Betriebssysteme
7. Kommunikation und Kooperation

Tobias Lauer

Interaktion von Prozessen

- Interaktion = die geordnete Zusammenarbeit und der Austausch von Information zwischen Prozessen[1]
- Wozu müssen Prozesse interagieren?
  - Prozessfamilien/Threadgruppen mit gemeinsamer Aufgabe
  - Jeder Prozess/Thread realisiert Teil dieser Aufgabe
  - Z.B. Schreiber/Leser, Erzeuger/Verbraucher, Client/Server,..
  - u.v.a.m
- Interaktion beinhaltet
  - Synchronisation/Koordination (= zeitliche Abstimmung)
  - Informationsaustausch (= inhaltliche Abstimmung)
- Informationsaustausch hat zwei Grundformen:
  a) Kommunikation (asymmetrisch; Sender-Empfänger)
  b) Kooperation (symmetrisch; Arbeiten mit gemeinsamen Objekten)
- Eine der zentralen Aufgaben eines Betriebssystems ist es die Kommunikation und Kooperation von Prozessen zu unterstützen

[1]: wie zuvor werden Threads nicht separat betrachtet

Kommunikation (Nachrichtenaustausch)

- Einfachste Form:
  - Prozess 1 benutzt Betriebssystemoperation Send

    ```
    Send (IN: Zielprozess, IN: Nachricht)
    ```
  - Prozess 2 benutzt Betriebssystemoperation Receive

    ```
    Receive (OUT: Quellprozess, OUT: Nachricht)
    ```

- In der Regel bieten Betriebssysteme unterschiedliche Varianten dieser Kommunikation an
- Standardbegriff: IPC (Inter-Process Communication)
- 2 Hauptunterscheidungsmerkmale
  - Synchronisierungstyp: synchron vs. asynchron
  - Kanaltyp: direkt vs. indirekt

Synchrone und Asynchrone Kommunikation

Senderseite:

- Synchrones („Blockierendes“) Senden
  - Sender blockiert, bis Nachricht von Zielprozess empfangen
  - Vorteil: Kein Puffer (bzw. nur 1 Pufferplatz) nötig; implizite Quittierung möglich
  - Nachteil: Sender kann während dieser Zeit nichts anderes erledigen
- Asynchrones („Nichtblockierendes“) Senden
  - Sender fährt in seinem Kontrollfluss fort
  - Vorteil: Sender kann weitere Aufgaben erledigen
  - Nachteil: Puffer (potentiell unendlicher Größe) wird benötigt
    Sender weiß nicht, ob Nachrichten empfangen wurden
- Auch Mischformen möglich:
  - Sender verschickt solange Nachrichten, bis Pufferplatz voll
  - Erst dann wird der Sender blockiert und vom Betriebssystem deblockiert, wenn wieder Pufferplatz frei
  - Ideal einsetzbar, um Erzeuger/Verbraucher-Schema zu realisieren

Synchrone und asynchrone Kommunikation

Empfängerseite:

- Synchrones („Blockierendes“) Empfangen
  - Empfänger blockiert, bis Nachricht von Quellprozess eingetroffen
  - Nachteil: Empfänger kann während dieser Zeit nichts anderes erledigen
  - Vorteil: implizite Synchronisation; Empfänger wacht auf, wenn Arbeit da ist

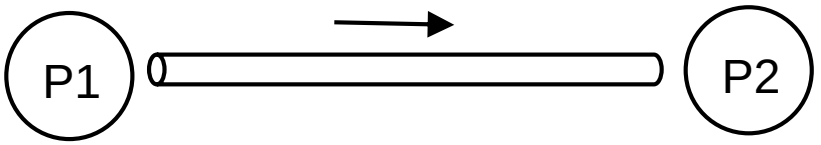
- Asynchrones („Nichtblockierendes“) Empfangen
  - Variante A ("Busy Wait"):
    Empfänger fragt zyklisch an, ob neue Nachricht eingetroffen
    Wenn ja, erhält er die eingetroffene Nachricht
    Wenn nein, erhält er sofort (ohne Blockierung) eine Fehlermeldung
  - Variante B ("Asynchroner Prozeduraufruf"):
    Empfänger übergibt dem Betriebssystem eine Prozedur („Handler“)
    und fährt mit seiner Arbeit fort
    Wenn die Nachricht eintrifft, unterbricht das Betriebssystem das
    laufende Programm (Interrupt) und startet den Handler-Code
  - Vorteil beider Varianten: Empfänger kann an weiteren Aufgaben arbeiten
  - Nachteil: Komplexere Programmstruktur (Zyklische Aufrufe bzw. Synchronisation zwischen Hauptprogramm und Handler)

Synchrone und Asynchrone Kommunikation

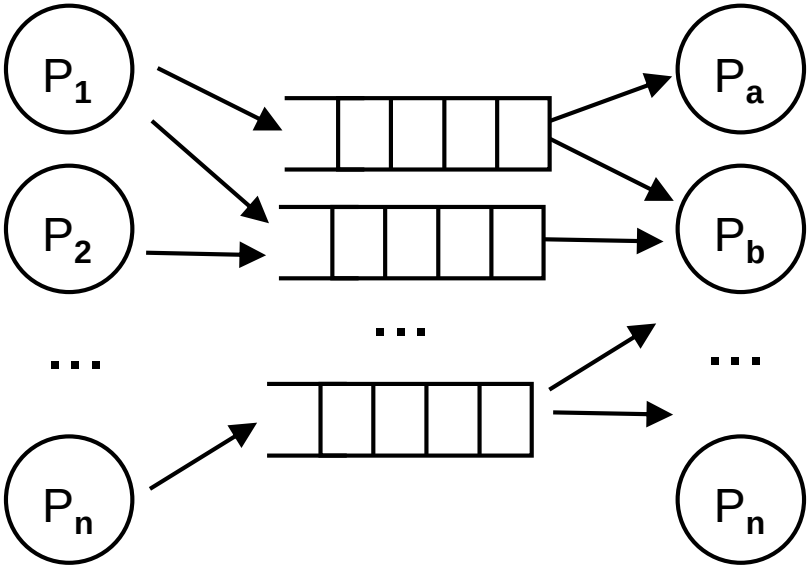
Sender / Empfänger	Synchrones Senden (Sender blockiert bis Nachricht empfangen wurde)	Asynchrones Senden (Sender macht weiter)
Synchrones Empfangen (Empfänger blockiert bis Nachricht da ist)	„Rendezvous“-Konzept gebräuchlich (Client-Server-Prog.)	Gebräuchlichste Kommunikationsform (auch zwischen Rechnern)
Asynchrones Empfangen (Empfänger macht weiter, wenn keine Nachricht da ist)	ungebräuchlich	ebenfalls gebräuchlich, eingesetzt in der Variante des zyklischen Überprüfens auf empfangbare Nachrichten oder der asynchronen E/A-Beendigung

Direkte und indirekte Kanaltypen

- Direkte Kanäle
  - Quelle+Ziel = Prozesse
  - i.d.R. 1:1-Kommunikation
  - meist statische Zuordnung zu Prozessen
  - Sender/Empfänger kennen Kommunikationspartner



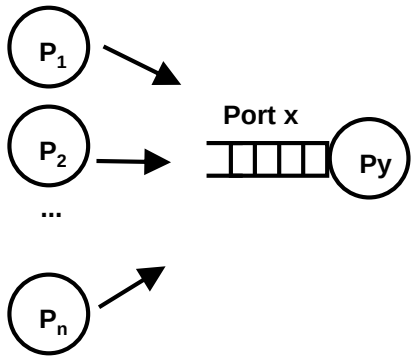
- Indirekte Kanäle
  - Eigene (unabhängige) Kommunikationsinstanzen
  - Separate Adressen
  - N:M-Kommunikation
  - Rechteverwaltung
  - statische oder dynamische Zuordnung zu Prozessen (Connect/Disconnect)



Indirekte Kanaltypen - 2

- Häufig verwendete Begriffe: Warteschlange, Queue, Mailbox, …
- Auslieferungsvarianten:
  - FIFO-Auslieferung (über alle Sender und Empfänger)
  - Typisierte Auslieferung (Sender spezifizieren Nachrichtentyp; Empfänger holen sich nur Nachrichten bestimmter Typen oder Priorität)
  - Empfänger wählt auszuliefernde Nachrichten selbst (Empfänger hat Zugriff auf Warteschlange, kann entscheiden, welche er empfangen will)

- Falls Mailbox einem einzelnen Prozess zugeordnet ist, wird der Begriff „Port“ verwendet:
- Sendevorgänge an Ports statt an Prozesse
- Beispiel: TCP/IP „well-known“ Ports (HTTP, FTP, ..) für Client/Server



Kommunikation mit Signalen

- Asynchrone Interprozesskommunikation mit einfachen Nachrichten
- Nachricht = Signal von bestimmtem Typ (Identifikator, i.d.R. Integer)
- Sender (= Anwendungsprozess oder Betriebssystem) triggert Signal an einen bestimmten Empfangsprozess
- „Default Signal Handlers“ führen vordefinierte Aktionen durch (z.B. Programmabbruch)
- Empfängerprozess kann eigenen „Signal Handler“ definieren
  = Prozedur, die bei Eintreffen eines Signals ausgeführt wird
- 1 Signal Handler pro Signaltyp (neben dem Default Signal Handler)
- Signal Handler unterbricht laufende Programmausführung
  (Signal = Software-Analogon zum Hardware-Interrupt)
- Erlaubt Empfängerprozess auf Signal zu reagieren
- Empfänger kann Signale „maskieren“, d.h. für bestimmte Zeit sperren
- Betriebssystem kann Signale (bis zu gewissem Grad) zwischenspeichern und verspätet ausliefern
- Alternativ kann ein Empfänger auch blockierend auf ein Signal warten

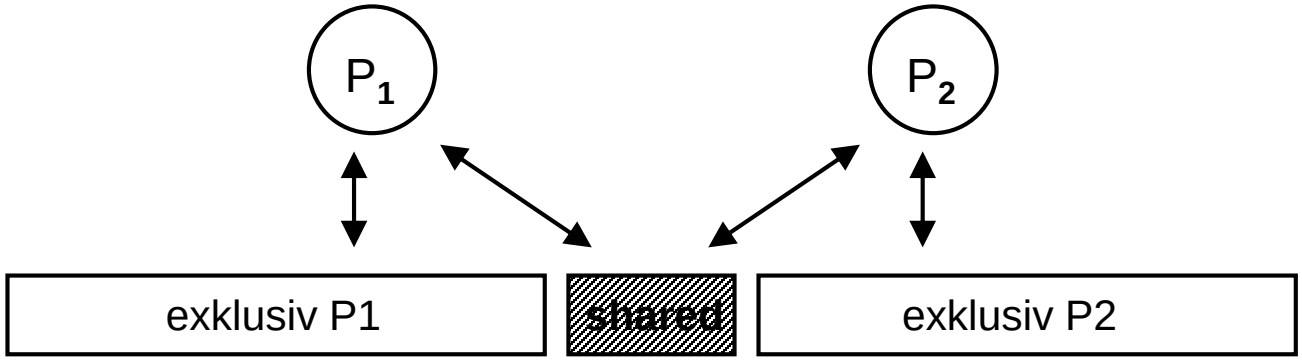
Kommunikation mit Signalen

- Spezielle Signaltypen:
  - Laufzeitfehlermeldungen durch Betriebssystem (Speicherzugriffsfehler, Overflow, Division by Zero, etc.)
  - Kill-Signale von anderen Prozessen (z.B. von Vater-Prozess, von Administrationskonsole,..)
  - Timer (zuvor installierte Alarme, die den Prozess nach einer bestimmten Zeit aufwecken)
  - Benutzerdefinierte Signale (SIGUSR) zwischen Prozessen
- Signal Handler muss kurz sein, da sonst möglicherweise gerade laufende Operationen (z.B. andere Systemaufrufe) gestört werden. Keine I/O-Operationen oder komplexe Systemaufrufe in einem Signal Handler!
- Good practice: Signal Handler setzt „Flag“, das in der Hauptschleife des Empfängerprogramms zyklisch abgefragt wird.

Kooperation zwischen Prozessen

- Kooperation = Arbeiten auf gemeinsamen Daten (Objekten)
- Kooperation erfordert → Synchronisation:
  a) Synchronisationsdienste (Kritische Abschnitte, Semaphore, ..)
  b) Kommunikation (Nachrichten)
- Kooperation erfordert Zugriff auf gemeinsame Datenstrukturen
  Threads → trivial (warum?)
  Prozesse → Betriebssystemunterstützung nötig
  (da getrennte Adressräume)
- Konzept des Gemeinsamen Speichers („Shared Memory“)
- Konzept von Sperren auf gemeinsamen Objekten („Locking“)

Gemeinsamer Speicher („Shared Memory“)



- P1 und P2 können beide auf die gleichen Daten im gemeinsamen Speicher zurückgreifen
- Gemeinsamer Datenbereich i.d.R. auf unterschiedliche Adressbereiche der jeweiligen Prozesse abgebildet (d.h. P1 greift z.B. via Adresse 0x3000000 auf Variable V zu und Prozess P2 über Adresse 0x5000000, aber beides ist derselbe Speicherort)
  → vgl. Virtual Memory (Kapitel 9)
- Synchronisation z.B. via kritische Abschnitte

Allgemeine Sperr-(„Lock“-)Operationen

- Verwendet für kombinierte Schreib-/Lesezugriffe auf gemeinsame Objekte (gemeinsame Speicherbereiche, gemeinsame Dateien, Teilmengen von Dateien)
- Unterschiedliche Sperrtypen („Lock Modi“)
  - Exclusive Write (nur 1 Schreiber, keine weiteren Leser)
  - Protected Write (nur 1 Schreiber, aber beliebig viele Leser)
  - Concurrent Write (beliebig viele Schreiber und Leser erlaubt)
  - Protected Read (gleichzeitig mehrere Leser aber keine Schreiber)
  - Concurrent Read (gleichzeitig weitere Leser und Schreiber erlaubt)

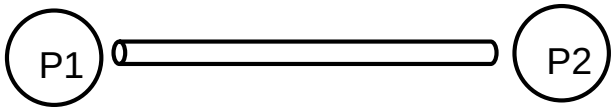
Mode	NL	CR	CW	PR	PW	EX
NL	Yes	Yes	Yes	Yes	Yes	Yes
CR	Yes	Yes	Yes	Yes	Yes	No
CW	Yes	Yes	Yes	No	No	No
PR	Yes	Yes	No	Yes	No	No
PW	Yes	Yes	No	No	No	No
EX	Yes	No	No	No	No	No

Aus: http://en.wikipedia.org/wiki/Distributed_lock_manager

- Parallele Operationen müssen „verträglich“ sein (z.B. Protected Write und Concurrent Read), sonst wird der sperrende Prozess blockiert.
- Locking ist auch zentrales Thema in → Datenbanksystemen

Beispiele für IPC-Dienste in UNIX/LINUX - 1

- (Named) Pipes:
  - spezieller 1:1-Pfad für kontinuierlichen gerichteten Zeichenstrom
  - FIFO-Auslieferung
  - Pipe hat spezifizierte Kapazität
  - Ist die Pipe voll, blockiert der Sender; ist die Pipe leer blockiert der Empfänger
  - Named Pipe: Dateideskriptor („Handle“), der dem Autorisierungskonzept von UNIX (Rechtevergabe) unterliegt
  - Schreibvorgang in Pipe: wie Schreiben in ein File
  - Lesevorgang von Pipe: wie Lesen aus einem File



- Sockets
  - Kommunikationskonzept für Netzwerkprogrammierung, häufig aber auch im lokalen Bereich eingesetzt („Unix Domain Sockets“)
  - bidirektional, 1:N, bytestream-orientiert
  - Mehr Details in der Vorlesung → „Computernetze“ (AI3, WIN4, WINp4)

Beispiele für IPC-Dienste in UNIX/LINUX - 2

- Signal Handling (Beispieltypen)

SIGABRT	Anormaler Abbruch des Prozesses
SIGALRM	Timer ist abgelaufen
SIGFPE	Fehler bei mathematischer Operation (z.B. Overflow)
SIGHUP	Terminal-Hangup
SIGILL	Illegale Maschineninstruktion
SIGINT	Drücken von CTRL/C
SIGKILL	Normale Prozessbeendigung (nicht ignorierbar)
SIGPIPE	Schreiben einer Pipe ohne Leser
SIGQUIT	Prozessbeendigung mit Dump (CTRL/ \)
SIGSEGV	Zugriff auf ungültige Speicheradresse
SIGTERM	Anforderung der geordneten Prozessbeendigung
SIGUSR1	Verfügbar für anwendungsbezogene Zwecke
SIGUSR2	Verfügbar für anwendungsbezogene Zwecke

- Hauptfunktionen:

```
sigaction()    - Einrichten eines Handlers
sigprocmask()  - Maskieren von Signals
sigsendset()   - Senden eines Signals
kill()         - Spezielles Kill-Signal
sigsuspend()   - Warten auf Signal
```

- Signal Handling (Beispiel)

```
#include <signal.h>

static int flag = 0;

void handler (int signal_number)
{    flag = 1;
}

int main ();
{ struct sigaction sa;
  memset (&sa, 0, sizeof(sa));
  sa.sa_handler = handler;
  sigaction (SIGUSR1, &sa, NULL);
  while (1==1)
  {  if (flag==1)
     { flag=0;
       ... act on signal..
     };
     ...
     // do other tasks //
     ...
  }
```

Shared Memory in UNIX/LINUX

- Dienste „shmget(), shmat(), shmdt()“
- Prozesse können den gemeinsamen Speicher an frei wählbare Adresse ihres Adressraumes legen.
- Prozesse müssen für Synchronisation sorgen (z.B. über Semaphore)
- Auch gemeinsamer Zugriff auf Files möglich („mmap()“)

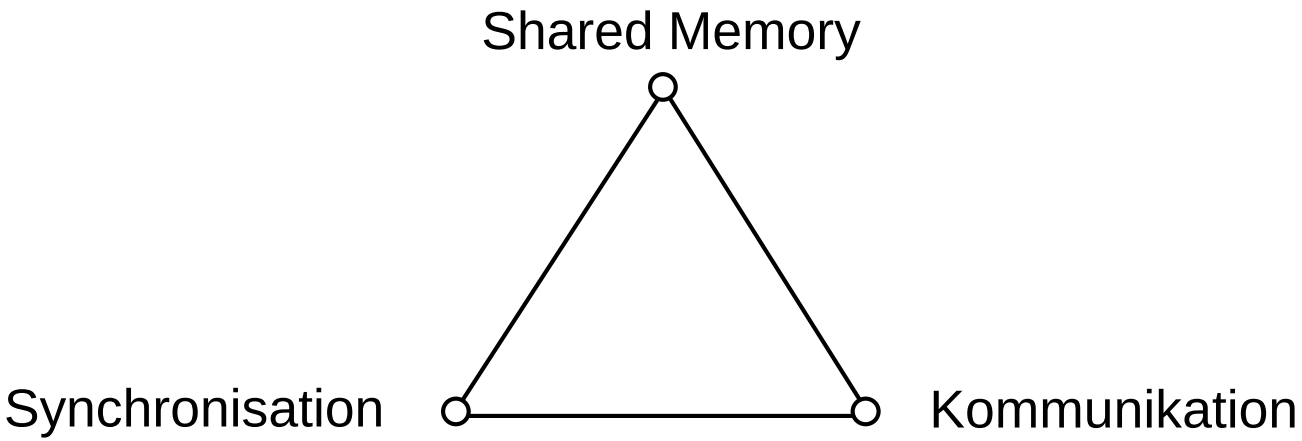
```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size.  */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

Beispiele für IPC-Dienste in Windows

- (Anonymous) Pipes
  - nur lokal auf einem Rechner, unidirektional
  - „CreatePipe“, danach File-Operationen
- Named Pipes
  - nicht auf einen Rechner beschränkt
  - auch bidirektional möglich (PIPE_ACCESS_DUPLEX)
  - „CreateNamedPipe()“ Systemaufrufe, Filefunktionen
  - zusätzliche Unterscheidung in 2 Pipe-Typen
    a) PIPE_TYPE_BYTE (wie UNIX, d.h. Bytestream, keine Nachrichtengrenzen)
    b) PIPE_TYPE_MESSAGE (feste Nachrichtengrenzen)
  - Optional 1:N Pipes (1 server, N clients)
  - „PeekNamedPipe()“ Systemaufruf für Vorschau auf Pipe-Inhalt
- Mailslots
  - einfache Mailbox
  - „CreateMailSlot“, Fileoperationen zum Lesen/Schreiben
  - erlauben Broadcast (1:N) , 1 Writer, N Reader
- Shared Memory (ähnlich UNIX, mit File-Mapping)

Zusammenhänge

- Synchronisation und Kommunikation sind verwandt:

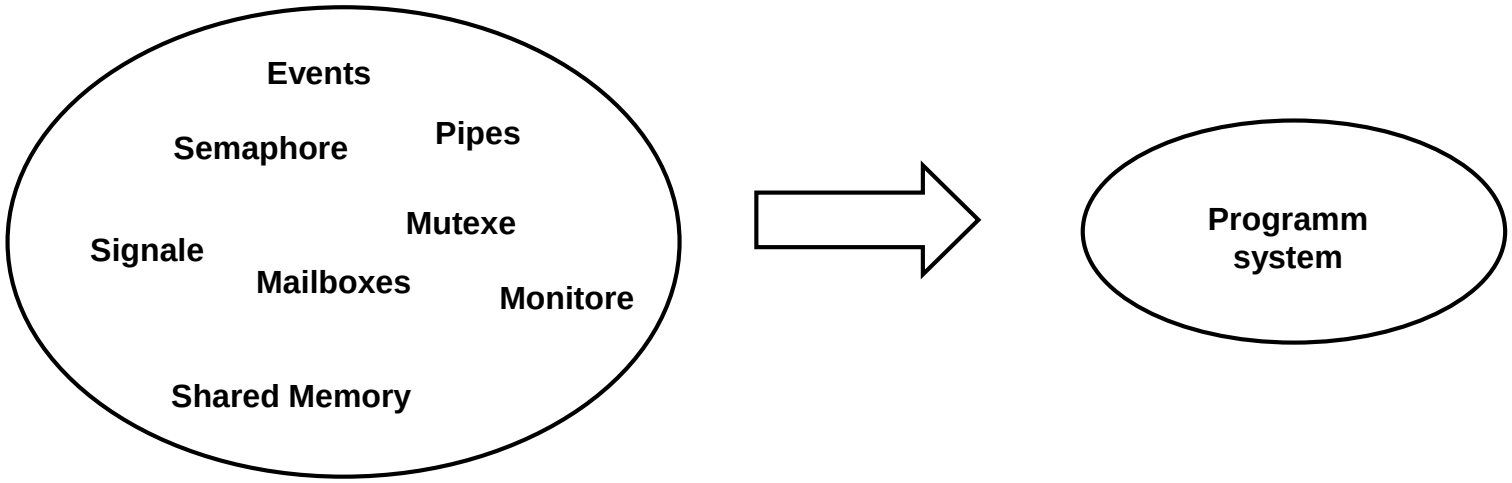


Beispiele:

- Wie kann man mit Shared Memory und Semaphoren einen Mailbox-Dienst aufbauen?
- Wie kann man mit Nachrichtenaustausch einen kritischen Abschnitt realisieren?

Systemprogrammierung

- Der Systemprogrammierer wählt aus der Menge der Synchronisations- und Kommunikationsdienste eines Betriebssystems die passende Untermenge und parametriert diese zur Realisierung einer Lösung für eine bestimmte Umgebung.



- Auswahlkriterien: Problemtyp, Anzahl beteiligter Akteure, Interprozess- oder Intraprozess(Thread-)kommunikation, Single-/Multiprozessor- System, Datenvolumen, Synchronisationsanforderungen
- Zielkriterien: Konsistenz, Effizienz, (Echt-)Zeittreue