

# Eingebettete Systeme

Prof. Dr. rer. nat. Peter Gerwinski

26. Juni 2025

# Eingebettete Systeme

<https://gitlab.cvh-server.de/pgerwinski/es>

- 1 Einführung**
- 2 Hardwarenahe Programmierung**
- 3 Einführung in Unix**
- 4 TCP/IP in der Praxis**
- 5 Bus-Systeme**
- 6 Echtzeit**
  - 6.1** Was ist Echtzeit?
  - 6.2** Echtzeitprogrammierung
  - 6.3** Multitasking
  - 6.4** Ressourcen
  - 6.5** Prioritäten

## 6.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Aber wie?

- ZigBee-Modul:  
Funk- vs. UART-Protokoll  
→ dedizierte Hardware

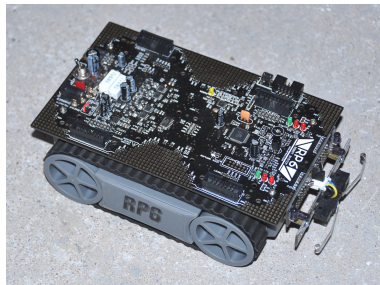


## 6.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Aber wie?

- ZigBee-Modul:  
Funk- vs. UART-Protokoll  
→ dedizierte Hardware
- RP6:  
Motorsteuerung vs. Anwender-Software  
→ spezielle Software



## 6.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Aber wie?

- ZigBee-Modul:  
Funk- vs. UART-Protokoll  
→ dedizierte Hardware
- RP6:  
Motorsteuerung vs. Anwender-Software  
→ spezielle Software
- Quadrocopter:  
Motorsteuerung vs. Sensoren-Abfrage  
vs. Funk-Fernsteuerung ...  
→ spezielle Software



## 6.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Aber wie?

- ZigBee-Modul:  
Funk- vs. UART-Protokoll  
→ dedizierte Hardware
- RP6:  
Motorsteuerung vs. Anwender-Software  
→ spezielle Software
- Quadrocopter:  
Motorsteuerung vs. Sensoren-Abfrage  
vs. Funk-Fernsteuerung ...  
→ spezielle Software
- Flugzeugkabinensimulatortür:  
Türsteuerung vs. Bedienung  
→ Echtzeitbetriebssystem



## 6.2 Echtzeitprogrammierung

Quadrocopter-Steuerung *MultiWii*

- Konfiguration durch bedingte Compilierung (Präprozessor)
- In der Hauptschleife wird 50mal pro Sekunde der RC-Task aufgerufen, ansonsten zyklisch einer von bis zu 5 weiteren Tasks.

## 6.2 Echtzeitprogrammierung

### Quadrocopter-Steuerung *MultiWii*

- Konfiguration durch bedingte Compilierung (Präprozessor)
- In der Hauptschleife wird 50mal pro Sekunde der RC-Task aufgerufen, ansonsten zyklisch einer von bis zu 5 weiteren Tasks.

### RP6-Steuerung

- Konfiguration durch bedingte Compilierung (Präprozessor)
- Lichtschranken an Encoder-Scheiben lösen bei Bewegung Interrupts aus. Die Interrupt-Handler zählen Variable hoch.
- 10000mal pro Sekunde: Timer-Interrupt  
Durch Zähler im Interrupt-Handler: verschiedene Taktraten  
1000mal pro Sekunde: Stopwatches  
5mal pro Sekunde: Blinkende Power-On-LED  
1000mal pro Sekunde: Bumper, ACS, PWM zur Motorsteuerung  
Geschwindigkeitsmessung durch Zählen der Ticks in 0.2 s  
Anpassung der Motorkraft in  $\pm 1$ -Schritten

## 6.2 Echtzeitprogrammierung

### Quadrocopter-Steuerung *MultiWii*

- Konfiguration durch bedingte Compilierung (Präprozessor)
- In der Hauptschleife wird 50mal pro Sekunde der RC-Task aufgerufen, ansonsten zyklisch einer von bis zu 5 weiteren Tasks.

### RP6-Steuerung

- Konfiguration durch bedingte Compilierung (Präprozessor)
- Lichtschranken an Encoder-Scheiben lösen bei Bewegung Interrupts aus. Die Interrupt-Handler zählen Variable hoch.
- 10000mal pro Sekunde: Timer-Interrupt  
Durch Zähler im Interrupt-Handler: verschiedene Taktraten  
1000mal pro Sekunde: Stopwatches  
5mal pro Sekunde: Blinkende Power-On-LED  
1000mal pro Sekunde: Bumper, ACS, PWM zur Motorsteuerung  
Geschwindigkeitsmessung durch Zählen der Ticks in 0.2 s  
Anpassung der Motorkraft in  $\pm 1$ -Schritten
- Nebenbei: Benutzerprogramm

## 6.2 Echtzeitprogrammierung

### Quadrocopter-Steuerung *MultiWii*

- Konfiguration durch bedingte Compilierung (Präprozessor)
- In der Hauptschleife wird 50mal pro Sekunde der RC-Task aufgerufen, ansonsten zyklisch einer von bis zu 5 weiteren Tasks.

### RP6-Steuerung

- Konfiguration durch bedingte Compilierung (Präprozessor)
- Lichtschranken an Encoder-Scheiben lösen bei Bewegung Interrupts aus. Die Interrupt-Handler zählen Variable hoch.
- 10000mal pro Sekunde: Timer-Interrupt  
Durch Zähler im Interrupt-Handler: verschiedene Taktraten  
1000mal pro Sekunde: Stopwatches  
5mal pro Sekunde: Blinkende Power-On-LED  
1000mal pro Sekunde: Bumper, ACS, PWM zur Motorsteuerung  
Geschwindigkeitsmessung durch Zählen der Ticks in 0.2 s  
Anpassung der Motorkraft in  $\pm 1$ -Schritten
- Nebenbei: 1 Benutzerprogramm

## 6.3 Multitasking

- *Kooperatives Multitasking*  
Prozesse geben freiwillig Rechenzeit ab
- *Präemptives Multitasking*  
Das Betriebssystem unterbricht laufende Prozesse  
(englisch: *to pre-empt* – jemandem zuvorkommen)

## 6.3 Multitasking

- *Kooperatives Multitasking*  
Prozesse geben freiwillig Rechenzeit ab
- *Präemptives Multitasking*  
Das Betriebssystem unterbricht laufende Prozesse  
(englisch: *to pre-empt* – jemandem zuvorkommen)
- *Scheduler*  
Steuerprogramm, das Prozessen Rechenzeit zuteilt
- *Kontextwechsel*  
Umschalten zwischen zwei Prozessen
- *Round-Robin-Verfahren (Rundlauf)*  
Zuteilung von *Zeitschlitz*en auf einer *Zeitscheibe* an die Prozesse

## 6.3 Multitasking

- *Kooperatives Multitasking*  
Prozesse geben freiwillig Rechenzeit ab
- *Präemptives Multitasking*  
Das Betriebssystem unterbricht laufende Prozesse  
(englisch: *to pre-empt* – jemandem zuvorkommen)
- *Scheduler*  
Steuerprogramm, das Prozessen Rechenzeit zuteilt
- *Kontextwechsel*  
Umschalten zwischen zwei Prozessen
- *Round-Robin-Verfahren (Rundlauf)*  
Zuteilung von *Zeitschlitz*en auf einer *Zeitscheibe* an die Prozesse
- *Ausblick: Zuteilung von Rechenzeit = wichtiger Spezialfall  
allgemein: Zuteilung von Ressourcen*

# Beispiele für Multitasking

## Quadrocopter-Steuerung *MultiWii*

- Konfiguration durch bedingte Compilierung (Präprozessor)
- In der Hauptschleife wird 50mal pro Sekunde der RC-Task aufgerufen, ansonsten zyklisch einer von bis zu 5 weiteren Tasks.

## RP6-Steuerung

- Konfiguration durch bedingte Compilierung (Präprozessor)
- Lichtschranken an Encoder-Scheiben lösen bei Bewegung Interrupts aus. Die Interrupt-Handler zählen Variable hoch.
- 10000mal pro Sekunde: Timer-Interrupt  
verschiedene Tasks werden unterschiedlich häufig aufgerufen
- Nebenbei: 1 Benutzerprogramm

# Beispiele für Multitasking

## Quadrocopter-Steuerung *MultiWii*

- Konfiguration durch bedingte Compilierung (Präprozessor)
- In der Hauptschleife wird 50mal pro Sekunde der RC-Task aufgerufen, ansonsten zyklisch einer von bis zu 5 weiteren Tasks.

## RP6-Steuerung

- Konfiguration durch bedingte Compilierung (Präprozessor)
- Lichtschranken an Encoder-Scheiben lösen bei Bewegung Interrupts aus. Die Interrupt-Handler zählen Variable hoch.
- 10000mal pro Sekunde: Timer-Interrupt  
verschiedene Tasks werden unterschiedlich häufig aufgerufen
- Nebenbei: 1 Benutzerprogramm

## Linux 0.01

- Timer-Interrupt: Zähler des aktuellen Tasks wird dekrementiert; Task mit höchstem Zähler bekommt Rechenzeit.
- Wenn es keinen lafbereiten Task mit positivem Zähler gibt, bekommen alle Tasks gemäß ihrer Priorität neue Zähler zugewiesen.
- *keine* harte Echtzeit

# Zombies

# Zombies

Wikipedia:

*Als Zombie wird die fiktive Figur eines zum Leben erweckten Toten (Untoter) oder eines seiner Seele beraubten, willenlosen Wesens bezeichnet. Der Begriff leitet sich von dem Wort nzùmbe aus der zentralafrikanischen Sprache Kimbundu ab und bezeichnet dort ursprünglich einen Totengeist.*

# Zombies

Wikipedia:

*Als Zombie wird die fiktive Figur eines zum Leben erweckten Toten (Untoter) oder eines seiner Seele beraubten, willenlosen Wesens bezeichnet. Der Begriff leitet sich von dem Wort nzùmbe aus der zentral-afrikanischen Sprache Kimbundu ab und bezeichnet dort ursprünglich einen Totengeist.*

Ein Zombie-Prozeß ist bereits beendet („tot“),  
bekommt keine Rechenzeit mehr („seiner Seele beraubt“),  
hat alle belegten Ressourcen wieder freigegeben („willenlos“),  
wird aber noch in der Prozeßliste geführt („untot“).

# Zombies

Wikipedia:

*Als Zombie wird die fiktive Figur eines zum Leben erweckten Toten (Untoter) oder eines seiner Seele beraubten, willenlosen Wesens bezeichnet. Der Begriff leitet sich von dem Wort nzùmbe aus der zentral-afrikanischen Sprache Kimbundu ab und bezeichnet dort ursprünglich einen Totengeist.*

Ein Zombie-Prozeß ist bereits beendet („tot“),  
bekommt keine Rechenzeit mehr („seiner Seele beraubt“),  
hat alle belegten Ressourcen wieder freigegeben („willenlos“),  
wird aber noch in der Prozeßliste geführt („untot“).

- Warum? —> Ein anderer Prozeß (Elternprozeß) wartet noch auf den Rückgabewert des beendeten Prozesses.

# Zombies

Wikipedia:

*Als Zombie wird die fiktive Figur eines zum Leben erweckten Toten (Untoter) oder eines seiner Seele beraubten, willenlosen Wesens bezeichnet. Der Begriff leitet sich von dem Wort nzùmbe aus der zentral-afrikanischen Sprache Kimbundu ab und bezeichnet dort ursprünglich einen Totengeist.*

Ein Zombie-Prozeß ist bereits beendet („tot“),  
bekommt keine Rechenzeit mehr („seiner Seele beraubt“),  
hat alle belegten Ressourcen wieder freigegeben („willenlos“),  
wird aber noch in der Prozeßliste geführt („untot“).

- Warum? → Ein anderer Prozeß (Elternprozeß) wartet noch auf den Rückgabewert des beendeten Prozesses.
- Schadet das? → Nein.

# Zombies

Wikipedia:

*Als Zombie wird die fiktive Figur eines zum Leben erweckten Toten (Untoter) oder eines seiner Seele beraubten, willenlosen Wesens bezeichnet. Der Begriff leitet sich von dem Wort nzùmbe aus der zentral-afrikanischen Sprache Kimbundu ab und bezeichnet dort ursprünglich einen Totengeist.*

Ein Zombie-Prozeß ist bereits beendet („tot“),  
bekommt keine Rechenzeit mehr („seiner Seele beraubt“),  
hat alle belegten Ressourcen wieder freigegeben („willenlos“),  
wird aber noch in der Prozeßliste geführt („untot“).

- Warum? → Ein anderer Prozeß (Elternprozeß) wartet noch auf den Rückgabewert des beendeten Prozesses.
- Schadet das? → Nein.
- Aber? → Wenn sich Zombie-Prozesse anhäufen, deutet dies auf einen Prozeß hin, der andere Prozesse erzeugt und anschließend „hängt“.

# Zombies

Wikipedia:

*Als Zombie wird die fiktive Figur eines zum Leben erweckten Toten (Untoter) oder eines seiner Seele beraubten, willenlosen Wesens bezeichnet. Der Begriff leitet sich von dem Wort nzùmbe aus der zentral-afrikanischen Sprache Kimbundu ab und bezeichnet dort ursprünglich einen Totengeist.*

Ein Zombie-Prozeß ist bereits beendet („tot“),  
bekommt keine Rechenzeit mehr („seiner Seele beraubt“),  
hat alle belegten Ressourcen wieder freigegeben („willenlos“),  
wird aber noch in der Prozeßliste geführt („untot“).

- Warum? → Ein anderer Prozeß (Elternprozeß) wartet noch auf den Rückgabewert des beendeten Prozesses.
- Schadet das? → Nein.
- Aber? → Wenn sich Zombie-Prozesse anhäufen, deutet dies auf einen Prozeß hin, der andere Prozesse erzeugt und anschließend „hängt“.
- Beispiel: Datenträger entfernt, zugreifender Prozeß „hängt“.  
→ Tochterprozesse werden zu Zombies.

## 6 Echtzeit

### 6.3 Multitasking

Qualitätsaspekte beim Multitasking

# 6 Echtzeit

## 6.3 Multitasking

Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*

# 6 Echtzeit

## 6.3 Multitasking

Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
  - *Latenz*: interaktive Anwendungen
  - *Jitter*: Echtzeitanwendungen
  - *Durchsatz*: Stapelverarbeitung

# 6 Echtzeit

## 6.3 Multitasking

Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*Mutexe*

# 6 Echtzeit

## 6.3 Multitasking

Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*Mutexe* (= spezielle *Semaphore*)

# 6 Echtzeit

## 6.3 Multitasking

Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*Mutexe* (= spezielle *Semaphore*)  
→ kommt gleich

# 6 Echtzeit

## 6.3 Multitasking

### Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*Mutexe* (= spezielle *Semaphore*)  
→ kommt gleich
- Verschiedene Methoden  
der Priorisierung  
→ später

# 6 Echtzeit

## 6.3 Multitasking

### Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*Mutexe* (= spezielle *Semaphore*)  
→ kommt gleich
- Verschiedene Methoden  
der Priorisierung  
→ später
- Umgehung der Probleme durch  
speziell geschriebene Software  
(MultiWii, RP6, ...)

# 6 Echtzeit

## 6.3 Multitasking

### Qualitätsaspekte beim Multitasking

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*Mutexe* (= spezielle *Semaphore*)  
→ kommt gleich
- Verschiedene Methoden  
der Priorisierung  
→ später
- Umgehung der Probleme durch  
speziell geschriebene Software  
(MultiWii, RP6, ...)

### Qualitätsaspekte für Netzwerke:

- Verschiedene Anforderungen:  
*Latenz* vs. *Jitter* vs. *Verluste*  
vs. *Durchsatz*
- Ressourcen reservieren:  
*IntServ* mit *Resource Reservation Protocol (RSVP)*
- Klassifizierung und Priorisierung:  
*DiffServ* mit Type-of-Service-Bits  
(IPv4) bzw. Traffic-Class-Bits (IPv6)  
im IP-Header
- Eigenes Protokoll (Telefondienste):  
*Asynchronous Transfer Mode (ATM)*

# 6 Echtzeit

## 6.4 Ressourcen

Ressourcen reservieren

- *Semaphor*  
gemeinsame Variable mehrerer Prozesse  
zur Regelung des Zugriffs auf eine Ressource  
Ressource belegt → Kontextwechsel  
  
griechisch: *sema* – Zeichen, *pherein* – tragen  
„Eisenbahnsignal“

# 6 Echtzeit

## 6.4 Ressourcen

### Ressourcen reservieren

- *Semaphor*  
gemeinsame Variable mehrerer Prozesse  
zur Regelung des Zugriffs auf eine Ressource  
Ressource belegt  $\longrightarrow$  Kontextwechsel
- *Mutex*  
Mechanismus, damit immer nur ein Prozeß gleichzeitig  
auf eine Ressource zugreifen kann  
  
englisch: *mutual exclusion* – wechselseitiger Ausschluß  
spezieller binärer Semaphor: nur „Besitzer“ darf freigeben

# 6 Echtzeit

## 6.4 Ressourcen

### Ressourcen reservieren

- *Semaphor*  
gemeinsame Variable mehrerer Prozesse  
zur Regelung des Zugriffs auf eine Ressource  
Ressource belegt  $\rightarrow$  Kontextwechsel
- *Mutex*  
Mechanismus, damit immer nur ein Prozeß gleichzeitig  
auf eine Ressource zugreifen kann
- *Spinlock (busy waiting)*  
leichtgewichtige Alternative zu Kontextwechsel  
englisch: *spin* – rotieren, *lock* Sperre  
*busy waiting* auf etwas Schnelles, z. B. auf einen Semaphor  
Hardware-Unterstützung: Prüfen, ob Variable bestimmten Wert hat;  
wenn ja, auf anderen Wert setzen; andere Prozessoren solange anhalten

# 6 Echtzeit

## 6.4 Ressourcen

### Ressourcen reservieren

- *Semaphor*  
gemeinsame Variable mehrerer Prozesse  
zur Regelung des Zugriffs auf eine Ressource  
Ressource belegt → Kontextwechsel
- *Mutex*  
Mechanismus, damit immer nur ein Prozeß gleichzeitig  
auf eine Ressource zugreifen kann
- *Spinlock (busy waiting)*  
leichtgewichtige Alternative zu Kontextwechsel
- *Kritischer Abschnitt – critical section*  
Programmabschnitt zwischen Reservierung  
und Freigabe einer Ressource  
→ sollte immer so kurz wie möglich sein

# 6 Echtzeit

## 6.4 Ressourcen

Ressourcen reservieren – Beispiel: `linux-3.7rc1`

- *Semaphor*  
kernel/semaphor.c  
drivers/usb/core/file.c
- *Mutex*  
kernel/mutex.c  
drivers/usb/serial/usb-serial.c
- *Spinlock*  
kernel/spinlock.c  
kernel/semaphor.c, kernel/mutex.c

**Beispiel:** `usb_serial_get_by_index()` – serielle Schnittstelle reservieren  
Datei `linux-3.7-rc1/drivers/usb/serial/usb-serial.c`, ab Zeile 62

```
struct usb_serial *usb_serial_get_by_index (unsigned index)
{
    struct usb_serial *serial;
    mutex_lock (&table_lock); ← exklusiven Zugriff auf Tabelle sichern
    serial = serial_table[index];
    if (serial)
    {
        mutex_lock (&serial->disc_mutex);
        if (serial->disconnected)
        {
            mutex_unlock (&serial->disc_mutex);
            serial = NULL;
        }
        else
            kref_get (&serial->kref);
    }
    mutex_unlock (&table_lock); ← exklusiven Zugriff auf Tabelle wieder freigeben
    return serial;
}
```

mutex\_lock() – Ressource beanspruchen, notfalls warten  
Datei linux-3.7-rc1/drivers/usb/serial/usb-serial.c, ab Zeile 62

```
void __sched mutex_lock (struct mutex *lock)
{
    might_sleep ();
    __mutex_fastpath_lock (&lock->count, __mutex_lock_slowpath);
    mutex_set_owner (lock);
}
```

Datei linux-3.7-rc1/arch/x86/include/asm/mutex\_32.h, ab Zeile 24  
Macro-Definition für \_\_mutex\_fastpath\_lock (expandiert)

Assembler:

```
lock dec (lock->count)
jns 1
call __mutex_lock_slowpath
1:
```

Datei linux-3.7-rc1/kernel/mutex.c, ab Zeile 398

```
static __used noline void __sched
__mutex_lock_slowpath (atomic_t *lock_count)
{
    struct mutex *lock = container_of (lock_count, struct mutex, count);
    __mutex_lock_common (lock, TASK_UNINTERRUPTIBLE, 0,
                        NULL, _RET_IP_);
}
```

Datei linux-3.7-rc1/kernel/mutex.c, ab Zeile 132

```
static inline int __sched
__mutex_lock_common (struct mutex *lock, long state, unsigned int subclass,
                    struct lockdep_map *nest_lock, unsigned long ip)
{
    struct task_struct *task = current;
    struct mutex_waiter waiter;
    unsigned long flags;

    preempt_disable ();
    mutex_acquire_nest (&lock->dep_map, subclass, 0, nest_lock, ip);

    /* ... */

    spin_lock_mutex (&lock->wait_lock, flags);

    debug_mutex_lock_common (lock, &waiter);
    debug_mutex_add_waiter (lock, &waiter, task_thread_info (task));

    /* add waiting tasks to the end of the waitqueue (FIFO): */
    list_add_tail (&waiter.list, &lock->wait_list);
    waiter.task = task;

    if (atomic_xchg (&lock->count, -1) == 1)
        goto done;

    lock_contended (&lock->dep_map, ip);

    for (;;)
    {
        /*
         * Lets try to take the lock again – this is needed even if
         * we get here for the first time (shortly after failing to
         * acquire the lock), to make sure that we get a wakeup once
         * it's unlocked. Later on, if we sleep, this is the
         * operation that gives us the lock. We xchg it to -1, so
         * that when we release the lock, we properly wake up the
         * other waiters:
         */
        if (atomic_xchg (&lock->count, -1) == 1)
            break;

        /*
         * got a signal? (This code gets eliminated in the
         * TASK_UNINTERRUPTIBLE case.)
         */
        if (unlikely (signal_pending_state (state, task)))
        {
            mutex_remove_waiter (lock, &waiter, task_thread_info (task));
            mutex_release (&lock->dep_map, 1, ip);
            spin_unlock_mutex (&lock->wait_lock, flags);

            debug_mutex_free_waiter (&waiter);
            preempt_enable ();
            return -EINTR;
        }
        __set_task_state (task, state);

        /* didn't get the lock, go to sleep: */
        spin_unlock_mutex (&lock->wait_lock, flags);
        schedule_preempt_disabled ();
        spin_lock_mutex (&lock->wait_lock, flags);
    }

done:
    lock_acquired (&lock->dep_map, ip);
    /* got the lock – rejoice! */
    mutex_remove_waiter (lock, &waiter, current_thread_info ());
    mutex_set_owner (lock);

    /* set it to 0 if there are no waiters left: */
    if (likely (list_empty (&lock->wait_list)))
        atomic_set (&lock->count, 0);

    spin_unlock_mutex (&lock->wait_lock, flags);

    debug_mutex_free_waiter (&waiter);
    preempt_enable ();

    return 0;
}
```

← exklusiven Zugriff auf Mutex sichern

← exklusiven Zugriff auf Mutex wieder freigeben

spin\_lock\_mutex() – Mutex beanspruchen, notfalls *busy waiting*  
Datei linux-3.7-rc1/kernel/mutex.h, ab Zeile 12

```
#define spin_lock_mutex(lock, flags) \
do \
{ \
    spin_lock (lock); \
    (void) (flags); \
} \
while (0)
```

Datei linux-3.7-rc1/kernel/spinlock.h, ab Zeile 283

```
static inline void spin_lock (spinlock_t *lock)
{
    raw_spin_lock (&lock->rlock);
}
```

Datei linux-3.7-rc1/kernel/spinlock.h, Zeile 170

```
#define raw_spin_lock(lock) _raw_spin_lock (lock)
```

Datei linux-3.7-rc1/include/linux/spinlock\_api\_smp.h, Zeile 47

```
#define _raw_spin_lock(lock) __raw_spin_lock (lock)
```

Datei linux-3.7-rc1/kernel/spinlock.c, ab Zeile 46 (expandiert):

```
void __lockfunc __raw_spin_lock (spinlock_t *lock)
{
    for (;;)
    {
        preempt_disable ();
        if (likely (do_raw_spin_trylock (lock)))
            break;
        preempt_enable ();

        if (!(lock)->break_lock)
            (lock)->break_lock = 1;
        while (raw_spin_can_lock (lock) && (lock)->break_lock)
            arch_spin_relax (&lock->raw_lock);
    }
    (lock)->break_lock = 0;
}
```

Datei linux-3.7-rc1/include/linux/spinlock.h, ab Zeile 150:

```
static inline int do_raw_spin_trylock (raw_spinlock_t *lock)
{
    return arch_spin_trylock (&(lock)->raw_lock);
}
```

Datei arch/x86/include/asm/spinlock.h, ab Zeile 116:

```
static __always_inline int arch_spin_trylock (arch_spinlock_t *lock)
{
    return __ticket_spin_trylock (lock);
}
```

Datei arch/x86/include/asm/spinlock.h, ab Zeile 65:

```
static __always_inline int __ticket_spin_trylock (arch_spinlock_t *lock)
{
    arch_spinlock_t old, new;

    old.tickets = ACCESS_ONCE (lock->tickets);
    if (old.tickets.head != old.tickets.tail)
        return 0;

    new.head_tail = old.head_tail + (1 << TICKET_SHIFT);

    /* cmpxchg is a full barrier, so nothing can move before it */
    return cmpxchg (&lock->head_tail, old.head_tail, new.head_tail) == old.head_tail;
}
```

Datei arch/x86/include/asm/cmpxchg.h, ab Zeile 147:

```
#define cmpxchg(ptr, old, new) \
__cmpxchg (ptr, old, new, sizeof (*(ptr)))
```

Datei arch/x86/include/asm/cmpxchg.h, ab Zeile 131:

```
#define __cmpxchg(ptr, old, new, size) \
__raw_cmpxchg ((ptr), (old), (new), (size), LOCK_PREFIX)
```

Datei arch/x86/include/asm/cmpxchg.h, ab Zeile 110:

```
asm volatile (lock "cmpxchgl_ %2,%1" \
: "=a" (__ret), "+m" (*__ptr) \
: "r" (__new), "0" (__old) \
: "memory");
```

atomarer und exklusiver  
Zugriff auf Spinlock  
durch Hardware-Unterstützung

# 6 Echtzeit

## 6.4 Ressourcen

### Ressourcen reservieren

- *Semaphor*  
gemeinsame Variable mehrerer Prozesse  
zur Regelung des Zugriffs auf eine Ressource  
Ressource belegt → Kontextwechsel
- *Mutex*  
Mechanismus, damit immer nur ein Prozeß gleichzeitig  
auf eine Ressource zugreifen kann
- *Spinlock (busy waiting)*  
leichtgewichtige Alternative zu Kontextwechsel
- *Kritischer Abschnitt – critical section*  
Programmabschnitt zwischen Reservierung  
und Freigabe einer Ressource  
→ sollte immer so kurz wie möglich sein

## 6 Echtzeit

### 6.4 Ressourcen

*Verklemmungen*: Gegenseitiges Blockieren von Ressourcen

- *Deadlock*: Prozeß wartet
- *Livelock*: Prozeß macht andere Dinge  
(z. B. *busy waiting*)

# 6 Echtzeit

## 6.4 Ressourcen

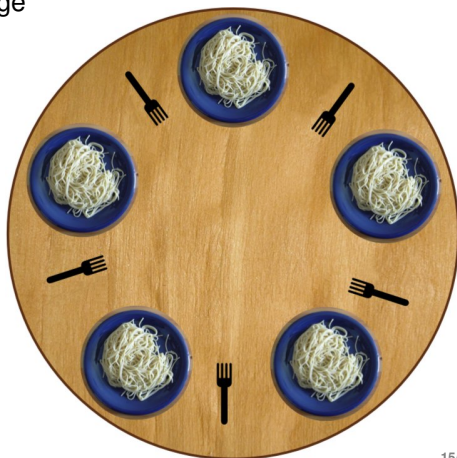
**Verklemmungen:** Gegenseitiges Blockieren von Ressourcen

- **Deadlock:** Prozeß wartet
- **Livelock:** Prozeß macht andere Dinge (z. B. *busy waiting*)

Beispiel: Philosophenproblem

- 5 Philosophen, 5 Gabeln
- 2 Gabeln zum Essen notwendig
- Wer essen will, nimmt eine Gabel und wartet notfalls auf die zweite.
- Keiner legt eine einzelne Gabel wieder zurück.

Jeder hält 1 Gabel → **Verklemmung**



# 6 Echtzeit

## 6.4 Ressourcen

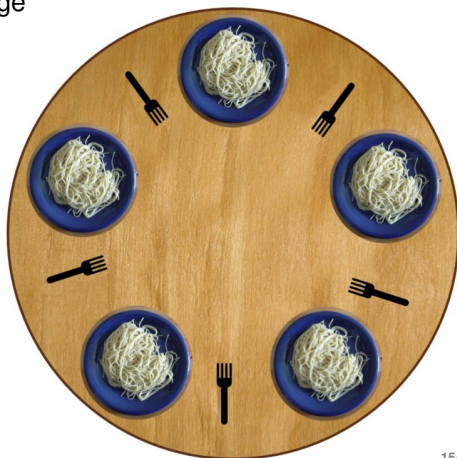
**Verklemmungen:** Gegenseitiges Blockieren von Ressourcen

- **Deadlock:** Prozeß wartet
- **Livelock:** Prozeß macht andere Dinge  
(z. B. *busy waiting*)

Beispiel: Philosophenproblem

- 5 Philosophen, 5 Gabeln
- 2 Gabeln zum Essen notwendig
- Wer essen will, nimmt eine Gabel und wartet notfalls auf die zweite.
- Keiner legt eine einzelne Gabel wieder zurück.

Jeder hält 1 Gabel → **Verklemmung**  
schweigen → **Deadlock**  
philosophieren weiter → **Livelock**



# 6 Echtzeit

## 6.4 Ressourcen

*Verklemmungen*: Gegenseitiges Blockieren von Ressourcen

- *Deadlock*: Prozeß wartet
- *Livelock*: Prozeß macht andere Dinge  
(z. B. *busy waiting*)

Beispiel: Philosophenproblem

Bedingungen für Verklemmungen:

- Exklusivität
- *hold and wait*
- Entzug nicht möglich
- zirkuläre Blockade

# 6 Echtzeit

## 6.4 Ressourcen

*Verklemmungen*: Gegenseitiges Blockieren von Ressourcen

- *Deadlock*: Prozeß wartet
- *Livelock*: Prozeß macht andere Dinge  
(z. B. *busy waiting*)

Beispiel: Philosophenproblem

Bedingungen für Verklemmungen:

- Exklusivität → Spooling
- *hold and wait* → simultane Zuteilung
- Entzug nicht möglich → Prozesse suspendieren, beenden, *Rollback*
- zirkuläre Blockade → Reihenfolge abhängig von Ressourcen

# 6 Echtzeit

## 6.5 Prioritäten

Linux 0.01

- Timer-Interrupt: Zähler des aktuellen Prozesses wird dekrementiert; Prozeß mit höchstem Zähler bekommt Rechenzeit.
- Wenn es keinen laufbereiten Prozeß mit positivem Zähler gibt, bekommen alle Prozesse gemäß ihrer *Priorität* neue Zähler zugewiesen.
- *keine* harte Echtzeit

→ *dynamische Prioritätenvergabe*:  
Rechenzeit hängt vom Verhalten des Prozesses ab

Echtzeitbetriebssysteme

- Prozesse können einen festen Anteil an Rechenzeit bekommen.
- Bei Ereignissen können Prozesse hoher Priorität Prozesse niedriger Priorität unterbrechen, aber nicht umgekehrt.

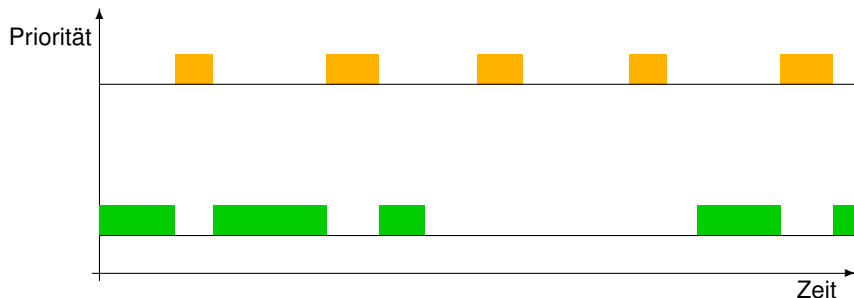
→ *statische Prioritätenvergabe*

# 6 Echtzeit

## 6.5 Prioritäten

### Echtzeitbetriebssysteme

- Prozesse können einen festen Anteil an Rechenzeit bekommen.
- Bei Ereignissen können Prozesse hoher Priorität Prozesse niedriger Priorität unterbrechen, aber nicht umgekehrt.

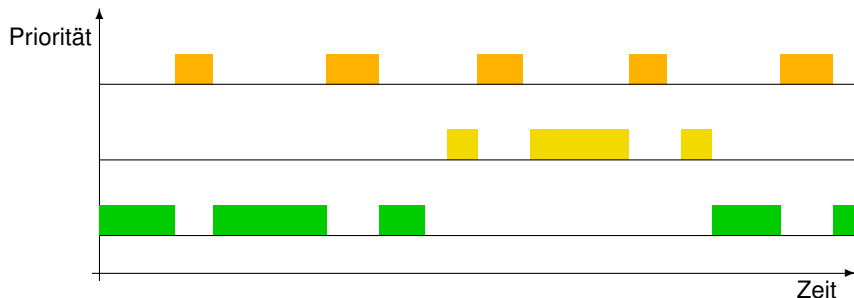


# 6 Echtzeit

## 6.5 Prioritäten

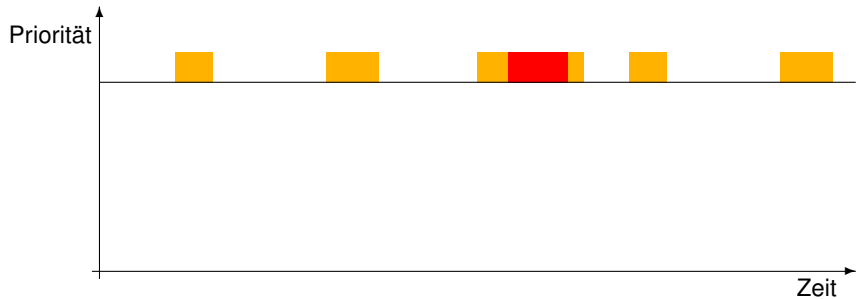
### Echtzeitbetriebssysteme

- Prozesse können einen festen Anteil an Rechenzeit bekommen.
- Bei Ereignissen können Prozesse hoher Priorität Prozesse niedriger Priorität unterbrechen, aber nicht umgekehrt.



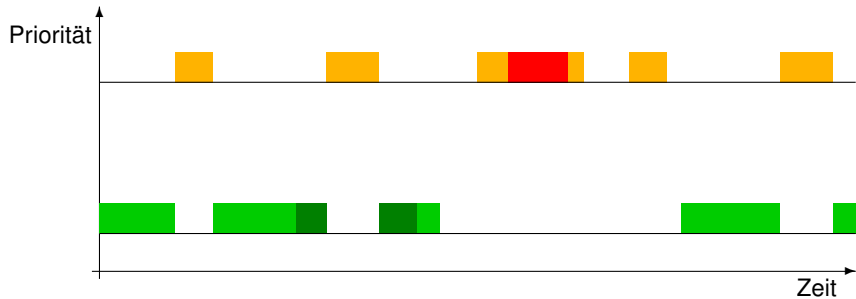
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen



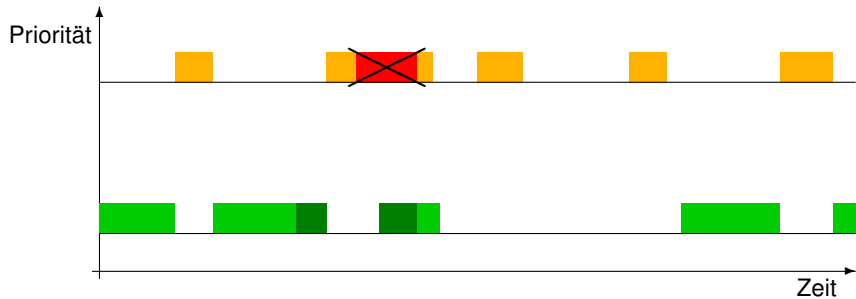
## 6 Echtzeit

## 6.5 Prioritäten und Ressourcen



## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen



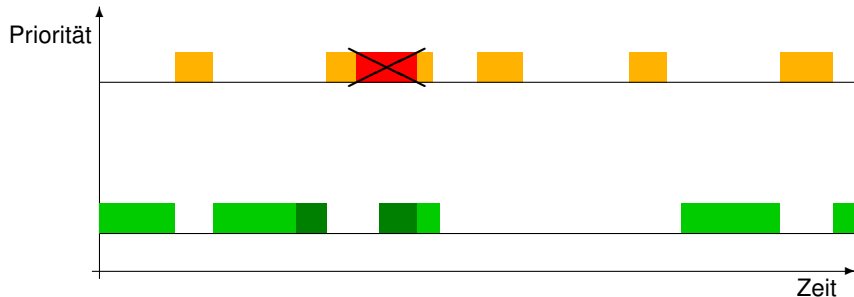
# 6 Echtzeit

## 6.5 Prioritäten und Ressourcen

Der höher priorisierte Prozeß bewirkt selbst, daß er eine Ressource verspätet bekommt.

→ *begrenzte Prioritätsinversion*

maximale Verzögerung: Länge des kritischen Bereichs



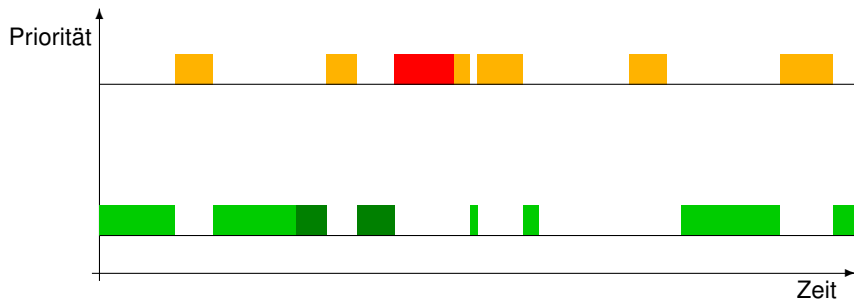
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Der höher priorisierte Prozeß bewirkt selbst, daß er eine Ressource verspätet bekommt.

→ *begrenzte Prioritätsinversion*

maximale Verzögerung: Länge des kritischen Bereichs



## 6 Echtzeit

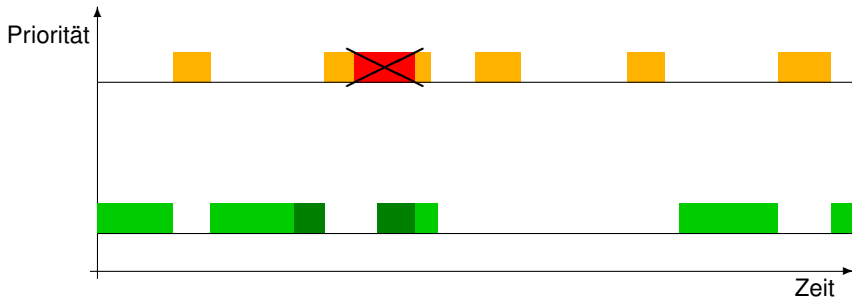
### 6.5 Prioritäten und Ressourcen

*unbegrenzte Prioritätsinversion*

## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

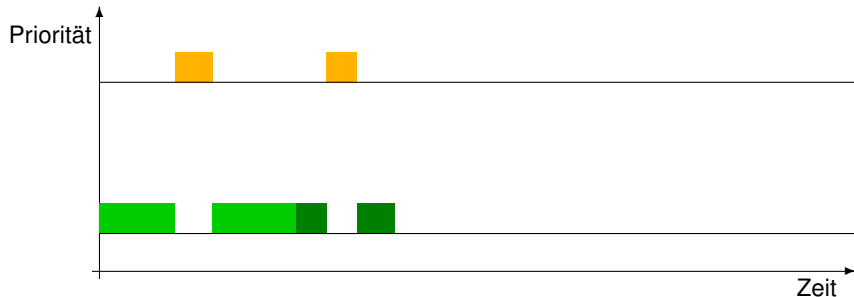
*unbegrenzte Prioritätsinversion*



## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

*unbegrenzte Prioritätsinversion*

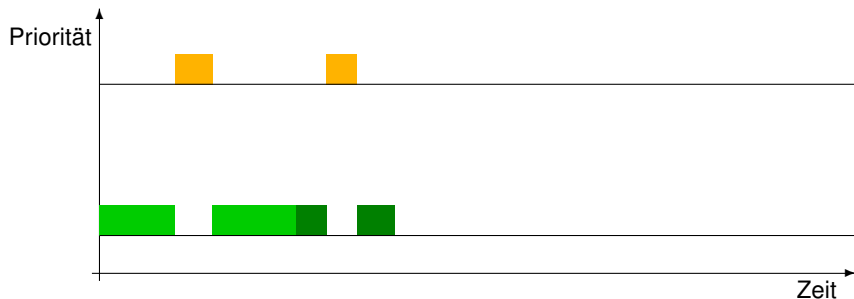


## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

→ *unbegrenzte Prioritätsinversion*

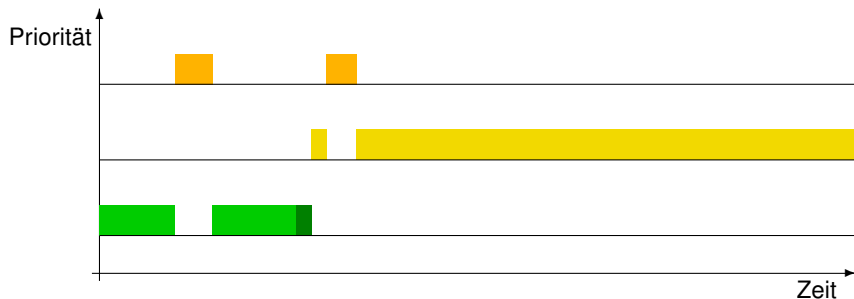


## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

→ *unbegrenzte Prioritätsinversion*



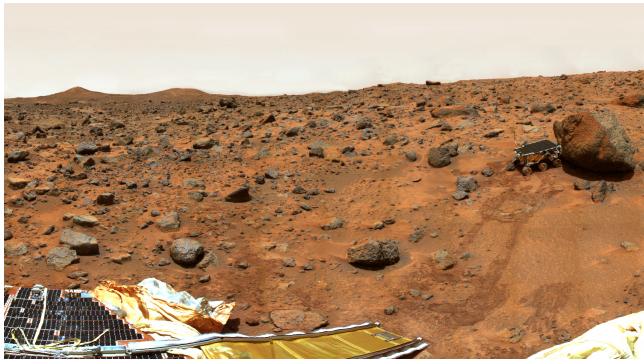
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

—→ *unbegrenzte Prioritätsinversion*

Beispiel: Beinahe-Verlust der Marssonde *Pathfinder* im Juli 1997



## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

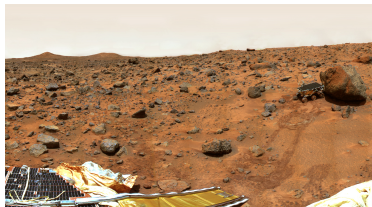
—→ *unbegrenzte Prioritätsinversion*

Beispiel: Beinahe-Verlust der Marssonde *Pathfinder* im Juli 1997

Gegenmaßnahmen

- *Priority Inheritance – Prioritätsvererbung*  
Der Besitzer des Mutex erbt die Priorität des Prozesses, der auf den Mutex wartet.

[http://research.microsoft.com/en-us/um/people/mbj/Mars\\_Pathfinder/](http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/)



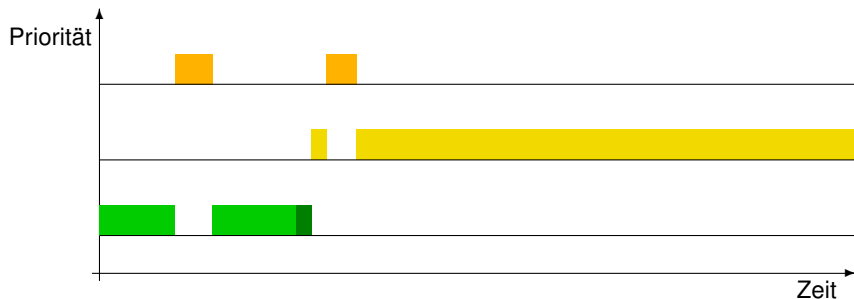
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

→ *unbegrenzte Prioritätsinversion*

Gegenmaßnahme: *Priority Inheritance – Prioritätsvererbung*



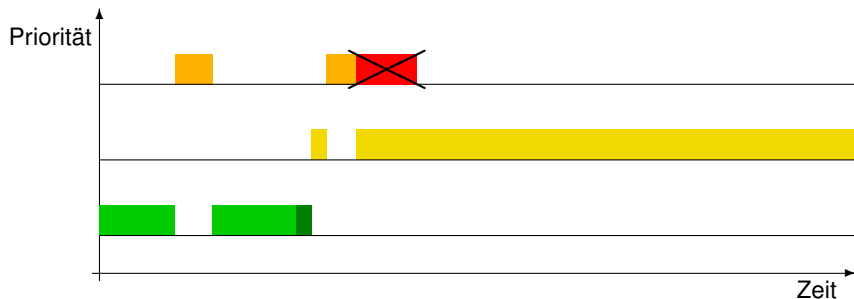
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

→ *unbegrenzte Prioritätsinversion*

Gegenmaßnahme: *Priority Inheritance – Prioritätsvererbung*



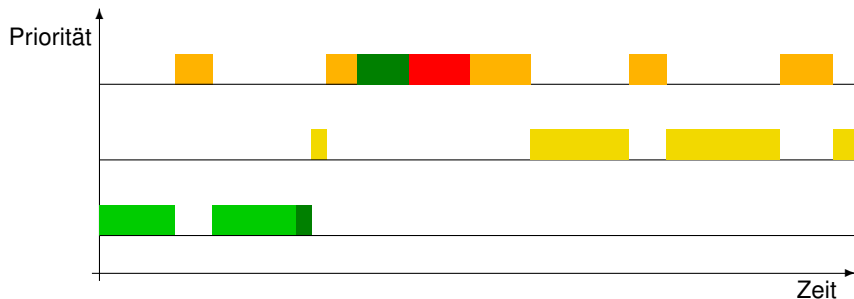
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

→ *unbegrenzte Prioritätsinversion*

Gegenmaßnahme: *Priority Inheritance – Prioritätsvererbung*



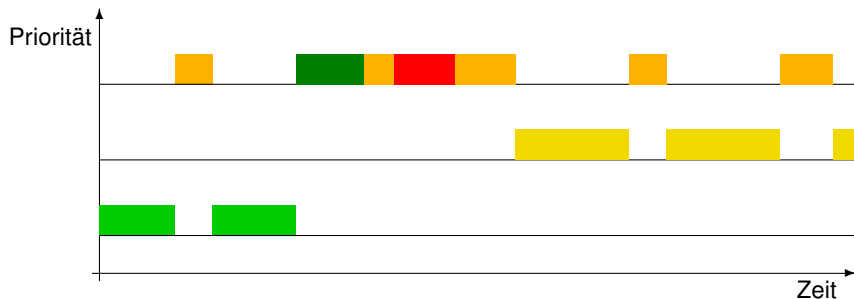
## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

→ *unbegrenzte Prioritätsinversion*

Gegenmaßnahme: *Priority Ceiling – Prioritätsobergrenze*



## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

—→ *unbegrenzte Prioritätsinversion*

Beispiel: Beinahe-Verlust der Marssonde *Pathfinder* im Juli 1997

Gegenmaßnahmen

- *Priority Inheritance – Prioritätsvererbung*  
Der Besitzer des Mutex erbt die Priorität des Prozesses, der auf den Mutex wartet.
- *Priority Ceiling – Prioritätsobergrenze*  
Der Besitzer des Mutex bekommt sofort die Priorität des höchstmöglichen Prozesses, der evtl. den Mutex benötigen könnte.

## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

—→ *unbegrenzte Prioritätsinversion*

Beispiel: Beinahe-Verlust der Marssonde *Pathfinder* im Juli 1997

Gegenmaßnahmen

- *Priority Inheritance – Prioritätsvererbung*  
Der Besitzer des Mutex erbt die Priorität des Prozesses, der auf den Mutex wartet.
  - *Priority Ceiling – Prioritätsobergrenze*  
Der Besitzer des Mutex bekommt sofort die Priorität des höchstmöglichen Prozesses, der evtl. den Mutex benötigen könnte.
- } nur möglich, wenn  
Mutexe im Spiel sind

## 6 Echtzeit

### 6.5 Prioritäten und Ressourcen

Ein Prozeß mit mittlerer Priorität bewirkt, daß ein Prozeß mit hoher Priorität eine Ressource überhaupt nicht bekommt.

—→ *unbegrenzte Prioritätsinversion*

Beispiel: Beinahe-Verlust der Marssonde *Pathfinder* im Juli 1997

Gegenmaßnahmen

- *Priority Inheritance – Prioritätsvererbung*  
Der Besitzer des Mutex erbt die Priorität des Prozesses, der auf den Mutex wartet.
  - *Priority Ceiling – Prioritätsobergrenze*  
Der Besitzer des Mutex bekommt sofort die Priorität des höchstmöglichen Prozesses, der evtl. den Mutex benötigen könnte.
  - *Priority Aging*  
Die Priorität wächst mit der Wartezeit.
- } nur möglich, wenn  
Mutexe im Spiel sind