Operativsystemer og Multirogrammering
G-opgave 2

Ronni Elken Lindsgaard - 0911831791
Hans-Kristian Bjerregaard - 0612862087
Alexander Winther Uldall - 2908872013

23. februar, 2010

1 En ring af tråde

I ring.h er defineret to datastrukturer; *data* og *baton*. Baton er den stafet som bliver delt mellem trådene, helt specifikt er den et specifikt adresserum som de alle kan tilgå efter tur. Data definerer et datasegment for den specifikke tråd som fortæller dens id, dens condition, hvem der er den næste i rækken og hvor den skal finde sit delte adresserum. Ved initialisering bliver datasegmenter og stafetten skabt og der sørges for at alle tråde er låst til at starte med.

Inde i tråden låser tråden først mutexen hvorefter den finder ud af om det egentlig er dens tur til at køre ved at teste om dens id er lig det id der skal køre hvis ikke sender den signal til den næste tråd der så forsøger på det samme. Dette fortsætter den med hver gang det ikke er dens egen tur.

Hvis det rent faktisk er den tråd der skal køre starter den med at sætte current til den næste tråd i ringen og sende et signal, dette gør at den stiller sig i "venteposition"og begynder så snart mutexen unlocker.

Der tælles nu om der er nået en omgang og hvor mange omgange der er kørt, hvis der er kørt det antal omgange specificeret i `NUM_RUNS` bliver stafettens stop-signal sat og værdien bliver kopieret over i en lokal variabel (for at undgå race conditions). Stafettens værdi kopieres over umiddelbart inden en opdatering for at sikre at tråden bliver kørt en sidste gang med stopværdien som angivet i opgavebeskrivelsen.

Til sidst udføres noget specifikt arbejde (såsom at printe en linie ud) og mutexen bliver aflåst.

Efter alle tråde er kørt færdig bliver de joinet hvorefter allokeret hukommelse bliver rydtet op.

2 Trådbeskyttet arbejdskø

2.1 Pririoritetskøen

Vi har valgt at minimere den krittiske region så meget som muligt og har derved implementeret selve trådsikringen helt inde i prioritetskøen `pqueue.c`. Dette ændrer ikke på selve måden programmet opfører sig på, men sikrer at al brug af pqueue er trådsikret. Ved kun at implementere trådsikringen hvor det er mest nødvendigt kan en optimal mængde kode køres parallelt, samtidig med at men får et simplere og mere gennemgående design (man skal f.eks. ikke huske at bruge specielle, trådsikrede funktioner når man arbejder på køen).

Selve implementeringen sker ved at en process tilegner sig en lås så snart den vil arbejde med den linkede liste der er kernen i pririoritetskøen. For `pqueue_insert()` gælder det når der scannes igennem listen for at finde den korrekte position samt når elementet reelt set indsættes. Der behøver dog ikke tilegne sig låsen når man allokerer hukommelse og gemmer informationer om det element der skal indsættes. Næsten hele `pqueue_remove()` skal

trådsikres da den kun arbejder med den linkede liste.

2.2 Tråd-poolen

`wqueue_ts_insert` skal således ikke indeholde nogen trådsikring. Derfor skal der blot signaleres til evt. ventende tråde hvis indsætningen lykkedes.

`wqueue_thread_pool` implementeres ved at modtage en arbejdskø, der allerede indeholder arbejde, og så oprette tråde til at udføre arbejdet. Hver tråd modtager en struct med dens arbejdskø samt id (dette er ikke tråd id'et!). Efter alle tråde er oprettet begynder `wqueue_thread_pool` at joine dem igen efterhånden som de terminerer.

Selve trådens arbejde bliver udført af `wqueue_thread`. Hver tråd kører så længe der er andre tråde der er ved at udføre et stykke arbejde eller der stadig er arbejde i køen. Når der skal udføres et arbejde opdateres arbejdstælleren `threads_working`, arbejdet udføres og arbejdstælleren dekrementeres igen. Ændring ef arbejdstælleren er selvfølgeligt sikret med en lås da det er krittisk at den er korrekt. Her efter kunne tråden potientielt fortsætte loopet men dette ville kræve en overflødige masse recourcer hele tiden at tjekke tæller og arbejdskø. Derfor undersøges der om køen er tom og om der er andre der arbejder. Er køen ikke tom gentages loopet for at udføre mere arbejde, ellers, hvis der er andre processer der arbejder, begynder den nuværende process at vente på signal om at der er indsat nyt arbejde i køen. Er der ikke mere i køen og er der ikke flere processer der arbejder afsluttes løkken og tråden terminerer.

Dog skal man være opmærksom på at der er tråde der ender i en potientiel dead-lock ved tråde der venter på et signal der måske aldrig bliver sendt. Det er jo ikke til at forudsige om arbejdsfunktionerne indsætter arbejde i køen eller ej. Denne dead-lock situation kan dog nemt undgås ved at alle tråde lige inden de terminerer også sender et signal. Når en tråd terminerer vil en af de tråde der venter på signal så fortsætte, og selv terminerer da der ikke er mere data i køen og ikke mere arbejde tilbage. Det er sikret at der altid er en tråd der terminerer da den sidste tråd der ikke er i dead-lock altid vil detektere at ingen andre arbejder og hvis den selv har udført det sidste arbejde må den terminere. Derved undgås dead-lock ved at lade trådene signalere når de terminerer.