

Musterlösungen zu den Tutoraufgaben auf  
Blatt 5  
der Übungen zur Vorlesung  
“Grundlagen Betriebssysteme und Systemsoftware

---

G.Groh, 15.11.2008

**Frage:** Erläutern Sie den Unterschied zwischen Thread und Prozess. Gehen Sie dabei auf deren Beziehung zueinander ein. Nennen Sie 2 Beispiele für Informationen bzw. Datenstruktur-Instanzen die für jeden (User-)Thread spezifisch sind und nennen Sie 2 Beispiele für Informationen bzw. Datenstruktur-Instanzen die für jeden Prozess spezifisch sind (die also (User-)Threads gemeinsam nutzen müssen)!

### Antwort:

Prozess = Adressraum + Thread(s)

Thread (oder leichtgewichtiger Prozess)

- gehört zu einem Adressraum (kein eigener Adressraum)
- teilt sich Daten, Code und Betriebsmittel mit anderen Threads des gleichen Prozesses
- hat Befehlszähler, aktuelle Registerwerte, Stack (Keller), Zustand

In traditionellen Umgebungen hat jeder Prozess genau einen Thread, der den Kontrollfluss des Prozesses repräsentiert, während in Multithreaded-Umgebungen ein Prozess mehrere Threads besitzen kann.

- Thread-spezifisch sind bspw.: Befehlszähler, aktuelle Registerwerte, Keller (Stack), Ablaufzustand (running, waiting, ready, terminated).
- Prozess-spezifisch sind bspw.: Adressraum, globale Variable, offene Dateien, Kindprozesse, eingetroffene Alarme bzw. Interrupts, Verwaltungsinformationen.

**Frage:** Erläutern Sie kurz, warum der Aufwand für das Erzeugen eines (User-)Threads i.A. geringer ist als für das Erzeugen eines Prozesses!

**Antwort:** (User-)Threads haben keinen eigenen Adressraum. Beim Erzeugen eines Prozesses muss dieser erst erzeugt werden.

**Frage:** Nennen Sie vier Gründe für die Einführung von Threads!

### **Antwort:**

- Durch die Nutzung des gemeinsamen Adressraums wird die Interaktion zwischen den Threads (Kommunikation und gemeinsame Nutzung von Daten) erleichtert.
- Aufwand für Erzeugen und Löschen von Threads ist geringer als für Prozesse.
- Verbesserung der Performanz der Applikationsausführung durch Nutzung mehrerer Threads, insbesondere bei I/O-intensiven Applikationen.
- Bei einem Multiprozessor-Rechensystem echte Parallelität innerhalb einer Applikation.

**Frage:** Wie können sich verschiedene Threads gegenseitig negativ beeinflussen?

**Antwort:** Prinzipiell können sich nebenläufige Threads – genau wie nebenläufige Prozesse - gegenseitig negativ beeinflussen, wenn sie unsynchronisiert auf dieselben Ressourcen zugreifen. Durch den gemeinsam genutzten Adressraum bestehen zusätzliche Einflussmöglichkeiten.

**Frage:** Warum darf der spezielle Leerlauf-Thread niemals den Zustand "warten" annehmen und wie kann dies erreicht werden?

**Antwort:** Falls alle Threads warten ist die CPU frei und es läuft ein Leerlauf-Thread (darf nicht anhalten, hat geringste Priorität, muss jederzeit verdrängbar sein). Er ist beispielsweise für Prüfungen oder Heap Kompaktifizierung nutzbar.

Die CPU darf dabei nicht in den echten „Leerlauf“ gehen, weshalb immer ein Thread ausführbar sein muss. Ein Thread im Zustand „warten“ kann nicht ausgeführt werden.

=> Leerlauf-Thread muss immer ausgeführt werden können.

=> Er darf sich nicht im Zustand „warten“ befinden.

**Frage:** Nennen Sie die Bedeutungen folgender Begriffe: kritischer Bereich, Race Condition, aktives Warten, passives Warten!

- **Kritischer Bereich**

Teil eines Programms, in dem auf gemeinsam genutzte Ressourcen zugegriffen wird. Kritische Bereiche müssen unter gegenseitigem Ausschluss ausgeführt werden.

- **Race condition**

Wenn nebenläufige Prozesse unkoordiniert den Inhalt einer gemeinsamen Variable modifizieren, können unterschiedliche Abläufe konkurrierender Zugriffe zu unterschiedlichen Ergebnissen führen. Grund ist der Ablauf von Modifikationen in mehreren Schritten: Holen des Operanden, Modifikation im Prozessor, Zurückschreiben.

*Folge:* Ergebnis der Ausführung nebenläufiger Prozesse ist nicht deterministisch.

- **Aktives Warten**

Eine Realisierungsmöglichkeit des wechselseitigen Ausschlusses besteht darin, eine globale Datenstruktur zur Verfügung zu stellen, deren Wert von den konkurrierenden Prozessen getestet wird. In Abhängigkeit von diesem Wert beginnt der jeweils testende Prozess mit der Ausführung seines kritischen Bereichs oder er testet aktiv weiter.

- **Passives Warten**

Im Gegensatz zum aktiven Warten wird ein Prozess, der nicht in einen kritischen Bereich eintreten kann, weil sich ein anderer Prozess gerade darin befindet, in einen Warte-Zustand versetzt. Er wird aufgeweckt, wenn der kritische Bereich wieder frei ist.

Nennen Sie **4 Bedingungen** die eine gute Lösung zur Garantierung des **wechselseitigen Ausschlusses** erfüllen sollte!

4 Bedingungen die eine gute Lösung zur Garantierung des **wechselseitigen Ausschlusses** erfüllen sollte: (Tanenbaum (Engl. Ausgabe 2001) S 102)

1. Keine zwei Prozesse dürfen **gleichzeitig im kritischen Bereich** sein
2. **Keine Annahmen** über Geschwindigkeit und Zahl der CPUs bzw des Process Scheduling
3. Kein Prozess, der außerhalb des kritischen Bereichs läuft ,darf dabei **andere Prozesse blockieren**
4. Kein Prozess sollte **für immer** auf das Betreten des kritischen Bereichs **warten müssen**

Diskutieren Sie die Vor und Nachteile folgender meist Busy Waiting basierender Strategien zum wechselseitigen Ausschluss:

- Interrupts disablen
- Einfache Lock Variable ohne TSL
- Strict Alteration
- Peterson Lösung
- Lock Variable mit TSL

### Disabling Interrupts: Nachteile

- Kontrolle über Interrupts durch User-Prozesse nachteilig (Sicherheit, Korrektheit) (Bsp Korrektheit: Fall dass User-Prozess bei abgeschalteten Interrupts in Endlosschleife gerät)

### Disabling Interrupts: Vorteile

- Einfach und sicher, wenn kritische Bereiche in OS-eigenen Strukturen von OS-Prozessen betreten werden → kritische Operationen können sicher beendet werden → OS bleibt in konsistentem Zustand  
(OS Prozeduren i.a. weniger kritisch bzgl Sicherheit und Korrektheit)

## Einfaches Lock

P1 and P2

```
while (TRUE){
    if(lock == 0){
        lock = 1;
        critical_region();
        lock = 0;
    }
    non_critical_region();
}
```

## Einfaches Lock

P1 and P2

```
while (TRUE){  
    if(lock == 0){  
        lock = 1;  
        critical_region();  
        lock = 0;  
    }  
    non_critical_region();  
}
```

(\*) Unterbrechung  
zwischen diesen  
beiden Anweisungen  
theoretisch möglich

## Einfaches Lock: Nachteile

- Wenn P1 an der Stelle (\*) vom Scheduler unterbrochen wird, wird P2 laufen (lock ist dann noch 0), lock = 1 setzen und in den kritischen Bereich eintreten. Wenn P2 dort wiederum unterbrochen wird, wird P1 an (\*) weitermachen, lock = 1 setzen und auch in den kritischen Bereich eintreten → race!

### Busy Waiting: Strict Alteration

```
while (TRUE){
    while(turn!=0);
    critical_region();
    turn = 1;
    non_critical_region();
}
```

```
while (TRUE){
    while(turn!=1);
    critical_region();
    turn = 0;
    non_critical_region();
}
```

# Aufgabe 4.8 Lösungsvorschlag

P1

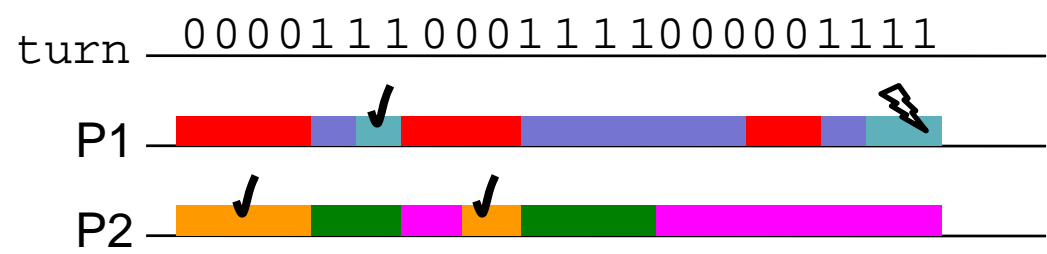
```
while (TRUE){  
    while(turn!=0);  
    critical_region();  
    turn = 1;  
    non_critical_region();  
}
```

P2

```
while (TRUE){  
    while(turn!=1);  
    critical_region();  
    turn = 0;  
    non_critical_region();  
}
```

## Busy Waiting: Strict Alteration: Nachteile

- Busy Waiting verschwendet CPU Zyklen
- Ggf: P1 muss auf P2 warten obwohl P2 in noncritical region ist → Verletzung ⚡ von Anforderung (3) ; Bsp:



## Busy Waiting: Strict Alteration: Vorteile

- Garantiert wechselseitigen Ausschluss: Keine Races.

## Aufgabe 4.8 Lösungsvorschlag

### Peterson Lösung (siehe Tanenbaum (Engl. Ausgabe 2001) S 105f)

#### P1 and P2

```
int N=2;           //N: number of Processes
int turn;         //who`s turn is it
int interested[N]; //all initialized to 0

void enterRegion(int process){
    int other = 1 - process;
    interested[process] = TRUE; //show interest
    turn = process;           //set flag
    while((turn==process)&&(interested[other]==TRUE)); //wait if
                                                    necessary
}

void leaveRegion(int process){
    interested[process] = FALSE; //indicate departure from critical region)
}
```

### Peterson Lösung: Vorteile

- Garantiert wechselseitigen Ausschluss: Keine Races.
- Keine Strict Alteration → Keine Verletzung von Forderung (3)

### Peterson Lösung: Nachteile

- Busy Waiting verschwendet CPU cycles

## Aufgabe 4.8 Lösungsvorschlag

Lock mit TSL (siehe Tanenbaum (Engl. Ausgabe 2001) S 107f)

P1 and P2

```
enterCriticalSection:
    TSL REGISTER, LOCK      copy lock to register and set lock to 1
    CMP REGISTER, #0        was lock zero?
    JNE enterCriticalSection if it was not zero (lock was set): loop
    RET                     return

leaveCriticalSection:
    MOVE LOCK, #0          store 0 in lock
    RET                     return
```

Works but has disadvantages of busy waiting

## Aufgabe 4.8 Lösungsvorschlag

If TSL instruction would not be used:

P1 and P2

```
enterCriticalSection:
    (*) LOAD REGISTER, LOCK      } copy lock to register and set lock to 1
    MOVE LOCK, #1                 }
    CMP REGISTER, #0              } was lock zero?
    JNE enterCriticalSection     } if it was not zero (lock was set): loop
    RET                           } return

leaveCriticalSection:            } store 0 in lock
    MOVE LOCK, #0                 } return
    RET
```

Process can be interrupted at (\*) → race possible

## Aufgabe 4.8 Lösungsvorschlag

Ist **Busy Waiting** (bspw. Lock mit TSL) auch zur Verwirklichung des wechselseitigen Ausschlusses **für Threads** geeignet?

### Lock mit TSL

P1 and P2

```
enterCriticalSection:
    TSL REGISTER, LOCK           copy lock to register and set lock to 1
    CMP REGISTER, #0             was lock zero?
    JNE enterCriticalSection     if it was not zero (lock was set): loop
    RET                           return

leaveCriticalSection:
    MOVE LOCK, #0                store 0 in lock
    RET                           return
```

## Aufgabe 4.8 Lösungsvorschlag

### P1 and P2

```
enterCriticalSection:
    TSL REGISTER, LOCK      copy lock to register and set lock to 1
    CMP REGISTER, #0        was lock zero?
    JNE enterCriticalSection if it was not zero (lock was set): loop
    RET                     return

leaveCriticalSection:
    MOVE LOCK, #0          store 0 in lock
    RET                     return
```

**NEIN!** Denn im Gegensatz zu Prozessen, die vom Prozess-Scheduler unterbrochen werden müssen User-Threads „selbst fürs Unterbrechen sorgen“. Wenn also zwei User-Threads des selben Prozesses versuchen würden, Busy Waiting zum WA zu benutzen, würde der wartende Thread wenn er denn ausgeführt wird, für immer warten, da der andere Thread nie die Gelegenheit hätte das Lock aufzuheben.

## Aufgabe 4.8 Lösungsvorschlag

Lösung: Mutexe mit call zu spezieller Funktion `thread_yield` verwenden, die auch innerhalb eines Prozesses das Thread-Scheduling regelt, indem Sie zu einem anderen Thread wechselt. In Assembler sähe das ungefähr so aus: (siehe Tanenbaum (Engl. Ausgabe 2001) S 113f)

### P1 and P2

```
mutex_lock:
    TSL REGISTER, LOCK           copy lock to register and set lock to 1
    CMP REGISTER, #0            was lock zero?
    JZE ok                      if it was zero mutex was unlocked, so return
    CALL thread_yield          mutex is busy, schedule another thread
    JMP mutex_lock             try again later
ok:    RET

mutex_unlock:
    MOVE MUTEX, #0             store 0 in mutex
    RET                        return
```

## Aufgabe 5

```
public void P (int *s) {
    *s = expr1 ;
    if ( expr2 ) { Prozess in die Menge der bezüglich *s
        wartenden Prozesse einreihen }
}

public void V (int *s) {
    *s = expr3 ;
    if ( expr4 ) { Führe genau einen der bezüglich *s wartenden
        Prozesse in den Zustand Rechenwillig über }
}
```

## Aufgabe 5.1, 5.2 Lösungsvorschlag

```
public void P (int *s) {
    *s = expr1 ;
    if ( expr2 ) { Prozess in die Menge der bezüglich *s
        wartenden Prozesse einreihen }
}

public void V (int *s) {
    *s = expr3 ;
    if ( expr4 ) { Führe genau einen der bezüglich *s wartenden
        Prozesse in den Zustand Rechenwillig über }
}
```

Ersetze:

*expr1* durch  $*s - 1$

*expr2* durch  $*s < 0$

*expr3* durch  $*s + 1$

*expr4* durch  $*s <= 0$

Mutex: Binäre Semaphore

## Aufgabe 6

Deklaration:

```
wa(1);
```

Erzeuger E:

```
while(true){  
    produziere Element;  
    wa.P();  
    schreibe Element nach W;  
    wa.V();  
}
```

Verbraucher V:

```
while(true){  
    wa.P();  
    entnimm Element aus W falls Element vorhanden, sonst warte;  
    wa.V();  
    verarbeite Element;  
}
```

## Aufgabe 6.1 Lösungsvorschlag

Deklaration:

```
wa(1);
```

Erzeuger E:

```
while(true){  
    produziere Element;  
    wa.P();  
    schreibe Element nach W;  
    wa.V();  
}
```

Verbraucher V:

```
while(true){  
    wa.P();  
    entnimm Element aus W falls Element vorhanden, sonst warte;  
    wa.V();  
    verarbeite Element;  
}
```

Problem: Es kann eine Verklemmung auftreten, wenn der Verbraucher `wa.P()` ausführt und warten muss, weil der Puffer kein Element enthält. Andererseits kann der Erzeuger den Puffer nicht betreten, da bereits der Verbraucher den Puffer exklusiv belegt hat.

**Genauere Diskussion: Siehe Skript**

## Aufgabe 6.2 Lösungsvorschlag

Modifizierte korrekte Variante:

Deklaration:

```
wa(1);  
voll(0);  
leer(n);
```

Erzeuger E:

```
while(true){  
    produziere Element;  
    leer.P();  
    wa.P();  
    schreibe Element nach W;  
    wa.V();  
    voll.V();  
}
```

Verbraucher V:

```
while(true){  
    voll.P();  
    wa.P();  
    entnimm Element aus W;  
    wa.V();  
    leer.V();  
    verarbeite Element;  
}
```

## Aufgabe 6.3 Lösungsvorschlag

Deklaration:

```
wa(1);  
voll(0);  
leer(n);
```

Erzeuger E:

```
while(true){  
    produziere Element;  
    leer.P();  
    wa.P();  
    schreibe Element nach W;  
    wa.V();  
    voll.V();  
}
```

Verbraucher V:

```
while(true){  
    voll.P();  
    wa.P();  
    entnimm Element aus W;  
    wa.V();  
    leer.V();  
    verarbeite Element;  
}
```

Die Reihenfolge der P-Operationen darf nicht vertauscht, werden, da es sonst zu einem Deadlock kommen kann

Bsp: Vertausche leer.P() und wa.P() im Erzeuger →

Puffer voll → Erzeuger blockiert mit wa=0 → Verbraucher wird wa.P() ausführen und auch blockieren → beide blockieren für immer. RIP 😊