Lyubomira Dimitrova, Dung Nguyen, Maryna Charniuk

Aufgabe 1

a) Mutexsperren insgesamt und Verteilung auf verschiedene Threads:

- 1 Thread: ∼0 Mal
- 2 Threads: ∼900 Mal (30: 870)
- 5 Threads: ∼2900 Mal (200: 1000: 800: 800: 100)
- 10 Threads: Min. 0 und max. ∼1200 Mutexsperren pro Thread, meistens zwischen 100 und 300
- 100 Threads: Min. 0 und max. ∼11500 Mutexsperren pro Thread, etwa die Hälfte zwischen 300 und 400

Es ist ersichtbar, dass die Verteilung der Mutexsperren auf die Threads nicht besonders 'fair' ist - in den 1., 4. und 5. Fall schafft der 0-te Thread keine einzige Mutexsperre. Das hat uns etwas überrascht, da er als erster erzeugt wurde.

b) Ausführen mit `taskset -c 3`

- 1 Thread: Keiner der Threads schafft es, Mutex zu sperren.
- 2 Threads: Keiner der Threads schafft es, Mutex zu sperren.
- 5 Threads: Keiner der Threads schafft es, Mutex zu sperren.
- 10 Threads: Keiner der Threads schafft es, Mutex zu sperren.
- 100 Threads: Ein einziger Thread (48) hat die Mutexsperre ∼322390 Mal erlangt. Bei anderen Ausführungen waren aber alle Ergebnisse = 0.

Aufgabe 2

- *Level 1*:

Thread 1

```
1 while (true) {
2  while (flag != false) {
3    ;
4  }
5  flag = true;
6  critical_section();
7  flag = false;
8 }
```

Thread 2

```
1 while (true) {
2  while (flag != false) {
3    ;
4  }
5  flag = true;
6  critical_section();
7  flag = false;
8 }
```

Vorgegebene globale Variable(n), die sich beim Ausführen der Threads verändern

```
bool flag = false;
```

Die beiden Threads haben die gleichen Code. Um in die kritische Region gleichzeitig bei beiden Threads einzutreten, muss die innere while-Schleife übersprungen werden. Lösung: Im Thread 1 führen wir den Code bis einschließlich Zeile 5 aus (mit `Step`), dann führen wir im Thread 2 auch bis einschließlich Zeile 5 aus. Da das Variable `flag == true` jetzt nach dem Ausführen in beiden Threads, können wir in beiden Threads gleichzeitig in die kritische Region eingehen (noch mal in jedem Thread `Step` aufdrücken).

- *Level 2:*

Thread 1

```
1 while (true) {
2    counter++;
3    if (counter == 5) {
4        critical_section();
5    }
6 }
```

Thread 2

```
1 while (true) {
2    counter++;
3    if (counter == 3) {
4        critical_section();
5    }
6 }
```

Vorgegebene globale Variable(n), die sich beim Ausführen der Threads verändern

```
int counter = 0;
```

`counter` wird in beiden Threads weiter inkrementiert, bis die Bedingungen in jedem Thread erreicht werden, um in die kritische Region einzugehen. In diesem Fall führen wir zuerst den Code im Thread 2 aus (`Step` aufdrücken), bis `counter` $== 3$, dann können wir in die kritische Region im Thread eingehen. Anschließend tun wir dasselbe im Thread 1, bis `counter` $== 5$.

- *Level 3:*

Thread 1

```
1 business_logic();
2 first++;
3 second++;
4 if (second == 2 && first != 2) {
5    Debug.Assert(false);
6 }
```

Thread 2

```
1 business_logic();
2 first++;
3 second++;
```

Vorgegebene globale Variable(n)

```
int first = 0;
int second = 0;
```

Expandierte Zeilen
Zeile 2

```
temp = first + 1;
first = temp;
```

Wir führen den Code in beiden Threads bis einschließlich Zeile 2. Hier benutzen wir die Operation `Expand` und die Zeile `first++;` wird in 2 weiteren Zeilen expandiert. Wir führen dann diese 2 Zeilen in beiden Threads aus. Danach führen wir die Zeile 3 in beiden Threads (also nacheinader). Wenn wir Zeile 4 im Thread 1 erreichen, haben wir `second` == 2 und `first` != 2 (== 1). Wird die Bedingung erfüllt, wird die Zeile 5 erreicht und `Assert` wird ausgelöst.

- *Level 4:*

Thread 1

```
1 while (true) {
2    Monitor.Enter(mutex);
3    i = i + 2;
4    critical_section();
5    if (i == 5) {
6        Debug.Assert(false);
7    }
8    Monitor.Exit(mutex);
9 }
```

Thread 2

```
1 while (true) {
2    Monitor.Enter(mutex);
3    i = i - 1;
4    critical_section();
5    Monitor.Exit(mutex);
6 }
```

Vorgegebene globale Variable(n)

```
object mutex;
int i = 0;
```

Wir führen den Code in Thread 1 einmal aus, dann `i` == 2. Des Weiteren führen wir den Code im Thread 2 einmal aus, dann `i` == 1. Wir führen den Code im Thread 1 noch zweimal aus, bis `i` == 5 und die Bedingung in Zeile 5 erfüllt ist, dann wird die Zeile 6 erreicht und `Assert` wird ausgelöst.

- *Level 5:*

Thread 1

```
1 Monitor.Enter(mutex);
2 Monitor.Enter(mutex2);
3 critical_section();
4 Monitor.Exit(mutex);
5 Monitor.Exit(mutex2);
```

Thread 2

```
1 Monitor.Enter(mutex2);
2 Monitor.Enter(mutex);
3 critical_section();
4 Monitor.Exit(mutex2);
5 Monitor.Exit(mutex);
```

Vorgegebene globale Variable(n)

```
object mutex;
object mutex2;
```

Wir führen den Code der Zeile 1 in beiden Threads jeweils aus. Dann wird `mutex` im Thread 1 blockiert und `mutex2` im Thread 2 blockiert. Im Thread 1 muss es warten bis `mutex2` im Thread 2 wieder frei gegeben wird, und umgekehrt, im Thread 2 muss es warten bis `mutex` im Thread 1 wieder frei gegeben wird. Die beiden Threads haben einander blockiert.

- *Level 6:*

Thread 1

```
1 while (true) {
2    if (Monitor.TryEnter(mutex)) {
3        Monitor.Enter(mutex3);
4        Monitor.Enter(mutex);
5        critical_section();
6        Monitor.Exit(mutex);
7        Monitor.Enter(mutex2);
8        flag = false;
9        Monitor.Exit(mutex2);
10        Monitor.Exit(mutex3);
11    } else {
12        Monitor.Enter(mutex2);
13        flag = true;
14        Monitor.Exit(mutex2);
15    }
16 }
```

Thread 2

```
1 while (true) {
2    if (flag) {
3        Monitor.Enter(mutex2);
4        Monitor.Enter(mutex);
5        flag = false;
6        critical_section();
7        Monitor.Exit(mutex);
8        Monitor.Enter(mutex2);
9    } else {
10        Monitor.Enter(mutex);
11        flag = false;
12        Monitor.Exit(mutex);
13    }
14 }
```

Vorgegebene globale Variable(n)

```
object mutex;
object mutex2;
object mutex3;
bool flag = false;
```

Wir führen den Code im Thread 2 bis einschließlich Zeile 11 (Zeile 11 wird auch ausgeführt) aus. `mutex` wird im Thread 2 blockiert. Dann führen wir den Code im Thread 1 bis einschließlich Zeile 14 (Zeile 14 wird auch ausgeführt) aus (Bedingung im Block **if** nicht erfüllt, deswegen wird Code im Block **else** ausgeführt). Danach führen wir den Code im Thread 2 bis einschließlich Zeile 3 aus. Hier wird `mutex2` wird im Thread 2 blockiert. Wir führen den Code

im Thread 1 bis einschließlich Zeile 7 aus. Hier werden `mutex3` und `mutex` im Thread 1 blockiert. Da `mutex2` schon im Thread 2 blockiert wird, werden die beiden Threads voneinander blockiert.

- Die Operation `Expand` wird benutzt wenn es in beiden (oder mehreren) Threads eine Zeile (oder mehrere Zeilen) gibt, die unbedingt in allen Threads gleichzeitig/nur einmal ausgeführt werden (also wie bei *Level 3*, `first` wird nur 1 mal inkrementiert, und kann deswegen nicht in beiden Threads inkrementiert werden, deswegen brauchen wir in diesem Fall `Expand`).
- Tatsächlich führen wir den Code in beiden Threads abwechselnd aus, bis das Ziel erreicht wird. Es kann 2 mal oder mehr Schritten gebraucht werden und es hängt von der Struktur von Code beider Threads ab, eine oder mehrere Abfolgen zu haben. Als Scheduler ist es nicht problematisch in beliebiger Abfolge auszulösen.

Aufgabe 3

Thread 0

```
1 while (TRUE) {
2    //erlange die Sperre
3    enter_region(0);
4    kritische Region
5    // setze die Sperre frei
6    leave_region(0);
7    restlicher Code
8 }
```

Thread 1

```
1 while (TRUE) {
2    //erlange die Sperre
3    enter_region(1);
4    kritische Region
5    // setze die Sperre frei
6    leave_region(1);
7    restlicher Code
8 }
```

Code für `enter_region` und `leave_region`

```
1 # define FALSE 0
2 # define TRUE 1
3 # define N 2
4 int turnWait;
5 int interested[N];
6
7 void enter_region (int process) {
8     int other = 1 - process;
9     interested[process] = TRUE;
10    turnWait = process;
11    while (turnWait == process &&  interested [other] == TRUE) ; }
12
13 void leave_region (int process) {
14    interested [process] = FALSE;
15 }
```

1. Die gesamte Routine `enter_region` wird genau einmal durchlaufen.
Ausführung 1, nur bei Thread 0, Durchlauf 1 von `while (TRUE)`-Schleife, `enter_region` in Zeile 3 wird aufgerufen

- Zeile 8 ausgefürht, ein Variable `other` wird erstellt und den Wert `1 - process` zugewiesen. `other = 1`
- Zeile 9 ausgeführt, `interested[process] (interested[0])` wird auf `TRUE` gesetzt:
  - Kopiere Inhalt des Speichers mit Adresse `interested[0] (= 0 ≡ FALSE)` in CPU-Register `EAX`: `MOV EAX, M[interested[0]]`. Inhalt des Register `RX = interested[0] = 0`.
  - Setze den Inhalt von `EAX` auf 1 (≡ TRUE): `RX = 1`.
  - Kopiere Inhalt des Registers `EAX` in den Speicher mit Adresse `interested[0]`: `MOV M[interested[0]], EAX`
- Zeile 10 ausgeführt, `turnWait` wird auf den Wert vom `process (= 0)` zugewiesen.
  - Kopiere Inhalt des Speichers mit Adresse `turnWait (= 0)` in CPU-Register `EAX`: `MOV EAX, M[turnWait]`. Inhalt des Register `RX = turnWait = 0`.
  - Setze den Inhalt von `EAX` auf `process` (≡ 0): `RX = 0`.
  - Kopiere Inhalt des Registers `EAX` in den Speicher mit Adresse `interested[0]`: `MOV M[turnWait], EAX`
- Zeile 11 ausgeführt, Bedingungen `turnWait == process && interested[other] == TRUE` werden geprüft, um zu testen, ob das Thread 0 warten muss (wenn die Bedingungen erfüllt werden.)
  - Kopiere Inhalt des Speichers mit Adresse `turnWait (= 0)` in CPU-Register `EAX`: `MOV EAX, M[turnWait]`. Inhalt des Register `RX = turnWait = 0`.
  - Vergleicht den Inhalt von Register `RX` mit dem Wert von `process` (= 0): `CMP RX, #0`, es stimmt. Dann wird die zweite Bedingung getestet.
  - Kopiere Inhalt des Speichers mit Adresse `interested[other] (= interested[1] = 0 ≡ FALSE)` in CPU-Register `EAX`: `MOV EAX, M[interested[0]]`. Inhalt des Register `RX` = `interested[1]` = 0.
  - Vergleicht den Inhalt von Register `RX` mit dem Wert von `TRUE` (≡ 1): `CMP RX, #1`. Es stimmt nicht, dann wird ein bedingter Sprung ausgeführt: `JNE`.
  - Da die letzte Zeile von `enter_region` erreicht wird, wird diese Routine verlassen und kehrt man zur Stelle ihres Aufrufs (Zeile 3 im Thread 0 und weiter zur Zeile 4) zurück: `RET`.

2. Die Schleife in `enter_region` wird beim zweiten Durchlauf verlassen.
Ausführung 1, bei Thread 0, Durchlauf 1 von `while (TRUE)`-Schleife, `enter_region` in Zeile 3 (Thread 0) wird aufgerufen.

- Im ersten Durchlauf bis Zeile 3 ausgeführt passiert es genauso wie im **1.**.

- Zeile 4 ausgeführt. Thread 0 tritt in die kritische Region ein. Dann wird Thread 1 blockiert und kann nicht Zeile 3 bei ihm ausführen, das Variable `LOCK = 1`.
- Zeile 6 ausgeführt. Die Sperre wird frei gesetzt.
  - Kopiere Inhalt des Speichers mit Adresse `interested[process] (interested[0] = 1 ≡ TRUE)` in CPU-Register `EAX`: `MOV EAX, M[interested[0]]`. Inhalt des Register `RX = interested[0] = 1`.
  - Setze den Inhalt von `EAX` auf `0` (≡ FALSE): `RX = 0`.
  - Kopiere Inhalt des Registers `EAX` in den Speicher mit Adresse `interested[0]`: `MOV M[interested[0]], EAX`
- Zeile 7 ausgeführt. Erster Durchlauf ist fertig.

Durchlauf 2 von `while (TRUE)`-Schleife, `enter_region` in Zeile 3 (Thread 0) wird aufgerufen. Es wird genau sowie im **1.** passieren.

Aufgabe 4

a) Bei der Verwendung von Bedingungsvariablen muss man selbst auf die Zustandsvariablen achten und sie pflegen (z.B., Slots Counter). Im Gegenteil dazu haben Semaphore eine eingebaute Zustandsvariable, die mit `wait()` und `signal()` geändert werden kann.

Zustandsvariablen benutzt man, um Konstrukte für den gegenseitigen Ausschluss und für das Warten unabhängig zu machen (souveränen). Man benutzt Mutexe für den wechselseitigen Ausschluss und Bedingungsvariablen für die Synchronisierung. Semaphoren können für beides benutzt werden.

Ein Mutex wird am Anfang eines Codeblocks gesperrt und am Ende desselben freigegeben. Im Gegensatz dazu können Semaphore an mehrmals und an verschiedenen stellen im Code die Sperre setzen und aufheben.

Wenn ein Thread ein `signal()` broadcastet und zu diesem Zeitpunkt kein Thread darauf wartet, der Signal wird auf den nächsten `wait`-Aufruf warten (merken).

b) Der Signal hat kein Gedächtnis oder Historie, von welchem Thread er aufgerufen wird. z.B., in dem Fall mit 2 Threads, die `signal()` aufrufen, der wartende Thread wird nicht sehen, dass den Signal aufgerufen wird und wird ewig warten. Darüber hinaus muss Mutex prüfen, dass es zwischen Änderung der Zustandsvariable und Signalisieren nichts von anderem Thread passieren kann.

Aufgabe 5

Angenommen `S` ist mit 1 initialisiert, und die folgenden Situationen gelten für alle Threads.

a) Reihenfolge von `wait()` und `signal()` vertauschen

```
signal(S);
    // critical section
wait(S);
```

Der Wert von `S` nach `signal(S)` ist 2, also ≥ 0. Während der Code in der kritischen Region ausgeführt wird, können andere Threads auch in die kr. Region gelangen, da `signal()` nicht blockierend ist.

b) `signal()` durch `wait()` ersetzen.

```
wait(S);
    // critical section
wait(S);
```

Nach dem ersten Aufruf wird `S` ganz normal dekrementiert und die kritische Region betreten. Beim zweiten Aufruf von `wait(S)` wird man in der Schleife bleiben, bis woanders 2-mal `signal(S)` aufgerufen wird (`S` muss positiv sein, damit `wait()` verlassen wird). Wann das passieren wird ist aber unklar, was zu viele blockierte Threads führen kann.

c) `wait()` und/oder `signal()` auslassen.

```
wait(S);
    // critical section
...
```

`S` wird dekrementiert, was heißt, dass kein anderer Thread den eigenen `wait()` verlassen kann. In dem Fall, dass ein anderer Thread `signal()` **vor** `wait()` (oder einfach nur `signal()`) aufruft, kann die Reihenfolge der Ausführung der Threads dadurch gesteuert werden.

```
...
    // critical section
signal(S);
```

Nichts sperrt den Zugang zur kritischen Region. Der Semaphor wird in jedem Thread nur inkrementiert.

```
...
    // critical section
...
```

Chaos. Anarchy. Who will sit on the iron throne???