Obligatorisk Opgave #2
BOSC 2015

Malthe Kirkbro
maek@itu.dk

Mikkel Gaub
mikg@itu.dk

Kenneth Ry Ulrik
kulr@itu.dk

Oktober 22, 2015

Indholdsfortegnelse

1 Multitrådet sum **3**
- 1.1 Sumsqrt . . . 3
- 1.2 Worker . . . 3
- 1.3 Testing . . . 3

2 Multitrådet FIFO buffer som kædet liste **5**
- 2.1 List add og list remove . . . 5
- 2.2 Multitråds-brug af List . . . 5
- 2.3 Mutex locking implementation . . . 6
- 2.4 Testing . . . 6

3 Producer-consumer **7**
- 3.1 Main . . . 7
- 3.2 Produce . . . 7
- 3.3 Consume . . . 8
- 3.4 Testing . . . 8

4 Bankers algoritme **9**
- 4.1 Bankers algoritme . . . 9
- 4.2 Mutex locking . . . 9
- 4.3 "Uendelig" løkke . . . 9
- 4.4 Testing . . . 10

A Opgave 1: sumsqrt.c **12**

B Opgave 1: sumsqrt.h **13**

C Opgave 1: Makefile **14**

D Opgave 2: main.c **14**

E Opgave 2 & 3: list.c **15**

F Opgave 2 & 3: list.h **18**

G Opgave 2: Makefile **19**

H Opgave 3: producerconsumer.c **19**

I Opgave 3: Print output for producerconsumer.c **23**

J Opgave 3: Makefile **25**

K Opgave 4: banker.c **26**

L Opgave 4: Makefile **38**

1 Multitrådet sum

Målet for denne opgave består i at summere kvadratrødder på tværs af flere tråde. Beregningsarbejdet deles således ligeligt mellem trådende og på flerkernede systemer derved også kernerne. Programmet køres ved `sumsqrt [N] [THREADS]` hvor `N` er sumrækkens længde og `THREADS` er antallet af tråde, som arbejdet udføres over. Figur 1 viser udførselshastighedens udvikling når antallet af tråde øges. Programmets implementering beskrives i de følgende afsnit.

1.1 Sumsqrt

Funktionen har til formål at dirigere arbejdet i trådene og danne et samlet resultat. Det givne antal tråde oprettes, og disse gives et strukturelement, `Work`, som beskriver hvilket interval den specifikke tråd skal summere over. Den samlede længde af sumrækken deles ligeligt mellem trådene i disse intervaller. Såfremt sumrækkens længde ikke er delelig med antallet af tråde, gives den sidste tråd resten (uligheden er herved højest én trukket fra antallet af tråde). Dernæst afventes fuldførslen af trådene, og for hver fuldførsel udtrækkes trådens resultet fra `Work`-elementet. Slutligt frigiver funktionen det allokerede hukommelse, som blev anvendt til opbevaring af trådenes id og `Work`-elementerne, og det sammenlagte resultat returneres.

1.2 Worker

Ved oprettelsen af en tråd gives denne funktion som argument, og har således til ansvar at udføre sin del af beregningsarbejdet. Der summeres over intervallet givet af `Work`-strukturelementet, og resultatet skrives efterfølgende til strukturen. Herefter lukkes tråden. Da `Work`-elementets adresse kendes af trådopretteren, kan resultatværdien nu tilgås af denne på hoben.

1.3 Testing

Funktionen er testet med test-frameworket MinUnit, som er en smal implementering af assertions. Funktionen køres med forskellige tråd- og længdeinput, hvorefter resultatet sammenlignes med det forventede.

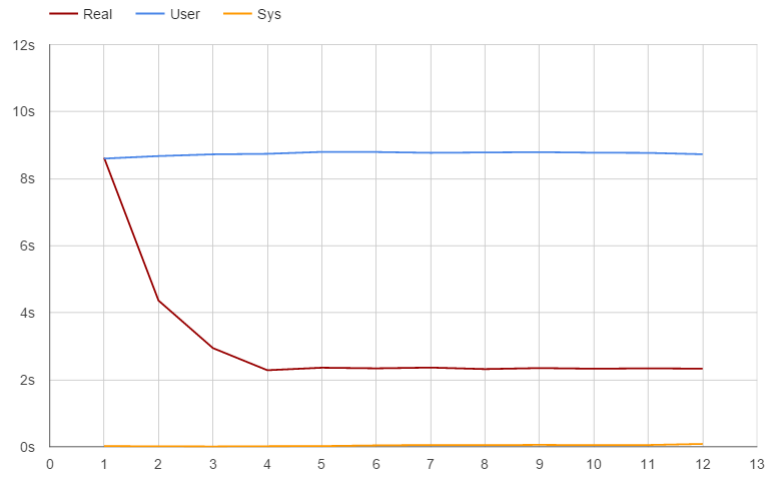

Figure 1: Beregningstid ved kvadratrodssummering, som funktion af trådantal, når sumrækkens længde er 10.000.000.

2 Multitrådet FIFO buffer som kædet liste

Denne opgave har til formål at udvikle en multitrådet FIFO (First In First Out) buffer som en kædet liste, som skal kunne understøtte at flere forsøger at ændrer på den kædede liste på samme tid (concurrency). Dette gøres ved brug af typen `pthread_mutex_t`, vis adresse kan gives som argument til `pthread_mutex_lock` og `pthread_mutex_unlock()` metoderne, som bruges til at begrænse samtidig adgang til kritiske sektioner i koden.

2.1 List add og list remove

FIFO princippet blev fulgt nøje, og der blev bemærket at `len` (length) egenskaben i List struct'en er central for at tilføjelse og fjernelse af Nodes i Listen kan give mening i alle tilfælde. I begge metoder tjekkes, som det første, om der allerede findes elementer i listen, da dette gør forskellen mellem hvordan listen skal behandles.
Note: I den følgende tekst, når der refereres til `first`, menes det første element i Listen, og altså ikke header-Noden, som således abstraheres væk, men som er taget højde for i koden.
I `list_add`, hvis ingen elementer findes i listen på forhånd, sættes den tilføjede Node som både `first` og verb—last— Node. Hvis der findes et eller flere elementer i listen i forvejen, erstattes `last` Noden med den nye, og der skabes et link fra `first` til `last`. I begge tilfælde lægges én til listens `len`.
Når `list_remove` kaldes, skelnes igen mellem om listen er tom eller har et eller flere elementer i sig. Hvis listen er tom, returneres blot `NULL`, da der ikke er noget element at finde i listen. Hvis der derimod findes mindst et element i listen laves en ny skelning; nemlig om `first` Noden har en pointer til en anden Node i sin `next` variabel. Hvis der ingen Node nr. 2 findes, sættes både `first` og `last` til `NULL`, og den udhentede `first` Node returneres. Findes der flere end blot én Node i listen, opdateres `first` til at være Noden som kunne findes i den tidligere `first` Nodes `next` variabel. I begge sidstnævnte tilfælde trækkes én fra listens `len` (Dermed gøres dette naturligvis ikke, hvis listen fra start var tom).

2.2 Multitråds-brug af List

Der kan opstå en række problemer i dette tilfælde.

- Variablen `len` modificeres i alle tilfælde af metoderne `list_add` og `list_remove`. Dette lægger op til race conditions, hvis flere forskellige tråde skulle forsøge at ændre på listen på samme tid.

- Variablerne `first` og `last` er i samme bås som `len`, men med den forskel at der er cases i begge metoder hvor den ene slet ikke berøres, og andre tilfælde hvor der kun læses fra den ene af de to variable. Der er dog stadig brug for en form for begrænset adgang, idet at det kan forekomme

at flere tråde for overskrevet hinandens tilgange til de to variable, hvilket kan resultere i dirty reads eller premature writes.

- Elementet `next` i hver Node i den linkede liste er på samme måde som `first` og `last` udsat for race conditions, da dette element modificeres i alle tilfælde af `list_add` og `list_remove` hvor der i den givne liste findes et eller flere elementer i forvejen.

2.3 Mutex locking implementation

Kun en enkel global variabel med typen `pthread_mutex_t` er blevet defineret for at løse concurrency problemet, og meget simplifistisk låses der blot i starten af `list_add` og `list_remove` og låses op i slutningen af metoderne, hvilket resulterer i, at listen kun kan modificeres i sekventiel tid.
En anden overvejelse var at én `pthread_mutex_t` variabel kunne låse for læsning og opdatering af `len` elementet, og en anden kunne låse for ændring på `first` og `last` elementerne. Denne anden lås ville være generel frem for at låse på `first` og `last` individuelt, da der i begge tilfælde ville modificeres på selve listens struktur. Denne fremgangsmetode, selv om den virkede til at give logisk mening, viste sig at fejle under stress-testning.
En test fil ved navn `test.c` er blevet lavet, for at kunne stress-teste listen til ukendelighed, og det er på denne baggrund at der er fundet frem til, at der tilsyneladende ikke findes anden løsning end at bruge én lås til alting, hvilket til gengæld essentielt set gør, at testen kører sekventielt.

2.4 Testing

Trådsikkerhed i den kædede liste er testet i separat test-fil. Heri undersøges overordnet hvorvidt listens længde er korrekt samt om dens indhold er som forventet. Dette undersøges ved parallel tilføjelse, parallel fjernelse og parallel tilføjelse og fjernelse samtidigt. Da fejl ved implementeringen af trådsikkerhed oftest vil komme til udtryk igennem race conditions, er det svært at garantere fremkomsten af fejl i en enkel test. Derfor defineres et antal tråde, `THREAD_NUM`, og et antal operation, `ACT_COUNT`, i starten af test-filen. Disse afgør henholdsvis antallet af tråde hver test eksekveres med, og antallet af operationer på listen hver tråd foretager. Ved at anvende mange tråde samt et højt operationsantal, som øger chancen for at tråden bliver stoppet af scheduleren inden den eksekverer færdig, forbedres chancen for fejlfinding.

3 Producer-consumer

For at tjekke på input til `producerconsumer.c` tjekkes, om der er givet mindst 3 argumenter samt om de 3 første argumenter kan konverteres til integers som evaluerer til tallet 1 eller højere. Således har vi et positivt antal producers, consumers samt en positiv størrelse på vores buffer. Hvis buffer-størrelsen sættes til 1, svarer det til at der blot bruges en mutex lås.

3.1 Main

`main` metoden initialiserer først listen, mutexes og semaforer (og bruger her et af input-argumenterne til at bestemme størrelsen på `empty` semaforen), behandler input-argumenterne og allokerer plads til det antal tråde der skal genereres (Igen ud fra det givne input).

Efterfølgende påbegyndes trådgenereringen, hvor en for-løkke bruges til at generere det antal af producers og consumers, som input-argumenterne definerer. Alle producer-tråde får givet `*produce()` metoden med som argument, og denne metode agerer som startpunkt for alle producer-trådene. Plads allokeres til en integer-pointer, som skrives til, og sendes med som `void *`, og fortæller produceren om dens unikke ID. Det samme kan siges om consumer-trådene og `*consume` metoden.

Når alle tråde er blevet igangsat, 'joines' der efterfølgende på alle trådene (Ved hjælp af samme iterationsform, som da de blev oprettet), således at det er sikkert at alle tråde er blevet færdige, inden `main` metoden terminerer.

Undervejs i eksekveringen af `*produce` og `*consume` metoderne, printes hver gang success i enten at producere eller konsumere et element i listen opnås.

Både producer- og consumer-tråde kalder `sleep` metoden, som lader tråden vente et tilfældigt tidsmængde i omegnen af den givne parameter. Dette giver idéen af, at konsumering og producering af elementer i listen sker ved tilfældige tidspunkter, men uden at der går alt for lang tid imellem aktioner, guidet vha. parametren til `sleep` funktionen.

3.2 Produce

`produce` metoden starter med at hente sit ID fra `void *` argumentet, og starter derefter en for-løkke, hvor den går i gang med at producere 5 elementer til listen. I for-løkken ventes først på `empty` semaforen (ved tilgang trækkes 1 fra - `empty` semaforen beskriver hvor mange elementer der er plads til i listen, inden `BUFFER_SIZE` er nået), som kan tilgås så længe listen ikke er fyldt. Efterfølgende låses med "`pcmutex`" `pthread_mutex_t` værdien, for at undgå at der tilføjes fra flere producers på samme tid (og at der ikke konsumeres på samme tid, da `consume` metoden bruger samme `pthread_mutex_t` på samme tid).

Når en producer har fået adgang via semaforen og mutexen, findes ID'et til næste element i listen (navngivning til elementet). Fordi der er brugt en mutex

lås, behøves ikke en specific lås til at tilgå og incremente' den globale `ITEM_ID` int variabel.

Som det næste kaldes `list_add` metoden med den globalt definerede liste samt en ny node som modtager et unikt elementnavn som parameter til sin oprettelse. Til sidst printes en success-besked, mutex låsen låses op og `full` semaforen "postes" til (der inkrementeres med 1), hvilket fortæller consumers, at der nu er et element at konsumere.

3.3 Consume

`consume` metoden gør stort set det samme som `produce` metoden, bortset fra at der i stedet ventes på `full` semaforen, postes til `empty` semaforen, og at der kaldes `list_remove` i stedet for `list_add`. Elementnavnet modtages at dette sidstnævnte metodekald, og der udføres også her en successprint.

Disse prints fungerer i kørslen af programmet som kontrol for, at listen ved mange producers og consumers fungerer som den skal, ved at se, at der altid kun trækkes fra eller adderes med 1 på listens nuværende mængde af elementer, samt at Item_**x** kun tilføjes og fjernes én gang.

3.4 Testing

For at teste `producerconsumer.c`, er programmet blevet kørt med argumenterne 10 10 5 (10 producers, 10 consumers og en buffer størrelse på 5), og printsnes rækkefølge og relation til hinanden er blevet analyseret, og er sundt igennem hele kørslen, som kan observeres ved scriptet i appendix I.

4 Bankers algoritme

Bankers algoritme er en metode til allokering af resurser hvor forekomsten af deadlocks umuliggøres. Dette opnås ved at forhindre cykler i resursegrafen, hvor processer kan risikere at vente på hinanden cirkulært. Dette opnås ved at undersøge om en resurseallokering vil bringe systemet i en usikker tilstand før denne udføres. Er dette tilfældet gives processen ikke adgang til resursen og må i stedet forsøge igen på et senere tidspunkt.

4.1 Bankers algoritme

For at undersøge om en tilstand er sikker, er bankers algoritme implementeret i funktionen `state_safe`, som simulerer en fuldstændig kørsel for at finde ud af om alle processer kan få tildelt alle de resurser de maksimum kan få brug for. Hvis de kan det, så er tilstanden sikker. Dette er implementeret ved at der itereres over processernes `need` vektorer og tjekker om der er en process der kan køres. Hvis der ikke findes en process der kan køres er tilstanden usikker. Hvis der er en process der kan køres, så markeres den som færdig, og der itereres igen. Dette fortsætter indtil der enten ikke er nogen processer der kan køres eller at alle processer er kørt.
Denne funktion kaldes i starten af programmet, med starttilstanden, og i funktionen `resource_request`, som er det eneste sted hvor der allokeres resurser og det derfor er relevant at tjekke om den resulterende tilstand er sikker. Ellers skal allokeringen ikke tillades.

4.2 Mutex locking

For at undgå race conditions er der implementeret en mutex lock, `state_mutex`, som låses når en proces tilgår programmets tilstand, `s`, og låses op når den er færdig. Dette er kun nødvendigt i metoderne `resource_request` og `resource_release`, som er de eneste funktioner processerne kører, der piller ved delte værdier. Idet både beskidt læsning og skrivning af data skal forhindres, sættes mutex låsen i starten af begge kald, før der sker noget som helst andet, og slippes ikke før hele metoden er kørt færdig.

4.3 "Uendelig" løkke

Under generationen af en randomiseret `request` vektor, hvilket sker i funktionerne `generate_request` og `generate_release`, fandtes et problem som opstod på grund af det implicitte cast til heltal, som sker når integeren fundet i hhv. need og allocation ganges med den double der genereres af `rand`. Når en double castes til en integer rundes den ned, hvilket gør at sandsynligheden for at få max-værdien er meget lav($1/RAND\_MAX$). Eksempelvis, hvis den fundne værdi i `need` er 1, så er værdien der sættes i `request` vektoren $1 * (rand()/RAND\_MAX)$, hvilket betyder at når `rand` returnerer andet end `RAND_MAX`, så rundes resultatet ned til 0. Dette er særligt et problem hvis en

`need` eller `allocation` vektor består af 0 og 1, da den resulterende `request` langt det meste af tiden vil bestå af nuller. Da dette kode ligger i et while-loop, `while(!sum)`, der tjekker om request vektoren ændres, kommer det til at køre "uendeligt".
Problemet kunne løses med en afrundingsfunktion der følger den almindelige konvention om at der rundes op når tallet er *.5 eller højere og ellers rundes der ned. Så bliver randomiseringen dog ikke uniformt fordelt. Den implementerede løsning, `randint`, undgår problemet ved at der beregnes en `end`, som definerer hvor returværdierne fra `rand` holder op med at bidrage til en uniform fordeling. Alle returvædier der er lig eller er højere end denne værdi, smides væk og der bliver bedt om en ny. Denne løsning er implementeret af Laurence Gonsalves[1].

4.4 Testing

Udover at teste med de givne filer, `input.txt` og `input2.txt`, testes nedenstående sekvens også i filen `banker.c`. Det testes at programmet godkender eller afviser som angivet og at available, need og allocation opdateres som angivet.

Initial state
$Available = \begin{bmatrix} 4 & 2 & 1 \end{bmatrix}$
$Need = \begin{bmatrix} 3 & 2 & 3 \\ 1 & 3 & 0 \\ 3 & 2 & 1 \end{bmatrix}$
$Allocation = \begin{bmatrix} 2 & 2 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 3 \end{bmatrix}$

p0 requesting
$Request = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$
Request denied

p2 requesting
$Request = \begin{bmatrix} 2 & 2 & 1 \end{bmatrix}$
Request approved
$Available = \begin{bmatrix} 2 & 0 & 0 \end{bmatrix}$
$Need = \begin{bmatrix} 3 & 2 & 3 \\ 1 & 3 & 0 \\ 1 & 0 & 0 \end{bmatrix}$
$Allocation = \begin{bmatrix} 2 & 2 & 0 \\ 1 & 0 & 0 \\ 3 & 4 & 4 \end{bmatrix}$

[1] http://stackoverflow.com/questions/822323/how-to-generate-a-random-number-in-c

p1 requesting

$Request = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$

Request approved

$Available = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$

$$Need = \begin{bmatrix} 3 & 2 & 3 \\ 0 & 3 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$Allocation = \begin{bmatrix} 2 & 2 & 0 \\ 2 & 0 & 0 \\ 3 & 4 & 4 \end{bmatrix}$$

p2 releasing

$Request = \begin{bmatrix} 3 & 0 & 0 \end{bmatrix}$

$Available = \begin{bmatrix} 4 & 0 & 0 \end{bmatrix}$

$$Need = \begin{bmatrix} 3 & 2 & 3 \\ 1 & 3 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

$$Allocation = \begin{bmatrix} 2 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 4 & 4 \end{bmatrix}$$

A Opgave 1: sumsqrt.c

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <math.h>

typedef struct work {
        int tid;
        int start;
        int end;
        double result;
} Work;

void *worker(void *data) {
        Work *work = (Work *) data;

        printf("Thread %d started [%d..%d]\n",
                work->tid,
                work->start,
                work->end);

        double sum = 0;
        int i = work->start;
        while (i <= work->end) {
                sum += sqrt(i++);
        }

        printf("Thread %d finished with result %f\n",
                work->tid,
                sum);

        work->result = sum;
        pthread_exit(NULL);
}

double sum_sqrt(int n, int tnum) {
        int i;
        double sum = 0;

        // Array of thread ids
        pthread_t tids[tnum];

        // Array of work structs
        Work works[tnum];
```

```
        i = -1;
        while (++i < tnum) {
                // Setup work data struct for thread
                works[i].tid = i;
                works[i].start = n/tnum*i+1;
                works[i].end = i + 1 == tnum ? n : n/tnum
                    *(i+1);

                pthread_create(&tids[i], NULL, worker, (
                    void *) &works[i]);
        }

        i = -1;
        while (++i < tnum) {
                pthread_join(tids[i], NULL);
                sum += works[i].result;
        }

        return sum;
}

int main(int argc, char* argv[]) {
        int n;
        int tnum;

        if (argc < 3 ||
                !sscanf(argv[1], "%d", &n) ||
                !sscanf(argv[2], "%d", &tnum) ||
                n < tnum ||
                n < 1 ||
                tnum < 1) {
                printf("Invalid arguments\n");
                exit(EXIT_FAILURE);
        }

        printf("Summing for %d using %d thread(s)\n", n,
            tnum);
        printf("Result: %f\n", sum_sqrt(n, tnum));
}
```

B Opgave 1: sumsqrt.h

```
#ifndef SUMSQRT_H
#define SUMSQRT_H
```

```
typedef struct work {
        int tid;
        int start;
        int end;
        double result;
} Work;

double sum_sqrt(int n, int tnum);

#endif
```

C Opgave 1: Makefile

```
CC = gcc -ggdb
LIBS = -lm -pthread

sumsqrt: sumsqrt.o
        ${CC} -o $@ sumsqrt.o ${LIBS}

clean:
        rm -rf *.o sumsqrt
```

D Opgave 2: main.c

```
/*
    ************************************************************************

    main.c

    Implementation of a simple FIFO buffer as a linked
        list defined in list.h.

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

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "list.h"

// FIFO list;
List *fifo;

int main(int argc, char* argv[])
{
```

```
  fifo = list_new();

  list_add(fifo, node_new_str("s1"));
  list_add(fifo, node_new_str("s2"));

  list_remove(fifo);
  list_remove(fifo);
  list_remove(fifo);

  list_add(fifo, node_new_str("s1"));
  list_add(fifo, node_new_str("s2"));

  Node *n1 = list_remove(fifo);
  if (n1 == NULL) { printf("Error no elements in list\n")
      ; exit(-1);}
  Node *n2 = list_remove(fifo);
  if (n2 == NULL) { printf("Error no elements in list\n")
      ; exit(-1);}
  printf("%s\n%s\n", (char *) n1->elm, (char *) n2->elm);

  return 0;
}
```

E Opgave 2 & 3: list.c

```
/*
    ************************************************************************

    list.c

    Implementation of simple linked list defined in list.h
        .

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include "list.h"

pthread_mutex_t mutex;

/* list_new: return a new list structure */
List *list_new(void)
```

```
{
  List *l;

  l = (List *) malloc(sizeof(List));
  l->len = 0;

  /* insert root element which should never be removed */
  l->first = l->last = (Node *) malloc(sizeof(Node));
  l->first->elm = NULL;
  l->first->next = NULL;
  return l;
}

/* list_add: add node n to list l as the last element */
void list_add(List *l, Node *n)
{
  pthread_mutex_lock(&mutex);
  int length = l->len; // Retrieve length

  if (length > 0) // Already contains one or more
      elements
  {
    // Link to next (new) node
    l->last->next = n; // Make previously last element
        link to new last element ("next")
  }
  else // Is empty
  {
    l->first->next = n; // Set first Node to given Node
  }

  l->last = n; // Define as last element
  l->len++; // Update length regardless
  pthread_mutex_unlock(&mutex);
}

/* list_remove: remove and return the first (non-root)
    element from list l */
Node *list_remove(List *l)
{
  pthread_mutex_lock(&mutex);
  int length = l->len; // Retrieve length
  if (length > 0) // Contains one or more elements
  {
    Node *removed = l->first->next; // Get currently
        first Node ('next' of header-Node)
```

```
    Node *newFirst = removed->next; // Get new first Node
    if (newFirst == NULL) // List is being emptied
    {
      l->first->next = NULL;
      l->last = NULL;
    }
    else { // Two or more elements
      l->first->next = newFirst; // Overwrite "first->
          next" (First Node apart from header-Node) to be
          the next element in FIFO list
    }

    // One element removed regardless
    l->len--;

    pthread_mutex_unlock(&mutex);

    // List no longer holds any connection to previously
        first element, return:
    return removed;
  }
  else // List is empty
  {
    pthread_mutex_unlock(&mutex);
    return NULL;
  }

}

/* node_new: return a new node structure */
Node *node_new(void)
{
  Node *n;
  n = (Node *) malloc(sizeof(Node));
  n->elm = NULL;
  n->next = NULL;
  return n;
}

/* node_new_str: return a new node structure, where elm
    points to new copy of s */
Node *node_new_str(char *s)
{
  Node *n;
  n = (Node *) malloc(sizeof(Node));
  n->elm = (void *) malloc((strlen(s)+1) * sizeof(char));
```

```
  strcpy((char *) n->elm, s);
  n->next = NULL;
  return n;
}
```

F Opgave 2 & 3: 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;
} 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
```

G Opgave 2: Makefile

```
CC = gcc -ggdb
LIBS = -pthread

all: fifo test

fifo: main.o list.o
	${CC} -o $@ ${LIBS} list.c main.c

test: test.o list.o
	${CC} -o $@ ${LIBS} list.c test.c;

clean:
	rm -rf *o fifo test
```

H Opgave 3: producerconsumer.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/times.h>
#include "list.h"

int PRODUCTIONS_PER_PRODUCER = 5;
int CONSUMPTIONS_PER_CONSUMER = 5;

// Semaphores
pthread_mutex_t pcmutex;
sem_t empty;
sem_t full;

// Item ID with mutex
pthread_mutex_t mutexItemId;
int ITEM_ID = 0;

// FIFO list
List *fifo;
int BUFFER_SIZE; // FIFO list max size @ production/
    consumption

/* Random sleep function */
void sleep(float wait_time_ms)
```

```
{
        // seed the random number generator
        struct timeval tv;
        gettimeofday(&tv, NULL);
        srand(tv.tv_usec);

        wait_time_ms = ((float)rand())*wait_time_ms / (
            float)RAND_MAX;
        usleep((int) (wait_time_ms * 1e3f)); // convert
            from ms to us
}

void *produce(void *data)
{
        int *producerId = (int *) data;
        int i;
        for (i = 0; i < PRODUCTIONS_PER_PRODUCER; i++) {
                /* Produce the ith item */
                sem_wait(&empty);
                pthread_mutex_lock(&pcmutex);
                        // Generate item name
                        // 5 chars in "Item_"
                        char itemName[5 + 1 + ITEM_ID
                            /10];
                        sprintf(itemName, "Item_%d",
                            ITEM_ID++);
                        // Produce the item
                        list_add(fifo, node_new_str(
                            itemName));
                        // Print
                        printf("Producer %d produced %s. 
                            Items in buffer: %d (Out of %d
                            )\n",
                                *producerId, itemName,
                                    fifo->len, BUFFER_SIZE
                                    );
                pthread_mutex_unlock(&pcmutex);
                sem_post(&full);
                sleep(1000); // Sleep for 1 second on
                    average
        }
        return;
}

void *consume(void *data)
{
```

```
        int *consumerId = (int *) data;
        int i;
        for (i = 0; i < CONSUMPTIONS_PER_CONSUMER; i++) {
                sem_wait(&full);
                pthread_mutex_lock(&pcmutex);
                        // Consume the item
                        char *itemName;
                        itemName = (char *) list_remove(
                            fifo)->elm; // Get element,
                            expect string (void pointer)
                        // Print
                        printf("Consumer %d consumed %s. 
                            Items in buffer: %d (Out of %d
                            )\n",
                                *consumerId, itemName,
                                    fifo->len, BUFFER_SIZE
                                    );
                pthread_mutex_unlock(&pcmutex);
                sem_post(&empty);
                sleep((float)1000); // Sleep for 1 second
                     on average
        }
        return;
}

int main(int argc, char *argv[])
{
        // Validate arguments
        if (argc < 4 ||
                atoi (argv[1]) < 1 || // 0 if
                    unsuccessful parse or negative or 0 if
                     otherwise invalid :)
                atoi (argv[2]) < 1 ||
                atoi (argv[3]) < 1) {

                printf("Invalid arguments. Please provide
                    three positive integers.\n");
                exit(EXIT_FAILURE);
        }

        // Retrieve arguments
        int producerAmount = atoi(argv[1]);
        int consumerAmount = atoi(argv[2]);
        BUFFER_SIZE = atoi(argv[3]);

        // Initialize list, semaphores and mutexes
```

```
fifo = list_new();
//list_add(fifo, node_new_str("lala"));
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);
pthread_mutex_init(&pcmutex, NULL);
pthread_mutex_init(&mutexItemId, NULL);

printf("Program start: %d producers, %d consumers
    and a buffer size of %d...\n",
        producerAmount, consumerAmount,
            BUFFER_SIZE);

// Arrays of thread ids
pthread_t *producer_ids = malloc(producerAmount *
     sizeof(pthread_t));
pthread_t *consumer_ids = malloc(consumerAmount *
     sizeof(pthread_t));

// Attempt to spawn producers and consumers
    somewhat at the same time
int i = 0, j = 0;
while (i < producerAmount || j < consumerAmount)
    // A producer or a consumer can be spawned
{
        if (i < producerAmount) {
                // Spawn producer thread
                int *id = malloc(sizeof(int));
                *id = i;
                pthread_create(&producer_ids[i],
                    NULL, produce, (void *) id);
                i++;
        }
        if (j < consumerAmount) {
                // Spawn consumer thread
                int *id2 = malloc(sizeof(int));
                *id2 = j;
                pthread_create(&consumer_ids[j],
                    NULL, consume, (void *) id2);
                j++;
        }
}
printf("Finished spawning threads.\n");

// Join threads
i = 0; j = 0;
```

```
        while (i < producerAmount || j < consumerAmount)
            // A producer or a consumer can be spawned
        {
                if (i < producerAmount) {
                        // Join producer thread
                        pthread_join(producer_ids[i],
                            NULL);
                        i++;
                }
                if (j < consumerAmount) {
                        // Join consumer thread
                        pthread_join(consumer_ids[j],
                            NULL);
                        j++;
                }
        }
        printf("Finished joining threads.\n");

        free(producer_ids);
        free(consumer_ids);
}
```

I Opgave 3: Print output for producerconsumer.c

```
/BOSC/oo2/opg3$ ./fifo 10 10 5
Program start: 10 producers, 10 consumers and a buffer size of 5...
Producer 0 produced Item_0. Items in buffer: 1 (Out of 5)
Producer 2 produced Item_1. Items in buffer: 2 (Out of 5)
Consumer 0 consumed Item_0. Items in buffer: 1 (Out of 5)
Producer 1 produced Item_2. Items in buffer: 2 (Out of 5)
Finished spawning threads.
Consumer 8 consumed Item_1. Items in buffer: 1 (Out of 5)
Producer 3 produced Item_3. Items in buffer: 2 (Out of 5)
Producer 4 produced Item_4. Items in buffer: 3 (Out of 5)
Consumer 5 consumed Item_2. Items in buffer: 2 (Out of 5)
Producer 7 produced Item_5. Items in buffer: 3 (Out of 5)
Producer 5 produced Item_6. Items in buffer: 4 (Out of 5)
Consumer 4 consumed Item_3. Items in buffer: 3 (Out of 5)
Consumer 6 consumed Item_4. Items in buffer: 2 (Out of 5)
Producer 8 produced Item_7. Items in buffer: 3 (Out of 5)
Producer 9 produced Item_8. Items in buffer: 4 (Out of 5)
Producer 6 produced Item_9. Items in buffer: 5 (Out of 5)
Consumer 7 consumed Item_5. Items in buffer: 4 (Out of 5)
Consumer 1 consumed Item_6. Items in buffer: 3 (Out of 5)
Consumer 3 consumed Item_7. Items in buffer: 2 (Out of 5)
Consumer 2 consumed Item_8. Items in buffer: 1 (Out of 5)
```

Consumer 9 consumed Item_9. Items in buffer: 0 (Out of 5)
Producer 0 produced Item_10. Items in buffer: 1 (Out of 5)
Producer 6 produced Item_11. Items in buffer: 2 (Out of 5)
Producer 2 produced Item_12. Items in buffer: 3 (Out of 5)
Consumer 8 consumed Item_10. Items in buffer: 2 (Out of 5)
Producer 7 produced Item_13. Items in buffer: 3 (Out of 5)
Consumer 1 consumed Item_11. Items in buffer: 2 (Out of 5)
Producer 5 produced Item_14. Items in buffer: 3 (Out of 5)
Producer 4 produced Item_15. Items in buffer: 4 (Out of 5)
Producer 3 produced Item_16. Items in buffer: 5 (Out of 5)
Consumer 6 consumed Item_12. Items in buffer: 4 (Out of 5)
Producer 8 produced Item_17. Items in buffer: 5 (Out of 5)
Consumer 7 consumed Item_13. Items in buffer: 4 (Out of 5)
Consumer 7 consumed Item_14. Items in buffer: 3 (Out of 5)
Consumer 0 consumed Item_15. Items in buffer: 2 (Out of 5)
Producer 0 produced Item_18. Items in buffer: 3 (Out of 5)
Producer 4 produced Item_19. Items in buffer: 4 (Out of 5)
Producer 8 produced Item_20. Items in buffer: 5 (Out of 5)
Consumer 0 consumed Item_16. Items in buffer: 4 (Out of 5)
Producer 4 produced Item_21. Items in buffer: 5 (Out of 5)
Consumer 8 consumed Item_17. Items in buffer: 4 (Out of 5)
Producer 8 produced Item_22. Items in buffer: 5 (Out of 5)
Consumer 9 consumed Item_18. Items in buffer: 4 (Out of 5)
Consumer 7 consumed Item_19. Items in buffer: 3 (Out of 5)
Consumer 3 consumed Item_20. Items in buffer: 2 (Out of 5)
Producer 0 produced Item_23. Items in buffer: 3 (Out of 5)
Producer 5 produced Item_24. Items in buffer: 4 (Out of 5)
Producer 9 produced Item_25. Items in buffer: 5 (Out of 5)
Consumer 5 consumed Item_21. Items in buffer: 4 (Out of 5)
Producer 8 produced Item_26. Items in buffer: 5 (Out of 5)
Consumer 1 consumed Item_22. Items in buffer: 4 (Out of 5)
Producer 3 produced Item_27. Items in buffer: 5 (Out of 5)
Consumer 4 consumed Item_23. Items in buffer: 4 (Out of 5)
Producer 7 produced Item_28. Items in buffer: 5 (Out of 5)
Consumer 2 consumed Item_24. Items in buffer: 4 (Out of 5)
Producer 1 produced Item_29. Items in buffer: 5 (Out of 5)
Consumer 3 consumed Item_25. Items in buffer: 4 (Out of 5)
Producer 1 produced Item_30. Items in buffer: 5 (Out of 5)
Consumer 9 consumed Item_26. Items in buffer: 4 (Out of 5)
Producer 5 produced Item_31. Items in buffer: 5 (Out of 5)
Consumer 8 consumed Item_27. Items in buffer: 4 (Out of 5)
Producer 4 produced Item_32. Items in buffer: 5 (Out of 5)
Consumer 8 consumed Item_28. Items in buffer: 4 (Out of 5)
Producer 6 produced Item_33. Items in buffer: 5 (Out of 5)
Consumer 4 consumed Item_29. Items in buffer: 4 (Out of 5)
Producer 9 produced Item_34. Items in buffer: 5 (Out of 5)

```
Consumer 1 consumed Item_30. Items in buffer: 4 (Out of 5)
Producer 7 produced Item_35. Items in buffer: 5 (Out of 5)
Consumer 3 consumed Item_31. Items in buffer: 4 (Out of 5)
Producer 2 produced Item_36. Items in buffer: 5 (Out of 5)
Consumer 7 consumed Item_32. Items in buffer: 4 (Out of 5)
Producer 3 produced Item_37. Items in buffer: 5 (Out of 5)
Consumer 6 consumed Item_33. Items in buffer: 4 (Out of 5)
Producer 1 produced Item_38. Items in buffer: 5 (Out of 5)
Consumer 1 consumed Item_34. Items in buffer: 4 (Out of 5)
Producer 0 produced Item_39. Items in buffer: 5 (Out of 5)
Consumer 0 consumed Item_35. Items in buffer: 4 (Out of 5)
Producer 5 produced Item_40. Items in buffer: 5 (Out of 5)
Consumer 9 consumed Item_36. Items in buffer: 4 (Out of 5)
Consumer 5 consumed Item_37. Items in buffer: 3 (Out of 5)
Consumer 2 consumed Item_38. Items in buffer: 2 (Out of 5)
Producer 2 produced Item_41. Items in buffer: 3 (Out of 5)
Producer 1 produced Item_42. Items in buffer: 4 (Out of 5)
Producer 6 produced Item_43. Items in buffer: 5 (Out of 5)
Consumer 4 consumed Item_39. Items in buffer: 4 (Out of 5)
Producer 7 produced Item_44. Items in buffer: 5 (Out of 5)
Consumer 5 consumed Item_40. Items in buffer: 4 (Out of 5)
Producer 2 produced Item_45. Items in buffer: 5 (Out of 5)
Consumer 3 consumed Item_41. Items in buffer: 4 (Out of 5)
Producer 9 produced Item_46. Items in buffer: 5 (Out of 5)
Consumer 4 consumed Item_42. Items in buffer: 4 (Out of 5)
Producer 9 produced Item_47. Items in buffer: 5 (Out of 5)
Consumer 6 consumed Item_43. Items in buffer: 4 (Out of 5)
Producer 3 produced Item_48. Items in buffer: 5 (Out of 5)
Consumer 6 consumed Item_44. Items in buffer: 4 (Out of 5)
Consumer 5 consumed Item_45. Items in buffer: 3 (Out of 5)
Consumer 2 consumed Item_46. Items in buffer: 2 (Out of 5)
Consumer 0 consumed Item_47. Items in buffer: 1 (Out of 5)
Consumer 9 consumed Item_48. Items in buffer: 0 (Out of 5)
Producer 6 produced Item_49. Items in buffer: 1 (Out of 5)
Consumer 2 consumed Item_49. Items in buffer: 0 (Out of 5)
Finished joining threads.
```

J Opgave 3: Makefile

```
CC = gcc -ggdb -g -O0
OBJS = producerconsumer.o list.o
LIBS = -pthread

fifo: ${OBJS}
```

```
	${CC} -o $@ ${OBJS} ${LIBS}

clean:
	rm -rf *.o fifo
```

K Opgave 4: banker.c

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

static void print_res(int *res) {
  int i;

  printf("[INFO] ");
  for (i = 0; i < n; ++i) {
    printf("R%d ", i);
  }

  printf("\n[INFO] ");
  for (i = 0; i < n; ++i) {
    printf("%d  ", i);
  }

  printf("\n\n");
}

static int **matrix_alloc() {
```

```
  int i, **matrix = malloc(m * sizeof(int *));
  for (i = 0; i < m; ++i) {
    matrix[i] = malloc(n * sizeof(int));
  }

  return matrix;
}

static void matrix_free(int **matrix) {
  int i;
  for (i = 0; i < m; ++i) {
    free(matrix[i]);
  }
  free(matrix);
}

static State *state_alloc() {
  State *state = malloc(sizeof(State));

  state->resource   = malloc(n * sizeof(int));
  state->available  = malloc(n * sizeof(int));
  state->max        = matrix_alloc();
  state->allocation = matrix_alloc();
  state->need       = matrix_alloc();

  return state;
}

static void state_free(State *state) {
  free(state->resource);
  free(state->available);
  matrix_free(state->max);
  matrix_free(state->allocation);
  matrix_free(state->need);
}

static State *state_cpy(State *state) {
  State *cpy = state_alloc();
  int i, j;

  for (i = 0; i < n; ++i) {
    cpy->resource[i]  = state->resource[i];
    cpy->available[i] = state->available[i];
  }

  for (i = 0; i < m; ++i)
```

```
    for (j = 0; j < n; ++j) {
      cpy->max[i][j]        = state->max[i][j];
      cpy->allocation[i][j] = state->allocation[i][j];
      cpy->need[i][j]       = state->need[i][j];
    }

  return cpy;
}

static bool vec_lte(int *v1, int *v2) {
  bool lte = true;
  int i;
  for (i = 0; i < n; ++i)
    if (v1[i] > v2[i]) {
      lte = false;
      break;
    }

  return lte;
}

static int *vec_add_ass(int *v1, int *v2) {
  int i;
  for (i = 0; i < n; ++i)
    v1[i] += v2[i];

  return v1;
}

static int *vec_sub_ass(int *v1, int *v2) {
  int i;
  for (i = 0; i < n; ++i)
    v1[i] -= v2[i];

  return v1;
}

static bool state_safe(State *state) {
  int i;

  int work[n];
  bool finish[n];
  for (i = 0; i < n; ++i) {
    work[i] = state->available[i];
    finish[i] = false;
  }
```

```
  bool progress;
  do {
    progress = false;

    for (i = 0; i < m; ++i) {
      if (finish[i]) continue;

      if (vec_lte(state->need[i], work)) {
        vec_add_ass(work, state->allocation[i]);
        finish[i] = progress = true;
      }
    }
  } while (progress);

  for (i = 0; i < n; ++i)
    if (!finish[i]) return false;

  return true;
}

static bool request_safe(int *req, int req_i, State *
    state) {
  if (!vec_lte(req, state->need[req_i])) {
    printf("[ERROR] P%d exceeded maximum claim", req_i);
    return false;
  }

  if (!vec_lte(req, state->available)) {
    return false;
  }

  State *cpy = state_cpy(state);
  vec_sub_ass(cpy->available, req);
  vec_add_ass(cpy->allocation[req_i], req);
  vec_sub_ass(cpy->need[req_i], req);

  bool safe = state_safe(cpy);
  state_free(cpy);
  return safe;
}

/* 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);
  printf("[INFO] Processing request from P%d (", i);
  int j;
  for (j = 0; j < n-1; ++j)
  {
    printf("%d, ", request[j]);
  }
  printf("%d)\n", request[n-1]);

  if (request_safe(request, i, s)) {
    vec_sub_ass(s->available, request);
    vec_add_ass(s->allocation[i], request);
    vec_sub_ass(s->need[i], request);

    printf("[INFO] Request from P%d accepted\n", i);
    printf("[INFO] Availability changed:\n");
    print_res(s->available);
    pthread_mutex_unlock(&state_mutex);
    return 1;
  }

  printf("[INFO] Request from P%d denied\n", i);
  pthread_mutex_unlock(&state_mutex);
  return 0;
}

/* Release the resources in request for process i */
void resource_release(int i, int *request)
{
  pthread_mutex_lock(&state_mutex);

  vec_add_ass(s->available, request);
  vec_sub_ass(s->allocation[i], request);
  vec_add_ass(s->need[i], request);
```

```
  printf("[INFO] P%d released resources (", i);
  int j;
  for (j = 0; j < n-1; ++j)
  {
    printf("%d, ", request[j]);
  }
  printf("%d)\n", request[n-1]);

  printf("[INFO] Availability changed:\n");
  print_res(s->available);
  pthread_mutex_unlock(&state_mutex);
}

// http://stackoverflow.com/questions/822323/how-to-
    generate-a-random-number-in-c
int randint(int n) {
  if (!n) return 0;
  n++;

  if ((n - 1) == RAND_MAX) {
    return rand();
  } else {
    // Chop off all of the values that would cause skew
        ...
    long end = RAND_MAX / n; // truncate skew
    end *= n;

    // ... and ignore results from rand() that fall above
         that limit.
    // (Worst case the loop condition should succeed 50%
        of the time,
    // so we can expect to bail out of this loop pretty
        quickly.)
    int r;
    while ((r = rand()) >= end);

    return r % n;
  }
}

/* Generate a request vector */
void generate_request(int i, int *request)
{
  int j, sum = 0;
  while (!sum) {
    for (j = 0;j < n; j++) {
```

```
      request[j] = randint(s->need[i][j]);
      sum += request[j];
    }
  }
  printf("Process %d: Requesting resources.\n",i);
}

/* Generate a release vector */
void generate_release(int i, int *request)
{
  int j, sum = 0;
  while (!sum) {
    for (j = 0;j < n; j++) {
      request[j] = randint(s->allocation[i][j]);
      sum += request[j];
    }
  }
  printf("Process %d: Releasing resources.\n",i);
}

/* Threads starts here */
void *process_thread(void *param)
{
  /* 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);
}

static bool test_matrix_eq(int mat1[n][m], int **mat2) {
  int i, j;
```

```
  for(i = 0; i < m; i++)
    for(j = 0; j < n; j++)
      if (mat1[i][j] != mat2[i][j]) return false;

  return true;
}

static char *test() {
  m = n = 3;
  s = state_alloc();

  // Set inital state
  s->available[0] = 4;
  s->available[1] = 2;
  s->available[2] = 1;


  s->need[0][0] = 3;
  s->need[0][1] = 2;
  s->need[0][2] = 3;

  s->need[1][0] = 1;
  s->need[1][1] = 3;
  s->need[1][2] = 0;

  s->need[2][0] = 3;
  s->need[2][1] = 2;
  s->need[2][2] = 1;


  s->allocation[0][0] = 2;
  s->allocation[0][1] = 2;
  s->allocation[0][2] = 0;

  s->allocation[1][0] = 1;
  s->allocation[1][1] = 0;
  s->allocation[1][2] = 0;

  s->allocation[2][0] = 1;
  s->allocation[2][1] = 2;
  s->allocation[2][2] = 3;


  // Reqest invalid vector
  int req0[3] = { 0, 0, 1 };
  mu_assert(
```

```
  "Request was not denied",
  !resource_request(0, req0));


// Request valid vector
int req2[3] = { 2, 2, 1 };
mu_assert(
  "Request was not accepted",
  resource_request(2, req2));

// Expected resulting state
int req2_available[3] = { 2, 0, 0 };
int req2_need[3][3] = {
  { 3, 2, 3 },
  { 1, 3, 0 },
  { 1, 0, 0}
};
int req2_allocation[3][3] = {
  { 2, 2, 0 },
  { 1, 0, 0 },
  { 3, 4, 4 }
};

// Assert resulting state
mu_assert(
  "Available vector invalid",
  0 == memcmp(req2_available, s->available, n));

mu_assert(
  "Need matrix invalid",
  test_matrix_eq(req2_need, s->need));

mu_assert(
  "Allocation matrix invalid",
  test_matrix_eq(req2_allocation, s->allocation));


// Request valid vector
int req1[3] = { 1, 0, 0 };
mu_assert(
  "Request was not accepted",
  resource_request(1, req1));

// Expected resulting state
int req1_available[3] = { 1, 0, 0 };
int req1_need[3][3] = {
```

```
  { 3, 2, 3 },
  { 0, 3, 0 },
  { 1, 0, 0}
};
int req1_allocation[3][3] = {
  { 2, 2, 0 },
  { 2, 0, 0 },
  { 3, 4, 4 }
};

// Assert resulting state
mu_assert(
  "Available vector invalid",
  0 == memcmp(req1_available, s->available, n));

mu_assert(
  "Need matrix invalid",
  test_matrix_eq(req1_need, s->need));

mu_assert(
  "Allocation matrix invalid",
  test_matrix_eq(req1_allocation, s->allocation));


// Release vector
int rel2[3] = { 3, 0, 0 };
resource_release(2, rel2);

// Expected resulting state
int rel2_available[3] = { 4, 0, 0 };
int rel2_need[3][3] = {
  { 3, 2, 3 },
  { 0, 3, 0 },
  { 4, 0, 0}
};
int rel2_allocation[3][3] = {
  { 2, 2, 0 },
  { 2, 0, 0 },
  { 0, 4, 4 }
};

// Assert resulting state
mu_assert(
  "Available vector invalid",
  0 == memcmp(rel2_available, s->available, n));
```

```
  mu_assert(
    "Need matrix invalid",
    test_matrix_eq(rel2_need, s->need));

  mu_assert(
    "Allocation matrix invalid",
    test_matrix_eq(rel2_allocation, s->allocation));


  state_free(s);
  return 0;
}

int main(int argc, char* argv[])
{
  if (argc >= 2 && strcmp(argv[1], "test") == 0) {
    char *result = test();
    if (result != 0) {
      printf("[TEST] %s\n", result);
      printf("[TEST] Test failure\n");
    }
    else {
      printf("[TEST] All tests passed\n");
    }

    return result != 0;
  }

  /* 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_alloc();

  /* Get current state as input */
  printf("Resource vector: ");
  for(i = 0; i < n; i++)
    scanf("%d", &s->resource[i]);
  printf("Enter max matrix: ");
  for(i = 0;i < m; i++)
    for(j = 0;j < n; j++)
      scanf("%d", &s->max[i][j]);
```

```
printf("Enter allocation matrix: ");
for(i = 0; i < m; i++)
  for(j = 0; j < n; j++) {
    scanf("%d", &s->allocation[i][j]);
  }
printf("\n");

/* Calcuate the need matrix */
for(i = 0; i < m; i++)
  for(j = 0; j < n; j++)
    s->need[i][j] = s->max[i][j]-s->allocation[i][j];

/* Calcuate the availability vector */
for(j = 0; j < n; j++) {
  int sum = 0;
  for(i = 0; i < m; i++)
    sum += s->allocation[i][j];
  s->available[j] = s->resource[j] - sum;
}

/* Output need matrix and availability vector */
printf("Need matrix:\n");
for(i = 0; i < n; i++)
  printf("R%d ", i+1);
printf("\n");
for(i = 0; i < m; i++) {
  for(j = 0; j < n; j++)
    printf("%d  ",s->need[i][j]);
  printf("\n");
}
printf("Availability vector:\n");
for(i = 0; i < n; i++)
  printf("R%d ", i+1);
printf("\n");
for(j = 0; j < n; j++)
  printf("%d  ",s->available[j]);
printf("\n");

/* If initial state is unsafe then terminate with error
    */
if (!state_safe(s)) {
  printf("[ERROR] Initial state unsafe\n");
  exit(EXIT_FAILURE);
}

/* Seed the random number generator */
```

```
  struct timeval tv;
  gettimeofday(&tv, NULL);
  srand(tv.tv_usec);

  /* Create m threads */
  pthread_t *tid = malloc(m*sizeof(pthread_t));
  for (i = 0; i < m; i++)
    pthread_create(&tid[i], NULL, process_thread, (void
        *) (long) i);

  /* Wait for threads to finish */
  pthread_exit(0);
  free(tid);

  /* Free state memory */
  state_free(s);
}
```

L Opgave 4: Makefile

```
CC = gcc -ggdb -pthread

banker: banker.o
        ${CC} -o $@ banker.c

clean:
        rm -rf *.o banker
```