CSCE 990: *Real-Time Systems*

Clock-Driven Scheduling

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/RealTimeSystems

Clock-driven (or Static) Scheduling

(Baker and Shaw and Chapter 5 of Liu)

- Model assumed in this chapter:
  - » n periodic tasks $T_1,\ldots,T_n$.
  - » The "rest of the world" periodic model is assumed.
  - » T_i is specified by (ϕ_i, p_i, e_i, D_i), where
    - ϕ_i is its phase,
    - p_i is its period,
    - e_i is its execution cost per job, and
    - D_i is its relative deadline.
    - Will abbreviate as (p_i,e_i,D_i) if ϕ_i=0, and (p_i,e_i) if $\phi_i=0 \wedge p_i=D_i$.
  - » We also have aperiodic jobs that are released at arbitrary times (later, we'll consider sporadic jobs too).

Schedule Table

- Our scheduler will schedule periodic jobs using a **static schedule** that is computed offline and stored in a table T.
  - » $T(t_k) = \begin{cases} T_i & \text{if } T_i \text{ is to be scheduled at time } t_k \\ I & \text{if no periodic task is scheduled at time } t_k \end{cases}$
  - » For most of this chapter, we assume the table is given.
  - » Later, we consider one algorithm for producing the table.
    - **Note:** This algorithm need not be highly efficient.
  - » We will schedule aperiodic jobs (if any are ready) in intervals not used by periodic jobs.

Static, Timer-driven Scheduling

```
/* H is the hyperperiod.  There are N "quanta" per hyperperiod */
Input: Stored schedule (t_k, T(t_k)) for k = 0, 1, …, N – 1

Task SCHEDULER:
   set the next decision point i and table entry k to 0;
   set the timer to expire at t_k;
   do forever
      accept timer interrupt;
      if an aperiodic job is executing, preempt it;
      current task T = T(t_k);
      i := i + 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 then
         let the job at the head of the aperiodic job queue execute;
      else
         let task T execute
      fi
      sleep
end SCHEDULER
```

These quanta aren't necessarily uniform.

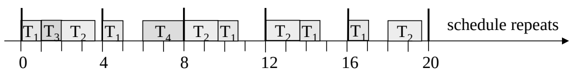
We call a schedule produced by this scheduler a **cyclic schedule**.

Although Liu doesn't say this explicitly, the assumption here seems to be that T finishes before the next interrupt.

Example

Consider a system of four tasks, $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ $T_4 = (20, 2)$.

Consider the following static schedule:



The first few table entries would be: $(0, T_1)$, $(1, T_3)$, $(2, T_2)$, $(3.8, I)$, $(4, T_1)$, …

Frames

- Let us refine this notion of scheduling…
- To keep the table small, we divide the time line into **frames** and make scheduling decisions only at frame boundaries.
  - » Each job is executed as a procedure call that must fit within a frame.
  - » Multiple jobs may be executed in a frame, but the table is only examined at frame boundaries (the number of "columns" in the table = the number of frames per hyperperiod).
  - » In addition to making scheduling decisions, the scheduler also checks for various error conditions, like task overruns, at the beginning of each frame.
- We let **f** denote the **frame size**.

Frame Size Constraints

We want frames to be sufficiently long so that every job can execute within a frame nonpreemptively. So,

$$\boxed{f \geq \max_{1 \leq i \leq n}(e_i).}$$

To keep table small, f should divide H. Thus, for at least one task T_i,

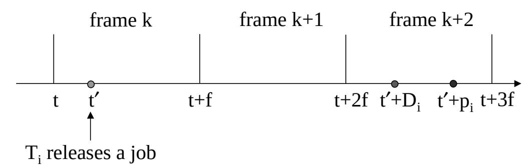
$$\boxed{\lfloor p_i/f \rfloor - p_i/f = 0.}$$

Let F = H/f. (Note: F is an integer.) Each interval of length H is called a **major cycle**. Each interval of length f is called a **minor cycle**. There are F minor cycles per major cycle.

Frame Constraints (Continued)

We want the frame size to be sufficiently small so that between the release time and deadline of every job, there is at least one frame.

- A job released "inside" a frame is not noticed by the scheduler until the next frame boundary.
- Moreover, if a job has a deadline "inside" frame k + 1, it essentially must complete execution by the end of frame k.



Thus, $\boxed{2f - \gcd(p_i, f) \leq D_i.}$

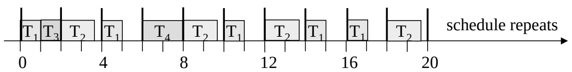
Example

Consider a system of four tasks, $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ $T_4 = (20, 2)$.

By first constraint, $f \geq 2$.

Hyperperiod is 20, so by second constraint, possible choices for f are 2, 4, 5, 10, and 20.

Only f = 2 satisfies the third constraint. The following is a possible cyclic schedule.

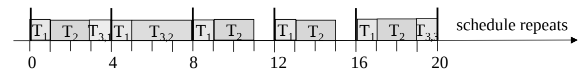


Job Slices

What do we do if the frame size constraints cannot be met?

Example: Consider T = {(4, 1), (5, 2, 7), (20, 5)}. By first constraint, $f \geq 5$, but by third constraint, $f \leq 4$!

Solution: "Slice" the task (20, 5) into subtasks, (20, 1), (20, 3), and (20, 1). Then, f = 4 works. Here's a schedule:



Summary of Design Decisions

- Three design decisions:
  - » choosing a frame size,
  - » partitioning jobs into slices, and
  - » placing slices in frames.
- In general, these decisions cannot be made independently.
- We will look at an algorithm for making these decisions later.

Pseudo-code for Cyclic Executive

```
Input: Stored schedule: L(k) for k = 0, 1, …, F – 1;
       Aperiodic job queue

Task CYCLIC_EXECUTIVE:
    current time t := 0;  current frame k := 0;
    do forever
        accept clock interrupt at time t·f;
        currentBlock := L(k);  t := t + 1;  k := t mod F;
        if the last job is not completed, take appropriate action;
        if any of the slices in currentBlock is not released, take appropriate action;
        wake up the periodic task server to execute the slices in currentBlock;
        sleep until the periodic task server completes;
        while the aperiodic job queue is nonempty do
            wake up the job at the head of the aperiodic job queue;
            sleep until the aperiodic job completes;
            remove the aperiodic job from queue;
        od;
        sleep until the next clock interrupt;
    od
end CYCLIC_EXECUTIVE
```

I'm not really sure why this check is needed — each slice is just a procedure call.

The periodic task server simply executes each slice in currentBlock as a procedure call.

Improving Response Times of Aperiodic Jobs

- Intuitively, it makes sense to give hard real-time jobs higher priority than aperiodic jobs.
- However, this may lengthen the response time of an aperiodic job.

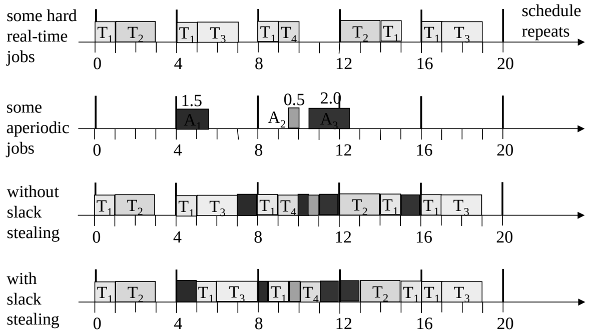
hard deadline is still met
but aperiodic job completes sooner

- Note that **there is no point in completing a hard real-time job early, as long as it finishes by its deadline.**

Slack Stealing

- Let the total amount of time allocated to all the slices scheduled in frame k be x_k.
- **Definition:** The **slack** available at the beginning of frame k is $f - x_k$.
- **Change to scheduler:**
  - » If the aperiodic job queue is nonempty, let aperiodic jobs execute in each frame whenever there is nonzero slack.

Example



Implementing Slack Stealing

- Use a pre-computed "initial slack" table.
  - » Initial slack depends only on static quantities.
- Use an **interval timer** to keep track of available slack.
  - » Set the timer when an aperiodic job begins to run. If it goes off, must start executing periodic jobs.
  - » **Problem:** Most OSs do not provide sub-millisecond granularity interval timers (as we shall see).
  - » So, to use slack stealing, temporal parameters must be on the order of 100s of msecs. or secs.

Scheduling Sporadic Jobs

- Sporadic jobs arrive at arbitrary times.
- They have hard deadlines.
- Implies we cannot hope to schedule every sporadic job.
- When a sporadic job arrives, the scheduler performs an **acceptance test** to see if the job can be completed by its deadline.
- We must ensure that a new sporadic job does not cause a previously-accepted sporadic job to miss its deadline.
- We assume sporadic jobs are prioritized on an earliest-deadline-first (EDF) basis.

Acceptance Test

Let $\sigma(i, k)$ be the initial total slack in frames i through k, where $1 \leq i \leq k \leq F$. (This quantity only depends on periodic jobs.)

Suppose we are doing an acceptance test at frame t for a newly-arrived sporadic job S with deadline d and execution cost e.

Suppose d occurs within frame $\ell + 1$, i.e., S must complete by the end of frame ℓ.

Compute the *current* total slack in frames t through ℓ using

$$\sigma_c(t, \ell) = \sigma(t, \ell) - \sum_{d_k \leq d}(e_k - \xi_k)$$

The sum is over previously-accepted sporadic jobs with equal or earlier deadlines. ξ_k is the amount of time already spent executing S_k before frame t.

Acceptance Test (Continued)

We'll specify the rest of the test "algorithmically"…

```
if σc(t, ℓ) < e then reject S
else
    record σ := σc(t, ℓ) – e as S's slack;
    for each previously-accepted sporadic task Sk with a deadline after d do
        let σk denote the slack recorded for Sk;
        if σk – e < 0 then reject S fi /* or else Sk will miss its deadline */
    od
fi;
if didn't reject S then accept it fi
```

To summarize, the scheduler must maintain the following data:

- pre-computed initial slack table $\sigma(i, k)$;
- ξ_k values to use at the beginning of the current frame t;
- the current slack σ_k of every accepted sporadic job S_k.

Executing Sporadic Tasks

- Accepted sporadic jobs are executed like aperiodic jobs in the original alg. (without slack stealing).
  - » Remember, when meeting a deadline is the main concern, there is no need to complete a job early.
  - » **One difference:** The aperiodic job queue is in FIFO order, while the sporadic job queue is in EDF order.
- Aperiodic jobs only execute when the sporadic job queue is empty.
  - » As before, slack stealing could be used when executing aperiodic jobs (in which case, some aperiodic jobs could execute when the sporadic job queue is not empty).

Practical Considerations

- **Handling frame overruns.**
  - » **Main Issue:** Should offending job be completed or aborted?
- **Mode changes.**
  - » During a mode change, the running set of tasks is replaced by a new set of tasks (i.e., the table is changed).
  - » Can implement mode change by having an **aperiodic or sporadic mode-change job**. (If sporadic, what if it fails the acceptance test???)
- **Multiprocessors.**
  - » Like uniprocessors, but table probably takes longer to pre-compute.

Network Flow Algorithm for Computing Static Schedules

Initialization: Compute all frame sizes in accordance with the second two frame-size constraints:

$\lfloor p_i/f \rfloor - p_i/f = 0$ $\qquad 2f - \gcd(p_i, f) \le D_i$

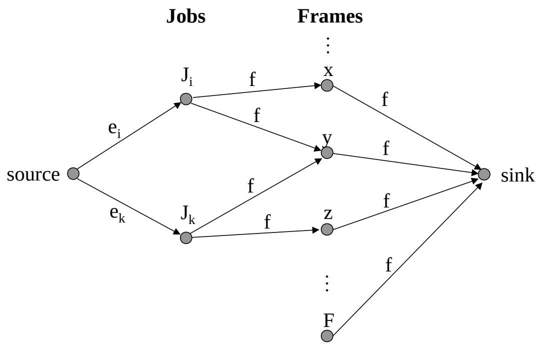
At this point, we ignore the first constraint, $f \ge \max_{1 \le i \le n}(e_i)$. Recall this is the constraint that can force us to "slice" a task into subtasks.

Iterative Algorithm: For each possible frame size f, we compute a network flow graph and run a max-flow algorithm. If the flow thus found has a certain value, then we have a schedule.

Flow Graph

- Denote all jobs in the major cycle of F frames as $J_1, J_2, \ldots, J_N$.
- **Vertices:**
  - » N *job vertices*, denoted $J_1, J_2, \ldots, J_N$.
  - » F *frame vertices*, denoted 1, 2, …, F.
  - » *source* and *sink*.
- **Edges:**
  - » (J_i, j) with capacity f iff J_i can be scheduled in frame j.
  - » (source, J_i) with capacity e_i.
  - » (f, sink) with capacity f.

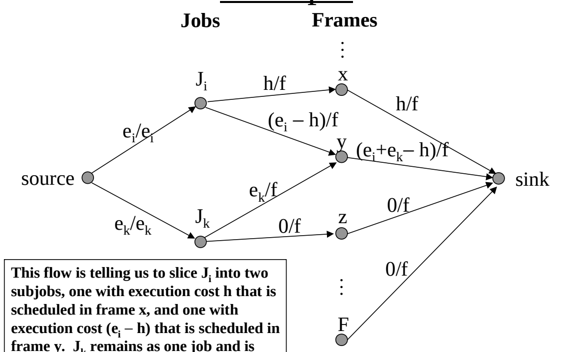
Illustration of Flow Graph



Finding a Schedule

- The maximum attainable flow value is clearly $\sum_{i=1,\ldots,N} e_i$. This corresponds to the exact amount of computation to be scheduled in the major cycle.
- If a max flow is found with value $\sum_{i=1,\ldots,N} e_i$, then we have a schedule.
- If a job is scheduled across multiple frames, then we must slice it into corresponding subjobs.

Example



This flow is telling us to slice J_i into two subjobs, one with execution cost h that is scheduled in frame x, and one with execution cost $(e_i – h)$ that is scheduled in frame y. J_k remains as one job and is scheduled in frame y.

Non-independent Tasks

- Tasks with **precedence constraints** are no problem.
  - » We can enforce precedence constraint like "J_i precedes J_k" by simply making sure J_i's release is at or before J_k's release, and J_i's deadline is at or before J_k's deadline.
  - » If slices of J_i and J_k are scheduled in the wrong order, we can just swap them.
- **Critical sections** pose a greater challenge.
  - » We can try to "massage" the flow-network schedule into one where nonpreemption constraints are respected.
  - » Unfortunately, there is no known efficient, optimal algorithm for doing this (the problem is actually NP-hard).

Pros and Cons of Cyclic Executives

- **Main Advantage:** CEs are *very* simple — you just need a table.
  - » For example, additional mechanisms for concurrency control and synchronization are not needed. In fact, there's really no notion of a "process" here — just procedure calls.
  - » Can validate, test, and certify with very high confidence.
  - » Certain anomalies will not occur.
  - » For these reasons, cyclic executives are the predominant approach in many safety-critical applications (like airplanes).

Aside: Scheduling Anomalies

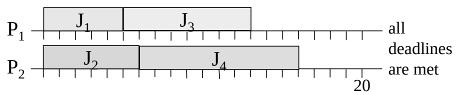
(Section 4.8.1 of Liu)

Here's an example: On a multiprocessor, decreasing a job's execution cost can *increase* some job's response time.

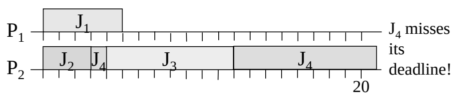
Example: Suppose we have one job queue, preemption, but no migration.

	r_i	d_i	e_i
J_1	0	10	5
J_2	0	10	6
J_3	4	15	8
J_4	0	20	10

Priority order:
J_1 J_2 J_3 J_4



Now, decrease e_2 to 3 …



Pros and Cons (Continued)

- **Disadvantages of cyclic executives:**
  - » **Very brittle:** Any change, no matter how trivial, requires that a new table be computed!
  - » Release times of all jobs must be fixed, i.e., "real-world" sporadic tasks are difficult to support.
  - » Temporal parameters essentially must be multiples of f.
  - » F could be huge!
  - » All combinations of periodic tasks that may execute together must *a priori* be analyzed.
  - » From a software engineering standpoint, "slicing" one procedure into several could be error-prone.