Threads

Effiziente Programmierung in C

Christian Widera

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2021-06-19

Gliederung

Einleitung

- Vorgänge nebeneinander laufen lassen
- scheinbare Nebenläufigkeit (concurrency), Beispiel:
  - Warten auf User-Eingabe
  - Darstellung des Interfaces

  $\rightarrow$ eine CPU reicht, die hin- und herspringt
- echte Nebenläufigkeit (parallelism), Beispiel:
  - unabhängige Operationen auf eine Matrix

  $\rightarrow$ mehrere CPUs rechnen gleichzeitig

Threads

- Prozesse haben mindestens einen Thread
- Gentrennt: Kontrollfluss, Register, Stack
- Geteilt: Code, `static`/globale Variablen, Heap
- (multi-processing: alles getrennt)

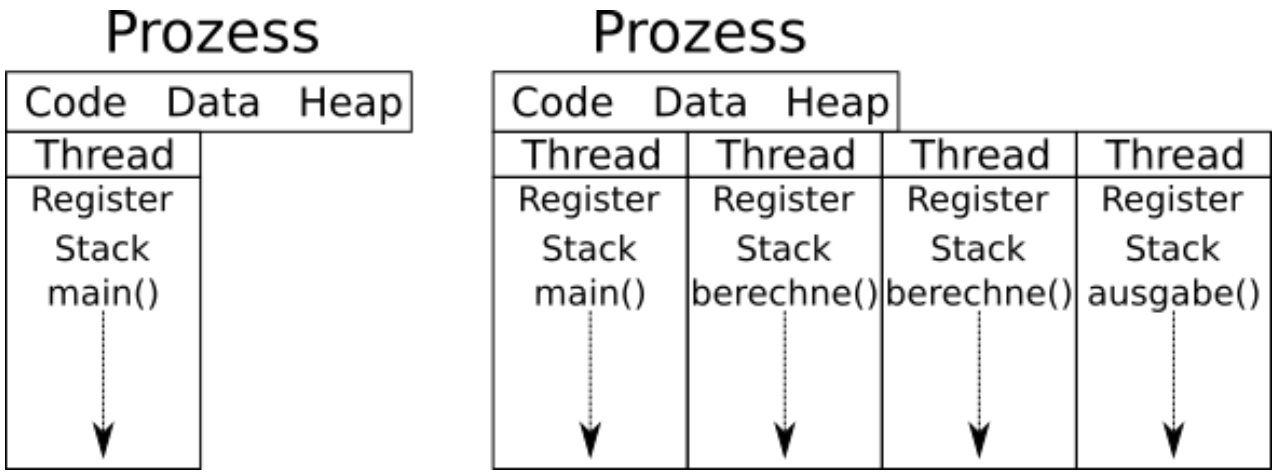


Abbildung: Single- vs. Multithreaded Process

Scheduler

- Scheduler bestimmt, wann ein Thread auf welcher CPU läuft
- verschieden Strategien (z.B. FIFO, Priorisierung . . .)

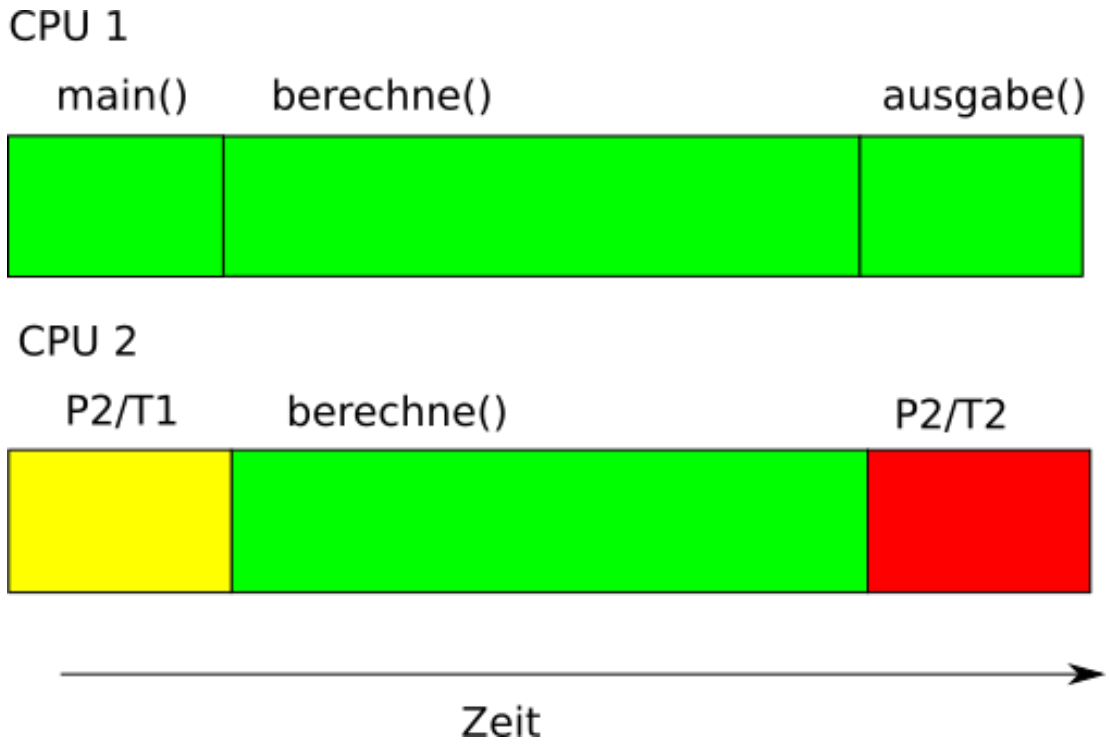


Abbildung: Schedulebeispiel

Kontextwechsel

- OS kümmert sich um Kontextwechsel
- Stack/Register werden gespeichert/wiederhergestellt

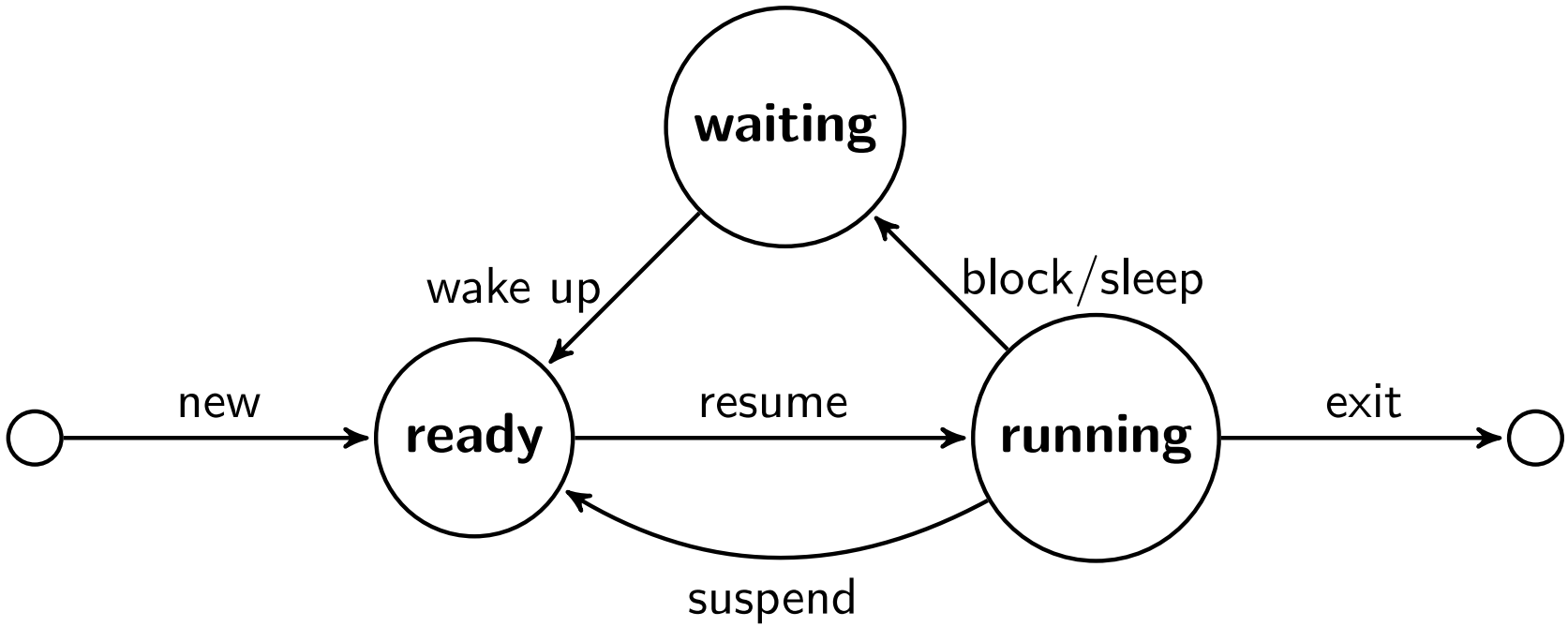


Abbildung: Thread states

pthreads.h vs. C11 threads.h

- POSIX threads, pthreads.h
  - älter, verbreiteter
  - plattformabhängig
- C11 threads.h
  - kleinere/einfachere API
  - erst ab C11-Standard
  - plattformunabhängig

Thread-Funktion

```c
#include <pthread.h>
#include <stdio.h>

void *work(void *argument) {
    // argument zeigt auf die Thread-ID
    printf("Thread %d: Hello!\n", *((int*)argument));
}
```

Listing 1: `work`-Funktion

- Argument: Pointer zum Argument, das der Thread erhalten soll
- Rückgabewert kann von anderen Threads ausgelesen werden

Aufrufende Funktion

```
1  int main() {
2      pthread_t threads[5]; int ids[5];
3
4      for (int i = 0; i < 5; i++) {
5          ids[i] = i;
6          pthread_create(&threads[i], NULL, work,
               ↪ &ids[i]);
7      }
8      printf("Threads gestartet\n");
9  }
```

Listing 2: 5 Threads

- `pthread_create` Argumente:
  - Pointer zu einem `pthread_t`
  - Thread-Attribute (`NULL` für Default)
  - Funktion, die der Thread ausführen soll
  - Pointer zum Argument für die Thread-Funktion

Ergebnis ...?

```
1 $ gcc -o main main.c -l pthread && ./main
2 Thread 0: Hello!
3 Thread 3: Hello!
4 Threads gestartet
```

- Nur zwei Threads wurden gestartet?
- Prozess wurde beendet nachdem der Hauptthread fertig war

Join threads

```
10 printf("Threads gestartet\n");
11 for (int i = 0; i < 5; i++)
12     pthread_join(threads[i], NULL);
13 printf("Threads fertig\n");
```

Listing 3: Thread joining

```
1 Thread 0: Hello!
2 Thread 3: Hello!
3 Threads gestartet
4 Thread 4: Hello!
5 Thread 1: Hello!
6 Thread 2: Hello!
7 Threads fertig
```

Listing 4: Erfolgreiche Ausgabe

Race condition

```
14 int counter = 0;
15 void *work(void *argument) {
16     int next = counter + 1;
17     counter = next;
18 }
19 ...
20 int main() {
21     ...
22     printf("Threads fertig, Ergebnis: %d\n", counter);
23 }
```

Listing 5: einfacher Zähler

- Bei 5 Threads: Ergebnis manchmal 5, aber manchmal 4!

Ressourcenzugriff

- Gleichzeitiger Zugriff auf Resourcen problematisch

Thread 1	counter	Thread 2
liest 0	0	
berechnet 1	0	liest 0
schreibt 1	1	berechnet 1
	1	schreibt 1
	1	

- Ergebnis vom Schedule abhängig
- Schwer zu debuggen

Atomic Instructions

- Laden, berechnen, und speichern ohne Unterbrechung
- ohne fremden Speicherzugriff (memory barrier)
- wenige Grundoperationen
- Abhängig von Compiler, Plattform und Prozessor
- Performant

Atomic Instructions, Beispiel

```
24 int counter = 0;
25 void *work(void *argument) {
26     __sync_fetch_and_add(&counter, 1);
27 }
```

Listing 6: GCC's atomic add

Mutex

- Mutex hat höchstens einen Besitzer (*mutual exclusion*)
- Mutex ist nur sperrbar, wenn es keinen Besitzer hat
- nur Besitzer kann Mutex entsperren
- Sperrversuch bei gesperrten Mutex: Thread-state $\rightarrow$ waiting

Mutex Beispiel

```
28 int counter = 0;
29 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
30
31 void *work(void *argument) {
32     pthread_mutex_lock(&mutex);
33     counter++;
34     pthread_mutex_unlock(&mutex);
35 }
```

Listing 7: Zähler mit Mutex

Mutex Ablauf

Thread 1	counter	Thread 2	Mutex-Besitzer
sperren	0	sperren	Thread 1
liest 0	0	warten	...
berechnet 1	0	...	...
schreibt 1	**1**		Thread 1
entsperren	1		
	1	sperren	Thread 2
	1	liest 1	...
	1	berechnet 2	...
	2	schreibt 2	Thread 2
	2	entsperren	

Semaphore

- Semaphore: Zähler, der nicht unter 0 fällt
- `sem_post` inkrementiert Zähler
- `sem_wait` dekrementiert Zähler oder wartet
- kein Besitzer
- z.B. Ausführung von n Worker-Threads

Beispiel

```
36 #include <semaphore.h>
37
38 sem_t sem;
39
40 void *work(void *argument) {
41     sem_wait();
42     printf("Hallo!\n");
43     sleep(1);
44     sem_post();
45 }
46
47 int main() {
48     sem_init(&sem, 0, 2 /* initialer Zaehler */);
49     ...
50 }
```

Listing 8: Semaphore-Beispiel

Conditional Values

- Thread mit Mutex im Besitz:
  - Mutex entsperren und warten auf CV
  - anderer Thread sendet Signal zu CV
  - Thread sperrt Mutex wieder
- z.B. producer möchte worker über neue Arbeit informieren

Beispiel

```
51 pthread_cond_t condition_value =
       ↪ PTHREAD_COND_INITIALIZER;
52 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
53
54 void *work(void *argument) {
55     pthread_mutex_lock(&mutex);
56     pthread_cond_wait(&condition_value, &mutex);
57     printf("Hallo!\n");
58     pthread_mutex_unlock(&mutex);
59 }
60 int main() {
61     ...
62     sleep(1);
63     pthread_cond_broadcast(&condition_value);
64     ...
65 }
```

Listing 9: Conditional Value-Beispiel

Deadlock

```
66 int counter = 0;
67 pthread_mutex_t mutex;
68
69 void *work(void *argument) {
70     pthread_mutex_lock(&mutex);
71     counter++;
72     // pthread_mutex_unlock(&mutex);
73 }
```

Listing 10: Deadlock-Beispiel

- andere Threads bleiben im Status waiting stecken
- A besitzt M_1, wartet auf M_2,
  B besitzt M_2, wartet auf M_1

static

```
74 int get_next_id() {
75     static int counter;
76     return counter++;
77 }
```

Listing 11: Deadlock-Beispiel

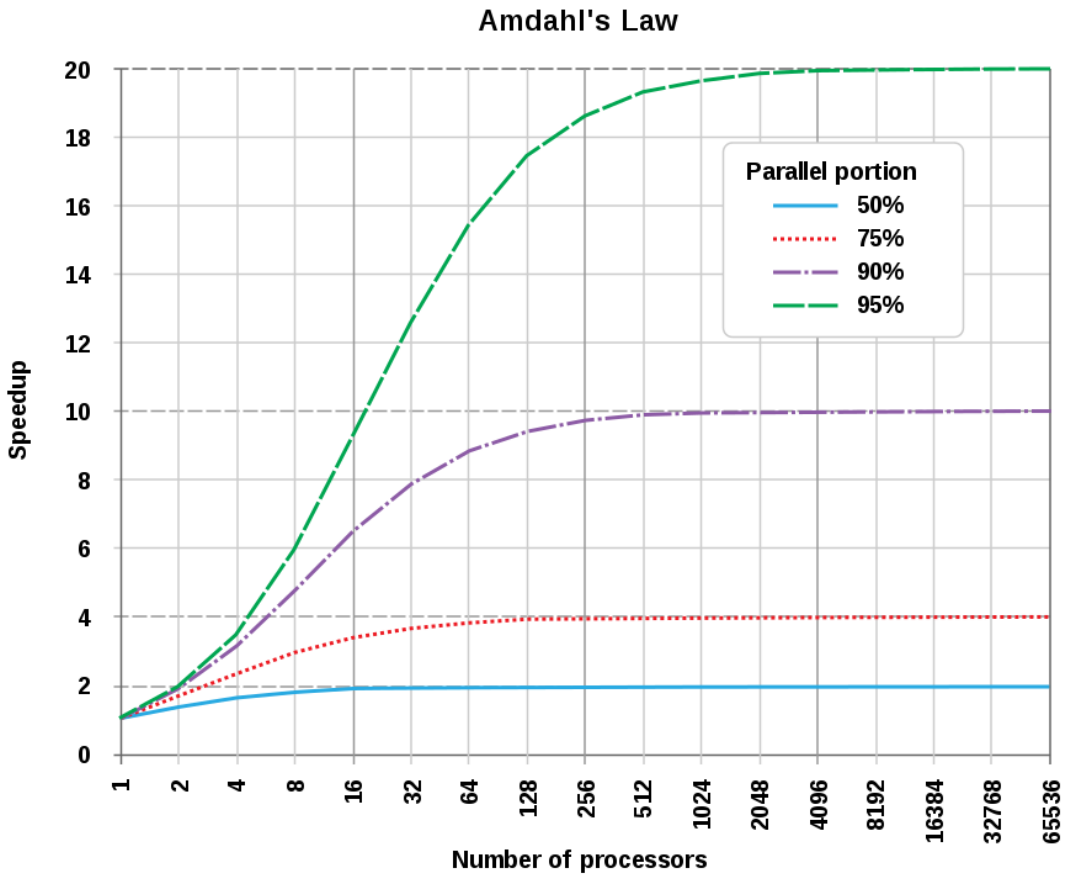
- Thread-Safety nicht durch Funktionsdefinition erkennbar
- z.B. `char *strtok(char *str, char *delimiter)` aus `string.h`
- GCCs `static __thread int counter;`

Speedup

- Speedup $S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1((1-f)+\frac{f}{p})}$
- T_p Laufzeit mit p Prozessoren
- f Anteil von T_1, der parallel ausgeführt werden kann
- $T_1 = 10s$, $f = 0.5$, $p = 4$:
  $S_p = \frac{10s}{10s(0.5+0.125)} = 1.6$

Amdahlsches Gesetz

- Maximaler Speedup $S_{max} = \frac{T_1}{T_1(1-f)}$



[3]

Zusammenfassung

- parallele Ausführung in einem Prozess
- schwer mit zu arbeiten
- Performancegewinn ist begrenzt
- . . . aber wenn f groß genug ist, macht es einen großen Unterschied

Literatur

[1] man pthreads.7
https://man7.org/linux/man-pages/man7/pthreads.7.html

[2] C11 threads.h *ISO/IEC 9899:2011*

[3] Daniels220 at English Wikipedia (CC BY-SA 3.0)
https://en.wikipedia.org/wiki/File:AmdahlsLaw.svg