Vlákna

- Model vlákna
- Tvorba
- Ukončenie vlákna
- Zrušenie
- Atribúty vlákna
- Špecifické údaje
- Cleanup handlery

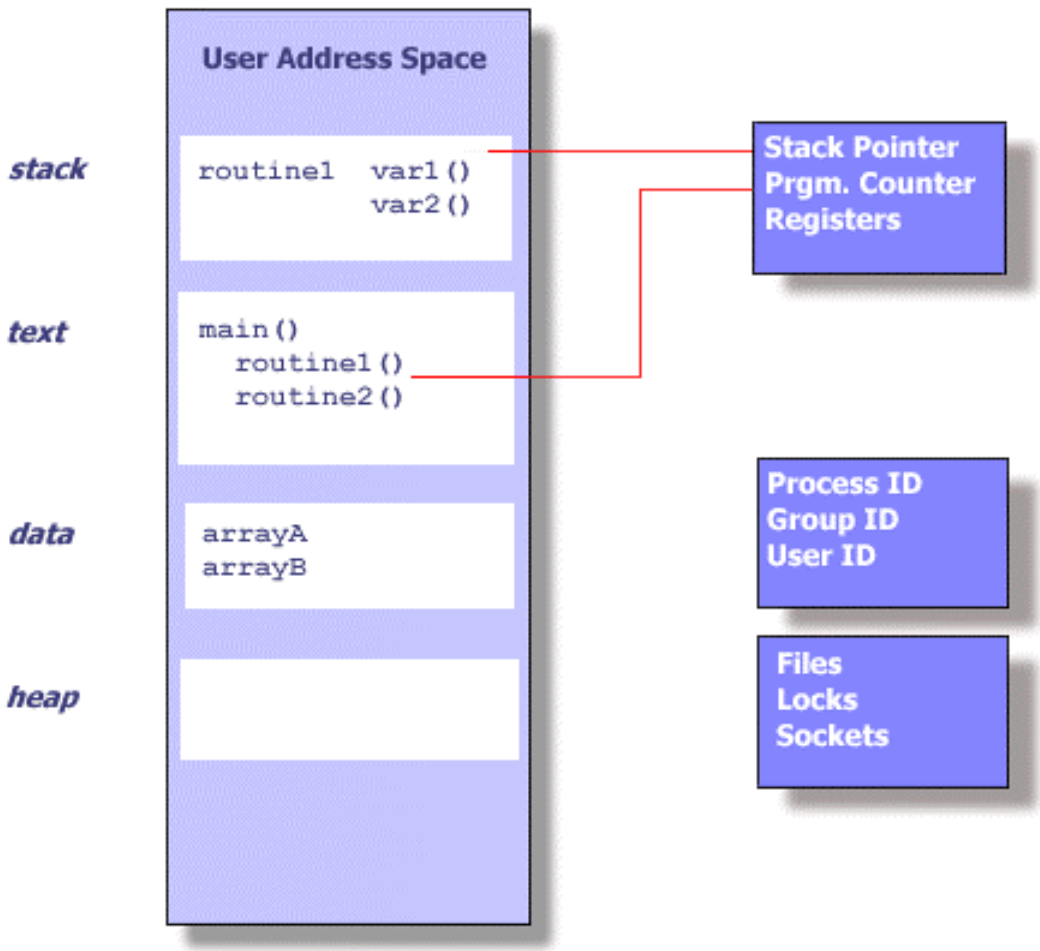
Koncepia vlákien v Linuxe

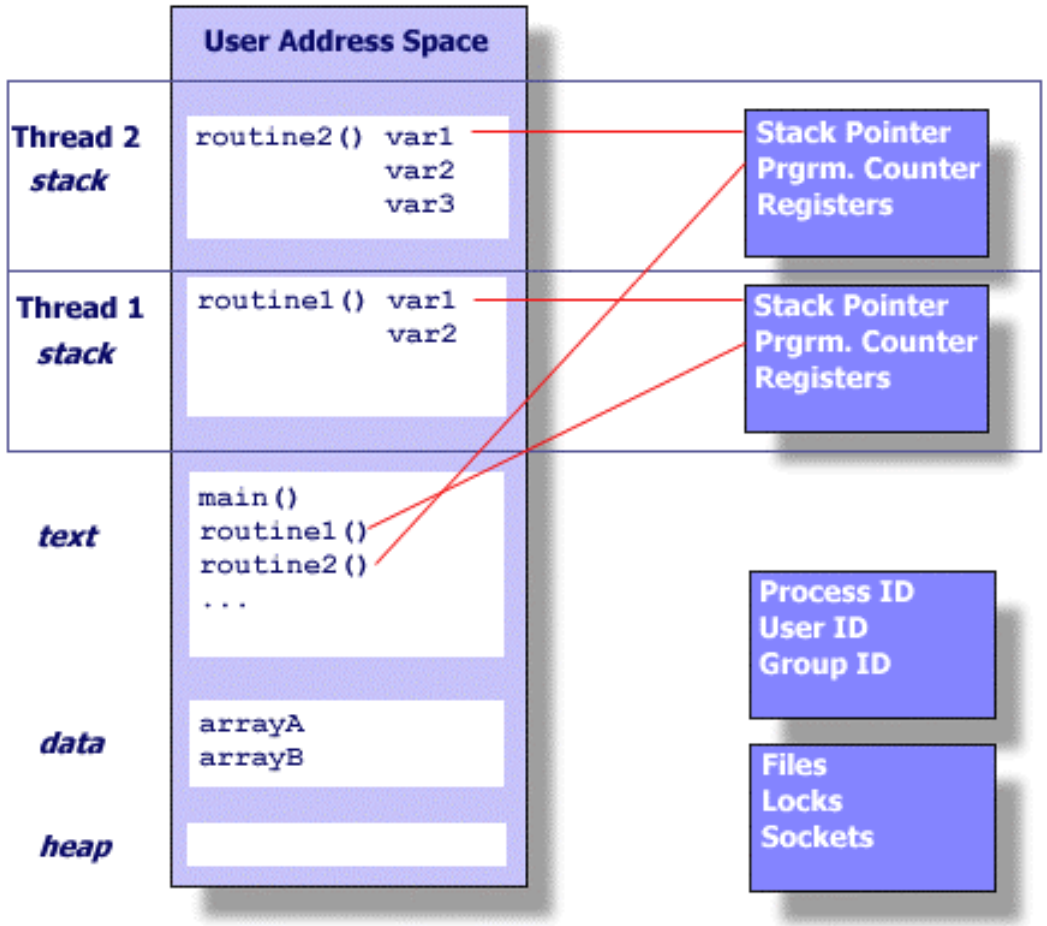
- V procese existuje vždy základné vlákno.
- Po spustení programu Linux vytvori nový proces s jedným vláknom, ktoré vykonáva program sekvenčne.
- Toto vlákno môže vytvárať ďalšie vlákna, ktoré vykonávajú ten istý proces v tom istom programe.
- Vlákna môžu vykonávať odlišné časti programu v tom istom čase.

Procesy a vlákna

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- prvky zdieľané všetkými vláknami procesu
- prvky patriace každému vláknu


User Address Space
stack
routine1 var1()
var2()
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets


User Address Space
Thread 2
stack
routine2() var1
var2
var3
Thread 1
stack
routine1() var1
var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets

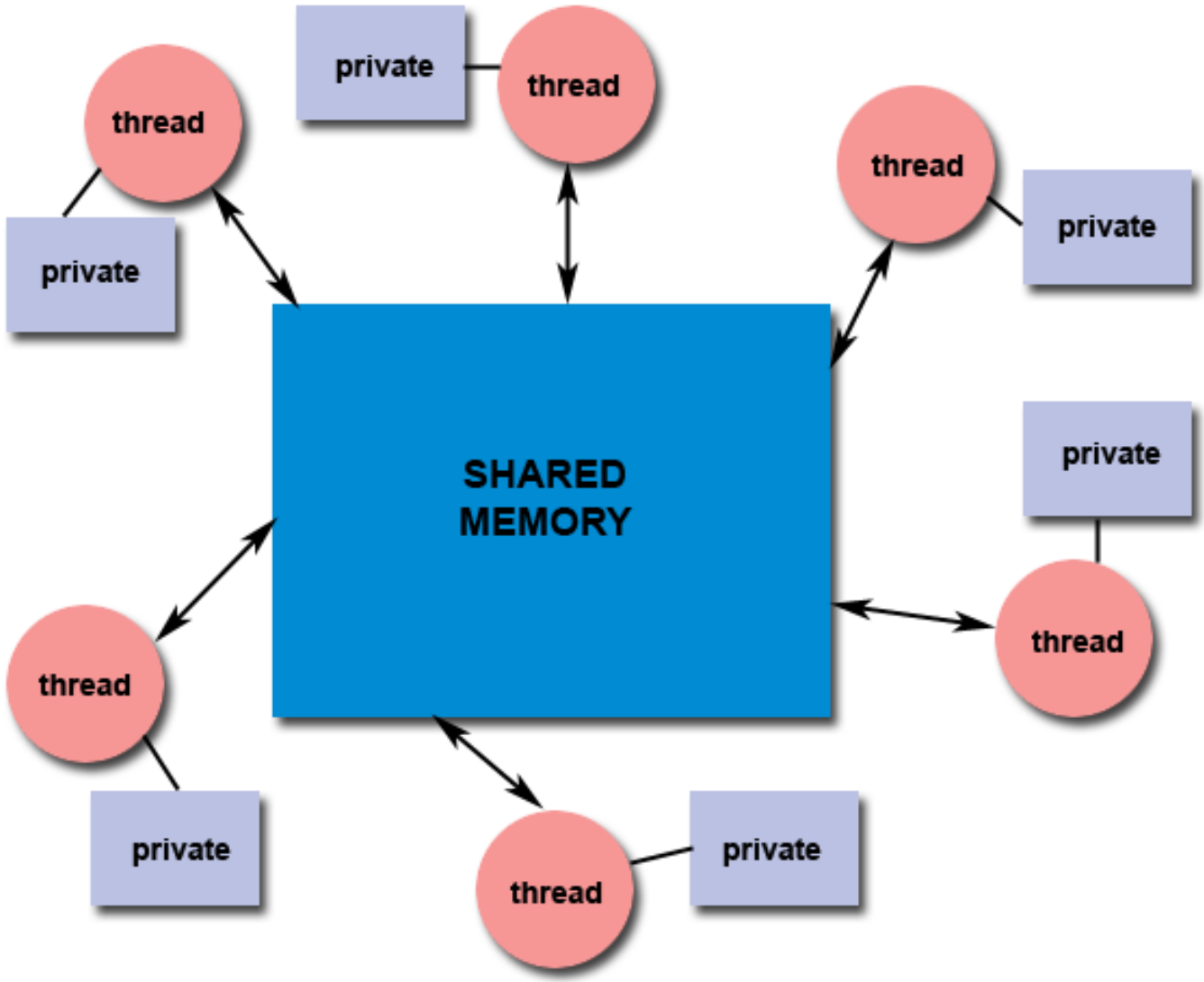
Vlákna

- Vlákna zdieľajú zdroje procesu a preto:

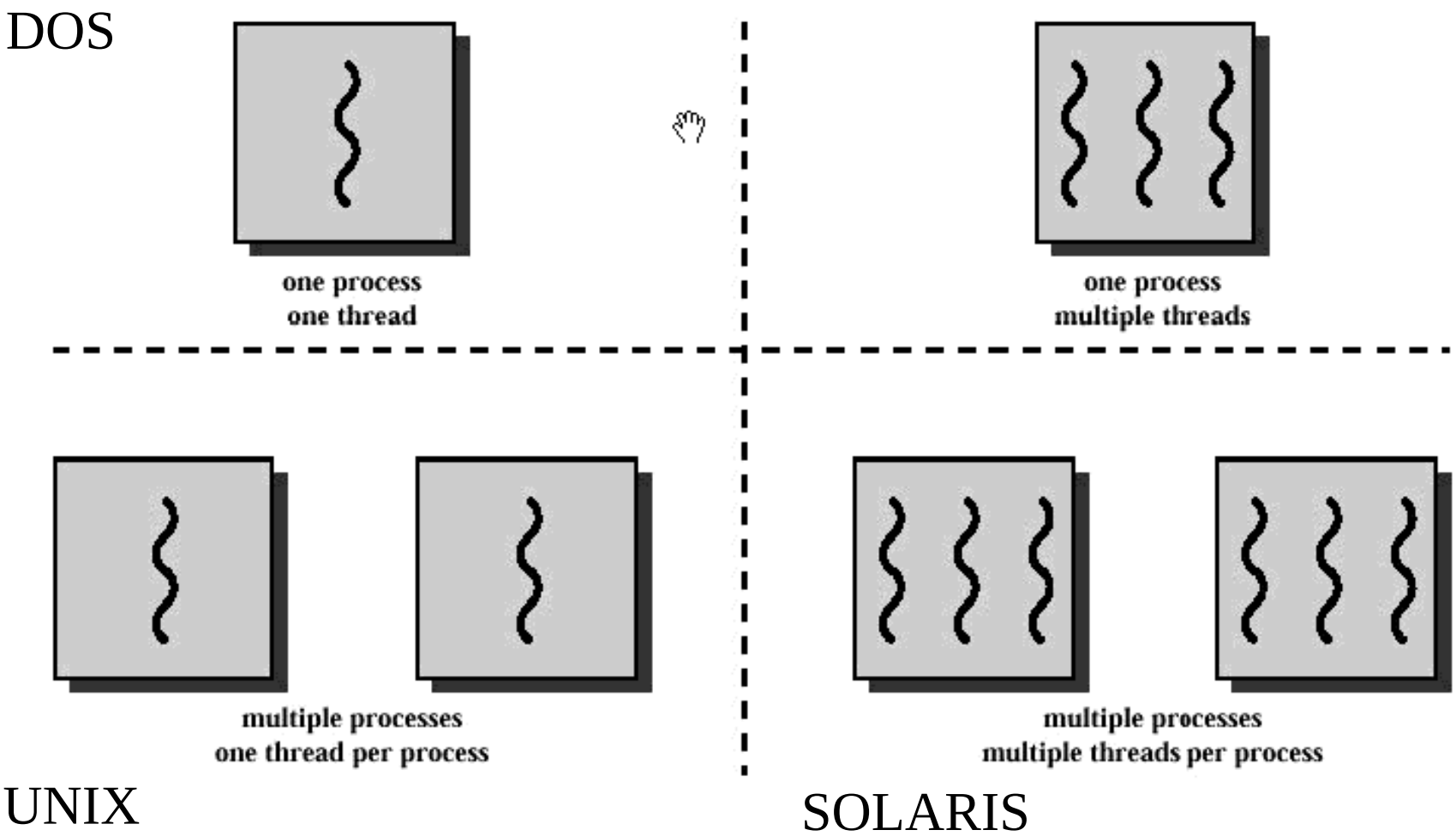
  - každá zmena, ktorú urobí jedno vlákno na systémových zdrojoch (napr. zatvorenie súboru) je viditelná všetkým vláknam

  - dva smerníky (v dvoch rôznych vláknach), ktoré majú rovnakú hodnotu ukazujú na to isté miesto

  - čítanie a zápis na to isté miesto v pamäti rôznymi vláknami musia byť explicitne programátorsky synchronizované


private
thread
thread
private
thread
private
SHARED
MEMORY
private
thread
thread
private
thread
private

Rôzne modely pre vlákna a procesy



Prínos vlákien

- Zvýšenie odozvy aplikácií – príkladom je viac vláknový GUI
- Efektívnejšie využitie multiprocesorovej architektúry – násobenie matíc je rýchlejšie keď je implementované vláknami na multiprocesoroch
- Zlepšujú programovú štruktúru – viacvláknový program je adaptívnejší k rôznym užívateľským požiadavkam
- Menej systémových zdrojov – vlákna na rozdiel od procesov vyžadujú menej času a pamäte na komunikáciu a synchronizáciu

Časové porovnania fork() a pthread_create()

Platforma	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.3 GHz	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz	54.5	1.1	22.2	2.0	1.2	0.6

Tabuľka znázorňuje časy pri tvorbe 50 000 procesov resp. vlákien

C Code for fork() creation test

```
#include <stdio.h>
#include <stdlib.h>
#define NFORKS 50000
void do_nothing() {
  int i;
  i= 0;
}
int main(int argc, char *argv[]) {
  int pid, j, status;
  for (j=0; j<NFORKS; j++) {
  /*** error handling ***/
  if ((pid = fork()) < 0 ) {
  printf ("fork failed with error code= %d\n", pid);
  exit(0);
  }
```

```
  /*** this is the child of the fork ***/
   else if (pid ==0) {
     do_nothing();
     exit(0);
   }
   /*** this is the parent of the fork ***/
   else {
     waitpid(pid, status, 0);
   }
 }
}
```

C Code for pthread_create() test

==========

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS 50000
void *do_nothing(void *null) {
  int i;
  i=0;
  pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
  int rc, i, j,
  detachstate;
  pthread_t tid;
  pthread_attr_t attr;
```

```
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
 for (j=0; j<NTHREADS; j++) {
   rc = pthread_create(&tid, &attr, do_nothing, NULL);
   if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
  }
  /* Wait for the thread */
  rc = pthread_join(tid, NULL);
  if (rc) {
     printf("ERROR; return code from pthread_join() is %d\n", rc);
     exit(-1);
  }
 }
 pthread_attr_destroy(&attr);
 pthread_exit(NULL);
}
```

Porovnanie rýchlosti komunikácie procesov a vlákien

- Procesy komunikujú cez zdieľanú pamäť knižnica MPI, čo si vyžaduje najmenej jednu operáciu kopírovania pamäti
- Vlákna zdieľajú ten istý adresný priestor a preto ich komunikácia si nevyžaduje žiadne prechodné kopírovanie pamäti

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Úrovne vlákien

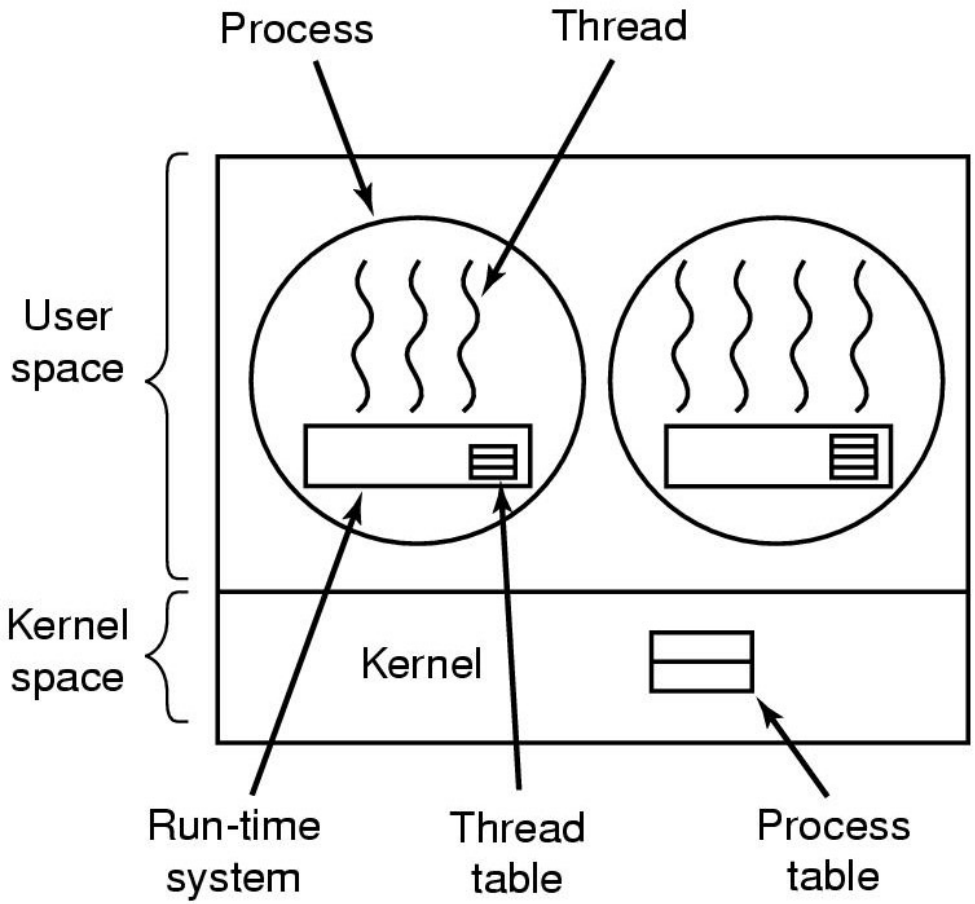
- Knižnica vlákien – User level Threads (ULT)

  Na tejto úrovni jadro nevie nič o existencii vlákien. Správa vlákien je na aplikačnej úrovni a využíva sa knižnica vlákien.

  Nevýhoda je, že jadro blokuje procesy a tým aj všetky vlákna v procese. Tiež, že vlákna toho istého procesu nemôžu súčasne bežať na dvoch procesoroch.
- Systémové volania – Kernel level Threads (KLT)

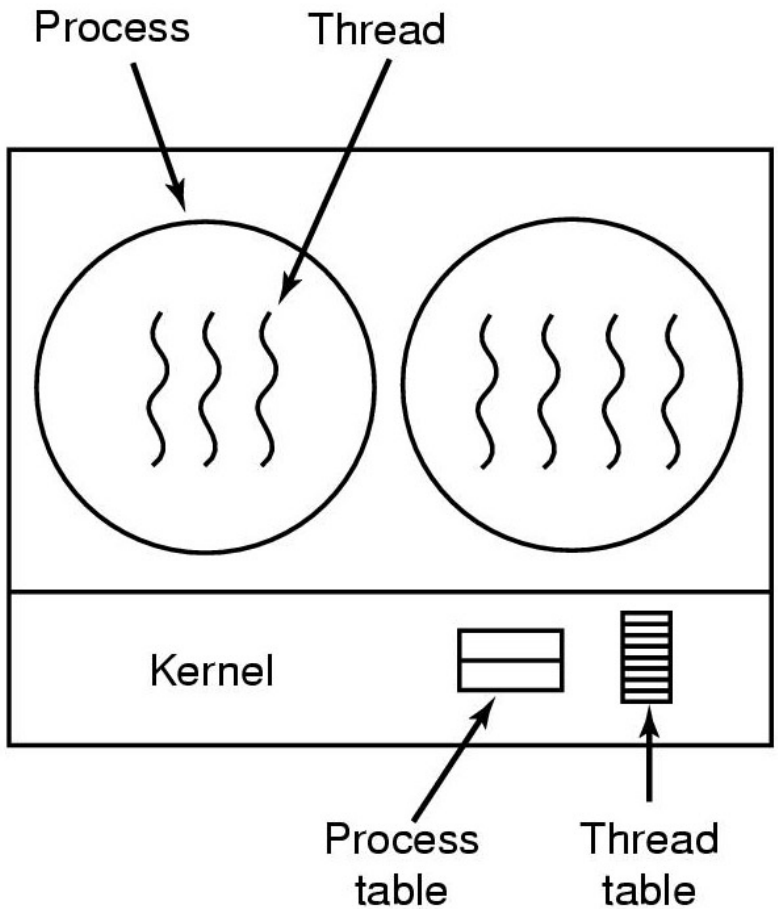
  Na tejto úrovni je správa vlákien vykonávaná jadrom cez API systémové volania. Jadro súčasne môže rozvrhovať viac vlákien jedného procesu na viacej procesoroch. Blokovanie je len na úrovni vlákien. Nevýhodou je, že prepínanie vlákien v rámci jedného procesu si vyžaduje služby jadra čo spomaľuje výpočet.
- Hybridná úroveň ULT/KLT – kombinuje výhody oboch

Implementácia ULT



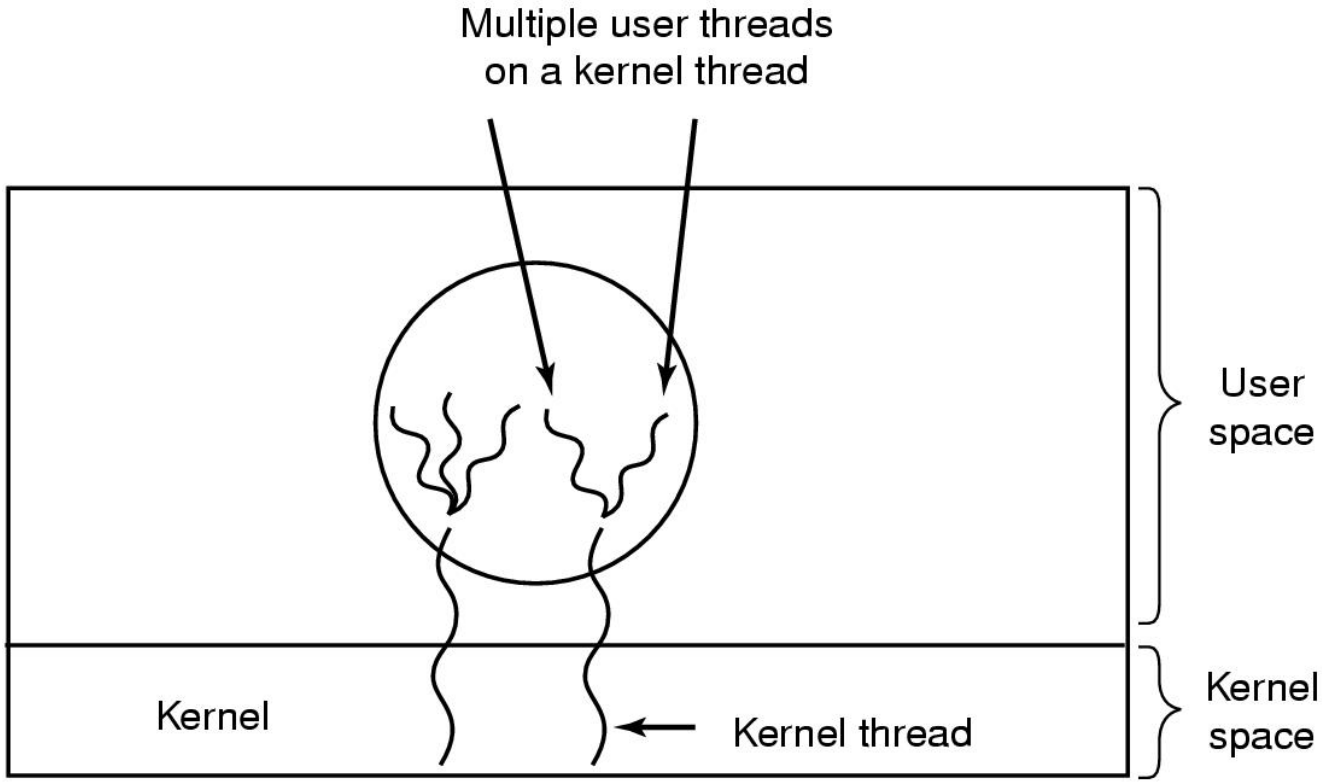
Správa vlákien knižnicou vlákien

Implementácia KLT



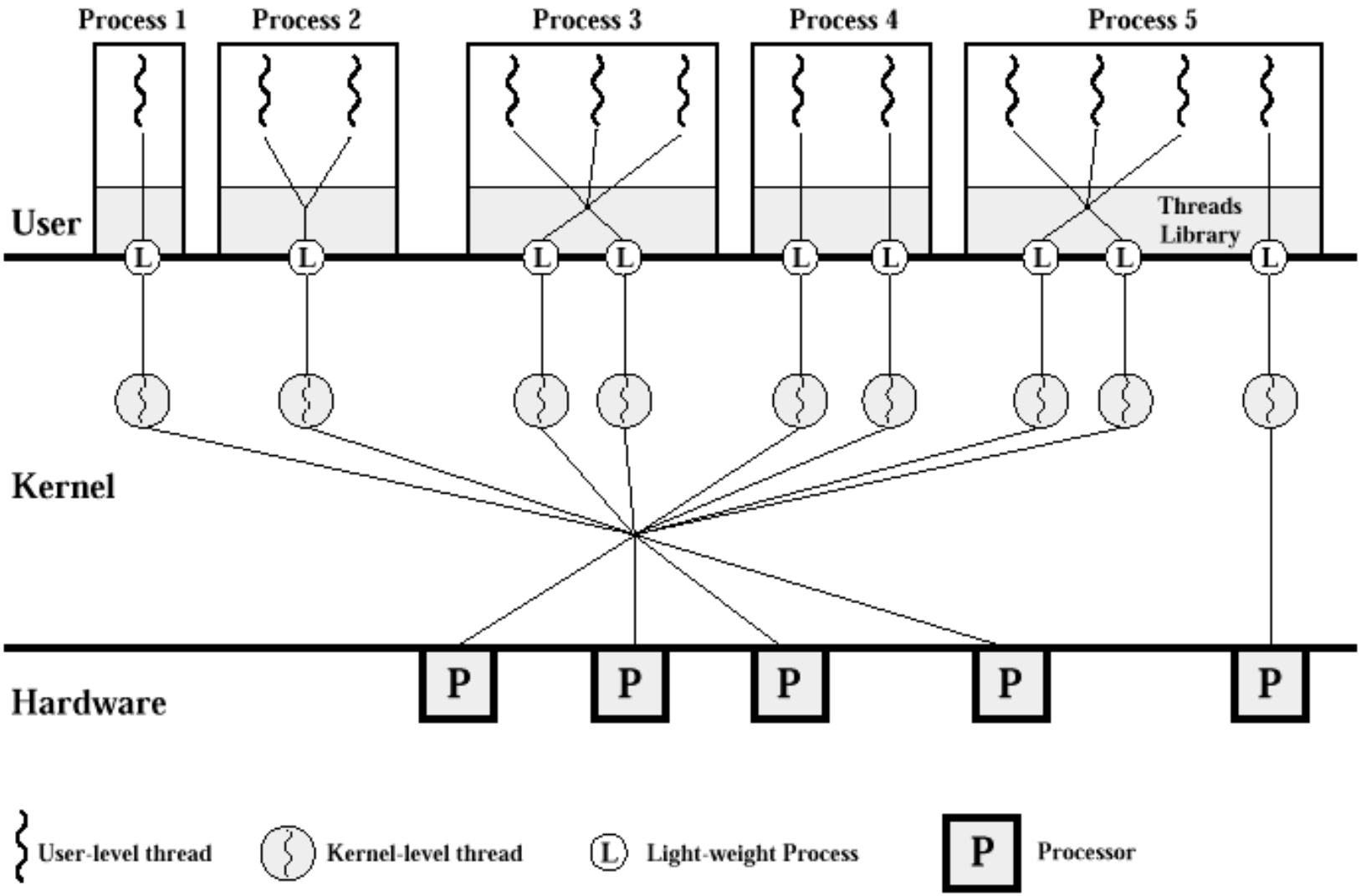
Správa vlákien jadrom OS

Hybridná implementácia



Niekoľko ULT vlákien do KLT vlákna

Hybridná úroveň - Solaris



Knižnica vlákien (Pthreads)

- IEEE POSIX 1003.1 standard (Portable Operating System Interface) – libpthread (pri linkovaní treba pridať – lpthread)
- Solaris – libthread

- Knižnice umožňujú:

  tvoriť a rušiť vlákna

  vymieňať si dáta a správy medzi vláknami

  rozvrhovať vlákna

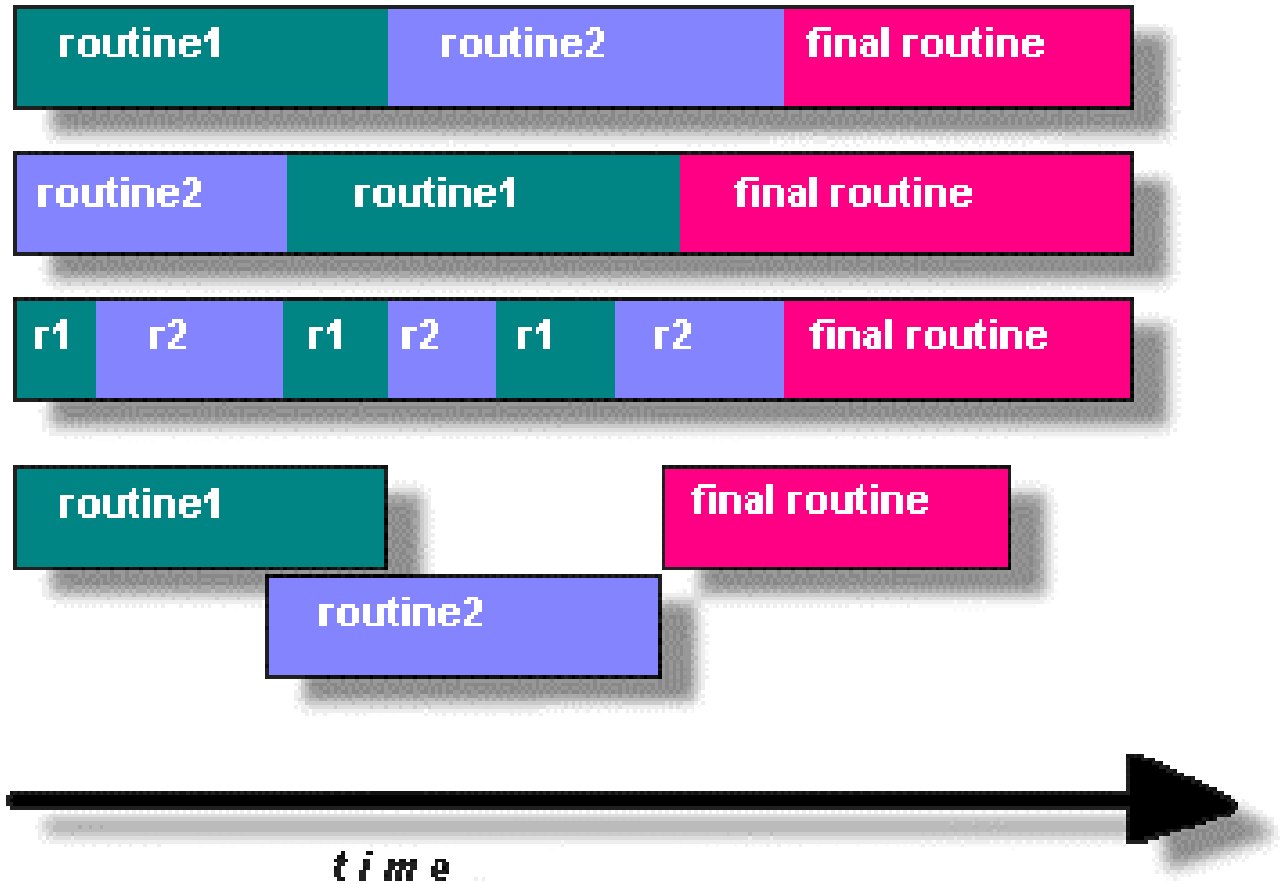
  pamätať a obnovovať kontext vlákna

Kedy je vhodné písať viacvláknový program

- Ak sú jeho podprogramy nezávislé úlohy, ktoré sa môžu vykonávať konkurentne

  Ak napr. poradie volania routine1 a routine2 môžme voľne zamieňať

  alebo sa môžu vykonávať po častiach vzájomným prepínaním (interleaving) potom ich výkon sa môže v čase prekrývať


routine1
routine2
final routine
routine2
routine1
final routine
r1
r2
r1
r2
r1
r2
final routine
routine1
final routine
routine2
time

- Ak je program často dlho blokovaný čakaním na I/O (zatiaľ čo jedno vlákno čaká na I/O iné môže využívať CPU)
- Ak musí reagovať na asynchrónne udalosti (každá obsluha udalosti sa môže vykonávať iným vláknom a tieto sa môžu navzájom prepínať)
- Prioritné plánovanie. Ak sú niektoré úlohy dôležitejšie môžu prerušiť úlohy s nižšou prioritou (prioritné prerušenia)

Tvorba vlákien

- Na tvorbu nového vlákna sa používa funkcia:

  pthread_create (pthread_t *thread_id, pthread_attr_t* thread_attribute, void*(*thread_function)(void *), void *arg)

  kde

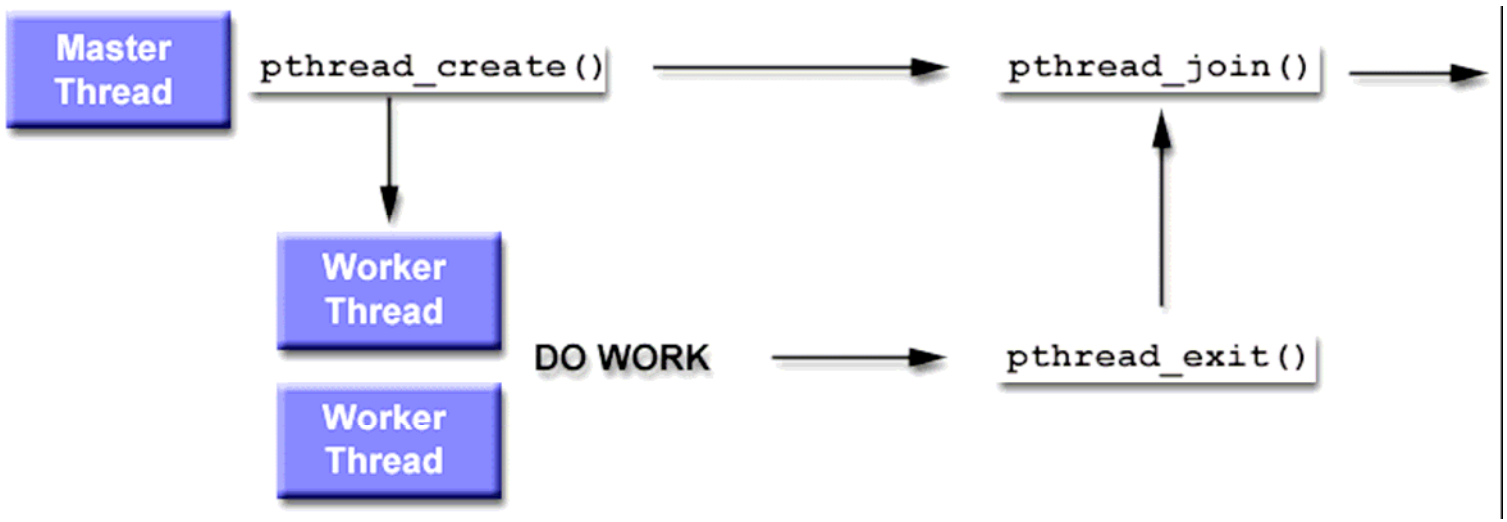
  thread_id je smerník na premennú typu pthread_t , v ktorej je pamätané ID nového vlákna.

  thread_attribute je smerník na štruktúru pthread_attr_t, ak je NULL potom je vlákno vytvorené s defaultovými hodnotami.

  thread_function je smerník na funkciu – vlastné telo vlákna, ktorá sa má vykonať a jej typ je void*(*)(void *).

  arg je smerník typu void, ktorý obsahuje argumenty funkcie vlákna, s ktorými sa bude vykonávať.

Tvorba vlákien



Tvorba vlákien

- Návrat z funkcie **pthread_create** je okamžitý a pôvodné vlákno pokračuje na nasledujúcej inštrukcii, zatiaľ čo nové vlákno začne vykonávať *thread_function.*

  Pr.

```
#include <pthread.h>
#include <stdio.h>
/* Prints x's to stderr. The parameter is unused. Does not return. */
void* print_xs (void* unused)
{
  while (1)
  fputc ('x', stderr);
  return NULL;
}
```

Tvorba vlákien

```
/* The main program. */
int main ()
{
  pthread_t thread_id;

/* Create a new thread. The new thread will run the print_xs
    function. */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Print o's continuously to stderr. */
  while (1)
  fputc ('o', stderr);
  return 0;
}
```

- Výstupom programu je nepredikovateľný reťazec „x“ a „o“ podľa toho ako OS prideľuje vláknam procesor.

Ukončenie vlákien

- Návratom zo thread funkcie – návratová hodnota vlákna je daná návratovou hodnotou thread funkcie.
- Volaním funkcie pthread_exit(void* arg), ktorá sa volá priamo zo thread funkcie alebo nepriamo v akejkoľvek inej funkcie, ktorá sa volá v thread funkcii - návratová hodnota vlákna bude hodnota v arg.

Odovzdanie argumentov funkcii vlákna

- Pre každú thread funkciu (vlákno) sa definuje štruktúra, ktorá obsahuje parametre, ktoré očakáva thread funkcia. Smerník na túto štruktúru sa uvádza ako 4 argument vo funkcii **pthread_create**.
- Týmto spôsobom sa dá jednoducho využiť tá istá thread funkcia pre viacej vlákien tým, že sa vykonáva ten istý kód programu s rôznymi datami.
- Nasledovný program túto možnosť využíva tým, že sa vytvoria dva vlákna, ktoré výkonávajú ten istý kód (tlačia konečný počet charakterov) ale oba vlákna majú rozdielny výstup použitím štruktúry

  struct char_print_parms.

Príklad

```
#include <pthread.h>
#include <stdio.h>
/* Parameters to print_function. */
struct char_print_parms
{
  char character;
  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 30,000 '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 20,000 o's. */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
  return 0;
}
```

Poznámky

- Program je chybný z toho dôvodu, že vlákno v ktorom sa vykonáva funkcia main vytvára argumenty ostatných vlákien ako lokálne premenné thread1_args a thread2_args, čím sa stane to, že ostatné vlákna sa pokúšajú získať svoje parametre z pamäti, ktorá je po ukončení vlákna volajúceho main funkciu už dealokovaná.

Čakajúce vlákna

- Jedným z riešení uvedeného problému je prinútiť vlákno, ktoré obsahuje parametre pre iné vlákna deklarované lokálne, aby čakalo na ukončenie vlákien, ktoré ich používajú.

  Na to slúži funkcia:

  pthread_join (pthrad_t thread_id, void return_value)*

  kde

  prvý argument špecifikuje ID vlákna na ukončenie, ktorého sa čaká

  druhý argument je smerník na premennú, do ktorého sa odpamätá návratová hodnota vlákna. Môže byť aj NULL ak nás návratová hodnota nezaujíma.

```
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 30,000 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 20,000 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;
}
```

Návratová hodnota vlákna

- Ak je druhý argument funkcie pthread_join nie NULL je návratová hodnota vlákna v tomto argumente.

  Pr.

  Nasledovný program počíta v osobitnom vlákne najväčšie prvočíslo. Toto vracia ako návratovú hodnotu vlákna. Zatiaľ čo jedno vlákno počíta prvočíslo môže hlavné vlákno vykonávať iný kód programu.

Príklad

```
#include <pthread.h>
#include <stdio.h>
/* Compute successive prime numbers (very inefficiently). Return the
    Nth prime number, where N is the value pointed to by *ARG. */
void* compute_prime (void* arg)
{
  int candidate = 2;
  int n = *((int*) arg);
  while (1) {
    int factor;
    int is_prime = 1;
    /* Test primality by successive division. */
    for (factor = 2; factor < candidate; ++factor)
    if (candidate % factor == 0) {
        is_prime = 0;
        break;
    }
```

```
    /* Is this the prime number we're looking for? */
    if (is_prime) {
      if (--n == 0)
      /* Return the desired prime number as the thread return  value. */
      return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}
```

```
int main ()
{
  pthread_t thread;
  int which_prime = 5000;
  int prime;
  /* Start the computing thread, up to the 5,000th prime
    number. */
  pthread_create (&thread, NULL, &compute_prime,
    &which_prime);
  /* Do some other work here... */
  /* Wait for the prime number thread to complete, and get
    the result. */
  pthread_join (thread, (void*) &prime);
  /* Print the largest prime it computed. */
  printf("The %dth prime number is %d.\n", which_prime,
    prime);
  return 0;
}
```

Poznámky k návratovej hodnote

- Tento spôsob konverzie návratovej hodnoty vlákna typ int na (void *) a opačne vo funkcii *pthread_join* nemusí byť portabilný
- Pri návratovej hodnote funkcii vlákna sa treba vyvarovať

  - návratu smerníka na lokálny objekt (štruktúra, pole, atď.). Po skončení vlákna ukazuje na nedefinovanú oblasť. Príklad:

```
void * thread_function ( void *)
{ char buffer [ 64 ] ;
      // Fill up the buffer with something good.
      return buffer ;
}
```

- Ak deklarujeme buffer ako static tomuto sa vyhneme, ale ak funkciu vlákna volá niekoľko vlákien tak sa hodnoty budú prepisovať:

```
void * thread_function ( void *)
{
        static char buffer [ 64 ] ;
        // Fill up the buffer with something good.
        return buffer ;
}
```

- Nasledovná verzia vrátenia návratovej hodnoty funkcie vlákna bude korektná aj keď ju vykonáva viac vlákien:

```
void * thread_function ( void *)
{
    char *buffer =(char *)malloc( 64) ;
    // Fill up the buffer with something good.
    return buffer ;
}
```

- Ak je vo funkcii vlákna dynamicky alokovaná pamäť potom ju rodičovské vlákno musí dealokovať (pomocou free()) – dealokácia sa nerobí automaticky po skončení vlákna.

```
void * exit_status;
// Wait for the thread to terminate.
pthread_join ( thread_ID , &exit_status) ;
char * thread_result;
thread_result = ( char *) exit_status;
printf ( "I got %s back from the thread .\n" , thread_result );
free ( exit_status);
```

Funkcie *pthread_self* a *pthread_equal*

- *pthread_self* vracia ID vlákna, v ktorom sa volá

  pthread_equal(pthrad_t thread_id1, pthread_t thread_id2)

  porovnáva ID dvoch vlákien. Ak sú totožné vracia 0.
- Funkcie sú užitočné pri určovaní či ID nejakého vlákna nie je napr. ID vlákna, ktoré volá funkciu pthread_join. Je chybou keď vlákno volá pthread_join s vlastným ID. Tomu možno zabrániť kódom:

  Pr.

  if (!pthread_equal (pthread_self (), other_thread)) pthread_join (other_thread, NULL);

Atribúty vlákna

- Používajú sa na definovanie vlastností vlákna.
- Sú 2 argumentom vo funkcii *pthread_create.*
- Aby sa vytvorilo vlákno s danými vlastnosťami treba prispôsobiť jeho argumenty nasledovne:
  1. Vytvoriť pthread_attr_t premennú (štruktúru).
  2. Volať funkciu *pthread_attr_init()* s argumentom smerník na štruktúru.
  3. Modifikovať atribúty štruktúry na požadované hodnoty.
  4. Vytvoriť vlákno s danými atribútmi.
  5. Uvolniť štruktúru atribútov volaním funkcie *pthread_attr_destroy().* Premenná pthread_attr_t sa tým nezruší môže byť neskôr opäť inicializovaná.
- Tá istá štruktúra atribútov sa môže použiť pre niekoľko vlákien.

Tabuľka atribútov vlákna

Atribute	Hodnota	Význam
scope	pthread_scope _process	Nové vlákno je nie trvalo pripojené k LWP
detachstate	pthread_create _joinable or _datached	Po skončení sa zachová výstupný stav aj vlákno alebo po skončení je automaticky zrušené
stackaddr	NULL	Vlákno má systémom alokovanú adresu stacku
stacksize	1 megabyte	Veľkosť stacku je definovaná systémom
inheritsched	pthread_inherit _sched	Vlákno zdedí prioritu rodiča
schedpolicy	sched_other	Vlákno má definovanú fixnú prioritu- Solaris

Atribút vlákna detachstate

- Pre aplikácie GNU-Linux je to jediný zaujímavý atribút. Ostatné sú predovšetkým pre programovanie v reálnom čase.
- Vlákno môže byť vytvorené ako:
  1. joinable thread (defaultovo)
  2. detached thread

  V prvom prípade po ukončení, nie je vlákno automaticky zrušené ale jeho výstupný stav „čaká“ v systéme (ako zombie proces) pokiaľ iné vlákno volaním funkcie pthread_join nezíska jeho návratovú hodnotu.

  V druhom prípade je po skončení vlákno automaticky zrušené a iné vlákno už nemôže získať jeho návratovú hodnotu a ani sa nemôže s ním synchronizovať volaním *pthread_join()*.
- Pozn.

  Vlákno, ktoré je vytvorené ako joinable môže byť neskôr zmenené na detach funkciou *pthread_detach* ale naopak nie!!

Funkcia pthread_attr_setdetachstate()

- Funkcia je na nastavenie atribútu detach state - má 2 argumenty.

  prvý argument je smerník na štruktúru atribútov

  druhý je PTHREAD_CREATE_DETACHED

  alebo defaultovo

  PTHREAD_CREATE_JOINABLE

Príklad

```
#include <pthread.h>
void* thread_function (void* thread_arg)
{
  /* Do work here... */
}
int main ()
{
  pthread_attr_t attr;
  pthread_t thread;
  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);
  /* Do work here... */
  /* No need to join the second thread. */
  return 0;
}
```

Zrušenie vlákna

- Za normálnych okolností vlákno skončí :
  1. návratom z funkcie vlákna
  2. volaním funkcie *pthread_exit*
  3. alebo jeho zrušením iným vláknom volaním funkcie *pthread_cancel( )* s argumentom ID vlákna, ktoré sa má zrušiť
- Zrušené vlákno môže byť neskôr synchronizované ak nie je vytvorené ako detached.
- Návratová hodnota zrušeného vlákna je PTHREAD_CANCELED.

Stavy vlákna vzhľadom na jeho zrušenie

- Vlákno môže byť zrušené **asynchrónne** – kedykoľvek počas výkonu
- Vlákno môže byť zrušené **synchrónne** – požiadavky na zrušenie sú vo fronte a vlákno je zrušené len ak dosiahne špecifikované miesto počas výkonu.

  Dôvod je nasledovný: vlákno môže alokovať nejaké zdroje a pokiaľ ich neuvolní nemalo by byť zrušené, inak ostanú v systéme „vysieť“ ako obsadené.
- Vlákno nemôže byť zrušené.
- Počiatočný stav vlákna:

  vlákno môže byť zrušené **synchrónne.**

Synchrónne a asynchrónne vlákna

- Miesta, v ktorých možno vlákno zrušiť sú body zrušenia (**cancelation points)**
- Ako vytvoriť a kde umiestniť body zrušenia?
- Body sa tvoria volaním funkcie: *pthread_testcancel* v takých miestach vlákna (ak je synchrónne zrušitelné), v ktorých zrušenie vlákna nespôsobí neuvolnenie zdrojov, ktoré používa.

Cancelation points (body zrušenia)

- Tiež všetky nasledovné funkcie, ktoré sa volajú vo funkcii vlákna tvoria body zrušenia

  pthread_join()

  pthread_cond_wait()

  pthread_cond_timedwait()

  sem_wait()

  sigwait()

Synchrónne a asynchrónne vlákna

- Vlákno sa stane asynchrónne zrušitelné volaním funkcie:

  pthread_setcanceltype (PTHREAD_CANCEL ASYNCHRONOUS, int* ptr)
- Návrat vlákna do synchróneho stavu:

  pthread_setcanceltype (PTHREAD_CANCEL DEFERRED, int* ptr)

  ptr je smerník (alebo NULL) na premennú, ktorá obsahuje predchádzajúci stav vlákna.

Nezrušitelné kritické sekcie

- Sú sekcie, v ktorých nemožno zrušiť vlákno.
- Tvoria sa volaním funkcie:

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, int* ptr)

  a ukončenie sekcie volaním:

  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, int* ptr)
- ptr má význam ako pri funkcii *pthread_setcanceltype*

Nezrušitelné kritické sekcie

- Funkcie umožnia implementovať kritické sekcie
- Kritická sekcia je postupnosť kódu programu, ktorý musí byť vykonaný celý
- Nasledovný kód ukazuje príklad kritickej sekcie bankového prevodu z účtu na účet.

  Sekcia medzi odobraním hodnoty z jedného účtu a pridaním tej istej hodnoty na druhý účet je kritická sekcia – vlákno, ktoré ju vykonáva nemôže byť v tejto sekcii **zrušené**.

Príklad

Protect a Bank Transaction with a Critical Section

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
/* An array of balances in accounts, indexed by account
   number. */
float* account_balances;
/* Transfer DOLLARS from account FROM_ACCT to
   account TO_ACCT. Return 0 if the transaction
   succeeded, or 1 if the balance FROM_ACCT is too
   small. */
```

```
int process_transaction (int from_acct, int to_acct, float dollars)
{
  int old_cancel_state;
  /* Check the balance in FROM_ACCT. */
  if (account_balances[from_acct] < dollars)
    return 1;
  /* Begin critical section. */
  pthread_setcancelstate (PTHREAD_CANCEL_DISABLE,
     &old_cancel_state);
  /* Move the money. */
  account_balances[to_acct] += dollars;
  account_balances[from_acct] -= dollars;
  /* End critical section. */
  pthread_setcancelstate (old_cancel_state, NULL);
  return 0;
}
```

- Poznámka k programu

  Obnovenie predchádzajúceho stavu je výhodnejšie urobiť podmienene ako nepodmienene nastaviť PTHRED_CANCEL_ENABLE, z toho dôvodu, že funkcia *process_transaction* sa potom môže volať aj z inej kritickej sekcie.

Špecifické premenné vlákna (Thread Specific Data - TSD)

- Na rozdiel od procesov zdieľajú vlákna v programe ten istý adresný priestor (globálne premenné). Keďže majú vlastný stack môže vykonávať každé vlákno odlišný kód programu a volať podprogramy bežným spôsobom. Každé volanie podprogramu v každom vlákne má svoje vlastné lokálne premenné, ktoré sa pamätajú do stacku vlákna.
- Vlákno si však môže vytvoriť svoje špecifické premenné. Sú to premenné, ktorých kópiu vlastní každé vlákno osobitne. Pretože zdieľajú všetky vlákna ten istý pamäťový priestor na tieto premenné (thread-specific data) sa nemožno odvolávať ako na normálne premenné.
- Linux poskytuje špeciálne funkcie pre prácu s nimi.

TSD

- Je pamäťový blok privátny vláknu
- TSD asociuje hodnoty typu (void *) ku TSD kľúču
- Kľúč je spoločný pre všetky vlákna ale jeho hodnota je v každom vlákne iná
- Vlákno môže vytvoriť viac TSD pomocou viac kľúčov
- TSD možno chápať ako pole void smerníkov, ktorého prvky sa indexujú kľúčmi a hodnota kľúča je hodnota zodpovedajúceho prvku poľa

Tvorba TSD

- TSD sú typu void*
- odvolávame sa na ne pomocou kľúča, ktorý je globálny pre všetky vlákna procesu
- počiatočná hodnota kľúča je pre každé vlákno NULL
- Tvorba:

  int pthread_key_create (pthread_key_t key, cleanup_function)*

  Prvým argumentom je smerník kľúča typu *pthread_key_t*, ktorého hodnota je špecifická pre každé vlákno.

  Druhým argumentom je funkcia (deštruktor).

  Ak je nie potrebné volať deštruktor argument je NULL.

Deštruktor asociovaný ku kľúču

- je v druhom argumente funkcie *pthread_key_create* a jeho argument je typu (void *)
- ak je vlákno ukončené alebo zrušené deštruktor sa automaticky volá s argumentom, ktorého hodnota je asociovaná ku kľúču
- obyčajne je ku kľúču asociovaný smerník na oblasť pamäti alokovanú dynamicky v tom prípade deštruktor slúži na dealokáciu pamäti

TSD - ďalšie funkcie

- *int pthread_setspecific (pthread_key_t key, void* thread_specific_data) -* nastavenie špec. premenej odpovedajúcejej kľúču
- *void* pthread_getspecific (pthread_key_t key)* – návratová hodnota obsahuje hodnotu špecifickej premenej pre daný kľúč

Príklad

- Predpokladajme, že úlohu môžno rozdeliť do vlákien
- Každé vlákno má osobitný súbor, do ktorého sa zaznamenávajú informačné záznamy o vývoji vlákna
- Špecifická premenná vlákna je miesto, kde sa odpamätá smerník na súbor pre každé vlákno

- V main funkcii sa vytvorí kľúč na odpamätanie smerníka na súbor , ktorý sa zapamätá do globálnej premennej thread_log_key, ktorú zdieľajú všetky vlákna
- Vo funkcií vlákna sa otvorí súbor a podľa kľúča sa doň odpamätá smerník na súbor
- Neskôr každé vlákno volaním funkcie write_to_thread_log zapíše správu do špecifického súboru vlákna

Príklad

```
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>
/* The key used to associate a log file pointer with each thread. */
static pthread_key_t thread_log_key;
/* Write MESSAGE to the log file for the current thread. */
void write_to_thread_log (const char* message)
{
  FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
  fprintf (thread_log, “%s\n”, message);
}
/* Close the log file pointer THREAD_LOG. */
void close_thread_log (void* thread_log)
{
  fclose ((FILE*) thread_log);
}
```

```
void* thread_function (void* args)
{
  char thread_log_filename[20];
  FILE* thread_log;
  /* Generate the filename for this thread's log file. */
  sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());
  /* Open the log file. */
  thread_log = fopen (thread_log_filename, "w");
  /* Store the file pointer in thread-specific data under thread_log_key. */
  pthread_setspecific (thread_log_key, thread_log);
  write_to_thread_log ("Thread starting.");
  /* Do work here... */
  return NULL;
}
```

```
int main ()
{
  int i;
  pthread_t threads[5];
  /* Create a key to associate thread log file pointers in
  thread-specific data. Use close_thread_log to clean up the file
  pointers. */
  pthread_key_create (&thread_log_key, close_thread_log);
  /* Create threads to do the work. */
  for (i = 0; i < 5; ++i)
  pthread_create (&(threads[i]), NULL, thread_function, NULL);
  /* Wait for all threads to finish. */
  for (i = 0; i < 5; ++i)
  pthread_join (threads[i], NULL);
  return 0;
}
```

- Poznámka k programu

  thread_function otvorí súbor ale ho nemusí uzavrieť pretože funkcia ktorá súbor uzavrie sa volá ako cleanup funkcia, vždy po exite z vlákna.

Cleanup handlery

- sú funkcie, ktoré sú volané keď vlákno volá pthread_exit(), je zrušené (canceled) alebo vlákno volá *pthread_cleanup_pop* () s nenulovým arg. za účelom uvolnenia zdrojov.
- Za normálnych okolností, keď vlákno skončí návratom z funkcie vlákna sú alokované zdroje automaticky dealokované.

- Cleanup handler sa registruje volaním
  pthread_cleanup_push(cleanup function, void arg)*
- Zrušenie registrácie:
  pthread_cleanup_pop(int flag)
  Ak je argument nulový registrácia funkcie cleanup je zrušená ak je nenulový automaticky sa volá cleanup funkcia.

Príklad1

- Nasledovný fragment programu ukazuje ako sa využíva cleanup handler pre dynamicky alokovanú pamäť pri skončení alebo zrušení vlákna

```
#include <malloc.h>
#include <pthread.h>
/* Allocate a temporary buffer. */
void* allocate_buffer (size_t size)
{
  return malloc (size);
}
/* Deallocate a temporary buffer. */
void deallocate_buffer (void* buffer)
{
  free (buffer);
}
```

```
void do_some_work ()
{
  /* Allocate a temporary buffer. */
  void* temp_buffer = allocate_buffer (1024);
  /* Register a cleanup handler for this buffer, to deallocate it in
  case the thread exits or is cancelled. */
  pthread_cleanup_push (deallocate_buffer, temp_buffer);
  /* Do some work here that might call pthread_exit or might be
  cancelled... */
  /* Unregister the cleanup handler. Because we pass a nonzero value,
  this actually performs the cleanup by calling deallocate_buffer. */
  pthread_cleanup_pop (1);
}
```

Príklad2

- Ako uzamknuť zámku takým spôsobom, že v prípade ak je vlákno zrušené a zamka je uzamknutá bude odomknutá

  pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);

  pthread_mutex_lock(&mut);

  /* do some work */

  pthread_mutex_unlock(&mut);

  pthread_cleanup_pop(0);
- Dôvod: Zrušené vlákno už neodomkne uzamknutú zámku a vlákna, ktoré na nej čakajú nemôžu skončiť

- Uvedený spôsob je bezpečný len ak je vlákno synchrónne zrušitelné
- Ak je vlákno asynchrónne zrušitelné požiadavka na zrušenie sa akceptuje ihneď po je doručení

  Uvedený kód je bezpečný len ak sa požiadavka vyskytne medzi

  pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);

  pthread_mutex_lock(&mut);

  alebo

  pthread_mutex_unlock(&mut);

  pthread_cleanup_pop(0);

- Aby bol uvedený kód bezpečný aj pre asynchrónne zrušitelné vlákna je treba zmeniť typ zrušenia na synchrónny

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,
&oldtype);
pthread_cleanup_push(pthread_mutex_unlock, (void *)
&mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
pthread_setcanceltype(oldtype, NULL);
```