

Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2018

Übung 10

Benedict Herzog
Sebastian Maier

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Threads

- 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

1

Threads vs. Prozesse



- Prozesse
 - geschützte Ausführungsumgebung für Programme
 - eigener virtueller Adressraum
 - 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

2

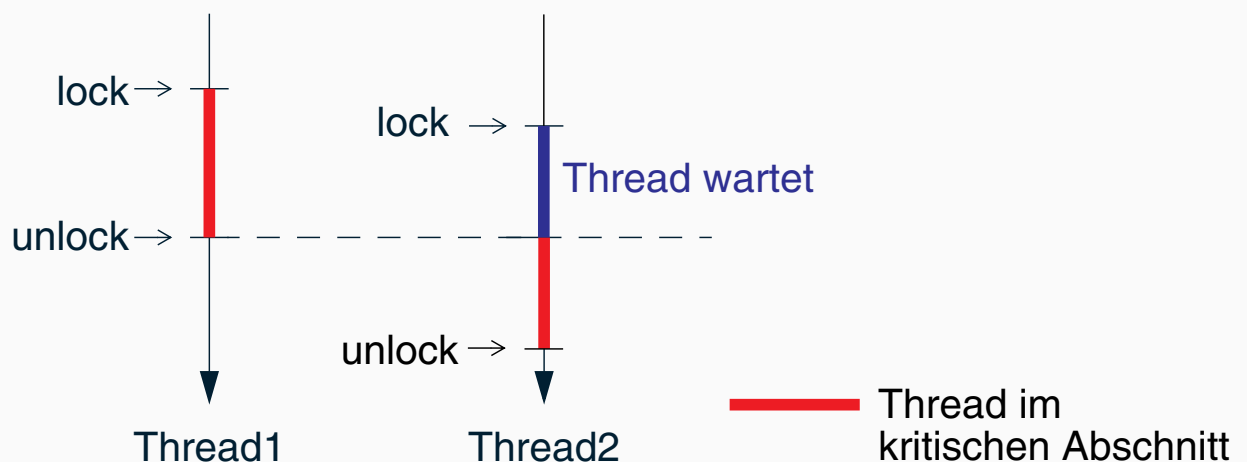
- Asymmetrische Nebenläufigkeit
 - Signal oder Interrupt unterbricht Hauptprogramm
 - Hauptprogramm unterbricht jedoch keine Signale bzw. Interrupts
 - Schutz durch einseitige Synchronisation
 - Interrupts bzw. Signale sperren
- Symmetrische Nebenläufigkeit
 - gleichberechtigte Threads
 - gegenseitige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

3

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

4

- Speedup bei paralleler Ausführung mit N Threads

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

- Amdahl's Gesetz

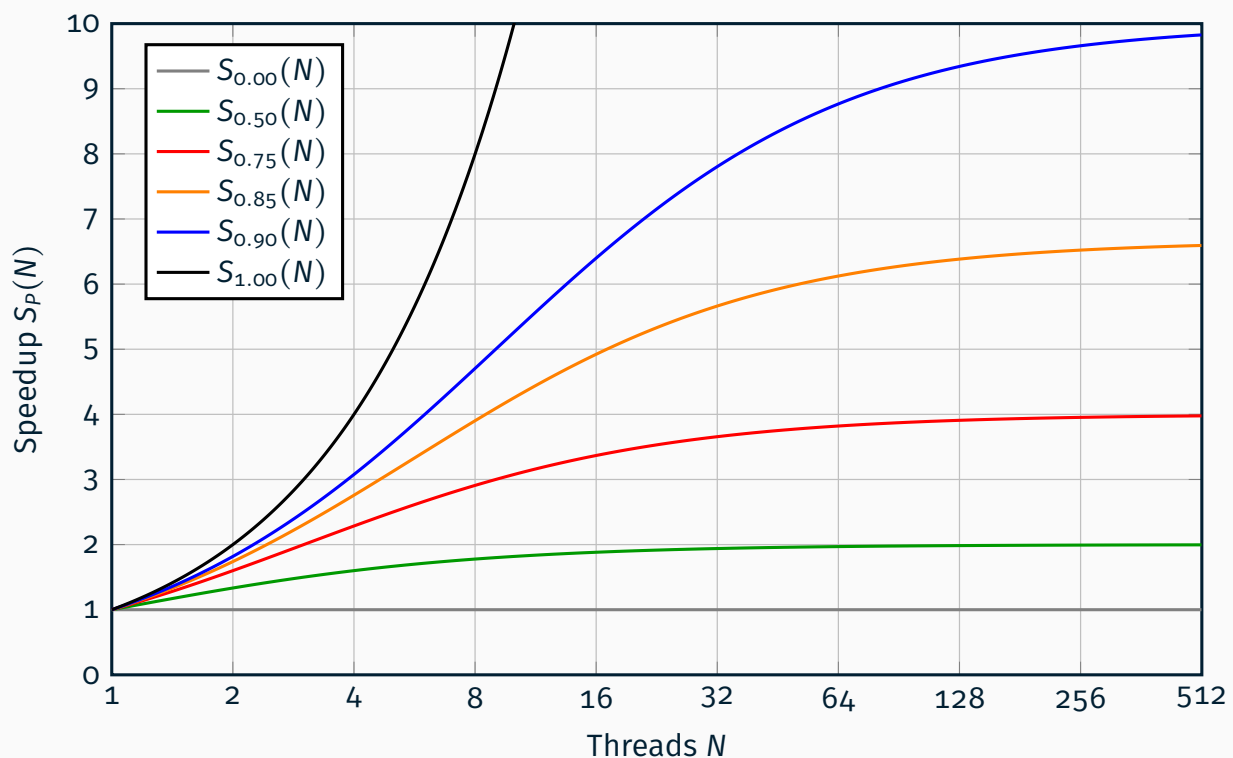
- Serieller Anteil beschränkt maximalen Speedup
- Theoretischer Speedup bei Parallelanteil P und N Threads

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

- Maximaler theoretischer Speedup bei Parallelanteil P

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

5



6



■ Starten eines Threads

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

thread Zeiger für Thread ID

attr Attribute des Threads (Standard: NULL)

start_routine Auszuführende Routine

arg Argument für Routine

■ Beenden eines Threads

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

■ Warten auf Beendigung eines Threads

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

7

pthread Minimalbeispiel



```
01 void *worker(void *arg){  
02     // do useful stuff  
03  
04     pthread_exit(NULL);  
05 }  
06  
07 int main(int argc, char *argv[]){  
08     pthread_t tid;  
09     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 }
```

8

- Schnittstelle

- Mutex erzeugen

```
01 pthread_mutex_t m;  
02  
03 errno = pthread_mutex_init(&m, NULL); // Fehlerbehandlung!
```

- Sperren und Freigeben

```
01 pthread_mutex_lock(&m);  
02 // kritischer Abschnitt  
03 pthread_mutex_unlock(&m);
```

- Mutex zerstören und Ressourcen freigeben

```
01 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`

Hands-on: Stoppuhr



```
01 $ ./stoppuhr
02 Press Ctrl+C (SIGINT) to start and stop
03 ^CStarted...
04 1 sec
05 2 sec
06 3 sec
07 4 sec
08 ^CStopped.
09 Duration: 4 sec 132 msec
```

■ Ablauf:

- Stoppuhr startet durch SIGINT Signal
 - gibt jede Sekunde die bisherige Dauer aus (Format: "3 sec")
- Stoppuhr stoppt bei weiterem SIGINT und gibt Dauer aus
 - gibt Gesamtdauer inkl. Millisekunden aus (Format: "4 sec 132 msec")
 - beendet sich anschließend

- Verwendet intern SIGALRM und alarm(3)
- Schutz kritischer Abschnitte beachten

10

Wiederholung Signale



1. Signalhandler installieren: sigaction(2)

```
01 struct sigaction act;
02 act.sa_handler = SIG_DFL; // Handlersignatur: void f(int signum)
03 act.sa_flags = SA_RESTART;
04 sigemptyset(&act.sa_mask);
05 sigaction(SIGINT, &act, NULL);
```

2. Signale blockieren/deblockieren: sigprocmask(2)

```
01 sigset_t set;
02 sigemptyset(&set);
03 sigaddset(&set, SIGUSR1);
04 sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
05 // kritischer Abschnitt
06 sigprocmask(SIG_UNBLOCK, &set, NULL); /* Deblockiert SIGUSR1 */
```

11

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

```
01 sigprocmask(SIG_BLOCK, &set, &old); /* Blockiert Signale */
02 while(event == 0){
03     sigsuspend(&old); /* Wartet auf Signale */
04 }
05 sigprocmask(SIG_SETMASK, &old, NULL); /* Deblockiert Signale */
```

11

Alarmer bzw. Timer



■ Timerkonfiguration mit `alarm(2)`

```
01 unsigned alarm(unsigned seconds);
```

Einmaliger Alarm nach definierter Wartezeit (in Sekunden)

seconds = 0 Alarm abbrechen

■ Timerkonfiguration mit `ualarm(3)`

```
01 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

12

Hands-on: Threads

Hands-on: Threads



- 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 Beendigung 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 könnte eine Implementierung von `time(1)` aussehen?