TTK4145 – Real-time Programming

Sondre Bø Kongsgård
sondrebk@stud.ntnu.no

Contents

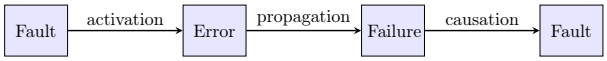
1 Fault Tolerance Basics

This chapter is based on Ch. 2 in Real-Time Systems and Programming Languages 4E (BW).

Reliability A measure of the success with which the system conforms to some authoritative specification of its behaviour.

Failure vs Fault vs Error A few definitions:

- *Fault:* A mechanical or algorithmic cause which, given the right conditions, produce an unexpected or incorrect result.
- *Error:* An internal unexpected problem in a system, caused by activation of a fault.
- *Failure:* System deviation from its specification.



Types of fault

- *Transient faults:* Occur at a particular time, remain in the system for a period, then disappear afterwards. An example of this is radioactivity.
- *Permanent faults:* A fault that persists in the system until it is fixed. All software faults are permanent faults.
- *Intermittent faults:* A special case of transient faults that occur repeatedly. Overheating is an example.

Failure modes Two general domains of failure modes can be identified:

- *Value failure:* The output is wrong or even corrupted/unusable.
- *Time failure:* The service is delivered at the wrong time.

Fault prevention Attempts to eliminate any possibility of faults creeping into a system before it goes operational. Two stages: fault avoidance and fault removal.

Fault tolerance Enables the system to continue functioning even in the presence of faults. Different levels of faults can be provided by a system:

- *Full fault tolerance:* The system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance.
- *Graceful degradation:* The system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair.
- *Fail safe:* The system maintains its integrity while accepting a temporary halt in its operation.

Redundancy All techniques for achieving fault tolerance rely on extra elements (redundancy) introduced into the system to detect and recover from faults. Can be implemented in both software and hardware.

Distinguish between static and dynamic redundancy for hardware:

- *Static:* Redundant components are used inside a system (or subsystem) to hide the effects of faults. (error masking)
- *Dynamic:* The redundancy supplied inside a component which indicates explicitly or implicitly that the output is in error. (error detection)

N-version programming Defined as the independent generation of N (where $N \geq 2$) functionally equivalent programs from the same initial specification. The programs execute concurrently without interaction, and the results (compared by a driver process) should be identical. However, if they do not produce the same result the driver might choose the most common result, ask the processes to compute it again, or simply terminate the faulty process.

To achieve diversity, different programming languages and different development environments could be used. Alternatively, if the same language is used, different compilers and support environments should be employed. (N-version programming is the software equivalent of static redundancy.)

Downsides to this solution:

- In a real-time system, the driver process and different programs may need to communicate – introducing a communication module.
- High cost of introducing a redundant language/hardware etc.
- Most software faults originate from the specification – all N versions may suffer from the same fault.

Dynamic redundancy The redundant components only come into action when an error has been detected. Has four stages presented in the following four paragraphs.

Error detection Two classes of error detection techniques can be identified. You should know this list by heart!

- Environmental detection
- Application detection. Since the recovery system is not continually running, there needs to be some way to detect a fault. There are many triggers that could be used.
  - Replication checks –
  - Timing checks –
  - Reversal checks –
  - Coding checks –
  - Reasonableness checks –
  - Structural checks –
  - Dynamic reasonableness checks –

Damage confinement and assessment Concerned with structuring the system as to minimize the damage caused by a faulty component (also known as firewalling). Two techniques: modular decomposition and atomic actions.

Error recovery Once an error situation has been detected and the damage assessed, error recovery procedures must be initiated.

Forward error recovery: Attempts to continue from an erroneous state by making selective corrections to the system state. Although it is efficient, it is system specific and depends on accurate predictions of the location and cause of errors.

Backward error recovery: Relies on restoring the system to a safe state (recovery point) previous to that in which the error occurred. Straightforward if no threads, but gets complicated with more threads. A big advantage: Does not rely on finding the location or cause of fault. A huge disadvantage: It is difficult to undo eg. a missile launch...

Fault treatment Two stages: Fault location and system repair.

Domino effect If one thread needs to roll back due to an error, then it must undo the communication with the other threads. This may propagate further back and forth, and will end up in multiple threads having to roll back as well.

Recovery blocks Blocks in the normal programming sense except that at the entrance is an automatic *recovery point* and at the end an *acceptance test.* (dynamic recovery)

Acceptance test Used to test that the system is in an acceptable state after the execution of the block. It can utilize several of the methods discussed for error detection. If it fails, the program will be restored to the recovery point at the beginning of the block and an alternative module will be executed.

2 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 (feil sideeffekt), messages and storage (static redundancy).*

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

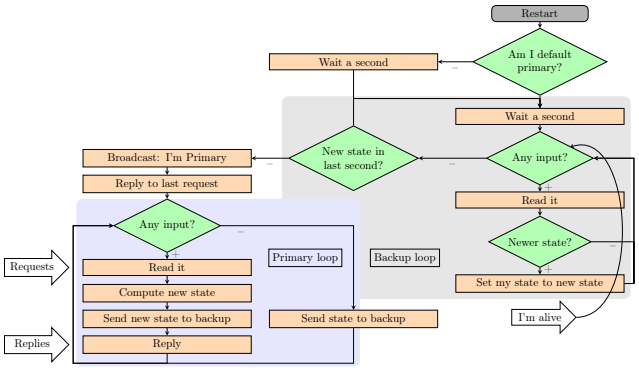
Checkpoint-Restart Write state to storage after each acceptance test, but before each event. This approach can be somewhat slow (hours/days).

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)



3 Transaction Fundamentals

This chapter is based on Ch. 1 in the book Java Transaction Processing (LMP).

Atomic operation An operation that is indivisible (it is not possible to decompose it into smaller operations), eg. a money transfer between two bank accounts.

Transaction Refers to the term *atomic transaction*, which provides an "all-or-nothing" property to work that is conducted within its scope. An atomic transaction has the same properties as an atomic action plus the added feature that if a failure happens, the components are returned to their natural state instead of being left possibly inconsistent as in an atomic action.

Controlling a transaction A few definitions:

- *Commit:* When all parts of the transaction commits, the transaction finishes and all its actions are made permanent.

- *Rollback:* An abortion of the transaction. All work done by the transaction is reverted.

ACID properties Properties of an atomic action:

- *Atomicity.* The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- *Consistency.* Transactions produce consistent results and preserve the internal consistency of the data it acts on.
- *Isolation.* Intermediate states produced while a transaction is executing are not visible to others. Furthermore, transactions appear to execute serially, even if they are actually executed concurrently.
- *Durability.* The effects of a committed transaction are never lost.

Termination A transaction can be terminated in two ways: *committed* or *aborted* (rolled back). When a transaction is committed, all changes made within it are made durable (forced onto stable storage). When a transaction is aborted, all of the changes are undone.

Coordinator Responsible for governing the outcome of the transaction. It can be implemented as a separate service or may be co-located with the user for improved performance. It communicates with enlisted participants to inform them of the desired termination requirements.

Transaction manager factory Responsible for managing coordinators for many transactions.

Two-phase commit This protocol is used to ensure consensus between participating members of the transaction (ie. ensure it has an atomic outcome).

In phase 1 the coordinator communicates with all the action participants to determine whether they will commit or abort. An abort reply, or no reply, from any participant acts as a veto, causing the entire action to abort. If the transactions commits, then phase 2 is entered and the coordinator forces the participants to carry out the decision. Each participant making a commit response to the coordinator in phase 1 is blocked until they have received the coordinator's phase 2 message.

Note that the two-phase commit protocol is not client-server based. It simply talks about a coordinator and participants, and makes no assumption about where they are located.

Interference free / Serializable If there are variables that are not shared between programs, some aspects of the programs can be executed concurrently. A partly concurrent execution order is termed serializable if it produces an equivalent result as doing it in serial order. If a program possess this property it is said to be atomic with respect to concurrency.

Concurrency control mechanisms Ensures serializability.

- *Two-phase concurrency control:* All operations on an object is of type "read" or "write". Many computations can hold a "read-lock", which is associated with an object by a computation before reading it. However, if someone wants to change (insert/modify/delete) the object, this requires a "write-lock". This can not be obtained if any of the other operations hold either a read or write lock on that object. This is to make sure that no one changes something others are using in their operations. If you have obtained a write-lock, this will block all other requests for read and write until you release the lock. All computations must follow a two-phase locking policy.

  In phase 1, the growing phase, locks are acquired, but not released. In phase 2, the shrinking phase, locks are released but can't be acquired. To avoid

the cascade roll back problem (see under), all locks are released simultaneosly in the shrinking phase.

- *Pessimistic concurrency control:* When a resource is accessed, a lock is obtained on it. The lock will remain held on that resource for the duration of the transaction. Disadvantages with this style:
  - Deadlocks
  - Concurrency may be limited by the granularity of the locks
  - The overhead of acquiring and maintaining concurrency control information in an environment where conflict or data sharing is not high.

- *Optimistic concurrency control:* Locks are only acquired at the end of the transaction when it is about to terminate. When updating a resource, the operation needs to check if this conflicts with updates that may have occurred in the interim, using timestamps. Most transaction systems that offer optimistic concurrency control will roll back if a conflict is detected, which may result in a lot of work being lost.

- *Type-specific concurrency control:* Concurrent read/write or write/write operations are permitted on an object from different transactions provided these operations can be shown to be non-interfering. Object-oriented systems are well suited to this approach, since semantic knowledge about the operations of objects can be exploited to control permissible concurrency within objects.

- *Deadlock detection and prevention:* A transaction is *blocked* if it cannot acquire a lock on a resource it waits for to be released. In some systems, it is possible for transactions to wait for each other – the system is *deadlocked*.

  The only way to resolve a deadlock is for at least one of the transactions to release its locks that are blocking another transaction (it must roll back). Techniques for deadlock detection:

  - *Timeout-based:* A transaction waits a specified period of time, then rolls back assuming there is a deadlock.
  - *Graph-based:* Tracks waiting transaction dependencies by constructing a waits-for graph: nodes are waiting transactions and edges are waiting situations. Guaranteed to detect all deadlocks, but may be costly to execute in a distributed environment.

Two-phase commit optimizations There are several variants to the standard two-phase commit:

- *Presumed abort:* If during system recovery from failure no logged evidence for commit of some transaction is found by the recovery procedure, then it assumes that the transaction has been aborted, and acts accordingly. This means that it does not matter if aborts are logged at all, and such logging can be saved under this assumption.
- *One-phase:* If there is only one participant involved in the transaction, there is implicit consensus on whether to commit or not and the coordinator need not drive it through the prepare phase.
- *Read-only:* A participant may indicate to the coordinator if it is responsible for a service that did not do any work during the course of the transaction, and can be left out of the second phase.
- *Last resource commit:* It is possible for a single resource that is one-phase aware (no prepare, only commit or roll-back), to be enlisted in a transaction with two-phase commit aware resources. The coordinator executes the prepare phase on only the latter, and if the decision is to commit, then the one-phase aware resource is informed about this.

Heuristic transactions *Heuristic outcome:* The case in which the coordinator informs the participant that the outcome and the decision are contrary. (May happen when blocking occurs in the two-phase commit protocol (where it is used to ensure atomicity). To break the blocking, participants that are past the prepare phase are allowed to make decisions as to whether commit or rollback.) It has a corresponding *heuristic decision*.

There are two levels of heuristic interaction:

- *Participant-to-coordinator:* If a participant makes an autonomous decision, it must assume that it caused an heuristic and remember the choice. At this point the participant is in a *heuristic state*.

- *Coordinator-to-application:* If the choice is different from the coordinator, then a heuristic outcome must be reported to the application. The transaction will still be marked as committed of rolled back, but it will also be in a heuristic state.

Outcome	Description
Heuristic rollback	The commit operation failed because some or all of the participants unilaterally rolled back the transaction.
Heuristic commit	An attempted rollback operation failed because all of the participants unilaterally committed. One situation where this might happen is if the coordinator is able to successfully prepare the transaction, but then decides to roll it back because its transaction log could not be updated. While the coordinator is making its decision, the participants decides to commit.
Heuristic mixed	Some participants commited, while others were rolled back.
Heuristic hazard	The disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

Transaction log Registration of the participants and where they have reached in the protocol. After the coordinator's decision has been conducted, the participant log can be deleted and informed to the coordinator.

Nested transactions transaction!nested Transactions that are contained within the resulting transaction. The enclosing transaction is referred to as a *parent* of a nested (or *child*) transaction → a hierarchical structure (with the root referred to as the *top-level transaction*).

The effect of a nested transaction is provisional upon the commit/rollback of its enclosing transaction (the effects will be recovered if the enclosing transaction aborts).

Subtransactions are useful for two reasons:

- *Fault isolation:* If the subtransaction rolls back then this does not require the enclosing transaction to roll back.
- *Modularity:* If there is already a transaction associated with a call when a new transaction is begun, then the transaction will be nested within it. If the object's methods are invoked without a client transaction, then the object's transaction will be top-level, otherwise they will be nested within the scope of the client's transactions.

Because nested transactions do not make any state changes durable until the enclosing top-level transaction commits, there is no requirement for failure recovery mechanisms for them.

Interposition The technique of using proxy (or subordinate) coordinators. Each machine that imports a transaction context may create a subordinate coordinator that enrolls with the imported coordinator as though it was a participant.

When and why interpositions occurs:

- *Performance:* If a number of participants reside on the same node, or are located physically close, can then improve performance for a remote coordinator to send a single message to a subcoordinator that is co-located with those participants and for that subcoordinator to disseminate the message locally, rather than send the same message to each participants.
- *Security and trust:* A coordinator may not trust indirect participants and vice versa, that makes direct registration impossible.
- *Connectivity:* Some participants may not have direct connectivity with a specific coordinator, requiring a level of indirection.
- *Separation of concerns:* Many domains and services may simply not want to export (possibly sensitive) information about their implementations to the outside world.

4 Atomic Actions

This chapter is based on Ch. 7 in Real-Time Systems and Programming Languages 4E (BW).

Atomic actions The activity of a component is said to be atomic if there are no interactions between the activity and the system for the duration of the action.

Nested atomic actions An atomic action, even though viewed as indivisible, can be composed by internal atomic actions.

Two-phase atomic actions For efficiency and increased concurrency, it is advantageous if tasks will wait as long as possible before requiring resources, and freeing them as soon as possible. This could however make it possible for other tasks to detect a change in state before the original task is done performing the atomic action.

To avoid this, atomic actions are split into two separate phases, a 'growing' phase where only acquisitions can be made, and a 'shrinking' phase where resources can be released while no new acquisitions can be made. This way, other tasks will always see a consistent system state.

Atomic transactions Has all the properties of an atomic action plus the added feature that its execution is allowed to either succeed or fail.

If an atomic action fails, the system may be left in an inconsistent state. If an atomic transaction fails, the system will return to the state prior to the execution.

There are two distinct properties of atomic transactions:

- *Failure atomicity:* The transaction must complete successfully or have no effect.
- *Synchronizations atomicity:* The transaction must be indivisible so that no other transaction can observe its execution.

Even though transactions provide error detection, they are not suitable for fault-tolerant systems. This is because they provide no recovery mechanism in case of errors, and will not allow any recovery procedures to be run.

Requirements for atomic actions For implementation of atomic actions in a real-time system, it is necessary with a well-defined set of requirements which must be fulfilled.

- *Well-defined boundaries:* Each atomic action should have a start, and end and a side boundary. A start boundary is the locations in the task where the action is initiated; the end is the location where the action will end in the task. The side boundary is the separation of the tasks involved in an action from the rest of the system.

- *Indivisibility:* Atomic actions must not allow any exchange of information between the tasks inside and outside the action. Additionally, there are no synchronization at the start of atomic actions, but there is an implied synchronization at the end; tasks are not allowed to leave the atomic action until all tasks are willing and able to leave.

- *Nesting:* Atomic actions may be nested as long as they do not overlap with other atomic actions.

- *Concurrency:* It should be possible to execute different atomic actions concurrently.

Asynchronous notification The key to implementing error handling is the asynchronous messages, and most programming languages have some feature allowing asynchronous messages to be sent. There are mainly two principles for this: Resumption and termination.

Resumption model: Behaves like a software interrupt. A task contains an event handler which will handle prede ned events. When the task is interrupted, the event handler is executed, and when it is finished, the task will continue execution from the point where it was interrupted.

Termination model: Each task specifies a domain in which it will accept an asynchronous signal. If the task is in the defined domain and receives an asynchronous signal, it will terminate the domain (*asynchronous transfer of control*, ATC. If a task is outside the defined domain, the ATC will be queued or ignored. When the ATC is finished, the control is returned to the calling task at a different location than from where it was called.

Main reasons for using asynchronous notifications: (enables a task to respond quickly)

- *Error recovery:* When errors occur it can mean that the system states are no longer valid, that processes may never finish or a timing error may have occurred. There is no longer a point of waiting for a process to finish, and the best solution is to respond immediately.
- *Mode changes:* A real-time system often has several modes of operation. When sudden changes in modes happen, we want the system to respond immediately – for example when a transition to an emergency state occurs.
- *Scheduling using partial/imprecise computations:* Many algorithms calculating specific results operates by giving more accurate results for each iteration. The task must be interrupted to stop further refinement of the result.
- *User interrupts:* Whenever a system receives interrupts from a user, for example when an error occur, the response must be immediate.

5 Shared Variable Synchronization

This chapter is based on Ch. 5 in Real-Time Systems and Programming Languages 4E (BW). Based on the concepts presented in that chapter, go through a few of the cases to in The Little Book of Semaphores to get the right mindset.

Inter-task communication Usually based upon either *shared variables* or *message passing*.

Critical section A sequence of statements that must appear to be executed indivisibly.

Mutual exclusion The synchronization required to protect a critical section.

Busy-wait To signal a condition, a task sets the value of a flag; to wait for this condition, another task checks this flag and proceeds only when the appropriate value is read. If the condition is not yet set, this task must loop round and recheck the flag.

Livelock An error condition where tasks get stuck in their busy-wait loops and are unable to make progress.

Data race condition A data race condition is a fault in the design of the interactions between two or more tasks whereby the result is unexpected and critically dependent on the sequence or timing of accesses to shared data.

Semaphore A non-negative integer variable that, apart from initialization, can only be acted upon by two procedures – `wait` and `signal`. It is imperative that the implementation of these methods are atomic, ie. two tasks executing wait at the same time will not interfere with each other. The two methods works as follows:

- `wait(S)` – If the value of the semaphore, `S`, is greater than zero the decrement its value by one; otherwise delay the task until `S` is greater than zero (and then decrement its value).
- `signal(S)` – Increment the value of the semaphore, `S`, by one.

Benefits of using semaphores:

- Simplifies the protocols for synchronization.
- Removes the need for busy-wait loops.

With the semaphore, we can implement mutual exclusion of two tasks as the following:

```
mutex : semaphore; // initially 1
task P1;
  loop
  wait(mutex);
  <critical section>
  signal(mutex);
task P2;
  <same as P1>
```

It seems like the `wait` procedure needs to perform some sort of busy waiting and wait for a `signal` from some other process. This is where the RTSS (run-time support system) comes in. The RTSS should keep tabs on which tasks are waiting and remove them from the processor. Eventually, if the program is correct (for example, no deadlocks), some other task will call signal and the RTSS will let the task continue.

Deadlock Entails a set of tasks being in a state from which it is impossible for any of them to proceed.

Necessary conditions that must hold if deadlock is to occur:

- *Mutual exclusion* – only one task can use a resource at once.
- *Hold & wait* – there must exist tasks which are holding resources while waiting for others.

- *No preemption* – a resource can only be released voluntarily by a task.
- *Circular wait* – a circular chain of tasks must exists such that each task holds resources which are being requested by the next task in the chain.

Deadlock prevention:

- Optimistic concurrency control.
- Allocate all resources at once.
- Preemption.
- Global allocation order.

Indefinite postponement (Also called *lockout* or *starvation*) When a task is not given access to a needed resource in finite time.

Liveness A description of a task that is free from livelocks, deadlocks and starvation.

Synchronization in Java A Java method may be synchronized, which guarantees that at most one thread can execute the method at a time. Other threads wishing access, are forced to wait until the currently executing thread completes. Thus

```
void synchronized update() { ... }
```

can safely be used to update an object, even if multiple threads are active.

There is also a `synchronized` statement in Java that forces threads to execute a block of code sequentially.

Synchronization Primitives: The following operations are provided to allow threads to safely interact:

- `wait()` – Sleep until awakened
- `wait(n)` – Sleep until awakened, or until n milliseconds pass
- `notify()` – Wake up one sleeping thread (the first thread that called `wait()`)
- `notifyAll()` – Wake up all sleeping threads

Synchronization in Ada A protected object is a module, a collection of functions, procedures and entries along with a set of variables.

- Functions are read-only, and can therefore be called concurrently by many tasks, but not concurrently with procedures and entries.
- Procedures may make changes to the state of the object, and will therefore run under mutual exclusion with other tasks.
- Entries: The important thing is that these are protected by guards – boolean tests – so that if the test fails, the entry will not be callable – the caller will block waiting for the guard to become true. These tests can only be formulated using the object's private variables.

6 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:

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

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.

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

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.

7 Concurrency & Parallelism

This chapter is based on (well, more or less equal to) the solution to exercise 1.

Concurrency vs. parallelism "Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once."

Reasons for concurrency

- The need to model that happen "simulatenously".
- The need to exploit multicore hardware.
- The need to utilize hardware efficiently (even a single core).

Degree of difficulty to use concurrency Creating concurrent programs is easier if the system consists of multiple (somewhat) independent parts, like

- Two periodic functions with different periods.
- Handling multiple network requests.
- Polling different independent hardware, and generating separate events for each.
- Things that are "embarrassingly parallel".

Creating concurrent programs is harder if the concurrent parts have to interact, as we need some way to share data safely – communication and/or synchronization:

- The $i = i + 1$ problem demonstrates that strange things may happen if we don't share data safely.
- All threads share the same processor and memory (for common CPU architectures, anyway), so this also creates a form of interaction: Running one thread means we're not running another. This is a problem for real-time (ie. timing-sensitive) applications.

Different kinds of concurrent execution

- *Process*
  - An executing program (as opposed to just the code itself)
    - Executable code
    - One or more threads
    - An address space – the OS and hardware enforces that we cannot access the memory of one process from another
- *Thread*
  - A "unit of execution"
    - Has its own stack
    - Scheduled by the OS
    - Contains data structures for saving the current state of execution (the "context"), so its execution can be paused and resumed
  - Share memory with the same process
    - Some languages/extensions (such as Go/D) offer "Thread-local storage" of the heap, such that also the heap is not shared
- *Green thread*
  - Managed & (preemptively) scheduled by the runtime (ie. not OS-managed)
    - A scheduler is part of the core library of that language
    - Often mapped N:M over several threads
  - May or may not share memory
- *Co-routine*
  - Cooperatively (ie. manually, non-preemptively) scheduled by the user/programmer. No runtime, not OS-managed
    - All co-routines must call "yield"/"reschedule" (or its equivalent), else other co-routines will not run (hence the need to be "cooperative")
    - Sometimes mapped N:M over several threads, but usually just multiplexed over a single thread
  - Usually share memory
  - May not need to save/restore the execution context, saving significant management overhead

Thread spawning functions

- `pthread_create` (C++/POSIX) → Creates an OS-managed thread
- `threading.Thread` (Python) → Creates an OS-managed thread
- `go` (Go) → Creates a co-routine

CHECKLIST: Class Quality

Abstract Data Types

- ☐ Have you thought of the classes in your program as abstract data types and evaluated their interfaces from that point of view?

Abstraction

- ☐ Does the class have a central purpose?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface present a consisten abstraction?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- ☐ Are the class's services complete enough that other classes don't have to meddle with its internal data?
- ☐ Has unrelated information been moved out of the class?
- ☐ Have you thought about subdividing the class into component class, and have you subdivided it as much as you can?
- ☐ Are you preserving the integrity of the class's interface as you modify the class?

Encapsulation

- ☐ Does the class minimize accessibility to its members?
- ☐ Does the class avoid exposing member data?
- ☐ Does the class hide its implementation details from other classes as much as the programming language permits?
- ☐ Does the class avoid making assumptions about its users, including its derived classes?
- ☐ Is the class independent of other classes? Is it loosely coupled?

Inheritance

- ☐ Is inheritance used only to model "is a" relationships – that is, do derived classes adhere to the Liskov Substitution Principle?
- ☐ Does the class documentation describe inheritance strategy?
- ☐ Do derived classes avoid "overriding" non-overridable routines?
- ☐ Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- ☐ Are inheritance trees fairly shallow?
- ☐ Are all data members in the base class private rather than protected?

Other Implementation Issues

- ☐ Does the class contain about seven data members or fewer?
- ☐ Does the class minimize direct and indirect routine calls to other classes?
- ☐ Does the class collaborate with other classes only to the extent absolutely necessary?
- ☐ Is all member data initialized in the constructor?
- ☐ Is the class designed to be used as deep copies rather than shallow copies unless there's a measured reason to create shallow copies?

Language-Specific Issues

- ☐ Have you investigated the language-specific issues for classes in your specific programming language?

CHECKLIST: High-Quality Routines

Big-Picture Issues

- ☐ Is the reason for creating the routine sufficient?
- ☐ Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- ☐ Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- ☐ Does the routine's name describe everything the routine does?
- ☐ Have you established naming conventions for common operations?

- ☐ Does the routine have strong, functional cohesion – doing one and only one thing and doing it well?
- ☐ Do the routines a loose coupling – are the routine's connections to other routines small, intimate, visible, and flexible?
- ☐ Is the length of routine determined naturally by its function and logic, rather than by an artificial coding standard?

Parameter-Passing Issues

- ☐ Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- ☐ Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- ☐ Are interface assumptions documented?
- ☐ Does the routine have seven or fewer parameters?
- ☐ Is each input parameter used?
- ☐ Is each output parameter used?
- ☐ Does the routine avoid using input parameters as working variables?
- ☐ If the routine is a function, does it return a valid value under all possible circumstances?

CHECKLIST: Naming Variables

General Naming Considerations

- ☐ Does the name fully and accurately describe what the variable represents?
- ☐ Does the name refer to the real-worls problem rather than to the programming-language solution?
- ☐ Is the name long enough that you don't have to puzzle it out?
- ☐ Are computed-value qualifiers, if any, at the end of the name?
- ☐ Does the name use *Count* or *Index* instead of *Num*?

Naming Specific Kinds of Data

- ☐ Are loop index names meaningful (something other than *i*, *j*, or *k* if the loop is more than one or two lines long or is nested)?
- ☐ Have all "temporary" variables been renamed to something more meaningful?
- ☐ Are boolean variables named so that their meanings when they're *true* are clear?
- ☐ Do enumerated-type names include a prefix or suffix that indicates the category – for example, *Color_* for *Color_Red*, *Color_Green*, *Color_Blue*, and so on?
- ☐ Are named constants named for the abstract entities they represent rather than the numbers they refer to?

Naming Conventions

- ☐ Does the convention distinguish among local, class, and global data?
- ☐ Does the convention distinguish among type names, named constants, enumerated types, and variables?
- ☐ Does the convention identify input-only parameters to routines in languages that don't enforce them?
- ☐ Is the convention as compatible as possible with standard conventions for the language?
- ☐ Are names formatted for readability?

Short Names

- ☐ Does the code use long names (unless it's necessary to use short ones)?
- ☐ Does the code avoid abbreviations that save only one character?
- ☐ Are all words abbreviated consistently?
- ☐ Are the names pronounceable?
- ☐ Are names that could be misread or mispronounced avoided?
- ☐ Are short names documented in translation tables?

Common Naming Problems: Have You Avoided...

- ☐ ... names that are misleading?
- ☐ ... names with similar meanings?
- ☐ ... names that are different by only one or two characters?
- ☐ ... names that sound similar?
- ☐ ... names that use numerals?
- ☐ ... names intentionally misspelled to make them shorter?
- ☐ ... names that are commonly misspelled in English?
- ☐ ... names that conflict with standard library routine names or with predefined variable names?
- ☐ ... totally arbitrary names?
- ☐ ... hard-to-read characters?

CHECKLIST: Self-Documenting Code

Classes

- ☐ Does the class's interface present a consistent abstraction?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?

Routines

- ☐ Does each routine's name describe exactly what the routine does?
- ☐ Does each routine perform one well-defined task?
- ☐ Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
- ☐ Is each routine's interface obvious and clear?

Data Names

- ☐ Are type names descriptive enough to help document data declarations?
- ☐ Are variables named well?
- ☐ Are variables used only for the purpose for which they're named?
- ☐ Are loop counters given more informative names than *i*, *j*, and *k*?
- ☐ Are well-named enumerated types used instead of makeshift flags or boolean variables?
- ☐ Are named constants used instead of magic numbers or magic strings?
- ☐ Do naming conventions distinguis among type names, enumerated types, named constants, local variables, class variables, and global variables?

Data Organization

- ☐ Are extra variables used for clarity when needed?
- ☐ Are references to variables close together?
- ☐ Are data types simple so that they minimize complexity?
- ☐ Is complicated data accessed through abstract access routines (abstract data types)?

Control

- ☐ Is the nominal path through the code clear?
- ☐ Are related statements grouped together?
- ☐ Have relatively independent groups of statements been packaged into their own routines?
- ☐ Does the normal case follow the *if* rather than the *else*?
- ☐ Are control structures simple so that they minimize complexity?
- ☐ Does each loop perform one and only one function, as a well-defined routine would?
- ☐ Is nesting minimized?
- ☐ Have boolean expressions been simplified by using additional boolean variables, boolean functions, and decision tables?

Layout

- ☐ Does the program's layout show its logical structure?

Design

- ☐ Is the code straightforward, and does it avoid cleverness?
- ☐ Are implementation details hidden as much as possible?
- ☐ Is the program written in terms of the problem domain as much as possible rather than in terms of computer-science or programming-language structures?

CHECKLIST: Good Commenting Technique

General

- ☐ Can someone pick up the code and immediately start to understand it?
- ☐ Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
- ☐ Is the Pseudocode Programming Process used to reduce commenting time?
- ☐ Has tricky code been rewritten rather than commented?
- ☐ Are comments up to date?
- ☐ Are comments clear and correct?
- ☐ Does the commenting style allow comments to be easily modified?

Statements and Paragraphs

- ☐ Does the code avoid endline comments?
- ☐ Do comments focus on *why* rather than *how*?
- ☐ Do comments prepare the reader for the code to follow?
- ☐ Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- ☐ Are surprises documented?
- ☐ Have abbreviations beeen avoided?
- ☐ Is the distinction between major and minor comments clear?
- ☐ Is code that works around an error or undocumented feature commented?

Data Declarations

- ☐ Are units on data declarations commented?
- ☐ Are the ranges of values on numeric data commented?
- ☐ Are coded meanings commented?
- ☐ Are limitations on input data commented?
- ☐ Are flags documented to the bit level?
- ☐ Has each global variable been commented where it is declared?
- ☐ Has each global variable been identified as such at each usage, by a naming convention, a comment, or both?
- ☐ Are magic numbers replaced with named constants or variables rather than just documented?

Control Structures

- ☐ Is each control statement commented?
- ☐ Are the ends of long or complex control structures commented, or when possible, simplified so that they don't need comments?

Routines

- ☐ Is the purpose of each routine commented?
- ☐ Are other facts about each routine given in comments, when relevant, including input and output data, interface assumptions, limitations, error corrections, global effects, and sources of algorithms?

Files, Classes, and Programs

- ☐ Does the program have a short document, such as that described in the Book Paradigm, that gives an overall view of how the program is organized?
- ☐ Is the purpose of each file described?
- ☐ Are the author's name, e-mail address, and phone numbers in the listing?

Index