Fakultät für Informatik

MNM TEAM
Munich Network Management Team

Institut für Informatik

LMU

Systempraktikum im Wintersemester 2009/2010 (LMU):
Vorlesung vom 26.11. – Foliensatz 5

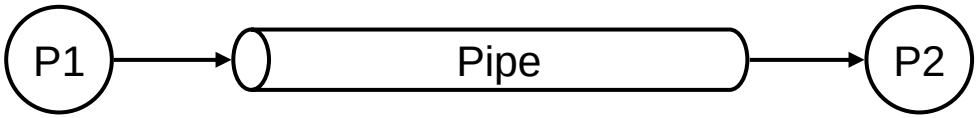
Speicherbasierte Kommunikation (T)
Realisierung von Semaphoren (T)
Shared Memory (P)
Synchronisation mittels Semaphoren (P)

Thomas Schaaf, Nils gentschen Felde

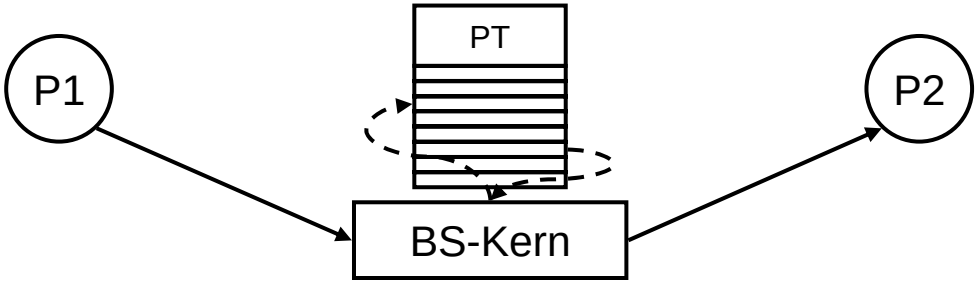
Interprozesskommunikation

MNM TEAM

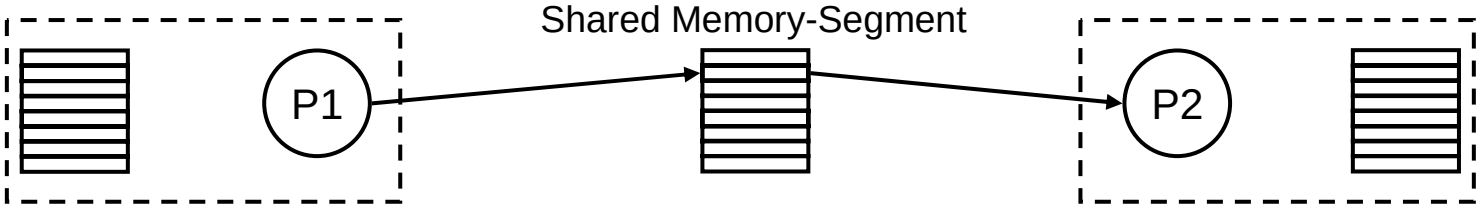
- Überblick: Interprozesskommunikation (stark abstrahiert)
  - Pipes und FIFOs:



  - Signale:



  - Speicherbasierte Kommunikation (Shared Memory-Konzept):



Heute

  - Netzbasierte Kommunikation (TCP/IP):

Speicherbasierte Kommunikation

MNM TEAM

- Shared Memory-Segment
  - Vom BS-Kern verwalteter Speicherbereich
  - Kann von mehreren Prozessen gelesen und beschrieben werden
  - Synchronisation der Zugriffe erforderlich

- Initiierung der speicherbasierten Kommunikation (Prinzip)
  - Prozess erzeugt Shared Memory-Segment, legt Größe und Name/Schlüssel fest
  - Erzeugtes Segment an den Adressraum des Erzeuger-Prozesses anhängen
  - Anderer Prozess kann das Segment mitbenutzen, wenn er den Schlüssel kennt
  - Synchronisation: Anlegen einer Semaphore zur Sicherstellung des konfliktfreien Zugriffs

Semaphoren

MNM TEAM

- Semaphore (Dijkstra, 1965)
  - Ist eine Datenstruktur
    - Zähler
    - Warteschlange
  - Art „Lock-Variable“
  - Zwei wesentliche Operationen (nach Dijkstra):
    - `wait(s)`:
      - Vor Eintritt in einen kritischen Bereich
      - auch down oder `acquire`
    - `signal(s)`:
      - Nach Verlassen des kritischen Bereichs
      - auch up oder `release`
- Arten von Semaphoren
  - Binärsemaphore
  - Zählsemaphore

Kritischer Bereich

MNM TEAM

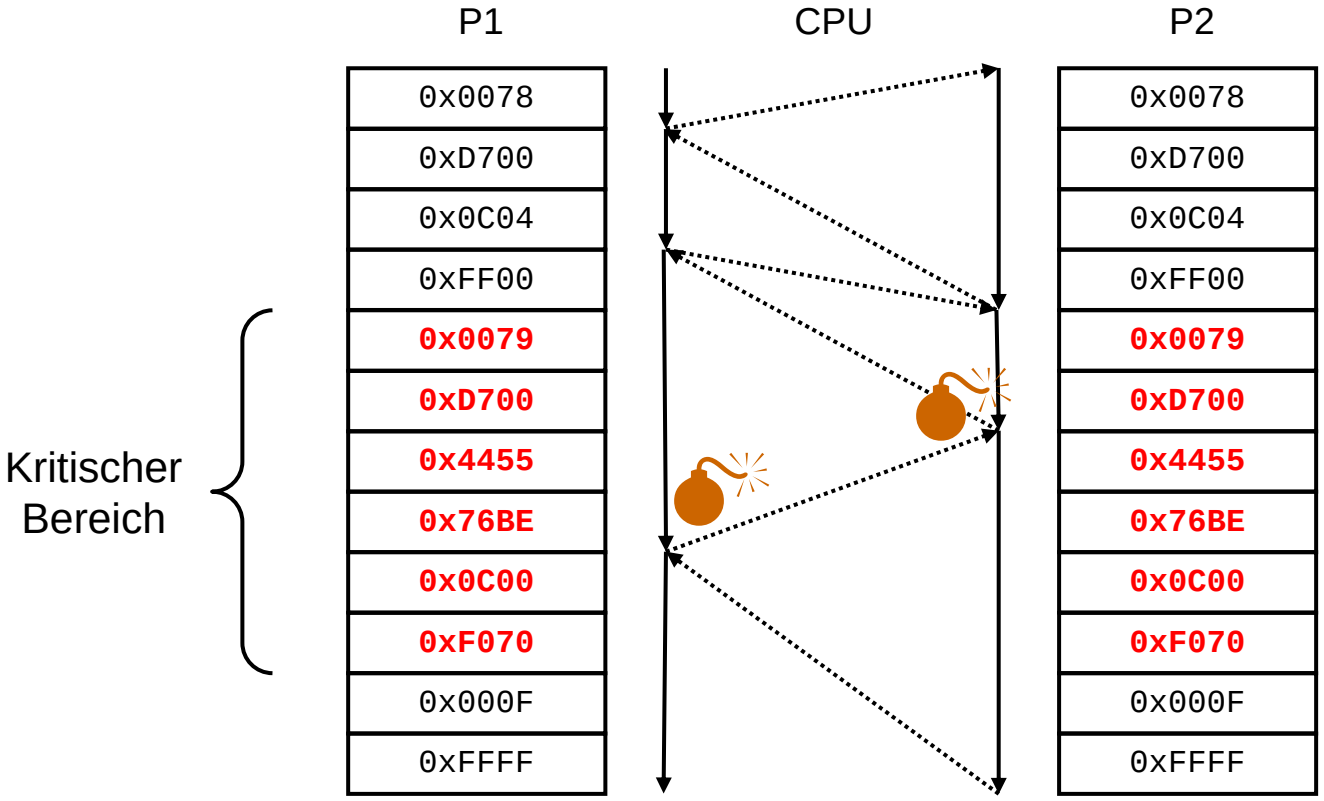
- Abschnitt in einem Programm, der nicht von mehreren Prozessen parallel/nebenläufig/unterbrechend ausgeführt werden darf

- Allgemeine Definition:
  Ein kritischer Bereich ist der Teil in einem Prozess, der auf gemeinsam genutzte Betriebsmittel zugreift.

Bsp. Kritischer Bereich

MNM TEAM

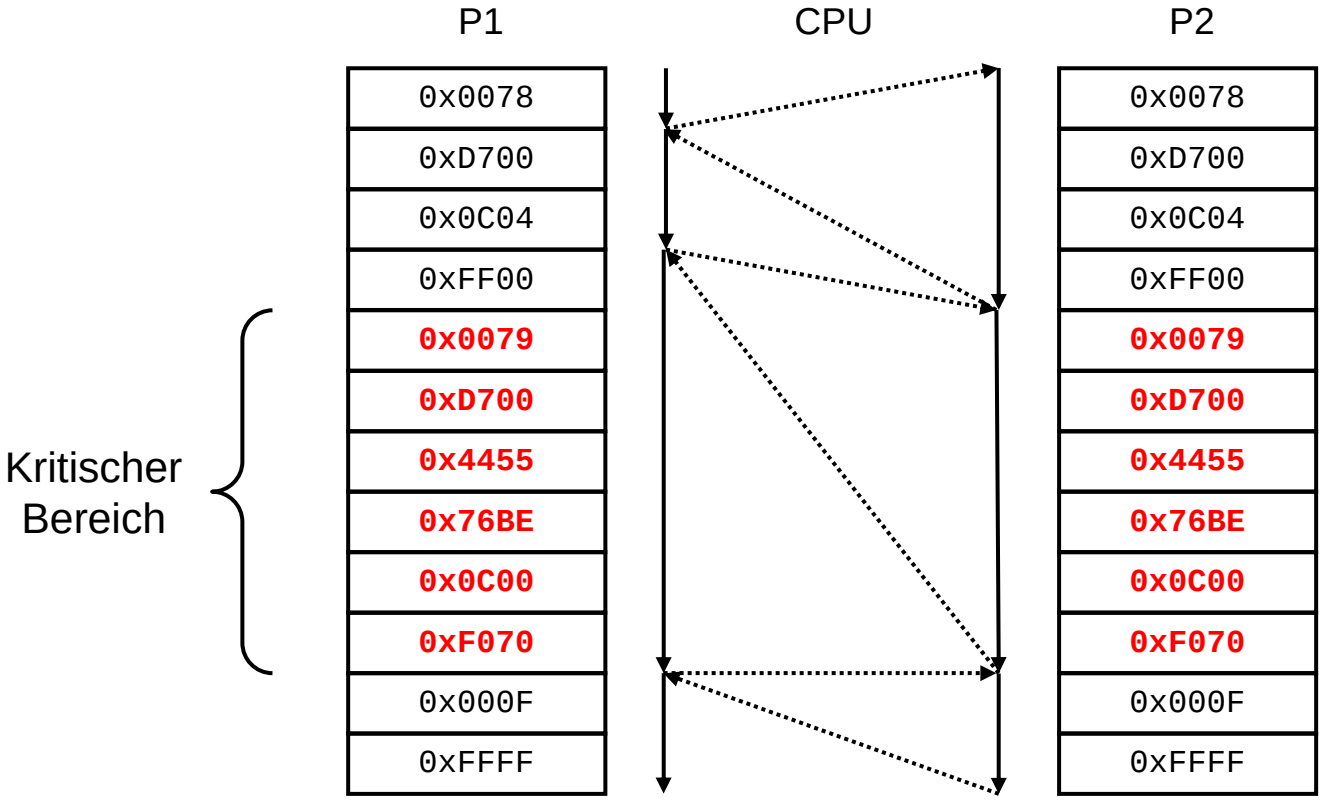
- Beispiel: Nebenläufige Programmausführung mit kritischen Bereichen – Race Condition/Deadlock möglich



Bsp. Kritischer Bereich

MNM TEAM

- Beispiel: Nebenläufige Programmausführung mit kritischen Bereichen – konfliktfrei



Binärsemaphore

MNM TEAM

- Realisierung in Pseudo-Code

```
DATATYPE binsemaphore = RECORD
                           value: (0, 1);
                           queue: LIST OF process;
                        END;

init(s:binsemaphore, init_value:INTEGER) {
   s.value := init_value;
}

wait(s:binsemaphore) {
   IF s.value = 1
   THEN s.value := 0;
   ELSE
      <Prozess in s.queue einfügen und blockieren>;
   ENDIF;
}

signal(s:binsemaphore) {
   IF <s.queue ist leer>
   THEN s.value := 1;
   ELSE
      <ersten Prozess aus s.queue aktivieren>;
   ENDIF;
}
```

Zählsemaphore

MNM TEAM

- Realisierung in Pseudo-Code

```
DATATYPE semaphore = RECORD
                        count: INTEGER;
                        queue: LIST OF process;
                     END;

init(s:semaphore, init_value:INTEGER) {
   s.value := init_value;
}

wait(s:semaphore) {
   s.count := s.count - 1;
   IF s.count < 0
   THEN
      <Prozess in s.queue einfügen und blockieren>;
   ENDIF;
}

signal(s:semaphore) {
   s.count := s.count + 1;
   IF s.count >= 0
   THEN
      <ersten Prozess aus s.queue aktivieren>;
   ENDIF;
}
```

Zusammenfassung: Semaphoren

MNM TEAM

- Herausforderungen
  - Funktionsdefinitionen zu `wait()` und `signal()` sind selbst kritische Bereiche
  - Sicherstellen, dass `wait()` und `signal()` logisch-atomar ausgeführt werden

- Synchronisation: Wichtige Begriffe
  - (Pseudo-)Parallelität, Nebenläufigkeit
  - Preemptives Scheduling
  - Race Condition
  - Kritischer Bereich/Abschnitt
  - Lock-Variable
  - Busy Waiting
  - Assoziierte FIFO-Queue
  - Logische Atomarität der Ausführung

Shared Memory

MNM TEAM

- System V-Interprozesskommunikation
  - Techniken/Objekte:
    - Shared Memory
    - Semaphore
    - Message Queues
  - Haupt-Charakteristik: IPC zwischen nicht-verwandten Prozessen
  - Informationen über SysV-Objekte: Kommando `ipcs`

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 1048578    user1      600        262144     1          dest
0x00000000 5439491    user1      664        199280     3

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 0          user1      640        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

Shared Memory

MNM TEAM

- Magic Numbers in SysV-Mechanismen
  - Idee: Nicht-verwandte Prozesse wollen SysV-Objekte (Semaphore, Shared Memory) gemeinsam nutzen bzw. darüber kommunizieren
  - Jedes SysV-Objekt erhält vom Kernel einen Identifier (ID)
  - Magic Number: Key, der mit dem jeweiligen Identifier assoziiert wird
  - C-Datentyp für Magic Numbers (Keys): `key_t` in <sys/types.h> (entspricht in der Größe dem Datentyp `long`)
  - Generieren eines Keys:
    - Als symbolische Konstante selbst definiert
    - Dynamisch/automatisch durch den Kernel

Shared Memory

MNM TEAM

- `int shmget(key_t key, int size, int flag)`
  - legt ein neues Segment an oder greift auf ein bestehendes Segment zu
    - `key`: Schlüssel (Magic Number) für das Segment (`IPC_PRIVATE`, um neuen Key automatisch generieren zu lassen)
    - `size`: Größe in Bytes
    - `flag`: Optionen, z.B.
      - `IPC_CREAT | 0644`: Anlegen eines neuen Segments mit `rw-r--r-` Zugriff
      - `0`: zum Zugriff auf ein vorhandenes Segment
    - Rückgabewert:
      - Identifikator des Segments bei Erfolg
      - `-1` im Fehlerfall

Shared Memory

MNM TEAM

- `void *shmat(int id, const void *addr, int flag)`
  - hängt ein Segment an den Adressraum des aufrufenden Prozesses an
    - `id`: Identifikator des Segments
    - `addr`: Adresse zur Einblendung des Segments in den Adressraum (0 für automatisch)
    - `flag`: Optionen, z.B.
      - 0 für les- und schreibbare Segmente
      - `SHM_RDONLY` für Read-only-Segmente
    - Rückgabewert: Pointer auf die Anfangsadresse im Adressraum des aufrufenden Prozesses
- `int shmdt(const void *addr)`: entfernt ein Segment aus dem Adressraum des aufrufenden Prozesses
  - `addr`: Adresse des Segments
  - Rückgabewert: 0 bei Erfolg, -1 im Fehlerfall

Shared Memory

MNM TEAM

- `int shmctl(int id, int cmd, struct shmid_ds *buf)`
  - führt Operationen (Steuerungsfunktionen) auf einem Segment durch
    - `id`: Identifikator des Segments
    - `cmd`: das auszuführende Kommando (kodiert als Integer)
    - `buf`: Parameter zur Ausführung des spezifizierten Kommandos
    - Rückgabewert:
      - 0 bei Erfolg
      - -1 im Fehlerfall

Shared Memory

MNM TEAM

- Beispiel: Shared Memory-Konzept
  - shm1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define MAXCOUNT 100000000
#define CHILDREN 4
#define SHMSEGSIZE sizeof(int)

int main() {
   int i, shmID, *shared_mem, count = 0;
   int pid[CHILDREN];

   shmID = shmget(IPC_PRIVATE, SHMSEGSIZE, IPC_CREAT | 0644);
   shared_mem = (int *)shmat(shmID, 0, 0);
   *shared_mem = 0;

   for(i=0; i<CHILDREN; i++) {
      if ((pid[i] = fork()) < 0) {
         perror("Fehler bei fork()");
         exit(-1);
      }
```

Shared Memory

MNM TEAM

- Beispiel: Shared Memory-Konzept (Forts.)
  - shm1.c (Forts.):

```
      if (pid[i] == 0) {
         while(*shared_mem < MAXCOUNT) {
            *shared_mem += 1;
            count++;
         }

         printf("Kind %i erhöht Wert um %i\n", i, count);
         shmdt(shared_mem);
         return EXIT_SUCCESS;
      }
   }

   for(i=0; i<CHILDREN; i++) {
      waitpid(pid[i], NULL, 0);
   }

   printf("Shared Memory = %i, MAXCOUNT = %i\n", *shared_mem, MAXCOUNT);
   shmdt(shared_mem);
   shmctl(shmID, IPC_RMID, 0);
   return EXIT_SUCCESS;
}
```

Shared Memory

MNM TEAM

- Beispiel: Shared Memory-Konzept (Forts.)
  - Ausführung und Ausgabe:

```
user@pcheger14:~/shm> ./shm1
Kind 2 erhöht Wert um 44429696
Kind 3 erhöht Wert um 48365022
Kind 0 erhöht Wert um 48143528
Kind 1 erhöht Wert um 42413770
Shared Memory = 100000001, MAXCOUNT = 100000000

user@pcheger14:~/shm> ./shm1
Kind 0 erhöht Wert um 48277514
Kind 1 erhöht Wert um 44488745
Kind 2 erhöht Wert um 44577803
Kind 3 erhöht Wert um 46605537
Shared Memory = 100000002, MAXCOUNT = 100000000
```

  - Frage: Was sind die Ursachen für die aufgetretenen Race Conditions?

    → bei Shared Memory: Synchronisation zwingend erforderlich!

Synchronisation mittels Semaphoren

MNM TEAM

- Das Semaphor-Konzept in Unix/C
  - Unix-Semaphore ist allgemeine Semaphore (Zählsemaphore)
  - Anlegen einer Gruppe von Semaphoren durch einen einzigen Aufruf
  - Für eine Gruppe von Semaphoren kann eine Operationsfolge definiert werden, die logisch-atomar ausgeführt wird
  - Die Struktur `struct sembuf`:
    - Definition:

```
struct sembuf {
   short sem_num;
   short sem_op;
   short sem_flg;
};
```

    - Erläuterungen:

      `sem_num`: Nummer des Semaphors in der Gruppe

      `sem_op`: auszuführende Operation (`-1` für `wait()`, `1` für `signal()`)

      `sem_flg`: Flags zur Steuerung der Operation

Synchronisation mittels Semaphoren

MNM TEAM

- `int semget(key_t key, int nsems, int flag)`
  - legt eine neue Semaphorengruppe an oder greift auf eine bestehende zu
    - `key`: Schlüssel (Magic Number) für die Gruppe (`IPC_PRIVATE`, um Key automatisch generieren zu lassen)
    - `nsems`: Anzahl der Semaphoren in der Gruppe
    - `flag`: Optionen, z.B.
      - `IPC_CREAT | 0644`: Anlegen einer neuen Gruppe mit `rw-r--r--` Zugriff
      - 0 zum Zugriff auf eine vorhandene Gruppe
    - Rückgabewert:
      - Identifikator der Gruppe bei Erfolg
      - `-1` im Fehlerfall

Synchronisation mittels Semaphoren

MNM TEAM

- `int semop(int id, struct sembuf *sops, int nsops)`
  - ändert die Werte von Semaphoren in einer Gruppe durch Ausführung von Semaphoroperationen (atomar in einem Block)
    - `id`: Identifikator der Semaphorengruppe (i.d.R. aus `semget()`)
    - `sops`: Pointer auf die Folge der auszuführenden Semaphoroperationen
    - `nsops`: Anzahl der Operationen in der Folge
    - Rückgabewert:
      - 0 bei Erfolg
      - -1 im Fehlerfall
- `int semctl(int id, int nsems, int cmd, union semun args)`
  - führt Operationen (Steuerungsfunktionen) auf einer Semaphorengruppe durch
    - `id`: Identifikator der Semaphorengruppe
    - `nsems`: Anzahl der Semaphoren in der Gruppe
    - `cmd`: das auszuführende Kommando (kodiert als Integer), z.B. `SETALL`, `GETALL`, `IPC_RMID`
    - `args`: Parameter zur Ausführung des Kommandos – abhängig vom jeweiligen Kommando
    - Rückgabewert:
      - 0 bei Erfolg
      - -1 im Fehlerfall

Synchronisation mittels Semaphoren

MNM TEAM

- Beispiel: Semaphore in C
  - semaphore.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define NSEMS 3
#define WAIT -1
#define SIGNAL 1

int main() {
   /*Semaphorengruppe mit NSEMS (=3) Semaphoren anlegen*/
   int id = semget(IPC_PRIVATE, NSEMS, IPC_CREAT | 0770);
   printf("Identifier der Gruppe: %i\n", id);

   /*Alle Semaphore der Gruppe initialisieren*/
   unsigned short init[NSEMS];
   int i;
   for(i=0; i<NSEMS; i++) {
      init[i] = 1;
   }
   semctl(id, NSEMS, SETALL, init);
```

Synchronisation mittels Semaphoren

MNM TEAM

- Beispiel: Semaphore in C (Forts.)
  - semaphore.c (Forts.):

```
   /*Werte der Semaphore abfragen*/
   unsigned short out[NSEMS];
   semctl(id, NSEMS, GETALL, out);
   printf("Werte des 1. und 2. Semaphors: %i, %i\n", out[0], out[1]);

   /*wait()/down()-Operation auf erstem Semaphor der Gruppe ausführen*/
   struct sembuf wait;
   wait.sem_num = 0;
   wait.sem_op = WAIT;
   wait.sem_flg = 0;

   semop(id, &wait, 1);

   /*Werte der Semaphore erneut abfragen*/
   semctl(id, NSEMS, GETALL, out);
   printf("Wert des 1. und 2. Semaphors: %i, %i\n", out[0], out[1]);

   /*Semaphorengruppe löschen*/
   semctl(id, NSEMS, IPC_RMID, 0);

   return EXIT_SUCCESS;
}
```

Zusammenfassung

MNM TEAM

- Interprozesskommunikation (IPC) über gemeinsamen Speicher
- Semaphore
- Kritische Bereiche
- Nebenläufige Programmausführung
- Realisierung von Binär- und Zählsemaphoren

- System V-Interprozesskommunikation
- Magic Numbers
- Systemfunktionen zum Programmieren mit Shared Memory
- Das System-V-Semaphorkonzept
- Systemfunktionen zum Programmieren mit Semaphoren