1. Deadlock: Code-Beispiel

Sie haben in der Vorlesung am Minimalbeispiel mit zwei Threads T1 und T2 und zwei Locks A und B gesehen, dass sich Deadlocks verhindern lassen, indem Sie die Locks A und B immer in gleicher Reihenfolge anfordern. Das lässt sich auf beliebig viele Threads und beliebig viele Locks wie folgt verallgemeinern:

> **Regel für Deadlock-Vermeidung**
>
> – Bringe alle Locks in eine (beliebige) Reihenfolge
>
> – Fordere Locks, wenn mehrere benötigt werden, immer unter Beachtung dieser Reihenfolge an.

Das macht das gleichzeitige Auftreten von Code-Abschnitten der Form

```
lock (A);                lock (B);
lock (B);                lock (A);
...
unlock (B);              unlock (B);
unlock (A);              unlock (A);
```

unmöglich: Wurde als Reihenfolge etwa A, B festgelegt, ist nur der linke Code-Block regelkonform, während der rechte Code-Block die Regel verletzt, weil dort B und A nicht in der festgelegten Reihenfolge angefordert werden.

Das folgende Listing zeigt einen Ausschnitt aus der Datei `deadlock.c`, die Sie im Archiv `swf.hgesser.de/bs-b1/prakt/bs1-ue10.zip` finden und auf die übliche Weise entpacken können.

```
31      int main () {
32        pthread_mutex_init(&A, NULL);
33        pthread_mutex_init(&B, NULL);
34
35        pthread_create (&t1, NULL, thread1, NULL);  // Threads erzeugen
36        pthread_create (&t2, NULL, thread2, NULL);
37
38        #ifdef CHEATING
39          // Cheating
40          sleep (2);
41          printf ("main:   Nach 2 Sekunden: Cheat (unlocking A)...\n");
42          pthread_mutex_unlock (&A);    // illegal!
43        #endif
```

Der Code in den Zeilen 38–43 wird zunächst nicht mit übersetzt, weil das Präprozessor-Makro `CHEATING` nicht definiert ist.

a) Übersetzen und starten Sie das Programm in der unveränderten Version, und überzeugen Sie sich davon, dass es in einen Deadlock läuft.

b) Entfernen Sie die Kommentarzeichen (`//`) in Zeile 29 und testen Sie das Programm (nun mit aktivierten Zeilen 38–43) erneut.

– Was ändert sich nun,
– woran liegt das,
– und ist diese Vorgehensweise empfehlenswert?

Deaktivieren Sie das Cheating in den Zeilen 38–43 nach dem Bearbeiten dieser Teilaufgabe wieder.

c) Da die Beachtung der Regel für Deadlock-Vermeidung (wir nennen sie *Regel 1*) umständlich ist, wird ein alternatives Verfahren (*Regel 2*) vorgeschlagen: In der Anwendung wird ein zusätzliches Lock `super` definiert, das immer vor dem ersten Aufruf von `lock()` als erstes gelockt werden muss. Wenn

`super` gelockt wurde, dürfen die übrigen Ressourcen in beliebiger Reihenfolge gelockt werden. Erst nachdem alle regulären Ressourcen zurück gegeben wurden, darf der Code auch `super` zurückgeben. Die beiden obigen Code-Blöcke sind nach den neuen Regeln also zulässig, erhalten aber die folgende Form (neuer Code **fett**):

```
lock (super);              lock (super);
lock (A);                  lock (B);
lock (B);                  lock (A);
...
unlock (B);                unlock (B);
unlock (A);                unlock (A);
unlock (super);            unlock (super);
```

(i) Passen Sie das Programm `deadlock.c` so an, dass es *Regel 2* verwendet. Lassen Sie das angepasste Programm dann mehrfach laufen. Treten jetzt Deadlocks auf?

(ii) Warum sorgt *Regel 1* immer für Deadlock-Freiheit?

(iii) Sorgt auch *Regel 2* immer für Deadlock-Freiheit?

(iv) Es ist viel einfacher, beim Programmieren *Regel 2* umzusetzen, als *Regel 1* umzusetzen. Aber ist dieser Ansatz empfehlenswert?

d) Entfernen Sie das `super`-Lock wieder (indem Sie alle Stellen auskommentieren, an denen es verwendet wird). Lösen Sie dann das Deadlock-Problem, wie schon in der Vorlesung gezeigt, indem Sie Regel 1 umsetzen.

2. Deadlock-Erkennung

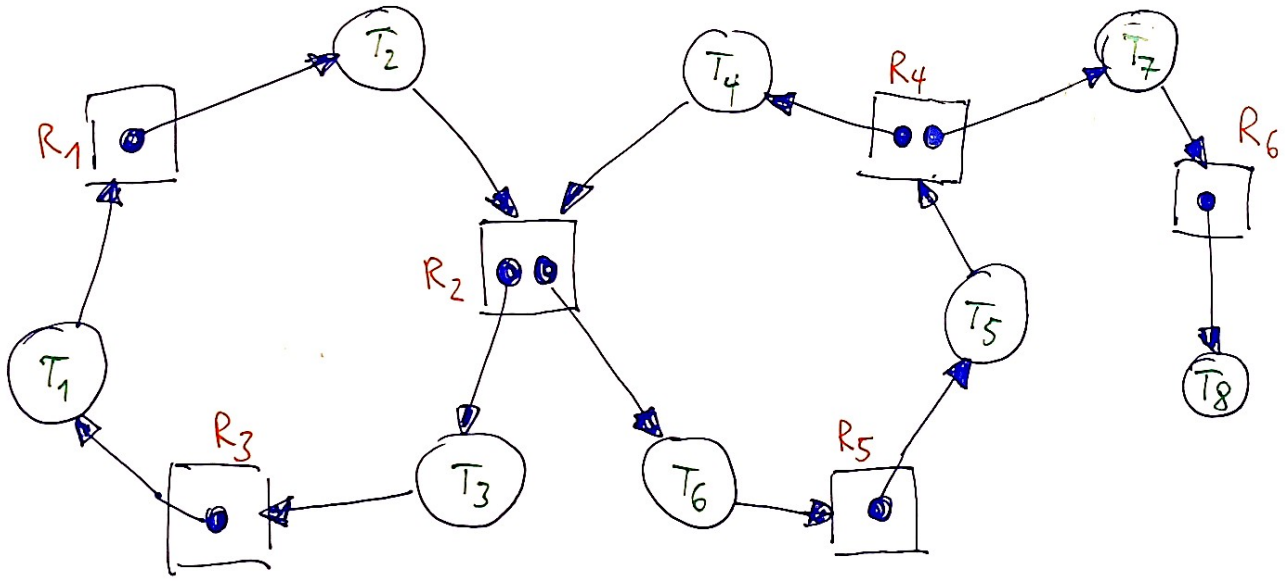
a) Es gebe fünf Threads T1 bis T5 sowie sechs Ressourcen R1 bis R6. Es gelten dabei die folgenden Belegungen und Anforderungen:

- T1 hat R2 belegt und fordert R5 an.
- T2 hat R4 belegt und fordert R2 und R3 an.
- T3 hat R3 belegt und fordert R1 an.
- T4 hat R1 belegt und fordert R6 an.
- T5 hat R5 belegt und fordert R4 an.

Zeichnen Sie den Ressourcen-Zuordnungsgraph für dieses Szenario und leiten Sie daraus ab, ob sich die fünf Threads im Deadlock-Zustand befinden. Begründen Sie Ihre Antwort.

b) Bonus-Aufgabe: Schreiben Sie ein Programm, das die Situation aus Aufgabe **a)** herstellt, und testen Sie es. Fordern Sie dabei zunächst die als schon belegt bezeichneten Locks an und warten Sie dann an einer Barriere – erst hinter der Barriere fordern Sie die weiteren Locks an. (Warum ist das nötig?)

c) Betrachten Sie den folgenden Ressourcenzuordnungsgraph:



(i) Gibt es einen orientierten Kreis im Graph?

(ii) Ist das System im Deadlock?