BuS-Übungsgruppe 7 Übungsblatt 6 Felix Kiunke, 357322
Philipp Hochmann, 356148
Daniel Schleiz, 356092

1,5 Punkte

6.1

a) 0,5

Zum Einen können dadurch in gewissen Fällen Deadlocks verhindert werden, da eine Rückgabe von false auf einen Fehler bzw. ein Deadlock hinweisen kann, woraufhin man mittels if-Verzweigung eine Behandlung des Problems programmieren könnte.
Außerdem kann es sein, dass ein Prozess mittels wait auf einen gewissen Bereich Zugriff will (oder z.B. Zugriff auf ein Betriebsmittel möchte), aber vielleicht auch ohne Zugriff darauf noch sinnvoll anderen Code ausführen könnte und dann zu einem späteren Zeitpunkt nochmal mittels wait einen Zugriff anfragt. Hierbei wäre dann so eine Implementierung von Semaphoren sinnvoll, da man mittels if-else und dem Rückgabewert ein solches System realisieren könnte.

1

b)

```
boolean wait(semaphore s, boolean blocking):
   if(blocking == false) {
      // Wenn die Semaphore gerade gelocked ist, kann nix gemacht werden, also return false
      if(s.lock == true) {
         return false;
      } else {
         s.lock == true;
         // Wenn der Wert der Semaphore kleiner als 1 ist, kommt der Prozess nicht rein
         if(s.value < 1) {
            s.lock = false;
            return false;
         } else {
            s.value = s.value - 1;
            s.lock = false;
            return true;
         }
      }
   } else {
      // Sonst Code wie vorher im blockierenden Fall
      while (TestAndSet(s.lock)) {
         noop;
      }
      s.value = s.value - 1;
      if (s.value < 0) {
         s.waiting_processes.add(get_pid());
         s.lock = false;
         block();
      } else {
         s.lock = false;
      }
   }
```

ein gleich zu viel

2 Punkte **6.2**

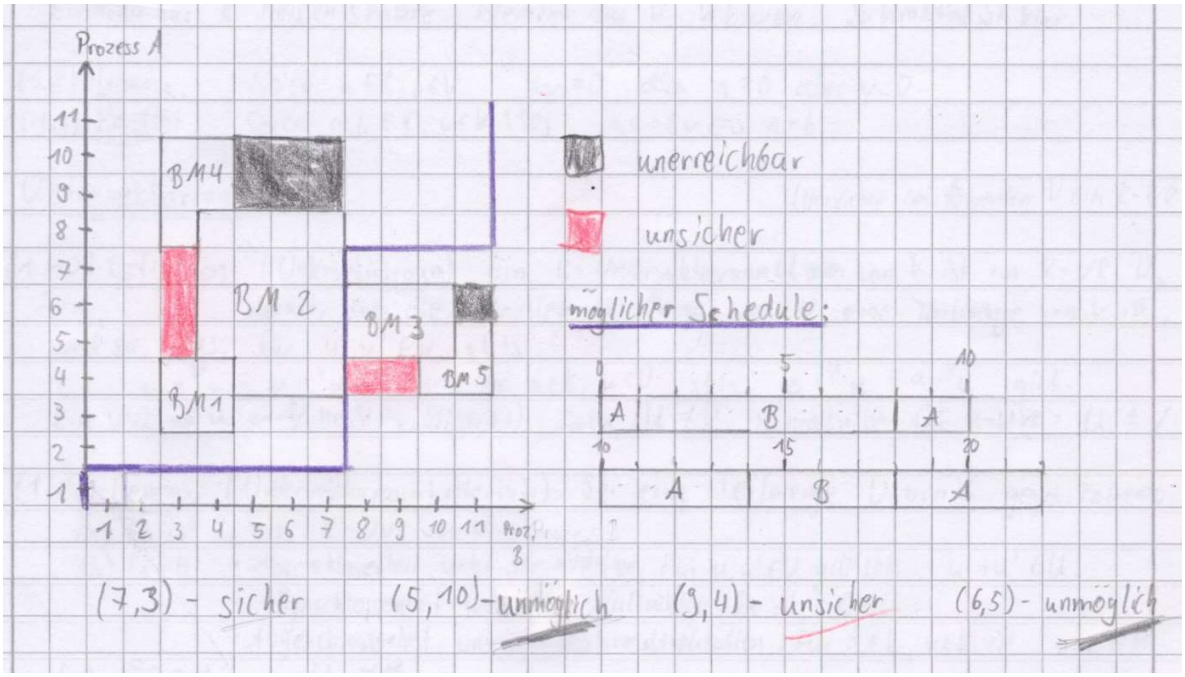
```
// Zählvariable, zählt die Anzahl der wartenden Prozesse
int process_count = 0;

// Mutex, synchronisiert Zugriff auf die Zählvariable (der wartenden Prozesse)
int count_access = 1;

// Zählsemaphor, lässt Prozess durch wenn alle n angekommen sind
int release = 0;

pass() {
   wait(count_access);
   if(process_count < n-1) {
      process_count++;
      signal(count_access);
      wait(release);
   }
   // Sonst ist nach Voraussetzung die Anzahl der Prozesse gleich n-1, also gebe frei
   else {
      int i;
      // Wecke alle Threads auf
      for(i = 1; i <= (n-1); i++) {
         signal(release);
      }
      process_count = 0;
      signal(count_access);
   }
}
```

6 Punkte **6.3**



6 Punkte **6.4**

a) 3

Verfügbar: (9, 5, 6)

1) Prozess 1: kann nicht laufen, benötigt (11, 1, 2)
   Prozess 2: kann nicht laufen, benötigt (3, 6, 4)
   Prozess 3: kann laufen, benötigt (2, 0, 0)
2) Prozess 3 wird markiert und gibt Betriebsmittel (3, 1, 1) zurück

Verfügbar: (12, 6, 7)

1) Prozess 1: kann laufen, benötigt (11, 1, 2)
2) Prozess 1 wird markiert und gibt Betriebsmittel (0, 2, 1) zurück

Verfügbar: (12, 8, 8)

1) Prozess 1: bereits markiert
   Prozess 2: kann laufen, benötigt (3, 6, 4)
2) Prozess 2 wird markiert und gibt Betriebsmittel (1, 0, 1) zurück

Verfügbar: (13, 8, 9)

1) Alle Prozesse wurden markiert, das System befindet sich daher in einem sicheren Zustand.

Es ergibt sich die Ausführungsreihenfolge Prozess 3, Prozess 1, Prozess 2.

b) 2

1. (0, 2, 0) darf an Prozess 2 gegeben werden, da Prozess 3 und 1 auch ohne diese Betriebsmittel wie in a) laufen können. Es wären dann noch (9, 3, 6) verfügbar, was für P3 reicht und nach Freigabe der Ressourcen von P3 auch für P1.

2. (0, 0, 2) darf an Prozess 2 gegeben werden, da Prozess 3 und 1 auch ohne diese Betriebsmittel wie in a) laufen können. Es wären dann noch (9, 5, 4) verfügbar, was für P3 reicht und nach Freigabe der Ressourcen von P3 auch für P1.

3. (0, 0, 1) darf an Prozess 3 gegeben werden, da dieser in der Abfolge ohnehin als erster läuft. Er wird seine Betriebsmittel daher nach der Ausführung wieder abgeben und die Prozesse 1 und 2 können wie in a) laufen.

4. (0, 1, 0) darf an Prozess 3 gegeben werden, da dieser in der Abfolge ohnehin als erster läuft. Er wird seine Betriebsmittel daher nach der Ausführung wieder abgeben und die Prozesse 1 und 2 können wie in a) laufen.

c) 1

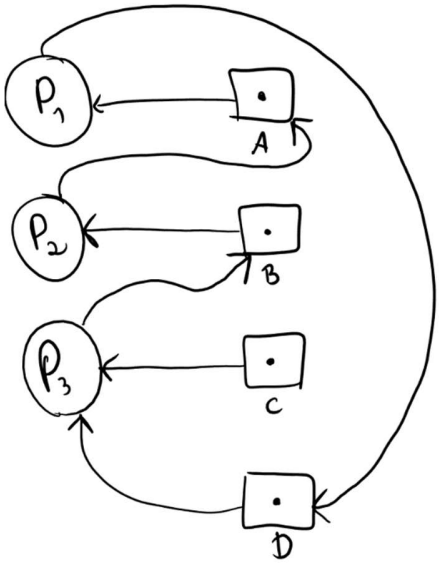
Wenn das System die Zuteilung von Ressourcen ablehnt, kann es sein, dass diese gerade von einem anderen Prozess exklusiv benutzt werden. Es könnte aber auch sein, dass der Bankier-Algorithmus ermittelt hat, dass der Systemzustand nicht mehr sicher ist, wenn ein Prozess bestimmte Betriebsmittel hält. Ein unsicherer Zustand muss nicht zwangsläufig zu einem Deadlock führen, da es möglich ist, dass Prozesse terminieren, ohne alle Betriebsmittel, die in Qmax angegeben sind, zu benötigen. Es handelt sich nur um Maximalwerte.

4 Punkte

6.5

Zum Zeitpunkt 7 (ZP 7) liegt ein Deadlock vor, da Prozess 1 (P1) Betriebsmittel A (BM A) hält, der zum ZP 6 wieder zur Verfügung stehen sollte. Dazu kommt es aber nicht, weil zum ZP 5 BM D angefordert wird, welches bereits von P3 gehalten wird, der nicht weiterarbeiten kann, weil es ihm an BM B fehlt, welches von P2 gehalten wird. P2 kann allerdings nur dann weiterarbeiten und BM B zum Zeitpunkt 8 schließlich freigeben, wenn Betriebsmittel A von P1 freigegeben wird.

Dies ist auch durch den Kreis im Resource Allocation Graph sichtbar. Weiterhin gilt nach Aufgabenstellung „Hold and Wait“ und „Non-Preemption“. Zusammen mit der festgestellten geschlossenen Kette und der exklusiven Nutzbarkeit der Betriebsmittel sind alle der vier Bedingungen für einen Deadlock erfüllt.



Insgesamt 19,5/20