Betriebssysteme und Netzwerke Vorlesung 6

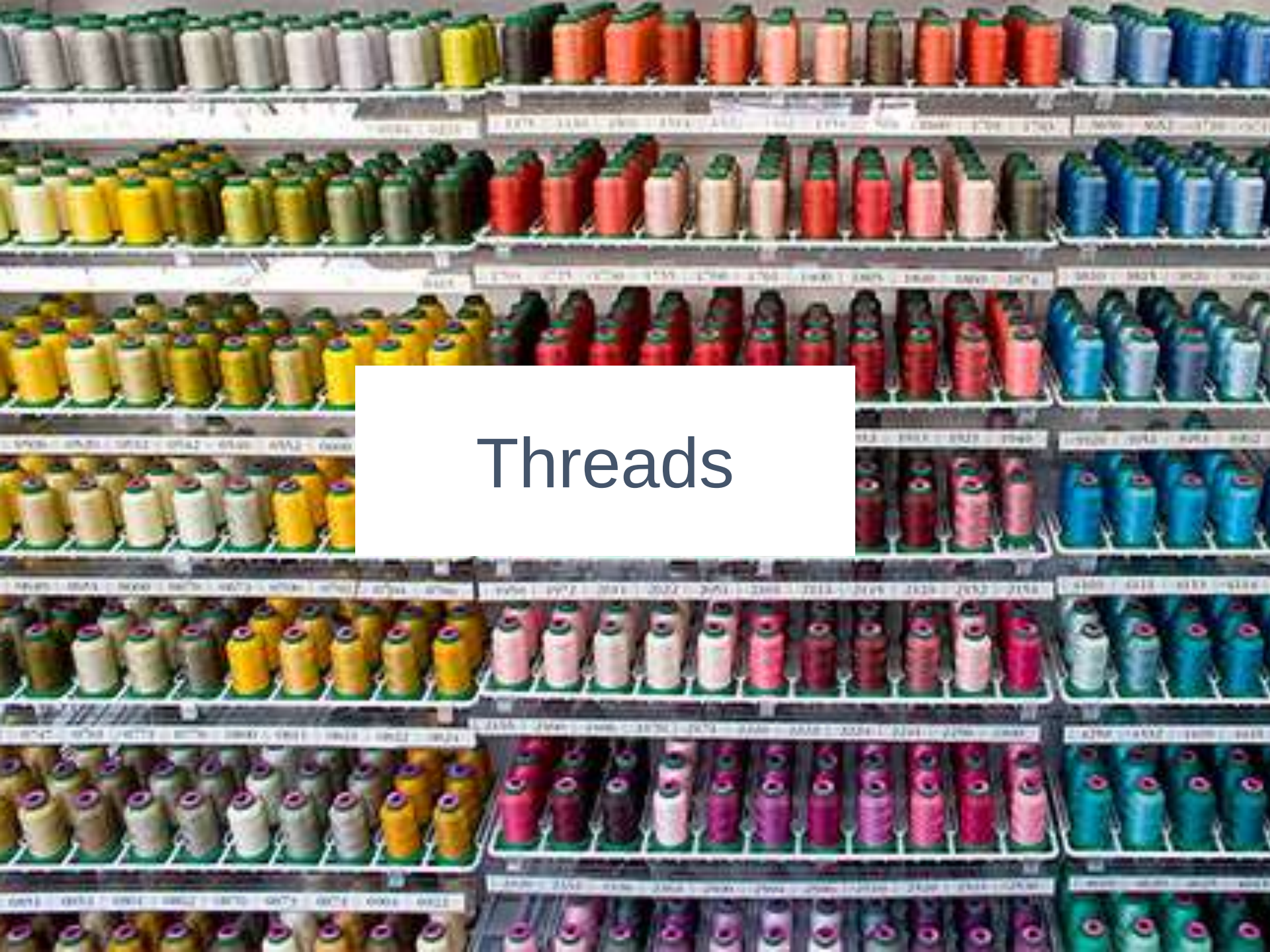
Artur Andrzejak

Umfragen: https://pingo.coactum.de/301541

Klausur (Achtung – Änderungen Zeit/Ort)

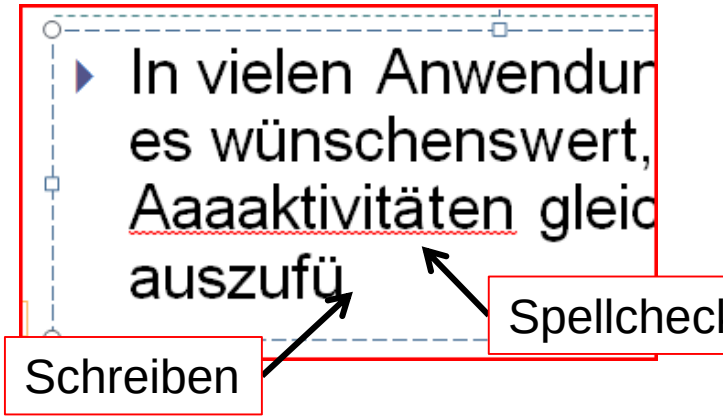
- **Datum**: **22. Juli 2019** (Montag)
  - Letzte Woche in der Vorlesungszeit
- **Zeit**: 09.30 Uhr bis ca. **11.00** Uhr
- **Orte**:
  - Großen Hörsaal der Chemie (INF 252)
  - HS Ost der Chemie (INF 252)
- **Inhalt**: beide Bereiche, d.h. Betriebssysteme und Netzwerke
- Keine Hilfsmittel sind zugelassen
- Personalausweis / Pass mitnehmen
- Bitte anmelden, sonst verfällt die Klausurzulassung!
  - Siehe http://www.informatik.uni-heidelberg.de/?id=335

Pipes und Dateideskriptoren: Siehe Folien der VL 5


Threads

Motivation

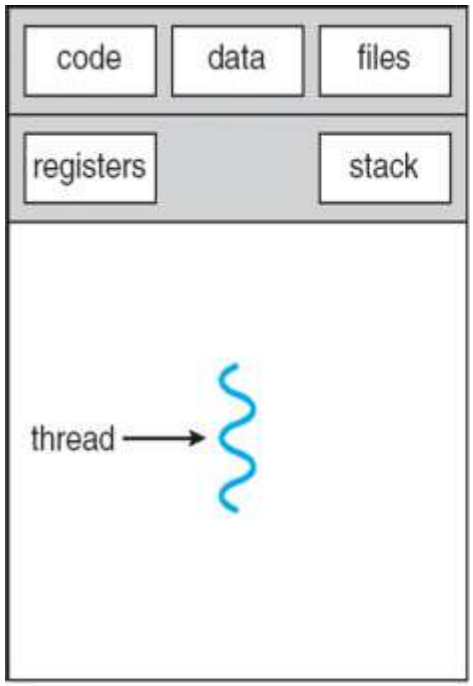
- Es ist oft wünschenswert, mehrere Aktivitäten gleichzeitig auszuführen
- Warum nicht mehrere Prozesse?



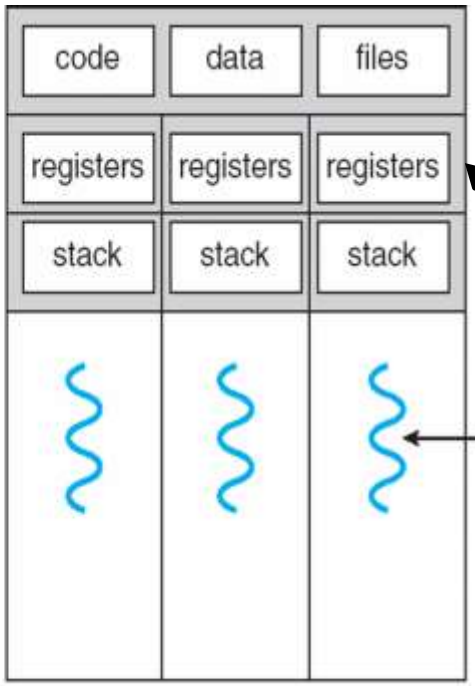
- Probleme
  - Man möchte die selben Daten nutzen: deren Austausch zwischen Prozessen ist i.A. aufwändig
  - Unnötige Replikation des Speichers: Mehrere PCBs, Text-Segmente, …
  - Wechsel zwischen den Prozessen kostet i.A. mehr Zeit

Threads und Multithreading

- Deshalb hat man „Miniprozesse“, sog. **Threads** eingeführt (deutsch: Ausführungsstrang, Faden, Aktivitätsträger)
- Moderne BS erlauben viele Threads per Prozess: **Multithreading**



Single-threaded

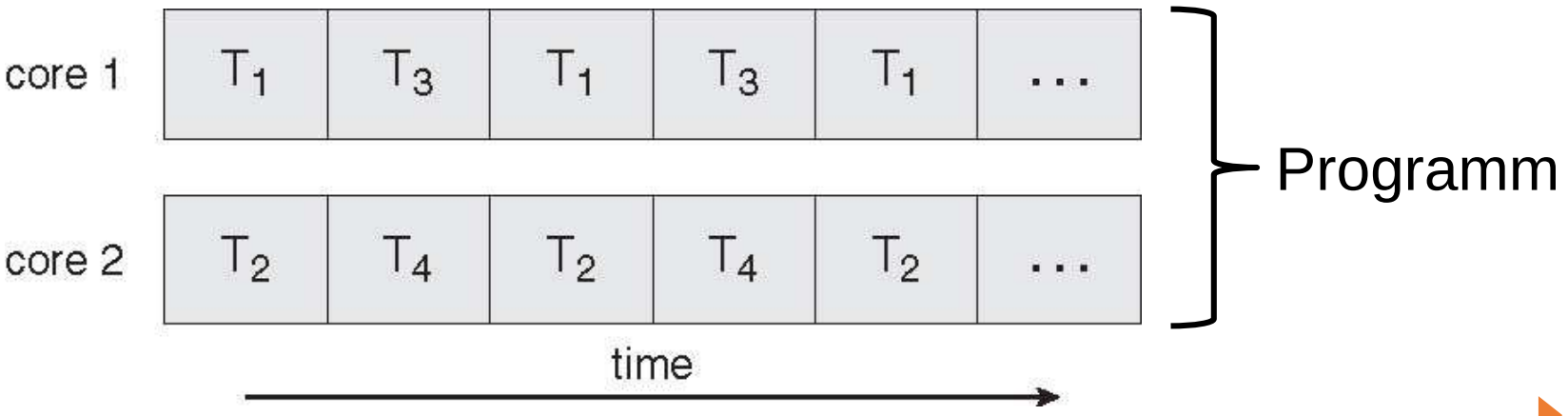


Multi-threaded

Jeder Thread (eines Prozesses) hat einen eigenen Stack und Register(kopien), aber alle teilen sich denselben Adressraum (eines Prozesses)

Vorteile von Threads

- Höhere Reaktionsfreudigkeit (**responsiveness**)
  - Verschiedene Funktionen eines Programmes werden in verschiedenen Threads ausgeführt
- Einfache Kommunikation zwischen Threads
  - Gemeinsamer Speicher des Prozesses als Default
- Skalierbarkeit
  - Ermöglicht bessere Nutzung moderner Multicore-CPUs

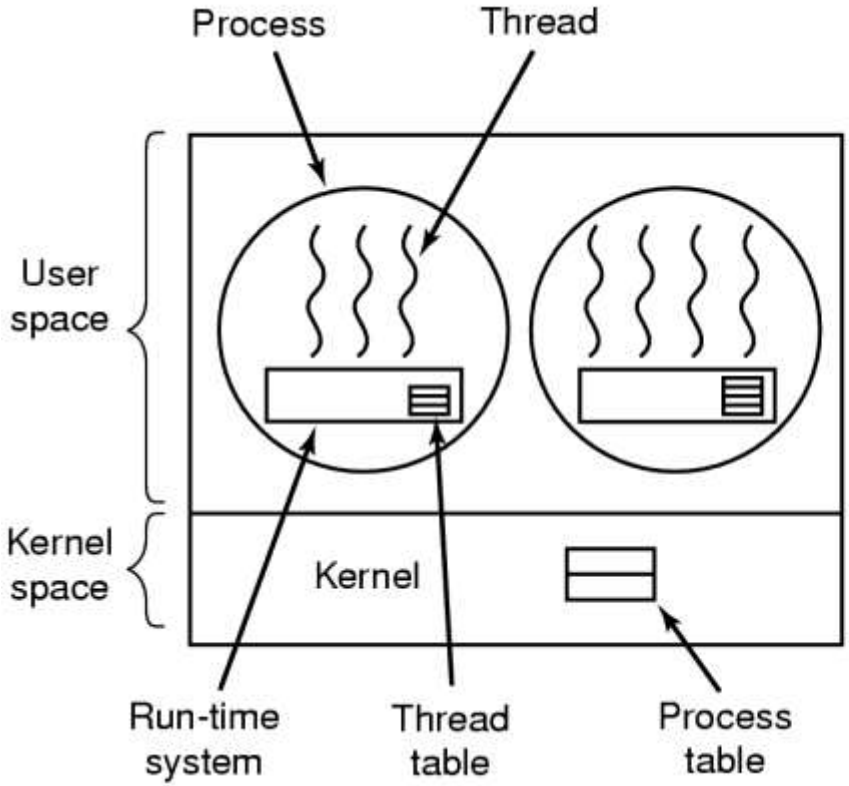


Leistung: Prozesse vs. Threads

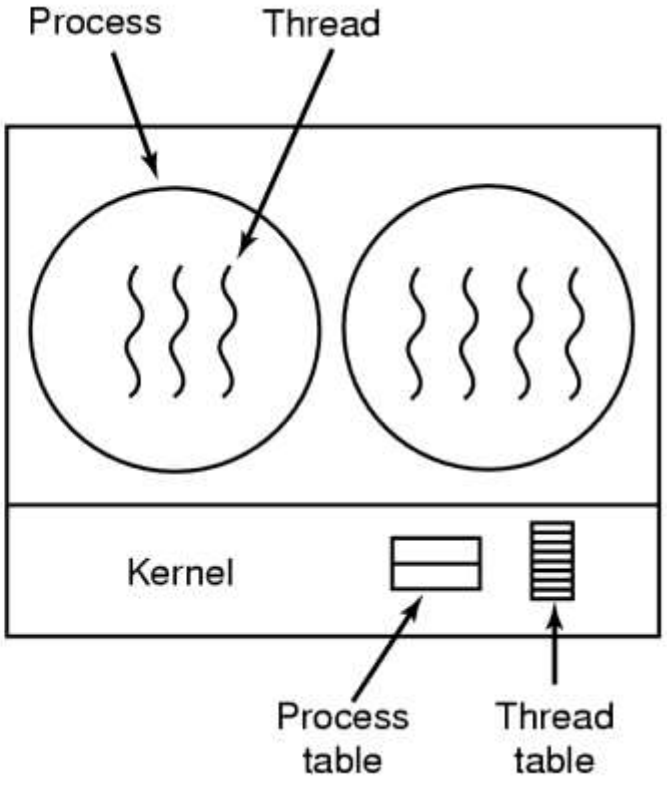
- Threadwechsel ist i.A. schneller als Prozesswechsel
- Betriebssysteme - Unterschiede
  - Auf Solaris dauerte ein Prozesswechsel ca. 30-mal so lange wie ein Threadwechsel
  - Windows: Prozesswechsel ~~ist~~ war 10-100 langsamer
  - Unter Linux ist Prozesswechsel fast so schnell wie Threadwechsel (eines der Systemkonzepte)
    - Prozesswechsel nur langsamer, da die Caches (L1, L2, …) invalidiert werden

Verwaltung von Threads

- Die Verwaltung der Threads kann entweder in dem **Benutzerraum** (**user space**) oder in dem Kern des BS (**kernel space**) stattfinden



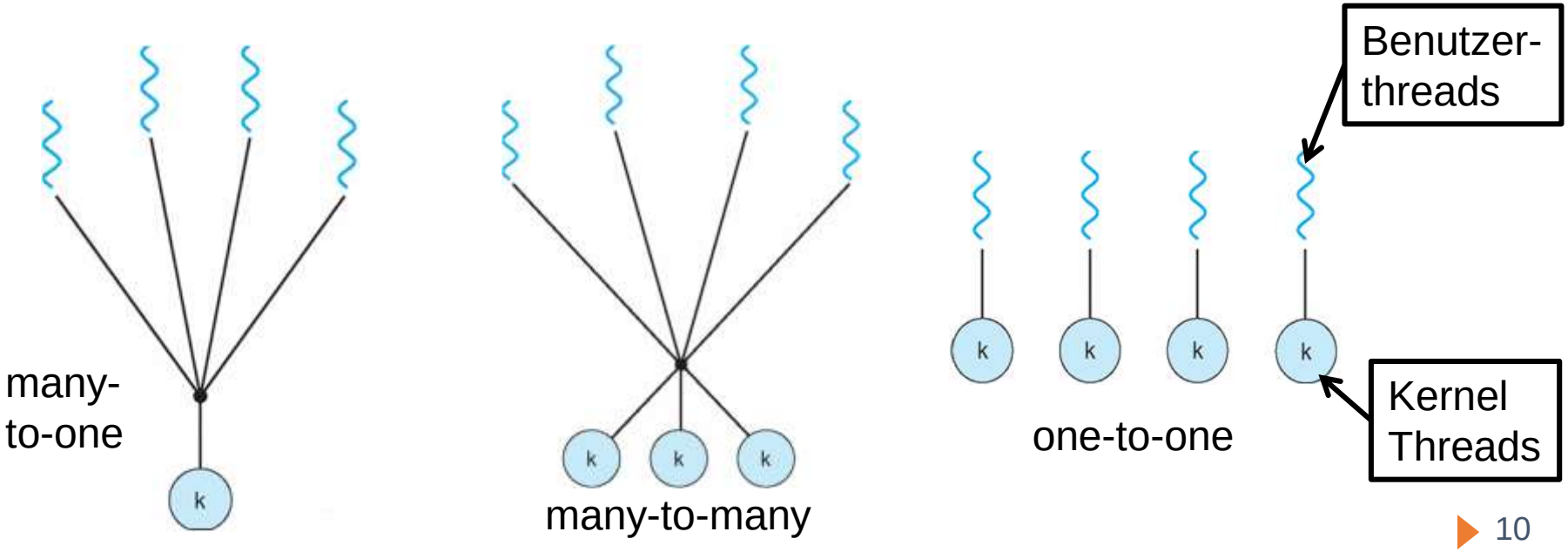
Im Benutzerraum



Im Kern des BS

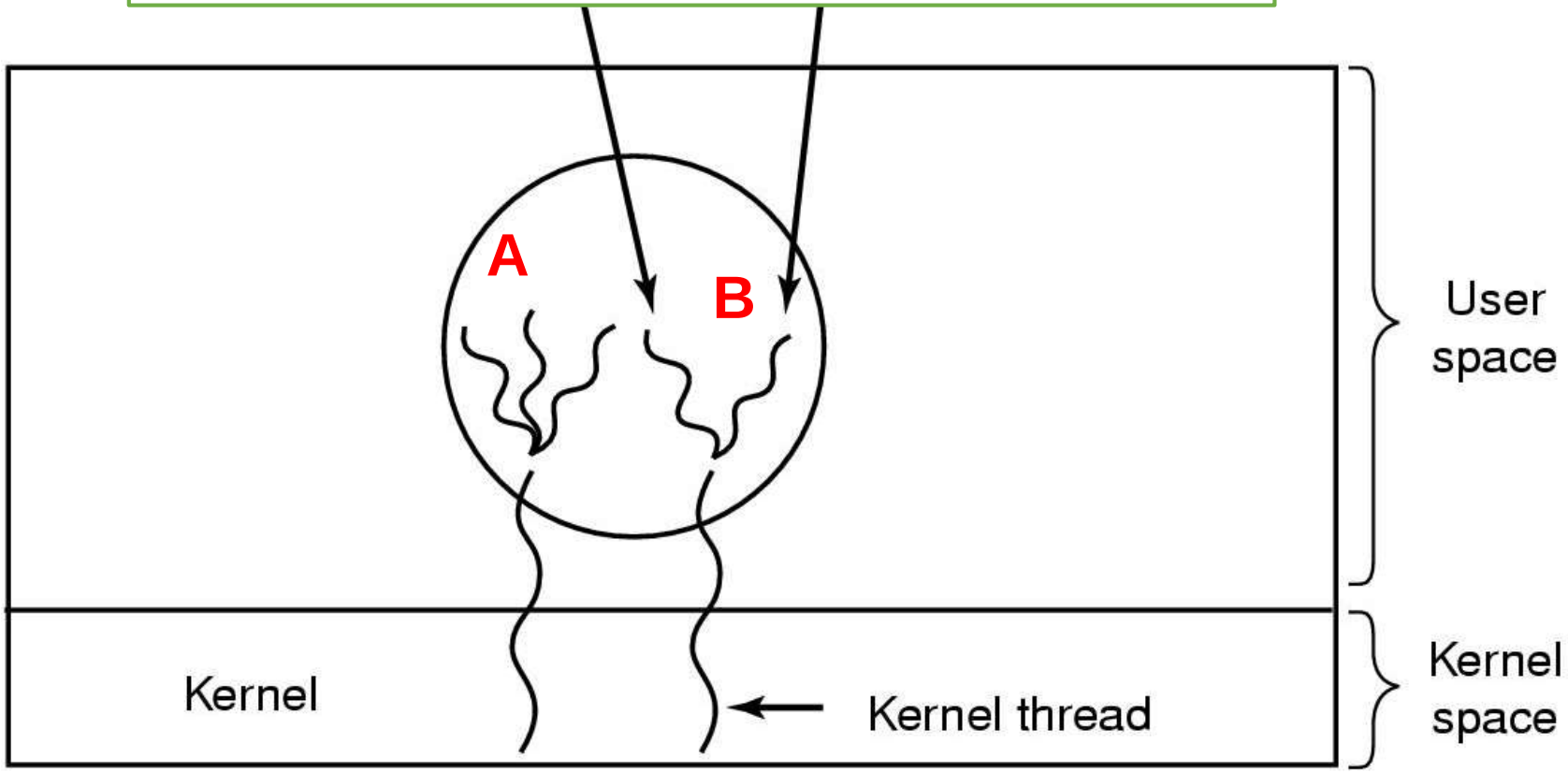
Gemischte Ausführungsmodelle

- Eine Bibliothek kann beide Implementierungen mischen
  - **one-to-one** Model: Jeder vom Benutzer erzeugter Thread entspricht einem (eigenen) Kernel Thread
  - **many-to-many**: Mehrere Benutzerthreads werden auf einige Kernel Threads abgebildet
  - **many-to-one**: Alle Benutzerthreads von einem Kernel Thread (oder einem Prozess) bedient



Many-To-Many Visualisierung

Threads in Gruppe A können sich bei I/O gegenseitig blockieren, aber blockieren nicht die Threads in der Gruppe B



Threads in Benutzermodus

- **User-level Threads** bzw. **User Threads**: Verwaltung der Threads erfolgt in dem CPU-Benutzermodus
  - Nicht nötig, beim Wechsel in den Kernel zu springen =>
  - Der Wechsel ist sehr schnell!
- Aber es gibt auch (entscheidende) Nachteile – welche?
- Ein Thread kann durch einen Systemaufruf (z.B. mit Warten auf I/O) das ganze Programm blockieren
  - Da das BS nicht weiß, dass der Prozess viele Threads hat, blockiert es den gesamten Prozess
- Multi-Core CPUs werden nicht ausgenutzt – nur 1 Core auf einmal wird dem Prozess zugeordnet

Threads in Kernmodus

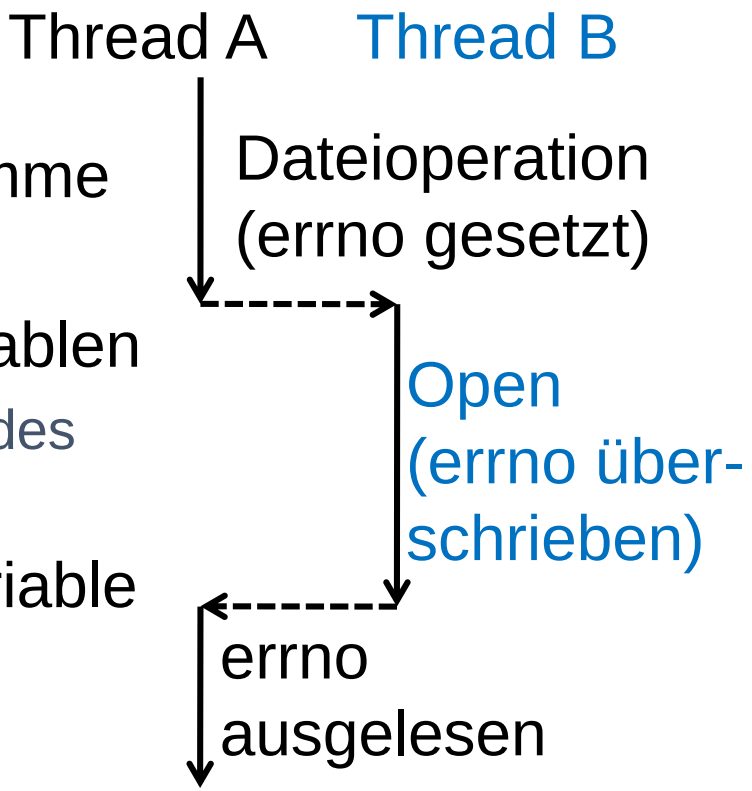
- **Kernel Threads**: Die gesamte Verwaltung der Threads und ihrer Daten erfolgt im Kernel des BS
  - Der BS-Kernel übernimmt die schwierigsten Aufgaben, und macht das Program dadurch einfacher
- Erlauben **preemptives Scheduling**, d.h. Threads können andere nicht blockieren, auch bei I/O
- Langsamer als User Threads, da bei jedem Wechsel der Sprung in den Kernel erfolgt
- Kernel Threads sind der de-facto Standard für die Implementierung von Threads
  - Verfügbar in allen wichtigen BS, und am häufigsten benutzt

Dauer eines Threadwechsels

- Video “User-level threads..... with threads. - Paul Turner - Google”, https://www.youtube.com/watch?v=KXuZi9aeGTw
  - Teil 1: Zeit des Wechsels (8:40 bis 9:33 min:sec)
  - Teil 2: Was sind die Gründe? (11:00 bis 14:20 min:sec)
- Erklärung: “pin/pinned” = thread oder Prozess wird nur auf einer festgelegten Core/CPU ausgeführt

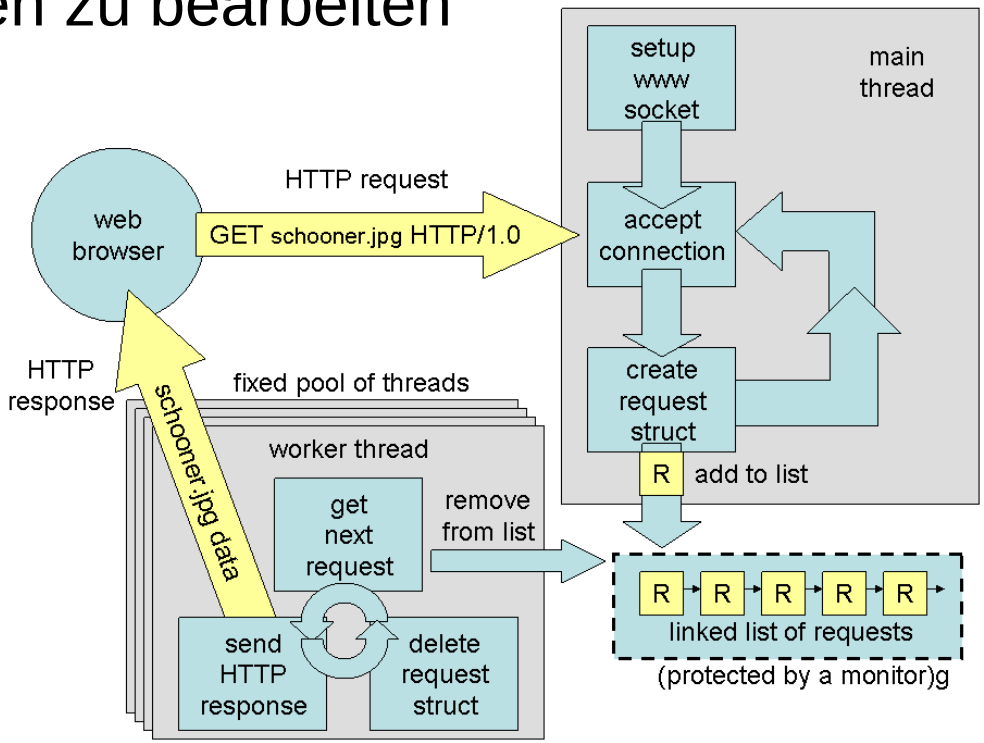
Probleme beim Umschreiben des Codes

- Eine Herausforderung ist es, existierende (Singlethread) Programme in Multithread Programme umzuschreiben
- Ein Problem sind z.B. globale Variablen
  - Z.B. **errno** in C: speichert den Code des letzten Fehlers
- Es kann passieren, dass diese Variable zwischen Dateioperation und dem Auslesen durch Ausführung eines anderen Threads überschrieben wird
- Mögliche Lösung: Führe Thread-private „globale“ Variablen ein



Thread Pools

- Man kann eine Anzahl von Threads in einem Pool (d.h. Menge) erstellen, wo sie auf Arbeit warten
- Es ist etwas schneller, ein Thread aus dem Pool zu holen, als ein neues zu erstellen
- Nützlich z.B. bei Webservern: Threads als „worker“, um die Anfragen zu bearbeiten



Threads: Implementierungen

POSIX Pthreads

- Ein **POSIX-Standard** (IEEE 1003.1c) **API** für das Erstellen von Threads und ihre Synchronisierung
  - API legt nur das Verhalten der Thread-Bibliothek fest
  - Die Implementierung ist die Sache der Entwickler
  - Es gibt diese als User Threads oder Kernel Threads
- Vorhanden in UNIX-Betriebssystemen FreeBSD, NetBSD, GNU/Linux, Mac OS X, Solaris
  - Es gibt auch eine Windows Version: pthreads-w32 (Teilmenge der API)
- Circa 100 Prozeduren, alle mit Prefix „**pthread_**“
  - Thread Management – erzeugen, löschen usw.
  - Dienste für Synchronisation

POSIX Pthreads – Erzeugen

- **pthread_create** (thread, attr, start_routine, arg)
- Argumente
  - thread: Ein eindeutiger Identifikator, der von der Routine zurückgegeben wird (Typ **pthread_t**)
  - attr: Ein Objekt zum Setzen der Attribute eines Threads, oder NULL für Standardwerte
  - start_routine: Zeiger zu der Funktion, die als der Thread-Code ausgeführt wird
  - arg: Das einzige Argument, das an *start_routine* übergeben werden kann; NULL, wenn sie keine Argumente braucht

```
rc = pthread_create( &threads [i], NULL, TaskCode,
                                        (void *) &thread_args[i]);
```

POSIX Pthreads – Terminieren

- Möglichkeiten der Terminierung
  - Thread kehrt aus *start_routine* (via return) zurück
  - Thread ruft **pthread_exit**() auf
  - Thread wird ausgeschaltet durch einen anderen Thread via **pthread_cancel** ()
  - Der ganze Prozess terminiert durch Aufruf von **exec** oder **exit**
- int **pthread_join** (pthread_t *thread*, void ***value_ptr*)
  - Der aufrufende Thread wartet, bis *thread* terminiert
  - value_ptr zeigt auf Daten, die *thread* beim Terminieren übergeben hat

```
for (i=0; i<NUM_THREADS; ++i) {
    rc = pthread_join (threads[i], NULL);
```

POSIX Pthreads - Beispiel

```
#include <pthread.h>, <stdio.h>,  <stdlib.h>, <assert.h>
#define  NUM_THREADS     5

void *TaskCode (void *argument) {
   int tid;  tid = *((int *) argument);
   printf ("It's me, dude! I am number %d!\n", tid);
   return NULL;
}

int main (int argc, char *argv[]) {
   pthread_t threads [NUM_THREADS];
   int thread_args [NUM_THREADS]; int rc, i;
```

POSIX Pthreads– Beispiel /2

```
  for (i=0; i<NUM_THREADS; ++i) {/* create all threads */
      thread_args[i] = i;
      printf("In main: creating thread %d\n", i);
      rc = pthread_create( &threads[i], NULL, TaskCode,
           (void *) &thread_args[i]);
      assert(0 == rc);
    }
    /* wait for all threads to complete */
    for (i=0; i<NUM_THREADS; ++i) {
      rc = pthread_join (threads[i], NULL);
      assert (0 == rc); }
    exit(EXIT_SUCCESS);
}
```

Threads in Linux

- Prozesse und Threads werden beide in Linux als **Tasks** bezeichnet
- Die erste Implementierung von POSIX Threads hieß **LinuxThreads**
- Die aktuelle (bessere) Implementierung heißt **Native POSIX Thread Library** (**NPTL**)
  - Sehr effizient, z.B. erzeugen von 100k Threads in 2 Sekunden – früher in 15 Minuten
  - Zuerst in Red Had Linux 9, in Linux Kernel ab Version 2.6

Threads in Linux

- Falls man nicht die POSIX Threads nutzt (sollte man aber), werden in Linux neue Threads mit dem Systemaufruf **clone**() erzeugt
- int **clone** (int (***fn**) (), void ***stack**, int **flags**, void ***arg**)
  - **fn**: Zeiger auf die Funktion, mit der der Thread startet
  - **stack**: Zeiger auf das Ende eines Speicherbereichs für den Stack des Threads
  - **flags**: legen fest, was von dem Erzeuger vererbt wird
    - Dateisysteme; Speicherraum; Signal-Handler; offene Dateien
    - Damit kann man das Spektrum von “normalen” Thread bis zu einem neuen Prozess (fast wie bei fork()) abdecken
  - **arg**: Argumente von fn

Zusammenfassung

- Dateideskriptoren und Ein/Ausgabenumlenkung
- Threads:
  - Einführung
  - Implementierungen in Linux und Windows
- Quellen
  - Silberschatz et al., Kapitel 4 (Threads & Concurrency)
  - Tanenbaum et al., Kapitel 2 und 7 (Linux – case study)

Danke schön.

Threads:
Zusätzliche Folien

Behandlung von Signalen

- **Signale** (**signals**) werden in UNIX-Systemen verwendet, um einem Prozess mitzuteilen, dass ein bestimmtes Ereignis eingetreten ist
  - z.B. INT Signal (**SIGINT**) - „terminiere“, erzeugt z.B. via „ctrl+c“
- Ein **Signal-Handler** ist eine spezielle Programmroutine, die Signale abarbeitet
- Was sind mögliche Alternativen in Bezug auf Threads?
  - Liefere das Signal an jeden Thread in dem Prozess
  - Liefere das Signal an bestimmte Threads im Prozess
  - Lege einen bestimmten Thread fest, der alle Signale für den Prozess verarbeitet

Alternativen bei Verhalten

- Soll das fork() alle Threads duplizieren, oder nur den aufrufenden Thread?
- Sollen Threads private Daten haben?
  - Nützlich, wenn man keine Kontrolle über die Erzeugung hat (Pools!)
- Terminierung, bevor der Thread fertig ist
  - **Asynchrone** Terminierung (**asynchronous cancellation**): beendet den Zielthread sofort
  - **Aufgeschobene** (latente) **Terminierung** (**deferred cancellation**): Erlaubt es dem Thread, periodisch zu prüfen, ob er „sich terminieren“ soll

Threads: Implementierungen Zusatzfolien

Implementierung

- Kernelthreads gibt es in:
  - Windows XP/ Vista, Solaris, Linux, Tru64 UNIX, Mac OS X
- Wichtigste Bibliotheken für die Benutzer sind
  - POSIX **Pthreads**, **Win32** threads, Java threads
- Implementierung in **one-to-one** Model
  - Windows NT/XP/Vista, Linux, Solaris 9 und später
- Implementierung in **many-to-many** Model
  - Solaris vor Version 9, Windows NT/2000 mit **ThreadFiber** Bibliothek
- Bibliotheken für **many-to-one** Model
  - Solaris Green Threads, GNU Portable Threads

Windows Threads

- Threads werden im one-to-one Modell implementiert, d.h. jeder Benutzerthread wird von einem Kernel Thread umgesetzt
- Jeder Thread enthält
  - Eine Thread-ID
  - Registersatz (Kopien der CPU Register)
  - Separate Benutzer und Kernel-Stacks
  - Privaten Datenspeicher
  - Der Registersatz, Stack, und private Daten werden als **Kontext** (**context**) bezeichnet

Windows Thread API

Sicherheits-Attribute

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize, // Stack-Größe (0 = default)
    LPTHREAD_START_ROUTINE  lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags, // z.B. CREATE_SUSPENDED
    LPDWORD lpThreadId ); // z.B. &_tid
DWORD WINAPI ThreadProc (LPVOID lpParameter);
```

Start-Adresse des Tasks

- lpStartAddress: Pointer auf Anfang des auszuführenden Codes
- lpParameter: Parameter der **ThreadProc**

Kontrolle des Threads via Handle

```
namespace Thread  {
    class Handle: public Sys::Handle  {
    public:
        Handle (HANDLE h = Sys::Handle::NullValue ()): Sys::Handle (h){}
        void Resume ()  {
            ::ResumeThread (_h); }
        void Suspend ()  {
            ::SuspendThread (_h); }
        void WaitForDeath (unsigned timeoutMs = INFINITE) {
            ::WaitForSingleObject (_h, timeoutMs);  }
        bool IsAlive () const {
            unsigned long code;
            ::GetExitCodeThread (_h, &code);
            return code == STILL_ACTIVE;
} }; }
```

Siehe auch: Windows API Tutorial - Using Threads
http://www.relisoft.com/win32/active.html