

Übungen zu Systemprogrammierung 1 (SP1)

Ü5 – Threads und Koordinierung

Christian Eichler, Andreas Ziegler, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 2017 – 19. bis 23. Juni 2017

http://www4.cs.fau.de/Lehre/SS17/V_SP1



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



ICPC - Was ist das?

- International Collegiate Programming Contest – veranstaltet von der ACM
- dabei sollen Teams aus **drei** Studenten innerhalb von **fünf** Stunden **neun bis elf** knifflige und originelle Programmieraufgaben lösen
- Problem: nur **ein** Computer steht zur Verfügung, aber kein Internet ☹
- dreistufiger Wettbewerb mit *Local Contest* in Erlangen, *Regional Contest* (irgendwo in Nordwesteuropa) und *World Finals* (irgendwo in der Welt)



Local Contest
(Erlangen)



Regional Contest
(Delft)



World Finals
(Orlando)



ICPC an der FAU

- am **Samstag, 01. Juli 2017** findet wieder ein FAU Local Contest statt
- von **11 bis 16 Uhr** im Informatikhochhaus
- teilnehmen darf jede/r Student/in der FAU, **Fachrichtung egal!**
- es wird jeweils **zu dritt** programmiert (**Einzelanmeldung** möglich)
- es wird außerdem eine **Practice Session** für alle Neulinge stattfinden, bei der (einfache) typische Probleme gezeigt und erklärt werden
 - Ort und Zeit werden noch bekannt gegeben
 - mehr **Infos/Anmeldung**: <http://icpc.cs.fau.de>

Wichtig: Anmeldung

Zur Teilnahme am Wettbewerb ist eine Anmeldung unter <http://icpc.cs.fau.de> unbedingt erforderlich. Deadline: **26.06.2017**.



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



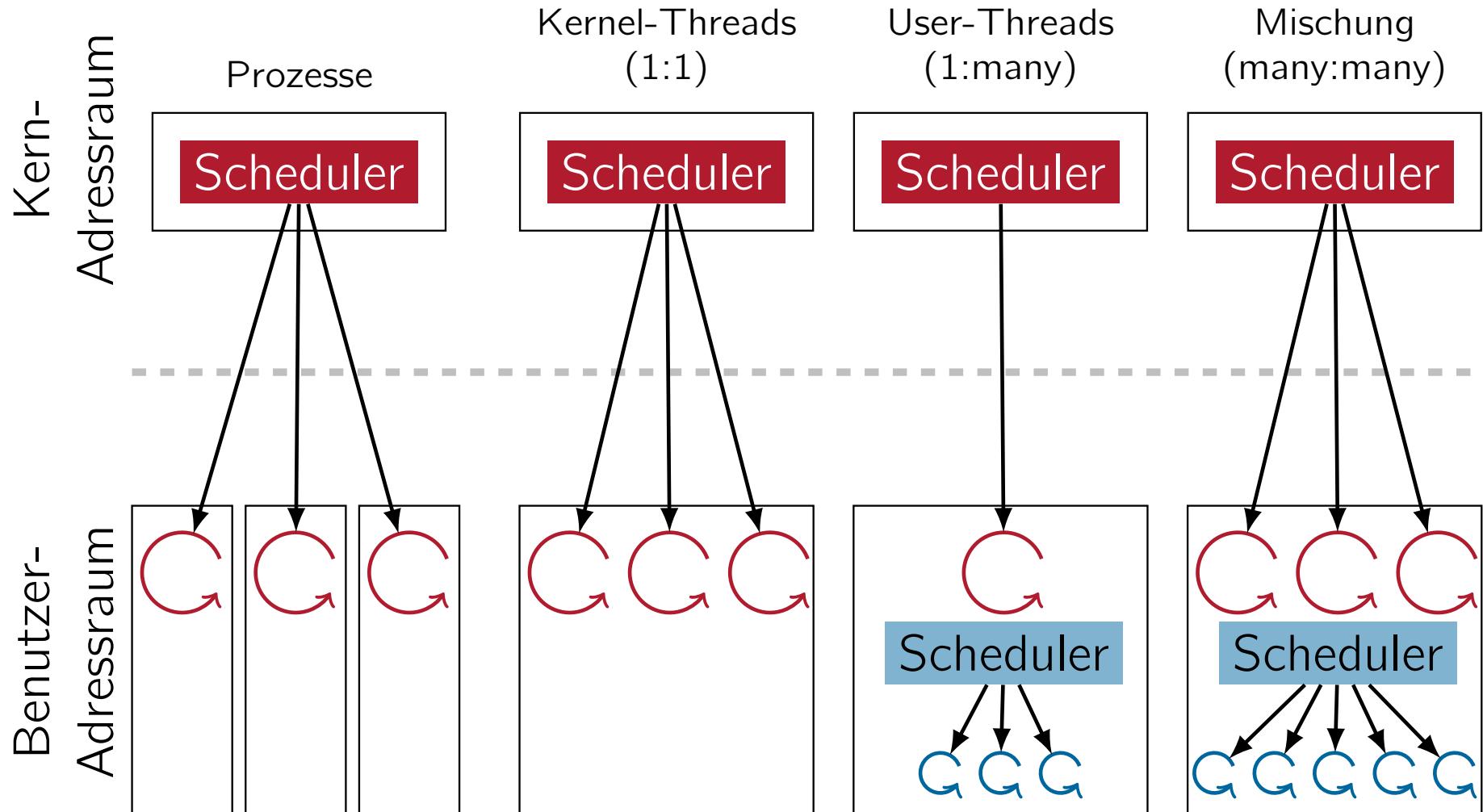
- UNIX-Prozesskonzept (Ausführungsumgebung mit einem Aktivitätsträger) für viele heutige Anwendungen unzureichend
 - keine parallelen Abläufe innerhalb eines logischen Adressraums auf Multiprozessorsystemen
 - typische UNIX-Server-Implementierungen benutzen die `fork`-Operation, um einen Server-Prozess für jeden Client zu erzeugen
 - Verbrauch unnötig vieler System-Ressourcen
 - zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adressraums nützlich
- Lösung: bei Bedarf weitere Aktivitätsträger in einem UNIX-Prozess erzeugen



- User-Threads (“federgewichtig”)
 - Realisierung auf Anwendungsebene
 - Systemkern sieht nur **einen** Kontrollfluss
 - + Erzeugung von Threads und Umschaltung extrem billig
 - Systemkern hat kein Wissen über diese Threads
 - in Multiprozessorsystemen keine parallelen Abläufe möglich
 - wird **ein** User-Thread blockiert, sind **alle** User-Threads blockiert
 - Scheduling zwischen den Threads schwierig
- Kernel-Threads (“leichtgewichtig”)
 - + Gruppe von Threads nutzt gemeinsam die Betriebsmittel eines Prozesses
 - + jeder Thread ist als eigener Aktivitätsträger dem Betriebssystemkern bekannt
 - Kosten für Erzeugung und Umschaltung zwar erheblich geringer als bei „schwergewichtigen“ Prozessen, aber erheblich teurer als bei User-Threads



Arten von Threads



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



■ Thread erzeugen

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine)(void *), void *arg);
```

- `thread` Thread-ID
- `attr` Modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.
- Nach der Erzeugung führt der Thread die Funktion `start_routine` mit Parameter `arg` aus
- Im Fehlerfall wird **errno nicht gesetzt**, aber ein Fehlercode als Ergebnis zurückgeliefert.
 - Um `perror(3)` verwenden zu können, muss der Rückgabewert erst in der **errno** gespeichert werden.

■ Eigene Thread-ID ermitteln

```
pthread_t pthread_self(void);
```

- Die Funktion kann nie fehlschlagen.



- Thread beenden (bei Rücksprung aus `start_routine` oder):

```
void pthread_exit(void *retval)
```

- Der Thread wird beendet und `retval` wird als Rückgabewert zurück geliefert (siehe `pthread_join(3)`)

- Auf Thread warten, Ressourcen freigeben und Rückgabewert abfragen:

```
int pthread_join(pthread_t thread, void **retval)
```

- Wartet auf den Thread mit der Thread-ID `thread` und liefert dessen Rückgabewert über `retval` zurück.

- Ressourcen automatisch bei Beendigung freigeben:

```
int pthread_detach(pthread_t thread)
```

- Die mit dem Thread `thread` verbundenen Systemressourcen werden bei dessen Beendigung automatisch freigegeben. Der Rückgabewert der Thread-Funktion kann nicht abgefragt werden.



Beispiel: Matrix-Vektor-Multiplikation

```
static double a[100][100], b[100], c[100];

int main(int argc, char* argv[]) {
    pthread_t tids[100];
    ...
    for (int i = 0; i < 100; i++)
        pthread_create(&tids[i], NULL, mult, (void *) i);
    for (int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
    ...
}

static void *mult(void *cp) {
    int i = (int) cp;
    double sum = 0;
    for (int j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return NULL;
}
```

■ Casts zwischen `int` und Zeiger (bei Parameterübergabe für `pthread_create()`) problematisch – **nicht zu Hause nachmachen!**



Parameterübergabe bei `pthread_create()`

- Generischer Ansatz mit Hilfe einer Struktur für die Argumente

```
struct param {  
    int index;  
};
```

- Für jeden Thread eine eigene Argumenten-Struktur anlegen
 - Speicher je nach Situation auf dem Heap oder dem Stack allozieren

```
int main(int argc, char* argv[]) {  
    pthread_t tids[100];  
    struct param args[100];  
  
    for (int i = 0; i < 100; i++) {  
        args[i].index = i;  
        pthread_create(&tids[i], NULL, mult, &args[i]);  
    }  
    for (int i = 0; i < 100; i++)  
        pthread_join(tids[i], NULL);  
    ...  
}
```



Parameterübergabe bei `pthread_create()`

```
static void* mult (void *arg) {
    struct param *par = (struct param *) arg;

    double sum = 0;
    for (int j = 0; j < 100; j++) {
        sum += a[par->index][j] * b[j];
    }
    c[par->index] = sum;
    return NULL;
}
```

- Zugriff auf den threadspezifischen Parametersatz über (gecasteten) Parameter (`void *arg → struct param *par`)



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



Koordinierung – Motivation

Was macht das Programm? Welches Problem kann auftreten?

```
static double a[100][100], sum;

int main(int argc, char* argv[]) {
    pthread_t tids[100];
    struct param args[100];

    for (int i = 0; i < 100; i++) {
        args[i].index = i;
        pthread_create(&tids[i], NULL, sumRow, &args[i]);
    }
    for (int i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
}

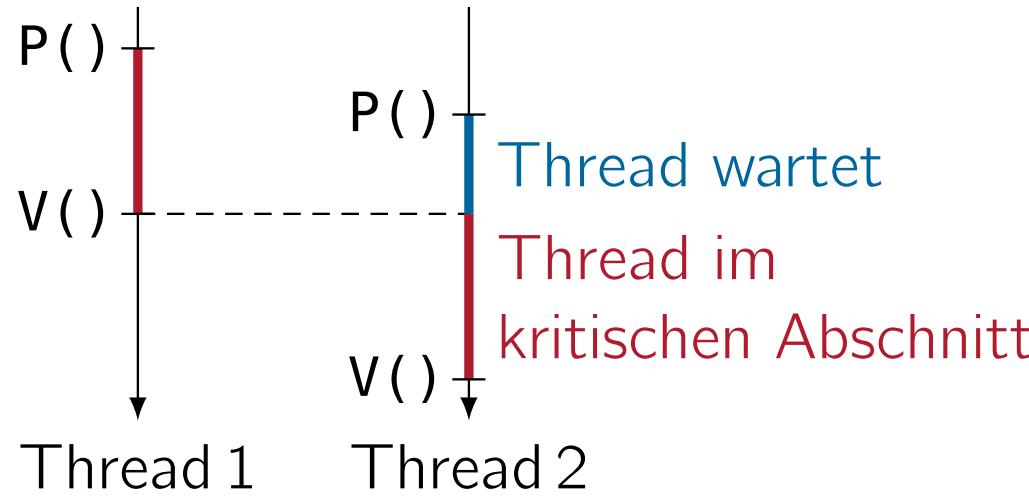
static void *sumRow(void *arg) {
    struct param *par = (struct param *) arg;
    double localSum = 0;
    for (int j = 0; j < 100; j++)
        localSum += a[par->index][j];
    sum += localSum;
    return NULL;
}
```



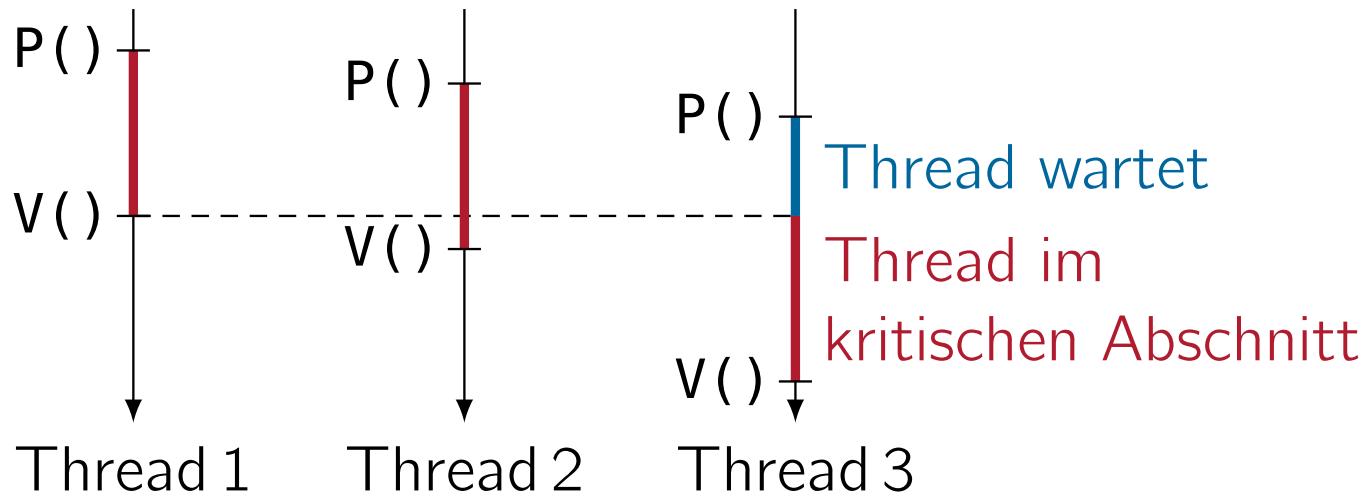
- Zur Koordinierung von Threads können Semaphore verwendet werden
- UNIX stellt zur Koordinierung von Prozessen komplexe Semaphor-Operationen zur Verfügung
 - Implementierung durch den Systemkern
 - komplexe Datenstrukturen, aufwändig zu programmieren
 - für die Koordinierung von Threads viel zu teuer
- Stattdessen Verwendung einer eigenen Semaphorimplementierung mit atomaren **P()**- und **V()**-Operationen
 - Datenstruktur mit (atomarer) Zählervariable
 - **P()** dekrementiert Zähler und blockiert Aufrufer, falls Zähler == 0
 - **V()** inkrementiert Zähler und weckt ggf. wartende Threads
 - Mehr Details: s. Vorlesung B | VI.1, Seite 22f.



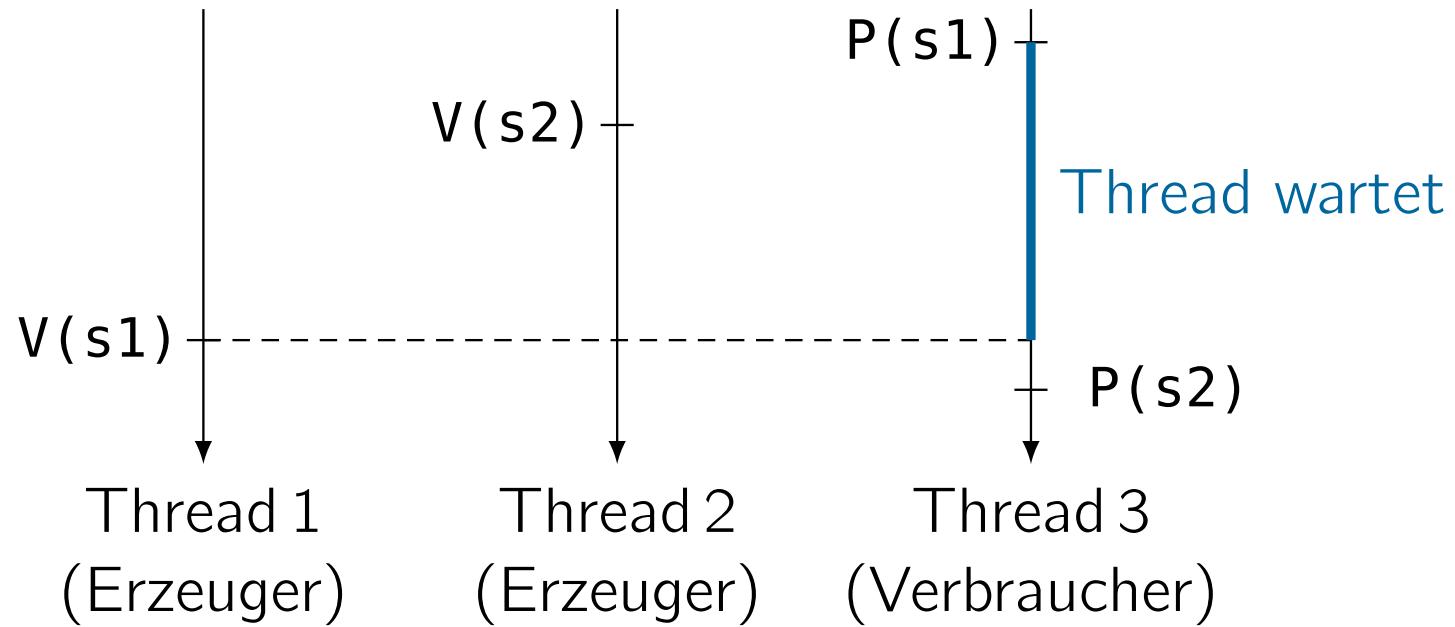
- Spezialfall des zählenden Semaphors: Binärer Semaphore
 - Initialisierung des Semaphors mit 1
- Beispiel: Schreibender Zugriff auf ein gemeinsames Datum



- Verwendung eines zählenden Semaphors
- Beispiel: Nur zwei aktive Threads gleichzeitig gewünscht
 - Initialisierung des Semaphors mit 2



- Benachrichtigung eines anderen Threads über ein Ereignis
- Beispiel: Bereitstellen von Zwischenergebnissen



- Semaphor erzeugen

```
SEM* semCreate (int initVal)
```

- P/V-Operationen

```
void P (SEM *sem)  
void V (SEM *sem)
```

- Semaphor zerstören

```
void semDestroy (SEM *sem)
```

- Semaphor-Modul und zugehörige Headerdatei befinden sich im pub-Verzeichnis.



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



- Ermöglicht lokale, unidirektionale Interprozesskommunikation
- Öffnen zum Schreiben blockiert solange bis ein (anderer) Prozess die Pipe zum Lesen öffnet
- Spezieller Dateityp
 - Erzeugung mit dem Shell-Kommando `mkfifo(1)`
 - Beispiel: Erzeugung einer UNIX-Pipe mit dem Namen `meinePipe`

```
user@host:~$ mkfifo meinePipe
```
- Öffnen der Pipe im Programm: `fopen(3)`
`FILE *fopen(const char *path, const char *mode);`
 - Öffnet Datei, die sich am Pfad `path` befindet
 - `mode` gibt Schreib-/Lese-Berechtigung an. Für diese Aufgabe: `"w"`
 - Zurückgegebener `FILE *` kann wie `stderr/stdout` genutzt werden z.B. `fputs(data, returned_pointer)`



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



- Ziele der Aufgabe
 - Programmieren mit der `pthread`-Bibliothek
 - Erkennen von Nebenläufigkeitsproblemen
 - Einsatz geeigneter Koordinierungsmaßnahmen
- Funktionsweise der `piper`
 - Bereitstellen der Eingaben von `stdin` auf mehreren, bereits vorhandenen UNIX-Pipes (*named pipes*)
 - Anlegen mit `mkfifo` notwendig
 - Pipes werden u.U. erst nach dem Start der `piper` von anderen Prozessen zum Lesen geöffnet
 - Zum Testen kann das Programm `cat` verwendet werden
- Zur Verwendung der Pthreads-Bibliothek ist die gcc-Option `-pthread` beim Übersetzen und Linken notwendig



- Haupt-Thread (`main()`)
 - Startet für jede als Kommandozeilenargument übergebene Pipe einen eigenen Schreib-Thread
 - Liest wiederholt eine Zeile von der Standardeingabe ein und wartet bis diese von den Schreib-Threads in die Pipes geschrieben wurden
- Schreib-Thread
 - Öffnet die als Parameter übergebene Pipe mit `fopen(3)`
 - Meldet sich mit Hilfe des `sbuf`-Moduls beim Haupt-Thread an
 - Wartet auf Zeilen vom Haupt-Thread und schreibt diese in seine Pipe



- Stellt Funktionen zum nebenläufigen Zugriff auf internen Puffer bereit
 - Synchronisation innerhalb des Moduls erforderlich
- Schnittstelle
 - `SBUF* sbufCreate(int maxNumberOfSems)`
Erzeugt einen neuen Puffer mit Platz für `maxNumberOfSems` Semaphore
 - `int sbuAddSem(SBUF* cl, SEM* sem)`
Speichert Semaphor in Puffer und gibt dessen Index im Puffer als ID zurück
 - `int sbuGetNumberOfSems(SBUF* cl)`
Gibt Anzahl der Semaphore im Puffer zurück
 - `SEM* sbuGetSem(SBUF* cl, int index)`
Gibt Semaphor mit der ID `index` zurück
- **Hinweis:** keine `semDestroy`-Funktionalität ⇒ In dieser Aufgabe Speicherlecks, die durch dieses Modul entstehen, erlaubt
 - Fehlerbehandlung ist trotzdem notwendig!



- 6.1 ICPC
- 6.2 Threads
- 6.3 Schnittstelle
- 6.4 Koordinierung
- 6.5 UNIX-Pipes
- 6.6 Aufgabe 5: piper
- 6.7 Gelerntes anwenden



„Aufgabenstellung“

- Beispiel von Folie 6–16 mit Hilfe eines Semaphors korrekt synchronisieren

