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

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

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 $50\text{msec}$, requires $10\text{ms}$ of CPU time

- $P_i$: $50\text{msec}$
- $C_i$: $10\text{ms}$
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 msecs of CPU time
- schedulable?

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

**Deadline Scheduling**

Current systems often try to provide real-time support by

- starting real time tasks as 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

**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

**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
Two tasks:

- **Sensor A:**
  - Data arrives every 20 ms
  - Processing takes 10 ms

- **Sensor B:**
  - Data arrives every 50 ms
  - Processing takes 25 ms

Scheduling decision every 10 ms

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1)</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>A(2)</td>
<td>20</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>A(3)</td>
<td>40</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>B(1)</td>
<td>0</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>B(2)</td>
<td>50</td>
<td>25</td>
<td>100</td>
</tr>
</tbody>
</table>

**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

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m \times (2^m - 1)
\]

For \( m = 1, 10, 100, 1000: 1, 0.7, 0.695, 0.693 \)
Periodic task timing diagram:

Slide 13

![Periodic task timing diagram](image)

Task set with RMS:

Slide 14

![Task set with RMS](image)

Task set with RMS:

Slide 15

![Task set with RMS](image)

**Why use RMS?**

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

- It has a lower overhead
- Simple
- In practice, performance similar
- Greater stability, predictability

Slide 16

![Why use RMS?](image)
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)

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

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: 25μs
- 32-entry ring buffer, max 50% full

Why use RMS?

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
**Basic Idea:** enforcing periodic behaviour of thread by assigning
- realtime priority: $P_r$
- background priority: $P_b$
- execution budget: $E$
- replenishment interval: $R$

**Slide 21**
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 $P_r$.

**Example:**
- execution budget: 5
- replenishment interval: 13

**Slide 22**
Thread does not block:

**Slide 23**
Thread blocks:

**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.

**Slide 25**
- period: $64\mu s \times 16 = 1024\mu s$
- execution time: $25\mu s \times 16 = 400\mu s$
- CPU load caused by receiver thread: $400/1024 = 0.39$, about 39%.
**Multi-Processor Systems**

We have a look at different
- applications
- architectures
- operating systems
for multi-processor systems

**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

(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

**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
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

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

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

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:**

- (a) CPU → CPU → M → Bus → CPU → CPU → M → Shared memory
- (b) CPU → CPU → Cache → Private memory
- (c) CPU → CPU → M

With caching and private memory (c):
- CPUs have their own cache and private memory
- Shared memory only used to "communicate" - i.e., shared variables, data structures
- Requires compiler support

**Slide 35**

**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 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 36**

**UMA Multiprocessor using Crossbar Switches:**

- Crosspoint switch is open
- Crosspoint switch is closed

Memories:
- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

Closed crosspoint switch
Open crosspoint switch

(a)
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 37

Slide 38

Directory
Node 0 CPU Memory
Local bus
Node 1 CPU Memory
Local bus
Node 255 CPU Memory
Local bus
Interconnection network

(a)

Bits 8 18 6
Node Block Offset
(b)

Slide 39

Slide 40

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 dependent processes

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
  - Dynamically assign thread to a processor
    - Higher overhead
    - Poor locality
    - Better load balancing

ASSIGNMENT OF THREADS TO PROCESSORS

SHARED-MEMORY MULTIPROCESSOR SCHEDULING
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

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

Load 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

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**

**Thread A0 running**

- CPU 0
- CPU 1
- Time 0 100 200 300 400 500 600
- Request 1
- Request 2
- Reply 1
- Reply 2

**Slide 46**

**Load Sharing: Space Sharing**

- Scheduling multiple threads of the same process across multiple CPUs

- Statically assigned to CPUs at creation time (figure) or
- Dynamic assignment using a central server

**Slide 47**

**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

**Slide 48**

**SMP Support in Modern General Purpose OS's**

- Solaris 10.0: up to 256
- Linux 2.4.6: 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
Windows 2000 Case Study

Slide 49
- Scheduling
- Virtual Memory Management

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 50

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
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
- ...

- 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
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.

WIN32 SCHEDULING-RELATED API

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

MEMORY MANAGEMENT

- Every process has 4GB virtual address space.

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.
Memory Mapped Files

- Memory mapped file 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)

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

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.

Win32 API for VM

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

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

Page frame database

<table>
<thead>
<tr>
<th>State</th>
<th>WS</th>
<th>Other</th>
<th>FT</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page tables

- List headers
  - Modified
  - Standby
  - Free
  - Zeroed

Page tables