

Verbessertes Konzept: Monitore

- Ein Nachteil von Semaphoren ist die Notwendigkeit zur expliziten Anforderung P und Freigabe V des kritischen Bereiches durch den Programmierer
- Vergißt der Entwickler z.B. die Freigabe V des Semaphors nach dem Durchlaufen des kritischen Abschnitts, dann kann es schnell zu einer Verklemmung kommen; solche Fehler sind sehr schwer zu finden!
- Zum einfacheren und damit weniger fehlerträchtigen Umgang mit kritischen Bereichen wurde deshalb das Konzept der *Monitore* (Hoare 1974, Brinch Hansen 1975) entwickelt:
 - Ein **Monitor** ist eine Einheit von Daten und Prozeduren auf diesen Daten, auf die zu jeden Zeitpunkt nur maximal ein Prozess zugreifen kann.
 - Wollen mehrere Prozesse gleichzeitig auf einen Monitor zugreifen, so werden alle Prozesse bis auf einen Prozess in eine Warteschlange eingereiht und blockiert.
 - Verlässt ein Prozess den Monitor, so wird ein Prozess aus der Warteschlange entnommen und dieser kann auf die Funktionen und Daten des Monitors zugreifen.
 - Die Signalisierung ist innerhalb des Monitors festgelegt, der Programmierer muss sie nicht selbstständig implementieren.

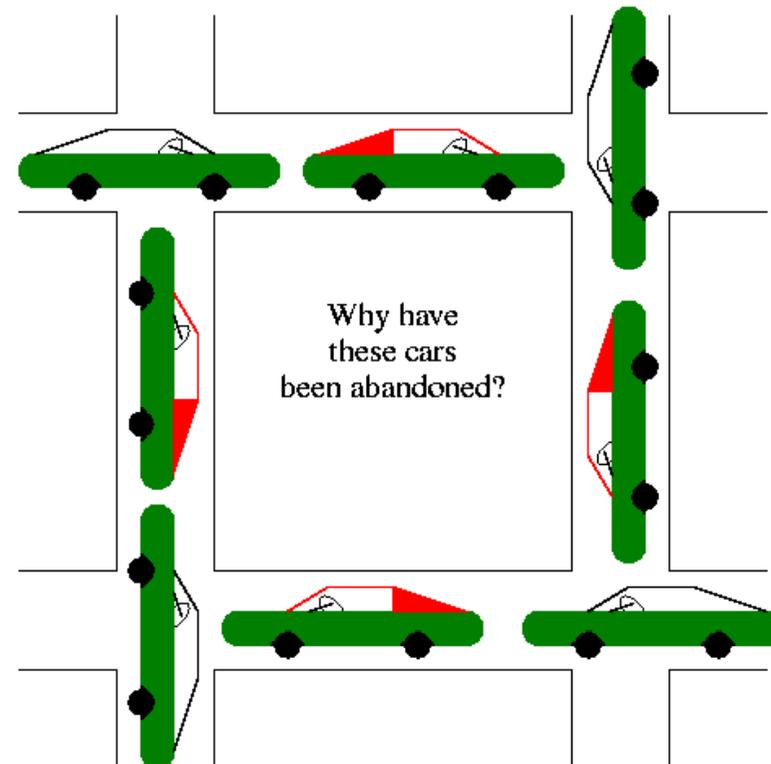
Beispiel: Monitore in Java

- In Java werden Monitore durch `synchronized`-Methoden implementiert. Zu jedem Zeitpunkt darf nur ein Prozess sich **aktiv** in einer dieser Methoden befinden.
- **Anmerkung:** normalerweise werden höhere Konstrukte wie Monitore durch einfachere Konstrukte wie den Semaphore implementiert. Siehe auch die Realisierung von Semaphoren durch das einfachere Konzept TSL-Befehl.
- In Java kann man das Monitorkonzept allerdings auch nutzen um selber Semaphore zu implementieren (siehe nebenstehenden Code).
- `wait()` und `notify()` sind zu jedem Objekt in Java definierte Methoden.

```
public class Semaphore {  
    private int value;  
  
    public Semaphore (int initial) {  
        value = initial;  
    }  
  
    synchronized public void up() {  
        value++;  
        if(value==1) notify();  
    }  
  
    synchronized public void down() {  
        while(value==0) wait();  
        value- -;  
    }  
}
```

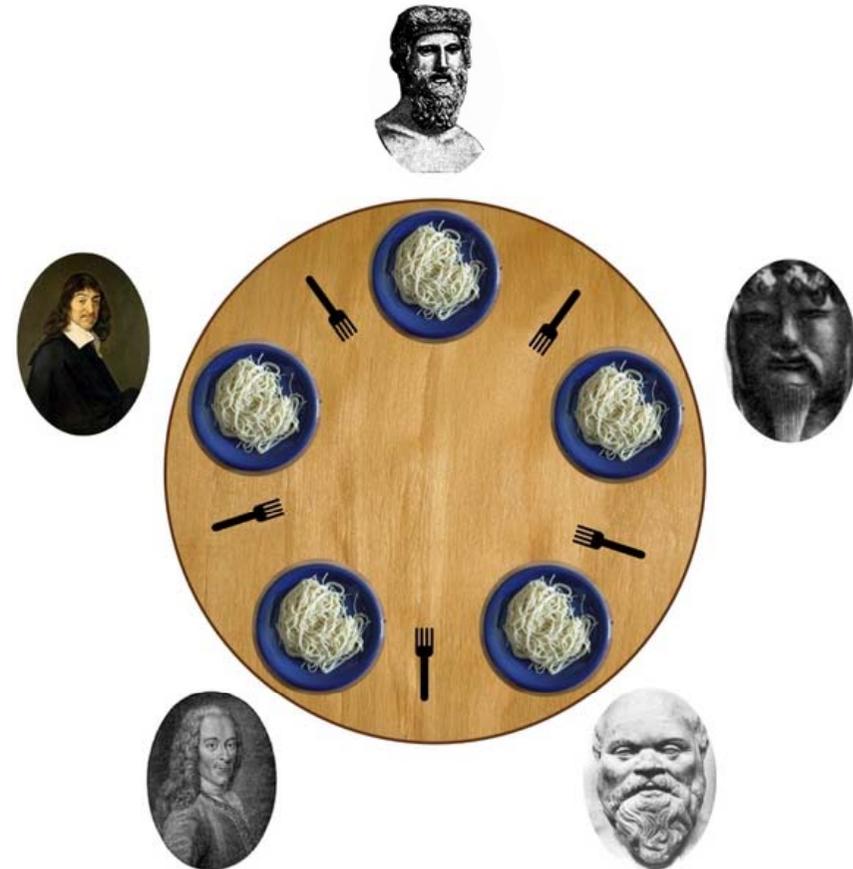
Bemerkung zu Verklemmungen (Deadlocks)

- Auch bei der korrekten Verwendung von Semaphoren und Monitoren kann es zu Deadlocks kommen, siehe Beispiel auf der folgenden Seite.
- Coffman, Elphick und Shoshani haben 1971 die vier konjunktiv notwendigen Voraussetzungen für einen Deadlock formuliert:
 1. Wechselseitiger Ausschluss: Es gibt eine Menge von exklusiven Ressourcen R_{exkl} , die entweder frei sind oder genau einem Prozess zugeordnet sind.
 2. Hold-and-wait-Bedingung: Prozesse, die bereits im Besitz von Ressourcen aus R_{exkl} sind, fordern weitere Ressourcen aus R_{exkl} an.
 3. Ununterbrechbarkeit: Die Ressourcen R_{exkl} können einem Prozess nicht entzogen werden, sobald er sie belegt. Sie müssen durch den Prozess explizit freigegeben werden.
 4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, die jeweils auf Ressourcen warten, die dem nächsten Prozess in der Kette gehören.
- Umgekehrt (und positiv) formuliert: ist eine der Bedingungen nicht erfüllt, so sind Verklemmungen ausgeschlossen.



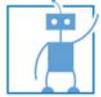
Klassisches Beispiel: Speisende Philosophen

- Klassisches Beispiel aus der Informatik für Verklemmungen: "Dining Philosophers" (speisende Philosophen, Dijkstra 1971, Hoare 1971)
- 5 Philosophen (Prozesse) sitzen an einem Tisch. Vor ihnen steht jeweils ein Teller mit Essen. Zum Essen benötigen sie zwei Gabeln (Betriebsmittel), insgesamt sind aber nur 5 Gabeln verfügbar.
- Die Philosophen denken und diskutieren. Ist einer hungrig, so greift er zunächst zur linken und dann zur rechten Gabel. Ist eine Gabel nicht an ihrem Platz, so wartet er bis die Gabel wieder verfügbar ist (ohne eine evtl. in der Hand befindliche Gabel zurückzulegen). Nach dem Essen legt er die Gabeln zurück.
- Problem: sind alle Philosophen gleichzeitig hungrig, so nehmen sie alle ihre linke Gabel und gleichzeitig ihrem Nachbarn die rechte Gabel weg. Alle Philosophen warten auf die rechte Gabel und es entsteht eine Verklemmung (deadlock).
- Gibt ein Philosoph seine Gabel nicht mehr zurück, so stirbt der entsprechende Nachbar den **Hungertod (starvation)**.



Fragestellung: Invers zählender Semaphor

- Aufgabenstellung: Implementierung des Leser-Schreiber-Problems mit Schreiber-Priorität
- Erläuterung:
 - Auf einen Datensatz können mehrere Leser gleichzeitig oder aber ein Schreiber zugreifen.
 - Sobald ein Schreiber den Schreibwunsch äußert, soll kein weiterer Leser (oder Schreiber) mehr auf den Datensatz zugreifen können. Zum Zeitpunkt der Signalisierung bestehende Lesevorgänge können regulär beendet werden, erst danach darf der Schreiber auf die Daten zugreifen.
- Problem: Häufig wird versucht das Problem mit einem „*invers zählenden Semaphor*“ zu lösen, also einem Semaphor, der bei 0 freigibt und sonst blockiert.
- Wie geht es richtig?



Nebenläufigkeit

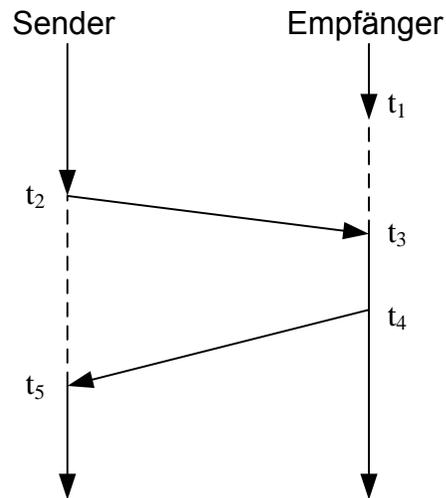
Interprozesskommunikation (IPC)

Interprozesskommunikation

- **Notwendigkeit der Interprozesskommunikation**
 - Prozesse arbeiten in unterschiedlichen Prozessräumen oder sogar auf unterschiedlichen Prozessoren.
 - Prozesse benötigen evtl. Ergebnisse von anderen Prozessen.
 - Zur Realisierung von wechselseitigen Ausschlüssen werden Mechanismen zur Signalisierung benötigt.
- **Klassifikation der Kommunikation**
 - synchrone vs. asynchrone Kommunikation
 - pure Ereignisse vs. wertbehaftete Nachrichten

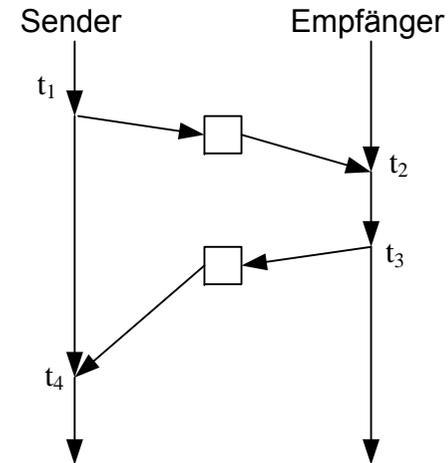
Synchron vs. Asynchron

Synchrone Kommunikation



- t₁ : Empfänger wartet auf Nachricht
- t₂ : Sender schickt Nachricht und blockiert
- t₃ : Empfänger bekommt Nachricht, die Verarbeitung startet
- t₄ : Verarbeitung beendet, Antwort wird gesendet
- t₅ : Sender empfängt Nachricht und arbeitet weiter

Asynchrone Kommunikation



- t₁ : Sender schickt Nachricht an Zwischenspeicher und arbeitet weiter
 - t₂ : Empfänger liest Nachricht
 - t₃ : Empfänger schreibt Ergebnis in Zwischenspeicher
 - t₄ : Sender liest Ergebnis aus Zwischenspeicher
- (Nicht eingezeichnet: zusätzliche Abfragen des Zwischenspeichers und evtl. Warten)*

IPC-Mechanismen

- Übermittlung von Datenströmen:
 - direkter Datenaustausch
 - Pipes
 - Nachrichtenwarteschlangen (Message Queues)
- Signalisierung von Ereignissen:
 - Signale
 - Semaphore
- Synchroner Kommunikation
 - Barrieren/Rendezvous
 - Kanäle wie z.B. Occam
- Funktionsaufrufe:
 - RPC
 - Corba

Nebenläufigkeit

IPC: Kommunikation durch Datenströme

Direkter Datenaustausch

- Mit Semaphoren und Monitoren geschützte Datenstrukturen eignen sich sehr gut für den Austausch von Daten:
 - schnelle Kommunikation, da auf den Speicher direkt zugegriffen werden kann.
- Allerdings kann die Kommunikation nur lokal erfolgen und zudem müssen die Prozesse eng miteinander verknüpft sein.
- Programmiersprachen, Betriebssysteme, sowie Middlewareansätze bieten komfortablere Methoden zum Datenaustausch.
- Grundsätzlich erfolgt der Austausch über das Ausführen von Funktionen `send(receiver address, &message)` und `receive(sender address, &message)`.

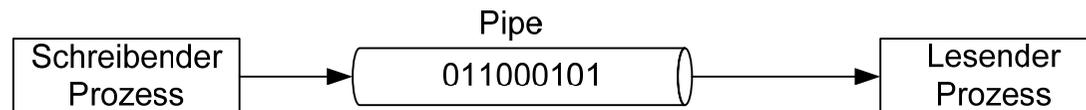
Fragestellungen beim Datenaustausch

- Nachrichtenbasiert oder Datenstrom?
- Lokale oder verteilte Kommunikation?
- Kommunikationsparameter:
 - mit/ohne Bestätigung
 - Nachrichtenverluste
 - Zeitintervalle
 - Reihenfolge der Nachrichten
- Adressierung
- Authentifizierung
- Performance
- Sicherheit (Verschlüsselung)

Heute: vor allem lokale Kommunikation, echtzeitfähige Protokolle zur Kommunikation in eigenem Kapitel

Pipes

- Die Pipe bezeichnet eine gepufferte, unidirektionale Datenverbindung zwischen zwei Prozessen nach dem **First-In-First-Out- (FIFO-)**Prinzip.
- Über den Namen der Pipe (ähnlich einem Dateinamen) können Prozesse unterschiedlichen Ursprungs auf eine Pipe lesend oder schreibend zugreifen. Zur Kommunikation zwischen Prozessen gleichen Ursprungs (z.B. Vater-, Kindprozess) können auch anonyme Pipes verwendet werden. Die Kommunikation erfolgt immer asynchron.



Pipes in Posix

- POSIX (Portable Operating System Interface) versucht durch Standardisierung der Systemaufrufe die Portierung von Programmen zwischen verschiedenen Betriebssystemen zu erleichtern.
- POSIX.1 definiert folgende Funktionen für Pipes:

```
int mkfifo(char* name, int mode);          /*Erzeugen einer benannten Pipe*/
int unlink ( char *name );                /*Loeschen einer benannten Pipe*/
int open ( char *name, int flags);        /*Oeffnen einer benannten Pipe*/
int close ( int fd );                     /*Schliessen des Lese- oder Schreibendes einer
                                           Pipe*/
int read ( int fd, char *outbuf, unsigned bytes ); /*Lesen von einer Pipe*/
int write ( int fd, char *outbuf, unsigned bytes ); /*Schreiben an eine Pipe*/
int pipe ( int fd[2] );                   /*Erzeugen eine unbenannte Pipe*/
```

Nachteile von Pipes

- Pipes bringen einige Nachteile mit sich:
 - Pipes sind nicht nachrichtenorientiert (keine Bündelung der Daten in einzelne Pakete (Nachrichten) möglich).
 - Daten sind nicht priorisierbar.
 - Der für die Pipe notwendige Speicherplatz wird erst während der Benutzung angelegt.
- Wichtig für die Implementierung:
 - Es können keine Daten aufgehoben werden.
 - Beim Öffnen blockiert der Funktionsaufruf, bis auch das zweite Zugriffsende geöffnet wird (Verhinderung durch `O_NDELAY` Flag).
- Lösung: Nachrichtenwarteschlangen

Nachrichtenschlangen (message queues)

- Nachrichtenschlangen (Message Queues) sind eine Erweiterung von Pipes. Im Folgenden werden Nachrichtenschlangen, wie in POSIX 1003.1b (Echtzeiterweiterung von POSIX) definiert, betrachtet.
- Eigenschaften der POSIX MessageQueues:
 - Beim Anlegen einer MessageQueue wird der benötigte Speicher reserviert. ⇒ Speicher muss nicht erst beim Schreibzugriff angelegt werden.
 - Die Kommunikation erfolgt nachrichtenorientiert. Die Anzahl der vorhandenen Nachrichten kann dadurch abgefragt werden.
 - Nachrichten sind priorisierbar → Es können leichter Zeitgarantien gegeben werden.

Nachrichtwarteschlangen

- Schreibzugriff in Standardsystemen: Der schreibende/sendende Prozess wird nur dann blockiert werden, falls der Speicher der Datenstruktur bereits voll ist. **Alternative in Echtzeitsystemen: Fehlermeldung ohne Blockade.**
- Lesezugriff in Standardsystemen: Beim lesenden/empfangenden Zugriff auf einen leeren Nachrichtenspeicher wird der aufrufende Prozess blockiert bis eine neue Nachricht eintrifft. **Alternative: Fehlermeldung ohne Blockade.**
- Ein anschauliches Beispiel für den Einsatzbereich ist der Spooler eines Druckers: dieser nimmt die Druckaufträge der verschiedenen Prozesse an und leitet diese der Reihe nach an den Drucker weiter.

