Betriebssysteme
8. Deadlocks

Tobias Lauer

Das Problem der speisenden Philosophen

- „Dining Philosophers Problem“ (Dijkstra)

Quelle: Wikipedia

Was sind Ausgangsbedingungen für Deadlocks?

1. Gemeinsame Ressourcen, die aber nur exklusiv von einem Prozess verwendet werden können (mutual exclusion)
2. Mehrere Prozesse, von denen jeder gleichzeitig mehr als eine der gemeinsamen Ressourcen benötigt (hold and wait)
3. Keine Möglichkeit, einmal zugewiesene Ressourcen wieder freizugeben, bevor der Prozess ihn selbst freigibt (no preemption)

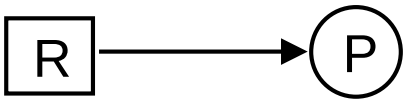
- Eine Gruppe von N Prozessen sind in einem Deadlock, wenn jeder der Prozesse eine oder mehrere der gemeinsamen Ressourcen besitzt und gleichzeitig auf weitere Ressourcen wartet, die ein anderer der Prozesse aus dieser Gruppe schon erworben hat (circular wait)
- Deadlock = zyklische Wartebeziehung

Deadlock im realen Leben

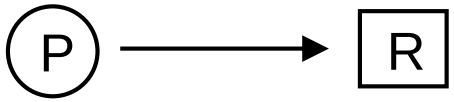
Quelle: http://mcs109.bu.edu/site/?p=deadlock

Deadlocks sind zyklische Wartebeziehungen

- Graphische Darstellung: Betriebsmittelgraph



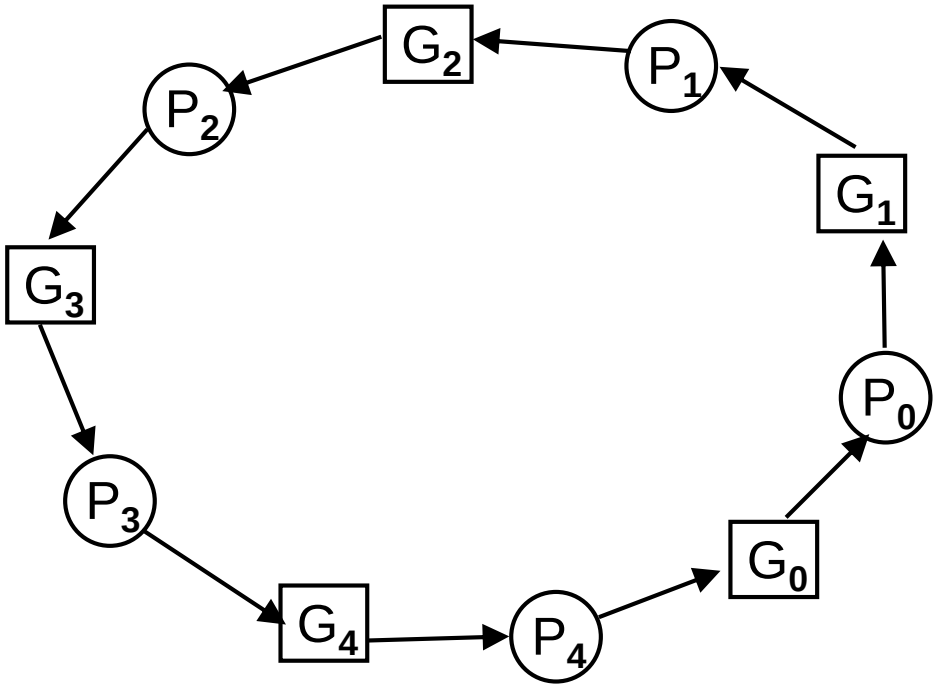
Ressource R ist Prozess P zugeordnet



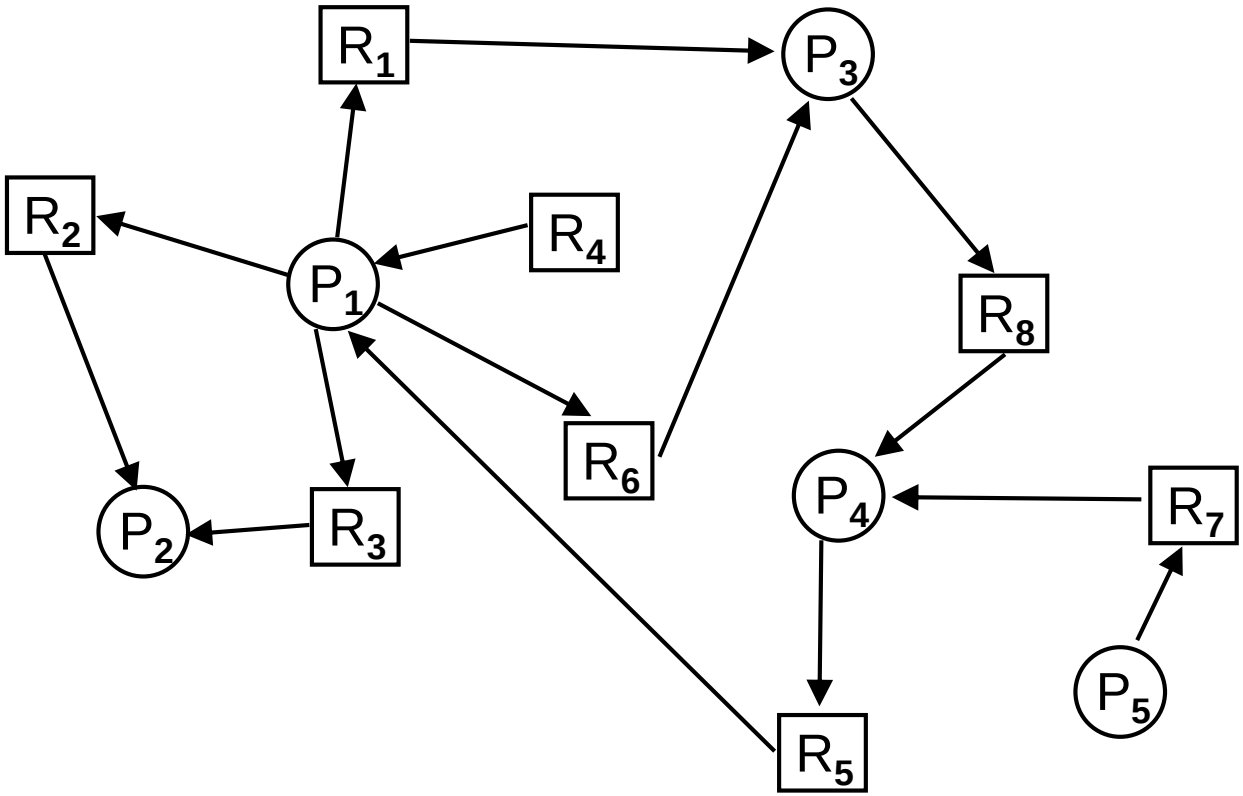
Prozess P wartet auf Ressource R

- Deadlock bei den speisenden Philosophen:

G_i = Gabel i



Deadlock – Ein etwas komplexeres Beispiel

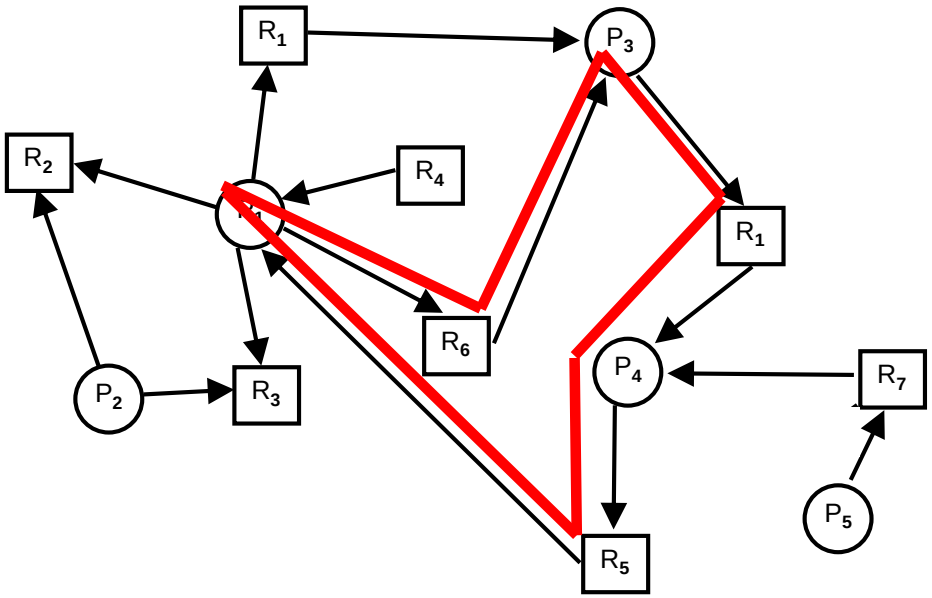


Wie geht man mit Deadlocks um?

1. Deadlocks ignorieren (Vogel-Strauß-Verfahren)
   - Anwendung/User soll Deadlocks manuell lösen (oder „watchdog“ löst irgendwann ein System-Reset aus)
   - Problematisch, aber kann sinnvoll sein
2. Deadlocks erkennen (aber nicht automatisch lösen)
   - Anwendung bekommt Fehlermeldung bei Deadlock
3. Deadlocks verhindern
   - „Konstruktive“ (prinzipielle) Vorsichtsmaßnahmen bei der Anforderung von Ressourcen
   - Teil der Programmentwicklung
4. Deadlocks vermeiden
   - Auf die konkrete Situation angepasste „vorsichtige“ Anforderungsstrategie für Ressourcen
   - Zur Laufzeit

Deadlock-Erkennung („Deadlock Detection“)

- Erkennen von Zyklen im durch die Wartebeziehungen aufgespannten Graph
- Tiefensuche oder Breitensuche
  (→ Vorlesung Algorithmen und Datenstrukturen)
- 1 erkannter Zyklus im Graph = Deadlock im System



Deadlock-Verhinderung - 1

- Einfache Methode:
  - Immer alle Ressourcen gleichzeitig anfordern
- Vorteil: Verhindert Deadlocks, einfacher Algorithmus
- Nachteil: Anwendung muss alle Ressourcen, die sie irgendwann braucht, im voraus kennen
- Nachteil: Ressourcen nicht effizient genutzt: müssen bereits am Anfang belegt werden, obwohl sie evtl. erst gegen Ende gebraucht werden.
- Beispiel (Speisende Philosophen):
  - (extrem) Philosoph fordert zum Essen ALLE Gabeln an und legt danach alle wieder ab
  - (besser) Philosoph fordert kurzzeitig (nur für das Aufnehmen seiner beiden Gabeln) ALLE Gabeln an
  - (noch besser) Philosoph schaut erst, ob BEIDE Gabeln frei sind
    WENN JA: nimmt sich beide Gabeln und isst
    WENN NEIN: wartet, bis beide frei sind

Deadlock-Verhinderung - 2

- Alternative Methode:
  - Ressourcen durchnummerieren
  - Ressourcen immer in Reihenfolge aufsteigender Nummern anfordern
- Vorteil: Verhindert Deadlocks, einfacher Algorithmus
- Nachteil: Anwendung muss so strukturiert sein, dass sie Anforderungen in dieser Reihenfolge akquirieren kann (keine nachträglichen Ressourcenanforderungen nötig, ..)
- Beispiel (Speisende Philosophen): Gabeln werden nach ihrer Ordnungsnummer angefordert:

```
P0 : G0      P1 : G1      P2 : G2      P3 : G3      P4 : G0
     G1           G2           G3           G4           G4
```

Deadlock-Verhinderung - 3

Gegenseitigen Ausschluss unnötig machen durch *Spooling*:

- Ordne Betriebsmittel X nur EINEN besonderen Prozess zu, der die Kontrolle darüber hat und als „Server“ dafür agiert
- Alle Zugriffe auf X gehen über diesen Spooler
- Vorteil: Kein Deadlock, das X beinhaltet (sofern Spooler-Queue groß genug)
- Nachteil:
  - nur begrenzt einsetzbar (z.B. Drucker)
  - Nicht für alle Betriebsmittel

Deadlock-Verhinderung - 4

In der Programmierpraxis sinnvoll:

- Synchronisationstypen hierarchisch einsetzen
- Beispiel
  - unterste Ebene: kritische Abschnitte für die einfachen Grundfunktionen eines Programms (kurze Sperrzeiten, keine Gefahr des Blockierens)
  - evtl. mehrere Level kritischer Abschnitte, die nur in einer bestimmten Reihenfolge angefordert werden
    z.B. Critical_Region (CB) zum Modifizieren eines Kontrollblocks CB
    darüber: Critical_Region (Func) zum Aufruf von Basisoperationen;
    Alle CB-Abschnitte werden nur von Func-Aufrufen verwendet.
  - Höhere Ebene: Semaphoroperationen zum Blockieren vor leeren Puffern, etc.
  - ABER NICHT: Aufruf eines Semaphors, während man in einem der kritischen Abschnitte ist.
- Allgemein wichtig ist die strukturierte Programmierung: „Synchronisationsarchitektur“, „Sperrkonzept“

Deadlock-Vermeidung

- Grundidee:
  - Sei bei jeder neuen Anforderung eines Prozesses wachsam
  - Gewähre die Anforderung nur, wenn der Zustand danach „sicher“ ist

  Sicher: Es gibt mindestens eine Scheduling-Reihenfolge, die nicht zum Deadlock führt, selbst wenn alle anderen Prozesse sofort ihre maximale Anzahl an Ressourcen anfordern.

  Unsicher: Wenn alle Prozesse sofort ihre maximale Anzahl an Ressourcen anfordern würden, gäbe es ein Deadlock.

- D.h. konservative Strategie; Sicherheitsreserven lassen
- Nutze auf Basis der aktuellen Situation den Betriebsmittelgraph und triff eine Vorhersage
- Vorteil: vermeidet Deadlocks, vermeidet Vorallokierung aller Ressourcen
- Nachteil: Gesamtmenge der Anforderungen muss bekannt sein

Deadlocks – Weitere Anmerkungen

- Was geschieht im Fall einer Deadlock-Erkennung?
  - Wenn möglich Revidierung der letzten Ressourcenanforderung
  - Wenn möglich Rücksetzen des Prozesses auf einen früheren „Checkpoint“
  - Abbruch von 1 oder mehreren der verklemmten Prozesse
  - Auswahl z.B. nach Effizienzkriterien (z.B.: Brich denjenigen Prozess mit der bisher geringsten aufgelaufenen CPU-Zeit ab)

- Für die Praxis am wichtigsten ist die Deadlockverhinderung auf der Basis einer guten Anwendungs-/Systemarchitektur
  - Robuste, überschaubare Ablaufstrukturen
  - Definition von Sperrhierarchien
  - Analyse des Nebenläufigkeitsverhaltens
  - Deadlockverhinderung ist Aufgabe des (System-)Programmierers

- In der Praxis auch häufig eingesetzt: Timeouts (watchdog)