8. Multicore systems

Credits to A. Burns and A. Wellings

to B. Andersson and J. Jonsson for their work in *Proc. of the the IEEE Real-Time Systems Symposium*, WiP Session, 2000, pp. 53–56

and to a student of this class a few years back

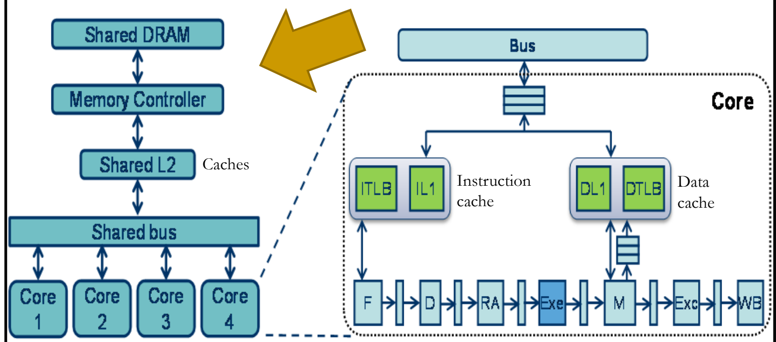
Fundamental issues

- Hardware architecture taxonomy
  - Homogeneous vs. heterogeneous processors
    - Research focused first on SMP (symmetric multiprocessors) which make a much simpler problem
- Scheduling approach
  - Global or partitioned or alternatives between these extremes
    - Partitioning is an allocation problem followed by single processor scheduling
- Optimality criteria are shattered
  - EDF no longer optimal and not always better than FPS
  - Global scheduling not always better than partitioned

Hardware architecture taxonomy

- A multiprocessor (or multi-core) is *tightly coupled*
  - Global status and workload information on all processors (cores) can be kept current at low cost
  - The system may use a centralized dispatcher and scheduler
  - When each processor (core) has its own scheduler, the decisions and actions of all schedulers are coherent
    - Scheduling in this model is an NP-hard problem
- A distributed system is *loosely coupled*
  - It is too costly to keep global status
  - There usually is a dispatcher / scheduler per processor

Understanding the hardware /2



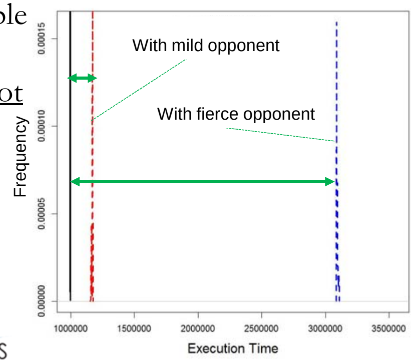
Courtesy of PROXIMA

Hardware interference /1

- Parallel execution on a multiprocessor causes vast opportunities of contention for hardware resources that are shared among the cores
- This phenomenon increases the execution time of running threads by causing them to use CPU cycles *without* progressing (!)
  - Not quite like software interference, which prevents a ready thread from running

Hardware interference /2

- The WCET of a simple single-path program running alone does not stay the same when other programs do execute on other CPUs



Courtesy of PROARTIS

State of the art

- Some task sets may be deemed unschedulable even though they have low utilization
  - Much less than the number of processors
  - This is known as the Dhall's effect [Dhall & Liu, 1978]
- The known *exact* schedulability tests have exponential time complexity
  - The known sufficient tests have polynomial time complexity but obviously are pessimistic
- Rate-monotonic priority assignment is not optimal
- No optimal priority assignment scheme with polynomial time complexity has been found yet

Software interference /1

- We know what is the interference I_i suffered by a task τ_i for single-processor scheduling
  - How does this change for multiprocessors?
- For *global* multiprocessor scheduling with m processors interference only occurs for tasks from $m+1$ onward
- Multiprocessor interference can be computed as the sum of all intervals when m higher-priority tasks execute in parallel on all m processors

Software interference /2

- A very pessimistic bound considers all higher-priority tasks to always fully interfere
  - $R_k^{max} = C_k + \frac{1}{m}\sum_{\tau_j \in hp(k)}(\left\lceil\frac{R_k^{max}}{T_j}\right\rceil C_j + Cj)$
- This naive bound can be improved, and has been, but for great computational complexity and still without becoming exact

Example (Dhall's effect) – 1

Task	T	D	C	U
a	10	10	5	0.5
b	10	10	5	0.5
c	12	12	8	0.67

On 2 processors $\sum_i U_i = 1.67 < 2$

- Under global scheduling, EDF and FPS would run tasks **a** and **b** first on each of the 2 processors
- But this would leave no time for task **c** to complete
  - 7 time units on each processor, 14 in total, but 8 on neither
- Even if the total system is underutilized (!)

Example – 2

Task	T	D	C	U
d	10	10	9	0.9
e	10	10	9	0.9
f	10	10	2	0.2

On 2 processors $\sum_i U_i = 2$

- Partitioned scheduling does not work here either
- After tasks **d** and **e** are allocated, task **f** cannot reside on just one processor
  - It needs to migrate from one to the other to find room for execution
- And it also needs that tasks **d** and **e** are willing to use cooperative scheduling for it complete in time

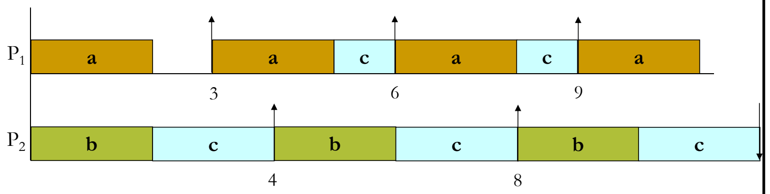
Global scheduling anomalies

- In single-processor real-time scheduling the deadline miss ratio often highly depends on the system load
  - This suggests that increasing the period should decrease the utilization and thus decrease the deadline miss ratio
- **Anomaly 1**
  - A *decrease* in processor demand from higher-priority tasks can *increase* the interference on lower-priority tasks because of the change in the time when tasks execute
- **Anomaly 2**
  - A *decrease* in processor demand of a task causes an *increase* in the interference suffered by that task

Anomaly 1: decrease in *hp* demand

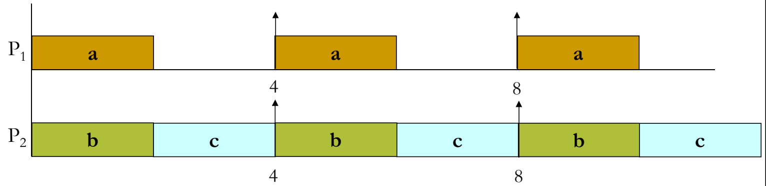
Task	T	D	C	U
a	3	3	2	0.67
b	4	4	2	0.50
c	12	12	8	0.67

$m = 2$ processors and $\sum_i U_i = 1.83$ but τ_c is *saturated* because $C_c + I_c = D_c$ hence any increase in I_c would make it unschedulable



Anomaly 1 (cont'd)

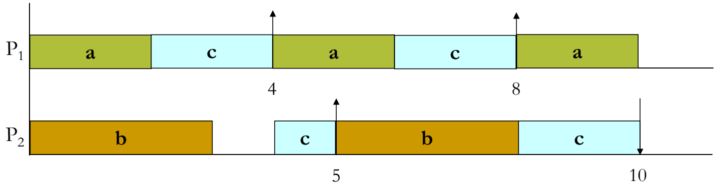
- If we reduce T_a to 4 we *decrease* system load to $U = 1.67$
- But in this way I_c *increases* from 4 to 6 and τ_c misses its deadline (!)



Anomaly 2: decrease in own demand

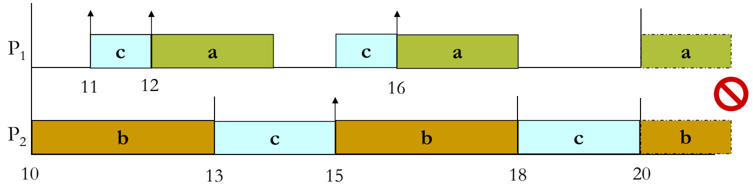
Task	T	D	C	U
a	4	4	2	0.5
b	5	5	3	0.6
c	10	10	7	0.7

$m = 2$ processors and $U = 1.8$ but τ_c with $I_c = 3$ is *saturated*



Anomaly 2 (cont'd)

- If we extend T_c to 11 we *decrease* system load to $U = 1.74$
- But in this way I_c *increases* from 3 to 5 (!) as it becomes visible in the second job of τ_c



The defeat of greedy schedulers /1

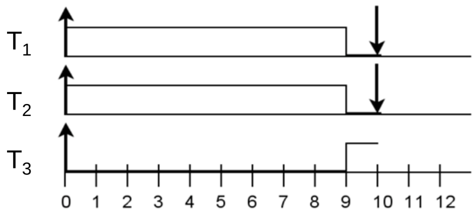
- Greedy algorithms are easy to explain, study, and implement
  - They work very well on single processors
  - EDF [1] and LLF [2] are optimal for single processors
- They collapse the urgency of a job into a single value and use it to greedily schedule jobs
- Unfortunately (and surprisingly) greedy algorithms fail when used on multiprocessors
  - EDF and LLF are no longer optimal

The defeat of greedy schedulers /2

- Does a feasible schedule exist on 2 processors for T (derivative of Example 2) where
  - $T = \{\tau_1 = (10,9), \tau_2 = (10,9), \tau_3 = (40,8)\}, U(T) = 2$
  - τ_1 and τ_2 have laxity 1 in each period
  - Hence they leave each processor idle for 1 unit of time and for 2 units in total every 10-unit period
  - In the interval $[0,40)$ τ_1 and τ_2 leave the 2 processors idle for a total of $2 \times 4 = 8$ units of time in which fits τ_3 exactly
- The answer should thus be yes since also τ_3 should be able to meet its deadline

The defeat of greedy schedulers /3

- Let us schedule T with LLF



  - τ_3 can execute only 1 unit of time in the interval $[0,10)$
  - One of the two processors is idle for 1 unit of time
- τ_3 misses its deadline!

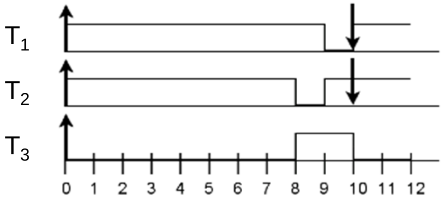
Why do greedy schedulers fail?

Theorem 1 (stating the obvious)

When the total utilization of a periodic task set is equal to the number of processors, then no feasible schedule can allow any processor to remain idle for any length of time

The defeat of greedy schedulers /4

- One schedule we want for T is



  - But at $t = 8$ τ_1 and τ_2 have earlier deadline, lower laxity, greater total and remaining utilization than τ_3
  - Greedy schedulers lack knowledge to be wiser!

The defeat of greedy schedulers /5

- Things work if we modify T to
  $T' = \{\tau_1 = (10,9), \tau_2 = (10,9), \tau'_3 = (10,2)\}$
  - At $t = 8$ we get a zero-laxity event for τ'_3
  - This is good for T but surely not in general ☹
- The ultimate problem is to determine when (in time) and how (by what means) jobs should be able to hit their *proportional rate quota*
- In seeking *proportionate fairness* we do not want to incur large overhead with scheduling calculations and task migrations

P-fair scheduling [Baruah et al. 1996]

- *Proportional progress* is a form of proportionate fairness also known as ***P-fairness***
  - Each task τ_i is assigned resources in proportion to its *weight* $W_i = {}^{C_i}/_{T_i}$ hence it progresses proportionately
  - Useful e.g., for real-time multimedia applications
- At every time t task τ_i must have been scheduled either $\lfloor W_i \times t \rfloor$ or $\lceil W_i \times t \rceil$ time units
  - Without loss of generality preemption is assumed to only occur at integral time units
  - The workload model is periodic

P-fair scheduling /2

- $\boldsymbol{lag}(S, \tau_i, t)$ is the difference between the total resource allocations that task τ_i should have received in $[0, t)$ and what it received under schedule S
- For a P-fair schedule S at time t
  - τ_i is *ahead* iff $\boldsymbol{lag}(S, \tau_i, t) < 0$
  - τ_i is *behind* iff $\boldsymbol{lag}(S, \tau_i, t) > 0$
  - τ_i is *punctual* iff $\boldsymbol{lag}(S, \tau_i, t) = 0$

P-fair scheduling /3

- $\boldsymbol{\alpha}(\tau_i, t)$ is the *characteristic substring* of task τ_i at time t
  - Finite string over {-, 0, +} of $\boldsymbol{\alpha}_{t+1}(x)\boldsymbol{\alpha}_{t+2}(x)\boldsymbol{\alpha}_{t'}(x)$
    - Where $t' = min\, i: i > t: \boldsymbol{\alpha}_i(x) = 0$
  - $\boldsymbol{\alpha_t}(x) = \boldsymbol{sign}(W_x \times (t+1) - \lfloor W_x \times t \rfloor - 1)$
- For a P-fair schedule S at time t
  - τ_i is is *urgent* iff τ_i is *behind* and $\boldsymbol{\alpha_t}(\tau_i) \neq -$
  - τ_i is is *tnegru* iff τ_i is *ahead* and $\boldsymbol{\alpha_t}(\tau_i) \neq +$
  - τ_i is is *contending* otherwise

Properties of a P-fair schedule S

- For task τ_i *ahead* at time t under S
  - *tnegru* {
    - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i not scheduled at t then τ_i is *ahead* at $t+1$
    - If $\boldsymbol{\alpha_t}(\tau_i) = 0$ and τ_i not scheduled at t then τ_i is *punctual* at $t+1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i not scheduled at t then τ_i is *behind* at $t+1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i scheduled at t then τ_i is *ahead* at $t+1$
- For task τ_i *behind* at time t under S
  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i scheduled at t then τ_i is *ahead* at $t+1$
  - If $\boldsymbol{\alpha_t}(\tau_i) = -$ and τ_i not scheduled at t then τ_i is *behind* at $t+1$
  - *urgent* {
    - If $\boldsymbol{\alpha_t}(\tau_i) = 0$ and τ_i scheduled at t then τ_i is *punctual* at $t+1$
    - If $\boldsymbol{\alpha_t}(\tau_i) = +$ and τ_i scheduled at t then τ_i is *behind* at $t+1$

P-fair scheduling /4

- General principle of P-fairness
  - Every task *urgent* at time t must be scheduled at t to preserve P-fairness
  - No task *tnegru* at time t can be scheduled at t without breaking P-fairness
- Problems with n_0 *tnegru*, n_1 *contending*, n_2 *urgent* tasks at time t with m resources and $n = n_0 + n_1 + n_2$
  - If $n_2 > m$ the scheduling algorithm cannot schedule all *urgent* tasks
  - If $n_0 > n - m$ the scheduling algorithm is forced to schedule some *tnegru* tasks

P-fair scheduling /5

- The **PF** scheduling algorithm
  - Schedule all *urgent* tasks
  - Allocate the remaining resources to the highest-priority *contending* tasks according to the total order function $\supseteq$ with ties broken arbitrarily
    - $x \supseteq y$ iff $\boldsymbol{\alpha}(x,t) \geq \boldsymbol{\alpha}(y,t)$
    - And the comparison between the characteristics substrings is resolved lexicographically with $- < 0 < +$
- With PF we have $\sum_{x \in [0,n]} W_x = m$
  - A dummy task may need to be added to the task set to top utilization up
- No problem situation can occur with the PF algorithm

Example (PF scheduling) /1

Task	C	T	W
v	1	3	0.333…
w	2	4	0.5
x	5	7	0.714…
y	8	11	0.727…
z	335	462	3-U

- $m = 3$ processors
- $n = 4$ tasks
- τ_z is a dummy task used to top system utilization up
- In general its period is set to the system hyperperiod
  - This time we halved it
- With PF we always have $n_2 > m$ and $n_0 \leq n - m$

Example (PF scheduling) /2

These tasks are scheduled and they become ahead

t	lag × period					characteristic string					urgent tasks	contending tasks	tnegru tasks
	v	w	x	y	z	v	w	x	y	z			
0	0	0	0	0	0	−	−	−	−	−	{}	$y > z > x > w > v$	{}
1	1	2	−2	−3	−127	−	0	+	+	+	{w}	$y > z > x > v$	{}
2	2	0	3	−6	−254	0	−	+	+	+	{v, x}	$w > y > z$	{}
3	0	−2	1	2	81	−	0	−	−	−	{}	$y > z > x > v$	{w}
4	1	0	−1	−1	−46	−	−	+	+	+	{}	$y > z > x > v = w$	{}
5	2	2	−3	−4	−173	0	0	+	+	+	{v, w}	$y > z > x$	{}
6	0	0	2	−7	162	−	−	0	+	+	{x, z}	$w > y > v$	{}
7	1	−2	0	1	35	−	0	−	−	−	{}	$y > z > x > v$	{w}
8	2	0	−2	−2	−92	0	−	+	+	+	{v}	$y > z > x > w$	{}
9	0	2	3	−5	−219	−	0	+	+	+	{w, x}	$y > z > v$	{}
10	1	0	1	−8	116	−	−	−	0	−	{}	$z > x > v = w$	{y}
11	−1	2	−1	0	−11	0	0	+	−	+	{w}	$y > z > x$	{v}
12	0	0	4	−3	−138	−	−	+	+	+	{x}	$y > z > w > v$	{}
13	1	2	2	−6	−265	−	0	0	+	+	{w, x}	$v > y > z$	{}
14	−1	0	0	2	70	0	−	−	−	−	{}	$y > z > x > w$	{v}
15	0	2	−2	−1	−57	−	0	+	+	+	{w}	$y > z > x > v$	{}
16	1	0	3	−4	−184	−	−	+	+	+	{x}	$y > z > v = w$	{}
17	2	2	1	−7	−311	0	0	−	+	+	{v, w}	$x > y > z$	{}
18	0	0	−1	1	24	−	−	+	−	−	{}	$y > z > x > w > v$	{}
19	1	2	−3	−2	−103	−	0	+	+	+	{w}	$y > z > v = x$	{}

Predictability [Ha & Liu, 1994]

- For arbitrary job sets on multiprocessors, if the scheduling algorithm is ***work-conserving***[1]), preemptive, global (with migration), with fixed job priorities is predictable
  - Job completion times monotonically related to job execution times
- Hence it is safe to consider only upper bounds for job execution times in schedulability tests
- This is not true for non-preemptive scheduling

1) A scheduling algorithm is *work conserving* if processors are not idle while tasks eligible for execution are not able to execute on other processors

DP-Fair motivation

- Focus on periodic, independent task set with implicit deadlines ($D_i = p_i$)
  - Scheduling overhead costs assumed in task requirements
  - $\sum_i U_i \leq m$ and $U_i \leq 1 \forall i$
  - Process migration allowed
- With unlimited context switches and migrations any task set meeting the above conditions will be feasible
  - This problem is easy
- What's difficult is to find a valid schedule that minimizes context switches and migrations

Deadline partitioning

- Partition time into slices demarcated by the deadlines of all tasks in the system
  - All jobs are allocated a workload in each slide and these workload share the same deadline

Theorem 2 (Hong and Leung)

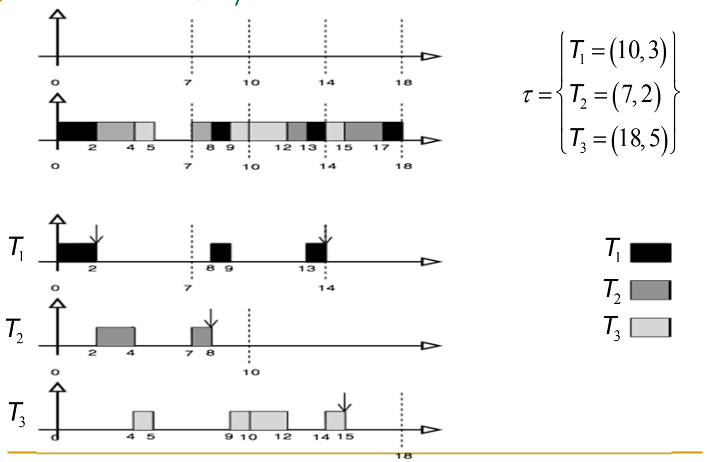
No optimal on-line scheduler can exist for a set of jobs with two or more distinct deadlines on any m multiprocessor system, where $m > 1$

- Why is DP so effective?

DP-Correct /1

- The time slice scheduler will execute all jobs' allocated workload within the end of the time slice whenever it is possible to do so
- Jobs are allocated workloads for each slice so that it is possible to complete this work within the slice
- Completion of these workloads causes all tasks' actual deadlines to be met

DP-Correct /2



Notation

- $t_0 = 0, t_i : i > 0$ denote distinct deadlines of all tasks in T
- σ_j is the j^{th} time slice in $[t_{j-1}, t_j)$
- $L_j = t_j - t_{j-1}$
- ***Local execution remaining*** $l_{i,t}$ is the amount of time that τ_i must execute before the next slice boundary
- ***Local utilization*** $r_{j,t} = l_{i,t}/(t_j - t)$
- $L_T = \sum_i l_i$ is the ***ler*** of the whole task set
- $R_T = \sum_i r_i$ is the ***lu*** of the whole task set
- *Slack* $S(T) = m - U(T)$ and represents a dummy job
- $a_{i,h}$ is the arrival time of the h^{th} job of τ_i

DP-Fair rules for periodic tasks set

- **DP-Fair allocation**
  - All tasks hit their *fluid rate curve* at the end of each slice by assigning each task a workload proportional to its utilization
  - At every σ_j assign $l_{i,t_{j-1}} = U_i \times L_j$ to τ_i
- **DP-Fair scheduling for time slices**
  - A slice-scheduling algorithm is DP-Fair if it schedules jobs within a time slice σ_i according to the following rules:
    1. Always run a job with zero local laxity
    2. Never run a job with no remaining local work
    3. Do not allow more than $S(\tau) \times L_j$ units of idle time to occur in σ_i before time t

DP-Fair optimality – Proof

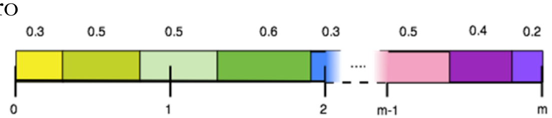
Theorem 5

Any DP-Fair scheduling algorithm for periodic task sets with implicit deadlines is optimal

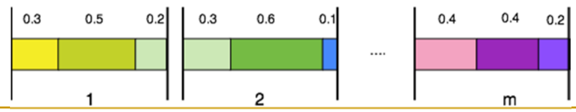
- **Lemma 3**
- If tasks in T are scheduled within a time slice by DP-Fair scheduling and $R_T \leq m$ at all times $t \in \sigma_i$, then all tasks in T will meet their local deadline at the end of the slice
- **Lemma 4**
- If a task set T of periodic tasks with implicit deadlines is scheduled in σ_i using DP-Fair algorithm, then $R_T \leq m$ will hold at all times $t \in \sigma_i$

A DP-Fair algorithm: DP-Wrap /1

- Make blocks of length δ_i for each τ_i and line these blocks up along a number line (in any order), starting at zero

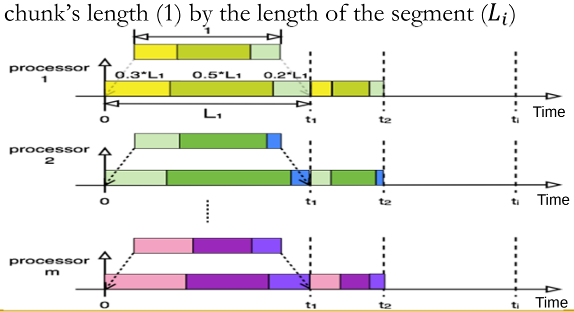


- Split this stack of blocks into chunks of length 1 at 1,2,...,m − 1



A DP-Fair algorithm: DP-Wrap /2

- Use deadline partitioning to divide time into slices
- Assign each chunk to its own processor and multiply each chunk's length (1) by the length of the segment (L_i)

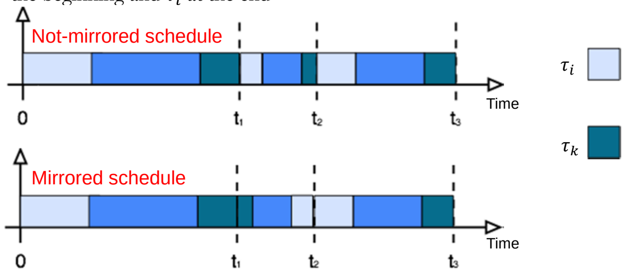


DP-Wrap features

- A very simple algorithm that satisfied all DP-Fair rules
- Almost all calculations can be done in a preprocessing step (with static task sets)
- No computational overhead at secondary events
- $n-1$ context switches and $m-1$ migrations per slice with *mirroring*
- Heuristics may exist to improve performance
  - Less migration and context switches

Mirroring

- For tasks that split across two slices
- If τ_i and τ_k are split and τ_i executes at the beginning and τ_k executes at the end of the slice σ_j then revert the schedule in slice σ_{j+1} so that τ_k executes at the beginning and τ_i at the end



Sporadic tasks and $D_i \le p_i$

- DP-Fair algorithms are still optimal when $\Delta(T) \le m$ and $\delta_i \le 1\ \forall i$
- Definitions
  - *Freeing slack*: unused capacity $(a_{i,h-1} + D_{i,a_{i,h}})$
  - *Active*: $(a_{i,h}, a_{j,h} + D_i)$
  - $\alpha_{i,j}(t)$, $f_{i,j}(t)$: amounts of time that task τ_i has been active or freeing slack during slice σ_j as of time t
  - *Local capacity*: $c_{i,t_{j-1}} = \delta_i \times L_i = \delta_i(\alpha_{i,j} + f_{i,j})$
  - *Freed slack* in σ_j as of time t: $F_j(t) = \sum_{i=1}^{n}(\delta_i \times f_{i,j}(t))$
  - *Slack:* $S(T) = m - \Delta(T)$

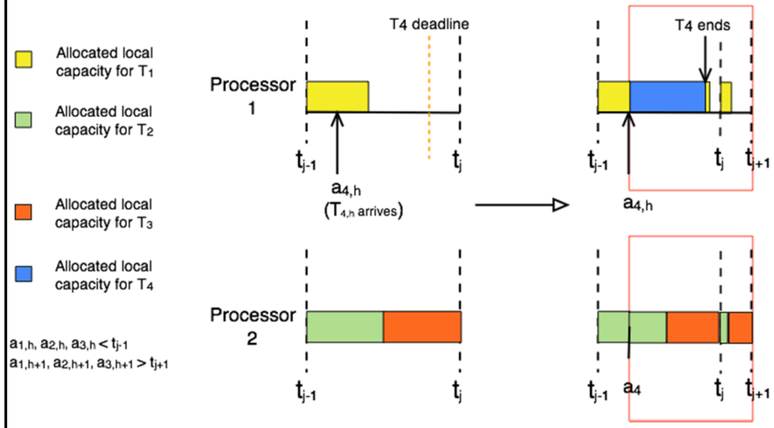
DP-Fair scheduling for time slices /1

- A slice-scheduling algorithm is DP-Fair if it schedules jobs within a time slice σ_i according to the following rules:
  1. Always run a job with zero local laxity
  2. Never run a job with no remaining local work
  3. Do not allow more than $S(T) \times L_j + F_j(t)$ units of idle time to occur in σ_i before time t
  4. Initialize $l_{i,t_{j-1}}$ to 0. At the start time t' of any active time segment for τ_i in σ_j (either $t' = t_{j-1}$ or $a_{i,h}$) that ends at time $t" = min\left\{a_{i,h} + D_{i,t_j}\right\}$, increment $l_{i,t}$ by $\delta_i(t" - t')$

DP-Fair scheduling for time slices /2

- Rules continued …
  5. When a task τ_i arrives in a slice σ_j at time t and its deadline falls within σ_j
     - Split the remainder of σ_j after t into two secondary slices σ_j^1, σ_j^2 so that the deadline of τ_i coincides with the end of σ_j^2
     - Divide the remaining local execution (and capacity) of all jobs in σ_j^1 (as well as the slack allotment from RULE 3) proportionally to the lengths of σ_j^1, σ_j^2
     - This step may be invoked recursively for any τ_k within σ_j

DP-Fair scheduling for time slices /3



Correctness

Theorem 9

Any DP-Fair scheduling algorithm is optimal for sporadic task sets with constrained deadlines where $\Delta(T) \leq m$ and $\delta_i \leq 1\ \forall i$

Proof

Lemma 7

A DP-Fair algorithm cannot cause more than $S(T) \times L_j + F_j(t)$ *units of idle time in slice* σ_j *prior to time* t

Lemma 8

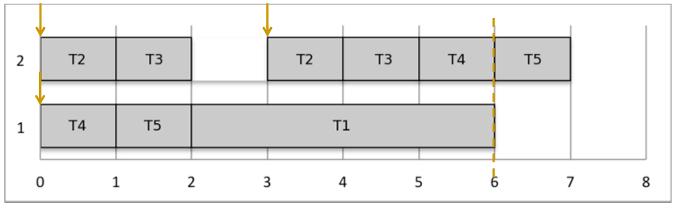
If a set T *of sporadic tasks with constrained deadlines is scheduled in* σ_j *using a DP-Fair algorithm, then* $R_t \leq m$ *will hold at all times* $t \in \sigma_j$

DP-Wrap modified

- If task τ_i issues a job at time t in slice σ_j and $t + D_i > t_j$ then allocate execution time $l_{i,t} = \delta_i(t_j - t)$ following RULE 4
- If instead $t + D_i < t_j$ then split the remainder of σ_j following RULE 5

Arbitrary deadlines /1

- Task set T below is <u>not</u> feasible on 2 processors
  - $m = 2, T = \{\tau_1 = (6,4), \tau_2 = \tau_3 = \tau_4 = \tau_5 = (3,1,6)\}$
  - $\Delta(T) = \frac{4}{6} + 4 \times \frac{1}{3} = 2$
  - 12 units of work to be completed by time 6



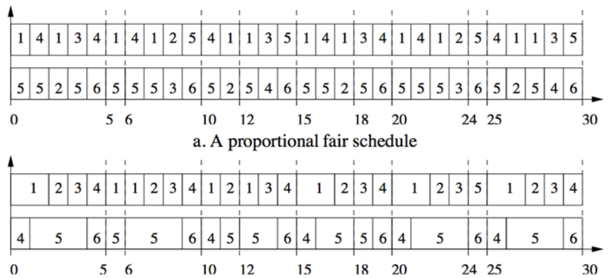
Arbitrary deadlines /2

- Is there a cure to this problem?
- If task τ_i has $D_i > p_i$ we simply impose an artificial deadline $D'_i = p_i$
- Density is not increased hence if D'_i is met, D_i will also be
- But this increases the number of context switches and migrations!

Related work: Boundary Fair /1

- Very similar to P-Fair
  - It still uses a function and a characteristic string to evaluate the fairness of tasks [4] with per-quantum task allocation
- It uses deadline partitioning
- It uses a less strict notion of fairness
  - At the end of every slice the absolute value of the allocation error for any task τ_i is less than one time unit
- Scheduling decisions made at the start of every slice
  - It reduces context switches packing two or more allocated time units of processor to the same task into consecutive units

Related work: Boundary Fair /2



a. A proportional fair schedule

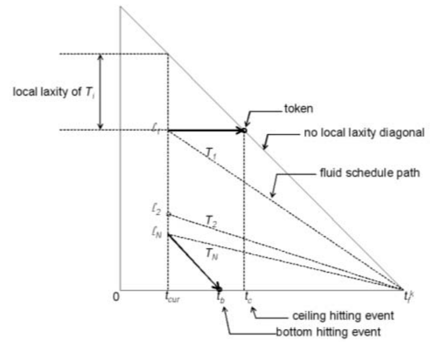
b. A boundary fair schedule

- Not DP-Fair but DP-Correct

Related work: LLREF [5] /1

- It uses deadline partitioning with DP-Wrap task allocation
- In each slice scheduling is made using the notion of T-L Plane
  - Each task T_j is represented by a token within a triangle and its position stands for the local remaining work of T_J at time i
  - The horizontal cathetus indicates the time
  - The length of the vertical cathetus is one processor's execution capacity
  - The hypotenuse represents the-no laxity line
  - Token can move in two directions. Horizontally if the task doesn't execute, diagonally down if it does
  - When a token hits the horizontal cathetus or the hypotenuse (secondary events) a scheduling decision is made
    - Tasks are sorted and m tasks with the least laxity are executed

Related work: LLREF /2



- DP-Fair algorithm but does unnecessary work

Related work: EKG [6]

- Tasks are divided into heavy and light
  - Each heavy task is assigned to a dedicate processor
  - Every light task is assigned to one group of K processors and it shares them with other light tasks
- Some light tasks are split in two processors and they are executed either before t_a or after t_b
- Light tasks that are not split are executed between t_a or and t_b and they are scheduled by EDF
- Heavy tasks start executing when they become ready
- EDF is not a DP-Fair allocation but the DP-Fair rules are satisfied

Comparisons with DP-Wrap /1

- DP-Wrap causes about 1/3 as many context switches and migrations as LLREF
- LLREF has some inefficiencies ([7],[8])
  - Inefficiencies stem from the non working-conservative propriety
  - BF and EKG should show improvements comparable to DP-Wrap
- EKG with appropriately tuned k parameter should outperform DP-Wrap and BF on task set with $U(T) < m$

Comparisons with DP-Wrap /2

- Algorithmic complexity
  - DP-Wrap is the best. *O(n)* work at the beginning and then each event just requires a constant time lookup
  - LLREF is $O(n^2)$
  - EKG is *O(n log n)* but is more efficient in practice
  - BF is *O(n)* per slice

Is DP-Fair scheduling sustainable? /1

- Consider model with sporadic tasks and arbitrary deadline
- Two cases may occur
  - The new value of the relaxed parameter is not used in the scheduling and allocation policies
  - The new value of the relaxed parameter becomes known a priori/at job arrival and it is used in the scheduling and allocation policies

Is DP-Fair scheduling sustainable? /2

- Shorter execution time
  - *Case 1 (shorter* c*, same density)*
    - Task set T is schedulable and the system allocates $\delta_i \times L_j$ workload per each task in each slice
    - If $c'_i \leq c_i$ then task τ_i uses part of assigned workload and surely completes before its deadline
  - *Case 2 (shorter* c*, lesser density)*
    - As DP-Fair is optimal when $\Delta(T) \leq m$ and $\delta_i \leq 1 \; \forall i = 1,..n$ a DF-Fair feasible schedule exists for T
    - A feasible schedule for T' exists as $c'_i < c_i \Rightarrow \delta'_i < \delta_i \Rightarrow \Delta(T') < D(T)$

Is DP-Fair scheduling sustainable? /3

- Longer inter-arrival time
  - *Case 1 (longer* p*, same density)*
    - Simply a less demanding instance of sporadic task
    - The allocation and scheduling rules cover this case
  - *Case 2 (longer* p*, lesser density)*
    - If $p'_i > p_i$ *and* $\delta'_i < \delta_i$ *then* $\Delta(T') < \Delta(T)$ whereby T' is feasible if T was feasible

Is DP-Fair scheduling sustainable? /4

- Longer deadline
  - *Case 1 (longer d, same density)*
    - $d_i < d'_i$
    - Task τ'_i completes its workload at time $t = \min(d_i, p_i)$
  - *Case 2 (longer d, lesser density)*
    - If $d'_i > d_i$ *and* $\delta'_i < \delta_i$ *then* $\Delta(T') < \Delta(T)$ whereby T' is feasible if T was feasible
- We may therefore conclude that DP-Fair scheduling is sustainable

Useful DP-Fair bibliography

1. C. Liu and J. Layland, “Scheduling Algorithms for Multi-programming in a Hard-Real-Time Environment”, Journal of the ACM (JACM), 20(1):46–61, 1973
2. A. K. Mok, “Fundamental design problems of distributed systems for the hard-real-time environment”, Technical report, Massachusetts Institute of Technology, 1983
3. S. K. Cho, S. Lee, A. Han, and K.-J. Lin, “Efficient Real- Time Scheduling Algorithms for Multiprocessor Systems”, IEICE Transactions on Communications, E85-B(12):2859– 2867, 2002
4. D. Zhu, D. Mossé and R. Melhem, “Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary?”, IEEE Real-Time Systems Symposium (RTSS), 2003
5. H. Cho, B. Ravindran and E. Jensen, “An Optimal Real-Time Scheduling Algorithm for Multiprocessors”, IEEE Real-Time Systems Symposium (RTSS), 2006
6. B. Andersson and, E. Tovar, “Multiprocessor Scheduling with Few Preemptions”, IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA), 2006
7. K. Funaoka, S. Kato and N. Yamasaki, “Work-Conserving Optimal Real-Time Scheduling on Multiprocessors” Euromicro Conference on Real-Time Systems (ECRTS), 2008
8. S. Funk and V. Nadadur “LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets”, Conference on Real-Time and Networked Systems (RTNS), 2009

Other results /1

- For the simplest workload model made of independent periodic and sporadic tasks
- A *P-fair* scheme can sustain $U = m$ for m processors but its run-time overheads are excessive
  - Especially because tasks incur very many preemptions and are frequently required to migrate across processors
- *Partitioned FPS first-fit* (on decreasing task utilization) can sustain $U \leq m(\sqrt{2} - 1)$
  - But this is a sufficient test only [Oh & Baker, 1998]

Other results /2

- *Partitioned EDF first-fit* can sustain

$$U \leq \frac{\beta m + 1}{\beta + 1}$$

$$\beta = \left\lfloor \frac{1}{U_{\max}} \right\rfloor$$

Per task

- For high U_{max} this bound gets rapidly lower than $0.6 \times m$, but can get close to m for some examples
  - Again this is a sufficient test only [Lopez *et al.*, 2004]

Other results /3

- *Global EDF* can sustain

$$U \le m - (m-1)U_{\max}$$

- For high U_{max} this bound can be as low as $0.2 \times m$ but also close to m for other examples
  - Again, only sufficient [Goossens *et al.*, 2003]

Other results /4

- Combinations
  - FPS (higher band) to those tasks with $U_i > 0.5$
  - EDF for the rest

$$U \le \left(\frac{m+1}{2}\right)$$

  - Again, only sufficient [Baruah, 2004]

Multiprocessor PCP /1

- Partitioned FPS with resources bound to processors [Sha, Rajkumar, Lehoczky, 1988]
  - The processor that hosts a resource is called the *synchronization processor* (SP) for that resource
    - It knows all the use requirements of all its resources
  - The critical sections of a resource execute on the processor that hosts that resource
    - Jobs that use *remote* resources are "distributed transactions"
  - The processor to which a task is assigned is the *local processor* for all of the jobs of that task

Multiprocessor PCP /2

- A task may need local and global resources
  - Local resources reside on the local processor of that task
  - Global resources are used by tasks residing on different processors
- Resource access control needs actual locks for protection from true parallelism
  - Lock-free algorithms then become attractive
- SP use M-PCP to control access to their global resources

Multiprocessor PCP /3

- The task that holds a global lock should not be preempted locally
  - All global critical sections are executed at higher ceiling priorities than local tasks on the SP and any other tasks in the system
- A task τ_h that is denied access to a global shared resource ρ_g suspends and waits in a priority-based queue for that resource
  - Tasks with lower-priority than τ_h on its local processor may thus acquire global resources with higher ceiling

Multiprocessor PCP /4

- If the global resource being acquired by task τ_l with priority lower than τ_h resides on the same SP as ρ_g then τ_h suffers an anomalous form of priority inversion
  - This obviously exposes resource nesting to the risk of deadlock → M-PCP disallows resource nesting
  - This is the reason why other protocols want τ_h to spin
- With global resources hosted on > 1 SP resource nesting is not allowed as deadlock may occur

Blocking under M-PCP

- With M-PCP task τ_i is *blocked* by lower-priority tasks in 5 ways (!)
  - *Local blocking* (once per execution): when finding a local resource held by a local lower-priority task that got running as a consequence of τ_i suspension on access to a remote resource
  - *Remote blocking* (once per access): when finding a remote resource held by remote lower-priority tasks
  - *Local preemption*: when global critical sections are executed on τ_i's processor by remote tasks of any priority (multiple times) and by local tasks of lower priority (once)
  - *Remote preemption* (once per access): when higher-ceiling global critical section execute on the remote processors where τ_i needs a global resource
  - *Deferred interference* as local higher-priority tasks suspend on access to remote resources because of blocking effects

Multiprocessor SRP

- Partitioned EDF with resources bound to processors [Gai, Lipari, Di Natale, 2001]
  - SRP is used for controlling access to local resources
  - Tasks that lock a global resource cannot be preempted
    - They become preemptable again when releasing the resource
  - Tasks that request a global resource that is busy are placed in a FIFO queue on the synchronization processor and spin-lock on their local processor
    - On release from the task that held it the global resource is assigned to the task (request) at the head of the queue

MrsP [Burns, Wellings, 2013] /1

- With lock-based resource control protocols locks can use either *suspension* or *spinning*
- With suspension the calling task that cannot acquire the lock is placed in a priority-ordered queue
  - To bound blocking time priority-inversion avoidance algorithms are used
- With spinning the task busy-waits
  - To bound blocking time the spinning task becomes non-preemptable and its request is placed in FIFO queue
- The lock owner may run non-preemptively

MrsP [Burns, Wellings, 2013] /2

- RTA for a partitioned multiprocessor should be *identical* to the single-processor case
  - The cost of accessing global resources should be *increased* to reflect the need to serialize parallel contention
- The property that once a task starts executing its resources *are* available is intrinsic to RTA
  - It should therefore be supported by global resource control protocols
    - Which speaks against suspension-based solutions!

MrsP [Burns, Wellings, 2013] /3

- Spinning non-preemptively may decrease feasibility
  - More urgent tasks suffer longer blocking
- Spinning at the *local* ceiling priority is better
  - With all processors using PCP/SRP at most one task per processor may contend globally
  - Access requests are served in FIFO order
- To bound blocking from preemption of the lock-holder task, spinning tasks should "donate" their cycles to it
    - The lock-holder job migrates to the processor of a spinning task and runs in its stead until it either completes or migrates again

MrsP [Burns, Wellings, 2013] /4

- Resource nesting can be supported with either *group locking* or *static ordering* of resources
  - With static ordering, resource access is allowed only with order number greater than any currently held resources
  - The implementation should provide an «out of order» exception to prevent run-time errors
- The ordering solution is better than banning nesting and has less penalty than group locking

OMIP [Brandenburg, 2013]

- **Theorem**
  - Under non-global scheduling (for clusters of size $c < m$) it is *impossible* for a resource access control protocol to simultaneously:
    - Prevent unbounded priority-inheritance blocking
    - Be independence-preserving
      - Tasks do not suffer PI-blocking from resources they do not use
    - Avoid inter-cluster job migration
- Seeking independence preservation and bounded PI-blocking requires inter-cluster job migration (!)

2013/14 UniPD / T. Vardanega Real-Time Systems 422 of 423

Summary

- Issues and state of the art
- Dhall's effect: examples
- Scheduling anomalies: examples
- P-fair scheduling
- Sufficient tests for simple workload model
- Recent extensions [2010]: DP-Fair
- Incorporating global resource sharing

2013/14 UniPD / T. Vardanega Real-Time Systems 423 of 423