Slide 1

Last Lecture

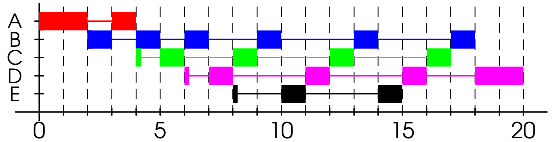
Scheduling Algorithms:

- ➜ FIFO
- ➜ Shortest job next
- ➜ Shortest remaining job next
- ➜ Highest Response Rate Next (HRRN)

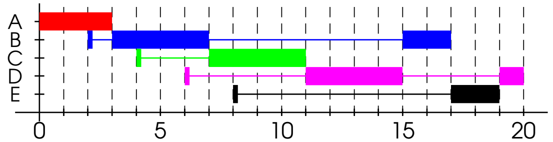
Slide 2

Round-robin

quantum=1:



quantum=4:



- ➜ Scheduled thread is given a time slice
- ➜ Running thread is preempted upon clock interrupt, running thread is returned to *ready* queue

Slide 3

Performance of round-robin scheduling:

- ➜ Average waiting time: not optimal
- ➜ Performance depends heavily on size of time-quantum:
  - too short: overhead for context switch becomes too expensive
  - too large: degenerates to FCFS policy
  - rule of thumb: about 80% of all bursts should be shorter than 1 time quantum
- ➜ no starvation

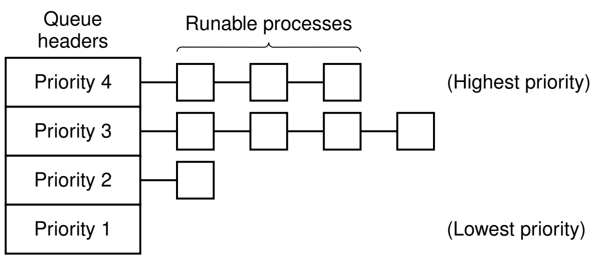
Slide 4

Priorities

- ➜ Each thread is associated with a priority
- ➜ Basic mechanism to influence scheduler decision:
  - Scheduler will always choose a thread of higher priority over one of lower priority
  - Implemented via multiple FCFS ready queues (one per priority)
- ➜ Lower-priority may suffer starvation
  - adapt priority based on thread's age or execution history
- ➜ Priorities can be defined internally or externally
  - internal: e.g., memory requirements, I/O bound vs CPU bound
  - external: e.g., importance of thread, importance of user

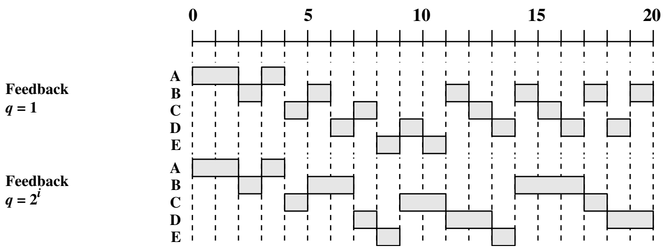
Slide 5

Priority queueing:



Slide 6

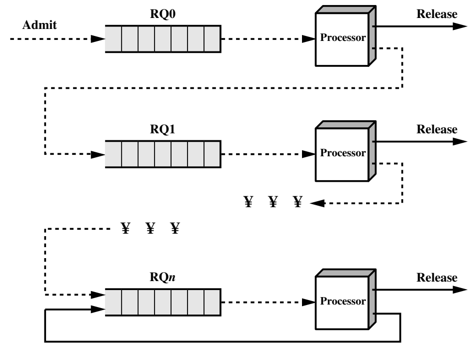
Feedback scheduling:



- ➜ Penalize jobs that have been running longer
- ➜ $q = 2^i$: longer time slice for lower priority (reduce starvation)

Slide 7

Feedback scheduling:



Priorities influence access to resources, but do not guarantee a certain fraction of the resource (CPU etc)!

Slide 8

LOTTERY SCHEDULING

- ➜ process gets "lottery tickets" for various resources
- ➜ more lottery tickets imply better access to resource

Advantages:

- ➜ Simple
- ➜ Highly responsive
- ➜ Allows cooperating processes/threads to implement individual scheduling policy (exchange of tickets)

Slide 9

Example (taken from *Embedded Systems Programming*:

Four processes a running concurrently

- ➜ Process A: 15% of CPU time
- ➜ Process B: 25% of CPU time
- ➜ Process C: 5% of CPU time
- ➜ Process D: 55% of CPU time

How many tickets should each process get to achieve this?

Number of tickets in proportion to CPU time, e.g., if we have 20 tickets overall

- ➜ Process A: 15% of tickets: 3
- ➜ Process B: 25% of tickets: 5
- ➜ Process C: 5% of tickets: 1
- ➜ Process D: 55% of tickets: 11

Slide 10

Traditional UNIX Scheduling (SVR3, 4.3 BSD)

Objectives:

- ➜ support for time sharing
- ➜ good response time for interactive users
- ➜ support for low-priority background jobs

Strategy:

- ➜ Multilevel feedback using round robin within priorities
- ➜ Priorities are recomputed once per second
  - Base priority divides all processes into fixed bands of priority levels
  - Priority adjustment capped to keep processes within bands
- ➜ Favours I/O-bound over CPU-bound processes

Slide 11

Note: UNIX traditionally uses counter-intuitive priority representation (higher value = less priority)

Bands:

- ➜ Decreasing order of priority
  - Swapper
  - Block I/O device control
  - File manipulation
  - Character I/O device control
  - User processes

Slide 12

Advantages:

- ➜ relatively simple, effective
- ➜ works well for single processor systems

Disadvantages:

- ➜ significant overhead in large systems (recomputing priorities)
- ➜ response time not guaranteed
- ➜ non-preemptive kernel: lower priority process in kernel mode can delay high-priority process

Slide 13

NON-PREEMPTIVE VS PREEMPTIVE KERNEL

➜ kernel data structures have to be protected

➜ basically, two ways to solve the problem:

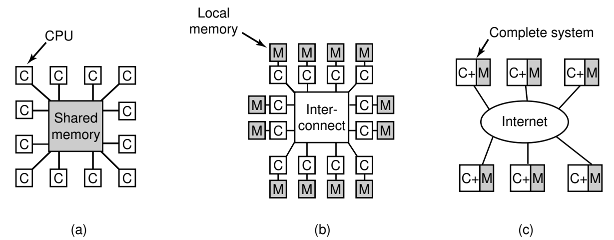
- Non-preemptive: disable all (most) interrupts while in kernel mode, so no other thread can get into kernel mode while in critial section
  - Priority inversion possible
  - Coarse grained
  - Works only for uniprocessor
- Preemptive: just lock kernel data structure which is currently modified
  - More fine-grained
  - Introduces additional overhead, can reduce throughput

Slide 14

MULTIPROCESSOR SCHEDULING

What kind of systems and applications are there?

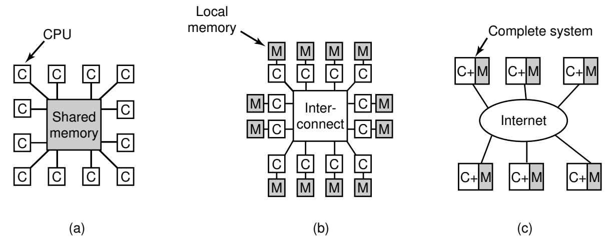
Classification of Multiprocessor Systems:



(a) Tightly coupled multiprocessing

- Processors share main memory, controlled by single operating system, called symmetric multi-processor (SMP) system

Slide 15



(b) Loosely coupled multiprocessor

- Each processor has its own memory and I/O channels
- Generally called a distributed memory multiprocessor

(c) Distributed System

- complete computer systems connected via wide area network
- communicate via message passing

Slide 16

PARALLELISM

Independent parallelism:

➜ Separate applications/jobs

➜ No synchronization

➜ Parallelism improves throughput, responsiveness

➜ Parallelism doesn't affect execution time of (single threaded) programs

Coarse and very coarse-grained parallelism:

➜ Synchronization among processes is infrequent

➜ Good for loosely coupled multiprocessors

- Can be ported to multiprocessor with little change

Slide 17

Medium-grained parallelism:

- ➜ Parallel processing within a single application
  - Application runs as multithreaded process
- ➜ Threads usually interact frequently
- ➜ Good for SMP systems
- ➜ Unsuitable for loosely-coupled systems

Fine-grained parallelism:

- ➜ Highly parallel applications
  - e.g., parallel execution of loop iterations
- ➜ Very frequent synchronisation
- ➜ Works only well on special hardware
  - vector computers, symmetric multithreading (SMT) hardware

Slide 18

Multiprocessor Scheduling

Multiprocessor Scheduling:

Which process should be run next and where?

We discuss:

- ➜ Tightly coupled multiprocessing
- ➜ Very coarse to medium grained parallelism
- ➜ Homogeneous systems (all processors have same specs, access to devices)

Design Issues:

- ➜ How to assign processes/threads to the available processors?
- ➜ Multiprogramming on individual processors?
- ➜ Which scheduling strategy ?
- ➜ Scheduling dependend processes

Slide 19

Assignment of Threads to Processors

- ➜ Treat processors as a pooled resource and assign threads to processors on demand
  - Permanently assign threads to a processor
    - Dedicate short-term queue for each processor
    - ✔ Low overhead
    - ✘ Processor could be idle while another processor has a backlog
  - Dynamically assign process to a processor
    - ✘ higher overhead
    - ✘ poor locality
    - ✔ better load balancing

Slide 20

Assignment of Threads to Processors

Who decides which thread runs on which processor?

Master/slave architecture:

- ➜ Key kernel functions always run on a particular processor
- ➜ Master is responsible for scheduling
- ➜ Slave sends service request to the master
- ✔ simple
- ✔ one processor has control of all resources, no synchronisation
- ✘ Failure of master brings down whole system
- ✘ Master can become a performance bottleneck

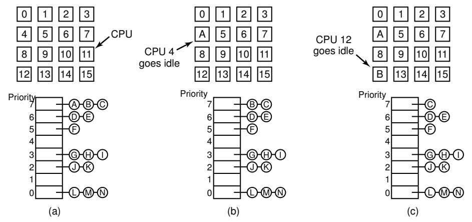
Slide 21

Peer architecture:

- ➜ Operating system can execute on any processor
- ➜ Each processor does self-scheduling
- ➜ Complicates the operating system
  - Make sure no two processors schedule the same thread
  - Synchronise access to resources
- ➜ Proper symmetric multiprocessing

Slide 22

LOAD SHARING: TIME SHARING



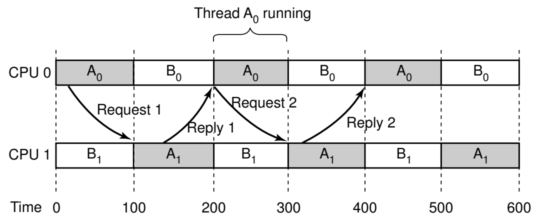
- ➜ Load is distributed evenly across the processors
- ➜ Use global ready queue
  - Threads are not assigned to a particular processor
  - Scheduler picks any ready thread (according to scheduling policy)
  - Actual scheduling policy less important than on uniprocessor
- ➜ No centralized scheduler required

Slide 23

Disadvantages of time sharing:

- ➜ Central queue needs mutual exclusion
  - Potential race condition when several CPUs are trying to pick a thread from ready queue
  - May be a bottleneck blocking processors
- ➜ Preempted threads are unlikely to resume execution on the same processor
  - Cache use is less efficient, bad locality
- ➜ Different threads of same process unlikely to execute in parallel
  - Potentially high intra-process communication latency

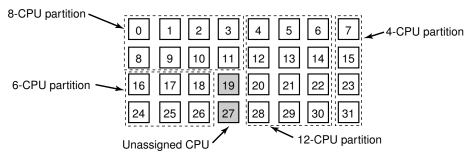
Slide 24



Slide 25

Load Sharing: Space Sharing

Scheduling multiple threads of same process across multiple CPUs



- ➜ statically assigned to CPUs at creation time (figure) or
- ➜ dynamic assignment using a central server

Slide 26

Gang Scheduling

Combined time and space sharing:

- ➜ 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
  - e.g., often need to synchronise with each other

Time slot	CPU 0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Slide 27

SMP Support in modern general purpose OS's[a]

- ➜ Solaris 8.0: up to 128
- ➜ Linux 2.4: up to 32
- ➜ Windows 2000 Data Center: up to 32
- ➜ OS/2 Warp: up to 64

SMP Scheduling in Linux 2.4:

- ➜ tries to schedule process on same CPU
- ➜ if the CPU busy, assigns it to an idle CPU
- ➜ otherwise, checks if process priority allows interrupt on preferred CPU
- ➜ uses spin locks to protect kernel data structures

[a](source `http://www.2cpu.com`)

Slide 28

Windows 2000 Scheduling

- ➜ priority driven, preemptive scheduling system
- ➜ if thread with higher priority becomes ready to run, current thread is preempted
- ➜ scheduled at thread granularity
- ➜ priorities: 0 (zero-page thread), 1-15 (variable levels), 16-31 (realtime levels — soft)
- ➜ each thread has a quantum value, clock-interrupt handler deducts 3 from running thread quantum
- ➜ default value of quantum: 6 Windows 2000 Professional, 36 on Windows 2000 Server
- ➜ most wait-operations result in temporary priority boost, favouring IO-bound threads

Slide 29

Realtime Systems

Overview:

- ➜ Real time systems
  - Hard and soft real time systems
  - Real time scheduling
  - A closer look at some real time operating systems

Slide 30

Real-Time Systems

What is a real-time system?

A real-time system is a system whose correctness includes its response time as well as its functional correctness.

What is a hard real-time system?

A real-time system with guaranteed worst case response times.

- ➜ Hard real-time systems fail if deadlines cannot be met
- ➜ Service of soft real-time systems degrades if deadlines cannot be met

Slide 31

Real-time systems:

- ➜ no clear separation
- ➜ system may meet hard deadline of one application, but not of other
- ➜ depending on application, time-scale may vary from microseconds to seconds
- ➜ most systems have some real-time requirements

Slide 32

Soft Real-time Applications:

- Many multi-media apps
- e.g., DVD or MP3 player
- Many real-time games, networked games

Hard Real-time Applications:

- Control of laboratory experiments
- Embedded devices
- Process control plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems

Slide 33

Hard real-time systems:

- often lack full functionality of modern OS
- secondary memory usually limited or missing
- data stored in short term or read-only memory
- no time sharing

Modern operating systems provide support for soft real-time applications

Hard real-time OS either specially tailored OS, modular systems, or customized version of general purpose OS.

Slide 34

CHARACTERISTICS OF REAL-TIME OPERATING SYSTEMS

Deterministic: How long does it take to acknowledge interrupt?

- Operations are performed at fixed, predetermined times or within predetermined time intervals
- Depends on
  - response time of system for interrupts
  - capacity of system
- Cannot be fully deterministic when processes are competing for resources
- Requires preemptive kernel

Responsive: How long does it take to service the interrupt?

- Includes amount of time to begin execution of the interrupt
- Includes the amount of time to perform the interrupt

Slide 35

CHARACTERISTICS OF REAL-TIME OPERATING SYSTEMS

User control: User has much more control compared to ordinary OS's

- User specifies priority
- Specify paging
- Which processes must always reside in main memory
- Disks algorithms to use
- Rights of processes

Reliability: Failure, loss, degradation of performance may have catastrophic consequences

- Attempt either to correct the problem or minimize its effects while continuing to run
- Most critical, high priority tasks execute

Characteristics of real-time operating systems

Slide 36

General purpose OS objectives like

- ➜ speed
- ➜ fairness
- ➜ maximising throughput
- ➜ minimising average response time

are not priorities in real time OS's!

Slide 37

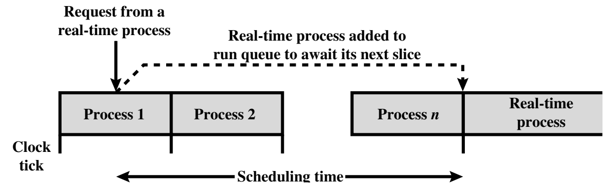
Features of real-time operating systems:

- ➜ Fast context switch
- ➜ Small size
- ➜ Ability to respond to external interrupts quickly
- ➜ Predictability of system performance!
- ➜ Use of special sequential files that can accumulate data at a fast rate
- ➜ Preemptive scheduling based on priority
- ➜ Minimization of intervals during which interrupts are disabled
- ➜ Delay tasks for fixed amount of time

Real-time scheduling

Slide 38

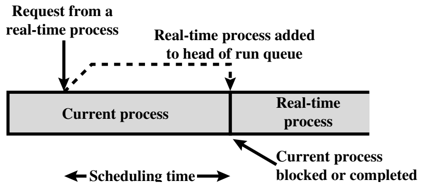
Preemptive round-robin:



Real-time scheduling

Slide 39

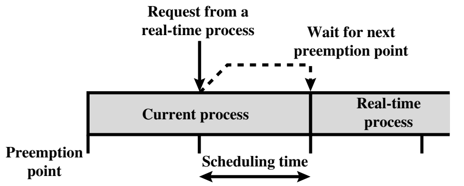
Non-preemptive priority:



Slide 40

REAL-TIME SCHEDULING

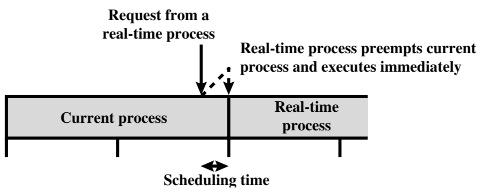
Preemption points:



Slide 41

REAL-TIME SCHEDULING

Immediate preemptive:



Slide 42

REAL-TIME SCHEDULING

Classes of Algorithms:

- ➜ Static table-driven
  - suitable for periodic tasks
  - input: periodic arrival, ending and execution time
  - output: schedule that allows all processes to meet requirements (if at all possible)
  - determines at which points in time a task begins execution
- ➜ Static priority-driven preemptive
  - static analysis determines priorities
  - traditional priority-driven scheduler is used
- ➜ Dynamic planning-based
  - feasibility to integrate new task is determined dynamically
- ➜ Dynamic best effort
  - no feasibility analysis
  - typically aperiodic, no static analysis possible
  - does its best, procs that missed deadline aborted

Slide 43

When are periodic events schedulable?

- ➜ P_i: period with which event i occurs
- ➜ C_i: CPU time required to handle event i

A set of events e_1 to e_m is schedulable if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Example:

- ➜ three periodic events with periods of 100, 200, and 500msecs
- ➜ require 50, 30 , and 100msec of CPU time
- ➜ schedulable?

$$\frac{50}{100} + \frac{30}{200} + \frac{100}{500} = 0.5 + 0.15 + 0.2 \leq 1$$

Slide 44

DEADLINE SCHEDULING

Current systems often try to provide real-time support by

- ➜ starting real time tasks are quickly as possible
- ➜ speeding up interrupt handling and task dispatching

Not necessarily appropriate, since

- ➜ real-time applications are not concerned with speed but with reliably completing tasks
- ➜ priorities alone are not sufficient

Slide 45

DEADLINE SCHEDULING

Additional information used:

- ➜ Ready time
  - sequence of times for periodic tasks, may or may not be known statically
- ➜ Starting deadline
- ➜ Completion deadline
- ➜ Processing time
  - may or may not be known, approximated
- ➜ Resource requirements
- ➜ Priority
- ➜ Subtask scheduler

Slide 46

DEADLINE SCHEDULING

Earliest deadline first strategy is provably optimal. It

- ➜ minimises number of tasks that miss deadline
- ➜ if there is a schedule for a set of tasks, earliest deadline first will find it

Earliest deadline first

- ➜ can be used for dynamic or static scheduling
- ➜ works with starting or completion deadline
- ➜ for any given preemption strategy
  - starting deadlines are given: nonpreemptive
  - completion deadline: preemptive

Slide 47

Two tasks:

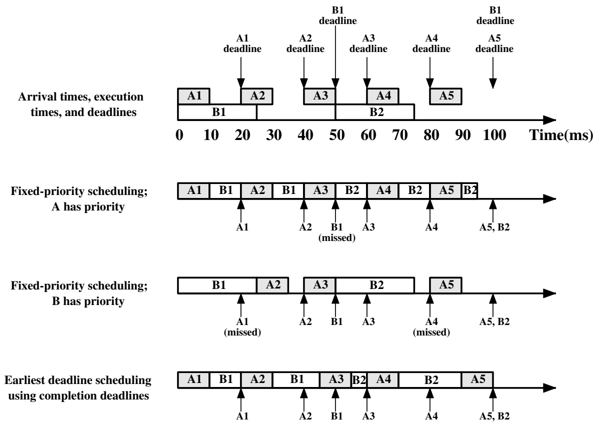
- ➜ Sensor A:
  - data arrives every 20ms
  - processing takes 10ms
- ➜ Sensor B:
  - data arrives every 50ms
  - processing takes 25ms

Scheduling decision every 10ms

Task	Arrival Time	Execution Time	Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
⋮	⋮	⋮	⋮
B(1)	0	25	50
B(2)	50	25	100
⋮	⋮	⋮	⋮

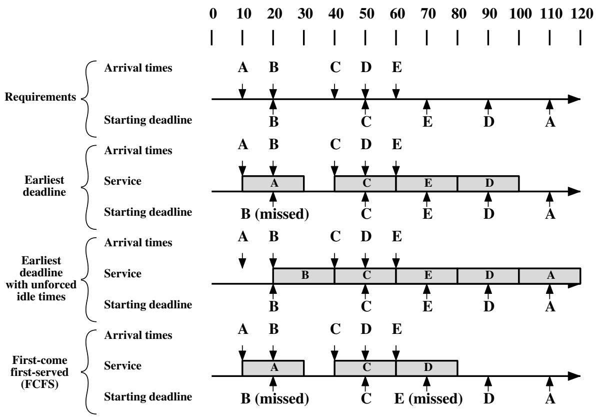
Slide 48

Periodic threads with completion deadline:



Slide 49

Aperiodic threads with starting deadline:



Slide 50

Rate Monotonic Scheduling

Works for processes which

- ➜ are periodic
- ➜ need the same amount of CPU time on each burst
- ➜ optimal static scheduling algorithm
- ➜ guaranteed to succeed if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m * (2^{\frac{1}{m}} - 1)$$

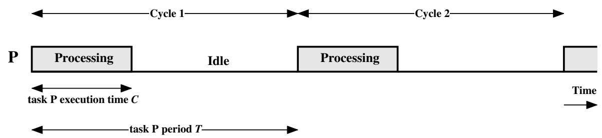
for m = 1,10,100,1000: 1, 0.7, 0.695, 0.693

Works by

- ➜ assigning priorities to threads on the basis of their periods
- ➜ highest-priority task is the one with the shortest period

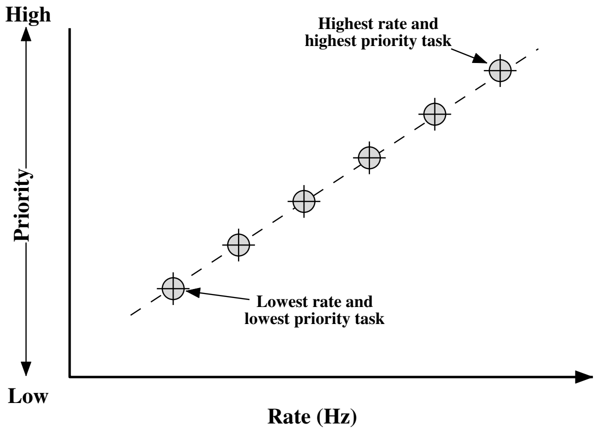
Slide 51

Periodic task timing diagram:



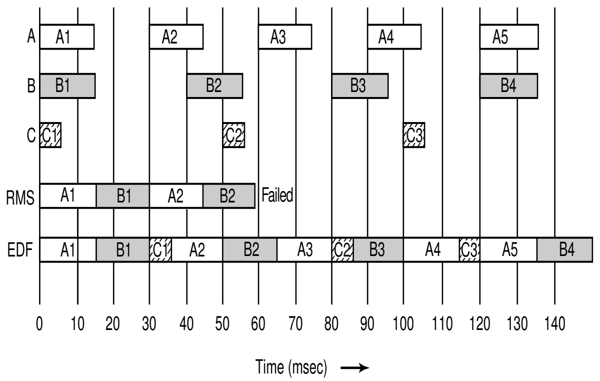
Slide 52

Task set with RMS:



Slide 53

- A: 15/30, B: 15/40, C: 5/50



Slide 54

Why use RMS?

Despite some obvious disadvantages of RMS over EDF, RMS is sometimes used

- ➜ it has a lower overhead
- ➜ simple
- ➜ in pratice, performance similar
- ➜ greater stability, predictability

Slide 55

Linux 2.4 Scheduling — Soft Real-time Support

- ➜ User assigns static priority to real time processes (1-99), never changed by scheduler
- ➜ Conventional processes have dynamic priority, always lower than real time processes
  - sum of base priority and
  - number of clock ticks left of quantum for current epoch
- ➜ Scheduling classes
  - 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
- ➜ Deadlines cannot be specified, no guarantees given
- ➜ Due to non-preemptive kernel, latency can be too high for real-time systems

Slide 56

Linux scheduling:

A	minimum
B	middle
C	middle
D	maximum

(a) Relative thread priorities

D ⟶ B ⟶ C ⟶ A ⟶

(b) Flow with FIFO scheduling

D ⟶ B ⟶ C ⟶ B ⟶ C ⟶ A ⟶

(c) Flow with RR scheduling

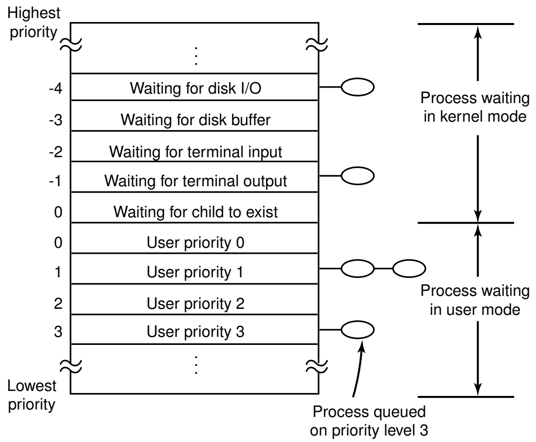
Slide 57

UNIX SVR4 Scheduling

- ➜ Highest preference to real-time processes
- ➜ Next-highest to kernel-mode processes
- ➜ Lowest preference to other user-mode processes
- ➜ Real time processes may block system services

Slide 58

SVR4 dispatch queues:

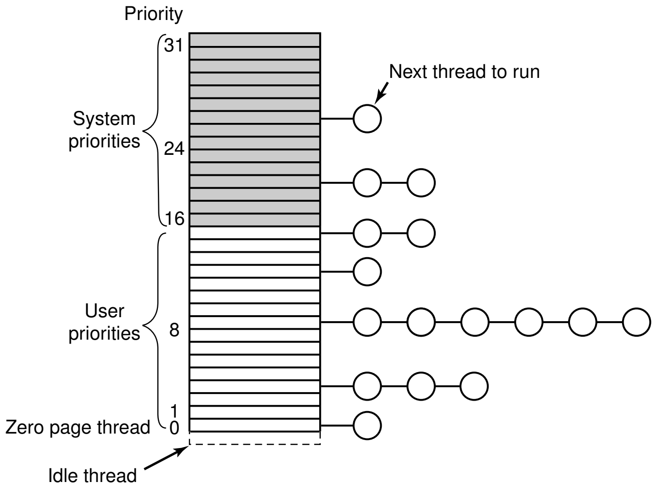


Slide 59

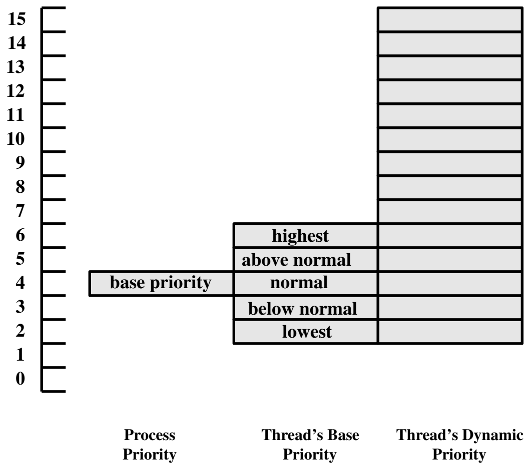
Windows 2000 Scheduling

- ➜ Priorities organized into two bands or classes
  - Real-time
  - Variable
- ➜ Priority-driven preemptive scheduler
- ➜ also, no deadlines, no guarantees

Slide 60



Slide 61

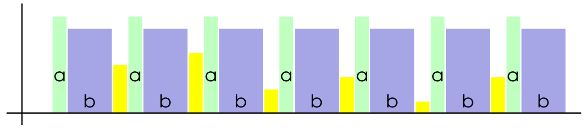


Slide 62

Problem:

- ➜ in real life applications, many tasks are not always periodic.
- ➜ static priorities may not work

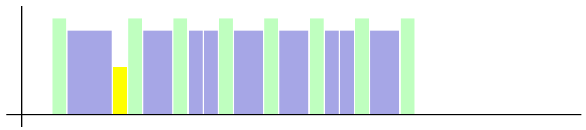
If real time threads run periodically with same length, fixed priority is no problem:



- a: periodic real time thread, highest priority
- b: periodic real time thread
- various different low priority tasks (e.g., user I/O)

Slide 63

But if frequency of high priority task increases temporarily, system may encounter overload:



- system not able to respond
- system may not be able to perform requested service

Slide 64

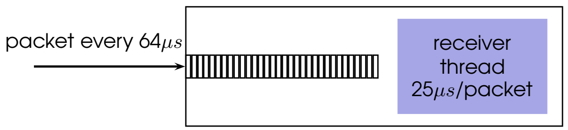
Example:

Network interface control driver, requirements:

- ➜ avoid if possible to drop packets
- ➜ definitely avoid overload

If receiver thread get highest priority permanently, system may go into overload if incoming rate exceeds a certain value.

- ➜ expected frequency: packet once every 64μs
- ➜ CPU time required to process packet: $25\mu s$
- ➜ 32-entry ring buffer, max 50% full



Slide 65

Sporadic Scheduling

POSIX standard to handle

- ➜ aperiodic or sporadic events
- ➜ with static priority, preemptive scheduler

Implemented in hard real-time systems such as QNX, some real-time versions of Linux, real-time specification for Java (RTSJ)(partially)

Can be used to avoid overloading in a system

Slide 66

Basic Idea: "simulation" of periodic behaviour of thread by assigning

- ➜ realtime priority: P_r
- ➜ background priority: P_b
- ➜ execution budget: E
- ➜ replenishment interval: R

to thread.

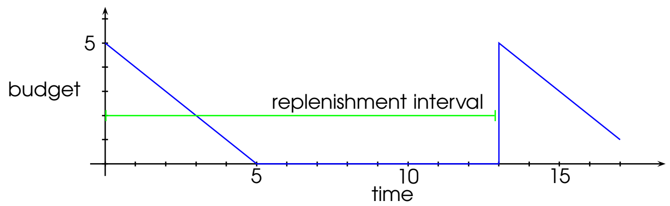
- ➜ Whenever thread exhausts execution budget, priority is set to background priority P_b
- ➜ When thread blocks after n units, n will be added to execution budget R units after execution started
- ➜ When execution budget is incremented, thread priority is reset to P_r

Slide 67

Example:

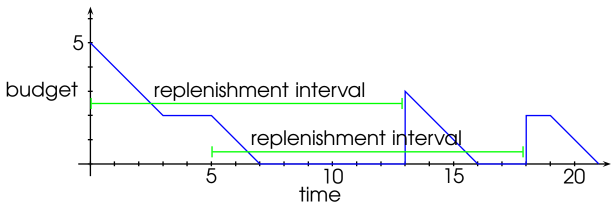
- ➜ execution budget: 5
- ➜ replenishment interval: 13

Thread does not block:



Slide 68

Thread blocks:

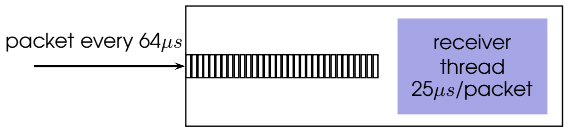


(0) exection starts, 1st replenishment interval starts
(3) thread blocks
(5) continues execution, 2nd replenishment interval starts
(7) budget exhausted
(13) budget set to 3, thread continues execution
(16) budget exhausted
(18) budget set to 2
(19) thread continues execution

Slide 69

Example: Network interface control Driver

- ➜ use expected incoming rate and desired max CPU utilisation of thread to compute execution budget and replenishment period
- ➜ if no other threads wait for execution, packets can be processed even if load is higher
- ➜ otherwise, packets may be dropped



- ➜ period: 64μs * 16 = 1024μs
- ➜ execution time: 25μs * 16 = 400μs
- ➜ CPU load caused by receiver thread: 400/1024 = 0.39, about 39%

Slide 70

HARD REAL TIME OS

We look at examples of two types of systems:

- ➜ configurable hard real time systems
  - system designed as real time OS from the start
- ➜ hard real-time variants of general purpose OSs
  - try to alleviate shortcomings of OS with respect to real time apps

Slide 71

REAL-TIME SUPPORT IN LINUX

- ➜ Scheduling:
  - POSIX `SCHED_FIFO`, `SCHED_RR`,
  - ongoing efforts to improve scheduler efficiency
- ➜ Virtual Memory:
  - no VM for real-time apps
  - `mlock()` and `mlockall()` to switch off paging
- ➜ Timer: resolution: 10ms, too coarse grained for real-time apps

Slide 72

High kernel latency in Linux

Possible solutions:

- Low Latency Linux
  - thread in kernel mode yields CPU
  - reduces size of non-preemptable sections
  - used in some real-time variants of Linux
- Preemptable Linux
  - kernel data protected using mutexes/spinlocks
- Lock breaking preemptable Linux
  - combination of previous two approaches

Slide 73

RTLinux

- abstract machine layer between actual hardware and Linux kernel
- takes control of
  - hardware interrupts
  - timer hardware
  - interrupt disable mechanism
- real time scheduler runs with no interference fron Linux kernel
- programmer must utilise RTLinux API for real time applications

Slide 74

QNX

- Microkernel based architecture
- POSIX standard API
- Modular — can be costumised for very small size (eg, embedded systems) or large systems
- Memory protection for user applications and os components

Scheduling:

- FIFO scheduling
- Round-robin
- Adaptive scheduling
  - thread consumes its timeslice, its priority is reduced by one
  - thread blocks, it immediately comes back to its base priority
- POSIX sporadic scheduling

Slide 75

Windows CE 3.0

Componentised OS designed for embedded systems with hard real-time support

- handles nested interrupts
- handles priority inversion based on priority inheritance

Offers

- guaranteed upper bound on high priority thread scheduling
- guaranteed upper bound on delay for interrupt service routines

Slide 76

Linux scheduling:

A	**minimum**
B	**middle**
C	**middle**
D	**maximum**

(a) Relative thread priorities

D ⟶ B ⟶ C ⟶ A ⟶

(b) Flow with FIFO scheduling

D ⟶ B ⟶ C ⟶ B ⟶ C ⟶ A ⟶

(c) Flow with RR scheduling

Slide 77

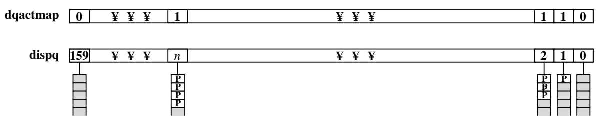
UNIX SVR4 Scheduling

➜ Highest preference to real-time processes

➜ Next-highest to kernel-mode processes

➜ Lowest preference to other user-mode processes

SVR4 dispatch queues:



Slide 78

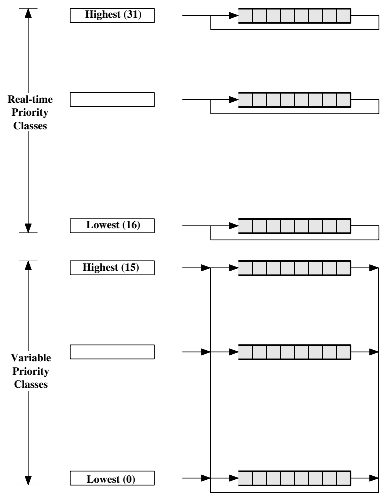
Windows 2000 Scheduling

➜ Priorities organized into two bands or classes

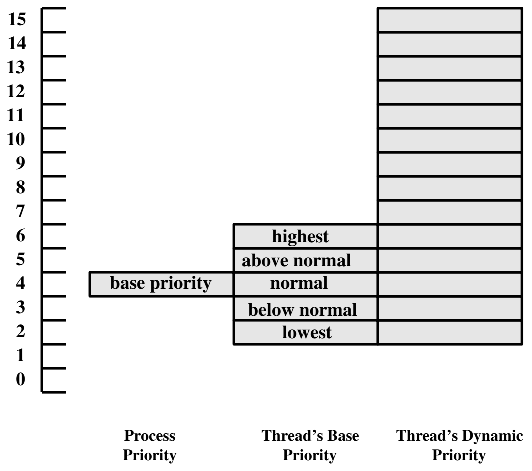
- Real-time
- Variable

➜ Priority-driven preemptive scheduler

Slide 79



Slide 80


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
highest
above normal
base priority
normal
below normal
lowest
Process Priority
Thread's Base Priority
Thread's Dynamic Priority