Slide 1

Scheduling

COMP3231 Operating Systems

2005 S2

- ➜ Real-time scheduling (continued)
- ➜ Hard real time systems
- ➜ Multiprocessor scheduling
- ➜ Case study 1: Windows 2000
- ➜ Case study 2: Linux 2.4 vs 2.6
- ➜ Discussion of existing hard real-time systems

Slide 2

ASSIGNMENT 2

- ➜ Simple file related system calls
- ➜ Process management system calls (fork)

Slide 3

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 4

SCHEDULING OF PERIODIC EVENTS

We know when a periodic event occurs and how long it will take to handle the event

- ➜ P_i: period with which event i occurs
- ➜ C_i: CPU time required to handle event i
- ➜ deadline: generally, event has to be processed before next event occurs

E.g., and event occurs every 50$msec$, requires 10ms of CPU time

- ➜ P_i: 50$msec$
- ➜ C_i: 10$msec$

Slide 5

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 500$msecs$
- ➜ require 50, 30 , and 100$msec$ of CPU time
- ➜ schedulable?

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

Slide 6

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 7

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 8

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 9

Two tasks:

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

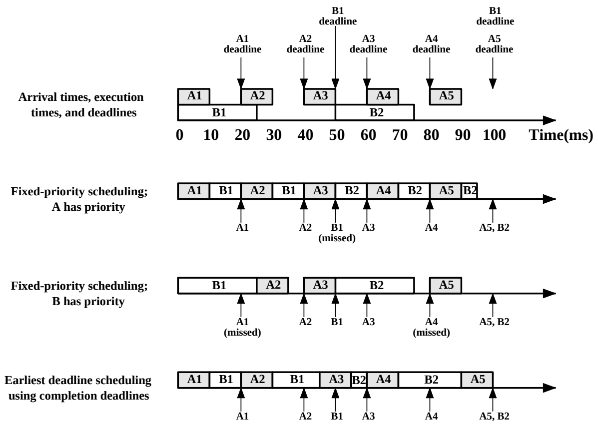
➜ Sensor B:
- data arrives every 50ms
- processing takes 25ms

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
⋮	⋮	⋮	⋮

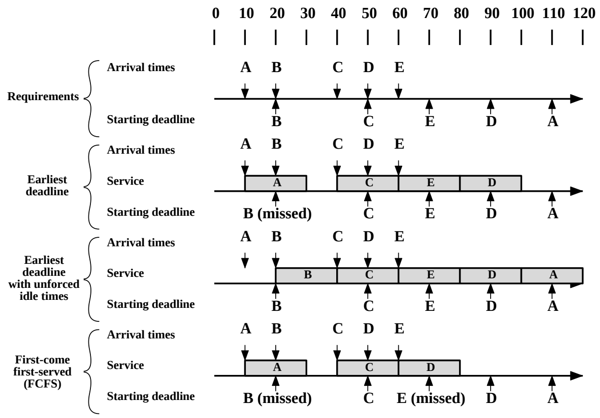
Slide 10

Periodic threads with completion deadline:



Slide 11

Aperiodic threads with starting deadline:



Slide 12

RATE MONOTONIC SCHEDULING

Works by

➜ assigning priorities to threads on the basis of their periods

➜ highest-priority task is the one with the shortest period

Works for processes which

➜ are periodic

➜ need the same amount of CPU time on each burst

➜ optimal static scheduling algorithm

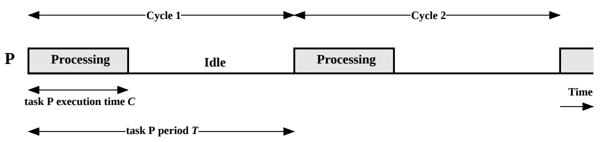
➜ guaranteed to succeed if

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

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

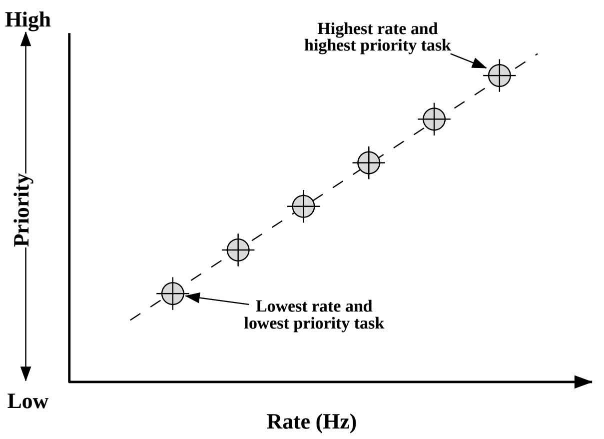
Slide 13

Periodic task timing diagram:



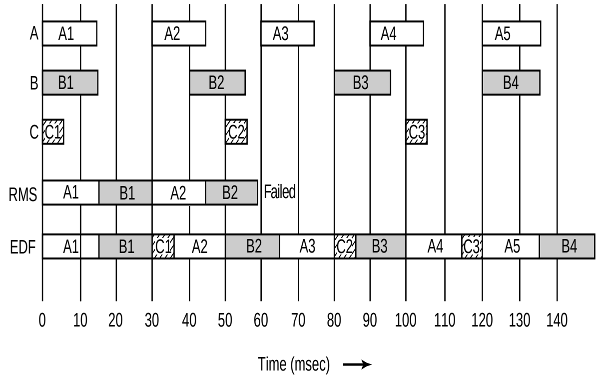
Slide 14

Task set with RMS:



Slide 15

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



Slide 16

Why use RMS?

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

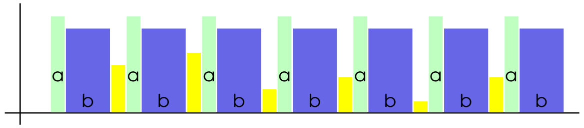
- ➜ it has a lower overhead
- ➜ simple
- ➜ in pratice, performance similar
- ➜ greater stability, predictability

Slide 17

Problem:

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

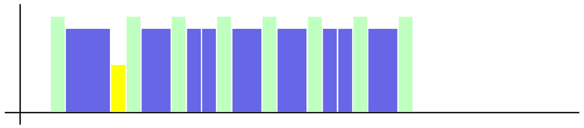
If real time threads run periodically with same length, fixed priority is no problem:



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

Slide 18

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

We need a scheduling strategy which can guarantee

- ➜ quality of service for "well-behaving" periodic real time tasks
- ➜ no system freeze if real-time tasks misbehave

Slide 19

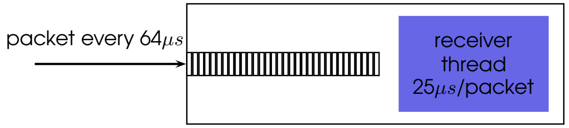
Example:[a]

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



[a] `embedded.com`, Scheduling Sporadic Events, Lonnie VanZandt

Slide 20

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 21

Basic Idea: enforcing periodic behaviour of thread by assigning

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

to thread.

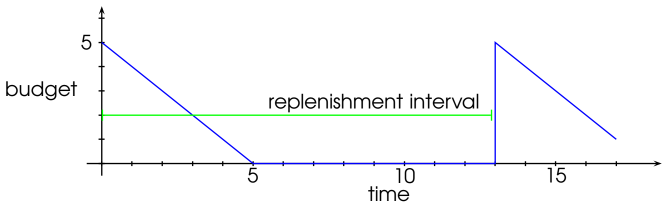
- ➜ Whenever thread exhausts execution budget, priority is set to background priority P_b
- ➜ When thread blocks after n units, n will be added to execution budget R units after execution started
- ➜ When execution budget is incremented, thread priority is reset to P_r

Slide 22

Example:

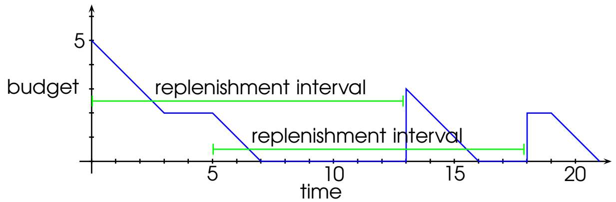
- ➜ execution budget: 5
- ➜ replenishment interval: 13

Thread does not block:



Slide 23

Thread blocks:

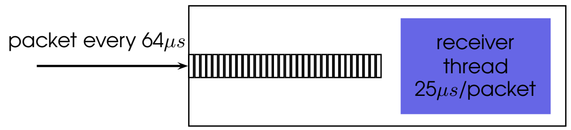


(0) exection starts, 1st replenishment interval starts
(3) thread blocks
(5) continues execution, 2nd replenishment interval starts
(7) budget exhausted
(13) budget set to 3, thread continues execution
(16) budget exhausted
(18) budget set to 2
(19) thread continues execution

Slide 24

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 25

Multi-Processor Systems

We have a look at different

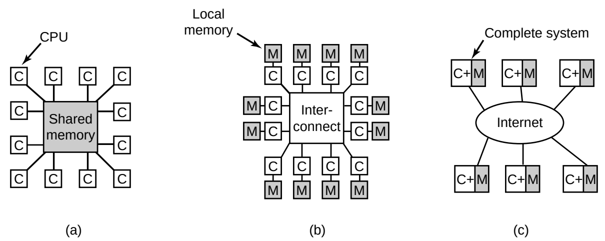
- ➜ applications
- ➜ architectures
- ➜ operating systems

for multi-processor systems

Slide 26

Multiprocessor Scheduling

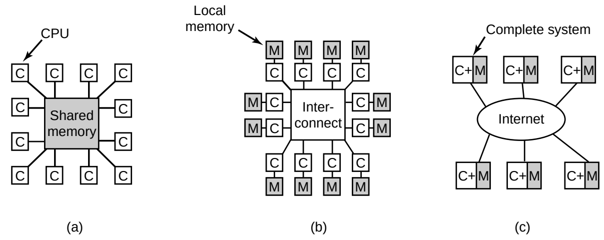
Classification of Multiprocessor Systems: What kind of systems and applications are there?



(a) Tightly coupled multiprocessing

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

Slide 27



(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 28

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 29

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 30

Multiprocessor Scheduling

Multiprocessor Scheduling:

Which process should be run next and where?

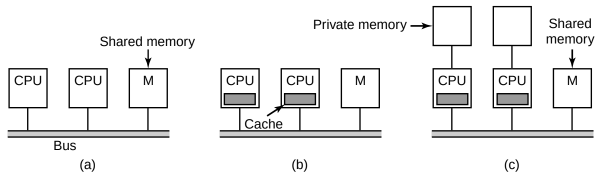
We discuss:

- ➜ Tightly coupled multiprocessing
- ➜ Very coarse to medium grained parallelism
- ➜ Shared-memory systems

Slide 31

Shared Memory Multiprocessor Hardware

UMA (uniform memory access) Bus-based SMP Architectures:



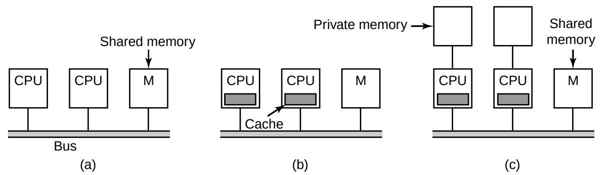
Without caching (a):

- ➜ limited by the bandwidth of the bus
- ➜ only feasible for a small number of CPUs

Slide 32

Shared Memory Multiprocessor Hardware

UMA Bus-based SMP Architectures:



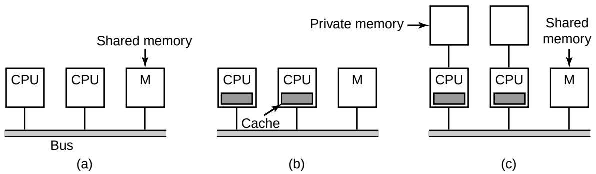
With caching (b):

- ➜ CPUs have their own cache
- ➜ each cache line is marked as read-only or read-write
- ➜ cache consistency an issue
- ➜ significantly reduces traffic on bus

Slide 33

Shared Memory Multiprocessor Hardware

UMA Bus-based SMP Architectures:



With caching and private memory (c):

- ➜ CPUs have their own cache and private memory
- ➜ shared memory only used to "communicate" — ie, shared variables, data structures
- ➜ requires compiler support

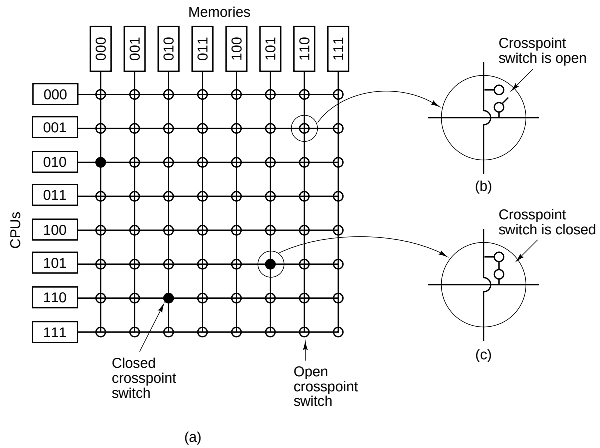
Slide 34

UMA Multiprocessor using Crossbar Switches:

- ➜ Even with cache and private memory, purely bus-based systems scale only to about 32 CPUs
- ➜ Crossbar switches dynamically set up connections between CPUs and different memory components

Slide 35

UMA Multiprocessor using Crossbar Switches:



Slide 36

UMA Multiprocessor using Crossbar Switches:

- ➜ Number of crosspoints grows quadratically
- ➜ Good solution for small to medium sized systems
- ➜ Many different, more complicated switching networks possible

Slide 37

NUMA Multiprocessors

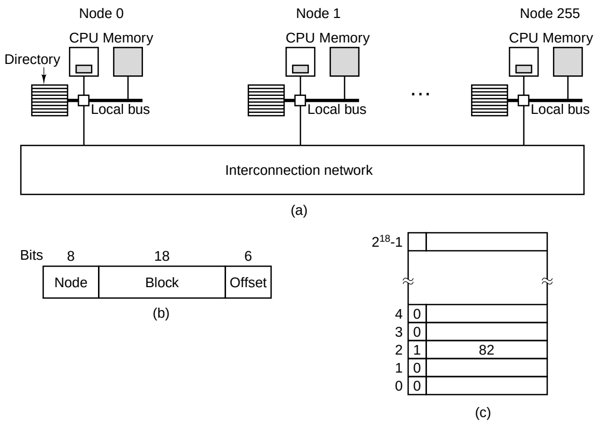
Uniform memory access time does not scale!

Characteristics of NUMA (non-uniform mem. access) systems:

- ➜ Single address space visible to all CPUs
- ➜ Access to remote memory via LOAD and STORE instructions
- ➜ Access to remote memory slower than to local memory

Cache coherent (CC-NUMA) and no caching (NC-NUMA) available

Slide 38



Slide 39

Shared-memory Multiprocessor Scheduling

Design Issues:

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

Slide 40

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 41

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

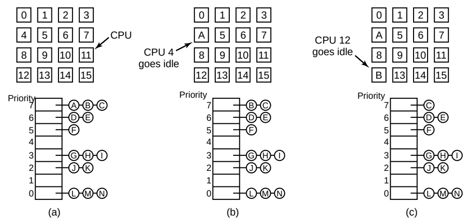
Slide 42

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 43

LOAD SHARING: TIME SHARING



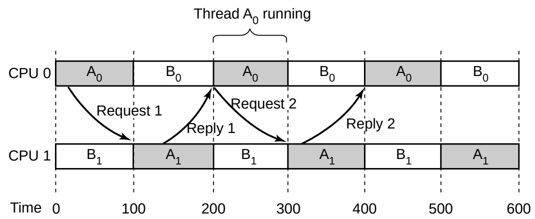
- ➜ Load is distributed evenly across the processors
- ➜ Use global ready queue
  - Threads are not assigned to a particular processor
  - Scheduler picks any ready thread (according to scheduling policy)
  - Actual scheduling policy less important than on uniprocessor
- ➜ No centralized scheduler required

Slide 44

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

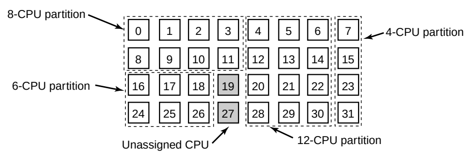
Slide 45



LOAD SHARING: SPACE SHARING

Slide 46

Scheduling multiple threads of same process across multiple CPUs



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

GANG SCHEDULING

Slide 47

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

SMP SUPPORT IN MODERN GENERAL PURPOSE OS'S

Slide 48

- ➜ Solaris 10.0: up to 256
- ➜ Linux 2.46: up to 32 (64)
- ➜ Windows Server 2003 Data Center: 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

Slide 49

WINDOWS 2000 CASE STUDY

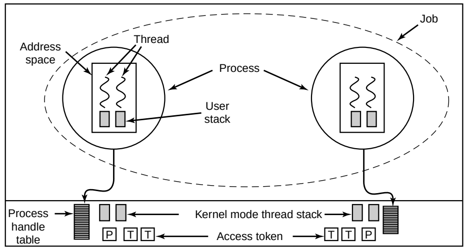
- ➜ Scheduling
- ➜ Virtual Memory Management

Slide 50

WINDOWS 2000 SCHEDULING

- ➜ priority driven, preemptive scheduling system
- ➜ SMP: set of processors a thread can run on may be restricted (processor affinity)
- ➜ scheduling decision may be necessary when
  - a new thread has been created
  - a thread released from wait state
  - time quantum of a thread is exceeded
  - a thread's priority changes
  - processor affinity of a thread changes
- ➜ no dedicated scheduler thread — each thread chooses successor while running in kernel mode

Slide 51



Slide 52

- ➜ if thread with higher priority becomes ready to run, current thread is preempted
- ➜ scheduled at thread granularity
  - processes with many threads get more CPU time

Slide 53

WINDOWS 2000 SCHEDULING

- ➜ Windows 2000 priority levels:
  - ➜ 0 (zero-page thread)
  - ➜ 1-15 (variable levels)
  - ➜ 16-31 (realtime levels — soft)
- ➜ Win32 API priority classes:
  - Real-time
  - High
  - Above Normal
  - Normal
  - Below Normal
  - Idle

  and relative priorities within these classes:
  - Time-critical
  - High
  - ...

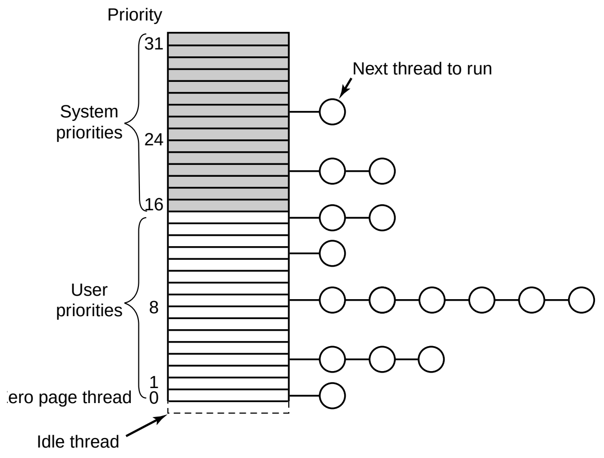
Slide 54

- ➜ 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
- ➜ priority of a user thread can be raised (eg, after waiting for a semaphore etc), but never above 15
- ➜ no adjustments to priorities above 15

Slide 55

		Win32 process class priorities					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

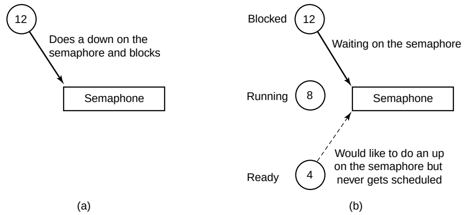
Slide 56



Slide 57

Dealing with Priority Inversion in Windows 2000

Example: Producer-Consumer problem



- ➜ System keeps track of how long a ready-thread has been in the queue
- ➜ if waiting time exceeds threshold, priority boosted to 15

Slide 58

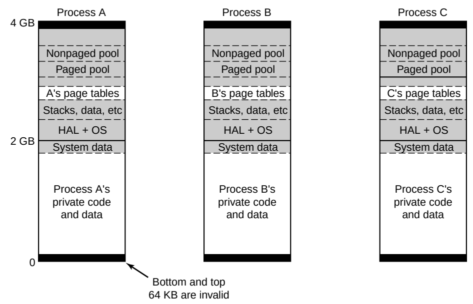
Win32 Scheduling-related API

- ➜ Suspend/ResumeThread
- ➜ Get/SetPriorityClass (base priority)
- ➜ Get/SetPriority (relative priority)
- ➜ Get/SetProcessAffinityMask
- ➜ Get/SetThreadAffinityMask
- ➜ Get/SetPriorityBoost
- ➜ SetThreadIdealProcessor
- ➜ SwitchtoThread
- ➜ Sleep

Slide 59

Memory Management

- ➜ Every process has 4GB virtual address space



Slide 60

Memory Management

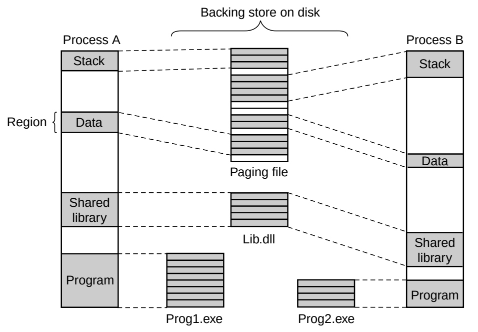
- ➜ A page can be in one of three states:
  - free: not in use, reference to such a page causes a page fault
  - committed: data or code mapped onto the page. If not in main memory, page fault occurs, OS swaps page from disk. No fixed mapping to swap space
  - reserved: not yet mapped, but also not available. Used, for example, to implement thread stacks

  and has the usual readable, writable, executable attributes

Slide 61

MEMORY MAPPED FILES

- ➜ memory mapped filed supported
- ➜ processes may share maps, updates visible to all processes
- ➜ if file is opened for normal reading, current version is shown
- ➜ copy-on-write (cow)



Slide 62

WIN32 API FOR VM

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file mapping object

Slide 63

MEMORY MANAGEMENT

- ➜ Unlike scheduler, who deals with threads and ignores processes, MM deals only with processes
- ➜ Mapping of pages happens in the usual way, two-level page table used
- ➜ In case of a page fault, a block of consecutive pages are read

Slide 64

PAGE REPLACEMENT ALGORITHM

Working Set:

- ➜ set of pages of a process which have been mapped into memory
- ➜ described by (process specific) max and min size
- ➜ all processes start with the same limits, but may change over time
- ➜ not hard bounds
- ➜ if page fault occurs and process has
  - less than min pages: add page
  - between min and max pages: add page if memory is not scarce
  - more than max pages: evict page from working set
- ➜ Working set of system is handled separately.

Slide 65

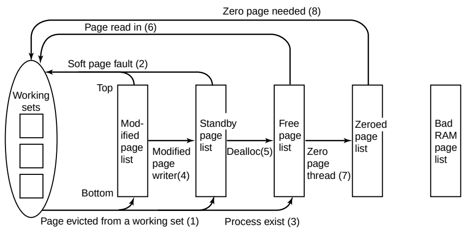
DAEMON THREADS TO MANAGE WORKING SETS

➜ Balance Set Manager: checks whether there are enough free pages, starts Working Set Manager if required

➜ Working Set Manager: searches for processes which have exceeded their maximum, didn't have page faults recently and removes some of their pages

Slide 66

A closer look at the free frames management:



There are actually four separate lists which contain free frames

① Modified Pages
② Standby Pages
③ Free Pages
④ Zeroed Pages

Slide 67

A closer look at the free frames management: