocessor/Multiprocessor
- Static/dynamic (offline/on-line)
- Periodic/aperiodic
- Priority driven/Non-priority driven
- Preemptive/Non-preemptive
- Hard real-time/soft real-time
- Homogeneous/heterogeneous multiprocessors
- Tasks with precedence constraints/independent tasks
- Scheduling with other constraints (power, fault tolerance, synchronization, resource constraints, and other quality-of-service constraints)
Classification of Scheduling Algorithms

- All scheduling algorithms
  - static scheduling (or offline, or clock driven)
    - static-priority scheduling
  - dynamic scheduling (or online, or priority driven)
    - dynamic-priority scheduling
Scheduling strategies

• **Non pre-emptive scheduling**
  - Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).

• **Pre-emptive scheduling**
  - The execution of an executing processes may be stopped if a higher priority process requires service.

• **Scheduling algorithms**
  - Round-robin;
  - Rate monotonic;
  - Shortest deadline first;
  - Etc.
Preemptive vs Nonpreemptive

• In general, nonpreemptive scheduling is not better than preemptive scheduling

• Question: do we have a rule to determine from given parameters which one is better? NO

• Special case:
  ▸ when jobs have the same release time, preemptive is better if overhead is ignored

• In a multi-processor system,
  ▸ the minimum makespan achievable by an optimal preemptive algorithm is shorter than the makespan achievable by an optimal nonpreemptive algorithm.

• Coffman and Garey:
  ▸ for two processors, the minimum makespan achievable by nonpreemptive algorithms is never more than $4/3$ times the minimum makespan achievable by preemptive algorithms when the cost of preemption is negligible.
Dynamic vs Static Systems

• **Multi-processors**
  - Statically partition tasks to processors
  - Runtime decide the processors for different tasks

• **More about dynamic systems**
  - Dynamic systems may be more responsive on the average
  - Worst case real-time performance poorer than static systems
  - No reliable techniques to validate the timing constraints
Different Scheduling Strategies

• Clock Driven

• Priority Driven
  ‣ EDF, LST, LRT,
  ‣ RM
Clock-Driven Scheduling Schemes

• **Clock-driven (time-driven) scheduling**
  - All parameters of hard real-time jobs are fixed and known
  - Scheduling is done off-line and stored for run-time use
  - Minimal runtime scheduling overhead
Clock-Driven Scheduling Schemes

- Weighted round-robin approach (job with weight $wt$ gets $wt$ units of execution during each round which is the sum of all weights). All jobs release at time 0 and execution is 1

\[
\begin{align*}
J_{1,1} & \quad J_{1,2} \\
J_{2,1} & \quad J_{2,2}
\end{align*}
\]

\[P_1 \quad J_{1,1} \& J_{2,1}\]

\[P_2 \quad \quad \quad \quad J_{1,2} \& J_{2,2}\]

\[P_1 \quad J_{1,1} \quad \quad J_{2,1}\]

\[P_2 \quad \quad \quad \quad J_{1,2} \quad \quad J_{2,2}\]

**FIGURE 4-1** Example illustrating round-robin scheduling of precedence-constrained jobs.
Priority-Driven Scheduling Schemes

• **Priority-driven (event-driven)**
  - At any decision time, jobs with highest priorities are scheduled
    - FIFO/LIFO; SETF/LETF (shortest/longest execution time first)
    - Is round-robin scheduling a priority-drive scheduling?
  - Never intentionally leave any resource idle
  - Greedy, makes locally optimal decisions
  - Greedy does not always pay

• **Example:**
  - Release time: all at 0, except job 5 at time 4
  - Execution time (shown next to the job)
  - Priority (J1, J2, J3, … J8)
Priority Driven Scheduling

- Static-priority scheduling
- Dynamic-priority scheduling
Priority-Driven Scheduling Schemes

\( J_1, 3 \)
\( J_2, 1 \)
\( J_5, 2 \)
\( J_7, 4 \)
\( J_3, 2 \)
\( J_6, 4 \)
\( J_8, 1 \)

\( J_1, J_2, \ldots, J_8 \)

\( J_1, J_4, J_7, J_6 \)

\( J_2, J_3, J_5, J_8 \)

\( P_1 \)
\( P_2 \)

FIGURE 4–2 Example of priority-driven scheduling. (a) Preemptive (b) Nonpreemptive.
Two most important priority-driven preemptive scheduling schemes

• Earliest Deadline First (EDF)

• Rate Monotonic Scheduling (RMS)
Earliest Deadline First (EDF) Scheduling (Cont’d)

- Can fully utilize the processor

- Dynamic priority assignment
  - Need to recompute and assign the priority
EDF (Earliest Deadline First)

- Optimal dynamic priority scheduling
- A task with a shorter deadline has a higher priority
- Executes a job with the earliest deadline
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline

```
T_1(4,1)  T_2(5,2)  T_3(7,2)
```

- Graphical representation of job execution timeline.
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline

T₁(4,1)  
T₂(5,2)  
T₃(7,2)
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline

\[ T_1(4,1) \]
\[ T_2(5,2) \]
\[ T_3(7,2) \]
EDF (Earliest Deadline First)

- Optimal scheduling algorithm
  - if there is a schedule for a set of real-time tasks, EDF can schedule it.
  - A given task set is feasible by EDF iff $U \leq 1$.

![Diagram of EDF scheduling]

- $T_1(4,1)$
- $T_2(5,2)$
- $T_3(7,2)$

Timeline: 0-5-10-15
**Demand Bound Function**: $dbf(t)$
- the maximum processor demand by workload over any interval of length $t$
EDF - Schedulability Analysis

• Real-time system is schedulable under EDF if and only if $dbf(t) \leq t$ for all interval $t$

Baruah et al.

• Demand Bound Function : $dbf(t)$
  ▶ the maximum processor demand by workload over any interval of length $t$
EDF – Overload Conditions

- Domino effect during overload conditions
  - Example: $T_1(4,3), T_2(5,3), T_3(6,3), T_4(7,3)$

![Diagram showing the domino effect and better schedules]

Deadline Miss!

Better schedules:
Optimality of the EDF and LST Algorithm

**Theorem:** When preemption is allowed and jobs do not contend for resources, the EDF algorithm is optimal.

\[ I_1 \]

\[ J_i \]

\[ I_2 \]

\[ J_k \]

\[ r_k \]

\[ d_k \]

\[ d_i \]

**FIGURE 4-4** Transformation of a non-EDF schedule into an EDF schedule.
Nonoptimality of the EDF and LST Algorithm

Non-preemptive EDF and LST scheduling algorithms are not optimal (uni-processor).

Example: J1 (0, 3, 10), J2 (2, 6, 14) and J3 (4, 4, 12)
Nonoptimality of the EDF and LST Algorithm

Non-preemptive EDF and LST scheduling algorithms are not optimal (multi-processors).

Example: J1 (0, 1, 1), J2 (0, 1, 2) and J3 (0, 5, 5)
Conclusion:

- Nonpreemptive EDF and LST algorithm are not optimal,
- No nonpreemptive priority-driven algorithm is optimal when jobs have arbitrary release times, execution times, and deadlines.
Anomalous Behavior of Priority-Driven Systems

Assume 4 jobs with timing info given in the table with J1 highest priority and two identical processors. Further assume jobs do not migrate between processors.

<table>
<thead>
<tr>
<th></th>
<th>$r_i$</th>
<th>$d_i$</th>
<th>$[e_i^-, e_i^+]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>$J_2$</td>
<td>0</td>
<td>10</td>
<td>[2, 6]</td>
</tr>
<tr>
<td>$J_3$</td>
<td>4</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>$J_4$</td>
<td>0</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>
Anomalous Behavior of Priority-Driven Systems

(a) P1: J1, J3
P2: J2, J4

(b) P1: J1
P2: J2, J4, J3, J4

(c) P1: J1
P2: J2, J4, J3, J4

(d) P1: J1, J3
P2: J2, J4
Anomalous Behavior of Priority-Driven Systems

Completion time of a set of nonpreemptive jobs with identical release times can be later when more processors are used to execute them, and when they have shorter execution times and fewer dependencies.

Anomalies can occur with single processor it can occur with single processor and with preemption if jobs have arbitrary release time and share resources.
Two most important priority-driven preemptive scheduling schemes

- Earliest Deadline First (EDF)
- Rate Monotonic Scheduling (RMS)
Rate Monotonic Scheduling

- Rate Monotonic Analysis (RMA)
Model

• All the tasks run on the single processor
• No data dependency between tasks
• No context switch penalty
• Priority driven preemptive scheduling
Model (Cont’d)

• **Tasks**
  - **Period (** $T_i$ **)**
    - Constant interval between the arrivals of two task consecutive jobs
      - Job
  - **Deadline (** $D_i$ **)**
    - Assume $D_i = T_i$
  - **Worst Case Execution Time (** $C_i$ **)**
  - **Initial arrival time (** $R_i$ **)**
    - Assume $R_0 = R_1 = \ldots = R_n = 0$ (The first job of all the tasks arrive at the same time)
RM (Rate Monotonic)

- Optimal static-priority scheduling
- It assigns priority according to period
- A task with a shorter period has a higher priority
- Executes a job with the shortest period

\[ T_1(4,1) \]
\[ T_2(5,2) \]
\[ T_3(7,2) \]
RM (Rate Monotonic)

- Assign priorities according to their periods (rates)
  - The longer the period, the lower the priority

- Executes a job with the shortest period

![Diagram of RM (Rate Monotonic) scheduling]

- $T_1(4,1)$
- $T_2(5,2)$
- $T_3(7,2)$
RM (Rate Monotonic)

- Executes a job with the shortest period

![Diagram showing RM algorithm execution with deadline miss]

T₁(4,1)  
T₂(5,2)  
T₃(7,2)  

Deadline Miss!
Response Time

- **Response time**
  - Duration from released time to finish time

![Diagram with T1(4,1), T2(5,2), T3(10,2)]
**Response Time**

- **Response time**
  - Duration from released time to finish time

![Diagram showing response time example](image)

- $T_1(4,1)$
- $T_2(5,2)$
- $T_3(10,2)$
More about RMS (Cont’d)

- RMS is optimal
  - Theorem: If a feasible fixed priority assignment exists for a task set, the RM priority assignment is also feasible for that task set.
RM – Utilization Bound

- Real-time system is schedulable under RM if for m jobs
  \[ \sum U_i \leq m \left(2^{1/m} - 1\right) \]

Liu & Layland,

RM – Utilization Bound

- Real-time system is schedulable under RM if
  \[ \sum U_i \leq m (2^{1/m} - 1) \]

- Example: \( T_1(4,1) \), \( T_2(5,1) \), \( T_3(10,1) \),

\[
\sum U_i = \frac{1}{4} + \frac{1}{5} + \frac{1}{10} \\
= 0.55 \\
3 \left( 2^{1/3} - 1 \right) \approx 0.78
\]

Thus, \( \{T_1, T_2, T_3\} \) is schedulable under RM.
Utilization Bound

- **Utilization**
  \[ U = \sum_{i} \frac{C_i}{T_i} \]

- **Theorem:** For a set of m tasks with fixed priority assignment, the least upper bound to processor utilization is
  \[ U = m(2^{1/m} - 1) \]

- E.g. m=2, U=0.83; m=3, U=0.78; for large m, U→ln2=0.69
• Real-time system is schedulable under RM if
\[ \sum U_i \leq m \left( 2^{1/m} - 1 \right) \]

**RM Utilization Bounds**

![Graph showing RM Utilization Bounds](image)
More about RMS

• Critical instant
  ‣ At which if a task arrives, it will have the longest response time

• Theorem:
  ‣ A critical instant for a task occurs whenever the task is released simultaneously with all higher priority tasks.
Utilization Bound (Cont’d)

- **A sufficient condition**
  - The task set is feasible as long as its utilization is no larger than the utilization bound
  - Many feasible task can have higher utilization

- **Many feasible fixed-priority task sets cannot 100% utilize the processor**
Improving Utilization Bound Using Linear Programming

- Exact timing analysis for fixed-priority scheduling
- Using a set of bounds
  - Define:
    \[ U_i = \sum_{j=1}^{i} \frac{C_j}{T_j} \]
  - \( B_i \) is called the utilization bound for \( \tau_i \)
    - \( \tau_i \) will meet its deadline as long as \( U_i \) is no more than \( B_i \)
RM vs. EDF

• **Rate Monotonic**
  - Simpler implementation, even in systems without explicit support for timing constraints (periods, deadlines)
  - Predictability for the highest priority tasks

• **EDF**
  - Full processor utilization
  - Misbehavior during overload conditions

• **For more details:** Buttazzo, “Rate monotonic vs. EDF: Judgement Day”, EMSOFT 2003.
Summary

- Scheduling techniques can be classified along many dimensions
- RMS and EDF are two most important uniprocessor, priority-based, preemptive scheduling policies
- RMS (DMS)
  - The optimal fixed priority assignment
  - Feasibility analysis
    - Exact timing analysis
    - Utilization bound
- EDF
  - The optimal scheduling algorithm
  - Feasibility analysis
    - Utilization bound
    - Work demand analysis