

2026

CS 332

Subject 6

NJIT

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details



Module 1

*Operating
Systems:
Internals
and
Design
Principles*

Chapter 10 Multiprocessor, Multicore and Real-Time Scheduling

Ninth Edition
By William Stallings



Subject 5

1-Dispatcher,

1-ReadyQueue,

1-Processor

Subject 6

≥ 1 Dispatchers,

≥ 1 ReadyQueues

> 1 Processor

('Multiprocessor')

ExecUnit

DataProcessingUnit $\wedge$ HasRegisterFile

SMP

$\#Computer=1 \wedge \#Processor > 1 \wedge \text{SameType}$

Multiprocessor:

$\#Computer=1 \wedge \#Processor > 1$ or equivalently

$\#Computer=1 \wedge \#ExecutionUnit > 1$

Multicomputer

$\#Computer > 1$

MulticoreProcessor

$\#Processor=1 \wedge \#ExecUnit > 1 \wedge \text{OnOneChip}$

Note: $\wedge$: conjunction , $\vee$ disjunction

Multiprocessor System: Multicomputer $\vee$ Multiprocessor

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

eg Hadoop Cluster
(NameNode, DataNode)

- 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

Order of magnitude : Factor of 10

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

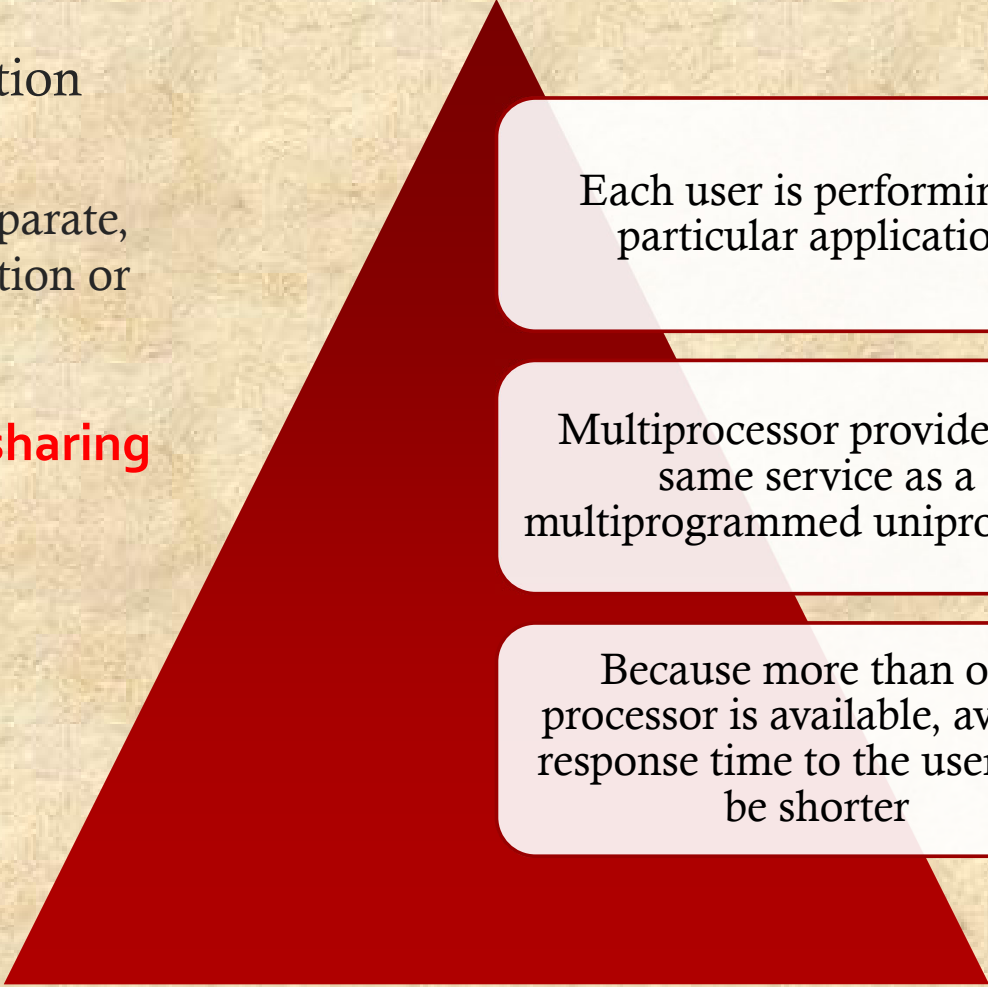
Table 10.1 Synchronization Granularity and Processes



Fine	Specialized parallel programs
Medium	Pi computation in Subject 4
Coarse	<code>ps -ef egrep somestring</code>
Very Coarse	Client + Server (eg Web server)
Independent	User A and User B on the same computer

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 shorter

Coarse and Very Coarse Grained Parallelism

- There is **synchronization among processes**, but at a very gross level
- Easily handled as a set of 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 processes only not threads

Design Issues

Scheduling on a multiprocessor involves three interrelated issues:

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

Actual dispatching of a process

Use of multiprogramming on individual processors

Assignment of processes to processors

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

Static vs Dynamic



Static

CPU 0 : 20 CPU1: 21 CPU2: 42 CPU 3: avail.

Schedule process i to $i\%4$ processor

Process 10 goes to proc $10\%4$
and states there (ready queue of 2)

Requires 1 ReadyQ per proc

Dynamic

Schedule to first available processor.

Process 10 goes to 3 since 3 is avail.
(it might stay there or not depending
on further scheduling)

Might use 1 ReadyQ per proc
or 1 ReadyQ per system

What if
10,20,30,40,50 show up?



Static Assignment

One ReadyQ/proc + One Scheduler/Proc i.e. p p

Dynamic Assignment

One ReadyQ/proc + One Scheduler/Proc i.e. p p

One ReadyQ/OS + One Scheduler/OS i.e. 1 1

One ReadyQ/OS + One Scheduler/ Proc i.e. 1 p

1 1 => master/slave (a.k.a. client/server)

2 p => peer approach

p schedulers => p processors' CPU utilization is SERIOUSLY affected

1 scheduler => 1 processor's CPU utilization VERY SERIOUSLY affected

p-1 processors' CPU utilization affected

SERIOUS => stays AWAY from RR may be FCFS

Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Approaches:
 - Master/Slave ... we prefer instead ...
Client/Server
 - Peer



Master : Server

One server determined at boot time ?

Slave : Client

Ask server

Peer approach

All for one (Ready Queue)

queues?

1 per system or 1 per processor

Bottleneck?

The Server or the only one ReadyQ

Single Point of failure?

Likewise

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



Process Scheduling

- In most traditional multiprocessor systems, 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, all feeding into the common pool of processors
- System is viewed as being a multi-server queuing architecture



Keep it simple

Almost as good:
Other options:

FCFS

might get complicated
implementation-wise



Scheduling processes and threads

Threads can be ULT or KLT

- (1) Threads and process that created them share data
Example. Process 10 creates threads 11,12,13
- (2) What if Process 10 on processor 0 does $x=10$ and
thread 11 on processor 1 does $x=11$ and
thread 12 on processor 2 does $x=12$.

Where is x ? (What L1, or L2 cache has it)?

What is the value of x ?

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

Processes are not assigned to a particular processor

Load Sharing

A set of related threads scheduled to run on a set of processors at the same time, on a one-to-one basis

Gang Scheduling

Four approaches for multiprocessor thread scheduling and processor assignment are:

Provides implicit scheduling defined by the assignment of threads to processors

Dedicated Processor Assignment

The number of threads in a process can be altered during the course of execution

Dynamic Scheduling



Load Sharing: Keep processors busy ('peer' method – like)

Dedicated Processor Assignment: Too many processORs, each one has a ReadyQ, can even use FCFS

Gang Scheduling Process 10 and its threads 11 and 12 better use same set of processors.

Dynamic Scheduling: Schedule from within code!

Never forget: What if process 20 moves from processor 0 to 2 to 1 to 3?
What if process 10 moves from processor 0 to 3 ?
(what happens to its threads 11, 12 then?)

What if: What happens to the cached copies of variables in processor 0 when process 20,10 move to 2,3 respectively?



Processor Affinity

Schedule next a process/thread to the same processor that run the process/thread before

Example. Process 10 started at 1 and will remain at 1 through the end of its execution
(if it had started at 2, it would have stayed at 2)

Processor pinning

Schedule next a process/thread to a specific processor

Example. Process 10 ought to start at 2 (and should stay there).



Load sharing aka share the process load (also temporary)

A,B,C,D have service times 1s,2s,3s,4s

Processor 0 gets A,B

[CPU service time is $1+2=3s$]

Processor 1 gets C,D

[CPU service time is $3+4=7s$]

Load is shared evenly (2 per processor)

Load balancing aka balance the CPUs' time (also permanent)

Processor 0 gets A,D

[CPU service time is $1+4=5s$]

Processor 1 gets B,C

[CPU service time is $2+3=5s$]

CPU's are evenly balanced



Load sharing aka share the process load (also temporary)

A,B,C,D have service times 1s,2s,3s,4s

Load is uneven: 3 vs 7

Processor 0 gets A,B,C,D

Processor 1 gets nothing

Load is shared evenly (7 per processor)

RQ Proc 0 : E,F,G

RQ Proc 1: H,I,J,K,L,M,N

RQ Proc 0 : E,F,G,A,B,C,D

RQ Proc 1 : H,I,J,K,L,M,N

Load Sharing

- Simplest approach and the one that carries over most directly from a uniprocessor environment

Advantages:

- Load is distributed evenly across the processors, assuring that no processor is idle while work is available to do
- 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 (FCFS)
 - Smallest number of threads first (track job's # threads) + FCFS
 - 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 [Affinity]
- 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
- 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

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



Real time systems

Reactors, Aircraft, Rockets etc

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

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

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

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
- The following features are common to most real-time OSs
 - A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
 - Interrupt latency is bounded and relatively short
 - More precise and predictable timing characteristics than general purpose OSs

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



Processes priorities: readyQHigh, readyQLow
(when readyQHigh is empty we use readyQLow)

when CPU has a process from readyQLow
process can get preempted if one arrives in
readyQHigh

**High priority(HP) process wants to use same resource
currently in use by Low priority (LP) process**

LP must release resource first i.e. use CPU to completion
ahead of HP, contradicting the principle of priority

HP is HP for a reason; but it must relinquish control of CPU
to LP

Chicken and egg problem



Why CS435 or CS610

are not useless courses...

or

Why CS114 / CS 116/CS505 are not enough

or

Use of a HashMap in Java is dangerous if you do not know what you are doing and can't reason with its use.

Linux Scheduling

- The three primary Linux scheduling classes are:
 - 0-99 SCHED_FIFO: First-in-first-out real-time threads
 - 0-99 SCHED_RR: Round-robin real-time threads
 - 100-139 SCHED_NORMAL: Other, non-real-time threads
- Within each class multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_NORMAL class

Non-Real-Time Scheduling

- Linux 2.6 uses a completely new priority scheduler known as the $O(1)$ scheduler
- Linux 2.6.23 uses a new $O(\lg P)$ scheduler ...

RED - BLACK TREES (CS435)

- The scheduler is designed so the 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
- The $O(1)$ scheduler proved to be unwieldy in the kernel because the amount of code is large and the algorithms are complex



Linux Kernel 2.6

Theory

$O(1)$ scheduler

($O(1)$ means constant time; $O(1)$ scheduling uses Hashing)
[most of you think Hashing == Java's HashMap or
whatever Python supports]

Reality

10000 instructions

(constant time translates into 10000 instructions)

10000 was too much for "constant" time

thus

$O(\lg p)$ scheduler using Red-black Trees

($O(\lg p)$ with p denoting # of processes; max p for Linux is 32768,
average around 200 processes in the system;
 $O(\lg p)$ is at most $2\lg p$ operations, roughly $4\lg p$ instructions)

~400 instructions

Max: 600 instructions
Ave: 320 instructions

In theory $O(1)$ sounds better than $O(\lg p)$ but it is not!

OUTCOME and coincidence

1. Kernel 2.6.26 started using red-black trees ($O(1)$ replaced!)
2. Java 8 started using red-black trees instead of variable length arrays ('linked list' without pointers) for the HashMap implementation!

3. **Conclusion: Use theory carefully!**

Summary

- Multiprocessor and multicore scheduling
 - Granularity
 - Design issues
 - Process scheduling
 - Thread scheduling
 - Multicore thread scheduling
- Linux scheduling
 - Real-time scheduling
 - Non-real-time 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
- UNIX SVR4 scheduling



1.1 Explain the concept of priority inversion in one sentence.

1.2 We are using load-sharing to allocate threads to processors. Name one problem with such an approach if a process has say 5 threads.

1.3 We are using dedicated processor assignment to allocate threads to processors. Name one problem with such an approach if a process has say 5 threads and number of available processors is 7.

1.4 "I want to use processor 3 said process 22' ". How would you describe this ? **Processor**

(fill the dots)

NJIT

A white curved line that starts under the 'N', arches over the 'J' and 'I', and ends under the 'T'.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department