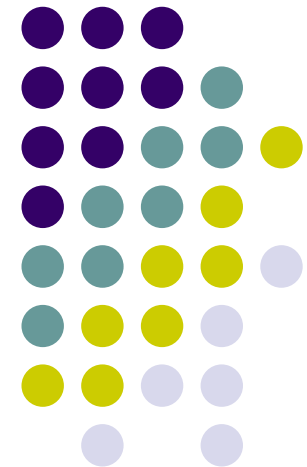


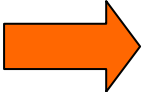
# Systemsoftware

4

Nebenläufigkeit I Forts.  
Wechselseitiger Ausschluss und  
Synchronisierung



# Nebenläufigkeit I

1. Verfahren der Nebenläufigkeit
2. Wechselseitiger Ausschluss: Software-Ansätze
3. Wechselseitiger Ausschluss: Hardware-Unterstützung
-  4. Semaphore
5. Monitore
6. Nachrichtenaustausch
7. Leser/Schreiber-Problem

# Semaphore

- Von Dijkstra 1965 entwickelt
- Kooperation von 2 oder mehr Prozesse über Signale
- Dadurch kann:
  - Ein Prozess angehalten werden (Zustand blockiert)
  - Ein Prozess in den Zustand „bereit“ versetzt werden
- Für die Signalisierung werden spezielle Variablen eingesetzt: Semaphore  $s$
- Prozesse führen die Operationen  $P(s)$  und  $V(s)$  aus.
  - $P(s)$ : Prozess möchte kritische Ressource nutzen
  - $V(s)$ : Prozess gibt kritische Ressource frei
- $P(s)$  und  $V(s)$  sind atomar

# Eigenschaften von Semaphore

- Ganzzahlige (nicht negative) Werte
- Initialisierung auf nicht negativen Wert
- Operation P(s)
  - Prüfen ob Ressource zur Verfügung steht und dekrementieren des Semaphors
    - Wenn Semaphor $>0$ : verringert den Wert um 1
    - Wenn Semaphor $=0$ : Prozess der P(s) ausführt wird blockiert
- Operation V(s)
  - Verlassen des kritischen Bereichs und inkrementieren des Semaphors
    - Semaphor wird um 1 erhöht
    - Wenn Semaphor $=0$ : Blockierter Prozess wird geweckt

# Eine mögliche Definition von P(s) und V(s)

```
/* Warteschlange */
Queue queue;

/* Initialisierung */
Init(s, v)
Semaphor s;
int v;
{
    s = v;
}
/* v gibt die Anzahl der zur
Verfügung stehenden
kritischen Ressourcen an:
Bsp: Puffer mit v
Speicherplätzen */
```

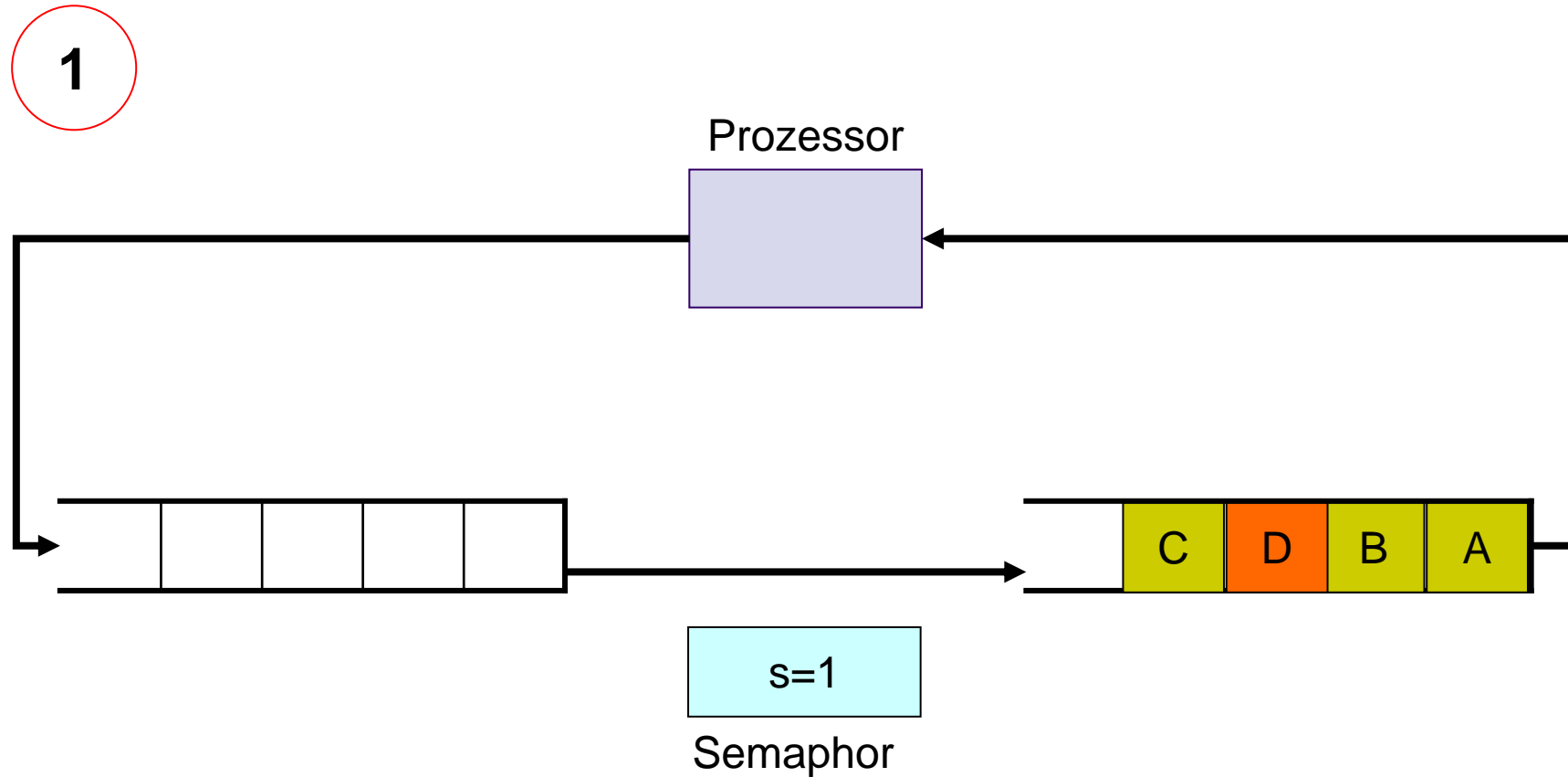
```
/* Implementation P(s) */
P(s)
Semaphor s;
{
    if (s==0) haltean(queue);
    s = s-1;
}

/* Implementation V(s) */
V(s)
Semaphor s;
{
    s = s+1;
    weckeauf(queue);
}
```

# Definitionen

- Binäres Semaphor
  - Das Semaphor kann nur die Werte 0 und 1 annehmen
- Starkes Semaphor
  - Reihenfolge des Herausholen eines Prozesses aus Warteschlange wird vom Semaphor geregelt
  - FIFO-Prinzip
- Schwaches Semaphor
  - Keine Festlegung der Reihenfolge wie Prozesse aus der Warteschlange geholt werden

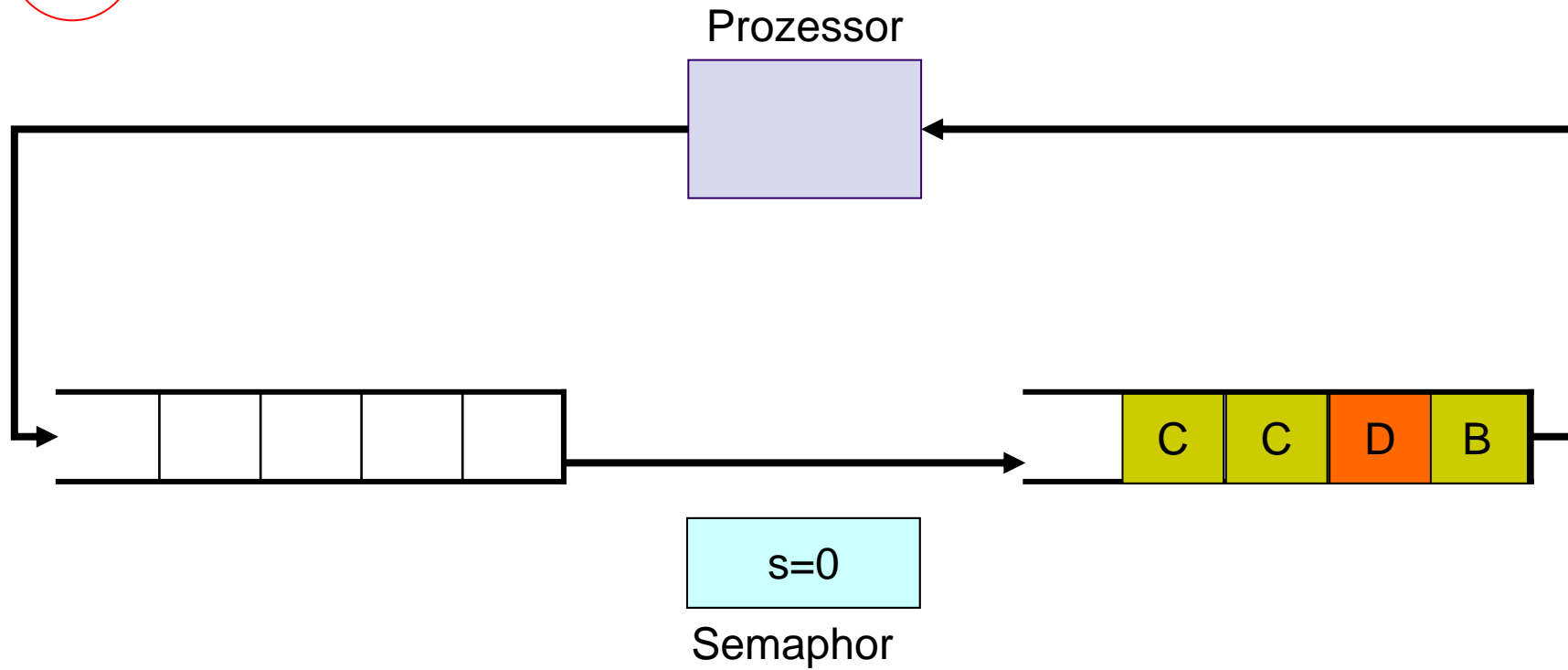
# Beispiel eines Semaphormechanismus



Prozesse A, B, C sind jeweils auf ein vorliegendes Ergebnis von D angewiesen

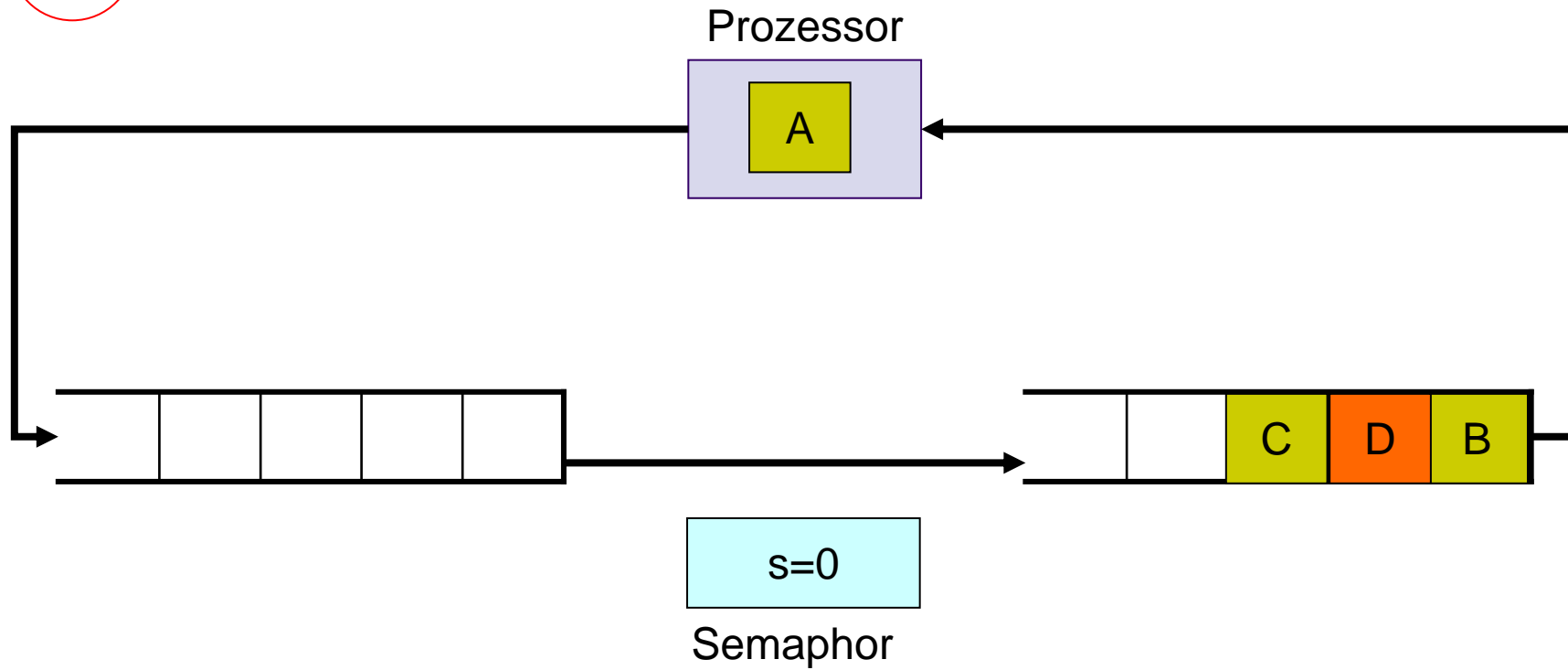
# Beispiel eines Semaphormechanismus

2



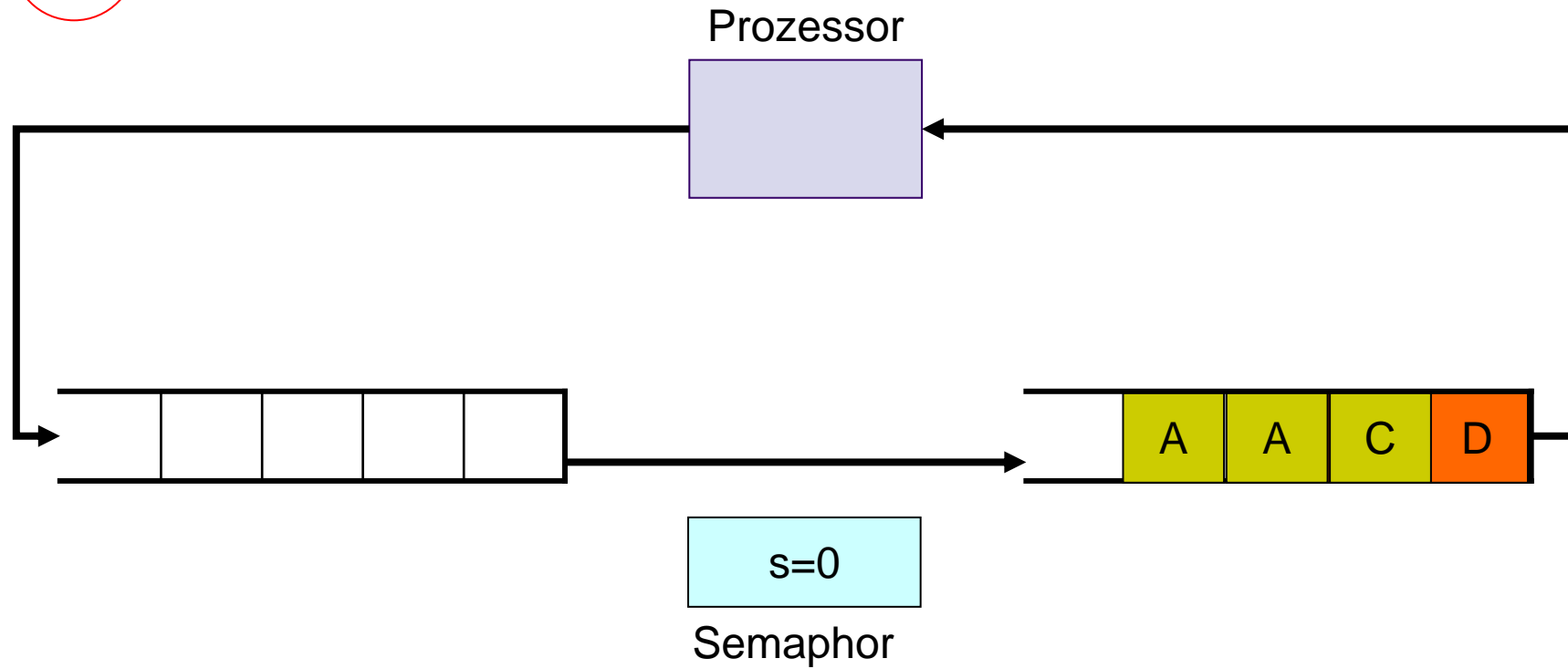
# Beispiel eines Semaphormechanismus

3



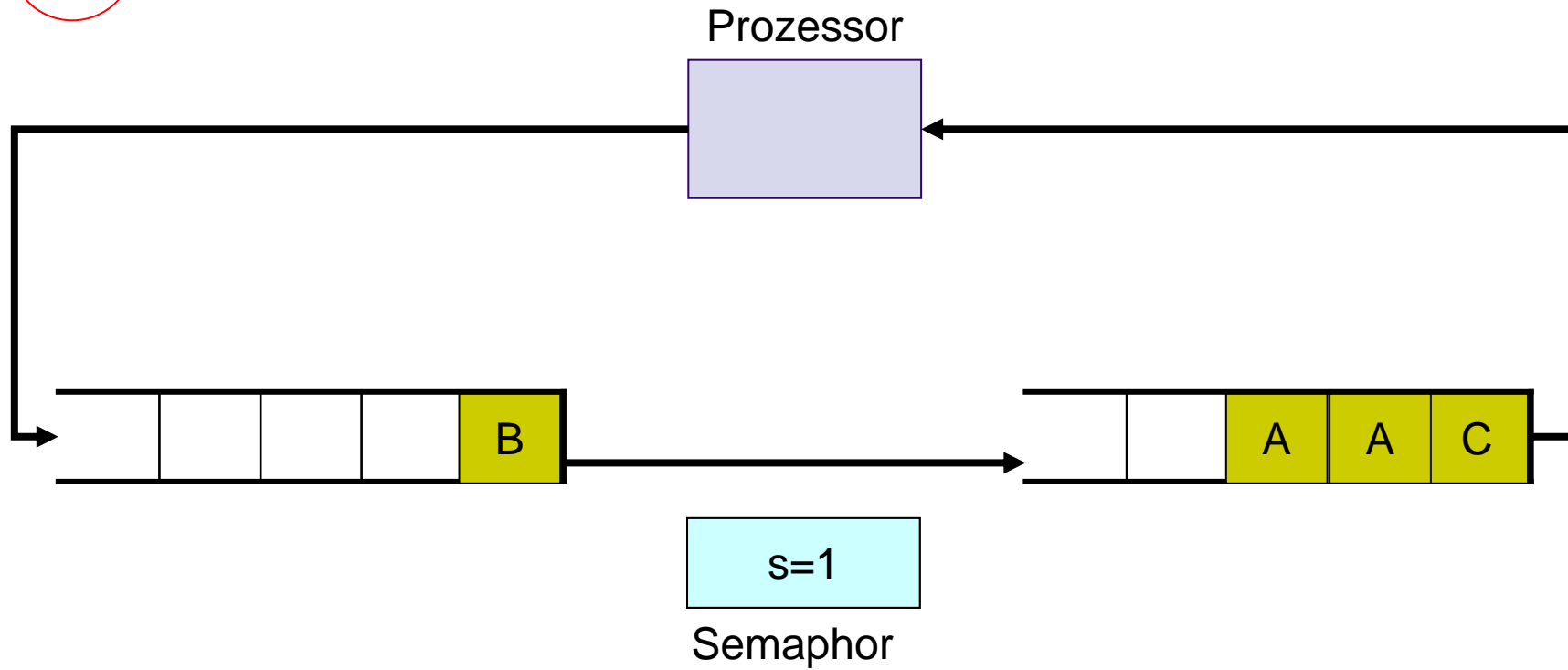
# Beispiel eines Semaphormechanismus

4



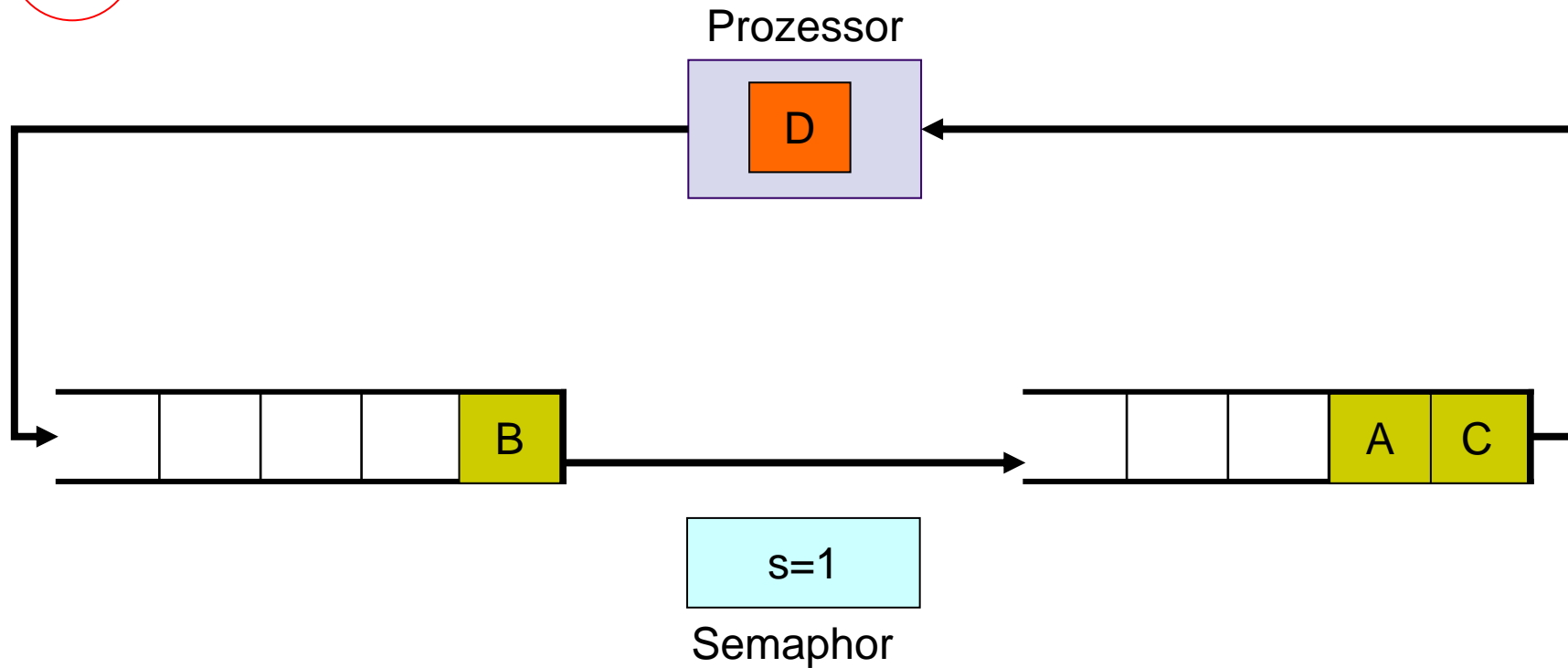
# Beispiel eines Semaphormechanismus

5



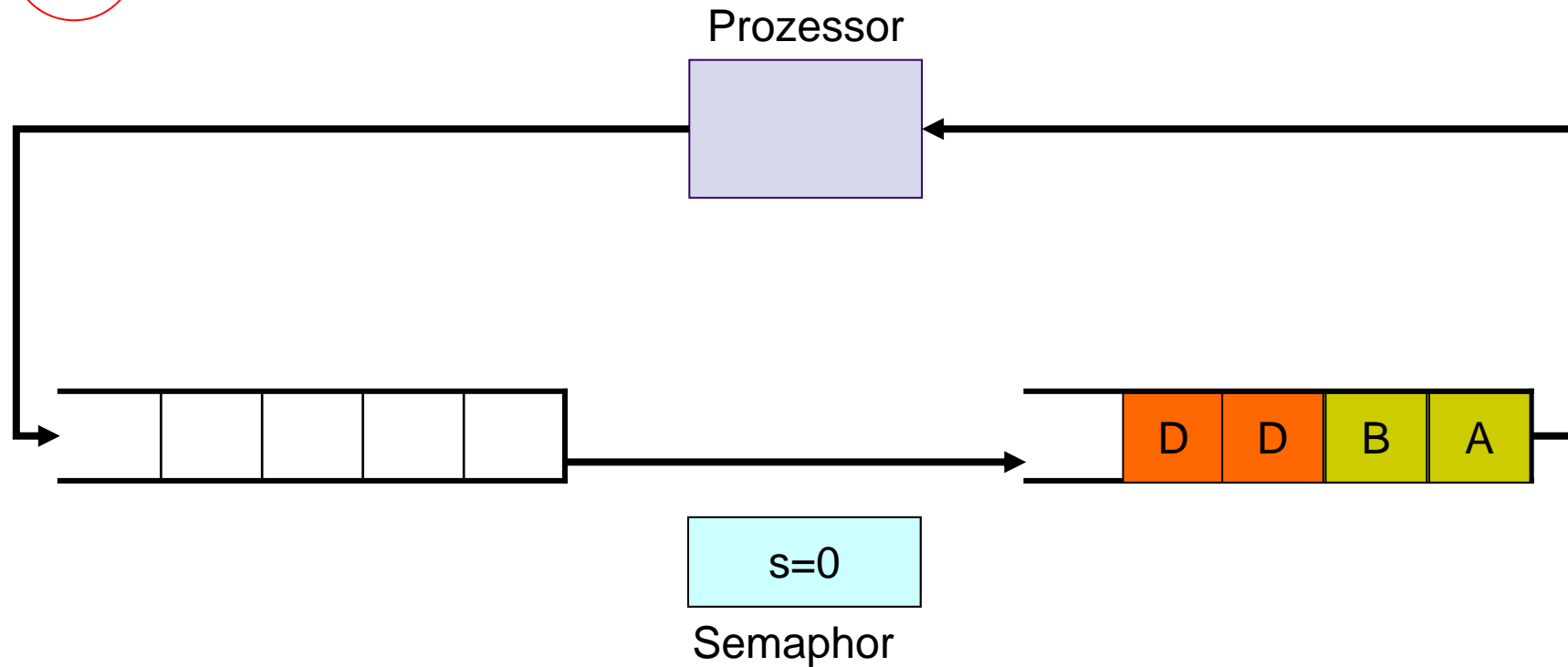
# Beispiel eines Semaphormechanismus

6



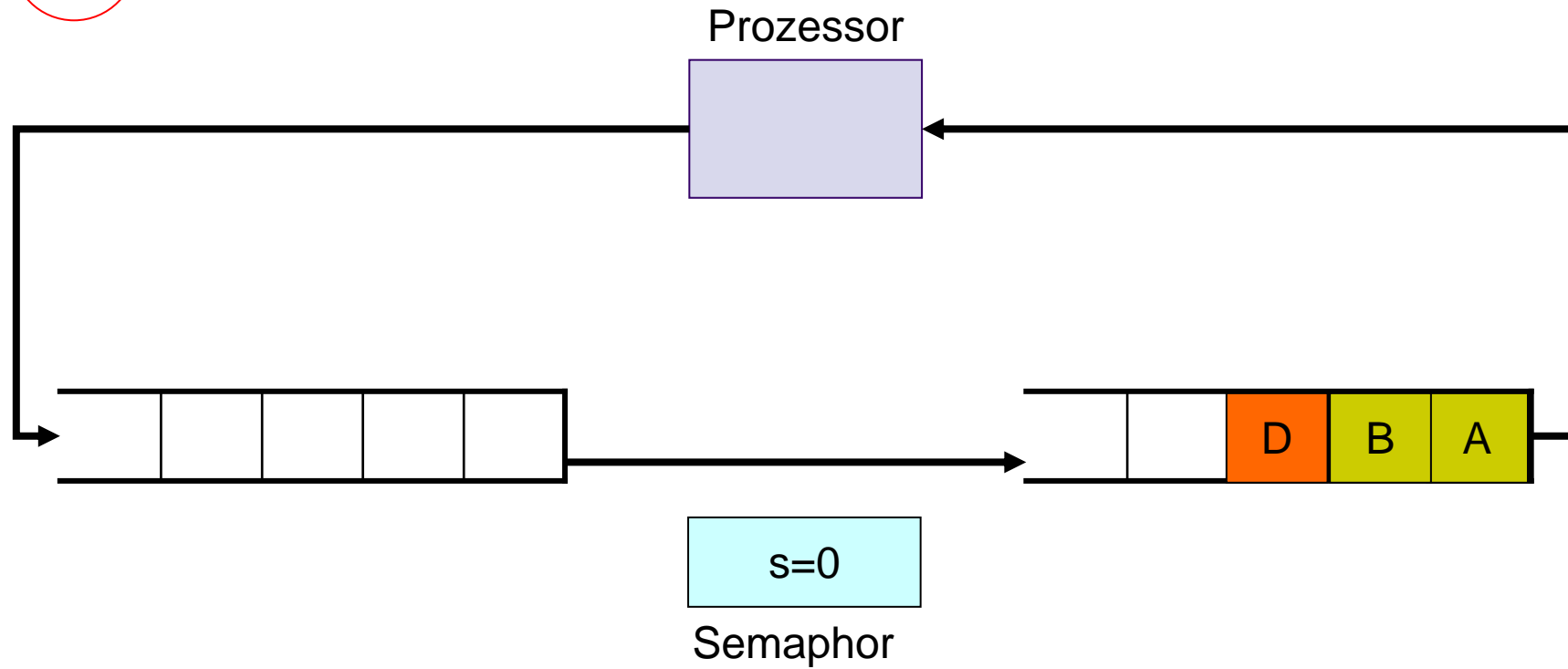
# Beispiel eines Semaphormechanismus

7



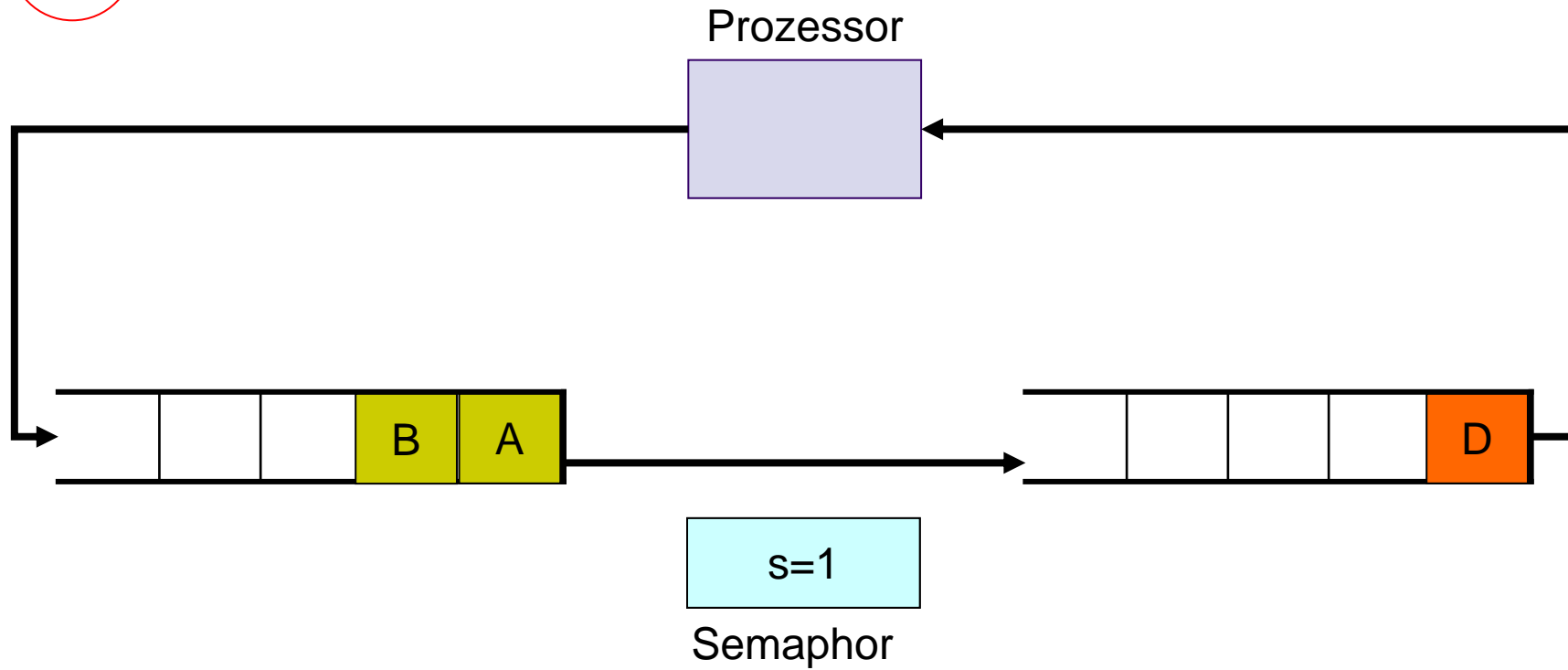
# Beispiel eines Semaphormechanismus

8



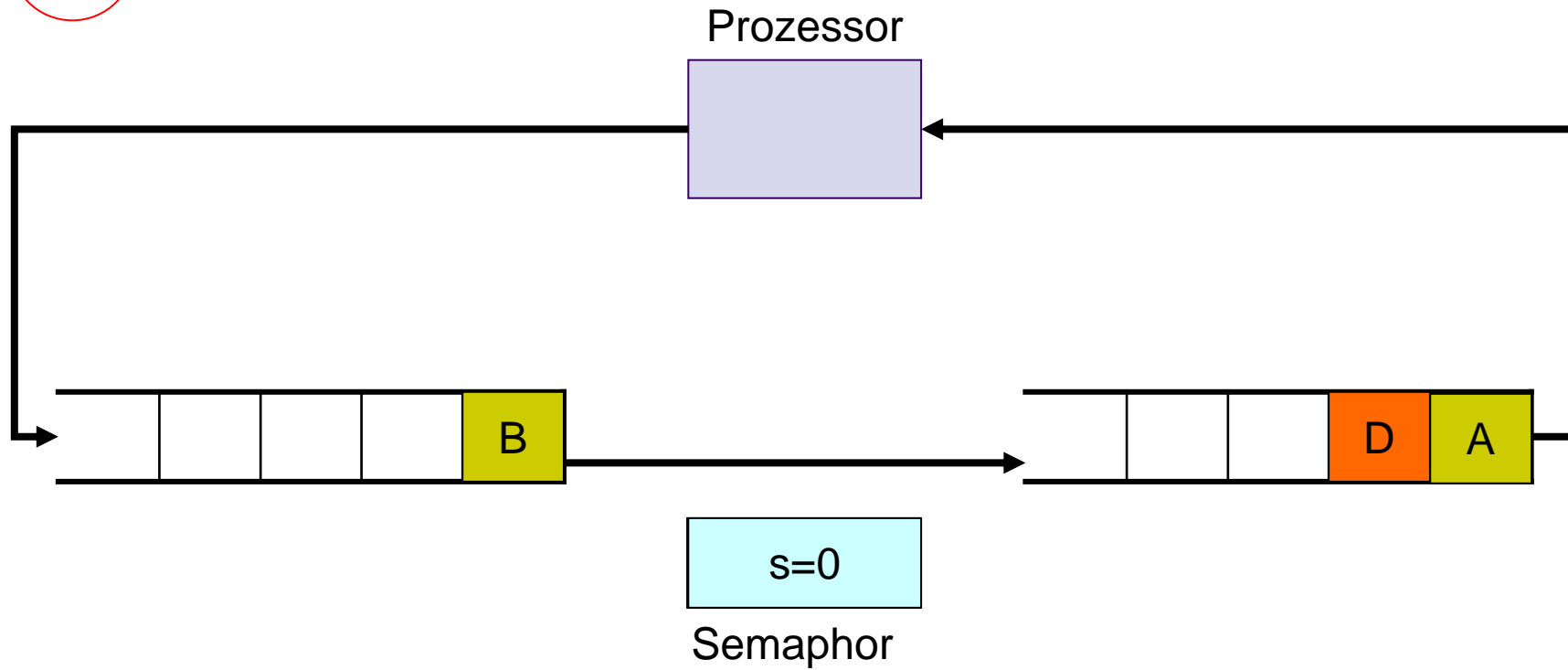
# Beispiel eines Semaphormechanismus

9



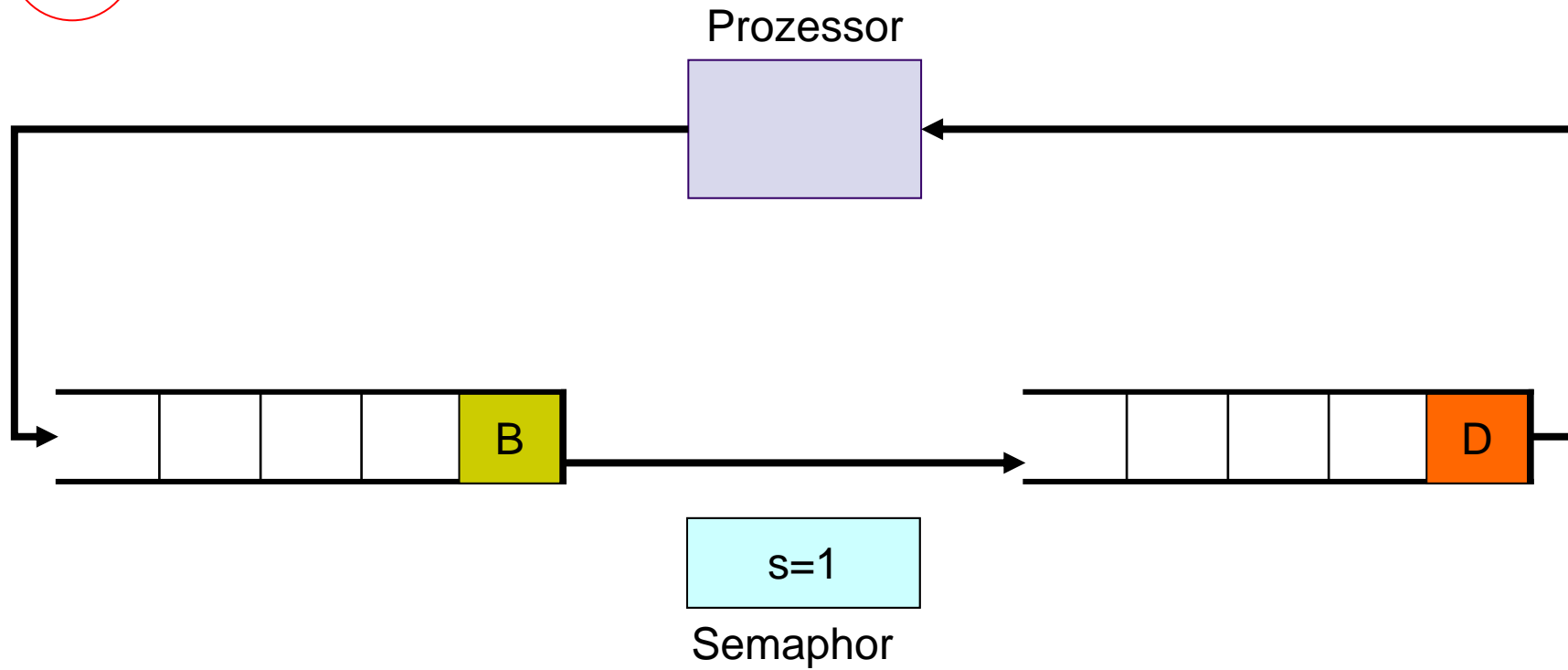
# Beispiel eines Semaphormechanismus

10



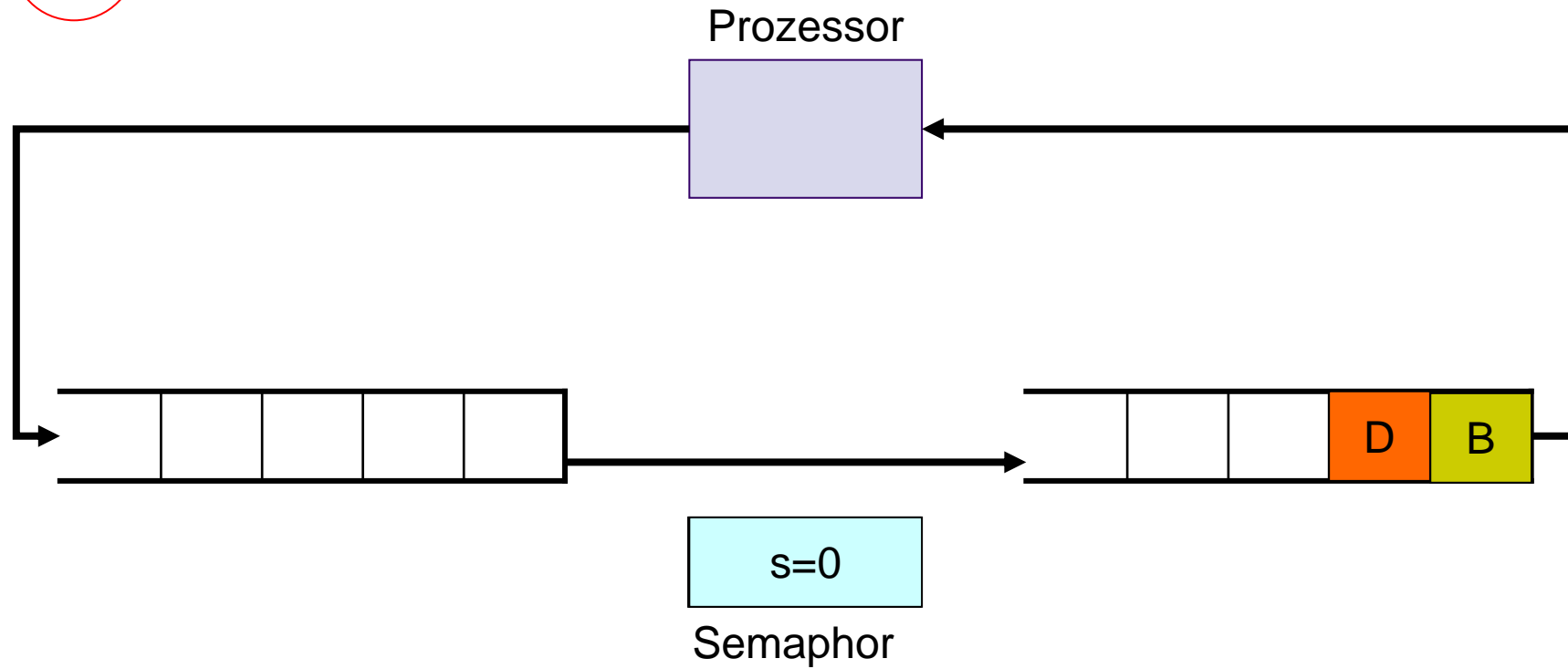
# Beispiel eines Semaphormechanismus

11



# Beispiel eines Semaphormechanismus

12

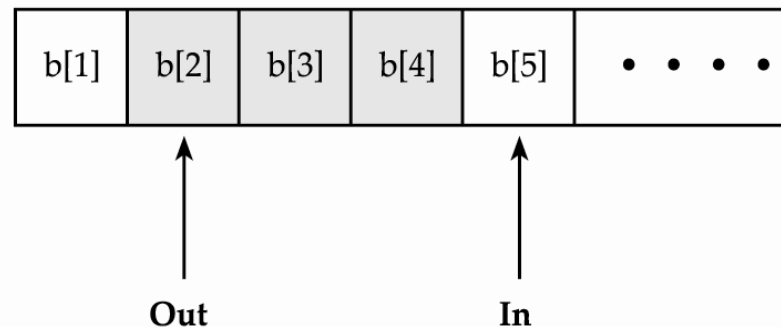


# Wechselseitiger Ausschluss mit Semaphoren

```
const int n= /*Anzahl der Prozesse */;
semaphore s=1;
void P(int i)
{
    while(true)
    {
        P(s);
        /* kritischer Abschnitt */
        V(s);
        /* restliche Programmzeilen */
    }
}
void main()
{
    parbegin (P(1), P(2), ..., P(n));
}
```

# Das Erzeuger/Verbraucher-Problem (unendlicher Puffer)

- Szenario:
  - Ein- oder mehrere Erzeuger, die Daten erzeugen und in Puffer ablegen
  - Einen einzigen Verbraucher, der Daten aus Puffer herausholt
  - Überschneidung von Pufferoperationen muss vermieden werden.
  - Reihenfolge Ablegen->Herausholen muss gewährleistet sein



Hinweis: Belegte Pufferbereiche sind schattiert dargestellt.

# Erzeuger/Verbraucher Implementierung

## Erzeuger

```
while(true)
{
    /* Erzeuge Element v */
    b[in]=v;
    in++;
}
```

## Verbraucher

```
while(true)
{
    while (in<=out)
        /* tue nichts */;
    w=b[out];
    out++;
    /* verbrauche Element w */
}
```

# Erzeuger/Verbraucher Implementierung 2

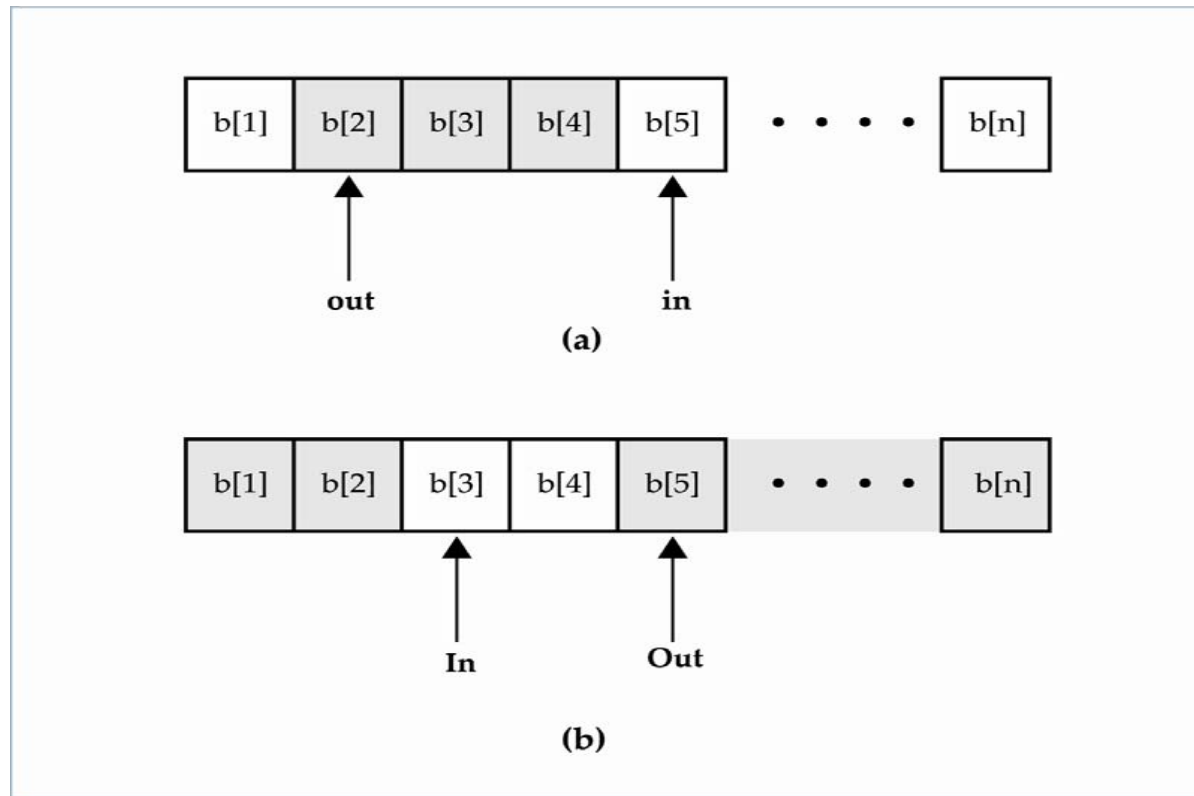
```
/*Programm
Erzeuger-Verbraucher */;
semaphore n=0;
semaphore s=1;
void producer()
{
    while(true)
    {
        produce();
        P(s);
        append();
        V(s);
        V(n);
    }
}
```

```
void consumer()
{
    while(true)
    {
        P(n);
        P(s);
        take();
        V(s);
        consume();
    }
}

void main()
{
    parbegin(producer, consumer);
}
```

# Das Erzeuger/Verbraucher-Problem (endlicher Puffer)

- Szenario:
  - wie vorher
  - Puffer besitzt eine endliche Größe



# Erzeuger/Verbraucher Implementierung

## Erzeuger

```
while(true)
{
    /* Erzeuge Element v */
    while((in+1)%n == out)
        /* nichts tun */;
    b[in]=v;
    in=(in+1)%n;
}
```

## Verbraucher

```
while(true)
{
    while (in==out)
        /* tue nichts */;
    w=b[out];
    out=(out+1)%n;
    /* verbrauche Element w */
}
```

# Erzeuger/Verbraucher Implementierung 2

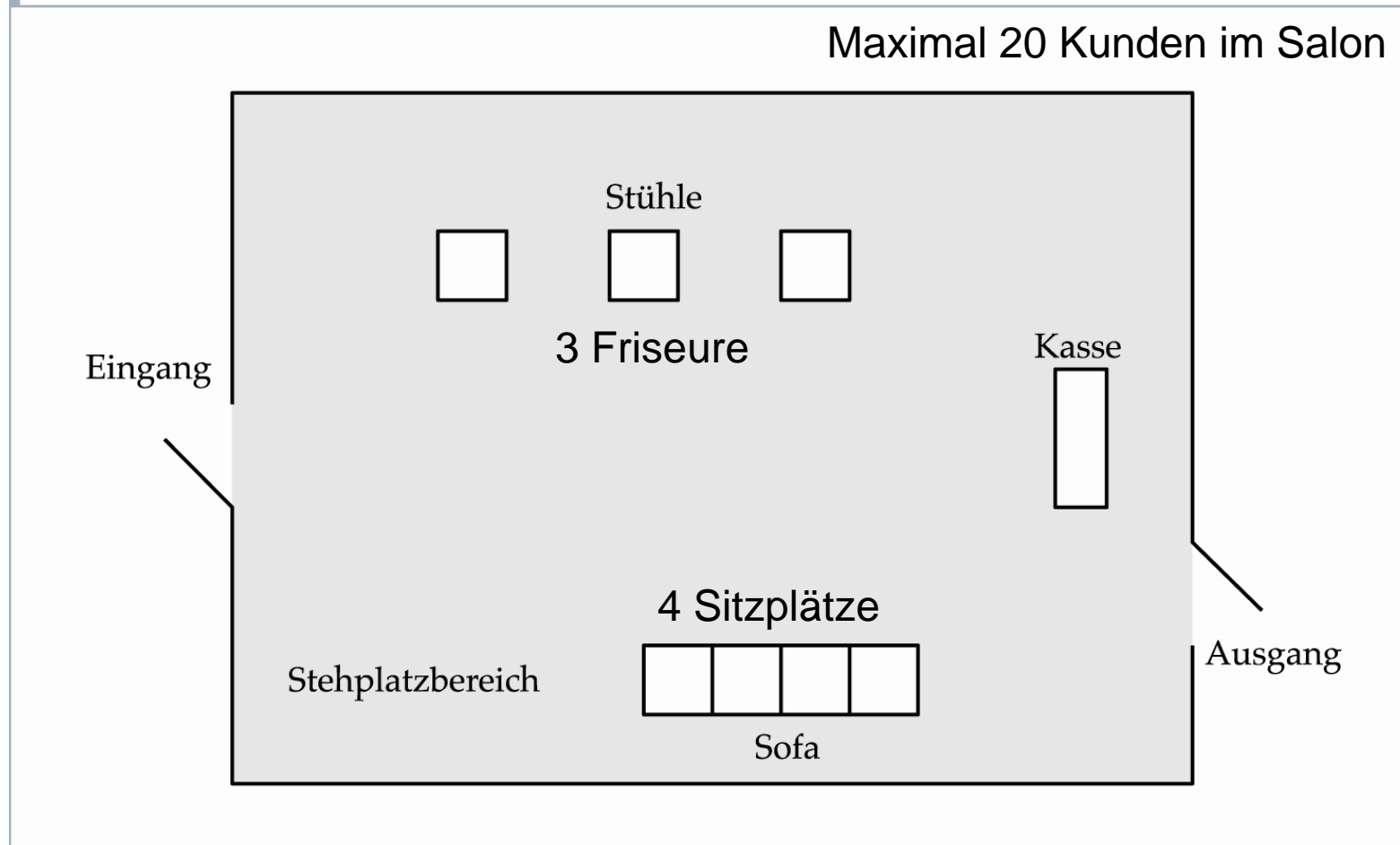
```
/*Programm
Erzeuger-Verbraucher */;
semaphore n=0;
semaphore s=1;
semaphore e=sizeofbuffer;
void producer()
{
    while(true)
    {
        produce();
        P(e);
        P(s);
        append();
        V(s);
        V(n);
    }
}
```

```
void consumer()
{
    while(true)
    {
        P(n);
        P(s);
        take();
        V(s);
        V(e);
        consume();
    }
}

void main()
{
    parbegin(producer, consumer);
}
```

# Ein Friseursalon-Problem

Abbildung 5.18 Der Friseursalon



# Friseursalon-Problem

- Kein Kunde betritt den vollen Salon
- Ein Kunde der den Salon betritt
  - nimmt auf Sofa platz wenn er warten muss und Sofa ist frei
  - bleibt stehen wenn er warten muss und Sofa ist belegt
  - wird bedient, sobald ein Friseur frei ist
- Ein Friseur nimmt den am längsten auf dem Sofa sitzenden Kunden
- Der am längsten stehende Kunde nimmt auf einem freien Sofaplatz platz
- Ist ein Kunde frisiert kann Friseur Bezahlung entgegennehmen
- Es gibt nur 1 Kasse, so dass immer nur 1 Kunde bezahlen kann
- Friseure verbringen ihre Zeit mit:
  - Frisieren
  - Kassieren
  - Schlafen (wenn kein Kunde da ist)

# Unfairer Friseursalon 1

```
/* Programm Friseursalon */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0;
semaphore payment = 0, receipt=0;

void main()
{
    parbegin (customer, /*50x*/..., customer, barber,
             barber, barber, cashier);
}
```

# Unfairer Friseursalon 2

```
void customer()
{
    P(max_capacity);
    enter_shop();
    P(sofa);
    sit_on_sofa();
    P(barber_chair);
    get_up_from_sofa();
    V(sofa);
    sit_in_barber_chair();
    V(cust_ready);
    P(finished);
    leave_barber_chair();
    V(leave_b_chair);
    pay();
    V(payment);
    P(receipt);
    exit_shop();
    V(max_capacity);
}
```

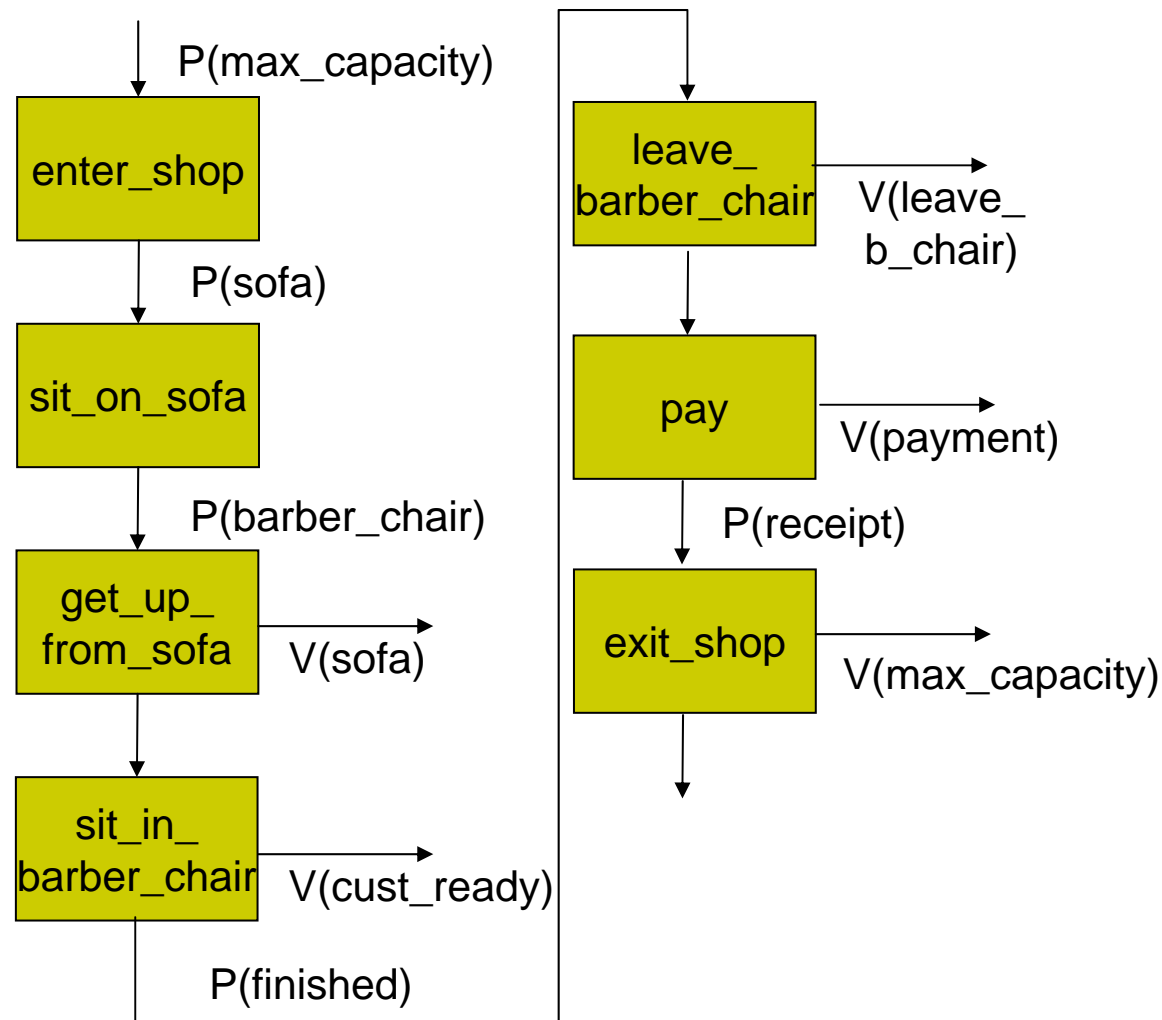
```
void barber()
{
    while(true)
    {
        P(cust_ready);
        P(coord);
        cut_hair();
        V(coord);
        V(finished);
        P(leave_b_chair);
        V(barber_chair);
    }
}
```

```
void cashier()
{
    while(true)
    {
        P(payment);
        P(coord);
        accept_pay();
        V(coord);
        V(receipt);
    }
}
```

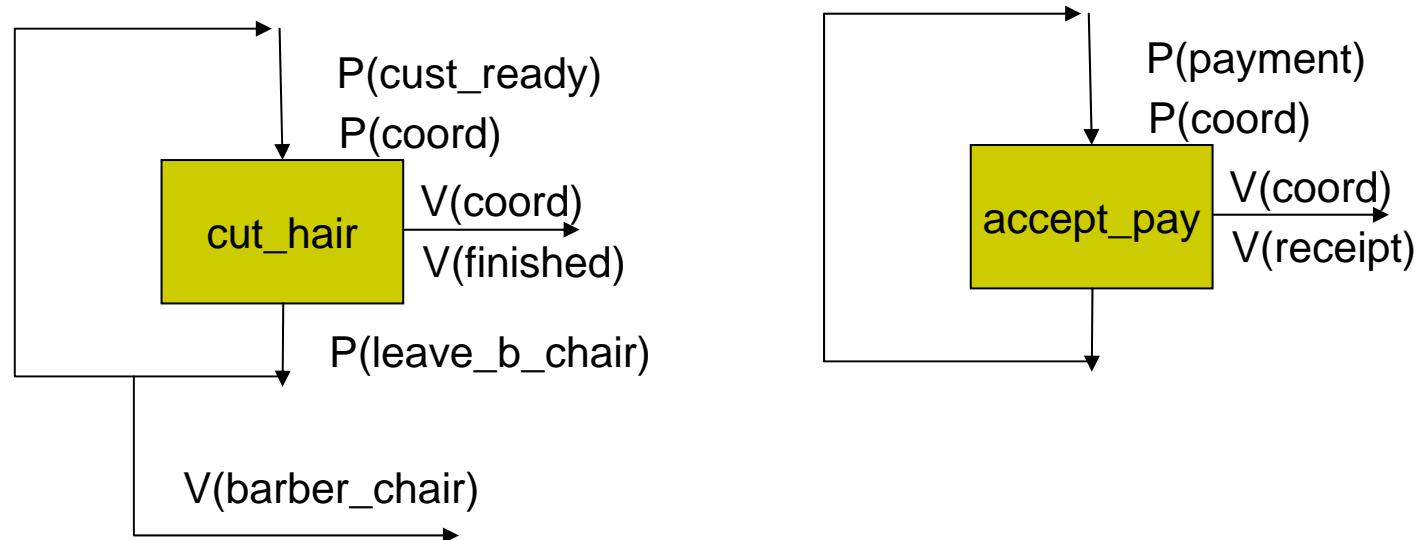
# Unfairer Friseursalon 3

Semaphor	P-Operation	V-Operation
<b>max_capacity</b>	Kunde wartet auf Platz um Salon zu betreten	Kunde signalisiert freien Platz im Salon
<b>sofa</b>	Kunde wartet auf Platz auf Sofa	Kunde signalisiert einen freien Platz auf dem Sofa
<b>barber_chair</b>	Kunde wartet auf Platz im Friseurstuhl	Friseur signalisiert freien Platz
<b>cust_ready</b>	Friseur wartet, dass Kunde im Friseurstuhl platz nimmt	Kunde signalisiert Friseur dass er platz genommen hat
<b>finished</b>	Kunde wartet auf fertigen Haarschnitt	Friseur signalisiert Kunden fertigen Haarschnitt
<b>leave_b_chair</b>	Friseur wartet bis Kunde vom Friseurstuhl aufsteht	Kunde signalisiert Friseur dass er aufgestanden ist
<b>payment</b>	Kassierer wartet, dass Kunde bezahlt	Kkunde singalisiert dem Kassierer, dass er bez. hat.
<b>receipt</b>	Kunde wartet auf Quittung	Kassierer signalisiert abgeschlossenen Bezahlvorg.
<b>coord</b>	Warten auf Friseurressource (frisieren oder zahlen)	Signalisierung dass Friseurressource frei ist

# Unfairer Friseursalon - Customer



# Unfairer Friseursalon – barber/cashier



- Warum ist der Friseursalon „unfair“?
- Kunden müssen in der Reihenfolge, in der sie auf die Friseurstühle saßen auch bezahlen
  - => Ein Kunde eines „schnellen“ Friseurs muss unnötig lange warten
  - => Ein Kunde eines „langsamen“ Friseurs muss zahlen, obwohl der Haarschnitt noch nicht fertig ist???

# Ein fairer Friseursalon 1

```
/* Programm Friseursalon */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1=1, mutex2=1;
semaphore cust_ready = 0, leave_b_chair = 0;
semaphore payment = 0, receipt=0;
semaphore finished[50]={0};
int count;

void main()
{
    count:=0;
    parbegin (customer, /*50x*/..., customer, barber,
             barber, barber, cashier);
}
```

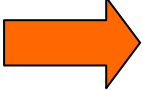
# Ein fairer Friseursalon 2

```
void customer()
{
    int custnr;
    P(max_capacity);
    enter_shop();
    P(mutex1);
    custnr=count;
    count++;
    V(mutex1);
    P(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    V(sofa);
    sit_in_barber_chair();
    P(mutex2);
    enqueue1(custnr);
    V(cust_ready);
    V(mutex2);
    P(finished[custnr]);
    leave_barber_chair();
    V(leave_b_chair);
    pay();
    V(payment);
    P(receipt);
    exit_shop();
    V(max_capacity);
}
```

```
void barber()
{
    int b_cust;
    while(true)
    {
        P(cust_ready);
        P(mutex2);
        dequeue1(b_cust);
        V(mutex2);
        P(coord);
        cut_hair();
        V(coord);
        V(finished[b_cust]);
        P(leave_b_chair);
        V(barber_chair);
    }
}
```

```
void cashier()
{
    while(true)
    {
        P(payment);
        P(coord);
        accept_pay();
        V(coord);
        V(receipt);
    }
}
```

# Nebenläufigkeit I

1. Verfahren der Nebenläufigkeit
2. Wechselseitiger Ausschluss: Software-Ansätze
3. Wechselseitiger Ausschluss: Hardware-Unterstützung
4. Semaphore
-  5. Monitore
6. Nachrichtenaustausch
7. Leser/Schreiber-Problem

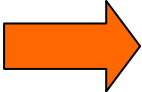
# Monitore

- Hoare (1974) und Brinch Hansen (1975)
- Ähnliche Funktionalität wie Semaphoren
- Leichter zu kontrollieren
- Implementiert z. B. in Java
  - `synchronized(Datenstruktur) {Anweisungen}`
- Programmbibliothek
- Monitor ist ein Softwaremodul bestehend aus:
  - einer oder mehrerer Prozeduren
  - Initialisierungssequenz
  - lokale Daten

# Merkmale eines Monitors

- Nur die Prozedur des Monitors kann auf die lokalen Variablen zugreifen
- Ein Prozess betritt den Monitor, indem er eine seiner Prozeduren aufruft
- Es kann zu jedem Zeitpunkt immer nur ein einziger Prozess im Monitor ausgeführt werden
- Alle anderen Prozesse, die den Monitor aufrufen werden suspendiert und müssen warten
- => Wechselseitiger Ausschluss gewährleistet
- Um für nebenläufige Prozesse einsetzbar zu sein:
- => Monitor muss Synchronisierungsmechanismen beinhalten

# Nebenläufigkeit I

1. Verfahren der Nebenläufigkeit
2. Wechselseitiger Ausschluss: Software-Ansätze
3. Wechselseitiger Ausschluss: Hardware-Unterstützung
4. Semaphore
5. Monitore
-  6. Nachrichtenaustausch
7. Leser/Schreiber-Problem

# Nachrichtenaustausch

- Prozesse die miteinander kooperieren
  - müssen sich synchronisieren
  - müssen Informationen austauschen
- Ansatz: Nachrichtenaustausch
  - eignet sich auch für verteilte Systeme und Mehrprozessorsysteme
- Nachrichtenaustausch wird realisiert durch die Operationen
  - send (Ziel, Nachricht)
  - receive(Quelle, Nachricht)

# Merkmale von Nachrichtensystemen

- ➔ ● Synchronisierung
  - Senden
    - blockierend <-> nicht blockierend
  - Empfangen
    - blockierend <-> nicht blockierend
    - Test auf Ankunft
- Adressierung
  - Direkt
    - Senden
    - Empfangen
      - explizit <-> implizit
  - Indirekt
    - statisch <-> dynamisch
    - Besitz einer Mailbox
- Format
  - Inhalt
  - Länge
    - fest <-> variabel
- Warteschlangenverfahren
  - FIFO <-> Priorität

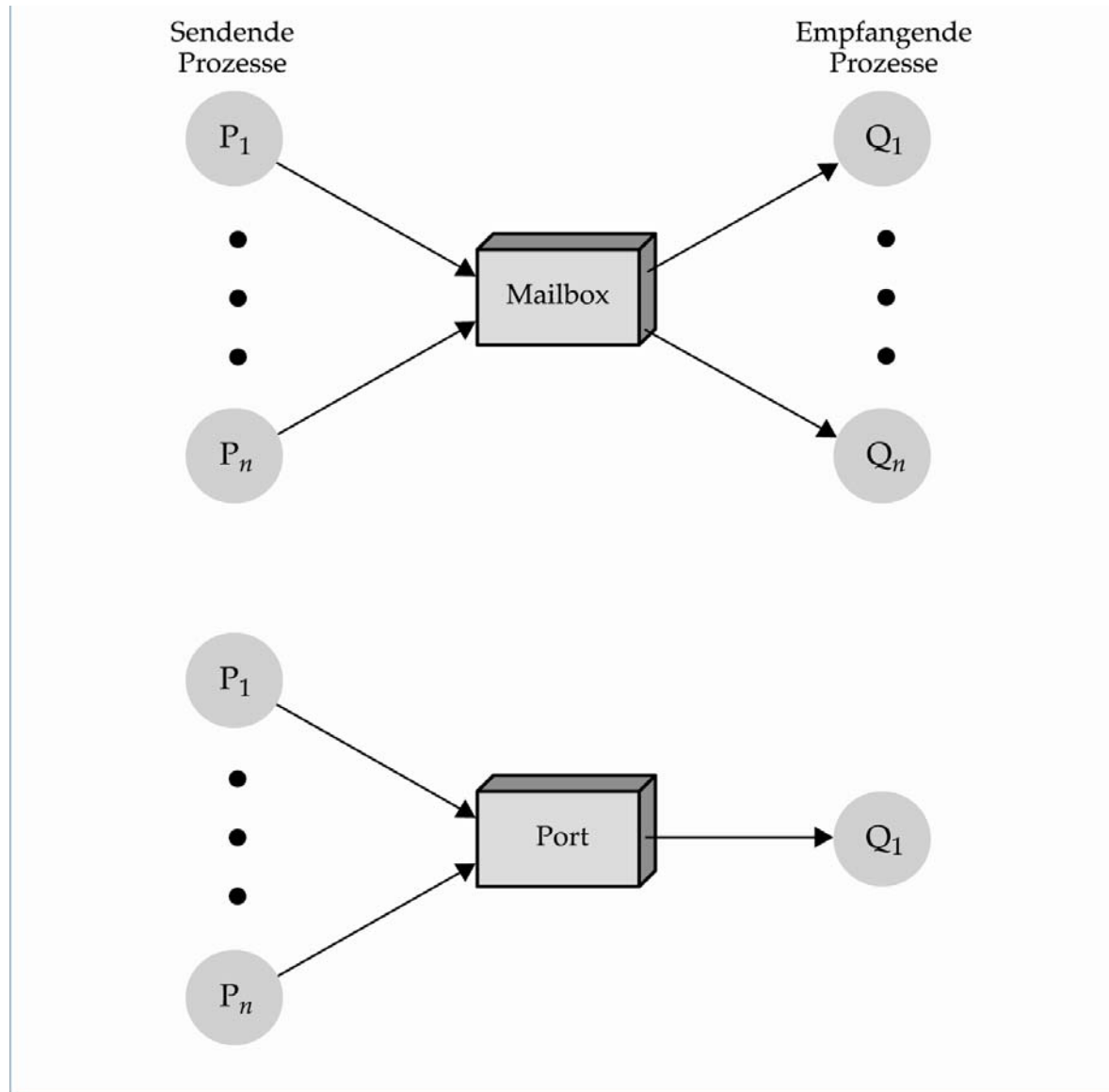
# Synchronisierung

- Blockierendes Senden, blockierendes Empfangen
  - Sender und Empfänger sind so lange blockiert, bis Nachricht zugestellt ist
  - *Rendezvous*
  - Enge Synchronisation
- Nicht blockierendes Senden, blockierendes Empfangen
  - Nur Empfänger bleibt blockiert, bis Nachricht zugestellt
  - Sinnvollste Kombination
  - Sender kann schnell mehrere Nachrichten versenden
- Nicht blockierendes Senden, nicht blockierendes Empfangen


# Adressierung

- Direkte Adressierung
  - Send:
    - enthält spezifische Kennung des Zielprozesses
  - Receive:
    - weiß im Voraus, von welchem Prozess die Nachricht kommt
- Indirekte Adressierung
  - Nicht direkt von Sender an Empfänger
  - Verschicken an gemeinsame Datenstruktur (Warteschlange, *Mailbox*, *Port*)
  - Entkopplung von Sender und Empfänger

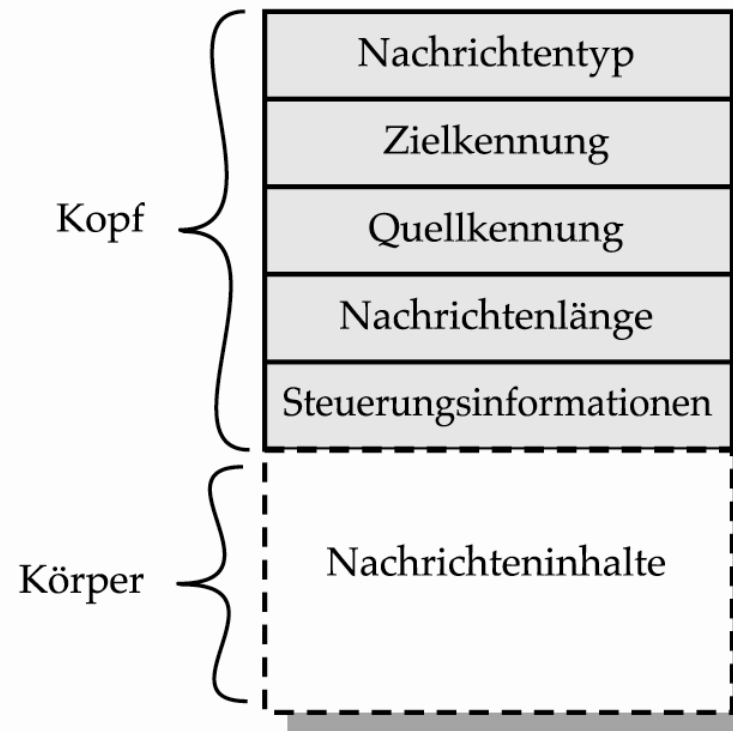
# Indirekte Prozesskommunikation



# Merkmale von Nachrichtensystemen

- Synchronisierung
    - Senden
      - blockierend <-> nicht blockierend
    - Empfangen
      - blockierend <-> nicht blockierend
      - Test auf Ankunft
  - Adressierung
    - Direkt
      - Senden
      - Empfangen
        - explizit <-> implizit
    - Indirekt
      - statisch <-> dynamisch
      - Besitz einer Mailbox
- 
- Format
    - Inhalt
    - Länge
      - fest <-> variabel
  - Warteschlangenverfahren
    - FIFO <-> Priorität

# Allgemeines Nachrichtenformat



# Wechselseitiger Ausschluss mit Nachrichten

```
/* Programm Wechselseitiger Ausschluss*/
const int n=/*Anzahl der Prozesse*/;
void P(int i)
{
    message msg;
    while(true)
    {
        receive (mutex, msg);           // Empfangen der Null-Nachricht
        /* kritischer Abschnitt */;
        send (mutex, msg);              // Versenden der Null-Nachricht
        /* Restlicher Programmcode */;
    }
}
void main()
{
    create_mailbox(mutex);             // mutex: Name der Mailbox
    send (mutex, null);                // Initialisierung der Mailbox
                                        // eine Nachricht ohne Inhalt
    parbegin (P(1), P(2), ... , P(n));
}
```

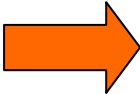
# Erzeuger/Verbraucher mit Nachrichten

```
const int
  capacity= /*Puffergröße*/;
  null= /*Leernachricht*/;
int i;
void producer()
{
  message pmsg;
  while(true)
  {
    receive(mayproduce, pmsg);
    pmsg=produce();
    send(mayconsume, pmsg);
  }
}
```

```
void consumer()
{
  message cmsg;
  while(true)
  {
    receive(mayconsume, cmsg);
    consume(cmsg);
    send(mayproduce, null);
  }
}
```

```
void main()
{
  create_mailbox(mayproduce);
  create_mailbox(mayconsume);
  for (int i=1;i<=capacity; i++) send (mayproduce, null);
  parbegin (producer, consumer);
}
```

# Nebenläufigkeit I

1. Verfahren der Nebenläufigkeit
2. Wechselseitiger Ausschluss: Software-Ansätze
3. Wechselseitiger Ausschluss: Hardware-Unterstützung
4. Semaphore
5. Monitore
6. Nachrichtenaustausch
-  7. Leser/Schreiber-Problem

# Leser-Schreiber-Problem

- Gemeinsam genutzter Datenbereich (Dateien)
- Prozesse die nur lesen (Leser)
- Prozesse die nur schreiben (Schreiber)
- Bedingungen:
  - Beliebig viele Leser können die Datei gleichzeitig lesen
  - Jeweils nur ein Schreiber kann in die Datei schreiben
  - Während ein Schreiber in die Datei schreibt, kann sie nicht von Lesern gelesen werden.
- Lösungsbeispiel:
  - Semaphoren mit Priorität für die Leser

# Leser-Schreiber-Problem mit Semaphoren

```
/* Programm Leser-Schreiber */
int readcount;
semaphore x=1, wsem=1;

void reader()
{
    while (true)
    {
        P(x);
        readcount++;
        if (readcount==1)
            P(wsem);
        V(x);
        READUNIT();
        P(x);
        readcount--;
        if (readcount==0)
            V(wsem);
        V(x);
    }
}
```

```
void writer()
{
    while (true)
    {
        P(wsem);
        WRITEUNIT();
        V(wsem);
    }
}

void main()
{
    readcount=0;
    parbegin (reader, writer);
}
```