# Systemnahe Programmierung in C

## 31 Nebenläufige Fäden

**J. Kleinöder, D. Lohmann, V. Sieh**

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2025

http://sys.cs.fau.de/lehre/ss25



# Multiprozessor-Systeme

In Multiprozessorsystemen sind echt-parallele Abläufe möglich

**aber**

Prozess-Erzeugung, Prozess-Terminierung und
Prozess-Umschaltungen sind teuer

Für die **Praxis** heißt das:

- nur selten Prozesse erzeugen/terminieren
- nicht mehr Prozesse erzeugen als echte Prozessoren vorhanden

**oder**

statt teuren Prozessen einfache **Fäden (Threads)** verwenden



# Fäden in einem Prozess

Lösung: **mehrere** Aktivitätsträger (Fäden, Threads) in **einer** Ausführungsumgebung

- jeder **Faden (Thread)** repräsentiert einen eigenen aktiven Ablauf
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack (für lokale Variablen)
- gemeinsame **Ausführungsumgebung** stellt Menge von Betriebsmitteln zur Verfügung
  - 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!)



# 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 währenddessen verändert.
  - gegenseitiges Warten (**Synchronisierung**)
    Beispiel:
    Ein Faden wartet auf andere Fäden, die jeweils Teilergebnisse berechnen sollen, die dann zusammengefasst werden.



# Koordinierung / Synchronisierung (3)

- Komplexe Koordinierungs-/Synchronisierungsprobleme (Beispiel)
  - **Bounded Buffer**:
    Fäden schreiben Daten in Pufferspeicher, andere entnehmen Daten; kritische Situationen:
    - Zugriff auf den Puffer
    - Puffer leer / voll

**Element einfügen:**

- Warten, bis Platz im Puffer
- Warten, bis kein anderer Faden mehr den Puffer liest/schreibt
- Puffer beschreiben
- Signalisieren, dass neues Element im Puffer

**Element herausnehmen:**

- Warten, bis Element im Puffer
- Warten, bis kein anderer Faden mehr den Puffer liest/schreibt
- Puffer auslesen
- Signalisieren, dass Platz im Puffer



# 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.



# Bounded Buffer (2)

Bounded-Integer-Buffer-Beispiel:

```
#define N 1000
volatile int mutex = 0;
volatile int alloc = 0, free = N;
volatile int head = 0, tail = 0;
volatile int buf[N];
```

Element einfügen:

```
void put(int x) {

    P(&free);
    lock(&mutex);
    buf[head] = x;
    head = (head + 1) % N;
    unlock(&mutex);
    V(&alloc);

}
```

Element herausnehmen:

```
int get(void) {
    int x;
    P(&alloc);
    lock(&mutex);
    x = buf[tail];
    tail = (tail + 1) % N;
    unlock(&mutex);
    V(&free);
    return x;
}
```



# 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 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();
}
```

- `P()` und `V()` ähnlich