Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Vorlesung 13: Semaphore und Sicherheit

Literatur:
Arpaci-Dusseau [1]
Kap. 27–32

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Passives Warten

- Idee: Prozesse geben die CPU frei, während sie warten
  - Bei Synchronisation blockiert ein Prozess sich selbst, während er auf ein Ereignis wartet
    - Der Prozess wird in eine Warteschlange eingereiht
  - Wenn das Ereignis eintritt, dann wird ***einer der Prozesse***, die darauf warten, deblockiert (es kann mehr als einen geben)
- Die Wartephase eines Prozesses wird als Blockierungsphase realisiert (wie ein I/O burst)
  - Der Prozess-Scheduler wird aktualisiert
  - Ein anderer Prozess im Zustand READY wird auf RUNNING gesetzt (*dispatching*)
  - *Was geschieht, wenn zu dem Zeitpunkt kein Prozess READY ist?*
- Mit dem Beginn der Blockierungsphase eines Prozesses endet sein CPU-Burst

Passives Warten: Semaphore

- Eine ***Semaphore*** ist definiert als "nicht negative Ganzzahl" mit **zwei atomaren Operationen:** [2]

  P (von niederl. "prolaag" = "dekrementieren"; auch *down* oder ***wait***)
  - wenn die Semaphore den Wert 0 hat, wird der Prozess, der **P** aufgerufen hat, **blockiert**
  - sonst wird der Wert der Semaphore dekrementiert

  V (von niederl. "verhoog" = "erhöhen"; auch *up* oder ***signal***)
  - ein Prozess, der auf die Semaphore wartet (als Folge eines vorherigen Aufrufs von P), wird deblockiert
  - sonst wird der Wert der Semaphore um 1 erhöht

- Semaphore sind eine **Betriebssystemabstraktion**, um Synchronisationssignale zwischen nebenläufigen Prozessen auszutauschen

Semaphore: Beispielimplementation

Universität Bamberg

```
/* C++ implementation taken from the teaching OS OO-StuBS */
class Semaphore : public WaitingRoom {
  int counter;
public:
  Semaphore(int c) : counter(c) {}
  void wait() {
    if (counter == 0) {
      Customer *life = (Customer*)scheduler.active();
      enqueue(life);
      scheduler.block(life, this);
    }
    else
      counter--;
  }
  void signal() {
    Customer *customer = (Customer*)dequeue();
    if (customer)
      scheduler.wakeup(customer);
    else
      counter++;
  }
};
```

Ein "WaitingRoom" ist eine Liste von Prozessen mit den Zugriffsmethoden `enqueue` und `dequeue`

Der *Scheduler* muss drei Operationen bereitstellen:

- `active` **gibt den aktuellen PCB zurück**
- `block` **setzt einen Prozess in BLOCKED**
- `wakeup` **setzt einen Prozess zurück auf die READY-Liste**

Verwendung von Semaphoren

“Gegenseitiger Ausschluss”: eine Semaphore, die auf den Wert 1 initialisiert ist, kann als Lockvariable verwendet werden

```
Semaphore lock; /* = 1: verwende Semaphore als Lockvariable */

/* Beispielcode */
void mythread (struct list *list, struct element *item) {
  wait (&lock);

  counter = counter + 1;

  signal (&lock);
}
```

- **der erste Process, der den krit. Abschn. betritt, decrementiert die Semaphore auf 0**
- **alle anderen blockieren**

- **wenn der kritische Abschnitt verlassen wird, wird entweder ein blockierter Prozess aufgeweckt oder der Wert der Semaphore wieder auf 1 inkrementiert**

…und dies ist nicht die einzige Anwendung von Semaphoren…

Semaphore: einfache Interaktionen

- "einseitige Synchronisation"

```
/* shared memory */
Semaphore elem;
struct list l;
struct element e;
```

```
void producer() {
  enqueue(&l, &e);
  signal(&elem);
}
```

```
void consumer() {
  struct element *x;
  wait(&elem);
  x = dequeue(&l);
}
```

```
/* Initialisierung */
elem = 0;
```

- "Ressourcenorientierte Synchronisation"

```
/* shared memory */
Semaphore resource;
```

```
/* Initialisierung */
resource = N; /* N > 1 */
```

Der Rest:
genau wie oben

Semaphore: komplexe Interaktionen

- Beispiel: das erste ***Leser-/Schreiber-Problem***

Auch in diesem Beispiel muss ein kritischer Abschnitt geschützt werden – genau wie beim gegenseitigen Ausschluss

Hier haben wir allerdings **zwei Klassen** nebenläufiger Prozesse:

- ***Schreiber:*** sie ändern Daten und benötigen daher eine Garantie für den gegenseitigen Ausschluss
- ***Leser:*** diese lesen nur Daten, daher dürfen mehrere Leser den kritischen Abschnitt gleichzeitig betreten

Semaphore: komplexe Interaktionen

- Beispiel: das erste ***Leser-/Schreiber-Problem***

```
/* shared memory */
Semaphore mutex;
Semaphore wrt;
int readcount;
```

```
/* Initialisierung */
mutex = 1;
wrt   = 1;
readcount = 0;
```

```
/* Schreiber */
wait(&wrt);

… write data …

signal(&wrt);
```

```
/* Leser */
wait(&mutex);
readcount++;
if (readcount == 1) {
   wait(&wrt);
}
signal(&mutex);

… read data …

wait(&mutex);
readcount--;
if (readcount == 0) {
  signal(&wrt);
}
signal(&mutex);
```

Semaphore: Diskussion

- Varianten und Erweiterungen von Semaphore
  - binäre Semaphore oder ***mutex***
  - nicht blockierendes **wait**()
  - timeout
  - Arrays von Zählern
- Fehlerquellen
  - Risiko von "***deadlocks***" → gleich und nächste Vorlesung
  - Schwierig, komplexere Synchronisationsmuster zu implementieren
  - Kooperierende Prozesse hängen voneinander ab
    - alle müssen genau das Protokoll befolgen
  - Verwendung von Semaphoren wird nicht erzwungen
- Unterstützung in Programmiersprachen

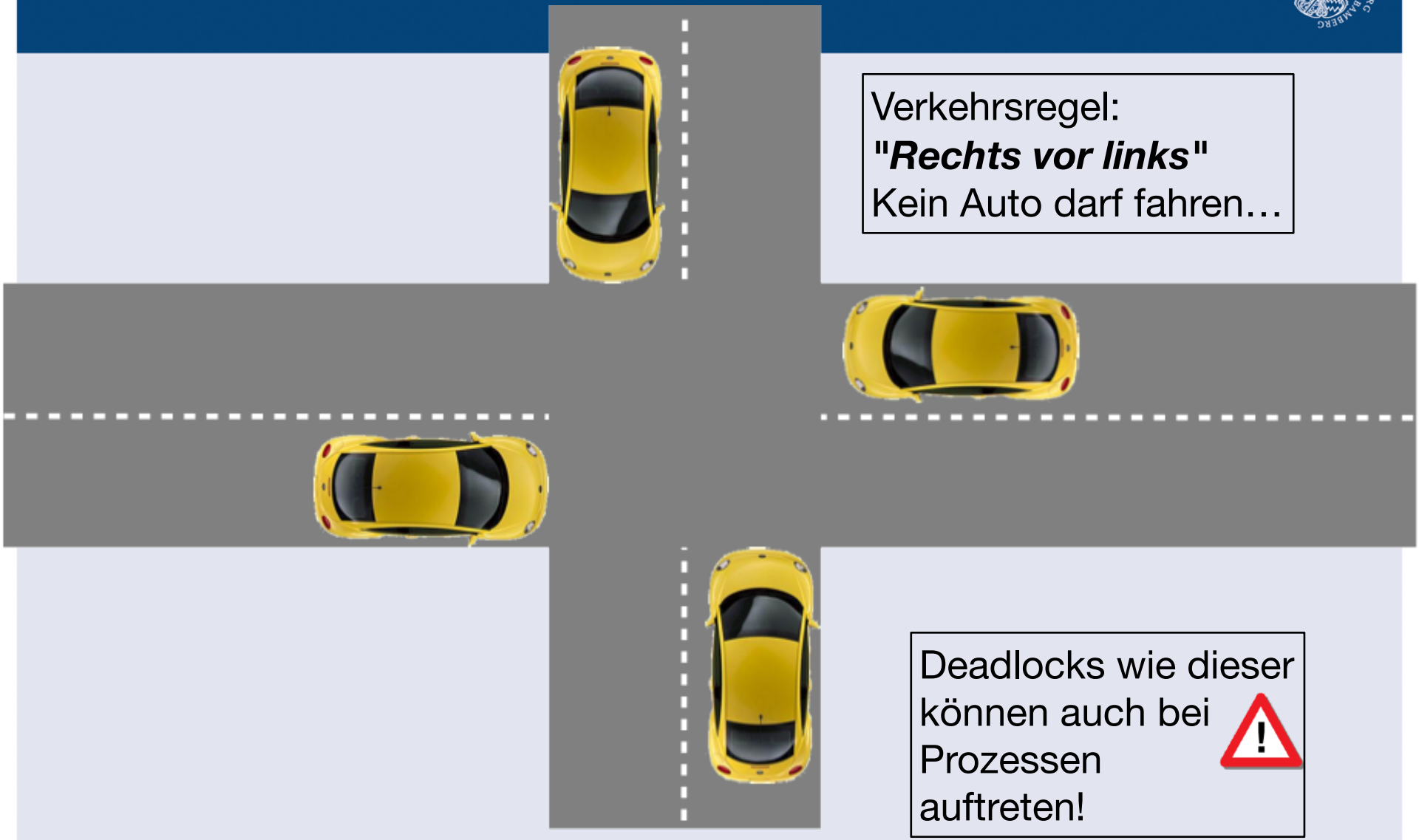
Prozesse, nochmal

- Prozesse laufen in einem Computer *nebenläufig*
- Um Prozesse zu koordinieren, verwenden wir *Synchronisationsprimitive*

- Grundidee: *passives Warten*
- Semaphore ermöglichen...
  - gegenseitigen Ausschluss
  - einseitige Synchronisation
  - ressourcenorientierte Synchronisation

- Wartemechanismen führen zu Deadlockproblemen

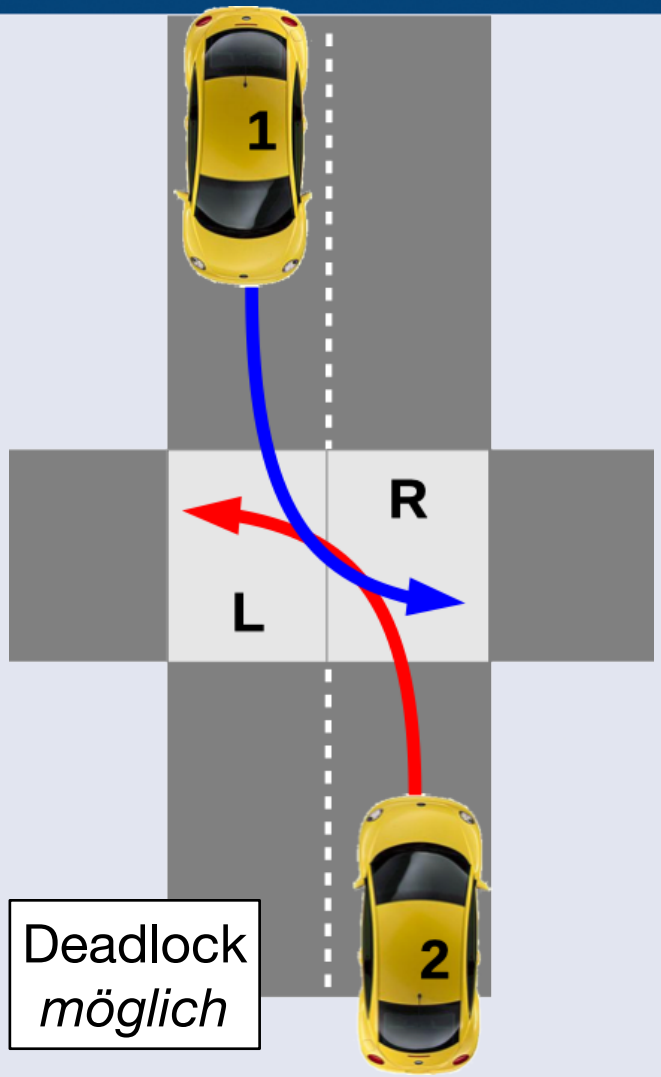
Deadlocks



Verkehrsregel:
"*Rechts vor links*"
Kein Auto darf fahren…

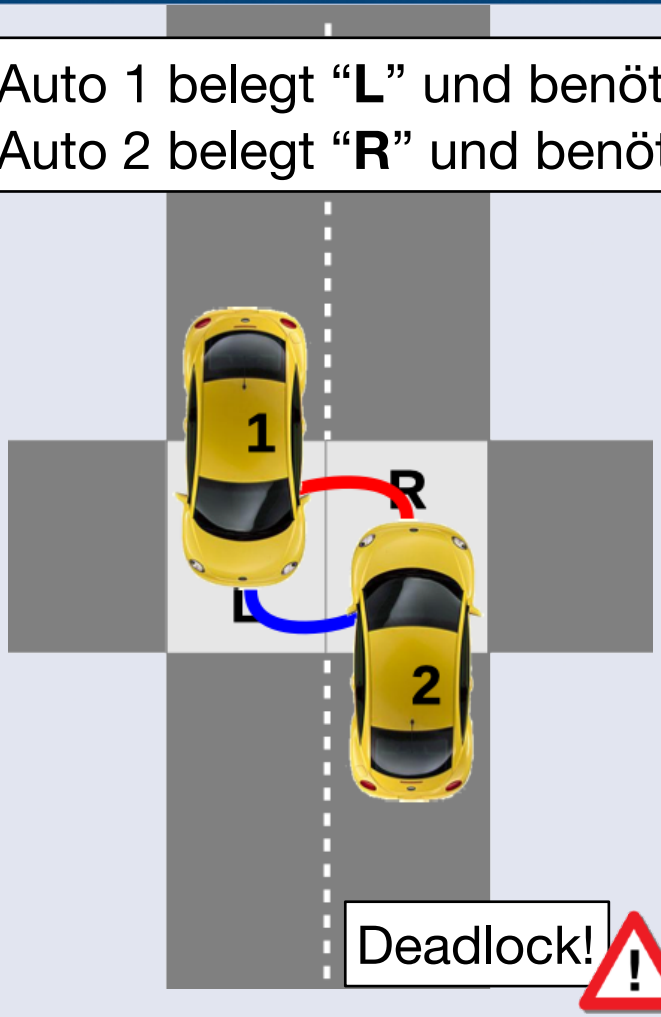
Deadlocks wie dieser können auch bei Prozessen auftreten!

Warum geschieht das?



Deadlock *möglich*

Auto 1 belegt “**L**” und benötigt “**R**”
Auto 2 belegt “**R**” und benötigt “**L**”



Deadlock!

Prozesse mit Deadlocks

- Der Begriff "*deadlock*" bedeutet (in der Informatik):

> "[…] eine Situation, in der zwei oder mehr Prozesse nicht in der Lage sind, ihre Ausführung fortzusetzen, da jeder darauf wartet, dass einer der anderen etwas ausführt."
>
> William Stallings.
> Operating Systems: Internals and Design Principles.

Prozesse mit Deadlocks

- Alternative 1: ***Deadlock***
  - Passives Warten
  - Prozesszustand ist BLOCKED
- Alternative 2: ***Livelock***
  - *Aktives* Warten (*busy waiting* oder “lazy” busy waiting)
  - Beliebiger Prozesszustand (incl. RUNNING), aber keiner der beteiligten Prozesse kann fortfahren
- Deadlocks sind das “kleinere Übel”
  - Dieser Zustand ist eindeutig feststellbar
    → Basis zum *”Auflösen”* von Deadlocks existiert
- Aktives Warten erzeugt eine sehr hohe *Systemlast*

Bedingungen für Deadlocks

Alle der folgenden drei Bedingungen müssen erfüllt sein, damit ein Deadlock eintreten kann ("***notwendige Bedigungen***"):

1. Exklusive Allokation von Ressourcen ("***gegenseitiger Ausschluss***")
   - Nur ein Prozess darf gleichzeitig eine Ressource nutzen. Kein Prozess darf eine Ressource nutzen, die gerade einem anderen Prozess zugeteilt ist
2. Allokation zusätzlicher Ressourcen ("***halten und warten***")
   - Ein Prozess darf allokierte Ressourcen weiter halten, während er auf die Zuteilung weiterer Ressourcen wartet
3. Kein Entzug von Ressourcen ("***keine Verdrängung***")
   - Das Betriebssystem kann einem Prozess keine Ressourcen gegen dessen "Willen" entziehen

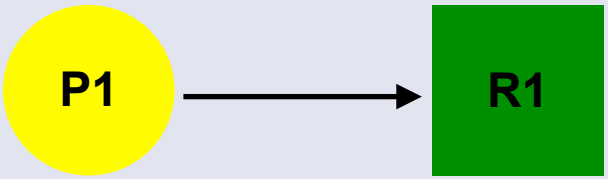
Bedingungen für Deadlocks

Alle der folgenden drei Bedingungen müssen erfüllt sein, damit ein Deadlock eintreten kann ("***notwendige Bedigungen***"):

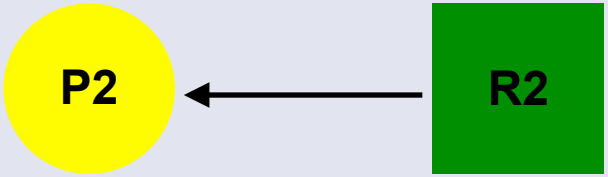
1. Exklusive Allokation von Ressourcen ("***gegenseitiger Ausschluss***")
   - Nur ein Prozess darf gleichzeitig eine Ressource nutzen. Kein Prozess darf eine Ressource nutzen, die gerade einem anderen Prozess zugeteilt ist
2. Allokation zusätzlicher Ressourcen ("***halten und warten***")
   - Ein Prozess darf allokierte Ressourcen weiter halten, während er auf die Zuteilung weiterer Ressourcen wartet
3. Kein Entzug von Ressourcen ("***keine Verdrängung***")
   - Das Betriebssystem kann einem Prozess keine Ressourcen gegen dessen "Willen" entziehen
4. Nur wenn ***eine zusätzliche Bedingung*** zur Laufzeit ***auftritt***, haben wir wirklich einen Deadlock:
   - ***”zirkuläres Warten”****: Eine geschlossene Kette von Prozessen existiert, so dass jeder Prozess mindestens eine Ressource hält, die vom nächsten Prozess in der Kette benötigt wird*

Resourcenallokations-Graphen

- ... werden verwendet, um Deadlock-Situationen zu visualisieren und auch automatisch zu erkennen
  - Sie beschreiben den *aktuellen Systemzustand*
  - Die *Knoten* sind *Prozess* und *Ressourcen*
  - Die *Kanten* stellen eine *Allokation* oder eine *Anfrage* dar



Ressource R1 wird von
Prozess P1 *angefordert*



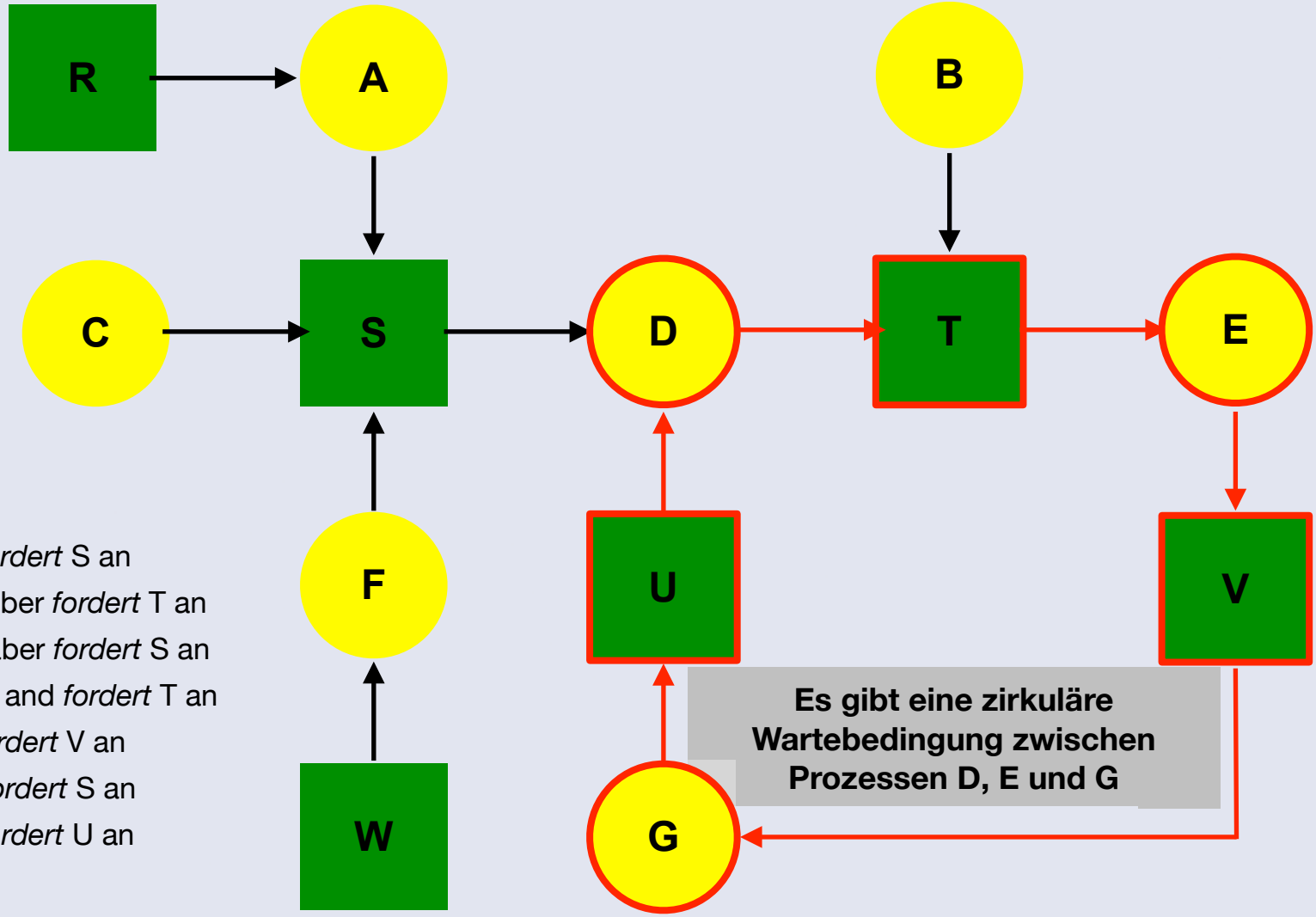
Prozess P2 *belegt*
Ressource R2

Resourcenallokations-Graphen

- Zu betrachtende Fragen:
  - Gibt es einen Zustand des zirkulären Wartens?
  - Welche Prozesse und Ressourcen sind daran beteiligt?
- Beispiel: 7 *Prozesse* A – G und 6 *Ressourcen* R – W
- Aktueller Zustand:
  - A *belegt* R und *fordert* S an
  - B *belegt* nichts, aber *fordert* T an
  - C *belegt* nichts, aber *fordert* S an
  - D *belegt* U und S und *fordert* T an
  - E *belegt* T und *fordert* V an
  - F *belegt* W und *fordert* S an
  - G *belegt* V und *fordert* U an

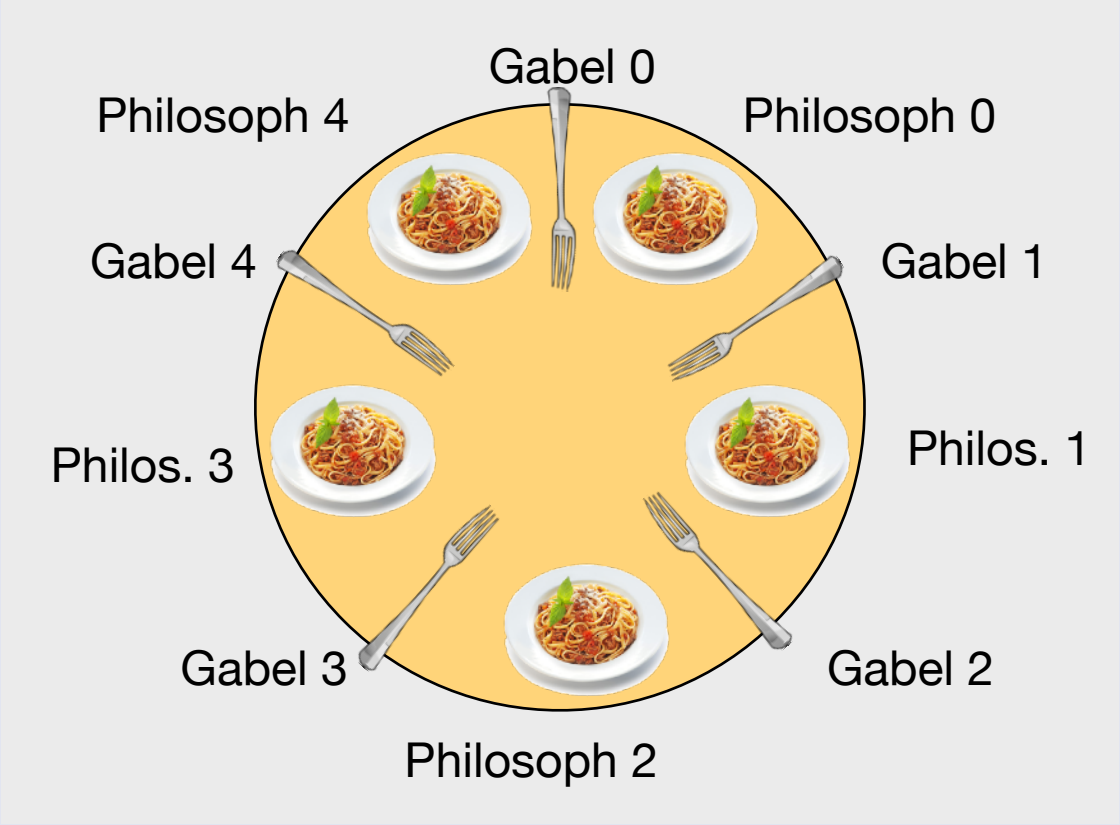
Deadlock im Resourcenallokations-Graphen

Universität Bamberg



A *belegt* R und *fordert* S an
B *belegt* nichts, aber *fordert* T an
C *belegt* nichts, aber *fordert* S an
D *belegt* U und S and *fordert* T an
E *belegt* T und *fordert* V an
F *belegt* W und *fordert* S an
G *belegt* V und *fordert* U an

Klassischer Deadlock: Speisende Philosophen



Prozess → Philosoph
Ressource → Gabel (unteilbar)

Fünf Philosoph(inn)en verbringen ihre Zeit entweder mit *denken* oder *essen.* Und sie lieben Spaghetti! [1]

Zum Essen sitzen die Philosophen an einem runden Tisch. Denken macht hungrig – jeder Philosoph muss essen!

Um Spaghetti zu essen benötigt ein Philosoph *beide Gabeln* neben ihrem oder seinem Teller!

Deadlock bei Philosophen?

Hier sind die *ersten drei notwendigen Bedingungen* erfüllt:

- *”**gegenseitiger Ausschluss**”*
  - Philosophen benötigen beide Gabeln zum Essen von Spaghetti
- *”**halten und warten**”*
  - Die Philosophen sind vor dem Essen so mit Denken beschäftigt, dass sie weder beide Gabeln gleichzeitig nehmen können noch auf die Idee kommen, eine einzelne Gabel wieder zurückzulegen.
- *”**keine Verdrängung**”*
  - Selbstverständlich ist es nicht schicklich, eine Gabel eines anderen Philosophen zu nehmen, während diese gerade benutzt wird.
- Aber führt dies immer zu einem Deadlock?

Speisende Philosophen: Version 1


```
/* alle Philosophen sind
   nebenläufig… */
void phil (int n) {
    while (1) {
        denken();
        aufnehmen(n);
        essen();
        weglegen(n);
    }
}

void denken () { ... }
void essen () { ... }
```

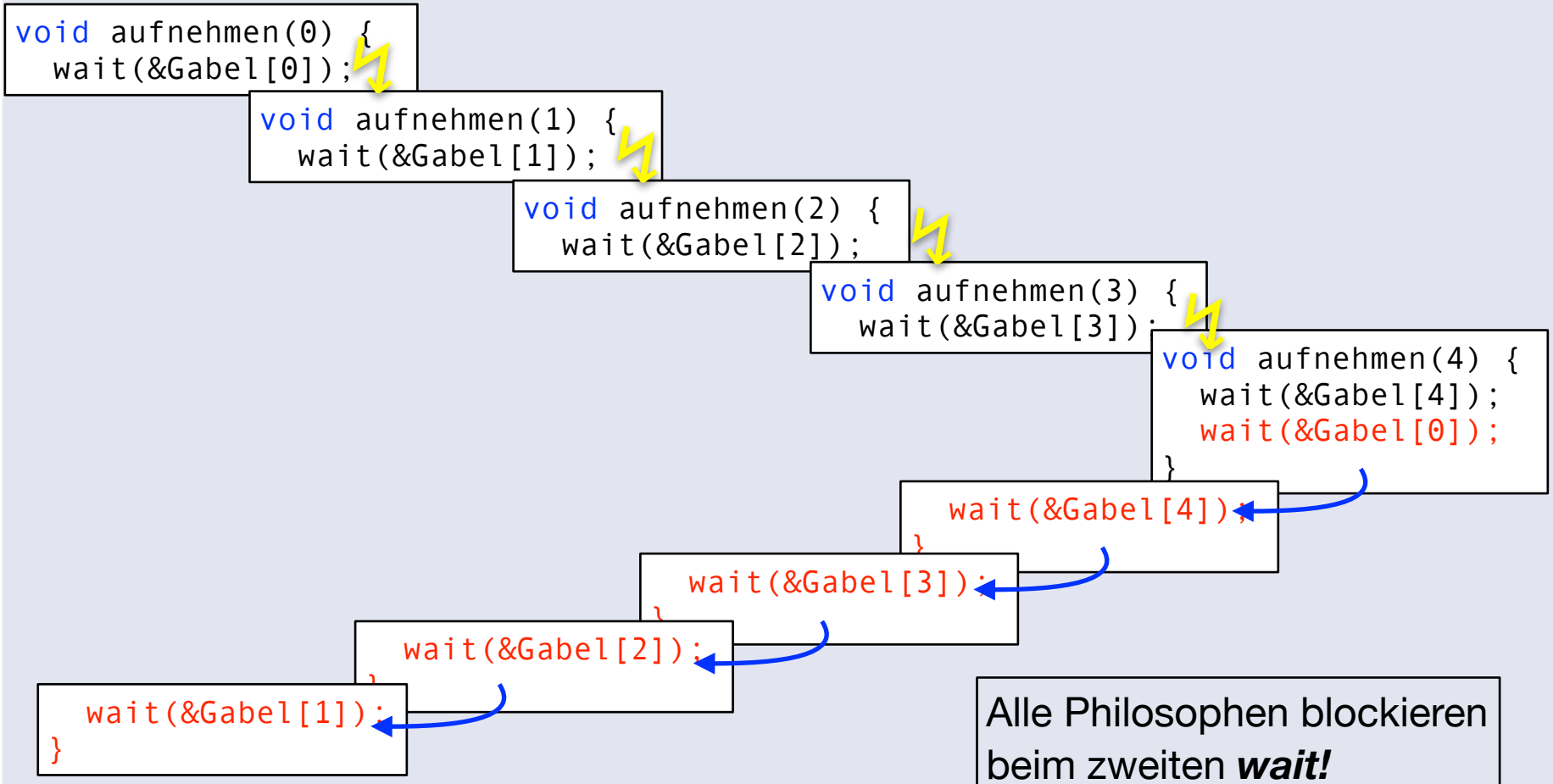
```
semaphore Gabel[NPHIL] = {
  {1, NULL}, ...
};

void aufnehmen (int n) {
    wait(&Gabel[n]);
    wait(&Gabel[(n+1)%NPHIL]);
}

void weglegen (int n) {
    signal(&Gabel[n]);
    signal(&Gabel[(n+1)%NPHIL]);
}
```

Die Verwendung einer Semaphore ***garantiert gegenseitigen Ausschluss*** bei Zugriff auf die Gabeln.
Entsprechend der Tradition nimmt jeder Philosoph zuerst die rechte und dann die linke Gabel.

…Deadlock-gefährdet…

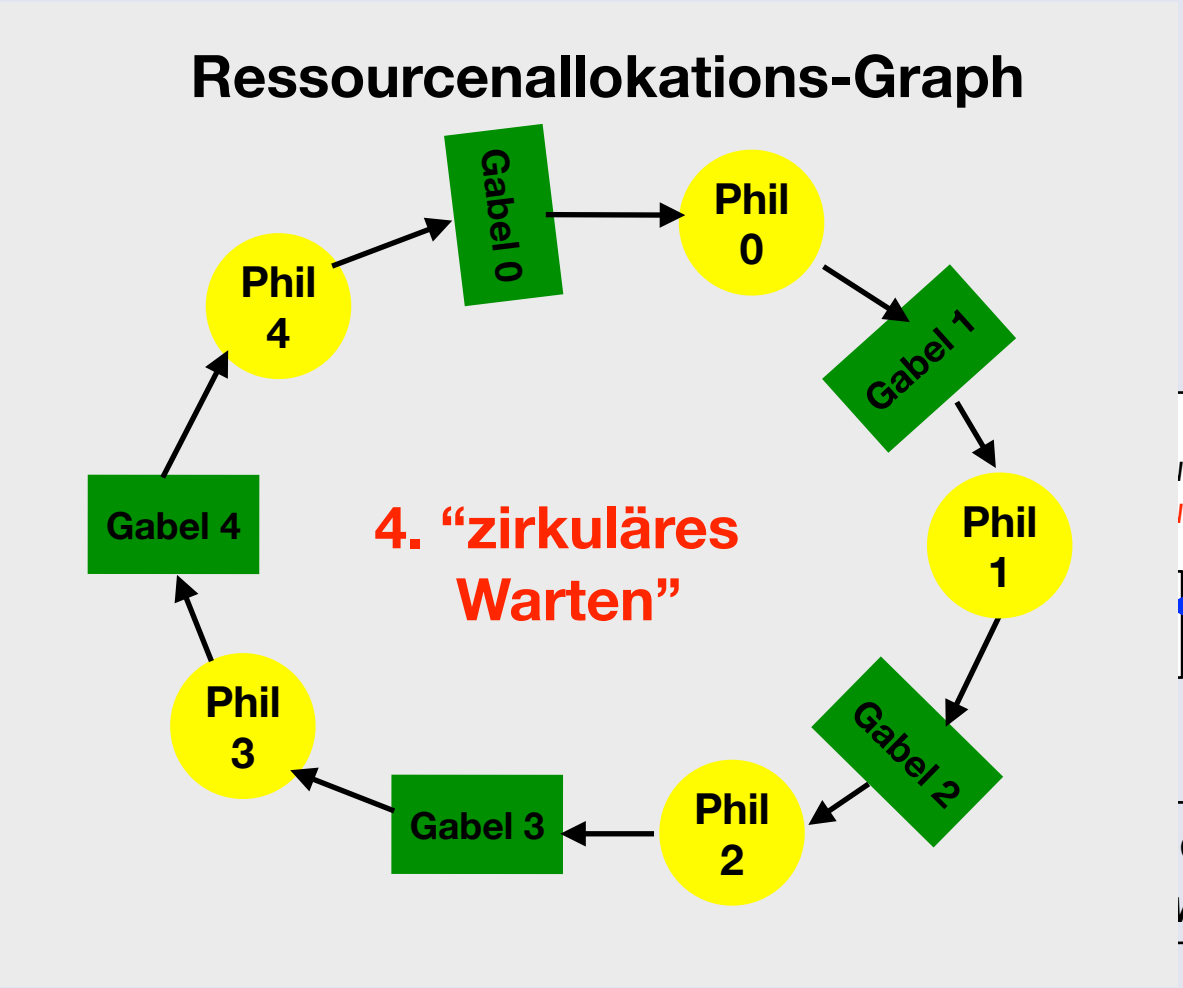


Alle Philosophen blockieren beim zweiten ***wait!***

Deadlock!

…Deadlock-gefährdet…


```
void aufnehm
  wait(&Gabe
```




```
d aufnehmen(4) {
ait(&Gabel[4]);
ait(&Gabel[0]);
```

```
  wait(&Gab
}
```

en blockieren
wait.

Deadlock!

Speisende Philosophen: Version 2


```
semaphore mutex = {1, NULL};

void aufnehmen (int n) {
    wait(&mutex);
    wait(&Gabel[n]);
    wait(&Gabel[(n+1)%NPHIL]);
    signal(&mutex);
}
```

Das Problem in Version 1 entstand durch eine Prozessumschaltung zwischen dem 1. und 2. ***wait*** – ein kritischer Abschnitt.

Version 2 schützt diesen kritischen Abschnitt durch gegenseitigen Ausschluss.

- Ist diese Lösung Deadlock-frei?
- Ist dies eine “gute” Lösung?

Speisende Philosophen: Version 2


```
semaphore mutex = {1, NULL};

void aufnehmen (int n) {
    wait(&mutex);
    wait(&Gabel[n]);
    wait(&Gabel[(n+1)%NPHIL]);
    signal(&mutex);
}
```

Das Problem in Version 1 entstand durch eine Prozessumschaltung zwischen dem 1. und 2. ***wait*** – ein kritischer Abschnitt.

Version 2 schützt diesen kritischen Abschnitt durch gegenseitigen Ausschluss.

- Ist diese Lösung Deadlock-frei? **Ja, …**
  - maximal 1 Prozess kann auf eine Gabel warten (Zyklus sind 2!)
  - Ein Prozess, der auf den Mutex wartet, hat keine Gabel
- Ist dies eine “gute” Lösung? **Nein, …**
  - Wenn $philo_n$ isst, blockiert $philo_{n+1}$ im kritischen Abschnitt. Alle anderen blockieren dann auch. Viele Spaghetti werden kalt…
  - **Geringe Menge an Nebenläufigkeit und ineffiziente Ressourcen-Nutzung!**

Speisende Philosophen: Version 3


```
const int N = 5;                          /* Anzahl Philosophen */
semaphore mutex = {1, NULL};              /* Gegenseitiger Ausschluss */
semaphore s[N] = {{0, NULL},...};         /* eine Semaphore pro Philos. */
enum { DENKEND, ESSEND, HUNGRIG } status[N]; /* Zustand Philos. */

int links(i) { return (i+N-1)%N; } /* Index linker Nachbar */
int rechts(i) { return (i+1)%N; }  /* Index rechter Nachbar */
```

```
void test (int i) {
  if (state[i] == HUNGRIG && state[links(i)] != ESSEND &&
      state[rechts(i)] != ESSEND) {
    state[i] = ESSEND;
    signal(&s[i]);
  }
}
```

```
void aufnehmen(int i) {
  wait(&mutex);
  state[i] = HUNGRIG;
  test(i);
  signal(&mutex);
  wait(&s[i]);
}
```

```
void weglegen(int i) {
  wait(&mutex);
  state[i] = DENKEND;
  test(links(i));
  test(rechts(i));
  signal(&mutex);
}
```

Diese Lösung ist ***Deadlock-frei*** und besitzt den höchsten Grad an Nebenläufigkeit!

Diskussion: Speisende Philosophen

- **Speziell:** es gibt meist viele unterschiedliche Ansätze, um zu garantieren, dass ein System Deadlock-frei ist
  - Lösungen unterscheiden sich im möglichen Grad der Nebenläufigkeit
  - Eine zu restriktive Lösung führt dazu, dass Ressourcen zumindest einen Teil der Zeit unnötig ungenutzt bleiben
- **Allgemein:** Die Speisenden Philosophen sind ein typisches Beispiel für die Verwaltung atomarer Ressourcen
  - Erfunden von E. Dijkstra (1965) [1]
  - Standardszenario, um Betriebssystem- und Sprachmechanismen zur nebenläufigen Programmierung zu evaluieren und demonstrieren

Verhindern von Deadlocks

- **Indirekte Methoden** invalidieren eine der Bedingungen 1–3
  1. Verwenden nicht blockierender Ansätze
  2. Nur atomare Ressourcenallokationen erlauben
  3. Präemption von Ressourcen durch Virtualisierung
     - **Virtueller Speicher**, virtuelle Geräte, virtuelle Prozessoren
- **Direkte Methoden** invalidieren Bedingung 4
  4. Einführung einer linearen/globalen Ordnung von Ressourcenklassen:
     - Ressource R_i kann nur dann erfolgreich vor R_j alloziert werden, wenn i linear vor j angeordnet ist (d.h. $i < j$)
- Regeln zur Vermeidung von Deadlocks
  - Methoden existieren zur Entwurfs- und Ausführungszeit

Deadlock-Erkennung

- Deadlocks werden oft (stillschweigend) akzeptiert („**Vogel-Strauß-Algorithmus**“)…
  - Nichts in einem System versucht, das Auftreten von Wartezyklen zu vermeiden!
  - Keine der vier Bedingungen wird invalidiert
- Ansatz: erzeuge **Allokationsgraphen** und suche nach Zyklen → O(n)
  - Zu häufiges Überprüfen verschwendet Ressourcen und Rechenzeit
  - Zu seltenes Überprüfen lässt Ressourcen ungenutzt
- **Zyklussuche** nur in großen Zeitintervallen durchführen, wenn…
  - Ressourcenanforderungen zu lange dauern
  - Die CPU-Last sinkt, auch wenn die Anzahl Prozesse steigt
  - Die CPU ist bereits lange ungenutzt (*idle*)

Deadlock-Auflösung

Wiederanlaufphase nach der Erkennungsphase

- **Terminieren von Prozessen** gibt Ressourcen frei
  - Terminiere Prozesse im Deadlock Schritt für Schritt (aufwendig)
    - Beginne mit dem ”effektivsten Opfer” (**welches ist das?**)
  - Terminiere alle Prozesse im Deadlock (evtl. großer Schaden)
- **Präemption von Ressourcen** beginnend bei der “effektivsten” (**?**)
  - *Roll back* oder *Neustart* des betroffenen Prozesses
    - Verwendung von Transaktionen, Checkpoints (aufwendig)
  - Ein *Verhungern* der Prozesse muss dabei vermieden werden
  - Zusätzlich: *Gefahr von Livelocks!*
- **Balance** zwischen Schaden und Aufwand:
  - Schäden sind unvermeidlich, daher sollten die Folgen berücksichtigt werden

Sicherheit

- Definitionen wichtiger Begriffe
  - ***Funktionale Sicherheit (Safety)***
    - Schutz gegen Risiken durch Hardware- und Softwarefehler oder -ausfälle
  - ***Informationssicherheit (Security)***
    - Schutz von Benutzern und Computern gegen *absichtlich herbeigeführte* Fehler (Angriffe)
- Beide Themen sind hochrelevant für Systemsoftware
  - Hier: nur Informationssicherheit
    (mehr: *Introduction to Security and Privacy* von Prof. Herrmann)
- Ausnutzung von *Sicherheitslücken*
  - *Malware*
  - Social Engineering

Betriebssystemsicherheit

- **Jemand...**
  - Unterscheidung von Personen und Gruppen von Personen
- **muss davon abgehalten werden...**
  - durch technische und organisatorische Methoden
- **einige...**
  - nur durch unsere Vorstellungskraft begrenzt
- **unerwartete Dinge zu tun!**
  - 1) Nicht autorisiertes Lesen von Daten (Geheimhaltung, Vertraulichkeit),
  - 2) Nicht autorisiertes Schreiben von Daten (Integrität),
  - 3) Arbeiten unter "falscher Flagge" (Authentizität),
  - 4) Nicht autorisierte Verwendung von Ressourcen (Verfügbarkeit)
  - **usw**…
- Wir unterscheiden zwischen **internen** und **externen Angriffen**

Malware-Beispiel: Viren

- Programmcode, der in ein regulär laufendes Programm eingefügt wird (kann so auch repliziert werden)
  - Virus schläft, bis das *infizierte* Programm ausgeführt wird
  - Start des infizierten Programms führt oft zur Reproduktion des Virus
  - Ausführung der Virus-Funktion kann zeitgesteuert erfolgen
- Arten von Viren
  - Bootsektorvirus: beim Systemstart ausgeführt
  - Makrovirus: in *scriptbaren* Programmen, z.B. Word oder Excel
    - Reproduziert sich über Dokumente (z.B. über mail)!
  - Ausführbares Programm als Virus
- Verteilung durch…
  - Austausch von Speichermedien (USB-Speichersticks usw.)
  - Emailanhänge
  - Webseiten…

Arten von Malware (1)

- ***Viren***
  - Programme, die unbeabsichtigt durch Benutzer verbreitet werden
  - *infizieren* andere Programme
  - …und verbreiten sich auf diese Art
- ***Würmer***
  - warten nicht auf Benutzeraktionen, um sich auf andere Computer zu verbreiten
  - versuchen *aktiv*, in neue Systeme einzudringen
  - nutzen *Sicherheitslücken* von Zielsystemen aus
- ***Trojanische Pferde*** ("Trojaner")
  - Programm, das als nützliche Anwendung (oder Spiel…) erscheint
  - zusätzlich zur nützlichen Funktion werden zusätzliche Funktionen ohne Kenntnis des Benutzers ausgeführt
    - z.B. Angreifern Zugriff auf den Computer via Internet ermöglichen

Arten von Malware (2)

- ***Rootkit***
  - Sammlung von Software-Werkzeugen, um…
    - zukünftige *logins* eines Angreifers zu verbergen
    - Prozesse und Dateien zu verstecken
  - wird installier, nachdem ein Computer *kompromittiert* wurde
  - kann sich selbst und seine Aktivitäten vor dem Benutzer verbergen
    - z.B. durch Manipulation von Programmen zur Anzeige von Prozessen (ps), Verzeichnisinhalten (ls), Netzwerkverbindungen (netstat) ...
    - …oder durch Manipulation systemweiter *shared libraries* (libc)
    - …oder direkt durch Manipulation des Betriebssystemkerns

- Malware verwendet oft eine **Kombination** dieser Malware-Arten

Beispiel: Social Engineering

- ***Kein*** Problem der Systemsoftware
  - …aber extrem wichtig
- Zugriff auf Informationen durch Ausnutzung menschlicher Fehler erlangen
- ***Phishing***
  - Datei eines Internetnutzers durch falsche Adressen erhalten (z.B. durch ähnliche Namen oder Tippfehler)
  - z.B. durch Verwendung gefälschter Emails von Banken
- ***Pharming***
  - Manipulation von DNS-Anfragen von Web-Browsern
  - Leitet Zugriffe um, z.B. auf gefälschte Bank-Websites
  - Die meisten Benutzer ignorieren Browser-Warnungen über ungültige Sicherheitszertifikate

Zugriffsrechtemanagement: Ziele

- Schützen gespeicherter Informationen vor
  - Bruch der ***Vertraulichkeit***
  - ***Diebstahl*** von Information
  - unerwünschter ***Manipulation***
    (inklusive Verschlüsselung: ***Ransomware***)

- in allen Mehrbenutzersystemen
  - …und jedes System, das mit dem Internet verbunden ist, muss als Mehrbenutzersystem angesehen werden!
  - …es läuft i.A. Code von nicht vertrauenswürdigen Anbietern, z.B. JavaScript auf einer Webseite im Browser

Zugriffsrechtemanagement: Anforderungen

- Alle **Objekte** in einem System müssen **eindeutig** und **unfälschbar identifizierbar** sein
- (externe) **Benutzer** eines Systems müssen **eindeutig** und **unfälschbar identifizierbar** sein
  - ➛ Authentisierung
- Zugriff auf Objekte nur erlaubt, wenn der Benutzer über die notwendigen **Zugriffsrechte** verfügt

- Zugriff auf Objekte sollte nur über die entsprechende **Objektverwaltung** möglich sein
  - Zugriffsrechte müssen auf **unfälschbare** Weise gespeichert werden
  - Übertragung von Zugriffsrechten muss **kontrolliert** stattfinden
  - es muss möglich sein, grundlegende Zugriffsmechanismen **mit geringem Aufwand** zu validieren

Zugriffsrechtemanagement: Entwurfsprinzipien

- ***Prinzip der geringsten Rechte***
  (principle of least privilege, PoLP)
  - Erlaube einer Person oder Softwarekomponente nur die Zugriffsrechte, die zur Realisierung der Funktionalität benötigt werden
  - Standardfall: ***Ablehnung*** von Zugriffsrechtes
  - Gegenbeispiel: "root" in Unix

- ***Sichere Standardeinstellungen***
  - Beispiel: neu installierte Server-Software

- ***Trennung von Aufgaben***
  - Verschiedene Bedingungen existieren, um eine Operation zu erlauben

Zugriffsmatrix

- Elemente der Matrix:
  - Subjekte (Personen/Benutzer, Prozesse)
  - Objekte (Daten, Geräte, Prozesse, Speicher, …)
  - Operationen (Lesen, Schreiben, Löschen, Ausführen, …)
- Frage: Ist `Operation(Subjekt, Objekt)` erlaubt?

		Objekte	
Subjekte		**Zugriffs-rechte**	

Grundmodell: Datei-/Prozessattribute

Universität Bamberg

- Eigenschaften im Bezug auf Benutzer:
  - von welchem Benutzer wird der Prozess ausgeführt?
  - welcher Benutzer ist **Eigentümer** einer Datei?
  - welche Zugriffsrechte gibt der Eigentümer einer Datei sich selbst und welche Zugriffsrechte an andere Benutzer?

- Zugriffsrechte eines Prozesses beim Zugriff auf eine Datei
  - Attribut des Prozesses: **Benutzer-ID**
  - Attribut der Datei: **Eigentümer-ID**

	Datei 1	Datei 2	Datei 3
Benutzer 1			
Benutzer 2		lesen	
Benutzer 3			
Benutzer 4			

Zugriffsmatrix-Varianten

- Spalten: ***ACL – Access Control Lists***
  - Bei jedem Zugriff auf ein Objekt werden die Zugriffsrechte auf Grundlage der Identität des anfordernden Subjekts (Benutzer) ausgewertet
- Zeilen: ***Capabilities***
  - Bei jedem Zugriff auf ein Objekt wird eine Eigenschaft ausgewertet, die dem Subjekt zugeordnet ist, und bei Bedarf auf andere Subjekte übertragen werden kann
- Regelbasiert: ***erzwungene Zugriffskontrolle***
  - *Regeln* werden bei jedem Zugriff ausgewertet

Zugriffskontroll-Listen: ACLs

- Spaltenweise Auswertung der Zugriffsmatrix: ***Zugriffskontroll-Liste*** (ACL)
- ACLs geben für jedes Objekt an, welche Subjekte welche Operationen auf dem Objekt ausführen dürfen

		Objekte	
Subjekte		**Zugriffs-rechte**	

ACLs

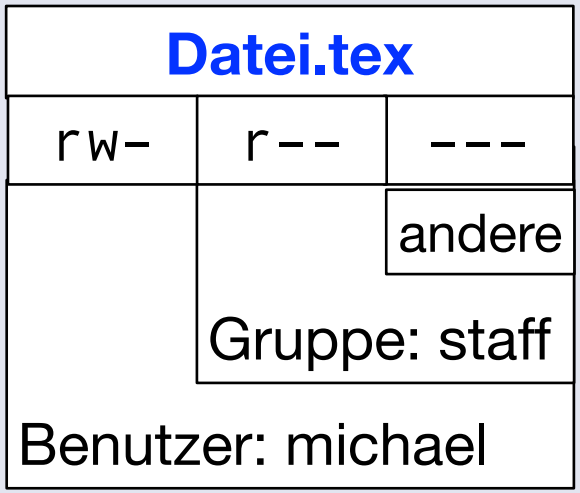
- ACLs können konfiguriert werden durch…
  - Subjekte, die ACL-Eintrag für dieses Zugriffsrecht haben
  - Den Erzeuger des Objekts (Daten)
- Beispiel: *Multics* OS – **Tripel** (Benutzer, Gruppe, Zugriffsrechte)

```
Datei 0 (Jan, *, RWX)
Datei 1 (Jan, system, RWX)
Datei 2 (Jan, *, RW-), (Els, staff, R--), (Maike, *, RW-)
Datei 3 (*, student, R--)
Datei 4 (Jelle, *, ---), (*, student, R--)
```

- Windows (ab Windows NT)
  - Objekt: erlauben oder verweigern
  - Rechte: voller Zugriff, Verändern, Lesen & Ausführen, ...

Unix-Zugriffsrechte

- Unix: einfache Zugriffskontroll-Listen
- Prozesse haben eine **Benutzer-ID** und eine **Gruppen-ID**
- Dateien haben **Eigentümer** *(owner)* und **Gruppe**
- Zugriffsrechte beziehen sich auf **Benutzer** (user), **Gruppe**, und alle anderen (**o**thers)



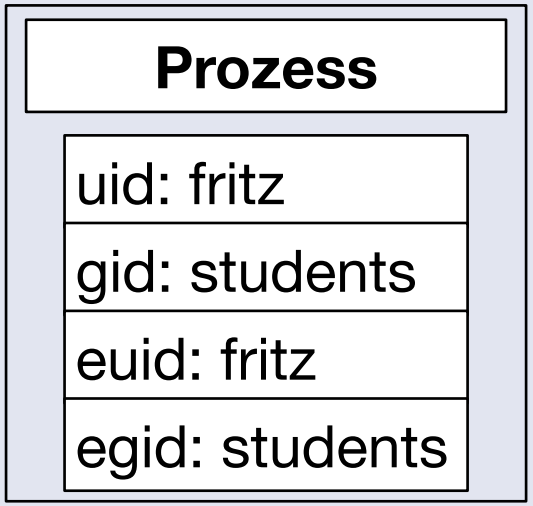
Datei-Attribute:

```
rwx
```

- **x**ecute (Ausführen): ja/nein
- **w**rite (Schreiben): ja/nein
- **r**ead (Lesen): ja/nein

Unix: Benutzer und Prozesse

- Jeder Prozess repräsentiert einen Benutzer
- Prozessattribute:
  - Benutzer-ID (**uid**), Gruppen-ID (**gid**)
  - Effektive uid (**euid**), effektive gid (**egid**)
    - Bestimmen Zugriffsrechte bei Zugriff auf Dateien
- Nur wenige hochprivilegierte Prozesse können ihre uid und gid ändern
  - z.B. der login-Prozess
- Nach Prüfung des Benutzer-Passworts setzt der login-Prozess uid, gid, euid und egid
  - Alle anderen Prozesse: Kinder von login
- Kindprozesse erben die Eltern-Attribute



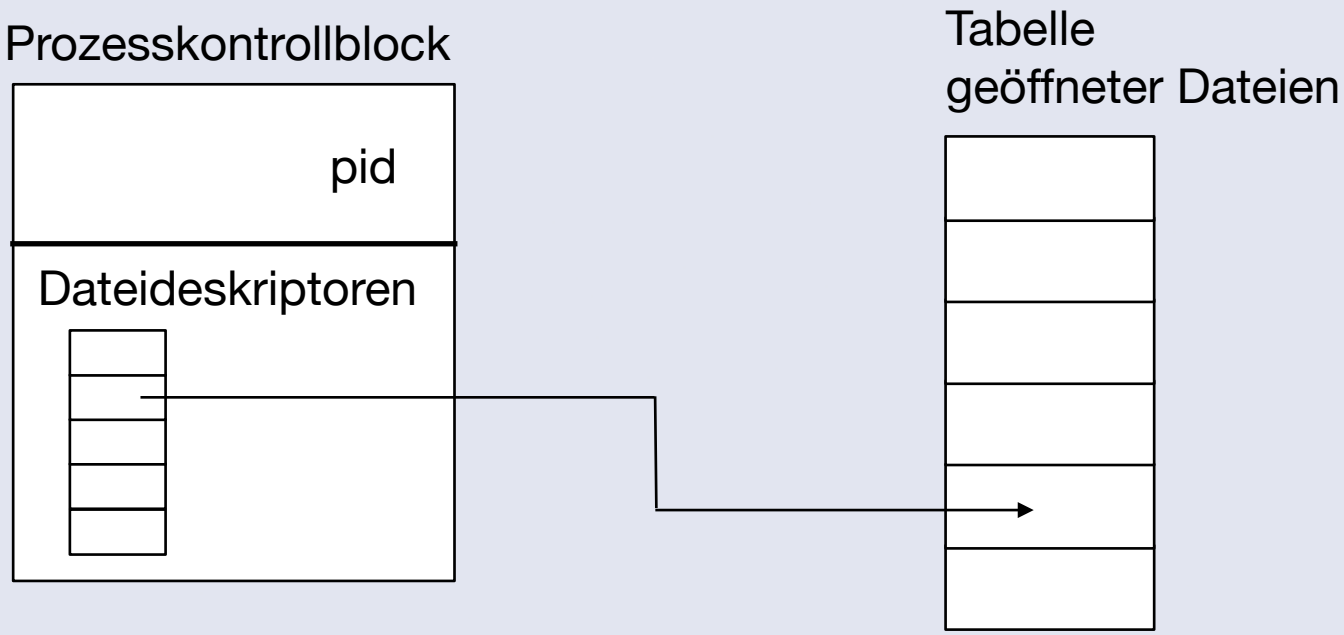
Capabilities

- Zeilenweise Auswertung der Zugriffsmatrix: ***Capability***
- Capabilities geben für jedes Subjekt an, auf welche Weise es auf ein gegebenes Objekt zugreifen darf

		Objekte	
Subjekte		**Zugriffs-rechte**	

Beispiel

- Einfache Implementierung: Dateideskriptoren in Unix
- Weitergegeben durch den Systemaufruf `fork`
  - Erlaubt Zugriff auf Dateien, ohne die Unix-Zugriffsrechte dauernd neu zu überprüfen



Regelbasierte Zugriffsmatrix

- ***Erzwungene Zugriffskontrolle***
- Konzept:
  - Subjekte und Objekte besitzen Attribute ("labels")
  - Entscheidung über Zugriff durch Auswertung von *Regeln*
- Implementiert in "security kernels", z.B. SELinux

		Objekte	
Subjekte		**Zugriffs-rechte**	

Wird für jeden Zugriff anhand einer Menge von Regeln ausgewertet

Referenzen

[1] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau
Operating Systems: Three Easy Pieces
https://pages.cs.wisc.edu/~remzi/OSTEP/

[2] Edsger W. Dijkstra
Hierarchical ordering of sequential processes
Acta Informatica 1(2): 115–138, Juni 1971

[3] Allen B. Downey
The Little Book of Semaphores
Green Tea Press 2016
https://greenteapress.com/wp/semaphores/