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 2](#_Toc438553007)

[Classifications 2](#_Toc438553008)

[Difference Equations 2](#_Toc438553009)

[e.g.) 2](#_Toc438553010)

[Task optimization 3](#_Toc438553011)

[Types of Scheduling 3](#_Toc438553012)

[Static 3](#_Toc438553013)

[FIFO 3](#_Toc438553014)

[Dynamic 4](#_Toc438553015)

[Multiprocessor 5](#_Toc438553016)

[Task Interactions 5](#_Toc438553017)

[Sporadic Server 6](#_Toc438553018)

[Clocks 7](#_Toc438553019)

[Cristian’s Algorithm 8](#_Toc438553020)

[Berkeley 8](#_Toc438553021)

[PID Control 8](#_Toc438553022)

[Sampling 9](#_Toc438553023)

[Designing a PID Controller 10](#_Toc438553024)

[Jitter 10](#_Toc438553025)

[Fail 11](#_Toc438553026)

[Voting Schemes 11](#_Toc438553027)

[Information Redundancy 12](#_Toc438553028)

[Execution Time 12](#_Toc438553029)

[Natural Language Standards 12](#_Toc438553030)

# 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)
* **Controller** [C(s)]:
* **Input** [E(s)]:
* **Output** [U(s)]:
* 

## Difference Equations

**Forward difference method**: derivatives using 

**Backwards Difference method**: derivatives using 

### e.g.)

u[n] – u[n–1] = 48e[n] – 40e[n–1]



Group:



Equate each section to the values from the equation:











d = 48 – 40 **= 8**



# 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]:

**Feasible**: a schedule that meets all deadlines

**Optimal**: will produce a schedule every time if *feasible* schedule exists

**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
* A.K.A. **Fixed priority**

### 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)**

**Cyclic Executive**: frame-based scheduling

* When you allocate an amount of time where a task can execute
* Can have multiple executions of the same task
* Tasks might not even fill the full frame

**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

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

**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 that follows all 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

* static
* 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

**Dynamic**: each of the tasks’ priorities can change. *Think*: while for static priorities it is constantly re-evaluating which task has the highest priority, dynamic scheduling also re-evaluates the actual priorities, themselves.

**Shortest Completion Time (SCT)**: dynamic

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 a shared resource already locked by another task

**Deadlock**: think if you and I are at a table with one fork and one knife and you need both to eat, but you take the fork and I take the knife.

**Priority Inversion**: an issue when a lower priority task is using a resource a higher priority task wants

**Priority Inheritance Protocol (PIP)**: a method of dealing with *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
* 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
* 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
* 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)**: tasks entering a critical section can only access the blocked resource if it has a priority higher than the priority ceiling

* **Priority Ceiling (PC)**: maximum priority of all tasks ever going to access a given resource
* Only need to check PC when entering a critical section
* If any task needs priority higher than the priority ceiling of ALL of the semaphores currently locked, it’s suspended
* Main advantages:
  + No locked resources, so free access
  + “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.”

# Sporadic Server

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

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

**Deadline** [di]: absolute deadline

**Release Time** [ri]: delay before the task is released to be executed

**Set of Sporadic Tasks** [θ]:

**Sporadic Task** [Si]:

* Non-periodic task (a.k.a. aperiodic)
* (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

* no expectation of when it finishes, only that a new one is queued every period

Assume:

* Φs scheduled with Ti according to RM

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

**Effective Utilization** [δ]:

U = Uperiodic + δUsporadic

**Error bound** [ε]:

[i]: all previous

[k]: current index

**Slack** [ω]:

**Acceptance Test**: check of stuff

[ζ]: time spent processing task already



1. If ω(Sk , t) < 0, reject task
2. If ω(Sk , t) ≥ 0, need to check if already accepted sporadic tasks are adversely aﬀected, i.e. ω(Sj , t) − ek ≥ 0 holds for all Sj ∈ θ with dj ≥ dk .

The set θ is maintained dynamically.

# Clocks

**Computer Clock** [C]:

**Standard Clock** [Cs]: perfect clock; a.k.a. **real time**

Attributes:

* Correctness
* Bounded Drift
* Monotonicity
* Chronoscopicity

**Error Per Second (EPS)**: a bounded/maximum difference between the clock time and the real time

|C(t) − CS (t)| ≤ EPS

**Reset time** [r]: the time on a *standard clock* at which you set the clock to when you reset it

**Drift** [E]: rate of change of the clock value away from *real time* (each second)

* There’s usually a reason why a clock drifts

**Drift Bound** [ρ]: maximum drift



**Reset Error** [ε]: error between *real time* and *reset time*

**Total Error** [E]: E(t) = ε + drift\_since\_reset

drift\_since\_reset ≤ ρ(C(t) − r)

E(t) = ρ(C(t) − r) + ε ≤ EPS

C(t) − r ≤ (EPS − ε)/ρ

Real time interval: [C(t) − E(t), C(t) + E(t)]

**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]: maximum changing drift

second derivative of stuff 

## Cristian’s Algorithm

**Minimum Latency** [Tmin]:

**Request Send Time** [T0]:

**Request Receive Time** [T1]:

**Server Time** [Tserver]: time returned by the server



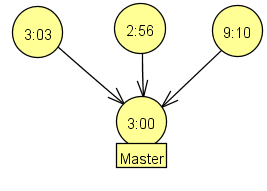
Accuracy is ±

**Round Trip Time (RTT)**:

## Berkeley

Not often used, but useful for learning

1. Elect 1 node to be the **master**, the one that runs the algorithm



1. Finds the average of the nodes. However, that’s probably going to find a value that isn’t near any of them.
2. Eliminate the outliers:
   1. Standard deviation: the more outliers, the harder to remove them, i.e. √ ½ ×Σ(xi –μ)2
   2. Median
   3. **Maximum deviation**: maximum clock drift × time since last synchronization; sometimes it’s good to use physical limitations as the minimum check to ensure accuracy

# 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. [More here](http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID).

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)**: 



**Dominant pole**: largest magnitude

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.

High sample rate 🡪 lots of high frequency noise

Finding a zero: Numerator = 0

Finding a pole: Denominator = 0

## Sampling

Closed loop: Good sampling rate is 10-20× the bandwidth

Convert pole from rads to Hz: 1Hz = 2π rad

Find a zero from the numerator and cancel the pole closest to it

fs ≈ chosen pole/2πrad







If sampling rate is given, use C(s) to get: 

If not, use –3dB frequency

if you have 1/(s + a) , bandwidth = a

**Bandwidth** {Hz} = fs {rads}

## 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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Increasing this** | **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**: a PID tuning rule

* a plant with neither integrators nor dominant complex-conjugate pairs
* Look at the *open loop response*. It could have a longer rise time / overshoot than preferred.
* Tangent to curve on upslope

**Gain** [H(s)]:

**Noise frequency** [ωn]:

**Noise amplitude** [an]:

**Open loop**: plant with no control

**Closed loop**: 





So, you need to rearrange your H(s) that is in the first formula to look more like the second formula

# Jitter

**Jitter** [J]: a delay

**Relative Jitter**: difference in response time between current and previous response times

maxk |Ri,k+1 − Ri,k|

**Absolute Jitter**: difference between largest response time and smallest response time

maxk Ri,k – mink `Ri,k

Jabsolute ≥ Jrelative

# Fail

* **Fail-safe**: safe state
  + in the event of a specific type of failure, responds in a way that will cause no harm, or at least a minimum of harm, to other devices or to personnel
* **Fail-stop**: freezes
  + failure in one component might not be visible until it leads to failure in another component
* **Fail-fast**: when a problem occurs, a fail-fast system fails immediately

**Passive Fault Tolerance**: fault masking, e.g. TMR

**Active Fault Tolerance**: reconfiguration, like turning off sets of servers, e.g. Pair and Spare

**Duplication**:

* primary produces an output
* secondary produces an output, compared with primary
* if outputs in disagreement, ﬂag error

**Echoing**: having a second copy of your data in different parts of memory

**Mirroring**: having a second copy of the data that has been bit flipped.

Types of faults:

* Permanent: fails until replaced
* Intermittent: fault that appears and comes back
  + e.g. loose wire
* Transient: fault that appears, then goes away (think settling time)

# Voting Schemes

**Plurality** [k]: number of votes needed for a majority

* **Median voter**: chooses median value as *correct* output (for this example, 2.00)
* **Majority voter**: given observations, di, and tolerance ε, i.e. willingness for error in *correct* value:
  1. Construct sets, Pk: x ∈ Pk ↔ |x – y| ≤ ε for all y ∈ Pk, where Pk has all elements within ε of each other, Pk is maximal, i.e. cannot add any points to it
  2. Choose Pk with largest |Pk|, where |Pk| = len(Pk)
     + If |Pk| > floor(N/2): choose any one of Pk as *correct* value or a combination of many
     + Else, no result
     + e.g. Choose:
       - ε = 0.1
       - P1 = {2.00, 2.01, 1.98, 2.05} => |P1| = 4 > floor(5/2) ← majority chooses value in Pk
       - P2 = {1.80}
     + What is the minimum value of ε that leads to the majority voter outputting a value?

ε = 0.03 (i.e. range of 2.00, 2.01, 1.98); d1, d2, d3 all satisfy |di – dj| ≤ 0.03

* **K-plurality**: make a section of size k
* **Pair-and-spare**: when you have 2 sets of 2

**Modular Redundancy (MR)**: when you have multiple separate redundant systems

**Triple Modular Redundancy (TMR)**: having 3 systems with the same purpose running together. This ensures that if a system is not working properly, the things it outputs is compared against the other 2 systems and can be verified as wrong

**Cold spare**: a redundant system that is off until needed

**Warm spare**: a redundant system that is in a standby state until needed

**Hot spare**: a redundant system that is functional, on, and actively collaborating with the primary system

**Byzantine General’s Problem**: useful for sensors with noisy values

n ≥ 3t + 1

[n]: number of generals

[t]: number of traitorous generals

A lot of these problems are based on binomial

## Information Redundancy

e.g. 1101100\_, ‘\_’ is a **parity** or **checksum**, where the number of 1’s are identified as even (1) for **even parity** and odd (0) for **odd parity**.

* Can detect single bit errors
* Cannot correct errors
* Adding more bits can also allow for correction
* Used in all **C**ommunicationp**R**oto**C**ols **(CRC)**

# Execution Time

Underestimate the time

**Best Case Execution Time (BCET)**:

**Worst Case Execution Time (WCET)**:

(BCET^): estimation of BCET

(WCET^): estimation of WCET

BCET^ < BCET < WCET, WCET^

# Natural Language Standards

**Formal requirement**: If <condition>, <action> shall occur

**Soft requirement**: within <response time>, <minimum probability> of the time

**Hard requirement**: within <response time>

QoS: a functional requirement with a hard / soft requirement

# RTOS

|  |  |
| --- | --- |
| **RTOS** | **RT Java (Programming)** |
| more flexible |  |
| WCET worse | WCET better |
| Modifications of the program may have different results on different OSs | Portable |
|  | Less power consumption |
|  | Might not actually happen real time |
| Better fine-grained control |  |
|  | better memory management |

Metronome is IBMs way of letting Java be used to develop real time systems by changing the way garbage collection is done. It’s an incremental garbage collector that has a mandatory minimum percentage of runtime of 70%, i.e. maximum 30% cleanup. This minimum is known as the Minimum Mutator Utilization (MMU)

Mark-and-sweep works by marking objects in use and reclaiming the unmarked objects