Real-Time Kernels and Systems

Gabriel Rovesti

Academic Year 2024-2025

Preface

These notes provide a comprehensive coverage of the material presented in the Real-Time Kernels and Systems course taught by Prof. Tullio Vardanega at the University of Padua. They offer a detailed exploration of the theoretical foundations and practical aspects of real-time systems, covering topics from basic scheduling algorithms to advanced topics like multicore scheduling and mixed-criticality systems.

The notes are organized following the structure of the course, with each chapter focusing on a specific topic. The material includes formal definitions, theorems with proofs where appropriate, illustrative examples, and detailed explanations of the underlying concepts to provide a deep understanding of real-time systems theory and practice.

Contents

Preface iii

1 Introduction to Real-Time Systems 1
- 1.1 Definition and Fundamental Concepts . . . 1
  - 1.1.1 Classes of Real-Time Systems . . . 1
  - 1.1.2 Essential Properties of Real-Time Systems . . . 2
- 1.2 Real-Time System Architecture . . . 2
  - 1.2.1 Hardware Architecture . . . 2
  - 1.2.2 Software Architecture . . . 3
  - 1.2.3 Resource Model . . . 3
- 1.3 Real-Time Workload Model . . . 3
  - 1.3.1 Tasks and Jobs . . . 3
  - 1.3.2 Task Characteristics . . . 4
  - 1.3.3 Task Classification . . . 4
  - 1.3.4 Task Constraints . . . 4
- 1.4 Real-Time Scheduling . . . 5
  - 1.4.1 The Scheduling Problem . . . 5
  - 1.4.2 Scheduling Taxonomy . . . 5
  - 1.4.3 Feasibility and Schedulability . . . 6
- 1.5 Real-Time Analysis Techniques . . . 6
  - 1.5.1 Response Time Analysis . . . 6
  - 1.5.2 Utilization-Based Analysis . . . 7
  - 1.5.3 Worst-Case Execution Time Analysis . . . 7
  - 1.5.4 Demand Bound Function . . . 8

2 Scheduling Basics 9
- 2.1 Clock-Driven Scheduling . . . 9
  - 2.1.1 Principles of Clock-Driven Scheduling . . . 9
  - 2.1.2 Cyclic Executive Model . . . 9
  - 2.1.3 Cyclic Executive Construction . . . 10
  - 2.1.4 Advantages and Disadvantages . . . 11
  - 2.1.5 Design Issues in Cyclic Executives . . . 11
- 2.2 Round-Robin Scheduling . . . 12

2.2.1 Principles of Round-Robin Scheduling 12
2.2.2 Weighted Round-Robin 12
2.2.3 Properties and Limitations 12
2.3 Priority-Driven Scheduling 13
2.3.1 Principles of Priority-Driven Scheduling 13
2.3.2 Static vs. Dynamic Priorities 13
2.3.3 Preemptive vs. Non-preemptive 14
2.3.4 Dispatching Points 14
2.4 Optimality in Uniprocessor Scheduling 14
2.4.1 Earliest Deadline First (EDF) 14
2.4.2 Least Laxity First (LLF) 15
2.4.3 Critical Instant and Busy Period 16
2.5 Predictability and Sustainability 16
2.5.1 Predictability 16
2.5.2 Sustainability 16
2.6 Scheduling Anomalies 17
2.6.1 Scheduling Anomalies in Single-Processor Systems . . 17
2.6.2 Scheduling Anomalies in Self-Suspension 17

3 Fixed-Priority Scheduling 19
3.1 Workload Model 19
3.1.1 Basic Assumptions 19
3.2 Rate Monotonic Scheduling (RMS) 19
3.2.1 Priority Assignment 19
3.2.2 Optimality of RMS 20
3.2.3 Schedulability Tests 20
3.3 Deadline Monotonic Scheduling (DMS) 21
3.3.1 Priority Assignment 21
3.3.2 Optimality of DMS 22
3.4 Response Time Analysis 22
3.4.1 Basic Concept 22
3.4.2 Response Time Calculation 22
3.4.3 Iterative Solution 22
3.5 Task Interactions 24
3.5.1 Hard and Soft Tasks 24
3.5.2 Strategies for Coexistence 24
3.6 Handling Aperiodic Tasks 24
3.6.1 Background Execution 25
3.6.2 Slack Stealing 25
3.6.3 Server Mechanisms 25
3.7 Considerations for Practical Implementation 26
3.7.1 Priority Levels 26
3.7.2 Release Jitter 27

4 Task Interactions and Blocking Effects **29**
4.1 Task Cooperation and Communication . . . 29
4.1.1 Communication Methods . . . 29
4.1.2 Challenges of Task Interaction . . . 29
4.2 Preemption and Critical Sections . . . 30
4.2.1 Atomic Operations . . . 30
4.2.2 Critical Sections . . . 30
4.2.3 Inhibiting Preemption . . . 30
4.2.4 Priority Inversion . . . 31
4.3 Self-Suspension . . . 32
4.3.1 Effects on Analysis . . . 32
4.3.2 Blocking Due to Self-Suspension . . . 32
4.4 Resource Access Protocols . . . 32
4.4.1 Priority Inheritance Protocol (PIP) . . . 33
4.4.2 Priority Ceiling Protocol (PCP) . . . 34
4.4.3 Stack Resource Policy (SRP) . . . 35
4.4.4 Ceiling Priority Protocol (CPP) . . . 35
4.5 Computing Blocking Times . . . 36
4.5.1 Direct, Inheritance, and Avoidance Blocking . . . 36
4.5.2 Comparison of Resource Access Protocols . . . 37

5 Further Model Extensions **39**
5.1 Cooperative Scheduling . . . 39
5.1.1 Deferred Preemption . . . 39
5.1.2 Fixed vs. Floating Non-Preemptive Regions . . . 39
5.1.3 Response Time Analysis with Deferred Preemption . . . 40
5.2 Release Jitter . . . 41
5.2.1 Sources of Jitter . . . 41
5.2.2 Jitter in Precedence Constraints . . . 41
5.2.3 Effects on Analysis . . . 42
5.3 Arbitrary Deadlines . . . 42
5.3.1 The Busy Period Analysis . . . 42
5.4 Offsets . . . 44
5.4.1 Benefits of Offsets . . . 44
5.4.2 Analysis with Static Offsets . . . 44
5.5 Transactions . . . 45
5.5.1 Task Concatenations . . . 45
5.5.2 End-to-End Analysis . . . 45
5.6 Worst-Case Execution Time Analysis . . . 46
5.6.1 The WCET Challenge . . . 46
5.6.2 WCET Analysis Techniques . . . 46
5.6.3 Challenges in WCET Analysis . . . 48
5.6.4 WCET Tools and Techniques . . . 48
5.7 Introduction to Real-Time Scheduling on Multicore Processors 48

5.7.1 Motivation for Multicore 48
5.7.2 Multicore Architectures 49
5.7.3 Resource Sharing in Multicore 49
5.7.4 Multicore Scheduling Paradigms 49
5.7.5 Challenges in Multicore Scheduling 50
5.7.6 Multicore Schedulability Analysis 50
5.8 Seeking the Lost Optimality 51
5.8.1 The Optimality Problem 51
5.8.2 Proportionate Fairness (P-Fair) 51
5.8.3 DP-Fair and LLREF 51
5.8.4 RUN: Reduction to Uniprocessor 52
5.8.5 QPS and Other Approaches 52
5.9 Sharing Resources Across Processors 52
5.9.1 The Multiprocessor Resource Sharing Problem 52
5.9.2 Suspension-Based Protocols 53
5.9.3 Spin-Based Protocols 53
5.9.4 Blocking Analysis for Multiprocessor Protocols 54
5.9.5 Priority Assignments and Blocking 54
5.10 Mixed-Criticality Systems 54
5.10.1 Motivation and Background 54
5.10.2 Vestal's Model 54
5.10.3 Scheduling in Mixed-Criticality Systems 55
5.10.4 Adaptive Mixed-Criticality (AMC) 56
5.10.5 Multicore Mixed-Criticality 56
5.10.6 Practical Considerations 57
5.11 Conclusion 57
.1 Glossary of Terms 58
.2 Bibliography 58

Chapter 1

Introduction to Real-Time Systems

1.1 Definition and Fundamental Concepts

A real-time system is one where correctness depends not only on the logical results of computation but also on the time at which these results are produced. These systems must satisfy explicit (finite) constraints on the timing of events to be considered correct.

1.1.1 Classes of Real-Time Systems

Real-time systems are classified according to the consequences of missing deadlines:

- **Hard real-time systems**: Missing a deadline is considered a system failure and can lead to catastrophic consequences. Examples include aircraft flight control systems, automotive braking systems, and nuclear power plant control systems.
- **Soft real-time systems**: Missing a deadline diminishes the system's quality of service but does not constitute a system failure. Examples include multimedia streaming and non-critical monitoring systems.
- **Firm real-time systems**: Missing a deadline renders the result useless, but does not cause system failure. Examples include video frame processing where a late frame is discarded.

The boundary between these classifications is not always clear-cut and many real systems include components with different timing requirements. For instance, a single automotive system might include hard real-time braking control, firm real-time display updates, and soft real-time climate control.

1.1.2 Essential Properties of Real-Time Systems

- **Timeliness**: Results must be delivered within specified time constraints.
- **Predictability**: The system behavior must be deterministic and analyzable, allowing for verification of timing constraints.
- **Efficiency**: The system must effectively use available resources to meet timing constraints.
- **Robustness**: The system must handle exceptional conditions without catastrophic consequences.
- **Fault Tolerance**: The system must continue to operate, possibly in a degraded mode, despite failures.
- **Reactiveness**: The system must respond to external events promptly and consistently.

1.2 Real-Time System Architecture

1.2.1 Hardware Architecture

Real-time systems are built on hardware platforms that include:

- **Processors**: From simple microcontrollers to complex multicore systems. Real-time processors often have predictable timing behavior, avoiding features like speculative execution that can introduce timing unpredictability.
- **Memory**: RAM, ROM, flash memory, etc. Memory access times must be predictable for real-time performance.
- **I/O Subsystems**: Sensors, actuators, and communication interfaces. These often include mechanisms to support predictable timing, such as dedicated DMA channels and interrupt priorities.
- **Specialized Hardware**: Timers, watchdogs, and hardware accelerators that support time-critical operations.
- **Inter-processor Communication**: In distributed systems, communication infrastructure (buses, networks) with predictable timing characteristics.

1.2.2 Software Architecture

The software structure of a real-time system typically consists of:

- **Real-Time Operating System (RTOS)**: Provides services for task management, scheduling, synchronization, and communication with predictable timing behavior.
- **Application Software**: Implements the specific functionality of the system.
- **Middleware**: Provides higher-level abstractions and services, often for distributed real-time systems.
- **Device Drivers**: Interface with hardware components, implementing device-specific operations with timing guarantees.

1.2.3 Resource Model

Resources in a real-time system can be categorized as:

- **Active resources**: Processors that execute tasks (e.g., CPU cores).
- **Passive resources**: Resources that tasks may need during execution (e.g., data structures, I/O devices, shared memory).

The appropriate management of both active and passive resources is crucial for the predictable execution of real-time tasks. Resource sharing protocols are essential to ensure predictable behavior, especially in preemptive systems.

1.3 Real-Time Workload Model

1.3.1 Tasks and Jobs

In real-time systems, the workload is typically modeled as a set of tasks:

- **Task (τ_i)**: A sequential program that executes periodically or in response to events. A task is characterized by its timing parameters and resource requirements.
- **Job ($J_{i,j}$)**: A single execution instance of a task. The j-th job of task τ_i is denoted as $J_{i,j}$.

1.3.2 Task Characteristics

Tasks in real-time systems are characterized by:

- **Release time (r_i)**: The time at which a task becomes ready for execution.
- **Computation time (C_i)**: The time required to complete the task's execution without interruption. For analysis purposes, this is often the Worst-Case Execution Time (WCET).
- **Deadline (D_i)**: The time by which the task must complete its execution, relative to its release time.
- **Period (T_i)**: For periodic tasks, the time interval between consecutive releases.
- **Priority (P_i)**: The importance of the task relative to others (used in priority-based scheduling).
- **Response time (R_i)**: The time between a task's release and its completion.
- **Utilization (U_i)**: The fraction of processor time required by the task, given by $U_i = C_i/T_i$ for periodic tasks.

1.3.3 Task Classification

Tasks can be classified based on their activation patterns:

- **Periodic**: Tasks that are released at regular intervals. A periodic task τ_i is characterized by (C_i, T_i, D_i) where T_i is the period.
- **Sporadic**: Tasks that are released irregularly but with a minimum inter-arrival time between consecutive releases. A sporadic task τ_i is characterized by (C_i, T_i, D_i) where T_i is the minimum inter-arrival time.
- **Aperiodic**: Tasks that are released at irregular intervals with no minimum inter-arrival time guarantee. These are typically handled through special mechanisms such as servers.

1.3.4 Task Constraints

Several types of deadlines are considered in real-time scheduling:

- **Implicit deadlines**: The deadline equals the period ($D_i = T_i$).

- **Constrained deadlines**: The deadline is less than or equal to the period ($D_i \leq T_i$).
- **Arbitrary deadlines**: No restriction on the relationship between the deadline and the period (D_i can be less than, equal to, or greater than T_i).

Tasks may also have dependencies:

- **Independent tasks**: Tasks that do not interact with each other.
- **Dependent tasks**: Tasks with precedence constraints (one task must complete before another can start) or resource sharing requirements.

1.4 Real-Time Scheduling

1.4.1 The Scheduling Problem

The real-time scheduling problem involves determining when and how to execute tasks to meet their timing constraints, given the system's resources and the tasks' characteristics. A schedule specifies the allocation of resources to tasks over time.

1.4.2 Scheduling Taxonomy

Scheduling algorithms can be classified along several dimensions:

- **Static vs. Dynamic**:
  - **Static**: Scheduling decisions are made offline at design time, resulting in a fixed schedule.
  - **Dynamic**: Scheduling decisions are made online during system execution, based on the current state.
- **Preemptive vs. Non-preemptive**:
  - **Preemptive**: The currently executing task can be interrupted by a higher-priority task.
  - **Non-preemptive**: Once a task starts execution, it runs to completion.
- **Priority-driven vs. Time-driven**:
  - **Priority-driven**: Scheduling is based on task priorities.
  - **Time-driven**: Scheduling is based on predefined time slots.
- **Online vs. Offline**:

  - **Online**: Scheduling decisions are made without knowledge of future task arrivals.
  - **Offline**: The complete task set is known in advance.

- **Optimal vs. Heuristic**:
  - **Optimal**: Guaranteed to find a feasible schedule if one exists.
  - **Heuristic**: Aims to find a good, but not necessarily optimal, solution.

1.4.3 Feasibility and Schedulability

- **Feasible task set**: A task set for which there exists at least one schedule that meets all deadlines.
- **Schedulable task set**: A task set that can be scheduled by a specific scheduling algorithm without missing deadlines.
- **Optimal scheduling algorithm**: An algorithm that can schedule any feasible task set.
- **Schedulability test**: A method to determine whether a given task set is schedulable under a specific scheduling algorithm.
  - **Exact test**: A test that gives a necessary and sufficient condition for schedulability.
  - **Sufficient test**: A test that guarantees schedulability if passed, but may reject some schedulable task sets.
  - **Necessary test**: A test that guarantees unschedulability if failed, but may accept some unschedulable task sets.

1.5 Real-Time Analysis Techniques

1.5.1 Response Time Analysis

Response Time Analysis (RTA) is a technique to determine the worst-case response time of tasks in a system, which is the time from when a task is released until it completes execution. For a task τ_i under fixed-priority scheduling:

$$R_i = C_i + I_i \tag{1.1}$$

where C_i is the worst-case execution time of τ_i and I_i is the interference from higher-priority tasks:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1.2}$$

where $hp(i)$ is the set of tasks with higher priority than τ_i.
The equation for R_i is solved iteratively:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{1.3}$$

Starting with $R_i^0 = C_i$, the iteration continues until either:

- $R_i^{n+1} = R_i^n$ (a fixed point is reached), or
- $R_i^{n+1} > D_i$ (the deadline is exceeded)

1.5.2 Utilization-Based Analysis

Utilization-based analysis examines the processor utilization as a test for schedulability. For a task set with n tasks:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1.4}$$

For Earliest Deadline First (EDF) scheduling with implicit deadlines, the schedulability condition is:

$$U \leq 1 \tag{1.5}$$

For Rate Monotonic Scheduling (RMS) with implicit deadlines, the schedulability condition is:

$$U \leq n \cdot (2^{1/n} - 1) \tag{1.6}$$

As $n \to \infty$, this bound approaches $\ln 2 \approx 0.693$.

1.5.3 Worst-Case Execution Time Analysis

Worst-Case Execution Time (WCET) analysis determines the maximum time a task can take to execute on a given hardware platform. The WCET must consider:

- All possible execution paths through the program.
- Hardware effects such as caching, pipelining, and branch prediction.
- Input data dependencies that affect execution time.

WCET analysis approaches include:

- **Static analysis**: Analyzing the program structure without execution.
- **Measurement-based analysis**: Measuring execution times of program segments.
- **Hybrid approaches**: Combining static analysis and measurements.

1.5.4 Demand Bound Function

The demand bound function (dbf) for a task set represents the maximum cumulative execution time demanded by all tasks with deadlines in an interval of length t:

$$dbf(t) = \sum_{i=1}^{n} \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \cdot C_i \tag{1.7}$$

For EDF, the schedulability condition is:

$$\forall t > 0 : dbf(t) \leq t \tag{1.8}$$

Chapter 2

Scheduling Basics

2.1 Clock-Driven Scheduling

2.1.1 Principles of Clock-Driven Scheduling

Clock-driven (time-driven) scheduling makes scheduling decisions at fixed time instants determined by a clock:

- Scheduling decisions are made at design time and activated at fixed time instants during execution via clock interrupts.
- The scheduler dispatches the job due in the current time interval and then suspends itself until the next scheduled time.
- Jobs must complete within their assigned time intervals.
- All scheduling parameters must be known in advance.
- The schedule, computed offline, is static and fixed.

Clock-driven scheduling is essentially a form of static, non-preemptive scheduling where the execution sequence is determined completely in advance.

2.1.2 Cyclic Executive Model

A cyclic executive is a specific implementation of clock-driven scheduling:

- Time is divided into fixed-size frames called minor cycles.
- Each minor cycle contains a sequence of jobs to be executed.
- A complete sequence of minor cycles forms a major cycle that repeats periodically.
- The major cycle's length is typically the hyperperiod of the tasks (the least common multiple of all task periods).

For a cyclic executive to be feasible, several constraints must be satisfied:

1. **Constraint 1**: Every job must complete within a single frame.

$$f \geq \max_{i=1,\ldots,n} (C_i) \tag{2.1}$$

where f is the frame size and C_i is the worst-case execution time of task τ_i.

2. **Constraint 2**: The frame size must be an integer divisor of the hyperperiod.

$$H = N \cdot f \text{ where } N \in \mathbb{N} \tag{2.2}$$

where H is the hyperperiod.

3. **Constraint 3**: There must be one full frame between a job's release time and its deadline.

$$2f - \gcd(p_i, f) \leq D_i \text{ for every task } \tau_i \tag{2.3}$$

where p_i is the period of task τ_i, D_i is its deadline, and gcd is the greatest common divisor.

2.1.3 Cyclic Executive Construction

The construction of a cyclic executive schedule involves three main steps:

1. Determine the frame size f that satisfies all constraints.

2. Break down (slice) jobs that are too large to fit within a single frame.

3. Assign jobs and slices to minor cycles, ensuring that:

   - All timing constraints are met.
   - The total execution time in each frame does not exceed the frame size.
   - Precedence constraints between slices are respected.

The schedule is typically represented as a table specifying which job or slice to execute at each minor cycle.

2.1.4 Advantages and Disadvantages

Advantages:

- Simple design and implementation.
- Predictable behavior with minimal runtime overhead.
- No need for complex task synchronization mechanisms.
- Straightforward to verify timing correctness.

Disadvantages:

- Inflexible — difficult to modify the schedule.
- Construction of the schedule is an NP-hard problem.
- Limited support for sporadic and aperiodic tasks.
- Inefficient use of CPU time when tasks' execution times vary.
- Tasks must be sliced if their execution times exceed the frame size.
- Less robust to changes in task parameters.

2.1.5 Design Issues in Cyclic Executives

Several design issues arise in cyclic executives:

- **Slack Stealing**: Allocating unused CPU time to aperiodic jobs. This can be achieved by processing aperiodic jobs at the beginning of each minor cycle, up to a pre-computed maximum.
- **Overrun Handling**: When a job executes past its due time, several strategies are possible:
  - Halt the job at the end of its allocated time.
  - Allow it to complete its critical actions before halting.
  - Delay the start of the next minor cycle (if timing constraints permit).
- **Mode Changes**: When the system needs to reconfigure its functions and workload parameters, transitions between operation modes must be managed to ensure timing correctness.

2.2 Round-Robin Scheduling

2.2.1 Principles of Round-Robin Scheduling

Round-robin scheduling is a simple dynamic scheduling algorithm with the following characteristics:

- All ready jobs are placed in a FIFO (First-In-First-Out) queue.
- CPU time is allocated in fixed time slices or quanta.
- The job at the head of the queue is dispatched to execution for one time slice.
- If the job does not complete by the end of its time slice, it is preempted and placed at the tail of the queue.
- The process repeats, giving each job in the queue one time slice per round (full traversal of the queue).

Round-robin scheduling divides the processor time equally among all ready jobs, without considering any priority or deadline information.

2.2.2 Weighted Round-Robin

Weighted round-robin extends the basic scheme by assigning different weights to tasks:

- Each job J_i of task τ_i gets ω_i time slices per round.
- The total time in one round corresponds to $\sum_i \omega_i$.
- This allows tasks with higher importance or tighter timing constraints to receive more CPU time.

For instance, in a weighted round-robin with two tasks τ_1 and τ_2 with weights $\omega_1 = 2$ and $\omega_2 = 1$, the execution sequence would be $\tau_1, \tau_1, \tau_2, \tau_1, \tau_1, \tau_2, \ldots$

2.2.3 Properties and Limitations

- Round-robin scheduling provides fairness in the sense that each job gets an equal share of CPU time (or proportional to its weight in weighted round-robin).
- It is well-suited for time-sharing systems but not for real-time systems with hard deadlines.
- The time slice size affects system performance:

  - Too small: High overhead from frequent context switches.
  - Too large: Poor response time for short jobs.

- Round-robin does not account for deadlines or other timing constraints, making it unsuitable for most real-time applications without modifications.
- Not fit for jobs with precedence relations (where one task must complete before another can start) whose execution spans multiple slices.

2.3 Priority-Driven Scheduling

2.3.1 Principles of Priority-Driven Scheduling

Priority-driven scheduling is based on the following principles:

- Each job is assigned a priority that reflects its urgency or importance.
- The job with the highest priority among all ready jobs is executed.
- The algorithm is greedy - it never leaves the processor idle if a job is ready.
- Priority-driven algorithms can be preemptive or non-preemptive.

Priority-driven scheduling algorithms are commonly used in real-time systems due to their flexibility and ability to express various scheduling policies.

2.3.2 Static vs. Dynamic Priorities

Priority-driven scheduling can be classified based on when priorities are assigned:

- **Static-priority scheduling**: Priorities are assigned to tasks at design time and never change during execution. Each job of a task inherits the same priority.
- **Dynamic-priority scheduling**: Priorities may change during execution:
  - **Fixed priority per job**: Each job receives a priority at release, which remains fixed for its execution but may differ from the priorities of other jobs of the same task.
  - **Dynamic priority per job**: The priority of a job can change while it executes.

Examples of static-priority scheduling include Rate Monotonic Scheduling (RMS) and Deadline Monotonic Scheduling (DMS). Examples of dynamic-priority scheduling include Earliest Deadline First (EDF) and Least Laxity First (LLF).

2.3.3 Preemptive vs. Non-preemptive

Priority-driven scheduling can also be classified based on whether preemption is allowed:

- **Preemptive scheduling**: The currently executing job can be interrupted if a higher-priority job becomes ready.
- **Non-preemptive scheduling**: Once a job starts execution, it runs to completion regardless of the arrival of higher-priority jobs.
- **Cooperative scheduling (deferred preemption)**: A compromise where preemption can only occur at specific points in the code, typically between segments of code called non-preemptive regions.

Preemptive scheduling generally offers better responsiveness for high-priority tasks but incurs overhead from context switches and may introduce problems like priority inversion when resources are shared.

2.3.4 Dispatching Points

In priority-driven scheduling, scheduling decisions are made at dispatching points, which include:

- Job releases (arrivals).
- Job completions.
- Resource release events (in systems with shared resources).
- In cooperative scheduling, when a job reaches a preemption point.

2.4 Optimality in Uniprocessor Scheduling

2.4.1 Earliest Deadline First (EDF)

EDF is a dynamic-priority scheduling algorithm where:

- The job with the earliest absolute deadline has the highest priority.
- Priorities are dynamically recomputed at each dispatching point.

- For a job released at time r with relative deadline D, its absolute deadline is $d = r + D$.

Optimality of EDF:
[Liu & Layland, 1973] For preemptive scheduling of independent tasks on a single processor, Earliest Deadline First is optimal: if a feasible schedule exists, EDF will find it.

Utilization-Based Schedulability Test for EDF:
For a set of n periodic tasks with implicit deadlines:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.4}$$

This is a necessary and sufficient condition for EDF schedulability with implicit deadlines.

For constrained deadlines, the demand bound function (DBF) must be used:

$$\forall t > 0 : \sum_{i=1}^{n} \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \cdot C_i \leq t \tag{2.5}$$

2.4.2 Least Laxity First (LLF)

LLF is another dynamic-priority scheduling algorithm where:

- The job with the least laxity (slack time) has the highest priority.
- Laxity $L_i(t) = d_i - t - Y_i(t)$, where d_i is the absolute deadline, t is the current time, and $Y_i(t)$ is the remaining execution time.
- Priorities must be recalculated continuously, as laxity changes with time.

Optimality of LLF:
[Liu & Layland, 1973] For preemptive scheduling of independent tasks on a single processor, Least Laxity First is optimal: if a feasible schedule exists, LLF will find it.

LLF has the same theoretical optimality properties as EDF but suffers from practical drawbacks:

- Higher computational overhead due to frequent priority recalculations.
- Prone to "thrashing" when multiple jobs have similar laxities, causing frequent preemptions between them.

2.4.3 Critical Instant and Busy Period

[Critical Instant] The critical instant for a task is the release time that leads to the maximum response time for that task.

For fixed-priority scheduling of independent tasks with constrained deadlines, the critical instant for a task occurs when it is released simultaneously with all higher-priority tasks.

[Busy Period] A level-i busy period is an interval during which the processor is continuously busy executing jobs of priority i or higher, with release times within the interval.

The concept of busy period is used in response time analysis, especially for tasks with arbitrary deadlines, where multiple jobs of the same task may be active simultaneously.

2.5 Predictability and Sustainability

2.5.1 Predictability

[Predictability] A scheduling algorithm is predictable if the response time and start time of each job vary monotonically with execution time.

In a predictable scheduling algorithm, reducing the execution time of any job cannot increase the response time of any job in the system.

The execution of independent jobs with given release times under preemptive priority-driven scheduling on a single processor is predictable.

Predictability is an important property for real-time systems as it allows worst-case analysis to be performed based on worst-case execution times.

2.5.2 Sustainability

[Sustainability] A scheduling algorithm is sustainable if any task set that is schedulable remains schedulable when:

- Execution times decrease.
- Periods increase.
- Deadlines increase.
- Release jitter decreases.

Sustainability is a desirable property as it ensures that if a system is verified to be schedulable under worst-case conditions, it will remain schedulable under less demanding conditions.

Preemptive EDF and fixed-priority scheduling algorithms are sustainable with respect to execution times, deadlines, and periods for independent tasks on a single processor.

2.6 Scheduling Anomalies

2.6.1 Scheduling Anomalies in Single-Processor Systems

In single-processor systems with fixed-priority scheduling of independent tasks:

- Reducing execution times cannot lead to deadline misses (if the original schedule was feasible).
- Increasing periods cannot lead to deadline misses.
- Increasing deadlines cannot lead to deadline misses.

These properties are a direct consequence of the predictability and sustainability of single-processor fixed-priority scheduling.

2.6.2 Scheduling Anomalies in Self-Suspension

When tasks can suspend themselves (e.g., waiting for I/O), scheduling anomalies can occur:

- Reducing execution times can lead to deadline misses.
- Reducing suspension times can lead to deadline misses.

Consider a task set with three tasks:

- $\tau_1 = (0, 10, [2, 2, 2], 6)$ with periods, where the execution pattern is [compute, suspend, compute].
- $\tau_2 = (5, 10, [1, 1, 1], 4)$
- $\tau_3 = (7, 10, [1, 1, 1], 3)$

This task set is schedulable under EDF. However, if we reduce τ_1's execution or suspension time by 1 unit, τ_3 will miss its deadline.

These anomalies occur because changing execution or suspension times can shift the execution windows of tasks, potentially increasing interference between them and disrupting the schedule in unexpected ways.

Chapter 3

Fixed-Priority Scheduling

3.1 Workload Model

3.1.1 Basic Assumptions

The basic workload model for fixed-priority scheduling assumes:

- A set of n tasks, for constant n.
- All tasks are periodic or sporadic with known periods/minimum inter-arrival times.
- Tasks are independent (no resource sharing or precedence constraints).
- All tasks have constrained or implicit deadlines ($D_i \leq T_i$).
- Each task has a single, fixed and known Worst-Case Execution Time (WCET).
- All runtime overheads (context switching, clock interrupts, etc.) are included in the WCET.

This model, though simplified, provides a foundation for analyzing fixed-priority scheduling systems and can be extended to include more complex features such as shared resources and task interactions.

3.2 Rate Monotonic Scheduling (RMS)

3.2.1 Priority Assignment

In Rate Monotonic Scheduling:

- Priorities are assigned based on task periods.
- The shorter the period, the higher the priority.

- For any two tasks τ_i, τ_j: if $T_i < T_j$ then $P_i > P_j$.

The intuition behind Rate Monotonic Scheduling is that tasks with shorter periods need to execute more frequently and thus should have higher priority to meet their deadlines.

3.2.2 Optimality of RMS

[Liu & Layland, 1973] Rate Monotonic priority ordering is optimal among fixed-priority assignments for periodic tasks with implicit deadlines.

[Sketch] The proof is by contradiction. Assume a task set is schedulable by some fixed-priority assignment but not by Rate Monotonic. Then there must be at least one pair of tasks τ_i and τ_j such that $T_i < T_j$ but $P_i < P_j$. By swapping their priorities and showing that this cannot make an unschedulable system schedulable, we reach a contradiction.

It's important to note that the optimality of RMS holds only for:

- Independent tasks.
- Implicit deadlines ($D_i = T_i$).
- Preemptive scheduling.
- Single-processor systems.

3.2.3 Schedulability Tests

Utilization-Based Test

For a set of n tasks with implicit deadlines under RMS:

$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{3.1}$$

As $n \to \infty$, this bound approaches $\ln 2 \approx 0.693$.

This test is sufficient but not necessary - some task sets with utilization higher than this bound may still be schedulable under RMS. The utilization bound is tight in the sense that for any $\epsilon > 0$, there exists a task set with utilization $n(2^{1/n} - 1) + \epsilon$ that is not schedulable under RMS.

For $n = 2$ tasks, the utilization bound is $2(2^{1/2} - 1) \approx 0.83$. Consider a task set with:

- $\tau_1 = (C_1 = 2, T_1 = 4)$
- $\tau_2 = (C_2 = 4, T_2 = 10)$

The utilization is $U = 2/4 + 4/10 = 0.5 + 0.4 = 0.9 > 0.83$.

However, calculating the response times:

- $R_1 = C_1 = 2$
- $R_2 = C_2 + \lceil R_2/T_1 \rceil \cdot C_1$

Iteratively:

- $R_2^0 = 4$
- $R_2^1 = 4 + \lceil 4/4 \rceil \cdot 2 = 4 + 1 \cdot 2 = 6$
- $R_2^2 = 4 + \lceil 6/4 \rceil \cdot 2 = 4 + 2 \cdot 2 = 8$
- $R_2^3 = 4 + \lceil 8/4 \rceil \cdot 2 = 4 + 2 \cdot 2 = 8 = R_2^2$

So $R_2 = 8 < 10 = D_2$, and the task set is schedulable despite exceeding the utilization bound.

Hyperbolic Bound

The hyperbolic bound (Bini & Buttazzo, 2001) improves the utilization test for RMS:

$$\prod_{i=1}^{n}(U_i + 1) \leq 2 \tag{3.2}$$

where $U_i = C_i/T_i$ is the utilization of task τ_i.

This test is also sufficient but not necessary. However, it is less pessimistic than the original Liu & Layland bound and can accept more schedulable task sets.

3.3 Deadline Monotonic Scheduling (DMS)

3.3.1 Priority Assignment

In Deadline Monotonic Scheduling:

- Tasks are assigned priorities based on their relative deadlines.
- The shorter the relative deadline, the higher the priority.
- For any two tasks τ_i, τ_j: if $D_i < D_j$ then $P_i > P_j$.

For tasks with implicit deadlines ($D_i = T_i$), DMS is equivalent to RMS. However, for tasks with constrained deadlines ($D_i < T_i$), DMS and RMS may assign different priorities.

3.3.2 Optimality of DMS

[Leung & Whitehead, 1982] Deadline Monotonic priority ordering is optimal among fixed-priority assignments for periodic tasks with constrained deadlines ($D_i \leq T_i$).

The proof follows a similar structure to that of RMS optimality, showing that any task set schedulable by some fixed-priority assignment must also be schedulable by DMS.

It's important to note that DMS is not optimal for arbitrary deadlines (D_i can be greater than T_i).

3.4 Response Time Analysis

3.4.1 Basic Concept

Response Time Analysis (RTA) is an exact schedulability test for fixed-priority scheduling. It calculates the worst-case response time of each task and compares it with the task's deadline.

The worst-case response time occurs when:

- The task is released at its critical instant (simultaneously with all higher-priority tasks).
- All tasks experience their worst-case execution times.
- Higher-priority tasks are released as frequently as possible.

3.4.2 Response Time Calculation

For task τ_i under fixed-priority scheduling, the worst-case response time R_i is given by:

$$R_i = C_i + I_i \tag{3.3}$$

where C_i is the task's WCET and I_i is the interference from higher-priority tasks:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.4}$$

where $hp(i)$ is the set of tasks with higher priority than τ_i.

3.4.3 Iterative Solution

The equation for R_i is solved iteratively:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{3.5}$$

Starting with $R_i^0 = C_i$, the iteration continues until either:

- $R_i^{n+1} = R_i^n$ (a fixed point is reached), or
- $R_i^{n+1} > D_i$ (the deadline is exceeded)

The iterative process is guaranteed to converge if the total utilization is less than 1.

Consider a task set with three tasks:

- $\tau_1 = (C_1 = 3, T_1 = 7, D_1 = 7)$
- $\tau_2 = (C_2 = 3, T_2 = 12, D_2 = 12)$
- $\tau_3 = (C_3 = 5, T_3 = 20, D_3 = 20)$

Calculate the response times:
For τ_1:

- $R_1 = C_1 = 3 < D_1 = 7$, so τ_1 is schedulable.

For τ_2:

- $R_2^0 = C_2 = 3$
- $R_2^1 = C_2 + \lceil R_2^0/T_1 \rceil \cdot C_1 = 3 + \lceil 3/7 \rceil \cdot 3 = 3 + 1 \cdot 3 = 6$
- $R_2^2 = C_2 + \lceil R_2^1/T_1 \rceil \cdot C_1 = 3 + \lceil 6/7 \rceil \cdot 3 = 3 + 1 \cdot 3 = 6 = R_2^1$

So $R_2 = 6 < D_2 = 12$, and τ_2 is schedulable.
For τ_3:

- $R_3^0 = C_3 = 5$
- $R_3^1 = C_3 + \lceil R_3^0/T_1 \rceil \cdot C_1 + \lceil R_3^0/T_2 \rceil \cdot C_2 = 5 + \lceil 5/7 \rceil \cdot 3 + \lceil 5/12 \rceil \cdot 3 = 5 + 1 \cdot 3 + 1 \cdot 3 = 11$
- $R_3^2 = C_3 + \lceil R_3^1/T_1 \rceil \cdot C_1 + \lceil R_3^1/T_2 \rceil \cdot C_2 = 5 + \lceil 11/7 \rceil \cdot 3 + \lceil 11/12 \rceil \cdot 3 = 5 + 2 \cdot 3 + 1 \cdot 3 = 14$
- $R_3^3 = C_3 + \lceil R_3^2/T_1 \rceil \cdot C_1 + \lceil R_3^2/T_2 \rceil \cdot C_2 = 5 + \lceil 14/7 \rceil \cdot 3 + \lceil 14/12 \rceil \cdot 3 = 5 + 2 \cdot 3 + 2 \cdot 3 = 17$
- $R_3^4 = C_3 + \lceil R_3^3/T_1 \rceil \cdot C_1 + \lceil R_3^3/T_2 \rceil \cdot C_2 = 5 + \lceil 17/7 \rceil \cdot 3 + \lceil 17/12 \rceil \cdot 3 = 5 + 3 \cdot 3 + 2 \cdot 3 = 20$
- $R_3^5 = C_3 + \lceil R_3^4/T_1 \rceil \cdot C_1 + \lceil R_3^4/T_2 \rceil \cdot C_2 = 5 + \lceil 20/7 \rceil \cdot 3 + \lceil 20/12 \rceil \cdot 3 = 5 + 3 \cdot 3 + 2 \cdot 3 = 20 = R_3^4$

So $R_3 = 20 = D_3 = 20$, and τ_3 is just schedulable.

3.5 Task Interactions

3.5.1 Hard and Soft Tasks

In many systems, both hard and soft real-time tasks coexist:

- Hard tasks must never miss their deadlines.
- Soft tasks can occasionally miss deadlines with degraded performance.

When hard and soft tasks coexist, it's important to ensure that soft tasks do not interfere with the timely execution of hard tasks.

3.5.2 Strategies for Coexistence

There are two main strategies for handling mixed hard/soft task sets:

- **Control knob 1:** Design all tasks to be schedulable using average execution times and arrival rates.
  - In this approach, all tasks are treated equally during design.
  - During transient overloads (when actual execution times exceed average), some tasks may miss deadlines.
  - The system is designed to gracefully degrade under overload.
- **Control knob 2:** Guarantee hard tasks using worst-case parameters for all tasks.
  - Hard tasks are placed at higher priorities.
  - Each hard task is guaranteed to meet its deadline even if all tasks experience their worst-case execution.
  - Soft tasks are placed at lower priorities and may miss deadlines during overloads.

The second approach provides stronger guarantees for hard tasks at the expense of potentially poorer performance for soft tasks.

3.6 Handling Aperiodic Tasks

Aperiodic tasks do not have minimum inter-arrival times and thus cannot claim hard deadlines. However, it's often desirable to provide good responsiveness to them without compromising the schedulability of periodic/sporadic tasks.

3.6.1 Background Execution

The simplest approach is to run aperiodic tasks at the lowest priority, only when no periodic task is ready. This ensures that aperiodic tasks do not interfere with the schedulability of periodic tasks but may result in poor responsiveness for aperiodic tasks.

3.6.2 Slack Stealing

Slack stealing allows aperiodic tasks to use the slack (unused CPU time) of periodic tasks. The slack $\sigma(t)$ at time t is the amount of execution that can be deferred without causing any task to miss its deadline.

For each task τ_i, the slack at time t is:

$$\sigma_i(t) = \max(0, D_i^e - \sum_{k=1}^{i} \left\lceil \frac{D_i^e}{T_k} \right\rceil C_k) \tag{3.6}$$

where D_i^e is the effective deadline of the task.

The system slack is the minimum slack across all tasks:

$$\sigma(t) = \min_i \sigma_i(t) \tag{3.7}$$

Slack stealing works by:

- Computing the available slack at runtime.
- Assigning this slack to aperiodic tasks.
- Reclaiming the slack when periodic tasks complete earlier than their worst-case.

Slack stealing can significantly improve aperiodic response times compared to background execution but has higher overhead due to the need to compute and track slack.

3.6.3 Server Mechanisms

Servers are special tasks dedicated to servicing aperiodic jobs. They have:

- A budget (execution time capacity) C_s.
- A replenishment period T_s.
- A priority assigned according to the system's scheduling policy.

Main types of servers include:

- **Polling Server**: A periodic task that checks for pending aperiodic requests.

  - If no aperiodic job is present when the server is activated, its budget is lost for the current period.
  - Simple but not bandwidth preserving.

- **Deferrable Server**: Preserves its budget when no aperiodic job is present.
  - Can service aperiodic jobs as soon as they arrive, up to its budget.
  - Budget is replenished periodically regardless of consumption.
  - Bandwidth preserving but can cause increased interference on lower-priority tasks.

- **Sporadic Server**: Replenishes its budget based on actual consumption.
  - When the server consumes budget, it schedules a replenishment for a time T_s units after the consumption began.
  - The amount replenished equals the amount consumed.
  - More complex but creates interference equivalent to a regular periodic task.

Consider a system with:

- Periodic tasks $\tau_1 = (C_1 = 1, T_1 = 3)$ and $\tau_2 = (C_2 = 1.5, T_2 = 5)$.
- A Deferrable Server with $(C_{DS} = 0.5, T_{DS} = 2)$ at the highest priority.

For the deferrable server, response time analysis must account for its worst-case interference pattern, which occurs when an aperiodic job arrives just before the end of the server's period, causing the server to use its full budget twice in a short interval.

3.7 Considerations for Practical Implementation

3.7.1 Priority Levels

Real systems may have limited priority levels, requiring tasks to share priorities:

- When tasks share a priority level, they are typically scheduled in FIFO order.
- This can lead to increased blocking and worse response times.

Priority mapping techniques help optimize the use of limited priority levels:

- **Uniform mapping**: Maps ranges of n priorities in the ideal system to single priorities in the real system.

$$\text{For } k = 1, \ldots, \Omega_s : \{(k-1)Q + 1, \ldots, kQ\} \to \pi_k \tag{3.8}$$

  where $Q = \Omega_n / \Omega_s$ is the size of each priority range, Ω_n is the number of ideal priorities, and Ω_s is the number of available priorities.

- **Constant ratio mapping** [Lehoczky & Sha, 1986]: Maps logarithmically increasing ranges of priorities.

$$\text{For } k = 1, \ldots, \Omega_s : \{\mathcal{R}_{k-1} + 1, \ldots, \mathcal{R}_k\} \to \pi_k \tag{3.9}$$

  where $\mathcal{R}_k = \lfloor (\mathcal{R}_{k-1} + 1)/g \rfloor$ and g is the constant ratio.

For systems with limited priority levels, the schedulable utilization bound for RMS with constant ratio mapping approaches:

$$f(g) = \begin{cases} \frac{\ln 2}{g+1-g}, & \frac{1}{2} < g \leq 1 \\ g, & 0 < g \leq \frac{1}{2} \end{cases} \tag{3.10}$$

3.7.2 Release Jitter

Release jitter refers to the variation in the actual release time of a task from its theoretical release time. This can occur due to:

- Clock inaccuracies.
- Delays in detecting release events.
- Variations in the behavior of the task activation mechanism.

For a task with jitter, the worst-case scenario occurs when:

- The task is released as late as possible (maximum jitter).
- But its deadline is still calculated from the earliest possible release time.

The response time equation accounting for release jitter becomes:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{3.11}$$

where J_j is the release jitter of task j.

Chapter 4

Task Interactions and Blocking Effects

4.1 Task Cooperation and Communication

In real-world real-time systems, tasks often need to cooperate and communicate to fulfill their functions. This interaction introduces additional complexities to the scheduling problem.

4.1.1 Communication Methods

Tasks can exchange data in two primary ways:

- **Synchronous communication**: Tasks send messages to one another and wait for responses.
  - Simple conceptually but can lead to unbounded waiting times.
  - Generally avoided in hard real-time systems due to unpredictability.
- **Asynchronous communication**: Tasks share memory locations and access them in read/write mode.
  - More complex to manage but provides better timing predictability.
  - Requires careful access control to prevent data races.
  - Standard approach in most real-time systems.

4.1.2 Challenges of Task Interaction

Task interaction introduces several challenges:

- **Synchronous communication** may lead to unbounded waiting times, which defeats timeliness.

- **Asynchronous communication** requires control mechanisms to ensure predictable behavior in the face of preemption, which can lead to data races or inconsistent views of shared data.
- **Priority inversion** occurs when a high-priority task is blocked waiting for a low-priority task that holds a needed resource.
- **Deadlocks** can occur when tasks wait for resources held by other tasks in a circular dependency.

4.2 Preemption and Critical Sections

4.2.1 Atomic Operations

At the hardware level, CPU instructions are atomic:

- Individual processor instructions cannot be preempted.
- The CPU executes a cycle of micro-operations (fetch, decode, execute, etc.) for each instruction.
- Preemption can only occur between instructions, not during an instruction.

However, most meaningful operations in programs consist of sequences of instructions, and these sequences can be preempted, potentially leading to data races if they access shared data.

4.2.2 Critical Sections

A critical section is a code sequence that accesses shared resources and must be executed atomically (without interference from other tasks).

[Critical Section] A critical section is a sequence of instructions that:

- Accesses shared resources.
- Must be executed without interference from other tasks accessing the same resources.
- Must be protected by synchronization mechanisms to ensure mutual exclusion.

4.2.3 Inhibiting Preemption

The simplest approach to protect critical sections is to inhibit preemption:

- Disable interrupts during the critical section.

- Execute the critical section without the possibility of being preempted.
- Re-enable interrupts after the critical section.

This approach ensures atomicity but at the cost of priority inversion:

- A high-priority task that becomes ready while a low-priority task is executing a non-preemptible critical section will be blocked.
- The blocking time depends on the duration of the critical section.

The maximum blocking time for a task τ_h due to non-preemptible execution is:

$$B_h(np) = \max_{k=h+1,\ldots,n}(\theta_k) \tag{4.1}$$

where θ_k is the longest non-preemptible execution segment of task τ_k.

This blocking occurs at most once per job release, specifically at the release time.

4.2.4 Priority Inversion

[Priority Inversion] Priority inversion occurs when a higher-priority task is blocked by a lower-priority task, typically due to resource sharing.

Consider three tasks with priorities $P_1 > P_2 > P_3$:

- τ_3 (low priority) acquires a shared resource R.
- τ_1 (high priority) becomes ready and tries to acquire R, but is blocked by τ_3.
- τ_2 (medium priority) becomes ready and preempts τ_3.
- τ_1 remains blocked until τ_2 completes and τ_3 can continue to eventually release R.

In this scenario, τ_1 is effectively blocked by τ_2, despite having higher priority. The blocking duration is unpredictable and depends on the execution time of τ_2.

Uncontrolled priority inversion can lead to unpredictable timing behavior and is a major concern in real-time systems.

4.3 Self-Suspension

4.3.1 Effects on Analysis

Self-suspension occurs when a task voluntarily suspends its execution, typically to wait for an external event such as I/O completion. This has significant implications for scheduling analysis:

- A self-suspending task increases its response time by at least the duration of its suspension.
- Self-suspension can lead to scheduling anomalies where reducing execution or suspension times paradoxically increases response times.
- Self-suspension changes the critical instant scenario, making the analysis more complex.

4.3.2 Blocking Due to Self-Suspension

The blocking suffered by task τ_i due to self-suspension includes:

$$B_i(ss) = \max(\delta_i) + \sum_{k=1}^{i-1} \min(e_k, \max(\delta_k)) \tag{4.2}$$

where:

- δ_i is the maximum self-suspension time of τ_i.
- e_k is the execution time of τ_k.
- The summation represents the interference from higher-priority tasks that may resume from self-suspension and preempt τ_i.

For a task τ_i that self-suspends K times during execution, the total blocking is:

$$B_i = B_i(ss) + (K + 1) \cdot B_i(np) \tag{4.3}$$

This is because non-preemptive blocking can occur after each self-suspension when the task resumes execution.

4.4 Resource Access Protocols

Resource access protocols are designed to control access to shared resources and mitigate priority inversion problems. They specify when and how tasks can acquire and release shared resources.

4.4.1 Priority Inheritance Protocol (PIP)

Basic Priority Inheritance Protocol (BPIP):

- When a high-priority task is blocked by a lower-priority task holding a needed resource, the lower-priority task inherits the priority of the blocked task.
- The inherited priority is retained until the resource is released, at which point the task reverts to its original priority.
- This approach prevents unbounded priority inversion by ensuring that if a task blocks a higher-priority task, it executes at the higher priority level, preventing preemption by medium-priority tasks.

Protocol Rules:

1. **Scheduling:** Tasks are scheduled according to preemptive priority-driven scheduling.
2. **Allocation:** When task τ_j requires access to resource R at time t:
   - If R is free, R is assigned to τ_j until it is released.
   - If R is busy, τ_j's request is denied and it becomes blocked.
3. **Priority Inheritance:** When task τ_j is blocked by task τ_l, τ_l inherits τ_j's priority. This inherited priority is retained until τ_l releases R.

Limitations of PIP:

- Does not prevent deadlocks.
- Does not prevent chained blocking (when a task is blocked multiple times, once for each resource it needs).
- Priority inheritance is transitive, which can lead to complex blocking patterns.

Consider three tasks with priorities $P_1 > P_2 > P_3$ and two resources R_1 and R_2:

1. τ_3 acquires R_1.
2. τ_1 becomes ready and attempts to acquire R_1 but is blocked. τ_3 inherits priority P_1.
3. τ_2 becomes ready but cannot preempt τ_3 because τ_3 is now running at priority P_1.
4. τ_3 releases R_1 and reverts to priority P_3.
5. τ_1 acquires R_1 and continues execution.

Under PIP, τ_1 is only blocked for the duration of τ_3's critical section, not for the execution of τ_2 as would occur without the protocol.

4.4.2 Priority Ceiling Protocol (PCP)

Basic Priority Ceiling Protocol (BPCP) extends PIP with additional constraints to prevent deadlocks and reduce blocking:

- Each resource R is assigned a **priority ceiling**, denoted as π_R, which is the highest priority of any task that may use the resource.
- At time t, the system has a **system ceiling** $\pi_s(t)$, which is the highest priority ceiling of all resources currently in use.
- If no resource is in use at time t, $\pi_s(t)$ is set to a value Ω lower than the lowest task priority.

Protocol Rules:

1. **Scheduling:** Tasks are scheduled according to preemptive priority-driven scheduling.
2. **Allocation:** When task τ_j with priority π_j requests resource R at time t:
   - If R is already assigned, the request is denied and τ_j becomes blocked.
   - If R is free and $\pi_j > \pi_s(t)$, the request is granted.
   - If τ_j currently owns the resource whose priority ceiling equals $\pi_s(t)$, the request is granted.
   - Otherwise, the request is denied and τ_j becomes blocked. This is known as **avoidance blocking**.
3. **Priority Inheritance:** When task τ_j is blocked by a lower-priority task τ_l, τ_l inherits τ_j's priority until τ_l releases all resources with priority ceiling $\geq \pi_j$.

[Sha, Rajkumar & Lehoczky, 1990] Under the Priority Ceiling Protocol, a task can be blocked for at most the duration of one critical section.

The maximum blocking time for task τ_i due to resource contention is:

$$B_i(rc) = \max_{k=i+1,\ldots,n} \{C_k(r) | \pi_r \geq \pi_i\} \tag{4.4}$$

where $C_k(r)$ is the execution time of task τ_k inside its critical section for resource r, and π_r is the priority ceiling of resource r.

PCP prevents:

- Deadlocks (through avoidance blocking).
- Chained blocking (through the priority ceiling rule).
- Transitive blocking (a task can be blocked at most once per job).

4.4.3 Stack Resource Policy (SRP)

Stack-based Ceiling Priority Protocol (SB-CPP or SRP) is a variant of PCP designed to allow stack sharing among tasks:

- Similar to PCP but designed to prevent blocking after task activation.
- A task is not allowed to start execution until its priority is higher than the system ceiling.
- Once a task starts execution, it will never be blocked on resource access.

Protocol Rules:

1. **System Ceiling Computation:** When all resources are free, $\pi_s(t) = \Omega$ (below the lowest priority). The ceiling is updated whenever a resource is assigned or released.
2. **Scheduling:** Upon release at time t, task τ_j with priority π_j remains **blocked** until $\pi_j > \pi_s(t)$. Tasks that are not blocked are scheduled according to preemptive priority-driven scheduling.
3. **Allocation:** Whenever a task issues a request for a resource, the request is always granted.

SRP has several advantageous properties:

- A task can be blocked at most once, and only before it starts execution.
- Tasks can share a single stack, as they never suspend execution once started.
- SRP prevents deadlocks and provides the same blocking bound as PCP.
- Simpler implementation than PCP because it avoids the need to track priority inheritance.

4.4.4 Ceiling Priority Protocol (CPP)

Ceiling Priority Protocol (CPP) is a variant of PCP that does not use the system ceiling concept:

- Each resource has a priority ceiling as in PCP.
- When a task acquires a resource, its priority is immediately raised to the resource's priority ceiling.
- The raised priority is maintained until the resource is released.

Protocol Rules:

1. **Scheduling:** Tasks are scheduled with fixed-priority preemptive scheduling with "FIFO within priorities" as the tie-breaking rule.

2. **Execution Priority:** A task that does not hold any resource runs at its assigned priority. A task that acquires a resource runs at the highest priority ceiling of all resources it holds.

3. **Allocation:** Whenever a task issues a request for a resource, the request is granted.

CPP has similar blocking bounds to PCP but with simpler implementation requirements.

4.5 Computing Blocking Times

4.5.1 Direct, Inheritance, and Avoidance Blocking

Under BPCP, a task can experience three types of blocking:

1. **Direct Blocking:** Occurs when a task attempts to access a resource that is already locked by a lower-priority task.

2. **Inheritance Blocking:** Occurs when a task is preempted by a lower-priority task that has inherited a higher priority.

3. **Avoidance Blocking:** Occurs when a task is denied access to a free resource because the system ceiling is too high (to prevent potential deadlocks).

To calculate blocking times, a system of tables is often used:

- "Directly blocked by" table shows the maximum blocking when a specific task accesses a specific resource.

- "Inheritance blocked by" table shows the maximum blocking due to priority inheritance.

- "Avoidance blocked by" table shows the maximum blocking due to ceiling-based resource denial.

The worst-case blocking for a task is the maximum value across all three tables.

4.5.2 Comparison of Resource Access Protocols

Protocol	Prevents Deadlocks	Prevents Chained Blocking	Maximum Blocking	Implen
BPIP	No	No	Multiple sections	
BPCP	Yes	Yes	One section	
SRP	Yes	Yes	One section	
CPP	Yes	Yes	One section	

- **BPIP:** Simplest to implement but provides weaker guarantees. Appropriate for systems with simpler resource usage patterns.
- **BPCP:** Strongest guarantees but most complex to implement. Appropriate for systems with complex resource usage patterns.
- **SRP:** Similar guarantees to BPCP with the additional benefit of stack sharing. Appropriate for memory-constrained systems.
- **CPP:** Good balance of guarantees and implementation simplicity. Appropriate for many practical systems.

Chapter 5

Further Model Extensions

5.1 Cooperative Scheduling

5.1.1 Deferred Preemption

Cooperative or deferred-preemption scheduling is a compromise between fully preemptive and non-preemptive scheduling:

- Tasks are divided into non-preemptible slots or regions.
- Preemption can only occur between these slots.
- Tasks voluntarily yield control at the end of each slot by calling a `yield` function.

This approach allows for better control over preemption points, reducing context-switching overhead and improving predictability.

5.1.2 Fixed vs. Floating Non-Preemptive Regions

Non-preemptive regions can be implemented in two ways:

- **Fixed non-preemptive regions:** The location of non-preemptive regions is predetermined at design time. Preemption points occur at fixed locations in the code.
- **Floating non-preemptive regions:** Tasks can decide when to enter a non-preemptive region based on runtime conditions. This provides more flexibility but may make timing analysis more complex.

Deferred preemption scheduling dominates both fully preemptive and non-preemptive scheduling: any task set schedulable under either fully preemptive or non-preemptive scheduling is also schedulable under deferred preemption scheduling.

5.1.3 Response Time Analysis with Deferred Preemption

For a task with a final non-preemptive region of length F_i, the response time equation is modified:

$$w_i^{n+1} = C_i + B_{max} + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j - F_i \tag{5.1}$$

$$R_i = w_i^n + F_i \tag{5.2}$$

The final non-preemptive region effectively reduces the interference from higher-priority tasks, as they cannot preempt during this region.

Consider a task set with three tasks:

- $\tau_1 = (C_1 = 2, T_1 = 10, D_1 = 10)$ with final non-preemptive region $F_1 = 1$
- $\tau_2 = (C_2 = 3, T_2 = 15, D_2 = 15)$ with final non-preemptive region $F_2 = 1$
- $\tau_3 = (C_3 = 5, T_3 = 20, D_3 = 20)$ with final non-preemptive region $F_3 = 2$

For τ_3, we would calculate:

$$\begin{aligned}
w_3^0 &= C_3 - F_3 = 5 - 2 = 3 \\
w_3^1 &= C_3 - F_3 + \lceil w_3^0/T_1 \rceil \cdot C_1 + \lceil w_3^0/T_2 \rceil \cdot C_2 \\
&= 3 + \lceil 3/10 \rceil \cdot 2 + \lceil 3/15 \rceil \cdot 3 \\
&= 3 + 1 \cdot 2 + 1 \cdot 3 = 8 \\
w_3^2 &= C_3 - F_3 + \lceil w_3^1/T_1 \rceil \cdot C_1 + \lceil w_3^1/T_2 \rceil \cdot C_2 \\
&= 3 + \lceil 8/10 \rceil \cdot 2 + \lceil 8/15 \rceil \cdot 3 \\
&= 3 + 1 \cdot 2 + 1 \cdot 3 = 8 = w_3^1
\end{aligned}$$

Therefore, $R_3 = w_3^2 + F_3 = 8 + 2 = 10$.

Without the final non-preemptive region, the response time would be:

$$R_3 = C_3 + \lceil R_3/T_1 \rceil \cdot C_1 + \lceil R_3/T_2 \rceil \cdot C_2$$

Solving iteratively:

$$\begin{aligned}
R_3^0 &= 5 \\
R_3^1 &= 5 + \lceil 5/10 \rceil \cdot 2 + \lceil 5/15 \rceil \cdot 3 = 5 + 1 \cdot 2 + 1 \cdot 3 = 10 \\
R_3^2 &= 5 + \lceil 10/10 \rceil \cdot 2 + \lceil 10/15 \rceil \cdot 3 = 5 + 1 \cdot 2 + 1 \cdot 3 = 10 = R_3^1
\end{aligned}$$

So, $R_3 = 10$ with or without the final non-preemptive region in this example. However, with a different task set, the final non-preemptive region could make a difference in schedulability.

5.2 Release Jitter

5.2.1 Sources of Jitter

Release jitter is the variation in the actual release time of a task from its theoretical release time. It can arise from several sources:

- **Clock inaccuracies:** Variations in the clock that triggers periodic tasks.
- **Precedence constraints:** When a task must wait for another task to complete before it can start.
- **Variations in response times:** When a task's release is triggered by the completion of another task, variations in the predecessor's response time translate to jitter in the successor's release.
- **RTOS overheads:** Delays in detecting and processing release events due to system overheads.

5.2.2 Jitter in Precedence Constraints

In systems where tasks have precedence constraints, jitter propagates through the task chain:

- If task τ_k releases task τ_v at the end of its execution, the release time jitter of τ_v is affected by the response time jitter of τ_k.
- The release jitter of τ_v is the difference between the worst-case and best-case response times of τ_k: $J_v = R_k - R_k^{best}$.
- This can lead to minimum inter-arrival times for τ_v that are shorter than the period of τ_k, specifically: $T_v = T_k - J_v$.

Consider a periodic task τ_k with period $T_k = 20$ that triggers a sporadic task τ_v. If τ_k has a worst-case response time $R_k = 15$ and a best-case response time $R_k^{best} = 1$, then τ_v has a release jitter $J_v = 15 - 1 = 14$.

The minimum inter-arrival time for τ_v could be as low as $T_v = T_k - J_v = 20 - 14 = 6$, which is much shorter than τ_k's period of 20.

5.2.3 Effects on Analysis

Release jitter affects the interference that higher-priority tasks impose on lower-priority tasks:

- A task with jitter may arrive in bursts, with consecutive arrivals separated by less than its nominal period.
- This increases the interference on lower-priority tasks within a given time interval.

The response time equation accounting for release jitter becomes:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{5.3}$$

where J_j is the release jitter of task j.

5.3 Arbitrary Deadlines

5.3.1 The Busy Period Analysis

When task deadlines can exceed their periods ($D_i > T_i$), multiple jobs of the same task may be active simultaneously. This complicates the analysis:

- We need to consider the response time of each job within a level-i busy period.
- A level-i busy period is an interval during which the processor is continuously busy executing tasks with priority i or higher.

For tasks with arbitrary deadlines, the response time analysis must consider multiple job releases:

$$w_i^{n+1}(q) = (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \tag{5.4}$$

where $q = 0, 1, 2, \ldots$ represents the job index within the busy period.
The response time of the q-th job is:

$$R_i(q) = w_i^n(q) - qT_i \tag{5.5}$$

The worst-case response time for task τ_i is:

$$R_i = \max_q R_i(q) \tag{5.6}$$

The number of jobs to check is bounded by the smallest q such that $R_i(q) \leq T_i$ or $w_i(q) \leq (q+1)T_i$.
Consider a task set with two tasks:

- $\tau_1 = (C_1 = 3, T_1 = 7, D_1 = 7)$
- $\tau_2 = (C_2 = 5, T_2 = 8, D_2 = 12)$

For τ_2, we need to check multiple jobs within the busy period:
For $q = 0$ (first job):

$$w_2^0(0) = C_2 = 5$$
$$w_2^1(0) = C_2 + \lceil w_2^0(0)/T_1 \rceil \cdot C_1 = 5 + \lceil 5/7 \rceil \cdot 3 = 5 + 1 \cdot 3 = 8$$
$$w_2^2(0) = C_2 + \lceil w_2^1(0)/T_1 \rceil \cdot C_1 = 5 + \lceil 8/7 \rceil \cdot 3 = 5 + 2 \cdot 3 = 11$$
$$w_2^3(0) = C_2 + \lceil w_2^2(0)/T_1 \rceil \cdot C_1 = 5 + \lceil 11/7 \rceil \cdot 3 = 5 + 2 \cdot 3 = 11 = w_2^2(0)$$

Therefore, $R_2(0) = w_2^3(0) - 0 \cdot T_2 = 11 - 0 = 11 < D_2 = 12$.
For $q = 1$ (second job):

$$w_2^0(1) = 2C_2 = 10$$
$$w_2^1(1) = 2C_2 + \lceil w_2^0(1)/T_1 \rceil \cdot C_1 = 10 + \lceil 10/7 \rceil \cdot 3 = 10 + 2 \cdot 3 = 16$$
$$w_2^2(1) = 2C_2 + \lceil w_2^1(1)/T_1 \rceil \cdot C_1 = 10 + \lceil 16/7 \rceil \cdot 3 = 10 + 3 \cdot 3 = 19$$
$$w_2^3(1) = 2C_2 + \lceil w_2^2(1)/T_1 \rceil \cdot C_1 = 10 + \lceil 19/7 \rceil \cdot 3 = 10 + 3 \cdot 3 = 19 = w_2^2(1)$$

Therefore, $R_2(1) = w_2^3(1) - 1 \cdot T_2 = 19 - 8 = 11 < D_2 = 12$.
Since $w_2(1) = 19 > (1 + 1) \cdot T_2 = 16$, we need to check $q = 2$:

$$w_2^0(2) = 3C_2 = 15$$
$$w_2^1(2) = 3C_2 + \lceil w_2^0(2)/T_1 \rceil \cdot C_1 = 15 + \lceil 15/7 \rceil \cdot 3 = 15 + 3 \cdot 3 = 24$$
$$w_2^2(2) = 3C_2 + \lceil w_2^1(2)/T_1 \rceil \cdot C_1 = 15 + \lceil 24/7 \rceil \cdot 3 = 15 + 4 \cdot 3 = 27$$
$$w_2^3(2) = 3C_2 + \lceil w_2^2(2)/T_1 \rceil \cdot C_1 = 15 + \lceil 27/7 \rceil \cdot 3 = 15 + 4 \cdot 3 = 27 = w_2^2(2)$$

Therefore, $R_2(2) = w_2^3(2) - 2 \cdot T_2 = 27 - 16 = 11 < D_2 = 12$.
Since $w_2(2) = 27 > (2 + 1) \cdot T_2 = 24$, we need to check $q = 3$:

$$w_2^0(3) = 4C_2 = 20$$
$$w_2^1(3) = 4C_2 + \lceil w_2^0(3)/T_1 \rceil \cdot C_1 = 20 + \lceil 20/7 \rceil \cdot 3 = 20 + 3 \cdot 3 = 29$$
$$w_2^2(3) = 4C_2 + \lceil w_2^1(3)/T_1 \rceil \cdot C_1 = 20 + \lceil 29/7 \rceil \cdot 3 = 20 + 5 \cdot 3 = 35$$
$$w_2^3(3) = 4C_2 + \lceil w_2^2(3)/T_1 \rceil \cdot C_1 = 20 + \lceil 35/7 \rceil \cdot 3 = 20 + 5 \cdot 3 = 35 = w_2^2(3)$$

Therefore, $R_2(3) = w_2^3(3) - 3 \cdot T_2 = 35 - 24 = 11 < D_2 = 12$.

We continue checking until we find a q such that $R_2(q) \leq T_2$ or $w_2(q) \leq (q+1)T_2$. Let's check $q = 4$:

$$w_2^0(4) = 5C_2 = 25$$
$$w_2^1(4) = 5C_2 + \lceil w_2^0(4)/T_1 \rceil \cdot C_1 = 25 + \lceil 25/7 \rceil \cdot 3 = 25 + 4 \cdot 3 = 37$$
$$w_2^2(4) = 5C_2 + \lceil w_2^1(4)/T_1 \rceil \cdot C_1 = 25 + \lceil 37/7 \rceil \cdot 3 = 25 + 6 \cdot 3 = 43$$
$$w_2^3(4) = 5C_2 + \lceil w_2^2(4)/T_1 \rceil \cdot C_1 = 25 + \lceil 43/7 \rceil \cdot 3 = 25 + 7 \cdot 3 = 46$$
$$w_2^4(4) = 5C_2 + \lceil w_2^3(4)/T_1 \rceil \cdot C_1 = 25 + \lceil 46/7 \rceil \cdot 3 = 25 + 7 \cdot 3 = 46 = w_2^3(4)$$

Therefore, $R_2(4) = w_2^4(4) - 4 \cdot T_2 = 46 - 32 = 14 > D_2 = 12$.

The fourth job of τ_2 misses its deadline. Therefore, the task set is not schedulable.

5.4 Offsets

5.4.1 Benefits of Offsets

Offsets specify the relative release times of tasks and can provide several benefits:

- Reduce peak processor load by spreading out task releases.
- Improve schedulability by avoiding the worst-case scenario where all tasks are released simultaneously.
- Create precedence relations between tasks to model functional dependencies.
- Reduce interference between tasks with the same period.

5.4.2 Analysis with Static Offsets

Static offsets complicate the analysis because the critical instant assumption (all tasks released simultaneously) no longer holds:

- The critical instant can occur at the release of any task within the hyperperiod.
- For task sets with many tasks, checking all possible release combinations becomes intractable.

For tasks with offsets, a non-optimal but tractable analysis approach is to:

- Replace task pairs with the same period with a notional task that represents their combined effect.
- The notional task has:

$$\begin{aligned} T_n &= T_a = T_b \\ C_n &= \max(C_a, C_b) \\ D_n &= \min(D_a, D_b) \\ P_n &= \max(P_a, P_b) \end{aligned} \tag{5.7}$$

- Perform standard analysis on the modified task set.

This approach is pessimistic but computationally feasible. More advanced techniques, such as offset-based RTA, have been developed to provide less pessimistic results.

5.5 Transactions

5.5.1 Task Concatenations

In a transaction, the completion of one task typically triggers the release of the next task in the chain. The original periodic (or sporadic) nature of the first task in the transaction propagates to the following tasks, but with the addition of release jitter. This jitter accumulates through the transaction as we have seen in the section on release jitter.

The end-to-end timing properties of a transaction are crucial:

- **End-to-end response time:** The time from the release of the first task in the transaction to the completion of the last task. This must be less than or equal to the end-to-end deadline for the transaction to be considered feasible.
- **End-to-end deadline:** The maximum allowable time for the entire transaction to complete.

5.5.2 End-to-End Analysis

Analyzing the timing properties of transactions requires adapting standard response time analysis techniques:

- Tasks later in the transaction can inherit release jitter from earlier tasks.
- The release jitter of a task in a transaction is equal to the worst-case completion time variation of its predecessor.

- This can be modeled as dynamic offsets, where each task's offset is dependent on the response times of preceding tasks.

$$J_i = R_{i-1}^{wc} - R_{i-1}^{bc} \tag{5.8}$$

where R_{i-1}^{wc} is the worst-case response time of the predecessor task and R_{i-1}^{bc} is its best-case response time.

The worst-case end-to-end response time of a transaction Γ is:

$$R_\Gamma = \sum_{i \in \Gamma} (R_i - J_i) + J_{\text{first}} \tag{5.9}$$

where J_{first} is the release jitter of the first task in the transaction.

5.6 Worst-Case Execution Time Analysis

5.6.1 The WCET Challenge

The Worst-Case Execution Time (WCET) is the maximum time a task could take to execute on a given hardware platform:

- Must account for all possible inputs and initial states
- Must consider the worst-case execution path through the program
- Must account for hardware effects like caches, pipelines, and branch prediction

Determining the exact WCET is generally not computable (a version of the halting problem), but safe upper bounds can be established.

WCET bounds must be:

- **Safe:** To upper-bound all possible executions
- **Tight:** To avoid costly over-dimensioning

5.6.2 WCET Analysis Techniques

Static Analysis

Static WCET analysis examines a program without executing it:

- **Flow analysis:** Determines possible execution paths
  - Control flow analysis builds a control flow graph (CFG)
  - Value analysis resolves memory accesses
  - Loop bound analysis determines maximum iteration counts

- **Processor behavior modeling:** Models the timing of instructions
  - Cache analysis predicts cache hits and misses
  - Pipeline analysis models instruction timing including stalls
  - Branch prediction analysis estimates branch misprediction penalties
- **Calculation:** Computes the WCET from the above analyses
  - Path-based approaches enumerate all paths
  - Tree-based approaches calculate bottom-up on the syntax tree
  - IPET (Implicit Path Enumeration Technique) uses integer linear programming

Measurement-Based Analysis

Measurement-based WCET analysis relies on empirical observation:

- Execute the program with various inputs and measure execution times
- Apply statistical methods to estimate WCET
- Add safety margins to account for unobserved cases

Limitations include:

- Difficulty in generating worst-case inputs
- Inability to guarantee coverage of all execution scenarios
- Hardware effects (e.g., caches) making measurements unstable

Hybrid Approaches

Hybrid WCET analysis combines static and measurement-based methods:

- Static analysis to identify the structure and possible paths
- Measurements of small code segments (basic blocks) on the target hardware
- Combination of measurements using the structure information

5.6.3 Challenges in WCET Analysis

Modern processors present numerous challenges for WCET analysis:

- **Caches:** Create history-dependent execution times
- **Pipelines:** Create complex interactions between instructions
- **Branch prediction:** Introduces variability based on execution history
- **Out-of-order execution:** Makes timing analysis more complex
- **Shared resources in multicore:** Creates interference between cores
- **Timing anomalies:** Local worst case doesn't always lead to global worst case

[Timing Anomaly] A situation where the local worst case (e.g., a cache miss) does not necessarily lead to the global worst-case execution time.

5.6.4 WCET Tools and Techniques

Several tools and techniques have been developed for WCET analysis:

- Commercial tools like aiT, RapiTime, and Bound-T
- Academic tools like SWEET, OTAWA, and Chronos
- Specialized programming languages and guidelines for timing predictability
- Processor architectures designed for predictability

5.7 Introduction to Real-Time Scheduling on Multicore Processors

5.7.1 Motivation for Multicore

The paradigm shift from single-core to multicore processors was driven by:

- **Power constraints:** The "power wall" limiting frequency scaling
- **ILP limits:** Diminishing returns from instruction-level parallelism
- **Moore's Law continuation:** Transistor density increases enabling more cores

This shift creates both opportunities and challenges for real-time systems:

- Opportunities for increased throughput and parallel execution
- Challenges for predictability due to shared resources and complex interactions

5.7.2 Multicore Architectures

Multicore processors come in various configurations:

- **Homogeneous:** All cores identical
- **Heterogeneous:** Different types of cores (e.g., big.LITTLE)
- **Cache organization:** Private vs. shared caches
- **Memory organization:** Uniform vs. non-uniform memory access (NUMA)

5.7.3 Resource Sharing in Multicore

In a multicore processor, cores share various resources:

- **Last-level cache:** Shared among all cores
- **Memory controller:** Controls access to main memory
- **Memory bus:** Transfers data between caches and memory
- **I/O devices:** Peripherals shared among cores

Contention for these shared resources creates interference, which affects execution times and can harm predictability.

5.7.4 Multicore Scheduling Paradigms

There are three primary approaches to multicore scheduling:

- **Partitioned scheduling:** Each task is statically assigned to a specific core, and each core is scheduled independently
  - Advantages: Simpler analysis, no migration costs, better cache locality
  - Disadvantages: Limited to task sets that can be partitioned effectively
- **Global scheduling:** All tasks can execute on any core, and there is a single ready queue for all cores
  - Advantages: Better load balancing, higher utilization possible
  - Disadvantages: Migration costs, more complex analysis, cache inefficiency
- **Hybrid scheduling:** Combines elements of both approaches

  - Clustered scheduling: Cores are grouped into clusters, with partitioning between clusters and global scheduling within clusters
  - Semi-partitioned scheduling: Most tasks are partitioned, but some tasks can migrate between specific cores

5.7.5 Challenges in Multicore Scheduling

Multicore scheduling faces several challenges not present in single-core systems:

- **Dhall's effect:** A phenomenon where high-utilization tasks can cause global EDF and RM to miss deadlines even at low total utilization
- **Scheduling anomalies:** Counterintuitive behavior where reducing execution time or increasing resources leads to worse schedulability
- **Task allocation:** The bin-packing problem of assigning tasks to cores
- **Work conservation:** Ensuring processors don't idle when work is available
- **Inter-core interference:** Accounting for shared resource contention

5.7.6 Multicore Schedulability Analysis

Schedulability analysis for multicore systems is more complex than for single-core:

- **Partitioned scheduling:** Apply single-core analysis per core, after solving the task allocation problem
- **Global scheduling:** Must account for migration costs and variable interference patterns
- **Response time analysis:** Must consider interference from tasks on other cores
- **Utilization bounds:** Generally lower than single-core bounds

  Sufficient utilization bounds for global EDF:

$$U_{sum} \leq m - (m-1) \cdot U_{max} \tag{5.10}$$

  where m is the number of cores and U_{max} is the maximum task utilization.

5.8 Seeking the Lost Optimality

5.8.1 The Optimality Problem

Optimality has different meanings in scheduling:

- **Optimal priority assignment:** For a given scheduling algorithm, finding the priority ordering that makes the most task sets schedulable
- **Optimal scheduling algorithm:** An algorithm that can schedule any feasible task set

In single-core systems, EDF is optimal for preemptive scheduling of independent tasks with implicit deadlines. However, this optimality is lost in multicore systems.

5.8.2 Proportionate Fairness (P-Fair)

P-fair scheduling was the first optimal multicore scheduling algorithm:

- Based on the fluid scheduling model, where each task makes progress at a constant rate proportional to its utilization
- Requires tasks to receive processor time in proportion to their utilization at each time unit
- Ensures each task τ_i with utilization u_i receives between $\lfloor t \cdot u_i \rfloor$ and $\lceil t \cdot u_i \rceil$ time units by time t
- Requires frequent preemptions and migrations, making it impractical

5.8.3 DP-Fair and LLREF

DP-Fair (Deadline-Partitioning Fair) and LLREF (Largest Local Remaining Execution First) reduce the overhead of P-fair:

- Divide time into slices based on task deadlines
- Ensure proportionate progress only at slice boundaries
- Use simpler dispatching rules within time slices
- Remain optimal while reducing preemptions

5.8.4 RUN: Reduction to Uniprocessor

RUN (Reduction to UNiprocessor) is an optimal multicore scheduling algorithm with lower overhead:

- Based on the insight that scheduling m tasks with total utilization $m - 1$ on m processors is equivalent to scheduling the dual task set with utilization 1 on a single processor
- Uses a reduction tree to recursively convert a multiprocessor scheduling problem into a uniprocessor one
- Employs packing and server techniques to reduce task set size
- Achieves optimality with significantly fewer preemptions and migrations than other optimal algorithms

5.8.5 QPS and Other Approaches

QPS (Quasi-Partitioned Scheduling) and other recent approaches aim to combine optimality with practical efficiency:

- QPS extends RUN with more efficient reductions
- U-EDF uses utilization information to make EDF decisions
- NVNLF (Non-Vary Non-Laxity First) combines LLF principles with limited migrations

5.9 Sharing Resources Across Processors

5.9.1 The Multiprocessor Resource Sharing Problem

Resource sharing in multiprocessor systems introduces new challenges:

- **Blocking chains:** Dependencies across processors can create complex blocking patterns
- **Remote blocking:** Tasks blocked by tasks on other processors
- **Priority inversion:** More complex due to parallel execution
- **Deadlocks:** More likely with distributed resources

5.9.2 Suspension-Based Protocols

Suspension-based protocols allow tasks waiting for a resource to suspend:

- **MPCP (Multiprocessor Priority Ceiling Protocol)**
  - Resources have a global ceiling priority
  - Tasks accessing global resources execute at ceiling priority
  - Tasks blocked on a global resource suspend and are placed in a priority queue
- **FMLP (Flexible Multiprocessor Locking Protocol)**
  - Classifies resources as short or long
  - Short resources use spin locks
  - Long resources use suspension and FIFO ordering
- **MPCP+ and DPCP (Distributed Priority Ceiling Protocol)**
  - Extend MPCP for various multiprocessor configurations
  - Use synchronization processors for global resources

5.9.3 Spin-Based Protocols

Spin-based protocols keep tasks actively waiting for resources:

- **MSRP (Multiprocessor Stack Resource Policy)**
  - Tasks spin when blocked on a global resource
  - FIFO ordering for access to global resources
  - Non-preemptible critical sections
- **SPEPP (Spinning Processor Executes for Preempted Processor)**
  - Tasks spin when blocked
  - If a lock holder is preempted, a spinning task executes on its behalf
- **MrsP (Multiprocessor Resource Sharing Protocol)**
  - Combines MSRP with helping mechanisms
  - Lock holders can migrate to spinning processors
  - Preserves response-time analysis compatibility

5.9.4 Blocking Analysis for Multiprocessor Protocols

Analyzing blocking in multiprocessor systems is more complex:

- **Suspension-oblivious analysis:** Treats suspensions as execution, simpler but pessimistic
- **Suspension-aware analysis:** Accounts for the effect of suspensions, more accurate but more complex
- **Blocking components:** Direct blocking, push-through blocking, inheritance blocking, and more

5.9.5 Priority Assignments and Blocking

Priority assignment interacts with resource sharing:

- Traditional assignments (RM, DM) may be suboptimal
- Blocking-aware priority assignments consider the effect of sharing
- Optimal assignment for resource sharing is NP-hard

5.10 Mixed-Criticality Systems

5.10.1 Motivation and Background

Mixed-criticality systems integrate components of different importance or criticality:

- **Safety-critical components:** Must meet stringent certification requirements
- **Non-critical components:** Provide added functionality but failures are tolerable
- **Resource efficiency:** Need to efficiently use resources while maintaining guarantees

Traditional approaches use static partitioning (temporal and spatial isolation) but can be inefficient.

5.10.2 Vestal's Model

Vestal's model (2007) introduced a formal framework for mixed-criticality:

- Tasks have different WCET estimates for different criticality levels

- More conservative estimates for higher criticality levels
- System operates in different modes corresponding to criticality levels
- Mode changes occur when a task exceeds its WCET estimate for the current mode

[Mixed-Criticality Task] A mixed-criticality task τ_i is characterized by the tuple $(T_i, D_i, L_i, \vec{C_i})$, where:

- T_i is the period
- D_i is the deadline
- L_i is the criticality level
- $\vec{C_i} = (C_i(1), C_i(2), \ldots, C_i(L_i))$ is the vector of WCET estimates for different criticality levels

5.10.3 Scheduling in Mixed-Criticality Systems

Several approaches have been developed for mixed-criticality scheduling:

- **Fixed-priority approaches**
  - Criticality-monotonic: Higher criticality tasks get higher priorities
  - Audsley's algorithm adapted for mixed criticality
  - Period transformation to achieve criticality-monotonic scheduling
- **EDF-based approaches**
  - EDF-VD: Uses virtual deadlines that are shortened for high-criticality tasks
  - EQDF: Incorporates elasticity in task parameters
- **Mode change approaches**
  - Adaptive Mixed Criticality (AMC): Standard model with mode changes
  - Bailout protocols: Allow some low-criticality execution after mode change
  - Elastic scheduling: Adjust task parameters during mode changes

5.10.4 Adaptive Mixed-Criticality (AMC)

AMC is a prominent approach for mixed-criticality scheduling:

- System starts in LO-criticality mode
- When a high-criticality task exceeds its LO-criticality WCET, the system switches to HI-criticality mode
- In HI-criticality mode, low-criticality tasks are abandoned or executed at reduced rates
- High-criticality tasks are guaranteed their HI-criticality WCET

AMC Schedulability Analysis

Schedulability analysis for AMC considers three scenarios:

1. LO-criticality mode: All tasks meet their deadlines using LO-criticality WCETs
2. HI-criticality mode: High-criticality tasks meet their deadlines using HI-criticality WCETs
3. Mode change: The transition from LO to HI criticality

$$R_i^{LO} = C_i(LO) + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j(LO) \tag{5.11}$$

$$R_i^{HI} = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j(HI) \tag{5.12}$$

$$R_i^{*} = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i^{*}}{T_j} \right\rceil C_j(HI) + \sum_{j \in hpL(i)} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j(LO) \tag{5.13}$$

where $hpH(i)$ and $hpL(i)$ are the sets of high and low criticality tasks with higher priority than task i.

5.10.5 Multicore Mixed-Criticality

Mixed-criticality scheduling extends to multicore platforms:

- **Partitioned approaches:** Assign tasks to cores based on criticality
- **Global approaches:** Allow all tasks to execute on any core
- **Semi-partitioned approaches:** Allow some tasks (typically low-criticality) to migrate
- **Fluid scheduling models:** Apply fairness principles to mixed-criticality

5.10.6 Practical Considerations

Implementing mixed-criticality systems involves practical considerations:

- **Certification:** Meeting standards like DO-178C, ISO 26262
- **Recovery strategies:** How to return to normal operation after mode changes
- **Service degradation:** Graceful degradation of low-criticality tasks
- **Runtime monitoring:** Detecting WCET violations and triggering mode changes
- **WCET estimation:** Obtaining different WCET estimates for different criticality levels

5.11 Conclusion

Real-time systems theory and practice continue to evolve to address the challenges of modern computing platforms:

- **Single-core foundations:** The fundamental principles of scheduling, response time analysis, and resource sharing protocols
- **Model extensions:** Release jitter, arbitrary deadlines, offsets, and transactions to model complex real-world systems
- **Multicore challenges:** New scheduling paradigms, resource sharing protocols, and analysis techniques
- **Mixed-criticality systems:** Integrating components with different importance and certification requirements

The future of real-time systems research includes:

- **Heterogeneous platforms:** Combining different types of processors and accelerators
- **Time-sensitive networking:** Extending real-time guarantees to distributed systems
- **Energy awareness:** Balancing real-time requirements with energy constraints
- **Safety and security:** Integrating real-time guarantees with safety and security properties

.1 Glossary of Terms

Busy period A time interval during which the processor is continuously busy executing tasks of a certain priority or higher.

Criticality The importance of a task or component, often related to safety certification requirements.

Deadline The time by which a task must complete its execution.

Feasibility The property that all tasks in a system can meet their deadlines.

Interference The delay experienced by a task due to the execution of higher-priority tasks.

Jitter Variation in the timing of a periodic event, such as task release or completion.

Laxity The slack time of a task, calculated as deadline minus remaining execution time.

Optimality The property of a scheduling algorithm that it can schedule any feasible task set.

Preemption The act of temporarily interrupting a task, with the intention of resuming it later.

Priority inversion A situation where a high-priority task is blocked by a lower-priority task.

Response time The time between the release of a task and its completion.

Schedulability The property that a particular scheduling algorithm can schedule a task set.

Utilization The fraction of processor time required by a task, calculated as execution time divided by period.

WCET Worst-Case Execution Time, the maximum time a task could take to execute.

.2 Bibliography

Bibliography

[1] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," Journal of the ACM, vol. 20, no. 1, pp. 46-61, 1973.

[2] N.C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," Technical Report YCS 164, Department of Computer Science, University of York, 1991.

[3] M. Joseph and P. Pandya, "Finding response times in a real-time system," The Computer Journal, vol. 29, no. 5, pp. 390-395, 1986.

[4] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," IEEE Transactions on Computers, vol. 39, no. 9, pp. 1175-1185, 1990.

[5] T.P. Baker, "Stack-based scheduling of realtime processes," Real-Time Systems, vol. 3, no. 1, pp. 67-99, 1991.

[6] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," Microprocessing and Microprogramming, vol. 40, no. 2-3, pp. 117-134, 1994.

[7] J.C. Palencia and M.G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in Proceedings of the 19th IEEE Real-Time Systems Symposium, pp. 26-37, 1998.

[8] R. Wilhelm et al., "The worst-case execution-time problem—overview of methods and survey of tools," ACM Transactions on Embedded Computing Systems, vol. 7, no. 3, pp. 1-53, 2008.

[9] S. Baruah and T. Baker, "Schedulability analysis of multiprocessor sporadic task systems," in Handbook of Real-Time and Embedded Systems, Chapman and Hall/CRC, 2007.

[10] S.K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," Algorithmica, vol. 15, no. 6, pp. 600-625, 1996.

[11] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in Proceedings of the 22nd Euromicro Conference on Real-Time Systems, pp. 3-13, 2010.

[12] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in Proceedings of the 32nd IEEE Real-Time Systems Symposium, pp. 104-115, 2011.

[13] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2013.

[14] A. Burns and A. Wellings, "A schedulability compatible multiprocessor resource sharing protocol - MrsP," in Proceedings of the 25th Euromicro Conference on Real-Time Systems, pp. 282-291, 2013.

[15] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in Proceedings of the 28th IEEE Real-Time Systems Symposium, pp. 239-243, 2007.

[16] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp. 145-154, 2012.

[17] A. Burns and R. Davis, "Mixed criticality systems - a review," Technical Report, Department of Computer Science, University of York, 2014.