Introduction to Real-Time Systems

Note: Slides are adopted from Lui Sha and Marco Caccamo
Overview

• Today: this lecture introduces real-time scheduling theory

• To learn more on real-time scheduling terminology:
  - see chapter 4 see basic concepts on “Hard Real-Time Computing Systems” book from G. Buttazzo

• Basic tutorial at:
  http://www.embedded.com/electronics-blogs/beginner-s-corner/4023927/Introduction-to-Rate-Monotonic-Scheduling#
So What is a Real-Time System?

- A **real-time system** is a system whose specification includes both logical and temporal correctness requirements.
  
  ✓ **Logical Correctness**: produces correct outputs.
  
  ✓ **Temporal Correctness**: produces outputs at the right time.
So What is a Real-Time System?

- A real-time system has different set of measure of merits:

<table>
<thead>
<tr>
<th></th>
<th>Real time systems</th>
<th>Non-real time systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>Schedulable Utilization (schedulability)</td>
<td>throughput</td>
</tr>
<tr>
<td>responsiveness</td>
<td>Worst case response time of each tasks</td>
<td>Average response time</td>
</tr>
<tr>
<td>overload</td>
<td>Stability (getting critical tasks done)</td>
<td>Fairness</td>
</tr>
</tbody>
</table>
What does Real-Time mean?

• The word **time** means that the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

• The word **real** indicates that the reaction of the system to external events must occur during their evolution. As a consequence, the system time (internal time) must be measured using the same time scale used for measuring the time in the controlled environment (external time).

  [in chapter 1 of Buttazzo’ s book]
Real-Time systems

- Advances in computer hardware will not take care of the temporal requirements needed by a real-time system.

  ✓ The old “buy a faster processor” argument does not work!
  ✓ An old Pentium can be used to run a real-time application
  ✓ A last generation pc with a general purpose operating system (windows, linux, etc.) can violate the temporal constraints of our real-time application.

- Rather than being fast, a real-time computing system should be **predictable**.

- What if you need **fast** AND **predictable**?
Are All Systems Real-Time systems?

- Question: is a payroll processing system a real-time system?
  - It has a time constraint: print the pay checks every two weeks

- Perhaps it is a real-time system in a definitional sense, but it doesn’t pay us to view it as such.

- We are interested in systems for which it is not a priori obvious how to meet timing constraints.
Typical Real-Time Systems

- Cell phones, digital cameras
- Avionic
- Radar Systems
- Factory Process control
- Sensing and Control
- Multi-media systems
- Cruise control system in a car
- All of them have explicit timing requirements to meet.
Jobs and Tasks

- A job is a unit of computation, e.g.,
  - handling the press of a keyboard
  - or compute the control response in one instance of a control loop

- A task is a sequence of the same type of jobs, say, a control task or the keyboard handling task.
Periodic Task Model

- Periodic tasks are the “work horse” of real-time systems and they play a key role in real-time systems.

- A task $\tau_i$ is said to be periodic if its inter-arrival time (period), $p_i$, is a constant

- A periodic task, $\tau_i$, is characterized by
  - Period, $p_i$
  - Release time, $r_{i,j}$. The default is $r_{i,j} = r_{i,j-1} + p_i$
  - Execution, $C_i$. The default is worst-case execution time
  - Relative deadline $D_i$. The default is equal to period
  - Phase $\phi_i$: the starting time of the task, i.e., the first release time ($r_{i,1}$).
Release Time and Deadlines

- Release time is the instant at which the job becomes ready to execute.
- The common form of deadlines are absolute deadlines where deadlines are specified in, well, absolute times. Train and airlines schedules have absolute deadlines.
- Normally, relative deadlines are related to the release time. For example, a relative deadline D=8 msec after the release time.
- The default absolute deadline of a task is the end of period. By convention, we will refer to an absolute deadline as “d”, and a relative deadline as “D”.

![Diagram showing release time and deadlines]
A Sample Problem

**Periodic tasks**
- $\tau_1$: 20 msec
- $\tau_2$: 40 msec
- $\tau_3$: 100 msec

**Periodic tasks**
- 100 msec (period)
- 150 msec (period)
- 350 msec (period)

**Shared resources**
- (protected by mutex)
- shared data1
- shared data2

**Aperiodics**
- Emergency
  - 50 msec (min interarrival time)
  - 5 msec
- Deadline 6 msec after arrival
- Non-critical display
  - 40 msec (avg interarrival time)
  - 2 msec
- Desired response 20 msec average

**Goal:** guarantee that no real-time deadline is missed!!!
Real-time scheduling algorithms

• Jobs can be scheduled according to the following scheduling algorithms:

  • **Rate Monotonic (RM):** the faster the rate, the higher is the priority. All the jobs in a task have the same priority and hence the name “fixed priority” algorithm.

  • **Earliest Deadline First (EDF):** the job with the earliest deadline has the highest priority. Jobs in a task have different priorities and hence the name, “dynamic priority” algorithm.
Priority and Criticality - 1

- Priority: priority is the order we execute ready jobs.
- Criticality (Importance): represents the penalty if a task misses a deadline (one of its jobs misses a deadline).

- Quiz: Which task should have higher priority?
- Task 1: The most important task in the system: if it does not get done, serious consequences will occur
- Task 2: A mp3 player: if it does not get done in time, the played song will have a glitch

- If it is feasible, we would like to meet the real-time deadlines of both tasks!
Priority and Criticality - 1

- Priority: priority is the order we execute ready jobs.
- Criticality (Importance): represents the penalty if a task misses a deadline (one of its jobs misses a deadline).

- Quiz: Which task should have higher priority?
- Answer: the task with higher rate (according to RM) unless the system is overloaded!

- Task 1: The most important task in the system: if it does not get done, serious consequences will occur
- Task 2: A mp3 player: if it does not get done in time, the played song will have a glitch

- If it is feasible, we would like to meet the real-time deadlines of both tasks!
If priorities are assigned according to importance, there is no lower bound of processor utilization, below which tasks deadlines can be guaranteed.

\[
\frac{C_1}{p_1} + \frac{C_2}{p_2} = U
\]

\[
U \rightarrow 0, \text{ when } C_2 \rightarrow 0 \text{ and } p_1 \rightarrow \infty
\]

Task \( T_2 \) will miss its deadline, as long as \( C_1 > p_2 \)
Priority and Criticality - 3

- An important find in real-time computing theory is that importance may or may not correspond to scheduling priority.
- In the previous example, giving the less important task higher priority results in both tasks meeting their deadlines.
- Importance matters only when tasks cannot be scheduled (overload condition) without missing deadlines.
Utilization and Schedulability

- A task’s utilization (of the CPU) is the fraction of time for which it uses the CPU (over a long interval of time).

- A periodic task’s utilization $U_i$ (of CPU) is the ratio between its execution time and period: $U_i = C_i/p_i$

- Given a set of periodic tasks, the total CPU’s utilization is equal to the sum of periodic tasks’ utilization:

$$U = \sum_{i} \frac{C_i}{p_i}$$

- Schedulability bound of a scheduling algorithm is the percentage of CPU utilization at or below which a set of periodic tasks can always meet their deadlines. You may think of it as a standard benchmark for the effectiveness of a scheduling algorithm.

- QUIZ: What is the obvious limit on $U$?
Utilization and Schedulability

- A task’s utilization (of the CPU) is the fraction of time for which it uses the CPU (over a long interval of time).

- A periodic task’s utilization $U_i$ (of CPU) is the ratio between its execution time and period: $U_i = C_i/p_i$

- Given a set of periodic tasks, the total CPU’s utilization is equal to the sum of periodic tasks’ utilization:

$$U = \sum_{i} \frac{C_i}{p_i}$$

- Schedulability bound of a scheduling algorithm is the percentage of CPU utilization at or below which a set of periodic tasks can always meet their deadlines. You may think of it as a standard benchmark for the effectiveness of a scheduling algorithm.

- QUIZ: What is the obvious limit on $U$?
- ANSWER: 1, you cannot utilize more than 100% of the processor capacity!
Real-time scheduling algorithms

- Scheduling algorithms need to be simple: cannot use many processor cycles

- Static vs. dynamic priorities
  - Static priority: All jobs of a task have the same priority
  - Dynamic priority: Different jobs of the same task may have different priorities

- Examples
  - Rate Monotonic Scheduling [RM]: Tasks with smaller periods are assigned higher priorities (static priority)
  - Earliest Deadline First [EDF]: Jobs are prioritized based on absolute deadlines (dynamic priority)
Dynamic vs “Static” Priority Scheduling in Theory

- An instance of a task is called a job. “Static” priority assigns a (base) priority to all the jobs in a task. Dynamic priority scheduling adjusts priorities in each task job by job.

- Quiz: what type of scheduling algorithm is used to schedule these two tasks?

- An optimal dynamic scheduling algorithm is the earlier deadline first (EDF) algorithm. Jobs closer to deadlines will have higher priority. With independent periodic tasks, all tasks will meet their deadlines as long as the processor utilization is less than or equal to 1.

- An optimal “static” scheduling algorithm is the rate monotonic scheduling (RMS) algorithm. For a periodic task, the higher the rate (frequency) the higher the priority.
Dynamic vs “Static” Priority Scheduling in Theory

- An instance of a task is called a job. “Static” priority assigns a (base) priority to all the jobs in a task. Dynamic priority scheduling adjusts priorities in each task job by job.

- Quiz: what type of scheduling algorithm is used to schedule these two tasks?
- Answer: EDF

- An optimal dynamic scheduling algorithm is the earlier deadline first (EDF) algorithm. Jobs closer to deadlines will have high priority. With independent periodic tasks, all tasks will meet their deadlines as long as the processor utilization is less than 1.

- An optimal “static” scheduling algorithm is the rate monotonic scheduling (RMS) algorithm. For a periodic task, the higher the rate (frequency) the higher the priority.
Importance of the scheduling algorithm

- To demonstrate the importance of a scheduling algorithm, consider a system with only two tasks, T1 and T2. Assume these are both periodic tasks with periods p1 and p2, and each has a deadline that is the beginning of its next cycle.

- Task 1 has p1 = 50 ms, and a worst-case execution time of C1 = 25 ms. Task 2 has p2 = 100 ms and C2 = 40 ms. Note that the utilization, Ui, of task i is Ci/Ti. Thus U1 = 50% and U2 = 40%. This means total requested utilization U = U1 + U2 = 90%. It seems logical that if utilization is less than 100%, there should be enough available CPU time to execute both tasks.

- Let's consider a static priority scheduling algorithm. With two tasks, there are only two possibilities:
  - Case 1: Priority(T1) > Priority(T2)
  - Case 2: Priority(T1) < Priority(T2)

- The two cases are shown in next figure. In Case 1, both tasks meet their respective deadlines. In Case 2, however, Task 1 misses a deadline, despite 10% idle time. This illustrates the importance of priority assignment.
Importance of the scheduling algorithm

Both possible outcomes for static-priority scheduling with two tasks \((T1=50, C1=25, T2=100, C2=40)\)

Importance of the scheduling algorithm

- It is theoretically possible for a set of tasks to require just 70% CPU utilization in sum and still not meet all their deadlines. For example, consider the case shown in the Figure below. The only change is that both the period and execution time of Task 2 have been lowered. Based on RM, Task 1 is assigned higher priority. Despite only 90% utilization, Task 2 misses its first deadline. Reversing priorities would not have improved the situation.

Some task sets aren't schedulable \((T1=50, C1=25, T2=75, C2=30)\)

Exercise: try to use EDF and check if the first deadlines are met!
Key scheduling results

- For periodic tasks with relative deadlines equal to their periods:
  - Rate monotonic priority assignment is the optimal static priority assignment
    - No other static priority assignment can do better
    - Yet, it cannot achieve 100% CPU utilization

- Earliest deadline first scheduling is the optimal dynamic priority policy
  - EDF can achieve 100% CPU utilization

- More details in the next lecture
Recap

- Job, Task

- Periodic task model
  - \( t_i = (C_i, P_i) \) or \( (C_i, P_i, D_i) \)

- Static/dynamic priority scheduling:
  - RM
  - EDF

- Utilization
  - \( \bar{U}_i = \sum_i \frac{C_i}{P_i} \)
Overview

- Today: this lecture explains how to use Utilization Bound, it introduces the POSIX.4 scheduling interface and the exact analysis

- To learn more on real-time scheduling:
  - see chapter 4 on “Hard Real-Time Computing Systems” book from G. Buttazzo

- To learn more on POSIX.4 scheduling interface:
    See pp.159-171 and 200-207 (available in the Lab)
  - Basic tutorial at http://www.netrino.com/Publications/Glossary/RMA.html
RMS: Less Than 100% Utilization but not Schedulable

In this example, 2 tasks are scheduled under RMS, an optimal static priority method
\[\frac{4}{10} + \frac{6}{14} = 0.83\]

The task set is schedulable but if we try to increase the computation time of task T1, the task set becomes unschedulable in spite of the fact that total utilization is 83%!

To achieve 100% utilization when using fixed priorities, assign periods so that all tasks are harmonic. This means that for each task, its period is an exact multiple of every other task that has a shorter period.

For example, a three-task set whose periods are 10, 20, and 40, respectively, is harmonic, and preferred over a task set with periods 10, 20, and 50.
The Liu & Layland Bound


- A set of $n$ periodic tasks is schedulable if:

$$\frac{c_1}{p_1} + \frac{c_2}{p_2} + ... + \frac{c_n}{p_n} \leq n\left(2^{1/n} - 1\right)$$

- $U(1) = 1.0$, $U(4) = 0.756$, $U(7) = 0.728$
- $U(2) = 0.828$, $U(5) = 0.743$, $U(8) = 0.724$
- $U(3) = 0.779$, $U(6) = 0.734$, $U(9) = 0.720$

- For harmonic task sets, the utilization bound is $U(n)=1.00$ for all $n$. Otherwise, for large $n$, the bound converges to $\ln 2 \sim 0.69$.

- The L&L bound for rate monotonic algorithm is one of the most significant results in real-time scheduling theory. It allows to check the schedulability of a group of tasks with a single test! It is a sufficient condition; hence, it is inconclusive if it fails!
Sample Problem: Applying UB Test

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task $\tau_1$:</td>
<td>20</td>
<td>100</td>
<td>0.200</td>
</tr>
<tr>
<td>Task $\tau_2$:</td>
<td>40</td>
<td>150</td>
<td>0.267</td>
</tr>
<tr>
<td>Task $\tau_3$:</td>
<td>100</td>
<td>350</td>
<td>0.286</td>
</tr>
</tbody>
</table>

- Are all the tasks schedulable?
  
  \[ U(2) = 0.828 \]
  \[ U(3) = 0.779 \]

- What if we double the execution time of task $\tau_1$?
Sample Problem: Applying UB Test

<table>
<thead>
<tr>
<th>Task τ1:</th>
<th>C</th>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task τ1:</td>
<td>20</td>
<td>100</td>
<td>0.200</td>
</tr>
<tr>
<td>Task τ2:</td>
<td>40</td>
<td>150</td>
<td>0.267</td>
</tr>
<tr>
<td>Task τ3:</td>
<td>100</td>
<td>350</td>
<td>0.286</td>
</tr>
</tbody>
</table>

- Are all the tasks schedulable?
- Check the schedulability of T1, T2, and T3: \( U_1 + U_2 + U_3 = 0.753 < U(3) \) \( \Rightarrow \) Schedulable!

- What if we double the execution time of task τ1?
- Check schedulability of T1 and T2: \( \frac{40}{100} + \frac{40}{150} = 0.4 + 0.27 = 0.67 < U(2) \) \( \Rightarrow \) Schedulable!

- Check schedulability of T1, T2 and T3: \( \frac{40}{100} + \frac{40}{150} + \frac{100}{350} = 0.953 > U(3) = 0.779 \)

- UB test is a sufficient condition and thus inconclusive if it fails!
Sample Problem: draw the schedule by using RM and EDF
Sample Problem: draw the schedule by using RM and EDF
Recap

• L&L upper bound
  • Fixed priority RM scheduling
  • Sufficient condition for schedulability
Posix.4 scheduling interfaces

- The real-time scheduling interface offered by POSIX.4 (available on Linux kernel)

- Each process can run with a particular scheduling policy and associated scheduling attributes. Both the policy and the attributes can be changed independently. POSIX.4 defines three policies:

  - SCHED_FIFO: preemptive, priority-based scheduling.
  - SCHED_RR: Preemptive, priority-based scheduling with quanta.
  - SCHED_OTHER: an implementation-defined scheduler
Posix.4 scheduling interfaces

- SCHED_FIFO: preemptive, priority-based scheduling.
- The available priority range can be identified by calling:
sched_get_priority_min(SCHED_FIFO) \rightarrow Linux 2.6 kernel: 1
  sched_get_priority_max(SCHED_FIFO); \rightarrow Linux 2.6 kernel: 99

- SCHED_FIFO can only be used with static priorities higher than 0, which means that
  when a SCHED_FIFO process becomes runnable, it will always preempt
  immediately any currently running normal SCHED_OTHER process. SCHED_FIFO
  is a simple scheduling algorithm without time slicing.

- A process calling sched_yield will be put at the end of its priority list. No other
  events will move a process scheduled under the SCHED_FIFO policy in the wait list
  of runnable processes with equal static priority. A SCHED_FIFO process runs until
  either it is blocked by an I/O request, it is preempted by a higher priority process, it
  calls sched_yield, or it finishes.
Posix.4 scheduling interfaces

- SCHED_RR: preemptive, priority-based scheduling with quanta.
- The available priority range can be identified by calling:
  sched_get_priority_min(SCHED_RR) ➔ Linux 2.6 kernel: 1
  sched_get_priority_max(SCHED_RR); ➔ Linux 2.6 kernel: 99

- *SCHED_RR* is a simple enhancement of *SCHED_FIFO*. Everything described above for *SCHED_FIFO* also applies to *SCHED_RR*, except that each process is only allowed to run for a maximum time quantum. If a *SCHED_RR* process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority.

- A *SCHED_RR* process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved by *sched_rr_get_interval*. 
Posix.4 scheduling interfaces

- **SCHED_OTHER**: an implementation-defined scheduler

- **Default Linux time-sharing scheduler**
  - `SCHED_OTHER` can only be used at static priority 0. `SCHED_OTHER` is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list.

- The dynamic priority is based on the nice level (set by the `nice` or `setpriority` system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all `SCHED_OTHER` processes.
Posix.4 scheduling interfaces

- Child processes inherit the scheduling algorithm and parameters across a fork.

- Memory locking is usually needed for real-time processes to avoid paging delays, this can be done with mlock or mlockall.

- Do not forget!!!!
  ➔ a non-blocking end-less loop in a process scheduled under SCHED_FIFO or SCHED_RR will block all processes with lower priority forever, a software developer should always keep available on the console a shell scheduled under a higher static priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected.

- Since SCHED_FIFO and SCHED_RR processes can preempt other processes forever, only root processes are allowed to activate these policies under Linux.
Posix.4 scheduling interfaces

#include <sched.h>
#include <sys/types.h>
#include <stdio.h>

int fifo_min, fifo_max;
int sched, prio, i;
pid_t pid;
struct sched_param attr;

main()
{
    fifo_min = sched_get_priority_min(SCHED_FIFO); fifo_max = sched_get_priority_max(SCHED_FIFO);

    printf("\n Scheduling informations: input a PID?\n");
    scanf("%d", &pid);
    sched_getparam(pid, &attr);
    printf("process %d uses scheduler %d with priority %d \n", pid, 
    sched_getscheduler(pid), attr.sched_priority);

    printf("\n Let’s modify a process sched parameters: Input the PID, scheduler type, and priority \n");
    scanf("%d %d %d", &pid, &sched, &prio);

    attr.sched_priority = prio;
    i = sched_setscheduler(pid, sched, &attr);
}
Linux Scheduling Framework

- Completely Fair Scheduler (CFS)
- Real-time Schedulers
Completely Fair Scheduler (CFS)

- Each task maintains its virtual time
  - \[ V_i = E_i \times \frac{1}{w_i} \], where \( E \) is executed time, \( w \) is a weight

- Pick the task with the **smallest virtual time**
  - Tasks are sorted according to their virtual times
  - Managed by a **red-black tree**, \( O(\log N) \)
Red-black Tree

- Self-balancing binary search tree
- Insert: $O(\log N)$, Remove: $O(1)$

Figure source: M. Tim Jones, “Inside the Linux 2.6 Completely Fair Scheduler”, IBM developerWorks
CFS: Example

Weights: gcc = 2/3, bigsim=1/3

X-axis: mcu (tick), Y-axis: virtual time

Fair in the long run
CFS: Some Edge Cases

- How to set the virtual time of a new task?
  - Can’t set as zero. Why?
  - System virtual time (SVT)
    - The minimum virtual time among all active tasks
    - cfs_rq->min_vruntime
  - The new task can “catch-up” tasks by setting its virtual time with SVT
CFS: Example 2

Weights: gcc = 2/3, bigsim=1/3

X-axis: mcu (tick), Y-axis: virtual time

gcc slept 15 mcu
kernel/sched/fair.c (CFS)

- calc_delta_fair(delta_exec, curr)
  - Compute scaled virtual runtime V.
  - \( V = \frac{\text{delta_exec} \times 1024}{\text{curr->se.load (task weight)}} \)

- Priority to CFS weight conversion table
  - Priority (Nice value): -20 (highest) ~ +19 (lowest)

- kernel/sched/core.c

```c
const int sched_prio_to_weight[40] = {
    /* -20 */  88761,  71755,  56483,  46273,  36291,
    /* -15 */  29154,  23254,  18705,  14949,  11916,
    /* -10 */   9548,   7620,   6100,   4904,   3906,
    /*  -5 */   3121,   2501,   1991,   1586,   1277,
    /*    0 */   1024,   820,    655,    526,    423,
    /*    5 */    335,    272,    215,    172,    137,
    /*   10 */    110,     87,     70,     56,     45,
    /*   15 */     36,     29,     23,     18,     15,
};
```
Agenda

- Exact schedulability analysis
The Exact Schedulability Test

Critical instant theorem: If a task meets its first deadline when all higher priority tasks are started at the same time, then this task’s future deadlines will always be met. The exact test for a task checks if this task can meet its first deadline [Liu73].

It holds only for fixed priority scheduling!
Exact Schedulability Test (Exact Analysis)

\[ r_i^{k+1} = c_i + \sum_{j=1}^{i-1} \left[ \frac{r_j^k}{p_j} \right] c_j, \quad \text{where } r_i^0 = \sum_{j=1}^{i} c_j \]

Test terminates when \( r_i^{k+1} > p_i \) (not schedulable)
or when \( r_i^{k+1} = r_i^k \leq p_i \) (schedulable).

Tasks are ordered according to their priority: \( T_1 \) is the highest priority task.

The superscript \( k \) indicates the number of iterations in the calculation.
The index \( i \) indicates it is the \( ith \) task being checked.

The index \( j \) runs from 1 to \( i-1 \), i.e. all the higher priority tasks. Recall from the convention - task 1 has a higher priority than task 2 and so on.

We check the schedulability of a single task at the time!!!
The Exact Schedulability Test

- Basically, “Enumerate” the schedule
- “Task by Task” schedulability test

\[(c_1 = 4, p_1 = 10), U_1 = 0.4\]
\[(c_2 = 4, p_2 = 15), U_2 = 0.27\]
\[(c_3 = 10, p_3 = 35), U_3 = 0.28\]

Q: Now, we can say Task 3 is schedulable. Is this correct?
How long should we enumerate the schedule?

**Critical instant theorem**: If a task meets its first deadline when all higher priority tasks are started at the same time, then this task’s future deadlines will always be met. The exact test for a task checks if this task can meet its first deadline [Liu73].

\begin{itemize}
  \item \((c_1 = 4, p_1 = 10), U_1 = 0.4\)
  \item \((c_2 = 4, p_2 = 15), U_2 = 0.27\)
  \item \((c_3 = 10, p_3 = 35), U_3 = 0.28\)
\end{itemize}

**Ans**: Checking the critical instant is OK!!
Intuitions of Exact Schedulability Test

- Obviously, the response time of task 3 should be larger than or equal to $c_1 + c_2 + c_3$

$$r_3^0 = \sum_{j=1}^{3} c_j = c_1 + c_2 + c_3 = 4 + 4 + 10 = 18$$
Intuitions of Exact Schedulability Test

\(c_1 = 4, p_1 = 10\), \(U_1 = 0.4\)

\(c_2 = 4, p_2 = 15\), \(U_2 = 0.27\)

\(c_3 = 10, p_3 = 35\), \(U_3 = 0.28\)

\[r_3^0 = 18\]
Intuitions of Exact Schedulability Test

- Obviously, the response time of task 3 should be larger than or equal to
  \[ c_1 + c_2 + c_3 \]
  \[ r_3^0 = \sum_{j=1}^{3} c_j = c_1 + c_2 + c_3 = 4 + 4 + 10 = 18 \]

- The high priority jobs released before \( r_3^0 \), should lengthen the response time of task 3
  \[ r_3^1 = c_3 + \sum_{j=1}^{2} \left( \frac{r_3^0}{p_j} \right) c_j = 10 + \left\lfloor \frac{18}{10} \right\rfloor 4 + \left\lfloor \frac{18}{15} \right\rfloor 4 = 26 \]
Intuitions of Exact Schedulability Test

(c_1 = 4, p_1 = 10), U_1 = 0.4

(c_2 = 4, p_2 = 15), U_2 = 0.27

(c_3 = 10, p_3 = 35), U_3 = 0.28

r_3 \textsuperscript{1} = 26
Intuitions of Exact Schedulability Test

- Keep doing this until either $r_3^k$ no longer increases or $r_3^k > p_3$

\[
\begin{align*}
    r_3^2 &= c_3 + \sum_{j=1}^{2} \left[ \frac{r_3^1}{p_j} \right] c_j = 10 + \left[ \frac{26}{10} \right] 4 + \left[ \frac{26}{15} \right] 4 = 30 \\
    r_3^3 &= c_3 + \sum_{j=1}^{2} \left[ \frac{r_3^2}{p_j} \right] c_j = 10 + \left[ \frac{30}{10} \right] 4 + \left[ \frac{30}{15} \right] 4 = 30 \quad \text{Done!}
\end{align*}
\]
Intuitions of Exact Schedulability Test

(c_1 = 4, p_1 = 10), U_1 = 0.4

(c_2 = 4, p_2 = 15), U_2 = 0.27

(c_3 = 10, p_3 = 35), U_3 = 0.28

\[ r_3^2 = r_3^3 = 30 \]
Intuition for the Exact Schedulability Test

• Suppose we have n tasks, and we pick a task, say i, to see if it is schedulable.
• We initialize the testing by assuming all the higher priority tasks from 1 to i-1 will only preempt task i once.
• Hence, the initially presumed finishing time for task i is just the sum of C_1 to C_i, which we call r^0.
• We now check the actual arrival of higher priority tasks within the duration r^0 and then presume that it will be all the preemption task i will experience. So we compute r^1 under this assumption.
• We will repeat this process until one of the two conditions occur:
  • 1. The r^n eventually exceeds the deadline of task i. In this case we terminate the iteration process and conclude that task i is not schedulable.
  • 2. The series r^n converges to a fixed point (i.e., it stops increasing). If this fixed point is less than or equal to the deadline, then the task is schedulable and we terminate the schedulability test.
Assumptions under UB & Exact Analysis

• Both the Utilization Bound and the Exact schedulability test make the following assumptions:

  • All the tasks are periodic
  • Tasks are scheduled according to RMS
  • All tasks are independent and do not share resources (data)
  • Tasks do not self-suspend during their execution
  • Scheduler overhead (context-switch) is negligible
Recap

- Schedulability analysis
  - Determine whether a given real-time taskset is schedulable or not
- L&L least upper bound
  - Sufficient condition
- Exact analysis
  - Critical instance theorem
  - Recursive process to determine Schedulability of each task.
Overview

- Today: aperiodic task scheduling.

To learn more on real-time scheduling:
- see chapter 5 on “Hard Real-Time Computing Systems” book from G. Buttazzo (useful chapters are in the Lab!): aperiodic tasks, background service and polling server.
Aperiodic tasks: concepts and definitions

Aperiodic task: runs at irregular intervals.

Aperiodic task’s deadline can be

- hard, with the following pre-conditions:
  - a lower bound exists on minimum inter-arrival time
  - an upper bound exists on worst-case computing time for the aperiodic

- soft ➞ it does not need pre-conditions.
  - no special requirement on inter-arrival time, typical assumption: exponential inter-arrival time (Poisson Process)
  - no special requirement on worst case execution, typical assumption: exponential execution time
The Fundamental Idea for handling aperiodic tasks: Server

- Rate monotonic scheduling is a periodic framework. To handle aperiodics, we must convert the aperiodic event service into a periodic framework.

- Except in the case of using interrupt handler to serve aperiodics, the basic idea is to periodically allocate CPU cycles to each stream of aperiodic requests. This CPU allocation is called “aperiodic server”:
  - Polling server
  - Sporadic server
Types of Aperiodic Requests

- The jobs of an aperiodic task have random release times
  - Soft aperiodic tasks:
    - random arrivals such as a Poisson distribution:
    - the execution time can also be random such as exponential distribution
    - typically it models users’ requests.
  - Aperiodic tasks with hard deadline:
    - there is a minimal separation between two consecutive arrivals
    - there is a worst-case execution time bound
    - models emergency requests such as the warning of engine overheat

![Diagram of Task Set with Aperiodic and Periodic Tasks]
Interrupt Handling or Background Service

- One way to serve aperiodic requests is handle them right at the interrupt handler.
  - This gives the best response time but can greatly impact the hard real-time periodic tasks causing deadline misses.
  - Use it as last resort only say pending power failure exception handling

- Another simple method is to give background class priority to aperiodic requests. This works as well but the response time is not too good. For example:
  - Assign Priority levels 256 to 50 for periodic tasks
  - Assign Priority levels 1 to 49 for aperiodic tasks
Interrupt Handling, Background, Polling

**Interrupt Handling**

- T1 = (3,1)
- T2 = (5,2)

**Deadline miss**

**Background**

- T1 = (3,1)
- T2 = (5,2)

**Polling**

- S = (2.5,0.5)
- T1 = (3,1)
- T1 = (5,2)
Polling Server - 1

- The simplest form of integrated aperiodic and periodic service is polling server.
  - For each aperiodic task, we assign a periodic service with budget \( e_s \) and period \( p_s \). This creates a server \((e_s, p_s)\)
  - The aperiodic requests are buffered into a queue
  - When polling server starts, 
    - Resumes the existing job if it was suspended in last cycle.
    - it checks the queue.
  - The polling server runs until 
    - All the requests are served
    - Or suspends itself when the budget is exhausted.

- Remark: a small improvement is to run the tasks in background priority instead of suspend. This background mode can be applied to any aperiodic server.
- If an aperiodic task arrives after the beginning of the server period, the task has to wait for the beginning of next period before being served.
Polling - 2

- A polling server behaves just like a periodic task and thus the schedulability of periodic tasks is easy to analyze. For example, if we use L&L bound,

\[
\sum_{i=1}^{n} \frac{e_i}{p_i} + \frac{e_s}{p_s} \leq (n + 1)\left(2^{1/(n+1)} - 1\right)
\]
Polling - 3

- Main attributes of a Polling Server:
  - it buffers all aperiodic requests in a FIFO queue
  - serve the buffered requests periodically with
    - a budget C
    - and a period P
    - the priority is assigned according to the server period (higher rate, higher priority just like periodic tasks)

- The utilization of a polling server is simply \( U = \frac{C}{P} \)

- NOTE: each time, the server will keep serving buffered requests until either
  - all the buffered requests are serviced (unused budget, if any, will be discarded),
  - or the budget \( C \) runs out. In this case, the server suspends until the beginning of next period with a new \( C \) budget again.
Example with a Polling Server

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Server

$C_s = 2$
$T_s = 5$

Example of a Polling Server scheduled by RM.
Performance of a Polling Server

Polling Server with $P=100$

Service delay of a polling server is, on average, roughly half of the server period.

- higher polling rate (shorter server period) will give better response time.
- low polling rate will have lower scheduling overhead.
Using Interrupt Handler

- Handle aperiodic requests within interrupt handler gives the best performance, since interrupt handlers run at priority higher than applications.

- Precisely for the same reason, a larger amount of such interrupts would cause deadlines of periodic tasks to be missed.

- It is a solution with serious side effects. Use it ONLY as a last resort for short fuse hard deadline aperiodic requests such as power failure warning.
The Sporadic Server (SS) differs from Polling Server in the way it replenishes its capacity. Whereas Polling periodically replenishes its capacity at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

We will see that Sporadic Server can be treated as if it is a periodic task too. However, SS has better response time than Polling server.

What is the main advantage of SS?

If Sporadic Server has the highest priority in the system, it can provide a service delay almost equivalent to an interrupt handler but without causing the deadline miss of other tasks!!!
A Sporadic Server with priority $Prio_s$ is said to be active when it is executing or another task with priority $Prio_t \geq Prio_s$ is executing. Hence, the server remains active even when it is preempted by a higher priority task.

If the server is not active, it is said to be idle.

**Replenishment Time (RT):** it is set as soon as SS becomes active and the server capacity $C_s > 0$. Let $T_A$ be such a time. The value of RT is set equal to $T_A$ plus the server period ($RT = T_A + p_s$).

**Replenishment Amount (RA):** The RA to be done at time RT is computed when SS becomes idle or the server capacity $C_s$ has been exhausted. Let $T_I$ be such a time. The value of RA is set equal to the capacity consumed within the interval $[T_A, T_I]$. 
**Sporadic Server - 3**

- Example of a medium-priority Sporadic Server.

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$T_s$</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>$T_2$</td>
<td>4</td>
<td>15</td>
</tr>
</tbody>
</table>
Sporadic Server - 4

- Example of a high-priority Sporadic Server.

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_s$</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>$T_1$</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$T_2$</td>
<td>4</td>
<td>15</td>
</tr>
</tbody>
</table>
The Sporadic Server can defer its execution and preserve its budget even if no aperiodic requests are pending. This allows SS to achieve better response time compared to Polling Server.

What about the schedulability analysis in the presence of Sporadic Server?

A periodic task set that is schedulable with a task $T_i$ is also schedulable if $T_i$ is replaced by a Sporadic Server with the same period and execution time.

⇒ In other words, Sporadic Server behaves like a regular periodic task, so nothing changes when you check the schedulability of the task set.