Betriebssysteme
Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 6: Speicherbasierte Prozessinteraktion

Felix C. Freiling
Lehrstuhl für Praktische Informatik 1
Universität Mannheim

Motivation

- Bisher kennengelernt: Basisabstraktionen für Speicher (Adressraum) und Prozessor (Thread)
- Threads treten oft in Wechselwirkung zueinander
- Variante 1: Konkurrenz
  - Zeitliche Abstimmung nebenläufiger Threads
  - Beispiel: Zugriff auf exklusive Betriebsmittel
- Variante 2: Kooperation
  - Gezielter Informationsaustausch zwischen Threads
  - Beispiel: Clients erhalten Aufträge vom Server
- Welche Abstraktionen sind für Prozessinteraktion sinnvoll und wie implementiert man sie?
  - Konzentration auf Realisierungen mit gemeinsamen Speicher, also für Laufzeit-Basismodelle A, C, D aus Kapitel 3

Positionsbestimmung

- Gliederung der Vorlesung:
  1. Einführung und Formalia
  2. Auf was baut die Systemsoftware auf?
     Hardware-Grundlagen
  3. Was wollen wir eigentlich haben?
     Laufzeitunterstützung aus Anwendersicht
  4. Verwaltung von Speicher: Virtueller Speicher
  5. Verwaltung von Rechenzeit: Virtuelle Prozessoren (Threads)
  6. **Synchronisation paralleler Aktivitäten auf dem Rechner**
     - **4 Wochen**
  7. Implementierungsaspekte

Übersicht

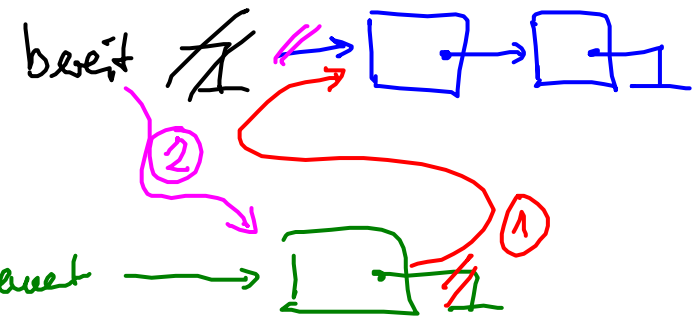
- **Einführung: Kritische Abschnitte**
- Hardwaregestützte Mechanismen
- Betriebssystemgestützter Mechanismus: Semaphore
- Sprachgestützter Mechanismus: Monitore
- Realisierungsbeispiele

Gemeinsame Datenstrukturen

- Beispiel aus der Implementierung eines Dispatchers:
  - Angenommen die Bereit-Liste wird durch eine einfach-verkettete Liste von TCBs implementiert
    - Jeder TCB enthält `next`-Feld mit Zeiger auf das nächste Listenelement (`NULL` falls Liste zuende)

```
TCB bereit = NULL  // Initialisierung

bereit.Put(TCB element) { // fügt vorne an
  element->next = bereit;
  bereit = element;
}
```



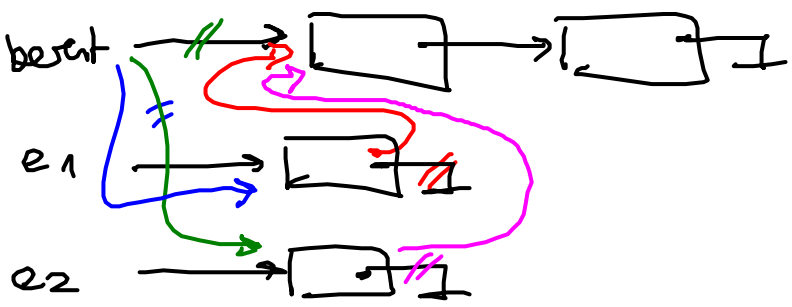
- Behauptung: Wenn mehrere Threads "gleichzeitig" auf der Liste hantieren, können Elemente verloren gehen

Korrektheit unter Nebenläufigkeit

- Gleichzeitigkeit = Nebenläufigkeit im Zeitmultiplex mit Interrupts
- Beispiel: Zwei Threads A und B mit Timer Interrupt
  - Beide blockiert, werden nacheinander durch Interrupts deblockiert

```
Prozess A                          Prozess B

// put e1 into bereit              // put e2 into bereit
1  e1->next = bereit;
                                   2  e2->next = bereit;
                                   3  bereit = e2;
4  bereit = e1;
```



Kritischer Abschnitt

- Das Einfügen eines Elements in die Liste ist ein kritischer Abschnitt
  - Kritischer Abschnitt = Abschnitt im Code eines Prozesses, in dem gemeinsame Ressourcen bearbeitet werden
- Die Manipulation von gemeinsamen Datenstrukturen wird durch ein Eintritts- und ein Austrittsprotokoll geschützt
  - Einfachste Variante: Synchronisation über eine gemeinsame Speicherzelle (Sperrflag, Bit), das entweder `frei` oder `belegt` ist
  - Programmierung:

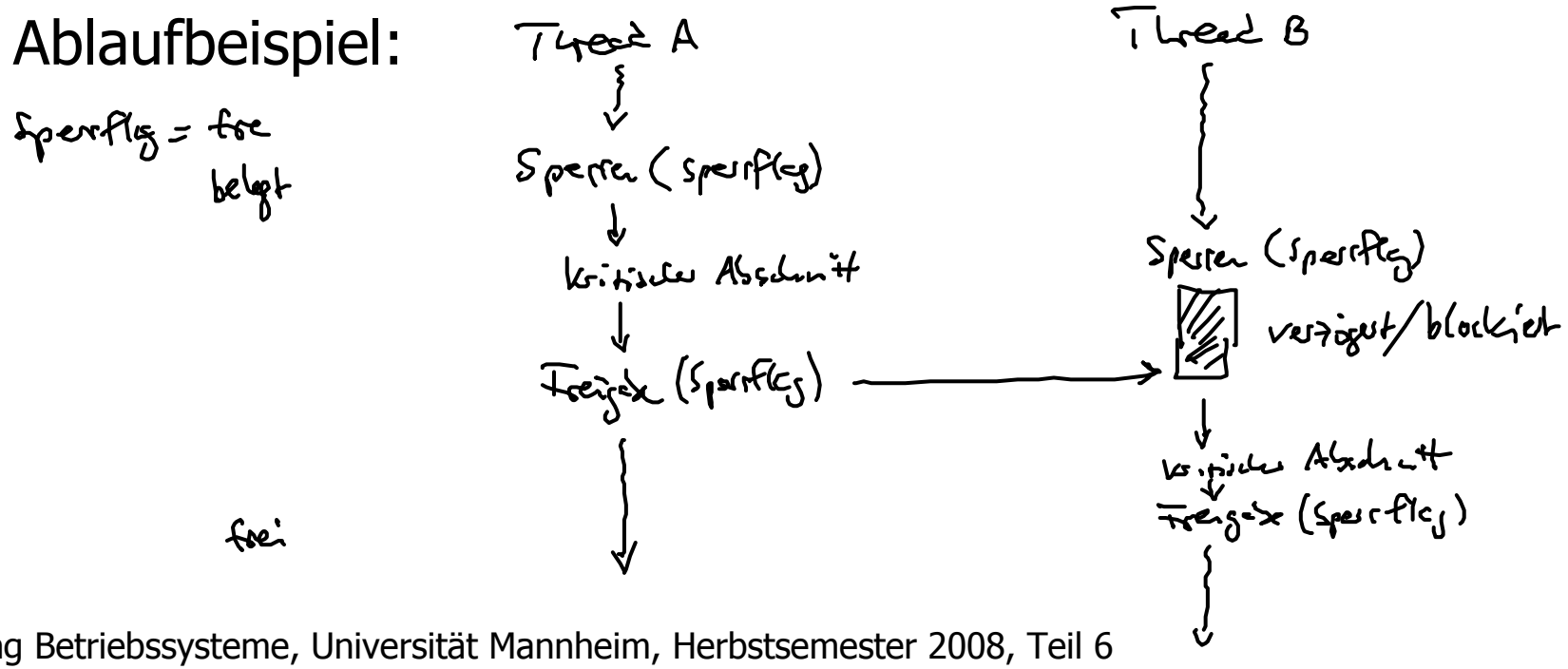
```
...
Sperren(flag)
// kritischer Abschnitt
Freigabe(flag)
...
```

  - Operationen Sperren und Freigabe realisieren die Eintritts- und Austrittsprotokolle

Wechselseitiger Ausschluss

- Die Operationen `Sperren` und `Freigabe` müssen folgende Bedingung garantieren
  - Wechselseitiger Ausschluss (mutual exclusion):

    Zu jedem Zeitpunkt ist maximal ein Prozess in seinem kritischen Abschnitt
  - Ein belegtes Sperrflag soll nachfolgende Prozesse blockieren
    - Wir vernachlässigen vorerst das Konzept der Fairness
- Ablaufbeispiel:



Beispiel: Bereit-Liste

- Die Implementierung der Bereit-Liste müsste dann wie folgt aussehen:

```
Bit bereit_flag = frei;

TCB bereit = NULL  // Initialisierung

bereit.Put(TCB element) { // fügt vorne an
  Sperren(bereit_flag); // Referenzparameter!
  element->next = bereit;
  bereit = element;
  Freigabe(bereit_flag);
}
```

- Universelles Problem: Tritt überall dort auf, wo es kritische Abschnitte gibt
- Die Frage jetzt: Wie implementiere ich die Eintritts- und Austrittsprotokolle?

Übersicht

- Einführung: Kritische Abschnitte
- **Hardwaregestützte Mechanismen**
  - Unterbrechungssperren
  - Spezielle Hardware-Befehle
- Betriebssystemgestützter Mechanismus: Semaphore
- Sprachgestützter Mechanismus: Monitore
- Realisierungsbeispiele

Atomare Operationen

- Implementierung von Sperren und Freigabe setzt Kenntnisse über die verwendete Hardware voraus
- Wichtigste Frage: Welche Operationen werden garantiert ununterbrochen (atomar) durchgeführt?
- Beispiel: Zuweisung `x=y;` in Java oder C

TAS

  - In der Regel nicht atomar: Wird in eine Reihe von Lade und Speicherbefehlen übersetzt:
    - `LOAD Register, Y`
    - `STORE X, Register`
  - Dazwischen kann eine Unterbrechung erfolgen
- Prozessoren besitzen in der Regel atomare Befehle, die speziell zur Implementierung von `Sperren` und `Freigabe` geeignet sind

Sperren der Interrupts

- Problem: Unterbrechungen im kritischen Abschnitt
- Lösung: Sperren der Interrupts
  - Kritischer Abschnitt kann jetzt einfach implementiert werden:

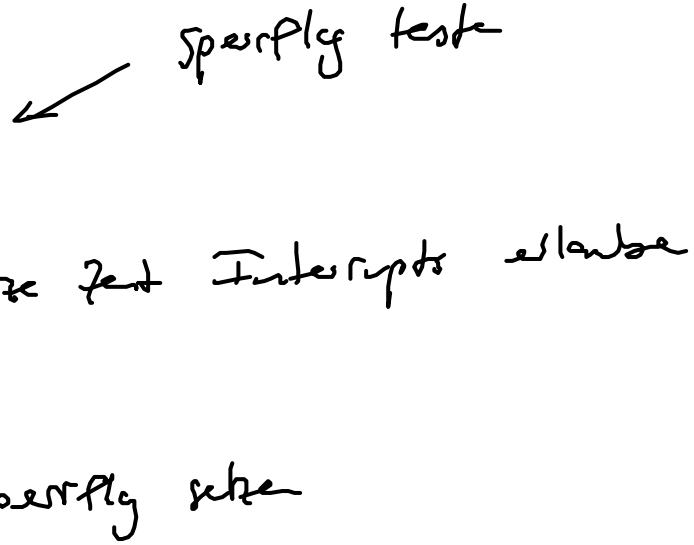
```
...
Disable; // Interrupts off
kritischer Abschnitt
Enable; // Interrupts on
...
```

- Eigenheiten:
  - Darf nur im Supervisor Mode gemacht werden
  - Gefahr bei langem Ausschalten der Interrupts
  - Funktioniert nicht bei Mehrprozessorsystemen
- Sollte nur bei relativ kurzen kritischen Abschnitten im Betriebssystemkern gemacht werden

Verbesserung mit Sperrflag

- Kritische Abschnitte können ausgedehnt werden, wenn man nur den Zugriff auf ein Sperrflag per `Enable/Disable` schützt

```
Bit Sperrflag = frei;
Sperren() {
  Disable;
  while (Sperrflag == belegt) {
    Enable;
    NOP;
    Disable;
  }
  Sperrflag = belegt;
  Enable;
}
Freigabe() {
  Sperrflag = frei;
}
```



Test&Set/Lock

- Viele Prozessoren bieten eine spezielle Hardware-Instruktion an, mit der man ohne `Enable` und `Disable` auskommt
- Beispiel: Test&Set
  - Globales Sperrflag `busy` initial auf `false`
  - `Test&Set(busy, local)`
    - Kopiert Sperrflag `busy` nach `local` (Test) und setzt anschliessend das Sperrflag (Set)
  - Pseudocode:

```
TAS(busy, local) =
  [[ local = busy;  busy = true; ]]
```

  - Man kann anschliessend durch Überprüfen von `local` herausfinden, ob man "der erste" war

[[...]] bedeutet: wird atomar von Hardware durchgeführt

- Anderes Beispiel: Lock

```
Bit Lock() =
  [[ tmp = busy; busy = true; return tmp; ]]
```

Implementierung

- Jetzt kann man Sperren und Freigabe ohne Rückgriff auf privilegierte Operationen implementieren

```
Bit busy = false;
Sperren() { // Variante mit TAS
  repeat
    TAS(busy, local);   // frei
  until (local == false);
}
Freigabe() {   // frei
  busy = false;
}

Sperren() { // Variante mit Lock()
  while (Lock() == true) NOP;
}
Freigabe() {
  busy = false;
}
```

Zusammenfassung

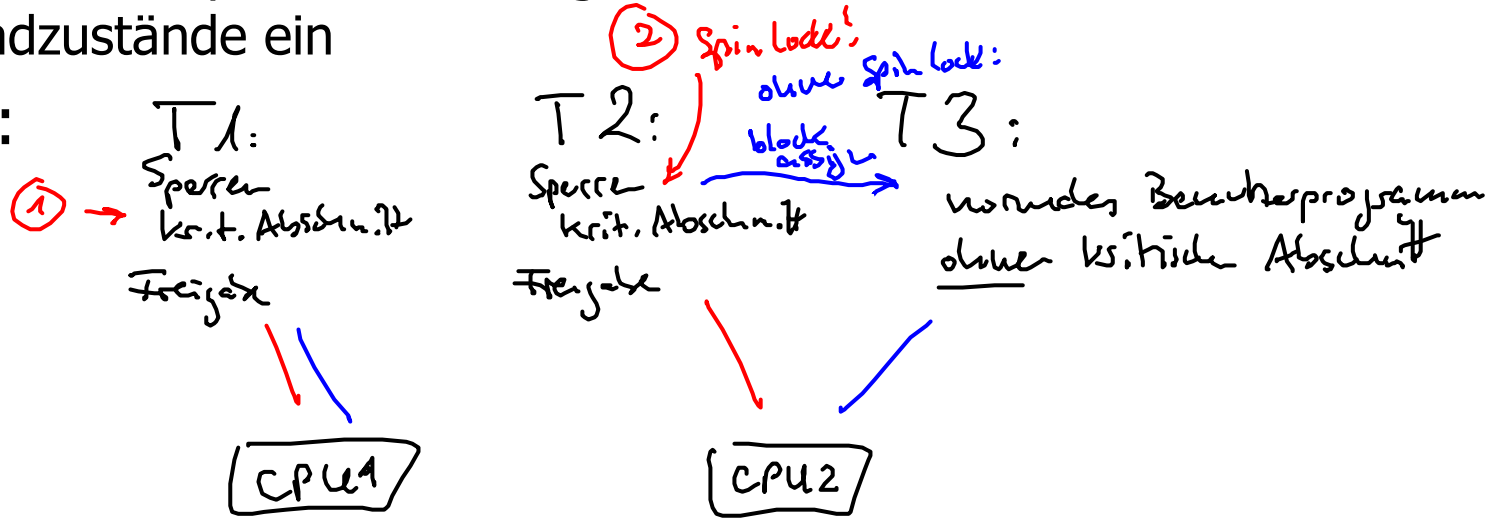
- Konstruktion mit TAS/Lock() wird als **Spin Lock** bezeichnet
  - Spin Lock ist eine "erlaubte Form aktiven Wartens"
  - Erlaubt nur dann, wenn kritische Abschnitte generell sehr kurz sind
- Auf Monoprozessorsystemen genügt auf Betriebssystemebene das Ausschalten der Interrupts
  - Spin Lock wird nicht benötigt
- Auf Multiprozessorsystemen muss man manchmal Spin Locks verwenden
  - Zunächst lokal auf dem Prozessor Interrupts ausschalten (lokaler Ausschluss)
  - Anschliessend im Spin Lock den globale Ausschluss erreichen
  - mehr in der Übung
- Auf höheren Ebenen kann man durch sinnvolle Abstraktionen aktives Warten (fast) ganz vermeiden

Übersicht

- Einführung: Kritische Abschnitte
- Synchronisation auf Basis atomarer Speicheroperationen
- Hardwaregestützte Mechanismen
- **Betriebssystemgestützter Mechanismus: Semaphore**
  - **Mächtige Programmierabstraktion**
  - **Implementierung vermeidet aktives Warten**
- Sprachgestützter Mechanismus: Monitore
- Realisierungsbeispiele

Aktives Warten

- Blockade kann immer durch aktives Warten implementiert werden
  - Aktives Warten ist ineffizient
    - Verbraucht Prozessorzyklen in einem Thread obwohl andere Threads laufbereit sind
- Wenn Betriebssystemunterstützung vorhanden ist, kann man aktives Warten durch Threadblockaden ersetzen
  - Operationen Sperren und Freigabe wirken dann direkt auf die Threadzustände ein
- Beispiel:



Semaphore

- Blockierende Synchronisationsmechanismen können verschiedene Semantiken haben
  - Wir schauen uns eine Abstraktion an, die in nahezu allen Betriebssystemen zur Verfügung steht: Semaphore
  - Ursprünglich in die Informatik eingeführt von Dijkstra
- Mit Semaphor bezeichnet man gewöhnlich ein "Formsignal", etwa bei Eisenbahnen
  - Analogie: kritische Abschnitte sind (z.B. eingleisige) Strecken im Eisenbahnnetz
  - Eintritt und Austritt müssen speziell gereget werden
  - Semaphore realisieren das Eintritts- und Austrittsprotokoll

Quelle: Wikipedia

Semantik von Semaphoren

- Sperren und Freigabe entspricht den Semaphor-Operationen P und V
  - P = Passieren = Thread will in den kritischen Abschnitt eintreten
  - V = Verlassen = Thread verlässt den kritischen Abschnitt
- Semaphore garantieren k-fach wechselseitigen Ausschluss
  - Es sind nie mehr als k Threads gleichzeitig im durch das Semaphor geschützten kritischen Abschnitt
  - Klassischer Wechselseitiger Ausschluss: k = 1
- Semaphor S wird mit k initialisiert
  - `P(S)` : Blockiere, falls bereits k Threads das Semaphor passiert und noch nicht verlassen haben
  - `V(S)` : Deblockiere den "nächsten" Thread, falls noch einer an einem `P(S)` blockiert ist

Beispiel: klassischer Mutex

- Mutex = wechselseitiger Ausschluss (mutual exclusion)
  - Üblicher Name des Semaphors zum Schutz eines klassischen kritischen Abschnitts
- Realisierungsbeispiel mit Semaphoren:

```
Semaphor Mutex = 1;

// Prozesscode
...
P(Mutex)
// kritischer Abschnitt
V(Mutex)
...
```

Implementierung von Semaphoren

- Standardimplementierung hat folgende Datenstrukturen:
  - Einen Zähler (integer, initialisiert mit einer nicht-negativen Zahl)
  - Eine Thread-Warteschlange (initialisiert als leere Schlange)
- Semaphorobjekte können wir uns zunächst auf Betriebssystemebene vorstellen
  - Warteschlange ist auf gleicher Abstraktionsebene wie bereit-Liste, blockiert-Liste etc.
- Beispielcode:

```
struct Semaphore {
  int zähler;
  List(PCB) liste;
}
new Semaphore(k) {
  zähler = k;
  liste = empty;
}
```

Implementierung von P und V

- Idee: Zählerstand gibt verbleibendes "Potential" des Semaphors an
  - Wieviele dürfen noch in den kritischen Abschnitt vor einer Blockade
- Für die Blockade kann man die Operationen des Dispatchers verwenden (in adaptierter Form)
  - `block(Queue q)` : markiert den aktuell laufenden Prozess als blockiert und reiht ihn in die Warteschlange `q` ein
  - ~~`ready`~~`(Queue q)` : deblockiert den nächsten Thread aus `q` und reiht ihn in die bereit-Liste ein
- Idee `P(S)`: Dekrementiere Zähler, blockiere auf Semaphor-Warteschlange falls Potential erschöpft
- Idee `V(S)`: Deblockiere einen ggf. blockierten Thread aus der Semaphor-Warteschlange, inkrementiere Zähler

Code-Beispiele

```
void P(Semaphor S) {
  S.zähler--;
  if (S.zähler < 0) {
    block(S.liste);
    assign();   // nächsten laufbereiten Thread
                // auf den Prozessor holen
  }
}

void V(Semaphor S) {
  if (S.zähler < 0) {
    deblock(S.liste);
  }
  S.zähler++;
}
```

Bemerkungen

- Semaphor-Warteschlange üblicherweise als FIFO-Liste realisiert
  - Deblockierung in der zeitlichen Reihenfolge, in der der Zugang zum kritischen Abschnitt beantragt wurde
- Logische Unteilbarkeit der Listenmanipulation
  - Am besten die Operationen P und V mit Betriebssystemmethoden für exklusiven Zugriff kapseln
  - Sperren und Freigabe auf unterster Ebene benutzen
    - Ausschalten der Interrupts
    - Zusätzlich Spin Lock bei Mehrprozessorsystemen

Gleichartige Betriebsmittel

- Einfacher Mutex ist Spezialform des k-fachen wechselseitigen Ausschlusses
  - Semaphor mit Wert k initialisieren
- Beispiel: Es gibt nur n DVD-Laufwerke
  - Falls mehr als n Threads ein DVD-Laufwerk benötigen, werden diese blockiert bis ein Laufwerk wieder frei wird
    - Implementieren über ein Semaphor, das mit n initialisiert wird
  - Eine gemeinsame Datenstruktur `Disc[n]` dokumentiert, welches Laufwerk frei oder belegt ist
  - Zugriff auf gemeinsame Datenstruktur muss wechselseitig ausgeschlossen geschehen
    - Implementieren über ein klassisches Mutex-Semaphor
- Codebeispiele für zwei Prozeduren:
  - `int GetDisc()` : allokiert ein freies DVD-Laufwerk
  - `void PutDisc(int i)` : gibt DVD-Laufwerk i wieder frei

Beispiel

```
Semaphore DiscSem = n, Mutex = 1;
enum {frei, belegt} Disc[n] = frei;

int GetDisc() {
  P(DiscSem);
  P(Mutex);
  int i = 1;
  while (Disc[i] == belegt) i++;
  Disc[i] = belegt;
  V(Mutex);
  return i;
}
void PutDisc(int i) {
  P(Mutex);
  Disc[i] = frei;
  V(Mutex);
  V(DiscSem);
}
```

maximal n Threads zwischen P(DiscSem) und V(DiscSem)

maximal ein Thread zwischen P(Mutex) und V(Mutex)

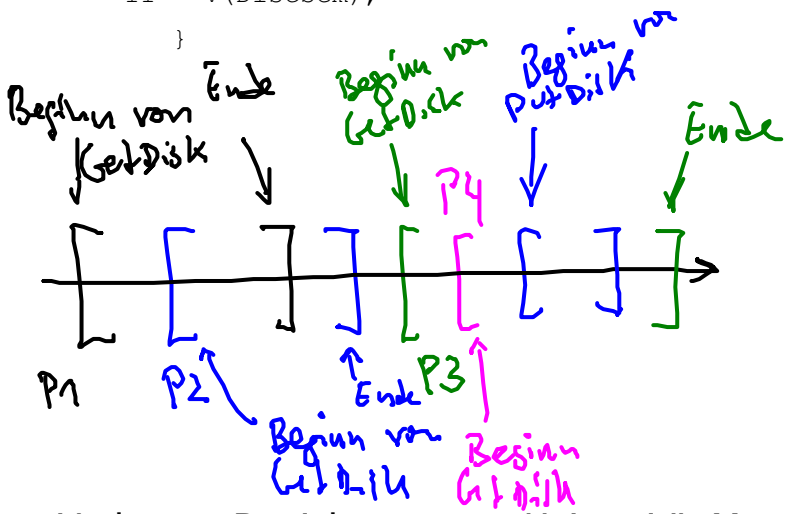
```
// Verwendung der
Operationen:

int DiscUnit;
...
DiscUnit = GetDisc();
// Verwendung von DiscUnit
PutDisc(DiscUnit);
```

Ablaufbeispiel

```
Semaphore DiscSem = 2, Mutex = 1;
enum {frei, belegt} Disc[n] = frei;

   int GetDisc() {
01   P(DiscSem);
02   P(Mutex);
03   int i = 1;
04   while (Disc[i] == belegt) i++;
05   Disc[i] = belegt;
06   V(Mutex);
07   return i;
   }
   void PutDisc(int i) {
08   P(Mutex);
09   Disc[i] = frei;
10   V(Mutex);
11   V(DiscSem);
   }
```



Disc [1]	Disc [2]	DiscSem z	DiscSem L	Mutex z	Mutex L	Wer	Wo	Was
frei	frei	2	<>	1	<>	P1	1	P(DiscSem)
"	"	1	<>	"	"	P2	1	P(DiscSem)
"	"	0	<>	"	"	P1	2	P(Mutex)
"	"	"	"	0	<>	P2	2	P(Mutex)
"	"	"	"	-1	<P2>	P1	3,4,5	
belegt	"	"	"	"	"	P1	6	V(Mutex)
"	"	"	"	0	<>	P2	3,4,5	
belegt	belegt	"	"	"	"	P1	7	return 1
"	"	"	"	"	"	P2	6	V(Mutex)
"	"	"	"	1	<>	P2	7	return 2
"	"	"	"	"	"	P3	1	P(DiscSem)
"	"	-1	<P3>	"	"	P4	1	P(DiscSem)
"	"	-2	<P3, P4>	"	"	P2	8	P(Mutex)
"	"	"	"	0	<>	P2	9	Disk [3] - frei
"	frei	"	"	"	"	P2	10	V(Mutex)
"	"	"	"	1	<>	P2	11	V(DiscSem)
"	"	-1	<P4>	"	"	P3	2	P(Mutex)
"	"	"	"	0	<>	P3	3,4,5	
"	belegt	"	"	"	"	P3	6	V(Mutex)
"	"	"	"	1	"	P3	7	return 2