Lecture #18

Introduction To Scheduling

18-348 Embedded System Engineering
Philip Koopman
Wednesday, 23-Mar-2016

Sewer And Pipe Inspection Camera

Where Are We Now?

◆ Where we’ve been:
  • Interrupts
  • Context switching and response time analysis
  • Concurrency

◆ Where we’re going today:
  • Scheduling

◆ Where we’re going next:
  • Analog and other I/O
  • System booting, control, safety, …
  • In-class Test #2, Wed 20-April-2016
  • Final project due finals week. No final exam.

Preview

◆ What’s Real Time?

◆ Scheduling – will everything meet its deadline?
  • Schedulability
  • 5 key Assumptions

◆ Application of scheduling
  • Static multi-rate systems
  • Dynamic priority scheduling: Earliest Deadline First (EDF) and Least Laxity
  • Static priority preemptive systems (Rate Monotonic Scheduling)

◆ Related topics
  • Blocking time
  • Sporadic tasks
Real Time Scheduling Overview

- Hard real time systems have a deadline for each periodic task
  - With an RTOS, the highest priority active task runs while others wait
  - System fault occurs every time a task misses a deadline
  - **Mathematical analysis is accepted practice for ensuring deadlines are met**
    - We’ll build up to Rate Monotonic Analysis in this lecture

Real Time Definitions

- **Reactive:**
  - **Computations occur in response to external events**
    - Periodic events (e.g., rotating machinery and control loops)
      - Most embedded computation is periodic
    - Aperiodic events (e.g., button closures)
      - Often they can be “faked” as periodic (e.g., sample buttons at 10 Hz)

- **Real Time**
  - Real time means that correctness of result depends on both functional correctness and time that the result is delivered
  - Too slow is usually a problem
  - Too fast sometimes is a problem
Flavors Of Real Time

- **Soft real time**
  - Utility degrades with distance from deadline

- **Hard real time**
  - System fails if deadline window is missed

- **Firm real time**
  - Result has no utility outside deadline window, but system can withstand a few missed results

“Real Time” != “Really Fast”

- **“Real Time” != “Really Fast”**
  - It means not too fast and not too slow
  - Often the “not too slow” part is more difficult, but it’s not the only issue
  - Also, a whole lot faster than you need to go can be wasteful overkill

  • Often, ability to be consistently on time is more important than “fast”

- **Consider what happens when a CPU goes obsolete**
  - Is it OK to write a software simulator on a really fast newer CPU?
    - Will timing be fast enough?
    - Will it be too fast?
    - Will it vary more than the old CPU?
  - What do designers actually do about this?
Types of Real-Time Scheduling

- **Dynamic vs. Static**
  - Dynamic schedule computed at run-time based on tasks really executing
  - Static schedule done at compile time for all possible tasks
- **Preemptive permits one task to preempt another one of lower priority**

![Figure 11.1: Taxonomy of real-time scheduling algorithms.](Kopetz)

Schedulability

- **NP-hard if there are any resource dependencies at all**
  - So, the trick is to put cheaply computed bounds/heuristics in place
    - Prove it definitely can’t be scheduled
    - Find a schedule if it is easy to do so
    - Punt if you’re in the middle somewhere

If the sufficient schedulability test is positive, these tasks are definitely schedulable

If the necessary schedulability test is negative, these tasks are definitely not schedulable

![Figure 11.2: Necessary and sufficient schedulability test.](Kopetz)
Periodic Tasks

◆ “Time-triggered” (periodic) tasks are common in embedded systems
  • Often via control loops or rotating machinery

◆ Components to periodic tasks
  • Period (e.g., 50 msec)
  • Offset past period (e.g., 3 msec offset/50 msec period -> 53, 103, 153, 203)
  • Jitter is random “noise” in task release time (not oscillator drift)
  • Release time is when task has its “ready to run” flag set
  • Release time_\text{n} = (n*period) + offset + jitter ; assuming perfect time precision

Scheduling Parameters

◆ Set of tasks \{T_i\}
  • Periods p_i
  • Deadline d_i
    (completion deadline after task is queued)
  • Execution time c_i
    (amount of CPU time to complete)
  • Worst case latency to complete execution W_i
    – This is something we solve for, it’s not a given

◆ Handy values:
  • Laxity l_i = d_i - c_i
    (amount of slack time before Ti must begin execution)
  • Utilization factor \mu_i = c_i/p_i
    (portion of CPU used)
**Major Assumptions**

- Five assumptions are the starting point for this area:
  1. Tasks \( \{T_i\} \) are periodic, with hard deadlines and no jitter
    - Period is \( P_i \)
  2. Tasks are completely independent
    - \( B=0 \); Zero blocking time; no use of a mutex; interrupts never masked
  3. Deadline = period
    - \( P_i = D_i \)
  4. Computation time is known (use worst case)
    - \( C_i \) is always the same for each execution of the task
  5. Context switching is free (zero cost)
    - Executive takes zero overhead, and task switching has zero latency

- These assumptions are often not realistic
  - But sometimes they are close enough in practice
  - Significantly relaxing these assumptions quickly becomes a grad school topic
    - We’re going to show you the common special cases that are “easy” to use

**Easy Schedulability Test**

- System is schedulable (i.e., it “works”) if for all \( i \), \( W_i \leq D_i \)
  - In other words, all tasks complete execution before their deadline

- \( \mu \) is processor utilization (fraction of time busy) must be less than 1

\[
\mu = \sum \frac{C_i}{P_i} \leq 1
\]

- “You can’t use more that 100% of available CPU power!”

- **This is necessary, but not sufficient**
  - Sometimes even very low percent of CPU power used is still unschedulable
  - e.g., if blocking time exceeds shortest deadline, impossible to schedule system
  - e.g., several short-deadline tasks all want service at exactly the same time, but rest of time system is idle
Remember this? Multi-Rate Round Robin Approach

◆ Simple brute force version
  • Put some tasks multiple times in single round-robin list
  • But gets tedious with wide range in rates

◆ More flexible version
  • For each PCB keep:
    – Pointer to task to be executed
    – Period (number of times main loop is executed for each time task is executed)
      i.e., execute this task every kth time through main loop.
    – Current count – counts down from Period to zero, when zero execute task

typedef void (*pt2Function)(void);

struct PCB_struct
{
  pt2Function Taskptr;   // pointer to task code
  uint8       Period;    // execute every kth time
  uint8       TimeLeft;  // starts at k, counts down
  uint8       ReadyToRun; // flag used later
};

PCB_struct PCB[NTASKS];  // array of PCBs

Remember this?

Time-Based Prioritized Cooperative Tasking

◆ Assume timer_ticks is number of TCNT overflows recorded by ISR

struct PCB_struct
{
  pt2Function Taskptr;   // pointer to task code
  uint8       Period;    // Time between runs
  uint8       NextTime;  // next time this task should run
};
...
... init PCB structures etc. ...

for(;;)
{
  for (i = 0; i < NTASKS; i++)
  {
    if (PCB[i].NextTime < timer_ticks)
      PCB[i].NextTime += PCB[i].Period; // set next run time
      // note - NOT timer_ticks + Period !!
      PCB[i].Taskptr();
    break; // exit loop and start again at task 0
  }
}

◆ This executes tasks in a particular order based on period and task #
  • But, there is no guarantee that you will meet your deadlines in the general case!
Static Multi-Rate Periodic Schedule

- Assume non-preemptive system with 5 Restrictions:
  1. Tasks \( \{T_i\} \) are perfectly periodic
  2. \( B=0 \)
  3. \( P_i = D_i \)
  4. Worst case \( C_i \)
  5. Context switching is free

- Consider least common multiple of periods \( p_i \)
  - This considers all possible cases of period phase differences
  - Worst case is time that is LCM of all periods
    - E.g., \( \text{LCM}(5,10,35) = 5 \times 2 \times 7 = 70 \)
  - If you can figure out (somehow) how to schedule statically this, you win
    - Program in a static schedule that runs tasks in exactly that order at those times
    - Schedule repeats every LCM time period (e.g., every 70 msec for LCM=10)
    - This is a long-running computational problem for large task sets!

- Performance
  - Optimal if all tasks always run; can get up to 100% utilization \( (\mu = 1.00) \)
  - If it runs once, it should always work

Example Static Schedule – Hand Positioned Tasks

<table>
<thead>
<tr>
<th>Task #</th>
<th>Period ( (P_i) )</th>
<th>Compute ( (C_i) )</th>
<th>Start Time</th>
<th>Task #</th>
<th>( C_i )</th>
<th>Elapsed Time For ( T_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>T1</td>
<td>1</td>
<td>…</td>
</tr>
<tr>
<td>T2</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>T5</td>
<td>4</td>
<td>…</td>
</tr>
<tr>
<td>T3</td>
<td>15</td>
<td>2</td>
<td>5</td>
<td>T1</td>
<td>1</td>
<td>5-0=5</td>
</tr>
<tr>
<td>T4</td>
<td>20</td>
<td>3</td>
<td>6</td>
<td>T2</td>
<td>2</td>
<td>…</td>
</tr>
<tr>
<td>T5</td>
<td>25</td>
<td>4</td>
<td>8</td>
<td>T3</td>
<td>2</td>
<td>…</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>T1</td>
<td>1</td>
<td>10-5=5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>T4</td>
<td>3</td>
<td>…</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>Idle</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>T1</td>
<td>1</td>
<td>15-10=5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>T2</td>
<td>2</td>
<td>16-6=10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>Idle</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>T1</td>
<td>1</td>
<td>20-15=5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21</td>
<td>Idle</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23</td>
<td>T3</td>
<td>2</td>
<td>23-8=15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>T1</td>
<td>1</td>
<td>25-20=5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26</td>
<td>T2</td>
<td>2</td>
<td>26-16=10</td>
</tr>
</tbody>
</table>

Ensuring schedulability requires hand-selecting the start time of every task (not the same as the previous scheduler code)!
Preemptive, Prioritized Schedulability

- To avoid missing deadlines, necessary for all the tasks to fit
  - Time to complete task $T_j$ is $W_j$
  - (i.e., we need to find out if this task set is “schedulable?”)

$$\forall j : W_j \leq P_j$$

- If true, we are schedulable; if false we aren’t
- Note that this is $W =$ time to complete task
  - It’s not $R =$ time to start execution of task (response time)
  - For cooperative scheduling, $W_i = R_i + C_i$
  - BUT, for preemptive scheduling $W$ can be longer because of additional preemptions

- In other words, schedulable if task completes before its period
  - Always true if time to complete task $T_j$ doesn’t exceed period
  - True because we assumed that $P_i = D_i$

What’s Latency For Preemptive Tasks?

- For the same 5 assumptions
  - And prioritized tasks (static priority – priority never changes)
    - Note that equation includes execution time of task, not just response time

$$W_{m,0} = B + C_0$$

$$W_{m,j+1} = B + \sum_{j=0}^{j=m} \left( \left\lfloor \frac{W_{m,i}}{P_j} \right\rfloor \cdot C_j \right)$$

- Note that in this math we are including the C term for task $m$ in the summation
- Highest priority task has only blocking time $B$ as latency
- Start the recursion with task 0, which could always execute first
- Schedulable if:

$$\forall j : W_j \leq P_j$$

- This math is complex, and easy to get wrong
  - Is there an easier way to make sure we can’t mess this up?
Remember the Major Assumptions

- Five assumptions throughout this lecture
  1. Tasks \(T_i\) are perfectly periodic
  2. \(B=0\)
  3. \(P_i = D_i\)
  4. Worst case \(C_i\)
  5. Context switching is free

EDF: Earliest Deadline First

- Assume a preemptive system with *dynamic priorities*, and
  { same 5 restrictions }

- Scheduling policy:
  - Always execute the task with the *nearest deadline*
    - Priority changes on the fly!
    - Results in more complex run-time scheduler logic

- Performance
  - Optimal for uniprocessor (supports up to 100% of CPU usage in all situations)
    - If it can be scheduled – but no guarantee that can happen!
    - Special case where it works is very similar to case where Rate Monotonic can be used:
      » Each task period must equal task deadline
      » But, still pay run-time overhead for dynamic priorities
  - If you’re overloaded, ensures that a lot of tasks don’t complete
    - Gives everyone a chance to fail at the expense of the later tasks
Least Laxity

◆ Assume a preemptive system with **dynamic priorities**, and
{ **same 5 restrictions** }

◆ Scheduling policy:
- Always execute the task with the
  **smallest laxity** \( l_i = d_i - c_i \)

◆ Performance:
- Optimal for **uniprocessor** (supports up to 100% of CPU usage in all situations)
  - Similar in properties to EDF
  - If it can be scheduled – but no guarantee that can happen!
- A little more general than EDF for multiprocessors
  - Takes into account that slack time is more meaningful than deadline for tasks of
    mixed computing sizes
- Probably more graceful degradations
  - Laxity measure permits dumping tasks that are hopeless causes

EDF/Least Laxity Tradeoffs

◆ Pro:
- If it works, it can get 100% efficiency (on a uniprocessor)
- Does not restrict task periods
- Special case works if, for each task, Period = Deadline

◆ Con:
- It is not always feasible to prove that it will work in all cases
  - And having it work for a while doesn’t mean it will always work
- Requires dynamic prioritization
- EDF has bad behavior for overload situations (LL is better)
- The laxity time hack for global priority has limits
  - May take too many bits to achieve fine-grain temporal ordering
  - May take too many bits to achieve a long enough time horizon

◆ Recommendation:
- Avoid EDF/LL if possible
  - Because you don’t know if it will really work in the general case!
  - And the special case doesn’t buy you much, but comes at expense of dynamic
    priorities
Remember the Major Assumptions

- Five assumptions throughout this lecture
  1. Tasks \( T_i \) are perfectly periodic
  2. \( B=0 \)
  3. \( P_i = D_i \)
  4. Worst case \( C_i \)
  5. Context switching is free

- Problems with previous approaches
  - Static scheduling – can be difficult to find a schedule that works
  - EDF & LL – run-time overhead of dynamic priorities
  - Wanted: an easy rule for scheduling with:
    - Static priorities
    - Guaranteed schedulability

Rate Monotonic Scheduling

1. Sort tasks by period (i.e., by “rate”)
2. Highest priority goes to task with shortest period (fastest rate)
   - Tie breaking can be done by shortest execution time at same period
3. Use prioritized preemptive scheduler
   - Of all ready to run tasks, task with fastest rate gets to run

- Static priority
  - Priorities are assigned to tasks at design time; priorities don’t change at run time

- Preemptive
  - When a high priority task becomes ready to run, it preempts lower priority tasks
  - This means that ISRs have to be so short and infrequent that they don’t matter

- Variation: Deadline Monotonic
  - Use \( \min(\text{period}, \text{deadline}) \) to assign priority rather than just period
  - Works the same way, but handles tasks with deadlines shorter than their period
Rate Monotonic Scheduling (RMS)

- Assume a preemptive system with static priorities, \( N \) tasks, and
  \{ same 5 restrictions \} +

\[
\mu = \sum \frac{c_i}{p_i} \leq N(\sqrt[3]{2} - 1) ; \mu \leq \ln(2) \approx 0.693 \text{ for large } N
\]

("CPU load less than about 70\%")

- Why not 100\%?
  - Two tasks with slightly different periods can drift in and out of phase
  - At just the wrong phase difference, there may not be time to meet deadlines

- Performance:
  - Provides a guarantee for schedulability with CPU load of \(~70\%\)
    - Even with arbitrarily selected task periods
    - Can do better if you know about periods & offsets
  - BUT – if you load CPU more than 69.3\%, you might miss deadlines!

Example of a Missed Deadline at 79\% CPU Load

- Task 4 misses deadline
  - This is the worst case launch time scenario

- Missed deadlines can be difficult to find in system testing
  - 5 time units per task is worst case
    - Average case is often a bit lighter load
  - Tasks only launch all at same time once every 224,808 time units
    \( \text{LCM}(19,24,29,34) = 224,808 \)
    \( \text{(LCM = Least Common Multiple)} \)
Harmonic RMS

- In most real systems, people don’t want to sacrifice 30% of CPU
  - Instead, use harmonic RMS

- Make all periods harmonic multiples
  - P_i is evenly divisible by all shorter P_j
  - This period set is harmonic: {5, 10, 50, 100}
    - 10 = 5*2; 50 = 10*5; 100 = 50*2; 100 = 10*5*2
  - This period set is not harmonic: {3, 5, 7, 11, 13}
    - 5 = 3 * 1.67 (non-integer), etc.

- If all periods are harmonic, works for CPU load of 100%
  - Harmonic periods can’t drift in and out of phase – avoids worst case situation

\[
\mu = \sum \frac{C_i}{P_i} \leq 1 \land \forall p_j < p_i \{ p_j \text{ evenly divides } p_i \}
\]

Practical Harmonic Deadline Monotonic Scheduling

- This is what you should do in most smaller embedded control systems
  - Assumes you need a preemptive scheduler

- Use Min(period, deadline) as the scheduling logical “period”
  - Ensures that deadline will be met even if shorter than period
  - But, set aside resources just as if tasks really were repeating at that period
  - This is the part that makes it “deadline” monotonic

- Use harmonic multiples of logical period
  - Every shorter period is a factor of every longer period (e.g., 1, 10, 100, 1000)
  - Avoids worst case of slightly out-of-phase periods that all clump together at just the wrong time
  - Speed up some tasks if needed to get harmonic multiples
    - E.g., \{1, 5, 11, 20\} => \{1, 5, 10, 20\}
    - Results in lower CPU requirement even though some tasks run faster!

- Watch out for blocking!
### Example Deadline Monotonic Schedule

<table>
<thead>
<tr>
<th>Task #</th>
<th>Period ($P_i$)</th>
<th>Deadline ($D_i$)</th>
<th>Compute ($C_i$)</th>
<th>Priority</th>
<th>$\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1/5 = 0.200</td>
</tr>
<tr>
<td>T2</td>
<td>16</td>
<td>23</td>
<td>2</td>
<td>2</td>
<td>2/6 = 0.333</td>
</tr>
<tr>
<td>T3</td>
<td>30</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>2/16 = 0.125</td>
</tr>
<tr>
<td>T4</td>
<td>60</td>
<td>60</td>
<td>3</td>
<td>4</td>
<td>4/30 = 0.133</td>
</tr>
<tr>
<td>T5</td>
<td>60</td>
<td>30</td>
<td>4</td>
<td>5</td>
<td>3/60 = 0.05</td>
</tr>
</tbody>
</table>

$$\mu = \sum \frac{C_i}{P_i} \leq N(\sqrt[\text{N}]{2} - 1) \quad ; \quad N = 5$$

$$\mu = 0.841 \quad (not \leq) \quad 0.743$$

Not Schedulable! (might be OK with fancy math)

### Example Harmonic Deadline Monotonic Schedule

<table>
<thead>
<tr>
<th>Task #</th>
<th>Period ($P_i$)</th>
<th>Deadline ($D_i$)</th>
<th>Compute ($C_i$)</th>
<th>Priority</th>
<th>$\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>5</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1/5 = 0.200</td>
</tr>
<tr>
<td>T2</td>
<td>15</td>
<td>23</td>
<td>2</td>
<td>2</td>
<td>2/5 = 0.400</td>
</tr>
<tr>
<td>T3</td>
<td>30</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2/15 = 0.133</td>
</tr>
<tr>
<td>T4</td>
<td>60</td>
<td>60</td>
<td>3</td>
<td>4</td>
<td>4/30 = 0.133</td>
</tr>
<tr>
<td>T5</td>
<td>60</td>
<td>30</td>
<td>4</td>
<td>5</td>
<td>3/60 = 0.05</td>
</tr>
</tbody>
</table>

$$\mu = \sum \frac{C_i}{P_i} \leq 1 \quad ; \quad \text{Harmonic periods \{5, 15, 30, 60\}}$$

$$\mu = 0.916 \quad \leq \quad 1$$

Schedulable, even though usage is higher!
Handling Non-Zero Blocking

- **Rate monotonic, but task blocking can occur**
  - $B_k$ is time task $k$ can be blocked (e.g., interrupts masked by lower priority task)
  - For highest priority task
    - Can ignore lower priority tasks, because we are preemptive
    - But, need to handle blocking time (possibly caused by lower priority task)
    \[
    \mu_1 = \left( \frac{c_1}{p_1} \right) + \frac{B_1}{p_1} \leq 1(\sqrt{2} - 1)
    \]
  - For 2nd highest priority task
    - Can ignore lower priority tasks, because we are preemptive
    - Have to account for highest priority task preempting us
    - Need to handle blocking time
      - Possibly caused by lower priority task
      - But, can’t be caused by higher priority task (since that preempts us anyway)
      - Does this sound a lot like the reasoning behind ISR scheduling???
    \[
    \mu_2 = \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \frac{B_2}{p_2} \leq 2(\sqrt{2} - 1)
    \]

Rate Monotonic With Blocking

- **Rate monotonic, but task blocking can occur**
  - $B_k$ is blocking time of task $k$ (time spent stalled waiting for resources)
  - Worst case blocking time for each task counts as CPU time for scheduling
  - Note that B includes all interrupt masking (ISRs and tasks waiting for CLI)
  - Harmonic periods make right hand side 100%, as before
  - Need on a per-task basis because blocking time can be different for each task
  - Performance:
    - In worst case, time waiting while blocked is counted as burning additional CPU or network time
    - This is yet another reason to use skinny ISRs!
    - If low priority task gets a mutex needed by a hi prio task, it extends B!
    - If RTOS takes a while to change tasks, that counts as blocking time too

\[ \forall k; \mu_k = \sum_{i \leq k} \mu_i = \sum_{i \leq k} \left( \frac{c_i}{p_i} \right) + \frac{B_k}{p_k} \leq k(\sqrt{2} - 1) \approx 0.7 \text{ for large } k \]  

[Sha et al. 1991]
Applied Deadline Monotonic With Blocking

- Use \( \min(\text{period, deadline}) \) for each task as logical period
  - Use harmonic logical periods
  - Assign tasks by priority
  - Otherwise, same as for deadline monotonic

- For each task,
  \[
  \mu_1 = \left( \frac{c_1}{p_1} \right) + \frac{B_1}{p_1} \leq 1 \\
  \mu_2 = \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \frac{B_2}{p_2} \leq 1 \\
  \mu_3 = \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \left( \frac{c_3}{p_3} \right) + \frac{B_3}{p_3} \leq 1 \\
  \forall k; \mu_k = \sum_{i \in k} \mu_i = \sum_{i \in k} \left( \frac{c_i}{p_i} \right) + \frac{B_k}{p_k} \leq 1 ; \text{ for harmonic periods}
  \]

But Wait, There’s More

- WHAT IF:
  1. Tasks \( \{T_i\} \) are NOT periodic
     - Use maximum fastest inter-arrival time
  2. Tasks are NOT completely independent
     - Worry about dependencies (another lecture)
  3. Deadline NOT = period
     - Use Deadline monotonic
  4. Worst case computation time \( c_i \) isn’t known
     - Use worst case computation time, if known
     - Build or buy a tool to help determine Worst Case Execution Time (WCET)
     - Turn off caches and otherwise reduce variability in execution time
  5. Context switching is free (zero cost)
     - Gets messy depending on assumptions
     - Might have to include scheduler as task
     - Almost always need to account for blocking time \( B \)
## Review

- **Real time definitions**
  - Hard, firm, soft

- **Scheduling – will everything meet its deadline?**
  - \( \mu \leq 1 \)
  - All \( W_i \leq P_i \)

- **Application of scheduling**
  - Static multi-rate systems
  - Rate Monotonic Scheduling
    - \( \mu \leq 1 \) if harmonic periods; else more like 70%
    - Works by assigning priorities based on periods (fastest tasks get highest prio)

- **Related topics**
  - Earliest Deadline First (EDF) and Least Laxity
  - Blocking
  - Sporadic server

---

### Five Standard Assumptions

*memorize them in exactly these words – notes sheet too*:

1. Tasks \( \{T_i\} \) are perfectly periodic
2. \( B = 0 \)
3. \( P_i = D_i \)
4. Worst case \( C_i \)
5. Context switching is free

### Statically prioritized task completion times:

\[
W_{m,0} = C_0
\]

\[
W_{m,i+1} = B + \sum_{j=0}^{i=m} \left( \frac{W_{m,j}}{P_j} + 1 \right) C_j
\]
Review

- Schedulability bound for Rate Monotonic with Blocking

\[
\begin{align*}
\mu_1 &= \left( \frac{c_1}{p_1} \right) + \frac{B_1}{p_1} \leq 1 \\
\mu_2 &= \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \frac{B_2}{p_2} \leq 1 \\
\mu_3 &= \left( \frac{c_1}{p_1} \right) + \left( \frac{c_2}{p_2} \right) + \left( \frac{c_3}{p_3} \right) + \frac{B_3}{p_3} \leq 1 \\
\forall k; \mu_k &= \sum_{i < k} \frac{c_i}{p_i} + \frac{B_k}{p_k} \leq 1 \text{; for harmonic periods}
\end{align*}
\]