

Einführung in die Betriebssysteme

Martin Spörl

Prozesse

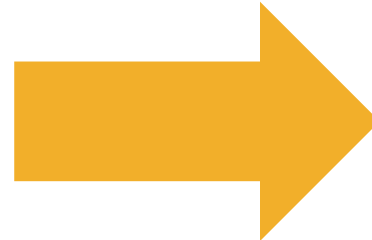
Grundlagen

- Prozess = Ausführung eines Programms auf einer CPU mitsamt der entsprechenden Umgebung

- Programmzähler (PC)
- Daten
- Code
- Alle Register (Daten, Status, Adressen)

„Prozessumgebung“

- Prozess kann anderen Prozess erzeugen (Parent Prozess – Child Prozess)
- in einem Prozessor kann immer nur ein Prozess gleichzeitig Aktiv sein
- Prozesse können voneinander unterschieden werden



Anforderungen an OS

- um „gleichzeitig“ ausführen zu können, muss OS Programm nur partiell ausführen können
- OS muss Prozess Resource zuteilen
- OS muss Interprozesskommunikation bereitstellen
- OS muss Prozess eindeutig erkennen können („Prozess ID“ – PID)

Prozessor – Instructions per Second

Zeitachse MIPS-Entwicklung

CPU	MIPS	Taktfrequenz	Jahr
Intel 8080	0,4	2 MHz	1974
Z80	0,625	2,5 MHz	1974
Motorola 68000	1	8 MHz	1979
Motorola 68020	4	20 MHz	1984
ARM2	4	8 MHz	1986
Motorola 68030	11	33 MHz	1987
ARM3	12	25 MHz	1989
Motorola 68040	44	40 MHz	1990
Intel 486DX	54	66 MHz	1992
DEC Alpha 21064 EV4	300	150 MHz ^[1]	1992
Motorola 68060	88	66 MHz	1994
ARM 7500FE	35,9	40 MHz	1996
Atmel AVR	10	10 MHz	1996 ^[2]
PowerPC G3	671	366 MHz	1997
Zilog eZ80	80	50 MHz	1998
ARM10	400	300 MHz	1999
Pentium 3	1.354	500 MHz	1999
Athlon FX-57	8.400	2,8 GHz	2005
Athlon FX60	18.938	2,6 GHz	2006
Xeon Harpertown	93.608	3 GHz	2007
ARM Cortex-A15	35.000	2,5 GHz	2010
AMD Phenom II X6 1100T	78.440	3,3 GHz	2010
AMD FX-8150	108.890	3,6 GHz	2011
Intel Core i7 2600K	128.300	3,4 GHz	2011
Intel Core i7 5960X	336.000	3,5 GHz	2014

- MIPS = Million instructions per second
- 8051 (original – 12 Mhz) = ~ 1 MIPS
 - „neuere“ Modelle laufen auch mit 100 Mhz

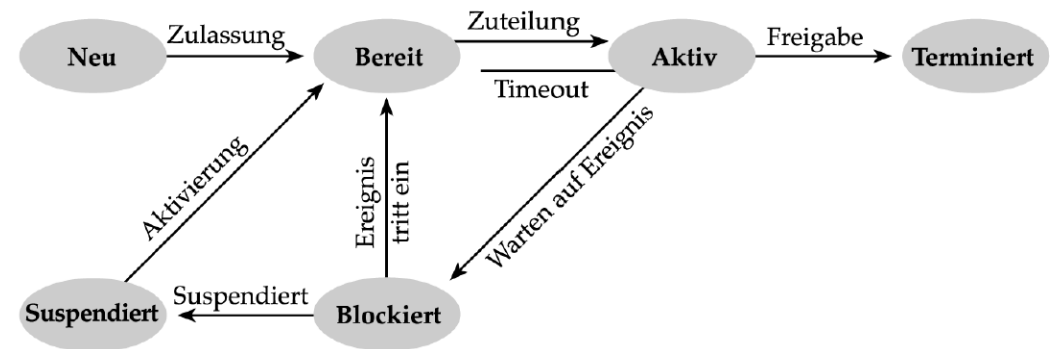
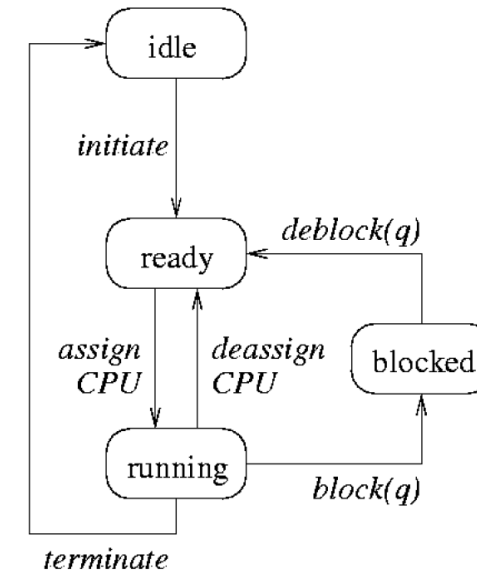
Prozesszustände I

5 Zustände

- „ready“ – Prozess kann ausgeführt werden, aber kein Prozessor frei
- „running“ – Prozess wird ausgeführt
- „blocked“ – wartet auf externes Ereignis (q)
- „idle“ – Prozess wurde gerade erzeugt oder ist terminiert (oft in „new“ und „terminated“ geteilt)
- „suspended“ – Prozess wurde aus dem Speicher auf die Festplatte ausgelagert



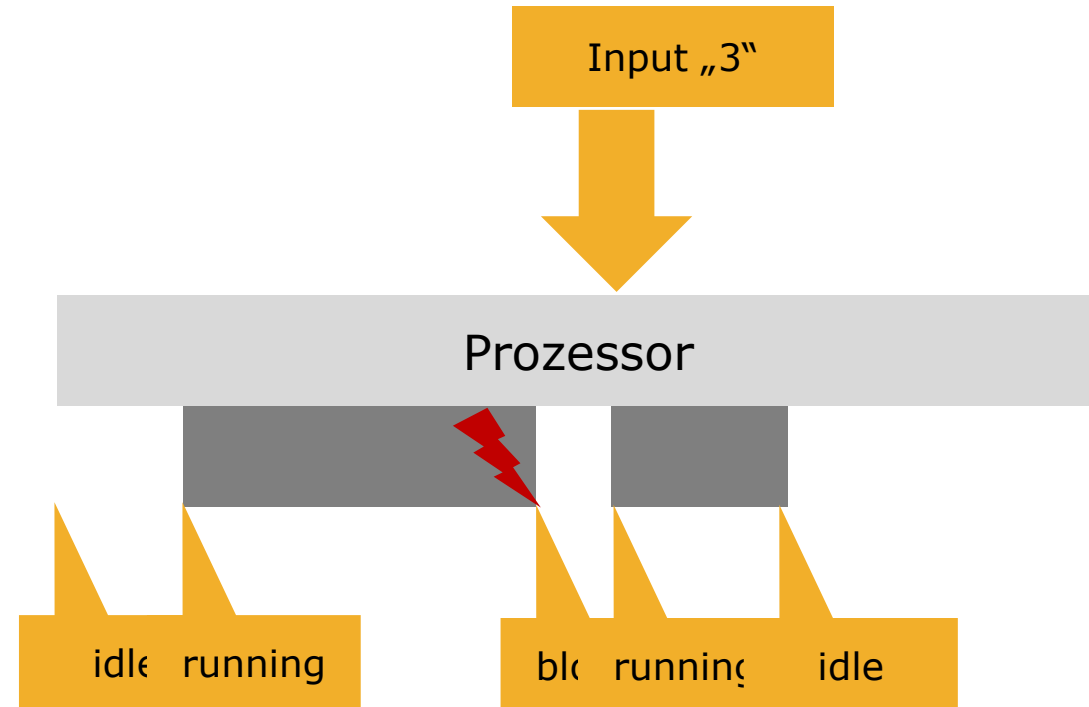
Scheduler steuert die Übergänge



Prozesszustände II – Beispiel

Wie sehen die Zustände in der Praxis aus?

```
int main(void){  
    int a = 1; ←  
    int b = 3;  
    int erg = 0;  
    erg = a * b + 4;  
    a = getc(stdin) - '0';  
    erg = a * b + 4;  
    return 0;  
}
```



Prozesszustände III – Praxis

```
pi@raspberrypi:~ $ ps ux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
pi        1241  0.0  0.3   5104  3320 ?        Ss   Aug28   0:00 /lib/systemd/systemd
pi        1245  0.0  0.1   7040  1456 ?        S    Aug28   0:00 (sd-pam)
pi        1251  0.0  0.4   6500  4424 tty1    S+   Aug28   0:00 -bash
pi        1482  0.0  0.0   1908   100 ?        S    Aug28   0:00 /bin/sh /usr/bin/sta
pi        1483  0.0  0.2   5844  2004 ?        S    Aug28   0:00 /usr/bin/xprop -root
pi        1505  0.0  0.6  30232  6016 ?       Ssl  Aug28   0:00 /usr/lib/menu-cache/
pi        1543  0.0  0.7  60744  7304 ?       Sl   Aug28   0:00 /usr/lib/gvfs/gvfsd-
pi       17969  0.0  0.3  12340  3076 ?        S    22:10   0:00 sshd: pi@pts/0
pi       17971  0.4  0.4   6488  4428 pts/0    Ss   22:10   0:00 -bash
pi       18005  0.0  0.2   4740  2108 pts/0    R+   22:11   0:00 ps ux
```

„ps ux“

„man ps“

PROCESS STATE CODES

Here are the different values that the `s`, `stat` and `state` output specifiers (header "STAT" or "S") will display to describe the state of a process:

D	uninterruptible sleep (usually IO)
R	running or runnable (on run queue)
S	interruptible sleep (waiting for an event to complete)
T	stopped, either by a job control signal or because it is being traced
W	paging (not valid since the 2.6.xx kernel)
X	dead (should never be seen)
Z	defunct ("zombie") process, terminated but not reaped by its parent

For BSD formats and when the `stat` keyword is used, additional characters may be displayed:

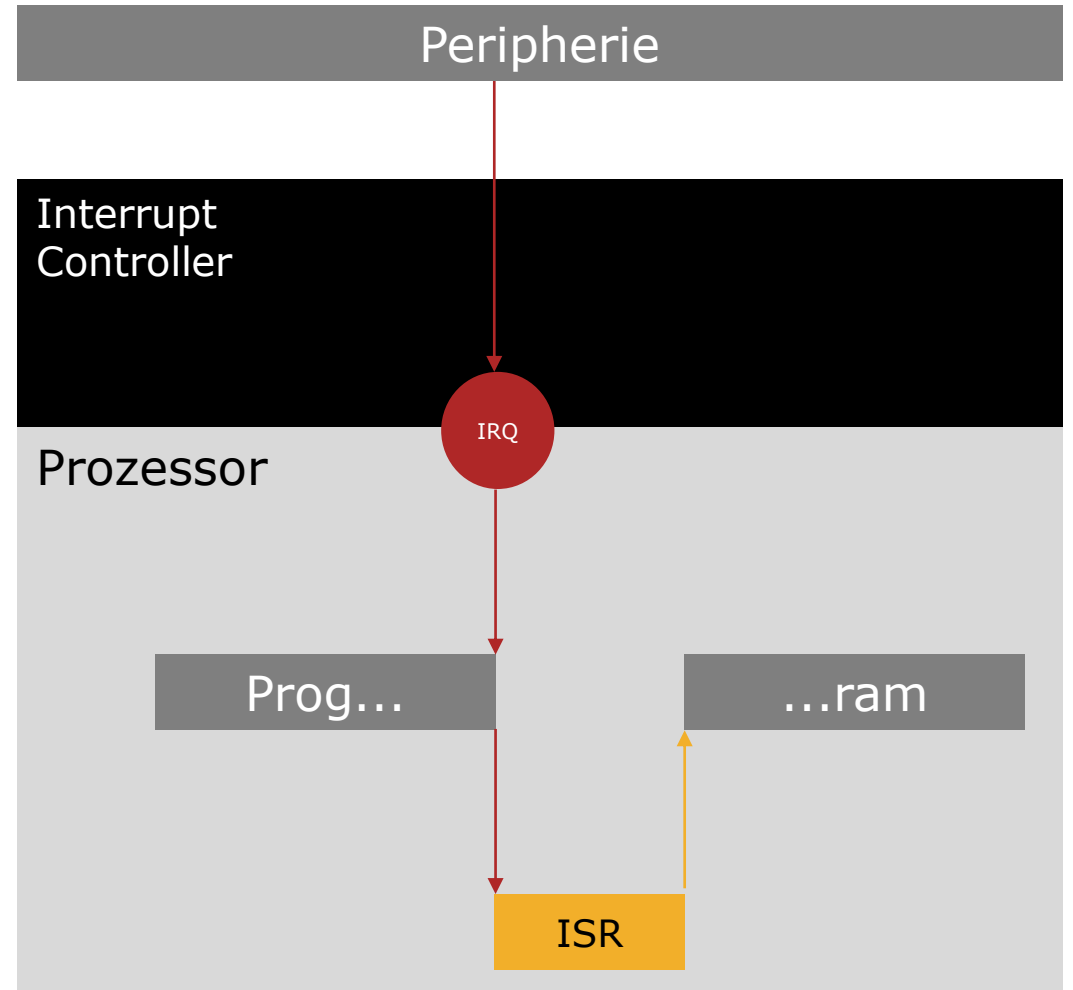
<	high-priority (not nice to other users)
N	low-priority (nice to other users)
L	has pages locked into memory (for real-time and custom IO)
s	is a session leader
l	is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
+	is in the foreground process group

Interrupts I - Grundlagen

- Unterbrechung eines laufenden Prozesses
- durch Event („Interrupt Request“ - IRQ) ausgelöst
- startet Unterbrechungsroutine („Interrupt-handler“ / „Interrupt Service Routine“ - ISR), sobald Befehl abgearbeitet



CPU mitteilen, dass jetzt externe Daten verfügbar sind



Interrupts II - Arten

Beispiele

- Tastatureingabe
- Audioaufnahme
- Puffer leer, aber Sound soll abgespielt werden
- Mausbewegung
- Datenempfang auf dem Netzwerk
- neues USB Gerät
- Grafikkarte hat Bild fertiggerechnet

Arten I

- Synchrone Interrupts
- Asynchrone Interrupts

Arten II

- Software Interrupts
- Hardware Interrupts

Interrupts III – Arten I

Synchrone Interrupts

- vorhersagbar
- reproduzierbar
- Architektur von Intel kennt 3 Klassen:
 - Traps (z.b. Breakpoint) - freiwillig
 - Exceptions (z.b. Page fault)-korrigierbar
 - Aborts (z.b. Double fault) - irreparabel

Asynchrone Interrupts

- Nicht vorhersehbar
- Können nicht exakt reproduziert werden (entstehen an anderer Stelle)
- meist von Peripherie ausgelöst

Interrupts IV – Arten II

Software Interrupts

- wird von Programm ausgelöst
- Trigger: Spezieller Maschinencode
- wird meist für System Calls und Exception Handling genutzt
 - System Calls ermöglichen Aufruf von Betriebssystem-Dienst ohne Adresse zu kennen
 - Parameter werden auf dem Stack abgelegt
 - Spezieller Maschinen Code löst Betriebssystem-Dienst aus
 - Exception Handling stellt stabilen Zustand, wenn möglich, wieder her (z.B. durch beenden des Programms)

Hardware Interrupts

- wird von der Hardware ausgelöst
- 2 Arten
 - Maskierbar (kann „ignoriert“ werden)
 - Nicht-maskierbar (muss ausgeführt werden)
 - Hardware Reset
 - Schwerer Hardwarefehler

Interrupts V – Interrupt Controller

Programmable Interrupt Controller (PIC)

- Chip auf/an der CPU
- verwaltet Hardware Interrupts
- Arbeitet als Art Multiplexer
- CPUs haben oft nur 1 Eingang für IRQ
- leitet Interrupts entsprechend der Priorität weiter

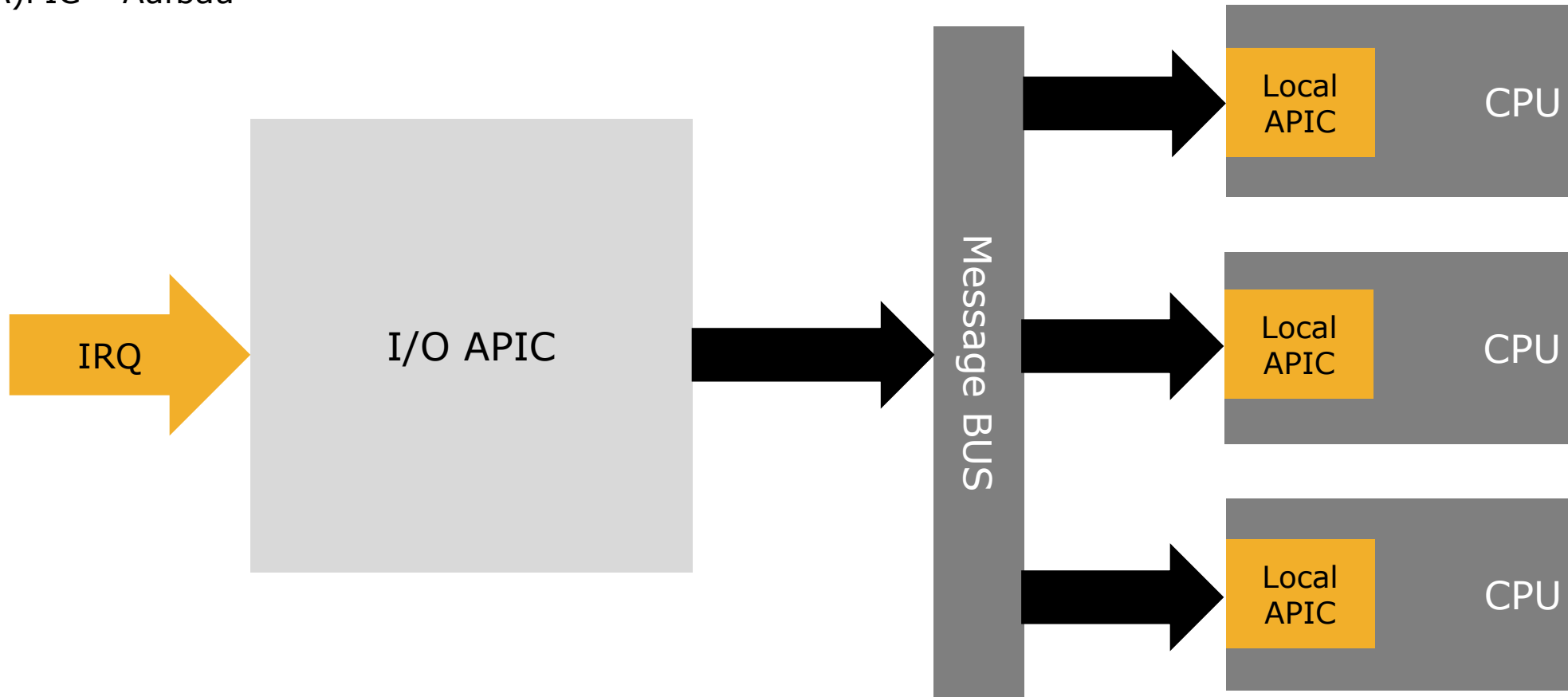
Nicht mit PICmicro vertauschen!

Advanced Programmable Interrupt Controller (APIC)

- verteilt Interrupts in Mehrprozessorsystemen
- Muss vom OS unterstützt werden
- 2 Teile
 - Local APIC (meist teil des Prozessors)
 - I/O APIC (liegt auf dem Motherboard)
- Funktionsweise
 - I/O APIC besitzt „Redirection Table“
 - I/O APIC nimmt interrupt von Hardware entgegen
 - Über Systembus an Local APICs je nach info in Redirection Table

Interrupts VI – Interrupt Controller

(A)PIC – Aufbau



Ausführungsmodi I

Benutzermodus - „User-Mode“

- eingeschränkter Zugriff auf Hardware / kein Zugriff
- führt Benutzerprogramme aus

Kernmodus – „Kernel-Mode“

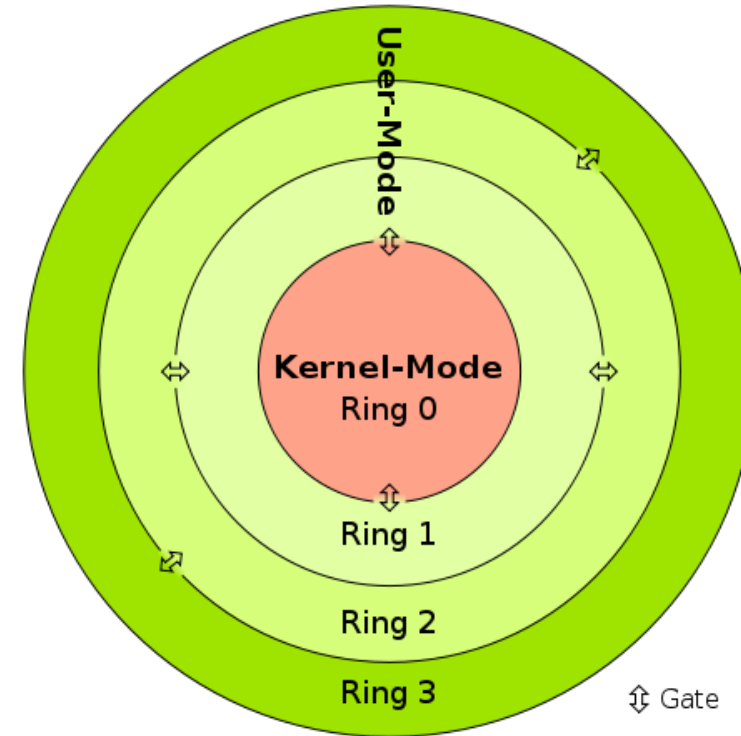
- Auch „Systemmodus“ genannt
- voller Hardwarezugriff
- führt Prozess- und Resourceverwaltung durch
- Betriebssystemprozess laufen in diesem Modus



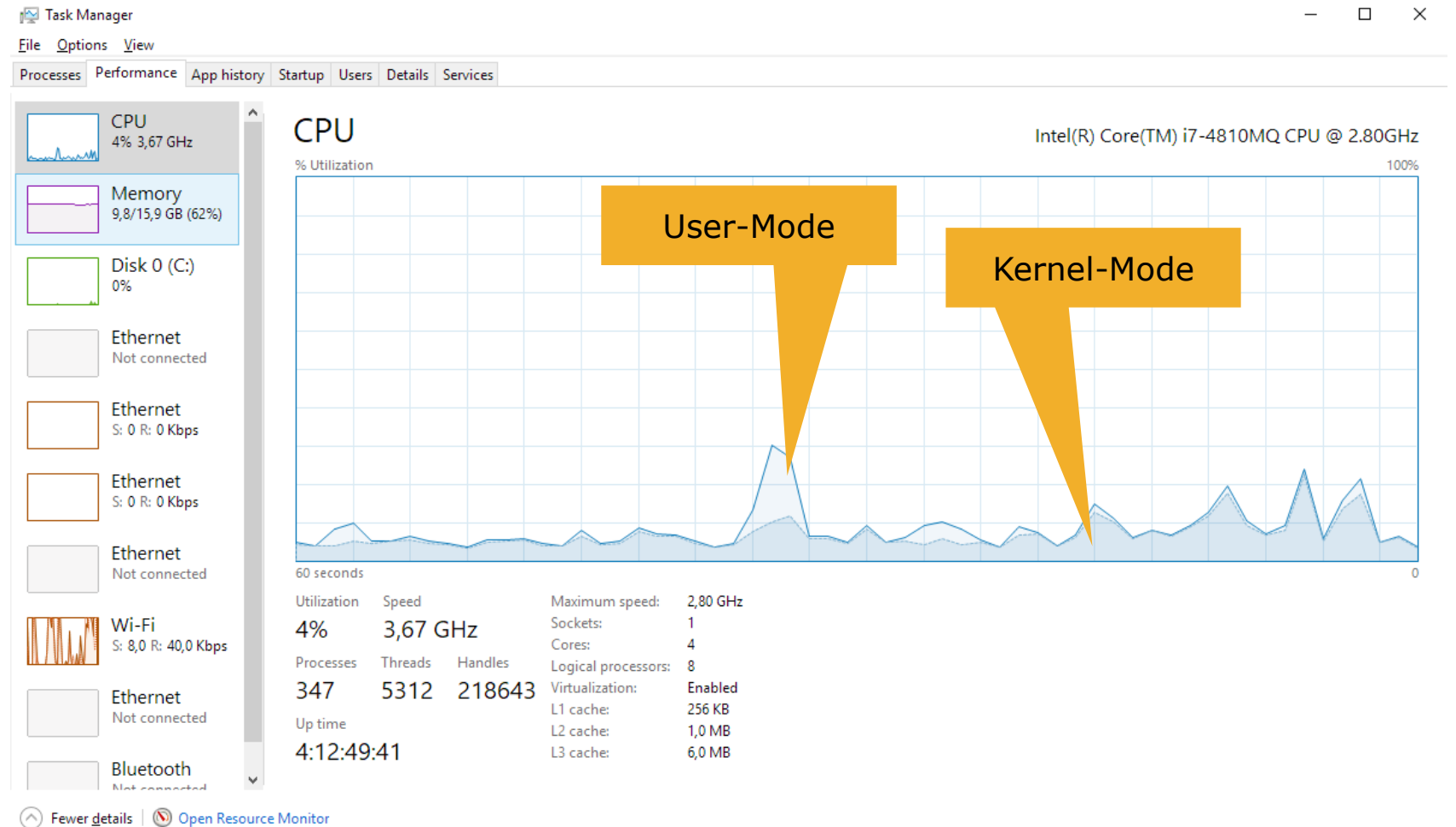
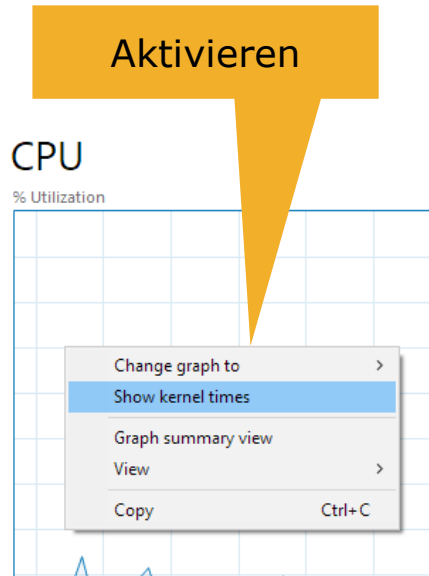
Ermöglicht modulares, sicheres
Betriebssystem

Umsetzung in x86 (Intel)

- 2 Modi sind zu wenig
- Etablierung von 4 „Sicherheitsringen“
 - Ring 0 & 1 = OS
 - Ring 2 & 3 = Anwendung
- „Kernel-Mode“
 - Ring 0 = Kernel
 - Ring 1 = Treiber
- „User-Mode“
 - Ring 2 = Treiber
 - Ring 3 = Anwendungen



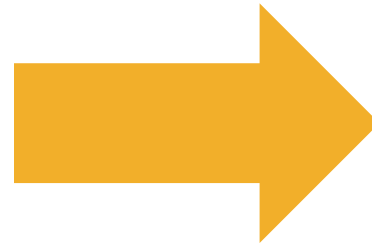
Ausführungsmodi III – Praxis (Windows)



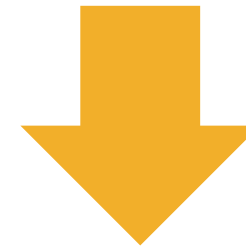
Prozessverklemmung I

Ziele des Process-Schedulers

- hohe CPU Auslastung
- hoher Durchsatz (Prozesse pro Zeiteinheit)
- kleine Ausführungszeiten pro Prozess
- kurze Antwortzeit
- faire Behandlung der Prozesse
- minimale Wartezeit



Ziele sind teilweise konträr

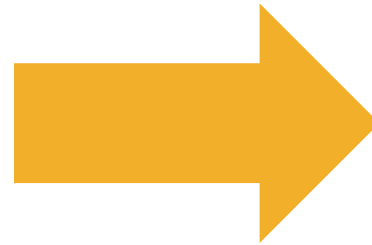


besondere Strategien nötig

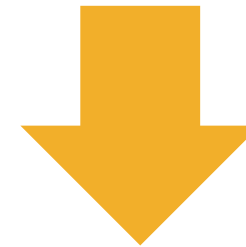
Prozessverklemmung II

Grundproblem

- pro Prozessor gibt es nur einen aktiven Prozess
- neben Hauptspeicher kann dem Prozess noch weiter Ressourcen zugewiesen werden
- ist Ressource bereits in Benutzung, wechselt Prozess in Status „Blockiert“
- Prozess im Status „Blockiert“, belegen Hauptspeicher & ggf. Ressourcen



Gefahr zur „Verklemmung“



besondere Strategien nötig

Prozessverklemmung III

Ein Praktisches Beispiel – Vorlesung beginnt

Ressource:
Student



Programm:
Vorlesung



Prozessverklemmung IV

Ein praktisches Beispiel – die speisenden Philosophen

Regeln

- nur mit 2 Gabeln kann man Spaghetti holen
- man darf nur die Gabeln direkt neben sich nutzen

Ablauf

- Ein Philosoph hungrig
 - nimmt Gabel links & rechts
 - isst
 - legt Gabeln zurück
- Alle 5 Philosophen hungrig
 - nehmen gleichzeitig linke Gabel
 - Warten bis rechte Gabel frei wird...

Deadlock



Bedingungen für Deadlocks

Exklusive Belegung von Ressourcen

- „mutual exclusion“
- (min.) 1 Prozess im kritischen Abschnitt
- *Notwendige Bedingung*

Nachforderung von Ressourcen

- „hold and wait“
- (min.) 1 Prozess hält Ressourcen und wartet auf neue Ressourcen
- *Notwendige Bedingung*

kein Entzug von Ressourcen

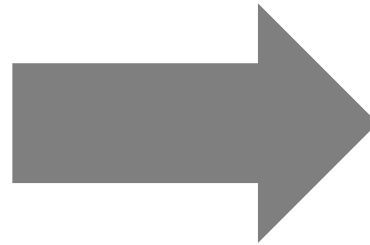
- „no preemption“
- Prozessen können die Ressourcen nicht entzogen werden
- *Notwendige Bedingung*

Zirkuläres Warten

- „circular wait“
- Geschlossene Kette an gegenseitig blockierende Prozesse
- *Hinreichende Bedingung*

Handeln im Ernstfall

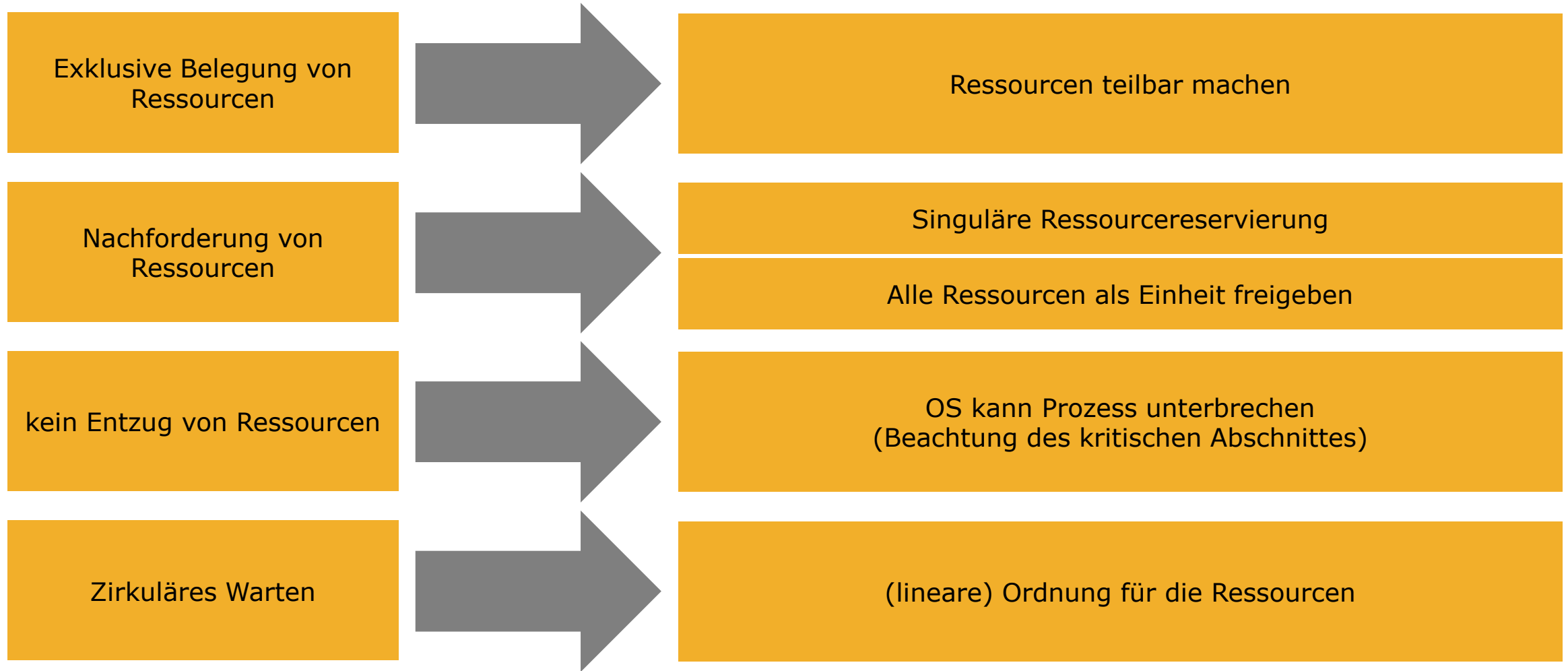
Zirkuläres Warten ist Folge
der notwendigen
Bedienungen



Möglich Handlung

- Ignorieren
 - *no comments...*
- Vorbeugen
 - Verhindern dass Deadlock überhaupt entsteht
- Vermeiden
 - Nutzung von Ressourcen kontrollieren
- Erkennen und Beheben
 - Zyklen erkennen und auflösen

Optionen gegen Deadlocks



Deadlocks verhindern

Vorgehen

- Prozesse geben Vorabinfo, welche Ressourcen benötigt werden
- OS gewährt Zugriff entsprechend
- Prozesse müssen warten, bis sie die Ressourcen zugeteilt bekommen
- Methoden
 - „resource allocation graph“ (dt. „Betriebsmittelgraph“)
 - „banker's algorithm“

Pro

- System wird stetig auf „festgefahrene“ Zustände geprüft
- Regulierung der Ressourcen verhindert Deadlock

Contra

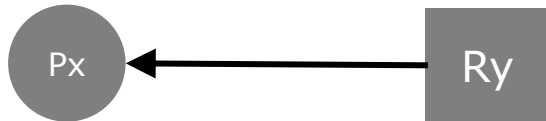
- Oft nicht alle Ressourcen im Voraus bekannt
- In der Theorie gut; in der Praxis schwer

Resource Allocation Graph (RAG)

Grundelemente

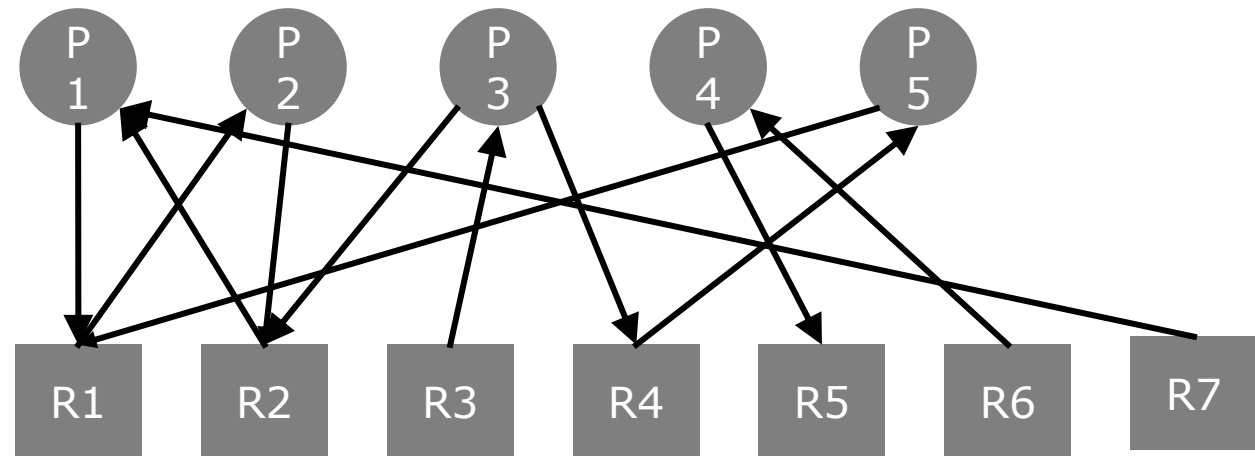


Prozess X wartet auf
Ressource Y



Prozess X hält Ressource Y

P1 hat R2 und R7 belegt und fordert R1 an
P2 hat R1 belegt und fordert R2 an
P3 hat R3 belegt und fordert R2 und R4 an
P4 hat R6 belegt und fordert R5 an
P5 hat R4 belegt und fordert R1 an



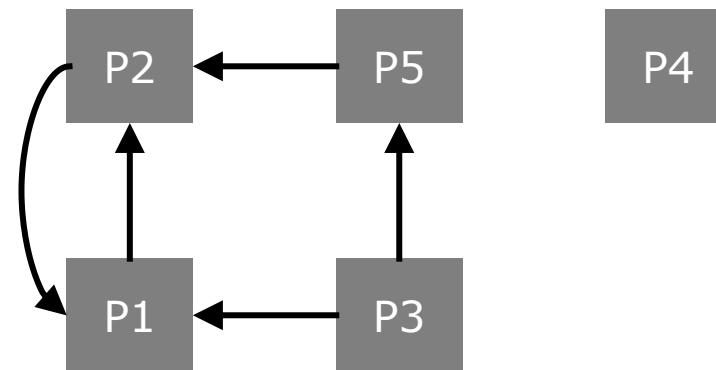
Alternative Darstellung

Grundelemente



Prozess X wartet auf
Ressource Y

P1 hat R2 und R7 belegt und fordert R1 an
P2 hat R1 belegt und fordert R2 an
P3 hat R3 belegt und fordert R2 und R4 an
P4 hat R6 belegt und fordert R5 an
P5 hat R4 belegt und fordert R1 an



Banker's Algorithm I

- Erfinder: Dijkstra
- Idee: „Kreditlimits“ vergeben
- Ermittlung der Ausführungsreihenfolge ohne „unsicheren Zustand“ zu erreichen (Deadlock)
- Datenstrukturen:
 - Current („Allocation“) = Ressourcen die jeder Prozess gerade hat
 - Request („Need“) (Ressourcen die vom Prozess noch gebraucht werden)
 - Existing („Maximum“) (maximale Ressourcen)
 - Available („Work“) (noch verfügbare Ressourcen)
- Vorgehen:
 - regelmäßig auf „unsicheren Zustand“ (Deadlock) prüfen

Available				
R0	R1	R2	R3	
4	3	42	7	

Existing				
R0	R1	R2	R3	
8	5	49	9	

	Current				
	R0	R1	R2	R3	
P1	1	0	3	0	
P2	0	1	0	1	
P3	3	0	4	1	
P4	0	1	0	0	

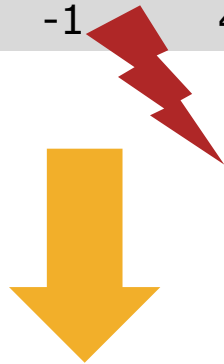
	Request				
	R0	R1	R2	R3	
P1	0	4	0	0	
P2	3	0	2	1	
P3	0	5	36	3	
P4	0	0	0	9	

Beispiel aus Wikipedia ;)

Banker's Algorithm II

Prüfung P1

	R0	R1	R2	R3
Available	4	3	42	7
Request	0	4	0	0
Differenz	4	-1	42	7



P1 geht nicht

Available				
R0	R1	R2	R3	
4	3	42	7	

Existing				
R0	R1	R2	R3	
8	5	49	9	

Current				
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

Request				
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm III

Prüfung P2

	R0	R1	R2	R3
Available	4	3	42	7
Request	3	0	2	1
Differenz	1	3	40	6



P2 geht

Available				
R0	R1	R2	R3	
4	3	42	7	

Existing				
R0	R1	R2	R3	
8	5	49	9	

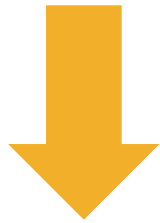
Current				
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

Request				
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm IV

P2 ausführen und beenden

	R0	R1	R2	R3
Available	4	3	42	7
Current	0	1	0	1
Summe	4	4	42	8



(new) Available				
R0	R1	R2	R3	
4	4	42	8	

Available				
R0	R1	R2	R3	
4	3	42	7	

Existing				
R0	R1	R2	R3	
8	5	49	9	

Current				
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

Request				
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm V

Prüfung P3

	R0	R1	R2	R3
Available	4	4	42	8
Request	0	5	36	3
Differenz	4	-1	6	5



P3 geht nicht

Available				
R0	R1	R2	R3	
4	4	42	8	

Existing				
R0	R1	R2	R3	
8	5	49	9	

	Current				
	R0	R1	R2	R3	
P1	1	0	3	0	
P2	0	1	0	1	
P3	3	0	4	1	
P4	0	1	0	0	

	Request				
	R0	R1	R2	R3	
P1	0	4	0	0	
P2	3	0	2	1	
P3	0	5	36	3	
P4	0	0	0	9	

Banker's Algorithm VI

Prüfung P4

	R0	R1	R2	R3
Available	4	4	42	8
Request	0	0	0	9
Differenz	4	4	42	-1



P4 geht nicht

Available			
R0	R1	R2	R3
4	4	42	8

	Current			
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

Existing			
R0	R1	R2	R3
8	5	49	9

	Request			
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm VII

Prüfung P1

	R0	R1	R2	R3
Available	4	4	42	8
Request	0	4	0	0
Differenz	4	0	42	8



P1 geht

Available				
R0	R1	R2	R3	
4	4	42	8	

Existing				
R0	R1	R2	R3	
8	5	49	9	

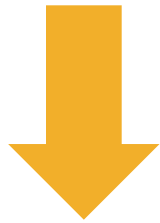
	Current				
	R0	R1	R2	R3	
P1	1	0	3	0	
P2	0	1	0	1	
P3	3	0	4	1	
P4	0	1	0	0	

	Request				
	R0	R1	R2	R3	
P1	0	4	0	0	
P2	3	0	2	1	
P3	0	5	36	3	
P4	0	0	0	9	

Banker's Algorithm VIII

P1 ausführen und beenden

	R0	R1	R2	R3
Available	4	4	42	8
Current	1	0	3	0
Summe	5	4	45	8



(new) Available				
R0	R1	R2	R3	
5	4	45	8	

Available				
R0	R1	R2	R3	
4	4	42	8	

Existing				
R0	R1	R2	R3	
8	5	49	9	

Current				
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

Request				
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm IX

Prüfung P3

	R0	R1	R2	R3
Available	5	4	45	8
Request	0	5	36	3
Differenz	5	-1	9	5



P3 geht nicht

Available			
R0	R1	R2	R3
5	4	45	8

Existing			
R0	R1	R2	R3
8	5	49	9

Current				
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

Request				
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm X

Prüfung P4

	R0	R1	R2	R3
Available	5	4	45	8
Request	0	0	0	9
Differenz	5	4	45	-1



P4 geht nicht

Available			
R0	R1	R2	R3
5	4	45	8

Existing			
R0	R1	R2	R3
8	5	49	9

	Current			
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

	Request			
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Banker's Algorithm XI

Prozesse führen zum Deadlock!

Available			
R0	R1	R2	R3
5	4	45	8

Existing			
R0	R1	R2	R3
8	5	49	9

	Current			
	R0	R1	R2	R3
P1	1	0	3	0
P2	0	1	0	1
P3	3	0	4	1
P4	0	1	0	0

	Request			
	R0	R1	R2	R3
P1	0	4	0	0
P2	3	0	2	1
P3	0	5	36	3
P4	0	0	0	9

Deadlocks finden & beheben I

Erkennung

- prinzipiell keine Komponente zur Erkennung vorgesehen
- Idee: Wartegraphen um Zyklen zu finden
 - Suche von Prozess / Ressource die blockiert
 - Zu oft ausgeführt == Prozessor verschwendet
 - Zu selten ausgeführt == Ressourcen unausgelastet
- In der Praxis: Erkennung von Symptomen
 - Ressourcenanforderung dauert sehr lange
 - Prozessor sehr lange untätig
 - Prozessorauslastung sinkt

Lösung I

Abbruch

Lösung II

Unterbrechung / Reset

Lösung III

Entzug von Betriebsmitteln

Deadlocks finden & beheben II

Lösung I - Abbruch

- Alle „toten“ Prozesse auf einmal
 - maximaler Schaden
 - alle Prozesse müssen neu starten
- nach und nach bis Zyklus aufgelöst
 - maximaler Aufwand
 - Nach jedem Abbruch muss Zyklus geprüft werden

Lösung II - Unterbrechung / Reset

- großer Aufwand
- Ermittlung von „effektivsten Opfer“
- Erfordert „Transaktionen“
 - Checkpoints um auf bestimmten Zustand zurück zu gehen
- Gefahr: Prozess „verhungert“ => muss vermieden werden

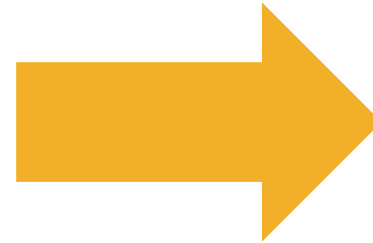
Lösung III – Entzug von Betriebsmitteln

- Nicht immer möglich
- Benötigte Ressourcen werden Prozess entzogen
 - Zuteilung zu anderen Prozessen
- Sind Ressourcen frei, kann Prozess sie wieder anfordern
- Geht nicht bei
 - Schreiben von Dateien
 - Ausgabe am Drucker

Prozess-Scheduler

Recap: Ziele des Schedulers

- hohe CPU Auslastung
- hoher Durchsatz (Prozesse pro Zeiteinheit)
- kleine Ausführungszeiten pro Prozess
- kurze Antwortzeit
- faire Behandlung der Prozesse
- minimale Wartezeit



Scheduler bestimmt welcher Prozess wann wie lange ausgeführt werden darf

Scheduling Strategien

Preemptive Strategien

- „präventiv“
- verdrängt Prozess
- Prozess wird „Prozessor entzogen“
- Grundlage für meisten Strategien: Hardware Uhr



Vorrangig in modernen OS eingesetzt

Non preemptive Strategien

- passive Strategien
- „Nicht verdrängend“
- Prozess läuft, bis er
 - Terminiert
 - Blockiert (z.b. warten auf I/O)



Gefahr das Prozessor nicht frei wird

Preemptive Strategien

Round Robin

- Zeitscheiben-verfahren
- Nacheinander hat jeder Prozess gleich viel Zeit in der CPU

Dynamic Priority based Round Robin

- Basiert auf Round-Robin
- Priorisierte Warteschlange wird Round-Robin vorgeschaltet
 - Nach jeder Zeitscheibe, wächst die Priorität eines Prozesses bis Schwellenwert erreicht
 - Danach einsortieren in die Hauptschlange
 - Reset der Prio. nach Ausführung

Shortest remaining time

- Prozesse werden nach (geschätzter) verbleibender Zeit sortiert
- Prozess mit kürzester Zeit darf zuerst

Preemptive Variante von „shortes Job next“

Non-Preemptive Strategien

Highest Response Ration Next

- Verhältnis Wartezeit/Laufzeit entscheidet
- Größtes Verhältnis zu erst – d.h. Prozesse mit geringer Rechenzeit bevorzugt
- Basiert auf Schätzung

First In First Out

- Bearbeitung in der Reihenfolge, wie aufgetreten
- sobald Prozess wartet, kommt der nächste
- gute CPU Auslastung
- schlechte Ressourcen Auslastung

Highest Priority First

- jeder Prozess bekommt bestimmt Priorität (vom OS)
- Scheduler arbeitet Prozesse nach Priorität ab
- „niedere“ Prozesse können von „höheren“ Prozessen verdrängt werden

Earliest due date

- Prozess mit geringster Deadline darf zuerst
- minimiert „Verspätung“
- setzt aber bekannte Deadlines voraus

Shortest Job next

- Prozesse werden nach (geschätzter) Ausführungszeit sortiert
- Prozess mit kürzester Zeit darf zuerst

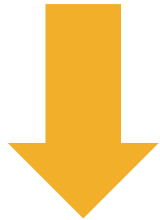
Multilevel Queue Scheduling

- Aufteilung der Prozesse in Gruppen (z.B. Vordergrund / Hintergrund)
- durch unterschiedliche Anforderungen ist gezieltes einteilen möglich

Scheduling in der Praxis

Die Mischung macht's

Keine ideale Lösung



Kombination mehrerer Strategien

Beispiel: Windows (seit NT)

- Mischung aus „Highest Priority First“, „FIFO“, „Round Robin“
- Windows definiert verschiedene Priority Level
- pro Level = 1 Queue
- Prozesse können Priorität manuell anpassen
- High Priority Queues == Round Robin
- Low Priority Queues == FIFO

Prozesskommunikation

Was ist ein Prozess?

Ausführung eines Programms auf einer CPU
mitsamt der entsprechenden Umgebung



Prozesse wissen nichts voneinander!

Beispiel Szenarien

- Prozesse nutzen Systemressourcen
- Prozesse müssen Daten austauschen
- Prozesse nutzen gemeinsamen Speicher



Prozesse wissen voneinander!

Kommunikationsarten

speicherbasierte Kommunikation

- Shared Memory
- Kommunikation über Dateien

nachrichtenbasierte Kommunikation

- Message Queue
- Pipes

Shared Memory

Shared Memory

- Nutzung von gemeinsamen Memory-Bereich
- schnellste Form des Datenaustausches
- besondere Speicherverwaltung seitens OS notwendig!
- kann große Datenmengen verarbeiten



Siehe „Speicherverwaltung“ > „Page Sharing“

Prozess 1

VPN	Valid	PFN
0x42	1	0x1
0x50	1	0x20

Prozess 2

VPN	Valid	PFN
0x55	1	0x1
0x61	1	0x33

Phy. Memory

4311
222
553
67
...
544

Kommunikation über Dateien

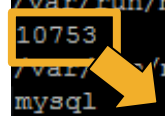
Kommunikation über Dateien

- Dateien liegen im Dateisystem
- benötigt Mechanismus für simultanen Dateizugriff
 - > meistens via „LOCK“ auf Datei

Beispiel: MySQL (Linux)

- mysqld darf nur 1x laufen
- /var/run/mysqld/mysql.pid listet die Process ID
- Gibt es die Prozess ID nicht mehr, läuft MySQL nicht mehr

```
/var/run/mysqld$ ls -l
total 4
-rw-rw---- 1 mysql mysql 6 Aug  7 06:36 mysqld.pid
srwxrwxrwx 1 mysql mysql 0 Aug  7 06:36 mysqld.sock
/var/run/mysqld$ sudo cat mysqld.pid
10753
/var/run/mysqld$ sudo ps aux | grep mysqld
mysql  10753  0.0  4.5 689724 46004 ?        Ssl  Aug07  27:03 /usr/sbin/mysqld
user   22825  0.0  0.0 11940   924 pts/1    S+   22:04   0:00 grep --color=auto mysqld
```



Message Queue

Funktionsweise

- Quellprozesse reihen Nachrichten in Warteschlange von Zielprozess ein
- Zielprozess kann Warteschlange nach und nach bearbeiten
- typische Umsetzung: First-in First-out
- Prioritäten möglich

Beispiel: Windows API - UI

- Jedes Win32 Programm hat „Message Queue“ für UI
- Interaktion mit UI reiht Nachricht ein
- Programm arbeitet Nachrichten ab

```
#include <windows.h>
```

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_CLOSE:
            DestroyWindow(hwnd);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

**bearbeitet
Nachrichten**

Nachrichten

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    //a lot of magic which brings up a window and sets WndProc() as Window Queue Handler
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```

**empfängt
Nachrichten**

Pipes

- Datenstrom zwischen 2 Prozessen
- Ausgabe des Quellprozesses == Eingabe des Zielprozesses
- arbeitet nach FIFO Prinzip
- Bedingung:
 - Daten gehen nur in eine Richtung (Einbahnstraße)
 - Prozess müssen gemeinsame Vorfahren haben

Beispiel: Pipes (Linux / Unix)

- Ausgabe eines Programms kann an Eingabe eines anderen Programm geleitet werden

```
/tmp$ ls -l | wc -l  
372
```

Prozess „ls“
listet alle
Dateien

„|“ (Pipe) übergibt
Dateiliste von „ls“
an „wc“

Prozess „wc“ zählt
die Zeilen

Prozesssynchronization I

Grundlagen

- Prozesskommunikation erfordert Prozesssynchronisation
- verhindert Gegenseitigen Ausschluss
- Ermöglicht Bedingungsynchronisation (Prozess kann auf die Erfüllung einer Bedingung durch einen anderen Prozess warten)



Art I

Konkurrenz

gegenseitiger Ausschluß
("mutual exclusion")

Art II

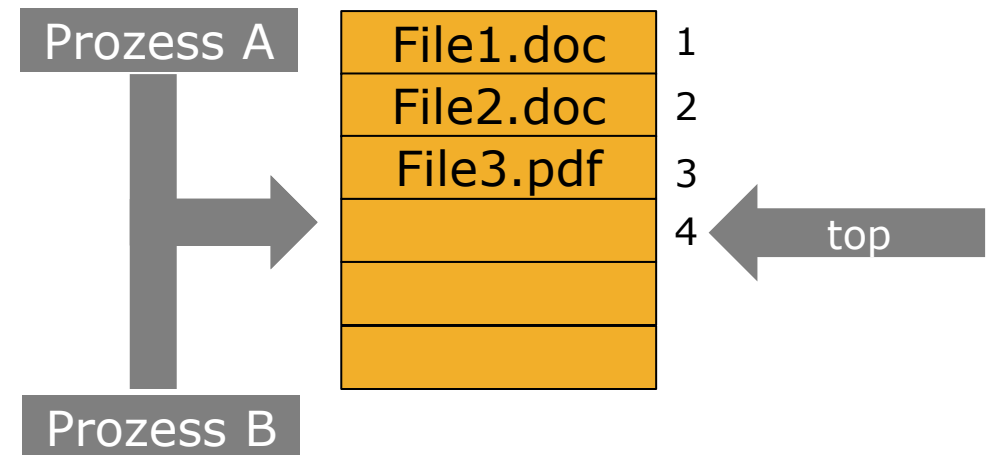
Abhängigkeit

Zustands- oder
Ereignissynchronisation

Prozesssynchronisation II

Beispiel

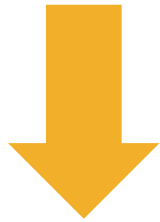
- Problem: 2 Prozesse wollen gleichzeitig einen Druckauftrag planen
- Variable „top“ speichert obersten Punkt in der Warteschlange
- Prozess A hat Prozessor => liest „top“ == 4 => INTERRUPT (Kontext Wechsel)
- Prozess B bekommt Prozessor => liest „top“ == 4 => schreibt Auftrag in die Warteschlange => top == 5
- Prozess A bekommt Prozessor wieder => schreibt in Warteschlangen an top = 4!



Prozesssynchronisation III

Lösungsideen

- Interrupts verhindern?
- zusammenhängenden Code markieren?



Prinzip des Kritischen Abschnitts

Grundlagen

- definiert nicht unterbrechbaren Code
- verhindert Informationsparadoxon
- Muss gewissen Anforderungen erfüllen:
 - „mutual exclusion“ (wenn ein Prozess im kritischen Abschnitt ist, darf kein zweiter im kritischen Abschnitt sein / kommen)
 - Prozess A kann nicht Prozess B nicht daran hindern in den kritischen Abschnitt zu wechseln
 - Wenn ein Prozess auf den kritischen Abschnitt wartet, muss er auch dahin kommen („no starvation“)
 - Alle Anforderungen müssen unabhängig von Leistung & Anzahl der Prozessoren erreicht werden

viele verschiedene Implementierungen!

Unterbrechungssperre

Grundidee:

- Während der Ausführung eines kritischen Abschnittes werden alle Interrupts maskiert
- Auch Betriebssystem kann Prozess nicht unterbrechen
- Umsetzung durch „acquire()“/ „release()“ Mechanismus
 - Prozess nutzt „acquire()“ um einen Lock zu haben
 - Lock gehört diesem Prozess bis „release()“ aufgerufen wird

Pro:

- Einfach umsetzbar

Contra:

- ggf. wird vergessen die Maskierung aufzuheben
- Zeit läuft asynchron (Hardware-Uhr kann keine Unterbrechung auslösen)
- ggf. hohe Reaktionszeiten bei Interrupts für Ein/Ausgabe
- Nicht für Mehrprozessorbetrieb nutzbar

Test & Set Instruktion

Grundidee:

- Lesen und Schreiben werden eine Instruktion
- Umsetzung mit Hardwareunterstützung
- Instruktion (logische Implementierung)

```
int TestnSet(int* lock) {  
    int initial = *lock;  
    *lock = 1;  
    return initial;  
}
```
- Beispiel

```
static int lock = 0;  
  
while(TestnSet(&lock) == 1; //jemand hat gerade lock!  
//kritischer Abschnitt  
lock = 0;
```

Pro:

- Im Programm einfach nutzbar

Contra:

- erfordert Hardwareunterstützung
- abhängig von Schedulingstrategie
Verklemmung oder unfaire Vergabe möglich

8081 hat Test & Set Instruktion:
JBC – “Jump if Direct bit is Set and Clear Bit”

Semaphore I

Grundidee:

- Nicht Prozess, sondern OS steuern kritischen Abschnitt
- Idee kam von Dijkstra mit Algol 68 (1965)
- Für jede zu schützende Datenmenge wird eine „Sperrvariable“ („Semaphor“) definiert
- Semaphor signalisiert den Belegungszustand durch einen Prozess



Semaphore II

Einfache Implementierung:

- Datenmenge und Sperrvariable getrennt

```
//Zugriff beantragen
int pass(int* S){
    while (*S <= 0){
        sleep(10);
    }
    *S++;
    return 0;
}
```

```
//Zugriff Freigeben
int release(int* S){
    *S--;
    return 0;
}
```

blockiert Prozess und spart
daher Prozessorzeit

```
void main(int argc, char* argv*) {
    int s = 0;
    int data = 0;

    pass(&s);
    //kritischer block
    //verändere data
    release(&s);
}
```

Semaphore III

Pro:

- Sehr flexibel
- Sehr „mächtig“
- Löst fast alle Synchronisationsprobleme
- Einfache Implementierung für gegenseitigen Ausschluss

Contra:

- Komplexe Problem / Bedingungsynchronisation kann schwierig werden
- Verklemmung möglich
- Durch Zusatzfunktionen (pass() und release()) im Code verstreut
- Fehler in der Implementierung möglich
 - pass() und release() vertauschen / vergessen
 - Vertauschen von mehreren Semaphoren

Monitor I

Grundidee:

- Synchronisationsmechanismus muss nicht explizit angegeben werden
- Synchronisationsprimitive wird vom Compiler automatisch eingefügt, wenn entsprechend gekennzeichnet
- Intern wird oft Semaphore eingesetzt.
- z.B. in Java eingesetzt („synchronized“)

Pro:

- einfach anwendbar

Contra:

- Programmiersprache muss Funktion unterstützen

Monitor II

Demo „Consumer –Producer“

Nachrichten I

Prinzip des Kritischen Abschnittes

- verhindert Unterbrechung
- Basiert auf Shared Memory



Alternativen?



Nachrichten!

Grundlagen

- Kommunikation über Nachrichten
- Quellprozess sendet Nachricht an Zielprozess
- Erfordert send() und receive() Funktion
- Kann in 2 Arten geteilt werden
 - Synchron
 - Asynchron

Nachrichten II

Synchrone Nachrichten

- Prozesse blockieren beim Senden / Empfangen
 - `send()` blockiert bis Nachricht empfangen ist
 - `receive()` blockiert bis neue Nachricht vorhanden
- Einfacher zu implementieren

Asynchrone Nachrichten

- Keiner der Prozesse blockiert
- `send()` überträgt Nachricht (Wiederholung falls keine Empfangsbestätigung nach Zeit t)
- `receive()` muss zwischen neuer & wiederholter Nachricht unterscheiden
 - bestätigt / wiederholt Bestätigung für Nachrichtenempfang
- Erfordert Puffer als Nachrichtenzwischenspeicher