Obligatorisk opgave 1

Operativsystemer og C

Bachelor in Software Development,
IT-University of Copenhagen

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

October 3rd, 2012

Contents

1 Forord

I denne rapport dokumenterer vi vores valg i forhold til implementationen af opgaverne i Obligatorisk opgave 2.
Vores implementation kan findes i "".

Kildekode og testdokumentation kan findes i appendix på side 6. Vores git repository kan findes på https://github.com/esfdk/BOSC .

2 Beskrivelse af implementation

2.1 Opgave 1

Vi skal optimerer en multrådet funktion, der udregner sum-funktioner, således at den faktisk kører hurtigere på en multicore maskine end den ellers ville have gjort uden implementationen af tråde. Og tjekke om vores ændringer gør funktionen hurtigere vha. en speedupgraf.

2.1.1 Del 1

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

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

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

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

Beregning af resultat For at finde det samlede resultat går vi igennem arrayet af structs og tilgår deres sumsqrt værdi. Alle disse værdier ligger vi sammen og printer dem ud.

2.1.2 Del 2

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

!!!!!!!!!!!!!!!indsæt speedup funktion !!!!!!!!!!

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

!!!!!!!!!!!!!!! indsæt ideal speedup graf!!!!!!!!!!!!!!!!!!!!!!!!

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

Uventet resultat ved kørsel med fire tråde !!!!!!!!!!!!!!!!!! indsæt speedup graf her !!!!!!!!!!!!!!!!

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

!!!!!!!!!!!!!!!!!!1 indsæt melnyk speedup graf her!!!!!!!!!!!!!!!!!!!!!!!!

Vi har kort testet vores funktion på en laptop med 4 processor og hyperthreading, hvor vi havde en normal speedup optil fire tråde. Dette er forståligt da fire processor med hyperthreading kun 'simulerer' otte processor, og derfor ikke arbejder ligeså optimalt som otte konkrete processor.

2.1.3 Tests

Til at teste om vores forbedret multitrådet sum-funktion kørte hurtigere med flere tråde, brugte vi en speedupgraf som nævnt i 2.1.2. For at lave en ordenlig speedupgraf lavede vi 10 runs af funktionen ved 1-2-4-6-8 tråde ved input 10.000. Input blev gjort 10 gange højere op til 100.000.000 samt fra 50.000 og så gjort 10 gange højere op til 500.000.000. Dette gjorde, at vi havde en stor mængde information til at lave en god gennemsnits graf.

!!!!!!!!!!!!!!!!!! bitly.com/U0EYAK !!!!!!!!!!!!!!!!!!

2.2 Opgave 2

2.2.1 Del 1

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

2.2.2 Del 2

Det mest åbenlyse problem er, at flere tråde kan editere listen på samme tid. For eksempel: To tråde tilgør listen på samme tid og finder frem til det sidste element ('last'). Begge tråde prøver at tilføje et element til listen, hvilket betyder at de begge prøver at tilføje et element til 'last' på samme tid. Tråd #1 tilføjer sit element til 'last', hvorefter tråd #2 tilføjer sit element til 'last' (det element som #1 lige har tilføjet sit element til), hvilket ødelægger linket mellem 'last' og tråd #1s element.
En anden udgave af problemet ovenover er, hvis to forskellige tråde vil remove på samme tid. Tråd #1 går ind og læser 'first's 'next' element (da det skal blive first efter removal). Samtidig går tråd #2 ind og kører hele removal, hvorefter #1 prøver at fjerne det element der lige er blevet fjernet. Derved er kun et element blevet fjernet, hvor to elementer burde have været fjernet. Den omvendte situation kan også opstå, hvor to elementer bliver fjernet, men hver tråd tror kun at et element er blevet fjernet.

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

2.2.3 Del 3

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

2.2.4 Tests

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

2.3 Opgave 3

I denne opgave har vi brugt 'pthread_mutex_t' objekter til at undgå problemer hvor flere tråde ændrer samme element på samme tid. 'pthread_mutex_t' er objekter, der kan låses/låses op via 'pthread_mutex_lock'-funktions kaldet. Hvis en funktion låser en mutex, forhindre den derved andre funktioner i at køre videre, forudsat at de selv skal bruge mutex'en. De andre funktioner går i stå indtil mutex'en låses op igen, hvilket sørger for at critical kode kun køres af en tråd ad gangen. Bruger man ikke mutex låse i et flertrådet program, risikerer man at løbe ind i situationer hvor flere tråde har modificeret samme element på samme tid, hvilket kan ødelægge programmet.
Vi har også brugt 'sem_t' objekter, også kaldet semaphore. Det er objekter, der indeholder en værdi, og som kan bruges af funktioner til synkronisation mellem tråde. De har to interessante funktioner: 'sem_wait(sem_t)' og 'sem_post(sem_t)'. 'sem_wait()' kigger på værdien af semaphoren. Hvis værdien er mindre end nul, vil den kaldende funktion pause, og vente på værdien bliver højere end nul. Når det sker, vil den kaldende funktion få lov til at køre videre og 'sem_wait()' reducerer værdien af semaphoren med en. 'sem_post()' forøger simpelthen bare værdien af semaphoren med en.
Både 'phtread_mutex_t' og 'sem_t' objekter virker som låse, omend på forskellige måder. 'pthread_mutex_t' virker som en ja/nej, og vil kun lade en tråd udføre sit arbejde ad gangen. 'sem_t' lader gerne flere tråde arbejde på samme tid, så længe semaphorens værdi er over nul.

2.3.1 Opfyldelse af punkter

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

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

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

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

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

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

2.3.2 Tests

2.4 Opgave 4

2.4.1 Tests

A Test

A.1 Opgave 1

A.2 Opgave 2

A.3 Opgave 3

A.4 Opgave 4

A.5 Opgave 3 output

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