CS_313 High Integrity Systems/ CS_M13 Critical Systems

Course Notes

Chapter 0: Introduction

Anton Setzer
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/∼csetzer/lectures/critsys/11/index.html

December 2, 2011

Definition

Definition: A **critical system** is a

- computer, electronic or electromechanical system
- the failure of which may have serious consequences, such as
  - substantial financial losses,
  - substantial environmental damage,
  - injuries or death of human beings.

Three Kinds of Critical Systems.

- **Safety-critical systems**.
  - Failure may cause injury or death to human beings or substantial environmental harm.
  - Main topic of this module.
- **Mission-critical systems.**
  - Failure may result in the failure of some goal-directed activity.
- **Business-critical system.**
  - Failure may result in the failure of the business using that system.

Examples of Critical Systems

- **Safety-Critical**
  - Medical Devices.
  - Aerospace
    - Civil aviation.
    - Military aviation.
    - Manned space travel
  - Chemical Industry.
  - Nuclear Power Stations.
  - Traffic control.
    - Railway control system.
    - Air traffic control.
    - Road traffic control (esp. traffic lights).
    - Automotive control systems.
  - Other military equipment.

Examples of Critical Systems (Cont.)

- **Mission-critical**
  - Navigational system of a space probe.

Examples of Critical Systems (Cont.)

- **Business critical**
  - Customer account system in a bank.
  - Online shopping cart.
  - Areas where secrecy is required.
    - Defence.
    - Secret service.
    - Sensitive areas in companies.
  - Areas where personal data are administered.
    - Police records.
    - Administration of data of customers.
    - Administration of student marks.

Failure of a Critical System

Picture copied from book not included

Primary vs. Secondary

There are 2 kinds of of safety critical software.

- **Primary safety-critical software.**
  - Software embedded as a controller in a system.
  - Malfunction causes hardware malfunction, which results directly in human injury or environmental damage.

Primary vs. Secondary

Secondary safety-critical software.

- Software indirectly results in injury.
- E.g. software tools used for developing safety critical systems.
  - Malfunction might cause bugs in critical systems created using those tools.
- Medical databases.
  - A doctor might make a mistake because of
    - wrong data from such a database,
    - data temporarily not available from such such a database in case of an emergency.

Plan

- **Learning outcome:**
  - Familiarity with issues surrounding safety-critical systems, including
    - legal issues,
    - ethical issues,
    - hazard analysis,
    - techniques for specifying and producing high integrity software.
  - Understanding of techniques for specifying and verifying high-integrity software.
  - Familiarity with and experience in applying programming languages suitable for developing high-integrity software for critical systems (e.g. SPARK Ada).

Plan

0. Introduction, overview.
1. Programming languages for writing safety-critical software.
2. SPARK Ada.
3. Safety criteria.
4. Hazard and risk analysis.
5. The development cycle of safety-critical systems.
6. Fault tolerance.
7. Verification, validation, testing.

Address

Dr. A. Setzer
Dept. of Computer Science
Swansea University
Singleton Park
SA2 8PP
UK

Room Room 211, Faraday Building
Tel. (01792) 513368
Fax. (01792) 295651
Email a.g.setzer@swansea.ac.uk

Two Versions of this Module

- There are 2 Versions of this Module:
  - **Level 3** Version:
    CS_313 High Integrity Systems.
    - 20% coursework.
    - 80% exam.
  - **Level M** Version (MSc, MRes and 4th year):
    CS_M13 Critical Systems.
    - 30% coursework.
    - 70% exam.
  - In both cases the coursework consists of one report (deadline Monday 12 December 2011, 11:00).
  - In case of the Level M version, the report is expected to more thorough and longer, and should demonstrate a deeper level of understanding.
- I usually refer to this module as "Critical Systems".

Report

- Goal of the report is to explore one aspect of critical systems in depth.

Rules concerning Plagiarism

- As for all coursework, **rules concerning plagiarism** have to be obeyed.
  - Text coming from other sources has to be
    - flagged as such (best way is to use **quotation marks**)
    - the source has to be **cited precisely**.
  - The same applies to **pictures, diagrams etc.**

3 Groups of Report Topics

1. **More essay-type topics.**
   - Suitable for those who are good at writing.
2. **Technically more demanding topics** in the area of formal methods.
   - Suitable for those who are good in mathematics/logic.
3. **Writing, specifying or verifying** a toy example of a critical system in one of a choice of languages.
   - Suitable for those with interest in programming.

Literature

- There might be some lack of literature, and the library has a limited stock of books in this area.
- Please share books amongst you.

Home Page of the Module

- The homepage for this module is located at
  http://www.cs.swan.ac.uk/∼csetzer/lectures/critsys/11/index.html
- There is an open version, and a password protected version.
  - The password is _____.
- Errors in the notes will be corrected on the slides continuously and noted on the **list of errata.**
- The homepage contains as well **additional material** for each section of the module.

Åsta Train Accident (January 5, 2000)

Picture copied from book not included

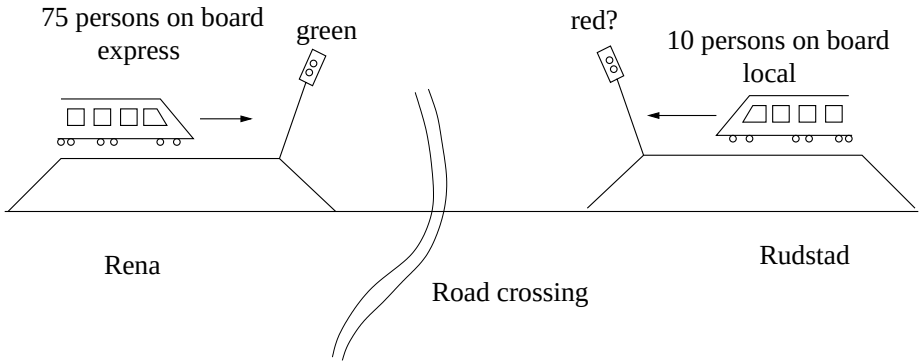
Report from November 6, 2000
http://odin.dep.no/jd/norsk/publ/rapporter/aasta/

Picture copied from book not included

Picture copied from book not included

Picture copied from book not included

Sequence of Events



Sequence of Events

Rena
Train 2302
75 passengers

Rudstad
Train 2369
10 passengers

- Railway with **one track** only. Therefore **crossing of trains only at stations possible**.
- According to timetable crossing of trains at Rudstad.

Sequence of Events

Rena
Train 2302
75 passengers

Rudstad
Train 2369
10 passengers

- Train 2302 is 21 minutes behind schedule.
  When reaching Rena, delay is reduced to 8 minutes.
  Leaves Rena after a stop with green exit signal **13:06:15**,
  in order to cross 2369 at Rudstad.

Sequence of Events

Rena
Train 2302
75 passengers

Rudstad
Train 2369
10 passengers

- Train 2369 leaves after a brief stop Rudstad **13:06:17**,
  2 seconds after train 2302 left and 3 minutes ahead of timetable,
  probably in order to cross 2302 at Rena.

Sequence of Events

Rena
Train 2302
75 passengers

Rudstad
Train 2369
10 passengers

- Local train shouldn't have had green signal.
- **13:07:22** Alarm signalled to the rail traffic controller (no audible signal).
- Rail traffic controller sees alarm approx. **13:12**.

Sequence of Events

Rena
Train 2302
75 passengers

Rudstad
Train 2369
10 passengers

- Traffic controller couldn't warn trains, because of use of mobile telephones (the correct telephone number hadn't been passed on to him).
- Trains collide 13:12:35, 19 persons are killed.

Picture copied from book not included

Investigations

- **18 April 2000** (after the accident): Train has green exit signal.
  When looking again, notices that the signal has turned red.
  Traffic controller hasn't given exit permission.

- Several safety-critical deficiencies in the software found (some known before!)
- The software used was entirely replaced.

Investigations

- No technical faults of the signals found.
- Train driver was not blinded by sun.
- **Four incidents** of wrong signals with similar signalling systems reported:
  - Exit signal green and turns suddenly red.
    Traffic controller says, he didn't give an exit permission.
  - Hanging green signal.
  - Distant signal green, main signal red, train drives over main signal, and pulls back.
    Traffic controller surprised about the green signal.

SINTEF

- **SINTEF** (Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology) found no mistake leading directly to the accident.
  Conclusion of SINTEF: No indication of abnormal signal status.
  $\Rightarrow$ Mistake of train driver (died in the accident).
  (Human Error).
- Assessment of report by **Railcert**
  - Criticism: SINTEF was only looking for single cause faults, not for multiple causes.

Conclusion

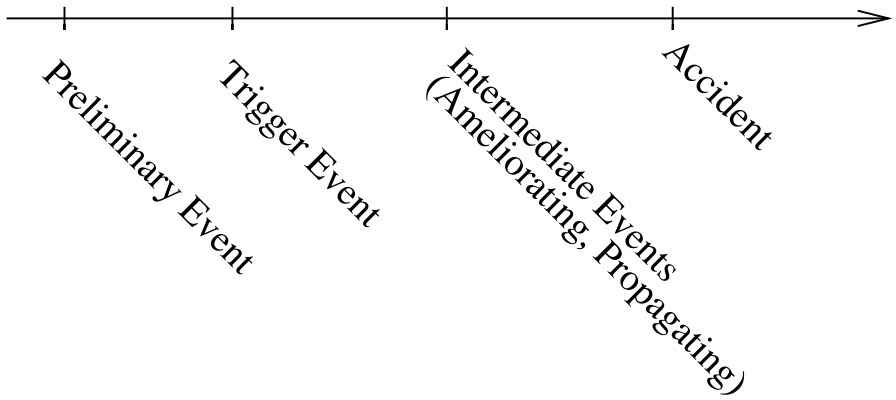
- It is **possible** that the train driver of the local train was **driving against a red signal**.
- The fact that **he was stopping and left almost at the same time as the other train and 3 minutes ahead of time**, makes it likely that he received an erroneous green exit signal due to some software error.
  It could be that the software under certain circumstances when giving an entrance signal into a block, for a short moment gives the entrance signal for the other side of the block.

Conclusion (Cont.)

- Even if this particular accident was not due to a software error, apparently this software has several **safety-critical errors.**
- In the protocol of an **extremely simple installation** (Brunna, 10 km from Uppsala), which was established 1957 and exists in this form in 40 installations in Sweden, a **safety-critical error was found** when verifying it with a theorem prover formally 1997.
- Lots of other **errors** in the Swedish railway system were **found during formal verification**.
  - Lecturer is together with PhD/MRes students involved in an industrial project on verification of railway systems.

Lessons to be Learned

A **sequence of events** has to happen in order for an accident to take place.



Events Leading to an Accident

- **Preliminary events.**
  = events which influence the initiating event.
  without them the accident cannot advance to the next step (initiating event).
  In the main example:
  - Express train is late. Therefore crossing of trains first moved from Rudstad to Rena.
  - Delay of the express train reduced. Therefore crossing of trains moved back to Rudstad.

Lessons to be Learned (Cont.)

- **Initiating event**, **trigger event.**
  Mechanism that causes the accident to occur.
  In the main example:
  - Both trains leave their stations on crash course, maybe caused by both trains having green signals.

Events Leading to an Accident

- **Intermediate events.**
  Events that may propagate or ameliorate the accident.
  - **Ameliorating events** can prevent the accident or reduce its impact.
  - **Propagating events** have the opposite effect.
- When designing safety critical systems, **one should**
  - **avoid triggering events**
    - if possible by using several independent safeguards,
  - add additional safeguards, which **prevent a triggering event from causing an accident** or reduces its impact.

Analysis of Accidents

- Goal of investigations of accidents usually concentrates on legal aspects.
  - Who is guilty for the accident?
- If one is interested in preventing accidents from happening, one has to carry out a more thorough investigation.
  - Often an accident happens under circumstances, in which an accident was bound to happen eventually.
  - It doesn't suffice to try to prevent the trigger event from happening again.
    - Problem with safety culture might lead to another accident, but that will probably be triggered by something different.

Causal Factors

We consider a **three-level model** (Leveson [Le95], pp. 48 – 51) in order to identify the real reasons behind accidents.

- **Level 1: Mechanisms, Chain of events.**
  - Described above.
- **Level 2: Conditions,** which allowed the events on level 1 to occur.
- **Level 3: Conditions and Constraints,**
  - that allowed the conditions on the second level to cause the events at the first level, e.g.
    - Technical and physical conditions.
    - Social dynamics and human actions.
    - Management system, organisational culture.
    - Governmental or socioeconomic policies and conditions.

Root Causes

- Problems found in a Level 3 analysis form the **root causes.**
  - **Root causes** are weaknesses in general classes of accidents, which contributed to the current accident but might affect future accidents.
  - **If** the problem behind a root cause is **not fixed**, almost inevitably an **accident will happen again**.
  - Many examples, in which despite a thorough investigation the **root cause was not fixed**, and the accident **happened again.**

Root Causes

- **Example: DC-10 cargo-door saga.**
  - Faulty closing of the **cargo door** caused **collapsing of the cabin floor.**
  - One DC-10 crashed 1970, killing 346 people.
    - DC-10 built by McDonnell-Douglas.
  - As a consequence a fix to the **cargo doors** was applied.
  - The root cause, namely the **collapsing of the cabin floor** when the cargo door opens, **wasn't fixed.**
  - 1972, the cargo door latch system in a DC-10 failed, the **cabin floor collapsed**, and only due to the extraordinary skillfulness of the pilot the plane was not lost.

Level 2 Conditions

- Level 1 Mechanism: The driver **left** the station **although the signal should have been red**.
  Caused by Level 2 Conditions:
  - The local train might have had for short period a green light, caused by a software error.

Level 2 Conditions

- Level 1 Mechanism: The local train **left early.**
  Caused by Level 2 Conditions:
  - Might have been caused by the train driver **relying on his watch** (which might go wrong), and not on an official clock.

Level 2 Conditions

- Level 1 mechanism: The local train **drove over a possibly** at that moment **red light**.
  Caused by Level 2 condition:
  - There was no ATP (automatic train protection) installed, which stops trains driving over a red light.

Level 2 Conditions (Cont.)

- Level 1 mechanism: The traffic controller **didn't see the control light**.
  Caused by Level 2 conditions:
  - Control panel was badly designed.
  - A visual warning signal is not enough, in case the system detects a possible collision of trains.

Level 2 Conditions (Cont.)

- Level 1 mechanism: The rail controller **couldn't warn the driver**, since he didn't know the mobile telephone number.
  Caused by level 2 conditions:
  - Reliance on a mobile telephone network in a safety critical system. This is extremely careless:
    - Mobile phones often fail.
    - The connections might be overcrowded.
    - Connection to mobiles might not work in certain areas of the railway network.
  - The **procedure for passing on the mobile phone numbers was badly managed**.

Level 2 Conditions (Cont.)

- Level 1 mechanism: A severe fire broke out as a result of the accident. Caused by Level 2 condition:
  - The **fire safety of the train engines** was not very good.

Level 3 Constraints and Conditions

- **Cost-cutting precedes many accidents.**
  - Difficult, to maintain such a small railway line.
  - Resulting cheap solutions might be dangerous.
  - The railway controller had too much to do and was **overworked**.
- **Flaws in the software.**
  - Software **wasn't written according to the highest standards**.
  - Control of railway signals is a safety-critical system and should be designed with high level of integrity.
  - Very difficult to write correct protocols for distributed algorithms.
  - Need for **verified design** of such software.

Level 3 Constraints and Conditions

Poor human-computer interface at the control panel.

- Typical for lots of control rooms.
  - Known problem for many nuclear power stations.
  - In the accident at the Three Mile Island nuclear power plant at Harrisburg, (a loss-of-coolant accident which costed between 1 billion and 1.86 billion US-$)
    - there were lots of problems in the design of the control panel that lead to human errors in the interpretation of the data during the accident;
    - one of the key indicators relevant to the accident was on the back of the control panel.

Level 3 Constraints and Conditions

- Criticality of the design of human computer interfaces in control rooms is not yet sufficiently acknowledged.
  - Example: Problems with the UK air traffic control system, which are sometimes difficult to read.
  - In case of an accident, the operator will be blamed.

Level 3 Constraints and Conditions

- **Overconfidence in ICT**.
  - Otherwise one wouldn't have used such a badly designed software.
  - Otherwise one wouldn't have simply relied on the mobile phones – at least a special agreement with the mobile phone companies should have been set up.

Level 3 Constraints and Conditions

- **Flaws in management practices**.
  - No protocol for dealing with mobile phone numbers.
  - No mechanism for dealing with incidents.
  - Incidents had happened before, but were not investigated.
    - A mechanism should have been established to thoroughly investigate them.
    - Most accidents are preceded by incidents, which are not taken seriously enough.

Lessons to be Learned

- Usually at the end of an investigation conclusion **"human error"**.
- **Uncritical attitude towards software:**
  The architecture of the software was investigated but no detailed search for a bug was done.
- Afterwards, **concentration on trigger events**, but not much attention to preliminary and intermediate events – root cause often not fixed.
- Most failures of safety-critical systems were caused by **multiple failures**.
- **Incidents precede accidents.**
  - If one doesn't learn from incidents, eventually an accident will happen.

(1) Software engineering Aspect

- Safety-critical systems are very complex – **System aspect.**
  - **Software**, which includes parallelism.
  - **Hardware**.
    - Might fail (light bulb of a signal might burn through, relays age).
    - Have to operate under **adverse conditions** (low temperatures, rain, snow).
- Interaction with **environment**.
- **Human-computer interaction.**
- **Protocols** the operators have to follow.

(1) Software engineering Aspect (Cont.)

  - **Training** of operators.
  - **Cultural habits.**
- For dealing with this, we need to look at aspects like
  - **Methods for identifying hazards** and measuring risk (HAZOP, FMTA etc.)
  - **Standards.**
  - **Documentation** (requirements, specification etc.)
  - **Validation and verification.**
  - **Ethical and legal aspects.**
- Based on **techniques used in other engineering disciplines** (esp. chemical industry, nuclear power industry, aviation).

(2) Tools for writing correct software

- Software bugs can not be avoided by careful design.
  - Especially with distributed algorithms.
- Need for verification techniques using **formal methods.**
- **Different levels of rigour:**
  - **(1)** Application of formal methods **by hand**, without machine assistance.
  - **(2)** Use of **formalised specification languages with some mechanised support tools**.
  - **(3)** Use of **fully formal specification languages with machine assisted or fully automated theorem proving.**

(2) Tools for Writing Correct Software (Cont.)

- However, such methods **don't replace software engineering techniques**.
  - Formal methods **idealise a system** and **ignore aspects** like hardware failures.
  - With formal methods one can show that some software fulfils its formal specification.
    But checking that the specification is sufficient in order to guarantee safety cannot be done using formal methods.

Race Conditions

- When investigating the Åsta train accident, it seems that **race conditions** could have been the technical reason for that accident.
- Race conditions occur in programs which have several **threads**.
- An well-known studied example of problems with race conditions is the **Therac-25**.
  - The Therac-25 was a computer-controlled radiation therapy machine, which 1985 – 1987 overdosed massively 6 people.
  - There were many more incidents.

Therac 25

- Two of the threads in the Therac 25 were:
  - The **keyboard controller**, which controls data entered by the operator on a terminal.
  - The **treatment monitor**, which controls the treatment.
- The data as received by the keyboard controller was passed on the treatment monitor using **shared variables**.

Therac 25

Threads:
- Keyboard Controller.
- Treatment Monitor.

- It was possible that
  - the operator enters data,
  - the keyboard controller signals to the treatment monitor that data entry is complete,
  - the treatment monitor checks the data and starts preparing a treatment,
  - the operator changes his data,
  - the treatment monitor never realises that the data have changed, and therefore never checks them,
  - but the treatment monitor uses the changed data, even if they violate the checking conditions.

Problems with Concurrency

- Complete story of the Therac-25 is rather complicated.
- In general, we have here an example of having several threads, which have to communicate with each other.
  - Note that the concurrency between the user interface and the main thread might **be ignored** in a formal verification.

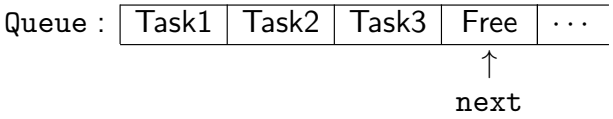
  $\Rightarrow$ **Limitations of formal verification** and need to take into account the **systems aspect**.

Race Conditions

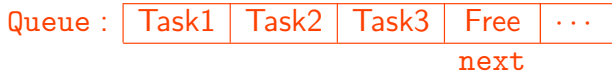
- Although in the Therac-25 case there was **no problem** of **race conditions in a strict sense** (it was a lack of fully communicating changes by one thread to another thread), when designing systems with concurrency one has to be aware of the possibility of race conditions.
- Problems with race conditions are very **common in critical systems**, since most critical systems involve some degree of **concurrency**.

Race Conditions

- **Race conditions** occur if two threads share the same variable.
- Typical scenario is if we have an array `Queue` containing values for tasks to be performed, plus a variable `next` pointing to the the first free slot:

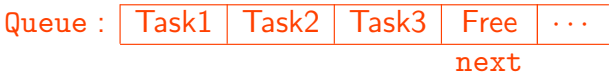


Race Conditions



- Assume two threads:
  - Thread 1:
    ```
    line 1.1:  Queue[next]= TaskThread1;
    line 1.2:  next := next + 1;
    ```
  - Thread 2:
    ```
    line 2.1:  Queue[next]= TaskThread2;
    line 2.2:  next := next + 1;
    ```
  - Assume that after line 1.1, Thread 1 is interrupted, Thread 2 executes line 2.1 and line 2.2, and then Thread 1 executes line 1.2.

Race Conditions




```
Thread 1   line 1.1:   Queue[next]= TaskThread1;
           line 1.2:   next := next+ 1;
Thread 2   line 2.1:   Queue[next]= TaskThread2;
           line 2.2:   next := next + 1;
```

Execution: 1.1 → 2.1 → 2.2 → 1.2.

- Let `next`∼ be the value of `next` before this run and write `next` for its value after this run.
- **Result:** `Queue[next`∼`] = TaskThread2`.
  `Queue[next`∼`+1]` = previous value of `Queue[next`∼`+1]` (could be anything).
  `next= next`∼`+ 2`.

Race Conditions


```
Thread 1   line 1.1:   Queue[next]= TaskThread1;
           line 1.2:   next := next + 1;
Thread 2   line 2.1:   Queue[next]= TaskThread2;
           line 2.2:   next := next + 1;
```

- **Problem:** In most test-runs Thread 1 will not be interrupted by Thread 2.
  - It's difficult to detect race conditions by tests.
  - It's difficult to debug programs with race conditions.

Critical Regions

- **Solution** for the race conditions.
  - Form groups of shared variables between threads, such that variables in different groups can be changed independently without affecting the correctness of the system, i.e. such that different groups are **orthogonal**.
    - In the example `Queue` and `next` must belong to the same group, since `next` cannot be changed independently of `Queue`.
    - However, we might have a similar combination of variables `Queue2`/`next2`, which is independent of `Queue`/`next`. Then `Queue2`/`next2` forms a second group.

Critical Regions

- Lines of code involving shared variables of one group are critical regions for this group.
  - A **critical region for a group** is a sequence of instructions for a thread reading or modifying variables of this group, such that if during the execution of these instructions one of the variables of the same group is changed by another thread, then the system might reach a state which results in incorrect behaviour of the system.
- Critical regions for the same group of shared variables have to be **mutually exclusive**.

Critical Regions

Queue :

Task1	Task2	Task3	Free	···

next

```
Thread 1   line 1.1:   Queue[next]= TaskThread1;
           line 1.2:   next := next + 1;
Thread 2   line 2.1:   Queue[next]= TaskThread2;
           line 2.2:   next := next + 1;
```

- Line 1.1/1.2 and line 2.1/2.2 are critical regions for this group.
- When starting line 1.1, thread 1 enters a critical region for `Queue/next`.

Critical Regions

Queue :

Task1	Task2	Task3	Free	···

next

```
Thread 1   line 1.1:   Queue[next]= TaskThread1;
           line 1.2:   next := next + 1;
Thread 2   line 2.1:   Queue[next]= TaskThread2;
           line 2.2:   next := next + 1;
```

- Before it has finished line 1.2, it is not allowed to switch to any thread with the same critical region, e.g. line 2.1/2.2 of thread 2.
- But it is not a problem to interrupt Thread 1 between line 1.1/1.2 and to change variables, which are independent of `Queue/next`.

Synchronised

- In Java achieved by keyword `synchronized`.
- A method, or block of statements, can be synchronised w.r.t. any object.
- Two synchronised blocks of code/methods, which are synchronised w.r.t. the same object, are mutually exclusive.
- But they can be interleaved with code of any other code, which is not synchronised w.r.t. the same object.
- Non-static code, which is synchronised without a specifier, is synchronised w.r.t to the object it belongs to.
- Static code, which is synchronised without a specifier, is synchronised w.r.t to the class it belongs to.

Synchronisation in the Example

Queue :

Task1	Task2	Task3	Free	···

next

The example above corrected using Java-like pseudo code is as follows:

```
Thread 1   synchronized(Queue){
             Queue.Queue[Queue.next]= TaskThread1;
             Queue.next := Queue.next + 1;}
Thread 2   synchronized(Queue){
             Queue.Queue[Queue.next]= TaskThread2;
             Queue.next := Queue.next + 1;}
```

Possible Scenario for Railways

- The following is a highly simplified scenario:

Signal 1 →		← Signal 2
Train1 ⟶	Segment 1	⟵ Train2

- Thread Train 1:

```
if (not segment1_is_blocked){
        signal1 := green;
        segment1_is_blocked:= true;}
```

- Thread Train 2:

```
if (not segment1_is_blocked){
        signal2 := green;
        segment1_is_blocked:= true;}
```


Possible Scenario for Railways

Signal 1 →		← Signal 2
Train1 ⟶	Segment 1	⟵ Train2

```
Train1  l 1.1:  if (not segment1_is_blocked){
        l 1.2:          signal1 := green;
        l 1.3:          segment1_is_blocked:= true;}
Train2  l 2.1:  if (not segment1_is_blocked){
        l 2.2:          signal2 := green;
        l 2.3:          segment1_is_blocked:= true;}
```

- If not synchronised, one might execute
  l 1.1 → l 2.1 → l 1.2 → l 1.3 → l 2.2 → l 2.3
  This sequence results in signal1 and signal2 being green.

Possible Scenario for Railways

- If one has a checker, it might recognise later the problem and repair it, but still for some time both trains have green light.
- The above scenario is of course too simple to be realistic, but one can imagine a more complicated variant of the above going on hidden in the code.
- Problems of this nature could be one possible reason, why in the system used in Norway it sometimes happened that a signal switched for a short moment to green.

Literature

- In general, the module is self-contained.
- In the following a list of books, which might be of interest, if you later have to study critical systems more intensively, or for your report.

Main Course Literature

- **Main course book:**

  [St96] Storey, Neil: *Safety-critical computer systems.* Addison-Wesley, 1996.

- **Additional Overview Texts**

  [Ba97a] Bahr, Nicolas J.: *System safety and risk assessment: a practical approach.* Taylor & Francis, 1997.
  Intended as a short book for engineers from all engineering disciplines.

  [So01] Part 4 and 5 in Sommerville, Ian: *Software Engineering.* 6th Edition, Addison-Wesley, 2001.
  Concise and nice overview over software engineering aspects of safety-critical systems.

Copyright

- Since this lecture is heavily based on the book by Neil Storey [St96], a lot of material in these slides is taken from that book.

Additional Overview Texts

[Le95] Nancy G. Leveson: *Safeware. System safety and computers.* Addison-Wesley, 1995.
Concentrates mainly on human and sociological aspects. More advanced book, but sometimes used in this module.

Preliminaries for Further Books

- The following book recommendations are more intended, if you need
  - literature for your reports,
  - need later literature when working in industry on critical systems.
- Gives as well an overview over various disciplines involved in this area.

Further General Books

[Ne95] Peter G. Neumann: *Computer related risks.* Addison-Wesley, 1995.
A report on a lot of (100?) accidents and incidents of critical errors in software.

[GM02] Geffroy, Jean-Claude; Motet, Gilles: *Design of dependable computing systems.* Kluwer, 2002.
Advanced book on the design of general dependable computer systems.

[CF01] Crowe, Dana; Feindberg, Alec: *Design for reliability.* CRC Press, 2001.
Book for electrical engineers, who want to design reliable systems.

Books on Reliability Engineering

- **Reliability engineering** uses probabilistic methods in order to determine the degree of reliability required for the components of a system.

[Mu99] Musa, John: *Software reliability engineering.* McGraw-Hill, 1999.
Bible of reliability engineering. Difficult book.

[Sm02] Smith, David J: *Reliability, maintainability and risk.* Butterworth Heinemann Oxford. 6th Ed., 2002
Intended for practitioners.

[Xi91] Xie, M. *Software reliability modelling.* World Scientific, 1991.
Introduces various models for determining reliability of systems.

Further General Books

- **HAZOP and FMEA** are techniques for identifying hazards in systems.

[RCC99] Redmill, Felix; Chudleigh, Morris; Catmur, James: *System safety: HAZOP and software HAZOP.* John Wiley, 1999.
General text on HAZOP.

[MMB96] McDermott, Robin E.; Mikulak, Raymond, J.; Beauregard, Michael, R.: *The basics of FMEA.* Productivity Inc, Portland, 1996.
Booklet on FMEA.

Further General Books

- **Software testing techniques.**

[Pa01] Patton, Ron: *Software testing.* SAMS, 2001.
Very practical book on techniques for software testing.

[DRP99] Dustin, Elfriede; Rashka, Jeff; Paul, John: *Automated software testing.* Addison-Wesley, 1999.
Practical book on how to test software automatically.

Formal Methods

- **Overview Books.**

[Pe01] Peled: *Software Reliability Methods.* Springer 2001.
Overview over formal methods for designing reliable software.

[Mo03] Monin, Jean-Francois: *Understanding formal methods.* Springer, 2003.
General introduction into formal methods.

Formal Methods Used in Industry

- **Z.** Industrial standard method for specifying software.

[Ja97] Jacky, Jonathan: *The way of Z. Practical programming with formal methods.* Cambridge University Press, 1997.
Practical application of the specification language Z in medical software.

[Di94] Diller, Antoni: *Z. An introduction to formal methods.* 2nd Ed., John Wiley, 1994.
Introduction into Z.

[Li01] Lightfoot, David: *Formal specification using Z.* 2nd edition, Palgrave, 2001.

Formal Methods Used in Industry

- **B-Method.** Developed from Z. Becoming the standard for specifying critical systems.

[Sch01] Steve Schneider: *The B-method.* Palgrave, 2001.
Introduction into the B-method.

[Ab96] Abrial, J.-R.: *The B-Book.* Cambridge University Press, 1996.
The "bible" of the B-method, more for specialists.

Formal Methods Used in Industry

- **SPARK-Ada**, a subset of the programming language Ada, with proof annotations in order to develop secure software. Developed and used in industry.

[Ba03] John Barnes: *High integrity Software. The SPARK approach to safety and security.* Addison-Wesley, 2003.
Contains a CD with part of the SPARK-Ada system.

[Ba97b] John Barnes: *High integrity Ada. The SPARK approach.* Addison-Wesley, 1997.
(First edition of [Ba03]).

[Na95] David J. Naiditch: *Rendezvous with Ada 95.* John Wildey, 2nd Edition, 1995.
(Good book on the programming language Ada; doesn't refer to SPARK Ada).

Formal Methods Used in Industry

- **Model checking**, used for automatic verification, especially of hardware.

[CGP01] Clarke, Edmund M. Jr.; Grumberg, Orna; Peled, Doron A.: *Model checking.* MIT Press, 3rd printing 2001.
Very thorough overview over this method.

[HR00] Michael R. A. Huth, Mark D. Ryan: *Logic in computer science. Modelling and reasoning about systems.* Cambridge University Press, 2000.
Introduction to logic for computer science students. Has a nice chapter on model checking.