# SFWR ENG 4AA4

Kemal Ahmed
Fall 2015
Dr. Down

Note: information from the pre-requisite, SFWR ENG 3DX4 will not be included in this summary (although corrections will be).

## Contents



## 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 $f'(x) = \frac{f(x+h) - f(x)}{h}$

**Backwards Difference method**: derivatives using $f'(x) = \frac{f(x) - f(x-h)}{h}$

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

# Task optimization

**Task** [T]: $T_i = (p_i, r_i, e_i, d_i)$
**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 $U = \sum_{i=1}^{n} \frac{e_i}{p_i}$

If U > 1, nothing is feasible

If $r_i = 0$ and $p_i = d_i$, then write $T_i = (p_i, e_i)$

# 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. $T_1 = (100, 3)$; $T_2 = (2, 1)$
- 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 $e_{max}$.
- Constraints:
  1. $f \geq \max\limits_{1 \leq i \leq n}(e_i)$
  2. H % f = 0
  3. $2f - \gcd(p_i, f) \leq d_i$

**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 $U \leq n\left(2^{\frac{1}{n}} - 1\right)$
  - 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 <u>which</u> task has the highest priority, dynamic scheduling also re-evaluates <u>the actual</u> priorities, themselves.

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:

- $S_1$ = lock($S_1$)
- $S_1$^ = unlock($S_1$)

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

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

**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 $T_1$ has access to shared resource, so the time not allocated can be pre-empted
- Connect the pre-empted by $T_1$ when $T_1$ 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)**: 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 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."
    - **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.

## Sporadic Server

**Execution Budget** [$e_s$]: periodic tasks aren't flexible…
**Execution time** [$e_i$]: …sporadic tasks are

**Deadline** [$d_i$]: absolute deadline
**Release Time** [$r_i$]:

**Set of Sporadic Tasks** [θ]:
**Sporadic Task** [$S_i$]:

- Non-periodic task
- ($r_i$, $e_i$, $d_i$)
- Typically interrupt-driven

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

**Sporadic Server** [$\Phi_s$]: ($p_s$, $e_s$, θ, ρ)
**Periodic Task**: ($p_s$, $e_s$) a type of sporadic server

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

Assume:

- $\Phi_s$ scheduled with $T_i$ according to RM

We don't use $K_d$ 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, $K_d$ 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]:

$U = U_{periodic} + \delta U_{sporadic}$

**Error bound** [ε]:

**Slack** [ω]:
**Acceptance Test**: check of stuff

$$\omega(S_k, t) = \left\lfloor \frac{d_k - t}{p_s} \right\rfloor e_s - e_k - \sum_{S_i \in \theta : d_i < d_k} e_i - \xi_i$$

1. If ω($S_k$ , t) < 0, reject task

2. If $\omega(S_k, t) \geq 0$, need to check if already accepted sporadic tasks are adversely affected, i.e. $\omega(S_j, t) - e_k \geq 0$ holds for all $S_j \in \theta$ with $d_j \geq d_k$.

The set θ is maintained dynamically.

## Clocks

**Computer Clock** [C]:
**Standard Clock** [$C_s$]: perfect clock; has real time

Attributes:

- Correctness
- Bounded Drift
- Monotonicity
- Chronoscopicity

**(EPS)**: a bounded/maximum difference between the clock time and the real time
$|C(t) - C_S(t)| \leq EPS$

**Reset time** [r]: the real time you set the clock to when you reset it

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

- There's usually a reason why a clock drifts

**Drift Bound** [ρ]: maximum drift

$$\left|\frac{\mathrm{d}C(t)}{\mathrm{d}t} - 1\right| \leq \rho$$

**Reset Error** [ε]: error between actual time and time clock was set to at reset

**Total Error** [E]: $E(t) = \varepsilon + drift\_since\_reset$
$drift\_since\_reset \leq \rho(C(t) - r)$
$E(t) = \rho(C(t) - r) + \varepsilon \leq EPS$
$C(t) - r \leq (EPS - \varepsilon)/\rho$

Real time will be within this 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 $\left|\frac{\mathrm{d}^2C(t)}{\mathrm{d}t^2}\right| \leq \gamma$

## Christian's Algorithm

**Minimum Latency** [$T_{min}$]:
**Request Send Time** [$T_0$]:
**Request Receive Time** [$T_1$]:
**Server Time** [$T_{server}$]: time returned by the server

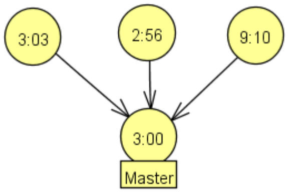
$$T_{\text{new}} = T_{\text{server}} + \frac{T_1 - T_0}{2}$$

Accuracy is $\pm \frac{T_1 - T_0}{2} - T_{\min}$

## Berkeley

Not often used, but useful for learning

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




2. Finds the average of the nodes. However, that's probably going to find a value that isn't near any of them.
3. Eliminate the outliers:
   a. Standard deviation: the more outliers, the harder to remove them
   b. Median
   c. **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. $\frac{1}{s^2 + 10s + 20}$

Remember this from 3DX4? Most of the stuff is still there, so refer to that. More here.
Each of the K's represent a different error or gain

4 types of controllers [P(s)]:

- **Proportional Controller (P),(PC)**: $\frac{K_p}{s^2 + 10s + (20 + K_p)}$
- **Proportional Integral (PI)**: $\frac{K_p s + K_i}{s^3 + 10s^2 + (20 + K_p s + K_i)}$
- **Proportional Derivative (PD)**: $\frac{K_d s + K_p}{s^2 + (10 + K_d)s + (20 + K_p)}$

- **Proportional Integral Derivative (PID)**: $\frac{K_p s + K_i + s^2 K_d}{s(s^2 + as + b) + K_p s + K_i + s^2 K_d}$

$$\frac{K_d s^2 + K_p s + K_i}{s^3 + (10 + K_d)s^2 + (20 + K_p)s + K_i}$$

$$u(t) = K_p e(t) + K_d \dot{e}(t) + K_i \int_0^t e(v)\mathrm{d}v$$

## Designing a PID Controller

1. Obtain an open-loop response and determine what needs to be improved
2. Add $K_p$ to improve the rise time
3. Add $K_d$ to improve the overshoot
4. Add $K_i$ to eliminate the steady-state error
5. Adjust each of $K_p$, $K_d$, and $K_i$ 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 |
|---|---|---|---|---|
| **$K_p$** | Decrease | Increase | Small Change | Decrease |
| **$K_d$** | Small Change | Decrease | Decrease | No Change |
| **$K_i$** | Decrease | Increase | Increase | Eliminate |

**Ziegler-Nichols 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
- For PID controllers

**Noise frequency** [$\omega_n$]:
**Noise amplitude** [$a_n$]:
**Open loop**: plant with no control
**Closed loop**: $\frac{C(s)G(s)}{1 + C(s)G(s)}$

## Jitter

**Jitter** [J]: a delay
**Relative Jitter**: difference in response time between current and previous response times
$\max_k |R_{i,k+1} - R_{i,k}|$
**Absolute Jitter**: difference between largest response time and smallest response time
$\max_k R_{i,k} - \min_k R_{i,k}$

Absolute jitter ≥ relative jitter

## Fail

- **Fail-safe**: 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**: detects exceptions, but doesn't worry about handling them or raising them
  - 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

## 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, $d_i$, and tolerance ε, i.e. willingness for error in *correct* value:
  1. Construct sets, $P_k$: $x \in P_k \leftrightarrow |x - y| \leq \varepsilon$ for all $y \in P_k$, where $P_k$ has all elements within ε of each other, $P_k$ is maximal, i.e. cannot add any points to it
  2. Choose $P_k$ with largest $|P_k|$, where $|P_k| = \text{len}(P_k)$
     - If $|P_k| > \text{floor}(N/2)$: choose any one of $P_k$ as *correct* value or a combination of many
     - Else, no result
     - e.g. Choose:
       - $\varepsilon = 0.1$
       - $P_1 = \{2.00, 2.01, 1.98, 2.05\} => |P_1| = 4 > \text{floor}(5/2) \leftarrow$ majority chooses value in $P_k$
       - $P_2 = \{1.80\}$
     - What is the minimum value of ε that leads to the majority voter outputting a value?

$\varepsilon = 0.03$ (i.e. range of 2.00, 2.01, 1.98); $d_1$, $d_2$, $d_3$ all satisfy $|d_i - d_j| \leq 0.03$

- **K-plurality**: make a section of size k