
Blekinge Tekniska Högskola

DV1492, DV1556 – (Realtids- och) Operativsystem

# Laboration - Trådprogrammering

Författare: Carina Nilsson, Erik Bergenholtz
Datum: augusti, 2017

# Innehåll

# 1 Syfte

I den här laborationen får du bekanta dig med programmering för flertrådade program. Att använda flera exekveringstrådar i ett och samma program kan ge fördelar i form av kortare svarstider och högre CPU-utnyttjande med prestandavinst som syfte. Du får konkret bekanta dig med POSIX Threads API (Pthreads) för att skapa flera trådar i samma process. Laborationen ger en liten inblick i vilket ansvar programmeraren måste ta för resurshantering och synkronisering vid flertrådsprogrammering.

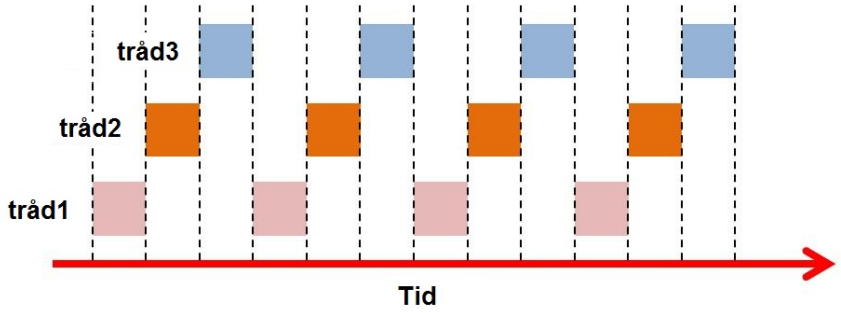
# 2 Förberedande uppgift

Läs i läroboken kap 2.2 om trådar samt kap2.3.6 om mutexvariabler. Innan det handledda laborationstillfället ska även du ha läst igenom hela det här dokumentet ordentligt så att du är införstådd med vad du ska göra. Du ska också ha läst den givna tillhörande programkoden grundligt och försökt förstå den.

# 3 Processer och trådar

En process är ett program under exekvering. En process ”äger” resurser såsom CPU-tid, minne och I/O-enheter, som den använder för att färdigställa sin uppgift. En tråd (ibland refererad till som lättviktsprocess) är grundenheten för CPU-utnyttjande inom en process. Det är trådar som kan allokera CPU-tid och som schemaläggs av operativsystemets scheduler. En tråd består av tråd-ID, en programräknare, status, en upplaga av CPUns registerinnehåll och en stack. Trådar som hör till samma process delar minne i form av kodsegment, global dataarea och heap samt resurser relaterade till operativsystemet.

En traditionell process (tungviktsprocess) har en enda exekveringstråd (kontext). En flertrådad process har flera oberoende exekveringstrådar och kan utföra flera saker samtidigt. En trådgrupp är en mängd trådar som alla exekverar inom samma process. De delar alla samma minne och har access till samma programkod, samma globala variabler, samma heap-area, samma file-descriptors m.m. Alla processens trådar exekveras parallellt, virtuellt genom timeslicing (se figur 1) eller med flera processorkärnor om systemet har det vilket ger äkta parallellitet. De flesta moderna operativsystem stödjer flertrådade processer och de flesta mjukvaruapplikationerna är flertrådade.



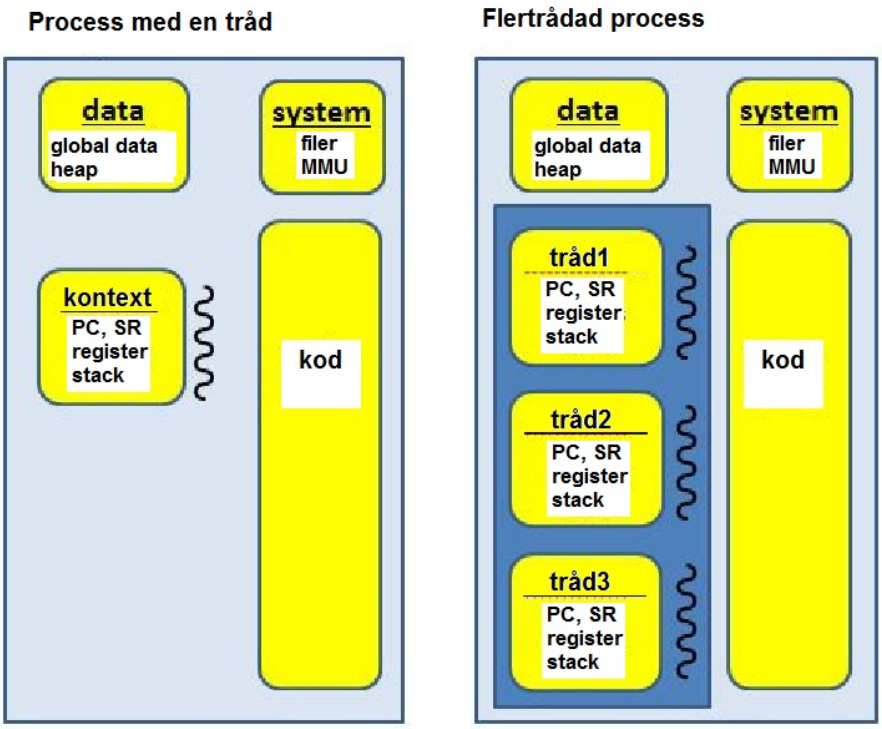

Figur 1: Flera trådar schemalagda på en CPU-kärna

Ett trådbibliotek tillhandahåller ett API för att skapa och hantera trådar. Historiskt sett har olika leverantörer implementerat sina egna versioner av trådar. Dessa implementationer hade betydande skillnader vilket har gjort det svårt för programmerare att utveckla portabla trådade applikationer. Ett standardiserat gränssnitt underlättar portabilitet och Pthreads är en standard beskriven i IEEE POSIX 1003.1c. De flesta leverantörer erbjuder nu Pthreads vid sidan av sina egna APIer.

De tre mest använda APIerna för trådprogrammering idag är POSIX Threads, Windows och Java. I POSIX Threads delas all data som deklareras globalt (utanför någon funktion) mellan alla trådar som hör till samma process. Varje POSIX-tråd har sin egen exekveringskontext, d.v.s. en programräknare (PC), status register (SR), en uppsättning registerinnehåll för CPU och en stack, se figur 2. Lokala variabler delas inte mellan trådar i processen.

Pthreads definieras som en mängd datatyper och funktioner i programspråket C. Pthreads brukar erbjudas i form av en header-fil (include) och ett bibliotek som länkas med programmet.

Det primära skälet att använda trådar är att förbättra prestanda hos program. Jämfört med att skapa och hantera en process kan en tråd skapas med mindre overhead från operativsystemet. Hantering av trådar kräver också mindre systemresurser i övrigt än hantering av processer. Alla trådar i en process delar samma adressarea i minnet, vilket också gör att kommunikation mellan trådar blir mer effektiv än kommunikation mellan processer.




Figur 2: Trådar och processer

Trådade applikationer ger prestandavinster och praktiska fördelar gentemot icke trådade applikationer på flera sätt. Man kan exempelvis låta CPU-intensiva programdelar överlappa I/O, så att en tråd kan utföra CPU-intensivt arbete medan en annan tråd väntar på att ett systemanrop med I/O-förfrågan ska färdigställas. Det kan göra att även ett system med en enda CPU-kärna får ett bättre uppförande genom flera trådar i form av kortare svarstid än en helt sekventiell lösning hade haft. Flertrådade applikationer fungerar på system med en enda CPU-kärna, men kan dra fördelar prestandamässigt av ett system med flera CPU-kärnor utan att koden behöver kompileras om. I en miljö med flera processorkärnor gör användning av trådning att man kan dra nytta av äkta parallellism och på så sätt få högre prestanda i CPU-intensiva parallelliseringsbara programdelar.

Det kan givetvis finnas nackdelar också med att använda flera trådar om det inte görs med eftertanke. Onödiga kontextbyten är ren tidsförlust på grund av den overheadtid de tar. Dessutom ger resursdelning problematik med ömsesidig uteslutning (*mutual exclusion*) och synkronisering, med risk för dödläge (*deadlock*) som följd.

# 4 Liten introduktion till POSIX Threads

Funktionerna i Pthreads API kan grovt delas in i tre grupper.(Det finns ytteligare verktyg i Pthreads, men de beskrivs inte här.)

- **Trådhantering:** Innehåller bland annat funktioner för att skapa och avsluta trådar. Även funktioner för att sätta trådattribut ingår här.
- **Hantering av mutexvariabler:** Innehåller funktioner för att skapa, förstöra, låsa och låsa upp mutexvariabler (variant av binär semafor). Det finns även funktioner för att sätta och modifiera attribut till mutexvariabler.
- **Hantering av villkorsvariabler (*conditions*):** Denna datatyp ger programmeraren möjlighet att synkronisera trådar utifrån egendefinierade villkor. Här finns funktioner för att skapa, förstöra, vänta på och signalera ”condition” baserat på olika variabelvärden. Här finns även funktioner för att sätta attribut för condition. Den här gruppen av funktioner kommer inte att beröras i laborationen.

En listning av ett urval av datatyper och funktionsdeklarationer finns som bilaga.

## 4.1 Funktioner för trådhantering

Innan man kan börja skapa trådar måste filen `pthread.h` inkluderas i programmet.

### 4.1.1 Skapa trådar

Innan man skapar trådar behöver man deklarera en variabel av typen `pthread_t` för varje ny tråd. För varje ny tråd behöver man också anropa funktionen `pthread_create()`, som skapar själva tråden i systemet.

```
1 int pthread_create (pthread_t *threadhandle,        /* referens till den deklarerade variabeln */
2                     pthread_attr_t *attribute,      /* NULL default */
3                     void *(*start_routine)(void *), /* den funktion tråden ska exekvera */
4                     void *arg );                    /* funktionsargument, måste skickas som void* */
```

Funktionen returnerar 0 då trådskapadet lyckas. Om det misslyckas returneras ett negativt värde. Om funktionen inte har någon parameter skickar man in `NULL`.

### 4.1.2 Terminera trådar

Det finns flera sätt en tråd kan terminera på. När funktionen som anropades vid skapadet av tråden återvänder terminerar också tråden, och alla systemresurser tråden håller återlämnas till operativsystemet, på samma sätt som när `main()` i ett program återvänder och

alla processens resurser återlämnas. Tråden kan också termineras med ett explicit anrop till `pthread_exit()`.

Programmet som skapat tråden bör någonstans i sin kod vänta på att tråden ska terminera. Det gör man med funktionen `pthread_join()`. Genom funktionerna `pthread_exit()` och `pthread_join()` har man också möjlighet att returnera data från tråden respektive ta emot returvärdet i den anropande funktionen.

```
1  void pthread_exit ( void *returnvalue );     /* returvärde skickas som void* */
2
3  int pthread_join ( pthread_t tid,           /* tråd-id */
4                     void **returnvalue );    /* referens till reserverad plats för returdata */
```

## 4.2 Exempel

Här följer några enkla programexempel med användning av trådar.

### 4.2.1 Exempel: Skapa en tråd

Här följer kod till ett enkelt exempelprogram som skapar en ny tråd.

```
1  #include <stdio.h>   // printf()
2  #include <pthread.h> // pthread types and functions
3
4
5  void* child()
6  {
7      printf("This is the child thread\n");
8  }
9
10
11 int main()
12 {
13     pthread_t thread;                  // struct for child-thread info
14                                        // spawn thread:
15     pthread_create(&thread,            //  the handle for it
16         NULL,                          //  its attributes
17         child,                         //  the function it should run
18         NULL );                        //  args to that function
19     printf("This is the parent (main) thread.\n");
20     pthread_join(thread, NULL);        // wait for child to finish
21
22     return 0;
23 }
```

### 4.2.2 Exempel: Trådfunktion med inparametrar

Ibland vill man ge en tråd inparametrar och ibland vill man även att en tråd ska returnera data. En tråd kan bara ta emot *en* parameter, och den ska vara av typen `void*`. Om det finns ett behov av att skicka flera datadelar som parametrar kan man göra en struct med dessa delar och skicka en pekare till denna struct typkonverterad till `void*`.

I följande exempelkod skickas tillgång till flera datadelar till en trådfunktion genom en pekare till en struct, som typkonverteras till en `void*`. I programmet rapporterar varje tråd hur lång tid det tog för systemet att skapa den. Antal trådar ges som argument till programmet när det startas.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <sys/time.h>
5
6  struct threadArgs {
7     unsigned int  id;
8     unsigned int numThreads;
9     struct timeval startTime;
10 };
11
12
13 void* child(void* params)
14 {
15    struct timeval endTime;
16    gettimeofday(&endTime, NULL);
17
18    struct threadArgs *args = (struct threadArgs*) params;
19    unsigned int childID = args->id;
20    unsigned int  numThreads = args->numThreads;
21    struct timeval startTime = args->startTime;
22
23    double creationTime = (endTime.tv_usec - startTime.tv_usec)/1000000.0;
24    printf("Greetings from child #%u of %u which took %f secs to create\n",
25          childID, numThreads, creationTime);
26
27    free(args);
28 }
29
30
31 unsigned int processCmdLine(int argc, char** argv)
32 {
33    unsigned int retValue = 0;
34
35    if (argc > 1)
36    {
37       retValue =  atoi(argv[1]);
38    }
39    return retValue;
40 }
41
```

```
42
43  int main(int argc, char** argv)
44  {
45      pthread_t* children;                // dynamic array of child threads
46      struct threadArgs* args;            // argument buffer
47      unsigned int numThreads = processCmdLine(argc, argv);  // get desired # of threads
48      children = malloc(numThreads * sizeof(pthread_t)); // allocate array of handles
49
50      for (unsigned int  id = 0; id < numThreads; id++)  // create threads
51      {
52          args = malloc(sizeof(struct threadArgs));
53          args->id = id;
54          args->numThreads = numThreads;
55          gettimeofday(&(args->startTime), NULL);
56          pthread_create(&(children[id]),   // our handle for the child
57                         NULL,              // attributes of the child
58                         child,     // the function it should run
59                         (void*)args);     // args to that function
60      }
61      printf("I am the parent (main) thread.\n");
62      for (unsigned int id = 0; id < numThreads; id++)
63      {
64          pthread_join(children[id], NULL );
65      }
66
67      free(children);                     // deallocate array
68      return 0;
69  }
```

### 4.2.3 Exempel: Trådfunktion med returvärde

Returvärde kan man skicka genom argumentet till funktionen `pthread_exit()`. Även det överförs som en `void*`. Värdet tas emot genom andra argumentet till funktionen `pthread_join()`. I följande exempel returnerar trådarna ett värde till `main()`. Varje tråd räknar ut och returnerar kvadraten till ett tal, sedan summeras alla returvärden till en resultatsumma. Antal trådar ges som argument till programmet när det startas.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4
5
6   void* child(void* arg)
7   {
8       unsigned int* childID = (int*)arg;
9       unsigned int *idSquared = malloc(sizeof(unsigned int));
10      *idSquared = *childID * *childID;
11      printf("Child #%u computed %u\n", *childID, *idSquared);
12      pthread_exit((void*)idSquared);
13  }
14
```

```
15
16  unsigned int processCommandLine(int argc, char** argv)
17  {
18      unsigned int retValue = 0;
19      if (argc >1)
20          retValue =  atoi(argv[1]);
21      return retValue;
22  }
23
24
25  int main(int argc, char** argv)
26  {
27      unsigned int sumSquares = 0;
28      void* childSquare = NULL;
29      unsigned int numThreads = processCommandLine(argc, argv);    // get desired # of threads
30      pthread_t* children = malloc(numThreads * sizeof(pthread_t));     // allocate dynamic array of handles
31      unsigned int* args = malloc(numThreads * sizeof(unsigned int));     //allocate args array
32      for (unsigned int id = 0; id < numThreads; id++)
33      {
34          args[id] = id + 1;
35          pthread_create(&(children[id]), NULL, child, (void*)&args[id]);
36      }
37      printf("The parent (main) thread computed %u.\n", sumSquares);
38      for (unsigned int id = 0; id < numThreads; id++)
39      {
40          pthread_join(children[id], &childSquare);
41          sumSquares += *(int*)childSquare;
42      }
43      printf("\nThe sum of the squares from  0 to %u is %u.\n\n", numThreads, sumSquares);
44      free(children);
45      free(childSquare);
46      free(args);
47      return 0;
48  }
```

### 4.3 Mutexvariabler och semaforer

Mutexvariabler ger möjlighet att skapa kritiska regioner. Benämningen mutex är en förkortning av *mutual exclusion* som betyder ömsesidig uteslutning.

Mutexvariabler används främst för att skydda delad data så att bara en tråd åt gången ska kunna accessa den. Observera att det bara är mutex-variabeln själv som egentligen är skyddad, och att det är programmerarens ansvar att skydda den delade datan mot inskonsistens genom riktig hantering av mutexvariablerna. Endast en tråd kan ”ta” mutexvariabeln och låsa den. Det är bara den tråd som låst variabeln som kan låsa upp den igen.

Mutexvariabeln ska låsas och låsas upp, alltid i den ordningen, av varje tråd som vill ha tillgång till den kritiska region som mutexvariabeln är avsedd att skydda. En mutex är ett enkelt lås. När en tråd låst den kan ingen annan tråd låsa upp den. Vid rätt användning för skapande av kritisk region betyder det att ingen annan tråd kan få access till den resurs

som låset skyddar förrän den låsande tråden låst upp den igen. Om någon tråd försöker låsa en mutexvarabel som är låst av en annan tråd sätts den att vänta tills mutexvariabeln är upplåst igen.

POSIX innehåller även semaforer som kan vara räknande, dvs de kan anta ett godtyckligt positivt heltalvärde, (datatypen `sem_t`). De används huvudsakligen för signalering mellan trådar eller processer för att synkronisera dem. Då kan en tråd/process utföra `sem_wait()` på semaforen och en annan signalera den med `sem_post()` (användbart för att exempelvis lösa producer-consumer-problemet).

### 4.3.1 Funktioner för mutexvariabler

En mutexvariabel deklareras med typen `pthread_mutex_t`. Innan den kan användas måste den också initieras med funktionen `pthread_mutex_init()`. När den inte ska användas mer förstörs den med funktionen `pthread_mutex_destroy()`.

En mutexvariabel låses genom att kalla på `pthread_mutex_lock()`. Om variabeln redan är låst blir den kallande tråden blockerad tills variabeln låsts upp av den tråd som låste den.

Funktionen `pthread_mutex_trylock()` är identisk med `pthread_mutex_lock()` så när som på att funktionen återvänder omedelbart med ett returvärde som är skilt från noll om mutexvariabeln redan är låst. Det gäller oavsett om variabeln är låst av samma tråd eller av en annan.

Funktionen `pthread_mutex_unlock()` låser upp mutexvariabeln om den är låst av tråden själv. När en tråd är klar med användandet av den skyddade datan måste den anropa funktionen `pthread_mutex_unlock()` om andra trådar ska kunna accessa den delade datan. Funktionen `pthread_mutex_unlock()` returnerar en felkod om mutexvariabeln inte är låst eller om den är låst av en annan tråd.

Det finns ingen magi i mutexvariabler. Det är helt och hållet programmerarens ansvar att se till att trådarna låser och låser upp mutexvariablerna på ett korrekt sätt.

```
1  int pthread_mutex_init ( pthread_mutex_t *mutex,      /* initiera mutexvariabel */
2                const pthread_mutexattr_t *attr );   /* ange attribut */
3
4  int pthread_mutex_destroy ( pthread_mutex_t *mutex ); /* förstör mutexvariabel */
5
6  int pthread_mutex_lock ( pthread_mutex_t *mutex );    /* lås mutex eller blockera tills den låses upp */
7
8  int pthread_mutex_trylock ( pthread_mutex_t *mutex ); /* returnerar direkt om variabeln är låst  */
9
10 int pthread_mutex_unlock ( pthread_mutex_t *mutex );  /* lås upp mutexvariabel */
```

Samtliga funktioner returnerar 0 vid lyckad exekvering, annars returneras en felkod.

### 4.3.2 Exempel: Simulering av bankkonto

I följande exempel väljer man hur många trådar som ska skapas via ett tal som ges som argument till programmet. Varje tråd gör 1000 insättningar (för trådar med udda id) eller 1000 uttag (för trådar med jämnt id). När alla trådar terminerat ska saldot vara 0, förutsatt att man skapat ett jämnt antal trådar. Här får man kapplöpning mellan trådar *(race condition)* om inte uppdateringen av saldot skyddas av en mutexvariabel. Provkör gärna programmet några gånger med t ex 4 respektive 8 trådar och notera resultaten. Tag sedan bort kommentarerna kring anropen till funktionerna som hanterar mutexvariabeln `lock` i funktionerna `withdraw()` och `deposit()`. Provkör edan igen och notera skillnaden.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  // Shared Variables
6  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
7  double bankAccountBalance = 0;
8
9
10 void deposit(double amount)
11 {
12    //pthread_mutex_lock(&lock);
13    bankAccountBalance += amount;
14    //pthread_mutex_unlock(&lock);
15 }
16
17
18 void withdraw(double amount)
19 {
20    //pthread_mutex_lock(&lock);
21    bankAccountBalance -= amount;
22    //pthread_mutex_unlock(&lock);
23 }
24
25 // utility function to identify even-odd numbers
26 unsigned odd(unsigned long num)
27 {
28    return num % 2;
29 }
30
31 // simulate id performing 1000 transactions
32 void do1000Transactions(unsigned long id)
33 {
34    for (int i = 0; i < 1000; i++)
35    {
36       if (odd(id))
37          deposit(100.00);   // odd threads deposit
38       else
39          withdraw(100.00);  // even threads withdraw
40    }
41 }
42
```

```
43
44  void* child(void* buf)
45  {
46     unsigned long childID = (unsigned long)buf;
47     do1000Transactions(childID);
48     return NULL;
49  }
50
51
52  unsigned int processCommandLine(int argc, char** argv)
53  {
54     unsigned int retValue = 1;
55     if (argc > 1)
56     {
57        retValue =  atoi(argv[1]);
58     }
59     return retValue;
60  }
61
62
63  int main(int argc, char** argv)
64  {
65     pthread_t *children;
66     unsigned long id = 0;
67     unsigned long nThreads = 0;
68
69     nThreads = processCommandLine(argc, argv);
70
71     children = malloc( nThreads * sizeof(pthread_t) );
72
73     for (id = 1; id < nThreads; id++)
74     {
75        pthread_create(&(children[id-1]), NULL, child, (void*)id);
76     }
77     do1000Transactions(0);  // main thread work (id=0)
78
79     for (id = 1; id < nThreads; id++)
80     {
81        pthread_join(children[id-1], NULL);
82     }
83
84     printf("\nThe final account balance with %lu threads is $%.2f.\n\n",
85          nThreads, bankAccountBalance);
86
87     free(children);
88     pthread_mutex_destroy(&lock);
89     return 0;
90  }
```

### 4.4 Trådsäker kod

En reentrant funktion är en funktion som kan anropas från flera trådar när som helst utan att programmet kraschar, men bara för att en funktion är reentrant behöver den inte vara trådsäker. En trådsäker funktion genererar garanterat rätt resultat även om den kallas från flera trådar. Kontrollera att alla funktioner en tråd anropar är trådsäkra. Om de inte är det blir exekveringen av dessa funktioner kritiska sektioner som måste skyddas av mutexvariabler.

## 5 Redovisning

Laborationsuppgiften görs lämpligen i grupper om två. Annan gruppstorlek ska beviljas av labbhandledaren. Grupper med fler än tre deltagare godtas inte.

Redovisningen sker skriftligt genom att ladda upp koden för uppgift 6.4 på lärplattformen som en arkivfil (.zip eller .tar) samt genom muntlig diskussion och demonstration med handledare.

## 6 Laborationsuppgift

Vi kommer att arbeta med ett program som utför ordfrekvensanalys av en textfil. Det finns en arkivfil med hjälpfiler på lärplattformen. Den innehåller källkoden till en sekvensiell implementation av frekvensanalysen och källkoden för de små exempelprogram som visas i det här dokumentet där finns även en script-fil som heter `repeatfile.sh` och en `Makefile`.

Kompilering och länkning av programmen i laborationen görs med nedanstående kommando eller med make-filen som finns bland hjälpfilerna.

```
1 gcc -std=gnu99  -o <exe-fil>  <c-fil> -lpthread -lm
```

### 6.1 Sekventiellt program

Den givna programkoden innehåller en sekventiell implementation av frekvensanalysen. Att skriva applikationer helt sekventiellt kan ha sina baksidor. Anta till exempel att vi vill implementera en applikation i form av en browser för webbsidor. För att vi ska tycka att den är bra att använda vill vi inte att den ska sakna responsivitet i gränssnittet under tiden som den laddar information från nätet. Om applikationen ska ladda in stora datamängder från exempelvis bilder tar det ju ganska lång tid. I vårt exempelprogram nöjer vi oss med att anropa en funktion `userInteracion()` som läser tecken från tangentbordet och ekar dem på skärmen.

Det finns några test-texter av varierande storlek på lärplattfomen som kan användas vid testkörning. I arkivfilen innehållande laborationens hjälpfiler finns också ett script som duplicerar innehållet från en fil önskat antal gånger i för att öka filstorleken utan att behöva

belasta nätverket onödigt mycket. Testkör programmet med en stor fil, minst 300MB. Under körning av analysprogrammet matas namnet på textfilen som analysen ska köras på in manuellt.

Iaktta och kommentera programmets beteende!
Kan vi interagera med programmet samtidigt som analysen sker?

### 6.2 Skapa en tråd

För att kunna interagera med programmet medan beräkningar pågår ska vi nu låta interaktionsfunktionen exekvera i en egen tråd. Funktioner för trådexekvering ska enligt POSIX API returnera typen `void*`, så funktionen `userInteraction()` får fixas till lite. Titta noga på exemplet i 4.2.1!

Gör sedan om det givna frekvensanalysprogrammet så att funktionen `userInteraaction()` får exekveras i en egen tråd. Kan du interagera med programmet nu under analysen? Jämför körningen med föregående deluppgift!

### 6.3 Skapa tråd för frekvensanalysen

Nu ska även en ny tråd för att exekvera funktionen `frequencyAnalysis()` för frekvensanalysen skapas. Det betyder att funktionen måste returnera typen `void*`. Den här funktionen har dessutom parametrar. Hur den hantering kan se ut ges av exempel i avsnitt 4.2.2

Modifiera nu frekvensanalysprogrammet så att analysen av filen körs i en ny tråd som också den skapas i `main()` vid sidan av interaktionstråden!

### 6.4 Parallellisera analysen

Om datorns hårdvara har flera processorkärnor kan man ibland nå bättre prestanda med flera trådar. Här ska vi parallellisera frekvensanalysen så att den går snabbare. Låt flera trådar läsa och analysera var sin del av filens innehåll och skriva in sitt analysresultat i den globala tabellen. Tänk på att det måste finnas en allokerad plats för argument-struct för varje tråd. När man har många trådar som kör samma kod är det enklast att placera argumentdata i arrayer. Tänk också på att alla funktioner som tråden anropar måste vara trådsäkra, och att skrivningen av en globala tabellen måste vara ömsesidigt uteslutande. Analysresultatet ska bli detsamma oavsett om den körs på en tråd eller flera. (Du får bortse ifrån att ord kan komma att delas vid uppdelning av filen, då det marginellt påverkar resultatet.) Kör analysen upprepade gånger med olika antal trådar, exempelvis 1, 2, 4 och 8 stycken på samma textfil. Filen bör ha en storlek på minst 300MB vid mätningarna. Dokumentera och jämför resultaten!

**Extrauppgift**: Experimentera gärna med olika sätt för trådarna att skriva till den globala tabellen jämför prestandan mellan de olika varianterna. Man kan till exempel definiera lås för hela tabellen eller för olika delar av den. Man kan välja att lägga in data i den efter varje ord, att lägga in större block av delresultat efterhand eller att sammanfoga lägga in allt när delanalysen är klar.

# A Pthreads datatyper och funktionsdeklarationer

De flesta datatyperna, undantaget `pthread_t` som är en omdefiniering av en teckenlös heltalstyp, är structar innehållande flera olika delar. Dessa delar ska aldrig manipuleras direkt, utan bara genom APIets funktioner. Några av dessa datatyper är `pthread_attr_t`, `pthread_mutex_t` och `pthread_mutexattr_t`.

Här följer några av de funktionsdeklarationer som APIet innehåller som kan vara bra att känna till vid arbetet med laborationen.

```
1  int pthread_init(pthread_attr_t*);
2  int pthread_create(pthread_t*, pthread_attr_t*, void* (*func)(void*), void*);
3  int pthread_join(pthread_t tid, void** return_value);
4  void pthread_exit(void* return_value);
5  int pthread_yield(void);
6  pthread_t pthread_self(void);
7  int pthread_equal(pthread_t tid1, pthread_t tid2);
8
9  int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
10 int pthread_mutex_lock(pthread_mutex_t *mutex);
11 int pthread_mutex_unlock(pthread_mutex_t *mutex);
12 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

För mer information om POSIX Threads, studera Linux man pages.
Det finns också många bra tutorials på nätet.