Multiprocessor Scheduling

Dr. A. Sahu
CSE, IIT Guwahati

Outline
- Yesterday Class
  - Scheduling
  - Multiprocessor Scheduling
- Today Class
  - Real Time Scheduling
  - Distributed Scheduling
  - Cilk Programming and Work Stealing
  - Scheduling in 2D NOC and 3D NOC

Periodic Task: Real Time Scheduler
- Task with periods
- Each task have to finish before deadline with in the period

Periodic Tasks
- Necessary schedulability test
- Sum of utilization factors $\mu_i$ must be less than or equal to $n$, where $n$ is the number of processors
  - $\mu = \sum \frac{c_i}{p_i} \leq n$
  - $\mu_i = \text{Percentage of time the task } T_i \text{ requires the service of a CPU}$

Periodic Task: Real Time Scheduler
Assumptions & Definitions
- Tasks are periodic
- No aperiodic or sporadic tasks
- Job (instance) deadline = end of period
- Tasks are preemptable
- Laxity of a Task $t_i = d_i - (t + c'_i)$ where $d_i$: deadline; $t$: current time; $c'_i$: remaining computation time.

Rate Monotonic Scheduling
- Task with the smallest period is assigned the highest priority.
- At any time, the highest priority task is executed.
Rate Monotonic Scheduling

- **Schedulability check (off-line)**
  - A set of $n$ tasks is schedulable on a uniprocessor by the RMS algorithm if the processor utilization (utilization test):
    \[
    \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)
    \]
  - The term $n(2^{1/n} - 1)$ approaches $\ln 2$, ($\approx 0.69$ as $n \to \infty$).
  - This condition is **sufficient**, but **not necessary**.

Earliest Deadline First

- **Task with the smallest deadline/laxity is assigned the highest priority. EDF or Least Laxity First (LLF)**
  - At any time, the highest priority task is executed.

- **Schedulability check (off-line)**
  - A set of $n$ tasks is schedulable on a uniprocessor by the EDF algorithm if the processor utilization:
    \[
    \sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1
    \]
  - This condition is both **necessary** and **sufficient**.

RMS & EDF -- Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Period, $T_i$</th>
<th>WCET, $C_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Distributed Scheduling

Centralized Vs Distributed Scheduling

- **Centralized Scheduling**
  - Master distributes work to Servers,
  - Master is responsible to load balancing and Scheduling
  - Easy to handle, not scalable to many core

- **Distributed Scheduling**
  - Every one participate in Scheduling
  - Message and Request based
  - Scalable to many processor
Load distributing algorithms

- Basic function: transfer load (tasks) from heavily loaded computers to idle or lightly loaded computers.
- Can be characterized as:
  - Static: decisions are hard-wired in the algorithm using a priori knowledge of the system.
  - Dynamic: use system state information (the loads at nodes), at least in part.
  - Adaptive: dynamically changing parameters of the algorithm to suit the changing system state.

Load balancing vs. load sharing

- Load sharing:
  - Strive to reduce the likelihood of an unshared state by transferring tasks to lightly loaded nodes.
- Load balancing:
  - A step further by attempting to equalize loads at all computers

Issues In Load Distributing

- Preemptive vs. Nonpreemptive Transfers
  - Preemptive task transfers involve the transfer of a task that is partially executed.
  - Nonpreemptive task transfers involve the transfer of a task that have not begun execution.
- Transfer Policy
  - Determines when a node needs to send tasks to other nodes or can receive tasks from other nodes: Threshold policy
- Location Policy: Find suitable nodes for load sharing
- Information policy
  - Deciding when information about the states of other nodes in the system should be collected
  - where it should be collected from, what information should be collected.
  - Demand-driven, Periodic, State-change-driven

Issues In Load Distributing

- Selection Policy: Determines which task(s) to transfer
  - Newly originated tasks that have caused the node to become a sender by increasing the load at the node > threshold
  - Estimated average execution time for task > threshold
  - Response time will be improved upon transfer
  - Overhead incurred by the transfer should be minimal
  - The number of location-dependent system calls made by the selected task should be minimal

Load Distributing Algorithms

- Sender-Initiated Algorithms
- Receiver-Initiated Algorithms
- Symmetrically Initiated Algorithms
- Adaptive Algorithms

Sender-Initiated Algorithms

- Load distributing activity: Is initiated by an overloaded node (sender) that attempts to send a task to an underloaded node (receiver).
- Transfer policy: uses a threshold policy based on CPU queue length.
- Selection policy: consider only newly arrived tasks for transfer.
Sender-Initiated Algorithms (cont.)

• Location Policy:
  – Random: A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in decision making.
  – Threshold: Polling a node (selected at random) to determine whether it is a suitable receiver. If so the task is transferred to the selected node, which must execute the task regardless of its state.
  – Shortest: Choose the best receiver for a task.
    • A number of nodes (PollLimit) are selected at random and are polled to determine their queue length. If so choose the node with the shortest queue length as the destination for transfer unless its queue length >= T.
    • The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task.

Receiver-Initiated Algorithms

• The load distributing activity is initiated from an underloaded node (receiver) that is trying to obtain a task from an overloaded node (sender).
• Transfer policy: threshold policy based on CPU queue length.
• Selection policy: any of the approaches discussed earlier.
• Location policy:

Symmetrically Initiated Algorithms

• Both senders and receivers search for receivers and senders.
• Advantages and disadvantages of both sender and receiver initiated algorithms.
  → Above average algorithm.

Cilk and Work Stealing Scheduler

• Developed at MIT, Leiserson Group
• Chapter 17, Algorithm Book CLR
• Modified C with dynamic parallelism
• Philosophy
  – The programmer should be responsible for exposing the parallelism, identifying elements that can safely be executed in parallel;
  – it should then be left to the run-time environment particularly the scheduler, to decide during execution how to actually divide the work between processors.

Information policy: demand-driven type.
• Stability: do not cause system instability.
• a drawback: most transfer are preemptive.
Cilk: Fibonacci

```c
int fib (int n) {
    if (n<2) return n;  // Base case
    else {
        int x,y;
        x = fib(n-1);
        y = fib(n-2);
        return (x+y);  // Recursive calls
    }
}
```

Cilk code

Cilk is a faithful extension of C. A Cilk program's serial elision is always a legal implementation of Cilk semantics. Cilk provides no new data types.

**Cilk Example: Fib(4)**

Assume for simplicity that each Cilk thread in `fib()` takes unit time to execute.

- **Work:** $T_1 = 17$
- **Span:** $T_w = 8$
- **Parallelism:** $T_1/T_w = 2.125$

When a Cilk thread completes an edge, it may spawn a new thread. A spawned thread may continue or return.

Cilk Run Time Scheduler

- Distributed load balancing
- Receiver initiated
- Work stealing
  - Free processor steal a task of busy processor
  - When ever a process spawns a new process,
    - This processor starts executing the spawned one
    - Parent goes to waiting/suspend mode
    - Parent can be transferred to other processor

Work stealing

- Optimal algorithm for load balancing
  - If select victim randomly algorithm is Optimal **Proved**
- Basic assumption in work stealing
  - All the memory access are take same time
  - UMA (Uniform Memory Access): shared memory
  - Can be feasible iff
    - Task transfer time is same for all pair of processor
    - Communication bandwidth is same for all pair of processor
**Parallelizing Vector Addition**

```c
void vadd (real *A, real *B, int n)
{
    int i; for (i=0; i<n; i++) A[i]=B[i];
}
```

**Parallelization strategy:**
1. Convert loops to recursion.
2. Insert Cilk keywords.

---

```c
void vadd (real *A, real *B, int n)
{
    int i; for (i=0; i<n; i++) A[i]=B[i];
}
```

```cilk
void vadd (real *A, real *B, int n)
{
    int i; for (i=0; i<n; i++) A[i]=B[i];
    vadd(A, B, n/2);
    vadd(A+n/2, B+n/2, n-n/2);
    sync;
}
```

---

**Vector Addition Analysis**

To add two vectors of length \( n \), where \( BASE = \Theta(1) \):

- **Work:** \( T_1 = \Theta(n) \)
- **Span:** \( T_1 = \Theta(\lg n) \)
- **Parallelism:** \( T_1/T_1 = \Theta(n/\lg n) \)

---

**Cilk Run Time Scheduler**

![Cilk Run Time Scheduler Diagram]

---

**Compiling & executing Cilk Program**

- **Cilk source**
  - `cilk2c` (source-to-source translator)
  - `C` compiler
  - `gcc` (C post-source)

`cilk2c` translates straight C code into identical C post source.

**The Cilk compiler encapsulates the process.**

- Num Core & input data
- Linking loader
- Output
- Task transfer
- Task return
- Request for task
Task Mapping into 2D/3D mesh

- Finding proper regions (Set of near by processors) to map
- Methods:
  - Always try to find a approximate Circular Region
  - Take own optimal space (Reduce internal fragmentation/Communications) and give space to others (Reduce external fragmentation/Communication)
  - Using Dispersion factor, Convex factor: Minimize L(R) and L(R-R')

Task Mapping

- Finding proper regions (Set of near by processors and memory element to map)

Task Mapping

QoS Based Scheduling

- QoS Based scheduling and allocation of LCMP
  - Application on 5 cores take 100 Sec, 10 core 605..
  - How to schedules an QoS for many appl. online
  - 2D and 3D region scheduling of Grids

Manage with 10 Processor
Or Change the allocation
- Suppose processor are configurable
  - Processor & memory parameters for different application may be different
- Learning based multiprocessor performance tuning algorithms
  - Numerical problem takes \( N \) iterations
  - Learn optimal architectural params in first \( m \) iter and used in last \((n-m)\) iteration to reduce time.