

# SW3ISU ORAL EXAM

CHEATSHEET BY MADS DITTMANN VILLADSEN



## AARHUS UNIVERSITET

### Resumé

This is a so called cheatsheet made by Mads Dittmann Villadsen, which will guide you through the important material in which you need to be able to understand and discuss during the oral exam. It mostly consists of Danish phrases to explain the different kinds of material from curriculum, but the technical terms are kept.

Mads Dittmann Villadsen  
mvmads@gmail.com

The following table contains everything you need to know to pass the exam. Do you master most, if not all of it, you're most likely guaranteed to pass.

Subject	Subtopics	Curriculum	Exercises
Programs in relation to the OS and Kernel	<ul style="list-style-type: none"> <li>Processes and threads</li> <li>Threading Model</li> <li>Process anatomy</li> <li>Virtual Memory</li> <li>Threads being executed on CPU, the associated scheduler &amp; Cache</li> </ul>	<ul style="list-style-type: none"> <li>Slides "Intro to OSs"</li> <li>Slides "Parallel Programs, Processes and Threads"</li> <li>OLA: "Anatomy of a Program in Memory" by Gustavo</li> <li>Duarte</li> <li>OLA: "The Free Lunch is Over"</li> <li>OLA: "Virtual Memory :p131-141(until AVL trees)"</li> <li>OLA: "Introduction to Operating Systems"</li> <li>OLA: "Multithreading"</li> <li>Kerrisk: "Chapter 3-3.4: System Programming Concepts"</li> <li>Kerrisk: "Chapter 29: Threads: Introduction"</li> </ul>	Posix Threads
Synchronization and protection	<ul style="list-style-type: none"> <li>Data integrity - Concurrency challenge</li> <li>Mutex &amp; Semaphore</li> <li>Mutex &amp; Conditionals</li> <li>Producer / Consumer problem</li> <li>Dinning Philosophers</li> <li>Dead locks</li> </ul>	<ul style="list-style-type: none"> <li>Slides "Thread Synchronization I &amp; II"</li> <li>Kerrisk: "Chapter 30: Thread Synchronization"</li> <li>Kerrisk: "Chapter 53: Posix Semaphores (Named not in focus for this exercise)"</li> <li>OLA: "pthread-Tutorial - chapters 4-6" .</li> <li>OLA: "Producer / Consumer problem"</li> <li>OLA "Dining Philosophers problem"</li> <li>OLA "How to use priority inheritance"</li> </ul>	Posix Threads & Thread Synchronization I & II

Thread Communication	<ul style="list-style-type: none"> <li>The challenges performing intra-process communication</li> <li>Message queue <ul style="list-style-type: none"> <li>The premises for designing it</li> <li>Various design solutions - Which one chosen and why</li> <li>Its design and implementation</li> </ul> </li> <li>Impact on design/implementation between before and after the Message Queue</li> <li>Event Driven Programming <ul style="list-style-type: none"> <li>Basic idea - The special while loop!</li> <li>Reactiveness</li> <li>Design - e.g. from sequence diagrams to code (or vice versa)</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>Slides “Inter-Thread Communication”</li> <li>OLA: “Event Driven Programming: Introduction, Tutorial, History - Pages 1-19 &amp; 30-51”</li> <li>OLA: “Programming with Threads - chapters 4 &amp; 6”</li> </ul>	Thread Communication
OS Api	<ul style="list-style-type: none"> <li>The design philosophy - Why OO and OS Api?</li> <li>Elaborate on the challenge of building it and its current design <ul style="list-style-type: none"> <li>The PIMPL / Cheshire Cat idiom - The how and why</li> <li>CPU / OS Architecture</li> </ul> </li> <li>Effect on design/implementation <ul style="list-style-type: none"> <li>MQs (Message queues) used with pthreads contra MQ used in OO OS Api.</li> <li>RAII in use</li> <li>Using Threads before and now</li> </ul> </li> <li>UML Diagrams to implementation (class and sequence) - How?</li> </ul>	<ul style="list-style-type: none"> <li>Slides: “OS Api”</li> <li>OLA: “OSAL SERNA SAC10”.</li> <li>OLA: “Specification of an OS Api”</li> <li>Kerrisk: “Chapter 35: Process Priorities and Scheduling”</li> </ul>	OS API

Message Distribution System (MDS)	<ul style="list-style-type: none"> <li>• Messaging Distribution System <ul style="list-style-type: none"> <li>◦ The Broadcasting design, irrelevant receiver(s)</li> <li>◦ Local &amp; Global IDs</li> <li>◦ Why &amp; how?</li> </ul> </li> <li>• The PostOffice <ul style="list-style-type: none"> <li>◦ The Specific receiver design</li> <li>◦ Why &amp; how?</li> </ul> </li> <li>• Decoupling achieved</li> <li>• Design considerations &amp; implementation</li> <li>• Patterns per design and in relation to the MDS and PostOffice design <ul style="list-style-type: none"> <li>◦ GoF Singleton Pattern</li> <li>◦ GoF Observer Pattern</li> <li>◦ GoF Mediator Pattern</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Slides: "A message system"</li> <li>• OLA: "GoF Singleton pattern"</li> <li>• OLA: "GoF Observer pattern"</li> <li>• OLA: "GoF Mediator pattern"</li> </ul>	The Message Distribution System
Resource handling	<ul style="list-style-type: none"> <li>• RAII - What and why?</li> <li>• Copy construction and the assignment operator</li> <li>• What is the concept behind a Counted SmartPointer?</li> <li>• What is boost::shared_ptr&lt;&gt; and how do you use it?</li> </ul>	<ul style="list-style-type: none"> <li>• Slides: "Resource Handling"</li> <li>• OLA: "RAII - Resource Acquisition Is Initialization"</li> <li>• OLA: "SmartPointer"</li> <li>• OLA: "Counted Body"</li> <li>• OLA: "boost::shared_ptr"</li> <li>• OLA: "Rule of 3"</li> </ul>	Resource Handling

Tabel 1: Basically, everything you need to know

## Indhold

Exam expectations.....	6
Programs in relation to the OS and Kernel .....	7
Kernel- VS Userspace .....	7
Kernelspace .....	7
Userspace .....	7
Threads/Tråde .....	8
Threading models .....	8
Ulemper ved Threads/Tråde.....	10
Brug af tråde i Linux.....	11
Processer.....	13
The different sections which a processes can be dissected into.....	13
States (birth, life & death) .....	13
Conext switching .....	13
Synchronization and Protection .....	14
Shared data.....	14
Eksempel med problem med delt data.....	15
Låse.....	16
Mutexes (MUTual Exclusion).....	16
Semaphores .....	17
Deadlocks.....	17
The Dining Philosophers Problem .....	18
Signalling .....	20
Spurious wakeup .....	20
RAII ift. Threads/Tråde.....	21
Eksempel .....	21
Priority inversion.....	21
Priority Inheritance Protocol (PIP) .....	22
Priority Ceiling Protocol (PCP) .....	23
Producer/Consumer problem .....	24
Thread Communication .....	25
Event Driven Programming (EDP).....	25
Event-loopet .....	25
Handleren.....	26

UML.....	27
Car Thread.....	27
GarageDoorControllerThread .....	27
Inheritance - idea and notation in C++ .....	28
Downcast .....	28
RTTI og typeid() .....	28
OS Api .....	29
OSAL .....	29
OO OS API (Object Oriented Operating System Abstraction Layer).....	30
Oprettelse af threads/tråde med API.....	30
Event-loopet.....	30
Message Distribution System (MDS) .....	32
Designs.....	32
Specific receiver .....	32
Broadcast - receiver is irrelevant.....	33
Design Patterns .....	34
Singleton .....	34
Mediator .....	36
Observer .....	37
Eksempel på Message Distribution System .....	38
Resource handling .....	40
RAII (Resource Acquisition Is Initialization) .....	40
Implementeret operator overloads .....	40
Smartpointere .....	40
Counted Smart Pointer.....	40
Rule of 3.....	42
Laboratorieøvelse (Recourse Handling) .....	43

## Exam expectations

### Expectations & Process

---

This oral exam is classical in the sense that you are to elaborate on subjects within the curriculum. However you do not get to randomly select a subject, as is the usual approach, rather we will be guiding you through one or more of them in which you will be challenged both from a theoretical and practical point of view. Do note that this is an oral exam, but both views are equally important and failing one may fail the entire exam.

This means that you will probably be presented with code snippets<sup>1</sup> that you have not seen before. Is this the case, you must be adequately versed in the curriculum to discuss the various principles, concepts and challenges. This includes being able to relate and put input perspective, the different concepts within a subject as well as between subjects.

Valid aid: an outline and lab solutions.

Note that computers are not accepted. This thus means that no slideshow is viable.

The examination will at *most* take 15 minutes, after which the examinee leaves. However since this is a pass/fail exam, you may be stopped mid sentence, if your performance is deemed to be adequate. Otherwise if the maximum allotted time is used, the normal grading will be done by censor and examiner. Upon these having reached an agreement the examinee will be presented with his/her grade.

---

<sup>1</sup>If relevant for the topic, certain functions and their signatures might be important as well. It says code, however UML diagrams are just as likely

## Programs in relation to the OS and Kernel

Operativsystemer (OS) er komplekse stykker software, der styrer computerens hardware og software ressourcer, og tilbyder fælles tjenester til computerprogrammer.

3 typer af OS:

1. **Mainframe OS** (Designet til at gøre bedst brug af specifik hardware)
2. **Desktop OS** (Designet til generelt brug)
3. **Embedded OS** (Designet til effektivitet, hastighed, m.m.)

## Kernel- VS Userspace

For at kommunikere med kernetilstand og udføre operationer, som kræver kernetilstandens privilegier, skal userspace-processer kalde kernetilstandens API'er (Application Programming Interface) gennem systemkald. Disse systemkald fungerer som grænseflader, der tillader applikationer at anmode om tjenester som filadgang, netværkskommunikation, proceskontrol osv. Operativsystemet håndterer derefter disse anmodninger på vegne af brugerrummet, sikrer at opgaver udføres sikkert og effektivt.

## Kernelspace

Kernelspace kører med **FULD** adgang til systemets hardware og hukommelse. Det omfatter kernetilstandens kode og drivere, der styrer hardwaren og giver grundlæggende tjenester til brugerrummet.

## Userspace

Programmer som tekstbehandlere, internetbrowsere eller spil, udfører de deres opgaver i brugerrummet. De har normalt **IKKE** direkte adgang til hardware. De skal gennem kernelspace for at udføre opgaver som filadgang, netværkskommunikation og enhedshåndtering. Det kan udføre opgaver som håndtering af input/output-operationer, og opererer med højere privilegier end brugerrummet.



## Threads/Tråde

En tråd er et lille set af instrukser, som CPU'en kan håndtere og udføre. Den er en del af en proces, hvilket betyder, at **en proces godt kan have flere tråde**. Tråde gør det muligt at udføre multitasking, og en enkel applikation kan dermed udføre flere forskellige processer samtidigt.

Hver proces har sit eget memory. Tråde deler dataplads, og man skal derfor tage forhold til at en anden tråds data ikke bliver slettet. En proces er en instans af en applikation. Proces A kan ikke skrive i Proces B's hukommelse.

## Threading models

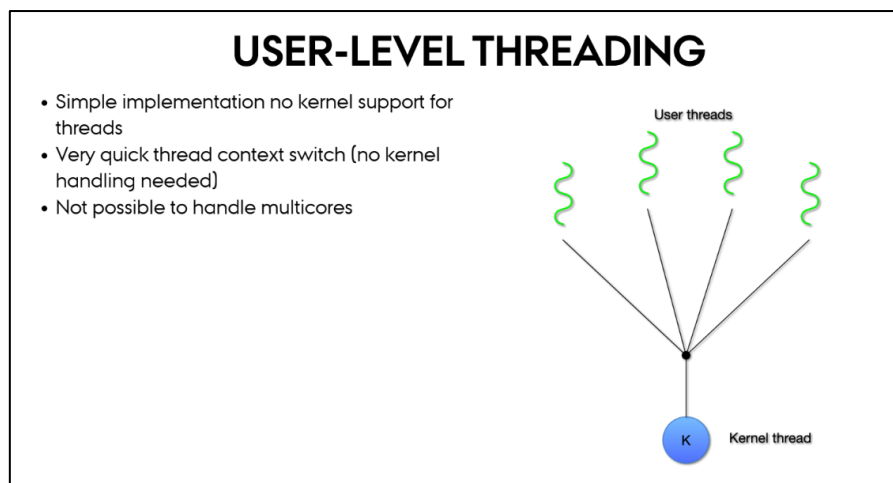
Der findes to forskellige modeller:

- **Single-threaded:** En tråd per process.
- **Multi-threaded:** Flere tråde i en enkelt proces, som deler resurser, men kører for sig selv.

### Avanceret

#### User-level Threads:

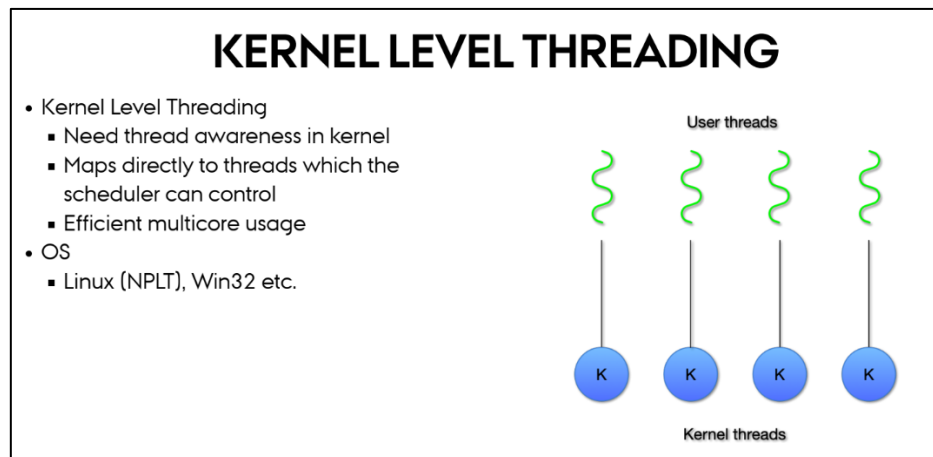
- Managed by a user-level library.
- Faster context switch since it doesn't involve kernel mode.
- The entire process can block if a thread makes a blocking system call.



Figur 2: User-level threading

#### Kernel-level Threads:

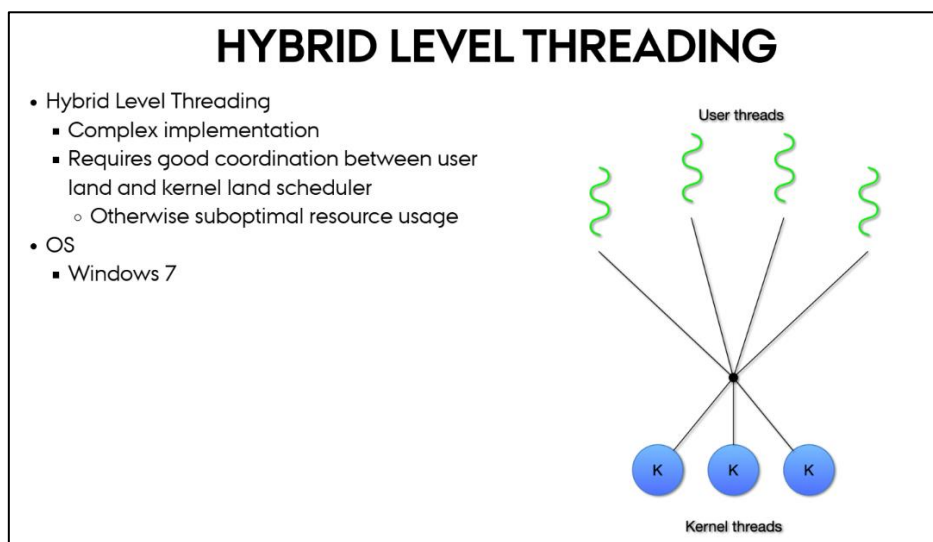
- Managed by the operating system.
- Can take advantage of multiprocessor architectures.
- Higher overhead due to system calls and context switching.



Figur 3: Kernel-level threads

### Hybrid Threads:

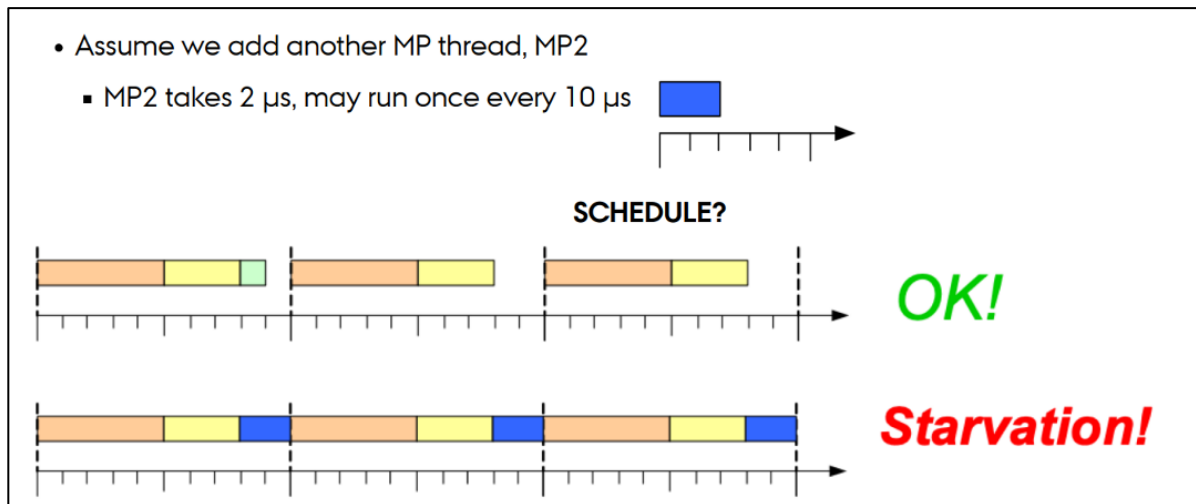
- Combines user-level and kernel-level thread management.
- Tries to balance the trade-offs between the two models.



Figur 4: Hybrid-level threads

## Ulemper ved Threads/Tråde

- Delt data kan redigeres af 2 forskellige tråde.
- **Starvation**, hvor prioriteret tråde tager alt CPU-kraft, og andre tråde dermed ikke kan komme til.



Figur 5: Eksempel på starvation

## Brug af tråde i Linux

I Linux når en tråd skal oprettes sker det med en funktionspointer eller callback-funktion. Man kalder en funktion fra en anden funktion ved at lave en pointer indeni parameteren., og derfor bruger den inde i funktionen.

### *Thread Creation*

```
int pthread_create(pthread_t *restrict tid, const pthread_attr_t *restrict tattr, void*(*start_routine)(void *), void *restrict arg);
```

1. Første parameter er en reference til int thread\_id
2. Anden parameter er til default værdier
3. Tredje parameter er funktionen den skal køre
4. Fjerde parameter er til at give data videre til en ny tråd. Kan også være NULL, hvis man ikke vil dette.

Når en tråd oprettes reserveres der noget virtuelt memory til tråden, og dens proces begynder at køre parallelt med parent-tråden.

### *Thread Termination*

```
int pthread_join(pthread_t thread, void **retval);
```

1. Første parameter er trådnævnet fra pthread\_create
2. Sættes normalt til NULL

*Simpelt eksempel med 1 thread/tråd*

```

#include <iostream>
#include <pthread.h>

void *hello(void*)
{
    std::cout << "Hello from hello function!\n";
    return NULL;
}

int main()
{
    pthread_t thread_id;
    int threadnr = pthread_create(&thread_id, NULL, hello, NULL);

    pthread_join(thread_id, NULL);

    return 0;
}

```

Figur 6: Eksempel fra laboratorieøvelse (1)

*Simpelt eksempel med 2 threads/tråde*

```

#include <iostream>
#include <pthread.h>
#include <unistd.h>

void *count(void *arg)
{
    // pthread_t ID = pthread_self(); // One way of getting the thread id, no argument in the function
    int ID = *((int*)arg);
    // pthread_t ID_self = pthread_self();
    int count = 0;
    while (count <= 10)
    {
        std::cout << "Thread ID: " << ID << std::endl;
        // std::cout << "Thread ID self " << ID_self << std::endl;
        std::cout << "Count: " << count << std::endl;
        count++;
        sleep(1);
    }

    return 0;
}

int main()
{
    std::cout << "Creating threads.\n";
    pthread_t thread_name1, thread_name2;
    int ID0 = 0;
    int ID1 = 1;
    // Creating threads with arguments
    int threadnr = pthread_create(&thread_name1, NULL, count, (void *)&ID0);
    int threadnr2 = pthread_create(&thread_name2, NULL, count, (void *)&ID1);

    // Creating threads without arguments
    // int threadnr = pthread_create(&thread_id, NULL, count, NULL);
    // int threadnr2 = pthread_create(&thread_id, NULL, count, NULL);

    std::cout << "Waiting for threads to finish.\n";

    pthread_join(thread_name1, NULL);
    pthread_join(thread_name2, NULL);
    std::cout << "Threads finished.\n";

    return 0;
}

```

Figur 7: Eksempel fra laboratorieøvelse (2)

## Processor

### The different sections which a processes can be dissected into

- **Code (Text) Segment:** Contains the executable code.
- **Data Segment:** Contains global and static variables.
- **Heap:** Dynamically allocated memory during runtime.
- **Stack:** Contains function parameters, return addresses, and local variables.

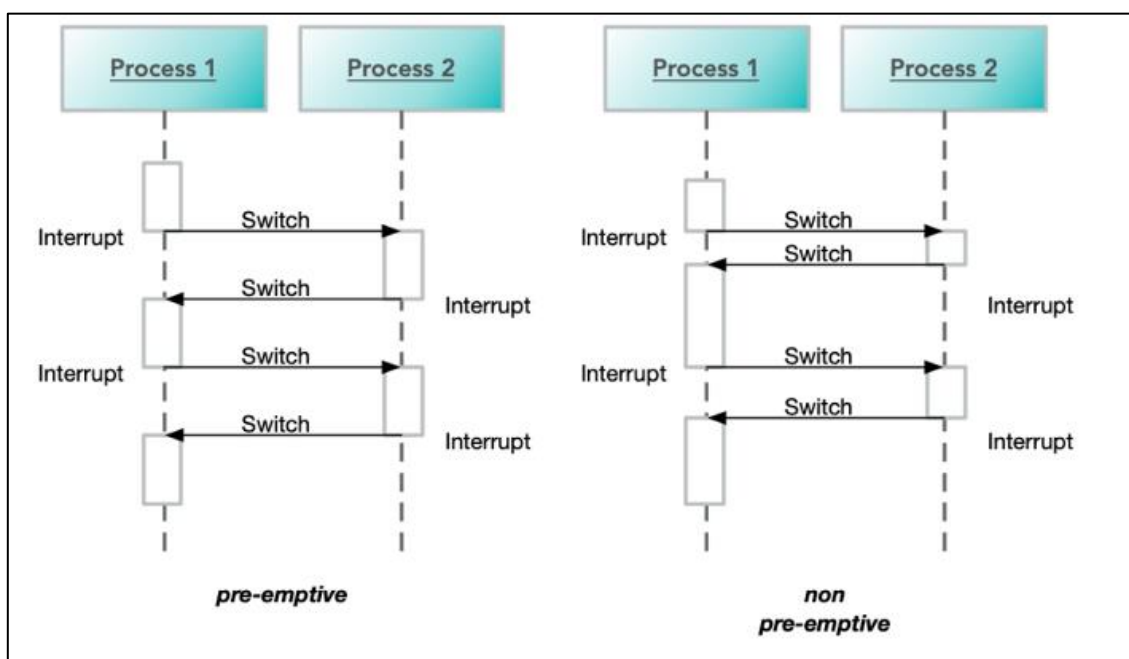
### States (birth, life & death)

- **New:** The process is being created.
- **Ready:** The process is ready to run but waiting for CPU time.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event (e.g., I/O operation).
- **Terminated:** The process has finished execution.

### Conext switching

**Context switch** udføres af OS, hvor en kørende proces pauses, og en anden proces fortsættes, hvor den senere kommer tilbage til processen, som blev pauset. Der er 2 måder, hvorpå den gør dette:

1. Preemptive scheduling: Tasks can be interrupted at any time.
2. Non-preemptive scheduling: Tasks voluntarily yield the CPU



Figur 8: Pre-emptive scheduling og Non Pre-emptive scheduling

# Synchronization and Protection

## Shared data

Shared data (delt data) er data, som flere forskellige variabler/funktioner/m.m. har adgang til, og som fælles kan ændre i.

Ved at have flere tråde, som deler samme data, kan der ske det, at en tråd kan ændre i noget data, som en anden tråd også ejer, hvilket kan skabe problemer. Hvis nu det er noget data, som den ene tråd holder styr på, så nytter det ikke noget, at en anden tråd ændrer i det data, da der så ikke er styr på det længere.

Enten kan man lade vær med at bruge delt data, eller også kan man gøre brug af såkaldte "**Critical Zones**", hvor man enten bruger **Mutexes** (simpel) eller **Semaphores** (advanceret). Disse er med til, at når en tråd arbejder med noget data, så kan en anden tråd ikke gå ind og ændre i det.

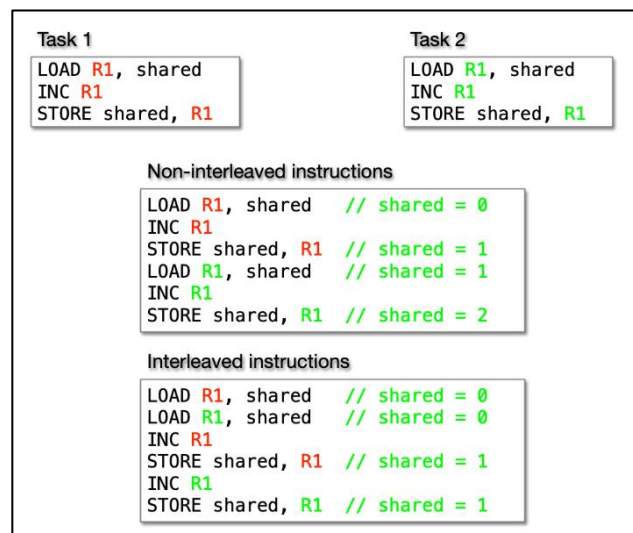
## Eksempel med problem med delt data

```

1 unsigned int shared;
2 void taskfunc()
3 {
4     for(;;)
5     {
6         shared++;           // Increment i, then wait
7         sleep(ONE_SECOND); // 1 second
8     }
9 }
10
11 int main()
12 {
13     shared = 0;
14     createThread(taskFunc); // Start two identical threads
15     createThread(taskFunc); // that run the same function
16     for(;;) sleep(ONE_SECOND); // 1 second
17 }

```

Figur 9: Shared Data Problem (1)



Figur 10: Shared Data Problem (2)



## Låse

Forskellige metoder for synkronisering:

Sync method	Behaviour
Mutex	$s=0$ or $s=1$ , belongs to one thread at a time
Conditionals	Signaling facility used with a mutex
Read/writable locks	Multiple readers - Exclusive writer
Counting semaphore	$s \geq 0$ , shared among threads
Binary semaphore	$s=0$ or $s=1$ , shared among threads

Figur 11: Forskellige metoder for synkronisering

## Mutexes (MUTual Exclusion)

Mutexes er enklere og typisk brugt til scenarier, hvor du skal beskytte en kritisk kodeafsnit eller en enkelt delt ressource.

Mutex gør brug af:

- `pthread_mutex_lock(Mutex &m)`
- `pthread_mutex_unlock(Mutex &m)`

**Lock()** kaldes først, **unlock()** til sidst. Imellem disse 2 punkter er der, hvor den kritiske zone finder sted. Når **m==0**, så er tråden blokeret indtil **m==1**.

```
1 lock(Mutex m)
2 {
3     wait until m==1, then m=0; /* ATOMIC operation */
4 }
```

```
1 unlock(Mutex m)
2 {
3     m=1; /* ATOMIC operation */
4 }
```

- If  $m==0$ , calling thread is BLOCKED until  $m==1$
- If  $m==1$ , calling thread proceeds
- Now  $m==1$  so a BLOCKED thread is made READY

Figur 12: Mutexes syntax

## Semaphores

Semaphores tilbyder mere fleksibilitet og er velegnede til scenarier, der involverer synkronisering af flere ressourcer eller signalering mellem tråde.

Semaphores gør brug af :

- **sem\_init**
- **take(s)** (A.K.A. `sem_wait(&s)...`)
- **release(s)** (A.K.A. `sem_post(&s)`)

**sem\_wait()** kaldes først, **sem\_post()** til sidst. Imellem disse 2 punkter er der, hvor den kritiske zone finder sted. Når  $s < 0$ , så er tråden blokeret indtil **sem\_post()** kaldes, hvor  $s$  vil blive inkrementeret. Derfor er det **VIGTIGT ALTID** at kalde `sem_init` med  $s > 0$ .

```

1 take(Semaphore s)
2 {
3     wait until s>0, then s=s-1; /* ATOMIC operation */
4 }

1 release(Semaphore s)
2 {
3     s=s+1; /* ATOMIC operation */
4 }
```

- If  $s == 0$ , calling thread is BLOCKED until  $s > 0$
- If  $s > 0$ , calling thread proceeds
- Now  $s > 0$  so a BLOCKED thread is made READY

Figur 13: Semaphores syntax

## Deadlocks

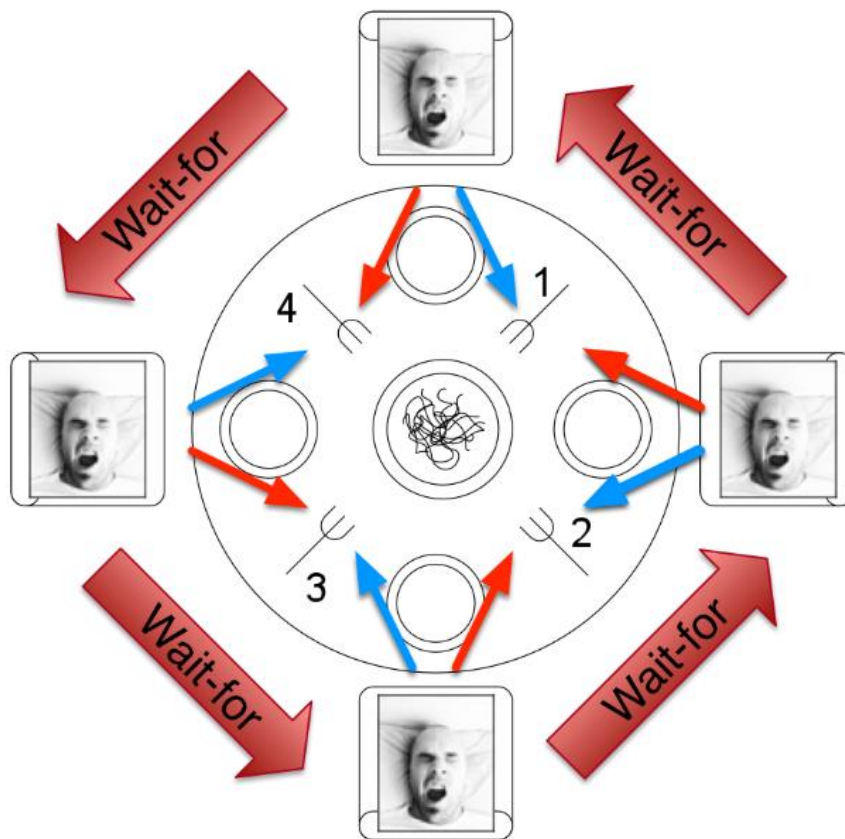
Hvis man glemmer den enten at benytte sig af `lock()` eller `unlock()` kan det føre til deadlocks. Hvis andre tråde venter på en resurse, bliver fri, men dette aldrig sker, vil de stå og vente uendeligt.

**ALDRIG** lås en tråd inde i et loop eller hvor der er en sleep. Det gør, at den ene tråd kan blokere den anden i et stykke tid, eller i værste tilfælde uendeligt. Man skal låse kritiske sektioner, hvor vigtig kode finder sted, som f.eks. kan være data, som ikke må ændres i samtidigt.

**Which conditions need to be present for a deadlock to occur?**

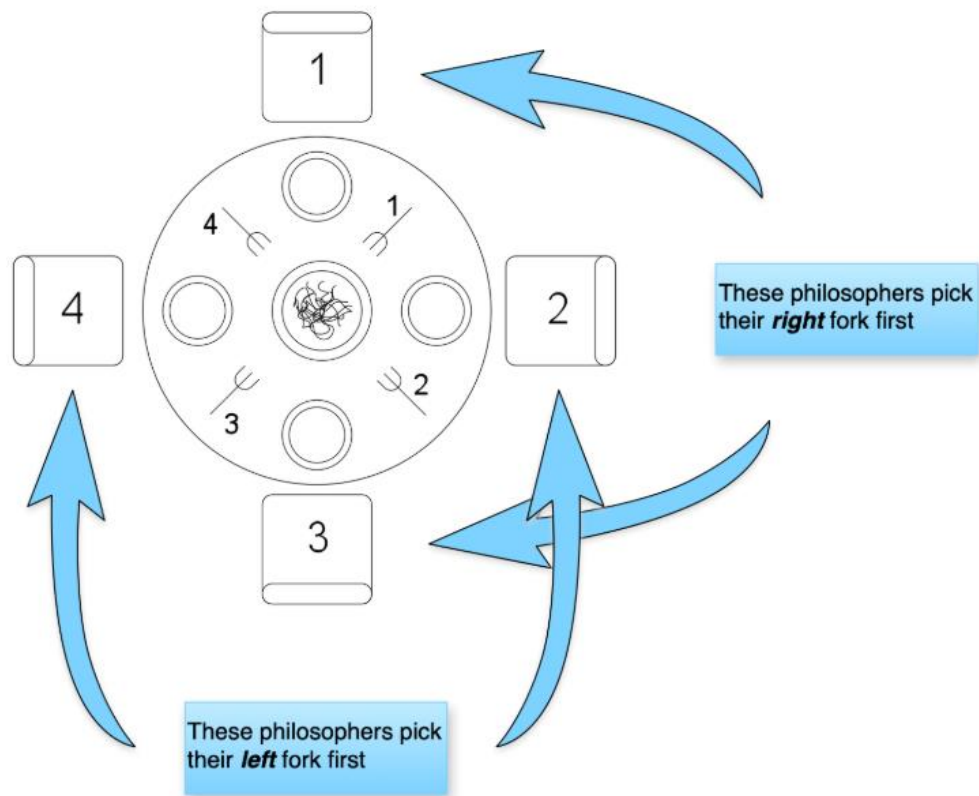
1. Mutual exclusion (The resource can only be held by one process at a time)
2. Hold-and-wait (Process already holding resources may request other resources)
3. No preemption (No resource can be forcibly removed from its owner process)
4. Circular wait condition (A cycle  $p_0, p_1, \dots, p_n, p_0$  exists where  $p_i$  waits for a resource that  $p_{i+1}$  holds)

## The Dining Philosophers Problem



Figur 14: The Dining Philosophers Problem

Problemet her er, at filosof 1 starter med at tage venstre gaffel. Det gør filosof 2, 3 og 4 så også. Når filosof 1 så skal til at tage sin højre gaffel, så skal han vente til, den bliver ledig, da filosof 4 har taget den. Samme med filosof 2, 3 og 4. Derfor vil der aldrig kunne blive spist med mere en en gaffel, og derfor sker en deadlock.



Figur 15: Solving the problem with Circular Wait Condition

Her starter filosof 1 og 3 med at tage deres højre gaffel først, og filosof 2 og 4 tager deres venstre først. Filosof 3 og 4 skal vente på filosof 1 og 2 ligger gaffel 4 og 2 ned før de kan begynde.

## Signalling

Man kan gøre brug af såkaldte conditionalvariables. Her en sender og en receiver. Man gør brug af et while-loop, som hele tiden checker variablen. Se eksemplet:

<pre> 1 carDriverThread() 2 { 3     driveUpToGarageDoor(); 4     lock(mut); 5     carWaiting = true; 6     condSignal(entry); 7 8     while(!garageDoorOpen) 9         condWait(entry, mut); 10 11    driveIntoGarage(); 12    carWaiting = false; 13    condSignal(entry); 14    unlock(mut); 15 }</pre>	<pre> 1 garageDoorControllerThread() 2 { 3     lock(mut); 4     while(!carWaiting) 5         condWait(entry, mut); 6 7     openGarageDoor(); 8     garageDoorOpen = true; 9     condSignal(entry); 10    while(carWaiting) 11        condWait(entry, mut); 12 13    closeGarageDoor(); 14    garageDoorOpen = false; 15    unlock(mut); 16 }</pre>
---	--

- This works!
  - 2-way synchronization
  - All waits are matched with signals

Figur 16: Eksempel på conditionalvariables

Conditional points er implementeret, så de 2 tråde arbejder synkront med hinanden.

1. Garagedøren låser en mutex, og da der ikke er en bil endnu, bruges condWait til at afvente inde i det specielle while-loop.
2. Først kører bilen op til garagedøren, mutex låses, carWaiting sættes til true, et conditional signal sendes til garagedøren, og da garagedøren ikke er åbent endnu bruges while-loopet.
3. Garagedøren vækkes, den åbnes, sættes til true, condSignal sendes, og da carWaiting er true, så rammes andet condWait while-loop.
4. Bilen kører ind i garagen, carWaiting sættes til false, condSignal sendes, og mutex unlockes.
5. Da carWaiting nu er false, går garagedøren videre til at lukke, sætter sin værdi til false, og unlocker sin mutex.

## Spurious wakeup

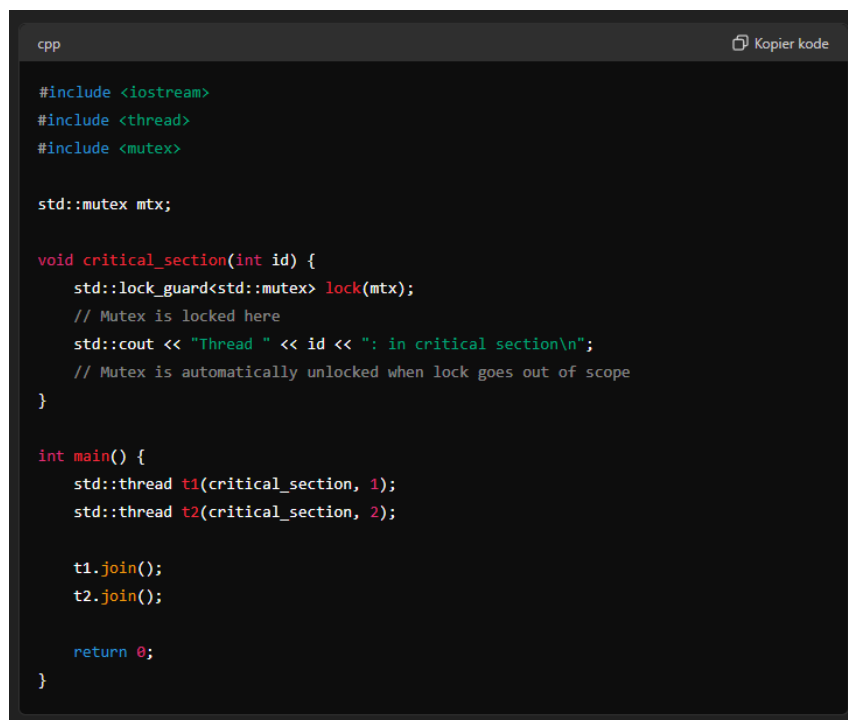
Spurious wakeup er, når en tråd venter på en conditionalvariabel, men vågner uden grund, hvilket kan give problemer. Måden at løse det på er med while-loopet, som gennemtjekker conditionen efter tråden vågner.

## RAII ift. Threads/Tråde

RAII bruge stil at koble resurser f.eks. mutexes, til en livscyklus af et objekt, som sikrer at der sker korrekt cleanup, når objektet går ud af scope.

- Lock() -> Constructor
- Unlock() -> Destructor

## Eksempel



```
cpp Kopier kode  
  
#include <iostream>  
#include <thread>  
#include <mutex>  
  
std::mutex mtx;  
  
void critical_section(int id) {  
    std::lock_guard<std::mutex> lock(mtx);  
    // Mutex is locked here  
    std::cout << "Thread " << id << ": in critical section\n";  
    // Mutex is automatically unlocked when lock goes out of scope  
}  
  
int main() {  
    std::thread t1(critical_section, 1);  
    std::thread t2(critical_section, 2);  
  
    t1.join();  
    t2.join();  
  
    return 0;  
}
```

Figur 17: Eksempel på RAII

## Priority inversion

Når en højere-prioriteret opgave venter på, at en lavere-prioriteret opgave frigiver en ressource (f.eks. en mutex eller en semafor), som den lavere-prioriterede opgave holder. Hvis en mellemprioriteret opgave, der ikke behøver ressourcen, afbryder den lavere-prioriterede opgave, bliver den højere-prioriterede opgave indirekte blokeret af den mellemprioriterede opgave. Denne inversion af prioriteter kan forårsage betydelige forsinkelser og reducere systemets reeltidsydelse.

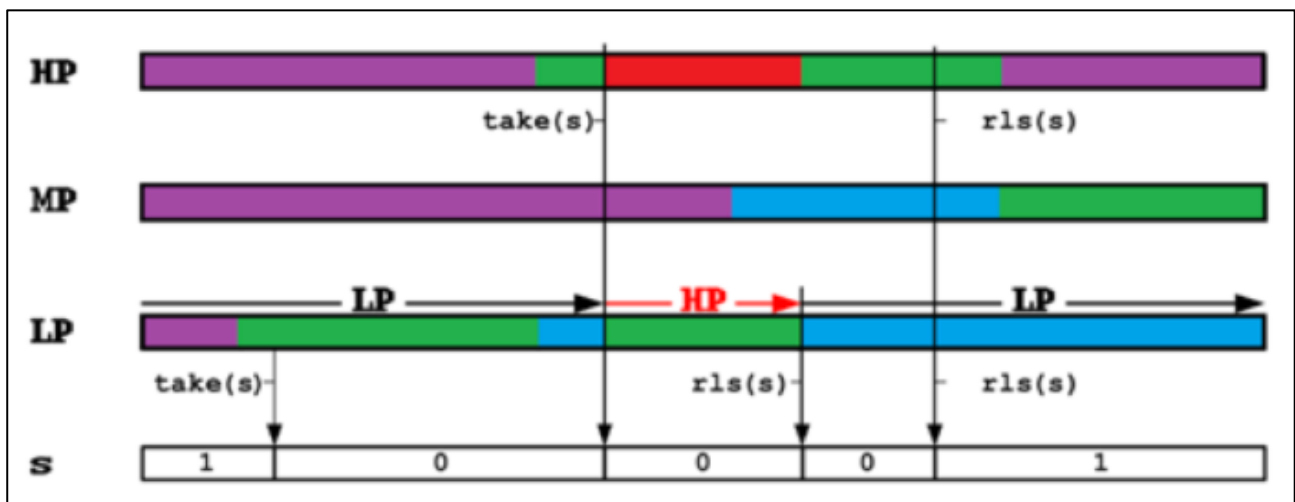
Der findes 2 løsninger:

1. Priority Inheritance Protocol (PIP)
2. Priority Ceiling Protocol (PCP)

Virker **KUN** med Mutexes

## Priority Inheritance Protocol (PIP)

Ved **Priority Inheritance Protocol (PIP)**, når en lavere-prioriteret opgave, der holder en ressource, som en højere-prioriteret opgave har brug for, bliver afbrudt, hæves dens prioritet midlertidigt til niveauet af den højeste-prioriterede opgave, der venter på ressourcen. Denne forhøjede prioritet sikrer, at den lavere-prioriterede opgave kan køre til ende og frigive ressourcen så hurtigt som muligt, hvilket minimerer ventetiden for den højere-prioriterede opgave. **Konsekvensen er**, at det kan forårsage kædeblokering, hvis flere opgaver med forskellige prioriteter venter på flere ressourcer.

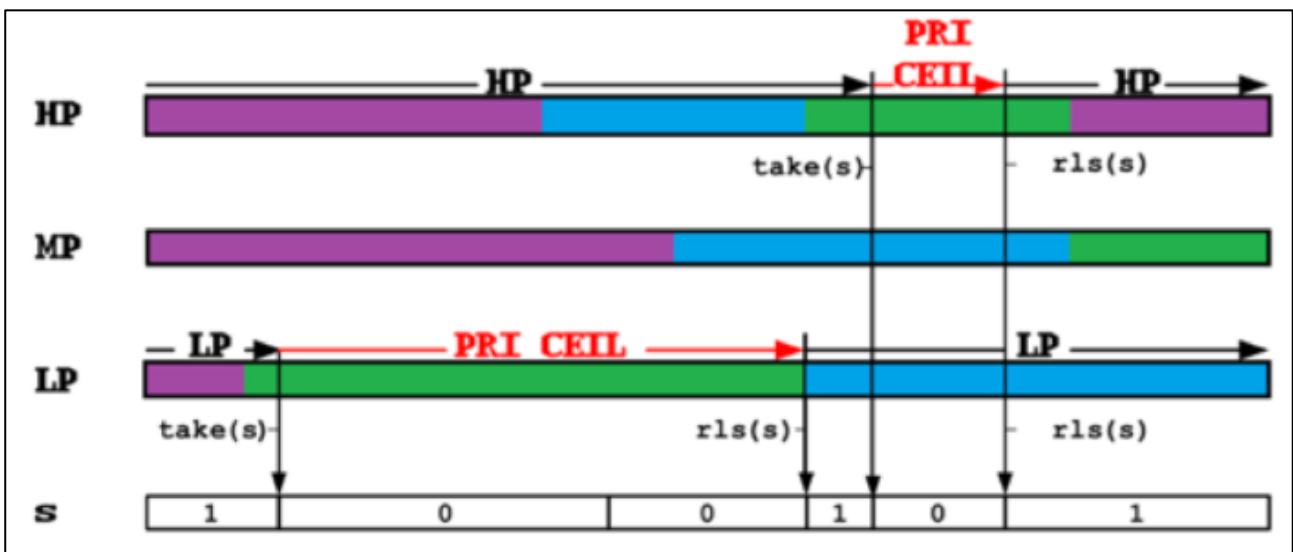


Figur 18: Priority Inheritance Protocol (PIP)

1. LP runs
2. LP acquires mutex
3. HP is prioritized to run, LP on waiting queue (WQ)
4. HP blocked due to mutex taken
5. LP runs until mutex release, but with HP priority (inheritance)
6. MP wants to run but due to lower priority -> WQ
7. HP acquires mutex and runs until done, MP & LP on WQ
8. MP runs until done, LP on WQ
9. LP runs until done

## Priority Ceiling Protocol (PCP)

Ved **Priority Ceiling Protocol (PCP)** tildeles hver ressource et prioritetloft, som er den højeste prioritet af enhver opgave, der kan låse ressourcen. En opgave kan kun låse en ressource, hvis dens prioritet er højere end prioritetlofterne for alle ressourcer, der i øjeblikket er låst af andre opgaver. Når en opgave låser en ressource, hæves dens prioritet straks til prioritetloftet for den pågældende ressource. **Konsekvensen er**, at det kan resultere i lavere samlet systemudnyttelse, da opgaver måske skal vente længere på at få ressourcer på grund af prioritetloftbegrænsningen

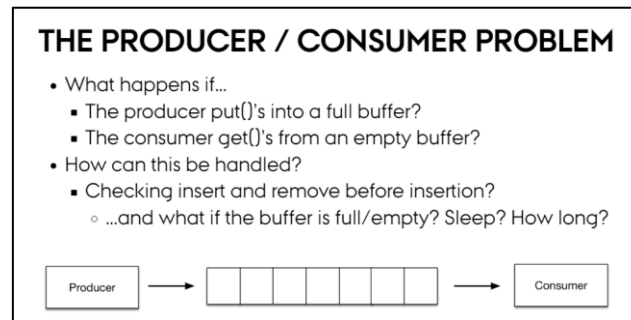


Figur 19: Priority Ceiling Protocol (PCP)

1. LP runs
2. LP acquires mutex - its priority is elevated to high priority - priority ceiling
3. HP wants to run but has lower priority -> waiting queue (WQ)
4. MP wants to run but has lower priority -> WQ
5. LP releases mutex and changes priority to low
6. HP acquires mutex and runs until done, MP & LP on WQ
7. MP runs until done, LP on WQ
8. LP run until done



## Producer/Consumer problem



Figur 20: Producer/Consumer problem (1)

```

1  class Buffer
2  {
3  public:
4      Buffer(size_t bufferSize) :
5          buffer_(new uint8_t[bufferSize]),
6          bufferSize_(bufferSize),
7          insert_(0), remove_(0)
8      {
9          emptySlotsLeftSem_ = createCountingSem(bufferSize_);
10         usedSlotsLeftSem_ = createCountingSem(0);
11     }
12
13     void put(uint8_t x) {
14         take(emptySlotsLeftSem_);
15         buffer_[insert_] = x;
16         insert_ = (insert_+1)%bufferSize_;
17         release(usedSlotsLeftSem_);
18     }
19
20     uint8_t get() {
21         take(usedSlotsLeftSem_);
22         uint8_t tmp = buffer_[remove_];
23         remove_ = (remove_+1)%bufferSize_;
24         release(emptySlotsLeftSem_);
25     }
26
27 private:
28     uint8_t* buffer_;
29     size_t   bufferSize_, insert_, remove_;
30     SEM_ID   emptySlotsLeftSem_;
31     SEM_ID   usedSlotsLeftSem_;
32 };

```

Figur 21: Producer/Consumer problem (2)

I eksemplet bliver 2 Semaphores oprettet. En til at holde styr på antallet af pladser, som er tilbage, og en til at holde styr på antallet af pladser, som er brugt. Hvis vi tager udgangspunkt i problemet, så er der 7 pladser i bufferen. Derfor bliver `emptySlotsLeftSem_` sat til 7.

I `void put()` tages en Semaphore fra `emptySlotsLeftSem_`, så den er nu nede på 6, og der "releases" til `usedSlotsLeftSem_`, som nu er på 1. Det betyder, at kører man `get()`-funktionen, vil man nu kunne bruge "take" på `usedSlotsLeftSem_` da den indeholder den tidligere "released" Semaphore. Der vil derfor kunne blive holdt styr på bufferen. Om den er fuld, tom, osv.

# Thread Communication

## Event Driven Programming (EDP)

Eventdrevet programmering er programmering, hvor man arbejder med events og beskeder. Hver besked bliver håndteret af en speciel handler-funktion. Eksempler kan være, når en temperaturafmåling bliver for høj, så skal systemet nedkøle, eller når en knap på en GUI trykkes, så skal der ske noget. Man gør brug af en "message-queue", hvor beskeder kommer ind, og bliver håndteret.

Et event er noget, som sker. Dette kan være at trykke en knap, gå forbi en sensor, eller noget tredje. Et signal kan være vores conditional variabel, som ændres, når der sker et event. Beskeden er, hvad der er sket, hvilket sendes gennem en message queue.

Eventsne bliver sendt videre gennem en "queue" af en tråd, som en anden tråd så håndterer.

## Event-loopet

Et event-loop er **MEGET VIGTIG**, da den sørger for, der hele tiden er mulighed for modtagelse af events. Dette loop er inde i selve tråden. F.eks. bilen, eller GarageDoorControlleren.

**Concept: "Only one call to receive() and one call to handle() in the loop"**

```
17
18 void* car(void*) {
19     for(;;) {
20         unsigned long id;
21         Message* msg = carMq.receive(id);
22         carHandler(id, msg);
23         delete(msg);
24     }
25 }
```

Figur 22: Eksempel for event-loop for GarageDoorControlleren fra laboratorieøvelsen

**Recieve, handle, delete.** Først bruges carMq.recieve(id) til at se, hvilken besked modtages. **CarMq er et objekt af Message.** Derefter bruges dette i carHandler, som står for at "handle" ud fra hvilket id modtages som parameter. Til sidst delete, da modtager ved, hvornår processen er udført, ikke senderen.

**Generelt gælder:** Delete(msg): == "Sender opretter (allokerer memory) and modtageren destruerer (deletes memory)"

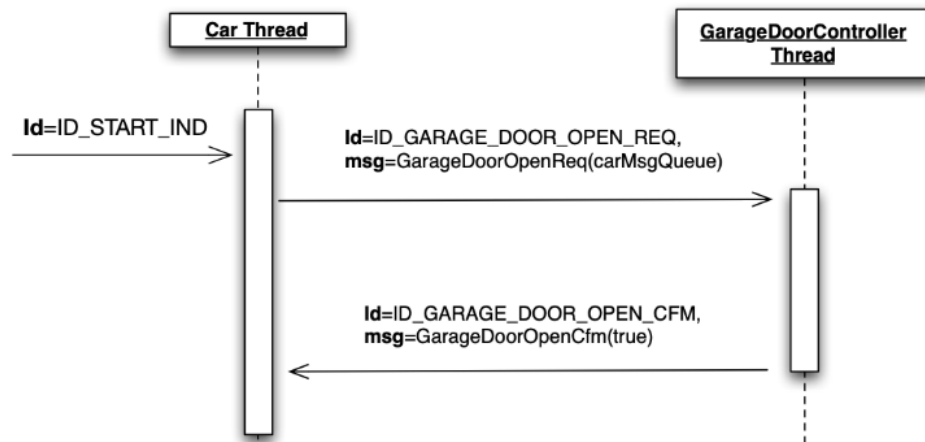
## Handleren

Handleren er den, som håndterer events ud fra ID. Den bruger "switch" cases, og kører funktioner ud fra id, som er modtaget.

```
6 void carHandler(unsigned id, Message* msg) {  
7     switch(id) {  
8         case ID_START_IND:  
9             carHandleIdStartInd();  
10            break;  
11            case ID_GARAGE_DOOR_OPEN_CFM:  
12                carHandleIdGarageDoorOpenCfm(  
13                    static_cast<GarageDoorOpenCfm*>(msg));  
14                break;  
15            }  
16 }
```

Figur 23: Eksempel for handleren carHandler fra laboratorieøvelsen

## UML



Figur 24: Eksempel fra slides

3 typer af signaler:

1. Request - Named XXXReq (a request requires a confirm)
2. Confirm - Named XXXCfm
3. Indication - Named XXXInd (purely one-way)

### Car Thread

Incoming messages are, that need to be handled:

- ID\_START\_IND
- ID\_GARAGE\_DOOR\_OPEN\_CFM

Messages sent

- ID\_GARAGE\_DOOR\_OPEN\_REQ

### GarageDoorControllerThread

Incoming messages are, that need to be handled:

- ID\_GARAGE\_DOOR\_OPEN\_REQ

Messages sent

- ID\_GARAGE\_DOOR\_OPEN\_CFM

## Inheritance - idea and notation in C++

Arv i C++ er, når en klasse får noget information (arver) fra en anden. I dette koncept har vi beskeden, som ejer fra en klasse, hvilket gør det muligt at vide, hvad beskeden er.

## Downcast

Et downcast er, når man ved hjælp af arv, ejer fra en anden klasse. F.eks. `GarageDoorOpenReq`, som arver fra `Message`, da man ved, den er en del af den queue.

## RTTI og `typeid()`

Samlet set er RTTI og `typeid()` vigtige værktøjer i C++, der muliggør dynamisk opdagelse og håndtering af objekttyper, hvilket er nyttigt i situationer, hvor polymorfisme og dynamisk casting er involveret.

# DETERMINING REAL MESSAGE TYPE

- How do we convert a `Message*` to a `GarageOpenDoorReq*`?
  - Via using `dynamic_cast<>`

```
1 GarageDoorOpenReq gdor;  
2 Message* msg_ = &gdor; // Illustration!  
3  
4 GarageDoorOpenReq* req = dynamic_cast<GarageDoorOpenReq*>(msg_);  
5 // Runtime check, req == NULL if not correct
```

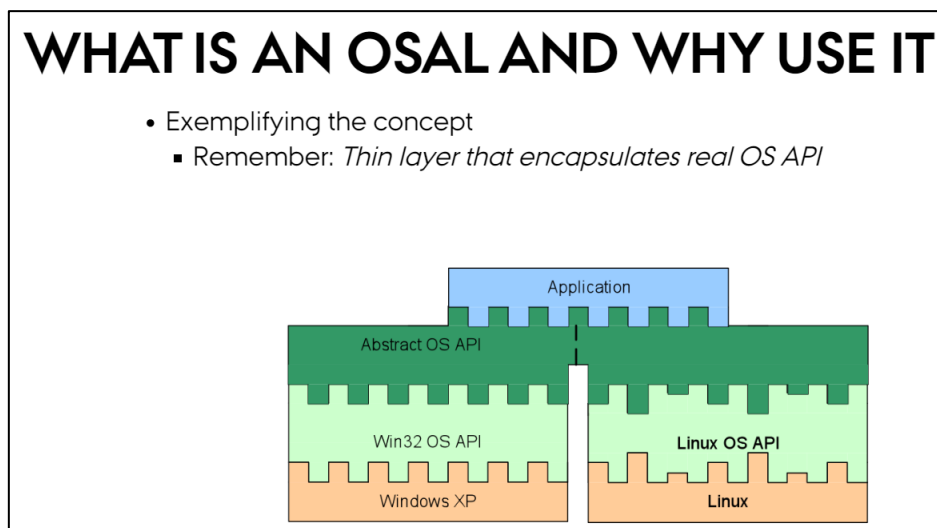
Figur 25: Konvertering af besked ved hjælp af `dynamic_cast<>`

## OS Api

**API** står for Application Programming Interface, og er et software interface for andre typer af software. Et **Framework** er et eller flere slags blueprint for, hvilke programmer kan og skal blive bygget, og kan hænge sammen. Det er en grundlæggende foundation for udvikling af applikationer indenfor det specifikke.

## OSAL

**OSAL** står for Operating System Abstraction Layer, og er en API for et abstrakt operativsystem, som gør det nemmere at udvikle kode for forskellige software og hardware platforme. F.eks. bliver tråde, mutex'es, osv. oprettet på forskellige måder på Linux vs Windows.



Figur 26: Eksempel fra slides (1)

## CREATING THREADS IN DIFFERENT OSES

```
1 //win32
2 HANDLE CreateThread(...);
```

```
1 //POSIX - Linux
2 void* pthread_create(...);
```

```
1 //VxWorks
2 void* pthread_create(...);
```

```
1 //FreeRTOS
2 portBASE_TYPE xTaskCreate(...);
```

Figur 27: Eksempel fra slides (2)

Hvert styresystem har sin egen mappe, hvor de forskellige metoder er lavet for de forskellige styresystemer. Disse bliver kaldt gennem forskellige. Et eksempel på dette ses i implementeringen af Semaphore-klassen i Windows-versionen af OS API. I Semaphore-implementationsfilen, der bruger windows.h, returneres en pointer med al informationen, som er blevet hentet, og sendes til det abstrakte lag, som vi har adgang til. Ved hjælp af denne pointer kan de Semaphore-funktioner, der er ens på tværs af alle OS-versioner af API'et, kaldes. Fordelene ved dette er, at det reducerer antallet af nødvendige headers, og det gør kompileringen hurtigere, da den ikke er direkte afhængig af windows.h, kun af det interface, der er adgang til.

## OO OS API (Object Oriented Operating System Abstraction Layer)

### Oprettelse af threads/tråde med API

1. Du laver en klasse, som arver fra OSAPI's ThreadFunctor. Den virtuelle run()-metode bliver implementeret. I main bliver en tråd lavet med klassen eks. "Mythread T;". Bagefter kaldes OSAPI::Thread Tråd(&T). Tråd.start og Tråd.join bruges derefter.
2. Event loopet (Run()) bliver implementeret i ThreadFunctor som pure virtual, og skal derefter implementeres i alle klasser, som arver. Metoden Threadmapper() kalder Run().
2. Eventsne bliver placeret i run()-funktionen.
3. En msgQueue klasse skal laves, og hver tråd der vil modtage beskeder skal "eje" en.

### Event-loopet

1. **Recieve** (Modtag)
2. **Handle** (Håndter)
3. **Delete** (Slet)

## Syntax

- Classic event loop
  - Receive
  - Handle
  - Delete

```

1 // LogSystem.cpp
2 #include <iostream>
3 #include <osapi/example/LogSystem.hpp>
4
5 void LogSystem::writeToLog(LogInd* l)
6 {
7     lf_ << l->text << std::endl;
8 }
9
10 void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
11 {
12     switch(id)
13     {
14         case ID_LOG_IND:
15             writeToLog(static_cast<LogInd*>(msg));
16             break;
17
18         default:
19             std::cout << "Unknown event..." << std::endl;
20     }
21 }
22
23 void LogSystem::run()
24 {
25     for(;;)
26     {
27         unsigned long id;
28         osapi::Message* msg = mq_.receive(id);
29         handleMsg(id, msg);
30         delete msg;
31     }
32 }

```

## Handle Message dispatcher

En switch-case, som kaldes ud fra specifikt ID

## Syntax

- Handler (dispatcher)
  - Cases out on the various ids
    - only one here
  - Handler placed in its own method writeToLog()

```

6 {
7     lf_ << l->text << std::endl;
8 }
9
10 void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
11 {
12     switch(id)
13     {
14         case ID_LOG_IND:
15             writeToLog(static_cast<LogInd*>(msg));
16             break;
17
18         default:
19             std::cout << "Unknown event..." << std::endl;
20     }
21 }

```



# Message Distribution System (MDS)

## Designs

### Specific receiver

*Der er 3 klasser involveret*

1. **Sender:** Objektet som genererer og sender specifikke handlinger til specifikke modtagere.
2. **Modtager:** Objektet som modtager og behandler handlingen sendt fra sender.
3. **Besked:** Besked/Datastruktur som indeholder information, som sendes fra sender til modtager.

Senderen har brug for receiverens adresse/id. Den skal kende til, hvor den skal sende beskeden til. Den skal også kende til den specifikke event handler, som skal benyttes. Senderen og modtageren skal ikke kende hinanden direkte, da der gør brug af en såkaldt event handler. Dette reducerer afhængigheder mellem objekterne.

*2 design patterns bruges (læs om dem længere nede)*

- **Observer Pattern:** Kan bruges hvis modtager abonnerer til specifikke events fra senderen.
- **Mediator Pattern:** Kan bruges til at centraliserer kommunikationen mellem sender og modtager, som reducerer afhængigheder mellem hinanden.

## Broadcast - receiver is irrelevant

*Der er 3 klasser involveret*

1. **Sender:** Broadcaster events til flere forskellige modtagere.
2. **Modtager:** Objekt som registrerer modtagning af events fra sender.
3. **Broker/Broadcaster:** Håndterer registrering af forskellige modtager-objekter, og sikre at alle dem, som er registreret, modtager eventsne.

Sammenlignet med specifik reciever, har senderen ikke brug for at kende til modtagerne, men kun broker. Senderen står også for, hvilken event skal broadcastes.

*1 design pattern bruges (læs om det længere nede)*

Her kan observer-implementationen bruges, da det er en "en-til-mange forbindelse" mellem objekter (En producer til mange consumere)

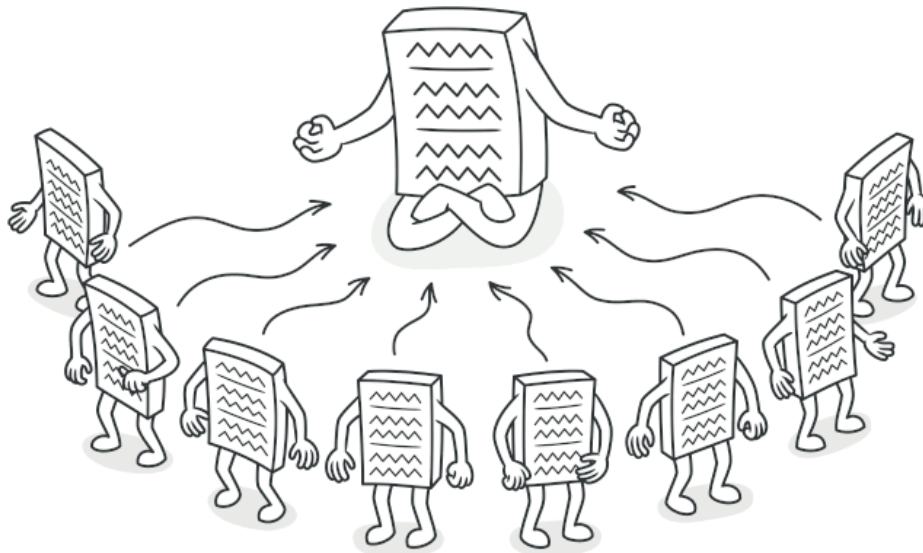
### *LOCAL VS GLOBAL ID?*

- **LOCAL:** Unikt i et specifikt content. (De forskellige modtagere)
- **GLOBAL:** Unikt henover hele systemet. (ID på specifik besked, som modtagere kan subscribe på)

MsgID er i et message distribution system GLOBAL, da det er en besked, som modtagerne "subscriber" på. Id for de forskellige modtagere er LOCAL, da det knyttet til hver modtager.

## Design Patterns

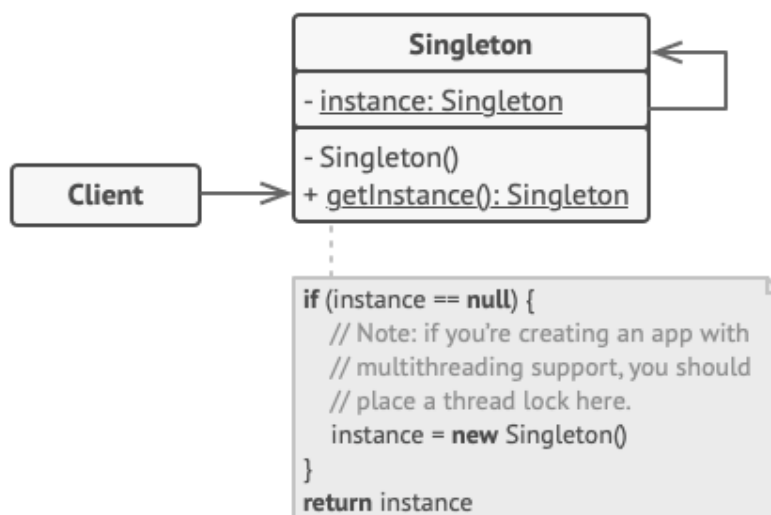
### Singleton



Figur 28: Singleton

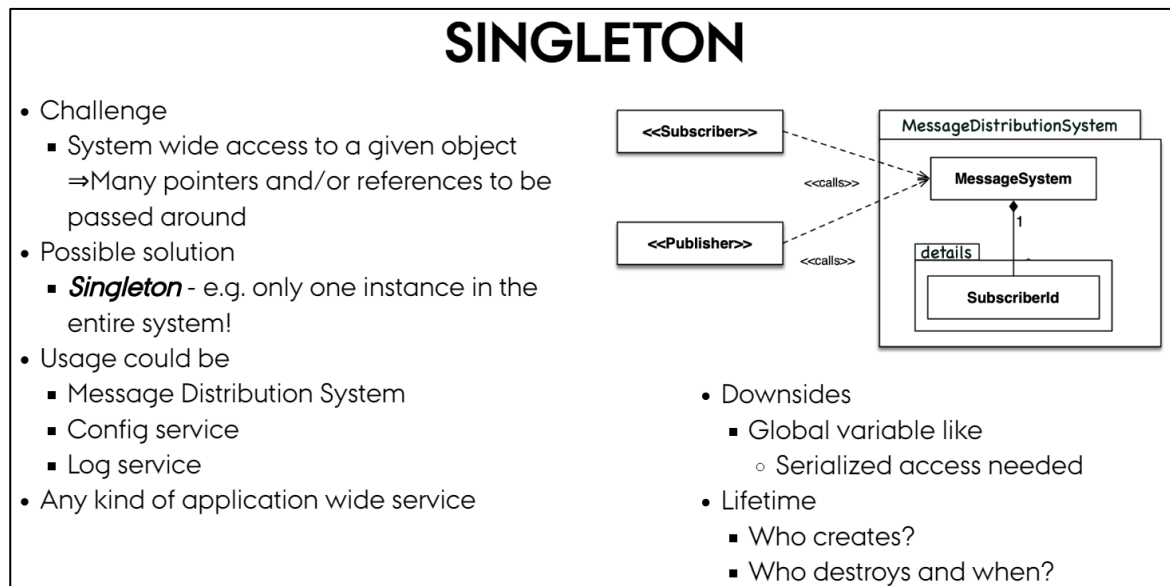
Singleton er, når man har en klasse, som kun tillader oprettelse af EN instans.

- **IMPORTANT!** Note that this behavior is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.



Figur 29: Singleton syntax

Dette implementeres ved hjælp af en private constructor + static metode, som kalder den private constructor.

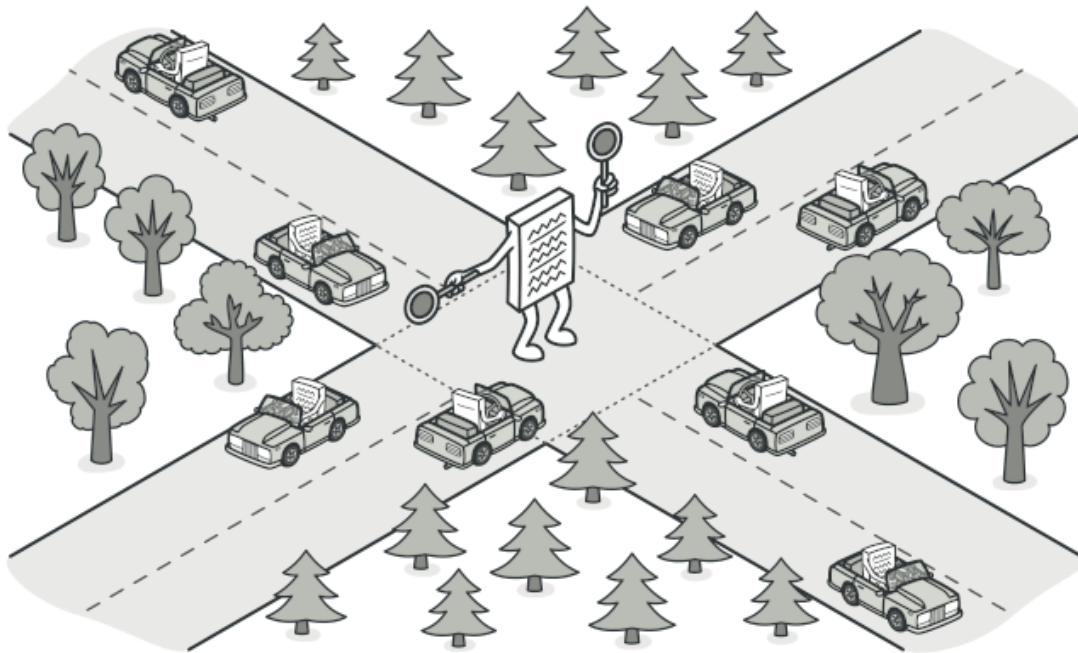


Figur 30: Singleton fra slides

Singleton er godt til systemer, hvor en enkelt instans er nyttig, f.eks. logging. Hvis man ikke tager forhold for implementeringen, kan forskellige tråde få adgang til at oprette instanser, hvilket ikke er meningen.

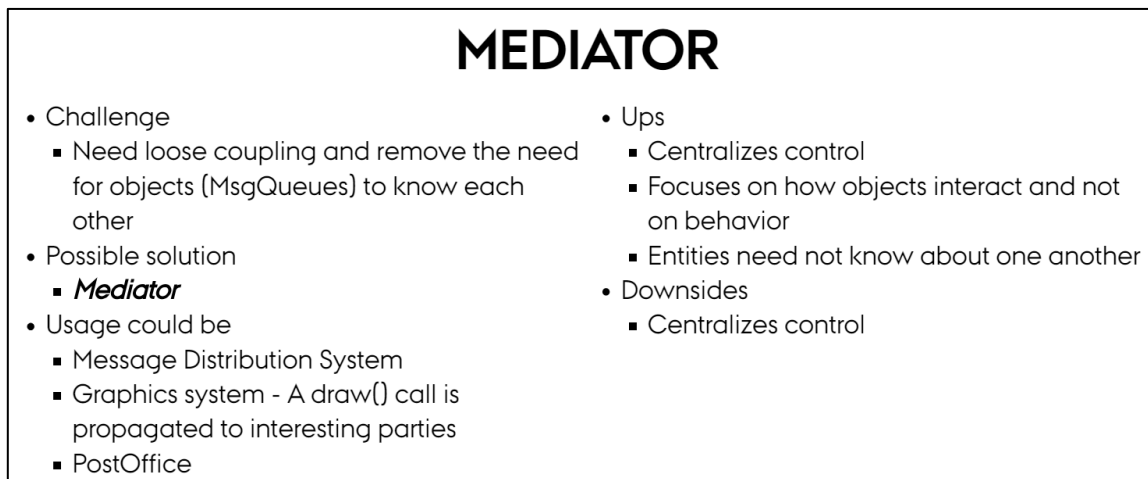
Når flere tråde får adgang til `getinstance()`-funktionen, og flere instanser bliver oprettet ved hjælp af Konstruktoren, brydes loven om Singleton Pattern. Man kan gøre brug af "synchronized" keywordet i C++ foran funktionen, som gør at kun en tråd kan eksekvere funktionen ad gangen.

## Mediator



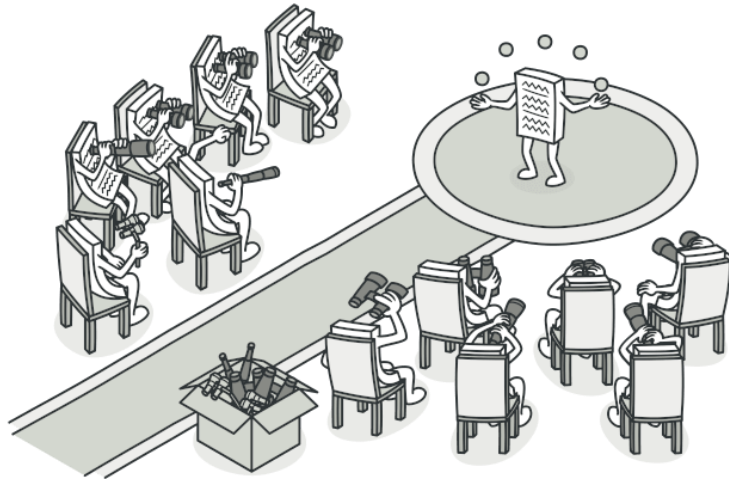
Figur 31: Mediator

**Et objekt styrer ALLE** objekter gennem sig. Objekter kan **KUN** kommunikere med hinanden gennem Mediator-objektet.



Figur 32: Mediator fra slides

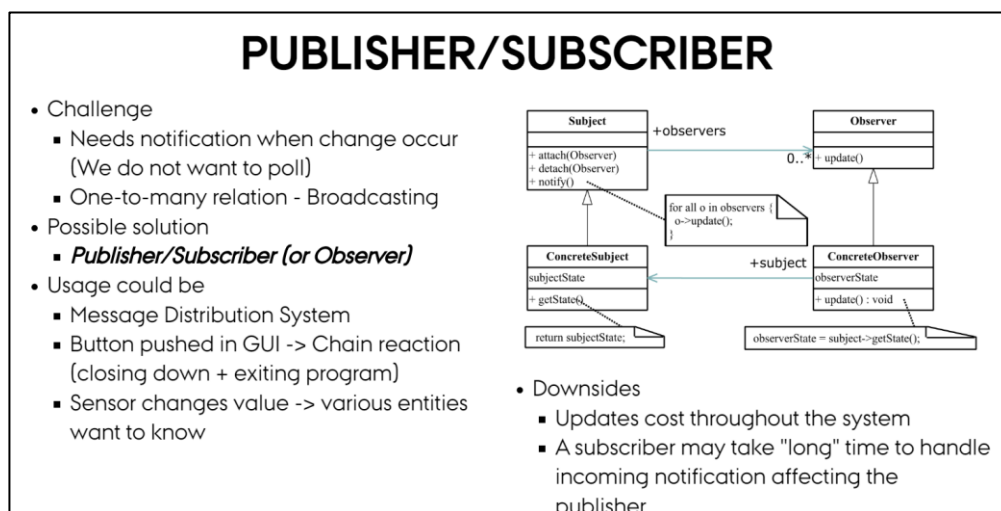
## Observer



Figur 33: Observer

**En-til-mange forbindelse** mellem objekter. Flere objekter kan ved hjælp af abonnement sikre sig beskeder hver eneste gang objektet de abonnerer foretager et event. Det svarer til, at man abonnerer på et nyhedsbrev. Nyhedsbrevet sendes kun ud til de folk, som abonnerer, og ikke alle.

**Observer (Subject)** klassen har styr på alle sine **consumere**, og har metoderne tilføje, slet, og notificer. Alle consumere har en opdater-metode, som håndterer opdateringen af disse ud fra, om observer ændres.



Figur 34: Observer fra slides

- **Observer Pattern:** Direkte kommunikation mellem subject og observers, oftest synkront og med tæt kobling.
- **Publisher/Subscriber Pattern:** indirekte kommunikation via en broker, oftest asynkront og med løs kobling, hvilket egner sig for "distributed systems."

## Eksempel på Message Distribution System

```

1  #pragma once
2
3  #include <string>
4  #include <map>
5  #include <vector>
6
7  class MessageDistributionSystem : osapi::Notcopyable
8  {
9  public:
10     /** Subscribes to the message that is globally unique and designated
11     * in msgId
12     * msgId Globally unique message id, the one being subscribed to
13     * mq The message queue to receive a given message once one is
14     * send to the msgId.
15     * id The receiver chosen id for this particular message
16     */
17     void subscribe(const std::string& msgId, osapi::MsgQueue* mq, unsigned long id);
18
19     /** Unsubscribes to the message that is globally unique and designated in
20     msgId
21     * msgId Globally unique message id, the one being subscribed to
22     * mq The message queue that received message designated by msgId.
23     * id The receiver chosen id for this particular message
24     */
25     void unsubscribe(const std::string& msgId, osapi::MsgQueue* mq, unsigned long id);
26
27     /** All subscribers are notified
28     * whereby they receive the message designated with 'm' below.
29     * msgId Globally unique message identifier
30     * m Message being send
31     */
32     template < typename M >
33     void notify(const std::string& msgId, M* m) const
34     {
35         osapi::ScopedLock lock(m_);
36         SubscriberIdMap::const_iterator iter = sm_.find(msgId);
37         if (iter != sm_.end()) // Found entries
38         {
39             const SubscriberIdContainer& subList = iter->second; // Why?
40             for (SubscriberIdContainer::const_iterator iterSubs = subList.begin(); iterSubs != subList.end(); ++iterSubs)
41             {
42                 M* tmp = new M(*m); // <- This MUST be explained!
43                 iterSubs->send(tmp);
44             }
45         }
46         delete m; // <- WHY? Could be more efficient implemented ,
47         // such that this de-allocation would be unnecessarily. Explain!
48     }
49
50     // Making it a singleton
51     static MessageDistributionSystem& getInstance()
52     {
53         static MessageDistributionSystem mds;
54         return mds;
55     }
56
57 private:
58     // Constructor is private
59     MessageDistributionSystem() {}
60
61     // Some form af key value pair , where value is signified by
62     // the msgId and the value is a list of subscribers
63
64     typedef std::vector <details::SubscriberId > SubscriberIdContainer;
65     typedef std::map <std::string, SubscriberIdContainer > SubscriberIdMap;
66     typedef std::pair < SubscriberIdMap::iterator, bool > InsertResult; SubscriberIdMap sm_; mutable osapi::Mutex m_;
67 };
68 #endif

```

Figur 35: Eksempel på et Message Distribution System

I eksemplet på sidste side, Figur 35, kan man se, at der bliver benyttet **singleton implementering**, da constructoren er privat, og den bliver kaldt gennem getInstance()-funktionen, som derefter opretter et objekt. Der er også gjort brug af **Observer pattern** som implementering, da flere subscribers modtager specifikke MsgID'er ud fra Notify()-funktionen.

1. Notify-funktionen starter med at sætte en mutex-lås så flere tråde ikke kan lave notify samtidigt.
2. Derefter kigger den i et map ved hjælp af en iterator, som indeholder alle subscriberne, som er subscribet til den specifikke MsgID. Hvis pointeren ikke når enden, sættes "subList" til alle subscriberne.
3. Derefter laves et for-loop, hvor en tmp instans af typen M dynamisk oprettes, som derefter bruges til at sikre, at hver eneste modtager modtager sin egen kopi af beskeden m ved hjælp af pointeren. Dette er med til, at hvis en subscriber har ændringer i beskeden, har det ikke en effekt på de andre subscribere/modtagere.
4. iteSubs->send(tmp) kalder send-metoden på den nuværende modtager, som er med til, at hver modtager håndterer beskeden enkeltvis.
5. Delete m; er med til at sikre, at der bliver ryddet op efter, beskeden er sendt til alle modtagerer så der ikke sker memory leaks. Man kunne have brugt smartpointers, som automatisk ville have ryddet op.



# Resource handling

## RAII (Resource Acquisition Is Initialization)

RAII (Resource Acquisition Is Initialization) er, når man binder resurser til et objekt, som gør, at når objektets livscyklus er ovre, så ryddes der automatisk op. Når et objekt går ud af scope kaldes destructoren automatisk og sørger for oprydning. Denne bliver kaldt selv hvis en exceptions bliver kastet.

Resurser kan være allokeret hukommelse oprettet med **new**, som skal sikres fjernes med delete for ikke at tage unødigt hukommelse, eller det kan også være noget ejet af mutex/semaphore tråde, som skal fjernes ordentlig for at undgå deadlocks.

## Implementeret operator overloads

### Dereference Operators:

- `operator*()`: Dereferences the stored pointer to access the object.
- `operator->()`: Allows member access to the pointed-to object.

## Smartpointere

SmartPointer rydder automatisk op. De håndterer automatisk livscyklusen af et dynamisk allokeret objekt.

I vores tidligere opgaver, hvor beskeder er blevet sendt mellem forskellige threads (EntryGuards, ExitGuards og Cars), skulle beskederne slettes ved den endelige modtager, efter at den var blevet håndteret af handleren. `boost::shared_ptr`, ville gøre håndteringen af beskeder lettere, eftersom en besked wrappet i en `shared_ptr` ville rydde op efter sig selv efter den går ud af scope.

## Counted Smart Pointer

- **Reference Tælling:** Opretholder en tælling af, hvor mange smart pointers der deler ejerskab af det samme objekt.
- **Automatisk Deallokering:** Sletter det administrerede objekt, når referencetællingen falder til nul.

Sker ved, at man opretter en "Counter" integer inde i constructoren, og i destructoren har man så først en dekrementer på counteren, og derefter et if-statement, som tjekker om denne Counter er lig med 0. Er den det, så bruges "Delete" på de forskellige membervariabler for at sikre oprydning:

```

#pragma once
#include <iostream>
#include <algorithm>
#include <string>

class SmartString
{
public:
    SmartString(std::string* str) : str_(str), counter_(new unsigned int(1)) {}

    ~SmartString()
    {
        --(*counter_);

        if (*counter_ == 0)
        {
            delete str_;
            delete counter_;
            std::cout << "Deleting smart string\n";
        }
    }

    SmartString(const SmartString &other)
    {
        str_ = other.str_;
        counter_ = other.counter_;
        ++(*counter_);
    }

    SmartString& operator=(const SmartString &other)
    {
        if (&other == this)
            return *this;

        str_ = other.str_;
        counter_ = other.counter_;
        ++(*counter_);
        return *this;
    }

    std::string* get() const { return str_; }
    std::string* operator->() { return str_; }
    std::string& operator*() { return *str_; }
    unsigned int getCount() const { return *counter_; }

private:
    std::string* str_;
    unsigned int* counter_;
};

```

Figur 36: Eksempel fra laboratorieøvelse

Der findes også en **boost::shared\_ptr**, som senere er blevet en del af C++ Ver. 11, og nu kan kaldes med **std::shared\_ptr**. Denne sørger selv for oprydning ved forladelse af scope.

## Rule of 3

Hvis en klasse har en brugerdefineret destructor, copy constructor eller copy assignment operator, skal den typisk implementere alle tre. Her er en forklaring på hver af dem:

1. **Destructor** (Destrukturen frigør ressourcer, når objektet går ud af scope)
2. **Copy Constructor** (sikrer, at en dyb kopi af objektet laves, så det nye objekt har sin egen kopi af ressourcerne)
3. **Copy Assignment Operator** (Copy assignment operatoren tillader tildeling af et objekt til et andet allerede eksisterende objekt. Den håndterer selv-tildeling og korrekt frigørelse og genallokering af ressourcer)

## Laboratorieøvelse (Recourse Handling)

I dette eksempel skal copy constructoren og copy assignment operatoren kunne fungere. Dette kræver en counter variabel, der holder øje med hvor mange kopier der findes af objektet. Hermed kan nye objekter oprettes som kopi af det originale objekt uden at der kan forekomme dangling pointers.

```
#pragma once

template <typename T>
class SmartPointer
{
public:
    SmartPointer(T* t) : t_(t), counter_(new unsigned int(1)){}

    ~SmartPointer()
    {
        if (*counter_ == 1)
        {
            delete t_;
            std::cout << "Deleting smart pointer\n";
        }
        else
            --(*counter_);
    }

    SmartPointer(const SmartPointer &other)
    {
        t_ = other.t_;
        counter_ = other.counter_;
        ++(*counter_);
    }

    SmartPointer& operator=(const SmartPointer &other)
    {
        if (&other == this)
            return *this;

        t_ = other.t_;
        counter_ = other.counter_;
        ++(*counter_);
        return *this;
    }

    T* get() const { return t_; }
    T* operator->() { return t_; }
    T& operator*() { return *t_; }
    unsigned int getCount() const { return *counter_; }

private:
    T* t_;
    unsigned int* counter_;
};
```

Figur 37: Implementationskode

```

#include <iostream>
#include "SmartPointer.h"

int main(int argc, char* argv[])
{
    std::cout << "---- SMART POINTER TEST WITH STRING ----\n";

    SmartPointer ss(new std::string("Hello world"));

    std::cout << "Address: " << ss.get() << "\tContent: " << *ss << "\tRef count: " << ss.getCount() << std::endl;

    {
        SmartPointer ss2 = ss;
        std::cout << "Address: " << ss2.get() << "\tContent: " << *ss2 << "\tRef count: " << ss2.getCount() << std::endl;

        SmartPointer ss3 = ss2;
        *ss3 = "Hello ISU'ers!";
        std::cout << "Address: " << ss3.get() << "\tContent: " << *ss3 << "\tRef count: " << ss3.getCount() << std::endl;
    }

    std::cout << "Address: " << ss.get() << "\tContent: " << *ss << "\tRef count: " << ss.getCount() << std::endl;

    std::cout << "\n---- SMART POINTER TEST WITH INT ----\n";

    SmartPointer num(new int(42));

    std::cout << "Address: " << num.get() << "\tContent: " << *num << "\tRef count: " << num.getCount() << std::endl;
    {
        SmartPointer num2 = num;
        std::cout << "Address: " << num2.get() << "\tContent: " << *num2 << "\tRef count: " << num2.getCount() << std::endl;

        SmartPointer num3 = num2;
        (*num3)++;
        std::cout << "Address: " << num3.get() << "\tContent: " << *num3 << "\tRef count: " << num3.getCount() << std::endl;
    }
    std::cout << "Address: " << num.get() << "\tContent: " << *num << "\tRef count: " << num.getCount() << std::endl;
}

```

Figur 38: Testkode

```

---- SMART POINTER TEST WITH STRING ----
Address: 0xaaaaccad36c0 Content: Hello world Ref count: 1
Address: 0xaaaaccad36c0 Content: Hello world Ref count: 2
Address: 0xaaaaccad36c0 Content: Hello ISU'ers! Ref count: 3
Address: 0xaaaaccad36c0 Content: Hello ISU'ers! Ref count: 1

---- SMART POINTER TEST WITH INT ----
Address: 0xaaaaccad3710 Content: 42 Ref count: 1
Address: 0xaaaaccad3710 Content: 42 Ref count: 2
Address: 0xaaaaccad3710 Content: 43 Ref count: 3
Address: 0xaaaaccad3710 Content: 43 Ref count: 1
Deleting smart pointer
Deleting smart pointer
~/isu/lecture8/ex8-3

```

Figur 39: Konsoloutput

Grunden til den fjerde adresse i hvert eksempel går tilbage til "Ref count: 1" er fordi, ss2 og ss3, eller num2 og num3, går ud af scope (funktionen) og dermed kaldes desctructoren, som dekrementer counteren. "Deleting smart pointer" udskrives til sidst, da "counter\_ == 1," i destrukterne, når main()-funktionen (scope) forlades.

## Ekstra

### Søren's Ping Pong eksempel fra livekodning

Dette C++ program demonstrerer et simpelt beskedkø-system ved brug af tråde, specifikt en "ping-pong" stil beskedudveksling mellem to tråde.

#### Beskedstruktur (PingelingInd)

A screenshot of a code editor showing the definition of a C++ struct named PingelingInd. The code is in a dark-themed editor with a light blue header bar that says 'cpp' and a 'Kopier kode' button. The struct is derived from osapi::Message and contains an integer counter and a string text. The constructor initializes these members.

```
cpp Kopier kode  
  
struct PingelingInd : osapi::Message  
{  
    PingelingInd(int counter, const std::string& text)  
        : counter{counter}, text{text}{}  
    int counter;  
    std::string text;  
};
```

Figur 40: Beskedstruktur

- Vi definerer en struktur PingelingInd, der er afledt af osapi::Message. Denne struktur indeholder en integer counter og en string text.
- Konstruktoren initialiserer disse medlemmer.

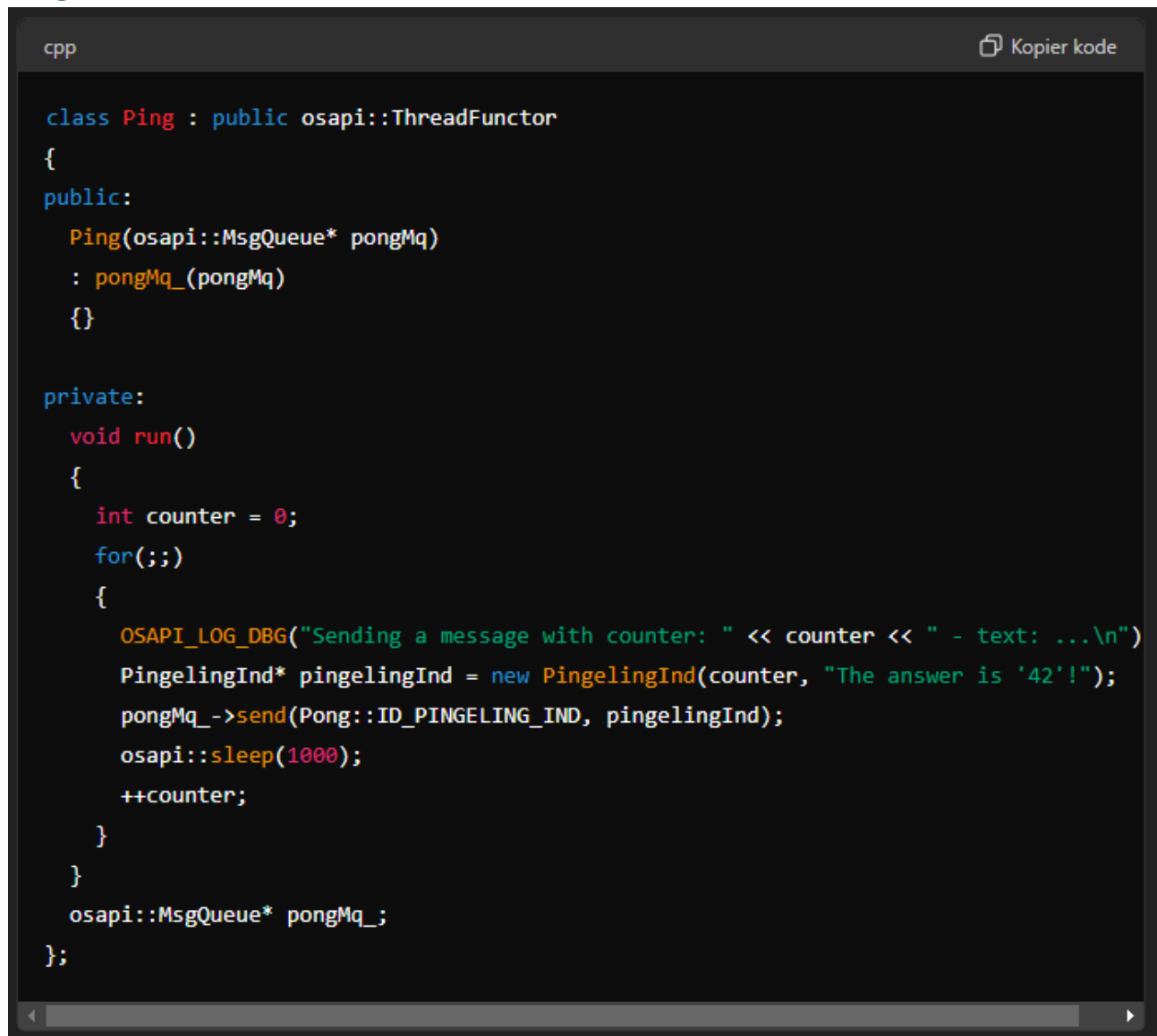
## Pong Klasse

```
cpp Kopier kode  
  
class Pong : public osapi::ThreadFunctor  
{  
public:  
    enum { ID_PINGELING_IND, ID_TERMINATE_IND };  
    osapi::MsgQueue* getMsgQueue(){ return &mq_; }  
private:  
    void handleMsgIdPingelingInd(PingelingInd* pingelingInd)  
    {  
        OSAPI_LOG_DBG("Got message PLI:: " <<  
            "Counter: " << pingelingInd->counter <<  
            " - Text: " << pingelingInd->text << "\n" );  
    }  
  
    void handleMsg(unsigned long id, osapi::Message* msg)  
    {  
        switch(id)  
        {  
            case ID_PINGELING_IND:  
                handleMsgIdPingelingInd(static_cast<PingelingInd*>(msg));  
                break;  
            case ID_TERMINATE_IND:  
                break;  
        }  
    }  
  
    void run()  
    {  
        while(running_)  
        {  
            unsigned long id;  
            osapi::Message* msg = mq_.receive(id);  
            handleMsg(id, msg);  
            delete msg;  
        }  
    }  
  
    osapi::MsgQueue mq_{10};  
    bool running_{true};  
};
```

Figur 41: Pong klasse

- Pong klassen nedarver fra `osapi::ThreadFunctor`, hvilket gør det muligt at køre den som en tråd.
- Den definerer en enumeration med besked ID'er: `ID_PINGELING_IND` og `ID_TERMINATE_IND`.
- Den indeholder en beskedkø `mq_` til at modtage beskeder.
- Metoden `getMsgQueue()` returnerer en pointer til denne beskedkø.
- Den private metode `handleMsgIdPingelingInd` behandler `PingelingInd` beskeder ved at logge deres indhold.
- En anden metode, `handleMsg`, sender beskeder til den relevante handler baseret på deres ID.
- Run-metoden er hovedløkken, der kontinuerligt modtager beskeder fra køen, behandler dem, og sletter beskedobjekterne.

## Ping Klasse

A screenshot of a C++ code editor with a dark theme. The editor shows the implementation of the Ping class, which inherits from osapi::ThreadFunctor. The class has a public constructor that takes an osapi::MsgQueue\* pongMq and a private run() method. The run() method contains a loop that sends PingelingInd messages to the pongMq queue, sleeps for 1000ms, and increments a counter. The code is color-coded: keywords in blue, identifiers in orange, and literals in green. A 'Kopier kode' button is visible in the top right corner of the editor.

```
cpp Kopier kode  
  
class Ping : public osapi::ThreadFunctor  
{  
public:  
    Ping(osapi::MsgQueue* pongMq)  
    : pongMq_(pongMq)  
    {}  
  
private:  
    void run()  
    {  
        int counter = 0;  
        for(;;)  
        {  
            OSAPI_LOG_DBG("Sending a message with counter: " << counter << " - text: ...\n")  
            PingelingInd* pingelingInd = new PingelingInd(counter, "The answer is '42'!");  
            pongMq_>send(Pong::ID_PINGELING_IND, pingelingInd);  
            osapi::sleep(1000);  
            ++counter;  
        }  
    }  
    osapi::MsgQueue* pongMq_;  
};
```

Figur 42: Ping klasse

- Ping klassen nedarver også fra osapi::ThreadFunctor.
- Den har en pointer til Pong trådens beskedkø, som bliver givet via konstruktøren.
- I run-metoden går den ind i en uendelig løkke, hvor den hver sekund opretter en PingelingInd besked, sender den til Pong trådens beskedkø, og inkrementerer tælleren.



## Main Funktion

```
cpp Kopier kode  
  
int main()  
{  
    OSAPI_LOG_DBG("Starting up...");  
  
    Pong pong;  
    osapi::Thread pongThread{&pong};  
  
    Ping ping{pong.getMsgQueue()};  
    osapi::Thread pingThread{&ping};  
  
    pongThread.start();  
    pingThread.start();  
    pongThread.join();  
    pingThread.join();  
  
    osapi::sleep(3000);  
}
```

Figur 43: int main()

- Hovedfunktionen er programmets indgangspunkt.
- Den logger en opstartsbesked.
- Den opretter en instans af Pong klassen og associerer den med en tråd pongThread.
- Derefter opretter den en instans af Ping klassen, giver den Pong's beskedkø, og associerer den med en anden tråd pingThread.
- Begge tråde startes med start().
- Hovedfunktionen venter derefter på, at begge tråde afslutter ved at bruge join().
- Til sidst pauser den i 3 sekunder for at lade de sidste beskeder blive behandlet, før programmet afslutter.

## Beskedudveksling

- Ping tråden sender en `PinglingInd` besked til Pong trådens beskedkø hver sekund.
- Pong tråden, som kører i en løkke, modtager disse beskeder, behandler dem ved at logge deres indhold, og sletter beskederne.

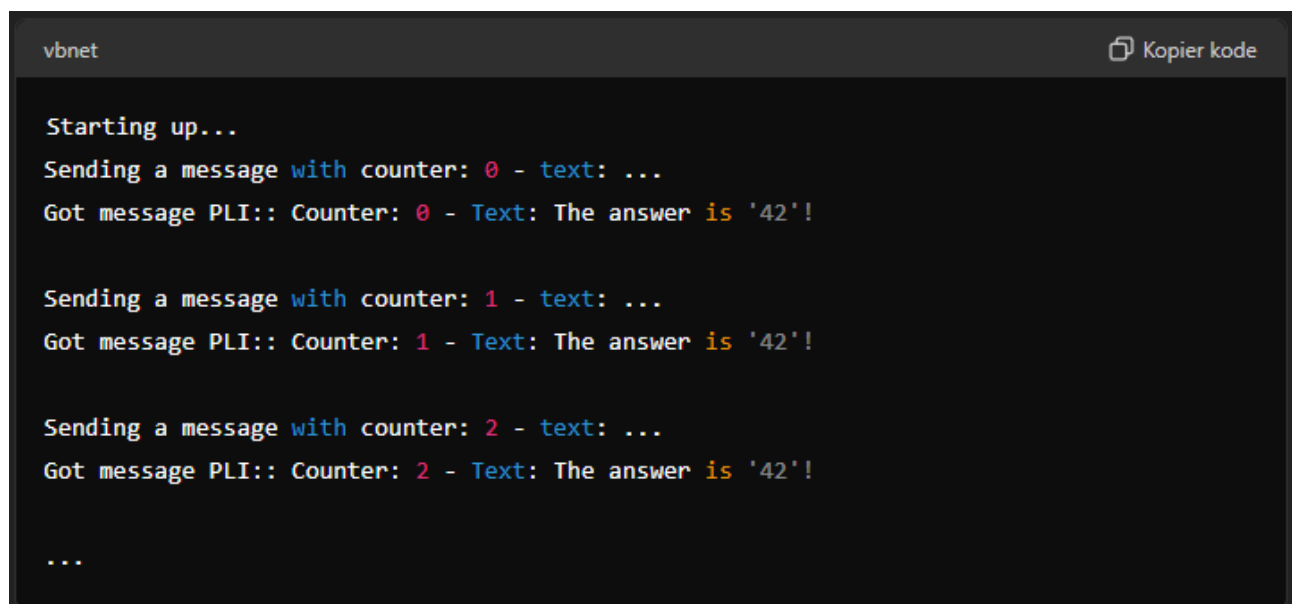
## Logning

- Gennem hele programmet bruges `OSAPI_LOG_DBG` til at logge forskellige beskeder, inklusive når programmet starter, når Ping sender en besked, og når Pong modtager og behandler en besked.

## Programoutput

Det forventede output vil være en serie af debug-logs, der viser tælleren og teksten af hver besked sendt af Ping og modtaget af Pong. Dette output vil fortsætte, så længe programmet kører, med tælleren, der øges hver sekund.

Eksempel på output:



```
vbnet Kopier kode

Starting up...
Sending a message with counter: 0 - text: ...
Got message PLI:: Counter: 0 - Text: The answer is '42'!

Sending a message with counter: 1 - text: ...
Got message PLI:: Counter: 1 - Text: The answer is '42'!

Sending a message with counter: 2 - text: ...
Got message PLI:: Counter: 2 - Text: The answer is '42'!

...
```

Figur 44: Konsol output