NTNU
Norwegian University of
Science and Technology

Assorted slides used in ttk4145 lectures

Sverre Hendseth
Department of Engineering Cybernetics

Course Learning Goals, TTK4145, 2016

- General maturation in software engineering/computer programming.
- Ability to use (correctly) and evaluate mechanisms for shared variable synchronization.
- Understanding how a deterministic scheduler lays the foundation for making real-time systems.
- Insight into principles, patterns and techniques for error handling, consistency and fault tolerance 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.

Quality System

- Check https://innsida.ntnu.no/wiki/-/wiki/Norsk/Kvalitetssikring+av+utdanning
- I hope you give me constructive and immediate feeldback! Other feedback goes to the quality system in the previous point.
- We are having a questionaire to be answered by everybody at the end of the semester.
- A colleague will be watching one of the lectures to share experiences.
- We are having a reference group (3+ meetings, pizza at the last one) (See next slide)
- "Course reports" will be made, and the department follows up on improvements. Old reports are available for anybody.

Reference Group

...Someone to represent you, and for me to consult...

- One from Kyb 5year
- One from Kyb 2year
- One not from kyb (elsys, idi, marin, maskin,... ?)
- One international
- One woman
- More categories? Do you want to contribute? Who do you want to represent you?

Three meetings (one of you writes the minutes...) Pizza on exam day :-)

Mandatory activities & Grading

- Exercises are mandatory, to be approved on the lab by the student assistants.
- A project (control 3 cooperating elevators) evaluated in 3 parts (Design, code quality, funcionality). Counts for 25% of your grade.
- An exam at the end of the semester counting 75%.

Lab Groups

Groups of 3; We decide the groups.

— Pedagogical gain from student cooperation is clear (or at least commonly accepted at the time by both Ntnu and students).

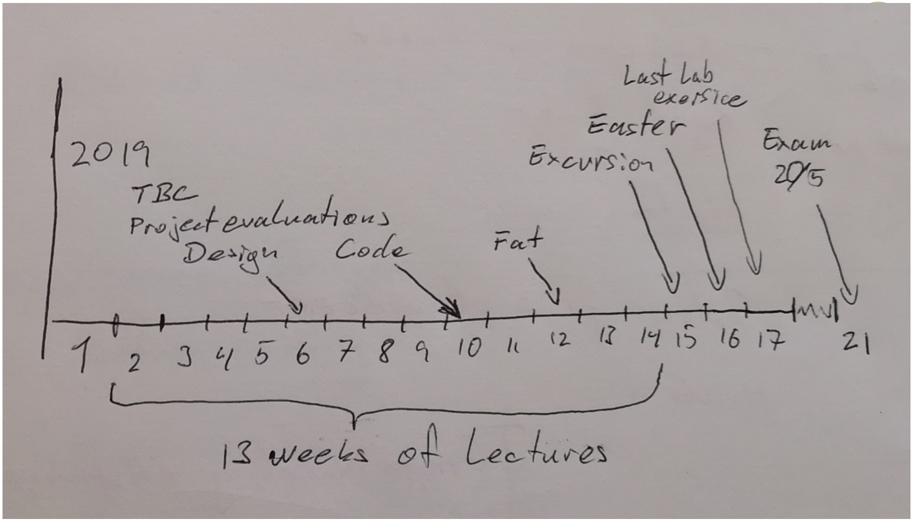
— "Random groups" are fair, more challenging and you have more to learn from people you do not know well already.

— A "low ambition" is harder to negociate in a random group. (Similarily; "I am best at this so I should make decitions" is not so viable).

— While work possibly will be more tense, it will not take more time. Maybe even the explicit ambition discussion will reduce work?

— How can we further stimulate you learning from each other?

The Semester



Topics per week

Week	Topic
2	Course intro and Project startup
3	Project; Languages, Design, Code Quality, Network, Fault Tolerance Intro.
4	Code Quality, Ada
5	Fault Tolerance Basics
6	Transactions
7	Atomic Actions
8	Asynchronous transfer of control, Exceptions?
9	Shared Variable Synchronization (Semaphores)
10	History and Introduction
11	Monitors, Ada and Java
12	Scheduling
13	Formal methods and messagebased systems
14	Guest Lecture

Learning Resources

... there are more than you can appreciate ...

- Lecture videos from 2016
- This years lectures
- Lecture notes 2017, lecture notes 2019
- recommended reading (pdfs) on blackboard
- Old exams (Skip 2018?) (I do not intend to make major changes)
- Learning Goals
- https://www.wikipendium.no/TTK4145_Real_Time_Programming
- http://www-ist.massey.ac.nz/csnotes/355/lectures/monitors.pdf
- ...
- Inform me if you find something recommendable!
- The internet

You need to make choices (see next slide)...

3 safe ways?

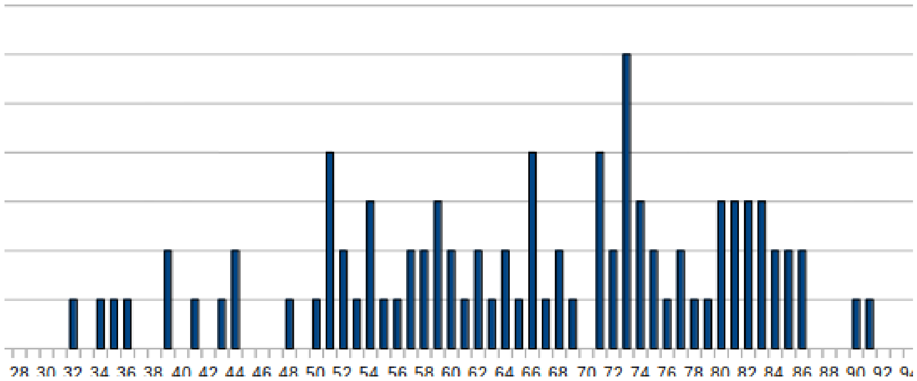
- Learning-goal based: Use the learning goals and old exams to get a feel for what you need to master.
- Video-based. The videos from 2016 is a complete (until further notice) walkthrough of the learning goals.
- Lecture-based. The lectures of 2019 will not be a complete walkthrough, but will create awareness enough for you to see what you need to master. Then ask!

See the ttk4145_sporreundersokelse.pdf document on blackboard.
But be aware: Some of the learning goals are quite deep, going even beyond knowledge and skills. This requires ... maturation.

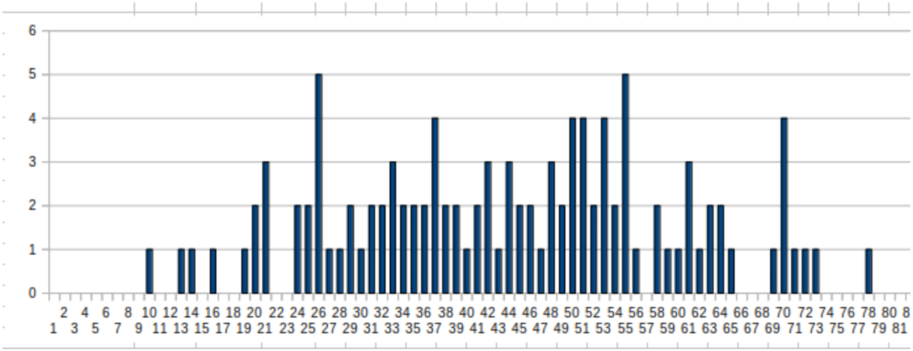
Task: Recognize numbers in stream of characters

- You read a text file character by character (char getNextChar()) and are to pick out and print all the numbers.
- Legal numbers:
  1,2,3,-234,1.2,.05,-0.43,1e-12,-199.23E98,.4e+98,...
- How would you structure this small program?

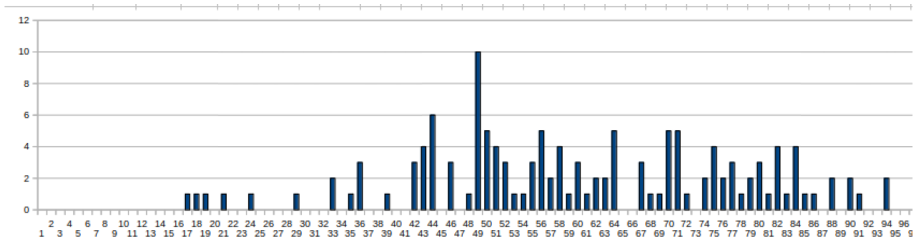
Exam results 2013



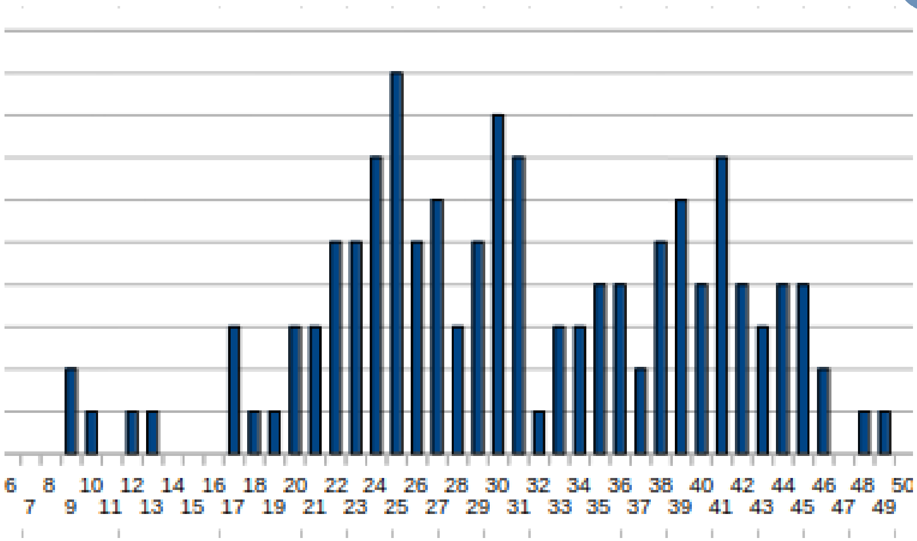
Exam results 2014



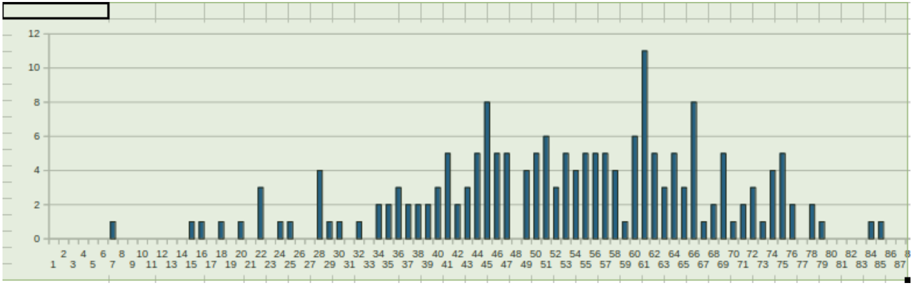
Exam results 2015



Exam results 2016



Exam results 2017



Study skills: How to learn Abstractions and Thought patterns, including SW Design

- Learn-by-example. Be with role models.
- When the number of examples reaches a treshold, you will abstract.
- Creating awareness, Setting focus
- Passively notice examples.
- Actively notice examples.
- Interview those that can solve problems you can not.
- Use teachers, Explain what you do not understand, Ask questions.

Is there a market for a walkthrough of these?

Recordning this years lectures

— Anybody against filming?

— (Some lectures *will* be filmed - those potentially useful for me next year.)

But lectures this spring will not have any ambition of being a complete walkthrough...

— ...and filming is not longer supported by Ntnu... Meaning quality/success rate will be limited.

— Would you still appreciate availability of such (sparse?) "for the record" videos?

Fault tolerance teaser

If you have a bug in your SW or if a "cosmic ray" at any time flips any bit in your computers memory; Is it still possible to make SW that behaves according to spec?

The embedded systems setting

On the desktop:

- — Most user processes are competing — independent of each other.
- — Some sense of fairness, responsiveness or performance decides who runs when.
- — If one fails it is a local problem — the rest should just continue.

In the embedded system:

- — All processes are cooperating — interacting.
- — Who runs when should be decided strictly - the most important, or under some regime for satisfying deadlines.
- — If one fails it is a system failure.
- — *Recovery* from an error often requires cooperation/interaction!

Redundancy

Static Redundancy

- Ex. Storing multiple copies of data
- Ex. Resending UDP messages whether they are necessary or not.
- Ex. Having more HW units working in paralell
- Does not work well with SW — If one copy makes an error, the other will also.

Dynamic Redundancy

- If at first you don’t succeed, try, try, try again
- ...possibly in another way
- Failures must be detected
- Ex. Resending a message if we time out waiting for acknowledgement.
- Ex. Restart/Reboot and try the same again (Works embarrassingly often)

Detection: Acceptance Tests

Do not test for the presence of errors.

Do make tests to check that everything is ok.

A small, but cool, observation:

— **Q:** How can we test that the systems works in presence of unknown errors?

— **A:** Inject failed acceptance tests.

Error Modes

"Error Modes of X" == "The number of ways the X can fail".

- Error modes is a part of X' interface!
- When designing X, making the error modes few/simple is an important task!
- "I failed" is a good one. "Message was not received".

Errormodes of communication frameworks

TCP vs. UDP vs. Broadcast vs. Message queues vs. odd communication libraries and transactional frameworks vs. Erlang vs...

Sverres view on this: It amounts approximately to the same complexity. Choose what suits you. Your challenges must be solved anyway:

- A process may be restarted at (theoretically) any time
- One node may loose network
- The system should never, even theoretically, loose orders.

Find out how to solve these; Then choose communication framework. ... may be a good advice.

Merging of error modes

- When you anyway have to handle the worstcase, why not handle all errors in that manner?
- If you detect an error: just restart the program - in spite of the fact that this error might have a less-drastic recovery?
- Were you really interested in *why* reading the configuration file did not result in a consistent configuration?

How to restart?

Some program state is probably necessary to keep over a restart.

1. **Checkpoint restart**. Save the state "occationally". Read the newest old state at startup.
2. **Process pairs**. The new instance of the process has already started, and just receives the checkpoints, ready to take over.
3. **(Running under a transaction-aware OS)**

When is occationally?

— Do all the work, up to and including the acceptance test (but do not do the 'side effects', like giving the reply, lighting the elevator lamp, etc).

— Store a checkpoint.

— Do the side effects.

A newly restarted process will begin by doing (repeating?) the side effects.

The Log

The monolothic checkpoint can be replaced with a sequence of state updates ("Log records").

- The checkpoint can be reconstructed by executing all log records
- If "before-information" is also stored in the log, the ability to run the log backwards enables an elegant UNDO feature that may be useful in error handling.

Atomic stuff and Transactions

Atomic stuff is wonderful! Programs will be kept simple (maintainable) since an Atomic Action has no, observable, intermediary state.

- Semaphore operations
- Synchonous communication
- Assembly instructions

We can design such Atomic Actions, typically by locking resouces (so that nobody see the internal state) and syncing the acceptancetest/unlocking between participating threads.
The error modes of such an Atomic Action is a challenge; If we simplify the error modes into "ABORT", then we have a **Transaction**.

Threads and Processes

All threads/processes in embedded systems are cooperating. Interaction is typically categorized into

— Synchronization

— Communication

What is best?

— From a SW design perspective: Communication

— From a Real-time programming design perspective: Synchronization is the only option

— (...but be aware. The elevator is not demanding on the RT side. Synchronization may still be viable.)

Synchronization vs. Communication

If communication is *conceptually* what you need, then implementing it with synchronization will be more complicated.

And communication most often *is* conceptually what you need...

Elevator project; Choosing threads

Communication based:

— One thread per module. All have while-select structure (see next slide)

Synchronization based:

— NOfThreads == Number of concurrent blocking operations + 1 (=3?)

— But if you teach yourself nonblocking versions of select++ you can manage "without threads".

Communication-oriented design

— Think in terms of responsibilities.

— ... and servers yielding services to each other.

All processes have the same structure:

```
func Elevator(ElevStateCh chan  ElevState,...) {
  ...
  for {
    select {
      case temp := <- ElevStateCh:
        ...
      case <-time.After(2*time.Millisecond):
        ...
      ...
    }
  }
}
```

Advice and pitfalls

- Think: Detect what needs handling, then handle it. Preserve the causal relationship.
- (Do not decouple effect from cause by global state. A "what needs to be done" analysis should be concerned only with real things; messages, HW, timeouts, not with state updates.)
- (If nothing needs doing: Do nothing!)
- Do not nest while loops (Also breaks causality).

The underlying rule is; When the system breaks, tracking down the bug in the code should be possible (And the causal relationships are wonderful for this.)

Languages in TTK4145

Three supported languages on the lab:

- **Python:** Python is seen to be lowest-treshold. (not so popular though)
- **C (/C++):** /The/ embedded systems language. Programming C is always useful experience. (But not simplest, and not forgiving)
- **Go:** Go is a supported language because it supports the messagepassing regime. (popular)
- (Sverre: ...but if you make a synchronization/"global state" based design anyway and want to avoid C: You might as well use python.)

Other popular languages: Erlang (also well supported). Let us see more Rust? Ada? (Elixir, Haskell?,...)

Two more languages relevant in lectures:

- **Ada** - have interesting mechanisms for synchronization and "asynch transfer of control".
- **Java** - have interesting mechanisms for synchronization.

Learning goals; Code Quality

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

Brainstorm: What is Code Quality?

What is the difference between high and low quality code? Any metrics, techniques, principles, signs, rules, monitors, etc.?

Evaluating the project

Use ~10 min to find the entry point of the code and answer the following questions:		
1	Does the entry point document what components/modules the system consists of? • You can see what threads/classes are initialized	(0-1)
2	Is the connection between modules clear from looking at the entry point? • You can see how different modules interact • You can (with relative ease) find out how modules depend on each other • Circular dependencies are avoided • If there are any global variables, their use is clear and are their names are excellent	(0-3)
Use ~15 min to answer the following questions about modules:		
3	Do the modules minimize accessibility to their internals? • Accessibility is enforced programmatically • That is, programming features (or conventions) for "private" and "public" are used • (Go: Lower case fn names, Erlang: `-export([])`, C: fn declarations in .h-files, etc.)	(0-1)

Code Quality Projection 1: Can you read it?

If I can look at a page of your code and understand what it does, it is good! (C)

If I can look at a page of your code and see that it is correct, it is perfect! (A+)

Example: Casette Player

```
bool play;
bool record;
bool pause;
bool ff;
bool rw;

play(){
  if(play == false && kasett()){
    if(ff || rw){
      mekanikk_stop();
      ff = false;
      rw = false;
    }
    mekanikk_play();
    motor_play();
    if(record == false){
      elektronikk_play();
    }
    play = true;
  }
}
```

Example: Casette Player

```
static enum {State_Stop,State_Play,...} g_state;

play(){
  switch (g_state) {
    case State_Open: break;
    case State_Stop: action_play(); g_state = State_Play; break;
    case State_Pause: action_play(); break;
    case State_Rewind:
           action_stop(); action_play(); g_state = State_Play; break;
    case State_FastForward:
           action_stop(); action_play(); g_state = State_Play; break;
    case State_Play: break;
    case State_RecordReady: action_record(); g_state = State_Record; break;
    case State_Record: break;
    case State_RecordPause: action_record(); g_state = State_Record; break;
  }
}
```

Example: Bounded buffer with semaphores

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

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

Example: A Hard to find bug

What do these two functions do? Do they work?

```
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);
    }
  }
}
```

```
#include "config.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#include <math.h>
#include <pthread.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <libheis/elev.h>

#include "comms.h"
#include "messages.h"
#include "operator.h"
#include "orderlist.h"
#include "target.h"
```

```
#define N_ORDERS 3

bool orders[N_ORDERS][N_FLOORS];
void command_signal_callback(int floor);
int fdout, fdin;


int main(){
  run_system();
}
```

Comments to explain complex code

If your code is hard to understand:

1. Fix it: Give functions and variables better names so we can see what happens.
2. If it is still hard: The code is bad; Throw it out and rewrite.
3. If that was not viable: There is something wrong with the context the code exists in: Fix the datastructures/design/architecture. (It is worth it!!!)
4. Well, *maybe* the problem you were solving was of a kind that had no readable solutions. Accept this very reluctanly and sorrowfully (like "Hell is real") because it means that if the problem was slightly harder it would be beyond you to solve.

Code Quality Projection 2: Maintainability

If a system is maintainable you can:

- easily find and fix bugs without adding new ones.
- easily add features without compromising existing design/structures.

A metric: Expected effort to fix the next bug.

A meaningful module

A meaningful module:

- You should be able to maintain the module without knowing anything about its users or usage patterns.
- You should be able to use the module without knowing details of its internals.
- Ideally: The principle you used for dividing the system into modules should allow **composition**: That supermodules can be made from modules.

Strong cohesion: The module has a purpose or responsibility that its parts contribute to. **Weak Coupling:** The dependencies between modules are simple, easy to understand.

Module Interfaces

Module Interfaces

- shall be as ’thin’ as possible; Avoid duplicating functionality and convenience functions
- shall not contain assumptions on usage patterns
- must be complete
- (The hidden complexity of the implementation shall not be insignificant)

What is the problem here?

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

Epilogue: Some questions for the class

- Why should limiting scope of variables to a minimum be good?
- Why is ’goto’ bad?
- ...and btw. what do we think about exceptions?
- Why are preserving causal relationships good?
- Why aim for a minimum state (not have variables for floor, nextfloor, prevfloor, moving and direction)?
- Why is building/choosing a good abstraction important?
- Is C++ a good programming language?

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.

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

Welcome to the dark side

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;
}
```

Terms and techniques

- Fault Tolerance (The term)
- Error modes
- Static and Dynamic redundancy
- Error detection
- Forward and Backward Error Recovery and the domino effect.
- Acceptance Tests
- Merging of error modes
- Writing checkpoints, checkpoint-restart

Sverres Design Process

1. Brainstorm for Use Cases: Span functionality space; do not aim for "completeness".
2. Make design decisions: Divide into modules (++). (This is not a systematic process)
3. Map the Use Cases from 1 on the design. This is both a cosistency check and it yields the sub-use-cases for the modules.
4. Draw module interaction diagram. Who calls who?
5. For each module:
   - Sum up use-cases from 3.
   - Either: Design the perfect module interface that satisfies the use-cases - or recurse from 2.
6. Move responsibilities between modules (reorganize how the system is divided into modules if necessary) so that the diagram in 4. gets fewer arrows and that the module interfaces becomes perfect abstractions.

Error Dection: The learn-by-heart list

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

Note that enabling some of these may require extra code/insfrastructure.

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.

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;
  }
}
```

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 mttsf 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() + mttsf*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())*mttsf;
    }
  }
}
```

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;
}
```

reliable_read

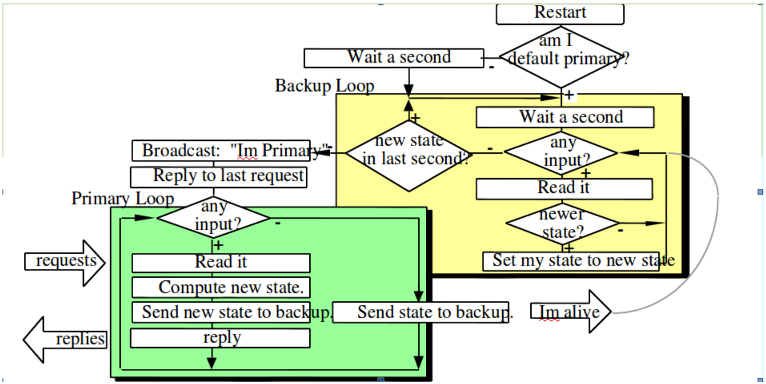
```
bool reliable_read(group,addr,&value){
  bool status, gotone = False, bad = False;
  Value next;

  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;
```

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);
    }
  }
}
```

Process Pairs



process pairs pseudocode

```
repeat
  // Backup mode
  Read Checkpoint and 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
      // The Assumption is that there will be more of these.
      send last Checkpoint & IAmAlive.
```

Project Design Evaluation

Show documents

- designevaluation.pdf (by Sverre)
- design_scoresheet.pdf (used v2018)

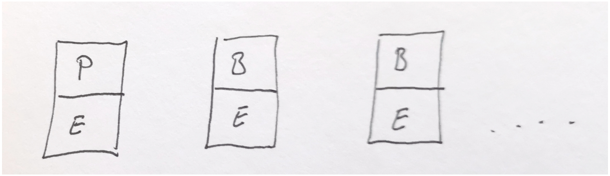
Big Scenarios/Use Cases

- Button
- Floor
- A node stops/crashes/disappears
- A node starts
- A node looses network
- A node gets network

Division into ’modules’ - ’Responsibilities’

(Step II: Gut feeling decitions!)
Responsibilities:

— Local elevator and HW
— Decitionmaking (in Primary)
— System Configuration



Decitions:

— Very thin Elevator; All decitions made by Primary

Scennario: Node Starts

A node starts:

- — Listens for 5 secs for a Primary broadcast
- — Got one: I am backup: continue listening for Primary broadcasts and updating state.
- — If not: I am primary: Start broadcasting Primary broadcasts. Start listening for new order messages.
- — Start elevator-control handling orders and setting lamps and checking buttons and floor sensor.

Preliminary Decitions (to be checked/confirmed with other scenarios):

- — The Primary broadcasts state at updates and regularly

A button is pressed

A button is pressed:

- — Register UnconfirmedOrder; Send unconfirmed-orders to Primary.
- — On Primary; Make decition, update state, send Primary broadcast.
- — All Elevators: Set relevant new lamps.
- — Originating Elevator: Check lamps to remove orders from UnconfirmedOrder set.

Decitions:

- — The Unconfirmed-orders are sent, at need and repeatedly, even when empty, to Primary (UDP)
- — UnconfirmedOrder module is roughly ready: Add(order), Remove(lamps) and Tick().

Terms and techniques, used

- Fault Tolerance (The term)
- Error modes (TCP vs UDP when loosing network)
- Static redundancy (UDP and broadcast resending, storing state in all nodes.)
- Dynamic redundancy (Take-over of orders when an elevator crashes)
- Error detection (Timeouts, Reentering too old orders)
- Backward Error Recovery (Restart goes back to safe state)
- Forward Error Recovery (Redistribution of orders)
- Acceptance Tests
- Merging of error modes (Any error leads to restart, timeout is caused by more reasons)
- Writing checkpoints, checkpoint-restart (Getting state from Primary at startup)

What now?

Learning Goals: Atomic Actions

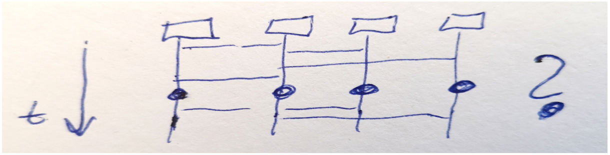
- A thorough understanding of the problems Atomic Actions are meant to solve and how these motivates the different aspects of Atomic Actions.
- Ability to use and implement Atomic Actions, including the mechanisms providing the start, side and end boundaries.
- Understanding the motivation for using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

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.

Syncing recovery points: The problem

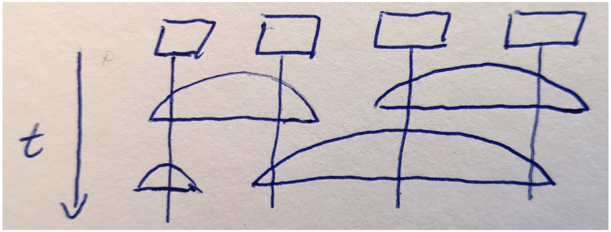
So; We are to sync recovery points to avoid the domino effect.



- Recovery must still be written after acceptance tests - when we know everything is in order.
- A recovery point represents the consistent state *at the start* of an operation, while the acceptance tests are run at the end...
- *Is* there a time where all actors, for all operations reaches a sync point?
- *Must* all actors sync at the same time?
- What if one actor can participate in more operations at the same time?
- If more operations are active concurrently, one part of the state may be relevant to one operation and another part to the other.

The Solution: Syncing Recovery Points

Basic Design Infrastructure; Three ’Boundaries’:



— Start Boundary: When an actor starts participation in an operation: (Register membership. If preparing for backwards error recovery; store recoverypoint of relevant state subset?)

— Side Boundary: No interaction with actors not participants of the operation. (No communication, ’Lock’ any relevant state subset to operation.)

— End Boundary: Sync acceptance tests, unlock state.

By this we achieve error *containment.* (No error recovery yet)

How to implement the AA infrastructure?

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/'supporting protocols'? (same, just better structured: like in Gray)
3. A number of procedures that is called by participating threads? (the B&W book)
    - Action == Class or Package/module?
    - Thread interaction through synchronization and shared variables.
4. Threads that is created and terminated with the AA?
    - select-then-abort in Ada
    - Asynch. exceptions in Java
    - pthread_cancel in POSIX
5. ...

Yes to all! This is a *design framework*!

Show Burns & Wellings slides on AA

Ada: Relevant slides from B&W: Chap. 7: 10-16
Java: Chapter 10 (ThirdEdition of the books slides) Slide 20-26
Comments on examples:

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

Sidestep: Thread synchronization in Java

Java Monitors:

— **Synchronized** methods:
  - Only one thread can call one of those in an object at the same time. (need object lock) (Btw. 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).

Sidestep: Thread synchronization in Ada

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.

Term: The ’Guard’:

— A test integrated in the language that opens or closes for execution

— In our context: blocking, suspending.

— Is not (necessarily,only) evaluated sequentially

Four discussion points

1. Locking (one form of non-interaction) - Two phases: Growing and Shrinking
   - (The other form of non-interaction is not sending messages :-))
2. How to sync at the end: The (two-phase) ’commit protocol’
   - (Well, a ’barrier’ would do it, if no error handling)
3. The ’Log’ and ’Checkpoints’ — An intelligent replacement for the recoverypoint.
4. How to distribrute error information.
   - Via the commit protocol.
   - Some polling regime (shared variables or messages).
   - *Asynchrouneous transfer of control*

Discussion point 1: Ensuring no-interaction by ’Locking’

Locking:

— This resource can only be accessed as part of this operation.

Purpose:

— Containment/Isolation: To ensure that an error does not have consequences outside the operation.

— An operation should, from the outside look like it is *atomic*.

Minimum Requirement:

— All locks need not be reserved at the beginning. But none must be unlocked until all have been locked.

— Two phases: Growing and shrinking.

Can deadlocks happen?

— Yes

— But notice: The infrastructure keeps track of memory used and prepares for error recovery: We can handle crashed threads and preempt locked resources.

Discussion point 2: Syncing acceptance tests

Making the end boundary

- — A barrier is the basic sync - but no information is exchanged.
- — The standard: "Two-phase commit protocol":
  - prepareToCommit $\longrightarrow$ OK/Fail $\longrightarrow$ Commit/Abort
  - Enables pass/fail decision.
  - Enough to enable backward error recovery.
- — Variations:
  - presumed abort (early abort, do not wait until vote) (Using e.g. ATC)
  - one-phase (only one participant - no use action manager counting votes)
  - read-only (commit/abort irrelevant)
  - Last Resource Commit: *One* participant/role gets to wait until after vote to make his decition: He can be responsible for world interaction?
  - (Irrelevant for us: Synchronizations (4-fase commit, some cache optimization))

Discussion point 3a: Replacing the recovery points

The Log

We have already reduced recovery points to be per-operation. We can reduce further to be per-resource and per-operation:

— A *Log Record* is written every time a variable is accessed.

— "As part of operation TId I change variable a from 2 to 3"

This enables:

— REDO: After a restart the complete log can be ’played’ to reconstruct the state before the restart.

- We need only do the log records of committed operations.
- At the end there may be undecided operations: These will be aborted since we restarted: Skip them to achieve a consistent state.

— UNDO: If an operation is aborted, the log can be played backwards undoing all effects of this operation.

Discussion point 3b: Replacing the recovery points

Checkpoints

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 is written to the log. This is the *Checkpoint*.

- 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.

Discussion point 4: How to distribrute error information?

- All well and good for *Transactions* (that merges error modes into ABORT and uses backwards error recovery).
- But if we want/need forward error recovery (e.g. 'ABORT' unacceptable, timing issues, side effects that can not be undone):
- We need more information to enable cooperation on the error handling.
  - We could extend the OK/Fail vote with 'errno'.
    - Worry: We must extend the action controller with domain knowledge...
    - Worry: Conceptually this is unnecessarily late.
  - We could poll on shared error variables or incomming error messages.
    - Worry: Are we able to poll after an unknown error?
    - Smaller Worry: Conceptually this is still unnecessarily late.
  - We could *signal* — interrupt the other threads whose work now is meaningless.
    - Conceptually nice.
    - Worry: Technically nontrivial.
    - This is 'ATC': Asynchroneous Transfer of Control.

Four implementational patterns

— The ’Transaction Manager’ or ’Action Controller’.

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

— The actor: ’Resource Manager’, thread, or ’action member’.

- offers *rm_doWork*(*tid*, ...) functionality which calls *tm_joinTransaction*(*me*) if not already a member
- participates in two phase commit protocol (prepareToCommit(), commit(), abort())
- Note: prepareToCommit(tid), commit(tid) and abort(tid) can be given reuseable forms - works only on the RMs data structures!

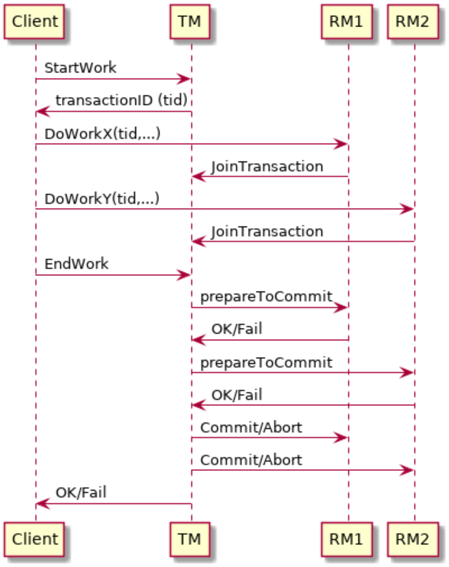
— The ’Lock Manager’.

- keeps track of locks associated with each transaction
- 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.

— The ’Log Manager’.

- Can queue more logrecords and optimize disk access.
- And if failing independently: Received log is safe log.

A Transaction



Some recovery cases

- **What happens at deadlock?**
  - Too old transactions are aborted. No Problem.
- **A RM is restarted:**
  - All transactions active when we restarted are aborted.
  - Transactions where we have voted but not gotten answer; We must ask TM.
- **A 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.

The i=i+1/i=0-1 problem

```
//Globals:
int i=0

//Threads:
t1(){                              t2(){
  for(int j=0; j<=100000; j++){      for(int j=0; j<=100000; j++){
    i=i+1;                             i=i-1;
  }                                  }
}                                  }
```

What values may *i* have after both threads have finished?

- 0?
- All numbers from -100000 to 100000?
- Any number, even larger than 100000?

All resources are shared!

— Memory, certainly.

— ’Hidden’ memory used by libraries (...your own modules and the kernel). If the library takes care of this itself, it is called ’reentrant’.

— Sensors and actuators

— ’CPU’ — computing capacity (*This* is real-time programming; We solve it by *Scheduling*.)

— ... any other interface...

The suspend/resume problem

```
//Globals:
bool g_initDone = False;

//Threads:
t1(){                                t2(){
  // Do initialization                 if(g_initDone == False){
  g_initDone = True;                     Suspend();
  resume(t2)                           }
  // Continue executing                // Continue executing
}                                    }
```

It is very, very difficult to write programs with Suspend and Resume.

— What is the problem?

— There is a general warning about if statements!

— ’Named’ suspend and resume may become "events" or "conditions". (They have the same problems, to start with...)

Solving mutual exclusion (The i=i+1 problem) with busy locks and flags

Class challenge (paper and pen):

Assuming no sychronization support by the OS; (How) Can the i=i+1/i=i-1 problem be solved?

Your arsenal:

- — You can waste CPU in loops.
- — You can assume ’flag’ (boolean) operations atomic.

Solution: Mutual exclusion with busy locks and flags

```
// Shared:
int g_i=0;
bool g_flag1 = False, g_flag2 = False;
int g_turn;

void t1(){
  g_flag1 = True;
  g_turn = 2;
  while(g_flag2 == True && g_turn == 2){}
    // We may run
    i=i+1;
  g_flag1 = False;
```

Looking more closely at the arsenal

- Test&Set (swap) assembly instruction (atomic, but not obvious)
- Spin locks (Wasting cpu)
- Disable interrupt (steals control from OS/scheduler)

Which features, exactly, are we looking for?

Yeah, mutual exclusion...

— 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)

— Condision Synchronization — Blocking on event or status. (Guards etc.)

— Resource allocation (More than mutual exclusion! Ref. the "lock manager")

— rendezvouz/barriere — synchronization point (Ref. AA ’end boundary’)

— Communication

— Broadcast

— Lock manager (reserve A, B or both)

Enter the Semaphores! (Dijkstra,1962)

A counting semaphore:

— *signal*(*SEM*) increments the counter.

— *wait*(*SEM*) decrements the counter — will block if 0; the semaphores value can not be negative.

Variations:

— The binary semaphore can have values 0 and 1.

— Ownership: The association to the thread that waited is maintained.

— A *GetValue*() function.

We solve beautifully:

— Mutual Exclusion.

— Solves Condisional Synchronization. (ref. Suspend/Resume)

— Solves basic resource allocation.

Classroom task; Make a bounded buffer!

A bounded buffer:

- Has space for N elements.
- $put(Element\ e)$ blocks if the buffer is full.
- $get(Element * e)$ blocks if the buffer is empty.

Hint: How many different reasons for blocking are there?

Bounded buffer with semaphores

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

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

The classical Deadlock!

```
t1(e){
  wait(A);
  wait(B);
    ...
  signal(B);
  signal(A);
}
```

```
t2(e){
  wait(B);
  wait(A);
    ...
  signal(A);
  signal(B);
}
```

Classroom task; Lock A, B or both.

Try, with semaphores to protect the resources A and B so that

- Some threads need A
- Some threads need B
- Some threads need both.

Allocating A, B or both; Sverres suggestion:

```
allocate(resourceList){
  wait(LockManager);
  if(lm_resoursesAreFree(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);
}
```

Starvation

The threads in the last example requiring both A and B may get to run too seldomly.
Other typical examples where this happens:

— Read/Write-locking.

— "Memory allocation" - fluid but limited resources where some threads require a large portion.

— If by some algorithm or OS artifact some thread always end up last in the queue.

And by the way

— ...who says that the intuitively ’fair’ solution, where everybody gets to run equally, is the good one?

— ...which of the waiting threads are awoken when a semaphore is signaled?

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);
    }
  }
}
```

The mutex extension of condision variables

POSIX Monitors:

- Condition synchronization (like named suspend/resume) can be used when having allocated a mutex.
- When waiting for a condition, the mutex can be unlocked.

SlideTemplate