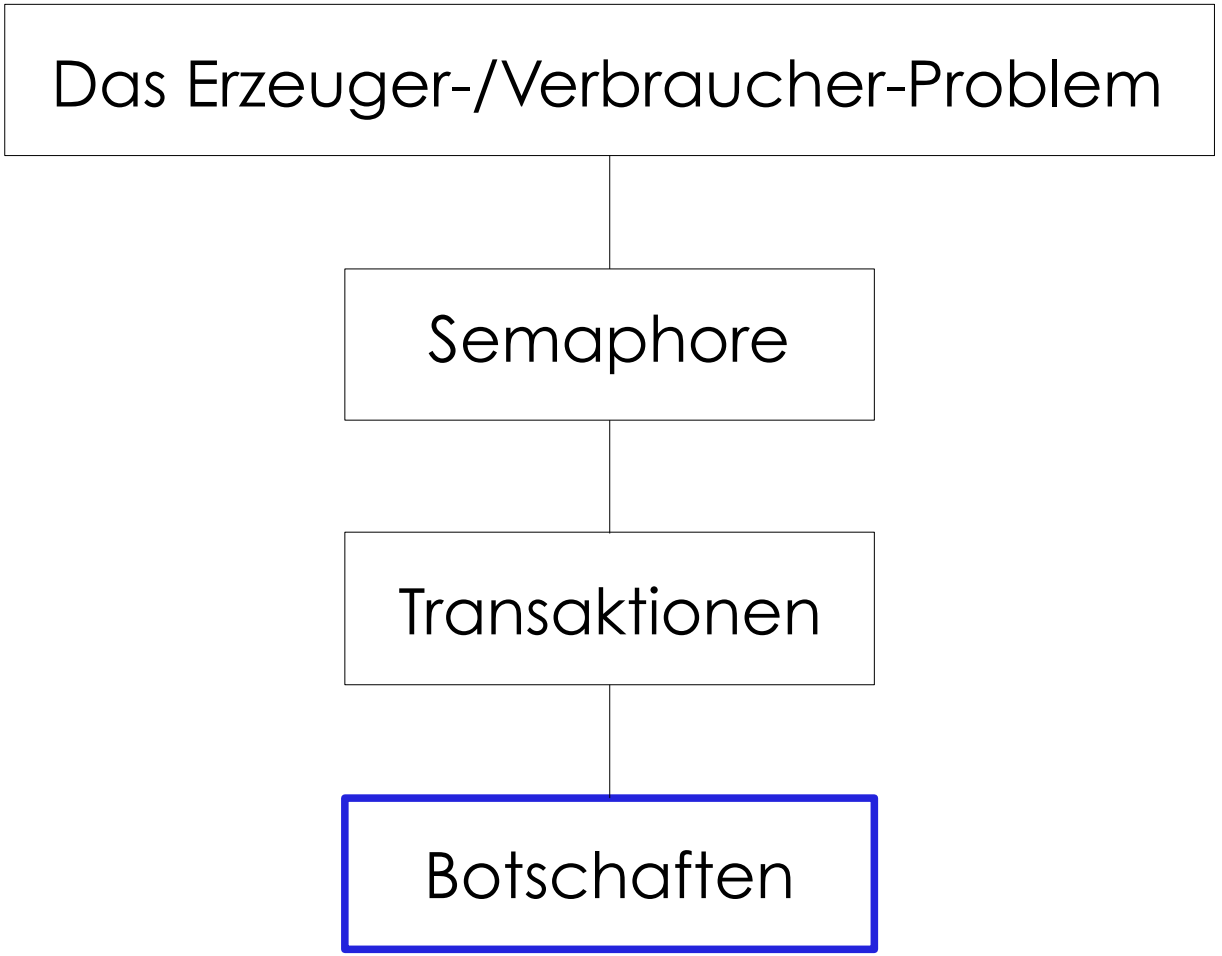
Wegweiser


Das Erzeuger-/Verbraucher-Problem
Semaphore
Transaktionen
Botschaften

Botschaften

Senden

- baue Botschaft auf
  (Daten, Daten, ..., Botschaft)
- sende Botschaft zum Ziel

```
void send(dest, message);

void receive(message);
```

Empfangen

- empfange Botschaft
- extrahiere
  (Daten, Daten, ..., Botschaft)

„marshalling“

EV-Problem mit Botschaften

Erzeuger

```
void producer() {

  while (true) {

     item = produce_item();




     send(item);
  }
}
```

Verbraucher

```
void consumer() {



 while (true) {

    receive(item);




    consume_item(item);

 }
}
```

EV-Problem mit Botschaften

Erzeuger

```
void producer() {

  while (true) {

     item = produce_item();

     receive(empty);



     send(item);
  }
}
```

Verbraucher

```
void consumer() {


send(empty);

 while (true) {

    receive(item);



    send(empty);

    consume_item(item);

 }
}
```

EV-Problem mit Botschaften

Erzeuger

```
void producer() {

  while (true) {

     item = produce_item();

     receive(empty);



     send(item);
  }
}
```

Verbraucher

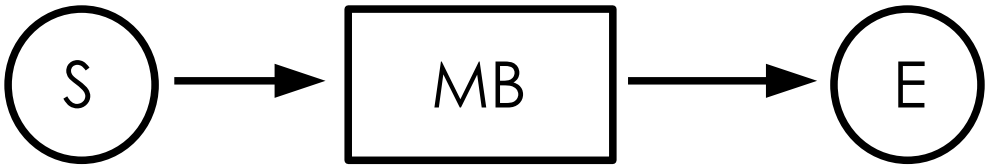
```
void consumer() {

 for(int i = 0; i < N; i++)
   send(empty);

 while (true) {

   receive(item);



   send(empty);

   consume_item(item);

 }
}
```

Botschaften

Varianten des Grundkonzepts

- Adressat:

  Prozess, Thread oder Briefkasten (mail box)



- Pufferung/Synchronität:

  mit/ohne

Synchrone Botschaften

Senden

- **`int send(message, timeout, TID);`**
- synchron, d. h. terminiert, wenn das korrespondierende **`receive/wait`** durchgeführt wurde
- Ergebnis : **`false`**, falls das Timeout greift

Warten

- **`int wait(message, timeout, &TID);`**
- akzeptiert eine Botschaft von irgendeinem Thread
- liefert die ID des Sender-Threads

Empfangen

- **`int receive(message, timeout, TID);`**
- akzeptiert eine Botschaft von einem bestimmten Sender-Thread

Binärer Semaphor mit Hilfe synchroner Botschaften

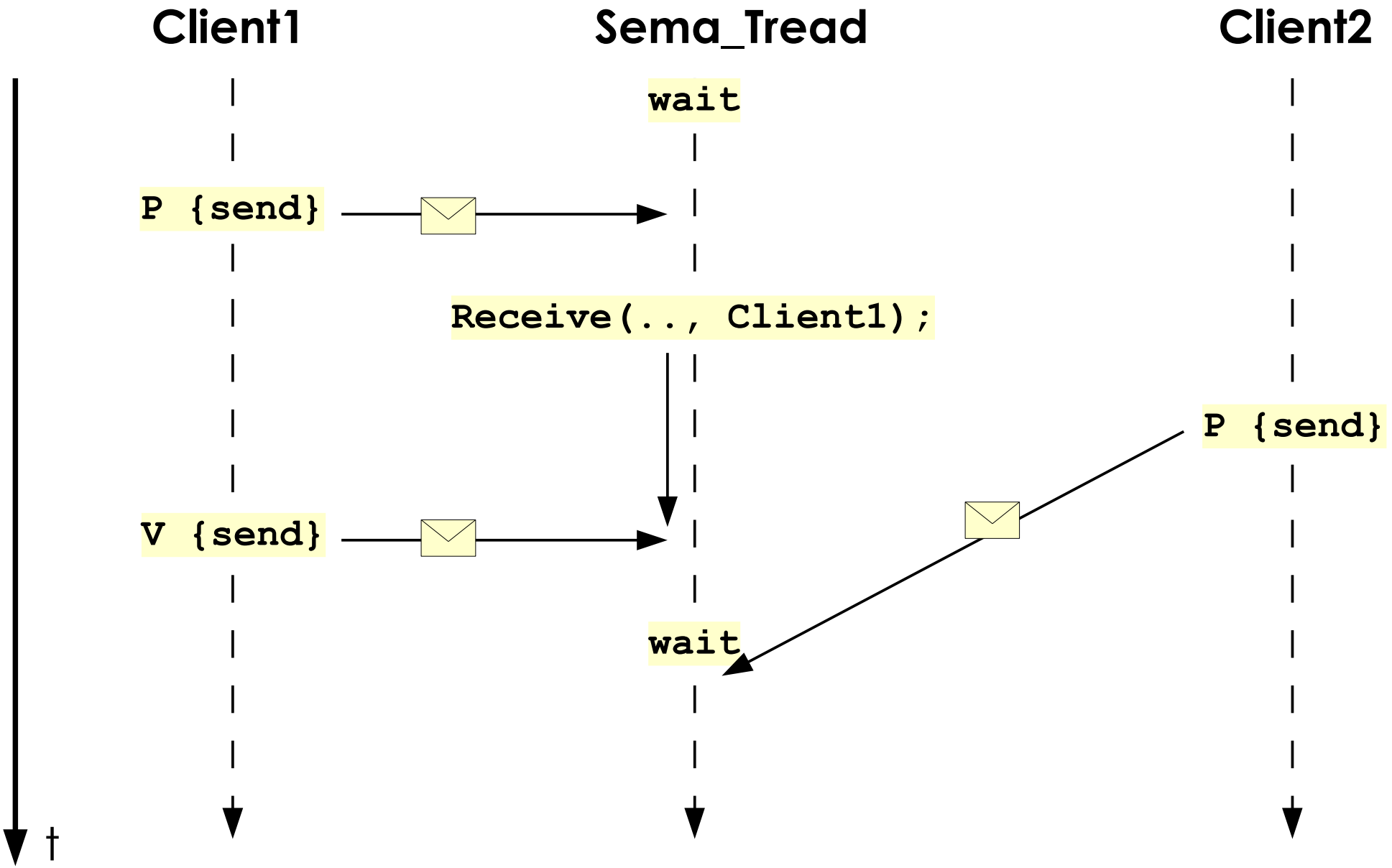
Semaphore-Thread

```
thread_T sema_thread;

while (1) {

 wait(message, threadId);
 receive(message,threadId);

}
```

Clients

```
void P() {

 send(empty_message,
      sema_thread);
}

void V() {

 send(empty_message,
      sema_thread);
}
```

Binärer Semaphor mit Hilfe synchroner Botschaften



Botschaften

Aufbau

- feste Länge
- beliebig, aber am Stück
- zerfleddert

Umgang mit Fehlersituationen (z. B. Netz)

- Quittung für jede Nachricht
- Folgenummern
- gar nichts

Botschaften

Systemarchitektur:

- Betriebssystemkerne (Linux, Mikrokerne, ...)
- Netzwerk-“Stack“
- Bibliotheken (z.B., MPI für Hochleistungsrechnen)

Wegweiser

Zusammenspiel (Kommunikation) mehrerer Threads

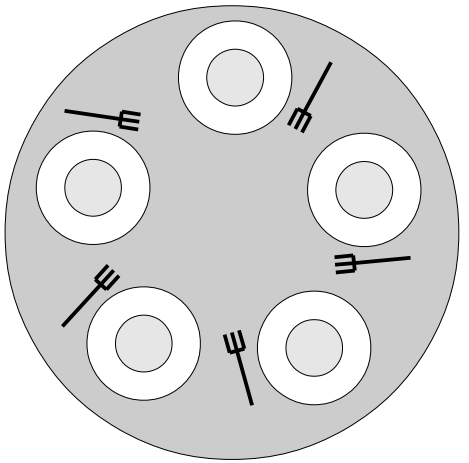
Einige typische Probleme

- Erzeuger / Verbraucher
- 5 Philosophen
- Leser/Schreiber

Das Zuschneiden von Threadsystemen
→ Verschiedene Schedulingverfahren

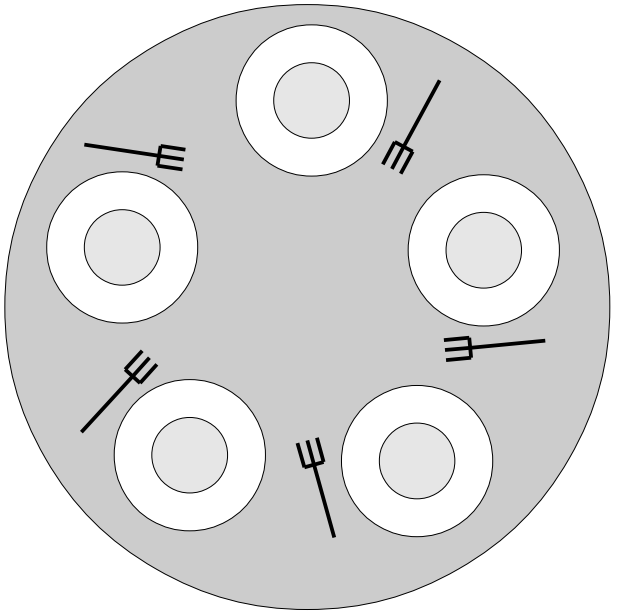
Dining Philosophers

- 5-Philosophen-Problem
- 5 Philosophen sitzen um einen Tisch, denken und essen Spaghetti
- zum Essen braucht jeder zwei Gabeln, es gibt aber insgesamt nur 5
- Problem: kein Philosoph soll verhungern



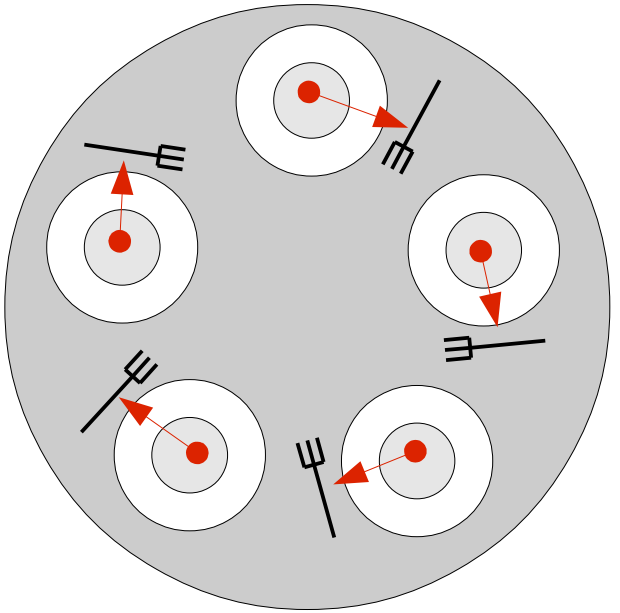
Die offensichtliche ( ) Lösung

```
void Philosoph(int i) {

  Denken();

  NimmGabel(i);

  NimmGabel((i+1)%5);

  Essen();

  LegeZurückGabel(i);
  LegeZurückGabel((i+1)%5);
}
```



Die offensichtliche (aber falsche) Lösung

```
void Philosoph(int i) {

  Denken();

  NimmGabel(i);

  NimmGabel((i+1)%5);

  Essen();

  LegeZurückGabel(i);
  LegeZurückGabel((i+1)%5);
}
```



Falsch

- kann zu Verklemmungen/Deadlocks führen
  (alle Philosophen nehmen gleichzeitig die linken Gabeln)
- zwei verbündete Phil. können einen dritten aushungern

Leser/Schreiber-Problem

Lesen und Schreiben von Daten (z. B. Datenbankeinträgen)

- mehrere Prozesse dürfen gleichzeitig lesen
- nur einer schreiben

Leser/Schreiber-Problem

Lesen und Schreiben von Daten (z. B. Datenbankeinträgen)

- mehrere Prozesse dürfen gleichzeitig lesen
- nur einer schreiben

```
SemaphoreT Mutex(1);
SemaphoreT DB(1);
int        RCount = 0;
```

Leser/Schreiber mit Semaphoren

```
void Reader() {

  //read

}
```

```
void Writer() {

  //write

}
```

Leser/Schreiber mit Semaphoren

```
void Reader() {


  DB.P();

  //read


  DB.V();
}
```

```
void Writer() {


  DB.P();

  //write

  DB.V();


}
```

Leser/Schreiber mit Semaphoren

```
void Reader() {


  Rcount++;
  if (Rcount == 1) DB.P();


  //read


  Rcount--;
  if (Rcount == 0) DB.V();

}
```

```
void Writer() {




  DB.P();

  //write

  DB.V();


}
```

Leser/Schreiber mit Semaphoren

```
void Reader() {

  Mutex.P();
  Rcount++;
  if (Rcount == 1) DB.P();
  Mutex.V();

  //read

  Mutex.P();
  Rcount--;
  if (Rcount == 0) DB.V();
  Mutex.V();
}
```

```
void Writer() {




  DB.P();

  //write

  DB.V();


}
```

Wegweiser

Zusammenspiel (Kommunikation) mehrerer Threads

Einige typische Probleme

Das Zuschneiden von Threadsystemen: Scheduling

- Gesichtspunkte für ...
- Round Robin
- Prioritäten
- Echtzeitsysteme

Scheduling

Aufgabe:

Entscheidung über Prozessorzuteilung

- an welchen Stellen (Zeitpunkte, Ereignisse, ...)
- nach welchem Verfahren

Scheduling

Kriterien

- Fairness: fairer Anteil an CPU für den Thread
- Effizienz: möglichst vollständige Auslastung der CPU
- Antwortzeiten: interaktiver Anwendungen
- Wartezeiten : von Hintergrundjobs
- Durchsatz: maximale Zahl von Aufgaben pro ZE
- Zusagen: ein Prozess muss spätestens *dann* fertig sein

Scheduling

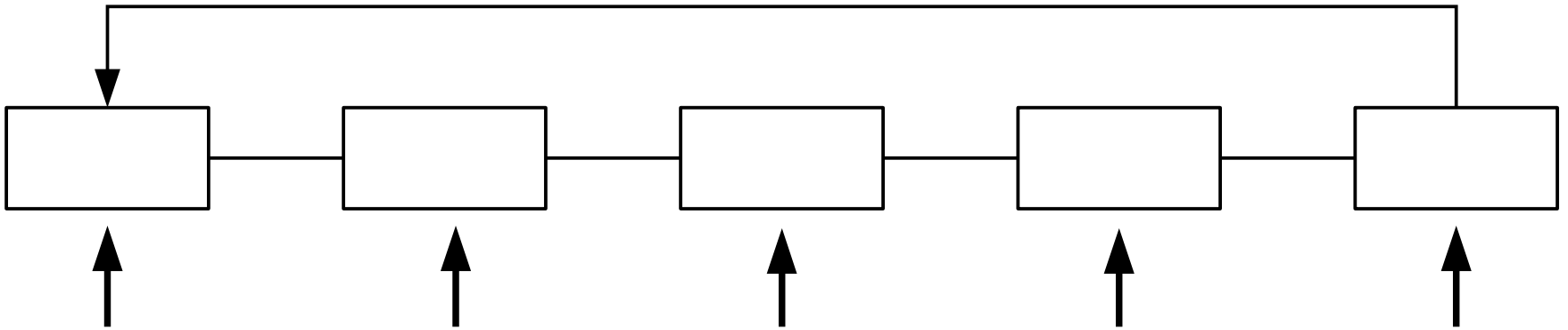
Kriterien

- Fairness: fairer Anteil an CPU für den Thread
- Effizienz: möglichst vollständige Auslastung der CPU
- Antwortzeiten: interaktiver Anwendungen
- Wartezeiten : von Hintergrundjobs
- Durchsatz: maximale Zahl von Aufgaben pro ZE
- Zusagen: ein Prozess muss spätestens *dann* fertig sein

Probleme

- Kriterien sind widersprüchlich
- Anwender-Threads sind in ihrem Verhalten oft unvorhersehbar

Round Robin mit Zeitscheiben



- jeder Thread erhält reihum eine ***Zeitscheibe*** (time slice), die er zu Ende führen kann

Wahl der Zeitscheibe

- zu kurz: CPU schaltet dauernd um
- zu lang: Antwortzeiten zu lang

Prioritätssteuerung

- Threads erhalten ein Attribut: Priorität
- höhere Priorität verschafft Vorrang vor niederer Priorität

Fragestellungen

- Zuweisung von Prioritäten
- Thread mit höherer Priorität als der rechnende wird bereit
  → Umschaltung: sofort, ... ? ("preemptive scheduling“)

➔ häufig: Kombination Round Robin und Prioritäten

Zuweisung von Prioritäten

- statisch ...
- dynamisch
  - Benutzer (Unix: “nice”)
  - Heuristiken im System
  - gezielte Vergabeverfahren

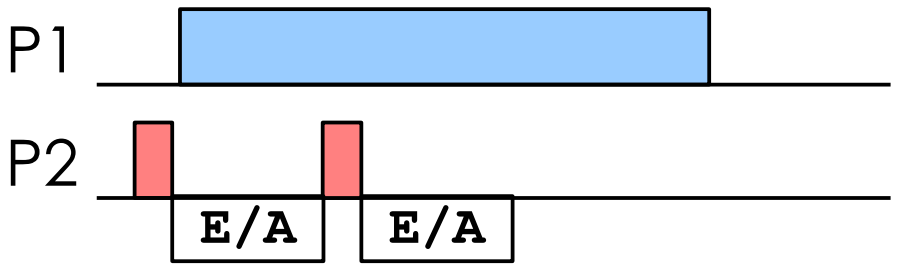
Beispiel für Heuristik

- hoher Anteil an Wartezeiten (z.B. E/A)
  hohe Priorität

➔ bessere Auslastung von E/A-Geräten

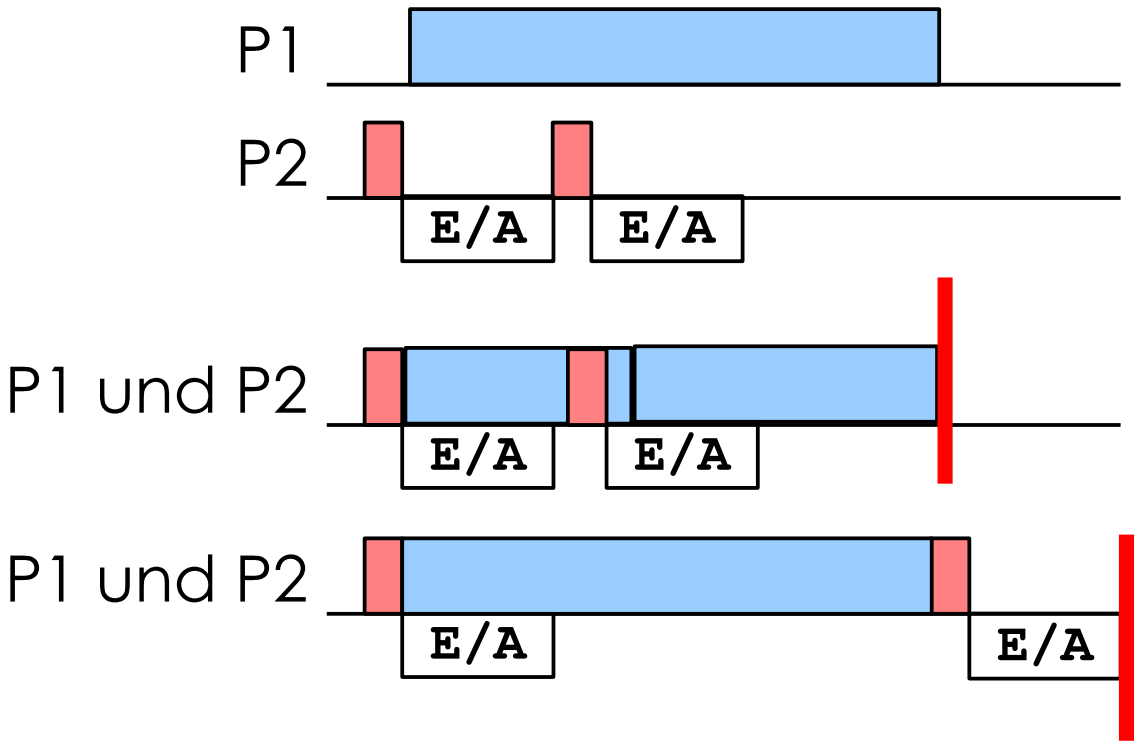
Beispiel

- Cobegin P2; P1 Coend
- P1 erst kurz Pause, dann lang CPU
- P2 kurz CPU, langsame EA



Beispiel

- Cobegin P2; P1 Coend
- P1 erst kurz Pause, dann lang CPU
- P2 kurz CPU, langsame EA



Zwei-Ebenen-Scheduling

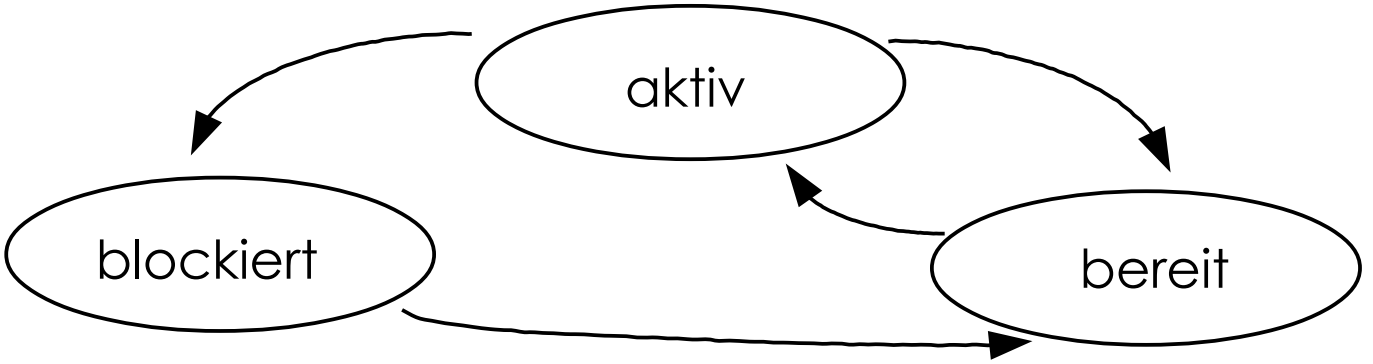
Ausgangspunkt

- Prozesse benötigen nicht nur CPU, sondern auch Speicher, manchmal müssen Prozesse ganz aus Hauptspeicher verdrängt werden.
- Nicht nur die Zuteilung von CPU zu Thread, sondern auch die Verdrängung auf Platte wird Gegenstand des Schedulings.

Zwei-Ebenen-Scheduling

Ausgangspunkt

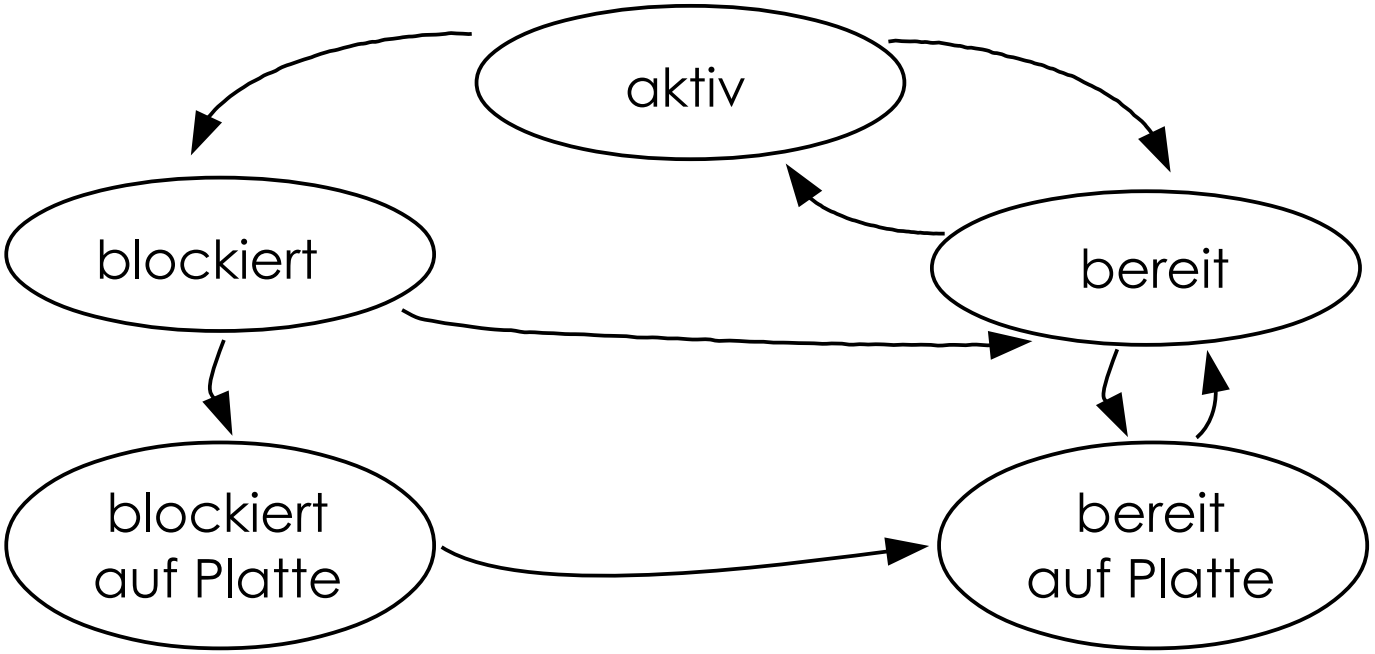
- Prozesse benötigen nicht nur CPU, sondern auch Speicher, manchmal müssen Prozesse ganz aus Hauptspeicher verdrängt werden.
- Nicht nur die Zuteilung von CPU zu Thread, sondern auch die Verdrängung auf Platte wird Gegenstand des Schedulings.



Zwei Ebenen Scheduling

Ausgangspunkt

- Prozesse benötigen nicht nur CPU, sondern auch Speicher, manchmal müssen Prozesse ganz aus Hauptspeicher verdrängt werden.
- Nicht nur die Zuteilung von CPU zu Thread, sondern auch die Verdrängung auf Platte wird Gegenstand des Schedulings.



„Prioritätsumkehr“ (Priority Inversion)

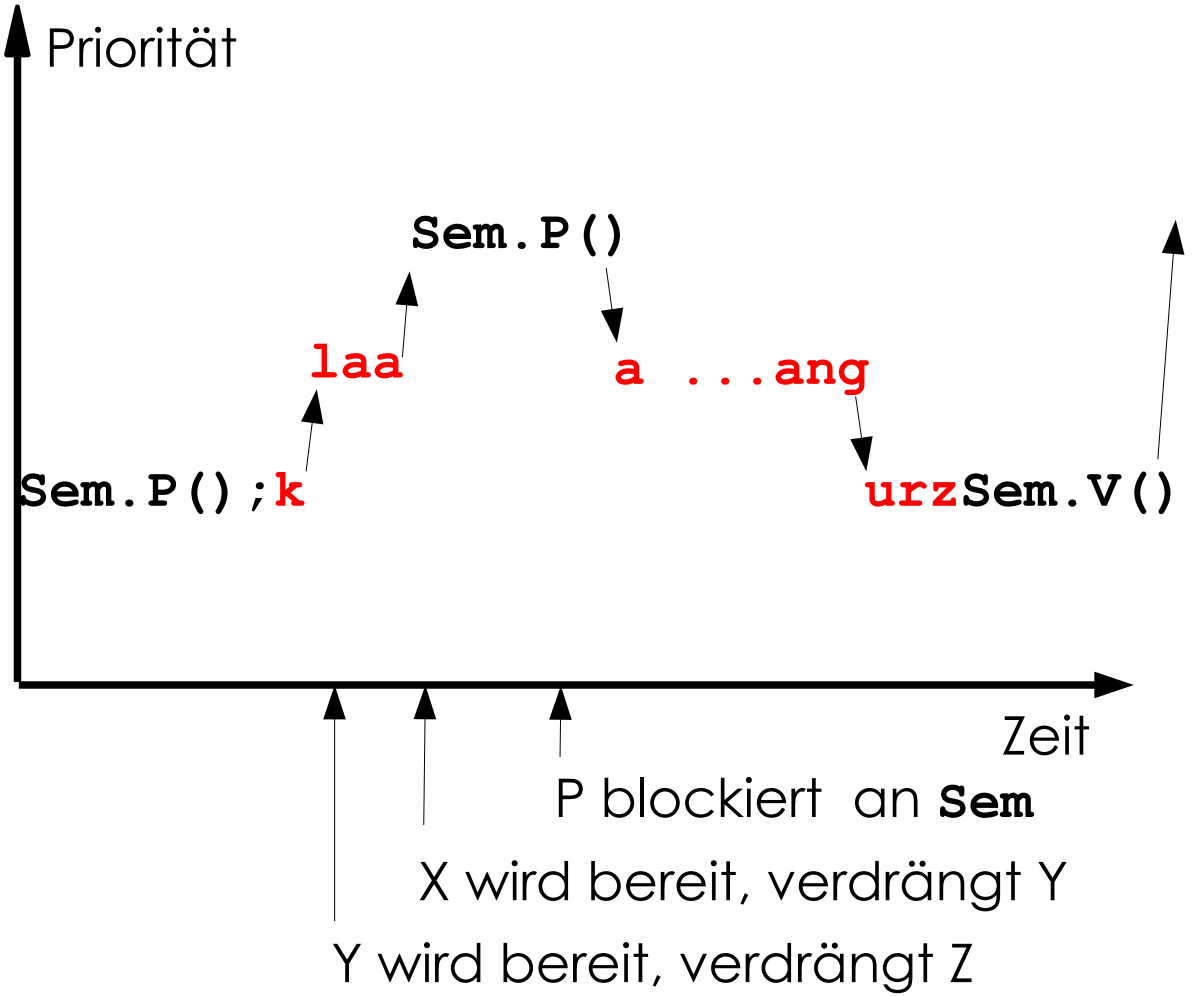
Beispielszenario:

- drei Threads X, Y, Z
- X höchste, Z niedrigste Priorität
- werden bereit in der Reihenfolge: Z, Y, X

```
cobegin
  //X:
  {Sem.P();kurz;Sem.V()};
  //Y:
  {laaaaaaaaaaaaaaaang};
  //Z:
  {Sem.P();kurz;Sem.V()};
coend
```

„Prioritätsumkehr“ (Priority Inversion)

```
cobegin
  //X:
  {Sem.P();kurz;Sem.V()};
  //Y:
  {laaaa ... aaaaaaaang};
  //Z:
  {Sem.P();kurz;Sem.V()};
coend
```



X hat höchste Priorität, kann aber nicht laufen,
weil es an von Z gehaltenem Semaphor blockiert und
Y läuft und damit R den Semaphor nicht freigeben kann.

Zusammenfassung Threads/Prozesse

- (Prozesse) Threads sind ein mächtiges Konzept zur Konstruktion von Systemen, die mit asynchonen Ereignissen umgehen können müssen.
- Kooperierende parallele Threads müssen sorgfältig synchronisiert werden.
  Fehler dabei sind extrem schwer zu finden, da zeitabhängig.
- Bei vielen Entscheidungen sind Kriterien zu berücksichtigen, die einander widersprechen.
  Ein solider Entwurf muss die Diskussion der "Trade Offs" einschließen.

Bald in dieser Vorlesung:

- „Echtzeitsysteme“ und Scheduling
- Interaktion mit Speicher und Adressierung
- Kommunikation von Prozessen in einem verteilten System