Systeme I: Betriebssysteme

Kapitel 7 Scheduling

Maren Bennewitz

Version 23.01.2013

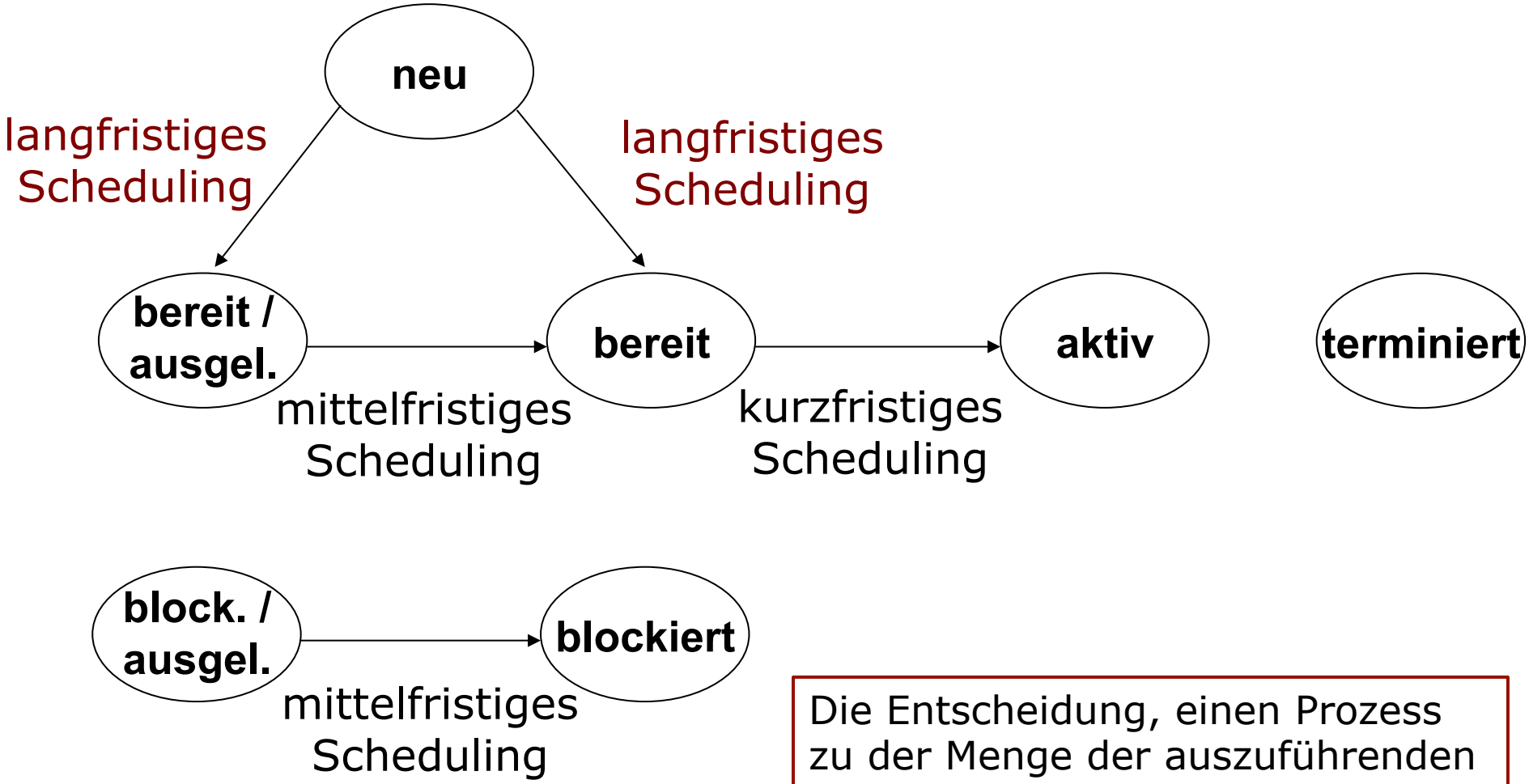
Inhalt Vorlesung

- Aufbau einfacher Rechner
- Überblick: Aufgabe, Historische Entwicklung, unterschiedliche Arten von Betriebssystemen
- Verschiedene Komponenten / Konzepte von Betriebssystemen
  - Dateisysteme
  - Prozesse
  - Nebenläufigkeit und wechselseitiger Ausschluss
  - Deadlocks
  - Scheduling
  - Speicherverwaltung

Einführung

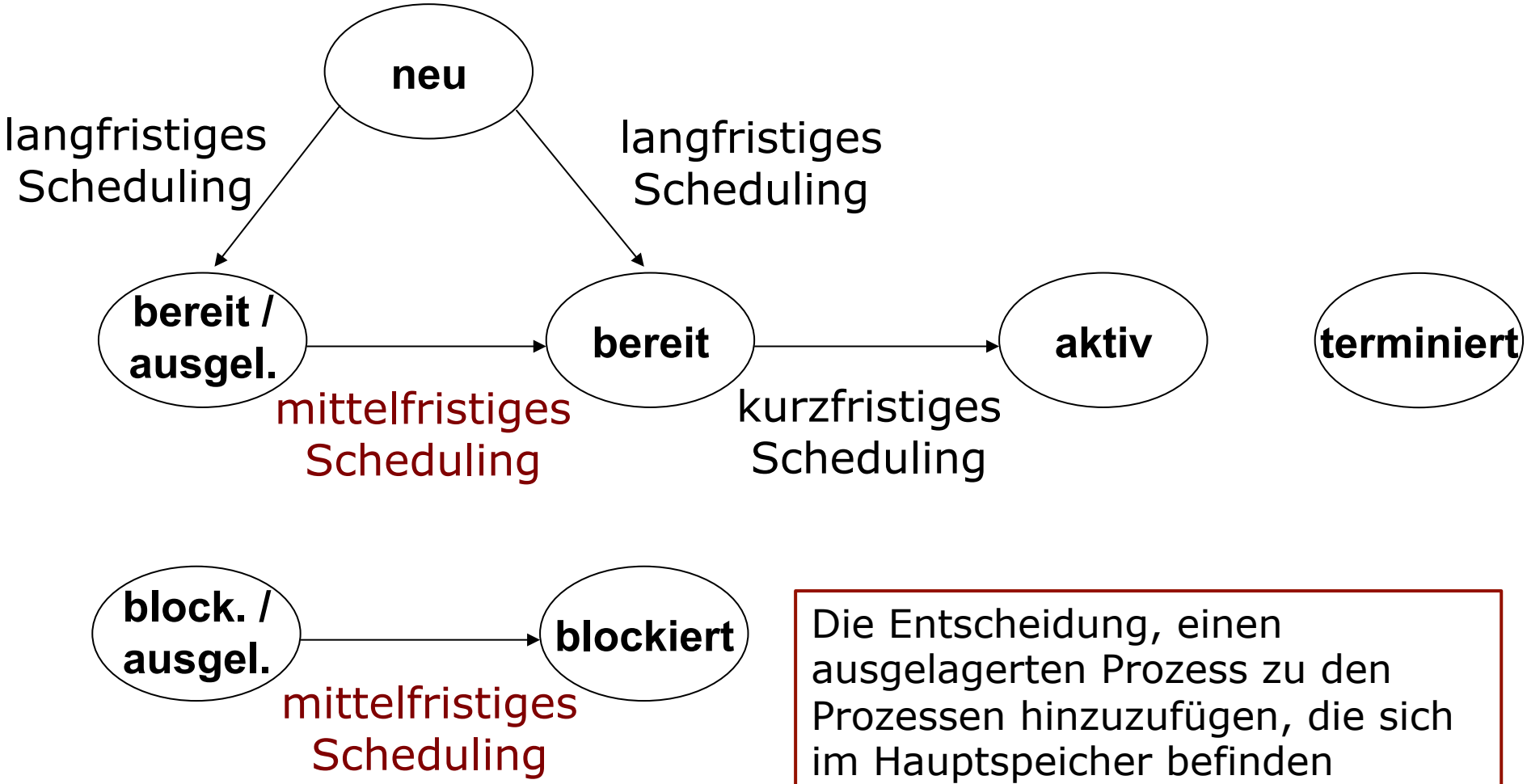
- Mehrprogrammsystem: Im Hauptspeicher werden mehrere Prozesse verwaltet
- Jeder Prozess wird entweder vom Prozessor bearbeitet oder wartet auf ein Ereignis
- Betriebssystem muss entscheiden, welche Prozesse auf der CPU/den CPUs Rechenzeit beanspruchen dürfen (Scheduling)
- Hier: Scheduling in Einprozessorsystemen zur Vereinfachung

Drei Arten von Scheduling



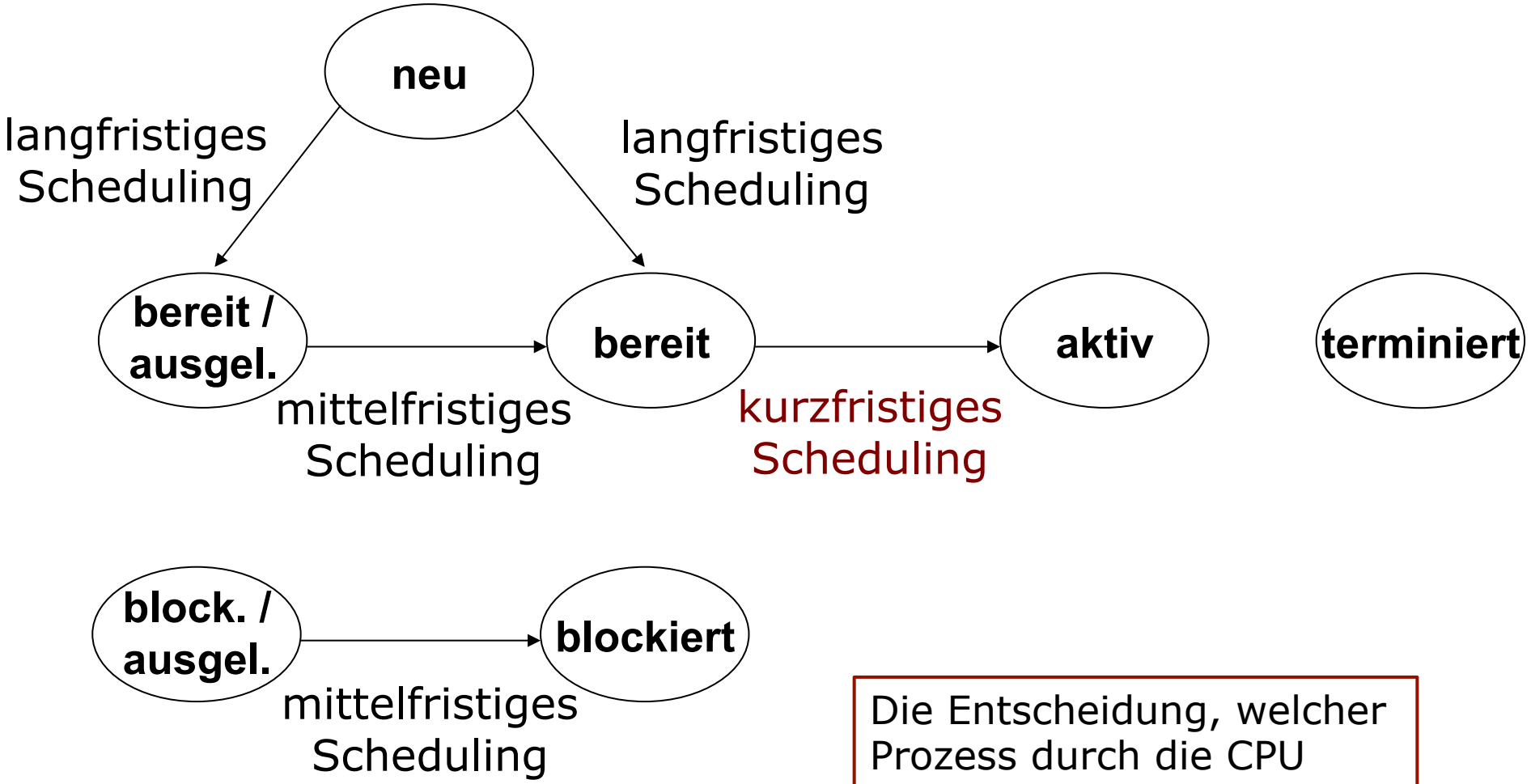
Die Entscheidung, einen Prozess zu der Menge der auszuführenden Prozess hinzuzufügen

Drei Arten von Scheduling



Die Entscheidung, einen ausgelagerten Prozess zu den Prozessen hinzuzufügen, die sich im Hauptspeicher befinden (oder umgekehrt)

Drei Arten von Scheduling



Die Entscheidung, welcher Prozess durch die CPU bearbeitet wird

Kurzfristiges Scheduling

- Der kurzfristige Scheduler weist die CPU verschiedenen (konkurrierenden) Prozessen zu, um “optimale Performance” zu erzielen
- Mehrere verschiedene Optimierungsziele sind denkbar, verschiedene Scheduling-Algorithmen existieren

Kriterien für das kurzfristige Scheduling (1)

- Benutzerorientiert:
  - Minimale Antwortzeit bei interaktivem System
  - Minimale Zeit zwischen Eingang und Abschluss eines Prozesses (Durchlaufzeit)
  - Gute Vorhersagbarkeit (unabhängig von Systemauslastung ähnliche Zeit)
- Systemorientiert:
  - Maximale CPU-Auslastung (aktive Zeit)
  - Maximale Anzahl von Prozessen, die pro Zeiteinheit abgearbeitet werden (Durchsatz, pro Stunde)

Zwischenbemerkung

Unterschied: Durchsatz/Durchlaufzeit

- Durchsatz: Anzahl der Prozesse, die vom System pro Stunde erledigt werden
- Durchlaufzeit: Durchschnittliche Zeit von Start bis Beendigung
- Hoher Durchsatz heißt nicht unbedingt niedrige Durchlaufzeit
- Für Benutzer ist eher niedrige Durchlaufzeit interessant
- Siehe auch Blatt 11, Aufgabe 3

Kriterien für das kurzfristige Scheduling (2)

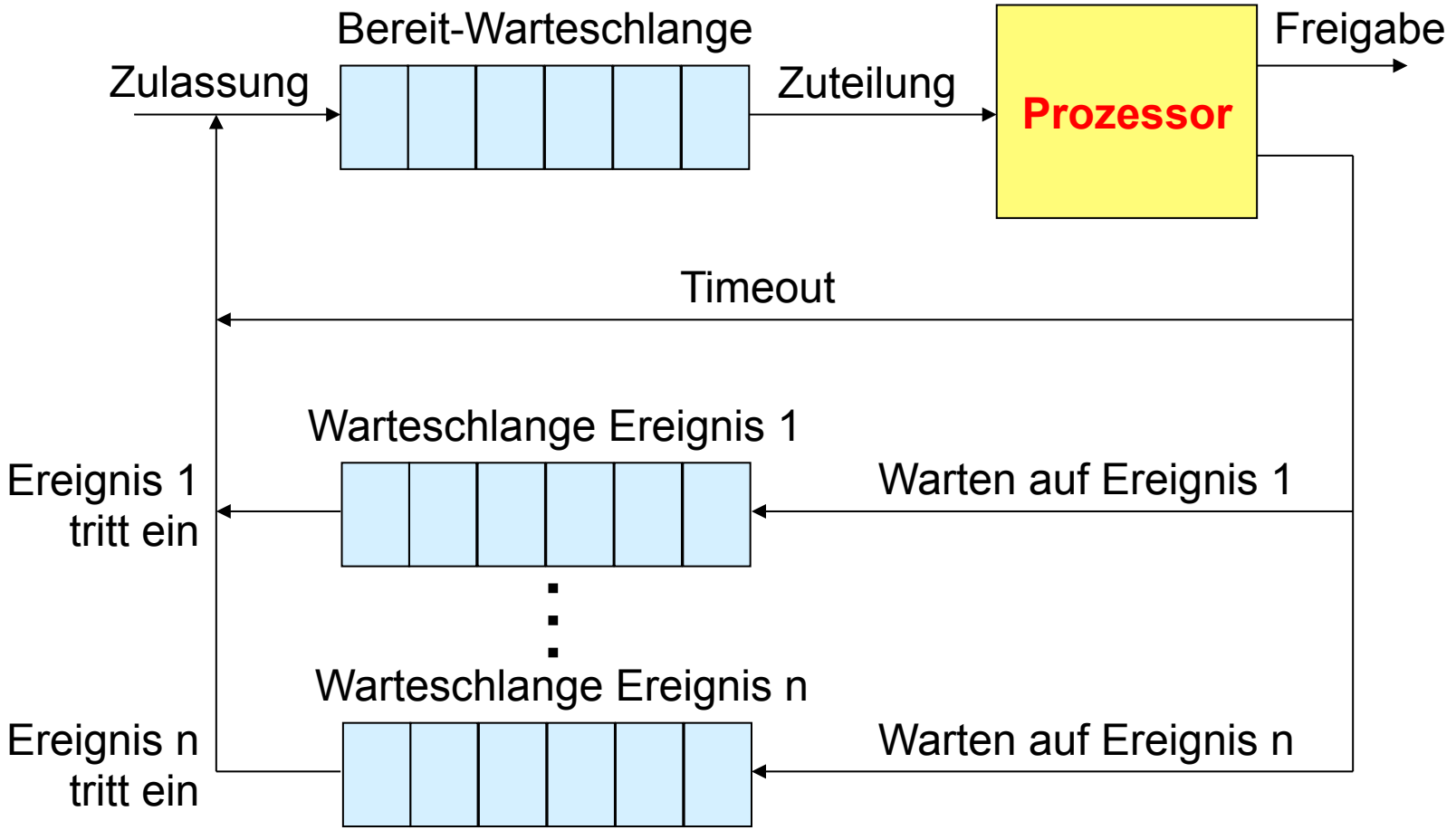
- Allgemein:
  - Fairness: Jeder Prozess erhält CPU (irgendwann)
  - Prioritäten müssen eingehalten werden
  - Effizienz: Möglichst wenig Aufwand für Scheduling selbst
- Echtzeitsysteme:
  - Einhalten von Deadlines
  - Vorhersehbares Verhalten

Kriterien für das kurzfristige Scheduling (3)

- Zwischen den Kriterien bestehen Abhängigkeiten
- Beispiel:
  - Eine gute Antwortzeit erfordert, dass zwischen Prozessen gewechselt wird,
  - Dann aber niedrigerer Durchsatz und mehr Aufwand durch Prozesswechsel
- Scheduling-Strategie muss Kompromiss schließen
- Strategiewahl abhängig von Art und Nutzung des Systems

Erinnerung: Warteschlangen

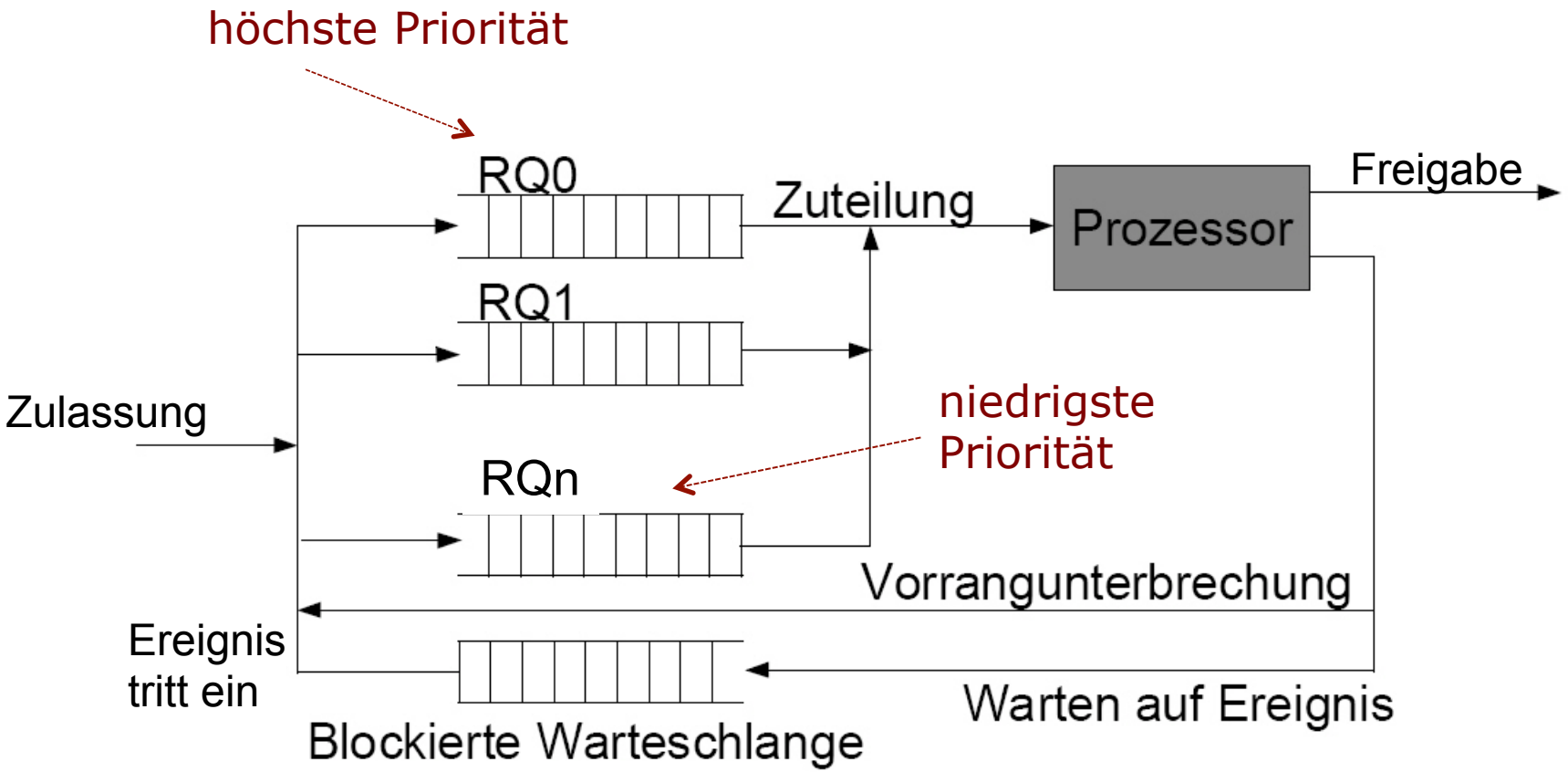
Warteschlangen für bereite Prozesse und für Prozesse, die auf Ereignisse warten



Prioritäten (1)

- In den meisten Systemen existieren Prioritäten zwischen den Prozessen
- Mehrere Warteschlangen mit bereiten Prozessen verschiedener Priorität
- Bei Entscheidung der Ablaufplanung beginnt der Scheduler mit der Warteschlange, die bereite Prozesse enthält und höchste Priorität hat
- Innerhalb Warteschlange: Scheduling-Strategie

Prioritäten (2)



Prioritäten (3)

- Erster Prozess in Warteschlange mit höchster Priorität erhält CPU
- Problem: Livelock (Verhungern) von Prozessen mit geringer Priorität
- Lösung: Ändere Prioritäten entsprechend Alter (später mehr dazu)

Scheduling-Algorithmen – Charakteristische Eigenschaften

Prozessauswahl:

- Auswahlfunktion legt fest, welcher der bereiten Prozesse als nächstes aktiv wird
- Basierend auf Ressourcenanfragen, Prioritäten oder auch Ausführungs-eigenschaften
- Drei Größen von Bedeutung:
  - w (Wartezeit auf CPU seit Erzeugung)
  - e (bisher verbrauchte CPU-Zeit)
  - s (insgesamt benötigte CPU-Zeit, geschätzt)

Scheduling-Algorithmen – Charakteristische Eigenschaften

Zeitpunkt der Auswahlentscheidung:

- Nicht-präemptives Scheduling:

  CPU kann einem Prozess nur entzogen werden, wenn er beendet ist oder blockiert ist

- Präemptives Scheduling:

  Aktueller Prozess kann vom Betriebssystem unterbrochen werden, wenn dies richtig erscheint;

  z.B. Hardwaretimer, bei Ablauf: Unterbrechung an CPU melden, Prozesswechsel, Aktivieren des Schedulers

First Come First Served (FCFS)

- Nicht-präemptive Strategie
- Strategie: Wenn ein Prozess beendet ist oder blockiert, dann kommt der Prozess an die Reihe, der schon am längsten wartet
- Auswahlfunktion: max(w)
- Implementiert durch einfache Warteschlange

First Come First Served (FCFS)

Analyse

- FCFS begünstigt lange Prozesse, kurze Prozesse können durch lange Prozesse stark verzögert werden
- FCFS begünstigt Prozesse ohne Ein-/Ausgabe (die den Prozessor vor Beendigung nicht abgeben)
- FCFS alleine nicht sehr interessant, aber häufig mit Prioritätsverfahren kombiniert

Round Robin (RR)

- Präemptive Strategie
- Strategie:
  - Scheduler wird nach Ablauf fester Zeitintervalle immer wieder aktiviert
  - Laufender Prozess wird dann in eine Warteschlange wartender Prozesse eingefügt.
  - Der am längsten wartende Prozess wird aktiviert
  - In realen Systemen ca. 20-100 ms

Round Robin (RR)

Analyse

- Länge der Zeitintervalle ist essentiell
- Zu kurz: Aufwand für viele Prozesswechsel
- Zu lang: Ähnlich FCFS
- Sinnvoll: Entsprechend durchschnittlicher benötigter CPU-Zeit
- RR begünstigt Prozesse ohne Ein-/Ausgabe etwas
- Prozesse mit Ein-/Ausgabe geben CPU ab, bevor ihr Zeitintervall abgelaufen ist, erhalten dafür aber keine “Gutschrift”

Shortest Job First (SJF) (1)

- Nicht-präemptive Strategie
- Auswahlfunktion: min(s)
- Strategie:
  - Benutzt Abschätzungen der Gesamtlaufzeit von Prozessen
  - Prozess mit kürzester geschätzter Laufzeit erhält CPU als erstes

Shortest Job First (SJF) (2)

- Beispiel: 4 Aufgaben A, B, C, D
  - A braucht 8 min, B, C, D jeweils 4 min
  - Alle Aufgaben stehen zur Zeit t = 0 bereit
- Zuerst wird B, C, oder D bearbeitet, am Schluss A, Reihenfolge also z.B. B, C, D, A
- Durchlaufzeiten:
  - B: 4 Minuten
  - C: 8 Minuten
  - D: 12 Minuten
  - A: 20 Minuten
  - Zusammen: 44 Minuten
  - Mittlere Durchlaufzeit: 44:4 =11 Minuten

Shortest Job First (SJF) (3)

- Andere Reihenfolge: A, B, C, D
- Durchlaufzeiten:
  - A: 8 Minuten
  - B: 12 Minuten
  - C: 16 Minuten
  - D: 20 Minuten
  - Zusammen: 56 Minuten
- Mittlere Durchlaufzeit: 56:4 =14 Minuten
- SJF deutlich besser

Shortest Job First (SJF) (4)

Satz:

- Seien n Prozesse P_1, ..., P_n mit Laufzeiten t_1, ..., t_n gegeben und alle zur Zeit t = 0 verfügbar
- Dann erzielt SJF die minimale durchschnittliche Durchlaufzeit

Shortest Job First (SJF) (5)

Beweis:

Annahme: Ausführungsreihenfolge $P_1, P_2, \ldots, P_n$

Berechne für alle Prozesse P_i die Durchlaufzeiten d_i:

$$
\begin{aligned}
d_1 &= t_1 \\
d_2 &= d_1 + t_2 = t_1 + t_2 \\
d_3 &= d_2 + t_3 = t_1 + t_2 + t_3 \\
&\ldots \\
d_n &= d_{n-1} + t_n = t_1 + t_2 + t_3 \ldots + t_{n-1} + t_n
\end{aligned}
$$

Also $d_i = \sum_{j=1}^{i} t_j$

Shortest Job First (SJF) (6)

Mittlere Durchlaufzeit:

$$
\begin{aligned}
d^* &= \frac{1}{n}\sum_{i=1}^{n} d_i \\
&= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{i} t_j \\
&= \frac{1}{n}(t_1 + \underline{t_1 + t_2} + \underline{t_1 + t_2 + t_3} + \ldots + \underline{t_1 + t_2 + t_3 + \ldots + t_n}) \\
&= \frac{1}{n}(n \cdot t_1 + (n-1) \cdot t_2 + (n-2) \cdot t_3 + \ldots + t_n) \\
&= \frac{1}{n} \cdot \sum_{i=1}^{n}(n+1-i) * t_i \\
&= \sum_{i=1}^{n} \frac{n+1-i}{n} \cdot t_i
\end{aligned}
$$

Shortest Job First (SJF) (7)

- Gewichtete Summe über alle t_i
- Gewicht von t_1 ist $\frac{n+1-1}{n} = \frac{n}{n} = 1$
- Gewicht von t_2 ist $\frac{n+1-2}{n} = \frac{n-1}{n}$
- Gewicht von t_n ist $\frac{n+1-n}{n} = \frac{1}{n}$

Shortest Job First (SJF) (7)

- Gewichtete Summe über alle t_i
- Gewicht von t_1 ist $\frac{n+1-1}{n} = \frac{n}{n} = 1$
- Gewicht von t_2 ist $\frac{n+1-2}{n} = \frac{n-1}{n}$
- Gewicht von t_n ist $\frac{n+1-n}{n} = \frac{1}{n}$
- Gewichtete Summe ist dann am kleinsten, wenn $t_1 \leq t_2 \leq \ldots \leq t_n$
- SJF führt zur geringsten mittleren Durchlaufzeit

Shortest Job First (SJF) (8)

Analyse

- Erzielt minimale durchschnittliche Durchlaufzeit, sofern alle Prozesse gleichzeitig verfügbar (s. Blatt 11, Aufg. 1)
- Kurze Prozesse bevorzugt
- Probleme:
  - Es besteht die Gefahr, dass längere Prozesse verhungern
  - Evtl. viel Aufwand für Prozesswechsel
  - Wie erhält man Abschätzungen der Gesamtlaufzeit von Prozessen?
  - SJF nicht geeignet für Allzweck-OS

Shortest Remaining Time (SRT)

- Präemptive Variante von SJF
- Auswahlfunktion: min(s-e)
- Strategie:
  - Benutzt Abschätzungen der Restlaufzeiten von Prozessen
  - Prozess mit kürzester geschätzter Restlaufzeit erhält CPU
  - Keine Unterbrechungen laufender Prozesse durch Timer
  - Stattdessen: Auswertung der Restlaufzeiten nur, wenn ein anderer Prozess bereit wird

Shortest Remaining Time (SRT)

Analyse

Ähnliche Probleme wie SJF

- Benachteiligt lange Prozesse
- Verhungern längerer Prozesse möglich
- Aufwand für Prozesswechsel und Aufzeichnen von Ausführungszeiten
- Abschätzungen der Gesamtlaufzeit von Prozessen müssen bekannt sein
- Aber u.U. bessere Durchlaufzeit, weil kurze bereite Prozesse aktiven längeren Prozessen sofort vorgezogen werden

Highest Response Ratio Next (HRRN)

- Nicht-präemptiv
- Auswahlfunktion: max(w+s / s)
- Strategie:
  - Basiert auf
    - *s* = Geschätzte Gesamtlaufzeit eines Prozesses
    - *w* = bisherige Wartezeit auf CPU eines Prozesses
    - Normalisierte Durchlaufzeit (“Response Ratio”)
      $R = (w+s) / s$
    - Ein Prozess startet mit $R = 1.0$
  - Der Prozess mit der höchsten Response Ratio *R* erhält die CPU

Highest Response Ratio Next (HRRN)

Analyse

- Für kurze wartende Prozesse wächst *R* schnell an
- HRRN begünstigt auch kurze Prozesse
- Aber: Keine Livelocks für längere Prozesse
- Ähnliches Problem wie SJF, SRT: Laufzeitabschätzungen benötigt

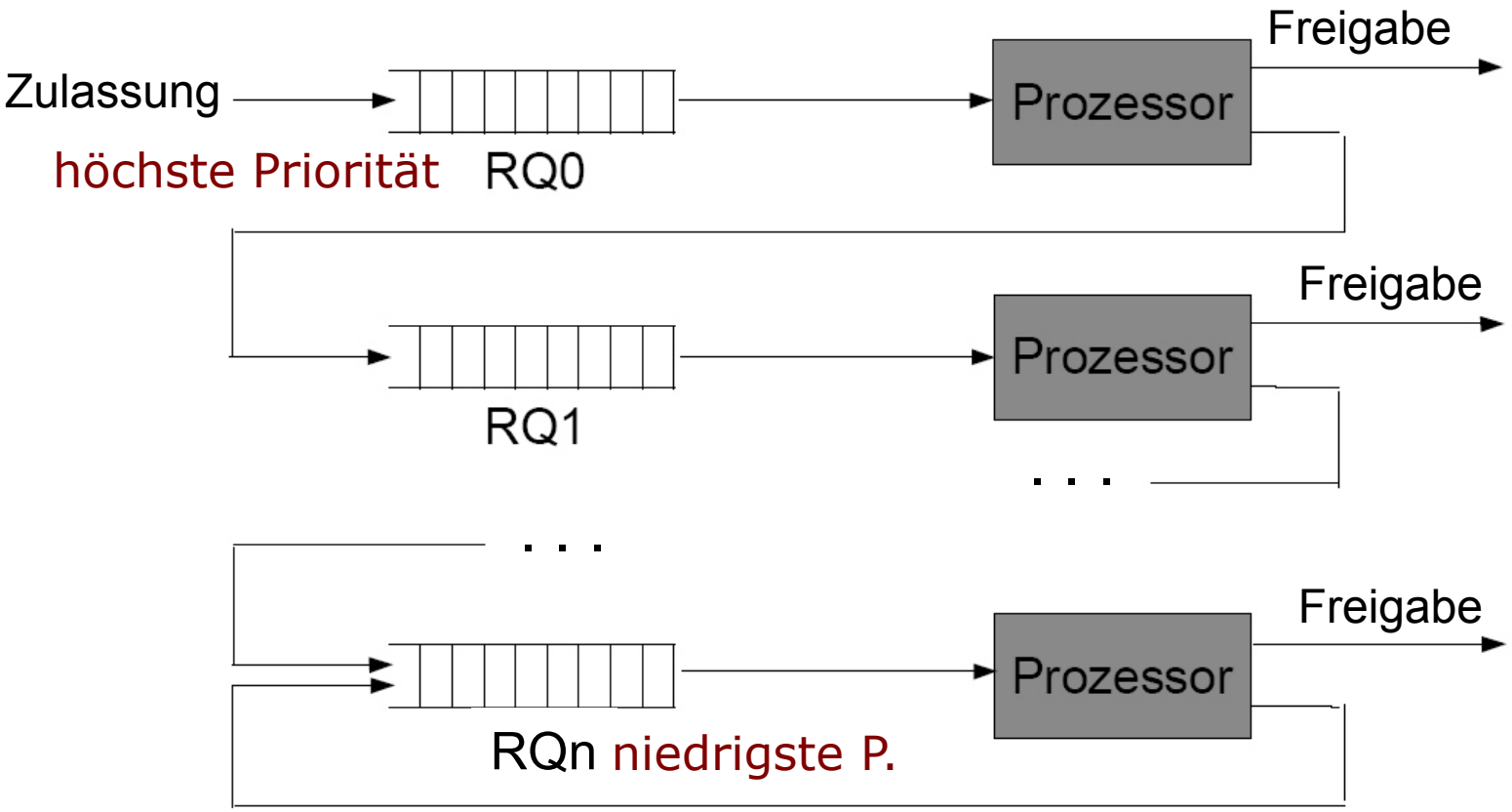
Feedback (1)

- Idee:

  Wenn wir die Gesamtlaufzeit von Prozessen nicht wissen, warum benutzen wir dann nicht einfach die bisher verbrauchte CPU-Zeit?

- Strategie:
  - Präemptiv (Zeitintervall), dynamische Prioritäten
  - Wenn ein Prozess die CPU abgeben muss, dann wird er in eine Warteschlange mit der nächst geringeren Priorität eingefügt
  - Dadurch: Verbrauchte CPU-Zeit wird angenähert durch Anzahl erzwungener CPU-Abgaben
  - Abarbeitung der Warteschlangen nach Priorität

Feedback (2)



- Innerhalb Warteschlangen: FCFS
- Bis auf letzte Warteschlange, dort RR

Feedback (3)

Analyse

- Bevorzugt E/A-lastige Prozesse
- Prozesse, die in der Vergangenheit viel CPU-Zeit verbraucht haben, werden bestraft
- Lange Jobs werden bestraft, können verhungern

Feedback (4)

- Variante 1:
  - Prozesse aus niedrigeren Warteschlangen erhalten längere Rechenzeiten, wenn sie drankommen, z.B. 2^i Zeiteinheiten für Prozesse aus Warteschlange RQ_i
  - Dadurch auch weniger Kontextwechsel
  - Längere Prozesse können immer noch verhungern
- Variante 2:
  - Neuberechnen der Prioritäten von Zeit zu Zeit
  - Wartezeit geht in die Priorität ein (UNIX)

„Traditionelles“ Unix-Scheduling

- Einsatz in interaktiven Timesharing-Umgebungen
- Ziele:
  - Gute Antwortzeiten für interaktive Benutzer
  - Gleichzeitig: Hintergrundaufträge mit geringer Priorität sollen nicht verhungern

Scheduling bei UNIX (1)

- Es gibt verschiedene Warteschlangen (wie bei Feedback) mit unterschiedlichen Prioritäten
- Ausgeführt wird anfangs der erste Prozess in der nichtleeren Warteschlange mit höchster Priorität
- Die Prozesse höchster Priorität werden anschließend untereinander nach Round Robin gescheduled
- Zeitintervall z.B. 100ms

Scheduling bei UNIX (2)

- Neuberechnung der Prioritäten in regelmäßigen Zeitabständen (z.B. jede Sekunde)
- priority = CPU_usage + nice + base
  (je kleiner der Wert, desto höher die Priorität)
- CPU_usage:
  - Maß für die CPU-Benutzung in der Vergangenheit
  - Bei z.B. Neuberechnung jede Sekunde:
    CPU_usage_{new} =
    ½ CPU_usage_{old} + CPU-Anteil in letzter Sekunde

Scheduling bei UNIX (3)

CPU_usage:

- a_1: Anteil in Sekunde 1, CPU_usage = a_1
- a_2: Anteil in Sekunde 2, CPU_usage = $0.5a_1 + a_2$
- a_3: Anteil in Sekunde 3, CPU_usage = $0.25a_1 + 0.5a_2 + a_3$
- a_4: Anteil in Sekunde 4, CPU_usage = $0.125a_1 + 0.25a_2 + 0.5a_3 + a_4$
- gewichtete Summe
- Gewicht der alten Werte nimmt exponentiell ab
- Alte Werte zählen mit der Zeit immer weniger

Scheduling bei UNIX (4)

- nice:
  - Durch den Benutzer kontrollierbarer Regulierungsfaktor
- base:
  - Durch System gewählter Basis-Prioritätswert
  - Einteilung in feste Prioritätsstufen
  - Dadurch effiziente Ausführung von E/A Operationen
  - Benutzerprozesse niedrigste Priorität
  - Bevorzugung von Prozessen, die durch Abschluss einer E/A-Operation wieder bereit wurden gegenüber CPU-lastigen Prozessen

Thread-Scheduling

- Prozesse können mehrere Threads besitzen
- Erinnerung: „leichtgewichtige“ Prozesse; gemeinsame Nutzung des Adressraumes
- Performanzgewinn z.B. bei rechenintensivem Teil und E/A
- Parallelität in 2 Ebenen: Prozesse / Threads
- Unterscheidung: Threads auf Benutzerebene / auf Systemebene

Threads auf Benutzerebene

- System weiß nicht Bescheid über die Existenz der Threads eines Benutzerprogramms
- Scheduling findet auf Prozessebene statt
- Thread-Scheduler entscheidet dann, welcher Thread von gewähltem Prozess laufen soll
- Thread wird nicht unterbrochen innerhalb CPU-Zeitintervall für Prozess
- Läuft, bis er warten muss oder fertig ist, oder bis das Zeitintervall abgelaufen ist und ein anderer Prozess vom Scheduler gewählt wird

Threads auf Systemebene

- Scheduling findet auf Thread-Ebene statt
- Zu welchem Prozess der Thread gehört, kann, aber muss nicht berücksichtigt werden
- Voller Prozesswechsel u.U. nötig, wenn Thread zu anderem Prozess gehört
- Dies kann das System bei Scheduling-Entscheidung berücksichtigen
- Beispiel: Zwei Threads gleichwichtig, einer gehört zum gleichen Prozess wie ein gerade blockierter Thread, gib diesem den Vorzug

Zusammenfassung

- Drei Arten von Scheduling (kurz-, mittel-, langfristig) existieren
- Es gibt eine Vielzahl von Kriterien (Benutzer-, Systemorientiert)
- Es gibt viele verschiedene Scheduling-Strategien für das kurzfristige Scheduling
- Wahl des Algorithmus hängt ab von der Anwendung
- Prioritäten und CPU-Nutzung sollten in Auswahlentscheidung mit eingehen

Hinweis

- Vorlesung Echtzeitbetriebssysteme und Zuverlässigkeit (Prof. Dr. Scholl) vermutlich im SS 2014
- Spezialvorlesung (Bachelor Informatik, Bachelor Embedded Systems Engineering, Master Informatik)