

## 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++;  
}
```

- 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();
```

- 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.

## 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.
- 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 1: 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 2: 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 3: 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 }

```

## 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 aufzuzeichnen, so dass 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  $t = 0$ .

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

Tabelle 1: Prozesse

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

- 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?

## Aufgabe 3.4: Priority Inversion

(Selbststudium)

In 1997 ist der Mars Pathfinder auf dem Mars gelandet.

Der hatte einen gemeinsamen 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.

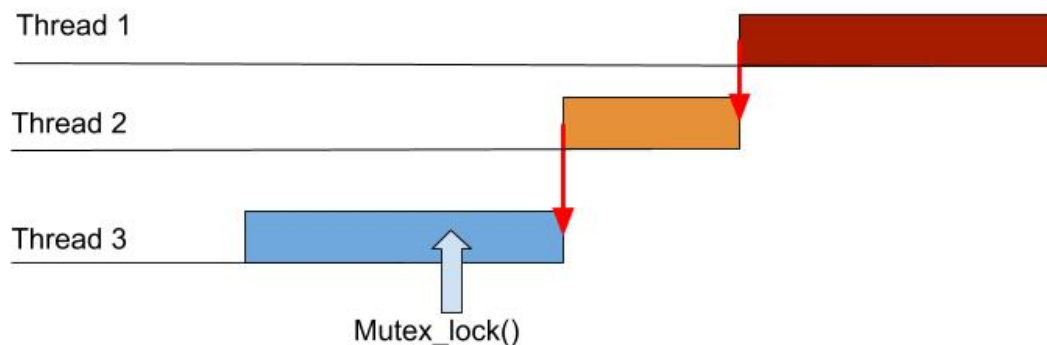
Drei Threads waren Periodisch (in Reinforme von Prioritäten):

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.



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

### 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.

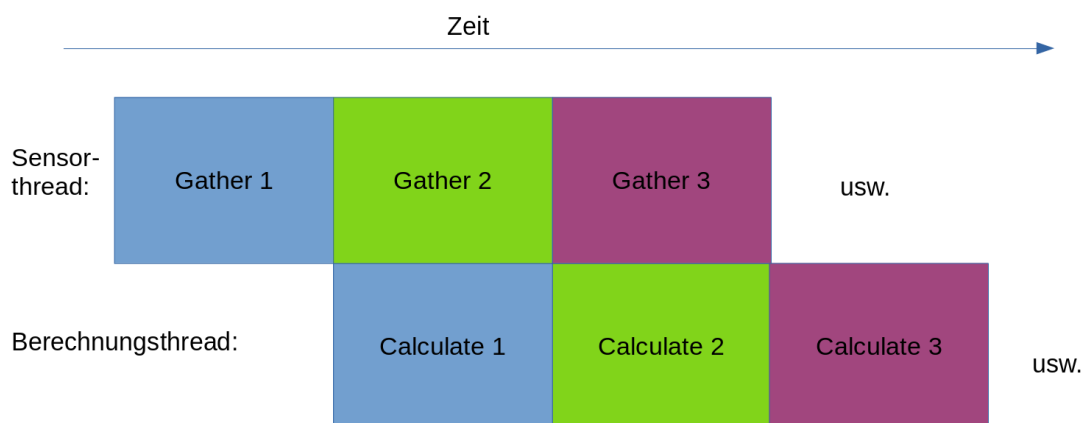
Die Funktion `calculate_forecast(char *buffer)` wertet die übergebenen Messdaten aus und schickt das Ergebnis an gewünschte Geräte im Heimnetzwerk.

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.
- b) Was sind *Spurious Wakeups*? Und wie kann man sicher stellen, dass die eigentliche Bedingung erfüllt ist, auch in fall von *Spurious wakeup*?
- 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.
- d) Welche Arten der expliziten Prozess- /Threadinteraktion gibt es? Um welche Art der Interaktion handelt es sich hier?