Einführung in die Funktionsweise von Computersystemen

7. Kapitel

Prozesse im Zusammenspiel: Prozessinteraktion

Prof. Matthias Werner

Professur Betriebssysteme

Interaktionsarten

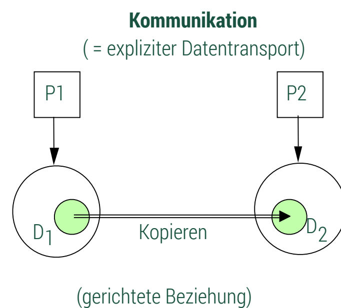
- Prozesse als Teile komplexer Programmsysteme müssen Daten austauschen:
  - sich aufrufen (bzw. beauftragen)
  - aufeinander warten
  - sich auslösen
  - sich abstimmen

  .... sie müssen interagieren.
- Operationen zur Prozessinteraktion bilden (neben der Prozessverwaltung) den zweiten wesentlichen Aufgabenbereich eines Betriebssystemkerns.

Arten der Interaktion

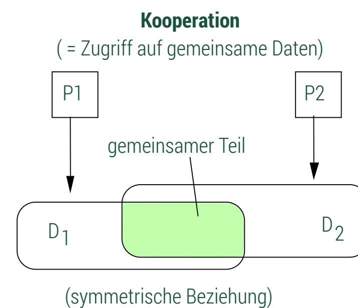
- Prozessinteraktion besitzt einen funktionalen und einen zeitlichen Aspekt:
- Wir unterscheiden:
  - Zeitlicher Aspekt: Ablaufabstimmung (**Koordination**)
  - Funktionaler Aspekt: Informationsaustausch (**Kommunikation**, **Kooperation**)

Kommunikation
(= expliziter Datentransport)



(gerichtete Beziehung)

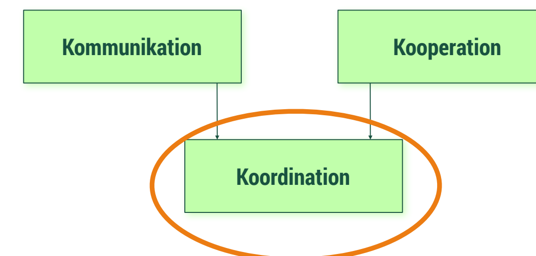
Kooperation
(= Zugriff auf gemeinsame Daten)



(symmetrische Beziehung)

Zusammenhang

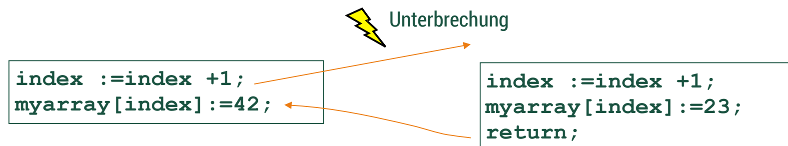
- Von den drei Grundformen der Interaktion ist die Koordination die elementarste, denn sowohl Kommunikation als auch Kooperation benötigen eine zeitliche Abstimmung zwischen den Interaktionspartnern ➡ Koordination



- Wir werden daher zunächst die Koordination behandeln

Warum Koordination?

- Wenn Programmflüsse unterbrochen werden können, kann es zu konkurrierenden Zugriffen auf Daten kommen
- ➡ Wettlaufsituationen (race conditions) sind möglich
- Ergebnis ist nicht sicher vorhersagbar
- Bereich ist **kritischer Abschnitt**



- Ergebnis:
  - `myarray[index-1]`: unbestimmt
  - `myarray[index]`: 42
  - „23" der Unterbrechung ist „verloren gegangen"

Koordinationskonzept: Kritischer Abschnitt

- Ein Bereich, auf den innerhalb einer Zeitspanne keine konkurrierenden Zugriffe (z.B. auf Daten zulässig) sind, heißt **kritischer Abschnitt** (**critical section**)
  - verschiedene Ausführungen haben häufig gemeinsame kritische Abschnitte
- Ziel: **gegenseitiger Ausschluss** (**mutual exclusion**)
  - Immer nur ein Befehlsstrom darf in einem Abschnitt sein
- Auch der **Kern** kann durch Interrupts unterbrochen werden
  - Lösung: den ganzen Kern als kritischen Abschnitt betrachten
  - Alternative: Im Kern stückweise Kritische Abschnitte einführen
    ➡ in jedem Fall braucht man einen **Kernausschluss**
- **Achtung:** Im Kern können **keine** Prozessmechanismen genutzt werden

Koordinationskonzept: Kritischer Abschnitt

- Ein Bereich, auf den innerhalb einer Zeitspanne **keine konkurrierenden Zugriffe** (z.B. auf Daten zulässig) sind, heißt **kritischer Abschnitt** (critical section)
- verschiedene Ausführungen haben häufig gemeinsame kritische Abschnitte
- Ziel: Gewährleistung eines **gegenseitigen Ausschluss** (mutual exclusion) ➡ Immer nur ein Befehlsstrom darf in einem Abschnitt sein
- Auch der Kern kann durch Interrupts unterbrochen werden
  - **Lösung**: wenn möglich, Interrupts sperren, solange Ausführung im Kern
  - Bei Mehrkernprozessoren oder wenn Interruptsperrung nicht möglich, wird das Problem komplizierter
  - Datenstrukturen für die Protokollierung von kritischen Abschnitt wären selbst kritisch ➡ Softwarelösung notwendig

Softwarelösung für gegenseitigen Ausschluss

- Erste echte (Software-)Lösung von Dekker, später Peterson
  - Hier: Lösung für zwei Prozesse, 0 und 1

```
const int N = 2;                              // Anzahl von Konkurrenten
volatile int turn;                            // Wer ist dran?  (shared)
volatile bool interested[N];                  // Wer will? (shared)

void enter_section(int processID)            // Wer ruft, 0 oder 1
{
    int other = 1 - processID;                // ID des anderen Prozess
    interested[processID] = true;             // Ich will
    turn = processID;
    while ((turn == processID)
           && (interested[other] == true));   // warten
}

void leave_section(int processID)             // Wer verlässt krit. Bereich, 0 or 1?
{
    interested[processID] = false;
}
```

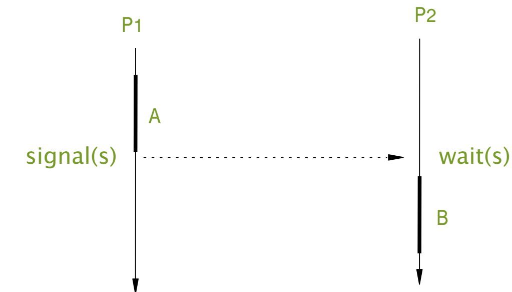
- Moderne Prozessoren haben HW-Unterstützung, um **atomares** Lesen/Schreiben zu ermöglichen ➡ wird vom Betriebssystem für Kern benutzt

Koordination auf Prozessebene

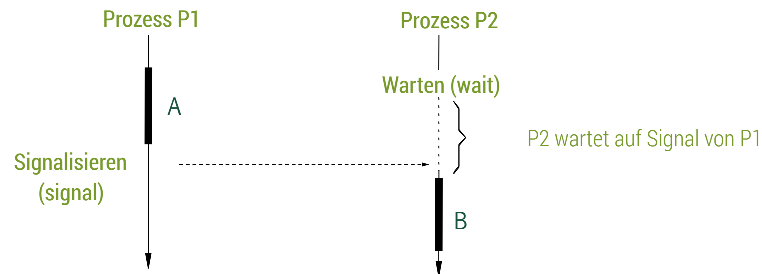
- Auch Prozesse können auch kritische Abschnitte miteinander teilen ➡ gegenseitiger Ausschluss nötig
- Programmierer will sich nicht mit Details des Prozessors beschäftigen ➡ **Abstraktionen** notwendig
- Koordination (u.a. Gewährleistung des gegenseitigen Ausschluss) ist **Dienst des Betriebssystems** und wird über **Systemrufe** realisiert
- Betrachten einheitlichen Ansatz für verschiedene Koordinationsprobleme

Konzept: Signalisierung

- Bei der Signalisierung soll eine **Reihenfolgebeziehung** hergestellt werden.
- Ein Abschnitt A in einem Prozess P1 soll vor einem Abschnitt B in einem Prozess P2 ausgeführt werden.
- Dazu bietet der Kern die Operationen **`signal`** und **`wait`** an, die eine gemeinsame (geschützte) binäre Variable s benutzen



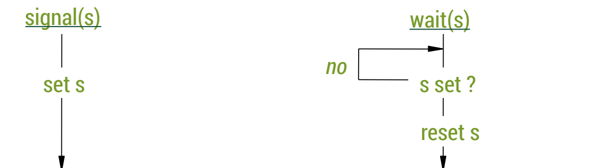
Signalisierung: Beispiel



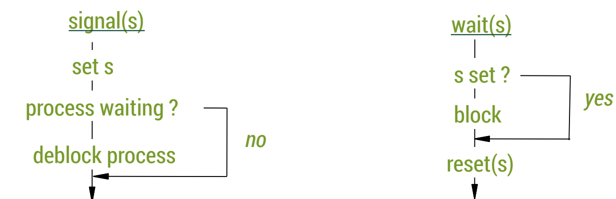
- **Beispiel**: Steuerung eines technischen Prozesses:
  - A: Einfüllen von Flüssigkeit in einen Behälter (Ventil offen)
  - B: Heizen (Spannung an Heizspirale)

Grundform der Signalisierung

- In ihrer einfachsten Form können die Operationen folgendermaßen realisiert werden:

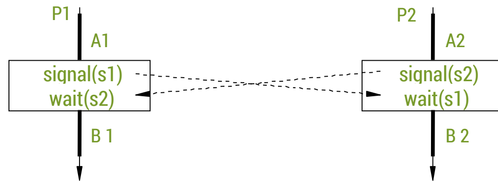


- Dies bedeutet ein aktives Warten an der Signalisierungsvariablen s.
- Ist die Wartezeit zu lange, so sollte der Prozessor freigegeben werden ➡ Signalisieren mit Wartezustand, nur ein Prozess kann warten

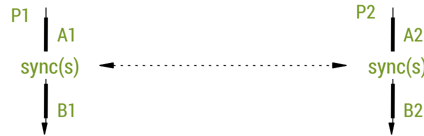


Wechselseitige Synchronisierung

- Ein symmetrischer Einsatz der Operationen bewirkt, dass sowohl A1 als auch A2 ausgeführt sind, bevor B1 oder B2 ausgeführt werden.



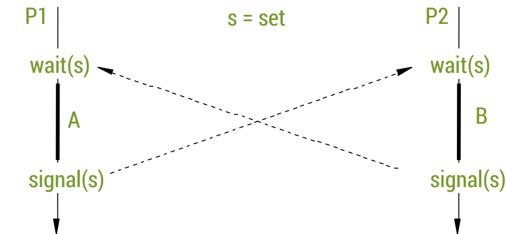
- Die Prozesse P1 und P2 synchronisieren sich an dieser Stelle ➡ Wir können das Operationspaar als eine Operation `sync` zusammenfassen:



- Da P1 und P2 an dieser Stelle aufeinander warten, spricht man auch von einem Rendezvous

Sperren

- Wir betrachten den folgenden Einsatz von Signalisierungsoperationen:

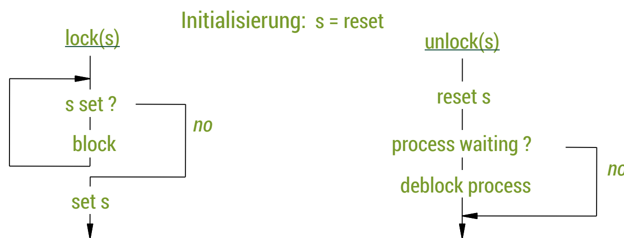


- A und B können **nicht nebenläufig** ausgeführt werden:

  Entweder A vor B oder B vor A, d.h. es findet keine Überlappung in der Ausführung von A und B statt. Die Ausführungen von A und B schließen sich gegenseitig aus.
- Die Signalisierungsoperationen können also auch dazu verwendet werden, um **kritische Abschnitte** *(**critical section**)* auf **Prozessebene** zu sichern.

Sperren

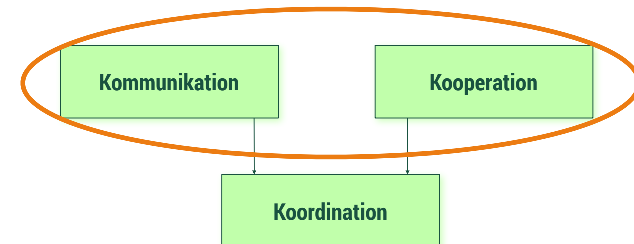
- Zweckmäßigerweise geben wir ihnen dann auch die entsprechenden Namen: **Sperren (`lock`)** und **Entsperren** (`unlock`)



- Strukturell entspricht das `lock` dem `wait` und das `unlock` dem `signal`
- **Anmerkung**:x
  Im Unterschied zur Formulierung des `signal` weiter oben wird hier jedoch die Variable s **in einer Schleife** abgefragt, um den Fall zu berücksichtigen, dass zwischen dem Deblockieren des auf die Sperre wartenden Prozesses und dem Setzen der Sperre ein weiterer Prozess die Sperre setzen könnte.

Kommunikation

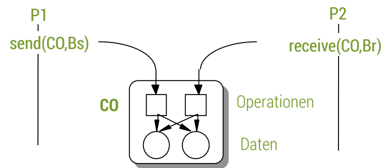
- Bisher: Koordinierung zwischen Prozessen
- Manchmal sollen Informationen verarbeitet werden, die anderen Prozessen gehören ➡ Kommunikation und Kooperation



- Kommunikation ➡ Austausch von **Nachrichten**
- Kooperation ➡ Arbeiten auf **gemeinsamen Speicher**
- Während Arbeiten auf Speicher aus Sicht des Prozesses „Normalfall" ist, muss für den Nachrichtenaustausch erst die Möglichkeit geschaffen werden ➡ Konzept des **Kanals**

Konzept: Kanal

- Ein **Kanal** ist ein Datenobjekt, das die Operationen **Senden** (**send**) und **Empfangen** (**receive**) zur Verfügung stellt
- Damit werden Adressen von Nachrichten übergeben
  - **Sender**: Adresse der zu verschickenden Nachricht (Datenbehälter sendeseitig, **buffer send** (Bs). Statt der Adresse kann hier auch die Nachricht selbst stehen)
  - **Empfänger**: Adresse, wohin die empfangene Nachricht geschrieben werden soll (Datenbehälter empfängerseitig, **buffer receive**(Br))

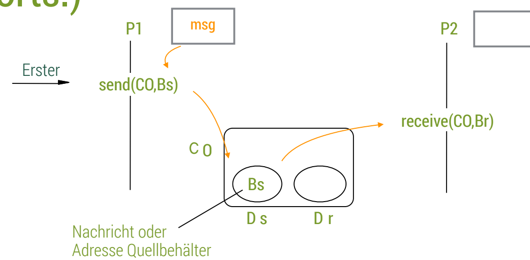


Zeitverhältnisse

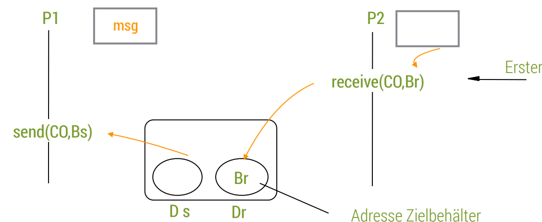
- Da Sender und Empfänger ihre Operationen zu beliebigen Zeitpunkten aufrufen können, sind zwei Fälle zu berücksichtigen:
  1. Erst Senden, dann Empfangen
  2. Erst Empfangen, dann Senden
- Wenn die aufrufenden Prozesse in den Operationen nicht aufgehalten (blockiert) werden sollen, besteht die Notwendigkeit der **Zwischenspeicherung** im Kanal.
  - **Sender zuerst**: Die Nachricht bzw. ihre Adresse wird im Kanal abgelegt und kann bei einem nachfolgenden Empfangen abgeholt werden
  - **Empfänger zuerst**: Die Adresse des Zielpuffers wird abgelegt, so dass bei einem nachfolgenden Empfangen die Nachricht dorthin kopiert werden kann
- Der Kanal hält entsprechende Variable zur Aufnahme dieser Daten vor

Zeitverhältnisse (Forts.)

- Sender zuerst:



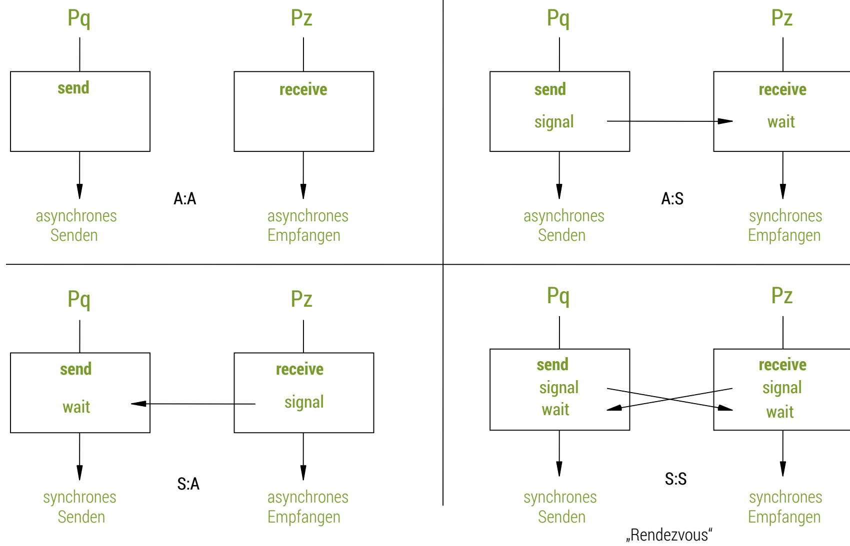
- Empfänger zuerst:



Koordinierte Kommunikation

- Bisher hatten wir keinerlei zeitliche Abstimmung zwischen Sender und Empfänger gefordert: Beide rufen ihre jeweilige Operation auf, legen ggf. Daten im Kanal ab, verlassen die Prozedur und arbeiten weiter, **ohne** auf den Kommunikationspartner zu warten.
- Man nennt dieses Vorgehen **asynchron** (**asynchrones Senden**, **asynchrones Empfangen**)
- Häufig ist jedoch z.B. der Empfänger dringend auf den Empfang der Nachricht angewiesen, d.h. er kann erst weiterarbeiten, wenn die Nachricht eingetroffen ist.
  - Er wird daher in der Empfangsoperation aufgehalten (blockiert).
  - Auf diese Weise synchronisiert er sich mit dem Sender (d.h. wartet auf ihn).
  - Man spricht dann von einem **synchronen Empfangen**.
- Analog dazu ist auch ein **synchrones Senden** möglich, bei dem der Sender solange blockiert wird, bis die dazugehörige Empfangsoperation aufgerufen wird.
- Durch Kombination ergeben sich **vier Varianten.**

Koordinationsvarianten der Kommunikation

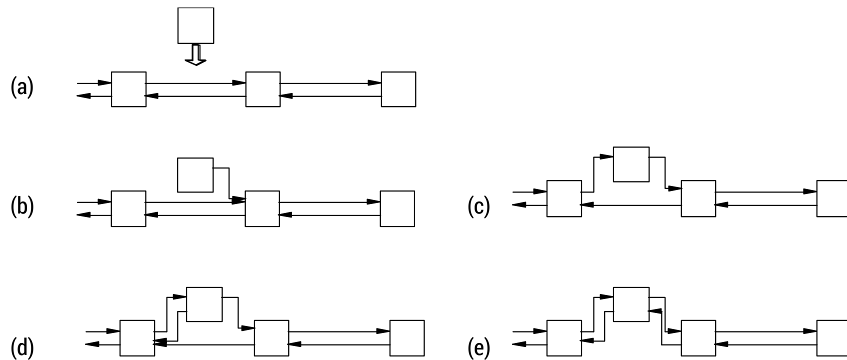


Kooperation

- Kooperation tritt auf, wenn mehrere Prozesse auf dieselben Daten zugreifen
- Speicherzugriff ist Standardfall ➡ keine „neuen“ Konzepte notwendig
- Aber: Probleme können auftreten
  - **Inkonsistenzen**
  - **Kapazitätsbeschränkung**

Beispiel für Inkonsistenzen

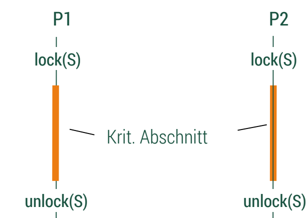
Beispiel: Listenoperation “Einfügen”, aufgelöst in Einzelschritte



Während der Situationen c) und d) ist die Listenstruktur **inkonsistent**. Ein konkurrierend zugreifender Prozess sähe eine fehlerhafte Datenstruktur.

Sperren

- Kooperation von Prozessen auf gemeinsamen Daten fällt offensichtlich mit dem Problem des **kritischen Abschnitts** bzw. des **gegenseitigen Ausschlusses** zusammen
- Zur Sicherung kritischer Abschnitte können bereits eingeführte Sperroperationen einsetzen:



Mehrere Prozesse im kritischen Abschnitt

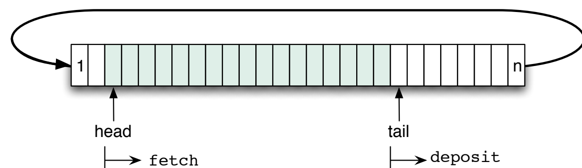
- Ein Kooperationsabschnitt ist also dadurch gekennzeichnet, dass sich zu einem Zeitpunkt genau ein Prozess „darin aufhält".
- Dieses Prinzip kann man erweitern, indem man andere Kapazitäten als „1" festsetzt.
- Man kann sowohl für die Zahl der „durchgelassenen" als auch für die Zahl der wartenden Prozesse Obergrenzen vorsehen:
  - 1
  - c>1 ➡ konstant
  - n beliebig
- Gründe, die Anzahl der Prozesse in einem bestimmten Bereich zu begrenzen:
  - Platzmangel
  - Leistungsabfall
  - Inkonsistenzen

Beispiel Konsistenz: Leser-Schreiber-Problem

- In einem Kooperationsabschnitt wie die Nutzung einer Verlinkten Liste können sich durchaus mehrere Prozesse aufhalten, wenn **alle nur lesen**
- Allerdings „verträgt" sich dies nicht mit einem Schreiben
- Mehrere Schreiber vertragen sich ohnehin nicht
- Im Kooperationsabschnitt dürfen daher sein:
  - **ein** Schreiber
    oder
  - **beliebig viele** Leser

Beispiel Kapazität: Ringpuffer

- Mehrere Prozesse benutzen einen gemeinsamen Pufferbereich, der als „Ring" aufgefasst wird
  - Prozesse können Daten dort ablegen: **deposit**(data)
  - Prozesse können Daten dort abholen: **fetch**(data)



- Neben der Sicherstellung des gegenseitigen Ausschlusses müssen offensichtlich noch weitere Bedingungen berücksichtigt werden:
  - deposit darf nur aufgerufen werden, wenn noch **Platz** im Puffer **vorhanden** ist
  - fetch darf nur aufgerufen werden, wenn Puffer **nicht leer** ist

Konzept: Semaphor

- Semaphor (E.W. Dijkstra, ca. 1965)
- Zählsperre S
  - Zähler
  - Operationen P(S) und V(S), entspricht LOCK(S) und UNLOCK(S)
- **Operation P**
  - dekrementiert Zähler
  - ist der Zähler <= 0, wird der rufende Prozess blockiert
- **Operation V**
  - inkrementiert Zähler
  - ist Zähler >0, wird evtl. blockierter Prozess deblockiert

Beispiel: Semaphoren unter UNIX

- Unix (System V) stellt als Semaphorenobjekte keine einzelnen Semaphoren, sondern Arrays von Semaphoren zur Verfügung
- Systemcalls
  - **semget()** – Anlegen eines Semaphore arrays
  - **semop()** – Bearbeiten von Semaphoren
  - **semctl()** - Abfrage und Setzen des Status
- Semaphore Table verwaltet Arrays von Semaphoren
  - Mehrere Semaphoren für mehrere kritische Abschnitte
  - Gleichzeitiges Bearbeiten der Semaphoren
  - Beliebiges In-/Dekrement

Beispiel: Semaphoren unter UNIX (Forts.)

- Was passiert, wenn ein Prozess terminiert, der ein Semaphor hält?
  - 1. Ansatz: Zeiger auf haltenden Prozess, Freigabe bei Terminierung
    ➡ Nicht möglich diesem Ansatz: mehrere Halter möglich
- Lösung: Undo Table
  - Für jeden Prozess Protokollierung der Änderungen für jeden betretenen Semaphor (Spielgelung)
  - Bei Terminierung wird Semaphor um Spiegelwert korrigiert