Mutliprocessor and Real-Time Scheduling

Julien Schmaltz

Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands
julien@cs.ru.nl

June 15, 2008
Part I

Multiprocessor Scheduling
Multiprocessor Systems

- **Loosely coupled multiprocessor, or cluster**: autonomous systems, each processor has its own main memory and I/O channels
- **Functionally specialized processors**: e.g. I/O processor. Slaves used by a master (e.g. general-purpose CPU).
- **Tightly couple multiprocessing**: processors share a common main memory, they are under the control of an operating system

Chapter 10.1 deals with the last category of systems
Synchronization Granularity

- **Fine**: Parallelism inherent in a single instruction stream (sync. interval <20 instructions)
- **Medium**: Parallel processing or multitasking within a single application (sync. interval 20–200 inst.)
- **Coarse**: Multiprocessing of concurrent processes in a multiprogramming environment (sync. inter. 200–2000)
- **Very Coarse**: Distributed processing across network nodes to form a single computing environment (sync. inter. 2000–1M)
- **Independent**: Multiple unrelated processes (see previous lecture)

Granulatity = important parameter when designing selection functions

Mainly consider (Coarse) **Medium**
Basic Problem

- Given a number of threads (or processes), and a number of CPUs, assign threads to CPUs
- Same issues as for uniprocessor scheduling:
  - Response time, fairness, starvation, overhead, ...
- New issues:
  - Ready queue implementation
  - Load balancing
  - Processor affinity
Single Shared Ready Queue

- Global queue
- CPU picks one process when ready

**Pros**
- Queue can be reorganized (e.g. priorities, ... see previous lecture)
- Load evenly distributed

**Cons**
- Synchronization (mutual exclusion of queue accesses)
- Overhead (caching, context switch, ...)

**CPU0**
**CPU1**
**CPU2**
**CPU3**
Per-CPU Ready Queue

- **Pros**
  - Simple/ no synchronization needed
  - Strong affinity

- **Cons**
  - Where put new threads?
  - Load balancing

One queue per CPU
Load Balancing

- Try to keep processors as busy as possible
- Global approaches
  - Push model – Kernel daemon checks queue lengths periodically, moves threads to balance
  - Pull model – CPU notices its queue is empty and steals threads from other queues
  - Do both!

- Load sharing
- Gang-scheduling
- Dedicated processor assignment
- Dynamic scheduling
Processor Affinity

- States of executed threads in the cache of the CPU
- Repeated execution on the same CPU may reuse the cache
- Execution on a different CPU:
  - Requires to load state in the cache
- Try to keep thread–CPU pairs constant

1 thread bound to 1 processor
Processor Affinity

- States of executed threads in the cache of the CPU
- Repeated execution on the same CPU may reuse the cache
- Execution on a different CPU:
  - Requires to load state in the cache
- Try to keep thread–CPU pairs constant

1 thread bound to 1 processor
Processor Affinity

- States of executed threads in the cache of the CPU
- Repeated execution on the same CPU may reuse the cache
- Execution on a different CPU:
  - Requires to load state in the cache
- Try to keep thread–CPU pairs constant

Previous state stored in cache
Processor Affinity

- States of executed threads in the cache of the CPU
- Repeated execution on the same CPU may reuse the cache
- Execution on a different CPU:
  - Requires to load state in the cache
- Try to keep thread–CPU pairs constant

No (less) cache misses
Processor Affinity

- States of executed threads in the cache of the CPU
- Repeated execution on the same CPU may reuse the cache
- Execution on a different CPU:
  - Requires to load state in the cache
- Try to keep thread–CPU pairs constant

![Diagram showing thread-CPU pairs and cache loading]

CPU0

CPU1

Need to load cache!
Job Scheduling

- Job = a set of processes (or threads) that work together (to solve some problem or provide some service)
- Performance depends on scheduling of job components
- Two major strategies
  - Space sharing
  - Time sharing
Why it matters?

!!! Threads in a job are not independent !!!

- Synchronize on shared variables
- Cause/effect relationship
  - e.g. Consumer/Producer problem
  - Consumer is waiting for data but Producer which is not running
- Synchronizing phases of execution (barriers)
  - Entire job proceeds at pace of slowest thread
Space Sharing

- Define **groups of processors**
  - Fixed, variable, or adaptive
- Assign **one job to one group** of processors
  - Ideal: one CPU/thread in job
- **Pros**
  - Low context switch
  - Strong affinity
  - All runnable threads execute at same time
- **Cons**
  - One partition may have pending threads/jobs while another is idle
  - Hard to deal with dynamically-changing job sizes
**Time Sharing**

- Divide **one processor** time between **several jobs**
- Each CPU may execute threads from different jobs
  - Key: keep awareness of jobs

**Pros**
- Allow gang-scheduling
- Easier to deal with dynamically-changing

**Cons**
- Filling available CPU slots with runnable jobs equiv. to the bin packing problem
- Heuristic based – (bad worst case)
Gang-Scheduling (1)

- CPUs perform context switch together
- CPUs execute threads from different jobs (time sharing)
- Thread of one job bound to one processor (space sharing)
- Strong affinity

First execute blue for $t_b$ seconds, which is enough to complete the job

Green bound to CPU2 and CPU3, Pink to CPU0 and CPU2
Gang-Scheduling (1)

- CPUs perform context switch together
- CPUs execute threads from different jobs (time sharing)
- Thread of one job bound to one processor (space sharing)
- Strong affinity

First execute blue for $t_b$ seconds, which is enough to complete the job

Green bound to CPU2 and CPU3, Pink to CPU0 and CPU2
Gang-Scheduling (1)

- CPUs perform context switch together
- CPUs execute threads from different jobs (time sharing)
- Thread of one job bound to one processor (space sharing)
- Strong affinity

Then, execute magenta for $t_m$ seconds, and put magenta back in the queue

Green bound to CPU2 and CPU3, Pink to CPU0 and CPU2
Gang-Scheduling (1)

- CPUs perform context switch together
- CPUs execute threads from different jobs (time sharing)
- Thread of one job bound to one processor (space sharing)
- Strong affinity

Gang-Scheduling:

- Red job: execute on CPUs 0 and 3
- Green job: execute on CPUs 2 and 3
- Pink job: execute on CPUs 0 and 2

Execute red job:

Green bound to CPU2 and CPU3, Pink to CPU0 and CPU2
Gang-Scheduling (1)

- CPUs perform context switch together
- CPUs execute threads from different jobs (time sharing)
- Thread of one job bound to one processor (space sharing)
- Strong affinity

 Execute green job, pink job blocked by green

Green bound to CPU2 and CPU3, Pink to CPU0 and CPU2
Gang-Scheduling (1)

- CPUs perform context switch together
- CPUs execute threads from different jobs (time sharing)
- Thread of one job bound to one processor (space sharing)
- Strong affinity

Green bound to CPU2 and CPU3, Pink to CPU0 and CPU2

Execute pink job
Gang-scheduling (2)

- CPUs perform context switch together
- Execute only all threads of one job
- Weak affinity but strong usage

No fixed thread/processor assignment
Gang-scheduling (2)

- CPUs perform context switch together
- Execute only all threads of one job
- Weak affinity but strong usage

No fixed thread/processor assignment

Execute blue.
Gang-scheduling (2)

- CPUs perform context switch together
- Execute only all threads of one job
- Weak affinity but strong usage

No fixed thread/processor assignment
Gang-scheduling (2)

- CPUs perform context switch together
- Execute only all threads of one job
- Weak affinity but strong usage

No fixed thread/processor assignment

Not enough CPUs for green.
Gang-scheduling (2)

- CPUs perform context switch together
- Execute only all threads of one job
- Weak affinity but strong usage

No fixed thread/processor assignment

Enough CPUs for green AND pink

[Diagram showing CPU configuration with groups of threads]

Julien Schmaltz
Multiprocessor and Real-Time Scheduling
Dynamic Scheduling

- Number of threads can be altered dynamically by applications
- O/S 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 processor (i.e., to all waiting new arrivals)
Part II

Real-Time Scheduling
Correct executions depend not only on computation results but also on the time when the results are available.

“Events occur in real-time”
- Tasks reaction/control w.r.t. events that take place in the outside world
- Dynamic process, talks must keep up with these events
Real-Time Systems (2)

- Control of laboratory experiments
- Process control in industrial plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems
Real-Time Tasks

- Tasks have deadlines (to start or finish)
- Hard vs. soft deadlines
  - **hard real-time tasks** must meet their deadlines
    - Space shuttle rendez-vous, Nuclear powerplants, ...
  - **soft real-time tasks** may not meet their deadline, this has no “dramatic” consequences
    - Execution of the tasks even after its deadline!
- Periodic vs. aperiodic
  - **Aperiodic**: fixed deadline that must (or may) be met.
  - **Periodic**: “once per period $T$” or “exactly $T$ units apart”
Determinism

- Operations are performed at fixed, predetermined times, or within predetermined time intervals
- Concerned with maximum delay before interrupt acknowledgment and the capacity to handle all the requests within the required time
Responsiveness
- Delay after acknowledgment to service the interrupt
- Includes time to begin the execution of the interrupt
- Includes time to perform the interrupt
- Effect of interrupt nesting

Response time to external events = determinism + responsiveness
User control

- User specified priorities
- User specified paging
- What processes must always reside in main memory
- User specified disk algorithms
- User specified processes rights
Reliability
- Degradation of performance may have catastrophic consequences (e.g. nuclear meltdowns)

Fail-soft operation
- Fail in such a way as to preserve capability and data
- Stability: deadlines of most critical tasks always met
Features of RTOS (1)

- Fast process or thread switch
- Small size (minimal functionality)
- Quick response to interrupts
- Multitasking with interprocess communication tools such as semaphores, signals, and events
Features of RTOS (2)

- Use of specific 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 a fixed amount of time
- Special alarms and timeouts
Scheduling (1)

- **Round-robin preemptive scheduling**
  - Request from RT process
  - RT process added to run queue
  - RT process to run queue to await next time slice
  - RT to run queue to await next time slice
  - Scheduling time unacceptable for RT apps

- **Priority-driven non-preemptive scheduler**
  - Request from RT process
  - RT process to head of run queue
  - RT process to head of run queue
  - P1 blocked or completed
  - Issue if P1 low prior. and slow
Scheduling (2)

- Priority-driven, preemptive at preemption points
  - Request from RT process
  - RT preempts current process
  - Wait until next preemption point
  - Which may come before end of P1

- Immediate preemptive
  - Request from RT process
  - RTP preempts P1
  - RTP executes immediately
  - RTP is executed immediately
Real-Time Scheduling

- **Static table-driven**
  - Static analysis if feasible schedules
  - Determines at run time when a task starts

- **Static priority-driven preemptive**
  - Analysis used to assign priority to tasks
  - Traditional priority-driven preemptive scheduler

- **Dynamic planning-based**
  - Feasibility determined at run time

- **Dynamic best effort**
  - No feasibility analysis is done
Deadline Scheduling

- Important metrics: meet deadlines (not too early, not too late) rather than speed
- Information used:
  - Ready time
  - Starting time
  - Completion deadline
  - Processing time
  - Resource requirements
  - Priority
  - Subtask structure
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Fixed priority: A has priority

- Sensor A: 10ms, every 20ms
- Sensor B: 25ms, every 50ms

A runs for 10ms
B interrupted by A
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Fixed priority: A has priority

Sensor A: 10ms, every 20ms
Sensor B: 25ms, every 50ms

A runs for 10ms
B interrupted by A
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Fixed priority: A has priority

![Scheduler Diagram]

- Sensor A: 10ms, every 20ms
- Sensor B: 25ms, every 50ms

- A runs for 10ms
- B interrupted by A
- Deadline of B missed!
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Fixed priority: B has priority

Sensor A: 10ms, every 20ms
Sensor B: 25ms, every 50ms

B has priority
Deadline of A missed!


Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Earlier Deadline First (EDF)

- Sensor A: 10ms, every 20ms
- Sensor B: 25ms, every 50ms

A deadline before B deadline
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Earlier Deadline First (EDF)

Sensor A: 10ms, every 20ms
Sensor B: 25ms, every 50ms

B interr. because of A deadline
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Earlier Deadline First (EDF)

Sensor A: 10ms, every 20ms
Sensor B: 25ms, every 50ms

B completes because earliest deadline
Example

- Collecting data from sensors A and B
- Scheduling decision every 10ms and based on completion deadlines
- Earlier Deadline First (EDF)

Sensor A: 10ms, every 20ms
Sensor B: 25ms, every 50ms

Last B and A complete
Rate-Monotonic Scheduling (RMS)

- Proposed by Liu and Layland 1973
- Use frequency to assign priority
- Highest priority to shortest period
- Priority is a monotonic function of the period
- Static priority

Example

Previously we had $T_A = 20ms$, $C_A = 10ms$ and $T_B = 50ms$, $C_B = 25ms$, so RMS would choose A. Issue:

$$\frac{C_A}{T_A} + \frac{C_B}{T_B} = 1$$
Liu and Layland Result

- One cannot use more than the full processor time
  \[ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1 \]

- For RMS, the following condition is sufficient for schedulability
  \[ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right) \]

Example

<table>
<thead>
<tr>
<th>name</th>
<th>C</th>
<th>T</th>
<th>U</th>
<th>Bound = 3 \cdot \left(2^{\frac{1}{3}} - 1\right) = 0.779</th>
<th>Sum of ( U_i ) = 0.753</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>20</td>
<td>100</td>
<td>0.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_2 )</td>
<td>40</td>
<td>150</td>
<td>0.267</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P_3 )</td>
<td>100</td>
<td>350</td>
<td>0.286</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Interrupt based system design is challenging
- Priority assignment: a **Black Art**. Great body of literature; many negative results.
- We presented 2 positive results for important special cases:
  - RATE MONOTONIC (RMS) Liu&Layland, 1973
    - Case: Periodic, static priority
    - Rule: priority based on frequency
    - Not always applicable (auto impact sensor gets LOWEST priority)
  - EARLIER DEADLINES (EDF) Knuth, Mok *et al.* ’70
    - Case: deadline specified at each request
    - Rule: schedule earliest deadline first
    - Optimal for 1 processor; no extension to optimal N-processor scheme