3.3. Probleme der Interprozeßkommunikation

3.3.1. Wettlaufbedingungen (race conditions)

- zu Wettlaufbedingungen

■ Bsp. Spoolerkatalog

- Wenn Prozeß eine Datei ausdrucken lassen will, gibt er den Namen der Datei in Spoolerkatalog.
- Anderer Prozeß (Druckerdämon) prüft ob Dateien gedruckt werden müssen

- Annahme:
  Spoolerkatalog verfügt über mehrere Plätze, die Dateinamen enthalten können, sind durchnumeriert.
  zwei Variable *in* und *out*
  out zeigt auf nächste zu druckende Datei
  in zeigt auf nächsten freien Speicher

  - Prozeß A will drucken, liest *in*= 7
  - Prozeß B unterbricht jetzt Prozeß A, liest ebenfalls *in*= 7, legt an dieser Stelle Namen für zu druckende Datei ab
  - Prozeß A gelangt wieder zur Ausführung, legt den Namen für zu druckende Datei ebenfalls auf Platz 7, überschreibt Dateinamen von Prozeß B
  - Prozeß B wartet ewig darauf, daß seine Daten gedruckt werden.

■ Bsp. Flugüberwachung.

- Zwei Prozesse (Task A und Task B)
- Task A verantwortlich für das Erkennen von möglichen Kollisionen
- Task B soll die Positionstabelle mit den neuen X- und Y-Koordinaten der Flugzeugpositionen aktualisieren

- Es muß gewährleistet sein, daß Task B die Positionstabelle nicht verändert, sonst zeigt Task A eine nicht existierende Kollision an.

  - Task A liest X-Koordinate der X-Position
  - Task B verdrängt Task A
  - Task B aktualisiert X-, Y-Koordinate
  - Task B beendet Arbeit, gibt CPU frei
  - Task A setzt fort, liest Y-Position ein

  → Task A nimmt an, daß Flugzeug sich an alter X-und neuer Y-Position befindet.

⇒ Starten mehrere Prozesse eine Operation, deren Ergebnis vom Zeitpunkt des Aufrufes abhängig ist, kann es zu unerwünschten Nebeneffekten bis hin zu fehlerhaften Ergebnissen kommen

Def.:
Wettlaufbedingungen liegen vor, wenn zwei o. mehrere Prozesse lesend o. schreibend auf gemeinsame Daten zugreifen und deren Ergebnisse vom Prozeßfortschritt abhängen

3.3.2. Kritische Abschnitte (critical section) und gegenseitiger Ausschluß (mutual exclusion)

Wie vermeidet man Wettlaufbedingungen?

- Es ist zu verhindern, daß mehr als ein Prozeß zur gleichen Zeit auf gemeinsame Daten lesend oder schreibend zugreifen darf.
- Benötigt wird gegenseitiger Ausschluß (mutual exclusion)
- ***mutual exclusion*** ist ein Verfahren, daß anderen Prozessen den Zugriff auf gemeinsam benutzte Variablen und Daten verwehrt, solange ein Prozeß damit arbeitet. Der Programmteil, aus dem zum gemeinsamen Speicher zugegriffen wird, heißt ***kritischer Abschnitt***
- **Wettkampfbedingungen können vermieden werden, wenn Prozesse niemals gleichzeitig in ihre kritischen Abschnitte eintreten.**
- Das ist noch nicht hinreichend damit parallele Prozesse korrekt zusammenarbeiten und effizient gemeinsame Informationen benutzen:

1. 2 Prozesse dürfen sich nicht gleichzeitig in ihren kritischen Abschnitten befinden.
2. Es dürfen keine Annahmen über die relativen Geschwindigkeiten der Prozesse oder der Prozessoren gemacht werden
3. Kein Prozeß sollte beliebig lange auf Eintritt in den kritischen Abschnitt warten müssen
4. Ein Prozeß der sich außerhalb seines krit. Abschnittes befindet, sollte andere nicht blockieren.

3.3.3 Gegenseitiger Ausschluß (mutual exclusions) durch aktives Warten

1. Sperren von Unterbrechungen

- Bei Betreten des kritischen Abschnittes alle Unterbrechungen sperren, bei Austritt wieder freigeben
- CPU wird nur dann von Prozeß zu Prozeß geschaltet, wenn Unterbrechungssignal vom Zeitgeber oder anderen Geräten eintrifft.
- Ansatz gefährlich, denn Benutzerprozesse sollen Interrupts nicht sperren oder freigeben dürfen.
- Wenn Benutzer Interrupts sperrt und diese nicht wieder frei gibt → Ende des Systems
- Betriebssystemkern sperrt häufig Interrupts für einige Befehle, während er Variablen und Tabellen aktualisiert. Für Kern nützlich, Anwender sollten Interrupts aber nicht sperren.

2. Synchronisationsvariable

lock {0;1}

lock = 0: kein Prozeß im krit. Abschnitt
lock = 1: Prozeß im krit. Abschn.

lock = 0
Prozeß A testet lock Prozeß A findet lock = 0 setzt lock =1 betritt krit. Abschn. nach Verlassen des krit. Abschn. setzt er lock wieder auf 0

- bevor der Prozeß lock setzen kann wird er unterbrochen durch Prozeß B, der prüft lock (lock ist immer noch 0), setzt lock auf 1 geht in den kritischen Abschn. Es erfolgt schließlich Umschalten auf Prozeß A, der hat lock bereits getestet, setzt lock auf 1.

→ Zwei Prozesse befinden sich gleichzeitig im krit. Abschn.

- Auch wenn zuerst lock gelesen und geprüft wird bevor lock gesetzt wird tritt Wettlaufbedingung ein, wenn Prozeß B lock ändert kurz nachdem Prozeß A zweite Prüfung vorgenommen hat.

3. Gegenseitiger Ausschluß Version 1
Abwechselnder Zutritt

- Dijkstra´s schrittweise Entwicklung von Dekker´s Lösung
- die meisten Fehler der parallelen Programmierung aufzeigen

Problem des gegenseitigen Ausschlusses:

- Zwei Prozesse führen in einer Endlosschleife zwei Abschnitte aus: kritische Abschnitte (critical_section) und nicht kritische Abschnitte (non_critical_section). Die Ausführungen von critical_section1 und critical_section2 dürfen sich nicht überlappen.

Methode:

- Protokolliglu, indem eine Tafel steht
- Eingang so schmal u. Iglu so klein, daß nur eine Person im Iglu sein kann
- Auf der Tafel steht die Nummer des Prozesses, der als nächster an der Reihe ist in den krit. Abschnitt zu gehen
- Prozeß, der in krit. Abschnitt möchte krabbelt ins Iglu. Steht auf der Tafel seine Nummer, so beginnt er glücklich seinen kritischen Abschnitt, wenn nicht verläßt er das Iglu und wartet auf den anderen Prozeß
- Warten:
  - im Idealfall "schlafen" (Nicht möglich mit gewöhnlicher Programmiersprachen)
  - "Runden laufen ums Iglu" (busy waiting, aktives warten)
- Nach dem Beenden des krit. Abschnittes schreibt der Prozeß die Nummer des anderen Prozesses auf die Tafel.

```
/* Gegenseitiger Ausschluß Version 1
   Abwechselnder Zutritt */

#define TRUE 1

/* Prozeß 1 */                          /* Prozeß 2 */

while(TRUE) {                           while (TRUE) {
        while (turn==2) /* warten */            while (turn==1) /* warten */
        critical_section();                     critical_section();
        turn=2;                                 turn=1;
        noncritical_section();                  noncritical_section();
        }                                       }
```

- Grundidee: (2 Prozesse)
  - Der 1. Prozeß gibt nach Austritt aus dem kritischen Abschnitt diesen für Prozeß 2 frei

turn =1 Prozeß 1 darf	turn =2 Prozeß 2 darf

  - turn anfangs auf 1
    turn legt fest wer in den kritischen Bereich darf

Prozeß 1 findet turn = 1 → Prozeß 0 betritt kritischen Abschnitt	
	Prozeß 2 findet turn = 1 → Prozeß fragt turn ständig ab, bis turn 2 ist → aktives warten (busy waiting) → Vergeudung von CPU-Zeit

Ergebnis

1. Wenn ein Prozeß wesentlich langsamer als der andere ist, dann ist die Methode "Abwechselnder Zutritt" nicht geeignet

Prozeß 0	Prozeß 1
while (turn==2) /* warten */	while (turn==1)/* warten */
critical_section();	critical_section();
turn=2;	turn=1;
noncritical_section();	noncritical_section();

2. vermeidet Wettkampfbedingungen, aber keine ernsthafte Lösung

Ben Ari:

- Erfüllt Anforderungen des gegenseitigen Ausschlusses
- Verklemmung ist auch nicht möglich. Es ist unmöglich, daß beide Prozesse gleichzeitig in while-Schleife stecken
- Erfüllt Anforderungen an die Korrektheit von parallelen Programmen
- Erfüllt nicht die Anforderungen der Abstraktion

→ Prozesse sind nicht eng verbunden
→ Das Recht zum Zutritt des kritischen Abschnittes geht immer von einem Prozeß zum anderen über

a) z.B. Prozeß 1 muß 100mal/ Tag den kritischen Abschnitt ausführen und Prozeß 2 nur 1mal/ Tag
   → Prozeß 1 wird gezwungen mit Geschwindigkeit von Prozeß 2 (1mal/ Tag) zu arbeiten.
b) Prozeß1 wartet auf Prozeß2 um in seinen kritischen Abschnitt zu gelangen. Wenn Prozeß2 gekillt wird, kann Prozeß1 auch nicht mehr weiterarbeiten

4. Gegenseitiger Ausschluß Version 2

- Jeder Prozeß erhält seinen eigenen Schlüssel für seinen kritischen Abschnitt
- Jeder Prozeß kann seinen krit. Abschn. betreten, selbst wenn er von einem Bären gefressen wird.
- Für jeden Prozeß gibt es eigenen Iglu
- Prozeß kann die Variable, die dem anderen Prozeß gehört nur lesen nicht beschreiben (leichtere Implementierung)

(Vgl. Version1 dort durfte *turn* von beiden Prozessen gelesen und beschrieben werden)

Methode:

- Will P_1 in den krit. Abschn. eintreten, so klettert er in das Iglu von P_2 bis er $c_2 = 1$ vorfindet, dann ist P_2 nicht im krit. Abschn.
- Hat P_1 $c_2 = 1$ vorgefunden, setzt er seine Tafel auf 0 (c_1)
- Nach Beendigung des krit. Abschn. setzt er $c_1 = 1$, damit zeigt er P_2, daß er nicht im krit. Abschn. ist

Ergebnis:

- Programm erfüllt nicht die Sicherheitserfordernisse des gegenseitigen Ausschlusses
- P_1 u. P_2 können sich gleichzeitig in ihren krit. Abschnitten befinden

	c_1	c_2
Initialwert	1	1
P_1 prüft c_2	1	1
P_1 prüft c_2	1	1
P_1 setzt c_1	0	1
P_2 setzt c_2	0	0
P_1 beginnt krit. Abschn.	0	0
P_2 beginnt krit. Abschn.	0	0

Analyse

a) P_1 hat c_2 geprüft und fordert sein Recht den krit. Abschn. zu betreten
b) c_1 soll Anzeigen, daß P_1 im krit. Abschn. ist, aber zwischen while und der 1. Anweisung kann beliebig lange Zeit liegen

5. Gegenseitiger Ausschluß Version 3

- siehe Analyse (b) zu Version 2
- Korrektur hier : Zuweisung vor der while-Schleife
- c_1 zeigt bereits vor der while-Schleife an, daß sich P_1 im krit. Abschn. befindet

→ führt zur V E R K L E M M U N G

	c_1	c_2
Initialwert	1	1
P_1 setzt c_1	0	1
P_2 setzt c_2	0	0
P_1 prüft c_2	0	0
P_2 prüft c_1	0	0

- Prüfung wird unendlich lange fortgesetzt
- aber gegenseitiger Ausschluß gewährleistet, d.h. wenn P_1 in $critical\_section_1$, dann P_2 nicht in $critical\_section_2$

Analyse:
a) P_1 bestand auf seinem Recht, seinen krit. Abschn. zu beginnen, nachdem er einmal c_1 auf 0 gesetzt hatte
b) Setzen von c_1 vor der Prüfung c_2 bewirkt Verklemmung

6. Gegenseitiger Ausschluß Version 4

- Prozeß gibt temporär seine Absicht auf, in den krit. Abschnitt einzutreten
- Gibt so dem anderen Prozeß eine Chance

Methode
- P_1 betritt seinen Iglu schreibt 0
- P_1 prüft dann Iglu von P_2
- wenn dort 0 steht, kehrt er in sein Iglu zurück, um 0 zu löschen → setzt auf 1
- nach ein paar Runden ums Iglu setzt er seine Tafel wieder auf 0, versucht nochmal

Ergebnis
- Es ist möglich, daß keiner in seinen krit. Abschn. eintritt

	c_1	c_2
Initialwert	1	1
P_1 setzt c_1	0	1
P_2 setzt c_2	0	0
P_1 prüft c_2	0	0

P_2 prüft c_1	0	0
P_1 setzt c_1	1	0
P_2 setzt c_2	1	1
P_1 setzt c_2	0	1
P_2 setzt c_1	0	0

7. Dekker´sche Lösung

- Kombination aus Versuch 1 +4
- Vers. 1 war abwechselnder Zutritt ⇒ Schlüssel zum kritischen Abschnitt konnte verloren gehen
- Vers. 4 separate Schlüssel führten zum unendlichen Aufschub unter den Prozessen (Aussperrung)

⇒ Dekker´s Lösung löst das Problem der Aussperrung, indem das Recht "Bestehen" auf den Eintritt in den krit. Abschn. übergeben wird.

- Jeder Prozeß hat eigenes Iglu, so daß er fortfahren kann, selbst wenn der andere Prozeß durch Polarbären terminiert.
- weitere Annahme, daß kein Prozeß in seinem krit. Abschnitt gekillt wird

Methode:

- Schiedsrichter-Iglu mit Variable "turn"
- "turn" zeigt an, wer das Recht erhält
- wenn
  P_1 0 auf c_1 schreibt und feststellt, daß
  P_2 0 auf c_2 geschrieben hat, so befragt er den Schiedsrichter
- wenn Schiedsrichter 1 zeigt ⇒ P_1 dran
  daraufhin befragt er zyklisch Iglu von P_2
- P_2 bemerkt, daß er an der Reihe ist zu warten und schreibt 1 auf seine Tafel
- das bemerkt evtl. P_1 ⇒ P_1 betritt krit. Abschnitt
- wenn P_1 krit. Abschnitt beendet hat, setzt er c_1 auf 1 und Schiedsrichteriglu auf 2

→ P_2 wird aus seiner inneren Schleife befreit.

→ Wert von turn beeinflußt die Entscheidung zum Eintritt in den krit. Abschnitt nicht

/* Algorithmus von Dekker */

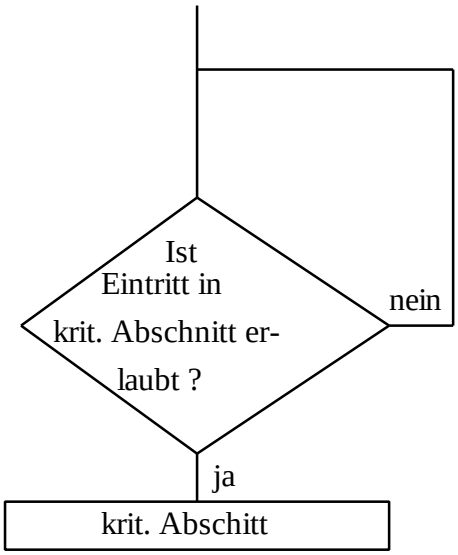
```
int c1=1;
int c2=1;
int turn=1;

/* Prozeß 1 */

while (TRUE) {
        c1=0;
        while (c2==0) {
                if (turn==2){
                        c1=1;
                        while (turn==2) /*warten */;
                        c1=0;
                        }
                }
        critical_section();
        turn=2;
        c1=1;
        noncritical_section();
        }

/* Prozeß 2 */

while (TRUE) {
        c2=0;
        while (c1==0) {
                if (turn==1){
                        c2=1;
                        while (turn==1) /*warten */;
                        c2=0;
                        }
                }
        critical_section();
        turn=1;
        c2=1;
        noncritical_section();
        }
```

8. Lösung von Peterson

- Dekker schlug erste Softwarelösung für Problem des gegenseitigen Ausschlusses vor

- Lösung von Peterson korrekt, aber Nachteil aktives Warten

- Prinzip:
  Wenn Prozeß seinen krit. Abschnitt betreten will, so prüft er, ob dies erlaubt ist, wenn nicht verharrt er in Schleife



[Erläuterung zu den C-Bestandteilen

E1 && E1
- wenn E1 ≠ 0 wahr und E2 ≠ 0 wahr, dann ist Ergebnis wahr (1) sonst falsch (0)
- Operatoren werden von links nach rechts ausgedehnt. Wenn E1 nach der Auswertung 0 (falsch) ist, so ergibt E1 && E2 0 (falsch). E2 wird nicht ausgewertet.

E1 == E2
- wenn der Wert von E1 gleich dem Wert von E2 ist, so ist das Ergebnis wahr (1), sonst falsch (0)

]

```
/* Lösung von Peterson für wechselseitigen Ausschluß */

#define FALSE 0
#define TRUE  1
#define N     2             /* Anzahl der Prozesse */

int turn;                   /* Wer ist dran ? */
int interested[N];          /* Alle Werte mit 0 (FALSE) initialisiert */

enter_region(process)       /* betreten des kritischen Abschnittes */
int process;                /* Prozeßnummer: 0 oder 1 */
{
        int other;                      /* Nummer des anderen Prozesses */

        other=1-process;
        interested[process]=TRUE;
                                        /*Zeigt, daß er interessiert ist */
        turn=process;       /* Setze Flagge */
        while(turn == process && interested[other] == TRUE)
                ;                       /* Nullanweisung */
}

leave_region(process)       /* Verlassen des krit. Abschnittes */
int process;
{
        interested[process] = FALSE;
                                        /* zeigt das Verlassen des
                                        kritischen Abschnittes an*/
}
```

- Jeder Prozeß ruft enter_region auf, vor Eintritt in den kritischen Abschnitt mit Parameter eigene Prozeßnummer

→ Nach Aufruf von enter_region, wird Prozeß gezwungen zu warten, bis sicherer Eintritt möglich

- nach Austritt aus krit. Abschnitt wird leave_region aufgerufen

Methode:

- Prozeß 0 will krit. Abschn. betreten
- ruft enter_region(0) auf
- Interesse von Prozeß 0 wird durch Eintrag im Feld interested[0] festgehalten
- da Prozeß1 nicht interessiert ist, wird enter_region verlassen

⇒ Bei gleichzeitigem Aufruf von enter_region bleibt das zuletzt geänderte turn erhalten, d.h. wenn P1 zuletzt turn = 1 setzt

→ interested[0] = True

→ P0 durchläuft Schleife 0-mal

→ P1 bleibt hängen

→ Ansatz verschwendet CPU-Zeit, unerwünschte Nebeneffekte möglich

Bsp.
- 2 Prozesse H (hohe Prio.) und L (niedr. Prio)
- H läuft immer wenn bereit (Regel des Schedulers)
- L in krit. Abschnitt, H wird bereit, aktives warten von H beginnt
- L bekommt den Prozessor nicht wieder (Prio.) und kann seinen kritischen Abschnitt nicht beenden

→ H wartet für immer

9. Test and Set Lock

- Liest den Inhalt eines Speicherwortes in ein Register und speichert dann einen von 0 verschiedenen Wert an diese Stelle zurück
- Operation des Lesens und Schreibens ist unteilbar
- kein anderer Prozeß kann zugreifen während der Dauer der Durchführung des Befehls
- Zum Kennzeichnen des Besetztzustandes des Busses benutz man Flag

Methode

- Wenn flag=0, darf jeder Prozeß Flag setzen

→ Eintritt in krit. Abschn.

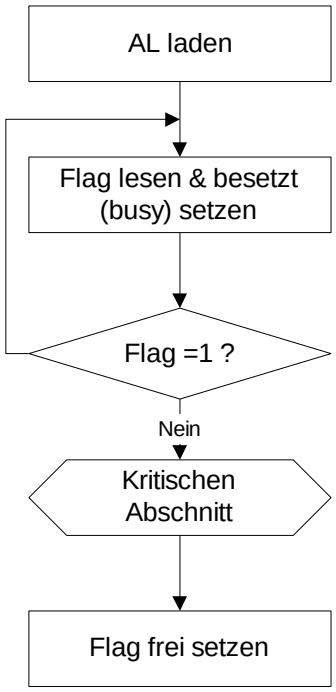
- Bei Beenden (Austritt aus krit. Abschn.) mit normalen *mov*

Allgemein:

```
enter_region:                          leave_region:
        tsl     register,flag                  mov    flag,0
        cmp     register,0      ;war flag 0 ?  ret
        jnz     enter_region
        ret
```

8086:
Lock: Schützt die Abarbeitung des nächsten Befehls vor Unterbrechungen durch Busanforderungen

```
enter_region:
        mov  al,1
wait:   lock xchg flag,al      ; Wenn flag gleich 0, dann wird flag auf 1
                               ; gesetzt, um den Besetzzustand anzuzeigen.
                               ; Wenn flag gleich 1, ändert sich
                               ; im Prinzip nichts- flag bleibt 1.
                               ; Routine wartet in ihrer Endlosschleife bis
                               ; ein anderes Programm dieses auf 0 setzt
                               ; (S. Routine leave_region)
        test al,al
        jnz  wait              ; Sprung auf Marke enter_region geht auch.
        ret
```




```
mov al,1

wait: lock xchg al,flag

test al,al
jnz wait

; ein kritischer Abschnitt

mov flag,0
```