# Clock-Driven Scheduling

Real-Time and Embedded Systems (M)

Lecture 4

UNIVERSITY
*of*
GLASGOW



# Lecture Outline

- Assumptions and notation for clock-driven scheduling
- Handling periodic jobs
  - Static, clock-driven schedules and the cyclic executive
- Handling aperiodic jobs
  - Slack stealing
- Handling sporadic jobs
- Advantages and disadvantages of clock driven scheduling

Material corresponds to chapter 5 of Liu's book

Copyright © 2006 University of Glasgow
All rights reserved

# Assumptions

- Clock-driven scheduling applicable to deterministic systems
- A restricted periodic task model:
  - The parameters of all periodic tasks are known a priori
  - For each mode of operation, system has a fixed number, $n$, periodic tasks
    - For task $T_i$ each job $J_{i,k}$ is ready for execution at its release time $r_{i,k}$ and is released $p_i$ units of time after the previous job in $T_i$ such that $r_{i,k} = r_{i,k-1} + p_i$
    - Variations in the inter-release times of jobs in a periodic task are negligible
  - Aperiodic jobs may exist
    - Assume that the system maintains a single queue for aperiodic jobs
    - Whenever the processor is available for aperiodic jobs, the job at the head of this queue is executed
  - There are no sporadic jobs
    - Recall: sporadic jobs have hard deadlines, aperiodic jobs do not

Copyright © 2006 University of Glasgow
All rights reserved.

# Notation

- The 4-tuple $T_i = (\phi_i, p_i, e_i, D_i)$ refers to a periodic task $T_i$ with phase $\phi_i$, period $p_i$, execution time $e_i$, and relative deadline $D_i$
  - Default phase of $T_i$ is $\phi_i = 0$, default relative deadline is the period $D_i = p_i$
    Omit elements of the tuple that have default values
  - Examples:

$T_1 = (1, 10, 3, 6) \Rightarrow \quad \phi_1 = 1 \quad p_1 = 10 \quad e_1 = 3 \quad D_1 = 6$

$J_{1,1}$ released at 1, deadline 7
$J_{1,2}$ released at 11, deadline 17
…

$T_2 = (10, 3, 6) \quad \Rightarrow \quad \phi_2 = 0 \quad p_2 = 10 \quad e_2 = 3 \quad D_2 = 6$

$J_{1,1}$ released at 0, deadline 6
$J_{1,2}$ released at 10, deadline 16
…

$T_3 = (10, 3) \quad \Rightarrow \quad \phi_3 = 0 \quad p_3 = 10 \quad e_3 = 3 \quad D_3 = 10$

$J_{1,1}$ released at 0, deadline 10
$J_{1,2}$ released at 10, deadline 20
…

Copyright © 2006 University of Glasgow
All rights reserved.

# Static, Clock-driven Cyclic Scheduler

- Since the parameters of all jobs with hard deadlines are known can construct a static *cyclic schedule* in advance
  - Processor time allocated to a job equals its maximum execution time
  - Scheduler dispatches jobs according to the static schedule, repeating each hyperperiod
  - Static schedule guarantees that each job completes by its deadline
    - No job overruns ⇒ all deadlines are met

- Schedule calculated off-line ⇒ can use complex algorithms
  - Run-time of the scheduling algorithm irrelevant
  - Can search for a schedule that optimizes some characteristic of the system
    - e.g. a schedule where the idle periods are nearly periodic; accommodating aperiodic jobs
  - The book gives an example, but scheduling algorithm unimportant as long as resulting schedule meets deadline

Copyright © 2006 University of Glasgow
All rights reserved.

# Example Cyclic Schedule

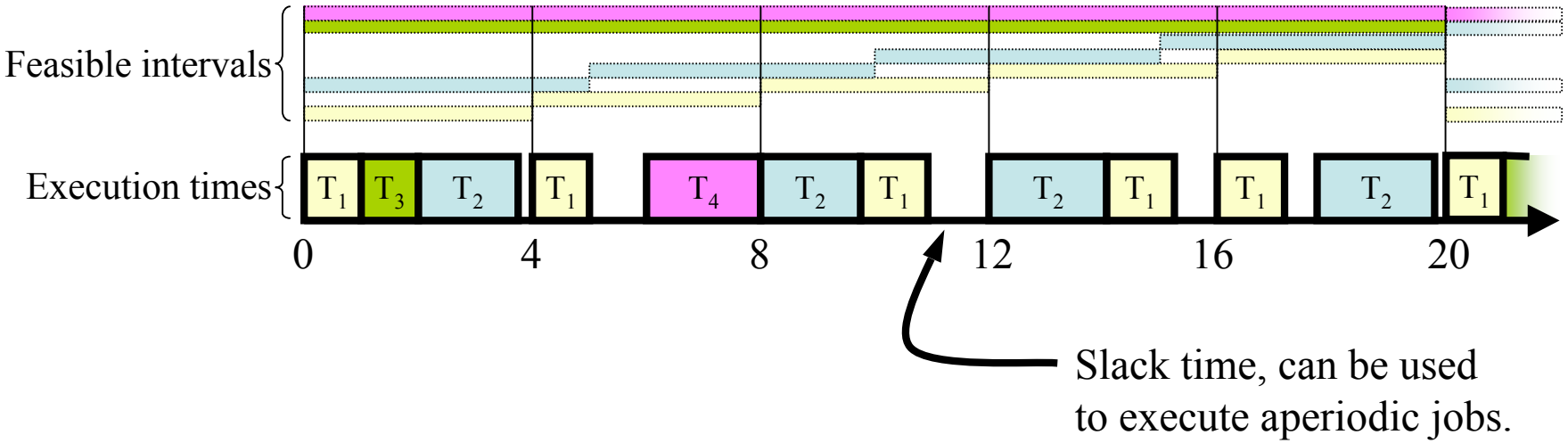
- Consider a system with 4 independent periodic tasks:
  - $T_1 = (4, 1.0)$
  - $T_2 = (5, 1.8)$
  - $T_3 = (20, 1.0)$
  - $T_4 = (20, 2.0)$

Execution time

Period

[Phase and deadline take default values]

- Hyper-period $H = 20$ (least common multiple of 4, 5, 20, 20)
- Can construct an arbitrary static schedule to meet all deadlines:




Copyright © 2006 University of Glasgow
All rights reserved

# Implementing a Cyclic Scheduler

- Store pre-computed schedule as a table
- The system creates all the tasks that are to be executed:
  - Allocates sufficient memory for the code and data of every task
  - Brings the code executed by the task into memory
- Scheduler sets the hardware timer to interrupt at the first decision time, $t_{k=0}$
- On receipt of an interrupt at $t_k$:
  - Scheduler sets the timer interrupt to expire at $t_{k+1}$
  - If previous task overrunning, handle failure
  - If $T(t_k) = I$ and aperiodic job waiting, start aperiodic job
  - Otherwise, start next job in task $T(t_k)$ executing

| $k$ | $t_k$ | $T(t_k)$ |
|---|---|---|
| 0 | 0.0 | $T_1$ |
| 1 | 1.0 | $T_3$ |
| 2 | 2.0 | $T_2$ |
| 3 | 3.8 | $I$ |
| 4 | 4.0 | $T_1$ |
| 5 | 5.0 | $I$ |
| 6 | 6.0 | $T_4$ |
| 7 | 8.0 | $T_2$ |
| 8 | 9.8 | $T_1$ |
| 9 | 10.8 | $I$ |
| 10 | 12.0 | $T_2$ |
| 11 | 13.8 | $T_1$ |
| 12 | 14.8 | $I$ |
| 13 | 17.0 | $T_1$ |
| 14 | 17.0 | $I$ |
| 15 | 18.0 | $T_2$ |
| 16 | 19.8 | $I$ |

Copyright © 2006 University of Glasgow
All rights reserved

# Implementing a Cyclic Scheduler

Input: stored schedule $(t_k, T(t_k))$ for $k = 0, 1, n - 1$.
Task SCHEDULER:

```
        set the next decision point i = 0 and table entry k = 0;
        set the timer to expire at t_k;
        do forever:
                accept timer interrupt;
                if an aperiodic job is executing, preempt the job;
                current task T = T(t_k);
                increment i by 1;
                compute the next table entry k = i mod n;
                set the timer to expire at [i / n] * H + t_k;
                if the current task T is I,
                        let the job at the head of the aperiodic queue execute;
                else
                        let the task T execute;
                sleep;
        end do.
End SCHEDULER.
```

Copyright © 2006 University of Glasgow
All rights reserved.

# Structured Cyclic Schedules

- Arbitrary table-driven cyclic schedules flexible, but inefficient
  - Relies on accurate timer interrupts, based on execution times of tasks
  - High scheduling overhead
- Easier to implement if structure imposed:
  - Make scheduling decisions at periodic intervals (*frames*) of length $f$
  - Execute a fixed list of jobs with each frame, disallowing pre-emption except at frame boundaries
  - Require phase of each periodic task to be a non-negative integer multiple of the frame size
    - The first job of every task is released at the beginning of a frame
    - $\phi = k \cdot f$ where $k$ is a non-negative integer
- Gives two benefits:
  - Scheduler can easily check for overruns and missed deadlines at the end of each frame
  - Can use a periodic clock interrupt, rather than programmable timer

Copyright © 2006 University of Glasgow
All rights reserved.

# Frame Size Constraints

- How to choose frame length?
  - To avoid preemption, want jobs to start and complete execution within a single frame:

$$f \geq \max(e_1, e_2, \ldots, e_n) \qquad \text{(Eq.1)}$$

  - To minimize the number of entries in the cyclic schedule, the hyper-period should be an integer multiple of the frame size ($\Rightarrow f$ divides evenly into the period of at least one task):

$$\exists\, i : \text{mod}(p_i,\ f) = 0 \qquad \text{(Eq.2)}$$

  - To allow scheduler to check that jobs complete by their deadline, should be at least one frame boundary between release time of a job and its deadline:

$$2*f - \gcd(p_i, f) \leq D_i \text{ for } i = 1, 2, \ldots, n \qquad \text{(Eq.3)}$$

    (see section 5.3.2 of book for derivation of Eq.3)

- All 3 constraints should be satisfied

Copyright © 2006 University of Glasgow
All rights reserved.

# Frame Size Constraints – Example

- Review the system of independent periodic tasks from our earlier example:

  $T_1 = (4, 1.0)$ $\quad T_2 = (5, 1.8)$

  $T_3 = (20, 1.0)$ $\quad T_4 = (20, 2.0)$

Hyper-period $H$ = lcm(4, 5, 20, 20) = 20

- Constraints:

  Eq.1 $\Rightarrow f \geq \max(1, 1.8, 1, 2) \geq 2$

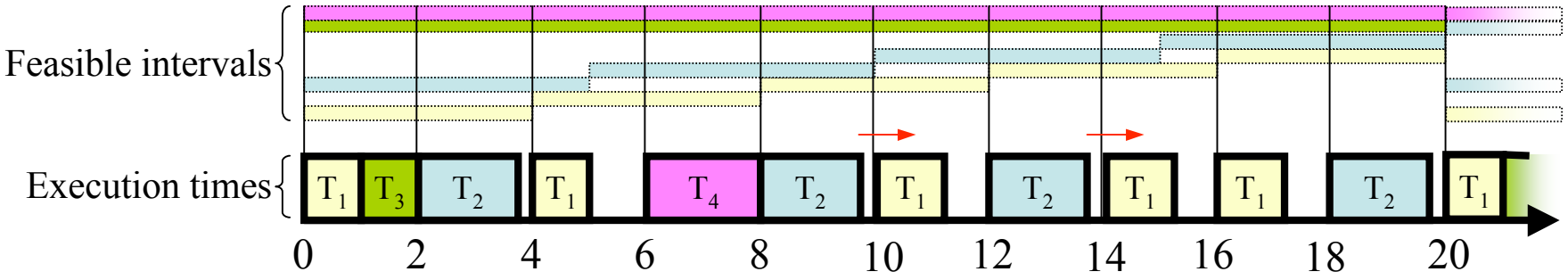
  Eq.2 $\Rightarrow f \in \{ 2, 4, 5, 10, 20 \}$

  Eq.3 $\Rightarrow 2f - \gcd(4, f) \leq 4 \quad (T_1)$

  $2f - \gcd(5, f) \leq 5 \quad (T_2)$

  $2f - \gcd(20, f) \leq 20 \quad (T_3, T_4)$

The values that satisfie all constraints are $f = 2$ or 4




Copyright © 2006 University of Glasgow
All rights reserved.

# Job Slices

- Sometimes, a system cannot meet all three frame size constraints simultaneously
- Can often solve by partitioning a job with large execution time into slices (sub-jobs) with shorter execution times/deadlines
  - Gives the effect of preempting the large job, so allow other jobs to run
  - Sometimes need to partition jobs into more slices than required by the frame size constraints, to yield a feasible schedule

- Example:
  - Consider a system with $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$, $T_3 = (20, 5)$
  - Cannot satisfy constraints: Eq.1 $\Rightarrow f \geq 5$ but Eq.3 $\Rightarrow f \leq 4$
  - Solve by splitting $T_3$ into $T_{3,1} = (20, 1)$, $T_{3,2} = (20, 3)$ and $T_{3,3} = (20,1)$
    - Other possible splits exist; pick based on application domain knowledge
  - Result can be scheduled with $f = 4$

Copyright © 2006 University of Glasgow
All rights reserved

# Building a Structured Cyclic Schedule

- To construct a cyclic schedule, we need to make three kinds of design decisions:
  - Choose a frame size based on constraints
  - Partition jobs into slices
  - Place slices in frames
- These decisions cannot be taken independently:
  - Ideally want as few slices as possible, but may be forced to use more to get a feasible schedule

Copyright © 2006 University of Glasgow
All rights reserved.

# Implementation: A Cyclic Executive

- Modify previous table-driven cyclic scheduler to be frame based, schedule all types of jobs in a multi-threaded system
- Table that drives the scheduler has $F$ entries, where $F = \frac{H}{f}$
  - Each corresponding entry $L(k)$ lists the names of the job slices that are scheduled to execute in frame $k$; called a scheduling block
  - Each job slice implemented by a procedure, to be called in turn
- Cyclic executive executed by the clock interrupt that signals the start of a frame:
  - Determines the appropriate scheduling block for this frame
  - Executes the jobs in the scheduling block in order
  - Starts job at head of the aperiodic job queue running for remainder of frame
- Less overhead than pure table driven cyclic scheduler, since only interrupted on frame boundaries, rather than on each job

Copyright © 2006 University of Glasgow
All rights reserved

# Scheduling Aperiodic Jobs

- Thus far, aperiodic jobs are scheduled in the background after all jobs with hard deadlines scheduled in each frame have completed
  - Delays execution of aperiodic jobs in preference to periodic jobs
  - However, note that there is often no advantage to completing a hard real-time job early, and since an aperiodic job is released due to an event, the sooner such a job completes, the more responsive the system
- Hence, minimizing response times for aperiodic jobs is typically a design goal of real-time schedulers

Copyright © 2006 University of Glasgow
All rights reserved.

# Scheduling Aperiodic Jobs: Slack Stealing

- Periodic jobs scheduled in frames that end before their deadline; there may be some *slack time* in the frame after the periodic job completes
- Since we know the execution time of periodic jobs, can move the slack time to the start of the frame, running the periodic jobs just in time to meet their deadline
- Execute aperiodic jobs in the slack time, ahead of periodic jobs
  - The cyclic executive keeps track of the slack left in each frame as the aperiodic jobs execute, preempts them to start the periodic jobs when there is no more slack
  - As long as there is slack remaining in a frame, the cyclic executive returns to examine the aperiodic job queue after each slice completes
- Reduces response time for aperiodic jobs, but requires accurate timers

Copyright © 2006 University of Glasgow
All rights reserved

# Scheduling Sporadic Jobs

- We assumed there were no sporadic jobs – what if this is relaxed?
- Sporadic jobs have hard deadlines, release and execution times that are not known a priori
  - Hence, a clock-driven scheduler cannot guarantee a priori that sporadic jobs complete in time
- However, scheduler can determine if a sporadic job is schedulable when it arrives
  - Perform an *acceptance test* to check whether the newly released sporadic job can be feasibly scheduled with all the jobs in the system at that time
  - If there is sufficient slack time in the frames before the new job's deadline, the new sporadic job is accepted; otherwise, it is rejected
    - Can be determined that a new sporadic job cannot be handled as soon as that job is released; earliest possible rejection
  - If more than one sporadic job arrives at once, they should be queued for acceptance in EDF order

Copyright © 2006 University of Glasgow
All rights reserved.

# Practical Considerations

- Handling overruns:
  - Jobs are scheduled based on maximum execution time, but failures might cause overrun
  - A robust system will handle this by either: 1) killing the job and starting an error recovery task; or 2) preempting the job and scheduling the remainder as an aperiodic job
    - Depends on usefulness of late results, dependencies between jobs, etc.
- Mode changes:
  - A cyclic scheduler needs to know all parameters of real-time jobs a priori
  - Switching between modes of operation implies reconfiguring the scheduler and bringing in the code/data for the new jobs
  - This can take a long time: schedule the reconfiguration job as an aperiodic or sporadic task to ensure other deadlines met during mode change
- Multiple processors:
  - Can be handled, but off-line scheduling table generation more complex

Copyright © 2006 University of Glasgow
All rights reserved.

# Clock-driven Scheduling: Advantages

- Conceptual simplicity
  - Ability to consider complex dependencies, communication delays, and resource contention among jobs when constructing the static schedule, guaranteeing absence of deadlocks and unpredictable delays
  - Entire schedule is captured in a static table
  - Different operating modes can be represented by different tables
  - No concurrency control or synchronization required
  - If completion time jitter requirements exist, can be captured in the schedule
- When workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary
- Choice of frame size can minimize context switching and communication overheads
- Relatively easy to validate, test and certify

Copyright © 2006 University of Glasgow
All rights reserved

# Clock-driven Scheduling: Disadvantages

- Inflexible
  - Pre-compilation of knowledge into scheduling tables means that if anything changes materially, have to redo the table generation
  - Best suited for systems which are rarely modified once built
- Other disadvantages:
  - Release times of all jobs must be fixed
  - All possible combinations of periodic tasks that can execute at the same time must be known a priori, so that the combined schedule can be pre-computed
  - The treatment of aperiodic jobs is very primitive
    - Unlikely to yield acceptable response times if a significant amount of soft real-time computation exists

Copyright © 2006 University of Glasgow
All rights reserved.

# Summary

- We have discussed:
  - Static, clock-driven schedules and the cyclic executive
  - Handling aperiodic jobs
    - Slack stealing
  - Handling sporadic jobs
  - Advantages and disadvantages of clock driven scheduling
    - Clock-driven scheduling applicable to static systems, small number of aperiodic jobs

- The next lecture begins our study of priority scheduling for more dynamic environments

Copyright © 2006 University of Glasgow
All rights reserved.