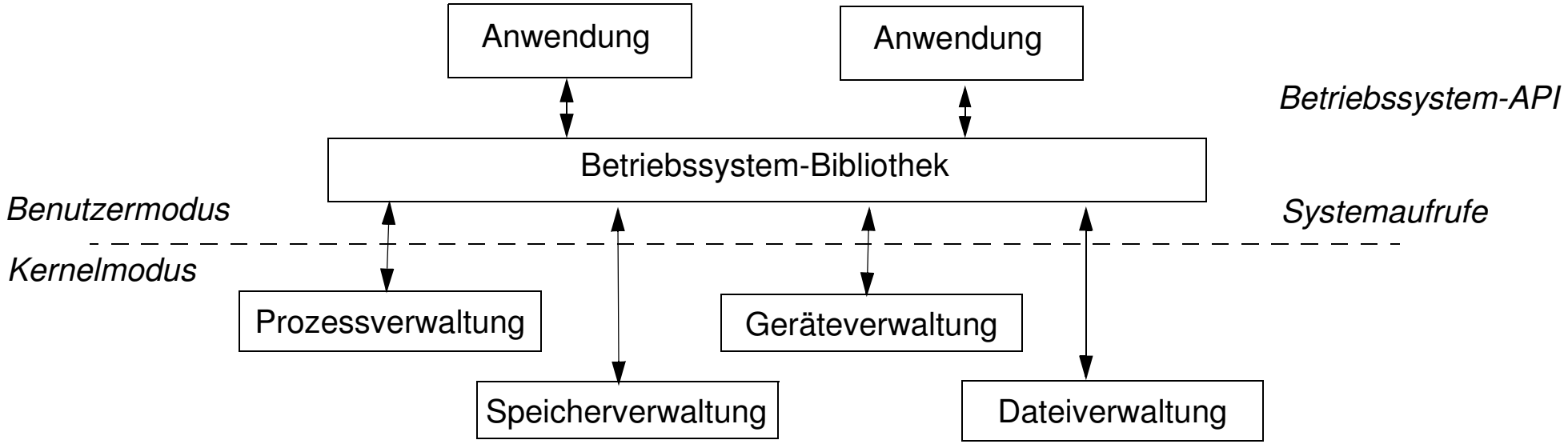
2 Systemaufrufe

Systemaufrufen (System Calls, syscalls) stellen die Schnittstelle Betriebssystembibliothek und Kern dar, d.h. es handelt sich um eine **Kommunikation über die User-Space/Kernel-Space Grenze hinweg**, die vom User-Space ausgelöst wird. Systemaufrufe werden für sämtliche Aktionen, die vom Betriebssystem verwaltete Ressourcen betreffen, benötigt.



Überlegung: Kann eine Anwendung auf Systemaufrufe verzichten?

Da alle Ressourcen (auch Ein-/Ausgabe) vom Betriebssystem verwaltet werden, kann eine Anwendung, die keine Systemaufrufe verwendet (d.h. auch nicht über Bibliotheken)

- nur in dem ihm zugewiesenen Speicher arbeiten
- keine Daten einlesen (z.B. von Dateien, Tastatur oder Netz)
- keine Daten ausgeben
- nicht mit anderen Programmen oder dem Betriebssystem kommunizieren.

Eine Anwendung, die keine Systemaufrufe verwendet kann nur interne Berechnungen durchführen, diese aber nicht ausgeben, d.h. eine solche Anwendung wäre nutzlos.

Verwenden von Systemaufrufen

Systemaufrufe erscheinen vom aufrufenden Programm wie ein Unterfunktionsaufruf. Die Unterfunktion wird von der Betriebssystem-Bibliothek implementiert, die dann den Systemaufruf auslöst.

Beispiel: Betriebssystem-Funktion/Systemaufruf `read`

Syntax

```
#include <unistd.h>
   ssize_t read(int fd, void *buf, size_t count);
```

Übergabeparameter

- `fd` file descriptor der zuvor geöffneten Datei
- `*buf` Pointer auf den Puffer, in den die gelesenen Daten geschrieben werden
- `count` max. Anzahl zu lesender Bytes

Rückgabewert

Anzahl der gelesenen Bytes, negative Werte bei Fehler

Blockierungen: Der Unterschied zu normalen Unterfunktionen

im Gegensatz zu normalen Bibliotheksfunktionen, können Systemaufrufe **blockieren**

Effekt der Blockierung

Die Anwendung wird angehalten und erst wieder dem Prozessor zugeteilt, wenn die per Systemaufruf angeforderten Daten zurückgegeben werden können.

Grund für Blockierung

Systemaufrufe betreffen oft die Ein-/Ausgabe, d.h. es muss gewartet werden, bis entsprechende Daten vorhanden sind oder weggeschrieben wurden

Dauer der Blockierung

- oft nach oben beschränkt, z.B. durch Zugriffszeit auf Festplatte
- zum Teil nicht vorhersehbar, z.B. beim Lesen von Tastatureingaben
- teilweise wird die Blockierung nie aufgehoben, z.B. beim Warten auf eingehende Netzverbindungen
- oft per timeout-parameter auf ein Maximum begrenzbar

Blockierungen sind ein Mechanismus, der **die Anwendungsprogrammierung vereinfacht** und CPU-Zeit einspart. Ohne Blockierungen müssten Anwendungen regelmäßig prüfen, ob neue Daten vorhanden sind, anstatt das Ende einer Blockierung abzuwarten.

Beispiele für Systemaufrufe

Aufruf	Beschreibung
s=execve(name, argv, environmentp)	Ausführen eines neuen Programs im aktuellen Prozess
exit(status)	Prozess beenden
s = kill(pid, signal)	Signal zu Prozess senden (z.B. zum Beenden)
fd=open(file, mode, ...)	Datei öffnen
s=close(fd)	Datei schließen
n=read(fd, buffer, nbytes)	Daten aus Datei in Puffer lesen
n=write(fs, buffer, nbytes)	Daten aus Puffer in Datei schreiben
position = lseek(fd, offset, whence)	Dateizeiger verschieben
s = stat(name, &buf)	Dateinformationen abfragen
s = mkdir(name, mode)	Verzeichnis erzeugen
s = rmdir(name)	Verzeichnis löschen
seconds = time(&seconds)	Aktuelle Zeit in Sekunden seit 1.1.1970

6.1 Prozessbegriff

Rechner können eine **Vielzahl von Aufgaben** durchführen, indem sie für jede Aufgabe einen **Prozess** ausführen, d. h. mehrere Programme auf einem oder mehreren Prozessoren ausführen. Hierbei kommt es oftmals vor, dass zeitweise **mehr Prozesse** ablaufbereit sind als **Prozessoren** zur Verfügung stehen. Eine zentrale Funktion des Betriebssystems ist es deshalb, Prozesse zu **verwalten**, die vorhandenen Rechenressourcen auf die Prozesse **aufzuteilen** und zu entscheiden, wann welche Prozesse auf welchem Prozessor ausgeführt werden sollen.

1 Begriffsdefinitionen

2 Prozess-Lebenszyklus

3 Threads

1 Begriffsdefinitionen

Task

Ein *Task* ist eine *abgeschlossene Teilaufgabe*, welche durch einen Rechner bearbeitet werden kann.

Wird eine Aufgabe in mehrere Tasks zerlegt, so kann es zwischen den Tasks **Abhängigkeiten** in der Form geben, dass ein Task erst dann bearbeitet werden kann, wenn bestimmte andere Tasks bereits abgeschlossen wurden. Die Bearbeitung eines Task kann dann aber **unabhängig** vom Bearbeitungszustand anderer Tasks erfolgen.

Prozess

Ein *Prozess* in einem Rechnersystem ist ein **Programm** (Folge von Maschinencode-Operationen) **in Ausführung**, welches einen Task ausführt. Das Prozesskonzept umfasst zwei Aspekte:

- **Ausführung**
  Die Ausführung des Programms erfolgt durch die sequenzielle Ausführung der Operationen des Programms durch einen Prozessor. Der Status der Ausführung wird im *Prozesskontext* vermerkt.
- **Gruppierung von Ressourcen**
  Prozesse benötigen zur Ausführung der Aufgabe Systemressourcen wie z. B. eine CPU und Speicherbereiche für Programmcode und Daten sowie Zugriff auf Dateien. Die Ressourcen werden dem Prozess vom Betriebssystem zugeteilt und im *Prozesskontext* aufgeführt.

Prozesskontext

Der Begriff *Prozesskontext* bezeichnet die Menge aller Informationen, die den **Status** der Programmausführung sowie die **Ressourcen** eines Prozesses beschreiben und für die Ausführung oder Verwaltung erforderlich sind. Er umfasst insbesondere:

- Status
  - Inhalt von Prozessor-Registern (z. B. Befehlsfolgezähler, Statuswort)
  - Prozesszustand (z. B. Running, Ready, Blocked)
- Ressourcen
  - Hauptspeicherbereiche (Virtueller Adressraum)
    - Text-Bereich (Programmcode)
    - Datenbereich (globale Variablen)
    - Stack (lokale Variablen, Rücksprungadressen von Funktionsaufrufen)
- Speicherverwaltungsinformationen (z. B. Seitentabelle für virtuelle Speicherverwaltung)
- Ressourcen zur Ein-/Ausgabe (z. B. I/O-Geräte, geöffnete Dateien, Arbeitsverzeichnis, Wurzelverzeichnis)
- Referenzen auf verwandte Prozesse (Child-Prozesse, Parent-Prozess)
- Signale und Signal-Handler

- Verwaltungsinformationen
  - Prozess-ID
  - Sicherheitsattribute (z. B. User-ID, Group-ID, Berechtigungen)
  - Accounting-Informationen (z. B. bisher verbrauchte Prozessor-Zeit)

Hinweis: Die Menge der Attribute des Prozesskontexts kann in verschiedenen Betriebssystemimplementierungen voneinander abweichen.

Multiprogrammbetrieb („Multitasking“)

Im *Multiprogrammbetrieb* können *mehrere Tasks* **(quasi-)simultan** auf einem Rechner abgearbeitet werden. Sind weniger Prozessoren als Tasks verfügbar, dann verteilt das Betriebssystem die verfügbaren Rechenressourcen auf die Tasks, indem es die Tasks nacheinander oder **abwechselnd** den verfügbaren Prozessoren zuteilt.

Multiprocessing

Der Begriff *Multiprocessing* bezeichnet die Nutzung von **mehreren Prozessoren** in einem Rechnersystem.

Nebenläufige Verarbeitung („Concurrent Processing“)

Bei *nebenläufigern Verarbeitung* werden **mehrere verschiedene** Tasks gleichzeitig auf mehreren Prozessoren ausgeführt.

Parallelverarbeitung („Parallel Processing“)

Bei *Parallelverarbeitung* wird **ein** Task aufgeteilt und die Teiltasks auf mehreren Prozessoren gleichzeitig ausgeführt.

Hinweis: Der Begriff *Parallelverarbeitung* wird oft auch für *nebenläufige Verarbeitung* verwendet.

2 Prozess-Lebenszyklus

2.1 Prozesserzeugung

Beim Start eines Betriebssystems wird mindestens ein initialer Prozess erzeugt. Prozesse können durch **Systemaufrufe** die Erzeugung von neuen Prozessen anfordern. Die Erzeugung des Prozesses wird vom Betriebssystem durchgeführt. Die **Adressräume** der Prozesse sind **getrennt**.

Bei der Erzeugung eines neuen Prozesses wird der ursprüngliche Prozess als *Parent-Prozess* und der neu erzeugte als *Child-Prozess* bezeichnet.

Beispiel: `fork` und `execve` in Linux-Systemen

> Der Systemaufruf `fork` erzeugt einen neuen Prozess, welcher zunächst als **Kopie** des aufrufenden Prozesses angelegt wird. Über den **Rückgabewert** des Systemaufrufs kann ermittelt werden, welcher der Prozesse der Child-Prozess ist, so dass dieser anschließend **andere Aufgaben ausführen** kann als der Parent-Prozess. Mit dem Systemaufruf `execve` kann ein anderes Programm in den Speicherbereich geladen und anstelle des gegenwärtigen Programmcodes ausgeführt werden.

Beispiel-Codeausschnitt (C)

```
pid = fork();
if (0 == pid) {
   // child code
   // execute the programm „/bin/ls“ with argument „-l“
   execl("/bin/ls", "ls", "-l", (char *)0);
} else {
   // parent code
   ...
}
```

- Der Aufruf von `fork` liefert im Child-Prozess den Rückgabewert 0, im Parent-Prozess hingegen die Prozess-ID des Child-Prozesses.
- Die Routine `execl` führt nach einer Konvertierung der Parameter den **Systemaufruf** `execve` aus. `execl` erwartet als Argumente den Dateinamen des auszuführenden Programms sowie eine Liste der hierbei zu übergebenden Argumente (die Liste wird durch einen `NULL`-Zeiger abgeschlossen).

Hinweis: Anstelle von `fork` können auch verwandte Routinen wie `vfork` (kein Kopieren der Seitentabelle) oder `clone` (erlaubt verschiedene Ressourcen mit dem Parent-Prozess zu teilen) verwendet werden.

2.2 Prozesshierarchie

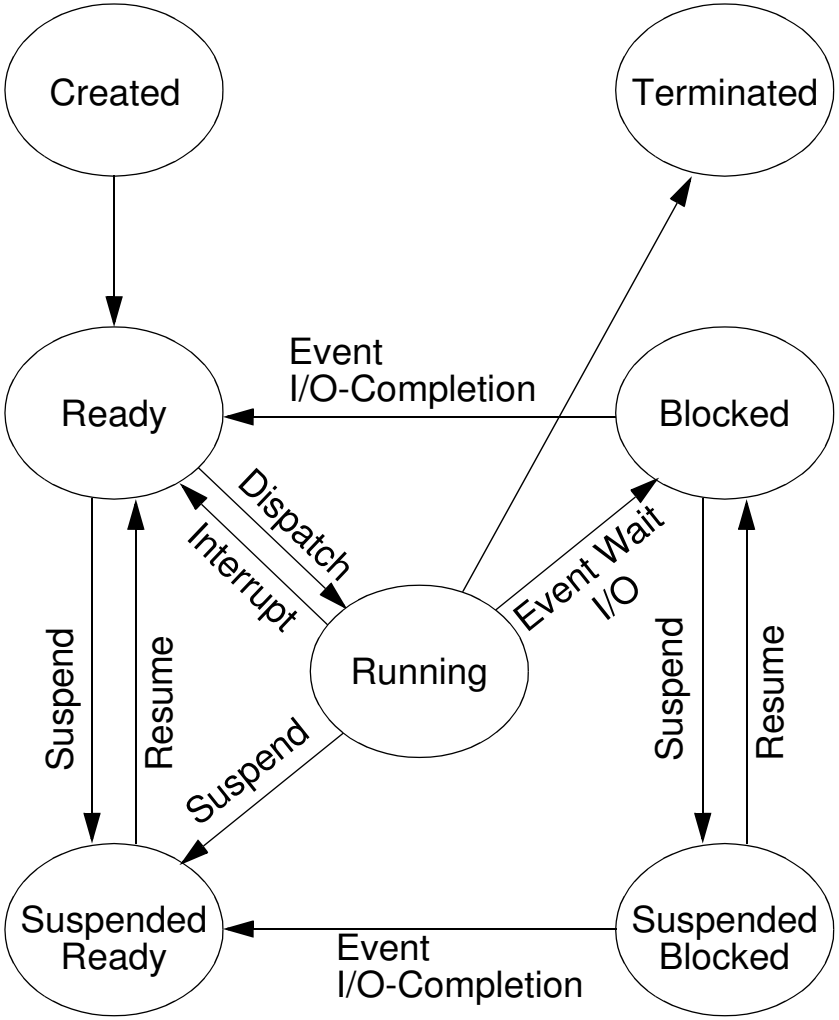
Nach der Erzeugung eines Prozesses besteht zwischen dem Parent- und dem Child-Prozess eine **Beziehung**, welche es dem Parent-Prozess ermöglicht, **Informationen** über den Child-Prozess zu erhalten (z. B. durch eine Benachrichtigung über das Terminieren des Child-Prozesses) und diesen zu **steuern** (z. B. über das Senden von Signalen). In der Regel hat jeder Prozess genau einen Parent-Prozess, so dass sich hieraus eine baumförmige **Hierarchie** von Prozessen ergibt.

2.3 Prozessterminierung

Ein Prozess kann über **Systemaufrufe** (z. B. `exit`) oder Signale (z. B. `SIGKILL`) sich selbst oder andere Prozess beenden. Treten **Fehler** bei der Programmausführung eines Prozesses auf (z. B. Division durch 0), dann kann der Prozess dadurch vorzeitig vom Betriebssystem beendet werden. Wird ein Prozess beendet, dann kann der zugehörige Parent-Prozess durch ein **Signal** über die Prozessterminierung benachrichtigt werden. Prozesse können ihrem Parent-Prozess durch einen **Statuswert** einen Hinweis auf die mögliche Ursache der Prozessterminierung geben. Child-Prozesse von terminierenden Prozessen werden i. d. R. **nicht beendet**.

2.4 Zustände des Prozesses

Im folgenden Zustandsdiagramm sind der **Zustandsraum** und die möglichen **Zustandsübergänge** dargestellt.



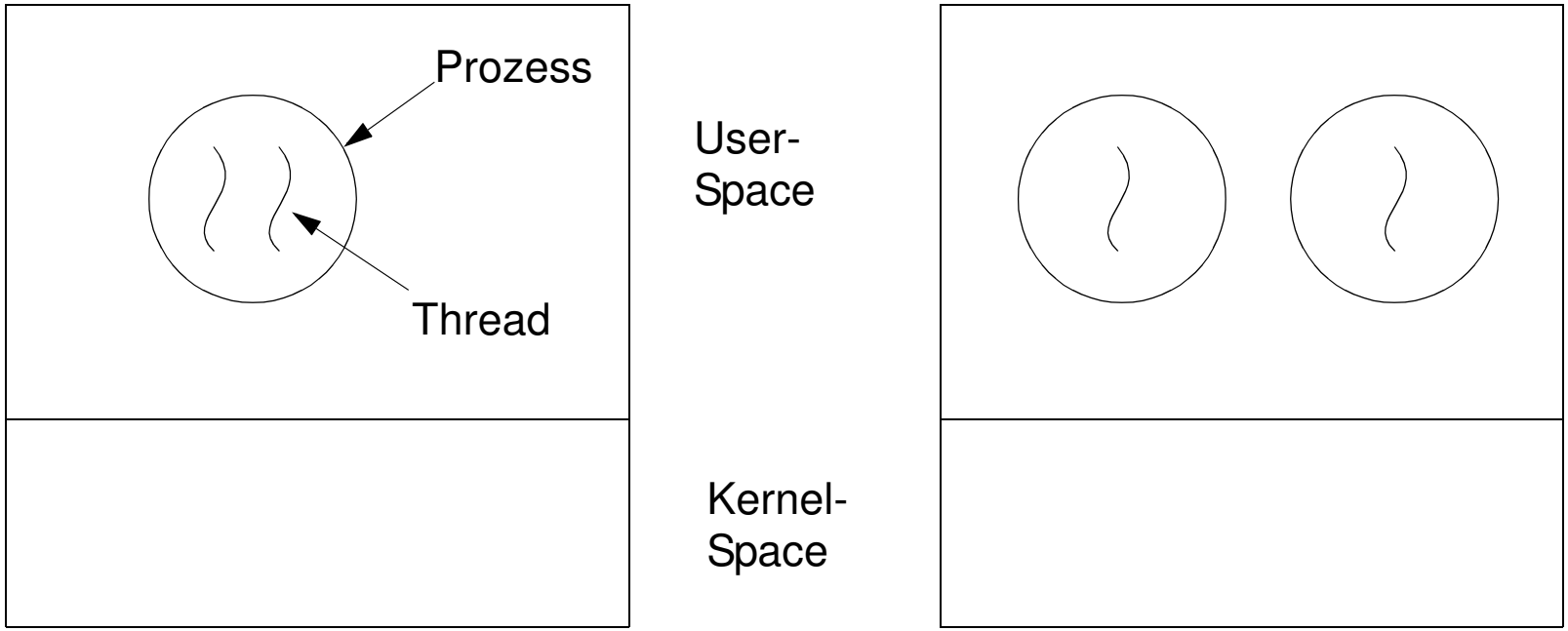
Die Zustände haben hierbei folgende Bedeutung:

Ready	Ablaufbereit; der Prozessor wurde noch nicht zugeteilt (auch als „Waiting“ bezeichnet)
Running	Ablaufend; der Prozessor wurde dem Prozess zugeteilt
Blocked	Blockiert; der Prozess wartet auf ein spezielles Ereignis, z. B. den Abschluss einer Ein-/Ausgabeoperation
Suspended	Suspendiert; der Prozess ist zeitweise deaktiviert, z. B. durch Verlagerung aus dem Hauptspeicher. Der Zustand kann weiter verfeinert werden hinsichtlich der Ablauffähigkeit (Ready, Blocked)
Created	Erzeugt
Terminated	Beendet

Hinweis: Es kann darüber hinaus noch weitere Prozesszustände geben (z. B. den „Zombie“-Zustand in UNIX-Systemen).

3 Threads

Zunächst wurde angenommen, dass die Ausführung eines Prozesses durch einen einzigen „Ausführungsstrang“ erfolgt. Oftmals ist es aber nützlich, **mehrere „Ausführungsstränge“** (sogenannte *Threads*) in **einem Prozesskontext** ausführen zu können, die (quasi-)simultan weitgehend unabhängig voneinander ablaufen, aber im **selben Speicherbereich** operieren und Zugriff auf alle Daten und Ressourcen des Prozesses haben.



ein Prozess mit zwei Threads

zwei Prozesse mit je einem Thread

Threads erleichtern die **Implementierung** von Programmen, in welchen mehrere Tasks ausgeführt werden sollen. Benötigen die Tasks Zugriff auf gemeinsame Daten, so ist eine Realisierung durch mehrere Prozesse nicht oder nur durch aufwändige **Interprozesskommunikation** möglich.

Ein weiterer Vorteil bei der Nutzung von Threads anstelle von Prozessen ist, dass der Wechsel zwischen zwei Threads eines Prozesses i. d. R. **weniger aufwendig** realisiert werden kann als der Wechsel zwischen zwei Prozessen, da nur ein Teil des Kontextes ausgetauscht werden muss und somit schneller erfolgen kann.

Beispiel

In einer Textverarbeitungsanwendung mit grafischer Oberfläche lassen sich beispielsweise die folgenden, weitgehend unabhängigen Tasks identifizieren:

- Verarbeitung der Benutzereingaben, z. B. die Eingabe von Text über die Tastatur, Mausklicks
- Formatierung des Dokuments, z. B. Setzen von Zeilenumbrüchen, Worttrennung, Platzierung von Grafiken
- regelmäßiges Speichern des Dokuments in einer Datei

Die Tasks sind weitgehend unabhängig voneinander, benötigen aber alle Zugriff auf die Daten des Dokuments. Die Implementierung vereinfacht sich erheblich, wenn die Tasks unabhängig voneinander jeweils als ein Thread realisiert werden und (quasi-)simultan ablaufen können.

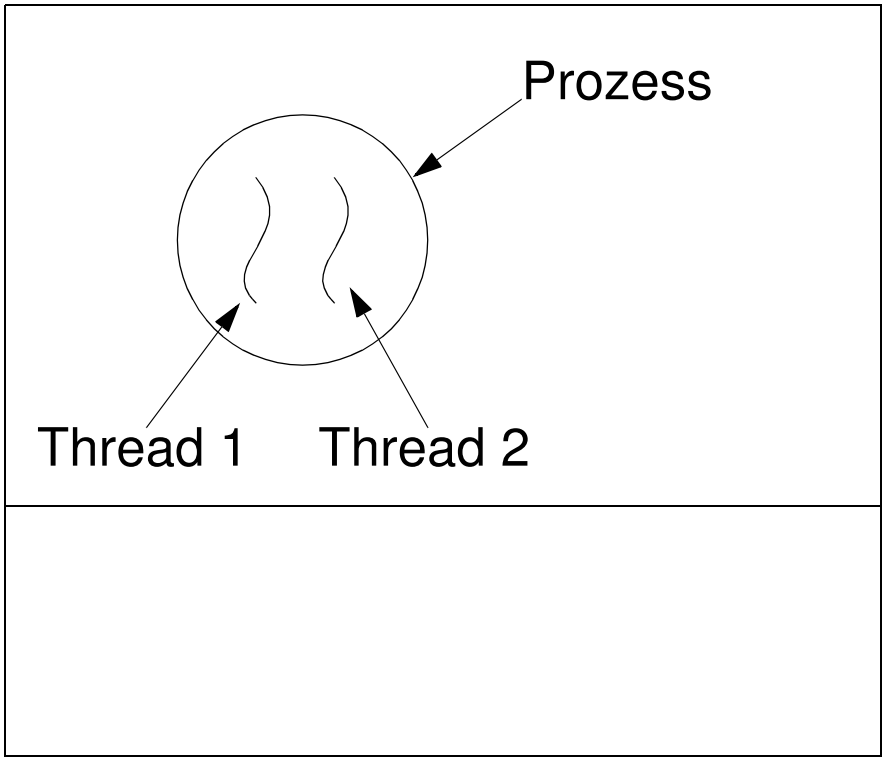
Begriffsdefinitionen

Thread

Ein *Thread* ist ein **„Ausführungsstrang“**, welcher in einem *Prozesskontext* abläuft. In einem **Prozesskontext** können mehrere Threads existieren, die nebenläufig auf mehreren Prozessoren ausgeführt werden können. Alle Threads eines Prozesses **teilen** sich die **Ressourcen** des Prozesses, insbesondere operieren alle Threads im selben **Speicherbereich**. Der Status der Ausführung eines Threads wird im *Threadkontext* vermerkt.

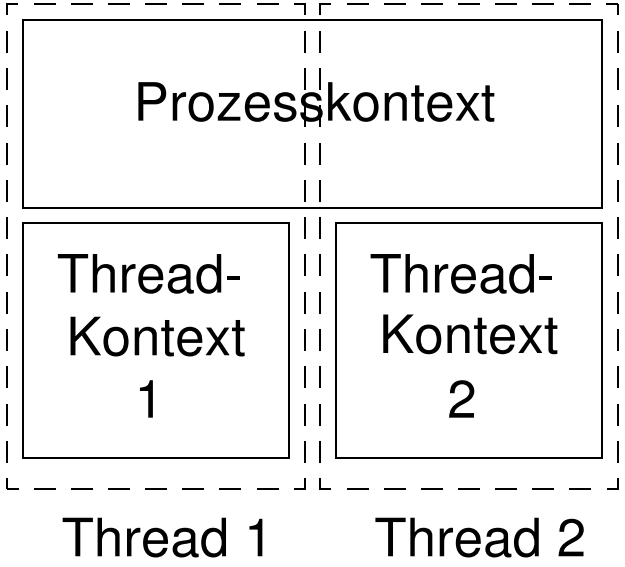
Thread-Kontext

Der Begriff Thread-Kontext bezeichnet analog zum Prozesskontext die Menge aller Informationen, die den Status eines Threads beschreiben und für die Ausführung oder Verwaltung erforderlich sind. Die Attribute des Prozesskontexts werden unterteilt in **Thread-spezifische** Attribute und Attribute, die dem **Prozesskontext** zugeordnet sind:



User-Space

Kernel-Space



Prozesskontext	Thread-Kontext
	Status • Inhalt von Prozessor-Registern (z. B. Befehlsfolgezähler, Statuswort) • Thread-Zustand (z. B. Running, Ready, Blocked)
Ressourcen • Text-Bereich (Programmcode) • Datenbereich (globale Variablen) • Speicherverwaltungsinformationen (z. B. Seitentabelle für virtuelle Speicherverwaltung) • Ressourcen zur Ein-/Ausgabe (z. B. geöffnete Dateien, Arbeitsverzeichnis, Wurzelverzeichnis) • Referenzen auf verwandte Prozesse (Child-Prozesse, Parent-Prozess) • Signale	**Ressourcen** • Stack (lokale Variablen, Rücksprungadressen von Funktionsaufrufen) • Referenzen auf verwandte Threads (Child-Thread, Parent-Thread) • Signal-Handler

Prozesskontext	Thread-Kontext
Verwaltungsinformationen • Prozess-ID • Sicherheitsattribute (z. B. User-ID, Group-ID, Capabilities) • Accounting-Informationen (z. B. bisher verbrauchte CPU-Zeit)	**Verwaltungsinformationen** • Thread-ID

Multithreading

Der Begriff *Multithreading* bezeichnet die Eigenschaft eines Rechnersystems, die (quasi-)simultane Ausführung von **mehreren Threads je Prozess** zu unterstützen.

Thread-Lebenszyklus und Thread-Zustände

Ähnlich wie Prozesse können Threads erzeugt und beendet werden, sie können eine Thread-Hierarchie aufweisen und die gleichen Zustände wie Prozesse durchlaufen. Wird ein Prozess beendet, dann werden hierbei alle Threads des Prozesses beendet.

6.3 Interprozesskommunikation und Synchronisation

1 Koexistierende Prozesse (Concurrent Processes)

2 Datenaustausch zwischen Prozessen

3 Wechselseitiger Ausschluss (Mutual Exclusion)

4 Prozesssynchronisation

5 Verklemmungen (Deadlocks)

6 Realisierung von atomaren Operationen und Kontextwechsel

1 Koexistierende Prozesse (Concurrent Processes)

Entkoppelte Prozesse

Zwei Prozesse P_1 und P_2 sind *entkoppelt*, wenn die **Datenbereiche** D_1 und D_2 **disjunkt** zueinander sind und **keine gemeinsamen Betriebsmittel** benötigt werden.

Gekoppelte Prozesse

Zwei Prozesse P_1 und P_2 sind *gekoppelt*, wenn sie

(a) **Betriebsmittel** gemeinsam benutzen oder

(b) auf gemeinsame **Daten** zugreifen oder

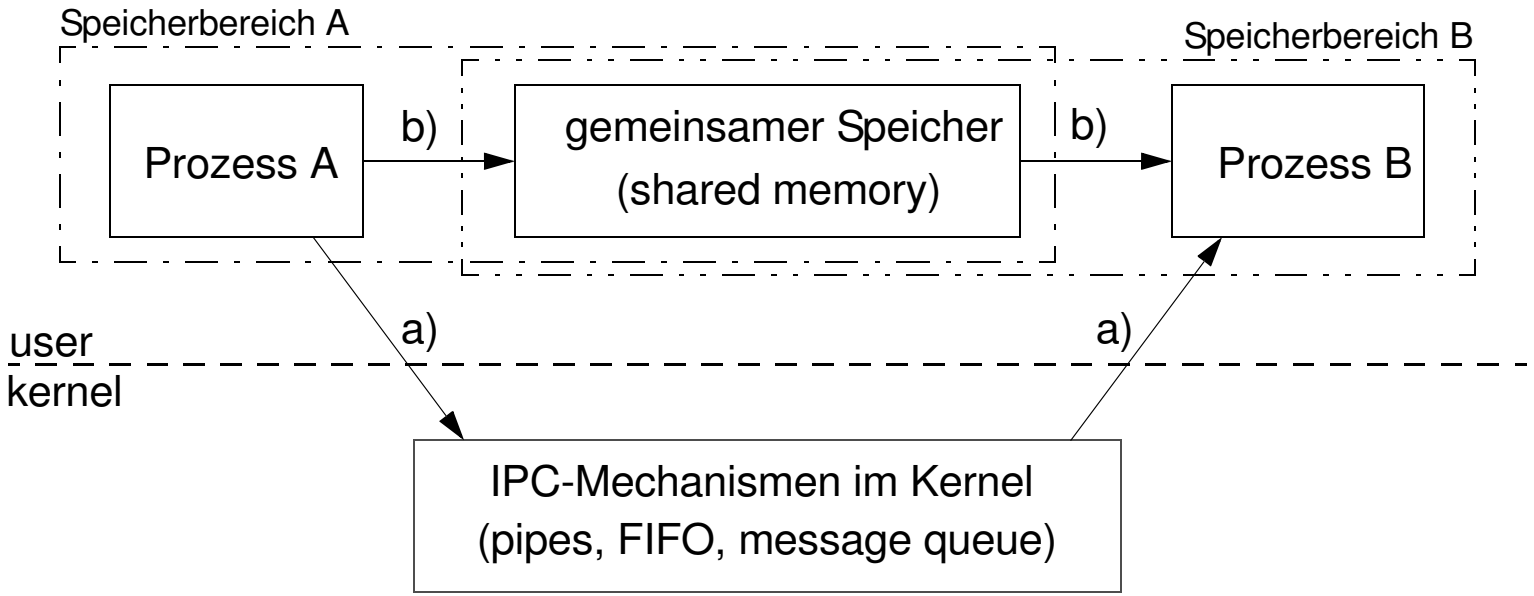
(c) zur Lösung einer Aufgabe kooperieren müssen, d. h. ihre Abläufe über Daten oder Botschaften (Signale) **synchronisiert** werden.

Grundprobleme entstehen durch sogenannte **kritische Bereiche**, in denen die Kopplung im vorgenannten Sinne erfolgt. Abhilfe wird durch Mechanismen wie **wechselseitiger Ausschluss** oder **Prozesssynchronisation** geschaffen.

2 Datenaustausch zwischen Prozessen

Problem

Prozesse dürfen zunächst nur auf ihre **eigenen Speicherbereiche** zugreifen. Zur **Interprozesskommunikation** (Inter-Process Communication, IPC) muss daher das Betriebssystem entsprechende Mechanismen bereitstellen.



Lösungsansätze

a. Datenaustausch über das Betriebssystem

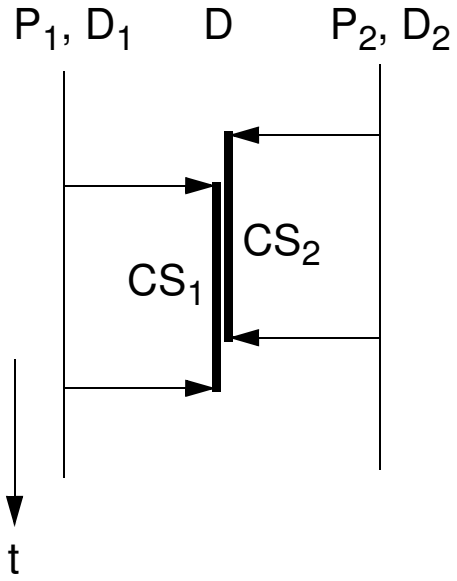
- **FIFO, Pipes:**
  Den beteiligten Prozessen zugeordnete Puffer, die wie Dateien behandelt werden können (read/write)
- **Message Queue:**
  Puffer unabhängig von Prozessen, berechtigte Prozesse können Nachrichten senden und empfangen
- **Signale:**
  Information über ein Ereignis mit vorgegebener Bedeutung, keine weiteren Daten
  (z. B. UNIX SIGTERM - termination)

b. Datenaustausch über gemeinsamen Speicher

- Vom Betriebssystem angeforderter Speicher, auf den **mehrere Prozesse zugreifen** können
- Datenaustausch ohne Einbeziehung des Betriebssystems
- Organisation bleibt den Prozessen überlassen
  - Signalisierung von Änderungen
  - Verhinderung gleichzeitigen Zugriffs

3 Wechselseitiger Ausschluss (Mutual Exclusion)

Problemstellung



P_1, P_2	Prozesse
D_1, D_2	Datenbereiche von P_1 bzw. P_2
D	Gemeinsamer Datenbereich oder gemeinsames Betriebsmittel
CS	**Kritischer Bereich** (Critical Section)
	Zeitspanne, während der beide Prozesse auf gemeinsame Daten oder Betriebsmittel zugreifen.

Beispiel **Zugriff auf eine gemeinsame Variable N**

N: Zählgröße, bedeutet die „Summe aller Zugriffe“

Jeder Prozess vollziehe folgende drei Operationen:

```
LOAD  N
ADD   1
STORE N
```

N sei zur Zeit t_0 17

P_1 werde aktiv zur Zeit t_0:

```
LOAD  N
ADD   1
```

← Interrupt infolge z. B. Ablauf der Zeitscheibe, Dispatcher lädt nächsten Prozess

P_2 werde aktiv zur Zeit t_0+2:

```
LOAD  N
ADD   1
STORE N
```

P_1 erhält den Prozessor und fährt ab Unterbrechungszeitpunkt fort:

```
STORE N
```

Ergebnis N = 18 (anstelle von 19)

Beobachtung

Das Ergebnis hängt von der zeitlichen Abfolge der Prozesse ab. Dies wird als **Wettlaufsituation** (**Race Condition**) bezeichnet.

Abhilfe

Es darf keine Unterbrechung des Vorganges aus den drei Operationen zugelassen werden.

Allgemein

Befindet sich ein Prozess in einem **kritischen Bereich**, darf **kein anderer Prozess** auf **gemeinsame Daten** zugreifen.

Konzepte zur Realisierung

- Deaktivierung der Interrupts
- Lock-Variablen
- Semaphore-Variablen
- Monitore

a) Deaktivierung der Interrupts

Prinzip

Während der Bearbeitung des kritischen Bereiches werden die Interrupts deaktiviert, sodass der Prozess **nicht unterbrochen** werden kann.

Im Folgenden sei

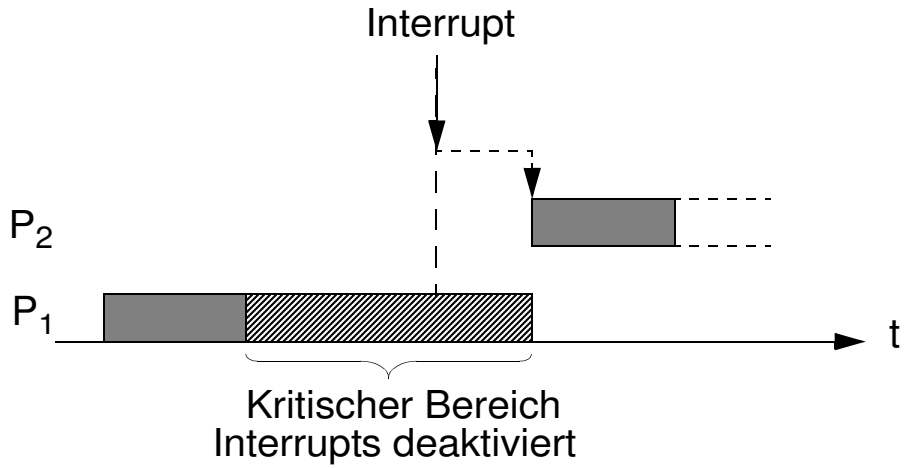
`critical_section()`	kritischer Bereich, innerhalb dessen auf ein gemeinsames Datenobjekt zugegriffen wird
`disable_interrupts()`	Anweisung zur Deaktivierung der Interrupts des Prozessors
`enable_interrupts()`	Anweisung zur Aktivierung der Interrupts des Prozessors
`concurrent (`P_x`, `P_y`,..)`	quasi-simultane Ausführung mehrerer Prozesse

Wechselseitiger Ausschluss (Mutual Exclusion)

```
Process
{
   run()
   {
      noncritical_section();

      disable_interrupts();
      critical_section();
      enable_interrupts();

      noncritical_section();
   }
}

main()
{
   concurrent
   (
      new Process().run(),
      new Process().run()
   )
}
```

Kritischer Bereich:
Keine Unterbrechung durch Interrupts

quasi-simultane
Ausführung



Nachteile

- **Blockierung des Prozessors**: kein anderer Prozess kann aktiv werden, selbst solche nicht, die unabhängig von den gemeinsamen Daten sind
- Deaktivierung der Interrupts ist eine **privilegierte Operation** (Betriebssystem), Benutzerprozesse können diesen Mechanismus nicht benutzen
- **Mehrprozessorsysteme**: Prozesse auf den anderen Prozessoren können weiterhin auf gemeinsam genutzte Variablen zugreifen

b) Lock-Variablen

Prinzip

Eine Lock-Variable beschreibt, ob ein kritischer Bereich frei oder gesperrt ist:

Lock-Variable = **`false`**: kritischer Bereich **frei**

Lock-Variable = **`true`**: kritischer Bereich **gesperrt**

Vor dem Betreten des kritischen Bereiches muss ein Prozess die Lock-Variable überprüfen und bei Betreten auf **`true`** setzen. **Verlässt** der Prozess den kritischen Bereich, so muss die Lock-Variable auf **`false`** gesetzt werden.

Problem

Das Überprüfen und Setzen einer Variablen bedeutet das **Ausführen mehrerer Maschinenbefehle**, zwischen denen der **Prozess unterbrochen** werden kann. Das Überprüfen und Setzen muss daher **unteilbar (atomar)** realisiert werden

- Auf Einprozessorsystemen: Deaktivierung der Interrupts
- Auf Mehrprozessorsystemen: Hardwareunterstützung notwendig, z.B. **test-and-set** Funktion

Lösung: atomare test-and-set() Funktion

- Lesen und Setzen einer Variablen **in einem Maschinenbefehl**
- Rückgabewert: Wert der Variablen vor der Änderung
  - **`false`**: Variable war **`false`** (wurde auf **`true`** gesetzt)
  - **`true`**: Variable war **`true`** (nicht geändert)

Beispiel

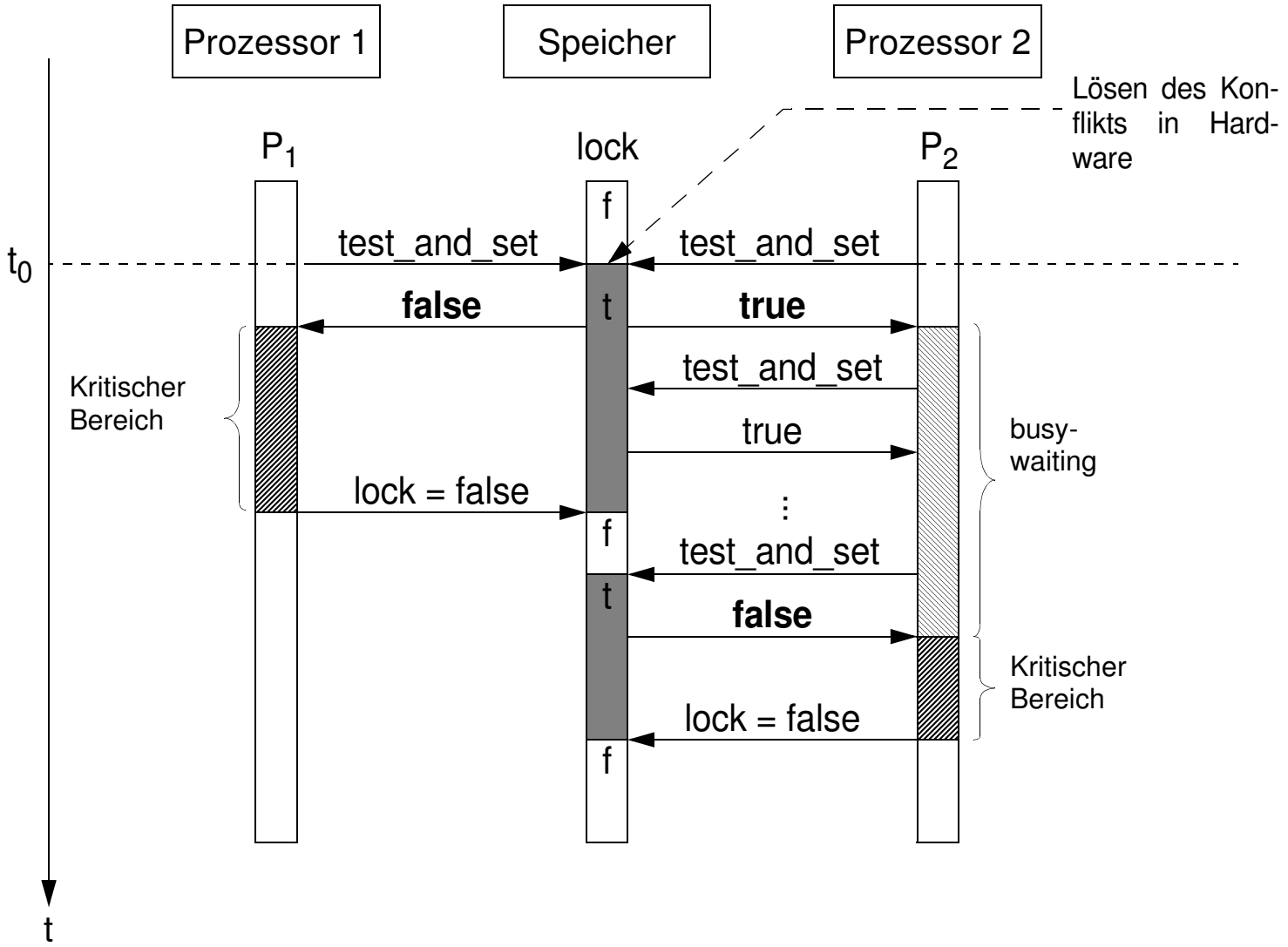
```
boolean lock;

Process
{
    run(){
        noncritical_section();

        while (test_and_set(lock) == true) {};
        critical_section();
        lock = false;

        noncritical_section();
    }
}

main(){
    lock = false;
    concurrent(
        new Process().run(),
        new Process().run()
    )
}
```

Überprüfen und Setzen der Sperre:
Warten bis test_and_set **`false`** liefert
(busy-waiting)

Zurücksetzen der Sperre

Initialisierung der
Lock-Variablen

Ablauf: zur Zeit t_0 wollen die Prozesse P_1 und P_2 den kritischen Bereich betreten



Wechselseitiger Ausschluss (Mutual Exclusion)

Ein Prozess, der sich in einem mit Lock-Variable geschützten Bereich befindet, **kann per Interrupt unterbrochen werden**. Es ist jedoch gewährleistet, dass kein anderer Prozess während dieser Zeit den kritischen Bereich betritt.

Problem

Während des Wartens auf das Freiwerden einer Sperre mittels while-Schleife (busy-waiting, spinlock) belegt der Prozess unnötigerweise den Prozessor.

Lösung

Der Prozess geht, wenn die Sperre aktiv ist, in den Zustand „blocked“ über und wird sobald die Sperre frei wird wieder auf „ready“ gesetzt. Dieser Mechanismus wird bei **Semaphore-Variablen** eingesetzt.

c) Semaphore-Variable (Dijkstra, 1965)

Im Gegensatz zu Lock-Variablen haben Operationen auf Semaphore-Variablen Einfluss auf den **Prozesszustand**. Eine **Semaphore-Variable** ist eine **vom Betriebssystem geschützte** Variable; ihr Wert kann nur zugegriffen und verändert werden durch drei Operationen:

- Initialisierung
- P-Operation
- V-Operation

Sei

sem	Semaphore-Variable
P(sem)	P-Operation; auch als Down(.), Wait(.), Delay(.) bekannt
V(sem)	V-Operation; auch als Up(.), Send(.), Signal(.) bekannt.

P und V sind sich wechselseitig ausschließende Operationen, welches durch Implementierung im **Betriebssystem** sichergestellt wird.

Werte:	sem = 0,1	binäre Semaphore-Variable („Mutex“)
	sem = 0,1, ..., n-1	Integer- oder Zähl-Semaphore

Definition der P-, V-Operationen

P(sem)
```
if (sem > 0) sem = sem - 1;
else (wait on sem);
```

V(sem):
```
if (≥ 1 processes are waiting on sem)
   (select one of them to proceed);
else sem = sem + 1;
```

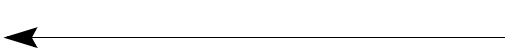
Falls also sem > 0 ist, wird die Variable dekrementiert und in den kritischen Bereich eingetreten; falls sem = 0 ist, muss gewartet werden, bis der kritische Bereich verlassen wird. Mit V(sem) werden **wartende Prozesse fortgesetzt**, andernfalls wird die Variable wieder inkrementiert. Bei **Mehrprozessorsystemen** wird der Zugriff auf Semaphore-Variablen mit **test-and-set** Anweisungen gesichert.

Lösung des wechselseitigen Ausschlusses mittels Semaphoren

Der wechselseitige Ausschluss während der kritischen Bereiche wird durch Einschluss des kritischen Bereichs in ein Paar von P- und V-Operationen gewährleistet.

Algorithmus

```
SEMAPHORE mutex;

Process
{
   run()
   {
      P(mutex);
      critical_section();
      V(mutex);
   }
}

main()
{
   init (mutex,1);
   concurrent
   (
      new Process().run(),
      new Process().run()
   )
}
```



Initialisierung der
Semaphore-Variablen