Operating Systems: Internals and Design Principles

Chapter 10 Multiprocessor, Multicore and Real-Time Scheduling

Classifications of Multiprocessor Systems

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

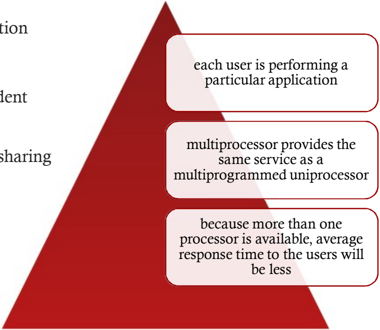
Multiprocessor and Multicore Scheduling

Grain Size	Description	Synchronization Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream.	<20
Medium	Parallel processing or multitasking within a single application	20-200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200-2000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2000-1M
Independent	Multiple unrelated processes	not applicable

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

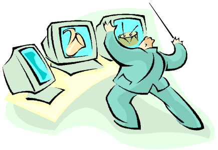


Medium-Grained Parallelism

- 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

Coarse and Very Coarse Grained Parallelism

- 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



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

Design Issues

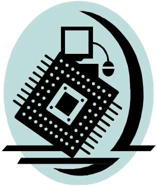
Scheduling on a multiprocessor involves three interrelated issues:

- actual dispatching of a process
- use of multiprogramming on individual processors
- assignment of processes to processors

- The approach taken will depend on the degree of granularity of applications and the number of processors available

Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Approaches:
  - Master/Slave
  - Peer



Assignment of Processes to Processors

Assuming all processors are equal, it is simplest to treat processors as a pooled resource and assign processes to processors on demand

static or dynamic needs to be determined

If a process is permanently assigned to one processor from activation until its completion, then a dedicated short-term queue is maintained for each processor

advantage is that there may be less overhead in the scheduling function

allows group or gang scheduling

- 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

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:

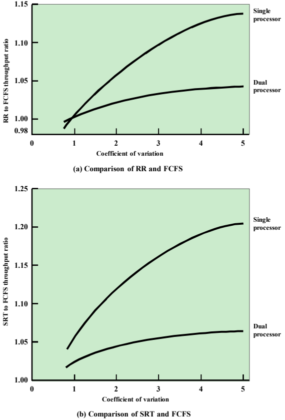
- failure of master brings down whole system
- master can become a performance bottleneck

Peer Architecture

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

Complicates the operating system

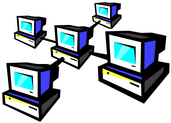
- operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue



Comparison of Scheduling Performance for One and Two Processors

Process Scheduling

- 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



Thread Scheduling

- 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

Approaches to Thread Scheduling

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

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

Load Sharing

- 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

Gang Scheduling

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

Benefits:
• synchronization blocking may be reduced, less process switching may be necessary, and performance will increase • scheduling overhead may be reduced

- 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

Uniform Division

	Group 1	Group 2
PE1		
PE2		idle
PE3		idle
PE4		idle
Time	1/2	1/2

37.5% Waste

Division by Weights

	Group 1	Group 2
PE1		
PE2		idle
PE3		idle
PE4		idle
	4/5	1/5

15% Waste

Example of Scheduling Groups with Four and One Threads [FEIT90b]

Number of threads per application	Matrix multiplication	FFT
1	1	1
2	1.8	1.8
4	3.8	3.8
8	6.5	6.1
12	5.2	5.1
16	3.9	3.8
20	3.3	3
24	2.8	2.4

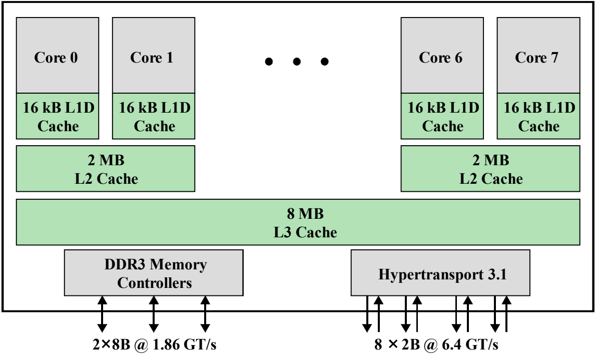
Application Speedup as a Function of Number of Threads

Dedicated Processor Assignment

- 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

Dynamic Scheduling

- 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



AMD BulldozerArchitecture

Real-Time Scheduling

Cache Sharing

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

Real-Time Systems

- The operating system, and in particular the scheduler, is perhaps the most important component

Examples:

- control of laboratory experiments
- process control in industrial plants
- robotics
- air traffic control
- telecommunications
- military command and control systems

- 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

Hard and Soft Real-Time Tasks

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

Characteristics of Real Time Systems

Real-time operating systems have requirements in five general areas:

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation

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

Determinism

- 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

Responsiveness

- 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

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

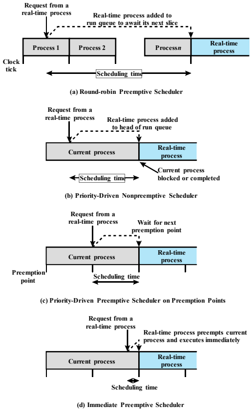
User Control

- 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

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



Scheduling of Real-Time Process

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

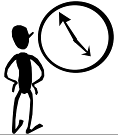
Dynamic best effort approaches

- no feasibility analysis is performed
- system tries to meet all deadlines and aborts any started process whose deadline is missed

Real-Time Scheduling

Scheduling approaches depend on:

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

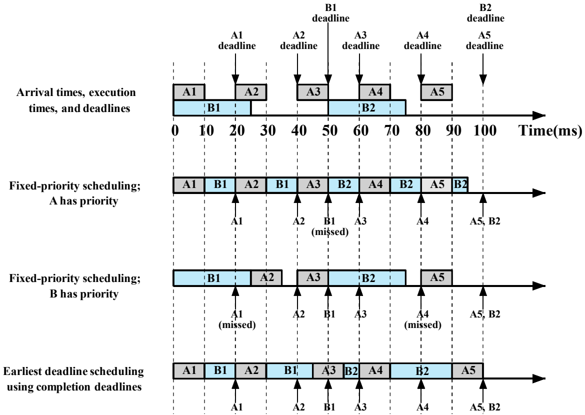


Deadline Scheduling

- 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

Ready time	• time task becomes ready for execution
Starting deadline	• time task must begin
Completion deadline	• time task must be completed
Processing time	• time required to execute the task to completion
Resource requirements	• resources required by the task while it is executing
Priority	• measures relative importance of the task
Subtask scheduler	• a task may be decomposed into a mandatory subtask and an optional subtask



Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on previous Table)

Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•

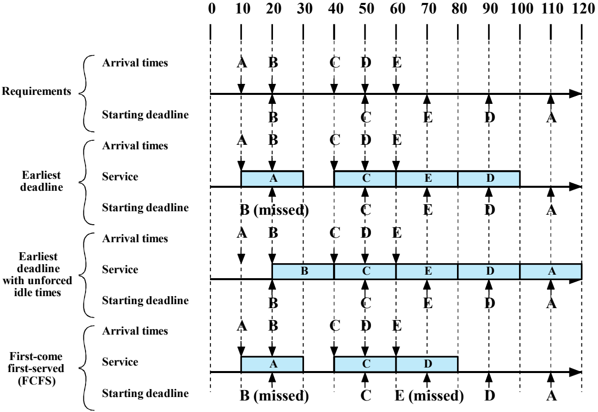
Processor Utilization: are both periodic tasks schedulable in the long run?

Execution Profile of Five Aperiodic Tasks

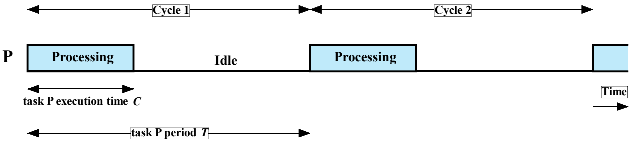
Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

How to schedule these tasks such that start deadlines are met (or alternatively, the fewest deadlines are missed)?

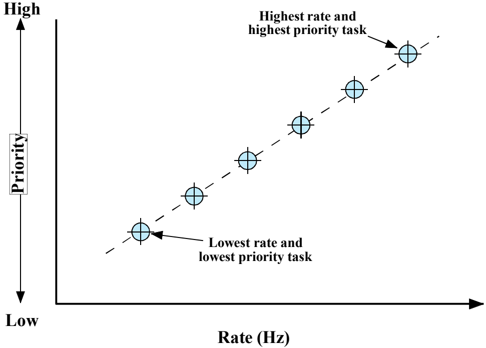
- Using pre-emption?
- FCFS?
- Earliest deadline?
- Others?



Scheduling of Aperiodic Real-time Tasks with Starting Deadlines



Periodic Task Timing Diagram

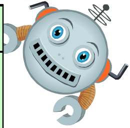


A Task Set with RMS

Value of the RMS Upper Bound

If rate sum less than or equal upper bound, tasks are schedulable

n	$n(2^{1/n}-1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
•	•
•	•
•	•
∞	$\ln 2 \approx 0.693$



Earliest deadline first has upper bound of 1

RMS is still popular – why?

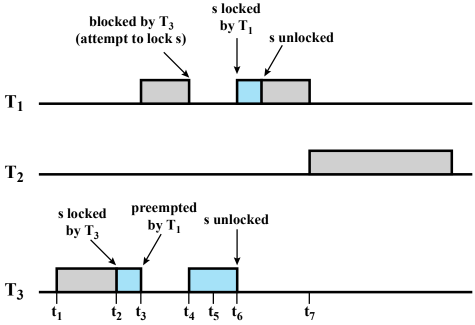
Priority Inversion

- 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

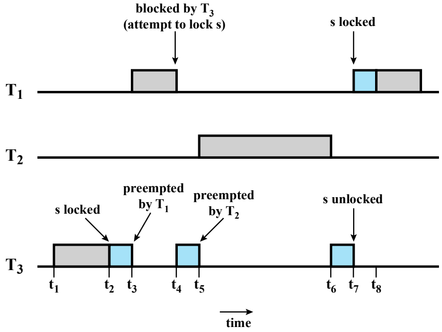
- the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks

Priority Inheritance



(b) Use of priority inheritance

Unbounded Priority Inversion



(a) Unbounded priority inversion