Last Time

- Priority-based scheduling
  - Static priorities
  - Dynamic priorities
- Schedulable utilization
- Rate monotonic rule: Keep utilization below 69%
Today

- Response time analysis
- Blocking terms
- Priority inversion
  - And solutions
- Release jitter
- Other extensions
Response Time vs. RM

- **Rate monotonic result**
  - Tells us that a broad class of embedded systems meet their time constraints:
    - Scheduled using fixed priorities with RM or DM priority assignment
    - Total utilization not above 69%
  - However, doesn’t give very good feedback about what is going on with a specific system

- **Response time analysis**
  - Tells us for each task, what is the longest time between when it is released and when it finishes
  - Then these can be compared with deadlines
  - Gives insight into how close the system is to meeting / not meeting its deadline
  - Is more precise (rejects fewer systems)
Computing Response Time

- WC response time of highest priority task $R_1$
  - $R_1 = C_1$
  - Hopefully obvious

- WC response time of second-priority task $R_2$
  - Case 1: $R_2 \leq T_1$
    - $R_2 = C_2 + C_1$
More Second-Priority

- **Case 2**: $T_1 < R_2 \leq 2T_1$
  - $R_2 = C_2 + 2C_1$

- **Case 3**: $2T_1 < R_2 \leq 3T_1$
  - $R_2 = C_2 + 3C_1$

- **General case of the second-priority task**: $R_2 = C_2 + \text{ceiling} \left( \frac{R_2}{T_1} \right) C_1$
Task i Response Time

◆ General case:

\[ R_i = C_i + \sum_{\forall j \in hp(i)} \left[ \frac{R_i}{T_j} \right] C_j \]

◆ hp(i) is the set of tasks with priority higher than i
  ➢ Only higher-priority tasks can delay a task

◆ Problem with using this equation in practice?
Computing Response Times

- Rewrite as a recurrence relation and solve by iterating:
  \[
  R_i^{n+1} = C_i + \sum_{\forall j \in \text{hp}(i)} \left( \frac{R_i^n}{T_j} \right) C_j
  \]

- Finished when \( R_i^{n+1} = R_i^n \)
  - Or when \( R_i^n > D_i \)

- Choose \( R_i^0 = 0 \) or \( R_i^0 = C_i \)
  - There may be many solutions to the recurrence
  - These starting points guarantee convergence to the smallest solution (unless there is divergence)

- Result is invalid if \( R_i > T_i \)
  - Why?
Response Time Example

- Task 1: T = 30, D = 30, C = 10
- Task 2: T = 40, D = 40, C = 10
- Task 3: T = 52, D = 52, C = 12
- Utilization = 81% – Rejected by the rate monotonic test!

\[
R_{i}^{n+1} = C_i + \sum_{\forall j \in h_p(i)} \left[ \frac{R_i^n}{T_j} \right] C_j
\]

- \(R_1 = 10\)
- \(R_2 = 20\)
- \(R_3 = 52\)
Sharing Resources

- So far tasks are assumed to be independent
  - Not allowed to block (e.g. on a network device)
  - Not allowed to contend for shared resources
- Big problem in practice!
- Solution:
  - Compute *worst-case blocking time* for each task
  - Longest time that task is delayed by a lower-priority task
  - Why just lower priority?
- Now we can analyze the system again:

\[
R_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left[ \frac{R_i^n}{T_j} \right] C_j
\]
Computing Blocking Terms

- How do we compute blocking terms?
  - Depends on the synchronization protocol
- Tasks synchronize by disabling interrupts
  - Best answer: Each task gets blocking term with length of the longest critical section in a lower-priority task
  - Simpler answer: Each task gets blocking term with length of the longest critical section in any task
  - Why do these work?
- Tasks synchronize using mutexes
  - Blocking term generally impossible to bound – oops!
  - Standard thread locks are unfriendly to real-time systems
    - Lock wait queue is FIFO
  - Possible solution: Priority queues for mutexes
Priority Inversion

- Priority inversion: Low-priority task delays a high priority task
  - Mutexes (even with priority queuing) provide unbounded priority inversion
Priority Inversion Case Study

- Mars Pathfinder
  - Lands on Mars July 4 1997
  - Mission is successful

- Behind the scenes...
  - Sporadic total system resets on the rover
  - Caused by priority inversion
  - Debugged on the ground, software patch uploaded to fix things

- Details
  - Rover controlled by a single RS6000 running vxWorks
  - Rover devices polled over 1553 bus
  - At 8 Hz bc_sched task sets up bus transactions
  - bc_dist task runs (also at 8 Hz) to read back data
More Pathfinder

◆ Symptom:
  ➢ bc_sched sometimes was not finished by the time bc_dist ran
  ➢ This triggered a system reset
    • Should never happen since these tasks are high priority

◆ Problem: bc_sched shared a mutex with ASI/MET task, which does meteorological science at low priority
  ➢ Occasionally the classic priority inversion happened when there were long-running medium priority tasks

◆ Solution:
  ➢ vxWorks supports “priority inheritance” with a global flag
  ➢ They turned it on
Priority Inversion Solutions

1. Avoid blocking – disable interrupts instead
   - Pros:
     - Efficient
     - Simple
   - Con:
     - Also delays unrelated, high priority tasks

2. Immediate priority ceiling protocol – before locking, raise priority to highest priority of any thread that can touch that semaphore
   - Pros:
     - Fairly simple
     - Less blocking of unrelated tasks
   - Cons:
     - Requires ahead-of-time system analysis
     - Still has some pessimistic blocking
Priority Inversion Solutions

3. Priority inheritance protocol – When a task is blocking other tasks (by holding a mutex) it executes at the priority of the highest-priority blocked task

- **Pros**
  - No pessimistic blocking

- **Cons**
  - Complicated in presence of nested locking
  - Not that efficient
  - Blocking terms larger than IPCP

- Other solutions exist, such as lock-free synchronization
In IPCP, raising priority prevents anyone else who might access a resource from running

- So why take a lock at all?
- Turns out that locking is not necessary – raising priority is enough
- HOWEVER: Task must not voluntarily block (e.g. on disk or network) while in a critical section
Overheads

◆ A real RTOS requires time to:
  ➢ Block a task
  ➢ Make a scheduling decision
  ➢ Dispatch a new task
  ➢ Handle timer interrupts

◆ For a well-designed RTOS these times can be bounded
  ➢ Worst-case blocking time of the RTOS needs to be added to each task’s blocking term
  ➢ 2x worst-case context switch time needs to be added to each task’s WCET
    • We always “charge” the cost of a context switch to the higher-priority task
Release Jitter

- Release jitter $J_i$ – Time between invocation of task $i$ and time at which it can actually run
  - E.g. task becomes conceptually runnable at the start of its period
    - But must wait for the next timer interrupt before the scheduler sees it and dispatches it
  - Or, task would like to run but must wait for network data to arrive before it actually runs

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left[ \frac{R_i + J_i}{T_j} \right] C_j$$
Other Extensions

- **Sporadically periodic tasks**
  - Task has an “outer period” and smaller “inner period”
  - Models bursty processing like network interrupts

- **Sporadic servers**
  - Provide rate-limiting for truly aperiodic processing
    - E.g. interrupts from an untrusted device

- **Arbitrary deadlines**
  - When $D_i > T_i$ previous equations do not apply
  - Can rewrite

- **Precedence constraints**
  - Task A cannot run until Task B has completed
    - Models scenario where tasks feed data to each other
  - Makes it harder to schedule a system
Summary

◆ Priority based scheduling
  ➢ It’s what RTOSs support
  ➢ A strong body of theory can be used to analyze these systems
  ➢ Theory is practical: Many real-world factors can be modeled

◆ Response time analysis – supports worst-case response time for each priority-based task
  ➢ Blocking terms
  ➢ Release jitter

◆ Priority inversion can be a major problem
  ➢ Solutions have interesting tradeoffs