
Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

Kommunikation und Synchronisation von Prozessen

Lernziele

- Sie beurteilen die Eignung nachrichten- und speicherbasierter Verfahren zur Kommunikation zwischen parallelen Abläufen.
- Sie analysieren Anwendungsprobleme von Unix-Pipes.
- Sie unterscheiden zwei Implementierungsarten von Monitoren.
- Sie erläutern das Grundprinzip eines Rendezvous paralleler Prozesse.
- Sie entwickeln eigene Programme, die Berkeley-Sockets bzw. RPC zur rechnerübergreifenden Datenverarbeitung nutzen.

Taxonomie



Wie können Prozesse kommunizieren? Kommunikation ist örtlich, temporal und zustandsbasiert mit Daten.

- IPC: Interprocess Communication
- Ein Prozess wartet auf die Terminierung eines anderen (Eltern-Kind Gruppe);
- Ein Prozess wartet auf Daten von einem anderen; Beispiel: Anforderung von Daten eines Web Servers über HTTP (remote) oder ein Prozess wartet auf Daten einer Datenbank;
→ **Kooperation**
- Prozesse wollen quasi gleichzeitig (oder zeitlich sehr nahe beieinander) einen gemeinsamen Punkt im Programmfluss erreichen; *Barriere*
- Prozesse greifen auf geteilte Ressourcen quasi parallel zu und der Zugriff muss sequenzialisiert werden; **Wettbewerb**
- Prozesse warten auf zeitliche Ereignisse (ist es der gleiche Timer und mehrere Prozesse warten darauf dann ist das IPC)

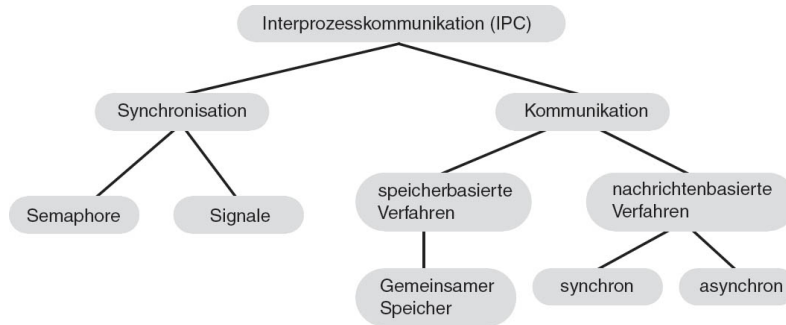
Taxonomie

1. Synchron versus asynchrone Kommunikation (mit Signalen oder Nachrichten)
 - Synchron bedeutet dass auf eine Antwort oder Bestätigung gewartet wird (mit Prozessblockierung und Prozesswechsel)
 - Asynchron bedeutet dass der Prozess nicht wartet und ggf. später durch eine Rückruffunktion (ähnlich Interruptbetrieb) eine Antwort erhält
2. Wir unterscheiden Signale (Events) und Nachrichten mit Daten; beide können zu einer Synchronisation von Prozessen führen;
3. Synchronisation kann Daten enthalten - aber kein Muss.
4. Es gibt eine Vielzahl von IPC Objekten. Auch eine Datei kann dazu gehören (Schreib/Lese Sperren)
 - Geteilte Speicherbereiche
 - Dateien, Pipes (Warteschlangen über das Dateisystem)
 - Semaphore
 - Nachrichten
5. Synchronisation führt i.A. zur Prozessblockierung und Prozesswechsel bzw. Warten.

Taxonomie



Prozesse und Threads können lokal auf dem gleichen Rechner oder mithilfe der Netzwerksoftware des Betriebssystems in einer verteilten Umgebung kommunizieren.



[bsdip]

Abb. 1. Überblick über Synchronisation und Kommunikation

Taxonomie

- Es wird unterschieden zwischen den zwei Gruppen »Synchronisation« und »Kommunikation«.
- Bei der Kommunikation wird weiter unterteilt in »speicherbasierte Verfahren« und »nachrichtenbasierte Verfahren«.
- Die Kommunikation unterscheidet sich von der Synchronisation, indem explizit Daten zwischen den Prozessen ausgetauscht werden und nicht nur eine Koordination im zeitlichen Ablauf oder Ressourcenzugriff realisiert wird.

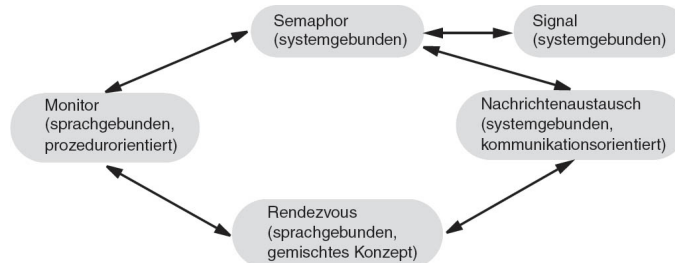


Abb. 2. IPC-Verfahren aus Sicht der Programmierung

Taxonomie

- Mit der Eigenschaft »systemgebunden« ist gemeint, dass das Verfahren als Funktionsaufruf der Betriebssystem-Programmierschnittstelle zur Verfügung steht.
 - Mit der Eigenschaft »sprachgebunden« wird der Fall umschrieben, bei dem ein Funktionsaufruf des Laufzeitsystems der Programmiersprache bzw. der Sprachbibliothek zur Verfügung steht.
-
- Jedes der Programmierkonzepte weist bezüglich einer bestimmten Problemklasse seine Vorzüge auf.
 - Wesentlich ist jedoch, dass alle Konzepte funktional äquivalent sind, d.h., jedes Konzept kann in seiner Wirkungsweise durch jedes andere nachgebildet werden.

Nachrichtebasierte Verfahren

Unter nachrichtebasierten Verfahren fassen wir alle Mechanismen zusammen, die einen Datenaustausch zwischen Prozessen und Threads mithilfe von Systemfunktionen bewerkstelligen.

Allgemeine Aspekte

Datenabgrenzung bei der Kommunikation

Die **Datenabgrenzung** bei der Kommunikation hängt von der Form der ausgetauschten Daten ab. Diese ist bestimmend für mögliche Lösungen der Datenabgrenzung.

1. Datenaustausch mittels Nachrichten (message passing)
2. Datenaustausch mittels Datenströmen (streaming)
3. Datenaustausch mittels Paketen (packeting)

- Beim Datenaustausch mittels Nachrichten haben wir eine abgegrenzte Datenmenge bei der Kommunikation in Form der Meldung (message).
 - Die Meldungsgröße kann systemabhängig fest oder variabel sein.

Nachrichtebasierte Verfahren

- Kommen hingegen für den Datenaustausch **Datenströme** zur Anwendung, so ist die Nachrichtengrenze für den Sender und den Empfänger unsichtbar.
 - Die übertragbare Datenmenge ist theoretisch unbeschränkt.
 - Typischerweise werden die Dateisystemfunktionen `read()` und `write()` für eine stromartige Datenübertragung genutzt.



E/A ist auch Kommunikation, aber zunächst nur zwischen Prozessen und Geräten.

Beim Datenaustausch mittels Paketen kommen feste, oft standardisierte Datenformate zum Einsatz.

Nachrichtebasierte Verfahren

- Sie sind im Rahmen von **Kommunikationsprotokollen** definiert (z.B. IP-Paketformat).
 - Für die Applikationsprogrammierung sind die Pakete nicht sichtbar (transparent).
 - Beim Übertragen der Pakete kann eine Fragmentierung (= Aufteilung in Teilpakete) und eine Defragmentierung (= Zusammensetzen aus Teilpaketen) stattfinden.
 - Dies wird versteckt (hidden) durch die Netzwerksoftware realisiert.
- Der Datenaustausch mittels Paketen ist ein Thema der **Computernetze** und wird daher hier nicht weiter berücksichtigt. Die zwei verbleibenden Arten des Datenaustauschs wollen wir jedoch anhand von Realisierungsformen kennenlernen.

Synchronität bei der Kommunikation

Unter einer synchronen Kommunikation versteht man den Fall, dass der Sender warten muss, bis der Empfänger zur Entgegennahme der Daten bereit ist (Rendezvous). Bei der asynchronen Kommunikation läuft der Sender hingegen weiter, auch wenn der Empfänger gerade nicht für den Datenempfang bereit ist.

Synchronität bei der Kommunikation

- Bei der synchronen Kommunikation muss entweder der Sender (A1) oder der Empfänger (A2) warten, wenn nicht gerade das Senden und Empfangen zeitlich zusammenfallen.
- Bei der asynchronen Kommunikation (B) kann hingegen das Senden sofort erfolgen, auch wenn der Empfänger nicht gerade für den Empfang bereit ist. Es ist also ein Puffer für die Zwischenlagerung von Daten, ein sogenannter Nachrichtenpuffer, notwendig.

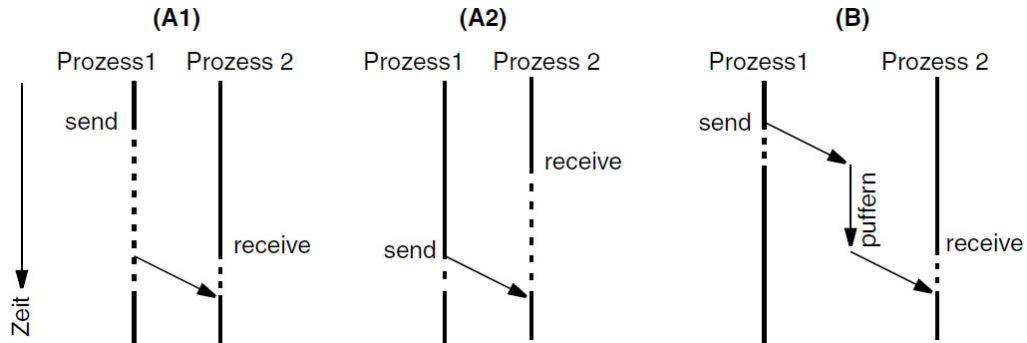


Abb. 3. Synchrone und asynchrone Kommunikation

Synchronität bei der Kommunikation

- Die asynchrone Kommunikation entkoppelt den Ablauf des Senders und des Empfängers und hilft, Geschwindigkeitsunterschiede in der Datenverarbeitung auszugleichen.
 - Genau genommen läuft bei der asynchronen Kommunikation der Sender erst dann weiter, wenn die Nachricht vollständig in einen Zwischenspeicher kopiert wurde.
 - Dies ist in der Abbildung durch die schräg verlaufenden Pfeile angedeutet und gilt auch für den Nachrichteneingang.

```
send    (destination, &message);  
receive (source,      &message);
```

Code 1. Programmierschnittstelle für die synchrone und asynchrone Nachrichtenkommunkation. Bei einer synchronen Art können beide Funktionen den Prozess blockieren. Ein Nachrichtenpuffer `message` trägt eine Identifikation (Textname, Deskriptor, Handle o.Ä.).

Synchrone versus Asynchrone Kommunikation

[bsdp]

Synchrone Kommunikation	Asynchrone Kommunikation
<ul style="list-style-type: none"> + Automatische Ablaufsynchronisation + Kein Zwischenpuffer nötig - Kein unabhängiger Programmablauf möglich (eingeschränkte Parallelität) 	<ul style="list-style-type: none"> + Unabhängige Programmabläufe - Evtl. Probleme, wenn Puffer voll und Empfänger nicht bereit (Verklemmung) - Unbestimmte Übertragungszeit

Tab. 1. Vergleich von synchroner und asynchroner Kommunikation



Bei der synchronen Kommunikation übergibt der Sender die Nachricht direkt an den Empfänger. Typischerweise wird der Datenaustausch in dieser Situation in der Art eines Rendezvous erledigt. Nur wenn sowohl Sender als auch Empfänger bereit sind, kann die Datenübertragung stattfinden.

Synchrone versus Asynchrone Ein- und Ausgabe

- Auch bei der Gerätekommunikation kann A. vs S. unterschieden werden.
 - Bei der S. Kommunikation muss beim Lesen und Schreiben der Prozess warten bis die Operation ausgeführt wurde (und beim Lesen die Daten verfügbar sind).
 - Bei der A. Kommunikation kann der Prozess eine E/A Operation starten aber mit der Programmausführung fortsetzen. Der Prozess kann auch mehrere E/A Operationen starten.
 - Nur dann ist die Frage wie er mitbekommt wann E/A Operationen ausgeführt wurden (also auch Daten verfügbar sind).
- Viele Programmiersprachen sind vom Paradigma her strikt sequenziell. Es ist nicht möglich mehrer Operationen parallel auszuführen.
- A. E/A ist daher in Virtuellen Maschinen eine gängige Möglichkeit auf der Programmierenebene das strikt sequenzielle Paradigma aufrecht zu erhalten und unter der Motorhaube E/A Operationen mit Threads parallel auszuführen.
- Mittels Callback Funktionen können dann erfüllte Operationen ihr Ergebnis wieder an das Programm übermitteln.

Synchrone versus Asynchrone Ein- und Ausgabe

```
function callback(status,data) {
  if (!status) print("Error:",status)
  else if (data) processData(data)
}
send('edu-9.de','mymessage1',callback) // Time 1
send('ag-0.de','mymessage2',callback) // Time 2
receive('edu-9.de',callback) // Time 3
receive('ag-0.de',callback) // Time 4
print('I am finished with my computation. Waiting for I/O completion')
// Time 10: callback(...)
// Time 12: callback(...)
```

Def. 1. Asynchrone E/A mit Callback Funktionen am Beispiel von JavaScript

- Das Problem: Die Callbachöhle, Verschachtelung, und schwierige Fehlerbehandlung.
- Z.B. wenn die zweite Send Operation auf Daten von der ersten Receive Operation warten muss ist eine Verlegung der Send Operation in den Callback Handler notwendig!

Vergleich von Kommunikationsverfahren

[bsdip]

Gemeinsamer Speicher (Lesen und Schreiben in gemeinsamen Speicherbereich)	Systemgestützter Datenaustausch (Senden und Empfangen von Nachrichten oder Datenströmen)
+ Geschwindigkeit + Transparenz (alles unter Selbstverwaltung) - Nur bei gemeinsamem Speicher möglich - Zusätzliche Synchronisation zur Erhaltung der Datenkonsistenz nötig - Transparenz (Software-Strukturierung) - Erlaubt keine saubere Kapselung der Softw. - Durchbricht Isolation durch getrennte Adressräume	+ Rechnerübergreifend verwendbar + »Eingebaute« Synchronisation - Effizienz (lokal & rechnerübergreifend)

Tab. 2. Vergleich von Kommunikationsverfahren: Speicherbasiert versus nachrichtenbasiert.

Ein Datenaustausch mittels Speicher kann mithilfe von Systemfunktionen erfolgen oder in Selbstverwaltung, nachdem ein gemeinsamer Speicherbereich eingerichtet wurde.

- Bei der Selbstverwaltung muss in der Regel mittels Semaphoren die Koordinierung der beteiligten Threads bzw. Prozesse sichergestellt werden.
- Somit stellt der Einsatz gemeinsamen Speichers meist eine Mischform dar, die sich für den schnellen Transfer großer Datenmengen besonders anbietet.

Geteilter Speicher



Threads besitzen ein inhärentes geteiltes Speichermodell und können somit unmittelbar ohne weitere System- und Hilfsfunktionen über geteilte Speichervariablen kommunizieren, anders als isolierte Prozesse.

- Prozesse können sich über das Betriebssystem einen Speicherbereich teilen.
 - Dieser ist aber nur ein Bytepuffer ohne Datentyp.
 - Ein sinnvoller Datentyp kann mit C durch Typecasting erstellt werden.

Geteilter Speicher

```
void* create_shared_memory(size_t size) {
    // Our memory buffer will be readable and writable:
    int protection = PROT_READ | PROT_WRITE;
    // The buffer will be shared (meaning other processes can access it), but
    // anonymous (meaning third-party processes cannot obtain an address for it),
    // so only this process and its children will be able to use it:
    int visibility = MAP_SHARED | MAP_ANONYMOUS;
    // The remaining parameters to `mmap()` are not important for this use case,
    // but the manpage for `mmap` explains their purpose.
    return mmap(NULL, size, protection, visibility, -1, 0);
}
...
void* shmem = create_shared_memory(16);
int * counter = &((int*)shmem)[0];
*counter=0;
int pid = fork();
if (pid == 0) { // child process
    (*counter)++;
} else { // parent process
    (*counter)--;
}
printf("[%d] counter[%x]=%d\n",pid,counter,*counter);
```

Geteilter Speicher



Welchen Wert hat die Zählervariable am Ende? Gibt es Gebote? Wir nehmen einen Mehrkernrechner an.

Geteilter Speicher



Welchen Wert hat die Zählervariable am Ende? Gibt es Gebote? Wir nehmen einen Mehrkernrechner an.



Keine Ahnung. Die möglichen Ergebniswerte von *counter* können $\{-1,0,1\}$ sein. Das Ergebnis ist nicht deterministisch.

- Und das ist das Problem bei SHM Kommunikation: Es gibt keine Programmflusssteuerung! Diese muss dann zusätzlich z.B. mit Semaphoren erfolgen.

Vergleich von Kommunikationsverfahren

(a) Message passing. (b) shared memory.

[Andrew Forney]

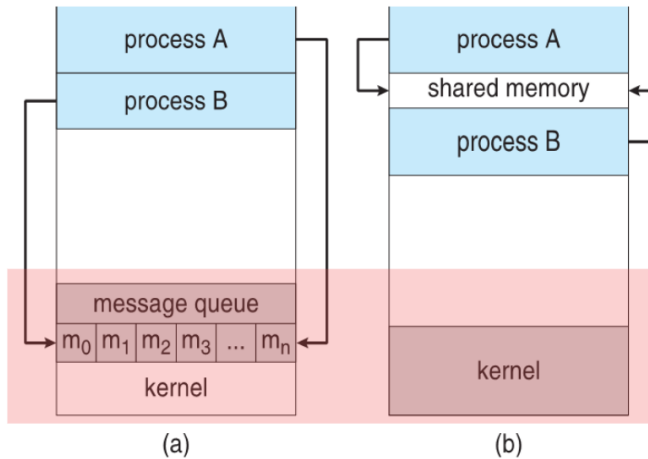


Abb. 4. Vergleich von Kommunikationsverfahren: Nachrichtenbasiert versus speicherbasiert.

Verbindungsorientierung

- Die Verbindungsorientierung spielt eine Rolle beim asynchronen Datenaustausch, wenn wiederholt Datenübertragungen mit gleichen Kommunikationspartnern anstehen.
- Unabhängig davon, ob man streng abgegrenzte Nachrichten oder einfach Datenströme austauschen will, können zwei Verbindungsformen unterschieden werden.

1. Arbeitet man **verbindungsorientiert (connection-oriented)**, so muss zuerst ein logischer Verbindungskanal eingerichtet werden. Dazu muss die Identität des Empfängers bekannt sein.
2. Ist die Verbindung aufgebaut, so werden Daten an den Verbindungskanal übergeben, der eine Kennung trägt.
3. Die Angabe der Empfängeridentität ist dann nicht mehr nötig.
4. Sind alle Daten ausgetauscht, was typischerweise nach einer Vielzahl an einzelnen Datentransfers zutrifft, wird die Verbindung explizit wieder abgebaut.
 - Dies kann über eine »disconnect«-Funktion oder ein Schließen des Verbindungskanals erfolgen.
 - Je nach Form des Verbindungskanals sind nur zwei Kommunikationspartner erlaubt, oder es sind auch m:n-Konstellationen möglich.

- Der logische Verbindungskanal kann zu jedem Zeitpunkt nur einen Datentransfer in eine Richtung (unidirektional) oder gleichzeitig in beide Richtungen erlauben (bidirektional).
- Der unidirektionale Datenaustausch (mit alternierender Richtung) wird als Halbduplex-, der bidirektionale Datenaustausch als (Voll-) Duplexbetrieb bezeichnet.
- Ist generell nur eine Kommunikationsrichtung möglich, so handelt es sich um einen Simplex-Betrieb.



Abb. 5. Halb- und Vollduplex-Betrieb

- Ein **verbindungsloser (connectionless, datagram-oriented)** Datenaustausch benutzt für jeden einzelnen Datentransfer die Identität des Empfängers als Adresse.
- Es können sofort Daten übertragen werden, ohne dass zuerst ein Verbindungskanal angelegt werden muss.



Diese Kommunikationsform empfiehlt sich, wenn nur einzelne Datentransfers, aber möglicherweise an viele verschiedene Empfänger, nötig sind.

[bsdip]

Verbindungsorientierte Kommunikation	Verbindungslose Kommunikation
+ Nachrichtenreihenfolge gewährleistet + Zuverlässige End-End-Verbindung + Timeout-Überwachung und Neuübertragung + Meist mit Datenflusssteuerung verbunden - Verbindungsauf- und -abbau nötig (unattraktiv für wenige Nachrichten)	+ Kein Verbindungsauf-/abbau nötig (attraktiv für vereinzelte Meldungen bzw. viele verschiedene Empfänger) - Nachrichtenreihenfolge nicht gewährleistet - Nachrichtenverlust möglich - Empfängeradressierung pro Nachricht

Tab. 3. Vergleich von verbindungsorientierter und verbindungsloser Kommunikation.

Empfängeradressierung

- Neben der Frage wie zwei Prozesse miteinander kommunizieren können und sich "finden" kann man adressierte Kommunikation in verschiedene Klassen einteilen.
- Diese Klassen bestimmen die Empfänger von Nachrichten (lösen aber nicht das Problem der eindeutige Adressierung der Prozesse an sich)

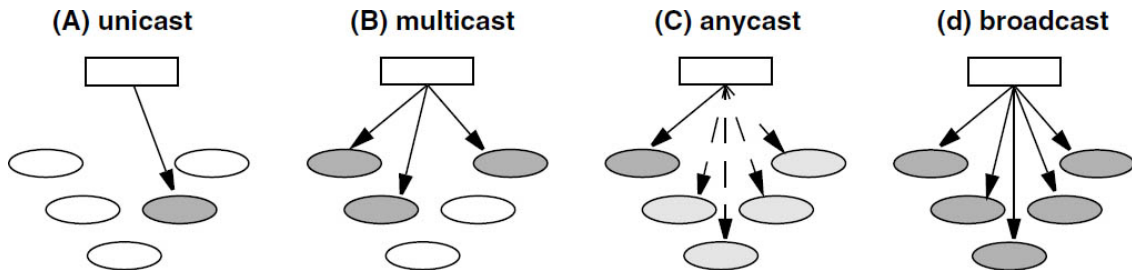


Abb. 6. Empfängeradressierung bei der Kommunikation

Empfängeradressierung

Bei einem unicast (1:1) wird genau ein bestimmter Empfänger angesprochen. Bei einem multicast (1:m) wird eine definierte Gruppe an Empfängern (m) adressiert. Der anycast übermittelt Daten an irgendeinen Empfänger, mindestens einen einzigen (1:1..n). Der broadcast schließlich geht an alle Empfänger (n), die sich finden lassen (1:n). Welche Empfängeradressierung eine Lösung unterstützt, ist implementationsabhängig. Meistens ist es aber nur der unicast.

Prioritäten

Erfolgen bei einem Datenaustausch alle Übertragungen mit der gleichen Priorität, so werden sie bei einer Zwischenpufferung in der FIFO-Reihenfolge abgelegt.

- Können unterschiedliche Prioritäten benutzt werden, so ist es möglich, neue Daten vor alten Daten in den Nachrichtenpuffer einzutragen. Bei Datenströmen sind Prioritäten meistens nicht sinnvoll, daher ist diese Möglichkeit in der Regel dem Meldungs austausch vorbehalten.
- Im einfachsten Fall sind zwei Prioritäten definiert, nämlich »normal« und »dringend«. Anspruchsvollere Lösungen können feinere Prioritätsabstufungen anbieten.

Anwendungsszenario Client/Server

Der gepufferte Nachrichtenaustausch eignet sich zur Realisierung eines Client/Server-Betriebs.

- Client: Prozess oder Thread, der eine Dienstleistung beansprucht (Auftraggeber).
- Server: Prozess oder Thread, der eine Dienstleistung erbringt (Auftragnehmer).
Beispiele: Dateiverzeichnis liefern, Faxmeldung übermitteln.
- Funktionsweise: Ein Client beansprucht vom Server eine Dienstleistung und sendet dem Server deshalb eine Meldung. Wenn der Server die notwendigen Arbeiten erledigt hat, liefert er mittels einer Meldung die Antwort an den entsprechenden Client zurück.
- Absenderkennung: Da Meldungen von sich aus in der Regel keine Absenderangaben enthalten, muss die Absenderadresse bzw. die Bezeichnung der Nachrichtenwarteschlange für die Rückmeldung in der Meldung selbst enthalten sein.

Anwendungsszenario Client/Server

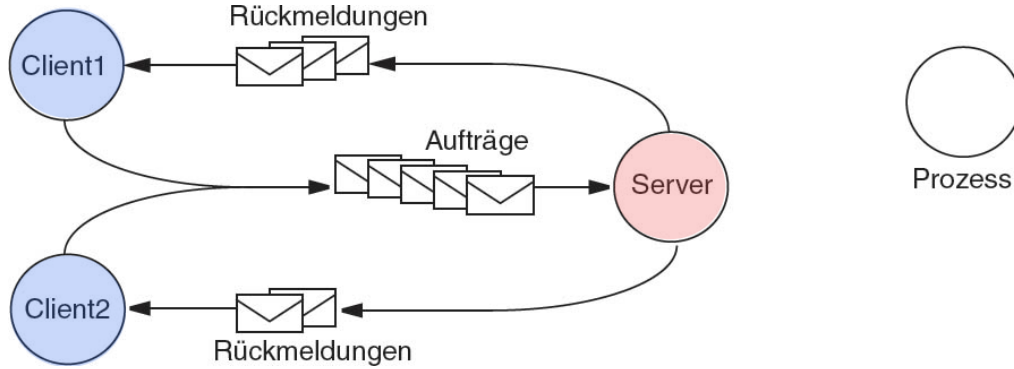


Abb. 7. Client/Server-Prinzip mittels Meldungen

Pipe als Verbindungskanal zweier Prozesse

- Eine namenlose Pipe für die Kommunikation zweier Prozesse anzulegen erfordert insgesamt drei Prozesse und die folgenden Schritte:

1. Der Elternprozess erzeugt die Pipe durch Aufruf von `pipe()`. Er erhält für beide Enden der Pipe die vom Betriebssystem zugeteilten Deskriptoren.
2. Der Elternprozess erzeugt nun zwei Kindprozesse (oder er selbst ist ein Kommunikationspartner).
3. Diese erhalten automatisch Kopien der Pipe File Deskriptoren.
4. Jeder Prozess schließt mittels `close()` die Enden der Pipe, die er nicht braucht.
5. Der Elternprozess schließt beide Enden, Kindprozess 1 das »obere« Ende und Kindprozess 2 das »untere« Ende.
6. Nun kann Kindprozess 1 mittels `write()` Daten über die Pipe an Kindprozess 2 senden, der diese mittels `read()` entgegennimmt.

Pipe als Verbindungskanal zweier Prozesse

```

int main()
{
    int fds[2];                // Für Pipe-Dateideskriptoren
    char *text = "Hallo da!\n"; // Zu transferierender Text
    char buffer [5];          // Lesebuffer
    int count, status;        // Leseanzahl bzw. Endstatus
    pipe(fds);                // Unnamed pipe erzeugen
    if (fork() == 0) {        // Kindprozess 1:
        dup2(fds[1], 1);      // stdout auf Pipe-Eingang legen
        close(fds[0]);        // Pipe-Ausgang schließen
        write(1, text, strlen(text)+1); // Text in Pipe schreiben
    } else if (fork() == 0) { // Kindprozess 2:
        dup2(fds[0], 0);      // Pipe-Ausgang auf stdin legen
        close(fds[1]);        // Pipe-Eingang schließen
        while ((count = read(0, buffer, 4)) != 0) { // Wiederholt auslesen
            buffer[count]= 0; // »string terminating zero«
            printf("%s", buffer); // Laufend ausgeben
        }
    } else {                  // Elternprozess:
        close(fds[0]);        // Pipe-Ausgang schließen
        close(fds[1]);        // Pipe-Eingang schlieÙe
        wait(&status);        // Warte auf erstes Kind
        wait(&status);        // Warte auf zweites Kind
    }
    exit(0);
}

```

Bsp. 2. Ein einfaches Zwei-Prozess-Kommunikationssystem über Pipers

Benannte Pipes

Hier werden die Pipes als "virtuelle" Dateien über ds Dateisystem von Prozessen mit üblichen Read und Write Operationen geteilt.

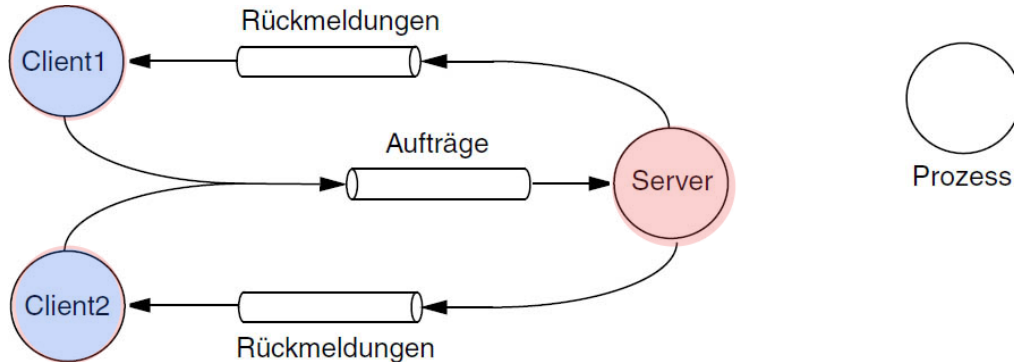


Abb. 8. Client/Server-Konstellation mit benannten Unix-Pipes

Message Queues



Die Unix-Pipes sind gut geeignet für die Übertragung von Daten, wenn eine klare Abgrenzung der einzelnen gesendeten Daten voneinander nicht notwendig ist (Bytestrom). Stehen gut definierte Meldungen im Vordergrund, so bieten sich die Message Queues an.

- Sie übertragen Meldungen unterschiedlicher Länge.
- Die einzelnen Meldungen sind so gegeneinander abgegrenzt, dass beim Lesen der Message Queue jeweils genau eine Meldung entnommen wird.
- Der Inhalt und die Länge einer einzelnen Meldung sind frei wählbar, wobei die Meldungsgröße nicht den für die Message Queue festgelegten Maximalwert überschreiten darf (`attr.mq_msgsize`).
- Beim Auslesen einer Meldung wird dem Empfänger die genaue Länge der Meldung mitgeteilt.
- Die in der Message Queue gepufferten Meldungen sind entsprechend ihrer Meldungspriorität sortiert, wobei Meldungen der gleichen Priorität eine FIFO-Reihenfolge einnehmen.
- Die Meldungspriorität wird durch den Sender zusammen mit der Meldung der Message Queue übergeben.

Message Queues

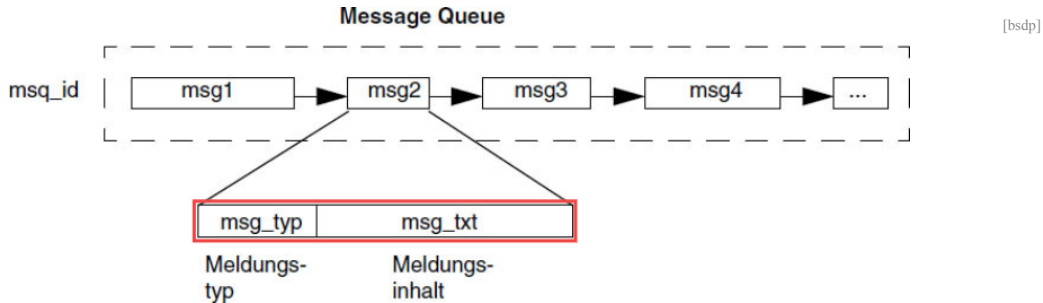


Abb. 9. Aufbau einer Message Queue

Eine Meldung besteht immer aus den zwei Teilen `msg_typ` und `msg_txt`, die den Meldungstyp und den Meldungsinhalt repräsentieren. Wie in einer eigenen Applikation ein Meldungstyp deklariert werden kann, zeigt nachfolgendes Beispiel:

```
struct meine_nachricht {
    long msg_typ;
    char msg_txt[100];
}
```

Mutualer Ausschluss und Semaphor

Bisher wurde nachrichtenbasierte Kommunikation eingeführt. Neben dem Nachrichtenaustausch (Kooperation) ist aber auch die Konfliktlösung bei Wettbewerb wichtig, also eine sequenzielle Synchronisation.



Vorteilhafter als eine Lösung mit Selbstverwaltung ist der Einsatz von sogenannten Semaphoren. Es handelt sich dabei um Betriebssystem- und Synchronisationsobjekte für die Absicherung und Synchronisation von Prozessen bzw. Datenkonsistenz. Der Semaphor und ihre Operationen P und V wurden 1962 von E. Dijkstra erstmals systematisch entwickelt und dargestellt

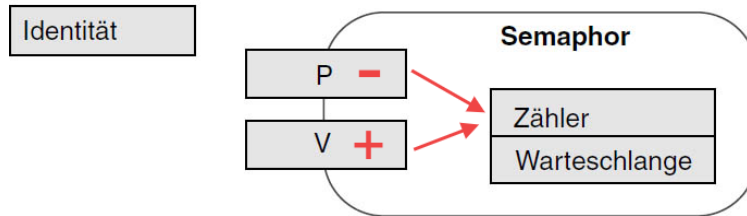
Wir unterscheiden wieder zwei Arten der Prozessinteraktion:

1. Kooperation, d.h. Produzenten-Konsumenten Anwendungen
2. Wettbewerb, d.h. Konflikte bei der Nutzung von geteilten Ressource.

Der Semaphor kann zur Synchronisation für beide Arten verwenden werden.

Mutualer Ausschluss und Semaphor

[bsdip]



E. Dijkstra

P = Passering (holländ.
= durchlassen)

V = Vrijgave (holländ.
= freigeben)

Abb. 10. Semaphor mit einer Identität (Instanzenbezeichnung), den Operationen P/V und den Attributen Zähler (für Marken) und Warteschlange (für Prozesse)

Der Semaphor implementiert einen geschützten Zähler (Marken, Tokens) mit einer Prozesswarteschlange:

1. Invariante: Der Zähler darf nie negativ werden
2. Prozesse werden blockiert die diese Invariante verletzen würden
3. Es gibt zwei Operationen: V(+) oder Up und P(-) oder Down. Beide Operationen verändern den Zähler um den Wert 1.

Mutualer Ausschluss und Semaphor

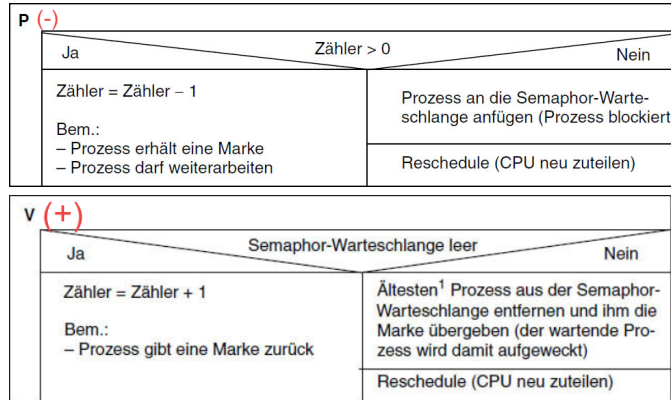


Abb. 11. (Oben) Definition der P-Operation: Der aufrufende Prozess erhält entweder sofort eine Marke oder muss warten, bis eine verfügbar wird (dazu wird er an die Warteschlange des Semaphors angefügt). (Unten) Definition der V-Operationen: Der aufrufende Prozess übergibt eine Marke entweder dem Semaphor oder einem am Semaphor wartenden Prozess, der damit wieder weiterlaufen kann.

Semaphortypen

Es existieren zwei Grundtypen von Semaphoren:

1. Binärer Semaphor (binary semaphore, mutex): Es sind genau 0 oder 1 Marke erlaubt \Rightarrow Lösung des mutualen Ausschlussproblems (entweder oder), Mutex Lock
2. Zählsemaphor (counting semaphore, general semaphore): Es sind beliebig viele Marken erlaubt, dies stellt die universellere Form dar.

Unteilbarkeit von P/V

Die Unteilbarkeit der P- und V-Operationen muss durch ihre Implementierung sichergestellt werden. Mögliche Lösungen dafür sind:

- Ausschalten der Prozessorneuzuteilung für die Operationsdauer
- Vorübergehend höhere Priorität als alle anderen Prozesse
- Ausschalten aller Interrupts (= Gewaltlösung) während der P- und V-Operationen; ist aber nur für Einprozessorsysteme ausreichend
- Verwendung atomarer Prozessorinstruktionen (v.a. der TAS-Befehl, siehe S. 201)

Aktives oder passives Warten

Eine zweite Implementierungsfrage betrifft die Art des Wartens an der Semaphore-Warteschlange.

- In der Regel heißt dies, dass der Prozess für die Dauer des Wartens blockiert wird, d.h. die CPU freigibt und in den Zustand »Wartend« wechselt (passives Warten). Dies erlaubt anderen Prozessen die Wartezeit zu nutzen.
- Unter speziellen Bedingungen kann jedoch ein aktives Warten die bessere Lösung sein:
 - Sehr kurze Wartezeit: Der Zeitbedarf für ein Blockieren kann größer sein als für ein aktives Warten. Das Blockieren und spätere Aufwecken des Prozesses beinhaltet zwei Prozessumschaltungen, für die eine zwar kleine, aber doch gegebene Zeitdauer nötig ist.
 - Multiprozessorsysteme: Da eine echte Parallelität möglich ist, wird in verschiedenen Situationen die Wartezeit sehr klein sein. Dies macht ebenfalls ein Blockieren unattraktiv.

Aktives oder passives Warten

Für diese speziellen Situationen bieten manche Betriebssysteme einen speziellen Semaphortyp an, der als Spinlock bezeichnet wird (ein passender deutscher Begriff fehlt).

- In einer ersten Variante wartet ein Spinlock beliebig lange in einer aktiven Warteschlange.
 - Dies ist unattraktiv, wenn nicht bekannt ist, wie lange das Warten dauert.
- Eine zweite Variante wartet in einer aktiven Warteschleife, kennt dafür aber ein Zeitlimit.
 - Wird dieses überschritten, so wird ein passives Warten, d.h. ein Blockieren des Prozesses, durchgeführt.

Wartereihenfolge

- Eine dritte Implementierungsfrage betrifft die Einreihung in der Semaphor-Warteschlange.
 - Erfolgt dies nach dem FIFO-Prinzip, so ist sichergestellt, dass jeder Prozess in endlicher Zeit drankommt, sofern die einer P-Operation folgende V-Operation stets in endlicher Zeit ausgeführt wird.
 - Wird für die Einreihung die Prozesspriorität benutzt, so besteht die Gefahr, dass niederpriore Prozesse nie drankommen (verhungern), wenn hochpriore Prozesse genügend oft die P-Operation aufrufen.

Anwendung der Semaphoren

Eine hervorragende Bedeutung fällt Semaphoren bei der Absicherung kritischer Bereiche zu, indem sie den wechselseitigen Ausschluss sicherstellen.

- Da ein kritischer Bereich jederzeit von maximal einem Prozess betreten werden darf, kommt ein binärer Semaphor zum Einsatz.
 - Betritt ein Prozess einen kritischen Bereich, so sperrt er diesen mithilfe des Semaphors ab.
 - Ein so eingesetzter Semaphor wird manchmal als Sperre (lock), die P- und V-Operationen als Sperren (lock) und Freigeben (release) bezeichnet.



Kritischer Bereich bedeutet der Schutz von Daten und Wahrung der Konsistenz von Daten. Nicht der Programmcode muss geschützt werden, sondern die Daten die er verarbeitet.

Anwendung der Semaphoren

- Konkret beinhaltet die Absicherung eines kritischen Bereichs die folgenden Elemente:
 1. Den Semaphorzähler mit 1 initialisieren (eine Berechtigungsmarke bereitstellen, d.h. kritischer Bereich als frei markieren)
 2. Die P-Operation vor dem Eintritt in den kritischen Bereich ausführen (= reservieren bzw. Berechtigungsmarke beziehen)
 3. Die V-Operation beim Austritt aus dem kritischen Bereich ausführen (= freigeben bzw. Berechtigungsmarke an Semaphor zurückgeben)
 4. Das Betriebssystem blockiert den Prozess bei der P-Operation, wenn die Berechtigungsmarke bereits vergeben ist
 5. Das Betriebssystem weckt einen wartenden Prozess bei der nächsten V-Operation, die auf diesem Semaphor ausgeführt wird (da Berechtigungsmarke nun verfügbar ist)

Anwendung der Semaphoren

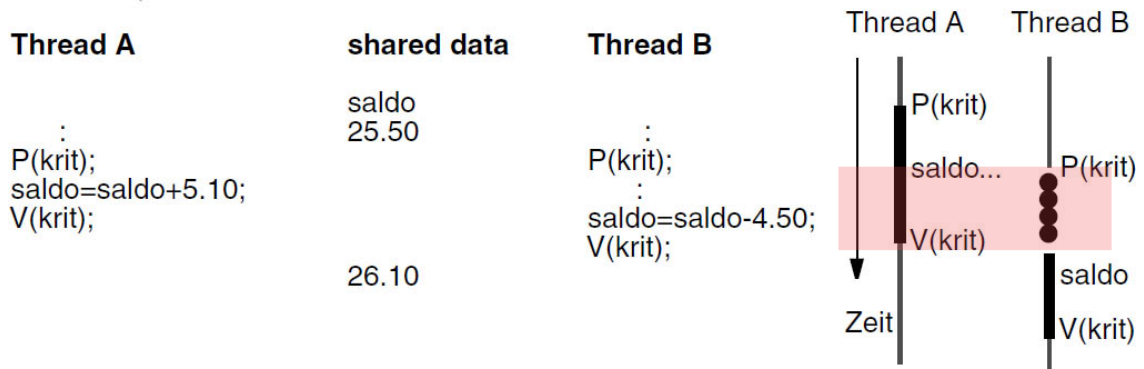


Abb. 12. Absicherung kritischer Bereiche (für Datenkonsistenz) mittels Semaphor (Beispiel)

Anwendung der Semaphoren

Eine weitere Anwendung ist Kooperation in Produzenten-Konsumenten Systemen mit einem Datenpuffer (Kapazität N Datensätze).

- Eine Menge von Prozessen P schreibt Daten in den Puffer (atomar)
- Ein oder mehrere Prozesse C lesen die Daten aus dem Puffer (atomar) für weitere Verarbeitung
- Es gibt zwei Situationen die mit Semaphoren abgesichert werden können:
 - Der Puffer ist leer \Rightarrow Semaphore Empty(0)
 - Der Puffer ist voll \Rightarrow Semaphore Full(N)
- Die Operationen (Write, Read) verändern dann die Semaphoren und können blockieren (; ist ein Zeitschritt):
 - Read() \Rightarrow P(Empty) ; V(Full)
 - Write() \Rightarrow P(Full) ; V(Empty)

Anwendung der Semaphoren

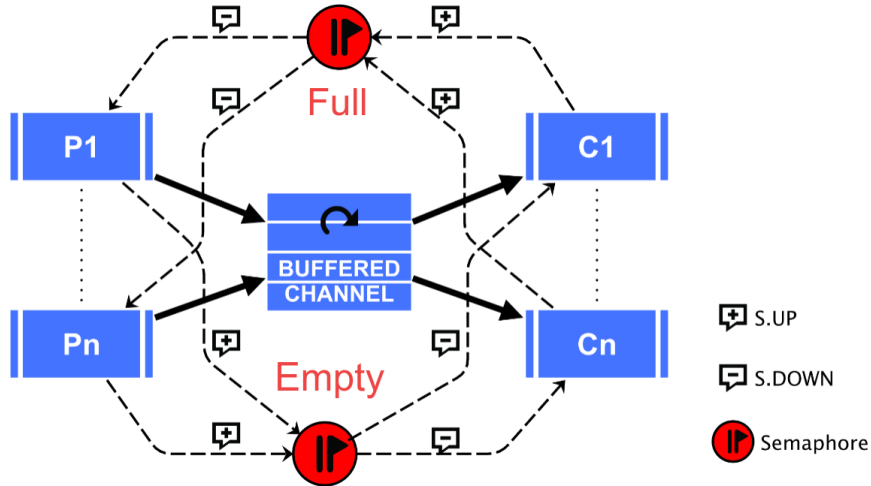
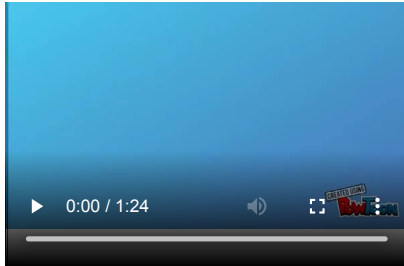


Abb. 13. Synchronisierte Datenwarteschlange mit Semaphoren für Produzenten-Konsumenten Systeme

Und es geht schief: Das Problem der fünf dinierenden Philosophen ist ein Beispiel für Deadlocks (Verklemmung) in verteilten und parallelen (asynchronen) Systemen mit Kommunikation!



- 5 Philosophen an einem Tisch
- 5 Gabeln, jeweils eine zwischen zwei Philosophen
- Ein Philosoph braucht zwei Gabeln zum Essen!
- Eine Gabel wird durch eine Semaphore repräsentiert (atomare Ressource, Startwert des Zählers ist 1)

Implementierung mit Kanälen



Bisher haben wir nur Kommunikationskanäle wie Sockets oder Pipes kennen gelernt. Mit denen kann unter bestimmten Bedingungen ein Semaphor implementiert werden.

- Wir nehmen an wir haben ein Prozesssystem aus $N+1$ Prozessen.
- Ein Prozess soll den Semaphor implementiert und überwachen.
- Die anderen N Prozess wollen sich über den Semaophor mittels Nachrichtenaustausch synchronisieren.
 - Dazu wird wie in der Übung ein Request und eine Acknowledge Kanäle (dort Pipe) verwendet.
 - je nachdem wie es mit der Atomarität von Nachrichten auf dem Kanal aussieht können Kanäle von Prozessen geteilt werden oder nicht
 - Also zwei Prozesse $P1$ und $P2$ schreiben eine zweistellige Nachricht $1+$ und $2-$ in den Requestkanal. Dann muss am anderen Ende entweder $1+2-$ oder $2-1+$ gelesen werden.
 - Beim Acknowledge Kanal (da N Leser) muss je einer pro Prozess existieren (es sei denn der Kanal ist wieder atomar in seinen Nachrichten und arbeitet im $1:N$ Multicasting, d.h. alle Leser bekommen alle Nachrichten).

Implementierung mit Kanälen

Protokoll: <ID>+ für V und <ID>- für P Operation und ID als Prozessnummer 1,2,3,...,N

```
channel req,ack1,ack2,... ; waiters = [] ; counter = INIT
while (!terminate) {
  received=read(req)
  messages=split(received,2) // 1+2- => [1+,2-]
  for(message of messages) {
    switch (message[1]) {
      case '+':
        if (length(waiters)) {
          next=head(waiters); waiters=tail(waiters)
          write(ack[next], '!')
        } else {
          counter++;
        }
        write(ack[Number(message[0]), '!') ; break
      case '-':
        if (counter>0) {
          counter--;
          write(ack[Number(message[0]), '!')
        } else {
          push(waiters, Number(message[0]))
        } ; break
    }
  }
}
```

Alg. 1. Der Semaphor Master Prozess

```
function P() {  
    write(req,String(myid)+'-')  
    reply=read(ack[myid])  
}  
function V() {  
    write(req,String(myid)+'+')  
    reply=read(ack[myid])  
}
```

Alg. 2. Der Semaphor Klienten Prozess

Zusammenfassung

- Neben impliziter Synchronisation wie z.B. beim Lesen aus Dateien bedarf es in Multiprozesssystemen explizite Synchronisation
- Ein Beispiel ist der Semaphor der:
 - mutualen Ausschluss (Wettbewerb)
 - Kooperation ermöglicht.
- Pipes sind i.A. inhärent synchron (geht aber auch asynchron)
- Ein Semaphore kann mit Kanälen und einem zentralen Masterprozess realisiert werden
- Prozessblockierung (also Prozesswechsel) ist wesentliches Merkmal von Synchronisation
- Es kann auch schief gehen wenn Prozesse mit mehreren Synchronisationsobjekten wie Semaphoren umgehen (Deadlock)