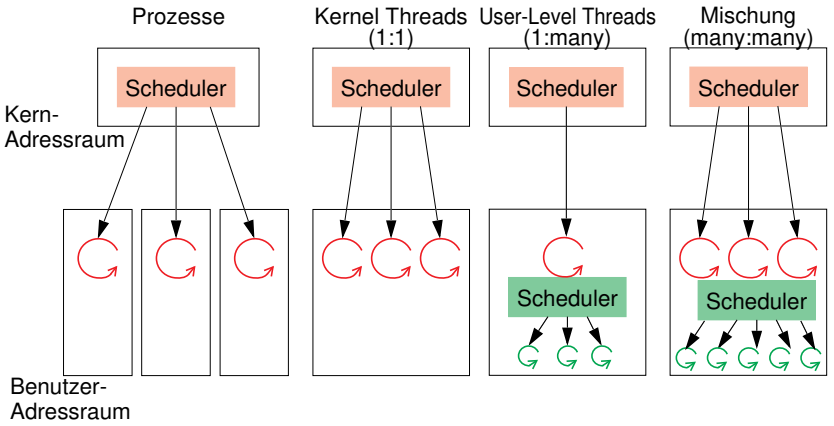
J.10 Thread-Konzepte in UNIX/Linux

- verschiedene Implementierungen von Thread-Paketen verfügbar
  - ➤ reine User-Level-Threads
    eine beliebige Zahl von User-Level-Threads wird auf einem Kernel Thread "gemultiplexed" (*many:1*)
  - ➤ reine Kernel-Level-Threads
    jedem auf User-Level sichtbaren Thread ist 1:1 ein Kernel-Level-Thread zugeordnet (*1:1*)
  - ➤ Mischungen: eine große Zahl von User-Level Threads wird auf eine kleinere Zahl von Kernel Threads abgebildet (*many:many*)
    - \+ User-Level Threads sind billig
    - \+ die Kernel Threads ermöglichen echte Parallelität auf einem Multiprozessor
    - \+ wenn sich ein User-Level-Thread blockiert, dann ist mit ihm der Kernel-Level-Thread blockiert in dem er gerade abgewickelt wird — aber andere Kernel-Level-Threads können verwendet werden um andere, lauffähige User-Level-Threads weiter auszuführen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

J.10 Thread-Konzepte in UNIX/Linux (2)



- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
  - ➥ IEEE-POSIX-Standard P1003.4a

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 pthread-Benutzerschnittstelle

- **Pthreads-Schnittstelle (Basisfunktionen)**:

pthread_create	Thread erzeugen & Startfunktion angeben
pthread_exit	Thread beendet sich selbst
pthread_join	Auf Ende eines anderen Threads warten
pthread_self	Eigene Thread-Id abfragen
pthread_yield	Prozessor zugunsten eines anderen Threads aufgeben

- Funktionen in Pthreads-Bibliothek zusammengefasst
  `gcc ... -pthread`

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 pthread-Benutzerschnittstelle (2)

- Thread-Erzeugung

```
#include <pthread.h>

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

`thread`	Thread-Id
`attr`	Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **`NULL`** für Standardattribute.

Thread wird erzeugt und ruft Funktion **`start_routine`** mit Parameter **`arg`** auf.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 pthread-Benutzerschnittstelle (3)

- Thread beenden (bei return aus `start_routine` oder):

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

  Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe pthread_join)

- Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

  Wartet auf den Thread mit der Thread-ID **thread** und liefert dessen Rückgabewert über **retvalp** zurück.

Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode als Ergebnis zurückgeliefert.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];

int main(int argc, char* argv[]) {
    pthread_t tids[100];
...
    for (i = 0; i < 100; i++)
        pthread_create(tids + i, NULL, mult,
                       (void *)(c + i));
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
...
}

void *mult(void *cp) {
    int j, i = (double *)cp - c;
    double sum = 0;

    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

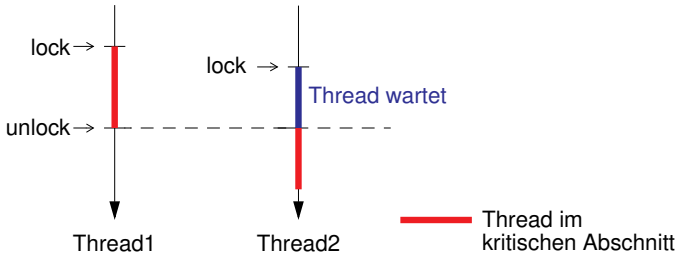
3 Pthreads-Koordinierung

- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
  - ◆ Implementierung durch den Systemkern
  - ◆ komplexe Datenstrukturen, aufwändig zu programmieren
  - ◆ für die Koordinierung von Threads viel zu teuer
- Bei Koordinierung von Threads reichen meist einfache ***Mutex***-Variablen
  - ◆ gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (2)

- ★ **Mutexes**
- Koordinierung von kritischen Abschnitten



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (3)

… Mutexes (2)

- Schnittstelle
  - ◆ Mutex erzeugen

```
pthread_mutex_t m1;
s = pthread_mutex_init(&m1, NULL);
```

  - ◆ Lock & unlock

```
s = pthread_mutex_lock(&m1);
... kritischer Abschnitt
s = pthread_mutex_unlock(&m1);
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (4)

… Mutexes (3)

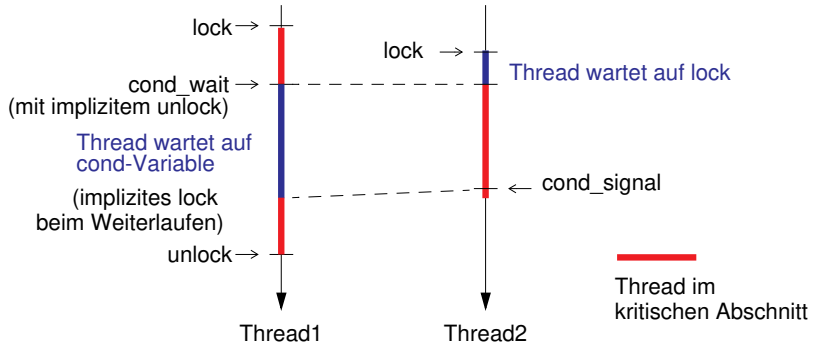
- Komplexere Koordinierungsprobleme können alleine mit Mutexes nicht implementiert werden
  - ➥ Problem:
    - Ein Mutex sperrt die eine komplexere Datenstruktur
    - Der Zustand der Datenstruktur erlaubt die Operation nicht
    - Thread muss warten, bis die Situation durch anderen Thread behoben wurde
    - Blockieren des Threads an einem weiteren Mutex kann zu Verklemmungen führen
  - ➥ Lösung: Mutex in Verbindung mit sleep/wakeup-Mechanismus
  - ➥ **Condition Variables**

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (5)

★ **Condition Variables**

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (6)

… Condition Variables (2)

- Realisierung
  - ◆ Thread reiht sich in Warteschlange der Condition Variablen ein
  - ◆ Thread gibt Mutex frei
  - ◆ Thread gibt Prozessor auf
  - ◆ Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
  - ◆ Deblockierter Thread muss als erstes den kritischen Abschnitt neu betreten (lock)
  - ◆ Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (7)

… Condition Variables (3)

- Schnittstelle
  - ◆ Condition Variable erzeugen

```
pthread_cond_t c1;
s = pthread_cond_init(&c1,NULL);
```

  - ◆ Beispiel: zählende Semaphore

P-Operation

```
void P(int *sem) {
  pthread_mutex_lock(&m1);
    while ( *sem == 0 )
      pthread_cond_wait
            (&c1, &m1);
    (*sem)--;
  pthread_mutex_unlock(&m1);
}
```

V-Operation

```
void V(int *sem) {
  pthread_mutex_lock(&m1);
    (*sem)++;
    pthread_cond_broadcast(&c1);
  pthread_mutex_unlock(&m1);
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Pthreads-Koordinierung (8)

… Condition Variables (4)

- Bei **`pthread_cond_signal`** wird mindestens einer der wartenden Threads aufgeweckt — es ist allerdings nicht definiert welcher
  - ➤ evtl. Prioritätsverletzung wenn nicht der höchstpriore gewählt wird
  - ➤ Verklemmungsgefahr wenn die Threads unterschiedliche Wartebedingungen haben
- Mit **`pthread_cond_broadcast`** werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

J.11 Threads und Koordinierung in Java

- Thread-Konzept und Koordinierungsmechanismen sind in Java integriert
- Erzeugung von Threads über Thread-Klassen
  - ➤ Beispiel

```
class MyClass implements Runnable {
       public void run()  {
              System.out.println("Hello\n");
       }
}

....
MyClass o1 = new MyClass();    // create object
Thread t1 = new Thread(o1);    // create thread to run in o1

t1.start();                    // start thread

Thread t2 = new Thread(o1);    // create second thread to run in o1

t2.start();                    // start second thread
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

J.11 Threads und Koordinierung in Java (2)

- ★ **Koordinierungsmechanismen**
- Monitore: exklusive Ausführung von Methoden eines Objekts
  - ➤ es darf nur jeweils ein Thread die **`synchronized`**-Methoden betreten
  - ➤ Beispiel:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

  - ◆ Conditions: gezieltes Freigeben des Monitors und Warten auf ein Ereignis

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.