Vlákna a lehké procesy

Threads and Lightweight Processes

Mnoho aplikací se skládá z vykonání několika úkolů, které se nemusí vykonávat po sobě v určeném pořadí, ale pracují se společnými prostředky.

Tomu neodpovídá sekvenční program.

V tradičních systémech UNIX, takové aplikace používají více (paralelních) procesů.

Server aplikace

- proces přijímač čeká na požadavek klienta
- po příchodu požadavky vykoná **`fork`** a vytvoří proces pro jeho obsluhu
- na multiprocesorových systémech paralelní vykonávání
- na jednoprocesorových systémech, zlepšení jestliže obsluha požadavku vyžaduje V/V operaci ve srovnání se sekvenčním programem

Matematické výpočty – výpočet různých maticových operací

- výpočty mohou být na sobě nezávislé
- můžeme vytvořit proces pro výpočet každého prvku
- na multiprocesorových systémech paralelní vykonání
- na jednoprocesorových systémech zlepšení např. když při výpočtu nějaké nezávislé části nastane výpadek stránky

Použití více procesů v jedné aplikaci přináší vysokou režii na vytváření procesů a jejich správu.

Protože každý proces má svůj vlastní adresový prostor musí se použít prostředky meziprocesové komunikace – roury a sdílená paměť.

vlákno je relativně nezávislá část aplikace vykonávaná sekvenčne, která má jeden tok řízení a může být plánována operačním systémem, existuje v procesu a může používat jeho prostředky

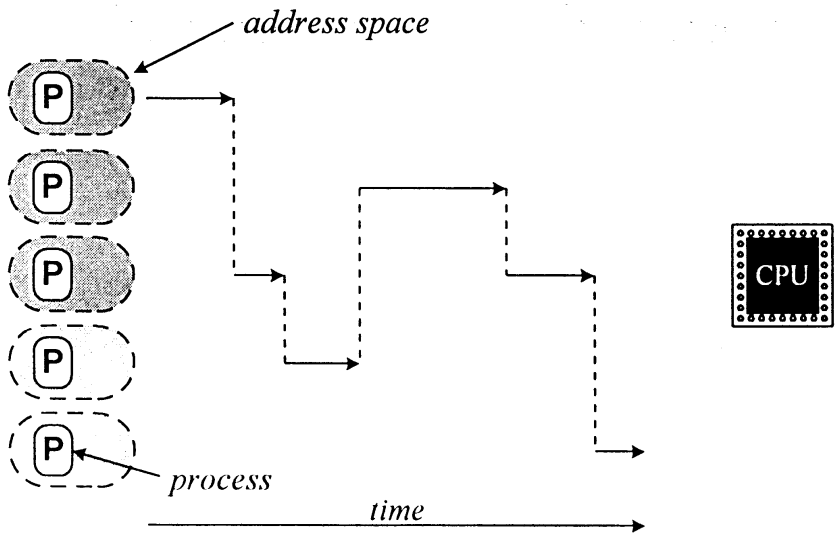
proces může mít jedno nebo více vláken, vykonávaných v tomtéž adresovém prostoru a sdílejících prostředky procesu, např. soubory

každé vlákno má vlastní zásobník, počítadlo instrukcí a HW kontext

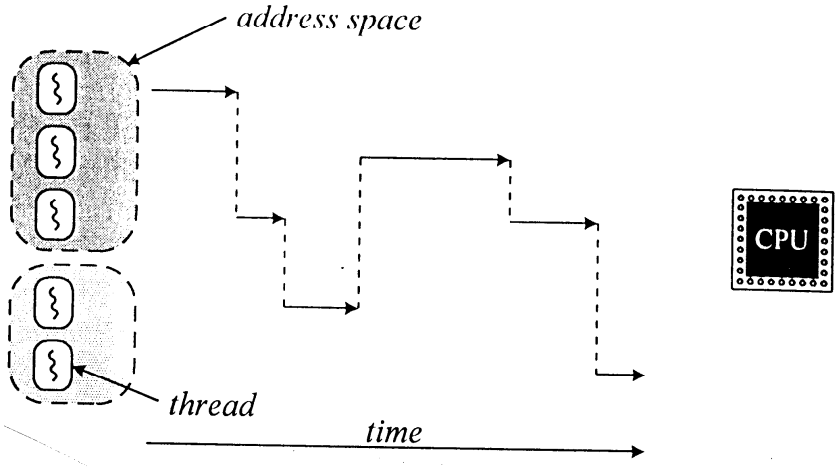
efektivní řešení v porovnání s procesy, vyžaduje však synchronizaci na úrovni vláken

tradiční procesy v sytému UNIX byly jednovláknové, celý výpočet procesu byl sekvenční

dva servery

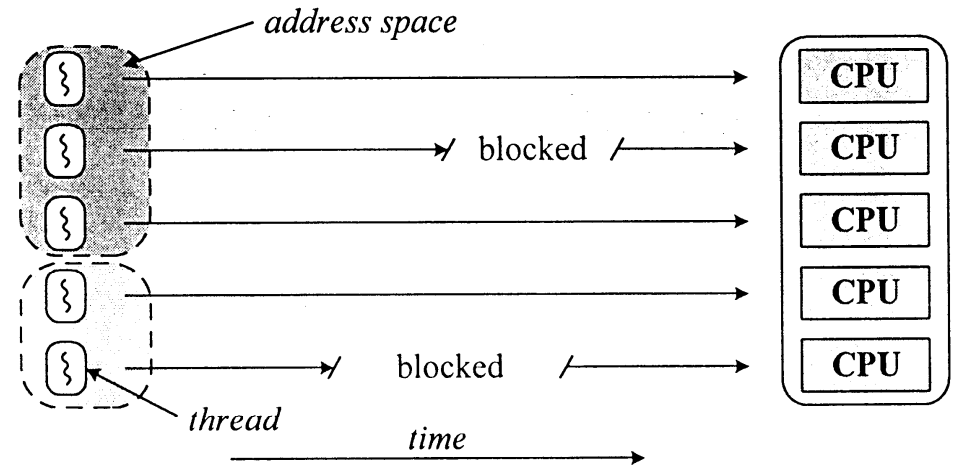


Traditional UNIX system—uniprocessor with single-threaded proces



Multithreaded processes in a uniprocessor system.

nutnost synchronizace vláken v procesech



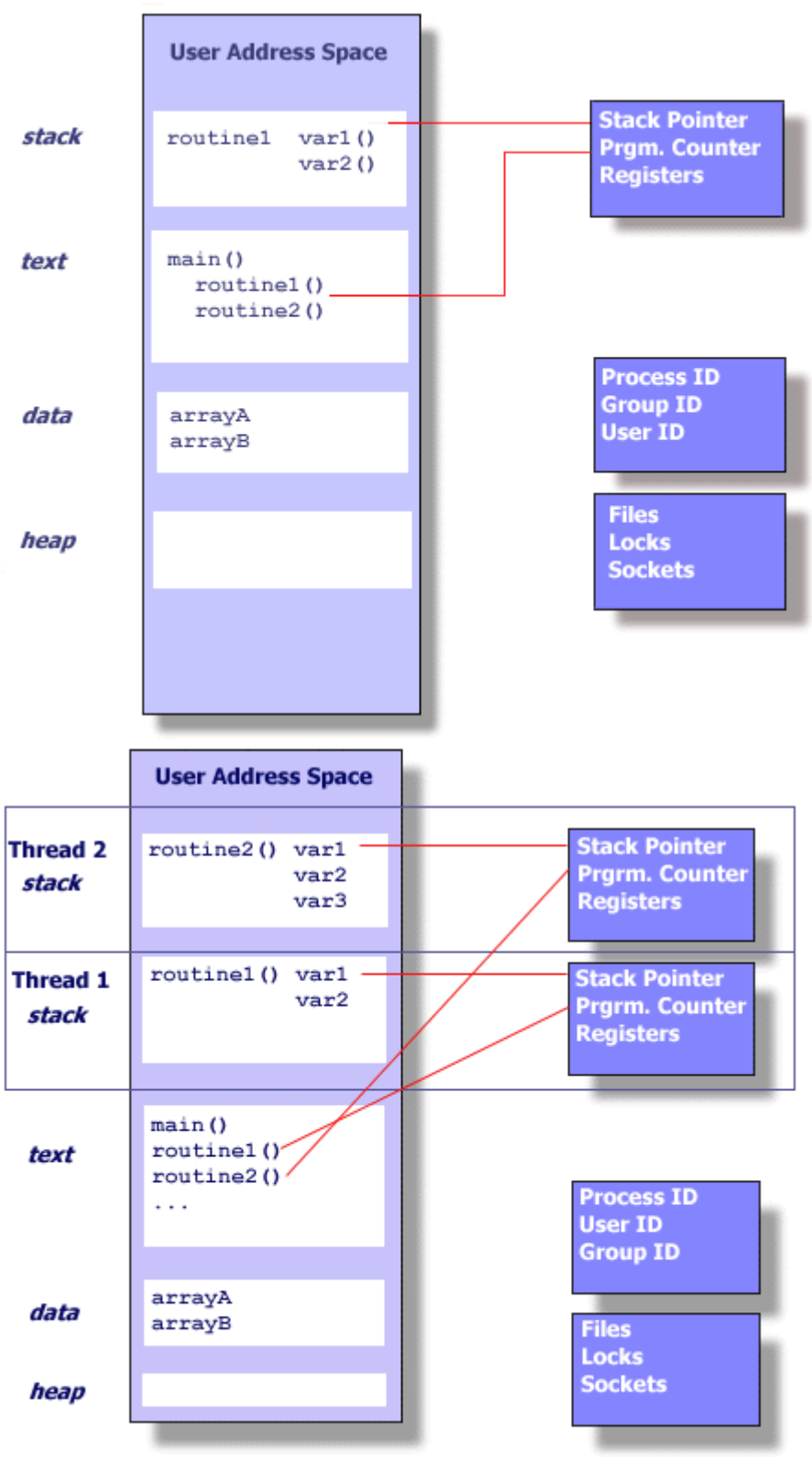
Multithreaded processes on a multiprocessor.

vlákna mohou běžet současně

na jednoprocesorovém stroji může aplikace (proces) pokračovat i když některé vlákno je blokováno

Zdroj: Vahalia, U.: UNIX Internals. Prentice Hall 1996.

Implementace



Typy vláken

- jádrová
- lehké procesy
- uživatelská

jádrová vlákna (*kernel threads*)

- jsou vytvářena a rušena uvnitř jádra pro určené funkce
- sdílí prostor jádra a má vlastní zásobník
- je nezávisle plánováno standardními mechanismy **`sleep()`, `wakeup()`**
- není sdruženo se žádným uživatelským procesem
- efektivní vytváření a používání

použití

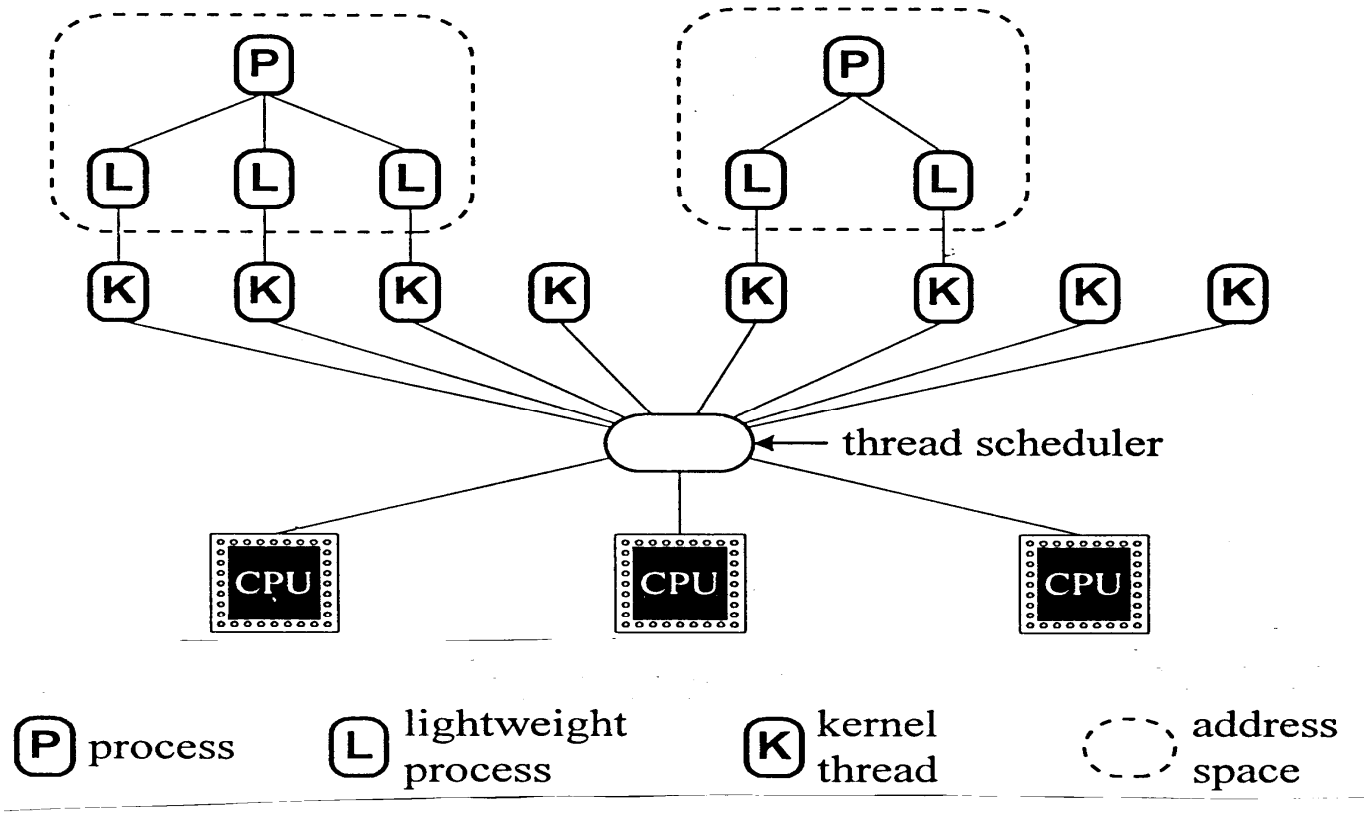
- asynchronní V/V operace
- ošetření přerušení
- práce s vyrovnávacími paměťmi disků
- práce se stránkami paměti
- síťové spojení

koncepčně shodné s démony, které nejsou sdruženy s uživatelským procesem a vykonávají systémové úkoly

vlákna umožňují jednodušší implementaci

lehké procesy

- lehký proces je uživatelské vlákno podporováno jádrovým vláknem
- proces může mít jeden nebo více lehkých procesů, každý podporován zvláštním jádrovým vláknem
- lehké procesy jsou nezávisle plánovány a sdílejí adresový prostor a ostatní prostředky procesu
- můžou vykonávat systémová volání a čekat na prostředky
- každý lehký proces může být vykonáván na jiném procesoru
- přístup k prostředkům více lehkými procesy musí být synchronizován



Zdroj: Vahalia, U.: UNIX Internals. Prentice Hall 1996.

omezení:

- většinu operací s lehkými procesy (vytvoření, synchronizace) vykonává jádro
- počet lehkých procesů je limitován prostředky OS
- lehké procesy musí být dostatečně obecné, aby univerzálně vyhovovali aplikacím
- lehké procesy jsou plánovány jádrem

uživatelská vlákna

vlákna možno poskytnout na uživatelské úrovni, bez toho aby o nich jádro vědělo

dosahuje se toho knihovními funkcemi

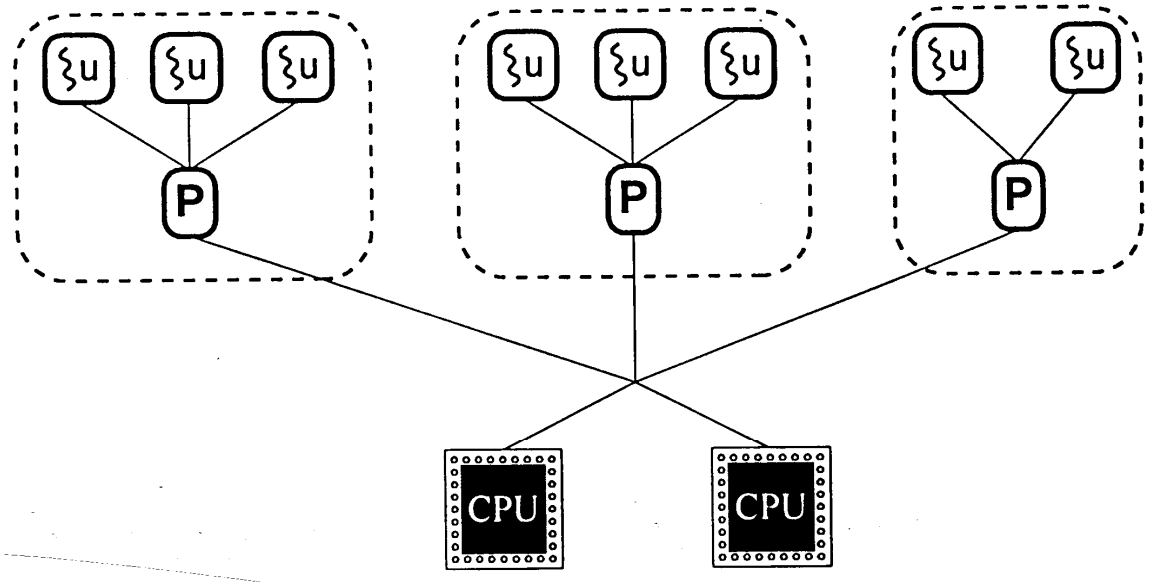
IEEE POSIX 1003.1c
Pthreads

GNU Pth – The GNU Portable Threads

...

interakce vláken nezahrnují jádro a jsou proto velice rychlé

uživatelská vlákna nemůžou využít paralelizmus multiprocesorových systémů

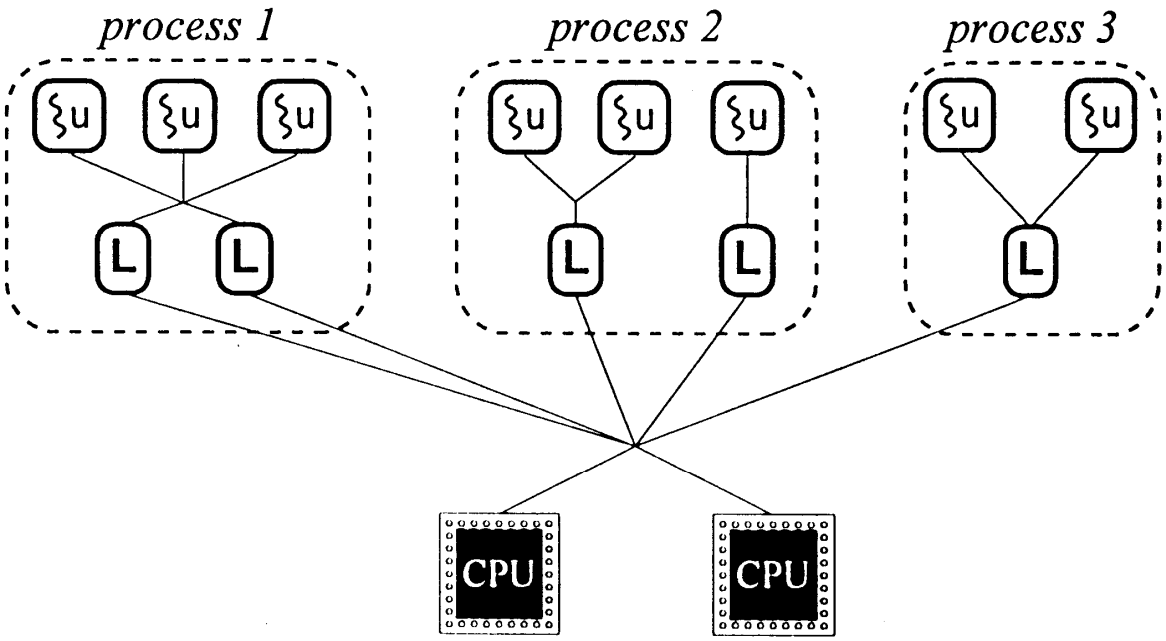


User threads on top of ordinary processes

Zdroj: Vahalia, U.: UNIX Internals. Prentice Hall 1996.

alternativně možno uživatelská vlákna přepínat nad lehkými procesy

- jádro vidí, spravuje a přepíná lehké procesy
- knihovna vykonává přepínaní uživatelských vláken nad lehkými procesy, zabezpečuje jejich synchronizaci bez vědomí jádra



User threads multiplexed on lightweight libraries

Zdroj: Vahalia, U.: UNIX Internals. Prentice Hall 1996.

výhody:

- přirozené programování, např. oknové systémy
- synchronní model programování
- můžeme poskytovat různé knihovny vláken vhodné pro různé aplikace, www.gnu.org/software/pth
- výkonnost
    - časy operací pro uživatelské vlákno, lehký proces a proces (v mikrosekundách) [Vahalia]

	čas vytvoření	synchronizace semaforem
uživatelské vlákno	52	66
lehký proces	350	390
proces	1700	200

Zdroj: Vahalia, U.: UNIX Internals. Prentice Hall 1996.

nevýhody:

- jeden adresový prostor a společné prostředky vyžadují synchronizaci na úrovni vláken (procesy využívají oprávnění)
- plánování lehkých procesů vykonává jádro a nevidí jaké a kolik uživatelských vláken je na nich přepínáno
    - může vykonat preempci lehkého procesu, na kterém běží proces vlastnící prostředek, který bude požadovat vlákno na novém lehkém procesu
    - může vykonat preempci lehkého procesu s vláknem s vysokou prioritou v procesu
    - uživatelská vlákna přepínána na jednom lehkém procesu nemůžou být vykonávána paralelně ani na multiprocesorových systémech

Solaris, SVR4.2/MP

- jádrová vlákna
- lehké procesy
- uživatelská vlákna

jádrová vlákna

- nezávisle plánována na procesory
- v jednom adresovém prostoru → efektivnější přepínání mezi vlákny než mezi procesy

implementace

- zásobník
- údajová struktura
  - uložení registrů jádra
  - priorita a plánovací informace
  - ukazatele pro zařazení do front
  - ukazatel na zásobník
  - ukazatele na sdružené záznamy **`lwp`** a **`proc`** (NULL, není-li združen s lehkým procesem)
  - ukazatele na udržování fronty všech vláken procesu a všech vláken v systému
  - informace o združeném lehkém procesu

některé jádrová vlákna vykonávají lehké procesy, jiná vnitřní funkce jádra, zápis na disk

lehké procesy

více lehkých procesů může být vykonáváno v jednom procesu

implementace

změna tradičních údajových struktur, u oblasti a **`proc`** záznamu

`proc` záznam obsahuje všechny údaje o procesu

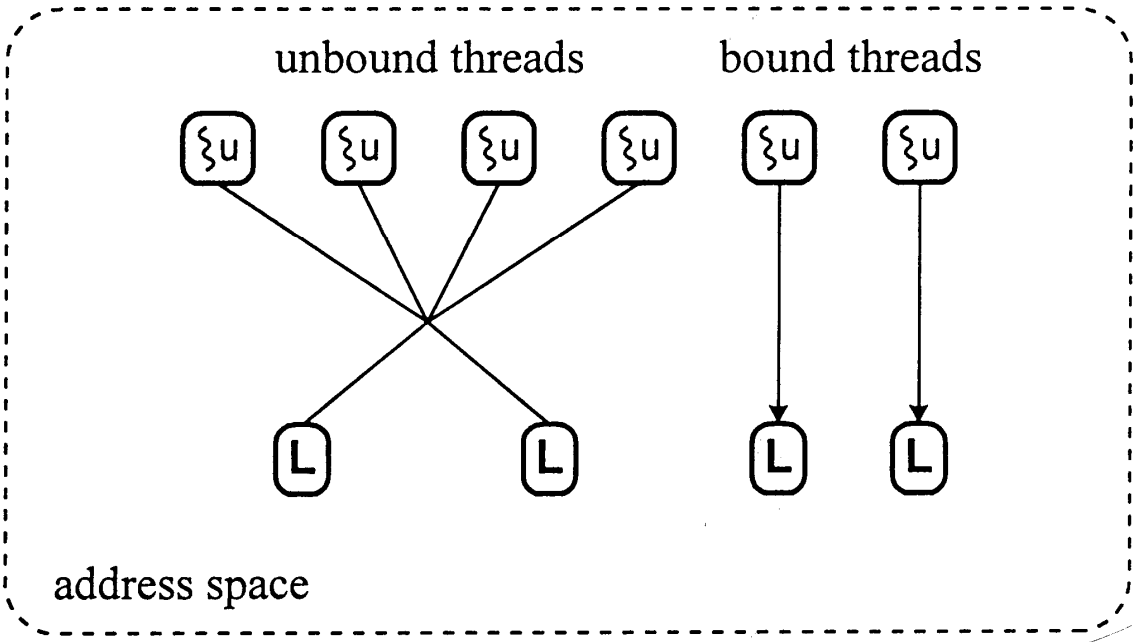
`lwp` záznam obsahuje údaje o lehkém procesu

- uložené hodnoty registrů uživatelské úrovně
- argumenty systémových volání, výsledky, chybový kód
- informaci pro zpracování signálů
- využití prostředků
- alarmy
- využití CPU
- ukazatel na jádrové vlákno
- ukazatel na **`proc`** záznam

uživatelská vlákna

implementována knihovnou

- jsou tvořena, rušena a spravována bez jádra
- knihovna poskytuje synchronizační a plánovací prostředky
- knihovna vytvoří bank lehkých procesů, na kterých přepíná uživatelská vlákna, M:N model
- uživatelskému vláknu může být vyhrazen lehký proces



The process abstraction in Solaris

Zdroj: Vahalia, U.: UNIX Internals. Prentice Hall 1996.

implementace

- identifikátor vlákna
- úschova registrů, počítadlo instrukcí a ukazatel na zásobník
- uživatelský zásobník
- maska signálů
- priorita
- lokální paměť, **`errno`** nemůže být jedno pro proces

Linux

poskytuje knihovní systémové volání **`__clone()`**

umožňuje definovat funkci vykonávanou ve vláknu, které části kontextu procesu se zdvojí, a které zůstanou společné

z pohledu jádra vytváří nový proces - potomka, který však může sdílet s rodičem různé části kontextu

některé publikace potom procesy, které nemají zdvojeny všechny části kontextu, nazývají lehké procesy
(i když nemají „pod sebou“ jádrový proces)

jádro s nimi zachází stejně jako s procesy

z pohledu naší klasifikace *jádrová vlákna, lehké procesy, uživatelské procesy* jde o jádrová vlákna

termín vlákno potom používají pro uživatelská vlákna
termín jádrová vlákna používají v užším smyslu pro ta jádrová vlákna, které vykonávají systémové úkoly

`__clone()` má čtyři parametry

`fn` specifikuje funkci, kterou má potomek vykonat, po jejím vykonání potomek končí

`arg` ukazatel na argumenty funkce **`fn()`**

`flags` obsahuje číslo signálu, který se pošle rodiči po skončení potomka a skupinu příznaků klonování, které specifikují části kontextu (prostředky) sdílené potomkem a rodičem

když jsou příznaky nastavené, jsou části kontextu sdílené

`CLONE_VM`
tabulka oblastí paměti procesu a všechny stránky

`CLONE_FS`
kořenový adresář a okamžitý pracovní adresář

`CLONE_FILES`
tabulku deskriptorů souborů

`CLONE_SIGHAND`
tabulka zpracování signálů

`CLONE_PID`
PID (jenom procesem 0)

CLONE_PTRACE
je-li rodič sledován voláním **ptrace()**, potomek je také sledován

CLONE_VFORK
použito pro **vfork()**

child_stack specifikuje ukazatel na uživatelský zásobník potomka

__clone() využívá volání nízko úrovňové služby **sys_clone()**, která má jenom dva parametry: **flags** a **child_stack**

po návratu **clone()** určí je-li v rodiči nebo potomkovi a potomek vykoná **fn(arg)**

fork()

Linux implementuje jako **sys_clone()** se specifikováním signálu **SIGCHLD** a vynulováním příznaků klonování v prvním parametru a hodnotou druhého parametru 0.

vfork()

Linux implementuje jako **sys_clone()** se specifikováním signálu **SIGCHLD** a nastavením příznaků **CLONE_VM** a **CLONE_VFORK** v prvním parametru a hodnotou druhého parametru 0.

Příklad

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>

int prom, fd;

int funkce() {
  prom = 20;
  close(fd);
  _exit(0);
}

main() {
  void **p_zasobnik;
  char ch;

  prom = 1;
  fd = open("test", O_RDONLY);
  p_zasobnik = (void**) malloc(16384);

  clone (funkce, p_zasobnik+16384, CLONE_VM|CLONE_FILES,
NULL);
  sleep(1);

  printf("prom je nyni: %d\n", prom);
  if (read(fd, &ch, 1) < 0) {
      perror("chyba READ");
    return(1);
  }
  write(1, &ch, 1);
  return(0);
}
```

```
prom je nyní: 20
chyba READ: Bad file descriptor
```

Využití jádrových vláken Linuxu

- vykonávání systémových funkcí jádra
- implementace uživatelských vláken

vykonávání systémových funkcí

vytvoří se jádrové vlákno pro požadovanou funkci

pseudokód:

```
int kernel_thread(int (*fn)(void *),
                  void * arg,
                  unsigned long flags)
{
    pid_t p;
    p = sys_clone(flags|CLONE_VM, 0 );
    if (p)    /* rodic */
        return p;
    else {    /* potomek */
        fn(arg);
        exit;
    }
}
```

Proces 0 vytvořen při inicializaci funkcí **`start_kernel()`**

Proces 1 vytvořen procesem 0

```
kernel_thread(init, NULL,
  CLONE_FS|CLONE_FILES|CLONE_SIGHAND);
```

Implementace uživatelských vláken

knihovny implementující **POSIX 1003.1c - Pthreads**

vytvoření vlákna

```
pthread_create (thread_id, attr,
                thread_function, arg)
```

implementace

```
/* Initialize the thread descriptor */
  new_thread->p_nextwaiting = NULL;
  new_thread->p_spinlock = 0;
  new_thread->p_signal = 0;
  new_thread->p_signal_jmp = NULL;
  new_thread->p_cancel_jmp = NULL;
  new_thread->p_terminated = 0;
  new_thread->p_detached = attr == NULL ? 0 :
                                attr->detachstate;
  new_thread->p_exited = 0;
  new_thread->p_retval = NULL;
  new_thread->p_joining = NULL;
  new_thread->p_cleanup = NULL;
  new_thread->p_cancelstate = PTHREAD_CANCEL_ENABLE;
  new_thread->p_canceltype = PTHREAD_CANCEL_DEFERRED;
  new_thread->p_canceled = 0;
  new_thread->p_errno = 0;
  new_thread->p_h_errno = 0;
  new_thread->p_initial_fn = start_routine;
  new_thread->p_initial_fn_arg = arg;
  new_thread->p_initial_mask = mask;
  for (i = 0; i < PTHREAD_KEYS_MAX; i++)
                    new_thread->p_specific[i] = NULL;
  /* Do the cloning */
  pid = __clone(pthread_start_thread, new_thread,
         (CLONE_VM | CLONE_FS | CLONE_FILES
           | CLONE_SIGHAND | PTHREAD_SIG_RESTART),
          new_thread);
```

[glibc-linuxthreads-2.0.1, http://ftp.gnu.org/gnu/glibc/]

jako pro procesy, i pro vlákna je deskriptor vlákna a zásobník jedna datová struktura

funkce **pthread_start_thread** inicializuje vlákno a vykoná **fn(arg)**

```
static int pthread_start_thread(void *arg)
{
  pthread_t self = (pthread_t) arg;
  void * outcome;
  /* Initialize special thread_self processing, if
     any.  */
#ifdef INIT_THREAD_SELF
  INIT_THREAD_SELF(self);
#endif
  /* Make sure our pid field is initialized, just in
     case we get there before our father has
     initialized it. */
  self->p_pid = getpid();
  /* Initial signal mask is that of the creating
     thread. (Otherwise, we'd just inherit the mask
     of the thread manager.) */
  sigprocmask(SIG_SETMASK, &self->p_initial_mask,
              NULL);
  /* Run the thread code */
  outcome =
          self->p_initial_fn(self->p_initial_fn_arg);
  /* Exit with the given return value */
  pthread_exit(outcome);
  return 0;
}
```

ukončení vlákna

`pthread_exit(stav)`

čekání na skončení vlákna

`pthread_join (thread_id, stav)`

volající vlákno, čeká na skončení vlákna **`thread_id`**

`stav` umožní získat stav skončení, je-li specifikován v **`pthread_exit`**

vzájemné vylučování (*mutual exclusion*) – mutex

synchronizace přístupu k prostředkům (datům)

inicializace

`pthread_mutex_init (mutex, attr)`

`mutex` je inicializován jako „odemknutý“ (*unlocked*)

`pthread_mutex_lock (mutex)`

vlákno „zamkne“ (*locks*) **`mutex`** a pokračuje,
je-li **`mutex`** zamknut, vlákno je do odemknutí blokováno

`pthread_mutex_unlock (mutex)`

vlákno „odemkne“ (*unlocks*) **`mutex`**

podmínkové proměnné (*condition variables***)**

synchronizace při dosažení nějakých hodnot proměnných

reprezentuje nějakou podmínku, např. **`pocet=limit`**

pro přístup k proměnným vyjadřujícím podmínku, musí být s každou podmínkovou proměnnou sdružený mutex

inicializace

```
pthread_cond_init (condition, mutex)
```

čekání na signalizaci (*signalling*) podmínky

```
pthread_cond_wait (condition, mutex)
```

pseudokód:

```
pthread_cond_wait (condition, mutex)
begin
    pthread_mutex_unlock (mutex);
    čekej_na_podmínku (condition);
    pthread_mutex_lock (mutex);
end
```

signalizování podmínky

pthread_cond_signal (condition)

vlákno čekající na podmínku je vzbuzeno a dokončí **pthread_cond_wait()**, signalizující vlákno musí odemknout sdružený **mutex**

obecně před úspěšným zamknutím **mutex-**u čekajícím vláknem, mohlo **mutex** zamknout jiné vlákno a splnění podmínky se musí znovu zkontrolovat

```
#define LIMIT 100

int pocet=0;
pthread_mutex_t pocet_mutex;
pthread_cond_t pocet_cv;

/*
vlakno inkrementujici promennou pocet
*/
...
for (i=0; i<1000; i++) {
  pthread_mutex_lock(&pocet_mutex);
  pocet++;
  if (pocet == LIMIT)
    pthread_cond_signal(&pocet_cv);
  pthread_mutex_unlock(&pocet_mutex);
}
...
```

```
/*
vlakno cekajici na dosazeni limitu promennou
pocet, neni-li splnena
*/
...
pthread_mutex_lock(&pocet_mutex);
while (pocet < LIMIT)
   pthread_cond_wait(&pocet_cv, &pocet_mutex);
pthread_mutex_unlock(&pocet_mutex);
...
```

Příklad

```
/******************************************************
* FILE: condvar1.c
* DESCRIPTION:
*   Example code for using Pthreads condition
*   variables.  The main thread creates three
*   threads.  Two of those threads increment a
*   "count" variable,while the third thread watches
*   the value of "count".  When "count" reaches a
*   predefined limit, the waiting thread is signaled
*   by one of the incrementing threads.
*
* SOURCE: Adapted from example code in "Pthreads
* Programming", B. Nichols et al. O'Reilly and
* Associates.
* LAST REVISED: 02/11/2002  Blaise Barney
******************************************************/

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

```
#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  int *my_id = idp;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting
    thread when condition is reached.  Note that this
    occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %d,
                count = %d    Threshold reached.\n",
                *my_id, count);
      }
    printf("inc_count(): thread %d,
              count = %d, unlocking mutex\n",
             *my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex
       lock */
    for (j=0; j < 1000; j++)
      result = result + (double)random();
    }
  pthread_exit(NULL);
}
```

```
void *watch_count(void *idp)
{
  int *my_id = idp;

  printf("Starting watch_count(): thread %d\n",
         *my_id);
  /*
  Lock mutex and wait for signal.  Note that the
  pthread_cond_wait routine will automatically lock
  and unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before
  this routine is run by the waiting thread, the loop
  will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv,
                      &count_mutex);
    printf("watch_count(): thread %d Condition signal
            received.\n", *my_id);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

```
void main()
{
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects
  */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /*
  For portability, explicitly create threads in a
  joinable state
  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
                              PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count,
                 (void *)&thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count,
                 (void *)&thread_ids[1]);
  pthread_create(&threads[2], &attr, watch_count,
                  (void *)&thread_ids[2]);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n",
          NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);

}
```

```
inc_count(): thread 0, count = 1, unlocking mutex
Starting watch_count(): thread 2
inc_count(): thread 1, count = 2, unlocking mutex
inc_count(): thread 0, count = 3, unlocking mutex
inc_count(): thread 1, count = 4, unlocking mutex
inc_count(): thread 0, count = 5, unlocking mutex
inc_count(): thread 0, count = 6, unlocking mutex
inc_count(): thread 1, count = 7, unlocking mutex
inc_count(): thread 0, count = 8, unlocking mutex
inc_count(): thread 1, count = 9, unlocking mutex
inc_count(): thread 0, count = 10, unlocking mutex
inc_count(): thread 1, count = 11, unlocking mutex
inc_count(): thread 0, count = 12  Threshold reached.
inc_count(): thread 0, count = 12, unlocking mutex
watch_count(): thread 2 Condition signal received.
inc_count(): thread 1, count = 13, unlocking mutex
inc_count(): thread 0, count = 14, unlocking mutex
inc_count(): thread 1, count = 15, unlocking mutex
inc_count(): thread 0, count = 16, unlocking mutex
inc_count(): thread 1, count = 17, unlocking mutex
inc_count(): thread 0, count = 18, unlocking mutex
inc_count(): thread 1, count = 19, unlocking mutex
inc_count(): thread 1, count = 20, unlocking mutex
Main(): Waited on 3  threads. Done.
```

Zdroj: www.llnl.gov

1996

LinuxThreads

model 1:1, jedno uživatelské vlákno je podporováno jedním jádrovým vláknem, jednoduchá implementace, možnost využití multiprocesorů pro paralelní vykonávání, není dvojité plánování

1999

GNU Portable Threads GNU Pth

přenositelné pro systémy UNIX, implementuje POSIX vlákna v uživatelském prostoru v rámci procesu, M:1 model

2002

NPTL, Native POSIX Thread Library, zachovává jádrový model 1:1, nahradily LinuxThreads od verze 2.6

NGPT, Next Generation POSIX Threading, vychází z GNU Pth, model M:N, ukončen uprostřed 2003

http://people.redhat.com/drepper/nptl-design.pdf (Fig.1, Fig.2)

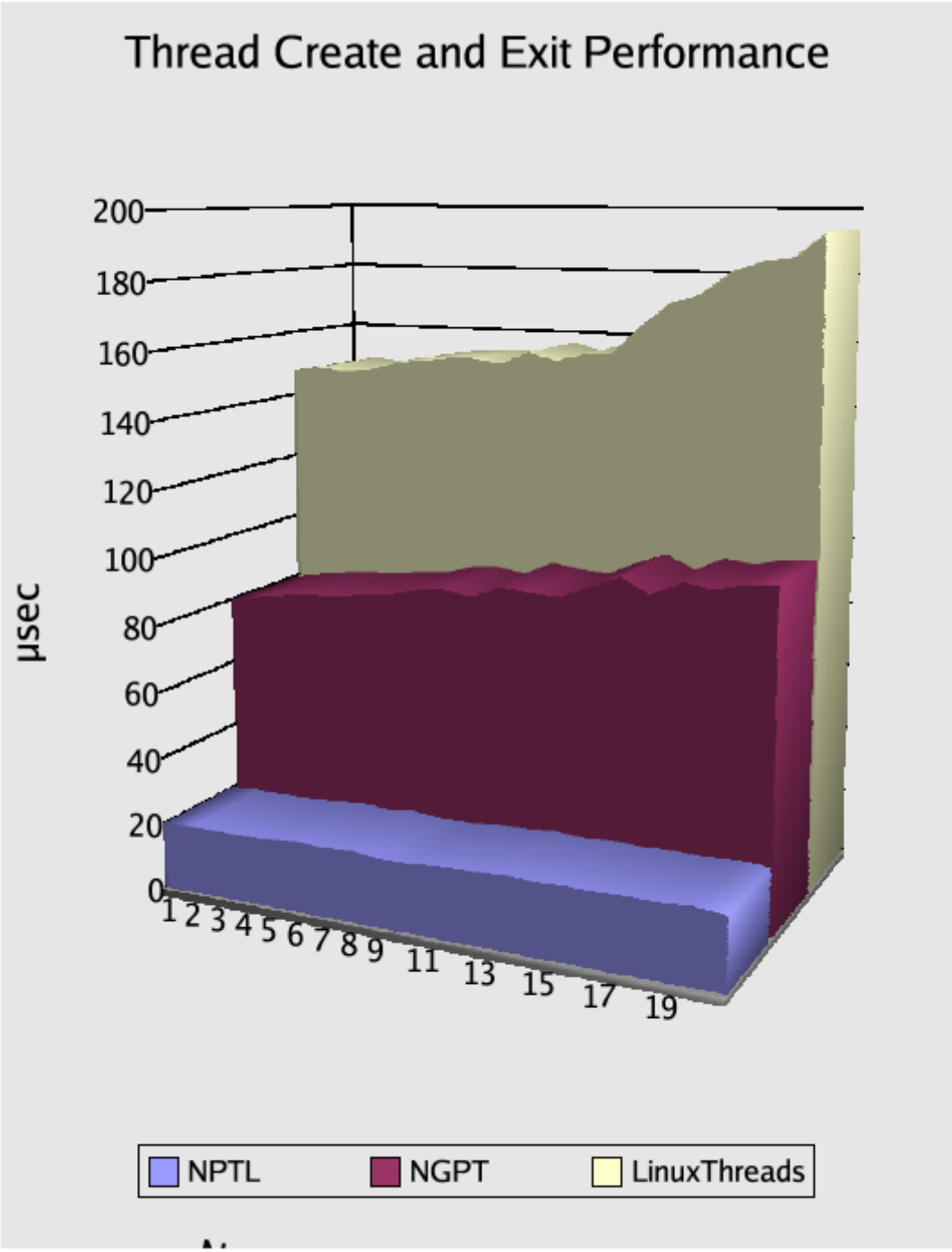


Figure 1: Varying number of Toplevel Threads

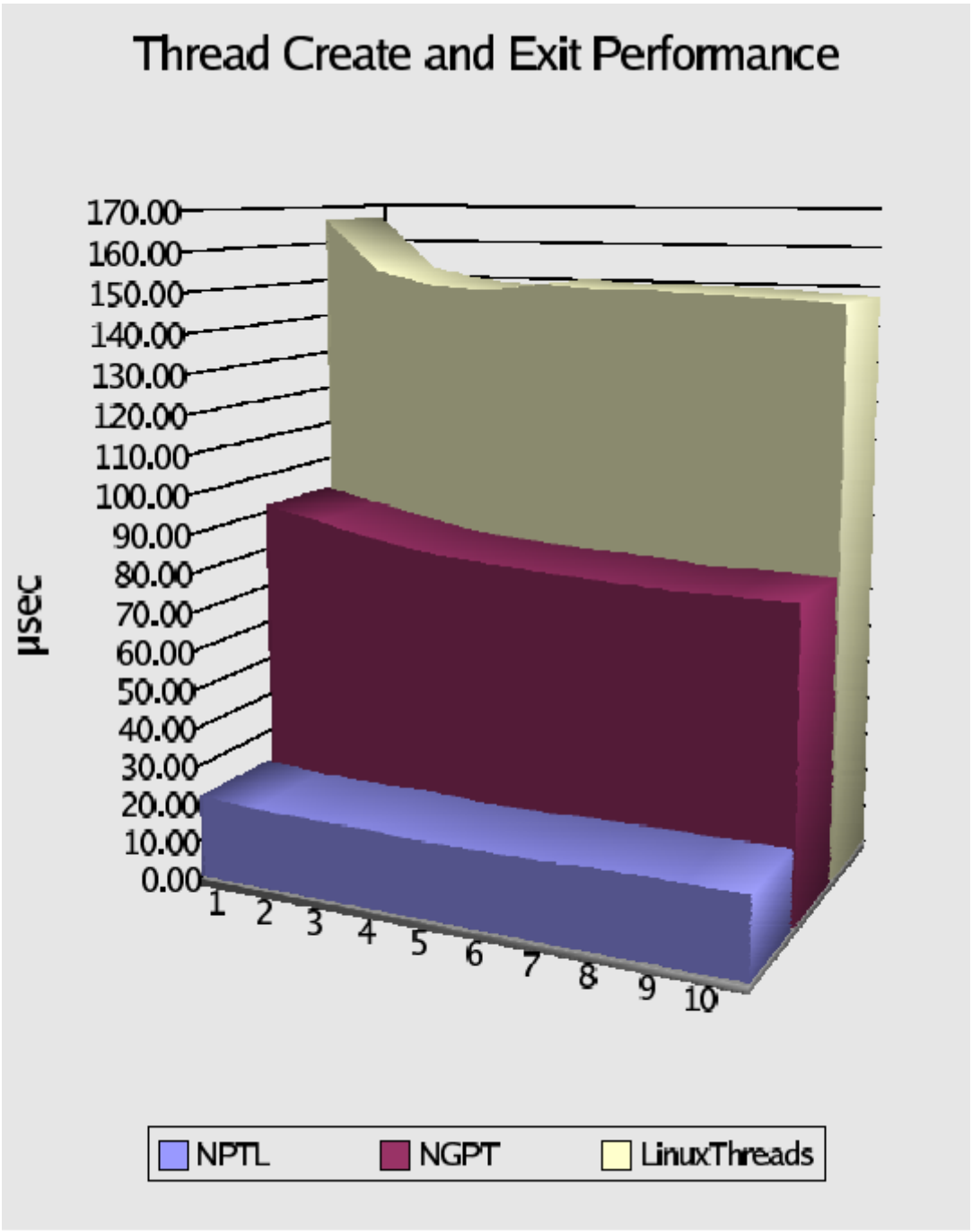


Figure 2: Varying number of Concurrent Children