Del 3 av kursen: Trådar mm. Möte 13 (Föreläsning, FL3): Trådar och processer, synkronisering

Inledning

Ett datorsystem har vi sett kan innehålla parallella processer och parallella processer representerar parallella skeenden; saker sker *samtidigt*. Vi ska nu se att även en process kan indelas i finare grader av parallella skeenden: så kallade *trådar.*

När vi startar en ny process så har den en egen adressrymd, skild från förälderns adressrymd. Om vi har samma maskinkod som den process som skapar processen (med `fork()`) betyder det (bland annat) att den nya processen får en egen uppsättning av alla variabler som den skapande processen hade. När vi skapar en tråd, däremot, så skapar vi ytterligare ett parallellt skeende (precis som när vi skapar en ny process), MEN detta parallella skeende (som kallas *tråd*) får ***inte*** en egen adressrymd. Således kan vi vara frestade att säga att ”trådar som skapats i en process har gemensamma variabler med processen”. Det är dock en olycklig formulering, det finns en begreppsförvirring i den, därför, innan vi forsätter är det dags att precisera vår terminologi:

En ***tråd***: är en aktiv del av en process och kan ses som en stack + en uppsättning register inklusive programräknare som kan köra processens kod. Det finns alltid minst en tråd i en process så när vi skapar en ny tråd har vi alltså 2 trådar i samma process. En tråd är en aktiv del av en process så varje tråd har en uppsättning processorregister och en stack.

En ***process***: har en egen adressrymd (data och instruktioner) + ett antal trådar (minst en).

En process *måste* alltså alltid ha minst en tråd (för att över huvudtaget kunna utföra någonting) och när en process skapar en ny tråd så får vi precisera oss lite och säga att det är egentligen en tråd som skapar en ny tråd. Man kan tala om en huvudtråd i en process som symboliseras av `main()`-slingan i ett *C*-program, men det ger inte så mycket. Tidigare har vi talat om att "en process skapar en ny process med `fork()`", men det är egentligen bättre att säga att "huvudtråden i en process skapar en ny process med `fork()`", men det blir lite tungt att formulera sig så. *Extra trådar inom en process är alltså extra parallella skeenden inom processen.*

Från Yolinux (Tutorial om p-trådar) (bra komplement till kurslitteraturen, ALP):

Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.

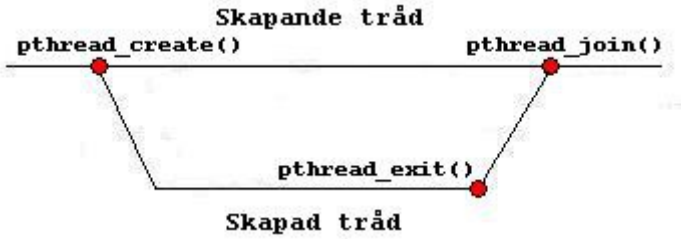
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
    - Process instructions
    - Most data
    - open files (descriptors)
    - signals and signal handlers
    - current working directory
    - User and group id
- Each thread has a unique:
    - Thread ID
    - set of registers, stack pointer
    - stack for local variables, return addresses
    - signal mask
    - priority
    - Return value: `errno`
- pthread functions return "0" if OK.

P-trådar

Den typ av trådar vi ska studera kallas p-trådar (*p-threads*), vilket är en förkortning för POSIX-trådar. För att skapa och hantera p-trådar kan vi lita lite på vår känsla från parallella processer: När vi skapade parallella processer behövde vi skapa, låta en process köra klart och sedan invänta den. P-trådsbiblioteket har något liknande. Motsvarigheterna är som följer:

Processer	*P-trådar*
`fork()`	`pthread_create()`
`exit()`	`pthread_exit()`
`wait()`	`pthread_join()`

Vi kan rita tidsdiagram även för trådar:



Vi ser här klart hur `fork()` motsvaras av `pthread_create()`, `exit()` av `pthread_exit()` och `wait()` av `pthread_join()`.

Vi ser på ett exempel från boken: (Sidan 64-66.)

```
#include <pthread.h>
#include <stdio.h>

/* Parameters to print_function.  */
struct char_print_parms
{
  /* The character to print.  */
  char character;
  /* The number of times to print it.  */
  int count;
};

/* Prints a number of characters to stderr, as given by PARAMETERS,
   which is a pointer to a struct char_print_parms.  */

void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}
/* The main program.  */

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
```

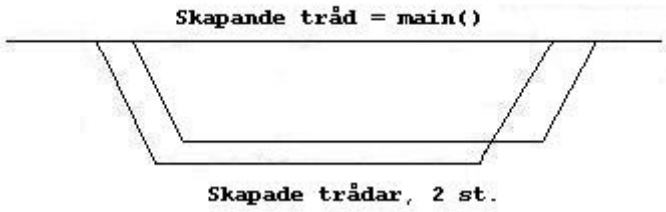
```
  /* Create a new thread to print 30000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

Detta program skapar två trådar som vardera skriver ut en massa '`o`' respektive '`x`' på skärmen. Ett tidsdiagram skulle kunna se ut så här:



Vi ser flera intressanta saker: en tråd körs genom att man skriver en trådfunktion, denna funktions namn skickas som parameter till `pthread_create()`, vi kan skicka parametrar till en tråd genom structen "`struct char_print_parms`". Detta måste dock ske genom en typlös parameter (`void*`) som senare får castas (`struct char_print_parms* p = (struct char_print_parms*) parameters;`). Dessa saker får ni undersöka exakt hur de går till på övningen.

Synkronisering av parallella skeenden

Vi ska nu introducera ett viktigt problemområde inom operativsystemteorin och datorteknik i allmänhet: *synkronisering*. Om vi har flera processer eller trådar som ska göra saker samtidigt så uppstår behovet av kontroll. Kontrollen måste se till att de delade/gemensamma resurserna/datana som trådarna/processerna behandlar inte tappar sin integritet/giltighet. Problematiken kan ofta beskrivas genom att studera följande situation. Process 1 och Process 2 har tillgång till variabeln `a`. Process 1 vill att `a` ska vara `10` och utför därför tilldelningen `a = 10;`. MEN samtidigt vill process 2 att variablen `a` ska ha värdet `20` och utför därför tilldelningen `a = 20;`. Om nu process 1 litar på att `a` har värdet 10 så kommer process 1 kanske att krascha, likadant är fallet för process 2. Problemet kallas kapplöpning "*race condition*" och uppkommer då det saknas kontroll av hur gemensamma/delade data hanteras. Om kapplöpning uppkommer kring en delad resurs finns inget sätt att veta säkert vad resultatet blir. Ovan skulle det inte finnas något sätt av veta vilket värde variabeln `a` hade. Det skulle kunna vara vilket som helst av `10` eller `20`.

Ett par punkter:

1. Flera processer kan **läsa** delade data samtidigt. Problem uppstår om flera processer vill **modifiera** delade data samtidigt. 2. Är samtidig modifikation förnuftigt? I databaser: JA! i Loggfiler: NEJ!

Serialisering

Att säkerställa att en sak sker efter en annan kallas *serialisering*. Att serialisera två processers tillgång till en gemensam resurs/data innebär att se till att de har tillgång till resursen efter varandra. (En i taget.) Vi introducerar något som vi kallas **kritisk handling** som är något som kan leda till kollisioner. Att tilldela variabeln `a` ovan värdet `10` (eller `20`) var en kritisk handling eftersom `a` var en gemensam variabel. Vi ska se på några vanliga strategier/tekniker för serialisering:

Mutexar

Ordet *MutEx* är en förkortning för *Mutual Exclusion* = Ömsesidig uteslutning. Det är detta vi vill uppnå: två processer (trådar) ska vara ömsesidigt uteslutna från att göra en kritisk handling samtidigt. Detta betyder att om process 1 vill göra en kritisk handling så ska inte process 2 kunna göra den. Ömsesidigheten betyder då att samma sak ska gälla fast tvärt om: om process 2 vill utföra en kritisk handling så ska process 1 inte kunna göra den. En MutEx (hädanefter *mutex*) är ett lås med vilket man kan låsa en gemensam resurs för att uppnå denna uteslutning. Vi skriver så här:

```
Get_Mutex(m);
Utför X;
Release_Mutex(m);
```

Här kallas mutexen `m` och den kritiska handlingen kallas `X`. Om två processer försöker utföra samtidigt `Get_Mutex(m);` så kommer den ena att bli väntandes och den andra kommer att "lyckas" låsa `m` och får komma vidare för att utföra `X`. När sedan den som kom vidare utfört `X` och låser upp `m` med `Release_Mutex(m);` så släpps den andra processen förbi `Get_Mutex(m);` och kan i sin tur utföra `X`. Nu har vi serialiserat processerna och beroende på vem som utförde `X` först/sist kan vi veta vad som hänt.

Kritiska sektioner: lösning med mutexar

En **kritisk sektion** i ett program (eller process eller tråd) är ett helt kodavsnitt där det är nödvändigt för processen/tråden att ha exklusiv tillgång till delade data. Bara en tråd/process får lov att vara i sin kritiska sektion åt gången. (Det är klart att en kritisk sektion gäller någon speciell resurs och noggrannare taget kan flera processer/trådar vara i sina kritiska sektioner samtidigt om det gäller olika områden/resurser.)

Problem med samtidig tillgång till gemensamma data yttrar sig när schedulern (OS-schemaläggare) tar CPUn från en process/tråd till en annan. Man skulle kunna stänga av schedulern när en process/tråd befinner sig i sin kritiska sektion, men det är alldeles för strängt och inte helt riskfritt (oändliga loopar kan inträffa och då låses hela systemet.) Vad man gör för att implementera kritiska sektioner är att använda mutexar, precis som vid kritiska handlingar, så här:

```
Get_Mutex(m);
|---------------|
|Kritisk sektion|
|...            |
|---------------|
Release_Mutex(m);
```

Detta blir formen för alla parallella processer/trådar.

Synkronisering av p-trådar

Vi ska nu återvända till p-trådar och se på hur mutexar med mera fungerar. Följande synkroniseringsmöjligheter finns för p-trådar:

* Join
* Mutexar
* Condition Variables
* Semaforer

Vi ska se på de första två, de andra två lämnas till självstudier.

Join

Anropet `join()` kan göras av en tråd och är ett sätt för en tråd att invänta att en annan tråd avslutar. Anropet ser ut så här: `pthread_join(thread_id, &ptr);` där `thread_id` är ett id för den tråd som ska inväntas och `ptr` är en pekare till `void` där ett returvärde kan lagras. Vi kan sätta pekaren till `NULL` i början, vi behöver inte fördjupa oss i det nu. Det här är lite liknande `wait()` för processer med skillnaden att man kan invänta trådar som man inte skapat. Det betyder att vi inte har samma tydliga släktförhållanden mellan trådar som råder mellan processer (förälder/barn etc.)

Mutexar

Vi deklarerar mutexar för ptrådar så här: `pthread_mutex_t m;` Efter man deklarerat den måste den få ett första värde, det finns initieringsrutiner men det är vanligare att man använder en fördefinerad konstant i i ett initieringsförfarande i samband med deklarationen. Då ser deklarationen ut så här:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Efter dessa manövrar så har vi en mutex med namnet `m` och vi kan sedan låsa den (eller vänta på) den respektive låsa upp den med anropen `pthread_mutex_lock()` (låsa/vänta) och `pthread_mutex_unlock()` (låsa upp). Vi ska se på ett exempel från `www.yolinux.com`:

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;
   /* Create independent threads each of which will execute functionC */
   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }
   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }
```

```
   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
   exit(0);
}
void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Här skapas en mutex som fantasilöst nog heter `mutex1`, den ska skydda variabeln `counter`. Två trådar, `thread1` och `thread2`, som båda kör funktionen `functionC()`, ökar på värdet av `counter`. Mutexen `mutex1` säkerställer att de båda trådarna gör detta i en kritisk sektion. Utan den kritiska sektionen skulle det teoretiskt kunna vara så att tråd 1 ökar på `counter` från 0 till 1 och blir avbruten av att schedulern lämnar över kontrollen till tråd 2 som också ökar på `counter` till 2. Sedan skriver tråd 2 ut värdet på `counter` (som är 2) och sedan lämnas kontrollen tillbaka till tråd 1 som skriver ut värdet på `counter` (som är 2). Resultatet blir att två 2:or kommer ut när vi egentligen vill se följande utskrift:

```
Counter value: 1
Counter value: 2
```

Ett annat mer typiskt exempel är vid transaktioner mellan bankkonton. Om en tråd hanterar pengar på konton och vill göra ta ut 1000 kronor, så här:

```
balance_at_account[123] = balance_at_account[123] - 1000;
```

så kan det hända att tråden endast hinner läsa vad som finns på kontot, det vill säga värdet på uttrycket `balance_at_account[123] - 1000;` etableras, det kanske är låt oss säga 2000 kronor, men skrivning av detta värde hinner inte ske, en annan tråd kliver in och tar ut, låt oss säga 100 kronor. Den tråden hinner dock fullborda sin transaktion. Kontrollen lämnas sen till sist över till den första tråden som nu gör tilldelningen `balance_at_account[123] =` det värde som etablerades tidigare = 2000 kronor. Resultatet blir att den andra trådens transaktion (100 kronor ut) aldrig noterades! Ännu värre blir det vid transaktioner då trådar behöver göra saker av typen

```
balance_at_account[123] = balance_at_account[123] - move_amount;
balance_at_account[234] = balance_at_account[234] + move_amount;
```

Alltså måste detta ske i en kritisk sektion. Koden måste då se ut ungefär så här:

```
pthread_mutex_lock(&account_mutex[123]);
pthread_mutex_lock(&account_mutex[234]);
balance_at_account[123] = balance_at_account[123] - move_amount;
balance_at_account[234] = balance_at_account[234] + move_amount;
pthread_mutex_unlock(&account_mutex[123]);
pthread_mutex_unlock(&account_mutex[234]);
```

Låsning - Deadlock

Ovanstående exempel leder in oss på begreppet *låsning*, eller *deadlock*, som det kallas på engelska. När flera parallella trådar delar på resurser/data och väntar på varandra kan det hända att de väntar cirkulärt: tråd 1 väntar på tråd 2 och tråd 2 väntar ÅTERIGEN på tråd 1. Båda trådarna är då låsta. Vi kan illustrera det i ett exempel:

Antag att både tråd 1 och tråd 2 vill utföra transaktioner mellan kontona 123 och 234. Tråd 1 vill utföra:

```
1: pthread_mutex_lock(&account_mutex[123]);
1: pthread_mutex_lock(&account_mutex[234]);
1: balance_at_account[123] = balance_at_account[123] - 100;
1: balance_at_account[234] = balance_at_account[234] + 100;
1: pthread_mutex_unlock(&account_mutex[123]);
1: pthread_mutex_unlock(&account_mutex[234]);
```

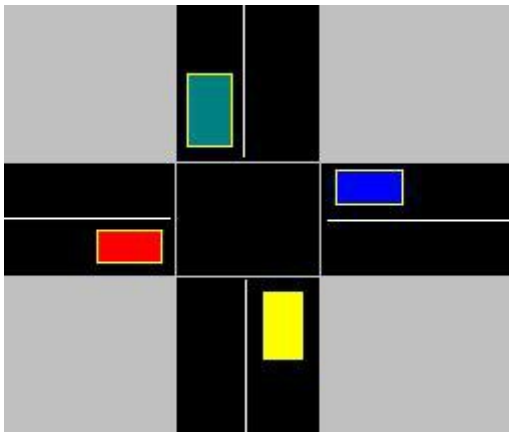
alltså flytta 100 kronor från konto 123 till 234 och tråd 2 vill utföra

```
2: pthread_mutex_lock(&account_mutex[234]);
2: pthread_mutex_lock(&account_mutex[123]);
2: balance_at_account[123] = balance_at_account[234] - 200;
2: balance_at_account[234] = balance_at_account[123] + 200;
2: pthread_mutex_unlock(&account_mutex[234]);
2: pthread_mutex_unlock(&account_mutex[123]);
```

alltså flytta 200 kronor från konto 234 till 123. (1:orna och 2:orna är bara för att markera att det är tråd 1 eller 2 som utför de aktuella operationerna.) OM schedulern nu schemalägger dessa trådar så att de operationer som utförs ser ut så här (i tiden):

```
1: pthread_mutex_lock(&account_mutex[123]);
2: pthread_mutex_lock(&account_mutex[234]);
1: pthread_mutex_lock(&account_mutex[234]);(Här börjar tråd 1 vänta på tråd 2.)
2: pthread_mutex_lock(&account_mutex[123]);(Här börjar tråd 2 vänta på tråd 1.)
```

så har de båda trådarna hamnat i en låsning. Tråd 1 (som låst konto 123) väntar på att tråd 2 ska släppa konto 234 så att tråd 1 kan köra klart. Men detta sker aldrig eftersom tråd 2 (som låst konto 234) väntar på att tråd 1 ska släppa konto 123. Tråd 1 väntar på tråd 2 som i sin tur väntar på tråd 1. En liknande situation kan uppkomma om fyra bilar samtidigt kommer till en fyrvägskorsning:



Alla bilar måste här stanna eftersom högerregeln säger att man måste ge företräde till den bil som kommer från höger. Men samtliga bilar har en bil som kommer från höger, samtliga bilar måste därför vänta. En annan typ av låsning. (Eller är det egentligen en annan typ av låsning?)

Nödvändiga villkor för låsning

Av bilexemplet och transaktionsexemplet kan vi se en gemensam egenskap: det handlar om cykler... Om bilarna heter A, B, C och D så har vi att A väntar på B som väntar på C som väntar på D som ÅTERIGEN väntar på A. I exemplet med transaktionerna ovan så väntar tråd 1 på tråd 2 som ÅTERIGEN väntar på tråd 1. Om det skulle finnas ett sätt att upptäcka om trådar/processer väntar på varandra så skulle låsning kunna upptäckas. Vi kan då formulera ett nödvändigt villkor för att låsning ska kunna uppstå:

Cyklisk väntan (cyclical waiting): För att låsning ska kunna uppstå måste processer/trådar kunna vänta på varandra cykliskt. Om n trådar, t1, t2, ... tn är inblandade i låsning ska alltså t1 vänta på t2, t2 vänta på t3 och så vidare till t(n-1) som väntar på tn och tn väntar återigen på t1.

Det finns vidare två till villkor som är nödvändiga för att låsning ska kunna uppkomma:

Inget återtagande av resurser (no pre-emption): När en process/tråd äger en resurs (exklusiv tillgång till data etc) så finns inget sätt för operativsystemet att återta (pre-empt) denna resurs.

Icke delbara resurser (non-sharable resources): De resurser som inkluderas i låsningen ska inte gå att dela (då behövde ju inte processerna/trådarna vänta på varandra.)

De sista två villkoren kan upplevas som självklara, men det kan vara värt att konstatera att de faktiskt utgör nödvändiga villkor. Genom att studera dessa tre villkor kan vi komma på sätt att förhindra låsning. Det finns tre alternativa sätt att närma sig problemet med låsning:

1. Förhindra låsning (*Prevention*)

Operativsystemet hanterar resurser på ett sätt som garanterat inte skapar låsning. Detta innebär att trådar/processer ibland kan nekas att vänta på en resurs eftersom det skulle kunna leda till en farlig situation. Detta inriktar sig på *cyklisk väntan*.

2. Återhämta sig från låsning (*Recovery*)

Operativsystemet tillåter låsning att uppkomma men då låsning uppkommit vidtar systemet åtgärder för att återhämta sig. Detta kan involvera återtagande av resurser (detta inriktar sig alltså på no pre-emption) men det är riskabelt att bara ta bort en resurs från en process/tråd.

3. Strunta i att låsning ibland uppkommer (Ostrich = struts = stoppa huvudet i sanden**)**

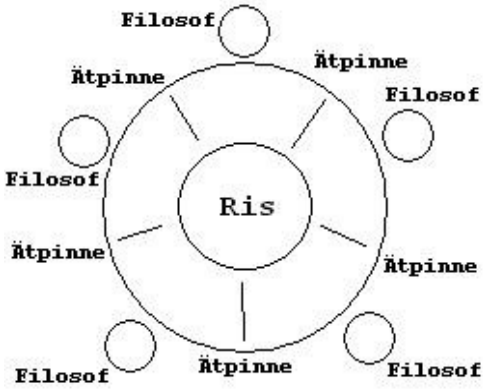
De flesta OS använder denna metod, som egentligen inte är en metod, det är bara att strunta i att låsning ibland uppkommer. Ansvaret läggs på programmerarna och användarna att skriva program som aldrig låser sig och i de sällsynta fall som de låser sig så får man avsluta programmen och starta om från början. Anledningen till att detta är den metod som mest används är att det är dyrt att ens upptäcka (*Detection*) att låsning uppkommit. Det är vidare dyrt att välja ut vilken av processerna som ska dö (vid *Recovery*) och det är dyrt att räkna en massa på vem som får vänta på vem (vid *Prevention*). Därför struntas det ofta i låsning.

Svält (Starvation)

I ett datorsystem/operativsystem finns parallella skeenden i form av (trådar/processer) och dessa parallella skeenden är ibland i behov av resurser. En resurs kan vara mycket, det kan vara delade data, en bit av minnet, en in/utmatningsenhet, en fil, etc. Då flera trådar måste samsas om begränsade resurser kan det ibland uppstå en situtaion då en tråd som försöker utföra någonting får

vänta så länge på det som den behöver att det inte längre är meningsfullt att vänta. Då har *Svält* uppkommit. Man säger att en tråd/process svälter. Ett exempel kan vara om en webbläsare får vänta väldigt länge på en upplysning om när en buss går och webbläsaren får vänta så länge att själva bussen hinner gå. En vanlig metafor är "de fem dinerande filosofernas problem".

De fem dinerande filosoferns problem (The Dining Philosopher's Problem)
Fem filosofer spenderar livet med att äta, tänka och sova. Alla aktiviteterna kostar näring. De sitter runt ett bord med en stor skål med ris i mitten. Mellan sig, på var sida om sig, ligger ätpinnar. Det finns precis fem ätpinnar och när en filosof tagit upp båda ätpinnarna och äter så kan de båda intillsittande filosoferna inte äta. Om alla filosoferna tar upp pinnen till höger om sig så kommer detta att resultera i att alla fem blir sittande väntandes på varandra.



Till slut kommer någon av dem att svälta ihjäl och då kan den filosof som sitter till vänter om den som svalt ihjäl ta upp ätpinnen som den döde lämnat efter sig och börja äta.

Detta problem (som först formulerades av den Nederländske datalogen Edsger Dijkstra) är ett klassiskt problem som beskriver mycket av synkroniseringsproblematik som uppkommer vid parallella processer/trådar. Det kommer att ligga till grund för huvuduppgiften i laboration 3.

Läs gärna på wikipedia om detta: *http://en.wikipedia.org/wiki/Dining_philosophers_problem*.

Mer kring trådar och processer

Vi ska nu poängtera relationer mellan trådar och processer. Det är dock *mycket* förrädiskt att i en textmening skilja på "process" och "tråd", det är ju så att varje process som köra har minst en körande tråd och en process som kör kan ha flera körande trådar. Man brukar då säga att en process är *flertrådad*. Således kan vi inte tala om *process* i motsats till *tråd* - de existerar *alltid* tillsammans.

Schemaläggninsaspekter rörande trådar och processer

När ett flerprocessoperativsystem kör så lämnas CPUn över från en process till process (*Round-robin*). Detta möjliggör/är multitasking och i själva överlämnandet finns mycket information som måste hanteras rätt. Innehållet i *Memory Management Unit* (*MMU*) ska bytas ut (för varje process har olika sidor i minnet att köra), filtabellen (*User FDT*) måste bytas ut och innehållet i CPUs register ska bytas ut. Det hela kallas *Context-Switch* och är ganska dyrt i tid räknat. Operativsystemet lägger en massa administrativ tid på det. Dock så är situationen en helt annan då det gäller växling mellan enskilda trådar *inom en och samma process*. Det som behöver bytas ut när CPU lämnas från en tråd till en annan är i princip endast innehållet i registrena + stackpekaren (varje tråd har ju en egen stack). Eftersom parallella trådar inom samma process delar samma adressrymd och maskinkod så behöver inga uppdateringar av sidtabellen eller filtabellen göras. Det betyder att om vi har parallella skeenden förlagda mellan ett antal trådar så sparas mycket tid

jämfört med om vi hade haft samma parallella skeenden förlagda mellan parallella processer. Det är till och med så att vissa författare inte ens kallar det en *Context-Switch* då CPU:n går från en tråd till en annan, men vi kan säga att "*Context-Switch*en vid överlämning av CPU från en tråd till en annan är väsentligt förkortad på grund av att allt som ska bytas ut är CPU-register + stackpekare. *MMU* och filtabell lämnas orörd." Viktigt är också att vid en context-switch mellan trådar behövs ingen inblandning av operativsystemet, switchen sker internt i den körande processen.

Förtjänster/särdrag med trådar

Om vi förlägger parallella aktiviteter i trådar istället för i egna processer (observera att det rent tekniskt är fel att säga så här!) så gäller följande:

Det blir mindre Context-switch

Vid en tentamen i operativsystemteknik för ett par år sedan ställdes frågan: "Ange en fördel med trådar jämfört med processer." En student svarade "Trådar är snabbare." Jag tycker att *både* frågan och svaret var ganska dåliga. För det första så är trådar och processer inte i motsats till varandra, de finns alltid båda två tillsammans som vi sett ovan. Vidare är svaret är väldigt oprecist: VARFÖR är trådar snabbare? Går datorn snabbare om man använder trådar? Varför det? På vilket sätt? Frågan och svaret pekar ändå på något viktigt: I och med att *context-switch* mellan trådar är så mycket mindre än om man jämför med *context-switch* mellan processer så spenderar datorn mindre tid med administration om de parallella aktiviteterna kan förläggas i parallella trådar istället för parallella processer. Det vore ett bättre svar på frågan ovan. En bättre formulering av frågan skulle kanske lyda: "Ett datorsystem ska utföra att antal parallella aktiviteter. Valet står mellan om dessa aktiviteter ska förläggas i varsinn process eller om de ska förläggas i ett antal parallella trådar inom samma process. Vad är skillnaden? Finns det fördelar/nackdelar? Motivera ditt svar."

Vi får en gemensam adressrymd

Ett annat drag som parallella trådar inom en och samma process har är att de har samma adressrymd. Det har vi nämnt förut, men jag nämner det här igen för fullständighetens skull. Det betyder att de globala variablerna i ett *C*-program som innehåller flera trådar alla delas mellan trådarna, detta undersöks även på övningen som också illustrerar hur man ska åstadkomma trådspecifika data som lokala variabler i de så kallade trådfunktionerna.

Mer aspekter för programmeraren

Att förlägga parallella aktiviteter i olika trådar ger för programmeraren större flexibilitet än om de förläggs i parallella processer eftersom det finns fler programmeringsmässiga styrmedel över parallelliteten. Varje tråd kan ges en egen prioritet, varje tråd kan vara mer eller mindre hårt knuten till programmet (*detached*/*non-detached*) och att den exekverar i samma adressrymd förenklar kommunikationen mellan de olika trådarna. Exempelvis kan man kommunicera direkt med `read()`/`write()` på en pipe som inte har dubbla läs och skrivändar som uppkommer vid `fork()`, vi slipper alltså stängningen av de extra läs- och skrivändarna som var nödvändiga vid parallella processer.

För att skapa parallella processer måste man göra `fork()` det resulterar i en helt ny process. För att skapa en ny tråd görs `pthread_create()` (eller liknande) och det resulterar endast i en ny tråd, väsentligt mindre nytt än vid en helt ny process. Trådar sparar således resurser. Detta blir även en tydlig besparing vid *context-switch* där ju (som vi nu säger for the zillionth time) det som behöver bytas ut i princip endast är CPUs register + stackpekare. (Kolla dock upp `vfork()`!)

Ett exempel på trådar är att i ordbehandlingsprogram som *Microsoft Word* eller *OpenOffice Writer* så kontrolleras det ord som sist matats in mot en rättstavningsdatabas.

Linux trådar

I *Advanced Linux Programming* (4.5) står det att trådar i *Linux* är implementerade "som processer". Det står så här:

"Whenever you call `pthread_create()` to create a new thread, Linux creates a process that runs that thread. However, this process is not the same as a process you would create with `fork()`; in particular, *it shares the same address space and resources as the original process* rather than receiving copies."

Jaha... hur ska man tolka det här då? Att "trådar i *Linux* är processer" låter ju som en begreppsförvirring, var det någon mening med att införa trådar i *Linux* över huvud taget då om de ändå blir parallella processer? Nja, om vi granskar texten lite noggrannt så ser vi att det står, om den skapade processen: "*it shares the same address space and resources as the original process*". Men frågan är ju då hur mycket av en **egen** process som den skapade processen är? Om den inte har en egen adressrymd!

Är det inte bäst att säga att "om två parallella skeenden har en egna åtskilda adressrymder är de olika processer" och "om två parallella skeenden delar på samma adressrymd så är de anses som två trådar inom samma process"...? Jag tolkar det så och kommer att prata som om detta gäller. Det verkar ju även här som att problemet ligger i *terminologi och definitioner*, det viktiga här är att i *Linux* finns en möjlighet att skapa parallella skeenden i olika adressrymder (parallella processer) eller låta flera parallella skeenden dela på samma adressrymd (parallella trådar). Att det sedan sägs att "I *Linux* är trådarna parallella processer" förvirrar saker och ting. Man kan inte se trådar med kommandot "`ps`" så att tala om trådar som att de vore processer skapar som sagt förvirring. Tveksamt språkbruk alltså! (Fy på er GNU! Men tack för att vi får ett gratis operativsystem!)

Vi ska nu titta lite på skelettet till laboration 3.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

char table[20] = "_W_W_W_W_W_";

/*Example of illustration of five philosphers with
five chopstick. Chopsticks are lying on the table*/

/*

_W_W|E|W_W_       This is when the third guy is eating.

_W|E|T_T_W_       This is when the second guy is eating.
                  third and fourth are thinking. (Philosophizing.)

_W|E|T|E|W_       This is when the second and fourth guys are eating.

|E|W_E_W_W|       This is when the first guy is eating.
                  To note: The first "|" symbolizes one chopstick to
                  the left of the first philosopher. But the last "|"
                  is also symbolizing the same chopstick since they
                  are sitting around a round table.
*/

struct phil_parms
{
  int pos;
  int noPhils;
};

void* philosophize (void* parameters) { /*Put philosopher-code here*/ return NULL; }
```

```
int main (int argc, char* argv[])
{

  pthread_t phils[9]; /*Some sort of array of phils are needed*/
  struct phil_parms control_phil[9];

  int i=0;
  int noPhils, lock;
  int round = 0;


  /*Overall design of the program

  1. Take in commandline arguments to set up how many phils are going to be
  simulated and if deadlock is going occur. Commandline arguments need to
  be checked and the program needs to exit if they are not in the correct format
  see Advanced Linux Programming for excellent advice on commandline arguments.

  2. Start simulation by starting the phil-threads and let the main program
  print out the contents of the string table declared above. No thread is going
  to communicate with the user but through the string table, it is the main
  program that prints out the contents of the string table. This means that
  we are separating the task of computation/simulation from the task of
  presentation of the results*/

  while(round<48)
  {
    printf("Round %2d: %s\n", round+1, table);
    sleep(1);
    round++;
  }

  /* The above loop runs in parallel to the threads/phils that affect the
  common resource table.

  IMPORTANT: The synchronization must not be through one mutex! We must have
  one mutex per chopstick, that means an array of mutexes is also needed!

  IMPORTANT: Remember that the program should also make deadlock possible
  through commandline arguments, there must be a way to force a deadlock to
  occur. Remember in this context that all thread-functions are to be based on
  one function, philosophize(), and that this function is the same for all
  threads.

  Of course it can behave differently for different philosopher-id's, but
  it must be one function which needs to be written to enable a forced deadlock
  so we can compare and understand why it normally avoids deadlock.

  3. When the loop has finished, all the threads are joined to the main program
  and then the main program exits.

  */

  return 0;
}
```

Körexempel:

```
Round  1: _W_W_W_W_W_
Round  2: _W_W_W_W_W_
Round  3: _W_W_W_W_W_
Round  4: _W_W_W_W_W_
...
Round 48: _W_W_W_W_W_
```

Programmet är tillgängligt på kurswebben.