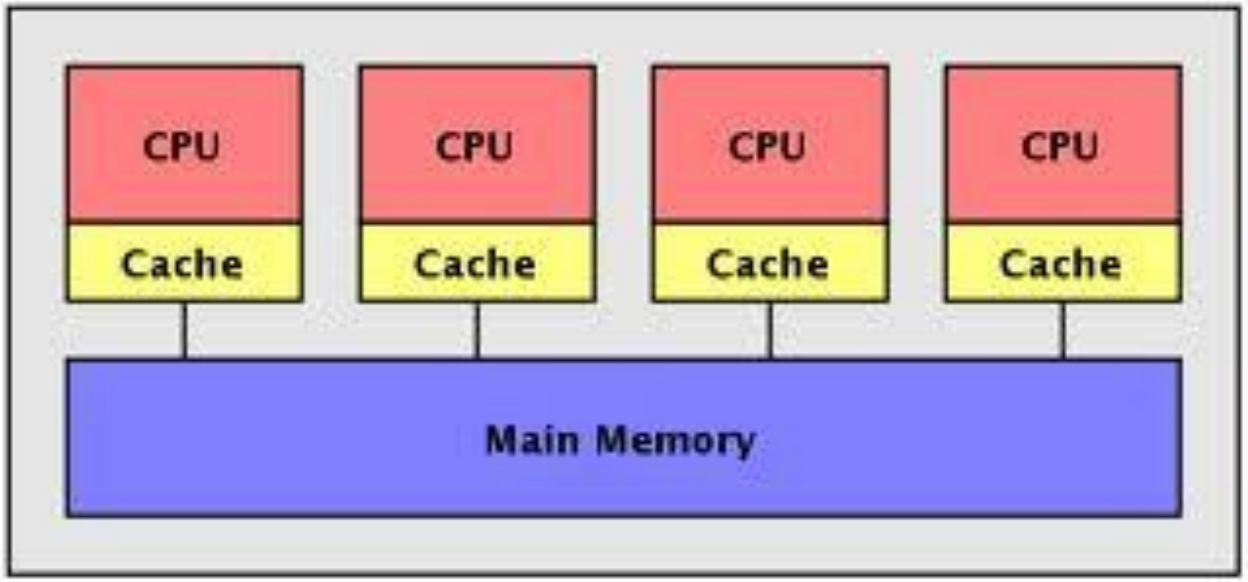
Porazdeljeni sistemi

3. Nitenje s knjižnico pThreads

Predavatelj: izr. prof. Uroš Lotrič
Asistent: Davor Sluga

Tesno sklopljeni večprocesorki sistemi



- Več CPE povezanih preko vodila s skupnim pomnilnikom
- Komunikacija poteka preko pomnilnika
- Do pomnilnika se dostopa izključno s pomočjo ukazov LOAD/STORE

Tesno sklopljeni večprocesorki sistemi

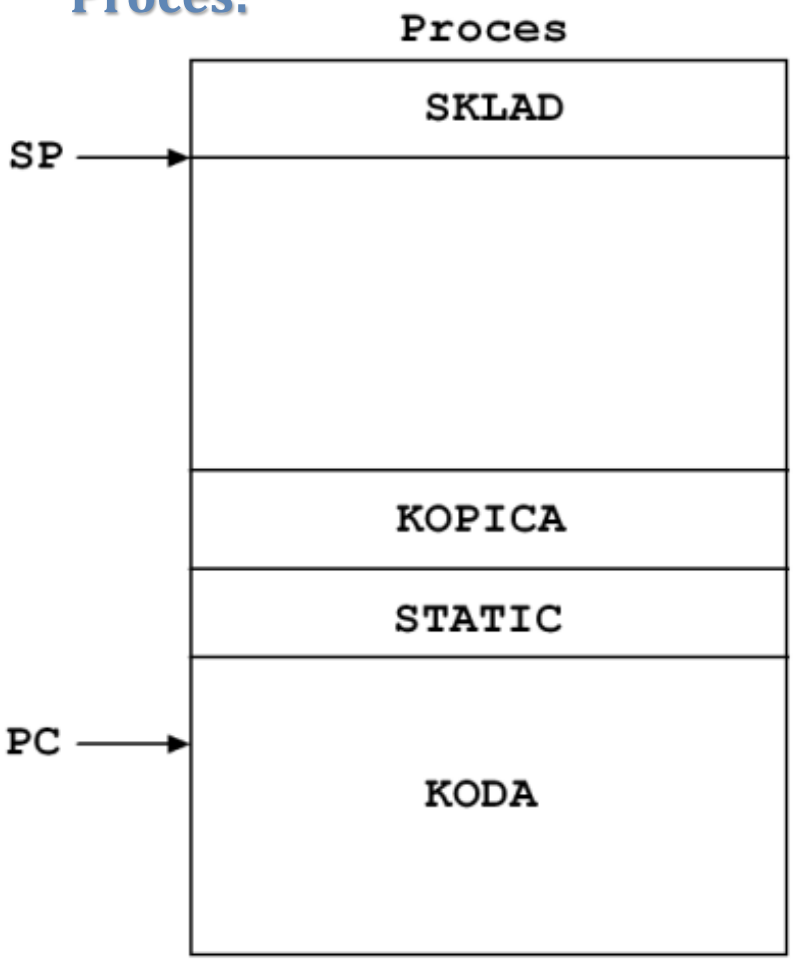
- Na takih sistemih lahko hkrati izvajamo več ločenih procesov
- Proces je v svoji osnovi izvajajoči se program
- Procese zna razvrščati in preklapljati OS
- OS določi kateri proces se bo v nekem trenutku izvajal in na katerem procesorju (jedru)
- **POMEMBNO: vsak proces ima svoj ločen naslovni prostor sestavljen iz sklada, kopice, kode, statičnih spremenljivk**
- Procesi med seboj lahko komunicirajo preko skupnega pomnilnika (shared memory segment), vendar je preklapljanje med procesi časovno in prostorsko drago
- Raje uporabimo niti

Niti

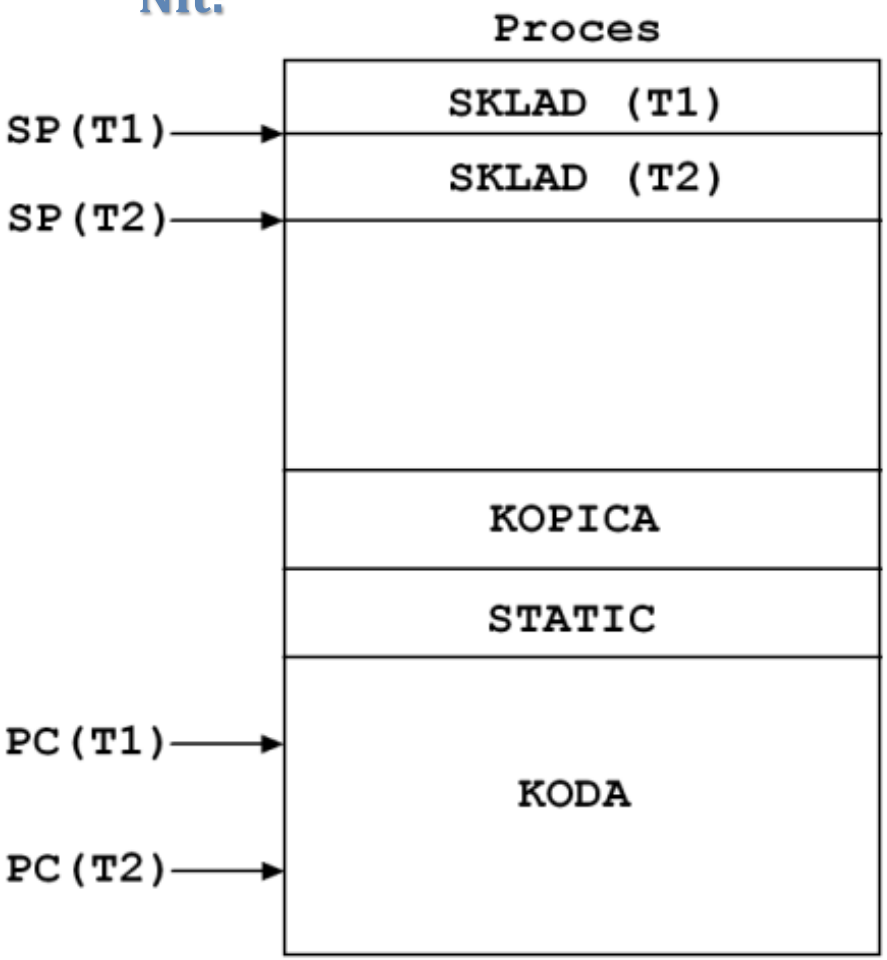
- Niti so sestavni deli nekega procesa
- Vsaki niti pripada lasten programski števec (PC), lasten sklad in lasten kazalec na sklad (SP), vendar si vse niti v procesu delijo isti kodni segment, isto kopico in statične podatke
- Niti se izvajajo v istem naslovnem prostoru, zato je komunikacija med njimi preko pomnilnika precej bolj enostavna
- Moderni OS znajo razvrščati in preklapljati posamezne niti – precej manj časovne in prostorske režije, kot pri procesih

Niti - procesi

Proces:



Nit:



Pthreads: POSIX Threads

- Pthreads je knjižnica funkcij jezika C, namenjena večnitnemu programiranju po standardu POSIX
  - IEEE PORTABLE Operating System Interface (POSIX), standard 1003.1
- Knjižnica vsebuje 60+ funkcij
  - upravljanje z nitmi: ustvari, zaključi,...
  - delo s ključavnicami
  - ...
- V programih moramo vključiti zaglavje `pthread.h`
- Prevedeno kodo moramo povezati s knjižnico pthread

Pthreads poimenovanje

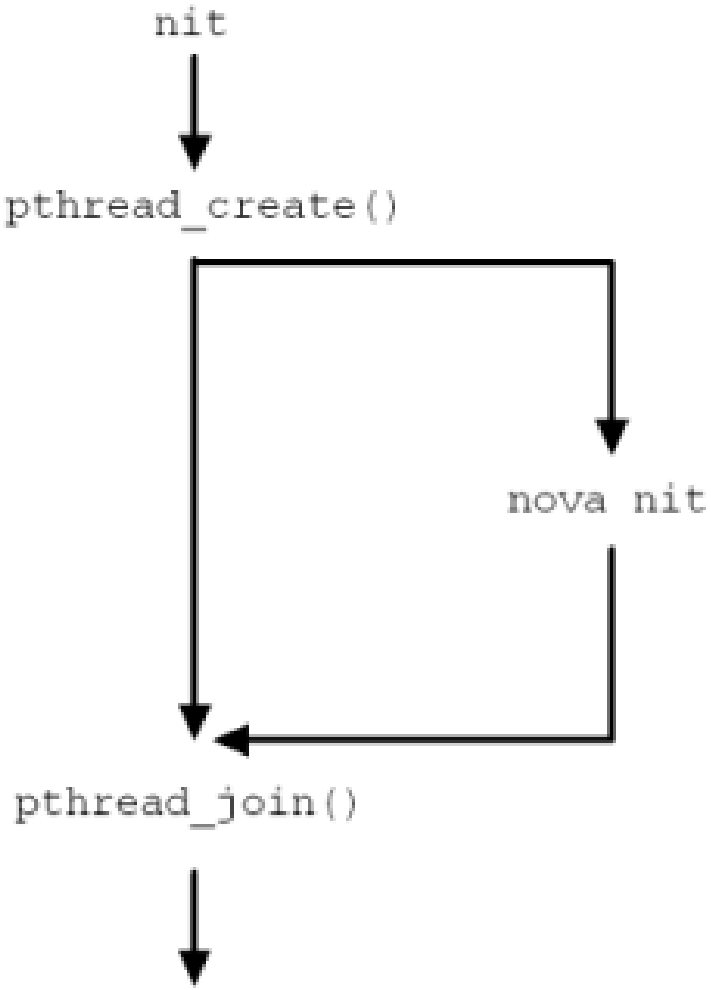
- Podatkovni tipi: **`pthread_[object]_t`**
- Funkcije: **`pthread[_object]_action`**
- Konstante/Makroji: **`PTHREAD_namen`**

- Primeri:
  - `pthread_t` : podatkovni tip nit
  - `pthread_create()` : funkcija za ustvarjanje niti
  - `pthread_mutex_t` : podatkovni tip ključavnica
  - `pthread_mutex_lock()` : funkcija za zaklepanje ključavnice
  - `PTHREAD_MUTEX_INITIALIZER`

Hello world

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

#define NUM 100

void *print_msg(void *);

main() {

        pthread_t t1, t2;          /* dve niti*/
        pthread_create(&t1, NULL, print_msg, (void *)"hello");
        pthread_create(&t2, NULL, print_msg, (void *)"world");

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
}

void *print_msg(void *message){

        char* pstring = (char*) message;
        int i;

        for (i = 0; i < NUM; i++) {
                printf("%s   ", pstring);
        }
}
```

phread_create, pthread_join



pthread_create()

pthread_create	
NAMEN	Ustvari novo nit
INCLUDE	`#include <pthread.h>`

UPORABA	`int pthread_create(pthread_t *thread,`
	`pthread_attr_t *attr,`
	`void *(*func) (void *)`
	`void *arg);`

ARGUMENTI	`thread` kazalec na spremenljivko tipa `pthread_t`
	`attr` kazalec na spremenljivko tipa `pthread_attr_t` ali `NULL`
	`func` funkcija, ki jo bo izvedla nit
	`arg` argumenti funkcije `func`

VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija ustvari novo nit in znotraj ustvarjene niti pokliče funkcijo func z argumenti arg.

pthread_join()

pthread_join	
NAMEN	Čaka, da se nit zaključi
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_join(pthread_t *thread, void **retval);`
ARGUMENTI	`thread` kazalec na spremenljivko tipa `pthread_t` `retval` kazalec na spremenljivko, ki sprejme povratno vrednost iz niti
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija povzroči, da klicoča nit čaka dokler se nit `thread` ne zaključi. Šele nato klicoča nit nadaljuje z izvajanjem.

errcode?

- The Ten Commandments for C Programmers (Henry Spencer)
  - http://doc.cat-v.org/henry_spencer/ten-commandments
  - 6. zapoved:

    If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest ``it cannot happen to me'', the gods shall surely punish thee for thy arrogance.

errcode?

```
#define errorexit(errcode, errstr) \
        fprintf(stderr, "%s: &d\n", errstr, errcode); \
        exit(1);
```

Hello world 2

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

#define NUM 5

#define errorexit(errcode, errstr) \
            fprintf(stderr, "%s: &d\n", errstr, errcode); \
            exit(1);

void *print_msg(void *);

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

            int errcode;
            pthread_t t1, t2;                   /* two threads*/

            //skusaj ustvariti niti-ob napaki vpisi kodo napake v errcode, izpisi in se vrni:
            if (errcode = pthread_create(&t1, NULL, print_msg, (void *)"hello")) {
                        errorexit(errcode, "pthread_create");
            }
            if (errcode=pthread_create(&t2, NULL, print_msg, (void *)"world")) {
                        errorexit(errcode, "pthread_create");
            }

            // pocakaj, da se obe niti koncata:
            if (errcode = pthread_join(t1, NULL)) {
                        errorexit(errcode, "pthread_join");
            }
            if (errcode = pthread_join(t2, NULL)) {
                        errorexit(errcode, "pthread_join");
            }

            // uspesno zakljuci proces:
            exit(0);
}
```

pthread_self()

- Vrne identifikator klicoče niti
  - kadar koli med izvajanjem nit lahko ugotovi ‘kdo je’
  - zelo uporabno, kadar želimo naslednji tok programa znotraj niti: „Če sem nit X, delaj to, sicer delaj nekaj drugega“
  - pazi: identifikator niti je tipa pthread_t !

- Uporaba: **`pthread_t pthread_self(void)`**

O poteku izvajanja

- Ko glavna nit ustvari novo nit, nadaljuje z izvajanjem svoje programske kode
- Pravkar ustvarjena nit bo začela z izvajanjem podane funkcije
- **POZOR: če se glavna nit predčasno zaključi, se prekine izvajanje vseh niti, ki jih je sama ustvarila!**
- Pomnilnik je skupen za vse niti v istem procesu:
  - Na ta način si lahko vse niti v procesu izmenjujejo podatke
  - Katerakoli nit lahko kadarkoli piše v skupni pomnilnik in bere iz skupnega pomnilnika, kar lahko povzroči težave
  - Na nek način bomo morali sinhronizirati dostope do skupnih podatkov

Prenos argumentov v funkcijo niti

- Funkcija, ki jo nit izvede, sprejme le en argument po referenci
- Več argumentov prenašamo s pomočjo struktur:

```
struct params {
        double* pArray1;        // kazalec na del vektorja 1
        double* pArray2;        // kazalec na del vektorja 2
        int myID;               // ID posamezne niti
};

struct params args;

...

pthread_create(&threads[i],
               NULL,
               delaj_nekaj,
               (void *) &args);
```

Prenos argumentov v funkcijo niti

- V funkciji, ki jo izvaja nit, moramo opraviti cast argumentov:

```
void *delaj_nekaj (void *arg) {

   struct params *argumenti;
   ...

   // CAST argumentov:
   argumenti = (struct params *) arg;

   // dostop do argumentov:
   argumenti -> ...

}
```

O poteku izvajanja

- Ne pozabimo v glavni niti počakati, da se vse ustvarjene niti zaključijo!
- Če pozabimo poklicati pthread_join(), bo verjetno glavna nit zaključila z izvajanjem svoje kode še preden se bodo zaključile vse ustvarjene niti – le te se bodo nemudoma prekinile
- Ko ustvarjamo več niti, moramo paziti, da ima vsaka nit svoj identifikator – tako bomo lahko za vsako nit posebej klicali pthread_join()

Medsebojna komunikacija

- Vse niti nekega procesa se izvajajo v skupnem naslovnem prostoru.
- Delijo si torej podatkovni(e) segment(e) nekega procesa.
- Vsaka nit vidi vse globalne spremenljivke.
- Lokalne spremenljivke so med nitmi nevidne, saj vsaka nit uporalja lasten skladovni segment.
- Komunikacija med nitmi torej lahko poteka preko spremenljivk v skupnem naslovnem prostoru.
- To pomeni, da lahko več niti naslavlja in dostopa do iste pomnilniške besede – dostop je lahko bralni in/ali pisalni.

Medsebojna komunikacija - zgled

- Predpostavimo, da želimo v neki preprosti iteraciji implementirati števec – program naj čim hitreje prišteje do N
- Ni pomembno, kje in zakaj bi to potrebovali, naj nam služi le kot zgled
- Namesto, da le ena nit povečuje vrednost števca do N, naj to počne NTHREADS niti
- Na tem zgledu bomo spoznali osnovne probleme pri uporabi skupnih spremenljivk in njihove rešitve

Zgled – rešitev 1

- Ustvarimo NTHREADS niti in vsaka poveča vrednost števca za 1
- Vsaka nit izvede funkcijo:

```
void *stej(void *);      // funkcija, ki jo izvedejo niti
```

- Vsak nit dobi kot argument naslednjo strukturo:

```
struct params {
   int nIteracij;          // kolikokrat steje posamezna nit
   int myID;               // ID posamezne niti
};
```

Zgled – rešitev 1

```
int main(int argc, char* argv[]) {
   int i;
   int errcode;
   struct params args[NTHREADS];            // NTHREADS argumentov funkcij
   pthread_t threads[NTHREADS];             // NTHREADS niti

   // inicializiraj stevec:
   counter = 0;

   //skusaj ustvariti niti - ob napaki vpisi kodo napake v errcode, izpisi in se vrni:
   for (i = 0; i < NTHREADS; i++) {
      // init parametrov za vsako nit:
      args[i].myID = i;
      args[i].nIteracij = N / NTHREADS;

      if (errcode = pthread_create(&threads[i], NULL, stej, (void *) &args[i])) {
                                 errorexit(errcode, "pthread_create");
      }
   }

   // pocakaj, da se vse niti koncajo:
   for (i = 0; i < NTHREADS; i++) {
      if (errcode = pthread_join(threads[i], NULL)) {
                     errorexit(errcode, "pthread_join");
      }
   }

    printf("Stevec = %d\n", counter);

    // uspesno zakljuci proces:
    exit(0);
}
```

Zgled – rešitev 1

- Funkcija za povečevanje števca:

```
void *stej(void *args){

   struct params *arguments = (struct params *) args;  // cast argumentov
                                                       // nazaj v pravi tip

   int j;
   int myID;
   int nIteracij;

   myID = arguments->myID;
   nIteracij = arguments->nIteracij;

   // stejemo....
   for (j = 0; j<nIteracij; j++) {
      counter = counter + 1; //namenoma pustimo taksen zapis: vsaka nit mora
                             //spremenljivko counter prebrati, pristeti 1 in
                             //zapisati nazaj
   }
}
```

- PROBLEM: spremenljivka counter je globalna – ena nit prebere vrednost v register in ji prišteje ena, druga ravno takrat prebere to vrednost iz pomnilnika, prva nit pa shrani rezultat nazaj vpomnilnik. Katero vrednost bo prebrala in povečala druga nit ?!?

Sinhronizacija dostopov

- Dostope do neke skupne spremenljivke moramo na nek način sinhronizirati
- Problematični so predvsem dostopi tipa beri, spremeni, shrani. Taki dostopi lahko pripeljejo do TVEGANEGA STANJA (ang. race condition), saj rezultat ni jasen v naprej
- Rešitev tega problema predstavljajo KLJUČAVNICE
- Ključavnice omogočijo izvajanje določenega zaporedja ukazov le eni niti – ostale morajo čakati
- Ena nit „zaklene“ dostop do določenega dela programske kode ostalim nitim
- Programski kodi, ki jo zaklepamo, rečemo KRITIČNA SEKCIJA

Ključavnice (medsebojno izključevanje)

- Ključavnica je ena pomnilniška beseda, katere vsebina je binarna vrednost : ZAKLENJENO/ODKLENJENO
- Vsaka nit pred vstopom v kritično sekcijo preveri stanje ključavnice.
  - Če je odklenjena, jo zaklene in vstopi v kritično sekcijo.
  - Če je zaklenjena, čaka pred kritično sekcijo (v zanki) dokler se ne odklene
- Po izstopu iz kritične sekcije mora nit odkleniti ključavnico, da omogoči vstop drugim nitkam

Ključavnice (medsebojno izključevanje)

- Kako zaklenemo ključavnico?
  - Nit mora vrednost ključavnice prebrati, spremeniti in shraniti nazaj
  - Med branjem in shranjevanjem nazaj lahko še ena ali več niti prebere ODKLENJENO
  - Zato morajo biti bralno-pisalni dostopi po vodilu do ključavnice neprekinjeni!
  - Ko se začne bralni dostop do ključavnice, moramo onemogočiti vse druge prenose po vodilu, dokler se ne zaključi še pisalni dostop do ključavnice
  - Procesorji imajo signal LOCK, ki se uporabi pri prenosih po vodilu

Ključavnice (medsebojno izključevanje)

- Kako zaklenemo ključavnico?
  - Procesorji imajo posebne ukaze, ki aktivirajo signal LOCK
  - To niso navadni ukazi za dostop do pomnilnika
  - Uporabljajo se
    - atomski ukazi

      (npr. test-and-set, fetch-and-add, compare-and-swap) ali
    - par ukazov LL-SC (load linked, store conditional)

Ključavnice (medsebojno izključevanje)

- Medsebojno izključevanje (zaklepanje)
  - Atomski ukaz naredi dve stvari
    - vsebino pomnilniške lokacije prebere v register
    - na pomnilniško lokacijo vpiše potrditev branja
    - med tem na vodilu ni drugih operacij
    - Ukaz Test-and-Set (TAS)
      - Prebere vrednost pomnilniške lokacije
      - če je prebrana vrednost 0 (odklenjeno), na lokacijo zapiše 1 (zaklenjeno)
      - Primer:

```
poskus:  TAS R, L        ; čakaj, dokler ni odklenjeno; ko je, zakleni
         JNZ poskus
kritično: ...            ; izvedi kritične operacije
         ...
         MOV L, 0        ; odkleni
```

Ključavnice (medsebojno izključevanje)

- Medsebojno izključevanje (zaklepanje)
  - Pentium
  - Nekateri ukazi zbirnika imajo predpono LOCK

```
lock add $2, x      ; prištej 2 v x
```

  - Ukaz naredi dve operaciji na pomnilniku:
    branje stare in pisanje nove vrednosti
  - Zaklene vodilo

  - Večprocesorski sistemi: protokoli kot je MESI ali vohunjenje na vodilu (samo določeni registri)

Ključavnice v pthreads

- Pthreads omogoča uporabo enostavnih ključavnic za medsebojno zaklepanje
- Uporaba ključavnic v Pthreads:
  - **Mutex** = **Mut**ual **Ex**clusion = medsebojno izključevanje
  - Deklaracija ključavnice: `pthread_mutex_t`
  - Inicijalizacija ključavnice: `pthread_mutex_init()`
  - Zaklepanje: `pthread_mutex_lock()`
  - Odklepanje: `pthread_mutex_unlock()`
  - Sproščanje pomnilnika namenjenaga ključavnici: `pthread_mutex_destroy()`

pthread_mutex_lock()

pthread_mutex_lock	
NAMEN	Zakleni ključavnico
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_mutex_lock(pthread_mutex_t *lock);`
ARGUMENTI	`lock` kazalec na ključavnico
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija zaklene ključavnico `lock`. Če je ključavnica odklenjena, jo zaklene in se vrne. Če je ključavnica `lock` že zaklenjena, potem se nit ustavi in čaka na odklenjeno ključavnico.

pthread_mutex_trylock()

pthread_mutex_trylock	
NAMEN	Zakleni ključavnico
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_mutex_lock(pthread_mutex_t *lock);`
ARGUMENTI	`lock` kazalec na ključavnico
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija poskuša zakleniti ključavnico `lock`. Če je ključavnica odklenjena, jo zaklene in se vrne. Če je ključavnica `lock` že zaklenjena, potem se klic funkcije zaključi in klicoča nit nadaljuje z izvajanjem.

pthread_mutex_unlock()

pthread_mutex_unlock	
NAMEN	Odkleni ključavnico
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_mutex_unlock(pthread_mutex_t *lock);`
ARGUMENTI	`lock` kazalec na ključavnico
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija odklene ključavnico `lock`. Če ostale niti čakajo na odklenjeno ključavnico, jo bo verjetno ena izmed njih takoj zatem zaklenila.

Če ključavnice ne odklenemo (pozaba, napaka, ...), se bodo vse niti, ki čakajo na odklenjeno ključavnico trajno ustavile....

Lepa praksa: samo nit, ki je zaklenila ključavnico `lock` jo lahko odklene!

pthread_mutex_init()

pthread_mutex_init	
NAMEN	Inicializira ključavnico
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_mutex_init(pthread_mutex_t *lock` `const pthread_mutexattr_t *attr);`
ARGUMENTI	`lock` kazalec na ključavnico `attr` atributi ključavnice; če je NULL, bodo prevzeti
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija inicializira ključavnico lock z atributi attr. Če je attr NULL, potem se ključavnico inicializira s prevzetimi vrednostimi in je po inicializaciji odklenjena.

Inicializacijo ključavnice lahko opravimo na dva načina:

- statično (ob deklaraciji):

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- dinamično:

```
pthread_mutex_init(&lock, NULL);
```

pthread_mutex_destroy()

pthread_mutex_destroy	
NAMEN	Odstrani ključavnico
INCLUDE	`#include <pthread.h>`
UPORABA	`int pthread_mutex_destroy(pthread_mutex_t *lock);`
ARGUMENTI	`lock` kazalec na ključavnico
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija odstrani ključavnico na katero kaže `lock`. Sprosti vire, ki jih uporablja ključavnica, ne pa pomnilniškega prostora same strukture. Isto ključavnico je zato mogoče spet uporabiti, ne da bi jo ustvarili. Pred ponovno uporabo jo moramo le inicializirati.

Odstranjevanje zaklenjene ključavnice ni varno !!!

Zgled – rešitev s ključavnico

- Vrimo se k našemu števcu
- Dostop do spremenljivke counter moramo sinhronizirati s ključavnico, ki jo najprej deklariramo in inicializiramo (ključavnica mora biti globalna!):

```
pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;
```

- Zaklenemo kritično sekcijo:

```
//ZAKLENI DOSTOP DO SKUPNE SPREMENLJIVKE:
if (errcode = pthread_mutex_lock(&counter_lock)){
   errorexit(errcode, "pthread_mutex_lock");
}

    counter = counter + 1;

// ODKLENI:
if (errcode = pthread_mutex_unlock(&counter_lock)){
   errorexit(errcode, "pthread_mutex_unlock");
}
```

Smrtni objem

- Smrtni objem je pojav, ko dve (ali več) niti skušata zakleniti dve kritični sekciji, vendar to počneta v različnem vrstnem redu. Posledica je t.i. **smrtni objem**, ko se dve niti ujameta v neskončnem čakanju na prosto odklenjeno ključavnico
- Najboljši način, da se izognemo smrtnemu objemu, je, da vse niti ključavnice zaklepajo v enakem vrstnem redu! Uporabimo t.i. **hierarhijo ključavnic – vse ključavnice številčimo v navideznem zaporedju**
- **Hierarhija ključavnic**: posamezna nit naj nikoli ne skuša zakleniti ključavnice n, če ima zaklejeno ključavnico m, pri čemer je n<m !

Smrtni objem - zgled

Nit 1	Nit 2

`pthread_mutex_lock(&m1);`
`...`
 	
`pthread_mutex_lock(&m2);`
`...`
 	
`...`
`pthread_mutex_unlock(&m2);`
`...`
`pthread_mutex_unlock(&m1);`
	`pthread_mutex_lock(&m2);`
	`...`
	
	`pthread_mutex_lock(&m1);`
	`...`
	
	`...`
	`pthread_mutex_unlock(&m1);`
	`...`
	`pthread_mutex_unlock(&m2);`

Možen scenarij:
Nit 1 zaklene ključavnico m1. Preden ji uspe zakleniti še ključavnico m2, le to zaklene Nit 2. Nit 2 sedaj čaka, da se odklene ključavnica m1, vendar jo Nit 1 ne bo odklenila, dokler se ne odklene ključavnica m2 – SMRTNI OBJEM!!!!

Pogojno zaklepanje

Včasih se ne moremo držati hierarhije niti. Takrat se poslužimo pogojnega zaklepanja:

Nit 1	Nit 2

`pthread_mutex_lock(&m1);`
`  ...`
 	
`pthread_mutex_lock(&m2);`
`  ...`
 	
`  ...`
`pthread_mutex_unlock(&m2);`
`  ...`
`pthread_mutex_unlock(&m1);`
	`while(1){`
	`  pthread_mutex_lock(&m2);`
	`  // previdno poskusi zakleniti m1:`
	`  if(pthread_mutex_trylock(&m1)==0) {`
	`  // uspelo mi je zakleniti...`
	`  break; // prekini while zanko`
	`  }`
	`  // Ni mi uspelo zakleniti m1, zato sprosti`
	`  // m2 in poskusi vse se enkrat:`
	`  pthread_mutex_unlock(&m2);`
	`}`
	
	`// tu se znajdem, ce mi je uspelo zakleniti obe.`
	`// in sprocesirati kriticno sekcijo. Zato ju`
	`// moram odkleniti:`
	
	`  /*kriticna sekcija*/`
	
	`pthread_mutex_unlock(&m1);`
	`pthread_mutex_unlock(&m2);`

Ali res vedno potrebujem ključavnice?

- Ključavnice imajo veliko pomankljivost: nenehno zaklepanje kritičnih sekcij ZELO upočasni izvajanje kode
- Kritične sekcije naj bodo čim krajše! Zato, da jo posamezna nit čimprej zapusti in omogoči ostalim nitim nadaljevanje izvajanja
- Premislimo ali res potrebujemo kritično sekcijo in ključavnice
- Problem se velikokrat da rešiti z uporabo lastnih spremenljivk

Ali res vedno potrebujem ključavnice?

- V našem primeru štetja bi vsaki niti dodelili lasten števec
- Vsaka nit šteje v določenem obsegu in skrani prešteto vrednost v lasten lokalni števec
- Glavna nit main() počaka, da se vse niti zaključijo in nato sešteje vrednosti vseh lastnih števcev

Zgled – rešitev z lastnimi spremenljivkami

- Vrimo se k našemu števcu
- Vsaka nit naj šteje v lasten števec in ga vrne klicoči niti. Zato ga damo v strukturo, s katero v nit prenašamo argumente:

```
struct params {
    int nIteracij;     // kolikokrat steje posamezna nit
    int myID;          // ID posamezne niti
    int myCounter;     // lasten stevec vsake niti
};
```

Zgled – rešitev z lastnimi spremenljivkami

- V funkciji štetja vsaka nit povečuje le lasten števec v podanem obsegu. Zato ni potrebe po zaklepanju:

```
void *stej(void *args){

   struct params *arguments = (struct params *) args;  // cast
   int j;
   int myID;
   int nIteracij;
   int errcode;
   myID = arguments->myID;
   nIteracij = arguments->nIteracij;

   // stejemo....
   for (j = 0; j<nIteracij; j++) {
      // vsaka nit pise neposredno v podano strukturo - na ta nacin se tudi
      // vrne vrednost stetja iz posamezne niti:
      arguments->myCounter = arguments->myCounter + 1;
   }
}
```

Zgled – rešitev z lastnimi spremenljivkami

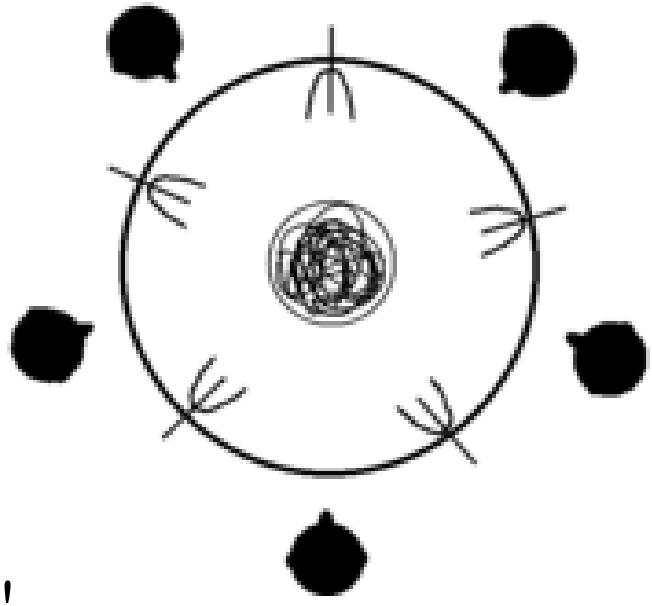
```
int main(int argc, char* argv[]) {
   int i;
   int errcode;
   struct params args[NTHREADS];                    // NTHREADS argumentov funkcij
   pthread_t threads[NTHREADS];                     // NTHREADS niti

   // inicializiraj stevec:
   counter = 0;
   //skusaj ustvariti niti - ob napaki vpisi kodo napake v errcode, izpisi in se vrni:
   for (i = 0; i < NTHREADS; i++) {
      // init parametrov za vsako nit:
      args[i].myID = i;
      args[i].nIteracij = N / NTHREADS;
      args[i].myCounter = 0;
      if (errcode = pthread_create(&threads[i], NULL, stej, (void *) &args[i])) {
                              errorexit(errcode, "pthread_create");
      }
   }

   // pocakaj, da se vse niti koncajo...
   for (i = 0; i < NTHREADS; i++) {
      if (errcode = pthread_join(threads[i], NULL)) {
                     errorexit(errcode, "pthread_join");
      }
   }
   //...in nato sestej vse lokalne vrednosti stevcev:
   for (i = 0; i < NTHREADS; i++) {
      counter += args[i].myCounter;
   }
   // uspesno zakljuci proces:
   exit(0);
}
```

Nekaj problemov

- 1. Skalarni produkt
  - Problem
    - Imamo dva vektorja, zmnožiti moramo soležne elemente in zmnožke sešteti
  - Ideje:
    - Brez zaklepanja niti
    - Zaklepanje z mutexom
    - Neodvisno reševanje delov problema v posameznih nitih in seštevanje delnih rezultatov

Nekaj problemov

- 2. Pet filozofov, pet krožnikov špagetov in pet vilic
  - Problem
    - Za okroglo mizo zboruje pet filozofov, ki v več ciklih razmišljajo, so lačni, jejo, potem spet razmišljajo, ...
    - Jedo špagete, ki jih ima vsak na svojem krožniku.
    - Pri jedi potrebuje vsak dvoje vilice, ki pa jih imajo skupaj na voljo samo pet!
    - Uporabijo lahko samo vilice na svoji levi in vilice na svoji desni.



  - Ideje
    - Vilice opišemo s ključavnicami
    - Ali lahko pride do smrtnega objema? Kako ga preprečiti?

Nekaj problemov

- 3. Pisatelji in bralci
  - Problem:
    - Imamo več bralcev, ki berejo tisto, kar napišejo pisatelji.
    - Med pisanjem enega pisatelja ne sme pisati noben drug pisatelj.
    - Noben pisatelj ne sme pisati med branjem.
  - Ideja:
    - Izvesti je potrebno učinkovit sistem zaklepanja s ključavnicami

Semaforji

- Standard predvideva, da ključavnico odklene tista nit, ki jo tudi zaklene
  - če ključavnico odklepa druga nit, obnašanje ni definirano
  - če nit odklepa že odklenjeno ključavnico, obnašanje ni definirano
- Bolj splošni od niti so semaforji
  - Semaforji niso last niti ali procesov – odklepanje/zaklepanje lahko izvajajo različne niti in celo različni procesi
    - Pri medprocesorski komunikaciji je semaforje potrebno poimenovati (to bomo spustili)
  - za razliko od ključavnic (binarne vrednosti odklenjeno/zaklenjeno) so semaforji števni

Semaforji

- Zgodovina
  - Dijkstra, nizozemska, 1968
- Semafor je podatkovna struktura, ki vključuje celoštevilčno spremenljivko s,
  - nad spremenljivko s sta definirani dve atomski operaciji
    - poskusi vstopiti (če s > 0 vstop in s ← s – 1),
    - sprosti (s ← s + 1).
  - po atomski operaciji sprosti nit obvesti ostale nit
  - spremenljivka s je inicializirana na neko začetno vrednost
- Semafor, ki lahko zavzame samo vrednosti 0 in 1 se imenuje binarni semafor
  - bolj splošen od ključavnic (prejšnja stran)

Semaforji

- Semaforji niso del knjižnice pThreads, gre za njeno razširitev
  - v pThreads so realizirani s ključavnicami in pogojnimi spremenljivkami (o teh kasneje)
  - Zaglavje `#include <semaphore.h>`
  - Pomembne funkcije

sem_init()

sem_init	
NAMEN	Inicializira semafor
INCLUDE	`#include <semaphore.h>`
UPORABA	`int sem_init(sem_t * semaphore_p, int shared, unsigned initial_value);`
ARGUMENTI	`semaphore_p` kazalec na semafor `shared` 0 – semafor je namenjen samo enemu procesu, običajno v pThreads, 1 – semafor se uporablja za komunikacijo med procesi `initial_value` začena vrednost semaforja
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija inicializira semafor `semaphore_p`

sem_wait()

sem_wait	
NAMEN	Inicializira semafor
INCLUDE	`#include <semaphore.h>`
UPORABA	`int sem_wait(sem_t * semaphore_p);`
ARGUMENTI	`semaphore_p` kazalec na semafor
VRNE	0, če je klic uspel errcode, če klic ni uspel

Če je vrednost semaforja več od 0, ga funkcija zmanjša.
V primeru, da je semafor 0, se bo nit, se bo izvajanje niti zaustavilo.

sem_post()

sem_post	
NAMEN	Inicializira semafor
INCLUDE	`#include <semaphore.h>`
UPORABA	`int sem_post(sem_t * semaphore_p);`
ARGUMENTI	`semaphore_p` kazalec na semafor
VRNE	0, če je klic uspel errcode, če klic ni uspel

Ob klicu sem_post se vrednost semaforja poveča za 1 in ena od niti, ki so se ustavile pri sem_wait, lahhko nadaljuje z izvajanjem.

sem_destroy()

Sem_destroy	
NAMEN	Uniči semafor
INCLUDE	`#include <semaphore.h>`
UPORABA	`int sem_destroy(sem_t * semaphore_p);`
ARGUMENTI	`semaphore_p` kazalec na semafor
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija sprosti vire, ki jih uporablja semafor na katerega kaže kazalec `semaphore_p`. Dejansko ne sprosti pomnilniškega prostora za strukturo semafor, zato moramo pred ponovno uporabo semafor le na novo inicializirati.

Problem: pregrada

- V mnogih aplikacijah se morajo vse niti občasno počakati in potem skupaj nadaljevati delo
  - Rešitev s ključavnicami
    - trošimo CPU cikle med čakanjem v zanki
    - Upočasnjeno delovanje, če je niti več kot procesorjev
    - pri naslednji prepreki ne moremo uporabiti iste spremenljivke
  - Rešitev s semaforji
    - nimamo več čakanja v zanki
    - še vedno imamo težave z uporabo iste spremenljivke

Pogojne spremenljivke

- Uporabljamo jih za zaustavitev in ponovni zagon niti v povezavi s ključavnicami
  - Pogojna signalizacija
  - Vedno v povezavi s ključavnicami

- Zaglavje: `#include <pthreads.h>`
- Spremenljivka je tipa `pthread_cond_t`
- Operacije:
  - wait
  - signal
  - broadcast

ptread_cond_init()

Pthread_cond_init	
NAMEN	Inicializira pogojno spremenljivko
INCLUDE	`#include <thread.h>`
UPORABA	`int pthread_cond_init(pthread_cond_t *condvar_p, pthread_condattr *attr);`
ARGUMENTI	`condvar_p` kazalec na pogojno spremenljivko `attr` kazalec na atribute s katerimi inicializiramo spremenljivko
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija inicializira pogojno spremenljivko na katero kaže kazalec `condvar_p`.

ptread_cond_destroy()

pthread_cond_destroy	
NAMEN	Ostrani pogojno spremenljivko
INCLUDE	`#include <thread.h>`
UPORABA	`int pthread_cond_destroy(pthread_cond_t *condvar_p);`
ARGUMENTI	`condvar_p` kazalec na pogojno spremenljivko
VRNE	0, če je klic uspel errcode, če klic ni uspel

Funkcija sprosti vire, ki jih uporablja pogojna spremenljivka, na katero kaže kazalec `condvar_p`. Pred ponovno uporabo je treba spremenljivko samo inicializirati.

ptread_cond_wait()

pthread_cond_wait	
NAMEN	ustavi izvajanje niti dokler ne dobi signala za nadaljevanje
INCLUDE	`#include <thread.h>`
UPORABA	`int pthread_cond_wait(pthread_cond_t *condvar_p, pthread_mutex *mutex_p);`
ARGUMENTI	`condvar_p` kazalec na pogojno spremenljivko `mutex_p` kazalec na ključavnico
VRNE	0, če je klic uspel errcode, če klic ni uspel

- Nit čaka v tej funkciji, dokler pogojna spremenljivka ne dobi signala za nadaljevanje.
- Nit se pri tem zaustavi in postavi v vrsto zaklenjenih niti (na zadnje mesto)
- Za uporabo funkcije je potrebno imeti zaklenjeno ključavnico; ko se nit zaustavi, odklene ključavnico, da druge niti lahko delajo naprej
- Ko pogojna spremenljivka prejme signal, se nit zbudi in ključavnica zaklene.

ptread_cond_signal()

pthread_cond_signal	
NAMEN	Pošlje signal za nadaljevanje za eno nit
INCLUDE	`#include <thread.h>`
UPORABA	`int pthread_cond_signal(pthread_cond_t *condvar_p);`
ARGUMENTI	`condvar_p` kazalec na pogojno spremenljivko
VRNE	0, če je klic uspel errcode, če klic ni uspel

- Pogojni spremenljivki pošlje signal za nadaljevanje.
- Prva nit, ki čaka v vrsti pogojne spremenljivke, lahko nadaljuje svoje delo.

ptread_cond_broadcast()

pthread_cond_broadcast	
NAMEN	Pošlje signal za nadaljevanje vsem nitim
INCLUDE	`#include <thread.h>`
UPORABA	`int pthread_cond_broadcast(pthread_cond_t *condvar_p);`
ARGUMENTI	`condvar_p` kazalec na pogojno spremenljivko
VRNE	0, če je klic uspel errcode, če klic ni uspel

- Pogojni spremenljivki pošlje signal, da lahko vse čakajoče niti nadaljujejo.

Problem: semafor

- Semafor se da narediti s ključavnicami in pogojnimi spremenljivkami

```
typedef struct __Sem_t
{
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Sem_t;


void Sem_init(Sem_t *s, int value)
{
    s->value = value;
    pthread_cond_init(&s->cond, NULL);
    pthread_mutex_init(&s->lock, NILL);
}
```

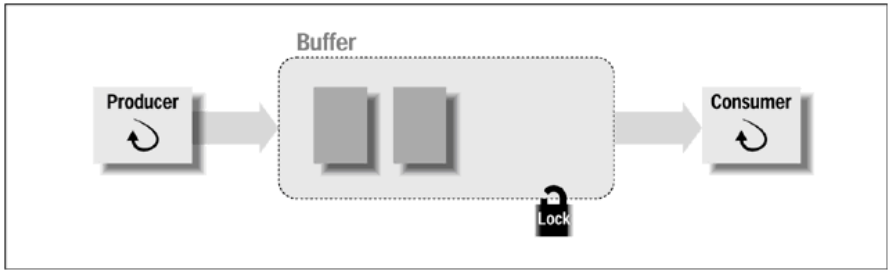
```
void Sem_wait(Sem_t *s)
{
    pthread_mutex_lock(&s->lock);
    while (s->value <= 0)
        pthread_cond_wait(&s->cond,
                          &s->lock);
    s->value--;
    pthread_mutex_unlock(&s->lock);
}

void Sem_post(Sem_t *s)
{
    pthread_mutex_lock(&s->lock);
    s->value++;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock);
}
```

Problem: pregrada (nadaljevanje)

- Rešitev s ključavnicami in pogojnimi spremenljivkami
- Posebni konstrukti v pthreads
  - spremenljivka: pthread_barrier_t
  - funkcije:
    - pthread_barrier_init,
    - pthread_barrier_destroy,
    - pthread_barrier_wait
  - Ni na MacOSX

Izmenjava podatkov med nitmi

- Niti si izmenjujejo podatke preko medpomnilnikov
- Pri izmenjavi lahko nit nastopa v dveh vlogah:
  - proizvajalec predaja podatke
  - porabnik prevzema podatke



  - Za vzpostavitev komunikacije potrebujemo
    - Medpomnilnik
    - Ključavnico
    - Mehanizem za uspavanje in bujenje niti
    - Podatke o stanju v medpomnilniku
  - Klasični problem: proizvajalec – porabnik
    - En proizvajalec, en porabnik, medpomnilnik za eno nalogo
    - Večji medpomnilnik, več proizvajalcev, več porabnikov
    - Statični/dinamični medpomnilnik

Problem: izdelovalec - porabnik

- Izdelovalec – porabnik (angl. producer – consumer)
  - Problem
    - Izdelovalec postavlja izdelke v skladišče.
    - Porabnik pobira izdelke iz skladišča.
    - Izdelovalec ne sme dodajati izdelkov, če je skladišče polno, porabnik jih ne sme jemati, če je skladišče prazno.
    - Paziti je treba na usklajen dostop izdelovalcev in porabnikov do skladišča.
  - Ideje:
    - Uporaba pogojnih spremenljivk

Varno delo z nitmi

- ang. thread-safe
- Nekatere funkcije v C-ju si med klici zapomnijo vrednost (uporabijo spremenljivko deklarirano kot `static`)
  - Spremenljivke deklarirane kot static so shranjene v deljenem pomnilniku
  - Možne napake, če funkcijo kličejo različne niti
- Funkcije, ki niso ustrezne za delo z nitmi, imajo običajno tudi izboljšane različice
- Primer: strtok
  - funkcija, ki iz niza izlušči ključne besede (nizi od separatorja do separatorja)
  - `strtok(char *string, const char *separators);`
  - `strtok_r(char *string, const char *separators, char **saveptr_p);`
  - Novi argument nadomešča statično spremenljivko
- Na funkcije, za katere se je izkazale, da niso varne za delo z nitmi, prevajalniki velikokrat opozorijo (ang. deprecated)

Varno delo z nitmi

- Funkcija je varna za delo z nitmi (thread safe), če pri večnitnem izvajanju vrne vrednost v skladu s specifikacijo
- To ne velja za vse funkcije
  - Nekatere funkcije si med izvajanjem zapomnijo vrednost
    - spremenljivke deklarirane kot static so shranjene v deljenem pomnilniku
    - primer:
      - `strtok(char *string, const char *separators);`

Varno delo z nitmi

- strtok
  - Ob prvem klicu povemo niz, ki ga želimo razdeliti
  - Ob vsakem naslednjem klicu dobi podniz do ločitvenega znaka
  - Kazalec na naslednji niz je hranjen v statični spremenljivki
  - Če delamo z več nitmi, bodo vse spreminjale isto statično spremenljivko
    - Ena nit bo tako dobila podniz, ki bi ga morala dobit druga nit

Varno delo z nitmi

- Rešitve
  - Izboljšane različice funkcij (angl. Reentrant)
    - Primer
      - `strtok_r(char *string, const char *separators, char **saveptr_p);`
      - `saveptr_p` nadomešča statično spremenljivko
  - Lokalni pomnilnik niti (angl. Thread Local Storage)
    - dopolnilo __thread pri definiciji spremenljivke
    - Za vsako nit inicializira svojo statično spremenljivko

Varno delo z nitmi

- Primer MonteCarlo PI
  - seed + rand:
    - ne vemo, kdaj bo kakšna nit klicala rand
    - uporabljena bodo ista naključna števila, vendar bodo različno razporejena med nitmi
    - rezultati ne bodo čisto enaki (zaokroževanje)
  - Enostaven generator naključnih števil

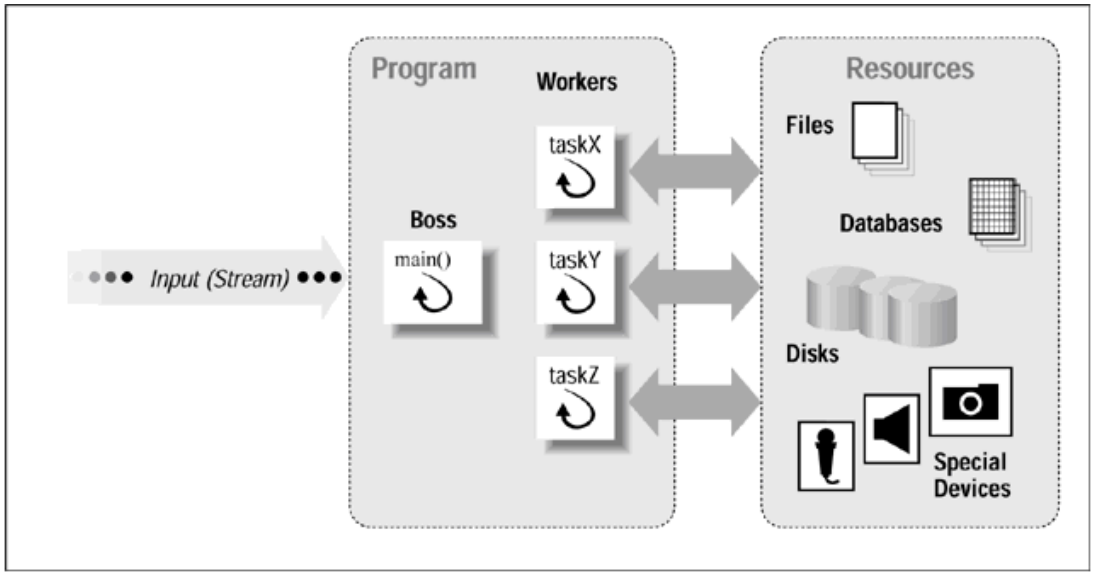
```
int a = 1103515245;
int b = 12345;
int m = 231;
int seed = 123456789;

int myrand()
{
    seed = (a * seed + c) % m;
    return seed;
}
```

Modeli večnitnih programov

- Ni pravil
- Nekaj prevladujočih modelov
  - Kako večnitne aplikacije razporejajo delo med niti
  - Kako niti komunicirajo med seboj
- Modeli
  - upravljalec/delavec (angl. boss/worker)
  - enakovredne niti (angl. peer)
  - cevovod (ang. pipline)

Modeli večnitnih programov

- Upravljalec/delavec
  - Ena sama nit (upravljalec) dostopa do vhodnih podatkov
  - Upravljalec
    - Za delo dinamično ustvari nit, delavca.
    - Delavcu dodeli delo glede na vhodne podatke.
    - Če je potrebno, počaka da delavec zaključi.
    - Se vrne na vrh zanke kjer čaka na novo delo.
  - Delavec
    - Procesira
    - Rezultat lahko shrani sam ali ga posreduje upravljalcu.



Modeli večnitnih programov

- Upravljalec/delavec
  - Dinamično ustvarjanje niti
    - Niti je toliko kot zahtevkov, precej režijskih stroškov

```
main()
{
    // upravljalec
    while (1)
    {
        pridobi zahtevo
        zaženi ustrezno nit
            zahtevaA: pthread_create(&nitA, NULL, zahtevaA, …)
            zahtevaB: pthread_create(&nitA, NULL, zahtevaB, …)
    }
}

zahtevaX()
{
    izvedi zahtevoX
    sinhroniziraj dostop do potrebnih virov
}
```

Modeli večnitnih programov

- Upravljalec/delavec
  - Bazen niti (angl. thread pool)
    - Upravljalec niti pripravi v naprej
    - Niti spijo in čakajo na delo

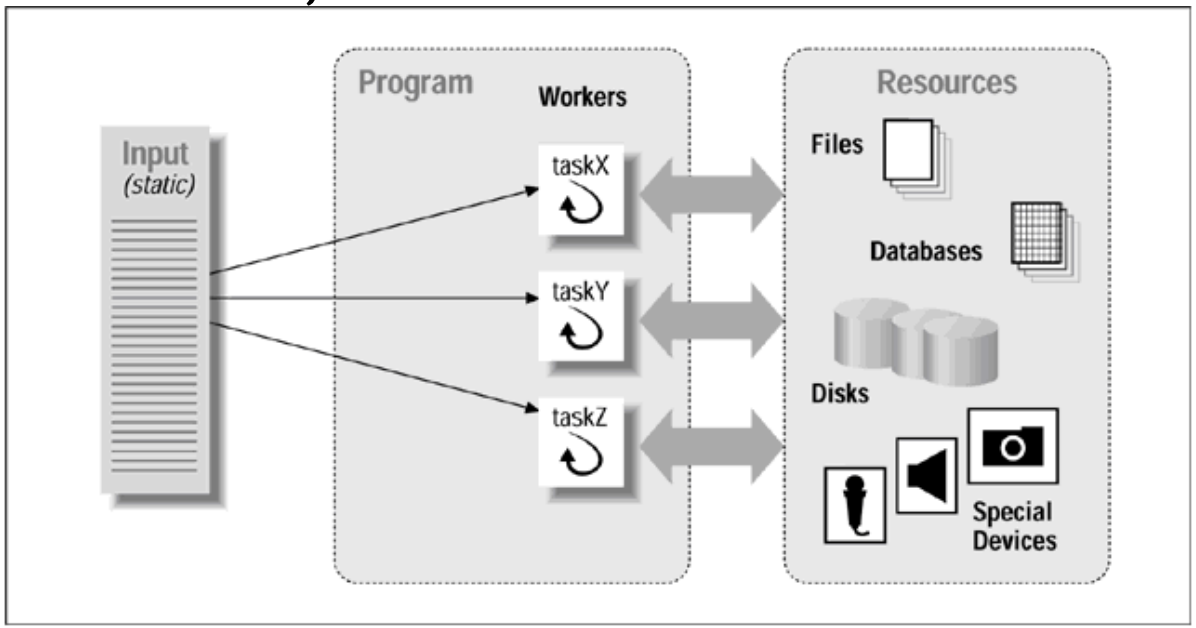
```
main()
{
    ustvari bazen niti
        pthread_create(…)
    while (1)
    {
        pridobi zahtevo
        postavi zahtevo v vrsto
        sporoči delavcem, da je delo na voljo
    }
}

delavec()
{
    while (1)
        nit spi, dokler je upravljalec ne zbudi
        vzame zahtevoX iz vrste
        za zahtevoX opravi nalogoX
}
```

Modeli večnitnih programov

- Upravljalec/delavec
  - Tipično se model uporabljan na strežnikih (baze, datotečni strežniki)
  - Zahteve prihajajo asinhrono, za korektno obdelavo poskrbi upravljalec
  - Skrbeti je treba, da med nitmi ni preveč komunikacije
    - Delavec ne sme blokirati upravljalca, saj potem ne bi mogel obravnavati vhodnih zahtev
    - Delavci med seboj se ne smejo preveč blokirati

Modeli večnitnih programov

- Enakovredne niti
  - Vse niti delajo sočasno, so enakovredne
  - Ob zagonu programa glavna nit ustvari vse ostale
  - Ta nit je nato lahko enakovredna ostalim ali pa samo čaka, da ostale zaključijo
  - Vsaka nit je odgovorna za svoj vhod
    - Ob kreiranju nit ve, kaj mora obdelati



Modeli večnitnih programov

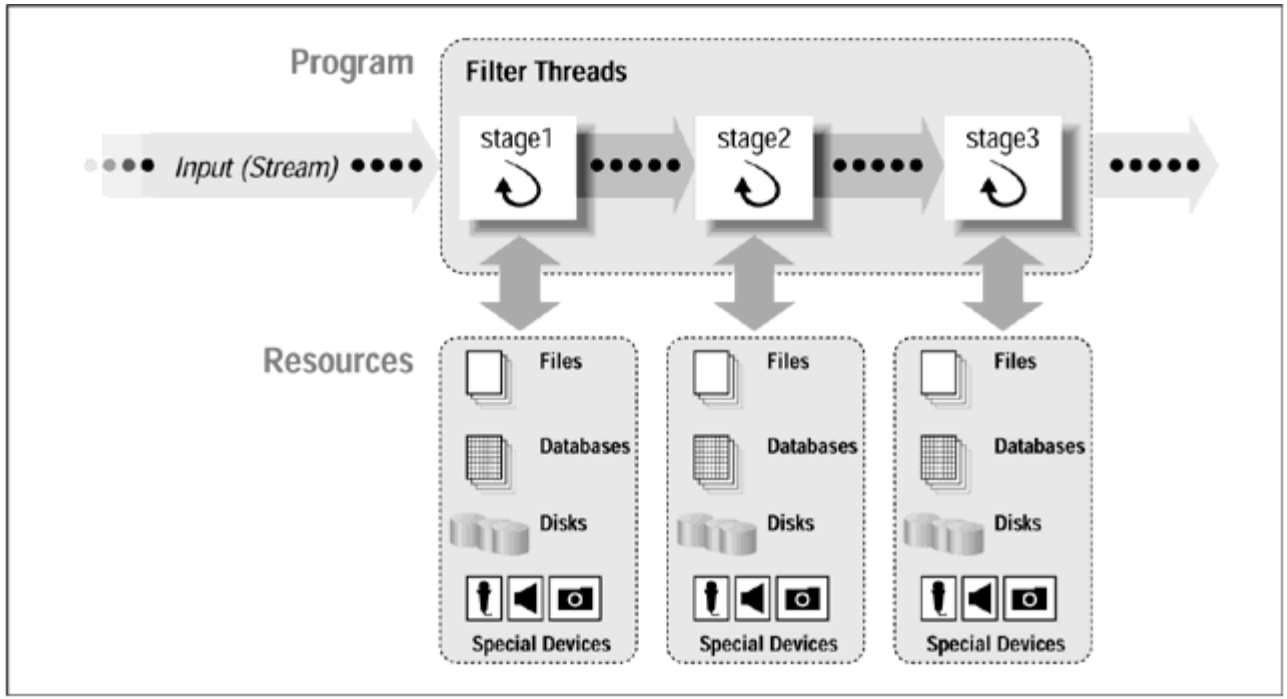
- Upravljalec/delavec
  - Enakovredne niti

```
main()
{
    ustvari potrebne niti, pthread_create(…, zahtevaX,…)
    sproži začetek procesiranja
    počakaj da niti zaključijo in jih pridruži
}

zahtevaX()
{
    počakaj na signal za začetek obdelave
    izvedi obdelavo
    sinhroniziraj dostop do potrebnih virov
}
```

    - Model na mestu za natančno definirane probleme (matematični, fizikalni problemi)
    - Počasna kumunikacija med nitmi upočasni izvajanje programa

Modeli večnitnih programov

- Cevovod
  - Predpostavke
    - Na voljo ogromno vhodnih podatkov
    - Obdelavo lahko razdelimo na več čim bolj enakovrednih stopenj
    - Vsaka stopnja lahko obdeluje drugo zahtevo (podatke)



Modeli večnitnih programov

- Cevovod
  - Prva nit v cevovodu skrbi za vhod celotnega programa
  - Podobno samo zadnja nit skrbi za izhod
  - Vmes si niti predajajo delo
  - Vsaka nit po potrebi dostopa do virov

  - Stopnje morajo biti kar se da enakovredne
  - Hitrost cevovoda je odvisna od najpočasnejše stopnje
  - Lahko imamo za eno stopnjo več niti, ki vsaka obdeluje svoje podatke

Modeli večnitnih programov

- Cevovod

```
main()
{
    ustvari niti za vseh N stopenj cevovoda,
        pthread_create(…, stopnjaX,…)
    počakaj da niti zaključijo
}

stopnja1()
{
    preberi vhodne podatke
    izvedi obdelavo
    predaj rezultate niti za stopnjo 2
}

stopnjaX()
{
    prevzemi podatke od stopnje X-1
    izvedi obdelavo
    predaj rezultate stopnji X+1
}

stopnjaN()
{
    prevzemi podatke od stopnje N-1
    izvedi obdelavo
    shrani rezultate
}
```