TTK4145 Summary

Martin Brandt

Course Learning Goals:

* General maturation in software engineering/computer programming.
* Ability to use (correctly) and evaluate mechanisms for shared variable synchronisation.
* 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.

CONTENTS

1 | Concurrency and parallelism

Concurrency: running multiple tasks at the same time, but not necessarily on several cores.
Parallelism: actually running tasks on multiple cores or on several devices in a distributed system.
Concurrency helps to utilise the CPU better, and is needed to utilise multicore systems.
Process: an executing program.
Thread: routines running concurrently as part of a process, shares memory.
Green thread: managed and scheduled at run-time, not by the OS, may or may not share memory.
Co-routine: subroutines that cooperatively switch i.e. the subroutines themselves give up control.

2 | FAULT TOLERANCE BASICS

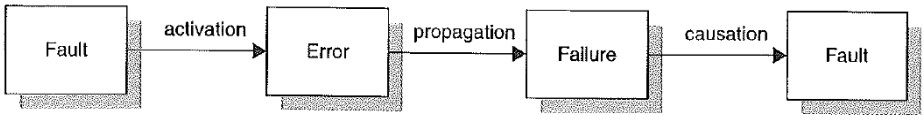
* *Understand and use terms (like): Reliability. Failure vs fault vs error. Failure modes. Acceptance test. Fault prevention vs. tolerance. Redundancy, Static vs. Dynamic. Forward/ Backward error recovery.*
* *Understand, use and evaluate techniques (like): N-version programming. Recovery blocks. Error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

2.1 | Reliability And Fault Tolerance

Failure: system behaviour that deviates from specification.

Faults: A mechanical or algorithmic cause which, given the right conditions, produce an unexpected or incorrect result, i.e. an error. It will then go from dormant to active.

Error: An internal unexpected problem in a system, caused by activation of a fault.



Transient fault: occurs at a particular time, remains in the system for some period and then disappears, e.g. most faults in communication systems.

Permanent faults: occurs at a particular time and remain until fixed.

Intermittent faults: transient faults that occur periodically.

Bohrbugs: reproducible and identifiable.

Heisenbugs: SW bugs that are only active under certain rate conditions, thereby being difficult to detect and even more difficult to debug.

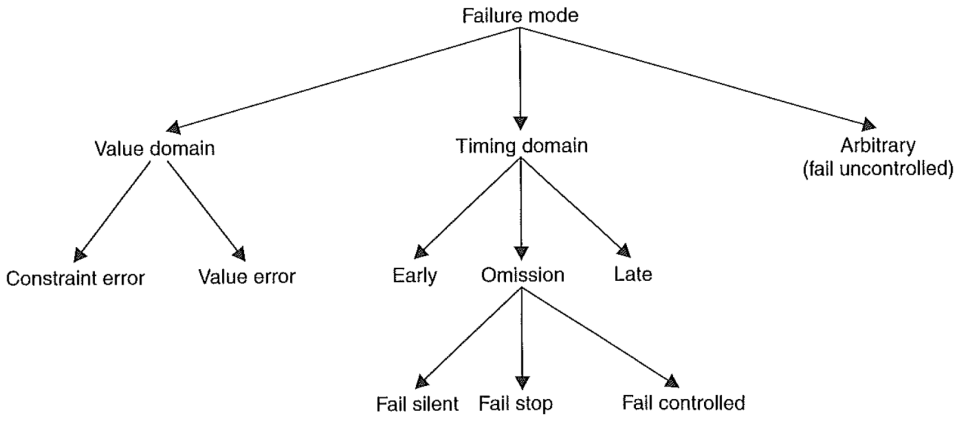
Failure modes: all the ways a system can fail. An important part of design is to to make the failure modes simple and few by merging error modes and handling them as worst case. Two domains of failure modes: *value failures* - value associated with the service is in error, and *time failures* - service is delivered at wrong time.

2.1.1 | Fault prevention

Fault prevention tries to eliminate possibilities of faults before the system is operational. There are two stages: fault avoidance and fault removal. Fault avoidance limits introduction of faulty components by using rigorous specifications, analysis tools, languages that support data abstraction and modularity and reliable hardware components. Fault removal is done by design reviews, verification and testing. Problem: testing can only show existence of errors. Embedded systems have high demands for reliability, safety and availability so testing is not good enough!

2.1.2 | Fault tolerance

We therefore need to consider designing fault tolerant systems! Fault tolerance levels: *full fault tolerance* - system continues to operate with no significant loss of functionality, *graceful degradation* - system continues to operate with



a partial degradation of functionality and *fail safe* - system maintains integrity while accepting a temporary halt in operation.

Redundancy: Introducing extra elements into system to detect and recover from faults. *Static redundancy* adds redundant components inside a system e.g. TMR, while *dynamic redundancy* provide error detection, not error masking, inside the system e.g. checksums.

2.2 | N-version programming

The independent generation of N functionally equivalent programs from the same spec. The programs execute concurrently and the outputs are compared by a driver process which calculates a consensus and acts on the result. The driver and the N versions communicate by *comparison vectors* - the outputs from the programs, *comparison status indicators* - the actions the programs need to perform as a result of the driver's comparison and *comparison points* - the points in time where the versions need to communicate the comparison vectors. The frequency of the comparison points is the *granularity* of the provision. Challenges: price, complexity, maintenance, output comparison is not always easy (inexact voting), doesn't solve ambiguous spec problem.

2.3 | Software dynamic redundancy

Phases: error detection, damage confinement and assessment, error recovery and fault treatment and continued service.

Error detection: *environmental detection* - detected by the environment in which the program runs and *application detection* - detected by application itself. Application detection can be categorised into *replication checks*, *timing checks* - watchdog timers, *reversal checks* - find input from output and compare to actual input, *coding checks* - test for corrupted data e.g. checksums, *reasonableness checks* - check that state of data is valid and reasonable, *structural checks* - check integrity of data structures e.g. status data of list and *dynamic reasonableness checks* - check if output is too different from previous output.

Damage confinement and assessment: want to structure system such that the damage from faults are confined as much as possible within the system. *Modular decomposition* aids to achieve this, as well as *atomic actions* - activities that

have no interactions between the activity and the system during the duration of the action.

Error recovery: two approaches, *forward error recovery* and *backward error recovery*. Forward error recovery tries to correct the erroneous state and let the system continue. Backward error recovery restores the system to a previous safe and consistent state, called a *recovery point*. Need to avoid the *domino effect* where threads rolling back to the previous recovery point triggers another rollback of another thread and so on. This is done by establishing *recovery lines*, which are consistent across all threads.

Fault treatment and continued service: eradicate fault from system, in two stages: *fault location* and *system repair*.

2.4 | Recovery blocks

Recovery blocks are blocks of code with starting points that are recovery points and end points that evaluate an acceptance test and restores the recovery point and tries an alternative module until none are left, in which case the recovery block fails. Acceptance tests in essence merge all the error modes to a single one i.e. if the state of the system is acceptable. They can be implemented by using the different error detection mechanisms described earlier in the chapter. This implements a *fail-fast system:* a system with merged error modes, no wrong outputs and acts as a good building block with easy to compute reliability figures.

3 | 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.*

3.1 | Fault model

Faulty behaviour of a system are either *expected* or *unexpected*, i.e. faults that are tolerated or not tolerated by the system. Unexpected faults are either *dense faults* - more than N faults in a N-fault tolerant system, or *Byzantine faults* - faults where the system does not conform to the model behaviour.
The process of software fault masking starts with finding failure modes, then detecting errors and simplifying the system, followed by error injection or testing and error handling by redundancy.
Highly available storage: write and read may fail. Status and pages with address, value and status. Error detection by checksums, status bits, merging all failure modes. Redundancy by n-plexing, repair threads.
Highly available messages: messages may be lost, delayed, corrupted, duplicated or have a wrong recipient. Error detection by acknowledgements, checksums, sequence numbers and session ids. Sessions make the messages failfast, and retransmission after timeout make them redundant. By also using process pairs we can make the message endpoints available.
Highly available processes: the process fails if it does not execute the next step correctly. This is detected by an acceptance test. Redundancy is added by textitcheckpoint-restarts - state is written to storage after each acceptance test, but before each event, textitpersistent processes where all calculations are transactions, or textitprocess pairs which are discussed below. Checkpoint-restarts are simple, but may have quite long repair times.

3.2 | Process pairs

We have two processes, the primary and the backup. The primary does all the work, and sends i'm alive messages to the backup. When the primary fails and the messages stop coming, the backup takes over for the primary. The takeover can either be by checkpoint-restarts, textitcheckpoint messages - the primary's i'm alive messages include the state, or textitpersistent restarts - the backup starts in null state and the transaction manager cleans up. Note that basically all fault-tolerant application-level process pairs should be programmed as persistent processes (see atomic actions).

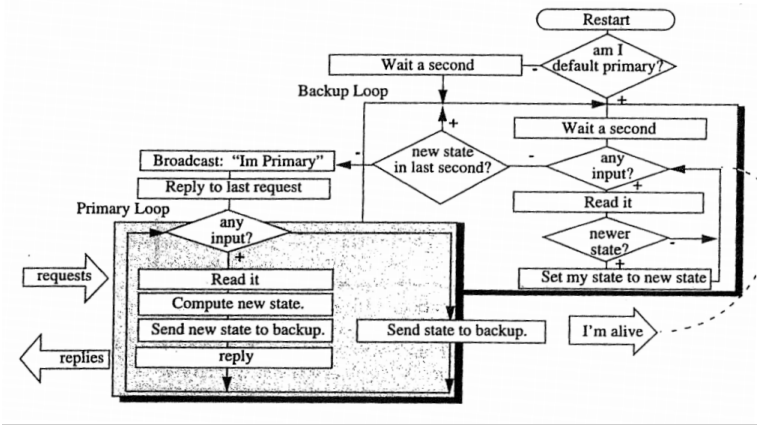


FIGURE 1 Process pairs

4 | ATOMIC ACTIONS, CONCURRENT TASKS AND RELIABILITY

- *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.*

4.1 | Atomic actions

Atomic actions are, as far as other tasks are concerned, considered to be indivisible and instantaneous, and they are not aware of the existence of any other active task. Nesting of atomic actions allows modular decomposition.
Two-phase atomic actions: A resource server is allowed to communicate with atomic actions. In order to maintain the integrity of the atomic action resources can only be obtained in the beginning "growing" phase and only released in the following "shrinking" phase.
Atomic transactions: atomic actions that can either fail or succeed. We say that they have failure atomicity in addition to synchronisation atomicity which all atomic actions are.
Atomic actions have the following requirements:
Well-defined boundaries: need a well-defined start, end and side boundary. The latter is the boundary between the tasks done inside the atomic action and outside, i.e. no communication with non-participants.
Indivisibility: an atomic action must not allow the exchange of any information between the tasks active inside the action and those outside(resource managers excluded).
Nesting: atomic actions may be nested as long as they do not overlap with other atomic actions.
Concurrency: different atomic actions should be possible to run concurrently.

4.2 | Atomic Actions in C/Real-Time POSIX, Ada and Java

POSIX: `PTHREAD_MUTEX_LOCK()`, `PTHREAD_COND_WAIT()` and `PTHREAD_COND_SIGNAL()` to create entry and exit points.

Ada: similar to POSIX, but uses entries with guards instead of condition variables. Guards are tests integrated in Ada that blocks, used to implement boundaries. Also note the outstanding `'Count` feature.

```
package body Action_X is
    protected Action_Controller is
        entry First;
        entry Second;
        entry Third;
        entry Finished;
    private
        First_Here : Boolean := False;
        Second_Here : Boolean :: False;
        Third_Here : Boolean := False;
        Release : Boolean := False;
    end Action_Controller;

    protected body Action_Controller is
        entry First when not First~Here is
        begin
            First_Here := True;
        end First;

        entry Second when not Second_Here is
        begin
            Second_Here := True;
        end Second;

        entry Third when not Third_Here is
        begin
            Third_Here := True;
        end Third;

        entry Finished when Release or Finished'Count = 3 is
        begin
            if Finished'Count = 0 then
                Release := False;
                First_Here := False;
                Second_Here := False;
                Third_Here := False;
            else
                Release := True;
            end if;
        end Finished;
    end Action_Controller;
```

Java: `synchronized` methods only execute one at a time. `wait()` suspends a thread and `notify()` wakes up one of the waiting threads arbitrarily(!). `notifyAll()` wakes up all threads that are waiting on this object's monitor.

4.3 | Asynchronous Notification

Polling for events is not always good enough for distribution of error information during forward error recovery. This is also true for transitions in mode of operation in the system, interrupting computations and user interrupts. The key is asynchronous notification.

Asynchronous notification can either be done by resumption with **asynchronous event handling** or termination with **asynchronous transfer of control**. The former behaves like a software interrupt, where an event handler is executed and the program resumes. In ATC on the other hand, each task specified a domain of execution in which it allows ATC. If a ATC request happens outside the domain, the request may be ignored or queued. After the ATC is handled, control is resumed to the program at a different location(!).

4.3.1 | Asynchronous Notification in C/Real-Time POSIX

C/Real-Time POSIX support resumption with signals and termination with thread cancelling. There are predefined signals like `SIGABRT`, but the program can also specify that it wants to receive signals at events. It can then deal with them by either blocking the signal and either accepting it or handling it later, handling it by setting a function to be called whenever it occurs or simply ignoring it.
We can use the excellent functions `setjmp` and `longjmp` for termination, as well as simply killing threads.

4.3.2 | Asynchronous Notification in Ada

We use the `select` statement in Ada to implement ATC:

```
select
    Trigger.Event;
    -- optional sequence of statements to be
    -- executed after the event has been received
then abort
    -- abortable sequence of statements
end select;
```

The abortable sequence of statements will be executed, and if the triggering statement completes before the execution of the abortable part completes, the abortable part is aborted and the optional sequence of statements are run instead. Note that we easily can create a timeout by using a delay statement as the trigger. Ada allows operations to be `abort deferred`, which means the sequence of statements cannot be cancelled until the abort deferred operation is completed.

4.3.3 | Asynchronous Notification in Real-Time Java

One thread can signal an interrupt to another thread with `interrupt()`. If the thread is waiting it is made runnable, and if it is already executing a slag is set. Kill with `destroy()`.

5 | 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 utilise these patterns in high-level design.*
* *Comprehension of terms: Optimistic Concurrency Control, Two-phase commit optimisations, Heuristic Transactions, Interposition.*

5.1 | ACID

Atomic transaction have the following properties often referred to as the ACID properties:

5.1.1 | Atomicity

The transaction completes successfully i.e. commits, or it fails i.e. aborts and all effects are undone. This is done with the **two-phase commit protocol**, in which the transaction manager tells each participant what to do and waits on C/A back in phase 1, and then decides if all the work should be committed, and forces each participant to commit or abort in phase 2. Issue: not all actions are abortable in the real world.

Success: *Start → prepare phase → commit → write log → commit phase → remove log*

Failure: *Start → prepare phase → abort*

5.1.2 | Consistency

Transactions should produce consistent results and maintain consistency of the resources used.

5.1.3 | Isolation

Intermediate states of the transaction are not visible to the outside, and transactions appear to happen sequentially to the outside.

Pessimistic concurrency control locks resources at the start of a transaction and holds it for the duration of the transaction. This requires a lot of overhead, only supports one layer of locks and deadlocks may occur. Optimistic concurrency control only locks when the transaction is about to terminate. Need to detect conflicts in this case and handle them as errors. Type-specific concurrency control allows concurrent read/write operations if they can be shown to be non-interfering.

Deadlocks: transactions are waiting for each other to release locks i.e. they are blocking each other, so none of the transactions can continue and the system is deadlocked. Deadlock detection can be timeout-based or graph-based. The latter constructs a waits-for graph that guarantees to detect all deadlocks, but is expensive to calculate.

5.1.4 | Durability

The effects of a committed transaction are never lost e.g. databases saving state to disk to be robust against machine failure.

5.1.5 | Services and participants

The transactional service is the object that encapsulates the work needed to be done in the transaction. The transactional participant is the entity that takes part in the two-phase commit and controls the outcome of the work performed by the service.

5.2 | Two-Phase Commit Optimisations

Presumed abort: tell all participants that we are aborting, there is no need to continue working if one of the participants have failed.
One-phase: no need for two-phase commit protocol in case of a single participant.
Read-only: participants that did not alter any state can be omitted from the second phase.
Last resource commit: one-phase commit resource (no prepare, only C/A) given control to commit/abort work after all normal two-phase commit resources are done.

5.3 | Synchronisations

Add synchronisation participants that flush caches and the like to improve performance of transactions.

5.4 | Heuristic Transactions

The fact that the two-phase commit protocol is blocking can block participants indefinitely at faults. A result of this might be a *heuristic outcome*: the case in which the manager informs the participant that the outcome and the decision are contrary, which has a corresponding *heuristic decision*. Two levels of heuristic interaction: participant-to-coordinator when the participant makes an autonomous decision and it is in a heuristic state, and coordinator-to-application in which the participant really did make a contrary decision and the heuristic outcome is reported to the application.
Possible heuristic outcomes are *heuristic rollback* - commit failed because participants rolled back, *heuristic commit* - abort failed because participants committed, *heuristic mixed* - some participants rolled back and some committed, and *heuristic hazard* - some results are unknown, you are now fucked.

5.5 | The Transaction Log

Need to keep track of state of transaction manager and each participant. Can be deleted when the transaction is complete.

5.6 | Failure Recovery

Since the transaction manager itself can fail we need some failure recovery system, on both the participant side and the manager side.

5.7 | Types of Transactions

So far we have considered top-level transactions, but we also have the following:

Nested transactions: enclosing transaction that consists of nested transactions. This allows for fault-isolation and modularity.
Independent top-level transactions: A top-level transaction can invoke another nested top-level transaction.
Concurrent transactions: Transactions or nested transactions may run concurrently. as the result is the same regardless.

5.8 | Distributed Transactions

5.8.1 | The Transactional Context

The context is the necessary information that flows between distributed services, and typically includes a transaction identifier, an manager endpoint address and implementation specific information e.g. support of nested transactions.

5.8.2 | Interposition

Using proxy/subordinate managers is often used in distributed systems. The subordinate manager acts as a participants to the manager, but has its own participants, which see it as a manager that executes its own two-phase commit. This can improve performance and security, as well as providing separation of concerns.

5.9 | The Eight Holy Design Patterns

We can finish up this chapter by stating the Eight Holy Design Patterns by Sverre:

* **Locking**
* **Two-phase commit protocol**
* **Transaction manager**
* **Resource manager** - in charge of delegating resources to processes.
* **Log** - all participants and transaction manager write to log. Execute log to get back to state after reset.
* **Checkpoints** - should add checkpoints of state in log to avoid it becoming too long.
* **Log manager** - It is easier said than done to add safe checkpoints that avoids the domino effect, therefore add an entity in charge of writing to the log and creating valid checkpoints.
* **Lock manager** - Releases all locks associated with a transaction, handles common lock resources and tidies up after restart.

6 | Shared-variable synchronisation

* *Ability to create (error free) multi thread programs with shared variable synchronisation.*
* *Thorough understanding of pitfalls, patterns, and standard applications of shared variable synchronisation.*
* *Understanding of synchronisation mechanisms in the context of the kernel/HW.*
* *Ability to correctly use the synchronisation mechanisms in POSIX, ADA (incl. knowledge of requeue and entry families) and Java.*
* *An understanding of how using messagebased synchronisation leads to a very different design than shared variable synchronisation.*
* *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 message-based synchronisation leads to fewer transitiondiagram states than shared variable synchronisation.*
* *Understanding the terms deadlock and livelock in context of transition diagrams.*

> *Knock knock.*
>
> *Race condition.*
>
> *Who's there?*
>
> *Albert Einstein*

6.1 | Shared-variable Synchronisation

Need inter-task communication to synchronise tasks, and this is usually based on either shared-variable synchronisation or message-passing. Message-passing involves the explicit exchange of data by means of a message and is therefore especially useful in distributed systems. In shared-variable synchronisation tasks have access to the same variables and communicate that way. This will be further studied in this chapter.

6.1.1 | Mutual Exclusion

Critical Section: the sequence of statements that must appear to be executed indivisibly.

Producer-consumer system: two or more tasks exchanging data via a finite buffer.

Since most operations are not atomic, we need mutual exclusion in the critical section.

6.1.2 | Busy Waiting

One task sets a flag called a **spin lock**, while another is **busy waiting** or **spinning**, i.e. looping and rechecking if the flag is set. This is really inefficient, and can lead to **livelocks** - an error condition where the task is looping the same work forever, and not actually doing any useful work.

6.1.3 | Suspend and Resume

One could introduce suspension of tasks instead of busy waiting to not waste all that processor time. But hold up right there, this is not atomic! We have to first check the flag and then suspend, so this introduces a data race condition:

Race condition: Fault in the design of concurrent tasks where the result is unexpected and critically dependent on the sequence or timing if accesses to shared data.

Ada does have an atomic implementation of suspension. But we need something better...

6.2 | Semaphores

A **semaphore** is a non-negative integer variable that, apart from initialisation, can only be acted upon by two atomic procedures: `wait(S)`: if the value of the semaphore S is greater than zero, decrement it by one, otherwise delay the task until S is greater than zero and decrement it.

`signal(S)`: increment the value of the semaphore S by one.

With this simple data structure mutual exclusion is implemented by simply wrapping the critical section of a task in a `wait(S) - signal(S)` block.

It seems like the wait procedure needs to perform some sort of busy waiting and wait for a signal from some other process. This is where the RTSS (run-time support system) comes in. The RTSS should keep tabs on which tasks are waiting and remove them from the processor. Eventually, if the program is correct (for example, no deadlocks), some other task will call signal and the RTSS will let the task continue.

Deadlock: a set of tasks are in a state from which it is impossible for any of them to proceed. When designing concurrent systems with semaphores it is important to consider deadlocks, they are absolutely nasty and rarely detected by testing. Can be prevented by optimistic concurrency control, allocation of all resources at once, preemption and global allocation order.

Starvation: a task is never allowed access to a resource in critical section because other tasks always gain access before them.

When utilising semaphores for mutual exclusion we usually use binary semaphores(mutexes) - either the resource is locked or it isn't.

Semaphores are simple, but are not scalable. Only really simple problems are easy to solve with semaphores. Add a few threads and a few different resources and it is very hard to guarantee **liveness** - no deadlocks, livelocks or starvation.

Semaphores are not directly supported in Ada, but can be implemented easily with the synchronisation primitives. Java has semaphore libraries. POSIX has semaphores directly implemented (as it is an Unix API).

6.3 | Conditional Critical Regions

A CCR is a section of code that is guaranteed to be executed in mutual exclusion. By programming the critical section as a CCR can we achieve mutual exclusion. We achieve this by wrapping a critical region in a guard that blocks the thread. An issue is that every time a thread exits the critical region all the waiting threads needs to wake up, reevaluate the guard and then either enter to continue to sleep.

6.4 | Monitors

Monitors try to provide more structured control regions as compared to the CCRs. A critical region is encapsulated in a single module called a monitor. All variables that need to be accessed by mutual exclusion are hidden, and all procedure calls into the module are guaranteed to execute with mutual exclusion. Uses conditional variables with `wait` and `signal`, in which `wait` (always) blocks a thread until a blocking thread releases its mutually exclusive hold on the monitor with `signal`, allowing one blocking thread to enter. If no tasks are blocking, it has no effect. The condition variable is basically a container of all the threads blocking on the condition, and when blocked a thread may temporarily give up the hold on the monitor until it is signalled.
Note that nested monitors are complicated and may have race conditions.

6.5 | Mutexes and condition variables in C/Real-Time POSIX, Ada and Java

C/Real-Time POSIX: Monitors can be implemented with mutexes. POSIX has read/write locks in addition to mutexes. They also have barriers that block all threads until a number of them have arrived at the barrier.
Ada: use guards instead of conditional variables → **protected objects**. They encapsulate data items and only allows access through protected entries which ensures mutual exclusion. Has **requeue**, which redirects flow to another entry. There are also **entry families**. Functions are read-only and can therefore be called concurrently by many tasks, but not procedures and entries.
Java: methods can be **synchronised**, which means the method can only proceed once the lock associated with the object has been obtained, hence they have mutually exclusive access to the data in the object, if that data is only accessed by synchronised methods. Conditional synchronisation is achieved with `notifyAll()`. Since `notify()` is arbitrary we cannot have different threads waiting for different things, and it is therefore recommended by the good man Sverre to always use `notifyAll()`. Also, inheritance and synchronisation does not work well together.

6.6 | Making a real time kernel

Feature 1: **preemption**: switching between threads. Allows for timer interrupt handling. Need to add a process queue.
Feature 2: **priorities**: sort the process queue by priority.
Feature 3: **synchronisation**: events, condition variables, locks, all that stuff that we have discussed.

7 | SCHEDULING

* *Be able to prove schedulability using the utilisation 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.*

A scheduling scheme provides an algorithm for ordering the use of system resources (particularly the CPUs) and a means of predicting worst-case behaviour of the system when the algorithm is applied. A scheduling scheme can be static or dynamic.

7.1 | The Cyclic Executive Approach

With a fixed set of purely periodic the simplest thing to do is to repeatedly execute a schedule that causes the task to run at their correct rate, basically a table of procedure calls which is called the **major cycle** that consists of a number of **minor cycles**. This model is too simple really, we need to be able to add tasks with long periods or sizeable computation times.

7.2 | Task-based Scheduling

Here we support task execution directly. The state of a task is either runnable, suspended waiting for timing event and suspended waiting for non-timing event.

7.2.1 | Schedulability Tests

A schedulability test is defined to be **sufficient** if a positive outcome guarantees that all deadlines are always met. It is defined to be **necessary** if failure of the test leads to a missed deadline at some point during execution. A both sufficient and necessary test is **exact**. A test is **sustainable** if it correctly predicts that a schedulable system will remain schedulable when its operational parameters improve.

7.2.2 | Preemption

In a **preemptive** scheme a low-priority task will be immediately switched to a high-priority task when it is released. In a **non-preemptive** scheme the low-priority task will be allowed to complete before the other executes. Preemption is more reactive, and therefore most often prefered. There is also **deferred preemption**, in which a low-priority task is allowed to continue, but only for a bounded amount of time.

7.2.3 | Simple Task Model

We assume the following for our simple task model used to analyse worst-case behaviour:

* Fixed set of independent, periodic tasks with known periods and fixed worst-case execution times.

* All system overhead is ignored.
* Deadlines equal the task's periods.
* No internal suspension points within a task.
* Execution is on a single CPU.

7.2.4 | Scheduling notation

B	Worst-case blocking time
C	Worst-case execution time
D	Task deadline
I	Task interference time
J	Task release jitter
N	Number of tasks
P	Task priority
R	Worst-case response time
T	Minimum time between task releases i.e. period
U	Task utilisation time (C/T)

Worst-case execution time can be analysed by decomposing code into blocks and collapsing it by finding the critical path in each block.

7.3 | Fixed-Priority Scheduling

FPS assumes that each task has a fixed, static priority which is computed before run-time. There exists a simple optimal priority assignment scheme for FPS known as **rate monotonic** priority assignment, where each task is assigned a unique priority from the ranking of its period. So the shorter the period, the higher the priority.

7.3.1 | Utilisation-based Schedulability Tests for FPS

If the following condition is true then all N tasks will meet their deadlines:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N \left(2^{\frac{1}{N}} - 1 \right) \tag{1}$$

Note that this is not exact, but *very nice* because it is simple. For large N the bound approaches 69.3%, hence any utilisation of less than this upper bound will always be schedulable.
An improved utilisation-based test:

$$\prod_{i=1}^{N} \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \tag{2}$$

7.3.2 | Response Time Analysis for FPS

Here we instead try to calculate the response time of each task, which then can be compared to the deadlines to see if the FPS is scheduable. The highest priority task will simply have that the response time equals the computations time,

while the other tasks will suffer **interference**:

$$R_i = C_i + I_i \tag{3}$$

Number of releases of higher-priority task j during R_i: $\left\lceil \frac{R_i}{T_j} \right\rceil$
Maximum interference: $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$
This forms the following recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{4}$$

When $w_i^n = w_i^{n+1}$ the solution has converged and $w_i^n = R_i$. The RTA test is exact.

7.4 | Sporadic and Aperiodic Tasks

We assume that a sporadic task will not arrive mote than once in any T_i interval. We also not assume that $D = T$ as before, since this is not reasonable for sporadic tasks. The response time algorithm from before is still applicable for $D < T$ as long as $w_i^n > D_i$.
Using worst-case figures when measuring schedulability may lead to poor processor utilisation, so we instead want that 1) all tasks should be scheduable with average execution and arrival times, and 2) all hard real-time tasks should be scheduable using worst-case execution and arrival times of all tasks. This might lead to **transient overload** - a situation in which not all tasks are able to meet the current deadlines, but the second rule ensures that this does not happen for the hard real-time tasks.
One can use servers to protect the tasking resources needed by hard tasks. A **DeferrableServer** is a new task at the highest priority, that allows aperiodic tasks to run as they arrive until the capacity is used up and it is suspended until we arrive at a new period. A **SporadicServer** replenishes slightly differently in that the server replenishes the capacity used by a soft task one period after the task arrived. This performs better, but adds more overhead. There is also **dual-priority scheduling**, where the range of priorities is split into three bands, and soft tasks execute in the middle band, and hard tasks start in the low band and move to the high band in order to meet their deadlines.
For $D = T$ the rate monotonic priority ordering was optimal for FPS. For $D < T$ we can formulate a similar optimal ordering, the **deadline monotonic** priority ordering. Here the fixed priority is inversely proportional to its relative deadline.

7.5 | Task Interactions and Blocking

Priority Inversion: a high-priority task is blocked, waiting on a low-priority task that share resources with the high-priority task.
Unbounded Priority Inversion: intermediate tasks run instead of a low-priority task that blocks a high-priority task, meaning that the high-priority task might possibly be blovked forever.
To minimise the amount of time a task is blocked because of priority inversion, **priority inheritance** is used i.e. when a high-priority task is blocked because a low-priority task have locked a shared resource, the low-priority task will inherit the priority of the high-priority task. For K critical resources in the system, the upper bound of the blocking time for task i is:

$$B_i = \sum_{k=1}^{K} \text{usage}(k, i)C(k) \tag{5}$$

where $\text{usage}(k, i) = 1$ if resource k is used by at least one task with a priority less than P_i and at least one task with priority greater or equal to P_i, otherwise 0. With blocking time taken into account we can reformulate the response time recurrence:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{6}$$

7.6 | Priority Ceiling Protocols

* A high-priority task can be blocked at most once during its execution by lower-priority tasks.
* Deadlocks are prevented.
* Transitive blocking is prevented.
* Mutual exclusive access to resources is ensured.

Mutual exclusion since a task with a lock on a resource will have the highest possible priority for this resource. No deadlocks since if a task has a resource and requests another, then the requested resource can not have a lower priority.

7.6.1 | The Original Ceiling Protocol

* Each task has a static default priority assigned.
* Each resource has a static ceiling value defined, which is the maximum priority of the tasks that use it.
* A task has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority tasks.
* A task can only look a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself).

The protocol ensures that after a first resource is locked, the second resource can only be locked if there does not exist any higher-priority task that uses both resources. This means that:

$$B_i = \max_{k=1}^{K} \text{usage}(k, i)C(k) \tag{7}$$

7.6.2 | The Immediate Ceiling Priority Protocol

* Each task has a static default priority assigned.
* Each resource has a static ceiling value defined, which is the maximum priority of the tasks that use it.
* A task has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.

This is simpler, as it only blocks tasks at the beginning of the task's execution.

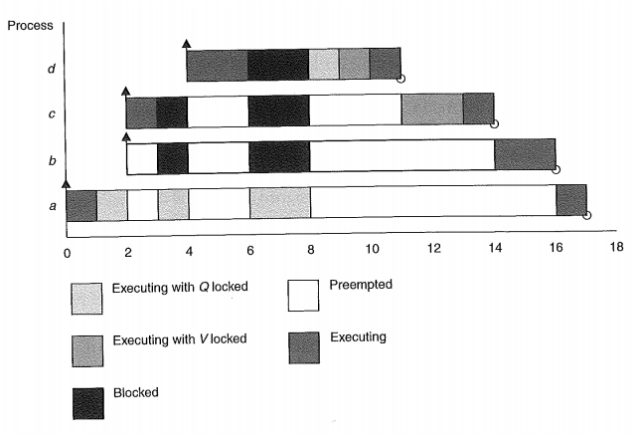


FIGURE 2 Example of OCPP

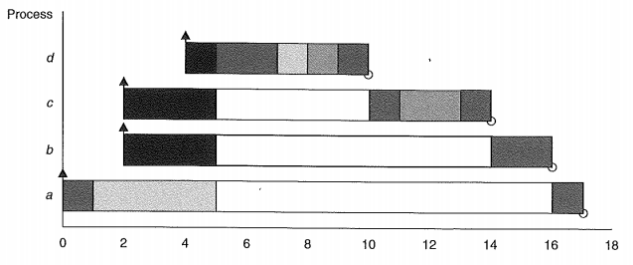


FIGURE 3 Example of ICPP

7.7 | An Extendable Model for FPS

We now consider an extended model where we allow $D < T$, sporadic and aperiodic tasks and task interactions. The maximum variation in a task's release is termed its **release jitter**. We alter the RTA:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{8}$$

For $D > T$ we need to consider q seperate windows:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \tag{9}$$

$$R_i(q) = w_i^n(q) - qT_i \tag{10}$$

Jitter is easily added to this equation. Time needed to handle errors can also easily be linearly added to the response time. If not all priorities correspond to a single task we simply need to sum over the higher- or equal priority set of tasks, not just the higher priority tasks. Overhead is added by adding cost of switching times in the RTA equation.

7.8 | Earliest Deadline First Scheduling

EDF executes the runnable tasks in the order determined by the absolute deadlines of the tasks, which is computed at run-time. The utilisation-based test is:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq 1 \tag{11}$$

FPS is more often used, as it is simpler to implement because of the static priorities and because it performs very bad if overloaded. But EDF does have higher utilisation.
When considering blocking, EDF suffers from **deadline inversion**, while FPS suffers from priority inversion. Since EDF is dynamic the ceiling protocols and inheritance is more complicated.

7.9 | Value-Based Scheduling

VBS is used when systems can become overloaded and a more adaptive scheme is required.

8 | NETWORK PROGRAMMING

I've got a really good UDP joke to tell you, but I don't know if you'll get it.

Barack Obama

Network programming is needed for data exchange, synchronisation, resource management etc.
Interprocess communication: the transfer of data among processes.
Synchronous communication: the processes synchronise at every message, so send and receive are blocking.
Asynchronous communication: Sending can proceed after send.
Remote invocation: sending proceeds after reply.
We can have direct or indirect, symmetric or asymmetric naming schemes.
Pipes: unidirectional communication. Named pipes exists as files, while normal pipes dies after the process dies.
Socket: bidirectional communication, like FTP and WWW.
Network access layer: high speed network over small region, packets routed based on MAC-address.
Internet layer: group of internetworking protocols used to transport packets across network boundaries specified by IP-address.
Transport layer: achieves data transfer between devices.
TCP: reliable, connection-oriented. The client process creates a stream socket bound to a port and makes a connection request to the server. The server process creates a listening socket bound to the server port and waits for connection request.
UDP: unreliable, connection-less, message-oriented, not as much overhead. Unicast, multicast, broadcast.

9 | CODE QUALITY

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

9.1 | Modules

9.1.1 | Abstract Data Types

* Have you thought of the classes in your program as abstract data types and evaluated their interfaces from that point of view?

9.1.2 | Abstraction

* Does the class have a central purpose?
* Is the class well named, and does its name describe its central purpose?
* Does the class's interface present a consisten abstraction?
* Does the class's interface make obvious how you should use the class?
* Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
* Are the class's services complete enough that other classes don't have to meddle with its internal data?
* Has unrelated information been moved out of the class?
* Have you thought about subdividing the class into component class, and have you subdivided it as much as you can?
* Are you preserving the integrity of the class's interface as you modify the class?

9.1.3 | Encapsulation

* Does the class minimize accessibility to its members?
* Does the class avoid exposing member data?
* Does the class hide its implementation details from other classes as much as the programming language permits?
* Does the class avoid making assumptions about its users, including its derived classes?
* Is the class independent of other classes? Is it loosely coupled?

9.1.4 | Inheritance

* Is inheritance used only to model "is a" relationships – that is, do derived classes adhere to the Liskov Substitution Principle?
* Does the class documentation describe inheritance strategy?
* Do derived classes avoid "overriding" non-overridable routines?
* Are common interfaces, data, and behavior as high as possible in the inheritance tree?
* Are inheritance trees fairly shallow?
* Are all data members in the base class private rather than protected?

9.1.5 | Other Implementation Issues

* Does the class contain about seven data members or fewer?
* Does the class minimize direct and indirect routine calls to other classes?
* Does the class collaborate with other classes only to the extent absolutely necessary?
* Is all member data initialized in the constructor?
* Is the class designed to be used as deep copies rather than shallow copies unless there's a measured reason to create shallow copies?

9.1.6 | Language-Specific Issues

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

9.2 | Routines

9.2.1 | Big-Picture Issues

* Is the reason for creating the routine sufficient?
* Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
* Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
* Does the routine's name describe everything the routine does?
* Have you established naming conventions for common operations?
* Does the routine have strong, functional cohesion – doing one and only one thing and doing it well?
* Do the routines a loose coupling – are the routine's connections to other routines small, intimate, visible, and flexible?
* Is the length of routine determined naturally by its function and logic, rather than by an artificial coding standard?

9.2.2 | 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?

9.3 | Variables Names

9.3.1 | General Naming Considerations

* Does the name fully and accurately describe what the variable represents?

* Does the name refer to the real-worls problem rather than to the programming-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*?

9.3.2 | Naming Specific Kinds Of Data

* Are loop index names meaningful (something other than *i*, *j*, or *k* if the loop is more than one or two lines long or is nested)?
* Have all "temporary" variables been renamed to something more meaningful?
* Are boolean variables named so that their meanings when they're *true* are clear?
* Do enumerated-type names include a prefix or suffix that indicates the category – for example, *Color_* for *Color_Red*, *Color_Green*, *Color_Blue*, and so on?
* Are named constants named for the abstract entities they represent rather than the numbers they refer to?

9.3.3 | 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?

9.3.4 | 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?

9.3.5 | Common Naming Problems: Have You Avoided...

* ... names that are misleading?
* ... names with similar meanings?
* ... names that are different by only one or two characters?
* ... names that sound similar?
* ... names that use numerals?
* ... names intentionally misspelled to make them shorter?
* ... names that are commonly misspelled in English?
* ... names that conflict with standard library routine names or with predefined variable names?
* ... totally arbitrary names?

* ... hard-to-read characters?

9.4 | Good commenting Technique

9.4.1 | General

* Can someone pick up the code and immediately start to understand it?
* Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
* Is the Pseudocode Programming Process used to reduce commenting time?
* Has tricky code been rewritten rather than commented?
* Are comments up to date?
* Are comments clear and correct?
* Does the commenting style allow comments to be easily modified?

9.4.2 | Statements and Paragraphs

* Does the code avoid endline comments?
* Do comments focus on *why* rather than *how*?
* Do comments prepare the reader for the code to follow?
* Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
* Are surprises documented?
* Have abbreviations beeen avoided?
* Is the distinction between major and minor comments clear?
* Is code that works around an error or undocumented feature commented?

9.4.3 | 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?

9.4.4 | 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?

9.4.5 | 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?

9.4.6 | 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 numbers in the listing?

9.5 | Self-documenting Code

9.5.1 | Classes

* Does the class's interface present a consistent abstraction?
* Is the class well named, and does its name describe its central purpose?
* Does the class's interface make obvious how you should use the class?
* Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?

9.5.2 | Routines

* Does each routine's name describe exactly what the routine does?
* Does each routine perform one well-defined task?
* Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
* Is each routine's interface obvious and clear?

9.5.3 | Data Names

* Are type names descriptive enough to help document data declarations?
* Are variables named well?
* Are variables used only for the purpose for which they're named?
* Are loop counters given more informative names than *i*, *j*, and *k*?
* Are well-named enumerated types used instead of makeshift flags or boolean variables?
* Are named constants used instead of magic numbers or magic strings?
* Do naming conventions distinguis among type names, enumerated types, named constants, local variables, class variables, and global variables?

9.5.4 | Data Organization

* Are extra variables used for clarity when needed?
* Are references to variables close together?
* Are data types simple so that they minimize complexity?
* Is complicated data accessed through abstract access routines (abstract data types)?

9.5.5 | Control

* Is the nominal path through the code clear?
* Are related statements grouped together?
* Have relatively independent groups of statements been packaged into their own routines?
* Does the normal case follow the *if* rather than the *else*?
* Are control structures simple so that they minimize complexity?
* Does each loop perform one and only one function, as a well-defined routine would?
* Is nesting minimized?
* Have boolean expressions been simplified by using additional boolean variables, boolean functions, and decision tables?

9.5.6 | Layout

* Does the program's layout show its logical structure?

9.5.7 | Design

* Is the code straightforward, and does it avoid cleverness?
* Are implementation details hidden as much as possible?
* Is the program written in terms of the problem domain as much as possible rather than in terms of computer-science or programming-language structures?