Übungen zu Systemnahe Programmierung in C (SPiC) – Wintersemester 2017/18

Übung 9

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

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 behandelbarerer Anfragen pro Sekunde steigern
  - Parallelisierbarkeit von Algorithmen ausnutzen
  - Auslastung moderner Multi- & Manycore CPUs

Threads vs. Prozesse

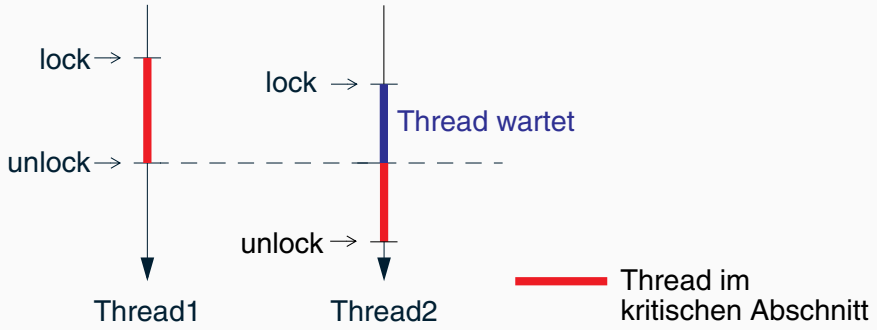
- Prozesse
  - geschützte Ausführungsumgebung für Programme
  - eigener virtueller Adressraum
  - Prozesswechsel und -erzeugung ist aufwändig
- Threads (in Prozessen)
  - gemeinsame Ressourcen (Speicher, geöffnete Dateien)
  - einfaches Teilen von Ressourcen zwischen Threads
  - Nutzung echter Parallelität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

Kritische Abschnitte

- 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

Schutz kritischer Abschnitte

- **Mut**ual **ex**clusion (wechselseitiger Ausschluss)
- Koordinierung von kritischen Abschnitten:



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

Speedup & Amdahl's Gesetz

- Speedup bei paralleler Ausführung mit N Threads

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

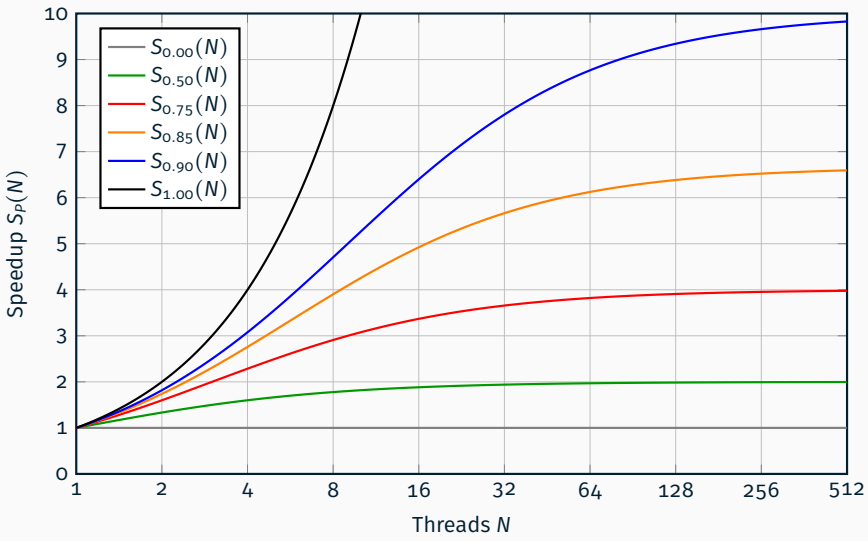
- Amdahl's Gesetz
  - Serieller Anteil beschränkt maximalen Speedup
  - Theoretischer Speedup bei Parallelanteil P und N Threads

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

  - Maximaler theoretischer Speedup bei Parallelanteil P

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

Speedup & Amdahl's Gesetz



POSIX Threads (pthreads)

- Starten eines Threads

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

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

- Warten auf Beendigung eines Threads

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


pthread Minimalbeispiel

```
void *worker(void *arg){
  // do useful stuff

  pthread_exit(NULL);
}

int main(int argc, char *argv[]){
  pthread_t tid;
  int arg = 5;

  errno = pthread_create(&tid, NULL, worker, &arg);
  if(errno != 0){
    perror("pthread_create");
    exit(EXIT_FAILURE);
  }

  // do useful stuff

  pthread_join(tid, NULL);

  exit(EXIT_SUCCESS);
}
```


pthread Mutex

- Schnittstelle
  - Mutex erzeugen

```
  pthread_mutex_t m;

  errno = pthread_mutex_init(&m, NULL); // Fehlerbehandlung!
```

  - Sperren und Freigeben

```
  pthread_mutex_lock(&m);
  // kritischer Abschnitt
  pthread_mutex_unlock(&m);
```

  - Mutex zerstören und Ressourcen freigeben

```
  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

Hands-on: Stoppuhr

```
$ ./stoppuhr
Press Ctrl+C (SIGINT) to start and stop
^CStarted...
1 sec
2 sec
3 sec
4 sec
^CStopped.
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

Wiederholung Signale

1. Signalhandler installieren: `sigaction(2)`

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

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

```c
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
// kritischer Abschnitt
sigprocmask(SIG_UNBLOCK, &set, NULL); /* Deblockiert SIGUSR1 */
```


Wiederholung Signale

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

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


Alarme bzw. Timer

- Timerkonfiguration mit `alarm(2)`

```c
unsigned alarm(unsigned seconds);
```

Einmaliger Alarm nach definierter Wartezeit (in Sekunden)
`seconds = 0` Alarm abbrechen

- Timerkonfiguration mit `ualarm(3)`

```c
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

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 funktioniert die Implementierung von `time(1)` ?