Contents

1 Introduktion

Denne rapport har til opgave at undersøge og bruge koncepterne: Mutex, Semafore, Producer Consumer problemet og teori omkring concurency control.

Brugen af koncepterne kommer iform af løsningen af 4 opgaver som hver har noget af koncepterne i sig. Dette gør at vi også får praktisk erfaring i at bruge disse koncepter hvor noget af vores erfaring kommer fra tidligere kurser som foreksempel Distriburede og mobile systemer - som omhandlede interaktionen mellem flere processer både lokalt og spredt omkring på netværk. Her havde vi om concurrency control og locking, teori der også kan bruges i parallel programmering, og er det som opgaverne går ud på.

2 Teori

2.1 Pthreads

Pthread er et standardiseret trådbibliotek som bruges til at oprette tråde sådan at man kan køre funktioner parrallelt, samle resultater fra barne tråde til forældretråden, og destruere tråde som blev skabt når man er færdig med at bruge dem.

2.2 Mutex

Mutex bruges når man har kritiske sectioner i sin kode og vil synkronisere sine processertråde. Hvis flere tråde bruger en værdi i den kritiske section, og ændre på den parrallelt, vides det ikke hvornår trådene ændre på værdien, dette kan skabe problemer og give forkerte læsninger også kaldet dirty reads. Ved brug af mutex kan man låse disse sectioner af så andre processer/tråde ikke kan tilgå samme section, når processen med låsen er færdig i den section vil mutex frigive låsen og lade andre processer/tråde tilgå den kritiske section.

[Comment: indsæt citatet fra hjemmesiden.]

2.3 Semaphores

Semaphores er en anden variation til at låse kritiske sectioner af med, de følger dog et andet princip de har en variabel med sig som betegner hvor mange pladser der er til processer/tråde kan køre simultant med hinanden. Når alle pladser er i brug vil semaphoren blocke adgangen til sectionen, og først give adgang når der er plads igen. Følgende kan forståes ved semaphorens metoder: wait og post Hvad de gør har en virkning på den variable som semaphoren har med sig, wait vil sænke variablen med 1 – reducere antallet a ledige pladser, og post vil hæve variablen med 1 – øge antallet af ledige pladser. Når variablen når 0 vil wait vente på et post.

[Comment: indsæt citatet fra hjemmesiden.]

2.4 Pthread_cond vs semaphores

Hvor semaphores blockerer kritiske sectioner. Vil Pthread_cond blockere på værdier den har brug for andet steds fra hvis man har 2 tråde hvor at tråd A gør brug af en global variable X som tråd B ændre kan man i tråd A vente på en at B har ændret præcis denne variable x. så foreskellen mellem pthread_cond og semaphore er at pthread_cond blocker på værdier fremfor sectioner, og derved kun blockere når det er højst nødvendigt for hvis tråd B er færdig før A så vil den bare fortsætte fordi variablen allerede er blevet ændret.

2.5 Concurenxy Control:

Når man har noget data som mange skal bruge på samme tid hvordan opnås dette sådan at de ændringer som bliver lavet ikke er forkerte når andre skal bruge dem? det er hvad concurrency control er. De tiltag man tager sådan at ens data er konsistent og korrekt. I concurency control har man transaktioner som er processer som udfører nogle handlinger der generelt betegnes som læse handlinger og skrive handlinger. Hvis man har to transaktioner som begge har adgang til resscource x, så kan en handling se sådan ud Read(x), Write(x,value). Hvis begge nu skulle lave en Write(x, 34) og Write(x,1000) hvilken skulle så være det der gjaldt dataen er ændret så den næste Process der laver en Read(x) kan ende med to resultater hvilket ikke er godt for at undgå dette skal man sørge for at ens transaktioner er serial eqiuvalent. dette kan gøres på mange måder en af dem er som Mutex hvor man låser den sektion der er data sensitiv af indtil man er sikker på at ændringerne har taget efekt og så er det first come first serve dette gør at man ikke får bad reads det er ikke alt omkring serial equvalence det er at for at de kan være det så kræver det at dataen ser ud efter kørsel af den ene transaktion og så den anden og omvendt og begge giver det samme slut resiultat.

2.6 Bankers Algorithm:

Er en resource alokerings algoritme som bruger matriser til at allokere ressourcer til processer og holder styr på hvor mange ressourcer af typen R en process har fået. I algoritmen er der 3 matrix´s med størrelsen n*m hvor n er antallet af processer, m er antallet af resource typer R og en vektor Available med længen m som angiver den maksimale mængde resoucer af en given type R der er tilgængelig, Max[nXm] er en matrise som holder styr på hvor mange resourser R en process kan modtage. Need[nXm] er en matrise som angiver algoritmen hvor mange resourcer en given process mangler for de specifikke typer R. Allocated[nXm] er en matrise som håndtere allokeringen af ressourcer på processerne på et givent tidspunkt. Safe state, not safe state er to ord som er en nøglerne til denne algoritme det er dette der afgør om der er nok resourcer til en proces kan udføre sit arbejde og afslutte dette vil være en safe state hvis en process kan acquire alle recsourcer den har brugfor og der er nok til at en anden process kan acquire rescourcer så er det en safe state. Hvis man antager at der er 5 resourcer af typen B og der er tre processer 1, 2, 3. 1 kommer med et request om at den vil have 3 B for at tilfredse den behov så vil bankers algoritme tjekke safe states ved at se om den kan allokere de resourcer hvis ja kigger den på tilstanden efter resourcen er blevet tildelt og ser om der er en proces der stadig kan få tilfredsstillet sit behov for rescourcer. og denne metode gør at man undgår deadlocks fordi der på intet tidspunkt er processer der venter på rescourcer som de aldrig kan få, mindst en process kan altid få nok rescourcer til at afslutte. Denne algoritme har dog nogle downsides som er ret markante, nemlig da man er nød til at vide på forhand hvilke og hvor mange ressourcer en process maksimum

skal bruge. Udover dette så at antage at en process skal frigive alle sine resourcer når den terminere hvilket i sigselv er vigtigt for algoritmen da man ikke kan sige levetiden på en given process kan man måske vente dage timer på at de ressourcer tilbage, dette er ikke praktisk for et realistisk system. sådan som vores verden er idag er det ikke logisk at have et statisk antal processore da verden er begyndt at bruge mange flere tråde sombliver oprettet og lukket igen og igen i løbet af et programs levetid.

3 Implementation

I dette afsnit er der beskrevet de tanker vi har gjort os omkring vores implementation af de fire opgaver. Bemærk at der ikke bliver beskrevet meget om testing, da det er i afsnittet Testing.

3.1 Sum(Sqrt)

Vi tager udgangspunkt i bogens implementation af et sum program, der benytter en tråd til at beregne summen fra 0 til et givent tal. Programmet er ret simpel siden den kun benytter en tråd, men det viser hvordan en tråd starter med `pthread` til at udføre en given funktion. Vores program er anderledes idet det skal benytte flere tråde, hvilket betyder at arbejdet skal opdeles.Desuden skal summen være af kvadratrod.

$$\sum_{i=0}^{N} \sqrt{i}$$

Programmet skal tage imod to typer af input tallet der skal summeres op til og et tal der angiver hvor mange tråde der skal køres. Ud fra en antagelse vi godt må gøre os, at resultatet af N/t er et heltal, hvor t er antallet af tråde. Med denne antagelse kan vi ligeligt opdele arbejdet mellem trådene.

Vi har lavet en `struct` til at give som argument, da `pthread_create(pthread_t *tid, pthread_attr_t *attr, void *method, void *param)` kun tager et parameter og vi har behov for at give to parametre, `n` og `sqrtsum`. Summen er dog det eneste tal der ændres på, mens `n`, N/t, bliver sat før nogen tråde starter og derfor kunne man i retrospekt godt have ladet være med at lave en `struct`.

Programmet bruger desuden en `mutex`, som den låser når der skal lægges til `sqrtsum`. Dette sikre os, at flere tråde ikke opdatere summen samtidig.

3.1.1 Programtid

Til at se program tiden har vi gjort brug af `<sys/time.h>` og dermed benytte de to `struct`, `timezone` og `timeval`, til at beregne tiden.

Vi har så kørt programmet med $N = 100000000$ og $t = 1, 4, 8, 16$. Derudover har vi kørt programmet i flere kerner, ved at bruge `parallel`, som skulle være hurtigere.

```
ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ ./sqrtsum 100000000 1
sqrtsum = 666666671666.567017
Total time(ms): 1690
```

```
ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ ./sqrtsum 100000000 4
sqrtsum = 666666671666.513916
Total time(ms): 751

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ ./sqrtsum 100000000 8
sqrtsum = 666666671666.464111
Total time(ms): 731

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ ./sqrtsum 100000000 16
sqrtsum = 666666671666.476440
Total time(ms): 727

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ parallel ./sqrtsum ::: 100000000 ::: 1
sqrtsum = 666666671666.567017
Total time(ms): 1685

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ parallel ./sqrtsum ::: 100000000 ::: 4
sqrtsum = 666666671666.513916
Total time(ms): 716

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ parallel ./sqrtsum ::: 100000000 ::: 8
sqrtsum = 666666671666.463989
Total time(ms): 716

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ parallel ./sqrtsum ::: 100000000 ::: 16
sqrtsum = 666666671666.476562
Total time(ms): 716
```

Der er en klar forskel mellem at bruge èn tråd eller flere tråde, men man kan ikke sige, at flere tråde giver et hurtigere program. Det kommer an på opgaven trådene har og i det her tilfælde er det at beregne kvadratrods summen, hvor belastningen afhænger af størrelsen på N. Den værdi der blev testet med viser at der ikke er den store forskel ved at bruge 4 eller 16 tråde, men det kunne der være hvis tallet nu var 10 gange større.

Når programmet bliver kørt med `parallel` er der ikke den store forskel når man benytter en tråd, hvilket er meningen. Når der tilgengæld benyttes flere tråde er den hurtigere, som antaget, men der er ingen forskel mellem at bruge 4 eller 16 tråde, hvilket er lidt overraskende. Vi har ingen konkret forklaring på dette tilfælde, men antager, at det skyldes operationens simplicitet.

3.2 Linked List

Linked list er en datastruktur der er bestående af noder med to værdier, deres element værdi(kunne f.eks. være en `string`, `int` osv.) og den næste node i listen.

Selve listen kender til sin første og sidste node samt sin længde. Når listen er tom, dvs. længden er nul, er der kun en node i den og det er `first` som peger på `NULL` som den næste node i listen. `first` kendes som root og kan/må ikke fjernes. Listen som vi skal implementere er en FIFO(first-in-first-out), hvilket betyder at når vi tilføjer en node ryger den bagerest i kæden og bliver til den sidste node. Omvendt når vi fjerner en node skal den første ikke root node fjernes.

Vi er allerede blevet givet strukturen af noden og listen, samt funktioner til at oprette lister og noder. Opgaven er at implementere tilføj og fjern funktioner til listen.

3.2.1 Tilføj og Fjern

Når man tilføjer en node skal man tage højde for om det er den første node efter root eller ej. Hvis dette er tilfældet, vil længden være 0 og dvs. at noden der tilføjes skal sættes som at være lig den root's næste node. Desuden er dette ikke blot den første node i listen, men også sidste node i listen så det skal den også sættes til.

```
if(l->len == 0)
{
    l->first->next = n;
    l->last = n;
}
```

Hvis det ikke er tilfældet er noden der skal tilføjes den nye sidste node i listen, så vi skal opdatere den sidste node.

```
else
{
    l->last->next = n;
    l->last = n;
}
```

Til sidst skal længden incrementes med 1.

Når vi skal fjerne en node skal vi opdatere root's næste node, da det nu skal være den næste nodes næste node. Desuden skal der tjekkes for om listen ikke er tom.

```
if(l->len > 0)
{
    n = l->first->next;
    l->first->next = n->next;
    l->len -= 1;
}
```

3.2.2 Problemer med flere tråde

Hvis flere tråde skal tilføje eller fjerne noder i listen kan der opstå problemer med integriteten af den data der er i listen. Listen behøver ikke nogen bestemt rækkefølge så vi kan godt tilføje noder tilfældigt, men hvis der bliver lavet et kald til tilføj via en tråd, så vil man gerne have at den node kommer ind i listen. Desuden hvis man lige har læst længden af listen, som ikke er tom, og man fjerner en node, er der ikke garanti for at listen ikke er tom på det tidspunkt denne tråd får lov til at udføre sin handling. Det resulterer i at du får en `NULL` node, og det kan i værste fald give en runtime error hvis programmet der kaldte på fjern afhang kraftigt af noden.

3.2.3 Mutex og Linked List

En god måde at sikre at flere tråde kan arbejde på listen samtidig er ved at lave et låse system med en nøgle, hvilket mutex er. Det betyder at når nøglen er fri kan man godt foretage operationer i listen, og hvis den ikke er så venter du indtil den bliver det.

Det man så skal overveje er hvor man skal sætte sine låse, er det brugeren af programmet der skal gøre det i sin tråd funktion? Det kunne godt lade sig gøre, men er ikke særlig hensigtsmæssig, som i forhold til hvis listen selv kunne håndtere dette. Det betyder så, at listen skal have en nøgle. Vi har tilføjet mutex i listens `struct` i `list.h`, da dette gør at man er sikret at en liste har en speciel lås.

Forrige sektion afklarede at det kun var nødvendigt at tilføje en låsemekanisme når man tilføjer eller fjerner noder, idet det er de kristiske sektioner i listen. Dette har vi gjort ved brug af `pthread_mutex_lock(l->mtx)` og `pthread_mutex_unlock(l->mtx)`.

3.3 Producer-Consumer problemet

Producer-Consumer problemet er et velkendt problem, hvor producenter producerer varer, som consumers konsumerer. Problemet ligger i at stoppe consumers med at konsumerer vare når der ikke er flere varer og ligeledes at stoppe producenter med at producerer varer når lageret er fyldt. Dette kan gøres ved brug af semaphores, da den netop har den funktionalitet der efterspørges i form af funktionerne `sem_wait()` og `sem_post()`. I programmet har vi implementeret en `struct PC`, der indeholder vores linked list, der fungerer som lager, og tre semaphores, `full` til at indikerer at der er vare i lageret, `empty` en til at indikerer at der er plads i lageret og `mutex` en der agerer som en mutex.

Programmet skal tage følgende fire input:

- Antallet af producers, `PRODUCERS`

- Antallet af consumers, `CONSUMERS`
- Størrelsen på lageret, `BUFFSIZE`
- Antallet af vare der skal produceres i alt, `PRODUCTS_IN_TOTAL`

Vi initialiserer vores semaphores således at `full` sættes til 0 og `empty` sættes til lagerets størrelse.

```
sem_init(&prodcons.full, 0, 0);
sem_init(&prodcons.empty, 0, BUFFSIZE);
sem_init(&prodcons.mutex, 0, 1);
```

Dette gøres grundet logikken i `sem_wait()` der formindsker værdien og `sem_post()` der forhøjer værdien. Dermed sætter vi producers til at holde øje med `empty` og sende et signal til `full` med post, og omvendt med consumers.

```
void *producer(void *param)
{
    .
    sem_wait(&empty);
    .
    sem_post(&full);
    .
}

void *consumer(void *param)
{
    .
    sem_wait(&full);
    .
    sem_post(&empty);
    .
}
```

For at holde styr på hvor mange produkter der produceres og konsumeres har vi to counters `p` og `c`, der bliver tjekket op med produkter i alt i deres respektive funktioners while løkker. Mutex semaphoren anvendes til at opdatere counters.

3.4 Banker's Algoritme

Til vores implementation af Banker's algoritme er vi blevet tildelt en fil med den grundliggende implementation af algoritmens struktur. I den givet implementation er der en `struct State` som består af de elementer Banker's algoritme kræver forklaret i Teori afsnittet. Det første der manglede at blive implementeret

i filen var allokeringen af hukommelse til `State`. Til dette anvender vi `malloc()` og `sizeof()` funktionerne. Med disse funktioner kan vi beregne det antal bytes i hukommelsen vi kommer til at anvende. Der hvor dette kan være problematisk er når der skal allokeres hukommelse til arrays og 2D-arrays. I et array skal man huske at allokere hukommelse i forhold til dens længde, så derfor ganges dette med `sizeof()` af typen. For 2D-arrays kræver det to skridt, hvor i det første skridt tildeles hukommelse af det andet array og i det andet skridt ved brug af en loop allokeres hukommelse til det første array.

```
s = (State *)malloc(sizeof(State));
s->resource = (int *)malloc(sizeof(int) * n);
s->available = (int *)malloc(sizeof(int) * n);
s->max = (int **)malloc(sizeof(int *) * n);
s->allocation = (int **)malloc(sizeof(int *) * n);
s->need = (int **)malloc(sizeof(int *) * n);

for(i = 0; i < m; i++)
{
    s->max[i] = (int *)malloc(sizeof(int) * m);
    s->allocation[i] = (int *)malloc(sizeof(int) * m);
    s->need[i] = (int *)malloc(sizeof(int) * m);
}
```

Det næste handlede om at tjekke om programmet var i en safe state eller en unsafe state. For at gøre dette lavede vi en metode `checksafety()`, der kopiere det kritiske indhold af staten og simulerer eksekvering af alle processerne. Hvis dette lykkedes er vi i en safe state, ellers er vi ikke og skal stoppe programmet.

Når dette er gjort laver programmet nogle forespørgsler på ressurser, som vi håndterer i `resource_request()`. I denne metode har vi fulgt teorien, med hensyn til at tjekke forespørgslen er inden for `need`, hvilket er et ekstra tjek eftersom når forespørgslen er beregnet ud fra `need` i `generate_request()`, men i tilfælde af at en bruger indtaster en forespørgsel er dette tjek godt at have. Derudover ser vi på om `available` ikke bliver overskredet af forespørgslen. Dette tjek er muligvis ikke nødvendigt, da en mulig simulation kan frigive nok ressourser på et andet tidspunkt. Til sidst hvis alt er gået godt, tildeler vi ressurserne, men vælger lave en kopi af de nuværende `available`, `need` og `allocation`. Dette gøres fordi vi efter tildelingen anvender `checksafety()` til at finde ud af om det er i orden, hvis ikke går vi tilbage til de tidligere værdier.

Til sidst implementeret vi funktionen `resource_release()`, der skal frigive ressurser fra en proces. Ligesom en bank tager vi imod det og opdatere `available`, `need` og `allocation`.

4 Testing

5 Reflektion

6 Konklusion