Humboldt University

Computer Science Department
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Operating Systems Principles

Lab 4 – Tax Collectors Dilemma

Organisatorisches

Datum	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
16.04. - 20.04.				**Praktikum 1**	Erste Aufgabe (Beginn) **Praktikum 3**
23.04. – 27.04.				**Praktikum 2**	**Praktikum 4**
30.04. – 04.05.				**Praktikum 1**	Zweite Aufgabe (Beginn) **Praktikum 3**
07.05. – 11.05.	Erste Aufgabe (Abgabe)			**Praktikum 2**	**Praktikum 4**
14.05. – 18.05.				**Frei (Himmelfahrt)**	Dritte Aufgabe (Beginn) **Praktikum 3 + 1**
21.05. – 25.05.	Zweite Aufgabe (Abgabe)			**Praktikum 2 + 1**	**Praktikum 4 + 1**
28.05. – 01.06.				**Praktikum 1**	Vierte Aufgabe (Beginn) **Praktikum 3**
04.06. – 08.06.				**Praktikum 2**	**Praktikum 4**
11.06. – 15.06.	Dritte Aufgabe (Abgabe)			**Praktikum 1**	**Praktikum 3**
18.06. – 22.06.				**Praktikum 2**	**Praktikum 4**
25.06. – 29.06.	Vierte Aufgabe (Abgabe)			**Praktikum 1**	**Praktikum 3**
02.07. – 06.07.				**Praktikum 2**	**Praktikum 4**
09.07. – 13.07.					

Viertes Praktikum

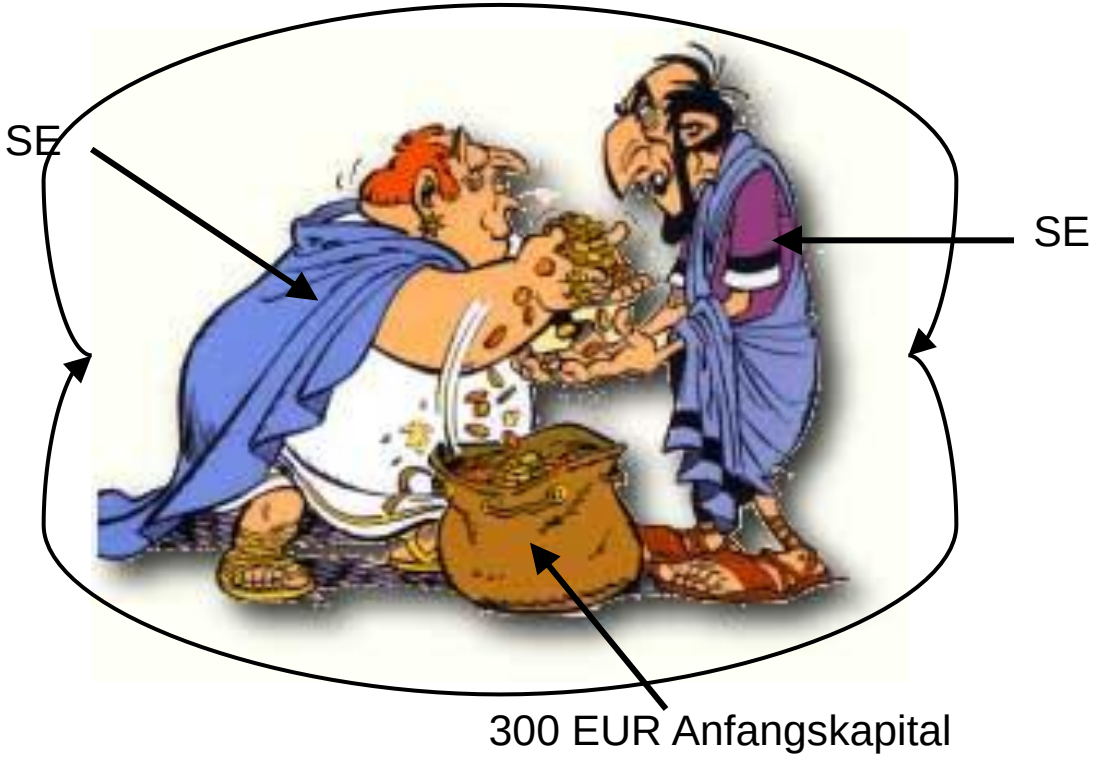
- 3 Wochen Zeit zum Lösen der Aufgaben
  - Aufgabenstellung auf der SAR Website
  - Abgabe über GOYA

- In dieser Woche
  - Erläutern der Aufgabenstellung
- Folgende 2 Wochen
  - Zeit zur Bearbeitung
- Nächste Veranstaltung
  - 14/15/21/22.06.12

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 4 – Tax Collectors Dilemma

- Szenario
  - Stadt bestehend aus einer festen Anzahl Steuereintreibern (SE)
  - Steuern eintreiben
  - Steuern bezahlen



Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 4 – Tax Collectors Dilemma

- Tagesablauf eines SE
  - SE A fordert Steuern von einem zufällig ausgewählten SE B
  - Gutschrift der Steuerschuld von B nach A
  - Falls B nicht zahlen kann, wartet A
  - Realisierung der SE durch Threads
    - Alle Threads wiederholen den Ablauf
    - Nach Gutschrift soll ein anderer Thread an die Reihe kommen (yield)

SE A

50% auf ganze 100er
gerundet, mind. 100

SE B

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 4 – Tax Collectors Dilemma

- Beenden nach einer vorgegebenen Zeit
  - Beenden aller SE-Threads
  - Ausgabe von Statistiken

Pro Steuereintreiber
- Anzahl Eingangsbuchungen
- Anzahl Ausgangsbuchungen
- Kontostand

Systemweit
- Summe Geld im System
- Summe aller Eingangsbuchungen
- Summe aller Ausgangsbuchungen
- (Anzahl erkannter Deadlocks)

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

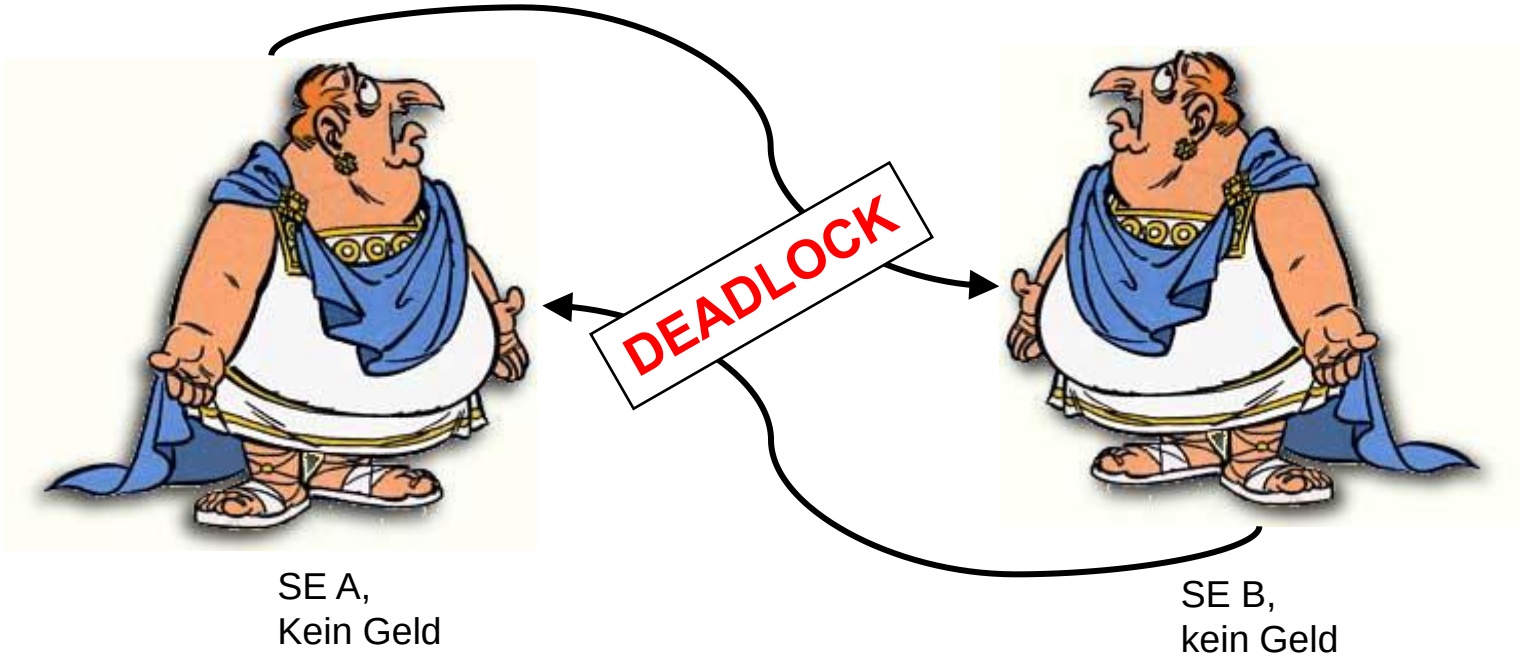
Lab 4 – Tax Collectors Dilemma

Weitere Anforderungen:

- Korrektheit
  - Gesamtmenge Geld im System ist konstant
  - Es darf kein Geld entstehen oder vernichtet werden
- Fairness
  - SE haben jeweils die gleiche Chance, an die Reihe zu kommen
- Deadlock-Behandlung

Lab 4 – Deadlock-Behandlung

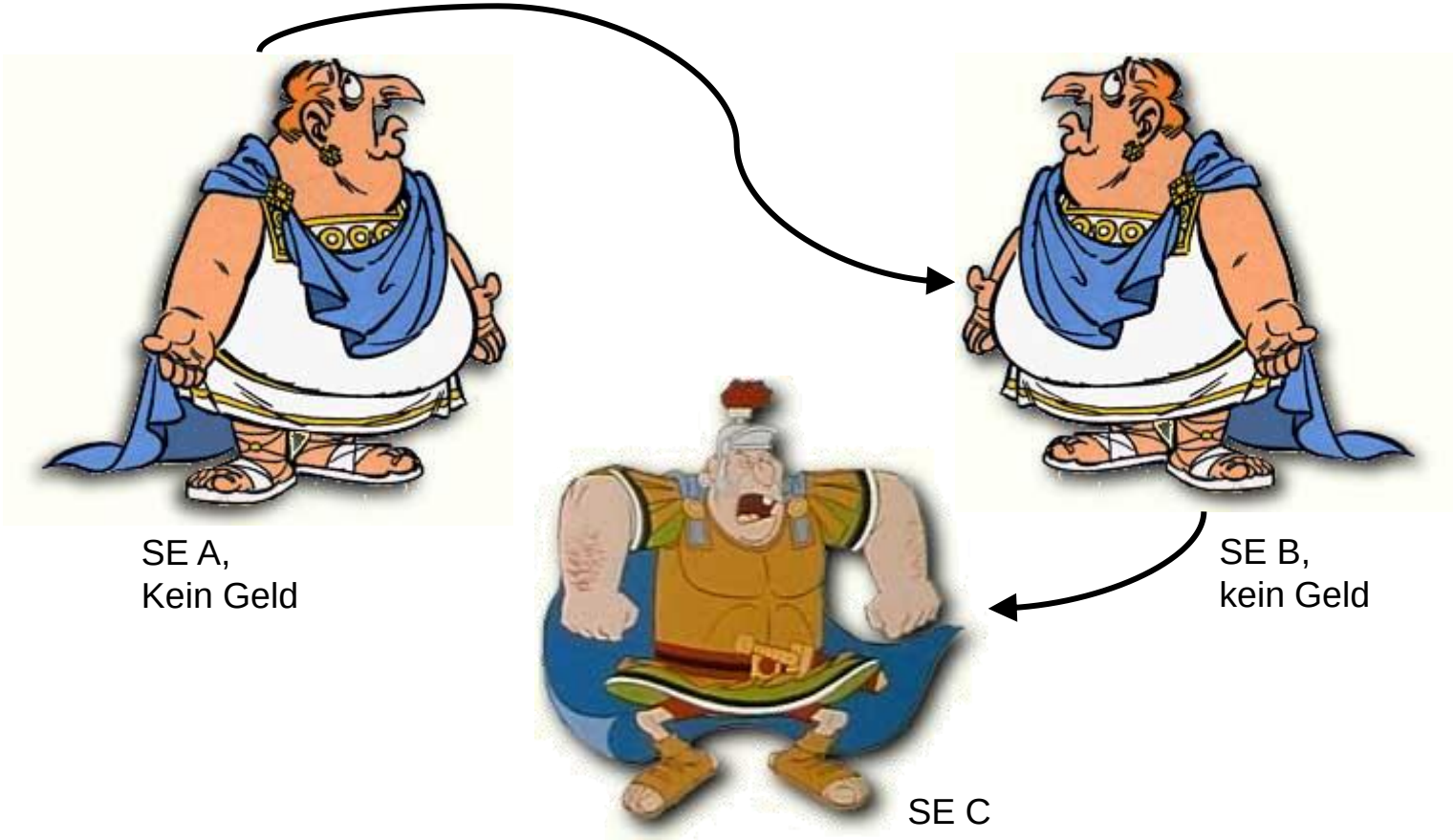
- Deadlock-Szenario
  - SE A und B besitzen beide kein Geld
  - A stellt Forderung an B, die nicht bedient werden kann, und wartet
  - B wählt A aus und wartet auch



Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 4 – Deadlock-Behandlung

- Mögliche Lösung
  - B fordert nicht von A, sondern von dem liquiden SE C
  - Was ist, wenn keine liquiden SE mehr übrig?
  - Andere Deadlock-Szenarien?



Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 4 – Tax Collectors Dilemma

- C/C++ unter Linux

- Subset der Pthreads API
  - Thread-Funktionen: pthread_
    - create, exit, join, cancel, equal, self, setcancelstate, setcanceltype, testcancel; sched_yield
  - Mutex-Funktionen
    - pthread_mutex_*
    - pthread_mutexattr_*

Deadlock-Behandlung

Notwendige Bedingungen für das Eintreten eines Deadlocks:

1. Mutual exclusion
   - Der Zugriff auf die Ressourcen ist exklusiv
2. Hold and wait
   - Es existiert ein Prozess der auf eine Ressource wartet, welche von einem anderen gehalten wird
3. No preemption
   - Ressourcen werden nur freiwillig wieder freigegeben
4. Circular wait:
   - Es existiert ein Zyklus mehrerer Prozesse, in dem jeder Prozeß mindestens eine Ressource anfordert, die ein anderer Prozeß besitzt

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Deadlock-Behandlung

Addressing Deadlock

- Ignore
  - Solve deadlocks by manual intervention, e.g. have the operator reboot the machine if it seems too slow
- Prevention
  - Design the system so that deadlock is impossible
- Avoidance
  - Construct a model of system states, then choose a strategy that will not allow the system to go to a deadlock state
- Detection & Recovery
  - Check for deadlock (periodically or sporadically), then recover

Vogel-Strauß-Algorithmus

Ignorieren von Deadlocks

- wenn Eintrittswahrscheinlichkeit hinreichend gering
- wenn Aufwand und Nutzen in keinem sinnvollen Verhältnis stehen
- Jedes BS nutzt diese “Strategie” zumindest zum großen Teil

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Detection & Recovery

Erkennung

- Bspw. mit Resource Allocation Graph
- Zyklen im Graphen

Behebung

- Entzug einer Ressource
- Rollback mit Checkpointing
- Abbruch eines Prozesses

Deadlock- Prevention

Idee:

- Konzeptioneller Ausschluß einer der Eintrittsbedingungen

1. Mutual exclusion
   - Serialisierung: Spooler für den Drucker
2. Hold and wait
   - Alle benötigten Ressourcen werden auf einmal angefordert
   - Alternative: beim Nachfordern zunächst alles zurückgeben
3. No preemption
   - Timeouts für das Warten auf Ressourcen
4. Circular wait
   - Zuteilung nur in festgelegter Reihenfolge

POSIX-Threads (Pthreads)

- Standard IEEE P1003.1c
- In vielen Systemen realisiert
- Konzepte in anderen Thread-Realisierungen meist ähnlich
- POSIX-konforme Realisierungen unter Linux

Die Pthread-Schnittstelle

Thread-Erzeugung

- Programmiermodell
  - Bei Start eines Prozesses existiert genau ein Thread
  - Dieser erzeugt ggf. weitere Threads und wartet auf deren Ende
  - Terminierung des Prozesses bei Terminierung des Master-Threads
- Funktion zur Thread-Erzeugung

```
int pthread_create(
        pthread_t *new_thrd_ID,
        const pthread_attr_t *attr,
        void *(*start_func)(void *),
        void *arg)
```


Systems Architecture Group
http://sar.informatik.hu-berlin.de

Die Pthread-Schnittstelle

Thread-Attribute

- Stackgröße
- Scheduling-Schema und Priorität
- **detachstate**: Verhalten bei Beendigung desThreads
  Bei **detached thread** erfolgt die Freigabe der Ressourcen sofort bei Beendigung, ansonsten erst nach Abschlußsynchronisation

Die Pthread-Schnittstelle

Thread-Verwaltung

- **pthread_self**
  - Liefert eigene Thread-ID
- **pthread_exit**
  - Eigene Terminierung
- **pthread_cancel**
  - Terminiert einen Thread
  - Terminierung nur an bestimmten Punkten
  - Vor Terminierung **cleanup handler** aufrufen

Die Pthread-Schnittstelle

Thread-Verwaltung...

- **pthread_join**
  - Wartet auf Terminierung des spezifizierten Threads (Abschlußsynchronisation)
- **pthread_sigmask**
  - Setzt Signalmaske (jeder Thread hat eigene Maske)
- **pthread_kill**
  - Sendet Signal an anderen Thread innerhalb des Prozesses
  - Von außen nur Signal an irgendeinen Thread möglich

Systems Architecture Group
http://sar.informatik.hu-berlin.de

Die Pthread-Schnittstelle

Mutex-Operation (wechselseitiger Ausschluß)

- **pthread_mutex_init**
  - Initialisiert Sperrvariable (mutex)
- **pthread_mutex_destroy**
- **pthread_mutex_lock**
  - Blockiert Thread, bis Sperre frei ist und belegt dann die Sperre
- **pthread_mutex_trylock**
  - Belegt Sperre, falls möglich / kein Blockieren
- **pthread_mutex_unlock**

- Anmerkungen
  - Bei Terminierung eines Threads werden Sperren nicht automatisch freigegeben