TRÅDAR

TRÅDAR

- Ibland hör man ordet "lättviktsprocesser", med detta menar man att man har "processer" som exekverar inom en vanlig process.
- Trådarna har till skillnad från processer gemensamt minnesutrymme.
- Det är inte tillräckligt att trådarna kan utbyta information med varandra... man måste också kunna synkronisera dem så att den information man utbyter är korrekt.
- Detta leder till att man måste vara försiktig då man använder sig av globalt data då vi kan drabbas av problem om flera trådar försöker göra detta samtidigt.

TRÅDAR

- Ett exempel är pthreads (POSIX-trådar)
- En process kan innehålla flera trådar
  - Olika delar av samma kod kan exekveras samtidigt
- Trådarna bildar en process och delar adressrymden, filer, signalhantering mm. Exempelvis:
  - När en tråd öppnar en fil, är den omedelbart tillgänglig för alla trådar
  - När en tråd skriver till en global variabel, är den läsbar av alla trådar

PTHREADS

- Varje tråd har egen
  - instruktionsström
  - stack
  - registervärden
  - signalmask
  - errno-variable
  - scheduleringsprioritet
- Det är billigare att
  - skapa trådar än att forka nya processer
  - växla mellan trådar

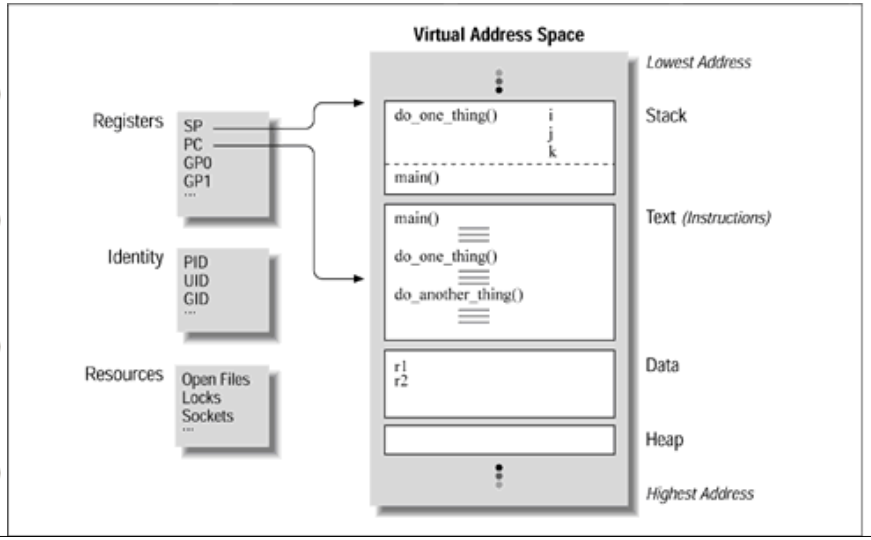
A SIMPLE C PROGRAM

```
int r1 = 0, r2 = 0;
main(void)    {
  do_one_thing(&r1);
  do_another_thing(&r2);
  do_wrap_up(r1, r2);
  return 0;
}
```

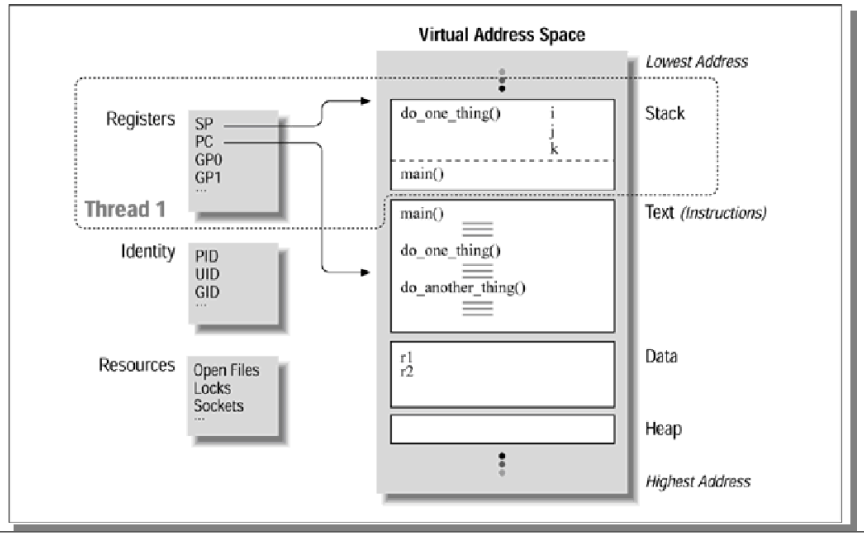
```
void
do_one_thing(int *x_times)
{
  int i, j, x;
  for (i = 0;  i < 4; i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000; j++)
      x = x + i;
    (*x_times)++;
  }
}
```

```
void
do_another_thing(int *x_times)
{
  int i, j, x;
  for (i = 0;  i < 4; i++) {
    printf("doing another");
    for (j = 0; j < 10000; j++)
      x = x + i;
    (*x_times)++;
  }
}
```

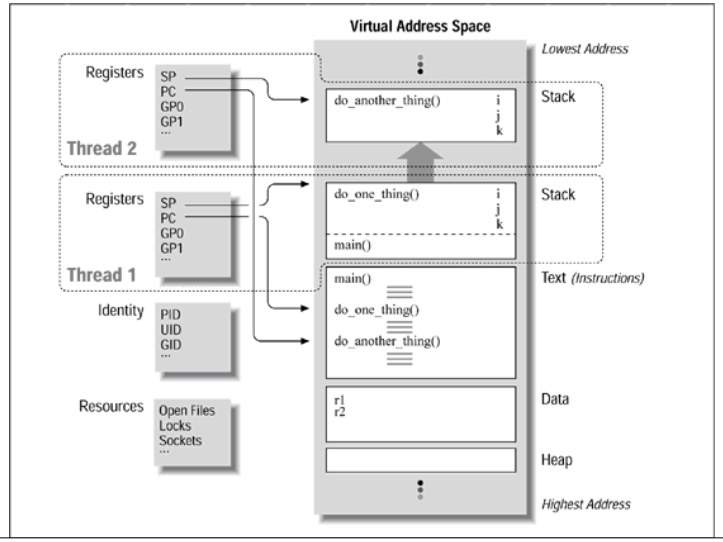
... SOM EN PROCESS



...SOM EN PROCESS MED EN TRÅD



...SOM EN PROCESS MED FLERA TRÅDAR



SKAPA EN TRÅD

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
- `pthread_create()` skapar en ny tråd inom en process.
- Den startar i rutinen `start_routine()` som tar ett startargument `arg`
- `attr` specar attribut (tex min stackstorlek), eller default-attribut om `NULL`
- Om `pthread_create`-rutinen lyckas, returneras 0 och den nya trådens ID läggs i `thread`, annars returneras en felkod.

AVSLUTA EN TRÅD

- `void pthread_exit(void * status);`
- Avslutar den för närvarande körande tråden och gör status tillgängligt till den tråd som joinar (som anropar `pthread_join()`) med den terminerande tråden.
- Ett annat sätt att avsluta: `return` i trådens startfunktion

HELLO WORLD ELLER WORLD HELLO?

```
void *print_message_function( void *ptr ) {
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
    return NULL;
}

int main(int argc,char **argv) {
    pthread_t thread1, thread2;
    char *message1 = "Hello", *message2 = "World";
    void *status1, *status2;
    pthread_create(&thread1, NULL, print_message_function,
                   (void*)message1);
    pthread_create(&thread2, NULL, print_message_function,
                   (void*)message2);
    pthread_join(thread1, &status1);
    pthread_join(thread2, &status2);
    return 0;
}
```

SYNKRONISERING AV TRÅDAR

- Synkronisering av trådarna kan bl a göras mha en mutex. För att åstadkomma detta har man en (eller flera) variabel av typen `pthread_mutex_t` i sitt program.
- En tråd kan sedan låsa denna mutex (om den redan är låst så kommer funktionen att vänta tills dess någon annan tråd låser upp den) mha funktionen `pthread_mutex_lock(pthread_mutex_t *)`
- När man är färdig med den delen i koden som krävde låset så anropar man `pthread_mutex_unlock(pthread_mutex_t *)` för att tillåta ev. andra trådar att låsa mutexen.
- Innan låset används så bör det initieras mha `phthread_mutex_init(…)`

DEADLOCK

- Man måste vara försiktig så att två trådar/processer inte hamnar en situation så att de båda väntar på att den andra ska släppa en resurs (t.ex. en mutex eller liknade). Om denna situation inträffar kommer de två (eller fler) trådarna/processerna aldrig att fortsätta.

DÖDLÄGE – NÖDVÄNDIGA VILLKOR

1. Ömsesidig uteslutning
   - Processer fordrar exklusiv kontroll över de resurser som begärs
2. Successiv tilldelning
   - Processer håller redan resurser medan de väntar på tillgång till andra resurser
3. Icke frigörbarhet
   - Resurser kan endast frigöras av den process som reserverat dessa
4. Cirkulär väntan
   - En circular kedja av processer existerar där varje process håller en resurs begärd av nästa process i kedjan

SYNKRONISERING

- Trådbiblioteket innehåller även mer avancerade metoder för synkronisering som ”read/write locks”, ”condition variables” och (semaforer)

CONDITION VARIABLER

- En condition variable (villkorsvariabel) är associerad med en händelse och har inget värde som vi kan undersöka.
- Till varje condition-variabel kopplar vi även en mutex-variabel
- Om en tråd måste vänta på att en händelse ska inträffa väntar den på motsvarande condition-variabel. Samtidigt som man börjar vänta låses mutexen upp (bör vara låst innan)
- Om en annan tråd orsakar att händelsen inträffar så signalerar den att händelsen har inträffat via condition-variabeln. Vilket väcker upp en ev. väntande tråd (och låser mutexen för den).
- Den uppvaknande tråden måste kolla villkoret (händelsen) och om den inte gäller återgå till att vänta

CONDITION VARIABLER – VANLIGASTE FUNKTIONERNA

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
  - Wait for a condition and lock the specified mutex.
- `int pthread_cond_signal(pthread_cond_t *cond)`
  - Unblock at least one of the threads blocked on the specified condition variable.
- `int pthread_cond_broadcast(pthread_cond_t *cond)`
  - Unblock all threads currently blocked on the specified condition variable.
- `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`
  - Wait no longer than the specified time for a condition and lock the specified mutex.

READ-WRITE LOCKS

- Read write locks tillåter en skrivare att låsa låset varvid ingen annan kan göra det.
- Flera "läsare" kan dock samtidigt ha låst variabeln.
- Kan förbättra prestandan mot att använda mutex då några trådar bara vill undersöka datat utan att ändra den

PTHREADS – DIVERSE

- `#include <pthread.h>` först i programmet
- Länka med `-lpthread`
- Kontrollera alltid felkoder!
- Tänk på att funktioner som t.ex. `exit, sleep` osv. påverkar hela processen (dvs. alla trådar). Vill ni endast påverka en tråd så finns det varianter av många av dessa funktioner som opererar på bara en tråd.
- Mer info:
  - man-sidor (en bra start är "`man pthread`" eller "`man pthreads`")

THREADSAFE

- Om en funktion "säkert" kan anropas av flera trådar samtidigt
  - Tex inte `getpwnam`, `getopt`
- Om inte beror det ofta på att funktionen returnerar data som lagrats I en statisk buffert (dvs samma address vid varje anrop)

TRÅDAR – SIGNALER

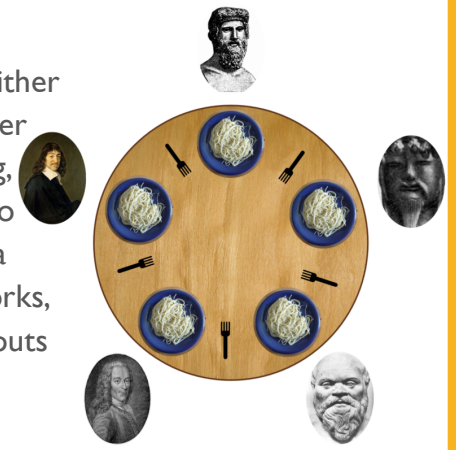
- Skickas till en specific tråd
  - Hårdvarurelaterade signaler skickas till orsakande tråd
  - Alla andra till slumpmässig tråd
  - Dvs signalhanteringen blir mer komplicerad

SIGNALER TILL TRÅDAR

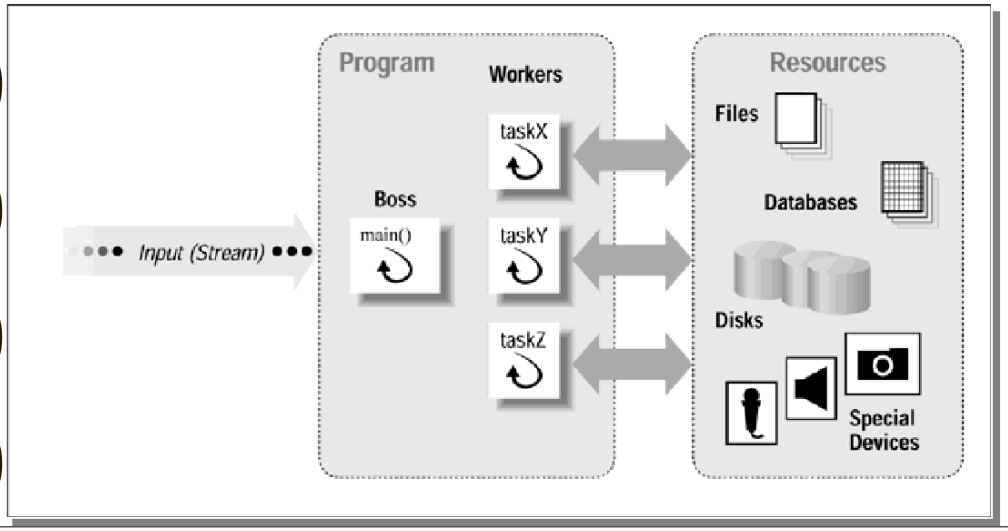
- `int pthread_kill(pthread_t thread, int sig)`
  - Skickar en signal till en tråd
- Funkar i övrigt ungefär som kill
- `int pthread_sigmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);`
  - Blockerar en signal för en tråd
  - Funkar i övrigt ungefär som sigmask

DINING PHILOSOPHERS PROBLEM

- Five philosophers are sitting around a circular table and each has a plate of spaghetti in front of him with a fork at either side. Suppose that the life of a philosopher consists of periods of eating and thinking, that each philosopher needs two forks to eat, and that forks are picked up one at a time. After successfully picking up two forks, a philosopher eats for a while and then puts down the forks and thinks. The problem consists in developing an **algorithm** to avoid **starvation** and **deadlock**.



BOSS/WORKER MODEL



BOSS/WORKER MODEL – PSEUDOCODE

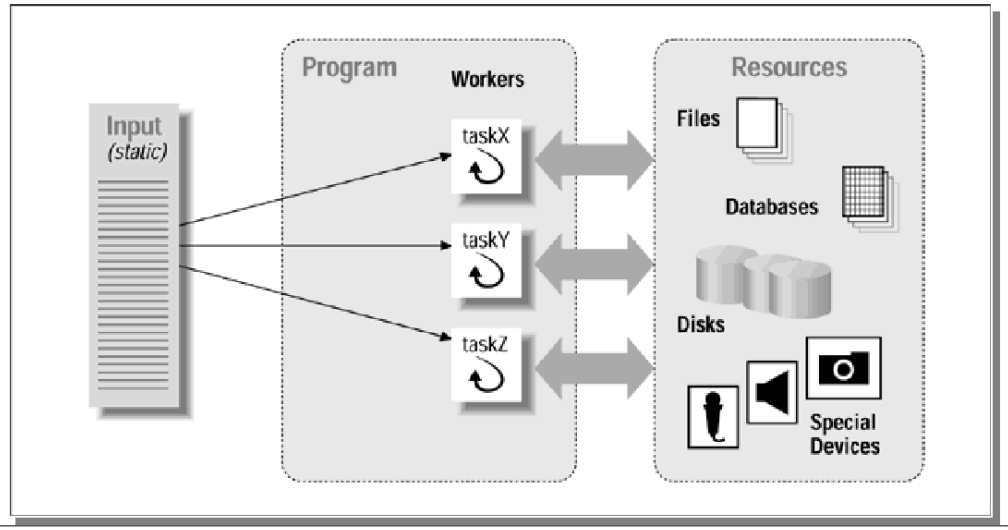
```
main()    /* The boss */
{
  forever {
    get a request
      switch request
        case X : pthread_create( ... taskX)
        case Y : pthread_create( ... taskY)
        ...
  }
}

taskX() /* Workers processing requests of type X */
{
  perform the task, synchronize if necessary
  done
}
```

BOSS/WORKER MODEL – THREAD POOL

```
main() {     /* The boss */
  for the number of workers
    pthread_create( ... pool_base )
  forever {
    get a request
    place request in work queue
    signal sleeping threads that work is
available
  }
}
```

```
pool_base() {  /* All workers */
  forever {
    sleep until awoken by boss
    dequeue a work request
    switch
      case request X: taskX()
      case request Y: taskY()
      ...
  }
}
```

PEER MODEL



PEER MODEL – PSEUDOCODE

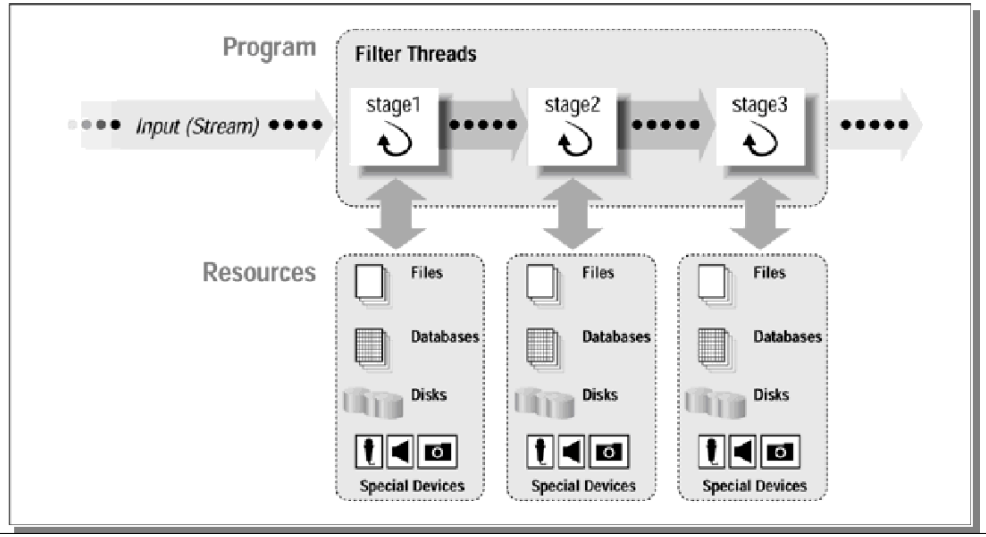
```
main()    {
  pthread_create( ... thread1 ... task1 )
  pthread_create( ... thread2 ... task2 )
  ...
  signal all workers to start
  wait for all workers to finish
  do any clean up
}
```

```
task1()   {
  wait for start
  perform task1
  synchronize if necessary
  done
}
```

```
task2()   {
  wait for start
  perform task2
  synchronize if necessary
  done
}
```

...

PIPELINE MODEL



PIPELINE MODEL – PSEUDOCODE

```
main() {
  pthread_create( ... stage1 )
  pthread_create( ... stage2 )
  ...
  wait for all pipeline threads to finish
  do any clean up
}
stage1()
  forever
     get next input for the program
     do stage 1 processing of the input
     pass result to next thread in pipeline
...
stageN()
  forever
     get input from previous thread in pipeline
     do stage N processing of the input
```