Overview of Real-Time Scheduling

Real-Time and Embedded Systems (M)
Lecture 3
Lecture Outline

• Overview of real-time scheduling algorithms
  – Clock-driven
  – Weighted round-robin
  – Priority-driven
    • Dynamic vs. static
    • Deadline scheduling: EDF and LST
    • Validation

• Outline relative strengths, weaknesses

Material corresponds to chapter 4 of Liu’s book
Approaches to Real-Time Scheduling

Different classes of scheduling algorithm used in real-time systems:

• Clock-driven
  – Primarily used for hard real-time systems where all properties of all jobs are known at design time, such that offline scheduling techniques can be used

• Weighted round-robin
  – Primarily used for scheduling real-time traffic in high-speed, switched networks

• Priority-driven
  – Primarily used for more dynamic real-time systems with a mix of time-based and event-based activities, where the system must adapt to changing conditions and events

Look at the properties of each in turn…
Clock-Driven Scheduling

• Decisions about what jobs execute when are made at specific time instants
  – These instants are chosen before the system begins execution
  – Usually regularly spaced, implemented using a periodic timer interrupt
    • Scheduler awakes after each interrupt, schedules the job to execute for the next period, then blocks itself until the next interrupt
    • E.g. the helicopter example with an interrupt every $\frac{1}{180}$th of a second
    • E.g. the furnace control example, with an interrupt every 100ms

• Typically in clock-driven systems:
  – All parameters of the real-time jobs are fixed and known
  – A schedule of the jobs is computed off-line and is stored for use at run-time; as a result, scheduling overhead at run-time can be minimized
  – Simple and straight-forward, not flexible

[Will discuss in more detail in lecture 4]
Weighted Round-Robin Scheduling

• Regular *round-robin* scheduling is commonly used for scheduling time-shared applications
  – Every job joins a FIFO queue when it is ready for execution
  – When the scheduler runs, it schedules the job at the head of the queue to execute for at most one time slice
    • Sometimes called a quantum – typically $O(\text{tens of ms})$
  – If the job has not completed by the end of its quantum, it is preempted and placed at the end of the queue
  – When there are $n$ ready jobs in the queue, each job gets one slice every $n$ time slices ($n$ time slices is called a round)

• Only limited use in real-time systems
Weighted Round-Robin Scheduling

• In *weighted round robin* each job $J_i$ is assigned a weight $w_i$; the job will receive $w_i$ consecutive time slices each round, and the duration of a round is $\sum_{i=1}^{n} w_i$
  – Equivalent to regular round robin if all weights equal 1
  – Simple to implement, since it doesn’t require a sorted priority queue

• Partitions capacity between jobs according to some ratio

• Offers throughput guarantees
  – Each job makes a certain amount of progress each round
Weighted Round-Robin Scheduling

• By giving each job a fixed fraction of the processor time, a round-robin scheduler may delay the completion of every job
  – A precedence constrained job may be assigned processor time, even while it waits for its predecessor to complete; a job can’t take the time assigned to its successor to finish earlier
  – Not an issue for jobs that can incrementally consume output from their predecessor, since they execute concurrently in a pipelined fashion
    • E.g. Jobs communicating using Unix pipes
    • E.g. Wormhole switching networks, where message transmission is carried out in a pipeline fashion and a downstream switch can begin to transmit an earlier portion of a message, without having to wait for the arrival of the later portion

• Weighted round-robin is primarily used for real-time networking; will discuss more in lecture 16
Priority-Driven Scheduling

• Assign priorities to jobs, based on some algorithm
• Make scheduling decisions based on the priorities, when events such as releases and job completions occur
  – Priority scheduling algorithms are event-driven
  – Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed
  – The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely defines a priority scheduling algorithm

• Priority-driven algorithms make locally optimal decisions about which job to run
  – Locally optimal scheduling decisions are often not globally optimal
  – Priority-driven algorithms never intentionally leave any resource idle
    • Leaving a resource idle is not locally optimal
Example: Priority-Driven Scheduling

- Consider the following task:
  - Jobs $J_1, J_2, \ldots, J_8$, where $J_i$ had higher priority than $J_k$ if $i < k$

- Jobs are scheduled on two processors $P_1$ and $P_2$
- Jobs communicate via shared memory, so communication cost is negligible
- The schedulers keep one common priority queue of ready jobs
- All jobs are preemptable; scheduling decisions are made whenever some job becomes ready for execution or a job completes
## Example: Priority-Driven Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>Not yet released</th>
<th>Released but not yet ready to run</th>
<th>Ready to run</th>
<th>P₁</th>
<th>P₂</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>J₁ 0/3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>J₁ 0/1</td>
<td></td>
<td>J₁ 0/2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>J₁ 0/4</td>
<td></td>
<td>J₁ 0/4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Release time**

**Execution time**
Example: Priority-Driven Scheduling

• Note: The ability to preempt lower priority jobs slowed down the overall completion of the task
  – This is not a general rule, but shows that priority scheduling results can be non-intuitive
  – Different priority scheduling algorithms can have very different properties

• Tracing execution of jobs using tables is an effective way to demonstrate correctness for systems with periodic tasks and fixed timing constraints, execution times, resource usage
  – Show that the system enters a repeating pattern of execution, and each hyper-period of that pattern meets all deadlines
  – Proof by exhaustive simulation
    • Provided the system has a manageably small number of jobs
Priority-Driven Scheduling

• Most scheduling algorithms used in non real-time systems are priority-driven
  – First-In-First-Out
  – Last-In-First-Out
  – Shortest-Execution-Time-First
  – Longest-Execution-Time-First

• Real-time priority scheduling assigns priorities based on deadline or some other timing constraint:
  – Earliest deadline first
  – Least slack time first
  – Etc.
Priority Scheduling Based on Deadlines

• Earliest deadline first (EDF)
  – Assign priority to jobs based on deadline
  – Earlier the deadline, higher the priority
  – Simple, just requires knowledge of deadlines

• Least Slack Time first (LST)
  – A job $J_i$ has deadline $d_i$, execution time $e_i$, and was released at time $r_i$
  – At time $t < d_i$:  
    • Remaining execution time $t_{\text{rem}} = e_i - (t - r_i)$
    • Slack time $t_{\text{slack}} = d_i - t - t_{\text{rem}}$
  – Assign priority to jobs based on slack time, $t_{\text{slack}}$
  – The smaller the slack time, the higher the priority
  – More complex, requires knowledge of execution times and deadlines
    • Knowing the actual execution time is often difficult a priori, since it depends on the data, need to use worst case estimates (⇒ poor performance)
Optimality of EDF and LST

• These algorithms are optimal
  – i.e. they will always produce a feasible schedule if one exists
  – Constraints: on a single processor, as long as preemption is allowed and jobs do not contend for resources
Optimality of EDF and LST: Proof

1. Any feasible schedule can be transformed into an EDF schedule
   - If $J_i$ is scheduled to execute before $J_k$, but $J_i$’s deadline is later than $J_k$’s either:
     - The release time of $J_k$ is after the $J_i$ completes $\Rightarrow$ they’re already in EDF order
     - The release time of $J_k$ is before the end of the interval in which $J_i$ executes:
       - Swap $J_i$ and $J_k$ (this is always possible, since $J_i$’s deadline is later than $J_k$’s)
       - Move any jobs following idle periods forward into the idle period

\[
\begin{align*}
&J_i & J_k &  \\
&r_k &  & d_k & d_i \\
\end{align*}
\]

$\Rightarrow$ the result is an EDF schedule

2. So, if EDF fails to produce a feasible schedule, no feasible schedule exists
   - If a feasible schedule existed it could be transformed into an EDF schedule, contradicting the statement that EDF failed to produce a feasible schedule

[Proof for LST is similar]
Non-Optimality of EDF and LST

• Neither algorithm is optimal if jobs are non-preemptable or if there is more than one processor
  – The book has examples which demonstrate EDF and LST producing infeasible schedules in these cases
    • This includes non-strict LST scheduling
    – Proof-by-counterexample

• EDF and LST are simple priority-driven scheduling algorithms; introduced to show how we can reason about such algorithms
  – Lectures 5-8 discuss other priority-driven scheduling algorithms
Dynamic vs. Static Systems

- If jobs are scheduled on multiple processors, and a job can be dispatched from the priority run queue to any of the processors, the system is *dynamic*
- A job *migrates* if it starts execution on one processor and is resumed on a different processor
- If jobs are partitioned into subsystems, and each subsystem is bound statically to a processor, we have a *static* system
- Expect static systems to have inferior performance (in terms of overall response time of the jobs) relative to dynamic systems
  - But it is possible to validate static systems, whereas this is not always true for dynamic systems
  - For this reason, most *hard* real time systems are static
Effective Release Times and Deadlines

• Sometimes the release time of a job may be later than that of its successors, or its deadline may be earlier than that specified for its predecessors

• This makes no sense: derive an effective release time or effective deadline consistent with all precedence constraints, and schedule using that

  – Effective release time
    • If a job has no predecessors, its effective release time is its release time
    • If it has predecessors, its effective release time is the maximum of its release time and the effective release times of its predecessors

  – Effective deadline
    • If a job has no successors, its effective deadline is its deadline
    • If it has successors, its effective deadline is the minimum of its deadline and the effective deadline of its successors
Effective Release Times and Deadlines

• Note: definition of effective deadline does not take into account execution time of successor jobs
  – Would be more accurate, and needs to be done on multiprocessor systems
  – But: unnecessary on single processor with preemptable jobs

  – Feasible to schedule any set of jobs according to their actual release times and deadline, iff feasible to schedule according to effective release times and deadlines
    • Ignore precedence constraints, schedule using effective release times and deadlines as if all jobs independent
    • Resulting schedule might meet deadlines but not precedence constraints
      – If so, always possible to swap order of jobs within the schedule to meet deadlines and precedence constraints
Validating Priority-Driven Scheduling

• Priority-driven scheduling has many advantages over clock-driven scheduling
  – Better suited to applications with varying time and resource requirements, since needs less a priori information
  – Run-time overheads are small

• But not widely used until recently, since difficult to validate
  – Scheduling anomalies can occur for multiprocessor or non-preemptable systems, or those which share resources
    • Reducing the execution time of a job in a task can increase the total response time of the task (see book for example)
    • Not sufficient to show correctness with worse-case execution times, need to simulate with all possible execution times for all jobs comprising a task
  – Can be proved that anomalies do not occur for independent, preemptable, jobs with fixed release times executed using any priority-driven scheduler on a single processor
    • Various stronger results exist for particular priority-driven algorithms
Summary

• Have outlined different approaches to scheduling:
  – Clock-driven
  – Weighted round-robin
  – Priority-driven

and some of their constraints

• Next session will be a tutorial to review the material covered to date, before we move onto detailed discussion of scheduling

• Problem set 1 now available: due at 1:00pm on 25th January