Chapter 10
Multiprocessor, Multicore and Real-Time Scheduling

Eighth Edition
By William Stallings
Loosely coupled or distributed multiprocessor, or cluster

- consists of a collection of relatively autonomous systems, each processor having its own main memory and I/O channels

Functionally specialized processors

- there is a master, general-purpose processor; specialized processors are controlled by the master processor and provide services to it

Tightly coupled multiprocessor

- consists of a set of processors that share a common main memory and are under the integrated control of an operating system
<table>
<thead>
<tr>
<th>Grain Size</th>
<th>Description</th>
<th>Synchronization Interval (Instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine</td>
<td>Parallelism inherent in a single instruction stream.</td>
<td>&lt;20</td>
</tr>
<tr>
<td>Medium</td>
<td>Parallel processing or multitasking within a single application</td>
<td>20-200</td>
</tr>
<tr>
<td>Coarse</td>
<td>Multiprocessing of concurrent processes in a multiprogramming environment</td>
<td>200-2000</td>
</tr>
<tr>
<td>Very Coarse</td>
<td>Distributed processing across network nodes to form a single computing environment</td>
<td>2000-1M</td>
</tr>
<tr>
<td>Independent</td>
<td>Multiple unrelated processes</td>
<td>not applicable</td>
</tr>
</tbody>
</table>

Table 10.1  Synchronization Granularity and Processes
Independent Parallelism

- No explicit synchronization among processes
  - each represents a separate, independent application or job
- Typical use is in a time-sharing system

- each user is performing a particular application
- multiprocessor provides the same service as a multiprogrammed uniprocessor
- because more than one processor is available, average response time to the users will be less
Synchronization among processes, but at a very gross level

Good for concurrent processes running on a multiprogrammed uniprocessor

- can be supported on a multiprocessor with little or no change to user software
Medium-Grained Parallelism

- Single application can be effectively implemented as a collection of threads within a single process
  - programmer must explicitly specify the potential parallelism of an application
  - there needs to be a high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization

- Because the various threads of an application interact so frequently, scheduling decisions concerning one thread may affect the performance of the entire application
Fine-Grained Parallelism

- Represents a much more complex use of parallelism than is found in the use of threads
- Is a specialized and fragmented area with many different approaches
Scheduling on a multiprocessor involves three interrelated issues:

- The approach taken will depend on the degree of granularity of applications and the number of processors available.
- Actual dispatching of a process.
- Use of multiprogramming on individual processors.
- Assignment of processes to processors.
A disadvantage of static assignment is that one processor can be idle, with an empty queue, while another processor has a backlog.

- to prevent this situation, a common queue can be used
- another option is dynamic load balancing.
Both dynamic and static methods require some way of assigning a process to a processor

Approaches:
- Master/Slave
- Peer
Master/Slave Architecture

- Key kernel functions always run on a particular processor
- Master is responsible for scheduling
- Slave sends service request to the master
- Is simple and requires little enhancement to a uniprocessor multiprogramming operating system
- Conflict resolution is simplified because one processor has control of all memory and I/O resources

Disadvantages:

- failure of master brings down whole system
- master can become a performance bottleneck
Peer Architecture

- Kernel can execute on any processor
- Each processor does self-scheduling from the pool of available processes

Complicates the operating system

- operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue
Usually processes are not dedicated to processors

A single queue is used for all processors
- if some sort of priority scheme is used, there are multiple queues based on priority

System is viewed as being a multi-server queuing architecture
Figure 10.1  Comparison of Scheduling Performance for One and Two Processors
Thread execution is separated from the rest of the definition of a process

An application can be a set of threads that cooperate and execute concurrently in the same address space

On a uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing

In a multiprocessor system threads can be used to exploit true parallelism in an application

Dramatic gains in performance are possible in multi-processor systems

Small differences in thread management and scheduling can have an impact on applications that require significant interaction among threads
Four approaches for multiprocessor thread scheduling and processor assignment are:

- **Load Sharing**: processes are not assigned to a particular processor
- **Gang Scheduling**: a set of related threads scheduled to run on a set of processors at the same time, on a one-to-one basis
- **Dedicated Processor Assignment**: provides implicit scheduling defined by the assignment of threads to processors
- **Dynamic Scheduling**: the number of threads in a process can be altered during the course of execution
Simplest approach and carries over most directly from a uniprocessor environment

Advantages:
- load is distributed evenly across the processors
- no centralized scheduler required
- the global queue can be organized and accessed using any of the schemes discussed in Chapter 9

Versions of load sharing:
- first-come-first-served
- smallest number of threads first
- preemptive smallest number of threads first
Disadvantages of Load Sharing

- Central queue occupies a region of memory that must be accessed in a manner that enforces mutual exclusion
  - can lead to bottlenecks
- Preemptive threads are unlikely to resume execution on the same processor
  - caching can become less efficient
- If all threads are treated as a common pool of threads, it is unlikely that all of the threads of a program will gain access to processors at the same time
  - the process switches involved may seriously compromise performance
Gang Scheduling

- Simultaneous scheduling of the threads that make up a single process

<table>
<thead>
<tr>
<th>Benefits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• synchronization blocking may be reduced, less process switching may be necessary, and performance will increase</td>
</tr>
<tr>
<td>• scheduling overhead may be reduced</td>
</tr>
</tbody>
</table>

- Useful for medium-grained to fine-grained parallel applications whose performance severely degrades when any part of the application is not running while other parts are ready to run

- Also beneficial for any parallel application
Figure 10.2 Example of Scheduling Groups with Four and One Threads [FEIT90b]
When an application is scheduled, each of its threads is assigned to a processor that remains dedicated to that thread until the application runs to completion.

If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread’s processor remains idle.

- There is no multiprogramming of processors.

Defense of this strategy:
- In a highly parallel system, with tens or hundreds of processors, processor utilization is no longer so important as a metric for effectiveness or performance.
- The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program.
<table>
<thead>
<tr>
<th>Number of threads per application</th>
<th>Matrix multiplication</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td>4</td>
<td>3.8</td>
<td>3.8</td>
</tr>
<tr>
<td>8</td>
<td>6.5</td>
<td>6.1</td>
</tr>
<tr>
<td>12</td>
<td>5.2</td>
<td>5.1</td>
</tr>
<tr>
<td>16</td>
<td>3.9</td>
<td>3.8</td>
</tr>
<tr>
<td>20</td>
<td>3.3</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>2.8</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Table 10.2 Application Speedup as a Function of Number of Threads
For some applications it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically.

- this would allow the operating system to adjust the load to improve utilization.

Both the operating system and the application are involved in making scheduling decisions.

- The scheduling responsibility of the operating system is primarily limited to processor allocation.

This approach is superior to gang scheduling or dedicated processor assignment for applications that can take advantage of it.
Figure 10.3  AMD Bulldozer Architecture
Cooperative resource sharing

- Multiple threads access the same set of main memory locations
- Examples:
  - applications that are multithreaded
  - producer-consumer thread interaction

Resource contention

- Threads, if operating on adjacent cores, compete for cache memory locations
- If more of the cache is dynamically allocated to one thread, the competing thread necessarily has less cache space available and thus suffers performance degradation
- Objective of contention-aware scheduling is to allocate threads to cores to maximize the effectiveness of the shared cache memory and minimize the need for off-chip memory accesses
The operating system, and in particular the scheduler, is perhaps the most important component.

Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

Tasks or processes attempt to control or react to events that take place in the outside world.

These events occur in “real time” and tasks must be able to keep up with them.

Examples:
- control of laboratory experiments
- process control in industrial plants
- robotics
- air traffic control
- telecommunications
- military command and control systems
Hard real-time task
- one that must meet its deadline
- otherwise it will cause unacceptable damage or a fatal error to the system

Soft real-time task
- has an associated deadline that is desirable but not mandatory
- it still makes sense to schedule and complete the task even if it has passed its deadline
Periodic and Aperiodic Tasks

- **Periodic tasks**
  - requirement may be stated as:
    - once per period $T$
    - exactly $T$ units apart

- **Aperiodic tasks**
  - has a deadline by which it must finish or start
  - may have a constraint on both start and finish time
Real-time operating systems have requirements in five general areas:

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation
Concerned with how long an operating system delays before acknowledging an interrupt

Operations are performed at fixed, predetermined times or within predetermined time intervals
- when multiple processes are competing for resources and processor time, no system will be fully deterministic

The extent to which an operating system can deterministically satisfy requests depends on:

- the speed with which it can respond to interrupts
- whether the system has sufficient capacity to handle all requests within the required time
Together with determinism make up the response time to external events
- critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system

Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt

Responsiveness includes:
- amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR)
- amount of time required to perform the ISR
- effect of interrupt nesting
User Control

- Generally much broader in a real-time operating system than in ordinary operating systems
- It is essential to allow the user fine-grained control over task priority
- User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
- May allow user to specify such characteristics as:
  - paging or process swapping
  - what processes must always be resident in main memory
  - what disk transfer algorithms are to be used
  - what rights the processes in various priority bands have
Reliability

- More important for real-time systems than non-real time systems

- Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences such as:
  - financial loss
  - major equipment damage
  - loss of life
Fail-Soft Operation

- A characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible

- Important aspect is stability
  - A real-time system is stable if the system will meet the deadlines of its most critical, highest-priority tasks even if some less critical task deadlines are not always met
Figure 10.4 Scheduling of Real-Time Process
Scheduling approaches depend on:

- whether a system performs schedulability analysis;
- whether the result of the analysis itself produces a scheduler plan according to which tasks are dispatched at run time;
- if it does, whether it is done statically or dynamically.
Classes of Real-Time Scheduling Algorithms

Static table-driven approaches
• performs a static analysis of feasible schedules of dispatching
• result is a schedule that determines, at run time, when a task must begin execution

Static priority-driven preemptive approaches
• a static analysis is performed but no schedule is drawn up
• analysis is used to assign priorities to tasks so that a traditional priority-driven preemptive scheduler can be used

Dynamic planning-based approaches
• feasibility is determined at run time rather than offline prior to the start of execution
• one result of the analysis is a schedule or plan that is used to decide when to dispatch this task

Dynamic best effort approaches
• no feasibility analysis is performed
• system tries to meet all deadlines and aborts any started process whose deadline is missed
Real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible and emphasize rapid interrupt handling and task dispatching.

Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times.

Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time.
### Information Used for Deadline Scheduling

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready time</td>
<td>• time task becomes ready for execution</td>
</tr>
<tr>
<td>Starting deadline</td>
<td>• time task must begin</td>
</tr>
<tr>
<td>Completion deadline</td>
<td>• time task must be completed</td>
</tr>
<tr>
<td>Processing time</td>
<td>• time required to execute the task to completion</td>
</tr>
<tr>
<td>Resource requirements</td>
<td>• resources required by the task while it is executing</td>
</tr>
<tr>
<td>Priority</td>
<td>• measures relative importance of the task</td>
</tr>
<tr>
<td>Subtask scheduler</td>
<td>• a task may be decomposed into a mandatory subtask and an optional subtask</td>
</tr>
</tbody>
</table>
### Table 10.3
Execution Profile of Two Periodic Tasks

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Ending Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1)</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>A(2)</td>
<td>20</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>A(3)</td>
<td>40</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>A(4)</td>
<td>60</td>
<td>10</td>
<td>80</td>
</tr>
<tr>
<td>A(5)</td>
<td>80</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>B(1)</td>
<td>0</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>B(2)</td>
<td>50</td>
<td>25</td>
<td>100</td>
</tr>
</tbody>
</table>
Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)
Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines
<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Starting Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>110</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>40</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>D</td>
<td>50</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>E</td>
<td>60</td>
<td>20</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 10.4
Execution Profile of Five Aperiodic Tasks
Figure 10.7  A Task Set with RMS
Processing
 idle
 Processing
 task P execution time $C$
 task P period $T$

Figure 10.8 Periodic Task Timing Diagram
Table 10.5 Value of the RMS Upper Bound

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n(2^{1/n} - 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>0.828</td>
</tr>
<tr>
<td>3</td>
<td>0.779</td>
</tr>
<tr>
<td>4</td>
<td>0.756</td>
</tr>
<tr>
<td>5</td>
<td>0.743</td>
</tr>
<tr>
<td>6</td>
<td>0.734</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\ln 2 \approx 0.693$</td>
</tr>
</tbody>
</table>
Can occur in any priority-based preemptive scheduling scheme

Particularly relevant in the context of real-time scheduling

Best-known instance involved the Mars Pathfinder mission

Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

**Unbounded Priority Inversion**

- the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks
Unbounded Priority Inversion

(a) Unbounded priority inversion
Priority Inheritance

Figure 10.9  Priority Inversion

(a) Unbounded priority inversion

(b) Use of priority inheritance
The three classes are:

- `SCHED_FIFO`: First-in-first-out real-time threads
- `SCHED_RR`: Round-robin real-time threads
- `SCHED_OTHER`: Other, non-real-time threads

Within each class multiple priorities may be used
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>minimum</td>
</tr>
<tr>
<td>B</td>
<td>middle</td>
</tr>
<tr>
<td>C</td>
<td>middle</td>
</tr>
<tr>
<td>D</td>
<td>maximum</td>
</tr>
</tbody>
</table>

(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

Figure 10.10  Example of Linux Real-Time Scheduling
Non-Real-Time Scheduling

- The Linux 2.4 scheduler for the SCHED_OTHER class did not scale well with increasing number of processors and processes.

- Linux 2.6 uses a new priority scheduler known as the O(1) scheduler.

- Time to select the appropriate process and assign it to a processor is constant regardless of the load on the system or number of processors.

- Kernel maintains two scheduling data structures for each processor in the system.
Figure 10.11  Linux Scheduling Data Structures for Each Processor

140-bit priority array for active queues

140-bit priority array for expired queues

Active Queues:
140 queues by priority;
each queue contains ready tasks for that priority

Expired Queues:
140 queues by priority;
each queue contains ready tasks with expired time slices for that priority

bit 0 (priority 0)

highest-priority non-empty active queue

bit 139 (priority 139)
A complete overhaul of the scheduling algorithm used in earlier UNIX systems

The new algorithm is designed to give:

- highest preference to real-time processes
- next-highest preference to kernel-mode processes
- lowest preference to other user-mode processes

Major modifications:

- addition of a preemptable static priority scheduler and the introduction of a set of 160 priority levels divided into three priority classes
- insertion of preemption points
<table>
<thead>
<tr>
<th>Priority Class</th>
<th>Global Value</th>
<th>Scheduling Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time</td>
<td>159</td>
<td>first</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Time-shared</td>
<td>59</td>
<td>last</td>
</tr>
<tr>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.12  SVR4 Priority Classes
SVR Priority Classes

Real time (159 – 100)
- Guaranteed to be selected to run before any kernel or time-sharing process
- Can preempt kernel and user processes

Kernel (99 – 60)
- Guaranteed to be selected to run before any time-sharing process, but must defer to real-time processes

Time-shared (59-0)
- Lowest-priority processes, intended for user applications other than real-time applications
Figure 10.13  SVR4 Dispatch Queues
Table 10.6
FreeBSD Thread Scheduling Classes

<table>
<thead>
<tr>
<th>Priority Class</th>
<th>Thread Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 63</td>
<td>Bottom-half kernel</td>
<td>Scheduled by interrupts. Can block to await a resource.</td>
</tr>
<tr>
<td>64 - 127</td>
<td>Top-half kernel</td>
<td>Runs until blocked or done. Can block to await a resource.</td>
</tr>
<tr>
<td>128 - 159</td>
<td>Real-time user</td>
<td>Allowed to run until blocked or until a higher priority thread becomes available. Preemptive scheduling.</td>
</tr>
<tr>
<td>160 - 223</td>
<td>Time-sharing user</td>
<td>Adjusts priorities based on processor usage.</td>
</tr>
<tr>
<td>224 - 255</td>
<td>Idle user</td>
<td>Only run when there are no time sharing or real-time threads to run.</td>
</tr>
</tbody>
</table>

Note: Lower number corresponds to higher priority
FreeBSD scheduler was designed to provide effective scheduling for a SMP or multicore system.

Design goals:

- address the need for processor affinity in SMP and multicore systems

  - processor affinity – a scheduler that only migrates a thread when necessary to avoid having an idle processor

- provide better support for multithreading on multicore systems

- improve the performance of the scheduling algorithm so that it is no longer a function of the number of threads in the system
Figure 10.14 Windows Thread Dispatching Priorities
A thread is considered to be *interactive* if the ratio of its voluntary sleep time versus its runtime is below a certain threshold.

Interactivity threshold is defined in the scheduler code and is not configurable.

Threads whose sleep time exceeds their run time score in the lower half of the range of interactivity scores.

Threads whose run time exceeds their sleep time score in the upper half of the range of interactivity scores.
- **Processor affinity** is when a Ready thread is scheduled onto the last processor that it ran on
  - significant because of local caches dedicated to a single processor

FreeBSD scheduler supports two mechanisms for thread migration to balance load:

- **Pull Mechanism**
  - an idle processor steals a thread from a nonidle processor
  - primarily useful when there is a light or sporadic load or in situations where processes are starting and exiting very frequently

- **Push Mechanism**
  - a periodic scheduler task evaluates the current load situation and evens it out
  - ensures fairness among the runnable threads
Priorities in Windows are organized into two bands or classes:

- **real time priority class**
  - all threads have a fixed priority that never changes
  - all of the active threads at a given priority level are in a round-robin queue

- **variable priority class**
  - a thread’s priority begins an initial priority value and then may be temporarily boosted during the thread’s lifetime

Each band consists of 16 priority levels

Threads requiring immediate attention are in the real-time class
  - include functions such as communications and real-time tasks
Figure 10.15 Example of Windows Priority Relationship
Windows supports multiprocessor and multicore hardware configurations

The threads of any process can run on any processor

In the absence of affinity restrictions the kernel dispatcher assigns a ready thread to the next available processor

Multiple threads from the same process can be executing simultaneously on multiple processors

Soft affinity

- used as a default by the kernel dispatcher
- the dispatcher tries to assign a ready thread to the same processor it last ran on

Hard affinity

- application restricts its thread execution only to certain processors

If a thread is ready to execute but the only available processors are not in its processor affinity set, then the thread is forced to wait, and the kernel schedules the next available thread
Summary

- Multiprocessor and multicore scheduling
  - Granularity
  - Design issues
  - Process scheduling
  - Multicore thread scheduling

- Linux scheduling
  - Real-time scheduling
  - Non-real-time scheduling

- UNIX SVR4 scheduling

- UNIX FreeBSD scheduling
  - Priority classes
  - SMP and multicore support

- Real-time scheduling
  - Background
  - Characteristics of real-time operating systems
  - Real-time scheduling
  - Deadline scheduling
  - Rate monotonic scheduling
  - Priority inversion

- Windows scheduling
  - Process and thread priorities
  - Multiprocessor scheduling