Godkendelsesopgave 3 - OSM 2012

Maria Caroline Miller, 040779, twq135
Søren Pilgård, 190689, vpb984

5. marts 2012

1 Opgave 1 - låse og betingelsevariabler

1.1 Låse til kernetråde i Buenos

1.1.1 Typedefinition af mutex

Lock_t har en spinlock til at låse med og en variabel som indikerer om låsen er låst eller ej.

1.1.2 lock_reset(lock_t *lock)

Der returneres -1, hvis lock_t ikke allerede er allokeret. Ellers sættes locked-variablen til 0, og der kaldes spinlock_reset med låsen.

1.1.3 lock_aquire(lock_t *lock)

Interrupt disables, og der hentes en spinlock med spinlock_acquire. Hvis låsen allerede er låst (locked == 1), bliver forespørgslen tilføjet til en sovekø, og der arbejdes videre med en anden forespørgsel. Når låsen er ledig (locked bliver sat til 0 et sted) og det er muligt at skaffe den låses variablen igen, og spinlocken løftes, og interrupts enables igen.

1.1.4 lock_release(lock_t *lock)

Interrupt disables, spinlocken hentes. Låsevariablen sættes til 0, og sovekøen vækkes. Derefter åbnes spinlocken igen og interrupt enables.

1.2 Betingelsesvariabler til kernetråde i Buenos

1.2.1 Typedefinition af condition variables

Da cond_t ikke rigtig skal indeholde nogle oplysninger bliver typen sat til en simpel int.

1.2.2 condition_init(cont_t *cond)

Typen sættes bare til at pege på 1337.

1.2.3 condition_wait(cont_t *cond, lock_t *lock)

Interrupt disables. Det tjekkes om kalderen rent faktisk har husket at låse låsen, inden den kalder wait. Sovekøen sættes igen, med betingesesvariablen, som parameter, og låsen åbnes op. Der skiftes til en anden tråd mens der ventes. Derefter samles låsen op igen, og interrupt enables igen.

1.2.4 condition_signal(cond_t *cond)

Der gives signal til at vække en tråd fra sovekøen. Der behøves ikke mere i forbindelse med 'signal and continue'-ideen da vi i condition_wait bruger funktionen lock_acquire, som først giver låsen fra sig når den er helt færdig, eller har lagt sig til at sove. Derfor klares det automatisk, at ingen signaler risikerer at gå tabt.

1.2.5 condition_broadcast(cont_t *cond)

Denne funktion skal bare vække alle i sovekøen, og dette gøres nemt med wake_all funktionen i sleepq.

2 Opgave 2 - Pthreads-baseret løsning til CREW-problemet

Udfordringen er at lave en løsning på 'Concurrent Read - Exclusive Write'-problemet. Tråde der læser data kan sagtens gøre dette på samme tidspunkt, hvorimod det er nødvendigt at skrivning foregår alene.

2.1 Basis implementation - Writer-starvation

2.1.1 write_counter(void *p)

Der laves en while-løkke, der kører et antal iterationer, så hver skrivetråd skriver flere gange. Der sættes en skrivelås, hvorefter der skrives (en tæller tælles op). Det tælles hvormange vi har skrevet i alt, og låsen åbnes igen.

2.1.2 read_counter(void *p)

Igen har vi en while-løkke, der kører igennem flere iterationer. Her starter vi dog med at sætte en lås på counteren. Derefter lægger vi en til vores læsetæller og tjekker om den er 1 (i begyndelsen sættes den til 0). Der tjekkes om det er

den første reader - i så fald låser vi skrivelåsen. Hvis læsetælleren er større end 1 har vi allerede låst for skrivelåsen, og vi kan derfor bare blive ved med at læse. Dette sikrer at vi kan have concurrent read. Tællelåsen lukkes op, og vi læser fra data. Derefter låses tællelåsen igen, og det tjekkes om der er flere læsere. Hvis der er flere låser vi bare tællerlåsen op igen, så en ny læser kan få chancen. Hvis der ikke er flere lukker vi skrivelåsen op inden vi åbner for tællerlåsen.

2.1.3 Main-funktionen

Selve mainfunktionen opretter plads til vores læse- og skrivetråde. Derefter oprettes de med pthread_create i en løkke, så der dannes et bestemt antal af hver. Derefter køres en løkke pthread_join, som venter på at vores tråde er færdige med at køre. Det udskrives hvormange tråde vi har kørt, og hvormange læsninger og skrivninger, der er udført. Derefter sletter vi låsene igen med pthread_mutex_destroy, og afslutter programmet med et kald til pthread_exit.

2.2 Resten

Vi nåede ikke at løse opgave 2b og 2c.