SFWR ENG 4AA4

Kemal Ahmed

Fall 2015

Dr. Down

Note: information from the pre-requisite, [SFWR ENG 3DX4](https://drive.google.com/open?id=0BxW61uJyyN8TUjN2X0dwbVBkTVk) will not be included in this summary (although corrections will be).

Contents

[Real-Time Systems 1](#_Toc432876475)

[Classifications 1](#_Toc432876476)

[Task optimization 1](#_Toc432876477)

[Types of Scheduling 1](#_Toc432876478)

[Static 1](#_Toc432876479)

[FIFO 1](#_Toc432876480)

[Dynamic 1](#_Toc432876481)

[Multiprocessor 1](#_Toc432876482)

[Task Interactions 1](#_Toc432876483)

[Sporadic Server 1](#_Toc432876484)

[Clocks 1](#_Toc432876485)

[PID Control 1](#_Toc432876486)

[Designing a PID Controller 1](#_Toc432876487)

[Jitter 1](#_Toc432876488)

# Real-Time Systems

## Classifications

What happens upon failure to meet deadlines:

* **Soft**: performance is degraded but not destroyed
* **Firm**: a few times will simply degrade performance, but after may lead to system failure
* **Hard**: complete and catastrophic system failure
  + **Safety Critical**: may cause injury / death (a type of hard)

**Forward difference method**: derivatives using 

**Backwards Difference method**: derivatives using 

**Controller** [C(s)]:

**Input** [E(s)]:

**Output** [U(s)]:



# Task optimization

**Task** [T]: 

**Period** [p]: time between tasks are repeatedly released

**Release time** [r]: time it takes to release task

**Execution time** [e]: slowest time task could take to be completed (but assume the tasks will take this long no matter what)

**Deadline** [d]: when task needs to be completed

**Number of tasks** [n]:

**Processor Utilization** [U]: used as a priority level

If U > 1, nothing is feasible

If ri = 0 and pi = di, then write *Ti* = (*pi* , *ei* )

# Types of Scheduling

## Static

**Static Scheduling**:

* task’s priority is assigned before execution and does not change
* If a task misses its deadline, you mess up all the deadlines after it like an airport at Christmas

### FIFO

**First In First Out (FIFO)**:

* Could cause problems for tasks whose execution time is significantly shorter than the rest when there are deadlines
  + E.g. T1 = (100, 3); T2 = (2, 1)
* A.K.A. **First Come, First Served (FCFS)**

**Schedule**: the order in which tasks will be executed

**Hyperperiod** [H]: the entire length of a cycle, least common multiple

**Harmonic**: every task period evenly divides every longer period

**Frame Size** [f]:

* The best way for computers to segment the schedule in a way that it verify that the appropriate tasks have been executed
* Process: try each see which is the largest frame size the follows the below constraints from 1 to emax.
* Constraints:

1. 
2. H % f = 0
3. 2f − gcd(pi , f ) ≤ di

**Least Compute Time (LCT)**: tasks with smallest execution times executed first

* Think *greedy*
* Works poorly; worse than RR

**Rate Monotonic (RM)**: shorter period, higher priority

* Think: tasks requiring frequent attention should have higher priority
* If harmonic, feasible as long as U ≤ 1
* If non-harmonic, guaranteed feasible if 
  + If the equation fails, it still might be, so draw the whole thing to be safe.

## Dynamic

**Pre-empting**: splitting a task up into multiple mini tasks. Also, if a task misses its deadline, halt the task at the deadline

The only two optimal dynamic priorities are:

* **Earliest Deadline First (EDF)**:
  + more flexible, better U
  + If deadlines < periods, still optimal, but determining feasibility is NP-hard
  + Always feasible if U ≤ 1
* **Least Slack Theorem (LST)**: not as popular as EDF

## Multiprocessor

Once you have multiple processors, neither EDF nor RM are guaranteed to work.

Look into first-fit algorithms

# Task Interactions

**Suspended**: active choice, of access prevention until algorithm allows it to

**Blocked**: as a result of waiting for a resource to be free

How to do the timing diagrams with locks:

* S1 = lock(S1)
* S1^ = unlock(S1)

**One-shot Tasks**: non-periodic tasks

**Critical Section**: when a task tries to acquire an already locked by another task resource

**Priority Inversion**: a method of avoiding deadlock by telling high priority tasks to share their resources with the lower priority tasks even when it’s not their turn

* Allocate time, where T1 has access to shared resource, so the time not allocated can be pre-empted
* Connect the pre-empted by T1 when T1 wants to access the resource
* Protect the resource with a semaphore
* You can make it so that tasks can use the resource even after they release the semaphore, but you risk overwriting in that time

**Priority Inheritance Protocol (PIP)**:

* Temporarily raise the priority of a task only if and when it actually blocks a higher priority task; on leaving the critical section, the task priority reverts to its original value
* Issues:
  + If only one shared resource, there’s only one possible schedule
  + If more than one resource blocking:
    - Blocking time may be excessively long
    - Deadlock may occur
  + If accessing multiple resources, you can only use them in the same order

**Priority Ceiling Protocol (PCP)**:

* Which tasks require which resources?
* Doesn’t give a shit about when they were released.
* **Priority Ceiling (PC)**: maximum priority that tasks will be given
  + For a current task, the PC doesn’t matter
* “The state of the art when resolving resource-contention issues”
* “Deadlock free for an arbitrary number of tasks with an arbitrary number of resources acted upon in an arbitrary way.”
* Main points:
  + No locked resources, so free access
  + If resource is locked by other tasks, S2 needs to have priority of T2 higher than the PC (S2). S1 is (suspended)
  + Priority higher than PC(S2)
  + If any task needs priority higher than the priority ceiling, it’s suspended
* When entering critical sections, check if any other tasks have resources

# Sporadic Server

**Execution Budget** [es]: periodic tasks aren’t flexible…

**Execution time** [ei]: …sporadic tasks are

**Deadline** [di]: absolute deadline

**Release Time** [ri]:

**Set of Sporadic Tasks** [θ]:

**Sporadic Task** [Si]:

* Non-periodic task
* (ri, ei, d­i)
* Typically interrupt-driven

**Rules** [ρ]: set of rules regulating a sporadic server

**Sporadic Server** [Φs]: (ps, es, θ, ρ)

**Periodic Task**: (ps, es) a type of sporadic server

Assume:

* Φs scheduled with Ti according to RM

We don’t use Kd because it looks at the derivative regardless of the size of the error function. If your error is a sine function with a small amplitude, Kd will only take the derivative into account and it will overcompensate.

**Open loop response**: plant with no control

**Ziegler-Nichols Tuning Rule**: a PID tuning rule

Look at the *open loop response*. It could have a longer rise time / overshoot than preferred.

1. Tangent to curve on upslope

High sample rate 🡪 lots of high frequency noise

**Effective Utilization** [δ]:

U = Uperiodic + δUsporadic

# Clocks

**Computer Clock** [C]:

Attributes:

* Correctness
* Bounded Drift
* Monotonicity
* Chronoscopicity

**Drift** [ρ]: rate of change of the clock value away from a perfect clock (each second)

There’s usually a reason why a clock drifts



(EPS):

**Monotonicity**: Clock will always have a consistent spacing and will only move in one order (forward / backwards)

SSL certs will fail signature if your clock is wrong as to ensure this

**Chronoscopicity** [γ]: changing drift

second derivative of stuff 

**Error bound** [ε]:

**Acceptance Test**: 

# PID Control

**Plant** [G(s)]: a transfer function, e.g. 

Remember this from 3DX4? Most of the stuff is still there, so refer to that.

Each of the K’s represent a different error or gain

4 types of controllers [P(s)]:

* **Proportional Controller (P)**,**(PC)**: 
* **Proportional Integral (PI)**:
* **Proportional Derivative (PD)**: 
* **Proportional Integral Derivative (PID)**:



## Designing a PID Controller

1. Obtain an open-loop response and determine what needs to be improved
2. Add Kp to improve the rise time
3. Add Kd to improve the overshoot
4. Add Ki to eliminate the steady-state error
5. Adjust each of Kp, Kd, and Ki until you obtain a desired overall response. You can always refer to the table below to find out which controller controls what characteristics.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **RESPONSE** | **RISE TIME** | **OVERSHOOT** | **SETTLING TIME** | **S-S ERROR** |
| **Kp** | Decrease | Increase | Small Change | Decrease |
| **Kd** | Small Change | Decrease | Decrease | No Change |
| **Ki** | Decrease | Increase | Increase | Eliminate |

**Ziegler-Nichols Tuning Rule**: for a plant with neither integrators nor dominant complex-conjugate pairs

**Noise frequency** [ωn]:

**Noise amplitude** [an]:

# Jitter

**Jitter**: a delay