Wiki »

Exercise 3 - Processes and Threads

Exercise 1 Creating Posix Threads ¶

Exercise 1.1 Fundamentals

I dette afsnit gennemgås kort de grundlæggende elementer i og egenskaber for programtråde.
Hvis man vil benytte funktionaliteten som tråde implementerer, er 1. step at lave en tråd. Dette gøres med funktionen *pthread_create*

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

pthread_create tager følgende argrumenter:

- En pointer til en *pthread_t* type, der indeholder ID'et for tråden.
- En pointer til en pthread_attr_t type, der indeholder attributter for tråden. Dvs. de indstillinger der oprettes for tråden(e.g. prioritet og om tråden skal være *detatched*).
  Hvis denne parameter gives *NULL* eller en *nullptr* oprettes tråden med default attributter.
- En funktionspointer, der peger på start-rutinen, dvs. den funktion som tråden kalder.
- En void pointer der modtager start-rutinens argumenter. Disse argumenter skal medgives som en void pointer-type, en void pointer kan dog pege på andre datatyper ved blot at give adressen som parameter. Desuden kan funktionen kun modtage et argument, dette kan dog omgåes ved at give en pointer til en struct som parameter.
  - OBS. Funktionsparameteren skal castes tilbage til dens respektive type inde i trådfunktionen. Det er programmørens opgave at sørge for at den medgivne parameter og den type der castes til stemmer overens.

Når der er oprettet en tråd kører tråden parallelt med main tråden. Dvs. pseudo-parallelt, i hvert fald for user-space tråde. Med dette forstås at processen på skift deler ud af dens tildelte processortid, til de enkelte tråde. trådene i programmet kører altså parallelt, indtil de termineres og enten *detatches* eller *joines* med en anden tråd. pthread er dog kernel-level threads, der kan benytte flere processor-kerner på samme tid, og derfor faktisk køre parallelt.

Som tidligere nævn modtager *pthread_create* en funktionspointer som argument. Denne pointer peger på den funktion som tråden skal eksekvere.
Denne funktion skal opfylde nogle krav. Den skal nemlig modtage præcis en void* som parameter og desuden returnere en void*. Dette virker upraktisk, da en void* jo ikke peger på noget. En void* kan dog sættes til at at pege på en vilkårlig datype, og så i funktionens body, castes til en pointer til den givne datatype.
Det samme gælder i og for sig for retur-værdien som også er en void*.
Funktionen *pthread_join* kan hente denne retur-værdi fra en tråd, vha. tråden *pthread_t* - ID.

```
#include <pthread.h>

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

Når *pthread_join* kaldes venter den tråd funktionen kaldes fra, på at tråden med det ID der medgives som parameter terminerer. Hvis tråden allerede er termineret sættes *void** retval* blot til at pege på returværdien for

den terminerede tråd.
Denne returværdi skal castes til en anden datatype for at der kan arbejdes med returværdien. Igen er det programmørens ansvar at caste til den rigtige type, og også returværdien kan være et *struct* der indeholder flere variable.

returværdien castes til en datatype som i nedenstående eksempel, hvor der castes til en integer:

```
#include <pthread.h>
       pthread_t tid;
       void *res;

       int err= int pthread_join(tid, &res);

       if(err!=0)
       {
           printf("Error\n");
       }
       int *returnVal =static_cast<int*> (res);
```

Det er også værd at bemærke at returværdien fra *pthread_join* gemmes i variablen *err* i ovenstående eksempel. Både *pthread_join* og *pthread_create* returnerer 0 hvis kaldene er succesfulde, og et positivt tal, er indikerer en evt. fejl.

Exercise 1.2 First threading program

I næste afsnit kommer et eksempel på et program der opretter og eksekverer en tråd parallelt med main tråden.

For at oprette en tråd, skrives først en tråd-funktion:

```
void* threadfunc(void *arg)
{
    std::string c=*static_cast<std::string*>(arg);
    std::cout<<c;
    return nullptr;

}
```

Ovenstående trådfunktion modtager en void pointer som argument. Denne castes til en string type inde i funktionen og skrives derefter ud med *cout*.

Derefter oprettes en *main* funktion, hvorfra tråden oprettes vha. *pthread_create* med en funktions-pointer til *threadfunc*.

```
int main(void)
{
    pthread_t thread;
    void* res;
    std::string c="Hello world.\n";
    pthread_create(&thread,NULL,threadfunc,&c);
    std::cout<<"print from main()\n";
    pthread_join(thread,&res);
    return 0;
}
```

Til sidst i programmet kaldes *pthread_join* der sørger for at *main* tråden venter på at trådfunktionen er termineret, så de kan "flettes sammen" igen. Hvis ikke *pthread_join* blev kaldt, ville det ikke væres sikkert at det argument tråden oprettes med ville blive skrevet ud på skærmen. Dette skyldes, at det ikke er til at vide om

tråden får CPU-tid til at eksekvere funktionen, før *main* terminerer, og alle tråde oprettet fra *main* derved også terminerer.

Programmet laves herefter eksekverbar via en makefile, og efterfølgnde eksekveres det i terminalen:

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe3/exe3_1$ ./bin/host/prog
print from main()
Hello world.
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe3/exe3_1$
```

Det ses her hvor det lykkedes at printe *hello world* i terminalen vi threads.

Exercise 2 Two threads

Nedenstående afsnit beskæftiger sig med et program der opretter to tråde, med hver deres ID som argument. De to tråde udskriver derefter "Hello" 10 gange hver, med deres ID som identifikation, af hvilken tråd der laver udskriften:

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
void* threadfunc(void *arg)
{
    int tid=*static_cast<int*> (arg);

    for(int count=1;count<=10;count++)
    {
        std::cout<<"Hello #"<<count<<" from thread "<<tid<<std::endl;
        sleep(1);
    }

    return nullptr;
}
int main(void)
{
    pthread_t tid1;
    pthread_t tid2;
    int no1=1;
    int no2=2;
    void* res;
    pthread_create(&tid1,NULL,threadfunc,&no1);
    //usleep(1000); sørger for at forskyde de to threads, så de ikke er "ready" på s
    usleep(1000);
    pthread_create(&tid2,NULL,threadfunc,&no2);
    pthread_join(tid1,&res);
    pthread_join(tid2,&res);
    return 0;
}
```

Jævnfør det forrige program kaldes 2x_pthread_join_ med de to trådes *pthread_t* som argumenter. Dette sørger for at de to tråde når at terminere inden main. Hvis man ikke havde kaldt join, var det ikke til at vide hvor meget, hvis noget, de to funktioner ville udskrive.

I dette program gives et ID med til de to trådfunktioner. Dette gøres ved at angive adressen(pointer) i den parameter i *pthread_join* der tager et argument til trådfunktionen. I trådfunktionen castes argumentet, der er en void pointer tilbage til en integer pointer, så funktionen kan læse sit ID.

Programmet laves herefter eksekverbar via en makefile, og efterfølgnde eksekveres det i terminalen:

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe3/exe3_2$ ./bin/host/prog
Hello #1 from thread 1
Hello #1 from thread 2
Hello #2 from thread 1
Hello #2 from thread 2
Hello #3 from thread 1
Hello #3 from thread 2
Hello #4 from thread 1
Hello #4 from thread 2
Hello #5 from thread 1
Hello #5 from thread 2
Hello #6 from thread 1
Hello #6 from thread 2
Hello #7 from thread 1
Hello #7 from thread 2
Hello #8 from thread 1
Hello #8 from thread 2
Hello #9 from thread 1
Hello #9 from thread 2
Hello #10 from thread 1
Hello #10 from thread 2
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe3/exe3_2$
```

Det ses her hvordan de to threads printer i par, og som ønsket overlappe de ikke hinanden.

Exercise 3 Sharing data between threads

I dette afsnit diskuteres et program der opretter to tråde. Den ene inkrementerer en variabel, som den anden læser. Begge tråde arbejder altså på samme variabel. Dette er nemt at implementere, da variablen er *passed by reference* til trådfunktionen. Derfor gives blot den samme variabel med som argument til de to trådfunktioner.

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
void* increment(void *arg)
{
    while(1)
    {
    unsigned int *inc=static_cast<unsigned int*>(arg);
    ++*inc;
    sleep(1);
    }
    return nullptr;
}
void* read(void *arg)
{
    while(1)
    {
    unsigned int *read=static_cast<unsigned int*>(arg);
    std::cout<<*read<<std::endl;
    sleep(1);
```

```
    }
    return nullptr;
}
int main(void)
{
    unsigned int shared=0;
    void* res;
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1,NULL,increment,&shared);
    usleep(1000);
    pthread_create(&tid2,NULL,read,&shared);
    pthread_join(tid1,&res);
    pthread_join(tid2,&res);
    return 0;
}
```

Der opstår dog et problem i dette program, eller i hvertfald har det ikke den tiltænkte funktionalitet.

```
59
60
61
62
63
64
65
65
```

Som set på ovenstående billede printes tallet 65 to gange. Dette skyldes igen at programmet ikke har kontrol over hvilken tråd der får CPU tid hvornår. Selvom de to tråde henholdsvis opdaterer og printer *shared* tilnærmelsesvis hvert sekund, kan man jo forestille sig at den ene tråd kan nå at udføre sin funktionalitet to gange i træk, hvorved der enten printes det samme tal to gange i træk, eller at tallet opdateres to gange i træk, så der kommer et hul i printrækken.
For delvist at modvirke dette har vi kaldt _usleep(1000); mellem oprettelsen af de to tråde. Derved bliver delayet i de to tråde lidt forskudt så ovenævnte problematik først opstår når programmet har kørt et stykke tid.

```
stud@stud-virtual-machine:~/Desktop/Courses/i3isu_f2020_beany_business/exe3/exe3_3/bin/nost$ ./prog
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

På billedet kan det ses at programmet printer korrekt for lave tal.

Exercise 4 Sharing a Vector class between threads

Dette afsnit omhandler et program der opretter et antal tråde mellem 1 og 100. Hver af disse tråde arbejder på en fælles vektor, der indeholder 10.000 elementer. Hver tråd kalder hvert sekund en funktion der fylder vektoren med et tal svarende til dets unikke ID. Derefter tester funktionen om det pågældende tal findes på alle pladser i vektoren.
Da trådfunktionen både skal have et tråd-ID og en pointer til den fælles vektor, oprettes et struct *"threadArg"* Der indeholder en Vector pointer og en integer. En pointer til et sådan struct gives med som parameter i *pthread_join*.

Til programmet benyttes en udleveret fil *Vector.hpp*

```
#include "Vector.hpp"
#include <pthread.h>
#include <iostream>
#include <unistd.h>
struct threadArg
{
    Vector* vecPtr;
    int tid;
};
void* threadFunc(void *arg)
{
    while(1)
    {
    threadArg*  tArg= static_cast<threadArg*> (arg);
    bool err=tArg->vecPtr->setAndTest(tArg->tid);
    if(!err)
    {
        std::cout<<"Error detected in thread with ID: "<<tArg->tid<<"\n";
    }
    sleep(1);
    }
    return nullptr;
}
int main(void)
{
    int n;
    std::cout<<"Indtast antal threads(1-100): ";
    std::cin>>n;
    pthread_t array[n];
    threadArg argArray[n];
    Vector *vec= new Vector;

    for(int i=0;i<n;++i)
    {
        argArray[i].tid=i+1;
        argArray[i].vecPtr=vec;
        pthread_create(&array[i],NULL,threadFunc,&argArray[i]);
    }
    void* res;
    for(int i=0;i<n;i++)
    {
        pthread_join(array[i],&res);
    }
```

```
    delete vec;
    return 0;
}
```

I dette program opstår der som forventet fejl. Dette skyldes jævnfør de forrige opgave at der ikke holdes kontrol med hvornår de enkelte tråde får CPU tid. Derfor er det ikke sikkert at en given tråd når at fylde vektoren med sit unikke ID, og derefter læse hele vektoren igennem. Dette ville kræve at tråden kontinuerligt havde CPU tid under hele forløbet.

```
stud@stud-virtual-machine:~/Desktop/Courses/i3isu_f2020_beany_business/exe3/exe3_4/bin/host$ ./prog
Indtast antal threads(1-100): 100
Error detected in thread with ID: 4
Error detected in thread with ID: 5
Error detected in thread with ID: 6
Error detected in thread with ID: 7
Error detected in thread with ID: 10
Error detected in thread with ID: 1
Error detected in thread with ID: 12
Error detected in thread with ID: 13
Error detected in thread with ID: 11
Error detected in thread with ID: 14
```

Jo flere tråde programmet køres med, jo oftere printes der fejl på skærmen. Dette skyldes at hver tråd kun fylder og tester vektoren 1 gang i sekundet. Derfor kan der komme op til antal vektorer fejl i sekundet.

Exercise 5 Tweaking parameters

For at lade brugeren vælge hvor ofte Hver tråd skal kalde *setAndTest* udvides structen med en integer *microseconds* som brugeren kan justere.

```
struct threadArg
{
    Vector* vecPtr;
    int tid;
    int microseconds;
};
```

Denne justering har indflydelse på programmets udskriver på følgende måde. Hver trådfunktion kalder *setAndTest* x gange i sekundet, hvor x er 1.000.000/_microseconds_.
Altså kan der i dette program potentiel komme (1.000.000/_microseconds_)*_antal tråde_. Altså er antalet af "fejl" proportionalt med antallet af tråde. og omvendt proportional med antallet af mikrosekunder. Ovenstående udledning antager at alle tråde når at få CPU tid til at gennemføre *setAndTest* mellem hvert kald af *usleep*. Selvom dette muligvis ikke er tilfældet, gælder proportionerne stadig.

Den sidste parameter der ændres på er antallet af elementer i vektoren.
For at undersøge dennes indvirkning, sættes antallet af elementer ned til først 1000, og senere 10.

```
Indtast antal threads(1-100): 10
Indtast antal mikrosekunder mellem eksekverind af setAndTest(): 100000
Error detected in thread with ID: Error detected in thread with ID: 8
3
Error detected in thread with ID: 4
Error detected in thread with ID: 3
Error detected in thread with ID: Error detected in thread with ID: 107

Error detected in thread with ID: 3
Error detected in thread with ID: Error detected in thread with ID: 3
4
Error detected in thread with ID: 8
Error detected in thread with ID: 4
```

Ovenstående billede viser fejlemeldinger når programmet kører med 10 tråde, 100.000 mikrosekunders delay og 10 elementer i vektoren.

Ved ændring ned til 1000 elementer i vektoren, sås umiddelbart ingen ændring, men måske lidt færre fejl. Hvilket også ville være forventet.
Da antal elementer blev sat ned til 10, så man dog tydeligt at der var markant færre fejl.
Som tidligere nævnt vil en tråd kun undgå en fejlmelding hvis den når gennem hele processen med at fylde og læse vektoren igennem, uden at miste CPU'en, eller at en anden tråd får fat i en anden CPU og forstyrrer den vej. Ved en mindre vektor, vil gennemløbningen tage kortere tid, og risikoen for at miste CPU'en derfor mindre, eller at en anden tråd får tid på en anden kerne og overskriver.

Exercise 6 Testing on target

Programmet fra Exercise 5 recompiles i denne øvelse, og exe-filen kopieres over på target(rpi).

Her undersøges det om problematikkerne fra programmet også eksisterer på Raspberrien, samt evt. afvigelser.

```
Indtast antal mikrosekunder mellem eksekverind af setAndTest(): 500000
Error detected in thread with ID: 1
Error detected in thread with ID: 4
Error detected in thread with ID: 2
Error detected in thread with ID: 3
Error detected in thread with ID: 6
Error detected in thread with ID: 9
Error detected in thread with ID: 8
Error detected in thread with ID: 3
Error detected in thread with ID: 10
Error detected in thread with ID: 8
Error detected in thread with ID: 7
Error detected in thread with ID: 1
Error detected in thread with ID: 3
Error detected in thread with ID: 1
Error detected in thread with ID: 5
Error detected in thread with ID: 2
Error detected in thread with ID: 1
Error detected in thread with ID: 10
Error detected in thread with ID: 9
Error detected in thread with ID: 3
Error detected in thread with ID: 8
Error detected in thread with ID: 10
Error detected in thread with ID: 2
Error detected in thread with ID: 8
Error detected in thread with ID: 7
```

Det konkluderes at samme problematik findes på Raspberrien.
Det observeres dog at forekomsten af fejl på Raspberrien, ved en ens indstilling af programmet på host og target, producerer færre fejl.

Dette må tilskrives at de to styresystemer kører på forskellige processorer. Derfor vil scheduleren også tildele CPU tid forskelligt.
Programmet vil derfor have forskellig adgang til CPU-tid, og vil derfor også tildele CPU-tid til trådene forskelligt fra hinanden. Desuden har raspberrien kun en kerne, der kan derfor ikke være flere tråde der har CPU tid på samme tidspunkt, hvilket mindsker risikoen for overskrivninger.

- **Tilføjet af Einar Jörgen Hansson for 3 måneder siden**

  Læringsmål::
  Objective 1 Det fundementale omkring tråde og oprettelse, smat fuldførelse
  I får forklaret rigtig godt hvordan det er man bruger pthread_create funktionen og hvilke parametre det er den medtager. I får ikke rigtig sagt nogle steder at grunden til vi klader pthread_join er at main bare bliver ved med at køre sin kode lige efter den har lavet tråden. Så hvis det var rigtig meget kode der skulle kaldes efter den havde lavet tråden så ville den komme til at køre færdig, det blot for at sikre os at vi får det hele med, for main er lige glade med de tråde den har oprette. I jeres første program med tråde der, retrunere i en nullptr, så ved i at der ikke er noget at skulle håndteres, så er jeres "res" variable ligegyldig og fylder egentlig bare unødvendigt. Punktet vurderes godkendt.
  Objective 2 Dele data mellem tråde
  Der mangler lidt forklaring i forhold til hvad i gør i jeres funktioner. I har forklaret fint i opgave et hvordan i caster frem og tilbage, der kunne i have referet til den forklaring. I jeres kommentar til koden i opgave to skriver i at fordi i kalder "usleep(1000)" så sørger i for at jeres tråpde ikke bliver klar samtidig, det er jo ikke det i gør i skuber bare kreationen af den ene tråd, som I også fik at se i opgave 3, så vil jeres løsning på lang sigt ikke holde. Jeg savner en bedre forklaring på en løsning til problemet for, det med at forksyde to trådes kreation virker ikke.
  Igen som i første funktion så er jeres retur værdi fra jeres funktioner en void pointer så der er igen grund til at oprette jeres "res" variabel, I kan blot skrive "nullptr" der hvor i skriver "&res" i join funktiuonen, så ved den bare at det er en nullpointer den skal have tilbage og hvis den ikke får det så siger den fejl.
  I jeres tekst til opgave tre der skriver i som om at i prøvede at lave programmet uden, "usleep(1000)", det kunne have været rigitg godt hvis I havde valgt at inkludere det med i jeres journal at, I rent faktisk viste hvordan det så ud før i implementerede jeres "fix". Punktet vurderes godkendt.

  Objective 3 Hvordan påvirker det programmet at køre med mange tråde
  Fin beskrivelse af, hvad der kan galt. Koden er god til at lave mange gennemkørsler efter hinanden til test. Der kunne godt være flere billeder, evt. lagt ind i rep, så man kunne se at I havde foretaget flere tests. Dette punkt vurderes til at være godkendt

  Objective 4 Er der forskel på at køre det på target
  Fine kommentarer og beskrivelser, det eneste jeg ville kunne sætte en finger på er at det godt kunne være inkluderet flere billeder på repo. Punktet vurderes til godkendt

  Feedback
  Rigtig god opgave hele vejen rundt, med gode kommentarer på billederne, og pæn opsætning.

  Must haves:
  Der er en makefile med, men den ligger ikke inde i hver mappe, så jeg kan ikke bare skrive make.
  I exe3_5 kunne jeg ikke få makefilen til at virke uden at ændre i den.
  Udover makefiles, der ikke altid ligger i de individuelle exercise mapper, så er alle relevante filer der.
  Conclusion
  I bund og grund rigtig fin opgave, lidt kortfattet men i får svaret på fint på opgaverne. Vi er desværre nødtil at sætte jer til ext review da der ikke er ordentlig styr på jeres repository.

increader.png (12,5 KB) Jens Nørby Kristensen, 2020-02-20 09:38

increader_err.png (1,55 KB) Jens Nørby Kristensen, 2020-02-20 09:46

exe3_4.png (21 KB) Jens Nørby Kristensen, 2020-02-20 10:12

RPI_vek.png (13,4 KB) Jens Nørby Kristensen, 2020-02-21 12:33

EX5.png (11,3 KB) Jens Nørby Kristensen, 2020-02-21 13:26

exe3_2_twothreads.png (46,4 KB) Magnus Nygaard Lund, 2020-02-27 17:25

exe3_1_threadstest.png (12,8 KB) Magnus Nygaard Lund, 2020-02-27 17:34