CS423 Special Topics Lecture

Multicore Scheduling

Antonio Franques
Senthil Kumar Karthik
Aakash Ketan Modi
Dong Hun Lee
Dongming Lei
Motivation to the topic

Antonio Franques
Motivation - How to map tasks into processors?

- Single-processor systems:
Motivation - How to map tasks into processors?

- **Single-processor systems:**
  - N threads for 1 processor → no mapping required
  - Local protocol-dependent time interleaving
Motivation - How to map tasks into processors?

- Multiprocessor systems:
Motivation - How to map tasks into processors?

- **Multiprocessor systems:**
  - $N$ threads for $M$ processors $\rightarrow$ **many factors to be considered**
  - **Global** protocol-dependent time interleaving

![Multi-core scheduling protocol diagram]
Motivation - How to map tasks into processors?

- Multiprocessor systems - Factors to be considered:
  - Delay of remote communications → How to predict it?

  Fairly easy
Motivation - How to map tasks into processors?

- Multiprocessor systems - Factors to be considered:
  - Delay of remote communications → How to predict it?
  - Very complex
Motivation - How to map tasks into processors?

- Multiprocessor systems - Factors to be considered:
  - Synchronization and task dependencies → How to minimize its overhead?
    - Place collaborating threads in same processor or nearby
    - Avoid far-apart delays
Motivation - How to map tasks into processors?

- **Multiprocessor systems - Factors to be considered:**
  - Migration costs
    - Upon thread relocation, cached data is lost
    - Many cold (compulsory) cache misses
    - (Re) consider if thread replacement is worth the cost
Motivation - How to map tasks into processors?

- **Multiprocessor systems** - Factors to be considered:
  - Memory interference → simultaneous accesses to same data
    - If threads within same processor → exploit cache time locality
    - If threads in different processors → unnecessary multiple requests to memory
Motivation - How to map tasks into processors?

- Multiprocessor systems - Factors to be considered:
  - Memory interference → simultaneous accesses to same data
    - If threads within same processor → exploit cache time locality
    - If threads in different processors → unnecessary multiple requests to memory
  - Multicore scheduling protocols must consider all these factors.
  - Otherwise, overall execution times of applications might be compromised.
Outline:

- Global Scheduling Schemes
- Partitioned Schemes
- Multiprocessor Synchronization
- Parallel Task Scheduling
References

● Software Engineering Institute - Carnegie Mellon University
  ○ http://www.sei.cmu.edu/cyber-physical/research/timing-verification/multicore-scheduling.cfm

● Scheduling and Synchronization for Multi-core Real-time Systems

● Lecture 27, Multi-Core Scheduling - Coursera
  ○ https://www.coursera.org/learn/real-time-systems/lecture/A1o2U/multi-core-scheduling-shared-resources

● Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform
Global Scheduling Schemes

Senthil Kumar Karthik
Global Scheduling

- Tasks stored in one global queue, shared among all processors
- Also called 1-queue-m-server model
- Preemption and task migration from processor to another is enabled
- Overview:
  - \( m \)-highest priority tasks running on \( m \) different cores or processors at any given instant of time

- Problems?

![Diagram of requests and queues with servers and processed tasks]
Global Scheduling

- Tasks stored in one global queue, shared among all processors
- Also called 1-queue-m-server model
- Preemption and task migration from processor to another is enabled
- Overview:
  - $m$-highest priority tasks running on $m$ different cores or processors at any given instant of time

- Problems?
  - Generate too many preemptions and migrations
  - Increased network and cache traffic
  - Low utilization bounds for some task sets
Global Scheduling Algorithms

Elaborate on 4 significant real-time multiprocessor scheduling algorithms under the global scheme:

1) Global EDF scheduling policy
2) EDZL scheduling policy
3) P-Fairness scheduling algorithm
4) DP-Wrap scheduling algorithm
Global EDF Scheduling Policy

- Earliest Deadline First policy applied on a global set of tasks executed on a set of processors
- All ready tasks are stored into a global ready-queue
- At each time $t$, the $m$ highest-priority ready tasks are executed on $m$ processors
- Best explained with an example
  - Consider the 4 tasks shown below to be scheduled on 3 processors

<table>
<thead>
<tr>
<th>Task</th>
<th>Total CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
Global EDF is not efficient for many task sets. As shown in the above example, there can be cases when a task (Task \( \tau_1 \)) misses its deadline due to the scheduling policy. Not suitable for real-time task sets with hard deadlines.
EDZL Scheduling Policy

Background

- **Earliest Deadline first until Zero Laxity**
- It combines two uniprocessor scheduling algorithms

\[
\text{EDZL} = \text{EDF} + \text{LLF}
\]

- What’s LLF scheduling algorithm?
EDZL Scheduling Policy

Background

- **Earliest Deadline first until Zero Laxity**
- It combines two uniprocessor scheduling algorithms

EDZL = EDF + LLF

- What’s LLF scheduling algorithm?
  - Least Laxity First is a scheduling algorithms that assigns highest priority to the task with least laxity
  - It is a measure of urgency of real-time tasks
  - If laxity of some task is zero it must be scheduled immediately or it will miss its deadline

Laxity (at time t) = Time to Deadline - Remaining execution time
EDZL Scheduling Policy

EDZL combines EDF and LLF such that it behaves as EDF if there is no task with zero laxity.

Disadvantage:
- EDZL is not optimal as it cannot schedule all feasible task sets.
**P-Fairness Scheduling Algorithm**

- Divide time into quanta
- Divide tasks into subtasks
- Pseudo release and deadline
- Problems?

**Problems:**
- Lot of preemptions and migrations due to large number of subtasks
- Complex computations are needed for decision making
DP-Wrap Scheduling Algorithm

Divide time into unequal slots

Compute workload for each task

Advantage:● Reduces number of scheduling points by 75% when compared to P-Fairness algorithm

Problem:● Still needs complex computations to make scheduling decision

Bin-packing heuristics
References

- Real-Time Scheduling on Multi-core: Theory and Practice, Paulo Baltarejo Sousa, CISTER
  - http://www.cister.isep.ipp.pt/docs/real_time_scheduling_on_multi_core__theory_and_practice/808/

Partitioned Scheduling

Schemes

Aakash Ketan Modi
Partitioned Scheduling

- Tasks stored in multiple local queues, typically 1 per processor
- Also called $m$-queue-$m$-server model
Partitioned Scheduling

- Tasks stored in multiple local queues, typically 1 per processor
- Also called m-queue-m-server model

**Strategy:**
- Global Schemes: Any instance of a task can be executed on any processor, or even be preempted and moved to a different processor before it is completed
- Partitioned Schemes: Once a task is allocated to a processor, it is executed exclusively on that processor.
Partitioned Scheduling - Advantages

- The scheduling overhead associated with a partitioning strategy is lower than the overhead associated with a global strategy.
- Partitioning strategies allow us to apply well-known uniprocessor scheduling algorithms to each processor.
- Earliest Deadline First (EDF) scheduling, which is an optimal uniprocessor scheduling algorithm, perform poorly when extended to global multiprocessor scheduling.
Partitioned Scheduling Algorithms - Outline

- Assigning processors to tasks using partitioned schemes is similar to the bin-packing problem:
  - Objects of different volumes must be packed into a finite number of bins or containers in a way that minimizes the number of bins used
Partitioned Scheduling Algorithms - Outline

- Assigning processors to tasks using partitioned schemes is similar to the bin-packing problem:
  - Objects of different volumes must be packed into a finite number of bins or containers in a way that minimizes the number of bins used.
  - Here:
    - Objects = Task
    - Bin = Processor
Partitioned Scheduling Algorithms - Outline

- Assigning processors to tasks using partitioned schemes is similar to the bin-packing problem:
  - Objects of different volumes must be packed into a finite number of bins or containers in a way that minimizes the number of bins used.
  - Here:
    - Objects = Task
    - Bin = Processor
  - In other words, we need to assign tasks to processors such that they all meet deadlines as well as minimize the number of processors used.
Bin-packing Algorithm for Partitioned Scheduling

1. Sort the tasks according to some criterion (EDF, SJF, RoundRobin etc.)
2. Select the first task and an arbitrary processor
3. Attempt to assign the selected task to the selected processor by applying a schedulability test (e.g. utilization bound test) for that processor
4. If the schedulability test fails, select the next available processor; if it succeeds, select the next task
5. Goto step 3
Partitioned Scheduling Algorithms

- Partitioned Fixed Priority w/o Task Splitting
- Partitioned Fixed Priority w/ Task Splitting
- Partitioned Dynamic Priority w/o Task Splitting
- Partitioned Dynamic Priority w/ Task Splitting
Partitioned Scheduling Algorithms - Utilization

- Utilization Bound for a given set of tasks is defined as the maximum CPU utilization that can be achieved using the most optimal scheduling scheme after which at least 1 task is guaranteed to miss the deadline.

- Utilization Bound for Partitioned Schemes
  - w/o task splitting ≤ 50%
  - w/ task splitting ≤ 65%
Design Challenge - Global v/s Partitioned Scheme

● Utilization Bound (higher the better) for
  ○ Global Schemes ≤ 85%
  ○ Partitioned Schemes ≤ 65%

● More overhead in migration and preemption in global schemes compared to partitioned schemes

● Hence, trade-off while designing an algorithm is between
  ○ Utilization of the System v/s Cost of Overhead

● Need to consider the trade-off based on the requirements of the applications running on your system
References

- Scheduling and Synchronization for Multi-core Real-time Systems, PhD Thesis


Multiprocessor Scheduling Synchronization

Dong Hun Lee
Multiprocessor Scheduling Synchronization

- Why is synchronization needed?
Multiprocessor Scheduling Synchronization

- Resource/Task dependency

Traditional Synchronization Protocols

- Uniprocessor Synchronization
  - Priority Inheritance
  - Priority Ceiling

- Problem?
Problem with Traditional Synchronization

● Problem with traditional protocols in multiprocessor environment
  ○ Locks mutually exclusive resource
    ■ Blocks dependant processors
    ■ Extreme performance drop
  ○ Sharing mutually exclusive resource
    ■ Data race

● Processors communication
  ○ Task mapping
Challenges in Multiprocessor Synchronization

- Remote blocking
  - Resource sharing across processors
  - Wastes processor utilization
Challenges in Multiprocessor Synchronization

- Remote blocking
  - Resource sharing across processors
  - Wastes processor utilization

Global critical section requires synchronization of all processors, blocking all processors except for one in the critical section. What if the processor in global critical section becomes blocked by another critical section? We want to avoid having global critical section as much as possible.

Synchronization overhead is another factor that needs to be taken into account for multiprocessor synchronization protocol. Minimizing the total cost of synchronization within the program is desired.

Inter-processor communication can be a significant factor in increasing the cost of the total synchronization in multiprocessor environment.
Challenges in Multiprocessor Synchronization

- Synchronization aware task mapping
- Global and Local Synchronization
- Task Granularity

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine</td>
<td>Parallelism Inherent in a single stream of instruction</td>
</tr>
<tr>
<td>Medium</td>
<td>Parallel processing or multitasking within a single application</td>
</tr>
<tr>
<td>Coarse</td>
<td>Multiprocessing of concurrent processes in a multiprogramming environment</td>
</tr>
<tr>
<td>Very Coarse</td>
<td>Distributed processing across network nodes to form a single computing environment</td>
</tr>
<tr>
<td>Independent</td>
<td>Multiple unrelated process</td>
</tr>
</tbody>
</table>
Challenges in Multiprocessor Synchronization

- **Task Granularity**
  - Task granularity determines the frequency of the synchronization between processors
  - Smaller tasks could require more frequent synchronization, but the potential remote blocking time would be shorter
  - Larger tasks could require less frequent synchronization, but the potential remote blocking time would be longer
  - If the bottleneck of performance is the synchronization overhead, larger tasks would be preferred to invoke less synchronization
  - On the other hand, if the blocking time is the primary concern, fine grain tasks would allow less blocking time per task dependency
Multiprocessor Synchronization Protocols

- Multiprocessor Priority Ceiling Protocol (MPCP)
- Multiprocessor Stack-based Resource allocation Protocol (MSRP)
- Flexible Multiprocessor Locking Protocol (FMLP)
- Multiprocessor Hierarchical Synchronization Protocol (MHSP)
Multiprocessor Synchronization Protocols

- **Multiprocessor Priority Ceiling Protocol (MPCP)**
  - Distinguishing global and local synchronization
    - Using global and local mutex
  - Assigns global critical section the highest priority
    - Global critical section will not be blocked by local tasks
  - Blocked global tasks are inserted to global priority queue
  - Minimizes remote blocking and priority inversion when global resource dependency exists
  - Global and local critical sections cannot be nested in each other
  - Resource dependency based task mapping
References

- Multiprocessor Synchronization and Hierarchical Scheduling, Farhang Nemati, Moris Behnam, Thomas Nolte. Mälardalen Real-Time Research Centre


- Scheduling and Synchronization on Multicores, Michal Risa, Brno University of Technology

- Scheduling and Synchronization for Multi-core Real-time Systems, Karthik S. Lakshmanan, Carnegie Mellon University
Parallel Task Scheduling

Dongming Lei
Parallel Tasks

● Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.

● Recall from energy lecture
  ○ Physical constraints preventing frequency scaling
  ○ Parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

● Why parallel tasks
  ○ Provide concurrency
  ○ Save time
  ○ Solve larger problems
  ○ Maximize load balancing
  ○ Make a good use of parallel hardware architecture

● Goal: to reduce the execution time of tasks
Workflow

Resources - e.g. I/O operations
Gang Scheduling

Problem?

Gang scheduling is a scheduling algorithm for parallel systems that schedules related threads or processes to run **simultaneously** on different processors.

Groups of related processes are scheduled as a unit, a gang.

All members of a gang run simultaneously, on different timeshared CPUs.

All gang members start and end their time slices together.

For inter-process communication so that two threads communicate with each other at the same time.

Gang EDF (Earliest Deadline First)

- Applies EDF policy to the Gang scheduling scheme:
  - Higher priorities are assigned to the jobs with earlier deadlines
- Limitation on available processors, where the number of ready jobs is chosen on the basis of Global EDF
  - Earliest deadline jobs are selected first and executed concurrently
  - If there are some limitation on the processor due to which the job cannot start executing then on the basis of first fit heuristic it select the next job for execution

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 2</td>
<td>Job 1</td>
<td>Job 2</td>
<td>Job 3</td>
<td>Job 4</td>
</tr>
<tr>
<td>Processor 3</td>
<td>Job 1</td>
<td>*</td>
<td>Job 3</td>
<td>Job 4</td>
</tr>
<tr>
<td>Processor 4</td>
<td>*</td>
<td>*</td>
<td>Job 3</td>
<td>*</td>
</tr>
<tr>
<td>Processor 5</td>
<td>*</td>
<td>*</td>
<td>Job 3</td>
<td>*</td>
</tr>
</tbody>
</table>
Backfilling

- Problem? How to fill these “holes”?
- Backfill is a scheduling optimization which allows a scheduler to make better use of available resources by running jobs out of order.
- Jobs are packed together in order to avoid fragmentation
- It identifies the “holes” in the schedule and fits the small jobs in those holes
- Improved responsiveness and utilization and avoided starvation for large jobs
Aggressive Backfilling

- Jobs take reservation which are on the head of queue
- The small jobs will be allowed to go forward only if they do not delay or affect the jobs on the head of the queue
- Starvation?
  - Cannot occur for the first job since queuing delay for the first job depends only on the running jobs
  - But jobs other than the first may be repeatedly delayed by newly arriving jobs

Figure 2: Aggressive backfilling technique [21].
Conservative Backfilling

- Makes reservations for all queued jobs
- Pushing small jobs if they don’t delay other jobs that are queued then it goes forward in queue
- Backfilling is done subject to checking that it does not delay any previous job in the queue
- Starvation cannot occur

Figure 3: Conservative Backfilling [19].
Reference

https://en.wikipedia.org/wiki/Parallel_computing


http://docs.adaptivecomputing.com/maui/8.2backfill.php

https://www.omicsonline.org/open-access/task-scheduling-in-parallel-processing-a
nalysis-2155-6180-1000257.php?aid=65607
Thank You