IT-Universitetet i København 22. september 2014

Operativsystemer og C

Obligatorisk opgave 2

Rapporten - som **pdf-fil** - samt kildekode skal pakkes og uploades til learnit senest torsdag den 10. oktober, klokken 10:00

Denne obligatoriske opgave består af flere separate del-opgaver. Alle opgaverne skal afleveres og indgår som del af rapporten.

Baggrund

Målet med denne obligatoriske opgave er at forstå hvordan POSIX tråde, semaforer og mutex låse fungerer, samt hvordan disse implementeres i C på en Linux platform.

Historisk set, har det været hardwareproducenten som implementerede sine egne trådbiblioteker hvilket gjorde det svært at portere flertrådede programmer mellem forskellige platforme. For familien af UNIX operativsystemer, herunder Linux, findes der dog et stadardiseret portabelt trådbibliotek specificeret ved den såkaldte IEEE POSIX 1003.1c standard. Trådbiblioteker, som anvender denne standard, kaldes POSIX tråde (eller Pthreads) og er i C defineret i header filen `<pthread.h>`.

POSIX trådbiblioteket tilbyder funktioner for `pthread_t` trådobjekter. Fx til at skabe, samle og terminere findes følgende funktioner:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
int pthread_join(pthread_t thread, void **value_ptr);
void pthread_exit(void *value_ptr);
```

POSIX trådbiblioteket tilbyder også funktioner til synkronisering af tråde, så som mutex låse og betingelsesvariabler:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_destroy(pthread_cond_t *cond);
```

Semaforer er også en del af POSIX 1003.1 standarden men defineres for sig i header filen `<semaphores.h>`. Semaforer er objekter af typen `sem_t` og anvendes fx gennem:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

Du kan bruge `man` til at få flere detaljer om alle de ovenstående funktioner.

Opgave 1. Multitrådet sum ($1\frac{1}{2}$ time)

Målet med denne opgave er, at du kan

- anvende tråde til at samarbejde om beregninger på en multicore processorer.

I Silberschatz på side 161 i 8. udgave eller side 171 i 9. udgave vises koden til et C program, der bruger Pthreads API til at starte en tråd som beregner sum-funktionen:

$$sum = \sum_{i=0}^{N} i.$$

Dette program illustrerer anvendelsen af Pthreads men har derudover ingen mærkbar effekt for brugeren af programmet.

1.1) Omskriv programmet sådan at det rent faktisk kører hurtigere (på en multicore maskine) end det ellers ville have gjort uden brug af tråde. Du skal dog gøre det for en funktion der (i stedet for *sum*) beregner summen af kvadratrødder:

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

Denne funktion tager lidt længere tid at beregne og kan gå til højere N uden overflow.

1.2) Du skal også måle udførselstiden af dit program ved forskellige antal tråde (f.eks. ved at bruge `time`) og lave en tilhørende speed-up graf. Brug f.eks. $N = 10000000$, der er tilpas stort til at beregningen tager ca. 1-5 sek. med én tråd.

Hints:

- Du skal starte flere tråde (1-2-4-8 etc.) til at samarbejde om beregningsarbejdet.
- Du skal desuden ændre fra at lave summen for integers (typen `int`) til at lave dem i flydende tal (typen `double`). Dette er nødvendigt da $\sqrt{i}$ ikke er et heltal.
- For at kunne bruge `sqrt()` skal du inkludere `<math.h>` samt linke med `-lm`.

- Du skal lave en `struct` med parametre til trådene, der fortæller hvilket arbejde tråden skal lave.
- Du kan med fordel antage, at N divideret med antallet af tråde er et heltal.
- Husk at sikre dig at din CPU har flere cores (`cat /proc/cpuinfo`). Du kan med fordel logge ind på `ssh.itu.dk` når du tester.

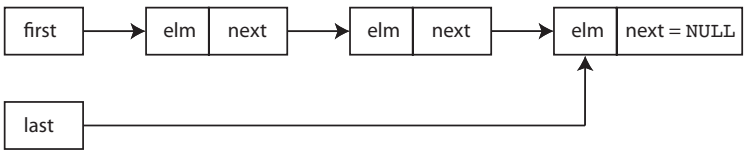
Opgave 2. Multitrådet FIFO buffer som kædet liste (2 timer)

Målet med denne opgave er, at du kan

- implementere en kædet liste i C.
- trådsikre dine programmer så de kan anvedes af flere tråde i parallel.

I denne opgave skal du implementere en FIFO buffer som en kædet liste i C til flertrådede programmer.

Kig på koden i de uploadede filer `list.c` og `list.h`, der viser en ufuldstændig udgave af en kædet liste i C uden brug af tråde.



2.1) Du skal starte med at færdiggøre den ufuldstændige 1-tråds udgave af en kædet liste i `list.c`, sådan at den kan anvendes som FIFO buffer. Bemærk, at bufferens indhold er af typen `void` for at den kan indeholde alle objekttyper (fx strenge).

Dette kræver, at du implementerer de to funktioner:

```
void list_add(List *l, Node *n);

Node *list_remove(List *l);
```

2.2) Antag nu, at vi anvender implementationen af en kædet liste i et program hvor flere tråde tilgår den samme liste med funktionerne implementeret i `list.c`.

Beskriv de problemer, der kan opstå.

2.3) Brug én eller flere mutex låse til at lave en version af `list.c`, der kan bruges i flertrådede programmer, samt et testprogram til at sikre at den virker.

Hints:

- Se kommentarerne i kildekoden for mere detaljerede beskrivelser.
- Bemærk, at der i listen altid indsættes et første element (her kaldes det `root`), som ikke må fjernes. Dvs. at `first` pointeren peger på dette første element og `first->next` peger på det første rigtige element indsat i listen (eller `NULL` hvis der ikke er noget). Når du fjerner et element skal du huske at overholde dette.

Opgave 3. Producer-Consumer med bounded buffer (3 timer)

Målet med denne opgave er, at du kan

- implementere Producer-Consumer problemet med Pthreads i C.
- anvende tælle-semaforer.

3.1) Implementér ved brug af POSIX tråde et producer-consumer system, som beskrevet i Silberschatz afsnit 6.6.1 i 8. udg. eller 6.7.1 i 9.. Programmet skal anvende en kædet liste fra forrige opgave til at realisere den delte buffer. Elementerne i bufferen kan for eksempel være tekststrenge. Som beskrevet i bogen skal bufferen have begrænset kapacitet, og man skal bruge to tælle-semaforer (`empty` og `full`) for at holde styr på hvor mange pladser der er frie, og hvor mange pladser der er i brug.

Når bufferen er tom skal consumer-trådene sættes til at vente indtil der kommer data i bufferen, og når den er fuld skal producer-trådene vente indtil der er blevet plads i bufferen til de nye data.

Dit producer-consumer program skal som minimum kunne følgende:

- Input (fx fra kommandolinjen) skal være antallet af producers, antallet af consumers, størrelsen af den delte buffer, samt antallet af produkter der ialt skal produceres.
- Hver producer eller consumer skal køre i sin egen tråd - der må kun være én producer-funktion og én consumer-funktion.
- Hver gang et produkt produceres eller konsumeres, skal tråden „sove“ i et tidsinterval med tilfældig længde. Til dette anvendes en tilfældighedsgenerator og funktionen `sleep()` fra `<sys/time.h>`.
- Programmet må ikke gå i baglås eller risikere at tråde udsultes.
- Output skal være information fra producers og consumers hver gang de producerer eller konsumerer et produkt samt hvor mange produkter der er i bufferen på dette tidspunkt. Fx noget i stil med:

```
[hbrs@cypher opg3]$ ./prodcons 6 6 10
Producer 5 produced Item_0. Items in buffer: 1 (out of 10).
Producer 2 produced Item_1. Items in buffer: 2 (out of 10).
Consumer 0 consumed Item_0. Items in buffer: 1 (out of 10).
Producer 0 produced Item_2. Items in buffer: 2 (out of 10).
Producer 1 produced Item_3. Items in buffer: 3 (out of 10).
Producer 3 produced Item_4. Items in buffer: 4 (out of 10).
Producer 0 produced Item_5. Items in buffer: 5 (out of 10).
Consumer 1 consumed Item_1. Items in buffer: 4 (out of 10).
Producer 2 produced Item_6. Items in buffer: 5 (out of 10).
```

```
Producer 4 produced Item_7. Items in buffer: 6 (out of 10).
Producer 5 produced Item_8. Items in buffer: 7 (out of 10).
Consumer 3 consumed Item_2. Items in buffer: 6 (out of 10).
Consumer 1 consumed Item_3. Items in buffer: 5 (out of 10).
Consumer 2 consumed Item_4. Items in buffer: 4 (out of 10).
...
```

- Sørg for at programmet afslutter når alle produkter er produceret og konsumeret (dette kan med fordel implementeres til sidst).

Hints:

- Et opgave der ligner - dog med en anderledes implementeret bufffer - er Silberschatz s. 274 i 8. udg. programmerings-projekt 6.37 eller s. 303 projekt 3 i 9. udg..
- En simpel måde til at få en tråd til at sove i et tilfældigt tidsrum kan skrives:

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

Dvs. at `Sleep(1000)` vil sove i gennemsnit 1 sekund.

Da `rand()` er en pseudo-tilfældighedsgenerator, skal den have et skiftende seed:

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

(ellers giver den de samme „tilfældige“ tal hver gang programmet køres).

Opgave 4. Banker's algorithm til håndtering af deadlock. (2 timer)

Opgaven er baseret på Silberschatz s. 339 Banker's Algorithm Projekt med pthreads.

Der uploadet en ufuldstændig programkode `banker.c`, som kan bruges som udgangspunkt. Koden er baseret på tilstandsstrukturen (gennemgået ved forelæsningen);

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

bestående af to vektorer (`resource` og `available`) og tre matricer (`max`, `allocation` og `need`). Disse er også beskrevet i lærebogen afsnit 7.5.

Begyndelsestilstanden specifices ved hjælp af en input `.txt` fil, fx givet ved formatet

```
<m>
<n>

<resource vector>

<max matrix>

<allocation matrix>
```

som så inlæses via stdin når programmet køres med `./banker < input.txt`. Her er `m` antallet af processer og `n` antallet af resurser.

Et eksempel på `input.txt`:

```
4
3

9 3 6

3 2 2
6 1 3
3 1 4
4 2 2

0 0 0
0 0 0
0 0 0
0 0 0
```

Værdierne for `<allocation matrix>` kan sættes forskellig fra 0 for at afprøve koden i bestemte kritiske situationer (prøv f.eks. `input2.txt`, som er den „ustabile“ tilstand vi så på til forelæsningen).

Efter at input er indlæst, vil den ufuldstændige kode `banker.c` starte m tråde, der simulerer de m processer, og skiftevis forespørger på resurser og frigiver resurser

uden at afslutte. Antallet af resurser der forespørges eller frigives er implementeret tilfældige. Det gøres ved at sikre sig, at der på intet tidspunkt frigives flere resurser end allokeret til processen eller forespørges på flere resurser end der maksimalt er tilladt for processen.

En forespørgsel fra en proces skal kun tildeles hvis tildelingen vil resultere i en sikker tilstand. Til dette skal banker's algoritme implementeres. Ellers skal processen vente til senere (`Sleep(100)`) og prøve igen med samme forespørgsel.

Du skal (som minimum) implementere følgende punkter i den ufuldstændige kode:

- Allokere hukommelse dynamisk til de vektorer og matricer, der skal anvendes.
- Banker's safety-algoritme bruges til at afgøre om en tilstand er sikker eller ej og dermed om en forespørgsel skal tildeles eller ej. Dette gøres i funktionen;

  ```
  int resource_request(int i, int *request) {}
  ```

- Frigivelse af resourcer skal implementeres i:

  ```
  void resource_release(int i, int *request) {}
  ```

- Du skal sikre dig at begyndelsestilstanden er sikker før trådene startes.
- Du skal sikre dig at al tilgang til delt hukommelse undgår race conditions.

Hints:

- En måde at allokere matricer i C er vist i eksemplet med matrix multiplikation.
- Begynd med at lave en funktion, der tager en tilstand som input og returnerer om den er sikker eller usikker.
- Du kan med fordel udskrive `available` vektoren hver gang den ændres.

Rapport

I rapporten skal du beskrive hvordan du har implementeret punkterne der spørges til i opgaverne. Heruden skal du redegøre for hvordan du har testet at disse fungerer efter hensigten. Al relevant kildekode (+evt. Makefile) skal inkluderes i et appendix.