TTK4145 – Real-time Programming

Sondre Bø Kongsgård
sondrebk@stud.ntnu.no

Innhold

1 Fault model and software fault masking

This chapter is based on Ch. 3.7 in the book Transaction Processing: Concepts and Techniques (GR).

Fault model The one used in this chapter involves three entities: *processes, messages and storage.*

Unexpected faults Faults that are not tolerated by the design. Two categorizations:

- *Dense faults*: The algorithms will be n-fault tolerant. If there are more than n faults within a repair period, the service may be interrupted (in this case the system should be designed to stop).
- *Byzantine faults*: The fault model postulates certain behavior – for example it may postulate that programs are failfast. Faults in which the system does not conform to the model behavior are called Byzantine.

Underlying progression

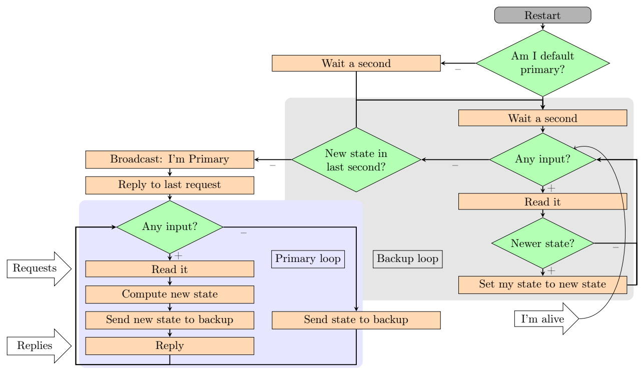
- *Failfast*: They either execute the next step, or they fail and reset to the null state.
- *Available*: Failfast + repairability
- *Reliable*: Continuous operation

Checkpoint-Restart Write state to storage after each acceptance test, but before each event. This approach can be somewhat slow (hours/days).

Process pairs OS generates a backup process for each new primary process. The primary sends *I'm Alive* messages to the backup on a regular basis.

The backup can take over in three different ways:

- *Checkpoint-restart.* The primary records its state on a duplexed storage module. At takeover, the backup starts by reading these duplexed storage pages. (quick repair)
- *Checkpoint message.* The primary sends its state changes as messages to the backup. At takeover, the backup gets its current state from the most recent message. (basic pairs must checkpoint)
- *Persistent.* The backup restarts in the null state and lets the transaction mechanism clean up (undo) any recent uncommitted state changes. (simple)



2 Scheduling real-time systems

This chapter is based on Ch. 11 in (BW).

Scheduling Restricting the non-determinism found within concurrent systems, by providing

- An algorithm for ordering the use of system resources.
- A means of predicting the worst case behaviour under this ordering.

Ie. it has to assign priorities to the different processes, and run a schedulability test. In a *static* scheduler this is decided/calculated prior to execution, while a *dynamic* scheduler calculates in run-time.

The scheduler must be predictable, so that we can analyze the system. We want the analysis to be simple, but not too conservative, as we want to facilitate high utilization. The result of the analysis must show that all tasks meet their deadlines.

Task-based scheduling An approach where task execution is directly supported. A task is deemed to be in a specific *state*:

- Runnable
- Suspended waiting for a timing event – appropriate for periodic tasks
- Suspended waiting for a non-timing event – appropriate for sporadic tasks

We consider three scheduling approaches:

- *Fixed-Priority Scheduling (FPS).* Each task has a fixed, static, priority which is computed pre-run-time. This is the most widely used approach.
- *Earliest Deadline First (EDF) Scheduling.* The tasks are executed in the order determined by the absolute deadlines (earliest deadline yields highest priority). Typically computed at run-time, so this scheme is dynamic.
- *Value-Based Scheduling (VBS).* Assigning a value to each task to decide which to run next. Adaptive.

Scheduling characteristics A schedulability test can be:

- *Sufficient.*
- *Necessary.*
- *Sustainable.* The test still predicts schedulability when the system parameters (eg. period, deadline) improve.

Preemption In a preemptive scheme, the scheduler automatically switches to tasks of higher priority.

(In non-preemptive schemes, the lower-priority task will be allowed to complete before the other executes.)

Simple task model In the following discussion we make the following assumptions of the tasks (comments on how realistic they are in paranthesis):

- Fixed set of tasks (No sporadic tasks. Not optimal, but can be worked around)
- Periodic tasks with known periods (Realistic in many systems)
- The tasks are independent (Completely realistic in an embedded system)
- Overheads, switching times can be ignored (Depends)
- Deadline == Period (Inflexible, but fair enough)
- Fixed worst-case execution time (Not realistic to know a tight (not overly conservative) estimate here)
- Rate-monotonic priority ordering (Our choice, so ok)

Standard notation

Notation	Description
B	Worst-case blocking time for the task (if applicable)
C	Worst-case execution time (WCET) of the task
D	Deadline of the task
I	The interference time of the task
J	Release jitter of the task
N	Number of tasks in the system
P	Priority assigned to the task (if applicable)
R	Worst-case response time of the task
T	Minimum time between task releases (task period)
U	The utilization of each task (equal to C/T)
$a - z$	The name of a task

Fixed-priority scheduling (FPS)
Rate monotonic priority assignment: shortest period yields highest priority. (For two tasks i and j, $T_i < T_j \Rightarrow P_i > P_j$). The higher value of P, the greater the priority (the other way around compared to other disciplines).

Utilization test. If the following condition is true then all N tasks will meet their deadlines:

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N \left(2^{1/N} - 1 \right) \tag{1}$$

The test is sufficient, but not necessary (a system may fail the test, and still be schedulable). Additionally, two drawbacks: not exact, not applicable to a more general task model.

Response time analysis (RTA). Exact (sufficient and necessary) test.

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{2}$$

w_i = A mathematical entity to solve the fixed-point equation above, equal to R_i after the final recursion.
$hp(i)$ = The set of indices that have a higher priority than i

Recursive test. Start with the highest-priority task (which will have response time equal to its computation time), and move to the lower priority tasks. The response time of a task i has been found if $w_i^{n+1} = w_i^n$.

Priority inversion That a high-priority task ends up waiting for a lower-priority task, which can happen if they share a resource. (In the previous discussion tasks have beeen viewed as independent, but tasks may share resources, and therefore have the possibility to be suspended; waiting for a future event (eg. an unlocked semaphore)).

Unbounded priority inversion That the high-priority task may potentially end up waiting forever. The unboundedness occurs when there are one or more medium-priority tasks that prevent the low-priority task from running - and thereby releasing the shared resource.

Priority inheritance If a task p is suspended waiting for task q to undertake som computation then the priority of q becomes equal to the priority of p (if it was lower to start with). Does NOT prevent deadlocks.

Priority ceiling When this kind of protocol is used...

- A high-priority task can be blocked at most once during its execution by lower-priority tasks
- Deadlocks are prevented
- Transitive blocking is prevented
- Mutual exclusive access to resources is ensured

The protocol ensures that if a resources is locked, by task a say, and could lead to the blocking of a higher-priority task b, then no other resources that could block b is allowed to be locked by any task other than a. A task can therefore be delayed by not only attempting to lock a previously locked resource, but also when the lock could lead to multiple blocking on higher-priority tasks.

Original ceiling priority protocol (OCPP):

- Each task has a static default priority.
- Each resource has a static ceiling value (the maximum priority of the tasks that use it).
- Tasks have dynamic priority (its own static priority or any (higher) priorities inherited from blocking higher-priority tasks).
- A resource may only be locked by tasks with higher priority than any currently locked tasks.

Immediate ceiling priority protocol (ICPP): Instead of waiting until it actually blocks a higher priority task, ICPP increases the priority of a task as soon as in locks a resource. ICPP will not start executing until all its needed resources are free, since unavailable resouces it needs indicates that a task of higher priority is running.

Ensures mutual exclusion: If a task under (eg.) ICPP has access to a resource it will have the highest possible priority for that resource, and thus no other task demanding this resource will run.

Ensures no deadlocks: If a task has a resource and requests another, then the new resource it requested can not have lower priority.