

Treiberentwicklung, Echtzeit- und Betriebssysteme

Prof. Dr. rer. nat. Peter Gerwinski

13. Juni 2022

4 Echtzeit

4.1 Was ist Echtzeit?

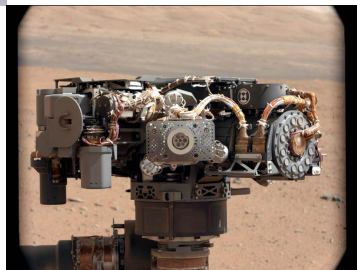
- Animation in Echtzeit:
schnelle Berechnung anstatt Wiedergabe einer Aufzeichnung
- Fantasy-Rollenspiel in Echtzeit:
Der Zeitverlauf der Spielwelt entspricht dem der realen Welt.
- Datenverarbeitung in Echtzeit:
Die Daten werden so schnell verarbeitet, wie sie anfallen.
- speziell: Echtzeit-Steuerung von Maschinen:
Die Berechnung kann mit den physikalischen Vorgängen schritthalten.

→ „Schnell genug.“

4.1 Was ist Echtzeit?

„Schnell genug.“ – „Und wenn nicht?“

- „Ganz schlecht.“ → *harte Echtzeit*



4.1 Was ist Echtzeit?

„Schnell genug.“ – „Und wenn nicht?“

- „Ganz schlecht.“ → *harte Echtzeit*
rechtzeitiges Ergebnis funktionsentscheidend

4.1 Was ist Echtzeit?

„Schnell genug.“ – „Und wenn nicht?“

- „Ganz schlecht.“ → *harte Echtzeit*
rechtzeitiges Ergebnis funktionsentscheidend
- „Unschön.“ → *weiche Echtzeit*



4.1 Was ist Echtzeit?

„Schnell genug.“ – „Und wenn nicht?“

- „Ganz schlecht.“ → *harte Echtzeit*
rechtzeitiges Ergebnis funktionsentscheidend
- „Unschön.“ → *weiche Echtzeit*
verspätetes Ergebnis qualitätsmindernd
 - verwenden und Verzögerung in Kauf nehmen
 - verwerfen und Ausfall in Kauf nehmen

4.1 Was ist Echtzeit?

„Schnell genug.“ – „Und wenn nicht?“

- „Ganz schlecht.“ → *harte Echtzeit*
rechtzeitiges Ergebnis funktionsentscheidend
- „Unschön.“ → *weiche Echtzeit*
verspätetes Ergebnis qualitätsmindernd
 - verwenden und Verzögerung in Kauf nehmen
 - verwerfen und Ausfall in Kauf nehmen
- „Es gibt keinen festen Termin. Möglichst schnell halt.“
→ *keine Echtzeit*

4.1 Was ist Echtzeit?

Das Problem:

Harte Echtzeitanforderungen



Ressourcen optimal nutzen

Beispiel:

- Eine Motorsteuerung benötigt alle $2000\ \mu\text{s}$ einen Steuerimpuls, dessen Berechnung maximal $10\ \mu\text{s}$ dauert.
- Entweder: Der Steuer-Computer macht noch andere Dinge.
→ Risiko der Zeitüberschreitung
- Oder: Der Steuer-Computer macht nichts anderes.
→ Verschwendung von Rechenzeit
→ Na und?

4.1 Was ist Echtzeit?

Das Problem:

„Verschwendung von Rechenzeit – na und?“

Große Stückzahlen

- 138 000 Toyota Prius V von Mai 2011 bis April 2012
- Software viel im Einsatz: Server, Rechenzentrum

4.1 Was ist Echtzeit?

Das Problem:

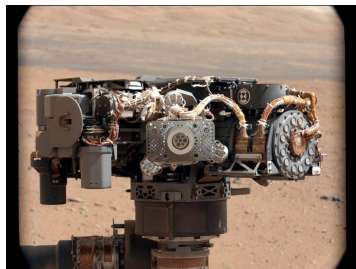
„Verschwendung von Rechenzeit – na und?“

Große Stückzahlen

- 138 000 Toyota Prius V von Mai 2011 bis April 2012
- Software viel im Einsatz: Server, Rechenzentrum

Wertvolle Ressourcen

- Fähigkeiten einer Raumsonde optimieren
(Marsumlaufbahn:
ab ca. 127 000 Euro pro kg)



4.1 Was ist Echtzeit?

Das Problem:

„Verschwendung von Rechenzeit – na und?“

Große Stückzahlen

- 138 000 Toyota Prius V von Mai 2011 bis April 2012
- Software viel im Einsatz: Server, Rechenzentrum

Wertvolle Ressourcen

- Fähigkeiten einer Raumsonde optimieren
(Marsumlaufbahn:
ab ca. 127 000 Euro pro kg)
- Implantat: Platz- und Stromverbrauch minimieren



4.1 Was ist Echtzeit?

Das Problem:

„Verschwendung von Rechenzeit – na und?“

Große Stückzahlen

- 138 000 Toyota Prius V von Mai 2011 bis April 2012
- Software viel im Einsatz: Server, Rechenzentrum

Wertvolle Ressourcen

- Fähigkeiten einer Raumsonde optimieren
(Marsumlaufbahn:
ab ca. 127 000 Euro pro kg)
- Implantat: Platz- und Stromverbrauch minimieren

→ **Echtzeitprogrammierung**

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

4.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Aber wie?

4.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Beispiele für Lösungen:

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

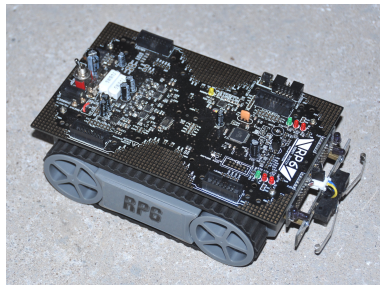


4.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Beispiele für Lösungen:

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



4.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Beispiele für Lösungen:

- 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



4.2 Echtzeitprogrammierung

Echtzeitanforderungen erfüllen, ohne Ressourcen zu verschwenden

Beispiele für Lösungen:

- 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



4.2 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.

4.2 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 ± 1 -Schritten

4.2 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 ± 1 -Schritten
- Nebenbei: Benutzerprogramm

4.2 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 ± 1 -Schritten
- Nebenbei: 1 Benutzerprogramm

4 Echtzeit

4.3 Multitasking

Qualitätssicherung beim Multitasking

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

Qualitätssicherung 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)

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 laufbereiten 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.

4 Echtzeit

4.4 Ressourcen

Ressourcen reservieren

- *Semaphor*
gemeinsame Variable mehrerer Prozesse
zur Regelung des Zugriffs auf eine Ressource
Ressource belegt → Kontextwechsel

griechisch: *sema* – Zeichen, *pherein* – tragen
„Eisenbahnsignal“

4 Echtzeit

4.4 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

4 Echtzeit

4.4 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
- *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

4 Echtzeit

4.4 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

4 Echtzeit

4.4 Ressourcen

Ressourcen reservieren: Beispiele

- *Semaphor*
linux-3.7-rc1/kernel/semaphor.c
linux-3.7-rc1/drivers/usb/core/file.c
- *Mutex*
linux-3.7-rc1/kernel/mutex.c
linux-3.7-rc1/drivers/usb/serial/usb-serial.c
- *Spinlock*
linux-3.7-rc1/kernel/spinlock.c
linux-3.7-rc1/kernel/semaphor.c, linux-3.7-rc1/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
    return serial;
    wieder freigeben
}
```


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 ninline 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_
```

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

4 Echtzeit

4.4 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

4 Echtzeit

4.4 Ressourcen

Verklemmungen: Gegenseitiges Blockieren von Ressourcen

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

4 Echtzeit

4.4 Ressourcen

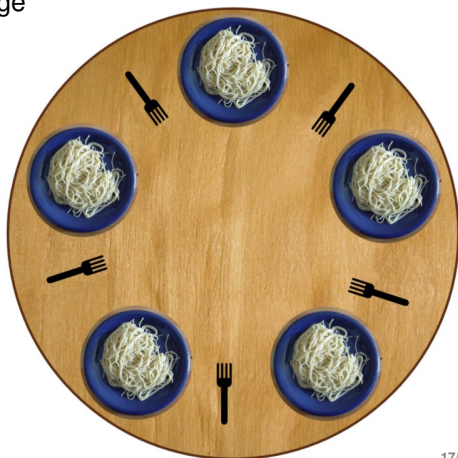
Verklemmungen: Gegenseitiges Blockieren von Ressourcen

- **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**



4 Echtzeit

4.4 Ressourcen

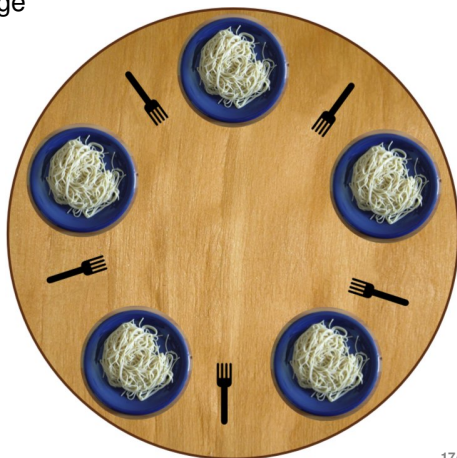
Verklemmungen: Gegenseitiges Blockieren von Ressourcen

- **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**



4 Echtzeit

4.4 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

4 Echtzeit

4.4 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 |
| • <i>hold and wait</i> | → simultane Zuteilung |
| • Entzug nicht möglich | → Prozesse suspendieren, beenden, <i>Rollback</i> |
| • zirkuläre Blockade | → Reihenfolge abhängig von Ressourcen |

4 Echtzeit

4.5 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.

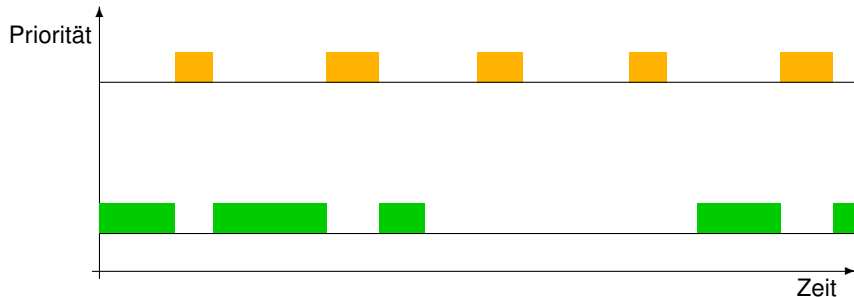
→ *statische Prioritätenvergabe*

4 Echtzeit

4.5 Prioritäten

Echtzeitbetriebssysteme

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

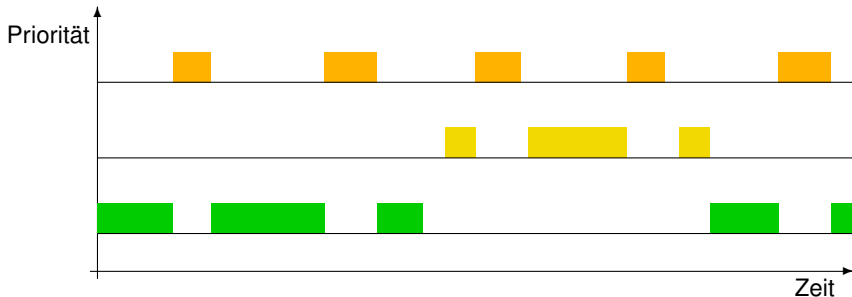


4 Echtzeit

4.5 Prioritäten

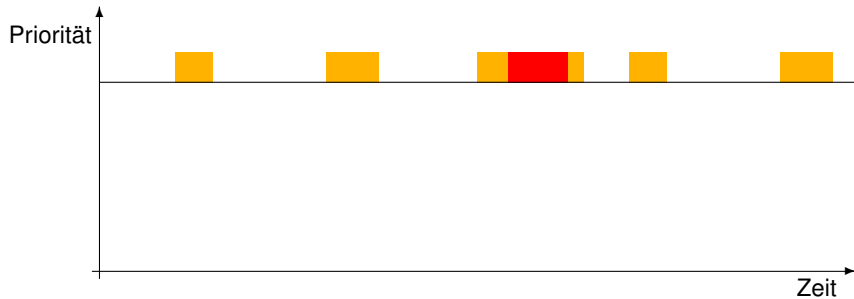
Echtzeitbetriebssysteme

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



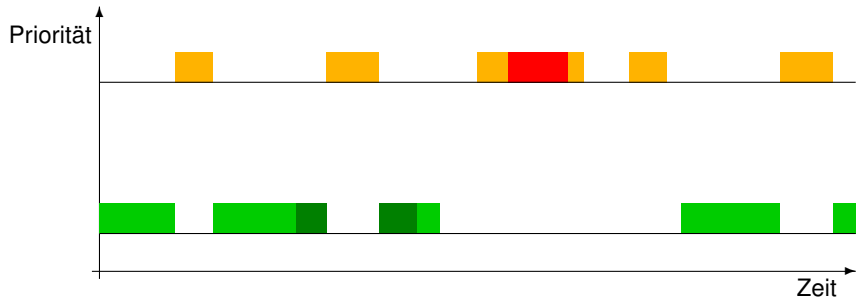
4 Echtzeit

4.5 Prioritäten und Ressourcen



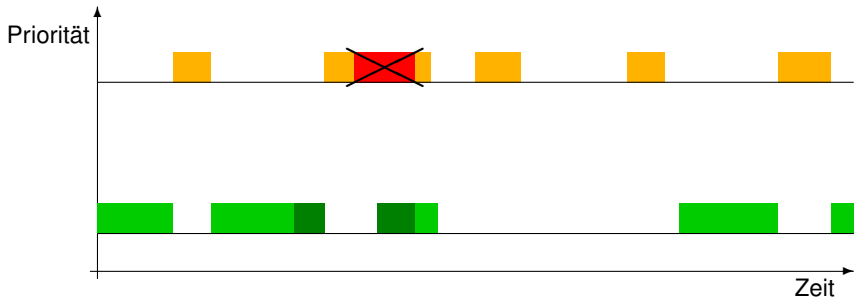
4 Echtzeit

4.5 Prioritäten und Ressourcen



4 Echtzeit

4.5 Prioritäten und Ressourcen



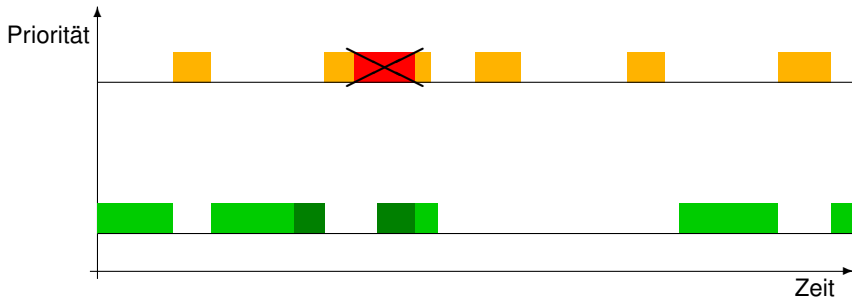
4 Echtzeit

4.5 Prioritäten und Ressourcen

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

→ *begrenzte Prioritätsinversion*

maximale Verzögerung: Länge des kritischen Bereichs



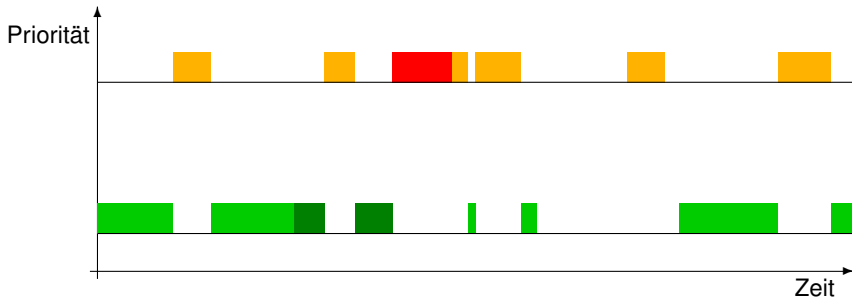
4 Echtzeit

4.5 Prioritäten und Ressourcen

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

→ *begrenzte Prioritätsinversion*

maximale Verzögerung: Länge des kritischen Bereichs



4 Echtzeit

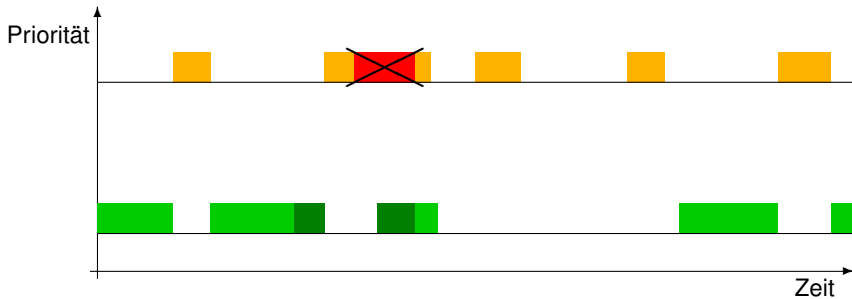
4.5 Prioritäten und Ressourcen

unbegrenzte Prioritätsinversion

4 Echtzeit

4.5 Prioritäten und Ressourcen

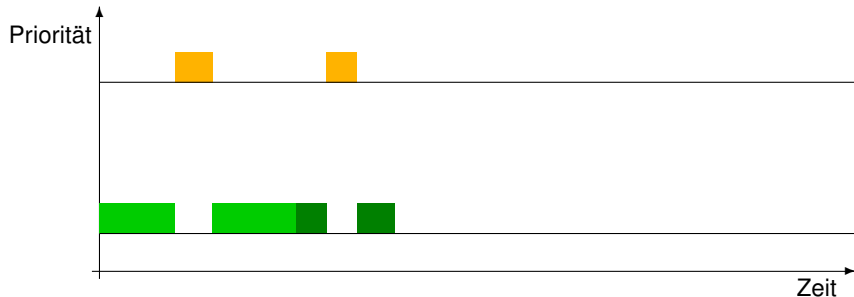
unbegrenzte Prioritätsinversion



4 Echtzeit

4.5 Prioritäten und Ressourcen

unbegrenzte Prioritätsinversion

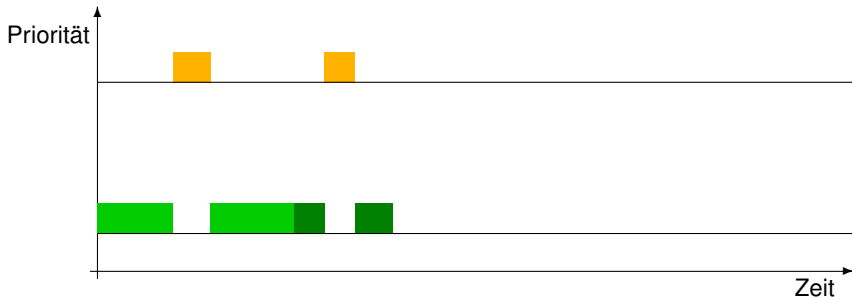


4 Echtzeit

4.5 Prioritäten und Ressourcen

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

→ *unbegrenzte Prioritätsinversion*

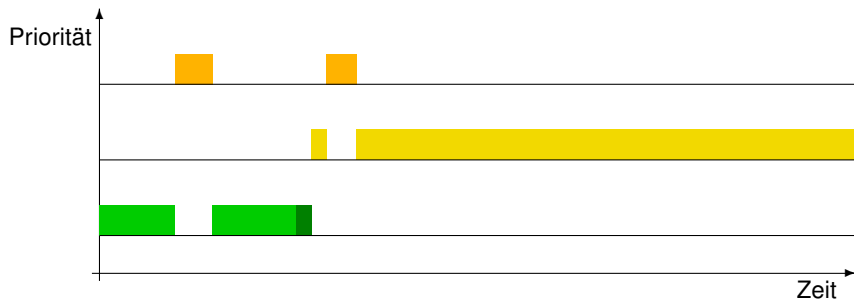


4 Echtzeit

4.5 Prioritäten und Ressourcen

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

→ *unbegrenzte Prioritätsinversion*



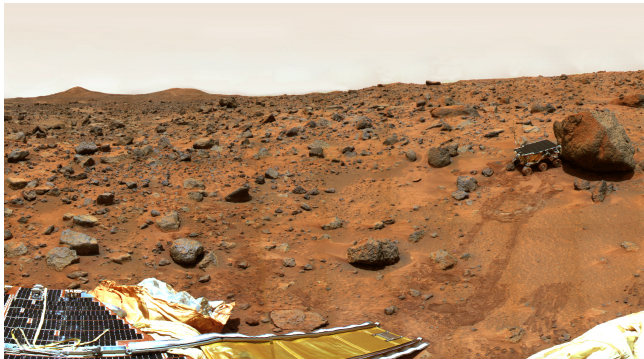
4 Echtzeit

4.5 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



4 Echtzeit

4.5 Prioritäten und Ressourcen

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

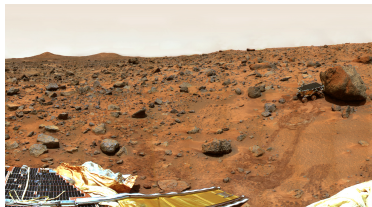
—→ *unbegrenzte Prioritätsinversion*

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/



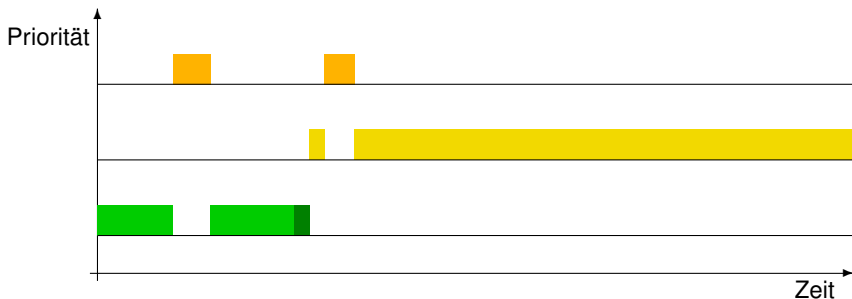
4 Echtzeit

4.5 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*



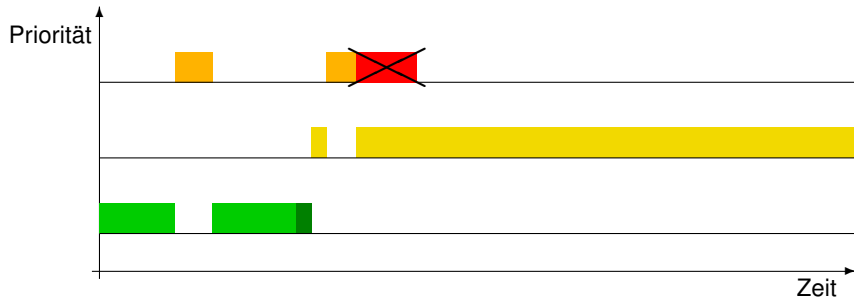
4 Echtzeit

4.5 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*



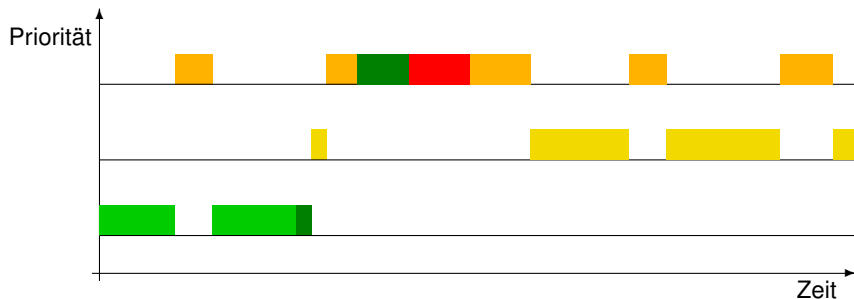
4 Echtzeit

4.5 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*



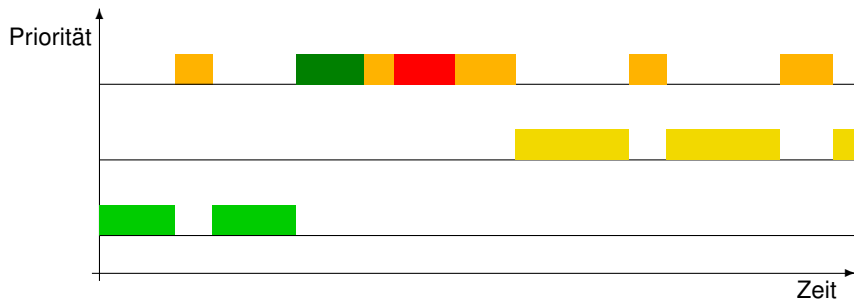
4 Echtzeit

4.5 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*



4 Echtzeit

4.5 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.

4 Echtzeit

4.5 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

4 Echtzeit

4.5 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