



IT-Universitetet
i København

OBLIGATORISK OPGAVE # 2 I BOSC

Operativsystemer og C

Author:

Tom Mørk Christensen

Jonas Elbækgaard Jørgensen

ITU-mail:

TMCH@ITU.DK

JELB@ITU.DK

9. oktober 2014

Indhold

1	Introduktion	3
2	Delopgave 1	
	– Multitrådet sum	3
2.1	Implementationen, overordnet set	3
2.2	De specifikke løsninger	3
2.2.1	Sum med multitråd	3
2.2.2	Sum af kvadratrod	4
2.2.3	Tidbesparelse ved multitrådsudregninger	4
2.3	Test	5
3	Delopgave 2	
	– Multitrådet FIFO buffer som kædet liste.	6
3.1	Implementationen, overordnet set	6
3.2	De specifikke løsninger	6
3.2.1	Tilføjelse af elementer	6
3.2.2	Fjernelse af elementer	6
3.2.3	Trådsikring	7
3.3	Fejl og mangler	7
3.4	Test	7
3.4.1	En anden tilgang til test	8
4	Delopgave 3	
	– Producer-Consumer med bounded buffer.	9
4.1	Implementationen, overordnet set	9
4.2	De specifikke løsninger	9
4.2.1	Producenter	9
4.2.2	Forbrugere	9
4.2.3	Tælle semafore	10
4.3	Fejl og mangler	10
4.4	Test	10
5	Delopgave 4	
	– Banker's algorithm til håndtering af deadlocks	12
5.1	Implementationen, overordnet set	12
5.2	De specifikke løsninger	12
5.2.1	Hukommelsesallokering	12
5.2.2	Resource request	12
5.2.3	Safety algorithm	13
5.2.4	Resource release	13
5.2.5	Islarger	13
5.2.6	Add- og subArr	14
5.2.7	KillProcesses	14
5.2.8	Trådsikring	14
5.3	Fejl og mangler	14
5.4	Test	14

A	Appendiks	16
A.1	Appendix for delopgave 1	16
A.1.1	Header file for delopgave 1 - <i>oo2_1.h</i>	16
A.1.2	Kildekode til delopgave 1 - <i>oo2_1.c</i>	16
A.2	Appendix for delopgave 2	18
A.2.1	Kildekode til delopgave 2 - <i>list.c</i>	18
A.2.2	Kildekode til test af delopgave 2 - <i>list/main.c</i>	20
A.3	Appendix for delopgave 3	22
A.3.1	Kildekode til delopgave 3 - <i>oo2_3.c</i>	22
A.4	Appendix for delopgave 4	25
A.4.1	Kildekode til delopgave 4 - <i>banker/banker.c</i>	25
A.4.2	Usikkert opstartesstadie - <i>banker/badstate.txt</i>	30
A.5	Makefiles	31
A.5.1	Makefile for delopgave 1 og 3	31
A.5.2	Makefile for delopgave 2	31
A.5.3	Makefile for delopgave 4	31

1 Introduktion

Denne rapport er udarbejdet af Tom Mørk Christensen og Jonas Elbækgaard Jørgensen i forbindelse med kurset Operativsystemer og C (BOSC) som besvarelse af 2. obligatoriske opgave.

Rapporten er inddelt i fire afsnit som hver omhandler en af de fire delopgaver i opgavebeskrivelsen.

2 Delopgave 1

– Multitrådet sum

Formålet med denne delopgave er at vise, at vi kan anvende (multiple) tråde til at foretage parallelle operationer, og på den måde udnytte multikerne hardware til at udføre beregningsopgaver hurtigere.

2.1 Implementationen, overordnet set

Med udgangspunkt i den udleverede kode, [1, s. 171], har vi konstrueret et program som kan beregne sumfunktionen.

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

Programmet tager 2 parametre: N (øvre grænse for summeringen) og t (antal tråde programmet skal starte).

Vi benytter en *struct Sumjob* til at sende parametre til trådene.

Programmet udskriver resultatet til terminalen efter alle beregninger er foretaget.

2.2 De specifikke løsninger

I første omgang er det værd at nævne, at ændringerne til den udleverede kode har betydet, at den kontrol som programmet foretager af input også skulle ændres. Da programmet tager en ekstra parameter, og at denne parameter skal være et positivt heltal er reflekteret med en ekstra *if*-statement og en ændring *if*. *check* af antal parametre.

2.2.1 Sum med multitråd

Da brugeren angiver antallet af tråde som inputparameter til programmet, er det nødvendigt at allokere hukommelse til trådenes id (*pthread_t*). Det er ligeledes nødvendigt at allokere hukommelse til de *Sumjob* trådene starter. Både vores *pthread_t* og *Sumjob* oprettes i arrays. Der beregnes desuden det interval af heltal, som hver tråd skal summere.

Ved at iterere (i en for-løkke) igennem vore *Sumjob* sætter vi (baseret på intervallet) det første og sidste heltal som skal summeres af hver tråd, hvorefter vi starter metoden *runner* i tråden med *Sumjob*'et som parameter. Herefter starter vi en ny for-løkke, hvori vi kalder *pthread_join* for hver tråd-id, hvormed vi afventer trådene bliver færdige. Til sidst frigiver vi (for en god ordens skyld) hukommelsen allokeret til tråde og jobs. Til sidst printer vi resultatet til terminalen.

2.2.2 Sum af kvadratrod

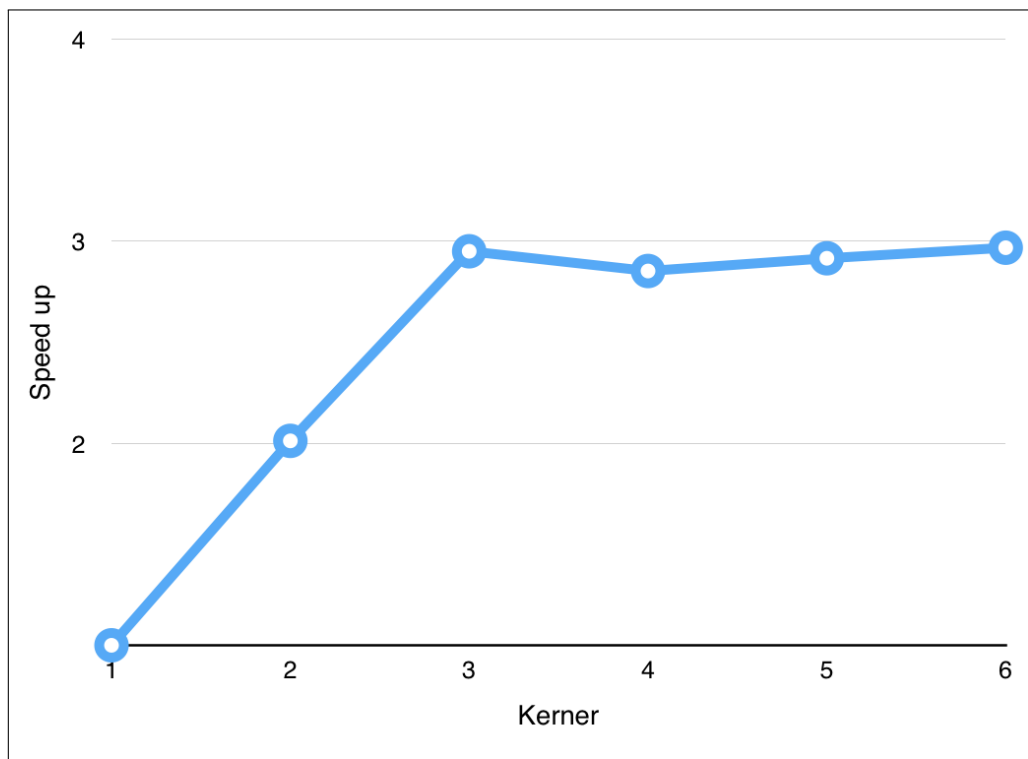
For at beregne summen af kvadratroden *sumsqrt* er det nødvendigt at regne i flydende tal i stedet for heltal. Det betyder dog ingen forskel for trådene som sådan: metoden *runner* som foretager summeringen arbejder med de samme input (intervallet), men benytter nu matematikbiblioteket *math.h* til at beregne den flydende talværdi af kvadratroden af heltallet. Der itereres igennem en for-løkke, kvadratroden beregnes og værdien summeres i en lokal variabel. Når denne iteration er færdig, lægges resultatet til en variabel defineret i mainmetoden, hvorefter tråden exit'er.

2.2.3 Tidbesparelse ved multitrådsudregninger

For at påvise, at programmer som dette kan drage fordel af at benytte flere kerner, har vi gjort følgende forsøg:

Ved at lade programmet beregne summen af $N = 100.000.000$ ved brug af et varierende antal tråde, kan vi se en tydelig forskel på den tid det tager systemet at returnere resultatet. Med 1 tråde tog summeringen ca. 8,4 sekunder realtid men med 2 tråde tog den kun 4.2 sekunder, 3 tråde 2,9 sekunder og med 4 tråde ligeledes 2,9 sekunder. Dette skyldes, at den maskine vi testede programmet på kun havde mulighed for at afsætte 3 kerne til den virtuelle maskine, så det giver god mening at vi ikke fik mere ud af at benytte flere tråde end vi havde fysiske kerner. Test med flere end 4 kerne gav da også det samme resultat: ingen forbedring på køretiden.

Det skal siges, at beregningstiden lå på ca. 8,4 sekunder under alle tests, hvilket også stemmer overens med vores forventninger, i det den faktiske beregningstid ikke er gået ned: den er blot spredt ud over flere kerne. Grafen herunder viser den faktor som beregningstiden faldt med, ad Y-aksen, som funktion af antallet af kerner, ad X-aksen.



Figur 1: Graf over test resultater.

2.3 Test

Vi testede programmet i stadier for løbende at sikre at alt fungere som det skulle. Vi foretog derfor test efter vi mente at vi havde vores multitråds implementation med heltal på plads, og igen efter vi havde implementationen med kvadratrodssummering.

Tests af programmet blev udført ved at først benytte talsæt vi let kunne kontrollere: f.eks. skulle $N = 10$ give et resultat på 55 (ved heltals summering). Ved at benytte flere eller færre tråde, og ændre størrelsen N var det let at påvise hvornår programmet fungerede.

Det samme gjorde sig gældende for kvadratsrods summeringen, dog med andre tal. F.eks. giver $N = 9$ ca. 19,306, mens $N = 8$ må give et resultat præcis 3 mindre.

I kombination med tidstagning af store regneopgave (som tidligere nævnt) samt printlines i *runner* funktionen var det muligt at demonstrere, at der blev oprettet multiple tråde, og at programmet benyttede dem til at foretage beregningerne.

Til at indsamle test resultater har vi skrevet den følgende shell command, som skriver resultat af testen til enden af filen *timing*.

```
1 { time ./sum 100000000 1 >> timing; } 2>> timing
```

3 Delopgave 2

– Multitrådet FIFO buffer som kædet liste.

Denne delopgave har til formål at vise, at vi har forståelse for hvordan hægtedelister fungerer samt hvordan disse kan implementeres i C. Udover at demonstrerer vores forståelse for hægtedelister tjener opgaven det formål at demonstrer hvordan nogle af risiciene ved flertrådet systemer kan forebygges.

3.1 Implementationen, overordnet set

Implementationen af vores løsning til denne delopgave begrænser sig til klassen `list.c`. I forbindelse med test af vores implementation har vi også ændret filen `main.c`, hvor vi har introduceret to nye funktioner `test()` og `add(void *param)`. Vi vil beskrive implementationen af disse funktioner samt deres formål i afsnit 3.4 på side 7.

I `list.c` kan vores implementation splittes i tre dele.

1. Tilføjelse af elementer til listen. Implementationen af denne funktionalitet begrænser sig til funktionen `list_add(List *l, Node *n)` og har til formål at hægte det nye elemnt `n` bagerst på listen.
2. Fjernelse af elementer fra listen. Implementationen af denne funktionalitet begrænser sig til funktionen `list_remove(List *l)` og har til formål at fjerne og returnere de foreste element i den hægtede liste.
3. Trådsikring af listen, ved hjælp af gensidig udelukkelse. I modsætning til de to foregående dele er denne del spredt over flere af implementationens funktioner. Dette skyldes at mutexen skal oprettes sammen med listen, men låses og frigives i forbindelse med metodekald.

3.2 De specifikke løsninger

3.2.1 Tilføjelse af elementer

For at tilføje et element til en liste skal vi bruge en pointer til både listen og til det element som skal hægtes på listen. Når et element tilføjes til en liste hægtes dette på listens bagerste element således at de ældste elementer altid ligger tættest på listens start. I vores implementation ændre vi listens bagerste element således at det peger på det nye element. Her efter undersøger vi om den knude der tilføjes til listen indgår i en sekvens af hægtede knuder. Er dette tilfældet traversere vi listen af knuder igennem indtil vi når det sidste element. Når vi har nået det sidste element ændre vi den hægtede listes `last` pointer således at denne nu også peger på det nye slut element. Herefter opdatere vi listens længde med antallet af nye elementer.

3.2.2 Fjernelse af elementer

Ved fjernelse af elementer starter vi med at gennem den hægtede listes `rod` element til en variabel for at simplificere arbejds processen. Da roden ikke må fjernes tjekker vi om `next` pointeren på roden peger på en værdi. Er dette tilfældet ændres rodens `next` pointer så den perger på det element der ligger efter elementet der skal fjernes, hvorefter længden reduceres med en og det fjernede element returneres. Peger rodens `next` pointer ikke på en værdi er listen tom og værdien `NULL` returneres.

3.2.3 Trådsikring

For at introducere trådsikring til vores program har vi gjort brug af én `mutex`. Mutexen har til formål at sikre at kun en tråd kan eksekvere en given kommando ad gangen. Dette fungerer ved at den tråd der kun den tråd der i en given situation har låst mutexen kan eksekvere den efterfølgende kode, mens andre tråder der ønsker at eksekvere den samme kode, eller eksekvere anden kode, som også er beskyttet af mutexen, bliver holdt tilbage indtil låsen på mutexen fjernes og en ny tråd kan låse den.

I vores implementation har vi gjort brug af én mutex som bliver brugt både når tråde ønsker at tilføje eller fjerne et element fra listen. Det gør sig gældende for både `list_add` og `list_remove` funktionen at når en tråd træder ind i funktionen skal den låse mutexen før end at den får lov at tilføje eller fjerne noget, og tilsvarende frigive mutexen når operationen er udført. Dette betyder at hvis en anden tråd er i gang med at redigere listen, må tråden vente på at mutexen bliver frigivet før den kan udføre sine ændringer på listen.

Mutexen bliver oprettet sammen med listen.

3.3 Fejl og mangler

Selvom at vi i vores implementation gør brug af mutex til at sikre at tråde ikke tilgår listens elementer samtidigt, og at brugen af kun én mutex skulle forhindre eventuelle deadlocks i at opstå er der stadig en måde hvorpå vores system ville gå i knæ. Hvis en bruger tilføjer en sekvens af knuder som danner en løkke vil vores implementation træde ind i en uendelig løkke da den ved indsættelse af nye elementer læder efter det sidste element i rækken af elementer.

Vi har gjort os nogle overvejelser om hvordan denne situation kunne afværges og er fundet frem til at den bedste løsning ville være at gemme det foreste af de nye elementer til en variabel og så tjekke at ingen af de efterfølgende elementer peger tilbage på dette element. Denne løsning til sikre at en liste hvor bagerste element tilbage på det forreste ville blive opdaget, men samtidig udelukker det ikke situationer hvor fx bagerste element peger tilbage på det andet element i listen. Den eneste måde hvorpå vi har kunnet se at det er muligt at afvære en sådan situation, hvor et element peger tilbage på et tidligere, er ved at gemme alle nye elementer til variabler og sikre at ingen af disse bliver peget på senere i listen. Vi har valgt ikke at give os i kast med at implementere en sådan løsning da det ikke virker til at være opgavens primære mål, men det er bestemt en situation der er værd at overveje hvis programmet skal anvendes i større stil.

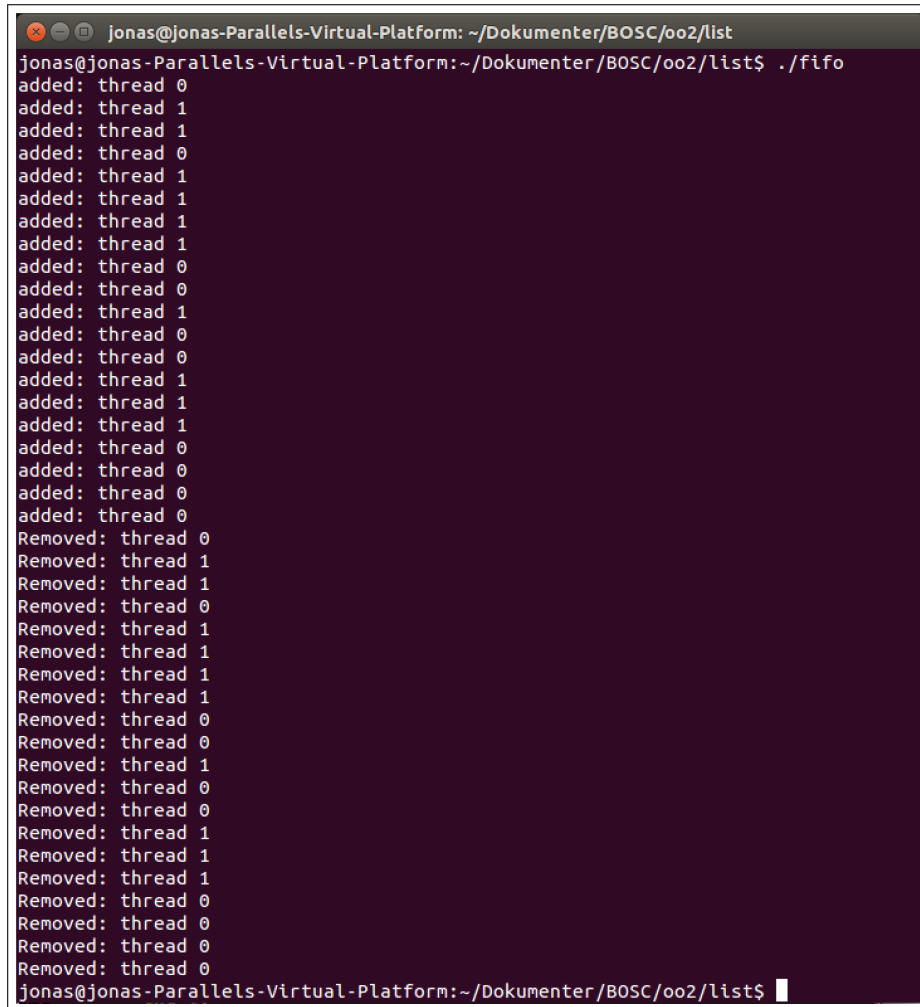
En anden ting som fangede vores opmærksomhed under implementationen af denne opgave var muligheden for at gøre brug af 2 mutex objekter. Så længe at den hægtede liste ikke er tom der er i og forsig ikke noget i vejen for at en process tilføjer et element mens en anden fjerner et element da de arbejder på hver sin del af listen og derfor ikke skal bruge de samme objekter. Den eneste situation hvor dette ikke er tilfældet er hvis listen er tom eller kun holder et element, da begge processer i såfald ville arbejde på listens rod og derved risikere at forårsage en race-condition. En tilgang til dette kunne være kun at låse `list_add` og `list_remove` enkeltvis så længe at listen holder to eller flere elementer, men låse begge funktioner hvis listen er tom eller kun holder et element.

3.4 Test

For at teste at vores implementation kan anvendes af flertrådet programmer har vi skrevet en test funktion i `main.c`. Funktionene opretter to tråde som begge har tilopgave tilføje ti elementer til listen. For at sikre at begge tråde forsøger at tilgå listen samtidig har til tilføjet

et kald til funktionen `sleep(1)` som får den aktuelle tråd til at sove i et sekund.

For at kontrollere at trådene ikke har overskrevet hinandens ændringer afsluttes funktionen med at printe alle elementerne i den hægtede liste til terminalen. Af figur 2 ses det at alle 20 elementer er blevet tilføjet til listen, og at trådene på skift har haft adgang til listen.



```
jonas@jonas-Parallels-Virtual-Platform: ~/Dokumenter/BOSC/oo2/list
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo2/list$ ./fifo
added: thread 0
added: thread 1
added: thread 1
added: thread 0
added: thread 1
added: thread 1
added: thread 1
added: thread 1
added: thread 0
added: thread 0
added: thread 1
added: thread 0
added: thread 0
added: thread 1
added: thread 1
added: thread 1
added: thread 0
added: thread 0
added: thread 0
added: thread 0
Removed: thread 0
Removed: thread 1
Removed: thread 1
Removed: thread 0
Removed: thread 1
Removed: thread 1
Removed: thread 1
Removed: thread 1
Removed: thread 0
Removed: thread 0
Removed: thread 1
Removed: thread 0
Removed: thread 0
Removed: thread 1
Removed: thread 1
Removed: thread 0
Removed: thread 0
Removed: thread 0
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo2/list$
```

Figur 2: Udskrift af test resultat til terminalen.

3.4.1 En anden tilgang til test

En anden mulig måde at teste hvorvidt vores hægtedeleliste understøtter at flere tråde forsøger at tilgå den er ved brugen af funktionskaldet `pthread_mutex_trylock()` som returnerer en fejlmeddelelse hvis den givne mutex allerede låst. På denne måde kan man få de enkelte tråde til at printe en besked til terminalen når de er tvunget til at vente på at låsen på mutexen bliver frigivet.

4 Delopgave 3

– Producer-Consumer med bounded buffer.

Formålet med denne delopgave er at demonstrere vores kendskab til tælle-semafore samt deres anvendelse i praksis. Opgaver baserer sig på et producent/forbruger system, hvor en gruppe producenter genererer en række produkter som en gruppe forbrugere benytter.

Løsningen til denne delopgave er implementeret i filen `002_3.c` og gør brug af implementation af den hægtede liste fra delopgave 2, se afsnit 3 side 6.

4.1 Implementationen, overordnet set

Implementationen baserer sig på at en gruppe producent-tråde, af typen `pthread_t`, oprettes sammen med en gruppe forbruger-tråde. Når producent-trådene er oprettede begynder de hver især at producere elementer og tilføje disse til en fælles buffer. Bufferen er en hægtet liste fra delopgave 2, som forbruger-trådene læser elementer ud af, efterhånden som de bliver produceret. Gruppernes størrelse, bufferens kapacitet samt mængden af elementer der skal produceres er baseret på brugerinput, som gives med når programmet startes.

For at sikre at trådene ikke foresager **race-conditions** gør vi brug af en række binære tælle-semafore, som har tilformål at låse forskellige data felter således at kun en tråd kan have adgang til dette ad gangen. Derudover bruger vi også tælle-semafore til sikre at bufferens kapacitet ikke overskrides.

Når det angivne antal produkter er produceret og et tilsvarende antal er forbrugt af forbruger-trådene skriver programmet en status meddelelse til terminalen, hvorefter programmet terminere.

4.2 De specifikke løsninger

4.2.1 Producenter

Producent funktionen er implementeret med en **while-løkke** som bliver kørt så længe der skal produceres nye elementer. Til at finde ud af om der skal produceres flere elementer tjekker tråden hvor mange elementer der er produceret, samt hvor mange der ligger i bufferen og sikre sig at summen af disse to tal ikke overstiger den totale mængde af elementer der skal produceres. Hvis alle elementer er produceret bryder tråden ud af løkken og lukker ned.

Det første der sker inde i løkken er at tråden sættes til at vente i omkring 1 sekund. Herefter øger den en tæller som holder styr på antallet af producerede elementer, hvorefter den opretter det element der skal tilføjes til bufferen. I vores implementation er elementerne tekststrenger som holder information om hvilket produktionsnummer elementet har. Når elementet er genereret og tråden har fået adgang til bufferen hægter den elementet på listen hvorefter den frigiver bufferen igen. Inden den skriver information om det nygenererede element til terminalen.

4.2.2 Forbrugere

Forbruger funktionen er implementeret med i still med producent funktionen. Den eneste forskel er at denne ikke opretter nye elementer, men i stedet henter eksisterende elementer ud af bufferen. Når en forbruger-tråd har hentet et nyt element og frigivet bufferen printer den information om dette element til terminalen. Således skulle terminalen efter endt programkørsel holde vise hvilke elementer der er produceret af hvilke processer og brugt af hvilke forbrugere.

4.2.3 Tælle semafore

Vi bruger tælle semafore flere steder i vores system. Først og fremmest bruger vi en binær tælle semafor til at sikre at kun en tråd har adgang til bufferen ad gangen. Dette er for at sikre mod race-conditions, hvor flere tråde forsøger at tilføje eller fjerne elementer samtidig. På samme måde gør vi brug af binære tælle semafore til at sikre at der ikke er flere tråde som forsøger at opdatere antallet af producerede/brugte elementer samtidigt. Alt i alt har bruer vi fire binære semafore til at forebygge race-conditions.

Udover de nævnte semafore bruger vi yderligere to tælle semafore til at holde styr på henholdsvis hvor mange elementer der er i bufferen, samt hvor mange flere elementer der er plads til. Hvergang en producent forsøger at tilføje et element til bufferen skal den først reducere værdien på den semafor som holder styr på antallet af tomme pladser i bufferen med én. Hvis denne har værdien 0 må tråden vente på at en forbruger-tråd læser et element ud fra bufferen og hæver værdien af tomme pladser med en. Når producent-tråden har tilføjet elementet til bufferen øger den værdien for den semafor som holder styr på antallet af elementer i bufferen. På tilsvarende vis skal en forbruger-tråd reducere semaforen der holder antallet af elementer inden den kan få lov at fjerne et. Er værdien af denne 0 må tråden vente til den kan få en værdi der er større.

4.3 Fejl og mangler

Hvis programmet startes med mindre end de fire krævede argumenter vil det terminere med en segmentfejl. Dette er hverken en særlig køn eller bruger venlig løsning da en bruger, som ikke kender systemet hurtigt vil kunne få en opfattelse af at det ikke virker efter hensigten. Her ville det have klædt programmet at det var implementeret således at hvis der blev givet for få eller for mange parameter med ved opstarte ville systemet printe en besked til brugeren of eventuelt bede om at få specificeret input parameterne.

4.4 Test

For at teste at programmet virker som forventet har vi kørt den en række gange med forskellige input.

Store input: En af de måder hvorpå vi har testet systemet er ved at sætte det til at producere en stor mængde produkter og kontrolleret at systemet ikke fejler over tid.

Flest producenter: Vi har også test systemet ved at køre de med indstillinger hvor der har været væsentligt flere producenter end forbrugere og omvendt. Ved disse test har vi haft fokus på at bufferens kapacitet ikke overskrides samt at den type tråde som er i overtal ikke begynder at lukke ned før end at alle elementer er produceret/forbrugt.

For mange producenter: En tredje måde vi har test systemet på er ved at lave flere producenter end der skulle genereres elementer. Her har vores fokus været at der kun blev genereret det antal elementer der skulle produceres og at de overflødige producenter ikke producerer noget.

Negative input: Den sidste måde hvorpå vi har teste vores system er ved at angive negative værdier. Her har formålet været at sikre at systemet håndterer dårlige bruger input som forventet.

```
jonas@jonas-Parallels-Virtual-Platform: ~/Dokumenter/BOSC/oo2
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo2$ ./procon 2 3 4 10
You have requested:
  2 producers producing a total of    10 items.
  3 consumers and a buffer containing up to    4 elements
Producer  0 produced   Item_0. Items in buffer:    1 (out of  4)
Consumer  2 consumed   Item_0. Items in buffer:    0 (out of  4)
Producer  1 produced   Item_1. Items in buffer:    1 (out of  4)
Consumer  1 consumed   Item_1. Items in buffer:    0 (out of  4)
Producer  1 produced   Item_2. Items in buffer:    1 (out of  4)
Consumer  1 consumed   Item_2. Items in buffer:    0 (out of  4)
Producer  1 produced   Item_3. Items in buffer:    1 (out of  4)
Consumer  0 consumed   Item_3. Items in buffer:    0 (out of  4)
Producer  0 produced   Item_4. Items in buffer:    1 (out of  4)
Consumer  2 consumed   Item_4. Items in buffer:    0 (out of  4)
Producer  1 produced   Item_5. Items in buffer:    1 (out of  4)
Consumer  0 consumed   Item_5. Items in buffer:    0 (out of  4)
Producer  0 produced   Item_6. Items in buffer:    1 (out of  4)
Consumer  2 consumed   Item_6. Items in buffer:    0 (out of  4)
Producer  1 produced   Item_7. Items in buffer:    1 (out of  4)
Consumer  2 consumed   Item_7. Items in buffer:    0 (out of  4)
Producer  0 produced   Item_8. Items in buffer:    1 (out of  4)
Consumer  1 consumed   Item_8. Items in buffer:    0 (out of  4)
Producer  1 produced   Item_9. Items in buffer:    1 (out of  4)
Consumer  0 consumed   Item_9. Items in buffer:    0 (out of  4)

Result:
  10 items has been produced.
  10 items has been consumed.
   0 elements are left in buffer.
jonas@jonas-Parallels-Virtual-Platform:~/Dokumenter/BOSC/oo2$
```

Figur 3: Udskrift af eksekvering med 2 producenter, 3 forbrugere, buffer kapacitet på 4 og et total antal elementer på 10.

5 Delopgave 4

– Banker's algorithm til håndtering af deadlocks

Formålet med denne opgave er vise forståelse for Edsger W. Dijkstras Banker's algorithm samt at kunne implementere denne som sikring mod deadlocks i et flertrådet program.

5.1 Implementationen, overordnet set

Vi tager udgangspunkt i den udleverede kode. For at løse de stillede opgaver har vi foretaget ændringer i main metoden (for at allokere hukommelse til tilstands-structen), samt udfyldt de tomme metoder `int resource_request(int i, int *request)` og `void resource_release(int i, int *request)`.

Vi har derudover oprettet hjælpermetoderne `int safe_state(State *s), int islarger(int *check, int *match, int size), void addArr(int *dest, int *add, int length), void subArr(int *dest, int *sub, int length)` og `void killProcesses()`. Brugen af disse vil blive forklaret nedenfor. Endelig har vi implementeret en funktionen `void printArr(int *arr, int length)` til printning af arrays.

5.2 De specifikke løsninger

5.2.1 Hukommelsesallokering

I starten af `main` metoden allokere vi hukommelsen til tilstands-structen. Efter programmet har modtaget input om antal processer og resourcetyper (m og n) ved vi hvor meget hukommelse der skal afsættes til vektorerne og matricerne. Allokeringen foregår ved at der først allokeres plads til state elementet. Herefter kan de to enkelt arrays, `resource` og `available`, allokeres til at have plads til m elementer. De 2-dimensionelle arrays allokeres ved først at allokere hukommelse til listens ene dimension, peger derefter til heltalspegere, med længde m , og herefter allokere et array med n elementer for hver plads i den yderste dimension.

```
145  /* Allocate memory for state */
146  s = (State *) malloc(sizeof(State));
147  s->resource = (int *) malloc(n * sizeof(int));
148  s->available = (int *) malloc(n * sizeof(int));
149  s->max = (int **) malloc(m * sizeof(int *));
150  s->allocation = (int **) malloc(m * sizeof(int *));
151  s->need = (int **) malloc(m * sizeof(int *));
152
153  for (i = 0; i < m; i++) {
154      s->max[i] = (int *) malloc(n * sizeof(int));
155      s->allocation[i] = (int *) malloc(n * sizeof(int));
156      s->need[i] = (int *) malloc(n * sizeof(int));
157  }
```

Listing 1: Allokering af resourcer i banker.c.

Efter hukommelsesallokeringen kan resten af dataet indlæses: dette tager den udleverede kode sig af.

5.2.2 Resource request

Efter at dataet er indlæst i structen checker vi om det indlæst data repræsenterer et safe state. Er dette ikke tilfældet frigives den allokerede hukommelse og programmet terminere. Dette check foretages af hjælpefunktionen `int safe_state(State *s)`, se afsnit 5.2.3 for yderligere information, og såfremt state'et er safe kører programmet videre: der oprettes en tråd hvori

vi kører den udleverede metode `void *process_thread(void *param)`. Denne funktion generere en tilfældig request via den udleverede funktion `void generate_request(int i, int *request)`, og denne request bruges nu til kalde metoden `int resource_request(int i, int *request)`.

Her checkes først at den generede request ikke overskrider processens `need`. Er dette tilfældet terminere programmet med en fejlmeddelelse om at en proces har overskredet dets forventede behov. Alternativt ændres state, således at det representere tilstanden efter at requestet er accepteret. Dette state checkes af `safe_state` funktionen for at verificere hvorvidt der er tale om et safe eller unsafe state.

Er det nye state safe, bibeholdes state'et og funktionen returnere 1. Er der derimod tale om et unsafe state rulles ændringerne tilbage og funktionen returnere 0, og processen må vente til senere og lave en ny foresøgelse.

5.2.3 Safety algorithm

Safety algoritmen er defineret i sin egen funktion, tager en reference til et state som parameter, og returnerer 1 eller 0 alt efter om det givne state er safe eller unsafe.

Funktionen følger algoritmen som defineret i [1], og opretter sine egne `finish`- og `work` arrays, samt allokerer hukommelse til disse.

Algoritmen leder efter en process som vil kunne få tildelt de resourcer den mangler ud fra det aktuelt tilgængelige resurcer. Findes dette ikke vil algoritmen returnere 0, som indikere at state'et er unsafe. Hvis der findes en sådan proces simulere algoritmen at processen terminere og frigiver sine resourcer hvorefter ved at opdatere `work` og processen markeres som afsluttet. Herefter leder algoritmen de resterende processer igennem efter en som kan terminere med udgangspunkt i de opdaterede resourcer. Denne process gentages indtil enten alle processer er markeret som afsluttet og eller intil det ikke er muligt at finde en proces der vil kunne terminere med de tilgængelige resourcer. Findes der er sekvens af processer det tillader at alle processer kan terminere returneres 1, hvilket indikere at state'et er safe.

5.2.4 Resource release

Efter at en process har godkendt en request genereres der en release request, som har til formål at simulere frigivelsen af resourcer. Det er metoden `void generate_release(int i, int *request)` som genere requesten og når dette er gjort sendes denne til metoden `void resource_release(int i, int *request)`.

`Void resource_release(int i, int *request)` kontrollerer at requesten ikke overskrider mængden af resourcer som er allokeret til processen. Er dette tilfældet frigives resourcerne hvorefter processen generere en ny resource request og processen starter forfra. Overskrider release requesten de allokerede resourcer terminere programmet med en fejlmeddelelse herom.

5.2.5 Islarger

Funktionen `int islarger(int *check, int *match, int size)` er en hjælpefunktion som har til formål at tjekke hvorvidt et array er større end et andet, jf. [1, s. 331] . Dette sker ved at hvert element i `check` er større end det tilsvarende element i `match`. Dette er et tjek der bruges flere steder i vores implementation og derfor har vi valgt at oprette en funktion til at udføre det.

5.2.6 Add- og subArr

Funktionerne `void addArr(int *dest, int *add, int length)` og `void subArr(int *dest, int *sub, int length)` har til formål at henholdsvis lægge to arrays sammen og trække to arrays fra hinanden. Dette sker ved at hvert element i arrayet `dest` forøges eller forminskes med værdien i `add/sub`.

5.2.7 KillProcesses

`Void killProcesses()` er funktionen som sikre at systemet lukker ned når der opstår fejl. Funktionen itterere listen af processer og lukker dem ned en efter en ved hjælp af systemkaldet `pthread_kill`. Når denne metode har kørt vil kun hoved tråden, den der eksekvere `main`, være aktiv og sikre at den allokerede hukommelse bliver frigivet inden programmet lukkes helt.

5.2.8 Trådsikring

For at sikre at der ikke er flere processer som får allokeret resurcer samtidigt har vi anvendt en `pthread_mutex` til at låse state'et når der allokeres og frigives resurcer. Dette er gjort for at forebygge at der opstår race-conditions i forhold til state'et.

5.3 Fejl og mangler

Umiddelbart kører programmet, men vi observerer en tendens til at de allocation-værdier vi printer til terminalen ender som de samme... igen og igen. Vi er ikke sikre på om det er et udtryk for at det overordnede state er blevet unsafe eller på anden vis ustabilt. Det forekommer os at problemet opstår grundet den måde tilfældighederne genereres på. Vi har derfor forsøgt at ændre koden i generator metoderne således at den nu bruger *double* til at beregningen og bruger `math.round()` i stedet for *cast* til integers. Dette lader til at have afhjulpnet problemet med at resurcerne ikke blev frigivet når der kun var en allokeret én resource til processen.

5.4 Test

Vores testning af programmet begrænser til de følgende 5 test.

1. Vi har startet programmet op med et *safe* state og kontrolleret at processerne begynder at låse og frigive resurcer.
2. Vi har startet programmet op med et *unsafe* state og programmet terminere uden at processerne bliver sat i gang.
3. Vi har haft modificeret koden således at en proces requester flere resurcer end denne har angivet i *max* tabellen, og kontrolleret at dette for processerne til at lukke ned.
4. Vi har haft modificeret koden, således at en proces releaser flere resurcer end denne har allokeret, og kontrolleret at dette for processerne til at lukke ned.
5. Vi har teste at hukommelsen allokeres dynamisk ved at skrive input filen *badstate.txt* således at den har 3 processer og 4 resurcer hvor de input filer der fulgte med opgaveformuleringen var implementeret med 4 processer og 3 resurcer.

Endvidere har vi haft programmet kørende i periode på ca. 5 minutter og observeret de meddelelser som programmet printer til terminalen. Ud fra denne observation er det vores umiddelbare indtryk af alle processerne låser og frigiver resurcer over tid.

Litteratur

- [1] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 9th edition, 2013.

A Appendiks

A.1 Appendix for delopgave 1

A.1.1 Header file for delopgave 1 - oo2_1.h

```
1 typedef struct _sumjob {
2     int first;
3     int last;
4 } Sumjob;
```

Listing 2: Definition af Sumjob struct.

A.1.2 Kildekode til delopgave 1 - oo2_1.c

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5 #include "oo2_1.h"
6
7 double sum; /* this data is shared by the
8             thread(s) */
9 void *runner(void *param); /* threads call this function */
10
11 int main(int argc, char *argv[])
12 {
13     pthread_attr_t attr; /* set of thread attributes */
14
15     if (argc != 3) {
16         fprintf(stderr, "usage: a.out <integer value>\n");
17         return -1;
18     }
19     if (atoi(argv[1]) < 0) {
20         fprintf(stderr, "N must be >= 0\n");
21         return -1;
22     }
23     if (atoi(argv[2]) < 1) {
24         fprintf(stderr, "Threads must be >= 0\n");
25         return -1;
26     }
27     pthread_attr_init(&attr); /* get the default attributes */
28
29     pthread_t *pthread_ids;
30     pthread_ids = (pthread_t *) malloc(atoi(argv[2]) * sizeof(pthread_t));
31     Sumjob *jobs;
32     jobs = (Sumjob *) malloc(atoi(argv[2]) * sizeof(Sumjob));
33
34     int interval = atoi(argv[1]) / atoi(argv[2]);
35     int i;
36
37     for (i = 0; i < atoi(argv[2]); i++) {
38         jobs[i].first = interval*i;
39         jobs[i].last = ((i == atoi(argv[2]) - 1) ? atoi(argv[1]) : interval*(i+1));
40         pthread_create(&pthread_ids[i], &attr, runner, &jobs[i]); /* create the
41                                                                    thread */
42     }
43
44     for (i = 0; i < atoi(argv[2]); i++) {
45         pthread_join(pthread_ids[i], NULL); /* wait for the thread to exit */
46     }
47
48     free(pthread_ids);
49     free(jobs);
```

```

48 |     printf("Threads: %i Sum = %f\n", atoi(argv[2]), sum);
49 | }
50 |
51 |
52 | /* The thread will begin control in this function */
53 | void *runner(void *param)
54 | {
55 |     Sumjob *job = param;
56 |     double localsum = 0;
57 |     int i;
58 |     for (i = job->first+1; i <= job->last; i++)
59 |         localsum += sqrt(i);
60 |
61 |     sum += localsum;
62 |
63 |     pthread_exit(0);
64 | }

```

Listing 3: Implementation af flertråde kvadratrods sum.

A.2 Appendix for delopgave 2

A.2.1 Kildekode til delopgave 2 - *list.c*

```
1  /*
2      *****
3
4      list.c
5
6      Implementation of simple linked list defined in list.h.
7
8      *****
9      */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <pthread.h>
15 #include "list.h"
16
17 pthread_mutex_t mutex;
18
19 /* list_new: return a new list structure */
20 List *list_new(void)
21 {
22     List *l;
23     if (pthread_mutex_init(&mutex, NULL) != 0) {
24         printf("Could not initialize mutex.\nOperation aborted...\n");
25         return NULL;
26     }
27
28     l = (List *) malloc(sizeof(List));
29     l->len = 0;
30
31     /* insert root element which should never be removed */
32     l->first = l->last = (Node *) malloc(sizeof(Node));
33     l->first->elm = NULL;
34     l->first->next = NULL;
35     return l;
36 }
37
38 /* list_add: add node n to list l as the last element */
39 void list_add(List *l, Node *n)
40 {
41     int nlen = 1;
42     pthread_mutex_lock(&mutex);
43     //sleep(1); // Used for testing;
44     Node *newlast = n;
45     Node *oldlast = l->last;
46     oldlast->next = newlast;
47     while(n->next != NULL) {
48         newlast = newlast->next;
49         nlen++;
50     }
51     l->last = newlast;
52     l->len += nlen;
53     pthread_mutex_unlock(&mutex);
54 }
55
56 /* list_remove: remove and return the first (non-root) element from list l */
57 Node *list_remove(List *l)
58 {
59     pthread_mutex_lock(&mutex);
60     Node *root = l->first;
61     if (root->next == NULL) {
62         pthread_mutex_unlock(&mutex);
63     }
64 }
```

```

60     return NULL;
61 }
62 Node *rm = root->next;
63 root->next = rm->next;
64 l->len -=1;
65 rm->next = NULL;
66 if (l->last == rm)
67     l->last = l->first;
68 pthread_mutex_unlock(&mutex);
69 return rm;
70 }
71
72 /* node_new: return a new node structure */
73 Node *node_new(void)
74 {
75     Node *n;
76     n = (Node *) malloc(sizeof(Node));
77     n->elm = NULL;
78     n->next = NULL;
79     return n;
80 }
81
82 /* node_new_str: return a new node structure, where elm points to new copy of
83     s */
84 Node *node_new_str(char *s)
85 {
86     Node *n;
87     n = (Node *) malloc(sizeof(Node));
88     n->elm = (void *) malloc((strlen(s)+1) * sizeof(char));
89     strcpy((char *) n->elm, s);
90     n->next = NULL;
91     return n;
92 }

```

Listing 4: Kode for implementation af hægtede lister

A.2.2 Kildekode til test af delopgave 2 - *list/main.c*

```
1  /*****
2      main.c
3      Implementation of a simple FIFO buffer as a linked list defined in list.h.
4      *****/
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <pthread.h>
9  #include "list.h"
10
11
12  // FIFO list;
13  List *fifo;
14  void *add(void *param);
15  void *testremove(void *param);
16  int test();
17
18  int main(int argc, char* argv[])
19  {
20      fifo = list_new();
21      test();
22      return 0;
23  }
24
25  void test() {
26      pthread_attr_t attr;
27      pthread_attr_init(&attr);
28      int i, j;
29      fifo = list_new();
30
31      pthread_t ids[2];
32      pthread_create(&ids[0], &attr, add, "thread 0");
33      pthread_create(&ids[1], &attr, add, "thread 1");
34
35      for (j = 0; j < 2; j++)
36          pthread_join(ids[j], NULL); /* wait for the thread to exit */
37
38      for (i = 0; i < 20; i++) {
39          Node *n = list_remove(fifo);
40          printf("Removed: %s\n", (char *)n->elm);
41      }
42  }
43
44  void *add(void *param)
45  {
46      int i;
47      for (i = 0; i < 10; i++) {
48          list_add(fifo, node_new_str((char *)param));
49          printf("added: %s\n", (char *)param);
50      }
51      pthread_exit(0);
52  }
```

Listing 5: Implementation af test funktion til delopgave 2.

A.3 Appendix for delopgave 3

A.3.1 Kildekode til delopgave 3 - oo2_3.c

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <semaphore.h>
5 #include <sys/time.h>
6 #include "list/list.h"
7
8 void *producer(void *param);
9 void *consumer(void *param);
10 void Sleep(float wait_time_ms);
11 int do_produce();
12 int do_consume();
13 void cleanUp();
14
15 int produced, consumed, o, b, product_count;
16
17 sem_t _mutex, full, empty, sem_produced, sem_consumed, sem_count;
18 List *buf;
19
20 /*
21  * Call the program with the following parameters:
22  * Producers (p)
23  * Consumers (c)
24  * Buffer size (b)
25  * Number of products to produce (o)
26  */
27 int main(int argc, char *argv[]) {
28     int p, c;
29     produced = consumed = 0;
30
31     /* Parse and check input and store in variables */
32     if ((p = atoi(argv[1])) < 1) {
33         printf("producers must be positive number\n");
34         exit(0);
35     }
36     if ((c = atoi(argv[2])) < 1) {
37         printf("consumers must be positive number\n");
38         exit(0);
39     }
40     if ((b = atoi(argv[3])) < 1) {
41         printf("buffer size must be positive number\n");
42         exit(0);
43     }
44     if ((o = atoi(argv[4])) < 1) {
45         printf("number of products to produce must be positive\n");
46         exit(0);
47     }
48
49     /* seed the random number generator */
50     struct timeval tv;
51     gettimeofday(&tv, NULL);
52     srand(tv.tv_usec);
53
54     /* Initiate semaphores */
55     if (sem_init(&sem_produced, 0, 1)
56         || sem_init(&sem_consumed, 0, 1)
57         || sem_init(&sem_count, 0, 1)
58         || sem_init(&_mutex, 0, 1)
59         || sem_init(&full, 0, 0)
60         || sem_init(&empty, 0, b)) {
61         printf("Unable to initialize semaphors.\n");
62     }
```

```

63
64 printf("You have requested:\n");
65 printf("%4i producers producing a total of %6i items.\n", p, o);
66 printf("%4i consumers and a buffer containing up to %6i elements\n", c, b);
67
68 buf = list_new();
69 int i;
70 int *ids;
71 ids = (int *) malloc((c > p ? c : p) * sizeof(int));
72
73 for (i = 0; i < (c > p ? c : p); i++)
74     ids[i] = i;
75
76 pthread_attr_t attr;
77 pthread_attr_init(&attr);
78
79 pthread_t *cids;
80 cids = (pthread_t *) malloc(c * sizeof(pthread_t));
81
82 for (i = 0; i < c; i++)
83     pthread_create(&cids[i], &attr, consumer, &ids[i]);
84
85 pthread_t *pids;
86 pids = (pthread_t *) malloc(p * sizeof(pthread_t));
87
88 for (i = 0; i < p; i++)
89     pthread_create(&pids[i], &attr, producer, &ids[i]);
90
91 for (i = 0; i < p; i++)
92     pthread_join(pids[i], NULL);
93
94 for (i = 0; i < c; i++)
95     pthread_join(cids[i], NULL);
96
97 printf("\nResult:\n");
98 printf("%6i items has been produced.\n", produced);
99 printf("%6i items has been consumed.\n", consumed);
100 printf("%6i elements are left in buffer.\n", buf->len);
101
102 cleanUp();
103 exit(0);
104 }
105
106 /*Producer function*/
107 void *producer(void *param) {
108
109     int id = *(int *)param;
110     Node *node;
111     char elm[10];
112
113     while(do_produce()) {
114         Sleep(1000);
115
116         sem_wait(&sem_count);
117         sprintf(elm, "Item-%i", product_count);
118         product_count++;
119         sem_post(&sem_count);
120
121         node = node_new_str(elm);
122
123         sem_wait(&empty);
124         sem_wait(&mutex);
125
126         list_add(buf, node);
127         printf("Producer %4i produced %9s. Items in buffer: %4i (out of %4i)\n",
            id, elm, buf->len, b);

```

```

128
129     sem_post(&_mutex);
130     sem_post(&full);
131 }
132
133 pthread_exit(0);
134 }
135
136 /* Consumer function */
137 void *consumer(void *param) {
138     int id = *(int *)param;
139     Node *n;
140
141     while(do_consume()) {
142         Sleep(1000);
143
144         sem_wait(&full);
145         sem_wait(&_mutex);
146
147         n = list_remove(buf);
148
149         sem_post(&_mutex);
150         sem_post(&empty);
151
152         printf("Consumer %4i consumed %9s. Items in buffer: %4i (out of %4i)\n",
153             id, (char *)n->elm, buf->len, b);
154
155         free(n);
156     }
157
158     pthread_exit(0);
159 }
160
161 /* Checks if more products needs to be produced */
162 int do_produce(void) {
163     sem_wait(&sem_produced);
164     int r = (produced < o ? ++produced : 0);
165     sem_post(&sem_produced);
166     return r;
167 }
168
169 /* Checks if more products are to be consumed */
170 int do_consume(void) {
171     sem_wait(&sem_consumed);
172     int r = (consumed < o ? ++consumed : 0);
173     sem_post(&sem_consumed);
174     return r;
175 }
176
177 /* Random sleep function */
178 void Sleep(float wait_time_ms) {
179     wait_time_ms = ((float)rand())*wait_time_ms / (float)RAND_MAX;
180     usleep((int) (wait_time_ms * 1e3f)); // convert from ms to us
181 }
182
183 void cleanUp() {
184     sem_destroy(&_mutex);
185     sem_destroy(&full);
186     sem_destroy(&empty);
187     sem_destroy(&sem_produced);
188     sem_destroy(&sem_consumed);
189     sem_destroy(&sem_count);
190     free(buf);
191 }

```

Listing 6: Implementation af Producer/consumer system til delopgave 3.

A.4 Appendix for delopgave 4

A.4.1 Kildekode til delopgave 4 - *banker/banker.c*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <pthread.h>
5  #include <signal.h>
6  #include <math.h>
7
8  typedef struct state {
9      int *resource;
10     int *available;
11     int **max;
12     int **allocation;
13     int **need;
14 } State;
15
16 // Global variables
17 int m, n;
18 State *s = NULL;
19 pthread_t *tid;
20
21 // Mutex for access to state.
22 pthread_mutex_t state_mutex;
23
24 int safe_state(State *s);
25 void printArr(int *arr, int lenght);
26 void subArr(int * dest, int *sub, int lenght);
27 void addArr(int * dest, int *add, int lenght);
28 void killProcesses();
29
30 /* Random sleep function */
31 void Sleep(float wait_time_ms)
32 {
33     // add randomness
34     wait_time_ms = ((float)rand())*wait_time_ms / (float)RAND_MAX;
35     usleep(((int) (wait_time_ms * 1e3f))); // convert from ms to us
36 }
37
38 /* Allocate resources in request for process i, only if it
39    results in a safe state and return 1, else return 0 */
40 int resource_request(int i, int *request)
41 {
42     pthread_mutex_lock(&state_mutex);
43
44     if(islarger(request, s->need[i], n)) {
45         printf(" !!!ERROR!!!\n");
46         printf("Process %i's request is greather than its need.\n", i);
47         killProcesses();
48     }
49
50     subArr(s->available, request, n);
51     addArr(s->allocation[i], request, n);
52     subArr(s->need[i], request, n);
53
54     if (safe_state(s)) {
55         pthread_mutex_unlock(&state_mutex);
56         return 1;
57     } else {
58         addArr(s->available, request, n);
59         subArr(s->allocation[i], request, n);
60         addArr(s->need[i], request, n);
61         pthread_mutex_unlock(&state_mutex);
62         return 0;
63     }
```

```

63     }
64 }
65
66 /* Release the resources in request for process i */
67 void resource_release(int i, int *request)
68 {
69     pthread_mutex_lock(&state_mutex);
70
71     if(islarger(request, s->allocation[i], n)) {
72         printf(" !!!ERROR!!!\n");
73         printf("Process %i tries to release more resources than it has allocated.\n", i);
74         killProcesses();
75     }
76
77     addArr(s->available, request, n);
78     subArr(s->allocation[i], request, n);
79     addArr(s->need[i], request, n);
80
81     pthread_mutex_unlock(&state_mutex);
82 }
83
84 /* Generate a request vector */
85 void generate_request(int i, int *request)
86 {
87     int j, sum = 0;
88     while (!sum) {
89         for (j = 0; j < n; j++) {
90             request[j] = round((double)s->need[i][j] * ((double)rand()) / (double)
                                RANDMAX);
91             //request[j] = s->need[i][j] * ((double)rand()) / (double)RANDMAX;
92             sum += request[j];
93         }
94     }
95     printf("Process %d: Requesting resources.\n", i);
96 }
97
98 /* Generate a release vector */
99 void generate_release(int i, int *request)
100 {
101     int j, sum = 0;
102     while (!sum) {
103         for (j = 0; j < n; j++) {
104             request[j] = round((double)s->allocation[i][j] * ((double)rand()) / (
                                double)RANDMAX);
105             //request[j] = s->allocation[i][j] * ((double)rand()) / (double)RANDMAX
106             ;
107             sum += request[j];
108         }
109     }
110     printf("Process %d: Releasing resources.\n", i);
111 }
112
113 /* Threads starts here */
114 void *process_thread(void *param)
115 {
116     /* Process number */
117     int i = (int) (long) param, j;
118     /* Allocate request vector */
119     int *request = malloc(n*sizeof(int));
120     while (1) {
121         /* Generate request */
122         generate_request(i, request);
123         while (!resource_request(i, request)) {
124             /* Wait */
125             Sleep(100);

```

```

125     }
126     /* Generate release */
127     generate_release(i, request);
128     /* Release resources */
129     resource_release(i, request);
130     /* Wait */
131     Sleep(1000);
132 }
133 free(request);
134 }
135
136 int main(int argc, char* argv[])
137 {
138     /* Get size of current state as input */
139     int i, j;
140     printf("Number of processes: ");
141     scanf("%d", &m);
142     printf("Number of resources: ");
143     scanf("%d", &n);
144
145     /* Allocate memory for state */
146     s = (State *) malloc(sizeof(State));
147     s->resource = (int *) malloc(n * sizeof(int));
148     s->available = (int *) malloc(n * sizeof(int));
149     s->max = (int **) malloc(m * sizeof(int *));
150     s->allocation = (int **) malloc(m * sizeof(int *));
151     s->need = (int **) malloc(m * sizeof(int *));
152
153     for (i = 0; i < m; i++) {
154         s->max[i] = (int *) malloc(n * sizeof(int));
155         s->allocation[i] = (int *) malloc(n * sizeof(int));
156         s->need[i] = (int *) malloc(n * sizeof(int));
157     }
158
159     if (s == NULL) { printf("\nYou need to allocate memory for the state!\n");
        exit(0); };
160
161     /* Get current state as input */
162     printf("Resource vector: ");
163     for (i = 0; i < n; i++)
164         scanf("%d", &s->resource[i]);
165     printf("Enter max matrix: ");
166     for (i = 0; i < m; i++)
167         for (j = 0; j < n; j++)
168             scanf("%d", &s->max[i][j]);
169     printf("Enter allocation matrix: ");
170     for (i = 0; i < m; i++)
171         for (j = 0; j < n; j++)
172             scanf("%d", &s->allocation[i][j]);
173     printf("\n");
174
175     /* Calculate the need matrix */
176     for (i = 0; i < m; i++)
177         for (j = 0; j < n; j++)
178             s->need[i][j] = s->max[i][j] - s->allocation[i][j];
179
180     /* Calculate the availability vector */
181     for (j = 0; j < n; j++) {
182         int sum = 0;
183         for (i = 0; i < m; i++)
184             sum += s->allocation[i][j];
185         s->available[j] = s->resource[j] - sum;
186     }
187
188     /* Output need matrix and availability vector */
189     printf("Need matrix:\n");

```

```

190     for(i = 0; i < n; i++)
191         printf("R%d ", i+1);
192     printf("\n");
193     for(i = 0; i < m; i++) {
194         for(j = 0; j < n; j++)
195             printf("%d ", s->need[i][j]);
196         printf("\n");
197     }
198     printf("Availability vector:\n");
199     for(i = 0; i < n; i++)
200         printf("R%d ", i+1);
201     printf("\n");
202     for(j = 0; j < n; j++)
203         printf("%d ", s->available[j]);
204     printf("\n");
205
206     /* If initial state is unsafe then terminate with error */
207     if(!safe_state(s)) {
208         printf("Start state is unsafe\n");
209     } else {
210         pthread_mutex_init(&state_mutex, NULL);
211
212         /* Seed the random number generator */
213         struct timeval tv;
214         gettimeofday(&tv, NULL);
215         srand(tv.tv_usec);
216
217         /* Create m threads */
218         tid = malloc(m*sizeof(pthread_t));
219         for (i = 0; i < m; i++)
220             pthread_create(&tid[i], NULL, process_thread, (void *) (long) i);
221
222         /* Wait for threads to finish */
223         for (i = 0; i < m; i++)
224             pthread_join(tid[i], NULL);
225         printf("Something went wrong processes has terminated.\n");
226     }
227     /* Free state memory */
228     free(s->resource);
229     free(s->available);
230     for (i = 0; i < m; i++) {
231         free(s->max[i]);
232         free(s->allocation[i]);
233         free(s->need[i]);
234     }
235     free(s);
236     sem_destroy(&state_mutex);
237     exit(1);
238 }
239
240 /* Checks if a given state is safe.*/
241 int safe_state(State *s) {
242     int count = m, i, isSafe;
243     int *finish = (int *)malloc(n * sizeof(int));
244     int *work = (int *)malloc(m * sizeof(int));
245     for (i = 0; i < m; i++) {
246         finish[i] = 0;
247         work[i] = s->available[i];
248     }
249
250     while (count != 0) {
251         isSafe = 0;
252         for (i = 0; i < m; i++) {
253             if (!finish[i]) {
254                 if (islarger(work, s->need[i], n)) {
255                     finish[i] = 1;

```

```

256         count--;
257         isSafe = 1;
258         addArr(work, s->allocation[i],n);
259         break;
260     }
261 }
262 }
263 if (!isSafe) {
264     printf("The requested state is unsafe.\n");
265     return 0;
266 }
267 }
268 printf("The requested state is safe.\n");
269 return 1;
270 }
271
272 /*Checks if the array check is larger than match.*/
273 int islarger(int *check, int *match, int size) {
274     int i;
275     for (i = 0; i < size; i++)
276         if (check[i] > match[i]) return 1;
277     return 0;
278 }
279
280 /*Prints array to terminal*/
281 void printArr(int *arr, int lenght) {
282     int i;
283     printf("{");
284     for (i = 0; i < lenght; i++)
285         printf("[%i]", arr[i]);
286     printf("}\n");
287 }
288
289 /*Adds up to arrays and store result in dest.*/
290 void addArr(int *dest, int* add, int lenght) {
291     int i;
292     for (i = 0; i < lenght; i++)
293         dest[i] += add[i];
294 }
295
296 /*Subtracts to arrays and store result in dest.*/
297 void subArr(int *dest, int *sub, int lenght) {
298     int i;
299     for (i = 0; i < lenght; i++)
300         dest[i] -= sub[i];
301 }
302
303 /*Terminates all processes.*/
304 /*Called if a process violates the rules.*/
305 void killProcesses() {
306     int i;
307     for (i = 0; i < m; i++)
308         pthread_kill(tid[i], SIGTERM);
309 }

```

Listing 7: Implementation af Bankers algoritmen til delopgave 4.

A.4.2 Usikkert opstartesstadiet - *banker/badstate.txt*

```
1 3
2 4
3
4 3 0 1 2
5
6 1 2 2 1
7 1 1 3 3
8 1 2 1 0
9
10 3 3 2 2
11 1 2 3 4
12 1 3 5 0
```

Listing 8: Tekst fil representerende et usikkert stadiet.

A.5 Makefiles

A.5.1 Makefile for delopgave 1 og 3

```
1 all: sum procon
2
3 PCOBS = oo2_3.o list/list.o
4 SUMOBS = oo2_1.o
5 LIBS = -lm -pthread
6 CC = gcc
7
8 sum: ${SUMOBS}
9     ${CC} -o $@ ${SUMOBS} ${LIBS}
10
11 procon: ${PCOBS}
12     ${CC} -o $@ ${LIBS} ${PCOBS}
13
14
15 clean:
16     rm -rf *.o sum procon
```

Listing 9: Makefile til kompilering af delopgave 1 og 3.

A.5.2 Makefile for delopgave 2

labelsec:make22

```
1 all: fifo
2
3 OJS = list.o main.o
4 LIBS = -pthread
5
6 OJSPC = list.o oo2_3.o
7
8
9 fifo: main.o ${OJS}
10     gcc -o $@ ${LIBS} ${OJS}
11
12 clean:
13     rm -rf *.o fifo
```

Listing 10: Makefile til kompilering af delopgave 2.

A.5.3 Makefile for delopgave 4

```
1 all: banker
2
3 OJS = banker.o
4 LIBS = -lm -pthread
5 CC = gcc
6
7 banker: ${OJS}
8     ${CC} -o $@ ${OJS} ${LIBS}
9
10 clean:
11     rm -rf *.o banker
```

Listing 11: Makefile til kompilering af delopgave 4.