TTK4145 - Compendium

Sander Furre

May 27, 2020

NTNU | Faculty of Information Technology and Electrical Engineering

Part I

Software quality

Ultimate SW quality metric: **maintainability**. **Readability**, among other things, is also important.

1 Learning Goals

- Be able to write software following selected Code Complete Checklists for modules, functions, variables and comments
- Be able to criticize program code based on the same checklists

2 Modules/Classes checklist

2.1 Abstraction

- Does the class have a central purpose?
- Is the class well-named?
- Does the class's interface present a consistent abstraction?
- Does the class's interface make obvious how you should use the class?
- Is the class's interface abstract enough that you can treat it like a black box?
- Are the class's services complete enough?
- Has unrelated information been moved out of the class?
- Have you thought about subdividing the class into component classes, 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?

2.2 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?

- Dos the class avoid making assumptions about its users, including its derived classes?
- Is the class independent of other classes? Is it loosely coupled?

2.3 Inheritance

- Is inheritance used only to model "is a" relationships?
- Does the class documentation describe the 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?

2.4 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?

3 High-Quality Routines checklist

Key points

- Most important reason for creating routines is to improve readability, reliability and modifiability.
- A routine name is indication of its quality.

3.1 Big-Picture Issues

- Is the reason for creating the routine sufficient?
- Have all parts of the routine that would benefit form 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?
- Does the routine have loose coupling - are the routine's connections to other routines small, intimate, visible and flexible?
- Is the length of the routine determined naturally by its function and logic, rather than by artificial coding standard?

3.2 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?

4 Naming variables checklist

4.1 General naming considerations

- Does the name fully and accurately describe what the variable represents?
- Does the name refer to the real-world 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?
  - Qualifiers are: Total, sum, Average, Max, Min, Record, String, Pointer etc.
- Does the name use Count or Index instead of Num?

4.2 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?

4.3 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?

4.4 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?

4.5 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?

5 Self-Documenting Code Checklist

5.1 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?

5.2 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?

5.3 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 distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

5.4 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)?

5.5 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?

5.6 Layout

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

5.7 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?

6 Good Commenting Technique Checklist

6.1 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?

6.2 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 been avoided?
- Is the distinction between major and minor comments clear?
- Is code that works around an error or undocumented feature commented?

6.3 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?
-

6.4 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?

6.5 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?

6.6 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 number in the listing?

Part II

Fault tolerance

7 Learning Goals:

- Understand/use: Reliability. Failure, fault, error. Failure Modes. Acceptance Test. Fault prevention vs. fault tolerance. Redundancy. Redundancy, static vs. dynamic. Forward/Backward error recovery
- Understand/use/evaluate techniques: N-version programming. Recovery blocks. Error detection. Failure mode merging. Acceptance tests

8 Reliability, Failure and faults

There are 4 sources of faults in embedded systems

1. Inadequate specification, i.e. misunderstanding the interactions between the program and the environment.
2. Design errors in software components.
3. Failure of hardware components.
4. Interference in the supporting communication subsystem.

Reliability is a measure of the success with which the system conforms to some specification of its behavior. **Failure** is when the behavior of the system deviates from that which is specified for it. It in other words fail to follow specification. **Errors** are unexpected problems internal to the system which eventually manifests themselves in system's external behavior. **Faults** are the mechanical or software-related cause of failure. A failure in one (sub)system can cause faults in other systems:

$$\text{Fault} \xrightarrow{\text{activation}} \text{Error} \xrightarrow{\text{propagation}} \text{Failure} \xrightarrow{\text{causation}} \text{Fault}$$

Fault types

- Transient faults: Occur at some point in time, then disappear at some later point in time. E.g. hardware response to external electromagnetic field, fault disappears when the field disappears. Common kind of fault in communication systems.
- Permanent faults: Occur at some point in time, lasts indefinitely until repair. E.g. software design faults.

- Intermittent faults: Transient faults that sometimes happen and sometimes not. E.g. fault due to heat on heat sensitive hardware component.

Software faults are also called Bugs and we have two types:

- Bohrbugs: Reproducible, usually identifiable. Can be found and fixed with testing usually or diversity techniques.
- Heisenbugs: Only activate under certain unknown, rare circumstances. Usually not found during testing. E.g. code shared between concurrent tasks that is not properly synchronized. Causes can be software ageing i.e. not freeing allocated memory.

8.1 Failure modes

In short, failure modes describe the ways a (sub)system can fail.

We have two general classes of failure modes

- Value failure: Error in the value associated with service. Can be the result of a data conversion for example. The two categories are:
  - Constraint error: A fault causes a constraint to be breached.
  - Value error: A fault causes a value to be incorrect.
- Time failure: Service not delivered at the correct time. The thee categories are:
  - Too early
  - Too late (Performance error)
  - Infinitely late (Omission error)

Combination is called arbitrary failures. We also have **Commission/Impromptu** failure which describes a situation where a service delivered is not expected.

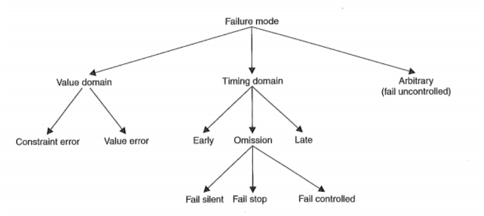


Figure 2.2 Failure mode classification.

Based on these failure modes we can sum up how the system might fail.

- Fail uncontrolled: Arbitrary or impromptu errors
- Fail late: delivers service too late
- Fail silent: Omission failure
- Fail stop: Omission failure, but other systems can detect it.
- Fail controlled: fails in a specified controlled manner.
- Fail never: Self-explanatory.

9 Fault prevention and fault tolerance

Fault prevention aims to eliminate any and all faults before the system goes into operation, whilst **fault tolerance** enables the system to continue functioning even in the presence of faults. Both approaches attempt to give the system well-defined failure modes.

9.1 Fault prevention

Consists of two stages;

- Fault avoidance - Achieved by choosing reliable components, eliminating interference, having a complete requirements specification, following strict design methodologies and using languages with data abstraction and modularity, like Ada and Java.
- Fault removal - Removing causes of errors mainly by system testing. Note that system testing is difficult.

Any system will fail eventually either in hardware or software, and for real-time systems we therefore need fault tolerance.

9.2 Fault tolerance

Three levels of fault tolerance

1. **Full fault tolerance** - The system continues to operate in the presence of faults for a limited time period with no significant loss of functionality or performance. Required in most safety-critical systems in theory.
2. **Graceful degradation** - also known as fail soft. The system continues to operate in presence of errors, accepting a partial degradation of functionality or performance until recovery or repair. Safety-critical systems often settle for this type of fault tolerance in practice. Graceful degradation necessity for applications with high availability requirements.

3. **Fail safe** - The system maintains its integrity while accepting temporary halt in its operation. In other words, it shuts down safely temporarily to allow for maintenance.

The goal of fault tolerance is to minimize redundancy while maximizing reliability subject to cost, size and power constraints of the system.

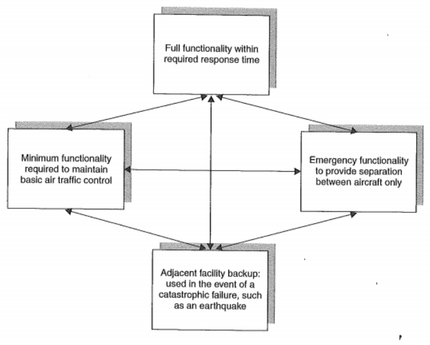

Full functionality within required response time
Minimum functionality required to maintain basic air traffic control
Emergency functionality to provide separation between aircraft only
Adjacent facility backup: used in the event of a catastrophic failure, such as an earthquake

Figure 2.3 Graceful degradation and recovery in an air traffic control system.

9.3 Redundancy

Protective redundancy introduces components that detects and recovers the systems from faults, but are unnecessary for normal operation. When designing a fault-tolerant system, the goal is to minimize redundancy while maximizing reliability, subject to constraints on cost, size and power consumption. The redundant components can (and will) increase complexity, and it is useful to separate them from the rest of the system. We separate between static and dynamic redundancy, both for hardware and software. First, let's have a look at hardware redundancy:

- Static redundancy (or masking) is based on redundant components "hiding" faults. An example is Triple Modular Redundancy (TMR), where a majority voting circuit is used. The output of three identical components are compared, and if one differs from the others, its output is masked out. It is assumed that faults are transient.

- Dynamic redundancy is an error-detection facility within a component, making it possible for that component to indicate if its output is in error. Note that the component does not hide or fix the error, that must be done by some other part of the system. Examples are checksums (see parity byte section on Wikipedia for very simple example) and parity bits.

For fault tolerance with regards to software design, we have *N-version programming* which works like masking, and *error detection and recovery*.

The latter is dynamic redundancy in the sense that recovery is only brought into action once an error has occurred.

10 N-version programming

From one initial specification, N independent programs are created. In operation, they run concurrently, and their outputs are compared by a driver process. The "correct" result is determined by majority of vote, like with masking.

Assumption: Program can be completely, consistently and unambiguously specified and independently developed programs will fail independently.

Example: On Boeing 777 flight control system, single Ada program was produced but three different processors and three distinct compilers were used to obtain diversity.

The driver process invokes the versions, waits for completion of voting, compares results and acts on these results.

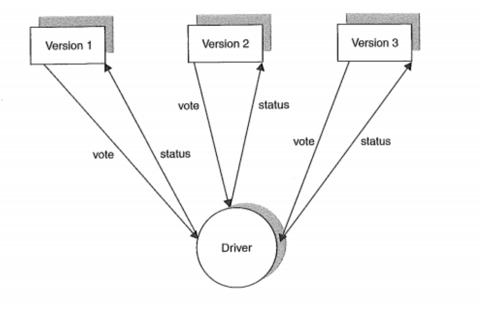


Figure 2.4 *N*-version programming.

In the context of embedded systems, we check the versions underway and not at the end. Interaction between the driver and the versions consists of 3 components

1. Comparison vectors: Also called votes, consists of output and metadata (attributes)
2. Comparison status indicators: Commands from driver. Indicates the actions that each version must perform as result of comparison. Can be *continuation*, *termination* or *continuation with vote modification* on some of the versions.
3. Comparison points: The points in the code where votes must be communicated to the driver process.

Challenges concerning N-version programming are:

- **Initial specification** - A specification error will manifest itself in all N versions of the implementation.
- **Independence of design effort** - The assumption that versions will fail completely independently of each other is not reality in practice.
- **Adequate budget** - N-version programming technique can require N times the budget of a 1 version technique.

11 Software dynamic redundancy

With dynamic redundancy, the redundant components only come into operation when an error manifests itself. There are four phases to dynamic redundancy in software:

1. **Error detection** - Most faults will manifest themselves in form of an error. No Fault tolerance scheme can be utilized until the error is detected.
2. **Error diagnosis** - There is a delay between a fault becoming active and error detection, the propagation of erroneous information in the system is assessed.
3. **Error recovery** - Transform the corrupted system into a state form which it can continue its normal operation (perhaps with degraded functionality).
4. **Fault treatment** - Maintenance must be performed to correct the underlying fault responsible for the error.

11.1 Error detection

There are two classes:

- **Environmental detection** - Detection by hardware or run-time support system. E.g. overflow error in hardware or out-of-bounds error for array in run-time)
- **Application detection** - Detect error with application specific code.
  - **Replication checks** - N-version programming with N=2
  - **Timing checks** - Watchdog or deadline checks
  - **Reversal checks** - Calculate input from output and check for match.
  - **Coding checks** - Test for corruption of data. Based on redundant data, i.e. checksum.

  - **Reasonableness checks** - Are based on knowledge of the internal design and construction of the system. I.e. range violation and assertions.
  - **Structural checks** - Metadata about structures

Note that all forms of application detection mentioned can be implemented in hardware and are then considered to be environmental detection.

11.2 Error diagnosis

Software designers aim to minimize the damage caused by a faulty component, this is called firewalling. Two techniques are:

- **Modular decomposition** - Modules only communicate through well defined interfaces, internal details are hidden. Provides a static structure.
- **Atomic actions** - are indivisible, and appear to happen instantaneously for the rest of the system. Often called **transactions** or atomic transactions. They are used to move the system from one consistent state to another, and limit the flow of information between components/modules.

11.3 Error recovery

The two approaches here are forward and backward error recovery.

- **Forward Error Recovery** - Attempts to continue from the corrupted state by making selective corrections to the state.
- **Backward Error Recovery** - Relies on restoring the system to a safe state previous to that in which the error occurred. This safe state is called an recovery point and the act of establishing it is called **checkpointing**.

Advantage of backward error recovery is that it does not rely on finding the location or cause of faults. Can therefore be used to recover from unanticipated faults including design errors. A disadvantage on the other hand is that it cannot undo effects the fault may have had on the environments (missile launch) and it can be time-consuming. Another disadvantage is the possibility of the domino effect.

The domino effect is a possible issue in state restoration with concurrent processes. Consider the following scenario in Figure **??**.

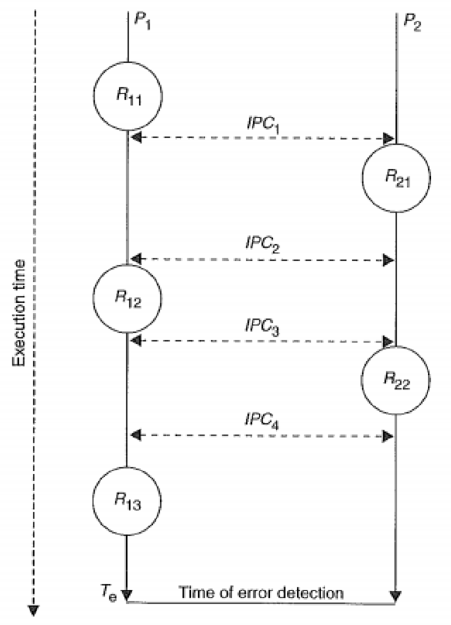


Figure 2.6 The domino effect.

If P1 detects an error at Te then it is simply rolled back to R13. If on the other hand P2 detects an error at Te it is rolled back to R22 and thus must undo IPC4. This means that P1 must be rolled back to R12 which means it must undo IPC3 which means P2 must rollback to R21 and undo IPC2 and so on. Potentially the two processes have to be rolled back to the beginning of their interaction. The only way to be guaranteed to stop this is with **recovery lines** which are consistent set of recovery points across all processes that communicates together.

12 Recovery blocks and acceptance tests

Recovery blocks are normal blocks in the programming language sense but where the entrance is an automatic recovery point and the exit an acceptance test.

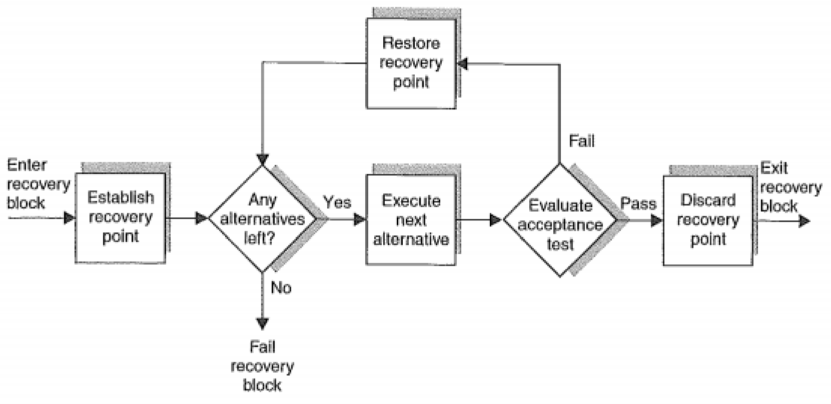


Figure 2.7 Recovery block mechanism.

12.1 Acceptance test

The acceptance test is used to test that system is in an acceptable state after the execution of the block (also called primary module). If it fails program is restored to recovery point at beginning of block and an alternative module is being executed. If all alternative modules fail the block fails.

One should keep in mind the tradeoff between comprehensive testing and keeping overhead at a minimum. All error detection techniques mentioned earlier may be used as acceptance tests.

12.2 Comparison between N-version programming and recovery blocks

- **Static vs. Dynamic**-version is static because all versions run all the time. Recovery blocks are dynamic because the alternative code only runs when acceptance test fails.
- **Associated overhead** - Both methods increase overhead/complexity.
- **Diversity of design** - Both exploit this and are therefore susceptible to errors that originate in the specification.
- **Error detection** - N-version uses vote and recovery blocks uses acceptance test. Exact voting less overhead than acceptance testing. Same cannot be concluded for inexact voting.
- **Atomicity** - Backward error recovery criticized because cannot undo effects or damage which may have occurred in the environment. N-version programming avoids this because all versions are assumed to not interfere with each other; they are atomic. They communicate only with the driver through voting. Not that it is possible to structure

program such that unrecoverable operations do not occur in recovery blocks.

Note that N-version programming and recovery blocks can be considered competing techniques, but also complementary techniques.

13 Dynamic redundancy and exceptions

The goal if this section is to introduce framework for implementing software fault tolerance based on dynamic redundancy and the notion of exceptions and exception handlers.

An **exception** is the occurrence of an error. **Raising** (or throwing or signaling) an exception is bringing the exception condition to the attention of the invoker of the operation which caused the exception. In less formal wording, this means letting the program know an error has occurred. **Handling** (or catching) an exception is the invoker's response to the raised exception.

Note that exception handling is a forward error recovery mechanism, but it can also be used to provide backward error recovery.

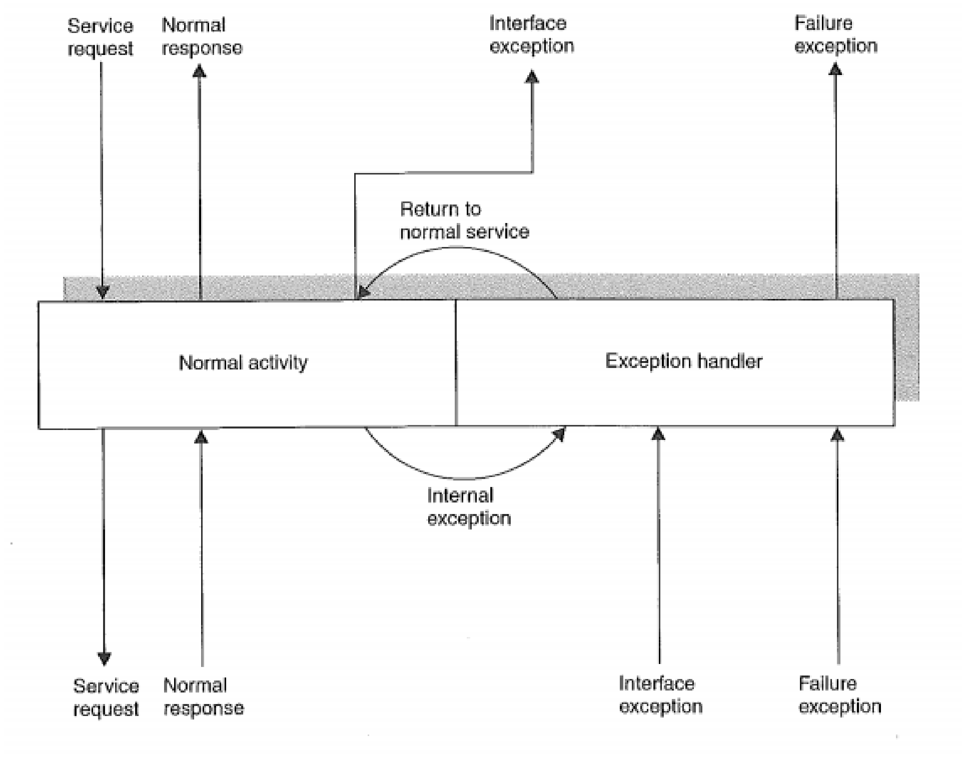


Figure 2.8 An ideal fault-tolerant component.

Part III

Fault model and software fault masking

14 Learning goals

- Understanding of the three cases in low level design for fault tolerance by redundancy: Storage, Computation and Communication.
- Understanding of the work method:
  1. Find error model
  2. Detect errors and merge failure modes (+ error injection for testing)
  3. Handling or masking with redundancy. Aiming for progression of failfast, reliable and available systems.
- Ability to implement (simple) Process-Pairs-like systems

15 Case 1: Storage

Imagine an array of data. Assume unreliable read and write functions.

15.1 Failure modes

status read(addr,data)

- Wrong data
- Old data
- Data from wrong address
- Fail

status write(addr,data)

- Writes wrong data
- Writes to wrong address
- Does not write
- Fail

15.2 Detection and merging error modes.

For detection write also address, checksum, versionID and statusBit telling if block is OK. The status bit can be used for injecting errors. To simplify we merge all error modes to one, and consider all as the *Fail* error mode.

15.3 Redundancy

Many options here, some include

- More copies or hard-drives
- If read fails, write a safe state back to memory.
- A repair thread reads regularly to maintain data.

16 Case 2: Messages

16.1 Failure modes

- Lost
- Delayed (too late or out of order)
- Corrupted
- Duplicated
- Wrong recipient

16.2 Detection and merging error modes

For detection include SessionID, checksum, acknowledgements and sequence numbers. To simplify error modes all errors are considered as *lost message*.

16.3 Handling with redundancy

Timeout and resend the message.

17 Case 3: Processes/calculations

17.1 Failure modes

Process does not do the next correct side effect.

17.2 Detection and merging error modes

Failed acceptance test used for detection typically if uses dynamic redundancy. Should be mentioned that we typically structure code such that we do acceptance test and only execute side effect if it is successfully. To simplify we merge all error modes into a Panic/Stop error mode.

17.3 Handling with redundancy

- **Checkpoint restart** - Writing state to storage after successful acceptance test. Yields error containment but requires good acceptance test.
- **Process pairs** - Two processes, primary and backup. Primary does work, sends "IAmAlive" and checkpoints to backup. Backup takes over if primary fails. Does not rely on a safe communication line because re-sending can be used to mask it out.
- **Persistent processes** - This is not really relevant for embedded systems because requires infrastructure for reliable storage, communication and calculations. It is a transactional, stateless infrastructure where all calculations are transactions and the processes are stateless.

Part IV

Transaction fundamentals

Disclaimer: This part is not very good pedagogically. It is unclear what the context is for the terminology and methodology used. It seems to be based around JAVA infrastructure or something. The notes I have therefore are unable to give any decent intuition and are more used as a high-level explanatory thingy for when questions regarding this appears on exams.

18 Learning Goals

- Knowledge of eight "design patterns" (Locking, Two-Phase Commit, Transaction Manager, Resource Manager, Log, Checkpoints, Log Manager, Lock Manager), how they work and which problems they solve. Ability to utilize these patterns in high-level design.
- Comprehension of terms: Optimistic Concurrency Control, Two-phase commit optimizations, Heuristic Transactions, Interposition.

19 Motivation and context

- We need error handling and containment for systems with multiple participants (threads, processes, distributed systems). These participants must often cooperate in the error handling.
- Transactions (and atomic actions) are techniques/frameworks that provide the means to do this. They fall under the category of dynamic SW redundancy.
- Transactions contribute towards the desired error assessment and confinement design and help avoid the *domino effect*

We will learn Transaction fundamentals through **eight design patterns** and some additional terms. Before we get into that note that a transaction is an indivisible, atomic action with backward error recovery. The properties of a transaction described by *ACID properties*

- **Atomicity** - The transaction either commits successfully or aborts completely.
- **Consistency** - Transactions preserve a consistent state.
- **Isolation** - Intermediate states during a transaction are not visible from the outside. Furthermore, transactions appear to be executing

serially even when they are not.

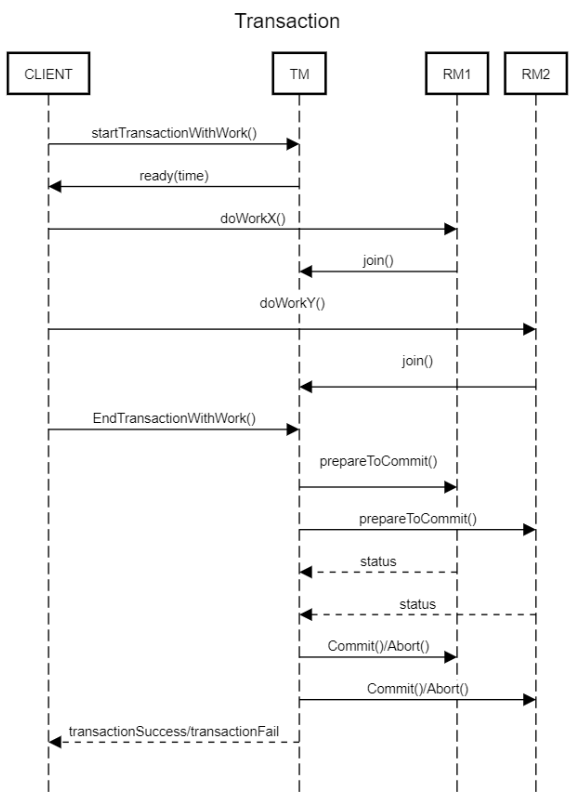
- **Durability** - The effects of a commited transition are never lost, they are stored in stable storage such as a hard drive.

With backward error recovery the flow of control of a single-threaded program is as follows:

```
allocate locks
create recovery point
do work(goto end if any problems)
label end:
if (error) {
roll back to recovery point
    status=FAIL
} else {
    status=OK
}
release locks
return status
```

It should be noted that the acceptance test that is responsible for detecting the errors in the program above.

The workings of a transaction can be further visualized by a sequence diagram. It can be a good idea to read about the eight design patterns and come back to this figure.



20 The 8 design patterns

20.1 Two-phase commits

Associated with each transaction is a coordinator/Transaction Manager, that communicates with the participants. The message flow is as follows:

```
Coordinator                                         Participant
                             QUERY TO COMMIT
                 -------------------------------->
                             VOTE YES/NO            prepare*/abort*
                 <-------------------------------
commit*/abort*               COMMIT/ROLLBACK
                 -------------------------------->
                             ACKNOWLEDGMENT         commit*/abort*
                 <--------------------------------
end
```

20.1.1 Basic Algorithm

Commit request (or voting) phase

1. Coordinator sends **query to commit** and waits until has received reply from all participants
2. Participants execute transaction up to the point where they will be asked to commit. Each write entry to undo lo and redo log.
3. Participants vote **Yes** if their actions succeeded or **No** if they have experienced a failure making it impossible to commit.

Commit/Completion phase

- **Success** - All participants voted yes. Coordinator responds with commit instruction.
- **Failure** - At least one participant voted no. Coordinator responds with rollback instruction.

Note that: A disadvantage with two-phase commits is that the protocol is blocking. Participants will block after they have voted, awaiting a commit or rollback message. If the coordinator fails, they will never receive either.

20.2 Locking

Ensures that intermediate states are not propagated. Additionally, ensures that participants can only communicate with other participants. The participant must be part of the ongoing transaction to be able to use the locked resource.

20.3 Lock Manager

Can have the following functionality:

- Release all locks associated with transaction X when it finishes.
- Tidy up after restart (open locks that should be open etc.)
- Handle resources shared by several Resource Managers.
- Include deadlock avoidance and detection algorithms.

Can also have other functions relating to locks.

20.4 Resource Manager

The resource managers are the *transaction participants*.

- Keeps track of locks and recovery points
- Communicates with transaction manager during voting and commit phases.

20.5 Transaction Manager

Useful if there are multiple participants

- Creates the transaction, keeps track of participants
- Plays the role of coordinator
- Can provide a transaction deadline and abort if the deadline is missed.

20.6 The log

- Alternative to checkpoints
- Every participant including the TM writes every planned change of state to the log and waits until the operation is confirmed safe to perform it.
- Processes can restore state after restart by executing the logrecords.
- Logrecords can be executed backwards to undo actions, therefore no need for checkpoints.

20.7 The Log manager

Queues logrecords, optimizes disk access and sends acknowledgments receipts on received log records.

20.8 Checkpoints

Not the recovery points discussed before, checkpoints are a smart way to stop the log from growing infinitely large. Once in a while a complete state, including a list of all active transactions is written to the log. This is a checkpoint. We can then delete log entries older than this checkpoint.

21 Other terms

21.1 Optimistic concurrency control:

This method assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use resources without utilizing locks or other synchronization primitives. This works because before committing, each transaction verifies that no other transaction has modified the data it has read. If conflicting modifications are revealed, the transaction is rolled back and restarted. OCC can work well when we have low data contention, meaning when conflicts are rare

21.2 Two-phase commit optimizations

Different methods exist here

- Early abort using e.g. ATC
- One-phase when only one participant
- Read-only operations will never be aborted thus will always yield commit.
- Last resource commit: One participant gets to wait until all the other votes are in and take these into consideration before making its vote.

21.3 Heuristic transactions

What if we give our vote and then loose connection? We have to make local guess and maybe to forward error recovery after.

21.4 Interposition

The TM's ability to play the role of a RM.

Part V

Atomic actions

22 Learning goals

- A thorough understanding of the problems Atomic Actions are meant to solve and how these motivates the different aspects of Atomic Actions.
- Ability to use and implement Atomic Actions, including the mechanisms providing the start, side and end boundaries.
- Understanding the motivation for using Asynchronous Notification in Atomic Actions.
- A coarse knowledge of how the mechanisms for Asynchronous Notification in C/Posix, ADA and Java works.

23 What are atomic actions

23.1 Motivation for using atomic actions

An Atomic Action is a design framework for damage confinement and error recovery. Before we go into the actual details, lets just give a heads up and say that a transaction discussed in the last part is in fact an atomic action structure with backward error recovery. We know from before that should have *recovery line* to avoid domino effect if we have backward error recovery. The question is how we achieve synchronization of recovery points. With atomic actions, we get the structure necessary to achieve both backward and forward error recovery. The recovery synchronization is not a straight line here, it is a registration of when a task enters an atomic action and if something goes wrong within it, all tasks in it are rolled back to the state they were in when they entered the atomic action. The general usage of atomic cations is for when several groups of concurrent tasks need to be structured to allow for coordination of their activities. The **atomic action** is required for each group of tasks to execute their joint activity. It should be mentioned that a single task may also want to protect itself from interference from other tasks. Atomic actions can therefore be said to involve one or more tasks and be a method for structuring these tasks such that they can operate on different parts of a single task concurrently. Figure **??** shows a simplistic representation of how atomic actions work.

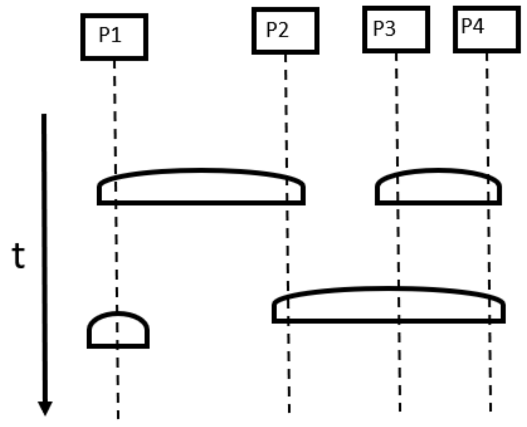


Figure 1: Illustration of how atomic actions can work

23.2 Different ways to define atomic actions

1. An action is atomic if the tasks performing it are not aware of the existence of any other active tasks, and no other active tasks is aware of the activity of the tasks during the time the tasks are performing the action.
2. An action is atomic if the tasks performing it do not communicate with other tasks while the action is being performed.
3. An atomic action has tasks that cannot detect state change except those performed by themselves and they do not reveal their state changes until the action is complete.
4. Atomic actions are by other tasks considered to be indivisible and instantaneous such that the effects on the system are as if they were interleaved as opposed to concurrent.

23.3 Requirements for atomic actions

- **Well-defined boundaries** - Each atomic action should have a *start*, an *end* and a *side boundary*. Start boundary is the location in each task involved in the atomic action where the action is deemed to start. The end boundary is where it is deemed to end and the side boundary separates tasks involved with the atomic action from those not involved.
- **Indivisibility (isolation)** - an atomic action must not allow the exchange of any information between the tasks on each side of the side

boundary. If two atomic actions share data trough a Resource Manager then the value is determined by strict sequencing. There is no implied *synchronization* at the start of an atomic action, but 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. In general only *strict nesting* allowed where one is completely contained within the other.
- **Concurrency** - It should be possible to execute different atomic actions concurrently

23.4 Standard Atomic Actions implementation

Transaction mentioned in last part is a good example of an atomic action. Start boundary:

1. Dynamic (call to e.g. *startTransactionWithWork()*

Side boundary:

1. Locking, no unlocking before safe state (end boundary).

End boundary:

1. Vote counting (two-phase commit)
2. Synchronization primitive barrier that no one can pas through before everyone has arrived.

24 Language-specific atomic action implementation

Fill out more here

Java:

- Synchronized methods, wait, notify / notifyAll
- Asynchronous exceptions

Ada:

- Protected objects
- Function procedures and entries with guards.

C/POSIX

- pthread cancel
- setjmp and longjmp

25 Asynchronous notification

Asynchronous notification is a bit like interrupts in hardware. It is a way to distribute error information instead of using the commit structure discussed or some polling regime to make forward error recovery possible instead of just backward error recovery. To get a recoverable action we must be able to get the attention of a task involved in the action that an error has occurred in another task involved in the action. This is achieved with an asynchronous notification mechanism which most programming languages and operating systems support. As with exceptions, there are two basic models; **resumption** and **termination**.

25.1 Resumption (event handling)

Behaves like a software interrupt. A task indicates which events it is willing to handle; when the event is signaled the task is interrupted and an event handler is executed. Afterwards the task resumes from where it was when the interrupt came in.

25.2 Termination model

Each task specifies a domain of execution during which it is prepared to receive an asynchronous notification that will cause the domain to be terminated. In less formal wording, this means that after we handle the error we terminate what we were doing and start somewhere else in the code, for example lower down.

25.3 The user need for asynchronous notification

Fundamental need is to enable tasks to respond *quickly* to a condition which has been detected by another task. There are several occasions where polling for events or waiting for the event to occur is inadequate:

- **Error recovery** - When groups of tasks undertake *atomic actions*, an error detected in one task requires all other tasks to participate in the recovery as the work they are now doing might be useless if the error has propagated. The form of asynchronous notification we will use here for termination is *Asynchronous Transfer of Control (ATC)*

25.4 Language specific asynchronous notification implementation

25.4.1 C/POSIX

Supports the resumption model. Has a set of predefined signals and 3 ways in which a process can deal with a signal.

1. **Block** - Handle it later or accept it
2. **Handle** - Setting a function to be called whenever it occurs
3. **Ignore** - The signal is simply lost

If we want asynchronous transfer of control we can use **setjmp** and **longjmp** mechanisms.

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");      // prints
    longjmp(buf,1);  // jumps back to where setjmp was called
}

void first(void) {
    second();
    printf("first\n");  // does not print
}

int main() {
    if ( ! setjmp(buf) ) {
        first();   // when executed, setjmp returns 0
    } else {  // when longjmp jumps back, setjmp returns 1
        printf("main\n"); // prints
    }
    return 0;
}
```

When program above executed it will return "second main"

25.4.2 Asynchronous notification in Ada

The asynchronous notification facilities in Ada allow an application to respond to among other things asynchronous transfer of control (ATC) requests on a task - supporting a termination model. There is no generalized model for resumption in Ada. To emphasize that ATC is a form of communication and synchronization, the mechanism is built on top of the inter-task communication facility.

We will look at of *Asynchronous select* here.

```
select
    Trigger.Event;
    -- optional sequence of statements to be
    -- executed after the event has been received
then abort
    -- abortable sequence of statements
end select;
```

It is best explained with an example

```
task Server is
        entry ATC_EVENT;
end Server;

task To_Be-Interrupted;

task body Server is
begin
        ...
    accept ATC_EVENT do
        Seq2;
    end ATC_EVENT;
    ...
end Server;

task body To_Be_Interrupted is
begin
        ...
    select -- ATC statement
        Server.ATC_EVENT;
        Seq3;
    then abort
        Seq1;
        -- the work here is the normal operation work
        -- Will be interrupted if it times out or event happens
    end select;
    Seq4;
    -- Seq4 comes after the asynchronous select, it cannot be interrupted
    ...
end To_Be_Interrupted;
```

When the above ATC statement is executed, the statements which are executed will depend on the order of events that occur. See page 257 in BW chapter 7. The Ada ATC facility described above is used to implement both forward and backward error recovery.

25.4.3 Asynchronous notification in Java

ATC here also. Focus on Real-Time Java.

- Built on Java exceptions
- Can be blocked

Part VI

Shared variable synchronization

1 Learning goals

- Ability to create (error free) multi thread programs with shared variable synchronization.
- Thorough understanding of pitfalls, patterns, and standard applications of shared variable synchronization
- Understanding of synchronization mechanisms in the context of the kernel/HW
- Ability to correctly use the synchronization mechanisms in POSIX, ADA (incl. knowledge of re-queue and entry families) and Java.

2 Motivation

The major difficulties associated with concurrent programming arise from task interactions. For tasks to run concurrently they must synchronize and communicate. Synchronization can also be considered to be a content-less form of communication.

Inter-task communication based upon **shared variables** or **message passing**. In this part we will look at shared variable-based communication and synchronization primitives. In particular, we will get into *busy waiting*, *semaphores*,*conditional critical regions*, *monitors*, *protected types* and *synchronized methods.* We will not discuss message-based synchronization.

3 Mutual exclusion and condition synchronization

A sequence of statements that must appear to be executed indivisibly is called a **critical section** and the synchronization required to protect a critical section is called **mutual exclusion**. Mutual exclusion is however not the only form of synchronization needed. If two tasks do not share any variables there is no need of mutual exclusion. **Conditional synchronization** is another form of synchronization where a task that wishes to perform an operation must wait until some other tasks finishes their operations first. I.e. lets look at a **producer-consumer** system where a buffer links two tasks. If the buffer is finite we need two condition synchronizations, **Buffer full** to stop producer from depositing to a full buffer and **Buffer empty** to stop consumer from extracting form empty buffer. We must also have mutual exclusion naturally. There are many synchronization primitives to make these methods reality, where the basic one is **busy-wait** loops with **flags** (only works well for conditional synchronization).

4 Busy Waiting

Waiting loop, if flag not yet set loops round and rechecks the flag. This is very inefficient and wastes processor resources. An issue with busy waiting is that they can easily lead to **live locks**. This is an error condition where tasks get stuck in their busy-wait loops and are unable to make progress. The key takeaway with synchronizing only with mutual exclusion and busy waiting cam be summarized as

- Protocols that use busy loops are difficult to design, understand and prove correct
- Testing programs may not examine rare interleavings
- Inefficiency
- An unreliable (rouge) task that misuses shared variables will corrupt the entire system.

5 Suspend and resume

Alternative to busy waiting that does not waste as much processor time. Strategy here is to suspend waiting tasks until they can proceed. One-stage suspend and resume unfortunately suffers from **data race condition**. Consider the case of two threads T1 and T2 and a flag. Assume we start with the flag down and that T1 runs and checks the flag before the OS preempts T1 and runs T2. T2 sets the flag and resumes T1. T1 is not suspended so the resume has no effect. When T1 next runs it thinks the flag is down, because it evaluated it before it was preempted and therefore suspends itself all though the flag is actually up. It will never be resumed again

There are several ways to prevent data race conditions, they all have some **two-stage suspend** operation where a thread or process mus announce that it plans to suspend in the near future before actually doing so. Note that although suspend and resume are useful low-level primitives, no operating system or language relies solely on these mechanisms for mutual exclusion and condition synchronization.

6 Semaphores

Semaphores are a simple mechanism for programming mutual exclusion and condition synchronization. Originally designed by Dijkstra (1968) they have the following two benefits:

1. They simplify the protocols for synchronization
2. They remove the need for busy-wait loops

Definition: A semaphore is a non-negative integer variable that apart from initialization can only be acted upon by two procedures. These procedures are *wait* and *signal* in this course.

1. wait(S) - If the value of the semaphore S is greater than zero then decrements its value by one and continues to next line; otherwise delay the task until S is greater than zero and then decrements its value and continue.
2. Signal(S) - Increment the value of the semaphore S by unity.

General semaphores often called **counting semaphores**. Important to note that wait and signal are atomic actions.

6.1 Suspended tasks

All synchronization primitives deal with delay by some form of suspension; the task is removed from the set of executable tasks. When a task is asked to wait on a zero semaphore, the run-time support system (RTSS) removes the task from the processor and places it in a queue of suspended tasks on that particular semaphore. We should assume that this queue is in non-deterministic order if we are not actively scheduling tasks.

6.2 Implementation

RTSS scheduler programmed so that it does not swap out a task during wait or signal operation; they are **non-preemptible operations**. The RTSS will also typically disable interrupts during the operations too if single processor. For a multiprocessor system if two processes both use wait or signal on same semaphore, the RTSS cannot do anything about it and thus a *lock* mechanism must be used on the semaphore. It should be noted that semaphores can only provide mutual exclusion if memory locations exhibits the essence of mutual exclusion.

6.3 Liveness provision

The synchronization primitives introduces error conditions. **Deadlock** is the most serious. Similar to livelock, but tasks are suspended. It is in the nature of an interdependent concurrent program that usually once a subset of the tasks become deadlocked, all the other tasks will eventually become part of the deadlocked set. A less severe error condition is **indefinite postponement (starvation/lockout)**. Task tries to gain access to resource but is never allowed due to other tasks taking it. *If a task is free from livelocks, deadlocks and indefinite postponement then it is said to possess **liveness**.*

6.4 Criticism of semaphores

A real-time program built only upon semaphores is error-prone. Needs just one occurrence of a semaphore to be omitted or misplaced for the entire program to collapse at runtime. We need a more structured synchronization primitive.

7 Monitors

The main problem with conditional critical regions is that they can be dispersed throughout the program. **Monitors** are intended to alleviate this problem by providing more structured control regions. The intended critical regions are written as procedures and encapsulated together into a single module called a monitor. As a module, all variables that must be accessed under mutual exclusion are hidden and additionally all procedure calls into the module are guaranteed to execute with mutual exclusion. Although providing mutual exclusion, there is still need for condition synchronization within the monitor. We have **condition variable** within the monitor with *wait* and *signal* functions. To prevent two or more tasks to become active at the same time within the monitor, four different approaches are suggested

1. A signal is allowed only as the last action of a task before it leaves the monitor.
2. A signal operation has the side-effect of executing a return statement.
3. A signal operation which unblocks another taks has the effect of blocking itself.
4. A signal operation which unblocks another task does not block and the freed task must compete for access to the monitor once the signalling task exits.

7.1 Criticism of monitors

The monitor gives a structured and elegant solution to mutual exclusion problems such as the bounded buffer. It does not however, deal well with condition synchronizations, resorting to low-level semaphore-like primitives.

8 Synchronization mechanisms in POSIX, Java and Ada

8.1 Java

The synchronized keyword is used to make methods thread-safe. Consider the following class

```
public class MyClass {
        private int i;

    public MyClass(int initValue) {
       i = initValue;
    }

    public synchronized void increment() {
        i++;
    }
}
```

When a thread T1 executes increment, all other threads that use synchronized methods (functions) of the same class object suspend until thread T1 releases the monitor lock. Note that we must make the split between synchronized methods and other methods in the class. Only synchronized methods require a thread to gain access of the class object mutex. If the method we try to use is not synchronized it can be accessed without having the mutex. Hence *to obtain full mutual exclusion, every method has to be labeled synchronized*. *Wait()* is used to suspend the current thread, like this for example

```
public synchronized void conditionalIncrement(){
    while(i < 3) wait();
    i++
 }
```

since conditionalIncrement() is a synchronized method, thread T1 must hold the monitor lock (mutex) before it can invoke it. When it calls wait(), it releases the lock and suspends execution. Some time later it will be woken up again when it is *notified*

```
public synchronized void importantChange() {
   i = 3;
   notifyAll();
}
```

All suspended threads waiting on the lock are notified when notifyAll() is called. notifyAll() wakes up all threads wakes up all suspended threads, but it does not release the lock and all awoken threads must compete for it when it is released.

8.2 Ada

Protected objects are data types protected from inappropriate simultaneous access by multiple tasks. Protected objects are like monitors, they provide the structuring facility of monitors with the high-level synchronization mechanism of conditional critical regions. They are split into an interface definition and a operation implementation

```
protected type Counting_Semaphore is
    procedure Release;
    entry Acquire;
    function Lock_Count return Natural;
private
        Count : Natural := 0; --Natural number, i.e. geq 0
end Counting_Semaphore;


protected body Counting_Semaphore is
    procedure Release is
    begin
        if Count > 0 then
            Count := Count - 1;
        end if;
    end Release;

    entry Acquire when Count < 11 is
    begin
        Count := Count + 1;
    end Acquire

    function Lock_Count return Natural is
    begin
        return Count;
    end Lock_Count;
end Counting_Semaphore;
```

Take note of the following important aspects in Ada:

- Procedures can modify the state, and protected procedures guarantee mutual exclusion
- Entries are similar, but they have a boundary condition that must be true before they can execute. By using protected entry we can achieve condition synchronization.
- Functions cannot change the state in protected body, only read it. They can therefore have concurrent access, not mutually exclusive even though in protected body.

The *requeue* keyword is used in an entry body to delay the execution of certain statements in the body. The effect of a requeue statement is to remove a task that is executing in an entry and place it on a different queue. In the example below we place the task *My_Entry* on the *Wait_For_Data* queue if Data is not available.

```
entry My_Entry when My_Cond = True is
begin
    if Data_Available then
        Do_Work;
    else
        requeue Wait_For_Data;
    end if;
end My_Entry;
```

8.3 C/POSIX

Example useage of pthread with mutexes:

```
int i = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void increment() {
    for (int j = 0; j < 100000; j++) {
        pthread_mutex_lock(&mutex);
        i++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void decrement() {
    for (int j = 0; j < 100000; j++) {
        pthread_mutex_lock(&mutex);
        i--;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t A;
    pthread_t B;

    pthread_create(&A, NULL,increment, NULL);
    pthread_create(&B, NULL,decrement,NULL);

    pthread_join(A, NULL);
    pthread_join(B, NULL);

    printf("i = %d\n", i);
    return 0;
}
```

Part VII

Modeling of concurrent programs

9 Learning Goals

- An understanding of how using message-based synchronization leads to a very different design than shared variable synchronization.
- Model, in FSP and by drawing transition diagrams, simple programs (semaphore-based or message-passing).
- Draw very simple compound (for parallel processes) transition diagrams.
- Sketch simple message-passing programs.
- An understanding of how using message-based synchronization leads to fewer transition-diagram states than shared variable synchronization.
- Understanding the terms deadlock and livelock in context of transition diagrams

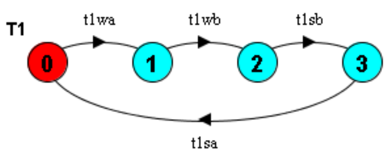
10 Modeling

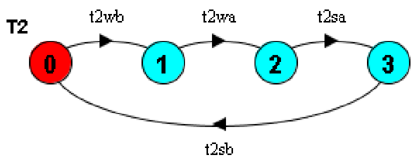
Classic deadlock example:

```
T1:
    while (1) {
        Wait(A)
        Wait(B)
        ...
        Signal(A)
        Signal(B)
    }
T2:
    while (1) {
        Wait(B)
        Wait(A)
        ...
        Signal(B)
        Signal(A)
    }
```

Let us introduce **processes** and **events**. All processes that 'care about' an event experience it at the same time. For modeling we use FSP (Finite State Processes) ⊂ CSP (Communicating Sequential Processes). Note that Ada `entries` can be modelled as events. For message-based systems (message passing, synchronous communication), modeling with processes, events and guards comes naturally. Some explanation of symbols before we get into the next part. The numbers below in the diagrams are the **states**. In the textual representation these are **arrows.** The text such as *t1sa* is an event. What this means is that to get for example an *t1sa* event, T1 process has to be in state 3 and SA process has to be in state 2. Check the diagrams below, this is a bit tricky but makes sense.

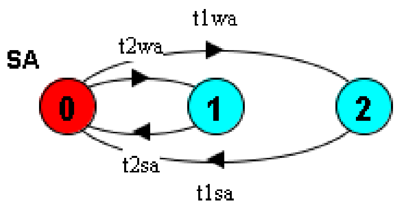
```
T1 = (t1wa -> t1wb -> t1sb -> t1sa -> T1).
T2 = (t2wb -> t2wa -> t2sa -> t2sb -> T2).
```

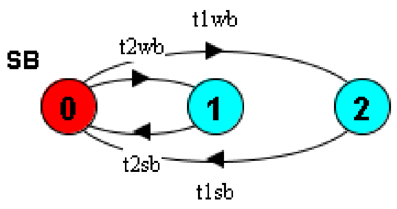




No deadlock is visible here, we need to model the semaphores:

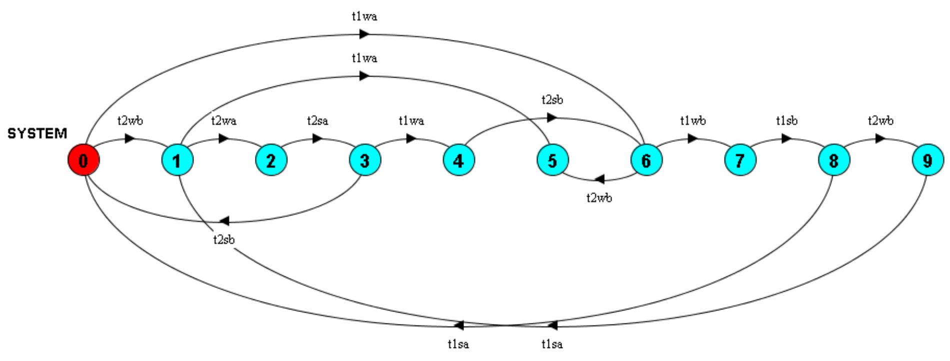
```
SA = (t1wa -> t1sa -> SA | t2wa -> t2sa -> SA).
SB = (t1wb -> t1sb -> SB | t2wb -> t2sb -> SB).
```





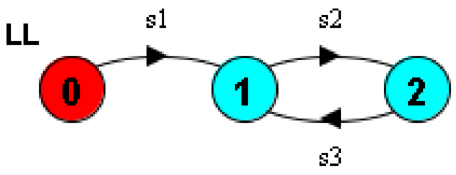
To model the complete, concurrent/parallel system:

```
||SYSTEM = (T1 || T2 || SA || SB).
```



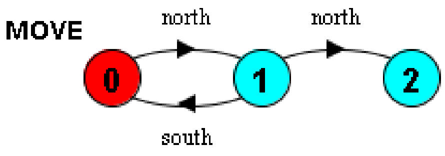
In a **transition diagram**, a deadlock is a state with no exit. Such a state can be observed in the figure above in state 5. A livelock is a subset of states we cannot exit.

```
LL = (s1 -> LOOP), LOOP = (s2 -> s3 -> LOOP).
```



Before mentioning the Dining philosophers problem, have a look at this basic code producing a deadlock

```
MOVE = (north->(south->MOVE|north->STOP)).
```



Dining philosophers with deadlock is shown next

```
    FORK = (get -> put -> FORK).
    PHIL = (sitdown -> right.get -> left.get -> eat ->
            right.put -> left.put -> arise -> PHIL).

    || DINERS(N=5) = forall [i:0..N-1]
            (phil[i]:PHIL ||
            {phil[i].left, phil[(i-1+N)%N].right}::FORK).
```

Some advance syntax is used here:

- `phil[i]:PHIL` exchanges `eat` with `phil1.eat`, and thus takes care of all events for all N philosophers.
- `phil[i].left, phil[(i-1+N)%N].right` means that either of the two options in curly braces are possible.
- `phil[i].left, phil[(i-1+N)%N].right::FORK` then means that either `phil[i].left.get` or `phil[(i-1+N) mod N].right.get` are possible for the same fork, i.e. it can be picked up by the philosopher to its right or left.

Transition diagram is huge, needs an analyzing tool such as the infamous LTS Analyzer to find the deadlock potential! Some notes on the Dining Philosophers problem:

- If we make some of the philosophers left-handed the deadlock is avoided. But hugely unfair as one philosopher next to the one causing the deadlock would have to finish his meal before the other philosopher gets both his forks. *This is not very well explained, sorry but I don't have a philosopher's stone that makes me godly enough to explain this well*
- A a resource hierarchy on the forks, where the philosophers would always pick up the lowest value fork first could improve fairness. *It is far beyond the scope of this compendium, this course, my sanity or the universe itself to properly explain why*

Part VIII

Scheduling

Concurrent programs exhibit non-determinism, and a real-time system needs to restrict this non-determinism through scheduling. A scheduling scheme provides two key features:

1. An algorithm for ordering the use of system resources
2. A way to predict the worst-case behavior of the system when the scheduling algorithm is applied.

There are **static** (prediction only before execution) and **dynamic** (run-time decisions are used) schemes for scheduling. The focus in this course is on static schemes, specifically *preemptive priority-based schemes on a single-processor system.* The task with the highest priority will always run, unless it is suspended or delayed for some reason. The two features above can be re-stated as:

1. Priority assignment algorithm
2. Schedulability test

10.1 What is non-preemptive scheduling (cyclic executive)

What is non-preemptive scheduling:

- Fixed set of tasks with fixed periods.
- Consists of a table of procedure calls, the majority cycle, compromised of smaller minor cycles with fixed duration.
- At run-time, no task can run concurrently, therefore mutual exclusion is guaranteed and we do not need synchronization primitives such as semaphores.
- All task periods must be a multiple of the minor cycle period.

What are some of its drawbacks?

- Difficult to incorporate sporadic tasks.
- Difficult to construct
- Difficult to incorporate tasks with long periods
- "Large" tasks will need to be split up into several procedures, which may hurt the code quality and make it more error-prone.

11 Task-based scheduling

Tasks exist at run-time, supported by real-time OS or run-time kernel. Each task is either *runnable/running*, *suspended waiting for timing event* or *suspended waiting for non-timing event*. There are different approaches to Task-based scheduling

- Fixed-Priority Scheduling (FPS)
- Earliest Deadline First (EDF)
- Value-Based Scheduling (VBS)

11.0.1 Fixed-Priority Scheduling

Most widely used and main focus in this course. Each task has a static priority computed pre-run-time and the runnable tasks are executed in the order determined by their priority.

With priority-based scheduling, a high-priority task may be released during the executing of a lower priority one. In a **preemptive** scheme, there will be an immediate switch to the higher-priority task. With **non-preemption**, the lower priority task will be allowed to complete before the other executes.Preemptive allows for more reactive high-priority tasks. **Cooperative dispatching (deferred preemption)** is a hybrid of the two. EDF and VBS can also either take a preemptive or non-preemptive approach.

11.1 Scheduling characteristics

- **Sufficient scheduling** - passing the test results in a program that meet deadlines.
- **Necessary scheduling** - failing the scheduling test guarantees that deadlines will be missed.
- **Exact scheduling** - Necessary and sufficient
- **Sustainable scheduling** - system stays schedulable if conditions improve.

11.2 Simple task model

Simple Task Model makes the following assumptions:

- The application is assumed to consist of a fixed set of tasks
- All tasks are periodic, with known periods.
- The tasks are completely independent

- All system's overheads, context-switching times and so on are ignored (i.e. assumed to have zero cost)
- All tasks have a deadline equal to their period, that is each task must complete before it is next released
- All tasks have a fixed worst-case execution time.

11.3 Standard Notation

- **C** - Worst-case computation time of task
- **D** - Deadline of the task
- **I** - The interference time 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, jobs (task period). In simple task model `T=D`
- **U** utilization of each task, equal to `C/T`
- **a-z** The name of the task.

11.4 Rate Monotonic Priority Assignment

Each task is assigned a unique priority based on its period; the shorter the period the higher the priority. I.e. for two tasks i and j

$$T_i < T_j \implies P_i > P_j \tag{1}$$

This assignment is optimal in the sense that if any task can be scheduled, using preemptive scheduling, with a fixed-priority assignment scheme, then the given task set can also be scheduled with a rate monotonic assignment scheme. Note that priority 1 is lowest (least) priority.

11.5 Utilization-Based Analysis

This method only works for tasks following the simple task model as `D=T` must be affirmed. It is a simple *sufficient* but **not** *necessary* schedulability test

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

$$U \leq 0.69 \text{ as } N \to \infty \tag{3}$$

11.6 Response-Time Analysis

Here task i's worst-case response time **R** is calculated first and then checked trivially with it's deadline

$$R_i \leq D_i \tag{4}$$

$$R_i = C_i + I_i \tag{5}$$

where I_i is the interference on task i from higher priority tasks.

11.6.1 Calculating R

During R, each higher priority task j will execute a number of times:

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil \tag{6}$$

Total interference from task j is then given by:

$$\left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{7}$$

From this we get the **Response Time Equation**

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{8}$$

where $hp(i)$ is the set of tasks with higher priority than task i. Solve the Response Time Equation with the recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{9}$$

The set of values w_i^0 , —,w_i^n, ... is monotonically non-decreasing. When

$$w_i^n = w_i^{n+1} \tag{10}$$

the solution to the equation has been found. w_i^0 must not be greater than R_i , it can e.g. be 0 or C_i .

11.7 Example

Task	Period	ComputationTime	Priority
	T	C	P
a	7	3	3
b	12	3	2
c	20	5	1

$$R_a = 3$$

$$w_b^0 = 3 \qquad (11)$$
$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6 \qquad (12)$$
$$w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6 \qquad (13)$$
$$R_b = 6 \qquad (14)$$

$$w_c^0 = 5 w_c^1 = 5 + \left\lceil \frac{5}{12} \right\rceil 3 + \left\lceil \frac{5}{7} \right\rceil 3 = 11 \qquad (15)$$
$$w_c^2 = 5 + \left\lceil \frac{11}{12} \right\rceil 3 + \left\lceil \frac{11}{7} \right\rceil 3 = 14 \qquad (16)$$
$$w_c^3 = 5 + \left\lceil \frac{14}{12} \right\rceil 3 + \left\lceil \frac{14}{7} \right\rceil 3 = 17 \qquad (17)$$
$$w_c^4 = 5 + \left\lceil \frac{17}{12} \right\rceil 3 + \left\lceil \frac{17}{7} \right\rceil 3 = 20 \qquad (18)$$
$$w_c^5 = 5 + \left\lceil \frac{20}{12} \right\rceil 3 + \left\lceil \frac{20}{7} \right\rceil 3 = 20 \qquad (19)$$
$$R_c = 20 \qquad (20)$$

We have that for both task a, taks b and task c, $R_i \leq D_i = T_i$

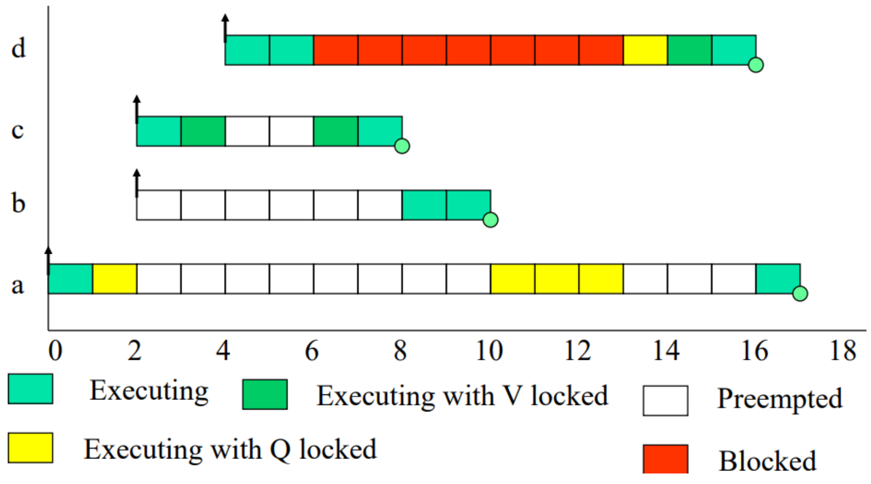
11.7.1 Summary

Response time analysis is **sufficient and necessary (exact)**. If the task set passes the test they will meet all their deadlines and if they fail the test then at run-time a task will miss its deadline under the assumption that computation time estimations themselves are not pessimistic)

11.8 Task Interactions and Blocking

If a task is suspended waiting for a lower-priority task to complete some required computation then the priority model is in some sense being *undermined*. It is said to suffer **priority inversion** and the high-priority task waiting for low-priority is said to be **blocked**. To show this, consider an example where four periodic tasks a,b,c and d and two resources Q and V are at play:

Task	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



What we see here is that if task p is blocking task q, then p runs with q 's priority. Above d runs with a's priority.

11.9 Priority Ceiling Protocols

A priority ceiling protocol avoids **unbounded priority inversion** and deadlock. Each shared resource is assigned a ceiling priority equal to the highest priority of any task that can lock it. There are two forms

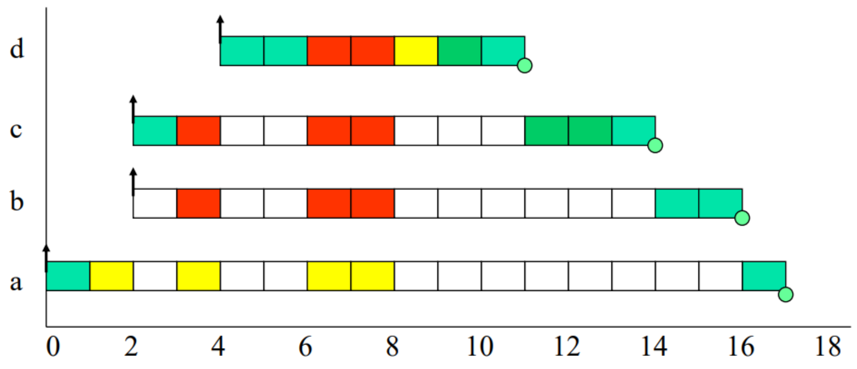
- Original Ceiling Priority Protocol (OCPP)
- Immediate ceiling priority protocol (ICPP)

If we are on a single processor, the protocol ensures the following

- 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 by the protocol itself.

11.9.1 OCPP

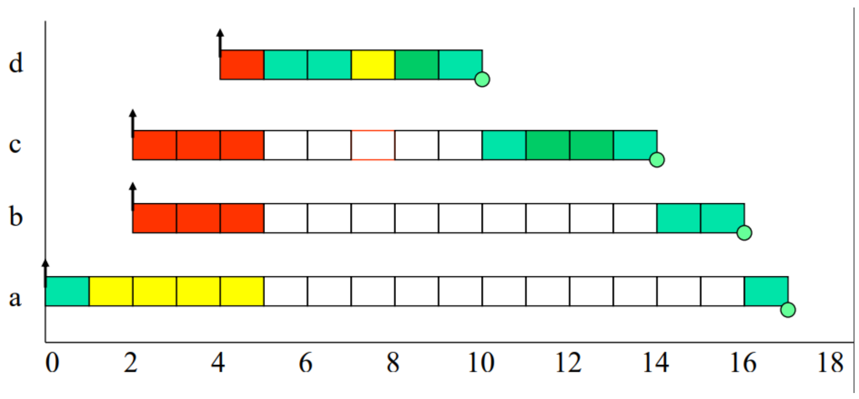
- Each task has a static default priority assigned.
- Each resource has a static ceiling value defined, this is the maximum priority of the tasks that use it.
- A task has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority tasks,
- A task can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource excluding any that it has already locked itself.



11.9.2 ICPP

- Each task static default priority
- Resource has static ceiling value defined, this is maximum priority of the tasks that use it.
- A task has a dynamic priority that is the maximum of static and ceiling value of any resource it has locked.
- Consequence of ICPP; task will only suffer block at the very beginning of its execution.

- Consequence of ICPP; once task starts executing, all resources it needs must be free; if they were not, then some task would have an equal or higher priority and the task's execution would be postponed.



11.9.3 OCPP versus ICPP

Worst-case behavior is identical but are some differences:

- ICPP easier to implement as blocking relationships need not be monitored.
- ICPP leads to less context switching
- ICPP requires more priority movements as this happens with all resource useage.
- OCPP changes priority only if an actual block has occurred.