9. Foliensatz Betriebssysteme

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences (1971-2014: Fachhochschule Frankfurt am Main) Fachbereich Informatik und Ingenieurwissenschaften christianbaun@fb2.fra-uas.de

Lernziele dieses Foliensatzes

- Am Ende dieses Foliensatzes kennen/verstehen Sie. . .
 - was kritische Abschnitte und Wettlaufsituationen sind
 - was Synchronisation ist
 - wie Signalisierung die Ausführungsreihenfolge der Prozesse beeinflusst
 - wie mit Blockieren kritische Abschnitte gesichert werden
 - welche Probleme (Verhungern und Deadlocks) beim Blockieren entstehen können
 - wie Deadlock-Erkennung mit Matrizen funktioniert
 - verschiedene Möglichkeiten der Kommunikation zwischen Prozessen:
 - Gemeinsamer Speicher (Shared Memory)
 - Nachrichtenwarteschlangen (Message Queues)
 - Pipes
 - Sockets
 - verschiedene Möglichkeiten der Kooperation von Prozessen
 - wie **Semaphore** kritische Abschnitte sichern können
 - den Unterschied zwischen Semaphor und Mutex

Interprozesskommunikation (IPC)

- Prozesse müssen nicht nur Lese- und Schreibzugriffe auf Daten ausführen, sondern auch:
 - sich gegenseitig aufrufen
 - aufeinander warten
 - sich abstimmen
 - kurz gesagt: Sie müssen miteinander interagieren
- Bei Interprozesskommunikation (IPC) ist zu klären:
 - Wie kann ein Prozess Informationen an andere weiterreichen?
 - Wie können mehrere Prozesse auf gemeinsame Ressourcen zugreifen?

Frage: Wie verhält es sich hier mit Threads?

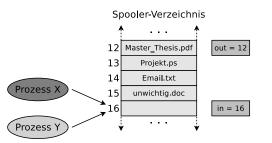
- Bei Threads gelten die gleichen Herausforderungen und Lösungen wie bei Interprozesskommunikation mit Prozessen
- Nur die Kommunikation zwischen den Threads eines Prozesses ist problemlos möglich, weil sie im gleichen Adressraum agieren

Kritische Abschnitte

- Laufen mehrere parallel ausgeführte Prozesse, unterscheidet man:
 - Unkritische Abschnitte: Die Prozesse greifen gar nicht oder nur lesend auf gemeinsame Daten zu
 - Kritische Abschnitte: Die Prozesse greifen lesend und schreibend auf gemeinsame Daten zu
 - Kritische Abschnitte dürfen nicht von mehreren Prozessen gleichzeitig durchlaufen werden
- Damit Prozesse auf gemeinsam genutzten Speicher (Daten)
 zugreifen können, ist wechselseitiger Ausschluss (Mutual Exclusion)
 nötig

Kritische Abschnitte – Beispiel: Drucker-Spooler

Prozess X Prozess Y next free slot = in: (16) Prozesswechsel next free slot = in: (16) Speichere Eintrag in next_free_slot; (16) in = next_free_slot + 1; (17) Prozesswechsel Speichere Eintrag in next_free_slot; (16)



0000000

in = next_free_slot + 1; (17)

- Das Spooler-Verzeichnis ist konsistent
 - Aber der Eintrag von Prozess Y wurde von Prozess X überschrieben und ging verloren
- Eine solche Situation heißt. Race Condition

Race Condition (Wettlaufsituation)

- Unbeabsichtigte Wettlaufsituation zweier Prozesse, die den Wert der gleichen Speicherstelle ändern wollen
 - Das Ergebnis eines Prozesses hängt von der Reihenfolge oder dem zeitlichen Ablauf anderer Ereignisse ab
 - Häufiger Grund für schwer auffindbare Programmfehler
- Problem: Das Auftreten und die Symptome hängen von unterschiedlichen Ereignissen ab
 - Bei jedem Testdurchlauf können die Symptome unterschiedlich sein oder verschwinden
- Vermeidung ist u.a durch das Konzept der Semaphore (⇒ Folie 64) möglich

Therac-25: Race Condition mit tragischem Ausgang (1/2)

- Therac-25 ist ein
 Elektronen-Linearbeschleuniger
 zur Strahlentherapie von
 Krebstumoren
- Verursachte Mitte der 80er Jahre in den USA Unfälle durch mangelhafte Programmierung und Qualitätssicherung
 - Einige Patienten erhielten eine bis zu hundertfach erhöhte Strahlendosis

Bildauelle: Google Bildersuche



Therac-25: Race Condition mit tragischem Ausgang (2/2)

An Investigation of the Therac-25 Accidents. Nancy Leveson, Clark S. Turner IEEE Computer, Vol. 26, No. 7, July 1993, S.18-41 http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html

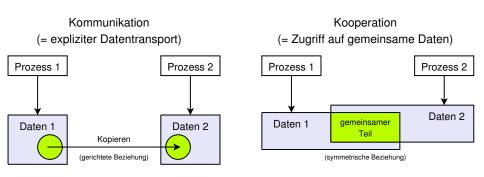
- 3 Patienten starben wegen Programmfehlern
- 2 Patienten starben durch eine Race Condition, die zu inkonsistenten Einstellungen des Gerätes und damit zu erhöhter Strahlendosis führte
 - Der Kontroll-Prozess synchronisierte nicht korrekt mit dem Prozess der Eingabeaufforderung
 - Der Fehler trat nur dann auf, wenn die Bedienung zu schnell erfolgte
 - Bei Tests trat der Fehler nicht auf, weil es Erfahrung (Routine) erforderte, um das Gerät so schnell zu bedienen





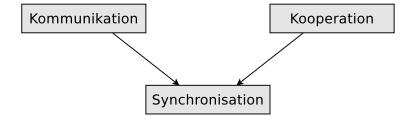
Kommunikation vs. Kooperation

- Die Prozessinteraktion besitzt 2 Aspekte:
 - Funktionaler Aspekt: Kommunikation und Kooperation
 - Zeitlicher Aspekt: Synchronisation



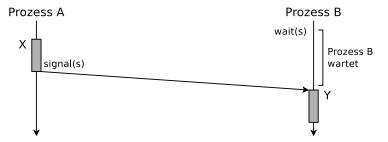
Interaktionsformen

- Kommunikation und Kooperation basieren auf Synchronisation
 - Synchronisation ist die elementarste Form der Interaktion
 - Grund: Kommunikation und Kooperation benötigen eine zeitliche Abstimmung zwischen den Intaraktionspartnern, um korrekte Ergebnisse zu erhalten
 - Darum behandeln wir zuerst die Synchronisation

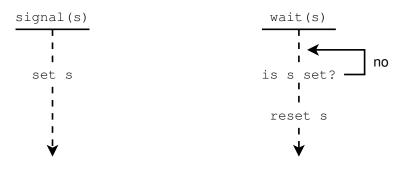


Signalisierung

- Eine Möglichkeit um Prozesse zu synchronisieren
- Mit Signalisierung wird eine Ausführungsreihenfolge festgelegt
- Beispiel: Abschnitt **X** von Prozess P_A soll **vor** Abschnitt **Y** von Prozess P_B ausgeführt werden
 - Die Operation signal signalisiert, wenn Prozess P_A den Abschnitt X abgearbeitet hat
 - Prozess P_B muss eventuell auf das Signal von Prozess P_A warten



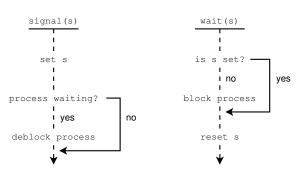
Einfachste Form der Signalisierung (aktives Warten)



- Die Abbildung zeigt aktives Warten an der Signalvariable s
 - Die Signalvariable kann sich zum Beispiel in einer lokalen Datei befinden
 - Nachteil: Rechenzeit der CPU wird verschwendet, weil die wait-Operation den Prozessor in regelmäßigen Abständen belegt
- Diese Technik heißt auch Warteschleife oder Spinlock

Signalisieren und Warten

- Besseres Konzept: Prozess P_B blockieren, bis Prozess P_A den Abschnitt ${\bf X}$ abgearbeitet hat
 - Vorteil: Vergeudet keine Rechenzeit des Prozessors
 - Nachteil: Es kann nur ein Prozess warten
 - Diese Technik heißt in der Literatur auch passives Warten

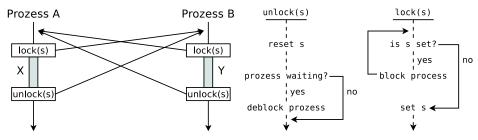


Eine Möglichkeit, um unter Linux eine Ausführungsreihenfolge mit passivem Warten festzulegen, ist die Funktion sigsuspend. Damit blockiert sich ein Prozess so lange selbst, bis ein anderer Prozess ihm mit der Funktion kill (oder dem gleichnamigen Systemaufruf) ein passendes Signal (meist SIGUSR1 oder SIGUSR2) sendet und somit signalisiert, dass er weiterarbeiten soll

Alternative Systemaufrufe und Funktionsaufrufe, mit denen sich ein Prozess selbst so lange blockieren kann, bis er durch einen Systemaufruf wieder geweckt wird, sind pause und sleep.

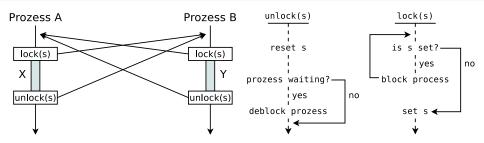
Schutz kritischer Abschnitte durch Sperren / Blockieren

- Beim Signalisieren wird immer eine Ausführungsreihenfolge festlegt
 - Soll aber einfach nur sichergestellt werden, dass es keine Überlappung in der Ausführung der kritischen Abschnitte gibt, können die beiden Operationen lock und unlock eingesetzt werden



- Sperren (Blockieren) vermeidet Überlappungen bei der Abarbeitung von 2 kritischen Abschnitten
 - ullet Beispiel: Kritische Abschnitte ${f X}$ von Prozess P_A und ${f Y}$ von Prozess P_B

Sperren und Freigeben von Prozessen unter Linux (1/2)



 $\label{thm:lifter} \mbox{Hilfreiche Systemaufrufen und Bibliotheksfunktion um die Operationen lock und unlock unter Linux zu realisieren$

sigsuspend, kill, pause und sleep

- Alternative 1: Realisierung von Sperren mit den Signalen SIGSTOP (Nr. 19) und SIGCONT (Nr. 18)
 - Mit SIGSTOP kann ein anderer Prozess gestoppt werden
 - Mit SIGCONT kann ein anderer Prozess reaktiviert werden

Sperren und Freigeben von Prozessen unter Linux (2/2)

- Alternative 2: Eine lokale Datei dient als Sperrmechanismus für wechselseitigen Ausschluss
 - Jeder Prozess prüft vor dem Eintritt in seinen kritischen Abschnitt, ob er die Datei exklusiv öffnen kann
 - z.B. mit dem Systemaufruf open oder der Bibliotheksfunktion fopen
 - Ist das nicht der Fall, muss er für eine bestimmte Zeit pausieren (z.B. mit dem Systemaufruf sleep) und es danach erneut versuchen (aktives Warten)
 - Alternativ kann er sich mit sleep oder pause selbst pausieren und hoffen, dass der Prozess, der bereits die Datei geöffnet hat ihn nach Abschluss seines kritischen Abschnitts mit einem Signal deblockiert (passives Warten)

Zusammenfassung: Unterschied zwischen Signalisieren und Blockieren

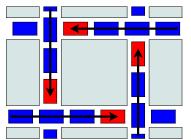
- Signalisieren legt die Ausführungsreihenfolge fest
 Beispiel: Abschnitt X von Prozess P_A vor Abschnitt Y von P_R ausführen
- Sperren / Blockieren sichert kritische Abschnitte
 Die Reihenfolge, in der die Prozesse ihre kritische Abschnitte abarbeiten, ist nicht festgelegt! Es wird nur sichergestellt, dass es keine Überlappung in der Ausführung der kritischen Abschnitte gibt

Probleme, die durch Blockieren entstehen

Bildquelle: Google Bildersuche

- Verhungern (Starvation)
 - Hebt ein Prozess eine Sperre nicht wieder auf, m

 üssen die anderen Prozesse unendlich lange auf die Freigabe warten
- Verklemmung (Deadlock)
 - Es warten mehrere Prozesse gegenseitig auf die von ihnen gesperrten Ressourcen, sperren sie sich gegenseitig
 - Da alle am Deadlock beteiligten Prozesse (ewig) warten, kann keiner ein Ereignis auslösen, dass die Situation auflöst





Bedingungen für Deadlocks

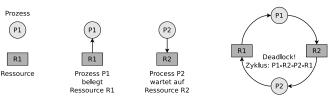
System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. Computing Surveys, Vol. 3, No. 2, June 1971, S.67-78. http://people.cs.umass.edu/-mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

- Damit ein Deadlock entstehen kann, müssen folgende Bedingungen gleichzeitig erfüllt sein:
 - Wechselseitiger Ausschluss (mutual exclusion)
 - Mindestens 1 Ressource wird von genau einem Prozess belegt oder ist verfügbar

 nicht gemeinsam nutzbar (non-sharable)
 - Anforderung weiterer Betriebsmittel (hold and wait)
 - Ein Prozess, der bereits mindestens 1 Ressource belegt, fordert weitere Ressourcen an, die von einem anderen Prozess belegt sind
 - Ununterbrechbarkeit (no preemption)
 - Die Ressourcen, die ein Prozess besitzt, k\u00f6nnen nicht vom Betriebssystem entzogen, sondern nur durch ihn selbst freigegeben werden
 - Zyklische Wartebedingung (circular wait)
 - Es gibt eine zyklische Kette von Prozessen
 - Jeder Prozess fordert eine Ressource an, die der nächste Prozess in der Kette besitzt
- Fehlt eine Bedingung, ist ein Deadlock unmöglich

Betriebsmittel-Graphen

- Mit gerichteten Graphen können die Beziehungen von Prozessen und Ressourcen dargestellt werden
- So lassen sich auch Deadlocks modellieren
 - Die Knoten sind.
 - Prozesse: Sind als Kreise dargestellt
 - Ressourcen: Sind als Rechtecke dargestellt
 - Eine Kante von einem Prozess zu einer Ressource heißt:
 - Der Prozess ist blockiert, weil er auf die Ressource wartet
 - Eine Kante von einer Ressource zu einem Prozess heißt:
 - Der Prozess belegt die Ressource



Eine umfangreiche Beschreibung zu Betriebsmittel-Graphen enthält das Buch Betriebssysteme – Eine Einführung, Uwe Baumgarten, Hans-Jürgen Siegert, 6.Auflage, Oldenbourg Verlag (2007), Kapitel 6

Deadlock-Erkennung mit Matrizen

- Ein Nachteil der Deadlock-Erkennung mit Betriebsmittel-Graphen ist, dass man damit nur einzelne Ressourcen darstellen kann
 - Gibt es mehrere Kopien (Instanzen) einer Ressource, sind Graphen zur Darstellung bzw. Erkennung von Deadlocks ungeeignet
 - Existieren von einer Ressource mehrere Instanzen, kann ein matrizenbasiertes Verfahren verwendet werden, das 2 Vektoren und 2 Matrizen benötigt
- Wir definieren 2 Vektoren
 - Ressourcenvektor (Existing Resource Vektor)
 - Zeigt an, wie viele Ressourcen von jeder Klasse existieren
 - Ressourcenrestvektor (Available Resource Vektor)
 - Zeigt an, wie viele Ressourcen von jeder Klasse frei sind
- Zusätzlich sind 2 Matrizen nötig
 - Belegungsmatrix (Current Allocation Matrix)
 - Zeigt an, welche Ressourcen die Prozesse aktuell belegen
 - Anforderungsmatrix (Request Matrix)
 - Zeigt an, welche Ressourcen die Prozesse gerne hätten

Deadlock-Erkennung mit Matrizen – Beispiel (1/2)

Quelle des Beispiels: Tanenbaum, Moderne Betriebssysteme, Pearson, 2009

Ressourcenvektor = $\begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$

- 4 Ressourcen von Klasse 1 existieren
- 2 Ressourcen von Klasse 2 existieren
- 3 Ressourcen von Klasse 3 existieren
- 1 Ressource von Klasse 4 existiert

Belegungsmatrix =
$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

- Prozess 1 belegt 1 Ressource von Klasse
 3
- Prozess 2 belegt 2 Ressourcen von Klasse 1 und 1 Ressource von Klasse 4
- Prozess 3 belegt 1 Ressource von Klasse
 2 und 2 Ressourcen von Klasse 3

Ressourcenrestvektor = $\begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$

- 2 Ressourcen von Klasse 1 sind frei
- 1 Ressource von Klasse 2 ist frei
- Keine Ressourcen von Klasse 3 sind frei
- Keine Ressourcen von Klasse 4 sind frei

- Prozess 1 ist blockiert, weil keine Ressource von Klasse 4 frei ist
- Prozess 2 ist blockiert, weil keine Ressource von Klasse 3 frei ist
- Prozess 3 ist nicht blockiert

Deadlock-Erkennung mit Matrizen – Beispiel (2/2)

• Wurde Prozess 3 fertig ausgeführt, gibt er seine Ressourcen frei

$$\mathsf{Ressourcenrestvektor} = \left(\begin{array}{cccc} 2 & 2 & 2 & 0 \end{array} \right)$$

- 2 Ressourcen von Klasse 1 sind frei
 Prozes
- 2 Ressourcen von Klasse 2 sind frei
- 2 Ressourcen von Klasse 3 sind frei
- Keine Ressourcen von Klasse 4 sind frei
- Keine Ressourcen von Klasse 4 sind fre
- Wurde Prozess 2 fertig ausgeführt, gibt er seine Ressourcen frei

Anforderungsmatrix =
$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

- Prozess 1 kann nicht laufen, weil keine Ressource vom Typ 4 frei ist
- Prozess 2 ist nicht blockiert

$$\mbox{Ressourcenrestvektor} = \left(\begin{array}{cccc} 4 & 2 & 2 & 1 \end{array} \right) \qquad \mbox{Anforderungsmatrix} = \left[\begin{array}{cccc} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{array} \right]$$

● Prozess 1 ist nicht blockiert ⇒ kein Deadlock in diesem Beispiel

Fazit zu Deadlocks

- Manchmal wird die Möglichkeit von Deadlocks akzeptiert
 - Entscheidend ist, wie wichtig ein System ist
 - Ein Deadlock, der statistisch alle 5 Jahre auftritt, ist kein Problem in einem System das wegen Hardwareausfällen oder sonstigen Softwareproblemen jede Woche ein mal abstürzt
- Deadlock-Erkennung ist aufwendig und verursacht Overhead
- In allen Betriebssystemen sind Deadlocks möglich:
 - Prozesstabelle voll
 - Es können keine neuen Prozesse erzeugt werden
 - Maximale Anzahl von Inodes vergeben
 - Es können keine neuen Dateien und Verzeichnisse angelegt werden
- ullet Die Wahrscheinlichkeit, dass so etwas passiert, ist gering, aber $\neq 0$
 - Solche potentiellen Deadlocks werden akzeptiert, weil ein gelegentlicher Deadlock nicht so lästig ist, wie die ansonsten nötigen Einschränkungen (z.B. nur 1 laufender Prozess, nur 1 offene Datei, mehr Overhead)

Gemeinsamer Speicher – Shared Memory

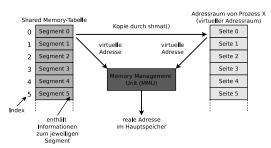
- Prozesskommunikation über einen gemeinsamen Speicher (Shared Memory) heißt auch speicherbasierte Kommunikation
- Gemeinsame Speichersegmente sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können
 - Diese Speicherbereiche liegen im Adressraum mehrerer Prozesse
- Die Prozesse müssen die Zugriffe selbst koordinieren und sicherstellen, dass ihre Speicherzugriffe sich gegenseitig ausschließen
 - Der Empfänger-Prozess darf nichts aus dem gemeinsamen Speicher lesen, bevor der Sender-Prozess fertig geschrieben hat
 - ullet Ist die Koordinierung der Zugriffe nicht sorgfältig \Longrightarrow Inkonsistenzen

Bei den anderen Formen der Interprozesskommunikation garantiert das Betriebssystem die Synchronisation der Zugriffe



Gemeinsamer Speicher unter Linux/UNIX

- Unter Linux/UNIX speichert eine Shared Memory Tabelle mit Informationen über die existierenden gemeinsamen Speichersegmente
 - Zu diesen Informationen gehören: Anfangsadresse im Speicher, Größe, Besitzer (Benutzername und Gruppe) und Zugriffsrechte



Ein gemeinsames
 Speichersegment wird
 immer über seine
 Indexnummer in der
 Shared
 Memory-Tabelle
 angesprochen

- Vorteil:
 - Ein gemeinsames Speichersegment, das an keinen Prozess gebunden ist, wird nicht automatisch vom Betriebssystem gelöscht

Mit gemeinsamem Speicher arbeiten

Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe für die Arbeit mit gemeinsamem Speicher bereit

- shmget(): Gemeinsames Speichersegment erzeugen oder auf ein bestehendes zugreifen
- shmat(): Gemeinsames Speichersegment an einen Prozess anhängen
- shmdt(): Gemeinsames Speichersegment von einem Prozess lösen/freigeben
- shmct1(): Status (u.a. Zugriffsrechte) eines gemeinsamen Speichersegments abfragen, ändern oder es löschen

Ein Beispiel zur Arbeit mit gemeinsamen Speicherbereichen unter Linux finden sie zum Beispiel auf der Webseite der Vorlesung

ipcs

Informationen über bestehende gemeinsame Speichersegmente liefert das Kommando ipcs

Gemeinsames Speichersegment erzeugen (in C)

```
1 #include <sys/ipc.h>
 2 #include <sys/shm.h>
 3 #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
 7
       int shared memory id = 12345:
 8
       int returncode_shmget;
9
10
       // Gemeinsames Speichersegment erzeugen
11
       // IPC CREAT = Speichersegment erzeugen, wenn es noch nicht existiert
12
       // 0600 = Zugriffsrechte auf das neue gemeinsame Speichersegment
13
       returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14
15
       if (returncode_shmget < 0) {
16
           printf("Das gemeinsame Speichersegment konnte nicht erstellt werden.\n");
17
           perror("shmget"):
       } else {
18
19
           printf("Das gemeinsame Speichersegment wurde erstellt.\n");
20
21 }
```

```
$ ipcs -m ------ Shared Memory Segments ------ key shmid owner perms bytes nattch status 0x00003039 56393780 bnc 600 20 0 $ printf "%d\n" 0x00003039 # Umrechnen von Hexadezimal in Dezimal 12345
```

Gemeinsames Speichersegment anhängen (in C)

```
1 #include <sys/types.h>
  #include <sys/ipc.h>
  #include <svs/shm.h>
   #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
 9
       int returncode_shmget;
10
       char *sharedmempointer:
11
12
       // Gemeinsames Speichersegment erzeugen
13
       returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14
15
16
           // Gemeinsames Speichersegment anhängen
17
           sharedmempointer = shmat(returncode shmget, 0, 0):
18
           if (sharedmempointer == (char *)-1) {
19
               printf("Das gemeinsame Speichersegment konnte nicht angehängt werden.\n");
20
               perror("shmat"):
21
           } else {
22
               printf("Das Segment wurde angehängt an Adresse %p\n", sharedmempointer);
23
           }
24
25
```

In ein Speichersegment schreiben und daraus lesen (in C)

```
1 #include <sys/types.h>
 2 #include <sys/ipc.h>
 3 #include <svs/shm.h>
 4 #include <stdio.h>
  #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
9
       int returncode_shmget, returncode_shmdt, returncode_sprintf;
10
       char *sharedmempointer:
11
12
       // Gemeinsames Speichersegment erzeugen
13
       returncode_shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
14
15
           // Gemeinsames Speichersegment anhängen
16
           sharedmempointer = shmat(returncode shmget, 0, 0);
17
18
19
           // Eine Zeichenkette in das gemeinsame Speichersegment schreiben
20
           returncode_sprintf = sprintf(sharedmempointer, "Hallo Welt.");
21
           if (returncode sprintf < 0) {
22
               printf("Der Schreibzugriff ist fehlgeschlagen.\n");
23
           } else {
24
               printf("%i Zeichen in das Segment geschrieben.\n", returncode sprintf):
25
           }
26
27
           // Die Zeichenkette im gemeinsamen Speichersegment ausgeben
28
           if (printf ("%s\n", sharedmempointer) < 0) {
29
               printf("Der Lesezugriff ist fehlgeschlagen.\n");
30
           }
31
```

Gemeinsames Speichersegment lösen (in C)

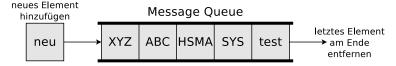
```
1 #include <sys/types.h>
  #include <sys/ipc.h>
 3 #include <svs/shm.h>
  #include <stdio.h>
  #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
 9
       int returncode_shmget;
10
       int returncode shmdt:
11
       char *sharedmempointer:
12
13
       // Gemeinsames Speichersegment erzeugen
14
       returncode shmget = shmget(shared memory id. MAXMEMSIZE. IPC CREAT | 0600):
15
16
17
           // Gemeinsames Speichersegment anhängen
18
           sharedmempointer = shmat(returncode_shmget, 0, 0);
19
20
21
           // Gemeinsames Speichersegment lösen
22
           returncode_shmdt = shmdt(sharedmempointer);
23
           if (returncode shmdt < 0) {
24
               printf("Das gemeinsame Speichersegment konnte nicht gelöst werden.\n"):
25
               perror("shmdt");
26
           } else {
27
               printf("Das Segment wurde vom Prozess gelöst.\n"):
28
           }
29
       }
30
```

Gemeinsames Speichersegment löschen (in C)

```
1 #include <sys/types.h>
 2 #include <svs/ipc.h>
 3 #include <sys/shm.h>
  #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
 8
       int shared memory id = 12345:
       int returncode_shmget;
10
       int returncode_shmctl;
11
       char *sharedmempointer;
12
13
       // Gemeinsames Speichersegment erzeugen
14
       returncode shmget = shmget(shared_memory_id, MAXMEMSIZE, IPC_CREAT | 0600);
15
16
17
           // Gemeinsames Speichersegment löschen
18
           returncode shmctl = shmctl(returncode shmget, IPC RMID, 0):
19
           if (returncode shmctl == -1) {
20
               printf("Das gemeinsame Speichersegment konnte nicht gelöscht werden.\n");
21
               perror("semctl");
22
           } else {
23
               printf("Das Segment wurde gelöscht.\n");
24
           }
25
26
```

Nachrichtenwarteschlangen - Message Queues

- Sind verketteten Listen mit Nachrichten
- Arbeiten nach dem Prinzip FIFO
- Prozesse können Daten darin ablegen und daraus abholen
- Vorteil: Auch nach Beendigung des Erzeuger-Prozesses verbleiben die Daten in der Nachrichtenwarteschlange



Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe für die Arbeit mit Nachrichtenwarteschlangen bereit

- msgget(): Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
- msgsnd(): Nachrichten in Nachrichtenwarteschlange schreiben (schicken)
- msgrcv(): Nachrichten aus Nachrichtenwarteschlange lesen (empfangen)
- msgct1(): Status (u.a. Zugriffsrechte) einer Nachrichtenwarteschlang abfragen, ändern oder sie löschen

Informationen über bestehende Nachrichtenwarteschlangen liefert das Kommando ipcs

Nachrichtenwarteschlangen erzeugen (in C)

1 #include <stdlib.h>

0x00003039 98304

12345

hn c

\$ printf "%d\n" 0x00003039

```
2 #include <sys/types.h>
 3 #include <sys/ipc.h>
  #include <stdio h>
  #include <sys/msg.h>
 6
   int main(int argc, char **argv) {
 8
       int returncode msgget:
 9
10
       // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
11
       // IPC CREAT => neue Nachrichtenwarteschlange erzeugen, wenn sie noch nicht existiert
12
       // 0600 = Zugriffsrechte auf die neue Nachrichtenwarteschlange
13
       returncode_msgget = msgget(12345, IPC_CREAT | 0600);
14
       if(returncode_msgget < 0) {
           printf("Die Nachrichtenwarteschlange konnte nicht erstellt werden.\n");
15
16
           exit(1):
17
       } else {
           printf("Die Nachrichtenwarteschlange 12345 mit der ID %i ist nun verfügbar.\n",
18
                 returncode_msgget);
19
20 1
$ ipcs -a
       Message Queues
kev
           msaid
                                  perms
                                             used-bytes
                      owner
                                                          messages
```

Umrechnen von Hexadezimal in Dezimal

600

In Nachrichtenwarteschlangen schreiben (in C)

```
1 #include <stdlib.h>
 2 #include <sys/types.h>
 3 #include <sys/ipc.h>
  #include <stdio.h>
 5 #include <svs/msg.h>
  #include <string.h>
                                             // Diese Header-Datei ist nötig für strcpy()
  struct msgbuf {
                                             // Template eines Puffers fuer msgsnd und msgrcv
 9
       long mtype;
                                             // Nachrichtentyp
10
      char mtext[80];
                                             // Sendepuffer
11
   } msg;
12
13
   int main(int argc, char **argv) {
14
       int returncode_msgget;
15
16
       // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
17
       returncode_msgget = msgget(12345, IPC_CREAT | 0600);
18
19
20
       msg.mtvpe = 1;
                                            // Nachrichtentyp festlegen
21
       strcpy(msg.mtext, "Testnachricht"); // Nachricht in den Sendepuffer schreiben
22
23
       // Eine Nachricht in die Nachrichtenwarteschlange schreiben
24
       if (msgsnd(returncode_msgget, &msg, strlen(msg.mtext), 0) == -1) {
25
           printf("In die Nachrichtenwarteschlange konnte nicht geschrieben werden.\n"):
26
           exit(1);
27
       }
28
```

• Den Nachrichtentyp (eine positive ganze Zahl) definiert der Benutzer

Ergebnis des Schreibens in die Nachrichtenwarteschlange

Vorher...

```
$ ipcs -q
----- Message Queues ------
key msqid owner perms used-bytes messages
0x00003039 98304 bnc 600 0 0
```

Nachher...

```
$ ipcs -q
----- Message Queues -------
key msqid owner perms used-bytes messages
0x00003039 98304 bnc 600 80 1
```

Aus Nachrichtenwarteschlangen lesen (in C)

```
1 #include <stdlib.h>
 2 #include <sys/types.h>
 3 #include <sys/ipc.h>
 4 #include <stdio h>
 5 #include <sys/msg.h>
 6 #include <string.h>
                                       // Diese Header-Datei ist nötig für strcpy()
                                       // Template eines Puffers fuer msgsnd und msgrcv
 7 typedef struct msgbuf {
       long mtype;
                                       // Nachrichtentyp
 9
       char mtext[80];
                                       // Sendepuffer
10
   } msg:
11
12
   int main(int argc, char **argv) {
13
       int returncode_msgget, returncode_msgrcv;
14
       msg receivebuffer:
                                       // Einen Empfangspuffer anlegen
15
16
       // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
17
       returncode msgget = msgget(12345, IPC CREAT | 0600)
18
19
       msg.mtype = 1;
                                       // Die erste Nachricht vom Typ 1 empfangen
20
       // MSG NOERROR => Nachrichten abschneiden, wenn sie zu lang sind
21
       // IPC NOWAIT => Prozess nicht blockieren, wenn keine Nachricht vom Tvp vorliegt
22
       returncode_msgrcv = msgrcv(returncode_msgget, &msg, sizeof(msg.mtext), msg.mtype,
            MSG NOERROR | IPC NOWAIT):
23
       if (returncode msgrcv < 0) {
24
           printf("Aus der Nachrichtenwarteschlange konnte nicht gelesen werden.\n");
25
           perror("msgrcv");
26
       } else {
27
           printf("Diese Nachricht wurde aus der Warteschlange gelesen: %s\n", msg.mtext);
28
           printf("Die empfangene Nachricht ist %i Zeichen lang.\n", returncode msgrcv);
29
       }
30
```

Nachrichtenwarteschlangen löschen (in C)

```
1 #include <stdlib.h>
 2 #include <sys/types.h>
 3 #include <sys/ipc.h>
  #include <stdio h>
 5 #include <svs/msg.h>
 6
   int main(int argc, char **argv) {
8
       int returncode_msgget;
 9
       int returncode_msgctl;
10
11
       // Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
12
       returncode_msgget = msgget(12345, IPC_CREAT | 0600);
13
14
15
       // Nachrichtenwarteschlange löschen
16
       returncode_msgctl = msgctl(returncode_msgget, IPC_RMID, 0);
       if (returncode msgctl < 0) {
17
           printf("Die Nachrichtenwarteschlange mit der ID %i konnte nicht gelöscht werden.\
18
                n", returncode_msgget);
19
           perror("msgctl");
20
           exit(1):
21
       } else {
22
           printf("Die Nachrichtenwarteschlange mit der ID %i wurde gelöscht.\n",
                returncode_msgget);
23
24
       exit(0);
25 }
```

Ein Beispiel zur Arbeit mit Nachrichtenwarteschlangen unter Linux finden sie zum Beispiel auf der Webseite der Vorlesung

Pipes (1/2)

Eine anonyme Pipe...

- ist ein gepufferter unidirektionaler Kommunikationskanal zwischen 2 Prozessen
 - Soll Kommunikation in beide Richtungen gleichzeitig möglich sein, sind 2
 Pipes nötig eine für jede mögliche Kommunikationsrichtung
- arbeitet nach dem FIFO-Prinzip
- hat eine begrenzte Kapazität
 - $\bullet \ \mathsf{Pipe} = \mathsf{voll} \Longrightarrow \mathsf{der} \ \mathsf{in} \ \mathsf{die} \ \mathsf{Pipe} \ \mathsf{schreibende} \ \mathsf{Prozess} \ \mathsf{wird} \ \mathsf{blockiert}$
 - $\bullet \ \mathsf{Pipe} = \mathsf{leer} \Longrightarrow \mathsf{der} \ \mathsf{aus} \ \mathsf{der} \ \mathsf{Pipe} \ \mathsf{lesende} \ \mathsf{Prozess} \ \mathsf{wird} \ \mathsf{blockiert}$
- wird mit dem Systemaufruf pipe() angelegt
 - Dabei erzeugt der Betriebssystemkern einen Inode (\Longrightarrow Foliensatz 6) und 2 Zugriffskennungen (Handles)
 - Prozesse greifen auf die Zugriffskennungen mit read() und write()-Systemaufrufen (oder Bibliotheksfunktionen) zu, um Daten aus der Pipe zu lesen bzw. um Daten in die Pipe zu schreiben



Pipes (2/2)

- Bei der Erzeugung von Kindprozessen mit fork() erben die Kindprozesse auch den Zugriff auf die Zugriffskennungen
- Anonyme Pipes ermöglichen Prozesskommunikation nur zwischen eng verwandten Prozessen
 - Nur Prozesse, die via fork() eng verwandt sind, können über anonyme Pipes kommunizieren
 - Mit der Beendigung des letzten Prozesses, der Zugriff auf eine anonyme Pipe hat, wird diese vom Betriebssystem beendet
- Via benannte Pipes (Named Pipes), können auch nicht eng miteinander verwandte Prozesse kommunizieren
 - Auf diese Pipes kann mit Hilfe ihres Namens zugegriffen werden
 - Sie werden in C erzeugt via: mkfifo("<pfadname>",<zugriffsrechte>)
 - Jeder Prozess, der den Namen kennt, kann über diesen die Verbindung zur Pipe herstellen und darüber mit anderen Prozessen kommunizieren
- Wechselseitigen Ausschluss garantiert das Betriebssystem
 - Zu jedem Zeitpunkt kann nur 1 Prozess auf eine Pipe zugreifen

Ein Beispiel zu anonymen Pipes (in C) – Teil 1/2

Ein Beispiel zur Arbeit mit benannten Pipes unter Linux finden sie zum Beispiel auf der Webseite der Vorlesung

```
#include <stdio.h>
   #include <unistd.h>
   #include <stdlib.h>
  void main() {
     int pid_des_Kindes;
 7
     // Zugriffskennungen zum Lesen (testpipe[0]) und Schreiben (testpipe[1]) anlegen
     int testpipe[2]:
10
     // Die Pipe testpipe anlegen
11
     if (pipe(testpipe) < 0) {
12
       printf("Das Anlegen der Pipe ist fehlgeschlagen.\n");
13
       // Programmabbruch
14
       exit(1):
15
     } else {
16
       printf("Die Pipe testpipe wurde angelegt.\n");
17
18
19
     // Einen Kindprozess erzeugen
20
     pid_des_Kindes = fork();
21
22
     // Es kam beim fork zu einem Fehler
23
     if (pid_des_Kindes < 0) {
24
       perror("Es kam bei fork zu einem Fehler!\n");
25
       // Programmabbruch
26
       exit(1);
27
```

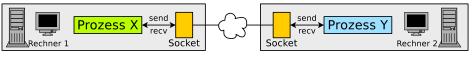
Ein Beispiel zu anonymen Pipes (in C) – Teil 2/2

```
28
     // Elternprozess
29
     if (pid_des_Kindes > 0) {
30
       printf("Elternprozess: PID: %i\n", getpid()):
31
       // Lesekanal der Pipe testpipe blockieren
32
       close(testpipe[0]);
33
       char nachricht[] = "Testnachricht";
34
       // Daten in den Schreibkanal der Pipe schreiben
35
       write(testpipe[1], &nachricht, sizeof(nachricht));
36
37
38
     // Kindprozess
39
     if (pid_des_Kindes == 0) {
40
       printf("Kindprozess: PID: %i\n", getpid());
41
       // Schreibkanal der Pipe testpipe blockieren
42
       close(testpipe[1]);
43
       // Einen Empfangspuffer mit 80 Zeichen Kapazität anlegen
44
       char puffer[80]:
45
       // Daten aus dem Lesekanal der Pipe auslesen
46
       read(testpipe[0], puffer, sizeof(puffer));
47
       // Empfangene Daten ausgeben
48
       printf("Empfangene Daten: %s\n", puffer);
49
50
```

```
$ gcc pipe_beispiel.c -o pipe_beispiel
$ ./pipe_beispiel
Die Pipe testpipe wurde angelegt.
Elternprozess: PID: 6363
Kindprozess: PID: 6364
Empfangene Daten: Testnachricht
```

Sockets

- Vollduplexfähige Alternative zu Pipes und gemeinsamem Speicher
 - Ermöglichen Interprozesskommunikation in verteilten Systemen
- Ein Benutzerprozess kann einen Socket vom Betriebssystem anfordern, und über diesen anschließend Daten verschicken und empfangen
 - Das Betriebssystem verwaltet alle benutzten Sockets und die zugehörigen Verbindungsinformationen



- Zur Kommunikation über Sockets werden Ports verwendet
 - Die Vergabe der Portnummern erfolgt beim Verbindungsaufbau
 - Portnummern werden vom Betriebssystem zufällig vergeben
 - Ausnahmen sind Ports bekannter Anwendungen, wie z.B. HTTP (80), SMTP (25), Telnet (23), SSH (22), FTP (21),...
- Einsatz von Sockets ist blockierend (synchron) und nicht-blockierend (asynchron) möglich

Verschiedene Arten von Sockets

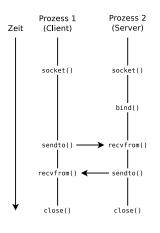
- Verbindungslose Sockets (bzw. Datagram Sockets)
 - Verwenden das Transportprotokoll UDP
 - Vorteil: Höhere Geschwindigkeit als bei TCP
 - Grund: Geringer Mehraufwand (Overhead) f
 ür das Protokoll
 - Nachteil: Segmente können einander überholen oder verloren gehen
- Verbindungsorientierte Sockets (bzw. Stream Sockets)
 - Verwenden das Transportprotokoll TCP
 - Vorteil: Höhere Verlässlichkeit
 - Segmente können nicht verloren gehen
 - Segmente kommen immer in der korrekten Reihenfolge an
 - Nachteil: Geringere Geschwindigkeit als bei UDP
 - Grund: Höherer Mehraufwand (Overhead) für das Protokoll

Sockets nutzen

- Praktisch alle gängigen Betriebssystemen unterstützen Sockets
 - Vorteil: Bessere Portabilität der Anwendungen
- Funktionen f
 ür Kommunikation via Sockets:
 - Erstellen eines Sockets:
 socket.()
 - Anbinden eines Sockets an eine Portnummer und empfangsbereit machen: bind(), listen(), accept() und connect()
 - Senden/Empfangen von Nachrichten über den Socket: send(), sendto(), recv() und recvfrom()
 - Schließen eines Sockets: shutdown() oder close()

Übersicht der Sockets unter Linux/UNIX: netstat -n oder lsof | grep socket

Verbindungslose Kommunikation mit Sockets – UDP



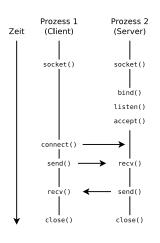
Client

- Socket erstellen (socket)
- Daten senden (sendto) und empfangen (recvfrom)
- Socket schließen (close)

Server

- Socket erstellen (socket)
- Socket an einen Port binden (bind)
- Daten senden (sendto) und empfangen (recvfrom)
- Socket schließen (close)

Verbindungsorientierte Kommunikation mit Sockets – TCP



Client

- Socket erstellen (socket)
- Client mit Server-Socket verbinden (connect)
- Daten senden (send) und empfangen (recv)
- Socket schließen (close)

Server

- Socket erstellen (socket)
- Socket an einen Port binden (bind)
- Socket empfangsbereit machen (listen)
 - Richtete eine Warteschlange für Verbindungen mit Clients ein
- Server akzeptiert Verbindungsanforderung (accept)
- Daten senden (send) und empfangen (recv)
- Socket schließen (close)

Einen Socket erzeugen: socket

int socket(int domain, int type, int protocol);

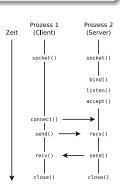
- Ein Aufruf von socket() liefert einen Integerwert zurück
 - Der Wert heißt Socket-Deskriptor (socket file descriptor)
- domain: Legt die Protokollfamilie fest
 - PF_UNIX: Lokale Prozesskommunikation unter Linux/UNIX
 - PF_INET: IPv4
 - PF_INET6: IPv6
- type: Legt den Typ des Sockets (und damit auch das Protokoll) fest:
 - SOCK_STREAM: Stream Socket (TCP)
 - SOCK_DGRAM: Datagram Socket (UDP)
 - SOCK_RAW: RAW-Socket (IP)
- Der Parameter protocol hat meist den Wert Null
- Einen Socket mit socket() erzeugen:

```
1 sd = socket(PF_INET, SOCK_STREAM, 0);
2    if (sd < 0) {
3        perror("Der Socket konnte nicht erzeugt werden");
4        return 1;
5    }</pre>
```

Adresse und Portnummer binden: bind

int bind(int sd, struct sockaddr *address, int addrlen);

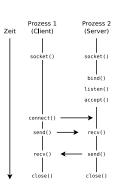
- bind() bindet den neu erstellen Socket (sd) an die Adresse (address) des Servers
 - sd ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von socket()
 - address ist eine Datenstruktur, die die IP-Adresse des Server und eine Portnummer enthält
 - addrlen ist die Länge der Datenstruktur, die die IP-Adresse und Portnummer enthält



Server empfangsbereit machen: listen

int listen(int sd, int backlog);

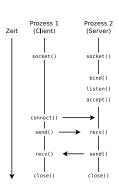
- listen() definiert, wie viele Verbindungsanfragen am Socket gepuffert werden können
 - Ist die listen()-Warteschlange voll, werden weitere Verbindungsanfragen von Clients abgewiesen
 - sd ist der Socket-Deskriptor aus dem vorhergehenden Aufruf von socket()
 - backlog enthält die Anzahl der möglichen Verbindungsanforderungen, die die Warteschlange maximal speichern kann
 - Standardwert: 5
 - Ein Server für Datagrame (UDP) braucht listen() nicht aufzurufen, da er keine Verbindungen zu Clients einrichtet



Eine Verbindungsanforderung akzeptieren: accept

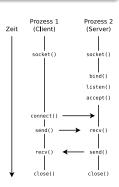
int accept(int sd, struct sockaddr *address, int *addrlen);

- Mit accept() holt der Server die erste Verbindungsanforderung aus der Warteschlange
- Der Rückgabewert ist der Socket-Deskriptor des neuen Sockets
- Enthält die Warteschlange keine Verbindungsanforderungen, ist der Prozess blockiert, bis eine Verbindungsanforderung eintrifft
- address enthält die Adresse des Clients
- Nachdem eine Verbindungsanforderungen mit accept() angenommen wurde, ist die Verbindung mit dem Client vollständig aufgebaut



Verbindung durch den Client herstellen

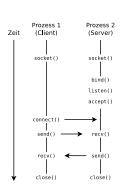
- Via connect() versucht der Client eine Verbindung mit einem Server-Socket herzustellen
- sd ist der Socket-Deskriptor
- servaddr ist die Adresse des Servers
- addrlen ist die Länge der Datenstruktur, die die Adresse enthält



Verbindungsorientierter Datenaustausch: send und recv

```
int send(int sd, char *buffer, int nbytes, int flags);
int recv(int sd, char *buffer, int nbytes, int flags);
```

- Mit send() und recv() werden über eine bestehende Verbindung Daten ausgetauscht
- send() sendet eine Nachricht (buffer) über den Socket (sd)
- recv() empfängt eine Nachricht vom Socket sd und legt diese in den Puffer (buffer)
- sd ist der Socket-Deskriptor
- buffer enthält die zu sendenden bzw. empfangenen Daten
- nbytes gibt die Anzahl der Bytes im Puffer an
- Der Wert von flags ist in der Regel Null



Verbindungsorientierter Datenaustausch: read und write

```
int read(int sd, char *buffer, int nbytes);
int write(int sd, char *buffer, int nbytes);
```

- Unter UNIX könnten im Normalfall auch read() und write() zum Empfangen und Senden über einen Socket verwendet werden
 - Der Normalfall ist, wenn der Parameter flags bei send() und recv() den Wert 0 hat
- Folgende Aufrufe haben das gleiche Ergebnis:

```
send(socket, "Hello World", 11,0);
write(socket, "Hello World", 11);
```

Verbindungsloser Datenaustausch: sendto und recvfrom

- Weiß ein Prozess, an welche Adresse (Host und Port), also an welchen Socket er Daten senden soll, verwendet er dafür sendto()
- sendto() übermittelt mit den Daten immer die lokale Adresse
- sd ist der Socket-Deskriptor
- buffer enthält die zu sendenden bzw. empfangenen Daten
- nbytes gibt die Anzahl der Bytes im Puffer an
- to enthält die Adresse des Empfängers
- from enthält die Adresse des Senders
- addrlen ist die Länge der Datenstruktur, die die Adresse enthält

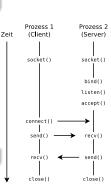
Socket schließen: close

int shutdown(int sd, int how);

- shutdown() schließt eine bidirektionale Socket-Verbindung
- Der Parameter how legt fest, ob künftig keine Daten mehr empfangen werden sollen (how=0), keine mehr gesendet werden (how=1), oder beides (how=2)

```
int close(int sd);
```

 Wird close() anstatt shutdown() verwendet, entspricht dies einem shutdown(sd,2)



Sockets via UDP - Beispiel (Server)

```
1 #!/usr/bin/env pvthon
 2 # -*- coding: iso-8859-15 -*-
  # Server: Empfängt eine Nachricht via UDP
  # Modul socket importieren
  import socket
   # Stellvertretend für alle Schnittstellen des Hosts
    '' = alle Schnittstellen
   HOST = ''
   # Portnummer des Servers
   PORT = 50000
13
14
  # Socket erzeugen und Socket-Deskriptor zurückliefern
   sd = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
16
17 try:
18
     sd.bind((HOST, PORT))
                                    # Socket an Port
          binden
19
     while True:
20
       data = sd.recvfrom(1024)
                                    # Daten empfangen
21
       # Empfangene Daten ausgeben
22
       print 'Received:', repr(data)
23
   finally:
24
       sd.close()
                                      # Socket schließen
```

```
Prozess 1
                         Prozess 2
Zeit
        (Client)
                          (Server)
        socket()
                          socket()
                           bind()
        sendto() -
                      recvfrom()
       recvfrom() ←
                        — sendto()
         close()
                           close()
```

\$ python udp_server.py

Sockets via UDP – Beispiel (Client)

Received: ('Hello World', ('127.0.0.1', 39834))

\$ python udp server.py

```
1 #!/usr/bin/env python
 2 # -*- coding: iso-8859-15 -*-
                                                                                    Prozess 2
                                                                     Prozess 1
  # Client: Schickt eine Nachricht via UDP
                                                               7eit
                                                                      (Client)
                                                                                    (Server)
  import socket
                               # Modul socket importieren
   HOST = 'localhost'
                             # Hostname des Servers
   PORT = 50000
                               # Portnummer des Servers
                                                                      socket()
                                                                                     socket()
   MESSAGE = 'Hello World'
                               # Nachricht
10
   # Socket erzeugen und Socket-Deskriptor zurückliefern
   sd = socket.socket(socket.AF INET. socket.SOCK DGRAM)
                                                                                     bind()
13
  # Nachricht an Socket senden
14
  sd.sendto(MESSAGE, (HOST, PORT))
16
                                                                      sendto() ------ recvfrom()
  sd.close()
                             # Socket schließen
                                                                     recvfrom() ← sendto()
   $ python udp_client.py
```

close()

close()

Sockets via TCP - Beispiel (Server)

```
#!/usr/bin/env python
2 # -*- coding: iso-8859-15 -*-
  # Echo Server via TCP
  import socket
                              # Modul socket importieren
  HOST = ''
                              # '' = alle Schnittstellen
  PORT = 50007
                              # Portnummer des Servers
  # Socket erzeugen und Socket-Deskriptor zurückliefern
   sd = socket.socket(socket.AF INET, socket.SOCK STREAM)
  # Socket an Port binden
   sd.bind((HOST, PORT))
  # Socket empfangsbereit machen
  # Max. Anzahl Verbindungen = 1
  sd.listen(1)
  # Socket akzeptiert Verbindungen
   conn, addr = sd.accept()
17
18
   print 'Connected by', addr
19
20
   while 1:
                              # Endlosschleife
21
       data = conn.recv(1024) # Daten empfangen
22
                              # Endlosschleife abbrechen
      if not data: break
23
       # Empfangene Daten zurücksenden
24
       conn.send(data)
25
  conn.close()
                              # Socket schließen
```

```
Prozess 1
                           Prozess 2
Zeit
         (Client)
                           (Server)
         socket()
                            socket()
                            bind()
                           listen()
                            accept()
        connect()
         send() -
                            recv()
          recv()
                            send()
         close()
                            close()
```

\$ python tcp_server.py

Sockets via TCP - Beispiel (Client)

```
#!/usr/bin/env python
2 # -*- coding: iso-8859-15 -*-
3 # Echo Client via TCP
  # Modul socket importieren
  import socket
  HOST = 'localhost'
                             # Hostname des Servers
  PORT = 50007
                             # Portnummer des Servers
10 # Socket erzeugen und Socket-Deskriptor zurückliefern
  sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  # Mit Server-Socket verbinden
  sd.connect((HOST, PORT))
14
15 sd.send('Hello, world') # Daten senden
16 data = sd.recv(1024) # Daten empfangen
  sd.close()
                             # Socket schließen
18
19 # Empfangene Daten ausgeben
20 print 'Empfangen:', repr(data)
```

```
$ python tcp_client.py
Empfangen: 'Hello, world'
```

```
$ python tcp_server.py
Connected by ('127.0.0.1', 49898)
```

```
Prozess 1
                           Prozess 2
Zeit
         (Client)
                           (Server)
         socket()
                            socket()
                             bind()
                            listen()
                            accept()
        connect()
          send()
                             recv()
          recv()
                             send()
         close()
                            close()
```

Blockierende und nicht-blockierende Sockets

- Wird ein Socket erstellt, ist er standardmäßig im blockierenden Modus
 - Alle Methodenaufrufe warten, bis die von ihnen angestoßene Operation durchgeführt wurde
 - z.B. blockiert ein Aufruf von recv() den Prozess bis Daten eingegangen sind und aus dem internen Puffer des Sockets gelesen werden können
- Die Methode setblocking() ändert den Modus eines Sockets
 - ullet sd.setblocking(0) \Longrightarrow versetzt in den nicht-blockierenden Modus
 - ullet sd.setblocking(1) \Longrightarrow versetzt in den blockierenden Modus
- Es ist möglich, während des Betriebs den Modus jederzeit umzuschalten
 - z.B. könnte man die Methode connect() blockierend und anschließend read() nicht-blockierend verwenden

Quelle: Peter Kaiser, Johannes Ernesti. Python – Das umfassende Handbuch. Galileo (2008)

Nicht-blockierende Sockets – Einige Auswirkungen

- recv() und recvfrom()
 - Die Methoden geben nur dann Daten zurück, wenn sich diese bereits im internen Puffer des Sockets befinden
 - Sind keine Daten im Puffer, werfen die Methoden eine Exception und die Programmausführung läuft weiter
- send() und sendto()
 - Die Methoden versenden die angegebenen Daten nur, wenn sie direkt in den Ausgangspuffer des Sockets geschrieben werden können
 - Ist der Puffer schon voll, werfen die Methoden eine Exception und die Programmausführung läuft weiter
- connect()
 - Die Methode sendet eine Verbindungsanfrage an den Zielsocket und wartet nicht, bis diese Verbindung zustande kommt
 - Wird connect() aufgerufen, während die Verbindungsanfrage noch läuft, wird eine Exception geworfen
 - Durch mehrmaliges Aufrufen von connect() kann man überprüfen, ob die Operation immer noch durchgeführt wird

Vergleich der Kommunikations-Systeme

	Gemeinsamer Speicher	Nachrichten- warteschlangen	(anon./benannte) Pipes	Sockets
Art der Kommunikation	Speicherbasiert	Nachrichtenbasiert	Nachrichtenbasiert	Nachrichtenbasiert
Bidirektional	ja	nein	nein	ja
Plattformunabhäng	nein	nein	nein	ja
Prozesse müssen verwandt sein	nein	nein	bei anonymen Pipes	nein
Kommunikation über Rechnergrenzen	nein	nein	nein	ja
Bleiben ohne gebundenen	ja	ja	nein	nein
Prozess erhalten				
Automatische Synchronisierung	nein	ja	ja	ja

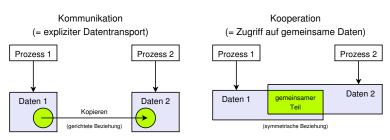
- Vorteile nachrichtenbasierter Kommunikation gegenüber speicherbasierter Kommunikation:
 - \bullet Das Betriebssystem nimmt den Benutzerprozessen die Synchronisation der Zugriffe ab \Longrightarrow komfortabel
 - Einsetzbar in verteilten Systemen ohne gemeinsamen Speicher
 - Bessere Portabilität der Anwendungen

Speicher kann über Netzwerkverbindungen eingebunden werden

- Das ermöglicht speicherbasierte Kommunikation zwischen Prozessen auf verschiedenen, unabhängigen Systemen
- Das Problem der Synchronisation der Zugriffe besteht aber auch hier

Kooperation

- Kooperation
 - Semaphor
 - Mutex



Semaphoren

- Zur Sicherung (Sperrung) kritischer Abschnitte können außer den bekannten Sperren auch Semaphoren eingesetzt werden
- 1965: Veröffentlicht von Edsger W. Dijkstra
- ullet Ein Semaphor ist eine Zählersperre ullet mit Operationen P(S) und V(S)
 - ullet V kommt vom holländischen verhogen = erhöhen
 - P kommt vom holländischen *proberen* = versuchen (zu verringern)
- Die Zugriffsoperationen sind atomar ⇒ nicht unterbrechbar (unteilbar)
- Kann auch mehreren Prozessen das Betreten des kritischen Abschnitts erlauben
 - Im Gegensatz zu Semaphoren können Sperren (⇒ Folie 14) immer nur einem Prozess das Betreten des kritischen Abschnitts erlauben

Die korrekte Grammatik ist das Semaphor, Plural die Semaphore

 $\textbf{Cooperating sequential processes}. \ \textit{Edsger W. Dijkstra} \ (1965)$

https://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF

Zugriffsoperationen auf Semaphoren (1/3)

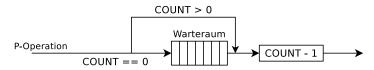
Ein Semaphor besteht aus 2 Datenstrukturen

- COUNT: Eine ganzzahlige, nichtnegative Zählvariable.
 Gibt an, wie viele Prozesse das Semaphor aktuell ohne Blockierung passieren dürfen
- Ein Warteraum für die Prozesse, die darauf warten, das Semaphor passieren zu dürfen.
 Die Prozesse sind im Zustand blockiert und warten darauf, vom Betriebssystem in den Zustand bereit überführt zu werden, wenn das Semaphor den Weg freigibt
- Initialisierung: Zuerst wird ein Semaphor erzeugt oder ein bestehendes Semaphor geöffnet
 - Bei einem neuen Semaphor wird zu Beginn die Zählvariable mit einem nichtnegativen Anfangswert initialisiert

Zugriffsoperationen auf Semaphoren (2/3)

Bildquelle: Carsten Vogt

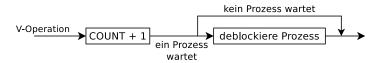
- P-Operation (verringern): Prüft den Wert der Zählvariable
 - Ist der Wert 0, wird der Prozess blockiert
 - Ist der Wert > 0, wird er um 1 erniedrigt



Zugriffsoperationen auf Semaphoren (3/3)

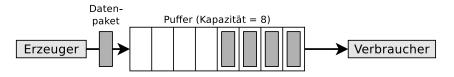
Bildquelle: Carsten Vogt

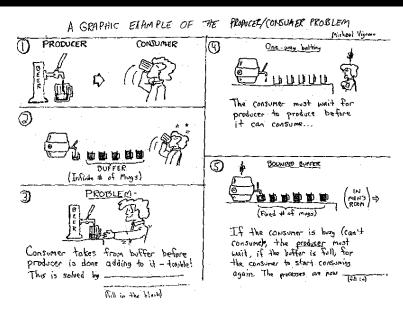
- V-Operation (erhöhen): Erhöht als erstes die Zählvariable um 1
 - Befinden sich Prozesse im Warteraum, wird ein Prozess deblockiert
 - Der gerade deblockierte Prozess setzt dann seine P-Operation fort und erniedrigt als erstes die Zählvariable



Erzeuger/Verbraucher-Beispiel (1/3)

- Ein Erzeuger schickt Daten an einen Verbraucher
- Ein endlicher Zwischenspeicher (Puffer) soll Wartezeiten des Verbrauchers minimieren
- Daten werden vom Erzeuger in den Puffer gelegt und vom Verbraucher aus diesem entfernt
- Gegenseitiger Ausschluss ist notwendig, um Inkonsistenzen zu vermeiden
- ullet Puffer = voll \Longrightarrow Erzeuger muss blockieren
- Puffer = leer ⇒ Verbraucher muss blockieren





Quelle: http://www.ccs.neu.edu/home/kenb/synchronize.html

Erzeuger/Verbraucher-Beispiel (2/3)

- Zur Synchronisation der Zugriffe werden 3 Semaphore verwendet:
 - leer
 - voll
 - mutex
- Semaphore voll und leer werden gegenläufig zueinander eingesetzt
 - leer zählt die freien Plätze im Puffer, wird vom Erzeuger (P-Operation) erniedrigt und vom Verbraucher (V-Operation) erhöht
 - ullet leer $=0\Longrightarrow$ Puffer vollständig belegt \Longrightarrow Erzeuger blockieren
 - voll zählt die Datenpakete (belegte Plätze) im Puffer, wird vom Erzeuger (V-Operation) erhöht und vom Verbraucher (P-Operation) erniedrigt
 - $voll = 0 \Longrightarrow Puffer leer \Longrightarrow Verbraucher blockieren$
- Semaphor mutex ist für den wechselseitigen Ausschluss zuständig

Binäre Semaphore

- Binäre Semaphore werden mit dem Wert 1 initialisiert und garantieren, dass 2 oder mehr Prozesse nicht gleichzeitig in ihre kritischen Bereiche eintreten können
- Beispiel: Das Semaphor mutex aus dem Erzeuger/Verbraucher-Beispiel

Erzeuger/Verbraucher-Beispiel (3/3)

```
typedef int semaphore;
                                              // Semaphore sind von Typ Integer
   semaphore voll = 0;
                                              // zählt die belegten Plätze im Puffer
  semaphore leer = 8:
                                              // zählt die freien Plätze im Puffer
   semaphore mutex = 1:
                                              // steuert Zugriff auf kritische Bereiche
   void erzeuger (void) {
     int daten:
 9
     while (TRUE) {
                                              // Endlosschleife
10
       erzeugeDatenpaket(daten);
                                              // erzeuge Datenpaket
11
       P(leer):
                                              // Zähler "leere Plätze" erniedrigen
12
       P(mutex);
                                              // in kritischen Bereich eintreten
13
       einfuegenDatenpaket(daten);
                                              // Datenpaket in den Puffer schreiben
14
       V(mutex):
                                              // kritischen Bereich verlassen
15
       V(voll);
                                              // Zähler für volle Plätze erhöhen
16
17
18
19
   void verbraucher (void) {
20
     int daten:
21
22
     while (TRUE) {
                                              // Endlosschleife
23
       P(voll):
                                              // Zähler "volle Plätze" erniedrigen
24
       P(mutex):
                                              // in kritischen Bereich eintreten
25
       entferneDatenpaket(daten);
                                              // Datenpaket aus dem Puffer holen
26
     V(mutex);
                                              // kritischen Bereich verlassen
27
       V(leer):
                                              // Zähler für leere Plätze erhöhen
28
       verbraucheDatenpaket(daten);
                                              // Datenpaket nutzen
29
30
```

Beispiel zu Semaphore: PingPong

```
1 // Initialisierung der Semaphore
2 s init (Sema Ping, 1);
  s init (Sema Pong, 0);
  task Ping is
  begin
7
      loop
           P(Sema Ping);
           print("Ping");
           V(Sema_Pong);
10
11
      end loop;
  end Ping;
13
  task Pong is
  begin
      loop
16
           P(Sema Pong):
17
           print("Pong, ");
18
           V(Sema Ping);
19
       end loop;
21 end Pong;
```

 Die beiden Endlosprozesse Ping und Pong geben endlos folgendes aus: PingPong, PingPong, PingPong...

Beispiel zu Semaphore: 3 Läufer (1/3)

- 3 Läufer sollen hintereinander eine bestimmte Strecke laufen
 - Der zweite Läufer darf erst starten, wenn der erste Läufer im Ziel ist
 - Der dritte Läufer darf erst starten, wenn der zweite Läufer im Ziel ist
- Ist diese Lösung korrekt?

```
1 // Initialisierung der Semaphore
2 s init (Sema. 0):
3
  task Erster is
           < laufer >
           V(Sema);
  task Zweiter is
           P(Sema):
           < laufen >
10
           V(Sema);
11
12
13 task Dritter is
           P(Sema);
14
           < laufen >
15
```

Beispiel zu Semaphore: 3 Läufer (2/3)

- Die Lösung ist nicht korrekt!
- Es existieren 2 Reihenfolgebeziehungen:
 - Läufer 1 vor Läufer 2
 - Läufer 2 vor Läufer 3
- Beide Reihenfolgebeziehungen verwenden das gleiche Semaphor
 - Es ist nicht ausgeschlossen, dass Läufer 3 mit seiner P-Operation vor Läufer 2 das Semaphor um den Wert 1 erniedrigt

```
    Wie könnte eine korrekte
```

```
// Initialisierung der Semaphore
  s_init (Sema, 0);
  task Erster is
           < laufen >
           V(Sema);
  task Zweiter is
           P(Sema):
           < laufen >
10
           V(Sema):
11
12
  task Dritter is
14
           P(Sema):
           < laufen >
15
```

Beispiel zu Semaphore: 3 Läufer (3/3)

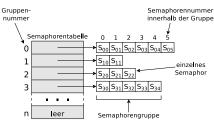
- Lösungsmöglichkeit:
 - Zweiten Semaphor einführen
 - Das zweites Semaphor wird ebenfalls mit dem Wert 0 initialisiert
 - Läufer 2 erhöht mit seiner V-Operation das zweite Semaphor und Läufer 3 erniedrigt dieses mit seiner P-Operation

```
1 // Initialisierung der Semaphore
2 s_init (Sema1, 0);
  s_init (Sema2, 0);
  task Erster is
           < laufen >
           V(Sema1):
  task Zweiter is
           P(Sema1):
10
           < laufen >
11
           V(Sema2):
12
13
  task Dritter is
           P(Sema2):
15
           < laufen >
16
```

Semaphoren unter Linux

Bildquelle: Carsten Vogt

- Linux weicht vom Konzept der Semaphore nach Dijkstra ab
 - Die Z\u00e4hlvariable kann mit einer P- oder V-Operation um mehr als 1 erh\u00f6ht bzw. erniedrigt werden
 - Es können mehrere Zugriffsoperationen auf verschiedenen Semaphoren atomar, also unteilbar, durchgeführt werden
- Linux-Systeme verwalten eine Semaphortabelle, die Verweise auf Arrays mit Semaphoren enthält
 - Einzelne Semaphoren werden über den Tabellenindex und die Position in der Gruppe angesprochen



Linux/UNIX-Betriebssysteme stellen 3 Systemaufrufe für die Arbeit mit Semaphoren bereit

- semget(): Neues Semaphor oder eine Gruppe von Semaphoren erzeugen oder ein bestehendes Semaphor öffnen
- semct1(): Wert eines existierenden Semaphors oder einer Semaphorengruppe abfragen, ändern oder ein Semaphor löschen
- semop(): P- und V-Operationen auf Semaphoren durchführen
- Informationen über bestehende Semaphore liefert das Kommando ipcs

Mutexe

- Wird die Möglichkeit eines Semaphors zu zählen nicht benötigt, kann die vereinfachte Version eines Semaphors, der Mutex, verwendet werden
 - Mutexe (abgeleitet von Mutual Exclusion = wechselseitiger Ausschluss) dienen dem Schutz kritischer Abschnitte, auf die zu jedem Zeitpunkt immer nur ein Prozess zugreifen darf
 - Mutexe können nur 2 Zustände annehmen: belegt und nicht belegt
 - Mutexe haben die gleiche Funktionalität wie binäre Semaphore

2 Funktion zum Zugriff existieren

```
\begin{array}{ll} \mathtt{mutex\_lock} & \Longrightarrow & \mathtt{entspricht} \ \mathsf{der} \ \mathsf{P-Operation} \\ \mathtt{mutex\_unlock} & \Longrightarrow & \mathtt{entspricht} \ \mathsf{der} \ \mathsf{V-Operation} \end{array}
```

- Will ein Prozess auf den kritischen Abschnitt zugreifen, ruft er mutex_lock auf
 - Ist der kritische Abschnitt gesperrt, wird der Prozess blockiert, bis der Prozess im kritischen Abschnitt fertig ist und mutex_unlock aufruft
 - Ist der kritische Abschnitt nicht gesperrt kann der Prozess eintreten

IPC-Objekte kontrollieren und löschen

- Informationen über bestehende gemeinsame Speichersegmente liefert das Kommando ipcs
- Die einfachste Möglichkeit, Semaphore, gemeinsame Speichersegmente und Nachrichtenwarteschlangen auf der Kommandozeile zu löschen, ist das Kommando ipcrm

```
ipcrm [-m shmid] [-q msqid] [-s semid]
      [-M shmkey] [-Q msgkey] [-S semkey]
```

- Oder alternativ einfach...
 - ipcrm shm SharedMemoryID
 - ipcrm sem SemaphorID
 - ipcrm msg MessageQueueID