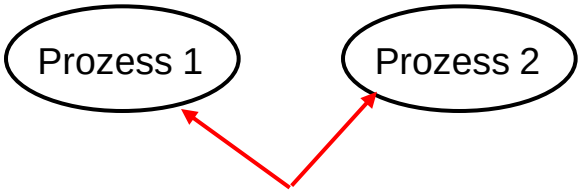
Betriebssysteme
5. Threads

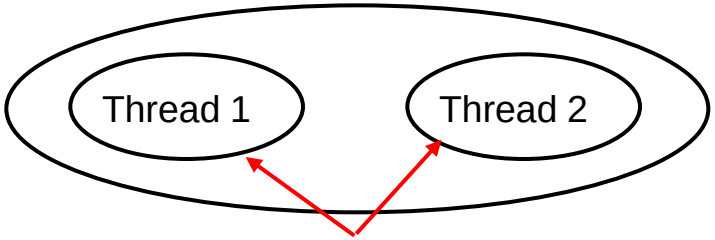
Tobias Lauer

Was sind Threads?

- Thread = engl. „Faden“ (im Sinne von Ablauf-Strang)
- Thread = (Teil-)programm in Ausführung
- Threads sind „leichtgewichtige Prozesse“
- Jeder Prozess kann einen oder mehrere Threads beherbergen
- Was gilt für die Threads eines Prozesses:
  - die Threads laufen (konzeptionell) alle parallel zueinander („nebenläufig“) ab (d.h. als wären sie eigene Prozesse)
  - jeder Thread hat eigenen Threadkontrollblock (Thread-Zustand, Thread-Priorität, Thread-Kontext/Registerinhalte, etc.)
  - Alle Threads haben denselben Adressraum
    1. gemeinsame globale Variable
    2. gemeinsamer Heap
    3. aber: separate Stacks (Kellerstapel)
  - Alle Threads haben dieselben externe Betriebsmittel (z.B. geöffnete Dateien, zugewiesene E/A-Geräte, etc.)
- D.h. nach außen agieren sie wie 1 Prozess, nach innen wie *N* Prozesse mit Zugriff auf gemeinsame Anwendungsdaten

Threads und Prozesse





Prozess 1 und Prozess 2 ...	**Thread 1 und Thread 2 ....**
laufen parallel	laufen parallel
haben jeder eigene lokale Daten, die der jeweils andere nicht sieht	haben jeder eigene lokale Daten, die der jeweils andere nicht sieht
werden i.d.R. unabhängig voneinander erzeugt und terminiert (außer: Prozessfamilie)	ebenfalls unabhängig, jedoch terminieren automatisch alle Threads wenn der beherbergende Prozess terminiert
können nicht auf globale Datenstrukturen des jeweils anderen zugreifen	können auf globale Datenstrukturen des jeweils anderen zugreifen
greifen (i.d.R.) nicht auf die gleichen Betriebsmittel (z.B. Dateien) zu	greifen auf die gleichen externen Betriebsmittel (z.B. Dateien zu)

Warum zusätzlich Threads zu Prozessen?

- Threads sind „billiger“ als Prozesse
  - weniger CPU-Zeit zum Einrichten und Terminieren
  - Scheduling zwischen Threads desselben Prozesses ist schneller
- Threads erlauben die Arbeit auf gemeinsamen Daten (Prozesse brauchen Interprozesskommunikation: aufwändiger, teurer)

Aber Achtung (nichts ist umsonst):

- Anwendungsprogramm verliert natürliche Schutzmechanismen, wie sie zwischen Prozessen existieren
  - ein Thread kann bei falscher Programmierung den Ablauf eines anderen Threads stören
  - Unsynchronisierter Zugriff auf gemeinsame Variable kann zu undefinierten Dateninhalten führen

Beispielprogramm mit Threads

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

void *print_char (void *ch)
{
  int i;
  for (i=0; i<10; i++)
    printf ("%c", *(char*)ch);
  return NULL;
}

int main ()
{
  char ch1='-', ch2='*';
  pthread_t p1, p2;

  pthread_create (&p1, NULL, print_char, &ch1);
  pthread_create (&p2, NULL, print_char, &ch2);

  pthread_join (p1, NULL);
  pthread_join (p2, NULL);

  printf ("\n");
  return 0;
}
```

Thread Code

Thread Erzeugung

Warten auf Thread-Ende

Was würde passieren, wenn dieses Warten fehlen würde?

Beispielprogramm mit Threads

```
public class Threading {

  static void print_char(char ch) {
    for (int i=0; i<50; i++) {
      System.out.print(ch);
    }
  }

  public static void main(String[] args) {
    final char ch1 = '-', ch2 = '*';
    Thread p1 = new Thread() {
                  public void run() { print_char(ch1); }
                };
    Thread p2 = new Thread() {
                  public void run() { print_char(ch2); }
                };
    p1.start(); p2.start();

    try {
      p1.join(); p2.join();
    } catch(InterruptedException e) {};

    System.out.print(" END ");
  }
}
```

Thread Code

Thread Erzeugung

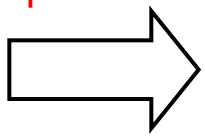
Warten auf Thread-Ende

Was würde passieren, wenn dieses Warten fehlen würde?

Gemeinsame Daten: Vorteile und Probleme

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

char ch;

void *print_stern (void *dummy)
{
  ch = '*';
  sleep (1);
  printf ("%c\n", ch);
  return NULL;
}

void *print_minus (void *dummy)
{
  ch = '-';
  sleep (1);
  printf ("%c\n", ch);
  return NULL;
}

int main ()
{
  pthread_t p1, p2;

  pthread_create (&p1, NULL, print_minus, NULL);
  pthread_create (&p2, NULL, print_stern, NULL);

  pthread_join (p1, NULL);
  pthread_join (p2, NULL);

  return 0;
}
```

Vorgriff auf späteres !



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

pthread_mutex_t mutex;
char ch;

void *print_stern (void *dummy)
{
  pthread_mutex_lock (&mutex);
  ch = '*';
  sleep (1);
  printf ("%c\n", ch);
  pthread_mutex_unlock (&mutex);
  return NULL;
}

void *print_minus (void *dummy)
{
  pthread_mutex_lock (&mutex);
  ch = '-';
  sleep (1);
  printf ("%c\n", ch);
  pthread_mutex_unlock (&mutex);
  return NULL;
}

int main ()
{
  pthread_t p1, p2;

  pthread_mutex_init (&mutex, NULL);

  pthread_create (&p1, NULL, print_minus, NULL);
  pthread_create (&p2, NULL, print_stern, NULL);

  pthread_join (p1, NULL);
  pthread_join (p2, NULL);

  return 0;
}
```

Gemeinsame Daten: Vorteile und Probleme

```
public class Threading_shared_resource {
  static char ch;
  static void print_star() {
    try {
      ch = '*';
      Thread.sleep(1);
    } catch(InterruptedException e) {};
    System.out.print(ch);
  }

  static void print_minus() {
    try {
      ch = '-';
      Thread.sleep(1);
}   } catch(InterruptedException e) {};
    System.out.print(ch);
  }

  public static void main(String[] args) {
    Thread p1 = new Thread() {
                  public void run() { print_minus(); }
                };
    Thread p2 = new Thread() {
                  public void run() { print_star(); }
                };
    p1.start(); p2.start();

    try {
      p1.join(); p2.join();
    } catch(InterruptedException e) {};

    System.out.println(" END ");
  }
}
```

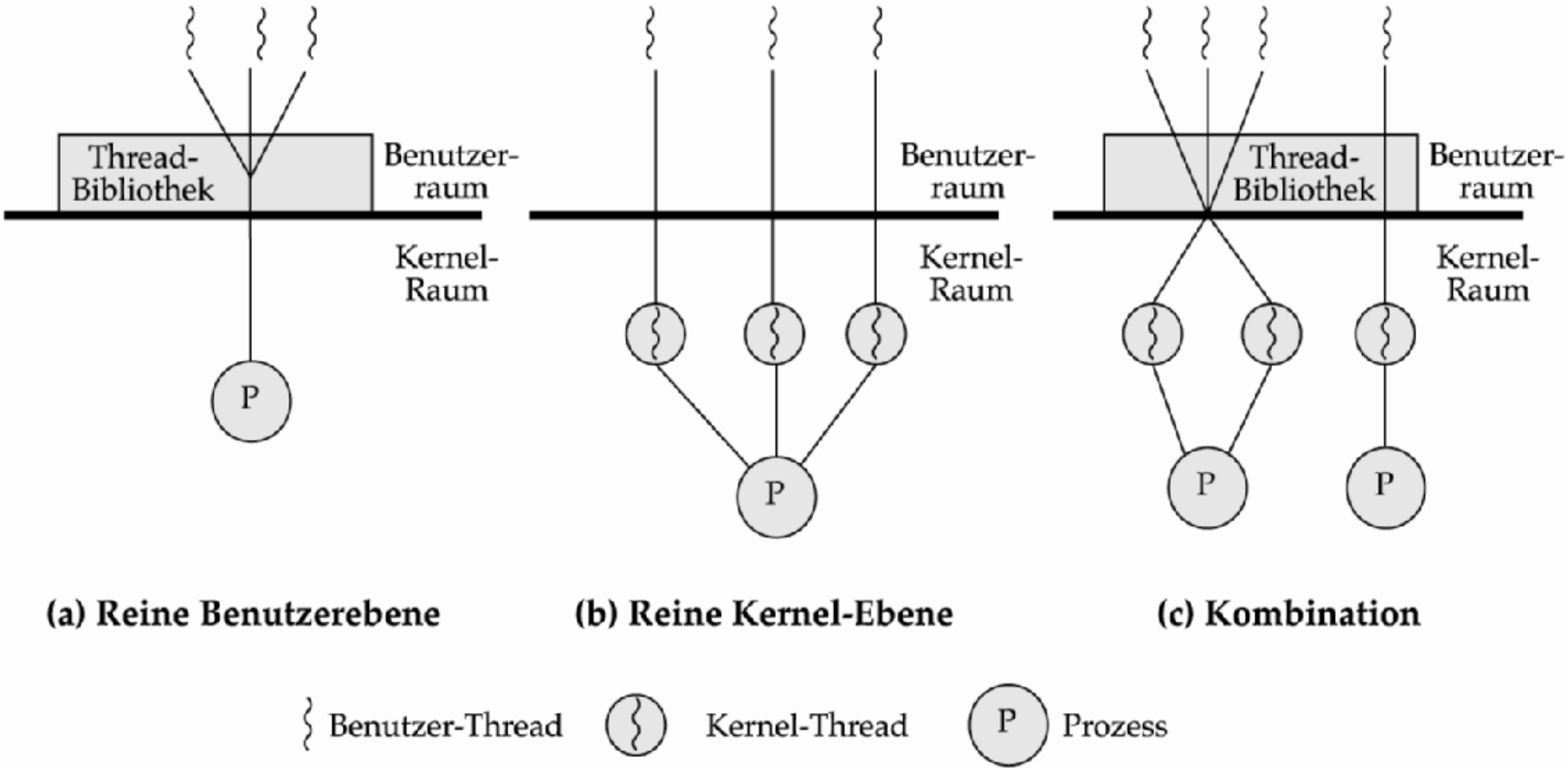
Vorgriff auf späteres !

```
public class Threading_shared_resource {
  static char ch;
  static synchronized void print_star() {
    try {
      ch = '*';
      Thread.sleep(1);
    } catch(InterruptedException e) {};
    System.out.print(ch);
  }

  static synchronized void print_minus() {
    try {
      ch = '-';
      Thread.sleep(1);
}   } catch(InterruptedException e) {};
    System.out.print(ch);
  }

  public static void main(String[] args) {
    Thread p1 = new Thread() {
                  public void run() { print_minus(); }
                };
    Thread p2 = new Thread() {
                  public void run() { print_star(); }
                };
    p1.start(); p2.start();

    try {
      p1.join(); p2.join();
    } catch(InterruptedException e) {};

    System.out.println(" END ");
  }
}
```

Einsatz von Threads

- Prinzipiell ähnliche Aufgabenstellungen wie für Prozesse
- Parallelisierung von Aufgaben (z.B. Pipelining, Foreground/Background tasks, etc.)
- Threads werden eingesetzt für die Ausführung komplexer, mehrteiliger Aufgaben auf gemeinsamen Daten
- Modularisierung von Aufgaben (jeder Thread eine separate Task) oder Parallelisierung EINER Aufgabe auf Multicore-Rechnern
- Threads (innerhalb eines Prozesses) statt Prozessfamilien, um den Zugriff auf gemeinsamen Daten zu erleichtern
- Threads statt Prozesse, wenn es sinnvoll ist, schnell (und billig) einzelne Ablaufstränge zu erzeugen und wieder zu terminieren (z.B. für Server-Requests)

Benutzer-Threads und Kernel-Threads



Quelle: [Stal03]

Benutzer-Threads (Thread-Bibliotheken)

- Unsichtbar für das Betriebssystem
- „Mini-Betriebssystem“ (Dispatcher, Scheduler) als Teil jedes Benutzerprozesses
- Während Prozess aus Sicht des Betriebssystems im Zustand Aktiv bleibt, wechselt der Thread-Scheduler die einzelnen Benutzer-Threads in einem Mini-Zyklus zwischen Aktiv→Blocked→Bereit
- Threads verwenden "Yield"-Operation = freiwilliger CPU-Verzicht
- Wann wechselt der Prozess als Ganzes den Zustand?
  - Z.B. ein Thread verlangt E/A
    → Prozess geht in Blocked (verliert CPU)
    → Alle Threads des Prozesses hören auf zu rechnen
  - Z.B. Prozess hat Zeitscheibe aufgebraucht
    → Prozess geht in Ready (verliert CPU)
    → Alle Threads des Prozesses hören auf zu rechnen
  - Z.B. E/A wird befriedigt
    → Benutzer-Thread-Scheduler erhält Kontrolle
    → alle Threads fangen wieder an zu laufen

Benutzer-Threads: Pro und Contra

- Vorteile
  - Können auch genutzt werden für Betriebssysteme, die selbst kein Thread-Konzept haben (Thread-Bibliothek oberhalb BS)
  - Erfordern keinen Übergang zum Kernel-Modus für Scheduling
    → schneller, billiger
  - Unterschiedliche Scheduling-Strategien für unterschiedliche Anwendungen (= Prozesse) möglich

- Nachteile
  - Blockierende Aufrufe eines Threads (z.B. receive_from_network()) blockieren automatisch den ganzen Prozess und damit alle Threads (Abhilfe: „Jacketing“)
  - 1 Prozess = 1 CPU, d.h. Mehrprozessorsysteme können nicht effektiv genutzt werden

Kernel-Threads

- Betriebssystem „sieht“ die Threads eines Prozesses
- Betriebssystem übernimmt nicht nur das Scheduling des Prozesses, sondern auch das Scheduling der einzelnen Threads
- Threaderzeugung, Threadkommunikation/-synchronisation, etc. = Aufrufe an das Betriebssystem (Kernel Calls)
- Vorteile
  - Betriebssystem kann Threads besser untereinander schedulen
  - Betriebssystem kann Threads auf mehrere Prozessoren verteilen
  - E/A eines einzelnen Threads führt nicht zur Blockade aller Threads im gleichen Prozess
- Nachteile
  - schwergewichtiger (jedes Mal Kernel-Aufruf)
  - Benutzer verliert „Autonomie“ über seine Threads
- Sowohl Windows als auch UNIX/LINUX verwenden Kernel-Threads
- Windows benutzt zusätzlich Benutzerthreads ("Fibre"-Konzept)

Hyperthreading und Multicore-Prozessoren

- Hardware Technologie, die Threading besonders unterstützt
- Hyperthreading/Multithreading CPUs haben
  - mehrere separate Registersätze
  - mehrere separate Programmzähler (Program Counters)
  - gemeinsamen Cache
- Multi-Core CPUs haben zusätzlich
  - separate L1 Caches (CPU-naher Cache)
  - separate Rechenwerke
  - weitere gemeinsame Caches
- CPUs mit Hyperthreading oder Multi-Core Architektur
  - simulieren mehrere „virtuelle“ CPUs
  - erlauben einen höheren Grad an Parallelität als Standard-CPUs
- Hyperthreading bzw. Multicore-CPUs (HW) sind ideal geeignet, um Kernel-Threads (SW) zu unterstützen
  - Threads sind dem Betriebssystem bekannt
    → Betriebssystem kann SW-Threads den HW-Threads zuordnen
  - Threads arbeiten auf gemeinsamen Anwendungsdaten
    → gemeinsame CPU-Caches werden optimal genutzt