

Lernzettel

Synchronisation und Kommunikation
nebenläufiger Prozesse: Mutex, Semaphoren,
Condition Variables, Pipes, Message Passing

Universität:	Technische Universität Berlin
Kurs/Modul:	Systemprogrammierung
Erstellungsdatum:	September 6, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Systemprogrammierung

Lernzettel: Synchronisation und Kommunikation nebenläufiger Prozesse: Mutex, Semaphoren, Condition Variables, Pipes, Message Passing

(1) **Grundidee und Ziele.** Nebenläufige Prozesse benötigen Mechanismen zur Koordination und Kommunikation, damit Zugriffe auf gemeinsame Ressourcen konsistent bleiben und Ablaufreihenfolgen vorhersagbar sind.

(2) **Mutex (Mutual Exclusion).** Ein Mutex schützt einen kritischen Bereich, sodass immer nur eine Entität ihn betreten darf.

- Eigenschaften:
 - Exklusive Ausführung eines Codesegments.
 - Verhindert Race Conditions.
 - Gefahr von Deadlocks, falls mehrere Ressourcen in falscher Reihenfolge gehalten werden.
- Grundoperationen: `lock(m)` und `unlock(m)`.
- Grundregel: Vor dem Eintritt in den kritischen Bereich `lock(m)`, danach `unlock(m)`.

$\text{lock}(m) \Rightarrow$ kritischer Bereich abgeschlossen durch m
 $\text{unlock}(m) \Rightarrow$ freigibt den Bereich

(3) **Semaphore.** Semaphoren modellieren Ressourcenanzahl. Es gibt zählende (counting) und binäre Semaphoren.

- Operationen:
 - **P(S)** (auch *wait* oder *down*): Falls $S > 0$, $S \leftarrow S - 1$; sonst Blockieren.
 - **V(S)** (auch *signal* oder *up*): $S \leftarrow S + 1$; wartende Prozesse werden ggf. aufgeweckt.
- Beispiel: Producer-Consumer mit zwei Semaphoren
 - **empty** zählt freie Slots, N initial.
 - **full** zählt belegte Slots, 0 initial.

$P(S) : \text{ falls } S > 0 \Rightarrow S \leftarrow S - 1, \text{ sonst blockieren}$

$V(S) : S \leftarrow S + 1$

Hinweis: Binäre Semaphoren ($S \in \{0, 1\}$) verhalten sich wie Mutexes, bieten jedoch keine automatische Mutual Exclusion bei mehreren Ressourcen.

(4) Condition Variables (CV). CVs ermöglichen Threads das Warten auf eine Prädikatsbedingung unter Schutz durch einen Mutex.

- Grundoperationen: `wait(cv, mutex)`, `signal(cv)`, `broadcast(cv)`.
- Ablauf:
 - `wait` setzt den Mutex frei, wartet, bis die Bedingung erfüllt ist, und reacquires den Mutex.
 - `signal` weckt einen wartenden Thread auf.
 - `broadcast` weckt alle wartenden Threads auf.

`wait(cv, m)` verbraucht m , bis Bedingung gilt

`signal(cv)` weckt einen wartenden Thread

(5) Pipes (Interprozesskommunikation). Pipes ermöglichen unidirektionale Kommunikation zwischen Prozessen bzw. Threads.

- Unnamed pipes (anonym): zwei Enden, `fd[0]` zum Lesen, `fd[1]` zum Schreiben.
- Named pipes (FIFO): Datei im Dateisystem, `mkfifo` erzeugt eine Kommunikationsschnittstelle.
- Typische Nutzung:
 - `pipe(fd);`
`write(fd[1], data, n);`
`read(fd[0], buffer, n);`

`read` und `write` blockieren, falls keine/zu wenig Daten verfügbar sind

(6) Message Passing. Nachrichtenbasiertes Modellierung der Kommunikation zwischen Prozessen oder Threads.

- Direktes Message Passing: Sender sendet eine Nachricht an einen konkreten Empfänger.

- Synchronous vs. Asynchronous:

$\text{Send}(m) \Rightarrow \text{Receive}()$ (synchron)

$\text{Send}(m) \Rightarrow \text{Nachricht wird in Queue abgelegt}$ (asynchron)

- Typische Implementierungen: Postfächer, Kanäle, Message Queues.

$\text{Receive}(m)$ blockiert, bis eine Nachricht vorhanden ist

$\text{Send}(m)$ blockiert ggf. bis der Empfänger bereit ist

(7) Wahl der Mechanismen (Grobempfehlungen).

- Mutex: kurze kritische Abschnitte, hoher Durchsatz.
- Semaphoren: Ressourcenmanagement (Pools, Zähler), einfache Synchronisation.
- Condition Variables: komplexe Wartebedingungen, höhere Flexibilität.
- Pipes: einfache, lineare IPC, geeignet für Flüsse von Daten.
- Message Passing: lose Kopplung, Skalierung, verteilte Systeme.

(8) Kurze Beispielgedanken. - Producer-Consumer: Nutzen von **empty** und **full** Semaphoren zusammen mit einem Mutex zum Schutz des Puffers. - Warteschlangen mit CV: Wenn der Puffer leer ist, warten; bei Einfügen eines Elements, Signalisierung der wartenden Verbraucher.

(9) Sicherheit und Nachhaltigkeit. Sichere Synchronisation minimiert Datenverlust (Konsistenzverletzungen) und Energieverbrauch durch effizientere Ablaufsteuerung. Berücksichtige Verwechslungsrisiken, Deadlocks und Priority Inversion bei der Auswahl von Synchronisationsmechanismen.

(10) Übungsaufgaben (Kurzskizzen). - Implementiere in C einen Mutex-geschützten Zähler, der von zwei Threads inkrementiert wird. - Skizziere eine Producer-Consumer-Lösung mit Semaphoren und Mutex. - Beschreibe, wie eine Pipes-Kommunikation zwischen zwei Prozessen aufgebaut wird (Unkodiert, Pseudocode). - Erkläre den Unterschied zwischen einer Condition Variable und einer Semaphore in einem konkreten Szenario.