Real-Time Systems

3. Erweitertes Real-Time Scheduling

Übersicht

1. Ziel diese Kapitels
2. Schutz von Ressourcen
3. Blockiert-Zeiten
4. Behandlung von Interrupt-Routinen
5. Release Jitter
6. Präzedenzen
7. Offsets
8. Betriebssystem-Overhead
9. Beliebige Deadlines
10. Zusammenfassung
11. Case Study

1. Ziel dieses Kapitels

Um die im vorherigen Kapitel gezeigten Scheduling-Techniken in der **Praxis** anwenden zu können, müssen **zahlreiche Erweiterungen** durchgeführt werden. Diese werden in diesem Kapitel vorgestellt, indem aus Gründen der Vereinfachung immer nur auf eine bestimmte Problemstellung eingegangen wird.

Wir konzentrieren uns dabei auf statisches Scheduling und die Response-Time Analysis.

Zunächst ist es dazu erforderlich, Begriffe und Konzepte wie *Ressourcen* und *Semaphoren* aus der Vorlesung Betriebssysteme zu wiederholen.

2. Schutz von Ressourcen

2.1 Ressourcen (Betriebsmittel)

Eine **Ressource** ist ein

- (angeschlossenes) Gerät oder
- gemeinsam genutzte Daten.

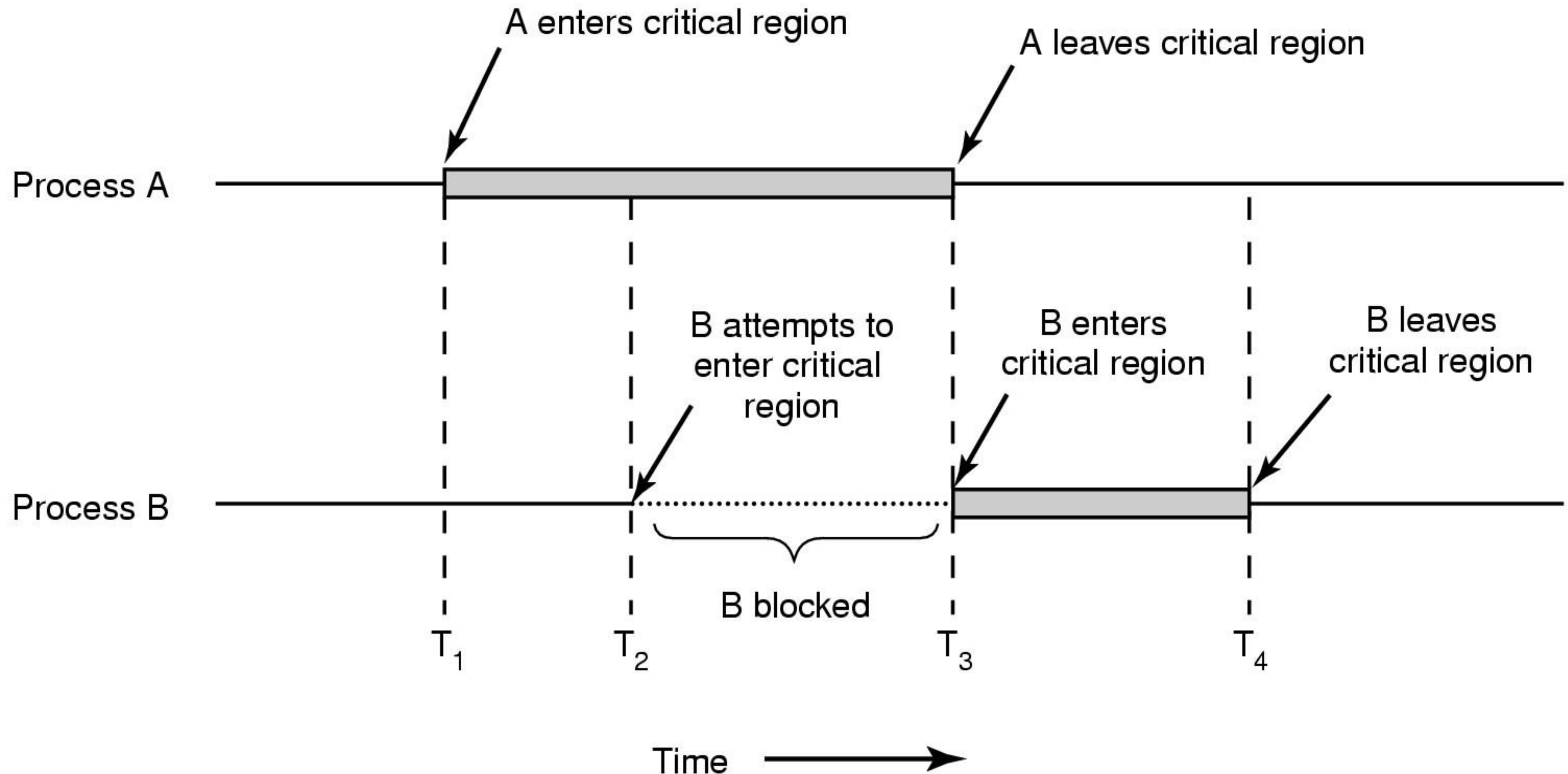
Unterbrechbare Ressourcen können einer Task ohne unerfreuliche Nebenerscheinungen entzogen werden (z.B. CPU). **Ununterbrechbare Ressourcen** können ihrem Besitzer nicht entzogen werden, ohne dass dessen Ausführung fehlschlägt (z.B. CD-Brenner). Allerdings gibt es in manchen Fällen recht einfache aber sehr spezielle Möglichkeiten den Zugriff zu regeln (z.B. Drucker-Spooling).

In allen anderen Fällen tritt ein Problem auf, wenn mehrere Tasks auf gemeinsam genutzte Ressourcen zugreifen, so dass es die Aufgabe des Betriebssystems ist, dafür Schutzmechanismen zur Verfügung zu stellen.

Zur abstrakten Formulierung des Problems für den allgemeinen Fall, werden die Teile eines Computer-Programms, in denen auf gemeinsam genutzte Daten zugegriffen wird, **kritischer Abschnitt** (= Ressource) genannt.

2. Schutz von Ressourcen

Das Betriebssystem stellt **primitive Operationen** zur Verfügung, um **wechselseitigen Ausschluss** zu erzielen. Diese sind vor dem Betreten und nach dem Verlassen eines kritischen Abschnitts zu verwenden.



2. Schutz von Ressourcen

2.2 Semaphoren

Ein Mittel (neben anderen) zur **Interprozess-Kommunikation (IPC)** sind Semaphoren [Dijkstra, 1965]. Ein (zählendes) **Semaphor** ist eine Integer-Variable, die eine bestimmte Anzahl an Weckrufen (Kapazität) verwaltet, und

- deren Wert Null anzeigt, dass alle Weckrufe verbraucht sind,
- ein Wert größer Null bedeutet, dass noch Kapazität vorhanden ist.

Operationen auf Semaphoren sind:

- **`DOWN()`** (oder **`p()`** – *passeren* – **`sem_wait()`** in Linux) prüft, ob der Wert des Semaphors größer als Null ist: dann wird der Wert vermindert.

  Ist der Wert Null, **blockiert** die Task und kann später durch einen anderen Thread per **`UP()`** wieder aktiviert werden.

- **`UP()`** (oder **`v()`** – *vrijgeven* – **`sem_post()`** in Linux) inkrementiert das Semaphor und weckt irgendeine wartende Task.

Beides sind **atomare Operationen**, d.h. es ist sichergestellt, dass während der Ausführung keine andere Task auf das Semaphor zugreifen kann. Wir beschränken uns auf eine maximale Kapazität von ***1*** (auch **binären Semaphoren** bzw. Mutexe).

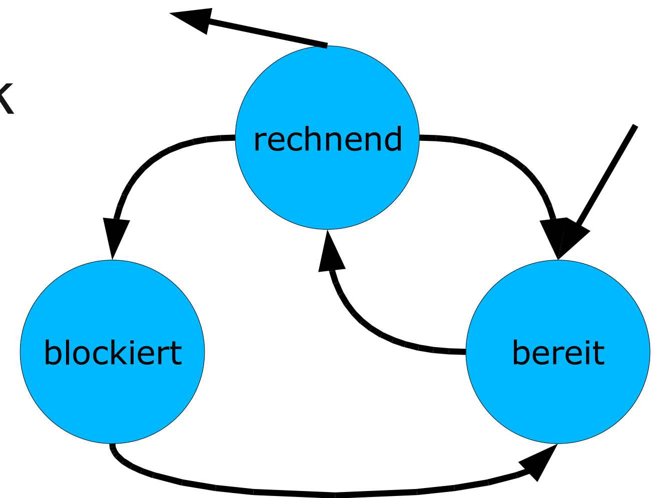
2. Schutz von Ressourcen

Beispiel Semaphoren unter Linux für Konto-Buchungen:

```
#include <semaphore.h>
...
sem_t sem;    /* DEFINE semaphore */
int konten[100];

void buchen (int KNr, int betrag){
   /* passeren */
   sem_wait(&sem);

   int alt = konten[KNr];
   int neu = alt + betrag;
   konten[KNr] = neu;

   /* frijgeven */
   sem_post(&sem);
}
...
main(...

   sem_init(/* name */ &sem, /* shared flag */ 0, /* init value */ 1);

...
```

3. Blockiert-Zeiten

Durch ein **`DOWN()`** auf ein Semaphor, geht eine Task also in den Zustand ***blockiert***, wenn der kritische Abschnitt bereits belegt ist.



Aus Sicht eines RTS ist es dabei problematisch, dass dadurch eine niedrig priorisierte Task eine höher priorisierte für eine bestimmte Dauer blockieren kann. Dies ist in der bisherigen Analyse (RTA) noch nicht berücksichtigt worden. Daher wird ein neuer Parameter eingeführt:

> **Definition**: Die **Blockiert-Zeit** B_i ist die maximale Zeit, die eine Task **`i`** durch niedrig priorisierte Tasks verzögert werden kann.

Die RTA wird erweitert zu:

$$R_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

3. Blockiert-Zeiten

Das Phänomen als solches, dass eben eine hoch-priorisierte Task ***H*** auf eine niedrig priorisierte Task ***L*** warten muss, wird **Priority-Inversion (Prioritätsumkehr)** genannt.
Die **Dauer des Wartens** hängt dabei nicht allein von ***L*** ab sondern womöglich von einer ganzen Anzahl weiterer Tasks, deren Prioritäten zwischen denn von ***L*** und ***H***. Dies ist so bei dem eingangs erwähnten Pathfinder-Beispiel eingetreten.
Im Folgenden wird die allgemeine Problemstellung noch einmal im Detail dargestellt:
Simples Beispiel:

- Eine Task ***L*** mit niedriger Priorität belegt eine Ressource ***R***
- Eine Task ***H*** mit hoher Priorität fordert ***R*** an

➔ ***H*** muss auf die Freigabe durch ***L*** warten

Diese Situation führt zwar dazu, dass eine hoch priorisierter Task auf eine niedrig priorisierten Task warten muss, jedoch ist dies nicht allzu problematisch. Es lässt sich nämlich leicht vorher feststellen, ob dieses passieren kann, d.h. die Blockiert-Zeit von ***H*** lässt sich einfach bestimmen.

3. Blockiert-Zeiten

Noch ein Beispiel:

- Eine Task ***L*** mit niedriger Priorität belegt eine Ressource ***R***
- Eine Task ***M*** mit mittlerer Priorität unterbricht ***L*** (Preemption)
- Eine Task ***H*** mit hoher Priorität unterbricht ***M***
- Task ***H*** fordert ***R*** an und blockiert, so dass ***M*** weiter macht

➔ ***H*** muss darauf warten, dass ***M*** fertig wird.

In dieser Situation muss eine hoch priorisierter Task ***(H)*** auf eine von ihr **völlig unabhängigen** niedriger priorisierten Task ***(M)*** warten. Ohne weitere Vorkehrungen ist es allerdings nicht möglich, hierfür die Blockiert-Zeit von ***H*** zu ermitteln.

Im Folgenden werden drei verschiedene Möglichkeiten vorgestellt, wie man das Problem lösen kann.

3. Blockiert-Zeiten

Ein komplexes Beispiel [1]:

Tasks: `a`, `b`, `c` und `d`; Ressourcen: `Q` und `V`

Task	Priority	Execution Sequence	Release Time
d	4	EEQVE	4
c	3	EVVE	2
b	2	EE	2
a	1	EQQQE	0

Aufgabe: Stellen Sie den Ablauf über der Zeitachse dar!

3. Blockiert-Zeiten

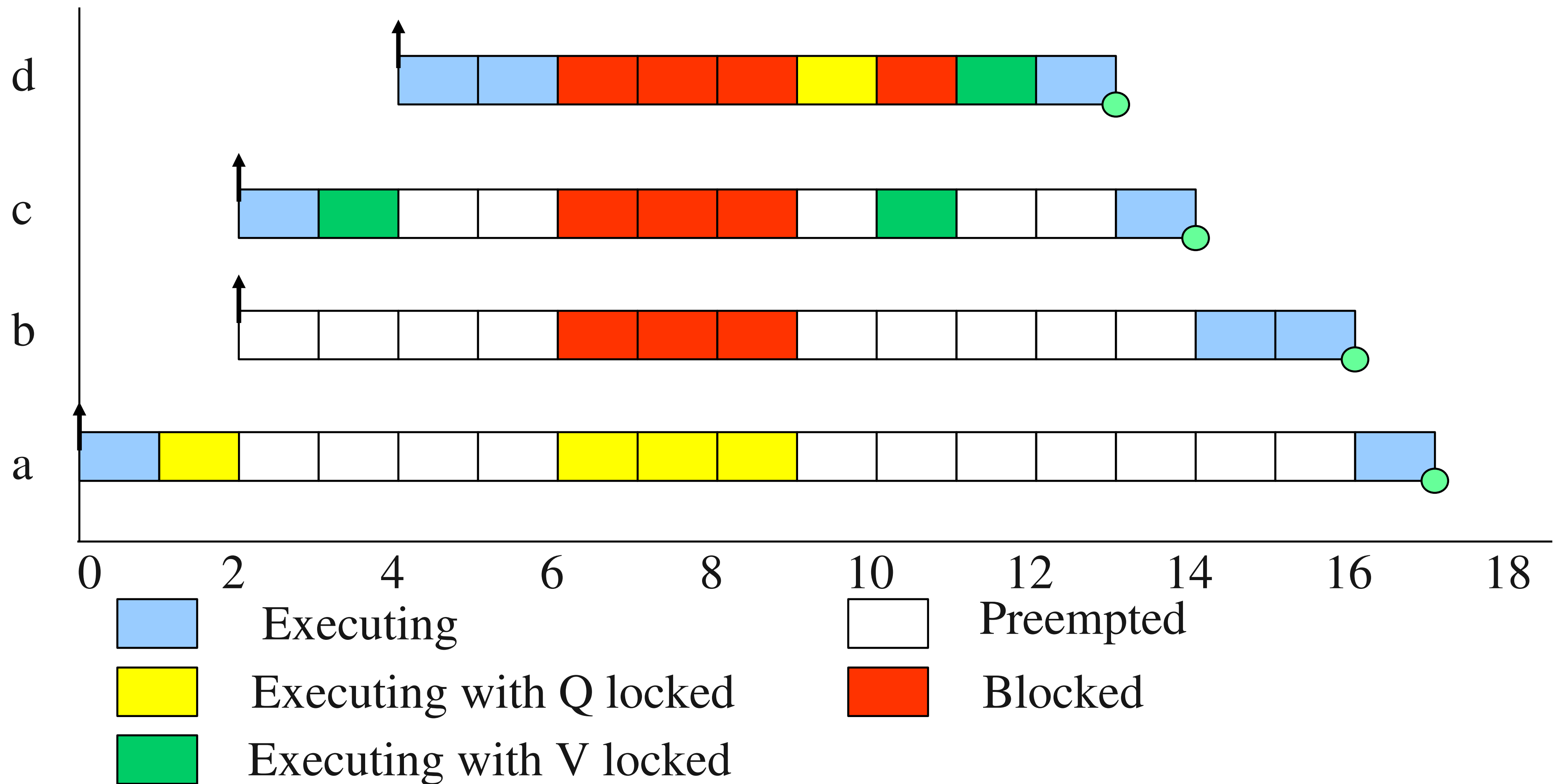
3.1 Priority Inheritance Protocol (PIP)

Bei diesem einfachen Ansatz wird die Priorität des blockierten Prozesses vererbt (inherited):

- Der Scheduler sorgt dafür, dass, wenn Task ***L*** die Ressource belegt, er die Priorität von ***H*** bekommt, sobald ***H*** sie anfordert. Gibt ***L*** die Ressource wieder frei, erhält er seine normale Priorität zurück.
- Also kann ***L*** nicht von ***M*** unterbrochen werden. - ***M*** muss nun allerdings auf ***L*** warten, obwohl seine Priorität eigentlich höher war, was aber in der Analyse zu berücksichtigen ist.
- Die Prioritätsvererbung ist transitiv: Bei 3 Tasks A, B, C mit fallenden Prioritäten, bekommt C die Priorität von A, wenn A durch B und B durch C blockiert wird.

3. Blockiert-Zeiten

Beispiel für Priority Inheritance [1]:



3. Blockiert-Zeiten

Bemerkungen:

- Gibt es **`m`** kritische Abschnitte (d.h. Semaphoren), kann eine Task maximal **`m`**-mal blockiert werden. Die Blockiert-Zeit B_i ergibt sich also i.W. aus der Summe der Ausführungszeiten der kritischen Abschnitte:

$$B_i = \sum_{k=1}^{m} usage(k, i) C(k) \qquad \text{mit}$$

  - **`usage(k, i) = 1`**, falls der kritische Abschnitt **`k`** benutzt wird von
    - mindestens einer Task mit niedrigerer Priorität als Task **`i`** **und**
    - mindestens einer Task mit höherer oder gleich hoher Priorität wie Task **`i`**.
  - **`usage(k, i) = 0`**, sonst
  - **`C(k)`** ist die maximale Ausführungszeit des kritischen Abschnitts **`k`**.
  - **Aufgabe**: Berechnen Sie B_b (aus dem Beispiel von eben)!
- Dabei kann es zu **Deadlocks** kommen!
  - **Aufgabe**: Geben Sie ein Beispiel dafür an!

3. Blockiert-Zeiten

3.2 Priority Ceiling Protocol

Das Priority Ceiling Protocol (PCP) – manchmal auch Ceiling Priority Protocol genannt – minimiert die Blockiert-Zeiten der Tasks und verhindert Deadlocks. Das Prinzip dabei ist wie folgt (***L*** und ***H*** wie eben):

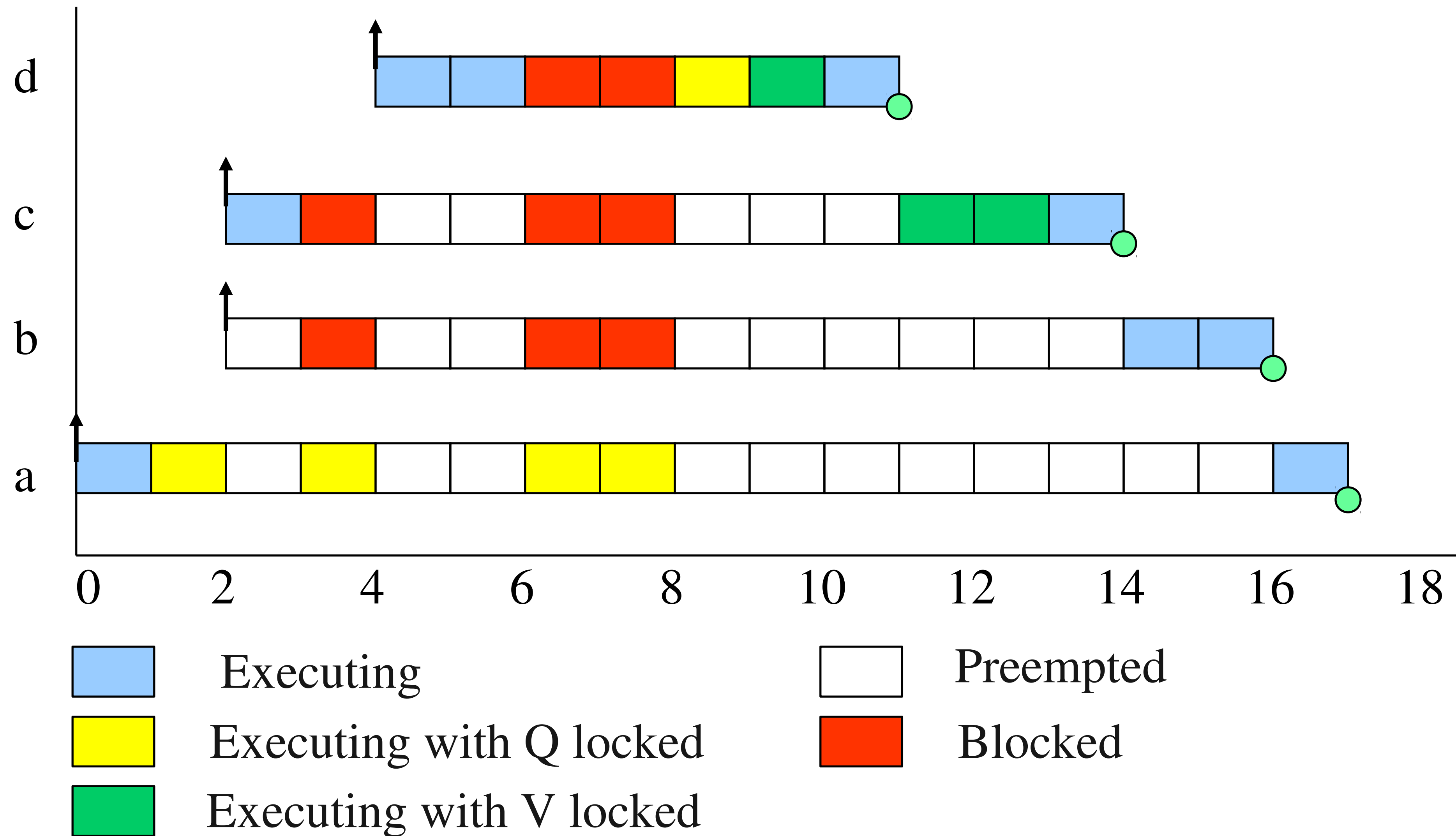
- Wenn ***L*** eine Ressource (d.h. einen kritischen Abschnitt) belegt (d.h. gesperrt) hat, die ***H*** blockieren könnte, dann darf keine weitere Ressource, auf die ***H*** zugreifen möchte, durch eine weitere Task gesperrt werden (außer durch ***L***).

Im Detail lässt sich dies wie folgt realisieren:

- Jede Ressource bekommt eine Ceiling Priority, welche die maximale Priorität der Tasks, die sie nutzen wollen, darstellt.
- Jede Task erhält zusätzlich eine dynamische Priorität: dem Maximum aus
  - ihrer statischen Priorität und
  - einer vererbten Priorität, die sich aus dem Blockieren von höher priorisierten Tasks ergibt
- Eine Task bekommt nur dann eine Ressource, wenn ihre dynamische Priorität höher ist als die aktuelle Ceiling Priority aller bislang benutzten Resourcen (mit Ausnahme der selbst reservierten).

3. Blockiert-Zeiten

Beispiel für das ***Original*** *PCP* [1]:



3. Blockiert-Zeiten

Bemerkungen:

- Das Sperren der ersten angeforderten Ressource ist also immer erlaubt, sofern sie frei ist. Auf eine zweite Ressource kann nur zugegriffen werden, wenn es keine höher priorisierte Task gibt, die diese benötigt.
- Im Vergleich zum PIP ändert sich RTA dadurch nicht. Allerdings ergeben sich andere Werte für die Blockiert-Zeiten.
- Eine Task kann also maximal einmal durch eine niedrig priorisierte Task blockiert werden. Entsprechend ist die Blockiert-Zeit B_i de Ausführungszeit des längsten kritischen Abschnitts, den sich Task **i** und eine niedrig priorisierte Task teilen:

$$B_i = \max_{k=1}^{m} usage(k, i)\, C(k)$$

3. Blockiert-Zeiten

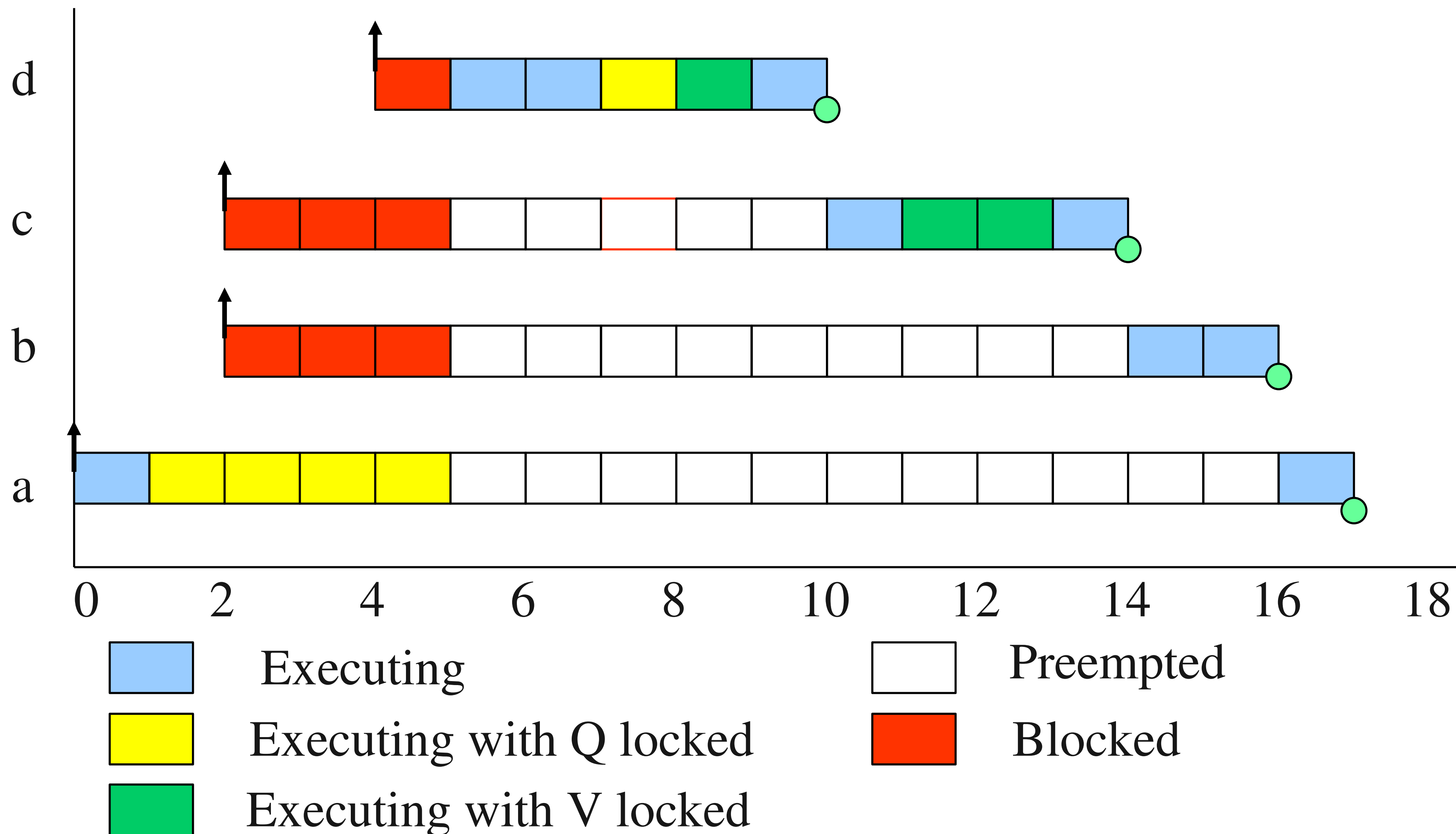
3.2 Immediate Priority Ceiling Protocol (IPCP)

Das originale PCP ist sehr schwierig zu implementieren und hat einen hohen Aufwand. Eine Vereinfachung/Verbesserung wird erziehlt, wenn die Priorität einer Task sofort (**immediate**) angehoben wird, wenn eine Ressource gesperrt wird. Im Detail lässt sich dies wie folgt realisieren:

- Jede Ressource bekommt eine Ceiling Priority, welche die maximale Priorität der Tasks, die sie nutzen wollen, darstellt.
- Jede Task erhält zusätzlich eine dynamische Priorität, dem Maximum seiner statischen Priorität und der Ceiling Priorities seiner bislang reservierten Ressourcen. Diese wird vom Scheduler verwendet.

3. Blockiert-Zeiten

Beispiel für Immediate Priority Ceiling [1]:



3. Blockiert-Zeiten

Bemerkungen:

- Falls beim IPCP eine Task überhaupt durch den Einfluss niedrig priorisierter Tasks blockiert, so wird dies zum Beginn seiner Laufzeit sein.
- An der RTA und der Berechnung der Blockiert-Zeiten ändert sich im Vergleich zum Original-Verfahren nichts.
- Die beiden Ceiling-Protokolle sind ein allgemeines Mittel zur Deadlock-Vermeidung!

Aufgabe: Berechnen Sie die Blockiert-Zeiten für das o.g. Beispiel!

3. Blockiert-Zeiten

Beispiel für die Deadlock-Vermeidung von IPCP:

```
Task H
    do_something();
    down(S1)
    down(S2)
    do_something_critical();
    up(S2)
    up(S1);
```

```
Task L
    do_something();
    down(S2)
    down(S1)
    do_something_critical();
    up(S1)
    up(S2);
```

Bei der Anwendung von PIP kann sich ein Deadlock ergeben, bei IPCP ist dies hingegen ausgeschlossen.

Aufgabe: Begründen Sie dies für beide Fälle!

4. Behandlung von Interrupt-Routinen

Interrupts (Unterbrechungen) werden von einem Gerät ausgelöst und führen dazu, dass eine entsprechende Interrupt-Routine (Behandlungsroutine) durch das Betriebssystem ausgeführt wird:

- Dazu wird die CPU (falls möglich) in den Kern-Modus umgeschaltet und während dieser Zeit werden keine weiteren Interrupts entgegen genommen.
- Die Zeit, die ein System braucht um Interrupts zu verarbeiten (d.h. die Zeitspanne bis zum Aufruf der Behandlungsroutine), wird **Interrupt Latency** genannt.
- Interrupt-Routinen haben üblicherweise eine höhere Priorität als andere Tasks.
- In Interrupt-Routinen wird oft ein gemeinsam genutzter Puffer benutzt, um Daten mit Tasks auszutauschen, welcher dadurch geschützt wird, dass
  - Interrupts unterbunden werden (**disable**), bevor auf die Daten zugegriffen wird und
  - Interrupts wieder erlaubt werden (**enable**), wenn man damit fertig ist.
- Dies ist wiederum nicht anderes, als ein **binäres Semaphor** zu benutzen!

4. Behandlung von Interrupt-Routinen

Bemerkung: In der Tat implementiert ein Betriebssystem ein Semaphor dadurch, während der Semaphor-Operationen die Interrupts zu unterbinden.

Schlussfolgerungen:

- **Interrupt-Routinen** lassen sich als hoch priorisierte Task mit einer bestimmten Periodenlänge modellieren und so in die Analyse einbringen
- Der **Disable/Enable**-Interupt-Mechanismus lässt sich in Blockiert-Zeiten umsetzen.

5. Release Jitter

Beim einfachen Modell wurde bisher angenommen, dass die Task immer zum Perioden-Beginn in den Zustand „bereit" versetzt wird (d.h. Release Time = 0). Nicht-periodische Tasks können modelliert und in die Analyse einbezogen werden, indem für sie eine Mindest-Periodenlänge angenommen wird.

Sollte allerdings eine Task erst starten können, wenn das Ergebnis einer anderen Task (auf einem anderen Netzwerk-Knoten) vorliegt, kann die Release-Time variieren, wie das folgende **Beispiel** zeigt:

Task	T	D	C	R
H	30	20	10	10
L	1000	25	15	25

Beim einfachen Modell wird **Task H** zu den Zeitpunkten **0, T, 2T, 3T** usw. in den Zustand „bereit" versetzt. Normalerweise halten beide Tasks ihre Deadlines ein.

Aufgabe: Was passiert aber, wenn sich die Aktivierung von H zum Zeitpunkt 990 um 10 verzögert?

5. Release Jitter

Definition: **Release Jitter J_i** ist der Unterschied zwischen dem frühesten und spätesten Zeitpunkt, an dem eine Task `i` ihre Ausführung beginnt (relativ zum Perioden-Beginn):

$$J_i = J_i^{biggest} - J_i^{smallest}$$

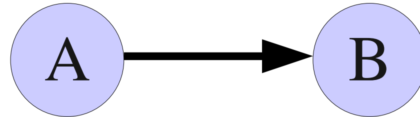
Die RTA wird wie folgt erweitert:

$$w_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j$$

$$R_i = w_i + J_i^{biggest}$$

6. Präzedenzen

Eine **Präzedenz** bedeutet, dass zwei **Tasks A** und **B** derart in Verbindung stehen, dass **B** erst dann mit ihrer Arbeit beginnen kann, wenn **A** fertig ist:



Es gibt i.W. zwei Möglichkeiten Präzedenzen **beim Ablauf** zu erzwingen:

- durch einen Synchronisationsmechanismus:
  - Message Passing: Operationen send() und receive()
  - Ein binäres Semaphor / Mutex je Abhängigkeit: nur für Single-Prozessor-Systeme

  jeweils zu Beginn und Ende einer Task

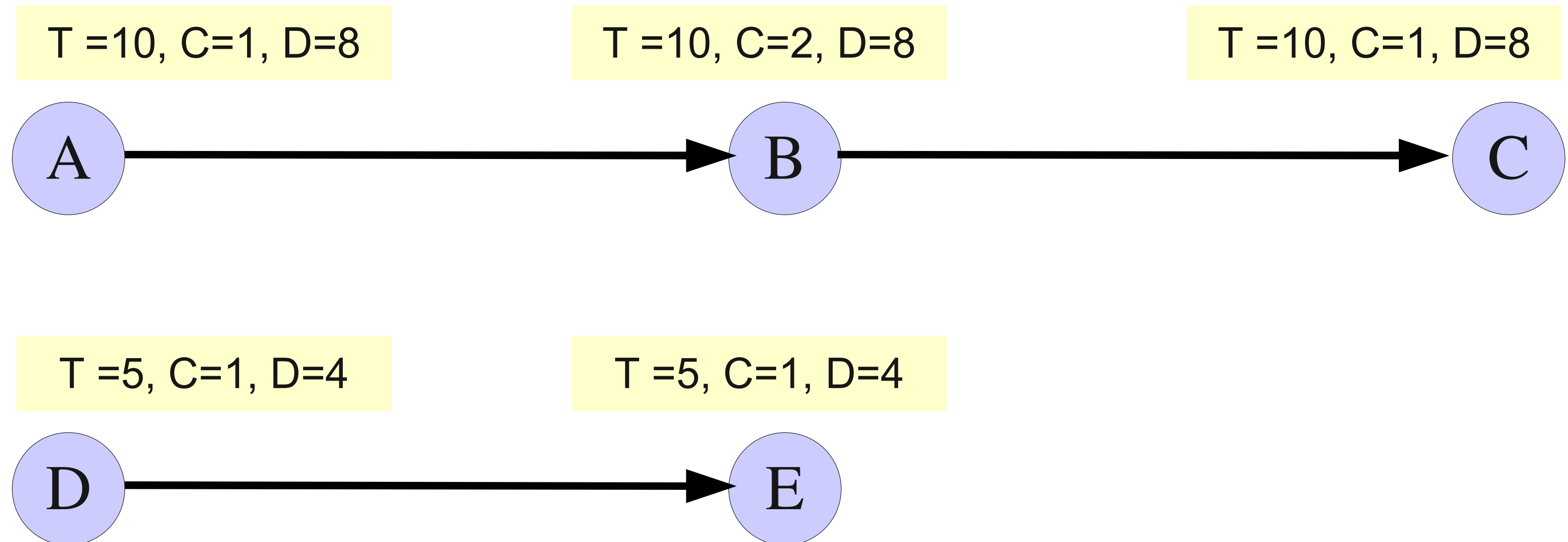
  [**Bemerkung**: ggf. gibt es dann keine Blockiert-Zeiten mehr.]
- durch die Vergabe von Prioritäten
  - **A** bekommt eine höhere Priorität als **B**
  - nur für Single-Prozessor-Systeme

6. Präzedenzen

Bemerkungen:

- Bei Präzedenzen, müssen die in Abhängigkeit stehenden Tasks natürlich dieselbe Periode haben.
- Um die **RTA anzuwenden**, reicht in beiden Fällen der Trick mit der Prioritätsvergabe (nur für Single-Prozessor-Systeme).

Beispiel [2]:



Aufgabe:

- Definieren Sie die Prioritäten so, dass die Präzedenzen eingehalten werden!
- Zeigen Sie, dass **Task C** und **E** ihre Deadlines einhalten!

6. Präzedenzen

Noch einfacher ist es **für die Analyse**, die von Präzedenzen betroffenen Tasks zusammen zu fassen. Im Beispiel von eben erhält man zwei virtuelle Tasks:

Task	T	D	C
T_{ABC}	10	8	4
T_{DE}	5	4	2

Je nach Deadlines, ist ein solcher Ansatz aber nicht immer möglich.

7. Offsets

Bisher wurde immer angenommen, dass alle Tasks dieselbe Release Time haben – und zwar zum Zeitpunkt 0 – woraus die Existenz des kritischen Zeitpunkts (ebenfalls 0) folgt. Liegt die Release Time einer Task aber ***grundsätzlich*** bei einem späteren Zeitpunkt (genannt **Offset, *O***), so ist diese Annahme nicht mehr gültig.
Das Problem ist dargestellt an folgendem Beispiel [1]:

Task	T	D	C	R	offset	R(neu)
a	8	5	4	4	0	4
b	20	10	4	8	0	8
c	20	12	4	16	10	8

Betrachtet man **Task c** (mit niedrigster Priorität) ohne Offset, so erhält man eine Response Time von 16, wobei dessen Deadline also verpasst wird.
Aufgabe: Stellen Sie die Situation dar, die sich ergibt, wenn **Task c** einen Offset von 10 erhält! - Achtung: Deadline und Response Time sind relativ zum Offset zu verstehen!

7. Offsets

Bemerkungen:

- Es ist schwierig (d.h. NP-hard [1]), den kritischen Zeitpunkt für diese Problemstellung überhaupt zu finden (anders als beim Release Jitter, wo eine Verzögerung eintreten kann aber nicht muss und sich **R** und **D** nach wie vor auf den Periodenbeginn beziehen).
- Insgesamt lassen sich allenfalls Näherungslösungen in die RTA einbauen, so dass Offset in dieser Veranstaltung nicht näher erörtert werden
- Die Ursache von Offsets sind aber oft Präzedenzen, so dass das Problem häufig mit denen im vorherigen Abschnitt vorstellten Lösungen adressierbar ist

8. Betriebssystem-Overhead

Um die RTA insgesamt anwendbar zu machen, muss der **Overhead**, der durch die Handlungen des Betriebssystems (RTOS) anfällt, mit in die Analyse einfließen [2]. Ein RTOS wird betrieben durch einen periodische **Prozessor-Takt (Tick)**.

- Ticks können modelliert werden als eine hoch priorisierte Task mit Aufwand C_{CLK} und und Periode T_{CLK}.
- Neben dem Tick existiert noch Overhead für die Aktivierung einer Task (d.h. für den Zustandsübergang von „bereit“ nach „rechnend“): C_{queue}
- Schließlich entsteht noch Aufwand für den Task-Wechsel, d.h. das Sichern des Zustandes (Register-Werte etc.) der laufenden Task und das Einlagern der neuen Task: C_{SW}

Einlagern — Task-Wechsel bei Störung — Zustandsübergänge — Ticks

$$w_i^{n+1} = C_i + C_{SW} + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil (C_j + 2C_{SW}) + \sum_{\forall j \in alltasks} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_{queue} + \left\lceil \frac{w_i^n}{T_{CLK}} \right\rceil C_{CLK}$$

$$R_i = w_i$$

8. Betriebssystem-Overhead

Beispiel [2] mit zwei Tasks A und B:

Aufgabe	Periode	Aufwand
C_{CLK}	1000	30
C_{SW}	-	50
C_{queue}	-	5
A	2000	1000
B	5000	3000

$$w_B^0 = C_i + 2C_{SW} = 3100$$

$$w_B^1 = 3000 + 100 + \lceil \frac{3100}{2000} \rceil (1000 + 100) + \lceil \frac{3100}{2000} \rceil 5 + \lceil \frac{3100}{5000} \rceil 5 + \lceil \frac{3100}{1000} \rceil 30 = 5385$$

$$w_B^2 = 3000 + 100 + \lceil \frac{5385}{2000} \rceil (1000 + 100) + \lceil \frac{5385}{2000} \rceil 5 + \lceil \frac{5385}{5000} \rceil 5 + \lceil \frac{5385}{1000} \rceil 30 = 6555$$

$$w_B^3 = 3000 + 100 + \lceil \frac{6555}{2000} \rceil (1000 + 100) + \lceil \frac{6555}{2000} \rceil 5 + \lceil \frac{6555}{5000} \rceil 5 + \lceil \frac{6555}{1000} \rceil 30 = 7690$$

$$w_B^4 = 3000 + 100 + \lceil \frac{7690}{2000} \rceil (1000 + 100) + \lceil \frac{7690}{2000} \rceil 5 + \lceil \frac{7690}{5000} \rceil 5 + \lceil \frac{7690}{1000} \rceil 30 = 7750$$

$$w_B^5 = 3000 + 100 + \lceil \frac{7750}{2000} \rceil (1000 + 100) + \lceil \frac{7750}{2000} \rceil 5 + \lceil \frac{7750}{5000} \rceil 5 + \lceil \frac{7750}{1000} \rceil 30 = 7750$$

$$R_B = 7750$$

9. Beliebige Deadlines

Beliebige Deadlines bedeutet, dass gelten darf **D > T**. Dies kann man zwar in die RTA aufnehmen, jedoch wird die Analyse dadurch sehr kompliziert. Zudem ist es meistens möglich, ohne diese Mechanismus aus zu kommen.

Der Vollständigkeit halber [1]:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

$$R_i(q) = w_i^n(q) - qT_i$$

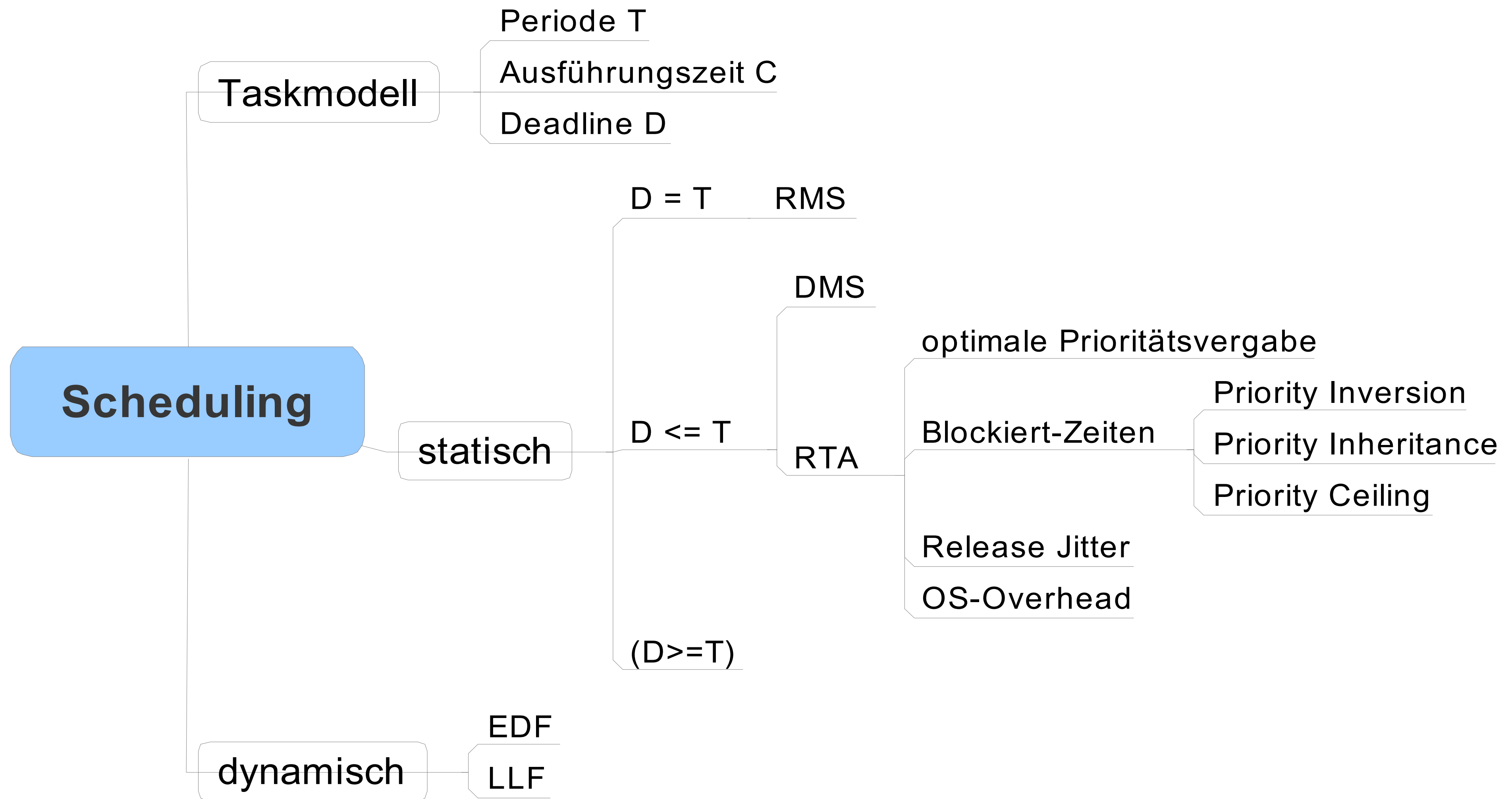
- The number of releases is bounded by the lowest value of q for which the following relation is true:

$$R_i(q) \leq T_i$$

- The worst-case response time is then the maximum value found for each q:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q)$$

8. Zusammenfassung



11. Case Study

Im Folgenden wird eine einfache Anwendung, die in C++ programmiert ist und POSIX-Threads benutzt, vorgestellt:

- Dabei wird ein autonomes Fahrzeug simuliert, das eine Linie folgt.
- Das Fahrzeug verfügt über einen Sensor, mit dem festgestellt werden kann, wie weit das Fahrzeug von der Linie entfernt ist.
- Der Lenkung wird dabei der Kurs (als Winkel in einem Koordinaten-System) mitgeteilt: Ist das Fahrzeug zu weit rechts, wird der Winkel nach links korrigiert und umgekehrt.

11. Case Study

Zur Erinnerung: So programmiert man in etwa einen Regelkreislauf:

```
void main (void){
  int state1, state2,…, staten;

  initializeClock();

  while(1){
    readSensors(…);

    if (CalculateNewSetValues(…))
        ActuateProcess(…);
    else
        Error();
    WaitForNextCycle(…);
  }
}
```

11. Case Study

Abstrakte Klasse ***Task***:

```
class Task {
public:
    Task(int);

protected:
    int T; // period in ms

public:
        friend void *timer_main( void * task);
        friend void *execute_main( void * task);
        virtual void execute() = 0;// abstract class

protected:
        /* Function to block the Process until next tick */
        void waitForNextCycle() { … }

};
```

11. Case Study

```
void *timer_main( void * task) {
...
}
void *execute_main( void * task) {
        cout << "Start execution ...." <<endl;
        Task *t = (Task *) task;

        t->execute();

        return NULL;
}
// Construktor starts threads:
Task::Task(int t=1000): T(t) {
        ...
        pthread_t timerthread;
        pthread_t executethread;

        pthread_create( &timerthread, NULL, &timer_main, (void *) this );
        pthread_create( &executethread, NULL, &execute_main, (void *) this );
};
```

11. Case Study

Fahrzeug auf dem Bildschirm anzeigen:

```
class Display : public Task {
public:
        Display(int T): Task(T) {};

        void execute();
};

void Display::execute() {
        cout << "... Display::execute T=" << T << endl;
        sem_post(& mutex); // don't start without me!

        while( true )
        {
                // critical region:
                sem_wait(& mutex);
                p.display("P= ");
                cout << "Abstand =" << (int) Abstand(p, Ldir) << endl << endl;

                sem_post(& mutex);

                waitForNextCycle();
        }
}
```

11. Case Study

Fahrzeug bewegen:

```
class Fahren : public Task {
public:
        Fahren(int T, int v, int d): Task(T), V(v), D(d) {};
        void execute();
private:
        int V; // speed
        int D; // direction
};
void Fahren::execute() {
        while( true ) {
                // simulate moving the vehicle:
                sem_wait(& mutex); // enter critical region
                // compute distance travelled during period
                double realV = (T * V) / 1000.0;
                p.move(realV, D);
                sem_post(& mutex); // end critical region
                // calculateNewSetValue() and actuateProcess():
                int entf = Abstand(p, Ldir);
                const int deltha = 1;
                if (entf > 0)
                        D += deltha;
                else if (entf < 0)
                        D -= deltha;
                waitForNextCycle();
        }
}
```

11. Case Study

Hauptprogramm:

```
int main( int argc, char *argv[] )
{
        InitAll(45); // direction of line [0, 90]

        sleep(1);


        Display D(...);

        cout << "Los geht's!!" << endl;
        Fahren F(/* period in ms */ ,
                 /* speed in pixels per second */ ,
                 /* start direction [0, 90] */ 0 );

        // sleep forever:
        sem_t block;
        sem_init( &block, 0, 0 );
        sem_wait( &block );

        return EXIT_SUCCESS;
}
```

11. Case Study

Aufgaben:

- Ermitteln Sie sinnvolle Werte für die Perioden der beiden Tasks
- Bestimmen Sie ein Task-Modell (Prioritäten, Blockierungen, etc.)
- Überlegen Sie sich sinnvolle Erweiterungsmöglichkeiten