Programmierung

Zeitverwaltung

Beschreibung

Die Funktion `gettimeofday()` liefert die Zeit in Sekunden und Mikrosekunden über die folgende (vereinfacht dargestellte) Datenstruktur (siehe Buch):

```
struct timeval {
   unsigned int tv_sec;
   unsigned int tv_usec;
};
```

Mit Hilfe der Funktion soll eine Differenzzeitmessung durchgeführt werden, die auch für größere Zeitabstände ausgelegt ist. Das Ergebnis soll wieder in Form der Datenstruktur `struct timeval` vorliegen (also in Sekunden und in Mikrosekunden).

Hinweis: Der Datentyp `unsigned int` ist 32 Bit breit und kann damit einen Wert von 0 bis 4294967295 annehmen.

Aufgaben

1. Welcher Zeitbereich lässt sich in einer derartigen Datenstruktur unterbringen?

> A: `tv_sec` beschreibt die vergangene UTC Zeit seit dem 1.1.1970 in Sekunden. Da der Datentyp ein `unsigned int` ist, erlaubt dieser auf einer 32-Bit Maschine einen Wertebereich von 0 bis 4294967295. Damit sind also 4294967296 Sekunden möglich.
> Generell ist `tv_sec` vom Typen `time_t`, der sich je nach Implementierung unterscheiden kann. Zusätzlich lassen sich 0 bis maximal 999.999 Mikrosekunden sichern - 1.000.000 Mikrosekunden dürften nicht möglich sein, da die Zeit normiert ist: 1.000.000 Mikrosekunden ergeben nämlich wieder 1 Sekunde, die hochgezählt wird. Und ab 2^32 Sekunden gibt es bei 1.000.000 Mikrosekunden einen Überlauf.
> Selbiges gilt für 64-Bit Maschinen, die 2^64 Sekunden und ebenfalls 999.999 Mikrosekunden sichern können.

> 32-Bit Maschinen mit einem `unsigned int` als `tv_sec` decken einen Zeitbereich vom 01.01.1970 bis zum 7.02.2106, 06:28:15 ab - ca. 136 Jahre. 64-Bit Maschinen decken einen Zeitbereich von 500 Milliarden Jahren ab.

2. In der Programmierpraxis kann die per `gettimeofday()` zurückgelieferte Zeit in eine einzelne Variable (Typ Integer) umgerechnet werden. Welchen Vorteil bringt das mit sich? Was sind die Nachteile?

> A: Die Funktion `gettimeofday()` liest die Zeit in ein `struct timeval`. Dieses enthält die Zeitangaben in Sekunden `tv_sec` und Mikrosekunden `tv_usec` - in den meisten Systemen laut http://rabbit.eng.miami.edu/info/functions/time.html#time_t eher ungenau und außer

> Acht gelassen. `tv_sec` ist im Format `time_t`. In Unix und POSIX-Systemen wird dieses als `signed int` implementiert.

Vorteile:

2.1

- weniger Speicherplatz benötigt, da nur 4 Byte respektive 8 Byte in 64-Bit Systemen
- einheitliches Format, da die Angabe in Sekunden seit 1.1.1970 weit verbreitet ist
- dadurch einfache Konvertierung in andere Formate

Nachteile:

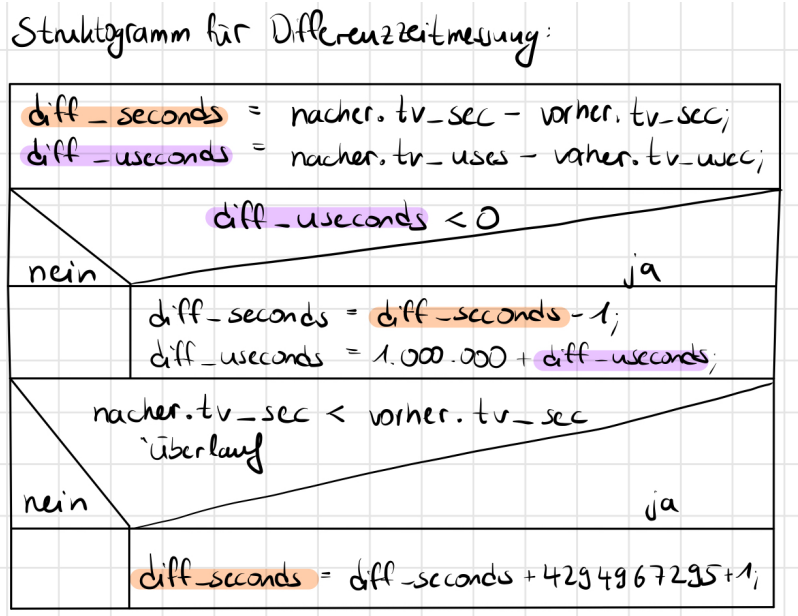
- Überlauf, weil der Datentyp `signed int` bei einem 32-Bit System nur einen Wertebereich von `[-2,147,483,648 to 2,147,483,647]` hat.
- Überlauf findet am Montag, dem 18. Januar 2038, 22:14:08 statt.

3. Erläutern Sie das Prinzip der Differenzzeitmessung. (Buch S. 117)

> A: Um eine Zeitdauer bestimmten zu können, muss häufig die Anwendung einer Differenzzeitmessung erfolgen. Hierbei wird ein Zeitstempel vor und nach der jeweiligen Aktion gemacht und die Differenz dieser beiden Zeitstempel berechnet, welches als Differenzzeitmessung bezeichnet wird. Quasi: die Differenz/der Unterschied zwischen zwei "Zeiten".
> Differenzzeitmessung = Ende(Zeitstempel) - Anfang(Zeitstempel);

4. Skizzieren Sie in Form eines Struktogramms einen Algorithmus, der die Differenzzeit berechnet und in die Variablen diff_seconds und diff_useconds ablegt. Der Algorithmus soll für alle angegebenen Zeitwerte das jeweils richtige Ergebnis liefern.

> A: Die angegebenen Zeitwerte sind die Werte, die in der unteren Tabelle stehen.



5. Geben Sie für die folgenden drei Paare von Zeitstempeln die jeweilige Differenz in Sekunden und Mikrosekunden an.

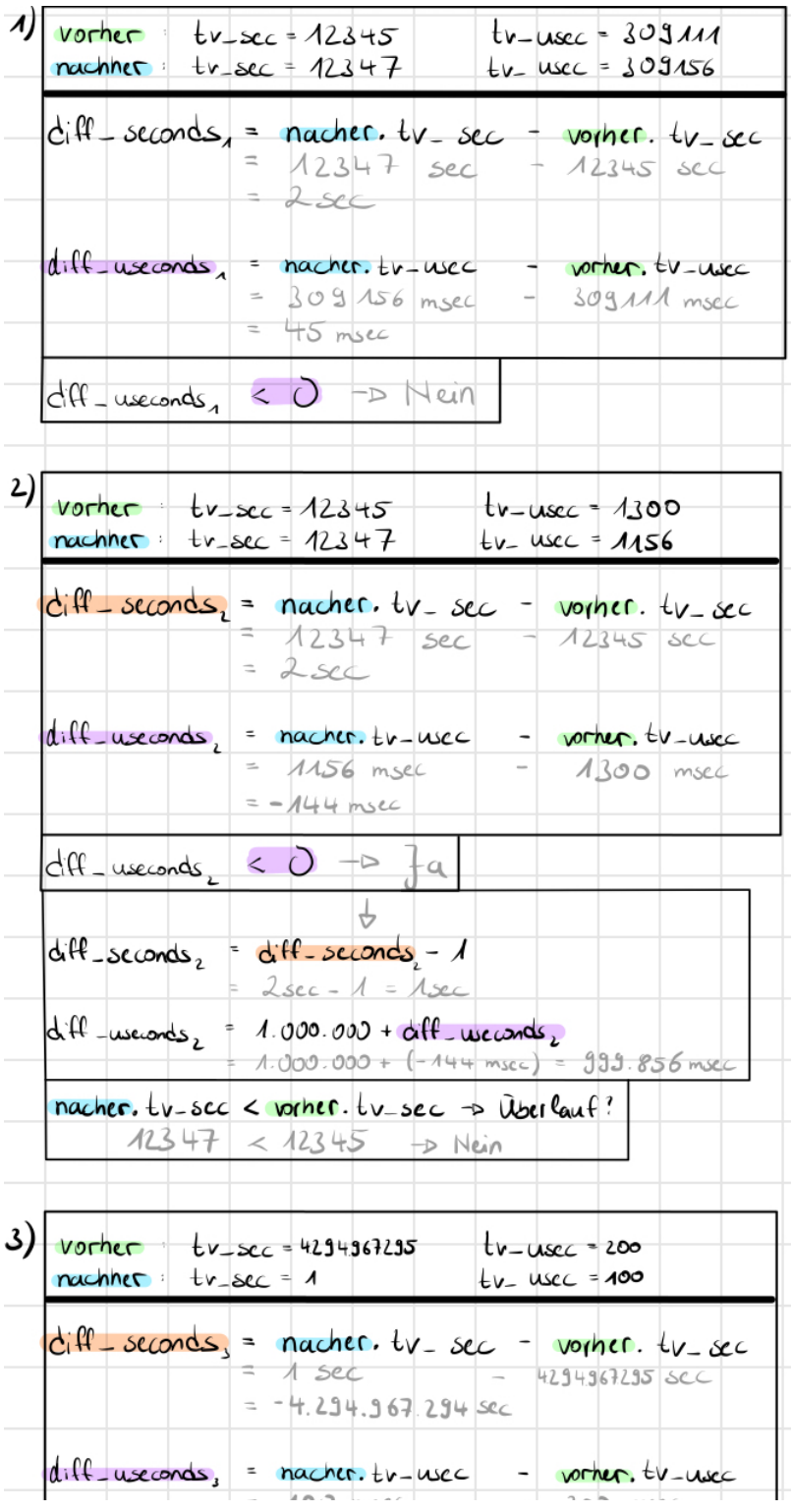
Vorher	tv_sec=12345	tv_usec=309111

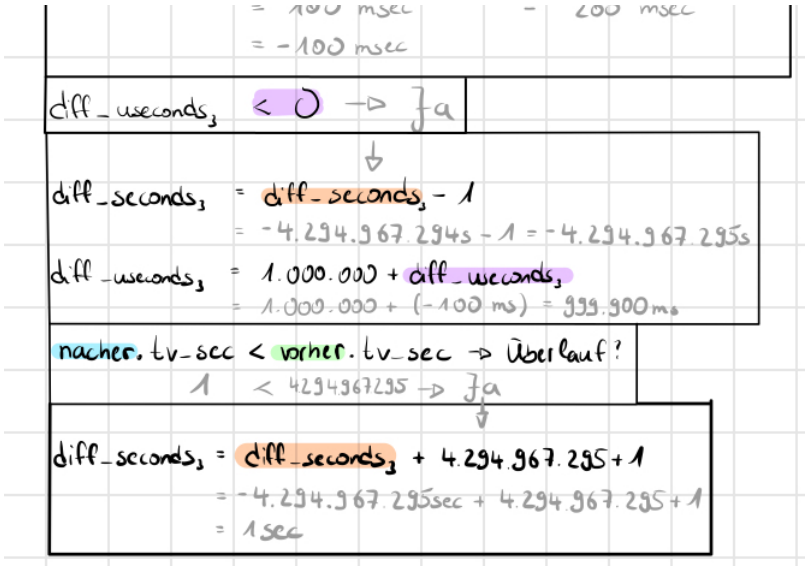
Nachher	tv_sec=12347	tv_usec=309156

Vorher	tv_sec=12345	tv_usec=1300
Nachher	tv_sec=12347	tv_usec=1156

Vorher	tv_sec=4294967295	tv_usec=200
Nachher	tv_sec=1	tv_usec=100

> A: Um eine bessere Übersicht über die jeweiligen Rechnungen zu haben, wurde das Struktogramm in der Form leicht verändert.





Code zur Berechnung der Differenz

```
times = [[(12345, 309111), (12347, 309156)], [(12345, 1300), (12347,
1156)], [(4294967295, 200), (1, 100)]]

for time in times:
    vorher = time[0]
    nachher = time[1]
    diff_seconds = nachher[0] - vorher[0]
    diff_useconds = nachher[1] - vorher[1]

    if diff_useconds < 0:
        diff_seconds = diff_seconds - 1
        diff_useconds = 1000000 + diff_useconds

    if nachher[0] < vorher[0]:  # Überlauf
        diff_seconds = diff_seconds + (2 ** 32)
    print(diff_seconds, diff_useconds)
```

Output

```
2 45
1 999856
1 999900
```

> A: Die Ergebnisse sind hier nochmal aufgezählt

diff_seconds	diff_useconds
2	45
1	999856

diff_seconds	diff_useconds
1	999900

Critical Section - Messwerterfassung

Beschreibung

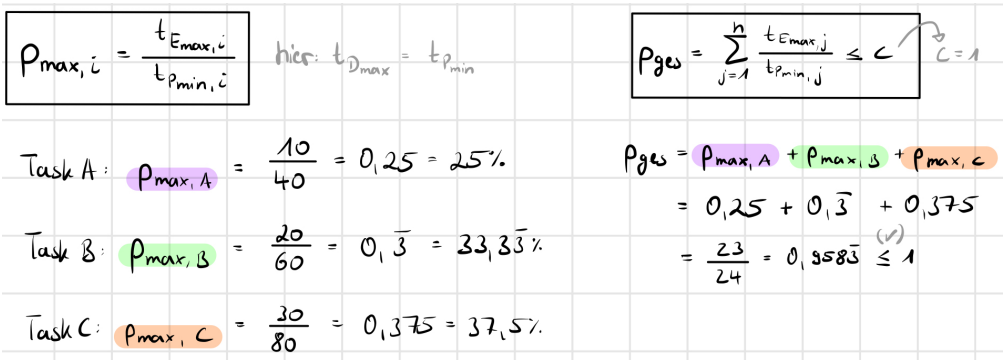
Ein Realzeitrechner führt eine Meßwerterfassung durch. Für drei unterschiedliche Eingangskanäle (Temperatur, Druck, Durchfluss) gibt es drei Tasks, die jeweils für die Steuerung eines Kanals zuständig sind. Da mit dem Messwert auch die Dauer der Meßwertaufnahme zu bestimmen ist, gibt es genau einen Zähler, der zu Beginn der Task gestartet wird und nach dem Ende der Meßwerterfassung der Task mit dem Stoppen des Zählers ausgelesen wird. Die erfaßten Meßwerte werden von den drei Tasks zusammen mit dem Zählerstand in einen gemeinsamen Speicher abgelegt. Der Zugriff auf den Speicher ist für alle Tasks ohne Synchronisation möglich. Die drei Tasks haben die folgenden Verarbeitungs- und maximal zulässigen Reaktionszeiten (in ms):

Task	t_E	t_Dmax
Task A	10	40
Task B	20	60
Task C	30	80

Aufgaben

1. Ist eine Lösung der Aufgabe in Realzeit prinzipiell möglich (Überprüfen der Auslastungsbedingung)? Sie können zur Lösung der Aufgabe davon ausgehen, dass die **Prozesszeit mit der maximal zulässigen Reaktionszeit der jeweiligen Task identisch** ist. Bestimmen Sie dazu die Gesamtauslastung des Realzeitrechners durch die drei Tasks.

   > A: Für diese Aufgabe wird zunächst die Auslastungsbedingung (S. 19) für jeden Task und anschließend die Gesamtauslastung bestimmt. Das Fazit lautet wie folgt: diese Aufgabe ist in einem Realzeitsystem möglich, weil die Auslastung kleiner als 1 ist.



5.1

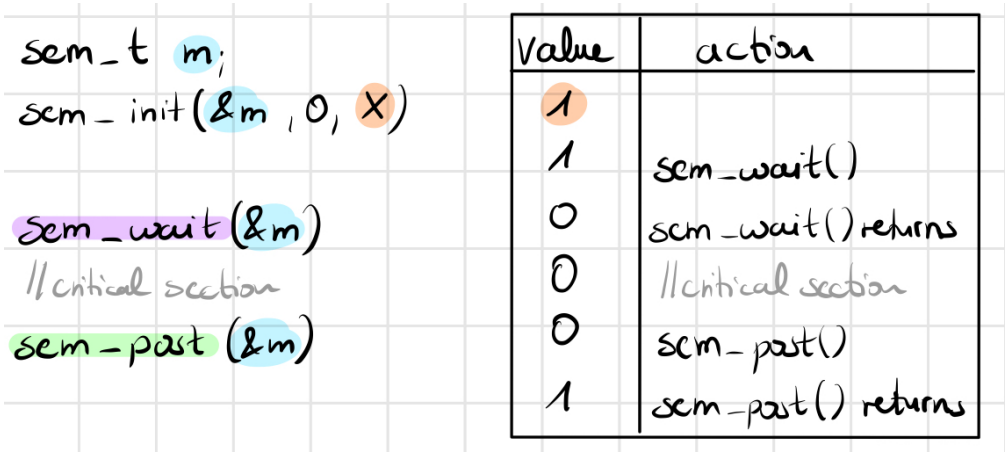
2. Beschreiben Sie kurz die „critical section“ bei der Meßwerterfassung. Wie kann softwaretechnisch sichergestellt werden, dass es bei dieser „critical section“ zu keiner race condition kommt?

   > A: Die Critical Section (= kritischer Abschnitt) ist ein Teil im Code, bei dem es zu einer Race Condition oder zu einem Deadlock kommt, indem beispielsweise zwei oder mehrere Tasks

> **gleichzeitig** auf eine Ressource zugreifen. Es kommt zu einer sogenannten *Verklemmung*. Soll eine Critical Section und die daraus folgende Race Condition umgegangen werden, muss dafür gesorgt werden, dass nie mehr als ein Task den kritischen Bereich betritt. Dies lässt sich beispielsweise durch Semaphoren oder Mutexe sicherstellen. Diese sorgen für Mutual Exclusion! Bei der Semaphore handelt es sich um eine Integer-Variable, die folgendermaßen angewendet wird:

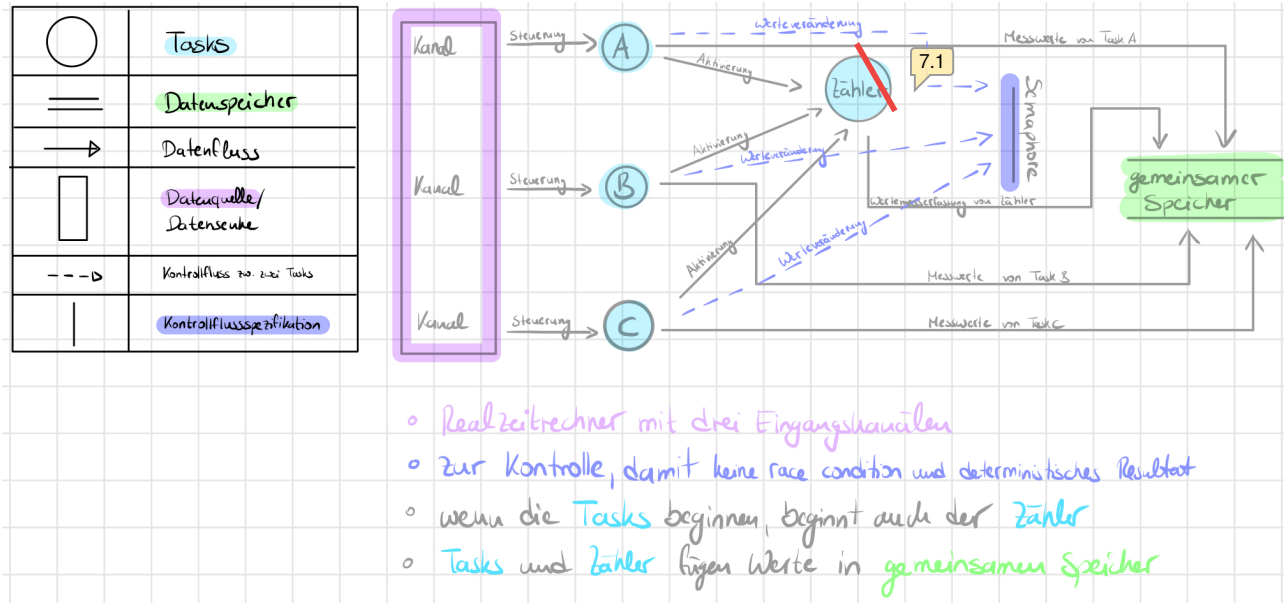
> Die Semaphore muss zu Beginn mit einem selbst gewählten Wert initialisiert werden - meistens ist es die Anzahl an Tasks, die gleichzeitig in den kritischen Bereich dürfen. Die Funktion `sem_wait()` dekrementiert, wenn ein Thread den kritischen Bereich betritt und `sem_post()` inkrementiert, wenn ein Task den kritischen Bereich verlässt. Sobald die Integer-Variable den Wert 0 >= erreicht, ist der Bereich für alle anderen Tasks, die keine Semaphore haben, gesperrt (Mutual Exclusion). Sobald ein Task (s)ein Semaphor freigibt, wird der wartende Task mit der höchsten Priorität geweckt. Durch die Semaphoren wird erzielt, dass der Output einer Task erwünscht und deterministisch ist -> die Wettlaufsituation wird damit umgangen.

> Das folgende Bild zeigt ein Beispiel wie der Aufbau und die Anwendung eines Semaphors erfolgt.



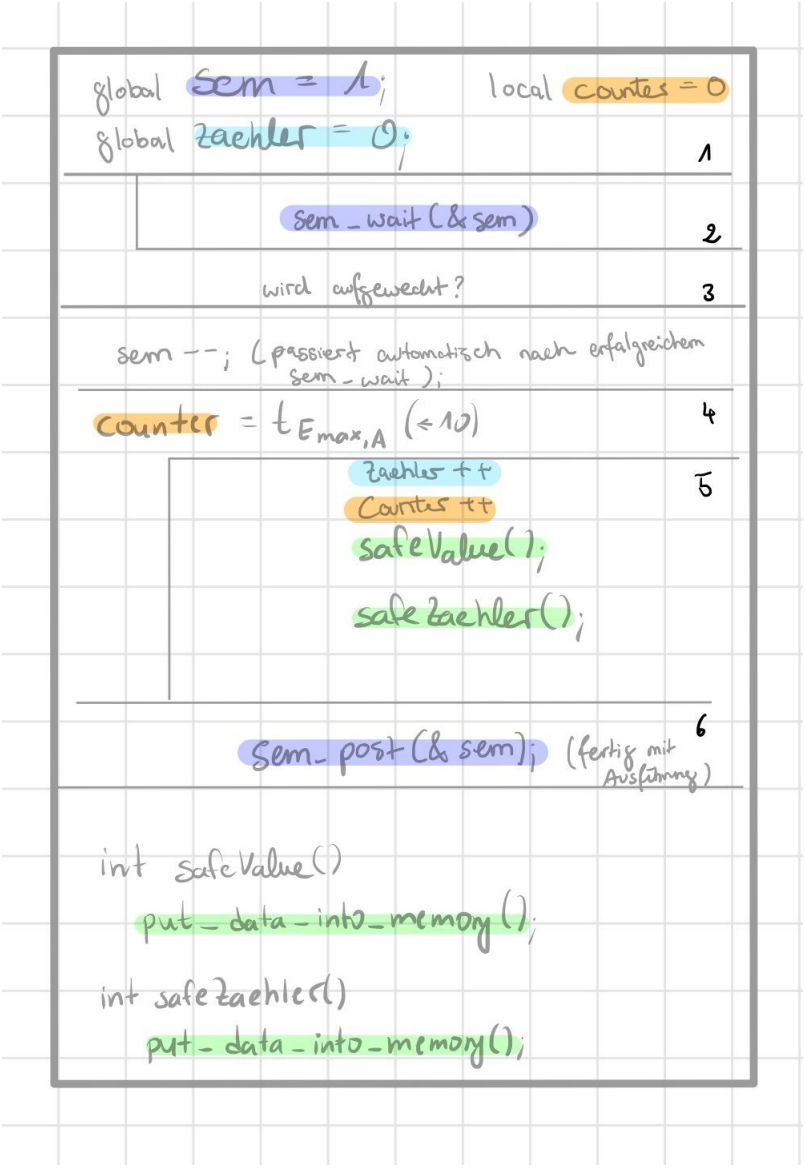
3. Erstellen Sie ein DFD der Konstellation. Ergänzen Sie das DFD durch die Kontrollflüsse.

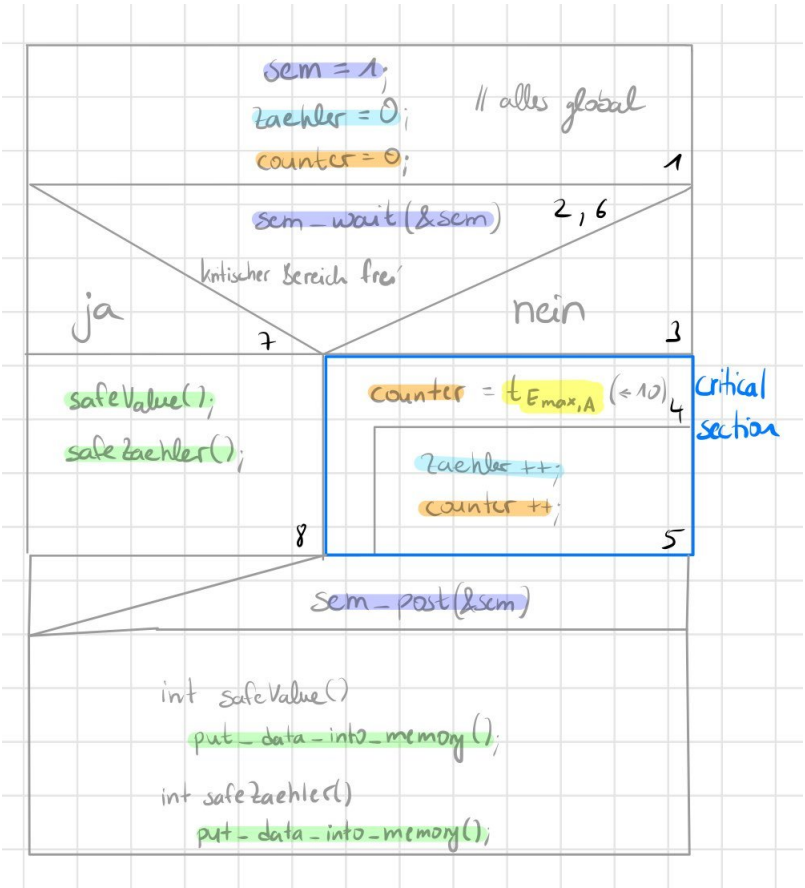
> A: Ein `DFD` (S. 202) (=Data-Flow-Diagram) beschreibt die Prozesse, die in einem System zur Übertragung von Daten von der Eingabe bis zur Dateiablage und Berichterstattung ablaufen. Aus der Aufgabenstellung lernen wir, dass wir einen Realzeitrechner mit drei Eingangskanälen haben, wovon jeder durch eine Task kontrolliert wird. Wenn nun die Daten durch eine Task ausgelesen werden, startet gleichzeitig ein Zähler. Dieser ist für die Dauer der Messwertaufnahme zuständig. Dieser Kontrollfluss muss jedoch durch eine Semaphore kontrolliert werden, damit keine Race Condition passiert. Am Ende werden die ausgelesenen Daten durch die Task und der Wert für den Zähler an einem gemeinsamen Speicher abgelegt. Dies passiert synchron.



4. Skizzieren Sie den Code einer der drei Tasks in Form eines Struktogramms.

> A: Die Struktogramme dienen zur strukturierten Darstellungen von unseren Codesequenzen. (S. 205). Das folgende Bild zeigt unsere Lösung. Dabei wurde sich an dem DFD orientiert und versucht im Code wiederzugeben. Der Value `counter` ist hierbei eine Hilfsvariable um zu kontrollieren, ob Task A vollständig ausgeführt wurde. Zu Beginn gibt es eine allgemeine Initialisierung für alle verwendeten Werte. Sobald Task A läuft, verändern sich diese Werte (decrement und increment). Wenn die Execution-Zeit nicht erreicht ist, läuft Task A weiter und die Werte `counter` und `zaehler` verändern sich weiterhin. Sobald Task A vollständig ausgeführt wurde, werden die Messwerte von Task A und die Wertmesserfassung von dem Zähler im gemeinsamen Speicher abgelegt. Auch resettet sich die Semaphore, um anderen Tasks zu signalisieren, dass der kritische Bereich frei und diese nun laufen können.


global sem = 1;
local counter = 0
global zaehler = 0;
1
sem_wait(&sem)
2
wird aufgeweckt?
3
sem--; (passiert automatisch nach erfolgreichem sem_wait);
counter = t Emax,A (<10)
4
zaehler++
counter++
safeValue();
safeZaehler();
5
6
sem_post(&sem); (fertig mit Ausführung)
int safeValue()
put_data_into_memory();
int safeZaehler()
put_data_into_memory();

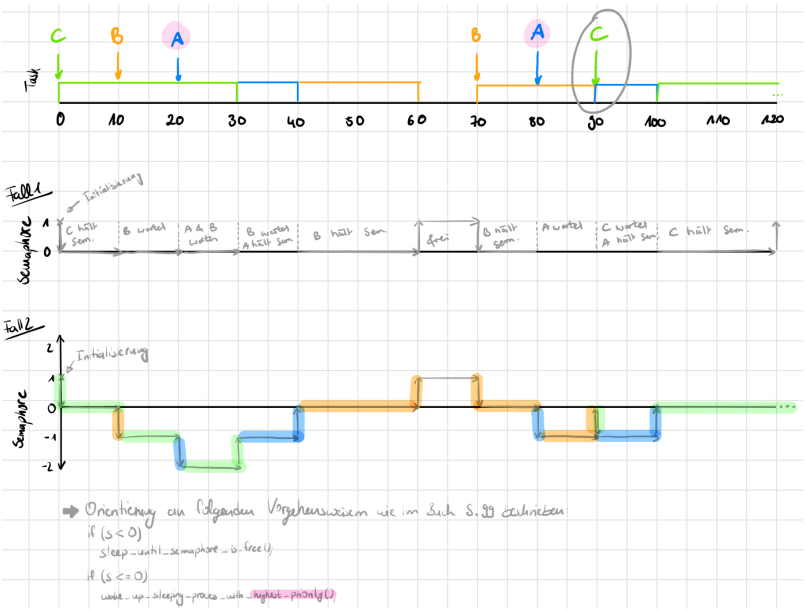

sem = 1;
zaehler = 0;
counter = 0;
// alles global
1
sem_wait(&sem)
2, 6
kritischer Bereich frei
ja
nein
7
3
safeValue();
safeZaehler();
counter = t Emax,A (<10)
4
critical section
zaehler++;
counter++;
8
5
sem_post(&sem)
int safeValue()
put_data_into_memory();
int safeZaehler()
put_data_into_memory();

5. Geben Sie den drei Tasks Prioritäten.

> A: Allgemeine Regel für Prioritätenvergabe: je kürzer die Prozess- und Ausführungszeit eines Tasks ist, desto höher ist dessen Priorität.
> Task A -> Prio 1
> Task B -> Prio 2
> Task C -> Prio 3

6. Skizzieren Sie die Rechnerkernbelegung und den Wert des Semaphor im Zeitraum von 0 bis 120 ms. Gehen Sie davon aus, dass die einzelnen Tasks zu den folgenden Zeitpunkten **lauffähig** werden:

```
Task A: 20ms, 80ms
Task B: 10ms, 70ms
Task C: 0ms, 90ms
```



> A: Wir haben zwei Darstellungen für die Semaphore. Eine nur mit Werten größer gleich 0 und eine, die Werte auch unter 0 erlaubt. Laut der man page darf der Wert einer Semaphore nicht in den negativen Bereich gehen, also < 0. Dies zeigt der erste Fall. In dem Fach Betriebssysteme haben wir jedoch Werte im negativen Bereich verwendet. Diese dienten als Hilfsmittel um zu verstehen, ob die Semaphore den kritischen Bereich reserviert oder für eine schlafende Task freigibt. Dies präsentiert Darstellung 2. Aufgrunddessen verwenden wir zwei Darstellungen.

7. Geben Sie die maximale Reaktionszeit tRmax,A der Task A für den Fall an, dass kein Semaphor notwendig wäre (ohne Synchronisation) **UND** für den hier beschriebenen Fall mit dem Semaphor (Worst Case).

> A: Bei dem Worst-Case ohne Semaphor: 10 ms.

> A: Bei dem Worst-Case mit Semaphore: Task A wird solange blockiert, wie Task C das Semaphor in Anspruch nimmt - also die gesamte Ausführungszeit von Task C (30 ms). Ab diesem Zeitpunkt wird A ausgeführt, auch wenn B zuerst lauffähig wurde, da, nach der Freigabe des Semaphors, die Task, mit der höchsten Priorität, aufgeweckt wird. Task A hat dann eine maximale Reaktionszeit von 20 ms (40 ms [max. Reaktionszeit, wenn A bei 0 ms lauffähig wird] - 20 ms [Zeitpunkt ab dem A lauffähig wird]).

> Wie bereits beschrieben, muss mit einberechnet werden, dass Task A erst ab 20 ms lauffähig wird und die maximale Reaktionszeit erst ab diesem Zeitpunkt berechnet werden sollte.

Code zur Errechnung der t_Rmax(A) ohne Semaphor

> Zeitpunkt der Lauffähigkeit der Tasks wird hier ignoriert.

```
# mit Semaphor
# taskset = [(40, 10), (80, 30)]
# t_Emax = 30

# ohne Semaphor
taskset = [(40, 10)]
t_Emax = 10

t = t_Emax

for _ in range(10):
    print(t)
    _t = 0
    for task in taskset:
        _t += math.ceil(t / task[0]) * task[1]
    t = _t
```

Output ohne Semaphor

```
10
10
...
```

Output mit Semaphor

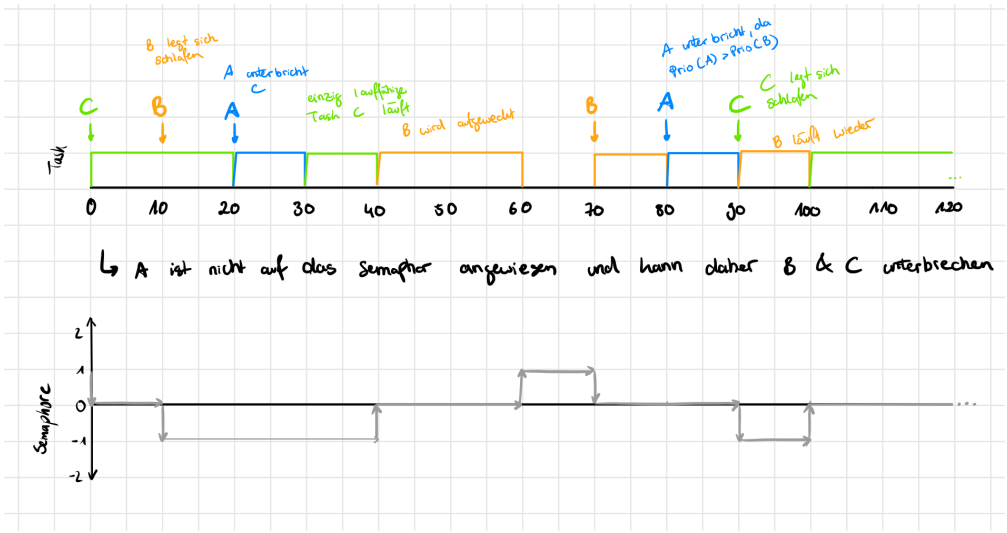
```
30
40
40
...
```

8. Um die maximale Reaktionszeit (Antwortzeit) der Task A zu verringern, wird ein zweiter Zähler zur Verfügung gestellt. Wie wird jetzt das Semaphor verwendet?

> A: Eine Task würde damit ihren eigenen Zähler erhalten. Angenommen Task A würde den zweiten Zähler erhalten, so würde Folgendes gelten: A: Task A erhält seinen eigenen Zähler, während Task B und Task C sich einen Zähler teilen - es besteht also noch ein kritischer Bereich. Das Semaphor dient dann dem Schutz des kritischen Bereichs, den sich nur noch die Tasks B und C teilen. Task A benötigt keinen Semaphor, da es das einzige Task ist, das auf den Zähler zugreift.

9. Zeichnen Sie für den Fall, dass zwei Zähler zur Verfügung stehen, die Rechnerkernbelegung und den Wert des Semaphor.

> A: In diesem Fall gehen wir wie in Aufgabe 6 davon aus, dass der Wert des Semaphors negativ werden kann (wie im Buch: **s < 0**). Weiterhin gehen wir davon aus, dass A den zweiten Zähler erhält, da diese Kombination zur besten maximalen Reaktionszeit für Task A führen würde.



10. In welchem Fall wird durch die beschriebene Maßnahme die maximale Reaktionszeit der Task A tatsächlich verbessert?

> A: Die maximale Reaktionszeit von Task A wird nur verbessert, wenn der zweite Zähler für A alleine nutzbar ist. Sobald ein anderes Task {B, C} den Zähler erhält, müsste sich A wieder den Zähler teilen und wäre vom Wert des Semaphors abhängig. Kurz: A könnte dann nicht mehr unterbrechen.

Weiterführende Links:

- http://rabbit.eng.miami.edu/info/functions/time.html#time_t
- https://linux.die.net/man/2/gettimeofday

Index der Kommentare