Real-Time Systems

Marcus Völp

Hard Real-Time

Multiprocessor Scheduling
Outline

- Introduction
- Terminology, Notation and Assumptions
- Anomalies + Imp possibility Results
- Partitioned Scheduling (no migration)
- Global Scheduling (task- / job-level migration)
  - G-FP
  - G-EDF
- Optimal MP Scheduling
- MP – Resource Access Protocols
- Open Research Issues
Lessons Learned From UP

- Liu-Layland Criterion for fixed task priority algorithms:
  - can schedule any workload up to a utilization of $U_{RMS}(n) = n^{\frac{n}{\sqrt{2}}} - 1 \leq 0.693$
  - scheduling of workloads with higher utilization not guaranteed

- fixed job priority algorithms are optimal: (e.g., EDF)

- there are optimal greedy algorithms
  - with a single measure characterizing the “importance” of a job (e.g., time to deadline, laxity, ...)

- all pre-emptive FTP, FJP algorithms are predictable
  - response times cannot increase when decreasing execution times

- all pre-emptive FTP algorithms and EDF are sustainable
  - no period / deadline anomalies

- simultaneous release is critical instance

- response times depend on set but not on order of high-prio. tasks
Taxonomy of Multiprocessor Scheduling

- Two problems to solve:
  - Priority Problem: When to run a given job of the workload?
    - fixed task priority (e.g., RMS, ...)
    - fixed job priority (e.g., EDF, ...)
    - dynamic job priority (e.g., least laxity, PFAIR, DP-FAIR, ...)

Real-Time Systems, Multiprocessor Scheduling / Marcus Völp
Taxonomy of Multiprocessor Scheduling

- Two problems to solve:
  - Priority Problem: When to run a given job of the workload?
    - fixed task priority (e.g., RMS, ...)
    - fixed job priority (e.g., EDF, ...)
    - dynamic job priority (e.g., least laxity, PFAIR, DP-FAIR, ...)
  - Allocation Problem: Where to run this job?
    - no migration
    - task-level migration (no migration of running jobs)
    - job-level migration (migrate also running jobs)
Taxonomy of Multiprocessor Scheduling

- Two problems to solve:
  - Priority Problem: When to run a given job of the workload?
    - fixed task priority (e.g., RMS, ...)
    - fixed job priority (e.g., EDF, ...)
    - dynamic job priority (e.g., least laxity, PFAIR, DP-FAIR, ...)
  - Allocation Problem: Where to run this job?
    - no migration
    - task-level migration (no migration of running jobs)
    - job-level migration (migrate also running jobs)
    - global
    - partitioned

- Pre-emption Costs – UP / MP
- Migration Costs - MP
Pre-emption Costs

- direct costs:
  - timer / device interrupt
  - save register state
  - manipulate ready list
    - UP: no synchronization required
  - load register state of next thread

- indirect costs
  - cache evictions between two consecutive runs
  - TLB refills after evictions / shootdown
Pre-emption Costs

- direct costs:
  - timer / device interrupt
  - save register state
  - manipulate ready list
    - UP: no synchronization required
  - load register state of next thread

- indirect costs
  - cache evictions between two consecutive runs
  - TLB refills after evictions / shutdown
Migration Costs

- job-level migration
  - migration of running job implies pre-emption at source CPU

- task-level migration
  - job is already pre-empted

- direct costs
  - manipulate remote / global ready list
    - synchronization
  - fetch register state

- indirect costs
  - fetch active cache working set from remote cache
  - load remaining data from remote memory
Multiprocessor Architectures

- AMD Opteron / Intel Core Duo: SMT + multi core + ccNuma
Multiprocessor Architectures

- Symmetric Multi-Threaded (SMT) Processors
  - m hardware threads
  - shared pipeline
    - m HW threads

- Hardware Scheduler (picks one HW Thread at a time)
Multiprocessor Architectures

- Symmetric Multi-Threaded (SMT) Processors
  - $m$ hardware threads
  - shared pipeline
    - $m$ HW threads

```
<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
</tr>
<tr>
<td>EX</td>
</tr>
<tr>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load / Store Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD/ST</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Write Back / Commit Results

HW Scheduler (picks one HW Thread at a time)
Multiprocessor Architectures

- Symmetric Multi-Threaded (SMT) Processors
  - operating system multiplexes \( n \) SW threads on \( m \) HW threads
  - caches + pipeline is shared => no indirect migration costs

![Diagram of a multiprocessor architecture](example.png)
Multiprocessor Architectures

- Multi-Core Processors
  - operating system multiplexes $n$ SW threads on $m$ cores
  - timing of last level cache dominates migration costs
Multiprocessor Architectures

- Symmetric Multiprocessors
  - operating system multiplexes \( n \) SW threads on \( m \) dies
  - timing of interconnect dominates migration costs
Multiprocessor Architectures

- (cache coherent) NUMA
  - like SMP
  - non-uniform memory access: fetch from remote memory
Multiprocessor Architectures

- AMD Opteron [Corey: OSDI '08]
Multiprocessor Architectures

- AMD Opteron [Corey: OSDI '08]
Multiprocessor Architectures

- AMD Opteron [Corey: OSDI ’08]
Migration Costs

- Active Cache Working Set
  - cachelines a thread would access again if it would run
  - varies over time
  - ages out after preemption

age: 0 1 2 3
Migration Costs

- Active Cache Working Set
  - cachelines a thread would access again if it would run
  - varies over time
  - ages out after preemption

<table>
<thead>
<tr>
<th>age</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

![Diagram showing cache blocks and ages](image)
Migration Costs

- Active Cache Working Set
  - cachelines a thread would access again if it would run
  - varies over time
  - ages out after preemption

age: 0 1 2 3
Migration Costs

- Summary
  - migration costs are highly architecture dependent
  - non-trivial to predict
  - may cause a significant delay when a thread resumes execution

- Assumption for the remainder of this lecture:
  - zero pre-emption and migration costs / attributed to WCET
Design Space of MP Scheduling

- Dynamic job priority / partitioned
- Dynamic job priority / task level migration
- Dynamic job priority / job level migration
- Fixed job priority / partitioned
- Fixed job priority / task level migration
- Fixed job priority / job level migration
- Fixed task priority / partitioned
- Fixed task priority / task level migration
- Fixed task priority / job level migration
Design Space of MP Scheduling

Partitioned
- dyn. job prio. / partitioned
- fixed job prio. / partitioned
- fixed task prio. / partitioned

Global
- dyn. job prio. / task level migration
- fixed job prio. / task level migration
- fixed task prio. / task level migration
- fixed job prio. / job level migration
- fixed job prio. / job level migration
- fixed task prio. / job level migration
Design Space of MP Scheduling

Relative ordering between classes of scheduling algorithms

- dyn. job prio. / job level migration
- dyn. job prio. / task level migration
- fixed job prio. / partitioned
- dyn. job prio. / partitioned
- fixed job prio. / task level migration
- fixed job prio. / job level migration
- fixed task prio. / partitioned
- fixed task prio. / task level migration
- fixed task prio. / job level migration

Later in this lecture
Outline

- Introduction
- Terminology, Notation and Assumptions
- Anomalies + Impossibility Results
- Partitioned Scheduling
- Global (Task-Lvl migration) Scheduling
  - G-FTP (e.g., G-RMS)
  - G-EDF
- Optimal MP Scheduling
- MP – Resource Access Protocols
- Open Research Issues
Terminology, Notation and Assumptions

- **Periodic Tasks**
  - Task $t_i = (P_i, D_i, C_i)$  \quad $P_i = \text{const.}$

- **Sporadic Tasks**
  - $P_i = \text{Minimal Interrelease Time}$

- **Deadlines**
  - implicit deadline: $D_i = P_i$  \quad (relative deadline = period)
  - constrained: $D_i \leq P_i$  \quad (relative deadline < period)
  - arbitrary  \quad (deadline may be after period end)

- **Utilization**
  \quad $U_i = \frac{C_i}{P_i}$

- **Density**
  \quad $\partial_i = \frac{C_i}{\min(D_i, P_i)}$
Terminology, Notation and Assumptions

- Assumptions for the remainder of this lecture
  - independent tasks
  - fully pre-emptible / migratable (negligible costs)
  - unlimited number of priorities
  - tasks are single threaded: a job can utilize only 1 CPU at a time
  - jobs do not block (shared resources later in this lecture)
- pictures show schedules for 2 CPUs
**Terminology: P-DMS / G-RMS / G-EDF**

- **Scheduling Algorithms:**
  - **Deadline Monotonic Scheduling:**
    
  - **Rate Monotonic Scheduling:**
    
  - **Earliest Deadline First:**

- **P-DMS**
- **G-RMS / G-EDF**
Terminology: P-DMS / G-RMS / G-EDF

- Scheduling Algorithms:
  - **Deadline Monotonic Scheduling:**
    - job priority is inversely proportional to deadline
  - **Rate Monotonic Scheduling:**
    - job priority is inversely proportional to period
  - **Earliest Deadline First:**
    - job priority is inversely proportional to deadline
  - **P-DMS**
  - **G-RMS / G-EDF**
  - **Global:**
    - threads may migrate to other CPUs
    - scheduler picks thread from global ready queue
    - accesses to ready queue must be synchronized
  - **Partitioned:**
    - assign threads to processors
    - scheduler picks threads from local (per CPU) ready queue
    - no synchronization overhead for accessing the ready queue
Anomalies

- Simultaneous Release is not Critical Instance [Lauzac '98]
  - longer response time in second period

yellow misses its deadline

simultaneous release of all tasks
Anomalies

- Response time (of green) depends not only on set of higher prioritized tasks but also on their relative priority ordering.
Sustainability [Baruah '06]

- A schedulable workload remains schedulable if we
  - decrease the execution time of a task (predictability)
    - otherwise, WCET won't work as admission criterion
  - increase the minimal interrelease time (period) of a task
    - otherwise, more frequent recurrence is no safe approximation
  - increase the relative deadline of a task
    - otherwise, earlier deadline is no safe approximation

- G-FTP + G-EDF are not sustainable if #CPUs > 1

- all pre-emptive FJP / FTP algorithms are predictable

  Fixed Job Priority  Fixed Task Priority
Dhall Effect

- The utilization bound of Global EDF is as low as $U_{\text{EDF}} = 1 + e$
  - $m$ tasks with short periods and infinitesimal low $U_i$ (e.g., $U_i = e$)
  - 1 task with larger period and $U_j$ close to 1 (e.g., $U_j > (2 - e) / 2$)

- Dhall Effect does not manifest if $U_i < 41\%$
Dhall Effect

- The utilization bound of Global EDF is as low as $U_{EDF} = 1 + e$
  - $m$ tasks with short periods and infinitesimal low $U_i$ (e.g., $U_i = e$)
  - 1 task with larger period and $U_j$ close to 1 (e.g., $U_j > (2 - e) / 2$)

- Dhall Effect does not manifest if $U_i < 41\%$
some Impossibility Results

- [Hong '88]
  - No optimal online MP scheduling algorithm for arbitrary jobs unless all jobs have the same relative deadline.

- [Dertouzos '89]
  - Even if execution times are known precisely

- [Fisher '07]
  - No optimal online algorithm for sporadic tasksets with constrained or arbitrary deadlines.

\[ t \]
some Impossibility Results

- [Hong '88]
  - No optimal online MP scheduling algorithm for arbitrary jobs unless all jobs have the same relative deadline.

- [Dertouzos '89]
  - Even if execution times are known precisely

- [Fisher '07]
  - No optimal online algorithm for sporadic tasksets with constrained or arbitrary deadlines.
Partitioned Scheduling

Partitioned
- dyn job prio. / partitioned
- fixed job prio. / partitioned
- fixed task prio. / partitioned

Global
- dyn job prio. / task level migration
- fixed job prio. / task level migration
- fixed task prio. / task level migration
- fixed job prio. / job level migration
- fixed task prio. / job level migration
Partitioned Scheduling

- Split workload by allocating tasks to CPUs
- Run allocated task with UP scheduling algorithm
  - reap benefit of well known UP results
  - optimal task allocation is NP complete:
    - pack $n$ tasks with density $d_i$ on $m$ CPUs with capacity $d_{\text{max}} = 1$
    - Bin-packing

\[ \text{CPU}_0 \rightarrow \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S} \]

\[ \text{CPU}_1 \rightarrow \]

\[ \text{...} \]

\[ \text{CPU}_m \rightarrow \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S} \]
Partitioned Scheduling

- Utilization bound for implicit deadline workloads [Anderson '01]

\[ U_{opt} = \frac{m + 1}{2} \]

No partitioning scheduling algorithm can produce a feasible schedule of \( m+1 \) tasks with execution time \( 1+e \) and period of 2 on \( m \) processors.
Partitioned Scheduling

- Utilization bound for implicit deadline workloads [Anderson '01]

\[ U_{\text{opt}} = \frac{m + 1}{2} \]

No partitioning scheduling algorithm can produce a feasible schedule of \( m+1 \) tasks with execution time \( 1+e \) and period of 2 on \( m \) processors.
Partitioned Scheduling

- Utilization bound for implicit deadline workloads [Anderson '01]

\[ U_{opt} = \frac{m + 1}{2} \]

No partitioning scheduling algorithm can produce a feasible schedule of \( m+1 \) tasks with execution time \( 1+e \) and period of 2 on \( m \) processors.

Easy if blue and green can migrate to CPU₂
Can we improve on Anderson's Utilization Bound?
  - by allowing a few jobs to migrate

PDMS Partitioned Deadline Monotonic Scheduling
HPTS Split Highest Priority Task
DS Allocate according to Highest Density First
PDMS-HPTS-DS

\[ t_1 = (4,3,1) \]

\[ t_2 = (6,2,2) \]

\[ t_3 = (4,4,1) \]

\[ t_4 = (6,4,2) \]

\[ t_5 = (6,5,1) \]

\[
\begin{align*}
u_1 &= 0.25 \quad \delta_1 = 1/3 = 0.33 \\
u_2 &= 0.33 \quad \delta_2 = 1 \\
u_3 &= 0.25 \quad \delta_3 = 1/4 = 0.25 \\
u_4 &= 0.33 \quad \delta_4 = 1/2 = 0.5 \\
u_5 &= 0.16 \quad \delta_5 = 1/5 = 0.2 \\
u_{\text{sum}} &= 1.33 \quad => \frac{u_{\text{sum}}}{2} = 0.66% 
\end{align*}
\]
\[ \tau_1 = (4,3,1) \quad \tau_2 = (6,2,2) \quad \tau_3 = (4,4,1) \quad \tau_4 = (6,4,2) \quad \tau_5 = (6,5,1) \]

\[ u_1 = 0.25 \quad \delta_1 = 1/3 = 0.33 \]
\[ u_2 = 0.33 \quad \delta_2 = 1 \]
\[ u_3 = 0.25 \quad \delta_3 = 1/4 = 0.25 \]
\[ u_4 = 0.33 \quad \delta_4 = 1/2 = 0.5 \]
\[ u_5 = 0.16 \quad \delta_5 = 1/5 = 0.2 \]

\[ u_{\text{sum}} = 1.33 \Rightarrow u_{\text{sum}} / 2 = 0.66\% \]
\[ \begin{align*}
\tau_1 &= (4,3,1) \\
\tau_2 &= (6,2,2) \\
\tau_3 &= (4,4,1) \\
\tau_4 &= (6,4,2) \\
\tau_5 &= (6,5,1)
\end{align*} \]

\[
\begin{align*}
u_1 &= 0.25 \quad \delta_1 = 1/3 = 0.33 \\
u_2 &= 0.33 \quad \delta_2 = 1 \\
u_3 &= 0.25 \quad \delta_3 = 1/4 = 0.25 \\
u_4 &= 0.33 \quad \delta_4 = 1/2 = 0.5 \\
u_5 &= 0.16 \quad \delta_5 = 1/5 = 0.2
\end{align*}
\]

\[
\sum u_i = 1.33 \quad \Rightarrow \frac{\sum u_i}{2} = 0.66\%
\]
\[ \begin{align*}
\tau_1 &= (4,3,1) \quad & u_1 &= 0.25 \quad \delta_1 &= 1/3 = 0.33 \\
\tau_2 &= (6,2,2) \quad & u_2 &= 0.33 \quad \delta_2 &= 1 \\
\tau_3 &= (4,4,1) \quad & u_3 &= 0.25 \quad \delta_3 &= 1/4 = 0.25 \\
\tau_4 &= (6,4,2) \quad & u_4 &= 0.33 \quad \delta_4 &= 1/2 = 0.5 \\
\tau_5 &= (6,5,1) \quad & u_5 &= 0.16 \quad \delta_5 &= 1/5 = 0.2 \\
\end{align*} \]

\[
\frac{u_{\text{sum}}}{2} = 0.66\%
\]
\[
\begin{align*}
\tau_1 &= (4,3,1) & u_1 &= 0.25 & \delta_1 &= 1/3 = 0.33 \\
\tau_2 &= (6,2,2) & u_2 &= 0.33 & \delta_2 &= 1 \\
\tau_3 &= (4,4,1) & u_3 &= 0.25 & \delta_3 &= 1/4 = 0.25 \\
\tau_4 &= (6,4,2) & u_4 &= 0.33 & \delta_4 &= 1/2 = 0.5 \\
\tau_5 &= (6,5,1) & u_5 &= 0.16 & \delta_5 &= 1/5 = 0.2 \\
\end{align*}
\]

\[u_{\text{sum}} = 1.33 \implies u_{\text{sum}} / 2 = 0.66\%\]
\( \tau_1 = (4,3,1) \)

\( \tau_2' = (6,2,1) \)

\( \tau_2'' = (5,1,1) \)

\( \tau_3 = (4,4,1) \)

\( \tau_4 = (6,4,2) \)

\( \tau_5 = (6,5,1) \)

\[ u_2 = 0.33 \quad \delta_2 = 1 \]

\[ u_3 = 0.25 \quad \delta_3 = \frac{1}{4} = 0.25 \]

\[ u_4 = 0.33 \quad \delta_4 = \frac{1}{2} = 0.5 \]

\[ u_5 = 0.16 \quad \delta_5 = \frac{1}{5} = 0.2 \]

\[ u_{\text{sum}} = 1.33 \Rightarrow u_{\text{sum}} / 2 = 0.66\% \]
\(\tau_1 = (4,3,1)\)  \(\tau_1' = (6,2,1)\)  \(\tau_1'' = (5,1,1)\)  \(\tau_3 = (4,4,1)\)  \(\tau_4 = (6,4,2)\)  \(\tau_5 = (6,5,1)\)

\[u_1 = 0.25 \quad \delta_1 = 1/3 = 0.33\]

\[u_2 = 0.33 \quad \delta_2 = 1\]

\[u_3 = 0.25 \quad \delta_3 = 1/4 = 0.25\]

\[u_4 = 0.33 \quad \delta_4 = 1/2 = 0.5\]

\[u_5 = 0.16 \quad \delta_5 = 1/5 = 0.2\]

\[u_{\text{sum}} = 1.33 \Rightarrow u_{\text{sum}} / 2 = 0.66\%\]
PDMS-HPTS-DS

- \( U_{\text{PDMS-HPTS-DS}} = 69.3\% \) if all tasks have a utilization \( U_i < 41\% \)
Global Scheduling

Partitioned

- **dyn job prio. / partitioned**
- **fixed job prio. / partitioned**
- **fixed task prio. / partitioned**

Global

- **dyn job prio. / task level migration**
- **fixed job prio. / task level migration**
- **fixed task prio. / task level migration**
- **dyn. job prio. / job level migration**
- **fixed job prio. / job level migration**
- **fixed task prio. / job level migration**

Real-Time Systems, Multiprocessor Scheduling / Marcus Völp
Global Scheduling

- Always pick the ready jobs of the m most “important” tasks
- A task may migrate
  - When a new job of a task is released it may receive a different CPU; once started, a job is no longer migrated
- No need for allocation / load balancing
  - Load balancing is automatic
Global Scheduling – Utilization Bound

- Utilization bound for global fixed-job priority algorithms
  - on m CPUs, G-FJP algorithms cannot schedule m+1 tasks with $C_i = 1 + e, \ P_i = 2 \ (e \to 0)$

  \[ U_{OPT} = \lim_{e \to 0} \frac{(m+1) (1 + e)}{2} = \frac{m+1}{2} \]
Global Scheduling - Job Level Migration

Partitioned
- dyn. job prio. / partitioned
- fixed job prio. / partitioned
- fixed task prio. / partitioned

Global
- dyn. job prio. / task level migration
- fixed job prio. / task level migration
- fixed task prio. / task level migration
- dyn. job prio. / job level migration
- fixed job prio. / job level migration
- fixed task prio. / job level migration
PFAIR [Baruah '96]

- Divide timeline into equal length quanta

- At each quanta of length $t$, allocate tasks to processors such that the accumulated processor time is either $[tu_i]$ or $[tu_i]$
PFAIR [Baruah '96]

- Divide timeline into equal length quanta
- At each quanta of length \( t \), allocate tasks to processors such that the accumulated processor time is either \([tu_i]\) or \([tu_i]\)
PFAIR [Baruah '96]

- Divide timeline into equal length quanta
- At each quanta of length $t$, allocate tasks to processors such that the accumulated processor time is either $[tu_i]$ or $[tu_i]$
PFAIR [Baruah '96]

- Divide timeline into equal length quanta
- At each quanta of length $t$, allocate tasks to processors such that the accumulated processor time is either $[tu_i]$ or $[tu_i]$
- PFAIR is optimal for periodic implicit deadline tasksets: $U_{OPT} = m$
- Very high preemption and migration costs
DP-Fair [Brandt '10]

- DP-Fair
  - Optimal scheduler for periodic implicit deadline tasksets with a minimal number of preemptions / migrations?
  - Recall [Hong '88]:
    - No optimal MP scheduling algorithm for arbitrary tasksets if not all tasks have the same relative deadline
  
  =>

- deadline partitioning
  - any task’s deadline becomes a deadline for all tasks

- always run zero laxity jobs
  - laxity = time to deadline – remaining execution time
  - zero laxity => job may miss its deadline if it is not run

- jobs who twine themselves around the fluid rate curve are somehow in good shape
DP-Fair [Brandt '10]

- Fluid Rate Curve

  work remaining

  fluid rate curve
  actual work remaining curve

  zero laxity event: no more time to run others
DP-Fair [Brandt '10]

- DP-Fair:
  - a family of optimal, deadline partitioning scheduling algorithms
  - Split timeline into chunks according to job deadlines
  - Allocate work to a chunk proportional to $U_i$
    - local execution time: $C_{i,j}^l = (t_{j+1} - t_j) U_i$
  - Rule 1: run all jobs with zero local laxity
    - jobs with remaining local execution time = time to end of chunk
  - Rule 2: do not run jobs for more than $C_{i,j}^l$
  - Rule 3: split up idle time proportional to length of chunk
    - allocate at most $S (t_{j+1} - t_j)$ idle time to chunk $j$
**DP-Fair [Brandt '10]**

- Deadline partitioning

Introduce additional releases / deadlines for all jobs whenever there is such an event for one job in the original schedule => chunks
DP-Fair [Brandt '10]

- Idle time

\[ S = m - \sum_{i=0}^{n} U_i \]

Treat idle time as just another job to schedule.
Allocate work proportional to $U_i$.

Allocate execution (and idle) time of a job proportionally to its utilization.

$=>$ amount of time that this job must run in a given chunk.
DP-Fair [Brandt '10]

- Jobs hit their fluid rate curve at the end of each chunk

![Diagram showing fluid rate curves and some arbitrary ordering]
DP-Wrap – a DP-Fair Scheduler

Work allocated to a chunk:
find a schedule
DP-Wrap – a DP-Fair Scheduler

Work allocated to a chunk:
find a schedule

arrange jobs consecutively
DP-Wrap – a DP-Fair Scheduler

Work allocated to a chunk: find a schedule

arrange jobs consecutively

wrap around to obtain schedule for 2nd CPU, ...
DP-Wrap – a DP-Fair Scheduler

Work allocated to a chunk: find a schedule

arrange jobs consecutively

wrap around to obtain schedule for 2\textsuperscript{nd} CPU, ...
DP-Wrap – a DP-Fair Scheduler
Unnecessary migration of red task at chunk boundaries

=> mirror processor assignment of every second chunk
m – 1 migrations per chunk
n – 1 context switches per chunk
Design Space of MP Scheduling

dyn job prio. / partitioned

fixed job prio. / partitioned

fixed task prio. / partitioned

dyn job prio. / task level migration

fixed job prio. / task level migration

fixed task prio. / task level migration

dyn. job prio. / job level migration

fixed job prio. / job level migration

fixed task prio. / job level migration
Design Space of MP Scheduling

- **dyn. job prio. / job level migration**
- **dyn. job prio. / task level migration**

- **fixed job prio. / partitioned**
- **fixed job prio. / task level migration**
- **fixed job prio. / job level migration**

- **fixed task prio. / partitioned**
- **fixed task prio. / task level migration**
- **fixed task prio. / job level migration**
Design Space of MP Scheduling

\[ U_{opt} = m \]
\[ U_{opt} = \frac{m+1}{2} \]

- **Fixed Job Prio. / Partitioned**
- **Dynamic Job Prio. / Job Level Migration**
- **Dynamic Job Prio. / Task Level Migration**
- **Fixed Job Prio. / Task Level Migration**
- **Fixed Task Prio. / Partitioned**
- **Fixed Task Prio. / Task Level Migration**
- **Fixed Task Prio. / Job Level Migration**

\[ A \rightarrow B \Rightarrow A \text{ can schedule any taskset that } B \text{ can schedule and more} \]
\[ A \leftrightarrow B \Rightarrow \text{dominance is not yet known} \]
Outline

- Introduction
- Terminology, Notation and Assumptions
- Anomalies + Impossibility Results
- Partitioned Scheduling
- Global (Task-Lvl migration) Scheduling
  - G-FTP (e.g., G-RMS)
  - G-EDF
- Optimal MP Scheduling
- MP – Resource Access Protocols
- Open Research Issues
MP Resource Access Protocols

- **UP:**
  - Basic Priority Ceiling Protocol BPCP
  - Stack Resource Protocol SRP (Ceiling Priority Protocol CPP)
    - bounded priority inversion: | CS |
    - BPCP does not influence unrelated threads

- **General Idea:**
  - run UP protocol on every CPU of MP system

- **Ceiling Priority i of Resource R_i:** \( \hat{R}_i = \max \text{prio}(t_j) \)
  - here, priorities have a global meaning

- **System Ceiling** \( \hat{S} = \max \hat{R}_i \text{ of held resources} \)

- **Synchronization Processor:** CPU on which \( R_i \) is executed
MP Resource Access Protocols

- UP-BPCP in a nutshell
  - resource holders inherits priority from blocked threads
  - resource granted if \( \text{prio}(t_j) > \hat{S} \)
  - only the resource holder, which holds a resource with \( \hat{R}_i = \hat{S} \) receives additional resources

- UP-SRP (actually UP-CPP) in a nutshell
  - resource holder runs at max \( \hat{R}_i \) of held resources
  => only higher prioritized threads may run (acquire resources)
MPCP – first try
MPCP – first try

unrelated threads can affect blocking time
MP Priority Ceiling Protocol (MPCP)

- Run the priority ceiling protocol on each processor
- Treat global resources as jobs with same relative but strictly higher priority than other threads on the sync. CPU

![Diagram showing the execution of jobs on processors P1 and P2]

- J1
- J2
- J3
- J4
MP Priority Ceiling Protocol (MPCP)

- Run the priority ceiling protocol on each processor
- Treat global resources as jobs with same relative but strictly higher priority than other threads on the sync. CPU
MP Priority Ceiling Protocol (MPCP)

- Run the priority ceiling protocol on each processor
- Treat global resources as jobs with same relative but strictly higher priority than other threads on the sync. CPU
MP Stack-Resource Protocol (MSRP)

- SRP on every CPU
- Spin non-preemptively while executing a GCS on the sync. CPU
Flexible Multiprocessor Locking Protocol

- works with global scheduling
- access short resources non-preemptively (busy wait if blocked)
- block on Semaphore if long resource is busy
  - Resource holder inherits priority
- resource groups to avoid deadlocks
Performance

- [Gai '03]:
  - MSRP typically outperforms MPCP for short + typically local CSs

- [Brandenburg '08]:
  - comparison of blocking (FMLP) vs. lock-free vs. wait-free
  - non-blocking are preferable for small + simple CSs
  - wait-free / spin-based are preferable for long CSs
  - spin-based almost always outperforms suspension-based algorithms SRP on every CPU
Locking for Clustered Scheduling

- [Brandenburg '11]:
  - clustered scheduling: global within the cluster; partitioned in between

- Idea:
  - Every task helps out resource holders for a bounded time
  - Only the n-highest prioritized threads may acquire resources
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

J₁ preempts J₃ who holds resource

=> J₁ becomes priority donor of J₃
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

J_6 suspends itself because R1 is held by J_3
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

J_4 preempts J_6 and becomes its priority donor
J_4 runs because J_6 blocks on J_3
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

- \( J_2 \) suspends because priority donors are not allowed to acquire resources
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

J₄ executes R2 because after J₅ completes J₆ is one of the k-highest priority jobs
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

J_1

J_2

J_3

J_4

J_5

J_6

R1

R2

R2

J_6 receives R1 after J_3 releases this resource
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster

J₃ no longer holds a resource so J₁ stops being its donor
Locking for Clustered Scheduling

- 2 clusters; 2 cores each; 3 jobs / cluster
Outline

● Introduction
● Terminology, Notation and Assumptions
● Anomalies + Impossibility Results
● Partitioned Scheduling
● Global (Task-Lvl migration) Scheduling
  ○ G-FTP (e.g., G-RMS)
  ○ G-EDF
● Optimal MP Scheduling
● MP – Resource Access Protocols
● Open Research Issues
Open Issues [Burns '09]

- Limited processor Utilization
  - minimally dynamic algorithms
  - novel partitioning approaches
    - increase the guaranteed processing capability; overheads

- Ineffective Schedulability Tests (in particular, sporadic workloads)
  - large gap between feasibility / infeasibility tests
  - identify finite collection of worst-case job arrival sequences

- Overheads
  - migration costs; run queue manipulations; context switching
  - algorithms that permit intra-cluster migration; task-level migr.

- Task Models
  - [Colette] benefit / overhead oriented task models
  - [Edmons] speedup from extra parallelism
Integrating Job Parallelism [Colette]

Task: $\tau_i \overset{\text{def}}{=} (C_i, P_i, \Gamma_i)$

$\Gamma_i \overset{\text{def}}{=} (\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,m})$

Job with $j$ processors completes $\gamma_{i,j} \times t$ units of execution
Runtime Integration

Threads

Threadpools

Apple’s Grand Central Dispatch
C++11 Futures

Real-Time Systems, Multiprocessor Scheduling / Marcus Völp
References

- [Burns '09]

- [Colette]

- [Edmonds]

- [Brandt '10]

- [Hong '88]

- [Anderson '01]

- [Lakshmanan '09]
References

- [Fischer '07]

- [Dertouzos '89]

- [Dhall]

- [Baruah '06]
  S. Baruah, A. Burns, “Sustainable Scheduling Analysis,” RTSS, 2006

- [Corey '08]

- [Carpenter '04]

- [Whitehead]

- [Brandenburg '08]
References

- [Gai '03]

- [Block]

- [MSRP]

- [MPCP]

- [Lauzac '98]

- [Baruah '96]

- [Brandenburg'11]