Summary of
TTK4145 - Real Time Programming

Asgeir Hunshamar

2019

Contents

1 Introduction **1**

2 Course Overview **2**
2.1 Course Learning Goals . . . 2
2.2 Course Introduction . . . 2

3 Code Quality **3**
3.1 Learning Goals . . . 3
3.2 Introduction . . . 3
3.3 Modules . . . 3
3.3.1 Code Complete checklists for modules . . . 3
3.3.2 Key points on modules . . . 4
3.4 Routines . . . 4
3.4.1 Code Complete checklist for routines . . . 5
3.4.2 Key points on routines . . . 6
3.5 Variable naming . . . 6
3.5.1 Code complete checklist for naming variables . . . 7
3.5.2 Key points on variable names . . . 8
3.6 Self-Documenting Code . . . 9
3.6.1 Code complete checklist for commenting code . . . 9
3.6.2 Key Points on Self Documenting Code . . . 11
3.7 Examples . . . 11
3.7.1 Continuation exam 2014 . . . 11
3.7.2 Continuation exam 2014, another example . . . 12
3.7.3 Bad interface example . . . 13
3.7.4 Bad naming example . . . 13

4 Fault Tolerance **15**
4.1 Learning Goals . . . 15
4.2 Introduction . . . 15
4.2.1 The normal way of handling bugs . . . 15
4.2.2 Cause of errors . . . 17
4.3 Short definitions of terms from learning goals . . . 17
4.4 Failure modes . . . 18
4.4.1 Merging of failure modes . . . 19
4.5 Acceptance tests . . . 19
4.6 N-version programming . . . 20
4.7 Dynamic redundancy . . . 20
4.7.1 Error Detection . . . 20
4.7.2 Damage confinement and assessment . . . 21
4.7.3 Error recovery . . . 21
4.7.4 Fault treatment . . . 23

4.8 Recovery blocks 23

5 Fault Model and Software Fault Masking **24**
5.1 Learning goals 24
5.2 Introduction 24
5.3 The underlying progression 24
5.4 Work method 24
5.5 The three cases in low level design for fault tolerance by redundancy 25
5.5.1 Case 1: Storage 25
5.5.2 Case 2: Messages/communication 25
5.5.3 Case 3: Processes/calculations 26

6 Transaction Fundamentals **28**
6.1 Learning Goals 28
6.2 Introduction 28

7 Atomic Actions **29**
7.1 Learning Goals 29
7.2 Introduction 29
7.3 From exams 29

8 Shared Variable Synchronization **30**
8.1 Learning Goals 30
8.2 Introduction 30

9 Scheduling **31**
9.1 Learning Goals 31
9.2 Introduction 31

10 Terms **32**

1 Introduction

This summary is mainly a collection of lecture notes, as well as other materials relevant for the exam in TTK4145 - Real-Time Programming with a focus on the learning goals of the course and relevance to the exam. The chapters in the summary is ordered in roughly the same order as they are introduced in the lectures.

Most of the summary is copy-paste from Wikipendium - TTK4145: Real Time Programming., exam solutions, lecture notes and the recommended reading books for the subject.

2 Course Overview

2.1 Course Learning Goals

- General maturation in software engineering/computer programming.
- Ability to use (correctly) and evaluate mechanisms for shared variable synchronization
- Understand how a deterministic scheduler lays the foundation for making real-time systems
- Insight into principles patterns and techniques for error handling and consistency in multi thread / distributed systems.
- Knowledge of the theoretical foundation of cuncurrency, and ability to see how this can influence design and implementation of real-time systems

2.2 Course Introduction

3 Code Quality

3.1 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.

3.2 Introduction

The ultimate software quality metric (according to Sverre) is **Maintainability**. Maintainable code is code that is easy to modify, fix bugs or extend. Sooner or later a system gets too large to maintain, but this can be delayed by increasing the maintainability of the system. The maintence cost increases more than linearly on project size. How do we make a system maintainable?

3.3 Modules

A module is a group of lines, a function, a file, an object, group of files, package, library etc.

Criteria for modules:

- You must be able to use a module without knowing its internals
- You must be able to maintain the module without knowing the rest of the system
- Composition: You should be able to build superior modules from submodules, providing scalability
- Cohesion: The parts of a module should be well connected
- Interfaces should be complete and minimal

3.3.1 Code Complete checklists for modules

Encapsulation:

- ☐ Does the module minimize accessibility to its data members?
- ☐ Does the module avoid exposing member data?

- ☐ Does the module hide its implementation details from other modues as much as the programming language permits?
- ☐ Does the module avoid making assumptions about its users, (including its derived classes?)
- ☐ is the module independent of other modules? Is it loosely coupled?

Other implementation issues:

- ☐ Does the module contain about seven members or fewer?
- ☐ Does the module minimize direct and indirect routine calls to other modules?
- ☐ Does the module collaborate with other model only to the extent absolutely necessary?
- ☐ Is all member data initialized appropriately?

Language-specific Issues

- ☐ Have you integrated the language-specific issues for modules in your specific programming language?

3.3.2 Key points on modules

Class interface should provide a consistent abstraction. An abstraction is the ability to view a complex operation in a simplified form. A module interface provides an abstraction of the implementation that is hidden behind the interface. A module interface should hide something.

Interfaces should be complete and minimal. In general helper functions, like get-functions, should not be put into the interface. This increases the complexity too much. The seven functions in the module interface should ideally be self explanatory.

3.4 Routines

In general a routine is a function (which returns a value) or a procedure (void function) in a program, and it is also a module. A routine encapsulates/hides something in the program, introducing abstraction and reducing complexity.

General criteria for routines

- Good, describing name
- Does one thing, and does it well.

- Readable; It is easy to see what it does
- You can say it works correctly only by reading it
- It is defensive; protects itself from bad usage and bad parameters (assumptions are programmatic)
- Uses all parameters and variables (but not reuse for other purpose)
- High cohesion; all its parts have to do with the same.

3.4.1 Code Complete checklist for 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 a strong, functional cohesion-doing one and only one thing and doing it well?
- □ Do the routines have loose coupling, i.e. 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 an artificial coding standard?

Parameter-Passing Issues:

- □ Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- □ Are 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?

3.4.2 Key points on routines

The **routine size** should be large enough to not represent a maintenance problem in itself, and small enough to still represent useful abstraction, encapsulate something worth encapsulating

The **routine name** should describe all a routine does. It should be specific (avoid "handle", "perform", "do", "process" etc.). In the case of a function, the name should describe the return value.

Parameter order and naming conventions should be standardized.

The most important reason for creating a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved readability, reliability, and modifiability are better reasons.

Sometimes the operation that most benefits from being put into a routine of its own is a simple one

You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.

The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.

Functions should be used only when the primary purpose of the function is to return the specific value described by the function's name.

Careful programmers use macro routines with care and only as a last

3.5 Variable naming

General criteria for variable naming

- The name of the variable should fully and accurately describe what the variable represents
- The name should refer to the real-life problem, rather than to the programming language solution
- The name of the variable should be long enought that you don't have to puzzle it out

- Computed value qualifiers should, if any, be at the end of the name.
- The name should use "Count", or "Index" instead of "Num"

3.5.1 Code complete checklist for naming variables

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 program- ming-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" 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? Key Points 289
- ☐ 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?

3.5.2 Key points on variable names

Good variable names are a key element of program readability. Specific kinds of variables such as loop indexes and status variables require specific considerations.

Names should be as specific as possible. Names that are vague enough or gen- eral enough to be used for more than one purpose are usually bad names.

Naming conventions distinguish among local, class, and global data. They dis- tinguish among type names, named constants, enumerated types, and variables.

Regardless of the kind of project you're working on, you should adopt a variable naming convention. The kind of convention you adopt depends on the size of your program and the number of people working on it.

Abbreviations are rarely needed with modern programming languages. If you do use abbreviations, keep track of abbreviations in a project dictionary or use the standardized prefixes approach.

Code is read far more times than it is written. Be sure that the names

3.6 Self-Documenting Code

General criteria:

- If something is not clear from the code. Improve code by choosing better names, divide complex statements, move complex code into well-named functions; can you use layout or whitespace?
- Avoid useless comments
- Comments are to be used to emphasize structure, as headlines, give summaries or describe intent.
- comments should be easy to maintain

3.6.1 Code complete checklist for commenting code

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 been 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 number in the listing?

3.6.2 Key Points on Self Documenting Code

The question of whether to comment is a legitimate one. Done poorly, commenting is a waste of time and sometimes harmful. Done well, commenting is worthwhile.

The source code should contain most of the critical information about the program. As long as the program is running, the source code is more likely than any other resource to be kept current, and it's useful to have important information bundled with the code.

Good code is its own best documentation. If the code is bad enough to require extensive comments, try first to improve the code so that it doesn't need extensive comments.

Comments should say things about the code that the code can't say about itself - at the summary level or the intent level.

Some commenting styles require a lot of tedious clerical work. Develop a style that's easy to maintain.

3.7 Examples

3.7.1 Continuation exam 2014

```
void printNameAndAddress(int i){
sem_wait(personSem);
printName(i);
printAddress(i);
sem_signal(personSem);
}
```

```
#ifndef PERSON_H
#define PERSON_H

typedef struct {
char * firstName;
char * lastName;
char * street;
```

```
int streetNumber;
} TPerson;
void reallocateArray(int newSize);
TPerson ** getArray();
void printName(int personNumber);
void printAddress(int personNumber);
void printNameAndAddress(int personNumber);
...
#endif
```

The interface is not minimal any more (which is a bad thing). Continuing this trend will lead to code duplication in the module. (More obscurely: We get dependencies between functions in the module which increases module complexity). But of course; we sometimes do make convenience functions, if the convenience is great enough :-)

- *The type should not be a part of the module interface - breaks the encapsulation!*
- *returning the list in getArray() is *really* terrible! - breaks the encapsulation!*
- *reallocateArray() is inconsistent abstraction and breaks encapsulation revealing more of the implementation than should be necessary.*

3.7.2 Continuation exam 2014, another example

```
#ifndef lift_cost_h
#define lift_cost_h

int calculateCost(int currentFloor, int direction,
                  int orderedFloor, int orderedDirection);
int downCost[MAX_ELEVATORS][N_FLOORS];
int upCost[MAX_ELEVATORS][N_FLOORS];
void fillCostArrays();
void clearCosts(void);
int lowestCostFloor(int elevator);
int lowestCostDirection(int elevator);
int findBestElevator(int floor, int direction);
void designateElevators();
void clearDesignatedElevator();
int designatedElevator[N_FLOORS][2];

#endif
```

Any reasonable comment the student does should be rewarded. However, the "ideal solution" argues along the lines of the Code Complete checklists. The main issue with this module is that this is not a module interface! It is (probably) a list of all functions in the module, including the variables!

- *Ideally it should be clear from the interface exactly what the responsibility of the module is, and how it should be used correctly. It is not.*
- *There are (probably) nonobvious dependencies between the functions in the interface. (When do you need to call clearCosts for example?)*
- *The interface is not minimal. The functionalities of calculateCost, findBestElevator, lowestCostFloor and lowestCostDirection is overlapping. Also probably clearCosts and fillCostArrays.*
- *The abstraction is not consistent. The name "Cost" indicates that the module calculates or manages costs in some way. Either calculateCost or findBestElevator must be the main purpose of the module? But then there is the manipulation of some "costArrays" in addition - and keeping track of a "designatedElevator"?*
- *The data members are not encapsulated.*
- *(Having non-external variables in the headerfile like this is a bug, in addition to the fact that exporting the module's data is bad form) Possibly some of these thinks could have been mitigated by commenting, but this would still be a badly designed module interface*

3.7.3 Bad interface example

```
#ifndef COMMANDSTACK_H
#define COMMANDSTACK_H

void InitializeCommandStack();
void PushCommand( Command command );
Command PopCommand();
void ShutdownCommandStack();
void InitializeReportFormatting();
void FormatReport( Report report );
void PrintReport( Report report );
void InitializeGlobalData();
void ShutdownGlobalData();

#endif
```

This is an example of a bad interface. This module is not only a command stack module, but includes code for report formating, report printing, shut down global data. etc. This module is not consistent

3.7.4 Bad naming example

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
```

```
void AddEmployee( Employee employee );
void RemoveEmployee( Employee employee );
Employee NextItemInList();
Employee FirstItem();
Employee LastItem();

#endif
```

Here we can see that the three last functions are poorly named. Could have been done better.

4 Fault Tolerance

4.1 Learning Goals

- Understand and use terms (like): Reliability. Failure vs. fault vs error. Failure modes. Acceptance test. Fault prevention vs tolerance. Redundancy, Static vs Dynamic. Forward / Backward error recovery.
- Understand, use and evaluate techniques (like): N-version programming. Recovery blocks. Error detection. Failure mode merging. Acceptance tests.

4.2 Introduction

Fault Tolerance encompasses more than just minimizing the number of bugs in the system. The system should behave as specified even though there are bugs there

The ambition should always be to make systems without bugs. Keep ambitions high. But in practice all systems have bugs.

Fault tolerance enables the system to continue functioning even in the presence of faults. Different levels of fault tolerance can be provided by the 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.

All techniques for achieving fault tolerance rely on extra elements (redundancy) introduced into the system to detect and recover from faults.

4.2.1 The normal way of handling bugs

1. Make your program to fill the functional spec.
2. Run/Test
3. Errors happen

4. Find "cause" in code
5. Add/change code

for example:

```
FILE *
openConfigFile(){
    FILE * f = fopen("/home/sverre/.config.cfg","r");
    if(f == NULL){
        switch(errno){
            case ENOMEM: {
                ...
                break;
            }
            case ENOTDIR:
            case EEXIST: {
                // ERROR!
                break;
            }
            case EACCESS:
            case EISDIR: {
                ...
                break;
            }
            ....
        }
    }
    return f;
}
```

This is traditional error handling. A ton of cases to handle situations that appear when the file is not found. We can not be certain that we will handle all cases.

Testing is not good enough because:

- Can only show existence of errors. Can not show that there are not any errors.
- Cannot find errors in specification.
- Is realistic testing possible?
- Cannot find race conditions.
- Embedded systems have higher demands on reliability / safety / availability etc.

4.2.2 Cause of errors

"Unhandled situations" are a much more common cause of error than "software bugs". A software specification seldom handles all the unexpected situations that should be handled, and never (?) distinguishes between situations that should be handled and those that allows the program to fail

This is not good enough! Our embedded system must be able to handle all errors, also the unexpected ones. Since more threads are cooperating in the system, then sometimes they must also corporate on error handling.

....

4.3 Short definitions of terms from learning goals

This is a list with very short definitions of the fault tolerance terms from the learning goals. Many of the terms are more fleshed out in separate sections.

Reliability

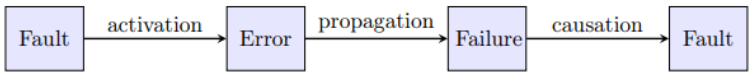
A measure of the success with which the system conforms to some authoritative specification of its behaviour.

Failure vs fault vs error

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.



Failure modes

A cause of failure or possible ways a system can fail. See 4.4

Acceptance test

A test conducted to determine if the requirements of a specification or contract are met. See 4.5

Fault prevention vs tolerance

Fault prevention is an approach that aims to reduce the failures in a system by reducing the number of faults

Fault tolerance accepts faults in a limited capacity.

Redundancy, Static vs Dynamic

In software engineering extra elements are often introduced to the system in order to detect and handle faults, This method is called redundancy.

Static redundancy runs continuously no matter if the system fails or not.

Dynamic redundancy have modules in the background and only runs when an error is detected. This could decrease resource usage while still handling errors properle. See 4.7

Forward/backward error recovery

When the error is found and we know the extent of it, the recovery process can start. There are two methods of error recovery: Forward and backward recovery.

Forward recovery involves continuing as before with selected corrections.

Backward recovery steps back to a previous state, called a recovery point, and tries to accomplish the same action with a different method.

For more details see 4.7.3

N-version programming

Static redundancy, relies on several pieces of software to compute the same result. If the programs does 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. For more details see 4.6

Recovery blocks

Dynamic redundancy. An implementation of dynamic recovery applied to software. It defines all entries to a software block as a recovery point, and then have an acceptance test before the block can exit. See 4.8

Error detection

Ways to detect faults in the software. See 4.7.1

4.4 Failure modes

- In which manners can the system fail?
- Which situations do we need to handle?

fopen() in C has the following failure modes:

```
On error NULL is returned, and the global variable errno is set:
EINVAL The mode provided to fopen was invalid. The fopen
function may also fail and set errno for any of the
errors specified for the routine malloc(3). (ENOMEM)
The fopen function may also fail and set errno for any of the
errors specified for the routine open(2). (EEXIST,
EISDIR, EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, ENXIO,
ENODEV, EROFS, ETXTBSY, EFAULT, ELOOP, ENOSPC,
ENOMEM, EMFILE, ENFILE)
```

The subset of failure modes that we need to handle are found by understanding the internals of fopen().

In general there are two domainso of failure modes that can be idenfitifed

- **value failure** - the value associated with the service is in error
- **time failure** - the service is delivered at the wrong time

4.4.1 Merging of failure modes

It is impossible to determine every single way a system might fail, and develop error handling procedures for each different failure mode. Instead, consider what the worst case error that needs to be handled is, and then treat every other error as this type of error. This technique is refered to as merging of error modes, and simplifies error handling, but might result in some errors having more drastic consequences than strictly necessary.

Gains:

- Simplification of the system. (If handling the worstcase error anyway, maybe all other errors can be handled the same way)
- Error modes is part of module interface: Fewer error modes enhances modularity / maintenance / composition by reducing size of interface.
- Handling unexpected errors, since merging of failure modes also can encompass unknown error modes...

4.5 Acceptance tests

Fault tolerance does not require that the system is fault free. We use testing for detecting whether /anything/ is wrong, not to identify any fault utilize several of the methods discussed for error detection in section 4.7.1. 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.

4.6 N-version programming

As mentioned in 4.3, N-version programming is based on static redundancy.

N-version programming relies on several pieces of software to compute the same result. There are several degrees of diversity, ranging from using different compilers for the same program, to let several teams build the same functionality in different programming languages. During execution, a driver process handles starts the programs, receive the result and act depending on the output. If the programs does 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. One of the downsides to this solution is that, in a real-time system, the driver process and the different programs might need to communicate between them, introducing a communication module as well. Add this to the fact that software is by far the most extensive and time consuming part of a real-time system, and the project costs might double by introducing a redundant language. There are also increased strains on hardware, or even need for separate hardware pieces, which increases the cost and/or complexity of the system. As a final note, recall that most software faults originate from the specification, thus all N versions might suffer from the same fault.

4.7 Dynamic redundancy

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

4.7.1 Error Detection

How to detect faults? The "learn by heart" list

Replication checks

Uses N-version programming to tolerate software faults. If you can do something in two different ways and they arrive at the same results, the propobility that it is correct increases enormeosly.

Timing Checks

You check the expected time of the process. For example by using a watchdog timer which during normal operation, the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout

signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

Reversal Checks
A lot of problem are easier to check if the answer is correct, than to find the correct answer, for example for a component which finds the square root of a number, the reversal check is simply to square the output and compare it with the input.

Coding Checks
Used to check for corruption of data. For example checksum.

Reasonableness Checks
these are based on knowledge of the internal design and construction of the system. They check that the state of data or value of an object is reasonable, based on its intended use.

Structural Checks
structural checks are used to check the integrity of data objects such as lists or queues. They might consist of counts of the number ‘ of elements in the object, redundant pointers or extra status information.

Dynamic Reasonableness Checks
with output emitted from some digital controllers, there is usually a relationship between any two consecutive outputs. Hence an error can be assumed if a new output is too different from the previous value

4.7.2 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 see 7

4.7.3 Error recovery

When the error is found and we know the extent of it, the recovery process can start. There are two methods of error recovery: Forward and backward recovery.

- **Backward error recovery:** Stores recovery points after successful acceptance tests containing the full (and consistent) state of the system. If an acceptance test fails at a later point, the system is restored to the previous safe state (from the recovery point) and executes an alternative section of the program.

- **Forward error recovery:** When an acceptance test fails, the system attempts to continue from an erroneous state by making selective corrections to the system state to arrive at a consistent state. Simultaneously the system makes safe any aspect of the controlled environment which may be hazardous or damaged because of the failure.

Backwards error recovery works straightforward when there are no threads, but it quickly gets complicated with multiple threads, because of communication between threads: If we need to roll one thread back to an earlier recovery point, we might have to undo a message sent to the other thread. If this is the case, the other thread might have to roll back, requiring messages sent to other threads to be undone. Thus since one thread has had an error, multiple other threads will have to roll back as well. This is called the domino effect. See figure 1

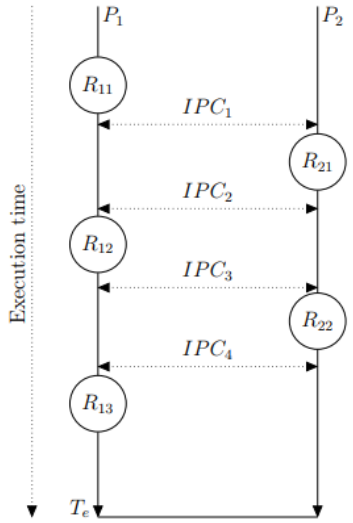


Figure 1: Domino effect

In more detail, the domino effect happens when a threads need to revert to a recovery point after detecting an error, but having exchanged possibly inconsistent data during inter-process communication with another thread after the recovery point. Consider figure 1, if an error occurs at time T_e in thread P_2, the thread needs to revert to recovery point R_{22}. However, the IPC_4 also needs to be undone since the data might be inconsistent, so P_1 needs to revert to R_{12} as well. This leads to IPC_3 needing to be reverted, and so on. As is the case with the figure, the worst case of the domino effect is that both threads need to roll back their state all the way to the start of the computation, which might be unfeasible for a real-time system.

4.7.4 Fault treatment

While error corrections remedy the error, it does not remove the underlying fault. This fault might be easy to identify and/or remedy, and the experienced error will not happen again. Both hardware and software faults are relatively easy to fix once identified if the system is allowed to be brought oine. If the program have to be modified while executing, things get a little more complicated.

4.8 Recovery blocks

Method of dynamic redundancy. Recovery blocks are an implementation of dynamic recovery applied to software. It defines all entries to a software block as a recovery point, and then have an acceptance test before the block can exit.

5 Fault Model and Software Fault Masking

5.1 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/masking with redundancy ...aiming for progression of fail fast, reliable and available systems
- Ability to Implement (simple) Process Pairs-like systems

5.2 Introduction

Fault model: Failure modes, probabilites and spec

Fault masking: Masking errors by redundancy

Designing fault-tolerant program require a model. The model must define correct behavior, and if the programs are to deal with faults, the model must describe the kind of faults and their relative frequencies. Given such a model programs can be written, system reliability can be estimated using probabilistic methods, and proofs that the programs are correct can be made with any desired degree of formality. The model involves three entity types: *processes, messages and storage*. Each has a set of desired behaviors and a set of failure behaviors. This section shows how to transform each of the three entities from failfast entities into highly reliable and even highly available entities.

5.3 The underlying progression

1. fail-fast: They either execute the next step, or they fail and reset to the null state
2. reliable: Failfast + repairability
3. available: Continuous operation

5.4 Work method

1. Find the failure modes
2. Detect erros and merge failure modes (+error injection for testing)

3. Handling/masking with redudancy ...aiming for progression of fail-fast, reliable and available systems.

5.5 The three cases in low level design for fault tolerance by redundancy

5.5.1 Case 1: Storage

- **Failure modes:**

Write	Read
Writes wrong data	Gives wrong data
Writes wrong place	Gives old data
Does not write	Gives data from wrong adress
Fails	Fails

- **Detection:**

  How to know if any error modes have happened?

  - Also write address, so you can check that it is correct.
  - All errors → fail
  - Make the "decay" thread thread that runs in paralell and flips status bits (For testing).

- **Handling with redundancy:**

  - Static redundancy: Probability failure rates specifies the required redundancy to ensure availability according to specification. Use n copies of storage modules (hard drives). Read and write to all redundant modules, the probability for n bad reads so small that it can be ignored. To ensure consistent data, compare version id and use data with newest id
  - Dynamic redundancy: All reads leads to write-back on error, assuming that at least one storage module returned valid data. Repair thread reads all pages regularly and performs writebacks as required. Data used seldom might deteriorate due to age, repair thread refreshes the pages

5.5.2 Case 2: Messages/communication

- **Failure modes:**

  - Lost

  - delayed
  - corrupted
  - duplicated
  - wrong recipient
- **Detection:**
  - Session ID
  - Checksum
  - Acknowledgement (when recieving a message, we send a reply so the sender knows the message is received)
  - sequence numbers ()
  - All errors → Lost message
- **Handling with redundancy:**

  Timeout and retransmission:

  Dynamic redundancy implemented using a timer and resend mechanisms. After sending message, start timer and wait for acknowledgment back from receiver, and then resend the message on absent acknowledgment. Threat all errors as message lost and resend.

5.5.3 Case 3: Processes/calculations

- **Failure modes:**

  Does not do the next correct side effect.

- **Detection:**

  All failed acceptance tests → PANIC/STOP (Failfast)

- **Handling with redundancy:**

  Three methods

  - Checkpoint - restart: Similar to backwards recovery using recovery points. Calculation is executed in steps
    1. Calculate result and side effect
    2. Acceptance test, if it fails, restart from previous stored checkpoint of last execution

3. Store full state as recovery point in reliable storage, and write the complete state to reliable storage module
4. Do side effect, in case of restart, previous side effect might be executed twice

– Process Pairs: Two processes, Primary Backup. Primary performs the work and sends periodic "I'm alive" messages to Backup, and sends full state to Backup after state change. Backup is the checkpoint, takes over when the Primary fails. Failfast implementation, Primary instant crash and restart on failure. Masks hardware failure (processor failures) as well as transient software failures (Heisenbugs). Redundancy by resending masks communication errors.

– Persistent processes: Assumes transactional infrastructure with databases to preserve data and operating system that implement transactions. All calculations are transactions, atomic transformations from one consistent state to another, either done completely or not at all. The processes are stateless with all data safely stored in highly available databases. First read data from database, then before side effects, write updated data back to database

6 Transaction Fundamentals

6.1 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 highlevel design.
- Comprehension of terms: Optimistic Concurrency Control, Two-phase commit optimizations, Heuristic Transactions, Interposition.

6.2 Introduction

Transactions are a way to synchronize error handling between multiple participants involved in an action. A transaction is an action that

- Keeps track of participants in the action
- Defines a consistent "starting point" for the action
- Enforces borders for the action
- Ensures that the action is consistent when completed

and has the following properties

- Atomicity: The transaction completes successfully (commits) or if it fails (abort) all of its effects are undone (rolled back)
- Consistency: Transactions produce consistent results and preserve application-specific invariants
- 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 (except by a catastrophic failure)

Transactions prevent the domino effect by synchronizing the recovery points of all participants at the start of the action, and at the end of a successfully committed action. This ensures that a backward error recovery only needs to roll back to the start of the transaction to be consistent.

7 Atomic Actions

7.1 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.

7.2 Introduction

7.3 From exams

5-4) What is an Atomic Action? Which problem(s) is Atomic Actions meant to solve? Expectations: More other good answers exist for the first part here. Pointing out the three boundaries (side, start and end), possibly with standard mechanisms to achieve them (locking, explicit membership, and two-phase commit protocol) is reasonable enough.

8 Shared Variable Synchronization

8.1 Learning Goals

- Ability to create (error free) multi thread programs with shared variable synchronization. 2
- 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 requeue and entry families) and Java.

8.2 Introduction

9 Scheduling

9.1 Learning Goals

- Be able to prove schedulability using the utilization test and the response time analysis for simple task sets.
- Know and evaluate the assumptions underlying these proofs and what is proven.
- Understand the bounded and unbounded priority inversion problems.
- Understand how the ceiling and inheritance protocols solves the unbounded priority inversion problem.
- Understand how the ceiling protocol avoids deadlocks.

9.2 Introduction

10 Terms

- A race condition : A bug that surfaces by unfortunate timing or order of events.

  Occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.