Warum Threads

By rac
Published: 12.01.2008 - 15:38

- Anfangs gab es nur langsame Computer -> serielles Arbeiten sinnvoll
- Später kamen Multiuser, Timesharing, Multitasking (Unix 197x)
- Warum Multitasking / Multithreading?
- kürzere Ausführungszeiten, bessere Antwortzeiten, Computer besser Ausgenutzt
- Threads sind effizienter inerhalb von Prozessen

- Heutige Computer sind Sammlung von Aufgaben
- Threads füren zu kürzeren Ausführungszeiten, besseren Antwortzeiten (fühlt sich besser an!), Computer ist besser ausgenutzt
- Befor es Threads gab wurde Prozess forking gemacht
- Threads sind effizienter innerhalb von Prozessen
- Sie werden auch "Light Weight Processes" - LWP genant

Das Thread Modell:

- prozesseigene Ressourcen: globale Daten, Programm, Instruktionen etc.
- threadeigene Ressourcen: Stack, PC / SC, Register

PThread Definition

Ein Thread wird inerhalb einer Funktion Ausgeführt.

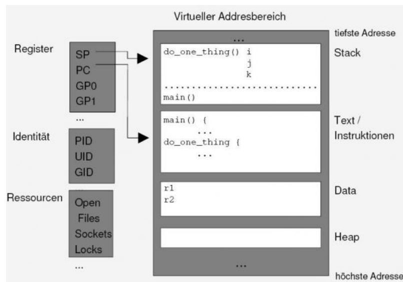
kreire_thread(function, argument)

- Prozesslayout
- Potentieller Parallelismus
- Konkurrentes Programmieren
- Synchronization
- Teilen von Ressourcen
- Kommunikation
- Identität
- Terminierung
- Exit Status & Rückgabewert
- Bibliotheksaufrufe & Fehler
- Warum Threads und nicht Prozesse?
- Aufgabenbereiche für Threading
- Beispiele für Multithreaded Applikationen

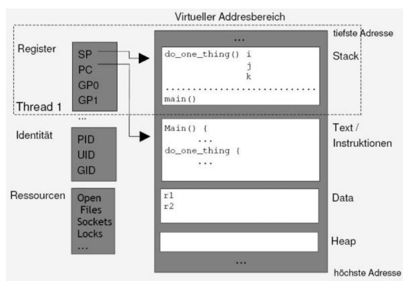
‹ PThreads up Prozesslayout ›

Prozesslayout

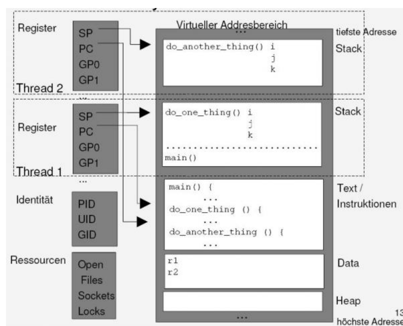
By rac
Published: 12.01.2008 - 16:20



Ein Thread



Zwei Threads



‹ Warum Threads up Potentieller Parallelismus ›

Potentieller Parallelismus

By rac
Published: 12.01.2008 - 16:34

- Voneinander unabhängige Aufgaben
- Resultat ist unabhängig von der Reihenfolge der Ausführung

Gründe für Potentiellen Parallelismus

- Blockierender I/O
- Sich überlapender I/O
- Asynchrone Ereignisse (Async events)
- Netz
- Tastatur
- Interrupts

- Realtime (Vorgaben aus der Realität)

Ist Parallelität immer sinvoll?

Nein, OLPC Projekt als gutes Beispiel (Stromverbrauch senken)

‹ Prozesslayout up Konkurrentes Programmieren ›

Konkurrentes Programmieren

By rac
Published: 12.01.2008 - 16:43

Konkurrent heisst, es könnte möglich sein, dass Parallel Ausgeführt wird.

- Um konkurrentes Programmieren zu ermöglichen, muss dies von der Umgebung / dem System unterstützt werden.

Dazu gehören:

- Parallele Ausführung muss möglich sein
- Synchronisation muss möglich sein
- Kommunikation muss möglich sein

Parallelität vs. konkurrent

- Parallelität -> Wirklich parallele Ausführung
- Konkurrent -> Kann auch timeshared sein
- Reihenfolge potentiell unbekannt / irrelevant

Unter UNIX: Multiprocessing

- Fork -> Elternteil / Kind

- eigene PID
- return Wert ist PID oder 0

- Vor / Nachteile:
- Speicher / Ressourcenschutz
- Synchronisation aufwändiger
- fork einfacher, aber unübersichtlich

Thread Multiprozessing

- weniger Programmoverhead
- weniger Systemoverhead
- keine Eltern
- Threads sind gleichwertig
- jedoch existiert ein ”main” Thread

```
extern int pthread_create(
        pthread_t *thread_handle,
        const pthread_attr_t *thread_attribute,
        void * (*)funktion(void *),
        void *parameter);
```

‹ Potentieller Parallelismus up Synchronization ›

Synchronization

By rac
Published: 12.01.2008 - 16:50

- Bei seriellen Programmen -> Reihenfolge des Codes
- Unix multiproc -> waitpid
- Pthreads:
- pthread_join
- Mutexe (mutual exclusion)
- Bedingungsvariablen -> weniger Wartezeit, feiner, bessere Synchro.
- Mutex muss vom System zur Verfügung gestellt werden

‹ Konkurrentes Programmieren up Teilen von Ressourcen ›

Teilen von Ressourcen

By rac
Published: 12.01.2008 - 17:20

- Bei Prozessen wird nichts geteilt
- Bei Threads wird alles geteilt
- Kommunikation über Globale Variablen -> Fehlerquelle
- Potential für Fehler aufgrund der "alles geteilt" Prämisse

‹ Synchronization up Kommunikation ›

Kommunikation

By rac
Published: 12.01.2008 - 17:22

- über Mutexe
- über Globale Variablen
- IPC (Inter Process Communication) -> Jedes mal findet ein OS Aufruf statt (=teuer)
- Sockets
- Shared memory
- Message passing

‹ Teilen von Ressourcen up Identität ›

Identität

By rac
Published: 12.01.2008 - 17:24

Wer bin ich?

- Vergleiche anhand von PThread Handle
- pthread_self -> eigene Identität
- pthread_equal -> Vergleich mit anderer Identität

‹ Kommunikation up Terminierung ›

Terminierung

By rac
Published: 12.01.2008 - 17:27

- Prozess wird am Ende von main() beendet
- "main_entry" und "main_exit"
- werden zu jedem Programm gelinkt
- zuständig für Rückgabe von Exit Wert
- Befreiung von Systemressourcen

- Thread wird am Ende der entsprechenden Funktion beendet
- alternativ pthread_exit() oder pthread_cancel()

‹ Identität up Exit Status & Rückgabewert ›

Exit Status & Rückgabewert

By rac

Published: 12.01.2008 - 17:31

- Pthread kann joinable oder detached sein > Rückgabewert

- pthread_join() nur für joinable Pthread möglich

```
int pthread_join( pthread_t th, void **thread_return);
```

- Rückgabewert liefert entweder pthread_exit() oder die Funktion, welche pthread_create() ausführt

- pthread_create() gibt (void *) zurück

- evtl. Wert umwandeln (cast)

- Achtung: den Wert PTHREAD_CANCELED nicht verwenden! Dieser ist per Def. weder eine gültige Adresse (Pointer!) noch NULL.

‹ Terminierung up Bibliotheksaufrufe & Fehler ›

Bibliotheksaufrufe & Fehler

By rac
Published: 12.01.2008 - 17:39

Pthread Bibliotheksaufrufe geben bei Erfolg Null und ansonsten einen Fehlercode zurück, die Fehlercodes sind in errno.h definiert.

```
1.
#include <errno.h>

2.
#include <stdio.h>

3.
...

4.
if (rtn = pthread_create(...)) {

5.
   /* Fehler ! */

6.
   fprintf(stderr, "Fehler: pthread_create, ");

7.
   if (rtn == EAGAIN)

8.
          fprintf(stderr, "Ressourcen reichen nichtn");

9.
```

```
   else if (rtn == EINVAL)

10.
       fprintf(stderr, "Ungültige Argumenten");

11.
   exit(1);

12.
}

13.
/* kein Fehler! */

14.
...
```

alternativ kann der Klartext des Fehlers ausgegeben werden, wenn die Funktionalität zur Verfügung steht

```
1.
#include <string.h>

2.
#include <stdio.h>

3.
...
```

```
4.
if (rtn = pthread_create(...)) {

5.
   fprintf(stderr, "Fehler: pthread_create, %sn", strerror(rtn));

6.
   exit(1);

7.
}

8.
/* kein Fehler! */

9.
...

10.
/* oder alternativ */

11.
if (rtn = pthread_create(...)) {

12.
   perror("Fehler aufgetreten: ");

13.
   exit(1);

14.
}
```

‹ Exit Status & Rückgabewert up Warum Threads und nicht Prozesse? ›

Warum Threads und nicht Prozesse?

By rac

Published: 12.01.2008 - 17:40

- Prozesse teuerer (Setupzeit, Speicherverbrauch, IPC Kontextswitches)

- Threads einfacher

- grosser Teil kann in Userspace erledigt werden (pthreads Bibliothek, Kommunikation, Synchronisation)

‹ Bibliotheksaufrufe & Fehler up Aufgabenbereiche für Threading ›

Aufgabenbereiche für Threading

By rac
Published: 12.01.2008 - 17:52

- Potentieller Parallelismus
- Unterschiedliche Ressourcen werden verwendet
- Ist unabhängig vom Resultat eines anderen Tasks

Zu Beachten:
Maximale Konkurrenz und minimale Synchronisation.
Je mehr Abhängigkeiten, um so mehr geblockte Tasks, welche aufeinander warten.

- Überschneidende oder blockierende I/O
- Währenddessen andere Arbeit durchführbar?
- Zuweisung einer I/O Aufgabe zu einem Thread

- Starke CPU Beanspruchung
- Kryptografische Funktionen, Matrizen, Kompression. etc.
- Während ein oder mehrere Tasks Berechnungen durchführen, kann das Programm auf I/O reagieren
- Eventuell Zuordnung einer CPU zu einer Berechnung

- Asynchrone Ereignisse
- Zufällige Intervalle zwischen Daten I/O
- Benutzer I/O, Netzwerk Aktivität, Hardware Interrupts, Sensoren

- Behandlungsroutinen können in einem Thread gekapselt werden

- Realtime sheduling
- Einige Aufgaben sind wichtiger (haben höhere Priorität)
- Schnellere Reaktionszeit
- Feste Bearbeitungszeit
- Ausführung an spezifischen Zeitpunkten

‹ Warum Threads und nicht Prozesse? up Beispiele für Multithreaded Applikationen ›

Beispiele für Multithreaded Applikationen

By rac
Published: 12.01.2008 - 18:17

- Server: Datenbanken, Fileserver, Printserver, HTTPd, P2P
- Rechnen & Signalprocessing
- Realtime für Server & Multitasking Apps
- effizienter als Multiprocessing
- Scheduling
- weniger Komplexität bei asynch. Programmen
- Threads warten auf Ereignisse, serielles Programm springt zwischen Kontexten durch Interrupts

‹ Aufgabenbereiche für Threading up Modelle ›