Multiprocessor Scheduling and Scheduling in Linux Kernel 2.6

Winter Term 2008 / 2009

Jun.-Prof. Dr. André Brinkmann

Andre.Brinkmann@uni-paderborn.de

Universität Paderborn – PC²

PADERBORN
CENTER FOR
PARALLEL
COMPUTING

Agenda

- Multiprocessor and Real-Time Scheduling
  - Based on Stalling, Chapter 10
- Linux Scheduling
  - Taken from “Scheduling in Kernel 2.6“, Mahendra M, infosys
  - Process Scheduling
  - O(1) scheduler – design, performance
  - Pre-emption

Classifications of Multiprocessor Systems

- Loosely coupled or distributed multiprocessor, or cluster
  - Each processor has its own memory and I/O channels
- Functionally specialized processors
  - Such as I/O processor
  - Controlled by a master processor
- Tightly coupled multiprocessing
  - Processors share main memory
  - Controlled by operating system

Independent Parallelism

- Separate application or job
- No synchronization among processes
- Example is time-sharing system

Coarse and Very Coarse-Grained Parallelism

- Synchronization among processes at a very gross level
- Good for concurrent processes running on a multiprogrammed uniprocessor
  - Can by supported on a multiprocessor with little change

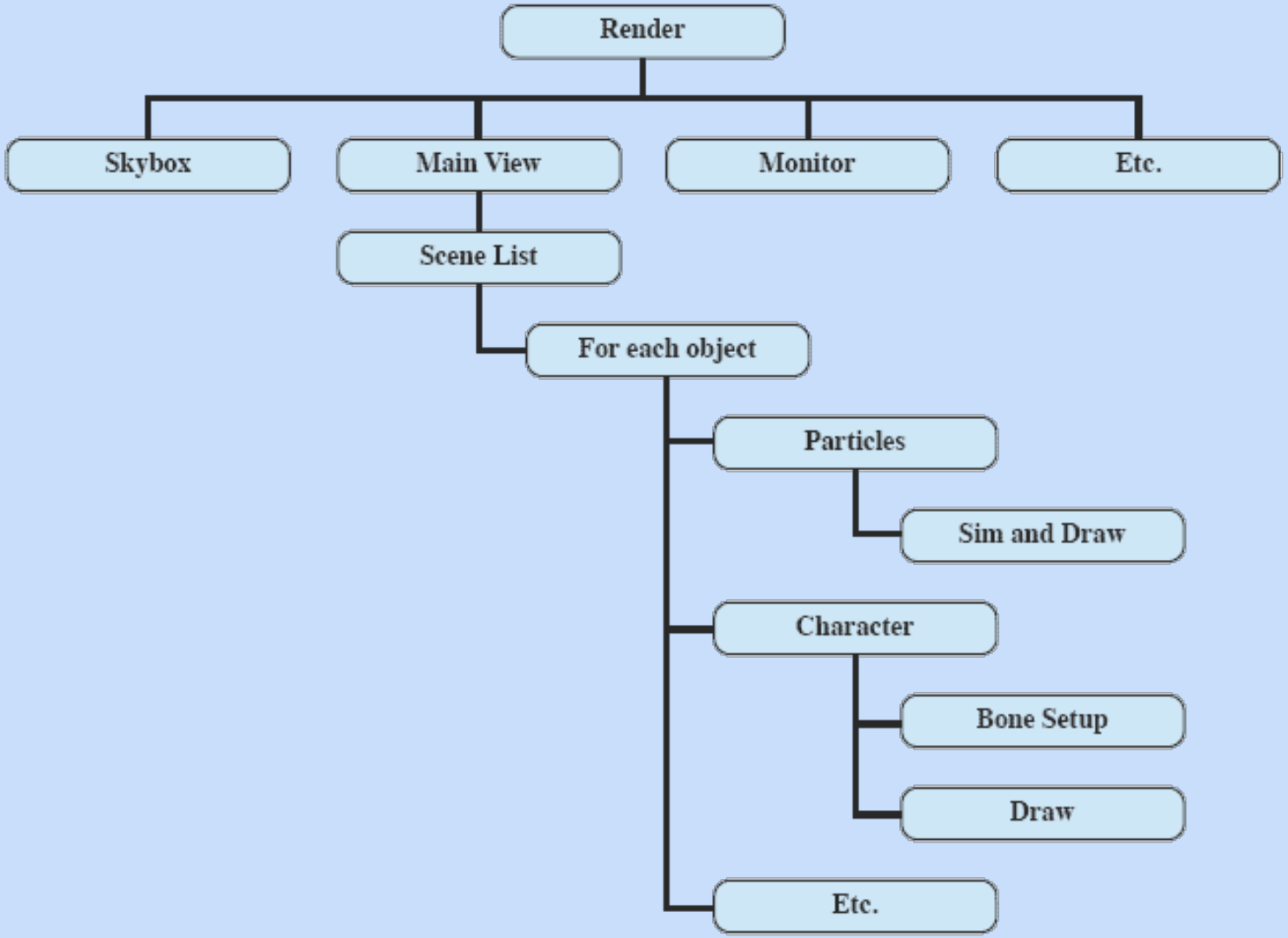
Medium-Grained Parallelism

- Single application is a collection of threads
- Threads usually interact frequently

Fine-Grained Parallelism

- Highly parallel applications
- Specialized and fragmented area

Thread Structure for Rendering Module



Scheduling Design Issues

- Assignment of processes to processors
- Use of multiprogramming on individual processors
- Actual dispatching of a process

Assignment of Processes to Processors

- Treat processors as a pooled resource and assign process to processors on demand
- Permanently assign process to a processor
  - Known as group or gang scheduling (not correct)
  - Dedicate short-term queue for each processor
  - Less overhead
  - Processor could be idle while another processor has a backlog

Assignment of Processes to Processors

- Global queue
  - Schedule to any available processor

Assignment of Processes to Processors

- Master/slave architecture
  - Key kernel functions always run on a particular processor
  - Master is responsible for scheduling
  - Slave sends service request to the master
  - Disadvantages
    - Failure of master brings down whole system
    - Master can become a performance bottleneck

Assignment of Processes to Processors

- Peer architecture
  - Kernel can execute on any processor
  - Each processor does self-scheduling
  - Complicates the operating system
    - Make sure two processors do not choose the same process

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

Process Scheduling

- Single queue for all processes
- Multiple queues are used for priorities
- All queues feed to the common pool of processors

Thread Scheduling

- Executes separate from the rest of the process
- An application can be a set of threads that cooperate and execute concurrently in the same address space

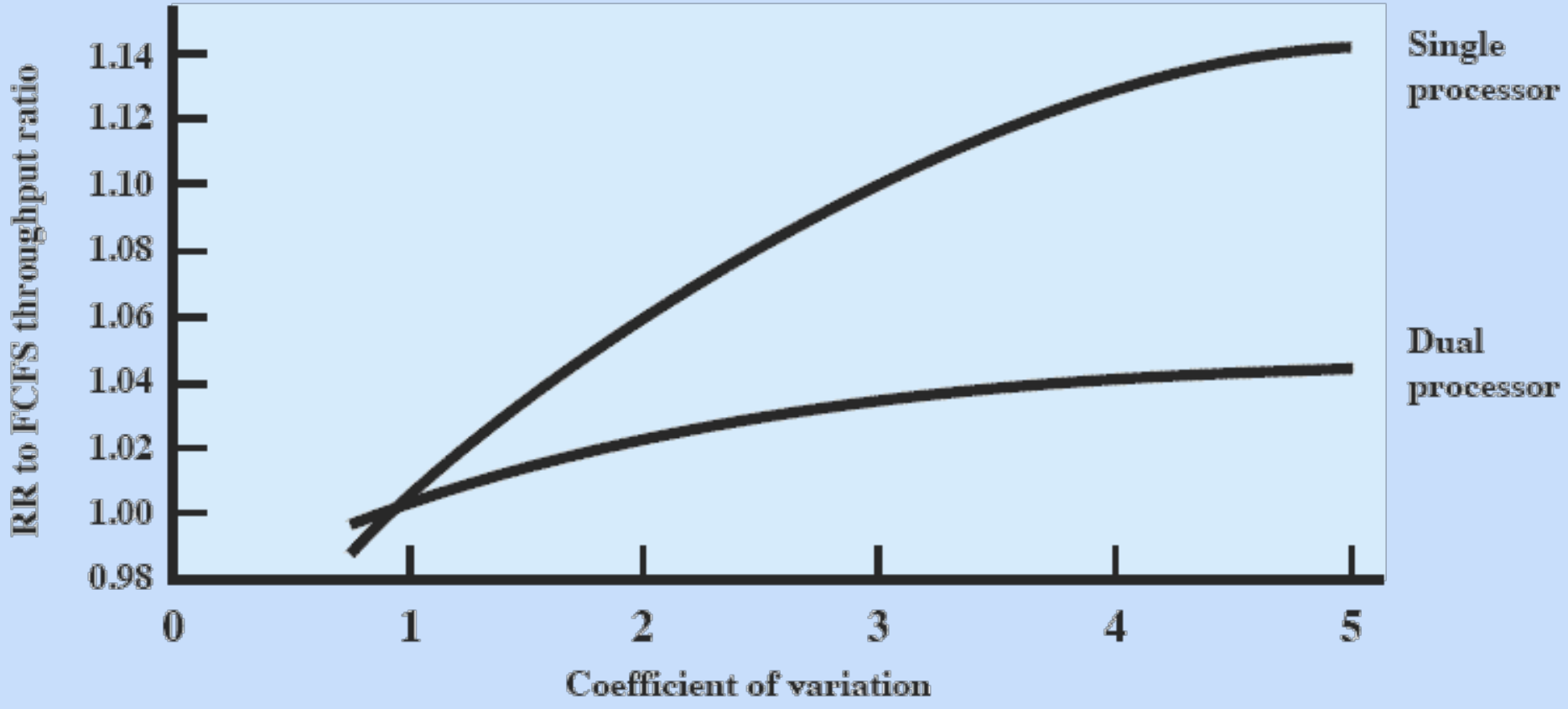
Multiprocessor Thread Scheduling

- Load sharing
  - Processes are not assigned to a particular processor
- Gang scheduling
  - A set of related threads is scheduled to run on a set of processors at the same time

Multiprocessor Thread Scheduling

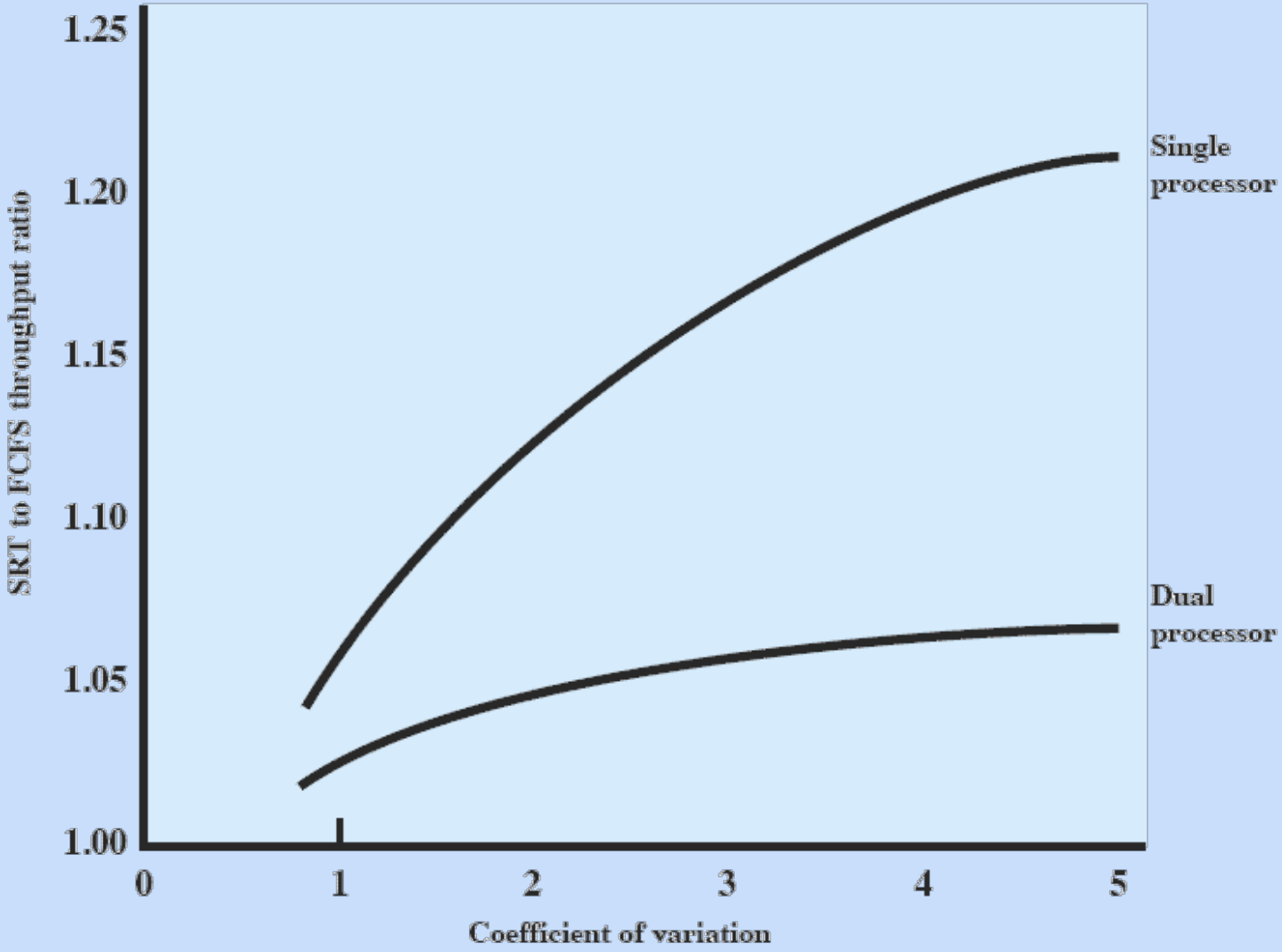
- Dedicated processor assignment
  - Threads are assigned to a specific processor
- Dynamic scheduling
  - Number of threads can be altered during course of execution

Comparison One and Two Processors



(a) Comparison of RR and FCFS

Comparison One and Two Processors



(b) Comparison of SRT and FCFS

Load Sharing

- Load is distributed evenly across the processors
- No centralized scheduler required
- Use global queues

Disadvantages of Load Sharing

- Central queue needs mutual exclusion
- Preemptive threads are unlikely to resume execution on the same processor
- If all threads are in the global queue, all threads of a program will not gain access to the processors at the same time

Gang Scheduling

- Simultaneous scheduling of threads that make up a single process
- Useful for applications where performance severely degrades when any part of the application is not running
- Threads often need to synchronize with each other

Example Scheduling Groups

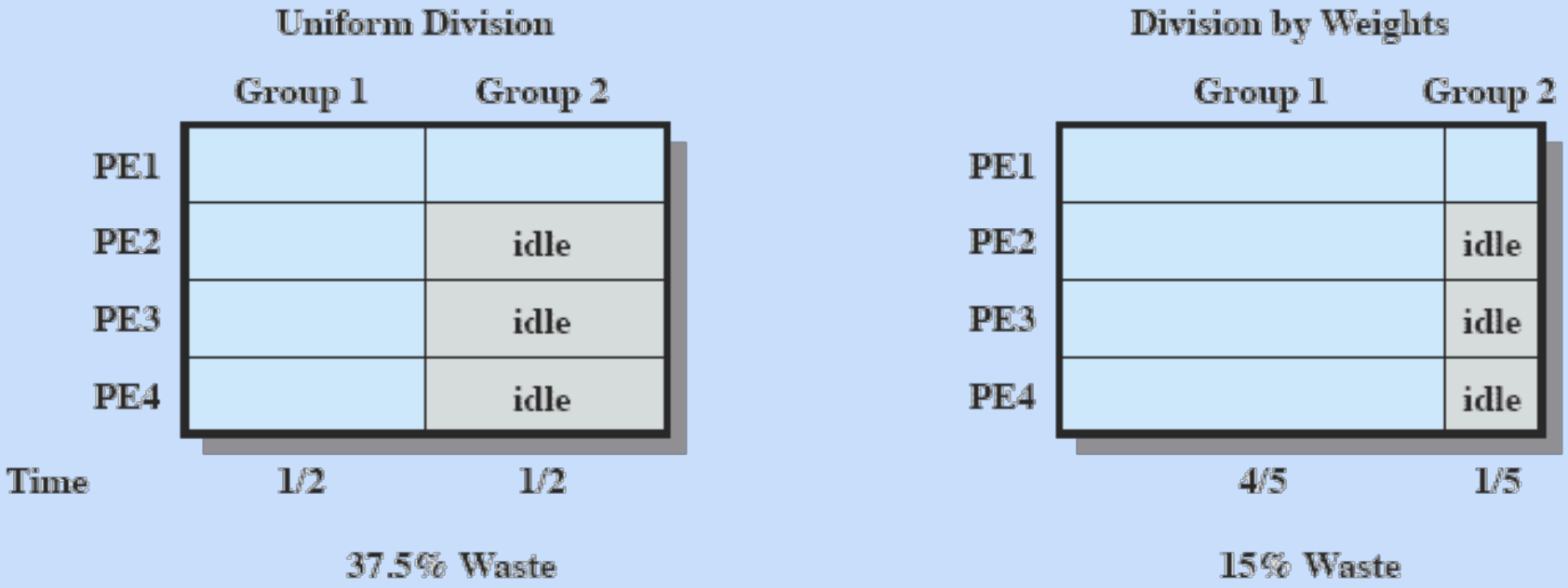


Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]

Dedicated Processor Assignment

- When application is scheduled, its threads are assigned to a processor
- Some processors may be idle
- No multiprogramming of processors

Application Speedup

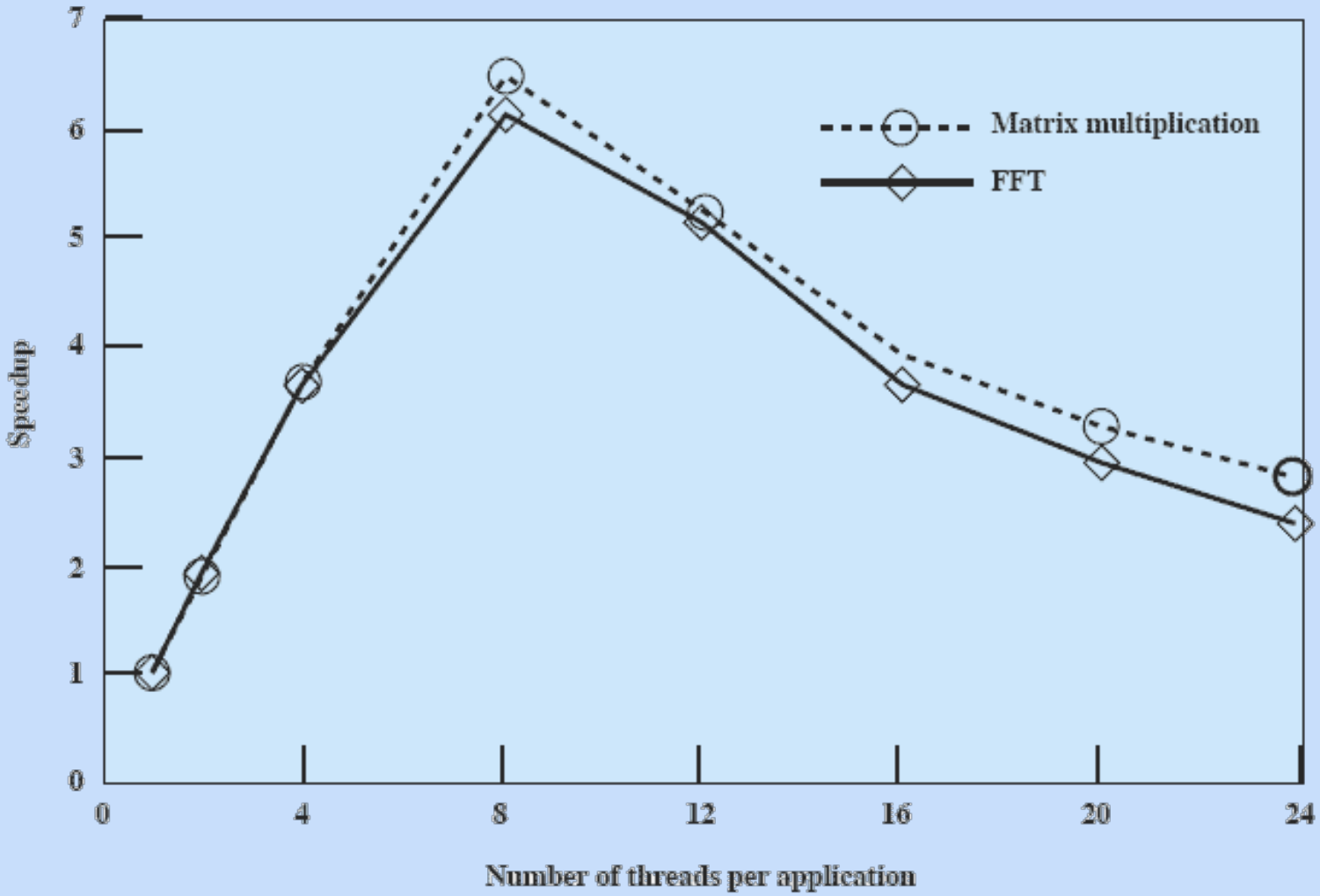


Figure 10.4 Application Speedup as a Function of Number of Threads

Dynamic Scheduling

- Number of threads in a process are altered dynamically by the application
- Operating system adjust the load to improve utilization
  - Assign idle processors
  - New arrivals may be assigned to a processor that is used by a job currently using more than one processor
  - Hold request until processor is available
  - Assign processor a job in the list that currently has no processors (i.e., to all waiting new arrivals)

Agenda

- Multiprocessor and Real-Time Scheduling
  - Based on Stalling, Chapter 10
- Linux Scheduling
  - Taken from “Scheduling in Kernel 2.6“, Mahendra M, infosys
  - Process Scheduling
  - O(1) scheduler – design, performance
  - Pre-emption

Considerations in Scheduler design

- Fairness
  - Prevent starvation of tasks
- Scheduling latency
  - Reduction in delay between a task waking up and actually running
  - Time taken for the scheduler decisions
- Interrupt latency
  - Delay in processing h/w interrupts
- Scheduler decisions

Process Scheduler

- Goals
  - Good interactive performance during high load
  - Fairness
  - Priorities
  - SMP efficiency
  - SMP affinity
    - Issues of random bouncing taken care
    - No more 'timeslice squeeze’
  - RT Scheduling

(From /usr/src/linux-2.6.x/Documentation/sched-design.txt)

Goals (contd ....)

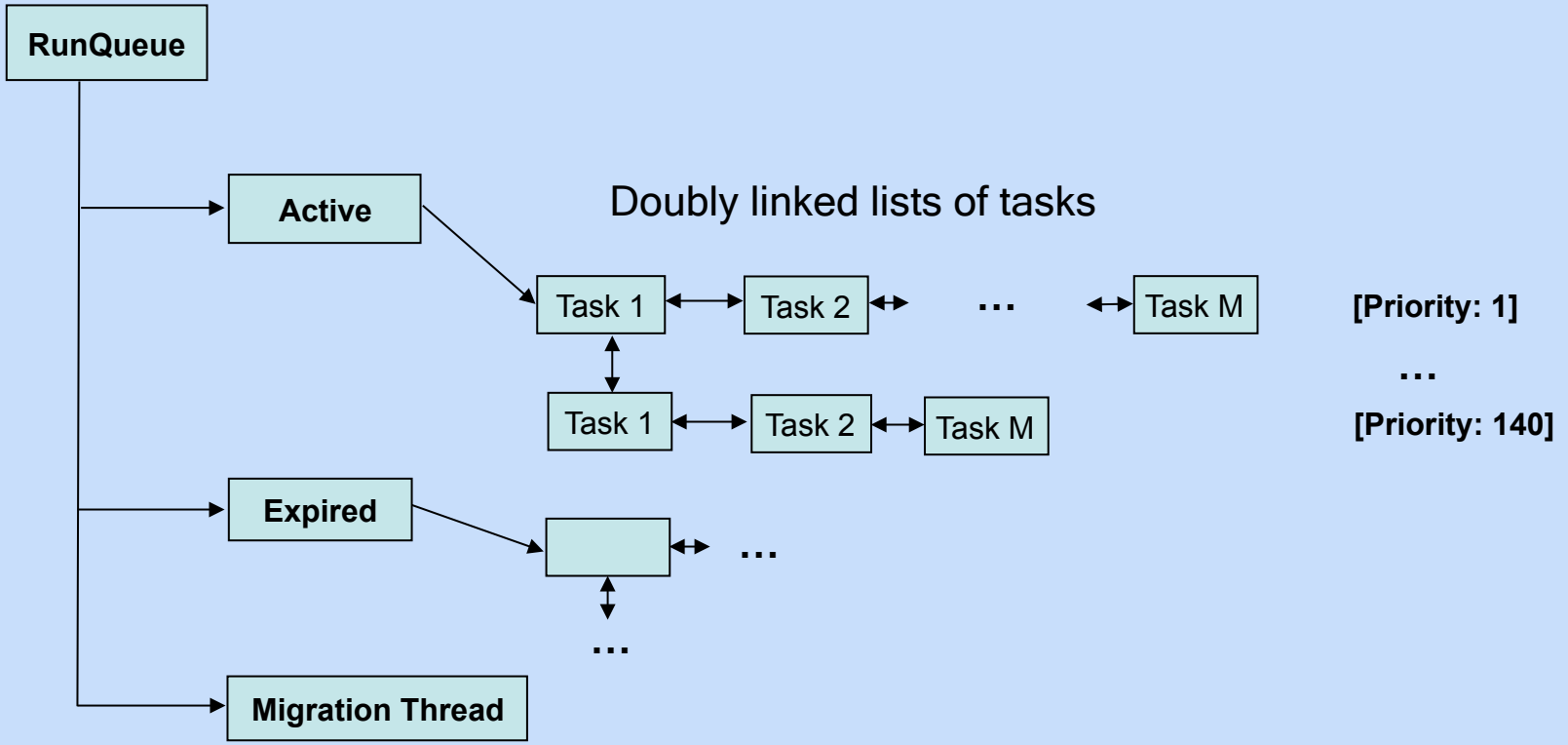
- Full O(1) scheduling
  - Great shift away from O(n) scheduler
- Perfect SMP scalability
  - Per CPU runqueues and locks
  - No global lock/runqueue
  - All operations like wakeup, schedule, context-switch etc. are in parallel
- Batch scheduling (bigger timeslices, RR)
- No scheduling storms
- O(1) RT scheduling

Design

- 140 priority levels
  - The lower the value, higher is the priority
  - Eg : Priority level 110 will have a higher priority than 130.
- Two priority-ordered 'priority-arrays' per CPU
  - 'Active' array : tasks which have timeslices left
  - 'Expired' Array : tasks which have run
  - Both accessed through pointers from per-CPU runqueue
- They are switched via a simple pointer swap

Per-CPU runqueue

Each CPU on the system has it's own RunQueue



From kernel / sched. c

O(1) Algorithm (Constant time algorithm)

- The highest priority level, with at-least ONE task in it, is selected
  - This takes a fixed time (say *t1*)
- The first task (head of the doubly-linked list) in this priority level is allowed to run
  - This takes a fixed time (say *t2*)
- The time taken for selecting a new process will be fixed (constant time irrespective of number of tasks)
- Deterministic algorithm !!

Linux Data Structures

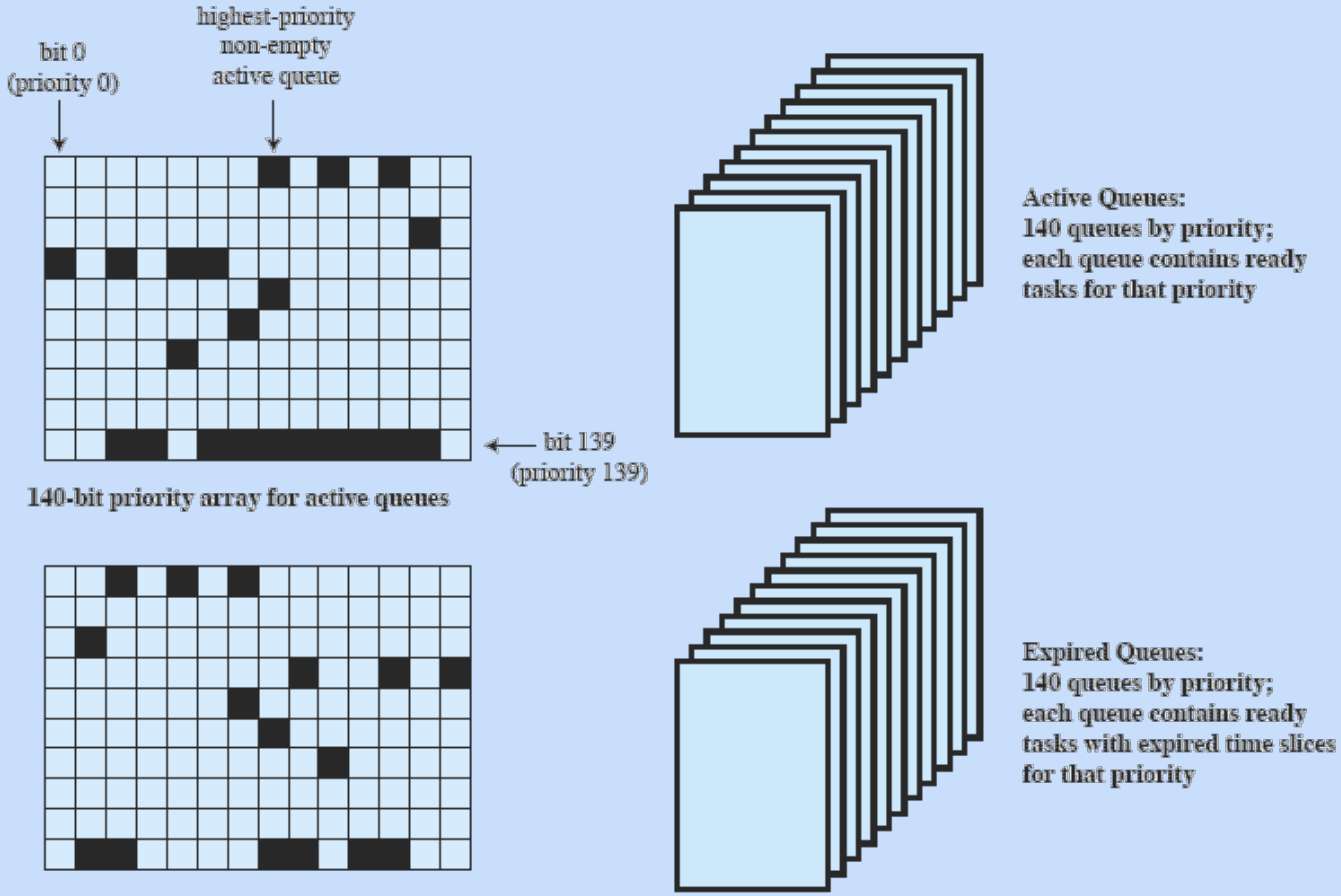


Figure 10.12 Linux Scheduling Data Structures for Each Processor

2.6 v/s 2.4

- Kernel 2.4 had
  - A Global runqueue.
  - All CPUs had to wait for other CPUs to finish execution.

- An O(n) scheduler.
  - In 2.4, the scheduler used to go through the entire “ global runqueue” to determine the next task to be run.
  - This was an O(n) algorithm where 'n' is the number of processes. The time taken was proportional to the number of active processes in the system.

- This lead to large performance hits during heavy workloads.

Scheduling policies in 2.6

- 140 Priority levels
  - 1-100 : RT prio (MAX_RT_PRIO = 100)
  - 101-140 : User task Prio (MAX_PRIO = 140)
- Three different scheduling policies
  - One for normal tasks
  - Two for Real time tasks
- Normal tasks
  - Each task assigned a “Nice” value
  - PRIO = MAX_RT_PRIO + NICE + 20
  - Assigned a time slice
  - Tasks at the same prio are round-robined.
  - Ensures Priority + Fairness

Policies (contd ...)

- RT tasks (Static priority)
  - FIFO RT tasks
    - Run until they relinquish the CPU voluntarily
    - Priority levels maintained
    - Not pre-empted !!
  - RR RT tasks
    - Assigned a timeslice and run till the timeslice is exhausted.
    - Once all RR tasks of a given prio level exhaust their timeslices, their timeslices are refilled and they continue running.
    - Prio levels are maintained
- The above can be unfair !! - Sane design expected !!

Interactivity estimator

- Dynamically scales a tasks priority based on it's interactivity
- Interactive tasks receive a prio bonus [-5]
  - Hence a larger timeslice
- CPU bound tasks receive a prio penalty [+5]
- Interactivity estimated using a running sleep average.
  - Interactive tasks are I/O bound. They wait for events to occur.
  - Sleeping tasks are I/O bound or interactive !!
  - Actual bonus/penalty is determined by comparing the sleep average against a constant maximum sleep average.
- Does not apply to RT tasks

Recalculation of priorities

- When a task finishes it's timeslice :
  - It's interactivity is estimated
  - Interactive tasks can be inserted into the 'Active' array again.
- Else, priority is recalculated
  - Inserted into the NEW priority level in the 'Expired' array.

Finegrained timeslice distribution

- Priority is recalculated only after expiring a timeslice.
- Interactive tasks may become non-interactive during their LARGE timeslices, thus starving other processes.
- To prevent this, time-slices are divided into chunks of 20ms
- Task of equal priority may preempt running task every 20ms
- The preempted task is requeued and is round-robined in it's priority level.
- Also, priority recalculation happens every 20ms.

For programmers

From /usr/src/linux-2.6.x/kernel/sched.c

- void schedule()
  - The main scheduling function.
  - Upon return, the highest priority process will be active

- Data
  - struct runqueue()
    - The main per-CPU runqueue data structure
  - struct task_struct()
    - The main per-process data structure

For programmers (contd....)

Process Control methods

- void set_user_nice (...)
  - Sets the nice value of task p to given value
- int setscheduler(...)
  - Sets the scheduling policy and parameters for a given pid
- rt_task(pid)
  - Returns true if pid is real-time, false if not.
- yield()
  - Place the current process at the end of the runqueue and call schedule().

Handling SMP (multiple CPUs)

- A run-queue per CPU
  - Each CPU handles it's own processes and do not have to wait till other CPU tasks finish their timeslices.
- A 'migration' thread runs for every CPU.
- void load_balance()
  - This function call attempts to pull tasks from one CPU to another to balance CPU usage if needed.
- Called
  - Explicitly if runqueues are inbalanced
  - Periodically by the timer tick
- Processes can be made affine to a particular CPU.