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

Sys. Inf. III (Betriebssysteme)

Wintersemester 2004/05 — **Klausur** — 11. Dezember 2004, 10:00–13:00

Termin: 11. Dezember 2004, 10:00–13:00

Um die Korrektur zu vereinfachen und zu beschleunigen:

- Geben Sie auf jedem Blatt Ihren **Namen**/Matrikelnummer an.
- Fangen Sie jede Aufgabe auf einem **neuen Blatt** an (nicht nur auf einer neuen Seite!) und kennzeichnen Sie das Blatt entsprechend!
- Die Verwendung von Unterlagen, einschließlich "dem Comer" ist erlaubt. Elektronische Unterlagen wie Handy und Computer sind verboten. Tachenrechner sind 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, irgendwelche `include`-Befehle anzugeben.

Für "abstrakte" Synchronisationsaufgaben (hier Aufgabe 3) 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!

Name:
Matrikelnummer:

Aufg 1	Aufg 2	Aufg 3	Aufg 4	Gesamt	Note

Aufgabe 1 (Gegenseitiger Ausschluss (5 Punkte)) Der angegebene Pseudocode 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. Die Werte für die Variablen seien mit `c0 = false`, `c1 = false`, und `last = 0` vorbelegt.

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

process P1 =
begin
  while true do                    /* Endlossschleife */
    some_actions;                  /* unkritischer Code */
    c1   := true;
    If c0 = true then last := 1;
    while  (c0 = true) and (last = 1) do
      skip;                        /* tue nichts       */
    od
    critical_section_1;
    c1  := 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 7–9 bzw. 21–23) 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, unabhängig wann er an die Reihe kommt, solange jeder Prozess unendlich oft dran kommt.

Aufgabe 2 (Semaphoren (5 Punkte)) Die übliche Semantik von einem *wait* auf eine Semaphore *sem* ist wie folgt definiert: Wenn $sem > 0$ dann ziehe eins ab und fahre fort, wobei das testen auf $sem > 0$ und das abziehen atomar stattfindet. Falls $sem \leq 0$ dann setzte den Prozess in einen Wartezustand (der Prozess wartet an der Semaphore *sem*), wobei testen und in Wartezustand setzen atomar stattfindet. Die Semantik von einem *signal* auf eine Semaphore *sem* ist wie folgt definiert: Erhöhe den Wert von *sem* um eins. Falls Prozesse an der Semaphore *sem* warten, dann aktiviere einen von diesen, wobei zufällig bestimmt wird welcher von diesen aktiviert wird. Dieser aktivierte Prozess wird, wenn er an die Reihe kommt, wieder *wait* auf *sem* ausführen.

Gegeben sind folgende zwei Prozesse, die die selbe Priorität haben:

```
1  sem:  semaphore = 1;
2
3  process P1 =
4  begin
5    while true do
6        non-critical;
7        wait(sem);
8          critical;
9        signal(sem);
10   od
11 end
12
13 process P2 =
14 begin
15   while true do
16       non-critical;
17       wait(sem);
18         critical;
19       signal(sem);
20   od
21 end
```

1. Geben Sie einen Programmablauf an, so dass Prozess P2 nie in seine critical section kommen kann, obwohl er mit seiner non-critical section terminiert ist. Nehmen Sie hierbei an, dass das time sharing des Betriebssystem dafür sorgt, dass P1 und P2 sich abwechseln.

2. Kann die obige Situation auch in Xinu auftreten? Begründen Sie detailliert Ihre Antwort, d.h. beschreiben Sie was in Xinu passiert.

3. Wie verändert sich das Verhalten von Xinu, wenn die Prozesse verschiedene Prioritäten haben?

Aufgabe 3 (Problem der Zigarettenraucher (5 Punkte)) Unter dem „Problem der Zigarettenraucher“ versteht man folgendes Synchronisationsproblem. Gegeben seien sieben Prozesse, sechs Raucher und ein Agent oder Server. Jeder der sechs 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, zwei haben Tabak, weitere zwei haben Papier und die letzten zwei haben 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. Weiterhin darf der Agent nicht festlegen welcher Raucher an die Reihe kommt. 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, wobei ein Wert von *true* bedeutet, dass die Ressource steht zur Verfügung (“sie liegt auf dem Tisch”) und *false*, dass sie nicht zur Verfügung steht.

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 4 (Kontextwechsel (Kapitel 4, Seite 53ff) (5 Punkte))

1. Wann beginnt die CPU tatsächlich die Instruktionen des neuen Prozesses bei einem Kontextwechsel durch Aufruf von *ctxsw* auszuführen?

2. Vergleichen Sie einen Prozeß der *resched()* aufruft und direkt zurückkehrt mit einen bei dem der Kontextwechsel *ctxsw* durchgeführt wird! Was ist der Unterschied? (Hinweis: In beiden Fälle wird die selbe resched Funktion betrachtet und **nicht** eine version wo der Aufruf zu *ctxsw* fehlt.)

3. Warum wird am Ende von *ctxsw* rtt ausgeführt im Gegensatz zu *rts* bei normalen Prozeduraufrufen? Kann rtt mittels *rts* und weiteren Befehle simuliert werden? Mit Begründung.

4. Worauf zeigt der Stackpointer nach Rückkehr von *ctxsw* und warum?

5. 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, Seite 89)!