6. Foliensatz Betriebssysteme und Rechnernetze

Prof. Dr. Christian Baun

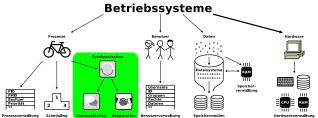
Frankfurt University of Applied Sciences (1971-2014: Fachhochschule Frankfurt am Main) Fachbereich Informatik und Ingenieurwissenschaften christianbaun@fb2.fra-uas.de

Prozessinteraktion

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
 - mögliche Probleme (Verhungern und Deadlocks) beim Blockieren
 - wie Deadlock-Erkennung mit Matrizen funktioniert
 - verschiedene Möglichkeiten der Kommunikation zwischen Prozessen:
 - Gemeinsamer Speicher, Nachrichtenwarteschlangen, Pipes, Sockets
 - verschiedene Möglichkeiten der Kooperation von Prozessen
 - wie Semaphore und Mutexe kritische Abschnitte sichern können

Übungsblatt 6 wiederholt die für die Lernziele relevanten Inhalte dieses Foliensatzes



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?

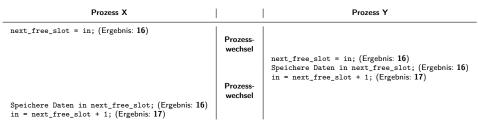
Frage: Wie verhält es sich hier mit Threads?

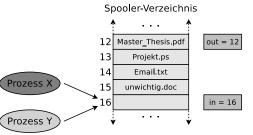
- Bei Threads gelten die gleichen Herausforderungen und Lösungen wie bei Interprozesskommunikation mit Prozessen
- Nur die Kommunikation zwischen den Threads eines Prozesses ist problemlos möglich, weil sie im gleichen Adressraum agieren

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





- 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 60) 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 tödliche Unfälle durch mangelhafte Programmierung und Qualitätssicherung
 - Einige Patienten erhielten eine bis zu hundertfach erhöhte Strahlendosis

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



Bildquelle: Google Bildersuche. Häufig gezeigtes Bild in diesem Kontext. (Autor und Lizenz: unbekannt)

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

- Eine Race Condition ("Texas-Bug") führte zu fehlerhaften Einstellungen des Geräts und damit zu erhöhter Strahlendosis
 - Der Kontroll-Prozess synchronisierte nicht korrekt mit dem Prozess der Eingabeaufforderung
 - Der Fehler trat nur während einer schnellen Eingabekorrektur (Zeitfenster: 8 Sekunden) durch den Benutzer auf
 - Bei Tests trat der Fehler nicht auf, weil es Erfahrung (Routine) erforderte, um das Gerät so schnell zu bedienen

The Worst Computer Bugs in History: Race conditions in Therac-25: https://www.bugsnag.com/blog/bug-day-race-condition-therac-25

"Once the data entry phase was marked complete, the magnet setting phase began. However, if a specific sequence of edits was applied in the Data Entry phase during the 8 second magnet setting phase, the setting was not applied to the machine hardware, due to the value of the completion variable. The UI would then display the wrong mode to the user, who would confirm the potentially lethal treatment."

Weitere interessante Quellen

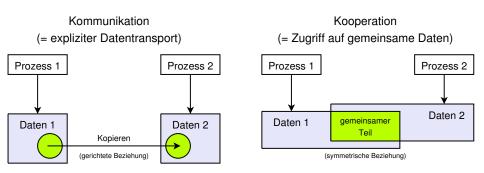
Prozessinteraktion

00000000

https://www-dssz.informatik.tu-cottbus.de/information/slides_studis/ss2009/mehner_RisikoComputer_zs09.pdf Killer Bug. Therac-25: Quick-and-Dirty. https://www.viva64.com/en/b/0438/ Killed by a machine: The Therac-25: https://backaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

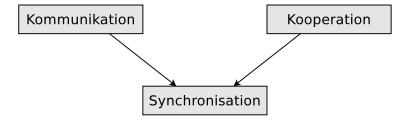
Kommunikation vs. Kooperation

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



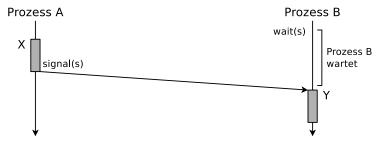
• Kommunikation und Kooperation basieren auf Synchronisation

- 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

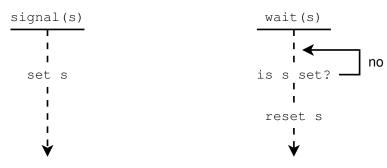


Signalisierung

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



Einfachste Form der Signalisierung (aktives Warten)

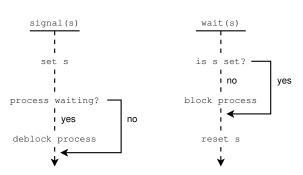


- Die Abbildung zeigt aktives Warten an der Signalvariable s
 - Die Signalvariable kann sich zum Beispiel in einer lokalen Datei befinden
 - Nachteil: Rechenzeit der CPU wird verschwendet, weil die wait-Operation den Prozessor in regelmäßigen Abständen belegt
- Diese Technik heißt auch Warteschleife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

Signalisieren und Warten

Prozessinteraktion

- 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

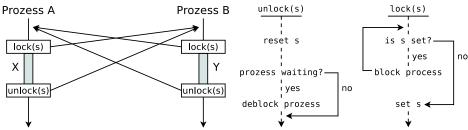


Eine Möglichkeit, um unter Linux eine Ausführungsreihenfolge mit passivem Warten festzulegen, ist die Funktion sigsuspend. Damit blockiert sich ein Prozess so lange selbst, bis ein anderer Prozess ihm mit der Funktion kill (oder dem gleichnamigen Systemaufruf) ein passendes Signal (meist SIGUSR1 oder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

Schutz kritischer Abschnitte durch Sperren / Blockieren

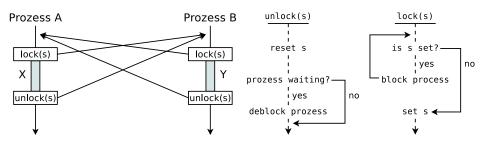
- Beim Signalisieren wird immer eine Ausführungsreihenfolge festlegt
 - Soll aber einfach nur sichergestellt werden, dass es keine Überlappung in der Ausführung der kritischen Abschnitte gibt, können die beiden Operationen lock und unlock eingesetzt werden



- 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

Kooperation von Prozessen

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



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_B 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

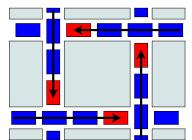
Probleme, die durch Blockieren entstehen

Verhungern (Starvation)

 Hebt ein Prozess eine Sperre nicht wieder auf, müssen 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





Quelle: https://i.redd.it/vvu6v8pxvue11.jpg (Autor und Lizenz: unbekannt)

Bedingungen für Deadlocks

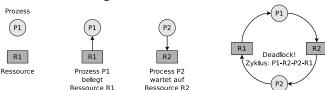
System Deadlocks. E. G. Coffman, M. J. Elphick, A. Shoshani. Computing Surveys, Vol. 3, No. 2, June 1971, S.67-78. http://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

- 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 der genannten Bedingungen, kann kein Deadlock entstehen

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
 - Fine Kante von einem Prozess zu einer Ressource heißt:
 - Der Prozess ist blockiert, weil er auf die Ressource wartet
 - Fine 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, Belegungsmatrix und Anforderungsmatrix sind gegeben! Nur der Ressourcenrestvektor muss zu Beginn und nach ieder Prozessausführung neu berechnet werden.

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

$$\mathsf{Ressourcenrestvektor} = \left(\begin{array}{cccc} 2 & 1 & 0 & 0 \end{array} \right)$$

- 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

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

- 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

Prozessinteraktion

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

Wurde Prozess 3 fertig ausgeführt, gibt er seine Ressourcen frei

$$Ressourcen rest vektor = \left(\begin{array}{cccc} 2 & 2 & 2 & 0 \end{array} \right)$$

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

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

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

Wurde Prozess 2 fertig ausgeführt, gibt er seine Ressourcen frei

Ressourcenrestvektor =
$$\begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$
 Anforderungsmatrix = $\begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$

Prozess 1 ist nicht blockiert ⇒ kein Deadlock in diesem Beispiel

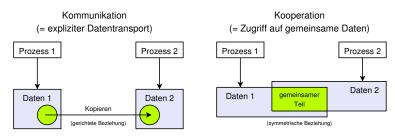
Fazit zu Deadlocks

Prozessinteraktion

- 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. Beispiele:
 - Prozesstabelle (⇒ Foliensatz 4) voll
 - Es können keine neuen Prozesse erzeugt werden
 - Maximale Anzahl von Inodes (\Longrightarrow Foliensatz 3) vergeben
 - Es können keine neuen Dateien und Verzeichnisse angelegt werden
- ullet Die Wahrscheinlichkeit, dass so etwas passiert, ist gering, aber eq 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)

Kommunikation von Prozessen

- Kommunikation
 - Gemeinsamer Speicher (Shared Memory)
 - Nachrichtenwarteschlangen (Message Queues)
 - Pipes
 - Sockets



Gemeinsamer Speicher – Shared Memory

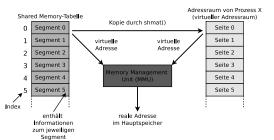
- Prozesskommunikation über einen gemeinsamen Speicher (Shared Memory) heißt auch speicherbasierte Kommunikation
- **Gemeinsame Speichersegmente** sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können
 - Diese Speicherbereiche liegen im Adressraum mehrerer Prozesse
- Die Prozesse müssen 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
 - \bullet 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

Beim Neustart des Betriebssystems sind die gemeinsamen Speichersegmente und deren Inhalte verloren

Mit gemeinsamem Speicher arbeiten (System V vs. POSIX)

Linux/UNIX-Betriebssysteme stellen 4 Systemaufrufe (System V) 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
- Informationen über bestehende gemeinsame Speichersegmente (System V) liefert das Kommando ipcs

Ein Beispiel zu gemeinsamen Speicherbereichen (System V) unter Linux finden auf der Webseite der Vorlesung

Einige Entwickler bevorzugen die System V API und andere die POSIX API... \\(\(\bar{V} \)_/\'

C-Funktionsaufrufe für gemeinsame POSIX-Speichersegmente (teilweise in der Header-Datei mman.h definiert)

- shm_open(): Ein Segment erzeugen oder auf ein bestehendes zugreifen
- ftruncate(): Die Größe eines Speichersegments definieren
- mmap(): Ein Segment an einen Prozess anhängen
- munmap(): Ein Segment von einem Prozess lösen/freigeben
- close(): Den Deskriptor eines Speichersegments schließen
- shm_unlink(): Ein Segment löschen
- Unter Linux liegen POSIX-Speichersegmente im Verzeichnis /dev/shm

Ein Beispiel zu gemeinsamen POSIX-Speichersegmenten unter Linux finden sie auf der Webseite der Vorlesung

Gemeinsames Speichersegment (System V) erzeugen (in C)

```
1 #include <sys/ipc.h>
  #include <svs/shm.h>
 3 #include <stdio.h>
   #define MAXMEMSIZE 20
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
       int returncode shmget:
 g
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");
18
       } else {
19
           printf("Das gemeinsame Speichersegment wurde erstellt.\n");
20
       }
21 }
```

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

Gemeins. Speichersegment (System V) anhängen (in C)

```
1 #include <svs/tvpes.h>
 2 #include <svs/ipc.h>
   #include <sys/shm.h>
   #include <stdio.h>
  #define MAXMEMSIZE 20
 6
 7
   int main(int argc, char **argv) {
       int shared_memory_id = 12345;
       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");
           } else {
21
22
               printf("Das Segment wurde angehängt an Adresse %p\n", sharedmempointer);
23
           }
24
25
```

In ein Speichersegment (System V) schreiben und daraus lesen (in C)

```
1 #include <svs/tvpes.h>
 2 #include <svs/ipc.h>
 3 #include <sys/shm.h>
  #include <stdio.h>
 5 #define MAXMEMSIZE 20
6
7
   int main(int argc, char **argv) {
       int shared memory id = 12345:
 8
9
       int returncode_shmget, returncode_shmdt, returncode_sprintf;
       char *sharedmempointer;
10
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) {
               printf("Der Schreibzugriff ist fehlgeschlagen.\n");
22
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 (System V) lösen (in C)

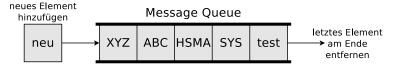
```
1 #include <svs/tvpes.h>
 2 #include <svs/ipc.h>
  #include <sys/shm.h>
   #include <stdio.h>
  #define MAXMEMSIZE 20
 6
 7
   int main(int argc, char **argv) {
 8
       int shared memory id = 12345:
9
       int returncode_shmget;
10
       int returncode_shmdt;
11
       char *sharedmempointer:
12
13
       // Gemeinsames Speichersegment erzeugen
       returncode shmget = shmget(shared memory id. MAXMEMSIZE. IPC CREAT | 0600):
14
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
       7
30 1
```

Gemeinsames Speichersegment (System V) löschen (in C)

```
#include <sys/types.h>
  #include <sys/ipc.h>
  #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");
               perror("semctl");
21
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



$Linux/UNIX-Betriebs systeme\ stellen\ 4\ Systemaufrufe\ (System\ V)\ f\"ur\ die\ Arbeit\ mit\ Nachrichtenwarteschlangen\ bereit$

- msgget(): Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
- msgsnd(): Nachricht in Nachrichtenwarteschlange schreiben (schicken)
- msgrcv(): Nachricht aus Nachrichtenwarteschlange lesen (empfangen)
 - msgct1(): Status (u.a. Zugriffsrechte) einer Nachrichtenwarteschlange abfragen, ändern oder sie löschen
- Informationen über bestehende Nachrichtenwarteschlangen (System V) liefert das Kommando ipcs

#include <stdlib.h>

Nachrichtenwarteschlangen (System V) erzeugen (in C)

```
#include <svs/tvpes.h>
  #include <sys/ipc.h>
   #include <stdio.h>
   #include <svs/msg.h>
   int main(int argc, char **argv) {
       int returncode msgget:
 g
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) {
15
            printf("Die Nachrichtenwarteschlange konnte nicht erstellt werden.\n"):
16
            exit(1):
17
       } else {
18
            printf("Die Nachrichtenwarteschlange 12345 mit der ID %i ist nun verfügbar.\n",
                 returncode msgget):
19
       7
20
$ ipcs -q
```

```
Message Queues
key
           msqid
                                             used-bytes
                      owner
                                  perms
                                                           messages
0x00003039 98304
                                  600
                      hn c
$ printf "%d\n" 0x00003039
                                   # Umrechnen von Hexadezimal in Dezimal
12345
```

```
1 #include <stdlib.h>
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <stdio.h>
  #include <svs/msg.h>
   #include <string.h>
                                             // Diese Header-Datei ist nötig für strcpv()
  struct msgbuf {
                                             // Template eines Puffers fuer msgsnd und msgrcv
       long mtvpe:
                                             // Nachrichtentvp
10
       char mtext[80];
                                             // Nachrichtengroesse
11
   } msg;
                                             // msg = Name des Datenverbunds
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 (System V) lesen (in C)

Prozessinteraktion

```
1 #include <stdlib.h>
 2 #include <svs/tvpes.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()
7 typedef struct msgbuf {
                                       // Template eines Puffers fuer msgsnd und msgrcv
       long mtype;
                                       // Nachrichtentvp
       char mtext[80];
                                       // Nachrichtengroesse
                                       // msg = Name des Datenverbunds
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.mtvpe = 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 (System V) löschen (in C)

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

Ein Beispiel zur Arbeit mit System V-Nachrichtenwarteschlangen unter Linux finden sie auf der Webseite der Vorlesung

Nachrichtenwarteschl. unter Linux (System V vs. POSIX)

- Die bislang beschriebenen Funktionen zur Arbeit mit Nachrichtenwarteschlangen sind Teil der System V-Schnittstelle
- Einige Entwickler bevorzugen die System V API und andere die POSIX API. . . ¯_(ツ)_/¯

In der Header-Datei mqueue.h definierte C-Funktionsaufrufe der POSIX-Nachrichtenwarteschlangen

- mq_open(): Nachrichtenwarteschlange erzeugen oder auf eine bestehende zugreifen
- mq_send(): Nachricht in eine Nachrichtenwarteschlange schreiben (schicken). Blockierende Anweisung
- mq_timedsend(): Nachricht in eine Nachrichtenwarteschlange schreiben (schicken). Blockierende Anweisung mit Timeout
- mq_receive(): Nachricht aus einer Nachrichtenwarteschlange lesen (empfangen). Blockierende Anweisung
- mq_timedreceive(): Nachricht aus einer Nachrichtenwarteschlange lesen (empfangen). Blockierende Anweisung mit Timeout
- mq_getattr(): Eigenschaften einer Nachrichtenwarteschlange abfragen. Diese sind: Anzahl der Nachrichten in der Warteschlange, maximale Nachrichtengröße, maximale Anzahl an Nachrichten, etc.
- mq_setattr(): Eigenschaften einer Nachrichtenwarteschlange ändern
- mq_notify(): Der Prozess soll benachrichtigt werden, sobald eine Nachricht vorliegt
- mq_close(): Nachrichtenwarteschlange schließen

Prozessinteraktion

- nq_unlink(): Nachrichtenwarteschlange löschen
- Unter Linux liegen POSIX-Speichersegmente im Verzeichnis /dev/mqueue

Ein Beispiel zur Arbeit mit benannten POSIX-Nachrichtenwarteschlangen unter Linux finden sie auf der Webseite der Vorlesung

Anonyme Pipes (1/2)

Prozessinteraktion

- Pipes können anonyme oder benannte Pipes (siehe Folie 44) sein
- 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
 - Pipe = voll ⇒ der in die Pipe schreibende Prozess wird blockiert
 - Pipe = leer ⇒ der aus der Pipe lesende Prozess wird blockiert
 - wird mit dem Systemaufruf pipe() angelegt
 - Dabei erzeugt der Betriebssystemkern einen Inode (⇒ 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

Anonyme Pipes (2/2)

Prozessinteraktion



- 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

Übersicht der Pipes unter Linux/UNIX: 1sof | grep pipe

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

Sie können die anonyme Pipe unter Linux/UNIX mit 1sof -n -P | grep <PID> und im Verzeichnis /proc/<PID>/fd überwachen

```
#include <stdio.h>
   #include <unistd.h>
   #include <stdlib h>
  void main() {
 6
     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 anonymen Pipe ist fehlgeschlagen.\n");
13
       // Programmabbruch
14
       exit(1):
     } else {
15
16
       printf("Die anonyme Pipe testpipe wurde angelegt.\n"):
17
     }
18
19
     // Einen Kindprozess erzeugen
20
     pid_des_Kindes = fork();
21
22
     if (pid_des_Kindes < 0) {
23
       perror("Es kam bei fork zu einem Fehler!\n");
24
       // Programmabbruch
25
       exit(1):
26
```

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

```
27
     // Elternprozess
28
     if (pid des Kindes > 0) {
29
       printf("Elternprozess: PID: %i\n", getpid());
30
       // Lesekanal der Pipe testpipe blockieren
31
       close(testpipe[0]);
32
       char nachricht[] = "Testnachricht";
33
       // Daten in den Schreibkanal der Pipe schreiben
34
       write(testpipe[1], &nachricht, sizeof(nachricht));
35
36
37
     // Kindprozess
38
     if (pid des Kindes == 0) {
39
       printf("Kindprozess: PID: %i\n", getpid());
40
       // Schreibkanal der Pipe testpipe blockieren
41
       close(testpipe[1]);
42
       // Einen Empfangspuffer (80 Zeichen Kapazität) anlegen
43
       char puffer[80]:
44
       // Daten aus dem Lesekanal der Pipe auslesen
45
       read(testpipe[0], puffer, sizeof(puffer));
46
       printf("Empfangen: %s\n", puffer);
47
48
```

```
$ gcc anonyme_pipe_beispiel.c -o anonyme_pipe_beispiel

$ ./anonyme_pipe_beispiel

Die anonyme Pipe testpipe wurde angelegt.

Elternprozess: PID: 6363

Kindprozess: PID: 6364

Empfangen: Testnachricht
```

Benannte Pipes

- 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
- Benannte Pipes werden vom Betriebssystem nicht automatisch gelöscht (im Gegensatz zu anonymen Pipes)

Ein Beispiel zu benannten Pipes (in C) – Teil 1/4

```
#include <stdio.h>
  #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <fcntl.h>
  #include <svs/stat.h>
  void main() {
     int pid des Kindes:
 g
10
     // Die benannte Pipe anlegen
11
     if (mkfifo("testfifo",0666) < 0) {
12
       printf("Das Anlegen der benannten Pipe ist fehlgeschlagen.\n"):
13
       exit(1):
14
     } else {
15
       printf("Die benannte Pipe testfifo wurde angelegt.\n"):
16
17
18
     // Einen Kindprozess erzeugen
19
     pid des Kindes = fork():
20
21
     if (pid_des_Kindes < 0) {
22
       perror("Es kam bei fork zu einem Fehler!\n");
23
       exit(1);
24
```

Der Funktionsaufruf erzeugt im aktuellen Verzeichnis einen Dateisystemeintrag mit dem Namen testfifo. Der erste Buchstabe in der Ausgabe des Kommandos 1s zeigt, dass testfifo eine benannte Pipe ist.

```
$ ls -la testfifo
prw-r--r-- 1 bnc bnc 0 1. Feb 10:15 testfifo
```

Ein Beispiel zu benannten Pipes (in C) – Teil 2/4

```
25
     // Elternprozess
26
     if (pid des Kindes > 0) {
27
       printf("Elternprozess: PID: %i\n", getpid());
28
29
       // Zugriffskennung für die benannte Pipe anlegen
30
       int fd;
31
32
       // Die zu übertragene Nachricht definieren
33
       char nachricht[] = "Testnachricht":
34
35
       // Die benannte Pipe für Schreibzugriffe öffnen
36
       fd = open("testfifo", O WRONLY):
37
38
       // Daten in die benannte Pipe schreiben
39
       write(fd. &nachricht, sizeof(nachricht));
40
41
       // Die benannte Pipe schließen
42
       close(fd):
43
```

Ein Beispiel zu benannten Pipes (in C) – Teil 3/4

```
44
     // Kindprozess
45
     if (pid_des_Kindes == 0) {
       printf("Kindprozess: PID: %i\n", getpid());
46
47
48
       // Zugriffskennung für die benannte Pipe anlegen
49
       int fd:
50
       // Einen Empfangspuffer anlegen
51
       char puffer[80];
52
53
       // Die benannte Pipe für Lesezugriffe öffnen
54
       fd = open("testfifo", O_RDONLY);
55
56
       // Daten aus der Pipe auslesen
57
       read(fd, puffer, sizeof(puffer)):
58
       printf("Empfangen:: %s\n", puffer);
59
60
       // Die beannnte Pipe schließen
61
       close(fd):
62
63
       // Die benannte Pipe löschen
64
       if (unlink("testfifo") < 0) {
65
         printf("Das Löschen der benannten Pipe ist fehlgeschlagen.\n");
66
         exit(1):
67
       } else {
68
         printf("Die benannte Pipe wurde gelöscht.\n");
69
70
71
```

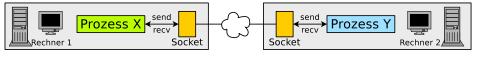
Ein Beispiel zu benannten Pipes (in C) – Teil 4/4

```
$ gcc benannte_pipe_beispiel.c -o benannte_pipe_beispiel
$ ./benannte_pipe_beispiel
Die benannte Pipe testfifo wurde angelegt.
Elternprozess: PID: 403660
Kindprozess: PID: 403661
Empfangen: Testnachricht
Die benannte Pipe wurde gelöscht.
```

Sie können die benannte Pipe unter Linux/UNIX mit lsof -n -P | grep <PID> und im Verzeichnis /proc/<PID>/fd überwachen

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),...

Verschiedene Arten von Sockets

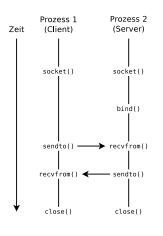
- Verbindungslose Sockets (bzw. Datagram Sockets)
 - Verwenden das Transportprotokoll UDP
 - Vorteil: Höhere Geschwindigkeit als bei TCP
 - Grund: Geringer Mehraufwand (Overhead) f
 ür das Protokoll
 - Nachteil: Segmente können einander überholen oder verloren gehen
- Verbindungsorientierte Sockets (bzw. 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

- Praktisch alle gängigen Betriebssysteme 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

Beispiele zur Interprozesskommunikation via Sockets (TCP and UDP) unter Linux finden Sie auf der Webseite der Vorlesung

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)

Sockets via UDP – Beispiel (Server)

Prozessinteraktion

```
#include <stdio.h>
   #include <stdlib.h>
 3 #include <string.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
 6 #include <unistd h>
   #include <arpa/inet.h>
   int main(int argc, char *argv[]) {
10
     int sd, adresse_laenge;
11
     char puffer[1024] = { 0 };
12
     struct sockaddr in adresse, client adresse;
13
     memset(&adresse, 0, sizeof(adresse));
14
     memset(&client_adresse, 0, sizeof(client_adresse));
15
     adresse.sin family = AF INET:
16
     adresse.sin addr.s addr = INADDR ANY:
17
     adresse.sin_port = htons(atoi(argv[1]));
18
19
     sd = socket(AF INET, SOCK DGRAM, 0):
20
     bind(sd, (struct sockaddr *) &adresse, sizeof(adresse));
21
     adresse_laenge = sizeof(client_adresse);
22
     recvfrom(sd, (char *)puffer, sizeof(puffer), 0,
23
             (struct sockaddr *) &client_adresse, &adresse_laenge);
24
     printf("Empfangene Nachricht: %s\n",puffer);
25
     char antwort[]="Server: Nachricht empfangen.\n":
26
     sendto(sd. (const char *)antwort, sizeof(antwort), 0,
27
            (struct sockaddr *) &client_adresse, adresse_laenge);
28
     close(sd):
29
     exit(0):
30
```

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

\$ gcc udp_server.c -o udp_server
\$./udp_server 50002

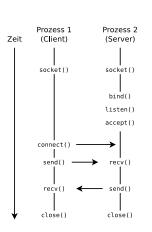
Prozessinteraktion

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

\$ gcc udp_client.c -o udp_client \$./udp_client 127.0.0.1 50002 Bitte Nachricht eingeben: Test Server: Nachricht empfangen.

\$./udp_server 50002
Empfangene Nachricht: Test

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)
 - Warteschlange für Verbindungsanfragen einrichten. Definiert wie viele Verbindungsanfragen gepuffert werden können
- Verbindungsanforderung akzeptieren (accept)
 - Erste Verbindungsanforderung aus der Warteschlange holen
- Daten senden (send) und empfangen (recv)
- Socket schließen (close)

```
#include <stdio.h>
   #include <stdlib.h>
  #include <string.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <unistd.h>
   #include <arpa/inet.h>
   int main(int argc, char *argv[]) {
10
     int sd, fd, adresse_laenge;
11
     char puffer[1024] = { 0 };
12
     struct sockaddr in adresse:
13
     memset(&adresse, 0, sizeof(adresse));
14
     adresse.sin_family = AF_INET;
15
     adresse.sin addr.s addr = INADDR ANY:
16
     adresse.sin port = htons(atoi(argv[1])):
17
18
     sd = socket(AF INET, SOCK STREAM, 0):
     bind(sd, (struct sockaddr *) &adresse, sizeof(adresse));
19
20
     listen(sd, 5);
21
     adresse_laenge = sizeof(adresse);
22
     fd = accept(sd. (struct sockaddr *) &adresse. &adresse laenge):
23
     read(fd, puffer, sizeof(puffer));
24
     printf("Empfangene Nachricht: %s\n",puffer);
25
     char antwort[]="Server: Nachricht empfangen.\n":
26
     write(fd, antwort, sizeof(antwort)):
27
     close(fd):
28
     close(sd):
29
     exit(0):
30
```

```
Prozess 1
                           Prozess 2
7eit
         (Client)
                            (Server)
         socket()
                             socket()
                             bind()
                            listen()
                             accept()
         connect()
          send()
                              recv()
          recv()
                             send()
          close()
                             close()
```

gcc tcp_server.c -o tcp_server \$./tcp_server 50003

```
#include <stdio.h>
   #include <stdlib.h>
 3 #include <string.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
  #include <unistd.h>
   #include <arpa/inet.h>
   int main(int argc, char *argv[]) {
10
     int sd;
11
     char puffer[1024] = { 0 };
12
     struct sockaddr in adresse:
13
     memset(&adresse, 0, sizeof(adresse));
14
     adresse.sin_family = AF_INET;
15
     adresse.sin port = htons(atoi(argv[2]));
16
     adresse.sin addr.s addr = inet addr(argv[1]):
17
18
     sd = socket(AF INET, SOCK STREAM, 0):
19
     connect(sd, (struct sockaddr *) &adresse, sizeof(adresse));
20
21
     printf("Bitte Nachricht eingeben: ");
22
     fgets(puffer, sizeof(puffer), stdin);
23
     write(sd, puffer, strlen(puffer));
24
     memset(puffer, 0, sizeof(puffer));
25
     read(sd. puffer, sizeof(puffer)):
26
     printf("%s\n",puffer);
27
28
     close(sd):
29
     exit(0):
30
```

```
Prozess 1
                           Prozess 2
7eit
         (Client)
                            (Server)
         socket()
                            socket()
                             bind()
                            listen()
                            accept()
        connect()
          send()
                             recv()
          recv()
                             send()
         close()
                            close()
```

\$ gcc tcp_client.c -o tcp_client \$./tcp_client 127.0.0.1 50003 Bitte Nachricht eingeben: Test Server: Nachricht empfangen.

\$./tcp server 50003 Empfangene Nachricht: Test

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 Prozess erhalten	ja	ja	nein	nein
Automatische Synchronisierung	nein	ja	ja	ja

- Vorteile nachrichtenbasierte vs. speicherbasierte 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

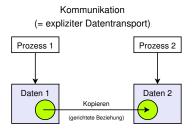
Speicher kann über Netzwerkverbindungen eingebunden werden

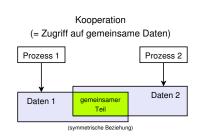
- Oas ermöglicht speicherbasierte Kommunikation zwischen Prozessen auf verschiedenen, unabhängigen Systemen
- Das Problem der Synchronisation der Zugriffe besteht aber auch hier

 Aufgabe des verwendeten Protokolls (z.B. NFS) oder verteilten Dateisystems (z.B. GlusterFS oder Ceph)

Kooperation von Prozessen

- Kooperation
 - Semaphor
 - Mutex





Semaphoren

Prozessinteraktion

- 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 $verhogen = erh\"{o}hen$
 - P kommt vom holländischen *proberen* = versuchen (zu verringern)
- Kann auch mehreren Prozessen das Betreten des kritischen Abschnitts erlauben
 - Im Gegensatz zu Semaphoren können Sperren (\Longrightarrow Folie 14) immer nur einem Prozess das Betreten des kritischen Abschnitts erlauben

Die korrekte Grammatik ist das Semaphor, Plural die Semaphore

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

$\overline{\mathsf{Z}}\mathsf{ugriff}\mathsf{soperationen}$ auf $\mathsf{Semaphoren}$ (1/3)

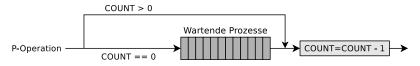
Ein Semaphor besteht aus 2 Datenstrukturen

- 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

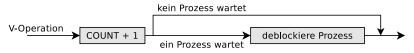
- P-Operation (verringern): Prüft den Wert der Zählvariable
 - Ist der Wert 0, wird der Prozess blockiert
 - Ist der Wert > 0, wird er um 1 erniedrigt



$\overline{\text{Zugriffsoperationen auf Semaphoren (3/3)}}$

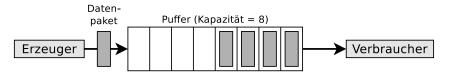
Bildquelle: Carsten Vogt

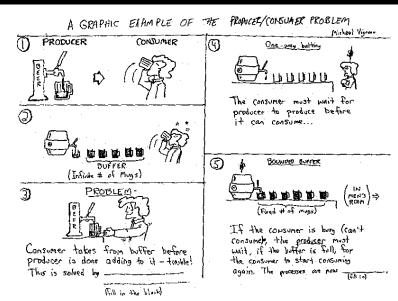
- V-Operation (erhöhen): Erhöht als erstes die Zählvariable um 1
 - Befinden sich Prozesse im Warteraum, wird ein Prozess deblockiert
 - Der gerade deblockierte Prozess setzt dann seine P-Operation fort und erniedrigt als erstes die Zählvariable



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
- Puffer = voll ⇒ Erzeuger muss blockieren
- Puffer = leer ⇒ Verbraucher muss blockieren





Quelle: Kenneth Baclawski (Northeastern University in Boston), Bildquelle: Michael Vigneau (Lizenz: unbekannt) http://www.ccs.neu.edu/home/kenb/tutorial/example.gif

Erzeuger/Verbraucher-Beispiel (2/3)

- Zur Synchronisation der Zugriffe werden 3 Semaphore verwendet:
 - leer

Prozessinteraktion

- 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
 - ullet vol1 = 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)

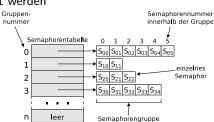
Prozessinteraktion

```
typedef int semaphore;
                                              // Semaphore sind von Tvp 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;
 8
     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 }
```

Semaphoren unter Linux (System V)

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 System V-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

Ein einfaches Beispiel zu Semaphore (in \overline{C}) – Teil 1/5

Das Programm erstellt einen Kindprozess. Eltern- und Kindprozess versuchen beide, Zeichen in der Shell auszugeben (kritischer Abschnitt). Die Prozesse sollen abwechselnd je ein Zeichen ausgeben. Zwei Semaphore ermöglichen den gegenseitigen Ausschluss

```
1 #include <stdio.h> // für printf
 2 #include <stdlib.h> // für exit
   #include <unistd.h> // für read, write, close
   #include <sys/wait.h> // für wait
   #include <svs/sem.h> // für semget, semctl, semop
 6
   void main() {
     int pid des kindes:
9
     int sem kev1=12345:
10
     int sem_key2=54321;
11
     int returncode semget1. returncode semget2. returncode semct1:
12
     int output;
13
14
     setbuf(stdout, NULL); // Das Puffern Standardausgabe (stdout) unterbinden
15
16
     // Neue Semaphorgruppe 12345 mit einer Semaphore erstellen
17
     // IPC CREAT = Semaphore erzeugen, wenn Sie noch nicht existiert
18
     // IPC_EXCL = Neuen Semaphorgruppe anlegen und nicht auf evtl. existierende Gruppe zugreifen
19
     returncode_semget1 = semget(sem_key1, 1, IPC_CREAT | IPC_EXCL | 0600);
20
     if (returncode semget1 < 0) {
21
       printf("Die Semaphorgruppe %i konnte nicht erstellt werden.\n", sem kev1):
22
       perror("semget"):
23
       exit(1);
24
```

Gute Dokumentation von semget

Ein einfaches Beispiel zu Semaphore (in C) – Teil 2/5

```
// Neue Semaphorgruppe 54321 mit einer Semaphore erstellen
25
26
     returncode semget2 = semget(sem kev2. 1. IPC CREAT | IPC EXCL | 0600):
27
     if (returncode_semget2 < 0) {</pre>
28
       printf("Die Semaphorgruppe %i konnte nicht erstellt werden.\n", sem kev2):
29
       perror("semget");
30
       exit(1);
31
32
33
     // P-Operation definieren. Wert der Semaphore um eins dekrementieren
34
     struct sembuf p operation = {0, -1, 0};
35
36
     // V-Operation definieren. Wert der Semaphore um eins inkrementieren
37
     struct sembuf v_operation = {0, 1, 0};
38
39
     // Erste Semaphore der Semaphorgruppe 12345 initial auf Wert 1 setzen
40
     returncode_semctl = semctl(returncode_semget1, 0, SETVAL, 1);
41
42
     // Erste Semaphore der Semaphorgruppe 54321 initial auf Wert O setzen
     returncode semctl = semctl(returncode semget2, 0, SETVAL, 0);
43
44
45
     // Initialen Wert der ersten Semaphore der Semaphorgruppe 12345 zur Kontrolle ausgeben
     output = semctl(returncode_semget1, 0, GETVAL, 0);
46
47
     printf("Wert der Semaphore mit ID %i und Key %i: %i\n", returncode semget1, sem key1, output);
48
49
     // Initialen Wert der ersten Semaphore der Semaphorgruppe 54321 zur Kontrolle ausgeben
50
     output = semctl(returncode_semget2, 0, GETVAL, 0);
51
     printf("Wert der Semaphore mit ID %i und Kev %i: %i\n", returncode semget2, sem kev2, output):
```

Gute Dokumentation von semct1

https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semctl/semctl.html

Ein einfaches Beispiel zu Semaphore (in C) – Teil 3/5

```
52
     // Einen Kindprozess erzeugen
53
     pid des kindes = fork():
54
55
     // Kindprozess
56
     if (pid des kindes == 0) {
57
       for (int i=0;i<5;i++) {
58
         semop(returncode_semget2, &p_operation, 1); // P-Operation Semaphore 54321
59
         // Kritischer Abschnitt (Anfang)
60
         printf("B");
61
         sleep(1);
62
         // Kritischer Abschnitt (Ende)
63
         semop(returncode semget1, &v operation, 1): // V-Operation Semaphore 12345
64
65
       exit(0):
66
67
68
     // Elternprozess
69
     if (pid des kindes > 0) {
70
       for (int i=0;i<5;i++) {
71
         semop(returncode_semget1, &p_operation, 1); // P-Operation Semaphore 12345
72
         // Kritischer Abschnitt (Anfang)
73
         printf("A");
74
         sleep(1);
75
         // Kritischer Abschnitt (Ende)
76
         semop(returncode semget2, &v operation, 1): // V-Operation Semaphore 54321
77
78
```

Gute Dokumentation von semop

https://www.nt.th-koeln.de/fachgebiete/inf/diplom/semwork/unix/semop/semop.html

Ein einfaches Beispiel zu Semaphore (in C) – Teil 4/5

```
79
      // Warten auf die Beendigung des Kindprozesses
80
      wait(NULL):
81
82
      printf("\n"):
83
84
      // Semaphorgruppe 12345 entfernen
85
      returncode semctl = semctl(returncode semget1, 0, IPC RMID, 0);
86
        if (returncode semctl < 0) {
87
          printf("Die Semaphorgruppe %i konnte nicht entfernt werden.\n", returncode semget1);
88
          exit(1):
89
      } else {
90
          printf("Die Semaphorgruppe mit ID %i und Key %i wurde entfernt.\n", returncode_semget1, sem key1);
91
92
93
      // Semaphorgruppe 54321 entfernen
94
      returncode_semctl = semctl(returncode_semget2, 0, IPC_RMID, 0);
95
        if (returncode semctl < 0) {
          printf("Die Semaphorgruppe %i konnte nicht entfernt werden.\n", returncode semget2);
96
97
          exit(1):
98
      } else {
99
          printf("Die Semaphorgruppe mit ID %i und Kev %i wurde entfernt.\n", returncode semget2, sem kev2):
100
101
102
      exit(0);
103 }
```

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

Ein einfaches Beispiel zu Semaphore (in C) – Teil 5/5

```
$ gcc semaphore_beispiel_systemv.c -o semaphore_beispiel_systemv

$ ./semaphore_beispiel_systemv

Wert der Semaphore mit ID 98362 und Key 12345: 1

Wert der Semaphore mit ID 98363 und Key 54321: 0

ABABABABAB

Die Semaphorgruppe mit ID 98362 und Key 12345 wurde entfernt.

Die Semaphorgruppe mit ID 98363 und Key 54321 wurde entfernt.
```

```
$ ipcs -s
----- Semaphore Arrays -----
kev
           semid
                       owner
                                  perms
                                              nsems
                                  600
0x00003039 98362
                       bnc
0x0000d431 98363
                       bnc
                                  600
$ printf "%d\n" 0x00003039
                                   # Convert from hexadecimal to decimal
12345
$ printf "%d\n" 0x0000d431
54321
```

- Ohne gegenseitigen Ausschluss mittels der Semaphoren ist die Ausgabe
 z. B. so: ABBABABABA oder ABBAABABAB oder ABABABABABA . . .
- Ohne gegenseitigen Ausschluss mittels der Semaphoren und ohne die sleep-Befehle ist die Ausgabesequenz normalerweise AAAAABBBB und in relativ seltenen Fällen ähnlich wie AABAAABBBB

Semaphoren unter Linux (System V vs. POSIX)

- Das in bislang beschriebene Konzept zum Schutz kritischer Abschnitte heißt in der Literatur auch System V-Semaphoren
- Einige Entwickler bevorzugen die System V API und andere die POSIX API... ~_(ツ)_/¯

In der Header-Datei semaphore.h definierte C-Funktionsaufrufe der POSIX-Semaphoren

- sem_init(): Neue unbenannte Semaphore erzeugen und dabei den initialen Wert definieren
- sem_open(): Neue **benannte** Semaphore erzeugen und dabei den initialen Wert definieren
- sem_post(): Den Wert einer Semaphore inkrementieren (V-Operation)
- sem_wait(): Den Wert einer Semaphore dekrementieren (P-Operation). Blockierende Anweisung
- sem_trywait(): Den Wert einer Semaphore dekrementieren (P-Operation). Nicht-blockierende Anweisung
- sem wait(): Den Wert einer Semaphore dekrementieren (P-Operation). Blockierende Anweisung mit definiertem Timeout
- sem_getvalue(): Den Wert einer Semaphore abfragen
- sem_destroy(): Eine unbenannte Semaphore löschen
- sem_close(): Eine benannte Semaphore schließen
- sem unlink(): Eine benannte Semaphore löschen
- Benannte POSIX-Semaphoren werden unter Linux im Verzeichnis /dev/shm with names of the form sem.<name>

Ein Beispiel zur Arbeit mit benannten POSIX-Semaphoren unter Linux finden sie auf der Webseite der Vorlesung

Mutexe

Prozessinteraktion

- Wird die Möglichkeit eines Semaphors zu zählen nicht benötigt, kann die vereinfachte Version, 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

Verschiedene Implementierungen des Mutex-Konzepts existieren

- C-Standard-Bibliothek: mtx_init, mtx_unlock ("V-Operation"), mtx_lock ("P-Operation"), mtx_trylock, mtx_timedlock, mtx_destroy
- POSIX-Threads: pthread_mutex_init, pthread_mutex_unlock, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_timedlock, pthread_mutex_destroy
- C-Standard-Bibliothek (Sun/Oracle Solaris): mutex_init, mutex_unlock, mutex_lock, mutex_trylock, mutex_destroy
- Fokus: Kooperation von Threads eines Prozesses (Intra-Prozesssynchronisation)
 - Kooperation von Prozessen (Inter-Prozesssynchronisation) ist nicht immer möglich und wenn, dann über den Umweg via ein gemeinsames Speichersegment (System V oder POSIX)

IPC-Objekte kontrollieren und löschen

- Informationen über bestehende (System V)-Speichersegmente, (System V)-Nachrichtenwarteschlangen und (System V)-Semaphoren liefert das Kommando ipcs
- Die einfachste Möglichkeit, solche gemeinsamen Speichersegmente, Nachrichtenwarteschlangen und Semaphoren auf der Kommandozeile zu löschen, ist das Kommando ipcrm

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

- **POSIX**-Speichersegmente und **POSIX**-Semaphoren können im Verzeichnis /dev/shm untersucht und von Hand gelöscht werden
- POSIX-Nachrichtenwarteschlangen im Verzeichnis /dev/mqueue untersucht und von Hand gelöscht werden