SHARED MEMORY, FORK & SEMAPHOR

1. SHARED-MEMORY-INTERFACE FÜR *VEHICLE CONTROL*

Wir wollen heute unseren Prozess *Vehicle Control* mit einem Interface zu anderen Prozesse ausstatten. Dieses Interface soll die 5 Sollgrößen (Sollgeschwindigkeit links/rechts, Lenkwinkel und Kameraausrichtung (pan/tilt) für andere Prozesse lesend und schreibend zugreifbar machen. Das Interface wird mit dem Konzept *Shared Memory* realisiert, wobei der exklusive Zugriff mit einer Semaphor sicherzustellen ist.

Für eine fundierte Einführung in die nebenläufige Programmierung, insbesondere unter dem Betriebssystem Linux, ist das Buch von Carsten Vogt sehr zu empfehlen:

> Carsten Vogt: *Exploring Raspberry Pi: interfacing to the real world with embedded Linux*, Wiley, 2016

Kopieren Sie sich das PDF-Dokument des Buches von der Hochschulbibliothek auf Ihren Hostrechner (unter Ihrem Account).

Die Beispielprogramme zu diesem Buch finden Sie in einer Zip-Datei auf Moodle. Kopieren Sie sich die Beispielprogramme – falls noch nicht vorhanden[1] – auf den Raspberry Pi Ihres Fahrzeugs:

`/home/pi/Vogt_Beispielprogramme`

a) Shared-Memory-Interface und 2 Kindprozesse programmieren

Erstellen Sie ein **Shared-Memory-Interface** mit folgender Struktur:

```
struct VC_IFC {
     float speed_l;
     float speed_r;
     float steeringAngle;
     float panAngle;
     float tiltAngle;
     int cmd;
};
```

Als Vorbild kann Ihnen das Programm 4.1 von Seite 173 im Buch von C. Vogt dienen.

Nehmen Sie Ihr Programm *Vehicle Control* vom letzten Aufgabenblatt als Ausgangsbasis und erweitern Sie es so, dass es zwei Kindprozesse startet und verwaltet.

[1] Falls Sie die SD-Karte Ihres Fahrzeugs nicht komplett überschrieben haben, sollten sich die Beispielprogramme bereits im Home-Verzeichnis des Nutzers **`pi`** befinden.

Bevor in die Kind-Prozesse verzweigt wird, legt der Hauptprozess einen *Shared-Memory*-Bereich des obigen Typs an.

Der erste Kindprozess (im Folgenden **Zyklus-Prozess** genannt) öffnet den *Shared Memory*-Bereich **`VC_IFC`** und initialisiert alle Größen mit **`0.0f`**. Dann setzt er im *Shared-Memory*-Bereich **`cmd`** auf **`1`**. Anschließend geht er in eine **`while`**-Schleife und bleibt darin, solange **`cmd==1`** ist. In der Schleife liest er die 2 Motorgeschwindigkeiten und die 3 Sollwinkel von **`VC_IFC`** aus und übergibt Sie den Treiberobjekten für die DC-Motoren und Servos. Anschließend schläft der Prozess 40 ms lang. Wird vom einem anderen Prozess **`cmd==0`** gesetzt, geht der Zyklus-Prozess aus der Schleife heraus und beendet sich.

Der zweite Prozess (wir nennen ihn **Event-Prozess**) öffnet ebenfalls den *Shared-Memory*-Bereich **`VC_IFC`** und schreibt zu Testzwecken im Sekundentakt Sollgrößen in **`VC_IFC`** hinein. Nach seiner Testsequenz setzt er die Variable **`cmd`** von **`VC_IFC`** auf **`0`** und beendet sich.

Der Elternprozess wartet auf die Beendigung seiner beiden Kindprozesse. Anschließend löscht der den *Shared-Memory*-Bereich, setzt den PWM-Treiberbaustein schlafend und beendet sich dann ebenfalls.

b) Erzwingen des exklusiven Zugriffs mit einem Semaphor

Wir haben bisher noch nicht den exklusiven Zugriff auf den Shared-Memory-Bereich erzwungen. Das könnte später zu Problemen führen, wenn z. B. gleichzeitig in diesen Bereich gelesen und geschrieben wird. Stellen Sie deshalb den exklusiven Zugriff durch einen Semaphor sicher.

Sie können bei der Lösung der Teilaufgabe vorgehen, wie es auf Seite 129 des Buches von C. Vogt beschrieben ist (siehe auch Beispielprogramme 3.2 und 3.3). Gehen Sie davon aus, dass die Nutzer von **`VC_IFC`** keine Kindprozesse von *Vehicle Control* sein müssen. Es muss demzufolge auch fremden Prozessen möglich sein, auf die Semaphor und seine Semaphorgruppe zuzugreifen. Wir definieren deshalb einen Schlüssel für die Semaphorgruppe mit 4710. Das Analoge gilt auch für den Shared-Memory-Bereich von Teilaufgabe a). Legen Sie deshalb nachträglich den Schlüssel des Shared-Memory-Bereiches auf 4711 fest.[2]

Testen Sie Ihre Implementierung dadurch, in dem Sie an einer Stelle in Ihrem Programm absichtlich den Semaphor nicht mehr freigeben und so einen Deadlock erzeugen.

c) Messen der Zykluszeit von *Vehicle Control* mit dem Oszilloskop

Erweitern Sie den Zyklus-Prozess von Teilaufgabe a) um einen weiteren GPIO-Zugriff. Der GPIO23 (am Pin 16) soll in jedem Zyklus zwischen den Zuständen **`0`** und **`1`** hin- und hergeschaltet werden. Messen Sie Sie anschließend mit dem Oszilloskop die erzielte Zykluszeit in ms. Kommentieren Sie im Zyklus-Prozess vorübergehend den **`sleep`**-Befehl aus

[2] Es bietet sich an, die beiden IDs im dem Headerfile als Makro zu definieren, wo Sie auch die Struktur **`VC_IFC`** deklarieren.

und messen Sie die maximal erzielbare Zykluszeit. Wie groß ist sie? Wiederholen Sie den letzten Schritt mit der Option -O3 (Optimierung) bei der Übersetzung des Programms.

Meilenstein 1: Die Zykluszeit des Zyklus-Prozesses kann im Oszilloskop gemessen werden, während die Testprozedur des Event-Prozesses abläuft.

2. ÜBUNGSFRAGEN

a) Betrachten Sie die folgende Lösung eines wechselseitigen Ausschlussproblems, bei dem zwei Prozesse, P0 und P1, beteiligt sind:

Angenommen, die Variable **`turn`** wird mit 0 initialisiert. Der Code von Prozess P0 lautet:

```
/* weiterer Code */
while (turn !=0) {} /*unternimm nichts und warte */
Critical Section /* ... */
turn = 0;
/* weiterer Code */
```

Ersetzen Sie in obigem Code für Prozess P1 die 0 durch 1. Stellen Sie fest, ob die Lösung alle erforderlichen Bedingungen für eine Lösung des wechselseitigen Ausschlusses beinhaltet.

b) Wie könnte ein Betriebssystem, das Interrupts ausschalten kann, Semaphore realisieren?

c) Ein Schnellimbiss hat vier Arten von Angestellten:

(1) Bedienungen, die die Bestellungen der Kunden aufnehmen,
(2) Köche, die das Essen zubereiten,
(3) Einpacker, die das Essen in Tüten einpacken, und
(4) Kassierer, die den Kunden die Tüten geben und das Geld entgegennehmen.

Jeder Angestellte kann als ein kommunizierender, sequenzieller Prozess betrachtet werden. Welche Art von Interprozesskommunikation benutzen sie? Setzen Sie dieses Modell in Beziehung zu Prozessen in UNIX.

d) Round-Robin-Scheduler verwalten in der Regel eine Liste aller lauffähigen Prozesse, wobei jeder Prozess genau einmal in der Liste vorkommt. Was würde passieren, wenn ein Prozess zweimal vorkäme? Können Sie sich einen Grund vorstellen, so etwas zuzulassen?

e) Kann durch das Analysieren des Quellcodes ein Maß dafür angegeben werden, ob der Prozess voraussichtlich rechenintensiv oder E/A-intensiv ist? Wie lässt sich das zur Laufzeit bestimmen?