



**Hochschule
Augsburg** University of
Applied Sciences

Vorlesung: Betriebssysteme

Interprozesskommunikation und
Synchronisation

Prof. Dr. Lothar Braun, Dr.-Ing. Volodymyr Brovkov
Sommersemester 2024

Notwendigkeit der Synchronisation

Mechanismen zur Synchronisation

Interprozesskommunikation in POSIX-Systemen

Deadlocks

Notwendigkeit der Synchronisation

Mechanismen zur Synchronisation

Interprozesskommunikation in POSIX-Systemen

Deadlocks

- **Threads und Prozesse können gleichzeitig Ressourcen verwenden**
 - Threads teilen sich den gleichen Speicher
 - Prozesse können Interprozesskommunikation verwenden um Ressourcen zu teilen
- **Synchronisation notwendig für**
 - A) Austausch von Informationen**
 - B) Zugriff auf gleiche Ressourcen**
 - Geräte (z.B. Drucker), Daten, Speicher
 - C) Abhängigkeiten von Daten**
 - Beispiel: Prozesse benötigen Ergebnisse anderer Prozesse

```
Messages queue[];  
int tail;
```

Thread A

```
...  
next = tail;  
queue[next] = "Result A"  
tail = next + 1;  
...
```

Thread B

```
...  
next = tail;  
queue[next] = "Result B"  
tail = next + 1;  
...
```

- Probleme können unter bestimmten Umständen zur Laufzeit auftreten

- Wenn kritische Abschnitte eines Tasks unterbrochen werden
- Wenn Unterbrechung zu inkonsistenten Zuständen führen kann
- Dies betrifft Operationen die *atomar* ausgeführt werden sollten

Empfehlung

Fehler bei der Synchronisation sind in der Praxis schwer zu erkennen, da sie in der Regel schwer reproduzierbar sind. Deshalb:

- Notwendigkeit der Synchronisation beim Design berücksichtigen und **dokumentieren**
- Bekannte (und bewährte!) Standard-Mechanismen zur Lösung verwenden

Race Condition: Je nach zeitlichem Ablauf treten *unerlaubte Endergebnisse* auf

Kritisches Objekt: Objekt (z.B. Datenstruktur, Gerät, ...), auf das mehrere Tasks zugreifen

Kritischer Abschnitt: Code-Abschnitt, der auf ein kritisches Objekt zugreift

Deadlock: Task oder Gruppe von Tasks ist endgültig blockiert

Livelock: Task oder Gruppe von Tasks befindet sich in einer aktiven Warteschleife, die nicht mehr verlassen werden kann

- **Konkurrenz um Zugriff auf Daten**

- Mehrere Tasks wollen auf die selben Daten zugreifen
- Wechselseitiger Ausschluss notwendig

- **Erzeuger-Verbraucher Probleme**

- Ein oder mehrere Prozesse erzeugen Daten
- Ein oder mehrere Prozesse verarbeiten Daten
- Verbraucher müssen auf Ergebnisse warten und sich beim entnehmen der Daten synchronisieren

- **Leser-Schreiber Problem**

- Mehrere Tasks greifen auf Daten lesend zu
- Ein Prozess manipuliert Daten
- Während Daten manipuliert werden dürfen Tasks nicht lesend auf Daten zugreifen

Notwendigkeit der Synchronisation

Mechanismen zur Synchronisation

Interprozesskommunikation in POSIX-Systemen

Deadlocks

- **Konkurrenz um Zugriff auf Daten**

- Mehrere Tasks wollen auf die selben Daten zugreifen
- Wechselseitiger Ausschluss notwendig

- **Erzeuger-Verbraucher Probleme**

- Ein oder mehrere Prozesse erzeugen Daten
- Ein oder mehrere Prozesse verarbeiten Daten
- Verbraucher müssen auf Ergebnisse warten und sich beim entnehmen der Daten synchronisieren

- **Leser-Schreiber Problem**

- Mehrere Tasks greifen auf Daten lesend zu
- Ein Prozess manipuliert Daten
- Während Daten manipuliert werden dürfen Tasks nicht lesend auf Daten zugreifen

```
Messages queue[];  
int tail;  
mutex_t queueLock;
```

Thread A

```
...  
lock(&queueLock);  
next = tail;  
queue[next] = "Result A"  
tail = next + 1;  
unlock(&queueLock);  
...
```

Thread B

```
...  
lock(&queueLock);  
next = tail;  
queue[next] = "Result B"  
tail = next + 1;  
unlock(&queueLock);  
...
```

Von Mutex *lock/unlock* zu leisten:

- **Maximal ein Prozess in kritischem Abschnitt**
- **Keine unnötige Blockade**
 - Prozess muss sofort in kritischen Abschnitt gelassen werden wenn Abschnitt leer
- **Keine Annahme über Anzahl der Prozesse oder Ablaufgeschwindigkeit**

Von Entwickler der Anwendung zu leisten:

- **Kooperation der Prozesse**
 - Ein Prozess darf nur endliche Zeit in kritischem Abschnitt aufhalten

- **Lock/Unlock mit einfacher Variable:**

```
void lock(int* mutex) {  
    while (*mutex == true) {}  
    *mutex = true;  
}
```

```
void unlock(int* mutex) {  
    *mutex = false;  
}
```

Prozess 0

```
while (turn != 0) {};  
// Kritischer Abschnitt  
turn = 1;
```

Prozess 1

```
while (turn != 1) {};  
// Kritischer Abschnitt  
turn = 0;
```

- **Vorteil:** Korrekter wechselseitiger Ausschluss
- **Nachteil:** Beide Prozesse können nur gleich häufig in kritischen Abschnitt

Prozess 0

```
int flag[2] = (false, false)
...
while (flag[1]) /* nichts tun */;
flag[0] = true;
// Kritischer Abschnitt
flag[0] = false;
```

Prozess 1

```
int flag[2] = (false, false)
...
while (flag[0]) /* nichts tun */;
flag[1] = true;
// Kritischer Abschnitt
flag[1] = false;
```

- **Vorteil:** Wenn ein Prozess häufiger in den kritischen Abschnitt will, dann kann er
- **Nachteil:** Garantiert keinen wechselseitigen Ausschluss (wenn beide gleichzeitig in while-Schleife sind)

Prozess 0

```
flag[0] = true;  
while (flag[1]) /* nichts tun */;  
// Kritischer Abschnitt  
flag[0] = false;
```

Prozess 1

```
flag[1] = true;  
while (flag[0]) /* nichts tun */;  
// Kritischer Abschnitt  
flag[1] = false;
```

- **Vorteil:** Garantiert wechselseitigen Ausschluss
- **Nachteil:** Führt *Deadlock* ein wenn beide Prozesse flag setzen bevor while-Schleife ausgeführt wird

Prozess 0

```
flag[0] = true;
while (flag[1]) {
    flag[0] = false;
    // Verzögerung
    flag[0] = true;
}
// Kritischer Abschnitt
flag[0] = false;
```

Prozess 1

```
flag[1] = true;
while (flag[0]) {
    flag[1] = false;
    // Verzögerung
    flag[1] = true;
}
// Kritischer Abschnitt
flag[1] = false;
```

- **Vorteil:** Garantiert wechselseitigen Ausschluss
- **Nachteil:** Kann potentiell zu *Live*lock führen

```
int turn;  
int interested[2] = {0, 0}  
  
void lock(int me) { // me: 0 oder 1  
    int other = 1 - me;  
    interested[me] = 1;  
    turn = other;  
    while (turn == other && interested[other] == 1) {}  
}  
  
void unlock(int me) {  
    interested[me] = 0;  
}
```

- **Grundproblem: Kritische Abschnitte können unterbrochen werden**
- **Deaktivieren von Interrupts**
 - Verhindert Prozesswechsel durch deaktivieren von (Timer-)Interrupts
 - Nur wirksam innerhalb einer CPU bei Systemen mit mehreren Prozessoren
 - Sperre sollte nur kurz sein, da sonst andere Interrupts verpasst werden
- **Spezielle *atomare* Befehle**
 - Befehle wie *test-and-set* oder *exchange*
 - Hardware garantiert: Lesen und schreiben von Werten wird nicht unterbrochen
 - Hinweis: Befehle wie “a++” in C oder “inc [addr]” (Assembler) sind **nicht** *atomar*.

Semantik des Befehls (Achtung: Benötigt in Realität Hardware-spezifische Befehle)

```
int test_and_set(int* x) {  
    if (*x == false) {  
        *x = true;  
        return false;  
    } else {  
        return true;  
    }  
}
```

Implementierung von lock/unlock

```
void lock(int* mutex) {  
    while (test_and_set(mutex) == true);  
}  
  
void unlock(int* mutex) {  
    *mutex = false;  
}
```

Semantik des Befehls (Achtung: Benötigt in Realität Hardware-spezifische Befehle)

```
int exchange(int* x, int* y) {  
    int tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

Implementierung von lock/unlock

```
void lock(int* mutex) {  
    int x = true;  
    while (*x == true) exchange(mutex, &x);  
}  
  
void unlock(int* mutex) {  
    *mutex = 0;  
}
```

- **Vorteile:**
 - korrekt, einfach & wenig Overhead
- **Nachteile:**
 - Vergeudung der Ressource CPU durch aktives Warten (“busy waiting”)
 - falls Prozess mit hoher Priorisierung aktiv wartet: andere Prozesse bekommen wenig CPU Zeit
- **Einsatzgebiete:**
 - mehrere Prozessoren
 - sehr kurze kritische Abschnitte
 - zur Implementierung von Mechanismen der Synchronisation

- **Zwei interne Operationen: *sleep* und *wakeup***
- **in *lock*:**
 - solange der kritische Abschnitt belegt ist (Mutex-Variable = 1):
 - versetze aktuellen Prozess in Zustand *blockiert* und schedule anderen Prozess (*sleep*)
 - setze Mutex Variable auf 1
- **in *unlock*:**
 - setze Mutex-Variable auf 0
 - Prüfe ob sich wartende Prozesse in der Warteschlange befinden
 - falls ja: aktiviere ältesten und setze in Zustand *laufbereit* (*wakeup*)

```
void lock(int* mutex) {  
    while (test_and_set(mutex) == true) {  
        sleep(mutex);  
    }  
}
```

```
void unlock(int* mutex) {  
    *mutex = false;  
    wakeup(mutex);  
}
```

- Warum funktioniert die Implementierung nicht zuverlässig?

- **Problem: Test- und Warte-Operation müssen *atomar* ausgeführt werden**

```
typedef ... cond_t; // Datenstruktur zur Verwaltung blockierter Prozesse

void cond_wait(cond_t *c, spinlock_t* m) {
    // WICHTIG: muss so implementiert sein, dass unlock/sleep atomar durchgefuehrt werden
    unlock(m);
    sleep(c);
    lock(m);
}

void cond_broadcast(cond_t *c) {
    wakeup(c);
}
```

```
struct mutex_t {
    int locked;           // Flag ob belegt
    spinlock_t spinlock;  // zum Schutz der Datenstruktur
    cond_t cond;          // zur Verwaltung der wartenden Prozesse
};

void mutex_lock(mutex_t* m) {
    spinlock_lock(&m->spinlock); // Zugriff auf m schuetzen
    while (m->locked) {
        cond_wait(&m->cond, &m->spinlock); // warten bis jemand m->locked auf 0 gesetzt hat
    }
    m->locked = 1;
    spinlock_unlock(&m->spinlock);
}

void mutex_unlock(mutex_t *m) {
    spinlock_lock(&m->spinlock); // Zugriff auf m schuetzen
    m->locked = 0;
    cond_broadcast(&m->cond);    // wartende Tasks aufwecken
    spinlock_unlock(&m->spinlock);
}
```

- **Mutex**
 - `pthread_mutex_init`, `pthread_mutex_destroy`
 - `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`
- **Condition (variable)**
 - `pthread_cond_init`, `pthread_cond_destroy`
 - `pthread_cond_broadcast`
 - `pthread_cond_wait`, `pthread_cond_timedwait`
- **Details: `man pthreads`**

- **Verallgemeinerung von Mutex-Variablen**
- **Ein Semaphore hat einen ganzzahligen nicht-negativen Wert**
 - Häufiger Einsatz: Anzeige der Anzahl freier Ressourcen
 - Wert == n bedeutet, dass noch n Elemente einer Ressource frei sind
- **Operationen**
 - `sem_wait()`
 - Falls Wert == 0: warte bis Wert > 0
 - Dann dekrementiere Wert und fahre fort
 - `sem_post()`
 - Inkrementiere Wert
 - Falls wert zuvor 0: Wecke wartende Tasks auf

```
Messages queue[];  
int tail;  
semaphore_t queueSem = 1;
```

Prozess A

```
...  
sem_wait(&queueSem);  
next = tail;  
queue[next] = "Result A"  
tail = next + 1;  
sem_post(&queueSem);  
...
```

Prozess B

```
...  
sem_wait(&queueSem);  
next = tail;  
queue[next] = "Result B"  
tail = next + 1;  
sem_post(&queueSem);  
...
```

- **Konkurrenz um Zugriff auf Daten**

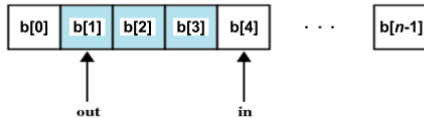
- Mehrere Tasks wollen auf die selben Daten zugreifen
- Wechselseitiger Ausschluss notwendig

- **Erzeuger-Verbraucher Probleme**

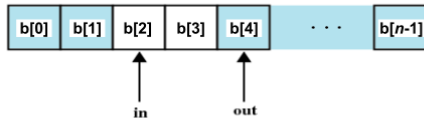
- Ein oder mehrere Prozesse erzeugen Daten
- Ein oder mehrere Prozesse verarbeiten Daten
- Verbraucher müssen auf Ergebnisse warten und sich beim entnehmen der Daten synchronisieren

- **Leser-Schreiber Problem**

- Mehrere Tasks greifen auf Daten lesend zu
- Ein Prozess manipuliert Daten
- Während Daten manipuliert werden dürfen Tasks nicht lesend auf Daten zugreifen



(a)



(b)

Erzeuger

```
while (true) {  
    v = produce();  
    while ((in+1) % N == out) {}  
    b[in] = v;  
    in = (in + 1) % N;  
}
```

Verbraucher

```
while (true) {  
    while (in == out) {}  
    w = b[out];  
    out = (out + 1) % N;  
    consume(w);  
}
```



```
#define N 100           // Groesse des Puffers
semaphore_t n = 0;      // Anzahl der Elemente im Puffer
semaphore_t m = 1;      // Mutex zum Schutz der Datenstruktur
```

Erzeuger

```
while (true) {
    v = produce();
    sem_wait(&m);
    b[in] = v;
    in = (in + 1) % N;
    sem_post(&m);
    sem_post(&n);
}
```

Verbraucher

```
while (true) {
    sem_wait(&n);
    sem_wait(&m);
    w = b[out];
    out = (out + 1) % N;
    sem_post(&m);
    consume(w);
}
```

```
#define N 100           // Groesse des Puffers
semaphore_t n = 0;      // Anzahl der Elemente im Puffer
semaphore_t e = N;      // Anzahl freier Plaetze
semaphore_t m = 1;      // Mutex zum Schutz der Datenstruktur
```

Erzeuger

```
while (true) {
    v = produce();
    sem_wait(&e);
    sem_wait(&m);
    b[in] = v;
    in = (in + 1) % N;
    sem_post(&m);
    sem_post(&n);
}
```

Verbraucher

```
while (true) {
    sem_wait(&n);
    sem_wait(&m);
    w = b[out];
    out = (out + 1) % N;
    sem_post(&m);
    sem_post(&e);
    consume(w);
}
```

- **Aufgaben**
 - Synchronisation
 - Austausch von Informationen
- **Operationen**
 - `send(target, message);`
 - `receive(target, message);`
 - Beide können blockierend sein oder nicht

A) Nur Empfänger blockiert

- Empfänger blockiert in *receive()*
- Sender kehrt sofort aus *send()* zurück (Botschaft gepuffert)
- Implementiert **Barriere**

B) Empfänger und Sender blockieren

- Sender blockiert in *send()* bis Empfänger *receive()* implementiert
- Empfänger blockiert in *receive()* bis Nachricht angekommen
- Implementiert: **Rendezvous**-Mechanismus

C) Keine Blockade

- *receive()* kehrt mit Fehler/Statuscode zurück

Erzeuger

```
message_t q_len , v;  
  
while (true) {  
    v = produce();  
    receive(consumer, &q_len);  
    send(consumer, v);  
}
```

- *receive()*: blockierend
- *send()*: nicht-blockierend

Verbraucher

```
message_t q_len , w;  
  
for (n = 0; n < N; n++)  
    send(producer, q_len);  
while (true) {  
    receive(producer, &w);  
    send(producer, q_len);  
    consume(w);  
}
```

- **Idee:**
 - Unterstützung durch automatische Synchronisation in einer Hochsprache
 - Technische Umsetzung wird Compiler überlassen (Mutex, Semaphore, ...)
- **Monitor == Software-Modul**
 - Lokale Daten + Prozeduren
 - Daten nur durch Prozeduren des Monitors zugänglich
 - Monitor-Prozeduren können jederzeit von mehreren Prozessen aufgerufen werden
 - *Ausgeführt* wird immer nur eine Prozedur gleichzeitig
 - Ausnahme: Prozedur kann “schlafen” und dabei Zugang für andere erlauben
- **Beispiel: Java**
 - Attribut *synchronized*

```
public class ProducerConsumer {
    static final int N = 100;           // buffer size
    static Producer p = new Producer(); // create new producer thread
    static Consumer c = new Consumer(); // create a new consumer thread
    static MonitorQueue queue = new MonitorQueue(); // create new monitor

    public static void main(String args[]) {
        p.start();
        c.start();
    }

    static class Producer extends Thread {
        public void run() {
            int item;
            while(true) {
                item = productItem();
                queue.insert(item);
            }
        }

        private int produceItem() { ... } // produce something
    }
}
```

```
...
static class Consumer extends Thread {
    public void run() {
        int item;
        while (true) {
            item = queue.remove();
            consumeItem(item);

        }
    }

    private void consumeItem(int item) { ... }
}
...
```



```
...
static class MonitorQueue {
    private int buffer[] = new int[N];
    private int count= 0, out = 0, in = 0;

    public synchronized void insert(int val) {
        while (count == N) doWait();
        buffer[in] = val;
        in = (in + 1) % N;
        count = count + 1;
        if (count == 1) notify();
    }

    public synchronized int remove() {
        int val;
        while (count == 0) doWait();
        val = buffer[out];
        out = (out + 1) & N;
        count = count - 1;
        if (count == N - 1) notify();
        return val;
    }

    private void doWait() {
        try { wait(); } catch (InterruptedException e) {};
    }
}
```

- **Konkurrenz um Zugriff auf Daten**

- Mehrere Tasks wollen auf die selben Daten zugreifen
- Wechselseitiger Ausschluss notwendig

- **Erzeuger-Verbraucher Probleme**

- Ein oder mehrere Prozesse erzeugen Daten
- Ein oder mehrere Prozesse verarbeiten Daten
- Verbraucher müssen auf Ergebnisse warten und sich beim entnehmen der Daten synchronisieren

- **Leser-Schreiber Problem**

- Mehrere Tasks greifen auf Daten lesend zu
- Ein Prozess manipuliert Daten
- Während Daten manipuliert werden dürfen Tasks nicht lesend auf Daten zugreifen

- **Naive Lösung: Lock für kritischen Abschnitt der nur einen Task erlaubt**
- **Problem: Mehrere Leser können sich gleichzeitig in Abschnitt aufhalten**
 - Exklusive Mutex führt zu unnötigem Aussperren anderer Leser
- **Gewünschte Locks:**
 - `read_lock, read_unlock`
 - `write_lock, write_unlock`

```
int num_readers = 0;
semaphore_t lock_reader = 1, lock_writer = 1;
```

```
void read_lock() {
    sem_wait(lock_reader);
    num_readers++;
    if (num_readers == 1)
        sem_wait(lock_writer);
    sem_post(lock_reader);
}
```

```
void read_unlock() {
    sem_wait(lock_reader);
    num_readers--;
    if (num_readers == 0)
        sem_post(lock_writer);
    sem_post(lock_reader);
}
```

```
void write_lock() {
    sem_wait(lock_writer);
}
```

```
void write_unlock() {
    sem_post(lock_writer);
}
```

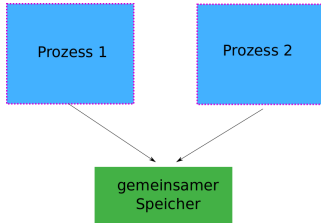
- **Vorige Lösung benachteiligt Schreiber:**
 - Solange noch Leser beschäftigt sind kann Schreiber nicht aktiv werden
 - Extremfall: Schreiber bekommt keinen Zugriff auf kritischen Abschnitt wenn immer Leser aktiv sind
 - Resultat: Schreiber *verhungert*
- **Lösung:**
 - Höhere Priorität für Schreiber
 - Sobald Schreiber bereit ist werden keine neuen Leser in Abschnitt gelassen

Notwendigkeit der Synchronisation

Mechanismen zur Synchronisation

Interprozesskommunikation in POSIX-Systemen

Deadlocks



- Speicher zweier Prozesse getrennt
- `int shmget(key_t key, size_t size, int shmflg);`
- `void *shmat(int shmid, const void *shmaddr, int shmflg);`
- `int shmdt(const void *shmaddr);`

- **Zugriff auf gemeinsamen Speicher muss synchronisiert werden**
- **Semaphoren:**
 - `sem_init`, `sem_destroy`, `sem_wait`, `sem_post`, `sem_getvalue`
- **Messages (Botschaften)**
 - `msgsnd`, `msgrcv`, ...

- **Pipes verhalten sich wie Dateien die direkte Kommunikation erlauben**
 - Synchronisierte Kommunikation zwischen Schreiber / Leser
- **Unnamed Pipes:**
 - Müssen vor `fork()` von Parent mit `pipe()` oder `pipe2()` erstellt werden
- **Named Pipe:**
 - Erstellen Datei im Dateisystem mit normalem Dateinamen
 - Jeder Prozess kann Pipe lesend / oder schreibend als Datei öffnen
- **Pipe auf der Kommandozeile:**
 - Beispiel: `$ ls | less`

- **Erlauben Kommunikation zwischen Prozessen, ähnlich Pipes**
- **Unterschiedliche Kommunikations-Domänen**
 - Unix-Domain-Sockets (AF_UNIX / AF_LOCAL): Lokale Kommunikation
 - IP-Sockets (AF_INET, AF_INET6): Kommunikation über Netzwerk
- **Verschiedene Socket-Typen:**
 - *SOCK_STREAM*: verbindungsorientiert (z.B. TCP)
 - *SOCK_DGRAM*: verbindungslos, ungesichert (z.B. UDP)
 - ...

- **Asynchrone Kommunikation an Prozesse als Signal eines Ereignisses**
 - Prozesse können Signale schicken wenn Rechte bestehen
 - Kernel kann Signale schicken
- **Reaktion von Signale:**
 - Prozess beenden oder anhalten
 - Ignorieren
 - Ausführen einer speziellen Funktion zur Behandlung des Signals

Signal	Bedeutung
SIGABRT	Prozess beenden und Core Dump erzeugen
SIGALRM	Zeitgeber löst Alarm aus
SIGFPE	Ein Gleitkommafehler ist aufgetreten (z.B. Division durch 0)
SIGHUP	Verbindung zum Kontrollterminal hängt
SIGILL	Ungültige CPU Instruktion
SIGQUIT	Endes des Programms durch die Tastatur
SIGKILL	Beenden des Prozesses (kann nicht gefangen oder ignoriert werden)
SIGPIPE	Geschlossene Pipe oder nach Schreiben auf Pipe ohne Leser
SIGSEGV	Illegaler Speicherzugriff
SIGTERM	Signal um Prozess zu bitten sich zu beenden
SIGINT	Interrupt von Tastatur (Strg + C)
SIGUSR1	Kann frei durch Anwendung definiert werden
SIGUSR2	Kann frei durch Anwendung definiert werden

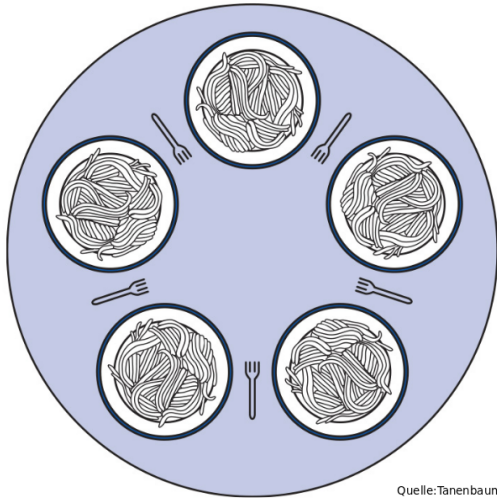
- Vollständige Dokumentation: [man 7 signal](#)

Notwendigkeit der Synchronisation

Mechanismen zur Synchronisation

Interprozesskommunikation in POSIX-Systemen

Deadlocks



Quelle: Tanenbaum

Naiver Ansatz:

```
#define N 5

void philosopher(int i) {
    while (1) {
        think();
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

Definition Deadlock

Eine Menge von Prozessen befindet sich in einem **Deadlock**-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, dass nur ein (anderer) Prozess aus der Menge auslösen kann.

- Kann in allen Situationen auftreten, in denen
 - *kritische Abschnitte* mit Locks gesichert werden
 - Ressourcen exklusiv registriert werden

Notwendige Bedingungen für einen (Ressourcen-)Deadlock:

A) Wechselseitiger Ausschluss

- Eine Ressource kann nur von einem Prozess gleichzeitig verwendet werden

B) Keine Verdrängung

- Ressourcen, die einem Prozess bewilligt wurden, können ihm nicht gewaltsam entzogen werden

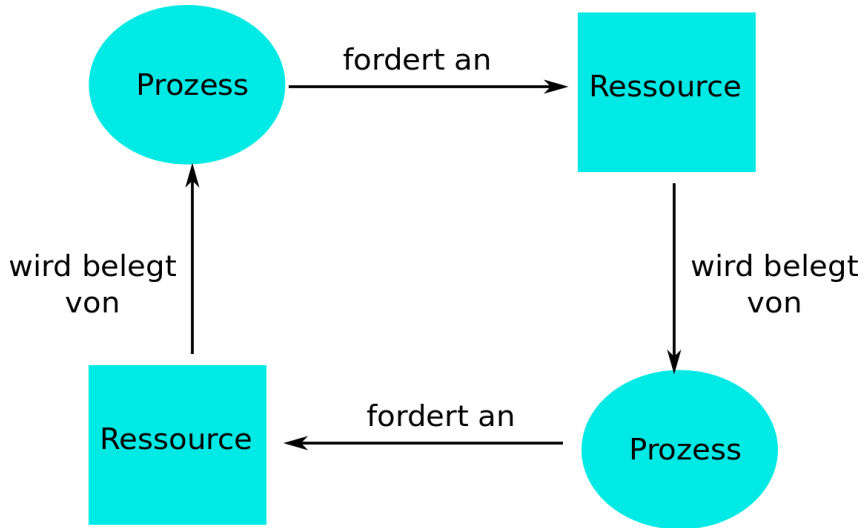
C) Hold-and-Wait

- Prozesse, die bereits Ressourcen reserviert haben, können weitere anfordern

D) Zyklische Wartebedingung

- Es gibt eine zyklische Kette von Prozessen, von denen jeder auf eine Ressource wartet, die vom nächsten Prozess in der Kette belegt wird





Code-Beispiel mit Deadlock

Prozess A

```
void process_A() {  
    sem_wait(&semA);  
    sem_wait(&semB);  
    use_both_resources();  
    sem_post(&semB);  
    sem_post(&semA);  
}
```

Prozess B

```
void process_A() {  
    sem_wait(&semB);  
    sem_wait(&semA);  
    use_both_resources();  
    sem_post(&semA);  
    sem_post(&semB);  
}
```

Code-Beispiel ohne Deadlock

Prozess A

```
void process_A() {  
    sem_wait(&semA);  
    sem_wait(&semB);  
    use_both_resources();  
    sem_post(&semB);  
    sem_post(&semA);  
}
```

Prozess B

```
void process_A() {  
    sem_wait(&semA);  
    sem_wait(&semB);  
    use_both_resources();  
    sem_post(&semB);  
    sem_post(&semA);  
}
```

A) “Vogel-Strauß-Algorithmus”

- nichts unternehmen und auf gute Programmierung hoffen
- **Ein Betriebssystem kann keine allgemeine, automatische Lösung zum Verhindern von Deadlocks zur Verfügung stellen**

B) Deadlocks erkennen und auflösen

- z.B. beteiligte Prozesse beenden und Ressourcen wieder freigegeben

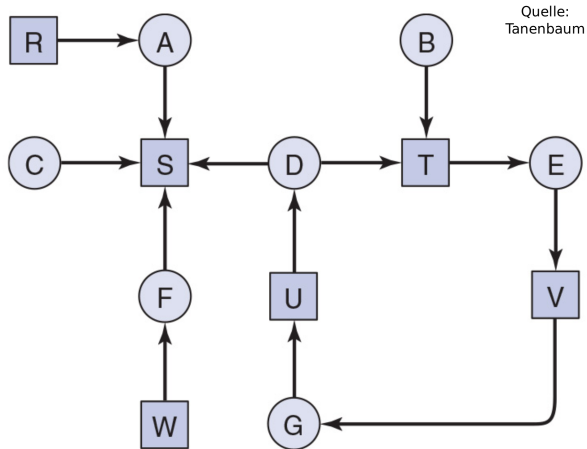
C) Deadlocks ausweichen

- Ressource erst zuteilen, wenn kein *unsicherer* Zustand entstehen kann

D) Deadlocks ausschließen

- Konventionen einhalten, die Deadlocks prinzipiell unmöglich machen

Schritt 1: Erkennung von Zyklen in Graphen



- **Temporäre Unterbrechung**

- Einem Prozess wird die Ressource zeitweise entzogen
- Prozess wird solange angehalten
- Problem: Darf die Ressource weggenommen werden?
 - Speicher → kein Problem
 - Drucker, Scanner, Datenbank → ???

- **Wiederholung (Rollback)**

- Ein Prozess wird in früheren Zustand zurückgesetzt (Checkpoint)
- Problem: Seiteneffekte

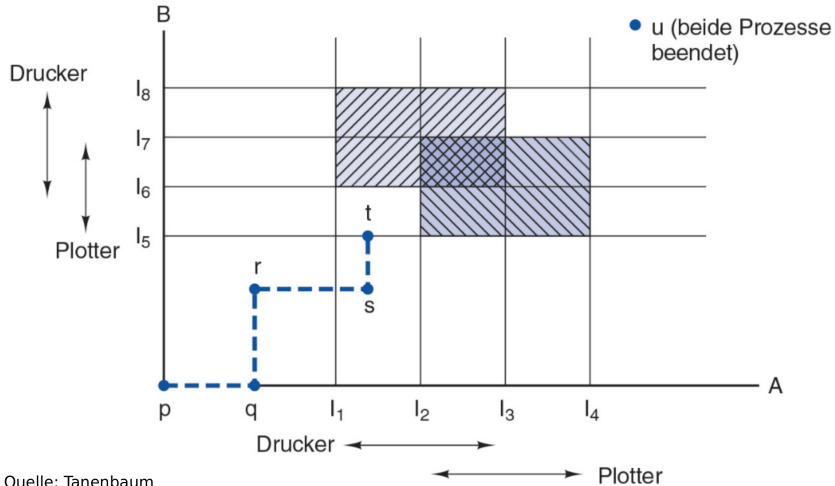
- **Prozessabbruch**

- Beispiel: Prozess wird abgeschossen wenn kein Speicher mehr im System
- Ziel: *Möglichst geringe Auswirkung auf das System*
- Problem: Wie das Opfer auswählen?
 - Möglichst kurze Laufzeit des Prozesses um Datenverlust zu minimieren?
 - Möglichst wenig Prozesse abbrechen?

- **Fazit:**

- Deadlock erkennen ist möglich
- Auflösen jedoch nur destruktiv möglich oder wenn Auswirkungen bekannt

- **Idee:**
 - Unterscheide **sichere** und **unsichere** Zustände.
 - Ressourcen werden nur geblockt wenn der Folgezustand sicher ist
- **Sicherer Zustand**
 - Es gibt (mindestens) eine Scheduling-Reihenfolge, die nicht zu einem Deadlock führt
 - Selbst wenn alle Prozesse ihre Maximalzahl an Ressourcen anfordern
- **Unsicherer Zustand**
 - Wenn Maximalforderungen gestellt werden, entsteht ein Deadlock
- **Voraussetzung**
 - Maximale Ressourcen müssen vorher bekannt sein



- **Beispiel: 10 Einheiten an Ressourcen verfügbar**
 - Zustand (a) ist **sicher**:

Belegt Max.

A	3	9
B	2	4
C	2	7

Frei: 3

a

Belegt Max.

A	3	9
B	4	4
C	2	7

Frei: 1

b

Belegt Max.

A	3	9
B	0	–
C	2	7

Frei: 5

c

Belegt Max.

A	3	9
B	0	–
C	7	7

Frei: 0

d

Belegt Max.

A	3	9
B	0	–
C	0	–

Frei: 7

e

Quelle: Tanenbaum

- **Beispiel: 10 Einheiten an Ressourcen verfügbar**
 - Zustand (b) ist **nicht sicher**:

Belegt Max.

A	3	9
B	2	4
C	2	7

Frei: 3

a

Belegt Max.

A	4	9
B	2	4
C	2	7

Frei: 2

b

Belegt Max.

A	4	9
B	4	4
C	2	7

Frei: 0

c

Belegt Max.

A	4	9
B	–	–
C	2	7

Frei: 4

d

Quelle: Tanenbaum

- **Bank mit**
 - Kunden mit maximalem Kreditlimit \rightarrow maximale Ressourcen
 - verfügbares Kapital \rightarrow vorhandene Ressourcen
- **Aufgabe**
 - Ein Kunde fordert Kredit an
 - Frage: Kann der Kredit gewährt werden, ohne dass ein unsicherer Zustand entsteht?

- A) Sei L die Liste aller Kunden
- B) Frage: Ist noch genug Geld übrig um einen beliebigen Kunden aus L bei Maximalforderung zu befriedigen?
- Nein \rightarrow **Zustand ist unsicher**
 - Ja \rightarrow Guthaben des Kunden zum vorhandenen Kapital addieren, Kunden aus L streichen. Weiter mit 2.)
- C) Wenn L leer \rightarrow **Zustand ist sicher**

Belegt Max.

A	0	6
B	0	5
C	0	4
D	0	7

Frei: 10

Belegt Max.

A	1	6
B	1	5
C	2	4
D	4	7

Frei: 2

Belegt Max.

A	1	6
B	2	5
C	2	4
D	4	7

Frei: 1

Quelle: Tanenbaum

a) sicher, b) sicher, c) unsicher

- **Problem: Maximalforderungen sind in der Regel vorher nicht bekannt**
- **Problem: Ressourcen oder Prozesse können hinzukommen oder wegfallen**

Fazit

Deadlock-Vermeidung erfordert zusätzliche Informationen durch das Anwendungsprogramm sowie das Einhalten definierter Randbedingungen.

Ansatz

Entwurf einer Software so dass *mindestens eine der vier Deadlock-Bedinungen gebrochen ist*:

- A)** Wechselseitiger Ausschluss
- B)** Keine Verdrängung
- C)** Hold-and-Wait
- D)** Zyklische Wartebedingungen

A) Wechselseitiger Ausschluss

- Virtualisierung von Geräten / Betriebssystem (z.B. Drucker-Spooler)

B) Keine Verdrängung

- Prozess gibt Ressource kurz frei und fordert sie sofort wieder an
- Verdrängung in manchen Situationen möglich (z.B. physikalischer Speicher)

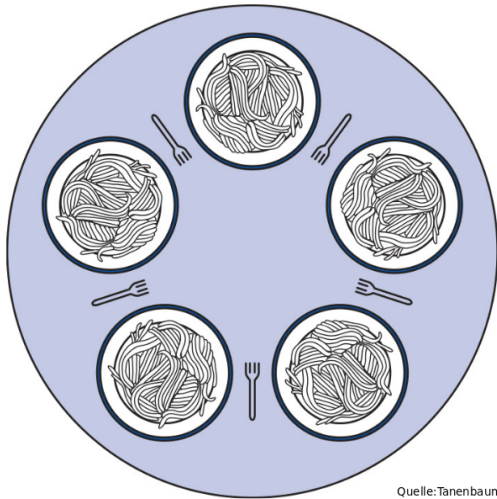
C) Hold-and Wait

- Gleichzeitige Belegung vermeiden (soweit möglich nur eine Ressource anfordern)
- Gleichzeitig benötigte Ressourcen ohne "Hold-and-Wait"-Situation anfordern (z.B. `trylock()`)

D) Zyklische Wartebedingung

- Definiere Ordnung der Ressourcen und stelle sicher, dass Ressourcen immer in der Reihenfolge angefordert werden

```
void lock_two(TMutex *m1, TMutex *m2) {  
    boolean ok1, ok2;  
  
    while (true) {  
        ok1 = trylock(m1);  
        ok2 = trylock(m2);  
        if (ok1 && ok2) return;  
        else {  
            if (ok1) unlock(m1);  
            if (ok2) unlock(m2);  
        }  
    }  
}
```



Quelle: Tanenbaum

Naiver Ansatz:

```
#define N 5

void philosopher(int i) {
    while (1) {
        think();
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

Mögliche Lösungen?

- **Mutex allokieren vor take_fork() und freigeben nach put_fork()**
 - Vorteil: Vermeidet Deadlock
 - Nachteil: Nur ein Philosoph kann gleichzeitig essen
- **Trylock(): Linke Gabel nehmen, prüfen ob rechte Gabel frei ist. Wenn nicht: linke Gabel hinlegen**
 - Vorteil: Vermeidet Deadlock
 - Nachteil: Verhungern möglich wenn alle Philosophen gleichzeitig linke Gabel nehmen
- **Trylock() Verbesserung: Warte zufällige Zeit vor erneutem Versuch Gabel zu nehmen**
 - Vorteil: verkleinert Wahrscheinlichkeit für verhungern
 - Nachteil: Bei manchen Zufallszahlenkombination kann viel Zeit mit warten verbracht werden

```
#define N          5          /* Anzahl der Philosophen */
#define LEFT      (i+N-1)%N    /* Nummer von i's linkem Nachbarn */
#define RIGHT     (i+1)%N      /* Nummer von i's rechtem Nachbarn*/
#define THINKING  0          /* Philosoph denkt */
#define HUNGRY    1          /* Philosoph versucht Gabeln zu bekommen */
#define EATING    2          /* Philosoph isst */

int state[N];                /* Feld mit Status jedes Philosophen */
semaphore mutex = 1;         /* Wechselseitiger Ausschluss fuer kritische Region */
semaphore s[N] = {0};        /* ein Semaphor pro Philosoph */

void philosopher(int i) {    /* i: Nummer des Philosophen */
    while (TRUE) {           /* Endlosschleife */
        think();             /* Philosoph denkt */
        take_forks();         /* nimm zwei Gabeln oder blockiere */
        eat();                /* essen */
        put_forks(i);         /* lege beide Gabeln zurueck auf Tisch */
    }
}
```

```
void take_forks(int i) {           /* i: Nummer des Philosophen (von 0 bis N-1) */
    down(&mutex);                  /* Eintritt in kritische Region */
    state[i] = HUNGRY;             /* registriere, dass Philosoph i hungrig ist */
    test(i);                       /* Teste ob zwei Gabeln genommen werden koennen */
    up(&mutex);                    /* Verlasse kritische Region */
    down(&s[i]);                   /* Blockiere wenn Gabel nicht bekommen */
}

void put_forks(int i) {           /* Eintritt in kritische Region */
    down(&mutex);                  /* Philosoph hat essen beendet */
    state[i] = THINKING;          /* kann der linke Nachbar essen? */
    test(LEFT);                   /* kann der rechte Nachbar essen? */
    test(RIGHT);                  /* verlasse kritische Region */
    up(&mutex);
}

void test(int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

- **Oft sind Deadlock-Quellen nicht offensichtlich**
- **Es gibt keine allgemeinen Mechanismen in Betriebssystemen zur Verhinderung von Deadlocks**
 - Jeder Software-Entwickler ist für Gegenmaßnahmen verantwortlich
 - Kritische Stellen im Code müssen selbst erkannt werden
- **Guter Ansatz: Ausschluss von Deadlocks durch Regeln**
 - Diese Regeln müssen im Code dokumentiert werden
 - Beispiel: Reihenfolge in der Ressourcen angefordert werden

Fragen?