

2 Prozesse und Threads

- 1 Einführung
- 2 *Prozesse und Threads*
- 3 Speicherverwaltung
- 4 Dateisysteme
- 5 Eingabe und Ausgabe
- 6 Deadlocks
- 7 Virtualisierung und die Cloud
- 8 Multiprozessorsysteme
- 9 IT-Sicherheit
- 10 Fallstudie 1: Linux
- 11 Fallstudie 2: Windows
- 12 Entwurf von Betriebssystemen

2 Prozesse und Threads

2.1 Prozesse

2.2 Threads

2.3 Interprozesskommunikation

2.4 Scheduling

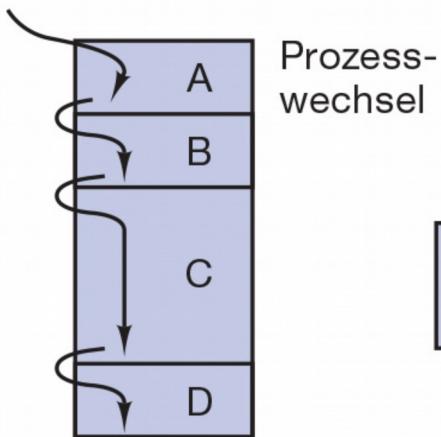
2.5 Klassische Probleme der
Interprozesskommunikation

2.1 Prozesse

- 2.1.1 Das Prozessmodell
- 2.1.2 Prozesserzeugung
- 2.1.3 Prozessbeendigung
- 2.1.4 Prozesshierarchien
- 2.1.5 Prozesszustände
- 2.1.6 Implementierung von Prozessen
- 2.1.7 Modellierung der Multiprogrammierung

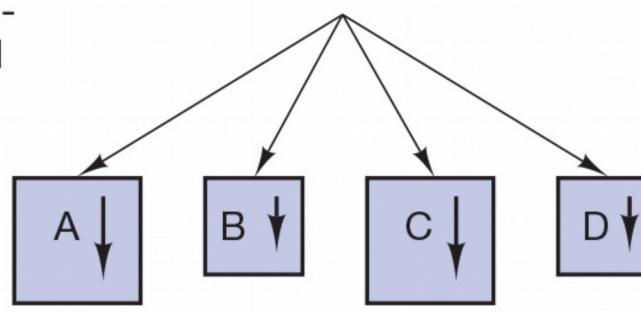
Das Prozessmodell

Ein Befehlszähler

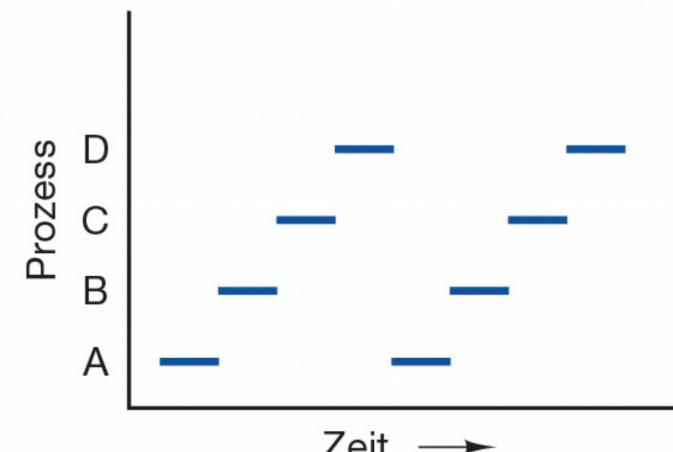


a

Vier Befehlszähler



b



c

Abbildung 2.1: (a) Multiprogrammierung von vier Programmen. (b) Konzeptionelles Modell von vier individuellen sequenziellen Prozessen. (c) Zu jedem Zeitpunkt ist immer nur ein Programm aktiv.

Prozesserzeugung

Vier Hauptereignisse welche zur Erzeugung von Prozessen führen:

1. Initialisierung des Systems
2. Ausführung eines Systemaufrufs zur Prozesserzeugung durch einen laufenden Prozess
3. Nutzeranforderung zur Erzeugung eines neuen Prozesses
4. Einleitung eines Batch Jobs

Prozessbeendigung

Typische Bedingungen welche zur Beendigung eines Prozesses führen:

1. Normal exit (freiwillig).
2. Error exit (freiwillig).
3. Fatal error (unfreiwillig).
4. Killed by another process (unfreiwillig).

Prozesshierarchien

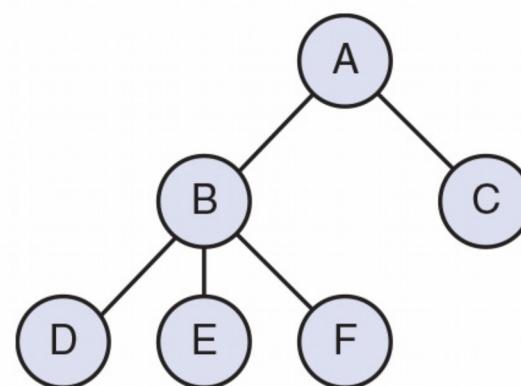


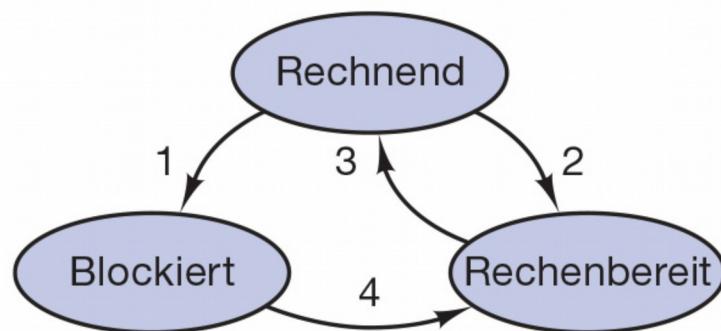
Abbildung 1.13: Ein Prozessbaum. *A* hat zwei Kindprozesse *B* und *C* erzeugt. Prozess *B* hat wiederum die Prozesse *D*, *E* und *F* erzeugt.

Prozesszustände (1)

Ein Prozess kann in folgenden drei Zuständen sein:

1. Rechnend / Running (benutzt in diesem Moment tatsächlich den Prozessor).
2. Rechenbereit / Ready (lauffähig; vorübergehend gestoppt um einen anderen Prozess laufen zu lassen).
3. Blockiert / Blocked (kann nicht ausgeführt werden bis ein externes Ereignis eintritt).

Das Prozessmodell (2)



1. Prozess blockiert,
weil er auf Eingabe wartet
2. Scheduler wählt einen
anderen Prozess aus
3. Scheduler wählt
diesen Prozess aus
4. Eingabe vorhanden

Abbildung 2.2: Ein Prozess kann sich in den Zuständen rechnend, rechenbereit und blockiert befinden. Es existieren die dargestellten Übergänge zwischen diesen Zuständen.

Das Prozessmodell (3)

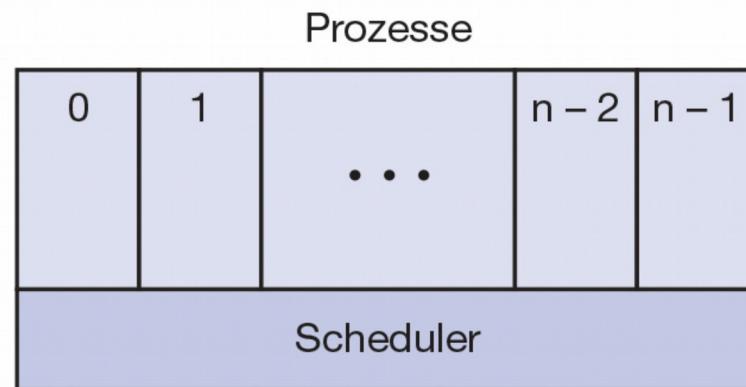


Abbildung 2.3: Die unterste Schicht eines prozessstrukturierten Betriebssystems behandelt Interrupts und Scheduling. Oberhalb dieser Schicht befinden sich sequenzielle Prozesse.

Implementierung von Prozessen (1)

Prozessverwaltung	Speicherverwaltung	Dateiverwaltung
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Stacksegment	Dateideskriptor
Stackpointer		Benutzer-ID
Prozesszustand		Gruppen-ID
Priorität		
Scheduling-Parameter		
Prozess-ID		
Elternprozess		
Prozessgruppe		
Signale		
Startzeit des Prozesses		
Benutzte CPU-Zeit		
CPU-Zeit der Kindprozesse		
Zeitpunkt des nächsten Alarms		

Abbildung 2.4: Einige Felder eines typischen Eintrags einer Prozesstabellen.

Implementierung von Prozessen (2)

1. Hardware sichert Befehlszähler usw.
2. Hardware holt neuen Befehlszähler vom Interruptvektor
3. Assemblerprozedur speichert Register
4. Assemblerprozedur richtet neuen Stack ein
5. C-Unterbrechungsroutine läuft (liest und puffert i.d.R. Eingaben)
6. Scheduler entscheidet, welcher Prozess als Nächstes läuft
7. C-Prozedur kehrt zum Assemblercode zurück
8. Assemblerprozedur startet neuen aktuellen Prozess

Abbildung 2.5: Aufgaben der untersten Schicht des Betriebssystems bei Auftreten eines Interrupts.

Modellierung der Multiprogrammierung

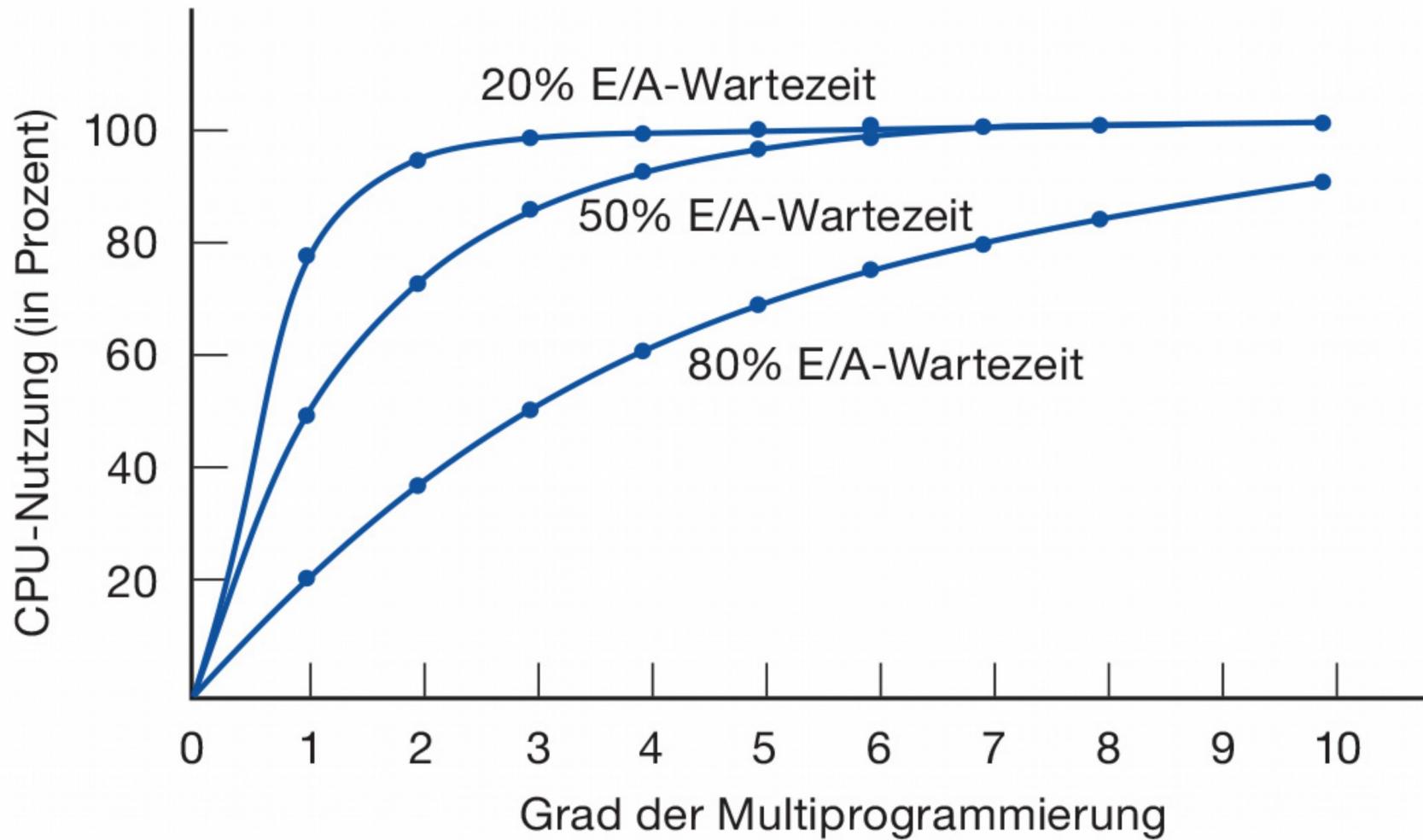


Abbildung 2.6: CPU-Ausnutzung als eine Funktion der Anzahl der Prozesse im Speicher.

2.2 Threads

- 2.2.1 Der Gebrauch von Threads
- 2.2.2 Das klassische Thread-Modell
- 2.2.3 POSIX-Threads
- 2.2.4 Implementierung von Threads im Benutzeradressraum
- 2.2.5 Implementierung von Threads im Kern
- 2.2.6 Hybride Implementierungen
- 2.2.7 Scheduler-Aktivierungen
- 2.2.8 Pop-up-Threads
- 2.2.9 Einfachthread-Code in Multithread-Code umwandeln

Der Gebrauch von Threads (1)

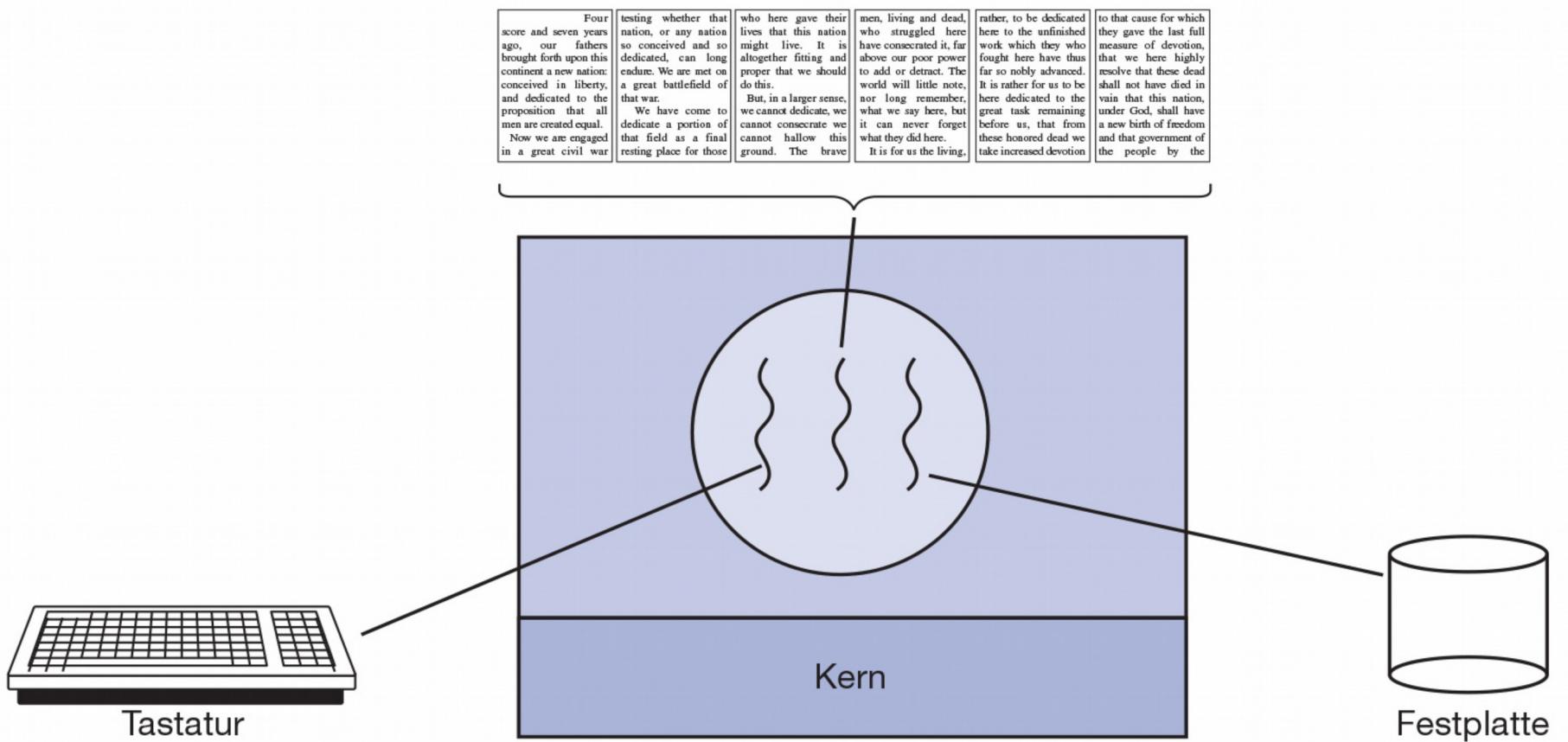


Abbildung 2.7: Ein Textverarbeitungsprogramm mit drei Threads.

Der Gebrauch von Threads (2)

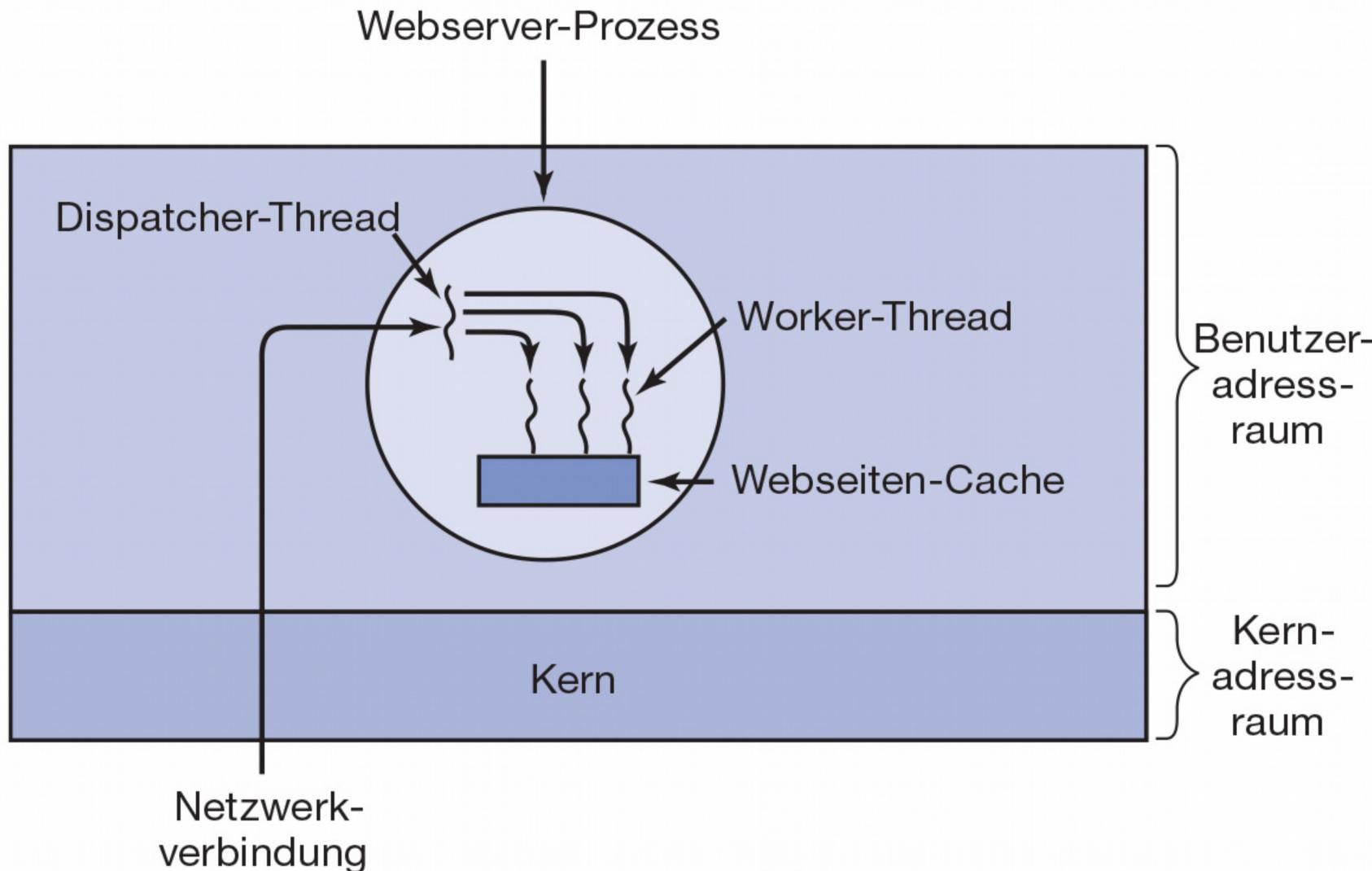


Abbildung 2.8: Ein Webserver mit mehreren Threads.

Der Gebrauch von Threads (3)

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

a

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

b

Abbildung 2.9: Eine grobe Skizze des Codes von ▶ Abbildung 2.8: (a) Dispatcher-Thread; (b) Worker-Thread.

Tanenbaum, A. S.; Bos, H.: Moderne
Betriebssysteme. Pearson Studium 2016

Der Gebrauch von Threads (4)

Modell	Eigenschaften
Threads	Parallel, blockierende Systemaufrufe
Prozesse mit einem Thread	Nicht parallel, blockierende Systemaufrufe
Endlicher Automat	Parallel, nicht blockierende Systemaufrufe, Interrupts

Abbildung 2.10: Drei Möglichkeiten zur Konstruktion eines Servers.

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

Das klassische Thread-Modell (1)

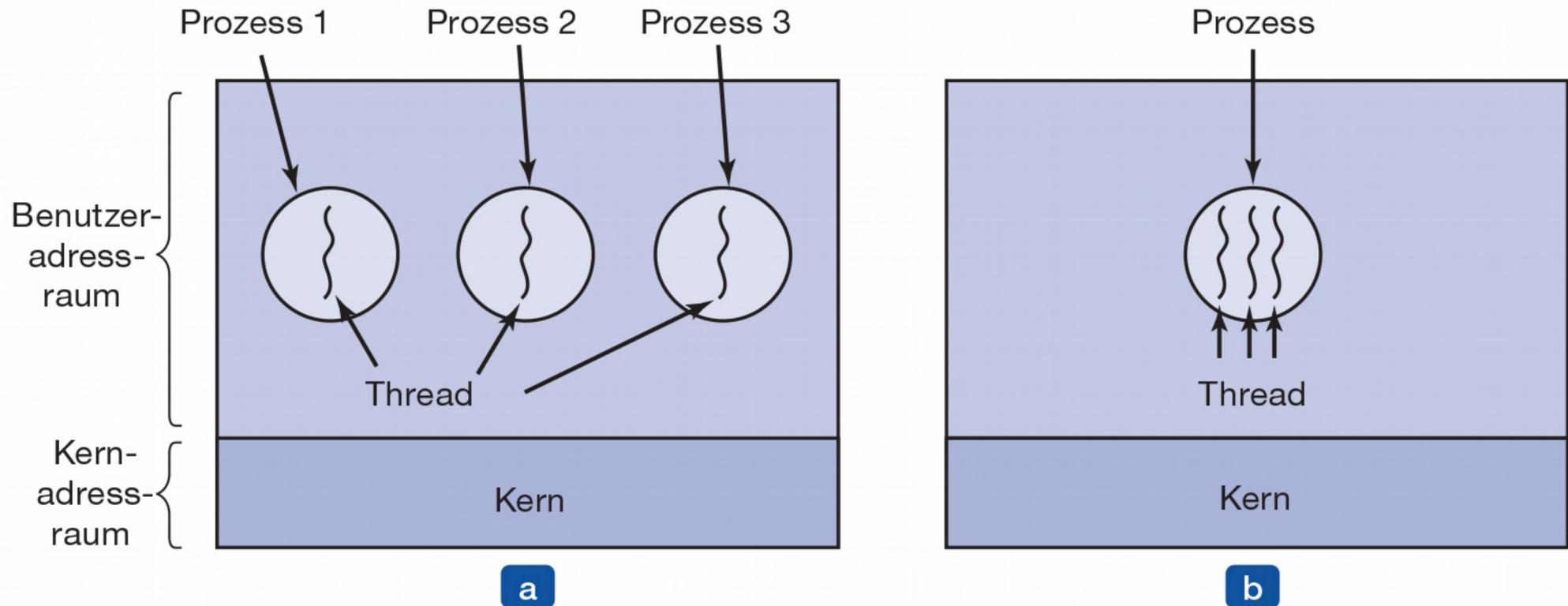


Abbildung 2.11: (a) Drei Prozesse mit je einem Thread. (b) Ein Prozess mit drei Threads.

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

Das klassische Thread-Modell (2)

Elemente pro Prozess	Elemente pro Thread
Adressraum	Befehlszähler
Globale Variable	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Ausstehende Signale	
Signale und Signalroutinen	
Verwaltungsinformationen	

Abbildung 2.12: Die erste Spalte führt Elemente auf, die alle Threads des Prozesses teilen. Die zweite Spalte zeigt Elemente, die zu einem individuellen Thread gehören.

Das klassische Thread-Modell (3)

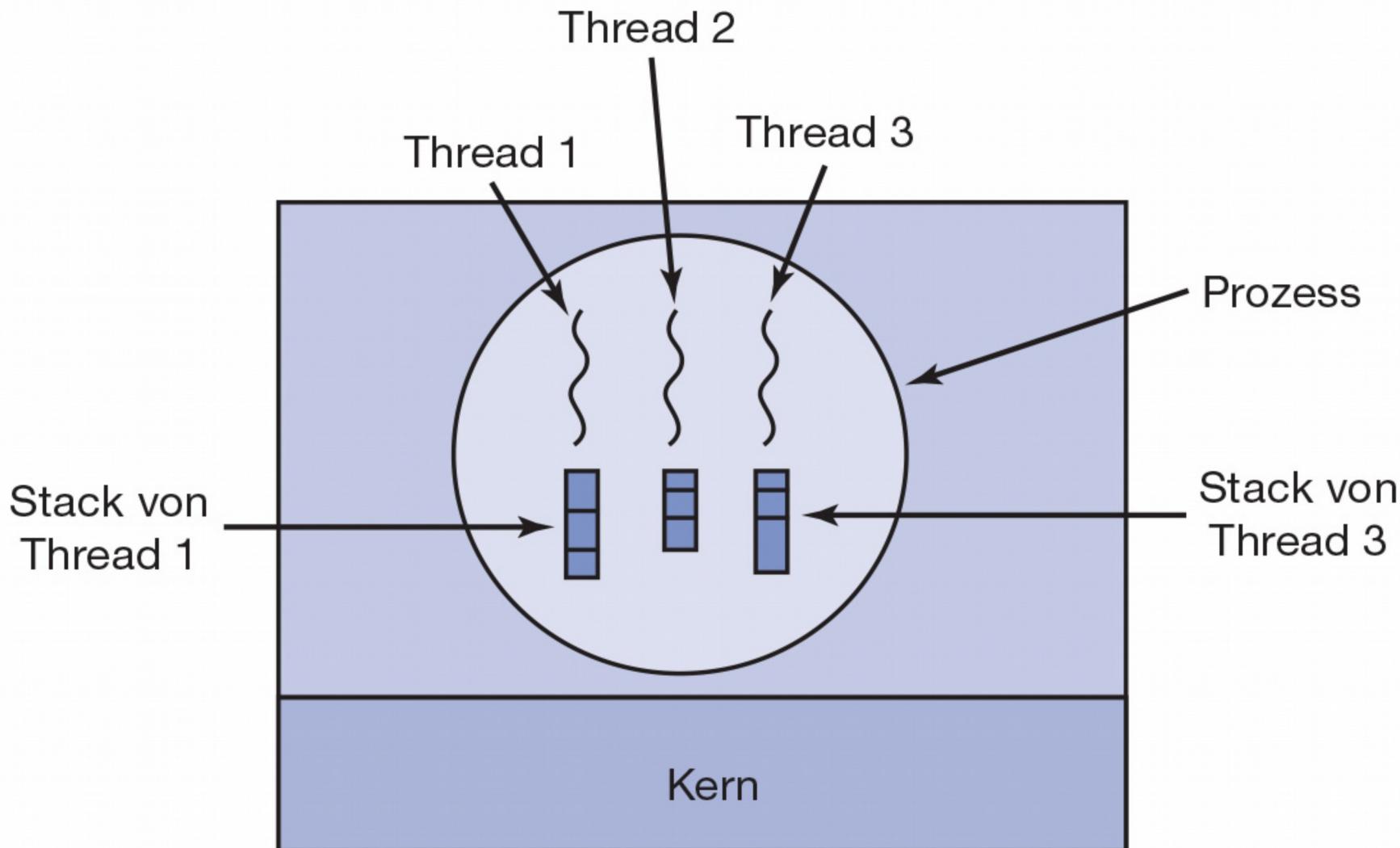


Abbildung 2.13: Jeder Thread hat seinen eigenen Stack.

POSIX-Threads (1)

Thread-Aufruf	Beschreibung
pthread_create	Erzeugt einen neuen Thread
pthread_exit	Beendet den aufrufenden Thread
pthread_join	Wartet auf die Beendigung eines bestimmten Threads
pthread_yield	Gibt die CPU frei, damit andere Threads laufen können
pthread_attr_init	Erzeugt und initialisiert eine Attributstruktur
pthread_attr_destroy	Löscht die Attributstruktur eines Threads

Abbildung 2.14: Einige der Pthread-Funktionsaufrufe.

POSIX-Threads (2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* Diese Funktion druckt die Thread-ID und beendet sich dann. */
    printf("Hello World. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* Das Hauptprogramm erzeugt 10 Threads und beendet sich dann.*/
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Implementierung von Threads im Benutzeradressraum / im Kern

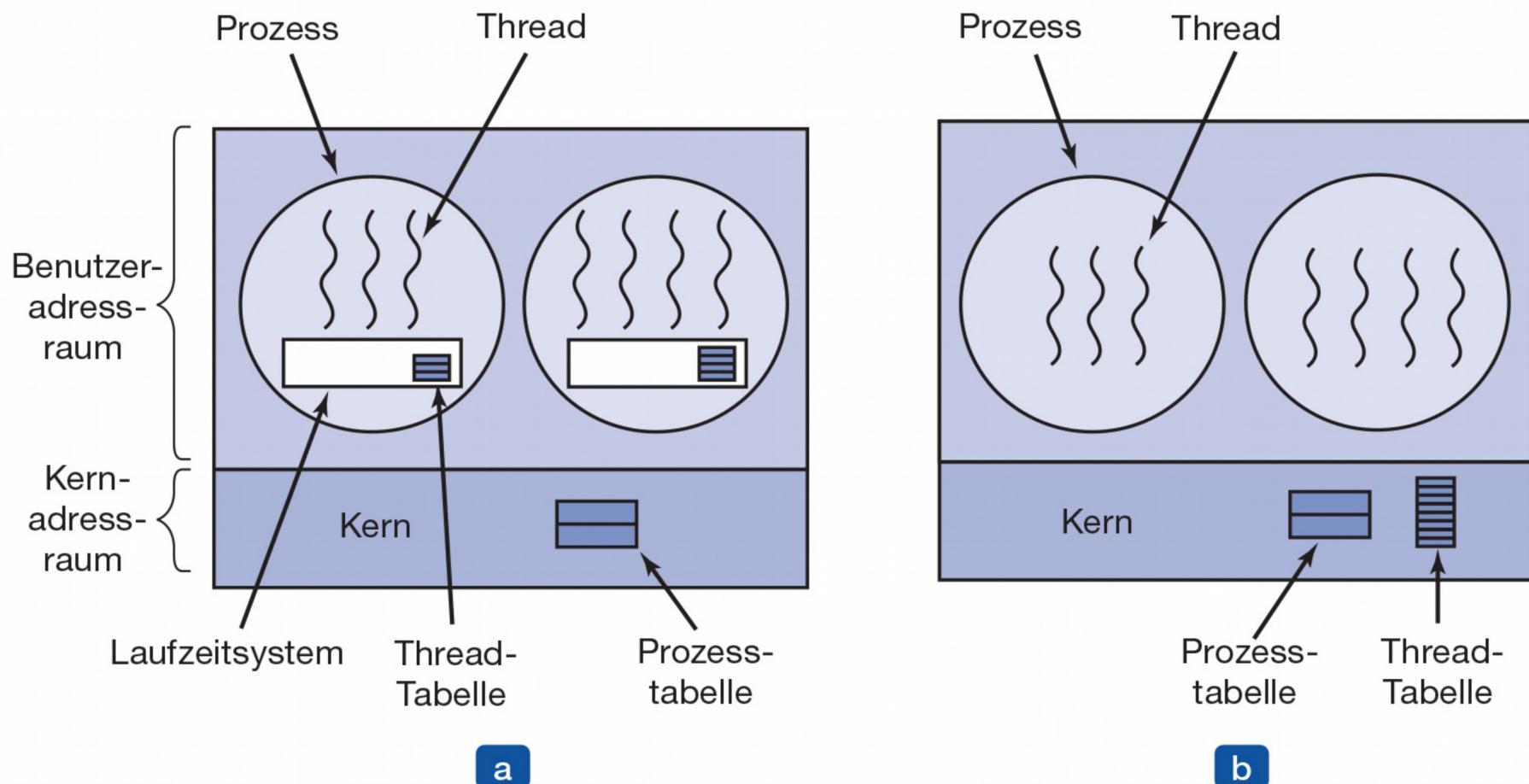


Abbildung 2.16: (a) Ein Thread-Paket auf Benutzerebene. (b) Ein Thread-Paket, das vom Kern verwaltet wird.

Hybride Implementierungen

mehrere Benutzer-Threads
auf einem Kern-Thread

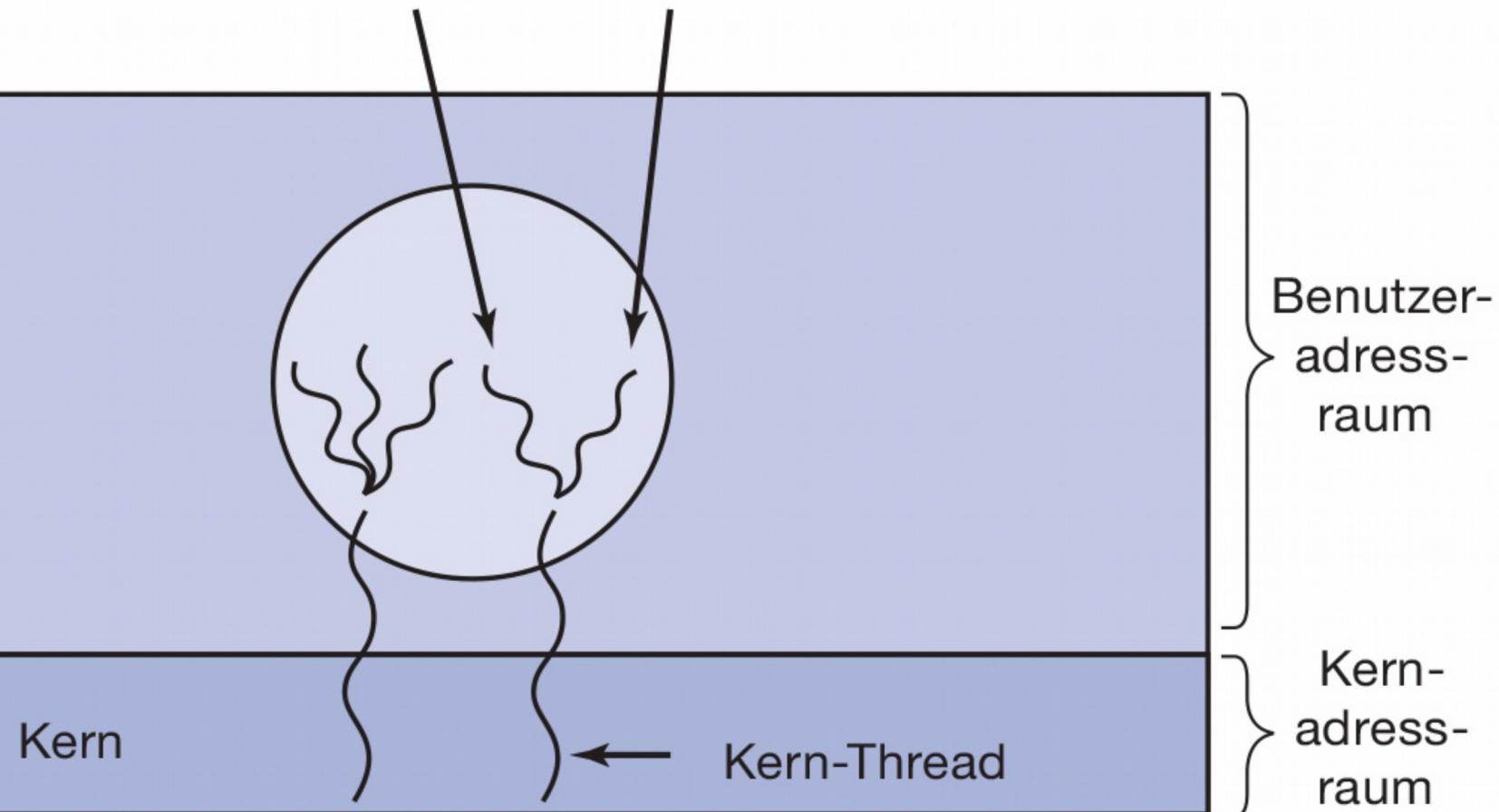


Abbildung 2.17: Bündeln von Benutzer-Threads auf Kern-Threads.

Pop-Up-Threads

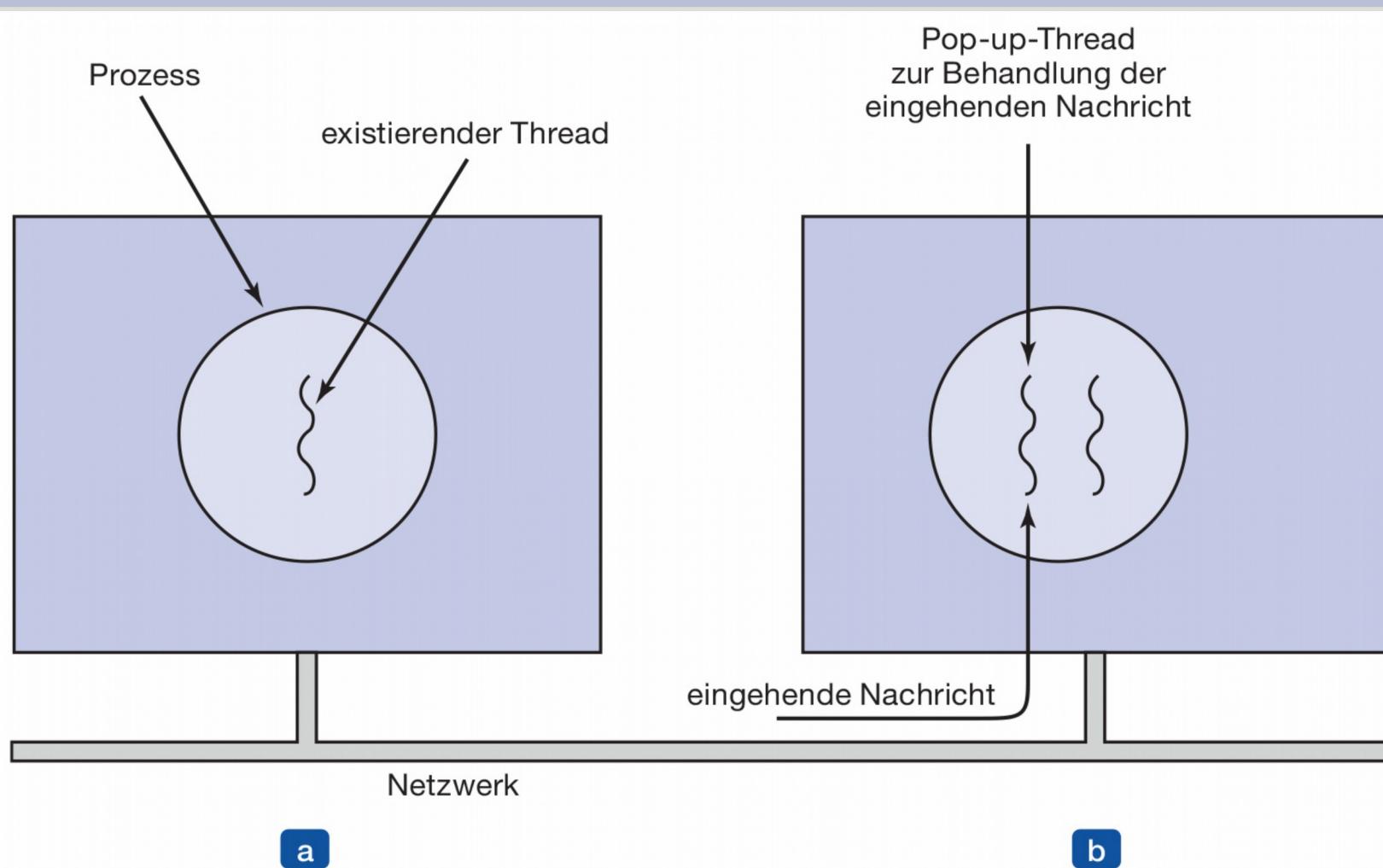


Abbildung 2.18: Erzeugung eines neuen Threads, wenn eine Nachricht eintrifft: (a) bevor die Nachricht eintrifft; (b) nachdem die Nachricht eingetroffen ist.

Einfachthread-Code in Multithread-Code umwandeln (1)

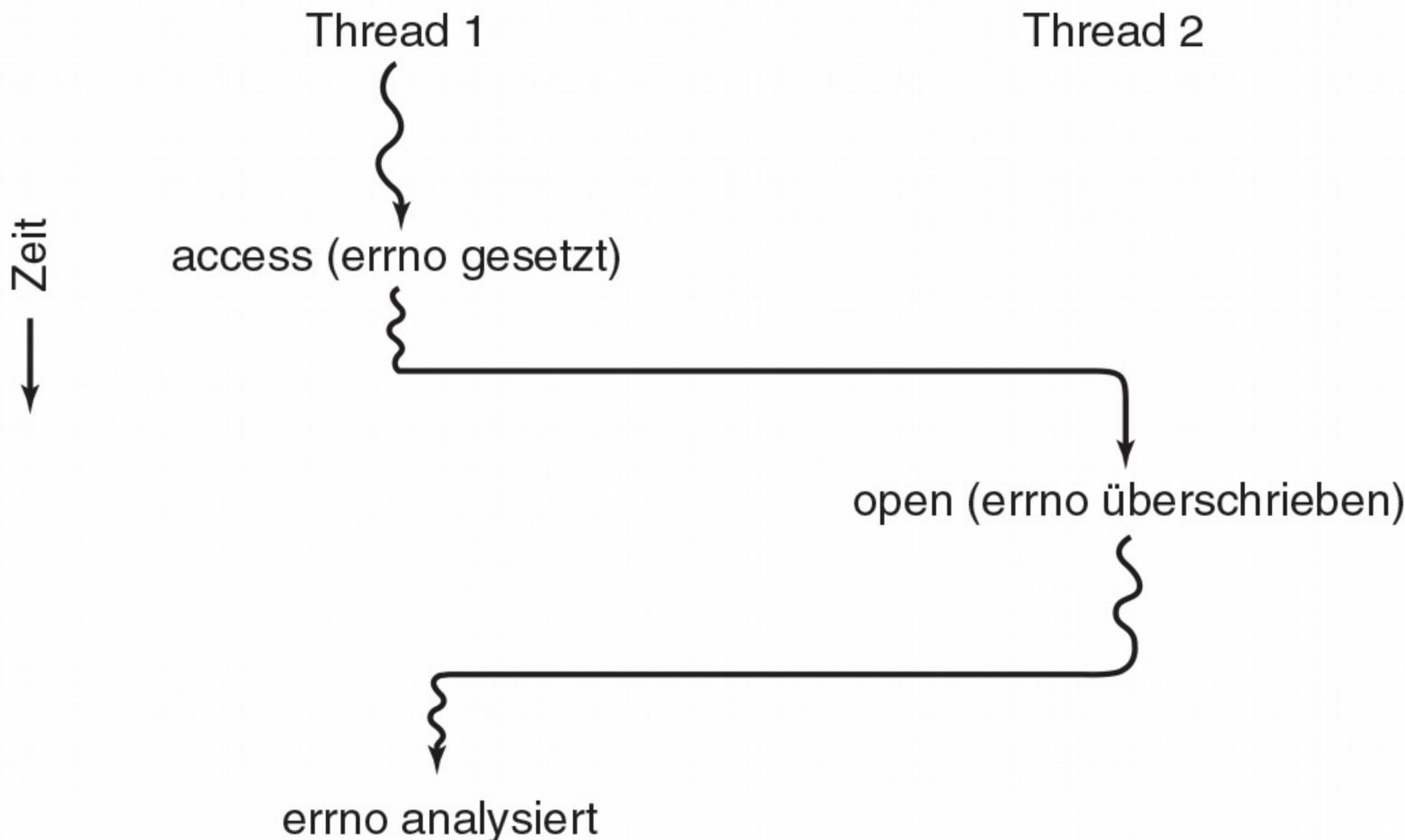


Abbildung 2.19: Konflikte zwischen Threads bei der Verwendung einer globalen Variablen.

Einfachthread-Code in Multithread-Code umwandeln (2)

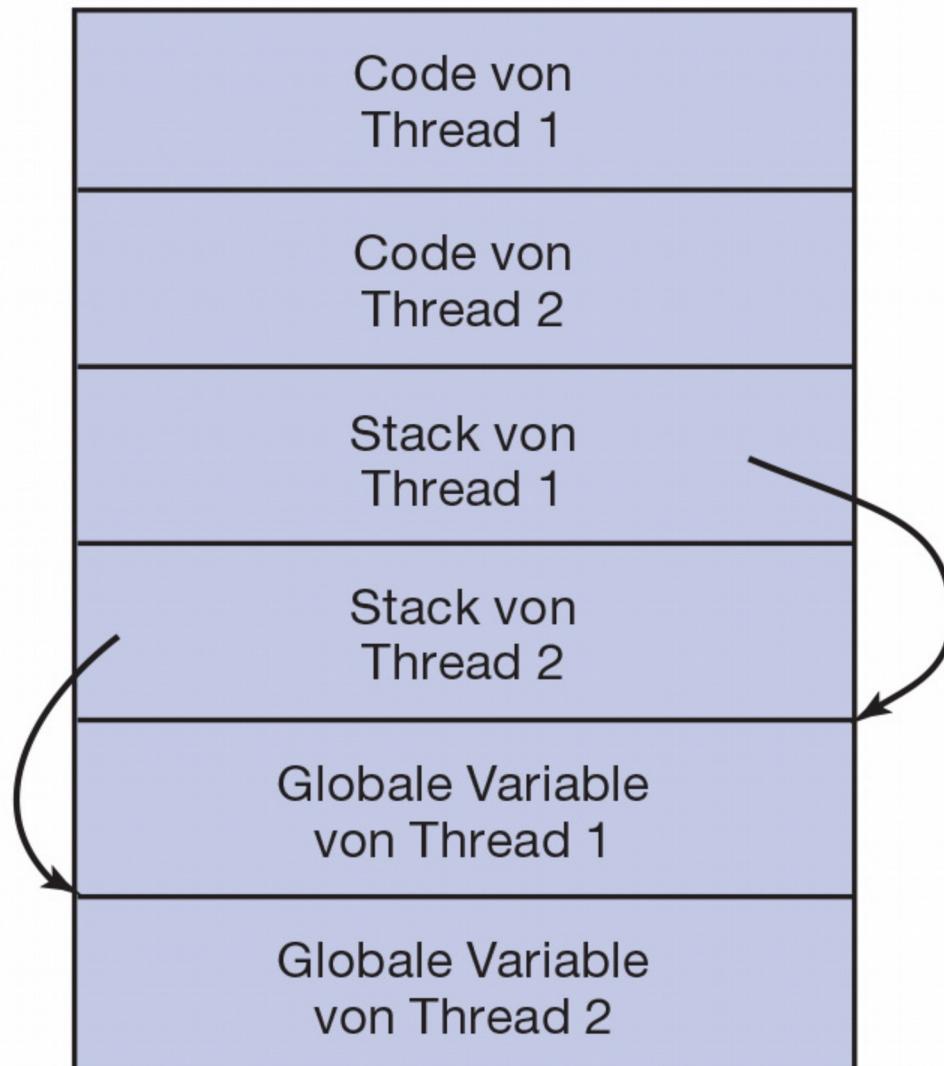


Abbildung 2.20: Threads können private globale Variablen besitzen.

2.3 Interprozesskommunikation

- 2.3.1 Race Conditions
- 2.3.2 Kritische Regionen
- 2.3.3 Wechselseitiger Ausschluss mit aktivem Warten
- 2.3.4 Sleep und Wakeup
- 2.3.5 Semaphor
- 2.3.6 Mutex
- 2.3.7 Monitor
- 2.3.8 Nachrichtenaustausch
- 2.3.9 Barrieren
- 2.3.10 Sperren vermeiden:
das Read-Copy-Update-Schema

Race Conditions

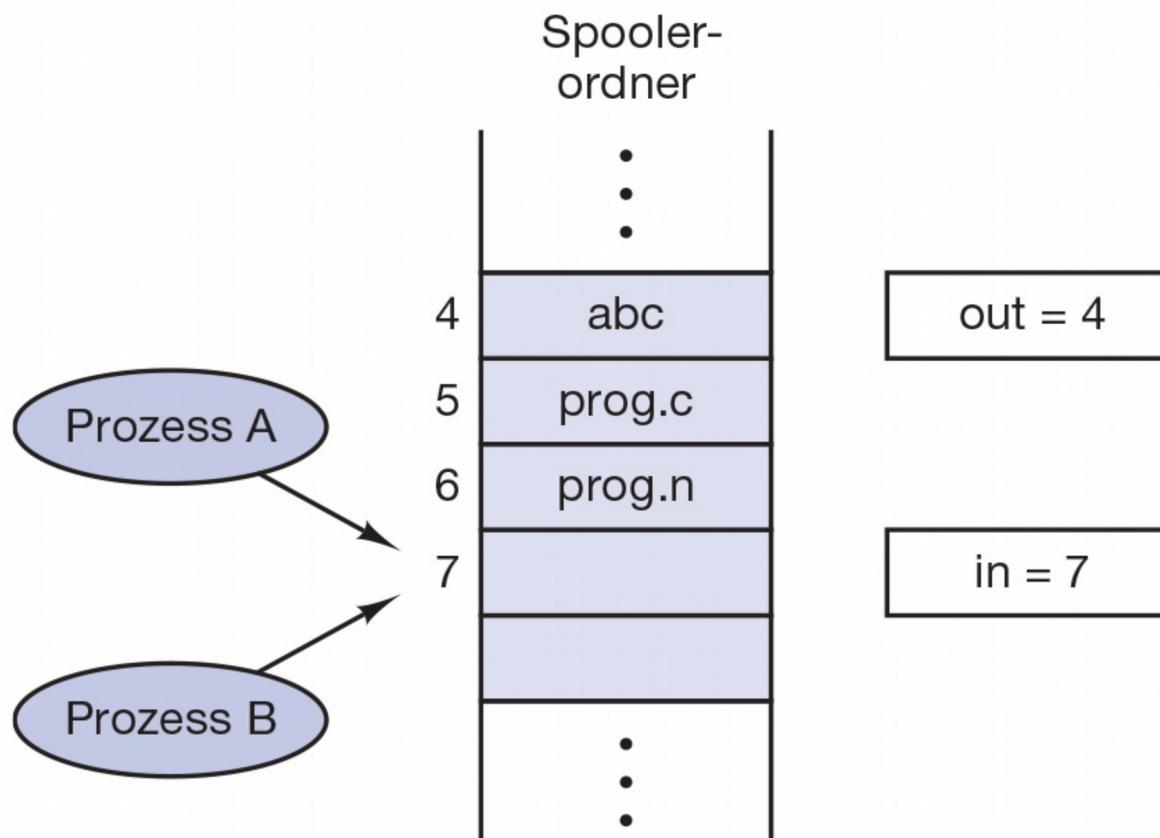


Abbildung 2.21: Zwei Prozesse möchten auf gemeinsam genutzten Speicher gleichzeitig zugreifen.

Kritische Regionen (1)

Voraussetzungen für die Vermeidung von Race Conditions:

1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein.
2. Es dürfen keine Annahmen über Leistungsfähigkeiten oder die Anzahl der Prozessoren gemacht werden.
3. Kein Prozess, der außerhalb seiner kritischen Region ausgeführt wird, kann andere Prozesse blockieren.
4. Kein Prozess sollte für immer auf seine kritische Region warten müssen.

Kritische Regionen (2)

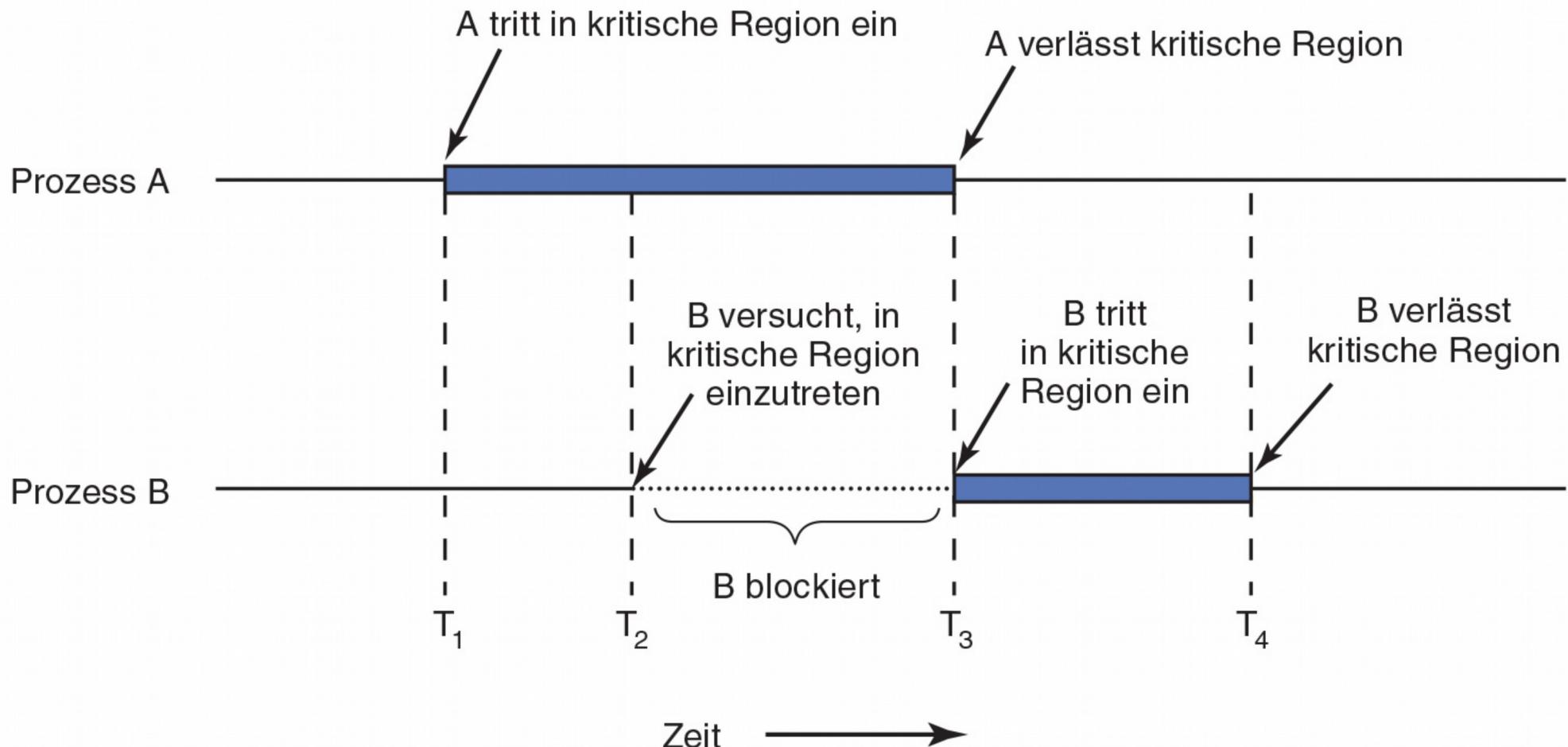


Abbildung 2.22: Wechselseitiger Ausschluss unter Verwendung von kritischen Regionen.

Wechselseitiger Ausschluss mit aktivem Warten – die Lösung von Peterson

```
#define FALSE 0
#define TRUE 1
#define N    2          /* Anzahl der Prozesse */

int turn;                  /* wer ist am Zug? */
int interested[N];         /* alle Werte anfangs 0 (FALSE)*/

void enter_region(int process); /* Prozess: wer tritt ein (0 oder 1)? */
{
    int other;               /* Nummer des anderen Prozesses */

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;          /* der andere Prozess */
    /* Interesse zeigen */
    /* Flag setzen */
    while (turn == process && interested[other] == TRUE) /* Leeranweisung */
}

void leave_region(int process) /* Prozess: wer verlässt die Region? */
{
    interested[process] = FALSE; /* zeigt Ausstieg aus kritischer Region an */
}
```

Abbildung 2.24: Die Lösung von Peterson für wechselseitigen Ausschluss.

Wechselseitiger Ausschluss mit aktivem Warten – Nutzung des TSL-Befehls

enter_region:	
TSL Rx,LOCK	kopiere Sperrvariable und sperre mit 1
CMP Rx,#0	war die Sperrvariable 0?
JNE enter_region	wenn nicht 0, dann war gesperrt --> in Schleife gehen
RET	Rücksprung; kritische Region wurde betreten
leave_region:	
MOVE LOCK,#0	speichere 0 in Sperrvariable
RET	Rücksprung

Abbildung 2.25: Eintreten und Verlassen einer kritischen Region unter Verwendung des TSL-Befehls.

Wechselseitiger Ausschluss mit aktivem Warten – Nutzung des XCHG-Befehls (x86)

enter_region:

```
MOVE Rx,#1  
XCHG Rx,LOCK  
CMP Rx,#0  
JNE enter_region  
RET
```

| speichere eine 1 im Register
| vertausche Inhalte von Register und Sperrvariable
| war Sperrvariable 0?
| wenn nicht 0, dann war gesperrt --> in Schleife gehen
| Rücksprung; kritische Region wurde betreten

leave_region:

```
MOVE LOCK,#0  
RET
```

| speichere 0 in Sperrvariable
| Rücksprung

Abbildung 2.26: Eintreten und Verlassen einer kritischen Region unter Verwendung des XCHG-Befehls.

Sleep und Wakeup

```
define N 100
count = 0;

l producer(void)
    nt item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);

        /* Anzahl der Einträge im Puffer */
        /* Anzahl der Elemente im Puffer */

        /* Endlosschleife */
        /* erzeuge nächstes Element */
        /* wenn Puffer voll, gehe schlafen */
        /* schreibe Elemente in Puffer */
        /* erhöhe Anzahl der Elemente im Puffer */
        /* war der Puffer leer? */

    }

l consumer(void)
    nt item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;

        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }

        /* Endlosschleife */
        /* wenn Puffer voll, gehe schlafen */
        /* hole Elemente aus Puffer */
        /* dekrementiere Anzahl der */
        /* Elemente im Puffer */
        /* war der Puffer voll? */
        /* gebe Element aus */

    }
```

bildung 2.27: Das Erzeuger-Verbraucher-Problem mit einer verhängnisvollen Race Condition.

Semaphor

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

/* Anzahl der Einträge im Puffer */
/* Semaphore sind spezielle Integerwerte */
/* steuert den Zugriff auf kritische Regionen */
/* zählt leere Puffereinträge */
/* zählt volle Puffereinträge */
```

Abbildung 2.28: Das Erzeuger-Verbraucher-Problem unter Verwendung von Semaphoren.

Mutex

mutex_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

ok: RET

| kopiere Mutex in Register, setze Mutex = 1
| war Mutex 0?
| wenn 0, dann war Mutex nicht gesperrt, Rücksprung
| Mutex belegt; führe anderen Thread aus
| versuche es wieder
| Rücksprung; kritische Region wurde betreten

mutex_unlock:

```
MOVE MUTEX,#0  
RET
```

| speichere eine 0 in Mutex
| Rücksprung

Abbildung 2.29: Implementierung von *mutex_lock* und *mutex_unlock*.

Mutexe in Pthreads (1)

Thread-Anruf	Beschreibung
Pthread_mutex_init	Erzeuge einen Mutex
Pthread_mutex_destroy	Zerstöre einen vorhandenen Mutex
Pthread_mutex_lock	Erlange Sperre oder blockiere
Pthread_mutex_trylock	Erlange Sperre oder erzeuge Fehlermeldung
Pthread_mutex_unlock	Gebe Sperre frei

Abbildung 2.30: Einige der Pthreads-Aufrufe im Zusammenhang mit Mutexen.

Mutexe in Pthreads (2)

Thread-Anruf	Beschreibung
Pthread_cond_init	Erzeuge eine Zustandsvariable
Pthread_cond_destroy	Zerstöre vorhandene Zustandsvariable
Pthread_cond_wait	Blockiere, um auf Signal zu warten
Pthread_cond_signal	Sende Signal an anderen Thread, um ihn aufzuwecken
Pthread_cond_broadcast	Sende Signale an mehrere Threads, um alle aufzuwecken

Abbildung 2.31: Einige der Pthread-Aufrufe im Zusammenhang mit Zustandsvariablen.

Nutzung von Threads zur Lösung des Erzeuger-Verbraucher-Problems (1)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000

pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;

int buffer = 0;

void *producer(void *ptr)
{ int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);           /* wie viele Nummern sollen */
                                                /* erzeugt werden? */

        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);   /* genutzt zur Signalisierung */
                                                                /* durch Zustandsvariablen */
                                                                /* Puffer zwischen Erzeuger */
                                                                /* und Verbraucher*/

        buffer = i;                                /* erzeuge Daten */

        pthread_cond_signal(&condc);               /* erlange exklusiven Zugriff */
                                                /* auf Puffer */

        pthread_mutex_unlock(&the_mutex);          /* lege Element in Puffer */
                                                /* wecke Verbraucher auf */
                                                /* gib Zugriff auf Puffer frei */

    }
    pthread_exit(0);
}
```

Nutzung von Threads zur Lösung des Erzeuger-Verbraucher-Problems (2)

```
void *consumer(void *ptr)                                /* verbrauche Daten */  
{   int i;  
  
    for (i = 1; i <= MAX; i++) {  
        pthread_mutex_lock(&the_mutex);                  /* erlange exklusiven Zugriff */  
        /* auf Puffer */  
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);  
        buffer = 0;  
        pthread_cond_signal(&condp);                     /* entnimm Element aus Puffer */  
        /* wecke Erzeuger auf */  
        pthread_mutex_unlock(&the_mutex);                 /* gib Zugriff auf Puffer frei */  
    }  
    pthread_exit(0);  
}  
  
int main(int argc, char **argv)  
{  
    pthread_t pro, con;  
    pthread_mutex_init(&the_mutex, 0);  
    pthread_cond_init(&condc, 0);  
    pthread_cond_init(&condp, 0);  
    pthread_create(&con, 0, consumer, 0);  
    pthread_create(&pro, 0, producer, 0);  
    pthread_join(pro, 0);  
    pthread_join(con, 0);  
    pthread_cond_destroy(&condc);  
    pthread_cond_destroy(&condp);  
    pthread_mutex_destroy(&the_mutex);  
}
```

Abbildung 2.32: Die Benutzung von Threads zur Lösung des Erzeuger-Verbraucher-Problems.

```
monitor example
    i;
    condition c;
    procedure producer( );
    .
    .
    .
end;

procedure consumer( );
.
.
.
end;
end monitor;
```

Monitor (1)

Abbildung 2.33: Ein Monitor.

Monitor (2)

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
```

Monitor (3)

```
count := count - 1;
if count = N - 1 then signal(full)
end;

count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
end;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
end;
```

Abbildung 2.34: Eine Skizze des Erzeuger-Verbraucher-Problems mit Monitoren. Es ist jeweils nur eine Monitorprozedur aktiv. Der Puffer hat N Einträge.

```
public class ProducerConsumer {  
    static final int N = 100;                                // konstante Puffergröße  
    static producer p = new producer( );                      // instanziere einen neuen  
    static consumer c = new consumer( );                      // Erzeuger-Thread  
    static our_monitor mon = new our_monitor( );              // instanziere einen neuen  
                                                               // Verbraucher-Thread  
                                                               // instanziere einen  
                                                               // neuen Monitor  
  
    public static void main(String args[ ]) {  
        p.start( );          // starte den Erzeuger-Thread  
        c.start( );          // starte den Verbraucher-Thread  
    }  
  
    static class producer extends Thread {  
        public void run( ) { // run-Methode enthält den Thread-Code  
            int item;  
            while (true) {   // Schleife des Erzeugers  
                item = produce_item( );  
                mon.insert(item);  
            }  
        }  
        private int produce_item( ) { ... }           // erzeuge etwas  
    }  
}
```

Monitor (5)

```
static class consumer extends Thread {  
    public void run( ) { // run-Methode enthält den Thread-Code  
        int item;  
        while (true) { // Schleife des Verbrauchers  
            item = mon.remove( );  
            consume_item (item);  
        }  
    }  
    private void consume_item(int item) { ... } // verbrauche etwas  
}  
  
static class our_monitor { // dies ist ein Monitor  
    private int buffer[ ] = new int[N];  
    private int count = 0, lo = 0, hi = 0; // Zähler und Indizes  
  
    public synchronized void insert(int val) {  
        if (count == N) go_to_sleep( ); // wenn Puffer voll, gehe schlafen  
        buffer [hi] = val; // lege ein Element in den Puffer  
        hi = (hi + 1) % N; // Index des Eintrags für nächstes Element  
        count = count + 1; // jetzt ist ein Element mehr im Puffer
```

Monitor (6)

```
if (count == 1) notify( );    // wenn Verbraucher schläft, dann wecke ihn auf
}

public synchronized int remove( ) {
    int val;
    if (count == 0) go_to_sleep( );    // wenn der Puffer leer ist, gehe schlafen
    val = buffer [lo];                // hole ein Element aus dem Puffer
    lo = (lo + 1) % N;                // Index des Eintrags für nächstes Element
    count = count - 1;                // jetzt ist ein Element weniger im Puffer
    if (count == N - 1) notify( );    // wenn Erzeuger schläft, dann wecke
                                    // ihn auf
    return val;
}
private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```

Abbildung 2.35: Eine Lösung des Erzeuger-Verbraucher-Problems in Java.

```
#define N 100          /* Anzahl der Einträge im Puffer */

void producer(void)
{
    int item;
    message m;           /* Nachrichtenpuffer */

    while (TRUE) {
        item = produce_item();      /* erzeuge etwas und lege es in den Puffer */
        receive(consumer, &m);     /* warte auf einen leere Nachricht */
        build_message(&m, item);   /* erzeuge Nachricht zum Versenden */
        send(consumer, &m);        /* sende Element zum Verbraucher */
    }
}

void consumer(void)
{
    int item;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* sende N leere Nachrichten */
    while (TRUE) {
        receive(producer, &m);      /* empfange Nachricht mit Element */
        item = extract_item(&m);    /* hole Element aus der Nachricht */
        send(producer, &m);         /* sende leere Antwortnachricht zurück */
        consume_item(item);        /* benutze das Element */
    }
}
```

Abbildung 2.36: Das Erzeuger-Verbraucher-Problem mit N Nachrichten.

Barrieren

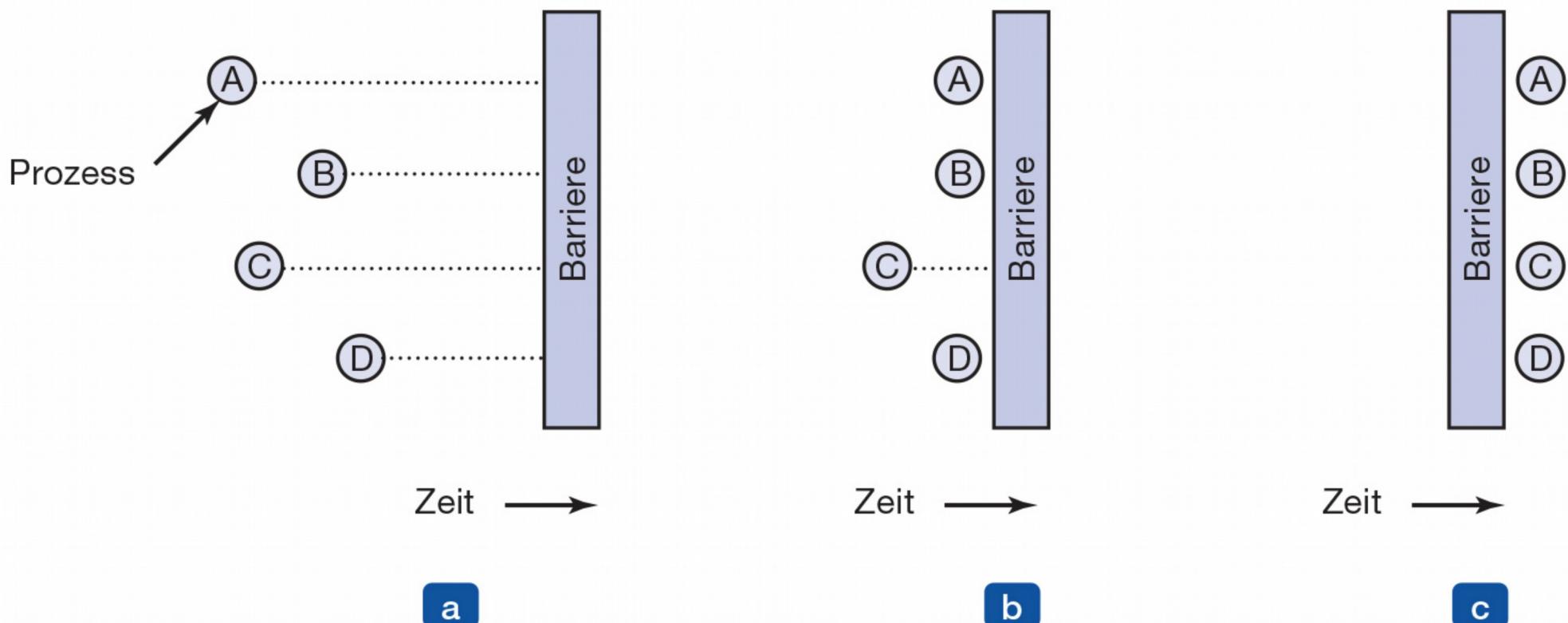
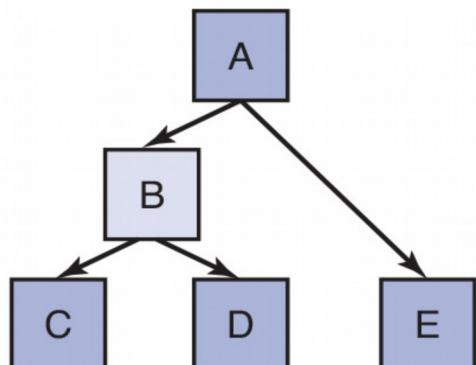


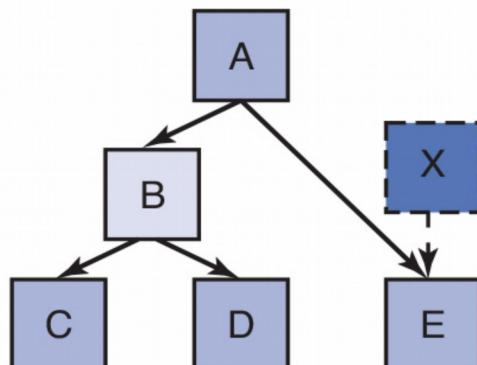
Abbildung 2.37: Verwendung einer Barriere: (a) Prozess nähert sich einer Barriere. (b) Alle Prozesse bis auf einen sind an der Barriere blockiert. (c) Wenn der letzte Prozess ankommt, werden alle durchgelassen.

Sperrvermeiden: das Read-Copy-Update-Schema

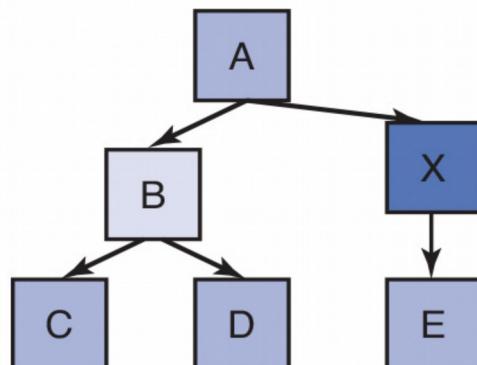
Hinzufügen eines Knotens:



a Ausgangsbbaum.

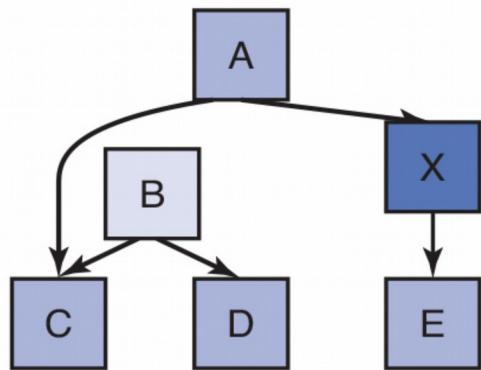


b Initialisiere Knoten X und verbinde E mit X. Leser von A und E sind nicht betroffen.

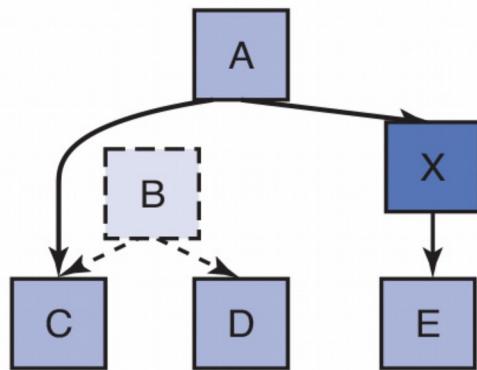


c Wenn X vollständig initialisiert ist, verbinde X mit A. Aktuelle Leser von E lesen die alte Version, während Leser von A die neue Version des Baums lesen.

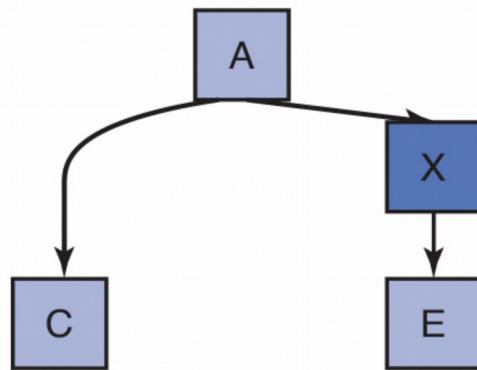
Entfernen von Knoten:



d Enkoplele B von A. Achtung, es könnte noch Leser von B geben. Alle Leser von B werden die alte Version des Baums sehen, alle aktuellen Leser von A die neue Version.



e Warten, bis wir sicher sind, dass alle Leser B und C verlassen haben. Auf diese Knoten kann nicht mehr zugegriffen werden.



f Jetzt können wir B und D sicher entfernen.

2.4 Scheduling

- 2.4.1 Einführung in das Scheduling
- 2.4.2 Scheduling in Stapelverarbeitungssystemen
- 2.4.3 Scheduling in interaktiven Systemen
- 2.4.4 Scheduling in Echtzeitsystemen
- 2.4.5 Strategie versus Mechanismus
- 2.4.6 Thread-Scheduling

Einführung in das Scheduling – das Verhalten von Prozessen

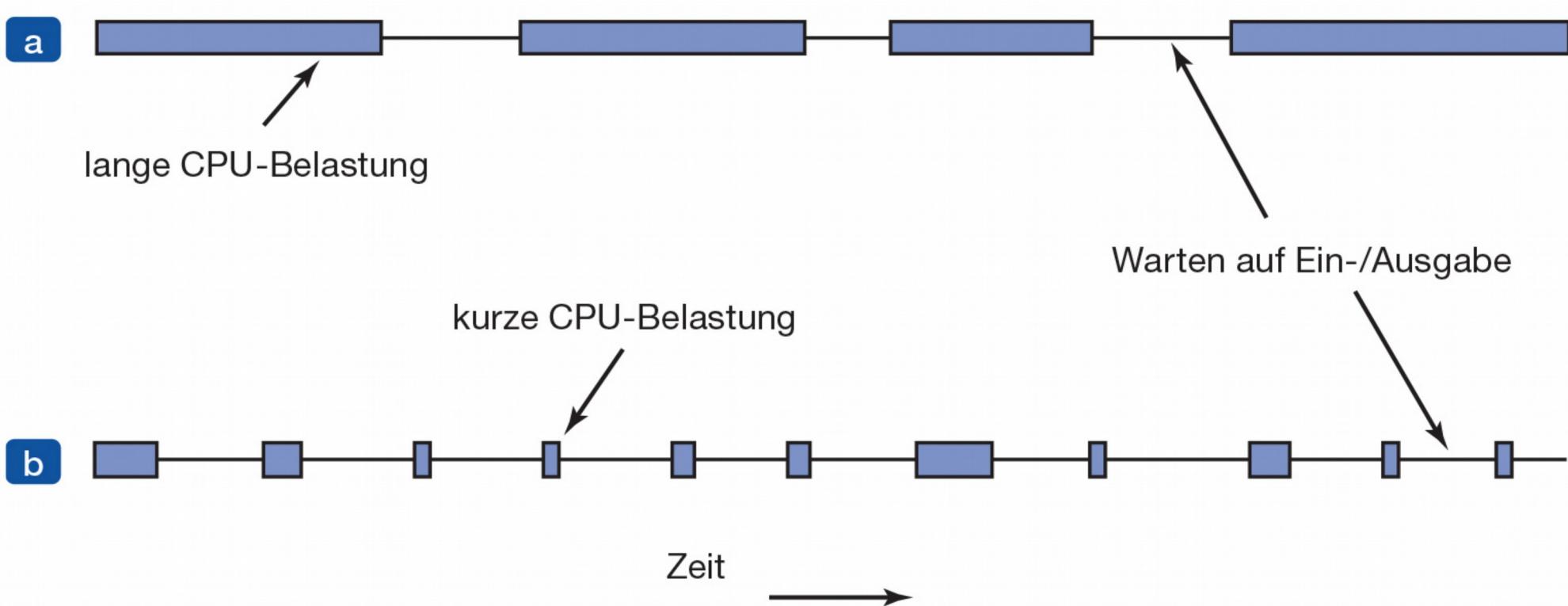


Abbildung 2.39: Häufung von CPU-Gebrauch alternierend mit Zeiträumen, in denen auf E/A gewartet wird: (a) ein CPU-intensiver Prozess; (b) ein E/A-intensiver Prozess.

Kategorien von Scheduling-Algorithmen

1. Batch (Stapelverarbeitung)
2. Interactive (Interaktiv)
3. Real time (Echtzeit)

Ziele von Schedulingstrategien

Alle Systeme

Fairness - jeder Prozess bekommt einen fairen Anteil der Rechenzeit

Policy Enforcement - vorgegebene Strategien werden durchgesetzt

Balance - alle Teile des Systems sind ausgelastet

Stapelverarbeitungssysteme

Durchsatz – Maximieren der Jobs pro Stunde

Durchlaufzeit – Minimieren der Zeit vom Start bis zur Beendigung

CPU – die CPU ist immer ausgelastet

Interaktive Systeme

Antwortzeit – schnelle Antwort auf Anfragen

Proportionalität – Erwartungen des Benutzers erfüllen

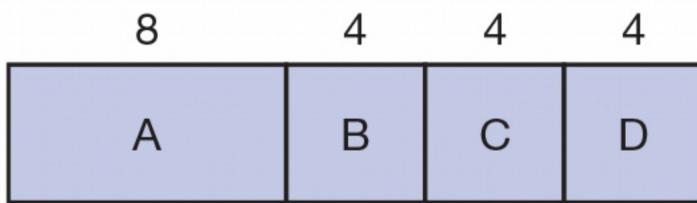
Echtzeitsysteme

Deadlines einhalten – Datenverlust vermeiden

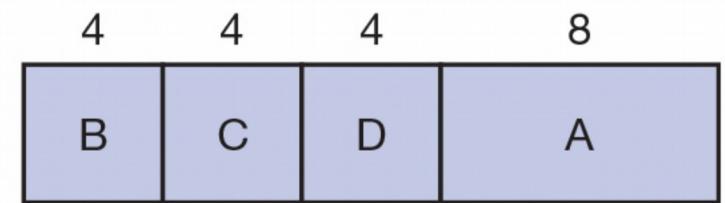
Vorhersagbarkeit – Qualitätseinbußen in Multimediasystemen vermeiden

Abbildung 2.40: Einige Ziele von Schedulingstrategien in verschiedenen Umgebungen.

Shortest-Job-First



a



b

Abbildung 2.41: Ein Beispiel für Shortest-Job-First-Scheduling: (a) Ablauf der vier Jobs in der Originalfolge. (b) Ablauf in der Shortest-Job-First-Reihenfolge.

Tanenbaum, A. S.; Bos, H.: Moderne
Betriebssysteme. Pearson Studium 2016

Scheduling in Batch-Systemen

1. First-Come First-Served („Wer zuerst kommt, mahlt zuerst“)*
2. Shortest Job First (Kürzester Job zuerst)
3. Shortest Remaining Time Next (Kürzeste verbleibende Zeit als Nächster)

* Ursprung des deutschen Textes: Der Sachenspiegel, ein Rechtsbuch des Eike von Repgow (zwischen 1220 und 1235 entstanden). Es gilt als eines der bedeutendsten und ältesten Rechtsbücher des deutschen Mittelalters. Gültig in Sachsen bis 1865, in Holstein, Anhalt und Thüringen als subsidiäre Rechtsquelle bis zur Ablösung durch das BGB 1900.

Scheduling in interaktiven Systemen

1. Round-Robin Scheduling (Reihum Scheduling)
2. Priority Scheduling (Priorität)
3. Multiple Queues (Mehrere Warteschlangen)
4. Shortest Process Next (Kürzester Prozess als Nächster)
5. Guaranteed Scheduling (Garantiertes Scheduling)
6. Lottery Scheduling (Zufälliges Scheduling)
7. Fair-Share Scheduling (gerechter Anteil)

Round-Robin Scheduling

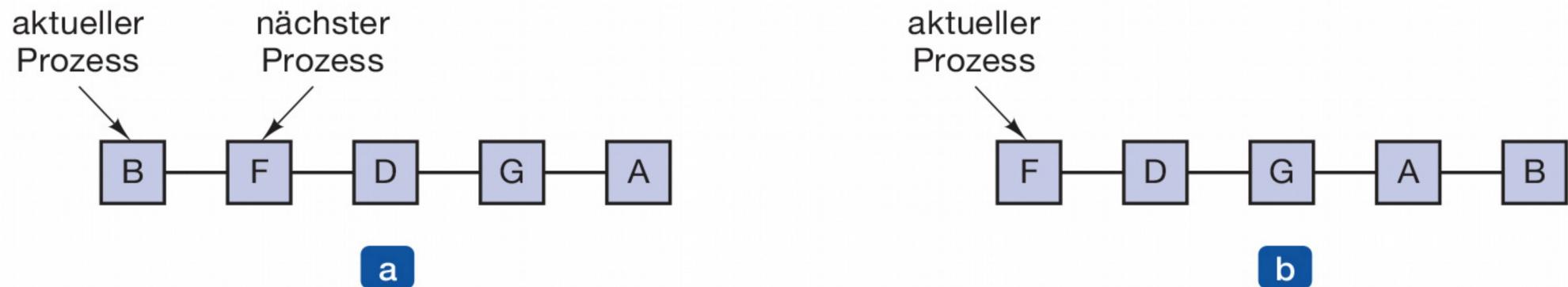


Abbildung 2.42: Round-Robin-Scheduling: (a) Liste der lauffähigen Prozesse; (b) Liste der lauffähigen Prozesse, nachdem B sein Quantum aufgebraucht hat.

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

Priority Scheduling

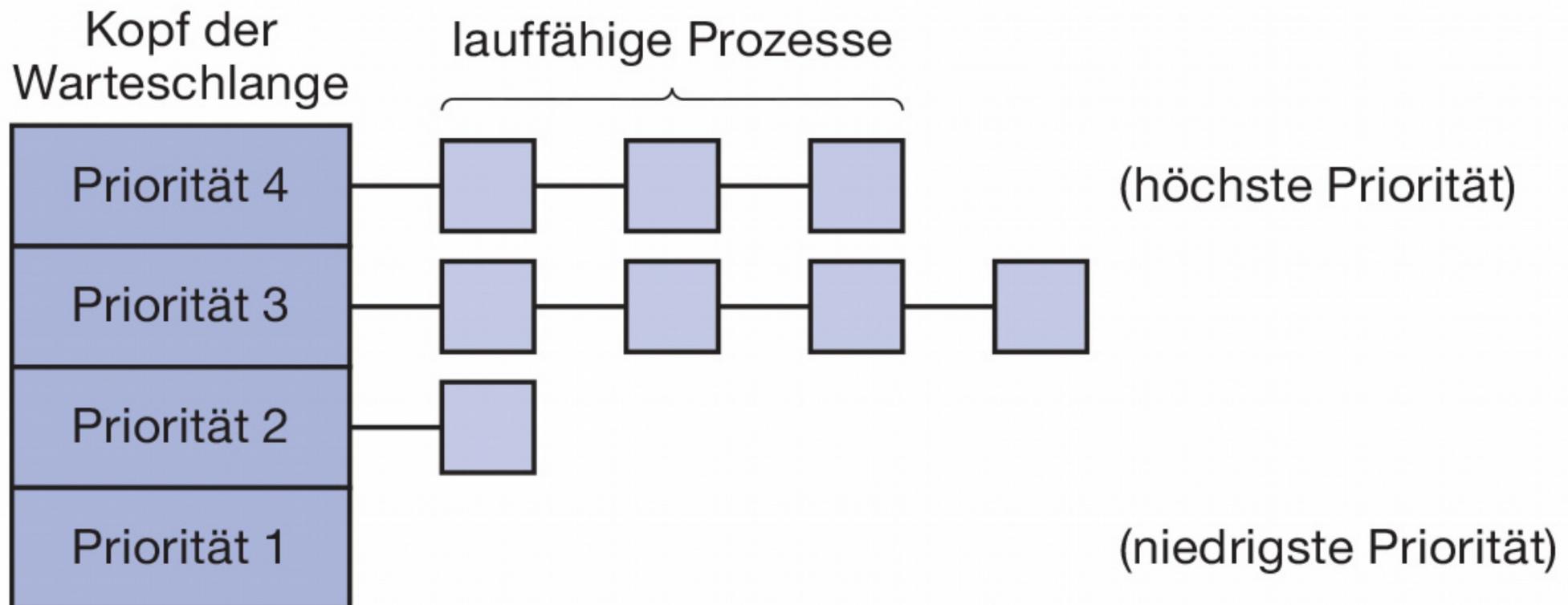


Abbildung 2.43: Eine Schedulingstrategie mit vier Prioritätsklassen.

Tanenbaum, A. S.; Bos, H.: Moderne
Betriebssysteme. Pearson Studium 2016

Scheduling in Real-Time Systems

1. Zeit spielt eine wesentliche Rolle

2. Kategorien

- (Harte) Echtzeit
- „Soft real time“ → ist laut Def. keine Echtzeit! *
- Periodisch oder Aperiodisch

3. Scheduling muss erfüllen:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

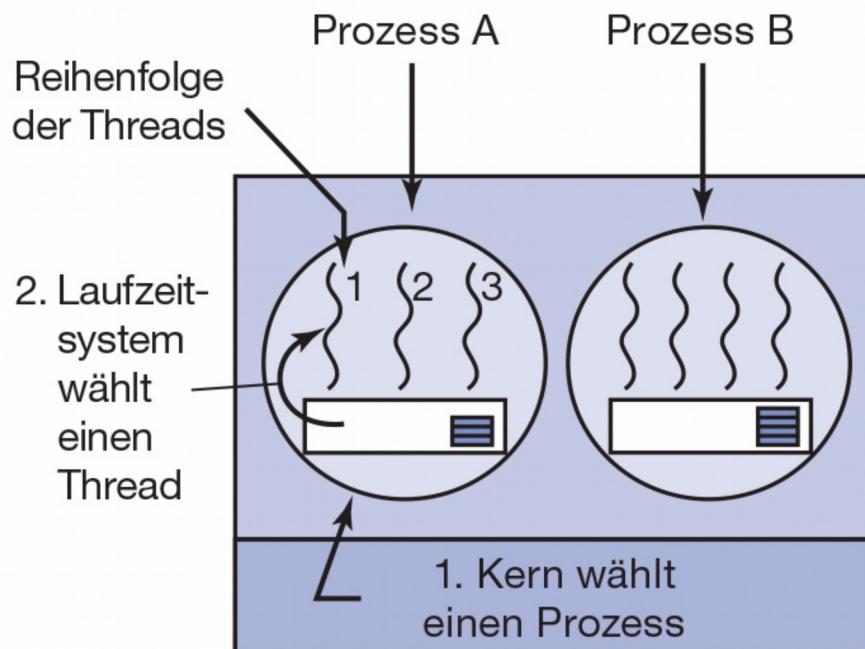
P_i : Intervall des Auftretens

C_i : Benötigte CPU Zeit

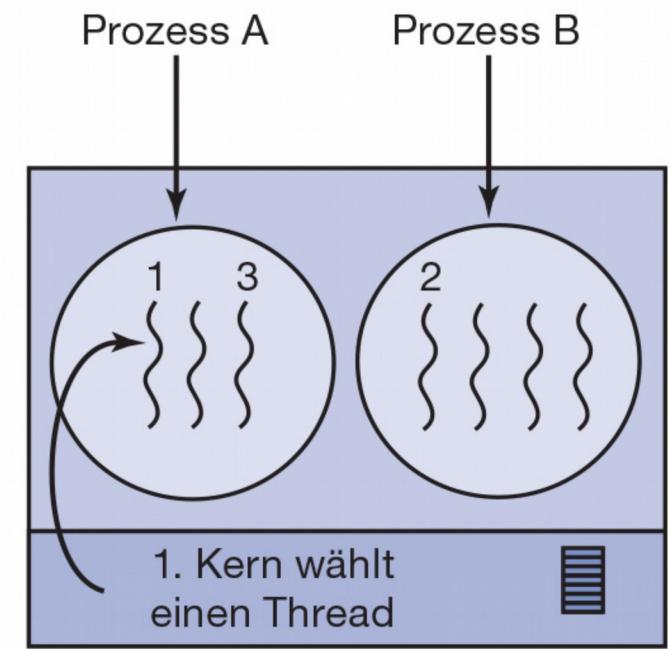
i : Ereignis/Task

* Echtzeitsysteme sind „Systeme zur unmittelbaren Steuerung und Abwicklung von Prozessen“, die quantitative Echtzeitanforderungen erfüllen müssen. Eine häufige Anforderung ist, dass ein Ergebnis innerhalb eines vorher fest definierten Zeitintervalles garantiert berechnet ist. Die Größe des Zeitintervalles spielt dabei keine Rolle.

Thread Scheduling



a



b

Abbildung 2.44: (a) Ein möglicher Ablauf von Threads auf Benutzerebene mit 50-ms-Prozess-Quantum und Threads, die 5 ms pro CPU-Zuteilung laufen. (b) Ein möglicher Ablauf von Threads auf Kernebene, mit denselben Charakteristika wie in (a).

2.5 Klassische Probleme der Interprozesskommunikation

2.5.1 Das Philosophenproblem

2.5.2 Das Leser-Schreiber-Problem

Das Philosophenproblem

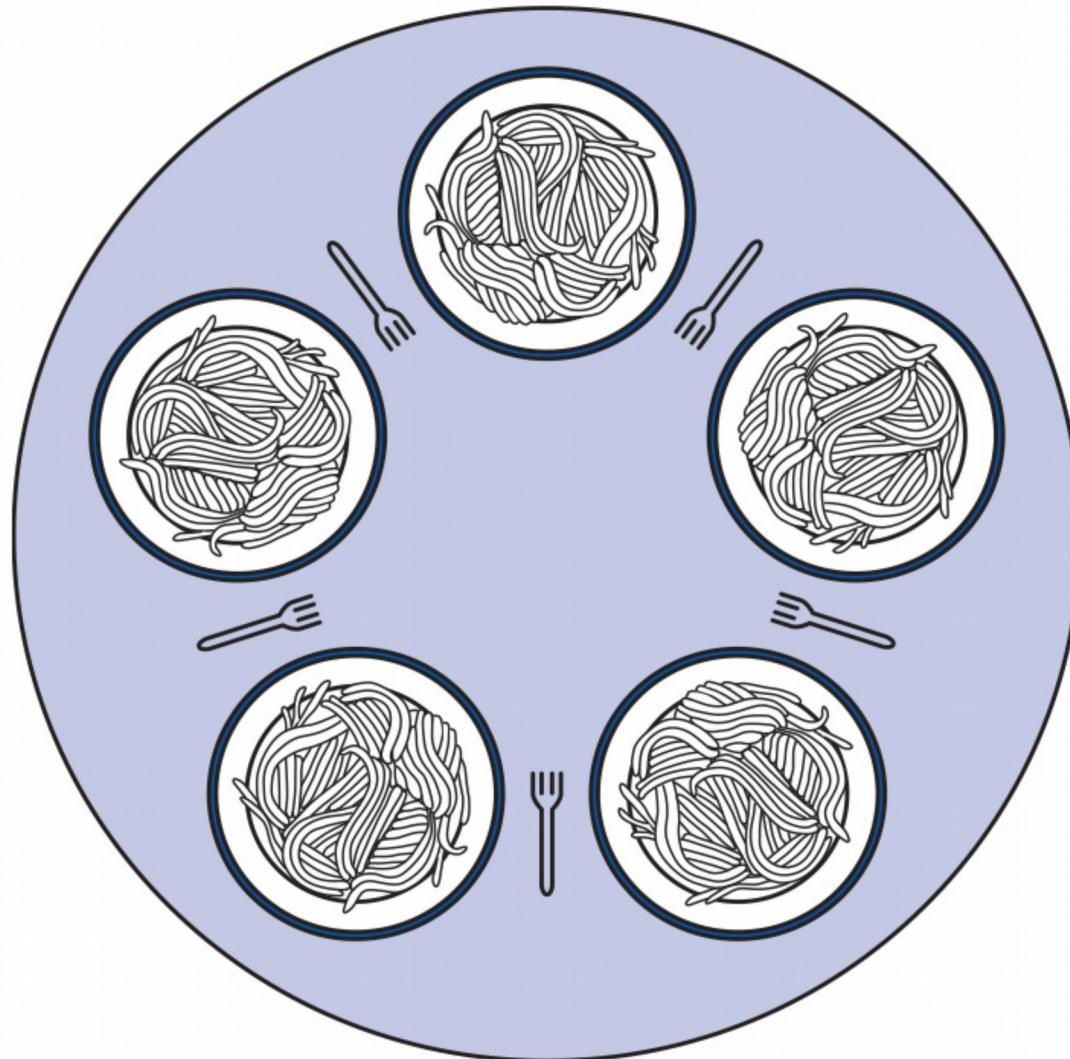


Abbildung 2.45: Mittagessen bei den Philosophen.

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

Eine falsche Lösung für das Philosophenproblem

```
#define N 5                                     /* Anzahl der Philosophen */\n\nvoid philosopher(int i)                      /* i: Nummer des Philosophen (von 0 bis 4)*/\n{\n    while (TRUE) {\n        think();\n        take_fork(i);\n        take_fork((i+1) % N);\n        eat();\n        put_fork(i);\n        put_fork((i+1) % N);\n    }\n}
```

Abbildung 2.46: Eine falsche „Lösung“ für das Philosophenproblem.

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

Eine Lösung für das Philosophenproblem (1)

```
#define N 5 /* Anzahl der Philosophen */  
#define LEFT (i+N-1)%N /* Nummer von i's linkem Nachbarn */  
#define RIGHT (i+1)%N /* Nummer von i's rechtem Nachbarn */  
#define THINKING 0 /* Philosoph denkt */  
#define HUNGRY 1 /* Philosoph versucht Gabeln zu bekommen */  
#define EATING 2 /* Philosoph isst */  
  
typedef int semaphore;  
int state[N];  
semaphore mutex = 1;  
  
semaphore s[N];  
  
void philosopher (int i) /* i: Nummer des Philosophen (von 0 bis N-1) */  
{  
    while (TRUE) {  
        think();  
        take_forks(i);  
        eat();  
        put_forks(i);  
    }  
}
```

/* Semaphore sind spezielle Integervariablen */
/* Feld mit dem Status jedes Philosophen */
/* wechselseitiger Ausschluss für */
/* kritische Regionen */
/* ein Semaphor pro Philosoph */

/* Endlosschleife */
/* Philosoph denkt */
/* nimm zwei Gabeln oder blockiere */
/* mjam, mjam, Spaghetti */
/* lege beide Gabeln zurück auf den Tisch */

Eine Lösung für das Philosophenproblem (2)

```
void take_forks (int i)                                /* i: Nummer des Philosophen (von 0 bis N-1) */  
{  
    down(&mutex);  
    state[i] = HUNGRY;  
    test(i);  
    up(&mutex);  
    down(&s[i]);  
}  
  
void put_forks (i )                                    /* i: Nummer des Philosophen (von 0 bis N-1) */  
{  
    down(&mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    up(&mutex);  
}  
  
void test (i)                                         /* i: Nummer des Philosophen (von 0 bis N-1) */  
{  
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
        state[i] = EATING;  
        up(&s[i]);  
    }  
}
```

Abbildung 2.47: Eine Lösung für das Philosophenproblem.

Das Leser-Schreiber-Problem

Tanenbaum, A. S.; Bos, H.: Moderne Betriebssysteme. Pearson Studium 2016

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader (void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer (void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

/* Raten Sie mal!
 * steuert Zugriff auf rc
 * steuert Zugriff auf die Datenbank
 * Anzahl lesende u. lesebereite Prozesse */

/* Endlosschleife */
/* exklusiver Zugriff auf rc */
/* ein Leser mehr */
/* wenn dies der erste Leser ist ... */
/* exklusiven Zugriff auf rc freigeben */
/* Zugriff auf die Daten */
/* exklusiver Zugriff auf rc */
/* ein Leser weniger */
/* wenn dies der letzte Leser war ... */
/* exklusiven Zugriff auf rc freigeben */
/* nicht kritische Region */

/* Endlosschleife */
/* nicht kritische Region */
/* exklusiver Zugriff */
/* schreibe die neuen Daten */
/* exklusiven Zugriff freigeben */
```

Abbildung 2.48: Eine Lösung für das Leser-Schreiber-Problem.