ulm university universität ulm

Grundlagen der Betriebssysteme

[CS2100]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

ulm university universität ulm

Kapitel 4

Prozesse und Nebenläufigkeit

Inhalte der Vorlesung

1. Einführung
2. Zahlendarstellungen und Rechnerarithmetik
3. Einführung in Betriebssysteme
4. **Prozesse und Nebenläufigkeit**
5. Filesysteme
6. Speicherverwaltung
7. Einführung in MIPS-Assembler
8. Rechteverwaltung
9. Ein-/Ausgabe und Gerätetreiber

Inhalte des Kapitels (1)

4. Prozesse und Nebenläufigkeit

- Einordnung & Motivation
- Prozesse
  - Prozessrepräsentation (Prozesskontrollblock)
  - Prozesswechsel *(Context Switch)*
  - Prozesszustände
- Auswahlstrategien *(Scheduling)*
  - *First Come First Served*
  - *Shortest Job First*
  - Prioritäten
  - *Round Robin*
  - *Multilevel-Queue Scheduling*
  - *Multilevel-Feedback-Queue Scheduling*
- ...

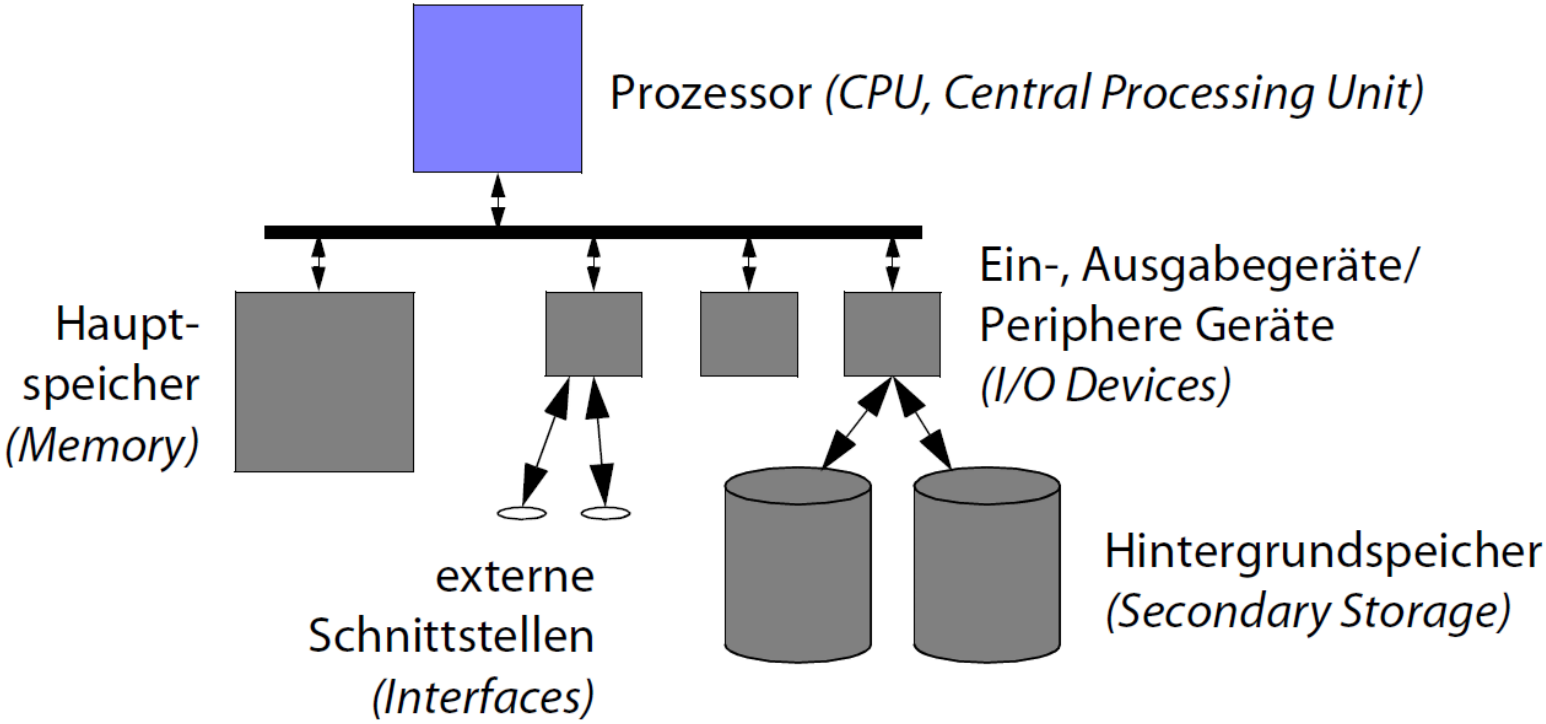
Inhalte des Kapitels (2)

4. Prozesse und Nebenläufigkeit

- ...
- Aktivitätsträger *(Threads)*
  - *User-level Threads*
  - *Kernel-level Threads*
  - *Lightweight Processes*
- Parallelität und Nebenläufigkeit
- Koordinierung
  - Kritische Abschnitte *(Critical Sections)*
  - Wechselseitiger Ausschluss (*Deadlocks*, Algorithmus von Peterson)
  - Spezielle Maschinenbefehle (*Test-and-Set*, *Swap*)
  - Semaphore

Einordnung

Betroffene physikalische Ressourcen

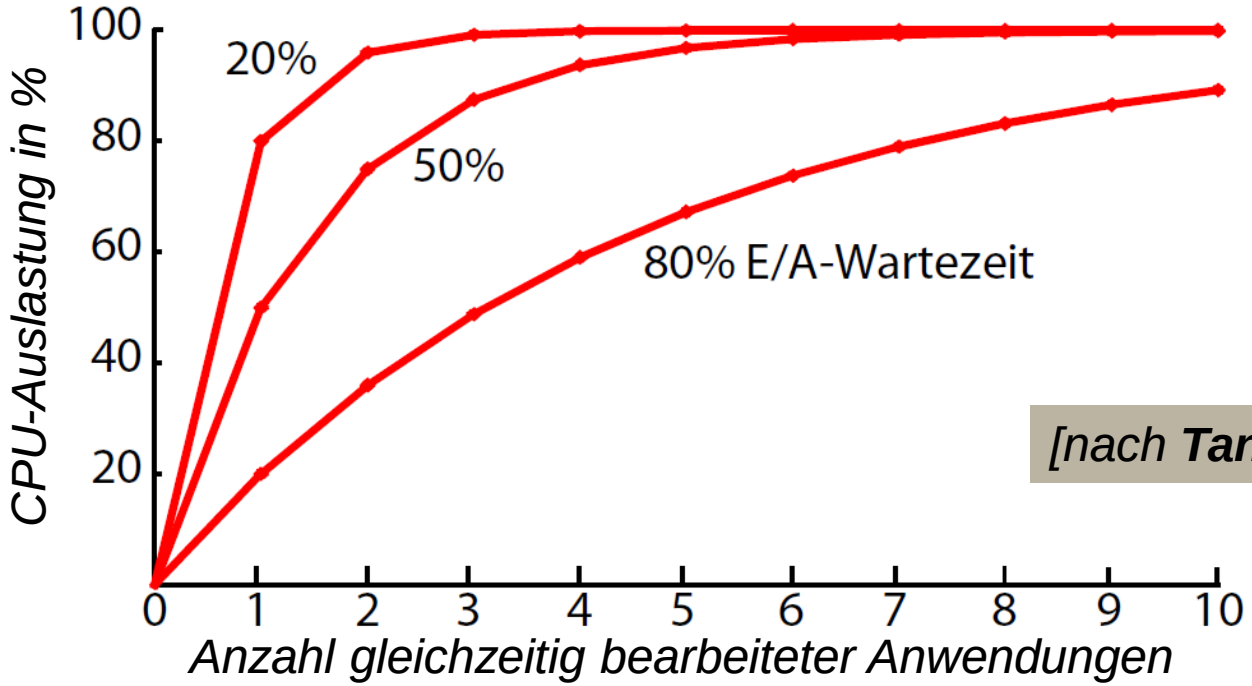


Motivation (1)

Mehrprozessbetrieb *(Multitasking)*

– Bessere Auslastung der CPU, falls Anwendungen Wartezeiten haben

  – Z.B. bei Ein-/Ausgabe



*[nach **Tanenbaum**, 1995]*

– Einfachere Struktur bei großen Anwendungen

Motivation (2)

Mehrbenutzerbetrieb *(Multi-User)*

- Mehrere Benutzer führen auch mehrere Programme aus
- Ähnlich *Multitasking*-Betrieb
  - Jedoch: Unterscheidung von Benutzern und evtl. Zugriffsberechtigungen

Prozesse

Terminologie

- *Programm:* Folge von Anweisungen und dazugehörigen Daten
  - Hinterlegt bspw. als Datei auf dem Hintergrundspeicher
- *Prozess:* Programm, das sich in Ausführung befindet, sowie dessen aktuellen Daten
  - *Beachte: Ein* Programm kann sich *mehrfach* in Ausführung befinden, d.h. es kann mehrere Prozesse zu einem Programm geben

Prozess stellt Ausführungsumgebung bereit

- Adressraum (Schutzumgebung für den Speicher eines Prozesses)
- Kontext für Ressourcenanforderungen (Speicher, Dateien, etc.)
- Prozess als virtueller Prozessor
  - Anweisungen des Prozesses werden abgearbeitet

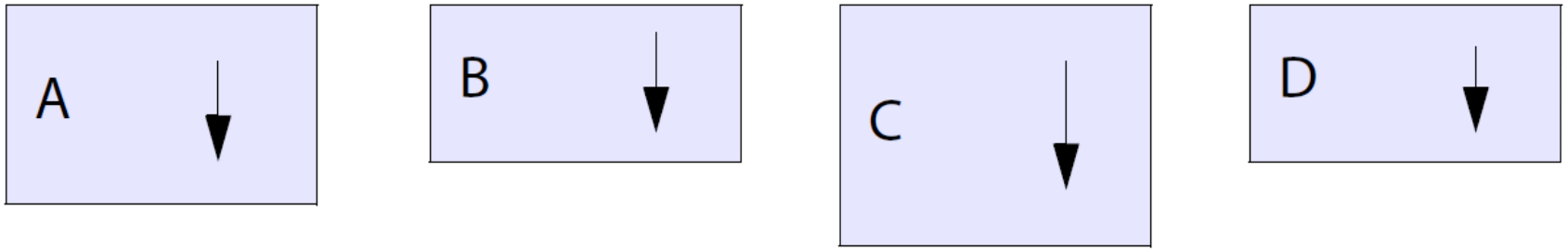
Repräsentation / Datenstruktur pro Prozess

Prozesskontrollblock *(Process Control Block, PCB)*

- Enthält alle nötigen Daten für einen Prozess
- Beispielsweise in UNIX/Linux:
  - Prozessnummer *(process identifier, PID)*
  - Verbrauchte Rechenzeit
  - Erzeugungszeitpunkt
  - Kontext (Register etc., insbes. PC)
  - Speicherabbildung (Information zur Schutzumgebung)
  - Eigentümer *(user ID UID, group ID GID)*
  - Wurzelverzeichnis, aktuelles Verzeichnis
  - Offene Dateien
  - ...

Prozesswechsel (1)

Konzeptionelles Modell



Vier Prozesse mit eigenständigen Programmzählern

Umschaltung zwischen den Prozessen

- Ein Prozessor kann zu jeder Zeit immer nur einen Prozess abarbeiten
- Umschaltung zwischen den Prozessen
  - Prozesse erleben Fortschritt ihrer Programmzähler und ihrer ausgeführten Anweisungen
- Transparentes Umschalten
  - Prozess bemerkt andere Prozesse (zunächst) nicht

Prozesswechsel (2)

Prozessumschaltung (Kontextwechsel, *Context Switch*)

- Sichern der Register des laufenden Prozesses, inkl. Programmzähler (Kontext)
- Auswahl des neuen Prozesses
- Ablaufumgebung des neuen Prozesses herstellen (z.B. Speicherabbildung, etc.)
- Gesicherte Register laden, und
- Prozessor aufsetzen, d.h. Programmzähler laden

Prozesswechsel (3)

Prozesswechsel unter Kontrolle der Prozesse

- Gerade laufender Prozess bestimmt Zeitpunkt des Kontextwechsels
  - Konzept der Co-Routine
- Kein transparentes Umschalten
- Keine Fairness

Wunsch nach fairer Zuteilung der Ressource „Prozessor“

Prozesswechsel (4)

Prozesswechsel unter der Kontrolle des Betriebssystems

- Mögliche Eingriffspunkte:
  - Systemaufrufe
  - Unterbrechungen
- Wechsel nach/in Systemaufrufen
  - Warten auf Ereignisse (z.B. Zeitpunkt, Nachricht, Lesen Plattenblock)
  - Terminieren des Prozesses
- Wechsel nach *Interrupts*
  - Ablauf einer Zeitscheibe
  - Bevorzugter Prozess wurde laufbereit

Auswahlstrategie zur Wahl des nächsten Prozesses

- *Scheduler*-Komponente des Betriebssystems

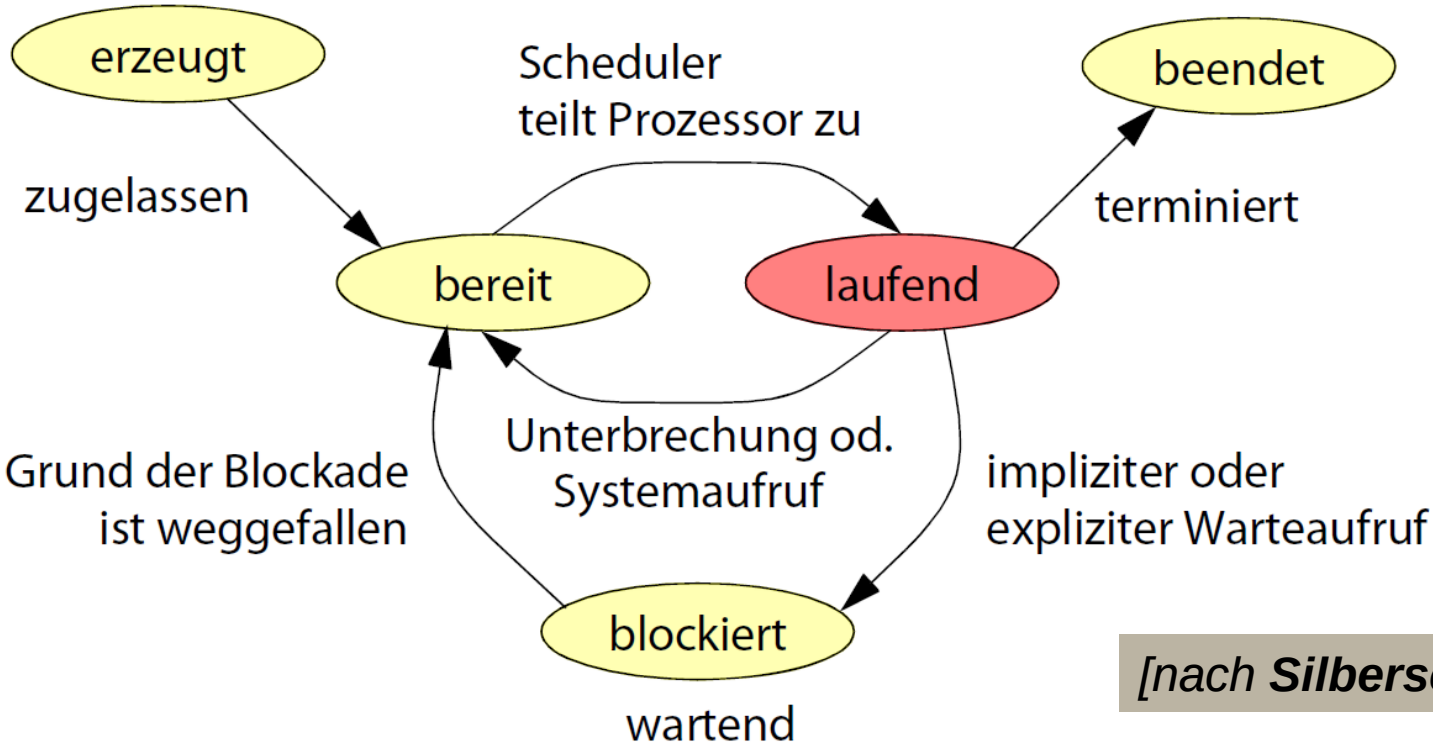
Prozesszustände (1)

Ein Prozess befindet sich in einem der folgenden Zustände

- Erzeugt *(New)*
  Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Ressourcen
- Bereit *(Ready)*
  Prozess besitzt alle nötigen Ressourcen und ist bereit zum Laufen
- Laufend *(Running)*
  Prozess wird auf einem realen Prozessor ausgeführt
- Blockiert *(Blocked / Waiting)*
  Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer I/O-Operation, Zuteilung einer Ressource, Empfang einer Nachricht); zum Warten wird er blockiert
- Beendet *(Terminated)*
  Prozess ist beendet, einige Ressourcen sind aber noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

Prozesszustände (2)

Zustandsdiagramm



*[nach **Silberschatz,** 1994]*

– Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors vornimmt

Roter Faden

4. Prozesse und Nebenläufigkeit

- Einordnung & Motivation
- Prozesse
  - Prozessrepräsentation (Prozesskontrollblock)
  - Prozesswechsel *(Context Switch)*
  - Prozesszustände
- Auswahlstrategien *(Scheduling)*
- Aktivitätsträger *(Threads)*
- Parallelität und Nebenläufigkeit
- Koordinierung

Strategien zur Auswahl des nächsten Prozesses

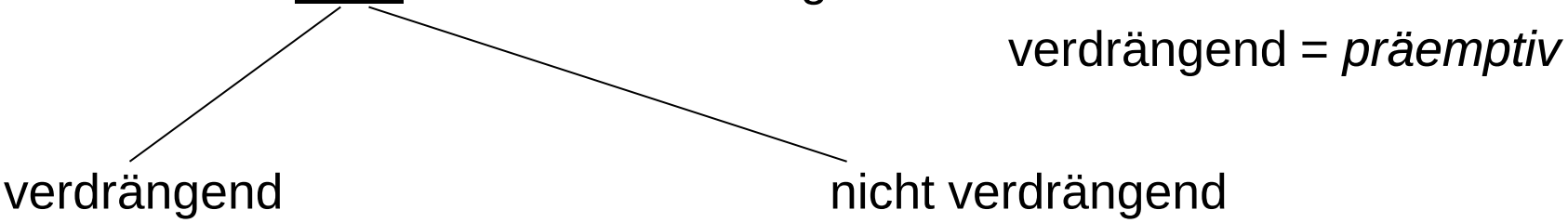
Scheduling Strategies

– Mögliche Stellen zum Treffen von *Scheduling*-Entscheidungen

1. Prozess wechselt von „laufend“ nach „blockiert“ (z.B. bei I/O)
2. Prozess wechselt von „laufend“ nach „bereit“ (z.B. bei einer Unterbrechung des Prozessors)
3. Prozess wechselt von „blockiert“ nach „bereit“
4. Prozess terminiert

– Bei 1. und 4. *muss* Neuauswahl erfolgen

– Bei 2. und 3. *kann* Neuauswahl erfolgen



Kriterien für *Scheduling*-Strategien

- CPU-Auslastung
  - Möglichst zu 100% ausgelastete CPU
- Durchsatz
  - Möglichst hohe Anzahl bearbeiteter Prozesse pro Zeiteinheit
- Verweilzeit
  - Möglichst geringe Gesamtzeit des Prozesses in der Rechenanlage
- Wartezeit
  - Möglichst kurze Gesamtzeit, in der der Prozess im Zustand „bereit“ ist
- Antwortzeit
  - Möglichst kurze Reaktionszeit des Prozesses in interaktivem Betrieb

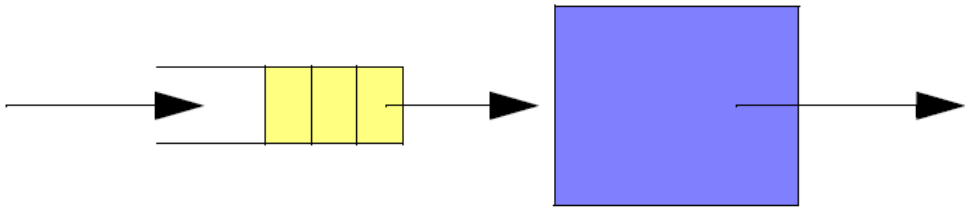
First Come, First Served (1)

Der erste Prozess wird zuerst bearbeitet (FCFS)

- „Wer zuerst kommt...“
- Nicht-präemptiv

Warteschlange zum Zustand „bereit“

- Prozesse werden hinten eingereiht
- Prozesse werden vorne entnommen



Bewertung

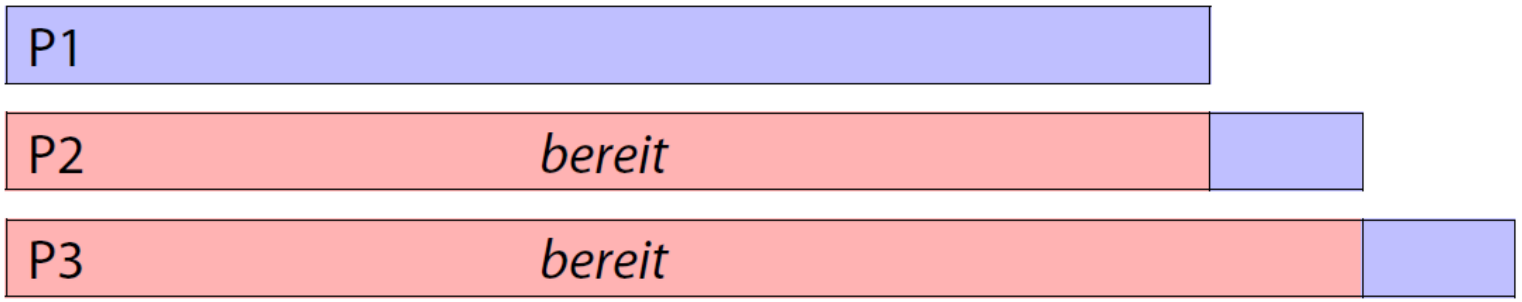
- Fair (?)
- Wartezeiten nicht minimal
- Nicht für *Time-Sharing* Betrieb geeignet

First Come, First Served (2)

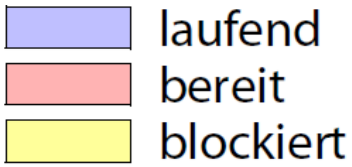
Beispiel zur Betrachtung der Wartezeiten

Prozess 1: 24
Prozess 2: 3
Prozess 3: 3
} Zeiteinheiten

– Reihenfolge: P1, P2, P3



Mittlere Wartezeit: (0 + 24 + 27) / 3 = 17



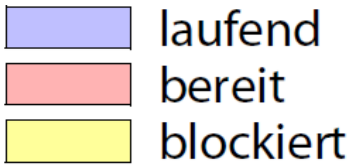
First Come, First Served (3)

Beispiel zur Betrachtung der Wartezeiten

Prozess 1: 24
Prozess 2: 3
Prozess 3: 3
} Zeiteinheiten

– Reihenfolge: P2, P3, P1

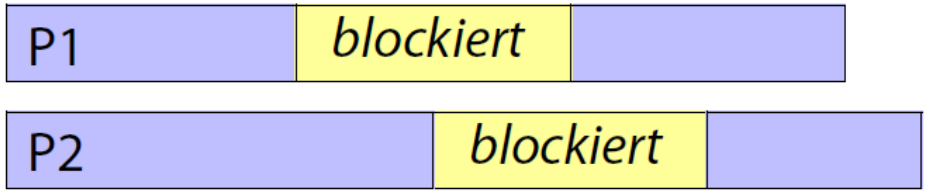
Mittlere Wartezeit: (6 + 0 + 3) / 3 = 3



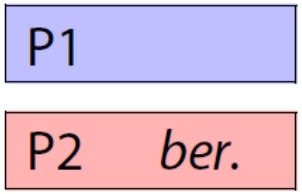
First Come, First Served (4)

Behandlung deblockierter Prozesse

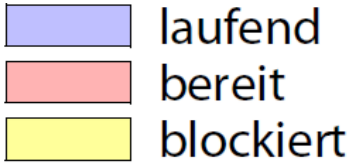
- Einreihung an das Ende der Warteschlange
- Beispiel
  - Laufzeitanforderung



  - FCFS-Ablauf auf einem Prozessor (initial: P1 vor P2)



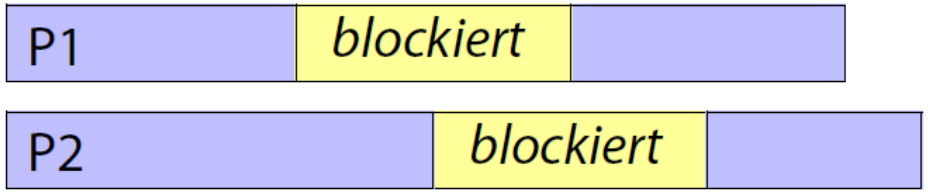
Warte-schlange: P1 P2 P2



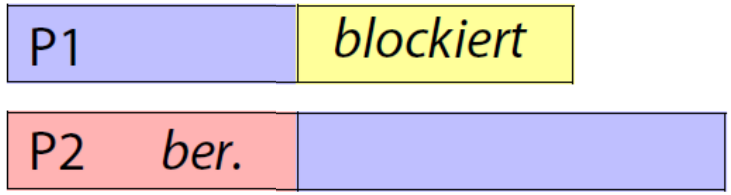
First Come, First Served (5)

Behandlung deblockierter Prozesse

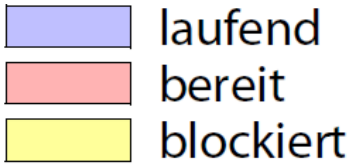
- Einreihung an das Ende der Warteschlange
- Beispiel
  - Laufzeitanforderung



  - FCFS-Ablauf auf einem Prozessor (initial: P1 vor P2)



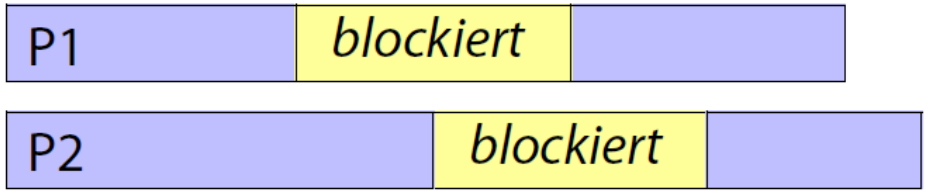
Warte-schlange: P1 P2 P1 P2



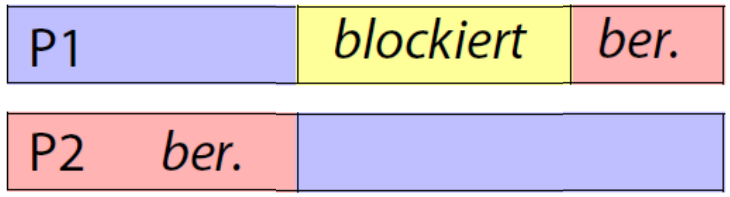
First Come, First Served (6)

Behandlung deblockierter Prozesse

- Einreihung an das Ende der Warteschlange
- Beispiel
  - Laufzeitanforderung



  - FCFS-Ablauf auf einem Prozessor (initial: P1 vor P2)



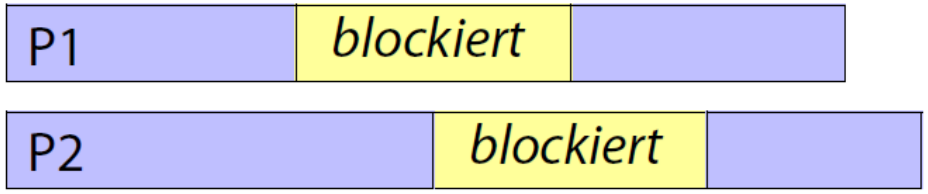
Warte-schlange: P1 P2 P1 P1 P2

laufend
bereit
blockiert

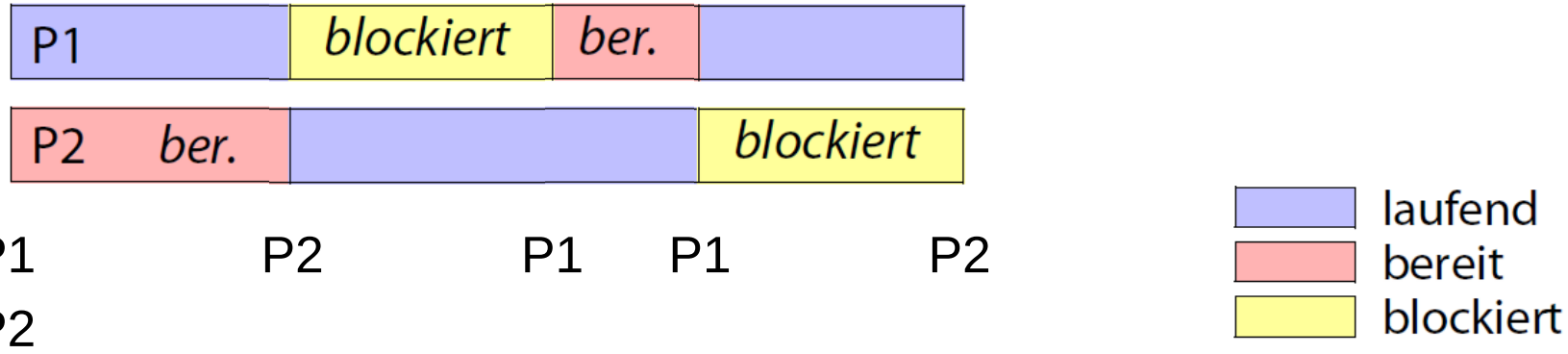
First Come, First Served (7)

Behandlung deblockierter Prozesse

- Einreihung an das Ende der Warteschlange
- Beispiel
  - Laufzeitanforderung



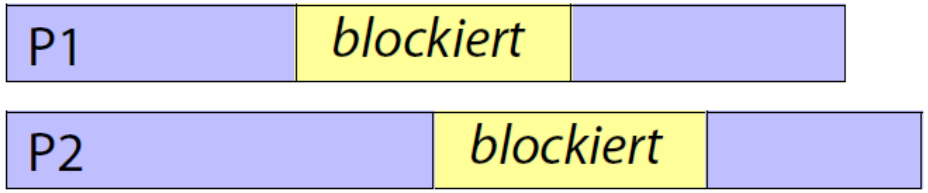
  - FCFS-Ablauf auf einem Prozessor (initial: P1 vor P2)



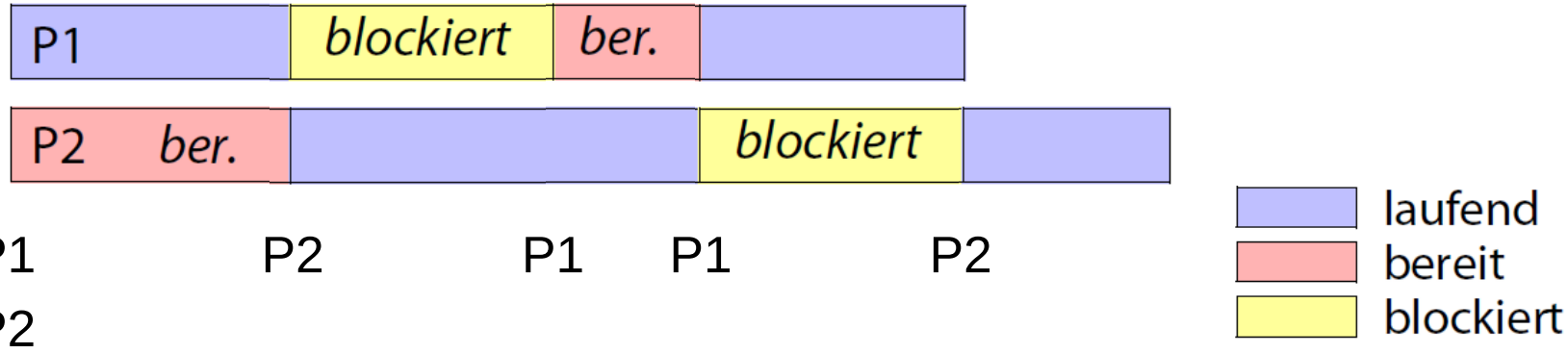
First Come, First Served (8)

Behandlung deblockierter Prozesse

- Einreihung an das Ende der Warteschlange
- Beispiel
  - Laufzeitanforderung



  - FCFS-Ablauf auf einem Prozessor (initial: P1 vor P2)



Shortest Job First (1)

Kürzester Job wird ausgewählt (SJF)

– Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. I/O)

„Bereit“-Warteschlange wird nach Länge der nächsten Rechenphase sortiert

– Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, exponentielle Glättung)

– ... Protokollierung der Länge der vorherigen Rechenphase

SJF optimiert die mittlere Wartezeit

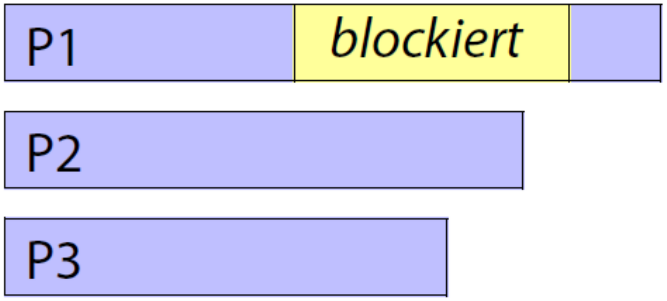
– Da Länge der Rechenphase in der Regel nicht genau vorhersagbar, nicht ganz optimal

Varianten: präemptiv (PSJF) und nicht-präemptiv

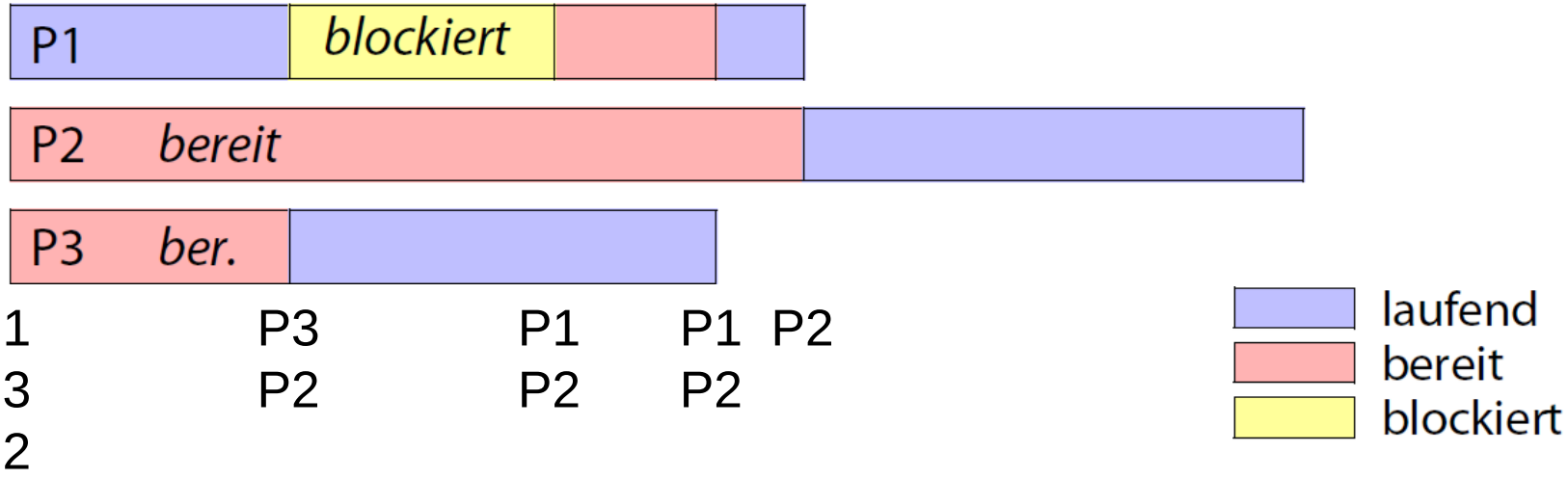
Shortest Job First (2)

Beispiel: SJF

– Laufzeitanforderung



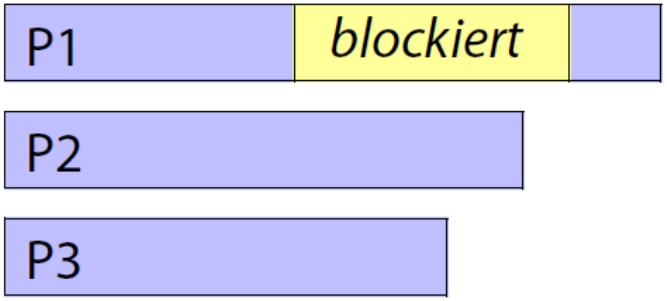
– SJF-Ablauf auf einem Prozessor



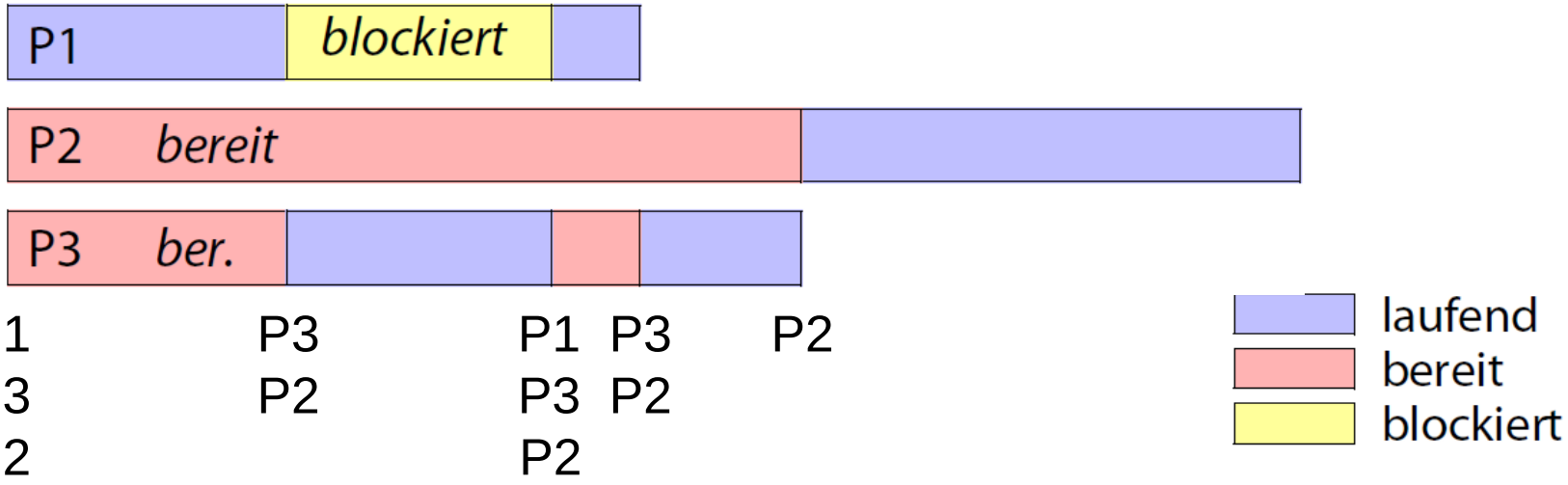
Shortest Job First (3)

Beispiel: PSJF

– Laufzeitanforderung



– PSJF-Ablauf auf einem Prozessor



Prioritäten (1)

Prozess mit höchster Priorität wird ausgewählt

(HPF, Highest Priority First)

– Dynamisch – statisch

(z.B. SJF: dynamische Vergabe von Prioritäten gemäß Länge der nächsten Rechenphase)

(z.B. statische Prioritäten in Echtzeitsystemen; Vorhersagbarkeit von Reaktionszeiten)

– Verdrängend – nicht-verdrängend

Probleme

– Prioritätsumkehr *(Priority Inversion)*

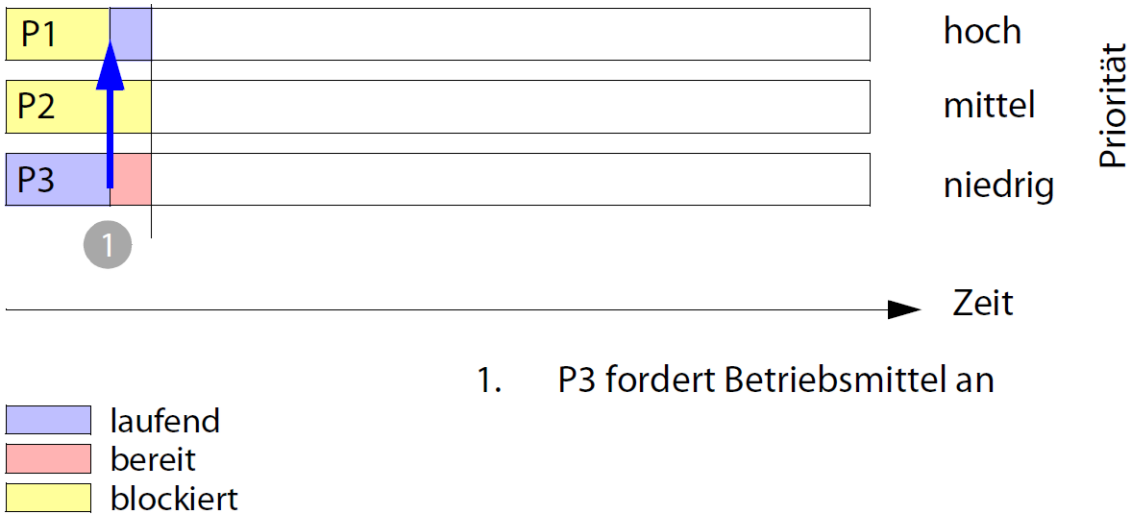
– Aushungerung *(Starvation)*

Prioritäten (2)

Prioritätsumkehr

– Hochpriorer Prozess wartet auf eine Ressource, die ein niedrigpriorer Prozess besitzt.
  Dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher die Ressource nicht freigeben

☞ Hochpriorer Prozess muss auf andere mit niedrigerer Priorität warten!

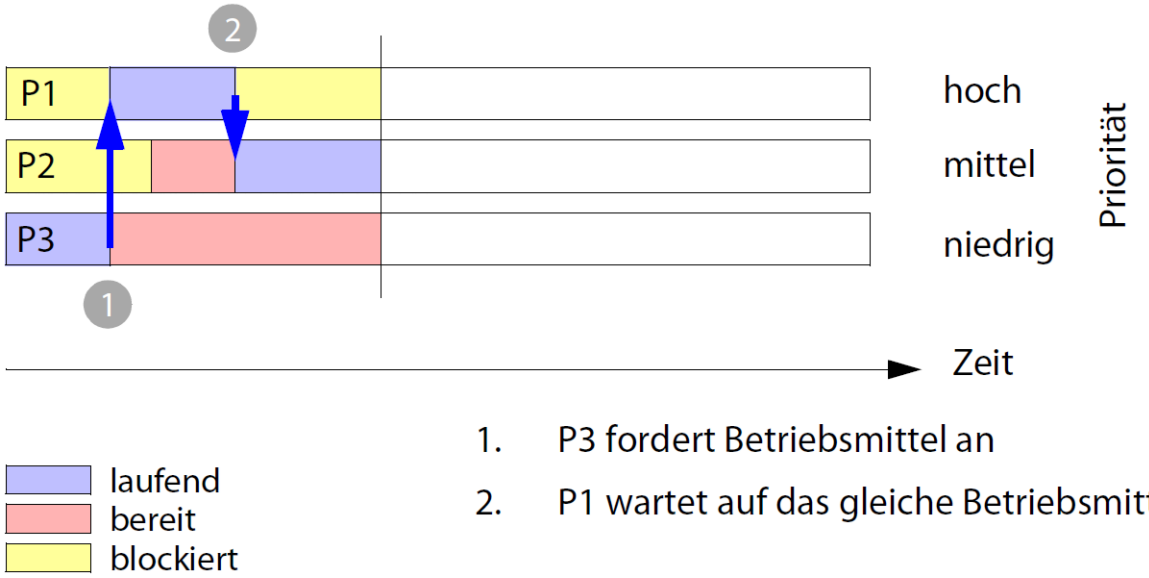


Prioritäten (3)

Prioritätsumkehr

– Hochpriorer Prozess wartet auf eine Ressource, die ein niedrigpriorer Prozess besitzt.
Dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher die Ressource nicht freigeben

☞ Hochpriorer Prozess muss auf andere mit niedrigerer Priorität warten!



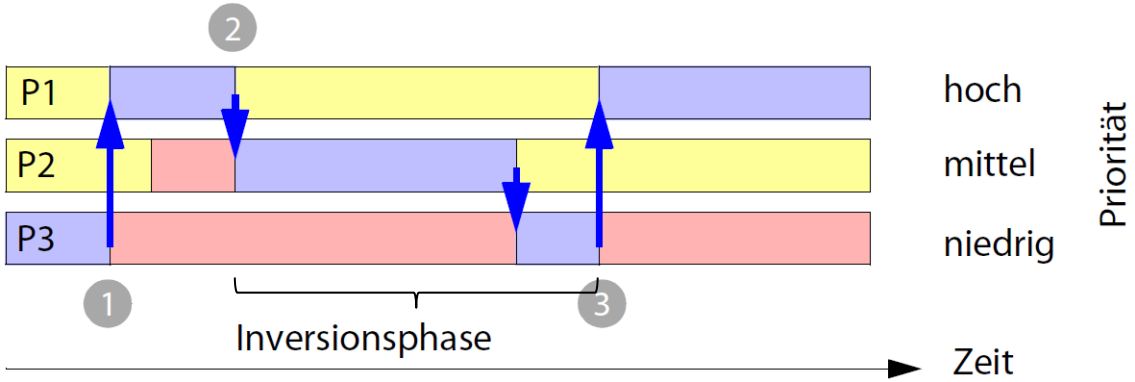
1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel

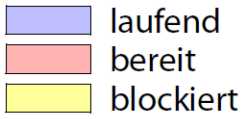
Prioritäten (4)

Prioritätsumkehr

– Hochpriorer Prozess wartet auf eine Ressource, die ein niedrigpriorer Prozess besitzt.
Dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher die Ressource nicht freigeben

☞ Hochpriorer Prozess muss auf andere mit niedrigerer Priorität warten!





1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei

Prioritäten (5)

Lösung: Prioritätsvererbung

– Dynamische Anhebung der Priorität für Prozesse, die eine Ressource besitzen, auf die ein hochpriorer Prozess wartet
  (Im Beispiel: P3 erhält bei Prioritätsvererbung zwischen Zeitpunkten 2 und 3 gleiche Priorität wie Prozess P1)

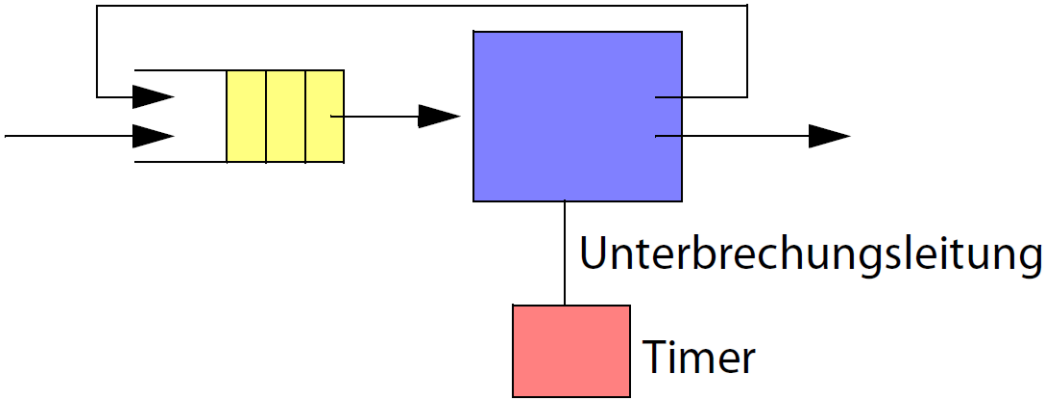
Aushungerung

– Ein Prozess kommt nie zum Zuge, da immer andere mit höherer Priorität vorhanden sind.
– Lösung: Dynamische Anhebung der Priorität für lange wartende Prozesse (Alterung, *Aging*)

Round Robin (1)

Zuteilung und Auswahl erfolgt reihum

- Ähnlich wie FCFS, aber mit Präemption
- Festes Zeitquantum *(Time Quantum)* oder Zeitscheibe *(Time Slice)* wird Prozessen jeweils zugeteilt
- Geeignet für *Time-Sharing* Betrieb



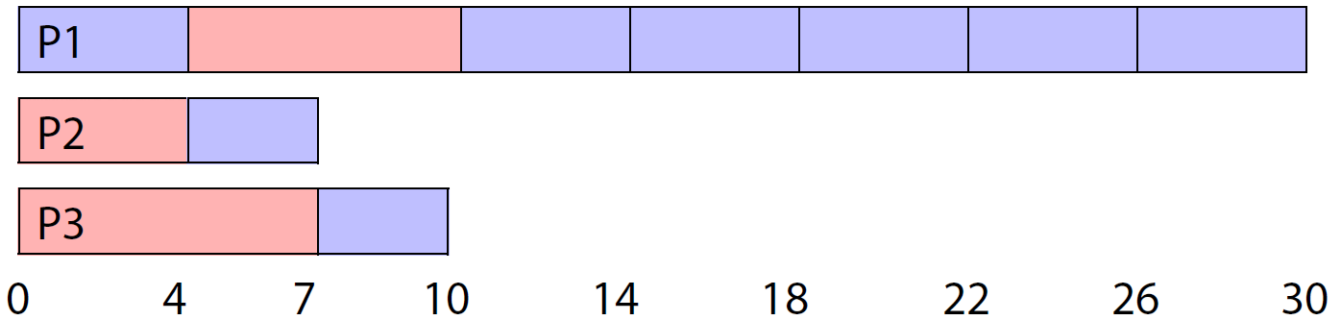
- Wartezeit ist jedoch eventuell relativ lang

Round Robin (2)

Beispiel zur Betrachtung der Wartezeiten

Prozess 1: 24
Prozess 2: 3
Prozess 3: 3
} Zeiteinheiten

– Zeitscheibe ist 4 Zeiteinheiten
– Reihenfolge in der „Bereit“-Warteschlange *(Ready Queue)*: P1, P2, P3



Mittlere Wartezeit: (6 + 4 + 7) / 3 = 5,7

Round Robin (3)

Effizienz hängt von der Länge der Zeitscheibe ab

- Kurze Zeitscheiben: Zeit zum Kontextwechsel wird dominant
- Lange Zeitscheiben: *Round Robin* nähert sich FCFS an

Implementierungsabhängige Details

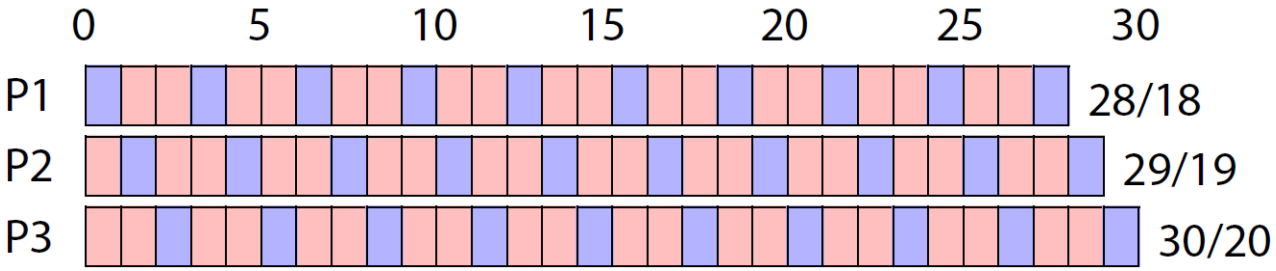
- Terminiert ein Prozess mitten in einer Zeitscheibe, kann das Ende der laufenden Zeitscheibe abgewartet und dann der Kontext gewechselt werden, oder der Kontextwechsel erfolgt vorzeitig direkt mit Prozessende
- Wird ein blockierter Prozess ausführbereit, so kann er hinten oder weiter vorne in die *Ready Queue* einsortiert werden

Verweilzeit und Wartezeit hängen ebenfalls von Zeitscheibenlänge ab

- Beispiel: 3 Prozesse mit je 10 Zeiteinheiten Rechenbedarf
  - Szenario 1: Zeitscheibenlänge 1
  - Szenario 2: Zeitscheibenlänge 10

Round Robin (4)

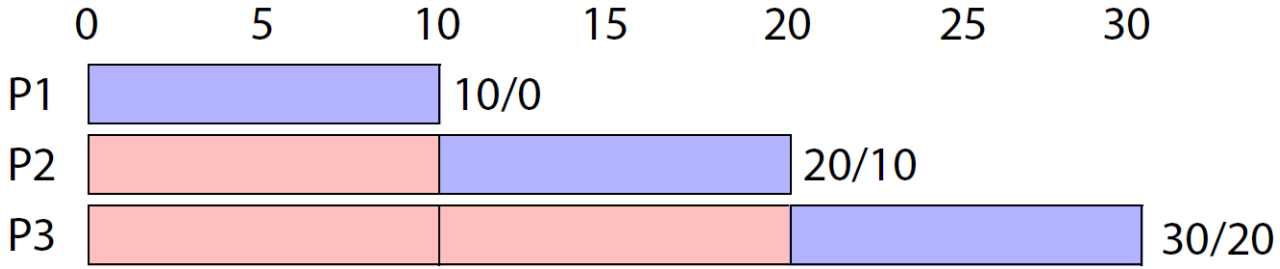
– **Szenario 1: Zeitscheibenlänge 1**



Verweil-/Wartezeit

Mittlere Verweilzeit: (28 + 29 + 30) / 3 = 29 Zeiteinheiten
Mittlere Wartezeit: (18 + 19 + 20) / 3 = 19 Zeiteinheiten

– **Szenario 2: Zeitscheibenlänge 10**



Verweil-/Wartezeit

Mittlere Verweilzeit: (10 + 20 + 30) / 3 = 20 Zeiteinheiten
Mittlere Wartezeit: (0 + 10 + 20) / 3 = 10 Zeiteinheiten

Multilevel-Queue Scheduling (1)

Verschiedene *Scheduling*-Klassen

- Z.B. Hintergrundprozesse *(Batch)* und Vordergrundprozesse (interaktive Prozesse)
- Jede Klasse besitzt ihre eigenen Warteschlangen *(Queues)* und verwaltet diese nach einer eigenen *Scheduling*-Strategie
- Zwischen den Klassen gibt es ebenfalls eine *Scheduling*-Strategie
  Z.B. feste Prioritäten (Vordergrundprozesse immer vor Hintergrundprozessen)
- Prozessauswahl
  1. *Scheduling*-Klasse auswählen
  2. In ausgewählter Klasse Prozess auswählen

Multilevel-Queue Scheduling (2)

Beispiel: Solaris / SunOS

- *Scheduling*-Klassen
  - Systemprozesse
  - *Real-Time* Prozesse
  - *Time-Sharing* Prozesse
  - Interaktive Prozesse
- *Scheduling* zwischen den Klassen mit fester Priorität
  (z.B. *Real-Time* Prozesse vor *Time-Sharing* Prozessen)

Multilevel-Queue Scheduling (3)

Beispiel: Solaris / SunOS *(fortges.)*

- Auswahlstrategien der *Scheduling*-Klassen
  - Systemprozesse: FCFS
  - *Real-Time* Prozesse: Statische Prioritäten
  - *Time-Sharing* und interaktive Prozesse:
    Komplexes Verfahren zur Gewährleistung von
    - Kurzen Reaktionszeiten
    - Fairer Zeitaufteilung zwischen rechenintensiven und I/O-intensiven Prozessen
    - Gewisser Benutzersteuerung

    realisiert mit *Multilevel-Feedback-Queue Scheduling*

Multilevel-Feedback-Queue Scheduling (1)

Ähnlich *Multilevel-Queue Scheduling*

- *Scheduling*-Klassen
  - Meist viele Klassen, doch innerhalb der Klassen gleiche Strategie (z.B. *Round Robin*)
- Strategie zur Auswahl der *Scheduling*-Klassen
  - Meist prioritätengesteuert mit statischen Prioritäten (z.B. 1. Klasse vor 2. Klasse)

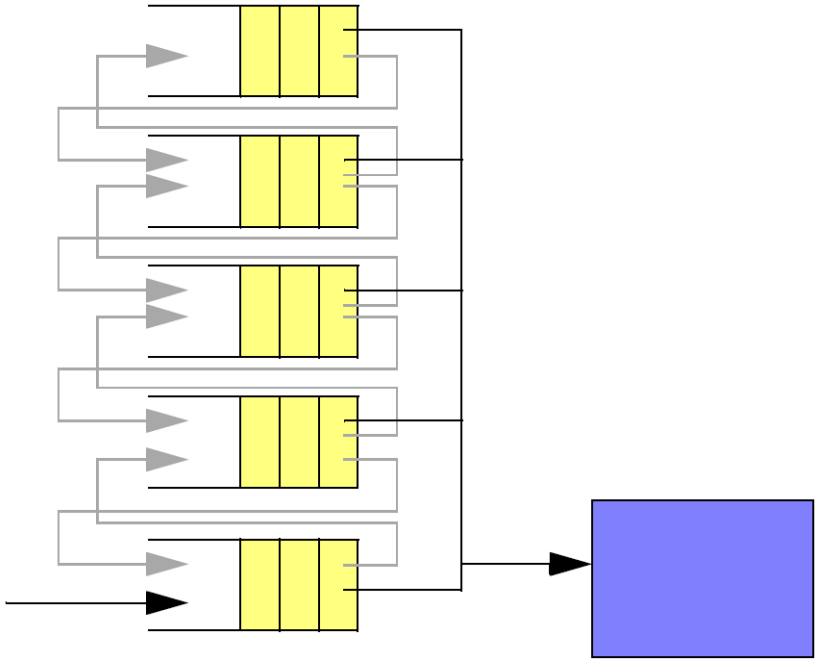
Zusätzlich bei *Multilevel-Feedback-Queue Scheduling (MLFB)*

- Transfer von Prozessen von einer Klasse in die andere *(Feedback)*

Multilevel-Feedback-Queue Scheduling (2)

Beispiel

- 5 *Scheduling*-Klassen mit eigener *Ready Queue* und Strategie
- Prozesse können von einer zur anderen *Queue* transferiert werden



Multilevel-Feedback-Queue Scheduling (3)

Beispiel: Solaris / SunOS

- 60 *Scheduling*-Klassen mit jeweils *Round Robin* Strategie
  - Unterschiedliche Zeitscheibenlängen
- Auswahl der Klasse über statische Priorität
  - Solange Prozesse in hochpriorer Klasse existieren, kommen keine anderen zum Zuge
- Transfer von Prozessen in andere Klassen/Prioritäten
  - Prozesse, die lange rechnen, wandern langsam in Klasse mit niedrigerer Priorität (bevorzugt interaktive Prozesse)
  - Prozesse, die lange warten müssen, wandern langsam wieder in höherpriore Klassen *(Aging)*

Roter Faden

4. Prozesse und Nebenläufigkeit

- Einordnung & Motivation
- Prozesse
- Auswahlstrategien *(Scheduling)*
  - *First Come First Served*
  - *Shortest Job First*
  - Prioritäten
  - *Round Robin*
  - *Multilevel-Queue Scheduling*
  - *Multilevel-Feedback-Queue Scheduling*
- Aktivitätsträger *(Threads)*
- Parallelität und Nebenläufigkeit
- Koordinierung

Zusammenfassung (Prozesse)

Mehrere Prozesse zur Strukturierung von Problemlösungen

- Aufgaben eines Prozesses leichter modellierbar, wenn in mehrere kooperierende Prozesse unterteilt
  z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
  z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
- Multiprozessor-Systeme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
  z.B. wissenschaftliches Hochleistungsrechnen (Klima-Simulation, Strömungssimulation etc.)
- *Client-Server* Anwendungen unter UNIX: pro Anfrage wird ein neuer Prozess gestartet
  z.B. Webserver

Prozesse mit gemeinsamem Speicher

Kommunikation zwischen kooperierenden Prozessen

– Üblicherweise über gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse

Vorteile

– Sehr einfach zu programmieren
– In Multiprozessor-Systemen sind echt parallele Abläufe möglich

Nachteile

– Viele Ressourcen sind zur Verwaltung eines Prozesses notwendig
  – *File*-Deskriptoren
  – Speicherabbildung
  – Prozesskontrollblock
– Prozesswechsel sind aufwändig

Aktivitätsträger (1)

Alternative:

Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(lightweight processes, LWPs)*

- Eine Gruppe von *Threads* nutzt gemeinsam eine Menge von Ressourcen
  - Instruktionen
  - Datenbereiche
  - *Files*, *Sockets* etc.
- Jeder *Thread* repräsentiert eine eigene Aktivität
  - Eigener Programmzähler
  - Eigener Registersatz
  - Eigener *Stack*

Aktivitätsträger (2)

- Umschalten zwischen zwei *Threads* einer Gruppe ist erheblich billiger als ein normaler Kontextwechsel
  - Es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
  - Speicherabbildung muss nicht gewechselt werden
  - Alle Systemressourcen bleiben verfügbar

Linux/UNIX-Prozesse

- Sind Adressräume mit zunächst einem *Thread*
- Weitere *Threads* können dynamisch zur Laufzeit erzeugt werden

Implementierungen von Threads

- *User-level Threads*
- *Kernel-level Threads*

User-level Threads

Implementierung

– Instruktionen im Anwendungsprogramm schalten zwischen den *Threads* hin- und her (ähnlich wie der *Scheduler* im Betriebssystem)
– Betriebssystem sieht nur einen *Thread*
– Beispiel: Java Green Threads, Windows Fibers, Solaris-Threads

Vorteile

– Keine Systemaufrufe zum Umschalten von *Threads* erforderlich
– Effiziente Umschaltung
– *Scheduling*-Strategie in der Hand des Anwenders

Nachteile

– Bei blockierenden Systemaufrufen bleiben alle *User-level Threads* eines Programms stehen
– Kein Ausnutzen eines Multiprozessors möglich

Kernel-level Threads

Implementierung

– Betriebssystem kennt *Kernel-level Threads*
– Betriebssystem schaltet *Threads* um

Vorteile

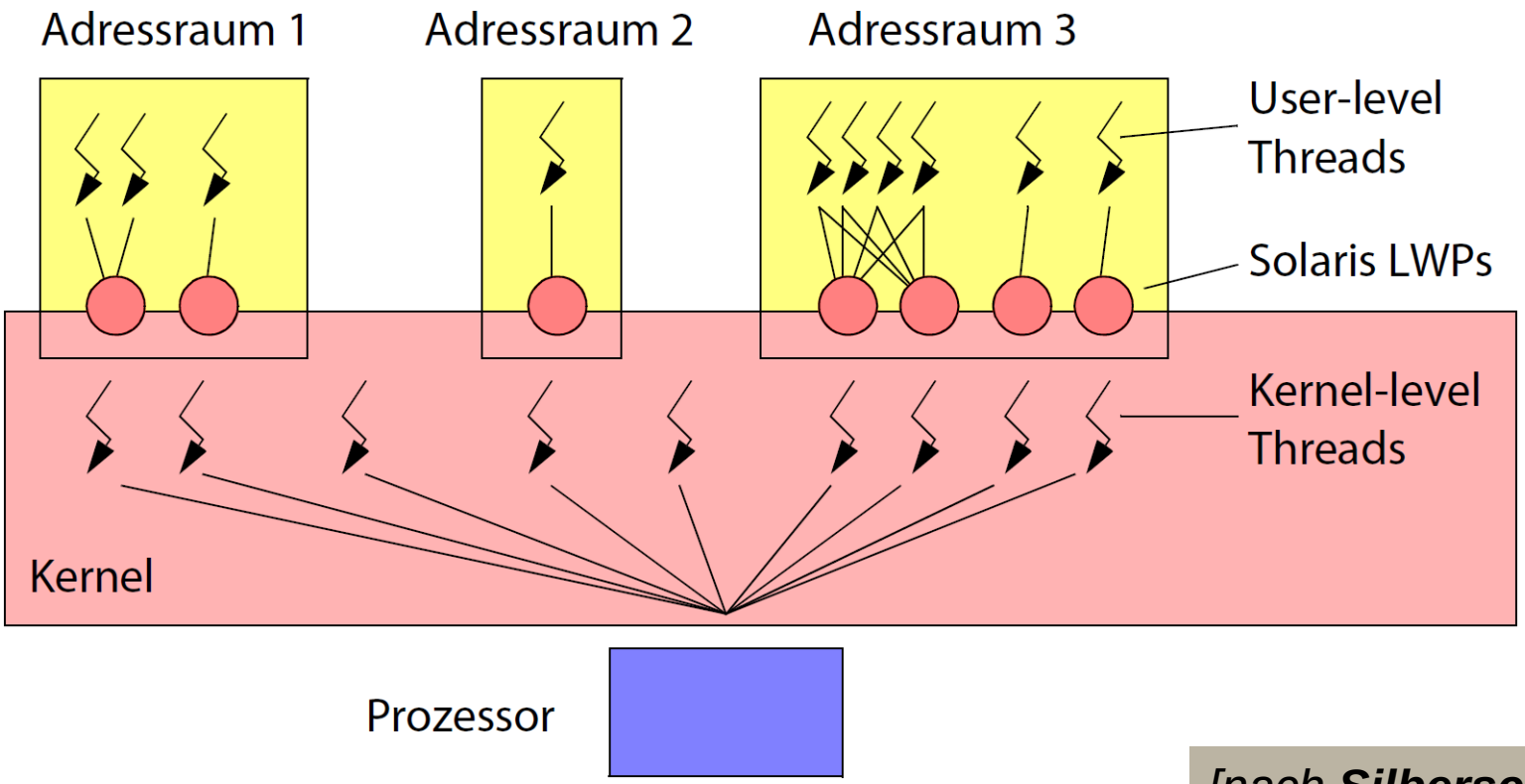
– Kein Blockieren unbeteiligter *Threads* bei blockierenden Systemaufrufen

Nachteile

– Weniger effizientes Umschalten
– Fairnessverhalten nötig
  (zwischen Prozessen mit vielen und solchen mit wenigen *Threads*)
– *Scheduling*-Strategie meist vorgegeben

Beispiel: LWPs und *Threads* (Solaris)

Solaris kennt *Kernel-*, *User-level Threads* und LWPs



*[nach **Silberschatz,** 1994]*

[http://en.wikipedia.org/wiki/Light-weight_process]

Roter Faden

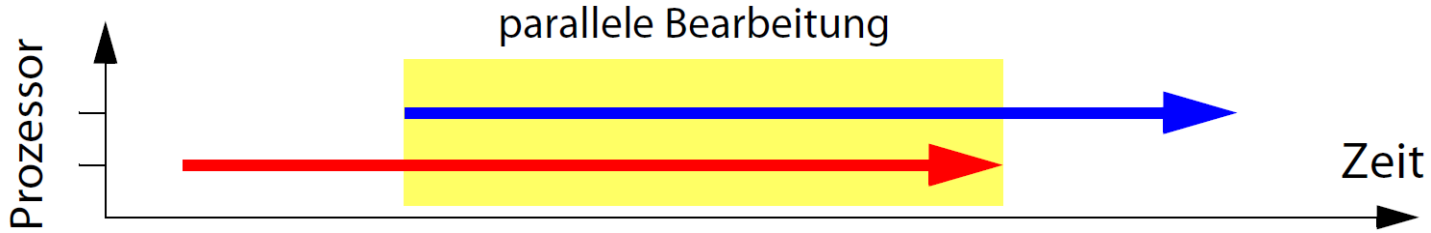
4. Prozesse und Nebenläufigkeit

- Einordnung & Motivation
- Prozesse
- Auswahlstrategien *(Scheduling)*
- Aktivitätsträger *(Threads)*
  - *User-level Threads*
  - *Kernel-level Threads*
  - *Lightweight Processes*
- Parallelität und Nebenläufigkeit
- Koordinierung

Parallelität und Nebenläufigkeit (1)

Parallelität bei mehreren Prozessen oder *Threads*

- Die Anweisungen zweier Prozesse werden *parallel* bearbeitet, wenn die Anweisungen unabhängig voneinander zur gleichen Zeit ausgeführt werden
- Parallele Bearbeitung *nur* auf Multiprozessoren
  - Zwei oder mehr Prozessoren können parallel Anweisungen bearbeiten

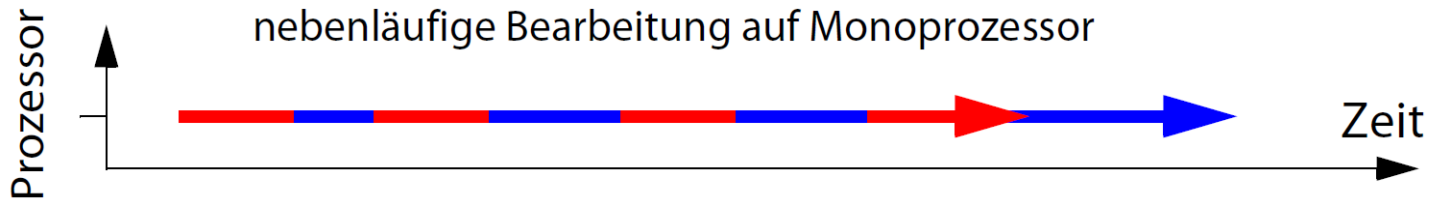


- Monoprozessor
  - Keine parallele Abarbeitung von Befehlen möglich

Parallelität und Nebenläufigkeit (2)

Nebenläufigkeit

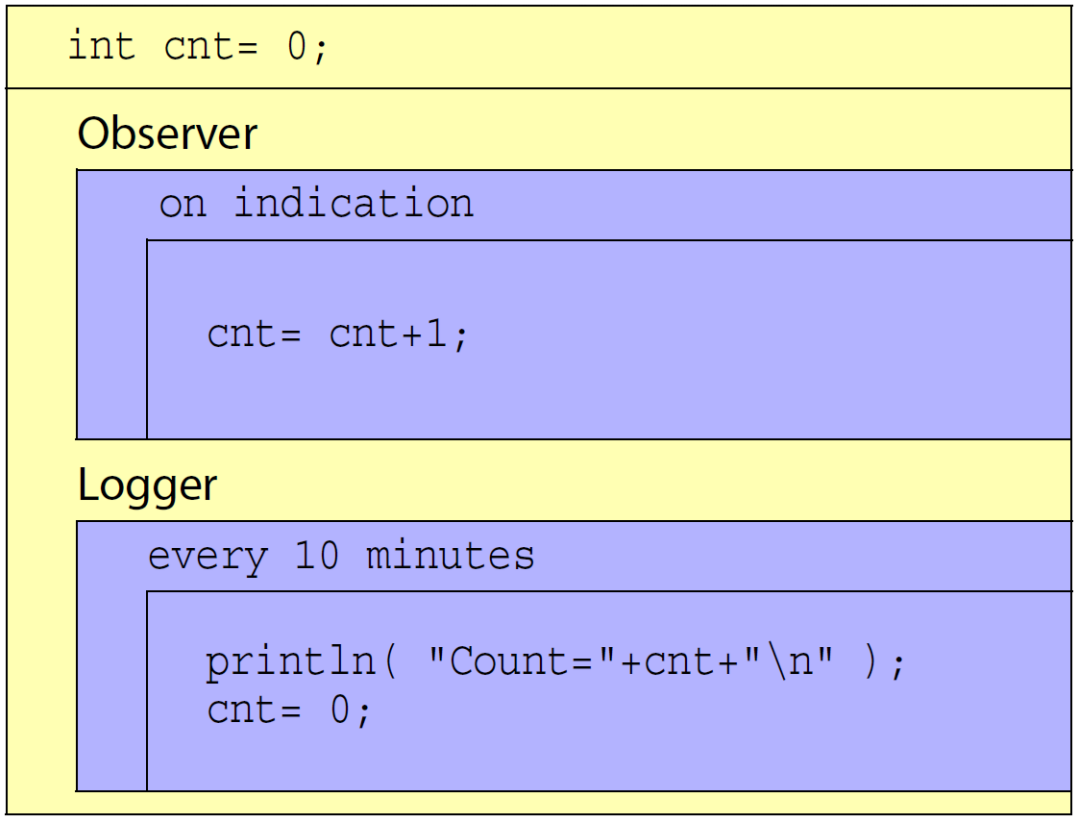
- Zwei Prozesse heißen *nebenläufig*, wenn ihre Anweisungen unabhängig voneinander abgearbeitet werden. Dabei spielt es keine Rolle, ob die Anweisungen zeitlich durchmischt oder auch echt gleichzeitig bearbeitet werden.
- Parallele Bearbeitung ist nebenläufig
- Nebenläufige Bearbeitung auch auf Monoprozessoren möglich
  - Zeitliche Durchmischung der Befehle mehrerer Prozesse / *Threads*
  - *Scheduling*-Strategie oft unabhängig von den bearbeiteten Befehlen (z.B. Round Robin, MLFB)



Koordinierung (1)

Beispiel: zwei Prozesse – Beobachter und Protokollierer

– Per Induktionsschleife werden Fahrzeuge gezählt. Alle 10 Minuten druckt der Protokollierer die im letzten Zeitraum vorbeigekommene Anzahl aus.



Koordinierung (2)

Effekte

– Fahrzeuge gehen „verloren“
– Fahrzeuge werden doppelt gezählt

Ursachen

– Befehle einer Programmiersprache werden nicht unteilbar (atomar) abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden
– Keinesfalls werden mehrere Anweisungen zusammen atomar abgearbeitet
– Prozesswechsel innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen können zu Inkonsistenzen führen

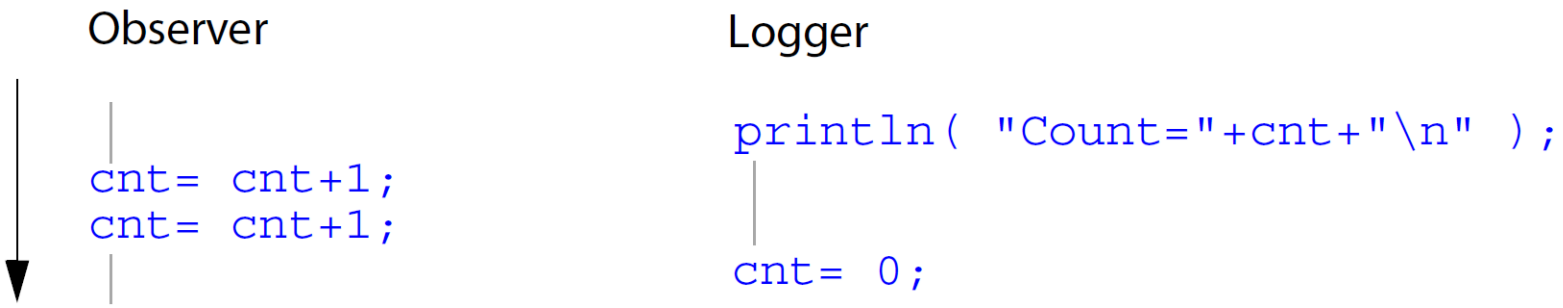
Koordinierung (3)

Fahrzeuge gehen „verloren“

– Nach dem Drucken wird der Protokollierer unterbrochen.

Beobachter zählt weitere Fahrzeuge.

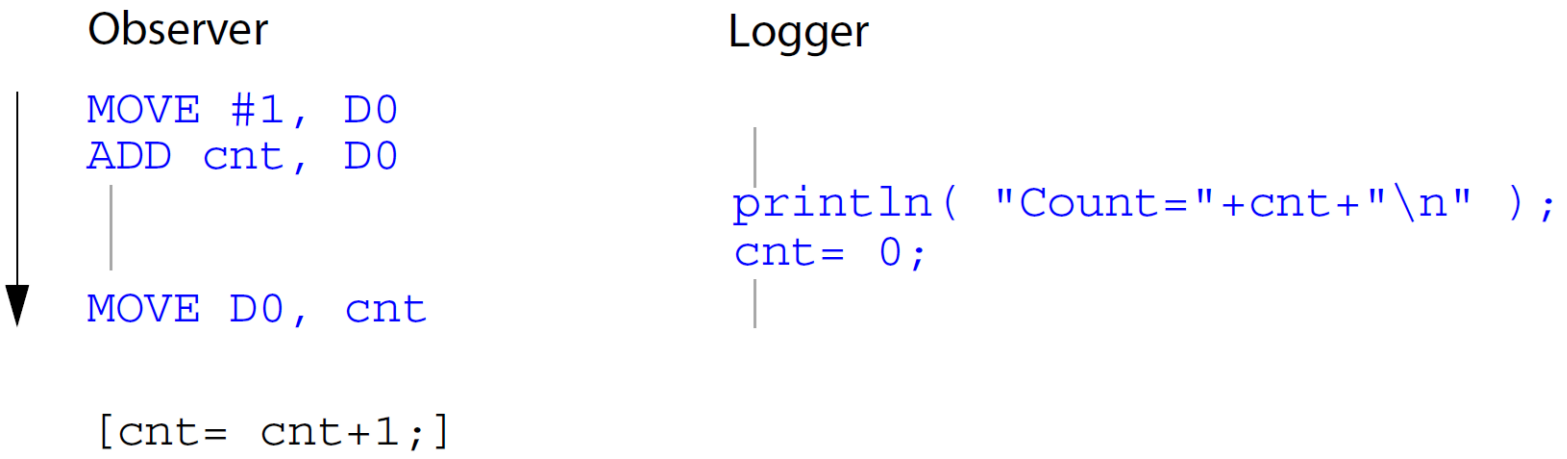
Zähler wird danach ohne Beachtung vom *Logger* auf Null gesetzt.



Koordinierung (4)

Fahrzeuge werden doppelt gezählt

– Beobachter will Zähler erhöhen und holt sich diesen dazu in ein Register.
Beobachter wird unterbrochen, *Logger* setzt Zähler auf Null.
Beobachter erhöht Registerwert und schreibt diesen zurück. Dieser Wert wird erneut vom Protokollierer registriert.



Koordinierung (5)

Problem gekoppelt an gemeinsame Nutzung von Daten / Ressourcen

Lösung: Kritische Abschnitte *(Critical Sections)*

– Nur ein Prozess/*Thread* soll Zugang zu Daten/Ressourcen haben (wechselseitiger Ausschluss, *Mutual Exclusion*, *Mutex*)
– Kritische Abschnitte erscheinen als zeitlich unteilbar

Wie kann der wechselseitige Ausschluss in kritischen Abschnitten erzielt werden?

☞ Vorkehrungen, dass nicht mehrere Prozesse gleichzeitig im kritischen Abschnitt sind

Koordinierung (6)

Koordinierung allgemein

– Einschränkung der zeitlichen Durchmischung von Befehlsfolgen in nebenläufigen Prozessen/*Threads*
– Letztlich Einschränkung der Nebenläufigkeit

☞ Gezielte Aufgabe der Unabhängigkeit bei der Befehlsausführung

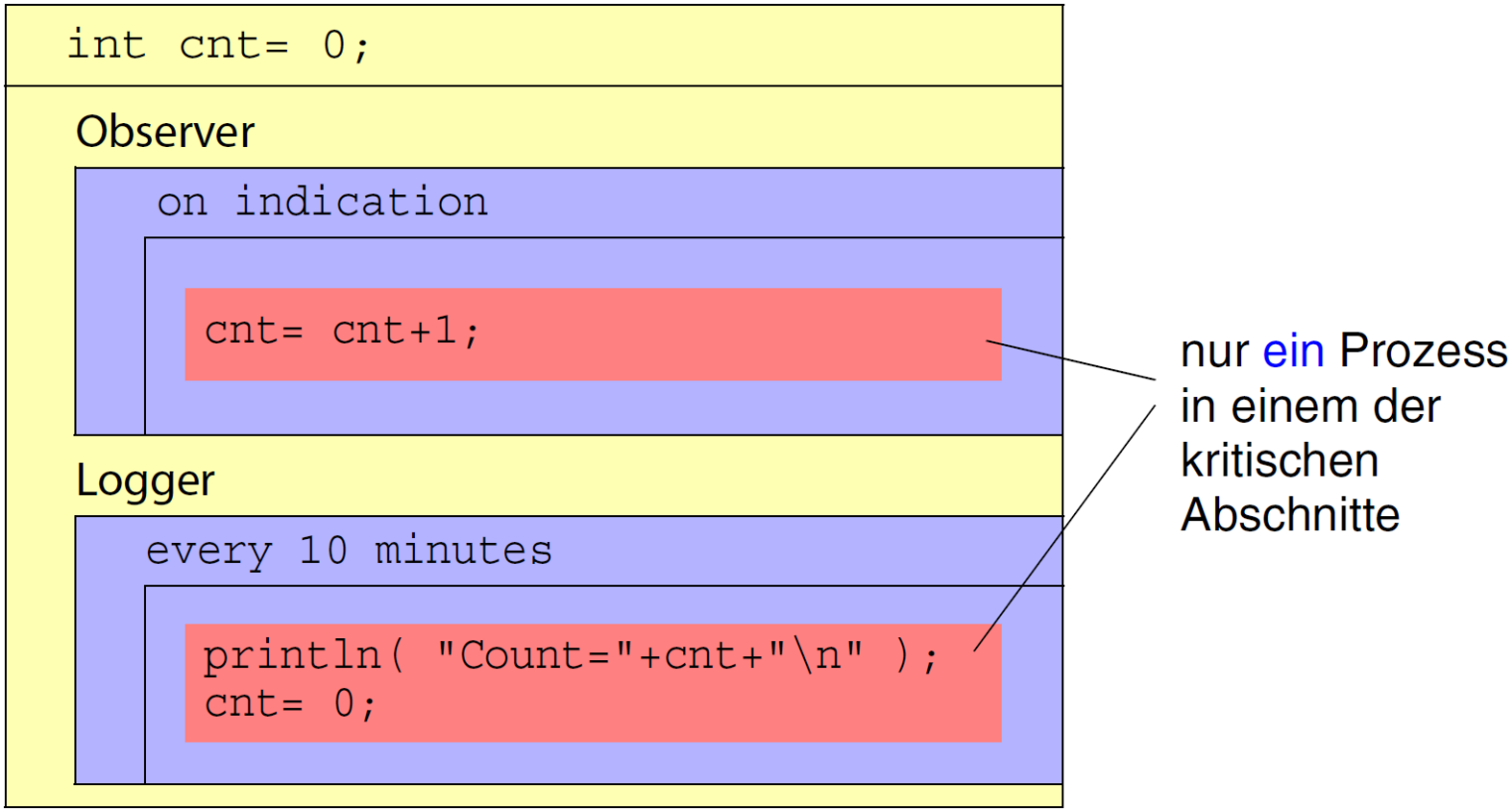
Hinweis

☞ Im Folgenden wird immer von Prozessen die Rede sein. Koordinierung kann/muss selbstverständlich auch zwischen *Threads* stattfinden.

Wechselseitiger Ausschluss (1)

Lösung des Beispiels

– Zwei zusammengehörige kritische Abschnitte



Wechselseitiger Ausschluss (2)

Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten

– Annahme: Maschinenbefehle sind unteilbar (atomar)

1. Versuch

```
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    while( turn == 1 );
    ...
    /* critical sec. */
    ...
    turn= 1;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    while( turn == 0);
    ...
    /* critical sec. */
    ...
    turn= 0;

    ... /* uncritical */
}
```

Wechselseitiger Ausschluss (3)

Probleme der Lösung

– Nur alternierendes Betreten des kritischen Abschnitts durch P_0 und P_1 möglich
– Implementierung ist unvollständig
– Aktives Warten

Ersetzen von `turn` durch zwei Variablen `ready0` und `ready1`

– **`ready0`** zeigt an, dass Prozess 0 bereit für den kritischen Abschnitt ist
– **`ready1`** zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist

Wechselseitiger Ausschluss (4)

2. Versuch

```
boolean ready0= false;
boolean ready1= false;
```

Prozess 0

```
while( 1 ) {
    ready0= true;
    while( ready1 );

    ... /* critical sec. */

    ready0= false;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= true;
    while( ready0 );

    ... /* critical sec. */

    ready1= false;

    ... /* uncritical */
}
```

Wechselseitiger Ausschluss (5)

Wechselseitiger Ausschluss wird erreicht

– Leicht nachweisbar durch Zustände von **`ready0`** und **`ready1`**

Probleme der Lösung

– Aktives Warten

– Verklemmung *(Deadlock)* möglich

Wechselseitiger Ausschluss (6)

Betrachtung der nebenläufigen Abfolgen

P_0

```
ready0= true;
while( ready1 ); +
<critical>       +
ready0= false;
<noncritical>    +
ready0= true;
...
```

P_1

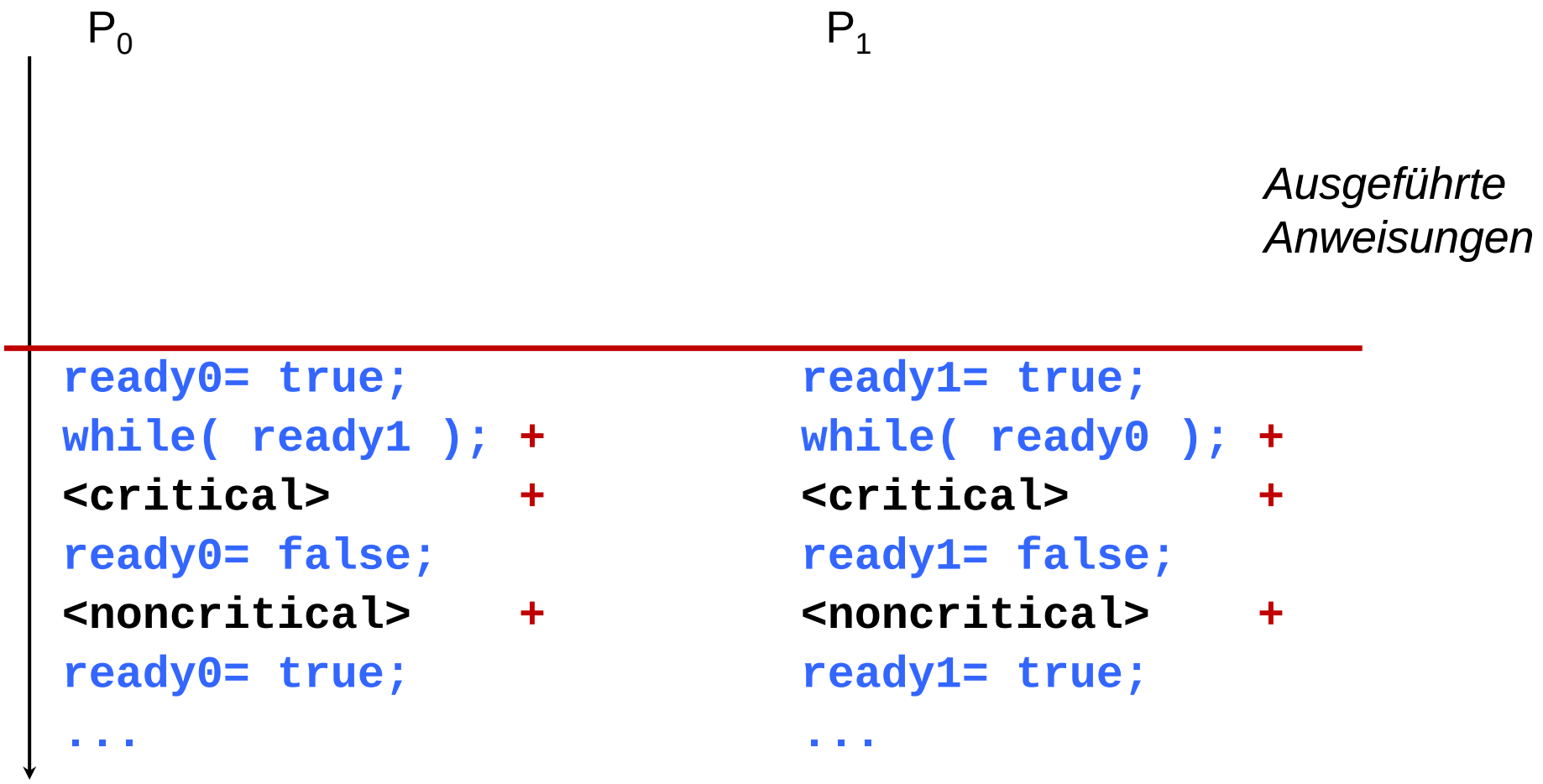
```
ready1= true;
while( ready0 ); +
<critical>       +
ready1= false;
<noncritical>    +
ready1= true;
...
```

+ = mehrfach, mind. einmal
***** = mehrfach oder gar nicht

– Durchspielen aller möglichen Durchmischungen

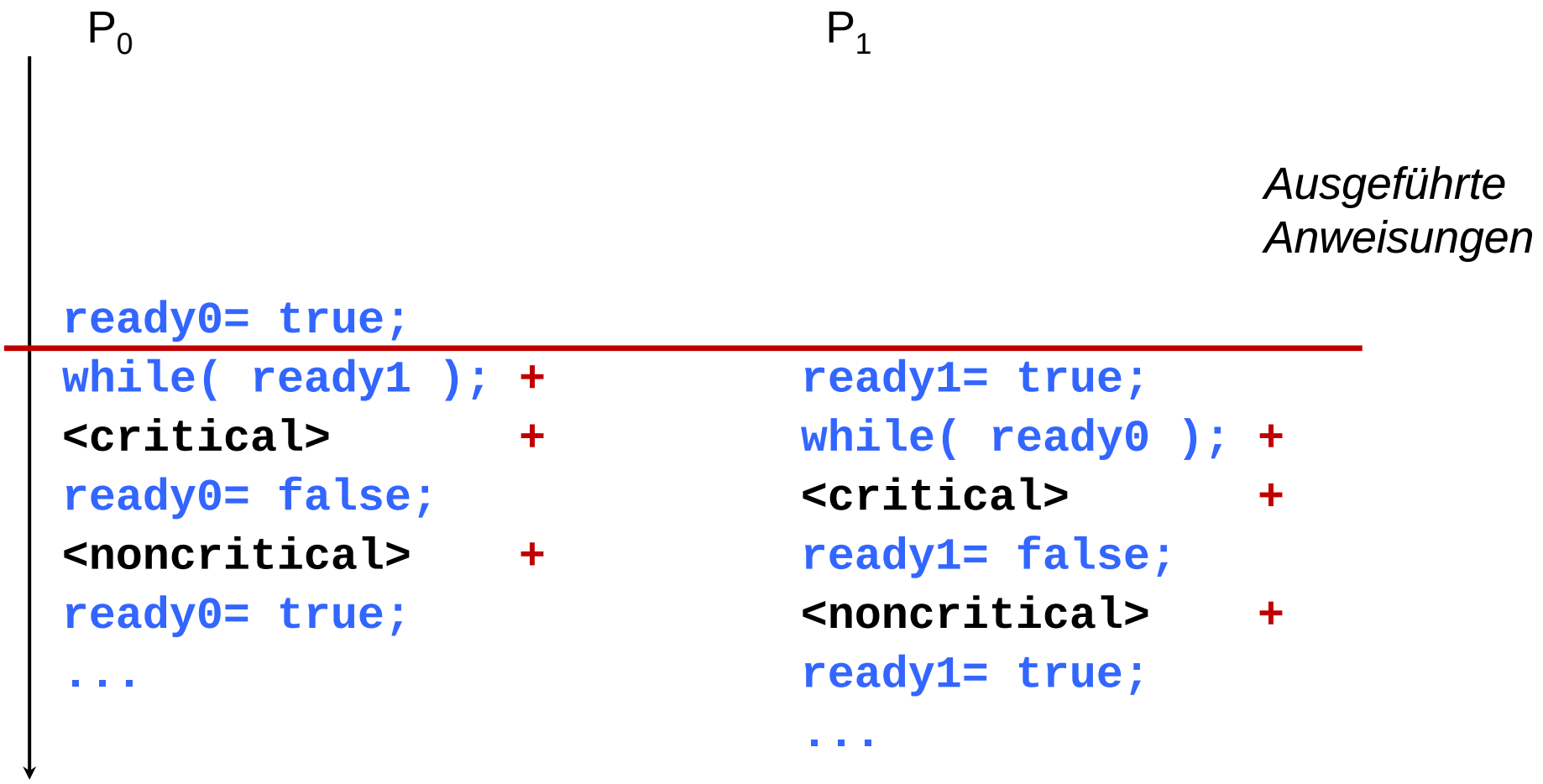
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



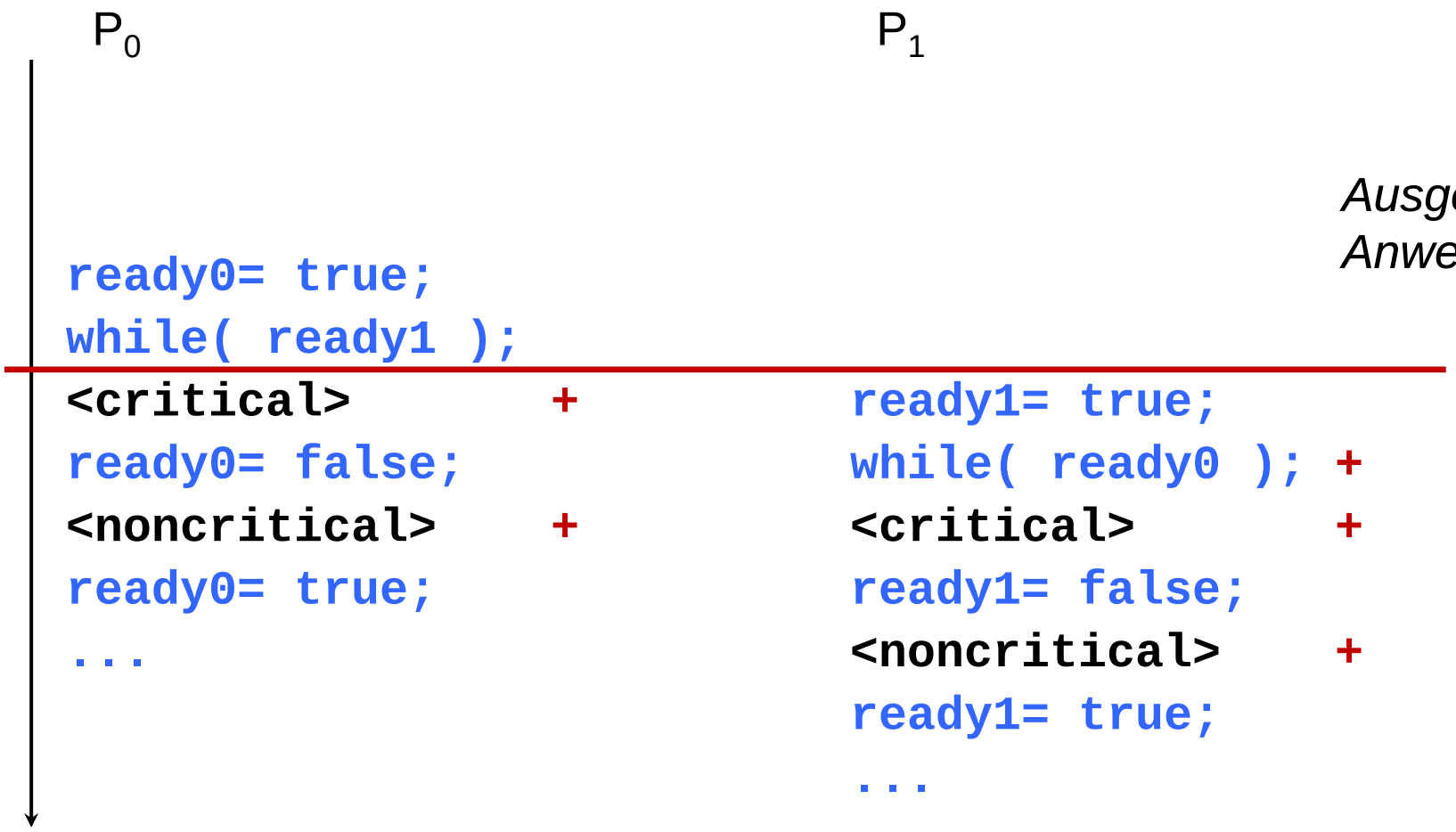
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



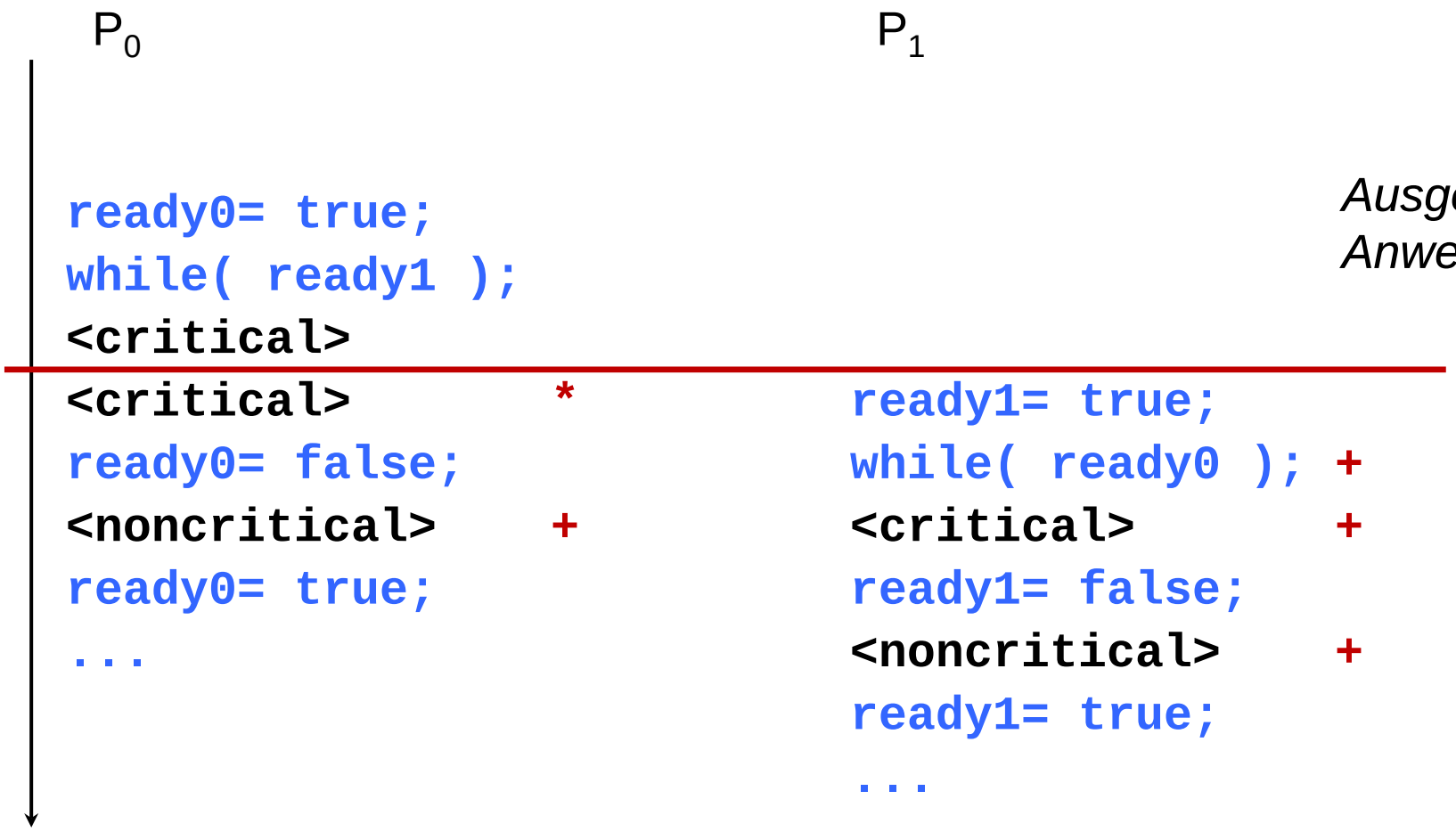
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



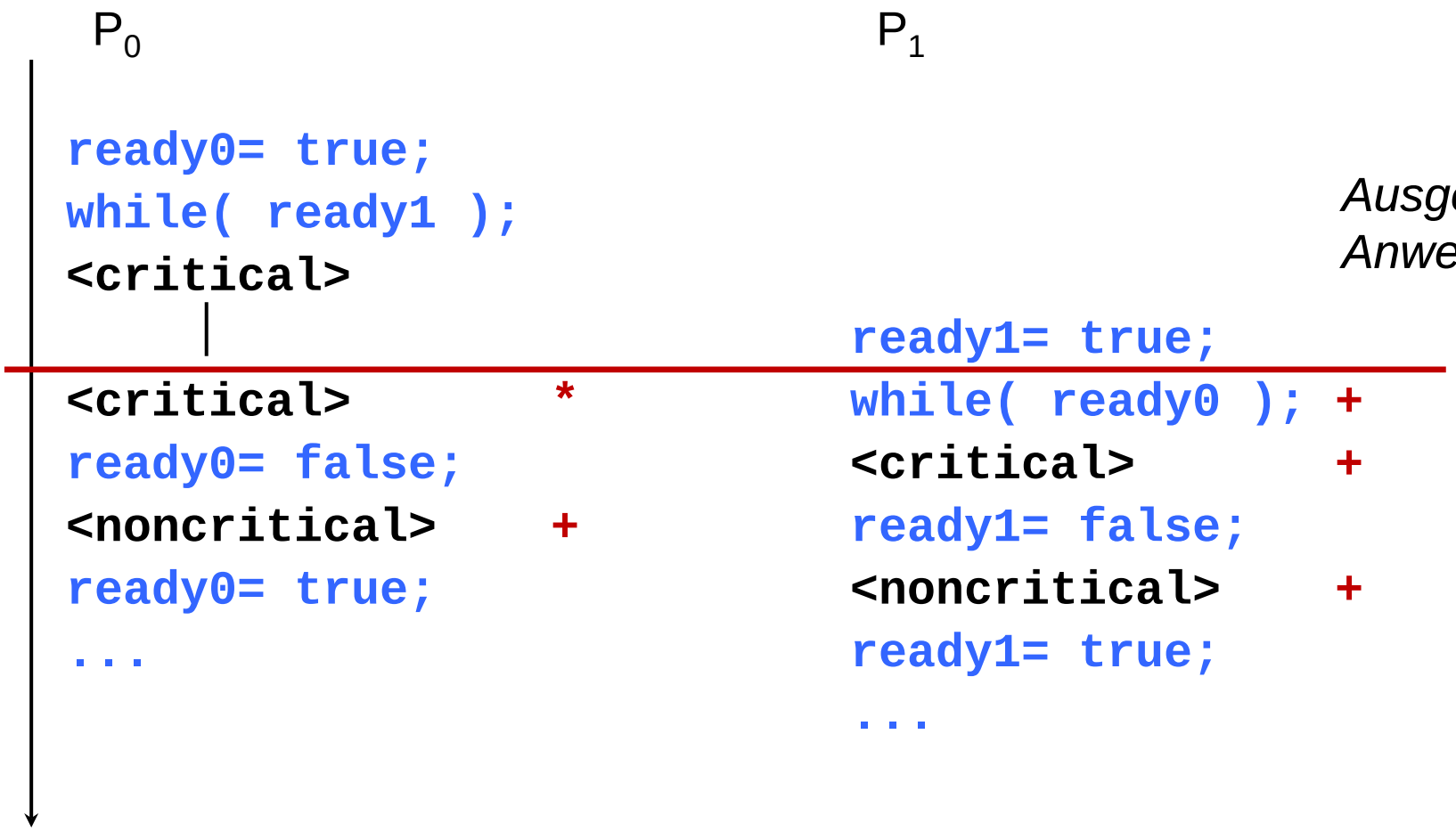
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



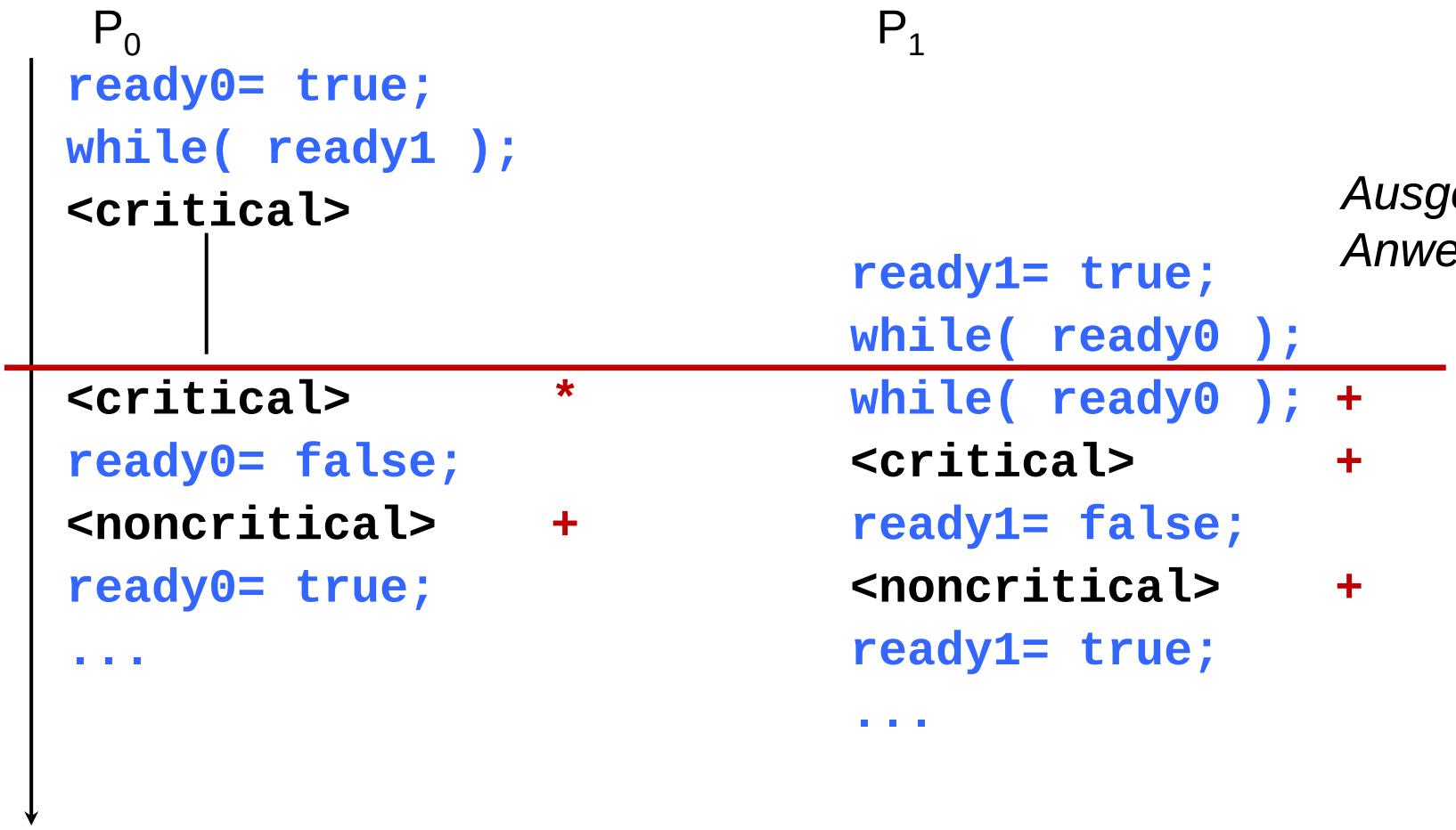
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



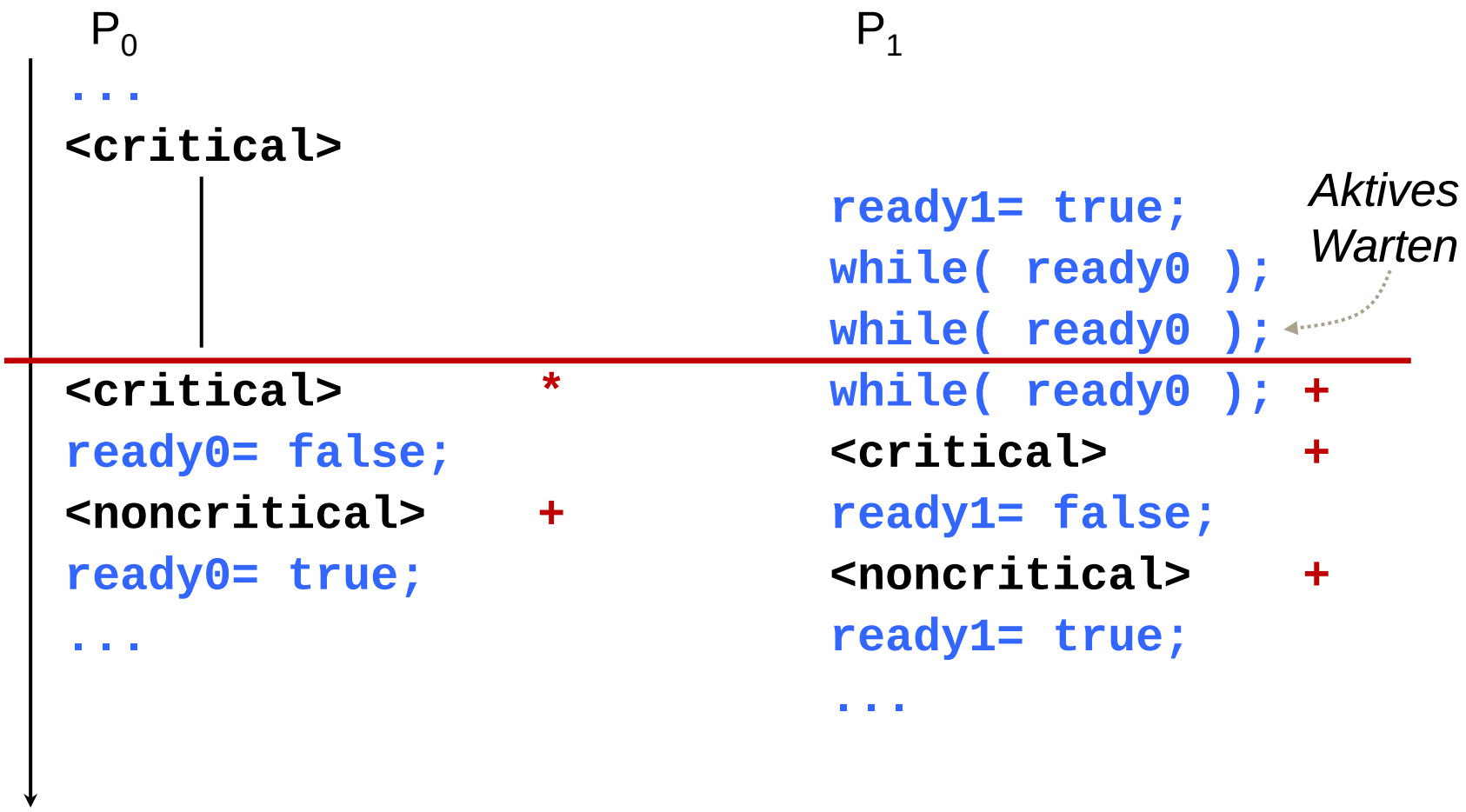
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



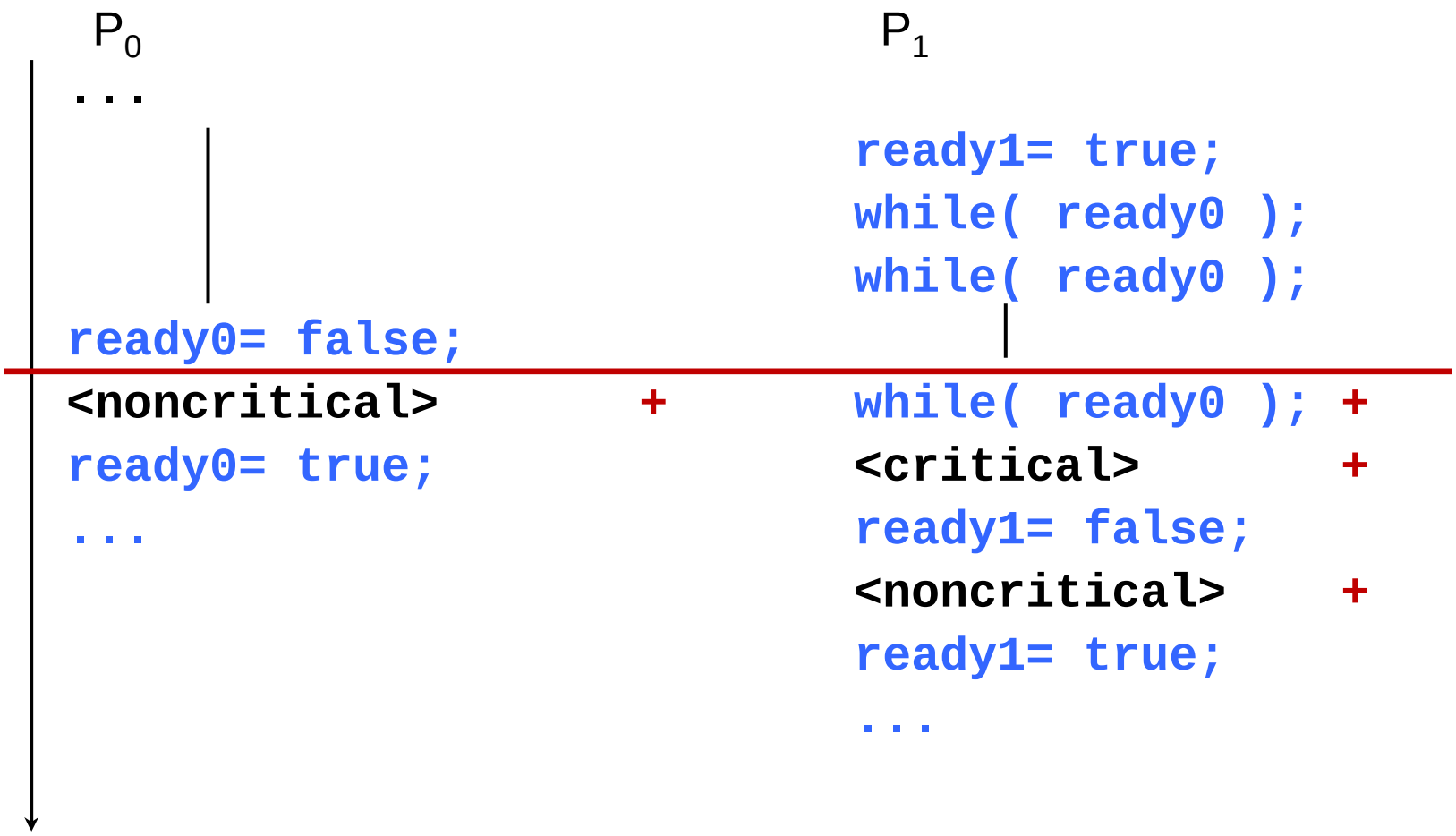
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



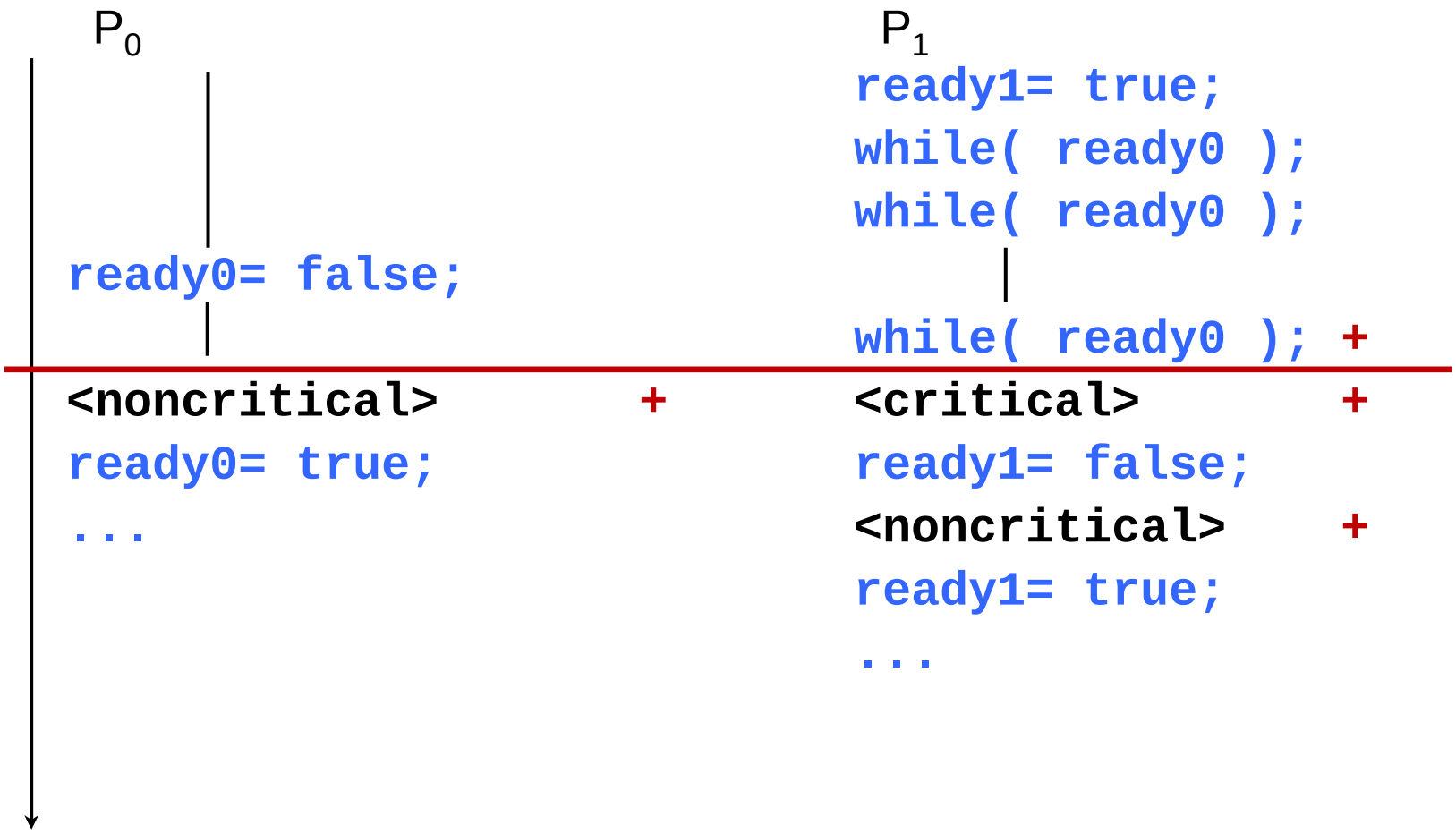
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



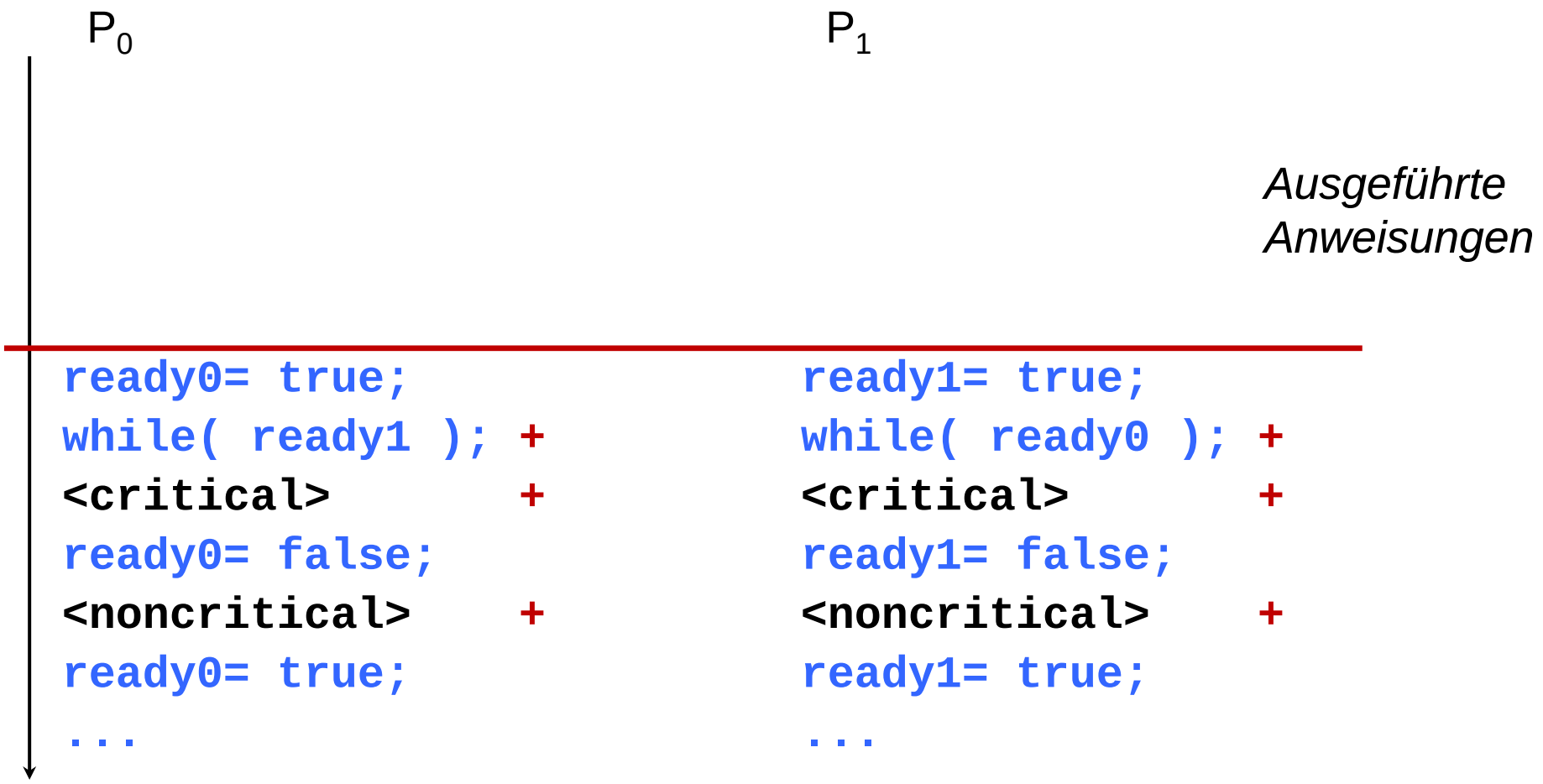
Wechselseitiger Ausschluss (7)

Harmlose Durchmischung



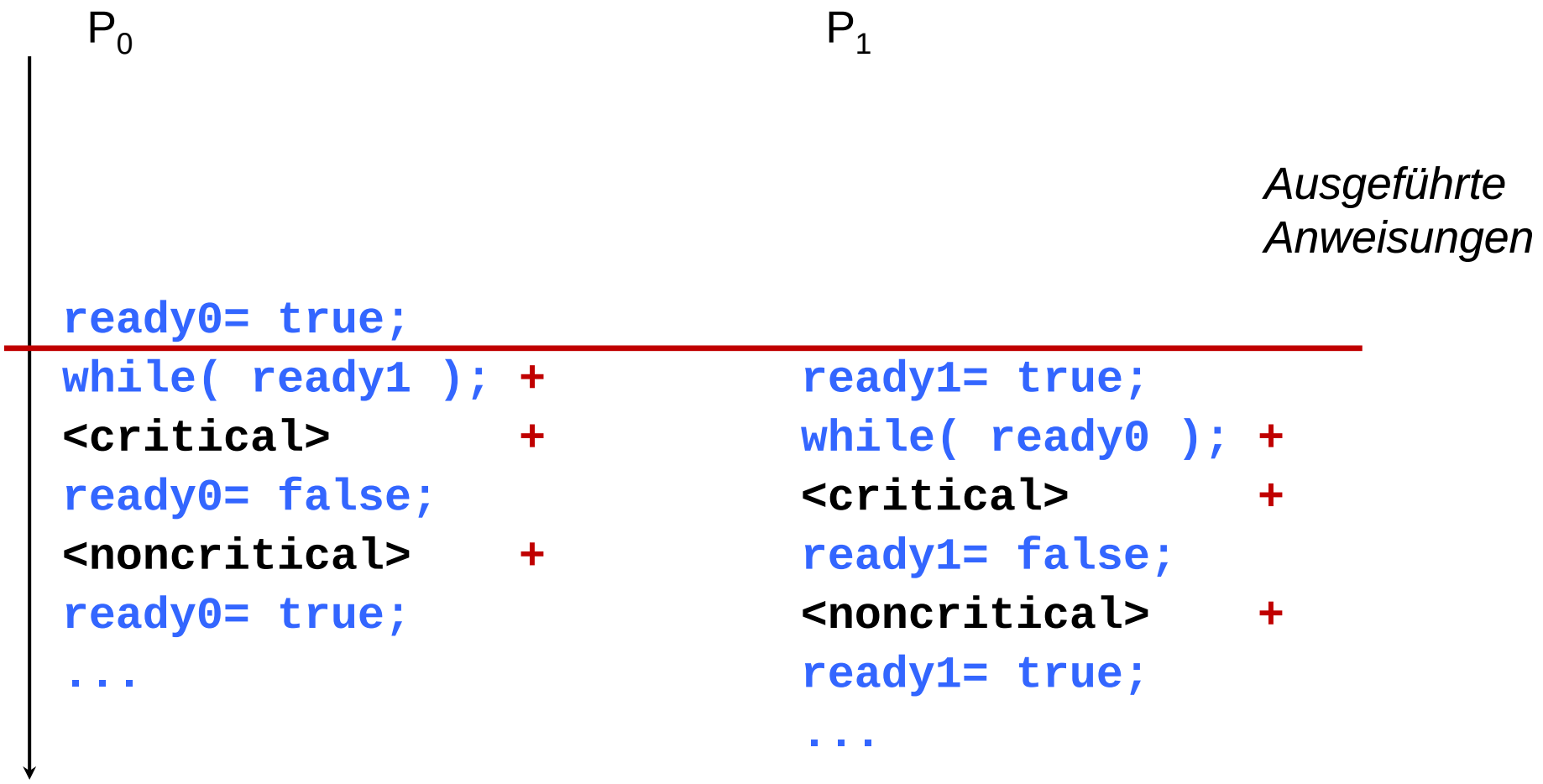
Wechselseitiger Ausschluss (8)

Deadlock



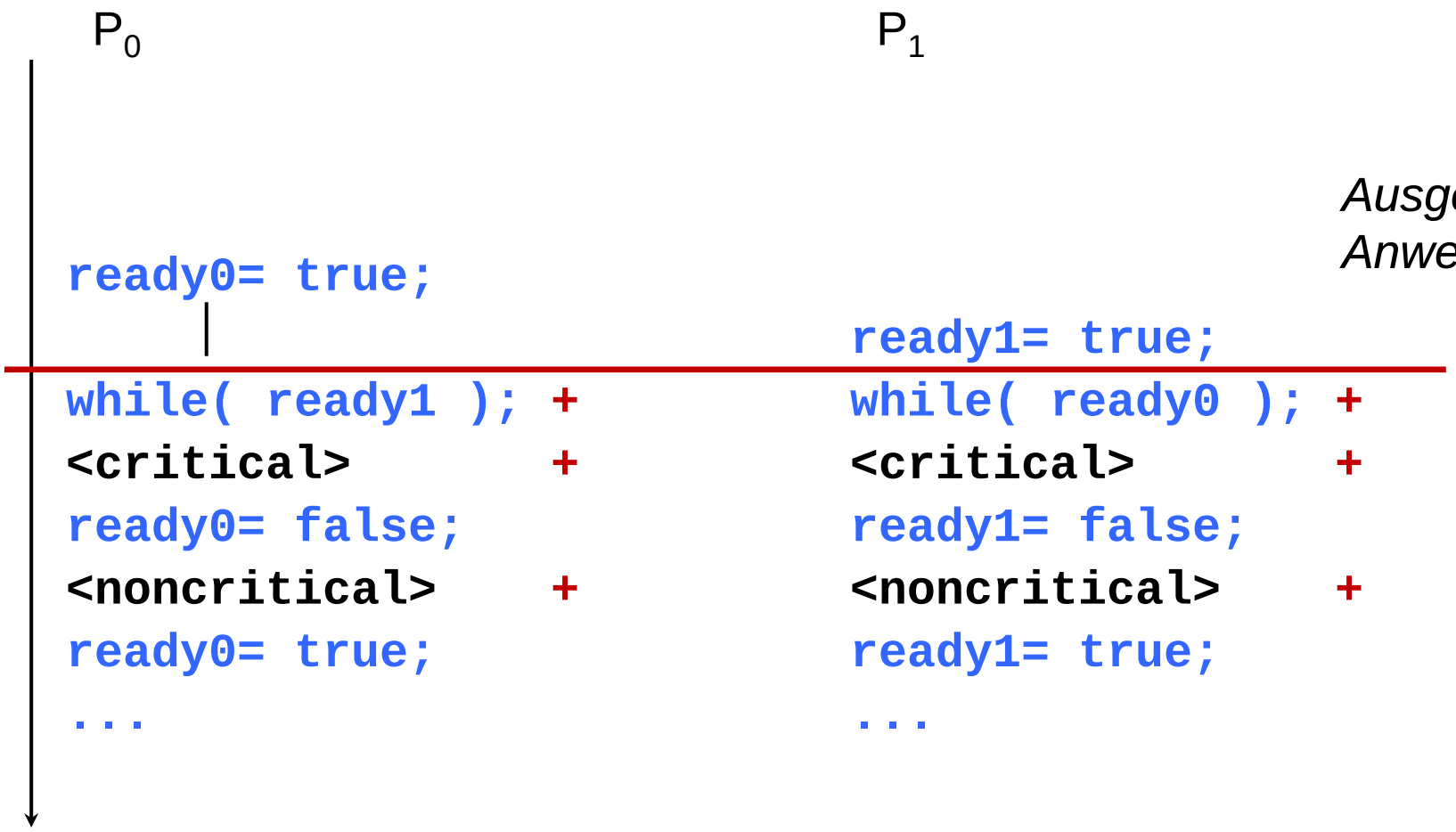
Wechselseitiger Ausschluss (8)

Deadlock



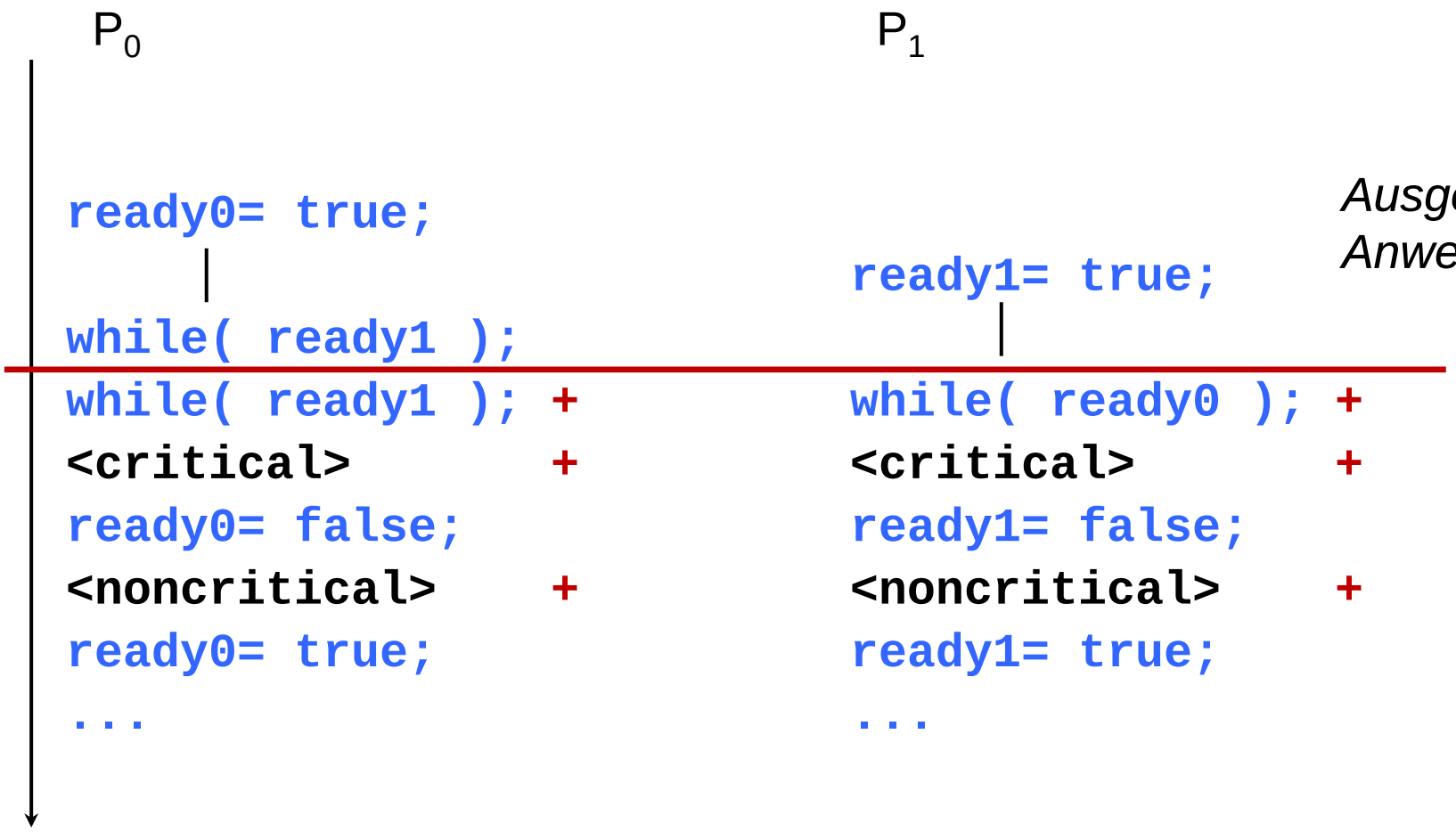
Wechselseitiger Ausschluss (8)

Deadlock



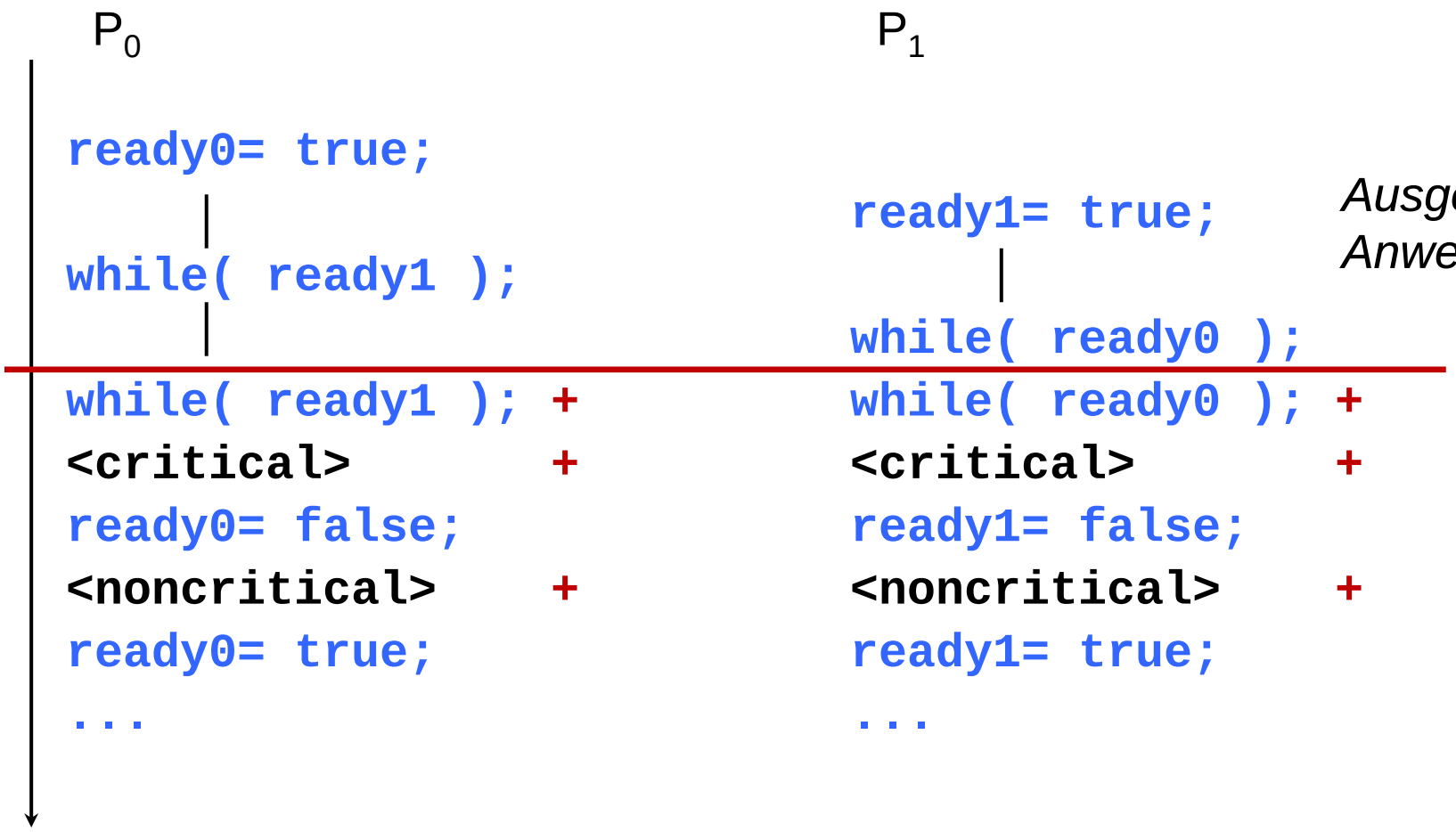
Wechselseitiger Ausschluss (8)

Deadlock



Wechselseitiger Ausschluss (8)

Deadlock



Wechselseitiger Ausschluss (9)

3. Versuch (Algorithmus von Peterson, 1981)

```
boolean ready0= false;
boolean ready1= false;
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    ready0= true;
    turn= 1;
    while( ready1 &&
           turn == 1 );

    ... /* critical sec. */

    ready0= false;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= true;
    turn= 0;
    while( ready0 &&
           turn == 0 );

    ... /* critical sec. */

    ready1= false;

    ... /* uncritical */
}
```

Wechselseitiger Ausschluss (10)

Algorithmus implementiert wechselseitigen Ausschluss

- Vollständige und sichere Implementierung
- **`turn`** entscheidet für den kritischen Fall von Versuch 2, welcher Prozess nun wirklich den kritischen Abschnitt betreten darf
- In allen anderen Fällen ist **`turn`** unbedeutend

Problem der Lösung

- Aktives Warten

Algorithmus auch für mehrere Prozesse erweiterbar

- Lösung ist relativ aufwendig

Spezielle Maschinenbefehle (1)

Spezielle Maschinenbefehle können Programmierung kritischer Abschnitte unterstützen und vereinfachen

- *Test-and-Set* Instruktion
- *Swap* Instruktion

Test-and-Set

- Maschinenbefehl mit folgender Wirkung (Java-ähnliche Syntax)

```
boolean test_and_set( MemoryBit lock )
{
  boolean tmp = lock.getBit();
  lock.setBit( true );
  return tmp;
}
```

  - Bit wird gesetzt; vorheriger Wert wird zurückgegeben
- Ausführung ist atomar

Spezielle Maschinenbefehle (2)

Kritische Abschnitte mit *Test-and-Set* Befehlen

```
MemoryBit lock= new MemoryBit(false);
```

Prozess 0

```
while( 1 ) {
  while(
    test_and_set(lock) );

  ... /* critical sec. */

  lock.setBit(false);

  ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
  while(
    test_and_set(lock) );

  ... /* critical sec. */

  lock.setBit(false);

  ... /* uncritical */
}
```

– Code ist identisch und für mehr als zwei Prozesse geeignet

Spezielle Maschinenbefehle (3)

Swap

– Maschinenbefehl mit folgender Wirkung

```
void swap( MemoryWord mem1, MemoryWord mem2 )
{
    int tmp= mem1.getWord();
    mem1.setWord( mem2.getWord() );
    mem2.setWord( tmp );
}
```

  – Inhalt zweier Speicherworte wird vertauscht

– Ausführung ist atomar

Spezielle Maschinenbefehle (4)

Kritische Abschnitte mit *Swap* Befehlen

```
MemoryWord lock= new MemoryWord( 0 );
```

Prozess 0

```
MemoryWord key=
    new MemoryWord( 0 );
while( 1 ) {
  key.setWord(1);
  while( key.getWord()==1 )
    swap( lock, key );

  ... /* critical sec. */

  lock.setWord(0);
  ... /* uncritical */
}
```

Prozess 1

```
MemoryWord key=
    new MemoryWord( 0 );
while( 1 ) {
  key.setWord(1);
  while( key.getWord()==1 )
    swap( lock, key );

  ... /* critical sec. */

  lock.setWord(0);
  ... /* uncritical */
}
```

– Code ist identisch und für mehr als zwei Prozesse geeignet

Kritik an bisherigen Verfahren

Spinlock

– Bisherige Verfahren werden auch *Spinlocks* genannt
– Aktives Warten

Problem des aktiven Wartens

– Verbrauch von Rechenzeit ohne Nutzen
– Behinderung „nützlicher“ Prozesse
– Abhängigkeit von der *Scheduling*-Strategie
  – Bspw. schlechte Effizienz bei langen Zeitscheiben

***Spinlocks* heute fast ausschließlich in Multiprozessoren eingesetzt**

– Bei kurzen kritischen Abschnitten effizient
– Koordinierung zwischen Prozessen von mehreren Prozessoren

Semaphore (1)

Semaphor (griech. „Zeichenträger“)

- System-Datenstruktur mit zwei Operationen (nach Edsger W. Dijkstra)
  - P-Operation *(proberen; passeren; wait; down)*
    - Wartet bis Zugang frei
  - V-Operation *(verhogen; vrijgeven; signal; up)*
    - Macht Zugang für anderen Prozess frei
- Datenstruktur für Zugangssteuerung: *Integer*-Wert

```
class Semaphore
{
    int s;
    Semaphore( int init ) { s= init; }

    void P() { ... }
    void V() { ... }
}
```

Semaphore (2)

Schematische Funktionsweise der Operationen

– P-Operation

```
void P()
{
    while( s <= 0 );
    s--;
}
```

atomare Funktion

– V-Operation

```
void V()
{
    s++;
}
```

atomare Funktion

Semaphore (3)

Implementierung kritischer Abschnitte mit einem Semaphor

```
Semaphore lock= new Semaphore( 1 );
```

Prozess 0

```
...
while( 1 ) {
  lock.P();

  ... /* critical sec. */

  lock.V();

  ... /* uncritical */
}
```

Prozess 1

```
...
while( 1 ) {
  lock.P();

  ... /* critical sec. */

  lock.V();

  ... /* uncritical */
}
```

Problem

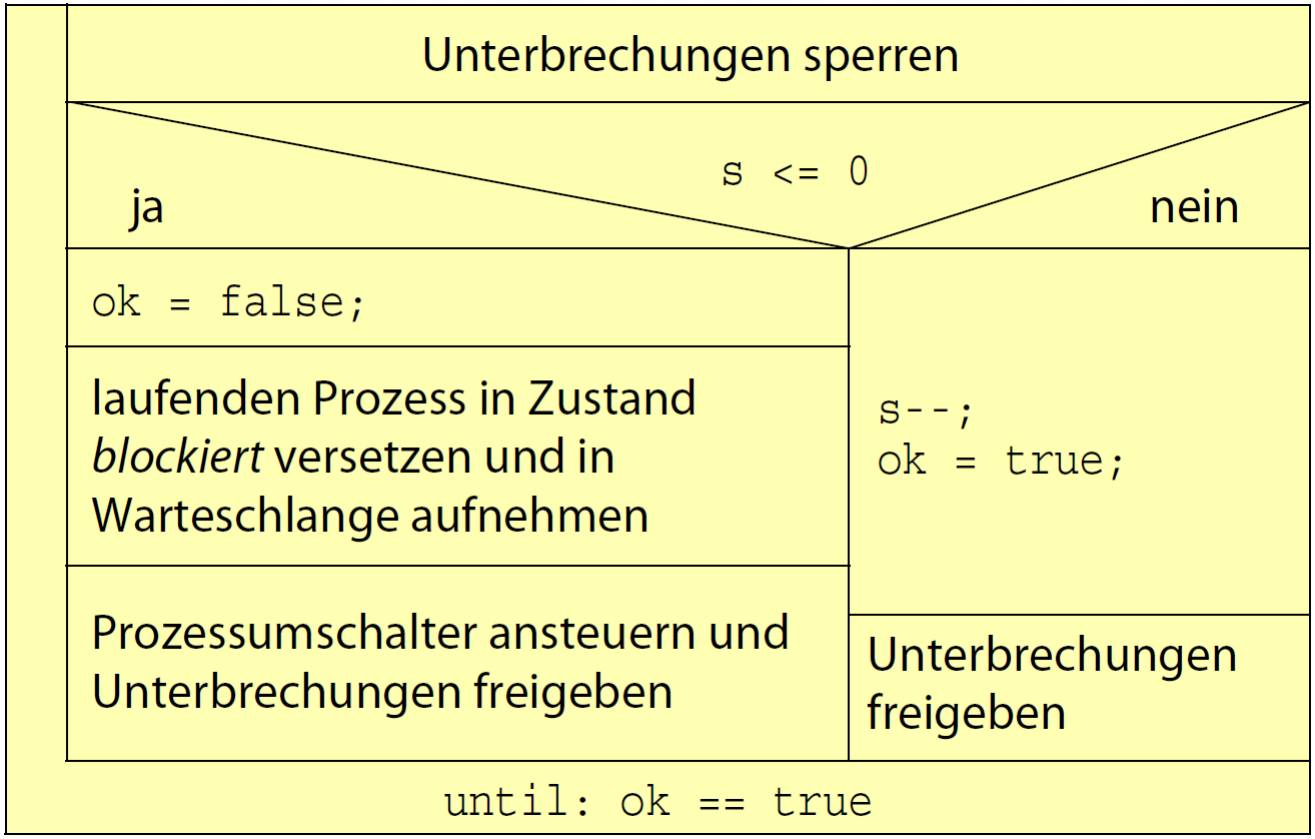
– Konkrete Implementierung von P und V

Semaphore (4)

Implementierung als Systemaufrufe im Betriebssystem (nur Monoproz.)

P-Operation:

ok ist eine prozesslokale Variable



– Jeder Semaphor besitzt *Queue*, die blockierte Prozesse aufnimmt

Semaphore (5)

V-Operation:

Unterbrechungen sperren
`S++;`
alle Prozess aus der Warteschlange in den Zustand *bereit* versetzen
Unterbrechungen freigeben
Prozessumschalter ansteuern

- Prozesse probieren immer wieder, die P-Operation erfolgreich abzuschließen
- *Scheduling*-Strategie entscheidet über Reihenfolge und Fairness
  - Leichte Ineffizienz durch Aufwecken aller Prozesse
  - Mit Einbezug der *Scheduling*-Strategie effizientere Implementierungen möglich

Semaphore (6)

Vorteile einer Semaphor-Implementierung im Betriebssystem

– Einbeziehen des *Schedulers* in die Semaphor-Operationen

– Kein aktives Warten: Ausnutzen der Blockierzeit durch andere Prozesse

Implementierung einer Synchronisierung

– Zwei Prozesse P1 und P2

– Anweisung S1 in P1 soll vor Anweisung S2 in P2 stattfinden

```
Semaphore lock= new Semaphore( 0 );
```

Prozess 1

```
...
S1;
lock.V();
...
```

Prozess 2

```
...
lock.P();
S2;
...
```

Zusammenfassung (1)

Einordnung & Motivation

– Mehrprogramm- und Mehrbenutzerbetrieb: erlauben bessere Ausnutzung der Ressource „CPU“, falls Programme Wartezeiten haben

Prozesse

– Ein Prozess ist ein Programm, das sich in Ausführung befindet, UND dessen aktuellen Daten
– Der Prozesskontrollblock als zentrale Datenstruktur des Betriebssystems verwaltet Prozess-IDs, Speicherbereiche, Eigentümer, etc.
– Prozesse können auf einer CPU umgeschaltet werden, i.d.R. unter Kontrolle des Betriebssystems über einen *Scheduler*
– Prozesse sind in einem von 5 Zuständen *(new, ready, running, blocked/waiting, terminated)*

Zusammenfassung (2)

Auswahlstrategien

- Beim Umschalten von Prozessen muss der *Scheduler* entscheiden, welcher Prozess als nächstes die CPU erhält (*Scheduling*-Strategien)
- FCFS: „fair“; nicht-minimale Wartezeiten; nicht für *Time-Sharing*
- SJF: optimiert mittlere Wartezeit; präemptiv und nicht-präemptiv
- HPF: Prioritätsumkehr; Aushungerung; Prioritätsvererbung
- *Round Robin:* präemptiv; Wahl der Zeitscheibenlänge kritisch
- *Multilevel-Queue Scheduling: Scheduling*-klassen mit eigenen *Queues*; *Scheduling* zwischen und innerhalb der Klassen

Zusammenfassung (3)

Aktivitätsträger *(Threads)*

- *Threads:* teilen sich Ressourcen; *Thread*-Wechsel sehr effizient
- *User-level Threads:* Anwendungsprogramm schaltet *Threads* um; keine Systemaufrufe erforderlich; blockierende Systemaufrufe blocken alle *Threads* eines Anwendungsprogramms
- *Kernel-level Threads:* Betriebssystem schaltet *Threads* um; kein Blockieren unbeteiligter *Threads*; weniger effizientes Umschalten

Parallelität und Nebenläufigkeit

- Parallelität: Anweisungen zweier Prozesse werden unabhängig voneinander zeitgleich ausgeführt; keine Parallelität auf Monoprozessor
- Nebenläufigkeit: Anweisungen können unabhängig ausgeführt werden, entweder zeitlich durchmischt oder echt zeitgleich
- Parallelität ist Spezialfall von Nebenläufigkeit

Zusammenfassung (4)

Koordinierung

- Problem: bei gemeinsamer Nutzung von Daten/Ressourcen durch mehrere Prozesse können Inkonsistenzen auftreten
- Ursache: Befehle einer Programmiersprache werden nicht atomar abgearbeitet; Prozesswechsel können innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen auftreten
- Kritischer Abschnitt: Bereich in denen Prozesse gemeinsame Ressourcen nutzen, wo aber nur ein Prozess zu einer Zeit Zugang haben soll *(mutual exclusion)*
- Algorithmus von Peterson: *mutual exclusion* per Kontroll-Variablen; aktives Warten; nur aufwändig auf mehrere/viele Prozesse erweiterbar
- Atomare Maschinenbefehle: elegant; aktives Warten
- Semaphore: Betriebssystem-Mechanismus („Zähler“) zur Steuerung von Prozessen; wartende Prozesse werden blockiert; kein aktives Warten