University of Glasgow | School of Computing Science

Clock-driven Real-time Scheduling

Advanced Operating Systems (M)
Lecture 3

Lecture Outline

- Assumptions and notation
- Handling periodic jobs: static schedules and cyclic executives
- Handling aperiodic jobs: slack stealing
- Handling sporadic jobs
- Advantages and disadvantages

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 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 minimised
  - Simple and straight-forward, not flexible

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 each task T_i, 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 such that $r_{i,k} = r_{i,k-1} + p_i$
  - Aperiodic jobs may exist; assume that the system maintains a single queue for aperiodic jobs, and that the job at the head of this queue executes whenever the processor is available for aperiodic jobs
  - There are no sporadic jobs

Notation

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

$T_1 = (1, 10, 3, 6) \Rightarrow \varphi_1 = 1 \quad p_1 = 10 \quad e_1 = 3 \quad D_1 = 6$

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

$T_2 = (10, 3, 6) \Rightarrow \varphi_2 = 0 \quad p_2 = 10 \quad e_2 = 3 \quad D_2 = 6$

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

$T_3 = (10, 3) \Rightarrow \varphi_3 = 0 \quad p_3 = 10 \quad e_3 = 3 \quad D_3 = 10$

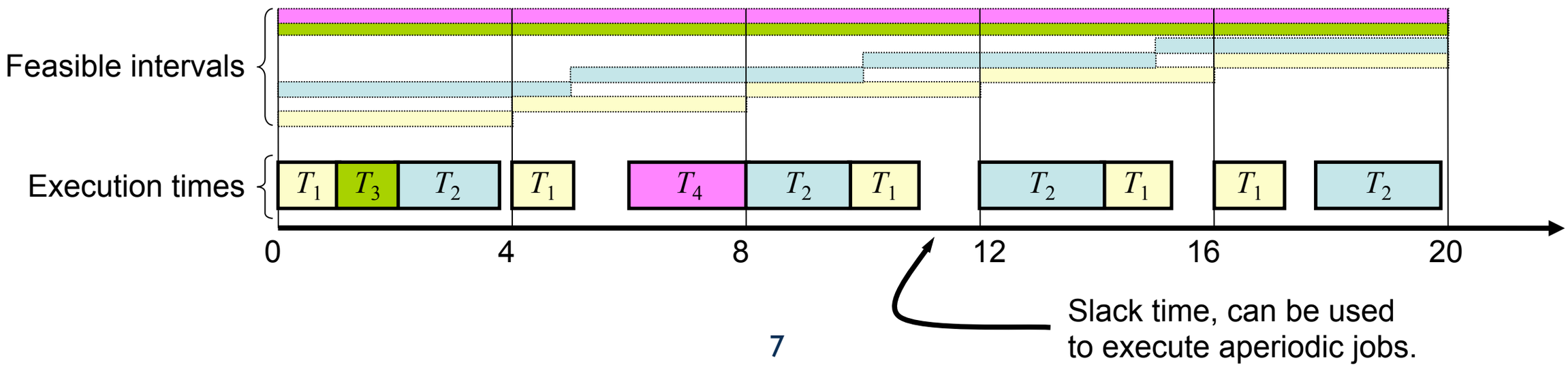
$J_{3,1}$ released at 0, deadline 10
$J_{3,2}$ released at 10, deadline 20
…

Static, Clock-driven Cyclic Scheduler

- Parameters of all jobs known in advance, so can construct a static *cyclic schedule*
  - Processor time allocated to a job equals its maximum execution time
  - Scheduler dispatches jobs according to the static schedule, repeating each hyper-period
  - Static schedule guarantees that each job completes by its deadline; no overruns ⇒ all deadlines are met
- Schedule calculated off-line, so can use complex algorithms
  - Run-time of the scheduling algorithm irrelevant
  - Can search for a schedule that optimises some characteristic of the system, e.g., a schedule where the idle periods are nearly periodic; accommodating aperiodic jobs

Example Cyclic Schedule

- Consider a system of 4 independent periodic tasks:
  - $T_1 = (\ 4, 1.0)$
  - $T_2 = (\ 5, 1.8)$ [Phase and deadline take default values]
  - $T_3 = (20, 1.0)$
  - $T_4 = (20, 2.0)$
  - Hyper-period $H = 20$ (least common multiple of 4, 5, 20, 20)
- Can construct an arbitrary static schedule to meet all deadlines:



Implementing a Cyclic Scheduler

- Store pre-computed schedule as table
  - System creates all the tasks that are to be executed, allocates sufficient memory, and ensures resources are available
- Scheduler sets hardware 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

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.
```

Structured Cyclic Schedules

- Arbitrary table-driven schedules flexible; inefficient
  - Relies on accurate timer interrupts, to has high timer 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; first job of every task is released at the start of a frame
- 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

Frame Size Constraints

- How to choose frame length? 3 constraints:
  - 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 minimise 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):

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

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

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

Frame Size Constraints – Example

- Review the system of independent periodic tasks from our earlier example:
  - $T_1 = (4, 1.0)$ $T_2 = (5, 1.8)$
  - $T_3 = (20, 1.0)$ $T_4 = (20, 2.0)$

Hyper-period $H = \text{lcm}(4, 5, 20, 20) = 20$

- 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 $\Rightarrow$ $2f - \text{gcd}(4, f) \leq 4$ (T_1)
    $2f - \text{gcd}(5, f) \leq 5$ (T_2)
    $2f - \text{gcd}(20, f) \leq 20$ (T_3, T_4)
- Possible solutions are $f = 2$ or $f = 4$

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:
  - $T_1 = (4, 1), T_2 = (5, 2, 7), T_3 = (20, 5)$ fails since $\text{Eq.1} \Rightarrow f \geq 5$ but $\text{Eq.3} \Rightarrow f \leq 4$
  - Solve by splitting T_3 into $T_{3,1} = (20, 1)$, $T_{3,2} = (20, 3)$ and $T_{3,3} = (20,1)$ so can be scheduled with $f = 4$
    (other splits exist; pick based on application domain knowledge)

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

Implementation: A Cyclic Executive

- Modify table-driven scheduler to be frame based, with F entries, where $F = H/f$
  - Each corresponding entry $L(k)$ lists the names of the job slices 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 and executes the jobs in order
- Less overhead than pure table driven scheduler, since only interrupted on frame boundaries

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, minimising response times for aperiodic jobs is typically a design goal of real-time schedulers

Slack Stealing for Aperiodic Jobs

- Periodic jobs scheduled in frames that end before their deadline; there may be some *slack time* in the frame after the periodic job completes
- Move the slack time to the start of the frame, run periodic jobs just in time to meet deadlines, and aperiodic jobs in slack time ahead of periodic jobs
  - Scheduler keeps track of slack in each frame as 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

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 in advance, so a static clock-driven schedule cannot guarantee that they meet their deadlines
- However, scheduler can determine if a sporadic job can be scheduled *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
  - If more than one sporadic job arrives at once, they should be queued for acceptance in EDF order

Practical Considerations

- Handling overruns:
  - Jobs are scheduled based on maximum execution time, but failures might cause overrun
  - Can be handled by either: 1) kill the job and recover from error; or 2) preempt the job and schedule remainder as an aperiodic job. Choice depends on usefulness of late results, dependencies between jobs, etc.
- Mode changes:
  - Switching between modes of operation implies reconfiguring scheduler, and bringing in the code/data for the new jobs
  - 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

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
- Efficient
  - No concurrency control or synchronisation required
  - Choose frame size to minimise context switching overheads
- Relatively easy to validate, test and certify
  - When workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary

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 primitive and unlikely to yield acceptable response times if a significant amount of soft real-time computation exists

Summary

- We have discussed:
  - Static, clock-driven schedules and the cyclic executive
  - Handling aperiodic jobs via slack stealing
  - Handling sporadic jobs
  - Advantages and disadvantages of clock driven scheduling

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