Technische Universität Berlin
Fachgebiet
Distributed and Operating Systems

Sommersemester 2022

Aufgabenblatt 3

zu – Systemprogrammierung –

Prof. Dr. Odej Kao

Aufgabe 3.1: Synchronisation (Tafelübung)

Die Abläufe zwischen Verkäufern und Kunden in einem Dönerladen sollen synchronisiert werden. In diesem Dönerladen gibt es einen Spieß und mindestens zwei Verkäufer. Der Spieß kann nur von einem Verkäufer gleichzeitig genutzt werden. Auf den Salat kann von allen gleichzeitig zugegriffen werden.

a) Im Folgenden ist der Verkäufer-Prozess in Pseudocode beschrieben. Dabei machen wir uns erst einmal noch keine Sorgen um zu viel produzierte Döner. Ergänzen Sie die nötige Synchronisation.

```
Variablen:
int döner = 0;

Verkäufer:
while(true){
  fleischSchneiden();
  salatUndSauce();
  döner++;
}
```

Lösung:

Listing 1: Global

```
1 int döner = 0;
2 Mutex spieß_m;        // Lockvariable für Spieß
3 Mutex döner_m;        // Lockvariable für Dönervariable
```

Listing 2: Verkäufer

```
1 while(true) {
2   lock(spieß_m);
3   fleischSchneiden();
4   unlock(spieß_m);
5   salatUndSauce();
6   lock(döner_m);
7   döner++;
8   unlock(döner_m);
9 }
```

Die unterstrichenen Zeilen zeigen an was sich gegenüber der Vorgabe bzw. letzten Lösung geändert hat. Zusammengehöriger Synchronisationscode hat jeweils die selbe Textfarbe.
Sofern möglich, empfiehlt sich zur besseren Nachvollziehbarkeit auch ein eingefärbtes Tafelbild.

b) Kunden betreten in unvorhersagbaren Abständen den Laden, um einen Döner zu kaufen (wir simulieren das durch startende Kunden-Prozesse). Sie können nur Döner essen, wenn auch Döner fertig sind, andernfalls müssen sie warten. Erweitern Sie Ihre Lösung aus der letzten Aufgabe dafür um den nachfolgenden angegebenen Kunden-Prozess und die notwendige Synchronisation.

```
Kunde:
  döner--;
  dönerEssen();
```

Lösung:

Die Unteraufgabe sollte gemeinsam mit den Teilnehmern entwickelt werden. Die nummerierten Kommentare in der Musterlösung geben hierbei eine sinnvolle Erweiterungs-Reihenfolge vor.

Listing 3: Global

```
1  int döner = 0;
2  Mutex spieß_m;
3  Mutex döner_m;
4  Mutex kunde_halt_m;     // 6) Siehe Anmerkung unten
5  Signal dönerFertig_S; // 3) für Info an Kunde
```

Listing 4: Verkäufer

```
1  while(true){
2    lock(spieß_m);
3    fleischSchneiden();
4    unlock(spieß_m);
5    salatUndSauce();
6    lock(döner_m);
7    döner++;
8    signal(dönerFertig_S);     // 3) informiere wartenden Kunden
9    unlock(döner_m);
10 }
```

Listing 5: Kunde

```
1  lock(kunde_halt_m);      // 6) Siehe Anmerkung unten
2
3  lock(döner_m);           // 2) Variablenzugriff schützen
4
5  // 1) Provisorium, in 5) ersetzen
6  if (döner <= 0) {
7
8  // 5) jetzt if durch while ersetzen:
9  //(prüfe Wartebedingung nach Wakeup!)
10 while (döner <= 0) {
11     unlock(döner_m); // 4) Mutex freigeben (Deadlocks!)
12     wait(dönerFertig_S);  // 3) Schlafen, bis Döner fertig
13     lock(döner_m);     // 4) vor Variablentest, Mutex neu locken
14 }
```

```
15  döner --;
16  unlock(döner_m);      // 2) Variablenzugriff freigeben
17  unlock(kunde_halt_m);       // 6) Siehe Anmerkung unten
18  dönerEssen();
```

Bei Verwendung von `wait()` (wie in der Vorlesung vorgestellt) muss mittels `kunde_halt_m` ein zusätzlicher kritischer Abschnitt definiert werden: Angenommen, zwei Kunden würden nach dem `unlock` in **??**, Z. 7 vom Scheduler unterbrochen. Würden dann zwei Döner gleichzeitig fertig werden, würde nur eines der beiden Signale gespeichert werden. Der als erstes weiterlaufende Kundenprozess würde dieses auslesen und löschen, der zweite daraufhin blockieren - obwohl ein zweiter Döner vorhanden ist! Eleganter ist hier eine `wait`-Operation, die einen zuvor belegten `lock` atomar freigeben kann, wie z.B. `pthread_cond_wait()`.

Allgemeiner Hinweis zu `signal`/`wait`: Es sollte nie auf Signale selbst, sondern immer auf das Eintreten bestimmter Bedingungen (hier: `döner > 0`) gewartet werden. Das Programm sollte auch nach dem Entfernen sämtlicher `signal`- und `wait`-Aufrufe noch funktionieren!

c) Verkäufer sollen nur dann etwas produzieren, wenn auch ein Kunde auf den Döner wartet. Ergänzen Sie Ihre Lösung aus der letzten Aufgabe um die dafür notwendige Synchronisation.

Lösung:

Analog zum bereits vorhandenen Dönerzähler muss nun zusätzlich noch ein Kundenzähler eingefügt werden. Die weitere Vorgehensweise entspricht der in der vorherigen Unteraufgabe.

Listing 6: Global

```
1  int döner = 0;
2  int kunden = 0; // neue Variable
3  Mutex spieß_m;
4  Mutex döner_m;
5  Mutex kunde_halt_m;
6  Mutex verkäufer_halt_m; // neuer Mutex
7  Mutex kunden_m; // neuer Mutex
8  Signal dönerFertig_S;
9  Signal neuerKunde_S; // neues Signal
```

Listing 7: Verkäufer

```
1  while(true){
2    lock(verkäufer_halt_m); // 3) siehe Anmerkung in b
3    lock(kunden_m);          // 2) auf Kunden kontrollieren
4    while(kunden <= 0){     // 2)
5      unlock(kunden_m);     // 2)
6      wait(neuerKunde);     // 2)
7      lock(kunden_m);       // 2)
8    }
9    kunden --;             // 2)
10   unlock(kunden_m);      // 2)
```

```
11    unlock(verkäufer_halt_m);    // 3)
12
13    lock(spieß_m);
14    fleischSchneiden();
15    unlock(spieß_m);
16    salatUndSauce();
17    lock(döner_m);
18    döner++;
19    signal(dönerFertig_S);
20    unlock(döner_m);
21  }
```

Listing 8: Kunde

```
1  lock(kunden_m);        // 1) Kudenanzahl verändern
2  kunden++;              // 1)
3  signal(neuerKunde);    // 1)
4  lock(kunden_m);        // 1)
5
6  lock(kunde_halt_m);
7  lock(döner_m);
8  while (döner <= 0) {
9    unlock(döner_m);
10   wait(dönerFertig_S);
11   lock(döner_m);
12 }
13 döner --;
14 unlock(döner_m);
15 unlock(kunde_halt_m);
16 dönerEssen();
```

Aufgabe 3.2: POSIX Threads (Tafelübung)

a) Was ist der Unterschied von Threads und Prozessen? Wie sieht dieser im Hinblick auf die POSIX-Bibliothek pThreads aus? Geben Sie zudem Möglichkeiten an, wie Threads untereinander kommunizieren können, sowie berechnete Ergebnisse an den Parent weitergeben, bzw. von diesem bei Start bekommen können.

Lösung:

Bei pThreads laufen die Threads alle im gleichen Adressraum (vgl. Prozesse, jeder Prozess hat einen eigenen Adressraum); jeder Thread hat zwar seinen eigenen Stack und damit auch seine eigenen lokalen Variablen, jedoch globale Variablen und per malloc angeforderter Speicher können von jedem Thread gelesen und geschrieben werden.

Jeder Thread führt eine Funktion aus und endet entweder mit `pthread_exit` oder wenn die Funktion terminiert. Ein Thread kann einen Pointer als Rückgabewert und als Eingabewert be-

kommen. Hierdurch lassen sich Argumente, sowie Rückgabewerte übergeben.

b) Es ist das folgende Ping/Pong-Programm gegeben. Dieses soll mit der pThreads POSIX-Bibliothek so implementiert werden, dass die Threads im Wechsel „Ping“ und „Pong“ ausgeben. Spurious Wakeups sollen berücksichtigt werden.

Listing 9: main

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  enum action{PING, PONG};
5
6  int main() {
7    enum action *nextAction = malloc(sizeof(enum action));
8    *nextAction = PING;
9
10   thread_ping(&nextAction);
11   thread_pong(&nextAction);
12
13   free(nextAction); // not really necessary
14 }
```

Listing 10: Thread 1

```
1  void *thread_ping(void *nAction) {
2    enum action *nextAction = (enum action *) nAction;
3
4    while(1) {
5      printf("Ping\n");
6      *nextAction = PONG;
7    }
8  }
```

Listing 11: Thread 2

```
1  void *thread_pong(void *nAction) {
2    enum action *nextAction = (enum action *) nAction;
3
4    while(1) {
5      printf("Pong\n");
6      *nextAction = PING;
7    }
8  }
```

Lösung:

Listing 12: main

```
1  #include <stdio.h>
```

```
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 enum action{PING, PONG};
 6 pthread_mutex_t mutex;
 7 pthread_cond_t cond;
 8
 9 int main() {
10   pthread_t t_ping;
11   pthread_t t_pong;
12
13   pthread_mutex_init(&mutex, NULL);
14   pthread_cond_init(&cond, NULL);
15
16   enum action *nextAction = malloc(sizeof(enum action));
17   if (nextAction == NULL) exit(EXIT_FAILURE);
18   *nextAction = PING;
19
20   pthread_create(&t_ping, 0, thread_ping, nextAction);
21   pthread_create(&t_pong, 0, thread_pong, nextAction);
22 // ab hier theoretisch nicht zwingend nötig, da endlos-Threads
23 // dennoch den Studis zeigen, wie normalerweise zu enden...
24   pthread_join(t_ping, NULL);
25   pthread_join(t_pong, NULL);
26   pthread_mutex_destroy(&mutex);
27   pthread_cond_destroy(&cond);
28
29   free(nextAction);
30 }
```

Listing 13: Thread 1

```
 1 void *thread_ping(void *nAction) {
 2   enum action *nextAction = (enum action *) nAction;
 3
 4   while(1) {
 5     printf("Ping\n");
 6
 7     pthread_mutex_lock(&mutex);
 8     *nextAction = PONG;
 9     pthread_cond_signal(&cond);
10
11     while(*nextAction != PING) {
12       pthread_cond_wait(&cond, &mutex);
13     }
14     pthread_mutex_unlock(&mutex);
15   }
16 }
```

Listing 14: Thread 2

```
1  void *thread_pong(void *nAction) {
2    enum action *nextAction = (enum action *) nAction;
3
4    while(1) {
5      pthread_mutex_lock(&mutex);
6      while(*nextAction != PONG) {
7        pthread_cond_wait(&cond, &mutex);
8      }
9      pthread_mutex_unlock(&mutex);
10
11     printf("Pong\n");
12
13     pthread_mutex_lock(&mutex);
14     *nextAction = PING;
15     pthread_cond_signal(&cond);
16     pthread_mutex_unlock(&mutex);
17   }
18 }
```

Aufgabe 3.3: Periodische Prozesse (Selbststudium)

Die Firma „Pen&Pencil“ möchte einen neuartigen Stift auf den Markt bringen. Dieser soll speziell in Meetings eingesetzt werden können und folgende Funktionen bieten: A) Die Beschleunigung aufzuzeichen, sodass Geschriebenes einfach digitalisiert werden kann, B) Diese Daten (aus einem Puffer) auf die enthaltene MicroSD-Karte zu schreiben und C) Geschriebenes sofort ohne Verzögerung auf entsprechenden Boards über eine drahtlose Verbindung übertragen. Hierbei ist es wichtig, dass diese Aufgaben ohne Verzögerung möglichst schnell (ohne Verletzung der Deadline) und zuverlässig ausgeführt werden. In der nachfolgenden Tabelle sind die beispielhaften Eckdaten einer solchen Benutzung dargestellt: Dauer der Aufgabe und Periode, die zeitgleich auch die Frist (Deadline) ist. Alle Prozesse starten zeitgleich bei $t = 0$.

Tabelle 1: Prozesse

Prozesse	**Dauer (D)**	**Periode (P)**
A	1	3
B	1	5
C	1	5

a) Existiert für diese Prozesse ein zulässiger Schedule? Wird das notwendige Kriterium erfüllt?

b) Wie könnte dieser aussehen? Geben Sie etwaige Leerzeiten an und markieren Sie die Hyperperiode.

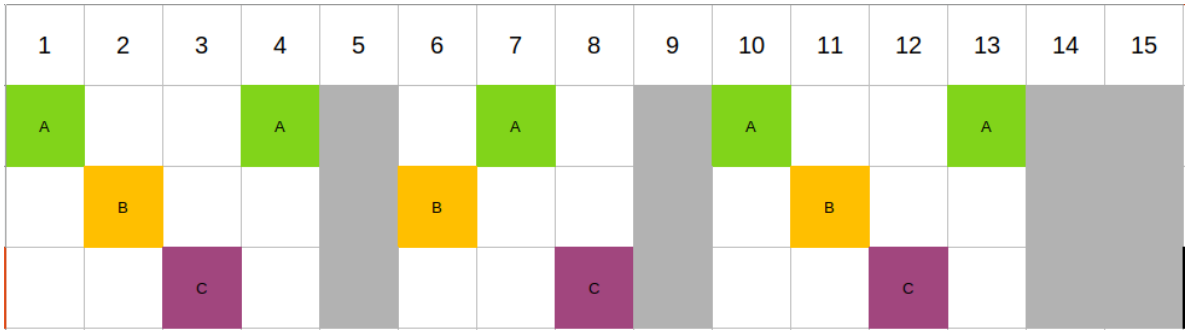
c) Ist Rate-Monotonic-Scheduling (RMS) ein gültiger Schedule? Begründen Sie Ihre Antwort.

d) Was passiert, wenn zu den Prozessen ein weiterer Prozess, Prozess D mit (D=2, P=9), hinzugefügt wird?

Lösung:

a) Ja, gibt es. Notwendige Bedingung: $\sum_{i=1}^{n} \frac{D_i}{P_i} = (\frac{1}{3} + \frac{1}{5} + \frac{1}{5}) \approx 0,73 \leq 1$.

b) Die Hyperperiode beträgt $t_{HP} = LCM(P_i) = LCM(3,5,5) = 15$, also reicht es aus, nur diese darzustellen, da sich danach alles wiederholt.



c) Ja, da die hinreichende Bedingung erfüllt wird: $\sum_{i=1}^{n} \frac{D_i}{P_i} = (\frac{1}{3} + \frac{1}{5} + \frac{1}{5}) \approx 0,73 \leq n * (\sqrt[n]{2} - 1) \approx 0,7798$

d) Es gibt keinen zulässigen Schedule mehr für RMS aber sonst, ja, da die hinreichende Bedingung ($\sum_{i=1}^{n} \frac{D_i}{P_i} = (\frac{2}{6} + \frac{1}{4} + \frac{1}{6} + \frac{2}{9}) \approx 0,9555 \not\leq n * (\sqrt[n]{2} - 1) \approx 0,7568$) nicht erfüllt ist, und die notwendinge Bedingung ($\sum_{i=1}^{n} \frac{D_i}{P_i} = (\frac{2}{6} + \frac{1}{4} + \frac{1}{6} + \frac{2}{9}) \approx 0,9555 \leq 1$ erfüllt ist.

Aufgabe 3.4: Priority Inversion (Selbststudium)

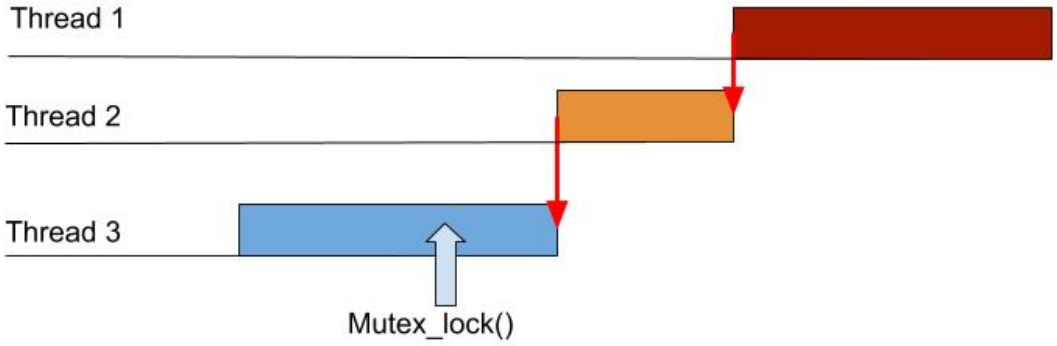
In 1997 ist der Mars Pathfinder auf dem Mars gelandet.
Der hatte einen gemeinsamnen Informationsbus und einen watchdog timer der überprüft, ob das System noch arbeitet. Aufgaben wurden auf Threads verteilt mit verschiedene Prioritäten. Als scheduling algorithmus wurde eine Priority-queue mit Verdrängung benutzt.
Drei Threads waren Periodisch (in Reinfolge von Prioritäten):

Thread 1. Informationsbus-Thread: sehr häufig und hohe Priorität

Thread 2. Kommunikations-Thread, mittel häufig und mittel hohe Priorität (kann sehr lange laufen)

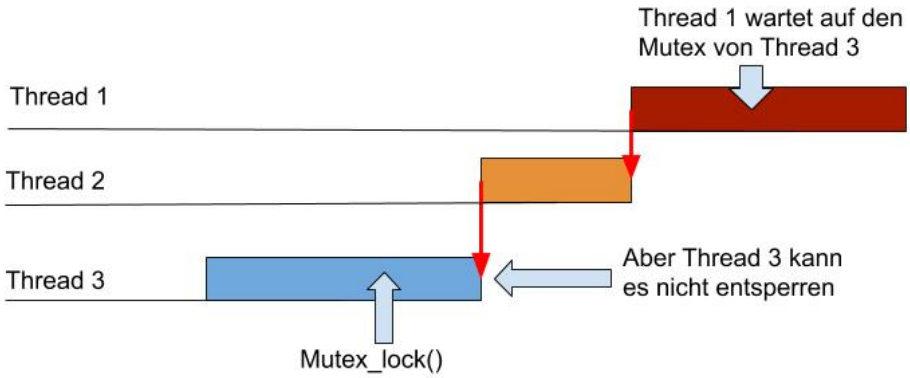
Thread 3. Wetter Daten sammeln: nicht häufig, niedrige Priorität (Braucht wie Thread 1 und 2 auch den Speicher)

a) Nach paar Tagen hat der Watchdog Timer immer des System neu gestartet wegen eines Problems. Überlegen sie was passiert, wenn Thread 3 dran kommt, den geteilten Speicher mit Thread 1 sperrt, dann will Thread 2 den langen Kommunikationsprozess anfangen, und danach will Thread 1 starten.



Lösung:

Thread 3 wurde unterdrukt und Thread 2 hat angefangen. Häufige Thread 1 started und unterdrükt Thread 2. Thread 1 wartet da der Speicher noch geblockt ist von Thread 3. (Priority Inversion)



b) Überlege Sie wie in diesen Fall Thread 3 den Speicher wieder freischalten könte.

Lösung:

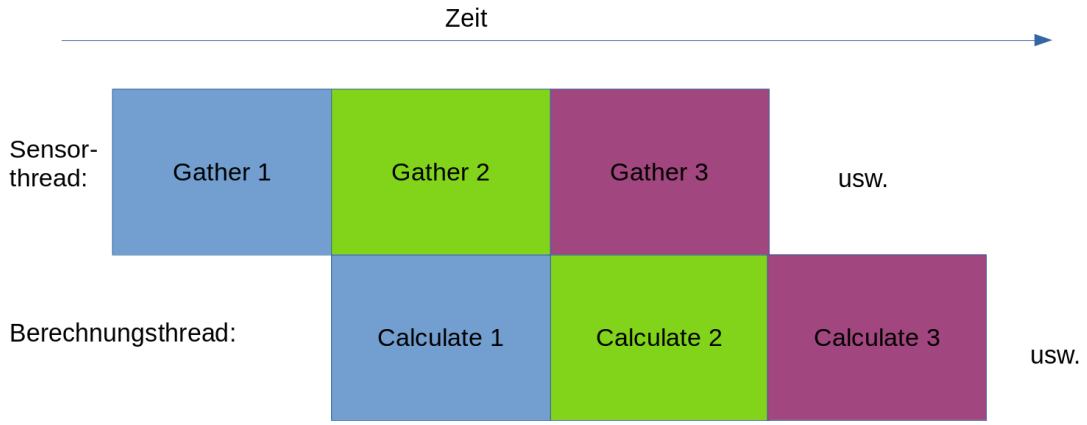
Wenn ein Thread anfängt und auf ein blockeirtes Element wartet, bekommt der Thread der den Lock gesetzt hat die Priorität von anfragenden Thread. (Priority Inheritance)

Aufgabe 3.5: Synchronisation/Kooperation (Selbststudium)

Sie sollen einen Smart-Home Wettersensor entwickeln. Dieser soll mithilfe von eingebauten Sensoren Wetterdaten erfassen und diese danach auswerten.
Die Funktion `gather_data(char *buffer)` wird zur Datenmessung aufgerufen, und schreibt während des Messens Messdaten in den übergebenen Speicher.
Die Funktion `calculate_forecast(char *buffer)` wertet die übergebenen Messdaten aus und schickt das Ergebnis an gewünschte Geräte im Heimnetzwerk.

Der Sensor misst die Umgebungsdaten mithilfe von `gather_data` (dies dauert ca. 30 min). Nach jedem neuen Datensatz soll eine neue Wettervorhersage berechnet werden (`calculate_forecast`). Das Berechnen der Vorhersage dauert ebenfalls ca. 30 min. Während die alten Messdaten ausgewertet werden, sollen die Sensoren direkt wieder mit dem Messen von neuen Werten beginnen.

Hinweis:
Ein beispielhafter zeitlicher Ablauf
(die Nummern stehen hier für beispielhafte Datensätze):



Global:

```
char buffer[1024];
state s = gather;        // Werte: 'gather' oder 'copy_data'
```

Sensor-Thread:

```
while(1) {
        gatherData(&buffer);
        s = copy_data;
}
```

Berechnungs-Thread:

```
while(1) {
        char internal_buffer[1024];
        memcpy(internal_buffer,buffer,1024);
        s = gather;
        calculate_forecast(internal_buffer);
}
```

Hinweis: Gehen Sie in dieser Aufgabe davon aus, dass Signal, auf die zum Zeitpunkt des Sendens nicht gewartet wird, gespeichert werden.

a) Die beiden Threads sollen nun nebenläufig ausgeführt werden. Nennen Sie ein praktisches Problem, das dabei auftreten kann.

Lösung:

Es können Lese/Schreib Konflikte beim Buffer auftreten.

b) Was sind *Spurious Wakeups*? Und wie kann man sicher stellen, dass die eigentliche Bedingung erfült ist, auch in fall von Spurious wakeup?

Lösung:

Die POSIX-Threading-Implementierung erlaubt es sich, zusätzliche Wakeups zu erzeugen. Diese Wakeups werden „Spurious Wakeup“ genannt.
pthread_cond_wait() Aufruf immer mit einer Schleife die die eigentliche Bedingung prüft, umschliesßen.Folie 50 Koordination nebenläufiger Prozesse.

c) Verbessern Sie obiges Programm mit *signal/wait* und *mutex*, sodass Probleme verhindert werden. Da die Datenerhebung(`Gather_Data`) sehr lange dauert, sollte damit schon begonnen werden, während der alte Datensatz ausgewertet wird(`calculate_forecast`)(siehe Beispielablauf). Außerdem sollten beim Auswerten keine Datensätze übersprungen werden, um stets aktuelle Daten zu gewährleisten. Ihre Lösung sollte auch Spurious Wakeups berücksichtigen.
Hinweis: Sie können zusätzliche globale Variablen verwenden.

Lösung:

Global:

```
char buffer[1024];
Signal sensor_fertig,Datensatz_kopiert;
mutex m;
state s = gather;
```

sensor-Thread:

```
while(1) {
        lock(m);
        gatherData(&buffer);
        s = copy_data;
        signal(sensor_fertig)
        while(s != gather) {
                unlock(m)
                wait(Datensatz_kopiert);
                lock(m)
        }
```

```
        unlock(m);
}
```

Berechnungs-Thread:

```
while(1) {
        lock(m);
        while(s != copy_data) {
                unlock(m);              //atomic op here?
                wait(sensor_fertig);
                lock(m);
        }
        char internal_buffer[1024];
        memcpy(internal_buffer,buffer,1024);
        s = gather;
        signal(Datensatz_kopiert);
        unlock(m);
        calculate_forecast(internal_buffer);
}
```

d) Welche Arten der expliziten Prozess- /Threadinteraktion gibt es? Um welche Art der Interaktion handelt es sich hier?

Lösung:

Es gibt die Interaktionsarten Kooperation und Konkurenz. (vgl, Foliensatz Prozesskoordination", Folie 2ff). Es handelt sich hierbei um Kooperation, sowie Konkurenz um den buffer.