NTNU

Innovation and Creativity

Assorted slides used in ttk4145 lectures

Sverre Hendseth

Department of Engineering Cybernetics

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)

NTNU
Innovation and Creativity

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.

NTNU
Innovation and Creativity

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 improvment 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!

NTNU
Innovation and Creativity

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 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

NTNU
Innovation and Creativity

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

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
Innovation and Creativity

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 TPerson, and the two functions *reallocateArray*() and *getArray*()

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
Innovation and Creativity

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?

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
Innovation and Creativity

Module Interfaces Checklist: Language-Specific Issues

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

NTNU
Innovation and Creativity

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
Innovation and Creativity

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
```

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
```

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
Innovation and Creativity

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
Innovation and Creativity

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
Innovation and Creativity

A good routine name

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

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
Innovation and Creativity

Other Tips

- Standardize parameter orders.
- Standardize naming conventions.

Innovation and Creativity

Big-Picture Issues

- Is the reason for creating the routine sufficient?
- Have all parts of the routine that would benefit from being put into rou- tines 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
Innovation and Creativity

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
Innovation and Creativity

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 inac- curate, 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.

A good variable name

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

NTNU
Innovation and Creativity

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
Innovation and Creativity

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 mean- ingful?
- — 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
Innovation and Creativity

Naming Conventions

— Does the convention distinguish among local, class, and global data?

— Does the convention distinguish among type names, named constants, enumerated types, and variables?

— Does the convention identify input-only parameters to routines in languages that don't enforce them?

— Is the convention as compatible as possible with standard conventions for the language?

— Are names formatted for readability?

Short Names

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

NTNU
Innovation and Creativity

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?

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 gen- eral enough to be used for more than one purpose are usually bad names.

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

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

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

— Code is read far more times than it is written. Be sure that the names you choose favor read-time convenience over write-time

NTNU Innovation and Creativity

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

NTNU
Innovation and Creativity

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
Innovation and Creativity

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-indul- gent 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
Innovation and Creativity

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
Innovation and Creativity

Control Structures

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

Routines

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

Files, Classes, and Programs

— Does the program have a short document, such as that described in the Book Paradigm, that gives an overall view of how the program is organized?

— Is the purpose of each file described?

— Are the author’s name, e-mail address, and phone number in the listing?

NTNU
Innovation and Creativity

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
Innovation and Creativity

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
Innovation and Creativity

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
Innovation and Creativity

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 comment[illegible] but this would still be a badly designed module interface.

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
Innovation and Creativity

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
Innovation and Creativity

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)

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
Innovation and Creativity

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;
}
```

NTNU
Innovation and Creativity

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.

NTNU
Innovation and Creativity

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
Innovation and Creativity

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;
    }
  }
}
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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;
```

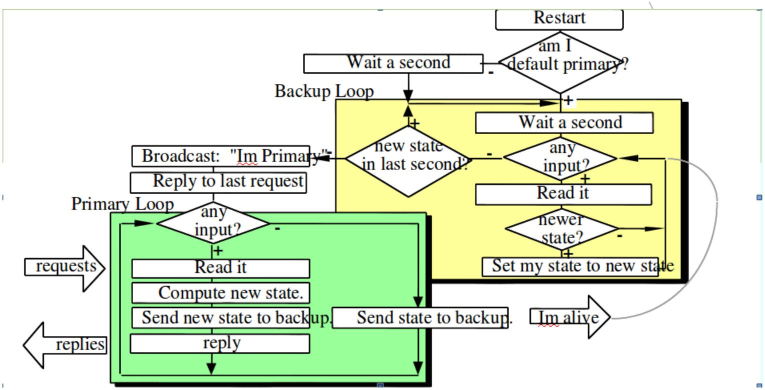
NTNU
Innovation and Creativity

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
Innovation and Creativity

Process Pairs



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
Innovation and Creativity

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
Innovation and Creativity

Locking; basic fixing of containment

```
Allocate locks
  Do work
Release locks
```

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
Innovation and Creativity

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
Innovation and Creativity

Introducing a Transaction Manager

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

The Transaction Manager

- 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 (Status prepareToCommit(tid)), counts votes, commits(tid) or aborts(tid) and returns status.

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 reuseable forms - works only on the RMs data structures!

NTNU
Innovation and Creativity

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
Innovation and Creativity

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.

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
Innovation and Creativity

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?

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 decition 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.

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.

Writing to the *log* – A genious extension!

— Also write before-state into logrecords.

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

NTNU
Innovation and Creativity

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.

NTNU
Innovation and Creativity

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 reciept for received logrecord rather than waiting for the disk access.

NTNU
Innovation and Creativity

... 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.

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.

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 loosing availability.
- Consistency: How ensure a consistent total system after a subsystem restart ?

NTNU
Innovation and Creativity

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.

NTNU
Innovation and Creativity

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 mappin
  }
}
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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 a
        case "fail"
          // increment nOfVotes, set failFlag
          // if nOfVotes == nOfParticipants, send abort to all.
  }
}
```

NTNU
Innovation and Creativity

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;
}
```

NTNU
Innovation and Creativity

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.

NTNU
Innovation and Creativity

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?

Failed attempt 1

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

  if count == n: barrier.signal()

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

One-time barrier Solution

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

  if count == n: barrier.signal()

  barrier.wait()
  barrier.signal()

// critical point
```

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
Innovation and Creativity

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
Innovation and Creativity

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
Innovation and Creativity

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()
}
```

Allocating A, B or both

```
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);
}
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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);
}                                }
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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;
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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
Innovation and Creativity

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();
}
```

NTNU
Innovation and Creativity

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

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

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;
```

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;
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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;
```

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
Innovation and Creativity

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
Innovation and Creativity

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;
}
```

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
Innovation and Creativity

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;
```

NTNU
Innovation and Creativity

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;
```

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
Innovation and Creativity

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;
```

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
Innovation and Creativity

Request parameters in Ada with Requeue - 2-III

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

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;
```

NTNU
Innovation and Creativity

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
Innovation and Creativity

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;
```

NTNU
Innovation and Creativity

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;
```

NTNU
Innovation and Creativity

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 loop;
    end loop;
  end Manager;
end Resource_Manager;
```

NTNU
Innovation and Creativity

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();
}
```

NTNU
Innovation and Creativity

Learning Goals, Modelling of concurrent programs

- An understanding of how using messagebased synchronization leads to a very different design than shared variable synchronization.
- Model, in FSP and by drawing transition diagrams, simple programs (semaphore-based or messagepassing).
- Draw very simple compound (for paralell 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.

NTNU
Innovation and Creativity

Learning Goals,Scheduling

- 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.

NTNU
Innovation and Creativity

—

Innovation and Creativity

—

Innovation and Creativity