Kapitel 2

Echtzeitbetriebssysteme

Aufgaben eines Echtzeitbetriebssystems

Karlsruher Institut für Technologie

Aufgaben eines Standardbetriebssystems:

- ***Taskverwaltung:*** Steuerung und Organisation der durchzuführenden Verarbeitungsprogramme, auch **Tasks** genannt. ⇒ Zuteilung des Prozessors an die Tasks.
- ***Betriebsmittelverwaltung:*** Zuteilung von Betriebsmittel an Rechenprozesse. Dies beinhaltet im wesentlichen:
  - Speicherverwaltung: Zuteilung von Speicher
  - IO-Verwaltung: Zuteilung von IO-Geräten
- ***Interprozesskommunikation:*** Die Kommunikation zwischen den Tasks
- ***Synchronisation:*** Zeitliche Koordination der Tasks
- ***Schutzmaßnahmen:*** Der Schutz der Betriebsmittel vor unberechtigten Zugriffen durch Tasks

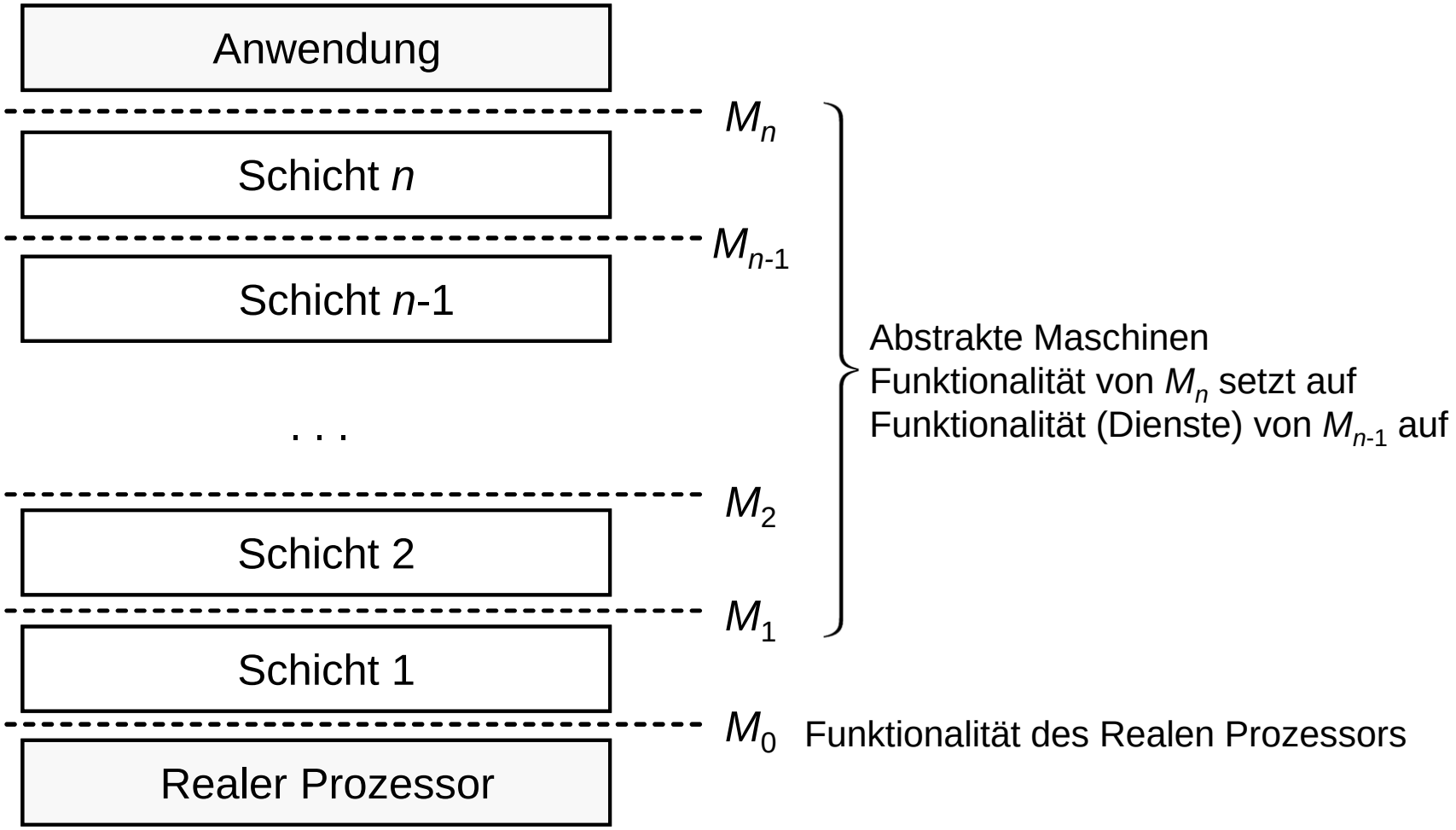
Zusätzliche Aufgaben eines Echtzeitbetriebssystems:

- Wahrung der ***Rechtzeitigkeit und Gleichzeitigkeit***
- Wahrung der ***Verfügbarkeit***

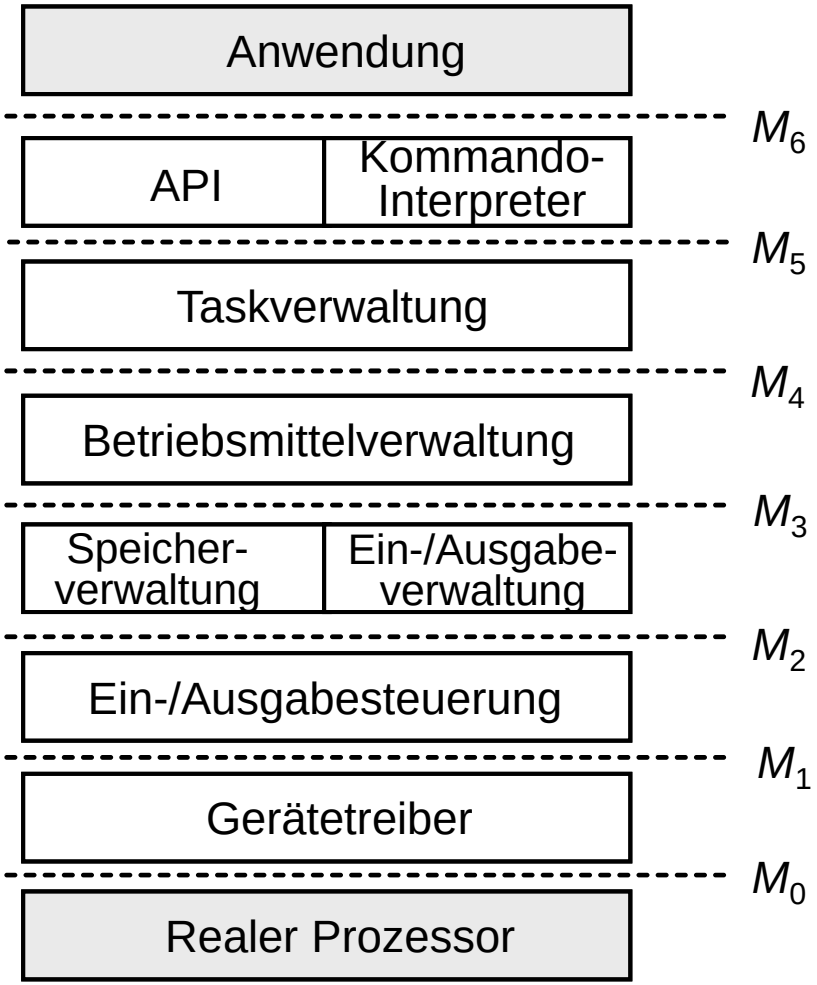
Non-exhaustive List of RTOSs

µC/OS, R&D Books Miller Freeman Inc.; **µITRON**, TRON Association - ITRON Technical Committee; **AIX**, IBM; **AMX**, KADAK Products Ltd; **Ariel**, Microware Systems Corporation; **ARTOS**, Locamation; **ASP6x**, DNA Enterprises, Inc.; **Brainstorm Object eXecutive**, Brainstorm Engineering Company; **Byte-BOS**, Byte-BOS Integrated Systems; **C Executive**, JMI Software Systems Inc.; **Chimera**, The Robotics Institute Carnegie Mellon University; **ChorusOS**, Sun Microsystems; **CMX**, CMX Company; **CORTEX**, Australian Real Time Embedded Systems (ARTESYS); **CREEM**, GOOFEE Systems; **CRTX**, StarCom; **DDC-I Ada Compiler Systems (DACS)**, DDC-I, Inc.; **Diamond**, 3L; **eCos**, Cygnus Solutions; **Embedded DOS 6-XL**, General Software, Inc.; **EOS**, Etnoteam S.p.A.; **ERCOS EK**, ETAS GmbH & Co.KG; **EUROS**, Dr. Kaneff Engineering Consultants; **Fusion OS**, Pacific Softworks; **Granada**, Ingenieursbureau B-ware; **Harmony Real-Time OS**, Institute for Information Technology, National Research Council of Canada; **Helios**, Perihelion Distributed Software; **HP-RT**, Hewlett-Packard; **Hyperkernel**, Imagination Systems, Inc.; **Inferno**, Lucent Technologies; **INTEGRITY**, Green Hills Software, Inc.; **INtime (real-time Windows NT), iRMX**, Radisys Corp.; **IRIX**, Silicon Graphics, Inc.; **iRMX III**, RadiSys Corporation; **ITS OS**, In Time Systems Corporation; **JBed**, Oberon microsystems, Inc.; **JOS**, JARP micro operating system; **Joshua**, David Moore; **LP-RTWin Toolkit**, LP Elektronik GmbH; **LP-VxWin**, LP Elektronik GmbH; **LynxOS**, Lynx Real-Time Systems; **MC/OS runtime environment**, Mercury Computer Systems, Inc.; **MotorWorks**, Wind River Systems Inc.; **MTEX**, Telenetworks; **NevOS**, Microprocessing Technologies; **Nucleus PLUS**, Accelerated Technology Inc.; **OS-9**, Microware Systems Corp.; **OS/Open**, IBM Microelectronics North American Regional Sales Office; **OSE**, Enea OSE Systems; **OSEK/VDX**, Universität Karlsruhe; **PDOS**, Eyring Corporation Systems; **PERC - Portable Executive for Reliable Control**, NewMonics; **pF/x**, Forth, Inc.; **PowerMAX OS**, Concurrent Computer Corporation; **Precise/MQX**, Precise Software Technologies Inc; **PRIM-OS**, SSE Czech und Matzner; **pSOS, pSOSystem**, Integrated Systems, Inc.; **PXROS**, HighTec EDV Systeme GmbH; **QNX**, QNX Software Systems, Ltd.; **QNX/Neutrino**, QNX Software Systems, Ltd.; **Real-time Extension (RTX) for Windows NT**, VenturCom, Inc.; **Real-Time Software**, Encore Real Time Computing Inc.; **REALTIME CRAFT**, TECSI; **Realtime ETS Kernel**, Phar Lap Software, Inc.; **RMOS**, Siemens AG; **Roadrunner**, Cornfed Systems, Inc.; **RT-Linux**, New Mexico Tech; **RT-mach**, Carnegie Mellon University; **RTEK**, Motorola; **RTEMS**, OAR Corporation; **RTKernel-C**, On Time Informatik GmbH; **RTMX O/S**, RTMX Inc.; **RTOS-UH/PEARL**, Universität Hannover; **RTTarget-32**, On Time Informatik GmbH; **RTX-51, RTX-251, RTX-166**, Keil Software; **RTXC**, Embedded System Products, Inc.; **RTXDOS**, Technosoftware AG; **RxDOS**, Api Software; **Smx**, Micro Digital, Inc.; **SoftKernel**, Microdata Soft; **SORIX 386/486**, Siemens AG; **SPOX**, Spectron Microsystems, Inc.; **SunOS, Solaris**, Sun Microsystems, Inc.; **Supertask!**, U S Software; **SwiftOS**, Forth, Inc.; **ThreadX**, Express Logic, Inc.; **Tics**, Tics Realtime; **TNT Embedded Tool Suite**, Phar Lap Software, Inc.; **Tornado/VxWorks**, Wind River Systems Inc.; **TSX-32**, S&H Computer Systems, Inc.; **UNOS**, Charles River Data Systems, Inc; **velOSity**, Green Hills Software, Inc.; **VERSAdos Real-Time Operating System**, Linden Technologies, Inc.; **Virtuoso**, Eonic Systems; **VRTX**, Microtec Research; **Windows CE**, Microsoft Inc.; **XOS/IA-32**, Nexilis Communications Corp.; **XTAL**, Axe, Inc.

Ein Schichtenmodell für informationsverarbeitende Systeme



Schichtenmodell eines Betriebssystems (Makrokernbetriebssystem, monolithisch)



API (Application Program Interface)

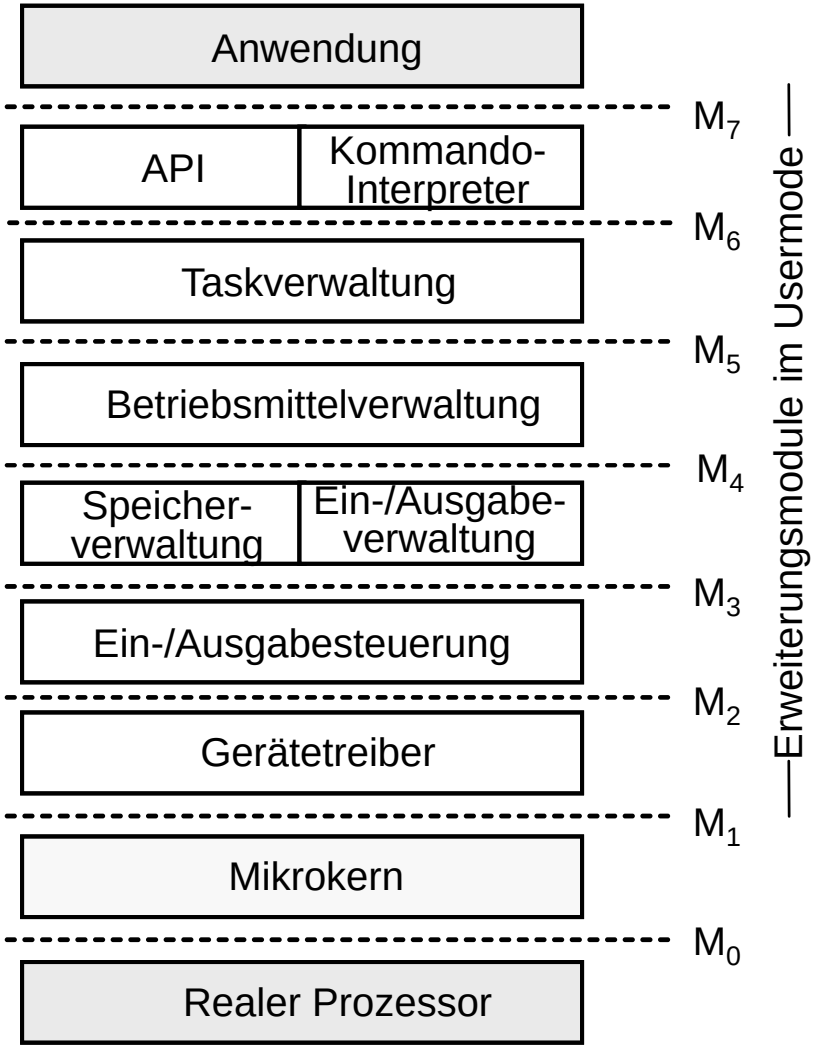
Zuteilung des Prozessors an die Tasks

Zuteilung von Ressourcen an die Tasks

Logische E/A Steuerung

Betriebssystemkern im Kernelmode, Usermode für Anwendung

Schichtenmodell eines Mikrokernbetriebssystems

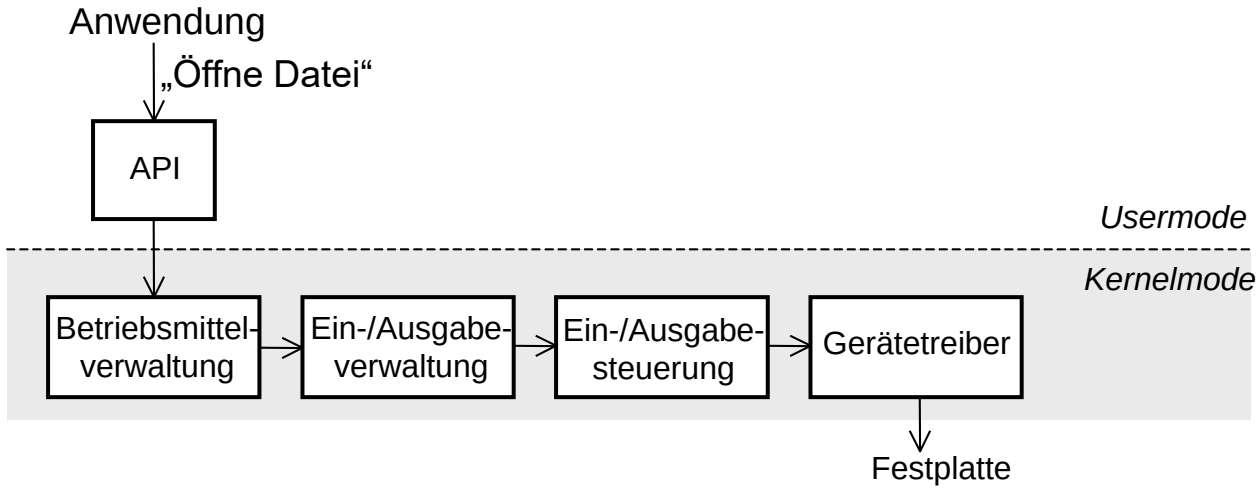


Vorteile Mikrokern:

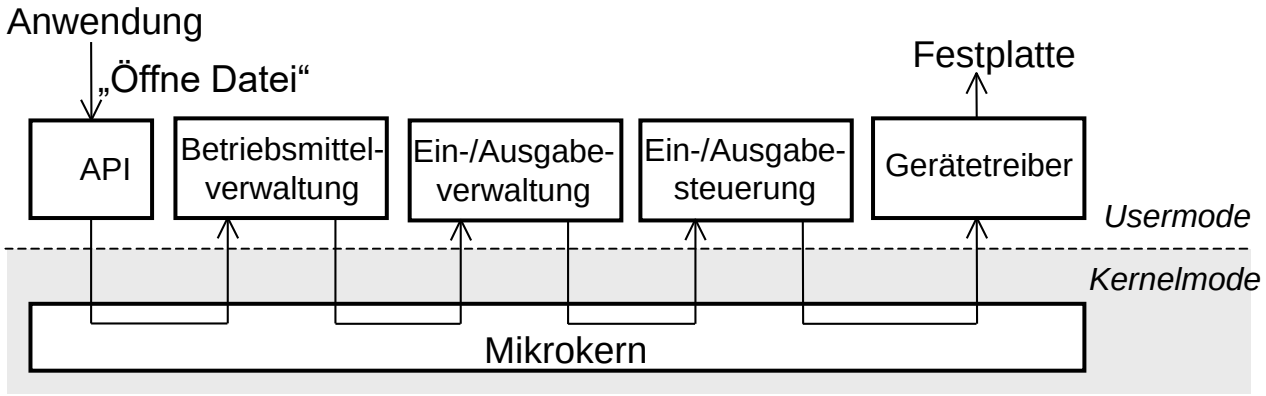
- sehr gut anpassbar an Aufgabe
- Hohe Skalierbarkeit durch Erweiterungsmodule
- Einfache Portierbarkeit
- Preemptiver Kern
- Kurze kritische Bereiche

- die **Interprozesskommunikation**
- die **Synchronisation**
- die **elementaren Funktionen** zur **Taskverwaltung**, dies sind i.A.
  - das Einrichten einer Task,
  - das Beenden einer Task,
  - das Aktivieren einer Task, und
  - das Blockieren einer Task

Usermode- und Kernelmode-Wechsel bei Makro- und Mikrokernbetriebssystem



a: Makrokernbetriebsystem (monolithisch)



b: Mikrokernbetriebsystem

Taskverwaltung: Tasks und Threads

Karlsruher Institut für Technologie

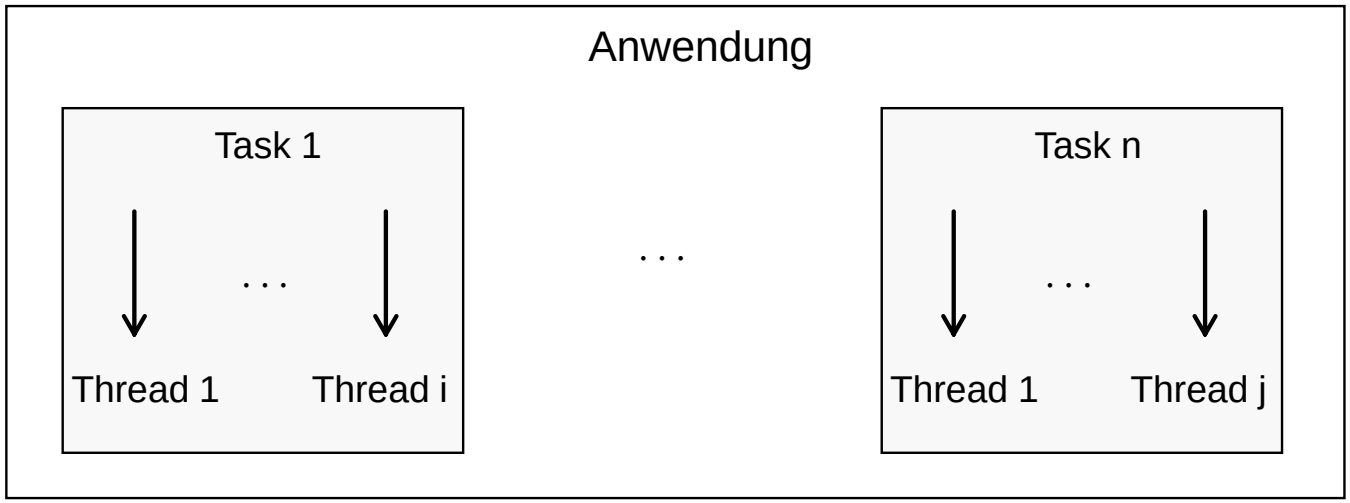
Task (schwergewichtiger Prozess)

- eigene Variablen und Betriebsmittel
- von den anderen Tasks abgeschirmt
- Eigener Adressraum
- Kommunikation nur über Interprozesskommunikation
- Task-Umschaltung bis zu mehrere 1000 Prozessortakte wegen Rett- und Ladeoperation.

Thread (leichtgewichtiger Prozess)

- innerhalb einer Task. Benutzt Variablen und Betriebsmittel der Task
- Gemeinsamer Adressraum aller Threads innerhalb einer Task
- Kommunikation über beliebige globale Variable.
- Sehr schnelle Kommunikation und Thread-Umschaltung (wenige Takte).
- Wenig Schutz zwischen Threads.

Taskverwaltung: Tasks und Threads



Aus Sicht der Betriebszustände, der Zeitparameter, des Schedulings und der Synchronisation sind Thread und Task gleich.

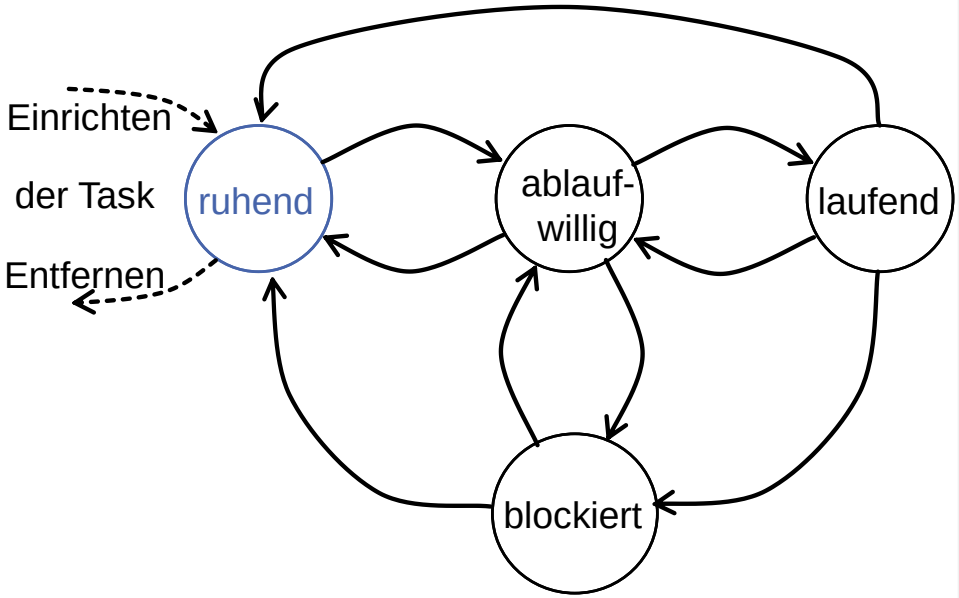
Taskzustände/Threadzustände

- ***Ruhend*** (*dormant*)
  Die Task ist im System vorhanden, aber momentan nicht ablaufbereit, da Zeitbedingungen oder andere Voraussetzungen (noch) nicht erfüllt sind.

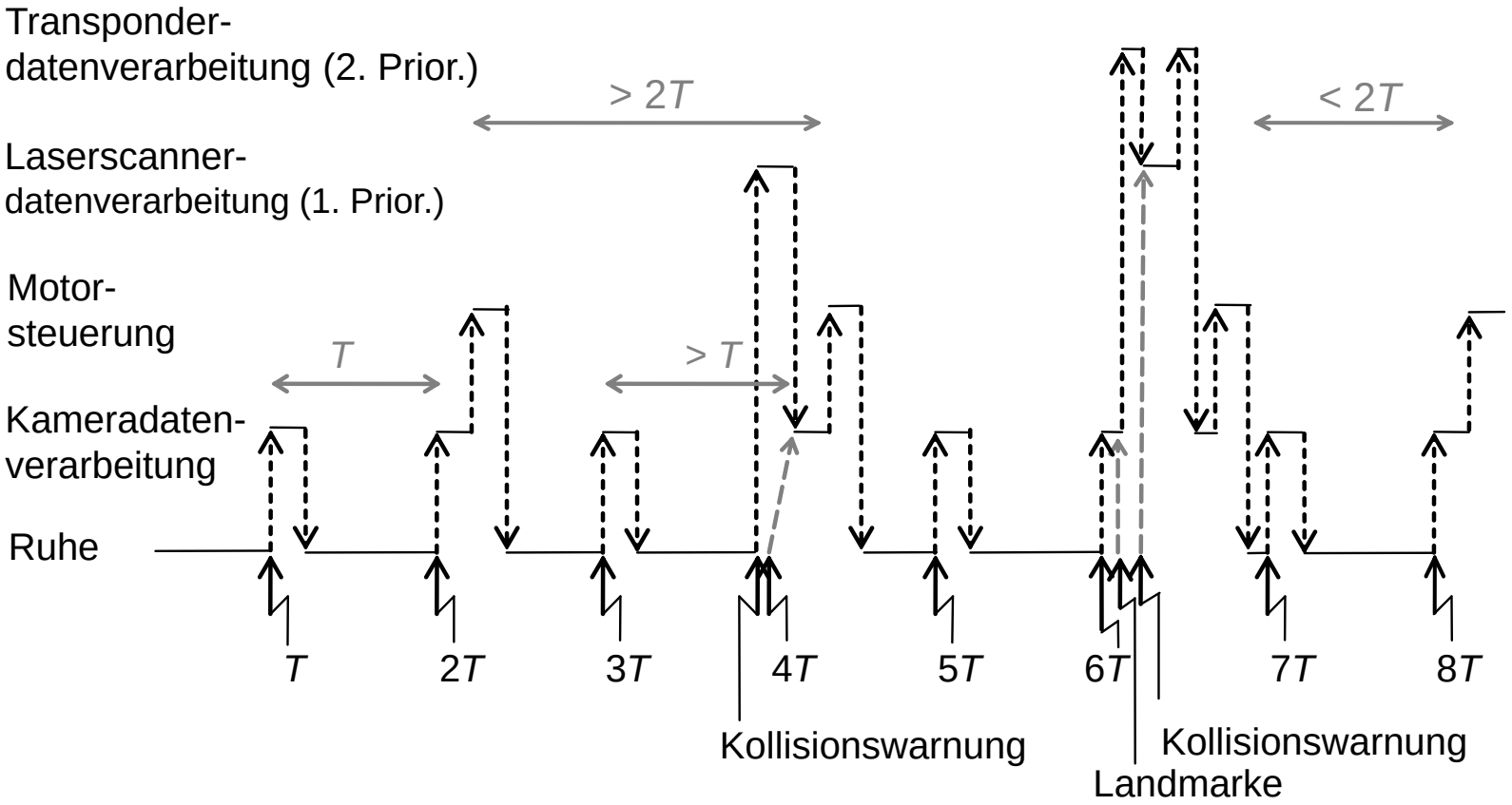
- ***Ablaufwillig*** (*runnable*)
  Die Task ist bereit, alle Zeitbedingungen oder andere Voraussetzungen zum Ablauf sind erfüllt, die Betriebsmittel sind zugeteilt, lediglich die Zuteilung des Prozessors durch die Taskverwaltung steht noch aus.

- ***Laufend*** (*running*)
  Die Task wird auf dem Prozessor ausgeführt. Bei einem Einprozessorsystem kann nur eine Task in diesem Zustand sein, bei einem Mehrprozessorsystem kann dieser Zustand von mehreren Tasks gleichzeitig angenommen werden.

- ***Blockiert*** (*suspended*, *blocked*)
  Die Task wartet auf das Eintreffen eines Ereignisses (z.B. einen Eingabewert, eine Interprozesskommunikation) oder das Freiwerden eines Betriebsmittels (Synchronisation).

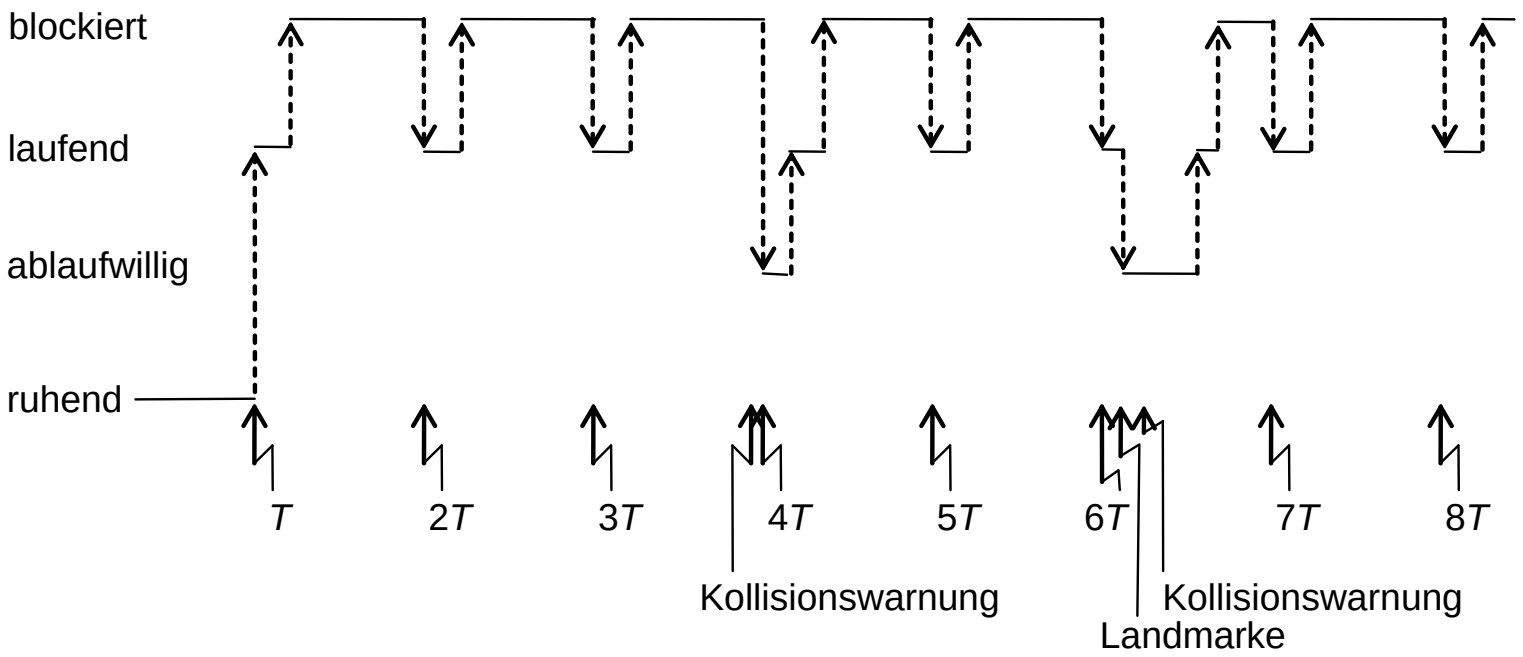


Beispiel: zeitlicher Ablauf der FTS-Steuerung

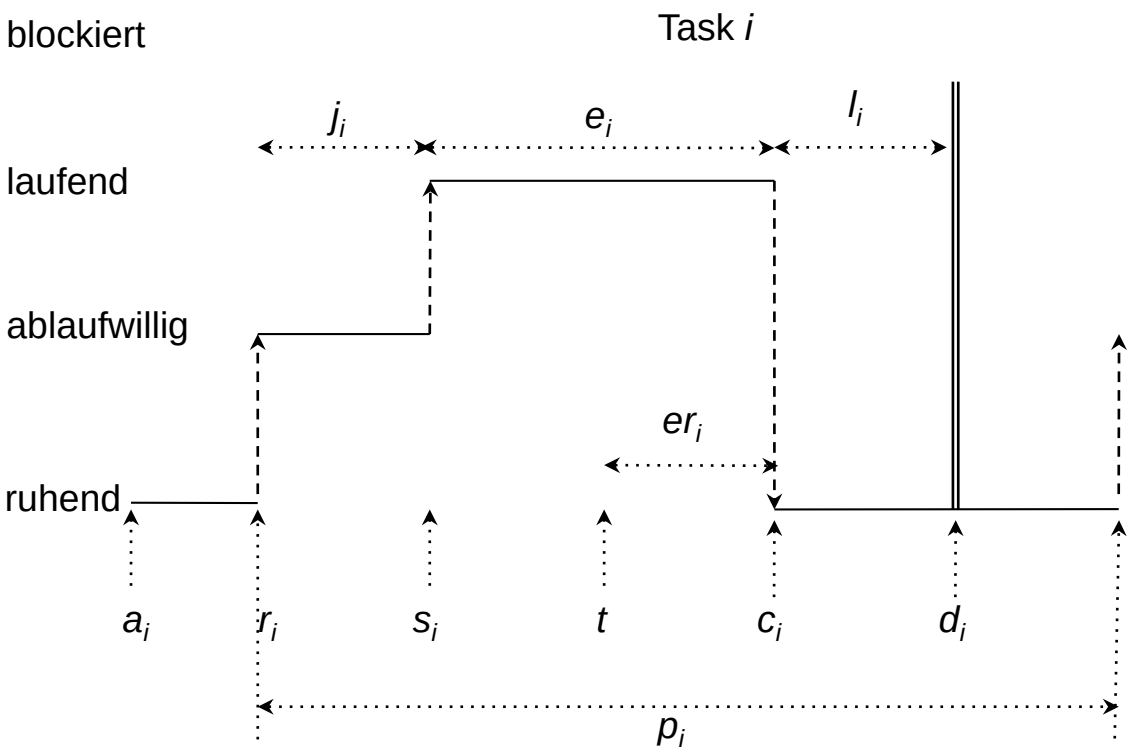


Periodendauer Kameradatenverarbeitung: T
Periodendauer Motorsteuerung: $2T$

Zustände der Task Kameradatenverarbeitung



Wesentliche Zeitparameter einer Echtzeittask



t: aktueller Zeitpunkt

a_i: Ankunftszeit (*Arrival Time*)

r_i: Anforderungszeit (*Request Time*)

s_i: Startzeit (*Start Time*)

j_i: Reaktionszeit
(*Reaction Time, Release Jitter*)

e_i: Ausführungszeit (*Execution Time*)

er_i: Restausführungszeit
(*Remaining Execution Time*)

c_i: Beendigungszeit (*Completion Time*)

d_i: Zeitschranke (*Deadline*)

p_i: Periode (*Period*)

l_i: Spielraum (*Laxity,* $l_i = d_i - (t + er_i)$)

Echtzeitscheduling

Aufgabe eines ***Echtzeitschedulers***

Aufteilung des Prozessors zwischen allen ablaufwilligen Tasks derart, dass – sofern möglich – alle Zeitbedingungen eingehalten werden.

Fragen zur Beurteilung eines Echtzeitschedulingverfahrens

- Existiert für ein gegebenes Taskset überhaupt ein **Schedule**, das alle Zeitbedingungen einhält?
- Findet das verwendete Schedulingverfahren diesen Schedule?
- Kann dieser Schedule in endlicher Zeit berechnet werden?

Ein Schedulingverfahren heißt **optimal**, wenn es einen Schedule findet, sofern dieser existiert

Analyse des Echtzeitverhaltens

Scheduling Analysis oder Feasibility Analysis

Prozessorauslastung (*Processor Demand*).

$$H = \frac{\text{Benötigte Prozessorzeit}}{\text{Verfügbare Prozessorzeit}}$$

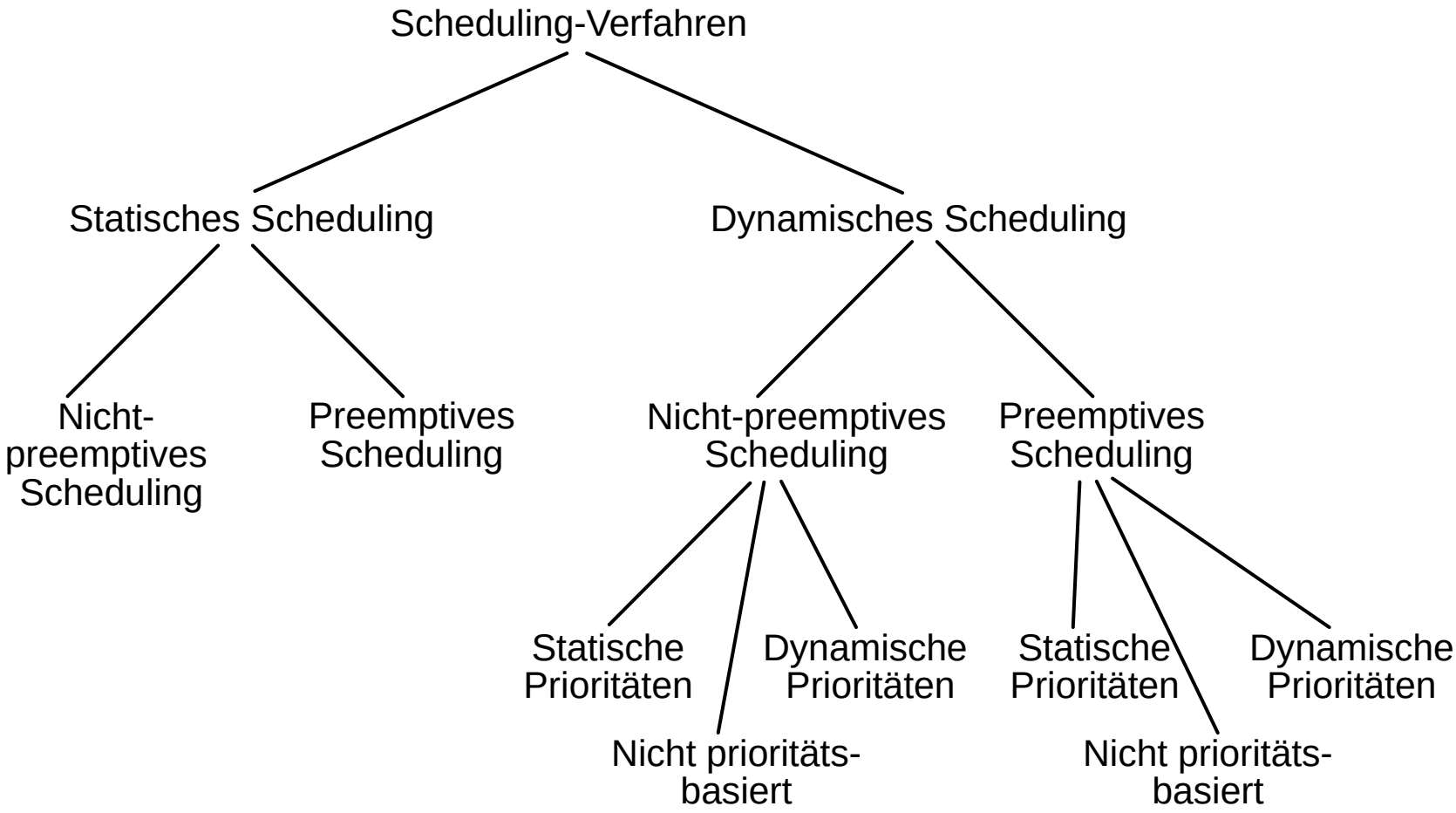
Prozessorauslastung auf einem Einprozessorsystem für ein Taskset aus n periodischen Tasks mit Zeitschranke = Periode:

$$H = \sum_{i=1}^{n} \frac{e_i}{p_i} \qquad \text{mit } e_i : \text{Ausführungszeit}, p_i : \text{Periodendauer von Task } i$$

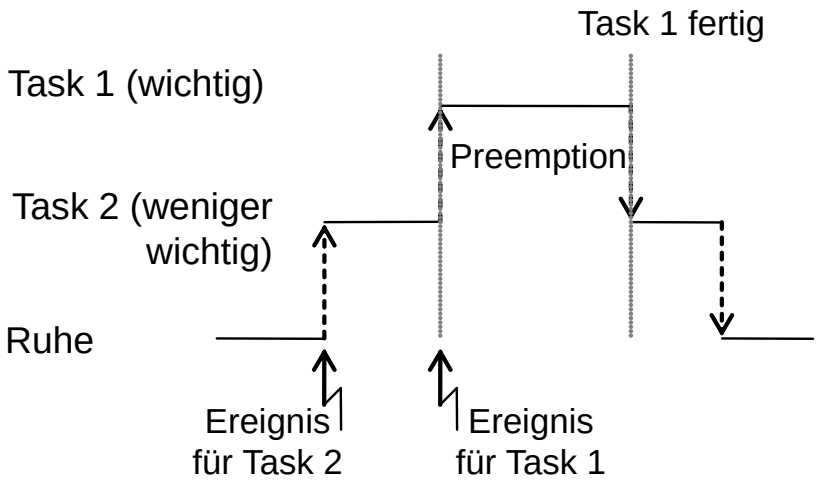
***H* > 100%:** kein Schedule
***H* ≤ 100%:** Schedule existiert

(wird jedoch je nach verwendetem Echtzeitschedulingverfahren nicht unbedingt gefunden)

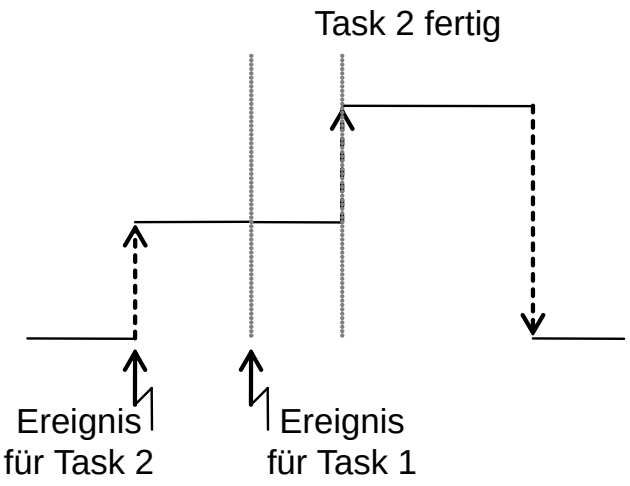
Klassifizierungsmerkmale von Scheduling-Verfahren



Preemptives und nicht-preemptives Scheduling



a) Preemptives Scheduling



b) Nicht-preemptives Scheduling

FIFO-Scheduling

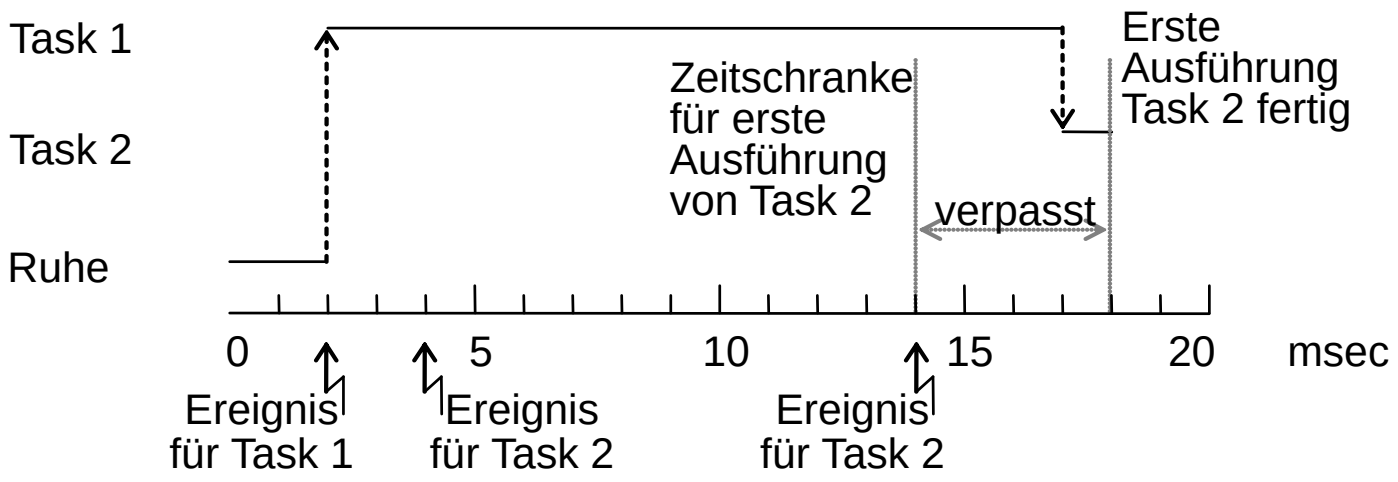
First In First Out Prinzip: wer zuerst kommt, erhält zuerst den Prozessor

Dynamisch, nicht-preemptiv, ohne Prioritäten

Wenig geeignet für Echtzeit, da keinerlei Zeitbedingungen eingehen

Beispiel.: Task 1: Periode p_1 = 150 msec Task 2: Periode p_2 = 10 msec
Ausführungszeit e_1 = 15 msec Ausführungszeit e_2 = 1 msec

H = 15 msec / 150 msec + 1 msec / 10 msec = 0,2 = 20%



Fixed-Priority-Scheduling

Jede Task erhält eine feste Priorität

Dynamisches Scheduling, statische Prioritäten

2 Varianten:

- ***Fixed-Priority-Preemptive-Scheduling (FPP)***
- ***Fixed Priority-Non-Preemptive Scheduling (FPN)***

Zuordnung der Prioritäten zu den Tasks ist entscheidend für das Echtzeitverhalten

Rate-Monotonic-Scheduling (RMS) für periodisch auszuführende Tasks

Gibt eine Regel der Prioritätenzuordnung für **periodische Tasks** an:

Es werden den periodisch auszuführenden Tasks Prioritäten umgekehrt proportional zu ihren Periodendauern zugewiesen:

$PR_i \sim 1 / p_i$ mit p_i: Periodendauer der Task i, PR_i: Priorität der Task i

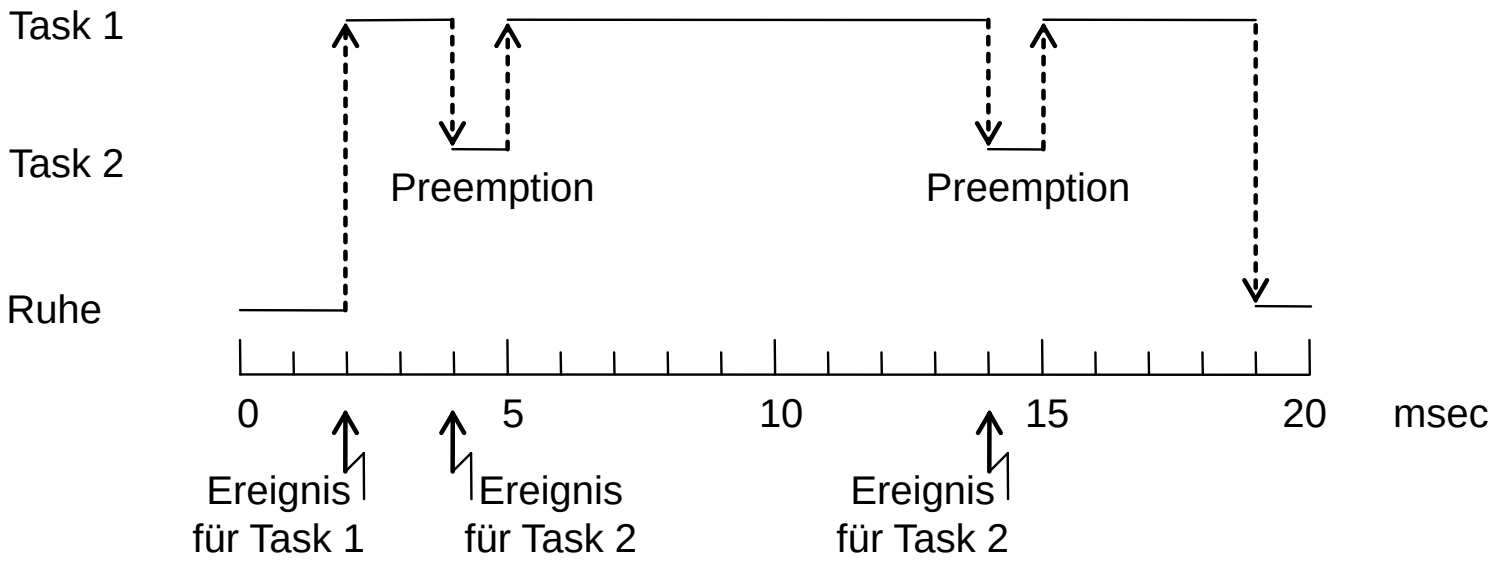
- es wird preemptives Scheduling verwendet,
- die Periodendauer p_i ist konstant,
- die Zeitschranke d_i ist gleich der Periodendauer p_i,
- die Ausführungszeit e_i ist konstant und bekannt, und
- die Tasks sind voneinander unabhängig, d.h. sie blockieren sich nicht gegenseitig z.B. durch Synchronisation
- Als Deadlines werden die Periodendauer angenommen

Beispiel RMS bei nicht gleichzeitigem Auftreten der Ereignisse

Fixed-Priority-Preemptive-Scheduling und Prioritätenverteilung gemäß dem Rate-Monotonic-Prinzip

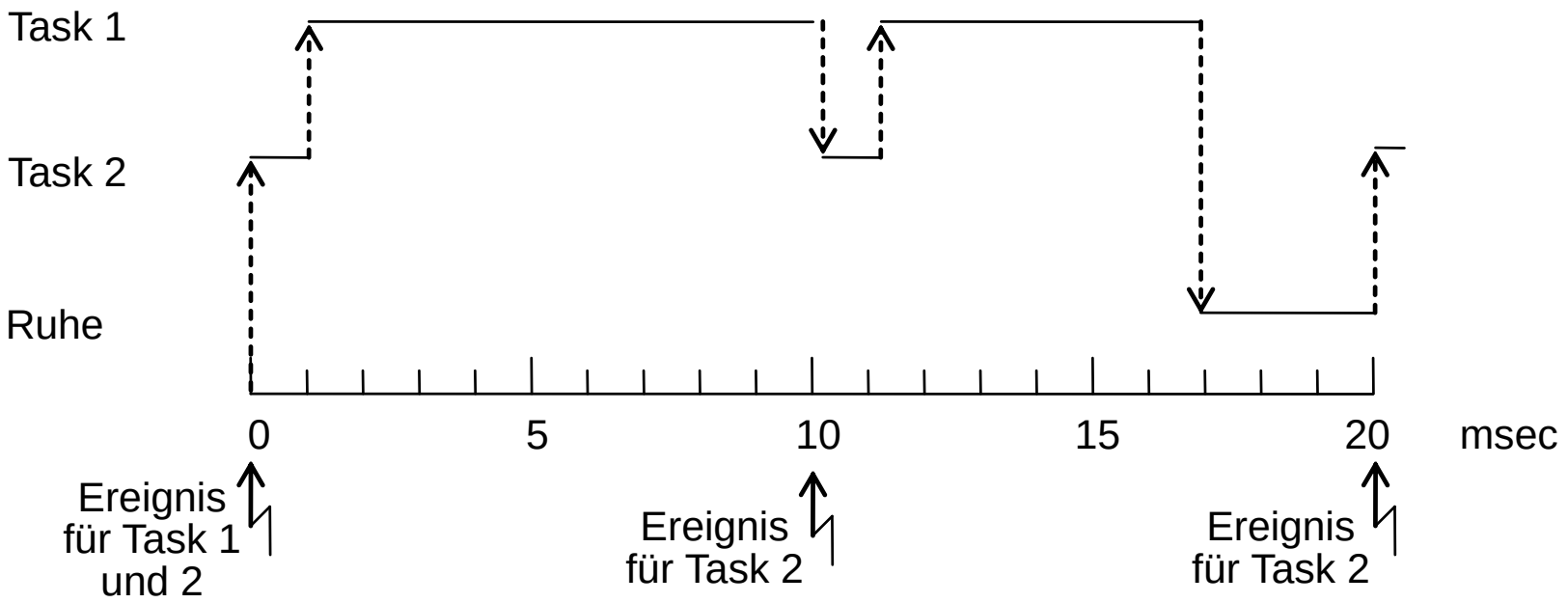
Task 1:	Periode	p_1 = 150 msec => niedere Priorität
	Ausführungszeit	e_1 = 15 msec
Task 2:	Periode	p_2 = 10 msec => hohe Priorität
	Ausführungszeit	e_2 = 1 msec

Deadlines sind die Periodendauern, H = 20%



Beispiel RMS bei gleichzeitigem Auftreten der Ereignisse

Task 1: p_1 = 150 msec, e_1 = 15 msec => niedere Priorität
Task 2: p_2 = 10 msec, e_2 = 1 msec => hohe Priorität



Zuteilung der Prioritäten gemäß dem Rate-Monotonic-Prinzip sind essentiell. Würde man keine Preemption zulassen oder die Prioritäten anders herum verteilen, so würde im obigen Beispiel Task 2 ihre Zeitschranken verletzen.

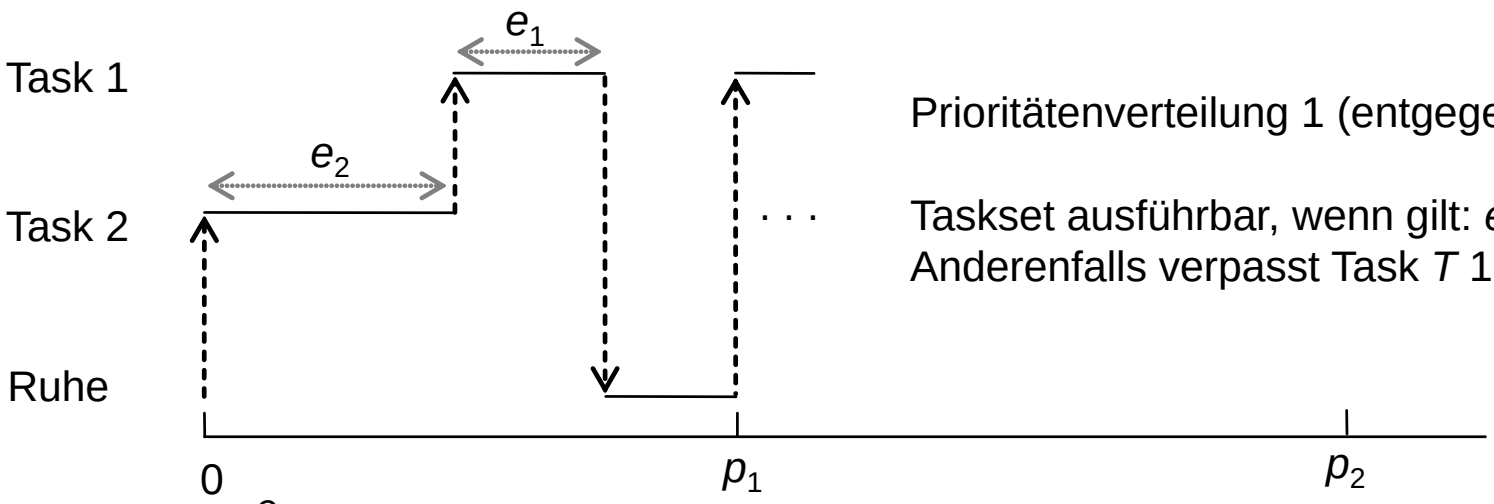
RMS = Optimale Prioritätenverteilung?

Behauptung:

Rate-Monotonic-Scheduling liefert bei festen Prioritäten und Preemption für periodische Tasks, bei denen die Zeitschranke identisch zur Periode ist, eine optimale Prioritätenverteilung.

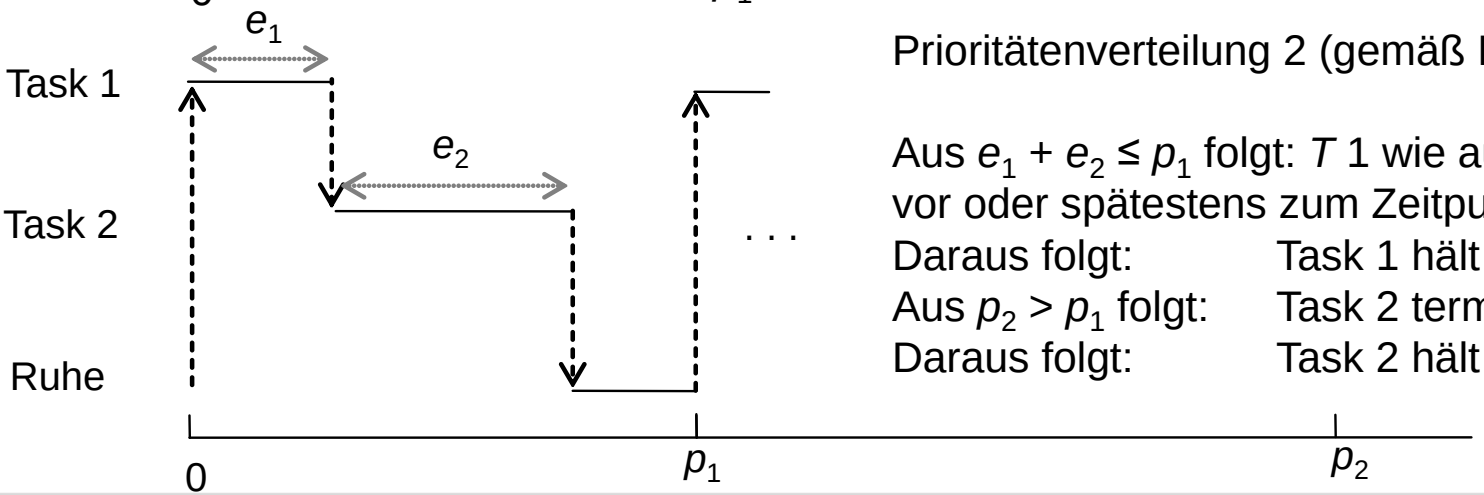
RMS = Optimale Prioritätenverteilung?

Prinzip des Beweises am Beispiel zweier Tasks, $p_2 > p_1$:



Prioritätenverteilung 1 (entgegen RMS)

Taskset ausführbar, wenn gilt: $e_1 + e_2 \leq p_1$
Anderenfalls verpasst Task T 1 ihre Zeitschranke



Prioritätenverteilung 2 (gemäß RMS)

Aus $e_1 + e_2 \leq p_1$ folgt: T 1 wie auch T 2 terminiert vor oder spätestens zum Zeitpunkt p_1
Daraus folgt: Task 1 hält ihre Zeitschranke
Aus $p_2 > p_1$ folgt: Task 2 terminiert vor p_2
Daraus folgt: Task 2 hält ihre Zeitschranke

RMS = Optimale Prioritätenverteilung?

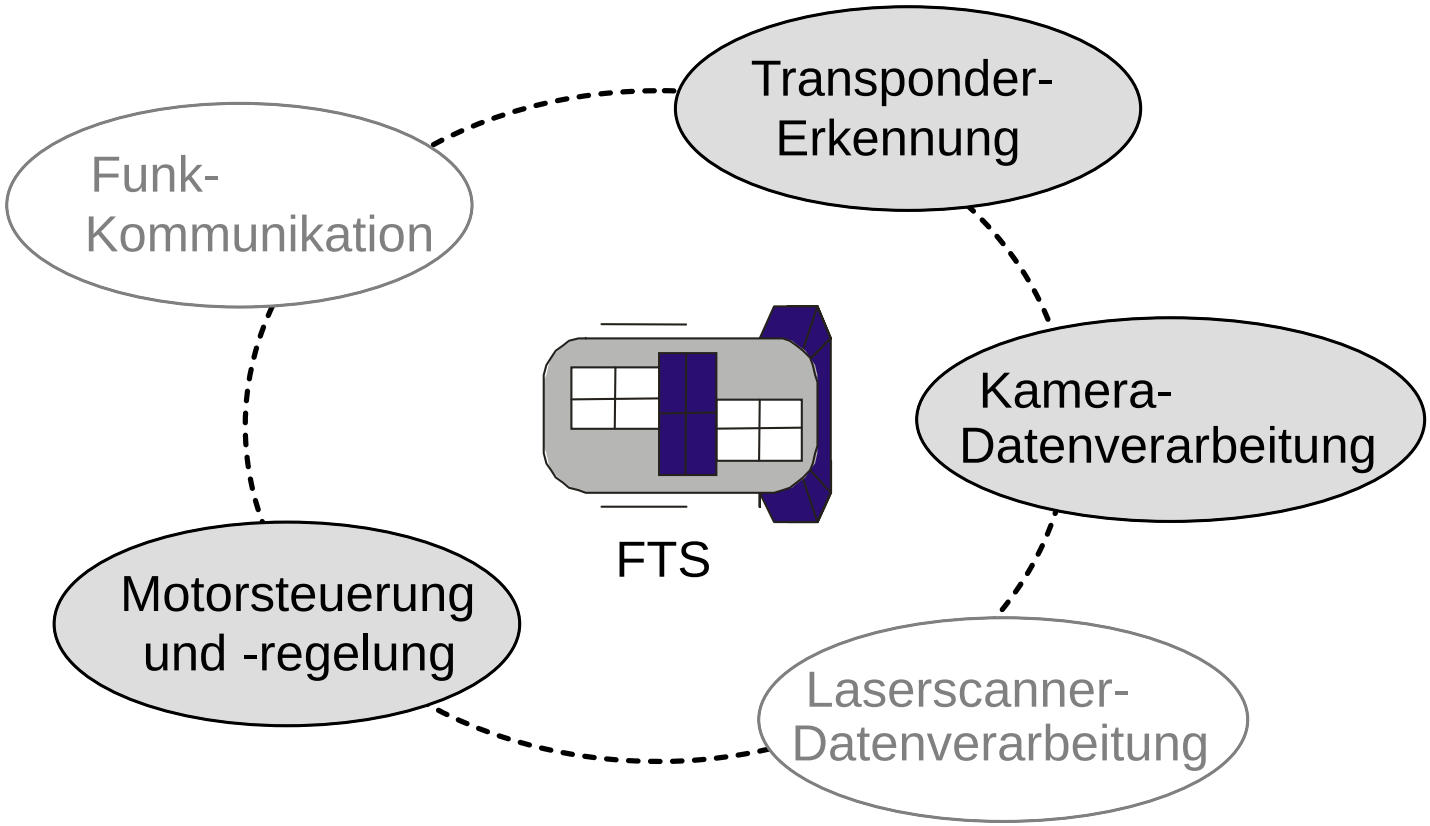
Dies gilt auch für alle späteren Aufrufe, da beim ersten Aufruf durch gleichzeitige Aktivierung von Task 1 und Task 2 der schlimmste Fall bereits abgedeckt ist.

Schlussfolgerung: ist Taskset mit 2 Tasks unter einer Prioritätenverteilung entgegen dem Rate-Monotonic-Prinzip (Variante 1) ausführbar, so ist es immer auch mit dem Rate-Monotonic-Prinzip (Variante 2) ausführbar.

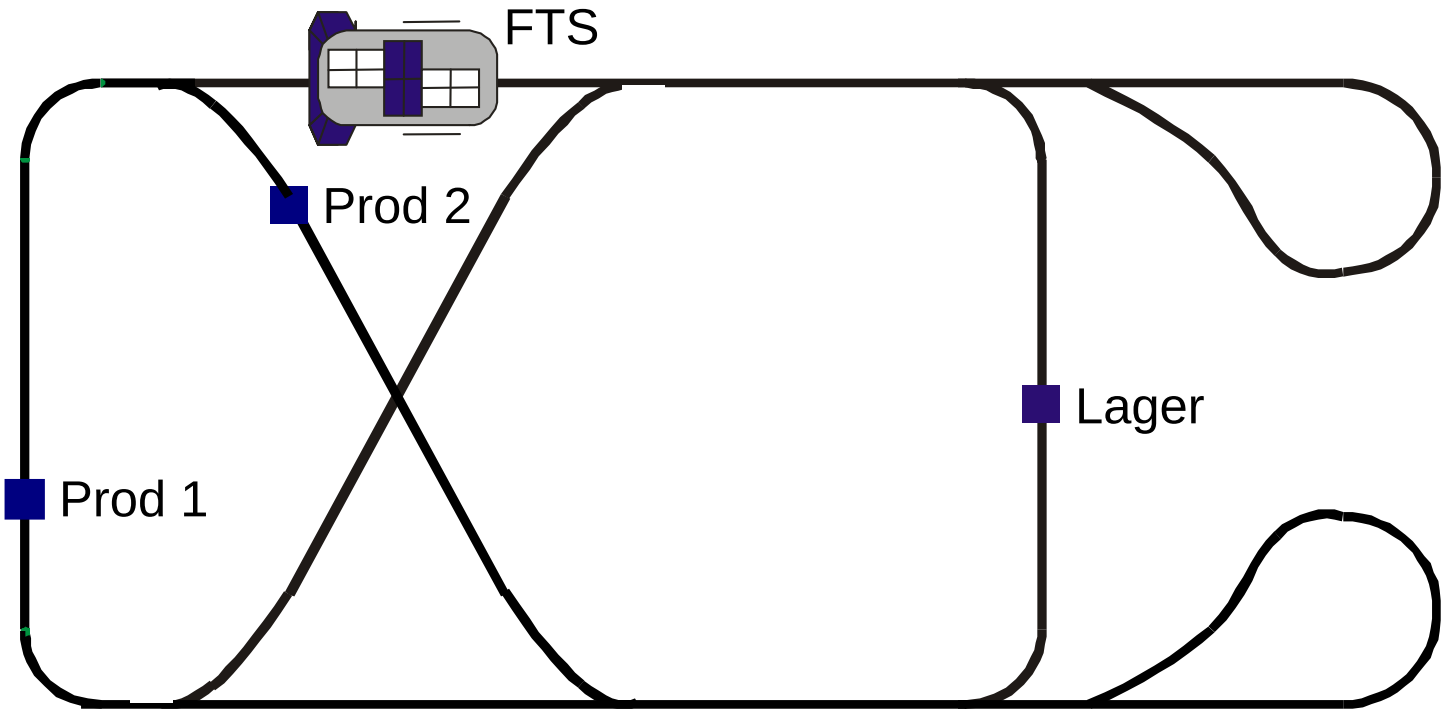
Für 2 Tasks ist Rate-Monotonic-Scheduling also eine optimale Prioritäten-zuteilung für feste Prioritäten.

Die Beweisführung lässt sich leicht von zwei auf eine beliebige Anzahl Tasks erweitern ([Liu Layland 1973])

FTS Beispiel mit drei Aufgaben



Daten aus industriellem FTS mit aufgeklebter Fahrspur und Landmarken



Fahrgeschwindigkeit: 0,65 m/sec, Positionsgenauigkeit 1 cm

*T*1: Kameradatenverarbeitung, $p_1 = d_1 = 10$ msec, $e_1 = 1$ msec
T2: Motorsteuerung, $p_2 = d_2 = 10$ msec, $e_2 = 5$ msec
*T*3: Transpondererkennung, $d_3 = 1$ cm : 0,65 m/sec = 15,4 msec, $e_3 = 5{,}5$ msec

Prozessorauslastung

$H = e_1 / p_1 + e_2 / p_2 + e_3 / p_3 =$
$= 1 \text{ msec}/10 \text{ msec} + 5 \text{ msec}/10 \text{ msec} + 5{,}5 \text{ msec}/15{,}4 \text{ msec} =$
$= 95{,}71\%$

(aperiodisches Transpondererereignis als schlimmster Fall mit seiner Zeitschranke periodisiert)

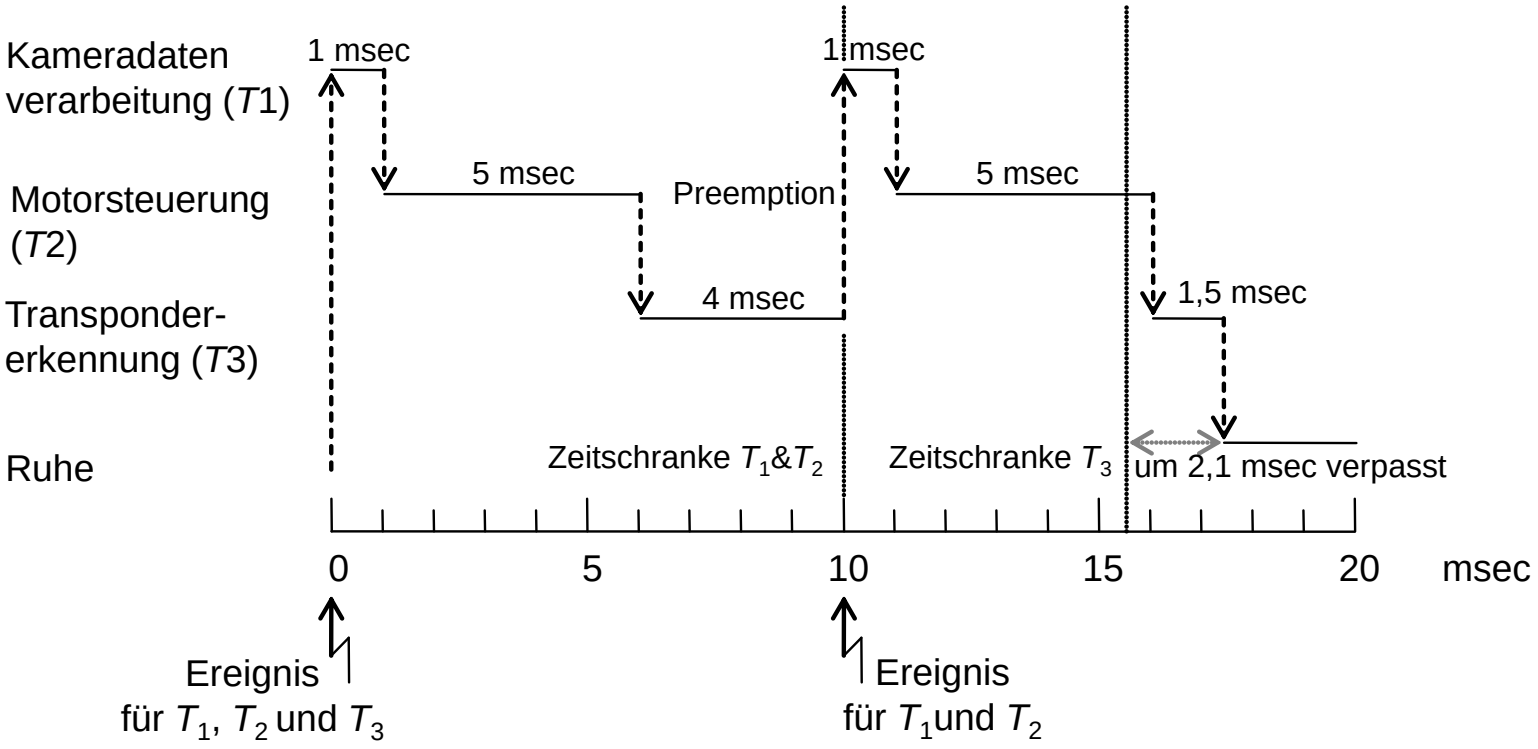
Prioritätenvergabe gemäß RMS:

*T*1 (Kameradatenverarbeitung):	hohe Priorität
*T*2 (Motorsteuerung):	mittlere Priorität.
*T*3 (Transpondererkennung):	niedrige Priorität

oder:

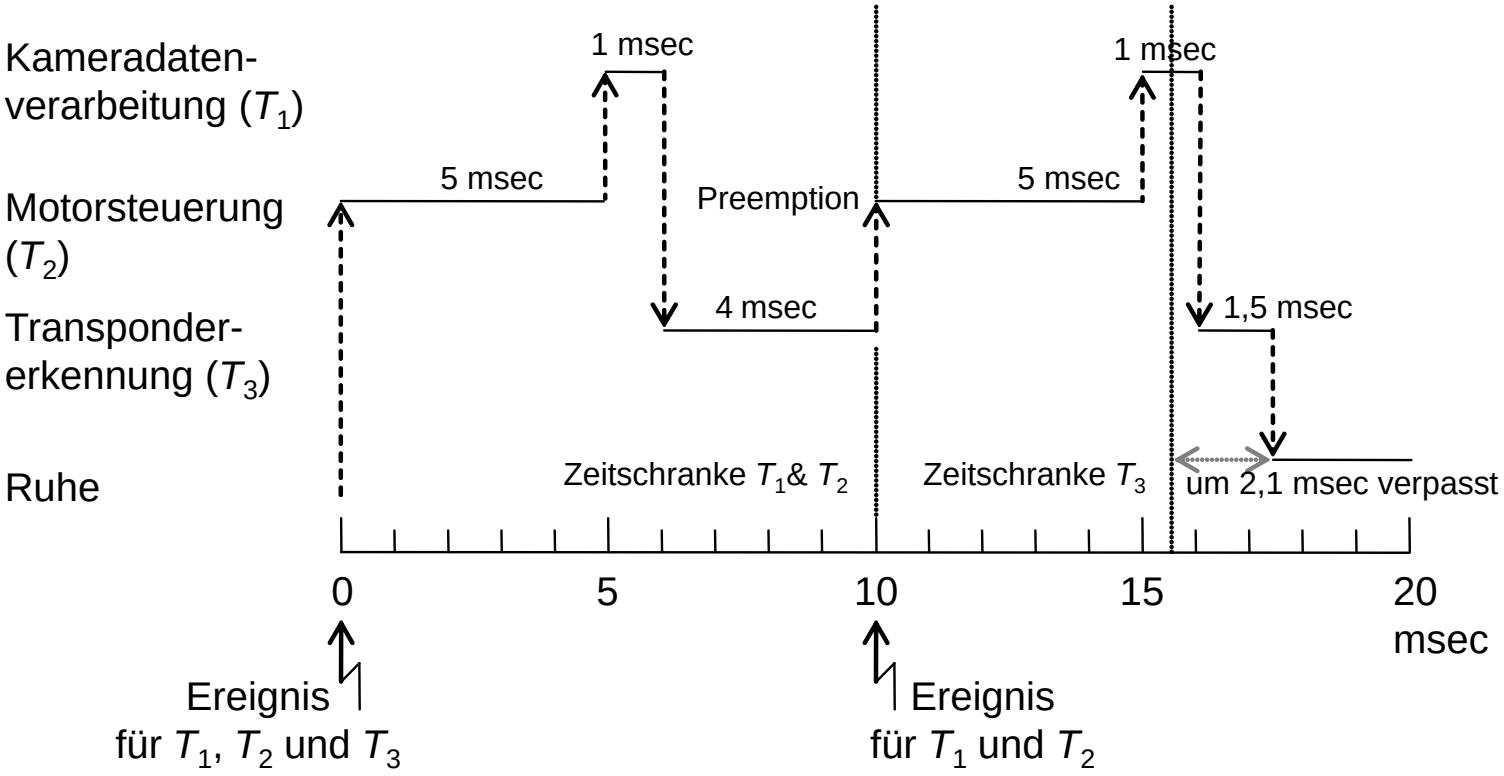
*T*1 (Kameradatenverarbeitung):	mittlere Priorität.
*T*2 (Motorsteuerung):	hohe Priorität
*T*3 (Transponderdatenerkennung):	niedrige Priorität

Ablauf des FTS Beispiels mit Priorität $T1 > T2 > T3$



Intelligente Prozess-automation und Robotik

Ablauf des FTS Beispiels mit Priorität $T2 > T1 > T3$



Ablauf des FTS Beispiels

Feste Prioritäten mit Preemption ist kein optimales Schedulingverfahren.

Obergrenze der Prozessorauslastung ist mathematisch ermittelbar, bis zu der immer ein ausführbarer Schedule für Rate-Monotonic-Scheduling mit Preemption gefunden wird.

$$H_{max} = n(2^{1/n} - 1), \quad n = \text{Anzahl der Tasks}$$

Bsp.: $H_{max} = 3\,(2^{1/3} - 1) = 0{,}78 = 78\%$

EDF-Scheduling

Bei ***Earliest-Deadline-First-Scheduling*** (EDF) erhält diejenige Task den Prozessor zugeteilt, die am nächsten an ihrer Zeitschranke (*Deadline*) ist.

=> Prioritäten ergeben sich automatisch und dynamisch aus den Zeitschranken.

Dynamisches Scheduling, dynamische Prioritäten, preemptiv oder nicht-preemptiv

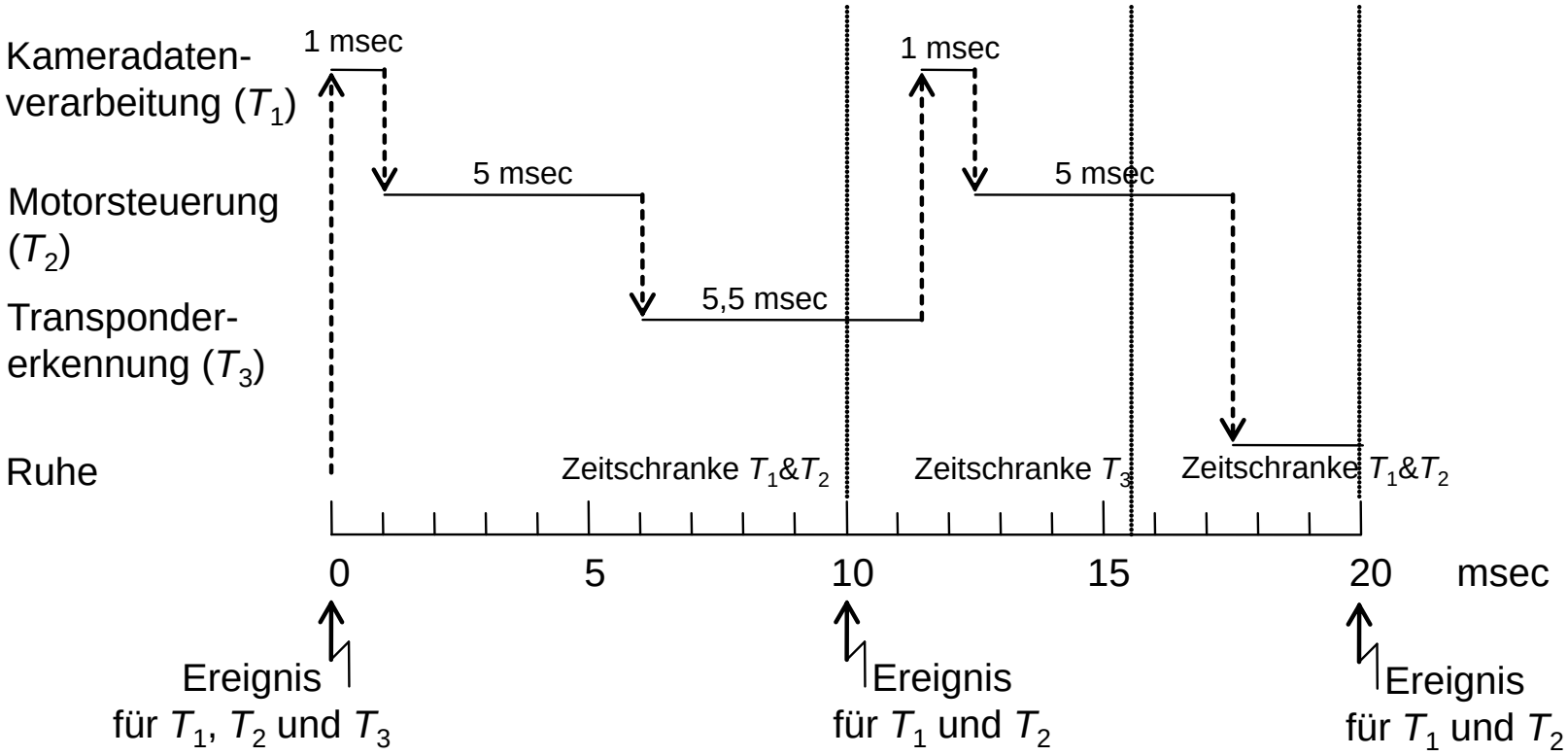
Preemptives EDF-Scheduling auf **Einprozessorsystemen** liefert **optimales Scheduling**

$H_{max} = 100\%$

⇒ EDF findet für periodische Tasks mit $d_i = p_i$ einen Schedule, wenn gilt:

$$\sum_{i=1}^{n} e_i / p_i \leq 1$$

Ablauf des FTS-Beispiels mit EDF-Scheduling



LLF-Scheduling

Bei ***Least-Laxity-First-Scheduling*** (LLF) erhält diejenige Task den Prozessor zugeteilt, die den geringsten Spielraum (*Laxity*) hat.

Dynamisches Scheduling, dynamische Prioritäten, preemptiv oder nicht-preemptiv

Preemptives **LLF** ist wie EDF ein **optimales Schedulingverfahren** auf **Einprozessorsystemen**,

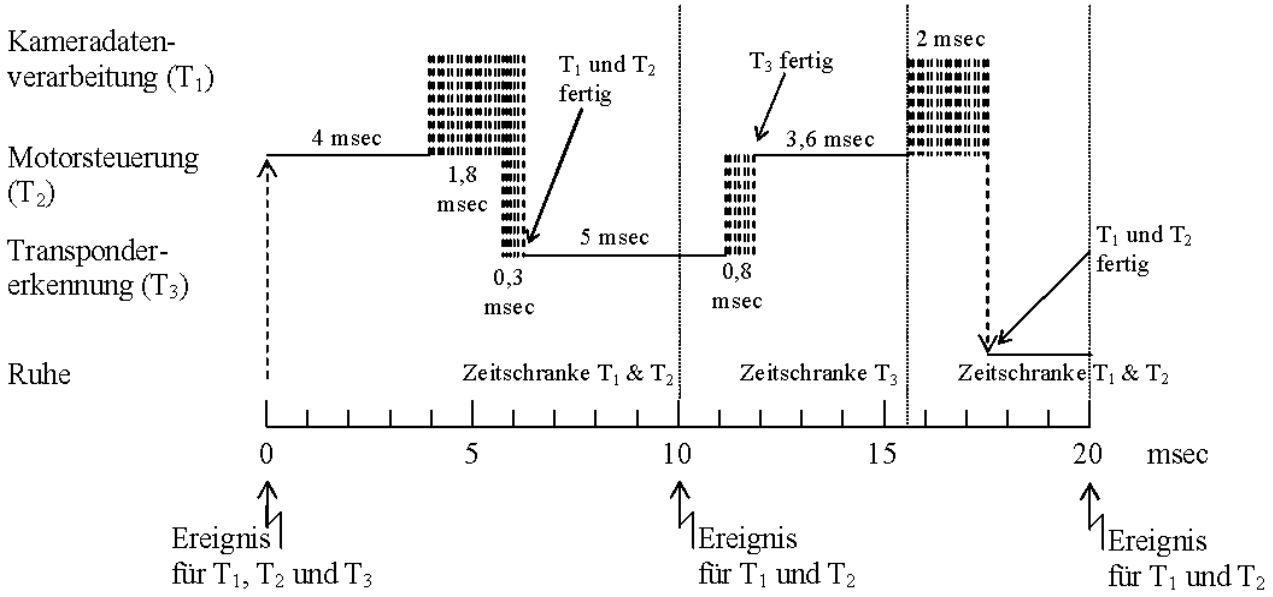
H_{max} = 100%.

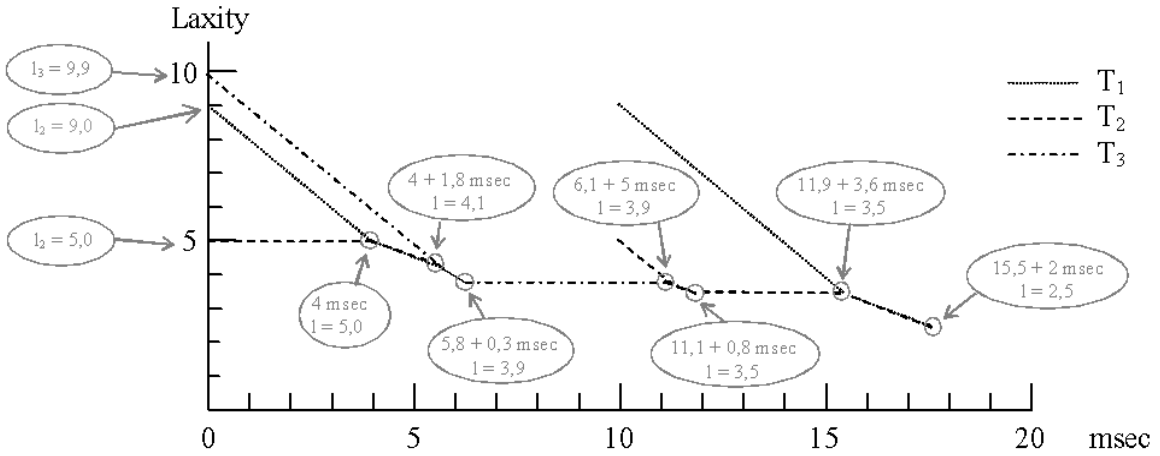
Für den Spielraum gilt $l_i = d_i - (t + er_i)$

⇒ LLF erzeugt eine drastisch höhere Anzahl an Taskwechseln als EDF

Aber besseres Verhalten als EDF bei Überlast

Ablauf des FTS Beispiels mit LLF-Scheduling





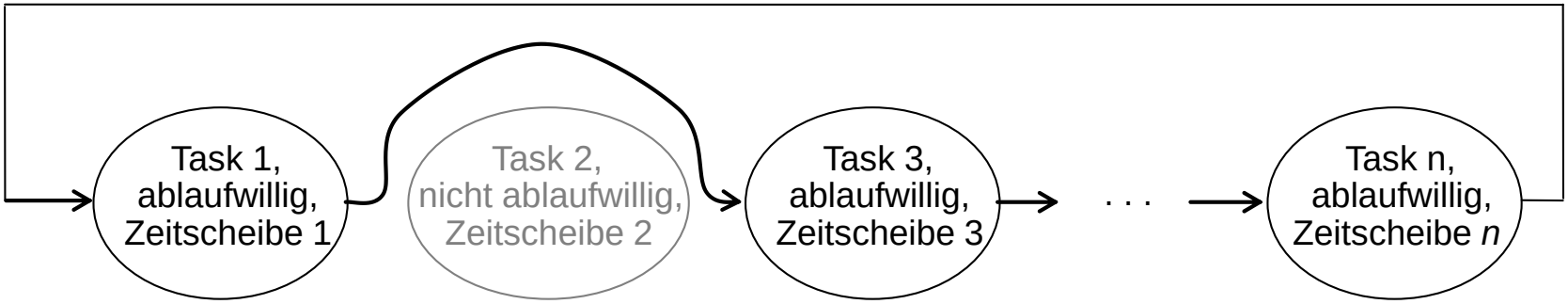
Time-Slice-Scheduling

Time-Slice-Scheduling (Zeitscheibenverfahren) ordnet jeder Task eine feste Zeitscheibe (*Time Slice*) zu.

Die Reihenfolge der Taskausführung entspricht hierbei der Reihenfolge der ablaufwilligen Tasks in der Taskverwaltungsliste des Betriebssystems.

Die Dauer der Zeitscheibe für eine Task kann individuell festgelegt werden.

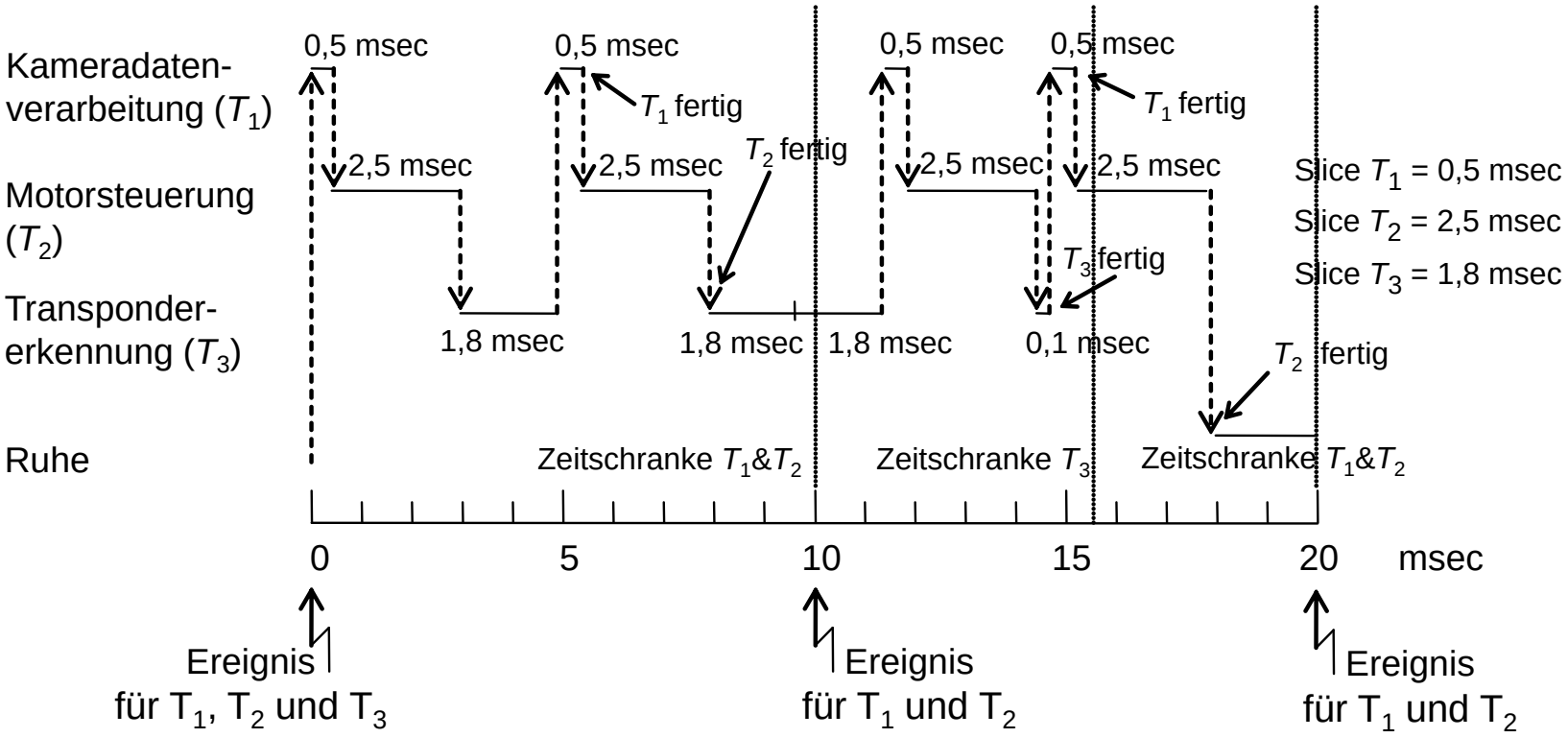
Preemptives dynamisches Scheduling ohne Prioritäten.



Einfaches Verfahren, bei feinkörnigen Zeitscheiben nahe an der Optimalität, aber Periodendauer abhängig von der Anzahl der Tasks

Ablauf des FTS-Beispiels mit Time-Slice-Scheduling

T_1: $H_1 = e_1 / p_1 = 1$ msec / 10 msec = 10%
T_2: $H_2 = e_2 / p_2 = 5$ msec / 10 msec = 50%
T_3: $H_3 = e_3 / p_3 = 5{,}5$ msec / 15,4 msec = 36% (35,714%)
Die Zeitscheiben wurden auf der Proportionalitätsbasis 1% ≡ 0,05 msec wie folgt gewählt:
T_1: Zeitscheibe 0,5 msec (~ 10%)
T_2: Zeitscheibe 2,5 msec (~ 50%)
T_3: Zeitscheibe 1,8 msec (~ 36%)



Guaranteed Percentage Scheduling

Guaranteed Percentage Scheduling (GP) ordnet jeder Task einen festen Prozentsatz der verfügbaren Prozessorleistung innerhalb einer Periode zu (virtueller Prozessor pro Task)

Preemptives dynamisches Schedulingverfahren ohne Prioritäten

Periodendauer unabhängig von der Anzahl der Tasks

Auf Einprozessorsystemen ein optimales Schedulingverfahren: $H_{max} = 100\%$

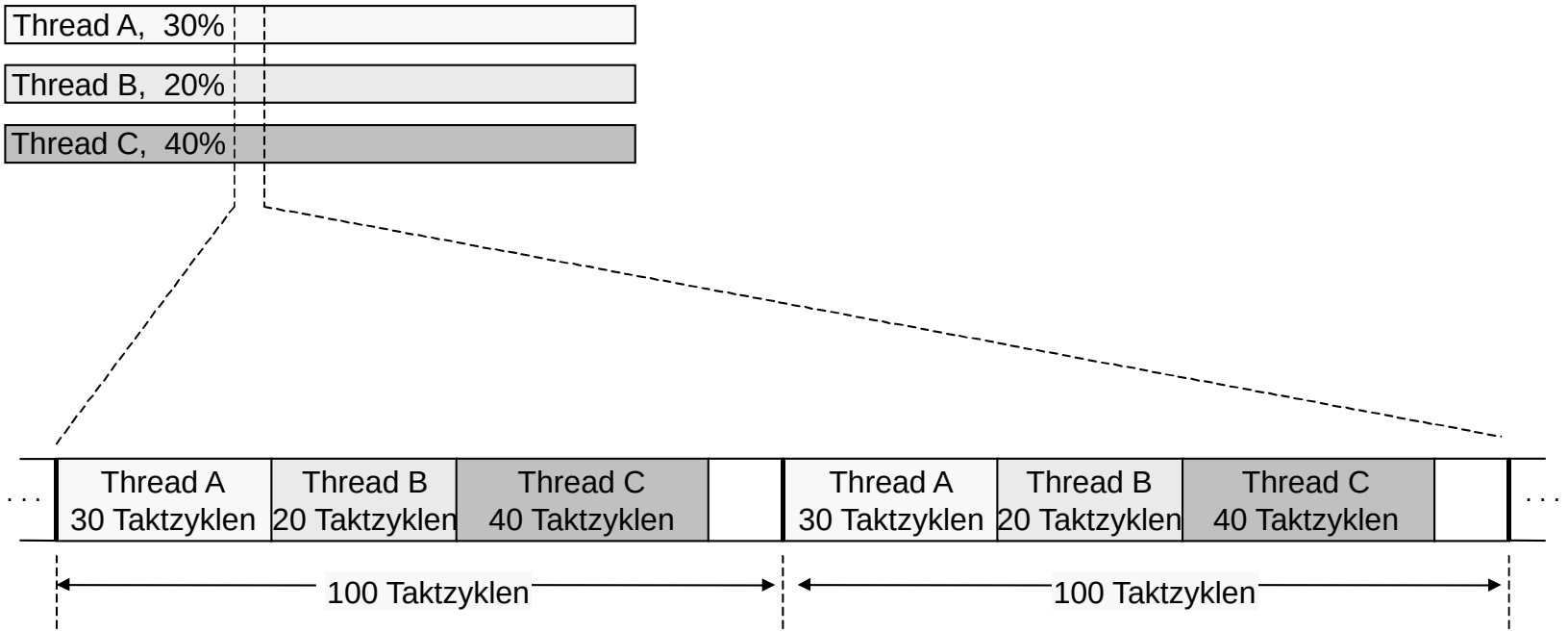
Bei periodischen Tasks werden die Zeitschranken erfüllt, wenn gilt: $\sum_{i=1}^{n} e_i / p_i \leq 1$

Viele Taskwechsel wie bei LLF

Für mehrfädige Prozessoren geeignet.

Guaranteed Percentage Scheduling auf dem Komodo Mikrocontroller

Mehrfädiger Java Mikrocontroller, GP Periode 100 Taktzyklen



Ablauf des FTS-Beispiels mit Guaranteed Percentage Scheduling

T_1: $H_1 = e_1 / p_1$ = 1 msec / 10 msec = 10%
T_2: $H_2 = e_2 / p_2$ = 5 msec / 10 msec = 50%
T_3: $H_3 = e_3 / p_3$ = 5,5 msec / 15,4 msec = 36% (35,714 %)
Hierdurch ergibt sich für jede Task eine Ausführungszeit, die ihrer Periode und damit ihrer Zeitschranke entspricht:
T_1: Ausführungszeit: 1 msec in 10 msec (10%)
T_2: Ausführungszeit: 5 msec in 10 msec (50%)
T_3: Ausführungszeit: 5,5 msec in 15,3 msec (36%)