



Vorlesung Betriebssysteme

Abschnitt 3 – Prozessverwaltung

Inhalt: Prozesse / Threads / Synchronisation / Interprozesskommunikation / Scheduling

M.Sc. Patrick Eberle

Symbol	Bedeutung
	Übung
	Beispiel
	Kommentar
	Definition

- Sie können erklären, warum Prozesse ein zentrales Konzept für Betriebssysteme darstellen
- Sie kennen die Bedeutung von Prozesshierarchien, sowie die verschiedenen Prozesszustände und -übergänge
- Sie kennen das einfache Prozessmodell und sind in der Lage, dieses anhand einer schematischen Darstellung zu erläutern
- Sie wissen, was ein Multiprogrammiersystem ist und können dessen Funktionsweise und Vorteile erklären



Kapitel VI

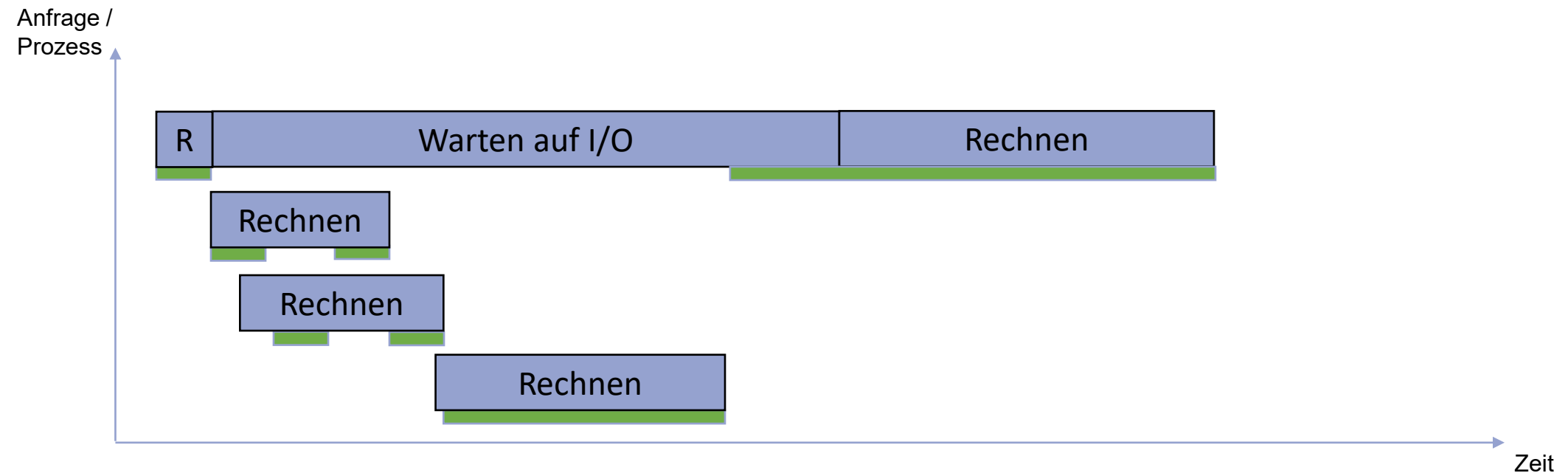
Prozesse

- Moderne Rechner sind in der Lage, mehrere Aktivitäten **parallel** bzw. **quasi-parallel** auszuführen
- In letzterem Fall: Gegenüber dem Anwender wird **Illusion erzeugt**, dass der Rechner die Aufgaben parallel abarbeitet
- Dabei wichtig: **Unterscheidung zwischen CPU-Rechenzeit und I/O-Wartezeit**
 - **CPU-Rechenzeit:** Prozessor führt tatsächliche Berechnungen durch
 - **I/O-Wartezeit:** Prozessor wartet auf Ergebnis von Eingabe- / Ausgabegerät und führt währenddessen keine Berechnungen durch
- Wunsch: CPU kann während I/O-Wartezeit andere Berechnungen durchführen, um Zeit sinnvoll zu nutzen

→ **Zentrales Konzept für die Umsetzung: Prozesse**



Beispiel 4.1: Prozessverwaltung auf einem Webserver



- Prozesse stellen zentrales Konzept eines Betriebssystems dar:
 - Prozess abstrahiert Benutzerprogramm für das Betriebssystem
 - Prozess stellt laufende Instanz eines Benutzerprogramms dar
 - Bei Einkern-Prozessoren: Prozesse ermöglichen Quasi-Parallelität
 - Unter dem Begriff Quasi-Parallelität versteht man den schnellen Wechsel einer CPU zwischen den laufenden Programmen bzw. Prozessen
 - Quasi-Parallelität kann mit Hilfe eines [Multiprogrammiersystems](#) umgesetzt werden



Definition 4.1: **Prozess**

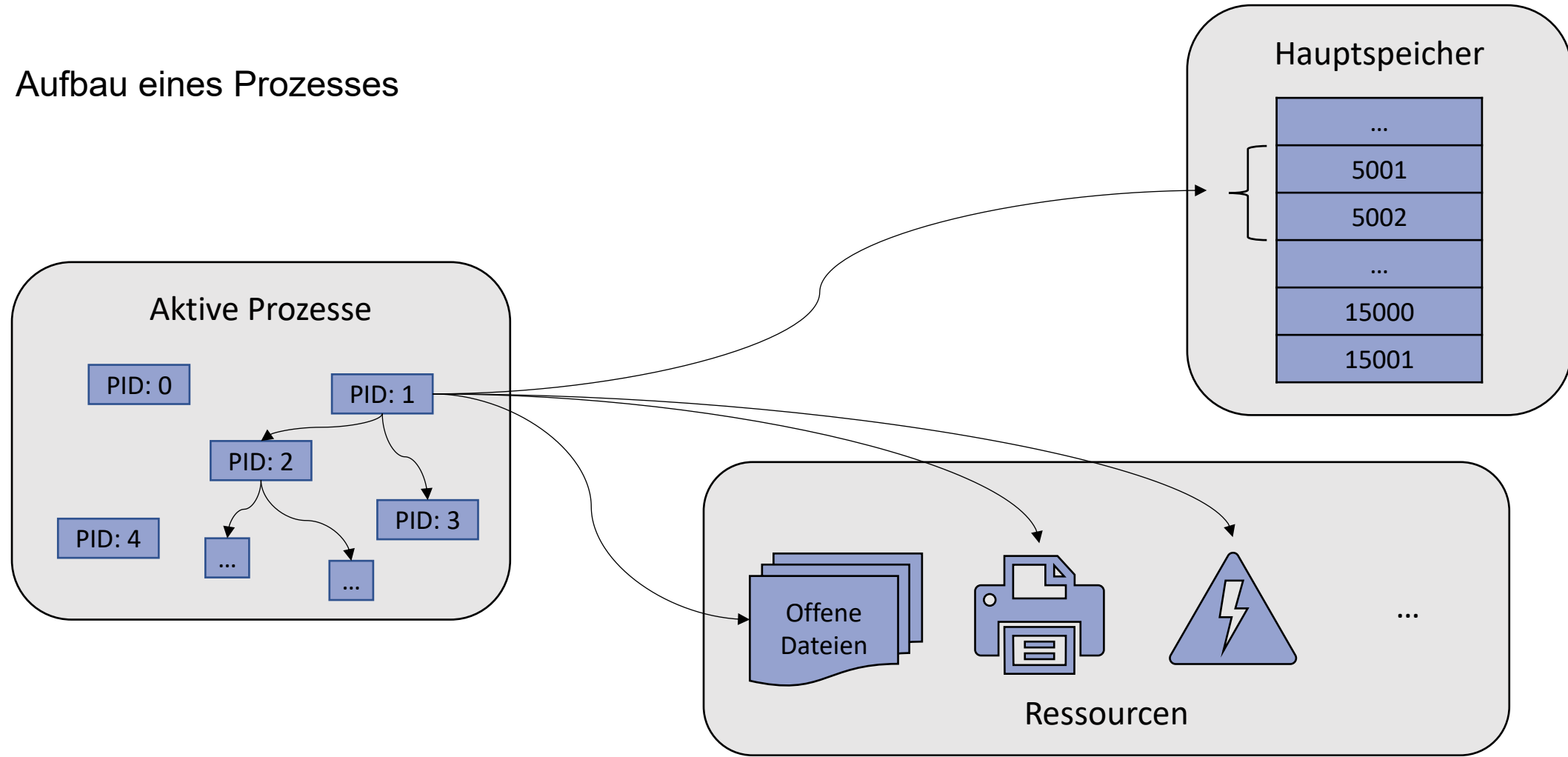
Als Prozess wird ein Programm in Ausführung (= Programminstanz) bezeichnet.

Im Prozess werden alle für die Ausführung des Programms benötigten Informationen und der aktuelle Ausführungsstatus abgespeichert.

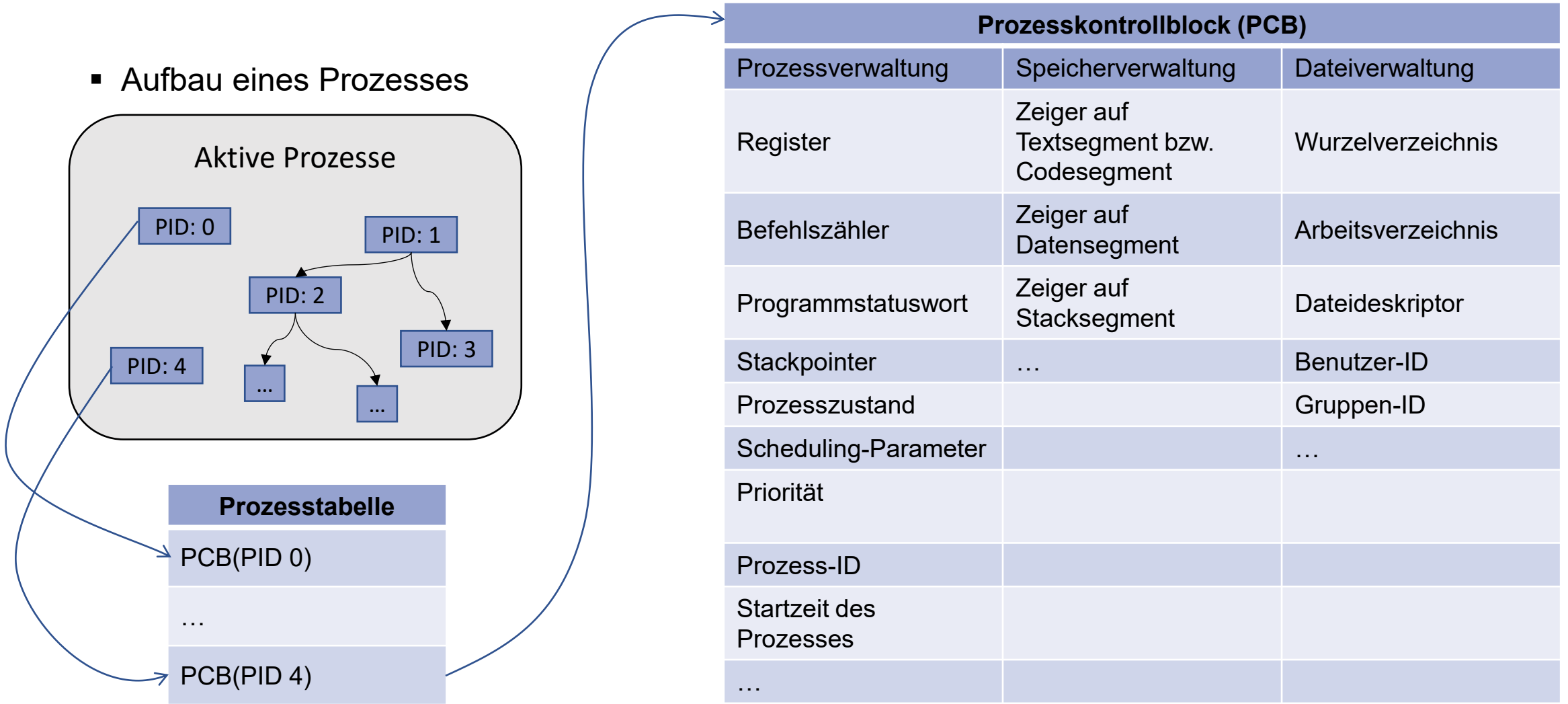
- Differenzierung: Prozess vs. Programm
 - Prozess stellt Aktivität dar
(vgl. Duden: „sich über eine gewisse Zeit erstreckender Vorgang [...]“)
 - Programm ist „Rezept“ dafür, wie die Aktivität auszuführen ist
 - Programm kann auf Festspeicher abgelegt werden, Prozess ist flüchtig
 - Dabei gilt: $Beziehung(Programm, Prozess) = 1 : (1..*)$

Der Prozess ist die Aktivität, die durch das Lesen des Programmes unter Berücksichtigung aller Eingaben, Ausgaben und des aktuellen Zustands entsteht.

- Aufbau eines Prozesses



▪ Aufbau eines Prozesses



- Aus [Prozess vs. Programm](#) geht hervor, dass ein Programm zur Ausführung einen Prozess benötigt
- Daraus folgt: Betriebssystem muss Verfahren zur Prozesserzeugung bereitstellen
 - unter Beachtung der aktuellen System-Berechtigungen
 - je nach Bedarf
 - ggf. mit Übergabeparametern
 - mit bestimmter Priorität
- Dabei: Unterscheidung zwischen zwei Prozess-Arten
 - Vordergrundprozesse: Laufen im Vordergrund, interagieren mit dem Benutzer
Beispiele: E-Mail Client, Entwicklungsumgebung, ...
 - Hintergrundprozesse: Laufen im Hintergrund, oftmals als System-Prozesse, auch „Daemons“ genannt
Beispiele: Druckerwarteschlange, Lokaler Webserver, ...

- Vier verschiedene Möglichkeiten der Prozesserzeugung:

1. **Bei System-Initialisierung:**

Der Prozess wird beim Systemstart erzeugt und ausgeführt

2. **Mittels Systemaufruf durch einen anderen Prozess:**

Ein bereits laufender Prozess führt einen Systemaufruf aus, der einen oder mehrere neue Prozesse erzeugt

3. **Mittels Benutzeranfrage zur Prozesserzeugung:**

Start eines Prozesses durch den Benutzer mittels Kommandozeile oder per Klick auf ein Anwendungssymbol

4. **Initiierung einer Stapelverarbeitung:**

Ausschließlich bei Stapelverarbeitungssystemen von Großrechnern

- Prozess-Erzeugung aus technischer Sicht:
 - Systeminterne Erzeugung eines Prozesses erfolgt **immer** durch bestehenden Prozess, der Systemaufruf zur Prozesserzeugung ausführt
 - Der bestehende Prozess (Aufruferkontext) ist dabei ein Element aus den in [Prozesserzeugung](#) vorgestellten Prozessen
 - Durch den Systemaufruf wird das Betriebssystem dazu aufgefordert, neuen Prozess zu erzeugen und auszuführendes Programm wird wie folgt übergeben:
 - Direkt bei Prozesserzeugung (Windows)
 - Indirekt nach Prozesserzeugung (UNIX)

- Prozesserzeugung unter UNIX:
 - Systemaufruf: `fork`
 - Erzeugt exakte Kopie des aufrufenden Prozesses (im Folgenden: Eltern-Prozess) mit:
 - Denselben Speicherabbild
 - Denselben Umgebungsvariablen
 - Den gleichen geöffneten Dateien
 - Zum wechseln des auszuführenden Programmes: `execve`

- Prozesserzeugung unter Windows:
 - Systemaufruf (Win32): [CreateProcess](#)
 - Erzeugt exakte Kopie des aufrufenden Prozesses / Elternprozesses mit:
 - Denselben Speicherabbild
 - Denselben Umgebungsvariablen
 - Den gleichen geöffneten Dateien
 - [CreateProcess](#) ermöglicht Prozesserzeugung und laden des auszuführenden Programmes in einem Schritt
 - [CreateProcess](#) enthält 10 Parameter, darunter u. a.:
Kommandozeilenparameter, Sicherheitsattribute, Bits für Vererbung geöffneter Dateien, Priorität, ...

- Adressräume nach Prozesserzeugung:
 - Grundsätzlich: Eltern- und Kind-Prozess haben getrennte, voneinander isolierte Adressräume
 - Jedoch: manche UNIX-Implementierungen stellen Ausnahme nach dem Copy-on-Write Prinzip dar:
 - Kind-Prozess nutzt dabei selben Speicherbereich wie Eltern-Prozess
 - Bei Veränderung durch einen der beiden Prozesse geschieht folgendes:
 1. Betroffener Speicherblock wird in privaten, dem Prozess zugehörigen Bereich kopiert
 2. Änderungen wirken sich nur auf den privaten Speicherbereich des Prozesses aus, der andere Prozess bekommt Änderungen nicht mit
 - Dadurch Verringerung des Speicheraufwands
 - Weitere Ausnahme unter UNIX: Speicherbereich mit Programmtext wird von den Prozessen geteilt, da ohnehin read-only
 - Unter Windows: Adressräume von Eltern- und Kind-Prozess von Anfang an separiert

- Nach Bearbeitung seiner Aufgabe wird Prozess i. d. R. wieder beendet
- Dabei: vier verschiedene Möglichkeiten der Prozess-Terminierung:

- 1. Normales Beenden:**

Dem Betriebssystem wird mittels Systemaufruf die ordnungsgemäße Abarbeitung des Programms mitgeteilt (UNIX: `exit`, Windows: `exitProcess`)

- 2. Freiwilliges Beenden aufgrund eines Fehlers:**

Das ausgeführte Programm stellt eine Unregelmäßigkeit im Programmablauf fest, teilt dies dem Benutzer mit (Logging, Konsolenausgabe, ...) und beendet sich selbst

- 3. Unfreiwilliges Beenden aufgrund eines Fehlers:**

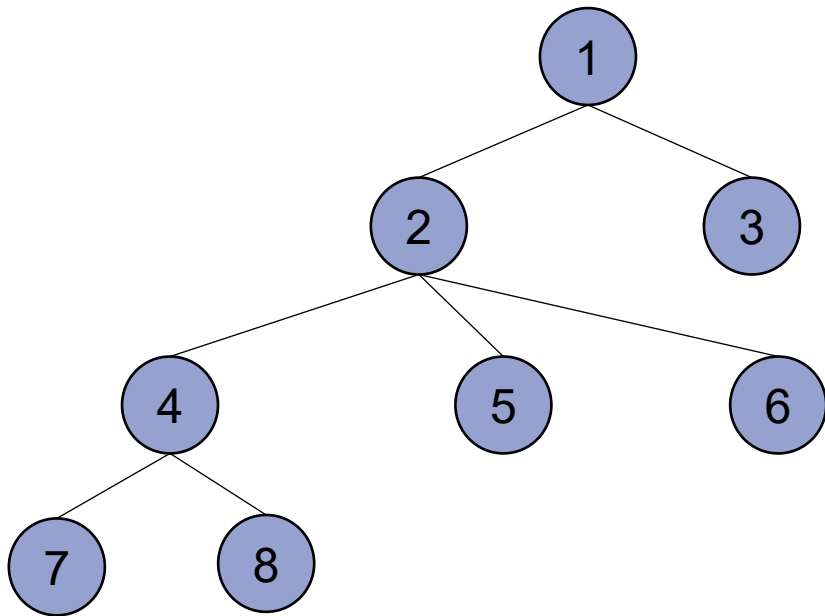
Das ausgeführte Programm wird durch einen schwerwiegenden Fehler unterbrochen (Programmierfehler: Speicherzugriffsverletzung oder Pufferüberlauf, Division durch null, ...) und vom Betriebssystem beendet

4. Unfreiwillige Beendigung durch einen anderen Prozess:

Ein anderer Prozess führt einen Systemaufruf aus, mit dem er das Betriebssystem auffordert, den Prozess zu beenden (UNIX: `kill`, Windows: `TerminateProcess`)

- Aufrufender Prozess benötigt zwingend entsprechende Berechtigung zur Beendigung des anderen Prozesses

- Aus dem Vorhandensein von Eltern- und Kind-Prozessen ergibt sich, dass Prozesse eine Art Hierarchie einnehmen können
- Dabei gilt: Ein Kind-Prozess weist nur einen Elternteil auf, und nicht wie in der Natur üblich zwei



Betrachteter Prozess	Eltern-Prozess	Kind-Prozess(e)
1	-	2, 3
2	1	4, 5, 6
3	1	-
4	2	7, 8
5	2	-
6	2	-
7	4	-
8	4	-

- Relevanz von Prozesshierarchien unter UNIX:
 - Signale / Kommunikation:
 - Prozess-Hierarchie wird als Prozessfamilie aufgefasst
 - Signale werden an alle Prozesse, die Bestandteil der Familie sind, gesendet
 - Jeder Prozess der Familie entscheidet für sich selbst, wie ankommendes Signal verarbeitet wird
 - Init-Prozess:
 - UNIX initialisiert sich direkt nach dem Systemstart mit Hilfe des `init` Prozesses selbst
 - Alle nachfolgenden Prozesse werden vom `init` Prozess abgespalten
 - Dadurch:
`init` Prozess ist Eltern-Prozess aller Prozesse innerhalb des Systems \Leftrightarrow
es existiert nur genau ein Prozessbaum, dessen Root-Element der `init` Prozess ist

▪ Einschub: Signale und Kommunikation

- Signale stellen eine Möglichkeit zur Interprozess-Kommunikation dar, die auf Software-Interrupts basiert
- Prozess-Sicherheit: Ein Prozess kann Signale nur an Mitglieder seiner eigenen Prozessgruppe bzw. Prozess-Familie senden
- Mitglieder sind dabei: der Elternprozess (und weitere Vorfahren), Geschwister, Kinder und weitere Nachfahren
- Prozesse können dem System mitteilen, was bei einem ankommenden Signal passieren soll:
 1. Signal ignorieren
 2. Signal abfangen (und entsprechende Behandlungsroutine implementieren)
 3. Prozess beenden

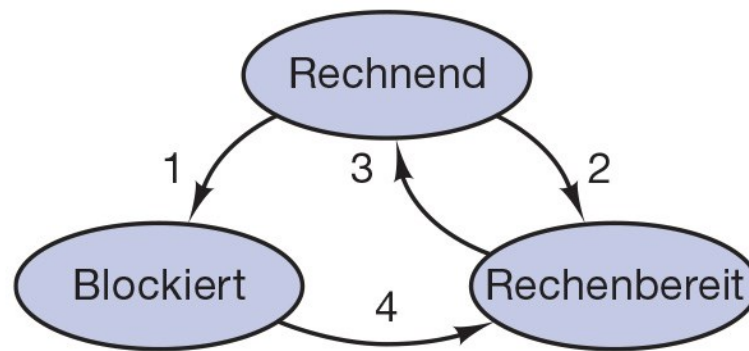
- Einschub: Signale und Kommunikation
 - Beispielhafter Auszug über vom POSIX-Standard geforderte Signale:

Signal	Ursache
SIGALRM	Zeitgeber löst Alarm aus
SIGFPE	Gleitkommafehler aufgetreten, z. B. Teilen durch 0
SIGKILL	Abbruch des Prozesses, der nicht abgefangen oder ignoriert werden kann
SIGSEGV	Prozess hat ungültige Speicheradresse referenziert
SIGTERM	Anfrage nach einem geregelten Beenden des Prozesses

- Relevanz von Prozesshierarchien unter Windows:
 - Keine explizite hierarchische Anordnung
 - Alle Prozesse gleichwertig
 - Eine implizite Prozesshierarchie wird mit Hilfe spezieller Tokens (auch Handle genannt) erreicht:
 - Eltern-Prozess erhält das Token bei der Erzeugung eines Kind-Prozesses, um diesen zu steuern
 - Eltern-Prozess kann Token an anderen Prozess weitergeben (Kind-Prozess enterben), wodurch die Hierarchie außer Kraft gesetzt wird
 - UNIX erlaubt dies nicht und erzwingt damit Einhaltung der Prozess-Hierarchie

- Wie aus [Motivation](#) hervorgeht, ist (Quasi-) Parallelität in modernen Systemen wichtiges Merkmal
- Prozesszustände bilden Grundlage zur Umsetzung der (Quasi-) Parallelität, indem sie Auskunft über Rechenbereitschaft eines Prozesses geben
- Dabei: Unterscheidung zwischen drei Zuständen eines Prozesses:
 - **Rechnend (Running):**
Der Prozess verarbeitet Befehle auf dem Prozessor
 - **Rechenbereit (Ready):**
Die Befehlsverarbeitung des Prozesses wurde vorrübergehend pausiert, um anderen Prozess rechnen zu lassen
 - **Blockiert (Blocked):**
Der Prozess wartet auf ein externes Signal / Ereignis und pausiert die weitere Befehlsverarbeitung

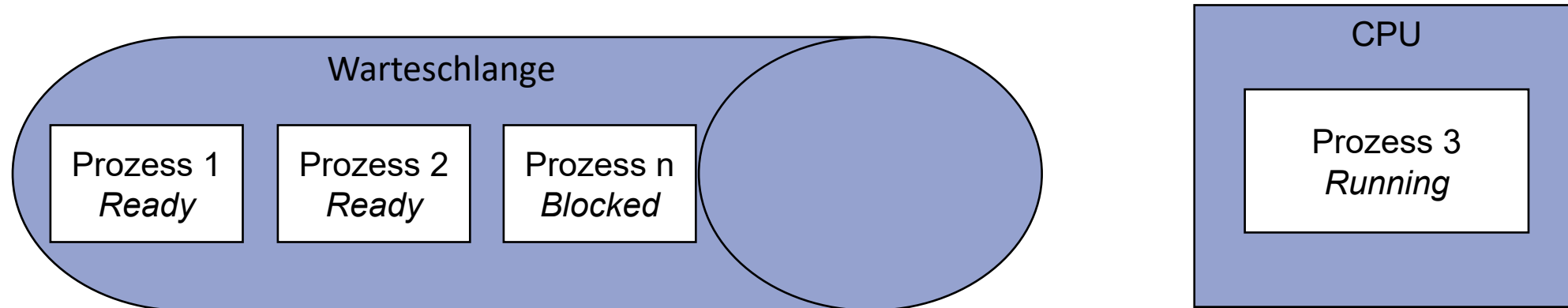
- Einfaches Prozessmodell:



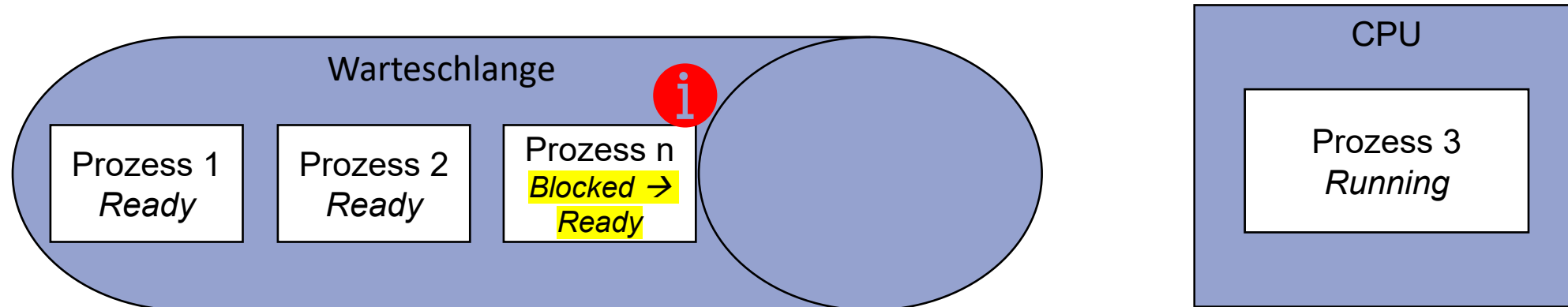
1. Prozess blockiert, weil er auf Eingabe wartet
2. Scheduler wählt einen anderen Prozess aus
3. Scheduler wählt diesen Prozess aus
4. Eingabe vorhanden

- Prozesse im Prozessmodell teilen sich physikalisch vorhandene CPU(s)
- Sogenannter Prozess-Scheduler übernimmt zentrale Aufgabe der Rechenzeit-Zuteilung
- Dabei: Änderung der Prozess-Zustände erfolgt durch sogenannten **Kontextwechsel**:
einem Prozess wird Rechenzeit entzogen, ein anderer erhält die CPU zugeteilt
- Die Zustandsinformationen des Prozesses werden in einer speziellen Tabelle, dem sogenannten **Process Control Block** gespeichert. Darunter sind z. B.: Prozess-Priorität, Zeiger auf Prozess-Warteschlangen, ...)
- Im Folgenden: Schematische Darstellung der grundsätzlichen Vorgehensweise beim Zustandsübergang

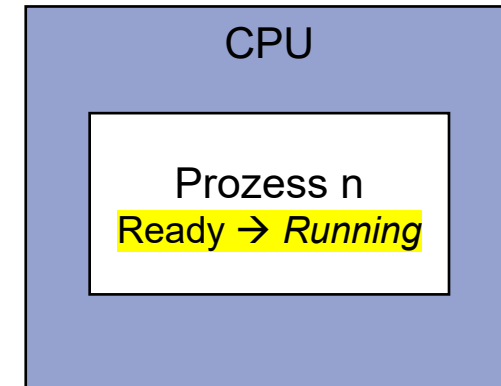
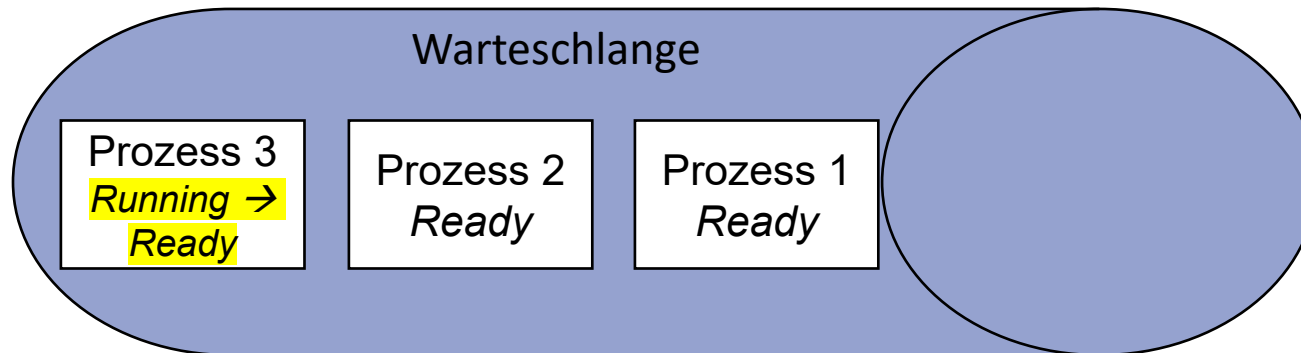
- Initial-Zustand: Prozess 3 hat Prozessor zugeteilt und wird ausgeführt



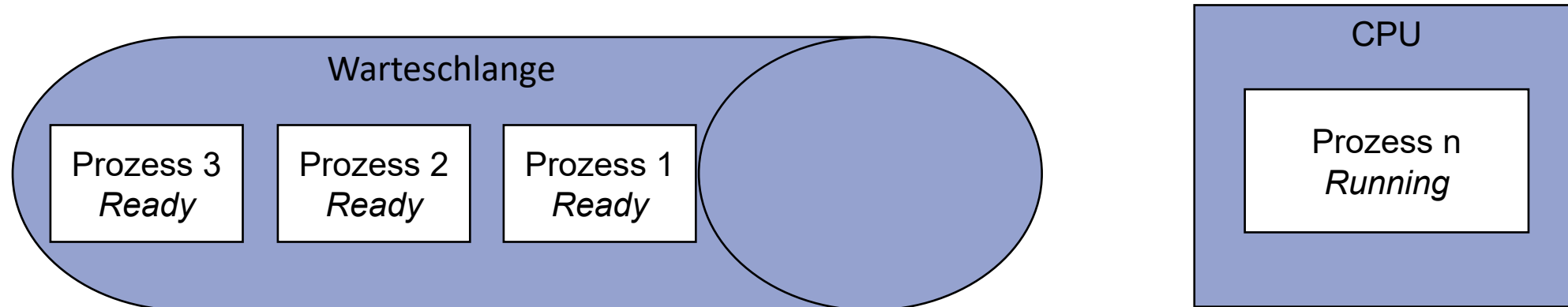
- Festplatteninterrupt tritt auf, Prozess n erhält Daten für weitere Verarbeitung
→ Zustandsübergang von *Blocked* nach *Ready*



- Prozess-Scheduler unterbricht Prozess 3 und lässt Prozess n weiterarbeiten
 - Prozess n: Zustandsübergang von *Ready* nach *Running*
 - Prozess 3: Zustandsübergang von *Running* nach *Ready*



- Prozess n hat Rechenzeit erhalten, alle anderen Prozesse sind rechenbereit





Kapitel VII

Einprogrammbetrieb und Mehrprogrammbetrieb

- Bisher: Einführung des Begriffes der Quasi-Parallelität und des Zeitmultiplexing.
„Dem Benutzer gegenüber wird die Illusion erzeugt, dass mehrere Dinge parallel ablaufen.“
- Dabei: Keine Aussage über die betriebssystemseitige Umsetzung bzw. Implementierung von Quasi-Parallelität
- Antwort darauf gibt der Mehrprogrammbetrieb

- Einprogrammbetrieb und Mehrprogrammbetrieb sind sogenannte *Betriebsarten* eines Betriebssystems
- Die *Betriebsart* eines Betriebssystems gibt Aufschluss über:
 - Art der Aufgabenverarbeitung: sequenziell, quasi-parallel oder parallel
 - Aufteilung des Hauptspeichers in Prozessräume
 - Notwendigkeit von Scheduling
- Daher zunächst: Definition der Begriffe Einprogrammbetrieb und Mehrprogrammbetrieb



Definition 5.1: **Einprogrammbetrieb**

Unter dem Einprogrammbetrieb versteht man die Betriebsart eines Rechnersystems, in der zu einer Zeit nur ein einziges Programm in den Speicher geladen und ausgeführt werden kann. Es findet folglich keine parallele und auch keine quasi-parallele Verarbeitung statt.



Definition 5.2: **Mehrprogrammbetrieb**

Unter dem Mehrprogrammbetrieb (auch: Multiprogramming, Multiprogrammierung) versteht man die Betriebsart eines Rechnersystems, in der zu einer Zeit mehrere Programme in den Speicher geladen und quasi-parallel oder echt-parallel ausgeführt werden können.



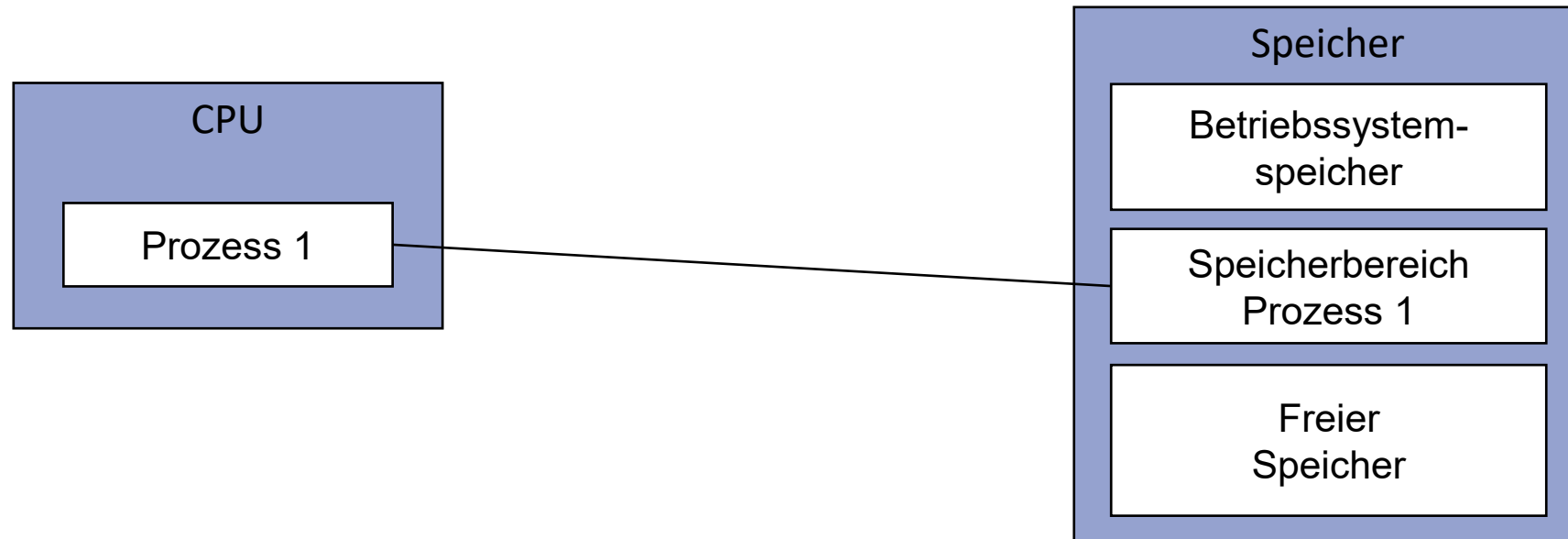
Entgegen der Darstellung mancher Fachbücher entspricht der Mehrprogrammbetrieb nicht zwangsläufig dem Multiprocessing und sollte daher nicht synonym verwendet werden (Vgl. hierzu: [Definition Multiprocessing](#)).

- In der Praxis oftmals: Ähnliche oder gar synonyme Verwendung der Begriffe *Multiprogrammierung, Multitasking und Multiprocessing*, welche sich auf Prozesse beziehen
- Weiterer Begriff: *Multithreading*, bezieht sich auf Threads
- Tatsächlich bestehen Unterschiede zwischen den Begriffen, die im Folgenden näher erläutert werden
- Zunächst jedoch:
Betrachtung der Funktionsweise des Einprogrammbetriebs bzw. Uniprogramming und des Mehrprogrammbetriebs bzw. Multiprogramming

▪ Einprogrammbetrieb: Uniprogramming

- Uniprogramming ist ursprüngliche Betriebsart zur Programmverwaltung
- Im Hauptspeicher befindet sich zu einem Zeitpunkt nur ein einziges Programm
→ Keine Speicherunterteilung in Prozessspeicherbereiche
- Auf dem Prozessor kann zu einem Zeitpunkt nur ein einziger Prozess ausgeführt werden
- Ein Programm wird vom Start bis zur Terminierung unterbrechungsfrei ausgeführt, bevor das nächste Programm geladen und ausgeführt wird
- Uniprogramming heute bis auf wenige Einzelfälle (einfache Embedded Systeme) nicht mehr vorzufinden

- Schematische Darstellung des Uniprogramming



▪ Bewertung des Uniprogramming:

😊 Einfache Prozessverwaltung

- Kein Verwaltungsaufwand für Hin- und herschalten zwischen Prozessen (Kontextwechsel, Scheduling)
- Keine konkurrierenden Zugriffe

😊 Einfache Speicherverwaltung

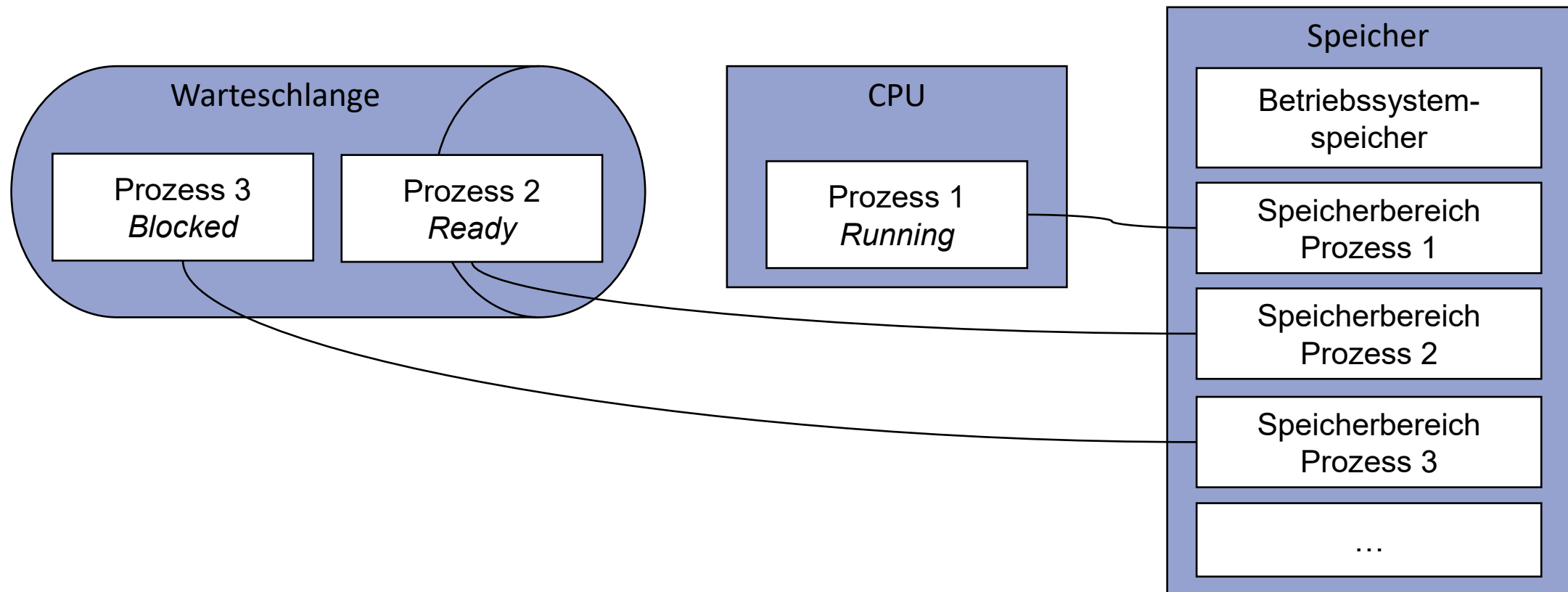
- Keine Speicherunterteilung erforderlich
- Keine Speicherzugriffsverletzungen zwischen Benutzerprozessen möglich

😞 Keine effiziente Prozessor-Auslastung: Wartet Befehlsausführung auf I/O-Operation, befindet sich Prozessor im Leerlauf

▪ Mehrprogrammbetrieb: Multiprogramming

- Multiprogrammierung liefert Antwort auf schlechte Prozessor-Effizienz beim Uniprogramming
- Hiermit: Ermöglichung des Mehrprogrammbetriebs unter Einkern-Prozessorsystemen
- Idee:
Ein Prozess wird so lange auf der CPU ausgeführt, bis er blockiert oder terminiert. Danach wird dem nächsten Prozess CPU-Rechenzeit zugeteilt.
- Anzahl x der geladenen Prozesse wird als Grad bezeichnet:
„Der Grad der Multiprogrammierung ist x “
- Dadurch: Effizientere Ausnutzung des Prozessors → Leerlaufzeiten werden auf ein Minimum reduziert

- Schematische Darstellung der Multiprogrammierung



- Effizienz der Multiprogrammierung
 - Multiprogrammierung ermöglicht mittels quasiparalleler Verarbeitung effiziente Auslastung der Ressourcen:
 - Prozessor: Reduzierung der Leerlaufzeiten, Erhöhung der Arbeitslast
 - Speicher: Zur Verfügung stehender Hauptspeicher wird besser ausgenutzt, da mehrere Prozesse gleichzeitig geladen werden können
 - Im Folgenden: Quantifizierung der Effizienz mit Hilfe rechnerischer Methoden

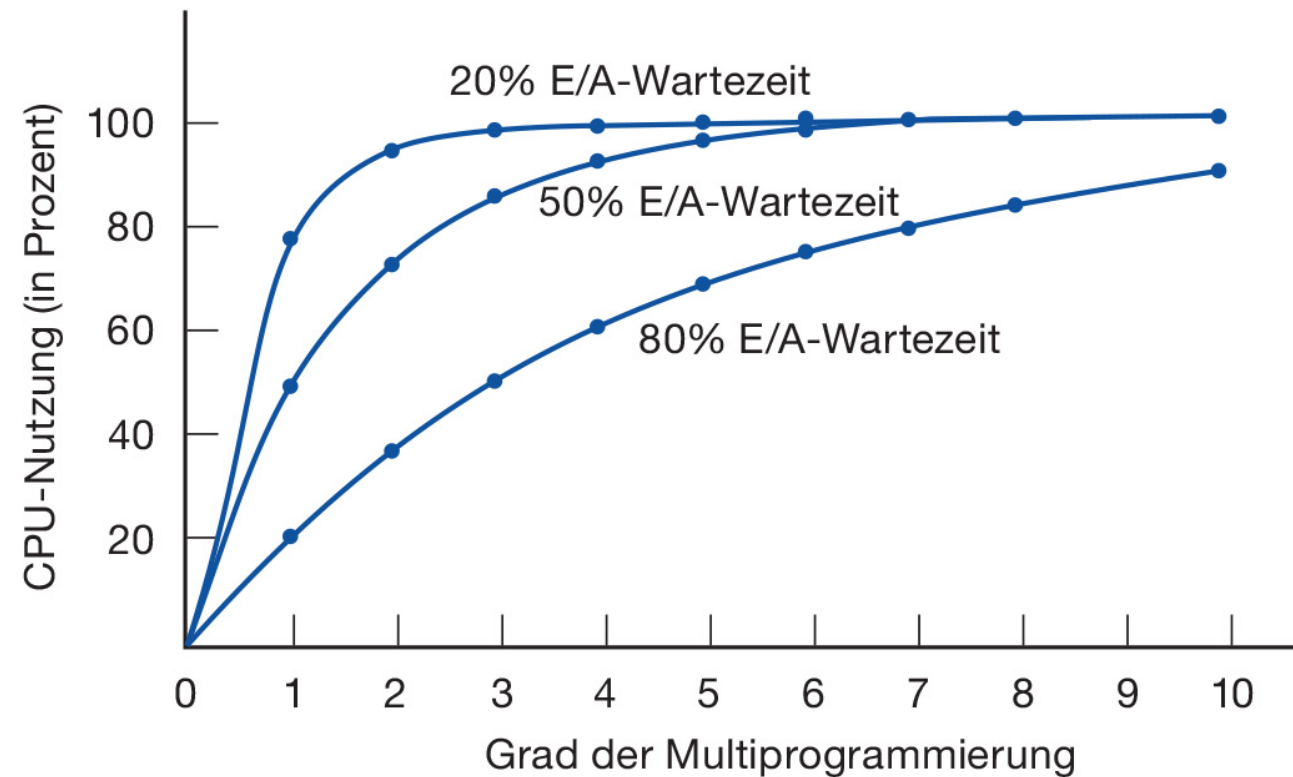
- Effizienzverhalten beim Multiprogramming
 - **Annahme:** Alle Prozesse im Hauptspeicher warten niemals gleichzeitig auf I/O-Operationen
(anders ausgedrückt: es ist immer mindestens ein Prozess rechenbereit)
 - **Gegeben:** *Prozentsatz p = Durchschnittlicher zeitlicher Anteil, den ein Prozess im rechnenden Zustand verbringt, gemessen an der Gesamtverweildauer des Prozesses im Hauptspeicher*
 - Dann ergibt sich der Grad der Multiprogrammierung zur effizienten Auslastung des Prozessors zu

$$\text{Grad}_{\text{Effiziente Auslastung}} = 1 / p$$

- Probabilistischer Ansatz zum Effizienzverhalten beim Multiprogramming
 - Stellt bessere Herangehensweise zur Messung der CPU-Effizienz dar
→ beruht nicht auf der Annahme, dass alle Prozesse niemals gleichzeitig auf I/O-Operationen warten
 - Stattdessen:
Betrachtung der Eintrittswahrscheinlichkeit für den Worst-Case (alle Prozesse warten gleichzeitig auf I/O)
mit:
Prozentsatz p = Durchschnittlicher zeitlicher Anteil, den ein Prozess auf I/O wartet
Anzahl n = Anzahl der Prozesse im Speicher
 - Dann ergibt sich die CPU-Ausnutzung wie folgt:

$$CPU\text{-Ausnutzung} = 1 - p^n$$

- Probabilistischer Ansatz zum Effizienzverhalten beim Multiprogramming



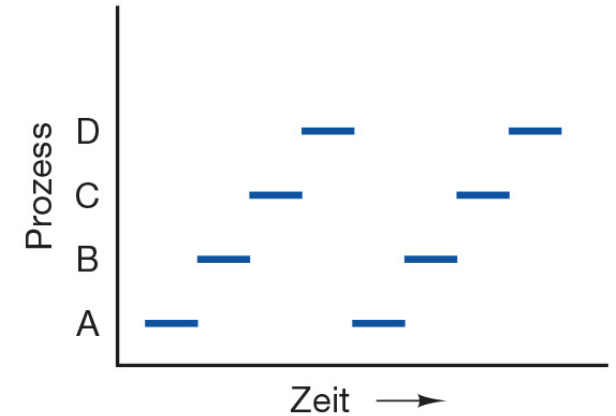


Definition 5.3: **Multiprogrammiersystem**

In einem Multiprogrammiersystem wechselt die CPU schnell zwischen den laufenden Programmen hin und her. Zu einem Zeitpunkt läuft dabei immer nur ein einziger Prozess auf der CPU.

Multiprogrammiersystem: Merkmale

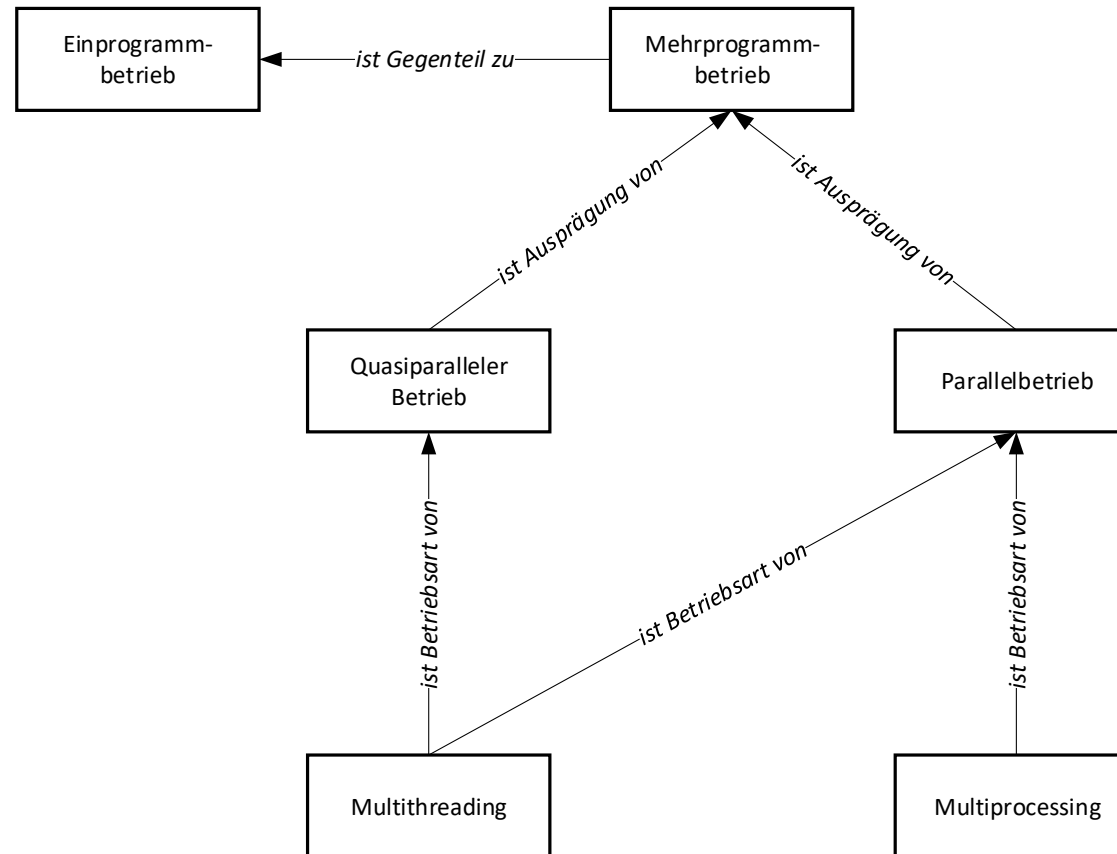
- Rechendauer pro Programm: 10 – 100 Millisekunden / Time Slot
- Schneller Wechsel zwischen den Programmen
→ Illusion der Parallelität, weshalb man von **Quasi-Parallelität** oder **Pseudoparallelität** spricht (da das System ja nicht echt parallel arbeitet)
- Zu einem Zeitpunkt: Verarbeitung eines einzigen Programms bzw. Prozesses
- Es können im Vorfeld keine Annahmen über zeitlichen Verlauf eines Prozesses gemacht werden → ungleichmäßige und schwer nachvollziehbare Prozesswechsel
- Gegenteil: **Uniprogrammiersystem**, bei dem Prozesse sequenziell, vollständig und ohne Unterbrechung nacheinander abgearbeitet werden



Multiprogrammiersystem: Bewertung

- 😊 Effiziente Nutzung des Prozessors durch Vermeidung von I/O-Wartezeiten
- 😊 Gegenüber dem Anwender: Illusion der parallelen Verarbeitung / bessere „Performance“
- 😞 Überblick über mehrere parallele Vorgänge zu behalten ist komplexe Aufgabe
- 😞 Hin- und herschalten zwischen Prozessen (→ Kontextwechsel) ist aufwändig und kostet Zeit

- Begrifflichkeiten im Ein- und Mehrprogrammbetrieb





Definition 5.4: **Multitasking**

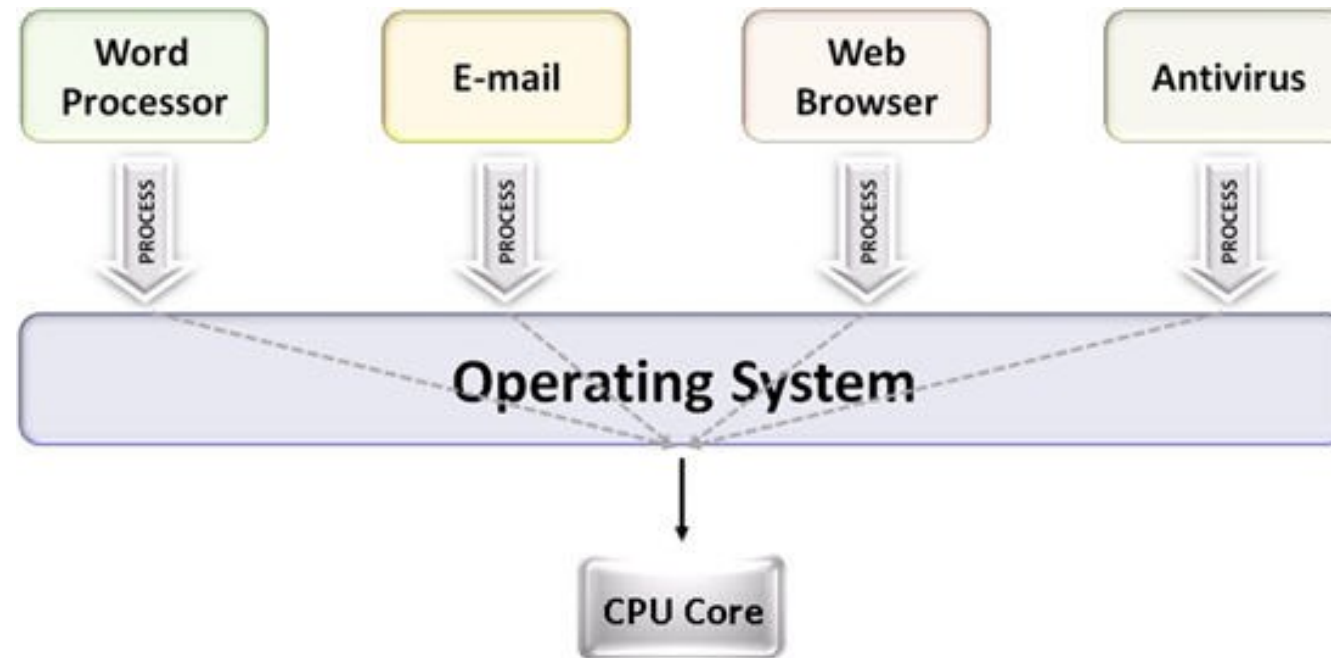
In einem Multitasking-System teilen sich mehrere Tasks (nicht näher definiert, was ein Task ist) gemeinsame Ressourcen, wie beispielsweise den Prozessor mit Hilfe von Time-Sharing Mechanismen. Das Betriebssystem übernimmt dabei die Aufgabe des schnellen hin- und herschaltens zwischen den Tasks (= Scheduling), sodass der Eindruck der Parallelität entsteht.



Definition 5.5: **Singlethreading**

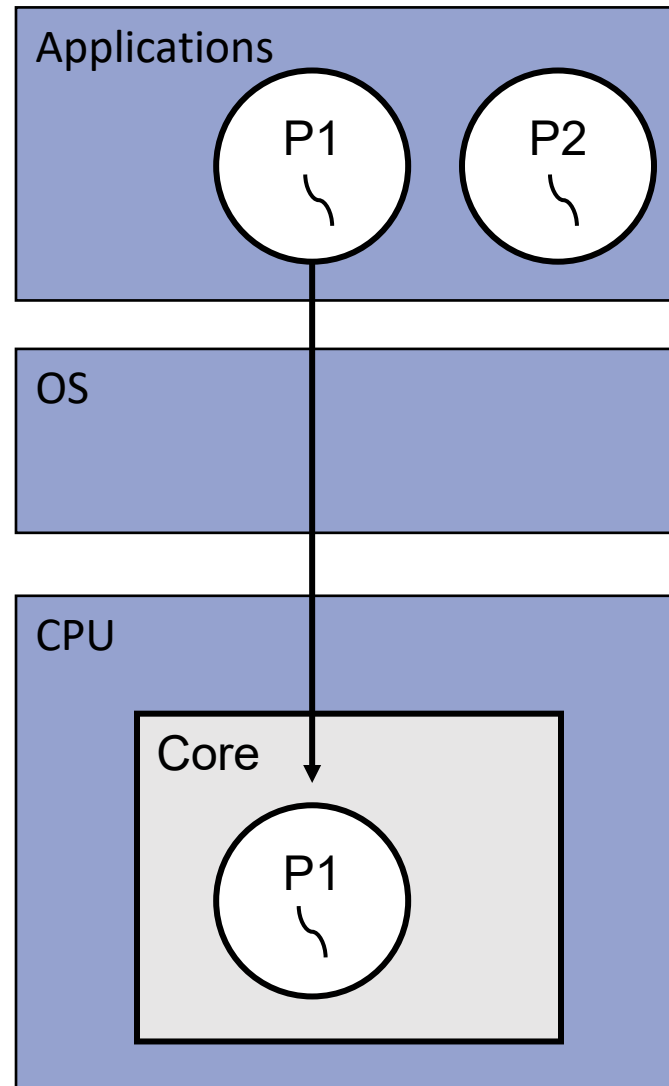
Beim Singlethreading wird zu jedem Zeitpunkt nur genau ein Task auf einmal bearbeitet.
Der Task wird dabei ohne Unterbrechung durch einen anderen Task von Anfang bis Ende abgearbeitet.

- Singlethreading



Quelle: <http://www.ni.com/white-paper/6424/de/>

- Singlethreading

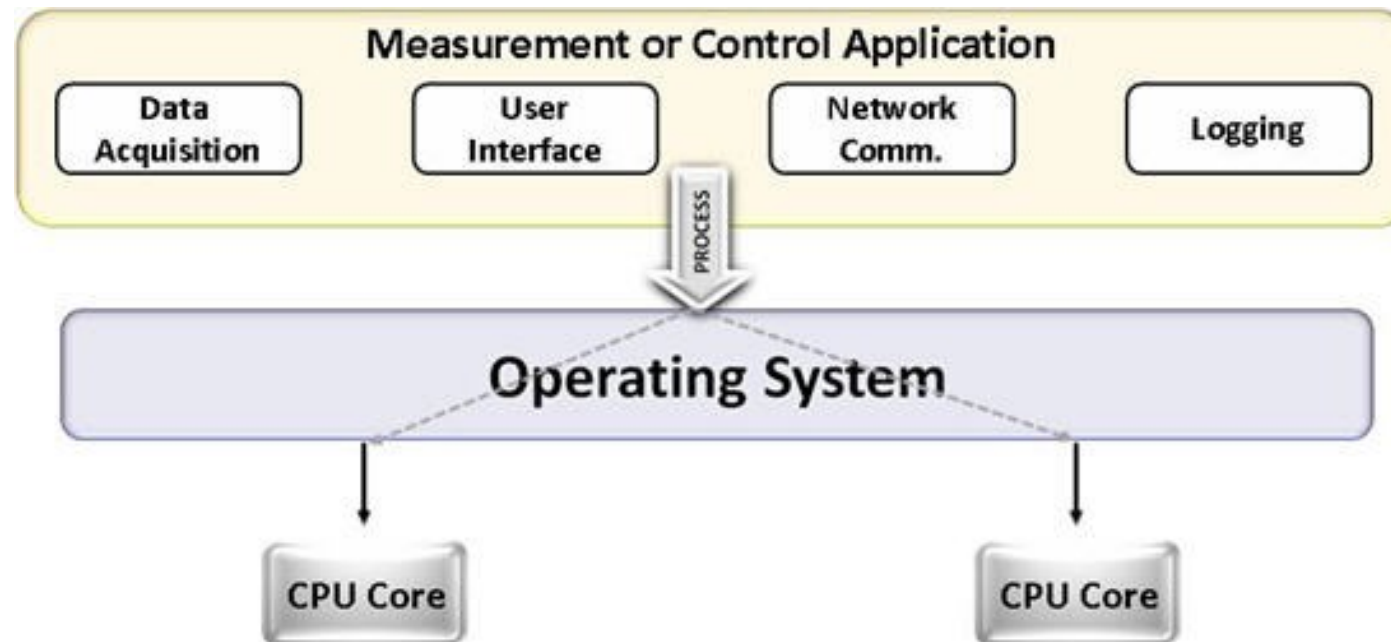




Definition 5.6: **Multithreading**

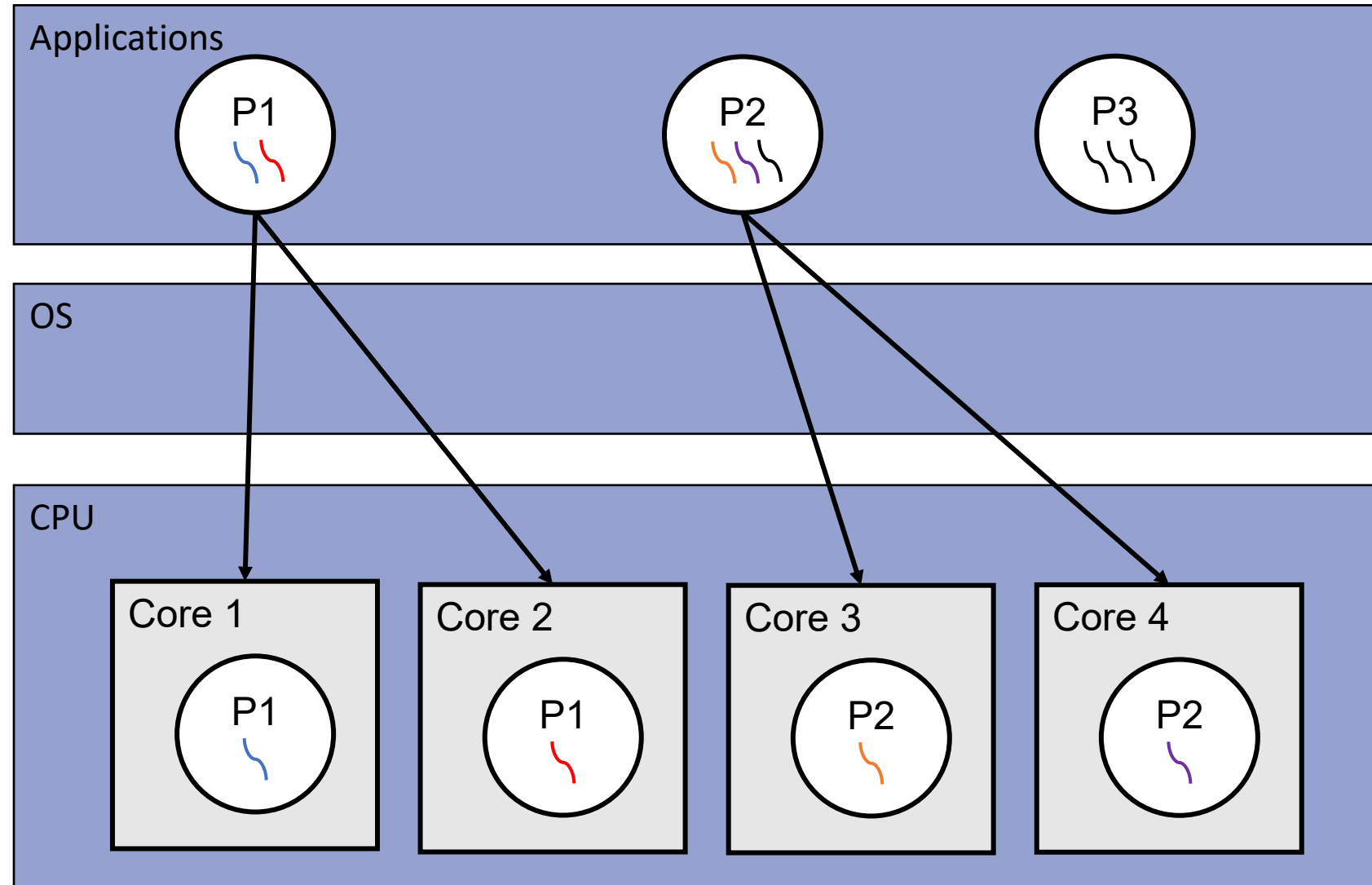
Beim Multithreading wird das Konzept des Multitasking auf ein Benutzerprogramm abgebildet. Dementsprechend ermöglicht das Multithreading die parallele bzw. quasi-parallele Verarbeitung innerhalb einer Anwendung. Das Betriebssystem verteilt dann die Rechenzeit nicht nur auf verschiedene Anwendungen, sondern zusätzlich auf die verschiedenen Ausführungsfäden der jeweiligen Anwendungen.

- Multithreading



Quelle: <http://www.ni.com/white-paper/6424/de/>

- Multithreading

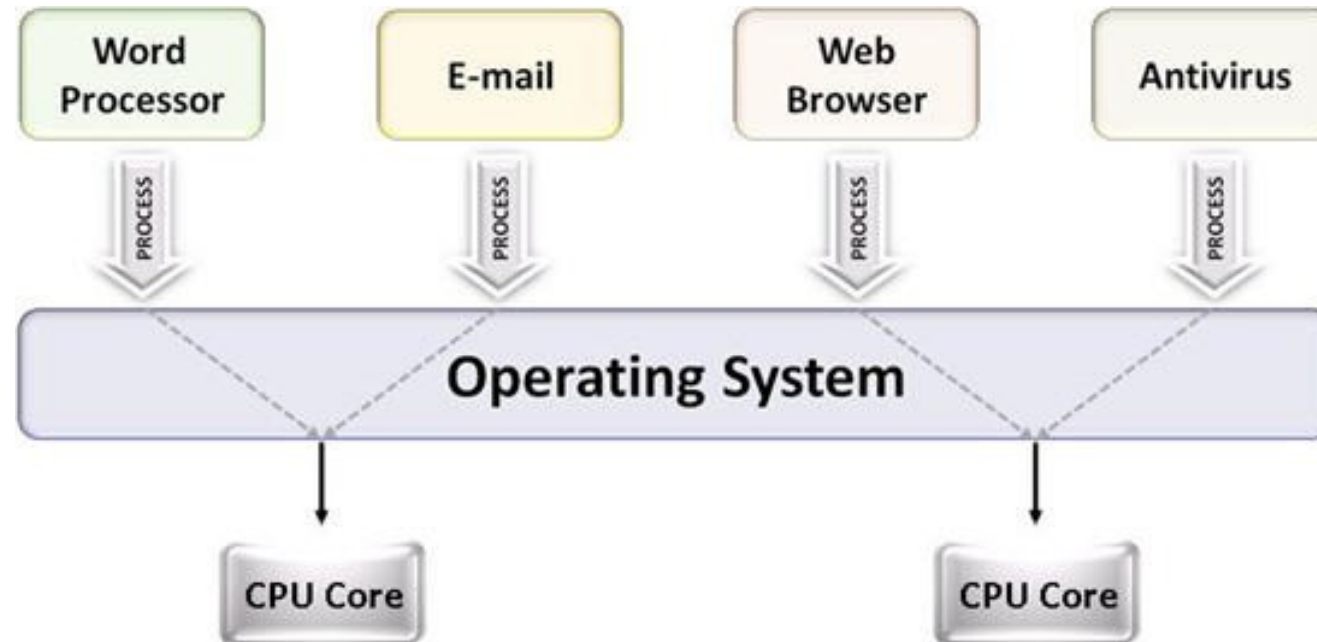




Definition 5.7: **Multiprocessing**

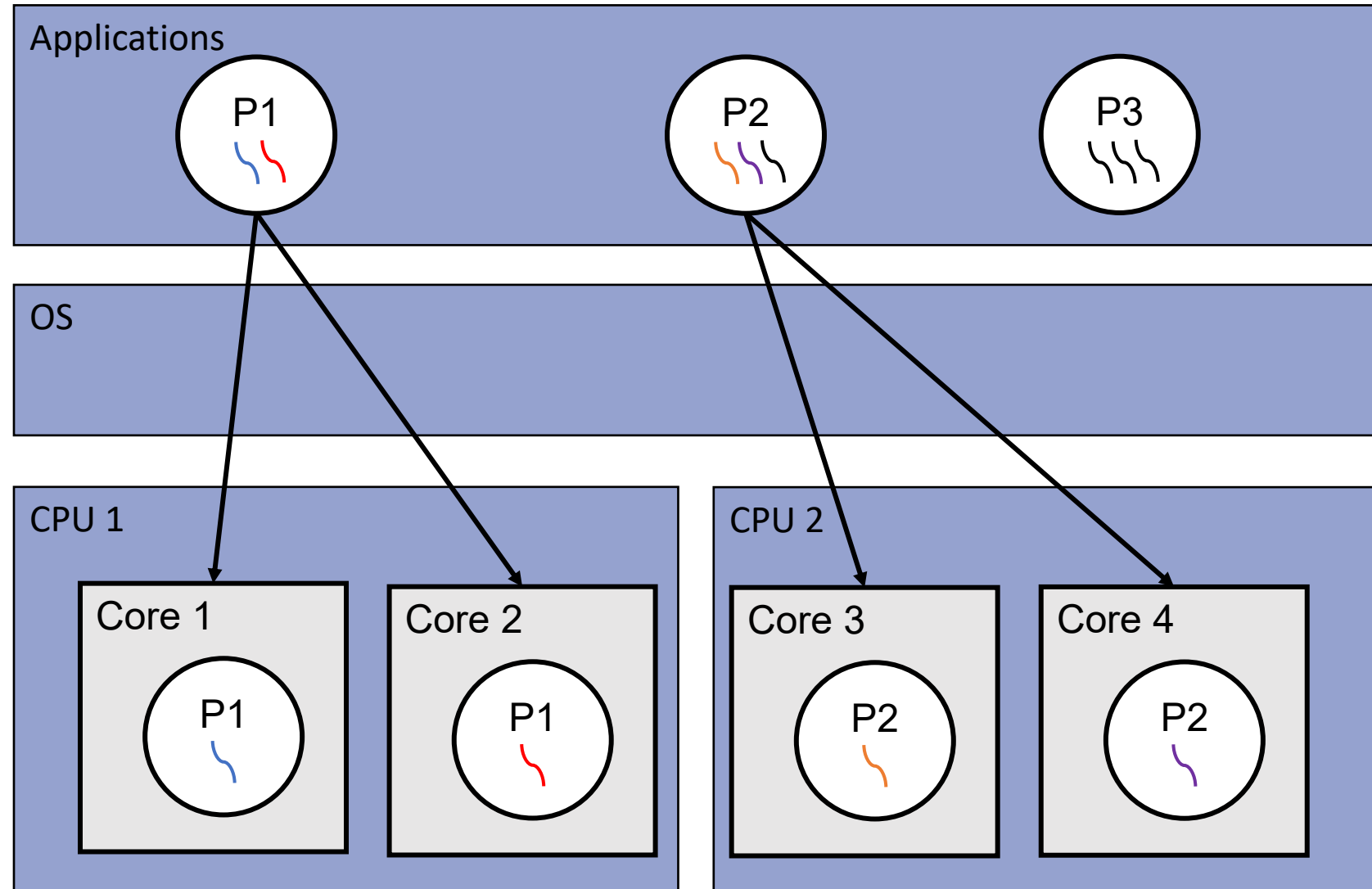
Beim Multiprocessing wird das Konzept des Multitasking auf eine echt-parallele Verarbeitung erweitert. Dies wird ermöglicht durch die Ausführung auf einem Multicore-System, welches über mehrere physikalische Prozessoren verfügt. Die Verarbeitung der einzelnen Tasks erfolgt dabei unabhängig voneinander.

- Multiprocessing



Quelle: <http://www.ni.com/white-paper/6424/de/>

- Multiprocessing



- Gegenüberstellung der Begrifflichkeiten (synonym verwendete in derselben Farbe)

