

Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier
(Lehrstuhl Informatik 4)

Übung 10



Sommersemester 2016



Inhalt

- Threads
 - Motivation
 - Threads vs. Prozesse
 - Kritische Abschnitte
 - Speedup & Amdahl's Gesetz
 - POSIX Threads

Hands-ons

Inhalt

Threads

- Motivation
- Threads vs. Prozesse
- Kritische Abschnitte
- Speedup & Amdahl's Gesetz
- POSIX Threads

Hands-ons

- Stoppuhr
- Threads

Motivation

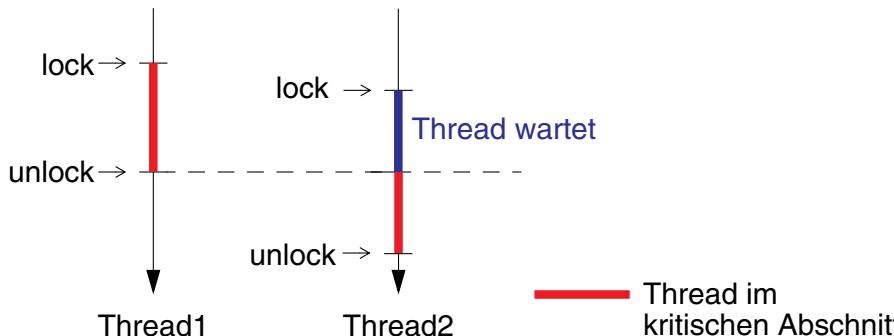
- Strukturierung von Problemlösungen in mehrere Kontrollflüsse
 - Client, Server
 - Benutzeroberfläche, Hintergrundaufgaben
 - Zerlegung in Subsysteme mit eigenen Kontrollflüssen
 - ein Kontrollfluss pro Anfrage
 - ...
- Performancesteigerung
 - echte Parallelität zur Leistungssteigerung nutzen
 - Anzahl parallel behandelbarer Anfragen pro Sekunde steigern
 - Parallelisierbarkeit von Algorithmen ausnutzen
 - Auslastung moderner Multi- & Manycore CPUs

- Prozesse
 - geschützte Ausführungsumgebung für Programme
 - eigener virtueller Addressraum
 - Prozesswechsel und -erzeugung ist aufwändig
- Threads (in Prozessen)
 - gemeinsame Ressourcen (Speicher, geöffnete Dateien)
 - einfaches Teilen von Ressourcen zwischen Threads
 - Nutzung echter Parallelität innerhalb eines Prozesses
 - (relativ) günstiger Fadenwechsel



Schutz kritischer Abschnitte

- Mutual exclusion (wechselseitiger Ausschluss)
- Koordinierung von kritischen Abschnitten:



- Nur ein Thread kann gleichzeitig den Mutex sperren und somit den kritischen Abschnitt durchlaufen



- Asymmetrische Nebenläufigkeit
 - Signal oder Interrupt unterbricht Hauptprogramm
 - Hauptprogramm unterbricht jedoch keine Signale bzw. Interrupts
 - Schutz durch einseitige Synchronisation
 - Interrupts bzw. Signale sperren
- Symmetrische Nebenläufigkeit
 - gleichberechtigte Threads
 - gegenseitige Verdrängung, echte Parallelität
 - Schutz durch mehrseitige Synchronisation
 - wechselseitiger Ausschluss (Mutex)
 - aktives Warten: spin lock
 - passives Warten: sleeping lock



Speedup & Amdahl's Gesetz

- Speedup bei paralleler Ausführung mit N Threads

$$S_N = \frac{T_1}{T_N} \quad (1)$$

- Amdahl's Gesetz

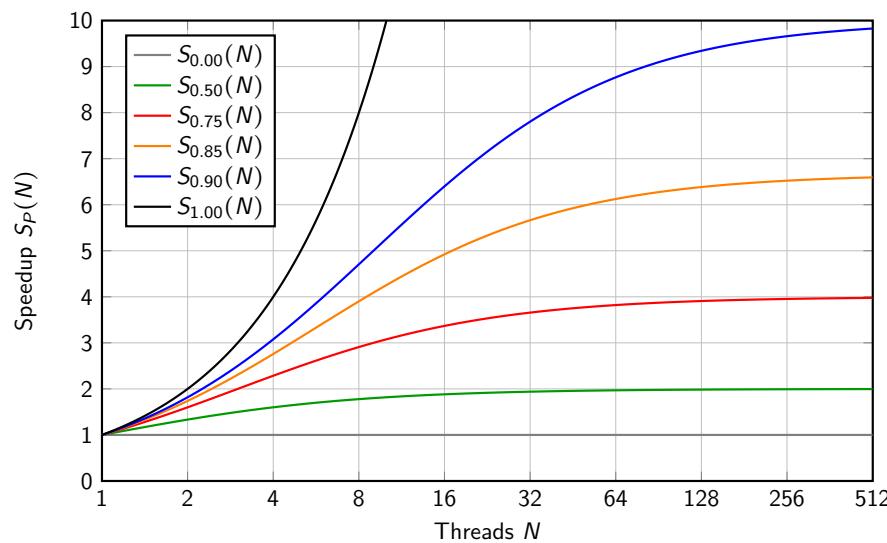
- Serieller Anteil beschränkt maximalen Speedup
- Theoretischer Speedup bei Parallelanteil P und N Threads

$$S_P(N) = \frac{1}{(1-P) + \frac{P}{N}} \quad \text{mit} \quad 0 \leq P \leq 1 \quad (2)$$

- Maximaler theoretischer Speedup bei Parallelanteil P

$$\lim_{N \rightarrow \infty} S_P(N) = \frac{1}{(1-P)} \quad (3)$$





pthread Minimalbeispiel

```

1 void *worker(void *arg){
2     // do useful stuff
3
4     pthread_exit(NULL);
5 }
6
7 int main(int argc, char *argv[]){
8     pthread_t tid;
9     int arg = 5;
10
11     errno = pthread_create(&tid, NULL, worker, &arg);
12     if(errno != 0){
13         perror("pthread_create");
14         exit(EXIT_FAILURE);
15     }
16
17     // do useful stuff
18
19     pthread_join(tid, NULL);
20
21     exit(EXIT_SUCCESS);
22 }
```

Starten eines Threads

```

1 int pthread_create(pthread_t *restrict thread,
2                   const pthread_attr_t *restrict attr,
3                   void *(*start_routine)(void *),
4                   void *restrict arg);
```

thread Zeiger für Thread ID

attr Attribute des Threads (Standard: NULL)

start_routine Auszuführende Routine

arg Argument für Routine

Beenden eines Threads

```

1 void pthread_exit(void *value_ptr);
```

Warten auf Beendigung eines Threads

```

1 int pthread_join(pthread_t thread, void **value_ptr);
```

pthread Mutex

Schnittstelle

- Mutex erzeugen

```

1 pthread_mutex_t m;
2
3 errno = pthread_mutex_init(&m, NULL); // Fehlerbehandlung!
```

- Sperren und Freigeben

```

1 pthread_mutex_lock(&m);
2 // kritischer Abschnitt
3 pthread_mutex_unlock(&m);
```

- Mutex zerstören und Ressourcen freigeben

```

1 errno = pthread_mutex_destroy(&m); // Fehlerbehandlung!
```

■ Alle pthread-Funktionen setzen **errno** nicht implizit, sondern geben einen Fehlercode zurück (im Erfolgsfall: 0)

■ **errno** ist keine globale Variable, sondern eine Thread-lokale Variable
– jeder Thread besitzt seine eigene **errno**

Threads

Hands-ons

Stoppuhr

Threads



Wiederholung Signale

1. Signalhandler installieren: `sigaction(2)`

```
1 struct sigaction act;
2 act.sa_handler = SIG_DFL; // Handlersignatur: void f(int signum)
3 act.sa_flags = SA_RESTART;
4 sigemptyset(&act.sa_mask);
5 sigaction(SIGINT, &act, NULL);
```

2. Signale blockieren/deblockieren: `sigprocmask(2)`

```
1 sigset_t set;
2 sigemptyset(&set);
3 sigaddset(&set, SIGUSR1);
4 sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
5 // kritischer Abschnitt
6 sigprocmask(SIG_UNBLOCK, &set, NULL); /* Deblockiert SIGUSR1 */
```

3. Auf Signale warten: `sigsuspend(2)`

```
1 sigprocmask(SIG_BLOCK, &set, &old); /* Blockiert Signale */
2 while(event == 0){
3     sigsuspend(&old); /* Wartet auf Signale */
4 }
5 sigprocmask(SIG_SETMASK, &old, NULL); /* Deblockiert Signale */
```

Hands-on: Stoppuhr

```
1 $ ./stoppuhr
2 Press CTRL+C to start.
3 ~CStarted...
4 5000 msec
5 10000 msec
6 ~CStopped.
7 Duration: 13020 msec
```

■ Ablauf:

- Stoppuhr startet durch SIGINT Signal
 - gibt alle 5 Sekunden Dauer aus (Format: "5000 msec")
 - Stoppuhr stoppt bei weiterem SIGINT und gibt Dauer aus
 - gibt Gesamtdauer aus (Format: "13020 msec")
 - beendet sich anschließend
- Verwendet intern SIGALRM und alarm(3)
 - Schutz kritischer Abschnitte beachten



Alarne bzw. Timer

■ Timerkonfiguration mit `alarm(2)`

```
1 unsigned alarm(unsigned seconds);
```

Einmaliger Alarm nach definierter Wartezeit (in Sekunden)

`seconds = 0` Alarm abbrechen

■ Timerkonfiguration mit `ualarm(2)`

```
1 useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

Erster Alarm nach `useconds` Mikrosekunden,
anschließend alle `interval` Mikrosekunden

`useconds = 0` Alarm abbrechen

`interval = 0` einmaliger Alarm

■ SIGALRM: Timer ist abgelaufen bzw. Alarm eingetreten

→ Standardbehandlung: Programm beenden

→ Eigenen Signalhandler installieren



- Sequentielles Programm ohne Threads
 - Bestimmt die Anzahl der Primzahlen in einem Array
- Parallelisierung mit Threads
 - Zerlegung in Teilarrays
 - Starten mehrerer Threads
 - Bestimmung der Anzahl Primzahlen in den Teilarrays
 - Warten auf Beendingung aller Threads
 - Ausgabe der Gesamtzahl
- Evaluation
 - Bestimmung des Speedups mit Hilfe von `time(1)`
 - Berechnungsergebnis ok ?
 - Kritische Abschnitte geschützt ?
 - Wechselseitiger Ausschluss
 - Einfluss auf Speedup ?
- Wie funktioniert die Implementierung von `time(1)` ?

