Work in progress

Überblick Nebenläufigkeit

Ute Bormann, TI2

2023-10-13

Inhalt

1. Grundlegendes
2. Kritische Abschnitte vs. Leser/Schreiber
3. Erzeuger/Verbraucher
4. Speisende Philosophen

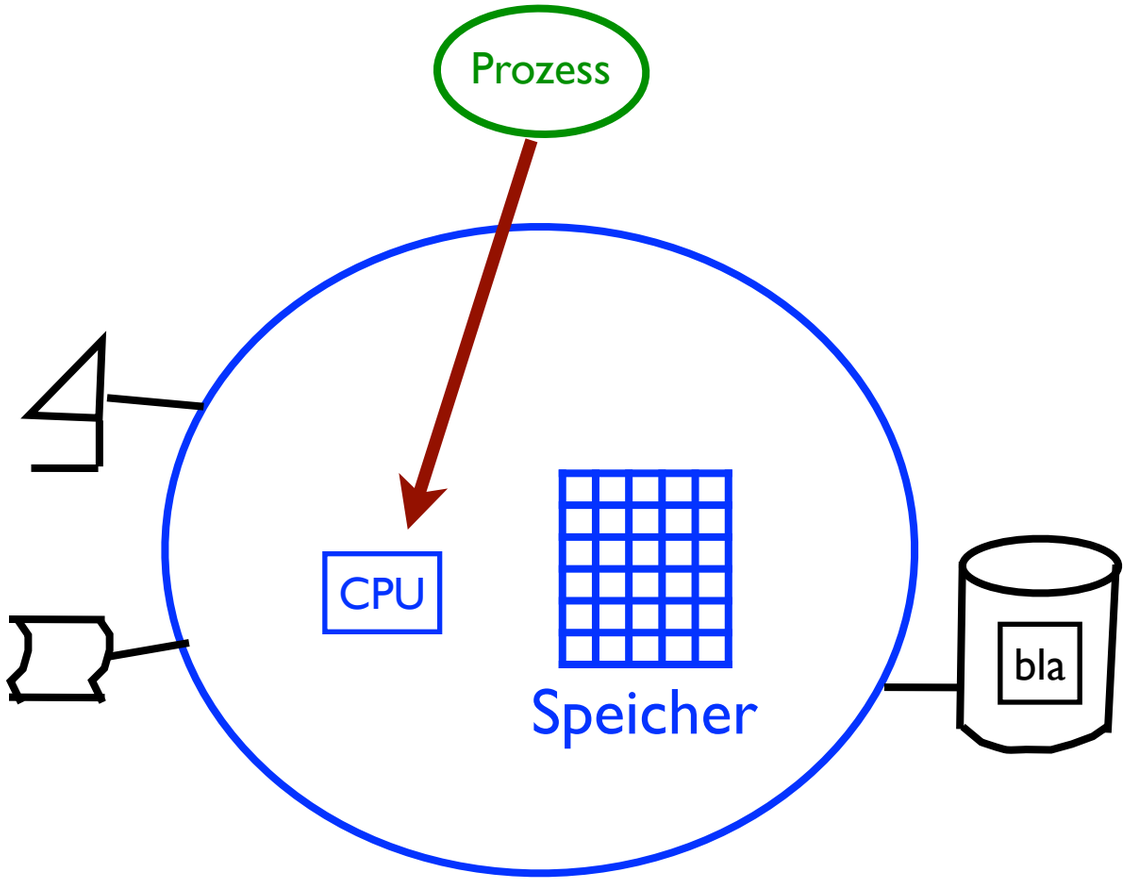
Teil 1:
Grundlegendes

14 Nebenläufigkeit, Ute Bormann

Was bisher geschah:

Einführung in Betriebssysteme

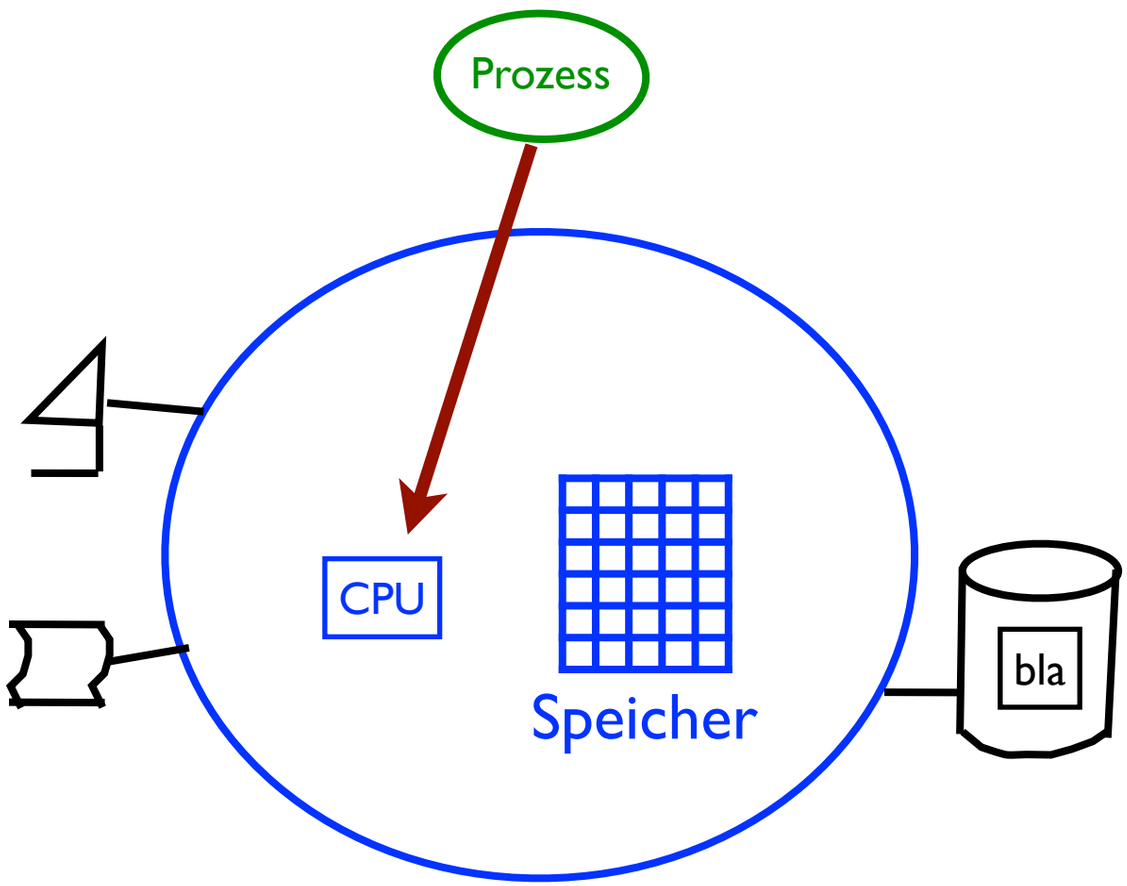
- Verwaltung von Betriebsmitteln



Was bisher geschah:

Einführung in Betriebssysteme

- Verwaltung von Betriebsmitteln



- Einzelner Prozess nutzt Betriebsmittel exklusiv
- Einzelnes Programm wird sequentiell abgearbeitet

  ⇒ „klassische“ Programmierung

- Ausführung ist vorhersehbar (deterministisch)

  ⇒ gleiche Eingabe führt zu:

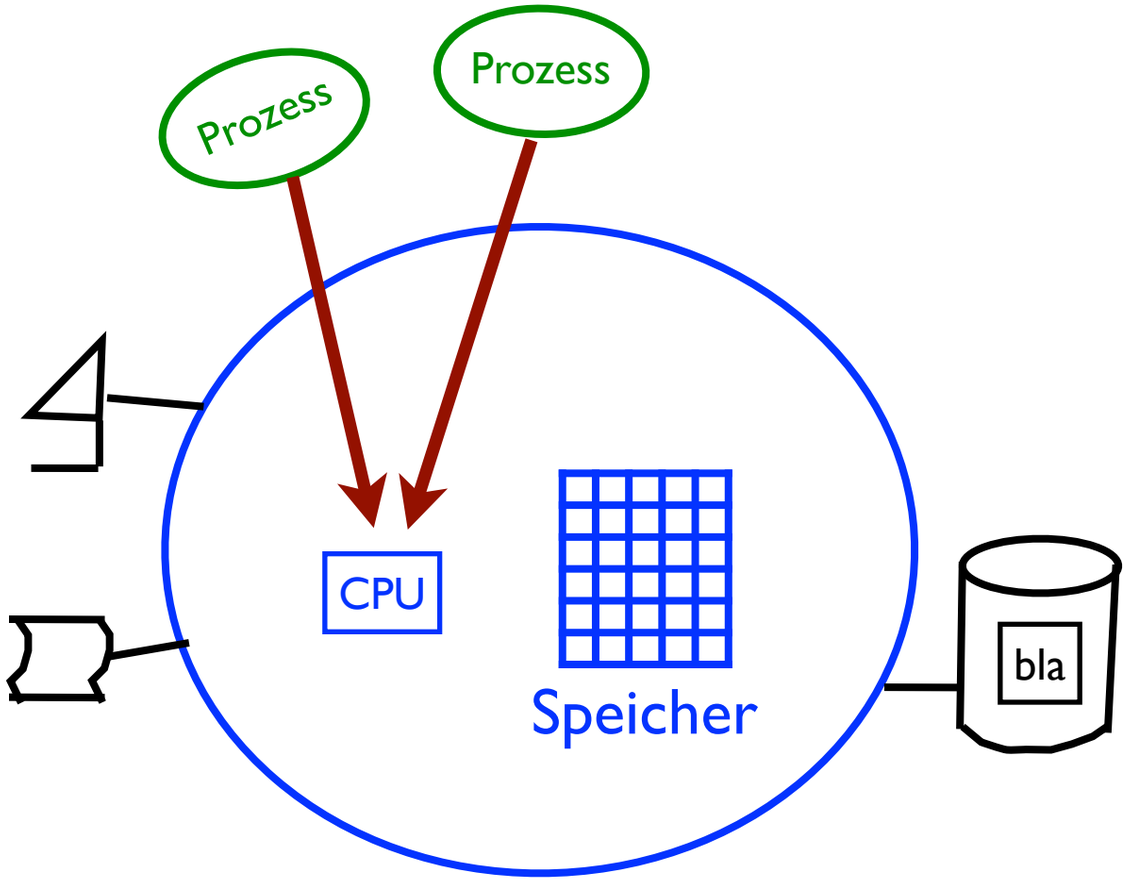
  - gleichem Programmablauf
  - gleichem Ergebnis

- Aber: zu inflexibel (Ressourcenverschwendung, u.U. lange Wartezeiten...)

Was bisher geschah:

Einführung in Betriebssysteme

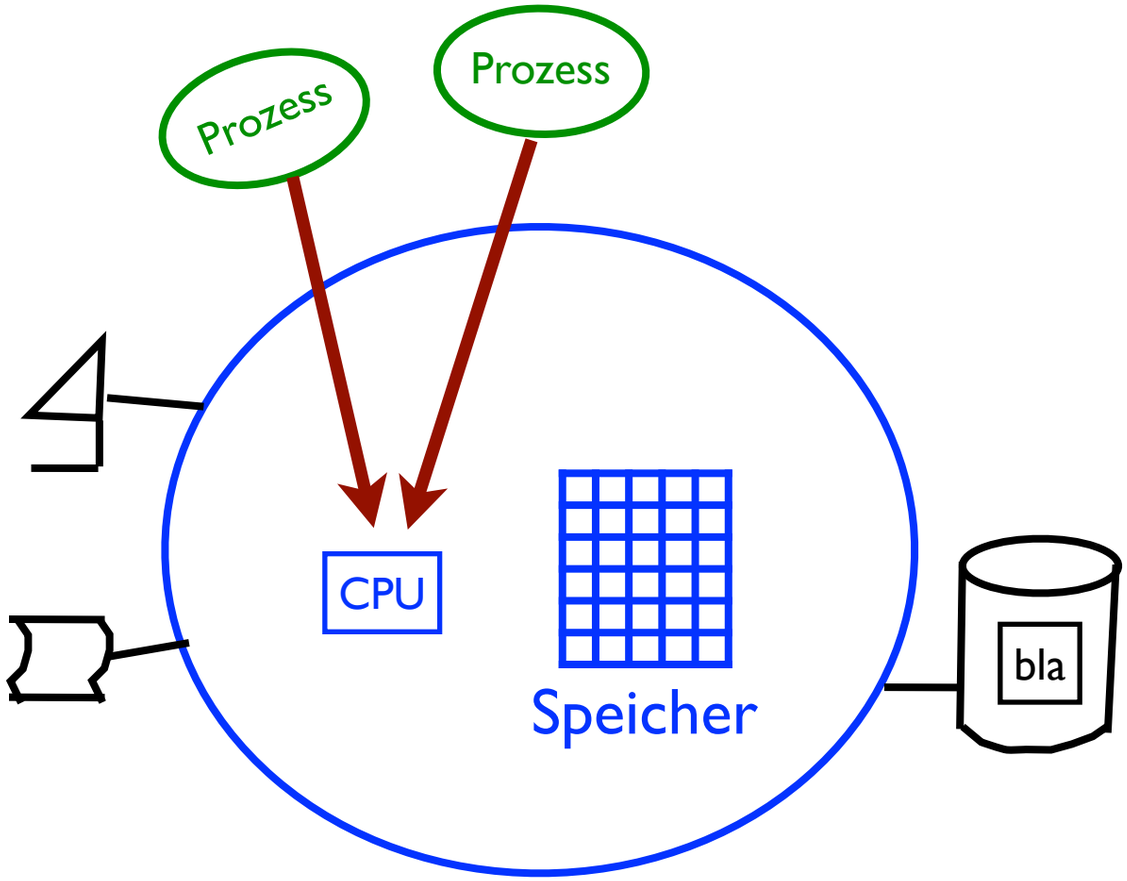
- Verwaltung von Betriebsmitteln
- Verwaltung von Prozessen



Was bisher geschah:

Einführung in Betriebssysteme

- Verwaltung von Betriebsmitteln
- Verwaltung von Prozessen



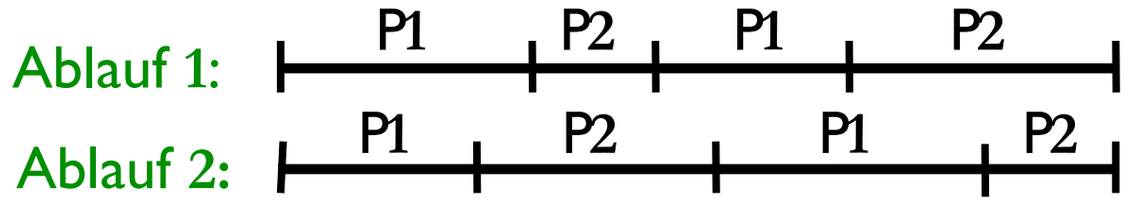
- Mehrere sequentielle Prozesse teilen sich die CPU

  ⇒ Scheduler entscheidet über Zuteilung (z.B. nach Prioritäten)

  ⇒ Verschachtelte Ausführung

- CPU-Umschaltungen i.d.R. nicht genau vorhersehbar

  ⇒ Gesamtablauf nicht-deterministisch



⇒ Nebenläufigkeit (Concurrency)

⇒ aber keine zeitgleiche Ausführung (lediglich Quasi-Parallelität)

- Unkritisch, solange sich Prozesse nicht gegenseitig beeinflussen

  ⇒ Einzelablauf weiter deterministisch

Wechselwirkungen zwischen Prozessen

a) Unbeabsichtigt ⇒ Konkurrenz

- Zugriff auf gemeinsame Betriebsmittel
  - Lesend? Unkritisch, solange reproduzierbar

	Plattenblock	Speicherzelle	Datei	Tastatur	Drucker
Lesen	✓	✓	(✓)	?	(—)

Wechselwirkungen zwischen Prozessen

a) Unbeabsichtigt ⇒ Konkurrenz

- Zugriff auf gemeinsame Betriebsmittel
  - Lesend? Unkritisch, solange reproduzierbar
  - Schreibend? In der Regel problematisch

	Plattenblock	Speicherzelle	Datei	Tastatur	Drucker
Lesen	✓	✓	(✓)	?	(—)
Schreiben	?	?	(?)	—	?

- Wiederholung Beispiel Pipe

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

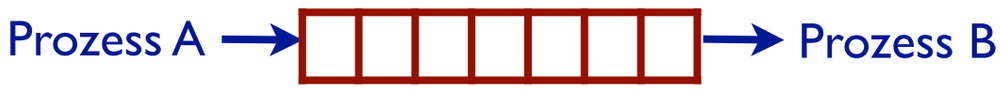
```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange



- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Maschineninstruktionen, z.B.

```
LOAD count     =7
                          LOAD count     =7
                          SUB 1          =6
                          STORE count    =6
ADD 1          =8
STORE count    =8
```

⇒ Wert von count falsch

Betriebssystem reduziert viele dieser Probleme:

- Eigener Adressraum pro Prozess
- Zugriffsschutz für Dateien
- Kein direkter Zugriff auf Geräte/Kern-Datenstrukturen
- Reservierung von Betriebsmitteln (vs. Spooling)

Betriebssystem reduziert viele dieser Probleme:

- Eigener Adressraum pro Prozess
- Zugriffsschutz für Dateien
- Kein direkter Zugriff auf Geräte/Kern-Datenstrukturen
- Reservierung von Betriebsmitteln (vs. Spooling)

Allerdings: Betriebssystem-interne Verwaltungsstrukturen sind auch gemeinsam genutzte Betriebsmittel:

a) Probleme möglichst vermeiden

⇒ z.B. Nicht-Präemption im Einprozessor-Unix-Kern

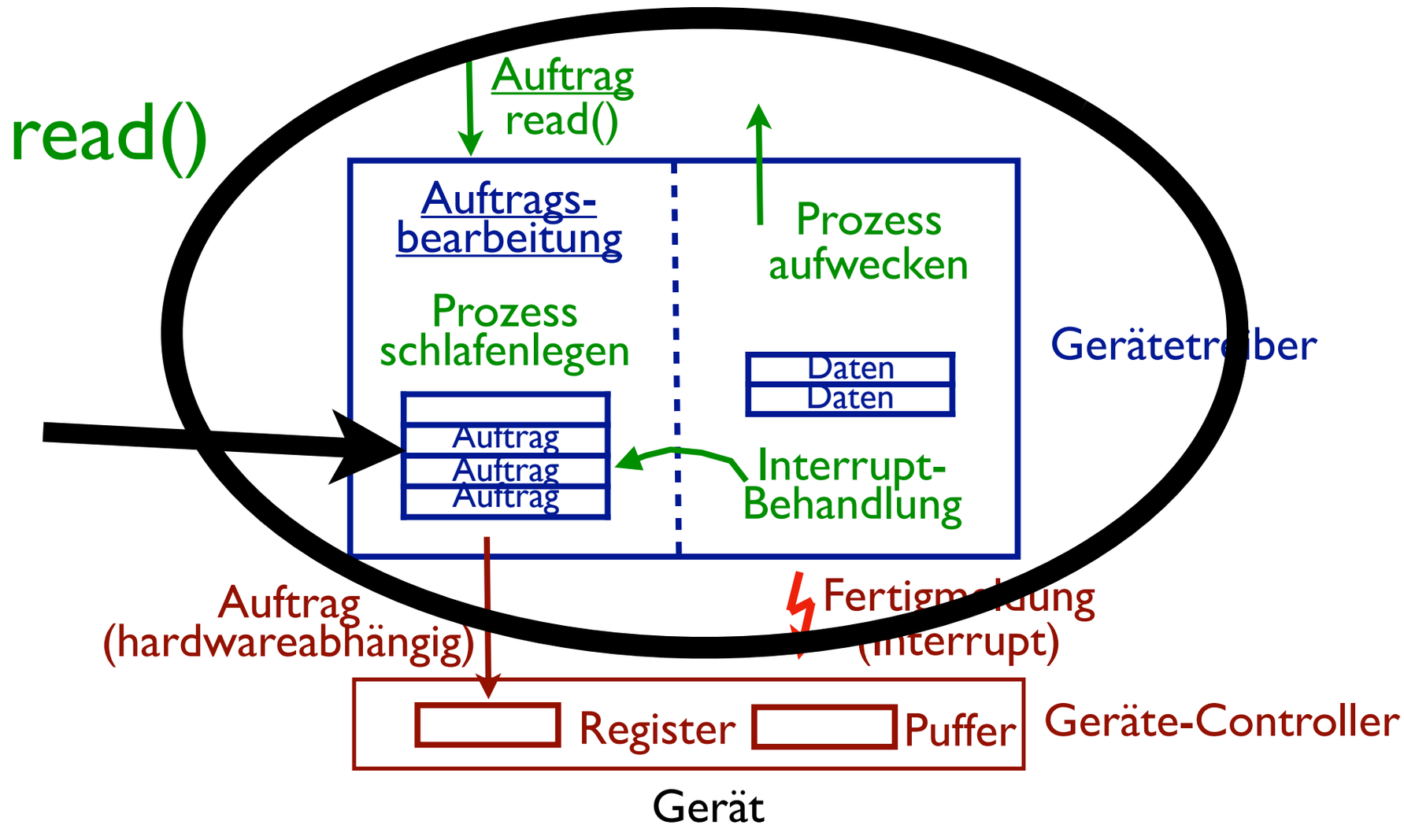
(nicht zwei Systemaufrufe gleichzeitig, auch bei Realisierung von Pipes)

→ b) Wo erforderlich Synchronisationsmaßnahmen vorsehen, z.B. bei Systemaufruf vs. Interrupthandler

Beispiel:

Gerätetreiber (Allgemeines)

- Code innerhalb des Betriebssystemkerns zur Geräteverwaltung
- Ein Treiber pro Gerätetyp (Major Number)
- Parametrisiert mit Minor Number zur Auswahl des konkreten Geräts
- Grundsätzliche Arbeitsweise (Beispiele: read()/write(), vereinfacht)



Fragen – Teil 1

- Skizziere kurz einige Probleme des *nebenläufigen* Zugriffs auf Betriebsmittel.
- Grenze die Begriffe *Nebenläufigkeit*, *Quasi-Parallelität* und *Parallelität* voneinander ab.
- Was verstehen wir unter *Nichtdeterminismus*?

Teil 2:
Kritische Abschnitte vs. Leser/Schreiber

Aber nicht nur unbeabsichtigte Wechselwirkungen...

b) Beabsichtigt ⇒ Kooperation

- Interprozesskommunikation über:
  - gemeinsame Dateien
  - gemeinsame Page Frames (Shared Memory)
  - explizite Kommunikationsobjekte (Pipes, Sockets, ...)

⇒ nur teilweise durch Betriebssystem geschützt (z.B. Pipes, Sockets)

⇒ ggf. Synchronisationsmaßnahmen in den Prozessen selbst erforderlich (ggf. unterstützt von Betriebssystem)

Aber nicht nur unbeabsichtigte Wechselwirkungen...

b) Beabsichtigt ⇒ Kooperation

- Interprozesskommunikation über:
  - gemeinsame Dateien
  - gemeinsame Page Frames (Shared Memory)
  - explizite Kommunikationsobjekte (Pipes, Sockets, ...)

  ⇒ nur teilweise durch Betriebssystem geschützt (z.B. Pipes, Sockets)

  ⇒ ggf. Synchronisationsmaßnahmen in den Prozessen selbst erforderlich (ggf. unterstützt von Betriebssystem)

Einige typische Kommunikationssituationen betrachten

- stellvertretend für viele Varianten
- im folgenden als Anwendungsbeispiele genutzt
  ⇒ teilweise realistisch/teilweise Literatur
- Lösung des einen mag anderes hervorbringen
  ⇒ nicht unabhängig betrachten

1) Der kritische Abschnitt (Critical Section)

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

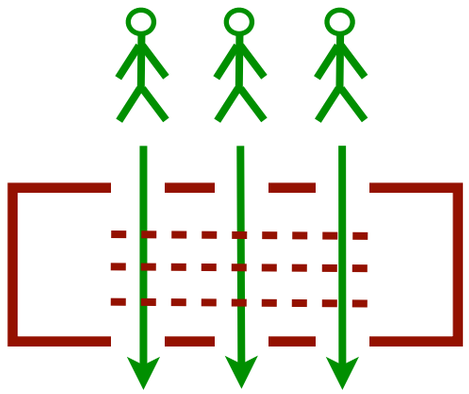
Maschineninstruktionen, z.B.

```
LOAD count    =7
                          LOAD count    =7
                          SUB 1         =6
                          STORE count   =6
ADD 1         =8
STORE count   =8
```

⇒ Wert von count falsch

Kritische Abschnitte

- Ungeschützter kritischer Abschnitt erzeugt Wettlaufbedingung (Race Condition)



kritischer Abschnitt

⇒ Ergebnis hängt von genauer Ausführungsreihenfolge (Verschachtelung) ab

⇒ U.U. unerwünschte Nicht-Determiniertheit

- (Achtung: auch verschiedene sequentielle Schreibvorgänge erzeugen u.U. unterschiedliche Ergebnisse ⇒ o.k.)

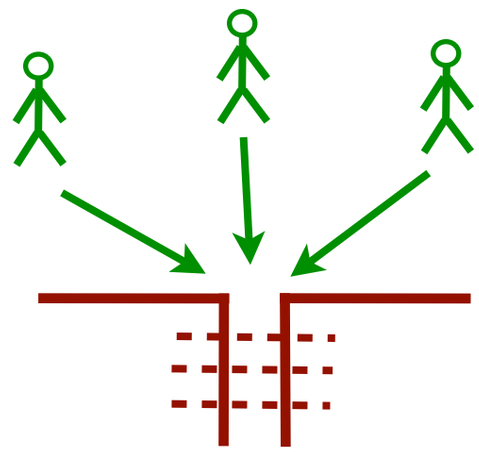
- Abhilfe: Während des Zugriffs eines Prozesses kein weiterer Prozess zulässig

  ⇒ Unteilbarkeit/Atomarität gegenüber anderen Prozessen

  ⇒ Entweder Prozess1 oder Prozess2 oder ...

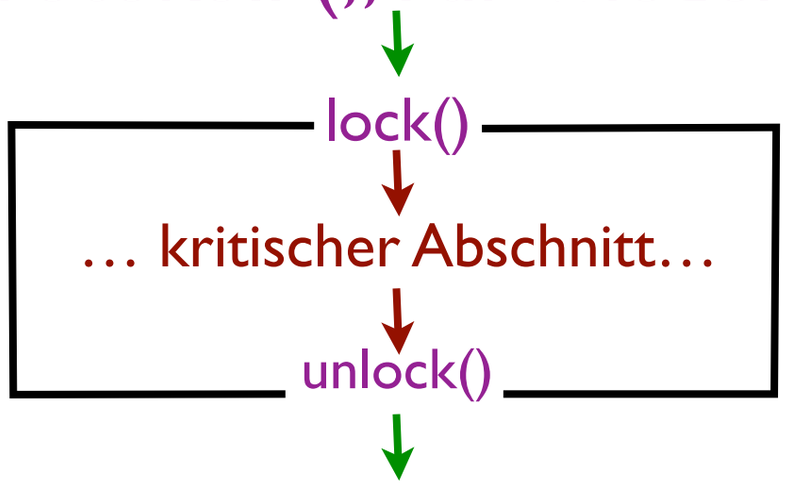
Gegenseitiger Ausschluss (Mutual Exclusion)

Mehrseitige Synchronisation (Prozesse gegenseitig voneinander abhängig)



kritischer Abschnitt

- Erkennen: Was muss überhaupt geschützt werden?
  - Verändern von gemeinsamen „Daten“
  - Zugriff auf exklusiv zu nutzende Geräte ...
- Gewährleisten des gegenseitigen Ausschlusses:
  - Sichern bei Betreten des kritischen Abschnitts
    ⇒ Eintrittsprotokoll („Tür hinter sich zuschließen“)
  - Freigeben bei Verlassen
    ⇒ Austrittsprotokoll („Tür wieder öffnen“)



⇒ Verschiedene Verfahren zur Realisierung von lock()/unlock()

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
lock();
count++;
unlock();
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
lock();
count--;
unlock();
```

Algorithmus hat jedoch noch weitere Nebenläufigkeitsprobleme

2) Das Leser-/Schreiber-Problem (Reader-/Writer-Problem)

- Datenbestand, auf den verschiedene Prozesse lesend bzw. schreibend zugreifen

  z.B. Datenbank (Datensätze mit mehreren Komponenten)

- Lesende Zugriffe stören sich nicht gegenseitig
  ⇒ beliebig viele Leser nebenläufig zulässig

- Nebenläufiges Schreiben ⇒ Falsche Werte möglich
  ⇒ Schreibender Zugriff ist kritischer Abschnitt innerhalb des Programms

Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};

Cursor::move(Point q) {
  p.x = q.x;
  p.y = q.y;
};

Point Cursor::position() {
  Point i;
  i.x = p.x;
  i.y = p.y;
  return(i);
};
```

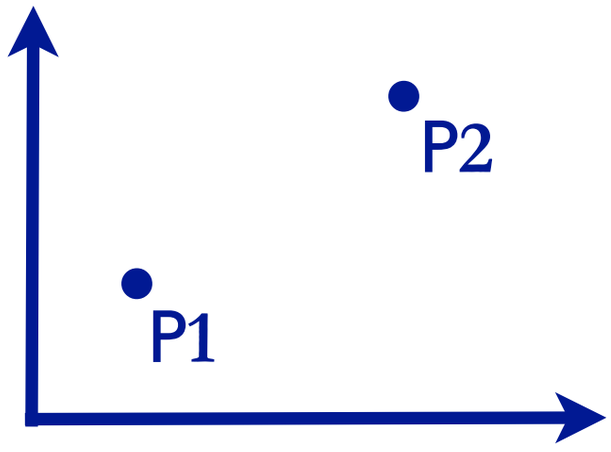
Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};

Cursor::move(Point q) {
  p.x = q.x;
  p.y = q.y;
};

Point Cursor::position() {
  Point i;
  i.x = p.x;
  i.y = p.y;
  return(i);
};
```



Schreiben P1 vs. Schreiben P2

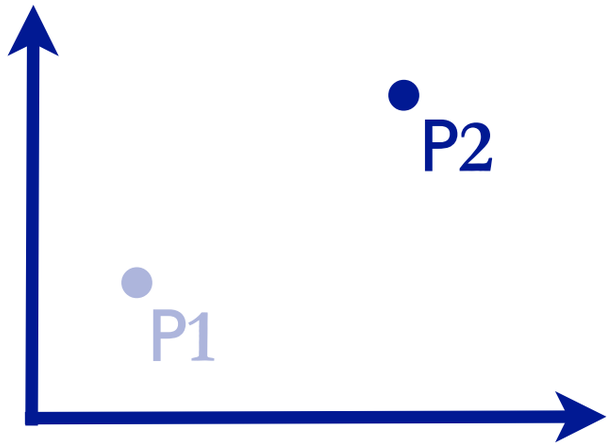
Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};

Cursor::move(Point q) {
  p.x = q.x;
  p.y = q.y;
};

Point Cursor::position() {
  Point i;
  i.x = p.x;
  i.y = p.y;
  return(i);
};
```



Schreiben vs. Schreiben

x1

y1

x2

y2

P2 ✓

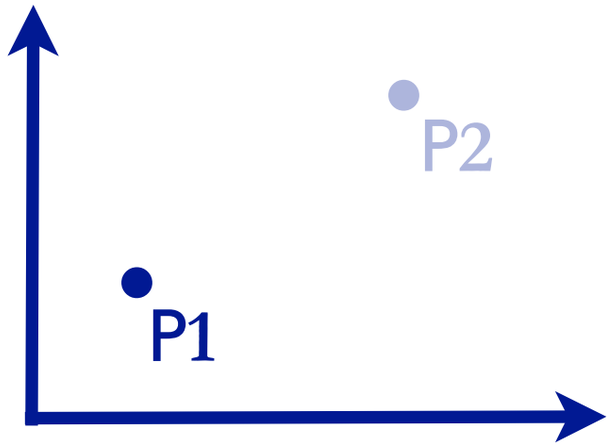
Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};

Cursor::move(Point q) {
  p.x = q.x;
  p.y = q.y;
};

Point Cursor::position() {
  Point i;
  i.x = p.x;
  i.y = p.y;
  return(i);
};
```



Schreiben	vs.	Schreiben
		x2
		y2
x1		
y1		

P1 ✓

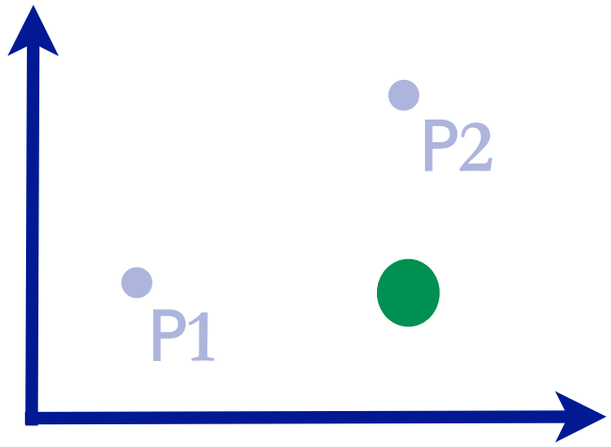
Beispiel: Cursorbewegung

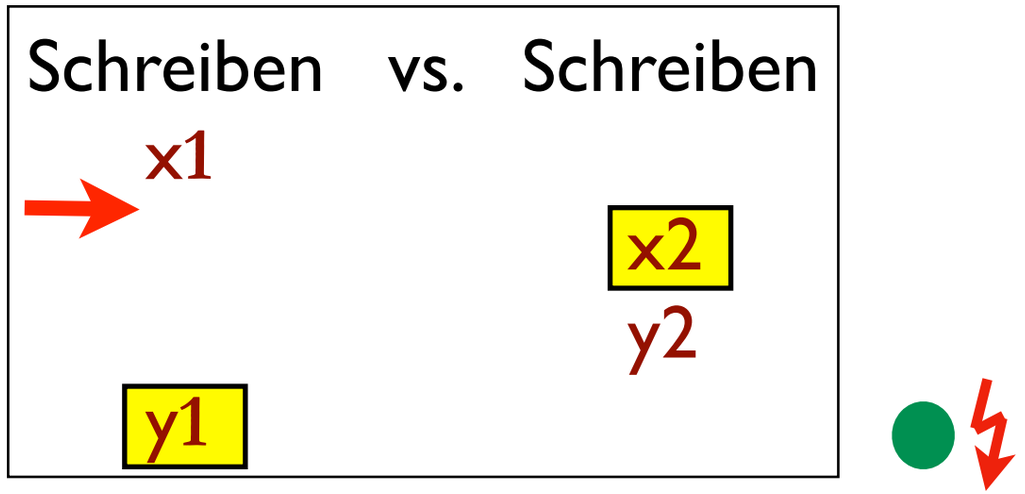
```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};

Cursor::move(Point q) {
  p.x = q.x;
  p.y = q.y;
};

Point Cursor::position() {
  Point i;
  i.x = p.x;
  i.y = p.y;
  return(i);
};
```





2) Das Leser-/Schreiber-Problem (Reader-/Writer-Problem)

- Datenbestand, auf den verschiedene Prozesse lesend bzw. schreibend zugreifen

  z.B. Datenbank (Datensätze mit mehreren Komponenten)

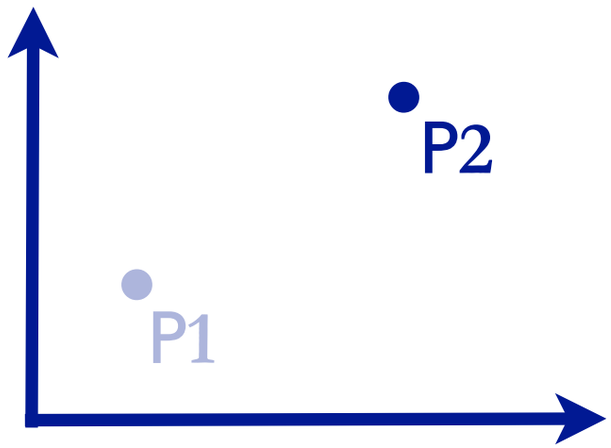
- Lesende Zugriffe stören sich nicht gegenseitig
  ⇒ beliebig viele Leser nebenläufig zulässig

- Nebenläufiges Schreiben ⇒ Falsche Werte möglich
  ⇒ Schreibender Zugriff ist kritischer Abschnitt innerhalb des Programms
  ⇒ Abhilfe: z.B. Verwendung eines Locking-Verfahrens

Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};
```




```
Cursor::move(Point q) {
  lock();
  p.x = q.x;
  p.y = q.y;
  unlock();
};

Point Cursor::position() {
  Point i;

  i.x = p.x;
  i.y = p.y;

  return(i);
};
```

Schreiben	vs.	Schreiben
x1		
y1		
		x2
		y2

P2 ✓

2) Das Leser-/Schreiber-Problem (Reader-/Writer-Problem)

- Datenbestand, auf den verschiedene Prozesse lesend bzw. schreibend zugreifen

  z.B. Datenbank (Datensätze mit mehreren Komponenten)

- Lesende Zugriffe stören sich nicht gegenseitig
  ⇒ beliebig viele Leser nebenläufig zulässig

- Nebenläufiges Schreiben ⇒ Falsche Werte möglich
  ⇒ Schreibender Zugriff ist kritischer Abschnitt innerhalb des Programms
  ⇒ Abhilfe: z.B. Verwendung eines Locking-Verfahrens

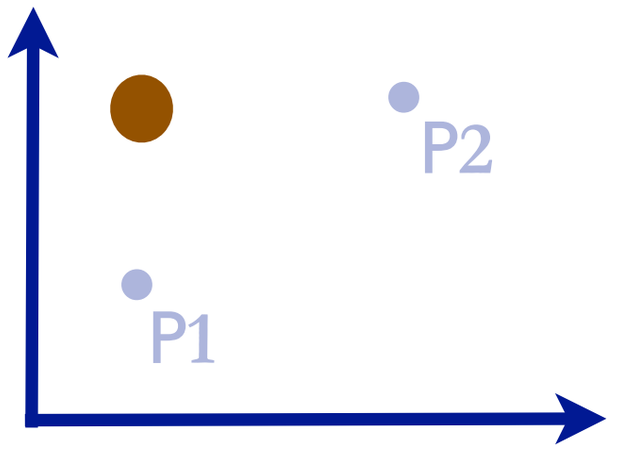
- Während des Schreibens auch kein Lesen zulässig
  (bzw. während des Lesens kein Schreiben)

  ⇒ komplexe Datenstruktur könnte zeitweise inkonsistent sein

Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};
```

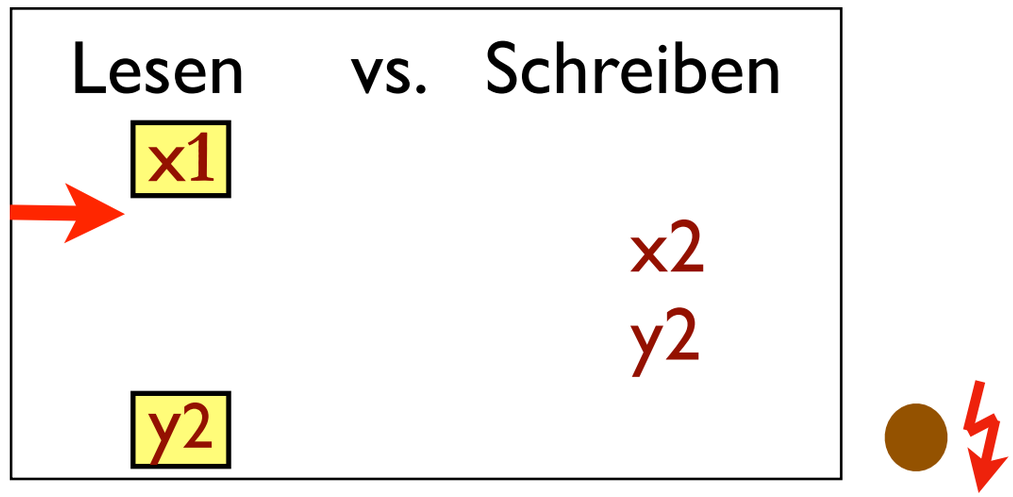



```
Cursor::move(Point q) {
  lock();
  p.x = q.x;
  p.y = q.y;
  unlock();
};

Point Cursor::position() {
  Point i;

  i.x = p.x;
  i.y = p.y;

  return(i);
};
```



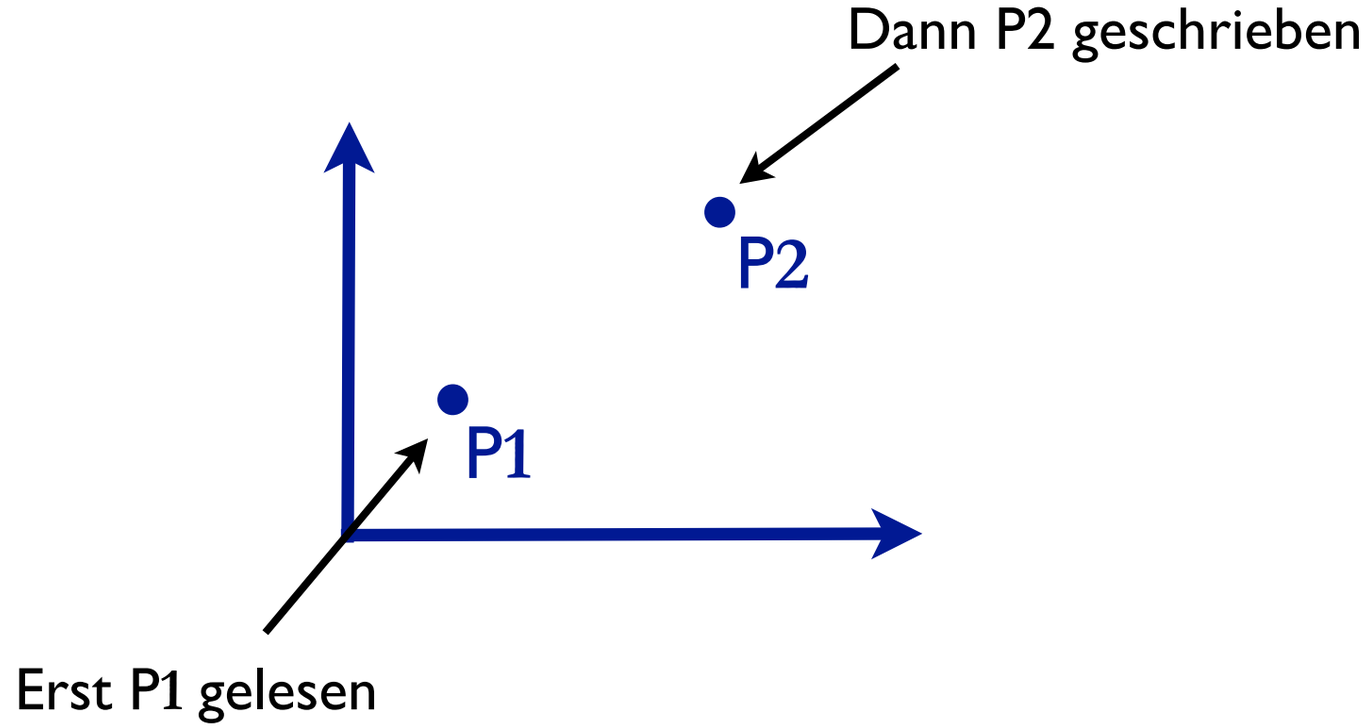
Beispiel: Cursorbewegung

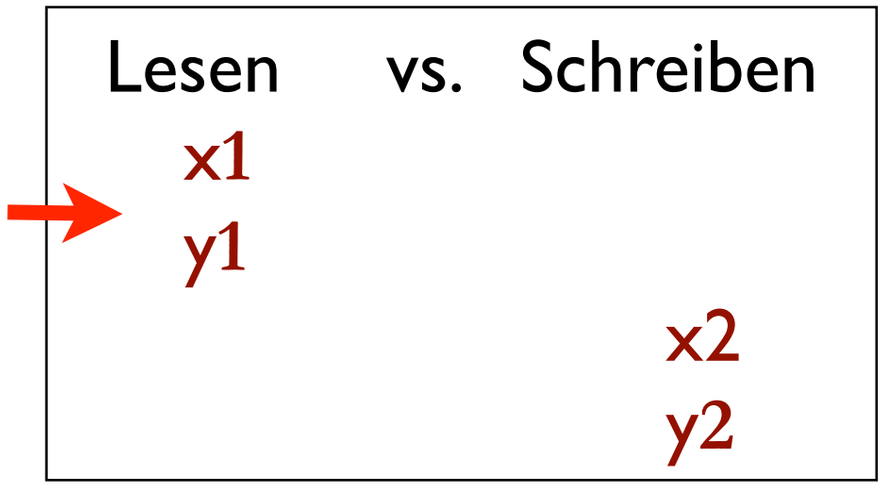
```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};
```

```
Cursor::move(Point q) {
  lock();
  p.x = q.x;
  p.y = q.y;
  unlock();
};

Point Cursor::position() {
  Point i;
  lock();
  i.x = p.x;
  i.y = p.y;
  unlock();
  return(i);
};
```





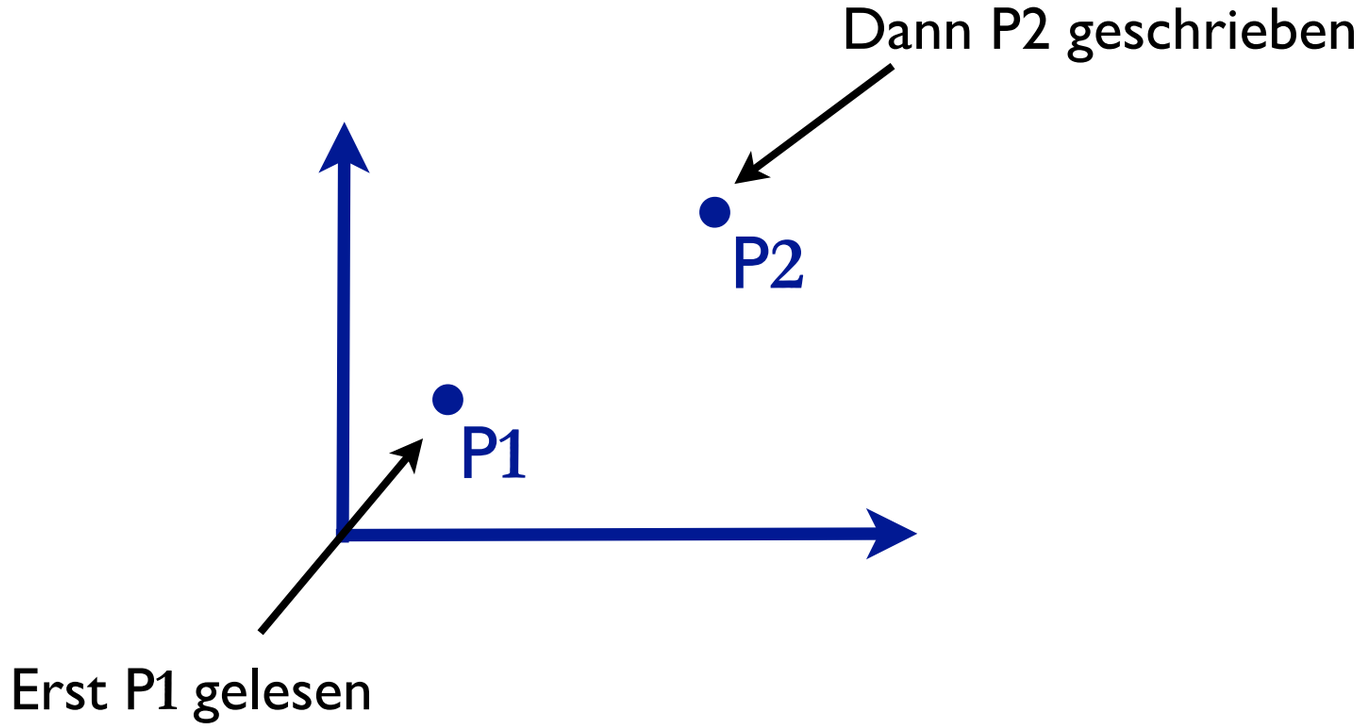
Beispiel: Cursorbewegung

```
class Point {
  public: int x;
          int y;
};

class Cursor {
  Point p;
  Cursor();
  move(Point);
  Point position():
};
```

```
Cursor::move(Point q) {
  lock();
  p.x = q.x;
  p.y = q.y;
  unlock();
};

Point Cursor::position() {
  Point i;
  lock();
  i.x = p.x;
  i.y = p.y;
  unlock();
  return(i);
};
```



Lesen	vs.	Schreiben
x1		
y1		
		x2
		y2

Damit allerdings zuviel geschützt

⇒ nun kein nebenläufiges Lesen mehr zulässig

⇒ noch keine Lösung des Leser-/Schreiber-Problems!

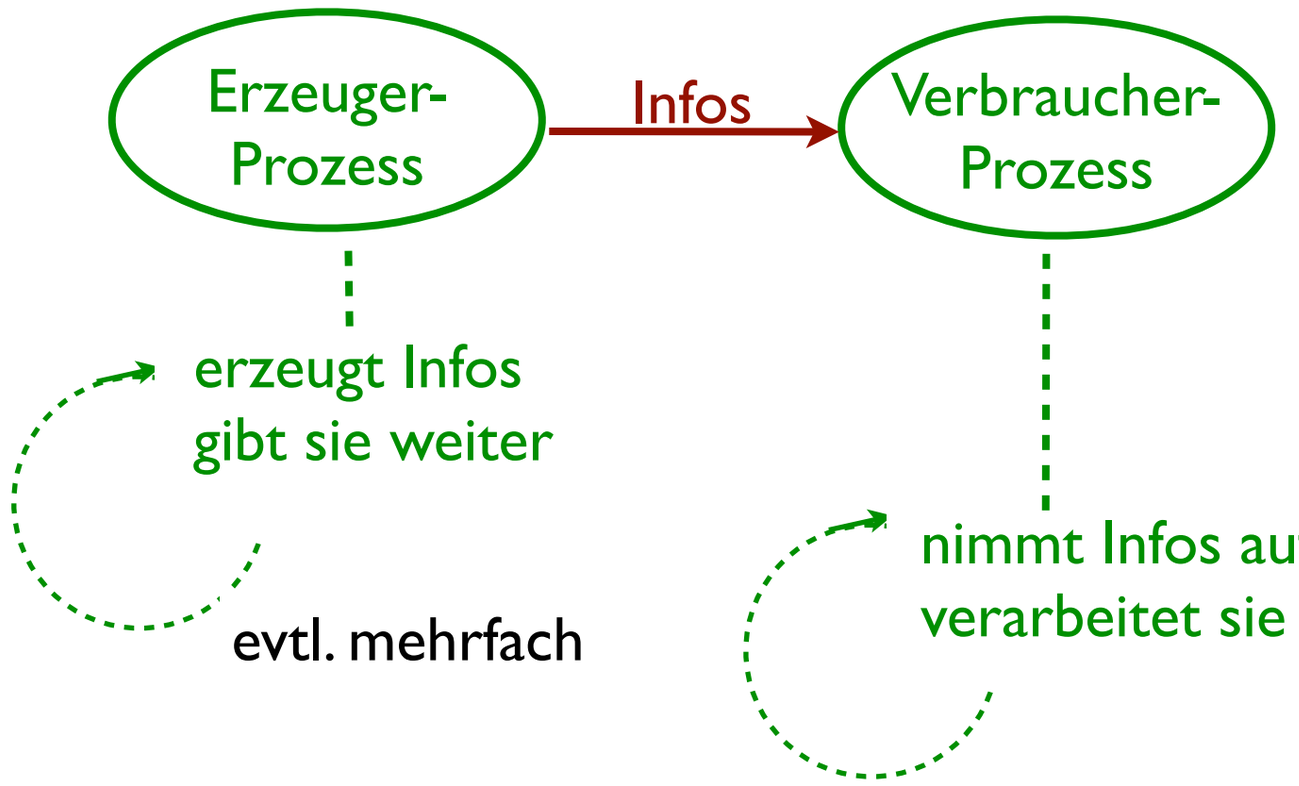
Fragen – Teil 2

- Welche Nebenläufigkeitseigenschaften bzw. -probleme werden durch die folgenden „klassischen“ Szenarien ausgedrückt:
  - Kritischer Abschnitt *(Critical Section)*,
  - Leser/Schreiber *(Reader/Writer)*,
- Was versteht man unter *mehrseitiger* Synchronisation? Gib ein Anwendungsbeispiel an.

Teil 3:
Erzeuger/Verbraucher

14 Nebenläufigkeit, Ute Bormann

3) Das Erzeuger-/Verbraucher-Problem (Producer-/Consumer-Problem)



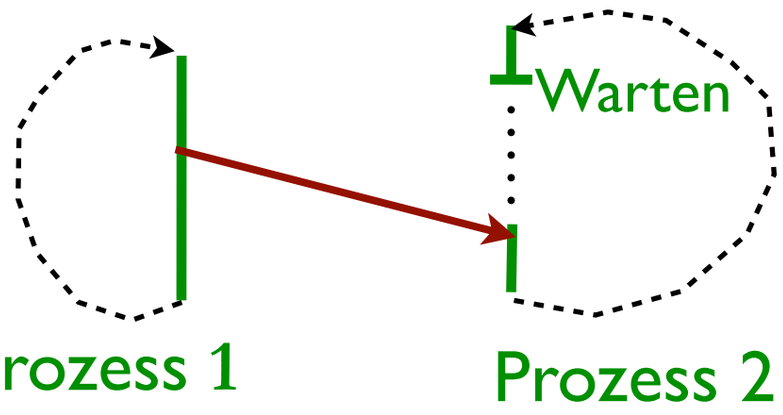
- Gemeinsames Kommunikationsobjekt erforderlich (Warteschlange: Speicherbereich, Datei, Pipe,...)

- Erst schreiben, dann lesen

  ⇒ Gewisse Sequentialisierung erforderlich

  ⇒ Einseitige Synchronisation

  (Verbraucher muss auf Erzeuger warten, nicht umgekehrt)

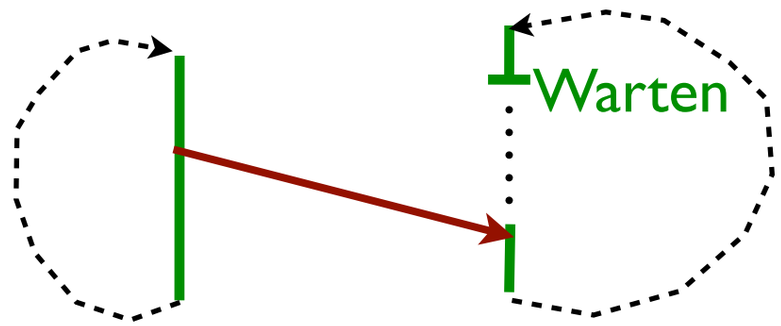


- Erst schreiben, dann lesen

  ⇒ Gewisse Sequentialisierung erforderlich

  ⇒ Einseitige Synchronisation

  (Verbraucher muss auf Erzeuger warten, nicht umgekehrt)

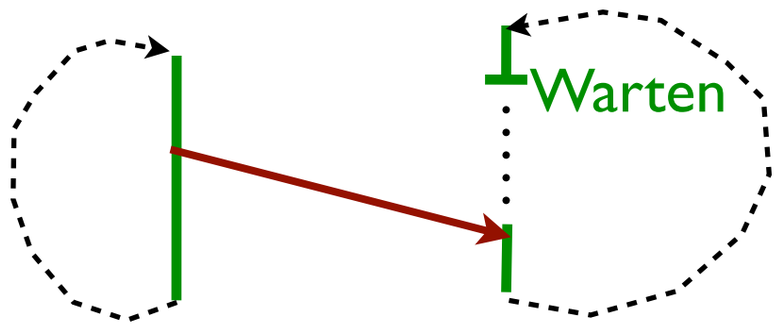


- Bei beschränktem Pufferplatz auch einseitige Synchronisation zwischen Verbraucher und Erzeuger erforderlich

- Erst schreiben, dann lesen

  ⇒ Gewisse Sequentialisierung erforderlich

  ⇒ Einseitige Synchronisation
  (Verbraucher muss auf Erzeuger warten, nicht umgekehrt)



- Bei beschränktem Pufferplatz auch einseitige Synchronisation zwischen Verbraucher und Erzeuger erforderlich

- Realisierung kann mehrseitige Synchronisation erfordern

  (z.B. Shared Memory als Kommunikationsmedium
  + Verwaltung eines Füllstandszählers)

  ⇒ ggf. kritischer Abschnitt

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

einseitige Synchronisation

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Einige potentielle Probleme

a) Nicht-Unteilbarkeit von count++/count--

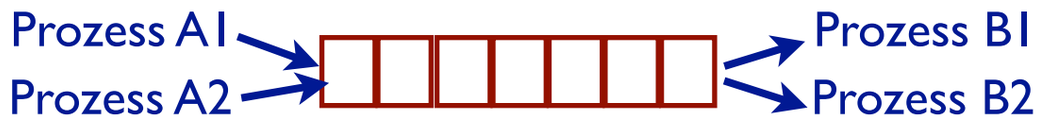
⇒ Count-Zähler führt nicht korrekt Buch ⇒ kritischer Abschnitt

⇒ Elemente in scheinbar leerer Warteschlange,
Platz in scheinbar voller Warteschlange

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount



Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Einige potentielle Probleme:

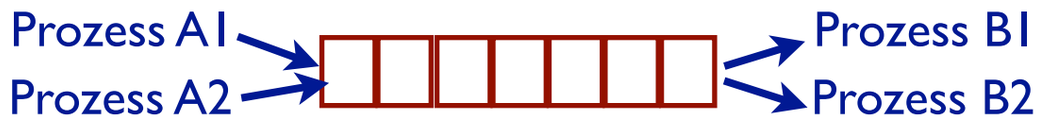
b) Nicht-Unteilbarkeit von Element anhängen/entfernen (abhängig von Implementierung und Anzahl der Erzeuger/Verbraucher)

⇒ ggf. kritischer Abschnitt

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount



Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
lock();
/* Element anhängen */
count++;
unlock();
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
lock();
/* 1. Element entfernen */
count--;
unlock();
```

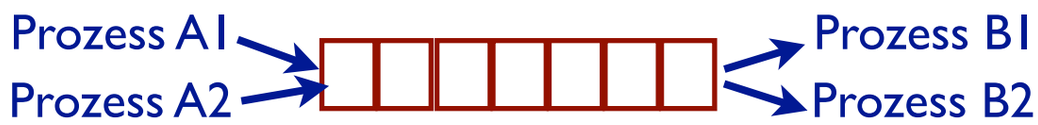
Probleme a)+b) durch Lock-Verfahren vermeidbar…

⇒ Reicht das?

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount



Füllen (A)

```
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
```

Leeren (B)

```
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
```

Einige potentielle Probleme:

c) Falls mehrere Erzeuger/Verbraucher:

⇒ nebenläufiger Zugriff auf count-Abfrage möglich

⇒ Schreiben/Lesen bei voll/leer möglich
(bisherige Lock-Lösung hilft nicht, da Abfrage nicht geschützt)

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
while (count == maxcount) {
  /* sleep(nicht-voll,… ) */
};
/* Element anhängen */
count++;
/* wakeup(nicht-leer,…) */
```

Leeren (B)

```
while (count == 0) {
  /* sleep(nicht-leer,…) */
};
/* 1. Element entfernen */
count--;
/* wakeup(nicht-voll,… ) */
```

Einige potentielle Probleme:

d) Bei blockierendem Warten:

⇒ Lost-Wakeup, bei maxcount=1 sogar Verklemmung möglich

Beispiel: „Realisierung“ einer Pipe (vereinfacht)

⇒ Modellierung einer Warteschlange

- Füllstandszähler count
- maximale Länge maxcount

Füllen (A)

```
lock();
while (count == maxcount) {
  /* warten */
};
/* Element anhängen */
count++;
unlock();
```

Leeren (B)

```
lock();
while (count == 0) {
  /* warten */
};
/* 1. Element entfernen */
count--;
unlock();
```

Einige potentielle Probleme:

e) Bei naivem Schutz vor a)–d) durch Lock/Unlock:
⇒ Verklemmung bei voller/leerer Warteschlange

Kleine Aufgabe

Anna und Hugo haben ein gemeinsames Konto und können mit den folgenden Methoden Ein- und Auszahlungsvorgänge anstoßen. Kann es dabei Nebenläufigkeitsprobleme geben?

```
int konto;

void einzahlen(int i)  {
   lock();
   konto = konto + i;
   unlock();
}
```

```
bool auszahlen(int i)  {
   if (konto >= i) {
     lock();
     konto = konto - i;
     unlock();
     return true;
   };
   return false;

}
```

Fragen – Teil 3

- Welche Nebenläufigkeitseigenschaften bzw. -probleme werden durch das folgende „klassische“ Szenario ausgedrückt:

  – Erzeuger/Verbraucher (Producer/Consumer)?

- Was versteht man unter *einseitiger* Synchronisation?
  Gib ein Anwendungsbeispiel an.

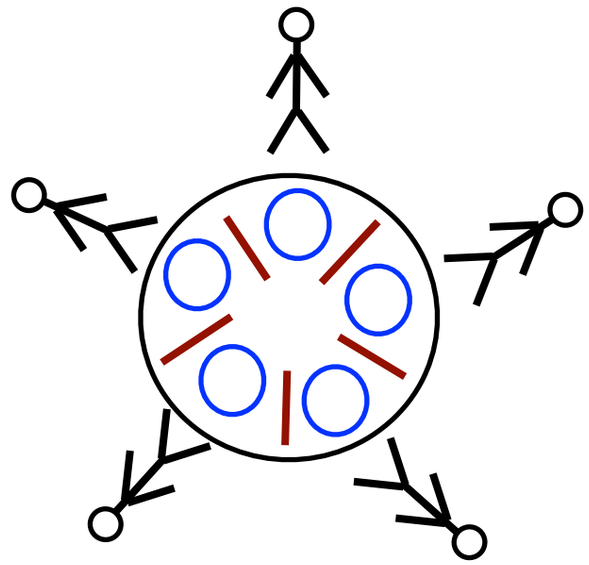
Teil 4:
Speisende Philosophen

14 Nebenläufigkeit, Ute Bormann

4) Die „speisenden“ Philosophen (Dining Philosophers)

- Beliebtes Problem aus der Literatur
- Mehrere Synchronisationaspekte enthalten
  - 5 Philosophen denken und essen Reis
  - aber nur 5 Stäbchen vorhanden

    ⇒ wird jeder satt?



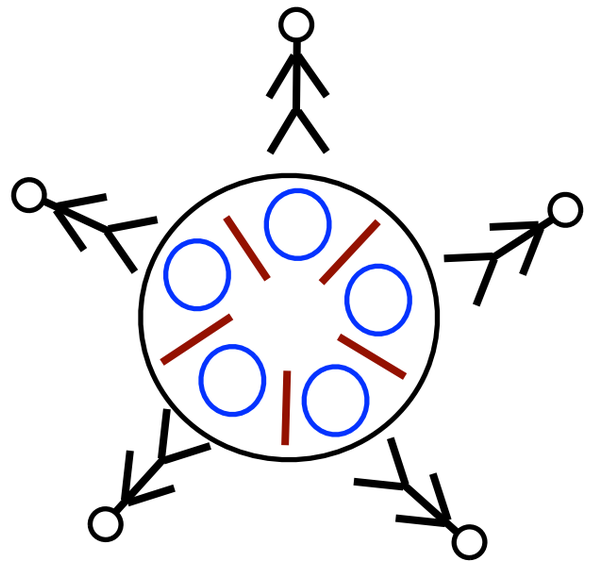
- Stäbchen sind kritische Abschnitte (nur 1 Besitzer pro Zeitpunkt)

  ⇒ mehrseitige Synchronisation

4) Die „speisenden“ Philosophen (Dining Philosophers)

- Beliebtes Problem aus der Literatur
- Mehrere Synchronisationaspekte enthalten
  - 5 Philosophen denken und essen Reis
  - aber nur 5 Stäbchen vorhanden

    ⇒ wird jeder satt?



- Stäbchen sind kritische Abschnitte (nur 1 Besitzer pro Zeitpunkt)

  ⇒ mehrseitige Synchronisation

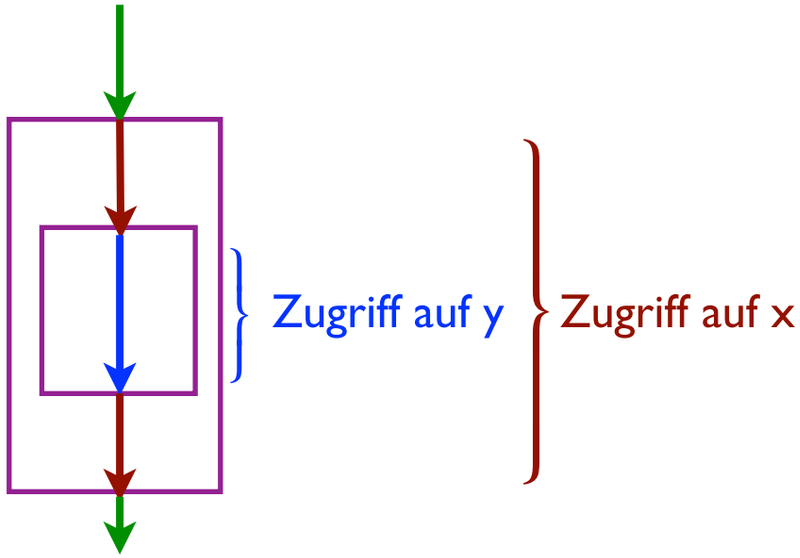
- Sequentielles Greifen von zwei Stäbchen: geschachtelter kritischer Abschnitt

  ⇒ jeder greift erstmal sein „linkes“ Stäbchen

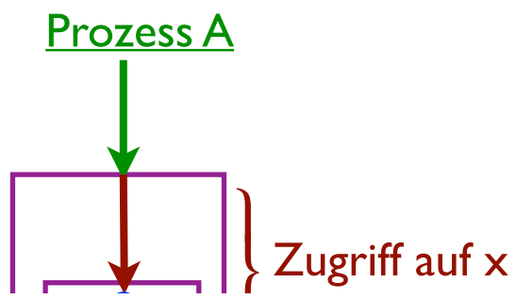
  ⇒ Gefahr von Verklemmungen (Deadlocks)

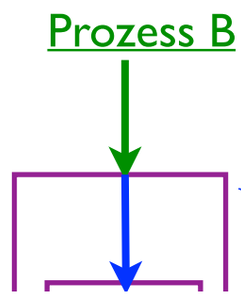
Geschachtelte kritische Abschnitte

- Nach Betreten eines kritischen Abschnitts zum Schutz von Betriebsmittel x wird Betriebsmittel y benötigt



⇒ Birgt Gefahr von Verklemmungen, wenn Prozesse unterschiedlich schachteln





Verklemmungen (Deadlocks)

- Situation: Zwei (oder mehr) Prozesse warten auf „Betriebsmittel“, die nur der/die andere(n) Wartende(n) freigeben kann/können
- Entsteht unter folgender Kombination von Randbedingungen

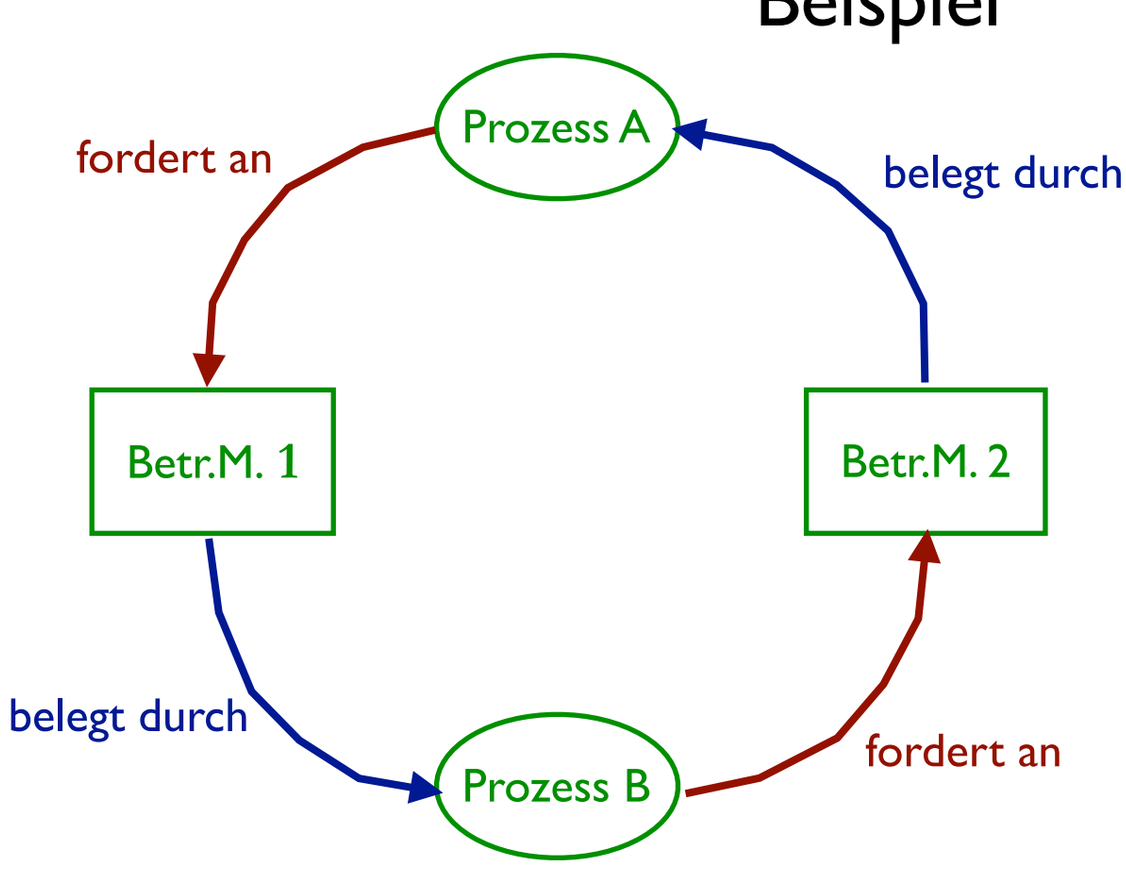
1. Nur bei Betriebsmitteln, die ein Prozess exklusiv für sich belegt hat
2. Jeder beteiligte Prozess hat bereits solche Belegungen und wartet auf weitere
3. Belegte Betriebsmittel bleiben belegt (keine freiwillige Abgabe, kein Zwangsentzug)

⇒ geschachtelte kritische Abschnitte

4. Es entsteht ein Zyklus von „Fordert-An/Belegt-Durch“-Relationen

⇒ unterschiedliche Schachtelung

Beispiel



Behandlung von Verklemmungen

a) Ignorieren

- Wenn Problem selten auftritt und nicht sicherheitskritisch ist, Lösung u.U. unangemessen „teuer“

  ⇒ stattdessen: Prozesse „hängen sich auf“ und werden vom Nutzer terminiert

⇒ Philosophen merken es selbst und reagieren

Behandlung von Verklemmungen

a) Ignorieren

...

b) Entdecken und Beheben

- z.B. Betriebsmittelgraph periodisch auf Zyklen untersuchen

  ⇒ einen Prozess zwangsentfernen ⇒ geht's jetzt?

⇒ „Diener“ der Philosophen merkt es und reagiert

Behandlung von Verklemmungen

a) Ignorieren

...

b) Entdecken und Beheben

...

c) Verhindern

- Eine der Randbedingungen der Entstehung verhindern. Welche?

1. Exklusiven Zugriff mehrerer Prozesse vermeiden? ⇒ Spezialfall Spooling

Exkurs: „Reservierung“ von Geräten vs. „Spooling“

- Nebenläufiger Zugriff mehrerer Prozesse auf ein Gerät nicht immer wünschenswert
- Beispiel: Drucker
- Jedoch: Exklusives Reservieren kann Verklemmungen verursachen

Alternative:

- Spezieller Prozess verwaltet Zugriff auf Gerät
- Beispiel: Drucker-Dämon
- Prozesse legen Aufträge als Dateien in speziellem Verzeichnis ab

  ⇒ Drucker-Dämon kümmert sich drum

Behandlung von Verklemmungen

a) Ignorieren

...

b) Entdecken und Beheben

...

c) Verhindern

- Eine der Randbedingungen der Entstehung verhindern. Welche?

1. Exklusiven Zugriff mehrerer Prozesse vermeiden? ⇒ Spezialfall Spooling

⇒ Philosophen werden von Diener „gefüttert“

Behandlung von Verklemmungen

a) Ignorieren

...

b) Entdecken und Beheben

...

c) Verhindern

- Eine der Randbedingungen der Entstehung verhindern. Welche?

1. Exklusiven Zugriff mehrerer Prozesse vermeiden? ⇒ Spezialfall Spooling
2. Alles auf einmal anfordern? ⇒ Unteilbarkeit modellieren, oft Verschwendung

⇒ Philosophen nehmen beide Stäbchen gleichzeitig

Behandlung von Verklemmungen

a) Ignorieren

...

b) Entdecken und Beheben

...

c) Verhindern

- Eine der Randbedingungen der Entstehung verhindern. Welche?

1. Exklusiven Zugriff mehrerer Prozesse vermeiden? ⇒ Spezialfall Spooling

2. Alles auf einmal anfordern? ⇒ Unteilbarkeit modellieren, oft Verschwendung

3. Freiwillige Abgabe? ⇒ ggf. alten Zustand wiederherstellen
   vs. Zwangsentzug? ⇒ i.d.R. Chaos

⇒ Philosophen: Erstes Stäbchen wieder hinlegen
vs. aus der Hand reißen

Behandlung von Verklemmungen

a) Ignorieren

...

b) Entdecken und Beheben

...

c) Verhindern

- Eine der Randbedingungen der Entstehung verhindern. Welche?

1. Exklusiven Zugriff mehrerer Prozesse vermeiden? ⇒ Spezialfall Spooling
2. Alles auf einmal anfordern? ⇒ Unteilbarkeit modellieren, oft Verschwendung
3. Freiwillige Abgabe? ⇒ ggf. alten Zustand wiederherstellen
   vs. Zwangsentzug? ⇒ i.d.R. Chaos
4. Zyklus verhindern? (z.B. Betriebsmittel nach aufsteigender „Nummer“ anfordern)

   ⇒ oft wenig praktikabel bzw. ziemlich einschränkend

⇒ Ein Philosoph ist „Linkshänder“

d) Vermeiden

- Beispiel: Bankiers-Algorithmus
- Annahme: Maximale Anforderungen von Betriebsmitteln im voraus bekannt
- Belegung wird nur gewährt, wenn System dadurch nicht in unsicheren Zustand kommt (= Zustand, der zu Verklemmung führen kann)
- Modell eines Bankiers, der mit Kunden Kreditrahmen aushandelt, der nach und nach eingelöst werden kann

⇒ Allerdings keine „Realzeit“-Anforderungen berücksichtigt

Beispiel: 10 gleichartige Betriebsmittel vorhanden

	Max. Anforderung	Aktuell belegt
Prozess A	5	2
Prozess B	3	1
Prozess C	9	5
	∑17 > 10	∑8

- Gegenwärtiger Zustand ist sicher: Es gibt zulässige Belegungsfolge, die alle Prozesse sicher zum Ende führt:
  - Erst B (braucht noch max. 2 Betriebsmittel)
  - Dann A (nach Ende von B 3 Betriebsmittel verfügbar)
  - Dann C (nach Ende von A 5 Betriebsmittel verfügbar)

Beispiel: 10 gleichartige Betriebsmittel vorhanden

	Max. Anforderung	Aktuell belegt
Prozess A	5	2
Prozess B	3	~~1~~ 2
Prozess C	9	5
	∑17 > 10	∑~~8~~ 9

- Gegenwärtiger Zustand ist sicher: Es gibt zulässige Belegungsfolge, die alle Prozesse sicher zum Ende führt:
  - Erst B (braucht noch max. 2 Betriebsmittel)
  - Dann A (nach Ende von B 3 Betriebsmittel verfügbar)
  - Dann C (nach Ende von A 5 Betriebsmittel verfügbar)

⇒ Anforderung von B wird genehmigt

Beispiel: 10 gleichartige Betriebsmittel vorhanden

	Max. Anforderung	Aktuell belegt
Prozess A	5	2
Prozess B	3	1
Prozess C	9	~~5~~ 6
	∑17 > 10	∑~~8~~ 9

- Gegenwärtiger Zustand ist sicher: Es gibt zulässige Belegungsfolge, die alle Prozesse sicher zum Ende führt:
  - Erst B (braucht noch max. 2 Betriebsmittel)
  - Dann A (nach Ende von B 3 Betriebsmittel verfügbar)
  - Dann C (nach Ende von A 5 Betriebsmittel verfügbar)

⇒ Anforderung von C wird abgelehnt, da Übergang in unsicheren Zustand:

- nur noch 1 Betriebsmittel übrig, aber A,B,C brauchen u.U. noch mehr als 1 weiteres Betriebmittel

⇒ Verklemmung möglich

d) Vermeiden

- Beispiel: Bankiers-Algorithmus
- Annahme: Maximale Anforderungen von Betriebsmitteln im voraus bekannt
- Belegung wird nur gewährt, wenn System dadurch nicht in unsicheren Zustand kommt (= Zustand, der zu Verklemmung führen kann)
- Modell eines Bankiers, der mit Kunden Kreditrahmen aushandelt, der nach und nach eingelöst werden kann

  ⇒ Allerdings keine „Realzeit“-Anforderungen berücksichtigt

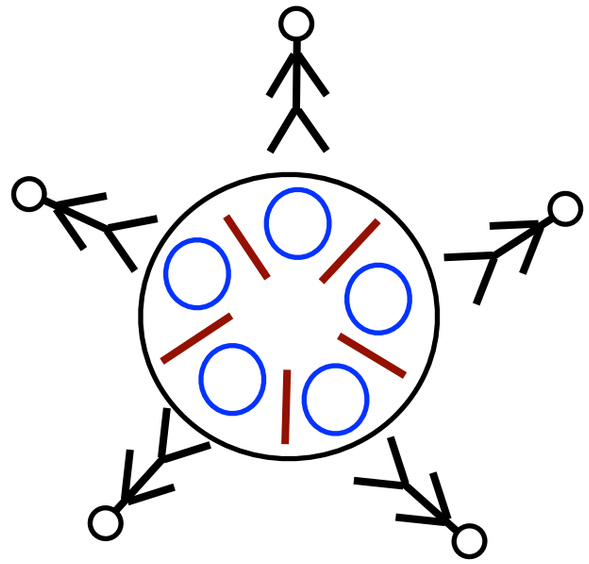
→ • Allerdings: Verfahren oft nicht realisierbar, da Bedürfnisse der Prozesse nicht im Vorfeld bekannt

⇒ 5. Stäbchen nicht an 5. Philosophen geben

4) Die „speisenden“ Philosophen (Dining Philosophers)

- Beliebtes Problem aus der Literatur
- Mehrere Synchronisationaspekte enthalten
  - 5 Philosophen denken und essen Reis
  - aber nur 5 Stäbchen vorhanden

    ⇒ wird jeder satt?



- Stäbchen sind kritische Abschnitte (nur 1 Besitzer pro Zeitpunkt)

  ⇒ mehrseitige Synchronisation

- Sequentielles Greifen von zwei Stäbchen: geschachtelter kritischer Abschnitt

  ⇒ jeder greift erstmal sein „linkes“ Stäbchen

  ⇒ Gefahr von Verklemmungen (Deadlocks)

- Mögliche Lösung („Vermeiden“)

  ⇒ z.B. nur 4 Philosophen dürfen gleichzeitig an den Tisch

  (ggf. Modellierung von 4 Stühlen)

- Bisherige Probleme durch Mehrprozessbetrieb eines „klassischen“ Timesharing-Systems gegeben

  ⇒ mehrere Prozesse teilen sich eine CPU

  ⇒ Konkurrenz und Kooperation

- Ähnliche Probleme bei

  a) Mehrprozessorsystem?

  b) Verteiltem System (Rechnernetz)?

  c) Multi-Threading? (⇒ später)

- Bisherige Probleme durch Mehrprozessbetrieb eines „klassischen“ Timesharing-Systems gegeben

  ⇒ mehrere Prozesse teilen sich eine CPU

  ⇒ Konkurrenz und Kooperation

- Ähnliche Probleme bei

  a) Mehrprozessorsystem?

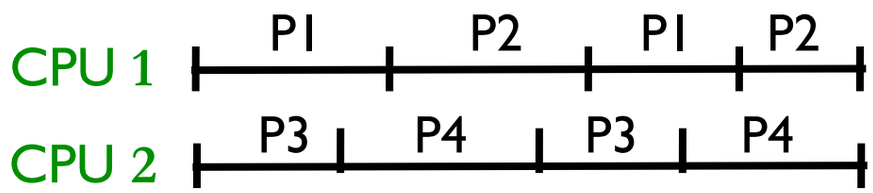
  b) Verteiltem System (Rechnernetz)?

  c) Multi-Threading? (⇒ später)

Ad a) Mehrprozessorsystem

- Mehrere CPUs

  ⇒ mehrere Prozesse parallel ausführbar (spezielle Form der Nebenläufigkeit)

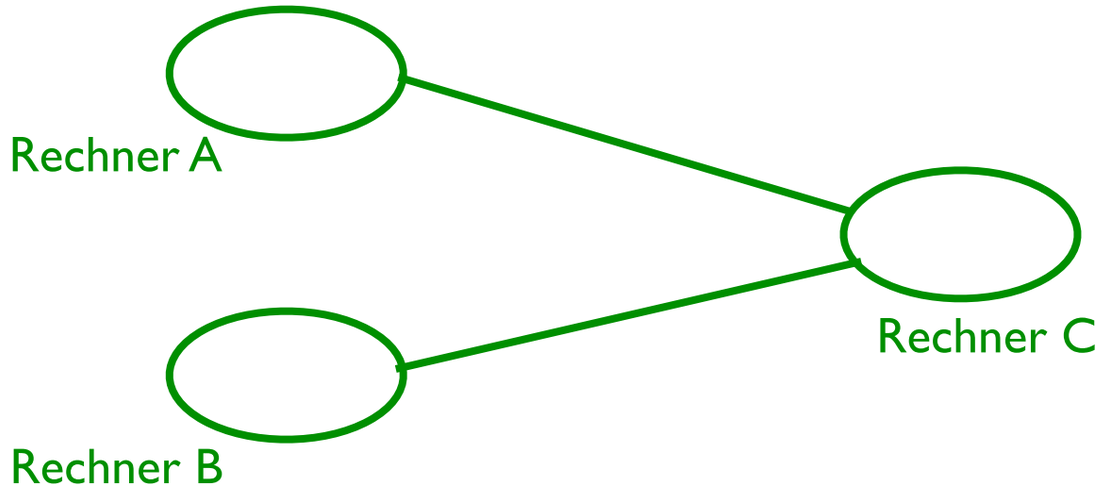


- Durch verstärkte nebenläufige Aktivitäten u.U. höheres Konfliktpotential

  ⇒ weiterhin Konkurrenz und Kooperation

- Erforderliche Synchronisationsmaßnahmen u.U. komplexer, da Parallelität von Ereignissen berücksichtigt werden muss (⇒ später)

Ad b) Verteilte Systeme

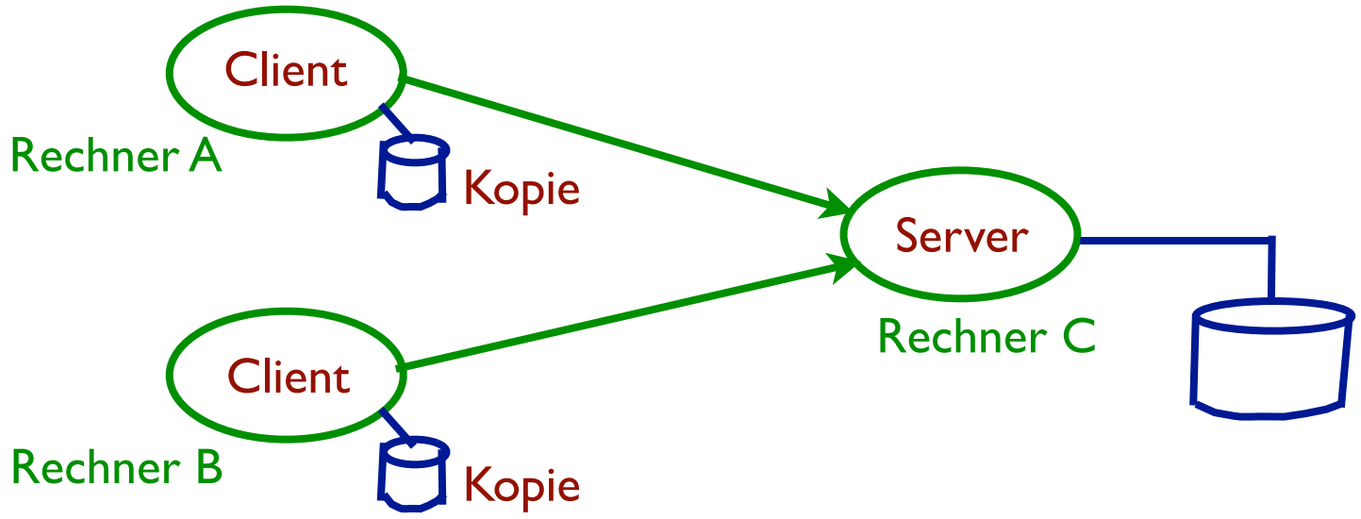


- Systeme oft isolierter ⇒ weniger Konfliktpotential
- Interprozesskommunikation durch Nachrichtenaustausch:
  - dauert Zeit
  - u.U. unzuverlässiges Kommunikationsmedium
  - Absprachen zwischen Kommunikationspartnern erforderlich

    ⇒ Partner in unterschiedlichen Zuständen

    ⇒ (verstärkte) Synchronisationsprobleme

Ad b) Verteilte Systeme



- Systeme oft isolierter ⇒ weniger Konfliktpotential
- Interprozesskommunikation durch Nachrichtenaustausch:
  - dauert Zeit
  - u.U. unzuverlässiges Kommunikationsmedium
  - Absprachen zwischen Kommunikationspartnern erforderlich

    ⇒ Partner in unterschiedlichen Zuständen

    ⇒ (verstärkte) Synchronisationsprobleme
- u.U. weiterhin Zugriff auf gemeinsame Betriebsmittel
- Beispiel: Fernzugriff auf Dateien

  ⇒ zusätzliche Probleme durch lokale Kopien

Fragen – Teil 4

- Welche Nebenläufigkeitseigenschaften bzw. -probleme werden durch das folgende „klassische“ Szenario ausgedrückt:

  – Speisende Philosophen *(Dining Philosophers)*?

- Auf welche verschiedene Arten kann man *Verklemmungen* angehen? Wie arbeitet der Bankiersalgorithmus ?

Zusammenfassung

- Nichtdeterministische Abläufe
  - zwischen Prozessen im User-Mode (Shared Memory, gemeinsame Dateien,...)
  - im Betriebssystemkern (Prozesse vs. Interrupt-Handler)
- Nebenläufigkeit: Parallelität oder Quasi-Parallelität
- Gegenseitige Beeinflussung möglich/üblich durch:
  - Konkurrenz von Prozessen (ggf. Interrupt-Handlern)
  - Kooperation von Prozessen (ggf. Interrupt-Handlern)
- Mehrseitige vs. einseitige Synchronisation
- Verklemmungen (Deadlocks)
- Beispielszenarien: Kritischer Abschnitt, Leser/Schreiber, Erzeuger/Verbraucher, speisende Philosophen

Nebenläufigkeit – Fragen

1. Skizziere kurz einige Probleme des *nebenläufigen* Zugriffs auf Betriebsmittel.
2. Grenze die Begriffe *Nebenläufigkeit*, *Quasi-Parallelität* und *Parallelität* voneinander ab.
3. Was verstehen wir unter *Nichtdeterminismus*?
4. Welche Nebenläufigkeitseigenschaften bzw. -probleme werden durch die drei folgenden „klassischen“ Szenarien ausgedrückt:
   a) Kritischer Abschnitt *(Critical Section)*,
   b) Leser/Schreiber *(Reader/Writer)*,
   c) Erzeuger/Verbraucher *(Producer/Consumer)*,
   d) Speisende Philosophen *(Dining Philosophers)*?
5. Was versteht man unter *einseitiger* bzw. *mehrseitiger* Synchronisation? Gib jeweils ein Anwendungsbeispiel an.
6. Auf welche verschiedene Arten kann man *Verklemmungen* angehen? Wie arbeitet der Bankiersalgorithmus ?