Marek Kubica, kubica@in.tum.de

Punkteverteilung:

1	2	3	Σ

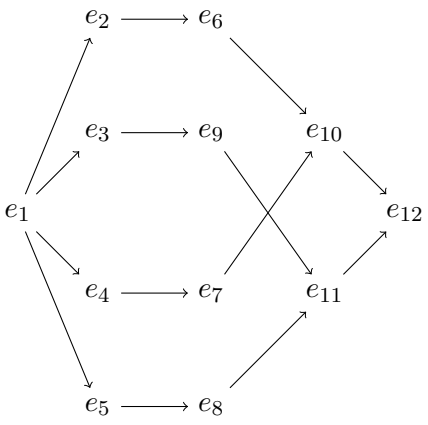
Aufgabe (1)

(a)

e_1	Laura und Fred vor der Bank	a_1
e_2	Fred geht zu Schalter A	a_2
e_3	Laura geht zu Schalter B	a_2
e_4	Fred geht zu Schalter B	a_2
e_5	Laura geht zu Schalter A	a_2
e_6	Fred wird an Schalter A beraten	a_3
e_7	Fred wird an Schalter B beraten	a_3
e_8	Laura wird an Schalter A beraten	a_3
e_9	Laura wird an Schalter B beraten	a_3
e_{10}	Fred ist fertig	a_4
e_{11}	Laura ist fertig	a_4
e_{12}	Laura und Fred verlassen die Bank	a_5

(b) $e_1 \leq e_2$, $e_1 \leq e_3$, $e_1 \leq e_4$, $e_1 \leq e_5$, $e_1 \leq e_6$, $e_3 \leq e_9$, $e_4 \leq e_7$, $e_5 \leq e_8$, $e_6 \leq e_{10}$, $e_8 \leq e_{11}$, $e_9 \leq e_{11}$, $e_{10} \leq e_{12}$ $e_{11} \leq e_{12}$

(c)



(d) Gegenseitige Nebenläufigkeit von e_2, e_6, e_{10} zu e_3, e_9, e_{11} sowie e_4, e_7, e_{10} zu e_5, e_8, e_{11}.

(e) P_1': $e_1 \rightarrow e_2 \rightarrow e_6 \rightarrow e_3 \rightarrow e_9 \rightarrow e_{10} \rightarrow e_{11} \rightarrow e_{12}$

Algorithmus: Für jedes Element, das von keinem voerherigen Ereignis abhängt: Führe es aus und lösche Abhängigkeit zu diesem Ereignis aus der Abhängigkeits-Liste aller anderen Ereignisse. Wiederhole, bis alle Ereignisse durchgelaufen sind.

(f) $2 * 6!$ Möglichkeiten

(g) Annahme: Menge aller Sequentialisierungen beschreibt Prozess nicht eindeutig.

Folge: Prozess lässt sich in einer Reihenfolge ausführen, die nicht einer seiner Sequentialisierungen entspricht. Diese Ausfürung wäre allerdings wieder eine Sequentialisierung des Prozesses. Widerspruch zur Annahme!

(h) 1.2 beschreibt einen Zustandsautomaten wenn man die Zweige e_2 mit e_3, e_6 mit e_9, e_{10} mit e_{11}, e_4 mit e_5, e_7 mit e_8 zu Knoten zusammenfasst.

Aufgabe (2)

(a) a) **Race condition**: ein unerwarteter Fall, der nur bei einer speziellen, meist unerwarteten Reihenfolge der Threadausführung auftritt

b) **Mutual exclusion**: Wenn mehrere Threads den Zugriff auf eine exklusive Resource benötigen, ist es notwendig dass sie sichergehen, dass sie diese Resource nicht beide zur gleichen Zeit nutzen – sie müssen sich also wechselseitig ausschließen.

c) **Critical region/section**: Codeabschnitt, der nur von einem Thread zum gleichen Zeitpunkt ausgeführt werden darf.

d) **Busy waiting**: Warten auf eine Resource in einer Endlosschleife

e) **Deadlock**: Wenn ein Thread auf eine bestimmte Aktion eines anderen Threads wartet und der andere Thread wiederrum auf den ersten Thread wartet.

(b) a) Höhere Ressourcenbelegung, Deadlocks

b) Ohne atomare Befehle wie TSL wirds schwerer, Multiprozessoren machen Speicherzugriff zudem noch komplizierter

c) Ja, es reichen aber atomare Operationen aus.

(c) a) Zwischen der Prüfung im While-Kopf und dem blokieren des anderen Prozesses kann ein Threadwechsel stattfinden und der andere Thread greift auf den kritischen Bereich zu. Daraufhin wechselt der Kontext wieder und beide Threads sind also im kritischen Bereich. Fail!

b) Ohne TSL oder ähnlichen Befehl nicht möglich, da es sich womöglich um Mehrkern/Mehrprozessorsysteme handelt.

(d)
```
P1 & P2
BEGIN
P();
-- kritischer Bereich
V();
END;
```

Aufgabe (3)

(a) Der Graph sieht kreisförmig aus. Eine Kante von einer Ressource (Straße) zu einem Prozess (Auto) bedeutet dabei, dass die Ressource das Auto am fahren hindert, weil keine Vorfahrt gewährt wird. Jede Ressource wird von genau einem Auto belegt, und jeder Prozess fordert genau eine Ressource ("links" von sich) an.

(b)
- Kreuzung nicht exklusiv machen / mehrere Ebenen / Autobahnkreuz-Modell
- Kreuzung durch Ampel regeln / unterbrechbar machen
- Straßen Prioritäten zuordnen
- Zyklus entfernen