Systeme I: Betriebssysteme

Kapitel 7 Scheduling

Maren Bennewitz

Version 15.01.2014

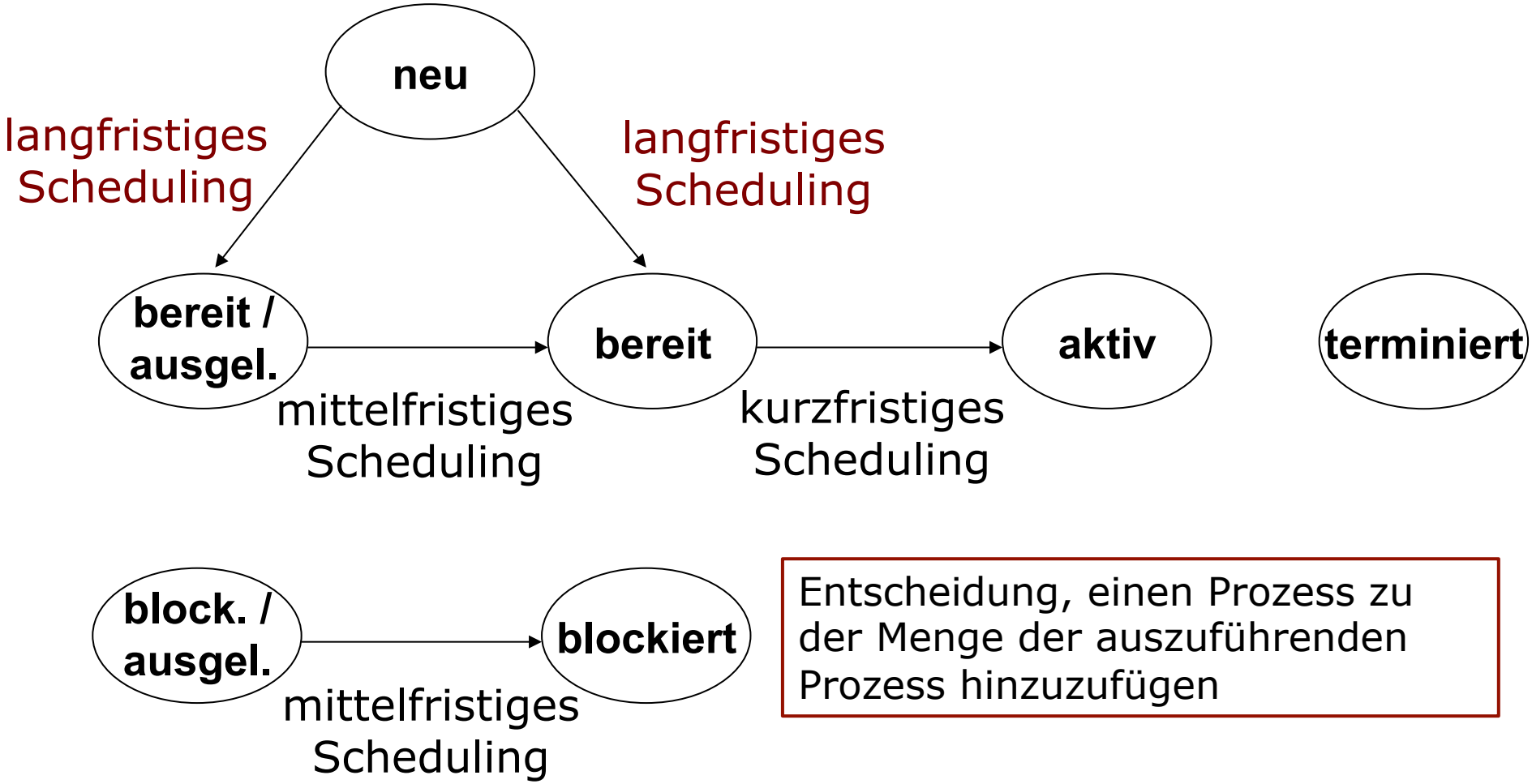
Inhalt Vorlesung

- 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

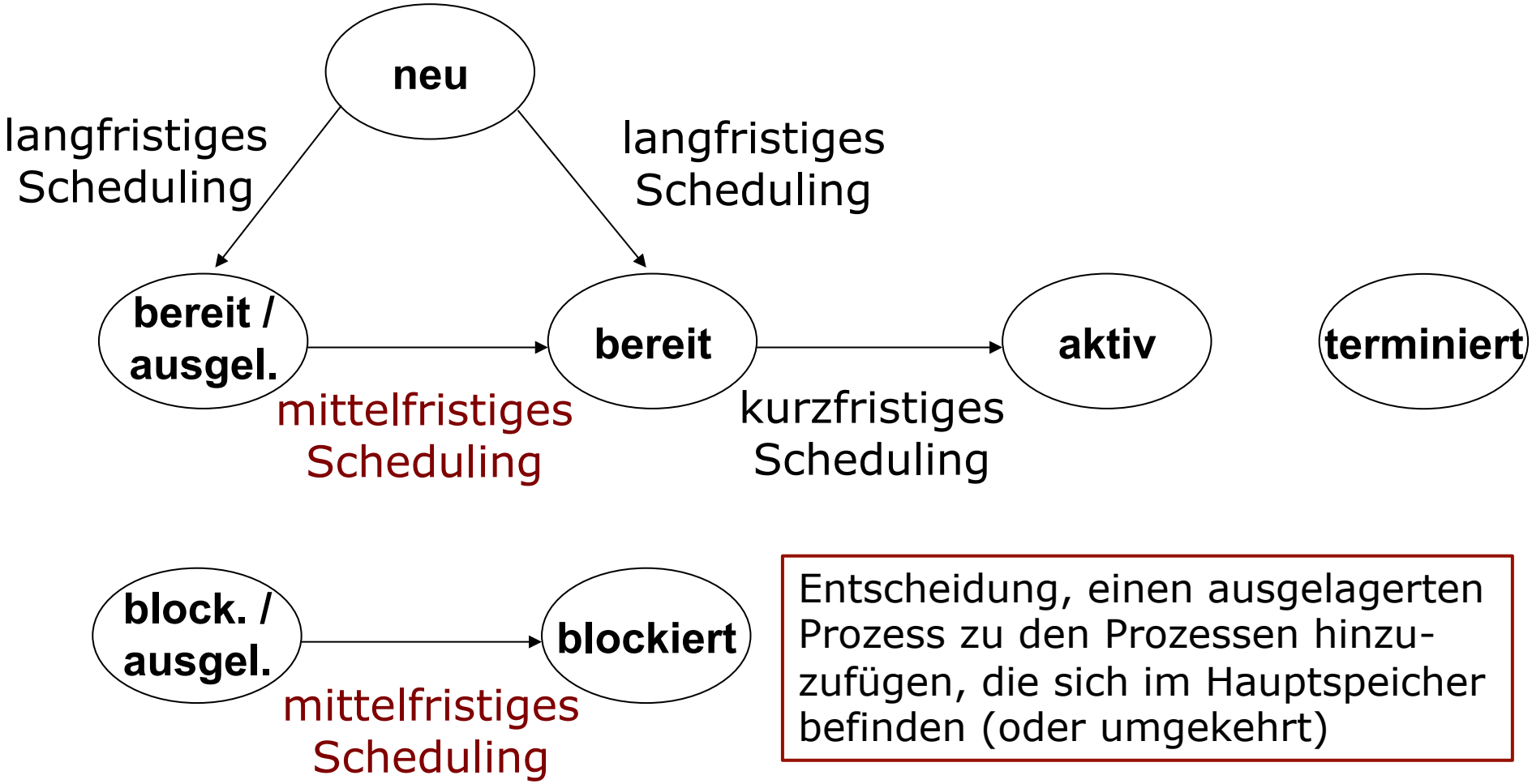
- Mehrprogrammsystem: Im Hauptspeicher werden mehrere Prozesse verwaltet
- Jeder Prozess wird entweder vom Prozessor bearbeitet oder wartet auf ein Ereignis
- Scheduling: Betriebssystem muss entscheiden, welche Prozesse auf den CPU-Kernen Rechenzeit beanspruchen dürfen

Drei Arten von Scheduling



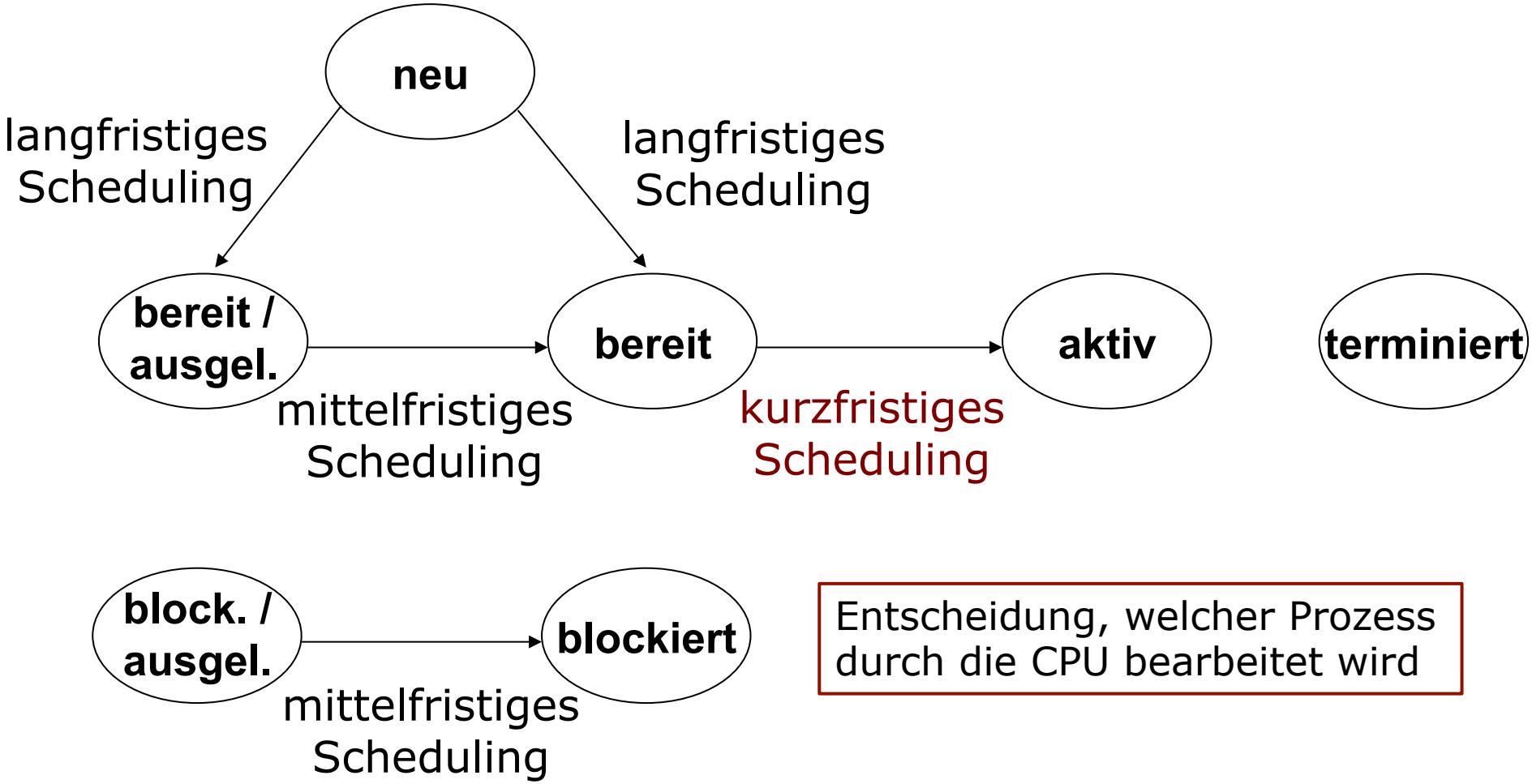
Entscheidung, einen Prozess zu der Menge der auszuführenden Prozess hinzuzufügen

Drei Arten von Scheduling



Entscheidung, einen ausgelagerten Prozess zu den Prozessen hinzu-zufügen, die sich im Hauptspeicher befinden (oder umgekehrt)

Drei Arten von Scheduling



Entscheidung, welcher Prozess durch die CPU bearbeitet wird

Kurzfristiges Scheduling

- Rechenzeit wird Prozessen so zugewiesen, dass “optimale Performance” erreicht wird
- Verschiedene Scheduling-Algorithmen existieren für verschiedene Optimierungsziele
- Beachte: Kontextwechsel kosten Rechenzeit

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 Anzahl von Prozessen, die pro Zeiteinheit abgearbeitet werden (Durchsatz, z.B. pro Stunde)
  - Maximale CPU-Auslastung (aktive Zeit)

Durchsatz vs. Durchlaufzeit

- Durchsatz: Anzahl der Prozesse, die vom System z.B. pro Stunde erledigt werden
- Durchlaufzeit: Zeit von Start bis Abschluss
- Hoher Durchsatz heißt nicht unbedingt niedrige Durchlaufzeit
- Für Benutzer ist eher niedrige Durchlaufzeit interessant

Kriterien für das kurzfristige Scheduling (2)

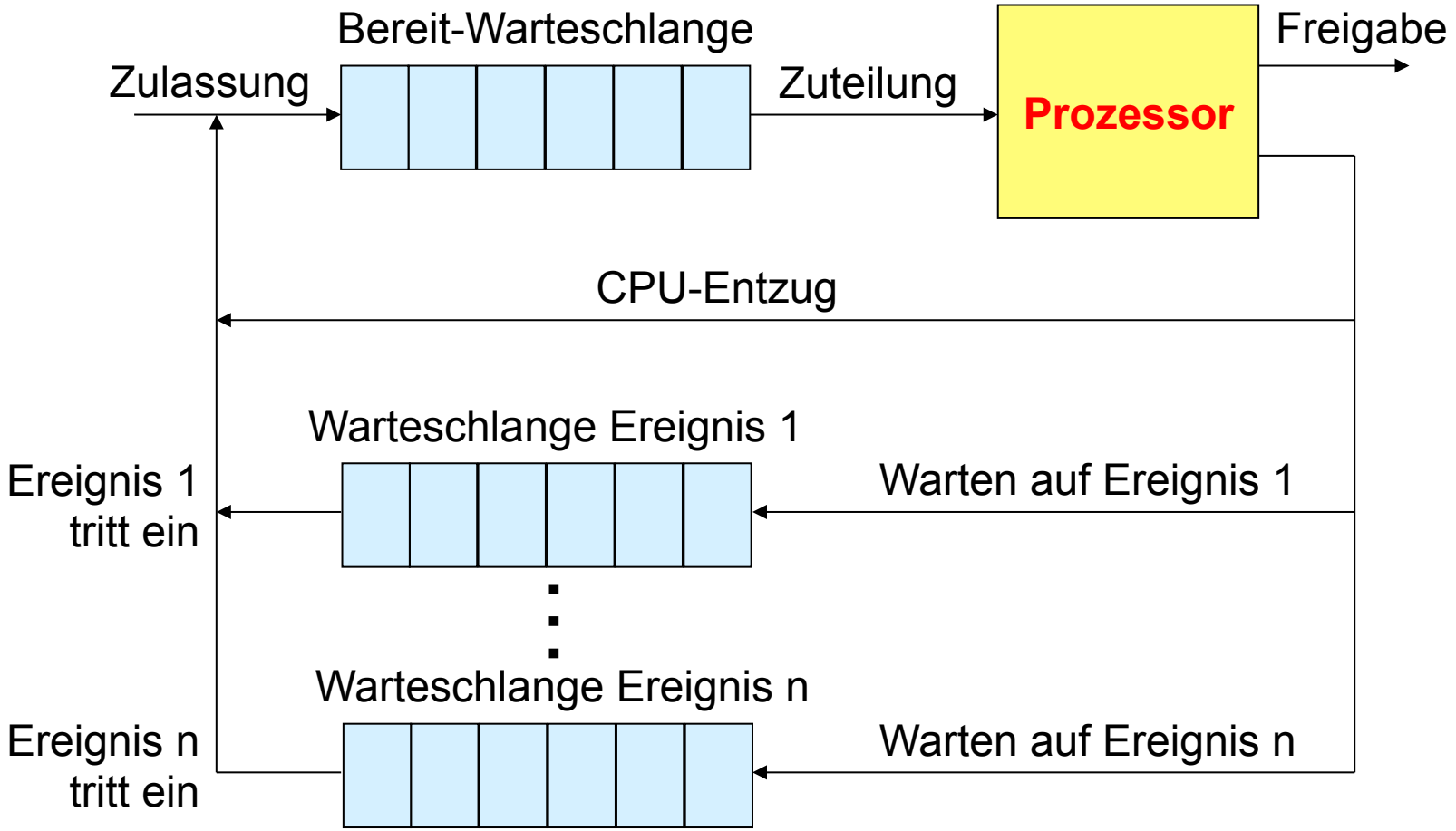
- Allgemein:
  - Fairness: Jeder Prozess erhält CPU irgendwann
  - Prioritäten müssen eingehalten werden
  - Effizienz: Möglichst wenig Aufwand für Scheduling selbst
- Echtzeitsysteme:
  - Vorhersehbares Verhalten
  - Einhalten von Deadlines

Kriterien für das kurzfristige Scheduling (3)

- Abhängigkeiten zwischen den Kriterien
- Beispiel:
  - Kurze Antwortzeit: Viele Wechsel zwischen Prozessen
  - Aber dann: Niedrigerer Durchsatz und mehr Aufwand durch Prozesswechsel
- Scheduling-Strategie muss Kompromiss schließen

Erinnerung: Warteschlangen

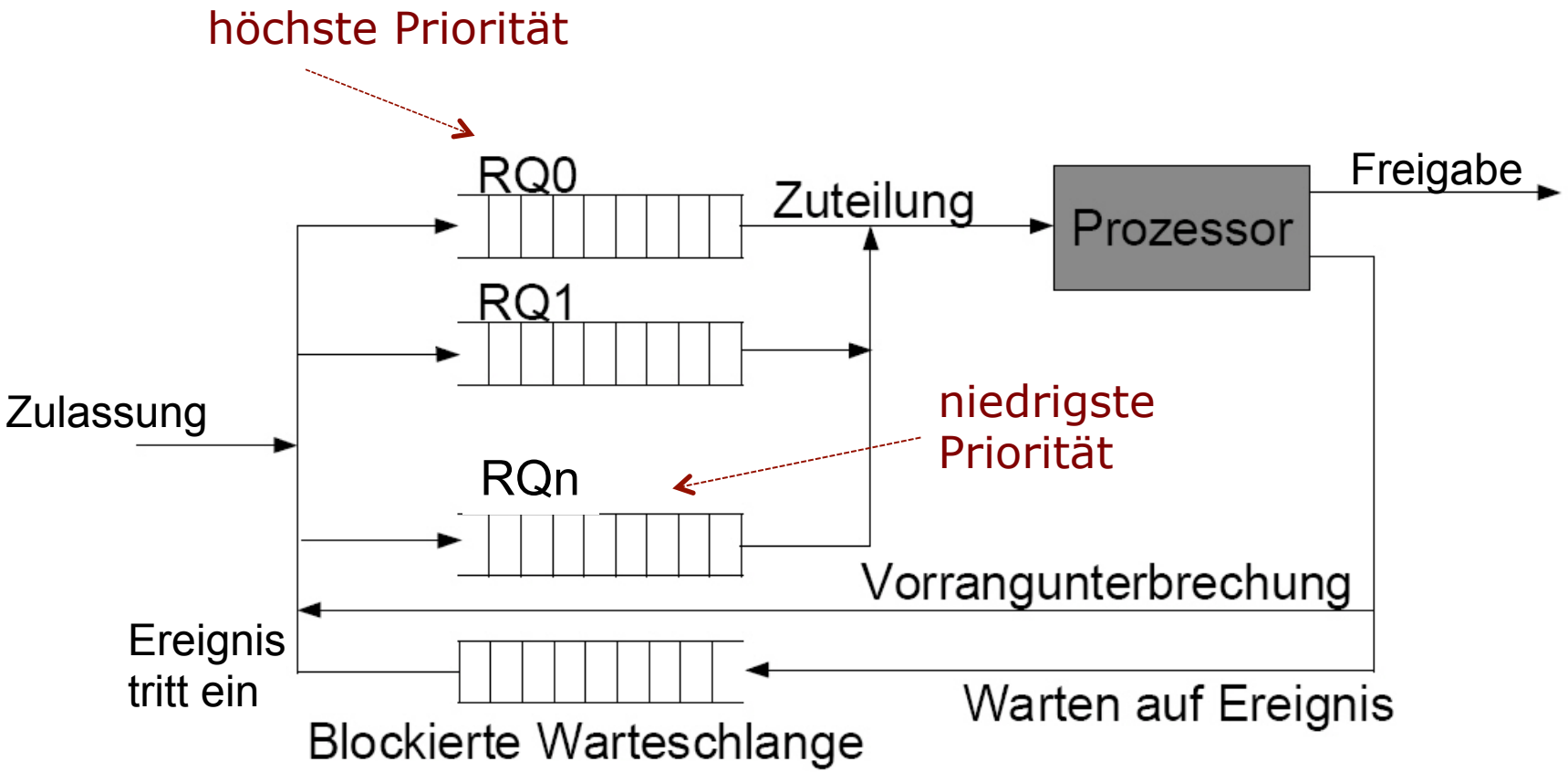
Warteschlangen für bereite Prozesse und für Prozesse, die auf Ereignisse warten



Prioritäten (1)

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

Prioritäten (2)



Prioritäten (3)

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

Scheduling-Algorithmen: Prozessauswahl

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

Scheduling-Algorithmen: Zeitpunkt der Auswahlentscheidung

- Nicht-präemptives Scheduling:

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

- Präemptives Scheduling:

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

First Come First Served (FCFS)

- Nicht-präemptive Strategie
- Strategie:
  Wenn ein Prozess beendet oder blockiert ist:
  Bereiter Prozess, der schon am längsten wartet, wird aktiv
- Auswahlfunktion: max(w)
- Implementiert durch einfache Warteschlange

First Come First Served (FCFS)

Analyse

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

Round Robin (RR)

- Präemptive Strategie
- Strategie:
  - Scheduler wird nach Ablauf fester Zeitdauer immer wieder aktiviert
  - Laufender Prozess wird dann in eine Warteschlange eingefügt
  - Der am längsten wartende Prozess wird aktiviert

Round Robin (RR)

Analyse

- Länge des Zeitintervalls ist essentiell
- Zu kurz: Aufwand für viele Prozesswechsel
- Zu lang: Ähnlich FCFS
- Sinnvoll: Entsprechend durchschnittlich benötigter CPU-Zeit
- Prozesse ohne Ein-/Ausgabe etwas begünstigt
- Prozesse mit E/A geben CPU vor Ablauf Zeitintervall ab und sind dann erst blockiert

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 Prozesse A, B, C, D
  - A braucht 8 min, B, C, D jeweils 4 min
  - Alle Prozesse 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, nicht SJF: 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 bereit
- 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:

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

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 $t_1 \leq t_2 \leq \ldots \leq t_n$
- Also: SJF führt zur geringsten mittleren Durchlaufzeit

Shortest Job First (SJF) (8)

Analyse

- Erzielt minimale durchschnittliche Durchlaufzeit, sofern alle Prozesse gleichzeitig verfügbar
- Kurze Prozesse bevorzugt
- Gefahr, dass längere Prozesse verhungern
- Abschätzungen der Gesamtlaufzeit von Prozessen müssen gegeben sein

Shortest Remaining Time (SRT)

- Präemptive Variante von SJF
- Auswahlfunktion: min(s-e)
- Strategie:
  - Prozess mit kürzester geschätzter Restlaufzeit erhält CPU
  - Keine Unterbrechung aktiver Prozesse nach bestimmtem Zeitintervall
  - Stattdessen: Auswertung der Restlaufzeiten nur, wenn ein anderer Prozess bereit wird

Shortest Remaining Time (SRT)

Analyse

- Benachteiligt lange Prozesse, auch Verhungern möglich (wie SJF)
- Aufwand für Prozesswechsel und Aufzeichnen von Ausführungszeiten
- Abschätzungen der Gesamtlaufzeit von Prozessen müssen gegeben 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 normalisierter Durchlaufzeit („Response Ratio“)
  - R = (w+s) / s
  - Bei Prozessstart: R = 1.0
  - Prozess mit höchstem R erhält Rechenzeit

Highest Response Ratio Next (HRRN)

Analyse

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

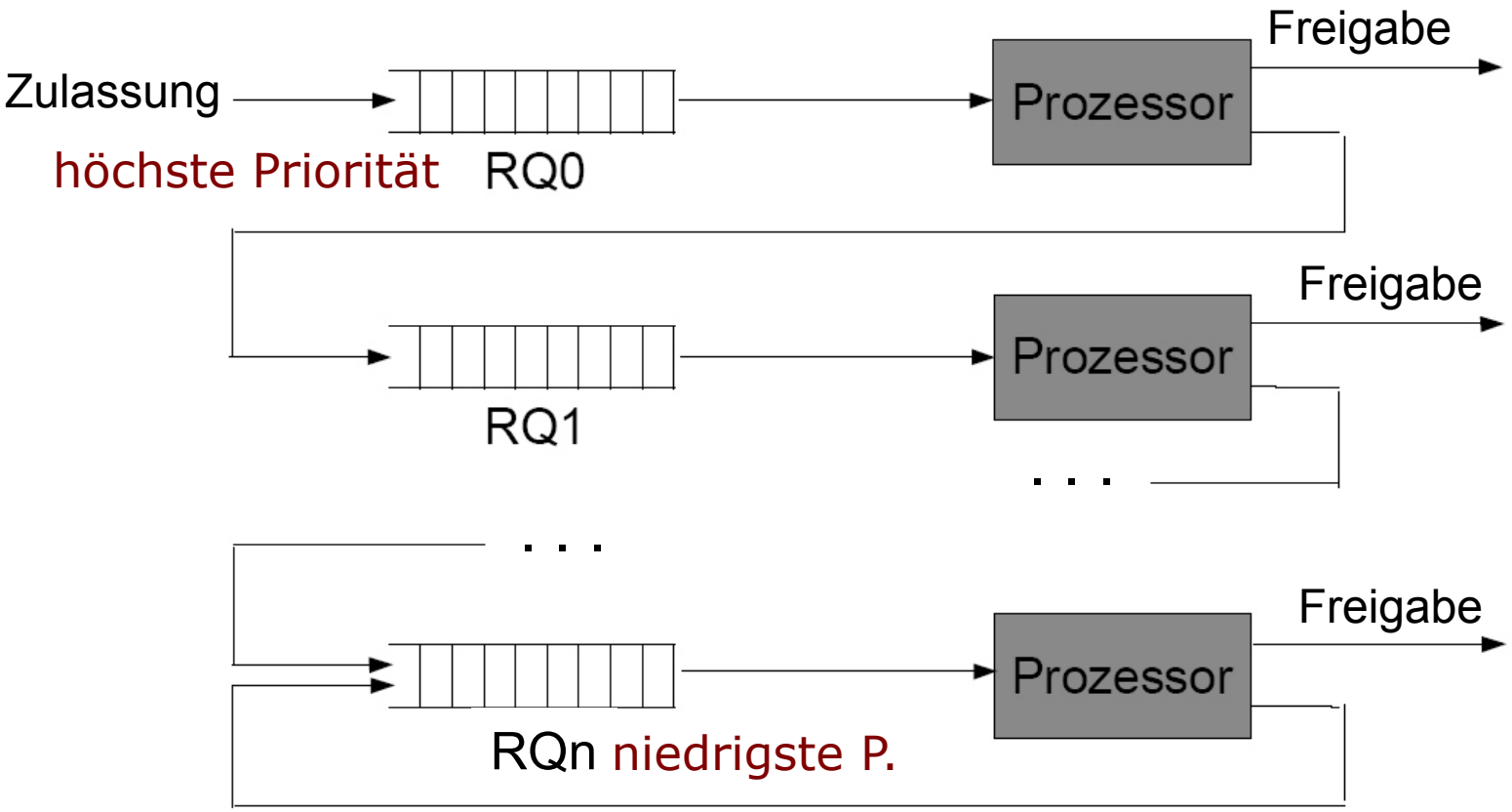
Feedback (1)

- Idee:

  Benutze statt Gesamtlaufzeit von Prozessen die bisher verbrauchte CPU-Zeit

- Strategie:
  - Präemptiv (Zeitintervall), dynamische Prioritäten
  - Bei Abgabe der CPU: Einreihen in Warteschlange mit der nächst geringeren Priorität
  - Abarbeitung der Warteschlangen nach Priorität
  - Dadurch: Verbrauchte CPU-Zeit wird angenähert durch Anzahl erzwungener CPU-Abgaben

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 Prozesse 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

- 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
- Anfangs: Erster Prozess der nichtleeren Warteschlange mit höchster Priorität ausgeführt
- Anschließend: Prozesse höchster Priorität werden untereinander nach Round Robin gescheduled

Scheduling bei UNIX (2)

- Neuberechnung der Prioritäten in regelmäßigen Zeitabständen
- priority = CPU_usage + nice + base
  (je kleiner der Wert, desto höher die Priorität)
- CPU_usage:
  - Maß für die CPU-Benutzung in der Vergangenheit
  - Bei 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

Scheduling bei UNIX (4)

- nice:
  - Durch den Benutzer kontrollierbarer Faktor, um mehr Rechenzeit für Prozesse zu fordern
- base:
  - Durch System gewählter Basis-Prioritätswert
  - Einteilung in feste Prioritätsgruppen
  - Höchste Priorität: Swapper
  - Niedrigste Priorität: Benutzerprozesse
  - Bei Benutzerprozessgruppe: Bevorzugung von Prozessen, die durch Abschluss einer E/A-Operation wieder bereit werden gegenüber CPU-lastigen Prozessen

Scheduling-Algorithmen Zusammenfassung (1)

- First Come First Served: Prozess, der bereits am längsten wartet, nicht präemptiv
- Round Robin: Aktive Prozesse werden nach bestimmter Zeit abgebrochen
- Shortest Job First: Prozess mit kürzester erwarteter Rechenzeit; keine Unterbrechung
- Shortest Remaining Time: Prozess mit kürzester geschätzter Restlaufzeit; Unterbrechung nur wenn anderer Prozess rechenbereit wird

Scheduling-Algorithmen Zusammenfassung (2)

- Highest Response Ratio Next: Prozess mit größter normalisierter Durchlaufzeit; nicht präemptiv
- Feedback: Warteschlangen von Prozessen, in die sie u.a. entsprechend ihrer Ausführungsgeschichte eingeteilt werden; Unterbrechung nach bestimmter Zeitdauer

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 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
- Voller Kontextwechsel u.U. nötig, wenn neuer aktiver Thread zu anderem Prozess gehört
- 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 Strategien für das kurzfristige Scheduling
- Wahl des Algorithmus‘ hängt ab von der Anwendung
- Prioritäten und bisherige Rechenzeit sollten in Auswahlentscheidung mit eingehen

Hinweis

- Vorlesung Echtzeitbetriebssysteme und Zuverlässigkeit (Prof. Dr. Scholl) vermutlich im SS 2014