Betriebssysteme I

FH München – WS 2006/07

Hans-Georg Eßer

Zusammenfassung (2/2)

Gliederung

2. Prozesse und Threads
3. Interrupts
4. Scheduler
5. **Synchronisation**
6. **Interprozess-Kommunikation (IPC)**
7. **Deadlocks**

} **heute**

BS II: 8. Speicherverwaltung
9. Dateisysteme

5. Synchronisation

Kritische Abschnitte

- Programmteil, der auf gemeinsame Daten zugreift
- Block zwischen erstem und letztem Zugriff
- Nicht den Code schützen, sondern die Daten
- Formulierung: kritischen Bereich „betreten“ und „verlassen“
- Anforderung an parallele Threads:
  - maximal ein Thread gleichzeitig im kritischen Abschnitt
  - Kein Thread, der außerhalb kritischer Bereiche ist, darf einen anderen blockieren
  - Kein Thread soll ewig auf Betreten des kritischen Bereichs warten
  - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen krit. Bereichen und blockieren sich gegenseitig)

Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich ein, heißt das „**gegenseitiger Ausschluss**" (englisch: mutual exclusion, kurz: mutex)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmierfehlern

Programmtechnische Synchr.

Dekker: Kombination aus Lock-Variablen und wechselnder Reihenfolge

```
while (true) {
      C1=true;
      while (C2) {
        if (turn != 1) {
          C1=false;
          while (turn != 1) {
            /* wait */
          };
          C1=true;
        };
      };
      kritischer_bereich();
      turn=2;
      C1=false;
    }
```

```
while (true) {
  C2=true;
  while (C1) {
    if (turn != 2) {
      C2=false;
      while (turn != 2) {
        /* wait */
      };
      C2=true;
    };
  };
  kritischer_bereich();
  turn=1;
  C2=false;
}
```

Alternative:
Petersons Algorithmus

```
C1=true;
turn=2;
while (C2 && turn==2)
  /* warten */;
kritischer_abschnitt();
C1=false;
```

```
C2=true;
turn=1;
while (C1 && turn==1)
  /* warten */;
kritischer_abschnitt();
C2=false;
```


Hardware: Test and set lock (TSL)

- Maschineninstruktion (z.B. mit dem Namen TSL = Test and Set Lock), die atomar eine Lock-Variable liest und setzt, also ohne dazwischen unterbrochen werden zu können.

```
enter:
  tsl register, flag        ; Variablenwert in Register kopieren und
                            ; dann Variable auf 1 setzen
  cmp register, 0           ; War die Variable 0?
  jnz enter                 ; Nicht 0: Lock war gesetzt, also Schleife
  ret

leave:
  mov   flag, 0             ; 0 in flag speichern: Lock freigeben
  ret
```

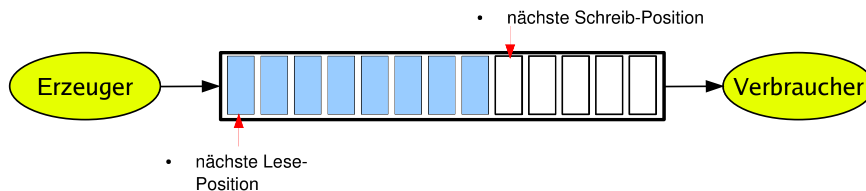
- TSL muss zwei Dinge leisten:
  - Interrupts ausschalten, damit der Test-und-Setzen-Vorgang nicht durch einen anderen Prozess unterbrochen wird
  - Im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU (deren Interrupts nicht gesperrt sind!) auf die gleiche Variable zugreifen kann

Aktives und passives Warten

- **Aktives Warten** (busy waiting):
  - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
  - Der Thread ist bereit und belegt die CPU.
  - Die Variable muss von einem anderen Thread gesetzt werden.
- **Passives Warten** (sleep and wake):
  - Ein Thread blockiert und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit" versetzt.
  - Der blockierte Thread verschwendet keine CPU-Zeit.
  - Ein anderer Thread muss das Eintreten des Ereignisses bewirken.
    - (Kleines) Problem, wenn der andere Thread endet.
  - Bei Eintreten des Ereignisses muss der blockierte Thread geweckt werden, z. B.
    - explizit durch einen anderen Thread,
    - durch Mechanismen des Betriebssystems.

Erzeuger-Verbraucher-Problem (1)

- Beim **Erzeuger-Verbraucher-Problem** (producer consumer problem, bounded buffer problem) gibt es zwei kooperierende Threads:
  - Der Erzeuger speichert Informationen in einem beschränkten Puffer.
  - Der Verbraucher liest Informationen aus diesem Puffer.



- **Synchronisation**
  - Puffer nicht überfüllen
  - Nicht aus leerem Puffer lesen

Erzeuger-Verbraucher-Problem (2)

- Realisierung mit **passivem Warten**:
  - Eine gemeinsam benutzte Variable „count“ zählt die belegten Positionen im Puffer.
  - Wenn der Erzeuger eine Information einstellt und der Puffer leer war (count == 0), weckt er den Verbraucher; bei vollem Puffer blockiert er.
  - Wenn der Verbraucher eine Information abholt und der Puffer voll war (count == max), weckt er den Erzeuger; bei leerem Puffer blockiert er.

Semaphore

Semaphor: Integer- (Zähler-) Variable, die man wie folgt verwendet:

- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
- Beim Anfordern eines Semaphors (P- oder **Wait**-Operation):
  - Semaphor-Wert um 1 erniedrigen, falls er positiv ist,
  - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.
- Bei Freigabe eines Semaphors (V- oder **Signal**-Operation):
  - einen Thread aus Warteschlange wecken, falls diese nicht leer ist,
  - Semaphor-Wert um 1 erhöhen (wenn kein wartender Thread da ist)
- Code sieht dann immer so aus:

```
wait (&sem);
/* Code, der die Ressource nutzt */
signal (&sem);
```

- Variante: *Negative* Semaphor-Werte

Mutexe

- **Mutex:** boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert (true: erlaubt; false: verboten)
- **blockierend:** Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert ➞ Warteschlange
- Bei Freigabe: Warteschlange enthält Threads ➞ einen wecken
Warteschlange leer: Mutex auf true setzen
- **Mutex** (**mut**ual **ex**clusion) = **binärer Semaphor**, also ein Semaphor, der nur die Werte 0 / 1 annehmen kann

```
wait (mutex) {
  if (mutex==1)
    mutex=0;
    else BLOCK_CALLER;
}

signal (mutex) {
  if (P in QUEUE(mutex)) {
    wakeup (P);
    remove (P, QUEUE);
  }
  else mutex=1;
}
```

- Neue Interpretation: wait ➞ lock, signal ➞ unlock
- Mutexe für exklusiven Zugriff (kritische Bereiche)

Blockieren?

- Betriebssysteme können Mutexe und Semaphoren **blockierend** oder **nicht-blockierend** implementieren
- blockierend: wenn der Versuch, den Zähler zu erniedrigen, scheitert
  ➡ warten
- nicht blockierend: wenn der Versuch scheitert
  ➡ vielleicht etwas anderes tun

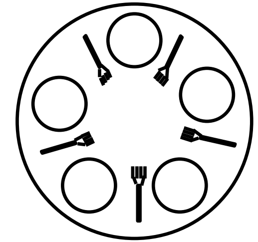
Atomare Operationen

- Bei Mutexen / Semaphoren müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:
  Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

Warteschlangen

- Mutexe / Semaphore verwalten Warteschlangen (der Prozesse, die schlafen gelegt wurden)
- Beim Aufruf von signal() muss evtl. ein Prozess geweckt werden
- Auswahl des zu weckenden Prozesses ist ein ähnliches Problem wie die Prozess-Auswahl im Scheduler
  - FIFO: **starker** Semaphor / Mutex
  - zufällig: **schwacher** Semaphor / Mutex

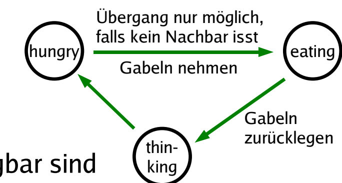
Philosophenproblem



- Fünf Philosophen an einem Tisch (Dijkstra 1965)
  - Jeder Philosoph hat vor sich einen Teller Spaghetti.
  - Zwischen je zwei Tellern liegt eine Gabel.
  - Jeder Philosoph wechselt ab zwischen Denken und Essen.
  - Zum Essen benötigt ein Philosoph die beiden Gabeln (bzw. Eßstäbchen) rechts und links von seinem Teller.
- Philosophen nicht verhungern lassen und maximale Parallelität

Lösung:

- Zustände der Philosophen in Array state[ ] speichern
- Semaphor sem[i] für jeden Philosophen: blockiert, wenn nicht beide Gabeln verfügbar sind



Monitore (1)

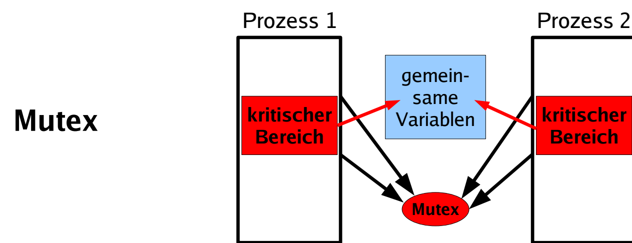
Motivation

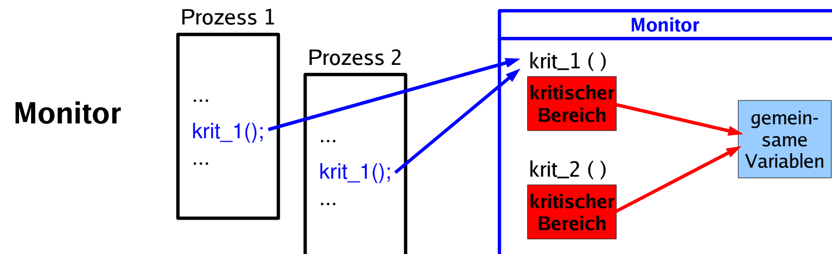
- Arbeit mit Semaphoren und Mutexen zwingt den Programmierer, vor und nach jedem kritischen Bereich wait() und signal() aufzurufen
- Wird dies ein einziges Mal vergessen, funktioniert die Synchronisation nicht mehr
- **Monitor** kapselt die kritischen Bereiche

Monitor: Sammlung von Prozeduren, Variablen, speziellen Bedingungsvariablen und Datenstrukturen:

- Prozesse können die Prozeduren des Monitors aufrufen, können aber nicht von außerhalb des Monitors auf dessen Datenstrukturen zugreifen.
- Zu jedem Zeitpunkt kann nur ein einziger Prozess aktiv im Monitor sein (d. h.: eine Monitor-Prozedur ausführen).
- Monitor wird durch Verlassen der Monitorprozedur frei gegeben

Monitore (2)





Monitore (3)

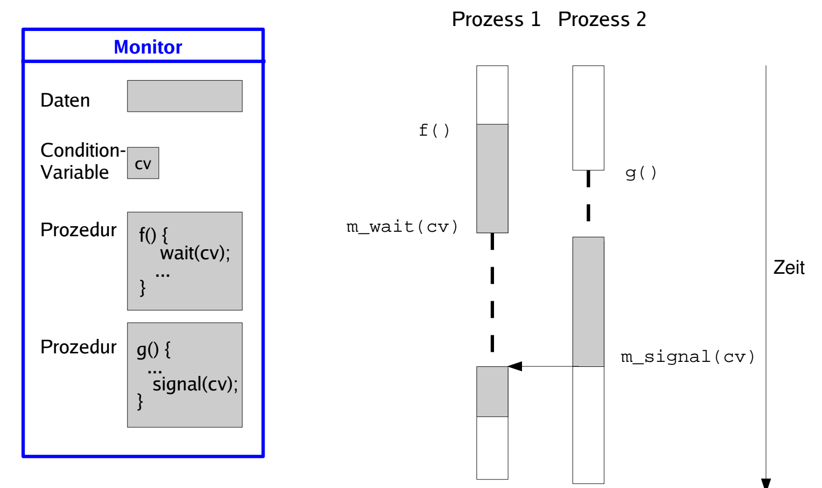
- Monitor-Konzept erinnert an Klassen oder Module
- Kapselung der Prozeduren und Variablen (außer über als public deklarierte Prozeduren kein Zugriff auf Monitor)
- Einfaches und übersichtliches Verfahren, um kritische Bereiche zu schützen, aber:
- Busy waiting → Schlafen/Wecken wäre besser

Zustandsvariablen (condition variables)

Für jede Zustandsvariable Wait- und Signal-Funktionen:

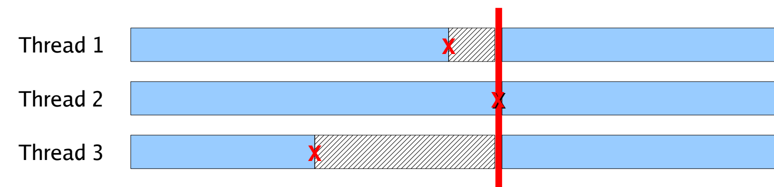
- m_wait (var): aufrufenden Prozess sperren (er gibt den Monitor frei)
- m_signal (var): gesperrten Prozess entsperren (weckt einen Prozess, der den Monitor mit m_wait() verlassen hat); erfolgt unmittelbar vor Verlassen des Monitors

Monitore (4)



Barrieren

- Idee: Komplexere Berechnung in mehrere Phasen unterteilen
- Vor Eintritt in eine neue Phase warten alle Threads, bis jeder einzelne die alte Phase abgeschlossen hat
- Dann kann z.B. ein Austausch der Zwischen-ergebnisse erfolgen
- Schließlich setzen die Threads (unabhängig) ihre Berechnungen fort – bis zur nächsten Barriere



- Threads rufen **barrier()**-Funktion auf und blockieren
- Erst wenn alle (Mitglieder einer Gruppe) **barrier()** aufgerufen haben, geht es weiter

Locking

Locking erweitert die Funktionalität von Mutexen, indem es verschiedene Lock-Modi (Zugriffsarten) unterscheidet, und deren „Verträglichkeit“ miteinander festlegt:

- Concurrent Read: Lesezugriff, andere Schreiber erlaubt.
- Concurrent Write: Schreibzugriff, andere Schreiber erlaubt.
- Protected Read: Lesezugriff, andere Leser erlaubt, aber keine Schreiber (share lock)
- Protected Write: Schreibzugriff, andere Leser erlaubt, aber kein weiterer Schreiber (update lock)
- Exclusive: Schreibzugriff, keine anderen Zugriffe erlaubt

Thread fordert Lock in bestimmtem Modus an.

- Ist der Lock-Modus mit den vorhandenen Locks anderer Threads verträglich, wird das Lock gewährt.
- Ist der Lock-Modus zu einem Lock eines anderen Threads unverträglich, blockiert der Thread, bis das Lock gewährt werden kann.

Nachrichten (2)

- **synchron** vs. **asynchron**
  - synchron: *send* / *receive* blockieren, bis zugehörige Operation auf Gegenseite abgeschlossen ist
  - asynchron: *send*-Call kehrt sofort zurück; Erfolg des Versands ist evtl. überprüfbar, z. B.:
    - Gegenseite schickt explizit Antwort
    - Messaging-System sendet Signal bei Zustellung
- **verbindungsorientiert** vs. **verbindungslos**
  - verbindungsorientiert: „stehende Verbindung“ (TCP)
  - verbindungslos (vgl. UDP)

Nachrichten (1)

- Nachrichtenaustausch über zwei Systemaufrufe
  - `send (destination, &message);`
  - `receive (source, &message);`
- Synchrone Kommunikation: Threads blockieren, wenn *send* bzw. *receive* nicht sofort ausgeführt werden können
- Vorteil: funktioniert auch bei Systemen ohne gemeinsamen Hauptspeicher (distributed systems, client-server-computing)
- Nachteile:
  - aufwändiger durch Duplizieren der Daten
  - Verwaltung der Namen für Quelle und Ziel nötig
  - Vorkehrungen gegen Verlust der Meldung nötig
- Implementierung z. B. durch Pipes oder Mailslots (Windows) oder RPCs.

Linux: Synchr. von Threads (1)

POSIX-Mutexe und Semaphore

pthread_mutex_init	sem_init
pthread_mutex_lock	sem_wait, sem_trywait
pthread_mutex_unlock	sem_post
pthread_mutex_destroy	sem_destroy
	sem_getvalue

POSIX-Bedingungsvariablen

Threads warten darauf, dass eine bestimmte Bedingung erfüllt ist, und schlafen solange (vgl. *Monitore*)

pthread_cond_init

pthread_cond_wait

pthread_cond_signal & pthread_cond_broadcast

Linux: Synchr. von Threads (2)

	Mutexe	Semaphore	Bedingungsvariablen
Warten bzw. Blockieren	pthread_mutex_lock, pthread_mutex_trylock	sem_wait, sem_trywait	pthread_cond_wait
Signalisieren	pthread_mutex_unlock	sem_post	pthread_cond_signal, pthread_cond_broadcast
Erzeugen	pthread_mutex_init	sem_init	pthread_cond_init
Zerstören	pthread_mutex_destroy	sem_destroy	pthread_cond_destroy

Synchronisation im Linux-Kernel

- **Atomare Operationen**
  - auf Integer-Variablen: atomic_set, atomic_add/sub, atomic_inc/dec, atomic_read, atomic_sub/dec/inc_and_test,
  - Bit-Operationen auf Bitvektoren:set_bit, clear_bit, change_bit, test_and_set/clear/change_bit, test_bit, find_first_(zero_)bit
- **Spin Locks / Reader-Writer Spin Locks**
- **Semaphore / Reader-Writer-Semaphore**
- **„Big Kernel Lock“**

Linux: Synchr. von Prozessen

- **Benannte POSIX-Semaphore**
  - `posix_sem = sem_open("/MeinSemaphor", ...);`
  - `sem_init(posix_sem, 0, INIT_WERT);`
  - `sem_post(posix_sem);   /* signal */`
  - `sem_wait(posix_sem);   /* wait   */`
- **Mutex: als binärer Semaphor**
- **System-V-IPC-Semaphore**
  - Semaphor-Set (kann mehrere Semaphore enthalten)
  - private / public
  - semget () erzeugt Semaphor
  - semctl (): enthält wait- und signal-Operationen

Spin Locks (1)

- Lock mit Mutex-Funktion: Gegenseitiger Ausschluss
- Code, der ein Spin Lock anfordert und nicht erhält, läuft in Schleife weiter, bis das Lock verfügbar wird („spinning“)
- Typ: *spinlock_t*

```
spinlock_t xy_lock = SPIN_LOCK_UNLOCKED

spin_lock (&xy_lock);
/* kritischer Abschnitt */
spin_unlock (&xy_lock);
```

- *Da Spin Locks nicht schlafen, kann man sie in Interrupt-Handlern verwenden*
- In dem Fall: zusätzlich Interrupts sperren:

```
unsigned long flags;
spin_lock_irqsave (&xy_lock, flags);
/* kritischer Abschnitt */
spin_unlock_irqrestore (&xy_lock, flags);
```

(Interrupts sichern, dann sperren; urspr. Zustand wiederherstellen)

Spin Locks (2)

- Wenn zu Beginn alle Interrupts aktiviert sind, geht es auch einfacher:

```
spin_lock_irq (&xy_lock);
/* kritischer Abschnitt */
spin_unlock_irq (&xy_lock);
```

  schaltet alle Interrupts aus bzw. wieder an
- Spin Locks sind nicht „rekursiv“,
  d.h.: es ist nicht möglich, das gleiche Spin Lock zweimal nacheinander anzufordern, etwa beim rekursiven Aufruf einer Funktion
- Alternative zu Spin Locks: Reader Writer Locks

Kernel-Semaphore (1)

- Kernel-Semaphore sind „schlafende“ Locks
- Ist ein Semaphor schon gelockt, werden weitere Interessenten in eine Warteschlange eingereiht.
- Bei Freigabe eines Semaphors wird der erste wartende Thread in der Warteschlange geweckt
- Semaphore eignen sich für Sperren, die über einen längeren Zeitraum gehalten werden: keine Verschwendung von Rechenzeit
- Semaphore sind nur im Prozess-Kontext einsetzbar, nicht in Interrupt-Handlern (Interrupt-Handler werden nicht vom Scheduler behandelt)
- Code, der einen Semaphor verwenden will, darf nicht bereits ein normales Lock besitzen (Semaphor-Zugriff kann dazu führen, dass der Thread sich schlafen legt.)
- Semaphore können auch mehr als einen Thread auf die Ressource zugreifen lassen

Kernel-Semaphore (2)

- Verwendung mit *up()* und *down()*

```
down (&sem);
/* kritischer Abschnitt */
up (&sem);
```

- Varianten von *down()*
  - *down (&sem);*
    nicht unterbrechbarer Schlaf, falls Semaphor nicht verfügbar
  - *down_interruptible (&sem);*
    unterbrechbarer Schlaf, falls Sem. nicht verfügbar
  - *down_trylock (&sem);*
    versucht, den Semaphor zu erhalten – falls das nicht gelingt, kehrt die Funktion sofort mit False-Wert zurück
- Reader Writer Semaphore: sind Mutexe (nur 0/1)

„Big Kernel Lock“ (BKL)

- Überbleibsel aus älteren Kernel-Versionen
- Globales Lock für den gesamten Kernel (das alle Code-Teile betrifft, die damit Datenzugriff schützen)
- lock_kernel(), unlock_kernel()
- BKL nur im Prozess-Kontext benutzbar (nicht in Interrupt-Routinen)
- Prozess, der das BKL hält, darf schlafen
  - Beim Schlafenlegen wird das BKL automatisch aufgegeben
  - Beim Aufwecken wird es wieder erworben
- BKL ist rekursiv: Prozess, der bereits das BKL hält, darf also erneut *lock_kernel()* ausführen
- <u>Nicht benutzen!</u>

6. Inter Process Communication

IPC-Charakterisierung

Kommunikationsmodell:

Punkt-zu-Punkt

Publish-Subscribe

Broadcast

Übertragungsrichtung:

simplex / unidirektional

duplex / bidirektional

Synchronität

synchron / blockierend

asynchron / nicht-blockierend

Art der Nachricht

Nachrichten- oder Stream-orientiert

Plattform(-un-)abhängigkeit

Portierbarkeit

Lokalität

systemgebunden oder Netzwerkkommunikation möglich (über Rechner-grenzen hinweg)

Sockets (1)

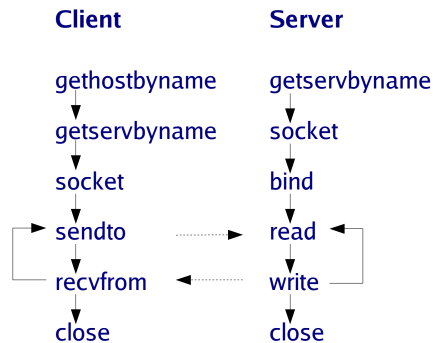
- Sockets stellen einen Kommunikationskanal mit folgenden Eigenschaften bereit:
  - sie erlauben IPC zwischen Prozessen
  - lokal oder rechnerübergreifend
  - über verschiedene Protokolle
  - plattformübergreifend
- bidirektionale Kommunikation
- **Adressierung:** lokal (AF_UNIX; Pfadname) oder mit Netzwerkadresse (AF_INET; host + port)
- **Zuverlässigkeit:**
  - reliable, verbindungsorientiert (z.B. **TCP**-Protokoll)
  - unreliable, verbindungslos (z.B. **UDP**-Protokoll / Datagram)

Sockets (2)

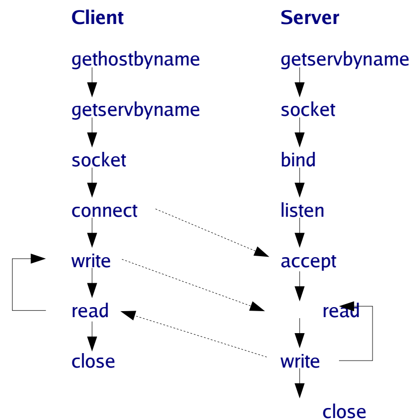
- **Kommunikationsarten:**
  - Nachrichten-orientiert: recvmsg() oder sendmsg()
  - Stream-orientiert: sendto(), recfrom(), read(), write()
  - Senden blockiert bei umfangreicheren Daten, bis Gegenstelle gelesen hat.
  - optional nicht-blockierendes Verhalten wählbar
  - Linux: Zugriff über Dateideskriptoren

UDP / TCP Sockets

Verbindungslos:
Datagramme / UDP

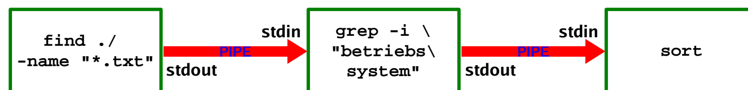


Verbindungsorientiert:
Streams / TCP



Pipes

```
find ./ -name "*.txt" | grep -i "betriebssystem" | sort
```



- Die Pipe ist ein **unidirektionaler** Kommunikationsmechanismus
- Es gibt benannte und unbenannte Pipes

Unbenannte Pipes

Prozess erzeugt Pipe (Anfangs- und Endstück): **pipe(fds)**

Prozess verdoppelt sich mit **fork()**

Vater und Sohn verwenden die beiden Pipe-Enden: **read()**, **write()**

Pipe bleibt offen, bis einer der Prozesse sie mit **close()** schließt

Prozesse, die nicht verwandt sind (fork), brauchen benannte Pipes

In Pipe schreiben und lesen wie in Datei

Benannte Pipes

- Mit benannten Pipes ist auch Kommunikation zwischen „nicht verwandten" Prozessen möglich:
- Prozesse legen gemeinsamen Namen fest (vgl. Semaphore etc.)

```
int mkfifo(const char *pathname, mode_t mode);
```

das erzeugt eine neue FIFO-Spezialdatei im Dateisystem

- Prozesse öffnen Pipe wie eine normale Datei
- anschließend lesen und schreiben die beiden Prozesse ebenfalls wie bei Dateien, open(PIPENAME, ...)
- Kommunikation in beide Richtungen: zwei Pipes verwenden

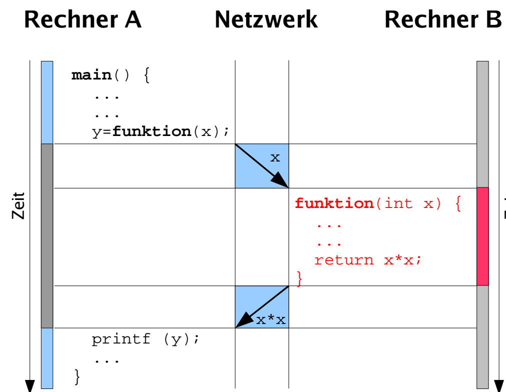
Remote Procedure Calls (1)

- Funktionsaufrufe, die Prozess- / Rechner-übergreifend sind
- Programmierer muss sich über die nötige Kommunikation keine Gedanken machen
  - Message Passing, I/O etc. vor Programmierer verbergen
- Funktionsargumente und -ergebnisse mit einem RPC-Protokoll zwischen den beteiligten Prozessen hin und her übertragen
- Client- / Server-Prinzip
- Prozess, der Funktion aufruft, enthält **client stub**
  - Funktionsaufruf (über das Netzwerk) an anderen Prozess weiterleiten, Antwort empfangen und Ergebnis zurückgeben
- Prozess, der Funktion ausführt, enthält **server stub**
  - Argumente von einem client stub entgegennehmen
  - Funktion ausführen, Ergebnis zurück senden

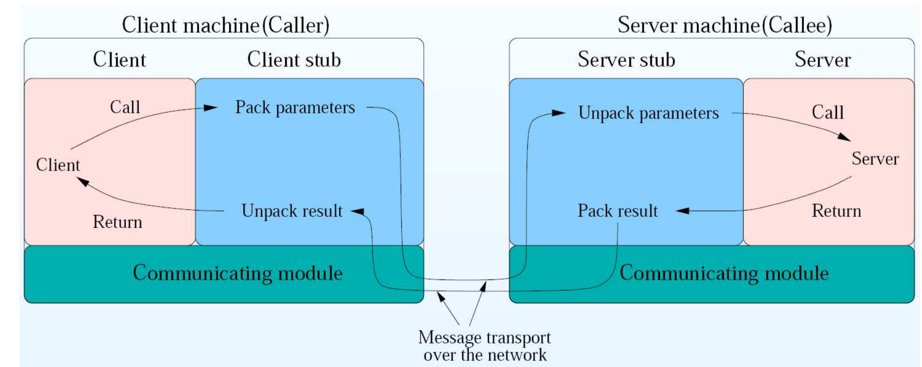
Remote Procedure Calls (2)

Remote-Funktionsaufruf (anschaulich):

- Programm führt *main*()-Funktion aus.
- Vor dem Start der Funktion wird das Argument (über das Netzwerk) an den Prozess übermittelt, der die Funktion ausführt.
- Dort wird das Ergebnis berechnet und zurück übertragen.
- Während der Berechnung und den beiden Übertragungen ist der aufrufende Prozess untätig (synchroner Auftrag).



Remote Procedure Calls (4)



Quelle: http://www-wjp.cs.uni-sb.de/lehre/seminar/ss04/reports/A_Shadrin-RPC-folien.pdf

Remote Procedure Calls (3)

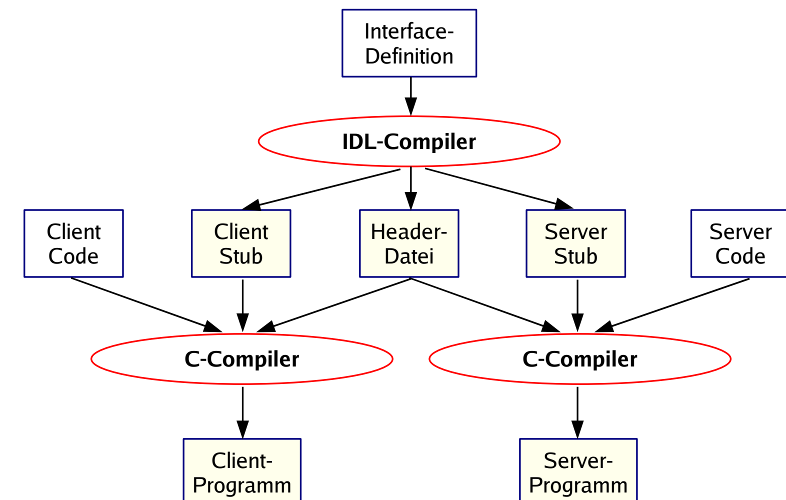
Unterschiede zu normalen Funktionen:

- kein **call by reference** möglich
- client stub kann aber calls by reference annehmen und dann die Daten für den Transport vorbereiten
- keine **Nebeneffekte** möglich

Marshalling / Serialisierung

- Problem: Client- und Server-Rechner verwenden evtl. unterschiedliche interne Repräsentationen für Datentypen
  - klassisches (und einfachstes) Beispiel: Byte order
  - wie packt man Strings, Floats, komlexere Strukturen (abstrakte Datentypen), Objekte etc. ein?
- pack / unpack

RPC-Programmierung



NFS

- NFS: Network File System
- Sun hat RPC („Sun RPC") für NFS entwickelt
- NFS-Prozeduren: mkdir, rmdir, readdir, create, remove, read, write, getattr, setattr, link, symlink, readlink, statfs
- Beobachten, wie Nachrichten übertragen werden:

```
tcpdump -s 1024 host 192.168.1.2 | grep nfs > /tmp/tcpdump.log
```

- Dann von NFS-Client aus Verzeichnis erzeugen, Datei anlegen, umbenennen, löschen
- NFS-Client führt via RPC die NFS-Prozeduren mkdir, create, write, read, remove aus

LPC: Local Procedure Call

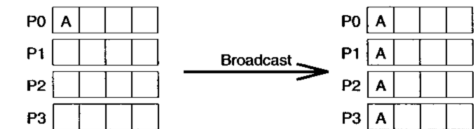
- Variante von RPC, bei der Client und Server auf demselben Rechner laufen
- Einige „Einsparungen" möglich:
  - kein Marshalling nötig (Sender und Empfänger verwenden gleiche Datenrepräsentation)
  - *call by reference* evtl. möglich, falls Sender- und Empfänger-Prozess gemeinsamen Speicher nutzen
  - Besondere RPC-Fehler (etwa: Funktionsergebnis „Server abgestürzt") können nicht auftreten
- Microsoft LPC: Windows wechselt automatisch auf ein einfacheres Protokoll, wenn es eine lokale Verbindung erkennt

MPI-Features

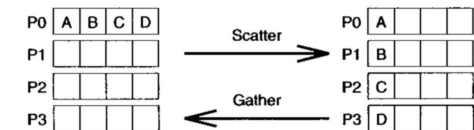
- Verschiedene Arten von Point-to-Point Message Passing
  - blockierend (*e.g.* MPI_SEND)
  - nicht-blockierend (*e.g.* MPI_ISEND)
  - synchron (*e.g.* MPI_SSEND)
  - gepuffert (*e.g.* MPI_BSEND)
- „kollektive" Kommunikation (alle Prozesse führen einen MPI-Befehl gemeinsam aus), z. B. MPI_REDUCE
- Synchronisation, z. B. MPI_BARRIER
- benutzerdefinierte Datentypen
- unterstützt Topologien (z. B. 3x4-Grid), dadurch entstehen „Nachbarschafts"-Konzepte"

MPI-Übersicht

Broadcast:
`MPI_BCast ()`

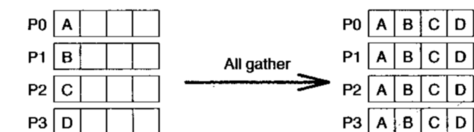


Scatter:
`MPI_Scatter ()`



Gather:
`MPI_Gather ()`

All gather:
`MPI_Allgather ()`



All to all:
`MPI_Alltoall`

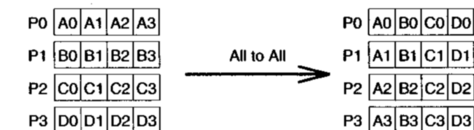
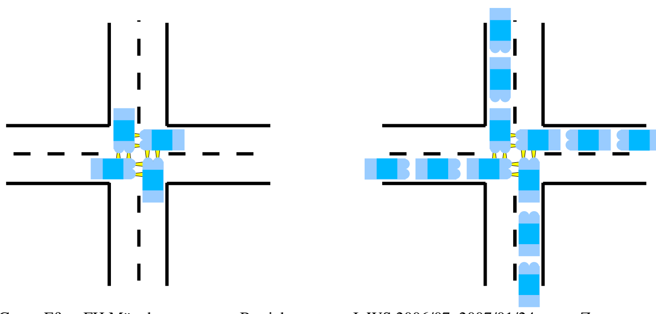


Bild: Gropp/Lusk/Skjellum: „Using MPI", 1994

7. Deadlocks

Was ist ein Deadlock?

- **Deadlock-Situation:**
  - jeder Prozess wartet auf eine Ressource, die von einem anderen Prozess blockiert wird
  - keine der Ressourcen kann freigegeben werden, weil die besitzenden Prozesse (die selbst warten) blockiert sind
- In einer Deadlock-Situation verharren die Prozesse dauerhaft -> Deadlocks sind unbedingt zu vermeiden



Deadlock: kleinstes Beispiel

Prozess P

```
lock (A);
lock (B);
/* krit. Ber. */
unlock (A);
unlock (B);
```

Prozess Q

```
lock (B);
lock (A);
/* krit. Ber. */
unlock (B);
unlock (A);
```

Problematische Reihenfolge:

P: lock(A)
Q: lock(B)
P: lock(B) <- blockiert
Q: lock(A) <- blockiert

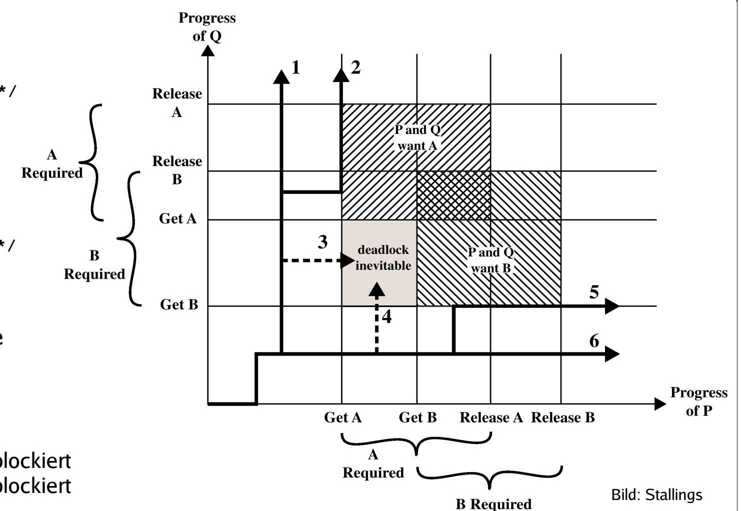


Bild: Stallings

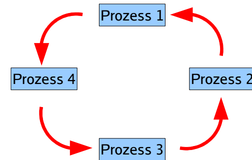
Deadlock-Bedingungen (1)

1. **Gegenseitiger Ausschluss (mutual exclusion)**
   Ressource ist exklusiv
2. **Hold and Wait (besitzen und warten)**
   Ein Prozess ist bereits im Besitz einer oder mehrerer Ressourcen, und er kann noch weitere anfordern
3. **Ununterbrechbarkeit der Ressourcen**
   Ressource kann nicht durch das Betriebssystem entzogen werden

- (1) bis (3) sind **notwendige** Bedingungen für einen Deadlock
- (1) bis (3) sind aber auch „wünschenswerte“ Eigenschaften eines Betriebssystems, denn:
  - gegenseitiger Ausschluss ist nötig für korrekte Synchronisation
  - Hold & Wait ist nötig, wenn Prozesse exklusiven Zugriff auf mehrere Ressourcen benötigen
  - Bei manchen Betriebsmitteln ist eine Präemption prinzipiell nicht sinnvoll (z. B. DVD-Brenner, Streamer)

Deadlock-Bedingungen (2)

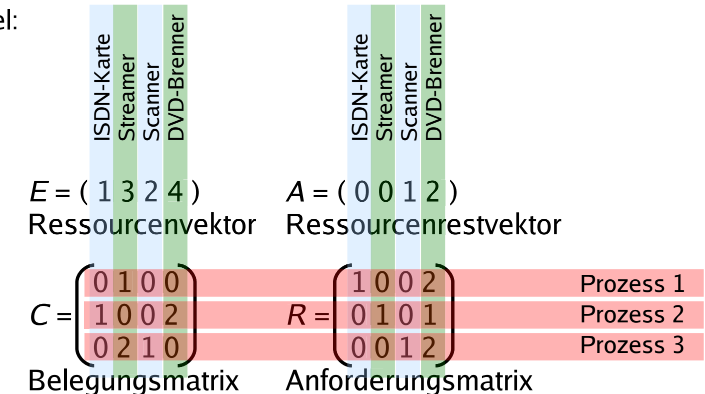
4. **Zyklisches Warten**
   Man kann die Prozesse in einem Kreis an-ordnen, in dem jeder Prozess eine Ressource benötigt, die der folgende Prozess belegt hat



- (1) bis (4) sind **notwendige und hinreichende** Bedingungen für einen Deadlock
- Das zyklische Warten (4) (und dessen Unauf-lösbarkeit) sind Konsequenzen aus (1) bis (3)
- (4) ist der erfolgversprechendste Ansatzpunkt, um Deadlocks aus dem Weg zu gehen

Deadlock-Erkennung (1)

- Idee: Deadlocks zunächst zulassen
- System regelmäßig auf Vorhandensein von Deadlocks überprüfen und diese dann abstellen
- Beispiel:



Ressourcen-Zuordnungs-Graph (1)

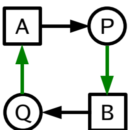
- Belegung und (noch unerfüllte) Anforderung grafisch darstellen:

R Ressource   P Prozess

R → P   P hat R belegt

R ← P   P hat R angefordert

- P, Q aus Minimalbeispiel



- Deadlock = Kreis im Graph

- Variante für Ressourcen, die mehrfach vorkommen können

R   Ressource (mit zwei Instanzen)

P   Prozess

R → P   P hat (ein) R belegt

R ← P   P hat (irgendein) R angefordert

- Kreis nur notwendig für Deadlock

Deadlock-Erkennung (2)

Algorithmus

1. Suche einen unmarkierten Prozess P_i, dessen verbleibende Anforderungen vollständig erfüllbar sind, also $R_{ij} \leq A_j$ für alle j
2. Gibt es keinen solchen Prozess, beende den Algorithmus
3. Ein solcher Prozess könnte erfolgreich abgearbeitet werden. Simuliere die Rückgabe aller belegten Ressourcen: $A := A + C_i$ (i-te Zeile von C)
   Markiere den Prozess – er ist nicht Teil eines Deadlocks
4. Weiter mit Schritt 1

Deadlock-Behebung

Entziehen einer Ressource?
In den Fällen, die wir betrachten, unmöglich (ununterbrechbare Ressourcen)

Abbruch eines Prozesses, der am Deadlock beteiligt ist

Rücksetzen eines Prozesses in einen früheren Prozesszustand, zu dem die Ressource noch nicht gehalten wurde (erfordert regelmäßiges Sichern der Prozesszustände)

Deadlock-Vermeidung (1)

Deadlock Avoidance (Vermeidung)

- **Idee:** BS erfüllt Ressourcenanforderung nur dann, wenn dadurch auf keinen Fall ein Deadlock entstehen kann
- Das funktioniert nur, wenn man die **Maximalforderungen aller Prozesse** kennt
  - Prozesse registrieren **beim Start** für alle denkbaren Ressourcen ihren Maximalbedarf
  - für die Praxis i. d. R. irrelevant, nur in wenigen Spezialfällen nützlich

Sichere vs. unsichere Zustände

- Ein Zustand heißt **sicher**, wenn es eine Ausführreihenfolge der Prozesse gibt, die auch dann keinen Deadlock verursacht, wenn alle Prozesse sofort ihre maximalen Ressourcen-forderungen stellen.
- Ein Zustand heißt **unsicher**, wenn er nicht sicher ist.
- Unsicher bedeutet nicht zwangsläufig Deadlock!

Deadlock-Vermeidung (2)

Banker-Algorithmus

- Datenstrukturen wie bei Deadlock-Erkennung:
  - n Prozesse $P_1 \dots P_n$, m Ressourcentypen $R_1 \dots R_m$ mit je E_i Ressourcen-Instanzen (i=1,...,m) -> **Ressourcenvektor** $E = (E_1 \dots E_m)$
  - **Ressourcenrestvektor** A (wie viele sind noch frei?)
  - **Belegungsmatrix** C
    C_{ij} = Anzahl Ressourcen vom Typ j, die Prozess i belegt
  - **Maximalbelegung** *Max*:
    Max_{ij} = max. Bedarf, den Prozess i an Ressource j hat
  - **Maximale zukünftige Anforderungen:** $R = Max - C$,
    R_{ij} = # Ress. vom Typ j, die Prozess i noch maximal anfordern kann

Feststellen, ob ein Zustand sicher ist = Annehmen, dass alle Prozesse sofort ihre Maximalforderungen stellen, und dies auf Deadlocks überprüfen

Deadlock-Verhinderung (1)

- **Deadlock-Verhinderung (prevention):**
  Vorbeugendes Verhindern
  - mache mindestens eine der vier Deadlock-Bedingungen unerfüllbar (gegenseitiger Ausschluss, Hold and Wait, Ununterbrechbarkeit der Ressourcen, Zyklisches Warten)
  - dann sind keine Deadlocks mehr möglich (denn die vier Bedingungen sind notwendig)

Deadlock-Verhinderung (2)

1. **Gegenseitiger Ausschluss**
   - Ressourcen nur dann exklusiv Prozessen zuteilen, wenn es keine Alternative dazu gibt
   - Beispiel: Statt mehrerer konkurrierender Prozesse, die einen gemeinsamen Drucker verwenden wollen, eine Drucker-Spooler einführen
     - keine Konflikte mehr bei Zugriff auf Drucker (Spooler-Prozess ist der einzige, der direkten Zugriff erhalten kann)
     - aber: Problem evtl. nur verschoben (Größe des Spool-Bereichs bei vielen Druckjobs begrenzt?)

Deadlock-Verhinderung (3)

2. Hold and Wait

- Alle Prozesse müssen die benötigten Ressourcen gleich beim Prozessstart anfordern (und blockieren)
- hat verschiedene Nachteile:
  - Ressourcen-Bedarf entsteht oft dynamisch (ist also beim Start des Prozesses nicht bekannt)
  - verschlechtert Parallelität (Prozess hält Ressourcen über einen längeren Zeitraum)

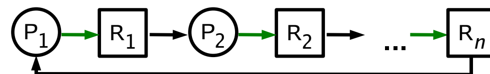
3. Ununterbrechbarkeit der Ressourcen

- Ressourcen entziehen?
- siehe Deadlock-Behebung (Abbruch / Rücksetzen)

Deadlock-Verhinderung (5)

4. Zyklisches Warten (1)

- Ressourcen durchnumerieren
  - ord: $\mathbf{R} = \{R_1, ..., R_n\} \to \mathbb{N}$, $ord\,(R_i) \neq ord\,(R_j)$ für $i \neq j$
- Prozess darf Ressourcen nur in der durch ord vorgegebenen Reihenfolge anfordern
- Das macht Deadlocks unmöglich. Annahme: Es gibt einen Zykel



  Für jedes i gilt: $ord\,(R_i) < ord\,(R_{i+1})$ und wegen des Zykels auch $ord\,(R_n) < ord\,(R_1)$, daraus folgt $ord\,(R_1) < ord\,(R_1)$: Widerspruch
- Problem: Gibt es eine feste Reihenfolge der Ressourcenbelegung, die für alle Prozesse geeignet ist?
- reduziert Parallelität (Ressourcen zu früh belegt)


```
# shutdown -h now "Vorlesung vorbei"
```

- Wir sehen uns zur Prüfung
- ... und bei der Klausureinsicht
- Viel Erfolg beim Lernen!

- **Nicht vergessen:**
  Fragen bis zum Klausurtermin jederzeit
  - per Mail: h.g.esser@gmx.de
  - Antworten kommen per Mail
  - und landen im Prüfungs-Blog: http://fhm.hgesser.de/blog