Campus Sontheim

HOCHSCHULE HEILBRONN

SEMAPHOREN UND NEBENLÄUFIGKEIT

Betriebssysteme | Fakultät IT / SEB | WiSe 20/21

Prof. Dr.-Ing. Andreas Heil

HOCHSCHULE HEILBRONN

› TEIL 1: SEMAPHOREN

OSTEP: Kapitel 31 - Semaphores

HOCHSCHULE HEILBRONN

LERNZIELE UND KOMPETENZEN

Konzept hinter Semaphoren verstehen und grundlegende Anwendungsmöglichkeiten **kennen lernen**

Typische Probleme und Fehlerursachen bei Nebenläufigkeit verstehen und mögliche Lösungsansätze **kennen lernen** und **verstehen**

SEMAPHOREN

Zunächst technisch betrachtet:

Eine Semaphore ist ein Objekt mit einem Integer-Wert und dazugehörigen Methoden[1)]

- *sem_wait()*
- *sem_post()*

Der initiale Wert der Semaphore bestimmt ihr Verhalten, weswegen sie initialisiert werden muss…

[1)] Wir betrachten, wie schon zuvor, den POSIX Standard

SEMPAHORE INITIALISIEREN

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

- Zunächst wird eine Semaphore s deklariert
- s wird an *sem_init()* übergeben und mit dem Wert 1 initialisiert
- Der zweite Parameter 0 gibt an, dass die Semaphore zwischen Threads im selben Prozess geteilt wird
- Semaphoren dienen also dazu, Threads zu synchronisieren
- Semphoren können auch prozessübergreifend zur Synchronisation genutzt werden, wird von uns aber nicht weiter behandelt

SEM_WAIT UND SEM_POST

```
int sem_wait(sem_t *s) {
  verringere den Wert der Semaphore s um 1
  warte falls der Wert der Semaphore s negative ist
}

int sem_post(sem_t *s) {
  erhöhe den Wert der Semaphore s um 1
  falls ein oder mehr Threads warten, wecke einen davon auf
}
```

SEM_WAIT

- *sem_wait()* kehrt sofort zurück, falls der Wert größer oder gleich 0 war
- Oder, der Aufrufer wartet auf die Rückkehr aus der Routine, wenn der Wert negativ war
- Ein ähnliches Konzept mit dem Warten auf das Lock habe wir bei den Locks verwende, sie erinnern sich? Wir kommen darauf gleich nochmal zurück… Bitte
- Wird *sem_wait()* von mehreren Threads aufgerufen und der Wert war oder wird negativ, müssen alle aufrufenden Threads warten

SEM_POST

- *sem_post()* erhöht einfach den Wert um 1
- **Ohne weitere Prüfung** wird dann ein schlafender Thread aufgeweckt

Wie kann das funktionieren?! Das schauen wir auf den folgenden Beispielen an!

- Interessantes Teil 1: Der negative Wert einer Semaphore gibt die Nummer der wartenden Threads an. Das sollte uns die Grundidee der Semaphore geben.
- Interessantes Teil 2: Der Initialisierungswert einer Semaphore ist die Anzahl an Ressourcen, die man initial „hergeben“ möchte (oder kann)
- Beides werden wir im Laufe der kommenden Beispiele besser bestehen lernen

BINÄRE SEMAPHORE (1)

Verwendung der **Semaphore als Lock** (das Prinzip ist uns bekannt, wenn nicht sollten Sie Einheit 6 nochmals wiederholen und die Kapitel 28 und 28 aus OSTEP wiederholen)

> Wir setzen den Wert der **Semaphore initial auf 1**

> Beispiel 1:

- Thread 0 ruft *sem_wait()* auf
- Wert der Sempahore wird um 1 verringert
- Thread wartet nur, wenn der Wert nicht größer gleich 0 ist (vgl. Folie *sem_wait und sem_post*)
- Da der Wert 0 ist, kehr *sem_wait()* sofort zurück und Thread 0 kann den kritischen Abschnitt ausführen
- Sofern kein anderer Prozess versucht hat das Lock zu erhalten, ruft Thread 0 am Ende *sem_post()* auf und der Wert der Semaphore wird zurück auf 1 gesetzt

BINÄRE SEMAPHORE (2)

> Beispiel 2:
  - Thread 0 hat *sem_post()* noch nicht aufgerufen und Thread 1 versucht das Lock zu erhalten
  - Thread 1 verringert den Wert der Semaphore auf -1 und wartet
  - Wenn Thread 0 irgendwann ausgeführt wird und fertig ist, wird durch den Aufruf von *sem_post()* der Wert der Semaphore auf 0 erhöht und ein anderer Thread (Thread 1) aufgeweckt
  - Thread 1 kann das Lock jetzt erhalte (der Wert ist ja auf 0) und ruft nach Beendigung *sem_post()* auf und setzt somit den Wert der Semaphore wieder auf 1

> Ein Trace dieses Beispiels finden Sie in Kapitel 31.2 in OSTEP. „Berechnen“ Sie ein eigenes Beispiel mit mehr als 1 wartenden Thread entsprechend dem Beispiel im Buch.

WARUM BINÄRE SEMAPHORE?

Mit dem vorherigen Beispiel haben wir eine Semaphore genutzt, um ein Lock zu erzeugen

- Locks haben genau zwei Zustände: gesperrt oder nicht gesperrt bzw. engl. *held* und *not held*
- Aufgrund der zwei Zustände heißt eine solche Semaphore **binäre Semaphore**

SEMAPHORE ZUR SIGNALISIERUNG

Semaphoren können auch genutzt werden, um Threads zu synchronisieren

> Prinzip: ein Thread wartet darauf, dass etwas passiert, ein anderer kümmert sich darum, dass etwas passiert und signalisiert somit etwas dem anderen Thread

> Etwas konkreter: Ein Thread erzeugt einen anderen Thread und wartet auf dessen Beendigung. Wie funktioniert das im Detail

HOCHSCHULE HEILBRONN

PARENT – CHILD SYNCHRONIAATION (1)

> Parent Thread erzeugt den Child Thread
> In diesem Fall initialisieren wir den Wert der Semaphore mit 0
> Parent Thread ruft *sem_wait()* auf, Child Thread ruft *sem_post()* auf

> Szenario 1: Child Thread wurde zwar erzeugt, läuft aber noch nicht
  - Parent Thread ruf *sem_wait()* auf bevor der Child Thread *sem_post()* aufruft
  - Parent läuft , ruft *sem_wait(),* verringert den Wert der Semaphore auf -1 und schläft dann
  - Erst wenn der Child Thread beendet ist und *sem_post()* aufruft wird der Wert auf 0 gesetzt und der Parent Thread kann wieder weiter laufen

PARENT – CHILD SYNCHRONISATION (2)

> Szenario 2: Child Thread wurde erzeugt und läuft bis zum Ende, bevor Parent Thread *sem_wait()* aufrufen konnte
  - Child Thread ruft *sem_post()* auf
  - Der Wert der Semaphore wird von 0 auf 1 erhöht (Parent Thread hatte ja noch keine Chance *sem_wait()* aufzurufen, und der Wert wurde noch nicht verringert)
  - Wenn nun der Parent Thread wieder läuft und *sem_wait()* aufruft, wird der Wert der Semaphore zurück auf 0 gesetzt – ohne zu warten, da die Routine sofort beendet

PRODUCER/CONSUMER PROBLEM

Producer/Consumer Problem (auch Bounded Buffer Problem)

> Bereits bei Conditional Variables behandelt – falls Sie sich nicht (mehr) erinnern, wiederholen Sie Kapitel 27/28 aus OSTEP

> Klassisches Problem in der Prozesssynchronsiation, die den konkurrierenden Zugriff von Erzeugern (engl. producer) und Verbrauchern (engl. consumer) regelt
> Grundidee: Consumer sollen keine Elemente aus einer Datenstruktur nehmen können, wenn dort keine enthalten sind und Producer sollen keine Elemente in die Datenstruktur schreiben können, wenn dort die maximale Kapazität erreicht ist

HOCHSCHULE HEILBRONN

DER PUFFER

- Zunächst gehen wir davon aus, es gibt genau einen Puffer, d.h. *MAX = 1*
- Zwei Threads, Producer, Consumer und eine CPU
- Weiterhin gehen wir davon aus, dass der Consumer zuerst läuft

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value; // Line F1
  fill = (fill + 1) % MAX; // Line F2
}

int get() {
  int tmp = buffer[use]; // Line G1
  use = (use + 1) % MAX; // Line G2
  return tmp;
}
```

CONSUMER

> Zwei Semaphore
  - empty
  - full

```
sem_t empty;
sem_t full;
```

> Consumer wartet bis der Puffer voll ist

```
void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);  // Line C1
    tmp = get();      // Line C2
    sem_post(&empty); // Line C3
    printf("%d\n", tmp);
  }
}
```

PRODUCER

> Zwei Semaphore
  - empty
  - full

```
sem_t empty;
sem_t full;
```

> Producer wartet, bis Puffer leer ist

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty); // Line P1
    put(i); // Line P2
    sem_post(&full); // Line P3
  }
}
```

VERSUCH 1

- Consumer startet und ruft *sem_wait(&full)* auf
- Da *full* mit 0 initialisiert war, wird *full* um 1 reduziert (d.h. -1)
- Consumer wird nun blockiert und wartet bis ein andere Thread *sem_post(&full)* aufruft
- Nun läuft der Producer und ruft *sem_wait(&empty)* auf
- *empty* wurde mit *MAX* (=1) initialisiert, daher läuft der Producer zunächst und *empty* wird auf 0 reduziert

```
int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0); // 0 are full
  // ...
}
```

- Nun wird der Puffer befüllt und *sem_post(&full)* aufgerufen
- *full* wird von -1 auf 0 erhöht und der Consumer kann wieder laufen

VERSUCH 1: ERGEBNIS

Szenario 1:

> Producer läuft noch weiter, versucht wieder *sem_wait(&empty)* aufzurufen und wird diesmal blockiert, *empty* steht nämlich derzeit auf 0

Szenario 2:

> Producer stoppt, Consumer startet, würde nun von *sem_wait(&full)* zurückkehre und get() aufrufen
> Da der Producer *&full* inzwischen erhöht hatte, kann der Consumer wieder laufen

→ Beide Fälle führen zum gewünschten Ergebnis!

WAS PASSIERT WENN…

… MAX > 1 wäre?

- Annahme: Mehrere Producer und mehrere Consumer
- In unserem Beispiel gibt es eine Race Condition
- Sowohl in *put* als auch in *get* können zwei Threads auf den gleichen Zähler (*fill* bzw. *use*) zugreifen, wenn Sie durch ein Interrupt direkt nach dem Schreiben, aber vor dem Erhöhen unterbrochen werden

- Was fehlt? Mutual Exclusion oder eben eine Lock

MUTEX

> Nutzen wir eine binäre Semaphore als Lock für den kritischen Abschnitt

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++)
    sem_wait(&empty); // Line P1
    sem_wait(&mutex); // Line P1.5 (MUTEX)
    put(i); // Line P2
    sem_post(&mutex); // Line P2.5 (AND)
    sem_post(&full);  // Line P3
  }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&full);  // Line C1
    sem_wait(&mutex); // Line C1.5 (MUTEX)
    int tmp = get();  // Line C2
    sem_post(&mutex); // Line C2.5 (MUTEX)
    sem_post(&empty); // Line C3
    printf("%d\n", tmp);
  }
}
```

> Hinweis: Werden die Zeilen des Mutex mit der jeweiligen anderen Semaphore paarweise vertauscht (z.B. Pi mit P1.5) entsteht ein Deadlock

WEITERE BEISPIELE

Weitere Beispiele für die Anwendung von Semaphoren in OSTEP Kapitel 31.5 und 31.6

REFERENZEN

[1] “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics” by Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. ASPLOS ’08, March 2008, Seattle, Washington

[2] The Design and Implementation of the FreeBSD Operating System, Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, 2014, ISBN 978-0321968975