Systemnahe Programmierung in C (SPiC)

20 Nebenläufige Prozesse

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2019

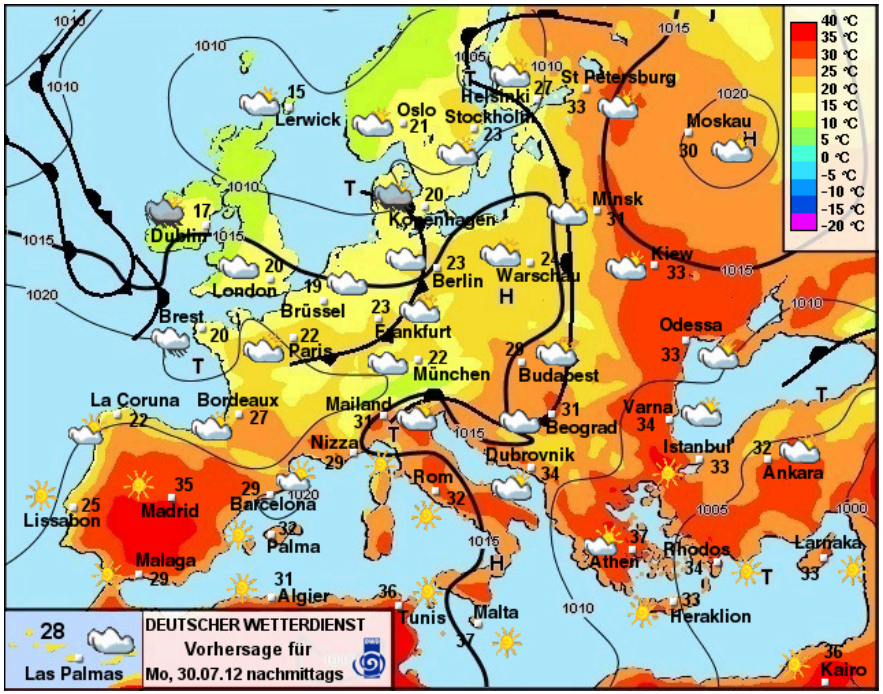
http://www4.cs.fau.de/Lehre/SS19/V_SPIC

Aktivitätsträger (Fäden / Threads)

- Mehrere Prozesse zur Strukturierung von Problemlösungen
  Aufgaben einer Anwendung leichter modellierbar, wenn sie in mehrere kooperierende Prozesse unterteilt wird
  - z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
  - z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Web-Browser)
  - z.B. Client-Server-Anwendungen;
    pro Anfrage wird ein neuer Prozess gestartet (Web-Server)
- Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
  - früher nur bei Hochleistungsrechnern (Aerodynamik, Wettervorhersage)
  - durch Multicore-Systeme jetzt massive Verbreitung

Beispiel: Berechnung einer Wetterkarte

- Berechnung der Wetterkarte muss so schnell wie möglich erfolgen

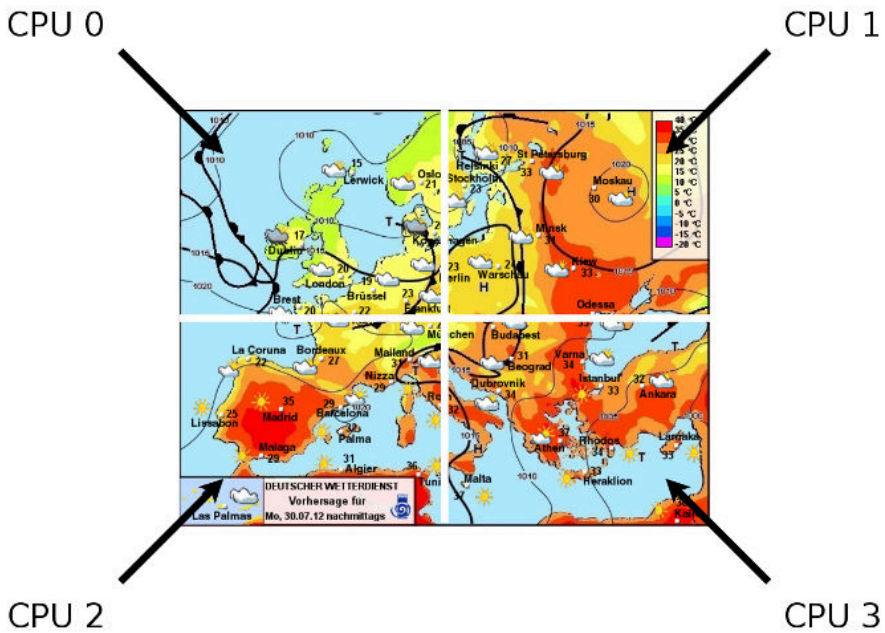


Quelle: www.wetterdienst.de

- Ansatz: Mehrere Prozessoren berechnen jeweils einen Teil der Karte

Beispiel: Berechnung einer Wetterkarte (2)

- Z.B. Berechnung der Wetterkarte aufgeteilt auf 4 Prozessoren:



- Alle Prozessoren greifen auf einen gemeinsamen Speicher zu, in dem das Ergebnis berechnet wird.

Prozesse mit gemeinsamen Speicher

- Nutzung von gemeinsamen Speicher durch mehrere Prozesse

```
char *ptr = mmap(NULL, NBYTES, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (ptr == MAP_FAILED) ... // Fehler

for (i = 0; i < NPROCESSES; i++) {
    pid[i] = fork();
    switch (pid[i]) {
    case -1: ...              // Fehler
    case 0:
        do_work(i, ptr);
        _exit(0);
    default:;
    }
}
for (i = 0; i < NPROCESSES; i++) {
    ret = waitpid(pid[i], NULL, 0);
    if (ret < 0) ...          // Fehler
}

ret = munmap(ptr, NBYTES);
if (ret < 0) ...              // Fehler
```

Beispiel: Vektorlänge

- Berechnung der Länge/Norm eines N-Elemente-Vektors mit einem Prozess:

```
#include <math.h>

double
veclen(double vec[])
{
    double sum = 0.0;

    for (int i = 0; i < N; i++) {
        sum += vec[i] * vec[i];
    }

    return sqrt(sum);
}
```


Beispiel: Vektorlänge (2)

- Berechnung der Länge eines N-Elemente-Vektors mit vier Prozessen:

```
double veclen(double vec[]) {
    pid_t pid[4];
    double *ptr = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                       MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    for (int p = 0; p < 4; p++) {
        if ((pid[p] = fork()) == 0) {
            double sum = 0.0;
            for (int i = p * N / 4; i < (p + 1) * N / 4; i++)
                sum += vec[i] * vec[i];
            ptr[p] = sum;
            _exit(0);
        }
    }
    for (int p = 0; p < 4; p++)
        waitpid(pid[p], NULL, 0);
    double sum = 0.0;
    for (int p = 0; p < 4; p++)
        sum += ptr[p];
    munmap(ptr, 4096);
    return sqrt(sum);
}
```

Beispiel: Vektorlänge (3)

- Hinweis: Beispiel unvollständig
  - `#includes` fehlen
  - Fehlerbehandlung fehlt
  - ...
- Trotzdem sieht man
  - Programmierung sehr viel aufwändiger
  - Programm sehr viel unübersichtlicher
  - eigentlicher Algorithmus kaum noch erkennbar
- Ergebnis ernüchternd
  - Aufwand lohnt sich bei aktuellen Rechnern erst ab etwa $N = 100000$

Prozesse mit gemeinsamen Speicher (2)

- Vorteil der obigen Lösung: in Multiprozessorsystemen sind echt parallele Abläufe möglich; aber
- jeder Prozess hat eigene Betriebsmittel
  - Speicherabbildung
  - Rechte
  - offene Dateien
  - Wurzel- und aktuelles Verzeichnis
  - ...
- => Prozess-Erzeugung, Prozess-Terminierung und Prozess-Umschaltungen sind teuer

Fäden in einem Prozess

- Lösung: mehrere Aktivitätsträger in einem Prozess
  - Fäden
  - Threads
  - Light-weight Processes (LWP)
  - ...
- jeder Faden repräsentiert einen eigenen aktiven Ablauf
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack (für lokale Variablen)
- eine Gruppe von Fäden nutzt gemeinsam eine Menge von Betriebsmitteln (gemeinsame Ausführungsumgebung)
  - Speicherabbildung
  - Rechte
  - offene Dateien
  - Wurzel- und aktuelles Verzeichnis
  - ...

Fäden in einem Prozess (2)

- Das Konzept eines Prozesses wird aufgespalten in eine **Ausführungsumgebung** und ein oder mehrere **Aktivitätsträger**
- Ein klassischer UNIX-Prozess ist ein Aktivitätsträger in einer Ausführungsumgebung

Fäden in einem Prozess (3)

- Erzeugen/Terminieren eines Fadens in einem Prozess erheblich billiger als das Erzeugen/Terminieren eines Prozesses (keine eigenen Betriebsmittel)
- Umschalten zwischen Fäden innerhalb eines Prozesses erheblich billiger als das Umschalten zwischen Prozessen
  - es müssen nur die Register und der Programmzähler gewechselt werden (entspricht in etwa einem Funktionsaufruf)
  - Speicherabbildung muss nicht gewechselt werden (Cache-Inhalte bleiben gültig!)
- Implementierung von Fäden auf
  - Anwender-/Bibliotheks-Ebene (User-Level Threads)
  - Betriebssystem-Kern-Ebene (Kernel-Level Threads)

User-Level Threads

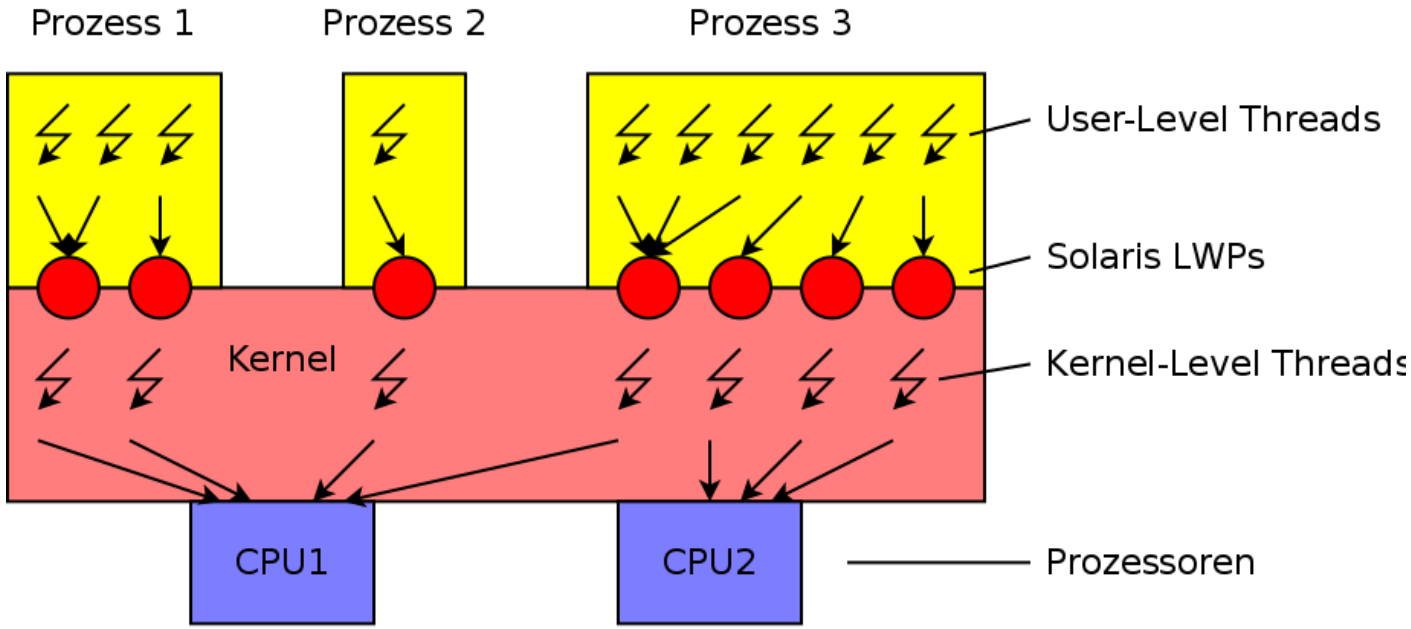
- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Fäden hin und her (ähnlich wie Scheduler im Betriebssystem)
  - Realisierung durch Bibliotheksfunktionen
  - Betriebssystem sieht nur einen Faden
- Vorteile
  - keine Systemaufrufe zum Umschalten erforderlich
  - effiziente Umschaltung (einige wenige Maschinenbefehle)
  - Scheduling-Strategie in der Hand des Anwendungsprogrammierers
- Nachteile
  - bei blockierenden Systemaufrufen bleibt die ganze Anwendung (und damit alle User-Level Threads) stehen
  - kein Ausnutzen eines Multiprozessors möglich

Kernel-Level Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level Threads
  - Betriebssystem schaltet zwischen Threads um
- Vorteile
  - kein Blockieren unbeteiligter Fäden bei blockierenden Systemaufrufen
  - Betriebssystem kann mehrere Fäden einer Anwendung gleichzeitig auf verschiedenen Prozessoren laufen lassen
- Nachteile
  - weniger effizientes Umschalten zwischen Fäden (Systemaufruf notwendig)
  - Scheduling-Strategie meist durch Betriebssystem vorgegeben

Mischform: LWPs und Threads (Bsp. Solaris)

- Solaris kennt Kernel-, User-Level Threads und LWPs



Nach: Silberschatz, 1994

=> wenige Kernel-Level Threads um Parallelität zu erreichen, viele User-Level Threads, um die unabhängigen Abläufe in der Anwendung zu strukturieren

Koordinierung / Synchronisierung

- Fäden arbeiten nebenläufig/parallel und haben gemeinsamen Speicher

  => alle von Unterbrechungen und Signalen bekannten Probleme beim Zugriff auf gemeinsame Daten treten auch bei Fäden auf
- Unterschied zwischen Fäden und Interrupt-Service-Routinen bzw. Signal-Handler-Funktionen:
  - „Haupt-Faden" der Anwendung und eine ISR bzw. ein Signal-Handler sind nicht gleichberechtigt
    - ISR bzw. Signal-Handler unterbricht den Haupt-Faden aber ISR bzw. Signal-Handler werden nicht unterbrochen
  - zwei Fäden sind gleichberechtigt
    - ein Faden kann jederzeit zugunsten eines anderen unterbrochen werden (Scheduler) oder parallel zu einem anderen laufen (MPS)

=> Unterbrechungen sperren oder Signale blockieren hilft nicht!

Koordinierung / Synchronisierung (2)

- Grundlegende Probleme
  - gegenseitiger Ausschluss (**Koordinierung**)
    Beispiel:
    Ein Faden möchte einen Datensatz lesen und verhindern, dass ein anderer Faden ihn während dessen verändert.
  - gegenseitiges Warten (**Synchronisierung**)
    Beispiel:
    Ein Faden wartet auf andere Fäden, die jeweils Teilergebnisse berechnen sollen, die dann zusammengefasst werden.

Koordinierung / Synchronisierung (3)

- Komplexere Koordinierungs-/Synchronisierungsprobleme (Beispiele)
  - **Bounded Buffer**:
    Fäden schreiben Daten in Pufferspeicher, andere entnehmen Daten; kritische Situationen:
    - Zugriff auf den Puffer
    - Puffer leer
    - Puffer voll
  - **Philosophenproblem**:
    ein Faden reserviert sich zuerst den Zugriff auf Datenbereich 1, dann auf Datenbereich 2; ein anderer Faden umgekehrt;
    Problem:
    - kann zu Verklemmungen führen

Gegenseitiger Ausschluss (Mutual Exclusion)

- Einfache Implementierung durch **mutex**-Variablen

```
volatile int m = 0; /* 0: free; 1: locked */
volatile int counter = 0;
```

```
...         /* Thread 1 */
lock(&m);
counter++;
unlock(&m);
...
```

```
...         /* Thread 2 */
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
...
```

Nur der Faden, der `lock` aufgerufen hat, darf `unlock` aufrufen!

- Realisierung (nur konzeptionell!)

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

```
void unlock(volatile int *m) {
    *m = 0;
}
```

`lock` (und ggf. `unlock`) müssen **atomar** ausgeführt werden!

Zählende Semaphore

- Ein `Semaphor` (griech. Zeichenträger) ist eine Datenstruktur mit zwei Operationen (nach *Dijkstra*):
  - `P-Operation` (*proberen; passeren; wait; down*)

```
void P(volatile int *s)
{
    while (*s <= 0) {
        /* Wait/sleep... */
    }
    *s -= 1;
}
```

  - `V-Operation` (*verhogen; vrijgeven; signal; up*)

```
void V(volatile int *s)
{
    *s += 1;
    /* Wakeup... */
}
```

`P` und `V` müssen **atomar** ausgeführt werden!

`P` und `V` müssen nicht vom selben Faden aufgerufen werden.

Zählende Semaphore (2)

- Semaphor-Beispiel:

```
volatile int barrier = 0;
int result;
```

```
...         /* Thread 1 */
result = f1();
V(&barrier);
...
```

```
...         /* Thread 2 */
P(&barrier);
f2(result);
...
```

- Faden 1 läuft immer ungehindert durch.
- Faden 2 blockiert an `P`, falls Faden 1 die V-Operation noch nicht ausgeführt hat (und wartet auf die V-Operation) – andernfalls läuft Faden 2 auch durch.

Spin Lock vs. Sleeping Lock

- **Spin Lock**
  - aktives Warten, bis Mutex-Variable frei ($= 0$) wird
  - entspricht konzeptionell einem Pollen
  - Faden bleibt im Zustand „laufend”

  Problem: wenn nur ein Prozessor verfügbar ist, wird Rechenzeit vergeudet, bis durch den Scheduler eine Umschaltung erfolgt
  - nur ein anderer, laufender Faden kann den Mutex freigeben
- **Sleeping Lock**
  - passives Warten
  - Faden geht in den Zustand „blockiert”
  - im Rahmen von `unlock` wird der blockierte Faden in den Zustand „bereit” zurückgeführt

  Problem: bei sehr kurzen kritischen Abschnitten ist der Aufwand für das Blockieren/Aufwecken und die Umschaltung unverhältnismäßig teuer

Implementierung Spin Lock

- zentrales Problem: Atomarität von mutex-Abfrage und -Setzen

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

kritischer Abschnitt

- Lösung: spezielle Maschinenbefehle, die atomar eine Abfrage und eine Modifikation einer Hauptspeicherzelle ermöglichen
  - *Test-and-Set*, *Compare-and-Swap*, *Load-Link/Store-Conditional*, ...

Implementierung Spin Lock (2)

- Beispiel: Implementierung mit *Compare-and-Swap*-Befehl

```
void lock(volatile int *m)
{
    while (compare_and_swap(m, 0, 1) != 0) {
        /* Wait... */
    }
}
```

mit

```
int compare_and_swap(volatile int *ptr, int oldval, int newval)
{
    int res = *ptr;
    if (*ptr == oldval) {
        *ptr = newval;
    }
    return res;
}
```

Diese Funktion existiert z.B. bei x86-Prozessoren als ein atomar ablaufender Maschinenbefehl (`cmpxchg`).

Implementierung Sleeping Lock

- zwei Probleme:

1. Konflikt mit einer zweiten `lock`-Operation:
   Atomarität von mutex-Abfrage und -Setzen

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

kritischer Abschnitt 1

2. Konflikt mit einem `unlock`: *lost-wakeup*-Problem

```
void lock(volatile int *m) {
    while (*m == 1) {
        sleep();
    }
    *m = 1;
}
```

kritischer Abschnitt 2

- Ursachen:

1. Prozessumschaltung während der `lock`-Operation
2. Echt-parallel laufende `lock`- und/oder `unlock`-Operationen

Implementierung Sleeping Lock (2)

- Behebung von Ursache (1):
  Prozessumschaltungen verhindern
  - Prozessumschaltung ist eine Funktion des BS-Kerns
    - erfolgt im Rahmen eines BS-Aufrufs (z.B. `exit`)
    - oder im Rahmen einer Unterbrechungs-Behandlung
      (z.B. Zeitscheiben-Unterbrechung)

  => `lock`/`unlock` werden ebenfalls im BS-Kern implementiert; BS-Kern mit Unterbrechungs-Sperre

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    *m = 0;
    wakeup_waiting_threads();
    sei();
    leave_OS();
}
```


Implementierung Sleeping Lock (3)

- Behebung von Ursache (2):
  Parallele Ausführung auf anderem Prozessor verhindern

```
void lock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    while (*m == 1) {
        block_thread_and_schedule();
    }
    *m = 1;
    spin_unlock();
    sei();
    leave_OS();
}
```

```
void unlock(volatile int *m)
{
    enter_OS();
    cli();
    spin_lock();
    *m = 0;
    wakeup_waiting_threads();
    spin_unlock();
    sei();
    leave_OS();
}
```


Beispiel: POSIX Threads (pthread)

- Programmierschnittstelle standardisiert: **pthread-Bibliothek** (IEEE-POSIX-Standard P1003.4a)
- pthread-Schnittstelle (Basisfunktionen):

  `pthread_create:` Faden erzeugen
  `pthread_exit:` Faden beendet sich selbst
  `pthread_join:` auf Ende eines Fadens warten
  `pthread_self:` eigene Faden-ID abfragen
  `pthread_yield:` Faden gibt CPU freiwillig auf
- Funktionen in pthread-Bibliothek zusammengefasst

```
gcc ... -pthread ...
```


pthread-Schnittstelle

- Faden-Erzeugung

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *param);
```

- Parameter

  `tid`: Zeiger auf Variable, in der die Faden-ID abgelegt werden soll.

  `attr`: Zeiger auf Attribute (z.B. Stack-Größe) des Fadens. `NULL` für Standard-Attribute.

  `func`, `param`: Der neu erzeugte Faden führt die Funktion `func` mit dem Parameter `param` aus.

- Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode (ähnlich `errno`) zurückgeliefert.

pthread-Schnittstelle

- Faden beenden (bei `return` aus `func` oder):

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Der Faden wird beendet und `retval` wird als Rückgabewert zurückgeliefert (siehe `pthread_join`.

- Auf Faden warten und `pthread_exit`-Status abfragen:

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **retvalp);
```

Wartet auf den Faden mit der Faden-ID `tid` und liefert dessen Rückgabewert über `retvalp` zurück.
Als Rückgabewert wird 0 geliefert. Im Fehlerfall wird ein Fehlercode (ähnlich `errno`) zurückgeliefert.

pthread-Beispiel

- Beispiel (Multiplikation Matrix mit Vektor; $\vec{c} = A\vec{b}$):

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

static void *mult(void *ci) {
    int i = (double *) ci - c;

    double sum = 0.0;
    for (int j = 0; j < 100; j++) {
        sum += a[i][j] * b[j];
    }
    c[i] = sum;
    return NULL;
}

int main(void) {
    pthread_t tid[100];

    for (int i = 0; i < 100; i++) {
        pthread_create(&tid[i], NULL, mult, &c[i]);
    }
    for (int i = 0; i < 100; i++) {
        pthread_join(tid[i], NULL);
    }
}
```

pthread-Koordinierung und -Synchronisierung

- Koordinierung durch Mutex-Variablen
  - Erzeugung von Mutex-Variablen

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

  - `lock`-Operation

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *m);
```

  - `unlock`-Operation

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *m);
```


pthread-Beispiel

- Mutex-Beispiel:

```
volatile int counter = 0;
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

```
...      /* Thread 1 */
pthread_mutex_lock(&m);
counter++;
pthread_mutex_unlock(&m);
...
```

```
...      /* Thread 2 */
pthread_mutex_lock(&m);
printf("counter = %d\n", counter);
counter = 0;
pthread_mutex_unlock(&m);
...
```

pthread-Koordinierung und -Synchronisierung (2)

- Synchronisierung durch Bedingungs-Variablen (Condition Variable)
  - auf eine Bedingung kann gewartet werden (sleep)
  - eine Bedingung kann signalisiert werden (wakeup)
  - Erzeugung einer Condition-Variablen

```
pthread_cond_t c;
pthread_cond_init(&c, NULL);
```

  - auf eine Bedingung warten

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

  - eine Bedingung signalisieren

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *c);
int pthread_cond_broadcast(pthread_cond_t *c);
```

`pthread_cond_signal` weckt *einen* Faden, `pthread_cond_broadcast` weckt *alle* auf die Bedingung wartenden Fäden auf

pthread-Beispiel (2)

- Beispiel: zählende Semaphore

```
pthread_mutex_t m;
pthread_cond_t c;

pthread_mutex_init(&m, NULL);
pthread_cond_init(&c, NULL);
```

```
void P(volatile int *s) {
    pthread_mutex_lock(&m);
    while (*s == 0) {
        pthread_cond_wait(&c, &m);
    }
    *s -= 1;
    pthread_mutex_unlock(&m);
}
```

```
void V(volatile int *s) {
    pthread_mutex_lock(&m);
    *s += 1;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```

Fäden, Koordinierung, Synchronisierung in Java

- Faden-Konzept, Koordinierung und Synchronisierung in Java integriert
- Erzeugung von Fäden über die Thread-Klasse; Beispiel:

```
class MyClass implements Runnable {
    public void run() {
        System.out.println("Hello!");
    }
}
...
MyClass o = new MyClass();    // create object
Thread t1 = new Thread(o);    // create thread to run in o
t1.start();                   // start thread
Thread t2 = new Thread(o);    // create second thread
t2.start();                   // start second thread
```


Fäden, Koordinierung, Synchronisierung in Java (2)

- Koordinierung und Synchronisierung über jedes beliebige Objekt
  - Koordinierung über `synchronized`-Blöcke

```
synchronized(obj) {
    ...
}
```

    Ein solcher Block ruft zu Block-Beginn ein `lock` auf das Objekt `obj` auf, führt die angegebenen Anweisungen aus, und ruft vor dem Verlassen des Blockes das entsprechende `unlock` auf.
  - Synchronisierung über `wait`, `notify` und `notifyAll`

    `obj.wait()`: wartet auf die Signalisierung einer Bedingung auf dem angegebenen Objekt `obj`.

    `obj.notify()`: signalisiert eine Bedingung auf dem angegebenen Objekt `obj` an *einen* wartenden Faden.

    `obj.notifyAll()`: signalisiert eine Bedingung auf dem angegebenen Objekt `obj` *allen* wartenden Fäden.

Fäden, Koordinierung, Synchronisierung in Java (3)

- Beispiel Koordinierung und Synchronisierung:

```
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public void P() {
        synchronized(this) {
            while (s == 0)
                this.wait();
            s--;
        }
    }
    public void V() {
        synchronized(this) {
            s++;
            this.notifyAll();
        }
    }
}
```

Entspricht dem pthread-Beispiel...

Fäden, Koordinierung, Synchronisierung in Java (4)

- Vereinfachte Schreibweise (entspricht „Monitor"-Konzept):

```
public class Semaphore {
    private int s;

    public Semaphore(int s0) {
        s = s0;
    }
    public synchronized void P() {
        while (s == 0) {
            wait();
        }
        s--;
    }
    public synchronized void V() {
        s++;
        notifyAll();
    }
}
```