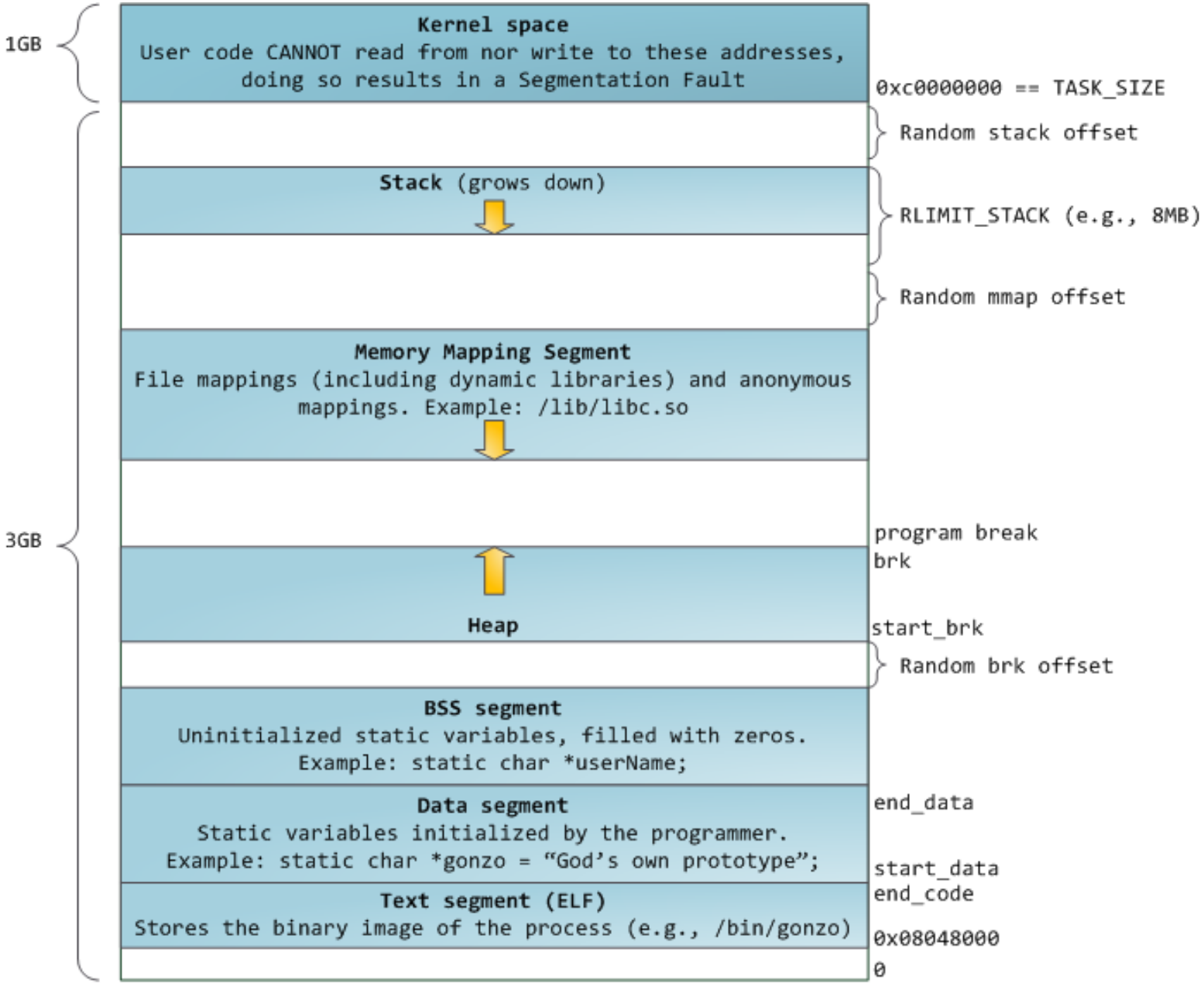
1. Kernel-Level und User-Level Threads
2. Beispiel POSIX Threads
3. Performance: `fork` versus `pthread_create`
4. Beispiel Java-Threads

Wiederholung:

Ein Prozeß (process, task) ist ein in Ausführung befindliches Programm bestehend aus

- einer Folge von Maschinenbefehlen, die durch das ausgeführte Programm festgelegt sind (program code, text section) ,
- dem Inhalt des Stapel-Speichers (stack), auf dem temporäre Variablen und Parameter für Funktionsaufrufe verwaltet werden,
- dem Inhalt des Speichers, in dem die globale Daten des Prozesses gehalten werden (data section) und
- Verwaltungsinformationen, die den aktuellen Bearbeitungszustand (Kontext) beschreiben.



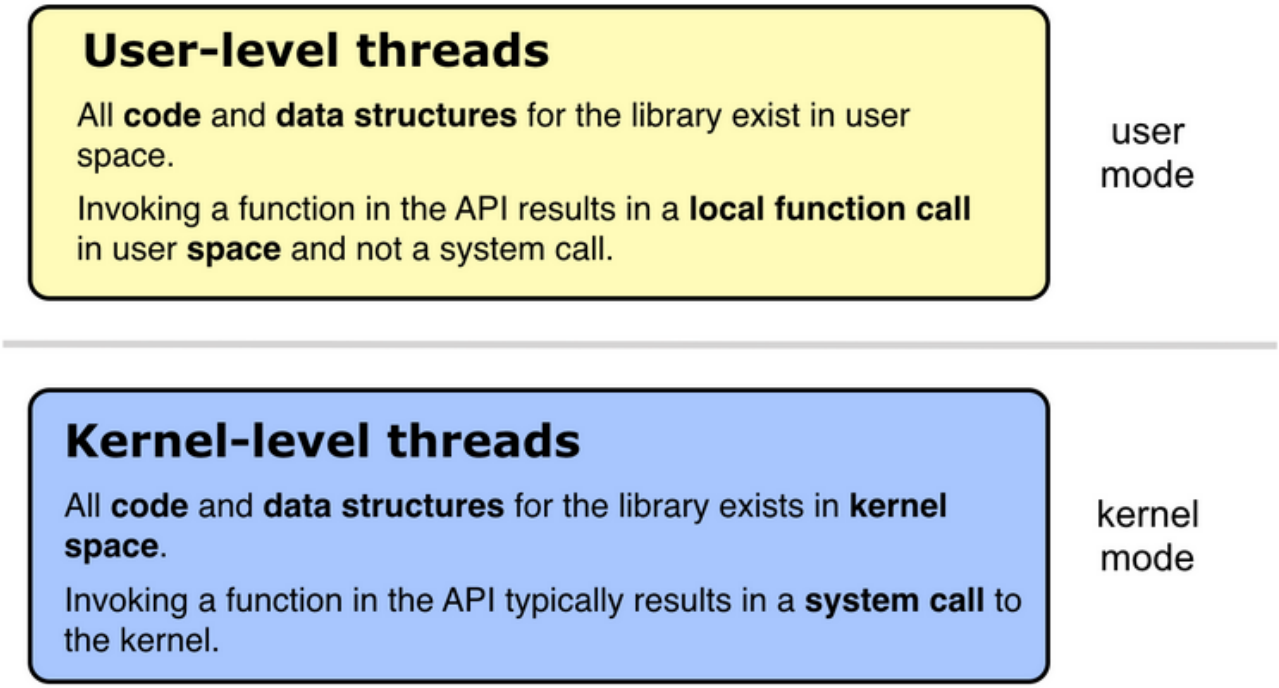
Quelle: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Wiederholung:

Definition 19: Ein Leichtgewichtprozeß (Thread) ist eine Ausführungseinheit mit minimalen Zustandsinformationen (z.B. Programmzähler, Registerinhalte, Thread-Priorität).

Definition 20: Eine Task ist eine Verwaltungseinheit, in der die Systemressourcen wie z.B. Hauptspeicher, I/O-Kanäle etc., die einer oder mehreren Threads zugeteilt sind, zusammengefaßt sind.

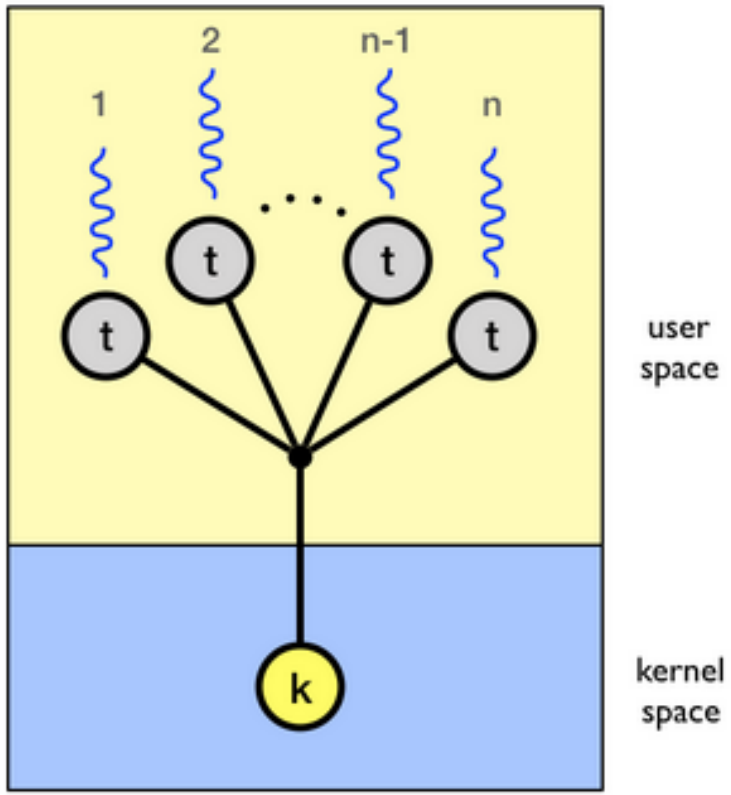
User-Level versus kernel-level-thread



Quelle: [1]

Zeitreise:

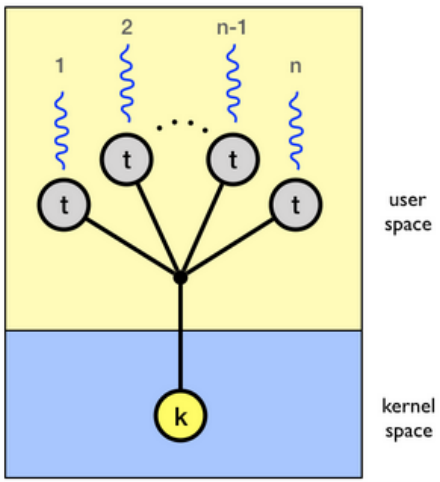
- Betriebssysteme der 80er Jahre unterstützen **Prozesse**, keine Threads



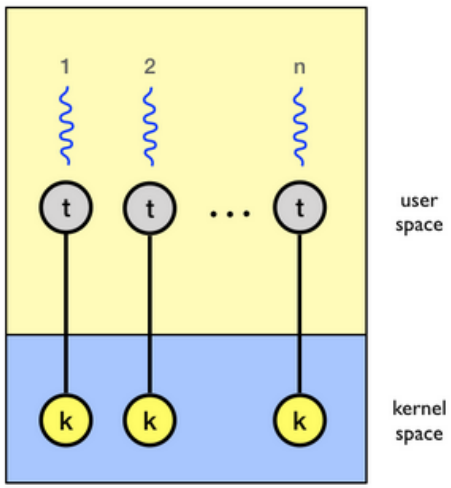
Quelle: [1]

- Betriebssysteme der 90er Jahre (SunOS, Solaris, Windows NT, ...) kennen erstmals Unterstützung für kernel-level Threads

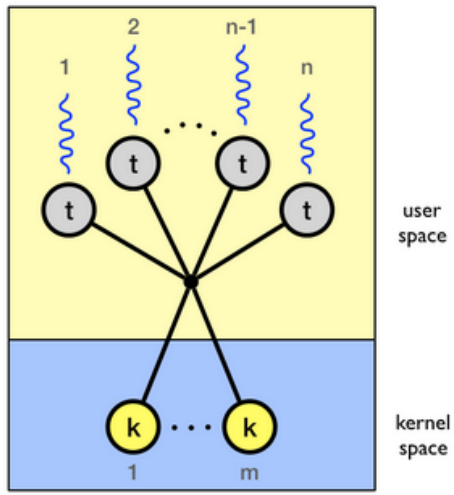
$\Longrightarrow$ Implementierung: User-level Threads können auf kernel-level Threads gemappt werden:



Many-to-one



One-to-one



Many-to-many

Beispiel Solaris Quelle: [3]

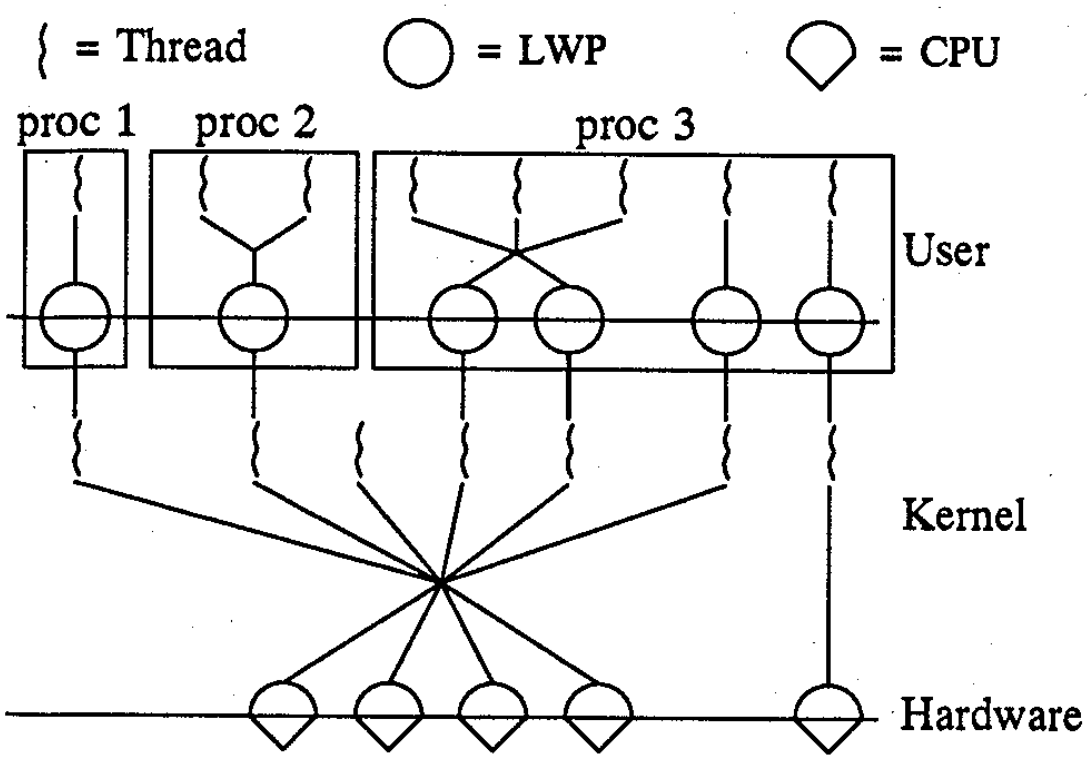


Figure 1: Multi-thread architecture examples

Die User-level Threads sind für das OS unsichtbar und werden von der Laufzeitumgebung auf sogenannte Leichtgewichtprozesse gemappt. [3]

Quellen:

[1] Operating systems, Spring 2018, Department of information technology, Uppsala university

[2] Brett Slatkin: *Effective Python - 90 Specific Ways to Write better Python*, second Edition, Pearson Addison-Wesley, 2020.

[3] Eykholt et al. (SunSoft, Inc.): *Beyond Multiprocessing:Multithreading the SunOS Kernel*, Summer '92 USENIX

[4] Donovan, Kernighan: *The Go Programming Language*, Addison-Wesley, 2016.

Wie schreibe ich Multi-threaded Anwendungen?

1. Threadbibliotheken als Ergänzung zu gängigen Programmiersprachen.
   Beispiel: POSIX-Threads, SunOS 4.1x LWP, ...
2. (Parallele) Sprachen mit neuer Laufzeitumgebung.
   Beispiel: Cedar, Mesa, Java, OpenMP für C/C++/Fortran, ...
3. C++11 Sprachunterstützung für Multi-Threaded Anwendungen

std::thread

Defined in header <thread>

```
class thread;        (since C++11)
```

The class thread represents a single thread of execution. Threads allow multiple functions to execute concurrently.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, std::terminate is called. The top-level function may communicate its return value or an exception to the caller via std::promise or by modifying shared variables (which may require synchronization, see std::mutex and std::atomic)

std::thread objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may not be associated with any thread objects (after detach).

Wie setzt die Laufzeitumgebung das Thread-Konzept um?

1. **At the beginning: n-1-Mapping**
   - Das BS weiß nichts über Threads und sieht nur einen Prozeß mit einem Ausführungsstrang.
   - Die *Laufzeitumgebung* verwaltet und plant (scheduled) User-Level-Threads.
   - Die Laufzeitbibliothek verwaltet eine Thread-Tabelle im User-Space.
2. **90er: n-m-Mapping**
   Beispiel: Sun/Solaris: 2-stufiges System
   1. Kernel unterstützt LWPs (Lightweight Processes).
   2. Es werden zwei Threadbibliotheken (POSIX libpthread, Solaris libthread) angeboten.

   User-level Java Threads werden auf LWPs abgebildet, die auf einen oder mehrere Kernel-Threads abgebildet werden.

21. Jh. (Mehrprozessor-/Multicoresysteme): 1-1-Mapping[14]

Jeder User-Level-Thread wird auf einen Kernel-Level-Thread abgebildet. Das Betriebssystem verwaltet/scheduled die Threads.

Wer macht was?

1. Java
2. Go (Go-Routinen sind User-Level-Threads, Go-Compiler versucht ein 1-1-Mapping)
3. Python

[14] Nicht verwechseln mit dem 1-1-Mapping zwischen Prozessen und Cores in KPP

Beispiel: POSIX libpthread

Die libpthread-Spezifikation enthält Funktionen für:

- Thread-Verwaltung (Erzeugen, Löschen, ...),
- Synchronisation (Mutex-, Bedingungsvariable),
- Scheduling (prioritätengesteuertes FIFO, zeitscheibengesteuertes Round Robin),
- Signalbehandlung, ...

Ausgewählte POSIX-Methoden

- pthread_create()
- pthread_self()
- pthread_exit()
- pthread_join()
- pthread_kill()

- pthread_mutex_lock()
- sem_init()
- sem_wait()
- sem_trywait()
- sem_up()
- sem_destroy()

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

Setzen der Attribute:

joinable $\longleftrightarrow$ detach

Joinable := Ein anderer Thread kann den Exit-Status mit `pthread_join()` abfragen.
Im Fall eines *joinable* Threads werden

- Thread Control Block
- Thread Stack

solange gespeichert, bis ein anderer Thread `pthread_join` aufruft.

Alternativ kann der Thread auf detach gesetzt werden: Die Ressourcen eines *detached* Threads werden nach Beendigung sofort aufgeräumt.

`attr` = NULL $\Longrightarrow$ Default-Attribute werden gesetzt, d.h. der Thread ist joinable und hat die *Default Scheduling Policy* (Timesharing, nicht Realtime)

```c
/* gcc -O -Wall beispiel_posix.c -lpthread -o beispiel_posix */

#define _REENTRANT                  /* basic 2-lines for threads */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define SLEEP_TIME 10
#define NUM_THREADS 5

void *sleeping (void *);            /* thread routine */

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

  /* Argument of thread function is pointer to integer */
  int *arg = malloc(sizeof(int));
  if (arg == NULL) {
    perror("Could not allocate memory for arg");
    exit(EXIT_FAILURE);
  }
  *arg = SLEEP_TIME;

  pthread_t tid[NUM_THREADS];       /* array of thread IDs */
```

```
  for (i = 0; i < NUM_THREADS; i++){
    status = pthread_create (&tid[i], NULL, sleeping,(void *) arg);
    if (status != 0){
      printf("Error␣pthread_create(),␣error␣code␣%i\n",status);
      exit(EXIT_FAILURE); /* terminates the process with all thread
    }
  }
  for (i = 0; i < NUM_THREADS; i++) {
    status = pthread_join (tid[i], NULL);
    if (status != 0){
      printf("Error␣pthread_join(),␣error␣code␣%i\n",status);
      exit(EXIT_FAILURE);
    }
  }

  printf("main()␣reporting␣that␣all␣%d␣threads␣have␣terminated\n",i
  return 0;
} /* main */
```

```
void *sleeping (void *sleep_time){
  /* pthreadtypes.h : */
  /* typedef unsigned long int pthread_t; */
  printf("thread %lx belongs to process %d \n",
         (unsigned long int) pthread_self(), (int) getpid());
  printf ("thread %lx sleeping %d seconds ...\n",
         (unsigned long int) pthread_self(), *(int *) sleep_time);
  /* Application logic: argument points to integer */
  sleep (*(int *) sleep_time);
  printf ("\nthread %lx awakening\n",
         (unsigned long int) pthread_self());
}
```

Bemerkung: Die Funktion `sleeping` hat im Beispiel einen undefinierten Rückgabewert.
Soll der Rückgabewert in der Anwendungslogik abgefragt werden, geschieht dies im `pthread_join`. Der zweite Parameter von `pthread_join` ist ein Zeiger auf den Rückgabewert.

Ausgabe auf einem Linux-System:

```
> ps -eLf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
schnor    4522  2904  4522  0    6 21:56 pts/2    00:00:00 ./a.out
schnor    4522  2904  4523  0    6 21:56 pts/2    00:00:00 ./a.out
schnor    4522  2904  4524  0    6 21:56 pts/2    00:00:00 ./a.out
schnor    4522  2904  4525  0    6 21:56 pts/2    00:00:00 ./a.out
schnor    4522  2904  4526  0    6 21:56 pts/2    00:00:00 ./a.out
schnor    4522  2904  4527  0    6 21:56 pts/2    00:00:00 ./a.out
schnor    4528  2904  4528 99    1 21:56 pts/2    00:00:00 ps -eLf
```

When used with -L, the NLWP (number of threads) and LWP (thread ID) columns will be added.

- POSIX Thread mittlerweile in Sprachumfang von C enthalten. Die Zeile

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

  ist überflüssig.
- C++11 Sprachunterstützung für Multi-Thread-Anwendungen basiert auf der Vorgänger-Threadbibliothek Boost Thread Library

Thread Safety

Problem: Was passiert, wenn mehrere Threads nebenläufig dieselbe Funktion aufrufen?

Definition 21: Eine Funktion heißt thread-safe bzw. reentrant, wenn sie das korrekte Ergebnis liefert, auch wenn sie von mehreren Threads nebenläufig aufgerufen wird.

Beispiel: POSIX 1003.1: Das POSIX Application Interface enthält System Calls und Bibliotheksfunktionen. Im Standard wird allgemein von *functions* gesprochen.

Die POSIX 1003.1 Funktionen sind nicht alle thread-safe. Ausnahmen sind z.B. `crypt`, `encrypt`, `getenv`, `rand`, `drand48`, ...

Einige Systeme bieten thread-safe Variante an: `funktionsname_r`, beispielsweise `rand_r`. Will man die thread-safe Variante benutzen, muß dies dem Präprozessor mitgeteilt werden.

```
#define _REENTRANT
```

Threads in Python ...

... ja, aber!

`module threading`

```
from threading import Thread
class FactorizeThread(Thread):
```

Die Referenzumsetzung, der Python-Interpreter CPython, benutzt einen Global Interpreter Lock (GIL), der garantiert, daß stets nur einer der Threads läuft.
$\Longrightarrow$ Die Maschine verhält sich wie ein 1-Core-System :-(
Die Threads laufen nebenläufig, aber nicht parallel[15].
$\Longrightarrow$ Daher für Parallelverarbeitung[16] ungeeignet

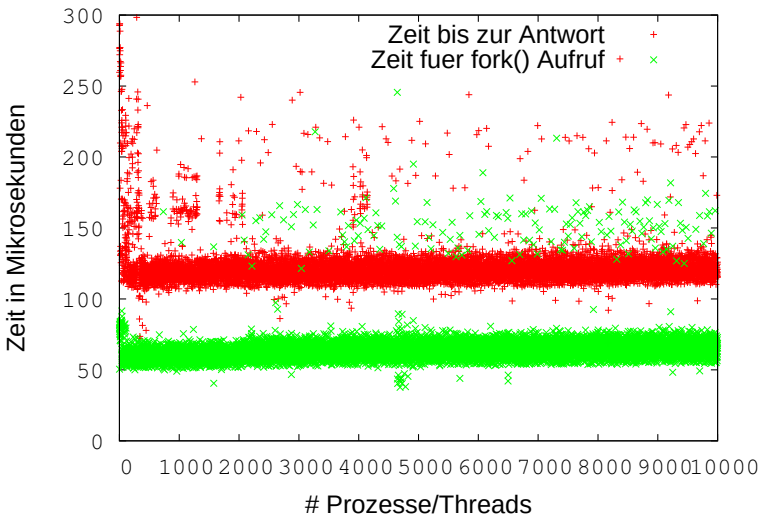
Stattdessen: `Subprocess Module`, um parallele Kindprozesse zu starten

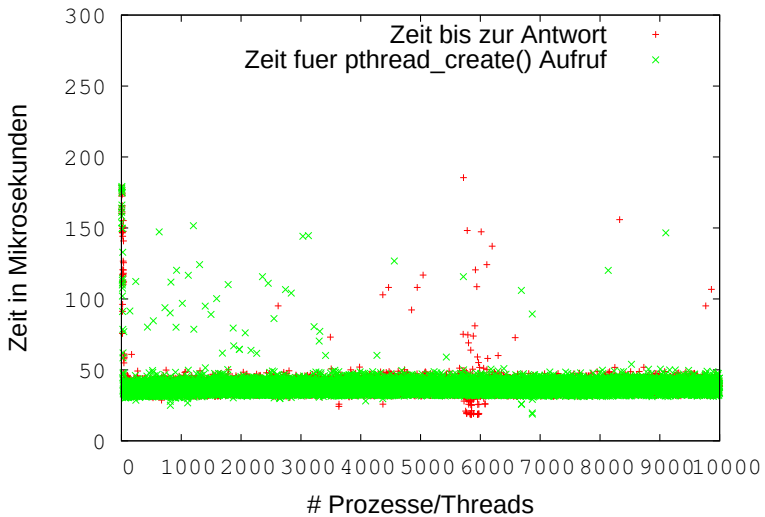
[15] Achtung: Auch nebenläufige Threads, die auf denselben Daten arbeiten, müssen synchronisiert werden.

[16] Wenn Performance relevant ist, sollte man eh eine andere Programmiersprache wählen.

pthread_create versus fork: Was ist schneller?

Experiment: Parent erzeugt in einer Schleife 10000 Kindprozesse bzw. Threads und wartet jedes Mal auf eine Reaktion des erzeugten Kindprozesses bzw. Threads (Parent liest ein Zeichen aus einer Pipe, das das Kind in die Pipe schreibt).





Messungen auf dem node011 des Leibniz-Clusters: Intel Xeon E5520 Quadcore 2.26 GHz, Debian GNU/Linux 6.0, Version 2.6.32-5-amd64, EGlibc 2.11.3

- sehr gute Skalierung
- `fork()`-Aufruf benötigt durchschnittlich ca. 60 μs. Der `pthread_create`-Aufruf hingegen durchschnittlich ca. 40 μs.
- Die Antwortzeit des Kindprozessess liegt bei ca. 120 μs
- Die Antwortzeit des erzeugten Threads liegt bei ca. 40 μs
- Warum ist die Antwortzeit beim Kindprozeß höher?
  Schreiben in die Pipe erzwingt ein Copy-on-Write.

Quelle: Sijing You: *Analyse der Lastverteilungsverfahren des Apache HTTP-Servers*, Bachelorarbeit, Uni Potsdam, 2014

Beispiel: Java-Threads

Java unterstützt nativ das Thread-Konzept!

Durch Vererbung können neue Klassen definiert werden. Die neue Klasse

- erbt die Methoden der Basisklasse,
- kann bestehende Methoden überschreiben,
- neue Methoden hinzufügen.

$\Longrightarrow$ Klasse `Thread` mit Methode `run`

Zugriff auf Elemente und Methoden eines Objekts:

- `public`: Zugriff sowohl von außen als auch von allen Nachfolgern aus erlaubt.
- `private`: Zugriff nur innerhalb der Klasse möglich.
- ...

Beispiel: Die Klasse ParallelSleepTest erzeugt 4 Threads, die SLEEP_TIME schlafen sollen

```
public class ParallelSleepThread extends Thread {

    int sleeptime;
    int myid;
    public ParallelSleepThread(int t, int id) {
        sleeptime = t;
        myid = id;
    }

    public void run() {
        try {
            sleep(sleeptime);
        }
        catch (InterruptedException e){
        }
        System.out.println("Demo-Thread␣"+myid+"␣awakes");
    }
}
```

```
//import ParallelSleepThread;
public class ParallelSleepTest {
  private static final int nThreads = 4;
  private static int SLEEP_TIME = 5000;

  private static ParallelSleepThread threads[];

  public static void main(String args[]){
    threads = new ParallelSleepThread[nThreads];
    for (int i = 0; i < nThreads; i++) {
      System.out.println("creating␣thread␣"+i);
      threads[i] = new ParallelSleepThread(SLEEP_TIME, i);
      System.out.println("starting␣thread␣"+i);
      threads[i].start();
    }

    for (int i = 0; i < nThreads; i++) {
      try {
        threads[i].join();
      }
      catch (InterruptedException e) {
        System.out.println("join␣"+i+"␣caught␣InterruptedException
      }
    }
  }
}
```

Methoden und Zustandsübergänge

NEW: When a new thread object is created, for example with Thread t = new Thread(a); its thread enters the *new* state.

RUNNABLE: When the start method of a new thread is called, for example,

```
t.start();
```

the thread enters the *runnable* state. All runnable threads in a Java program are organized by the JVM into a data structure called the runnable set, also sometimes called the round robin queue or the ready queue. The code in the `run` method of the thread will be executed by the CPU.

RUNNING: Whenever the thread is actually executing – that is, allocated CPU cycles by the computer's operating system scheduler – the thread is in the *running* state. *If a thread in the running state calls its* `yield` *method, it gives up the CPU and is put back in the runnable set in the runnable state. (nicht mehr seit Release 5.0)*

NOT RUNNABLE: A thread enters the *NOT RUNNABLE* state when it calls its `sleep()` method (TIMED_WAITING), when it calls `wait()` inside a synchronized method of some object (WAITING), or when it performs some blocking input-output operation (BLOCKED) (like reading from the keyboard). From the NOT RUNNABLE state, a thread reenters the runnable state and is put back in the runnable set.

Depending on how it blocked, this happens when its `sleep` method call completes, when its blocking input-output request is fulfilled by the operating system, or when some other thread calls `notify()` or `notifyAll()` in a synchronized method of the object in which the blocked thread earlier called `wait()`.

SUSPENDED: *A runnable or running thread enters the suspended state when its suspend method is called, for example,*

```
t.suspend();
```

A thread can supend itself or be suspended by some other thread. From the suspended state, a thread reenters the runnable state and is placed in the runnable set when its resume method is called by some other thread.

`t.resume();` *(nicht mehr seit Release 5.0)*

SUSPENDED-BLOCKED: *If a blocked thread is suspended by another thread, it enters the suspended-blocked state. If the blocking operation subsequently completes, it enters the suspended state. If, on the other hand, the thread is resumed by another thread before the blocking operation completes, it reenters the blocked state. (nicht mehr seit Release 5.0)*

DEAD: *A thread terminates and enters the dead state when its run method completes execution or when its* **stop** *method is called, usually in some other thread.*

`t.stop();` *(nicht mehr seit Release 5.0)*

TERMINATED: The `run`-method terminates.

Java hat dem Anwendungsprogrammierer ursprünglich Primitiven an die Hand gegeben für **User-level-Thread-Scheduling**: `yield()`, `suspend()`, `resume()`, `stop()`. Dies hat sich als zu fehleranfällig erwiesen und wird daher nicht mehr unterstützt.

Thread-Zustandsdiagramm

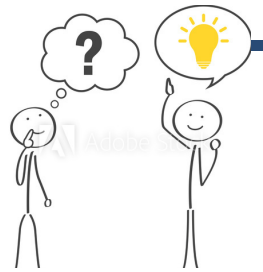
Anwendungsbeispiele:

1. Die JVM benutzt den Main-Thread, der die `main`-Methode ausführt, und verschiedene Threads für *Housekeeping Tasks* (Garbage Collection, Finalization).
2. Brian Goetz: *Frameworks introduce concurrency into applications by calling application components from framework threads.*
   - Servlets and Java Server Pages: The servlet framework is designed to handle all the infrastructure of deploying web applications. A request arriving at the server is dispatched to the appropriate servlet or JSP. Each servlet represents a component of application logic, and in high-volume web sites, multiple clients may require the services of the same servlet at once.
     Hence, servlets have to be thread-safe.
   - Swing: Grafikbibliothek für Java für die Programmierung von grafischen Benutzeroberflächen, ist nicht thread-safe. Herzstück ist der Event Dispatch Thread, der die Ereignisverarbeitung durchführt.

Beispiel: Android

- Dalvik Virtual Machine bzw. seit Android 5.0
  Ahead-of-time-Compiler Android Runtime (ART): speziell für Mobilgeräte angepasste JVM
- Single Thread Model: Alle Komponenten einer Anwendung laufen in einem Thread, dem Main-Thread (auch UI Thread genannt).
- Folglich blockieren lang dauernde Aktionen das gesamte User Interface.
- Daher wird empfohlen, länger dauernde Operationen in einem neuen Thread zu starten (Background oder Worker Thread genannt).

Was Sie jetzt wissen sollten ...



- Was teilen sich Threads einer Anwendung: Register, Stack, Heap und/oder Text Segment?
- Wozu braucht man das Thread-Attribut *joinable*?
- Achtung: Viele Bibliotheksfunktionen sind nicht thread-safe!
- Die Referenzumsetzung von Python macht es sich (zu) einfach, weil ...
- Skaliert die Prozeß- bzw. Threaderzeugung auf Linux?
- In welchen Anwendungen werden Threads eingesetzt?