Obligatorisk opgave 1

Operativsystemer og C

Bachelor in Software Development,
IT-University of Copenhagen

Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

October 3rd, 2012

Contents

1 Forord

I denne rapport dokumenterer vi vores valg i forhold til implementationen af opgaverne i Obligatorisk opgave 2.

Kildekode og testdokumentation kan findes i appendix på side 8. Vores git repository kan findes på https://github.com/esfdk/BOSC/tree/master/oo2.

Vi ville gerne have lavet mere fyldestgørende test af opgave 4.

2 Beskrivelse af implementation

2.1 Opgave 1

Vi skal optimere en sekvensiel funktion, der udregner summen af kvadratrødder, således at den kører hurtigere på en multicore maskine end den ellers ville have gjort. Dette skal gøres ved hjælp af tråde.

Koden til funktionen kan findes i 'mul_sum/mulsum.c'. Den er løst baseret på koden fra Operating System Concepts, 8th edition, side 161.

2.1.1 Del 1

- Udregning af hvilke tal hver tråd skal arbejde med.
- Lav tråde med oprettede structs.
- Udregning af sum af kvadratrødder.
- Resultat af alle tråde.

Arbejde Før vi opretter vores tråde, udregner vi hvor mange tal hver tråd skal arbejde med. Derefter laver vi en beregning til at finde de laveste og højeste tal. Det gør vi blandet andet ved "double minnum = floor(work * n) + 1;". Vi bruger floor, da vi kun er intersseret i heltal. Når det er gjort, opretter vi struct og placerer dem i arrays såsom "calc_result[n].minimum_number = minnum;" (se linje 44-52).

Oprettelse af tråde Når vi har fundet ud af hvilket arbejdet hver tråd skal udføre, opretter vi dem. Hver tråd bliver sat til at køre TaskCode med et struct fra punkt 1 som parameter (se linje 57-60).

Udregning Derefter laver vi selve udregningen, hvor vi finder summen af kvadratrødderne. Når en tråd er færdig med sin del, bliver der kaldt pthread_exit på den, hvilket får den til at terminere (se linje 17-29).

Beregning af resultat For at finde det samlede resultat går vi igennem arrayet af structs og tilgår deres sumsqrt værdi. Alle disse værdier ligger vi sammen og printer dem ud (se linje 71-75).

2.1.2 Del 2

I opgave 1.2 bliver vi bedt om at tjekke vores multitrådet sum-funktion vha. en speedup graf. En speedup graf viser hvor meget hurtigere en funktion bliver, når man lader arbejdet foregå på flere tråde.

$$S_p = \frac{T_1}{T_p}$$

Figure 2.1: Formel til udregning af speedup

Speedup udregnes ved at tage kørselstiden ved 1 tråd og dividere det med kørselstiden med n tråde, som i vores tilfælde er 1, 2, 4, 6 og 8. Det optimale resultat ville være en linær speedup, hvor speedupen er lig antallet af tråde, som funktionen er blevet kørt med.

$$S_p = p.$$

Figure 2.2: Ideal speedup

Valg af maskine til speedupgraf Til test af vores multitrådet sum-funktion havde vi valget mellem to eller fire processore eller fire processor med hyperthreading. Vi valgte at bruge computeren med fire processor. Det gjorde vi fordi det var medianen og vi mente, det var hvad en standard computer ville have. Dette ville give os nogle mere generelle resultater.

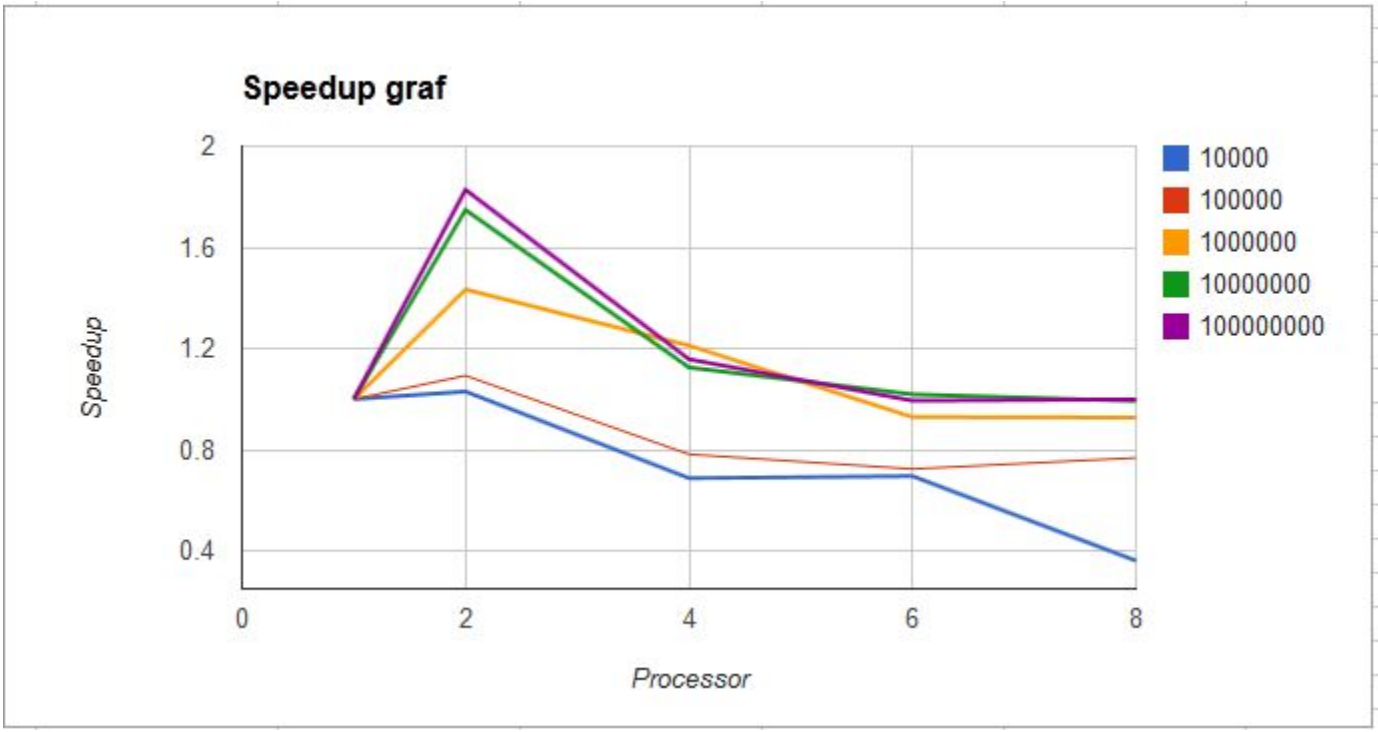

Figure 2.3: Speedup graf for maskine med 4 processors

Uventet resultat ved kørsel med fire tråde Som man kan se udfra vores speedup graf, er der næsten en ideal speedup fra en tråd til to tråde. Ved fire tråde bliver den dog kun en smule hurtigere end ved en tråd. Dette resultat synes vi er fovirrende, da det burde være muligt at få en speedup svarende til speeduppen fra en tråd til to tråde. Computeren har trods alt fire realle processor. Vi har ingen forklaring på hvorfor det er sådan. Vi havde forventet at få problemer ved seks og otte tråde da vi ikke har mulighed for at tildele hver tråd en processer.

Vi har kort testet vores funktion på en laptop med fire processor og hyperthreading, hvor vi havde en normal speedup optil fire tråde. Dette er forståligt da fire processor med hyperthreading kun 'simulerer' otte processor, og derfor ikke arbejder ligeså optimalt som otte konkrete processor.

2.1.3 Tests

For at teste om vores forbedret multitrådet sum-funktion kørte hurtigere med flere tråde, brugte vi en speedupgraf som nævnt i 2.1.2. For at lave en ordenlig speedupgraf lavede vi 10 runs af funktionen ved

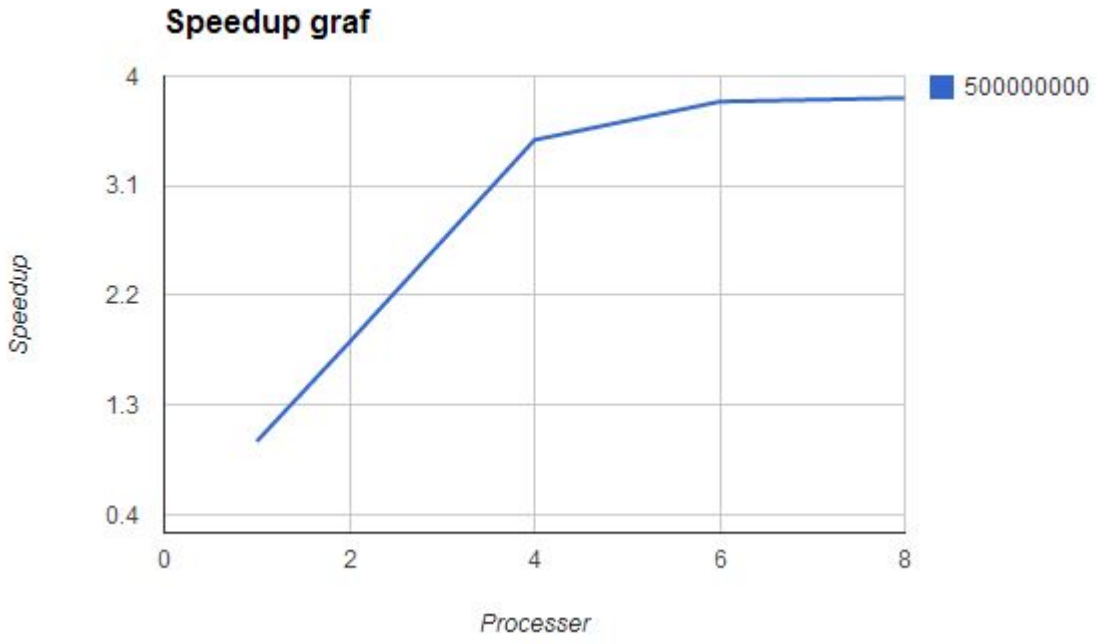

Figure 2.4: Speedup graf for maskine med 4 processors og hyper-threading

1-2-4-6-8 tråde ved input 10.000. Input blev gjort 10 gange højere per iteration op til 100.000.000 samt fra 50.000 og så gjort 10 gange højere op per iteration til 500.000.000. Dette gjorde, at vi havde en stor mængde data til at lave en graf over gennemsnittet af vores timings.

Vores resultater kan findes både i appendix A.1.1 på side 9 og GoogleDocs på bitly.com/U0EYAK

2.2 Opgave 2

Vores løsning af opgave 2 er beskrevet i 'FIFO/list.c' og 'FIFO/list.h'. Testkoden til listen er beskrevet i 'FIFO/testNoThreads.c' og 'FIFO/tetstThreads.c'. 'list.c', 'list.h' og 'testNoThreads.c' er baseret på kode fra 'opg2' zip-filen fra bloggen[1].

2.2.1 Del 1

Vi har valgt at implementere 'list_add(List *l, Node *n)' på en meget simpel måde (se linje 35 til 41 i list.c). Listens sidste element ('l->last') bliver sat til at pege på den nye node ('l->last->next = n'), hvorefter 'l->last' bliver sat til at være n. På denne måde vil det anden sidste element pege på det nye element, og 'last' peger på det nye element, da det er det sidste i listen.
'*list_remove(List *l)' implementationen kan ses på linje 44 til 60 i list.c filen. I funktionen bliver der lavet en 'Node *n'. Denne node bliver sat til at være det første element i listen, altså 'l->first->next' da 'first' er root elementet og aldrig skal pilles ved. Rodelementets 'next' bliver sat til at pege på n's 'next', hvilket er det andet element i listen. På denne måde er det første element blevet "fjernet" fra listen. Funktionen

[1] https://blog.itu.dk/BOSC-E2012/files/2012/10/opg2.zip

tjekker derefter om 'first->next' er null, for hvis den er, skal 'last' pege på først, ligesom da listen lige var blevet lavet. Til slut retuneres n.

2.2.2 Del 2

Det mest åbenlyse problem er, at flere tråde kan editere listen på samme tid. For eksempel: To tråde tilgør listen på samme tid og finder frem til det sidste element ('last'). Begge tråde prøver at tilføje et element til listen, hvilket betyder at de begge prøver at tilføje et element til 'last' på samme tid. Tråd #1 tilføjer sit element til 'last', hvorefter tråd #2 tilføjer sit element til 'last' (det element som #1 lige har tilføjet sit element til), hvilket ødelægger linket mellem 'last' og tråd #1s element.

En anden udgave af problemet ovenover er, hvis to forskellige tråde vil remove på samme tid. Tråd #1 går ind og læser 'first's 'next' element (da det skal blive first efter removal). Samtidig går tråd #2 ind og kører hele removal, hvorefter #1 prøver at fjerne det element der lige er blevet fjernet. Derved er kun et element blevet fjernet, hvor to elementer burde have været fjernet. Den omvendte situation kan også opstå, hvor to elementer bliver fjernet, men hver tråd tror kun at et element er blevet fjernet.

Alt efter størrelsen af listen kan der også opstå problemet. Hvis listen kun er et element langt og en tråd prøver at adde mens en anden prøver at remove, kan der opstå forkerte resultater. Sker de samtidig, kan elementet blive addet (hvilket gør at listen er to elementer lang), på samme tid som det første (og eneste) element bliver removet. Risikoen er, at det nye element bliver appendet på det element der lige er blevet removet, hvorved ingen af elementerne er i listen.

2.2.3 Del 3

Vi har brugt mutex låse i list_add og list_remove funktionerne. I list_add er de to linjer logik inde i en mutex lock, da det ikke skal være muligt for flere tråde, at tilføje elementer på samme tid. I list_remove er alt undtagen return-statementet indkapslet af mutex locks af samme årsag som i list_add; det skal ikke være mulgit at fjerne flere elementer samtidig. At 'return n' ikke er i en lock, gør dog ingen skade. Når funktionen når til 'return', bruger den ikke listen mere, og det kan derfor køres uden at være i en lock. Vi har beskrevet vores test at den flertrådet liste i afsnit 2.2.4.

2.2.4 Tests

Vi har lavet to tests til first-in-first-out listen. En test uden brug af tråde for at se om lsiten overhovedet virker, og en test med tråde, for at sikre at flere tråde kan bruge listen samtidig, uden der opstår problemer. Testen uden brug at tråde findes i testNoThreads.c. I denne test laver vi en liste, tilføjer to elementer til listen, fjerner to elementer fra listen og printer deres værdier for at sikre os, at de er kommet ud i den rigtige rækkefølge.
Testen med tråde findes i testThreads.c og er lidt mere omfattende. Testen tager to parametre: Antallet af tråde der skal laves, og antallet af elementer hver tråd skal håndtere. 'main' funktionen laver et array med det valgte antal tråde, og sætter dem alle samme til at køre '*TaskCode(void *argument)' funktionen. Hver tråd får sit eget nummer i arrayet med, for at man nemmere kan holde styr på hvilken tråd der gør hvad. Hver tråd laver det valgte antal elementer (som er strings med formatet ("Thread #%d, element %d", threadNumber, elementNumber)) og adder dem til listen. Derefter fjerner tråden det antal elementer, som den har addet til listen og printer værdien af disse elementer.
Det man kan se med testThreads testen er, at trådene går ind og låser listen når de bruger den. Som oftest vil elementerne være i rækkefølge, så det er alle tråd #1's elementer først, så tråd #2's, osv.

2.3 Opgave 3

Vores løsning af opgave 3 er beskrevet i 'prod_cons/prodCons.c' samt 'prod_cons/prodCons.h'. 'prod_cons/List' mappen indeholer vores list fra opgave 2, som vi også inkluderer i 'FIFO'-mappen.

I denne opgave har vi brugt 'pthread_mutex_t' objekter til at undgå problemer hvor flere tråde ændrer samme element på samme tid. 'pthread_mutex_t' er objekter, der kan låses/låses op via 'pthread_mutex_lock'-funktions kaldet. Hvis en funktion låser en mutex, forhindre den derved andre funktioner i at køre videre, forudsat at de selv skal bruge mutex'en.
De andre funktioner går i stå indtil mutex'en låses op igen, hvilket sørger for, at critical code kun køres af en tråd ad gangen. Bruger man ikke mutex låse i et flertrådet program, risikerer man at løbe ind i situationer hvor flere tråde har modificeret samme element på samme tid, hvilket kan ødelægge programmet.

Vi har også brugt 'sem_t' objekter, også kaldet semaphore. Det er objekter, der indeholder en værdi, og som kan bruges af funktioner til synkronisation mellem tråde. De har to interessante funktioner: 'sem_wait(sem_t)' og 'sem_post(sem_t)'. 'sem_wait()' kigger på værdien af semaphoren.
Hvis værdien er mindre end nul, vil den kaldende funktion pause, og vente på værdien bliver højere end nul. Når det sker, vil den kaldende funktion få lov til at køre videre og 'sem_wait()' reducerer værdien af semaphoren med en. 'sem_post()' forøger simpelthen bare værdien af semaphoren med en.

Både 'phtread_mutex_t' og 'sem_t' objekter virker som låse, omend på forskellige måder. 'pthread_mutex_t' virker som en ja/nej, og vil kun lade en tråd udføre sit arbejde ad gangen. 'sem_t' lader gerne flere tråde arbejde på samme tid, så længe semaphorens værdi er over nul.

2.3.1 Opfyldelse af punkter

Punkt 1 - Vores implementation gør det muligt at definere antallet af producers, antallet af consumers, størrelsen på bufferen og antallet af produkter der skal produceres. Dette gør vi, ved først at tjekke om der er den rigtige mænge inputs, hvorefter vi konveterer inputs til ints og bruger dem til at lave de forskellige ting (se linje 36-48).

Punkt 2 - Punkt 2 opfyldes af koden. Vi har kun en producer funktion og en consumer funktion (henholdsvis linje 120 og 160), og consumer/producer tråde laves og sættes til at køre det respektive kode (se linje 84-104).

Punkt 3 - Både producer funktionen og consumer funktionen kalder 'sleepRandom' funktionen hver gang de udfører deres logik (se linje 155 og 188), som sætter tråden til at sove i et tilfældigt antal sekunder (se linje 214-221).

Punkt 4 - Vi sørger for at tråde ikke udsultes ved at lade tråde sove i et tilfældigt antal sekunder, når de har arbejdet. På denne måde forhindrer vi tråde i at tage alle opgaverne og derved udsult andre tråde.

Punkt 5 - Se afsnit A.2.1 i Appendix for output.

Punkt 6 - Vi opfylder dette krav igennem et tjek i starten af både consumer og producer funktionerne. I starten af producer funktionen, tjekkes der om der stadig skal produceres flere produkter. Hvis der ikke skal, afsluttes producer tråden. Consumeren tjekker, om der er konsumeret lige så mange produkter som skulle produceres. Hvis der er, afsluttes consumer tråden.

2.4 Opgave 4

Vores løsning af opgave 4 er beskrevet i 'bankeralgorithm/banker.c' samt i appendix. Den er baseret på den ufuldstændige banker.c fra zip-filen[2] på kursusbloggen.

I forhold til beskrivelsen af Banker's algoritmen i Operating System Concepts, 8th edition, så er m og n byttet om, således at i vores kode er 'm' antallet af processer og 'n' er antallet af resurser.

2.4.1 Opfyldelse af punkter

Punkt 1 - Vi allokerer memory til state dynamisk ved bruge af malloc. Dette gør vi efter at antallet af processer og resurser er blevet læst ind. Vi var nødt til at inkludere et for-loop, da 'max', 'allocation' og 'need' er "arrays of pointers to arrays". (Se linje 226-238).
I slutningen af 'main()' funktionen frigiver det memory, som vi har allocated til state. Vi gør dette i omvendt rækkefølge, da vi er nødt til at tilgå resurse arrays'ne før vi kan frigive processer arrays'ne. (Se linje 311-322).

Punkt 2 - Vi har implementeret safety algoritmen som beskrevet på side 299 i Operating System Concepts, 8th Edition. Vi initialiserer 'work' og 'finish' som beskrevet. Derefter har vi et while-loop, som kører step 2 og step 3 af algoritmen indtil der er en iteration, hvor der ikke er en process, hvis 'finish' status skifter til true(1). Dette gør vi, fordi det er ligegyldigt, i hvilken rækkefølge processerne bliver færdig i forhold til safe eller unsafe state. (Se linje 95-159).
I hvert kald til 'resource_request(int i, int *request)' låser vi 'state_mutex' således at state ikke tilgås af mere end en process. Herefter udfører vi første step af 'resource-request' algoritmen[3]. Hvis dette step fejler, så crasher vi programmet. Ellers går vi videre til step 2, hvor vi unlocker låsen og return 0, hvis requesten ikke kan opfyldes.
Hvis requesten måske kan opfyldes, laver vi ændringerne til state og kalder vores 'safety_check()' funktion. Hvis 'safety_check()' er false, så laver vi rollback på state, releaser lock og returner 0. Ellers unlocker vi låsen, beholder vores ændringer til state og returner 1. (Se linje 32-78).

Punkt 3 - I 'resource_release(int i, int *request)' låser vi 'state_mutex', så andre tråde ikke kan tilgå state og gør det modsatte af step 3 af 'resource-request' algoritmen. Herefter unlocker vi låsen og funktionen afslutter. (Se linje 81-92).

Punkt 4 - Efter state er blevet oprettet i 'main(int argc, char* argv[])', kalder vi 'safety_check()'. Hvis det returner false, crasher vi, ellers forsætter. (Se linje 286-294).

Punkt 5 - Da vi locker 'state_mutex' i 'resource_request(int i, int *request)', 'resource_release(int i, int *request)', 'generate_request(int i, int *request)' og 'generate_release(int i, int *request)' er der ikke flere tråde, som kan tilgå state på samme tid. 'safety_check()' tilgår state, men den bliver kun kaldt inde i en block kode, hvor 'state_mutex' er låst og i 'main(int argc, char* argv[])' før trådene bliver oprettet.

2.4.2 Tests

Programmet kører og rapporterer, at begge de to eksempel input filer fra zip-filen fra kursusbloggen[4] er safe. Hvis den første linje i 'allocation matrix' (matrix nr. 2 i input filen) ændres fra '1 0 1' til '2 0 1', bliver den unsafe og dette rapporteres også korrekt.
Ellers har vi testet og set, at det ikke lader til, at programmet går i 'deadlock' eller lignende.

[2] https://blog.itu.dk/BOSC-E2012/files/2012/10/banker.zip
[3] Beskrevet på side 299 i Operating System Concepts, 8th edition.
[4] https://blog.itu.dk/BOSC-E2012/files/2012/10/banker.zip

A Test

A.1 Opgave 1

A.1.1 Opgave 1.2 speedup resultater

Figure A.1: Speedup output for input 10.000

Cores	1	2	4	6	8
1st run	0.003	0.004	0.005	0.006	0.009
2nd run	0.003	0.003	0.005	0.008	0.01
3rd run	0.004	0.003	0.005	0.008	0.008
4th run	0.005	0.003	0.004	0.007	0.01
5th run	0.003	0.003	0.004	0.006	0.11
6th run	0.004	0.004	0.005	0.007	0.009
7th run	0.003	0.003	0.004	0.006	0.008
8th run	0.003	0.004	0.005	0.007	0.008
9th run	0.003	0.003	0.006	0.008	0.011
10th run	0.003	0.003	0.005	0.006	0.008
	0.0034	0.0033	0.0048	0.0069	0.0191

Figure A.2: Speedup output for input 100.000

Cores	1	2	4	6	8
1st run	0.005	0.004	0.007	0.009	0.01
2nd run	0.004	0.005	0.004	0.006	0.01
3rd run	0.005	0.004	0.005	0.007	0.01
4th run	0.004	0.005	0.006	0.007	0.009
5th run	0.004	0.004	0.006	0.008	0.012
6th run	0.004	0.004	0.006	0.008	0.009
7th run	0.004	0.004	0.005	0.01	0.01
8th run	0.007	0.004	0.005	0.007	0.009
9th run	0.005	0.004	0.006	0.007	0.009
10th run	0.005	0.005	0.005	0.007	0.011
	0.0047	0.0043	0.0055	0.0076	0.0099

Figure A.3: Speedup output for input 1.000.000

Cores	1	2	4	6	8
1st run	0.019	0.012	0.011	0.011	0.015
2nd run	0.019	0.017	0.011	0.012	0.015
3rd run	0.022	0.024	0.012	0.011	0.012
4th run	0.019	0.015	0.013	0.013	0.017
5th run	0.019	0.011	0.011	0.012	0.013
6th run	0.022	0.012	0.012	0.011	0.011
7th run	0.022	0.012	0.012	0.016	0.016
8th run	0.022	0.013	0.012	0.011	0.012
9th run	0.022	0.013	0.012	0.014	0.013
10th run	0.019	0.014	0.012	0.016	0.013
	0.0205	0.0143	0.0118	0.0127	0.0137

Figure A.4: Speedup output for input 10.000.000

Cores	1	2	4	6	8
1st run	0.163	0.088	0.087	0.087	0.081
2nd run	0.17	0.11	0.086	0.088	0.089
3rd run	0.175	0.091	0.086	0.086	0.085
4th run	0.173	0.095	0.083	0.082	0.087
5th run	0.17	0.082	0.083	0.082	0.082
6th run	0.164	0.09	0.085	0.084	0.09
7th run	0.169	0.115	0.083	0.083	0.084
8th run	0.166	0.084	0.081	0.082	0.083
9th run	0.166	0.108	0.089	0.084	0.082
10th run	0.166	0.099	0.092	0.081	0.082
	0.1682	0.0962	0.0855	0.0839	0.0845

Figure A.5: Speedup output for input 100.000.000

Cores	1	2	4	6	8
1st run	1.582	0.844	0.763	0.763	0.777
2nd run	1.638	0.847	0.755	0.772	0.755
3rd run	1.599	0.85	0.766	0.782	0.753
4th run	1.584	1.015	0.758	0.765	0.777
5th run	1.604	0.945	0.793	0.764	0.773
6th run	1.617	0.929	0.76	0.779	0.763
7th run	1.681	0.84	0.776	0.762	0.768
8th run	1.626	0.844	0.752	0.763	0.768
9th run	1.612	0.812	0.751	0.768	0.776
10th run	1.621	0.916	0.764	0.765	0.78
	1.6164	0.8842	0.7638	0.7683	0.769

Figure A.6: Speedup output for input 50.000

Cores	1	2	4	6	8
1st run	0.004	0.005	0.005	0.008	0.011
2nd run	0.006	0.004	0.004	0.006	0.009
3rd run	0.005	0.004	0.004	0.01	0.008
4th run	0.004	0.004	0.004	0.009	0.009
5th run	0.006	0.004	0.004	0.006	0.009
6th run	0.004	0.004	0.003	0.007	0.009
7th run	0.004	0.003	0.006	0.008	0.009
8th run	0.004	0.005	0.004	0.006	0.009
9th run	0.004	0.004	0.004	0.007	0.009
10th run	0.004	0.003	0.006	0.008	0.009
	0.0045	0.004	0.0044	0.0075	0.0091

Figure A.7: Speedup output for input 500.000

Cores	1	2	4	6	8
1st run	0.011	0.007	0.008	0.01	0.011
2nd run	0.011	0.007	0.007	0.011	0.01
3rd run	0.012	0.007	0.007	0.01	0.01
4th run	0.012	0.01	0.009	0.008	0.011
5th run	0.012	0.008	0.009	0.008	0.012
6th run	0.01	0.008	0.008	0.008	0.011
7th run	0.011	0.011	0.006	0.008	0.012
8th run	0.011	0.007	0.008	0.011	0.013
9th run	0.011	0.011	0.008	0.009	0.011
10th run	0.011	0.007	0.009	0.01	0.012
	0.0112	0.0083	0.0079	0.0093	0.0113

Figure A.8: Speedup output for input 5.000.000

Cores	1	2	4	6	8
1st run	0.084	0.045	0.042	0.049	0.043
2nd run	0.091	0.043	0.046	0.045	0.044
3rd run	0.089	0.061	0.043	0.049	0.044
4th run	0.087	0.046	0.044	0.046	0.046
5th run	0.091	0.067	0.049	0.041	0.049
6th run	0.089	0.053	0.042	0.048	0.046
7th run	0.088	0.044	0.044	0.044	0.046
8th run	0.084	0.058	0.043	0.043	0.047
9th run	0.089	0.059	0.049	0.055	0.043
10th run	0.085	0.044	0.046	0.043	0.043
	0.0877	0.052	0.0448	0.0463	0.0451

Figure A.9: Speedup output for input 50.000.000

Cores	1	2	4	6	8
1st run	0.843	0.436	0.39	0.391	0.377
2nd run	0.812	0.492	0.387	0.394	0.392
3rd run	0.815	0.454	0.405	0.39	0.392
4th run	0.806	0.408	0.385	0.396	0.403
5th run	0.837	0.424	0.394	0.385	0.384
6th run	0.799	0.414	0.397	0.381	0.398
7th run	0.814	0.429	0.39	0.383	0.387
8th run	0.827	0.441	0.393	0.389	0.381
9th run	0.788	0.431	0.38	0.383	0.393
10th run	0.821	0.417	0.384	0.397	0.392
	0.8162	0.4346	0.3905	0.3889	0.3899

Figure A.10: Speedup output for input 500.000.000

Cores	1	2	4	6	8
1st run	7.99	4.61	3.815	3.884	3.768
2nd run	8.045	4.539	3.801	3.804	3.82
3rd run	7.891	4.322	3.779	3.814	3.746
4th run	8.007	4.077	3.792	3.806	3.792
5th run	8.036	4.347	3.737	3.747	3.743
6th run	7.968	4.223	3.75	3.768	3.797
7th run	8.105	4.17	3.789	3.837	3.766
8th run	7.948	4.249	3.749	3.815	3.746
9th run	7.935	4.19	3.775	3.763	3.792
10th run	7.982	4.233	3.76	3.795	3.747
	7.9907	4.296	3.7747	3.8033	3.7717

A.2 Opgave 3

A.2.1 Opgave 3 output

```
./prodcons 5 3 3 10 15
Producer 1 produced ITEM_0. Items in buffer: 1 (out of 10)
Producer 2 produced ITEM_1. Items in buffer: 2 (out of 10)
Consumer 2 consumed ITEM_0. Items in buffer: 1 (out of 10)
Consumer 1 consumed ITEM_1. Items in buffer: 0 (out of 10)
Producer 0 produced ITEM_2. Items in buffer: 1 (out of 10)
Consumer 0 consumed ITEM_2. Items in buffer: 0 (out of 10)
Producer 1 produced ITEM_3. Items in buffer: 1 (out of 10)
Consumer 2 consumed ITEM_3. Items in buffer: 0 (out of 10)
Producer 0 produced ITEM_4. Items in buffer: 1 (out of 10)
Consumer 0 consumed ITEM_4. Items in buffer: 0 (out of 10)
Producer 0 produced ITEM_5. Items in buffer: 1 (out of 10)
Consumer 1 consumed ITEM_5. Items in buffer: 0 (out of 10)
Producer 1 produced ITEM_6. Items in buffer: 1 (out of 10)
Producer 2 produced ITEM_7. Items in buffer: 2 (out of 10)
Producer 0 produced ITEM_8. Items in buffer: 3 (out of 10)
Consumer 2 consumed ITEM_6. Items in buffer: 2 (out of 10)
Consumer 0 consumed ITEM_7. Items in buffer: 1 (out of 10)
Consumer 1 consumed ITEM_8. Items in buffer: 0 (out of 10)
Producer 1 produced ITEM_9. Items in buffer: 1 (out of 10)
Consumer 0 consumed ITEM_9. Items in buffer: 0 (out of 10)
Producer 1 produced ITEM_10. Items in buffer: 1 (out of 10)
Consumer 0 consumed ITEM_10. Items in buffer: 0 (out of 10)
Producer 2 produced ITEM_11. Items in buffer: 1 (out of 10)
Producer 0 produced ITEM_12. Items in buffer: 2 (out of 10)
Consumer 2 consumed ITEM_11. Items in buffer: 1 (out of 10)
Consumer 1 consumed ITEM_12. Items in buffer: 0 (out of 10)
Producer 0 produced ITEM_13. Items in buffer: 1 (out of 10)
Producer 2 produced ITEM_14. Items in buffer: 2 (out of 10)
Consumer 0 consumed ITEM_13. Items in buffer: 1 (out of 10)
Consumer 1 consumed ITEM_14. Items in buffer: 0 (out of 10)
```

B Kode

B.1 Opgave 1

B.1.1 Makefile til mulsum

```
1  all: mulsum
2  LIBS= -lpthread -lm
3  CC = gcc
4  mulsum: mulsum.o
5          ${CC} -o $@ mulsum.o ${LIBS}
6  clean:
7          rm -rf *o mulsum
```

B.1.2 mulsum

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5  #include <math.h>
6
7  typedef struct calc_struct calc;
8
9  struct calc_struct {
10         double minimum_number;
11         double maximum_number;
12         double sumsqrt;
13 };
14
15 void *TaskCode(void *argument)
16 {
17         // Calculation
18         calc *c = ((calc *) argument);
19         double n;
20         double sum = 0;
21
22         for(n = c->minimum_number; n < (c->maximum_number + 1); ++n)
23         {
24                 sum = sum + sqrt(n);
25         }
26
27         c->sumsqrt = sum;
28
29         pthread_exit(0);
30 }
31
```

```
32 int main(int argc, char *argv[])
33 {
34         int number_of_threads = atoi(argv[3]);
35         int input_integer = atoi(argv[2]);
36 
37         calc calc_result[number_of_threads];
38         pthread_t threads[number_of_threads];
39 
40         double input_double = (double) input_integer;
41         double work = input_double / (double) number_of_threads;
42 
43         int n;
44         for(n = 0; n < number_of_threads; ++n)
45         {
46 
47                 double minnum = floor(work * n) + 1;
48                 double maxnum = floor(work * (n + 1));
49                 calc_result[n].minimum_number = minnum;
50                 calc_result[n].maximum_number = maxnum;
51                 calc_result[n].sumsqrt = 0;
52         }
53 
54         int rc, i;
55 
56         /* create all threads */
57         for (i=0; i<number_of_threads; ++i) {
58                 rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &calc_result[i]);
59                 assert(0 == rc);
60         }
61 
62         /* wait for all threads to complete */
63         for (i=0; i<number_of_threads; ++i) {
64                 rc = pthread_join(threads[i], NULL);
65                 assert(0 == rc);
66         }
67 
68         double sumtotal = 0;
69         int p;
70 
71         for(p = 0; p < number_of_threads; p++)
72         {
73                 sumtotal = sumtotal + calc_result[p].sumsqrt;
74         }
75         printf("Total amount: %f \n", sumtotal);
76 
77         exit(EXIT_SUCCESS);
78 }
```

B.2 Opgave 2

B.2.1 Makefile til FIFO

```
1  all: fifoTestNoThreads fifoTestThreads
2
3  OBJS = list.o
4  LIBS = -lm -lpthread
5
6  fifoTestNoThreads: testNoThreads.o ${OBJS}
7          gcc -o $@ testNoThreads.o ${OBJS} ${LIBS}
8
9  fifoTestThreads: testThreads.o ${OBJS}
10         gcc -o $@ testThreads.o ${OBJS} ${LIBS}
11
12 clean:
13         rm -rf *o fifo
```

B.2.2 list.c

```
1  /*****************************************************************************
2     list.c
3
4     Implementation of simple linked list defined in list.h.
5
6  *****************************************************************************/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11 #include <pthread.h>
12 #include "list.h"
13
14 /* list_new: return a new list structure */
15 List *list_new(void)
16 {
17         List *l;
18
19         l = (List *) malloc(sizeof(List));
20         l->len = 0;
21
22         if (pthread_mutex_init(&l->lock, NULL) != 0)
23     {
24         printf("\n mutex init failed\n");
25     }
26
27         /* insert root element which should never be removed */
28         l->first = l->last = (Node *) malloc(sizeof(Node));
29         l->first->elm = NULL;
30         l->first->next = NULL;
31         return l;
```

```
32 }
33
34 /* list_add: add node n to list l as the last element */
35 void list_add(List *l, Node *n)
36 {
37         pthread_mutex_lock(&l->lock);
38         l->last->next = n;
39         l->last = n;
40         pthread_mutex_unlock(&l->lock);
41 }
42
43 /* list_remove: remove and return the first (non-root) element from list l */
44 Node *list_remove(List *l)
45 {
46         pthread_mutex_lock(&l->lock);
47         Node *n;
48
49         n = l->first->next;
50         l->first->next = n->next;
51
52         if(l->first->next == NULL)
53         {
54                 l->last = l->first;
55         }
56
57         pthread_mutex_unlock(&l->lock);
58
59         return n;
60 }
61
62 /* node_new: return a new node structure */
63 Node *node_new(void)
64 {
65         Node *n;
66         n = (Node *) malloc(sizeof(Node));
67         n->elm = NULL;
68         n->next = NULL;
69         n->previous = NULL;
70         return n;
71 }
72
73 /* node_new_str: return a new node structure, where elm points to new copy of s */
74 Node *node_new_str(char *s)
75 {
76         Node *n;
77         n = (Node *) malloc(sizeof(Node));
78         n->elm = (void *) malloc((strlen(s)+1) * sizeof(char));
79         strcpy((char *) n->elm, s);
80         n->next = NULL;
81         n->previous = NULL;
82         return n;
```

```
83 }
```

B.2.3 list.h

```
1  /*************************************************************************
2     list.h
3
4     Header file with definition of a simple linked list.
5
6  *************************************************************************/
7
8  #ifndef _LIST_H
9  #define _LIST_H
10
11 /* structures */
12 typedef struct node {
13         void *elm; /* use void type for generality; we cast the element's type to void type
14         struct node *next;
15         struct node *previous;
16 } Node;
17
18 typedef struct list {
19         int len;
20         pthread_mutex_t lock;
21         Node *first;
22         Node *last;
23 } List;
24
25 /* functions */
26 List *list_new(void);             /* return a new list structure */
27 void list_add(List *l, Node *n); /* add node n to list l as the last element */
28 Node *list_remove(List *l);       /* remove and return the first element from list l*/
29 Node *node_new(void);             /* return a new node structure */
30 Node *node_new_str(char *s);      /* return a new node structure, where elm points to new co
31
32 #endif
```

B.2.4 testNoThreads.c

```
1  /*************************************************************************
2
3  testNoThreads.c
4
5  Testign that the FIFO list actually works as we want it to.
6
7  *************************************************************************/
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <pthread.h>
12 #include "list.h"
```

```
13
14 int main(int argc, char* argv[])
15 {
16     // Creates a FIFO list;
17     List *fifo = list_new();
18
19     // Adds two elements to the list.
20     list_add(fifo, node_new_str("s1"));
21     list_add(fifo, node_new_str("s2"));
22
23     // Removes the first element from the list and checks that is it not null
24     Node *n1 = list_remove(fifo);
25     if (n1 == NULL) { printf("Error no elements in list\n"); exit(-1);}
26     // Removes the second element from the list and checks that is it not null
27     Node *n2 = list_remove(fifo);
28     if (n2 == NULL) { printf("Error no elements in list\n"); exit(-1);}
29
30     // Saves the elm of the two nodes as strings.
31     char *n1elm = (char *)n1->elm;
32     char *n2elm = (char *)n2->elm;
33
34     // Prints the two elements.
35     printf("%s\n%s\n", n1elm, n2elm);
36
37     return 0;
38 }
```

B.2.5 testThreads.c

```
1 #include <assert.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <pthread.h>
5 #include "list.h"
6
7 List *fifo;
8 int number_of_elements_per_thread;
9
10 // The code for the threads.
11 void *TaskCode(void *argument)
12 {
13     // Converts the parameter to an integer that holds the number
14     // of the thread.
15     int *threadNo = (int *) argument;
16
17     int i;
18     // Adds elements to the list.
19     for (i = 0; i < number_of_elements_per_thread; ++i)
20     {
21         char x[250];
22         char *s = "Thread";
```

```
23          sprintf(x, "%s #%d, element %d", s, *threadNo + 1, i);
24          list_add(fifo, node_new_str(x));
25      }
26
27      // Removes elements from the list.
28      for (i = 0; i < number_of_elements_per_thread; ++i)
29      {
30          Node *node = list_remove(fifo);
31          char *nodeElem = (char *)node->elm;
32          printf("%s\n", nodeElem);
33      }
34          free(argument);
35      pthread_exit(0);
36  }
37
38  // Method for testing that the list works with threads.
39  int main(int argc, char* argv[])
40  {
41      int rc, i;
42      int number_of_threads = atoi(argv[3]);
43      number_of_elements_per_thread = atoi(argv[2]);
44      fifo = list_new();
45      pthread_t threads[number_of_threads];
46
47      // Creates all the threads and makes them do what they are supposed to.
48      for (i=0; i<number_of_threads; ++i)
49      {
50                  int *r = malloc(sizeof(int));
51                  *r = i;
52          rc = pthread_create(&threads[i], NULL, TaskCode, (void *) r);
53          assert(0 == rc);
54      }
55
56      // Wait for all threads to finish
57      for (i=0; i<number_of_threads; ++i)
58      {
59          rc = pthread_join(threads[i], NULL);
60          assert(0 == rc);
61      }
62
63      return 0;
64  }
```

B.3 Opgave 3

B.3.1 Makefile til prodcons

```
1  all: prodcons
2
3  OBJS = List/list.o
4  LIBS = -lrt -lpthread
5
6  prodcons: prodCons.o ${OBJS}
7          gcc -o $@ prodCons.o ${OBJS} ${LIBS}
8
9  clean:
10         rm -rf *o prodcons
```

B.3.2 prodcons.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5  #include "List/list.h"
6  #include "prodCons.h"
7
8  // List
9  List *itemList; // The product buffer.
10
11 // Locks
12 pthread_mutex_t produce_lock; // Lock for produced_products.
13 pthread_mutex_t consume_lock; // Lock for consumed_products.
14 pthread_mutex_t products_buffer_lock; // Lock for products_in_buffer
15
16 // Semaphores
17 sem_t empty; // Number of empty slots in the buffer.
18 sem_t full; // Number of filled slots in the buffer.
19 sem_t products_remaining; // Number of products still to be produced.
20
21 // Consumer and producer variables
22 int number_of_producers; // Number of product producers.
23 int number_of_consumers; // Number of product consumers.
24
25 // Product variables
26 int total_number_of_products; // Number of products to be produced in total.
27 int produced_products = 0; // Number of produced products.
28 int consumed_products = 0; // Number of consumed products.
29 int products_in_buffer = 0; // Amount of products in the buffer
30
31 // Buffer size
32 int buffer_size;
33
34 int main(int argc, char* argv[])
```

```
35 {
36         if(atoi(argv[1]) != 5)
37         {
38                 printf("Wrong number of input arguments! ProdCons needs 5 input arguments.\
39                 printf("Argument 1: Number of arguments.\nArgument 2: Number of producers.\
40                 printf("Argument 4: Size of the buffer.\nArgument 5: Total amount of produ
41                 exit(EXIT_FAILURE);
42         }
43
44         // Initialise variables from input arguments
45         number_of_producers = atoi(argv[2]);
46         number_of_consumers = atoi(argv[3]);
47         total_number_of_products = atoi(argv[5]);
48         buffer_size = atoi(argv[4]);
49
50         // Initialise semaphores
51         if(sem_init(&empty, 0, buffer_size) != 0)
52         {
53                 printf("\n Failed to initialise empty semaphore\n");
54         }
55         if(sem_init(&full, 0, 0) != 0)
56         {
57                 printf("\n Failed to initialise full semaphore\n");
58         }
59         if(sem_init(&products_remaining, 0, total_number_of_products) != 0)
60         {
61                 printf("\n Failed to initialise total_number_of_products semaphore\n");
62         }
63
64         // Initialise list
65         itemList = list_new();
66
67         // Initialise locks
68         if(pthread_mutex_init(&produce_lock, NULL) != 0)
69     {
70         printf("\n Failed to initialise produce_lock\n");
71     }
72
73         if(pthread_mutex_init(&consume_lock, NULL) != 0)
74     {
75         printf("\n Failed to initialise consumed_lock\n");
76     }
77
78         if(pthread_mutex_init(&products_buffer_lock, NULL) != 0)
79     {
80         printf("\n Failed to initialise products_buffer_lock\n");
81     }
82
83         // Consumer and producer thread arrays
84         pthread_t consumer_thread_ids[number_of_consumers];
85         pthread_t producer_thread_ids[number_of_producers];
```

```
86
87          // Create threads
88          int tn;
89          for (tn = 0; tn < number_of_consumers; tn++){
90                  int *cn = malloc(sizeof(int));
91                  *cn = tn;
92          if(pthread_create(&consumer_thread_ids[tn], NULL, &consumer, (void*) cn))
93                  {
94                  printf("Failed to create consumer number %d \n", tn);
95          }
96      }
97          for (tn = 0; tn < number_of_producers; tn++){
98          int *pn = malloc(sizeof(int));
99                  *pn = tn;
100                 if(pthread_create(&producer_thread_ids[tn], NULL, &producer, (void*) pn))
101                 {
102                 printf("Failed to create producer number %d \n", tn);
103         }
104     }
105
106         // Join threads
107         for (tn = 0; tn < number_of_consumers; tn++)
108         {
109         pthread_join(consumer_thread_ids[tn], NULL);
110     }
111         for (tn = 0; tn < number_of_producers; tn++)
112         {
113         pthread_join(producer_thread_ids[tn], NULL);
114     }
115
116         return 0;
117 }
118
119 // Producer code
120 void *producer(void *argument)
121 {
122         int *prodNo = (int *) argument; // Producers identifying number
123         Node *node; // Produced node
124
125         while(1)
126         {
127                 // Produce or stop
128                 int tw = sem_trywait(&products_remaining);
129                 if(tw == -1) // If there are no more products to be produced, end thread.
130                 {
131                         pthread_exit(0);
132                 }
133                 else if(tw == 0) // There are more products be produced.
134                 {
135                          node = produceProduct(); // Final product
136                 }
```

```
137                 else
138                 {
139                         printf("Some error occured when trying to wait on products_remainir
140                 }
141
142                 // Add product to buffer
143                 sem_wait(&empty); // Wait for room in buffer.
144                 list_add(itemList, node); // Add node to list.
145
146                 // Increase count of number products in buffer.
147                 pthread_mutex_lock(&products_buffer_lock);
148                 products_in_buffer++;
149                 printf("Producer %d produced %s. Items in buffer: %d (out of %d) \n", *prod
150                 pthread_mutex_unlock(&products_buffer_lock);
151
152                 sem_post(&full); // Signal full so buffer space is decreased by 1.
153
154                 // Sleep for random time.
155                 sleepRandom(10000);
156         }
157 }
158
159 // Consumer code
160 void *consumer(void *argument)
161 {
162         int *consNo = (int *) argument; // Consumers identifying number
163         Node *node; // Consumed node.
164
165         while(1){
166                 pthread_mutex_lock(&consume_lock); // Lock consumed_products.
167
168                 if(consumed_products == total_number_of_products) // If we have consumed a
169                 {
170                         pthread_mutex_unlock(&consume_lock); // Release lock on consumed_p
171                         pthread_exit(0); // Exit thread.
172                 }
173
174                 sem_wait(&full); // Wait for a product to exist in the buffer.
175                 node = list_remove(itemList); // Remove product from buffer.
176
177                 // Decrease count of number products in buffer.
178                 pthread_mutex_lock(&products_buffer_lock);
179                 products_in_buffer--;
180                 pthread_mutex_unlock(&products_buffer_lock);
181
182                 consumed_products++; // Increase amount of consumed products
183                 pthread_mutex_unlock(&consume_lock); // Unlock consumed_products.
184                 sem_post(&empty); // Signal empty so buffer space is increased by 1.
185
186                 printf("Consumer %d consumed %s. Items in buffer: %d (out of %d) \n", *cons
187
```

```
188                     sleepRandom(10000);
189             }
190     }
191
192     // Produces a new node.
193     Node *produceProduct()
194     {
195             Node* node;
196
197             // Lock produced_products
198             pthread_mutex_lock(&produce_lock);
199
200             // Create node
201             char tmp[250];
202             sprintf(tmp, "ITEM %d", produced_products);
203
204             produced_products++; // Increased produced products.
205
206             pthread_mutex_unlock(&produce_lock); // Unlocked produced_products.
207
208             node = node_new_str(tmp); // Initialisation of the new node.
209
210             return node;
211     }
212
213     // Code taken from assignment description v2
214     void sleepRandom(float wait_time_ms)
215     {
216             struct timeval tv;
217             gettimeofday(&tv, NULL);
218             srand(tv.tv_usec);
219             wait_time_ms = ((float)rand())*wait_time_ms / (float)RAND_MAX;
220             usleep((int) (wait_time_ms * 1e3f)); // convert from ms to us
221     }
```

B.3.3 prodcons.h

```
1   // Header
2   void *producer(void*);
3   void *consumer(void*);
4   Node *produceProduct();
5   void sleepRandom(float wait_time_ms);
```

B.4 Opgave 4

B.4.1 Makefile til banker

```
all: banker
LIBS= -lpthread
CC = gcc
mulsum: banker.o
	${CC} -o $@ banker.o ${LIBS}
clean:
	rm -rf *o banker
```

B.4.2 banker.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<pthread.h>
#include<math.h>

typedef struct state {
  int *resource;
  int *available;
  int **max;
  int **allocation;
  int **need;
} State;

// Global variables
int m, n;
State *s = NULL;

// Mutex for access to state.
pthread_mutex_t state_mutex;

/* Random sleep function */
void Sleep(float wait_time_ms)
{
  // add randomness
  wait_time_ms = ((float)rand())*wait_time_ms / (float)RAND_MAX;
  usleep((int) (wait_time_ms * 1e3f)); // convert from ms to us
}

/* Allocate resources in request for process i, only if it
   results in a safe state and return 1, else return 0 */
int resource_request(int i, int *request)
{
  pthread_mutex_lock(&state_mutex);
  int j;
  for(j = 0; j < n; j++)
  {
```

```
38          if(request[j] > s->need[i][j])
39          {
40                  fprintf(stderr, "Request higher than Need for process %d!", i);
41                  pthread_mutex_unlock(&state_mutex);
42                  exit(0);
43          }
44    }
45
46    for(j = 0; j < n; j++)
47    {
48          if(request[j] > s->available[j])
49          {
50                  pthread_mutex_unlock(&state_mutex);
51                  return 0;
52          }
53    }
54
55    for(j = 0; j < n; j++)
56    {
57          s->available[j] = s->available[j] - request[j];
58          s->allocation[i][j] = s->allocation[i][j] + request[j];
59          s->need[i][j] = s->need[i][j] - request[j];
60    }
61
62    if(!safety_check())
63    {
64          for(j = 0; j < n; j++)
65          {
66                  s->available[j] = s->available[j] + request[j];
67                  s->allocation[i][j] = s->allocation[i][j] - request[j];
68                  s->need[i][j] = s->need[i][j] + request[j];
69          }
70          pthread_mutex_unlock(&state_mutex);
71          return 0;
72    }
73    else
74    {
75          pthread_mutex_unlock(&state_mutex);
76          return 1;
77    }
78  }
79
80  /* Release the resources in request for process i */
81  void resource_release(int i, int *request)
82  {
83    pthread_mutex_lock(&state_mutex);
84    int j;
85    for(j = 0; j < n; j++)
86    {
87                  s->available[j] = s->available[j] + request[j];
88                  s->allocation[i][j] = s->allocation[i][j] - request[j];
```

```
89                  s->need[i][j] = s->need[i][j] + request[j];
90          }
91          pthread_mutex_unlock(&state_mutex);
92  }
93
94  // Safety algorithm
95  int safety_check()
96  {
97          // Variables
98          int i, j, lt;
99          int work[n], finish[m];
100
101         // Step 1
102         // Initialise work
103         for(i = 0; i < n; i++)
104         {
105                 work[i] = s->available[i];
106         }
107
108         // Initialise finish
109         for(i = 0; i < m; i++)
110         {
111                 finish[i] = 0;
112         }
113
114         int tryagain = 1;
115
116         while(tryagain)
117         {
118                 tryagain = 0;
119                 // Step 2
120                 for(i = 0; i < m; i++)
121                 {
122                         if(finish[i] != 1)
123                         {
124                                 int j, comparison = 0;
125                                 for(j = 0; j < n; j++)
126                                 {
127                                         if(s->need[i][j] > work[j])
128                                         {
129                                                 comparison = -1;
130                                                 break;
131                                         }
132                                 }
133
134                                 // Step 3
135                                 if(comparison != -1)
136                                 {
137                                         int k;
138                                         for(k = 0; k < n; k++)
139                                         {
```

```
140                                                     work[k] = work[k] + s->allocation[i][k];
141                                             }
142                                             finish[i] = 1;
143                                             tryagain = 1;
144                                     }
145                             }
146                     }
147             }
148
149             // Step 4
150             for(i = 0; i < m; i++)
151             {
152                     if(finish[i] != 1)
153                     {
154                             return 0;
155                     }
156             }
157
158             return 1;
159 }
160
161 /* Generate a request vector */
162 void generate_request(int i, int *request)
163 {
164   pthread_mutex_lock(&state_mutex);
165   int j, sum = 0;
166   while (!sum) {
167     for (j = 0;j < n; j++) {
168       request[j] = round(s->need[i][j] * ((double)rand())/ (double)RAND_MAX);
169       sum += request[j];
170     }
171   }
172   pthread_mutex_unlock(&state_mutex);
173   printf("Process %d: Requesting resources.\n",i);
174 }
175
176 /* Generate a release vector */
177 void generate_release(int i, int *request)
178 {
179   pthread_mutex_lock(&state_mutex);
180   int j, sum = 0;
181   while (!sum) {
182     for (j = 0;j < n; j++) {
183       request[j] = round(s->allocation[i][j] * ((double)rand())/ (double)RAND_MAX);
184       sum += request[j];
185     }
186   }
187   pthread_mutex_unlock(&state_mutex);
188   printf("Process %d: Releasing resources.\n",i);
189 }
190
```

```
191 /* Threads starts here */
192 void *process_thread(void *param)
193 {
194   /* Process number */
195   int i = (int) (long) param, j;
196   /* Allocate request vector */
197   int *request = malloc(n*sizeof(int));
198   while (1) {
199     /* Generate request */
200     generate_request(i, request);
201     while (!resource_request(i, request)) {
202       /* Wait */
203       Sleep(100);
204     }
205
206     /* Generate release */
207     generate_release(i, request);
208     /* Release resources */
209     resource_release(i, request);
210     /* Wait */
211     Sleep(1000);
212   }
213   free(request);
214 }
215
216 int main(int argc, char* argv[])
217 {
218   /* Get size of current state as input */
219   int i, j;
220   printf("Number of processes: \n");
221   scanf("%d", &m);
222   printf("Number of resources: \n");
223   scanf("%d", &n);
224
225   /* Allocate memory for state */
226   s = malloc(sizeof(State));
227   s->resource = malloc(sizeof(int) * n);
228   s->available = malloc(sizeof(int) * n);
229   s->max = malloc(sizeof(int *) * m);
230   s->allocation = malloc(sizeof(int *) * m);
231   s->need = malloc(sizeof(int *) * m);
232
233   for(i = 0; i < m; i++)
234   {
235         s->max[i] = malloc(sizeof(int) * n);
236         s->allocation[i] = malloc(sizeof(int) * n);
237         s->need[i] = malloc(sizeof(int) * n);
238   }
239
240   /* Get current state as input */
241   printf("Resource vector: \n");
```

```
242   for(i = 0; i < n; i++)
243     scanf("%d", &s->resource[i]);
244   printf("Enter max matrix: \n");
245   for(i = 0;i < m; i++)
246     for(j = 0;j < n; j++)
247       scanf("%d", &s->max[i][j]);
248   printf("Enter allocation matrix:");
249   for(i = 0; i < m; i++)
250     for(j = 0; j < n; j++) {
251       scanf("%d", &s->allocation[i][j]);
252     }
253   printf("\n");
254
255   /* Calcuate the need matrix */
256   for(i = 0; i < m; i++)
257     for(j = 0; j < n; j++)
258       s->need[i][j] = s->max[i][j]-s->allocation[i][j];
259
260   /* Calcuate the availability vector */
261   for(j = 0; j < n; j++) {
262     int sum = 0;
263     for(i = 0; i < m; i++)
264       sum += s->allocation[i][j];
265     s->available[j] = s->resource[j] - sum;
266   }
267
268   /* Output need matrix and availability vector */
269   printf("Need matrix:\n");
270   for(i = 0; i < n; i++)
271     printf("R%d ", i+1);
272   printf("\n");
273   for(i = 0; i < m; i++) {
274     for(j = 0; j < n; j++)
275       printf("%d  ",s->need[i][j]);
276     printf("\n");
277   }
278   printf("Availability vector:\n");
279   for(i = 0; i < n; i++)
280     printf("R%d ", i+1);
281   printf("\n");
282   for(j = 0; j < n; j++)
283     printf("%d  ",s->available[j]);
284   printf("\n");
285
286   if(safety_check())
287   {
288     printf("State was safe. \n");
289   }
290   else
291   {
292         fprintf(stderr, "State was not safe.\n");
```

```
293             return 1;
294     }
295
296     /* Seed the random number generator */
297     struct timeval tv;
298     gettimeofday(&tv, NULL);
299     srand(tv.tv_usec);
300
301     /* Create m threads */
302     pthread_t *tid = malloc(m*sizeof(pthread_t));
303     for (i = 0; i < m; i++)
304       pthread_create(&tid[i], NULL, process_thread, (void *) (long) i);
305
306     /* Wait for threads to finish */
307     pthread_exit(0);
308     free(tid);
309
310     /* Free state memory */
311             for (i = 0; i < m; i++) {
312                     free(s->max[i]);
313                     free(s->allocation[i]);
314                     free(s->need[i]);
315             }
316
317             free(s->resource);
318             free(s->available);
319             free(s->max);
320             free(s->allocation);
321             free(s->need);
322             free(s);
323 }
```