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 (extra)** — 16. Januar 2004, 12:00–15:00

Aufgaben mit Lösungshinweisen

Termin: 16. Januar 2004, 12:00–15:00

- 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 (Xinus Speicherverwaltung (5 Punkte)) Schreiben Sie eine Routine `showmem`, die einen Überblick über den vorhandenen *freien Speicher* ausdruckt. Die Information soll eine Liste sein, die für jeden freien Speicherblock *Anfangsadresse* und *Länge* angibt.

Lösung: Hier sind wieder ein wenig Kenntnisse in Xinu gefragt. Wer sich den Code zu Gemüte geführt hat, sollte die relevanten Stelle relativ schnell ausfindig machen. Man muß nur Kapitel 8 finden und verstanden haben. Die Aufgabe läßt sich in diesem Falle sogar lösen, wenn man so gut wie gar keine C-Kenntnisse verfügt.

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, 1000, getprio(p1), "child", 1, p1);
   resume(p2);
   send(p2,5);
   return(1);
   return(0);

}

int child(int pid)
{
  int x;
  chprio(getpid(), getprio(getpid())-2);
  ''DO_SOME_INTERNAL_CALCULATIONS'';        // no system calls here!
  suspend(pid);
  x = receive(pid);
  return(x);
}
```

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:

[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`.

Aufgabe 3 (Produzent/Konsument (4 Punkte)) Ein klassisches Prozeßsynchronisationsproblem ist das *Produzenten/Konsumenten*-Problem[2] Gegeben sei folgendes Programmfragment.

```
/* producer/consumer Pseudocode                              */

#define N   100                    /* Plaetze im Puffer   */
int count   0                      /* belegte Plaetze     */

producer ()
{
  while (TRUE) {
    produce_item();                 /* Produzent tut seine Aufgabe  */
    if (count == N)                 /* Falls Puffer voll:           */
            suspend(currentpid);    /* Produzent suspendiert sich   */
    insert_item();
    count := count+1;
    if (count == 1)                 /* Konsument soll weitermachen */
            resume(consumer_pid); /* da Puffer nicht mehr leer   */
  }
}

consumer ()
{
  while (TRUE) {
    if (count == 0)
            suspend(currentpid);   /* Konsument suspendiert sich selbst */
    remove_item();
    count := count-1;
    if (count== N-1)
            resume(producer_pid); /* Produzent soll weitermachen    */
    consume_item();
  }
}
```

Die Absicht ist, die notwendige *Synchronisation* zwischen Konsument und Produzent durch wechselseitiges Suspendieren und Wiederaufwecken zu bewerkstelligen. Beachten Sie:

- daß Prozeduren zeilenweise atomar ausgeführt werden,
- `suspend` nur auf laufende und auf Prozesse in der Ready-Liste angewendet werden kann,
- und das `resume` nur auf suspendierte Prozesse angewendet werden kann.

Funktioniert die angegebene Lösung? Falls nein, finden Sie den Fehler. Falls ja, begründen Sie ihre Antwort.

Alternativ Unter diesen Annahmen funktioniert die angegebene Lösung nicht. Begründen Sie warum und beheben Sie den Fehler.

Lösung: Siehe Tanenbaum, Seite 59. Das Problem ist, daß ausschließlich *suspendierte* Prozesse wieder aufgeweckt werden können (s. [**?**] S. 67). Falls der Prozeß nicht suspendiert ist, gibt es einen Fehler.

Was nun passieren kann ist, daß bei leerem Puffer sich der Konsument selbst suspendieren will, aber bevor er dazu kommt, ist der Produzent an der Reihe. Er schreibt was in den Puffer

[2] Auch als das Problem des beschränkten Puffers bekannt (auf Englisch *producer/consumer*, *bounded-buffer*).

und will den Konsumenter wieder aufwecken, geht aber nicht, denn der war noch gar nicht suspendiert.

Der Standardvorschlag zur Lösung bestand darin, durch eine Abfrage zu verhindern, einen nicht suspendierten Prozess wieder aufzuwecken.

```
/* producer/consumer Pseudocode                              */

#define N   100                    /* Plaetze im Puffer  */
int count   0                      /* belegte Plaetze    */

producer ()
{
  while (TRUE) {
    produce_item();                 /* Produzent tut seine Aufgabe  */
    if (count == N)                 /* Falls Puffer voll:           */
             suspend(currentpid);   /* Produzent suspendiert sich   */

    enter_item();
    count := count+1;
    if ((count == 1) &&
        (&proctab[consumer_pid]->pstate == PRSUB))
    then if
         then resume(consumer_pid); /* da Puffer nicht mehr leer   */
         else skip
  }
}

consumer ()

{
  while (TRUE) {
    if (count == 0)
             suspend(currentpid);   /* Konsument suspendiert sich selbst */
    remove_item();
    count := count-1;
    if (count== N-1)
             resume(producer_pid); /* Produzent soll weitermachen    */
    consume_item();
  }

}
```

Es kann immer noch folgendes Szenario passieren: Der Konsument will sich suspendierend, wird aber direkt nach dem Test auf Gleichheit mit 0 durch den Produzenten ersetzt. Der Produzent tut seine Arbeit, stellt fest, daß er den Konsument aufwecken sollte, der ist nicht ein suspendiert, also macht nichts, produziert weiter. Wenn der Porduzent im nächsten Durchlauf an die selbe Stelle kommt, besitzt die Variable `count` bereits *den Wert 2*, d.h., wegen des *Tests auf Gleichheit mit 1* versucht er erst gar nicht mehr den Konsumenten aufzuwecken, der mittlerweilen supsendiert sein kann. Das geht dann so weiter, bis der Puffer voll ist, dann suspendiert sich der Produzent und das System ist *verklemmt!*

Es gibt eine etwas „pfiffigere“ Variante dieser falschen Lösung: eine Ursache des soeben beschriebenen Fehlerverhaltens scheint der *Test auf Gleichheit mit 1* zu sein. Was passiert wenn man, wie es einige getan haben, die Zeile im Produzenten

```
if (count >= 0) && issuspended (consumer_pid)
  resume(consumer_pid)
```

Analog im Konsumenten. Leider tritt der selbe Verklemmungsfehler immer noch auf: Der Konsument startet zu Beginn, stellt fest, daß noch kein Dings produziert wurde und will sich suspendieren, aber ein Kontextwechsel finded unmittelbar nach dem booleschen Test statt, der Konsument ist also noch nicht suspendiert. Daraufhin produziert der Produzent bis der Puffer voll ist. Währenddessen wir er den Konsumenten *nicht* aufwecken, denn er ist nicht suspendiert. Am Ende, sobald der Puffer voll ist, legt sich der Produzent selbst auf Eis. Nun muß ein Kontextwechsel zum Konsumenten stattfinden. Dieser macht nach dem if-Abfrage weiter und supsendiert sich ebenfalls und das System ist verklemmt.

Nebenbemerkungen: Zunächst einmal ist es zweifelhaft, anzunehmen, Dinge seien atomar **weil** und **sobald** man sie in eine Zeile schreibt. Die Konvention „*eine Zeile = atomar*“ war als Hilfestellung zur Beschreibung der Aufgabe gedacht, um nicht umständlich die Atomarizitätsannahmen bzgl. des Pseudocodes, seines Compilers, und der darunterliegenden Rechnerarchitektur erläutern zu müssen, jedoch nicht als Trick zur Lösung der Aufgabe. Wenn das im richtigen Leben so wäre, könnte man viele Aufgaben leicht (oder effizient) lösen, wenn man nur lang genuge Zeilen hat. Im Falle einer boolschen Verknüpfung mehrerer Bedingungen ist in der Realität die Annahme der Atomarizität *nicht* gerechtfertigt. Manche sind sogar soweit gegangen zu sagen: „unter der Annahme, daß Prozeduren atomar ablaufen . . . “. Dies ist nur dann der Fall, wenn die Prozedur die Interrupts ausschalten würde. Für normale Benutzerprozeduren stehen solche Tricks aber besser nicht zur Verfüngung. Das ist natürlich exakt der *Grund* warum Betriebssysteme Semaphoren und ähnliche Mechanismen zur Verfügung stellen!

Die meisten haben die Annahme

> „wenn ich den Vergleich auf 0 und den Test auf Suspendiertheit in eine Zeile schreibe, *dann* ist es atomar

in der einen oder anderen Form implizit angenommen und wir haben es nicht als Fehler gerechnet. Aber im richtigen Leben geht das nicht so, und Fehler die auftreten, weil man sich falsche Annahmen zur Atomarizität gemacht hat, sind *sehr* schwierig zu finden.

Aufgabe 4 (Gegenseitiger Ausschluss (8 Punkte)) Der angegebene Pseudocode stellt einen Vorschlag zur Lösung des Problems des *gegenseitigen Ausschlusses* dar. Der Algorithmus soll den gegenseitigen Ausschluß für zwei Prozesse P_0 und P_1 sicherstellen. Es wird angenommen, daß jede Zeile atomar ausgeführt wird, aber ansonsten jederzeit Kontextwechsel stattfinden können. Es ist der Code für P_0 angegeben, P_1 ist symmetrisch, das bedeutet, anstelle `c0` verwendet P_1 `c1` und auch die Werte für die Variable `last` sind endsprechend ausgewechselt. Die Werte für die Variablen seien mit `c0 = false`, `c1 = false`, und `last = 0` vorbelegt (aber die Vorbelegung spielt keine Rolle, da die Werte in jedem Fall überschrieben werden).

```
/* Process P0, P1 ist symmetrisch */
2
  process P0 =
4 begin
    while true do                    /* Endlossschleife */
6     some_actions;                  /* unkritischer Code */
      c0   := true;
8     last := 0;
      while (c1 = true) and (last = 0) do
10      skip;                        /* tue nichts       */
      od
12    critical_section_0;
      c0   := false;
14  od
  end;
```

Beweisen oder widerlegen Sie:

1. Der Algorithmus garantiert gegenseitigen Ausschluß.
2. Der Algorithmus birgt die Gefahr, daß beide Prozesse gemeinsam in ihrer Warteschleife (Zeilen 9–11) stecken und diese nie verlassen (live-lock).
3. Wenn es einen Prozeß gibt, der in die kritische Sektion möchte, so kommt dieser auch schließlich in die kritische Sektion.

Lösung: [?].

Literatur

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

[Pet83] G. L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems*, 5(1):56–65, 1983.