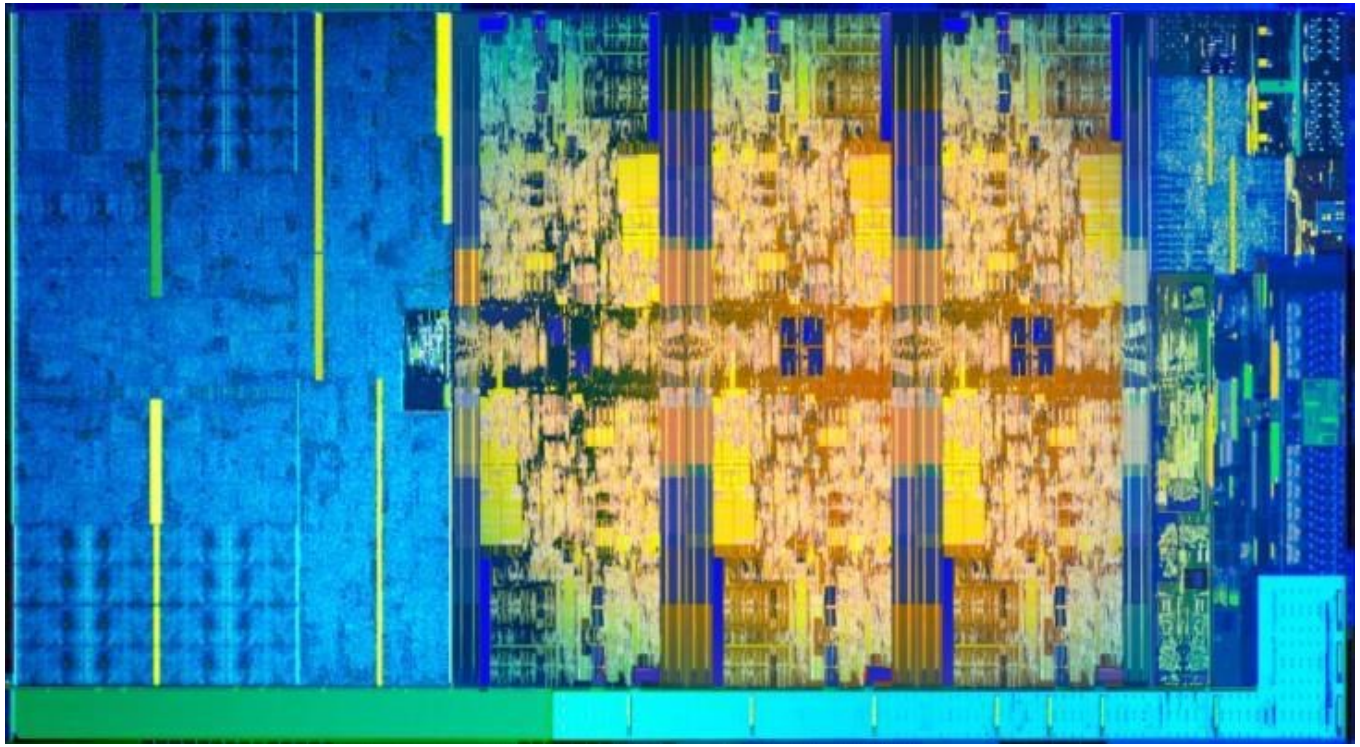
操作系统原理

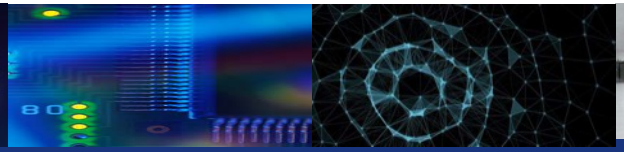
Operating Systems Principles

陈鹏飞
计算机学院
2021-05-25

第十一讲 — 多处理器和实时调度



酷睿i7-8700K



目标

- 了解线程粒度的概念；
- 讨论多处理器线程调度的主要设计问题和线程调度方法；
- 理解实时调度的需求；
- 掌握Linux、Unix SVR4和Windows 7中的调度算法；

多处理器系统分类

松耦合、分布式多处理器、集群；

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

专用处理器

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

紧耦合 多处理器

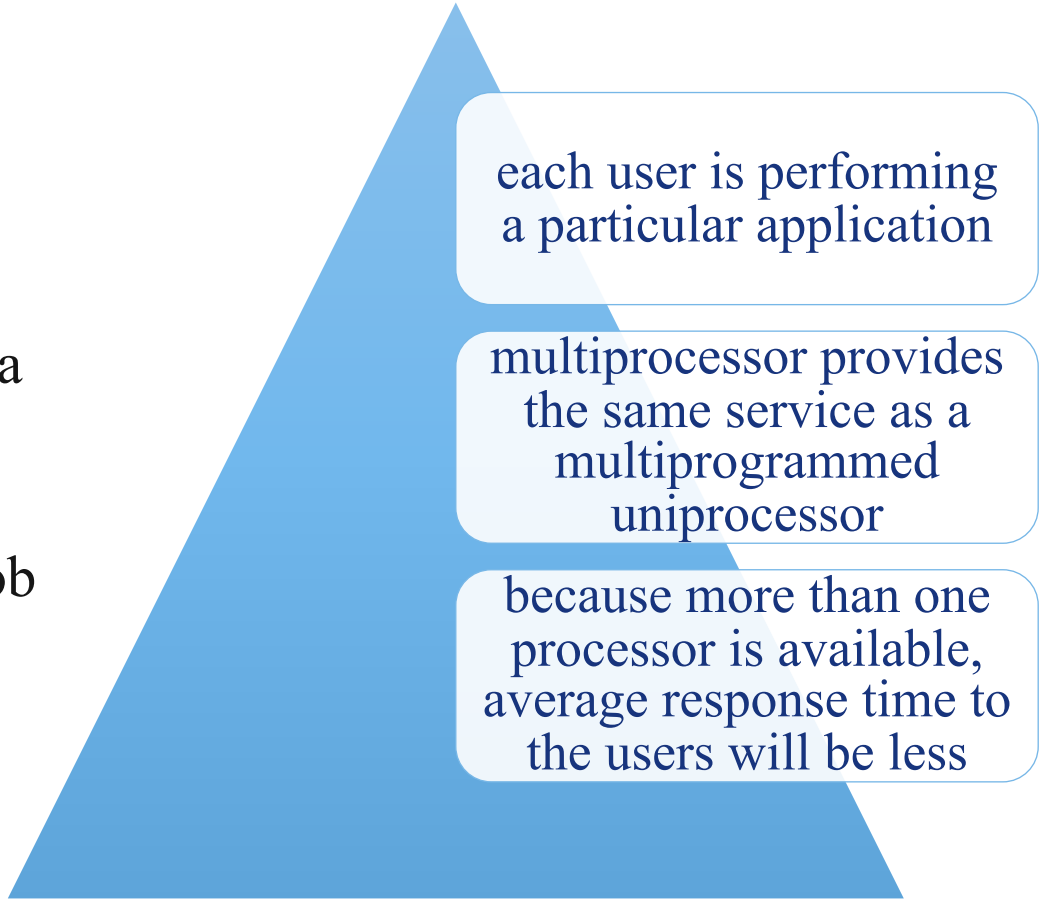
- consists of a set of processors that share a common main memory and are under the integrated control of an operating system

同步粒度和进程

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

独立并行

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



大颗粒度和极大颗粒度并行性

- 进程之间存在同步，但这种同步级别比较粗；
- 这种情况简单处理为一组运行在多道程序单处理器上的并发进程；
  - 对用户软件进行很少的改动或者不进行改动就可以提供支持；

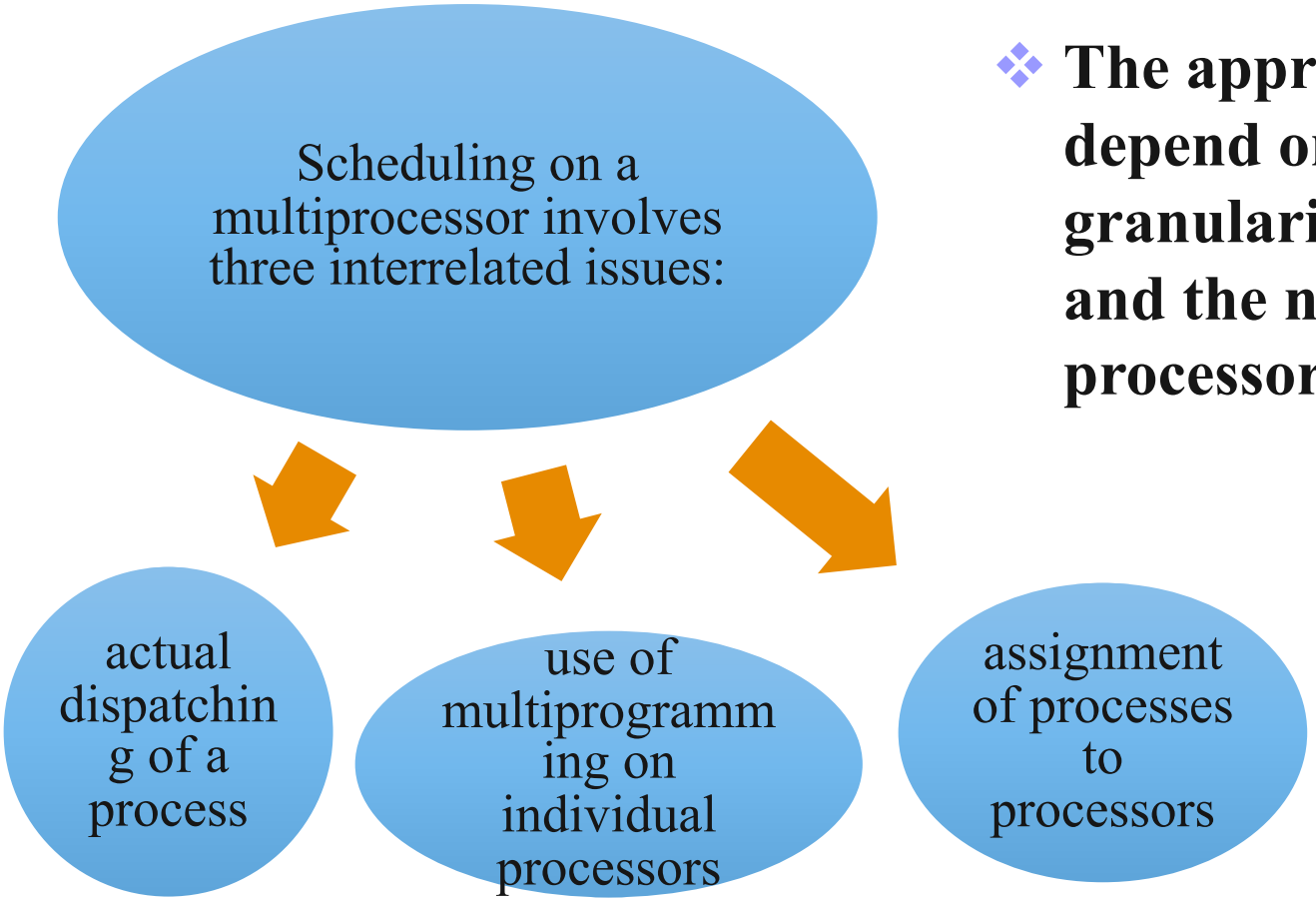
中颗粒度并行

- **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**

细颗粒度并行

- 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

设计问题



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

把进程分配到处理器上

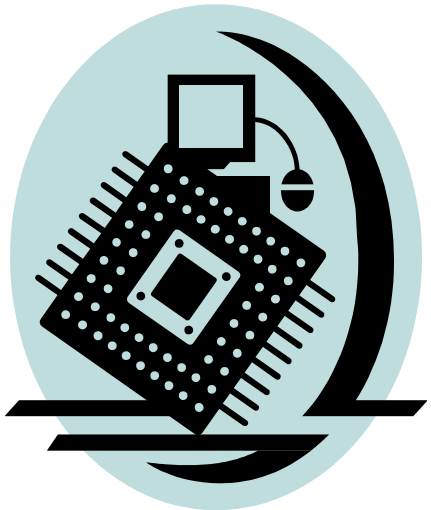
Assuming all processors are equal, it is simplest to treat processors as a pooled resource and assign processes to processors on demand → static or dynamic needs to be determined

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

进程分配给处理器

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



主从架构

- 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

对等结构架构

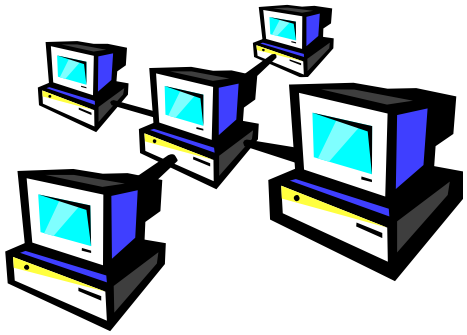
- Kernel can execute on any processor
- Each processor does self-scheduling from the pool of available processes

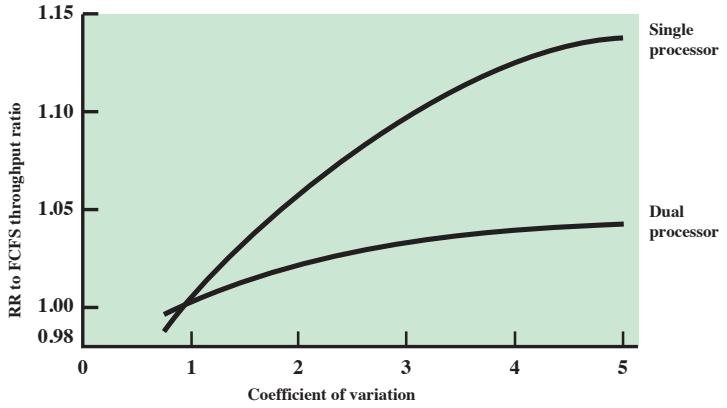
Complicates the operating system

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

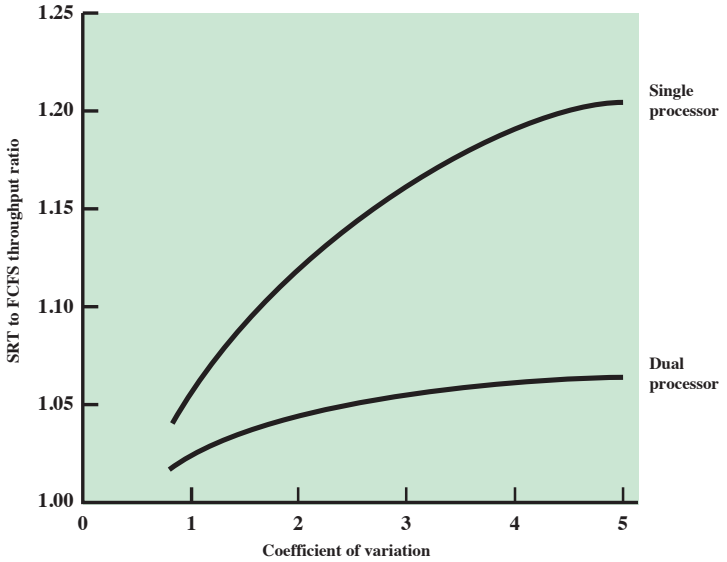
进程调度

- **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

线程调度

- 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

线程调度方法

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

负载共享

- **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

线程调度方法

- **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

组调度

- 同时调度组成一个进程的一组线程；

优点：
• 组内进程相关或大致平等时，同步阻塞会减少，且可能只需要很少的进程切换，因此性能会提高； • 调度开销可能会减少，因为一个决策可以同时影响许多处理器和进程

- 对于中粒度和细粒度的并行应用程序，组调度非常必要，因为这种应用程序的一部分准备运行，而另一部分却还未运行，它的性能会严重下降；
- 对所有的并行应用程序都有好处；
- 组调度获得人们的广泛认可；

组调度

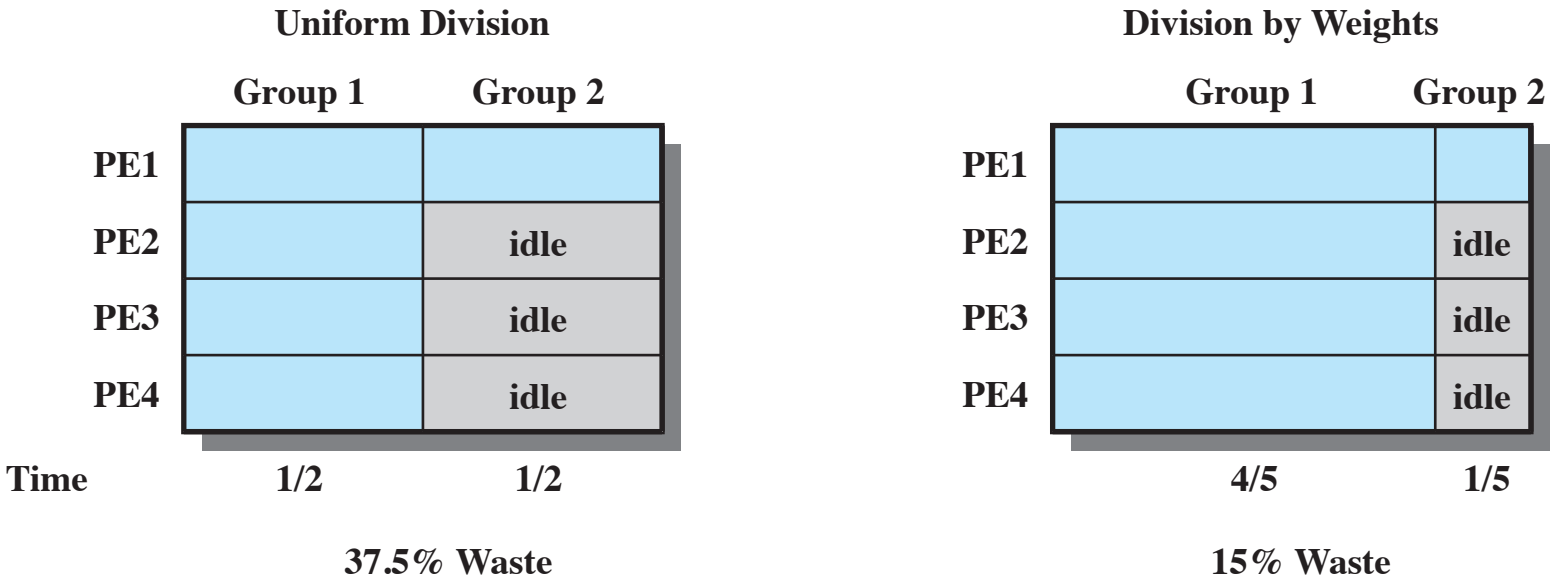


Figure 10.2 Example of Scheduling Groups with Four and One Threads [FEIT90b]

专用处理器调度

- **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

专用处理器调度

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

Table 10.2 Application Speedup as a Function of Number of Threads

动态调度

- **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

多核调度

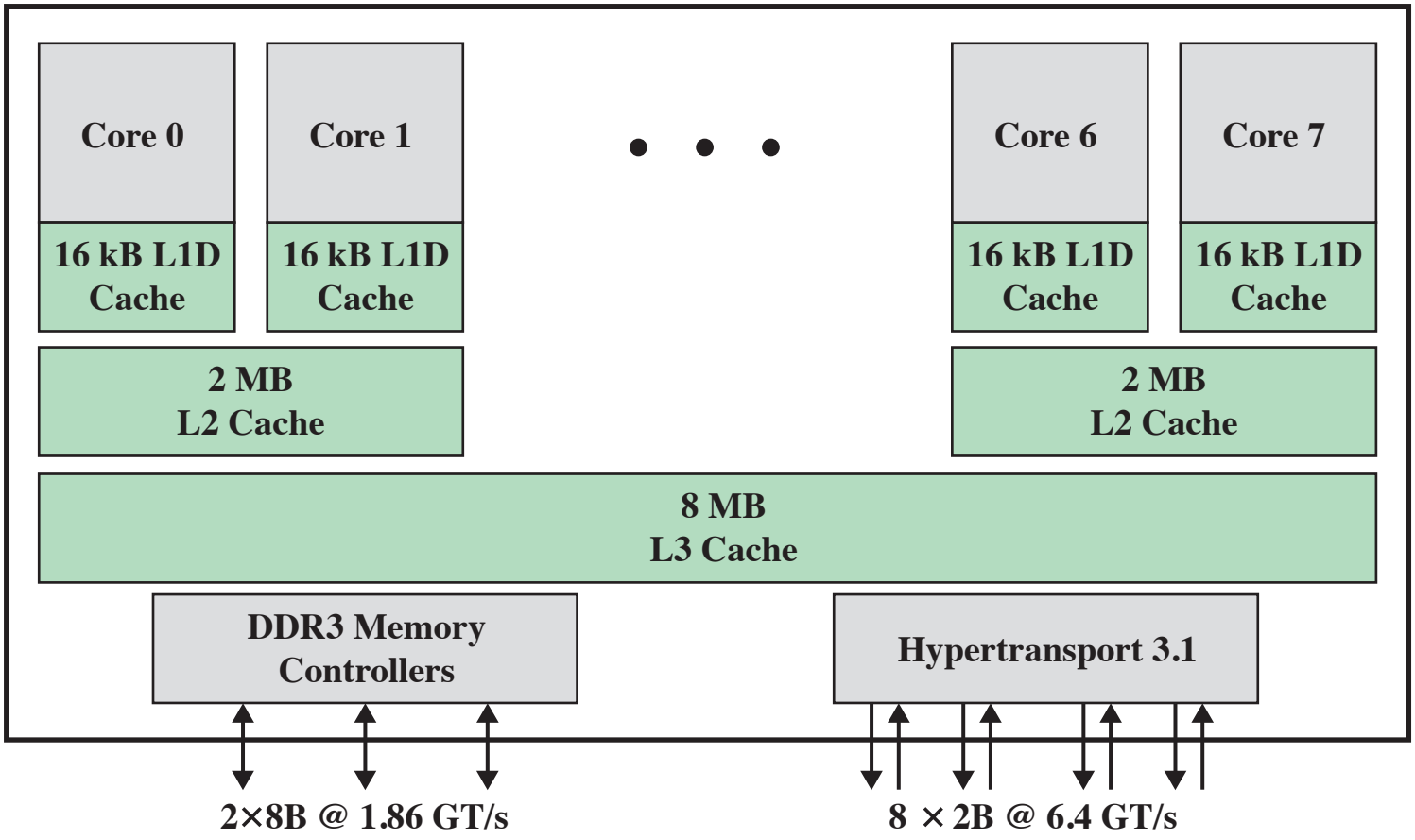


Figure 10.3 AMD Bulldozer Architecture

Cache共享

- **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**

实时系统

- 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

硬实时和软实时系统

- 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

定期和非定期的任务

- **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**

定期和非定期的任务

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

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

确定性

- **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**

可响应性(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:

- 最初处理中断并开始执行中断服务历程（ISR）所需的时间总量。若ISR的执行需要一次进程切换，则需要的延迟将比在当前进程下延迟长；
- 执行ISR所需要的时间总量，通常与硬件平台有关；
- 中断嵌套的影响。一个ISR会因另一个中断的到达而中断时，服务将被延迟；

用户控制（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:

可靠性（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

故障弱化操作（Fail-soft operation）

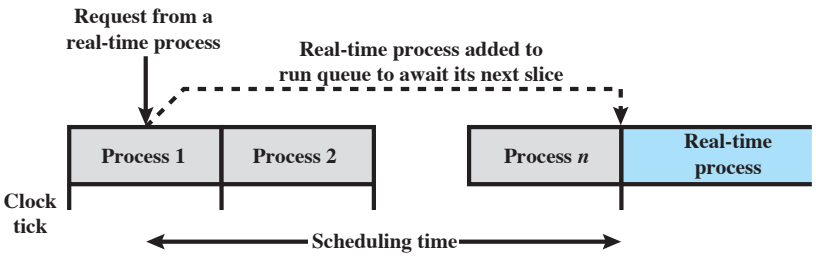
- 系统在故障时尽可能地多保存其性能和数据的能力；
- 故障弱化的一个很重要的特征是稳定性；
  - a real-time system is stable if the system will meet the deadlines of its most critical, highest-priority tasks even if some less critical task deadli nes are not always met

- 大部分实时操作系统具有以下功能：

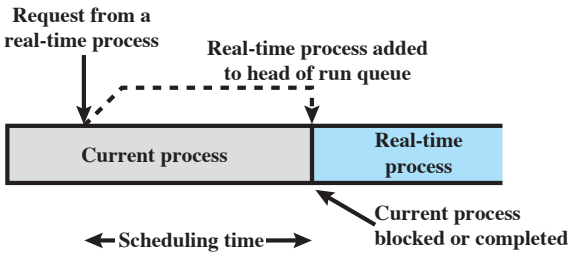
1. 与传统操作系统相比，有着更严格的使用优先级，以抢占式的调度满足实时性要求；

2. 中断延迟有界且相对较短；

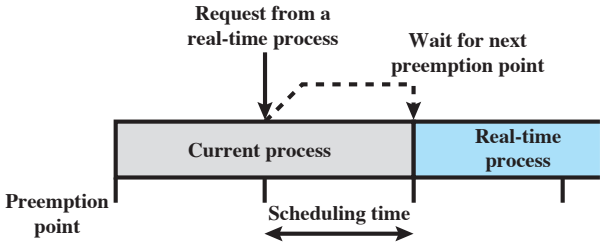
3. 与通用操作系统相比，有更精确和更可预测的时序特征；



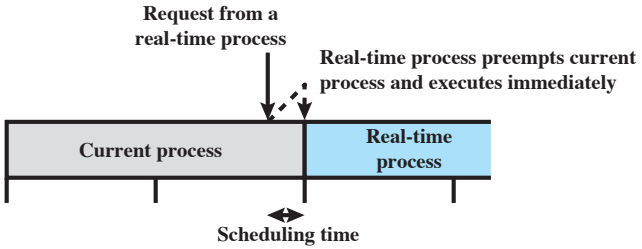
(a) Round-robin Preemptive Scheduler



(b) Priority-Driven Nonpreemptive Scheduler



(c) Priority-Driven Preemptive Scheduler on Preemption Points



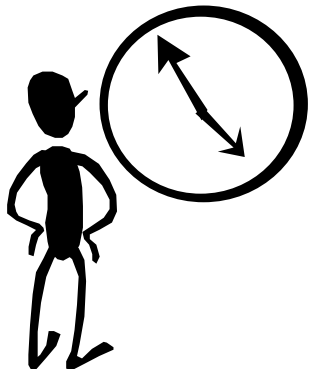
(d) Immediate Preemptive Scheduler

Figure 10.4 Scheduling of Real-Time Process

实时调度（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



实时调度算法

Static table-driven approaches

周期任务

- 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

非实时多道程序

- 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

动态调度任务

- 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

时限调度

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

限期调度

- 实时操作系统的目标是尽可能快速地启动实时任务，强调快速中断处理和任务分派。
- 事实上，实时操作系统并不关注绝对速度，而是关注在最有价值的时间完成或者启动任务，既不要太早也不要太晚；
- 按照优先级来提供工具，而并不是以最有价值的时间来完成（或启动）需求；

实时调度所利用的额外信息

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

两个周期任务

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
• • •	• • •	• • •	• • •

调度算法

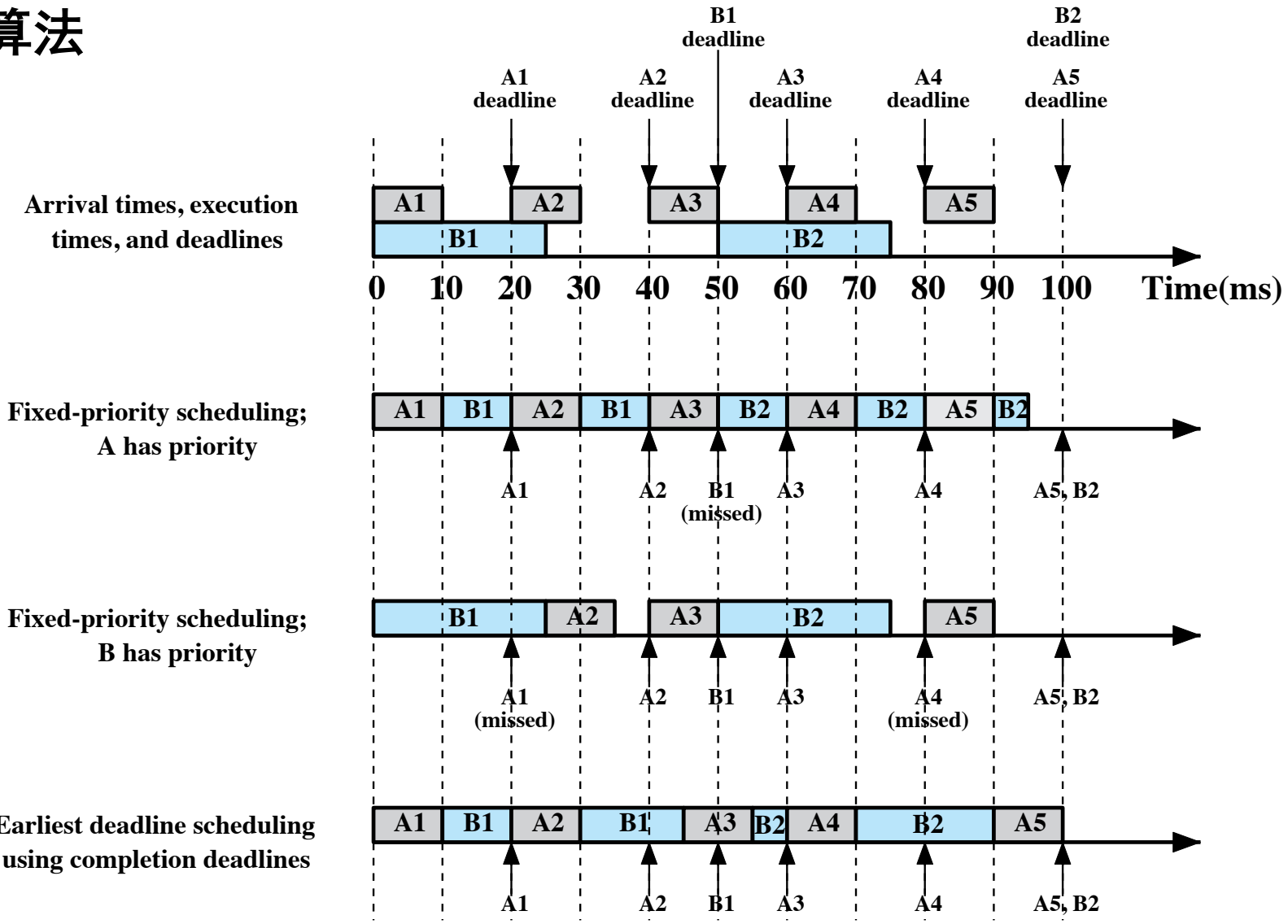


Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

调度算法

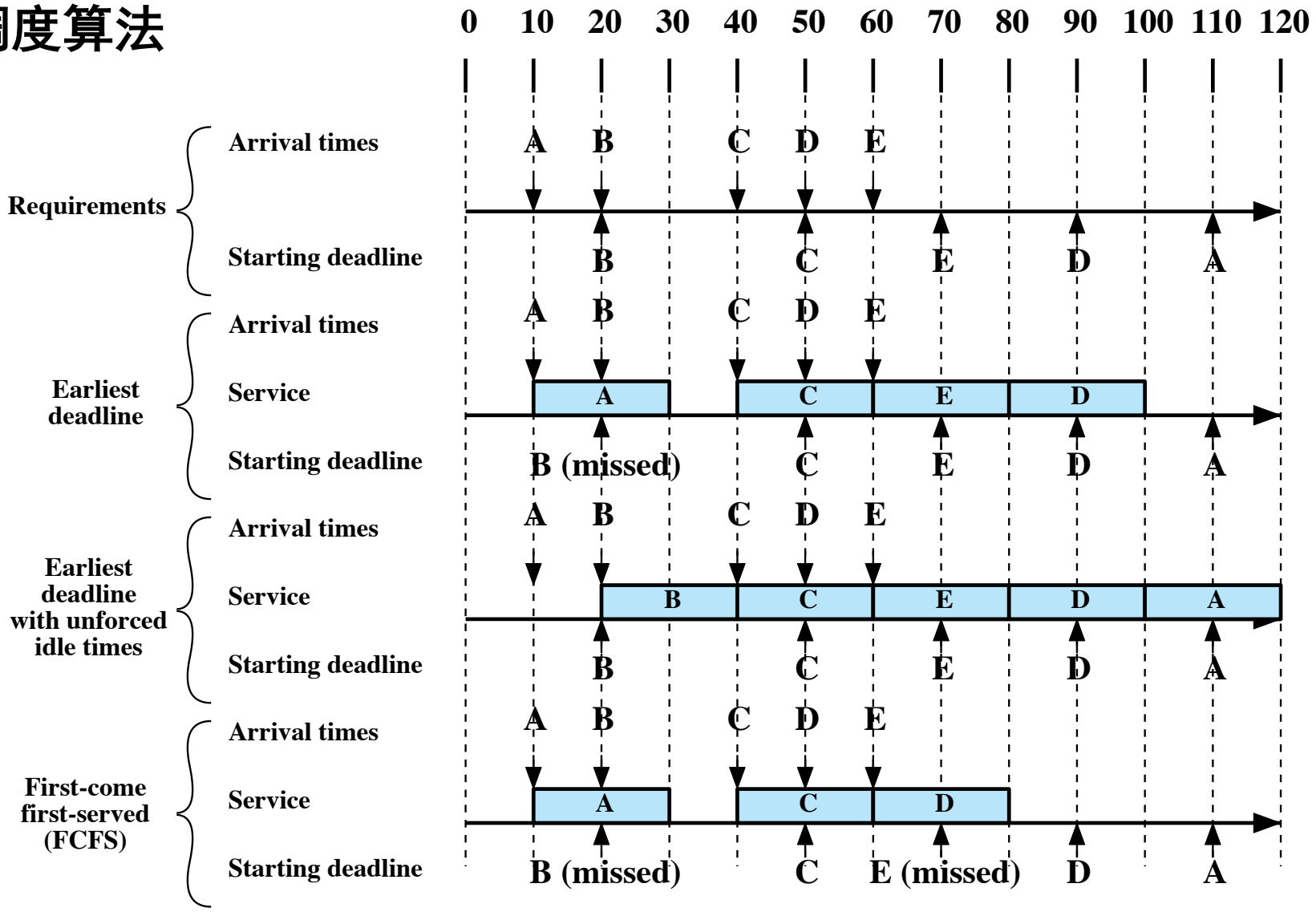
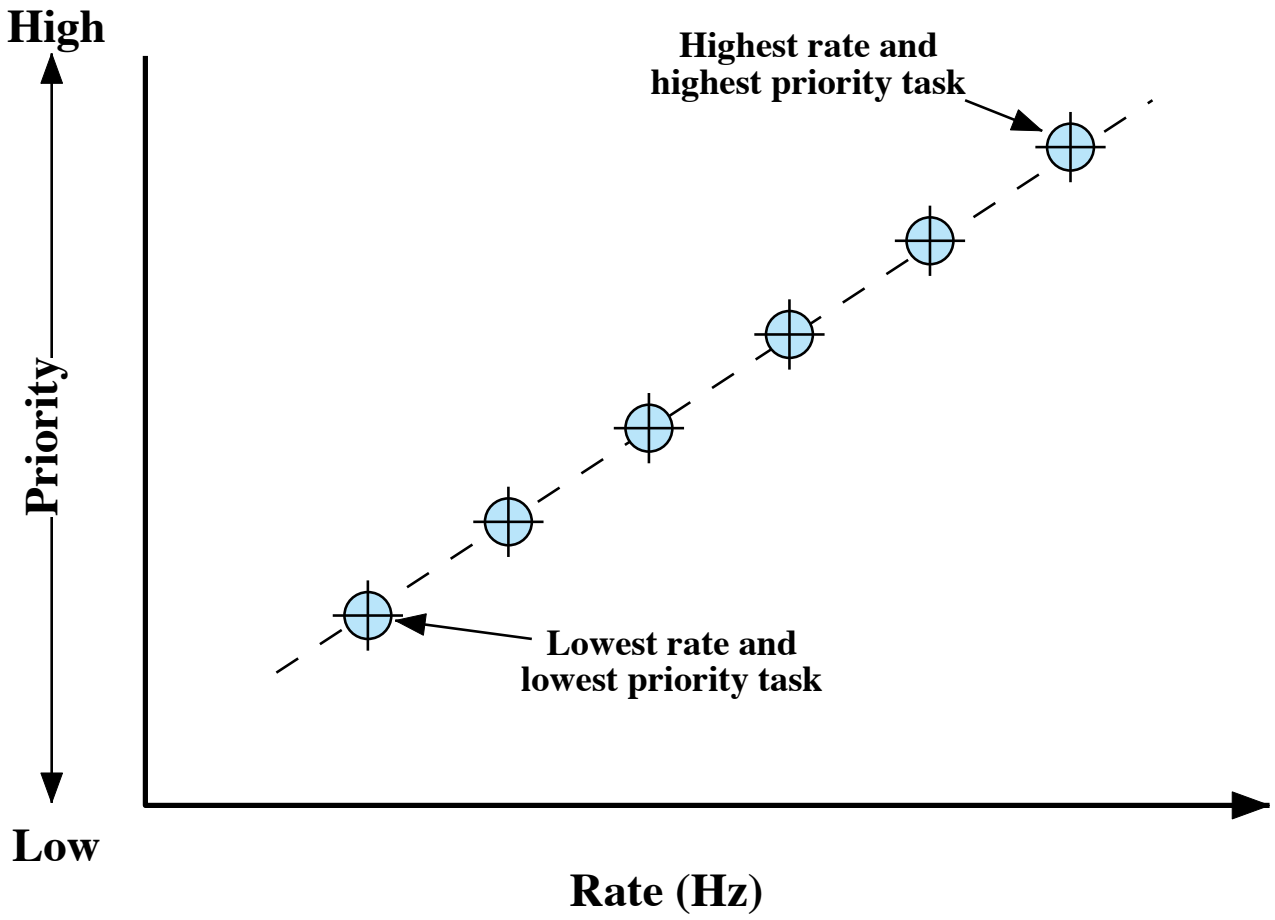


Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

5个非周期任务的执行过程

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

速率单调调度


High
Priority
Low
Highest rate and
highest priority task
Lowest rate and
lowest priority task
Rate (Hz)

周期性任务时序图

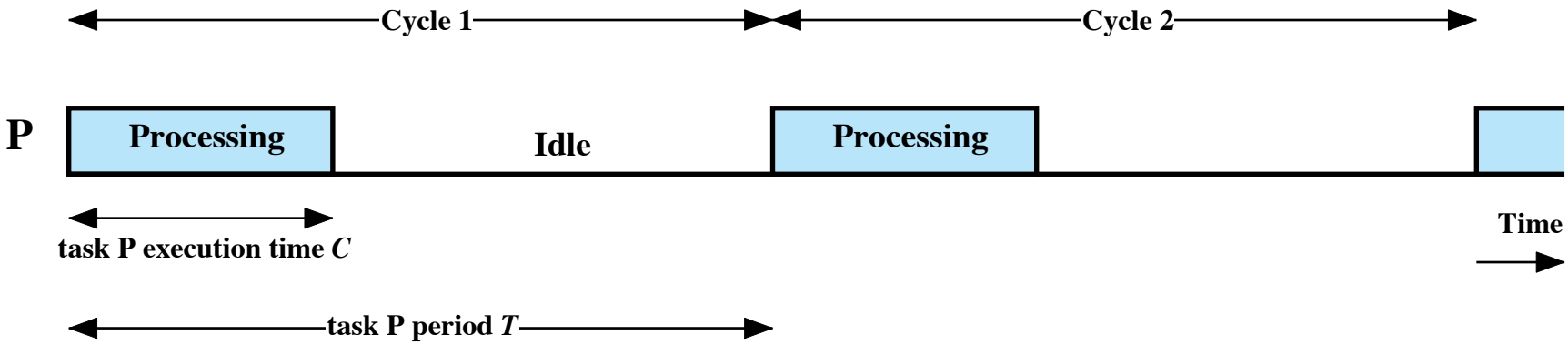
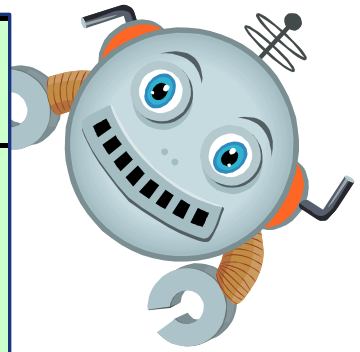


Figure 10.8 Periodic Task Timing Diagram

Table 10.5

Value of the RMS Upper Bound

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



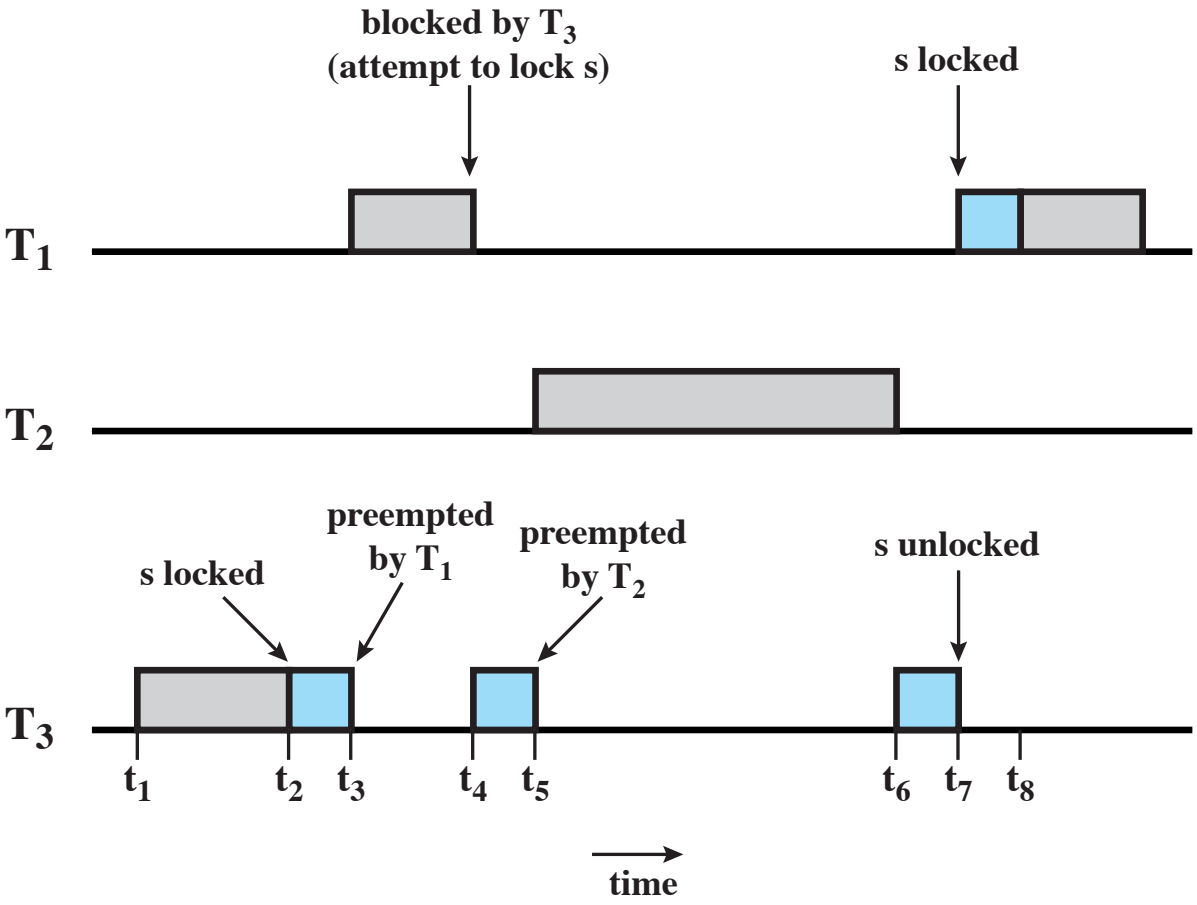
优先级翻转

- 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

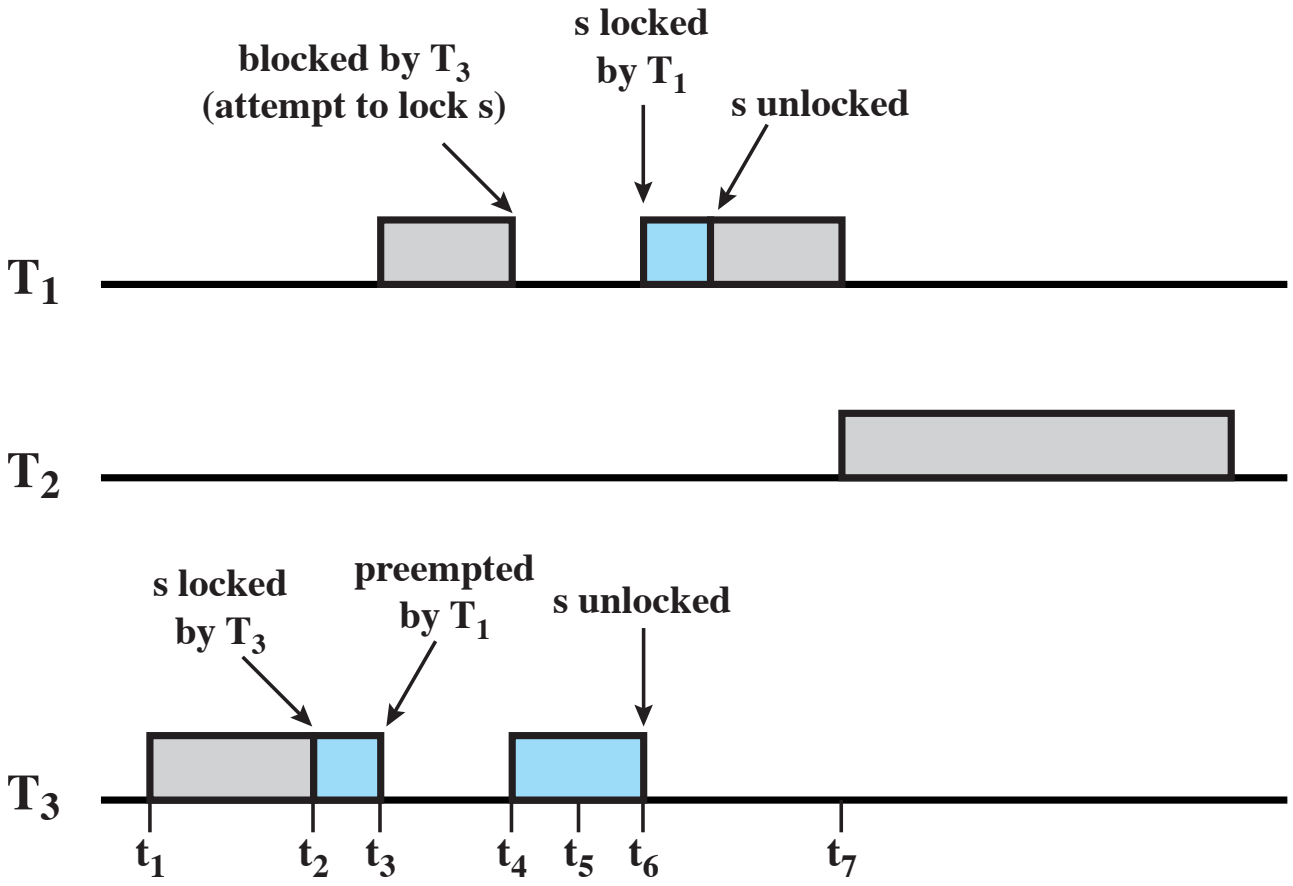
- the duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks

无边界优先级翻转



(a) Unbounded priority inversion

优先级继承



(b) Use of priority inheritance

Linux调度

- **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**

Linux调度

A	minimum
B	middle
C	middle
D	maximum

(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

Figure 10.10 Example of Linux Real-Time Scheduling

非实时调度

- **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 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**

非实时调度

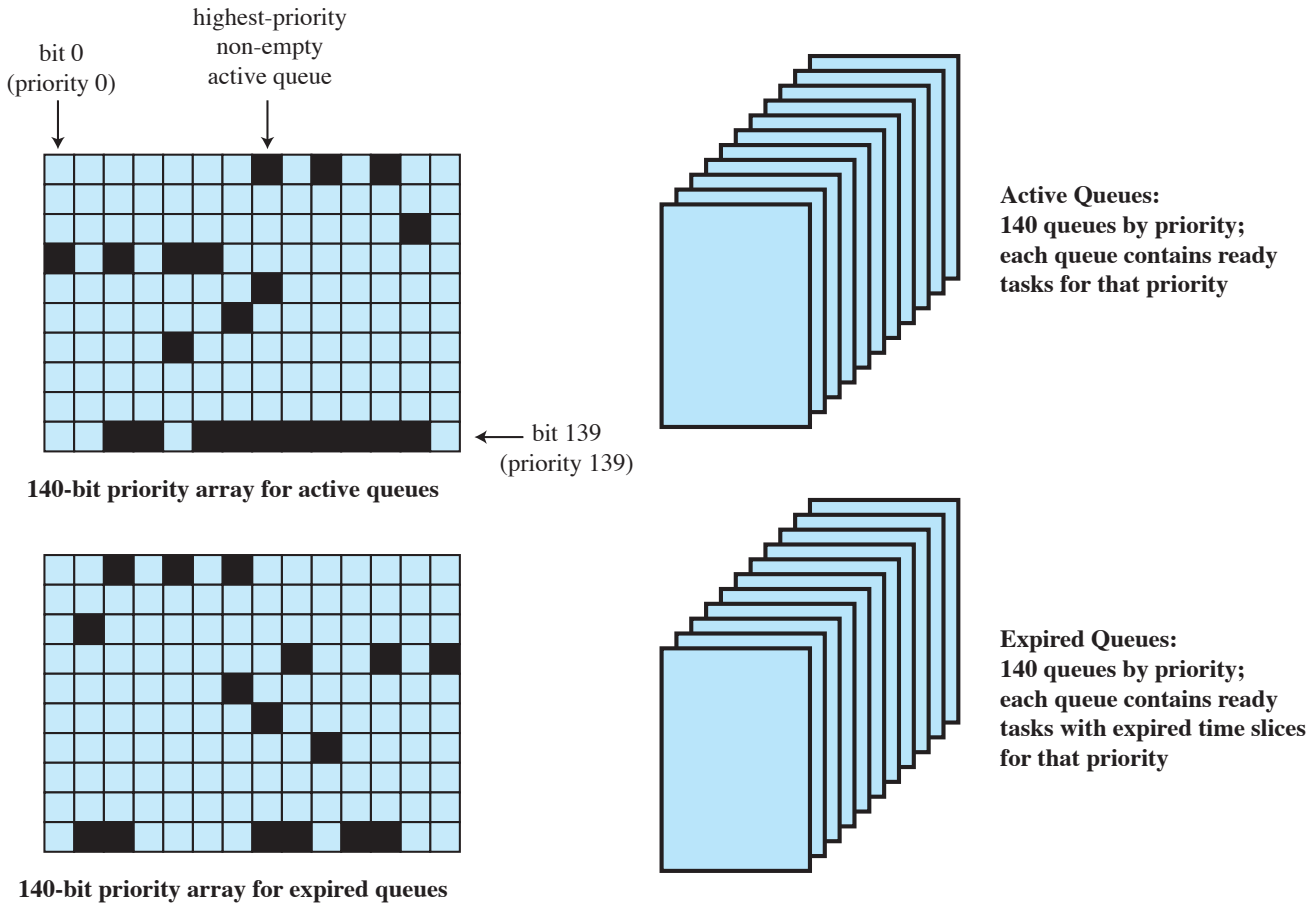


Figure 10.11 Linux Scheduling Data Structures for Each Processor

UNIX SVR4 调度

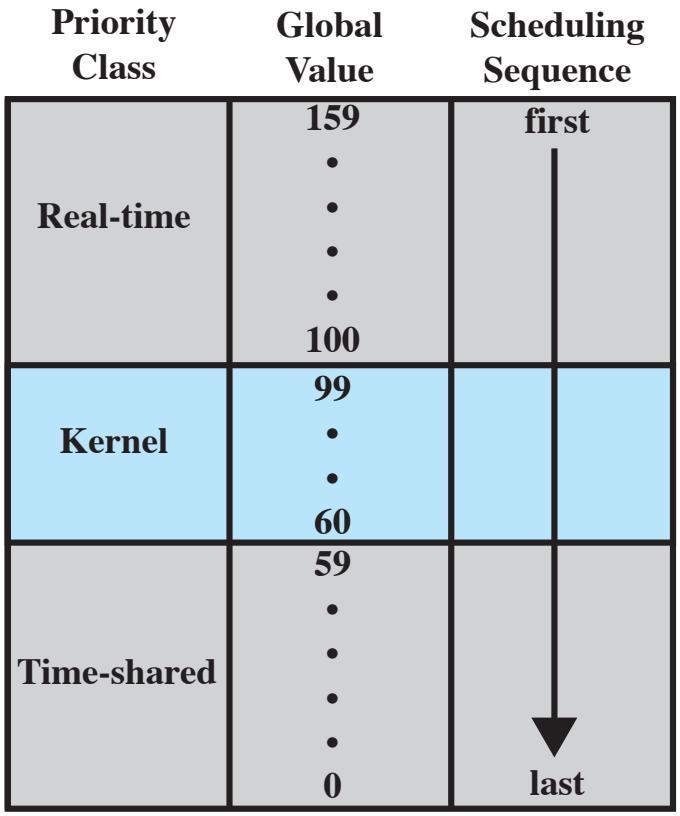
- **A complete overhaul of the scheduling algorithm used in earlier UNIX systems**

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

UNIX SVR4 优先级类型



UNIX SVR4 优先级类型

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

UNIX SVR4 分发队列

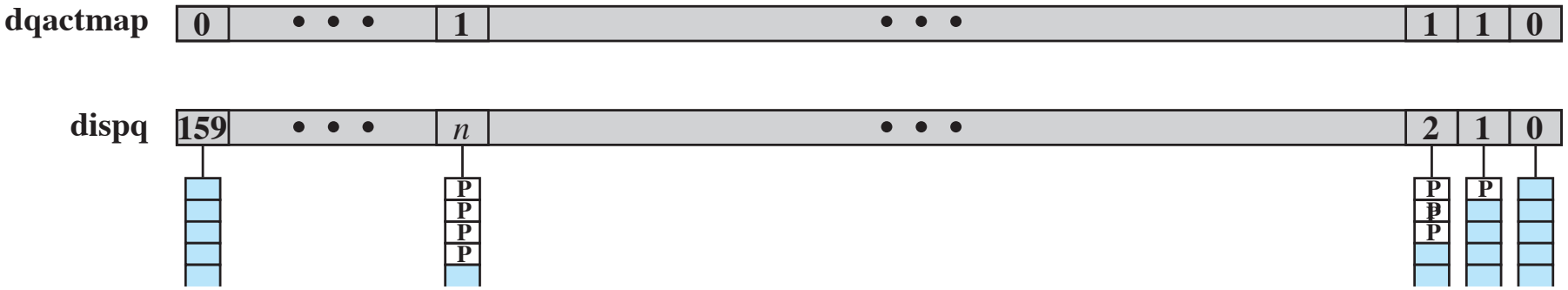


Figure 10.13 SVR4 Dispatch Queues

FreeBSD线程调度类

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

Note: Lower number corresponds to higher priority

FreeBSD线程调度类

- 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

FreeBSD线程调度类

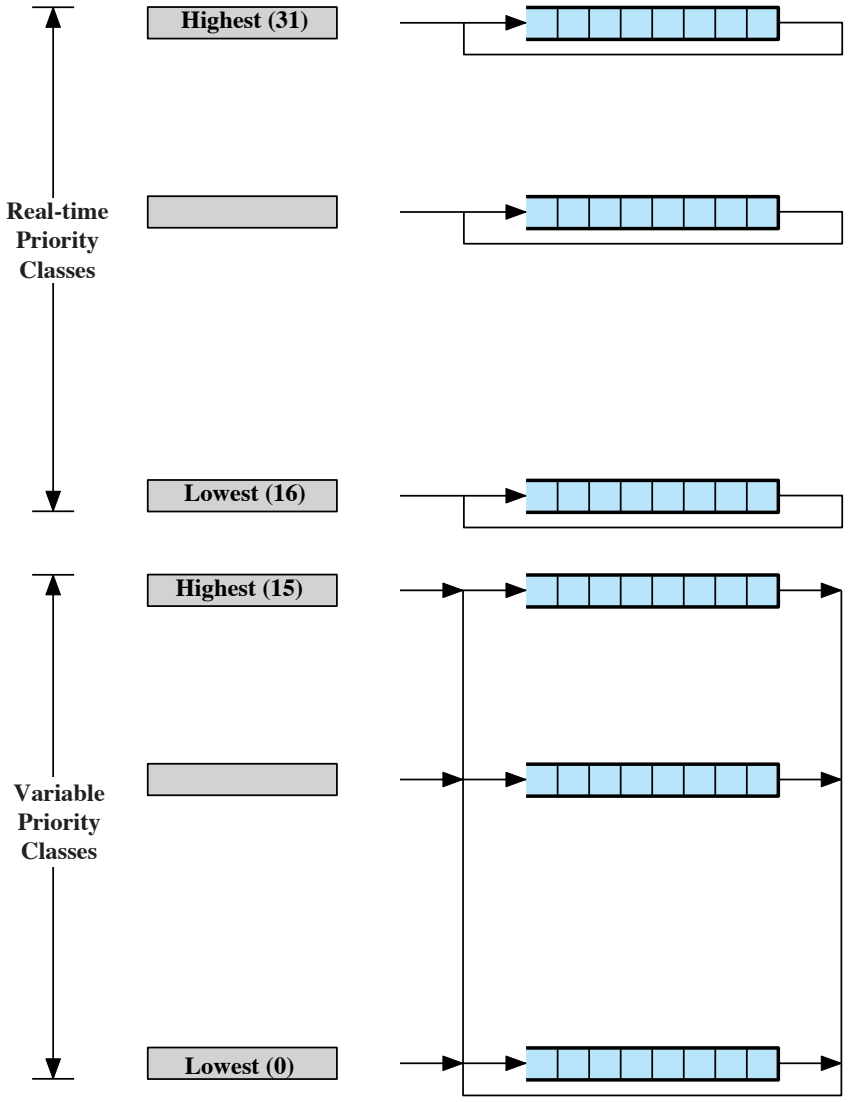
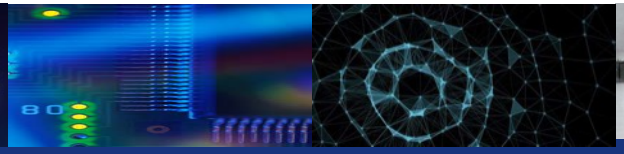


Figure 10.14 Windows Thread Dispatching Priorities

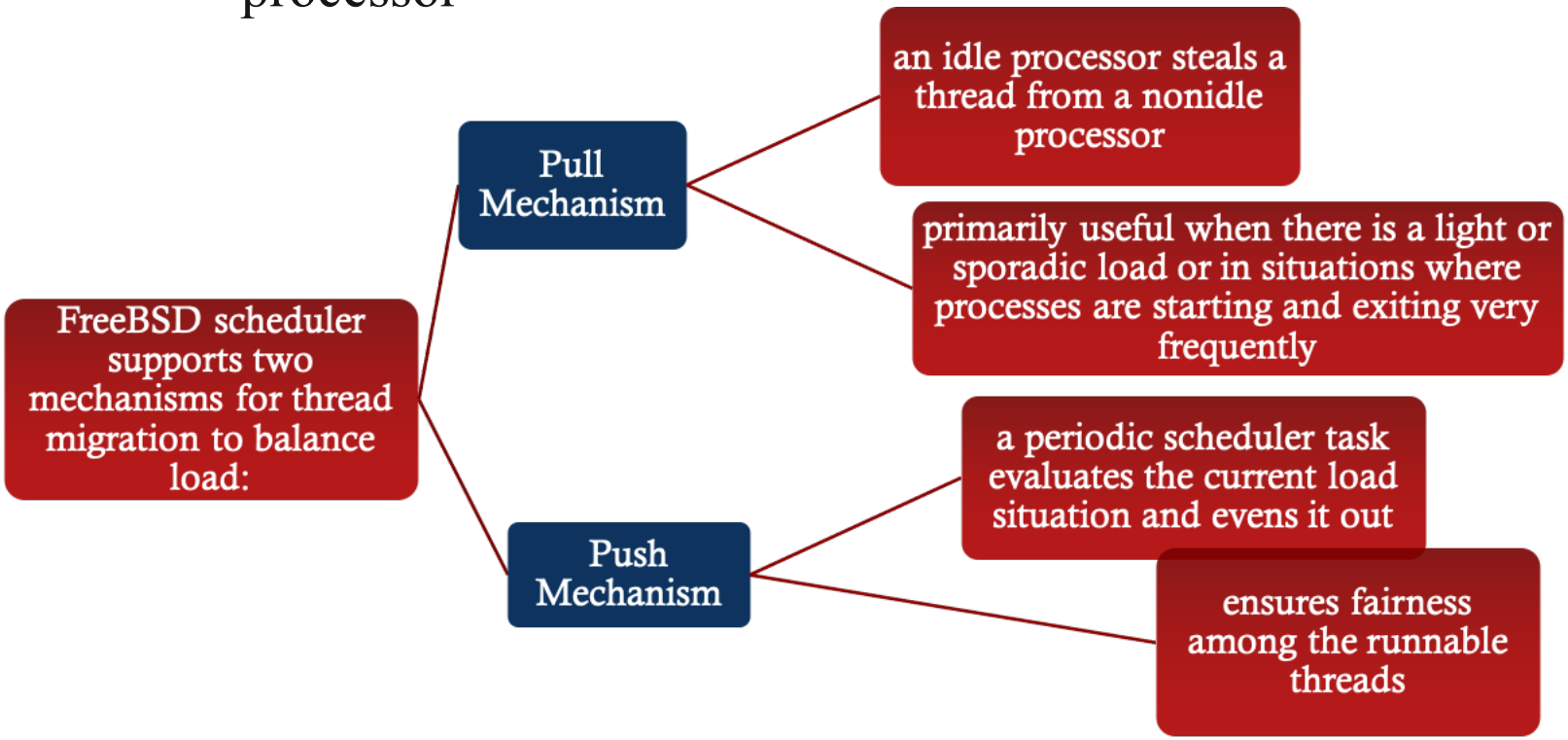


交互值

- 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

线程迁移

- ***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



Windows调度

- 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

Windows优先级

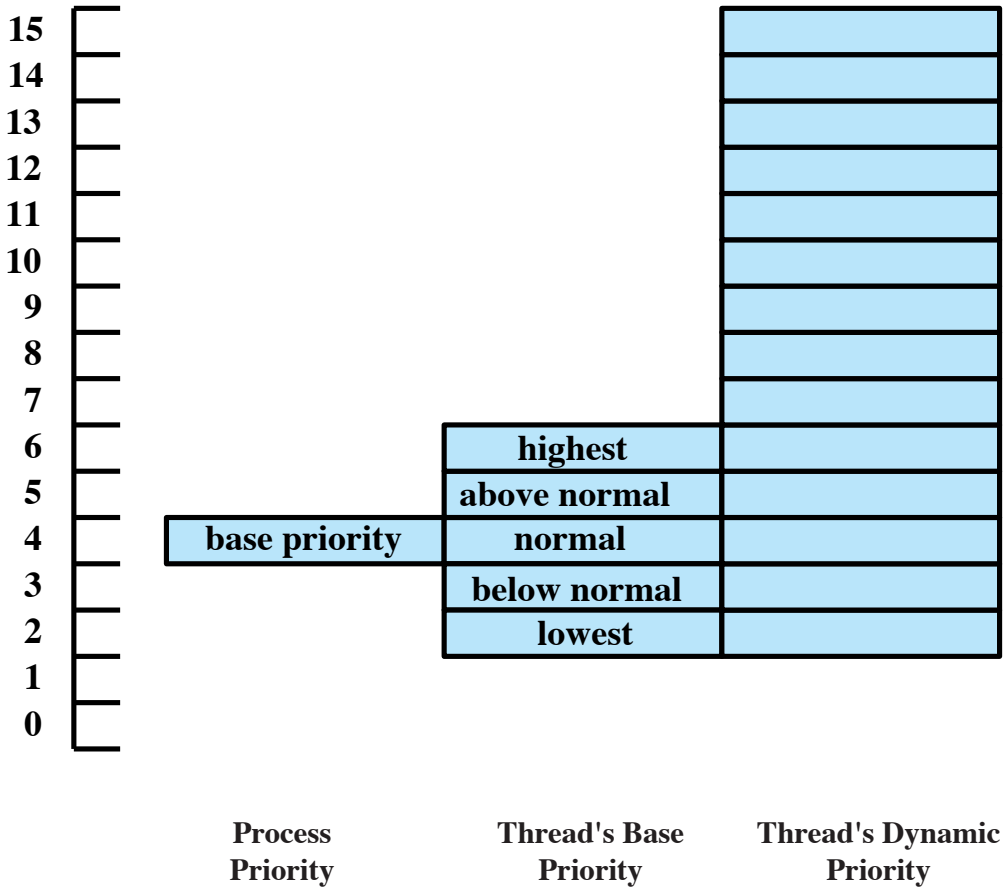


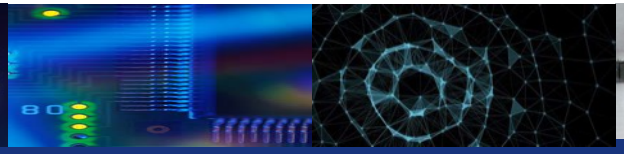
Figure 10.15 Example of Windows Priority Relationship

多处理器调度

- **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**

总结

- **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



谢谢