IT University of Copenhagen

Operativsystemer og C

BOSC

Obligatorisk Opgave 2

Author:
Omar Khan (omsh@itu.dk)
Mads Ljungberg (malj@itu.dk)

November 6, 2015

Contents

1 Introduktion

Formålet med denn rapport er at give indsigt i brugen af tråde i et operativsystem, ved brug af pthread, mutex og semaphore. I rapporten gennemgås teorien bag disse værktøjer og hvordan de implementeres i programmer for at opnå arbejde gennem flere tråde i et system. Kildekoden for programmerne er vedlagt som Appendix A.

2 Teori

2.1 Pthreads

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

2.2 Mutex

Mutex bruges når man har kritiske sektioner i sin kode og vil synkronisere sine processertråde. Hvis flere tråde bruger en værdi i den kritiske section, og ændre på den parrallelt, vides det ikke hvornår trådene ændre på værdien. Dette kan skabe problemer og giver forkerte læsninger også kaldet dirty reads.

Ved brug af mutex kan man låse disse sektioner af, så andre processer/tråde ikke kan tilgå sektionen, når processen med låsen er færdig i sektionen vil mutex frigive låsen og lade andre processer/tråde tilgå den kritiske sektion.

2.3 Semaphores

Semaphores er en anden variation til at låse kritiske sektioner af med. De følger et andet princip med, at de har en variabel med sig som betegner hvor mange pladser der er til processer/tråde kan køre simultant med hinanden. Når alle pladser er i brug vil semaphoren blokerer adgangen til sectionen, og først give adgang når der er plads igen.

Semaphorens metoder,`sem_wait()` og `sem_post()`, har en virkning på den variabel som semaphoren har med sig. `sem_wait()` vil sænke variablen med 1 – reducere antallet a ledige pladser, og post vil hæve variablen med 1 – øge antallet af ledige pladser. Når variablen når 0 vil wait vente på et post.

2.3.1 Pthread condition vs semaphores

Hvor semaphores blokerer kritiske sektioner vil pthread condition blokere på værdier den har brug fra et andet sted. Hvis man har 2 tråde, hvor tråd A gør brug af en global variable x som tråd B ændrer, kan man i tråd A vente på at B har ændret præcis denne variabel x.

Forskellen mellem pthread condition og semaphore er, at pthread condition blokerer på værdier fremfor sektioner, og derved kun blokerer når det er højst nødvendigt.

2.4 Concurenxy Control

Når man har noget data som mange skal bruge på samme tid, hvordan opnås dette uden at ændringerne der bliver foretaget ikke ender med at være forkerte, når andre skal bruge dem?

Dette er grundlaget for concurrency control, at sikre at samtidige ændringer håndteres korrekt og efter hensigten. I concurency control har man transaktioner med processer som udfører nogle handlinger der generelt betegnes som læse- og skrivehandlinger. Hvis man har to transaktioner som begge har adgang til ressource x, så kan en handling se sådan ud:

- $read(x), write(x, value)$

Hvis begge nu skulle lave en $write(x, 34)$ og $write(x, 1000)$, hvilken skulle så være den der får lov til at ændre den variabel?

Besvarelsen af det spørgsmål afgør om hvorvidt den næste process der laver en $read(x)$ ender med et resultat der er brugbart eller ej. For at undgå dette skal man sørge for at ens transaktioner er seriel ækvivalens. Dette kan gøres på mange forskellige måder. En af dem er som mutex, hvor man låser den sektion der er data sensitiv af, indtil man er sikker på at ændringerne har taget efekt. Dette gør at man ikke får dirty reads. Det er dog ikke nok til at kalde det en seriel ækvivalent transaktion. For at de kan være det så kræver det, at dataen ser ud som den ville hvis en transaktion havde kørt den isoleret.

2.5 Banker's Algoritme

Banker's algoritme er en ressource alokerings algoritme som bruges til hovedsageligt til at undgå deadlock situationer.

Algoritmen benytter matriser til at allokere ressourcer, R, til processer, P og holde styr på hvor mange ressourcer af en ressource type en process har. Den benytter to vektorer, *available* med længen n som angiver mængden ressourcer

af en given type R der er tilgængelige på et givent tidspunkt, og en vektor $ressource$ der angiver de maksimalt tilgængelige ressourcer af en type R.

I algoritmen er der 3 matriser i alt, med størrelsen $m \times n$, hvor m er antallet af processer og n er antallet af resourse typer R. $max[m \times n]$ er en matrise som holder styr på hvor mange resourser en process kan modtage. $need[m \times n]$ er en matrise som angiver hvor mange ressourcer en given process mangler for de specifikke ressourcetyper R. $allocated[m \times n]$ er en matrise som håndtere allokeringen af ressourcer på processerne på et givent tidspunkt. Matrisen $need$ er beregnet ud fra $max - allocated$.

Måden hvorpå algoritmen kan afgøre om det er sikkert at allokere ressourcer til en proces er ved at tjekke om tilstanden efter ressourcerne er allokeret, er en sikker tilstand. En sikker tilstand opnås når alle processer kan færdiggøres i en sikker sekvens. For at opnå en sikker tilstand kan man bruge en metode der benytter sig af to variabler, en vektor $Work$, der afspejler $available$ vektoren og en bool liste $Finish[m]$ med længden $m - 1$, altså antallet af processer. Disse variabler bliver brugt til at afgøre om en tilstand er sikker således:

1. Hvis $Finish[i] == false$ og $need[i] \leq Work$ er sandt forsæt ellers gå til trin 3
2. Sæt $Work = Work + allocation[i]$ og $Finish[i] = true$ gå til trin 1
3. Hvis $Finish[i] == true$ for alle i, hvor $0 \leq i < m$.

Hvis de ovenstående trin kan lade sig gøre, betyder dette, at der findes minimum en sikker sekvens hvorledes processerne kan få allokeret ressourcer til at udføre deres arbejde.

Teorien bag håndtering af en forespørgsel er at validere forespørgslen ved at tjekke at $request[i] \leq need[i]$ og $request[i] \leq available[i]$. Men det er kke nok med blot at validere forespørgslen, da det kan være forespørgslen invalidere andre processers kørsel og skaber en usikker tilstand, så man simulerer allokeringen af ressourcer og benytter ovenstående metode til at se om systemet stadig er i en sikker tilstand, hvis ikke annulleres forespørgslen således at de må vente og prøve senere.

3 Implementation

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

3.1 Sum(Sqrt)

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

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

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

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

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

3.1.1 Programtid

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

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

```
1 ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode$ ./sqrtsum
     100000000 1
2 sqrtsum = 666666671666.567017
3 Total time(ms): 1690
4
```

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

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

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

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

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

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

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

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

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

tilfælde, men antager, at det skyldes operationens simplicitet.

3.2 Linked List

Linked list er en datastruktur der er bestående af noder med to værdier, deres element værdi(kunne f.eks. være en `string`, `int` osv.) og den næste node i listen. Selve listen kender til sin første og sidste node samt sin længde. Når listen er tom, dvs. længden er nul, er der kun en node i den og det er `first` som peger på `NULL` som den næste node i listen. `first` kendes som root og kan/må ikke fjernes. Listen som vi skal implementere er en FIFO(first-in-first-out), hvilket betyder at når vi tilføjer en node ryger den bagerest i kæden og bliver til den sidste node. Omvendt når vi fjerner en node skal den første ikke root node fjernes.

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

3.2.1 Tilføj og Fjern

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

```
1 if(l->len == 0)
2 {
3     l->first->next = n;
4     l->last = n;
5 }
```

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

```
1 else
2 {
3     l->last->next = n;
4     l->last = n;
5 }
```

Til sidst skal længden incrementes med 1.

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

```
1 if(l->len > 0)
2 {
3     n = l->first->next;
4     l->first->next = n->next;
5     l->len -= 1;
6 }
```

3.2.2 Problemer med flere tråde

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

3.2.3 Mutex og Linked List

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

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

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

3.3 Producer-Consumer problemet

Producer-Consumer problemet er et velkendt problem, hvor producenter producerer varer, som consumers konsumerer. Problemet ligger i at stoppe consumers med at konsumerer vare når der ikke er flere varer og ligeledes at stoppe producenter med at producerer varer når lageret er fyldt. Dette kan gøres ved

brug af semaphores, da den netop har den funktionalitet der efterspørges, da vi kan waite og poste, som afklaret i teorien. I programmet har vi implementeret en `struct PC`, der indeholder vores linked list, der fungerer som lager, og tre semaphores, `full` til at indikerer vare i lageret, `empty` til at indikerer plads i lageret og `mutex` der agerer som en mutex.

Programmet skal tage følgende fire input:

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

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

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

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

```
1  void *producer(void *param)
2  {
3      .
4      sem_wait(&empty);
5      .
6      sem_post(&full);
7      .
8  }
9
10 void *consumer(void *param)
11 {
12     .
13     sem_wait(&full);
14     .
15     sem_post(&empty);
16     .
17 }
```

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

3.4 Banker's Algoritme

Til vores implementation af Banker's algoritme er vi blevet tildelt en fil med den grundliggende implementation af algoritmens struktur. I den givet implementation er der en `struct State` som består af de elementer Banker's algoritme kræver forklaret i Teori afsnittet. Det første der manglede at blive implementeret i filen var allokeringen af hukommelse til `State`. Til dette anvender vi `malloc()` og `sizeof()` funktionerne. Med disse funktioner kan vi beregne det antal bytes i hukommelsen vi kommer til at anvende. Der hvor dette kan være problematisk er når der skal allokeres hukommelse til arrays og 2D-arrays. I et array skal man huske at allokere hukommelse i forhold til dens længde, så derfor ganges dette med `sizeof()` af typen. For 2D-arrays kræver det to skridt, hvor i det første skridt tildeles hukommelse af det andet array og i det andet skridt ved brug af en loop allokeres hukommelse til det første array.

```
1  s = (State *)malloc(sizeof(State));
2  s->resource = (int *)malloc(sizeof(int) * n);
3  s->available = (int *)malloc(sizeof(int) * n);
4  s->max = (int **)malloc(sizeof(int *) * n);
5  s->allocation = (int **)malloc(sizeof(int *) * n);
6  s->need = (int **)malloc(sizeof(int *) * n);
7
8  for(i = 0; i < m; i++)
9  {
10     s->max[i] = (int *)malloc(sizeof(int) * m);
11     s->allocation[i] = (int *)malloc(sizeof(int) * m);
12     s->need[i] = (int *)malloc(sizeof(int) * m);
13 }
```

Det næste handlede om at tjekke om programmet var i en sikker tilstand eller en usikker tilstand. Ved at følge teorien lavede vi en metode `checksafety()` der benytter sig af to arrays, `work` og `finish`. Implementationen følger teorien ved at tjekke om $finish[i] == 0$ og efterfølgende tjekke om $need[i][j] \leq work[j]$. Metoden simulerer kørsel af processer i en while-løkke, der tjekker de to nævnte kriterier. Hvis kriterierne er sande frigives ressourcerne til `work` og `finish` sættes til 1. Til sidst tjekkes der om der er nogle processer der stadig ikke kan køres og der returneres 0 hvis der er og 1 hvis der ikke er.

Med denne metode implementeret fokuseret vi på funktionen der skal benytte den mest, og det er `resource_request()`, som skal tage imod forespørgsler om ressourcer og afgøre ved brug af `checksafety()` om det er sikkert at allokere ressourcerne. Denne metode tjekker først om forespørgslen er mindre eller lig behovet for processen og om der er nok ledige ressourcer på nuværende tidspunkt. Hvis disse to kriterier er opfyldt, laves der en backup af processens nuværende sikre tilstand, hvorefter ressourcerne allokeres. Efter dette benyttes `checksafety()` til at afgøre om den nye tilstand er sikker. Hvis den ikke er sikker, benyttes backup variablerne til at gå tilbage til den tidligere sikre tilstand.

Til sidst implementeret vi funktionen `resource_release()`, der skal frigive ressourcer fra en proces afhængig af en forespørgsel. Dette er ret ligetil da det eneste vi skal sikre er at forespørgslen ikke er større end de allokeret ressourcer for den givne process.

4 Testing

4.1 Sum(Sqrt)

For at teste dette program har vi udregnet nogle små sum af sqrt's som vi har tjekket op mod det endelige resultat.

```
./sqrtsum 4 2
sqrtsum = 6.146264

./sqrtsum 6 3
sqrtsum = 10.831822
```

Da disse har vist sig at være korrekte har vi antaget at beregningen er korrekt. Desuden har vi haft `printf` statements til at se hvor meget enkelte tråde har lagt til summen, men disse er fjernet nu da vi ved at den arbejder med flere tråde uden problemer. Det er dog værd at bemærke, at de sidste få decimaltal er varierernde i forhold til antallet af tråde man benytter, hvilket giver mening idet desto flere opdelinger af tråde desto mindre præcist bliver beregningen når der lægges til.

Da vi har gjort os antagelsen for input er to tal og de kan divideres til et heltal, har vi ikke gjort meget ud af, at tjekke om brugeren giver det rigtige input.

4.2 Linked List

For at teste om vores implementation af `list.c`'s tilføj og fjern funktioner virkede brugte vi den `main.c` fil der blev givet. Den samme fil er nu blevet til testprogrammet for at tjekke om listen kan håndtere flere tråde.

Programmet tager to inputs. Antallet af noder man vil sætte ind i listen og antallet man vil fjerne. Dette giver os muligheden for at tjekke flere scenarier. Vi har testet fire scenarier:

- Tilføj uden at fjerne
- Fjern uden at tilføje
- Tilføj 20 og Fjern 30
- Tilføj 25 og Fjern 10

```
ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode/list$ ./fifo 4 0
Success! List is correct. Diff:4 Length:4

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode/list$ ./fifo 0 10
Success! List is correct. Diff:-10 Length:0

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode/list$ ./fifo 20 30
Success! List is correct. Diff:-10 Length:0

ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode/list$ ./fifo 25 10
Success! List is correct. Diff:15 Length:15
```

Igen som i den tidligere opgave med summen, har vi haft `printf` statements til at angive produkter der kommer ind og ud, men efter vi fik bekræftet at dette lykkedes i forhold til de tre scenarier, fjernede vi dem og lavede en mere ren succes og fejl besked. Scenarierne viser differencen mellem tilføj og fjern og hvad listens længde er til sidst. Rækkefølgen af elementerne er tilfældig da vi ikke har lavet nogen restriktion for det.

4.3 Producer-Consumer

Dette program er testet på baggrund af sine counters. Hvis de begge er lig `PRODUCTS_IN_TOTAL`, betyder det at alle produkter er produceret og konsumeret. Derudover printer den alle produkter der bliver produceret og konsumeret ligesom i opgavebeskrivelsen. Under testing fandt vi dog ud af at der i visse tilfælde, efter det sidste produkt er konsumeret, sidder programmet fast. Dette antager vi skyldes en af semaphorerne har fejlet eller at en af trådene gik tabt og derfor venter programmet.

```
./prodcons 20 20 10 300
.
.
10. Consumed Item 298: P298. Items in buffer 2 (out of 10)
8. Consumed Item 299: P299. Items in buffer 1 (out of 10)
4. Consumed Item 300: P300. Items in buffer 0 (out of 10)
Success! All products produced and consumed.
```

Ved testning af dette program blev vi overrasket over, at man ikke får nogen warnings fra compileren hvis man benytter `wait()` med en semaphore, da vi havde overset at vi på det tidspunkt ikke brugte `sem_wait()` og derfor fik en buffer der blev større end tilladt.

4.4 Banker's Algoritme

Banker's algoritme har vi testet ved brug af tre inputfiler, fil 1 har en sikker start tilstand mens fil 2 og 3 har usikre start tilstande. For at tjekke om `checksafety()` fungerer efter hensigten og finder usikre start tilstande brugte vi inputfil 2 og 3. Input fil 1 er ret ligetil, siden intet er allokeret fra starten vil alle processer være i stand til at køre. Dette kan ses i transriptet forneden, hvordan programmet tjekker alle processer og når den kommer tilbage til process 0 afgør den at programmet er i en sikker tilstand.

```
1  ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode/banker$ ./banker <
       input.txt
2  Number of processes: Number of resources: Resource vector: Enter
       max matrix: Enter allocation matrix:
3  Need matrix:
4  R1 R2 R3
5  3  2  2
6  6  1  3
7  3  1  4
8  4  2  2
9  Availability vector:
10 R1 R2 R3
11 9  3  6
12 Checking safety of process 0
13 Checking safety of process 1
14 Checking safety of process 2
15 Checking safety of process 3
16 Checking safety of process 0
17 Initial state safe!
18 ..
```

Input fil 2 vil resulterer i en usikker tilstand da der på intet tidspunkt kan køre en process, grundet need vil altid være større end available, hvilket programmet også vurdere efter at have simuleret alle processers kørsel.

```
1  ok@ok:~/Dokumenter/BOSC/Assignment 2/Sourcecode/banker$ ./banker <
       input2.txt
2  ok@ok:~/Desktop/git-repos/BOSC/Assignment 2/Sourcecode/banker$
       ./banker < input2.txt
3  Number of processes: Number of resources: Resource vector: Enter
       max matrix: Enter allocation matrix:
4  Need matrix:
5  R1 R2 R3
6  1  2  0
7  1  0  2
8  1  0  3
9  4  2  0
```

```
Availability vector:
R1 R2 R3
0  1  0
Checking safety of process 0
Checking safety of process 1
Checking safety of process 2
Checking safety of process 3
Initial state unsafe!
```

Kørsel af input fil 3 vil også resulterer i en usikker tilstand, men den bekræfter at simuleringen af processerne er som forventet, da process 0 vil i det første tjek vil være sand og fortælle algoritmen at den minimum skal simulerer en runde af alle processerne en gang til. Dette kan ses forneden.

```
ok@ok:~/Desktop/git-repos/BOSC/Assignment 2/Sourcecode/banker$
   ./banker < input3.txt
Number of processes: Number of resources: Resource vector: Enter
   max matrix: Enter allocation matrix:
Need matrix:
R1 R2 R3
0  0  0
2  2  2
2  2  2
3  3  3
Availability vector:
R1 R2 R3
0  0  0
Checking safety of process 0
Checking safety of process 1
Checking safety of process 2
Checking safety of process 3
Checking safety of process 0
Checking safety of process 1
Checking safety of process 2
Checking safety of process 3
Initial state unsafe!
```

For at teste resten af funktionaliteten med hensyn til forespørgsler om allokering af ressourcer eller frigivelse af ressourcer kørte vi programmet med inputfil 1. Vi er dog opmærksomme på at der er nogle problemer med hensyn til den udleveret kode således at der kan forekomme en uendelig løkke grundet en deadlock forudsaget af en starvation, men denne har vi ikke gjort noget ved.

Forneden ses det at process 1 anmoder om ressourcerne 0 0 1, og får dem tildelt. Dette viser at tildeling af ressourcerne går efter hensigten, men det interessante er at se når der kommer forespørgsler der ikke bliver godkendt.

```
..
```

```
Process 1: Requesting resources.
Process 1 request vector: 0 0 1
..
Request leads to safe state. Request Granted!
Vector changed: Availability vector:
R1 R2 R3
9  3  5
..
```

I tilfælde af at en forespørgsel bliver anset som usikker skal den spørge igen efter noget tid som defineret i teorien.

```
..
Process 3 request vector: 1 0 0
..
Request leads to unsafe state. Request Denied!
Process 3 request vector: 1 0 0
..
Request leads to unsafe state. Request Denied!
Process 3 request vector: 1 0 0
..
Request leads to unsafe state. Request Denied!
..
```

Froven ses det at der er en forespørgsel fra process 3 på ressourcerne 1 0 0, som bliver anset som usikker i flere tilfælde. Men i senere tilfælde er det meget sandsynligt at der vil forekomme et scenarie hvor den kan betragtes som sikker og får ressourcerne tildelt, hvilket kan ses i det følgende transcript.

```
Process 3: Requesting resources.
Process 3 request vector: 1 0 0
..
Request leads to safe state. Request Granted!
Vector changed: Availability vector:
R1 R2 R3
4  1  3
..
```

Den sidste funktion i programmet er frigivelsen af ressourcer, hvilket er ret lige til, da den eneste fejl der kan forekomme er en forespørgsel der er større end det der er allokeret for processen, men det er der allerede taget forbehold for og burde ikke fremkomme. Transcriptet forneden viser en release fra process der gennemføres.

```
..
Process 3: Releasing resources.
Process 3 release vector: 2 0 0
```

```
4 Released resources
5 Availability vector:
6 R1 R2 R3
7 5  2  3
8 ..
```

På baggrund af ovenstående transcripts fra programkørsel konkludere vi at algoritmen opføre sig efter teorien.

5 Reflektion

I dette afsnit ser vi retrospekt på programmerne vi har implementeret og hvad vi kan bruge i fremtiden.

Ved at lave alle disse opgaver er vi blevet klogere på hvordan tråde arbejder i et operativ system, samt hvor og hvornår det er relevant at benytte semaphore eller mutexes.

I opgaven med sum valgte vi at gøre brug af en struct til et forholdsvis simpelt program, hvilket som nævnt under Testing, kunne være undgået, da der kun er en variabel der bliver ændret. Hvis det havde været tilfældet at n var anderledes for bestemte processer ville det give mening at beholde structen.

Med hensyn til implementeringen af linked list, brugte vi meget tid og fik mange interessante variationer af en thread safe liste. En af vores lister kaldet `mlist.c`(ikke med i sourcecode), kom vi frem til en løsning som helt selv stod for at holde styr på threads og brugeren skulle kun kalde `list_add()` og `list_remove()` på samme måde som i opgave 2.1. Denne løsning viste sig at være interessant, men meget komplekst og ud fra hvad vi kan forstå af det hele så vides det ikke om den er 100% safe. I sidste ende gik vi dog tilbage til at ændre i `list.c`, da det var mere overskueligt. I fremtiden bør vi nok holde os til de simple løsninger så der kan spares tid når man møder de svære opgaver.

Producer-Consumer opgaven var en god opgave til at lære brugen af semaphores, men grundet en overset `wait()`, der ikke var `sem_wait()` røg der et par timer i debugging.

Fra Banker's algoritme opgave tager vi det med os at forstå den udleveret kode og teorien fuldstændig før vi går i gang med at implementere det, da det var grunden til tiden løb fra os.

6 Konklusion

Efter implementeringen af opgaverne fra opgavebeskrivelsen har vi implementeret fire programmer der giver os indblik i koncepterne omkring pthread, mutex, semaphores og deadlock avoidance. Afprøvelserne af disse programmer har vist den ønskede funktionalitet og at de passer i forhold til teorien. Vi kan hermed konkludere at programmerne virker som beskrevet i opgavebeskrivelsen.

7 Appendix A - Sourcecode

sqrtsum.c

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

typedef struct
{
  int n;
  double sum;
} SQRTSUM;

// globally shared struct
SQRTSUM sqrtsum;

// PThread mutex variable
pthread_mutex_t mutex_sqrtsum;

// structs to determine time
struct timeval tp1, tp2;
struct timezone tpz1, tpz2;

// threads call this function
void *runner(void *param);

/* Takes 2 arguments, a number N for the summation limit and t as
   in number of threads */
int main(int argc, char *argv[])
{
  gettimeofday(&tp1, &tpz1);

  if (atoi(argv[1]) < 0)
  {
    fprintf(stderr, "%d must be over >= 0\n", atoi(argv[1]));
    return -1;
  }

  int i, NUM_THREADS = atoi(argv[2]);
  // the thread id array
  pthread_t tid[NUM_THREADS];
  // set of thread attributes
  pthread_attr_t attr;
  // assumption: argv[1] % NUM_THREADS = 0
```

```
42   int n = atoi(argv[1])/NUM_THREADS;
43   sqrtsum.n = n;
44
45   pthread_mutex_init(&mutex_sqrtsum, NULL);
46
47   // get the default attributes
48   pthread_attr_init(&attr);
49   // set the attribute as joinable, so the threads can join with
        the main thread
50   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
51
52   // create the threads
53   for (i = 0; i < NUM_THREADS; i++)
54   {
55     pthread_create(&tid[i], &attr, runner, (void *) (long) i);
56   }
57
58   // destroy attribute
59   pthread_attr_destroy(&attr);
60
61   // wait for the thread to exit
62   for (i=0; i < NUM_THREADS; i++)
63   {
64     pthread_join(tid[i],NULL);
65   }
66   printf("sqrtsum = %f\n", sqrtsum.sum);
67
68   // destroy mutex
69   pthread_mutex_destroy(&mutex_sqrtsum);
70
71   gettimeofday(&tp2, &tpz2);
72
73   // calculate program time
74   int time = (tp2.tv_sec - tp1.tv_sec) * 1000 + (tp2.tv_usec -
        tp1.tv_usec) / 1000;
75   printf("Total time(ms): %d\n", time);
76   time;
77   return 0;
78 }
79
80 /* Threads will use this function */
81 void *runner(void *param)
82 {
83   int i, n, start, tid, upper;
84
85   // local sum variable
```

```
86   double lsqrtsum = 0.0;
87   // short summation limit
88   n = sqrtsum.n;
89   // thread id
90   tid = (int) (long) param;
91   // start value for loop
92   start = n * tid + 1;
93   // upper value for loop
94   upper = start + n;
95
96   for (i = start; i < upper ; i++)
97   {
98     lsqrtsum += sqrt(i);
99   }
100
101  //printf("Thread %d: Local sqrtsum: %f\n", tid, lsqrtsum);
102
103  // lock mutex
104  pthread_mutex_lock(&mutex_sqrtsum);
105  // update global struct variable
106  sqrtsum.sum += lsqrtsum;
107  // unlock mutex
108  pthread_mutex_unlock(&mutex_sqrtsum);
109  // exit pthread
110  pthread_exit(0);
111 }
```

list/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   /* insert root element which should never be removed */
23   l->first = l->last = (Node *) malloc(sizeof(Node));
24   l->first->elm = NULL;
25   l->first->next = NULL;
26   l->mtx = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
27   pthread_mutex_init(l->mtx, NULL);
28   return l;
29 }
30
31 /* list_add: add node n to list l as the last element */
32 void list_add(List *l, Node *n)
33 {
34   // lock mutex in list
35   pthread_mutex_lock(l->mtx);
36   // check if it's the root
37   if(l->len == 0)
38   {
39     l->first->next = n;
40     l->last = n;
41   }
42   else
43   {
44     l->last->next = n;
```

```
45      l->last = n;
46    }
47    l->len += 1;
48    // unlock mutex in list
49    pthread_mutex_unlock(l->mtx);
50  }
51
52  /* list_remove: remove and return the first (non-root) element
       from list l */
53  Node *list_remove(List *l)
54  {
55    Node *n;
56    // lock mutex in list
57    pthread_mutex_lock(l->mtx);
58    // check if there are (non-root) nodes
59    if(l->len > 0)
60    {
61      n = l->first->next;
62      l->first->next = n->next;
63      l->len -= 1;
64    }
65    // unlock mutex in list
66    pthread_mutex_unlock(l->mtx);
67    return n;
68  }
69
70  /* node_new: return a new node structure */
71  Node *node_new(void)
72  {
73    Node *n;
74    n = (Node *) malloc(sizeof(Node));
75    n->elm = NULL;
76    n->next = NULL;
77    return n;
78  }
79
80  /* node_new_str: return a new node structure, where elm points to
       new copy of s */
81  Node *node_new_str(char *s)
82  {
83    Node *n;
84    n = (Node *) malloc(sizeof(Node));
85    n->elm = (void *) malloc((strlen(s)+1) * sizeof(char));
86    strcpy((char *) n->elm, s);
87    n->next = NULL;
88    return n;
```

```
89 }
```

list/list.h

```
/*****************************************************************************
   list.h

   Header file with definition of a simple linked list.

*****************************************************************************/

#ifndef _LIST_H
#define _LIST_H

/* structures */
typedef struct node {
  void *elm; /* use void type for generality; we cast the
     element's type to void type */
  struct node *next;
} Node;

typedef struct list {
  int len;
  Node *first;
  Node *last;
  pthread_mutex_t *mtx;
} List;

/* functions */
List *list_new(void);            /* return a new list structure */
void list_add(List *l, Node *n); /* add node n to list l as the
   last element */
Node *list_remove(List *l);      /* remove and return the first
   element from list l*/
Node *node_new(void);            /* return a new node structure */
Node *node_new_str(char *s);     /* return a new node structure,
   where elm points to new copy of string s */

#endif
```

list/main.c

```
1  /*****************************************************************************
2     main.c
3
4     Implementation of a simple FIFO buffer as a linked list defined
         in list.h.
5
6  *****************************************************************************
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <pthread.h>
11 #include <sys/time.h>
12 #include "list.h"
13
14 /* mutex */
15 pthread_mutex_t mtx;
16
17 /* global variable */
18 List *fifo;
19
20 /* thread functions */
21 void *thread_add(void *param);
22 void *thread_remove(void *param);
23
24 int main(int argc, char* argv[])
25 {
26   int i;
27   int ADDS = atoi(argv[1]);
28   int REMOVES = atoi(argv[2]);
29
30   // Add threads
31   pthread_t aid[ADDS];
32   // Remove threads
33   pthread_t rid[REMOVES];
34   pthread_attr_t attr;
35
36   // create list
37   fifo = list_new();
38
39   // Initialize and set state for attribute
40   pthread_attr_init(&attr);
41   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
42
43   // create the add threads
```

```
44   for(i = 0; i < ADDS; i++)
45   {
46     pthread_create(&aid[i], &attr, thread_add, (void *) (long) i);
47   }
48
49   for(i = 0; i < ADDS; i++)
50   {
51     pthread_join(aid[i], NULL);
52   }
53
54   // create the remove threads
55   for(i = 0; i < REMOVES; i++)
56   {
57     pthread_create(&rid[i], &attr, thread_remove, NULL);
58   }
59
60   // destroy attributes
61   pthread_attr_destroy(&attr);
62
63   for(i = 0; i < REMOVES; i++)
64   {
65     pthread_join(rid[i], NULL);
66   }
67
68   // check credibility of the list
69   int diff = ADDS - REMOVES;
70   if((diff <= 0) && (fifo->len == 0))
71   {
72     printf("Success! List is correct. Diff:%d Length:%d\n", diff,
         fifo->len);
73   }
74   else if((diff > 0) && (fifo->len == diff))
75   {
76     printf("Success! List is correct. Diff:%d Length:%d\n", diff,
         fifo->len);
77   }
78   else
79   {
80     printf("List is flawed. Diff:%d Length:%d\n", diff, fifo->len);
81   }
82
83   return 0;
84 }
85
86 /* Adds a node to the list */
87 void *thread_add(void *param)
```

```
88 {
89   int id = (int) (long) param;
90
91   char str[10];
92   sprintf(str, "P%d", id);
93   Node *n = node_new_str(str);
94   list_add(fifo, n);
95 }
96
97 /* Removes node from the list */
98 void *thread_remove(void *param)
99 {
100   Node *n = (Node *) malloc(sizeof(Node));
101   n = list_remove(fifo);
102 }
```

prodcons/prodcons.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <semaphore.h>
6  #include <sys/time.h>
7  #include "list.h"
8
9  typedef struct pc {
10   List *l;
11   sem_t full;
12   sem_t empty;
13   sem_t mutex;
14 } PC;
15
16 // product and consume counters
17 int p;
18 int c;
19
20 PC prodcons;
21
22 // arguments
23 int PRODUCERS;
24 int CONSUMERS;
25 int BUFFSIZE;
26 int PRODUCTS_IN_TOTAL;
27
28 // functions
29 void *producer(void *param);
30 void *consumer(void *param);
31 void Sleep(float wait_time_ms);
32
33 /* Producer-Consumer program */
34 int main(int argc, char *argv[])
35 {
36   if(argc != 5)
37   {
38     printf("Not a valid amount of arguments. Arguments: PRODUCERS
          CONSUMERS BUFFERSIZE TOTAL_PRODUCTS\n");
39   }
40
41   // seed random number sequence
42   struct timeval tv;
43   gettimeofday(&tv, NULL);
```

```
44  srand(tv.tv_usec);
45
46  // set arguments
47  PRODUCERS = atoi(argv[1]);
48  CONSUMERS = atoi(argv[2]);
49  BUFFSIZE = atoi(argv[3]);
50  PRODUCTS_IN_TOTAL = atoi(argv[4]);
51
52  int i;
53  pthread_attr_t attr;
54
55  // Initialize semaphores
56  sem_init(&prodcons.full, 0, 0);
57  sem_init(&prodcons.empty, 0, BUFFSIZE);
58  sem_init(&prodcons.mutex, 0, 1);
59
60  p = 0;
61  c = 0;
62
63  // create list
64  prodcons.l = list_new();
65
66  // producer id's
67  pthread_t pid[PRODUCERS];
68
69  // consumer id's
70  pthread_t cid[CONSUMERS];
71
72  // Initialize thread attributes
73  pthread_attr_init(&attr);
74  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
75
76  // create the producers
77  for(i = 0; i < PRODUCERS; i++)
78  {
79    pthread_create(&pid[i], &attr, producer, (void *) (long) i);
80  }
81
82  // create the consumers
83  for(i = 0; i < CONSUMERS; i++)
84  {
85    pthread_create(&cid[i], &attr, consumer, (void *) (long) i);
86  }
87
88  // destroy attribute
89  pthread_attr_destroy(&attr);
```

```
90
91    // wait for the producer threads
92    for(i = 0; i < PRODUCERS; i++)
93    {
94      pthread_join(pid[i], NULL);
95    }
96
97    // wait for the consumer threads
98    for(i = 0; i < CONSUMERS; i++)
99    {
100     pthread_join(cid[i], NULL);
101   }
102
103   // destroy semaphores
104   sem_destroy(&prodcons.empty);
105   sem_destroy(&prodcons.full);
106   sem_destroy(&prodcons.mutex);
107
108   // check if it all worked
109   if(c == p)
110   {
111     printf("Success! All products produced and consumed.\n");
112   }
113   else
114   {
115     printf("All products not consumed/produced.\n");
116   }
117
118   return 0;
119 }
120
121 /* Producer thread function */
122 void *producer(void *param)
123 {
124   int id = (int) (long) param;
125   Node *n;
126
127   while(p < PRODUCTS_IN_TOTAL)
128   {
129     // wait till empty is decremented
130     sem_wait(&prodcons.empty);
131     // wait for mutex
132     sem_wait(&prodcons.mutex);
133     // produce product
134     char str[10];
135     sprintf(str, "P%d", (p+1));
```

```
136     n = node_new_str(str);
137     list_add(prodcons.l, n);
138     p += 1;
139     // release the mutex
140     sem_post(&prodcons.mutex);
141     // notify waiting consumer threads that a space is filled
142     sem_post(&prodcons.full);
143     printf("%d. Produced Item %d: %s. Items in buffer %d (out of
           %d)\n", id, p, (char *)n->elm, prodcons.l->len, BUFFSIZE);
144     fflush(stdout);
145     Sleep(2000);
146   }
147 }
148
149 /* Consumer thread function */
150 void *consumer(void *param)
151 {
152   int id = (int) (long) param;
153   Node *n;
154
155   while(c < PRODUCTS_IN_TOTAL)
156   {
157     // wait till full is incremented
158     sem_wait(&prodcons.full);
159     // wait for mutex
160     sem_wait(&prodcons.mutex);
161     // consume
162     n = list_remove(prodcons.l);
163     c += 1;
164     // release the mutex
165     sem_post(&prodcons.mutex);
166     // notify waiting producer threads that a space is free
167     sem_post(&prodcons.empty);
168     printf("%d. Consumed Item %d: %s. Items in buffer %d (out of
           %d)\n", id, c, (char *)n->elm, prodcons.l->len, BUFFSIZE);
169     fflush(stdout);
170     Sleep(2000);
171   }
172 }
173
174 /* Random sleep function */
175 void Sleep(float wait_time_ms)
176 {
177   wait_time_ms = ((float)rand()) * wait_time_ms / (float)RAND_MAX;
178   usleep((int) (wait_time_ms * 1e3f));
179 }
```

banker/banker.c

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

// print availbility vector
void printstate();

/* 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)
{
  int j, safe = 0, safeAvailable[n], safeNeed[m][n],
     safeAllocation[m][n];

  pthread_mutex_lock(&state_mutex);
  printf("Process %d request vector: ", i);
  for(j = 0; j < n; j++)
  {
    printf("%d ", request[j]);
  }
```

```
44  printf("\n");
45
46  // check if resource request is eligible for allocation
47  for(j = 0; j < n; j++)
48  {
49    if(request[j] <= s->need[i][j])
50    {
51      if(request[j] <= s->available[j])
52      {
53        continue;
54      }
55      else
56      {
57        printf("Not enough available resources(%d) for request(%d)
              try again later...\n", s->available[j], request[j]);
58        pthread_mutex_unlock(&state_mutex);
59        return 0;
60      }
61    }
62    else
63    {
64      printf("Request greater than need!\n");
65      pthread_mutex_unlock(&state_mutex);
66      return 0;
67    }
68  }
69
70  // allocate resources
71  for(j = 0; j < n; j++)
72  {
73    // backup of current resources
74    safeAvailable[j] = s->available[j];
75    safeNeed[i][j] = s->need[i][j];
76    safeAllocation[i][j] = s->allocation[i][j];
77
78    s->available[j] = s->available[j] - request[j];
79    s->allocation[i][j] = s->allocation[i][j] + request[j];
80    s->need[i][j] = s->max[i][j] - s->allocation[i][j];
81  }
82
83  //check state
84  safe = checksafety();
85  if(safe != 1)
86  {
87    for(j = 0; j < n; j++)
88    {
```

```
89        s->available[j] = safeAvailable[j];
90        s->need[i][j] = safeNeed[i][j];
91        s->allocation[i][j] = safeAllocation[i][j];
92      }
93      printf("Request leads to unsafe state. Request Denied!\n");
94      pthread_mutex_unlock(&state_mutex);
95      return 0;
96    }
97    else
98    {
99      printf("Request leads to safe state. Request Granted!\n");
100     printf("Vector changed: ");
101     printstate();
102     pthread_mutex_unlock(&state_mutex);
103     return 1;
104   }
105 }
106
107 /* Release the resources in request for process i */
108 void resource_release(int i, int *request)
109 {
110   int j;
111   pthread_mutex_lock(&state_mutex);
112   printf("Process %d release vector: ", i);
113   for(j = 0; j < n; j++)
114   {
115     printf("%d ", request[j]);
116   }
117   printf("\n");
118
119   for(j = 0; j < n; j++)
120   {
121     // check if release request is less than the allocated
          resources
122     if(request[j] <= s->allocation[i][j])
123     {
124       continue;
125     }
126     else
127     {
128       printf("Release request is more than the allocated!\n");
129       pthread_mutex_unlock(&state_mutex);
130       return;
131     }
132   }
133
```

```
134   for(j = 0; j < n; j++)
135   {
136     // release the resources
137     s->allocation[i][j] = s->allocation[i][j] - request[j];
138     s->need[i][j] = s->max[i][j] - s->allocation[i][j];
139     s->available[j] = s->available[j] + request[j];
140   }
141   printf("Released resources\n");
142   printstate();
143   pthread_mutex_unlock(&state_mutex);
144 }
145
146 /* Generate a request vector */
147 void generate_request(int i, int *request)
148 {
149   int j, sum = 0;
150   while (!sum) {
151     for (j = 0;j < n; j++) {
152       request[j] = s->need[i][j] * ((double)rand())/
              (double)RAND_MAX;
153       sum += request[j];
154     }
155   }
156   printf("\nProcess %d: Requesting resources.\n",i);
157 }
158
159 /* Generate a release vector */
160 void generate_release(int i, int *request)
161 {
162   int j, sum = 0;
163   while (!sum) {
164     for (j = 0;j < n; j++) {
165       request[j] = s->allocation[i][j] * ((double)rand())/
              (double)RAND_MAX;
166       sum += request[j];
167     }
168   }
169   printf("\nProcess %d: Releasing resources.\n",i);
170 }
171
172 int checksafety()
173 {
174   int i, j, work[n], finish[m];
175
176   for(i = 0; i < m; i++)
177   {
```

```
178     finish[i] = 0;
179   }
180   for(i = 0; i < n; i++)
181   {
182     work[i] = s->available[i];
183   }
184
185   i = 0;
186   int dorepeat = 0;
187   while(i < m)
188   {
189     printf("Checking safety of process %d\n", i);
190     if(finish[i] == 0)
191     {
192       // flag for running process
193       int check = 1;
194       for(j = 0; j < n; j++)
195       {
196         if(s->need[i][j] > work[j])
197         {
198           check = 0;
199         }
200       }
201
202       // simulate run of process
203       if(check == 1)
204       {
205         for(j = 0; j < n; j++)
206         {
207           work[j] = work[j] + s->allocation[i][j];
208         }
209         finish[i] = 1;
210         dorepeat = 1;
211       }
212     }
213
214     i++;
215     if(i == m && dorepeat == 1)
216     {
217       i = 0;
218       dorepeat = 0;
219     }
220   }
221
222   for(i = 0; i < m; i++)
223   {
```

```
224     if(finish[i] == 0) return 0;
225   }
226   return 1;
227 
228 }
229 
230 void printstate()
231 {
232   int i, j;
233   printf("Availability vector:\n");
234   for(i = 0; i < n; i++)
235   {
236     printf("R%d ", i+1);
237   }
238   printf("\n");
239 
240   for(j = 0; j < n; j++)
241   {
242     printf("%d  ",s->available[j]);
243   }
244   printf("\n");
245 
246   printf("Allocaton Matrix:\n");
247   for(i = 0; i < m; i++)
248   {
249     for(j = 0; j < n; j++)
250     {
251       printf("%d ", s->allocation[i][j]);
252     }
253     printf("\n");
254   }
255 
256   printf("Need Matrix:\n");
257   for(i = 0; i < m; i++)
258   {
259     for(j = 0; j < n; j++)
260     {
261       printf("%d ", s->need[i][j]);
262     }
263     printf("\n");
264   }
265 }
266 
267 /* Threads starts here */
268 void *process_thread(void *param)
269 {
```

```
  /* Process number */
  int i = (int) (long) param, j;
  /* Allocate request vector */
  int *request = malloc(n*sizeof(int));
  while (1) {
    /* Generate request */
    generate_request(i, request);
    while (!resource_request(i, request)) {
      /* Wait */
      Sleep(100);
    }
    /* Generate release */
    generate_release(i, request);
    /* Release resources */
    resource_release(i, request);
    /* Wait */
    Sleep(1000);
  }
  free(request);
}

int main(int argc, char* argv[])
{
  /* Get size of current state as input */
  int i, j;
  printf("Number of processes: ");
  scanf("%d", &m);
  printf("Number of resources: ");
  scanf("%d", &n);

  /* Allocate memory for state */
  s = (State *)malloc(sizeof(State));
  s->resource = (int *)malloc(sizeof(int) * n);
  s->available = (int *)malloc(sizeof(int) * n);
  s->max = (int **)malloc(sizeof(int *) * n);
  s->allocation = (int **)malloc(sizeof(int *) * n);
  s->need = (int **)malloc(sizeof(int *) * n);

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

  /* Get current state as input */
```

```
316   printf("Resource vector: ");
317   for(i = 0; i < n; i++)
318     scanf("%d", &s->resource[i]);
319   printf("Enter max matrix: ");
320   for(i = 0;i < m; i++)
321     for(j = 0;j < n; j++)
322       scanf("%d", &s->max[i][j]);
323   printf("Enter allocation matrix: ");
324   for(i = 0; i < m; i++)
325     for(j = 0; j < n; j++) {
326       scanf("%d", &s->allocation[i][j]);
327     }
328   printf("\n");
329
330   /* Calcuate the need matrix */
331   for(i = 0; i < m; i++)
332     for(j = 0; j < n; j++)
333       s->need[i][j] = s->max[i][j]-s->allocation[i][j];
334
335   /* Calcuate the availability vector */
336   for(j = 0; j < n; j++) {
337     int sum = 0;
338     for(i = 0; i < m; i++)
339       sum += s->allocation[i][j];
340     s->available[j] = s->resource[j] - sum;
341   }
342
343   /* Output need matrix and availability vector */
344   printf("Need matrix:\n");
345   for(i = 0; i < n; i++)
346     printf("R%d ", i+1);
347   printf("\n");
348   for(i = 0; i < m; i++) {
349     for(j = 0; j < n; j++)
350       printf("%d  ",s->need[i][j]);
351     printf("\n");
352   }
353   printf("Availability vector:\n");
354   for(i = 0; i < n; i++)
355     printf("R%d ", i+1);
356   printf("\n");
357   for(j = 0; j < n; j++)
358     printf("%d  ",s->available[j]);
359   printf("\n");
360
361   /* If initial state is unsafe then terminate with error */
```

```
362   int r = checksafety();
363   if(r != 1)
364   {
365     printf("Initial state unsafe!\n");
366     exit(1);
367   }
368   printf("Initial state safe!\n");
369 
370   /* Seed the random number generator */
371   struct timeval tv;
372   gettimeofday(&tv, NULL);
373   srand(tv.tv_usec);
374 
375   /* Create m threads */
376   pthread_t *tid = malloc(m*sizeof(pthread_t));
377   for (i = 0; i < m; i++)
378     pthread_create(&tid[i], NULL, process_thread, (void *) (long)
          i);
379 
380   /* Wait for threads to finish */
381   pthread_exit(0);
382   free(tid);
383 
384   /* Free state memory */
385   free(s->resource);
386   free(s->available);
387   for(i = 0; i < m; i++)
388   {
389     free(s->max[i]);
390     free(s->allocation[i]);
391     free(s->need[i]);
392   }
393   free(s->max);
394   free(s->allocation);
395   free(s->need);
396   free(s);
397 }
```

Makefile

```
all: sqrtsum
  cd list; make
  cd prodcons; make
  cd banker; make

LIBS = -pthread

sqrtsum: sqrtsum.o
  gcc -o $@ ${LIBS} sqrtsum.o -lm

clean:
  rm -rf *o sqrtsum
  cd list; make clean
  cd prodcons; make clean
  cd banker; make clean
```

list/Makefile

```
all: fifo

FIFO = list.o main.o
LIBS = -pthread

fifo: main.o ${FIFO}
  gcc -o $@ ${LIBS} ${FIFO}

clean:
  rm -rf *o fifo
```

banker/makefile

```
all: banker

LIBS = -pthread

banker: banker.o
  gcc -o $@ ${LIBS} banker.o

clean:
  rm -rf *o banker
```