Aufgabe 1: Speicherverwaltung

a) Wir berechnen die virtuellen Adressen. Es gibt 2^16 Adressen, die auf 16 Seiten verteilt werden.
2^16 B / 16 = 4096 B
Eine Seite entspricht 4096 Bit

c) Die physikalischen Adressen ergeben sich aus folgender Rechnung mit den virtuellen Adressen:
0x5fe8 mod 16 = 8
0xfeee mod 16 = 14
0xa470 mod 16 = 0
0x0101 mod 16 = 1

d) Argumente für kleine Seiten:
Ein zufälliges Text-, Daten- oder Stack-Segment wird keine ganze Zahl von Seiten füllen. Das heißt, dass bei hoher Seitengröße oft eine Seite nur zum Teil ausgefüllt wird und dadurch Platz verschwendet wird.
Außerdem werden große Programme, die aus mehreren kleinen Phasen bestehen, in großen Seiten die ganze Zeit abgelegt sein, obwohl nur Teile des Programmes arbeiten und der Rest inaktiv ist.
Argument für große Seiten:
Durch kleinere Seiten bekommen wir eine größere Seitentabelle, wodurch es zu größeren Such- und Rotationszeiten kommt.

e) Innere Fragmentierung besagt, dass wenn ein Programm Speicher belegt, die Hälfte der letzten Seite im Durchschnitt leer bleibt und damit Platz verschwendet wird. Wenn die Seitentabellengröße sehr hoch ist, sorgt dies für eine geringere innere Fragmentierung, da wir im Durchschnitt zwar immer noch die letzte Seite nur zur Hälfte füllen, aber die Seite durch ihre geringere Größe weniger Platz verschwendet.
Laut Tanenbaum ergibt sich die optimale Seitengröße aus /wurzel (2* s * e), wobei s die Prozessgröße und e die Länge eines Seitentabelleneintrags ist.
Daraus ergibt sich für unseren Fall:
Seitengröße = /wurzel (8B * 4MiB) = /wurzel (8B * 4096000 B) = 8095B

Aufgabe 2: Seitenersetzungsalgorithmen

a) (i)

t	1	2	3	4	5	6	7	8	9	10	11	12
Angeforderte Seite	1	2	3	4	2	1	2	5	6	2	6	3
Zugriffart	r	w	w	r	r	r	r	w	r	w	w	r
Seitenalarm	j	j	j	j	n	j	n	j	j	j	n	j
Seiten im Speicher	1	1,2	1,2,3	4,2,3	4,2,3	1,2,3	1,2,3	1,5,3	1,5,6	2,5,6	2,5,6	3,5,6

(ii)

T	1	2	3	4	5	6	7	8	9	10	11	12
Angeforderte Seite	1	2	3	4	2	1	2	5	6	2	6	3
Zugriffart	r	w	w	r	r	r	r	w	r	w	w	r
Seitenalarm	j	j	j	j	n	j	n	j	j	j	n	j
Seiten im Speicher	1	1,2	1,2,3	2,3,4	2,3,4	4,2,1	4,2,1	2,1,5	1,5,6	5,6,2	5,6,2	6,2,3

Aufgabe 3: Synchronisation

a) Initialisierung:
W = true
NumberOfActiveReaders = 0
Mutex = true

```
processWriter():
if (W == true AND NumberOfActiveReaders == 0)
        W = false
        Mutex = false
        writeData()
        W = true
        Mutex = true

processReader():
if (W == true AND Mutex == true)
        Mutex = false
        NumberOfActiveReaders += 1
        Mutex = true
        readData()
        if (Mutex == true)
                Mutex = false
                NumberOfActiveReaders -= 1
                Mutex = true
```

Für jedes if sei ein else definiert, sodass nach einer kurzen Wartezeit die if-Anforderung wieder überprüft wird.

b) Schreibprozesse im Allgemeinen werden bei unserer Lösung benachteiligt, da die Variable NumberOfActiveReaders permanent über 0 sein kann, ohne dass der Schreibprozess etwas dagegen tun kann. Dies kann verhindert werden, indem man eine weitere Variable hinzufügt, die für die Priorität eines Prozesses steht. Jeder Prozess startet bei 0 und wird um 1 erhöht, wenn er durch den else-Teil von processWriter/Reader geht. Außerdem wird die derzeitig größte Priorität in einer globalen Variable gespeichert. Ein neuer Prozess fängt erst an zu lesen oder zu schreiben, falls er die höchste Priorität hat.