

Lecturenotes in ttk4145 Real-Time Programming spring 2015

Sverre Hendseth

April 27, 2015

Contents

1	Course intro, starting the project, info on exercises	7
1.1	Practical Details	8
1.2	Quality system	8
1.3	Reference Group	9
1.3.1	Its learning call for reference group members	10
1.4	Learning Goals	10
1.5	A special note on the programming languages.	11
1.5.1	The context of the project:	11
1.5.2	The context of the curriculum, lectures and the exam.	12
1.6	Lecture Plan, Real-Time Programming 2015	12
1.7	Project Q&A	14
2	Networking basics	14

3	Code Quality	14
3.1	Its Learning Call	14
3.2	Learning goals	15
3.3	Motivational Questions	16
3.4	Intro SW Quality	17
3.5	Modules	19
3.5.1	Modules, Intro	19
3.5.2	The checklists	20
3.5.3	Key Points on modules	21
3.5.4	Bad Example: CommandStack	22
3.5.5	Bad Example2: CommandStack	23
3.5.6	Other tips	24
3.6	HighQuality Routines	25
3.6.1	HighQuality Routines	25
3.6.2	Routine Size	26
3.6.3	Routine Name	27
3.6.4	Reasons to Create a Routine	28
3.6.5	Other tips	29
3.6.6	Checklists	30
3.6.7	Key Points	31
3.7	The power of variable names	32
3.8	Self-Documenting Code	35
3.9	Answer to motivational questions	39
4	Ada	41
4.1	Its learning call	41

5	Fault Tolerance Basics	41
5.1	Its Learning Call	41
5.2	Learning goals	42
5.3	Intro: Classic bug handling	42
5.3.1	Causes of errors	45
5.3.2	This is not good enough!	45
5.4	Fault Tolerance: Some terms	48
5.5	Fault tolerance: Some basic techniques	49
5.6	N-Version Programming	49
5.7	Recovery Blocks	51
5.8	SW dynamic redundancy	51
6	Gray: Fault Model & Software Fault Masking	53
6.1	Learning Goals	53
6.2	Its learning call	53
6.3	Intro	54
6.4	Case 1: Storage	55
6.5	Case 2: Messages	58
6.6	Case 3: Processes/Calculations	59
7	Fault tolerance with more participants: Transaction fundamentals	61
7.1	Its learning call	61
7.2	Learning Goals	62
7.3	Intro & Context	62
7.4	Locking	63
7.5	Bacwards error recovery	64
7.6	More things to do	65
7.7	Generalizing	66
7.7.1	Log II	70

7.7.2	Log III	71
7.7.3	Checkpoints	72
7.7.4	LogManager	73
7.7.5	LockManager	74
7.7.6	Summary 2	75
7.7.7	Cases	76
7.8	Further Comments on the transaction chapter	76
7.9	Repetition 8 patterns	77
7.9.1	Locking (w. growing/shrinking phases)	77
7.9.2	Tophase Commit (The vote)	77
7.9.3	Transaction Manager (Action Controller)	78
7.9.4	Resource Manager (A new way of making modules: The participant of a (trans)action)	78
7.9.5	Log	78
7.9.6	Checkpoints	78
7.9.7	Log Manager	78
7.9.8	Lock Manager	79
7.10	Sequence diagram for a transaction.	79
8	Atomic Actions and forward error recovery.	81
8.1	Its Learning Call	81
8.2	Learning Goals	82
8.3	Atomic Actions Intro	82
8.4	AA Standard implementation	83
8.5	How can we implement a AA?	84
8.5.1	Messagebased rm in go	85
8.5.2	Messagebased rm in Ada	86
8.5.3	Messagebased tm in Go	87
8.5.4	Sequence diagram for a transaction.	87

8.6	Sidestep: java monitors	89
8.7	Sidestep: ADA Protected Objects	89
8.8	Code Example: Making an AA framework in Ada.	90
8.9	Code Example: Making an AA framework in Java (without error handling)	90
8.10	Adding backwards error recovery to these code examples . . .	90
8.11	Asynch Notification Motivation	91
8.12	POSIX: Signals and threadCancel	92
8.13	C: setjmp/longjmp	93
8.14	ATC Implementasjon Java	94
8.15	Select then abort in ADA	94
9	Shared Variable Synchronization	96
9.1	Its learning call	96
9.2	Learning Goals.	99
9.3	Standard problems	100
9.4	Making a barrier	100
9.5	The cigarette-smokers problem exam task	106
9.6	Standard applications	106
9.7	Some harder problems	107
9.8	Pros and cons of preemptive and nonpreemptive scheduling .	109
9.9	Making a basic r-t kernel	110
9.10	Semaphores	112
	9.10.1 Presenting semaphores	112
9.11	Conditional Critical Regions	113
	9.11.1 Sidestep: Guards	113
9.12	Monitors	114
9.13	POSIX: Mutexes & Condition variables	115
9.14	Criticism of monitors	116

9.15	Java: Synchronized Objects	117
9.16	Java Example: Bounded buffer	118
9.17	Java discussion points:	118
9.18	Java Example: Read/Write locks in Java	120
9.19	Ada: Protected Objects	121
9.20	Ada Example: Update/Modify/Lock	124
9.21	Java Example: Update/Modify/Lock - Make it in Java	126
9.22	Java example: Request Parameters	127
9.23	Ada: Request parameters	127
9.24	Ada example with requeue I	128
9.25	Ada example with requeue II	130
9.26	Ada example with entry families	132
9.27	Java example: LIFO Scheduling	135
9.28	Summary	136
10	Deadlocks	136
10.1	Learning Goals	136
10.2	Deadlocks	136
11	Modeling of concurrent programs.	138
11.1	Its learning call	138
11.1.1	Last years call	138
11.2	Learning goals	139
11.3	Motivation	139
11.4	How to model a concurrent program	141
11.5	The modelchecking example:	144
12	Scheduling	148
12.1	Learning Goals	148
12.2	Its learning call	149

13 Guest lecture Teig and course ending.	149
13.1 Øyvinds slides.	149

1 Course intro, starting the project, info on exercises

Feedback and questions from everybody welcome at any time!

We have students with different backgrounds here: If I use a term that you do not understand, please make me aware of this!

I am always eager to improve the lectures. Give me feedback!

- Quality of blackboard notes?
- More code examples?

Be also aware Sverres known weaknesses:

- Side Tracks
- Half sentences
- Unanswered questions.

Interrupt me if I am trailing off.

I **am** willing to put in extra effort. Walkthrough of old exam questions, discussing project designs, "questions & answers" sessions before the exam etc. This happens on your initiative.

Anders' and Kyrres slides should be available at its learning. Introduction of project and exercises respectively.

Sverre talked about fault tolerance from the project perspective. Nothing that will not be repeated later in the course, but relevant as a preparation for the project.

"There will be bugs in your project! Your system should work correctly even though there are bugs there!"

1.1 Practical Details

Lecturer	Sverre Hendseth	sverre.hendseth@itk.ntnu.no
Scientific Assistant	Kyrre Gonsholt	kyrre.gonsholt@itk.ntnu.no
Unpaid super consultant	Anders Rønning Petersen	andepete@stud.ntnu.no
Student Assistant	Damir Anicic	anicic.damir91@gmail.com
Student Assistant	Steinar Kraugerud	steinaak@stud.ntnu.no
Student Assistant	Adelaide Marie Mellem	adelaidm@stud.ntnu.no
Student Assistant	Christoffer Ramstad-Evensen	chriram@stud.ntnu.no
Student Assistant	Jon-Håkon Bøe Røli	jon.hakon.boe.roli@gmail.com
Student Assistant	Bjørn Amstrup Spockeli	bjorn.spockeli@gmail.com

Project is due week 17

Exam: Saturday 22/5 0900-1300

Non-kyb students must go to Unni to get access to lab.

Evaluation:

- Project 25%
- Exercises: Mandatory (approved on lab)
- Exam 75%
- No Midterm

1.2 Quality system

- Check <https://innsida.ntnu.no/wiki/-/wiki/Norsk/Kvalitetssikring+av+utdanning>
- We are having a questionnaire to be answered by everybody.
- We are having a reference group (3+ meetings, pizza at the last one)

Quality System Action Points from 2014

- To motivate each lecture with relevant old exam questions
- Continually improve lecture notes.
- Try to set up a process for improvement of curriculum summary.
- Continually improve project execution (This year: make a better process on networking modules, change code quality evaluation?)

Note: Only students can enter items into quality system!

For reference: Action points from 2013 executed 2014:

- Lectures and exercises the first 4-5 weeks changed to support project startup better.
- Project milestones added.
- All (?) exercises reworked completely.
- Project slightly simplified.
- Curriculum somewhat reduced. Added curriculum on Code Quality, while reducing emphasis on a hard part: Asynch transfer of control.

1.3 Reference Group

Reference Group.

Erik Liland
Mikal Edlund
Patrik Levin
Gunnar Homb
Eirik Prestegårdshus Elektro

1.3.1 Its learning call for reference group members

I would very much like to have a representative for those who does not attend the lectures in the reference group, but you are a bit more difficult to recruit :-). Still you are a relevant part of the student group, and will be more dependent/critical on the quality of the other learning resources. Any volunteers?

Other groups that could well be represented:

- Weak or low ambition students (What can we do for you, to increase outcome or reduce workload?)
- Girls

There will be pizza

Sverre

1.4 Learning Goals

1

Learning Goals

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

Multicore important at the time: We **should** know how to do this! Will learn important basics here also.

Large parts of the industry does not know this - And need arise, as demands rise and multicore and wireless expands.

More detailed learning goals: Exist also for all parts of the course: Use them when preparing for the exam?

Question: What is the purpose of the course?

1.5 A special note on the programming languages.

1.5.1 The context of the project:

Each group selects freely the programming language for the project. Make the choice that makes the project easiest for you, or use the opportunity to learn a new language as you wish.

Three programming languages are "officially supported" by the student assistants on the lab:

- C (+ a reasonable subset of C++): This is a relevant programming language in the embedded setting, and most of you have some familiarity of it already.
- Python: Not so much an embedded language, but many of you know it, and the "fast and easy results" profile of python might be attractive. Who knows; maybe making network connections in python are simpler than in C? maybe there exist ready made transactional frameworks, distributed data managers, fault tolerance frameworks etc. that you can use?
- Go: The focus on Go at the lab stems from its support of the "communicating processes" way of designing parallel/concurrent software. If you design your system in terms of "servers" passing messages to each other, then Go is a nice choice.

If you choose another language we will still support you as best we can, but your ability to solve your own problems might be stressed slightly more. Google is your friend.

1.5.2 The context of the curriculum, lectures and the exam.

The standard design pattern in real-time systems is to distribute the responsibilities of the different timing demands to different threads. These threads then share resources (memory, hw, modules, libraries) and must **synchronize** their access to them.

Synchronization can be achieved by a large number of synchronization primitives, all with strengths and weaknesses, and it is a central goal of the course to learn to use and evaluate these.

In addition to primitives of historical interest (suspend and resume, events, . . .) and unimplemented suggestions (conditional critical regions, the generic monitor) we are studying the synchronization primitives of C/POSIX, Java and Ada since these are particularly interesting and well thought through.

- For C you should know of the mutex and its interaction with the condition variables. (+thread canceling and signals)
- For Java; synchronized object methods with the calls to wait(), notify() notifyAll() and interrupt(). For RT Java; that asynch transfer of control (ATC) is integrated into the exceptionhandling mechanism.
- For Ada; On one hand the protected object (with functions, procedures and entries with guards, requeue). ATC is here integrated with the select mechanism.

The languages is not important in themselves. You should be able to read example code in each of them though. On the exam, if asked to write pseudo code in language X, it is not important that the code looks like X code, but the relevant synchronization primitives should be used correctly.

1.6 Lecture Plan, Real-Time Programming 2015

This will change, during the semester.

1

Lecture Plan I

Week	Content
2	Project Startup! (course and exercise startup)
3	Networking Basics,
4	Code Quality (Curriculum from Code Complete)
5	Intro to Ada, Fault Tolerance Basics (B & W Ch. 2)
6	Grays 3 cases (Chapter from Gray), Transaction Fundamentals (Chapter)
7	Atomic Actions (B&W Ch. 7)
8	Atomic Actions/Asynch. transfer of control
9	Shared variable synch (B&W Little Book of Semaphores)

1

Lecture Plan II

Week	Content
10	Shared variable synch. (B&W Ch 5)
11	Shared variable synch. (B&W Ch 5) Deadlocks
12	Scheduling w. Amund. Excursion from Wednesday
13	Excursion Week
14	Easter Week
15	No lectures Monday and Tuesday.
16	Messagepassing
17	Guest Lecture, w. Teig.

There is one week of teaching missing here - week 18 that compensates for the lost excursion week. Hopefully we can use the spare exercise lectures during the semester to avoid lectures and exercise in this week.

1.7 Project Q&A

We talked a lot about the project... If anyone made notes I'll happily include them here.

2 Networking basics

Hello, all

This week we will continue making the foundation for the project. Monday morning (I hope to see you all; give this week a good start :-)) Mladen Skelin will start giving his great "introduction to networking" lectures, giving you the necessary background for making the project communication work.

The network communication has traditionally been a stumbling block for many groups in the project, and I sincerely recommend attending.

Tuesday Mladen will keep his third and last lecture on this topic, and Anders will continue with one hour of "language speed-dating", giving you the foundation for making a sensible choice of programming language for the project.

Sverre

See Mladens slides at its learning.

3 Code Quality

3.1 Its Learning Call

The theme for tomorrow will be code quality. Not only do we go through what is expected of you in the project (1/3 of project evaluation is on code quality), and how to answer code quality related exam questions. Code quality is also the criteria for evaluating tools, language mechanisms, techniques, designs etc. in all parts of later lectures: Why are global variables bad? What are the downsides to using semaphores? Which is the best programming language for shared variable synchronization? When is shared variables better than message passing? - It all boils down to what yields the higher code quality.

Note that I have uploaded some chapters of a fabulous book: Code Complete for supporting the lectures tomorrow. Chapter 6: Skip the things about

inheritance, (read only 6.2 "Good class interfaces" and do a mental search-and-replace for "class" with "module")

If you want to prepare for tomorrow; Each chapter has some checklists and "key points" summary; look through them.

Finishing the Code Quality section we will continue on basic fault tolerance.

The Real-time programming lectures monday morning is your opportunity to give your week a good start !! :-) See you!

Sverre

3.2 Learning goals

1

Learning goals; Code Quality

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

3.3 Motivational Questions

1

Exam question on Code Quality from Continuation 2014

As the program grows it turns out that the combination of `printName()` and `printAddress()` occurs many places in the program (when printing name tags, letterheads, business cards, envelopes,...), and we decide to make the function *printNameAndAddress* as a part of the module:

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

Look at the module interface; What do you think about this decision in a code quality perspective?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Exam question on Code Quality from Continuation 2014

(2x) A slightly larger part of our module is shown here:

```
#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
```

Criticize, from a code quality perspective, the inclusion of the type functions *reallocateArray()* and *getArray()*



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Exam question on Code Quality from Continuation 2014

In the listing under is a module interface (a C header file) from an old project deliverable. The modules name is "cost". Criticize (concisely, with bullet points) the design.

```
#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
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

3.4 Intro SW Quality

"Reflection on SW quality is the answer to all exam questions:

- What is good and what is bad? - What are strengths and weaknesses.
- Discuss goto, global variables, exceptions, semaphores, RT in general...

Brainstorms on:

- Why SW quality? (what are the goals?)
- What is SW quality?
- How do we achieve SW quality?

Sverres ultimate software quality metric: **Maintainability**.

- Sooner or later a system gets too large to maintain; delay this by increasing maintainability.

- Writing new SW is more fun than maintenance. (keep the maintenance ratio low) (Rule of thumb: Maintenance effort constitutes 20% of total development effort.)
- If we have maintainability then fixing bugs or adding features gets simple. (per definition)
- Law of nature: The quality of a program decreases over time.
- Maintenance cost increases more than linearly on project size.
- Measure maintainability expected effort to fix the next bug.

Also discuss: Code should do what is expected of it: The maintainers expectations must be made consistent with what the code in fact does. In an imperative language this is about goals and plans - steps to achieve the goals. "Causality".

How?

1. Modules!
2. Choose (Use or build) the right abstraction! The plan - the steps - should be concisely described. (Creative work, this! Choose the words to think with)
3. "Good design"?
4. "Intuition/Experience"?
5. Design Patterns -> Common Culture in project group.
6. Tools that generate code (or documentation)
7. Documentation. - in appropriate amounts

...so if something supports these, then it is good.

Example: Does our languages let us express our timing demands concisely?
Bad abstraction?

Ref. Chapter 8, Blooms criteria ("Expressive power and ease of use")

A module:

- A group of lines, a function, a file, an object, group of files, package, library, etc.
- You must be able to:
 - Maintain a module without knowing the rest of the system.
 - Use a module without knowing its internals.
- Composition: Building supermodules from submodules provides scalability!

3.5 Modules

3.5.1 Modules, Intro

- 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 supermodules from submodules.
- Coupling: Weak coupling between modules.
- Cohesion: The parts of a module should be well connected
- Interfaces should be complete and minimal.

3.5.2 The checklists

1

Module Interfaces Checklist: 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 modules 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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

1

Module Interfaces Checklist: Other Implementation Issues

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



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

1

Module Interfaces Checklist: Language-Specific Issues

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



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

3.5.3 Key Points on modules

1

Module Interfaces (classes, chapter 6.2) - Key Points

- Class interfaces 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)
- Modules are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.
- Sverre: Building/choosing abstractions is the secondmost important tool!



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

- What is hidden by a get-function? Get-functions hurts abstraction!
- Think about abstraction also internally in the "module".

3.5.4 Bad Example: CommandStack

1

A Bad Interface

```
#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
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseith S., name

3.5.5 Bad Example2: CommandStack

1

A Bad Interface - II

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

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

#endif
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

(This inherited list...)

3.5.6 Other tips

1

Other Tips

- Interfaces should be complete and minimal
- Services often comes in pairs: put/get, start/stop, init/shutdown, on/off...
- If your interface ends up not a great abstraction: Redesign!
- Make assumptions explicit in the code ("programmatic" rather than "semantic")
- Beware of erosion of the abstraction over time (ref pt 3)



NTNU
Norwegian University of
Science and Technology

3.6 HighQuality Routines

3.6.1 HighQuality Routines

1

High-Quality Routines (Chapter 7)

- Good, describing, name
- Does one thing and does it well
- Readable; It is easy to see what it does.
- You can say that 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)
- Less than seven parameters
- High cohesion



NTNU
Norwegian University of
Science and Technology

3.6.2 Routine Size

1

Routine Size

- How large can it be? (...to not represent a maintenance problem in itself...)
- How small can it be? (...and still represent useful abstraction, encapsulate something worth encapsulating, ...)



NTNU
Norwegian University of
Science and Technology

3.6.3 Routine Name

1

A good routine name

- describes all a routine does
- is specific (avoid "handle", "perform", "do", "process" etc.)
- A function: describe the return value
- A procedure: A verb and object
- Standardize naming and abbreviations
-



NTNU
Norwegian University of
Science and Technology

3.6.4 Reasons to Create a Routine

1

Reasons to Create a Routine

- **Encapsulate/hide something**
- **Introduce abstraction**
- Reduce complexity
- Avoid duplicate code
- Support subclassing
- Hiding sequences
- Hide pointer operations
- Improve portability
- simplify tests
- Improve performance
- Etc etc...



NTNU
Norwegian University of
Science and Technology

3.6.5 Other tips

1

Other Tips

- Standardize parameter orders.
- Standardize naming conventions.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

3.6.6 Checklists

1

Big-Picture Issues

- Is the reason for creating the routine sufficient?
- Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- Does the routine's name describe everything the routine does?
- Have you established naming conventions for common operations?
- Does the routine have strong, functional cohesion-doing one and only one thing and doing it well?
- Do the routines 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 an artificial coding standard?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

3.6.7 Key Points

1

Hig-Quality routines; Key Points

- 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 resort.



NTNU
Norwegian University of
Science and Technology

3.7 The power of variable names

1

A good variable name

- describing and specific
- (Can be shorter if scope is small)
- indicate type and/or scope by naming convention?
- standardize abbreviations
- Prefix enums with enum type?
- Differ between names for variables, routines(), TTypes, CONSTANTS, pPointers, etc...



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

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 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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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 pre-defined variable names?
- ...totally arbitrary names?
- ...hard-to-read characters?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

TPO Variable Names: Key Points

- 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 general enough to be used for more than one purpose are usually bad names.
- Naming conventions distinguish among local, class, and global data. They distinguish 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 NTNU approach.
- Code is read far more times than it is written. Be sure that the names you choose favor read-time convenience over write-time.

www.ntnu.no

Hendseth S., name

3.8 Self-Documenting Code

1

Self-Documenting Code

- Rule 1: If something is not clear from the code: Improve Code!
- Choose better names, divide complex statements, move complex code into well-named functions; can you use layout or whitespace?
- Rule 2: If the code is still not clear: The code has a weakness! Improve it!
- Comments add to the size of the project - comments must be maintained also.
-
- Use comments as headlines, to emphasize structure, give summaries or describe intent
- Make comments easy to maintain

www.ntnu.no



NTNU
Norwegian University of
Science and Technology

Hendseth S., name

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

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?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Self-Documenting Code: Key Points

- 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.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

3.9 Answer to motivational questions

1

Answer to exam question on Code Quality from Continuation 2014

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 :-)



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Answer to exam question on Code Quality from Continuation 2014

- 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.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Answer to exam question on Code Quality from Continuation 2014

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 comments, but this would still be a badly designed module interface.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

4 Ada

Kristoffer Gregertsen kept a 2 hour introduction to the ADA programming language.

4.1 Its learning call

Monday 26/1 Kristoffer Gregertsen will give an introduction to the Ada programming language. (I have guest lectures in another course at this time).

Ada is a large and carefully designed language, well suited for embedded software and some of the choices made on how to do timing and thread interaction are very interesting (and relevant for the exam :-). Check e.g. 1b, 1c, 1d, 1e from the main 2014 exam or 1j in the 2014 continuation exam). These mechanisms will be discussed explicitly in later lectures, but building familiarity with the language, enough for you to understand short pieces of Ada code, and to write small Ada programs (or "Ada pseudo code" on the exam) happens tomorrow.

Two of the exercises will be using Ada.

Do not miss the opportunity to give also this week a good start!

I will see you Tuesday, with the promised introduction to "Basic Fault Tolerance".

Sverre

5 Fault Tolerance Basics

5.1 Its Learning Call

Lecture Tomorrow

We start tomorrow at the proper walkthrough of the fault tolerance part of the course. This is the part of the course that the project is primarily ment to challenge.

Think about this for tomorrow: Is it even possible to make software handle the unexpected errors that exist in it - the bugs that you assume is not there? How can this be done?

Btw. I put out the plan for the semester.

5.2 Learning goals

1

Learning goals; Fault Tolerance Basics

- 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.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

5.3 Intro: Classic bug handling

Bugs:

- Do all systems have them?
- Yes...
- "1bug/50 lines before testing"
- "1/500 at release"
- "1/550 after a year, and then constant"

The "normal" way of handling bugs:

- Make your program to fill the (functional) specification
- Run/Test the program
- Errors happen

- Find "cause" in code
 - erroneous code
 - missing handling of some situation
 - incomplete spec
- Add/Change Code: Fix code or Detect/Handle situation

1

Traditional Error Handling

```
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;
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

This is how it sometimes ends up; One line of readable and intuitive code grows into something not signaling clearly the intent or causality of the code at all.

Are we adding these case's during debugging? Then we would never be certain that we handle all cases... And how do we maintain and test this code... (add even more code for error injection?)

Are we adding the case's systematically from the list of what can happen?

Failure modes of fopen() call from clib:

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 func-
 tion may also fail and set errno for any of the errors specified for the routine open(2). (EEX-
 IST, EISDIR, EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, ENXIO, ENODEV, EROFS,
 ETXTBSY, EFAULT, ELOOP, ENOSPC, ENOMEM, EMFILE, ENFILE)



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Many of these can not even happen... How to test?

Term: Failure modes:

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

Failure modes example

This is practical, project-relevant:

- What are the failuremodes of a TCP connection?, UDP?, Message Queues?...

Example slide: The failure modes of fopen (See the nerdy joke here?) Wel-
come to the dark side!

- Maintainability & modules: Use without knowing internals?
- failure modes destroys module boundaries!!!

(Also Java: "which exceptions can this method throw list")

Testing is not good enough

- Can only show existence of errors
- Cannot find errors in spec.
- Is realistic testing possible?
- Cannot find race conditions
- World interface: - can realistic testing be done (must simulate world...)
- Embedded systems may have higher demands of safety

Error handling destroys readability of code? Even worse: The error modes destroys a module's encapsulation: If the user of a module needs to know all the ways a module can fail, he cannot use the module without knowledge of its internals.

5.3.1 Causes of errors

"Unhandled situations" much more common than "software bugs". A software specification does seldomly handle 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.

5.3.2 This is not good enough!

Anyway: **This is not good enough!**

- We must handle **all** errors - also the unexpected ones
- Since more threads are cooperating in the system, then sometimes they must also cooperate on error handling.

How can we handle unexpected errors, transient errors (Ref. Cosmic Rays?), and yet unfixed errors?

How can we handle the bugs that are still left in the system after testing?

Answer: Fault Tolerance! – The Chapter headline!

Answer: Acceptance tests & Redundancy.

Answer: Merging of error modes

- merge also with the unexpected errors
- And it works wonders for a modules interface.

1

Merging error modes

```
FILE *
openConfigFile(){
    return fopen("/home/sverre/.config.cfg","r");
}

// The caller
void
initialize(){
    ...
    FILE f = openConfigFile();
    struct SConfig * config = readConfiguration(f);

    // The acceptance test:
    if(!checkConfiguration(config)){
        // Fall back to default config, warn user.
    }
    ...
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Technique: Merging of Failure Modes:

- There **are** bugs here - unknown bugs - that we have to handle...
- What is the worst that can happen?
- We must handle this anyway, so group with easier ones (lots of these are known situations!) (Ref. Fopen example - if we cannot open the configuration file, go with the default configuration)
- The cost of increased frequency (but it is still seldom, so...)
- We achieve:
 - A simpler system overall
 - Simpler error handling.
 - Better modules.
 - Handling of unknown errors.

- Examples:
 - All communication errors -> Lost message, resend
 - All SW bugs -> restart.
 - All exceptions -> ABORT/Try Again.

(Relevant for project again: When you are handling restart anyway...)

1

Acceptance test

```
int
checkConfiguration(struct SConfig * config){
    // check range of all variables
    if(config->nNodes < 1 && config->nNodes > 5) return 0;
    ...
    // Check any correlations
    if(config->maxOrders > config->nNodes * 4) return 0;
    ...
    // Possibly configuration have been extended with
    // parameters enabling better checks
    if(config->isSimpleSystem && config->nNodes > 4) return 0;
    ...
    return 1;
}
```

Technique: Acceptance tests (a change of perspective)

- Not the traditional focus on
 - error situations
 - error returns
 - (These will never handle unexpected errors.)
- Rather: We put demands on current status to be ok (**try** to get a configuration from file. Is the current configuration reasonable? If not go with the default)
- NB: Handles unexpected errors!

5.4 Fault Tolerance: Some terms

Terms:

- **error:** Extending the error term to include: unexpected situations + exceptions + situations that happen seldomly + other situations we must handle. (where the methods are the same)
- **Reliability** = consistency between system and spec.

Terms:

Failure	->	Fault	->	Error	->	Failure	->	Fault
Something went wrong to insert a fault		the bug (part of the system)		the thing that happened		Consequence compared to the spec		Failure in subsystem represents a fault in supersystem

Bugfixing - **Fault Prevention** - removes faults. **Fault Tolerance** prevents errors becoming failures.

Term: Fault Prevention

1. Avoidance

- Complete spec.
- Good methodology
- Good language/abstraction
- Good tools.
- (Basicly: Good work)

2. Removal

- Reviews/Inspections
- Verification
- Testing

5.5 Fault tolerance: Some basic techniques

How to make an acceptance test

- Replication Checks
- Timing Checks
- Reversal Checks
- Coding Checks
- Reasonableness Checks
- Structural Checks
- Dynamic Reasonableness Checks

Technique: Recovery Points

- Storing consistent states so that we can reset sensibly after an error.

Technique: Redundancy

- static: All parts are active and errors is masked (n-version prog.)
(What errors does this handle? SW errors?)
- dynamic: Detects & and handles (recovery blocks.)

Both increases the complexity of the system

5.6 N-Version Programming

Static Redundancy: "Classic redundancy", Learning from HW culture.
Masks errors: Handles transient errors well, including unexpected errors (if systems are different).

Figs on blackboard: Failfast, (What do we get from these) (1! error mode!)
pair & spare, 3vs 3 etc.

Failfast:

- Merging failuremodes into one!

- No wrong outputs
- suitable building block
- Also nplex, pair & spare
- NB: Fails more often than one module alone
- Easy to compute reliability figures.

Static redundancy Technique: N-version programming

Fig. 3 boxes and one voter

- Handles transient errors.
- Can handle unexpected errors (if systems are different).

Challenges:

- 4 parts increases complexity significantly. (Number of bugs increases more than linearly? Anyway maintenance is!)
- The driver must be correct.
- Price
- Must avoid the blocks have the same errors
 - N different versions, but
 - * common culture
 - * common spec
 - * difficult parts
- The Voting problem:
 - How to handle:
 - * discrete decisions
 - * More acceptable decisions

5.7 Recovery Blocks

(Dynamic redundancy: redundant in that we have more ways to do the same. Must have detection - acceptance tests.)

Dynamic redundancy Technique: Recovery Blocks (Dynamic Redundancy, Backward Recovery)

- Program more ways to solve the problem
- Make a good acceptance test
- Store recoverypoints to go back to when tests fail
- Try again in other way.
- More threads? Hmmm.

Term: Levels of Fault Tolerance

- Full
- Graceful degradation
- Failsafe

5.8 SW dynamic redundancy

detect	Kode
confine/assess	Design!
recover	Kode
treat	routines

Design is necessary here! For "Traditional error handling" error handling is easily done afterwards. Not here!

- After having detected an error we must know the "error domain" so that the error can be compensated for.
- (With more participants) we must ensure that the consequences of an error is not spread!

2.5.1 Error detection: (talked about this already)

2.5.2 Confinement & Assessment

Question: What must be done to to handle an error?

We must design the program so that this question have an answer (<- Design)

- Be careful not to spread unsafe data.
- Mark or store safe, consistent states (recovery points). <- a technique
- Make modules
 - with clear interfaces
 - with their own consistency checks & error handling.
 - that can participate in creating recovery points.
 - etc...
- Atomic Actions is a confinement pattern...
- ...

Ex. An order is wrongly deleted in the lift system...

2.5.3 Error Recovery: Two types:

Backward Error Recovery i.e. to a recovery point

Forward Error Recovery compensate for the error to get to a consistent state

Fig: Arrows, X back to recovery point or to some other consistent state.

Generalizing to more participants:

- May lead to domino effect

Fig: Domino effect

(Learn to draw this figure :-)

2.5.4: Error Treatment

- send mail
- write to log
- sugges upgrade
- + routines for follow-up.

6 Gray: Fault Model & Software Fault Masking

6.1 Learning Goals

1

Learning goals; Fault model and software fault masking

- 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.

6.2 Its learning call

Tomorrow we will examine a systematic workflow for making fault tolerant software systems, building on top of last weeks techniques of "merging error modes" and "acceptance tests", utilizing redundancy.

We will cover the chapter (copy on its learning) 3.7 in Gray&Reuters book "Transaction processing" going through three fundamental cases: fault tolerant storage, communication and processing. The "process pair" approach to fault tolerant processing is especially relevant, both for how you think on the project and for the exam. Check the continuation exam 2014 task 3a, 3b, 3d and 3g. (3d on explaining process pairs could be emphasized :-))

Do not miss the opportunity to give also this week a good start!

6.3 Intro

What we have: Acceptance tests, Merging failure modes, Redundancy

So, given these elements: How would a fault tolerant system in fact be built?

This chapter describes 3 examples to illustrate.

Chapter title:

Fault Model	Failure modes, probabilities and spec
Software Fault Masking	"Masking errors by redundancy"

Remember the Context here:

- To handle also unexpected errors -> Everything can fail -> Is everything hopeless?
- This kind of fault tolerance is a question of design
- But is it a global design? We would want composition: to be able to build fault tolerant systems from fault tolerant modules.
- This chapter: We are making *the* three basic modules (storage, communication and processing).

The underlying progression in the examples

- Failfast (all failures are detected immediately)
- Reliable (failfast + repair)
- Available (continuous operation)

Rough Process:

1. Find the failure modes.
2. Detect errors / simplify error model.
 - injection of errors for testing
3. Error handling by using redundancy -> reliable & available module

Error injection:

- lets us test error handling!

Brainstorm: **How can we *test* the handling of unexpected errors?**

- Yes, (if acceptance tests are good) inject failed acceptance tests
- Btw: This is also simplified by merging of failure modes.

6.4 Case 1: Storage

Assume unreliable functions: read & write. We are writing sectors to the HD, but imagine an array of data areas.

Step 1) **Failure modes:**

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

(Grays model also includes probabilities. -> We are reducing the chance of failure to less than some number here.)

Step 2) **Detection, Merging of error modes and error injection.**

- Writes also address, checksum, versionId, statusbit to the buffer. (version id isnt used yet, statusbit will be used for injection.)
- All errors -> fail
- Makes "decay" thread that runs in paralell and flips status bits (For testing).

Step 3) **Handling w. redundancy**

- More copies of buffer -> (The version id is used - The newest is returned)
- All reads implies writeback on error.
- Repair thread reading regularly.

Possible to calculate probabilities for error of the total system.

1

store_read

```
/* Reads a block from storage, performs acceptance test and returns status */
bool store_read(group, address,&value){
    int result = read(group,address,value);
    if(result != 0 ||
        checksum fails ||
        stored address does not correspond to addr ||
        statusBit is set){
        return False;
    }else{
        return True;
    }
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

The (error injection) Decay Thread

```
/* There is one store_decay process for each store in the system */
#define mttvf 7E5 /* mean time (sec) to a page fail, a few days */
#define mttfsf 1E8 /* mean time(sec) to disc fail is a few years */
void store_decay(astore store){
    Ulong addr;
    Ulong page_fail = time() + mttvf*randf();
    Ulong store_fail = time() + mttfsf*randf();
    while (TRUE){
        wait(min(page_fail,store_fail) - time());
        if(time() >= page_fail){
            addr = randf()*MAXSTORE;
            store.page[addr].status = FALSE;
            page_fail = time() - log(randf())*mttvf;
        }
        if (time() >= store_fail){
            store.status = FALSE;
            for (addr = 0; addr < MAXSTORE; addr++) store.page[addr].status = FALSE;
            store_fail = time() + log(randf())*mttfsf;
        }
    }
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Reliable Write

```
#define nplex 2 /* code works for n>2, but do duplex */

Boolean reliable_write(Ulong group, address addr, avalue value){
    Boolean status = FALSE;

    for(int i = 0; i < nplex; i++){
        status = status ||
            store_write(stores[group*nplex+i],addr,value);
    }
    return status;
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

reliable_read

```
bool reliable_read(group,addr,&value){
    bool status, gotone = False, bad = False;
    Value next;

    for(int i = 0; i < nplex; i++){
        status = store_read(stores[group*nplex+i],addr,next);
        if (! status ){
            bad = True;
        }else{
            /* we have a good read */
            if(! gotone){
                *value = next;
                gotone = TRUE;
            } else if (next.version != value->version){
                bad = TRUE;
                if (next.version > value->version)
                    *value = next;
            }
        }
    }
    if (! gotone) return FALSE; /* disaster, no good pages */
    if (bad) reliable_write(group,addr,value); /* repair any bad pages */
    return TRUE;
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

This **can** still fail silently! Under which circumstances?

1

The Repair Process

```
void store_repair(Ulong group){
    int i;
    avalue value;
    while(TRUE){
        for (i = 0; i < MAXSTORE; i++){
            wait(1);
            reliable_read(group,i,value);
        }
    }
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

(These code snippets are taken from some presentation held by Gray found on the internet.)

6.5 Case 2: Messages

Fault tolerant communication demands storage & processes in order.

Step 1) Failure modes Message: Lost, delayed, corrupted, duplicated, wrong recipient.

Step 2) Detection, Merging of error modes

- Session Id
- checksum
- ack
- sequence numbers
- All errors -> Lost message

Step 3) Handling w. redundancy

- Timeout & Retransmission.

6.6 Case 3: Processes/Calculations

Note: Does not require safe communication

Step 1) **Error mode**: Does not do the next correct side effect.

Step 2) **Detect&Simplify**: All failed acceptance tests -> PANIC/STOP (Failfast)

Step 3) **Handling w. redundancy**: 3 solutions:

Alt 1: 3a) **Checkpoint-Restart** (Reliable)

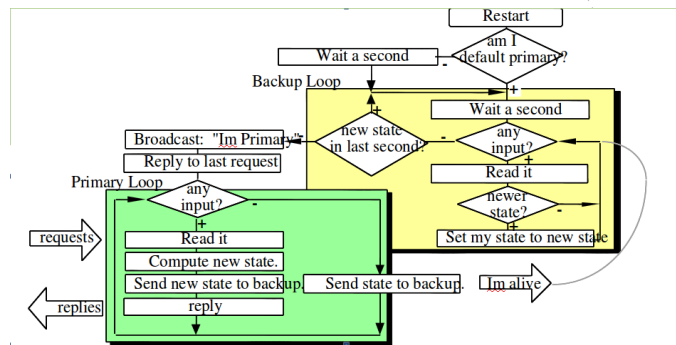
- Writes state to storage after each acceptance test, before each side effect
 - (Note this very important pattern.)
 - (Yes, we are very dependant on good acceptance tests!)
- This gives us error containment.

Alt 2: 3b) **Process pairs** (Gives us available)

- Two processes, Primary & Backup (primary does the work)
- Backup takes over then the primary fails
- IAmAlive messages from primary to backup
- Primary sends checkpoints to backup.

1

Process Pairs



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

process pairs pseudocode

```

repeat
  // Backup mode
  Read checkpoint- & IAmAlive-messages
  Update state
  Until last IAmAlive is too old

  Broadcast: IAmPrimary

  Finish active job // Possibly a duplicate

repeat
  // Primary mode
  if new request/task in job queue then // Part of the state
    do work
    if acceptance test fails then
      restart.
    else
      send checkpoint & IAmAlive // Unsafe communication!
      answer request/commit work.
  else
    if it is time to send then
      send last checkpoint & IAmAlive. // Assumption that there will be more of these.
  
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

NB! This does not rely on safe communication

- Redundancy by resending: Masks communication errors.

- Note new master can answer request sent to old master (no sessions)
- (Communication queues part of state).
- ...

Alt 3 3c) **Persistent Processes**

- Assumes a transactional infrastructure.
- All calculations are transactions ("atomic transformations from one consistent state to another").
- The processes becomes stateless - All states are in database
- For such simple processes the OS can take care of take-over & restart!

In grays vision: **This** is the context of databases!

- But of course: Reliable/available Storage/Communication/Calculations is necessary for making this transactional infrastructure.
- So for us, where it is not, we need to do Alt 1 or 2.

7 **Fault tolerance with more participants: Transaction fundamentals**

7.1 **Its learning call**

Today we will discuss sw dynamic redundancy in systems with more participants (threads, processes, distributed systems). Since these participants cooperates towards the functionality of the total system, they also, sometimes, must cooperate in error handling.

"Atomic Actions" (the term used in the Burns & Wellings chapter) or "Transactions" (used in the "Transaction Fundamentals" chapter) are the standard techniques that provides the necessary framework error containment and handling.

We will be developing these frameworks step by step, focusing on the relevant design patterns on the way.

7.2 Learning Goals

1

Learning goals; Transaction Fundamentals

- 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.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseith S., name

7.3 Intro & Context

Context:

- we search for the "error assessment and confinement" design
- We have more participants (more threads, processes, servers,...)
- The system might be doing more things in paralell (not only like "threads", but also conceptual tasks of the kind that requires cooperation between participants)
- We are continuing from "sw dynamic redundancy" "how to avoid the domino effect"

Motivating examples:

- The process control plant with a lot of local controllers, supervisory tasks, monitoring, safety, optimizing. Operator interfaces with scripting abilities. Multiple modes of operation.

- The mobile sensor network (ships, drones, bouoes)
- The "open embedded infrastructure", imagining cooperation between the safety fire, burglar alarms, smartgrid, internet, home automation

So, how to make "error assessment and confinement" in such a setting?

- To avoid the domino effect we need to negotiate the "starting point".
- and keep track of participants.
- The starting point of what? Enter the *action* - the operation - the thing we want to do.

Warning:

- The chapter is not written for us RT people: It is written for users of an transactional infrastructure that needs to know how it works. We do not have infrastructure; we must learn how to build it.
- Assumed implementation is RPC, but the principles stay the same.

8 Patterns:

- How do they work?
- When are these used / which problems is solved?

7.4 Locking

We need features!!!

How do we ensure nonspreading of intermediate states?

- Locking
- explicit membership (if not static)
- Participants are only allowed to communicate with other participants.

1

Locking; basic fixing of containment

Allocate locks
Do work
Release locks



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

7.5 Backwards error recovery

We will be thinking backward error recovery for now; "Abort" is the common failure mode. It is a bit simpler, and when the framework is in place we will recheck the assumption...

- an action like this with backward error recovery is a *transaction*.
- Errors can not have no consequences outside of the action (error containment) and since no inconsistent, intermediate states are visible to non-participants we call them *atomic actions*.
- an action is an atomic transformation from one state of the system to another.

1

Backwards Error Recovery v1: Recovery Points

```

Allocate locks
Store variable values (recovery points)
Do work, jump to end: if problems
label end:
if(error){
    set variables back to recoverypoint
    status = FAIL;
}else{
    status = OK;
}
Release locks
return status;

```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

7.6 More things to do

1

What if there are more things to do or more participants?

```

status = OK;
if(doWorkX(...) == FAIL) status = FAIL;
if(doWorkY(...) == FAIL) status = FAIL;
if(doWorkZ(...) == FAIL) status = FAIL;
if(status == OK){
    commitWorkX(); commitWorkY(); commitWorkZ();
}else{
    abortWorkX(); abortWorkY(); abortWorkZ();
}

```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

7.7 Generalizing

1

Introducing a Transaction Manager

```
TransactionId tid = tm_beginWork();  
doWorkX(tid,...);  
doWorkY(tid,...);  
doWorkZ(tid,...);  
result = tm_endWork(tid);
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

The Transaction Manager

- `tm_beginWork()`: Creates a transaction; generates unique id, keeps track of members.
- `tm_joinTransaction(participant)`: Adds participant to members
- `tm_endWork(tid)`: Asks all participants for status (Status `prepareToCommit(tid)`), counts votes, `commits(tid)` or `aborts(tid)` and returns status.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

The Resource Manager \equiv The transaction participant

- offers `rm.doWork(tid,...)` functionality
- calls `tm.joinTransaction(me)` if not already a member
- keeps track of locks associated with the transaction
- keeps track of recovery points
- participates in two phase commit protocol (`prepareToCommit()`, `commit()`, `abort()`)
- Note: `prepareToCommit(tid)`, `commit(tid)` and `abort(tid)` can be given reusable forms
 - works only on the RMs data structures!



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Sidetrack: Deadlocks, starvation and timing problems.

- These are allowed to happen! (But not good for Real-Time)
- Standard solution: Give each transaction a deadline, and abort if it does not reach it.
(A job for the TM)



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Sidetrack II: Interaction with the world.

- An effect on the outside world cannot generally be aborted.
- The solution must be dependent on the application:
 - Keep such effects outside of the transaction framework.
 - Wait until commit to do the action.
 - Develop “intelligent” HW that can handle abort or even participate in transactions.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Summary so far

- We now handle all failures detected by our acceptance tests.
- ... as long as the infrastructure (storage, communication, locking) works.
- ... and the processes are not restarted.
- Hmm. The recovery points could be used for restarting?



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Checkpoints revisited

- What if the RM state is too big?
- What if there are more concurrent transactions?
- What about: Only storing the used/locked variables?

1

Writing to the *log*

- Every participant (including the TM) writes to a log what it plans to do and waits until it is confirmed safe before doing it. (Not only side effects; every change of state).
- After restart a process can get back to the current state by "executing" all the logrecords in order.
 - Of course, when executing log records, only the ones belonging to committed transactions must be executed. (Commit and abort logrecords must also be written).
 - Some transactions may lack commit/abort logrecords. — The TM has not decided, or the decision was made when we were down. We need to be able to ask the TM about these.
 - If asked by the TM about transactions that were active before the crash we should vote for abort.

7.7.1 Log II

1

Writing to the *log* – But what if the TM is restarted ?

- After the votes are in, but before the results are sent out, the result is logged.
- All transactions active when the TM restarts are aborted.

7.7.2 Log III

1

Writing to the *log* – A genius extension!

— Also write before-state into logrecords.

- Logrecords can now be executed backwards - actions can be undone.
- We get rid of the recoverypoints!

7.7.3 Checkpoints

1

Writing to the *log* – But, will not the log get infinitely large ?

- Yes; the solution is to write **Checkpoints** to the log.
- Once in a while the complete state, including a list of all active transactions - the checkpoint - is written to the log.
 - Log that is older than the last checkpoint that does not contain any active transactions may be deleted.
 - NOTE: A checkpoint is not a consistent state alone.

7.7.4 LogManager

1

Time to introduce a *Log Manager* ?

- Yes, this can queue more logrecords and optimize disk access.
- ... and if it runs on another machine we may be satisfied with the receipt for received logrecord rather than waiting for the disk access.

7.7.5 LockManager

1

... and a *Lock Manager* ?

- Can "release all locks associated with transaction X".
- Can tidy up properly if we are restarted.
- Can handle resources common to more RMs
- Can be extended with deadlock detection or avoidance algorithms.

7.7.6 Summary 2

1

Summary

- This is philosophy - a way to think about error handling and consistency in distributed systems / systems with more threads/processes/participants.
- For us on the RT / embedded side (that does not have a ready-made transaction infrastructure) the challenge is to adapt this to our use.

7.7.7 Cases

1

Some Cases

- High availability: We want redundancy in calculating power - one computer should be ready to take over for the other if it is restarted.
- Scalability or load balancing: We should be able to put extra resources in at bottlenecks.
- Online upgrade: We need to be able to upgrade the system without losing availability.
- Consistency: How ensure a consistent total system after a subsystem restart ?

7.8 Further Comments on the transaction chapter

Further comments to the chapter:

Optimistic Concurrency Control:

- Assume non-interference, check afterwards and handle as error (/abort)
- Comes out as an alternative to locking.

Two-phase commit optimizations:

- presumed abort (early abort) (Using e.g. ATC)
- one-phase (only one participant - no use voting)
- read-only (commit/abort irrelevant) (Class challenge: do we really need to lock when doing a read-only operation?)
- Last Resource Commit: **One** participant/role gets to wait until after vote to make his decision: He can be responsible for world interaction?

- (Irrelevant for us: Synchronizations (4-fase commit, inn og utsjekking av cache))

Heuristic Transactions:

- What if give a vote and then lose connection?
 - We **have** to do a local guess w. ad-hoc error handling after. The system might become inconsistent/require forward recovery.

Interposition:

- A TM's ability to play the role of an RM ==> Nested **server**-modules.
- A GREAT way of dividing a system into modules!

Nested Transactions:

- (Sverre: Meh, this does not work so well, it is no fruitful way of building super-modules out of submodules, but even the B&W book covers it...)
- Locks are inherited by the super-transaction
 - -> subtransactions can be rolled back even after commit (Sverre: Is this even relevant for us?)

Now all is covered, skip the rest of the considerations.

7.9 Repetition 8 patterns

7.9.1 Locking (w. growing/shrinking phases)

The standard way of getting side boundary. Gives us error containment / Atomicity.

7.9.2 Topphase Commit (The vote)

Gives us End Boundary: Coordinate acceptance tests to avoid the domino effect.

prepareToCommit() and then either commit() or abort() when the votes are collected.

7.9.3 Transaction Manager (Action Controller)

Reusable infrastructure. Keeps track of Membership (Gives us Start Boundary), administers voting. Client calls startWork/ EndWork.

7.9.4 Resource Manager (A new way of making modules: The participant of a (trans)action)

A module/server/class that can take part in a transaction. Supports doWork(tid) calls, will contact TM for membership (join), and supports preparetoCommit(), commit() and abort().

7.9.5 Log

Useful for restart (An alternative to the recovery points in B&W) handles large states and parallel actions. Also useful for undoing logrecords/aborting actions.

Everybody must log: Remember how RM & TM recovers.

Failure recovery:

- RM fails (read log as usual, ask TM about the ones with unknown status)
- TM fails (read log as usual, all transactions active gets aborted)

(TODO: Write out some more detailed recovery scenarios for next year)

7.9.6 Checkpoints

Inconsistent snapshots of the state that lets us avoid the log becoming infinite. The log that is older than the last checkpoint where all actions are finished can be deleted. (- we know the outcome of each of these.)

7.9.7 Log Manager

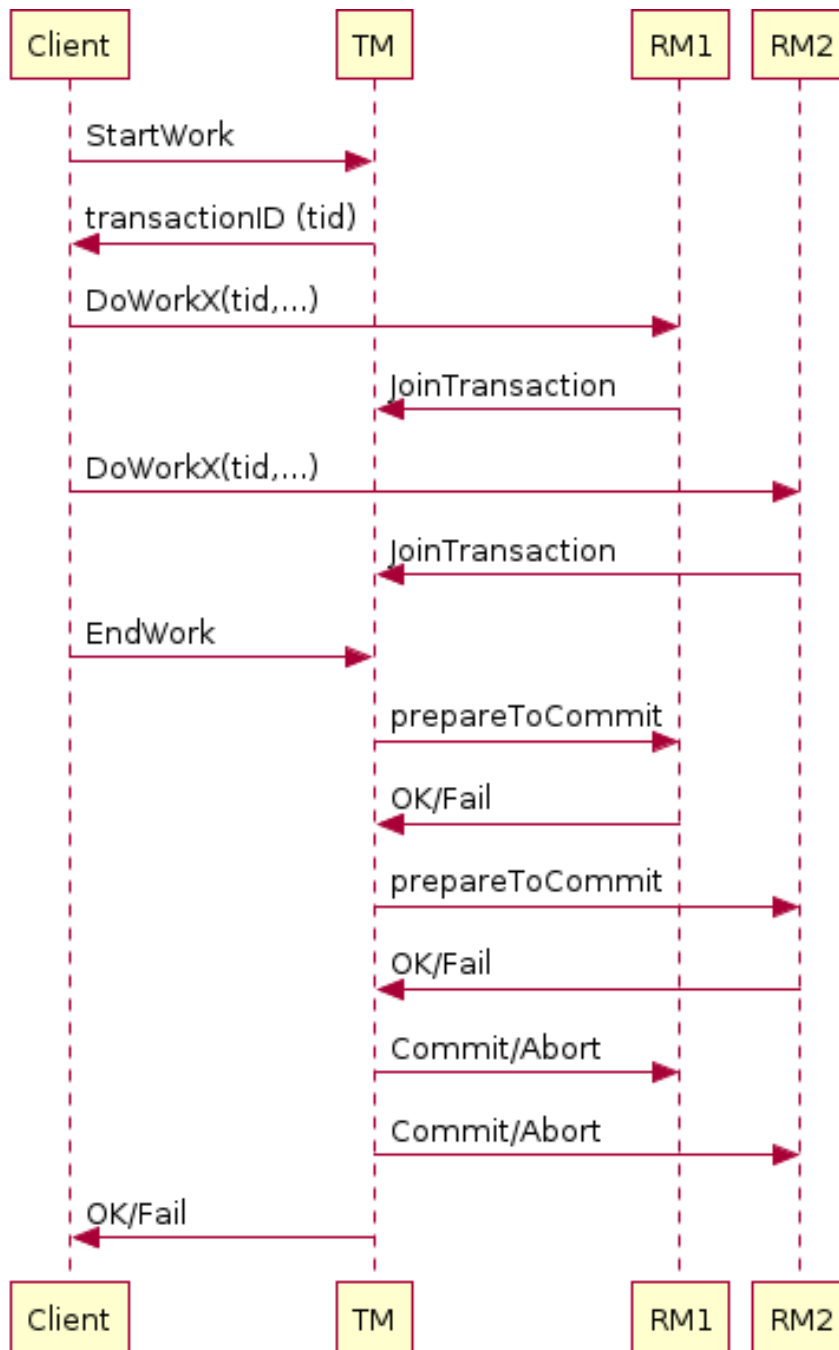
Yes, this can queue more logrecords and optimize disk access. ... and if it runs on another machine we may be satisfied with the receipt for received logrecord rather than waiting for the disk access.

7.9.8 Lock Manager

- Can "release all locks associated with transaction X".
- Can tidy up properly if we are restarted.
- Can handle resources common to more RMs
- Can be extended with deadlock detection or avoidance algorithms.

7.10 Sequence diagram for a transaction.

As a summary: Go through sequence diagram for transactions



Comment on Locking also, to explain side boundary.

8 Atomic Actions and forward error recovery.

8.1 Its Learning Call

This week we will be covering chapter 7 from Burns&Wellings on "Atomic Actions". We will continue from last weeks walk-through of how to construct a Transaction, on one hand being more specific on how it can be implemented in different languages. On the other hand we will generalize Transactions into Atomic Actions by allowing forward recovery.

Do not miss the opportunity to give also this week a good start!

— and for the Tuesday:

Today we will be seeing more code examples on how to implement Atomic Actions in Java and Ada. The examples are taken from the book chapter, are large, and not the easiest read. The relevant language features of Java and Ada (which are the reason the examples and choice of languages are relevant to start with) will be explained. (Java: synchronized methods, wait/notify/notifyAll. Ada: Protected Objects, functions, procedures, entries with guards)

Anders will be using the second hour for an exercise lecture.

— and for the next Monday:

We will finish the Fault Tolerance part of the course this week, ending with how forward error recovery can be embedded into the atomic actions framework.

Forward error recovery with more participants motivates the immediate signaling of error situations, and we will go through POSIX/C, Ada and Javas mechanisms for Asynchronous Notification and Asynchronous Transfer of Control.

Questions/answers as time allows. If destiny and computer skill are on my side a Kahoot session?

Tuesday will be an introduction to Go with an old Kyb Student Tor-Inge Johannessen Eriksen.

Do not miss the opportunity to give also this week a good start!

8.2 Learning Goals

1

Learning Goals: Atomic Actions

- 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, siden and end boundaries.
- Understanding the motivation for using Asynchronous Notification in Atomic Actions
- Knowing how to use the mechanisms for Asynchronous Notification in C/Posix, ADA and Java, including understanding how the ADA version are used for forward recovery in atomic actions.



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

8.3 Atomic Actions Intro

Now Finally, Atomic Actions: AA Has been mentioned before:

- Error handling with more participants.
- Under "Software dynamic redundancy": How to do design for damage confinement & assessment
- How to avoid the dominoeffect.
- A way of dividing the system into modules: a transformation from one consistent state to the next
- In Grays vision for reliable & available computing. (Well, he called it "transactions")

An AA is:

- A design framework for Damage Confinement and Error Recovery.

Other, good, definitions:

- A transaction just without the backward recovery error mode as standard error mode. (Backward error recovery is not that good in a RT setting, remember...)
- indivisible & instantaneous seen from the outside
- A transformation from one consistent state to another.
- A modular computation.

Features of an AA/Transaction

- Explicit participation/membership
- Well defined starting points (**Start boundary**) (possibly w. recovery points in the case of backward error recovery?) (Not necessarily coordinated in time)
- No communication/sharing of info with non-participants. (**Side boundary**)
- A Clear end point (**End boundary**) (Coordinated in time!)

Fig: The Half Circles.

Comments:

- A recovery point, if used at all, does not need to be the complete state.

8.4 AA Standard implementation

Start boundary:

- Dynamic: startwork/joinTransaction
- Hardcoded? (How initiated?/what plays role of "client"?)
-

Side Boundary: Locking

- TwoPhase locking: growing & shrinking.
- The transition must be coordinated,
- No unlocking before safe state (== End Boundary!)

End Boundary:

- Some vote-counting (two-face commit?) - at the end
- Presumed abort (quitting before vote)?
- Synchronization primitive "Barrier"

8.5 How can we implement a AA?

So, AA implementation?

1. Threads, interacting with each other? (like, in the domino-figure)
 - Participants supporting protocols
 - Go example?
 - Ada task
2. Server-processes that handles requests? (same, just structured: Gray)
 - Supporting protocols
3. A number of procedures that is called by participating threads? (the book, Focus on language features)
 - Action == Class or Package/module?
 - Thread interaction through synchronization and shared variables.
4. Threads that is created and terminated with the AA? (Cool, the way to get forward recovery in C)
 - select-then-abort in Ada
 - Asynch. exceptions in Java
 - pthread_{cancel} in POSIX
5. ...

This is a **design framework**

8.5.1 Messagebased rm in go

1

A resource manager in Go

```
func resourceManager(clients chan message, transactionManager chan message)
myTmChannel = make(message);
for {
    select {
    case message <- clients :
        switch message.request
        case "dowork1":
            transactionManager <- myId, "joinTransaction", tid, myTmChannel;
            // store recoverypoint (if not maintaining log)
            // Do work of type 1 using requestData, keep track of errorstatus
            replyChannel <- result;
        case "dowork2":
            // ...
        case message <- myTmChannel
            switch message.request
            case "prepareToCommit":
                transactionManager <- transactions[tid].errorStatus;
            case "commit":
                // Delete recoverypoint, unlock any locks, delete transaction item from mapping
            case "abort"
                // reset to recoverypoint, unlock any locks. delete transaction item from mapping
    }
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

In the Go way of thinking there should be no shared resources. This means no need for locks? (Y/N)

No: One server thread may maintain its local data, but when one action have updated the data, other clients should not be able to access them. Access to a RM in itself might be shared.

8.5.2 Messagebased rm in Ada

1

A resource manager in Ada

```
task body mytask is
  ...
begin
  loop
select
  Accept DoWork do
    ...
    exception // set ErrorFlag
  end Count;
or
  Accept PrepareToCommit(Vote: out Boolean) do
    // Vote = ErrorFlag
  end PrepareToCommit;
or
  accept Commit do
    ...
  end Commit;
or
  accept MyAbort do
    ...
  end MyAbort;
end select;
end loop;
end mytask;
```



NTNU
Norwegian University of
Science and Technology

8.5.3 Messagebased tm in Go

1

A transaction manager in Go

```
func transactionManager(clients chan message, resourceManagers chan message)
for {
    select {
    case message <- clients :
        switch message.request
        case "startWork":
            // generate a transaction id, and reply
        case "endWork":
            // Send PrepareToCommit to all clients
    case message <- resourceManagers
        switch message.request
        case "joinTransaction":
            // Add the clientChannel to the list of channels to ask for prepareToCommit
        case "ok":
            // increment nOfVotes
            // if nOfVotes == nOfParticipants, send commit/abort, depending on failFlag, to :
        case "fail"
            // increment nOfVotes, set failFlag
            // if nOfVotes == nOfParticipants, send abort to all.
    }
}
```



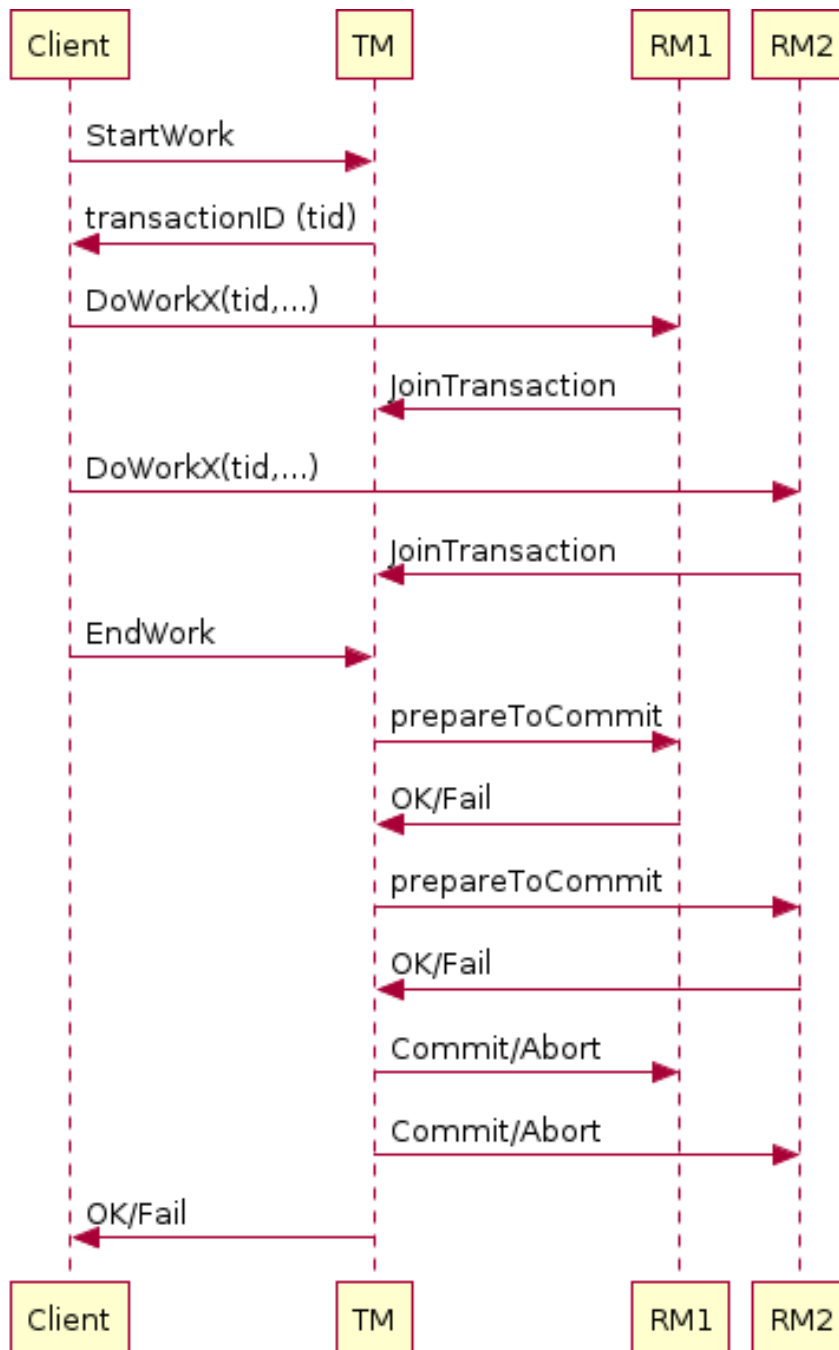
NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

8.5.4 Sequence diagram for a transaction.

As a summary: Go through sequence diagram for transactions



Comment on Locking also, to explain side boundary.

8.6 Sidestep: java monitors

Java Monitors:

- **synchronized** methods:
 - Only one thread can call one of those in an object at the same time. (need object lock) (Classes have locks also).
 - Almost like if all methods reserved a mutex when running.
- wait(): The thread is suspended and the lock released.
- notify()/notifyAll(): Awakens one/all threads
 - but they still need the lock, and cpu, to continue.

8.7 Sidestep: ADA Protected Objects

Note: Cannot block from the inside of a protected object!

ADA: Protected Objects.

- Modules (as with Monitors)
- Functions (readlocks)
- Procedures (writelocks)
- Entries w. Guards (not CondVar)
- Blocking from the inside is an error! (Waiting on other monitor is not blocking Waiting on guard or on task entry is...)
- Guards tests only private variables. -> reevaluation only when exiting entry&proc. // Hvorfor ikke funksjoner?

Term: Guard:

- A test integrated in the language that opens or closes for execution
- In our context: blocking
- Is not (necessarily,only) evaluated sequentially

8.8 Code Example: Making an AA framework in Ada.

Relevant slides from B&W: Chap. 7: 10-16

8.9 Code Example: Making an AA framework in Java (without error handling)

Chapter 10 (ThirdEdition of the books slides) Slide 20-26

Comments on example

- Start Boundary ok
- Nice Action == Class
- No error handling
- No side boundary
- static participants & actions.
- How is one of these actions initiated?
- reusable functionality in controller:
 - membership
 - voting (end boundary)
 - ...

8.10 Adding backwards error recovery to these code examples

Note: The structure is quite static in the examples

- A module per Action!
- Predefined Roles!

We can simply

- maintain a module-flag for error
- Handle error recovery in the finish procedure.

8.11 Asynch Notification Motivation

Where we are - summary:

- We have: A SW structure that **enables** multithread error handling.
- A voting at the end of an AA that facilitates (backwards) error recovery
- But too late: Is it meaningful to wait for everybody to finish after an error have been detected?
 - (Well, we had the "presumed abort" commit protocol optimization, but still, how to abort?)
- AA & Backward Error Recovery == Transactions
- Backward error recovery is simple - handle all errors in the same way

AA & Forward Error Recovery:

- We need a mechanism for distribution of error information:
- (1): Extend the commit protocol?
 - Would not cover all error modes (crashes, hangups)
 - Too late?
- (2) Error info distribution when it happens (Messages, global error variables, POSIX signals ->polling)
 - conceptually good (One error invalidates what is going on)
 - performance ok.
 - Would not cover all error modes
 - Still Too late?
- (3) "Solution": Asynch Transfer of Control (Kap 10 del 2)
POSIX:
 - (Signals & setjmp/longjmp)
 - Kill the processes

ADA:

- structured variant of thread termination based on select.

Java:

- ability to throw exceptions in other thread.

Terms: Asynch Notification (AN) = Lowlevel thread interaction = Asynch Transfer of Control (ATC) (termination)

- Asynch Event Handling. (resumption)

Term: **Resumption vs. Termination mode:** (Important terms used also about signals and asynch transfer of control)

- If we continue where we was (after the throw)? -> Resumption
- If we continue somewhere else (i.e. terminating what we were doing) -> Termination

Resumption most often leads to polling.

Why avoid Asynch Transfer of Control?

- It leaves us in undefined state!
- But then; If we have...
 - Full control over changed state
 - A lock manager that can unlock on behalf of killed thread
- ...it might be doable.

8.12 POSIX: Signals and threadCancel

AN: POSIX signals (Resumption)

- Modeled after HW interrupt
- Can be sent to the correct thread
- Can be handled, ignored, blocked -> the domain can be controlled
- Often leads to polling

- Could rather skip the signal and poll a status variable or a message queue?
- Useless (?)

But: Threads can be killed

- Can make termination model by letting domain be a thread
- can still control domain by disabling 'cancelstate'

Why do we need to control the domain?

- Some operations are critical
- imagine the implementation of the lock-manager
- locking must be atomic.

8.13 C: setjmp/longjmp

Copied from the wikipedia: Simple Example:

The example below shows the basic idea of setjmp. There, main() calls first(), which in turn calls second(). Then, second() jumps back into main(), skipping first()'s call of printf().

1

setjmp/longjmp

```
#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 - making setjmp no
}

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;
}
```

www.ntnu.no

 **NTNU**
Innovation and Creativity

Hendseth S., name

When executed, the above program will output:

second main

8.14 ATC Implementasjon Java

ATC in **Java**:

- (Before: suspend, resume, stop)
- Still have: interrupt, (+isInterrupted)
 - waking from wait state
- destroy.

ATC i **RT Java**

- Built on Java exceptions rather than select as in ADA
- An interruptible function must declare AIE
- Will never interrupt in any synchronized or in a finally block
- Can be blocked
- Each **thread** is associated with a generic AIE
- Interrupt() wil now throw the generic AIE asynchronously.
- Must then test and clear on the generic
- B&W tries to make sense out of this. Let **us** not.

8.15 Select then abort in ADA

```
select
  delay 1.0 -- or any event.
then abort
  -- Do the real work
  -- Will be interrupted if it times our or the event happens.
end select;
```

Classroom challenge: How to make forward error recovery with this?

Hint

- Use the controller to communicate with the participants
- Note: Cannot block from the inside of a protected object!

On the blackboard:

AA med Forward Error Recovery i Ada:

Role1:

```
begin
  select
    controller.waitFor(E1) // Event
    raise(E1)
  then abort
  begin
    // Useful work
  exception E2
    // Noe gikk galt - alle må få vite
    controller.signalAbort(E2) // Proc
// Gjør controller villig til å være med på waitFor (N ganger).
  end
exception
  // Exceptions her er felles for alle trådene!
  when E3
    // handle error
    controller.Cleanup(ok/fail) // Proc: avgi stemme -> forbereder for backward e
    controller.WaitCleanup(Decition) // høyere-nivå feilhåndtering
    // Håndter evt. Decition==fail
  end

procedure Controller.signalAbort(E)
begin
  Killed := True;
  Reason := E;
end signalAbort;
```

```

Entry Controller.WaitAbort(E: out ExceptionId) when Killed is
begin
    E := Reason;
    Informed := Informed + 1;
    if Informed = 3 then
        Killed := False;
        Informed := 0;
    end if;
end WaitAbort()

```

Drawing the arrows of execution flow in case of error.

9 Shared Variable Synchronization

9.1 Its learning call

Remember the "introduction to Go" lecture today.

It is not entirely clear to me whether it will take one or two hours. If there is significant time left when Tor-Inge has finished, I will motivate the second part of the course; Shared Variable Synchronization. Underlying, recommended Curriculum here will be the copied chapter five of the Burns and Wellings book, and "the little book of semaphores", which is freely available.

See you

—

The second part of the course, in addition to scheduling, of what I have called "classic real-time programming" is shared variable synchronization.

This is basically how to avoid problems like the "i=i+1/i=i-1"-problem and is common to also other kinds of multithreaded programming.

We will be progressing through lowlevel primitives (like suspend/resume) through semaphores and preparing for the modern mechanisms we have in POSIX/Java and Ada today.

Knowledge of the strengths and limitations of these mechanisms is a very central part of the curriculum.

If you want to prepare, read any chapter of "the little book of semaphores" that is available on the internet. This illustrates much of the challenges in

this area, and trains you in the way of thinking when using shared variable synchronization.

See you tomorrow.

— Hi, there will be proper lectures tomorrow, to make up for the lectures lost during the excursion weeks.

The theme is still shared variable synchronization, continuing from the semaphores we arrived at Friday, and going through the more highlevel mechanisms leading up to the ones we find in POSIX/Java and Ada.

My quest here is (in addition to teaching you the language mechanisms with their strengths and weaknesses) to train your skill at recognizing race conditions. Bring pen and paper :-)

See you.

—

The second part of the course (after having finished "fault tolerance") will be what I have called "classic real-time programming". This is, in addition to "scheduling", "shared variable synchronization". We start tomorrow! A large part of the lectures tomorrow will be motivating the problems of shared variable synchronisation; "What is the problem?" and "Why is it hard?". Check the learning goals. Quite deep skills are required on this topic.

Shared variable synchronization starts with how to avoid problems like the "i=i+1/i=i-1"-problem, but, having solved this, grows into the plethora of challenges on "deadlocks", "livelocks", "race conditions", "starvation", etc.

Curriculum on this part will be "the little book of semaphores" available on the internet and Burns and Wellings chapter 5.

Do not miss the opportunity to give also this week a good start!

—

After getting into the "detecting race-conditions" frame of mind yesterday, we will today take a few steps back and see how the development of a real-time kernel could happen, to put the different synchronization features into a "historical" and hardware perspective. The discussion spans the progression from non-preemptive systems to interrupts and preemption, spin locks, blocking and suspend/resume, events, locking, semaphores (finally!), conditional critical regions, monitors, and finally the modern synchronization mechanisms of Java/Ada and POSIX/C.

But testing yesterdays skills: The following code has a bug/ a race condition. Where? ...and what is the symptom?

1

A hard-to-find bug

```
void allocate(int priority){
    Wait(M);
    if(busy){
        Signal(M);
        Wait(PS[priority]);
    }
    busy=true;
    Signal(M);
}

void deallocate(){
    Wait(M);
    busy=false;
    waiting=GetValue(PS[1]);
    if(waiting>0) Signal(PS[1]);
    else{
        waiting=GetValue(PS[0]);
        if(waiting>0) Signal(PS[0]);
        else{
            Signal(M);
        }
    }
}
```

Having done our best to adapt to the "recognizing race conditions" way of studying code last week, and discussing non-preemptive scheduling, we will Monday 2/3 start by taking a few steps back and see how the development of a real-time kernel could happen, putting the different synchronization features into a "historical" and hardware perspective. The discussion spans the progression from interrupts and preemption, spin locks, blocking and suspend/resume, events, locking, semaphores (finally!), conditional critical regions, monitors, and finally the modern synchronization mechanisms of Java/Ada and POSIX/C.

Do not miss the opportunity to give also this week a good start!

Continuing our walkthrough of synchronization mechanisms and after finishing semaphores yesterday, we continue into the "Conditional Critical Region" and the "Monitor", before starting the walkthrough of state-of-the-art, real-world, synchronization primitives in POSIX, Java and Ada.

See you all.

—

Tomorrow morning we will go through Ada and Javas synchronization mechanisms in detail, and work through a number of examples.

Ada and Java are in this context just suitable example languages letting us carry our awareness of race conditions, deadlocks and starvation up from semaphores to the current abstractions of monitor-inspired conditional synchronization (POSIX/Java) and ConditionalCriticalRegion-inspired avoidance synchronization (Ada).

The semantics are not that difficult to learn; We are trying to build your skills here.

(This is the final part of the shared variable synchronization part of the course. Two themes remain: Scheduling/schedulability proofs, which will be kept by Amund and my quite theoretical introduction to communicating systems.)

Do not miss the opportunity to give also this week a good start!

9.2 Learning Goals.

1

Learning Goals: Shared Variable Synchronization

- 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 requeue and entry families) and Java.

9.3 Standard problems

Ref. $i=i+1$ problem: (find the slide)

Solved with semaphores... Is everything ok now?

Some standard problems:

- Deadlock: system blocked in circular wait
- Livelock: system locked in a subset of states (like deadlock but we use CPU)
- Starvation: A thread does "by accident" not get necessary resources. Ex: Unfair scheduling. (Ref discussion on whether you should make assumptions on how the scheduling works. Ref. Go vs. Occam: & Who is waked by a signal?)
- Race Condition: A bug that surfaces by unfortunate timing or order of events.

Class task: (You should know this) Classical example of race condition:

```
t1(){
    if(!ready) suspend();
    ...
}
t2(){
    ready = 1;
    resume(t1);
}
```

9.4 Making a barrier

Introducing the little book of semaphores: Read a few random cases to get into the way of thinking.

Sequence of slides:

1

What is a barrier

Rendezvous, Critical point, AA exit point, two-phase commit protocol.
All threads wait for all.
Class task: How to make it with semaphores?

Each thread have their initialization to do - and have to wait for the others.

Classroom task: Try!

Hints:

- Assume N threads, N known, count them as they arrive.
- One semaphore protecting N , one being the barrier.

1

Failed attempt 1

```
// rendezvous
mutex.wait()
count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
// critical point
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

Questions:

- (Easy) Why does this not work?
- (Hard) It might work sometimes... How?

1

One-time barrier Solution

```
// rendezvous
mutex.wait()
count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

// critical point
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

This is the "turnstile" semaphore pattern.

Questions:

- In which state is barrier after all threads have been through?
- If we were to fix this... How?

Now we want a reusable barrier: Imagine a while loop around the code.

1

Failed Attempt 2

```
while(1){
    // rendezvous
    mutex.wait()
    count = count + 1
    if count == n: barrier.signal()
    barrier.wait()
    barrier.signal()
    mutex.signal()

    // critical point
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

Question:

- (Easy) What about this? Does it work?

1

Failed Attempt 3

```
while(1){
    // rendezvous
    mutex.wait()
    count += 1
    mutex.signal()
    if count == n: turnstile.signal()

    turnstile.wait()
    turnstile.signal()

    // critical point

    mutex.wait()
    count -= 1
    mutex.signal()
    if count == 0: turnstile.wait()
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

Questions

- How does this not work?
 - One obvious by now: Too many signals.

1

Finally, Solving problem 1, but...

```
while(1){  
  // rendezvous  
  mutex.wait()  
  count += 1  
  if count == n: turnstile.signal()  
  mutex.signal()  
  
  turnstile.wait()  
  turnstile.signal()  
  
  // critical point  
  
  mutex.wait()  
  count -= 1  
  if count == 0: turnstile.wait()  
  mutex.signal()  
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

Questions:

- So what is the problem now?

We need two turnstiles! Each unlocking the other.

1

The Solution

```
while(1){
    // rendezvous
    mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait() // lock the second
        turnstile.signal() // unlock the first
    mutex.signal()

    turnstile.wait() // first turnstile
    turnstile.signal()

    // critical point

    mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait() // lock the first
        turnstile2.signal() // unlock the second
    mutex.signal()

    turnstile2.wait() // second turnstile
    turnstile2.signal()
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

This works:

Question:

- Are you convinced? If you were to bet your life?

9.5 The cigarette-smokers problem exam task

...from the v2014 exam. Just walkthrough of 1e-1g

9.6 Standard applications

Some standard applications of synchronization

- Critical Section - Code that must not be interrupted
- Mutual Exclusion - More pieces of code that must not interrupt each other
- Bounded Buffer. - Buffer with full/empty synchronization
- Read/Write Locks - (Concurrent readers are ok)

- Condition Synchronization - Blocking on event or status. (Guards e.g.)
- Resource allocation (More than mutual exclusion! Ref. the "lock manager")
- rendezvous/barriere - synchronization point (Ref. AA end boundary)
- Communication -
- Broadcast -
- Lock manager (reserve A, B or both)

Now take the hard-to-find bug.

9.7 Some harder problems

Classroom task: Lock Manager: If we want A, B, or both; How can we do this with semaphores.

1

Allocating A, B or both

```
allocate(resourceList){
    wait(LockManager);
    if(lm_resourcesAreFree(resourceList)){
        lm_reserve(resourceList);
        signal(LockManager)
        return;
    }else{
        qn = allocateQueueNumber();
        store_request(qn,rList);
        signal(LockManager);
        wait(semQ[qn]);
    }
}

free(rList){
    wait(LockManager);
    lm_unreserve(rList)
    while(Any requests fulfillable){
        set qn & rl for fulfillable request
        lm_reserve(rList);
        signal(semQ[qn])
    }
    signal(LockManager);
}
```



NTNU
Norwegian University of
Science and Technology

Bad code: www.ntnu.no

Hendseth S., name

Questions:

- Where is the problem?

- Answer: Might wake the wrong client if it is not yet waiting...

Possible problem: Protect a number of functions, where some of them calls each other. Possible problem: Lock to more than one user a'la AA/Transactions.

These are difficult problems to solve with semaphores; Often leads to race conditions.

Another hard-to-find bug:

The setting is; We have two priorities for getting the resource. We want to wait in the right queue...

1

A hard-to-find bug

```
void allocate(int priority){
    Wait(M);
    if(busy){
        Signal(M);
        Wait(PS[priority]);
    }
    busy=true;
    Signal(M);
}

void deallocate(){
    Wait(M);
    busy=false;
    waiting=GetValue(PS[1]);
    if(waiting>0) Signal(PS[1]);
    else{
        waiting=GetValue(PS[0]);
        if(waiting>0) Signal(PS[0]);
        else{
            Signal(M);
        }
    }
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

Questions:

- Where is the bug?
- What are the consequences?
- Can you solve it?

This is from B&W!

9.8 Pros and cons of preemptive and nonpreemptive scheduling

NonPreemptive Scheduling Preemptive Scheduling (Cyclic Executive)

- Simple, intuitive, predictable
- No Kernel
- Threads must cooperate <- a form of dependency breaking module boundaries
- Heavy threads must be divided
- blocking library functions?
- Unrobust to errors
- Unrobust to (heavy) Error handling
- tuning at end of project

Preemptive Scheduling

- No problems with heavy threads
- Works well with competing, non-interacting threads
- Concurrent Access-problem: $t1()\{ t2()\{ i=i+1; i=i-1; \} \}$ Not only memory & CPU: All kinds of resources are shared.
- predictability
- race conditions
- tuning at end of project

9.9 Making a basic r-t kernel

Perspective: We are making a preemptive kernel!

Feature 1 (preemption): Switch between threads

- handle timer interrupt
- Store all registers (including ip & sp)
- organize queue of processes (Round Robin e.g.)
- Can synchronize by: while(!ready); (busy waiting, "spin locks")

Question: Can we get mutual exclusion with spin locks?

Solution:

```
flag1 = 1
turn = 2;
while(flag2 == 1 && turn == 2){}
    // Vi may run
flag1 = down;
```

Feature 2: Priorities

- Sort the process queue (or have more queues)

Feature 3 (synchronization):

- Queue of blocked processes
- suspend & resume.
- fixes the bad performance of spin locks.

(Home)Question: Mutual Exclusion, with suspend & resume?

- Fortsatt kompilerert å unngå race conditions

Examples of bad solutions with race conditions

```

t1()
    while(busy == 1) suspend();
    // We own resource
    busy = 0;

    resume t2 // No

t1()
    while(TestNSet(busy,1) == 1) suspend();
    // We own resource
    busy = 0;

    resume t2 // No

```

Classical example:

```

t1(){
    if(!ready) suspend();
    ...
}
t2(){
    ready = 1;
    resume(t1);
}

```

Feature 4: (necessary when publishing suspend&resume or another sync. mechanism) protection of the schedulers data structures.

- Disable Interrupt. (not for large functions!)
- test&set instructions (or swap)
- And still: spin locks

(spin locks still relevant for some multicore mechanisms)

Feature 5: "Better" synchronization mechanisms.

- Events (wait(e), signal(e))
 - Just like anonymous suspend&resume
 - Separate queues for each event
 - still race condisions.
- Semaphores! (Usable!)

Recommend: "The little book of semaphores".

9.10 Semaphores

9.10.1 Presenting semaphores

Semaphores:

- Solves Mutual Exclusion
- Solves Conditional Synchronization
- Solves basic resource allocation

Classroom Task: Bounded buffer w. semaphores: Make the put and get functions.

Hints:

- You have the "enterIntoBuffer" and "getFromBuffer" functions, and the Element type.
- How many reasons for blocking are there?
 - Use one semaphore for each

1

Bounded buffer with semaphores

```
SEMAPHORE NInBuffer(0),NFree(N),Mutex(1);

put(e){
    wait(NFree);
    wait(Mutex);
    // enter into buffer
    signal(Mutex);
    signal(NInBuffer);
}

get(e){
    wait(NInBuffer);
    wait(Mutex);
    // get e from buffer
    signal(Mutex);
    signal(NFree);
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Semaphore drawbacks: (Continuing feature list)

- Breaks modules (both ways)
 - does not scale!
- Deadlocks
 - global analysis -> does not scale
- Cannot release "temporarily"

9.11 Conditional Critical Regions

Conditional Critical Regions

- (Imagined) language support for mutual exclusion + guards.
- Named blocks are regions
- Regions of same name mutually excludes
- Regions have guards
- It is more structured, but still not a proper module
- Difficult to avoid bad performance when evaluating guards.
- (make the regions entries (if guards) or functions/procs (if not) in protected objects, restrict guards to private variables (and add requeue) and we have ADA)

9.11.1 Sidestep: Guards

Term: Guard:

- A test integrated in the language that opens or closes for execution
- In our context: blocking
- Is not (necessarily,only) evaluated sequentially

Classroom questions:

- When are guards (re)evaluated?

- How do ADA make it?

1

Bounded buffer with CCR

```
task producer;
loop
  region buf when buffer.size < N do
    -- place char in buffer etc
  end region
end loop;
end producer

task consumer;
loop
  region buf when buffer.size > 0 do
    -- take char from buffer etc
  end region
end loop;
end consumer
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

9.12 Monitors

Monitors

- The region is a collection of procedures & local variables.
 - a module, great!
- No guards, but condition variables
 - Like events – "named suspend/resume" – but safe since they are only accessed from inside of the monitor
 - (Anonymize and we have java)
- Blocking on condition variable releases monitor temporarily
- Awakened process must wait for monitor lock (and cpu)
- Signal() awakens **one** process.

- Waiting on a condition variable must be done in a safe place, but it turns out that this is relatively easy.

Bad Code:

```
monitor M
unsafe(){
  t = i;
  wait(C);
  i = t+1;
}
```


1

Bounded buffers with monitors

```
procedure put(e);
begin
  if NumberInBuffer = size then
    wait(spaceavailable);
  end if;
  // Enter e into buffer
  signal(itemavailable)
end append;

Element procedure get();
begin
  if NumberInBuffer = 0 then
    wait(itemavailable);
  end if;
  // get an element from the buffer and return it.
  signal(spaceavailable);
end take;
```

www.ntnu.no

 NTNU
Norwegian University of
Science and Technology

Hendsest S., name

9.13 POSIX: Mutexes & Condition variables

POSIX: Almost like monitors: Mutex & Condition variable

- Make the module yourself
- signal() awakens all

Bounded buffers with posix

```

void put( int e ) {
    pthread_mutex_lock( &mutex );
    while ( numberInBuffer == size )
        pthread_cond_wait( &spaceAvailable, &mutex );
    // do work
    pthread_cond_signal( &itemAvailable );
    pthread_mutex_unlock( &mutex );
}

int get() {
    pthread_mutex_lock( &mutex );
    while ( numberInBuffer == 0 )
        pthread_cond_wait( &itemAvailable, &mutex );
    // do work
    pthread_cond_signal( &spaceAvailable );
    pthread_mutex_unlock( &mutex );
    return /* ... */;
}

```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

9.14 Criticism of monitors

- Locking in nested call does not release outer monitor(s)

Classroom task: Why? (Answer: These are not necessarily the safe places)

- -> Destroys Composition - we cannot build supermodules from modules.
- Deadlocks & Race Conditions as with semaphores (Just for slightly larger systems)
- What if we need other synchronization than Condition variables?
- Conflict: High vs. low level of abstraction?
- More complex than semaphores...

9.15 Java: Synchronized Objects

Java:

- Object as monitor (one lock per object/class)
- synchronized methods locks
 - may have also non-synchronized methods
 - But this is very different from r/w locks
- calls (also from the outside!): `o.synchronized(){...}` also locks (terrible! breaks encapsulation completely. Use either or!)
- No explicit Condition variable (more as suspend & resume again, but safe(r) this time, since wait is always called from a protected region)
- `wait()` - releases only the inner lock
- `notify()` - Awakens **one** process.
- `notifyAll()` - Awakens all waiting processes
- May also be awakened by `t.interrupt()` -> `InterruptedException`

9.16 Java Example: Bounded buffer

1

Bounded Buffer in Java

```
public synchronized void put(int item) throws InterruptedException
{
    while (numberInBuffer == size) wait();
    last = (last + 1) % size ;
    numberInBuffer++;
    buffer[last] = item;
    notifyAll();
}

public synchronized int get() throws InterruptedException
{
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size ;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

9.17 Java discussion points:

Notify vs NotifyAll

- Why notifyAll & while loops around wait?
- (Ref. Monitors awakenig one, but POSIX awakeing all.)
- "Conditions" are nameless in java, if there are more conditions for blocking, we would not know that notify wakes the right one.
- Conclusion: "Always" use notifyAll and while loops!

Inheritance Anomaly:

- What a sweet and obvious thought to integrate the monitor into the object.
- Unfortunately inheritance works surprisingly badly.

- synchronization and inheritance
- <http://wisnesky.net/anomaly.pdf>
- <http://eprints.soton.ac.uk/262297/1/anomalySurvey.pdf>
- Rule of thumb: do not use notify()
 - you have no control on who is awakened.
 - * -> Possible deadlock if more than one **type** of waiting?
 - we do now know because of inheritance/extensions/maintenance
- Design-pattern: while(!cond) wait(); % monitor forhindrer race condition. & notifyAll();
- (This is more flexible than condition variables...?)
- Inheritance Anomaly
 - Can be solved by prev point, but not always (eks p269-270)

(Book has also example of R/W locks without notifyAll...)

From <http://wisnesky.net/anomaly.pdf> Simply put, the anomaly is a failure of inheritance and concurrency to work well with each other, negating the usefulness of inheritance as a mechanism for code-reuse in a concurrent setting.

Sverre: The union of **classes** and **monitors** is not perfect: Inheritance and reuse means something for classes but not for monitors!

9.18 Java Example: Read/Write locks in Java

1

Read/Write locks in Java - I

```
public synchronized void StartWrite()
    throws InterruptedException
{
    while(readers > 0 || writing)
    {
        waitingWriters++;
        wait();
        waitingWriters--;
    }
    writing = true;
}

public synchronized void StopWrite()
{
    writing = false;
    notifyAll();
}
```

```
public synchronized void StartRead()
    throws InterruptedException
{
    while(writing || waitingWriters > 0) wait();
    readers++;
}

public synchronized void StopRead()
{
    readers--;
    if(readers == 0) notifyAll();
}
```

...and the obligatory comment on starvation...

9.19 Ada: Protected Objects

ADA: Protected Objects.

- Modules (as with Monitors)
- Functions (readlocks)
- Procedures (writelocks)
- Entries w. Guards (as with CCR)
- Requeue (know semantics abition here; no pseudocode)
- Entry Families (know semantics abition here; no pseudocode)

1

Read-Write locks in Ada

```
protected body Shared_Data_Item is
  function Read return Data_Item is
  begin
    return The_Data;
  end Read;
  procedure Write (New_Value : in Data_Item) is
  begin
    The_Data := New_Value;
  end Write;
end Shared_Data_Item;
```

R/W-locks in ada are trivial:
but then:

- We kind of assume that writers should have priority, that reading is more common than writing?
- (B&W slides also say that this cannot be used from another protected object because of blocking???)

- Blocking from the inside is an error! (Waiting on other monitor is not blocking Waiting on guard is...)
- Guards tests only private variables. -> reevaluation only when exiting entry&proc.

1

Bounded Buffer in Ada

```
protected body Bounded_Buffer is
  entry Get (Item : out Data_Item) when Num /= 0 is
  begin
    Item := Buf(First);
    First := First + 1; Num := Num - 1;
  end Get;
  entry Put (Item : in Data_Item) when
    Num /= Buffer_Size is
  begin
    Last := Last + 1; Num := Num + 1;
    Buf(Last) := Item
  end Put;
end Bounded_Buffer;
My_Buffer : Bounded_Buffer;
```

Bounded Buffer i Ada (1 slide! quite elegant!)

www.ntnu.no

- has count-attribute for number of waiting processes.

Classroom question: What is the type of First and Last?

BB in Ada - the Bounded Buffer type

```
Buffer_Size : constant Integer :=10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get (Item : out Data_Item);
  entry Put (Item : in Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Num : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendsest S., name

Classroom question: How is it with composition? Can we have Protected objects using other protected objects?

- Waiting from inside is a runtime error: An exception will be thrown (the third alternative to "releasing/not releasing the outer monitor")
- But more elegant than catching the exception is to select on the entry!

1

A barrier in Ada

```
protected body Blocker is
  entry Proceed when
    Proceed'Count = 5 or
    Release is
  begin
    if Proceed'Count = 0 then
      Release := False;
    else
      Release := True;
    end if;
  end Proceed;
end Blocker;
```

A barrier in Ada?

www.ntnu.no

Hendseth S., name

Classroom challenge: Does it work?

- It is one-time.
- has "Requeue" for "temporal release" - "Give up and retry, possibly in another queue"
- has "entry families" - parameterized entries.

Slides on this are alternatives:

- Entry Families 63-64
- Readers/Writers problem with priorities to writers: 74-78

9.20 Ada Example: Update/Modify/Lock

Discussion: In Ada, with a lot of readers (calling functions/entries) and writers (calling procedures). How is the queue organized?

Sverres answer: FIFO. Why? To avoid starvation.

But if we want something else? Two functions with other priorities?

Here: Update should have priority over Modify. (For good measure; add lock)

1

Update/Modify/Lock in Ada - I

```
protected Resource_Manager is
  entry Update(...);
  entry Modify(...);
  procedure Lock;
  procedure Unlock;
private
  Manager_Locked : Boolean := False;
  ...
end resource_manager;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

(Note: The lock part of this is really "server state")

1

Update/Modify/Lock in Ada - II

```
protected body Resource_Manager is

  entry Update(...) when not Manager_Locked is
  begin ... end Update;

  entry Modify(...) when not Manager_Locked and
Update'Count = 0 is
  begin ... end Modify;

  procedure Lock is
  begin Manager_Locked := True; end Lock;

  procedure Unlock is
  begin Manager_Locked := False; end Unlock;

end Resource_Manager;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

(So, the possible weaknesses of the default R/W locks of Ada can be easily compensated for.)

Classroom task:

- Would it be significantly different if given `allocateupdate` & `allocatemodify` form? (Answer: No, add a busy flag, and the appropriate guards.)

9.21 Java Example: Update/Modify/Lock - Make it in Java

1

Update/Modify/Lock in Java

```
synchronized Modify(...){
    while(locked || NWaitingUpdaters > 0) wait();
    ...
    notifyAll();
}
synchronized Update(...){
    while(locked){
        NWaitingUpdaters++;
        wait();
        NWaitingUpdaters--;
    }
    ...
    notifyAll();
}
synchronized lock(){
    locked = true;
}
synchronized unlock(){
    locked = false;
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Class challenge:

- What is best Ada or Java?
- How is the lock calls queued? Do we dare making lock and unlock not synchronized?
- For POSIX what would be the difference (from Java)?

9.22 Java example: Request Parameters

1

Handling request parameters in Java

```
public class ResourceManager
{
    private final int maxResources = ...;
    private int resourcesFree;

    public ResourceManager() { resourcesFree = maxResources; }

    public synchronized void allocate(int size)
    {
        while(size > resourcesFree) wait();
        resourcesFree = resourcesFree - size;
    }

    public synchronized void free(int size)
    {
        resourcesFree = resourcesFree + size;
        notifyAll();
    }
}
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Class Question:

- Out of many requests; Who will get the allocation done?
- How to avoid starvation...

9.23 Ada: Request parameters

ADA: Not so easy: Cannot test on parameters in guards.

- Option 1: "Double Interaction", Two calls: The first nonblocking announcement, the second blocking
 - Not atomic: makes error recovery difficult, destroys side boundary of AA.
 - Does this even work at all in Ada? B&W does not give convincing example.
- Option 2: Requeue

- Requeue helps slightly (Only server side sees more states.).
- Option 3: Entry families instead of parameters?

Requeue:

- Leaves the calling code there and then
- Can be another entry (must have the same parameters)
- Can be in another object
- Abortable while blocked? Can be controlled "with abort"

Entry families:

- An indexed "array" of entries.

9.24 Ada example with requeue I

1

Request parameters in Ada with Requeue - I

```

type Request_Range is range 1 .. Max;
type Resource ...;
type Resources is array(Request_Range range <>) of Resource;

protected Resource_Controller is
  entry Request(R : out Resources; Amount: Request_Range);
  procedure Release(R : Resources; Amount: Request_Range);
private
  entry Assign(R : out Resources; Amount: Request_Range);
  Free : Request_Range := Request_Range'Last;
  New_Resources_Released : Boolean := False;
  To_Try : Natural := 0;
end Resource_Controller;

```


1

Request parameters in Ada with Requeue - II

```
protected body Resource_Controller is

  entry Request(R : out Resources; Amount: Request_Range)
    when Free > 0 is
  begin
    if Amount <= Free then Free := Free - Amount;
    else requeue Assign; end if;
  end Request;

  procedure Release(R : Resources; Amount: Request_Range) is
  begin
    Free := Free + Amount;
    -- free resources
    if Assign'Count > 0 then
      To_Try := Assign'Count;
      New_Resources_Released := True;
    end if;
  end Release;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Request parameters in Ada with Requeue - III

```
entry Assign(R : out Resources; Amount: Request_Range)
  when New_Resources_Released is
begin
  To_Try := To_Try - 1;
  if To_Try = 0 then
    New_Resources_Released := False;
  end if;
  if Amount <= Free then
    Free := Free - Amount;
    -- allocate
  else
    requeue Assign;
  end if;
end Assign;
end Resource_Controller;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Class challenge:

- Who is awoken?

- There are some ugly assumptions making this code work: Which?
 - it assumes FIFO queuing!
 - it assumes no new requests while handling a round of assigns!

(Both may be true, but...)

9.25 Ada example with requeue II

1

Request parameters in Ada with Requeue - 2-I

```
protected Resource_Controller is
  entry Allocate(R: out Resource;
               Amount : Request_Range);
  procedure Release(R: Resource;
                  Amount : Request_Range);
private
  Free : Request_Range := ...;
  Queued : Natural := 0;
end Resource_Controller;
```

1

Request parameters in Ada with Requeue - 2-II

```
protected body Resource_Controller is
  entry Allocate( ... ) when Free > 0 and
    Queued /= Allocate'Count is
  begin
    if Amount < Free then
      Free := Free - Amount;
      Queued := 0;
    else
      Queued := Allocate'Count + 1;
      requeue Allocate;
    end if;
  end Allocate;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Request parameters in Ada with Requeue - 2-III

```
procedure Release (...) is
begin
  Free := Free + Amount;
  Queued := 0;
end Release;
end Resource_Controller;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

Class challenge:

- Who, now, gets the resource?

- Which assumptions are made this time?

9.26 Ada example with entry families

1

Request parameters in Ada with Entry Families - I

```
package Resource_Manager is
  Max_Resources : constant Integer := 100;
  type Resource_Range is new Integer range
  1..Max_Resources;
  subtype Instances_Of_Resource is
  Resource_Range range 1..50;

  procedure Allocate(Size : Instances_Of_Resource);
  procedure Free(Size : Instances_Of_Resource);
end Resource_Manager;
```

1

Request parameters in Ada with Entry Families - II

```
package body Resource_Manager is

  task Manager is
    entry Sign_In(Size : Instances_Of_Resource);
    entry Allocate(Instances_Of_Resource); -- family
    entry Free(Size : Instances_Of_Resource);
  end Manager;

  procedure Allocate(Size : Instances_Of_Resource) is
  begin
    Manager.Sign_In(Size); -- size is a parameter
    Manager.Allocate(Size); -- size is an index
  end Allocate;

  procedure Free(Size : Instances_Of_Resource) is
  begin
    Manager.Free(Size);
  end Free;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Request parameters in Ada with Entry Families - III

```
task body Manager is
  Pending : array(Instances_Of_Resource) of
    Natural := (others => 0);
  Resource_Free : Resource_Range := Max_Resources;
  Allocated : Boolean;
begin
  loop
    select -- wait for first request
      accept Sign_In(Size : Instances_Of_Resource) do
        Pending(Size) := Pending(Size) + 1;
      end Sign_In;
    or
      accept Free(Size : Instances_Of_Resource) do
        resource_free := resource_free + size;
      end Free;
    end select;
  end loop;
```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Request parameters in Ada with Entry Families - IV

```

loop -- main loop
  loop
    -- accept any pending sign-in/frees, do not wait
    select
      accept Sign_In(Size : Instances_Of_Resource) do
        Pending(Size) := Pending(Size) + 1;
      end Sign_In;
    or
      accept Free(Size : Instances_Of_Resource) do
        Resource_Free := Resource_Free + Size;
      end Free;
    else
      exit;
    end select;
  end loop;
end loop;

```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

1

Request parameters in Ada with Entry Families - IV

```

-- now service largest request
Allocated := False;
for Request in reverse Instances_Of_Resource loop
  if Pending(Request) > 0 and
  Resource_Free >= Request then
    accept Allocate(Request);
    Pending(Request) := Pending(Request) - 1;
    Resource_Free := Resource_Free - Request;
    Allocated := True;
    exit; --loop to accept new sign-ins
  end if;
end loop;
exit when not Allocated;
end Manager;
end Resource_Manager;

```



NTNU
Norwegian University of
Science and Technology

www.ntnu.no

Hendseth S., name

9.27 Java example: LIFO Scheduling

- Scheduler does FIFO or "arbitrary"
- This is reasonable to avoid starvation
- Other orders of requests? We can make the queue ourselves

1

Order of Requests in Java

```
synchronous allocate(){
    if(busy){
        queue.insertFirst(myThreadId);
        while(busy || queue.getFirst() != myThreadId) wait();
        queue.removeFirst();
    }
    busy = true;
    // notifyAll() ??? If more can be allocated at the same time.
}

synchronous free(){
    busy = false;
    notifyAll();
}
```



www.ntnu.no

Hendseth S., name

This one is a keeper - can implement any kind of orders and priorities by changing queue sorting.

- All RC problems can be transformed to server state.
 - We do not **have** to use the schedulers queue
 - Solution: Make our process queue the server state
 - ADA sometimes demands "double interaction" to get the same result.

9.28 Summary

Classroom challenge:

- So, What do you like the most? (Compare bounded buffers? Sverre: I like the ADA approach to composition, disallowing nested blocking)
- What would be the perfect synchronization mechanism? (I will propose an answer to this in the last lecture.)

10 Deadlocks

10.1 Learning Goals

Deadlocks

Understanding deadlocks, conditions for deadlocks, how they can be prevented, avoided and detected/handled.

10.2 Deadlocks

Necessary Conditions:

1. Mutual Exclusion
2. hold & Wait
3. No Preemption
4. Circular Wait

Deadlock **Prevention** (removing one of the conditions)

1. Optimistic concurrency control
2. allocate all resources at once.
 - efficiency?
 - starvation?
3. Preemption

4. Global allocation order
5. In addition: Global analysis - prove absence of deadlocks

Deadlock **Avoidance**:

- Resource allocation (Bankers algorithm) (Can block on free resources!)
(Assumes known allocation patterns!)
- Scheduling algorithms (Priority Ceiling)
 - efficiency?

Deadlock **detection & Recovery**:

- Sometimes it is impossible to avoid the chance of deadlock:
 - The user have scripting abilities (like in a large process control plant)
 - The operations are generated at runtime (travel agency)
 - The participants are dynamic (moving agents)
 - ...

Detection:

- Who owns and asks for what (detect circles)
- Timeout/Watchdog

Recovery:

- Breaking mutual exclusion (-> Inconsistent system)
- Preemption (Ex. -> Forward Error Recovery)
- Abort of thread or Atomic Action (-> backward E.R.)
- remember fairness

11 Modeling of concurrent programs.

11.1 Its learning call

The the lectures are almost over; Two-Three small subjects are left; This weeks lectures will be on formal modeling of concurrent systems, leading to both an understanding on how systems can be analyzed for deadlocks and race-conditions, and motivating thread interaction by synchronous communication.

Then after easter Amund Skavhaug will present a basic introduction to scheduling. (Remember that a predictable scheduler is a foundation for making real-time systems).

If there is some spare time anywhere I will spend it on looking a bit closer at deadlocks.

I really recomend attending these lectures:

- The subject matter is challenging: We are attacking thread interaction from the formal modelling angle, and we will be using a modelling language which is new to most of you and that you need to solve the exercise.
- Formal modelling is tough stuff, there is not much written curriculum on this matter; What you need on the exam is basicly defined by the exercise (!)

Check exam tasks 2 from last spring, and 4 from the continuation exam.

After easter Amund Skavhaug will be talking about Scheduling. I have also a very short part going a bit deeper on how to handle deadlocks.

Do not miss the opportunity to give also this week a good start!

11.1.1 Last years call

Out of all the lectures this spring these are the ones I would most strongly recommend you not to miss. These three hours represents, together with the guest lecture next Friday and the exercise, the "3rd part" of the course and there is not much written curriculum available. What we are after is the skills necessary to do the exercise.

However, the subject matter is challenging: We are attacking thread interaction from the formal modelling angle, and we will be using a modelling language which is new to most of you and that you need to solve the exercise.

The lecture tomorrow will be my last one. Next Friday will be a guest lecture. Unfortunately the last exercise will be scheduled in the week after easter. (but can be done on any PC, so feel free to work with it earlier :-))

See you!

11.2 Learning goals

Learning Goals, Modelling of concurrent programs

- An understanding of how using messagebased synchronization leads to a very different design than shared variable synchronization.

Ability to:

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

11.3 Motivation

(Ref. shared variable synchronization): Great, this is real-time programming.

Back to the beginning: we were making a RT kernel. We introduced preemption and published the Suspend/resume calls, and since then we have been struggling. What happened?

- We had RT demands on our system.
- Divided our system into one thread per RT demand (under preemption!)
- (Great, Modules)
- The threads were (by definition) cooperating -> Interaction was necessary.
- We chose spin locks -> suspend/resume -> Semaphores -> CCR -> Monitors...
- We got:
 - A system that was difficult to analyze/maintain
 - That had seldomly occurring and hard-to-find bugs
 - Bad composition/scalability (Neither threads nor monitors compose and deadlock and timing analysis is always global)

Have we chosen a good abstraction?

We have

- Questionable transformation from Timing demands -> Priorities
- Timing considerations always global -> does not scale
- Synchronization mechanisms are kernel mechanisms: more or less global entities -> global analysis necessary (like deadlock) -> does not scale.
- Monitors, threads and Atomic actions are ways of dividing a system into modules that do not compose well.

```
t1(){      t2(){
    i=i+1;    i=i-1;
}          }
```

What does this program **do**?

Class challenge: We have been talking about thread interaction like a fact of life, because of the threads cooperating rather than competing.

- Give example of possible interactions (as seen from the conceptual side.)

Is communication closer to what we need than synchronization?

B&W Chapter 6 is "Messagebased synchronization and communication". Is there a fundamental misunderstanding here? What is messagebased synchronization?

What does the word "priority" mean?

- Transforming timing demands into "priorities"
- We would like to transform back to prove schedulability...

How can we do better?

How can we find out how to do better?

Rather than discussing maintainability (this time), expressive power, abstractions, we will briefly look at formal modeling.

11.4 How to model a concurrent program

So, what to do: Start from completely new angle (as different from just exploring message-based systems.)

Sverres Hypothesis:

- Systems that are simple to model - yields simple models - will be simple to maintain.
- How model (wrt. deadlock, livelock, liveness, safety, progression, etc. - race conditions.)?
- How program such that models become simple. (Sverre hopes for maintainability)

(The program itself is, of course a model. Too complex though.)

Example: The classical deadlock:

```

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

```

Modelling as processes that participates in events.

```

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

```

```

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

```

(Oops, the semaphores are modeled as processes... What?)

Drawing the transition diagrams.

Terms:

- CSP, Communicating Sequential Processes: "Branch of mathematics" av Hoare,

- We can model a system and prove properties

FSP, Finite State Process: Subset of CSP LTSA, Labeled Transition System Analyzer: Java Applet that reads and analyses FSP-modeller.

FSP/CSP syntaks:

- **Processes** that participants in **events**
- events are global!
- prefixing "->": event -> Process

- rekursjon
- Choise "|"
- Parallele prosesser "||"

Yield here but leave it on the blackboard.

- Partial processes ", "
- indexes
- Parameters "P(N=5)": Just a constant
- if & guards (when)
- Labeling: a::SEM: Exchanges all event names (wait -> a.wait)
- Sharing: {a,b}::SEM : Both a.wait and b.wait becomes possible
- Relabeling "/" - change name of one event
-
- constants, sets
- properties: a partial model that the system is checked against
- Trace: En mulig sekvens av hendelser
- Transisjonsdiagrammer
- Alfabet: De hendelsene en prosess kan delta i. (kan manipuleres med @ og \)

Terms:

- Deadlock: A state we cannot leave (STOP)
- Livelock: A subsett of states we cannot leave
- progression: the absense of livelocks
- liveness: what should happen happens sooner or later.
- safety: Something bad never happens

- Model checking: does the model of the system correspond to the model of the specification.

New example: The bounded buffer (view the slide) Class challenge: Make a model of the bounded buffer.

```
SEM(N=1,MAXN=1) = SEM[N],
SEM[n:0..MAXN]
    = (when (n<MAXN) signal->SEM[n+1]
|when (n>0) wait->SEM[n-1]
).

PUTTER = (nFree.wait -> putter.mutex.wait ->
    putter.mutex.signal -> nInBuffer.signal -> PUTTER).
GETTER = (nInBuffer.wait -> getter.mutex.wait ->
    getter.mutex.signal -> nFree.signal -> GETTER).

||SYSTEM(S=3) = (PUTTER || GETTER || nFree:SEM(S,S) ||
    nInBuffer:SEM(0,S) ||{getter,putter}::mutex:SEM(1,1)).
```

11.5 The modelchecking example:

```
SEM(N=1,MAXN=1) = SEM[N],
SEM[n:0..MAXN]
    = (when (n<MAXN) signal->SEM[n+1]
|when (n>0) wait->SEM[n-1]
).

PUTTER = (nFree.wait -> putter.mutex.wait -> put ->
    putter.mutex.signal -> nInBuffer.signal -> PUTTER).
GETTER = (nInBuffer.wait -> getter.mutex.wait -> get ->
    getter.mutex.signal -> nFree.signal -> GETTER).

||SYSTEM(S=3) = (PUTTER || GETTER || nFree:SEM(S,S) ||
    nInBuffer:SEM(0,S) ||{getter,putter}::mutex:SEM(1,1)).

property
    BUF(N=5) = BUF[0],
    BUF[n:0..N] = (when n < N put -> BUF[n+1]
```



```
| when n > 0 get -> BUF[n-1]).

||SAFETY = (BUF(4) ||SYSTEM(3)).
```

The number of states

Bufferlength	#States
2	23
3	38
4	53
5	68

Conceptual model of a bounded buffer:

```
BUF(N=5) = BUF[0],
BUF[n:0..N] = (when n < N put -> BUF[n+1]
  | when n > 0 get -> BUF[n-1]).
```

The number of states: N+1

Demonstrate modelchecking.

Class challenge:

- Why is there too many states in the semaphore solution?
 - Something wrong with how to model?
 - The modelling abstraction level?
 - The way the bounded buffer was implemented...
- How should the BB be implemented to yield N+1 states?
- Which language primitives would we use?

OCCAM story, occams razor, OCCAM language made to make 1-to-1 models. Solves multithread programming. (like Go - if not using shared variables)

- So: How do we program so that the models get simple? (We wish for **events** that more threads participates in)
 - Ada entries (synch tow-way comm++)

- Synchronous communication !
- Barriers (as the end boundary of an Atomic Action)

A new way of defining a process follows:

- Process: Participates in events
- event: synchron communication
- Composition: More processes constitutes a process
 - Internal states disappear
 - Internal communication disappears
 - \Rightarrow Very nice scalability
- The primitive processes are tiny
 - \Rightarrow non preemptive scheduling becomes ok (and by non-preemptive scheduling a semaphore is just a flag)
 - \Rightarrow small overhead and few demands on scheduler.

Solves the problems introduced by preemption. NB: Does not solve problems like deadlocks, but they do get far easier to spot since we have fewer states.

Occam-like implementation

```
process
boundedBuffer(channel put,get){
  while(1)
    select{
      n<N: c = read put; store c; n++;
      n>0: read get; retrieve c; reply with c; n--;
    }
  }
}
```

How many states? $N+1$

The $i=i+1$ -example, messagebased:

```

server(){
  i = 0;
  while(1){
    select()
    read ch1: i = i+1;
    read ch2: i = i-1;
  }
}

```

No doubt that rest of the program cannot make this inconsistent: If t1 sends 10000 messages on ch1 and t2 sends 10000 messages on ch2 i will be 0 at the end.

NB: This module can be maintained!

How could we implement synch comm with semaphores?

```

T1(){
  c = 10;
  signal(T1Ready)
  wait(t2Ready)
}

T1(){
  wait(t1Ready)
  var = c;
  signal(T2Ready)
}

```

OCCAM SHORTHAND

- ? reads
- ! writes
- The semaphores and the variable: A channel

Now "process diagrams" with process circles and channel arrows. Great modularity!. Ref. Øyvind Teigs relation to deadlocks.

The challenge for you:

- Step 1: Think about building program with server threads and messages
 - "While(true) select()..."
 - Use lots of such processes.

- processes is a better way than objects for dividing the system into modules.
- Step 2: Skip buffered communication (Block on write!)
 - Buffers should be used when you need them, conceptually.
 - In other situation you just consume ram to store old data rather than doing useful work. Creating slanted arrows in the sequence diagrams.
- Step 3: Choose fine-granular processes (& preemptive scheduling)

But unfortunately:

- Performance, closeness to hw/kernel
 - publishing suspend/resume calls was tempting - for a reason!
- Need some messagepassing infrastructure - library or similar. (and these make most often buffered communication - "send and forget" is assumed simpler to understand???)
- Real-time: No schedulability proofs exist.

12 Scheduling

12.1 Learning Goals

The student should:

- 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.

12.2 Its learning call

This week Amund Skavhaug will cover "Scheduling". Having more threads with responsibilities for their own timing demands will work only if the threads gets to run when they need to, and knowing how the scheduler works is necessary both for the programmer designing the system and for the final argumentation that the system works - the "schedulability proof".

Amund will be using the lecture slots both Monday and Tuesday.

Do not miss the opportunity to give also this week a good start!

13 Guest lecture Teig and course ending.

An old colleague of mine, Øyvind Teig from Autronica, will be lecturing tomorrow on his experiences from a long career as a embedded/industrial systems software developer. Øyvind has always been an advocate of the message passing style of multithread programming, derived from the theory of communicating sequential processes that we talked about before Easter.

Tuesday will be the course ending. No special agenda; you can propose topics to revisit and/or ask questions.

I hope you all can make it to the last week of lectures in the course. See you!

13.1 Øyvinds slides.

<http://www.teigfam.net/oyvind/pub/pub.html>