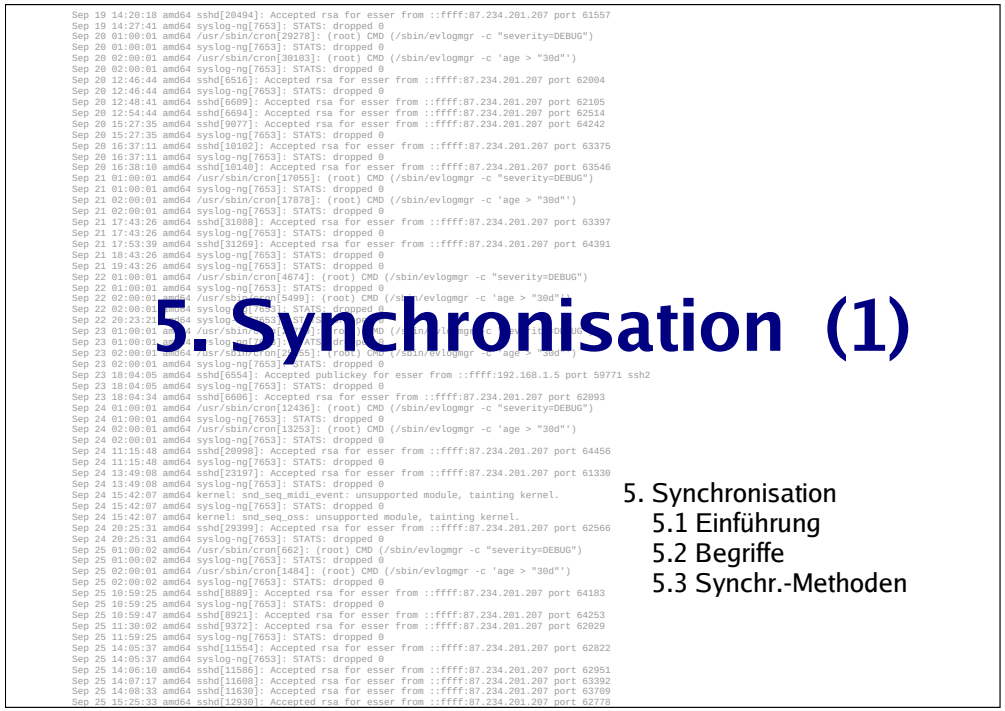

5. Synchronisation (1)

5. Synchronisation
   5.1 Einführung
   5.2 Begriffe
   5.3 Synchr.-Methoden

Einführung (1)

- Es gibt Prozesse (oder Threads oder Kernel-Funktionen) mit gemeinsamem Zugriff auf bestimmte Daten, z. B.
  - Threads des gleichen Prozesses: gemeinsamer Speicher
  - Prozesse mit gemeinsamer Memory-Mapped-Datei
  - Prozesse / Threads öffnen die gleiche Datei zum Lesen / Schreiben
  - SMP-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

Einführung (2)

- Synchronisation: Probleme mit „gleichzeitigem“ Zugriff auf Datenstrukturen
- Beispiel: Zwei Prozesse erhöhen einen Zähler

```
erhoehe_zaehler( )
{
  w=read(Adresse);
  w=w+1;
  write(Adresse,w);
}
```

Ausgangssituation: w=10

P1:	P2:
w=read(Adresse); // 10	
w=w+1; // 11	
	w=read(Adresse); // 10
	w=w+1; // 11
	write(Adresse,w); // 11
write(Adresse,w); // 11 !!	

Ergebnis nach P1, P2: w=11 – nicht 12!

Einführung (3)

- Gewünscht wäre eine der folgenden Reihenfolgen:

Ausgangssituation: w=10

P1:	P2:
w=read(Adr); // 10	
w=w+1; // 11	
write(Adr,w); // 11	
	w=read(Adr); // 11
	w=w+1; // 12
	write(Adr,w); // 12

Ergebnis nach P1, P2: w=12

Ausgangssituation: w=10

P1:	P2:
	w=read(Adr); // 10
	w=w+1; // 11
	write(Adr,w); // 11
w=read(Adr); // 11	
w=w+1; // 12	
write(Adr,w); // 12	

Ergebnis nach P1, P2: w=12

Einführung (4)

- Ursache: `erhoehe_zaehler()` arbeitet nicht **atomar**:
  - Scheduler kann die Funktion unterbrechen
  - Funktion kann auf mehreren CPUs gleichzeitig laufen
- Lösung: Finde alle Code-Teile, die auf gemeinsame Daten zugreifen, und stelle sicher, dass immer nur ein Prozess auf diese Daten zugreift (gegenseitiger Ausschluss, mutual exclusion)

Einführung (5)

- Analoges Problem bei Datenbanken:

```
exec sql CONNECT ...
exec sql SELECT kontostand INTO $var FROM KONTO
         WHERE kontonummer = $knr
$var = $var - abhebung
exec sql UPDATE Konto SET kontostand = $var
         WHERE kontonummer = $knr
exec sql DISCONNECT
```

  Bei parallelem Zugriff auf gleichen Datensatz kann es zu Fehlern kommen
- Definition der (Datenbank-) **Transaktion**, die u.a. **atomar und isoliert** erfolgen muss

Einführung (6)

Race Condition:

- Mehrere parallele Threads / Prozesse nutzen eine gemeinsame Ressource
- Zustand hängt von Reihenfolge der Ausführung ab
- Race: die Threads liefern sich „ein Rennen“ um den ersten / schnellsten Zugriff

Einführung (7)

Warum Race Conditions vermeiden?

- Ergebnisse von parallelen Berechnungen sind nicht eindeutig (d. h. Potenziell falsch)
- Bei Programmtests könnte (durch Zufall) immer eine „korrekte“ Ausführreihenfolge auftreten; später beim Praxiseinsatz dann aber gelegentlich eine „falsche“.
- Race Conditions sind auch Sicherheitslücken

Einführung (8)

Race Condition als Sicherheitslücke

- Wird von Angreifern genutzt
- Einfaches Beispiel:

```
read(command)
f=open("/tmp/script","w")
write(f,command)
f.close()
chmod("/tmp/script","a+x")
system("/tmp/script")
```

Angreifer ändert Dateiinhalt vor dem chmod; Programm läuft mit Rechten des Opfers

Einführung (9)

- Idee: Zugriff via Lock auf einen Prozess (Thread, ...) beschränken:

```
erhoehe_zaehler( ) {
  flag=read(Lock);
  if (flag == LOCK_UNSET) {
    set(Lock);
    /*  Anfang des „kritischen Bereichs“  */
    w=read(Adresse);  w=w+1;
    write(Adresse,w);
    /*  Ende des „kritischen Bereichs“  */
    release(Lock);
  };
}
```

- Problem: Lock-Variable nicht geschützt

Einführung (10)

- Nicht alle Zugriffe auf Daten sind problematisch:
  - Gleichzeitiges Lesen von Daten stört nicht
  - Prozesse, die „disjunkt“ sind (d.h.: die keine gemeinsamen Daten haben) können ohne Schutz zugreifen
- Sobald mehrere Prozesse/Threads/... gemeinsam auf ein Objekt zugreifen – und mindestens einer davon schreibend –, ist das Verhalten des Gesamtsystems **unvorhersehbar** und **nicht reproduzierbar**.

Inhaltsübersicht: Synchronisation

Kritische Abschnitte (1)

- Programmteil, der auf gemeinsame Daten zugreift
  - Müssen nicht verschiedene Programme sein: auch mehrere Instanzen des gleichen Programms!
- Block zwischen erstem und letztem Zugriff
- Nicht den Code schützen, sondern die Daten
- Formulierung: kritischen Bereich „betreten“ und „verlassen“

Kritische Abschnitte (2)

- Anforderung an parallele Threads:
  - Es darf maximal ein Thread gleichzeitig im kritischen Abschnitt sein
  - Kein Thread, der außerhalb kritischer Bereiche ist, darf einen anderen blockieren
  - Kein Thread soll ewig auf das Betreten eines kritischen Bereichs warten
  - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen krit. Bereichen und blockieren sich gegenseitig)

Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: mutual exclusion, kurz: mutex)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmier-fehlern

Programmtechnische Synchr. (1)

1. Versuch: Lock-Variable (wie in Einführung)

- Lock-Variable auf *false* initialisiert
- Prozess, der krit. Bereich betreten will, prüft lock==*false* – wenn Bedingung erfüllt ist:
  - lock=*true* setzen,
  - Bereich betreten und wieder verlassen
  - lock=*false* (zurück)setzen

```
while ( lock ) {
  /* warten */
} ;
lock=true;
kritischer_bereich();
lock=false;
```

- Verschiebt Problem nur auf die Lock-Variable

Programmtechnische Synchr. (2)

2. Versuch: Nächsten Prozess speichern

- Lock-Variable *turn* legt fest, welcher Prozess als nächster den krit. Bereich betreten darf:

```
while (true) {
  while (turn != 1) {
    /* warten */
  };
  kritischer_bereich();
  turn=2;
}
```

```
while (true) {
  while (turn != 2) {
    /* warten */
  };
  kritischer_bereich();
  turn=1;
}
```

- Verhindert Race Conditions
- Aber: kritischer Bereich kann nur abwechselnd betreten werden

Programmtechnische Synchr. (3)

3. Versuch: Für jeden Thread separate Variable, die „Thread ist in krit. Bereich“ anzeigt

```
while (true) {
  C1=true;
  while (C2) {
    /* wait */
  };
  kritischer_bereich();
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    /* wait */
  };
  kritischer_bereich();
  C2=false;
}
```

- Verhindert Race Conditions
- Deadlock tritt auf, wenn beide gleichzeitig den kritischen Bereich betreten wollen

Programmtechnische Synchr. (4)

4. Versuch (Dekker): Kombination aus Lock-Variablen und wechselnder Reihenfolge

```
while (true) {
  C1=true;
  while (C2) {
    if (turn != 1) {
      C1=false;
      while (turn != 1) {
        /* wait */
      };
      C1=true;
    };
  };
  kritischer_bereich();
  turn=2;
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    if (turn != 2) {
      C2=false;
      while (turn != 2) {
        /* wait */
      };
      C2=true;
    };
  };
  kritischer_bereich();
  turn=1;
  C2=false;
}
```


Programmtechnische Synchr. (5)

Alternative: Petersons Algorithmus

```
C1=true;
turn=2;
while (C2 && turn==2)
  /* warten */;
kritischer_abschnitt();
C1=false;
```

```
C2=true;
turn=1;
while (C1 && turn==1)
  /* warten */;
kritischer_abschnitt();
C2=false;
```

Programmtechnische Synchr. (6)

Petersons Algorithmus –
gegenseitiger Ausschluss gewährt:

- Wenn P_1 C_1 auf *true* setzt, kann P_2 seinen kritischen Bereich nicht mehr betreten
- War P_2 schon im kritischen Bereich, dann war C_2 schon *true*, d.h., P_1 durfte nicht in seinen kritischen Bereich

Programmtechnische Synchr. (7)

Petersons Algorithmus –
keine gegenseitige Blockade:

Angenommen, P1 ist in der While-Schleife blockiert, d. h.: C2=true und turn=2 (P1 kann den krit. Bereich betreten, wenn eine der Bedingungen nicht mehr gilt, also entweder C2=false oder turn=1 wird)

Dann nur 2 Möglichkeiten:

- P2 wartet auf Einlass in den krit. Bereich -> das kann nicht sein, denn mit turn=2 darf P2 in seinen kritischen Bereich
- P2 nutzt wiederholt den krit. Bereich, monopolisiert Zugang zu ihm -> das kann auch nicht sein, weil P2 vor dem Betreten die turn-Variable auf 1 setzt (und damit P1 den Vortritt lassen würde)