Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Dipl. Inf. K. Baukus · Dipl. Inf. K. Stahl

# Informatik III

Wintersemester 2001/02 **Serie 13** 7. Februar 2002

<u>Thema :</u> **Xinu-Rechnerübungen**

**Ausgabetermin: 7. Februar 2002**

**Abgabe: Vorführung bei Anwesenheit**

### Aufgabe 1: [Aufwärmen]

Xinu ist *paßwortgeschützt!* Finden Sie das Paßwort heraus! Programmieren Sie die Beispielprogramme aus [Com83], Kapitel 1!

### Aufgabe 2: [Prozeßtabelle]

Ziel der Aufgabe ist es, die zentrale Datenstruktur zur Prozeßverwaltung, die *Prozeßtabelle*, zu untersuchen (siehe auch [Com83], Kapitel 4). Zur Erinnerung: die Struktur der Prozeßtabelle ist in `proc.h` definiert.

Schreiben Sie eine Prozedur, um die Prozeßtabelle *auszudrucken.* Die Prozedur soll folgende Daten ausgeben:

- Prozeßidentifier
- Prozeßname
- Zustand (*current, ready,* ...)
- Priorität
- Semaphore, falls der Prozeß wartet
- Message, falls der Prozeß eine Message bekommen hat
- Stackpointer
- stack-base und Stack-Länge
- Anfangsadresse des Codes

Ein Vorschlag für die Ausgabe: Arrangieren Sie die Information in einer Tabelle:

```
Id Name  State  P S  Msg   SP Base  Len Code
 0 NULL  READY  0        FF38 FF80 0048 0372
```

**Aufgabe 3: [Leser und Schreiber]**

Programmieren Sie unter Xinu eine Lösung des Problems der *Leser und Schreiber.* Wie das Produzenten/Konsumenten-Problem stellt dies eine charakteristische Koordinationsaufgaben dar, typisch beispielsweise für verteilten Datenbankzugriff.

Das Problem sieht aus wie folgt. Es gibt zwei Arten von Prozessen, die *Leser* und die *Schreiber*, die auf gemeinsame Daten zugreifen. Wie der Name andeutet, die einen nur lesend und die anderen nur schreibend. Was die Zugriffskontrolle betrifft, besteht sie zwischen Lesen und Schreiben darin, daß zu einem gegeben Zeitpunkt beliebig viele Leser erlaubt sind, Schreiber jedoch, um Inkonsistenzen zu vermeiden, exklusiven Zugriff benötigen (d.h., während ein Schreiber Zugriff hat, darf weder gelesen noch geschrieben werden).

Programmieren Sie eine Lösung des Leser-Schreiber-Problems, d.h., programmieren Sie Leser und Schreiber und testen Sie Ihre Lösung mit mehreren Lese- und Schreibprozessen. Wie geht Ihre Lösung mit dem Problem des *Verhungerns* um?

**Aufgabe 4: [Rendez-Vous]**

Wir haben in den Übungen besprochen, wie man Xinu um die Möglichkeiten der *Rendez-Vous-Kommunikation* erweitern kann (siehe Aufgabe 1 auf Übungszettel 7 vom 14. Dezember). Setzen Sie ihre Lösung nun in Xinu um, d.h., implementieren Sie die zwei Systemaufrufe:

- `SYSCALL sendrv (pid, msg)` und
- `SYSCALL receiverv(pid)`

Testen Sie Ihre Lösung anhand anhand von (mindestens) zwei Prozessen, die mittels Rendez-Vous kommunizieren. Was passiert, wenn sich ein Prozeß selbst eine Nachricht schickt?

**Aufgabe 5: [Simultane Semaphoren]**

In der Vorlesung wurden die Verwendung und Implementierung von Semaphoren ausgiebig besprochen. Wir haben kurz auch die *Beschränkungen* angesprochen, die mit Semaphoren verbunden sind. Auf wenn Semaphoren verklemmungsfreien Zugriff auf *einen* kritischen Abschnitt bzw. eine Ressource in einfacher Weise lösen, bleibt die Gefahr bei gleichzeitigen Zugriff auf *mehrere* Ressourcen bestehe. Ein Beispiel, was wir in diesem Zusammenhang kennengelernt hatten, war das Problem der *speisenden Philosophen,* bei der *zwei* Gabeln also grünes Licht von zwei Semaphoren notwendig ist.

Zur Unterstützung derartiger Probleme kann das Betriebssystem *simultane Semaphoren* bereitstellen, d.h., einen Systemaufruf `simultaneous_wait`, der als Argument nicht eine Semaphore, sondern eine *variable Anzahl* von Semaphoren bekommt:[1]

[1]Das erste Argument $N$ gibt die Anzahl der Semaphoren an.

```
swait(N, sem1, sem2, sem3, ... ,  semN)
```

Die Semantik des Aufrufs ist eine naheliegende Verallgemeinerung des gewöhnlichen `wait`-Aufrufs. Der aufrufende Prozeß kann nur dann fortfahren, falls *alle* Semaphoren grünes Licht geben und zwar *simultan.* Anders ausgedruckt, es reicht nicht, daß der aufrufende Prozeß erfolgreich `wait` an allen aufgeführten Semaphoren in einer bestimmten Reihenfolge aufrufen kann. Er kann nur dann fortfahren, wenn er dies *gleichzeitig* alle Semaphore passieren kann, ansonsten hätte man das Problem der Verklemmung an mehreren Semaphoren nicht gelöst. Insbesondere heißt das, die Ordnung der Semaphor-Argumente spielt keine Rolle.

Die *Aufgabe* bestehe darin, simultane Semaphoren im Spezialfall $N = 2$ zu implementieren. Der neue Systemaufruf sei mit `swait2` bezeichnet und erwartet als Argument genau zwei Semaphoren.

Entsprechend dem simultanen *wait* braucht man natürlich als inverse Operation die gleichzeitige Freigabe (`ssignal`) von Semaphoren, in unserem Fall `ssignal2` für zwei Argumente. Die Semantik soll sein, daß die Zähler der betroffenen Semaphoren nur dann wieder inkrementiert werden, wenn dies für *alle gleichzeitig* möglich ist. Im Falle von *signal* heißt dies, falls der gewöhnliche `signal`-Aufruf an einer der Beteiligten Semaphoren einen Systemfehler produzierte, dürfen die Zähler der anderen Semaphoren *nicht* geändert werden.

Testen Sie Ihre Lösung anhand der *5 Philosophen.*

## Literatur

[Com83] Douglas Comer. *Operating System Design, The Xinu Approach.* Prentice Hall, 1983.