Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Martin Steffen

Informatik III (OS)

Wintersemester 2003/04 — **Mittsemestertest** — 13. Dezember 2003, 10:00–13:00

Aufgaben mit Lösungshinweisen

Termin: 13. Dezember 2003, 10:00–13:00

- Fangen Sie jede Aufgabe auf einem neuen Zettel an (nicht nur auf einer neuen Seite!) und kennzeichnen Sie das Blatt entsprechend!
- Geben sie ihren Namen auf jedem Blatt an.
- Die Verwendung von Unterlagen, einschließlich "dem Comer" ist erlaubt.

Für Aufgaben, die in Xinu eingreifen, sollen auch Xinu-Bezeichner verwendet werden. Wenn Sie beispielsweise die Prozeßtabelle verändern wollen, verwenden Sie im Code den Bezeichner `proctab` etc. Falls die Aufgabe lautet „Ändern Sie das Verhalten der Prozedur, so daß. . . " und Sie die Aufgabe lösen, indem Sie vorhandenen Code teilweise wiederverwenden, müssen die Stellen, an denen ersetzt wird, *eindeutig* zu erkennen sein (zumindest muß Prozedurname mit Argumenten angegeben sein, auch Seitenzahl im Comer). Es ist nicht verlangt, daß Sie irgendwelche `include`-Befehle angeben.

Für "abstrakte" Synchronisationsaufgaben ist es nicht verlangt, Xinu-spezifischen Code zu geben; in der Tat sollten konkrete hardwarenahe Eingriffe ("Interrupts ausschalten") nicht verwendet werden, da sie nicht zum Repertoire des *Benutzers* des Betriebssystems gehören.

In jedem Fall ist neben sinnvoll kommentiertem Code eine *Beschreibung* Ihrer Lösung in einigen Sätzen zu geben.

Viel Erfolg!

Aufgabe 1 (Semaphoren (5 Punkte)) Programmieren Sie einen neuen Systemaufruf

```
signaln(sem,n)
```

welche an der angegebenen Semaphore n Prozesse anstelle eines einzigen zur Ausführung bereit macht. Der Aufruf `signaln(sem,n)` soll effizienter als der n-fache Aufruf von `signal(sem)` sein (Seite 86 [Com83]).

```
.....
/*------------------------------------------------------------
* signaln -- signal a semaphore n times
* cf. p 86 of Comer
*------------------------------------------------------------
*/
SYSCALL signaln(sem,n)
      register int sem;
      int n;
{
  register struct sentry *sptr;
  char ps;

  disable(ps);
  if (isbadsem(sem) || (sptr = &semaph[sem])->sstate==FREE)
    {
      restore(ps);
      return(SYSERR);
    }

  while  (((sptr->semcnt) < 0) && (n > 0))
    {
      ready(getfirst(sptr->sqhead), RESCHNO);
      sptr->semcnt++;
      n--;
    }
  sptr->sencnt+=n;
  resched();
  restore(ps);
  return (OK);
}
```

Aufgabe 2 (Xinu-Prozeßabläufe (6 Punkte)) Nehmen Sie an, es gebe im laufenden Xinu einen einzigen Userprozeß mit Prozeß-id 1 und Prozeßpriorität 5. P_1 wird daher gerade ausgeführt (Zustand "`CURRENT`") und führt folgendes Programm aus:

```
int main()
{
   int p1;
   int p2;

   p1 = getpid();

   p2 =  create(child, 200, getprio(my_pid)+1, "child", 1, p1);
   resume(p2);
   resume(p2);
   return(1);

}

int child(int pid)
```

```
{
  chprio(getpid(), getprio(getpid())-2);
  ''DO_SOME_INTERNAL_CALCULATIONS'';        // no system calls here!
  suspend(pid);
  return(1);
}
```

Beschreiben Sie den Ablauf/die Abläufe in Xinu, d.h., geben Sie die Reihenfolgen der Systemaufrufe an. Wann treten Kontextwechsel auf und wann und wie terminieren (gegebenfalls) die Prozesse? Gehen Sie bei ihrer Erklärung davon aus, daß die Wirkungsweise des Kontextwechsels und von Prozeduraufrufen bekannt ist.[1]

Lösung: Hier eine informelle Beschreibung dessen, was passiert. Danke auch an Malte Wulf, der die Beschreibung (mit-)erstellt hat.

1. der erste Prozeß (pid `p1` kreiert den zweiten (`child` mit pid `p2`. Der neue hat eine *höhere* Priorität. `create` selbst führt nicht zum Kontextwechsel, denn nach den Annahmen, die wir gemacht haben, gibt es nur *einen* Benutzerprozeß zu Beginn, der damit eine höherer Priorität als der Null-Prozeß hat. Der neu erschaffene Prozeß befindet sich nicht auf der ready-Liste —er ist suspendiert— und kann demnach nicht ans Ruder kommen, selbst wenn er eine höhere Priorität hat.

2. Durch das `resume(p2)` wird der neue Prozeß auf die ready-Liste gesetzt. Da er eine höhere Priorität hat, führt das `resched` mit Parameter `RESCHYES` im `resume` dazu, daß sich der erste Prozeß (sagen wir P_1) auf die *ready*-Liste begibt und der neue Prozeß zu laufen beginnt.

3. P_2 beginnt und setzt als erstes qseine Priorität runter sodaß sie danach *geringer* als die von P_1 ist.

   Nun hängt es davon ab, ob man die (fehlerhafte) Version aus [Com83] oder die in den Übungen (Zettel 6) verbesserte Version verwendet. In der verbesserten Version findet ein Kontextwechsel statt. Das muß *nicht* direkt zum Kontextwechsel führen

4. Je nachdem ob die Zeitscheibe abläuft oder nicht, geht in zwei möglichen Alternativen weiter:

   - Kontextwechsel in P_2 *vor* dem `suspend`. Falls man die verbesserte Version verwendet, so tritt dieser Fall auf jeden Fall dadurch ein daß das `chprio` bereits den Kontextwechsel hervorruft.
     (a) Falls ein Kontextwechsel stattfindet[2] dann bedeutet das, daß P_1 ans Ruder kommt, da er die höchste Priorität besitzt. P_2 wird *ready*.
     (b) Wie in der ersten Alternative auch, macht P_1 weiter wo er stehengeblieben war, d.h., letzten Endes ruft er das *zweite resume* auf.
     (c) Da P_2 nicht suspendiert ist, kehrt der Aufruf mit einem Systemfehler zurück.

[1] Konkret: Es ist *nicht verlangt* beispielsweise den Code des Kontextwechsels zeilenweise durchzugehen ("dann wird Register r_5 in Speicherzelle ... gespeichert ..."). Dasselbe gilt für den zeilenweisen Machinencode von `csv` und `cret`.

[2] Als "Tipp" habe ich extra `DO-SOME-INTERNAL-CALCULATIONS` hingeschrieben.

(d) da der Systemfehler aber nichts anderes ist als eine Konstante, und er schlampigerweise nicht abgefangen ist, führt er nicht direkt zum Abbruch, sondern P_1 erreicht sein letztes `return(1)` und terminiert. Genauer führt dies dazu, daß über `userret kill(p1)` aufgerufen wird, welches ein erneutes `resched` bewirkt.

(e) Damit kommt P_2 als nun der Prozeß mit der höchsten Priorität dran — der null-Prozeß ist niederpriorer — und macht weiter wo er aufgehört hatte. D.h., er beendet eventuell ausstehende interne Aktionen und

(f) führt dann sein `suspend` auf an den nicht mehr existenten `pid1`; wiederum wird die Fehlermeldung ignoriert wird.

(g) Dann terminiert P_2 auf die selbe Art und Weise wie P_1, und am Ende gibt es nicht eine kurze Info, daß alle Prozesse fertig sind.

(a) Wenn P_2 am Ruder bleibt, führt er `suspend` aus, damit kommt P_2 wieder an die Reihe. Der macht weiter wo er stehengeblieben war, d.h., er kehrt aus seinem *ersten* `resume`-Aufruf zurück, wenn man es genau nimmt. Wie auch immer: als nächsten macht er das *zweite resume!* Diesmal jedoch ist die Priorität von P_2 niedriger, P_1 bleibt damit am Ruder, und terminiert.

(b) damit ist P_2 dran, und terminiert auch

Bemerkung: es ist ein klein wenig unschön gewesen, daß die Aufgabe von `chprio` abhängt.

Aufgabe 3 (Produzenten/Konsumenten mit binären Semaphoren (4 Punkte))
Wir haben in den Übungen über das Problem der *Produzenten und der Konsumenten* gesprochen, und eine der Aufgaben bestand darin, sie nicht auf die naheliegende Weise mittels *zählenden Semaphoren* sondern mittels der rudimentäreren *binären Semaphoren* zu lösen.[3] Betrachten Sie folgenden Pseudocode, der das Produzenten- und Konsumentenproblem mittels binärer Semaphore angeht:

```
/* producer/consumer: binary semaphores                              */
S:          binary_semaphore = 1;
not_empty:  binary_semaphore = 0;
not_fill:   binary_semaphore = 0;
count : int = 0 ;

process producer ()
{
  while (true) {

    datum   := produce_item();          /* Produzent tut seine Aufgabe      */

    if (count = n) then wait(not_full);
    Buf[in] := datum;                   /* Einfuegen                        */
    wait(S);                            /* just to be sure
            count := count + 1;
    signal(S);
    if (count =  1) then signal(not_empty);
    in := (in + 1) mod N;
  }
}

process consumer ()
{
  while (true) {
    if (count = 0) then wait(not_empty);
    datum =  Buf[out];                  /* Datum auslesen                   */
    wait(S);
      count       := count - 1;
    signal(S)
    if (count = (N-1)) then signal(not_full);

    out   :=  (out + 1) mod N;          /* weiterz"ahlen                    */
    consume(datum);
  }
}
```

Wie in den in den Übungen besprochenen Lösungen ist auch hier das *Ziel,* den Produzenten und den Konsumenten über den Puffer kommunizieren zu lassen wobei die Zählervariable `count` zusammen mit den Semaphoren dazu dienen soll zu verhindern, daß der Konsument aus einem *leeren* Puffer liest und der Konsument in einen vollen Puffer schreibt.

Frage: ist dies eine angemessene Lösung des Problems? Falls ja, erläutern und begründen Sie die Korrektheit. Falls nein, zeigen Sie, wo das Programm fehlschlägt.

Bemerkung: nehmen Sie an, das Programm werde *zeilenweise* atomar ausgeführt.

[3] Zur Erinnerung: Binäre Semaphore sind gegenüber der allgemeinen Form eingeschränkt, indem sie nur *zwei nicht-negativee Werte* annehmen können: 0 und 1. Binäre Semaphore können als Spezialfall der "zählenden" (counting) Semaphore, wie sie unter anderem in Xinu realisiert sind, angesehen werden. Der Spezialfall besteht darin, daß die Semaphore nur angeben kann, ob eine Resource vorhanden ist (Wert 1) oder nicht (Wert 0), und nicht die Anzahl der freien Resourcen zählen kann.

Lösung:

Hier die *Lösung:* Der Code ist keine angemessene Lösung des Problems. Siehe beispielsweise Ben-Ari [BA90], Seite 59.

1. Der Produzent produziert ein Element, setzt den (globalen) counter damit auf *1* und führt entsprechend der Bedingung am Ende, das `signal` aus, um dem Partner zu sagen, daß ein Element im Puffer ist, d.h., der Wert der Semaphore `not_empty` ist damit auf 1.

2. Der Konsument überprüft die Variable `count` (nicht mehr `local_count`), die auf 1 steht, und aus diesem Grund führt er das `wait` nicht aus sondern fährt in seinem "Konsumentenrumpf" fort.[4] Also konsumiert er sein Dingens und setzt den Zähler wieder auf 0. D.h., der Puffer ist leer, der `count` auf 0, aber `non_empty` auf 1!

3. Der Konsument mache weiter, und wendet sich an die Semaphore, weil der `count` auf 0 ist, aber die Semaphore sagt: er kann eintreten.

Bemerkung: Dies ist eine *Erweiterung* einer in den Übungen besprochenen Aufgabe. Die dort geforderte (und besprochene) Lösung des Problem ist allgemeiner als der hier vorgestellte Code, und zwar dadurch, daß jeder Prozess eine *lokale Kopie* der globalen Variable `count` besitz. Da in den Übungen gesagt wurde "denkt mal drüber nach, ob man ohne `local-count` auskommt.", sind diejenigen, die in die Übungen gehen (und mitmachen) im Vorteil. Unabhängig davon ist die Aufgabe im Stil mehrerer Übungsaufgaben. Als eine "Betrachten-Sie-den-gegebenen-Code"-Aufgabe, sollte sie also einigermaßen leicht zu lösen sein.

[4] In der lokalen Lösung mit `local_count` fährt der Produzent auch mit dem Rumpf fort, er hätte jedoch ein `wait` auf`not_full` ausgeführt, welches die Semaphore dekrementiert.

Aufgabe 4 (Das Unisex-Badezimmerproblem (8 Punkte)) Betrachten Sie folgende abstrakte Beschreibung eines Synchronisationsproblems:

> Gegeben sei ein *Badezimmer* als eine gemeinsame Resource mehrerer Personen, die in zwei Klassen W und M (für weiblich und männlich) unterteilt seien. Die Personen haben den Wunsch, immer wieder das Badezimmer zu betreten (und irgendwann danach auch wieder zu verlassen). Das zu lösende Synchronisationsproblem bestehe darin zu verhindern, daß sich zu einem Zeitpunkt Personen zweierlei Geschlechts im Badezimmer befinden.

Implementieren Sie das beschriebene *Synchronisationsproblem.*

Mutex: Führen Sie ein Argument, daß die von Ihnen beschriebene Lösung den *gegenseitigen Ausschluß* (in dem oben geschilderten Sinne) gewährleistet.

Verklemmungsfrei: Ihre Lösung sollte die Situation vermeiden, bei der *niemand duscht* und dennoch niemand das Badezimmer betreten kann ("Verklemmung"/ deadlock).[5]

Für Ihre Lösung können Sie sich an folgendem Fragment orientieren:

```
/*          Unisex bath skeleton (male)                  */

...                                     // some declarations?

process M(i)                            // male_i
  loop

       do_something_else;               // non-critical code

       .....                            // add some sync-code

       shower();                        // critical code, no ''female'' may shower here

       ....                             // add some sync-code

  end loop;
end process
```

Wie bei Mutex-Problemen üblich, sind triviale Lösungen[6] *keine* Lösungen. Auch darf die Lösung nicht darin bestehen, das je nur immer *einzelne* Personen das Bad betreten.

Lösung: The problem is related to the *readers/writers*-problem, but of course symmetric.

One approach to the problem is to see that it is basically the same as the core mutual exclusion problem, when considering the set of males and females as a whole. It does not matter for the females, whether there's one male or many males in the shower, or vice versa. All solutions based on *counting semaphores* are probably therefore not useful.

[5] Beachte: die Aufgabe verlangt *nicht*, daß, beispielsweise, die Frauen niemals das Bad betreten können in Situationen, bei denen ständig mindestens ein Mann (aber immer andere) im Badezimmer sind.

[6] Niemand wäscht sich, oder "die Männer waschen sich nie".

In some sense it is related also to the problem of producer and consumer, only that the number of frees or used places as in the buffer is not known. If one would assume

```
/*         Unisex bath, simple                                  */
/*  this solution uses one single semaphore for the bath        */



counter_m : int = 0;                    // counting the males
counter_f : int = 0;                    // and the females.

bath:   semaphore =  1;                 // 1 means = bath is completely empty

Process M(i)                            // male_i
  loop

       do_something_else;               // non-critical code

       wait(mutex_m);                   // besser ist das
         counter_m := counter_m + 1;      // one more male
         if    counter_m = 1              // the first male
         then  wait(bath)                 // waits for an empty bathroom
         fi
       signal(mutex_m);

       shower();                       // critical code

       wait(mutex_m);
       counter_m := counter_m - 1;     // one male less
       if        (counter_m = 0)
       then signal(bath);              //  the last man gives the bath free
       fi;
       signal(mutex_m);
  end loop;

end process
```

Literatur

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* International Series in Computer Science. Prentice Hall, 1990.

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