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 — **Klaussurreservoir** — 21. Oktober 2003

Termin: 21. Oktober 2003

Zusammenfassung

Dies ist eine Übersicht über alle Klausuraufgaben (oder zumindest alle, die ich gefunden habe). Manche der inputs sind rauskommentiert, weil sie sich gerade nicht übersetzen lassen.

Inhaltsverzeichnis

A Aufgaben

A.1 Allgemeine Aufgaben, i.d.R. Synchronisationsprobleme

Aufgabe 1 (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 */

  process P0 =
  begin
    while true do                    /* Endlossschleife */
      some_actions;                  /* unkritischer Code */
      c0   := true;
      last := 0;
      while  (c1 = true) and (last = 0) do
        skip;                        /* tue nichts       */
      od
      critical_section_0;
      c0   := false;
    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.

Aufgabe 2 (Produzenten/Konsumenten-Problem mit binären Semaphoren (5 Punkte))
Das Problem der Produzenten-Konsumenten-Koordination über einen beschränkten Puffer soll ein weiteres Mal angegangen werden. Diesmal soll nur eine eingeschränkte Form von Semaphoren zur Verfügung stehen, nämlich sogenannte *binäre* Semaphore. Diese sind gegenüber der allgemeinen Form eingeschränkt, indem sie nur *zwei Werte* annehmen können: 0 und 1.[1] Geben Sie eine Lösung für das Problem mit beschränktem Puffer an, indem Sie folgendes Codefragment vervollständigen! Erläutern Sie die Korrekheit Ihrer Lösung!

```
/* producer/consumer Pseudocode-Fragment                              */

int Buf[N];

producer ()
{
  int datum, in;

  while (TRUE) {

    datum    = produce();          /* Produzent tut seine Aufgabe      */
    Buf[in] = datum;                  /* Einfuegen                        */
    in       = (in + 1) mod N;        /* und modulo weiterz"ahlen         */

  }
}

consumer ()
```

[1] Läßt man negative Werte zu, z.B., wenn man, der Konvention in Xinu folgend, (mehrere) an der Semaphore wartende Prozesse mit einem negativen Zählerstand vermerkt, müßte man präziser sagen, binäre Semaphore lassen zwei *nicht-negative* Werte zu. Binäre Semaphore können als Spezialfall der "zählenden" 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.

```
{
  int datum, out;

  while (TRUE) {

    datum =  Buf[out];                /* Datum auslesen                */
    out   =  (out + 1) mod N;         /* weiterz"ahlen                 */
    consume(datum);

  }
}
```

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.[2] 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);
```

[2] 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.

```
35    }
}
```

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.

Aufgabe 4 (Problem der Zigarettenraucher) Unter dem „Problem der Zigarettenraucher“ versteht man folgendes Synchronisationsproblem. Gegeben seien vier Prozesse, drei Raucher und ein Agent oder Server. Jeder der drei Raucher raucht eine Zigarette nach der anderen, allerdings braucht er zum Rauchen drei Voraussetzungen: er braucht *Streichhölzer, Tabak* und *Papier.* Jeder der drei Raucher ist im Besitz eines der Güter, einer hat Tabak, der zweite Papier und der letzte Feuer. Der Agent besitzt alles drei, und jeder hat einen unendlichen Vorrat an den ihm jeweils zur Verfügung stehenden Materialien.

Das Verhalten des Agenten besteht darin, in einer Runde jeweils zwei unterschiedliche Güter auf den Tisch zu legen; wem genau diese beiden Dinge zum Rauchen fehlen, nimmt sie sich und raucht. Wenn er damit fertig ist, gibt er dem Agenten Bescheid und alles beginnt wieder von vorne. Agent und Klienten funktionieren also zyklisch.

Implementieren Sie das geschilderte *Synchronisationsproblem.* Ihre Lösung soll verklemmungsfrei sein.

Als Hinweis: Verwenden Sie für die drei Güter drei boolesche Variablen, sagen wir s, t und p, ein Wert von *true* bedeute, die Ressource steht zur Verfügung (“sie liegt auf dem Tisch”) und *false*, sie wurde konsumiert.

Um den Agenten nichtdeterministisch agieren zu lassen, können Sie einen +-Operator für die zufällige Auswahl zwischen verschiedenen Alternativen verwenden:

```
loop
  tue_dies;
 +
  tue_jenes;
 + ...
end loop;
```

Aufgabe 5 (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).[3]

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[4] *keine* Lösungen. Auch darf die Lösung nicht darin bestehen, das je nur immer *einzelne* Personen das Bad betreten.

Aufgabe 6 (Message passing) Xinu implementiert eine sehr einfache Art des Nachrichtenaustausches oder *message-passsings*. Es verwendet dazu die Systemaufrufe `send`, `recvclr` und `receive` (s. S. 94 ff. aus [Com83]).

- Programmieren Sie eine modifizierte Version des Sendens (`sendf`), die eine Nachricht dem Sender auf jeden Fall schickt, selbst wenn dieser bereits eine Nachricht besitzt (diese soll dann überschrieben werden).
- Programmieren und beschreiben Sie eine Version des Message-passings mit folgenden Eigenschaften:
  - Senden und Empfangen einer Nachricht sind *blockierend*, d.h., beim Sendeversuch an einen nicht-empfangsbereiten Prozeß muß der Sender warten, und umgekehrt.
  - Der Sender spezifiziert den Empfangsprozeß und umgekehrt.

  D.h., geben Sie zwei Systemaufrufe `send_rv(pid, msg)` und `receive_rv(pid)` an, die das beschriebene Verhalten realisieren.

[3] 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.

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

Aufgabe 7 (Test & Set) Wir haben in der Vorlesung eingehend über das Problem des gegenseitigen Ausschlusses und mögliche Lösungen —mittels Software oder durch Manipulation der Interrupts— gesprochen. Es gibt auch *hardwaregestützte* Lösungen, gegenseitigen Ausschluß zu garantieren. Ein bekannte Variante aus dieser Kategorie wird als *Test & Set* bezeichnet und ist Gegenstand dieser Aufgabe.

Gegeben seien 2 Prozesse P_0, P_1 und eine globale Variable C, mit möglichen Werten 0 und 1 und mit initialem Wert 0. Die *test-and-set*-Operation greift auf diese globale Variable zu, kopiert ihren Wert in eine lokale Variable und setzt C "gleichzeitig" auf 1. Der Aufruf `test_and_set(L)` entspricht somit der nicht-unterbrechbaren, atomaren Ausführung der beiden Zuweisungen:

```
L := C;
C := 1;
```

Betrachten Sie den angegebenen Code für gegenseitigen Ausschluß mittels *test-and-set.*

```
/*       Test & set                                                    */

1)     process P_i                      /* der i-te Prozess           */

       int L_i;                         /* lokale Kopie fuer P_i      */
2)       loop                           /* Endlosschleife             */

3)         unkritischer_Code;

4)         loop                         /* innere Schleife            */
5)           test_and_set(L_i);
6)           exit when (L_i == 0);      /* Abbruch der inneren Schleife */
7)         end loop;

8)         Beginn_kritischer_Abschnitt_i;
9)         Ende_kritischer_Abschnitt_i;
10)        C := 0;
11)      end loop;

12) end process;
```

Argumentieren Sie, daß die Implementierung den gegenseitigen Ausschluß garantiert, d.h., sie verhindert, daß sich zwei unterschiedliche Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden.

Zeigen Sie, auf welche Weise der gegenseitige Ausschluß scheitert, wenn die *test-and-set*-Operation *nicht atomar* ist, sondern durch das angegebene Zuweisungspaar implementiert würde. (Die Ziffern im Code sind „Zeilennummern“, die Sie in Ihren Argumenten verwenden können.)

Aufgabe 8 (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 9 (Produzent/Konsument (4 Punkte)) Ein klassisches Prozeßsynchronisationsproblem ist das *Produzenten/Konsumenten*-Problem[5] 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.

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

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.

Aufgabe 10 (Gegenseitiger Ausschluß & Swap (max. 5 Punkte)) Neben den in der Vorlesung besprochenen Lösungen des Problems des gegenseitigen Ausschlusses gibt es auch *hardwaregestützte* Lösungen, gegenseitigen Ausschluß zu garantieren. Eine bekannte Variante aus dieser Kategorie wird als *Test & Set* bezeichnet. Eine weitere Variante ist als *Swap* bekannt und ist Gegenstand dieser Aufgabe.

Swap ist dabei eine Maschineninstruktion, die es erlaubt, den Wert zweier (boolescher) Speicherzellen *atomar,* innerhalb eines Maschinenzyklus', miteinander auszutauschen. Der Aufruf einer *swap*-Instruktion entspricht somit dem nicht-unterbrechbaren Ausführen des folgenden Pseudocodes:

```
/*    swap-instruction    */
procedure swap (var: a, b: boolean);
  var temp: boolean;
  begin
    temp := a;
    a    := b;
    b    := temp
  end
```

Mittels der *swap*-Instruktion läßt sich das Problem des gegenseitigen Ausschlusses auf folgende Weise lösen. Gegeben seien n Prozesse P_i mit $1 \leq i \leq n$ und eine *globale* boolesche Variable C mit möglichen Werten *true* und *false*; sie sei mit *true* vorbesetzt. Der Pseudocode eines Prozesses P_i sieht wie folgt aus:[6]

```
/*   Gegenseitiger Ausschluss mit swap              */
process P_i =                /* der i-te Prozess       */
  var L_i = false: boolean   /* lokale Kopie fuer P_i  */

  loop                       /* Endlosschleife         */
    unkritischer-code;
    loop                     /* innere Schleife        */
      swap(C, L_i);
      exit when (L_i == true)
    end loop;
    kritischer-code;
    swap (C, L_i)
  end loop

end process
```

Argumentieren Sie, daß die Implementierung den gegenseitigen Ausschluß garantiert, d.h., er verhindert, daß sich zwei unterschiedliche Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden.

Zeigen Sie, auf welche Weise der gegenseitige Ausschluß scheitert, wenn die *swap*-Operation *nicht atomar* ist, sondern durch die angegebene Prozedur implementiert würde.

[6] Verwenden Sie ggf. die angegebenen Zeilennummern in ihren Argumenten.

A.2 Xinu-spezifische Aufgaben

A.2.1 Allg. Verständnisfragen

Aufgabe 11 Xinu-Prozeßzustände (max. 3 Punkte) Listen Sie die Xinu-Prozeßzustände auf, erläutern Sie, warum diese eingeführt wurden. Nennen Sie zudem die eventuell assoziierten Warteschlangen und erklären Sie, wie diese logisch organisiert sind.

A.2.2 Zum Implementieren

Aufgabe 12 (Semaphoren) Die in Xinu implementierten Semaphoren haben folgende bedenkliche Eigenschaft: löscht man mittels `sdelete` eine Semaphore, so werden alle an ihr wartenden Prozesse aufgeweckt (s. [Com83], Seite 89). Eine Alternative, die wir in einer Übung behandelt hatten, ist, daß man den Löschversuch einer Semaphore, an der Prozesse warten, mit einem Systemfehler quittiert.

Die vorliegende Aufgabe besteht darin, eine weitere Variante zu implementieren. Das Löschen einer existierenden Semaphore mittels `sdelete` soll, unabhängig davon, ob an der Semaphore Prozesse warten, erlaubt sein. Der Effekt soll allerdings sein, daß die Semaphore nicht sofort, sondern *verzögert* gelöscht wird, zu genau dem Zeitpunkt nämlich, wenn zum ersten mal kein Prozeß mehr an der Semaphore wartet. Implementieren Sie den beschriebenen Vorschlag, indem Sie die betroffenen Systemaufrufe und Datenstrukturen von Xinu anpassen.

Die beschriebene Strategie zum Löschen von Semaphoren ist zweifelhaft. Stellen Sie dar, mit welchem deutlich unerwünschten Verhalten man rechnen muß, auch wenn sich Prozesse, die sich der Semaphore bedienen, nicht absichtlich böswillig verhalten.

Aufgabe 13 (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.

A.2.3 Prozessabläufe

Aufgabe 14 (Xinu & Uhrinterrupts) Nehmen Sie an, ein Uhr-Interrupt wird ausgelöst, der zwei schlafende Prozesse aufweckt. Einer davon habe höhere Priorität als der momentan ausgeführte Prozeß (der "current"-Prozeß). Beschreiben Sie das Aufrufgeschehen.

Aufgabe 15 (Xinu-Prozeßabläufe (5)) 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 my_pid;
   my_pid = getpid();

   resume(create(child, 200, getprio(my_pid)+1, "child", 1, my_pid));
   receive();
   return(1);

}

int child(int pid)
```

```
{
  chprio(getpid(), getprio(getpid())-2);
  sleep(1);
  send(pid,10);
  return(1);
}
```

Beschreiben Sie das Ablaufgeschehen in Xinu, d.h., geben Sie die Reihenfolge der Systemaufrufe an. Wann treten Kontextwechsel auf und wann terminieren die Prozesse?

Aufgabe 16 (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.[7]

Aufgabe 17 (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;
```

[7] 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`.

```
    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.[8]

Aufgabe 18 (Xinu-Prozeßabläufe (5)) 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 das Ablaufgeschehen in Xinu, d.h., geben Sie die Reihenfolge der Systemaufrufe an, wann treten Kontextwechsel auf und wann terminieren die Prozesse.

[8] 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 19 (Xinu-Prozeßabläufe (max. 4 Punkte)) Nehmen Sie an, es gebe im laufenden Xinu zwei Prozesse mit Prozeß-id 1 und 2, aber keine mit höheren Prozeßidentifikatoren. Nehmen Sie ferner an, P_1 werde momentan ausgeführt (Zustand "`CURRENT`") und rufe `wait(S)` auf. Dabei stehe `S` für ein existierende Semaphore mit aktuellem Wert 0. Der Prozeß P_2 befinde sich im bereiten Zustand ("`READY`") und zwar direkt vor der Ausführung von `sleep (1); signal(S)`. Der Prozeß P_1 besitze dabei höhere Priorität als P_2.

Beschreiben Sie das Ablaufgeschehen in Xinu.

Aufgabe 20 (Xinu-I/O (max. 5 Punkte)) Beschreiben Sie das Ablaufgeschehen in Xinu, wenn ein auf der Tastatur eingegebenes Zeichen durch den Gerätetreiber Tty auf dem Bildschirm ausgegeben wird. Diskutieren Sie die Fälle `RAW` und `CBREAK` Mode getrennt. Wer ist jeweils für die Ausgabe verantwortlich?

Aufgabe 21 (Produzent/Konsument (TTY) und Echo (3 + 2 Punkte)) In Serie 10 von letzter Woche zum Konsumenten/Produzenten-Problem haben wir festgestellt, daß man zur Koordination bei *beschränktem* Puffer *zwei* Semaphoren braucht. Schaut man sich in Xinu den Kontrollblock für Tty's an ([Com83], S. 163), stellt man erstens fest, daß die Kommunikation mittels endlicher Puffer geschieht und zweitens, daß pro Produzenten/Konsumenten-Paar nur je *eine* Semaphore implementiert ist.

- Erklären Sie, warum zwei Semaphoren dort nicht verwendet werden können und auf welche Weise dennoch eine korrekte Funktionsweise sichergestellt ist.
- Neben dem Eingabe– und dem Ausgabepuffer gibt es noch den "Echo"-Puffer. Dieser ist ganz ohne eigene Semaphore implementiert. Warum?

Aufgabe 22 (Kontextwechsel (5 Punkte))

- Wann beginnt die CPU tatsächlich die Instruktionen des neuen Prozesses bei einem Kontextwechsel durch Aufruf von *ctxsw* auszuführen?
- Vergleichen Sie einen Prozeßder *resched()* aufruft und direkt zurückkehrt und einen bei dem der Kontextwechsel *ctxsw* durchgeführt wird! Was ist der Unterschied?
- Warum wird am Ende von *ctxsw* rtt ausgeführt im Gegensatz zu *rts* bei normalen Prozeduraufrufen?
- Worauf zeigt der Stackpointer nach Rückkehr von *ctxsw* und warum?
- In Xinu gibt es die Funktion *ready* um Prozesse in die Ready-Queue einzureihen. Neben dem Prozessidentifikator hat diese Funktion einen zweiten Parameter, welcher angibt, ob ein *resched()* stattfinden soll. Warum ist das sinnvoll? Denken Sie an ein Szenario, bei dem mehrere Prozesse gleichzeitig in den Ready-Zustand versetzt werden sollen (sdelete)!

Literatur

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