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)

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

Difference Equations

Forward difference method: derivatives using $f'(x) = \frac{f(x+h) - f(x)}{h}$

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

e.g.)

u[n] – u[n–1] = 48e[n] – 40e[n–1]

$$a\dot{U} + bU = c\dot{E} + dE$$

$$a\frac{u(kT_s) - u((k-1)T_s)}{T_s} + bu(kT_s) = c\frac{e(kT_s) - e((k-1)T_s)}{T_s} + de(kT_s)$$

$$a\frac{u(kT_s)}{T_s} - \frac{au((k-1)T_s)}{T_s} + bu(T_s) = c\frac{e(kT_s)}{T_s} + de(kT_s) - c\frac{e((k-1)T_s)}{T_s}$$

Group:

$$\left(\frac{a}{T_s} + b\right)u(kT_s) + \left(-\frac{a}{T_s}\right)u((k-1)T_s) = \left(\frac{c}{T_s} + d\right)e(kT_s) - \left(\frac{c}{T_s}\right)e((k-1)T_s)$$

Equate each section to the values from the equation:

$$\frac{a}{T_s} + b = 1$$

$$b = 1 - \frac{a}{T_s}$$

$$\boxed{b = 1 - 10a}$$

$$\frac{a}{T_s} = 1$$

$$\boxed{a = \tfrac{1}{10}}$$

$\boxed{b=0}$

$\frac{c}{T_s}+d=48$

$\boxed{d=48-10c}$

$-\frac{c}{T_s}=-40$

$\boxed{c=4}$

d = 48 – 40 **= 8**

$\frac{1}{10}\dot{U}+0U=4\dot{E}+8E$

$\frac{U}{E}(0.1s)=4s+8$

$\frac{U}{E}=\frac{4s+8}{0.1s}$

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 <u>needs</u> 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 $U=\sum_{i=1}^{n}\frac{e_i}{p_i}$

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. $f \geq \max_{1 \leq i \leq n}(e_i)$
    2. H % f = 0
    3. $2f - \gcd(p_i, f) \leq d_i$

Least Compute Time (LCT): tasks with smallest execution times executed first
- 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 $U \leq 1$
- 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 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:

- 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

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

Deadline [d_i]: absolute deadline
Release Time [r_i]: delay before the task is released to be executed

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

- Non-periodic task (a.k.a. aperiodic)
- (r_i, e_i, d_i)
- Typically interrupt-driven

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

Sporadic Server [Φ_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:

- Φ_s scheduled with T_i according to RM

Open loop response: plant with no control

Effective Utilization [δ]:

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

Error bound [ε]:

[i]: all previous
[k]: current index

Slack [ω]:
Acceptance Test: check of stuff
[ζ]: time spent processing task already

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

1. If $\omega(S_k, t) < 0$, 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; 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
$|C(t) - C_S(t)| \leq EPS$

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

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

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 $\left|\frac{\mathrm{d}^2 C(t)}{\mathrm{d}t^2}\right| \le \gamma$

Cristian'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

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

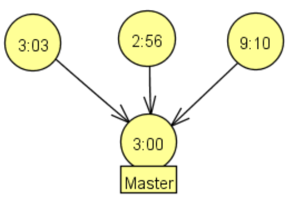
Accuracy is $\pm\frac{T_1 - T_0}{2} - T_{\min}$

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



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, i.e. $\sqrt{\frac{1}{2} \times \Sigma(x_i - \mu)^2}$
    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
    d.

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)**: $\frac{K_p}{s^2 + as + (b + K_p)}, \frac{K_p}{s^2 + 10s + (20 + K_p)}$

- **Proportional Integral (PI)**: $\frac{K_p s + K_i}{s^3 + as^2 + \left(b + K_p s + K_i\right)}$ $\frac{K_p s + K_i}{s^3 + 10s^2 + \left(20 + K_p s + K_i\right)}$
- **Proportional Derivative (PD)**: $\frac{K_d s + K_p}{s^2 + \left(a + K_d\right)s + \left(b + K_p\right)}, \frac{K_d s + K_p}{s^2 + \left(10 + K_d\right)s + \left(20 + K_p\right)}$
- **Proportional Integral Derivative (PID)**: $\frac{K_p s + K_i + s^2 K_d}{s\left(s^2 + as + b\right) + K_p s + K_i + s^2 K_d}$

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

$$s^3 + 10s^2 + K_d s^2 + 20s + K_p s + K_i$$

$$= s\left(s^2 + 10s + 20\right)$$

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

Dominant pole: largest magnitude

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.

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
$f_s \approx$ chosen pole $\frac{rad}{s}$ /2πrad

$$H(s) = \frac{1}{\frac{s^2}{\omega_n^2} + \frac{2\zeta s}{\omega_n} + 1}$$

$$\omega_h \approx \frac{\omega_n}{2\zeta}$$

$$\omega_h = \omega_n \sqrt{1 - 2\zeta^2 + \sqrt{2 - 4\zeta^2 + 4\zeta^4}}$$

If sampling rate is given, use C(s) to get: $\frac{C}{T_s}\left(u[k] - u[k-1]\right) + Du[k] = \frac{A}{T_s}\left(e[k] - e[k-1]\right) + Be[k]$

If not, use –3dB frequency

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

Bandwidth {Hz} = f_s {rads}

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 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 [ω_n]:
Noise amplitude [a_n]:
Open loop: plant with no control

Closed loop: $H(s) = \frac{C(s)G(s)}{1+C(s)G(s)}$

$$H(s) = K_C \frac{\prod_{i=0}^{N}(s - z_i)}{\prod_{j=0}^{M}(s - p_j)}$$

$$C(s) = K_D s + K_P + \frac{K_I}{s}$$

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
$\max_k |R_{i,k+1} - R_{i,k}|$

Absolute Jitter: difference between largest response time and smallest response time
$\max_k R_{i,k} - \min_k `R_{i,k}$

$J_{absolute} \geq J_{relative}$

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, 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:
       - $\varepsilon = 0.1$
       - $P_1 = \{2.00, 2.01, 1.98, 2.05\} \Rightarrow |P_1| = 4 > \text{floor}(5/2) \leftarrow$ 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?

ε = 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
- **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

$n \geq 3t + 1$
[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 <u>all</u> **C**ommunication p**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