

Implementation af trådsift på brugerniveau

Jørgen Sværke Hansen

22. februar 2007

1 Introduktion

Et trådbibliotek kan enten realiseres på brugerniveau eller på kerneniveau. Formålet med dette dokument er at give en kort introduktion til hvordan trådsift kan implementeres på brugerniveau.

En proces består af en samling ressourcer (lager, åbne filer, netværksforbindelse, ...) og en eller flere tråde, hvor trådene er den aktive del af processen. Som det allerede er gennemgået i lærebogen, skal man ved processkifte, gemme hele tilstanden for den aktive proces og reetablere tilstanden for en tidligere gemt proces. Ved trådsift foretages en lignende operation; dog skal man ikke gemme og skifte tilstand som er relateret til ressourcedelen af processens tilstand, idet denne deles mellem processens tråde. Der skal blot skiftes mellem forskellige udførsler af en delt programkode. Ser vi på hvilke dele af processens tilstand, en sådan udførsel af programkode påvirker (ud over de delte ressourcer), drejer det sig om:

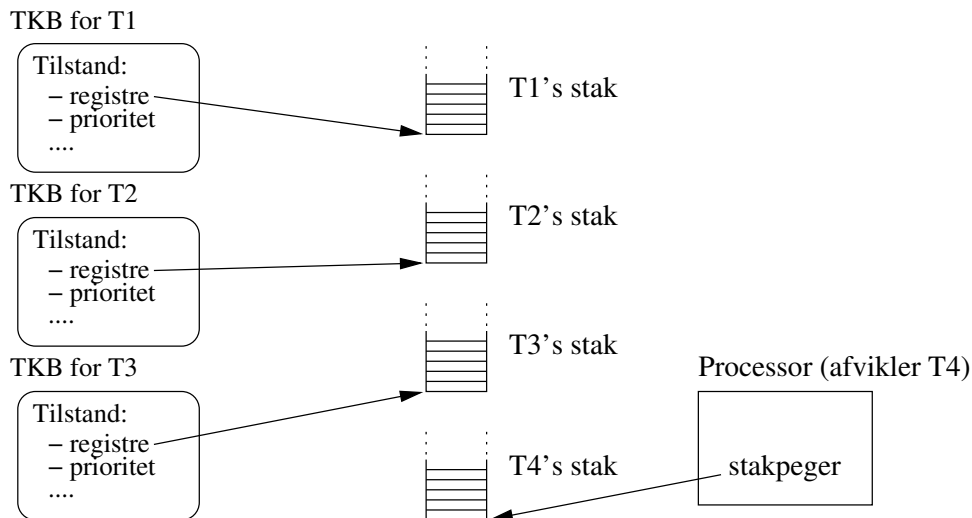
processorens registersæt tråden bruger de almindelige registre til opbevaring af delresultater. Derudover er der **programpegeren** (eng.: program counter) der angiver hvilken instruktion, der udføres som den næste, et **statusregister**, der f.eks. angiver resultatet af tidligere operationer, og en **stakpeger**, der angiver toppen af stakken.

stak selve stakken kan bruges af tråden til allokering af midlertidigt lager. Desuden bruges den til at gemme returadressen samt parametre på ved funktions-/procedurekald.

Når vi skal gemme en tråds tilstand for at skifte mellem forskellige tråde, skal vi derfor dels gemme indholdet af stakken; dels gemme indholdet af processorregistrene. Som ved processer gemmes disse informationer i en kontrolblok knyttet til den givne tråd; ud over trådens tilstand kan trådkontrolblokken (TKB'en) indeholde administrative informationer såsom trådens ID, prioritet og afviklingstilstand.

Den nemmeste måde at gemme indholdet af en tråds stak, er at allokere en stak per tråd, og det er oftest den metode der anvendes. Når der skiftes mellem forskellige tråde kan man derfor nøjes med at skifte stakpegeren fra en stak til en anden, i stedet for at skulle gemme hele indholdet af trådens stak. Da stakpegeren gemmes i trådens kontrolblok sammen med resten af processorens registersæt, er det nemt at finde stakken frem igen senere.

Figur 1 viser et skematisk øjebliksbillede af en proces med 4 tråde T1 til T4, hvoraf kun T4 kører. Det er illustreret hvordan alle tråde har allokeret hver deres stak i hovedlageret, og hvordan deres trådkontrolblokke indeholder en reference til denne stak. T4's tilstand er ikke eksplicit gemt på dette tidspunkt, men når/hvis der skal skiftes til en anden tråd, vil T4's tilstand blive gemt i en TKB.



Figur 1: En proces med 4 tråde, hvor hver tråd har fået allokeret en stak og en trådkontrolblok. Tråden T4 er under afvikling hvorimod trådene T1 til T3 har fået gemt deres tilstande. TKB'erne for de tre gemte tråde refererer til den respektive tråds stak.

2 Implementation af trådskeft og oprettelse af tråde

Beslutning om at foretage et trådskeft kan enten foretages af den aktive tråd eller af den komponent der styrer afviklingen af trådene (f.eks. et trådbibliotek eller styresystemet). I det første tilfælde tales der om et **frivilligt trådskeft** idet tråden jo af egen fri vilje vælger at frigive processoren, og i det andet tilfælde tales der om et **tvungent trådskeft**, idet tråden er sat uden for indflydelse og tvinges bort fra processoren. I dette afsnit vil vi holde os til frivillige trådskeft, da det gør det nemmest at følge afviklingen af flere tråde. Den underliggende mekanisme til at foretage selve trådskeftet er den samme i de to tilfælde, og kan derfor overføres fra frivilligt til tvungent trådskeft.

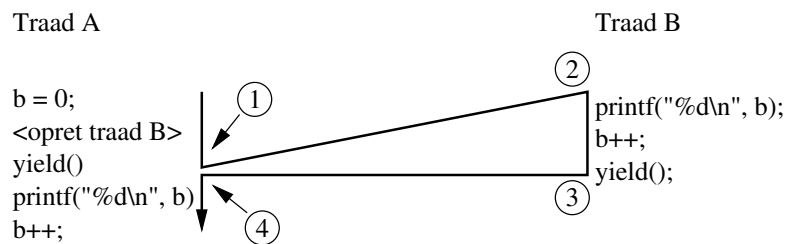
2.1 Et trådskeft i store træk

Nedenfor vises et kodeudsnit, der giver et eksempel på to tråde A og B, hvor tråd A opretter tråd B for derefter at anvende funktionen `yield()` til at foretage et frivilligt trådskeft. Idet vi antager at skeduleringen af trådene er FIFO, vil det resultere i at der skiftes til tråd B. Tråd B udskriver værdien af variabelen `b` (en global heltalsvariabel), tæller den 1 op og foretager derefter endnu et frivilligt trådskeft, der giver kontrollen tilbage til tråd A. A foretager endnu en udskrivning og opdatering af `b`.

Tråd A	Tråd B
<code>b=0;</code>	
<code><opret tråd B></code>	<code>printf("%d\n", b);</code>
<code>yield();</code>	<code>b++</code>
<code>printf("%d\n", b);</code>	<code>yield();</code>
<code>b++</code>	<code>...</code>
<code>...</code>	

I figur 2 illustreres det ovennævnte hændelsesforløb, hvor pilen angiver udførselsforløbet. Vi kan i dette simple tilfælde se hvordan kaldet af `yield()` skifter mellem de to aktiviteter, hvor kaldet af `yield()` i tråd A gemmer A's tilstand ved punkt 1 og reetablerer den i punkt 3. Ligeledes

vil tråd B's tilstand blive reetableret i punkt 2 og gemt igen i punkt 3. Idet tråd B aktiveres for første gang i punkt 2, skal man ved oprettelsen af tråd B sørge for at konstruere en initial tilstand for B der sørger for at kalde B's hovedfunktion når der skiftes til B første gang.



Figur 2: En tråd A opretter en anden tråd B, for derefter at udføre et frivilligt trådsift. Herefter udføres tråd B indtil næste frivillige trådsift, idet det antages at trådene skeduleres FIFO. Derefter skiftes tilbage til tråden A.

2.2 Skitse for implementation af trådsift

Vi skal nu se nærmere på hvordan vi kan implementere selve det frivillige trådsift - altså den kode, der skal implementere funktionen `yield()`. Som det indledningsvis er blevet diskuteret, så realiseres et trådsift fra tråd A til tråd B groft sagt ved at gemme tilstanden for den kørende tråd (A) og derefter erstatte det med et tidligere gemt tilstand for tråd B. Tilstanden er beskrevet af registersættet, da det jo indeholder en stakpeger og en programpeger, så aktiveringsposter på stakken og det aktuelle udførselspunkt også gemmes. Tilstanden for tråd B bliver ligeledes reetableret når Bs gemte registersæt bliver gjort til den aktuelle tilstand for processoren, og B vil fortsætte med at afvikle programkode fra det punkt, som programpegeren var nået til da B's tilstand blev gemt.

De gemte registersæt placeres oftest i en kontrolblok for tråden sammen med andre kontrolinformationer omkring tråden, f.eks. ID og prioritet. Disse kontrolblokke vil typisk være placeret i forskellige ventekøer: der er en *klarkø* der indeholder tråde der venter på adgang til processoren og desuden vil der være en ventekø per lås og per betingelsesvariabel. Endelig kan der være ventekøer forbundet med ydre enheder og andet hardware. Desuden skal der gemmes en reference til kontrolblokken for den aktive tråd, idet denne jo ikke er placeret i nogen ventekø - dette håndteres ved en global variabel, der peger til den aktive tråds kontrolblok (denne variabel har ofte navnet *current*). Et trådsifte vil dermed have en form som vist i figur 3. Idet vi gemmer og reetablerer tilstande som det sidste i `yield`, er det vigtigt at den gamle tråds tilstand gemmes på en måde, så den genoptager afviklingen med at udføre `return` fra `yield`, idet vi ellers ville kunne få en uendelig kæde af operationer, der reetablerer tilstanden for den nye aktive tråd.

Lad os vende tilbage til eksemplet med trådene A og B, hvor vi betragter trådsiftet angivet ved punkterne 3 og 4 i figur 2. Det vil give anledning til afviklingen af en sekvens af operationer som angivet i figur 4, hvor vi ser hvordan skiftet mellem B's og A's tilstand sker umiddelbart inden der returneres fra `yield`.

Betragter vi systemets for A og B under trådsiftet ved punkt 3 (lige efter at tråd B har kaldt `yield`), vil vi have en situation som skitseret på figur 5, hvor B er den aktive tråd (*current* peger på B's TKB) og hvor A venter som første (og eneste) TKB i klarkøen. Betragtes stakkene for de to tråde, er det angivet at begge er i en tilstand, hvor aktiveringsposten for `yield` er det øverste element. De to aktiveringposter vil være ens bortset fra returadresserne, der vil returnere til hhv. tråd A's hovedfunktion og tråd B's hovedfunktion.

```

void yield()
{
    <indsæt "current" TKB i klarkø>
    <udtag TKB for ny tråd fra klarkø>
    <opdater "current" peger til at pege på ny tråds TKB>
    <gem gammel tråds tilstand i den gamle tråds TKB>
    <reetabler ny aktiv tråds tilstand fra TKB refereret af "current">

    return;
}

```

Figur 3: Skitse af implementation af trådsift

```

.
.
tråd B's afvikling
.
.
printf("%d\n", b);
b++
<kald yield()>
<indsæt B's kontrolblok i klarkø>
<udtag kontrolblok for ny tråd (A) fra klarkø>
<opdater "current" peger til at pege på A's TKB>
<gem tråd B's tilstand i B's kontrolblok>
<reetabler aktiv tråds (A's) tilstand fra TKB refereret af "current">
<returner til returadresse på toppen af stakken (A's)>
printf("%d\n", b);
b++
.
.
tråd A's afvikling
.

```

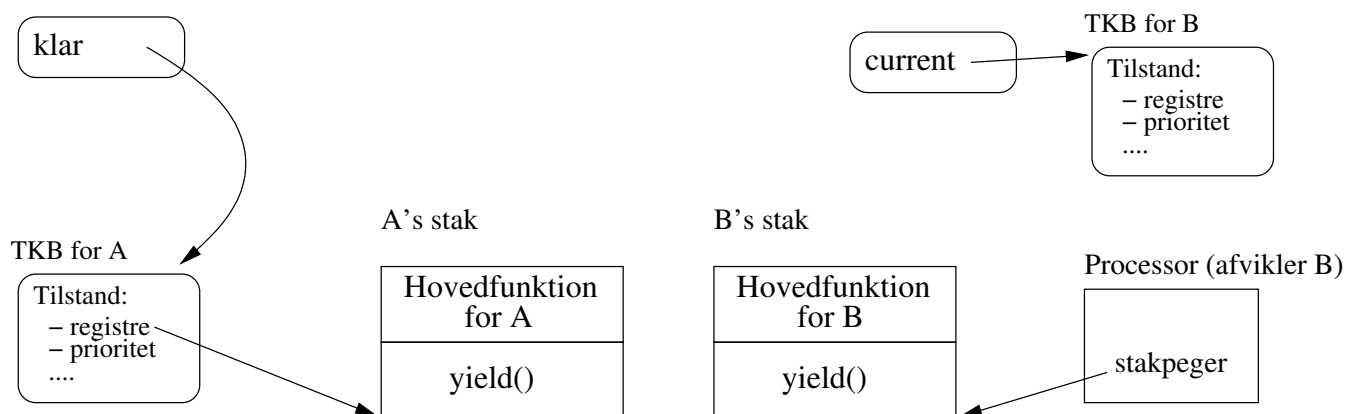
Figur 4: Sekvens af operationer under trådsift mellem trådene B og A

2.3 makecontext og venner

De fleste programmeringssprog understøtter ikke direkte det at gemme og hente registersæt for en tråd. På visse UNIX systemer, heriblandt Linux, findes dog kaldende `getcontext`, `setcontext`, `makecontext` og `swapcontext`, der som navnene antyder kan anvendes til at manipulere den aktive tilstand (kontekst) og derved bruges til både at foretage trådsift og oprette tråde.

2.3.1 Trådsift

Med `swapcontext(oucp, ucp)` kan man gemme tilstanden for den kaldende tråd og skifte til tilstanden for en anden tråd. Den gamle tilstand gemmes i en struktur af typen `ucontext_t` der specificeres ved parameteren `oucp` og den nye tråds tilstand er specificeret ved samme type struktur via parameteren `ucp`. Kaldet resulterer i at vi gemmer den gamle tråds kontekst og



Figur 5: Et øjebliksbillede af trådenes tilstand umiddelbart inden tråd B laver et frivilligt trådskeft til A.

skifter til den nye tråds kontekst - skulle vi på et senere tidspunkt skifte tilbage til den gamle tråds kontekst, vil den gamle tråd fortsætte sin afvikling ved at returnere fra `swapcontext`. Implementationen af `yield()` fra før vil med brug af `swapcontext` se ud som i figur 6, idet vi som en del af trådkontrolblokken har allokeret et felt af typen `ucontext_t` ved navn `tilstand`, som bruges til at gemme tilstanden for trådene i.

```
yield()
{
    <indsæt "current" TKB i klarkø>
    <udtag TKB for ny tråd fra klarkø>
    <opdater "current" peger til at pege på ny tråds TKB>
    swapcontext(&gammelTKB->tilstand, &current->tilstand);

    return;
}
```

Figur 6: Skitse af trådskeft mellem trådene A og B med brug af `swapcontext`

2.3.2 Oprettelse af tråde

Når vi starter har vi dog kun en enkelt tråd, så for at have noget at skifte til, skal vi have oprettet flere tråde. For at kunne gøre dette skal vi bruge:

- en ny stak
- en hovedfunktion
- en gemt tilstand, der anvender den nye stak, og starter med at kalde hovedfunktion.

Når vi anvender `swapcontext` til trådskeft, kan vi få oprettet en sådan ny tilstand ved hjælp af funktionen `makecontext`. Den tager som argumenter en peger `ucp` til en `ucontext` struktur, en funktionspeger `func`, antallet `argc` af argumenter til `func` samt de faktiske argumenter til funktionen (angivet ved ...):

```
void makecontext(ucontext_t *ucp, void *func(), int argc, ...)
```

`ucp` skal pege på en `ucontext` struktur med en frisk stak (vi anvender jo en stak per tråd), så denne struktur skal først oprettes. Dette gøres ved at tage en kopi af den kørende tråds tilstand med kaldet `getcontext`, derefter oprette en ny stak, og endelig gemme en reference til stakken samt stakken størrelse i bytes i felterne `uc_stack.ss_sp` og `uc_stack.ss_size` i `ucontext_t` strukturen. Hele oprettelsen af en ny tråd kommer således til at se ud som følger:

```
<alloker ny TKB og opret en stak med størrelse STACK_SIZE>
if(getcontext(&TKB.tilstand) != 0)
    <Uoprettelig fejl i getcontext>
TKB.tilstand.uc_stack.ss_sp = TKB.stak;
TKB.tilstand.uc_stack.ss_size = STACK_SIZE;
TKB.tilstand.uc_stack.ss_flags = 0;
TKB.tilstand.uc_link = NULL;
makecontext(&TKB.tilstand, hovedfunktion, <antal args>, ...);
```

idet vi lader `STACK_SIZE` angive størrelsen af stakken. Efter kaldet til `makecontext` vil et kald af `swapcontext` med den initialiserede `ucontext_t` struktur resultere i at den angivne hovedfunktion `func` afvikles som en ny tråd med sin egen stak.

2.3.3 Et større eksempel

I bilag A kan ses et eksempel på anvendelse af `makecontext` og `swapcontext`, hvor to tråde skiftes til blive afviklet ved hjælp af frivillige trådsift. Kildeteksten til dette eksempel har navnet `twothread.c` og kan hentes på hjemmesiden.

A Eksempel på frivilligt trådsift (twothread.c)

```
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>

/* Vi identificerer vores traad med en integer */
typedef int othread_t;

/* Vi gider ikke have noget med attributter at goere i foerste omgang */
typedef struct _othread_attr_t {
} othread_attr_t;

#define STACK_SIZE 1024

/* I forhold til teksten gemmer vi ogsaa en reference til
   de argumenter fra create, som vi skal bruge til at
   starte traaden med */
struct tkb {
    void *(*start_routine) (void *);
    void *arg;
    ucontext_t tilstand;
    int    stak[STACK_SIZE];
};

/* Vi skal holde rede paa vores traade via kontrolblokkene: i dette
   eksempel begrænser vi os til to traade der kan skifte mellem
   hinanden. Vi har altsaa brug for en peger til kontrolblokken for den
   aktive traad (her current) og en peger til kontrolblokken for den
   ventende traad (her ready).

   Vi har dog et ekstra problem: den initielt koerende traad har ikke
   kaldt thread create og har derfor ikke faaet allokeret en
   kontrolblok. Men det kan vi selv goere idet der kun er netop en
   initiel traad. current skal saa pege paa denne traad naar det hele
   starter. Indholdet af stak, start_routine og arg er ikke vigtigt
   for denne foerste traad, idet hele dens tilstand er allokeret fra
   start af. For nemheds skyld vaelger vi dog at have en hel TKB
   alligevel. */

static struct tkb first_thread;

struct tkb *current = &first_thread;
struct tkb *ready = NULL; /* Vi har ikke en ventende traad endnu */

typedef void (*arg_fn)(void);

void thread_wrapper(struct tkb *my_tkb)
{
    void *retval;

    /* kald traadens hovedfunktion - bemaerk at vi nu bruger current til
       at faa fat i tkb'en med. Vores gamle peger er vaek, idet den
       anden traad har returneret fra othread_create. */
    retval = my_tkb->start_routine(my_tkb->arg);
    /* terminer afvikling af traad - ikke implementet - vi stopper helt */
    printf("En traad har returneret med returvaerdi %p - alt stoppes\n", retval);
    exit(0);
}

/* Opretter en ny traad - denne kan kun kaldes en gang idet vi kun kan
   haandtere to traade */
int othread_create (othread_t *threadp,
    const othread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg)
{
    struct tkb *tkb_peger; /* Denne er til vores nye traadkontrolblok */

    /* Foerst allokeres en kontrolblok til den nye traad */
```

```

tkb_pegger = (struct tkb *) malloc(sizeof(struct tkb));
if(tkb_pegger == NULL) {
    /* afslut med fejl - vi fik ikke allokeret lageret */
    exit(-1);
}
/* Derefter gemmer vi traadens start_routine og argument, idet vi
   gerne vil bruge dem naar vi starter funktionen foerste gang, men
   denne funktion vil have returneret og parametre samt lokale
   variable er vaek */
tkb_pegger->start_routine = start_routine;
tkb_pegger->arg = arg;
if(getcontext(&tkb_pegger->tilstand) != 0) {
    printf("Uoprettelig fejl i getcontext\n");
    exit(-1);
}
tkb_pegger->tilstand.uc_stack.ss_sp = tkb_pegger->stak;
tkb_pegger->tilstand.uc_stack.ss_size = STACK_SIZE*sizeof(int);
tkb_pegger->tilstand.uc_stack.ss_flags = 0;
tkb_pegger->tilstand.uc_link = NULL;
makecontext(&tkb_pegger->tilstand, (arg_fn) thread_wrapper, 1, tkb_pegger);

/* Gem pegger til denne tilstand i en klarkoe - vi forenkler og
   tillader kun to aktive traade: en der er under afvikling og en
   der er klar men venter */
ready = tkb_pegger;

/* Vi bruger tkb_pegger som vores traad ID */
*threadp = (int) tkb_pegger;

return 0;
}

/* Skifter mellem vores to traade */
int othread_yield (void)
{
    if(ready != NULL) {
        /* Der er en anden traad der venter */
        /* Vi bytter om paa den aktive og den ventende traad */
        struct tkb *old = current;
        current = ready;
        ready = old;
        if(swapcontext(&old->tilstand, &current->tilstand) < 0) {
            printf("Uoprettelig fejl i getcontext\n");
            exit(-1);
        }
    }
}

return 0;
}

/*
 * Denne traad udskriver 3 linier til stdout og mellem hver linie
 * udfoeres et frivilligt traadskift.
 */
void *other(void *arg)
{
    printf("Hello world from other\n");

    othread_yield();

    printf("other still going strong\n");

    othread_yield();

    printf("Godbye world from other\n");

    return NULL;
}

```



```

/*
 * Hovedprogrammet opretter netop en traad specificeret ovenfor, og
 * disse to traade skiftes derefter til at udskrive en linie til
 * stdout ved hjælp af frivilligt traadskift
 */
int main(void)
{
    othread_t t1;

    /* Vi opretter en traad der kalder funktionen other med argument NULL */
    othread_create(&t1, NULL, other, NULL);

    printf("Hello world from main\n");

    othread_yield();

    printf("main still going strong\n");

    othread_yield();

    printf("Godbye world from main\n");

    othread_yield();

    /* Anden gang vi afgiver processoren til traaden other vil den
       returnere og hele afviklingen vil standse. Derfor kommer vi aldrig hertil */
    printf("Dette naas aldrig\n");

    return 0;
}

```