Kapitel IX

Synchronisation von Prozessen und Threads

Motivation

- Bisher: Prozesse und Threads erlauben (quasi-) parallele Verarbeitung von Applikationen bzw. innerhalb von Applikationen
- Obwohl Applikationen an sich unabhängig voneinander, ist bei der parallelen und gemeinsamen Nutzung von Betriebsmitteln ein Abstimmungsmechanismus erforderlich
- Ohne Abstimmungsmechanismus keine Erfüllung wichtiger Anforderungen:
  - Konsistente Datenhaltung
  - Korrektheit der Benutzerprogramme
  - Kontrollierter multipler Zugriff auf Betriebsmittel
- Erforderliche Abstimmungsmechanismen und Synchronisationsverfahren werden mit Hilfe der Synchronisation von Prozessen und Threads umgesetzt und im Folgenden näher betrachtet

Grundlagen (I)

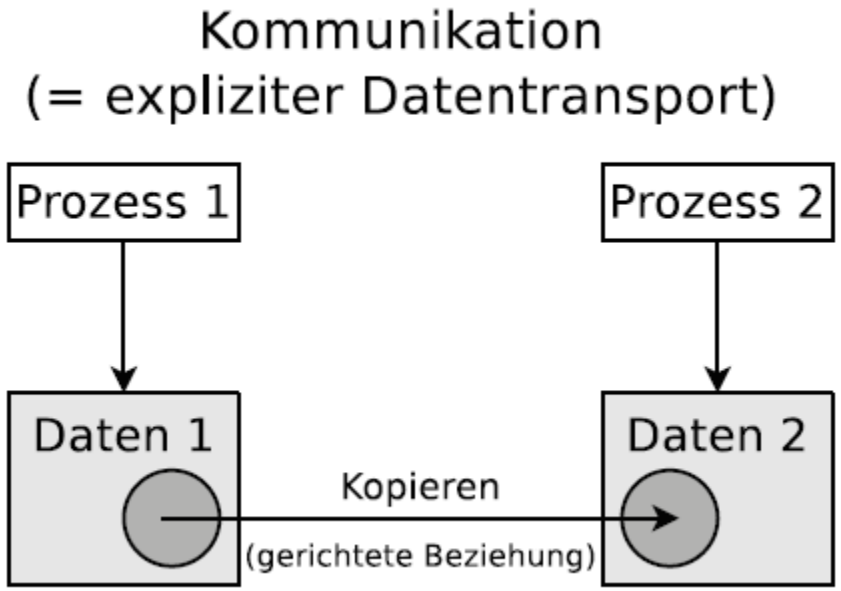
- Einleitend: Betrachtung des Synchronisationsbedarfes im Einprogrammbetrieb und Mehrprogrammbetrieb
- Dazu Wiederholung aus Definition 5.1 und Definition 5.2:
  - Einprogrammbetrieb kennt keine quasi-parallele oder echt-parallele Verarbeitung
  - Mehrprogrammbetrieb hingegen ermöglicht (quasi-) parallele Verarbeitung;
    Im Falle des Multiprocessing sogar Ermöglichung echter Parallelität
  - Achtung: Einprogrammbetrieb ungleich Einprozessorbetrieb:
    Beim Einprogrammbetrieb können keine Synchronisationsprobleme auftreten,
    beim Einprozessorbetrieb hingegen schon
- Daraus folgt:
  Synchronisation von Prozessen und Threads ist immer dann notwendig,
  wenn quasi-parallele oder echt-parallele Verarbeitung erfolgt

Grundlagen (II)

- Quasi-parallele oder echt-parallele Verarbeitung bildet Grundlage dafür, dass Prozesse miteinander interagieren (müssen) → Prozessinteraktion
- Bei der Prozessinteraktion erfolgt Unterscheidung zwischen
  - **Funktionalem Aspekt** (Wie erfolgt die Prozessinteraktion?)
  - **Zeitlichem Aspekt** (Wie läuft die Prozessinteraktion in zeitlicher Hinsicht ab?)
- Antwort auf den funktionalen Aspekt liefert dabei die Interprozesskommunikation
- Der zeitliche Aspekt wird mit Hilfe der Synchronisation beantwortet, auf die in diesem Kapitel näher eingegangen wird
- Zuvor jedoch: Herstellen des Zusammenhangs zwischen funktionalem und zeitlichem Aspekt bei der Prozessinteraktion

Grundlagen (III)

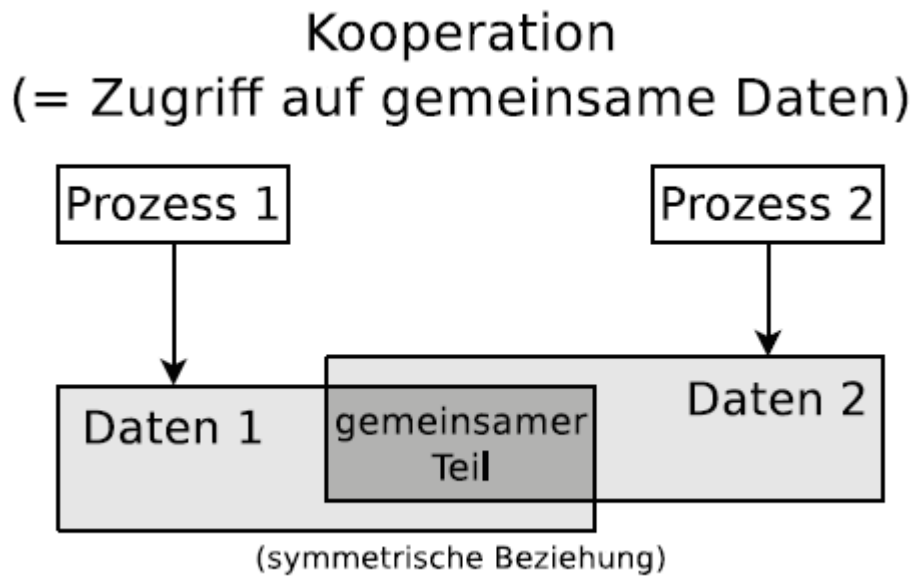
- Funktionaler Aspekt: Wie erfolgt die Prozessinteraktion?
  - Dabei Unterscheidung zweier Interaktions-Arten: **Kommunikation und Kooperation**
  - **Kommunikation** ist die Interaktion bezogen auf einen expliziten Datenaustausch zwischen Prozessen



Quelle: *[BS15]*

Grundlagen (IV)

- **Kooperation** ist der Zugriff auf gemeinsame Daten der betrachteten Prozesse



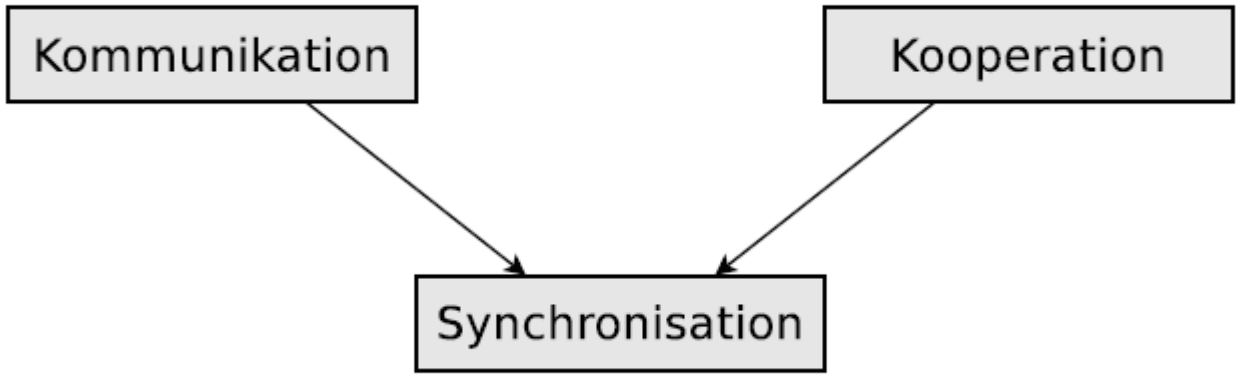
Quelle: *[BS15]*

Grundlagen (V)

- Kommunikation zwischen Prozessen koordiniert die Sende- und Empfangsbereitschaft der Prozesse
- Kooperation zwischen Prozessen koordiniert den geregelten, kontrollierten Zugriff auf die gemeinsam genutzten Ressourcen
- Eine zeitliche Steuerung der Ablaufreihenfolge ist sowohl bei der Kommunikation als auch der Kooperation vonnöten, sodass gilt:
  - Koordination der Kommunikation baut auf Synchronisation auf
  - Koordination der Kooperation baut auf Synchronisation auf

Grundlagen (VI)

- Grafische Darstellung: Synchronisation bildet Grundlage für die Interaktion zwischen Prozessen



Quelle: *[BS15]*

Grundlagen (VII)

- Bei der Prozessinteraktion kann der Zugriff auf unterschiedlichste Betriebsmittel erfolgen
- Im Folgenden:

  Einteilung der Betriebsmittel in drei Klassen und anschließend Betrachtung möglicher Problemsituationen, die Synchronisation erforderlich machen

Gemeinsame Betriebsmittel und Problemsituationen (I)

- Betrachtung gemeinsamer Betriebsmittel zwischen Prozessen und Threads
- Dabei Unterteilung der Betriebsmittel in drei Klassen:
  - **Gemeinsam benutzte Datenstrukturen (Shared Data Structures):**

    Z. B.: Programmvariablen, Applikationsbezogene Datenstrukturen, ...

  - **Gemeinsam benutzte Dateien (Shared Files):**

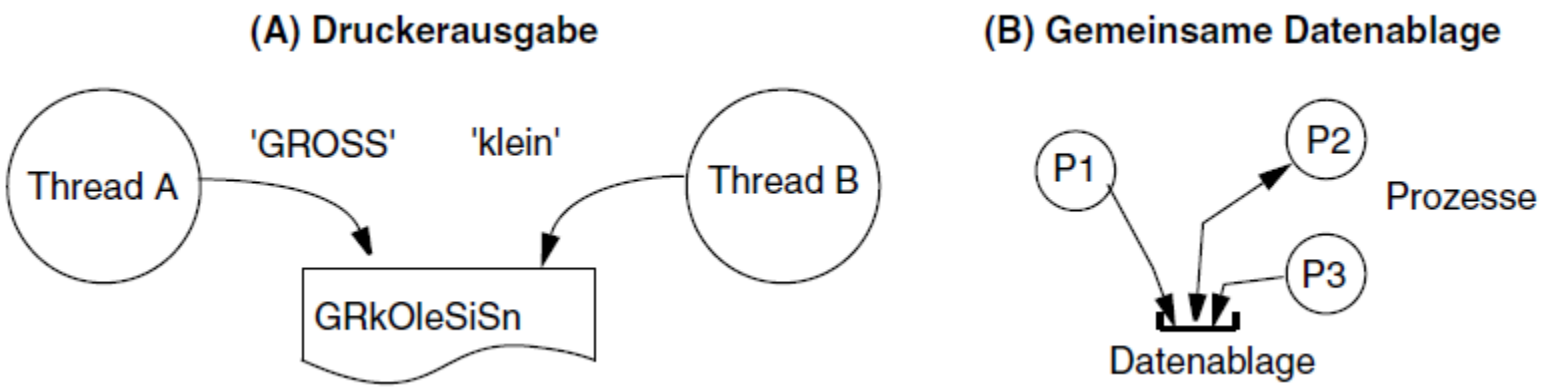
    Z. B.: Applikationsübergreifende Nutzung von: XML-Daten, Tabellenkalkulation, Log-Dateien, ...

  - **Gemeinsam benutzte Hardware (Shared Hardware):**

    Z. B.: Netzwerkkarte, Drucker, ...

Gemeinsame Betriebsmittel und Problemsituationen (II)

- Beim gemeinsamen Zugriff auf die Betriebsmittel ergeben sich Probleme, die Synchronisation der genannten Betriebsmittel erforderlich machen:



Quelle: *[BS15]*

Gemeinsame Betriebsmittel und Problemsituationen (III)

- Allgemein lassen sich Probleme beim Ressourcenzugriff mit Hilfe der folgenden Szenarien beschreiben:

  - Verlorene Aktualisierung (Lost Update Problem)
  - Inkonsistente Abfrage (Inconsistent Read)

Gemeinsame Betriebsmittel und Problemsituationen (IV)

- **Szenario 1: Verlorene Aktualisierung (Lost Update Problem):**

  - Szenario: Zwei parallel ablaufende Threads (Thread A, Thread B) führen Buchungen auf einem gemeinsamen Konto durch

  - Problem: Durch die parallelen Zugriffe geht ein Konto-Update eines Threads verloren; ab diesem Zeitpunkt wird mit fehlerbehafteten Daten weitergearbeitet

Gemeinsame Betriebsmittel und Problemsituationen (V)

- Verlorene Aktualisierung (Lost Update Problem): Ablauf

Thread A	shared data	Thread B
	saldo	
:	25.50	:
saldo=saldo+5.10;		saldo=saldo-4.50;
	?	

- *Shared Data* ist gemeinsame Ressource (Kontostand), die zu Beginn von beiden Prozessen geteilt wird
- Nachdem der gemeinsame Kontostand abgerufen wurde, führen die Threads jeweils intern und isoliert vom anderen Thread Operationen darauf aus, bevor sie den Kontostand zurückschreiben
- Fragestellung: Was ist das Endresultat des Kontostandes?

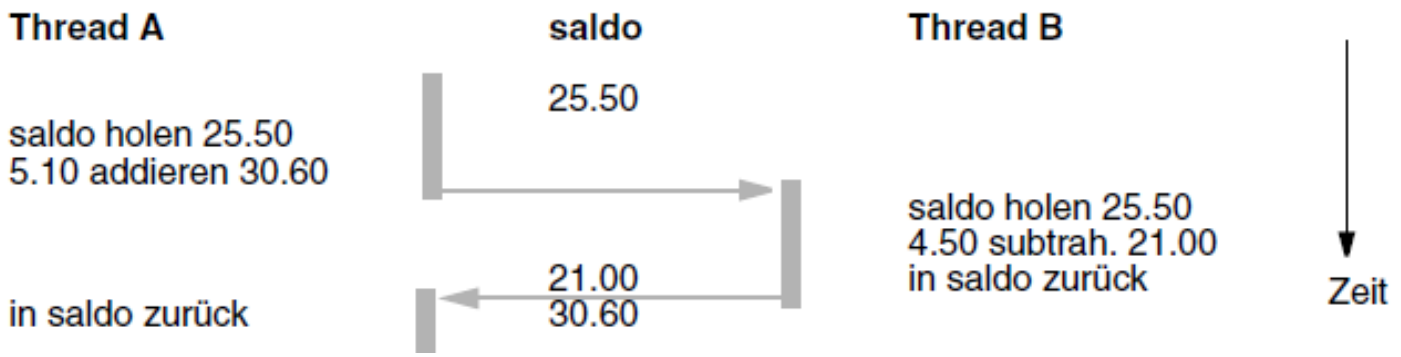
Quelle: *[BS15]*

Gemeinsame Betriebsmittel und Problemsituationen (VI)

- Beantwortung der Frage bedingt Annahmen darüber:
  - Wann ein Rescheduling stattfindet
  - Welcher Thread wann den Kontostand einliest
  - Welcher Thread wann welchen Wert in den gemeinsamen Kontostand zurückschreibt
- Auf dieser Basis ergeben sich mehrere mögliche Ablaufvarianten, die unterschiedliche Endresultate liefern.

  Im Folgenden: Betrachtung zweier Ablaufvarianten:

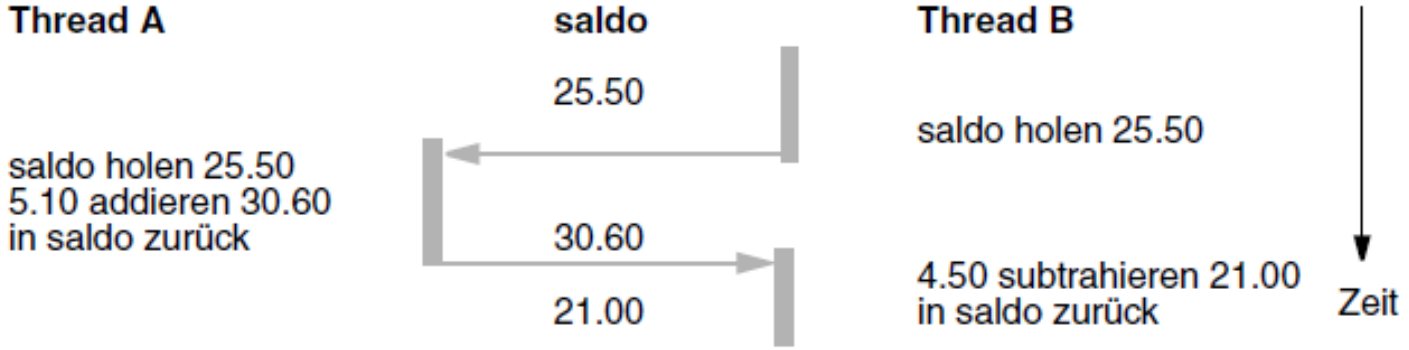
- Ablaufvariante 1:



Quelle: *[BS15]*

Gemeinsame Betriebsmittel und Problemsituationen (VII)

- Ablaufvariante 2:



- In beiden Ablaufvarianten steht am Ende ein fehlerhaftes Endresultat, welches durch den unsynchronisierten parallelen oder quasi-parallelen Ablauf zustande gekommen ist
- Zwei Begriffe, die im Zusammenhang mit derartigen Problemsituationen genannt werden, sind die sogenannte **Race Condition** und die **Critical Section**, welche im Folgenden näher betrachtet werden

Quelle: *[BS15]*

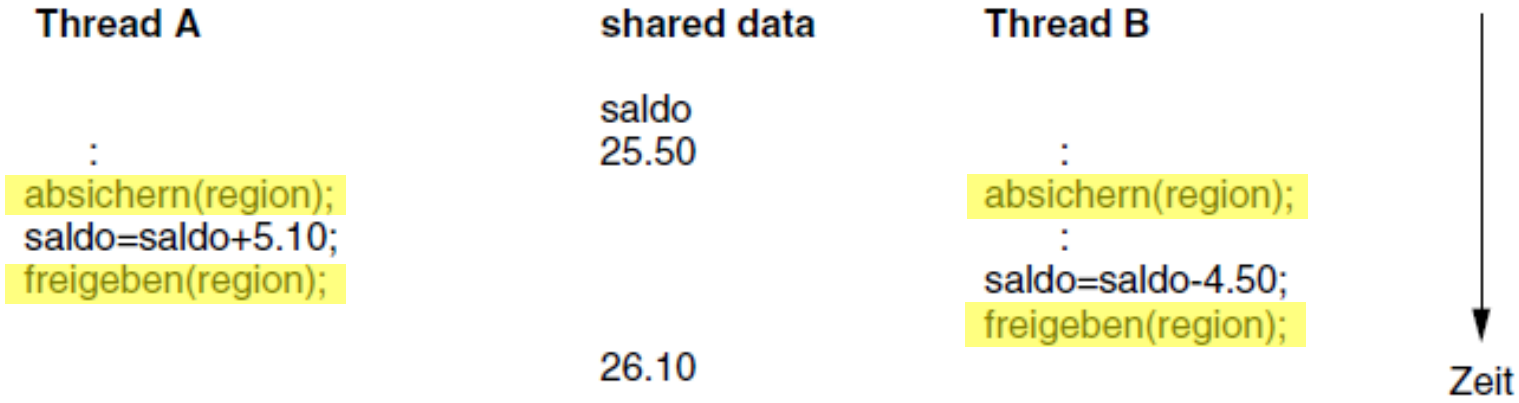
Gemeinsame Betriebsmittel und Problemsituationen (VIII)

- Der Begriff der Race Condition:
  - Eine Race Condition tritt dann auf, wenn das Resultat parallel ablaufender Operationen davon abhängig ist, welche Operation wann läuft
  - In diesem Zusammenhang kann man sagen: „Ein Thread überholt den anderen“
  - Ein Codeabschnitt, indem eine Race Condition auftreten kann, bezeichnet man als „Critical Section“

- Der Begriff der Critical Section bzw. des kritischen Abschnitts:
  - Bezeichnet einen Code-Abschnitt, in dem Operationen durchgeführt werden, die bei der parallelen Verarbeitung unterschiedliche Endresultate liefern können
  - Kritische Abschnitte kennzeichnen sich dadurch, dass ein Wert von mehreren Threads parallel gelesen, dann bearbeitet und verzögert in die gemeinsame Ressource zurückgeschrieben werden kann

Gemeinsame Betriebsmittel und Problemsituationen (IX)

- Absicherung von kritischen Bereichen beim Problem der verlorenen Aktualisierung:



- Mit Hilfe der Befehle `absichern()` und `freigeben()` wird ein wechselseitiger Ausschluss der Zugriffsoperationen ermöglicht → **Mutual Exclusion, Mutex**
- Die Bearbeitung des Kontostands wird dadurch unteilbar bzw. atomar ausgeführt
  → Fehlersituation ist ausgeschlossen

Quelle: *[BS15]*

Gemeinsame Betriebsmittel und Problemsituationen (X)

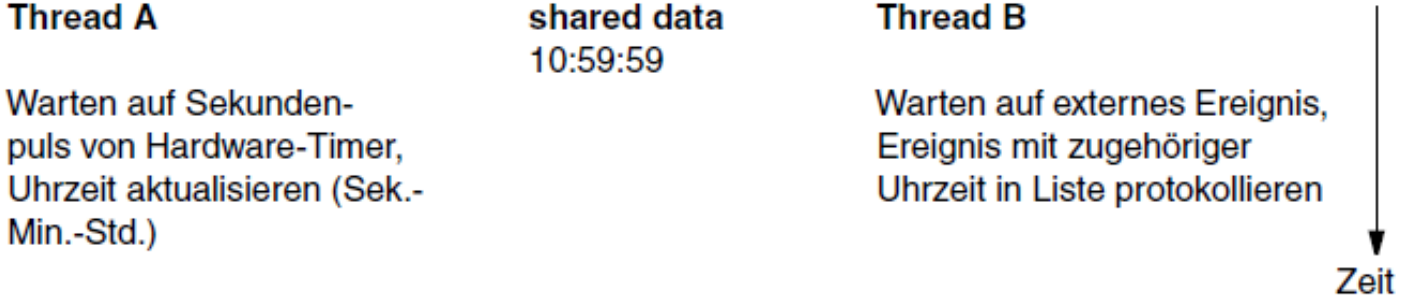
- **Szenario 2: Inkonsistente Abfrage (Inconsistent Read):**

  - Szenario: Ein Thread (Thread A) hält eine Datenressource aktuell, welche von einem anderen Thread (Thread B) kontinuierlich oder auf bestimmte Ereignisse hin ausgelesen wird

  - Problem: Wird die Datenressource nicht in einer unteilbaren Operation aktualisiert, können durch Thread B fehlerhafte, nur in Teilen aktualisierte Daten ausgelesen werden

Gemeinsame Betriebsmittel und Problemsituationen (XI)

- Szenario 2: Inkonsistente Abfrage (Inconsistent Read): Ablauf



- Thread A reagiert auf Ereignisse des Hardware-Timers zur sekündlichen Aktualisierung der Uhrzeit
- Thread B protokolliert diverse externe Ereignisse sowie den Zeitpunkt des Ereignisses
- Somit: Thread A stellt gemeinsam geteilte Daten bereit, welche von Thread B konsumiert werden
- Fragestellung: Wird die Uhrzeit von Thread B in jedem Fall richtig ausgelesen?

Quelle: *[BS15]*

Gemeinsame Betriebsmittel und Problemsituationen (XII)

- Dazu Betrachtung der gemeinsam geteilten Datenstruktur, welche z. B. folgenden Aufbau haben kann:

```
struct {
   unsigned int Stunden;
   unsigned int Minuten;
   unsigned int Sekunden;
} uhrzeit;
```

Gemeinsame Betriebsmittel und Problemsituationen (XIII)

- Der Programmcode zur Aktualisierung der Uhrzeit könnte folgendermaßen aussehen:

```
…
uhrzeit.Stunden = 10;
uhrzeit.Minuten = 59;
uhrzeit.Sekunden = 59;
…
```

Gemeinsame Betriebsmittel und Problemsituationen (XIV)

- Daraus ergibt sich folgende mögliche Problemsituation:
  Uhrzeit soll von 10:59:59 auf 11:00:00 aktualisiert werden

Thread A	Shared Data	Thread B
	`uhrzeit = 10:59:59`	
`uhrzeit.Stunden = 11;`		
…		`lese(uhrzeit); // ergibt: 11:59:59`
`uhrzeit.Minuten = 00;`		
`uhrzeit.Sekunden = 00;`		

- Aufgrund der nicht-atomaren Aktualisierung der Uhrzeit liest Thread B die falsche Uhrzeit aus
  → Auch hier: Absicherung der kritischen Region mit Hilfe eines Mutex

Implementierung der Synchronisation (I)

- Die beiden Problemsituationen *Lost Update* und *Inconsistent Read* illustrieren zwei Arten von Problemen, die in einer Multitasking-Umgebung auftreten können
- Verschiedene Implementierungen erlauben es, diese Problemsituationen zu vermeiden; dies geschieht entweder mittels Selbstverwaltung oder mit Hilfe von Systemmitteln
- Zunächst: Betrachtung mit Hilfe der Selbstverwaltung
- Anschließend: Betrachtung mit Hilfe von Systemmitteln: Semaphore, Signale
  → Problematik dabei: Deadlocks

Implementierung der Synchronisation (II)

- **Absicherung mit Selbstverwaltung: Ansatz mit Busy Loops und boolescher Variable**
  - Einfachster, intuitiver Ansatz zur Selbstverwaltung kann mit Hilfe des aktiven Wartens bzw. aktiver Warteschleifen (sog. Busy Loops) und einer booleschen Variablen realisiert werden
  - Aktives Warten kann z. B. mit Hilfe einer while-Schleife realisiert werden
  - Globale, boolesche Variable sichert kritischen Abschnitt

Implementierung der Synchronisation (III)


```
static bool ressource_ist_besetzt = false; // globale Variable sichert kritischen
                                           // Abschnitt.

…
while (ressource_ist_besetzt) {
        // nichts tun: aktives Warten.
}
ressource_ist_besetzt = true;
verwende_ressource(ressource);
ressource_ist_besetzt = false;
```

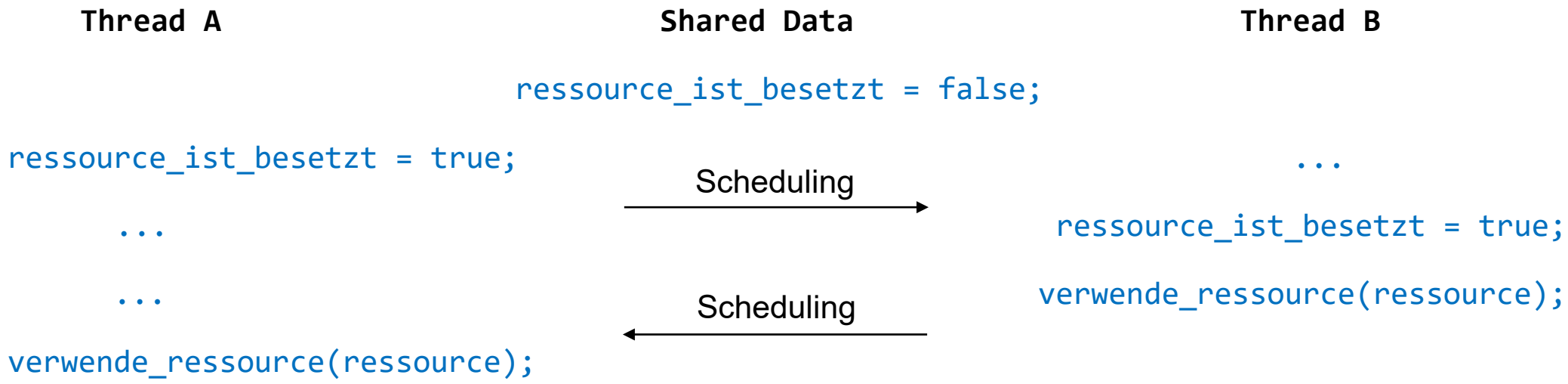
Implementierung der Synchronisation (IV)

- Aktives Warten bei diesem Ansatz stellt Nachteil dar.

  Denn: Ineffiziente Nutzung der CPU-Rechenzeit

  → Kann allerdings durch Verwendung einer Pausenfunktion gelindert werden: Thread wird für geraume Zeit „schlafen gelegt“, sodass andere Threads währenddessen CPU nutzen können

- Weiterer Nachteil besteht im nicht-atomaren prüfen und setzen der globalen Sicherungsvariablen:



Implementierung der Synchronisation (V)

- Nicht-atomares prüfen und setzen der Sicherungsvariablen kann bei parallelen Aufrufen dafür sorgen, dass wechselseitiger Ausschluss nicht stattfindet und kein exklusiver Ressourcenzugriff erfolgt
- Mit Hilfe eines sogenannten TAS-Befehls (Test And Set) kann das Prüfen und das Setzen der Sicherungsvariable als unteilbare Operation ablaufen
- Jedoch: Programm wird dadurch prozessorabhängig, da TAS-Befehl als Maschinenbefehl umgesetzt werden muss

  → Wartbarkeit und Portabilität der Anwendung stark eingeschränkt

Implementierung der Synchronisation (VI)

- **Absicherung mit Selbstverwaltung: Bakery-Algorithmus**
  - Bakery-Algorithmus stellt besseren Ansatz als den oben gezeigten dar, indem er unabhängig vom verwendeten Prozessor einheitlich umgesetzt werden kann
  - Algorithmus orientiert sich an der geregelten Bedienung von Kunden, wie sie insbesondere auf Ämtern oder in Verkaufsgeschäften realisiert ist:

    1. Kunde zieht bei der Ankunft eine Nummer
    2. Sobald alle vorher eingetroffenen Kunden bedient wurden, wird der Kunde anhand seiner Nummer aufgerufen und bedient

  - Im Rahmen dieser Vorlesung nicht näher betrachtet

Implementierung der Synchronisation (VII)

- **Absicherung mit Systemmitteln:**

  - Grundelement eines Betriebssystems für wechselseitigen Ausschluss bei begrenzter Ressourcenzahl: Semaphoren
  - Dabei potenziell auftretendes Problem: Deadlocks

Kapitel X

Interprozesskommunikation

Motivation

- Wdh.: Aus Synchronisation geht hervor, dass Prozesse Mechanismus zur gegenseitigen Abstimmung benötigen, um Synchronisation von Ressourcen durchführen zu können
- Folglich: Prozesse benötigen im Sinne der gemeinsamen Interaktion einen Mechanismus, der Kommunikation ermöglicht
- Darüber hinaus: Weitere Anwendungsfälle, die Kommunikation zwischen Prozessen erforderlich machen:
  - Pipes in Kommandozeile ermöglichen durch Benutzer gesteuerte Interprozesskommunikation (Bsp. Powershell):

    ```
    Get-Process | Measure-Object | Select-Object Count
    ```
  - Kommunikation in verteilten Systemen, z. B. Remote Procedure Call (RPC)

Grundlagen (I)

- Interprozesskommunikation auch als IPC (Interprocess Communication) bekannt
- Begriff der Interprozesskommunikation zunächst nicht eindeutig definiert:
  - Eine Interpretation:

    Abbildung der Kommunikation zwischen Prozessen bzw. Threads

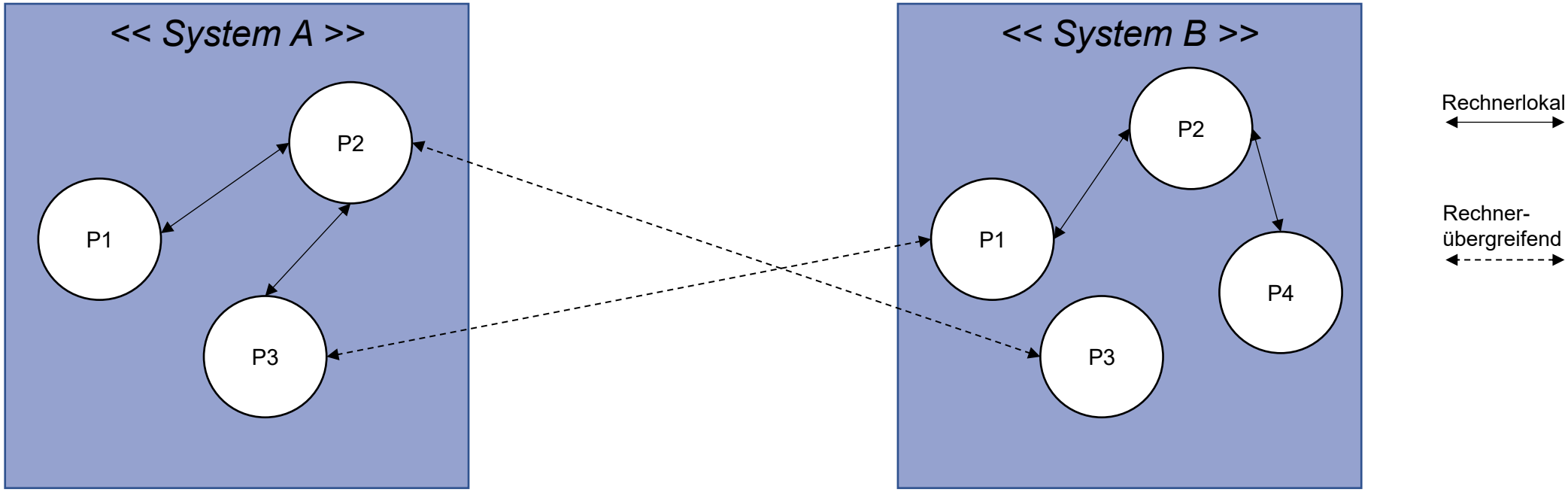
    → Fokus auf Kommunikation
  - Weitere Interpretation:

    Synchronisation von Prozessen und Threads beim gemeinsamen Ressourcenzugriff;

    → Fokus auf Koordination
- Deshalb Wdh. aus Kapitel Synchronisation: Unterscheidung Kommunikation und Synchronisation:
  - Kommunikation dient dem expliziten Datenaustausch zwischen Prozessen und Threads
  - Synchronisation ist die Koordination bezüglich des zeitlichen Ablaufs von Ressourcenzugriffen

Grundlagen (II)

- Interprozesskommunikation kann sowohl rechnerlokal, als auch rechnerübergreifend stattfinden:



Grundlagen (III)

- Im Folgenden:
  - Fokussierung auf Abbildung der Kommunikation zwischen Prozessen und Threads, Synchronisation bereits in separatem Kapitel behandelt
  - Lediglich Betrachtung der Konzepte, für weitere Vertiefung sei auf Literatur verwiesen

Grundlagen (IV)

- Interprozesskommunikation kann anhand der Verfahrensweise klassifiziert werden:

  - **Nachrichtenbasierte Verfahren:**

    Beruhen auf dem Nachrichtenaustausch über Systemfunktionen

  - **Speicherbasierte Verfahren:**

    Meinen die Kommunikation mit Hilfe spezieller Speicherbereiche, die prozessübergreifend zugänglich sind

Nachrichtenbasierte Verfahren (I)

- Nachrichtenbasierte Verfahren unterscheiden sich in der Form des Datenaustauschs sowie der Synchronität
- Drei grundsätzliche Möglichkeiten zur Abgrenzung des Datenaustauschs:
  - **Message Passing:** Datenaustausch mittels Nachrichten
  - **Streaming:** Datenaustausch mittels Datenströmen
  - **Packeting:** Datenaustausch mittels Paketen
- Unterscheidung in der Synchronität:
  - **Synchroner Nachrichtenaustausch**
  - **Asynchroner Nachrichtenaustausch**

Nachrichtenbasierte Verfahren (II)

- Message Passing:
  - Abgegrenzte Datenmenge in Form der Meldung (Message)
  - Meldungsgröße dabei fest oder variabel
  - Systemaufrufe unter Windows: `MPI_Send()` und `MPI_Receive()`

- Streaming:
  - Verwendung von Datenströmen
  - Dadurch: Meldungsgröße ist dem Sender und Empfänger nicht bekannt und in der Theorie unbeschränkt

Nachrichtenbasierte Verfahren (III)

- Packeting:
  - Verwendung fester, teilweise standardisierter Datenformate (Z. B.: IP-Paketformat)
  - Pakete sind im Rahmen der Applikationsprogrammierung transparent
  - Beim Übertragen: U. U. Fragmentierung und Defragmentierung der Pakete, was durch Netzwerksoftware realisiert wird

Nachrichtenbasierte Verfahren (IV)

- Synchrone Kommunikation:
  - Entweder Sender muss auf Empfänger warten, oder Empfänger muss auf Sender warten

- Asynchrone Kommunikation:
  - Senden kann direkt erfolgen, auch wenn Empfänger nicht empfangsbereit
  - Dies wird möglich durch sogenannten Nachrichtenpuffer, der Sender und Empfänger voneinander entkoppelt

Speicherbasierte Verfahren (I)

- Speicherbasierte Verfahren bieten den Vorteil, dass sie auf keinerlei Systemhilfe (Systemaufrufe zur IPC) angewiesen sind und den Datenaustausch über vorhandene Strukturen abwickeln
- Auf Basis dieses Ansatzes erfolgt bei Threads der Datenaustausch über globale Variablen, die für alle Threads innerhalb des Prozesses verfügbar sind
- Jedoch: Speicherbasierte Verfahren funktionieren nur so lange Threads nicht über Prozessgrenzen hinweg kommunizieren sollen
  → Stichwort: der Adressraum-Isolierung von Prozessen
- Abhilfe: Betriebssysteme bieten mit Hilfe des virtuellen Speichers die Möglichkeit, prozessübergreifende Speicherbereiche einzurichten

Speicherbasierte Verfahren (II)

- Solche Speicherbereiche bezeichnet man als *Shared Memory*
- Bei der Verwendung von Shared Memory ist zu beachten, dass Synchronisation zwischen den Prozessen nicht gewährleistet ist
- Semaphor-Implementierungen die prozessübergreifend einsetzbar sind, eignen sich, um Synchronisationsproblem zu lösen

Rechnerlokale Interprozesskommunikation (I)

Weitere Verfahren der Interprozesskommunikation (Teil der Übungsaufgaben):

- Monitor
- Rendezvous
- Berkeley Sockets
- Remote Procedure Call (RPC)

Kapitel XI

Scheduling

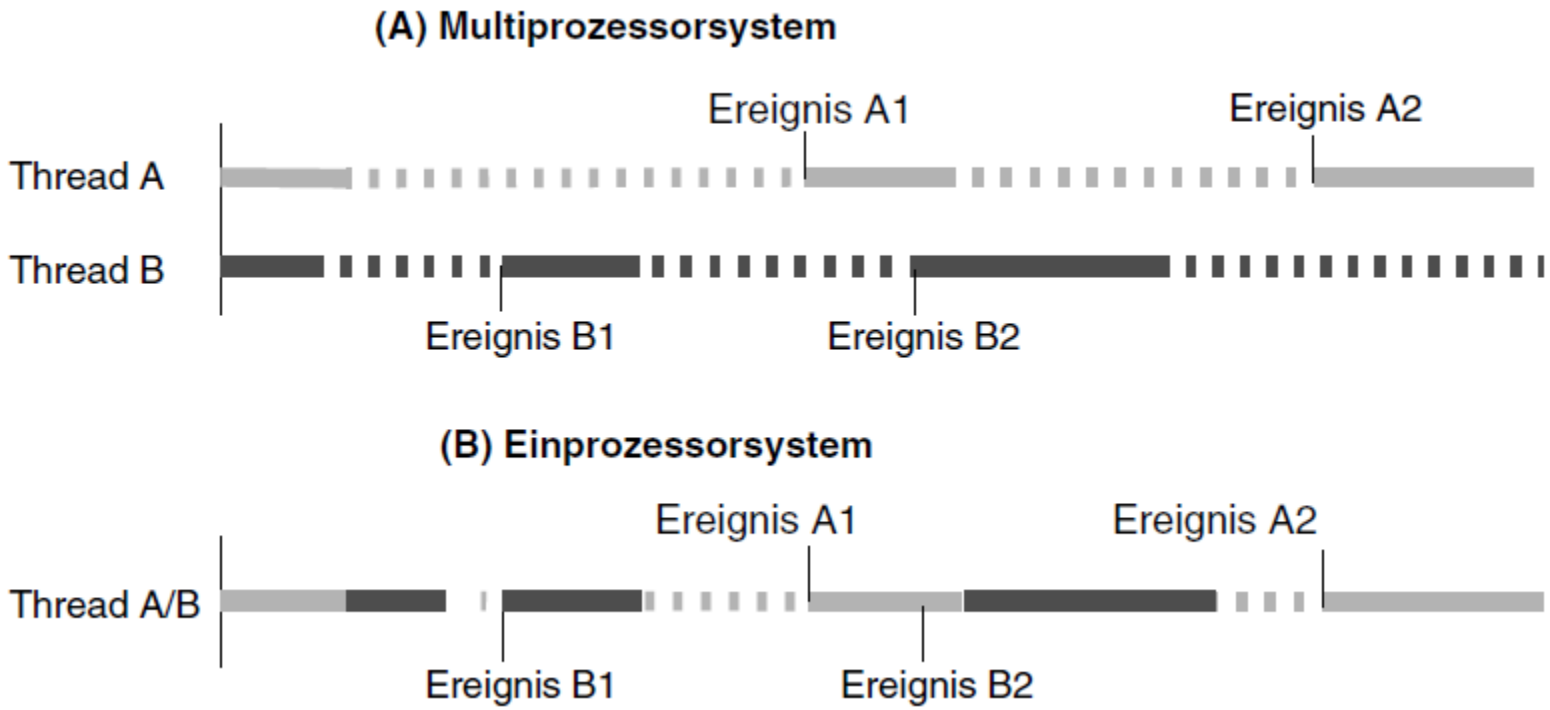
Motivation

- Die Motivation für das Prozess-Scheduling besteht in den Hardware-Gegebenheiten älterer und moderner Prozessoren
- Ältere Single-Core-Systeme erforderten Möglichkeit der quasi-parallelen Ausführung, welche mit Hilfe von Zeitmultiplexing der Rechenzeit umgesetzt wurde
- Moderne Systeme mit mehreren Prozessorkernen sind dahingehend optimiert, dass sie mit mehreren Rechenkernen und damit echter Parallelität aufwarten
- Aber: idealistische Anforderung, pro Prozess einen eigenen Prozessor / Kern zur Verfügung zu stellen liefern weder ältere noch moderne Systeme
- Diese Gegebenheit und die Annahme dass Prozesse i. A. nicht über ihre gesamte Prozesslebensdauer rechnen, führen zu dem Wunsch die Prozessor-Ressourcen zwischen den Prozessen aufzuteilen. Umgesetzt wird dies mit Hilfe des Scheduling.

Grundlagen (I)

- Als Scheduling bezeichnet man die Prozessorzuteilung zu Prozessen oder Threads
- Notwendigkeit des Schedulings besteht in der Tatsache, dass es in heutigen Systemen quasi unmöglich ist, pro Prozess einen eigenen Prozessor (-kern) bereitzustellen
- Dadurch: Anforderung an Aufteilung der Prozessor-Ressourcen auf verschiedene Prozesse bzw. Threads
- Für eine optimale Prozessorausnutzung geht man beim Scheduling davon aus, dass Prozesse niemals dauerhaft Rechenleistung benötigen, sondern von Zeit zu Zeit auf Ereignisse warten müssen
- Ein Ereignis kann dabei sein: Signale oder Meldungen im Sinne der Interprozesskommunikation, I/O-Wartezeiten, Geräteansteuerung, …

- Betrachtung des Scheduling im Einprozessorsystem vs. Multiprozessorsystem:



Quelle: *[BS15]*

Grundlagen (III)

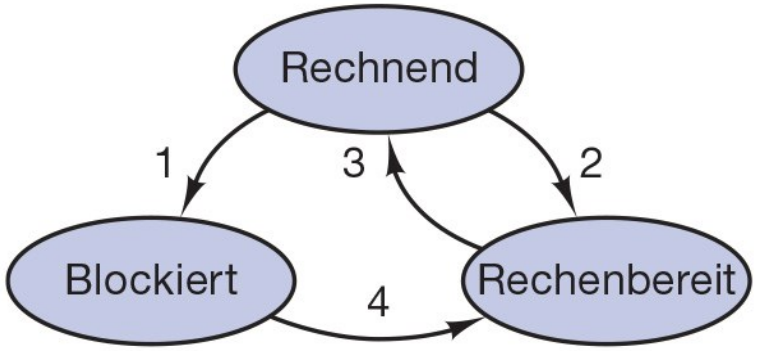
- Folgerungen, die sich aus dem Scheduling ergeben:
  1. Summe der von allen Prozessen benötigten Rechenzeit muss kleiner sein als die vom Prozessor zur Verfügung gestellte Rechenzeit
  2. Reaktionszeiten auf Ereignisse werden durch Scheduling beeinflusst, sodass verzögerte Ereignisbehandlung möglich ist
  3. Betriebssystem muss Prozesse blockieren können und es sind keine aktiven Warteschleifen erlaubt (anderenfalls wäre Scheduling und damit effiziente Nutzung der Rechenzeit nicht möglich)

- Aus 3. folgt weiter: Prozesszustände müssen verwaltet werden.

  Dazu Wiederholung von Prozess-Zustände.

Grundlagen (IV)

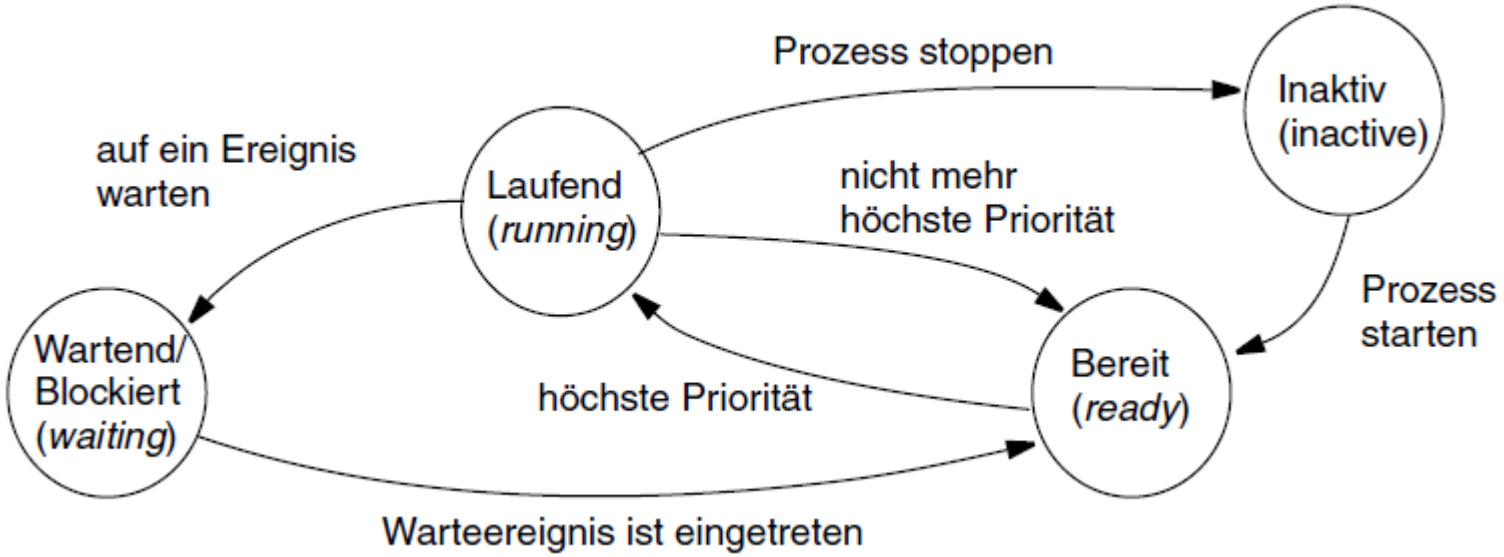
- Wiederholung Prozess-Zustände:



1. Prozess blockiert, weil er auf Eingabe wartet
2. Scheduler wählt einen anderen Prozess aus
3. Scheduler wählt diesen Prozess aus
4. Eingabe vorhanden

- Im einfachen Prozess-Modell: Lediglich Betrachtung der Zustände:
  - Rechenbereit
  - Rechnend
  - Blockiert
- Für Scheduling: Erweiterung des Prozess-Modells um den Zustand „Inaktiv (Inactive)“, welcher sich aus der Möglichkeit des Prozess-Starts und der Prozess-Beendigung ergibt

- Erweitertes Prozessmodell mit vier Zuständen:



- Ein Prozess befindet sich im Status *Inaktiv*, wenn er noch nicht gestartet oder bereits beendet wurde → Notwendig für Scheduling, da ein Prozess in diesem Zustand niemals Rechenzeit zugeteilt bekommt

Quelle: *[BS15]*

Implementierung des Scheduling (I)

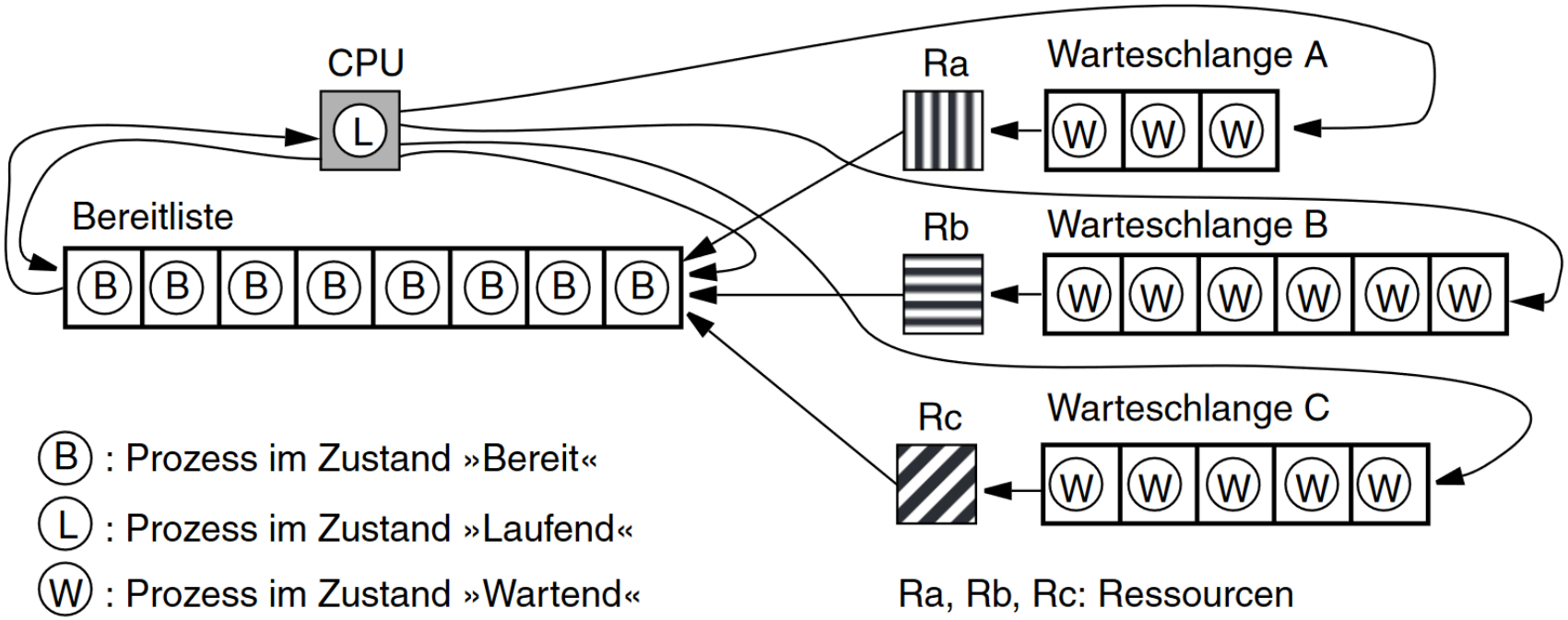
- Auf konzeptioneller Ebene:

  Betriebssysteme stellt verschiedene Warteschlangen zur Verwaltung wartender Prozesse bereit

- Warteschlangen fassen die Prozesse anhand der Art des Wartens zusammen:
  - Eine Warteschlange für ablaufbereite Prozesse, sogenannte *Ready List*
  - Entweder: Mehrere Warteschlangen für die jeweiligen Warteereignisse:
    - Warten auf Daten
    - Warten auf Zeit
    - Warten auf Meldungen
  - Oder: Zusammenfassung aller wartenden Prozesse in einer gemeinsamen Warteschlange

Implementierung des Scheduling (II)

- Schematische Darstellung der konzeptionellen Prozessverwaltung:



Quelle: *[BS15]*

Implementierung des Scheduling (III)

- Konzeptionelle Prozessverwaltung ist ein Modell, das Auskunft über Verknüpfungen von Prozessen mit CPU oder Ressourcen-Warteschlangen abbildet
- Sogenannte Prozesszuteilungsstrategie legt hierbei fest, welcher Prozess als nächster CPU-Rechenzeit zugeteilt bekommt und von einem inaktiven auf einen aktiven Zustand wechselt
- Im Folgenden: Nähere Betrachtung der Funktionsweise von Prozesszuteilungsstrategien

Prozessorzuteilungsstrategien (I)

- Prozessorzuteilungsstrategie legt fest, welcher Prozess als nächster CPU-Rechenzeit bekommt, bzw. wann ein Prozess von der CPU genommen wird
- Dabei vier Möglichkeiten aus Sicht eines Prozesses:
  - (1) Der Prozess bekommt die CPU zugeteilt
  - (2) Dem Prozess wird die CPU unfreiwillig entzogen
  - (3) Dem Prozess wird die CPU aufgrund des Ablaufs seiner Rechenzeit (Zeitquantum) entzogen
  - (4) Der Prozess wartet auf eine Ressource

Prozessorzuteilungsstrategien (II)

- (1) Der Prozess bekommt die CPU zugeteilt
  - Prozess wird aus der Bereitliste entfernt
  - Prozess wird der CPU zugeordnet
  - Welcher Prozess die CPU in diesem Schritt zugeteilt bekommt, ist abhängig von der verwendeten Strategie

Prozessorzuteilungsstrategien (III)

- (2) Dem Prozess wird die CPU unfreiwillig entzogen
  - Prozess hat CPU zwar zugeteilt, wird aber durch Betriebssystem aufgrund eines höher priorisierten Ereignisses unterbrochen
  - Prozess wechselt zurück in die Bereitliste an die vorderste Position

Prozessorzuteilungsstrategien (IV)

- (3) Dem Prozess wird die CPU aufgrund des Ablaufs seiner Rechenzeit (Zeitquantum) entzogen
  - Prozess hat die ihm zur Verfügung stehende CPU-Rechenzeit aufgebraucht
  - Prozess wechselt in die Bereitliste an die hinterste Position seiner Priorität

Prozessorzuteilungsstrategien (V)

- (4) Der Prozess wartet auf eine Ressource
  - Prozess ist im blockierenden Zustand
  - Prozess wechselt in die entsprechende Ressourcen-Warteschlange

Prozessorzuteilungsstrategien (VI)

- Bei der CPU-Zuteilung: Unterscheidung zwischen Techniken/Mechanismen und Strategien
- Techniken/Mechanismen zielen auf konkrete, physikalische Umsetzung der Datenhaltung und Realisierung der Zuteilung ab; Techniken fragen nach dem „Wie“
- Strategien befassen sich mit dem „Was“ und beantworten die Frage, was genau bei der CPU-Zuteilung gemacht werden soll
- Strategien sind langlebiger als Techniken, da sie von konkreten Technologien und Betriebssystem-Implementierungen abstrahieren

Prozessorzuteilungsstrategien (VII)

- Optimierungsziele bei der Prozessorzuteilung:
  - Sind Grund für die Vielzahl an verschiedenen Prozessorzuteilungsstrategien
  - Je nach verfolgtem Optimierungsziel muss andere Strategie bei der Prozessorzuteilung gewählt werden
  - Generelle Optimierungsziele sind:
    - Durchlaufzeit (Turnaround Time): Gesamtzeit von Prozessstart bis Prozessbeendigung
    - Antwortzeit (Response Time): Zeit zwischen Eingabe und Reaktion des Systems
    - Endtermin (Deadline): Zeitpunkt, zu dem vorgegebene Aktion erfolgt sein muss
  - Weitere Ziele sind im Sinne des optimalen Ressourceneinsatzes sind z. B.:
    - Vorhersagbarkeit
    - Durchsatz
    - Prozessorauslastung

Prozessorzuteilungsstrategien (VIII)

- Prozessklassen bei der Prozessorzuteilung:
  - Nehmen Einfluss auf Auswahl der geeigneten Zuteilungsstrategie
  - Unterteilung in drei Klassen:
    - Stapelaufträge (Batch Processes):
      Kein Benutzerdialog erforderlich, da alle Eingabe- und Verarbeitungsdaten von Anfang an feststehen
    - Dialogprozesse (Interactive Processes):
      Aktionen werden interaktiv im Benutzerdialog vom Benutzer abgefragt
    - Echtzeitprozesse (Real-time Processes):
      Einhaltung vorgegebener Zeitlimits muss gegeben sein

Prozessorzuteilungsstrategien (IX)

- Verdrängende vs. nicht verdrängende Zuteilungsstrategien:
  - Verdrängung macht Aussage darüber, wann eine Strategie eine Neuzuteilung zum Prozessor durchführt
  - Aus dem Prozessflussdiagramm ergeben sich fünf mögliche Zeitpunkte für eine Neuzuteilung:
    1. Prozess wechselt von *Laufend* in *Wartend* bzw. *Blockiert*
    2. Prozess wechselt von *Wartend* in *Bereit*
    3. Prozess wechselt von *Laufend* in *Bereit*
    4. Prozess wechselt in den Zustand *Inaktiv*
    5. Prozess wechselt von *Inaktiv* auf *Bereit*

  - Eine Zuteilungsstrategie ist dann verdrängend, wenn eine Neuzuteilung zu allen Zeitpunkten stattfinden kann → präemptives Scheduling
  - Bei einer nicht verdrängenden Zuteilungsstrategie erfolgt Neuzuteilung nur zum 1. oder 4. Zeitpunkt, der Prozess „gibt die CPU selbst wieder her“
    → kooperatives / non-präemptives Scheduling