UNIVERSITY OF AARHUS

Faculty of Science

Department of Engineering

Indlejret Software Udvikling
Eksamens Dispositioner

Bjørn Nørgaard
IKT
201370248
bjornnorgaard@post.au.dk

Joachim Andersen
IKT
20137032
joachimdam@post.au.dk

Sidste ændring: December 30, 2015 at 21:08

LaTeX-koden kan findes her
https://github.com/joachimda/I3ISU/tree/master/Eksamen/Dispositioner

Todo list

Indholdsfortegnelse

List of Figures

1 Programs in relation to the OS and the kernel

1.1 Sub topics

- Processes and threads.
- Threading model.
- Process anatomy.
- Virtual memory.
- Threads being executed on CPU, the associated scheduler and cache.

1.2 Curriculum

- Slides "Intro to OS's".
- Slides "Parallel programs, processes and threads".
- OLA: "Anatomy of a program in memory", Gustavo Duarte.
- OLA: "The free lunch is over".
- OLA: "Virtual memory", pages 131-141.
- OLA: " Introduction to operating systems".
- OLA: "Multithreading".
- Kerrisk: Ch. 3-3.4 - System programming concepts.
- Kerrisk: Ch. 29 - Threads: Introduction.

1.3 Exercises

- Posix Threads.

1.4 Processes and threads

- En **process** er en instans af et program, som eksekveres.
- En **thread** er en del af eksekveringen, alle processer har mindst én thread.

1.4.1 Processes

- Har hver sit memory space.
- Process A kan ikke skrive i Process B's hukommelse.
- Kan kun kommunikere gennem IPC[1], læs mere i afsnit 3.4.
- Kan skabe andre processer som kan eksekvere det samme eller andre programmer.

[1] Inter-Process Communication, mekanismer kontrolleret af OS.

1.4.2 Threads

- Alle tråde i en process deler hukommelse på heap'en.
- Alle tråde har hver sin stack og program counter. (Tæller instruktioner så CPU ved hvor i koden vi er kommet til).
- Tråde er *ikke* individuelle, som processer er, og deler derfor deres kode, data og ressourcer med hinanden, internt i processen. Da hver process' memory er virtuelt og ejet af processen, kan der ikke pilles heri udefra. Der kan med undtagelse kommunikeres med IPC.
  - Skal passe på at man ikke sletter de øvrige trådes data.

Tråde er forskellige fra processer selvom de deler flere egenskaber og kendetegn. En tråd eksekveres i en process. Man kan sige at en tråd er en enkelt sekvensstrøm inde i en process.

Tråde gør det muligt at eksekvere flere sekvensstrømme ad gangen, og er derved en måde af effektivisere i form af parallelisering. OS's kernel giver gennem system calls mulighed for at oprette og nedlægge tråde.

Thread states

- Running
- Blocked (Når tråden ikke vil have CPU time)
- Ready (Når tråden gerne vil have CPU time)
- Terminated

1.5 Threading model

Der findes tre forskellige modeller:

- User level threading.
- Kernel level threading.
- Hybrid level threading.

1.5.1 User level threading

- Simpel implementering, ingen kernel support for threads.
- Ekstremt hurtig thread kontekst skift ikke brug for kernel handling).
- Ikke muligt at håndtere flere CPU-core.

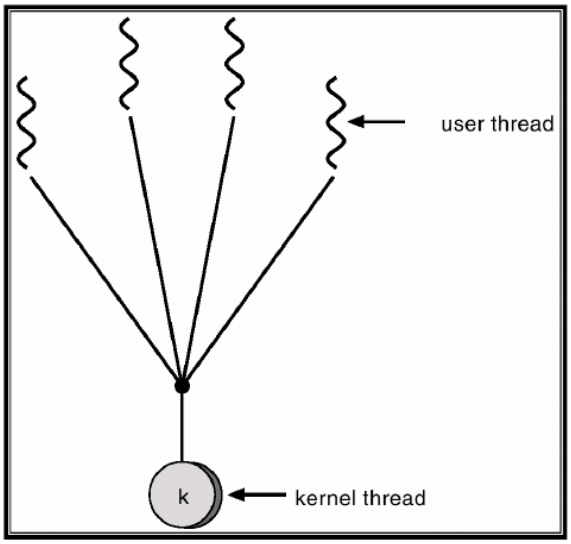

Figure 1: User level threading illusteret.

1.5.2 Kernel level threading

- OS kernel er bevidst om trådene. Giver overhead - er 100 gange langsommere end user level threads.
- Mapper direkte til "fysiske" threads som *scheduleren* kan kontrollere.
- Effektiv brug af flere kerner.

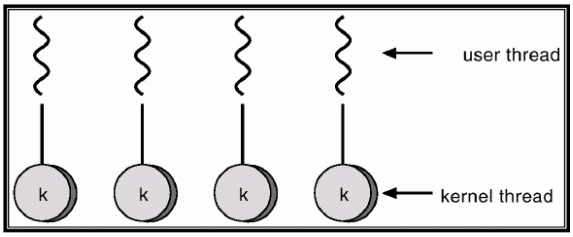

Figure 2: Kernel level threading illusteret.

1.5.3 Hybrid level threading

- Komplex implementering.
- Kræver god koordination mellem userspace og kernelspace scheduleren - ellers ikke optimal brug af resources.

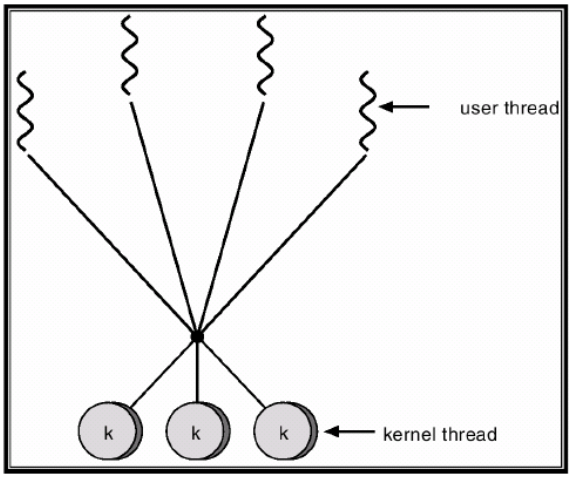

Figure 3: Hybrid level threading illusteret.

1.6 Process anatomy

- Når et program startes, starter en ny process.
- En process kører i sin egen memory sandbox, som et *virtual address space* (ca. 4GB på 32-bit platform).
- Hver process har sin egen **pagetable/virtual address space**.
- Den virtuelle memory mapper til fysisk memory addresser vha. pagetables.
- Alle processer har **virtual address space**, hvor en del er bestemt til kernel space.
- Kernel space er ens for alle processor og mapper til samme fysiske hukommelse.
- Kernel space er flagged i pagetable med privileged code, så kun kernel space programmer kan tilgå det memory. Der forekommer Page fault hvis en user-space process ulovigt forsøger at tilgå dette hukommelse.

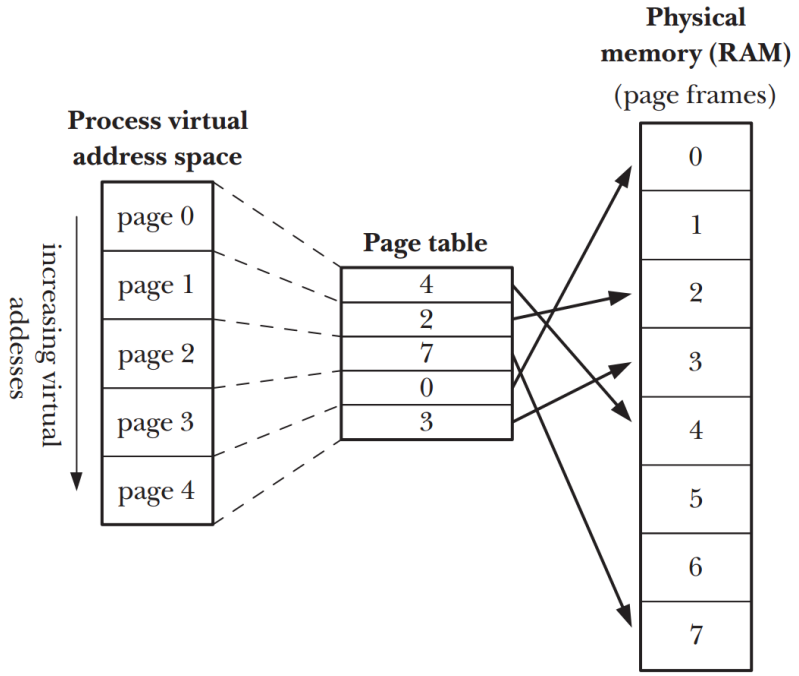

Figure 4: Virtual mapping via pagetable.

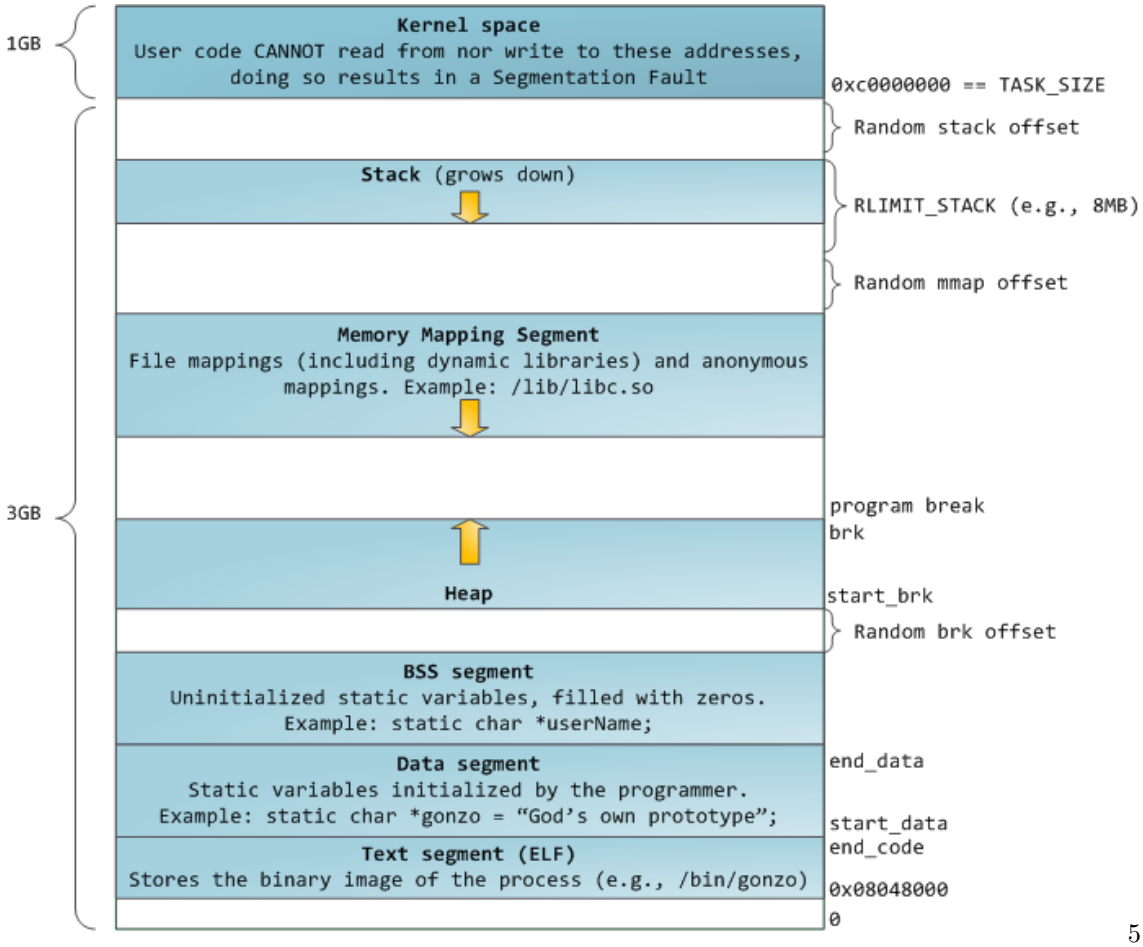

Figure 5: Diagram over standard segment layout.

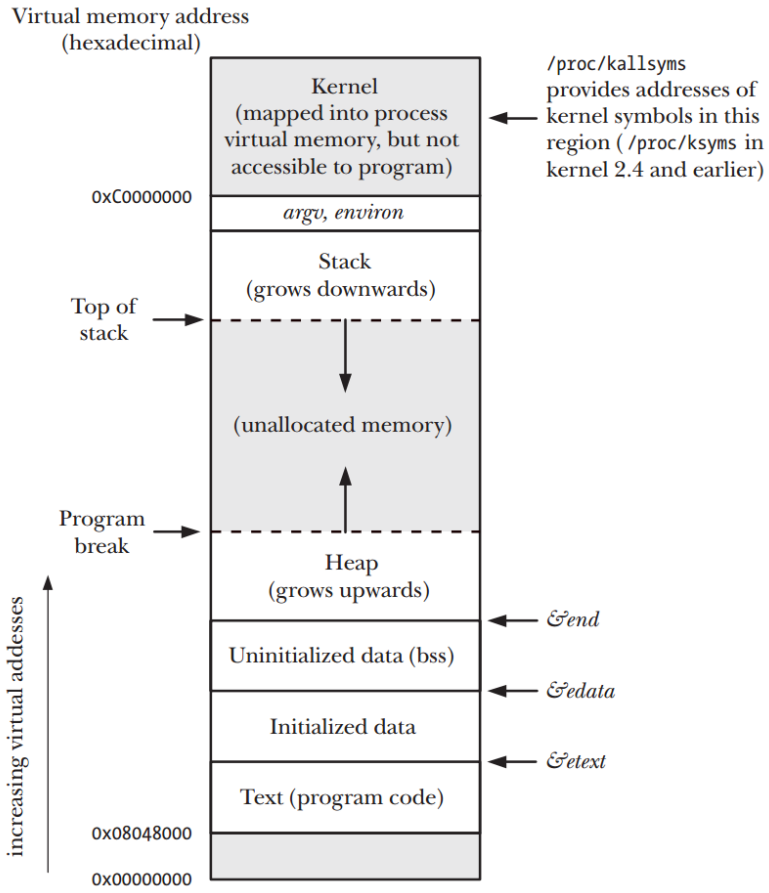

Figure 6: Typical memory layout of a process on Linux/x86-32.

Når der mappes virtuel memory til processer er der af sikkerhedshensyn indført random offsets for startaddressen på det virtuelle memory.

Resten, udover kernel space processens egen.

Her findes: Stack, heap, memory mapping, BSS, data og text/code segment.

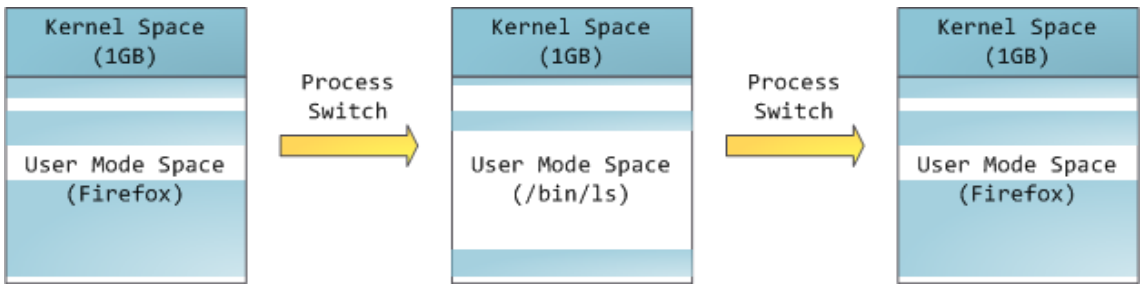

Figure 7: Diagram for process switch.

Alle processer har deres eget virtual address space, som der skiftes mellem ved context switches, se figur 7.

BSS Indeholder **ikke** initialiserede statiske variabler. Dette område er anonymt (mapper ikke til nogen fil, men er blok "reseveret").

Data segment Indeholder statisk allokerede og initialiserede variabler. Dette område er **ikke** anonymt og mapper til en fil på disken. Data segmentet kortlægger de initialiserede statiske værdier der er givet i source koden. Da dette er privat bliver ændringer ikke gemt i det mappede område.

Text segment Tag et eksempel: indholdet af en pointer (X-bit addresse) er i data segmentet men selve det den peger på ligger i **text segmentet**, som er *read-only* og indeholder alt din kode samt literals. Text segmentet mapper ens binære filer i hukommelsen (dette er read only, og forsøg på adgang resulterer i Segmentation fault).

1.7 Virtual memory

Linux processer bliver eksekveret i et virtuelt miljø. På denne måde tror hver process at den har al hukommelsen for sig selv.

Vigtige grunde til at vi bruger virtuel hukommelse:

- Resource virtualization.
  - En process skal ikke tænke på hvor meget hukommelse der er tilgængeligt.
  - Virtuel hukommelse gør en begrænset mængde fysisk hukommelse til en stor ressource.
- Information isolation.
  - Hver process arbejder i sit eget miljø, derved kan den ikke læse (eller skrive i) en anden process's hukommelse.
  - Forbedre sikkerheden, da en process således ikke kan "spionere" på en anden process.
- Fault isolation.
  - Kan ikke fucke andre processer op, da den ikke har adgang der deres hukommelse.
  - Hvis en process crasher/fejler/etc. ødelægger det ikke resten af systemet - problemet er isoleret i processen.

1.8 Threads being executed on CPU, associated Scheduler and Cache

Scheduleren sætter processer op til eksekvering på CPU og sørger for at skifte (switche) mellem processer:

1. Interrupt.
2. Save context.
3. Restore context.
4. Resume execution.

Der er to måder at schedule på:

- Preemptive - Preemptive modtager interrupt regulært og skifter proces.
- Non-preemptive - Non-preemptive kræver at processen selv siger "Nu må der skiftes".

1.8.1 Cache

De fleste CPU'er har minimum 3 typer caches:

- Instruction cache - fremskynder instruction fetching.
- Data cache - fremskynder fetch and store, og deles op i cache levels (L1, L2 osv.).
- TLB (Translation Lookaside Buffer) - fremskynder oversættelse af virtuel til fysiske addresser (TLB er en del af MMU'en[1])

TLB cache bruges hver gang en virtuel til fysisk oversættelse sker. De seneste data gemmes da i TLB'en. Dette forøger hastigheden på næste lookup til samme addresse.

[1] Memory Management Unit

2 Synchronization and protection

2.1 Sub topics

- Data integrity - Concurrency challenge.
- Mutex and Semaphore.
- Mutex and Conditionals.
- Producer / Consumer problem.
- Dining philosophers.
- Dead locks.

2.2 Curriculum

- Slides: "Thread Synchronization I and II".
- Kerrisk: Chapter 30: Thread Synchronization.
- Kerrisk: Chapter 31: Thread Safety and Per-Thread Storage (Speed read)".
- Kerrisk: Chapter 32: Thread Safety and Per-Thread Storage (Speed read)".
- Kerrisk: Chapter 53: Posix Semaphores (Named not in focus for this exercise)".
- OLA: "Pthread-Tutorial" - chapters 4-6.
- OLA: "Producer/Consumer problem".
- OLA: "Dining Philosophers problem".

2.3 Exercises

- Posix Threads
- Thread Synchronization I & II

2.4 Data integrity - Concurrency challenge

Selvom informationsdeling tråde imellem er en af de væsentligste fordele ved multitrådede applikationer, er det vigtigt at se sig for når flere tråde behandler den samme data. Man bruger termen *critical section* til at referere til et stykke kode der benytter en delt ressource, hvor dette bør gøres *atomic*[1]. Vi har derfor brug for en måde at sikre os eksklusiv adgang når der arbejdes på delte data.

2.5 Mutex and Semaphore

En ovenfornævnt kritisk sektion kan defineres ved hjælp af mutexes og semaphorer.

[1] Kerrisk: side 631

2.5.1 Mutexes

En tråd kan tage ejerskab over en mutex, dvs. at mutexen låses, og ikke kan tages af andre tråde, før den frigives af ejertråden og derved låses op.

En mutex skal initialiseres og destrueres.

- *lock(mut)* - tråden forsøger at tage mutexen. Funktionen er blokerende hvis mutexen er taget.
- *trylock(mut)* - samme som *lock()*, men i stedet for at blokere, returneres EBUSY.
- *timedlock(mut, timeOut)* - samme som *lock()*, hvor der dog her gives et timeout med som parameter, dvs. en tidsbegrænsing for hvor længe tråden skal vente på at få mutexen.
- *unlock(mut)* - tråden låser mutexen op.

2.5.2 Semaphorer

En semaphor kan beskrives som en counter der tæller hvor meget af en ressource der tilgængelig. En mutex er det samme som en semaphor der er begrænset til værdierne 0 og 1 (binær). To typer semaphorer:

1. Counted semaphore.
2. Binær semaphore.

En semaphor kan ved at inkrementere/dekrementere en counter, holde styr på og begrænse antallet af klienter der opererer på en ressource. En semaphor kan deles på tværs af processer. I en semaphor er der ikke ejerskab, hvilket medfører at en semaphor kan frigives af en client selvom denne ikke taget den.

Named og unnamed semaphorer En named semaphor kan benyttes på tværs af urelaterede processer ved brug af dens navn. Gøres med funktionen *sem_open()*.

En unnamed semaphor bruges som udgangspunkt kun inden for samme process med mindre den er gemt i delt hukommelse.

Om en semaphor er named eller unnamed defineres i semaphorens initiering (anden parameter - sharing level).

1. unnamed: *sem_init(ptr_sem, 0, init_val)*
2. named: *sem_init(ptr_sem, 1, init_val)*

Eksempler på brugen af semaphorer

- Et eksempel på brugen af en en semaphor, kunne fx være at begrænse antallet af forbindelser til en database til et bestemt antal.
- Begrænsning af CPU intensive opgaver til et bestemt antal CPU kerner.
- *take(sem)* - Hvis counteren er større end 0 dekrementeres denne og der returneres. Hvis counter er 0 blokeres der indtil den er større, hvorefter der så dekrementeres.
- *release(sem)* - inkrementerer counter med 1 og vækker evt. blokerende tråde.

2.6 Mutex and Conditionals

Conditionals gør det muligt for tråde at sende signaler til hinanden.

Vi har brugt det meget sammen med mutexes, for at sikre eksklusiv adgang til delte variabler. Bruges med fordel for ikke at stå og tjekke en variabel hele tiden (spilde CPU).

- *cond_wait*(*cond*, *mtx*)
  - Frigiver mutex.
  - Blokere den kaldende tråd, indtil den vækkes af signal/broadcast.
  - Låser mutex igen.
- *cond_timedwait*(*cond*, *mtx*, *time*)
  - Er det samme som *cond_wait*(), bortset fra at man kan angive en timeout. Når time-out'en er gået, vækkes tråden.
- *cond_signal*(*cond*)
  - Vækker *mindst* én af de sovende tråde. OBS! Man kan risikere at flere tråde vækkes!
  - Call *cond_signal*() if you want to get one thread to process a certain event.
- *cond_broadcast*(*cond*)
  - Vækker samtlige sovende tråde.
  - Call *cond_broadcast*() in case something happens everybody is concerned about.

2.6.1 Conditional variable - Fremgangsmåde

```
1  // How to use conditional variable
2  void someFunction()
3  {
4       lock(mtx);
5
6       while(!door_open)
7            cond_wait(cond, mtx);
8
9       // do something...
10
11      unlock(mtx);
12 }
```

Da *cond_wait*() først frigiver mutexen skal man altså låse den (mutexen) først.

Grunden til at *cond_wait*() er pakket ind i en *while*() er fordi vi kan risikere at *cond_wait*() "vækker" tråden uden at det nødvendige signal er "smidt". Således at hvis den vågner ved et uheld vil *while*() sikre at den bare ligger sig til at sove igen.

2.7 Priority Inversion

Priority inversion er når en høj prioritetstråd bliver "preempted" af en lav prioritetstråd (ved at en lavere-prioritetstråd tager en mutex) Vi kan løse dette med de to metoder:

2.7.1 Priority inheritance

Når en tråd tager en mutex assignes den midlertidigt til at have den højeste prioritet af de tråde der ellers venter på mutexen. Se figur 8.

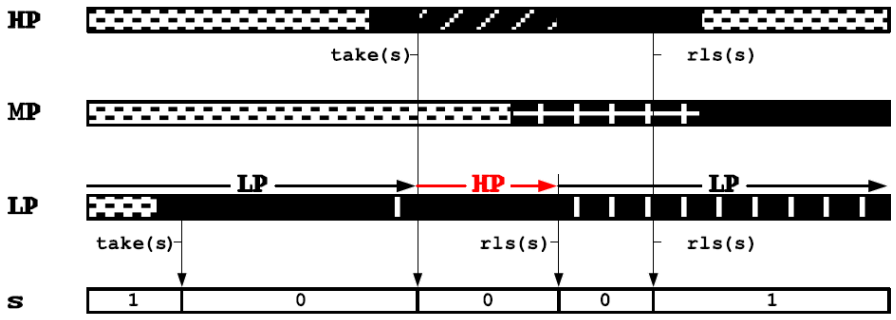

Figure 8: Eksempel med priority inheritance.

2.7.2 Priority ceiling

Alle mutexes assignes til høj prioritet (priority ceiling). Denne prioritet assignes til ejeren af mutexen så snart den tages. Se figur 9.

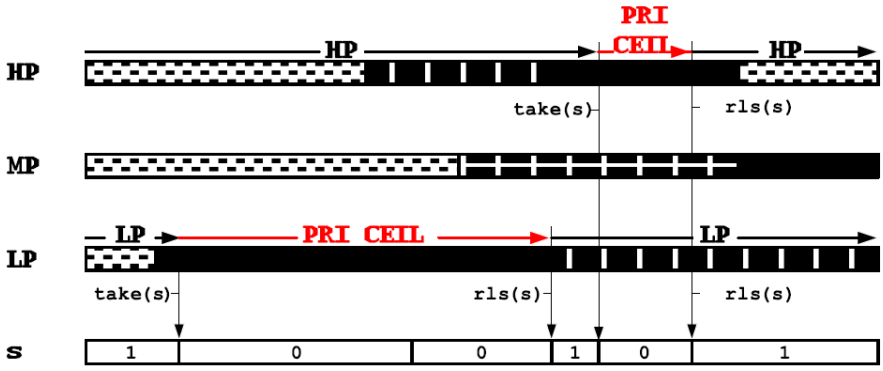

Figure 9: Eksempel med priority ceiling.

2.8 Producer/Consumer problem

Også kaldet *bounded-buffer problem.*

Når vi har to tråde som skal tilgå en data buffer.

- En **producer** som skal putte data i bufferen.
- En **consumer** som skal tage data ud af bufferen.

Problemet opstår når Produceren forsøger at pushe til en fuld buffer, eller når consumeren forsøger at poppe fra en tom buffer.

Hvis vi lader begge tråde checke om bufferen er tøm/fuld før de gør noget, spilder vi kostbar CPU tid.

2.8.1 Løsning med Mutex

```
// Producers put function
void Producer::put(item i)
{
      lock(mtx);

      while(size == max_size)
            wait(mtx, producer_cond);

      buffer.push(i);
      size++;

      unlock(mtx);
      signal(consumer_cond);
}
```

```
// Consumers get function
item Comsumer::get()
{
      lock(mtx);

      while(size == 0)
            wait(mtx, consumer_cond);

      buffer.pop();
      size--;

      unlock(mtx);
      signal(producer_cond);
}
```

2.8.2 Løsning med Semaphore

```
// Solutions with semaphores
empthySlotsSem = create_semaphore(buffer_size);
usedSlotsSem   = create_semaphore(0);

void Producer::put(item i)
{
      take(emptySlotsSem);
      buffer.push(i);
      release(usedSlotsSem);
}

item Consumer::get()
{
      take(usedSlotsSem);
      item = buffer.pop();
      release(emptySlotsSem);
      return item;
}
```

2.9 Dining philosophers

Wikipedia om problemet findes her.

I section 2.10 er fire betingelser for deadlocks beskrevet. Vores opgave er at bryde mindst én af disse.

1. **Mutual exclution** - Nope, de skal have hver deres egen gaffel.
2. **Hold and Wait** eller **Resource holding** - Heller ikke, de skal have to gafler for at spise.
3. **No preemption**[1] - Nix, de må ikke tage hinandens gafler.

[1] Læs om preemption i section 1.8

4. **Circular wait** - Bingo! Vi kan bryde denne cyklus!

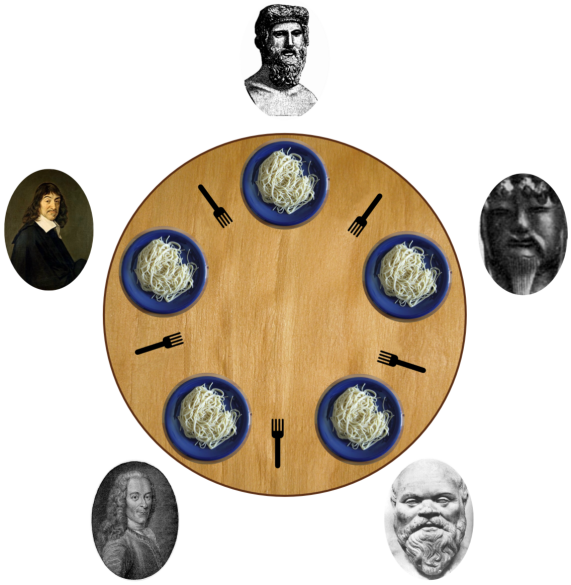

Figure 10: Dining philophers problem illustrated. Med uret fra toppen: Platon, Konfuzius, Socrates, Voltaire og Descartes.

Dette kan imidlertid gøres på flere måder:

2.9.1 Ressource hierarchy solution

Dette var Dijkstra's originale løsning på probelmet.

Ved at etablere en bestemt rækkefølge som ressourcerne skal "tages" i, kan vi bryde cyklen. For vores tilfælde vil gaflerne blive nummereret. Når en filosof så vil spise skal han først tage gaflen med de **mindste** nummer.

2.9.2 Arbitrator solution

Hvis de kun kan samle *begge* eller *ingen* af gaflerne kan samles op kan problemet løses. Dette gøres ved at filosofferne skal spørge en tjener[1] om tilladelse før de tage gaflerne, hvorefter de samler gafler op, indtil de har begge.

Problemet bliver imidlertid så at kun én filosof kan spise af gangen, selv hvis der er flere gafler ledige.

2.9.3 Chandry/Misra solution

I 1984 kom K. Mani Chandy og J. Misra med en anden løsning, som tillod at et arbitrært antal "agents" kan bruge et arbitrært antal ressourcer.

Den kræver ikke nogen central enhed, men den kræver tilgengæld at enhederne kommunikerer. Se wiki for præcis beskrivelse.

[1] Tjeneren kan implementeres som en mutex.

2.10 Dead locks

Deadlock er en situation hvor tråde ikke frigiver deres ressourcer fordi de venter på ressourcer som andre tråde ligeledes holder mens de venter på samme ting.

På wikipedia om deadlock kan man læse om de fire betingelser som skal være overholdt for at en deadlock kan opstå:

1. **Mutual exclution**
   Mindst én ressource skal være "holdt" i et *non-shareable mode*[1].
2. **Hold and Wait** eller **Resource holding**
   En tråd "holder" mindst én ressource og anmoder om flere ressourcer, hvor disse "holdes" af andre tråde.
3. **No preemption**[2]
   En ressource kan kun frigives af tråden som holder den.
4. **Circular wait**
   En tråd/process venter på en ressource som på tidspunktet holdes af en anden tråd/process. Denne anden tråd venter ligeledes på at den første tråd frigiver sin ressource.

For at deadlocks ikke skal opstå, skal mindst én af de ovenstående betingelser brydes.

[1] Kun én tråd kan bruge ressourcen af gangen.

[2] Læs om preemption i section 1.8

3 Thread communication

3.1 Sub topic

- The challenges performing intra-process communication.
- Message queue.
  - The premises for designing it.
  - Various design solutions - Which one chosen and why.
  - Its design and implementation.
- Impact on design/implementation between before and after the Message Queue.
- Event Driven Programming.
  - Basic idea.
  - Reactiveness.
  - Design - e.g. from sequence diagrams to code (or vice versa).

3.2 Curriculum

- Slides: "Inter-Thread Communication".
- OLA: "Event Driven Programming: Introduction, Tutorial, History - Pages 1-19 & 30-51".
- OLA: "Programming with Threads - chapters 4 & 6".

3.3 Exercises

- Thread Communication

3.4 The challenges performing Intra-Process Communication

Ulemper ved at implementere IPC med mutex[1] og conditional[2] variable, eller semephore[3].

- Det er en udfordring at undgå deadlocks[4] og timing issues (race conditions).
- Programmer bliver svære at læse og gennemskue (readability).
- Hvis en tråd venter på en bestemt betingelse (fx. condition variable), så laver den ikke noget fornuftigt i mellemtiden.
- Hvis tråde deler ressourcer:
  - Rækkefølgen hvori de tager/låser ressourcen skal planlægges grundigt for at undgå deadlocks.
  - Der skal muligvis låses flere mutexes, hvilket kan være meget tidskrævende.

Optimalt vil vi forsøge at lave en løsning hvor følgende gælder:

- Processering i en tråd kræver ikke noget lås (mutex etc).

[1] Mere om mutexes, se afsnit 2.5.1.
[2] Conditionals i samspil med mutexes, se afsnit 2.6.
[3] Semaphore, se afsnit 2.5.2.
[4] Mere i afsnit 2.10.

- Tråde kan kommunikere med hinanden.
- Kommunikation mellem tråde skal kunne indeholde data.
- Tråde skal kunne sende disse beskeder når de vil. De skal altså ikke være bundet af noget bestemt flow i programmet.
- Mange tråde kan kommunikere med flere andre tråde.

3.5 Message queue

Disse beskedkøer skal fungere som "mellemstation" hvor besked kan opbevares før tråden er klar til at håndtere beskeden. Brugte vi ikke en sådan mekanisme ville beskeden "gå tabt". Alternativt skal vi bruge ressourcer på at synkronisere den sendende og modtagende part, således at beskeden sendes og modtages på "samme tid".

3.5.1 The premises for designing it

Producer/consumer problemet[1] skal løses:

- *send()* putter en besked i køen og blokere hvis køen er fuld.
- *receive()* henter en besked fra køen og blokere hvis køen er tom.

Her kommer conditionals ind i billedet. To conditions - QueueFull og QueueEmpty.

3.5.2 Various design solutions - Which one chosen and why

Implementeres med en FIFO kø af typen STL *queue*, eventuelt med max på antal beskeder.

Beskedtypen Vi bruger i opgaven beskeder der arver fra en basisbesked. På denne måde er besked objekterne typesikre idet de kan slettes med deres **baseklasse pointer**. Det er muligt at lave beskeder på andre måder:

- void* - ingen typesikkerhed (delete problemer).
- Templates - For kompleks.

Vi caster vore basis message til den messagetype der ønskes.

Queue items De items som køerne indeholde. Kan eksempelvis implementeres med en struct, bestående af et id og et beskedobjekt af basisklassen Message.

Dispatcheren kan da switche på QueueItems id, og handleren kan reagere på beskeden.

Asynkrone køer

- Modtageren skal ikke have svar med det samme.
- I sekvensdiagrammet tegnes besked som en åben pil, se figur 11.
- Afsenderen kan forsætte operationer uden svar fra modtager.

[1] Yderligere beskrevet i afsnit 2.8

Synkrone køer

- Modtageren skal have svar med det samme.
- I sekvensdiagrammet tegnes besked som en lukket pil, figur 11.
- Afsenderen afventer svar fra modtageren før den kan forsætte operation.

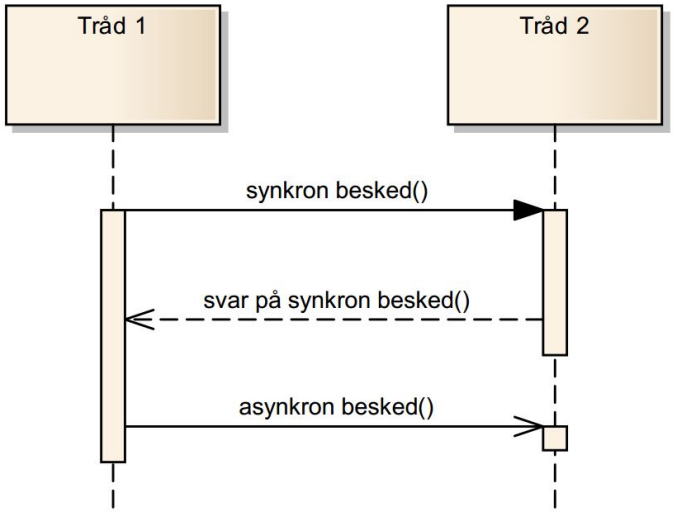

Figure 11: Sekvensdiagram for syntax ved brug af synkron og asynkron besked køer.

3.5.3 Its design and implementation

MessageQueue implementering med psudokode:

```
// send() function
lock(mtx);

while(num_of_items_in_queue == max_size)
     cond_wait(cond, mtx);

put_item_in_queue(msg, id);

unlock(mtx);
signal(cond);
```

```
// receive() function
lock(mtx);

while(num_of_items_in_queue == 0)
     cond_wait(cond, mtx);

item = get_item_from_queue();

unlock(mtx);
signal(cond);

id = item->id;
msg = item->msg;

delete item;
return msg;
```

3.5.4 Impact on design/implementation between before and after the Message Queue

Brugen af vores MessageQueue har betydet at vi udenfor køens implementering ikke har skulle bekymre os om at huske mutexes, conditionals etc. Man kan sige at vores MessageQueue tilbyder et ekstra abstraktionslag.

3.6 Event Driven Programming

3.6.1 Basic idea

Event drevet programmering er opgjort af event producenter og event subscribers. Tråde reagerer på indkommende events, og sover når der ikke sker noget. Event drevet programmeret er specielt egnet til GUI applikationer hvor user inputs ofte bruges.

Der er flere måder at implementere EDP på. I vores ISU øvelse har vi brugt en MessageQueue hvis funktionalitet er beskrevet ovenfor. Hvis vi ser på figur 12 vil vores MessageQueue være mellemledet før dispatcheren. Dette er smart da det ikke er sikkert at *dispatcher* kan nå at videresende alle events til deres respektive handlers. MessageQueue'en fungerer derfor som en buffer. Implementering kommer derfor til at følge dette mønster:

- Genereing af beskeden.
- Get message fra MsgQueue.
- Kør dispatch.
- Handle event.
- De-alloker besked.

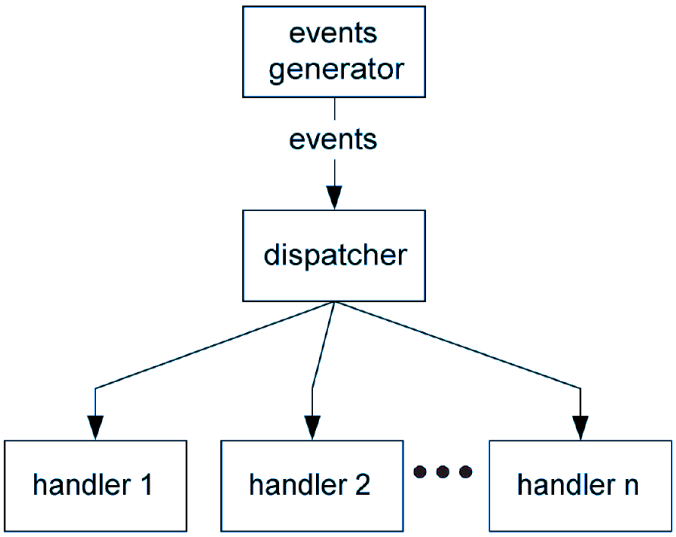

Figure 12: Illustration af et handler pattern.

3.6.2 Design - e.g. from sequence diagrams to code (or vice versa)

Vis evt. PLCS sekvensdiagram

4 OS API

4.1 Sub topics

- The design philosophy - Why OO and OS Api?
- Elaborate on the challenge of building it and its currenct design:
  - The PIMPL / Cheshire Cat idiom - The how and why.
  - CPU / OS Architecture.
- Effect on design/implementation:
  - MQs (Message queues) used with pthreads contra MQ used in OO OS Api.
  - RAII in use.
  - Using Threads before and now.
- UML Diagrams to implementation (class and sequence) - How?

4.2 Curriculum

- Slides: OS Api".
- OLA: OSAL SERNA SAC10".
- OLA: Speciffcation of an OS Api".
- Kerrisk: Chapter 35: Process Priorities and Schedul-ing".

4.3 Exercises

- OS API.

4.4 The design philosophy - Why OO and OS Api?

Et OS Api giver et abstraktionslag for operativsystemet. Denne abstraktion gør det muligt at *unify* OS arkitektur, og dermed opnå højere portabilitet for applikationer. Med andre ord giver et OS Api os et generelt interface der kan genbruges på tværs af platforme.

I vores tilfælde, i arbejdet med POSIX tråde, betyder et OS Api, at vi ikke skal omskrive vores kode, så den kan kompileres til Windows. Se illustration på figur 13. OS Api'et sikrer samtidigt at man ikke skal tænke på de lavpraktiske OS-specifikke detaljer, og at man i stedet kan fokusere på brugbarheden af applikationen.

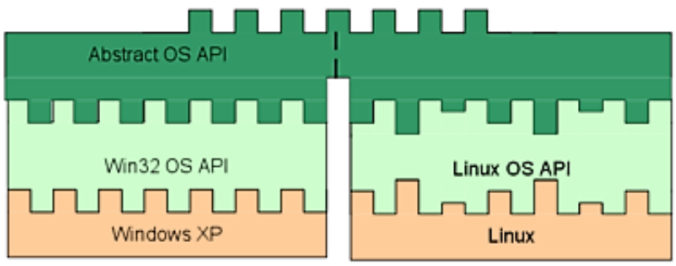

Figure 13: Illustration af OS abstraktion

4.4.1 Hvorfor et objekt-orienteret design?

- Lettere af arbejde med, når man er vant til objekter.
- Renere kode.
  - Koden er mere letlæselig.
  - Bedre mulighed for at tilføje/ændre i koden.
  - Bedre *testability*.
  - Nemt at lave interaktive designs (Eventbaseret programmering).

4.5 Elaborate on the challenge of building it and its currenct design

4.5.1 The PIMPL / Cheshire Cat idiom - The how and why

PIMPL - Pointer to Implementation er et SW programmerings idiom, der fremmer *implementerings abstraktion* for klasser. Lidt på samme måde som interfaces i C#. Dette gøres vha. en ekstra pointer og funktionskald.

Et eksempel med en *Book* klasse:

```
1 class Book
2 {
3 public:
4     void print();
5 private:
6     std::string  m_Contents;
7     std::string  m_Title;
8 }
```

Code listing 1: Klassen Book uden PIMPL

Som det kan ses i kodeudsnit 1, skal alle Books klienter recompilere hver gang der tilføjes eller ændres i klassen book.

Vi tilføjer et private medlem, ***class BookImpl*** samt ***BookImpl* m_p***. Vi har også brug for både destructor og constructor nu, for hhv at slette og sætte m_p! Med PIMPL implementeret får vi da:

```
1 /* public.h */
2 class Book
3 {
4 public:
5     Book();
6     ~Book();
7     void print();
8 private:
9     class BookImpl;
10    BookImpl* m_p;
11 }
```

Code listing 2: Klassen Book med PIMPL implementeret

Flere klienter kan nu have deres egen implementering af klassen book. En *privat* implementering kan se således ud:

```
1 Book::Book()
2 {
3     m_p = new BookImpl();
4 }
5
6 Book::~Book()
7 {
```

```
8       delete m_p;
9   }
10
11  void Book::print()
12  {
13      m_p->print();
14  }
15
16  /* then BookImpl functions */
17
18  void Book::BookImpl::print()
19  {
20      std::cout << "print from BookImpl" << std::endl;
21  }
```

Code listing 3: Privat implementering af klassen Book

Når vi i vores main ønsker en instans af Book, invokeres det således:

```
1  int main()
2  {
3      Book *b = new Book();
4      b->print();
5      delete b;
6  }
```

Code listing 4: Brug af Book klassen i main()

4.5.2 CPU / OS Architecture

OS Api'er fungerer som et strategy pattern, hvor der vælges strategi alt efter hvilken platform man er på.

4.6 Effect on design/implementation

Igen, har brugen af OS Api'et gjort det lettere for os at bruge tråde, mutex's m.m.

4.6.1 MQs (Message queues) used with pthreads contra MQ used in OO OS Api

Med henyn til pthreads gør vi blot som beskrevet nedenfor i kodeudsnit 8. I forhold til locking og unlocking af mutexes sker dette i vores MessageQueue. Hvis vi ser på MQ koden, bruges der blot et Mutex objekt hvorpå *lock()* og *unlock()* kaldes. ***Mutex.cpp*** i linux mappen står for at kalde de respektive lock/unlock funktioner der passer til POSIX threads.

4.6.2 RAII in use

Et idiom brugt til sikker brug af ressourcer, for at undgå memory-leaks. Se afsnit 6.4 om RAII. Ideen er at ressourcen allokeres under objektets oprettelse og deallokeres ved destruktion.

Kendt under flere navne:

- Resource Acquisition Is Initialization (RAII).
- Constructor Acquires, Destructor Releases (CADRe).
- Scope-based Resource Management (SBRM).

Mutex klassen i vores OSAPI er vist i kode udsnit 5 og viser tydeligt hvordan mutexen initialiseres i constructoren og destroy'es i destructoren.

```
Mutex::Mutex()
{
    pthread_mutex_init(&mutte, NULL);
}

Mutex::~Mutex()
{
    pthread_mutex_destroy(&mutte);
}

void Mutex::lock()
{
    pthread_mutex_lock(&mutte);
}

void Mutex::unlock()
{
    pthread_mutex_unlock(&mutte);
}
```

Code listing 5: Mutex implementering i OSAPI

4.6.3 Using Threads before and now

Før når vi skulle anvende tråde ville vi gøre noget i stil med det vist i kode udsnit 6, hvor en tråd erklæres, startes og blokere til den er færdig.

```
pthread_t thread;
pthread_create(&thread, NULL, func, (void*)&arg);
pthread_join(thread, NULL);
```

Code listing 6: Anvendelse af tråde før OSAPI

Med OSAPI er det hele mere Objekt Orienteret og generelt nemmere at bruge. Først skal en trådklasse laves (se kode udsnit 7) ved at arve fra *ThreadFunctor* som skulle færdiggøres i øvelse 6.

```
#include <osapi/Thread.hpp>

class ThreadClass : public ThreadFunctor
{
public:
    virtual void run()
    {
        // job of thread while running
    }
private:
    // class members
};
```

Code listing 7: Trådklasse via nedarvning fra ThreadFunctor

Kode udsnit 8 viser hvordan vi nu kan bruge tråde ved hjælp af trådklassen i udsnit 7 til let at starte en tråd.

```
ThreadClass threadclass();    // class to be run in thread
Thread thread(&threadclass);  // thread which the class will run in

thread.start();
thread.join();
```

Code listing 8: Anvendelse af tråde efter OSAPI

4.7 UML Diagrams to implementation (class and sequence) - How

5 Message Distribution System (MDS)

5.1 Sub topics

- Messaging distribution system - Why & how?
- The PostOffice design - Why and how?
- Decoupling achieved.
- Design considerations & implementation.
- Patterns per design and in relation to the MDS and PostOffice design:
  - GoF Singleton Pattern
  - GoF Observer Pattern
  - GoF Mediator Pattern

5.2 Curriculum

- Slides: "A message system".
- OLA: "GoF Singleton pattern".
- OLA: "GoF Observer pattern".
- OLA: "GoF Mediator pattern".

5.3 Exercises

- The Message Distribution System

5.4 Message Distribution system - Why & how?

5.4.1 Why?

- Lavere kobling.
  - Subscriberne kan tilføje/fjerne sig selv, uden at publisheren skal tænke på det.
- Lade subscriberne bestemmer selv hvad de vil have.
  - Eksempelvis kan en graf subscribe på data til x og y, men ikke z.

5.4.2 How?

- Subscriberen sender følgende til MDS ved subscribtion:
  - Hvilke data den vil subscribe på, via globalt id.
  - Pointer til egen MsgQueue.
  - Local ID, som skal bruges når den skal modtage, hvilket gør klassen uafhængig af global navngivning[1].
- Ved opdatering på data x, gør Publisher følgende:
  - Finder alle der subscriber på x.
  - Sender data x til subscribernes MsgQueue, med deres individuelle local ID.

[1] Delvist svar, har indtil videre ikke kunne bestemme den nøjagtige grund.

5.5 The Postoffice Design - Why and how?

5.5.1 Why?

Lavere kobling, som altid. Sender kender kun til modtagers navn og intet andet.

5.5.2 How?

- Bruger navn på modtager.
  - Sender skriver navn på modtageren.
  - PostOffice finder den rigtige MsgQueue på baggrund af dette navn.
- Bruger arv.
  - Alle arver det samme enum af (globale) ID'er, som kan sendes med.

5.6 Design considerations and implmentation.

5.7 Patterns per design and in relation to the MDS and Postoffice design

5.7.1 GoF Singleton Pattern

Et singleton pattern er et software design pattern, der gør det muligt at begrænse antallet af instanser af en klasse til 1.

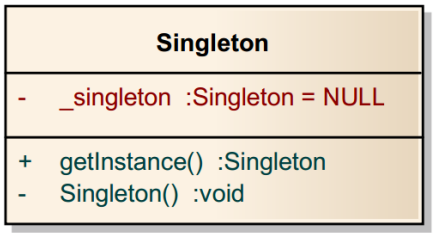

Figure 14: En Singleton klasse

How to Singleton

- Private constructor - Sikrer at kun klassen kan oprette en instans, ingen andre kan oprette eller kopiere den!
- statisk *getInstance()* funktion - Statisk så den kan kaldes uden et objekt. Denne funktion returnerer instansen, og laver en ny hvis den private pointer (instansen) til sig selv er NULL.
- Statisk private pointer til sig selv - Den eneste instans.

Singleton pattern i forhold til MDS og postoffice design

5.7.2 GoF Observer Pattern

Et GoF observer pattern, også kaldet Publisher/Subscriber pattern består af to primære elementer:

- Publisher (subject) - Kan have flere subscribere(observere) der abonnerer på den.
- Subscriber (observer) - Bliver notificeret når publisher ændrer state. Herefter anmodes publisher om faktiske ændringer.

Dette pattern er brugbart når ændringer i et objekt kræver ændring i andre, og hvor man ikke ved hvilke og hvor mange der er tale om.

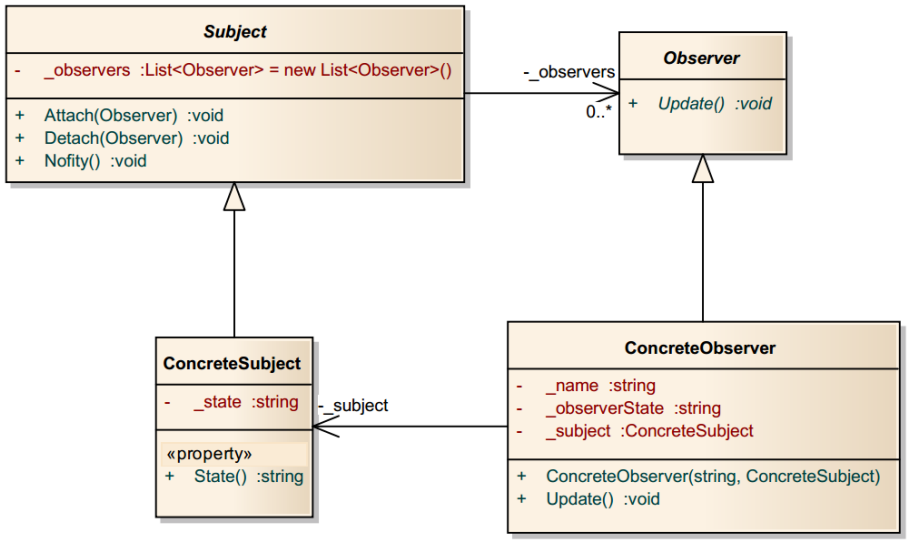

Figure 15: Et Observer pattern, konfigureret til *pull*.

Observer pattern i forhold til MDS og postoffice design

5.7.3 GoF Mediator Pattern

Når objekters funktionalitet distribueres ud mellem hinanden, vil der opstå høj kobling, og masser af interkonnektivitet. I et mediator pattern oprettes et separat mediator-objekt, som står for at kontrollere objekters interaktioner med hinanden.

Løsning Definition og identifikation af deltagende klassers type:

- Mediator.
  - Definerer et interface til kommunikation med "Colleague objekter".
- Concrete mediator.
  - Implementerer kommunikationsinterfacet, ved at koordinere Colleague objekter.
- Colleague klasser.
  - Hver colleague klasse kender sit mediator object.

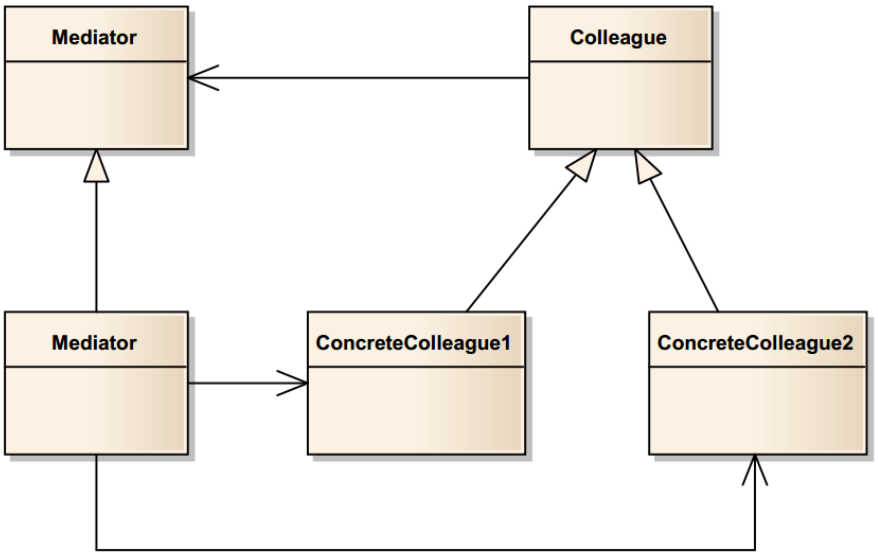

Figure 16: Generelt klassediagram om Mediator pattern.

- Når colleaguen vil snakke med en anden klasse, kommunikeres der udelukkende gennem mediatoren.

Brugen af et mediator pattern begrænser mængden af afledte klasser i et system, i og med mediatoren centraliserer funktionalitet der ellers ville være spredt ud på mange klasser. Ved at pakke objekters interkonnektivitet ind i et mediator pattern, får man samtidig skabt et ekstra abtraktionsniveau der gør funktionalitet mere overskuelig.

Konsekvenser Mediatoren samler en masse funktionalitet på ét sted, hvilket har den fordel at interaktionen mellem objekter bliver nemmere, men mediatoren får derved større ansvar og bliver mere kompleks. Mediatoren kan derfor hurtigt blive en monolit, som er svær at vedligeholde.

Mediator pattern er godt at bruge, i tilfælde, hvor der opstår mange forbindelser mellem mange forskellige klasser. Ved hjælp af mediatoren centraliseres referencerne til de forskellige objekter ét sted, og samler kald til mediatoren, i stedet for enkelte klasser. Dog skal der passes på at mediatoren ikke bliver til en monolit, hvis for meget funktionalitet bliver pakket ind i den.

6 Resource handling

6.1 Sub topics

- RAII - What and why?
- Copy construction and the assignment operator.
- What is the concept behind a Counted SmartPointer?
- What is *boost* :: *shared_ptr* <> and how do you use it?

6.2 Curriculum

- Slides: "Resource Handling".
- OLA: "RAII - Resource Acquisition Is Initialiation".
- OLA: "SmartPointer".
- OLA: "Counted Body".
- OLA: "*boost* :: *shared_ptr*".
- OLA: "Rule of 3".

6.3 Exercises

- Resource Handling.

6.4 RAII - What and why?

"Wrap up all recources in their own object that handles their lifetime and put object on stack"

Et idiom brugt til sikker brug af ressourcer, for at undgå memory-leaks. Ideen er at ressourcen **allokeres under objektets oprettelse** og **deallokeres ved destruktion**. På denne måde skal man ved brug af RAII idiomet ikke selv kalde *new* eller *delete* og undgår derved at "glemme" at deallokere ressourcer.

6.4.1 Stack vs. Heap

- **Stack** - Ressourcer deallokeres ved endt scope.
- **Heap** - Skal "manuelt" deallokeres.

6.4.2 Why we need RAII

Hvis vi har koden vist i udsnit 9. Heri kan vi risikere at funktionen *returner* før d og c bliver deallokeret og så dør baby.

```
1  void function() {
2       Client* c = new Client;
3       Data* d   = acuireData(c);
4
5       if(...)
6            return;
7
8       delete d;
9       delete d;
10 }
```

Code listing 9: Problem som skal løses af RAII

Men hvis vi wrapper d og c i deres eget objekt, vil de automatisk blive deallokeret når vi har ud af scope - om det så bliver på *return* eller når vi rammer linje 10.

6.5 Copy construction and the assignment operator

Disse skal være private i en RAII klassen da der ellers kan opstå situationer hvor to RAII-objekter peger på den samme ressource. Når den ene RAII nedlægges, så deallokeres ressourcen og tager så den andens med i faldet.

```
1 private:
2      SmartString(const SmartString&);
3      SmartString& operator=(const SmartString& other);
```

Code listing 10: Privat copy-ctor og assignment operator

6.6 What is the concept behind a Counted SmartPointer?

Når vi vil dele en ressource skal ikke ikke længere allokere ny plads i hukommelsen, vi kan bare dele pointeren. Ved hjælp af *counter_* vil hukommelsen først deallokeres når der ikke er flere som bruger pointeren.

I følge **rule of three** skal vi også lave vores egen udgave af copy-constructor og assignment operatoren. Det vigtige ved disse er at de også incrementere *counter_*, som holder styr på hvor mange referencer der findes til objektet.

I listing 11 kan copy-constructoren ses. Grunden til at *counter_* ikke incrementeres i denne metode er at assignment operatoren bruges og dén står for incrementeringen.

```
1 SmartString(const SmartString& other)
2 {
3      *this = other;
4 }
```

Code listing 11: SmartString::Copy-constructor

Videre i listing 12 findes vores egen udgave af assignment operatoren, hvor incrementering af *counter_* sker, samt tildeling af variabler.

```
1 SmartString& operator=(const SmartString& other)
2 {
3      str_ = other.str_;
4      counter_ = other.counter_;     // deling af ptr
5      (*counter_)++;                 // incrementering af ptr value
6 }
```

Code listing 12: SmartString::Assignment operator

Hver gang destructoren (ses i listing 13) kaldes vil *counter_* decrementeres og når der endelig ikke er flere der bruger pointeren (counteren), vil den deallokere elementerne.

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

    if(*counter_ == 0) {
        delete str_;
        delete counter_;
    }
}
```

Code listing 13: Counted SmartPointer::destructor

6.7 What is *boost :: shared_ptr <>* and how do you use it?

Shared_ptr <> er en class template, der indeholder en pointer til et dynamisk allokeret objekt. Der stilles garanti for at objektet der peges på slettes når den sidste *shared_ptr* der peger på det enten resettes eller destrueres. En *shared_ptr <>* er en counted smart pointer, hvilket vil sige at der inkrementeres en counter for hver gang objektet kopiers eller refereres.

6.7.1 Eksempel

Hvis en shared pointer, *p1* peger på variabel *a*, er dens counter = 1, hvis en anden shared pointer *p2*, sættes lig *p1* så er counter = 2. Hvis en tredje pointer også peger på *a* er counter lig 3 osv.

Kaldes populært en *reference counted pointer*.

6.7.2 How to use it

```
//Oprettelse af shared pointer
    boost::shared_ptr<T> myPointer(new T(InitValue));

//Assignment
    myPointer1 = myPointer2;

//Get references - returnerer antal referencer.
    myPointer.use_count()

//Reset - dekrementerer counter.
    myPointer.reset()
```

Code listing 14: Eksempel på brugen af en shared pointer

Dangling pointers kan opstå hvis to shared pointers har cirkulære referencer, og ingen har referencer til dem. Der kan bruges *weak_ptr <>* til at undgå dette.

boost::weak_ptr kan tildeles en shared_ptr, som den deler counter med. Det som weak pointeren peger på skal ikke være kritisk at have til rådighed, idet garbage collectoren sletter objektet der peges på, hvis der kun er en weak pointer der peger på det (det er her det kan bruges til at undgå cirkulære referencer)

lock() funktionen laver en shared pointer, som peger på det som weak pointeren peger på. Dette medfører at dennes hukommelse ikke deallokeres før vi går ud af det scope som den nye shared pointer findes i.

boost::scoped_ptr

- Boost version af RAII.

- Implementerer en smart pointer.
- Bundet til en funktion eller et objekt.
- Kan ikke kopieres - fordi copy-ctor og assignment operator er private!