Echtzeitsysteme

6. Echtzeitplanung (*Scheduling*)

Prof. Dr. Roland Dietrich

Echtzeitplanung (Scheduling)

- Problem:
  - N nebenläufige Tasks müssen sich K < N Ressourcen teilen
  - Insbesondere: N nebenläufige Tasks müssen auf einem Prozessor ausgeführt werden
  - Unproblematisch: wenn die N Tasks unabhängig sind und keine Zeitbedingungen erfüllen müssen
    - **Indeterminismus**: es gibt ***N!*** mögliche Reihenfolgen, ***N*** nebenläufige Tasks auf **einem** Prozessor auszuführen
    - alle Möglichkeiten führen zu einem korrekten Ergebnis
  - Problematisch:
    - Tasks können Zeitbedingungen erfüllen müssen (z.B. Deadlines, Perioden)
    - Tasks können durch Kommunikationsvorgänge blockiert sein
  - **Echtzeitplanung (*Real Time Scheduling*)**:
    - Die Reihenfolge, in der Tasks auf dem Prozessor ausgeführt werden, so festlegen, dass alle Zeitbedingungen erfüllt sind

    → Indeterminismus einschränken

Echtzeitplanung (Scheduling)

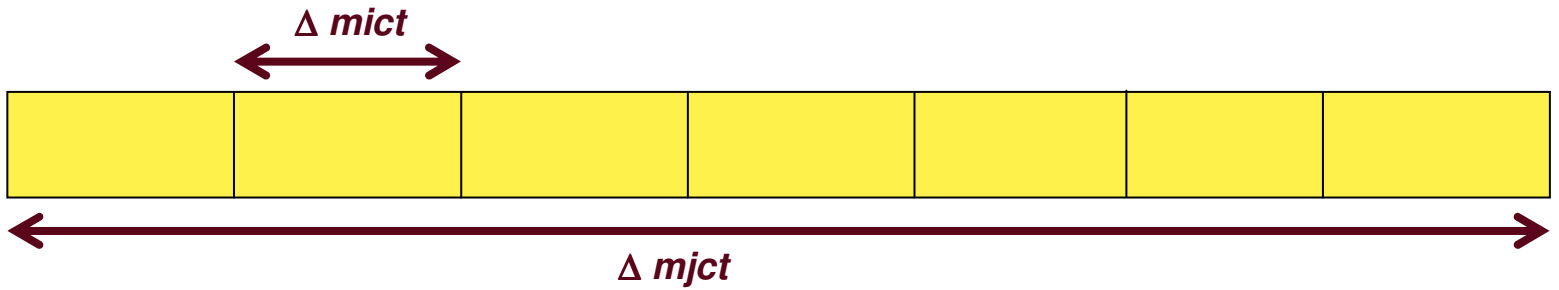
- Ein **Echtzeitplanungsverfahren** besteht aus zwei Komponenten
  - **Planungsalgorithmus**: legt eine Reihenfolge für die Nutzung des Prozessors fest (welche Task darf wann für wie lange den Prozessor nutzen)
  - Eine Methode, das **Rechenzeit-Verhalten** im ungünstigsten Fall unter Anwendung des Planungsalgorithmus **vorherzusagen**
    - Die Vorhersagbarkeit kann benutzt werden, die Realisierbarkeit der zeitlichen Anforderungen nachzuweisen
- Prioritäten: Jeder Task wird eine **Priorität** zugeteilt
  - Die Ausführung einer Task kann nach jeder Anweisung unterbrochen werden und der Prozessor einer anderen (mit höherer Priorität) zugeteilt werden (***preemptive multitasking***)
- Kategorien von Planungsverfahren:
    - **Statische Planung**: Die Prioritäten der Tasks liegen vor Ausführung fest
    - **Dynamische Planung**: Die Prioritäten der Tasks werden zur Laufzeit festgelegt (und können sich zur Laufzeit ändern)

Echtzeitplanung (Scheduling)

- **Definitionen**
  - Die Menge der Tasks: $\boldsymbol{T = \{t_1, \ldots, t_N\}}$
    - Vereinfachung: $\boldsymbol{T = \{ 1, \ldots, N \}}$
  - Task-Zeitparameter
    - Δ $\boldsymbol{t_i}$**:** Periodendauer der Task *ti*
    - Δ $\boldsymbol{e_i}$**:** Maximale Ausführungszeit für eine Ausführung von t_i
    - Δ $\boldsymbol{d_i}$**:** relative Deadline für eine Ausführung der Task t_i (Zeitraum vom Start bis zum spätesten Ende der Ausführung)
    - Δ $\boldsymbol{s_i}$ **:** Wartezeit bis zum Start (nach Periodenbeginn)
  - Ein **Plan (*scedule)*** ist eine Abbildung **s:** $\mathbb{N}_0 \rightarrow \{ 0 \} \cup T$ wobei
    - *s(i) = k:* im Zeitraum i wird der Prozess $\boldsymbol{t_k}$ auf dem Prozessor ausgeführt
    - *s(i) = 0:* im Zeitraum i wird keine Task auf dem Prozessor ausgeführt
    - Der Definition liegt ein diskretes Zeitmodell zugrunde
      - Zeit ist getaktet: 0,1,2,3,4,….,i,i+1,….
      - Δ i = [i,i+1)
  - Ein **brauchbarer Plan** ist ein Plan, der alle Zeitbedingungen erfüllt, wenn die Taskmenge nach diesem Plan ausgeführt wird

Zyklische Planungsverfahren

- Voraussetzung
  - Endliche Menge von periodischen Tasks: $\boldsymbol{T = \{t_1, \ldots, t_N\}}$
  - Die Tasks sind unabhängig (keine Kommunikation und Synchronisation, keine Prioritäten)
  - Die einzelnen Tasks werden atomar (ununterbrochen) ausgeführt
- Struktur eines zyklischen Plans
  - Ein äußerer Zyklus (major cycle) besteht aus mehreren gleich langen inneren Zyklen (minor cycle)
    - Δ ***mjct*** : Dauer eines äußeren Zyklus (***maj**or **c**ycle **t**ime*)
    - Δ ***mict*** : Dauer eines inneren Zyklus (***mi**nor **c**ycle **t**ime*)
    - Es gilt: $\Delta\ mjct = rnd * \Delta\ mict$ ($rnd \in \mathbb{N}$, "Runden")
  - Ein innerer Zyklus enthält eine oder mehrere Ausführungen von Tasks $t \in T$



Zyklische Planungsverfahren

- **Ziel**:
  - Die Ausführung der Tasks so auf die inneren Zyklen verteilen, dass
    - die einzelnen Perioden exakt eingehalten werden
    - die einzelnen Tasks möglichst jeweils komplett innerhalb eines inneren Zyklus ausgeführt werden

      → $\Delta\, mict \geq \Delta\, e_i$ für alle Tasks i
    - die Deadlines eingehalten werden können
    - Nach einem äußeren Zyklus alle Tasks mindestens einmal ausgeführt wurden

      → $\Delta\, mjct = \mathbf{kgV}(\Delta t_1 , \ldots , \Delta t_N)$

- *Beispiel:*

T	Δe	Δt
1	1	3
2	1	6
3	2	9

- Zeitangaben sind normiert auf "Zeiteinheiten", so dass die kürzeste Ausführungszeit einer Einheit entspricht
- Deadline ist jeweils das Periodenende, d.h.

  $\Delta d_i = \Delta t_i$ - ***Wartezeit***

Zyklische Planungsverfahren

- Programmschema [Zöbel 2008, Kap. 3.1.4]
  - Für die wiederholte Ausführung von **`rnd`** inneren Zyklen mit einer Zyklusdauer von **`dmict`** (= Δ *mict*)
  - Am Ende jeder "Runde" wird geprüft, ob es eine Fristverletzung gibt

```
cnt = 0;
t = gettime();
while(TRUE) {
  wait_until(t);
  switch (cnt) {
    ...
    case j: Ausführung des inneren Zyklus cnt ("Runde cnt")
    ...
  }
  cnt = (cnt+1) % rnd; // Nächste Runde
  t = t + dmict;
  if (gettime() > t) // Zu spät?
  { Verspätung behandeln }
} // end while
```

Zyklische Planungsverfahren

- *Bespiel*
  - Programm für die Ausführung der Tasks t_1, … t_3 (vgl. S. 6-6):

```
cnt =0;
t = gettime();
while (TRUE){
  wait_until(t);
  switch (cnt) {
    case 0: T1;T2;
    case 1: T1;T3;
    case 2: T1;T2;
    case 3: T1;T3;
    case 4: T1;T2;
    case 5: T1;
  }
  cnt =(cnt+1) % rnd;
  t=t+dmict;
  if(gettime()>t) "handle time overrun"
} // end while
```

Zyklische Planungsverfahren

- Problem:
  - Die Ausführung einer Task passt nicht in einen inneren Zyklus
    $\Delta\ mict < \Delta\ e_i$ für eine Task *i*
- Lösung
  - Die problematische Task wird aufgeteilt in mehrere, hintereinander auszuführende Codeabschnitte ("**Teiltasks**")
  - Die Ausführung jeder Teiltask passt in einen inneren Zyklus
- *Beispiel:*

T	Δe	Δt
1	1	3
2	1	6
3	4	9

T	Δe	Δt
1	1	3
2	1	6
3a	1	9
3b	2	9
3c	1	9

```
cnt =0;
t = gettime();
while (TRUE){
  wait_until(t);
  switch (cnt) {
    case 0: T1;T2;T3a
    case 1: T1;T3b;
    case 2: T1;T2; T3c
    case 3: T1;T3a;
    case 4: T1;T2;T3b;
    case 5: T1;T3c
  }
  cnt =(cnt+1) % rnd;
  t=t+dmict;
  if(gettime()>t)
     "handle time overrun"
} // end while
```

Zyklische Planungsverfahren

- Vorteile
  - kann ohne Betriebssystemunterstützung durchgeführt werden
    - lediglich Zeitmessung und Verzögerungen erforderlich
  - keine unvorhersehbare Prozesswechsel (der Plan ist deterministisch)
  - Tasks können "gefahrlos" über gemeinsame Daten kommunizieren
- Nachteile
  - Keine sporadischen Tasks möglich (Reaktion auf externe Ereignisse)
  - nicht robust gegen Änderung der Task-Parameter (z.B. Verlängerung von Perioden, Änderung von Fristen)
  - Die Aufteilung von "langen" Tasks in Codeabschnitte kann die Code-Struktur verschleiern

    → Der Quellcode ist schlechter lesbar und wartbar
  - Das Problem einen brauchbaren Plan zu finden, ist NP-vollständig (exponentielle Komplexität bezüglich der Anzahl der Tasks)

    → nur für kleine Taskmengen sinnvoll

Task-basierte Planung

- Bei der zyklischen Planung haben Tasks zur Laufzeit keine Bedeutung
  - Die Ausführung einer Task bzw. einer Teiltask wird nicht unterbrochen
  - Ein Zyklus ist im Prinzip eine Folge von Prozeduraufrufen
  - Es wird keine Unterstützung vom Betriebs- oder Laufzeitsystem benötigt
- **Task-basierte Planung**:
  - Tasks existieren als Objekte zur Laufzeit
  - Die Verwaltung der Tasks wird durch das Betriebs- oder Laufzeitsystem unterstützt
  - Eine vom Betriebssystem verwaltetes Task-Objekt ist in einem der folgenden Zustände (ohne Kommunikation und Synchronisation)
    - ausführbereit
    - blockiert (suspeded) – wartend auf ein Zeitereignis
      - bei Periodischen Tasks
    - blockiert (suspended) – wartend auf ein Nicht-Zeitereignis
      - bei sporadischen Tasks

Task-basierte Planung

- Planungsverfahren
  - **Planen nach festen Prioritäten** (***Fixed-Priority Scheduling, FPS***)
    - Jede Task hat eine feste **Priorität**, die vor der Laufzeit (**statisch**) berechnet wird
    - Die ausführbereiten Tasks werden in der Reihenfolge ihrer Priorität ausgeführt
      - Es darf nie eine ausführbereite Task mit höherer Priorität geben
    - D.h. Planung bedeutet festlegen der Prioritäten so, dass alle Zeitbedingungen erfüllt werden können
  - **Planen nach Fristen** (***Earliest Deadline First Scheduling***, ***EDF***)
    - Die ausführbereiten Tasks werden in der Reihenfolge ihrer **absoluten Fristen** (***deadlines***) ausgeführt
      - es darf nie eine ausführbare Task mit einer kürzeren (näheren) Deadline geben
    - Die absoluten Fristen werden zur Laufzeit (**dynamisch**) aus den gegebenen Zeitbedingungen (z.B. relative Fristen) berechnet

- **Planbarkeitstest (*schedulability test*)**
  - Ein Verfahren, welches eine Aussage darüber macht, ob
    - für eine Menge von Tasks mit bestimmten Parametern
    - nach einem bestimmten Planungsverfahren

    ein brauchbarer Plan existiert (→ positives Testergebnis) oder nicht (→ negatives Testergebnis)
  - Eigenschaften von Planbarkeitstests
    - **Hinreichend** (***sufficient***) ist ein Test dann, wenn ein positives Testergebnis garantiert, dass alle Deadlines immer eingehalten werden
    - **Notwendig** (***neccessary***) ist ein Test dann, wenn ein negatives Testergebnis bedeutet, dass mindestens eine Deadline nicht eingehalten wird während der Ausführung einer Taskmenge nach Plan
    - **Genau** (***exact***) ist ein Test, der sowohl hinreichend als auch notwendig ist
      - Genaue Tests sind häufig nicht handhabbar → Es werden oft hinreichende, aber nicht notwendige Tests verwendet ("**pessimistische**" Tests)
    - **Nachhaltig** (***sustainable***): Ein positives Testergebnis bedeutet, dass jeder brauchbare Plan auch brauchbar bleibt, wenn sich die Prozessparameter positiv verändern (z.B. längere Perioden oder Fristen, kürzere Rechenzeit)

Task-basierte Planung

- Planung mit **Vorrechten** und **Verdrängung** (***Preemtive Scheduling***)
  - Situation: Ein Task belegt den Prozessor und eine andere Task mit höherer Priorität wird rechenbereit
  - Beim **präemptiven Planen** (***preemtive scheduling***) bekommt die rechenbereite Task sofort den Prozessor die Task mit niedrigerer Priorität wird **verdrängt**
    - die Task mit niedrigerer Priorotät wird erst fortgesezt, wenn keine Tasks mit höherer Priorität mehr rechenbereit sind
  - Beim **nicht-präemptiven Planen** (***non-preemptive sceduling***) wird die rechnende Task vollständig ausgeführt, bevor die Task mit höherer Priorität den Prozessor bekommt (keine Verdrängung)
  - Bei der **verzögerten Präemption** (***deferred preemption***) darf die rechnende Task zunächst noch eine gewisse Zeit weiterrechnen, bevor sie verdrängt wird
  - Analog beim Planen nach Fristen

Task-basierte Planung

- Task Modell
  - Eine Anwendung besteht aus einer festen Menge von ***N*** Tasks $t_1, \ldots, t_N$
  - Alle Tasks sind Periodisch mit bekannter Periodendauer Δt_i
  - Die Tasks sind unabhängig voneinander
  - Die Rechenzeit für Verwaltungsoperationen des Betriebssystems (z.B. Task-Wechsel) sind vernachlässigbar
  - Die Deadline für jede Task ist gleich dem Ende ihrer Periodendauer
    - $\Delta t_i = \Delta s_i + \Delta d_i$
  - Alle Tasks haben eine feste maximale Ausführungszeit (*worst case execution time, WECT*)
  - Die Ausführung einer Task in einer Periode wird ***Release*** oder ***Job*** genannt
    - t^k_i: Das k-te Release der Task t_i:

Task-Parameter (vgl. S. 6-4):

Δt_i: Periodendauer der Task t_i

Δe_i: Maximale Ausführungszeit für eine Ausführung von t_i :

Δd_i: relative Deadline für eine Ausführung der Task t_i *(Zeitraum vom Start bis zum spätesten Ende der Ausführung)*

Δs_i: Wartezeit bis zum Start (nach Periodenbeginn)

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- Planen nach monotonen Raten (***rate monotonic scheduling, RMS***)
  - Annahme: alle Periodendauern Δt_i sind paarweise verschieden
  - Jede Task t_i bekommt eine eindeutige Priorität p_i zugeteilt
    - Priorität 1 ist die niedrigste, Priorität n die höchste (wichtigste) Priorität
  - Je kürzer die Periode, desto höher die Priorität
    - $\Delta t_i < \Delta t_j \Leftrightarrow p_i > p_j \quad i,j = 1, \ldots, N$
  - *Beispiel:*

Task	Periode Δt_i	Priorität p_i
1	24	5
2	60	3
3	42	4
4	105	1
5	75	2

  - *Satz:* Wenn eine Taskmenge einen brauchbaren Plan nach festen Prioritäten besitzt, dann auch einen nach monotonen Raten

    Beweis: [Zöbel 2008, Kap. 3.3.2, Satz 3.3.2]

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Planbarkeitstest für Planen mit monotonen Raten: **LL-Test**
  - Die **Auslastung** (***utilization***) *U(T)* für eine Taskmenge *T* ist definiert durch

$$U\ (T\ ) = \sum_{i=1}^{N} \frac{\Delta e_i}{\Delta t_i}$$

  - *Satz:* [**Liu & Layland** 1973] Eine Taskmenge T hat einen brauchbaren Plan nach monotonen Raten, falls gilt

$$U(T) \leq N(2^{1/N} - 1)$$

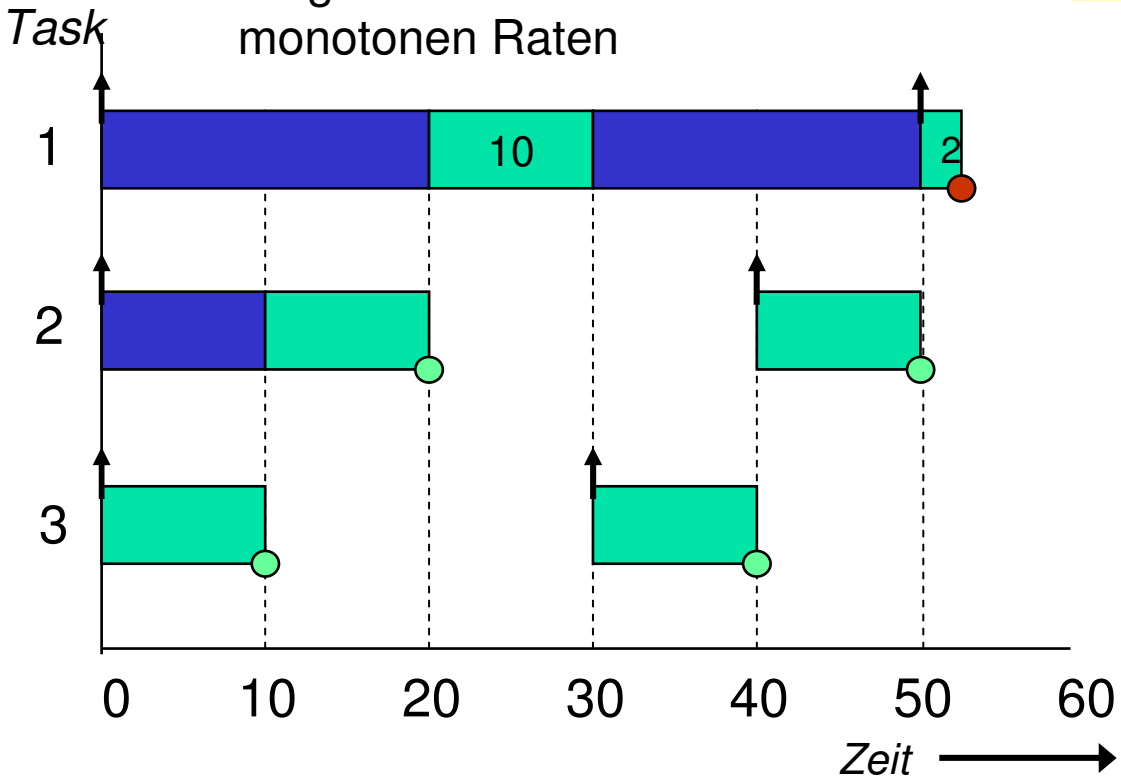
  - Der Term $N(2^{1/N} - 1)$ heißt **Auslastungsgrenze** für N
    - Für N → ∞ ist nähert sich die Auslastungsgrenze asymptotisch dem Wert 0,693…

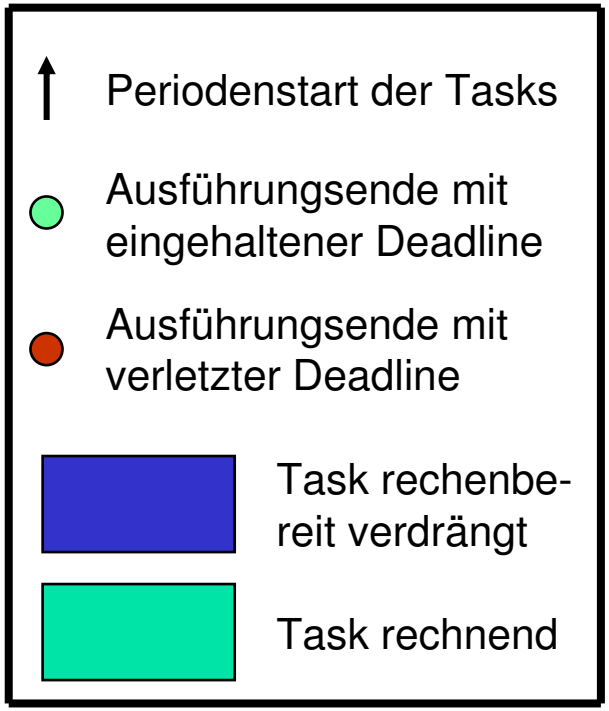
N	Auslastungsgrenze
1	1
2	0.828
3	0.78
4	0.757
5	0.743
10	0.718

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- Planbarkeitstest für Planen mit monotonen Raten: LL-Test
  - *Beispiel:* Task-Menge ***A***
    - Auslastung ***U(A) = 0.82***
    - Auslastungsgrenze für 3 Tasks: **0.78**
      → LL-Test negativ!
    - Es gibt keinen Plan nach monotonen Raten

Task	Δt_i	Δe_i	p_i	$\Delta e_i / \Delta t_i$
1	50	12	1	0.24
2	40	10	2	0.25
3	30	10	3	0.33

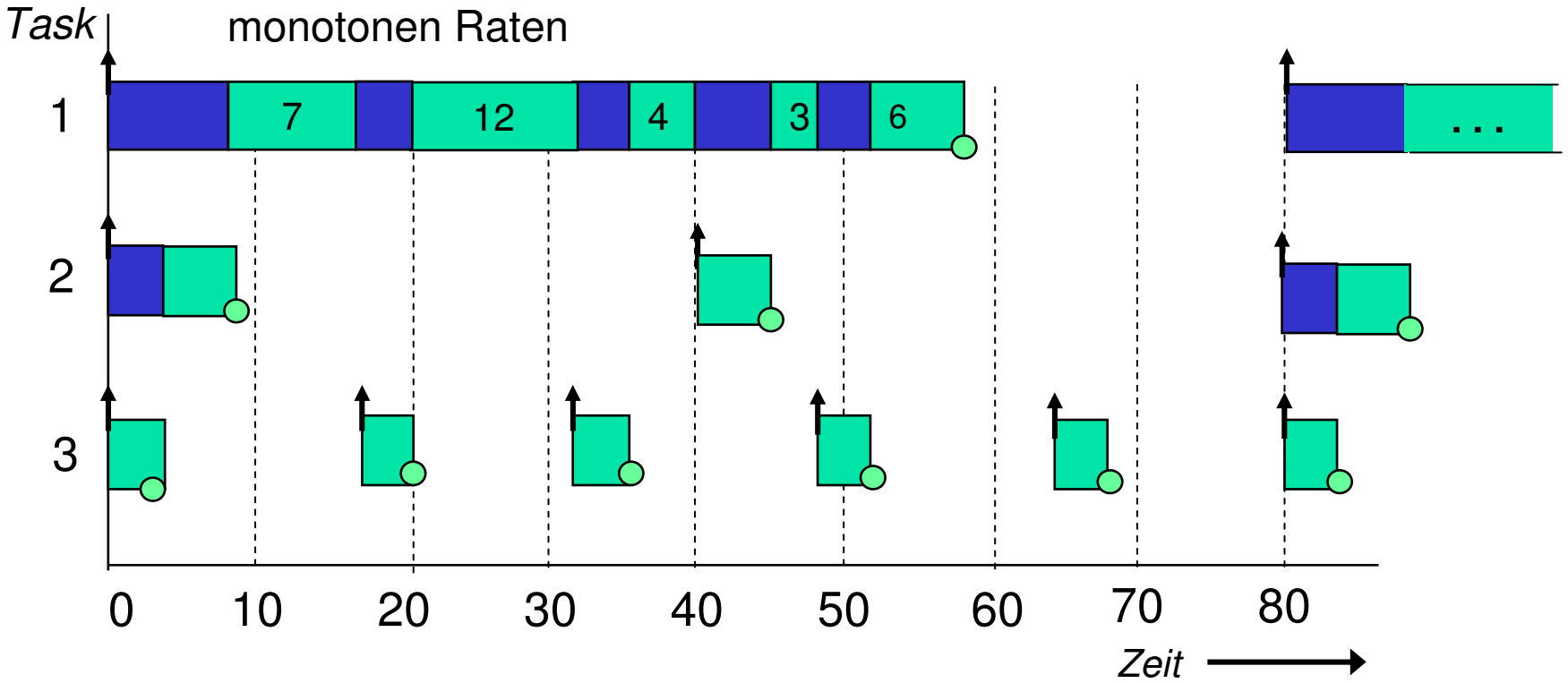




Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- Planbarkeitstest für Planen mit monotonen Raten: LL-Test
  - *Beispiel:* Task-Menge ***B***
    - Auslastung ***U(B) = 0.775***
    - Auslastungsgrenze für 3 Tasks: **0.78**
      → LL-Test positiv!
    - Es gibt (sicher) einen Plan nach monotonen Raten

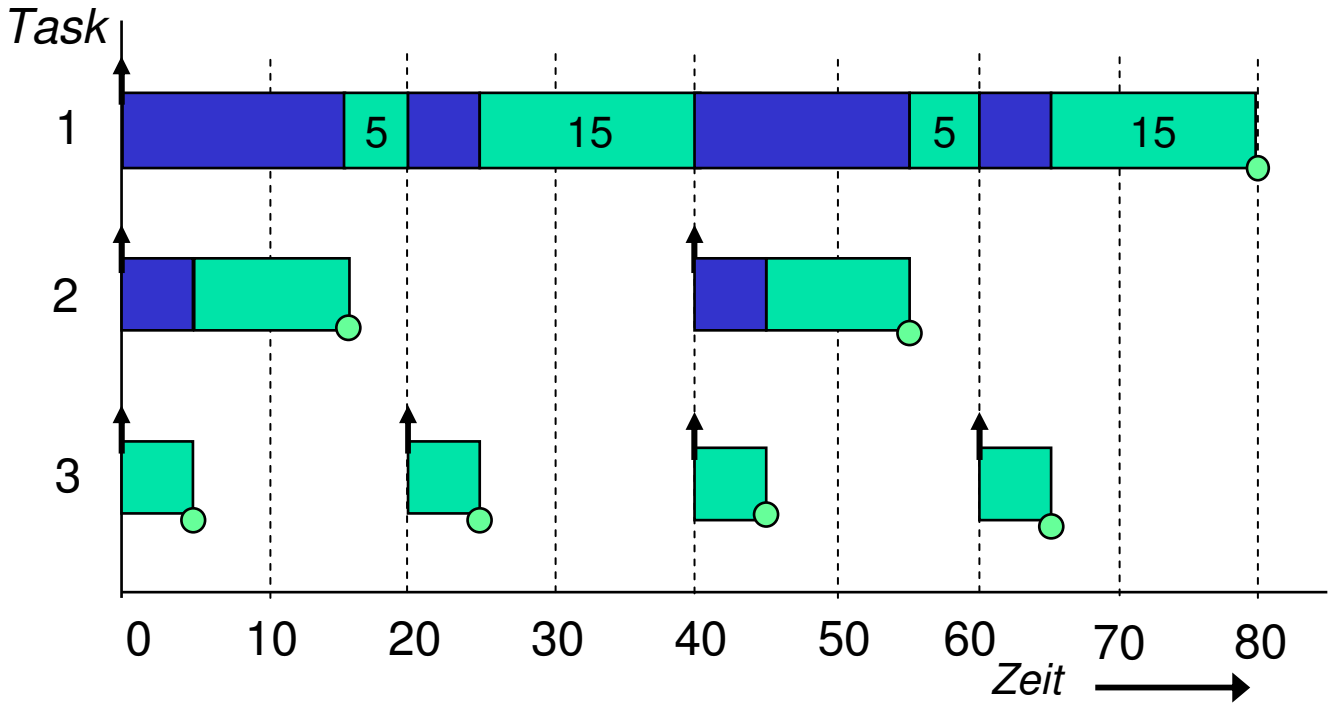
Task	Δt_i	Δe_i	p_i	$\Delta e_i / \Delta t_i$
1	80	32	1	0.400
2	40	5	2	0.125
3	16	4	3	0.250



Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Planbarkeitstest für Planen mit monotonen Raten: LL-Test
  - *Beispiel:* Task-Menge ***C***
    - Auslastung ***U(C) = 1.0***
    - Auslastungsgrenze für 3 Tasks: **0.78**
      → LL-Test negativ!
    - Es gibt (trotzdem) einen Plan nach monotonen Raten

Task	Δt_i	Δe_i	p_i	$\Delta e_i / \Delta t_i$
1	80	40	1	0.50
2	40	10	2	0.25
3	20	5	3	0.25



→ Der LL-Test ist **pessimistisch** (hinreichend, aber nicht notwendig)!

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Planbarkeitstest für Planen mit monotonen Raten: **HB-Test**
  - *Satz:* [Bini et al. 2003] Eine Taskmenge T hat einen brauchbaren Plan nach monotonen Raten, falls gilt

$$\prod_{i=1}^{N} \left(\frac{\Delta e_i}{\Delta t_i} + 1 \right) \leq 2$$

  - *Beispiele:*
    - Für Task-Menge C ist der Test immer noch negativ

      1.5 * 1.25 * 1.25 = 2.343 > 2

      → Der HB-Test ist auch pessimistisch!
    - Taskmenge B':
      - LL-Test negativ
        - » U(C') = 0.796 < 0.78
      - HB-Test positiv
        - » 1,421 * 1.125 * 1.25 = 1.998 < 2

      → Der HB-Test ist weniger pessimistisch als der LL-Test

Task	Δt_i	Δe_i	p_i	$\Delta e_i / \Delta t_i$
1	**76**	32	1	**0.421**
2	40	5	2	0.125
3	16	4	3	0.250

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- **Antwortzeitanalyse** (RT-Test, *response time test*)
  - Idee
    - Berechne die Antwortzeiten Δr_i der einzelnen Task
    - Vergleiche die Antwortzeiten mit der Deadline Δd_i:
  - Satz: die Taskmenge ist planbar, falls für alle Tasks $\Delta r_i < \Delta d_i$
  - **Antwortzeit (*response time*)**
    - Die Zeit Δr_i , die benötigt wird von Beginn der Ausführung der Tasks t_i bis zum Ende der Ausführung
  - **Interferenzzeit** (***interference time***)
    - Bei präemptivem Multitasking besteht die Antwortzeit aus der Ausführungszeit Δe_i , in der eine Task rechnet, und der Zeit Δi_i , in der die Task verdrängt ist durch Tasks mit höherer Priorität
      - $\Delta r_i = \Delta e_i + \Delta i_i$
      - Für die Task m mit der höchsten Priorität gilt $\Delta i_m = 0$
  - **Der RT-Test ist exakt (hinreichend und notwendig)!**

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Antwortzeitanalyse – Berechnung der Antwortzeit

  Annahme (o.B.d.A): Alle Tasks haben Wartezeit 0 ($\Delta s_i = 0$)

  – ***hp(i)***: die Menge der Tasks, deren Priorität höher ist als die von t_i.

  – Für j aus hp(i) ist

  •die **maximale Interferenzanzahl** für i mit j

  – sooft kann innerhalb der Antwortzeit von Task i die Task j gestartet werden

$$IN_{\max}(i,j) = \left\lceil \frac{\Delta r_i}{\Delta t_j} \right\rceil$$

  •Die **maximale Interferenzzeit** einer Task i mit j

  – solange kann die Task i von der Task j maximal verdrängt werden

$$IT_{\max}(i,j) = \left\lceil \frac{\Delta r_i}{\Delta t_j} \right\rceil \Delta e_j$$

  – *Satz:* [Jospeh & Pandya 1986]

  •Drückt die Antwortzeit einer Task unter Berücksichtigung der maximalen Interferenzzeit durch Tasks mit höherer Priorität aus.

Interferenzgleichung:

$$\Delta r_i = \Delta e_i + \sum_{j \in hp(i)} \left\lceil \frac{\Delta r_i}{\Delta t_j} \right\rceil \Delta e_j$$

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Antwortzeitanalyse – Berechnung der Antwortzeit
  - Die Interferenzgleichung kann mit Hilfe folgender Rekurrenz-Beziehung gelöst werden:

$$w_i^{n+1} = \Delta e_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{\Delta t_j} \right\rceil \Delta e_j$$

Lösungsansatz:

- n=0:

  wähle $w_i^0 \leq \Delta e_i$
  (z.B. $w_i^0 = 0$ oder $w_i^0 = \Delta e_i$*)*

- n → n+1:

  berechne solange w_i^{n+1} aus w_i^n,
  bis $w_i^n = w_i^{n+1}$

  → Lösung gefunden

  oder $w_i^n > \Delta t_i$

  → keine Lösung

Lösungsalgorithmus

```
for i in 1..N loop
-- for each process in turn
   n := 0
   w_i^n = Δe_i
loop
      calculate new w_i^(n+1) from w_i^n
      if w_i^n = w_i^(n+1) then
         Δr_i = w_i^n
         exit value found
      end if
      if w_i^(n+1) > Δt_i then
         exit value not found
      end if
      n := n + 1
   end loop
end loop
```

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Antwortzeitanalyse – Berechnung der Antwortzeit
  - *Beispiel:* Task-Menge D
    - $\Delta r_i < \Delta d_i = \Delta t_i$ *für i=1,…,3*
    - Aufgabe: Welche Aussage liefern der LL- und der HB-Test?

Task	Δt_i	Δe_i	p_i	Δr_i
1	7	3	3	3
2	12	3	2	6
3	20	5	1	20

  - *Beispiel:* Task-Menge C (vgl. S. 6-20)
    - $\Delta r_i < \Delta d_i = \Delta t_i$ *für i=1,…,3*
    - Konnte mit LL- und HB-Test nicht als planbar nachgewiesen werden!

Task	Δt_i	Δe_i	p_i	Δr_i
1	80	40	1	80
2	40	10	2	15
3	20	5	3	5

  - *Aufgabe:* Antwortzeitanalyse für die Taskmenge A (S. 6-18)

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- **Sporadische und aperiodische Tasks**
  - Bei sporadischen Tasks muss die "Ankunftszeit" von Ereignissen, die ein neues Release auslösen, betrachtet werden (statt einer Periode)
    - Δt_i := *mittlere Ankunftszeit* oder
    - Δt_i := *minimale Ankunftszeit*
  - $\Delta d_i < \Delta t_i$ ist erlaubt (weil typisch für sporadische Tasks)
  - Aussagen zur Antwortzeitanalyse
    - Funktioniert auch für sporadische Tasks mit diesen Eigenschaften und folgendem Abbruchkriterium (vgl. S. 6-24)

      $w_i^n > \Delta d_i$ (statt: $w_i^n > \Delta t_i$)
    - Funktioniert auch für andere Prioritätsordnungen (nicht nur nach monotonen Raten)
  - Harte und Weiche Tasks
    - "Harte" Tasks müssen Deadlines unter allen Umständen einhalten
    - Bei "weiche" Tasks sind gelegentliche Deadline-Verletzungen tolerierbar

    →Regeln für die Antwortzeitanalyse

    –Bei weichen Tasks wird die **mittlere** Ankunftszeit verwendet

    –Bei harten Tasks wird die **minimale** Ankunftszeit verwendet

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- **Periodische Tasks mit $\Delta d_i < \Delta t_i$**
  - Problem: Planen nach monotonen Raten ist nur optimal, wenn die Deadline gleich dem Periodenende ist ($\Delta d_i < \Delta t_i$)
  - Lösung: **Planen nach monotonen Fristen** (***deadline monotonic Scheduling, DMS***)
    - $\Delta d_i < \Delta d_j \Leftrightarrow p_i > p_j$
  - *Beispiel:*
    - Die Perioden, Deadlines, Ausführungszeiten sind vorgegeben
    - Die Antwortzeiten sind berechnet nach dem Verfahren auf S. 6-24 mit $w_i^n > \Delta d_i$ als Abbruchkriterium
      → Die Taskmenge ist planbar!
    - Die Prioritäten sind festgelegt nach DMS

Task	Δt_i	Δd_i	Δe_i	p_i	Δr_i
1	20	5	3	4	3
2	15	7	3	3	6
3	10	10	4	2	10
4	20	20	3	1	20

  - *Satz:* Jede Taskmenge, die planbar ist mit festen Prioritäten (FPS), ist auch planbar mit DMS
    - Beweis: [Burns & Wellings 2009, Kap. 11.7.1]

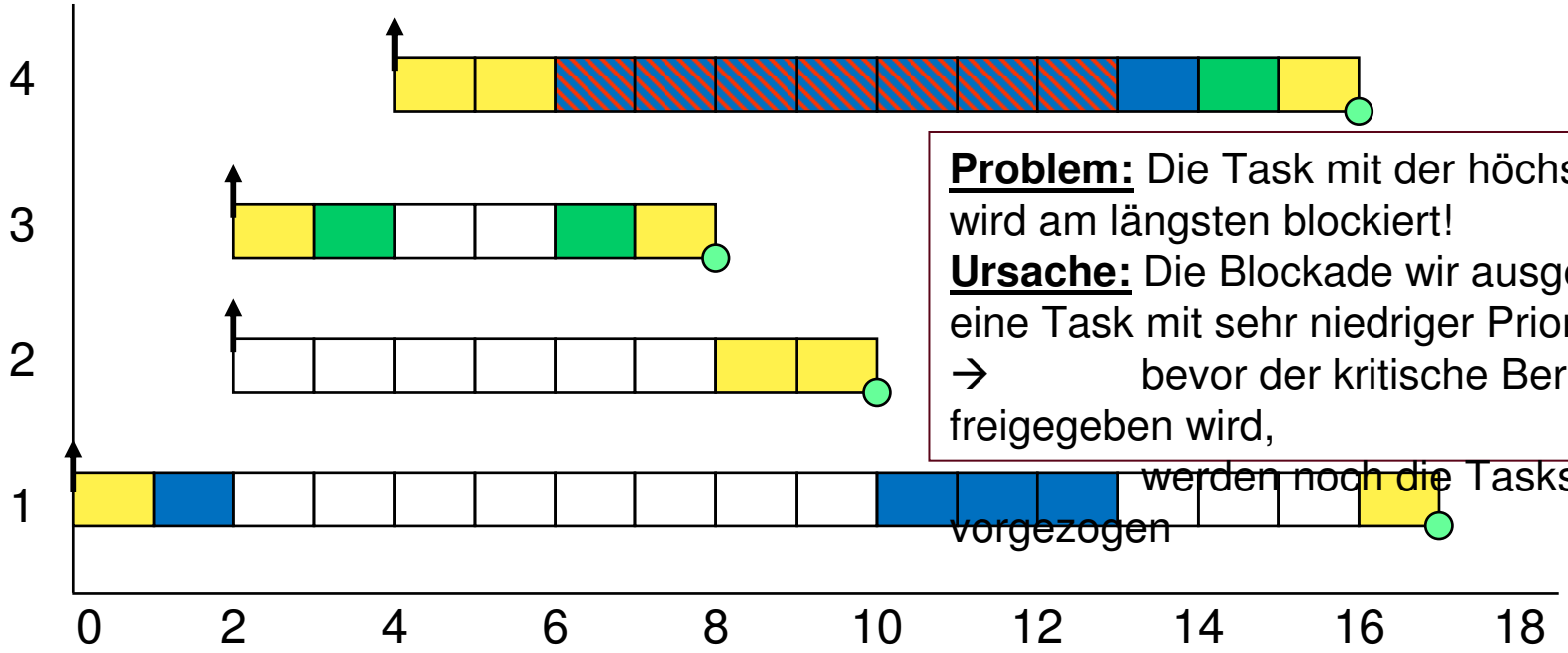
Planen mit festen Prioritäten (Fixed Priority Scheduling)

- **Planen von synchronisierten Tasks**
  - **Prioritäts-Umkehrung** (***priority inversion***)
    - Eine Task t mit höherer Priorität kann durch eine Task t' mit niedrigerer Priorität **blockiert** werden, z.B. (vgl. Kap. 4 und 5)
      - t möchte in einen kritischen Bereich eintreten, der von t' belegt ist (abgesichert durch Semaphore, Mutexe, geschützte Objekte,…)
      - t wartet auf ein Rendezvous mit t'
      - t wartet auf eine Nachricht von t'
    - In diesem Fall muss die durch die Priorität festgelegte Ausführungsordnung geändert werden
    - *Beispiel:*
      - 4 Tasks, Priroität p_i, Startzeit s_i
      - jeder Buchstabe in der **Ausführungs-folge** entspricht einer Zeiteinheit
      - **Q**: Task rechnet im kritischen Bereich Q (genutzt von Task 1 und 4)
      - **V**: Task rechnet im kritischen Bereich *V* (genutzt von Task 3 und 4)
      - **E**: Task rechnet

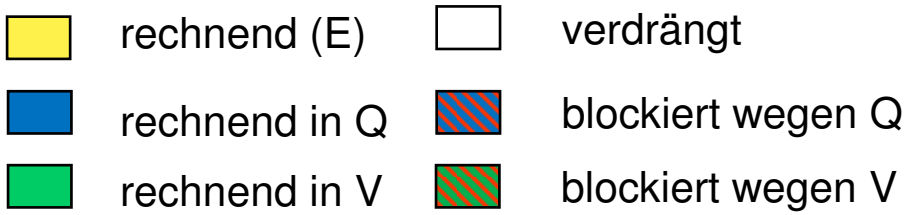
Task	p_i	Ausführungsfolge	s_i
1	4	EQQQQE	0
2	3	EE	2
3	2	EVVE	2
4	1	EEQVE	4

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Planen von synchronisierten Tasks
  - *Beispiel* (Forts.)



Problem: Die Task mit der höchsten Priorität wird am längsten blockiert!
Ursache: Die Blockade wir ausgelöst durch eine Task mit sehr niedriger Priorität
→ bevor der kritische Bereich freigegeben wird,
werden noch die Tasks 2 und 3 vorgezogen

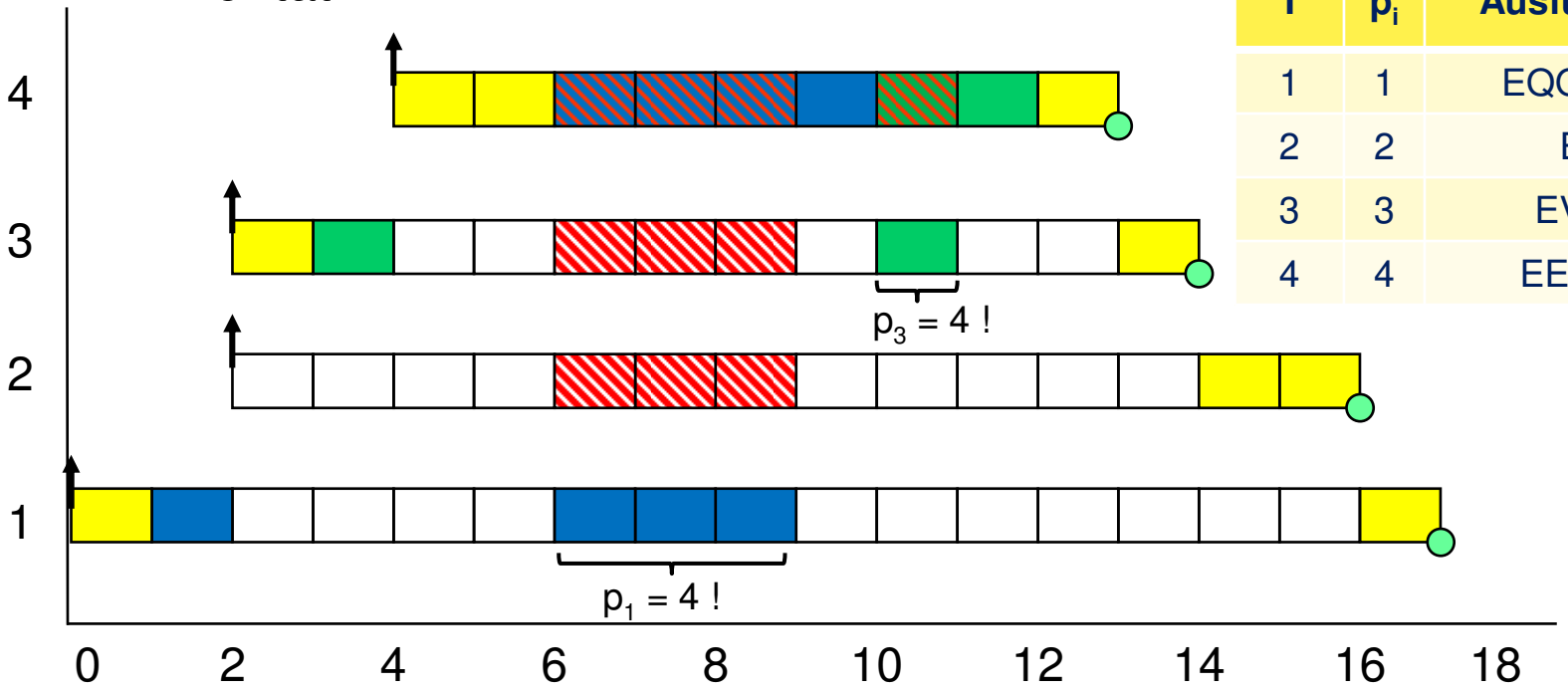


Task	p_i	Ausführungsfolge	s_i
1	1	EQQQQE	0
2	2	EE	2
3	3	EVVE	2
4	4	EEQVE	4

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- **Prioritätsvererbung** (***priority inheritance protocol, PIP***)
  - Wenn eine Task t eine andere Task t' blockiert, übernimmt sie deren Priorität

T	p_i	Ausführung	s_i
1	1	EQQQQE	0
2	2	EE	2
3	3	EVVE	2
4	4	EEQVE	4



rechnend (E)

rechnend in Q

rechnend in V

verdrängt durch eine Task mit höherer **statischer** Priorität

blockiert wegen Q

blockiert wegen V

verdrängt durch eine Task mit höherer **geerbter** Priorität

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Prioritätsvererbung (*priority inheritance*) - Berechnung der Blockierungszeit (*blocking time*)

  → *Wie lange kann eine Task maximal blockiert sein?*

– Parameter

  - Es gibt ***K*** kritische Bereiche in der Taskmenge ***T***
  - ***B(k,i)*** = *1*
    - falls es mindestens eine Task t_j gibt mit $p_j < p_i$ und mindestens eine Task t_l mit $p_l \geq p_i$, die im kritischen Bereich *k* rechnen möchte ($1 \leq k \leq K$)
  - ***B(k,i)*** = *0* sonst
  - Δc_k : maximale Ausführungszeit des kritischen Bereiches k

– Die **maximale Blockierungszeit** (***maximum blocking time***) der Task *i*:

$$\Delta b_i = \sum_{k=1}^{K} B(k,i)\Delta c_k$$

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Prioritätsvererbung (*priority inheritance*) – Antwortzeitanalyse
  - Bei der Antwortzeitanalyse muss zusätzlich zur maximalen Ausführungszeit die maximale Blockierungszeit berücksichtigt werden (vgl. S. 6-23f)

    - Antwortzeit: $\Delta r_i = \Delta e_i + \Delta b_i + \Delta i_i$

    - Interferenzgleichung: $$\Delta r_i = \Delta e_i + \Delta b_i + \sum_{j \in hp(i)} \left\lceil \frac{\Delta r_i}{\Delta t_j} \right\rceil \Delta e_j$$

    - Rekurrenzrelation: $$w_i^{n+1} = \Delta e_i + \Delta b_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{\Delta t_j} \right\rceil \Delta e_j$$

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Prioritätsvererbung (*priority inheritance*) – Nachteile
  - Antwortzeitanalyse ist jetzt ein pessimistischer Test
    - ohne Prioritätsvererbung: exakt (hinreichend und notwendig)
  - Grund: die maximale Blockierungszeit ist eine sehr grobe Abschätzung der Blockierungszeit
    - Wie nahe die tatsächliche Blockierungszeit an der maximalen ist, hängt sehr stark von den Perioden, Warte- und Startzeiten ab
    - Beispiel
      - Alle Tasks haben dieselbe Periode und Wartezeit 0
      - In einer planbaren Taskmenge mit festen Prioritäten gibt keine Verdrängung und keine Blockierung und damit auch keine Prioritätsvererbung
      - trotzdem kann die maximale Blockierungszeit > 0 sein
  - Tasks können mehrmals durch Tasks mit niedrigerer Priorität blockiert werden (→ Bsp. S. 6-30)
  - Transitive Blockaden sind möglich
    - Task a wird blockiert von Task b, b wird blockiert von Task c etc.
  - Deadlocks werden nicht verhindert

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- **Prioritätsobergrenzen (*Priority Ceiling Protocols, PCP*)**
  - Variante 1: *Original Ceiling Priority Protocol (OCPP)*
    - Siehe [Burns & Wellings 2009], Kap. 11.9
  - Variante 2: ***Immediate Ceiling Priority Protocol (ICPP)***
  - Eigenschaften (für beide Varianten)
    - Eine Task mit hoher Priorität kann höchstens einmal blockiert werden durch eine Task mit niedriger Priorität
    - Deadlocks werden verhindert (ICPP ist gleichzeitig eine Deadlock-Test!)
    - Es gibt keine transitiven Blockaden
    - Gegenseitiger Ausschluss bei kritischen Bereichen wird durch das Protokoll gleich mit realisiert
  - *Idee:*

    Wenn eine Task *a* einen kritischen Bereich belegt und dadurch eine Task *b* mit höherer Priorität blockieren kann

    dann darf kein anderer kritischer Bereich, der b auch blockieren könnte, von keiner anderen Task betreten werden

    - *(Gegen-) Beispiel* S. 6-30: zum Zeitpunkt 3 ist Task 1 im kritischen Bereich Q und Task 3 betritt den kritischen Bereich V, beide können Task 4 blockieren

Planen mit festen Prioritäten (Fixed Priority Scheduling)

- Prioritätsobergrenzen (*Priority Ceiling Protocols*)
  - ***Immediate Ceiling Priority Protocol (ICPP)***
    - **Definitionen** für Tasks $i \in \{1,\ldots N\}$, kritische Bereiche $k \in \{1,\ldots,K\}$:
      - ***use(i,k)* = *true*** genau dann, wenn Task i den kritischen Bereich k nutzt
      - ***lock(i,k,t)* = *true*** genau dann, wenn Task i zum Zeitpunkt t den kritischen Bereich k belegt
      - p_i: die **statische Priorität** der Task *i* (z.B. nach monotonen Raten/Fristen)
      - $p_{max}(k)$: die **statische Obergrenze** (***ceiling value***) des kritischen Bereichs k d.h die höchste Priorität einer Task, die *k* benutzt

$$p_{max}(k) := max(\{\, p_l \mid i \in \{1,\ldots,N\} \wedge use(i,k) \,\})$$

    - **Protokoll:** die **dynamische Priorität $p_{run}(i,t)$** einer Task *i* zum Zeitpunkt *t* ergibt aus dem Maximum der eigenen statischen Priorität und den statischen Obergrenzen aller kritischen Bereiche, die *i* belegt.

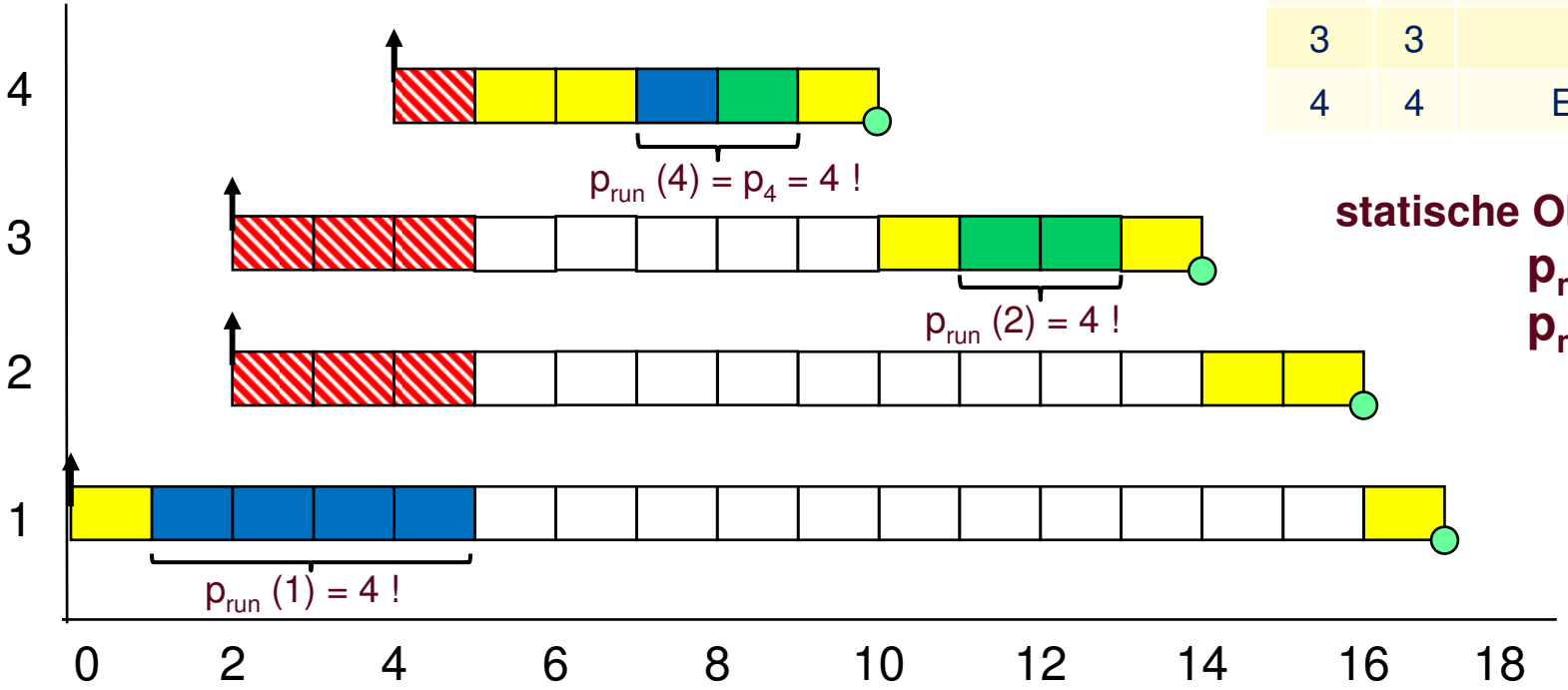
$$p_{run}(i,t) := \max(\{\, p_{max}(k) \mid k \in \{1,\ldots,K\} \wedge lock(i,k,t) \,\} \cup \{\, p_i \,\})$$

  - Eigenschaft des ICPP:
    - Eine Task kann höchstens vor Eintritt in einen kritischen Bereich blockiert werden, nicht während sie ihn nutzt

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- *Immediate ceiling prirority protocol (ICPP)*
  - *Beispiel* (vgl. S. 6-30)

T	p_i	Ausführung	s_i
1	1	EQQQQE	0
2	2	EE	2
3	3	EVVE	2
4	4	EEQVE	4



statische Obergrenzen:

$p_{max}(Q) = 4$

$p_{max}(V) = 4$

rechnend (E)

rechnend in Q

rechnend in V

verdrängt durch eine Task mit höherer **statischer** Priorität

blockiert wegen Q

blockiert wegen V

verdrängt durch eine Task mit höherer **geerbter (dynamischer)** Priorität

Planen mit festen Prioritäten *(Fixed Priority Scheduling)*

- Prioritätsobergrenzen (*priority ceiling Protocols*) – Antwortzeitanalyse
  - Berücksichtigung der maximalen Blockierungszeit (vgl. S. 6-32)
  - Abschätzung der maximalen Blockierungszeit ändert sich (vgl. S. 6-31)
  - Parameter
    - Es gibt ***K*** kritische Bereiche in der Taskmenge ***T***
    - ***B(k,i)*** = *1*
      - falls es mindestens eine Task t_j gibt mit $p_j < p_i$ und mindestens eine Task t_l mit $p_l \geq p_i$, die im kritischen Bereich *k* rechnen möchte ($1 \leq k \leq K$)
    - ***B(k,i)*** = *0* sonst
    - Δc_k: maximale Ausführungszeit des kritischen Bereiches k
    - Δb_i: Die **maximale Blockierungszeit** (***maximum blocking time***) der Task *i*:

$$\Delta b_i = \max_{1 \leq k \leq K} B(k,i) * \Delta c_k$$

  - Berechnung der Antwortzeiten mit der Rekurrenzrelation (S. 6-32) und dem Lösungsalgorithmus von S. 6-24)

Planen nach Fristen (Earliest Deadline First Scheduling)

- Die ausführbereiten Tasks werden in der Reihenfolge ihrer **absoluten Fristen** (***deadlines***) ausgeführt
  - es darf nie eine ausführbare Task mit einer kürzeren (näheren) Deadline geben
- Die absoluten Fristen werden **zur Laufzeit** (dynamisch) aus den gegebenen Zeitbedingungen (z.B. relative Fristen) berechnet
- Planbarkeitstest für EDF (**LL-Test**, [Liu & Layland 1973])
  - Unter den Voraussetzungen von S. 6-15 ist eine Taskmenge T planbar, wenn gilt:

$$U(T) = \sum_{i=1}^{N} \frac{\Delta e_i}{\Delta t_i} \leq 1$$

  - Folge: EDF ist "besser" als FPS (vgl. LL-Test für FPS, S. 6-17)
    - Jede mit FPS planbare Taskmenge ist auch mit EDF planbar
    - Nicht jede mit EDF planbare Taskmenge ist auch mit FPS planbar
      - *Beispiel:* Taskmenge A, S. 6-18

Planen nach Fristen (Earliest Deadline First Scheduling)

- FPS vs EDF
  - FPS ist leichter zu implementieren, da die Prioritäten statisch vergeben werden
  - EDF erfordert eine komplexere Laufzeit-Unterstützung, führt zu mehr Verwaltungsaufwand
  - Die meisten Protrammiersprachen/Betriebssysteme unterstützen FPS, nicht EDF
  - In FPS können leichter Tasks ohne deadline integriert werden (beliebige deadlines vergeben, wo keine sind, ist unnatürlich!)
  - Es ist leichter, andere Kriterien in FPS zu integrieren als in EDF (z.B. Kritikalität von Tasks)
  - Bei Überauslastung ist das Verhalten von FPS eher vorhersagbar
    - Tasks mit niedrigerer Priorität werden ihre Fristen als erstes verletzen
  - Das Verhalten von EDF bei Überauslastung ist nicht vorhersagbar
    - "Domino-Effekte" sind möglich: viele Tasks verletzen ihrer Fristen
  - Aber: EDF nutzt den Prozessor besser!

Planen nach Fristen (Earliest Deadline First Scheduling)

- Planbarkeitstests für EDF
  - Der LL-Test geht nur für den Fall $\Delta d_i = \Delta t_i$
  - Es gibt auch Test für allgemeiner Fälle
    - *Processor Demand Criteria* (PDC, [Burns & Wellings 2009, Kap. 11.11.2])
      - Idee: Prüfe zu bestimmten Zeitpunkten t (korreliert mit Deadlines), ob der Prozessor alles, was bis dahin zu tun ist, schaffen kann.
    - *Quick Processor Demad Analysis* (QPA-Test, [Ebd. Kap. 11.11.3)
      - Vereinfachung der PDC mit 1% des Aufwands!
  - Antwortzeitanalyse geht ebenfalls für EDF, aber es ist wesentlich komplizierter, Antwortzeiten zu berechnen
  - Synchronisation mit kritischen Bereichen kann in ähnlicher Weise berücksichtigt werden wie bei FPS
    - "deadline-Vererbung" statt Prioritätsvererbung
    - Verallgemeinerung des Priority Ceiling Protocols

Rechenzeitanalyse

- Bestimmung der **maximalen Ausführungszeit (*worst case execution time, WCET*)**
  - Durch Messen
    - Eventuell zu optimistisch
      - Ist der "schlimmste Fall" wirklich gemessen worden?
  - Durch statische Analyse des Programms
    - Eventuell zu pessimistisch
    - Schwierig durchzuführen für moderne Prozessoren
      - Caches
      - Befehlspipelines
      - *out-of-order-execution*
      - *branch-prediction*
  - Typisches Vorgehen
    - Repräsentation des Codes als gerichteter Graph von Basis-Blöcken
    - Messung oder Analyse der Rechenzeit von Basisblöcken
    - Linearisierung des Graphen (unter Berücksichtigung von maximalen Schleifendurchgängen, und den "schlechtesten" Verzweigungen.

Rechenzeitanalyse

- Beispiel

```
for i in 1..3 loop
  if Cond then
   Block1; -- Basic block of cost 100
  else
   Block2; -- Basic block of cost 10
  end if;
end loop;
```

  - Ohne weitere semantische Informationen ist die maximale Ausführungszeit 300
  - Wenn z.B. `Cond` maximal 1 mal `true` sein kann, ist die maximale Ausführungszeit 120
  - Eventuell müssen auch Steuerungsoperationen berücksichtigt werden
    - Initialisierung und Inkrementierung der Schleifenvariablen `i`, Schleifentest
    - Auswertung von `Cond` und Verzweigung (Sprungbefehl)

Literatur

[Burns & Wellings 2009] Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages. Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, 2009.

[Wörn & Brinkschulte 2005] Heinz Wörn, Uwe Brinkschulte: *Echtzeitsysteme*. Springer, 2005.

[Zöbel 2008] Dieter Zöbel: *Echtzeitsysteme. Grundlagen der Planung*. Springer, 2008.

[Liu & Layland 1973] C. L. Liu, James W. Leyland: *Scheduling algorithms für multiprogramming in a hard-real-time environment.* Journal of the ACM, 20(1): 46-61, January 1973

[Bini et al. 2003] Enrico Bini, Giorgio Buttazzo, Giuiseppe Buttazzo: *Rate monmotonic scheduling: The hyperbolic bound*. IEEE Transactions on Computers, 52(7): 993-942, July 2003.