Kapitel 8: Scheduling

8: Scheduling

- Dieses Kapitel beschäftigt sich mit Scheduling
- Im Fokus des Kapitels stehen Schedulingstrategien
- Task: Prozess oder Thread
- Dispatching: Umschalten des Prozessors bei einem Prozesswechsel
- Scheduling (Planen): Entscheidung welcher Task zu welchem Zeitpunkt für welche Zeitspanne abgearbeitet wird:
  - Kooperatives Multitasking
  - Präemptives Multitasking
- Bei Echtzeitsystemen werden nicht nur Tasks sondern auch andere Ressourcen (z. B. Speicher) geplant und zugeteilt

Dispatching

- Beim Prozesswechsel entzieht der Dispatcher dem rechnenden Prozess die CPU und teilt sie einem anderen Prozess zu
- Übergänge aus oder in den Zustand rechnend (running) implizieren einen Prozesswechsel
- Aufgaben des Dispatchers beim Prozesswechsel:
  - Der Kontext (Registers) des laufenden Prozesses wird in den Prozesskontrollblock (PCB) übertragen
  - Die CPU wird einem anderen Prozess zugeteilt
  - Der Kontext des neuen Prozesses wird aus seinem PCB wieder hergestellt

Prozesswechsel

Interrupt Service Routine (ISR, Interrupthandler) eines Dispatchers:

```
void ISR_dispatch() {
  cli(); // disable interrupts
  store_context();
  task t = next_task();
  load_context(t);
  sei(); // enable interrupts
  reti(); // return
}
```

Der Leerlaufprozess (Idle Task)

- Moderne Betriebssysteme verfügen über einen Leerlaufprozess
- Der Leerlaufprozess ist immer aktiv (Kernelprozess)
- Der Leerlaufprozess hat die niedrigste Priorität
- Ist kein anderer Prozess im Zustand bereit (runnable) oder running dann kommt der Leerlaufprozess zum Zug
- Durch den Leerlaufprozess muss der Sonderfall kein aktiver Prozess nicht berücksichtigt werden
- Der Leerlaufprozess versetzt die CPU in einen Stromsparmodus, falls dieses Feature durch die Hardware unterstützt wird

Die 5 Ziele des Schedulings

1. Minimierung der (mittleren) Lebensdauer

   Use-Case: Interaktive Systeme

2. Maximierung des Durchsatzes

   Use-Case: Server

3. Maximierung der Prozessorauslastung

   Use-Case: Supercomputer

4. Einhaltung der vorgegebenen Zeitschranken

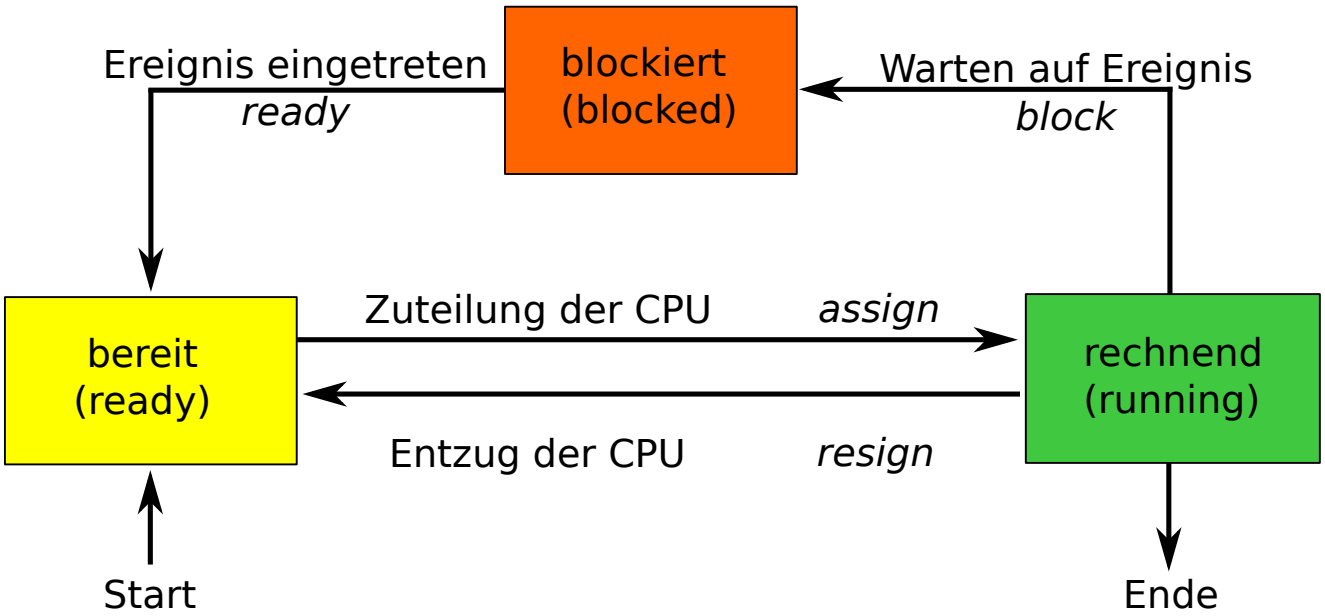
   Use-Case: Echtzeitsysteme

5. Fairness (Garantie des Fortschritts)

   Use-Case: Ausführung eines Tasks

   Ein Schedulingverfahren ist unfair falls Tasks verhungern können

3-Zustands-Taskmodell



Grundlage für dieses Kapitel ist das 3-Zustands-Taskmodell mit den Zuständen bereit, laufend und blockiert.

Arten von Schedulingverfahren

Kooperatives (nicht-präemptives) Scheduling

Es findet nur ein Kontextwechsel statt, falls ein Task im Zustand running . . .

- . . . in den Zustand blocked wechselt.
- . . . freiwillig in den Zustand ready wechselt.
- . . . sich beendet.

Präemptives Scheduling

Ein Task im Zustand running kann die Nutzung der CPU zu einem beliebigen Zeitpunkt entzogen werden, um sie einem anderen Task zu übertragen.

8.1: Klassische Schedulingstrategien

- First Come First Served (FCFS)
- Last Come First Served (LCFS)
- Round Robin (RR)
- Strict Priority Scheduling (SPC)

Zeitpunkte und Zeitspannen

- Arrival Time: t_a (Startzeitpunkt/ Ankunft)
- Starting Time: t_s (Laufzeit Begin)
- Completion Time: t_c (Laufzeit Ende)
- Lifetime: $t_{life} = t_c - t_a$ (Gesamtlaufzeit)
- Wait Time: $t_{wait} = t_{life} - t_{run}$ (Task im Zustand `ready` oder `blocked`)
- Running Time: $t_{run} = t_{life} - t_{wait}$ (Task im Zustand `laufend`)

Metriken

Qualitätsmetriken für n Tasks.

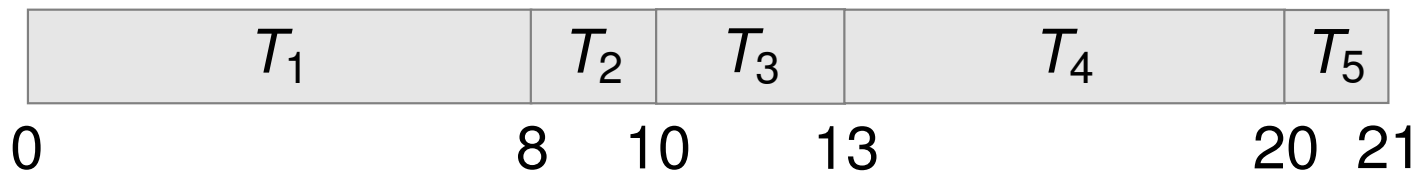
- Mittlere Lebensdauer: $r = \frac{1}{n} \sum_{i=1}^{n} t_{life,i}$
- Mittlere Wartezeit: $w = \frac{1}{n} \sum_{i=1}^{n} t_{wait,i}$

First Come First Served (FCFS)

- Die „Wer zuerst kommt, mahlt zuerst “- Strategie
- First In First Out (FIFO) Prinzip
- FCFS ist **kooperativ**
- Analogie: Warteschlange an der Supermarktkasse
- FCFS ist fair: Verhungern von Tasks ist nicht möglich.
- Die mittlere Wartezeit kann sehr hoch sein.

First Come First Served: Beispiel

Task	Laufzeit	Ankunft
T_1	8 ms	0
T_2	2 ms	1
T_3	3 ms	4
T_4	7 ms	6
T_5	1 ms	7



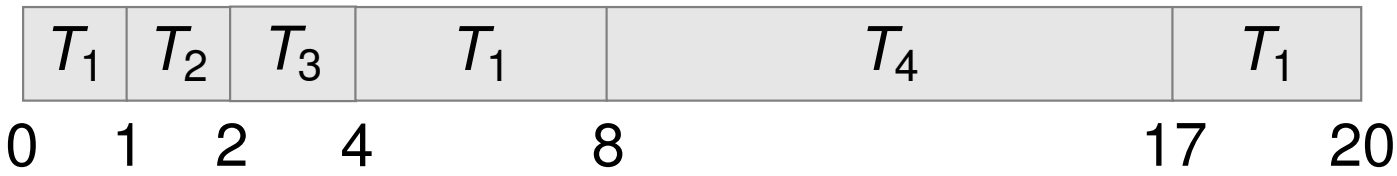
- Mittlere Lebensdauer: $(8 + 9 + 9 + 14 + 14)/5 = 10.8$ ms
- Mittlere Wartezeit: $(0 + 7 + 6 + 7 + 13)/5 = 6.6$ ms

Last Come First Served (LCFS)

- Last In First Out (LIFO) Prinzip
- Präemptiv: Ein neuer Task in der Ready-Queue verdrängt den aktuell laufenden Task von der CPU
- Tasks mit kurzer Laufzeit werden bevorzugt, da geringere Chance auf Verdrängung
- Task mit langer Laufzeit werden ggf. mehrfach verdrängt und dadurch stark verzögert
- LCFS ist unfair: Das Verhungern von Tasks ist möglich

Last Come First Served: Beispiel

Task	Laufzeit	Ankunft
T_1	8 ms	0
T_2	1 ms	1
T_3	2 ms	2
T_4	9 ms	8



- Mittlere Lebensdauer: $(20 + 1 + 2 + 9)/4 = 8$ ms
- Mittlere Wartezeit: $(12 + 0 + 0 + 0)/4 = 3$ ms

Round Robin (RR)

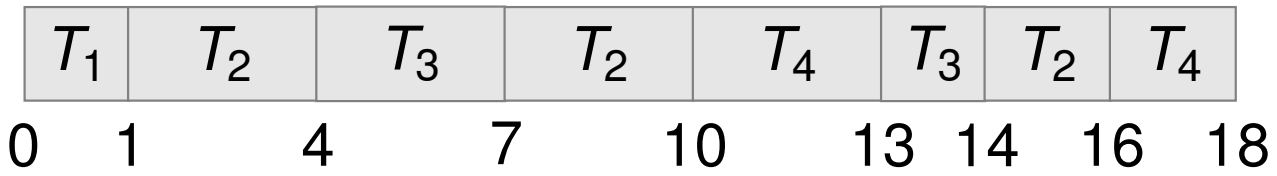
- Es werden Zeitspannen (engl. *Time Slices* fester Länge festgelegt
- Die Länge einer Zeitspanne wird Quantum genannt
- Die Taskw werden in einer Warteschlange verwaltet
- Der vorderste Task darf für ein Quantum die CPU nutzen und wird dann wieder ans Ende der Warteschlange eingereiht
- Round Robin folgt dem FIFO-Prinzip
- Ein laufender Task kann die CPU (un)freiwillig früher freigeben (z. B. Signal, Terminierung, oder `sleep()`-Aufruf)
- Nach Freigabe wird die CPU umgehend neu belegt

Round Robin: Anmerkungen

- Round Robin ist fair
- Round Robin ist präemptiv
- Für Quantum = ∞ verhält sich RR wie FCFS.
- Die Zugriffszeit auf die CPU wird gleichmäßig und fair auf die vorhandenen Tasks aufgeteilt
- Die Länge eines Quantums bestimmt die Performance:
  - Zu kurz: Es finden viele Dispatcher- und Scheduler-Aufrufe statt
    $\Longrightarrow$ hoher Verwaltungsoverhead
  - Zu lang: Die Gleichzeitigkeit geht verloren
    $\Longrightarrow$ das System *ruckelt*
- Die Länge eines Quantums ist üblicherweise im ein- oder zweistelligen Millisekundenbereich

Round Robin: Beispiel mit Quantum=3

Task	Laufzeit	Ankunft
T_1	1 ms	0
T_2	8 ms	1
T_3	4 ms	2
T_4	5 ms	5



- Mittlere Lebensdauer: $(1 + 15 + 12 + 13)/4 = 10.25$ ms
- Mittlere Wartezeit: $(0 + 7 + 8 + 8)/4 = 5.75$ ms

Prioritäten

- Jeder Task T hat eine Priorität p.
- Wir schreiben T_i^p für den i-ten Task mit Priorität p
- T_i^p hat eine höhere Priorität als $T_j^{p'}$ falls $p > p'$
- T_i^p hat eine niedrigere Priorität als $T_j^{p'}$ falls $p < p'$ gilt
- T_i^p hat die gleiche Priorität wie $T_j^{p'}$ falls $p = p'$ gilt
- **Frage:** Welcher Task hat die höchste/niedrigste Priorität?

$$T_1^5, \quad T_2^1, \quad T_3^0, \quad T_4^8 \quad \text{oder} \quad T_5^7.$$

Strict Priority Scheduling (SPS)

- Tasks werden nach ihrer Priorität (d. h. nach ihrer Dringlichkeit) abgearbeitet
- Der bereite Task mit der aktuell höchsten Priorität bekommt die CPU zugewiesen
- Falls präemptiv kann der laufende Task von einem neuen Task mit höherer Priorität verdrängt werden
- Normalerweise wird für jede Priorität eine RR-Warteschlange verwaltet (SPS-RR)

SPS-Probleme

Probleme

- SPS ist unfair, da Tasks mit niedriger Priorität verhungern können
- Deadlock durch Priority Inversion:
  Task mit hoher Priorität muss auf einen Task mit niedrigerer Priorität warten

Lösung: Priotitäsvererbung

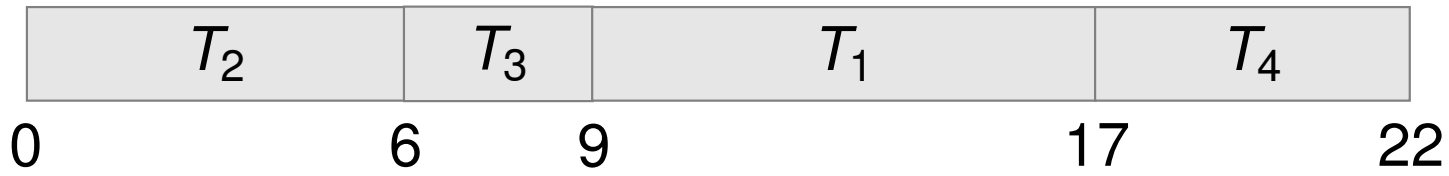
Die Priorität des Task welcher einen Mutex sperrt wird dynamisch auf das Maximum aller Task welche auf diesen Mutex warten gesetzt, solange bis er diesen wieder entsperrt.

SPS: Einfaches Beispiel

- Auf einer CPU sollen vier Tasks verarbeitet werden

Task	Laufzeit	Ankunft	Prio
T_1	8 ms	0	0
T_2	6 ms	0	6
T_3	3 ms	0	1
T_4	5 ms	1	0

- Ausführungsreihenfolge der Tasks (Gantt-Diagramm)

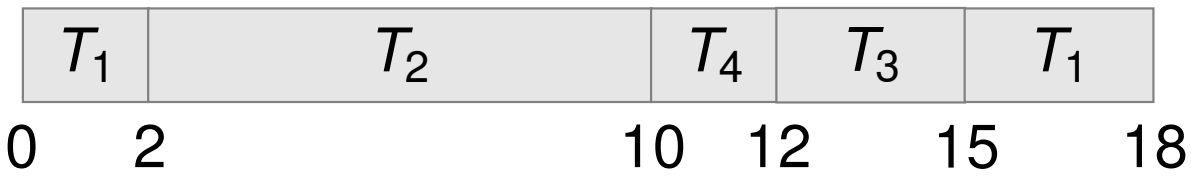


- Mittlere Lebensdauer: $(6 + 9 + 17 + 21)/4 = 13.25$ ms
- Mittlere Wartezeit: $(9 + 0 + 6 + 16)/4 = 7.75$ ms

SPS-Beispiel: Verdrängen

Task	Laufzeit	Ankunft	Prio
T_1	5 ms	0	0
T_2	8 ms	2	7
T_3	3 ms	2	3
T_4	2 ms	6	7

Ausführungsreihenfolge der Tasks (Gantt-Diagramm)



- Mittlere Lebensdauer: $(18 + 8 + 13 + 6)/4 = 11.25$ ms
- Mittlere Wartezeit: $(13 + 0 + 10 + 4)/4 = 6.75$ ms

SPS-Beispiel: Verhungern

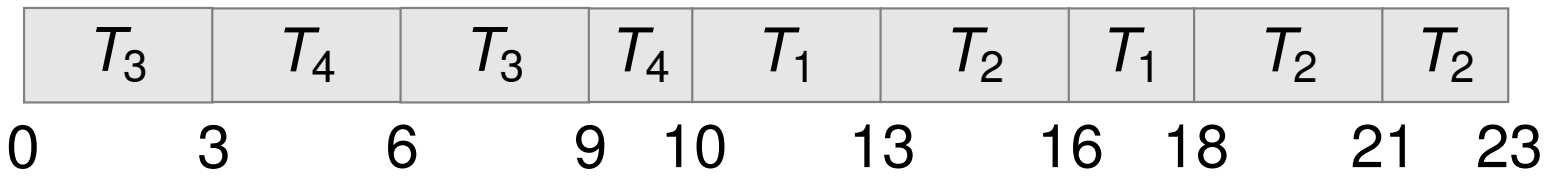
Task	Laufzeit	Ankunft	Prio
T_1	5 ms	0	0
T_2	1 ms	1	1
T_3	1 ms	2	1
...	...	...	...

Bei diesem Beispiel verhungert der Task T_1

SPS-RR: Einfaches Beispiel mit Quantum = 3

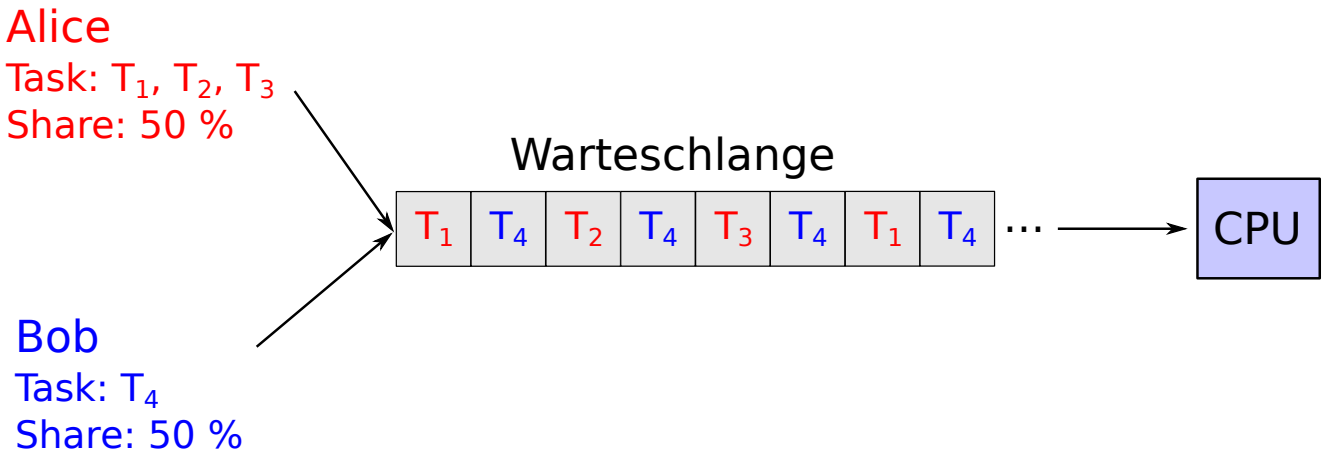
Task	Laufzeit	Ankunft	Prio
T_1	5 ms	0	0
T_2	8 ms	0	0
T_3	6 ms	0	1
T_4	4 ms	0	1

Ausführungsreihenfolge der Tasks (Gantt-Diagramm)



- Mittlere Lebensdauer: $(9 + 10 + 18 + 23)/4 = 15$ ms
- Mittlere Wartezeit: $(13 + 15 + 3 + 6)/4 = 9.25$ ms

Fair Share

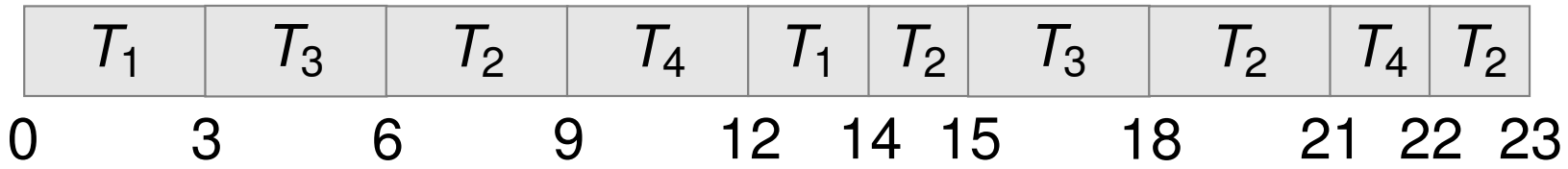


- Aufteilung der Ressourcen zwischen Gruppen von Tasks
- Rechenzeit wird den Benutzern und nicht den Tasks zugeteilt
- Die Rechenzeit, die ein Benutzer erhält, ist unabhängig von der Anzahl seiner Tasks
- Die Ressourcenanteile, die die Benutzer erhalten, nennt man `Shares`
- Populär beim Cluster- und Grid-Computing
- Für jeden Benutzer kann eine andere Schedulingstrategie angewandt werden (Standard oftmals Round Robin)

Fair-Share: RR-Beispiel mit Quantum = 3

Task	Laufzeit	Ankunft	User
T_1	5 ms	0	A
T_2	8 ms	0	A
T_3	6 ms	0	B
T_4	4 ms	0	B

Ausführungsreihenfolge der Tasks (Gantt-Diagramm)

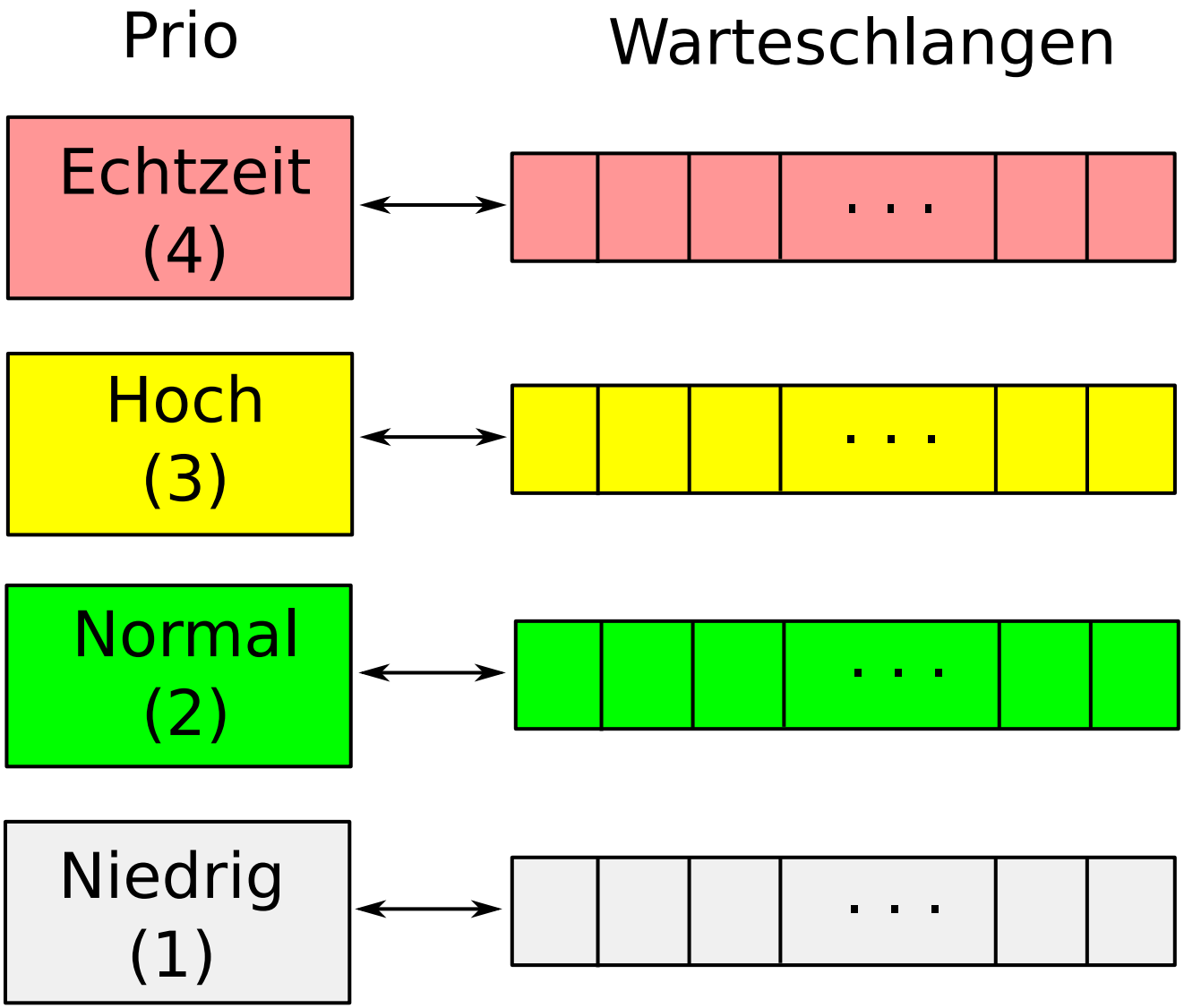


- Mittlere Lebensdauer: $(14 + 23 + 18 + 22)/4 = 19,25$ ms
- Mittlere Wartezeit: $(9 + 15 + 12 + 18)/4 = 13,5$ ms

Multilevel-Scheduling

- Bei jeder Scheduling-Strategie müssen Kompromisse bzgl. der unterschiedlichen Scheduling-Kriterien gemacht werden.
- Vorgehen in der Praxis: Mehrere Scheduling-Strategien kombinieren $\Longrightarrow$ Multilevel-Scheduling
- Einfachste Realisierung: **Statisches Multilevel-Scheduling**
  - Für jede Priorität gibt es eine Taskliste mit eigener Scheduling-Strategie.
  - Die Teillisten haben meist unterschiedliche Prioritäten oder verschiedene Zeitmultiplexe. (z. B. 80%:20% oder 60%:30%:10%)
  - Ermöglicht Trennung von zeitkritischen und zeitunkritischen Tasks

Statisches Multilevel-Scheduling: Illustration



Statisches Multilevel-Scheduling: Beispiel

Priorität	Klasse	Scheduling Strategie
4+	Echtzeit-Tasks	Strict Priority Scheduling
3	Interaktive-Tasks	Round Robin
2	I/O-Intensive-Tasks	Round Robin
1	Rechenintensive-Tasks	First Come First Served

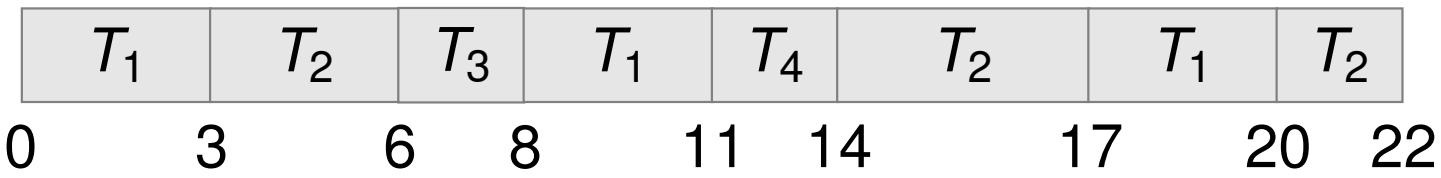
Multilevel-Feedback-Scheduling (MFS)

- Bei Mehrprogrammbetrieb und Mehrbenutzerbetrieb ist es unmöglich, die Rechenzeit im Voraus verlässlich zu prognostizieren
- Multilevel-Feedback-Scheduling arbeitet wieder mit mehreren Warteschlangen
- Innerhalb jeder Warteschlange wird Round Robin eingesetzt
  - Gibt ein Task die CPU freiwillig wieder ab, wird er wieder in die selbe Warteschlange eingereiht
  - Hat ein Task seine volle Zeitscheibe genutzt, kommt er in die nächsttiefere Warteschlange mit einer niedrigeren Priorität
- Neue Tasks werden schnell in eine Prioritätsklasse eingeordnet
- I/O-lastige Tasks werden bevorzugt
- **Unfair:** Ältere Tasks können verhungern

MFS: RR-Beispiel mit Quantum = 3

Task	Laufzeit	Ankunft
T_1	9 ms	0
T_2	8 ms	1
T_3	2 ms	2
T_4	3 ms	9

Ausführungsreihenfolge der Tasks (Gantt-Diagramm)



- Mittlere Lebensdauer: $(20 + 21 + 6 + 5)/4 = 13$ ms
- Mittlere Wartezeit: $(11 + 13 + 4 + 2)/4 = 7,5$ ms

Multi-Core Scheduling

- Hier ist nicht nur die Auswahl des Tasks, sondern auch des CPU-Kerns wichtig
- Alle Tasks einer Gruppe sollen auf einem CPU-Core abgearbeitet werden, um die Daten in den Registern und im Cache zu nutzen und somit unnötiges Verdrängen/Wiederherstellen von Daten zu vermeiden
- Gang-Scheduling: Paralleliserung von verteilten Anwendungen
  - Threads, die zum gleichen Tasks gehören und miteinander kommunizieren (sich synchronisieren), sollen gleichzeitig gestartet werden
  - Dadurch sollen die Wartezeiten verringert werden

Vergleichsmatrix

Strategie	**Präemptiv**	**Fair**
First Come First Served	Nein	Ja
Last Come First Served	Ja	Nein
Round Robin	Ja	Ja
Strict Priority Scheduling	Ja	Nein
Fair Share mit Round Robin	Ja	Ja
Statisches Multilevel-Scheduling	Ja	?
Multilevel-Feedback-Scheduling	Ja	Nein

Zusammenfassung

Nach diesem Kapitel sollten Sie . . .

- . . . wissen was die Aufgaben eines Dispatchers sind.
- . . . wissen was die Aufgaben eines Schedulers sind.
- . . . den Unterschied zwischen kooperativen und präemptiven Scheduling erklären können.
- . . . die folgenden Schedulingstrategien beherrschen:
  - First Come First Served (FCFS)
  - Last Come First Served (LCFS)
  - Round Robin (RR)
  - Strict Priority Scheduling (SPC)
  - Multilevel-Feedback-Scheduling (MFS)