Vorlesung Betriebssysteme II

Thema 2.5: Threads (Aktivitäten, die zweite)

Robert Baumgartl

10. April 2024

Threads (Leichtgewichtsprozesse)

- Prozess = Container für Ressourcen + (ein) identifizierbarer Handlungsablauf ("Thread of Execution")
- Idee: Trennung beider Konzepte:
  - Prozess = Container für Ressourcen (*passiv*)
  - Thread = identifizierbarer unabhängiger Handlungsablauf (*aktiv*)
- $\rightarrow$ ein oder mehrere Threads pro Prozess möglich, diese teilen sich die Ressourcen des Prozesses
- $\rightarrow$ parallele Abarbeitung innerhalb eines Adressraums

Der Begriff „Thread" ist auch im Deutschen weitestgehend etabliert, die korrekte Übersetzung „Faden" benutzen nur Fanatiker. Eher wird noch „Leichtgewichtsprozess" eingesetzt.

Veranschaulichung mehrerer Threads

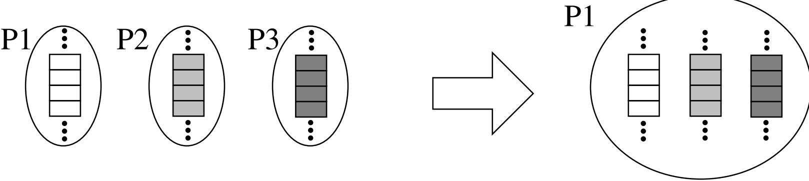

Abbildung: Übergang von mehreren Prozessen zu mehreren Threads in einem Prozess

Konsequenzen

- Thread kann nur innerhalb eines Prozesses existieren!
- alle Threads eines Prozesses teilen sich dessen Ressourcen und Adressraum bis auf die Register und den Stack
- $\Rightarrow$ kein Schutz zwischen Threads eines Prozesses
- Kooperation im Vordergrund

dem Prozess gehörend	*jedem Thread gehörend*
Adressraum	Program Counter
Globale Variablen	Register
eröffnete Dateien	Stack (!)
(Signale)	automatische Variablen
Kindprozesse	(Globalzustand)

Tabelle: Gemeinsame und private Ressourcen von Threads eines Prozesses (Beispiele)

Warum Threads?

- feingranulare Parallelität der Verarbeitung
- Erzeugung/Vernichtung viel schneller als von Prozessen
- effektive Ausnutzung mehrerer CPUs/Kerne
- Umschaltung zwischen Threads eines Prozesses schnell
- Dekomposition in Threads liefert Performancegewinn, insbesondere wenn Ein-/Ausgabe-beschränkte Funktionalität

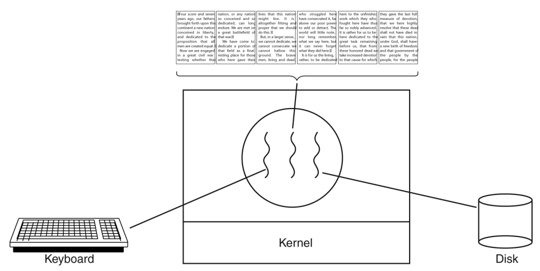

Abbildung: Textverarbeitung mit 3 Threads (Tanenbaum: *Modern Operating Systems*, S.95)

Variante 1: User-Level-Threads

- Threads werden im User Mode (d.h., ohne Intervention und Wissen des Betriebssystems) erzeugt, synchronisiert, vernichtet
- Threadbibliothek, die Routinen zum Erzeugen, Beenden, Synchronisieren und Umschalten von Threads realisiert, erforderlich
- Threadbibliothek übernimmt Management aller Threads (speichert deren Zustandsinformationen)
- Umschaltungvorgang: Sichern und Restaurieren des Thread-Kontextes (alle Register)
- kooperatives Programmiermodell: jeder Thread muss freiwillig (ab und zu) den Prozessor abgeben
- Kernel hat kein Wissen über Threads, kennt (und verwaltet) nur Prozesse
- m:1-Abbildung, d. h. , allen Threads ist genau eine Kernel-Aktivität zugeordnet, nämlich der zugehörige Prozess

Veranschaulichung von User-Level-Threads

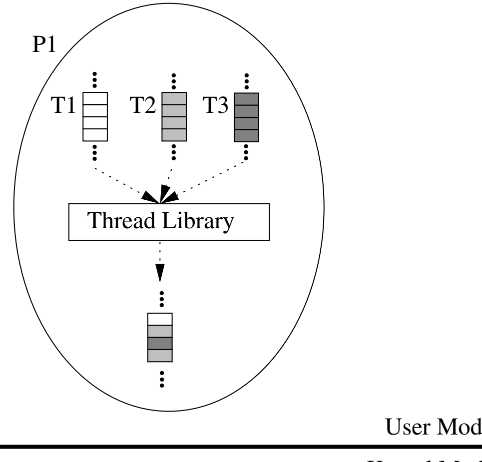

Abbildung: User-Level-Threads

User-Level-Threads, cont'd

Bewertung:

- \+ kann (auch) für Betriebssystem implementiert werden, welches kein Threadkonzept kennt (z. B. MS-DOS)
- \+ Threadoperationen besonders schnell, da kein Kernein-/austritt, kein Flush der Prozessorcaches, kein Adressraumwechsel
- – blockierender Systemruf blockiert *alle* Threads eines Prozesses (verhindert Parallelität), Abhilfe:
  - ⋆ nichtblockierende Systemrufe
  - ⋆ im Vorhinein ermitteln, ob Ruf blockieren wird (z. B. mittels `select()`); wenn ja, dann Weiterarbeit eines anderen Threads
- – *Page Fault* blockiert alle Threads eines Prozesses
- – kooperatives Programmiermodell mit freiwilliger Abgabe des Prozessors erforderlich (da keine unterbrechende Instanz aka Betriebssystem)

Variante 2: Kernel-Level-Threads

- 1:1-Abbildung (jedem Thread ist *genau eine* Aktivität des Kernels zugeordnet)
- Threads im Kernel verwaltet, dieser verteilt Threads auf alle existierenden Kerne
- Threadoperationen teurer, da mit Systemruf (˜eintritt, ˜austritt) verbunden (z. B. `CreateThread()` in Win32)
- Kombinationen aus User-Level- und Kernel-Level-Threads sind ebenfalls möglich

Veranschaulichung von Kernel-Level-Threads

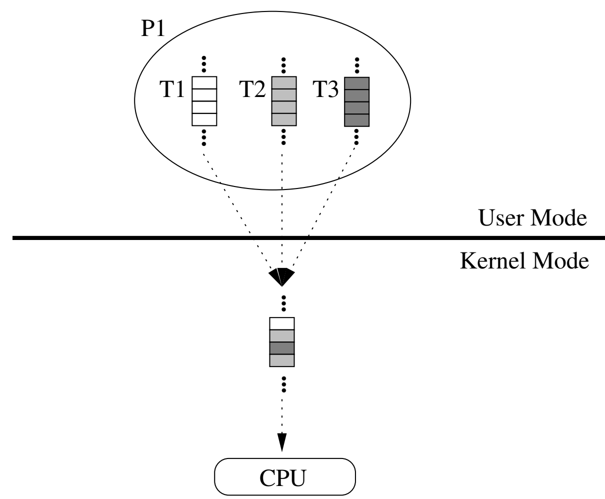

Abbildung: Kernel-Level-Threads

Threads in Windows

- Windows-Welt: beginnend ab Windows NT (Win32-API) sind Kernel-Level-Threads im System verankert.
  - `CreateProcess()` legt einen neuen Adressraum an und startet in ihm einen (ersten) Thread
  - mittels `CreateThread()` können in einem existierenden Prozess weitere Threads gestartet werden
- User-Level-Threads existieren *zusätzlich:* die so genannten *Fibers*
- d. h. , es können mehrere User-Level-Threads (Fibers) in einem Kernel-Level-Thread existieren

Threads in Linux

- ab Kernel 2.0 Bibliothek *LinuxThreads*
- weitgehend, aber nicht vollständig, POSIX-kompatibel
- limitiert (max. Threadanzahl 8192)
- nicht allzu performant
- daher ab 2002 Entwicklung der *Native POSIX Thread Library* (NPTL)
- behebt diese Nachteile, voll POSIX-kompatibel
- 1:1-Implementierung (1 Thread pro Kernelaktivität)
- andere Unixe besitzen u. U. andere Thread-Implementierungen, die aber stets POSIX-Threads realisieren

Abfrage der installierten Thread-Version in Linux:

```
~> getconf GNU_LIBPTHREAD_VERSION
```

POSIX-Threads (Pthreads)

- POSIX standardisiert ein Thread-API für Unix-artige Betriebssysteme
- API wird *Pthreads* genannt
- ca. 100 Funktionen
- Standard enthält keine Aussage zur Implementierung der Funktionen
- `man 7 pthreads`
- auch für Win32 verfügbar (*pthreads-w32*)

Pthreads-API – Übersicht (kleiner Ausschnitt)

`pthread_create()`	Anlegen eines neuen Threads
`pthread_join()`	Warten auf Ende des Threads
`pthread_exit()`	Beenden des rufenden Threads
`pthread_detach()`	Abkoppeln vom Vater
`pthread_kill()`	Zustellung eines Signals an Thread
`pthread_attr_init()`	Init der Thread-Attribute
`pthread_mutex_lock()`	Synchronisation an bin. Semaphor
`pthread_mutex_unlock()`	
`pthread_cond_init()`	Anlegen einer Bedingungsvariable
`pthread_cond_wait()`	Sync. an Bedingungsvariable
`pthread_cond_signal()`	

Tabelle: Einige Funktionen der Pthreads-API

Pthreads-Programmierung

- `#include <pthread.h>`
- Linken mit Schalter `-lpthread`
- Funktionsnamen beginnen stets mit `pthread_`
- alle Funktionen liefern 0 wenn erfolgreich, -1 bei Fehler
- Threads eines Prozesses kommunizieren über gemeinsame (globale) Variable
- IPC nur zu Threads anderer Prozesse

Erzeugung eines Threads

```
int pthread_create(pthread_t *tid, pthread_attr_t
  *attr, void* (*start_fkt)(void*), void *arg);
```

- erzeugt neuen Thread
- `tid`: Identifikator des neuen Threads
- `attr`: legt Attribute des Threads fest
- `start_fkt`: Startfunktion
- `arg`: Zeiger auf (beliebiges) Datum, welches der Thread als Parameter erhält
- keine Vater-Sohn-Beziehungen; jeder darf Join ausführen

Beendigung eines Threads

```
void pthread_exit(void *retval);
```

- beendet den rufenden Thread und liefert einen Rückgabewert mittels `retval`, sofern ein anderer Thread mittels `pthread_join()` wartet
- `*retval` sollte nicht auf dem Stack des endenden Threads liegen (keine lokale Variable sein)
- `return` aus der Hauptfunktion hat den gleichen Effekt

```
int pthread_join(pthread_t tid, void **ret);
```

- Analogon zum `waitpid()`, wartet auf Ende des Threads mit Id tid
- Rückgabewert über `ret`
- erst nach Join wird der Thread vernichtet (analog Zombie-Status)

Hello, world mittels Pthreads

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

#define MAXITER 200000

int x=42;

void *thread_f(void *arg)
{

  printf("%s", (char*) arg);
  pthread_exit(&x);
}

int main(void)
{
  pthread_t a;
  int ret;
  int *retval;

  printf("Hello, ");

  ret = pthread_create(
    &a,     /* pointer to variable containing thread ID */
    NULL,              /* pointer to thread attributes */
    (void*) &thread_f,               /* thread function */
    "world!\n");                      /* pointer to argument */
  if (ret != 0) {
    perror("creating 1st thread");
    exit(EXIT_FAILURE);
  }
  ret = pthread_join(a, (void**) &retval);
```

Detach

```
int pthread_detach(pthread_t tid);
```

- macht den Thread un-join-bar, d.h., er wird sofort vernichtet, wenn er seine Startfunktion verlässt

Synchronisation: Mutexe – Init und Zerstörung

Mutex – Mutual Exclusion:

- einfaches Synchronisationsprimitiv
- binärer Semaphor

```
int pthread_mutex_init(pthread_mutex_t *mutex,
  pthread_mutexattr_t *attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- initialisiert den Semaphor (stets offen)

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- zerstört den Mutex
- nur erlaubt, wenn Mutex geöffnet ist (!)

Synchronisation: Mutexe – P() und V()

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- P()

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- V()

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- nichtblockierendes P() (liefert `EBUSY` zurück, wenn Mutex gesperrt)

Synchronisation: Bedingungsvariablen

- komplexes Synchronisationsprimitiv
- repräsentiert Warteschlange (für Threads)
- Datentyp `pthread_cond_t`
- Threads können in Warteschlange wandern (*wait*-Operation) und wieder in die Bereit-Menge aufgenommen werden (*signal*-Operation)

Funktion	*Semantik*
`pthread_cond_init()`	Kreieren und Init einer BV.
`pthread_cond_signal()`	Wecken eines Threads
`pthread_cond_broadcast()`	Wecken *aller* Threads
`pthread_cond_wait()`	Threadblockierung an BV.
`pthread_cond_timedwait()`	Blockierung mit Timeout
`pthread_cond_destroy()`	Löschung der BV.

Leser-Schreiber-Locks

War klar, ne . . . ?

Funktion	*Semantik*
`pthread_rwlock_init()`	Initialisierung eines RW-Locks
`pthread_rwlock_rdlock()`	Sperren durch Leser
`pthread_rwlock_tryrdlock()`	Sperren durch Leser (nichtblock.)
`pthread_rwlock_wrlock()`	Sperren durch Schreiber
`pthread_rwlock_trywrlock()`	Sperren durch Schreiber (nichtblock.)
`pthread_rwlock_unlock()`	Entsperren (für beide)
`pthread_rwlock_destroy()`	Löschen des RW-Locks

Tabelle: Funktionen der Leser-Schreiber-Locks

- „Implementations may favor writers over readers to avoid writer starvation.“

GNU Portable Threads (GNU Pth)

- User-Level-Threads für Unix
- m:1-Implementierung
- *http://www.gnu.org/software/pth/*
- kooperatives Multitasking mit Prioritäten
- Linken mit Schalter `-lpth`

Hello, world! mit GNU Pth

```c
#include <pth.h>
#include <stdio.h>
#include <stdlib.h>

void *pth_thread(void *arg)
{
  printf("world!\n");
  while(1);
  pth_exit(NULL);
}

int main(void)
{
  pth_t tid;

  if (!pth_init()) {
    printf("No pth lib available\n");
    exit(EXIT_FAILURE);
  }
  printf("Hello, ");

  tid = pth_spawn(PTH_ATTR_DEFAULT, &pth_thread, NULL);
  if (!tid) {
    printf("Could not spawn thread\n");
    exit(EXIT_FAILURE);
  }

  pth_join(tid, NULL);

  return 0;
}
```

Was haben wir gelernt?

1. Threads sind eine Abstraktion zur effizienten Parallelisierung von Programmabläufen
2. meist effizienter als Prozesse, da Threads in ein- und demselben Adressraum liegen
3. Kernel-Level-Threads $\leftrightarrow$ User-Level-Threads
4. Windows bringt beide Arten „von Haus aus“ mit
5. wichtigste Implementierung unter Unix/Linux: POSIX Threads
6. pthreads-API-Funktionen zum Kreieren, Synchronisieren und Beenden von Threads

Literaturvorschläge

- David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997
- Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010, Kap. 29–33

- https://randu.org/tutorials/threads/
- https://www.youtube.com/watch?v=ynCc-v0K-do (englisches mehrteiliges pthreads-Tutorial von Brian Fraser)