

# Übungen zu Betriebssysteme

## Ü9 – Threads, Synchronisierung & Aufgabe: palim

---

Sommersemester 2023

Henriette Hofmeier, Manuel Vögele, Benedict Herzog, Timo Hönig

Bochum Operating Systems and System Software Group (BOSS)



RUHR  
UNIVERSITÄT  
BOCHUM

RUB

# Agenda

9.1 Threads

9.2 Koordinierung

9.3 Aufgabe: palim

9.4 Gelerntes anwenden

# Agenda

9.1 Threads

9.2 Koordinierung

9.3 Aufgabe: palim

9.4 Gelerntes anwenden

- UNIX-Prozesskonzept (vollständige Ausführungsumgebung mit einem Aktivitätsträger) für viele heutige Anwendungen unzureichend
  - keine parallelen Abläufe innerhalb eines logischen Adressraums auf Multiprozessorsystemen
  - typische UNIX-Server-Implementierungen benutzen die fork-Operation, um einen Server-Prozess für jeden Client zu erzeugen
    - langsam (Erzeugung & Prozesswechsel)
    - ressourcenintensiv
  - zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adressraums nützlich

- UNIX-Prozesskonzept (vollständige Ausführungsumgebung mit einem Aktivitätsträger) für viele heutige Anwendungen unzureichend
  - keine parallelen Abläufe innerhalb eines logischen Adressraums auf Multiprozessorsystemen
  - typische UNIX-Server-Implementierungen benutzen die fork-Operation, um einen Server-Prozess für jeden Client zu erzeugen
    - langsam (Erzeugung & Prozesswechsel)
    - ressourcenintensiv
  - zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adressraums nützlich
- Lösung: Weitere Aktivitätsträger in einem UNIX-Prozess erzeugen

## Federgewichtige Prozesse (User-Threads)

- Realisierung auf Anwendungsebene
- Systemkern sieht nur **einen** Kontrollfluss
- + Erzeugung von Threads extrem billig
- Systemkern hat kein Wissen über diese Threads
  - in Multiprozessorsystemen keine parallelen Abläufe möglich
  - wird **ein** User-Thread blockiert, sind **alle** User-Threads blockiert
  - Scheduling zwischen den Threads schwierig

## Federgewichtige Prozesse (User-Threads)

- Realisierung auf Anwendungsebene
- Systemkern sieht nur **einen** Kontrollfluss
- + Erzeugung von Threads extrem billig
- Systemkern hat kein Wissen über diese Threads
  - in Multiprozessorsystemen keine parallelen Abläufe möglich
  - wird **ein** User-Thread blockiert, sind **alle** User-Threads blockiert
  - Scheduling zwischen den Threads schwierig

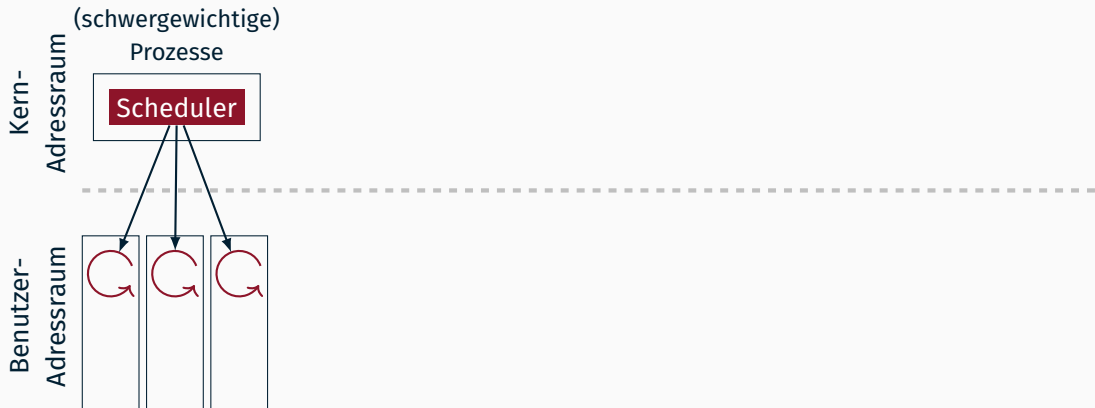
## Leichtgewichtige Prozesse (Kernel-Threads)

- + Gruppe von Threads nutzt gemeinsam die Betriebsmittel eines Prozesses
- + jeder Thread ist als eigener Aktivitätsträger dem Betriebssystemkern bekannt
- Kosten für Erzeugung erheblich geringer als bei Prozessen, aber erheblich teurer als bei User-Threads

# Arten von Threads

## Umschaltungskosten ("Gewichtsklasse")

Kosten für Umschaltung zwischen Threads hängt maßgeblich von der Anzahl der Adressraumwechsel ab.

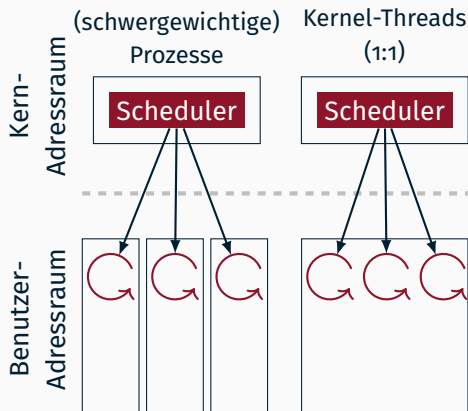




# Arten von Threads

## Umschaltungskosten ("Gewichtsklasse")

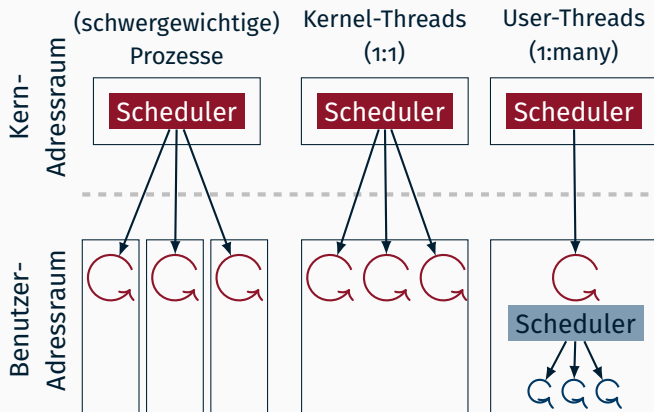
Kosten für Umschaltung zwischen Threads hängt maßgeblich von der Anzahl der Adressraumwechsel ab.



# Arten von Threads

## Umschaltungskosten ("Gewichtsklasse")

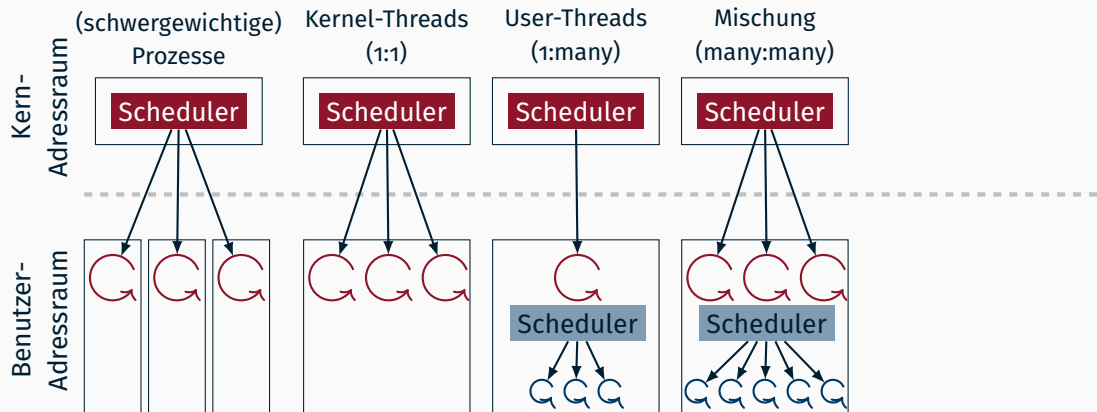
Kosten für Umschaltung zwischen Threads hängt maßgeblich von der Anzahl der Adressraumwechsel ab.



# Arten von Threads

## Umschaltungskosten ("Gewichtsklasse")

Kosten für Umschaltung zwischen Threads hängt maßgeblich von der Anzahl der Adressraumwechsel ab.



## ■ POSIX-Thread erzeugen

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

- thread Thread-ID (Ausgabeparameter)
- attr Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.
- Nach der Erzeugung führt der Thread die Funktion `start_routine` mit Parameter `arg` aus
- Im Fehlerfall wird **errno** nicht gesetzt, aber ein Fehlercode als Ergebnis zurückgeliefert.
  - Um `perror(3)` verwenden zu können, muss der Rückgabewert erst in der **errno** gespeichert werden.

## ■ POSIX-Thread erzeugen

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

- `thread` Thread-ID (Ausgabeparameter)
- `attr` Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). `NULL` für Standardattribute.
- Nach der Erzeugung führt der Thread die Funktion `start_routine` mit Parameter `arg` aus
- Im Fehlerfall wird **`errno` nicht gesetzt**, aber ein Fehlercode als Ergebnis zurückgeliefert.
  - Um `perror(3)` verwenden zu können, muss der Rückgabewert erst in der **`errno`** gespeichert werden.

## ■ Eigene Thread-ID ermitteln

```
pthread_t pthread_self(void);
```

- Die Funktion kann nie fehlschlagen.

- Thread beenden (bei Rücksprung aus `start_routine` oder):  
`void pthread_exit(void *retval);`
  - Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join(3)`)

- Thread beenden (bei Rücksprung aus `start_routine` oder):  
`void pthread_exit(void *retval);`
  - Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join(3)`)
- Auf Thread warten, Ressourcen freigeben und Rückgabewert 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.

- Thread beenden (bei Rücksprung aus `start_routine` oder):  
`void pthread_exit(void *retval);`
  - Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join(3)`)
- Auf Thread warten, Ressourcen freigeben und Rückgabewert 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.
- Ressourcen automatisch bei Beendigung freigeben:  
`int pthread_detach(pthread_t thread);`
  - Die mit dem Thread `thread` verbundenen Systemressourcen werden bei dessen Beendigung automatisch freigegeben. Der Rückgabewert der Thread-Funktion kann nicht abgefragt werden.



# Beispiel: Matrix-Vektor-Multiplikation

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

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

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

- Casts zwischen `int` und Zeiger bei `pthread_create()` problematisch
  - **nicht zu Hause nachmachen!**
  - C-Standard garantiert nicht, dass `int` verlustfrei in Zeiger umgewandelt werden können
  - z.B.: `sizeof(int) != sizeof(void *)`

# Parameterübergabe bei `pthread_create()`

- Generischer Ansatz mit Hilfe einer Struktur für die Argumente

```
struct param {  
    int index;  
};
```

# Parameterübergabe bei pthread\_create()

- Generischer Ansatz mit Hilfe einer Struktur für die Argumente

```
struct param {  
    int index;  
};
```

- Für jeden Thread eine eigene Argumenten-Struktur anlegen
  - Speicher je nach Situation auf dem Heap oder dem Stack allozieren

```
int main(int argc, char *argv[]) {  
    pthread_t tids[100];  
    struct param args[100];  
  
    for(int i = 0; i < 100; i++) {  
        args[i].index = i;  
        pthread_create(&tids[i], NULL, mult, &args[i]);  
    }  
    for(int i = 0; i < 100; i++)  
        pthread_join(tids[i], NULL);  
    ...  
}
```

## Parameterübergabe bei pthread\_create()

```
static void *mult(void *arg) {
    struct param *par = arg;

    double sum = 0;
    for(int j = 0; j < 100; j++) {
        sum += a[par->index][j] * b[j];
    }
    c[par->index] = sum;
    return NULL;
}
```

- Zugriff auf den threadspezifischen Parametersatz über  
(gecasteten) Parameter (`void *arg` → `struct param *par`)

# pthread\_detach()

```
static void *thread(void *x) {
    errno = pthread_detach(pthread_self());
    if (errno) {
        // ...
    }
    sleep(10); // seconds
    return NULL;
}

int main(void) {
    pthread_t tid;
    errno = pthread_create(&tid, NULL, thread, NULL); // test.c:15
    if (errno) {
        // ...
    }
}
```

# pthread\_detach()

```
static void *thread(void *x) {
    errno = pthread_detach(pthread_self());
    if (errno) {
        // ...
    }
    sleep(10); // seconds
    return NULL;
}

int main(void) {
    pthread_t tid;
    errno = pthread_create(&tid, NULL, thread, NULL); // test.c:15
    if (errno) {
        // ...
    }
}
```

==16891== 288 bytes in 1 blocks are possibly lost in loss record 1 of 1  
[...]

==16891== by 0x4A75B95: pthread\_create (pthread\_create.c:669)

==16891== by 0x1090B1: main (test.c:15)

# pthread\_detach()

```
static void *thread(void *x) {
    errno = pthread_detach(pthread_self());
    if (errno) {
        // ...
    }
    sleep(10); // seconds
    return NULL;
}

int main(void) {
    pthread_t tid;
    errno = pthread_create(&tid, NULL, thread, NULL); // test.c:15
    if (errno) {
        // ...
    }
}
```

```
==16891== 288 bytes in 1 blocks are possibly lost in loss record 1 of 1
[...]
```

```
==16891==    by 0x4A75B95: pthread_create (pthread_create.c:669)
```

```
==16891==    by 0x1090B1: main (test.c:15)
```

- Wettlaufsituation zwischen Thread- und main-Beendigung
- Nicht vermeidbar  $\Rightarrow$  kann ignoriert werden

# Agenda

9.1 Threads

**9.2 Koordinierung**

9.3 Aufgabe: palim

9.4 Gelerntes anwenden



# Koordinierung – Motivation

```
static double a[100][100], sum;

int main(int argc, char *argv[]) {
    pthread_t tids[100];
    struct param args[100];

    for(int i = 0; i < 100; i++) {
        args[i].index = i;
        pthread_create(&tids[i], NULL, sumRow, &args[i]);
    }
    for(int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
}

static void *sumRow(void *arg) {
    struct param *par = arg;
    double localSum = 0;
    for(int j = 0; j < 100; j++)
        localSum += a[par->index][j];
    sum += localSum;
    return NULL;
}
```

Was macht das Programm?  
Welches Problem kann  
auftreten?

# Koordinierung – Motivation

```
static double a[100][100], sum;

int main(int argc, char *argv[]) {
    pthread_t tids[100];
    struct param args[100];

    for(int i = 0; i < 100; i++) {
        args[i].index = i;
        pthread_create(&tids[i], NULL, sumRow, &args[i]);
    }
    for(int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
}

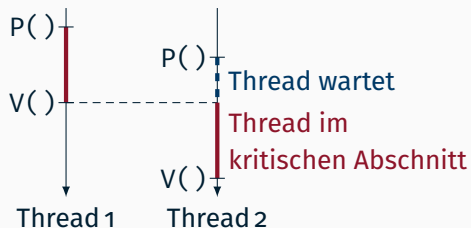
static void *sumRow(void *arg) {
    struct param *par = arg;
    double localSum = 0;
    for(int j = 0; j < 100; j++)
        localSum += a[par->index][j];
    sum += localSum;
    return NULL;
}
```

Was macht das Programm?  
Welches Problem kann  
auftreten?

- Zur Koordinierung von Threads können Semaphore verwendet werden
- 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
- Stattdessen Verwendung einer eigenen Semaphorimplementierung mit atomaren  $P()$ - und  $V()$ -Operationen
  - Datenstruktur mit (atomarer) Zählervariable
  - $P()$  dekrementiert Zähler und blockiert Aufrufer, falls Zähler  $\leq 0$
  - $V()$  inkrementiert Zähler und weckt ggf. wartende Threads

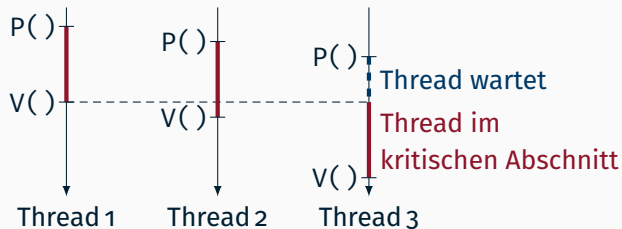
# Gegenseitiger Ausschluss

- Spezialfall des zählenden Semaphors: Binärer Semaphor
  - Initialisierung des Semaphors mit 1
- Beispiel: Schreibender Zugriff auf ein gemeinsames Datum

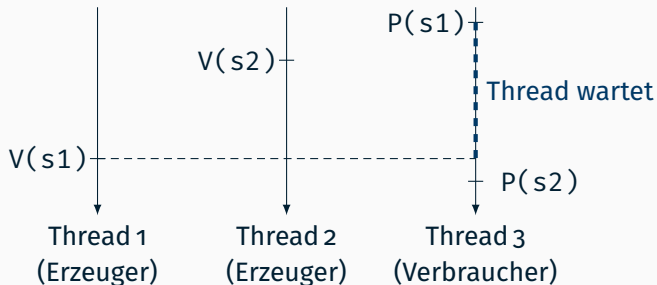


# Limitierung von Ressourcen

- Verwendung eines zählenden Semaphors
- Beispiel: Nur zwei aktive Threads gleichzeitig gewünscht
  - Initialisierung des Semaphors mit 2



- Benachrichtigung eines anderen Threads über ein Ereignis
- Beispiel: Bereitstellen von Zwischenergebnissen
  - Initialisierung des Semaphors mit 0



- Semaphor erzeugen

```
SEM *semCreate(int initVal);
```

- P/V-Operationen

```
void P(SEM *sem);
```

```
void V(SEM *sem);
```

- Semaphor zerstören

```
void semDestroy(SEM *sem);
```

- Semaphor-Modul und zugehörige Headerdatei befinden sich in der .zip-Datei.

- Semaphor-Funktionen bekannt machen mit `#include "sem.h"`

# Agenda

9.1 Threads

9.2 Koordinierung

**9.3 Aufgabe: palim**

9.4 Gelerntes anwenden



- Lernziele
  - Benutzen der Dateischnittstelle
  - Nebenläufige Programmierung
  - Synchronisation
- Mehrfädige, rekursive Suche nach einer Zeichenkette in mehreren Verzeichnisbäumen
- Aufteilung in drei Arten von Threads:
  - Hauptthread: Initialisierung, Ausgabe, Deinitialisierung
  - crawl-Threads: Durchsuchen der Verzeichnisbäume
  - grep-Threads: Durchsuchen der Dateien nach Zeichenkette

## ■ Hauptthread

- initialisiert benötigte Datenstrukturen
- Startet einen crawl-Thread pro übergebenen Verzeichnisbaum
- gibt anschließend nach jeder Werteänderung Statistiken aus
- wartet passiv auf Statistikänderungen
- gibt nach dem Ende aller crawl-/grep-Threads allokierte Ressource frei

## ■ crawl-Thread

- durchsucht rekursiv den übergebenen Verzeichnisbaum
- startet pro Datei einen grep-Thread
- aktualisiert ggf. Statistiken

## ■ grep-Thread

- durchsucht eine Datei nach dem gesuchten String
- aktualisiert ggf. Statistiken

# Agenda

9.1 Threads

9.2 Koordinierung

9.3 Aufgabe: palim

**9.4 Gelerntes anwenden**

## „Aufgabenstellung“

Thread-Beispiel zur Berechnung der Zeilensummen einer Matrix mit Hilfe eines Semaphors korrekt synchronisieren.