Betriebssysteme 1

Foliensatz F, Kap. 5 (Teil 2)

Prof. Dr. Hans-Georg Eßer

Sommersemester 2022

v3.1 – 18.05.2022

Selbstkontrollaufgabe 21: Semaphore mit negativen Zählern

Semaphore (klassisch)

```
wait (sem) {
  if (sem > 0)
    sem--;
  else {
    enqueue (T, QUEUE(sem));
    sleep();
  }
}
```

```
signal (sem) {
  if (T in QUEUE(sem)) {
    wakeup (T);
    remove (T, QUEUE(sem));
  }
  else
    sem++;
}
```

Semaphore (negative Zähler)

```
wait (sem) {
  sem--;      // immer!
  if (sem < 0) {
    enqueue (T, QUEUE(sem));
    sleep();
  }
}
```

```
signal (sem) {
  if (T in QUEUE(sem)) {
    wakeup (T);
    remove (T, QUEUE(sem));
  }
  sem++;      // immer!
}
```

Selbstkontrollaufgabe 22: Server-Steuerung

Skalierender Server via Ticket-Transfer im Lotterie-Scheduler

Ähnliches Szenario mit Semaphoren

[Lösungshinweis] Idee:

- Starten Sie mehrere Server-Threads, die alle sofort

  ```
  wait(sem);
  ```

  ausführen (und blockieren).
- Jeder Client, der eine Anfrage an den Server schickt, führt

  ```
  signal(sem);
  ```

  aus, um einen der Server-Threads zu deblockieren, damit dieser die Anfrage bearbeiten kann.

Selbstkontrollaufgabe 20: Erzeuger-Verbraucher-Problem

Live-Demo

Listing 5.5: Erzeuger-Verbraucher-Problem: Lösung mit Mutexen und Semaphoren.

```
1  typedef int semaphore;
2  #define  N  100        // Anzahl der Plätze im Puffer
3  semaphore mutex = 1;   // kontrolliert Zugriff auf Puffer
4  semaphore empty = N;   // zählt freie Plätze im Puffer
5  semaphore full  = 0;   // zählt belegte Plätze im Puffer
6
7  void producer() {
8    do forever {
9      produce_item (item);
10     wait (empty);       // leere Plätze –=1 oder blockieren
11     wait (mutex);       // Anfang kritischer Bereich
12     enter_item (item);  // Zugriff auf Puffer (schreiben)
13     signal (mutex);     // Ende kritischer Bereich
14     signal (full);      // belegte Plätze +=1,
15                         // evtl. consumer-Thread wecken
16   }
17 }
18
19 void consumer() {
20   do forever {
21     wait (full);        // belegte Plätze –=1 oder blockieren
22     wait (mutex);       // Anfang kritischer Bereich
23     remove_item(item);  // Zugriff auf Puffer (lesen)
24     signal (mutex);     // Ende kritischer Bereich
25     signal (empty);     // freie Plätze +=1,
26                         // evtl. producer-Thread wecken
27     consume_item (item);
28   }
29 }
```