Betriebssysteme und Sicherheit, WS 2025/26

5. Aufgabenblatt – Semaphore

Besprechungszeitraum: 25.11. bis 28.11.

Semaphore Ein Semaphor ist ein abstrakter Datentyp, bestehend aus einer Warteschlange, einer Zählvariable und zwei unteilbaren Operationen P( ) (prolaag oder oft auch *down*, *wait*) und V( ) (verhoog oder oft auch *up*, *signal*) . Ein Semaphor funktioniert wie eine Art Signalgeber für einen kritischen Abschnitt. Will ein Thread diesen Abschnitt betreten (`wait( )`), wird überprüft, ob der Bereich noch frei ist (Zählvariable positiv) – wenn ja, wird der Thread durchgelassen und die Zählvariable dekrementiert; wenn nein, wird der Thread blockiert und in die Warteschlange eingereiht. Beim Verlassen des Abschnittes (`signal( )`) wird unterschieden, ob mindestens ein blockierter Thread in der Warteschlange ist, oder nicht. Wenn ja, wird dieser aus der Warteschlange entfernt und aufgeweckt, und kann jetzt den Abschnitt betreten. Wenn nein, wird die Zählvariable inkrementiert. Je nach Initialisierung der Zählvariable darf nur ein Thread (*binärer Semaphor*) oder dürfen mehrere Threads (*zählender Semaphor*) den Abschnitt betreten.

Besondere Eingenschaften Threads, die auf das Betreten eines kritischen Abschnittes warten, werden im Betriebssystem blockiert und entsprechend nicht ausgeführt (*passives Warten*). Anders als bei einfachen *Spinlocks*, die dauerhaft prüfen, ob der kritische Abschnitt wieder frei ist (*aktives Warten*), schont dieser Ansatz Ressourcen, benötigt dafür aber besondere Unterstützung vom Betriebssystem.

Aufgabe 5.1 Beschreiben Sie die Steuerung für ein System, bei dem Fahrzeuge über eine Brücke wollen, für folgende Fälle der maximalen Belastbarkeit der Brücke:

(a) ein Fahrzeug, unabhängig von der Richtung (1 Fahrspur)

(b) ein Fahrzeug je Richtung gleichzeitig (2 Fahrspuren)

(c) drei Fahrzeuge je Richtung gleichzeitig (2 Fahrspuren)

Verwenden Sie Semaphore.

Aufgabe 5.2 **Why did the multithreaded chicken cross the road?** Die drei Funktionen des folgenden Programms werden in jeweils eigenen Threads ausgeführt, die alle zur selben Zeit laufbereit werden. Sorgen Sie durch geeignete Synchronisation der Threads dafür, dass das Programm den Text *„to get to the other side“* ausgibt. Dafür stehen Ihnen drei Semaphore zur Verfügung, die Sie geeignet initialisieren müssen. Setzen Sie an den freien Stellen Semaphor-Operationen (`signal`, `wait`) auf die Semaphore (S1, S2, S3) ein (z. B. `S1.wait()`). Initialwerte der Semaphore:

S1: [ ], S2: [ ], S3: [ ]

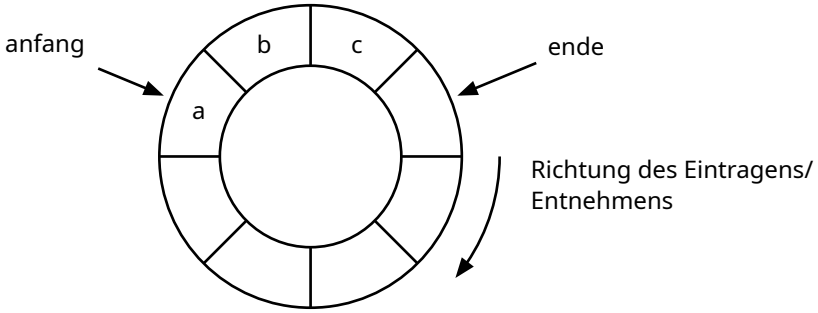
```
chicken1() {

  printf("to ");

  printf("to ");

  printf("other ");

}
```

```
chicken2() {

  printf("get ");

}
```

```
chicken3() {

  printf("the ");

  printf("side ");

}
```

Aufgabe 5.3 **Synchronisation mittels Erzeuger-Verbraucher-Problem**. Gegeben sei ein Programm, das

aus zwei Threads besteht, die über einen gemeinsamen Puffer Elemente austauschen. Der Puffer ist ein Ringpuffer, der durch den Thread A mit Elementen gefüllt und durch den Thread B geleert wird (gemäß FIFO). Implementiert wird der Ringpuffer dabei als ein n-elementiges Feld (n konstant), welches wie in der folgenden Abbildung gezeigt genutzt wird.



(a) Welche Situationen können bei dieser Implementierung auftreten, die eine Synchronisation bedingen?

(b) Beschreiben und diskutieren Sie eine Lösung des beschriebenen Programms unter der Nutzung des Erzeuger-Verbraucher-Problems in Pseudo-Code.

Klausuraufgabe I

Folgender C-Code sei gegeben:

```
1  int x = 0;
2
3  int inc(void)
4  {
5    x++;
6  }
```

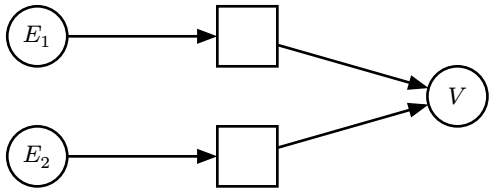
Die Funktion `inc()` werde von **zwei Threads** gleichzeitig aufgerufen.

(a) Erläutern Sie an diesem Beispiel den Begriff der Wettlaufsituation (englisch „Race Condition“)! Nennen Sie mögliche Ergebnisse nach Ausführung der Funktion durch beide Threads!

(b) Nennen Sie vier Techniken, die zu einem deterministischen Ergebnis führen! Erwähnen Sie für jede Technik, ob diese mit Busy Waiting arbeitet!

Hinweis: Die Funktionalität von `inc()` soll erhalten bleiben; ebenso die Parallelausführung der Funktion durch zwei Threads.

Ein Erzeuger-Verbraucher-System bestehe aus zwei Erzeugern E_1 und E_2, die wiederholt jeweils einen einelementigen Puffer befüllen. Diese Puffer werden durch einen gemeinsamen Verbraucher V geleert. Dabei kann V die Puffer in beliebiger Reihenfolge bearbeiten. Insbesondere kann V arbeiten, sobald einer der beiden Puffer gefüllt ist.



(c) Geben Sie Implementierungen für Erzeuger und Verbraucher in Pseudocode an, bei der kein Busy Waiting auftritt und die maximale Parallelität gewährleistet wird!

Klausuraufgabe II

In einem Programm mit 2 Threads werden die unten folgenden Codeteile zur Absicherung eines kritischen Abschnittes verwendet. Dabei bezeichnet `/* kA */` den kritischen Abschnitt, für dessen Ausführung die Kriterien des wechselseitigen Ausschlusses garantiert werden soll.

Globale Variablen

```
1  int next = 1;
2  int t1 = 0, t2 = 0;
```

Thread 1

```
1  t1 = 1;
2  next = 2;
3  while(t2 == 1 && next == 1) {}
4  /*
5      kA
6  */
7  t1 = 0;
```

Thread 2

```
1  t2 = 1;
2  next = 1;
3  while (t1 == 1 && next == 2){}
4  /*
5      kA
6  */
7  t2 = 0;
```

(a) Bei einem Codereview wird festgestellt, dass der verwendete Code wechselseitigen Ausschluss nicht gewährleistet. Begründen Sie, wieso der Reviewer Recht hat.

(b) Schlagen Sie eine Änderung des oben stehenden Codes vor, sodass das gefundene Problem behoben wird und der Code die Anforderungen an wechselseitigen Ausschluss erfüllt.

Klausuraufgabe III

(a) Nennen Sie jeweils einen Nachteil der folgenden Mechanismen für wechselseitigen Ausschluss.

- Interrupts ausschalten
- Atomare Instruktionen
- Dekker-Algorithmus

(b) In einem Supermarkt wird neue Ware von einem Lieferanten angeliefert und von zwei Lageristen eingeräumt. Dabei stehen auf der Laderampe insgesamt 10 Plätze zum Ablegen von Paketen zur Verfügung. Ergänzen Sie folgenden Code um die nötige Synchronisierung zwischen dem Lieferant und den Lageristen mittels Semaphoren (Datentyp `sem_t`). Fügen Sie außerdem Code ein, um die Pakete auf der Laderampe abzulegen bzw. dieser zu entnehmen. Dabei sollen alle Beteiligten maximal parallel arbeiten und es soll kein aktives Warten (*busy waiting*) auftreten.

```
int n_frei =      ; int n_voll =      ;

paket_t plaetze[      ];

                                                    


void lieferant(paket_t ware) {
                                                    
    plaetze[n_frei] = ware;
                                                    
}
void lagerist() {
                                                    
    paket_t ware = plaetze[n_voll];
                                                    
    einraeumen(ware);
                                                    
}
```