Einführung in die technische Informatik

Christopher Kruegel chris@auto.tuwien.ac.at

http://www.auto.tuwien.ac.at/~chris

Betriebssysteme

- Aufgaben
  - Management von Ressourcen
  - Präsentation einer einheitlichen Schnittstelle für Anwendungen
- 4 wichtige Gebiete
  - *Prozessmanagement*
  - Speichermanagement
  - Dateisystem
  - Eingabe und Ausgabe
- System Calls
  - Funktionen, die ein Betriebssystem den Anwendungen zur Verfügung stellt

Prozessmanagement

- Konzept
  - Betriebssystem stellt eine Reihe von virtuellen Computern zur Verfügung
  - Ausführung eines Programms auf einem dieser virtuellen Computer heißt Prozess
  - virtuelle Computer vermitteln den Eindruck, dass jeder Prozess auf einer eigenen CPU mit eigenem Speicher läuft
  - in Wirklichkeit wird zwischen den Prozessen hin- und hergeschaltet und diese laufen auf einer einzigen realen CPU

  ➢ Quasi-Parallelität oder logische Parallelität

Prozessmanagement

- Programm
  - statisches Objekt im Dateisystem
  - Folge von Instruktionen
  - unveränderlich über die Zeit

- Prozess
  - dynamische Abbild eines Programms
  - läuft in einer bestimmten, veränderlichen Umgebung ab → Context
  - Folge von ausgeführten Instruktionen auf der CPU
  - existiert nur für eine bestimmte Zeit
  - mehrere Prozesse können das selbe Programm ausführen

Prozessmanagement

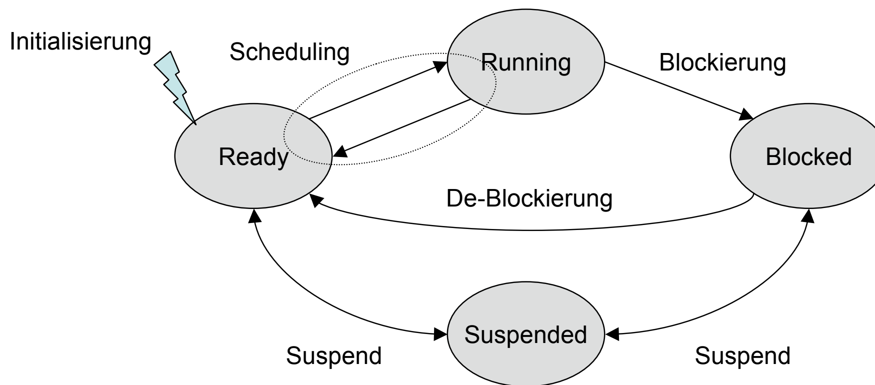
- Prozessablauf
  - Prozesse können erzeugt werden
    - durch das Betriebssystem
      - `init` task
    - durch andere Prozesse
      - `fork()`
  - Prozesse können beendet werden
    - freiwillig
      - `exit()`
    - wegen eines Fehlers (Absturz)
    - durch einen anderen Prozess
      - `kill()`

Prozessmanagement

- Prozesszustand
  - Prozesse kann auf der CPU exekutiert werden (RUNNING)
  - Prozess bereit sein, auf der CPU exekutiert zu werden (READY)
  - Prozess kann auf externe Ereignisse warten (BLOCKED)
- Zustandsübergänge
  - ausgelöst durch
    - Prozess selbst (Termination)
    - Scheduler
    - Ressourcen werden verfügbar
  - beschrieben durch Zustandsgraph

Prozessmanagement

- Zustandsgraph



Prozesse

- Betriebssystem verwaltet Prozesse mittels *process table*
- Process table
  - speichert Prozessdeskriptor (oder process control block) für jeden Prozess
- Prozessdeskriptor speichert relevante Informationen eines Prozesses
  - Prozess-ID
  - Zustand
  - Priorität
  - Besitzer des Prozesses (z.B., UID)
  - Zugriffsrechte (z.B., EUID)
  - Registerinhalte
  - Verweise auf benutzte Speicherbereiche (Daten, Stack)
  - Verweise auf offene Files

Prozesse

- Prozesse
  - besteht aus Ressourcen (Adressraum, geöffnete Dateien)
  - Ausführung von Instruktionen (threads of execution)
- Zustand des Prozesses bei der Ausführung
  - Werte in den Registern (und des Program Counters)
  - Verlauf (History) der Ausführung und lokale Variablen am Stack

Prozesse


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Adressraum (Daten)

i = ?

Registerinhalte (hier Program Counter)

PC = 15

Stack (Verlauf der Ausführung)

Prozesse


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Adressraum (Daten)

i = 42

Registerinhalte (hier Program Counter)

PC = 16

Stack (Verlauf der Ausführung)

Prozesse


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Adressraum (Daten)

i = 42

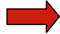
Registerinhalte (hier Program Counter)

PC = 10

Stack (Verlauf der Ausführung)

17 [return value of f()]

Prozesse



```
1: int i;
2:
3: g()
4: {
5:    printf(“Wert von i ist %d\n“, i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Adressraum (Daten)

i = 42

Registerinhalte (hier Program Counter)

PC = 5

Stack (Verlauf der Ausführung)

17 [return value of f()]

11 [return value of g()]

Prozesse

```
1: int i;
2:
3: g()
4: {
5:    printf(“Wert von i ist %d\n“, i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Wert von i ist 42

Adressraum (Daten)

i = 42

Registerinhalte (hier Program Counter)

PC = 6

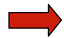
Stack (Verlauf der Ausführung)

17 [return value of f()]

11 [return value of g()]

Prozesse


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Adressraum (Daten)

i = 42

Registerinhalte (hier Program Counter)

PC = 11

Stack (Verlauf der Ausführung)

17 [return value of f()]

Prozesse


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Adressraum (Daten)

i = 42

Registerinhalte (hier Program Counter)

PC = 17

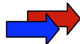
Stack (Verlauf der Ausführung)

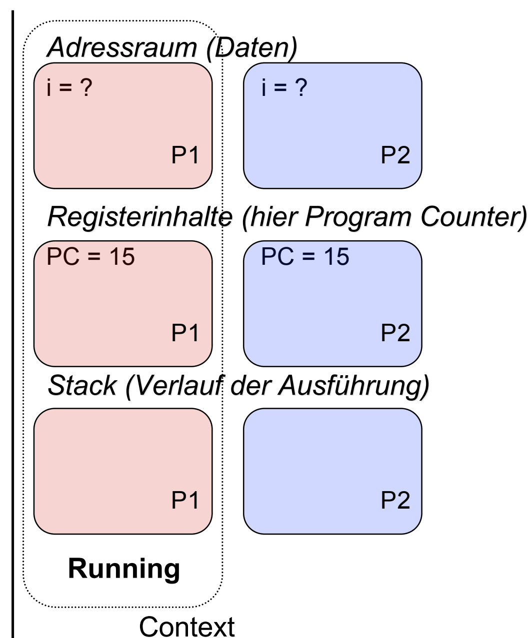
Prozesse

- Parallele Prozesse
  - unabhängige Ressourcen (Adressraum, geöffnete Dateien)
  - unabhängige Ausführung (Stack und Register)
  - können natürlich das selbe *Programm* ausführen

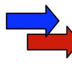
Parallele Prozesse


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

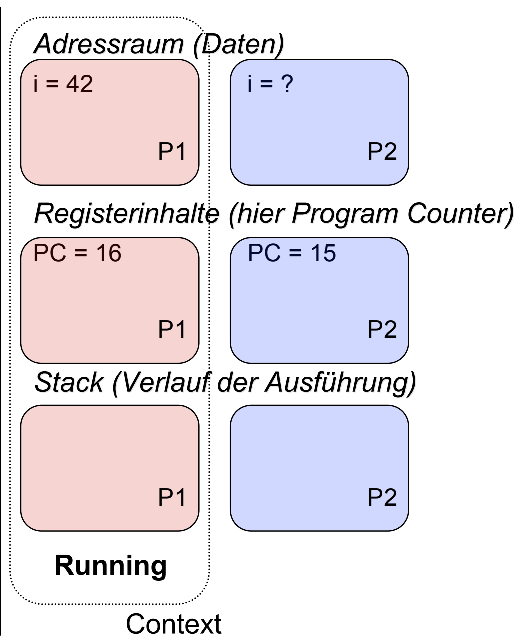




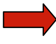
Parallele Prozesse



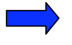
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf(“Wert von i ist %d\n“, i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

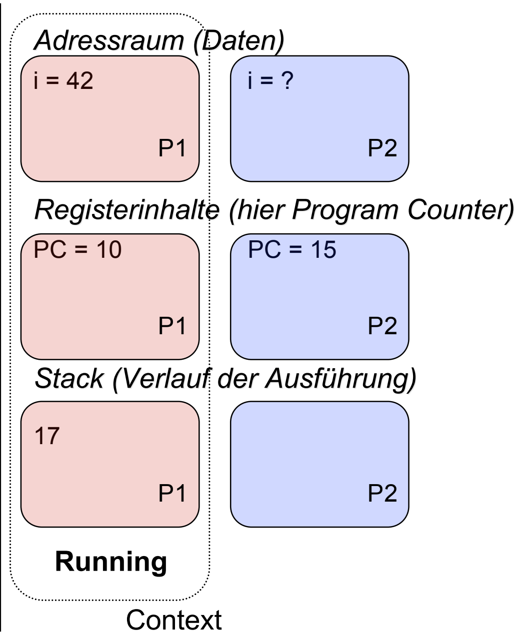


Parallele Prozesse



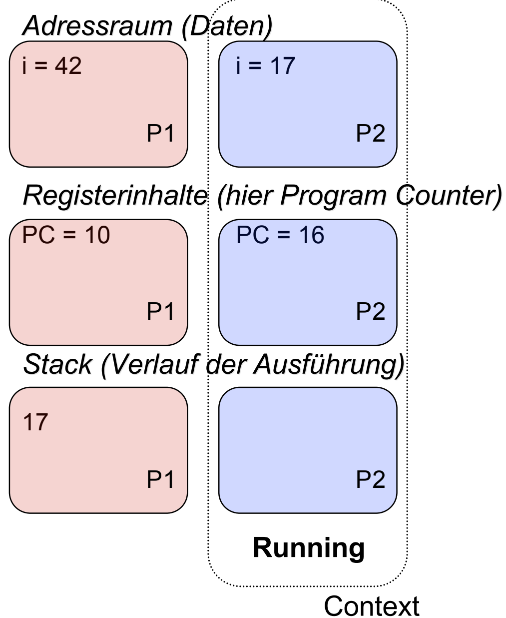
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf(“Wert von i ist %d\n“, i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```





Parallele Prozesse

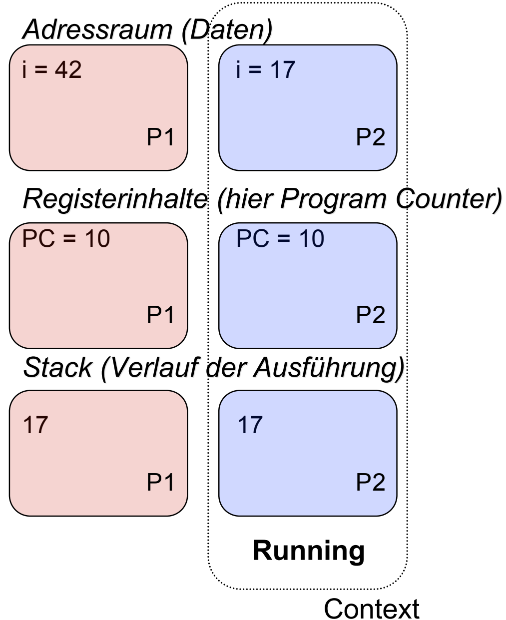
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



Parallele Prozesse

Automation Systems Group

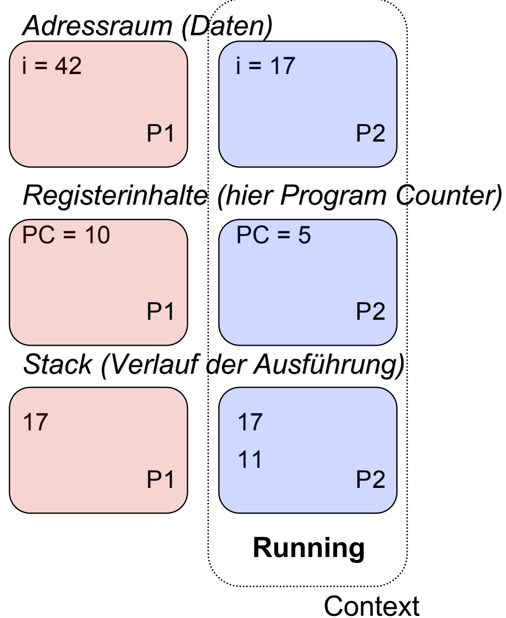
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



Parallele Prozesse

Automation Systems Group

```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

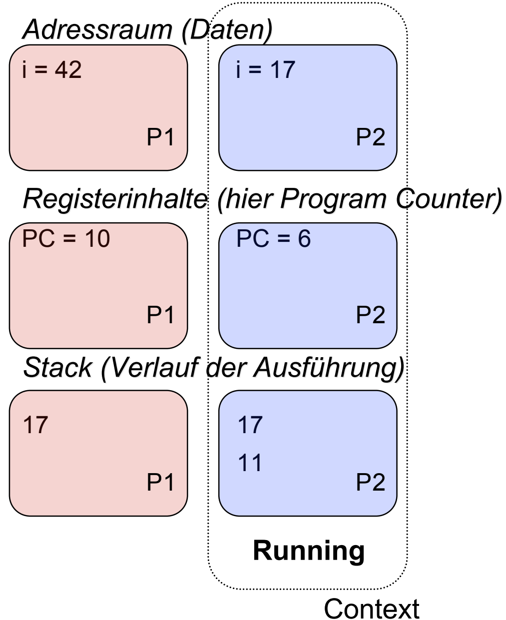


Parallele Prozesse

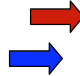
Automation Systems Group

```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

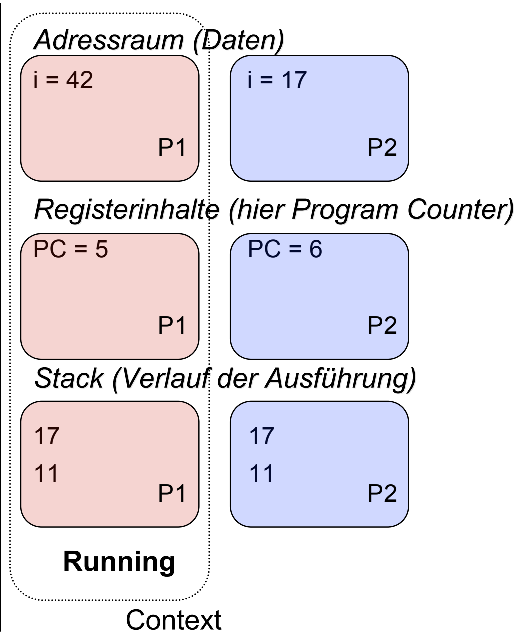
Wert von i ist 17



Parallele Prozesse

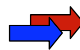


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



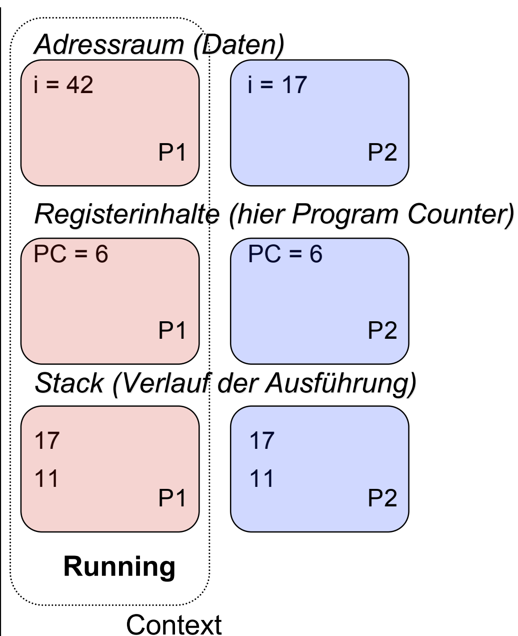
Context

Parallele Prozesse



```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Wert von i ist 42



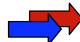
Context

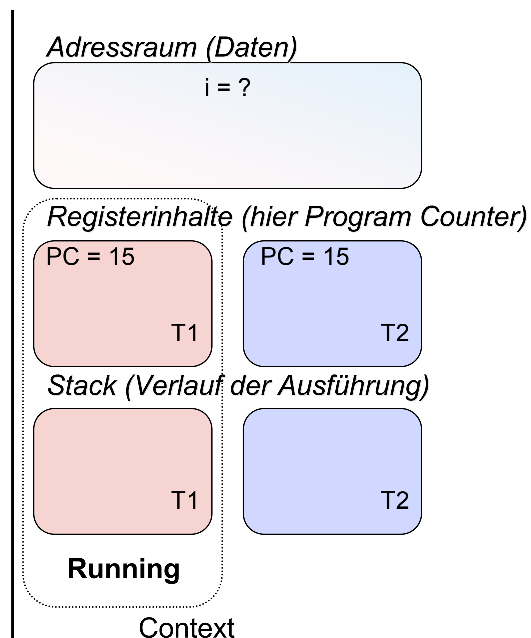
Threads

- Threads
  - mehrere, parallele Ausführungen von Instruktionen, aber mit den selben Ressourcen
  - sind wegen des gemeinsamen Adressraums
    - effizienter zu koordinieren
    - effizienter zu erzeugen
    - besser beim Wechsel zwischen Threads (context switch)
- Nützlich für Anwendungen mit mehreren Aufgaben
  - ein Prozess wartet auf Eingabe, ein anderer führt Berechnungen durch
  - z.B., Server Applikationen
- Natürlich kann man alles auch nur mit Prozessen realisieren
  - IPC (inter-process communication)

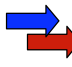
Threads


```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

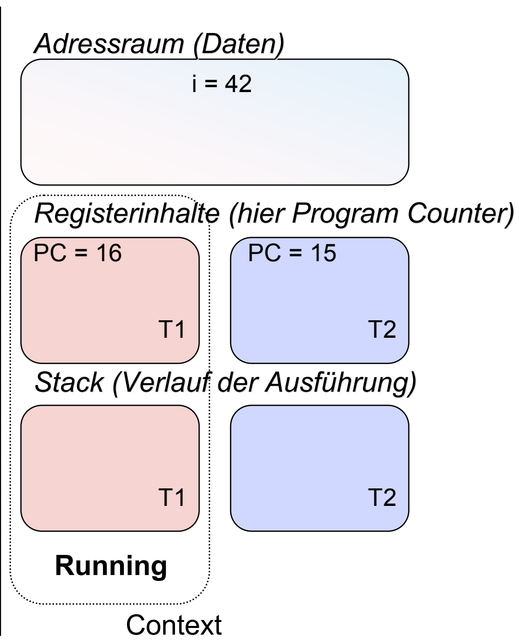




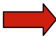
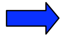
Threads



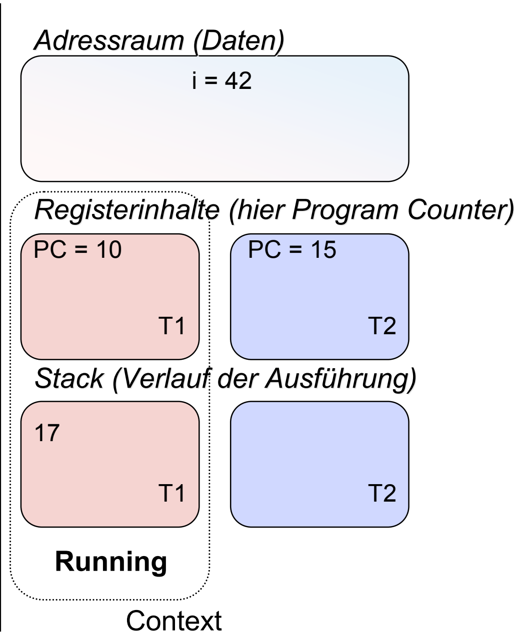
```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

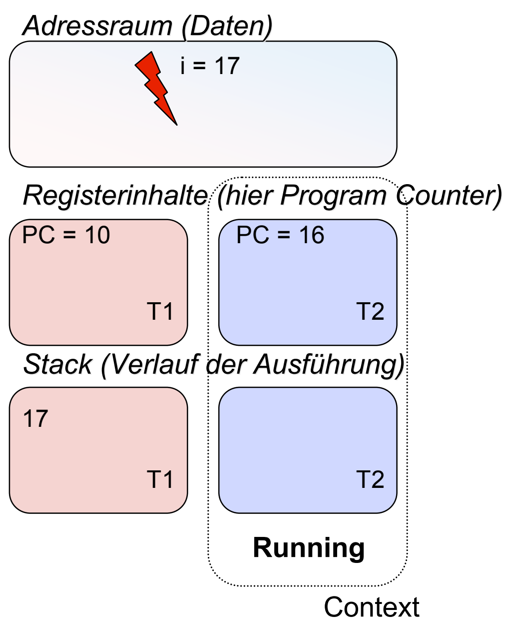



```
 1: int i;
 2:
 3: g()
 4: {
 5:    printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

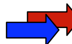


Threads

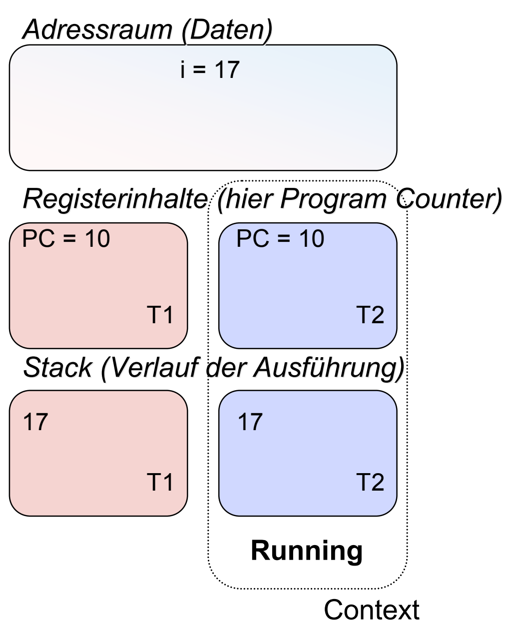
```
1: int i;
2:
3: g()
4: {
5:    printf("Wert von i ist %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

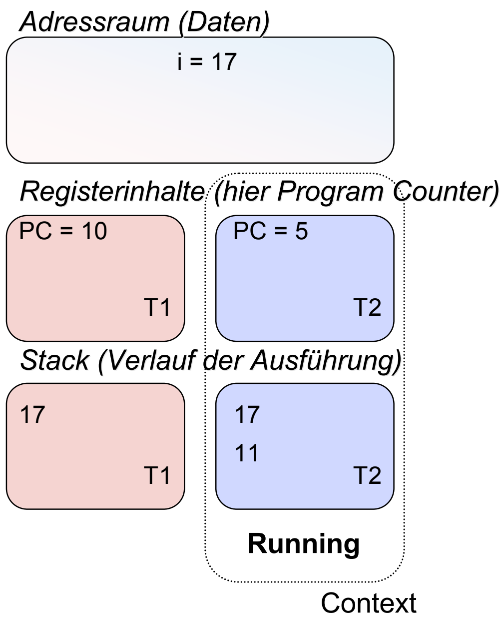


```
1: int i;
2:
3: g()
4: {
5:    printf("Wert von i ist %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```



Threads

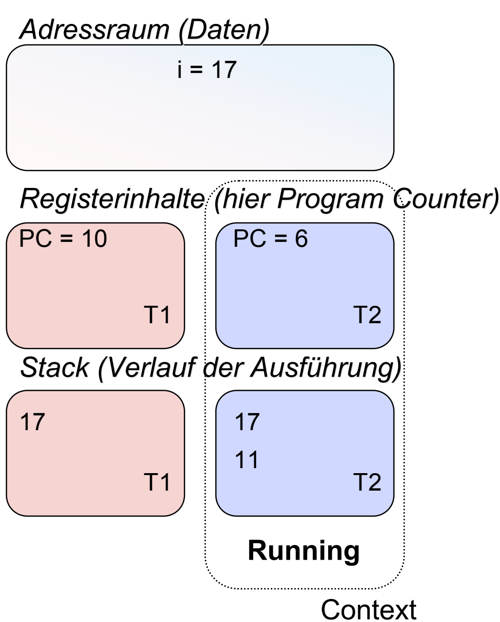
```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```



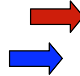
Threads

```
 1: int i;
 2:
 3: g()
 4: {
 5:     printf("Wert von i ist %d\n", i);
 6: }
 7:
 8: f()
 9: {
10:     g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:     i = get_input();
16:     f();
17:     return 0;
18: }
```

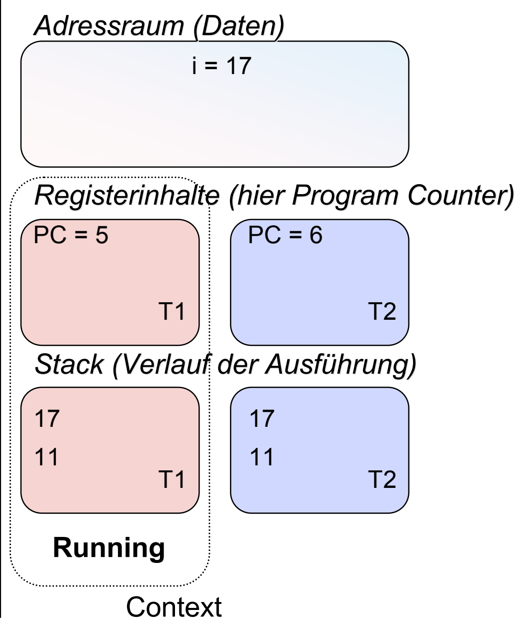
Wert von i ist 17



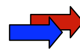
Threads



```
1: int i;
2:
3: g()
4: {
5:    printf("Wert von i ist %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

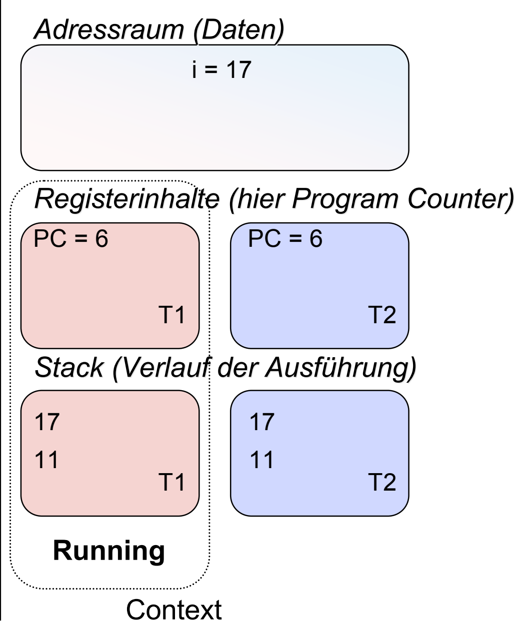


Threads



```
1: int i;
2:
3: g()
4: {
5:    printf("Wert von i ist %d\n", i);
6: }
7:
8: f()
9: {
10:    g();
11: }
12:
13: int main(int argc, char **argv)
14: {
15:    i = get_input();
16:    f();
17:    return 0;
18: }
```

Wert von i ist 17



Scheduling

- Viele Prozesse, aber nur eine CPU
  - Betriebssystem muss zwischen Prozessen wechseln
  - Betriebssystem braucht gelegentlich die CPU selbst
- Wechsel ist aufwendig
  - Zustand der Ausführung (Register) und Adressraum des alten Prozess muss gespeichert werden
  - neuer Prozess muss ausgewählt werden
  - Zustand des neuen Prozess und Adressraum muss geladen werden
- Auswahl des neuen Prozesses
  - ➢ Scheduling

Scheduling

- Ziele
  - Durchsatz (hoch)
  - Antwortzeit (niedrig)
  - CPU Auslastung
  - Fairness
- Unterscheidung
  - nicht-preemtive Verfahren
    - ein Prozess kann die CPU behalten, bis er fertig ist, oder freiwillig die Kontrolle abgibt
  - preemtive Verfahren
    - ein Prozess kann vom Betriebssystem gegen seinen Willen unterbrochen werden

Scheduling

- Algorithmen
  - first-come, first-serve
  - round-robin
  - priority scheduling
  - shortest job first
  - shortest remaining time
- Scheduling wird festgelegt durch
  - Angabe von den Zeitpunkten, zu denen Prozesse zu laufen beginnen (sich im Zustand RUNNING befinden)
- maximal ein Prozess kann sich im Zustand RUNNING befinden
- mehrere Prozesse können auf die CPU warten (READY)

Scheduling

- Praktisches Beispiel
- Angabe
  - Art des Scheduling Algorithmus
  - Angabe ob preemtives oder nicht-preemtives Scheduling vorliegt
  - Liste von Prozessen mit Information
    - wann ein Prozess in den Zustand READY wechselt
    - möglicherweise wie lange ein Prozess die CPU benötigt
    - möglicherweise Priorität
- Gesucht
  - Zeitpunkte, wann welche Prozesse zu laufen beginnen

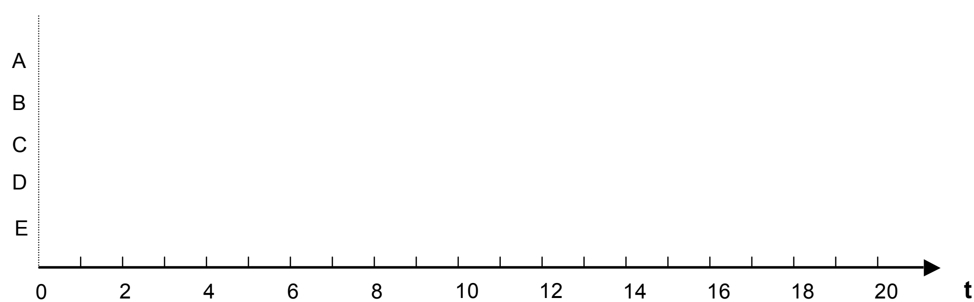
Scheduling

- Beispiel
  - nicht-preemtives priority scheduling
  - höherer Prioritätswert bedeutet höhere Priorität

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3

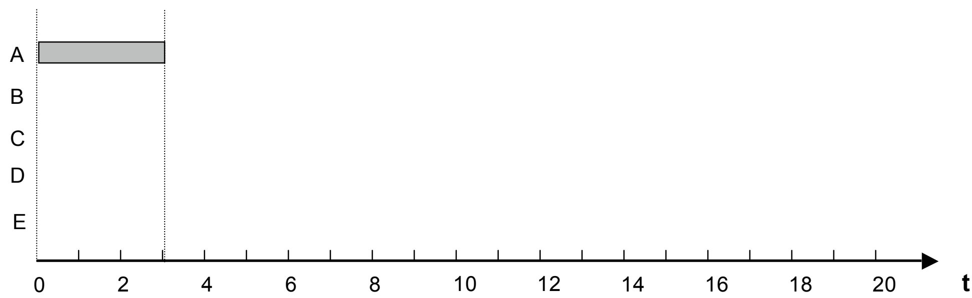
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



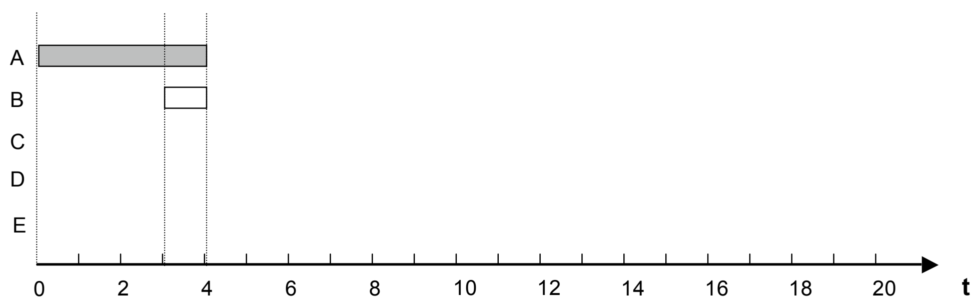
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



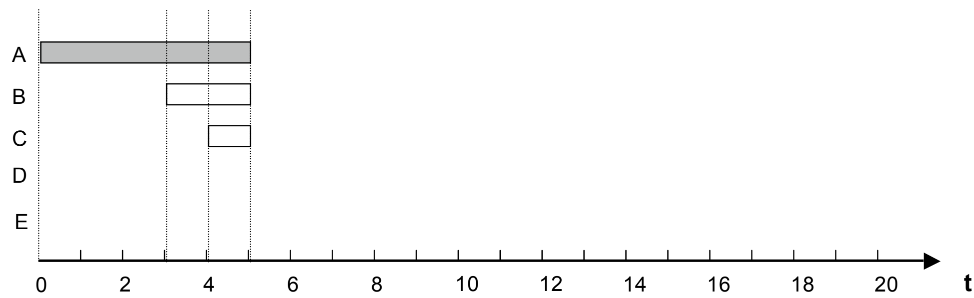
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



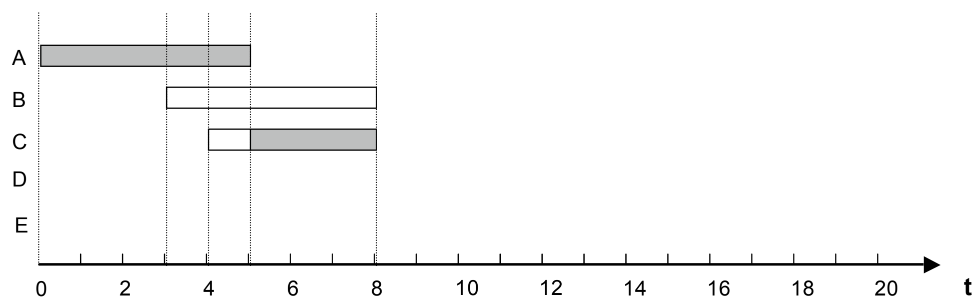
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



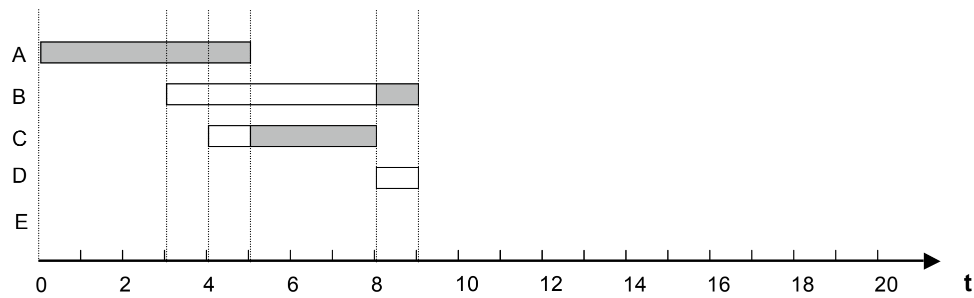
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



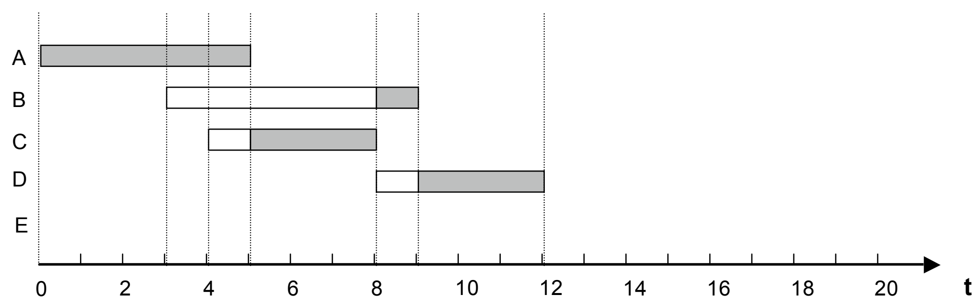
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

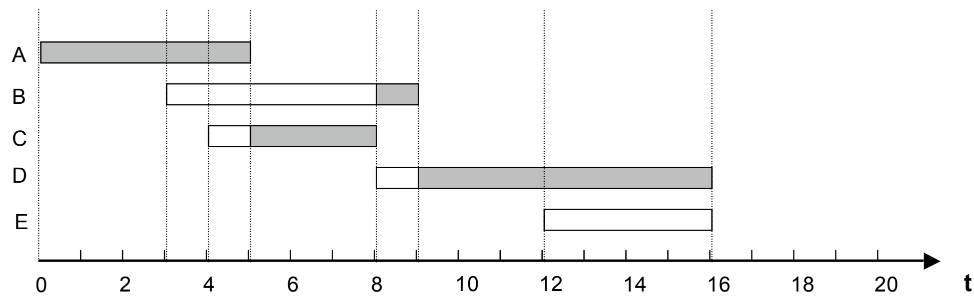
Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

Automation Systems Group

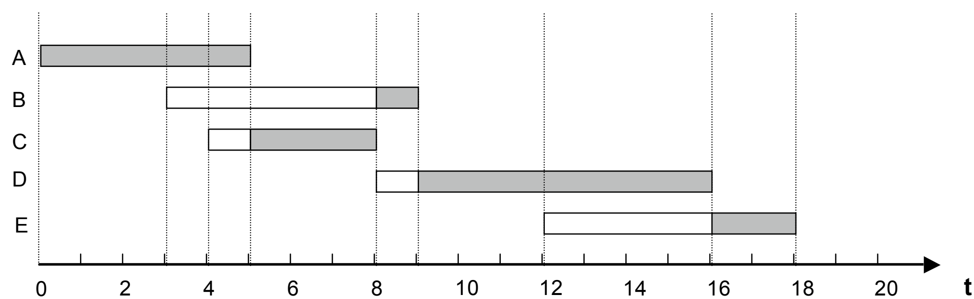
Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



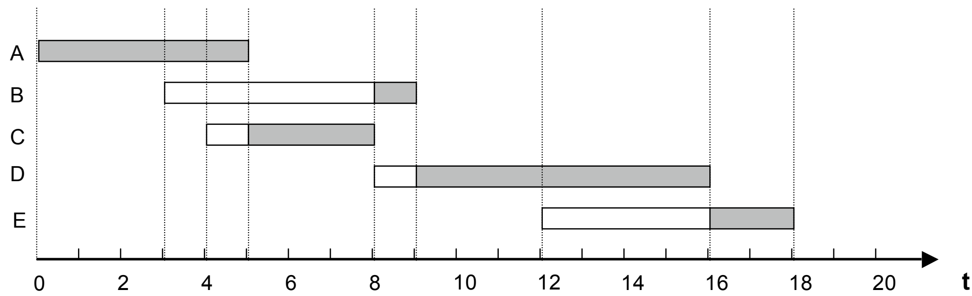
Scheduling

Automation Systems Group

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling



Prozess	A	C	B	D	E
Zeitpunkt (RUNNING)	0	5	8	9	16

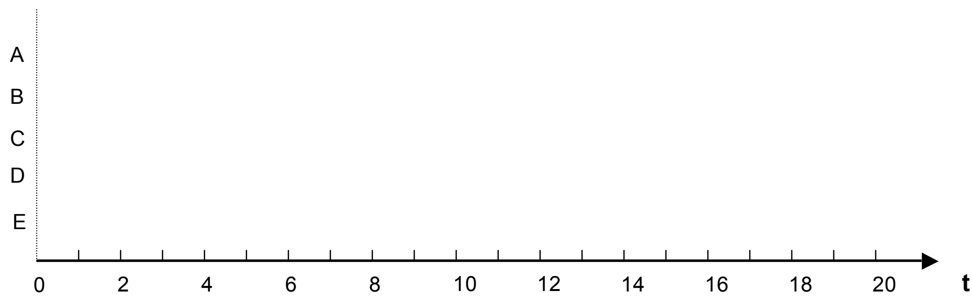
Scheduling

- Beispiel
  - *preemtives* priority scheduling
  - höherer Prioritätswert bedeutet höhere Priorität

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3

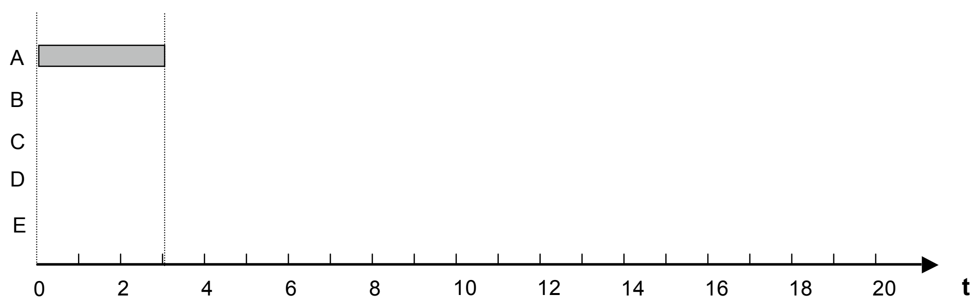
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



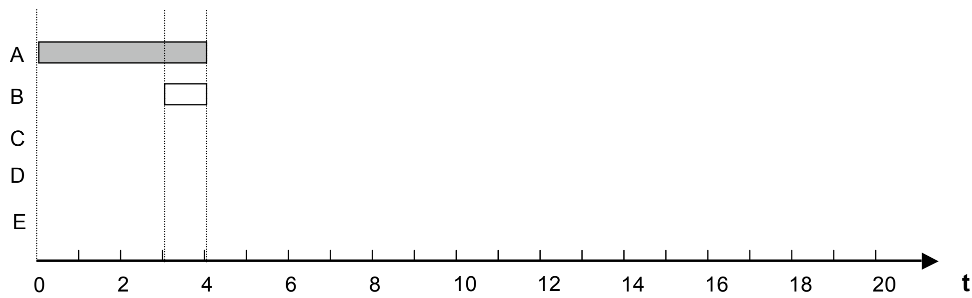
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

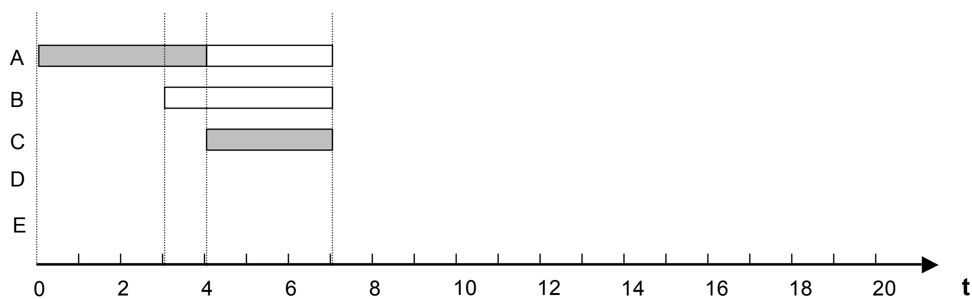
Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling

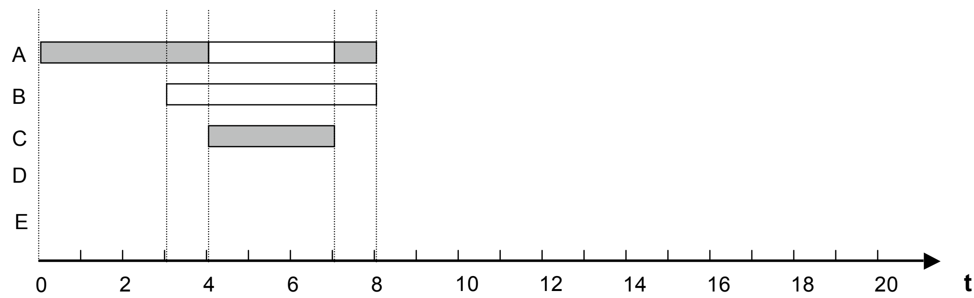
Automation Systems Group

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



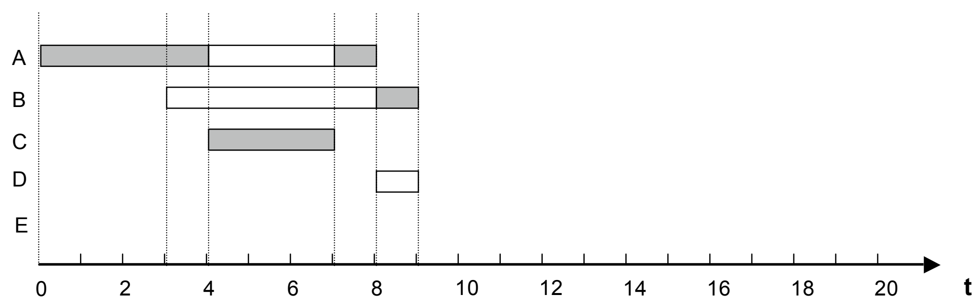
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



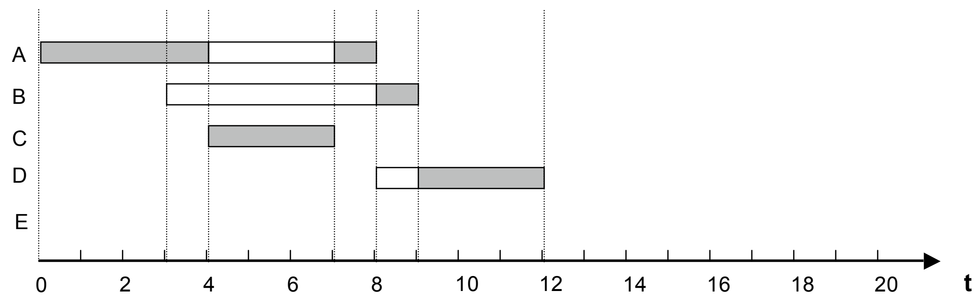
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



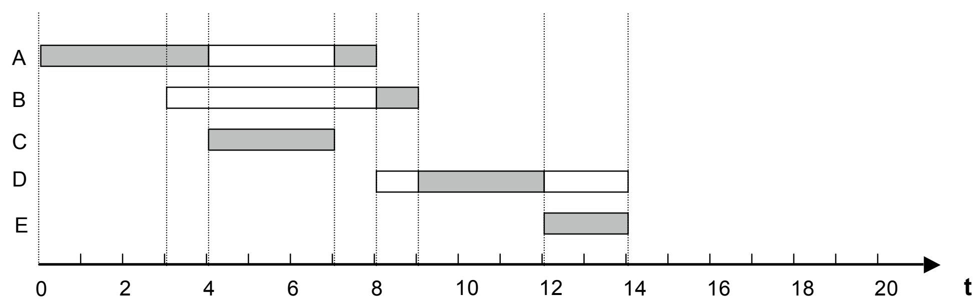
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



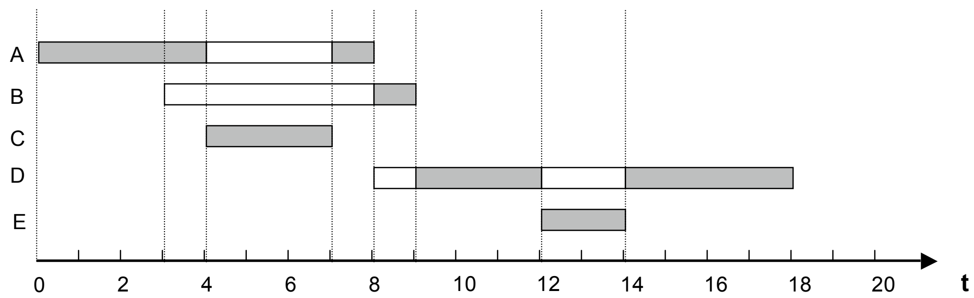
Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3

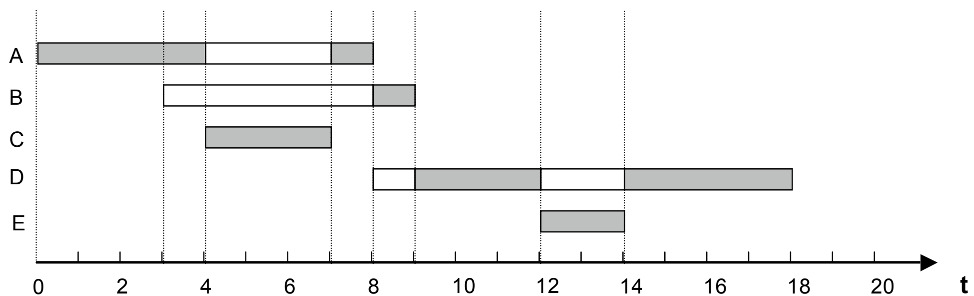


Scheduling

Prozess	Startzeitpunkt	Laufzeit	Priorität
A	0	5	2
B	3	1	1
C	4	3	4
D	8	7	0
E	12	2	3



Scheduling



Prozess	A	C	A	B	D	E	D
Zeitpunkt (RUNNING)	0	4	7	8	9	12	14

Interprozess-Kommunikation

- Aufgabe wird in mehrere Teile zerlegt
- Jeder Teil wird von einem Prozess (oder Thread) bearbeitet
- Prozesse müssen untereinander Nachrichten austauschen

➢ Interprozess-Kommunikation (IPC)

- Arbeitsschritte müssen in einer bestimmten Reihenfolge ausgeführt werden

➢ Synchronisation

Synchronisation

- Klassisches Problem
  - Zugriff auf eine gemeinsame Ressource (z.B., Drucker)
  - diese Ressource kann aber auch eine gemeinsame Variable sein
    - erinnern wir uns an das Problem der globalen Variable bei den Threads
- Oft ist notwendig, dass ein Prozess eine Reihe von Operationen auf einer Ressource ausführt, ohne dass irgendein anderer Prozess auf diese Ressource zugreift
- Die Reihe von Operationen, welche nicht unterbrochen werden dürfen, nennt man oft „critical section“
- Die Tatsache, dass sich nur ein Prozess in einer critical section befinden darf, wird „mutual exclusion“ genannt

Interprozess-Kommunikation

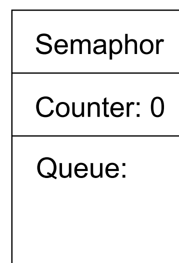
- Synchrone Methoden
  - Empfänger muss Nachricht aktiv abholen
  - Beispiel
    - Message Passing
    - Semaphore
  - bekannte Analogie ist ein Briefkasten
- Asynchrone Methoden
  - Empfänger wird durch Nachricht unterbrochen
  - Beispiel
    - Signale
    - Interrupts
  - bekannte Analogie ist ein Eilbote

Synchrone IPC

- Semaphore
  - dienen zur Synchronisation
  - von Dijkstra eingeführt
- Aufbau
  - Counter (Zähler)
  - Warteschlange für Prozesse
- Operationen

  P Operation

  V Operation



Synchrone IPC

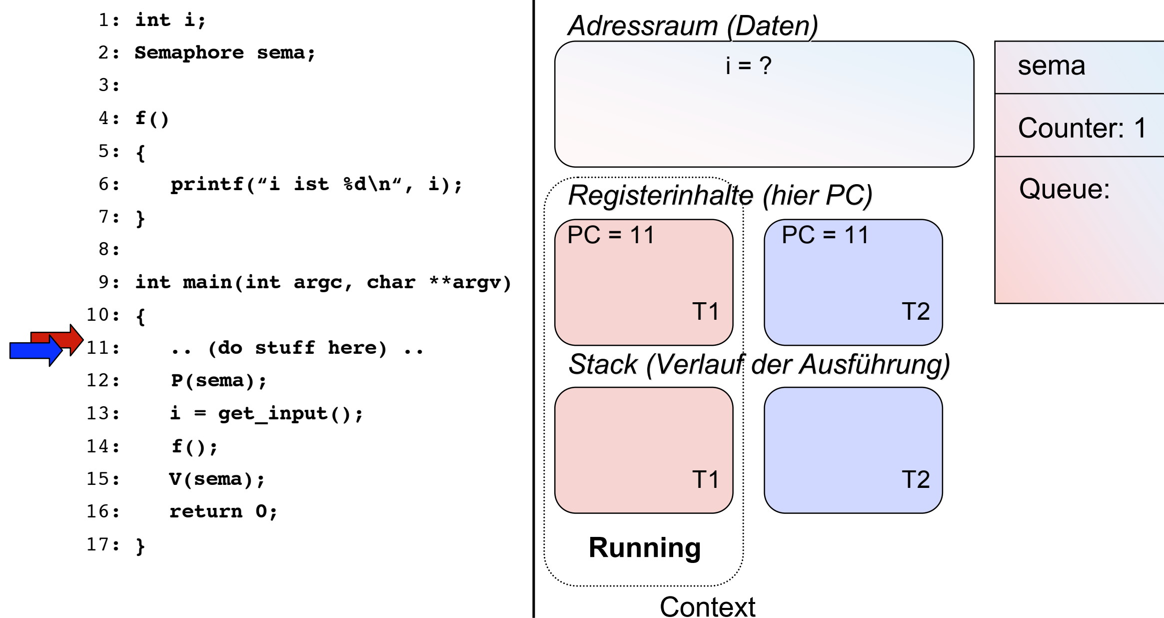
- **P** Operation
  - wenn Counter > 0
    - erniedrige Counter um 1
  - wenn Counter <= 0
    - erniedrige Counter um 1
    - blockiere den aufrufenden Prozess (oder Thread) und trage ihn in die Queue ein
- **V** Operation
  - wenn Counter >= 0
    - erhöhe Counter um 1
  - wenn Counter < 0
    - erhöhe Counter um 1
    - de-blockiere *einen* Prozess, der in der Queue wartet

Synchrone IPC

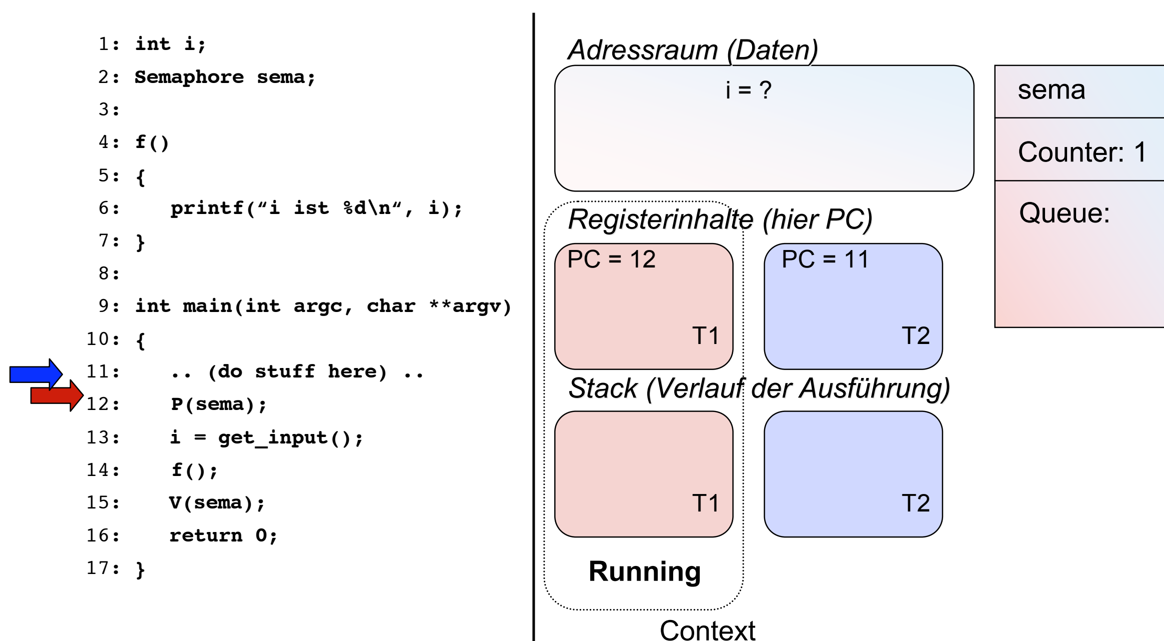
Automation Systems Group

- Idee
  - bevor ein Prozess eine critical section betritt, muss eine **P** Operation ausgeführt werden
  - am Ende der critical section wird eine **V** Operation ausgeführt
- Counter
  - wenn der Counter > 0 ist, gibt er an, wie viele Prozesse ohne zu blockieren in die critical section dürfen (üblicherweise 1, kann aber für manche Ressourcen durchaus mehr sein)
  - wenn der Counter < 0 ist, gibt er an, wie viele Prozesse in der Queue warten

Threads - Revisited

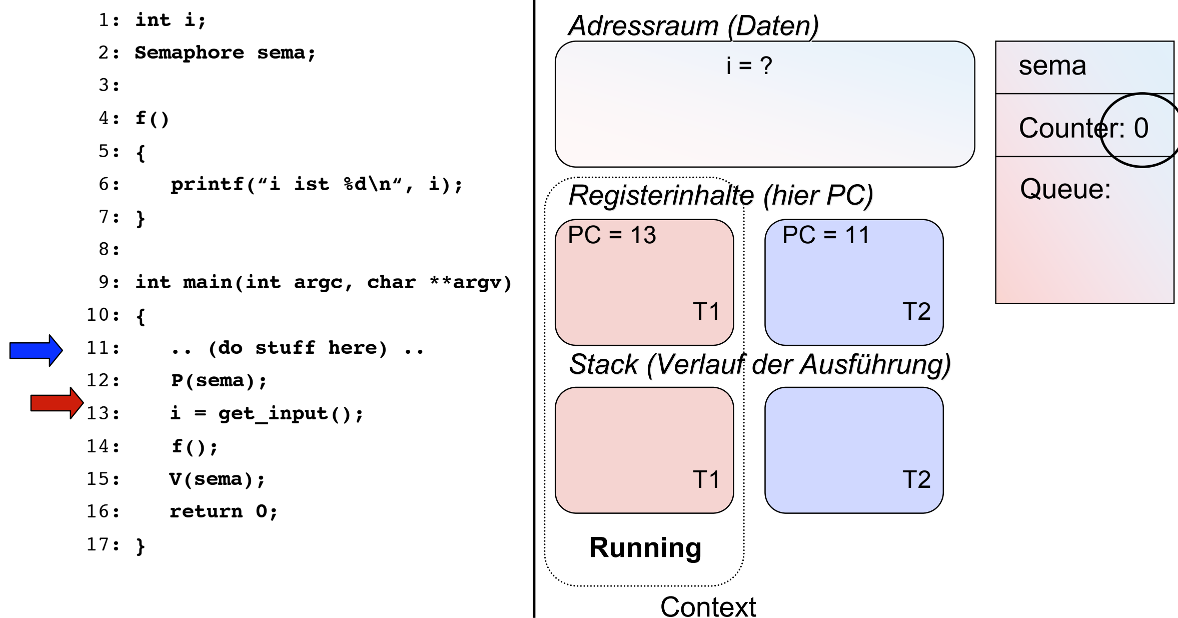


Threads - Revisited



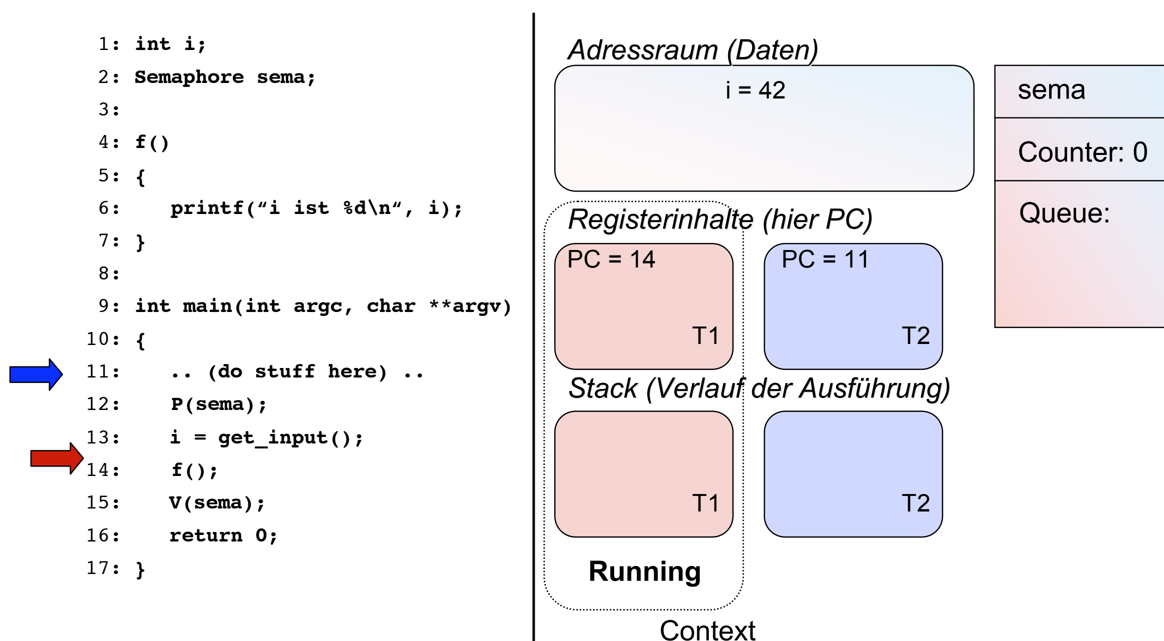
Threads - Revisited

Automation Systems Group



Threads - Revisited

Automation Systems Group



Threads - Revisited


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```

Adressraum (Daten)

i = 42

Registerinhalte (hier PC)

PC = 14 T1

PC = 11 T2

Stack (Verlauf der Ausführung)

T1

T2

Running

Context

sema
Counter: 0
Queue:

Threads - Revisited


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```

Adressraum (Daten)

i = 42

Registerinhalte (hier PC)

PC = 14 T1

PC = 12 T2

Stack (Verlauf der Ausführung)

T1

T2

Running

Context

sema
Counter: 0
Queue:

Threads - Revisited


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```

Adressraum (Daten)

i = 42

Registerinhalte (hier PC)

PC = 14 T1

PC = 13 T2

Stack (Verlauf der Ausführung)

T1

T2

Blocked

Context

sema

Counter: -1

Queue:
T2

Threads - Revisited

Automation Systems Group


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```


Adressraum (Daten)

i = 42

Registerinhalte (hier PC)

PC = 14 T1

PC = 13 T2

Stack (Verlauf der Ausführung)

T1

T2

Running

Blocked

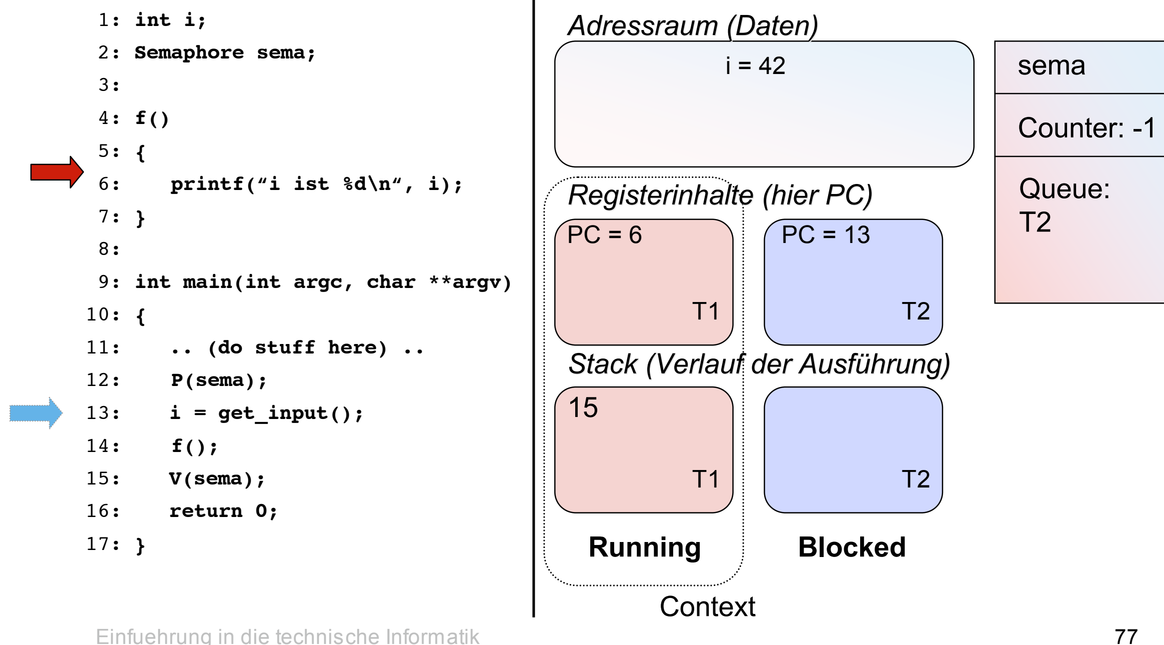
Context

sema

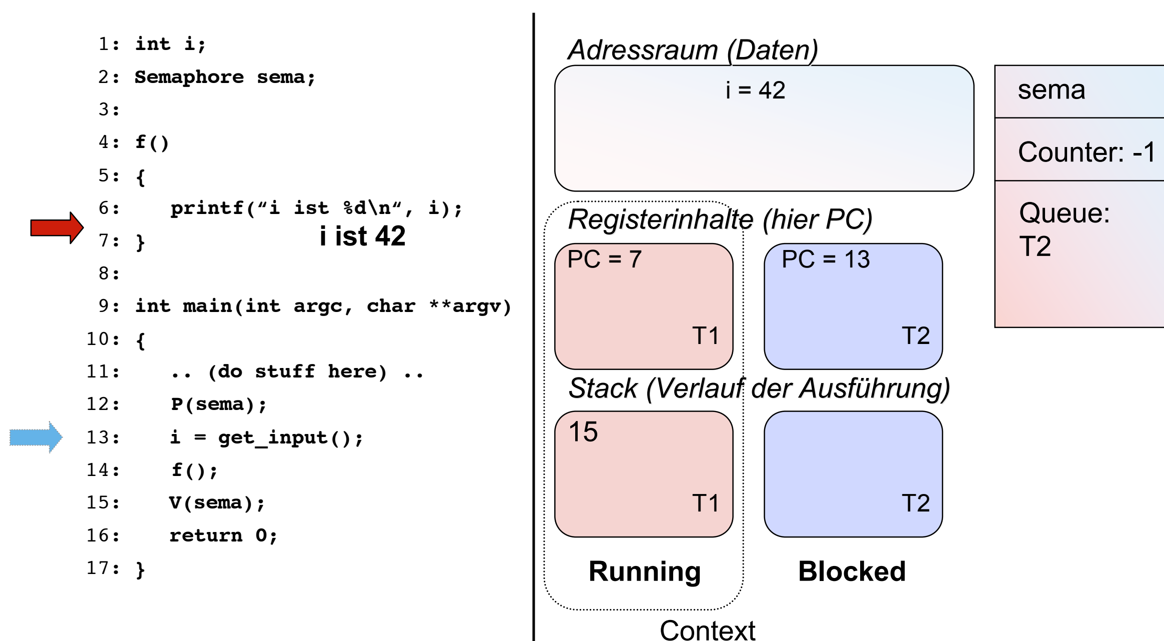
Counter: -1

Queue:
T2

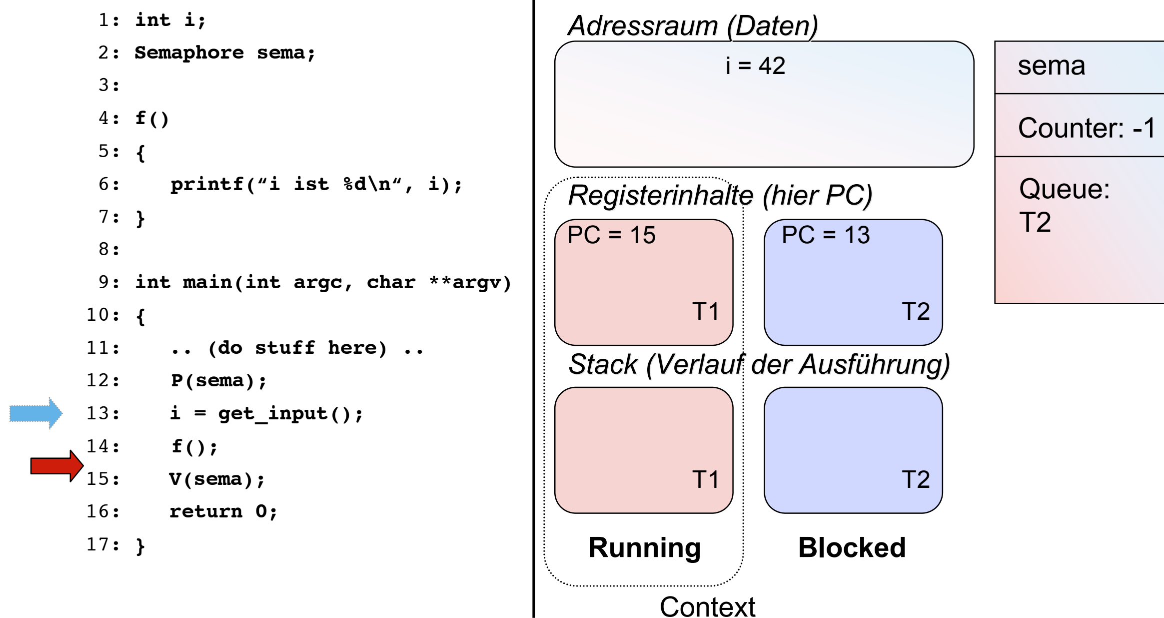
Threads - Revisited



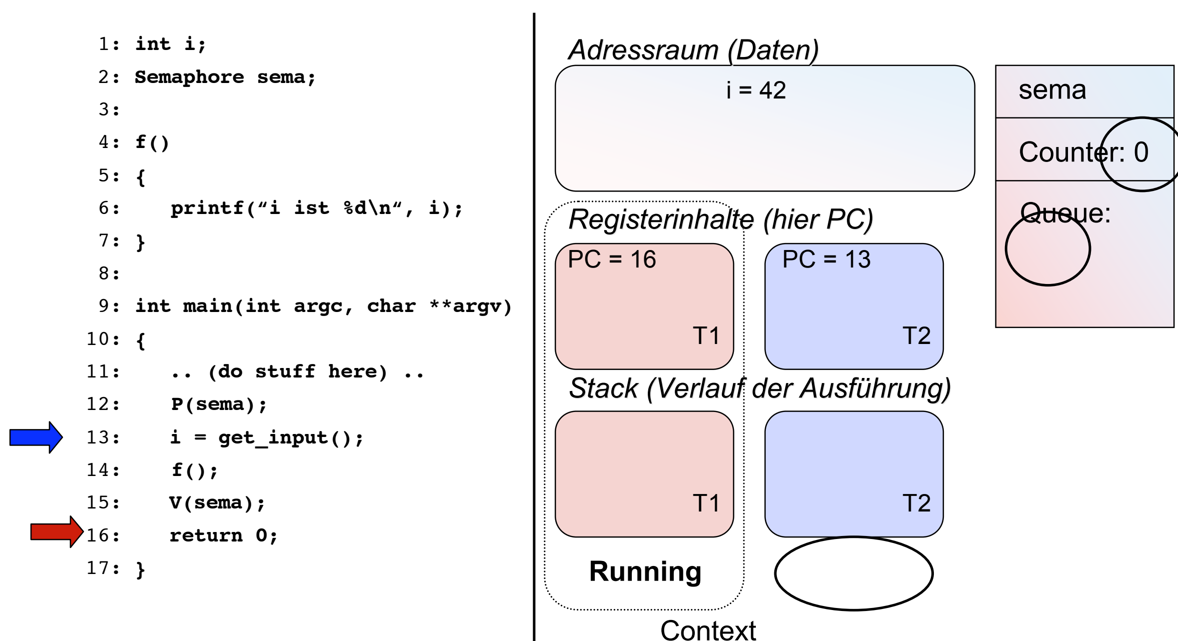
Threads - Revisited



Threads - Revisited

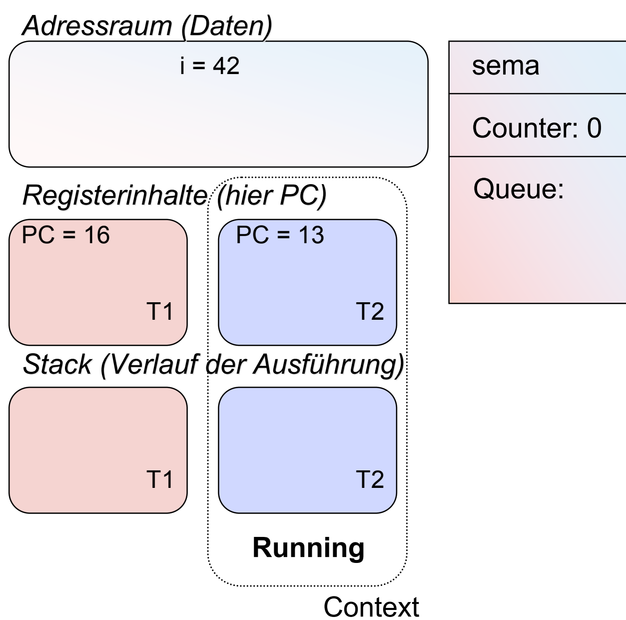


Threads - Revisited



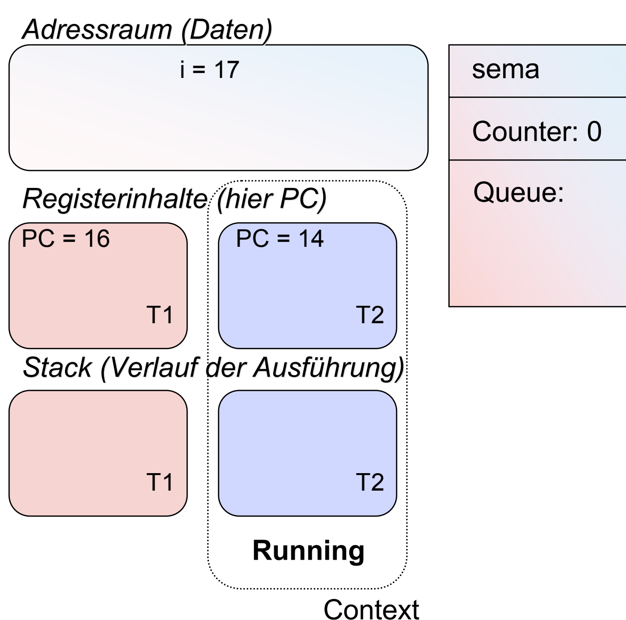
Threads - Revisited


```
1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:    printf("i ist %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```



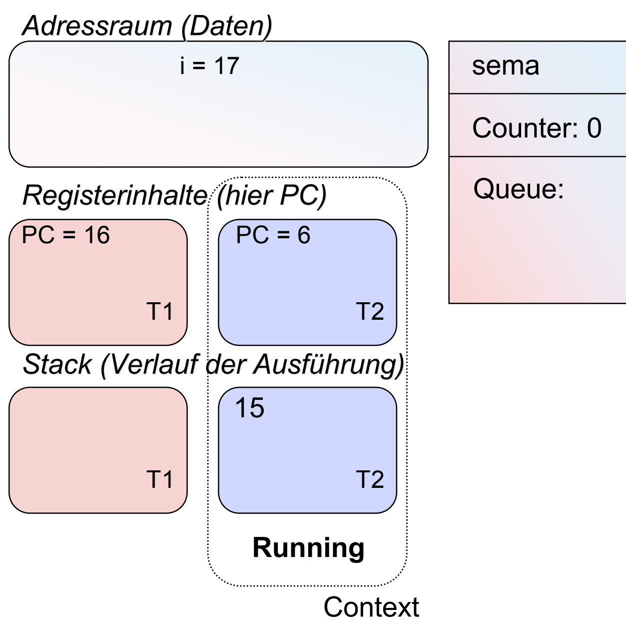
Threads - Revisited


```
1: int i;
2: Semaphore sema;
3:
4: f()
5: {
6:    printf("i ist %d\n", i);
7: }
8:
9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```



Threads - Revisited

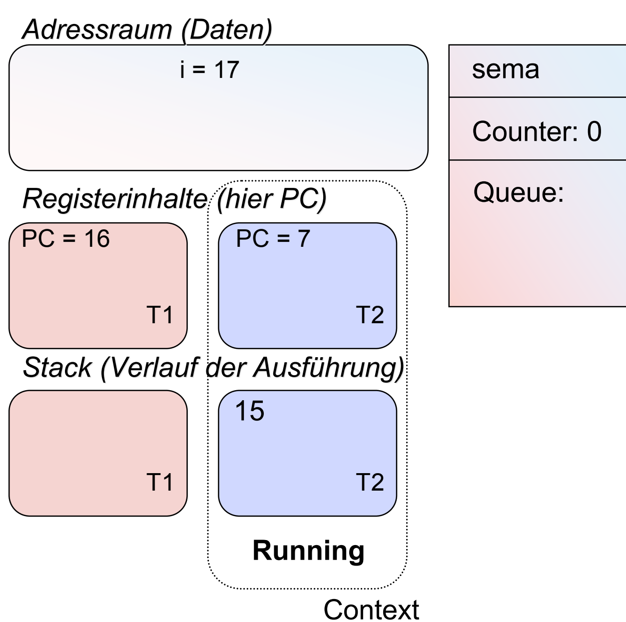

```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```



Threads - Revisited


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```

i ist 17



Threads - Revisited


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```

Adressraum (Daten)

i = 17

Registerinhalte (hier PC)

PC = 16 T1

PC = 15 T2

Stack (Verlauf der Ausführung)

T1

T2

Running

Context

sema

Counter: 0

Queue:

Threads - Revisited


```
 1: int i;
 2: Semaphore sema;
 3:
 4: f()
 5: {
 6:    printf("i ist %d\n", i);
 7: }
 8:
 9: int main(int argc, char **argv)
10: {
11:    .. (do stuff here) ..
12:    P(sema);
13:    i = get_input();
14:    f();
15:    V(sema);
16:    return 0;
17: }
```

Adressraum (Daten)

i = 17

Registerinhalte (hier PC)

PC = 16 T1

PC = 16 T2

Stack (Verlauf der Ausführung)

T1

T2

Running

Context

sema

Counter: 1

Queue:

Deadlock

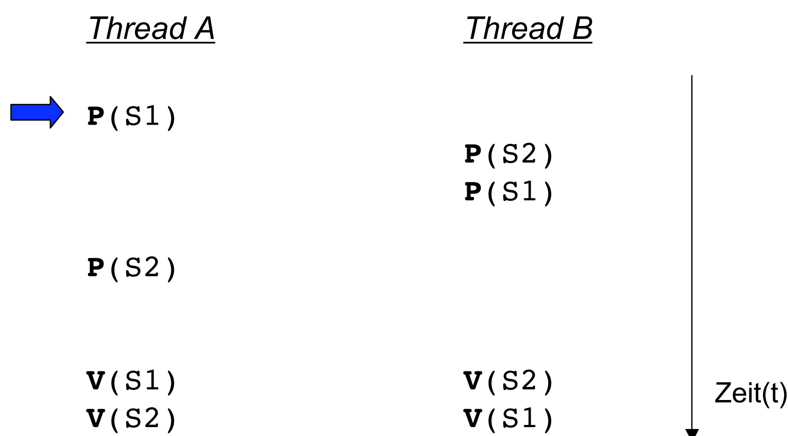
- Now, synchronisation is all good and fun ….

  … aber, Deadlocks können auftreten!

- Deadlock
  – Eine Gruppe von Prozessen wartet jeweils auf die Aktion eines anderen Prozesses in dieser Gruppe
  – Daher kommt es zum Stillstand, kein Fortschritt wird mehr erzielt

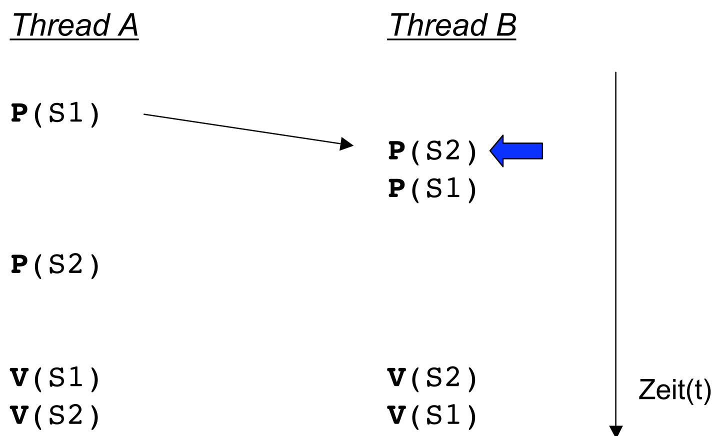
Deadlock

- Beispiel mit 2 Threads und 2 Semaphoren, die jeweils eine globale Variable „schützen“



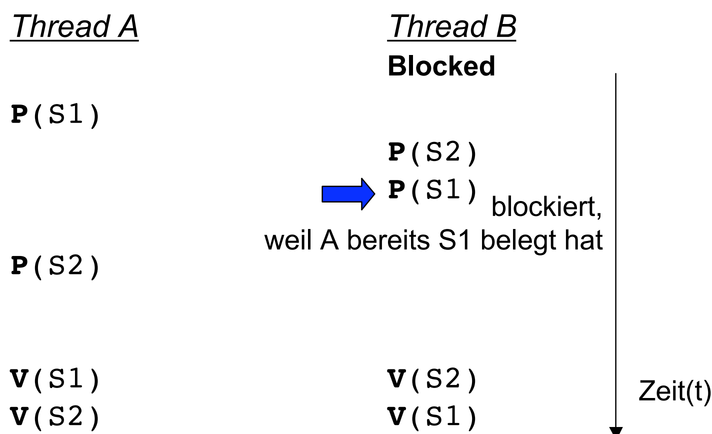
Deadlock

- Beispiel mit 2 Threads und 2 Semaphoren, die jeweils eine globale Variable „schützen“



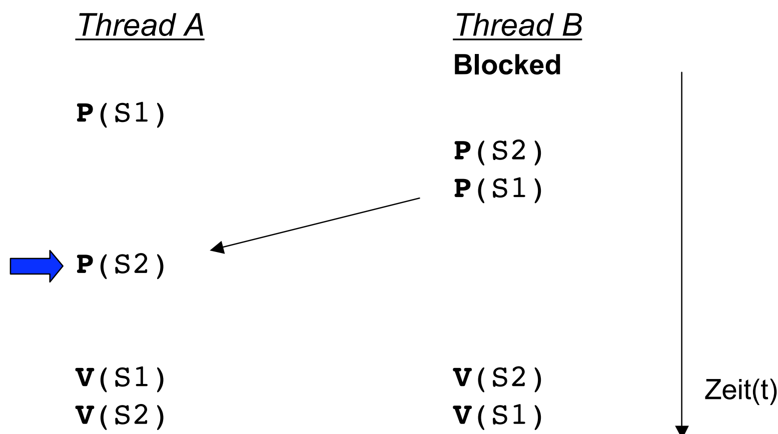
Deadlock

- Beispiel mit 2 Threads und 2 Semaphoren, die jeweils eine globale Variable „schützen“



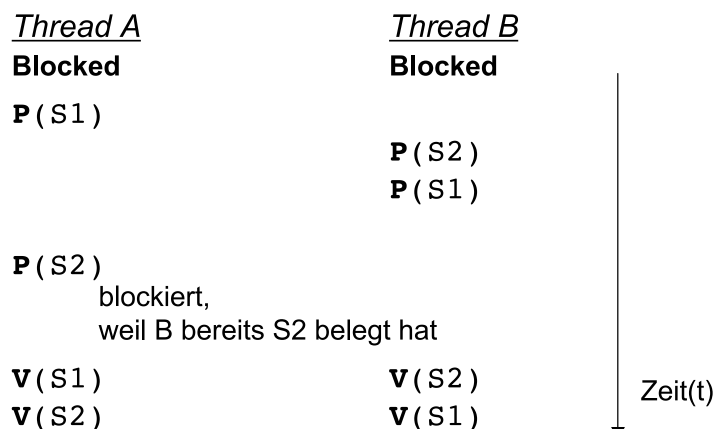
Deadlock

- Beispiel mit 2 Threads und 2 Semaphoren, die jeweils eine globale Variable „schützen“



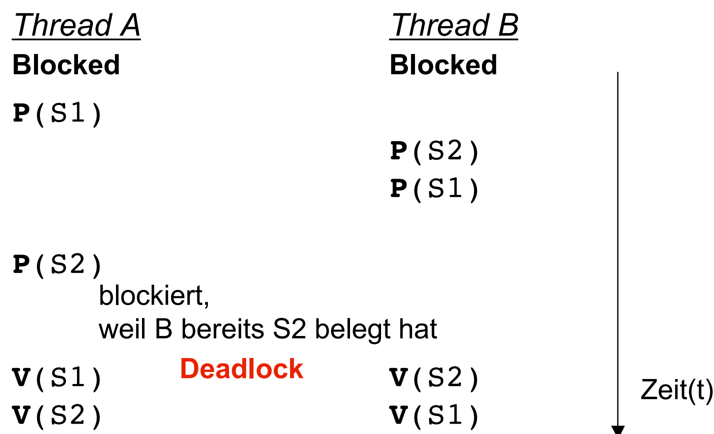
Deadlock

- Beispiel mit 2 Threads und 2 Semaphoren, die jeweils eine globale Variable „schützen“



Deadlock

- Beispiel mit 2 Threads und 2 Semaphoren, die jeweils eine globale Variable „schützen“



Deadlock

- Notwendige Bedingungen

1) Mutual Exclusion
   - Der Zugriff auf Ressourcen ist exklusiv
2) Hold and Wait
   - Prozesse können Ressourcen anfordern, obwohl sie schon welche besitzen
3) No preemption
   - Ressourcen werden nur durch Prozesse freigegeben
4) Circular Wait
   - Zumindest zwei Prozesse warten aufeinander

Deadlock

- Maßnahmen zur Verhinderung von Deadlocks

1) Deadlock Avoidance
   - Prozesse werden überwacht und alle Anforderungen, die möglicherweise zu einem Deadlock führen, werden nicht erlaubt oder verzögert

2) Deadlock Prevention
   - eine (oder mehrere) der vier notwendigen Bedingungen für einen Deadlock werden im System verboten

3) Deadlock Detection
   - Deadlock wird erkannt und (einige) beteiligte Prozesse beendet

4) Problem ignorieren

Zusammenfassung

- Betriebssysteme verwalten
  - *Prozesse*
  - Speicher
  - Dateisystem
  - Eingabe und Ausgabe

- Prozessmanagement
  - Prozessmanagement Datenstrukturen
  - Zustandsübergänge
  - Prozess
    - Ressourcen und thread(s) of execution
  - Thread
    - Registersatz und thread-spezifischen Daten (lokale Variable, Stack)
  - Scheduling

Zusammenfassung

- Interprozess-Kommunikation (IPC)
  - Datenaustausch zwischen Prozessen (oder Threads)
  - Synchronisation für mutual exclusion in critical section
- Synchrone IPC
  - Semaphore, Messages
- Asynchrone IPC
  - Signale, Interrupts
- Deadlock
  - schwerwiegendes Probleme bei Synchronisationsaufgaben