3. Scheduling

- Überblick
  - 3.1 Einführung
  - 3.2 Schedulingstrategien
  - 3.3 Multilevel-Scheduling
  - 3.4 Scheduling mit Sollzeitpunkten
  - 3.5 Fallbeispiele (Linux und Windows)

3.1 Einführung

- Scheduling (Ablaufplanung)
  - Räumliche und zeitliche Zuordnung von Aktivitäten zu Instanzen, welche diese Aktivitäten durchführen können

- Scheduling wichtig für subjektive Wahrnehmung der Rechnerleistung
  - P1: Schließen eines Fensters, Aktualisieren der Oberfläche (2 s)
  - P2: Senden einer E-mail (2 s)
  - Reihenfolge P2P1: Benutzer nimmt die Verzögerung sofort wahr, empfindet den Rechner als langsam
  - Reihenfolge P1P2: Verzögerung des E-mail-Versands um 2 Sekunden bleibt wahrscheinlich unbemerkt

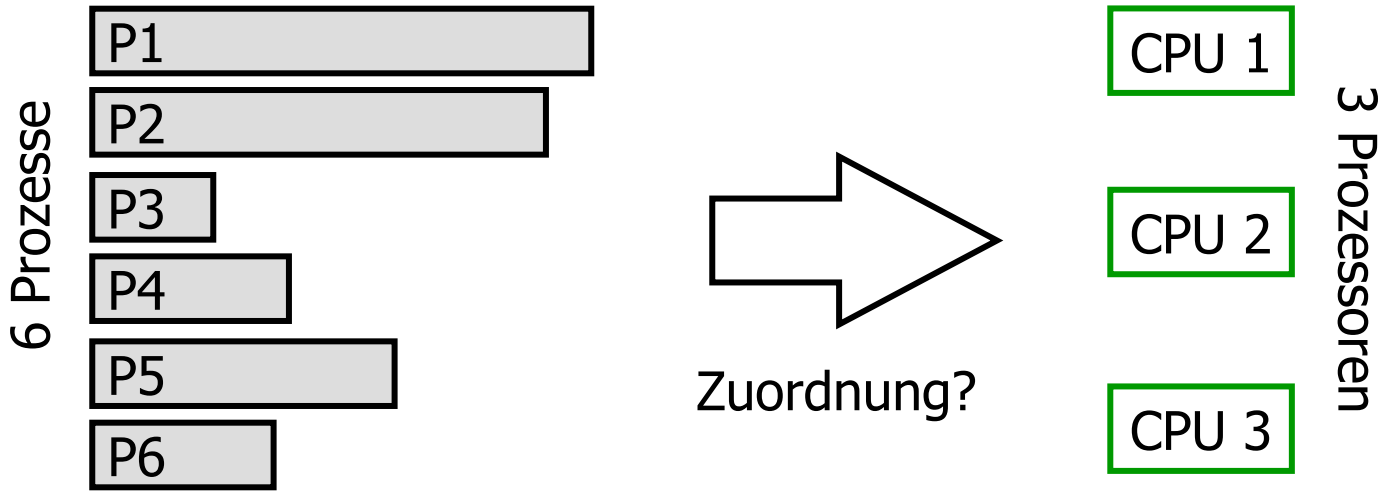
Zeitskalen

- Scheduling historisch in drei Bereiche eingeteilt:
  - Long-Term Scheduling (LTS): behandelt die Erzeugung (ausreichend Ressourcen vorhanden?) und Zerstörung von Prozessen
  - Medium-Term Scheduling (MTS): behandelt das Ein- und Auslagern von Prozessen
  - Short-Term Scheduling (STS): behandelt das Zuweisen der CPU(s) zu bereiten Prozessen
  - (und Dispatching: Durchführen der CPU-Zuweisung)

- Der Begriff „Scheduling“ meint gewöhnlich „CPU Scheduling“, also STS
  - Gesucht: einfache und effizient ausführbare Strategie zur Zuordnung von Prozessoren zu Prozessen
  - Fokus dieses VL-Kapitels

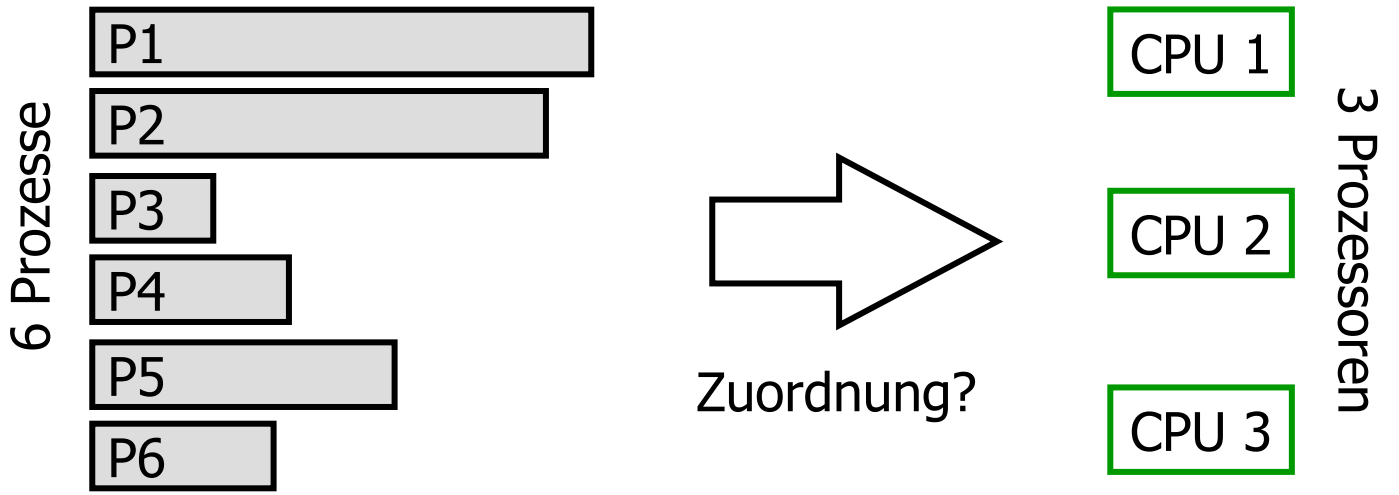
Klassisches Scheduling-Problem

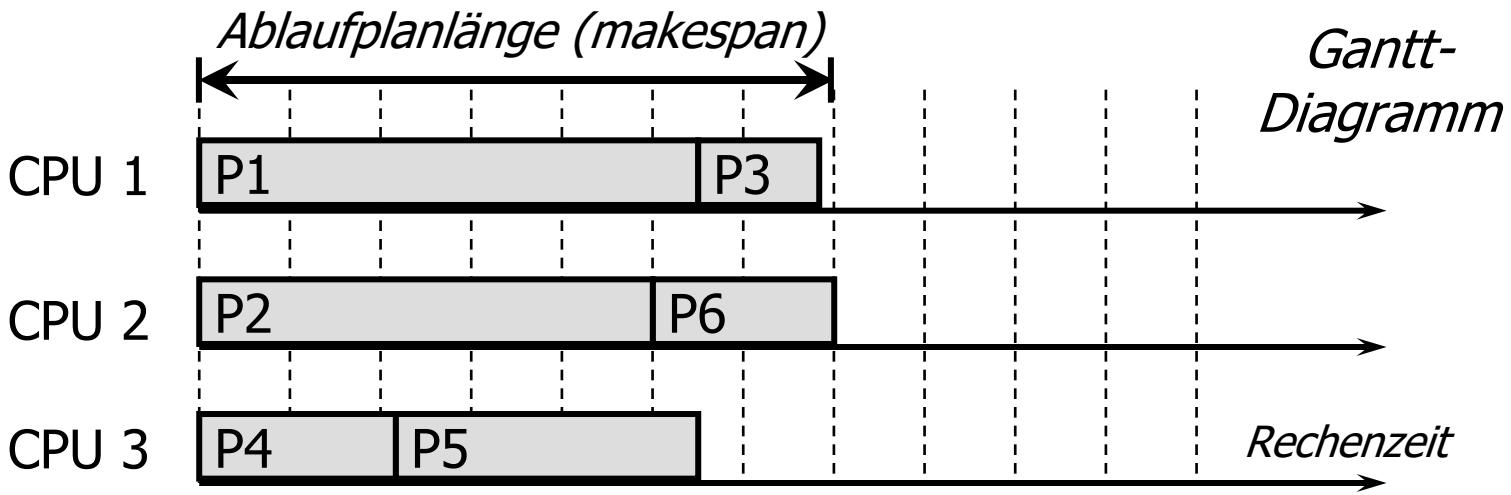
- Zuordnung von Prozessen zu Prozessoren so, dass die Ausführungszeit minimiert wird



Klassisches Scheduling-Problem

- Zuordnung von Prozessen zu Prozessoren so, dass die Ausführungszeit minimiert wird





Gestaltungsparameter für Scheduling

- Prozessmenge statisch oder dynamisch?
  - Statisch: Keine weiteren Prozesse kommen hinzu
  - Dynamisch: neue Prozesse kommen während der Ausführung hinzu
- On-line oder Off-line Scheduling
  - Off-line (irgendwann vorher): Alle Prozesse – auch zukünftige Ankünfte – sind bekannt → vollständige Information liegt vor
  - On-line (zur Laufzeit): Lediglich aktuelle Prozesse bekannt → Entscheidungsfindung auf Grund unvollständiger Information
- Plattform: Einkern- / Mehrkernprozessoren
- Verdrängung („Preemption“) möglich?
  - Scheduling-Entscheidung bei Änderung der Bereit-Liste
  - Mit Verdrängung können Schedulingziele besser erreicht werden

Gestaltungsparameter für Scheduling (2)

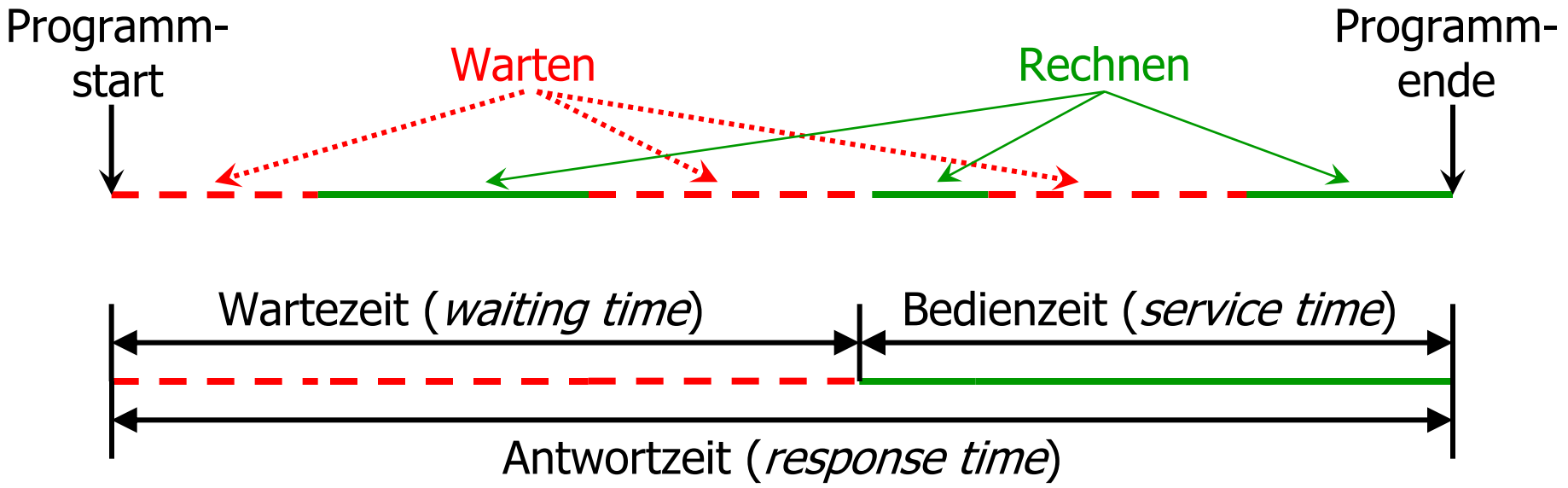
- Abhängigkeiten zwischen Prozessen vorhanden?
  - Reihenfolgebeziehung (partielle Ordnung)
  - Synchronisierte Zuordnung der parallelen Prozesse eines Programms
  - Kommunikationszeiten oder Umschaltzeiten zu berücksichtigen?
- Sind Prioritäten zu berücksichtigen?
  - Statisch (a-priori (von außen) vorgegeben) oder dynamisch (während der Ausführung)
- Sollzeitpunkte zu berücksichtigen? (oft bei Echtzeitsystemen)
  - Prozesse müssen zu bestimmten Zeitpunkten beendet werden
- Existieren periodische, regelmäßig wiederkehrende Prozesse?

Gestaltungsparameter für Scheduling (3)

- Zu erreichendes Ziel, d.h. zu optimierende Zielfunktion, wie z.B.
  - Länge des Ablaufplans (min)
  - Maximale Antwortzeit (min)
  - Mittlere (gewichtete) Antwortzeit (min)
  - Anzahl Prozessoren (min)
  - Durchsatz (max)
  - Prozessorauslastung (max)
  - Maximale Verspätung (min)
- Ziele für alle Systemarten
  - Fairness: gerechte Verteilung der Rechenzeiten an Bewerber
  - Policy Enforcement: Transparente Entscheidungskriterien
  - Balance: Alle Teile des Systems sind ausgelastet

3.2 Schedulingstrategien

- Zu erreichende Ziele
  - Hohe Effizienz → Hohe Auslastung des Prozessors
  - Geringe Antwortzeit bei interaktiven Prozessen
  - Fairness: gerechte Verteilung der CPU/GPU-Leistung und Wartezeit unter den Prozessen
- Definition



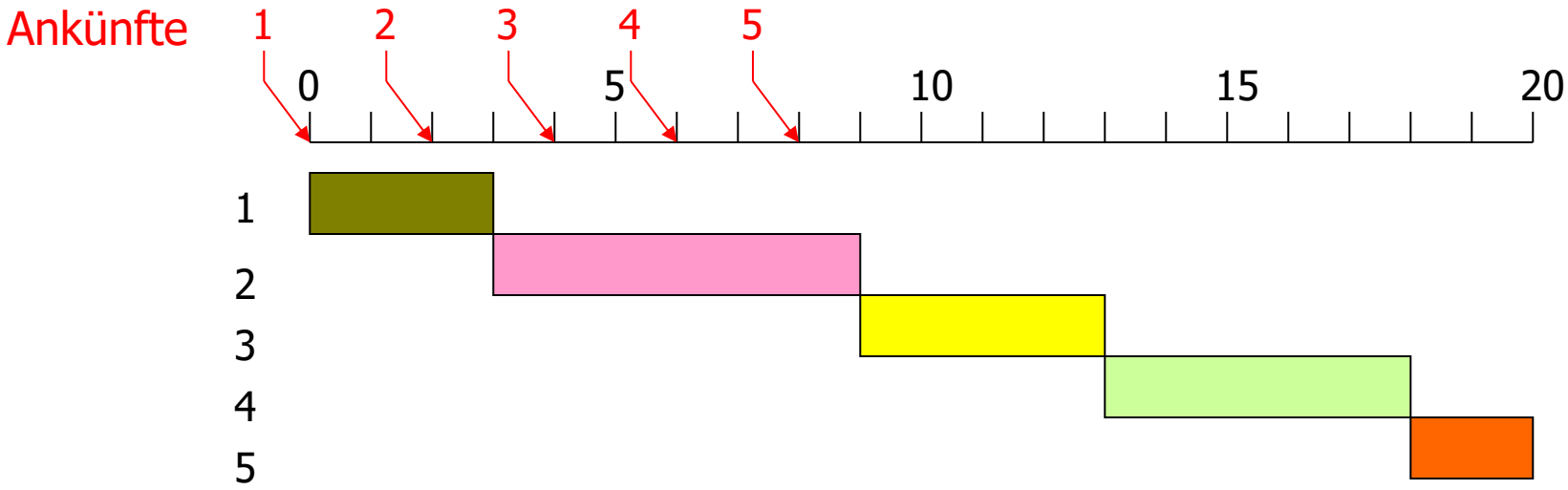
Beispieleingaben für Strategien

Gegeben seien die folgenden fünf Prozesse:

Nr.	Ankunft	Bedienzeit	Priorität
1	0	3	2
2	2	6	4
3	4	4	1
4	6	5	5
5	8	2	3

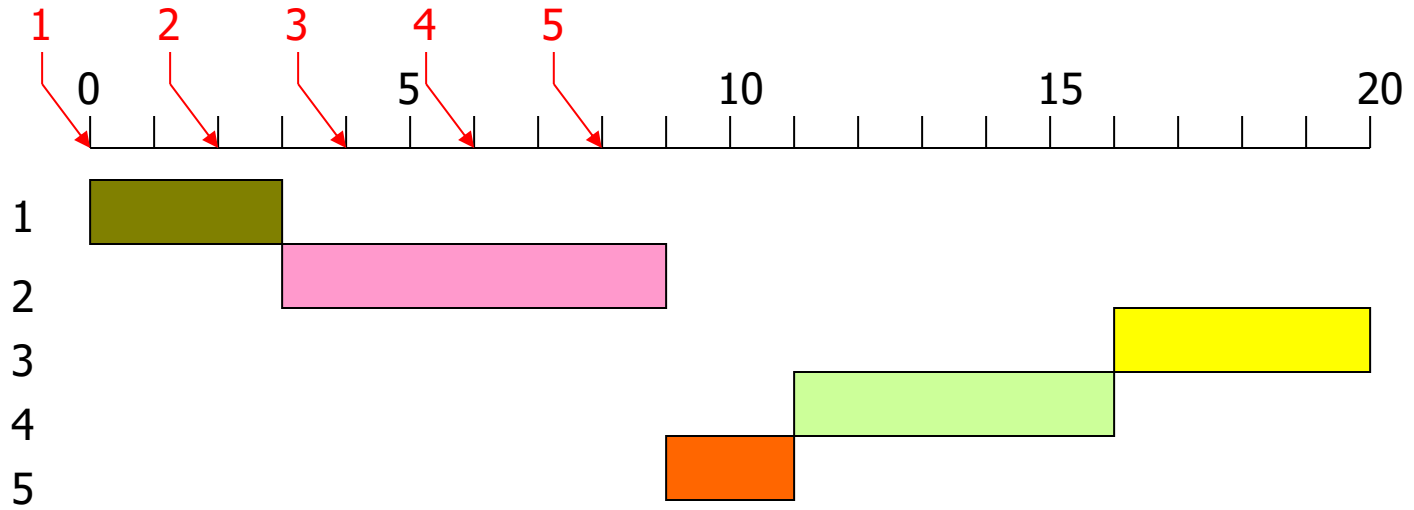
FCFS (First Come First Served) oder FIFO (First In First Out)

- Arbeitsweise
  - Bearbeitung der Prozesse in der Reihenfolge ihrer Ankunft in der Bereitliste
  - Prozessorbesitz bis zum Ende oder zur freiwilligen Aufgabe
  - Entspricht der Alltagserfahrung



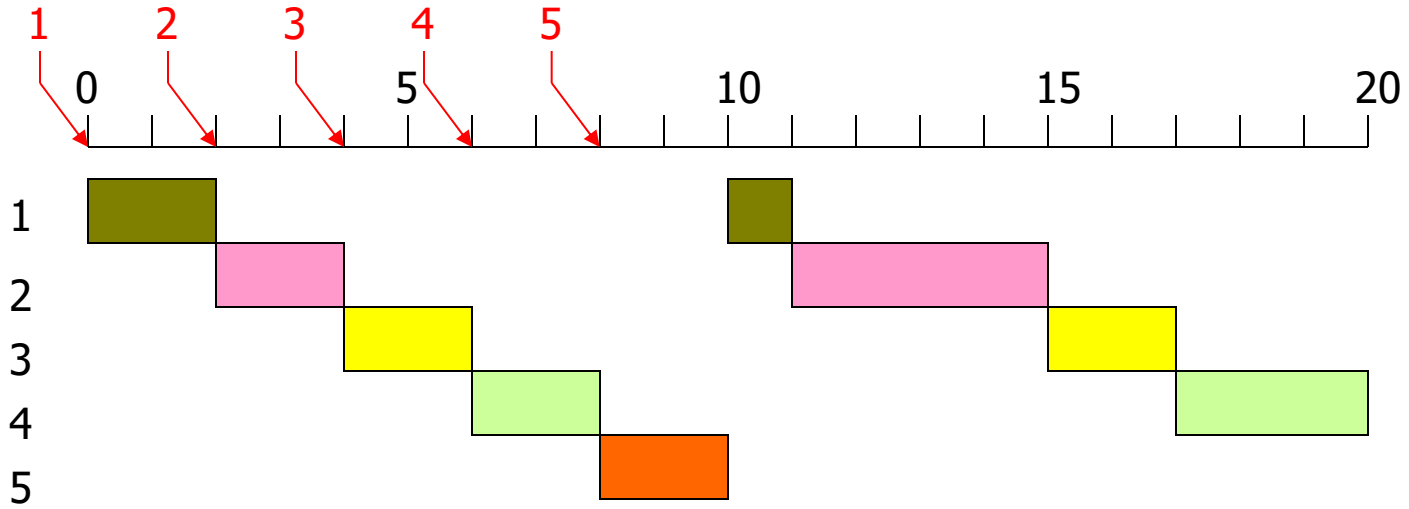
LCFS (Last Come First Served)

- Arbeitsweise
  - Bearbeitung der Prozesse in der umgekehrten Reihenfolge ihrer Ankunft in der Bereitliste
  - Prozessorbesitz bis zum Ende oder zur freiwilligen Aufgabe
  - In dieser reinen Form selten benutzt



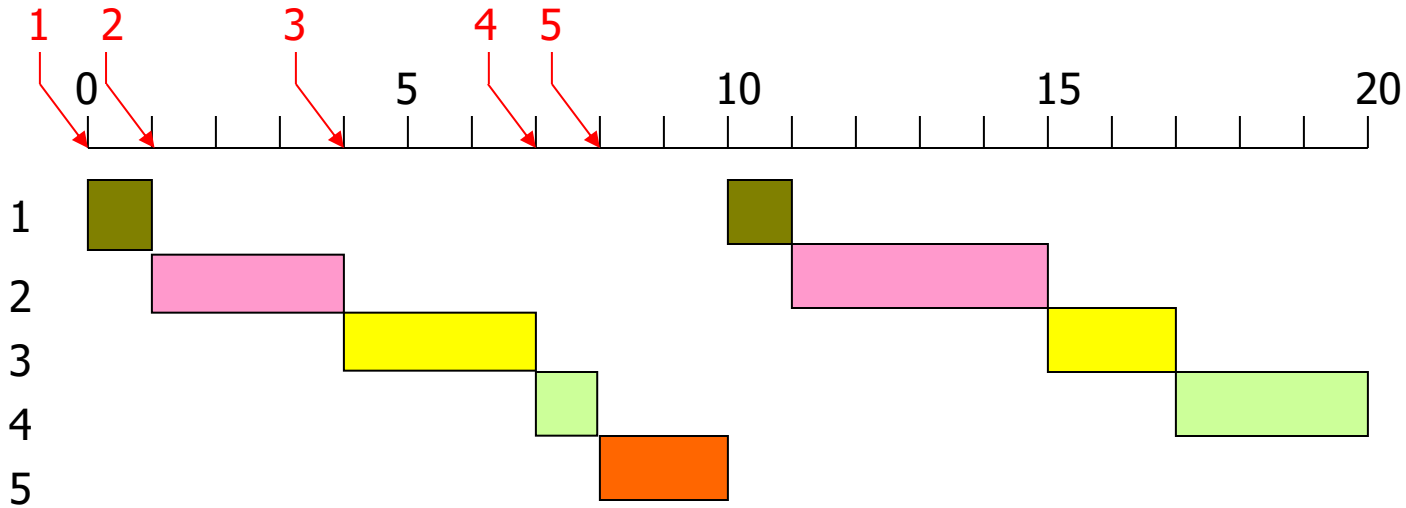
LCFS-PR (Last Come First Served - Preemptive Resume)

- Neuankömmling in Bereitliste verdrängt den rechnenden Prozess, verdrängter Prozess wird hinten in die Warteschlange eingereiht
- Ziel ist die Bevorzugung kurzer Prozesse.
  - Kurzer Prozess hat die Chance, schnell (vor der nächsten Ankunft) fertig zu werden
  - Ein langer Prozess wird u.U. mehrfach verdrängt
- Nach dem Eintreffen aller Prozesse und terminieren des letzten Prozesses wird die Warteschlange nach FIFO abgearbeitet



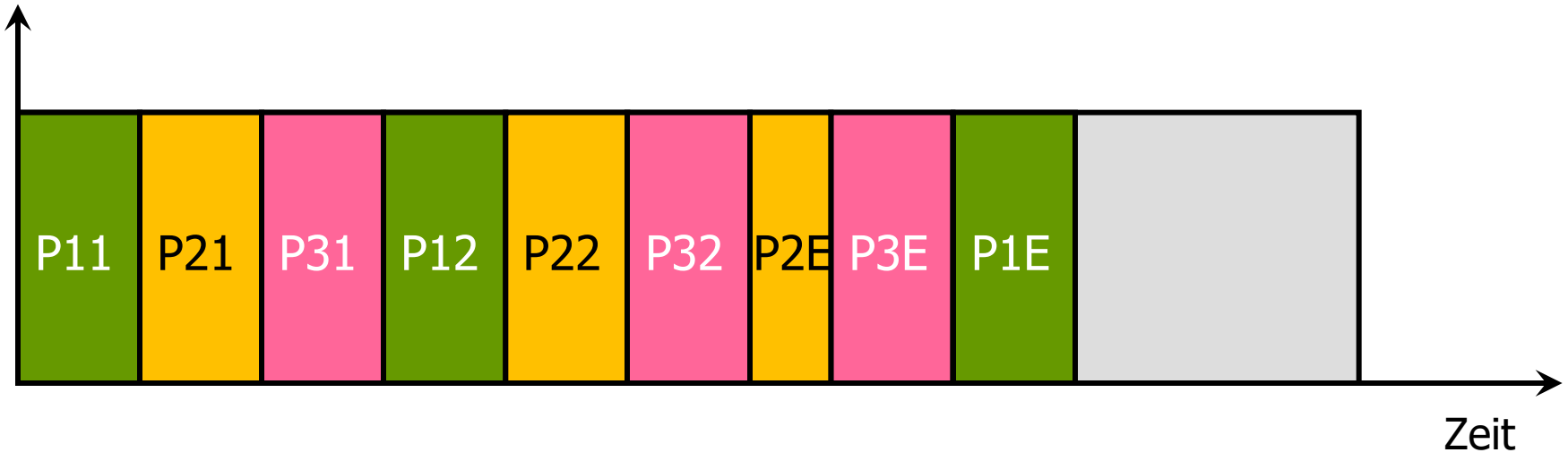
LCFS-PR (Last Come First Served - Preemptive Resume)

- Besseres Beispiel mit geänderten Ankunftszeiten und Prozesslängen



Zeitscheibenbetrieb (Time Sharing Mode)

- Abwicklung taktgesteuert und nebenläufig
  - Jeder Prozess erhält im festen Takt ein Zeitfenster zugeteilt
  - Ist er am Ende des Zeitfensters nicht fertig, dann wird er unterbrochen und in einer Warteschlange hinten angestellt
  - Änderung der Taktung durch Prioritäten, Warten auf Fertigstellung eines E/A-Auftrags, …
- Voraussetzung: Prozesse unterbrechbar, Prozessumschaltung möglich



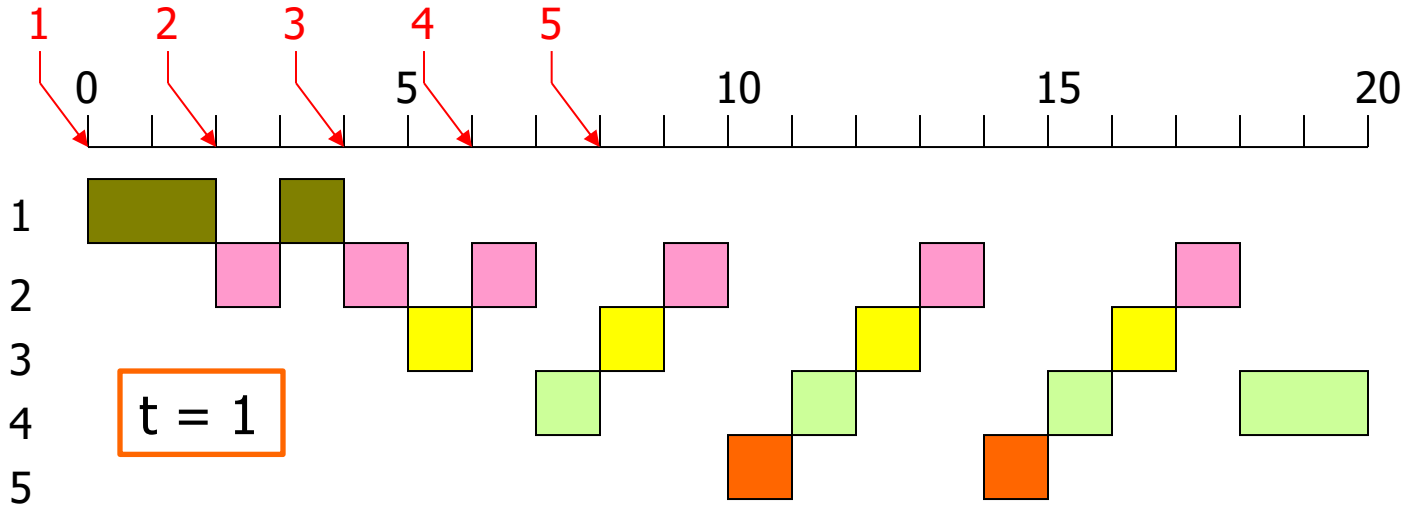
RR (Round Robin): Zeitscheibenlänge

- Ziel des Verfahrens ist die gleichmäßige Verteilung der Prozessorkapazität und der Wartezeit auf die Prozesse

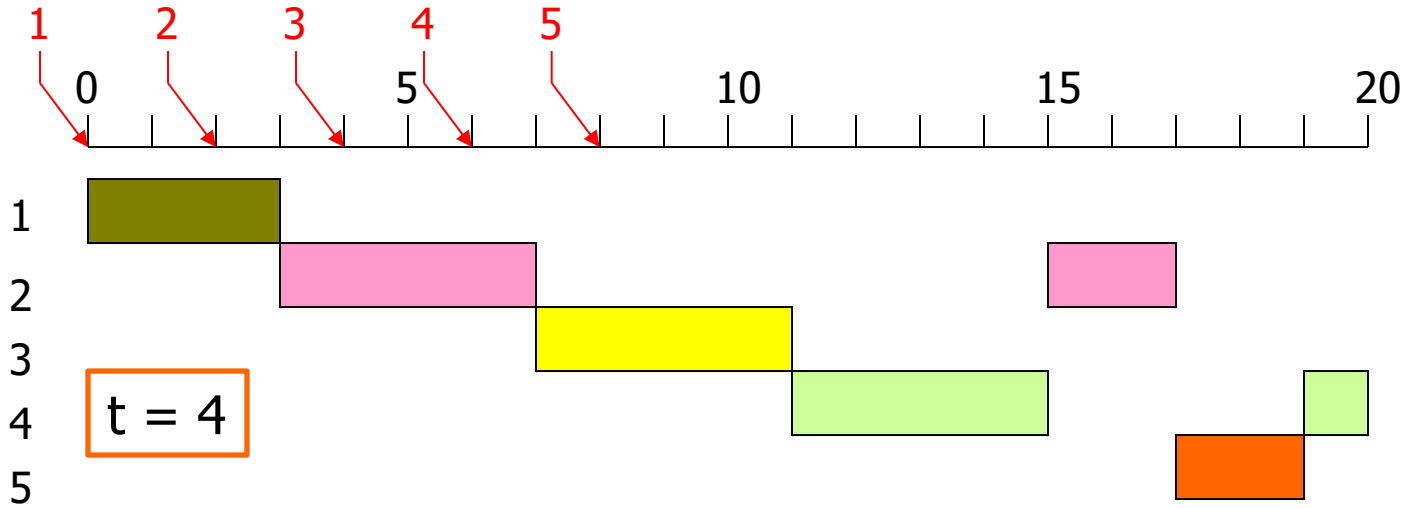
- Wahl der Zeitscheibenlänge t ist Optimierungsproblem
  - Für großes t nähert sich RR der Reihenfolgestrategie FCFS
  - Für kleines t schlägt der Aufwand für das häufige Umschalten negativ zu Buche

- Üblich sind Zeiten im Millisekunden-Bereich

RR (Round Robin) mit unter-schiedlichen Zeitscheiben

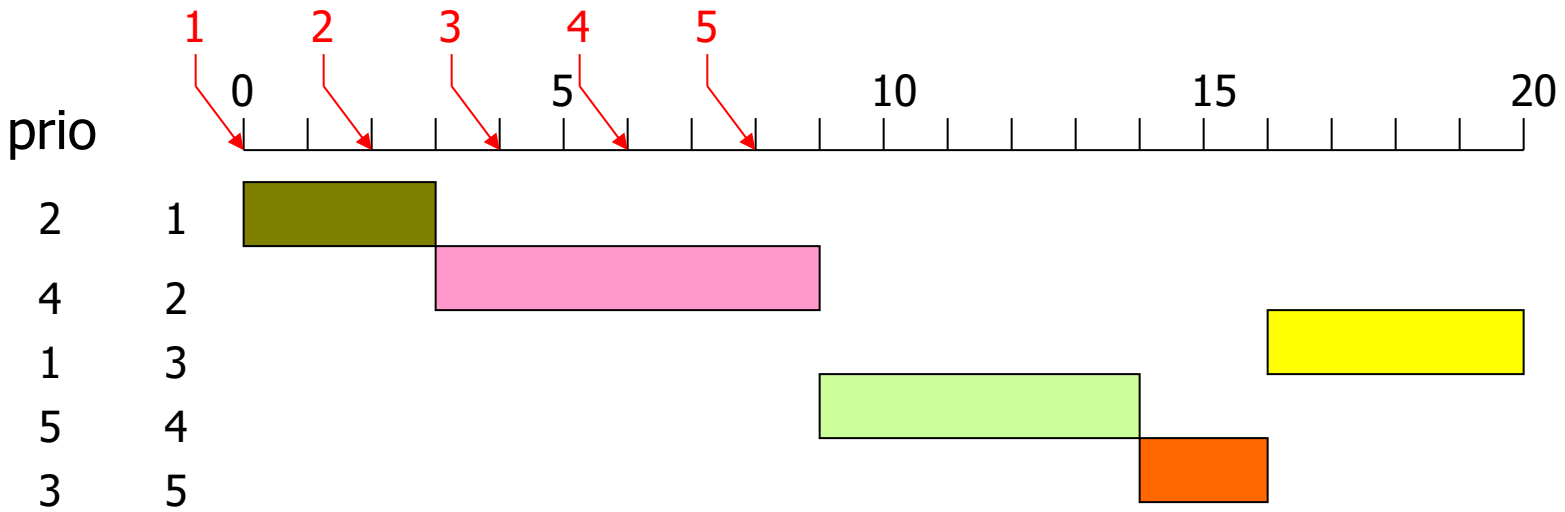


Reaktionsgeschwindigkeit vs. Overhead



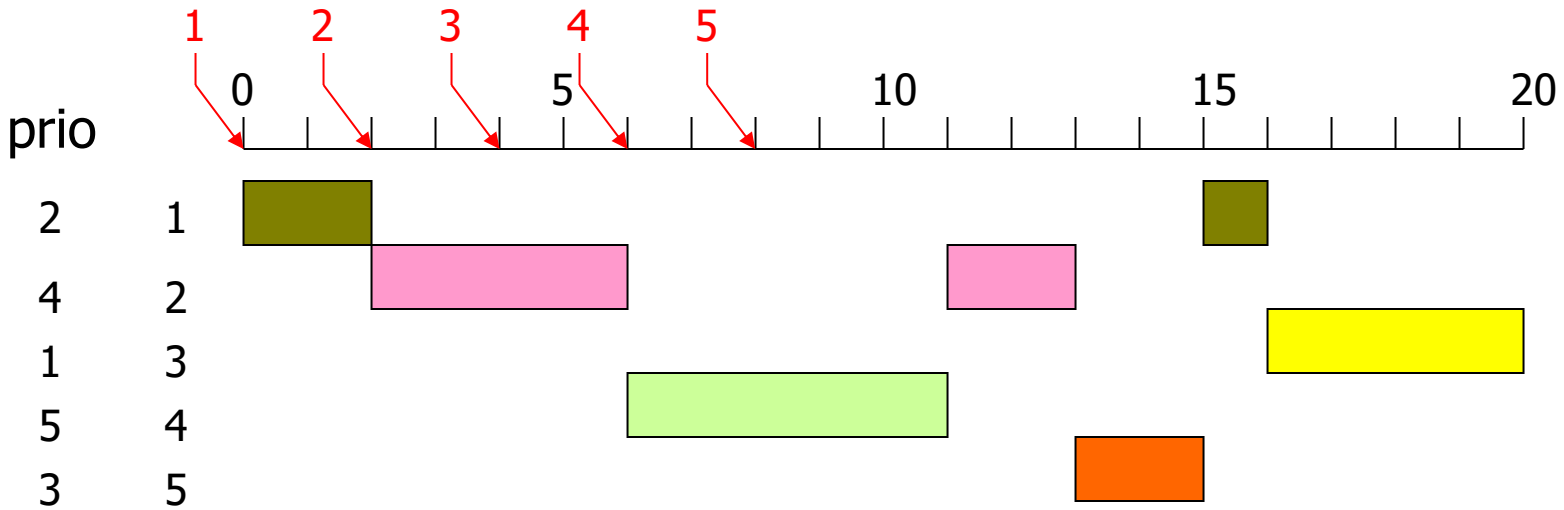
PRIO-NP
(Priorities – Non-preemptive)

- Neuankömmlinge werden nach ihrer Priorität in die Bereitliste eingeordnet
- Prozessorbesitz bis zum Ende oder zur freiwilligen Aufgabe



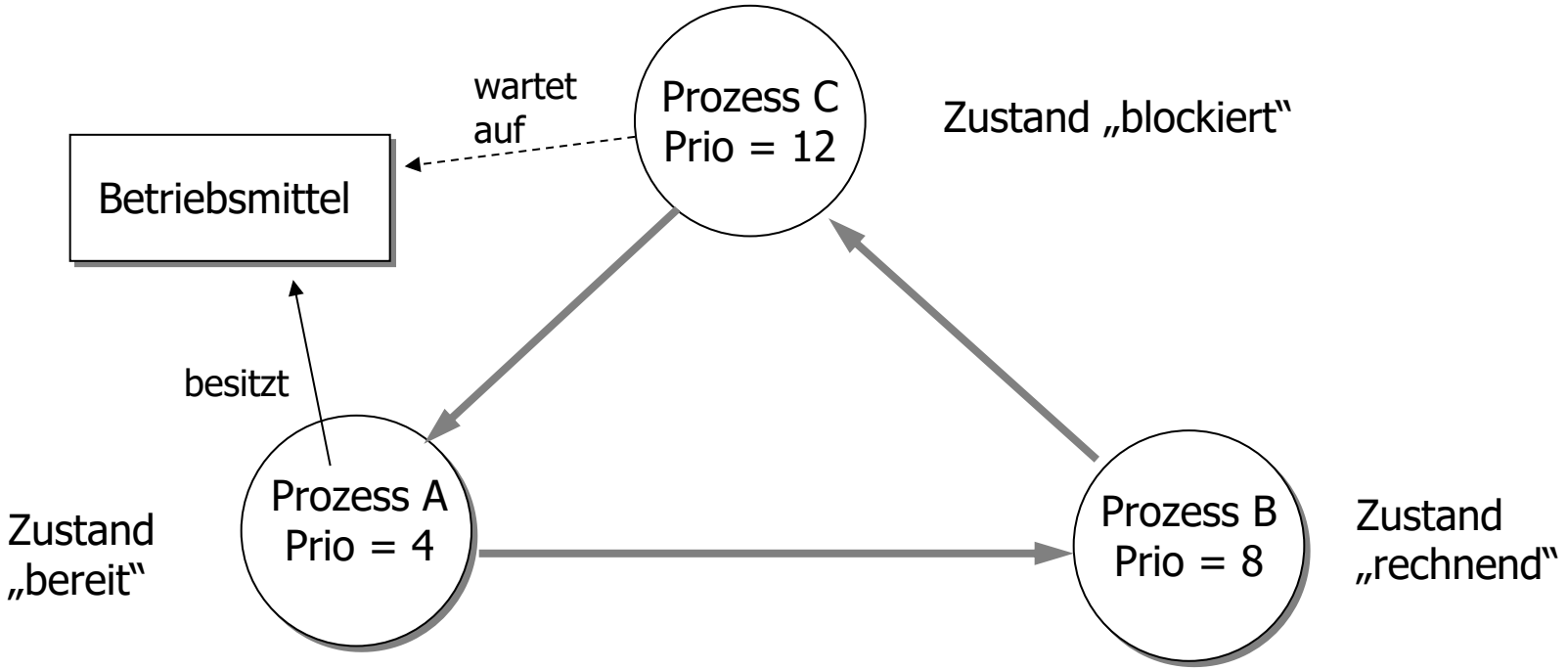
PRIO-P
(Priorities – Preemptive)

- Wie PRIO-NP, jedoch findet Verdrängungsprüfung statt
- Der rechnende Prozess wird verdrängt, wenn er eine geringere Priorität hat als der Neuankömmling



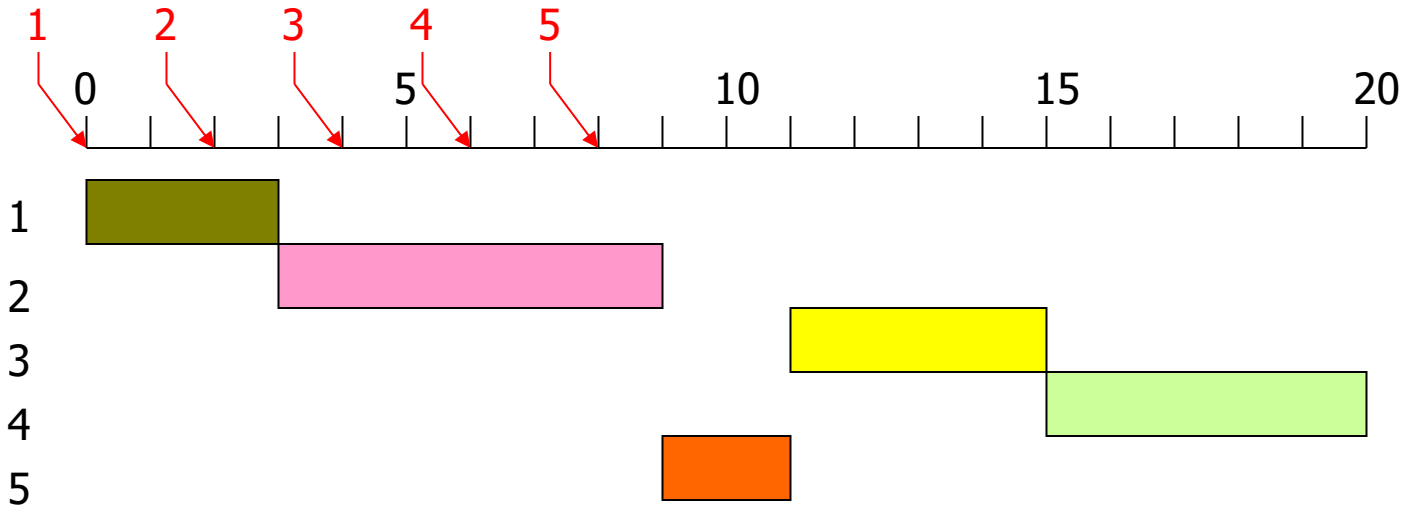
Prioritätsinvertierung

- Prozess C wird „verhungern", obwohl er der dringlichste ist:
  - B besetzt dauerhaft die CPU (als höchst-priorisierter lauffähiger Prozess), A kann das Betriebsmittel deshalb nicht freigeben, C bleibt auf unbestimmte Zeit blockiert
- Lösung: „Prioritätsvererbung" (priority inheritance): A bekommt Priorität von C, solange A das Betriebsmittel besitzt, auf das C wartet



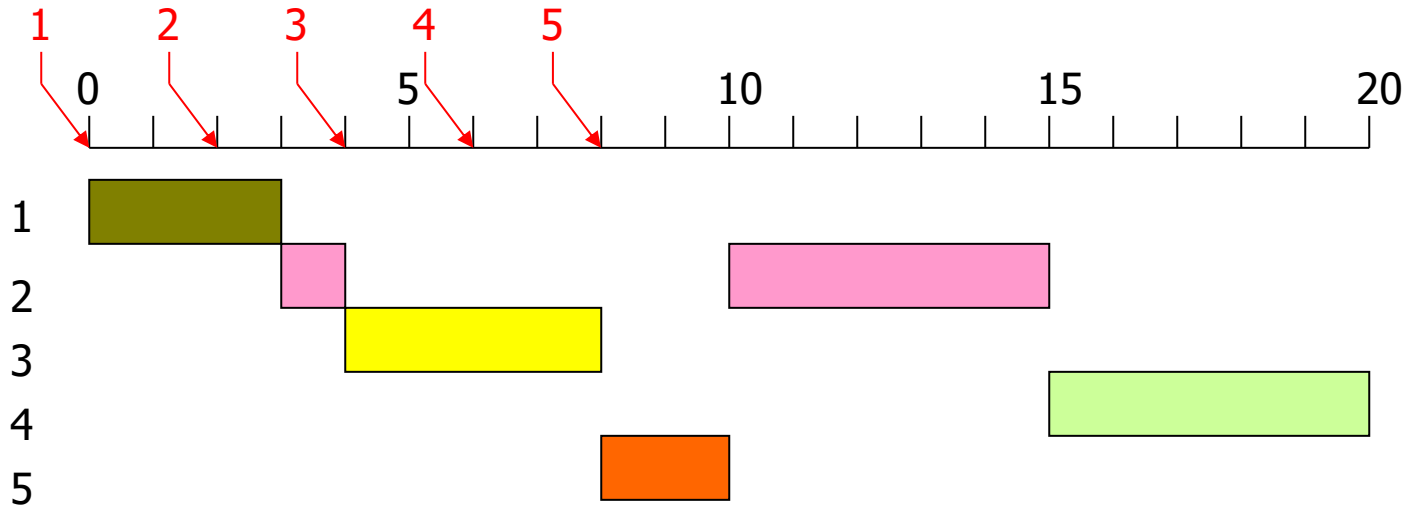
SJN (Shortest Job Next)

- Prozess mit kürzester Bedienzeit wird als nächster bis zum Ende oder zur freiwilligen Aufgabe bearbeitet
  - Wie PRIO-NP mit Bedienzeit als umgekehrtem Prioritätskriterium, kurze Bedienzeit → hohe Priorität
  - Bevorzugt kurze Prozesse



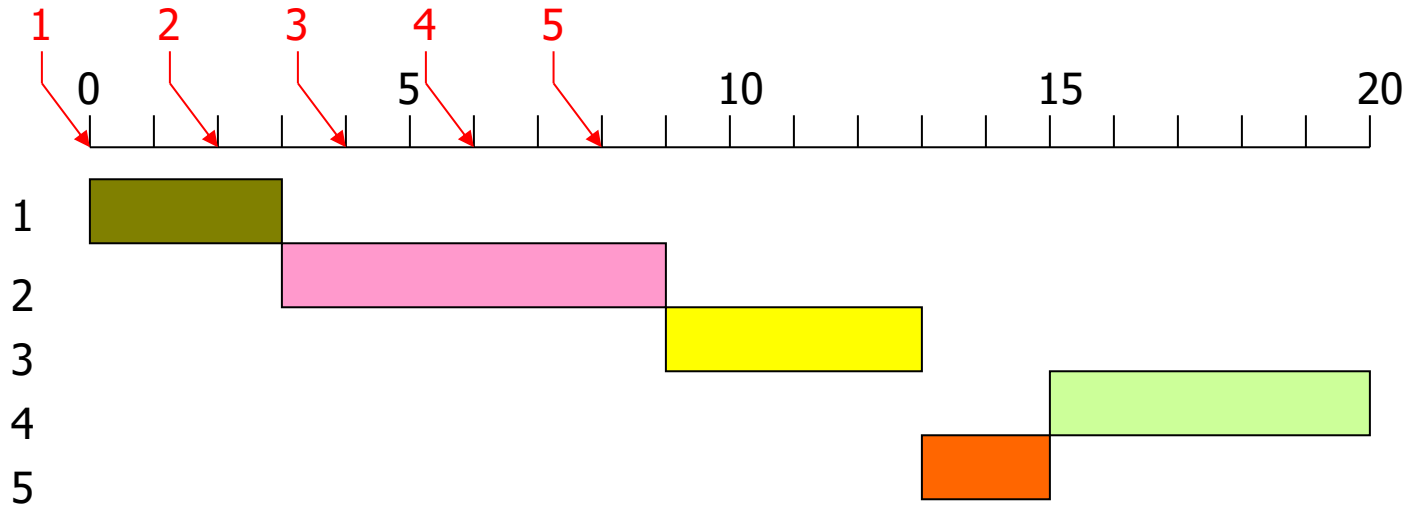
SRTN (Shortest Remaining Time Next)

- Prozess mit kürzester Restbedienzeit wird als nächster bearbeitet
  - Rechnender Prozess kann verdrängt werden
  - Nachteil: Schätzung der Bedienzeit stammt vom Benutzer
- Längere Prozesse können „verhungern“, wenn immer kürzere vorhanden sind



HRRN (Highest Response Ratio Next)

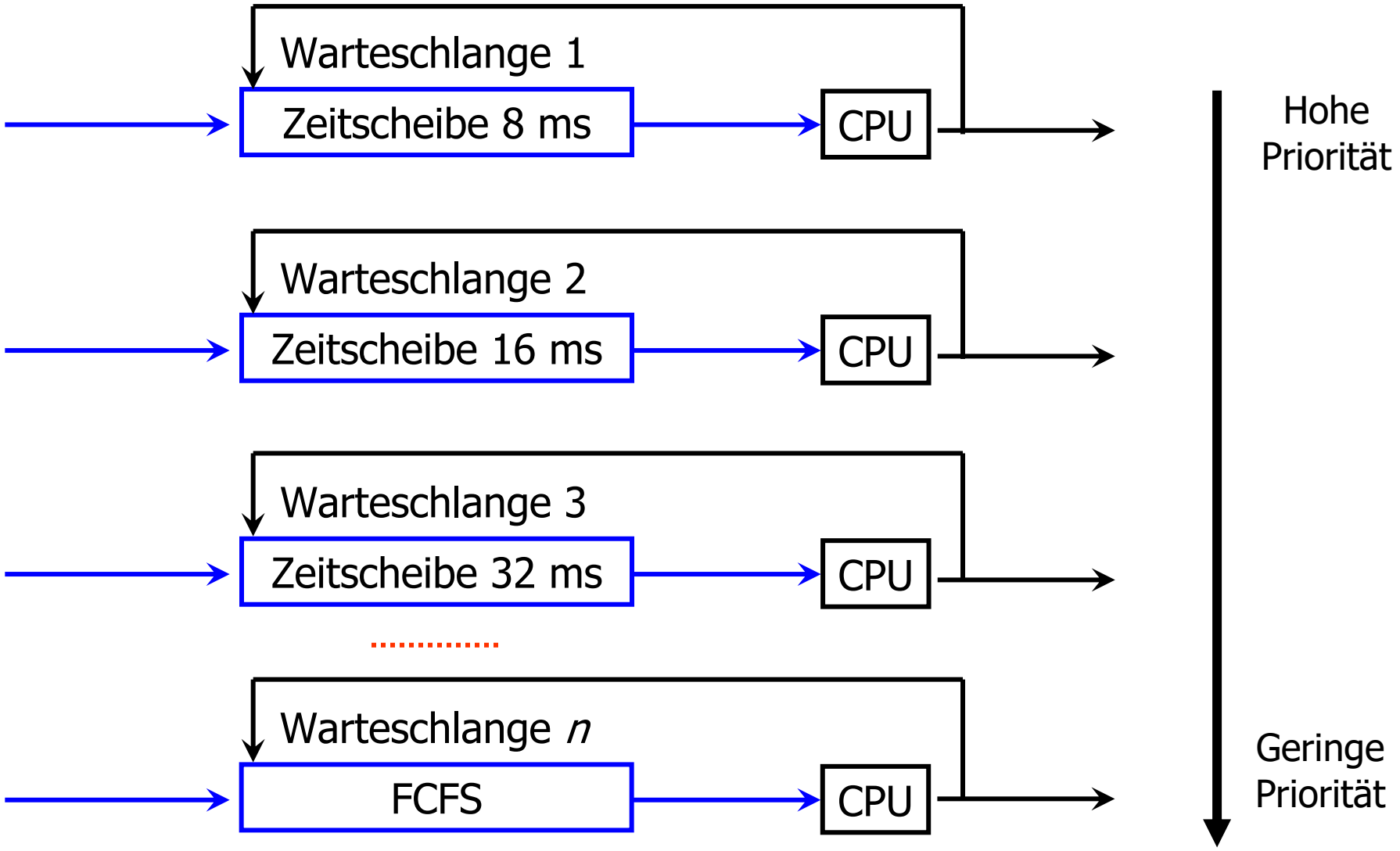
- Response Ratio rr ist definiert als $rr := \frac{\text{Wartezeit + Bedienzeit}}{\text{Bedienzeit}}$
- rr wird dynamisch berechnet und als Priorität verwendet
  - Prozess mit größtem rr-Wert wird als nächster ausgewählt.
  - Strategie ist nicht verdrängend
  - Bevorzugung kurzer Prozesse, lange Prozesse können durch Warten „Punkte sammeln".



3.3 Multilevel-Scheduling

- Kombination unterschiedlicher Scheduling-Verfahren
  - Durch die Verknüpfung können Schedulingstrategien besser auf die Betriebsform (Dialogbetrieb, Stapelbetrieb, …) abgestimmt werden
- Einfachste Realisierung
  - Unterteilung der Menge bereiter Prozesse in mehrere Listen
  - Prozesse werden – je nach gewünschter Betriebsform – bei ihrer Erzeugung einer Liste zugeordnet
  - Jede Liste kann mit einer eigenen Strategie verwaltet werden
  - Listen nach Priorität geordnet
  - Scheduler:
    - Suche höchst-priorisierte Liste, die nicht leer ist
    - Strategie dieser Liste bestimmt den nächsten Prozess

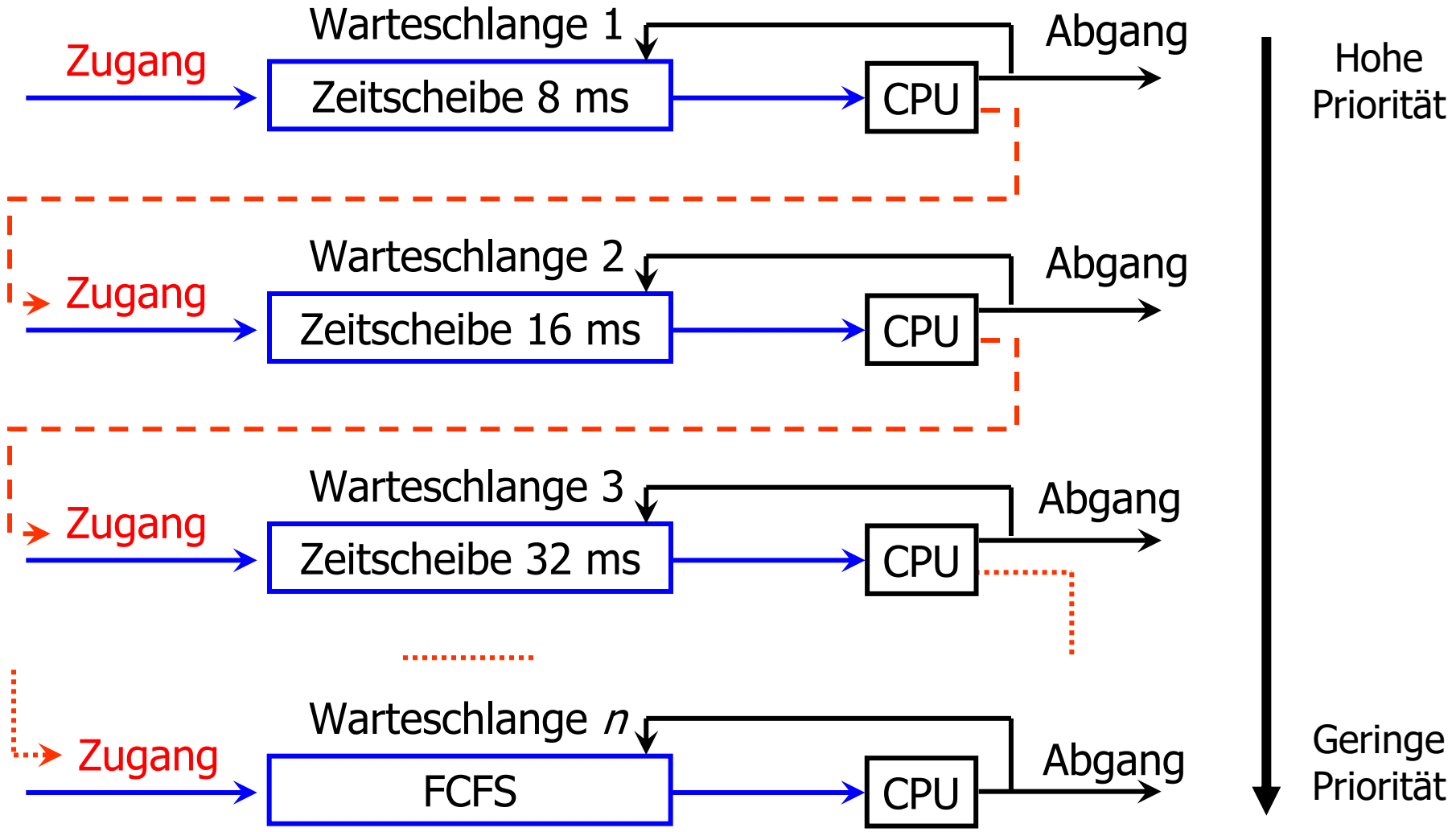
Multilevel-Scheduling



Multilevel-Feedback-Scheduling (Ursprung: Corbató, MULTICS)

- Zugehörigkeit von Prozess zu Multilevel-„Ebene“ nicht fest!
- Optimistische Annahme: Prozesse sind kurz, also schnell fertig (→ SJN), beginnen also mit hoher Priorität
- Lange (CPU-intensive) Prozesse → niedrigere Priorität!
  - behindern sonst Interaktivität des Systems
- Kurze (E/A-intensive; „schubweise“) Prozesse → höhere Priorität!
  - das sind gerade die interaktiven Prozesse (auf Tastendruck warten; kurz rechnen, z.B. Buchstabe an Textdokument anhängen; dann wieder warten)
- Optional: „Warte“-Historie eines bereiten Prozesses (→ HRRN)
  - Prozesse in den untersten Ebenen kommen u.U. nie dran, wenn immer höher priorisierte bereit sind (starvation)
  - Abhilfe: Aging (Erhöhung der Priorität bei langer Wartezeit)

Multilevel-Feedback-Scheduling (2)



Multilevel-Feedback-Scheduling (3)

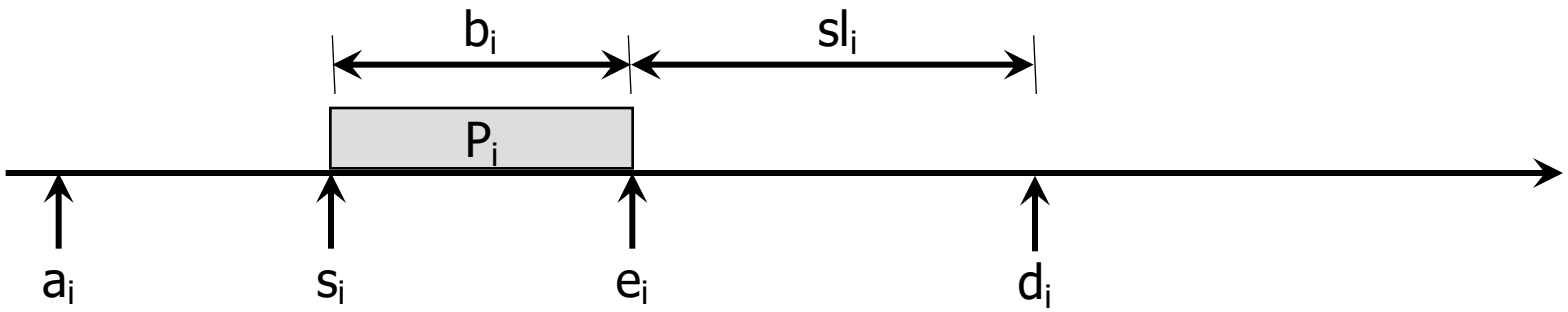
- Grundlage: RR-Verfahren mit Bereit-Liste, welche in mehrere Teillisten untergliedert wird
  - Unterschiedliche Länge der Zeitscheiben/Liste
  - Unterschiedliche Prioritäten/Liste
  - Unterste Warteschlange funktioniert nach dem FCFS-Prinzip
- Vorgehensweise
  - Verdrängte Prozesse (benötigen also mehr Zeit) kommen in eine Bereit-Liste mit längerer Zeitscheibe/geringerer Priorität
  - Prozesse, welche blockierende Operationen aufrufen oder den Prozessor freiwillig abgeben, verbleiben in der Warteschlange
    → Bevorzugung E/A-intensiver und interaktiver Anwendungen
  - Zusätzliche Feedback-Mechanismen ermöglichen eine Hochstufung (kürzere Zeitscheibe/höhere Priorität)

3.4 Scheduling mit Sollzeitpunkten

- Sollzeitpunkte treten in Echtzeitsystemen wie Steuerrechnern auf
  - Vollständige Bearbeitung eines Prozesses zum bestimmten, a-priori festgelegten Sollzeitpunkt (deadline), z.B. Auswertung von Messgrößen innerhalb knapper Zeitschranken
  - Einhaltung der Sollzeitpunkte teilweise kritisch für Gesamtsystem
- Können Verletzungen der Sollzeitpunkte toleriert werden?
  - Strikte Echtzeitsysteme (Hard real-time systems):
    - Verletzung wegen Systemausfall untolerierbar (z.B. Airbag, ABS, Öffnung von Ventilen beim Überdruck, …)
    - Oft Einsatz von Off-Line-Algorithmen notwendig
  - Schwache Echtzeitsysteme (Soft real-time systems):
    - Verletzung zwar tolerierbar, führt aber zu Qualitätsverlusten (z.B. Internet-Telefonie, Videoübertragung im Internet)

Wichtige Zeitpunkte bei Realzeitprozessen

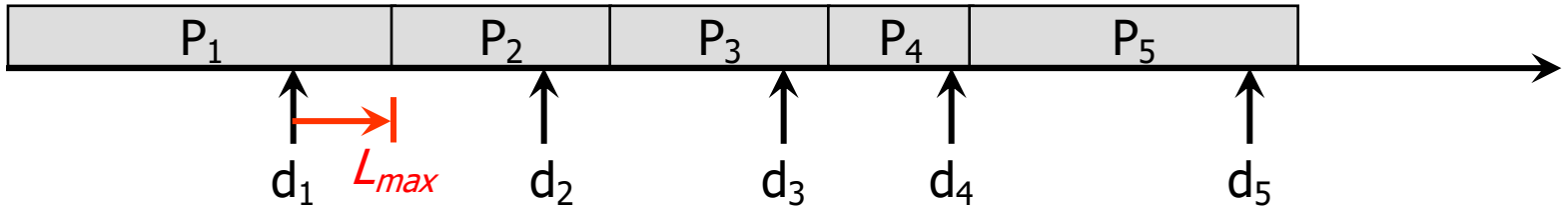
- Voraussetzung: Früheste Anfangszeit und späteste Endzeit a-priori bekannt
- Wichtige Zeitpunkte für einen Prozess Pi:
  - Frühester Startzeitpunkt ai
  - Tatsächlicher Startzeitpunkt si
  - Tatsächlicher Endzeitpunkt ei
  - Spätester Endzeitpunkt di (Sollzeitpunkt, deadline)
  - Bedienzeit (service time) bi = ei-si
  - Spielraum (slack time, laxity) sli = di-ei



Verspätungen

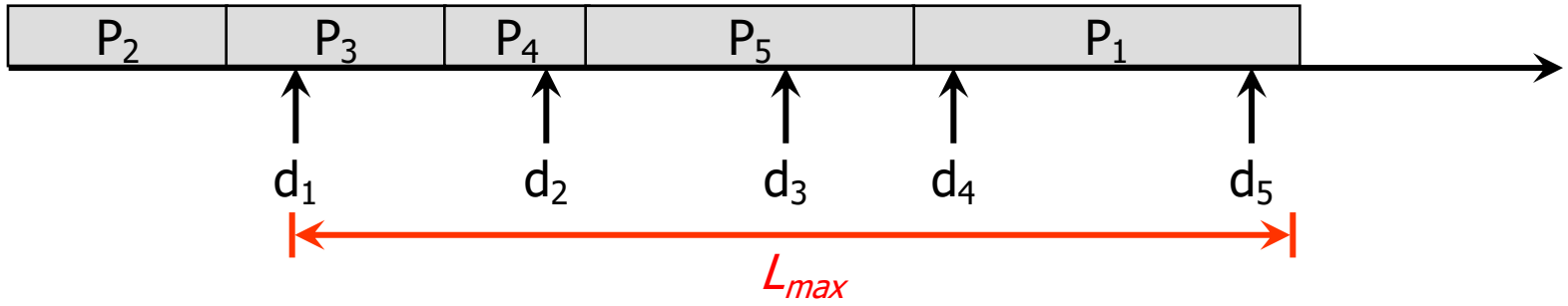
- Überschreitung der Sollzeit (ei > di): Verspätung (lateness) Li=ei–di
- Mögliche Zielfunktionen bei tolerierbaren Verspätungen (Soft-RTS)

1. Minimierung der maximalen Verspätung Lmax



Ergebnis: Maximale Verspätung minimiert, aber alle Deadlines verpasst

2. Einhaltung von möglichst vielen Deadlines, ohne Berücksichtigung der max. Verspätung



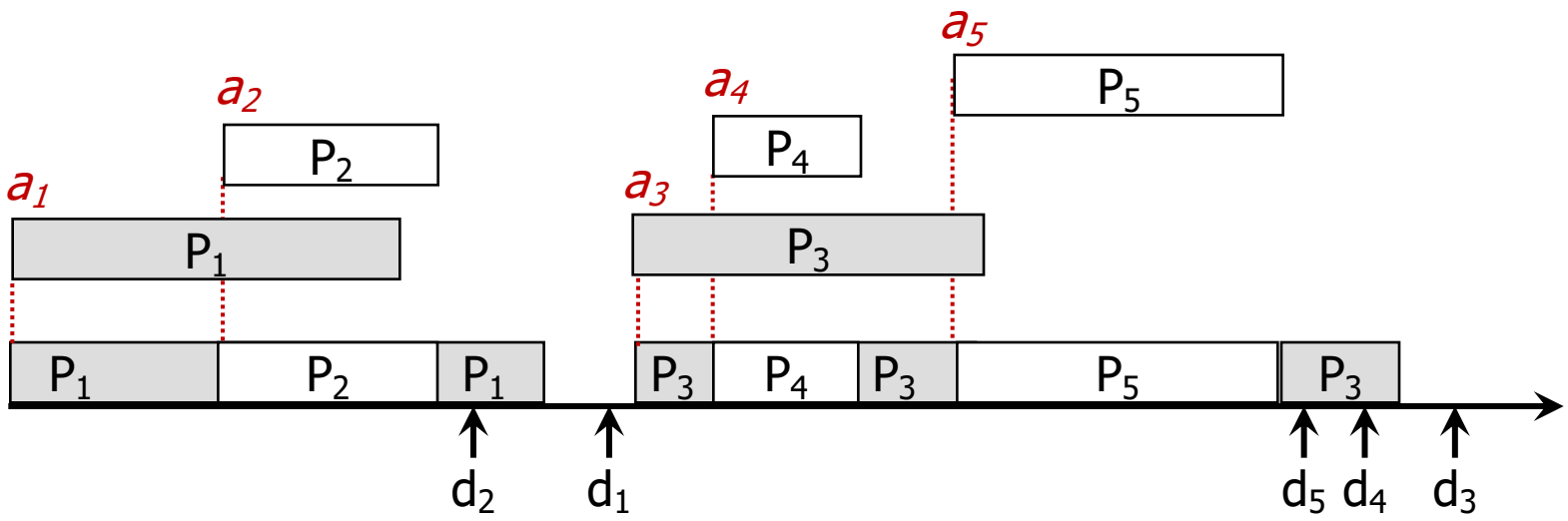
Minimierung der maximalen Verspätung

- Bei 1-CPU-Systemen ohne Verdrängung und ohne Abhängigkeiten zwischen Prozessen
  - Scheduling durch Permutation der Prozesse
- Theorem EDD (Earliest Due Date), Jacksons Regel
  - Relaxation: Alle Prozesse können zu jedem Zeitpunkt beginnen
  - Jeder Ablaufplan, in dem die Prozesse nach nicht fallenden Sollzeitpunkten geordnet ausgeführt werden, ist optimal bzgl. Lmax
  - Lediglich Sortiervorgang mit O(n log n) notwendig
- Praxis
  - Durch die Einführung von unterschiedlichen Startzeitpunkten ($\exists$ i,j: ai $\neq$ aj) wird das Problem NP-schwer, d.h. es ist nicht in polynomialer Zeit optimal lösbar

Minimierung der maximalen Verspätung (2)

- Voraussetzung: Vorgegebene Startzeitpunkte, Verdrängung möglich

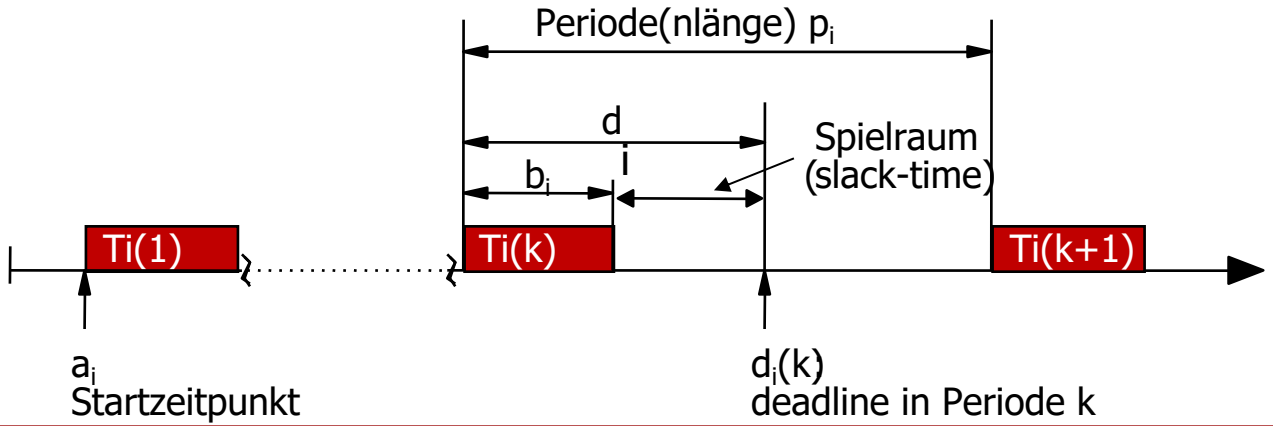
- Theorem EDF (Earliest Deadline First)
  - Jeder Ablaufplan, in dem zu jedem Zeitpunkt der Prozess mit dem frühesten Sollzeitpunkt unter allen ablaufbereiten Prozessen zugeordnet wird, ist optimal bezüglich der maximalen Verspätung



Periodische Prozesse

- Periodische Aufgaben mit Deadlines kommen zu bestimmten Zeitpunkten immer wieder
  - Jeder Prozess ist gekennzeichnet durch Periode bzw. die dazu reziproke Rate

- Ist die Schedulingaufgabe lösbar (schedulability test, feasibility test)?
  - Für jeden periodischen Prozess muss gelten: $0 < b_i \leq d_i \leq p_i$
  - Bei mehreren periodischen Prozessen muss gelten: $\sum_i \frac{b_i}{p_i} \leq 1$



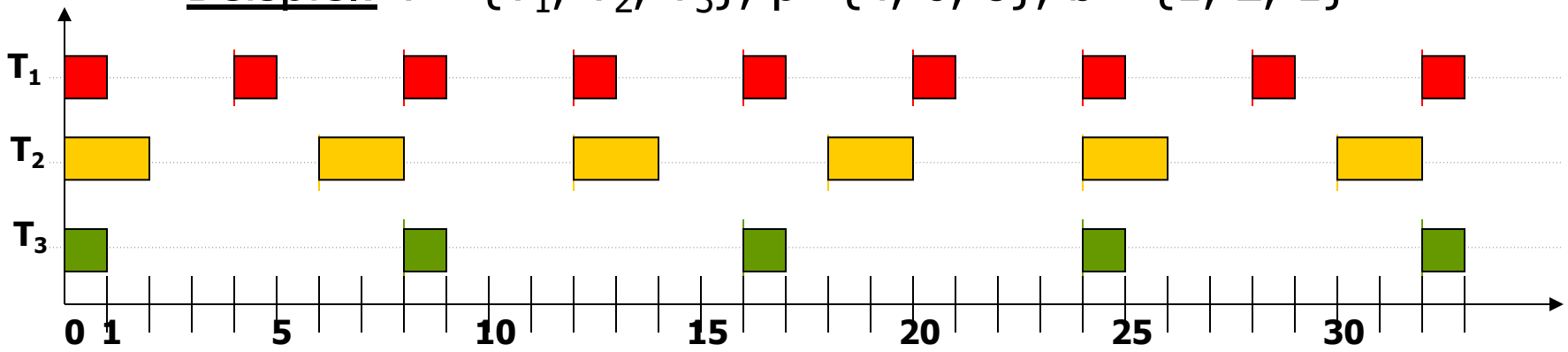
Periodische Prozesse

- Ratenmonotones Verfahren (rate monotonic scheduling)
  - Statische Priorität für jeden Prozess, die umgekehrt proportional ist zu der Periode, höhere Priorität verdrängt niedrige Priorität
  - Prozess mit der kleinsten Periode hat die höchste Priorität
- Voraussetzung: Prozesse sind unabhängig und Sollzeitpunkte fallen mit den Perioden zusammen
- Satz: Menge von n periodischen Prozessen kann durch ein ratenmonotones Verfahren eingeplant werden, wenn folgendes gilt (hinreichende Bedingung):
- $$\sum_{i=1}^{n} \frac{b_i}{p_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$
  - Links: benötigte Prozessorleistung
  - Rechts: obere Schranke für einen zulässigen Schedule
- Bei großen n $\Rightarrow$ CPU-Auslastung höchstens ln 2 $\approx$ 69,3 %

Rate Monotonic Scheduling (RMS)

- Annahmen
  - Prozess Ti ist periodisch mit Periodenlänge pi
  - Deadline ist di = pi
  - Ti ist unmittelbar nach pi erneut bereit
  - Ti hat eine konstante Bedienzeit bi (<= pi)
  - Je kleiner die Periode, desto höher die Priorität

Beispiel: T ={T_1, T_2, T_3}, p={4, 6, 8}, b ={1, 2, 1}



Wie sieht die Einplanung auf einem Prozessor aus?

Feasibility test: Geht das?

Beispiel: T ={T_1, T_2, T_3}, p={4, 6, 8}, b ={1, 2, 1}

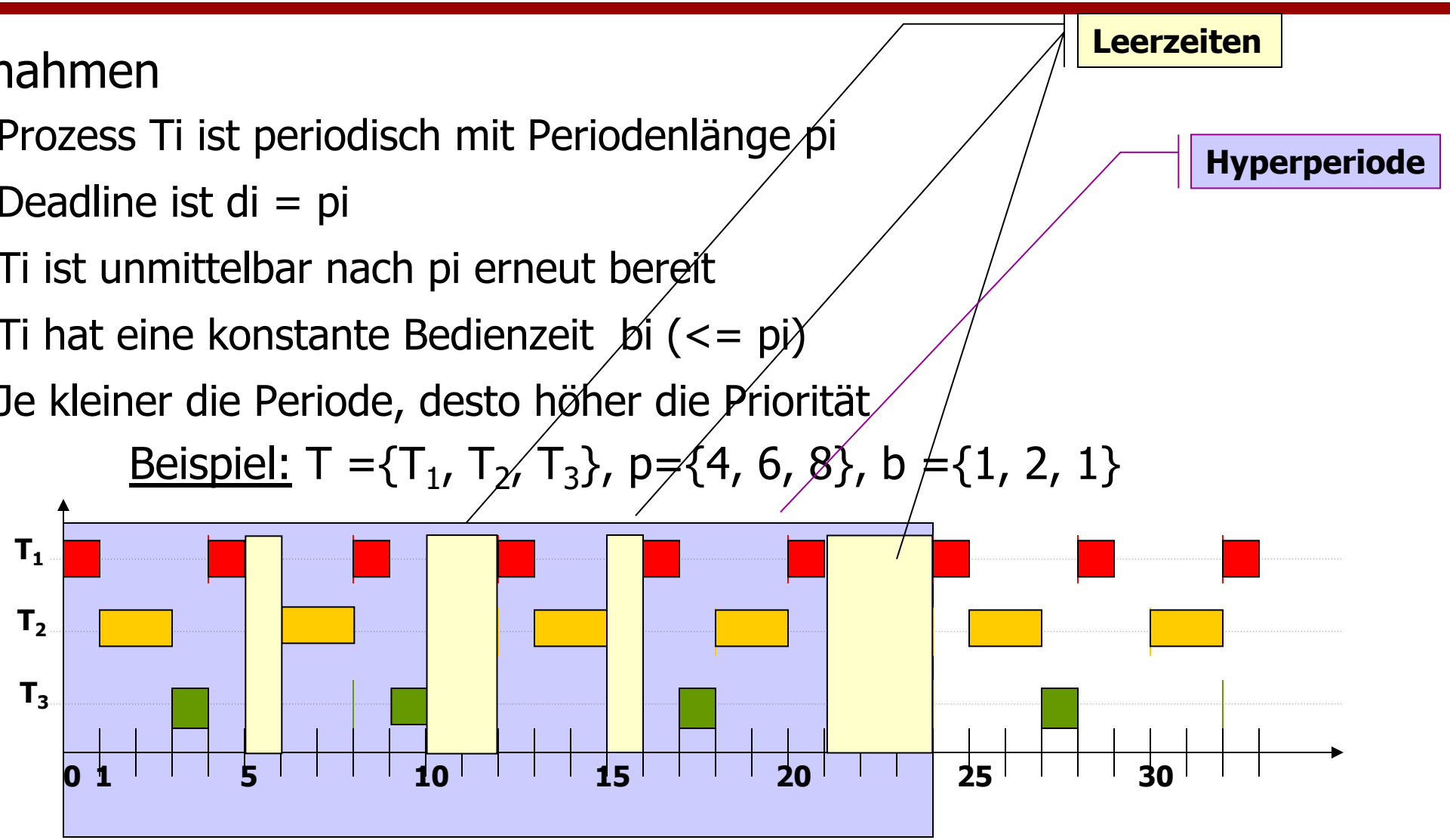
- Trivial $0 < b_i \leq d_i \leq p_i$ $\quad\quad \sum_i \frac{b_i}{p_i} \leq 1$

- Entscheidend: $\sum_{i=1}^{n} \frac{b_i}{p_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$

Rate Monotonic Scheduling (RMS)

- Annahmen
  - Prozess Ti ist periodisch mit Periodenlänge pi
  - Deadline ist di = pi
  - Ti ist unmittelbar nach pi erneut bereit
  - Ti hat eine konstante Bedienzeit bi (<= pi)
  - Je kleiner die Periode, desto höher die Priorität

Beispiel: T ={T_1, T_2, T_3}, p={4, 6, 8}, b ={1, 2, 1}



Ergebnis des RMS-Beispiels

Die allgemeine **notwendige Bedingung** ist erfüllt:

$$(1/4 + 2/6 + 1/8) = 17/24 <= 1$$

Auch das **hinreichende RMS-Kriterium** ist erfüllt:

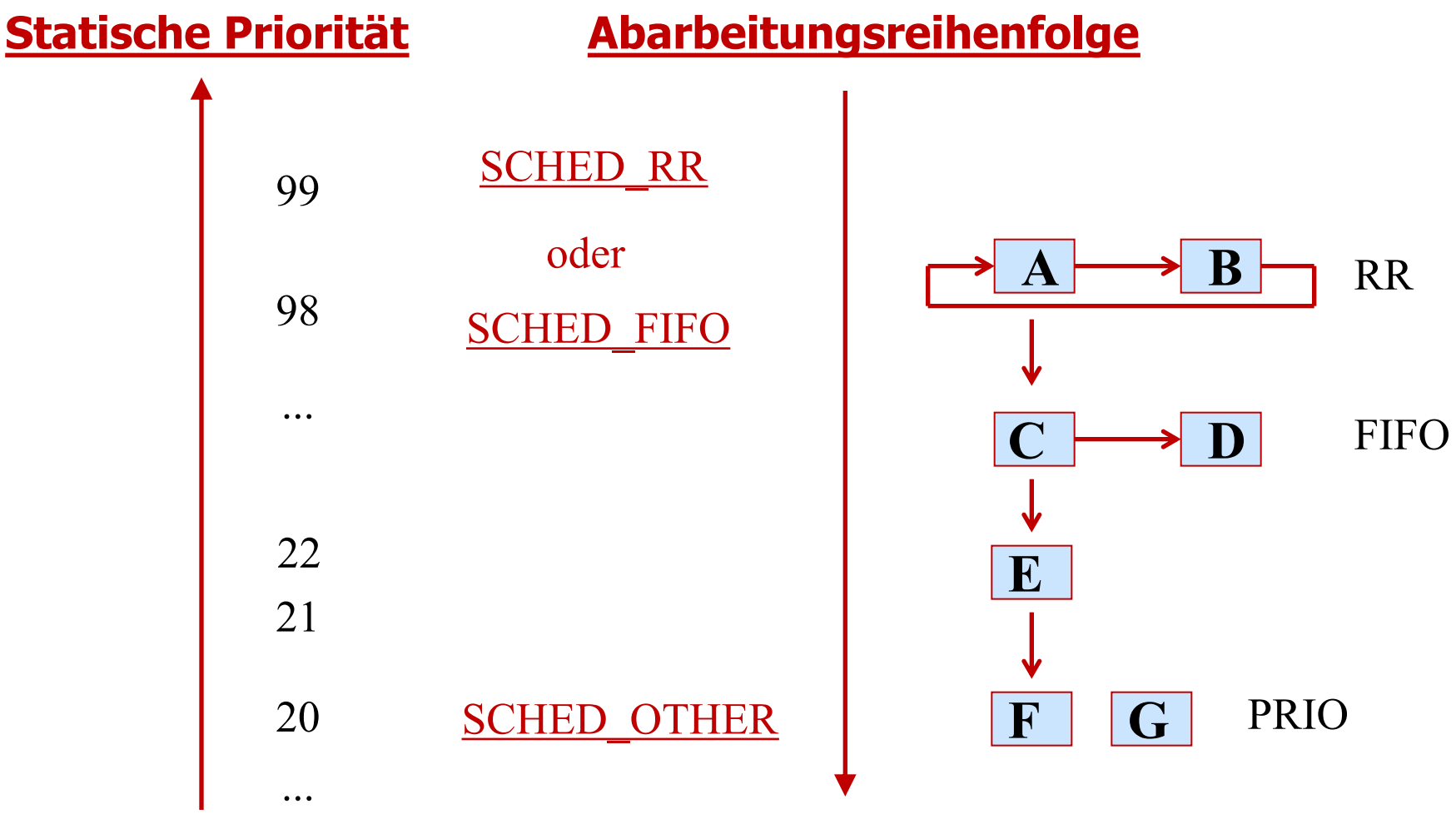
$$(1/4 + 2/6 + 1/8) = 17/24 = 0{,}7083 \approx 71\ \%$$

$$3\,(2^{1/3} - 1) \approx 78\ \%$$

3.5 Fallbeispiel: UNIX-Scheduling

- UNIX/Linux-Scheduling basiert auf Multilevel-Feedback-Scheduling
  - Statische und dynamische Prioritäten zur Prozessauswahl, Verwaltung der lauffähigen Prozesse in einer Liste (runqueue)
- Scheduling-Verfahren (policies)
  - SCHED_OTHER: basiert auf PRIO-P, für „normale" Prozesse
    - Statische Priorität = 20, dynamische Priorität (nice): [-20 19]
    - Verdrängung durch Prozesse mit höherer statischer Priorität, nicht verbrauchte Zeitscheibe bleibt als „Guthaben" erhalten
  - SCHED_FIFO: FCFS, für „Echtzeitprozesse" (Real-Time FIFO)
    - Statische Priorität: 1-99, bei gleicher Priorität Wahl des ersten Prozesses, Verdrängung durch Prozesse mit höherer statischer Priorität
  - SCHED_RR: Round-Robin, für „Echtzeitprozesse" (Real-Time RR)
    - Statische Priorität: 1-99, bei gleicher statischer Priorität -> RR
    - Nach Ablauf der Zeitscheibe -> Einordnung ans Ende der runqueue
    - Verdrängung durch Prozesse mit höherer statischer Priorität

Multilevel-Scheduling von Prozessen



Scheduling-Algorithmus in Linux

- Entferne alle Prozesse aus der Wartschlange runqueue, deren Zustand nicht TASK_RUNNING ist, d.h. die nicht ablaufbereit sind
- Bewerte jeden lauffähigen Prozess aus der runqueue und wähle den Prozess mit der höchsten Bewertung
- Wenn alle Zeitkonten (Quantum) der lauffähigen Prozesse (blockierte werden nicht berücksichtigt) abgelaufen sind, dann berechne die Zeitkonten aller Prozesse neu
  - Quantum: Bestimmte Anzahl sog. Uhrticks üblicherweise 20, ein Uhrtick etwa 10ms beträgt
  - Bestimme den auszuführenden Prozess und rufe Dispatcher auf
- CPU wird entzogen, wenn
  - Quantum vollständig verbraucht (Quantum=0)
  - Thread blockiert wegen E/A
  - Thread höherer Priorität ist ablaufbereit

Prozessbewertung

- Bewertung eines Prozesses durch die Funktion goodness(...)
  - Rückgabe –1: Freiwillige Abgabe der CPU
  - Rückgabe 0: Quantum aufgebraucht
  - Rückgabe [1-1000]: Normaler Prozess
  - Rückgabe > 1000: Echtzeitprozess
  - Je größer der Rückgabewert, desto besser ist die Bewertung

- Berechnung der Funktion goodness()
  - Bei normalen Prozessen
    - Allgemein: Güte = quantum + priorität
    - Gleicher Prozess hat noch Zeit übrig: Güte = quantum + priorität + 1
    - Gleicher Adressraum wie aktueller Prozess: Güte = quantum + priorität + 1
  - Bei Echtzeit-Prozessen: Güte = 1000 + Priorität
- Neuberechnung: Quantum_neu = Quantum_alt/2+Basispriorität

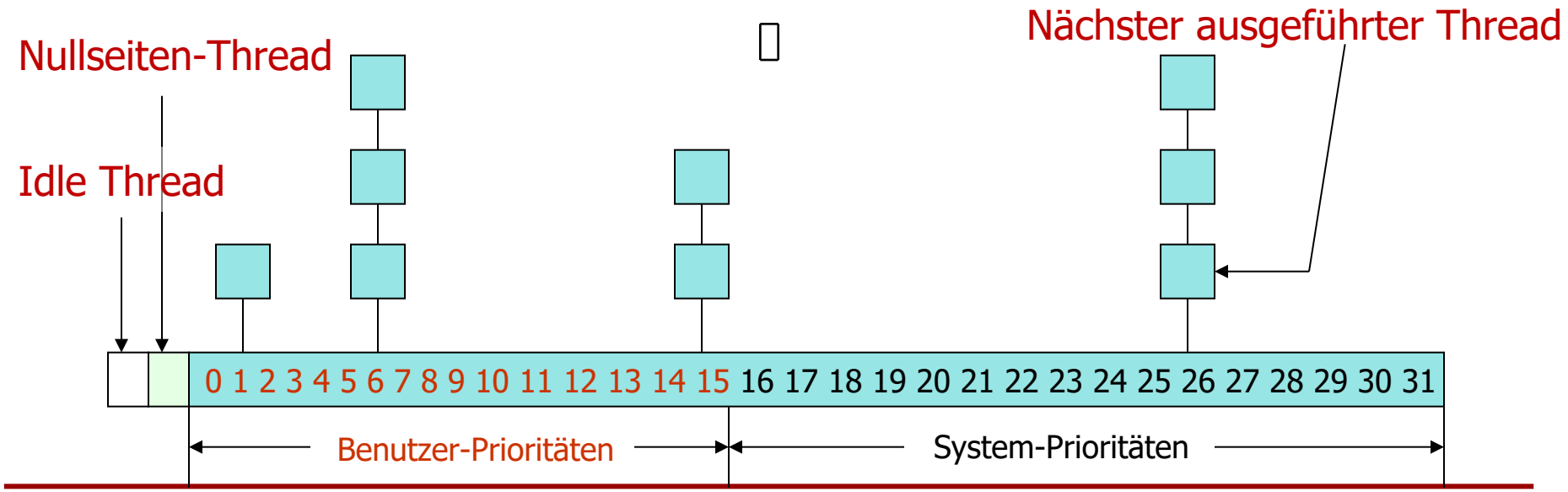
Scheduling in Windows

- Kombination aus Prozess- und Threadpriorität ergibt 32 Werte [0, 31]
  - Systemprioritäten 16..31 ⇒ Zuweisung durch Administrator
  - Benutzerprioritäten 0..15 (Änderung mit API SetThreadPriority)

		Win32-Prozessklassen-Prioritäten					
		Echtzeit	Hoch	Über normal	Normal	Unter normal	Idle
Win32-Thread-Prioritäten	Zeitkritisch	31	15	15	15	15	15
	Höchste	26	15	12	10	8	6
	Über normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Unter normal	23	12	9	7	5	3
	Niedrigste	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Windows: Arbeitsweise des Scheduling-Algorithmus

- Verwaltung der Prioritätsliste mit 32 Prioritäten
  - Jeder Eintrag enthält Liste mit allen wartenden Threads der gleichen Priorität
  - Durchlaufen der Liste von Priorität 31 bis Priorität 0
  - Bei nichtleerem Eintrag ⇒ führe die Prozesse nach Round-Robin aus
- Spezielle Threads
  - Null Thread: läuft im Hintergrund, überschreibt Speicherseiten mit Nullen
  - Idle Thread: Läuft, wenn keine anderen Threads inkl. Null Thread aktiv



Zeitscheibenlänge im Windows-Scheduler

- Zeitscheibenlängen
  - Standardeinstellung: 20ms, Einzelprozessor-Server: 120ms
  - Multiprozessor-Systeme: abhängig von Taktfrequenz
  - Einstellungen können um Faktor 2, 4 oder 6 erhöht werden
    - Kürzere Zeitscheiben bevorzugen interaktive Benutzer
    - Längere Zeitscheiben erfordern weniger Kontextwechsel und sind effizienter

- Verringerung der Priorität
  - Verbraucht ein Thread seine gesamte nächste Zeitscheibe, so wird seine Priorität um eine Einheit verringert, solange die aktuelle Priorität höher ist als seine Basispriorität
    (vgl. Multilevel Feedback Queue)