CSC 553 Operating Systems

Lecture 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

Synchronization Granularity and Processes

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

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

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

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

Medium-Grained Parallelism

- 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

Design Issues

- The approach taken will depend on the degree of granularity of applications and the number of processors available
- Scheduling on a multiprocessor involves three interrelated issues:
  - assignment of processes to processors
  - use of multiprogramming on individual processors
  - actual dispatching of a process

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

Assignment of Processes to Processors

- 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

Master/Slave Architecture

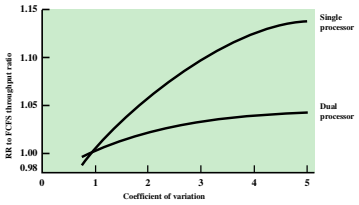
- Disadvantages:
  - Failure of master brings down whole system
  - Master can become a performance bottleneck

Peer Architecture

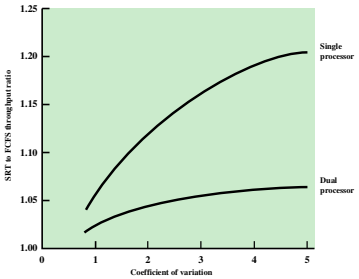
- Kernel can execute on any processor
- Each processor does self-scheduling from the pool of available processes
- Complicates the operating system
  - Operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue

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



(a) Comparison of RR and FCFS



(b) Comparison of SRT and FCFS

Figure 10.1 Comparison of Scheduling Performance for One and Two Processors

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

Thread Scheduling

- 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

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 Lecture 9

Load Sharing

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

Gang Scheduling

- 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

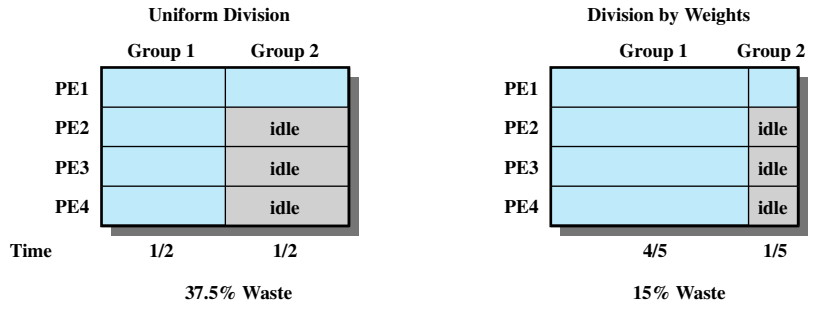


Figure 10.2 Example of Scheduling Groups with Four and One Threads [FEIT90b]

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

Dedicated Processor Assignment

- 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

Application Speedup as a Function of Number of Threads

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

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

Dynamic Scheduling

- 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

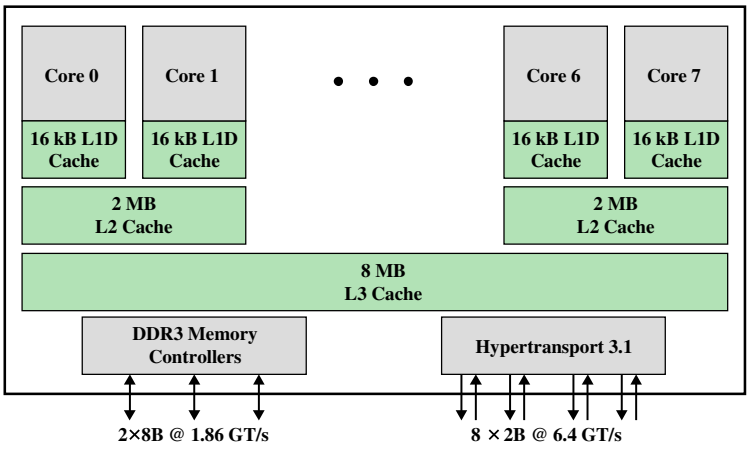


Figure 10.3 AMD Bulldozer Architecture

Cache Sharing

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

Cache Sharing

- 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

Real-Time 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

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

Characteristics of Real Time Systems

- Real-time operating systems have requirements in five general areas:
  - Determinism
  - Responsiveness
  - User control
  - Reliability
  - Fail-soft operation

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

Determinism

- 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

- 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

User Control

- 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

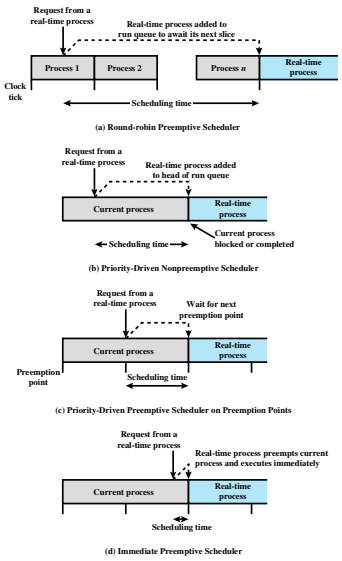


Figure 10.4 Scheduling of Real-Time Process

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

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

Classes of Real-Time Scheduling Algorithms

- 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

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

Information Used for Deadline Scheduling

- **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

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
• • •	• • •	• • •	• • •

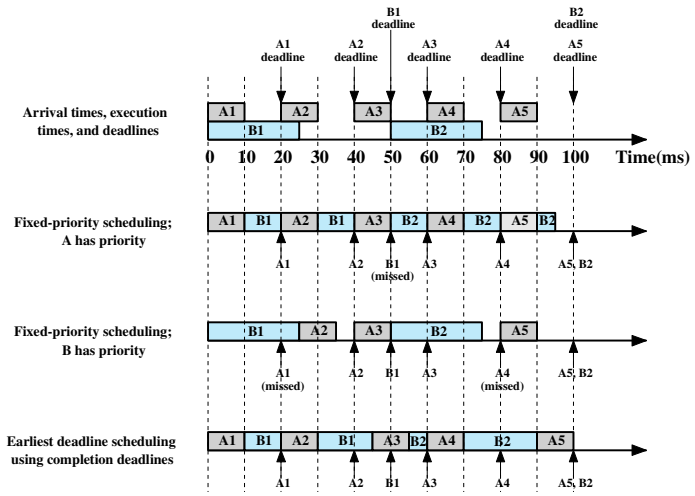


Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

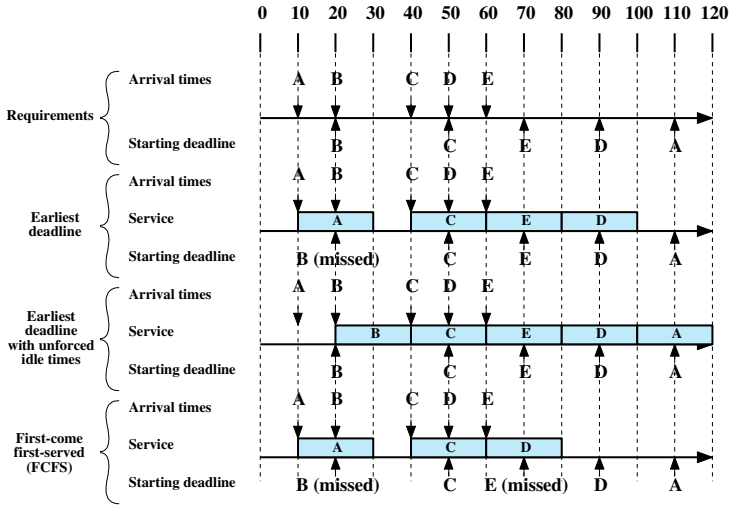


Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

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

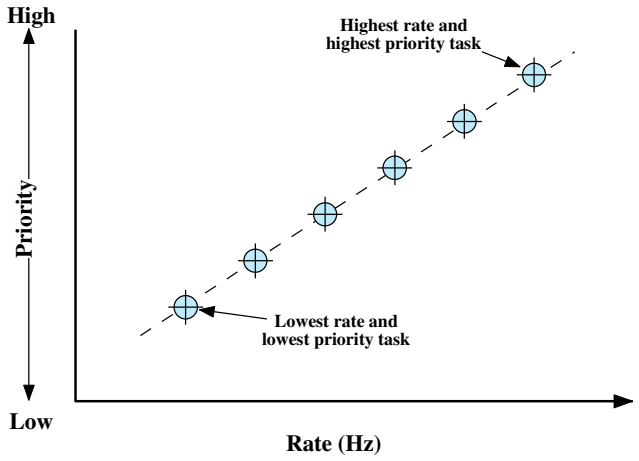


Figure 10.7 A Task Set with RMS

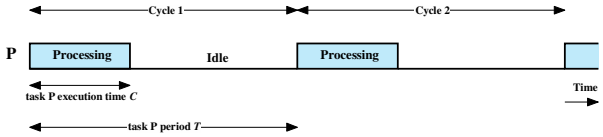


Figure 10.8 Periodic Task Timing Diagram

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 ≈ 0.693

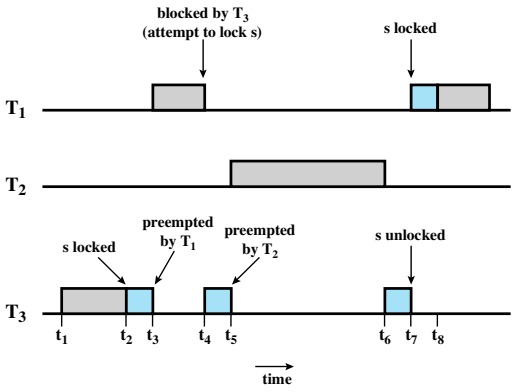
Value of the RMS Upper Bound

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

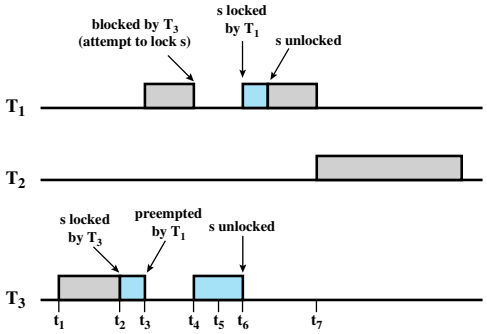
Unbounded Priority Inversion

- **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



(a) Unbounded priority inversion

Priority Inheritance



(b) Use of priority inheritance

Linux Scheduling

- 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

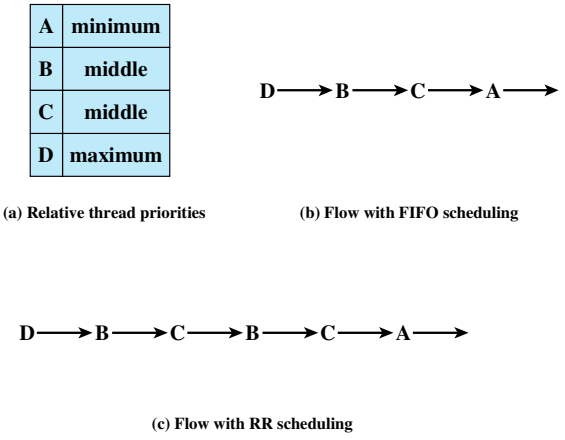


Figure 10.10 Example of Linux Real-Time Scheduling

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

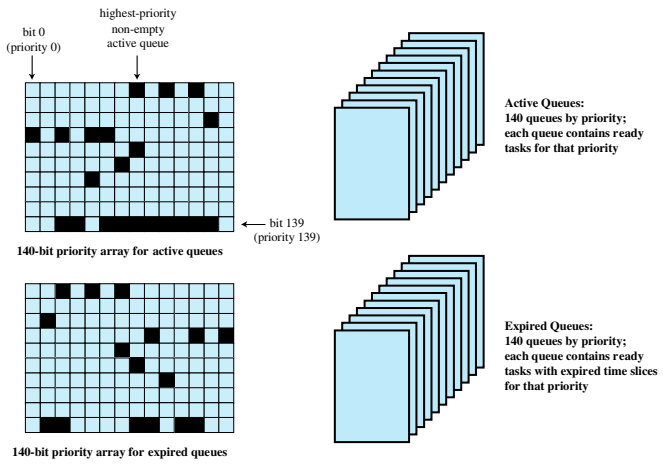


Figure 10.11 Linux Scheduling Data Structures for Each Processor