Today

- Intro to real-time scheduling
- Cyclic executives
  - Scheduling tables
  - Frames
  - Frame size constraints
  - Generating schedules
  - Non-independent tasks
  - Pros and cons
Real-Time Systems

- The correctness of a *real-time system* depends not just on the validity of results but on the times at which results are computed
  - Computations have *deadlines*
  - Usually, but not always, ok to finish computation early
- *Hard real-time system*: missed deadlines may be catastrophic
- *Soft real-time system*: missed deadlines reduce the value of the system
- Real-time deadlines are usually in the range of microseconds through seconds
Real-Time System Examples

◆ Hard real-time
  ➢ Most feedback control systems
    • E.g. engine control, avionics, ...
    • Missing deadlines affects stability of control
  ➢ Air traffic control
    • Missing deadlines affects ability of airplanes to fly

◆ Soft real-time
  ➢ Windows Media Player
  ➢ Software DVD player
  ➢ Network router
  ➢ Games
  ➢ Web server
  ➢ Missing deadlines reduces quality of user experience
Real-Time Abstractions

◆ System contains n periodic tasks $T_1, \ldots , T_n$

◆ $T_i$ is specified by $(P_i, C_i, D_i)$
  - P is period
  - C is worst-case execution cost
  - D is relative deadline

◆ Task $T_i$ is “released” at start of period, executes for $C_i$ time units, must finish before $D_i$ time units have passed
  - Often $P_i = D_i$, and in this case we omit $D_i$

◆ Intuition behind this model:
  - Real-time systems perform repeated computations that have characteristic rates and response-time requirements

◆ What about non-periodic tasks?
Real Time Scheduling

- Given a collection of runnable tasks, the scheduler decides which to run
  - If the scheduler picks the wrong task, deadlines may be missed

- Interesting schedulers:
  - Fixed priorities
  - Round robin
  - Earliest deadline first (EDF)
  - Many, many more exist

- A scheduler is optimal when, for a class of real-time systems, it can schedule any task set that can be scheduled by any algorithm
Real-Time Analysis

- **Given:**
  - A set of real-time tasks
  - A scheduling algorithm

- **Is the task set schedulable?**
  - Yes → all deadlines met, always
  - No → at some point a deadline might be missed

- **Important:** Answer this question at design time

- **Other questions to ask:**
  - Where does worst-case execution cost come from?
  - How close to schedulable is a non-schedulable task set?
  - How close to non-schedulable is a schedulable task set?
  - What happens if we change scheduling algorithms?
  - What happens if we change some task’s period or execution cost?
Cyclic Schedule

- This is an important way to sequence tasks in a real-time system
  - We’ll look at other ways later
- Cyclic scheduling is static – computed offline and stored in a table
  - For now we assume table is given
  - Later look at constructing scheduling tables
- Task scheduling is non-preemptive
  - No RTOS is required
- Non-periodic work can be run during time slots not used by periodic tasks
  - Implicit low priority for non-periodic work
  - Usually non-periodic work must be scheduled preemptively
Cyclic Schedule Table

\[ T(t_k) = \begin{cases} 
T_i & \text{if } T_i \text{ is to be scheduled at time } t_k \\
I & \text{if no periodic task is scheduled at time } t_k 
\end{cases} \]

- **Table executes completely in one hyperperiod** \( H \)
  - Then repeats
  - \( H \) is least common multiple of all task periods
  - \( N \) quanta per hyperperiod

- **Multiple tables can support multiple system modes**
  - E.g., an aircraft might support takeoff, cruising, landing, and taxiing modes
  - Mode switches permitted only at hyperperiod boundaries
    - Otherwise, hard to meet deadlines
Example

- Consider a system with four tasks
  - $T_1 = (4, 1)$
  - $T_2 = (5, 1.8)$
  - $T_3 = (20, 1)$
  - $T_4 = (20, 2)$
- Possible schedule:
- Table starts out with:
  - $(0, T_1), (1, T_3), (2, T_2), (3.8, I), (4, T_1), ...$
Refinement: Frames

- We divide hyperperiods into *frames*
  - Timing is enforced only at frame boundaries
  - Each task is executed as a function call and must fit within a single frame
  - Multiple tasks may be executed in a frame
  - Frame size is $f$
  - Number of frames per hyperperiod is $F = \frac{H}{f}$
Frame Size Constraints

1. Tasks must fit into frames
   - So, $f \geq C_i$ for all tasks
   - Justification: Non-preemptive tasks should finish executing within a single frame

2. $f$ must evenly divide $H$
   - Equivalently, $f$ must evenly divide $P_i$ for some task $i$
   - Justification: Keep table size small
3. There should be a complete frame between the release and deadline of every task
   
   - Justification: Want to detect missed deadlines by the time the deadline arrives

   Therefore: \( 2f - \gcd(P_i, f) \leq D_i \) for each task \( i \)
Example Revisited

- Consider a system with four tasks
  - $T_1 = (4,1), T_2 = (5, 1.8), T_3 = (20, 1), T_4 = (20, 2)$
  - $H = \text{lcm}(4,5,20) = 20$
- By Constraint 1: $f \geq 2$
- By Constraint 2: $f$ might be 1, 2, 4, 5, 10, or 20
- By Constraint 3: only 2 works
Task Slices

- What if frame size constraints cannot be met?
  - Example: \( T = \{ (4, 1), (5, 2, 7), (20, 5) \} \)
    - By Constraint 1: \( f \geq 5 \)
    - By Constraint 3: \( f \leq 4 \)
  - Solution: “slice” a task into smaller sub-tasks
    - So \( (20, 5) \) becomes \( (20, 1), (20, 3), \) and \( (20, 1) \)
    - Now \( f = 4 \) works

- What is involved in slicing?
Design Decision Summary

- Three decisions:
  - Choose frame size
  - Partition tasks into slices
  - Place slices into frames
- In general these decisions are not independent
Cyclic Executive Pseudocode

// L is the stored schedule

current time \( t = 0 \);
current frame \( k = 0 \);

do forever
   accept clock interrupt;
currentBlock = L(k);
t++;  \( k = t \mod F \);
if last task not completed, take appropriate action;
execute slices in currentBlock;
sleep until next clock interrupt;
Practical Considerations

- **Handling frame overrun**
  - Main issue: Should offending task be completed or aborted?
  - How can we eliminate the possibility of overrun?

- **Mode changes**
  - At hyperperiod boundaries
  - How to schedule the code that figures out when it’s time to change modes?

- **Multiprocessor systems**
  - Similar to uniprocessor but table construction is more difficult

- **Splitting tasks**
  - Painful and error prone
Computing a Static Schedule

- **Problem:** Derive a frame size and schedule meeting all constraints
- **Solution:** Reduce to a network flow problem
  - Use constraints to compute all possible frame sizes
  - For each possible size, try to find a schedule using network flow algorithm
    - If flow has a certain value:
      - A schedule is found and we’re done
    - Otherwise:
      - Schedule is not found, look at the next frame size
  - If no frame size works, system is not schedulable using cyclic executive
Network Flow Problem

- Given a graph of links, each with a fixed capacity, determine the maximum flow through the network
- Efficient algorithms exist

Figure 1a - Maximum Flow in a network
Flow Graph Definitions

- Denote all jobs in hyperperiod of F frames as $J_1 \ldots J_n$
- Vertices:
  - N job vertices $J_1$, $J_2$, ..., $J_N$
  - F frame vertices 1, 2, ..., F
- Edges:
  - $(source, J_i)$ with capacity $C_i$
    - Encodes jobs’ compute requirements
  - $(J_i, x)$ with capacity $f$ iff $J_i$ can be scheduled in frame $x$
    - Encodes periods and deadlines
  - $(f, sink)$ with capacity $f$
    - Encodes limited computational capacity in each frame
Flow Graph Illustration

Jobs

Frames

Source

Sink

C_i

J_i

f

J_k

C_k

x

y

z

x

y

z
Finding a Schedule

- Maximum attainable flow is $\Sigma_{i=1..N} C_i$
  - Total amount of computation in the hyperperiod
  - If a max flow is found with this amount then we have a schedule

- If a task is scheduled across multiple frames, we must slice it into subtasks
  - Potentially difficult
  - However, if we don’t allow the algorithm to split tasks, the problem becomes NP-complete
    - Common pattern in this sort of problem
      - E.g. optimal bin packing becomes easy if we can split objects
This flow is telling us to split $J_i$ into two jobs, one in $x$ and one in $y$, while $J_k$ executes entirely in $y$.
Non-Independent Tasks

- **Precedence constraints**: “T_i must execute before T_j”
  - Enforce these by adjusting tasks’ release times and deadlines

- **Critical sections**: “T_i must not be sliced in such a way that T_j runs in the middle”
  - These make the problem of finding a schedule NP-hard
CE Advantages

- Main advantage: Cyclic executives are very simple – you just need a table
  - Table makes the system very predictable
    - Can validate and test with very high confidence
  - No race conditions, no deadlock
  - No processes, no threads, no locks, ...
  - Task dispatch is very efficient: just a function call
  - Lack of scheduling anomalies
CE Disadvantages

- Cyclic executives are brittle – any change requires a new table to be computed
- Release times of tasks must be fixed
- F could be huge
  - Implies mode changes may have long latency
- All combinations of tasks that could execute together must be analyzed
- Slicing tasks into smaller units is difficult and error-prone
Summary

- Cyclic executive is one of the major software architectures for embedded systems
  - Historically, cyclic executives dominate safety-critical systems
  - Simplicity and predictability win
  - However, there are significant drawbacks
  - Finding a schedule might require significant offline computation