Overview

- Aperiodic tasks
- Response time analysis
- Blocking
- Priority inversion
  - Priority inheritance
  - Priority ceiling

Recall

Initial restricted model:
- Only periodic tasks
- Deadlines = periods
- No blocking (that is, no message communication / no mutex variables)

We have:
- Optimal priority assignment methods
  - RM and EDF
- Simple utilization-based feasibility test
  - sufficient for RM
  - necessary & sufficient for EDF

What properties hold if we loosen the restrictions?

Sporadic tasks

Actual behavior:

0 10 20 30 40 50 60 70 80

Worst case:

0 10 20 30 40 50 60 70 80

That is, in the worst case, a sporadic task with minimum inter-arrival time $T$ degenerates to a periodic task with period $T$.

Aperiodic tasks

- An aperiodic task is not periodic
- lacks a minimum inter-arrival time
- easily consume the whole CPU under bad circumstances (dense arrivals)
- suggests giving aperiodic tasks the lowest priorities?
- However, under "better" circumstances
  - relative importance of an aperiodic task might suggest a higher priority...
- common solution
  - associate an "account" of cpu cycles with aperiodic task
  - as long as there are cycles on the account, priority is high, otherwise it drops
- However, analysis of aperiodic tasks is beyond our scope...

Loosening $D = T$

- Let deadlines be less than periods, i.e., $D < T$.
- Deadline Monotonic priority assignment (DM)
  - "the shorter the relative deadline, the higher the priority"
  - note that RM is just a special case of DM
- EDF is already based on (absolute) deadlines
- $D < T$ imposes no algorithm changes
- Both DM and EDF can be proven optimal in the $D < T$ case
- schedulability analysis
  - more powerful technique than simple utilization-based testing is needed
- Response-time analysis
- We'll study the DM case (a similar analysis exists for EDF)
Response time analysis

- A more elaborate schedulability test for DM/RM
- Both sufficient and necessary
- Idea: compute the response time for each task, and compare with deadline
- Highest priority task: response time equals execution time
- Second but highest: response time is execution time + interference by highest priority task
- Repeat recursively
- Note: interference depends on period ratio

For each task i, we know:
- Its period Ti (given by control algorithm)
- Its execution time Ci (measured, or analyzed)
- Its relative deadline Di (given)
- Its priority Pi (determined by Di according to DM)

We want to compute worst-case Ri, and check whether Ri ≤ Di.

Example C revisited

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Deadline</th>
<th>Priority</th>
<th>WCET</th>
<th>Preemption</th>
<th>Response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>80</td>
<td>low</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>40</td>
<td>mid</td>
<td>10</td>
<td>(2,5)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>(10)</td>
<td>high</td>
<td>5</td>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

Slight variation compared to last lecture

- R_0 = C_1
- R_0 = C_1 + \left\lfloor \frac{R_0}{T_1} \right\rfloor C_2
- R_0 = C_1 + \left\lfloor \frac{R_0}{T_2} \right\rfloor C_2 + \left\lfloor \frac{R_0}{T_2} \right\rfloor C_3

Example C

- R_0 = 85
- R_0 = 10
- R_0 = 10 + \left\lfloor \frac{10}{80} \right\rfloor = 11
- R_0 = 10 + \left\lfloor \frac{15}{80} \right\rfloor = 12

Stable!
### Example C

All tasks have response times less than or equal to their deadlines. That is, the example C system is guaranteed to meet its deadlines at runtime.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Deadline</th>
<th>Priority</th>
<th>WCET</th>
<th>Preempted by</th>
<th>Response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>80</td>
<td>low</td>
<td>40</td>
<td>{2,3}</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>40</td>
<td>mid</td>
<td>10</td>
<td>{3}</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>10</td>
<td>high</td>
<td>5</td>
<td>{}</td>
<td>5</td>
</tr>
</tbody>
</table>

### Introducing blocking

Prio low

```c
int A (ClassA *self, int arg) {
    // some computations...
    AFTER(T1, self, A, arg);
}
```

Prio high

```c
int B (ClassB *self, int arg) {
    // some computations...
    AFTER(T2, self, B, arg);
}
```

(Independent tasks)

What happens if B calls `SYNC(&obj, C)` while `A` is already executing `C` in response to `A`?

### Introducing blocking

Prio low

```c
int A (ClassA *self, int arg) {
    // some computations...
    lock(&obj);
    C(&obj, arg);
    unlock(&obj);
    AFTER(T1, self, A, arg);
}
```

Prio high

```c
int B (ClassB *self, int arg) {
    // some computations...
    lock(&obj);
    C(&obj, arg);
    unlock(&obj);
    AFTER(T2, self, B, arg);
}
```

What happens if B tries to lock `obj` while A has the lock?

### Introducing blocking

What happens if B calls `SYNC(&obj, C)` while obj is already executing C in response to A?

Response times unaffected in this particular scenario...
Introducing blocking

Response times unaffected by the locking of obj in this particular scenario too.

Priorities and mutexes

- Note: B is blocked waiting for A, even though B has higher priority than A!
- This paradox is unavoidable, unless one would like to consider terminating A in such situations... (bad idea)
- Note, though, that A and B are deliberately connected via obj, and B will only block while A is actively working towards giving B something B wants (the lock)
- If the WCET of C is C, it seems an extra C would just have to be added to the response time for B
- But now consider a third task X with mid priority...

Priority inversion

The scenario:
- B must wait for A, even though B has higher priority. However, B is deliberately connected to A via the mutex obj, so this is ok
- While B waits and A runs, a mid-task X intervenes
- Now B effectively must wait for X, even though B has higher priority than X, and B and X are not connected via a common mutex!
- This effect is called priority inversion, and is a major problem in priority-based systems.
Curing priority inversion

Priority inversion can be avoided by letting A temporarily run at a higher priority, so that X won’t be able interrupt it.

Two popular mechanisms:
- Priority inheritance: raise the priority of A to the priority of B as long B is waiting for the lock held by A.
- Priority ceiling: raise the priority of A to a predefined ceiling value immediately it grabs the lock.

Calculating blocking time

With priority inheritance, any task i can become blocked once for every mutex that is shared by i or any higher priority task, and a lower priority task.

In the worst case, the blocking time for a mutex k is the length of the longest critical section (in all tasks) that acquires k.

Formally:  \( B_i = \sum_{k \in K} \max(C_i, C_k) \cdot C_k \)

where \( K \) is the set of mutexes, \( C_i \) is the length of the longest critical section locking k, and \( \max(C_i, C_k) \) is a function yielding 1 if a task less important than \( i \) shares \( k \) with \( i \) or any task above, and 0 otherwise.

Refined response times

Taking blocking times into account, the interactive formula for calculating task response times can now be stated as:

\[ R_i = C_i + B_i + \sum_{s \in \text{mutexes}} \frac{R_s}{C_j} \cdot C_j \]

Notice that since \( B_i \) is an upper bound - that is, a safe approximation - response time analysis based on the above formula is no longer a necessary test; it is only sufficient.

Priority ceiling

-缺点 with priority inheritance
  - High priority task blocked on mutex access several times
  - Because of low priority tasks
  - Once per mutex used
  - One (big) blocking period at the first mutex
  - Reduced blocking overhead
  - Priority is boosted to a ceiling value
  - When the first mutex is granted
  - Provided the ceiling value is higher than the normal priority of any contending tasks
Priority ceiling protocol

High pri. wants to start (and wants mutex)
High pri. starts (low pri. released mutex)
High pri. gets mutex immediately
Mid pri. starts
Low pri. continues
Mid pri. wants to start (delayed)

Both mechanisms solve the priority inversion problem; both used in practice
However, priority ceiling also requires that the ceiling value be calculated and set manually – requires knowledge of all users of a resource
Priority ceiling actually makes all but the first mutex of a task superfluous!
Priority inheritance is a somewhat less complex concept

Summarizing

The analysis techniques we’ve encountered can handle
• Periodic as well as aperiodic tasks
• Deadlines ≤ periods (or shortest inter-arrival times)
• Communication via messages or mutex-protected variables (amounts to the same thing in our model), assuming all task dependencies are statically known
Other techniques not covered here may take care of
• Interrupt handler WCET > 0
• Context-switch overhead > 0
• Budgets for fully aperiodic tasks

Summarizing

Still not handled very well:
• Deadlines ≥ periods (or shortest inter-arrival times)
• Arbitrary baseline offsets (not periodic)
• State-dependencies (WCETs, periods, deadlines)
• Dynamically changing task communication structures
• Memory usage (heap, thread stacks)
We have studied hard real-time systems
Deadlines are firm and Worst Case Execution Times are required
An alternative field of study: soft real-time systems
Statistical distributions and Mean Execution Times are of primary interest

Hard real-time systems

The analysis techniques we’ve encountered can handle
• Periodic as well as aperiodic tasks
• Deadlines ≤ periods (or shortest inter-arrival times)
• Communication via messages or mutex-protected variables (amounts to the same thing in our model), assuming all task dependencies are statically known
Other techniques not covered here may take care of
• Interrupt handler WCET > 0
• Context-switch overhead > 0
• Budgets for fully aperiodic tasks

Soft real-time systems

Still not handled very well:
• Deadlines ≥ periods (or shortest inter-arrival times)
• Arbitrary baseline offsets (not periodic)
• State-dependencies (WCETs, periods, deadlines)
• Dynamically changing task communication structures
• Memory usage (heap, thread stacks)
We have studied hard real-time systems
Deadlines are firm and Worst Case Execution Times are required
An alternative field of study: soft real-time systems
Statistical distributions and Mean Execution Times are of primary interest