Summary of TTK4145 Real-Time Programming

Morten Fyhn Amundsen

NTNU

August 31, 2016

Contents

1 Disclaimer

These are notes from some parts of the TTK4145 curriculum. None of this is really mine. add something about two phase commit

2 Reliability and fault tolerance

something about process pairs Four sources of faults:

- Inadequate spec.
- Software design errors.
- Hardware failure.
- Interference in communication.

2.1 Reliability, failure, and faults

A fault that becomes active leads to an error which can lead to a failure.

2.2 Failure modes

- Value failure.
- Time failure.
- Arbitrary failure: Combination of the above.

Types of system failures:

- aoditd

2.3 Fault prevention and fault tolerance (2.3)

2.3.1 Fault prevention

Fault avoidance (don't create faults) and fault removal (un-create the ones you've made).

2.3.2 Fault tolerance

- Full fault tolerance: No significant degradation.
- Graceful degradation: System continues, but partially degraded during recovery.
- Fail safe: Maintains integrity, temporarily halts.

2.3.3 Redundancy

Static redundancy: Several identical components with voting.

Dynamic redundancy: The module informs somehow whether its output is erroneous.

2.4 *N*-version programming (2.4)

N independent programs for the same task are made. A driver invokes all N programs, waits for them to finish, and compares outputs.

2.4.1 Vote comparison

Easy for exact values, tricky for 'analog' values.

2.4.2 Principal issues in *N*-version programming

- Initial spec: Will mess up all N if incorrect.
- Independence of design effort: All is lost if the versions still give identical errors.
- Budget: Super costly to develop and maintain N times as much software.

2.5 Software dynamic redundancy (2.5)

2.5.1 Error detection

- Replication checks (N-version programming).
- Timing checks (watchdog, deadline).
- Reversal checks (calculate input from output and compare).
- Coding checks (checksums).
- Reasonableness checks (variable range, assertions).
- Structural checks (integrity of data structures).
- Dynamic reasonableness check (reasonable compared to prev. value).

2.5.2 Damage confinement and assessment

Modular decompositon: Confine errors by modularising and good interfaces.

Atomic actions: An activity that is completed with *no* interaction to the system (also called transactions).

2.5.3 Error recovery

Forward error recovery tries to continue by making selective corrections to the state. Pretty fast, often necessary for time-critical things.

Backward error recovery restores the system to a safe, previous state (a *recovery point*). Creating a recovery point is called *checkpointing*. Problematic if something has happened in hardware since the last recovery point. For concurrent processes, one must use *recovery lines*.

Domino effect: If a thread discovers an error and rolls back past some communication, the other side of the communication must also roll back. This can cascade backwards, rolling back way too much. To avoid this, we need a consistent state across both processes that they can roll back to. Discussed more in Section 4.

2.6 The recovery block

A normal programming language block, but with a recovery point at the entrance and an acceptance test at the exit. If the test fails, the block recovers and tries to execute an alternative module (i.e. do the thing some other way and pray that works).

2.7 Comparison between *N*-version and recovery blocks (2.7)

- Static/dynamic redundancy: N-version uses static redundancy (errors don't affect execution).
- Overhead: N-version requires a driver, recovery blocks are extra overhead by themselves.
- Spec. errors: Both are vulnerable to bad spec.
- Error detection: In difficult voting situations, acceptance testing may be more flexible.
- Atomicity: N-version avoids damage to environment because erroneous results are discarded before use. Backward error recovery does not.

Lesson: Consider using both.

2.8 Dynamic redundancy and exceptions

Exceptions and handling can be used to:

- cope with abnormal environment conditions;
- enable toleration of program design faults;
- provide general-purpose error detection and recovery.

2.8.1 Ideal fault-tolerant system components

A component that takes requests, and possibly makes further requests before yielding a response. Two possible error types:

- Interface exceptions: Illegal requests.
- Internal malfunction.

2.9 Safety, reliability, and depentability

2.9.1 Dependability

- Threats: Circumstances causing non-dependability.
- Means: Means to deliver a dependable service with the required confidence.
- Attributes: The way the quality can be judged.

3 Shared variable-based synchronization and communication

Shared variables: Objects that are accessed by several tasks. Often reasonable when tasks use a common physical memory.

Message passing: Explicit exchange of data between tasks. Can be good in systems with no common memory.

3.1 Mutual exclusion and condition synchronization (5.1)

Critical section: A sequence of statements that must be executed indivisibly.

Mutual exclusion: The synchronisation to allow it.

Condition synchronisation: When a task must wait for another task to finish something.

3.2 Busy waiting (5.2)

It'a a spin lock. If the resource is busy, the process keeps checking until it is free. Works for condition sync., not so much for mutual exclusion (difficult/complex). Repeatedly testing a variable/flag can lead to livelock.

3.3 Suspend and resume (5.3)

Similar, but a process suspends (stops running) while it waits. Requires even more complicated code to avoid race conditions, and isn't really used on its own. Ada includes a safe version of suspend and resume.

3.4 Semaphores (5.4)

`wait(S):` Wait until S is greater than zero, then decrement it by one and proceed.

`signal(S):` Increment S by one.

3.4.1 Suspended tasks

Busy waiting is bad because it wastes a lot of CPU time. We use suspension instead: `wait` on a zero semaphore invokes the RTSS[1]. The RTSS puts the task in a queue of tasks waiting on that semaphore. At some point, we will be allowed to execute (if the code is correct).

3.4.2 Implementation

The scheduler will make sure to make `wait` and `signal` *non-preemtible*: Tasks may not be interrupted while executing these operations. The RTSS may disable interrupts briefly.

3.4.3 Liveness

Liveness is when a task does not suffer from any of the following:

- Deadlock: System stuck, cannot proceed.
- Livelock: E.g. being stuck in a spin lock.
- Starvation: A task is never executed because it is never scheduled.

[1] Run-time support system.

3.4.4 Binary and quantity semaphores

A binary semaphore can be either 1 or 0. A quantity semaphore can be $0 \ldots N$ where N is some maximum value. The latter can be used to allow a precice maximum number of tasks concurrent access to a resource.

3.4.5 Semaphores in Ada

Ada does not directly support semaphores, but you can write your own pretty easily as an abstract structure.

3.4.6 Semaphores in Java

Java has several standard packages with concurrency things. Semaphores are included in one of them.

3.4.7 Semaphores in C/POSIX

The POSIX API provides counting semaphores.

3.4.8 Criticism of semaphores

Semaphores are error-prone. One error can mess up your whole program. Semaphores aren't really used now, at least not on their own.

3.5 Conditional critical regions

A *critical region* is some code that is guaranteed to be run under mutual exclusion. A *conditional critical region* is a critical region with a guard. (It must both get inside the mutual exclusion and pass the guard.)

An example is a bounded buffer. A producer can only write if a) there is room in the buffer *and* b) it gets mutually exclusive access.

3.6 Monitors

CCRs are kinda bad because they can be spread throughout the program. A *monitor* encapsulates all critical regions as procedures in a single module. All procedure calls are guaranteed to execute with mutual exclusion. The RTSS implements correct entry and exit protocols automatically. Condition synchronisation is still needed.

Condition variable: A semaphore-like thing that also uses `wait` and `signal` operators. `wait` *always* blocks. A blocked task then releases the mutually exclusive hold on the monitor. A `signal` will release one blocked task, if any. If there are none, `signal` does nothing.

3.6.1 Criticism of monitors

Good for mutual exclusion, but clumsy for condition synchronisation (must resort to low-level semaphore-like primitives). The internals of a monitor can be hard to read

3.7 Mutexes and condition variables in C/POSIX

Mutexes and condition variables can be combined to function as a monitor.

3.8 Protected objects in Ada

A protected object encapsulates data and allows only mutually exclusive access. Similar to monitors and CCRs. The data can be accessed in three ways:

- **Procedures** give read/write access.
- **Functions** give read-only access.
- **Entries** are like procedures, but must pass a boolean guard before entering.

All these are guaranteed to operate in mutual exclusion. Many functions can do their reading concurrently, but not while a procedure or entry is running.

3.9 Synchronized methods in Java

A method with the `synchronize` modifier is protected by a lock. As long all functions on a module with concurrent access are synchronised, full mutual exclusion is guaranteed. If access is only needed in part of the function, *block synchronisation* may be used.

3.9.1 Waiting and notifying

- `wait()` tells the calling thread to release the lock and sleep until it's awakened.

- `notify()` wakes one waiting thread. It does not release the lock. The awokened thread keeps waiting until the lock is available.
- `notifyAll()` is like `notify`, but wakes all sleeping threads.

It is safe, but a bit wasteful, to always check the waiting conditions and call `wait` in a loop, and always wake with `notifyAll`.

3.10 Shared memory multiprocessors

A perfect program will work both on one processor and on SMP (symmetric multiprocessor). This is never the case, and programs that seem to be goon on one CPU might be awful on SMP.

4 Atomic actions, concurrent tasks and reliability

4.1 Atomic actions

Definition: An action that—so far as other tasks are concerned—is indivisible and instantaneous.

4.1.1 Two-phase atomic actions

1. The coordinator gathers votes on whether to commit.
2. The coordinator makes a decision and notifies.

4.1.2 Atomic transactions

An atomic action that may either succeed or fail (that is, it's okay to fail in a way it cannot recover from). In case of failure, it rollbacks. A normal atomic action would make a mess in case of unrecoverable failure.

4.1.3 Requirements for atomic actions

- Well-defined boundaries: The start boundary is the location in each task where the action begins. The end boundary is equivalent for the end of the action. The side boundaries separate participants from non-participants.

- Indivisibility: Allowing communication between participants and no one else.
- Nesting: Nesting is okay as long as the inner action is fully contained by the outer.
- Concurrency: Concurrent atomic actions should be possible.

4.2 Recoverable atomic actions

4.2.1 Backward error recovery

A *conversation* is an atomic action that also has a recovery block:

- Upon entry, all tasks make recovery points.
- As with atomic actions, no communication except between participants.
- All tasks must pass an acceptance test to exit.
- If any participant fails, all participants roll back to the recovery point.

In some definitions, all tasks must enter before any may leave. Alternatively, participation can be made non-compulsory. Then a task with a deadline can leave and do something else if it needs to.

4.2.2 Forward error recovery

Main difference: Uses exceptions. If an exception occurs, it is raised in all participants. (Asynchronous exceptions.)

4.3 Asynchronous notification

Resumption: (Also called event handling.) Behaves like a software interrupt. A handler responds to the asynchronous event. This changes the flow of control temporarily: The interrupted task continues when the handler is finished.

Termination: Each task specifies a domain in which it can accept asynchronous notifications. A notification will terminate the domain. Called ATC: Asynchronous transfer of control.

5 Scheduling real-time systems

Scheduling is either *static* or *dynamic*.

5.1 The cyclic executive approach

Given a fixed set of periodic tasks you can plan the schedule in advance. A number of fixed-duration minor cycles make up a major cycle. Repeatedly running the major cycle will give proper scheduling. This approach only works for simple systems.

5.2 Task-based scheduling

5.2.1 Scheduling approaches

- Fixed-priority scheduling
- Earliest deadline first
- Value-based scheduling

5.2.2 Scheduling characteristics

- *Sufficiency:* The test can guarantee that all deadlines are met.
- *Necessity:* Failing the test will lead to missing a deadline.
- *Sustainability:* If improved circumstances do not make a previosly passed test fail.

5.2.3 (Non-) Preemtion

In a preemptive scheme, a higher-priority task will immediately be allowed to run. In nonpreemtiveness, it must wait for the lower-priority task to finish.

5.2.4 Simple task model

For analysis, we assume the program follows a simple model:

- Fixed set of tasks.
- Periodic tasks, known periods.
- Independent tasks.

- No overhead.
- Deadlines equal to periods.
- Fixed worst-case execution time.
- No internal suspension points.
- A single CPU.

5.3 Fixed-priority scheduling

Each task is assigned a unique priority based on its period. (Frequent task gets high priority).

5.4 Utilisation-based schedulability tests for FPS

If the total utilisation for a task set is below a bound, all tasks will meet their deadlines. The bound decreases when the number of tasks increases, but no lower than 69.3 %. The utilisation and bound is given by $\sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right) \leq N(2^{\frac{1}{N}} - 1)$, where C is the worst-case execution time, T is the period, and N is the number of tasks.

This test is sufficient, but not necessary.

5.4.1 Improved utilisation-based tests for FPS

Group tasks into task families within which all tasks' periods are multiples of each other. The N from the equation above instead becomes the number of families. This is better, but not sustainable.

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

5.5 Response time analsis for FPS

Starting with the highest-priority task, calculate worst-case response time of each task:

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

where C is the computation time, T is the period, $hp(i)$ is the set of higher priority tasks, and $w_i^0 = C_i$. If at some point $w_i^n = w_i^{n+1}$, then that is the worst-case response time. mention stopping criteria

5.6 Sporadic and aperiodic tasks

Assume T is the minimum interval of a task, and that the deadline D can be shorter than T. The same method as in Section 5.5 works here, but with stopping criterion $w_i^{n+1} > D_i$.

5.7 Task systems with D ¡ T

For $D < T$, *deadline monotonic* priority ordering (DMPO) is optimal.

5.8 Task interactions and blocking

Priority inversion is when poor scheduling makes an important thread wait for less important threads. Example: A low-pri thread runs first, and gets a mutex. A mid-pri preempts it. A high-pri preempts that, and wants the mutex. It suspends because it's must wait on the mutex, and the mid pri gets to continue. When the mid-pri is done, the low-pri can run again, and finish with the mutex. Only after that will the high-pri be allowed to run again. (Here it would be better to prioritise finishing the thread that had the mutex required, and running the mutex-independent mid-pri after the high-pri.)

Priority inheritance: If a high-pri needs a lower-pri to do something first, the lower-pri inherits the priority of the high-pri, so it can finish quickly.

maybe something about max blocking time and usage and whatnot

6 Priority ceiling protocols

Running a priority ceiling protocol on one processor gives:

- A high-pri can be blocked no more than once by a lower-pri.
- Deadlocks are prevented.
- Transitive blocking is prevented.
- Mutual exclusive access to resources is ensured.

6.0.1 Original ceiling priority protocol

1. Tasks have a static default priority.

2. Resources have a static ceiling value (equal to the maximum priority of any task accessing it).

3. Tasks have a dynamic priority that is the maximum of its static priority and any inherited priority (from blocking higher-pris).

4. A task can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource.

With this, a second resource can only be locked if there is no higher priority task that uses both resources. A high-priority task can only be blocked once by any lower-priority task.

6.0.2 Immediate ceiling priority protocol

1. Tasks have a static default priority.

2. Recources have a static ceiling value defined (equal to the maximum priority of any task accessing it).

3. Tasks have a dynamic priority that is the maximum of its static priority and the ceiling value of any recources it has locked.

This way a task can only be blocked in the beginning of its execution.

6.1 An extendible task model for FPS

6.1.1 Release jitter

7 Java transactions

7.1 What is a transaction?

A transaction (or atomic transaction) have these properties:

- *Atomicity:* Completes successfully or rolls back all its effects.
- *Consistency:* Produces consistent results.
- *Isolation:* Intermediate states are hidden. Transactions appear to happen serially.
- *Durability:* The effects of a committed transaction are never lost.

A *coordinator* is associated with every transaction. Informs all participants about whether they should commit or rollback.

7.2 Atomicity

A two-phase commit protocol is usually used to ensure consensus and thus atomicity. If a participant replies with 'abort' or does not reply, the whole action aborts. If all answer positive, the coordinator stores the decision and enters phase 2.