Hochschule Bochum
Bochum University
of Applied Sciences

Vertiefung Systemtechnik

Prof. Dr. Peter Gerwinski

22. Oktober 2012

2 Theorie der Echtzeitsysteme

2.2 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
  (z. B. *binärer* Semaphor mit *Besitzer*)

- *Spinlock* (*busy waiting*)
  leichtgewichtige Alternative zu Kontextwechsel
  Anwendung: Zugriff auf den Semaphor selbst („Semaphor-Semaphor“)

  Hardware-Unterstützung erforderlich:
  atomar prüfen, ob eine Variable einen bestimmten Wert hat;
  wenn ja, auf anderen Wert setzen; andere Prozessoren solange anhalten

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 noinline 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);
```

← exklusiven Zugriff auf Mutex sichern

```
  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);
```

exklusiven Zugriff auf Mutex wieder freigeben

```
  debug_mutex_free_waiter (&waiter);
  preempt_enable ();

  return 0;
}
```

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

Literaturempfehlung

Prof. Dr. Joachim Wietzke, FH Darmstadt,
Prof. Dr. Manh Tien Tran, FH Kaiserslautern:

Automotive Embedded Systeme

Springer-Verlag, Berlin, Heidelberg 2005

Lizenz: proprietär

(gesetzt mit LaTeX, ca. 10 €)

2 Theorie der Echtzeitsysteme

2.2 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

- *Kritischer Abschnitt – critical section*
  Programmabschnitt zwischen Reservierung
  und Freigabe einer Ressource
  $\longrightarrow$ sollte immer so kurz wie möglich sein

2 Theorie der Echtzeitsysteme

2.2 Ressourcen

Verklemmungen: Gegenseitiges Blockieren von Ressourcen

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

2 Theorie der Echtzeitsysteme

2.2 Ressourcen

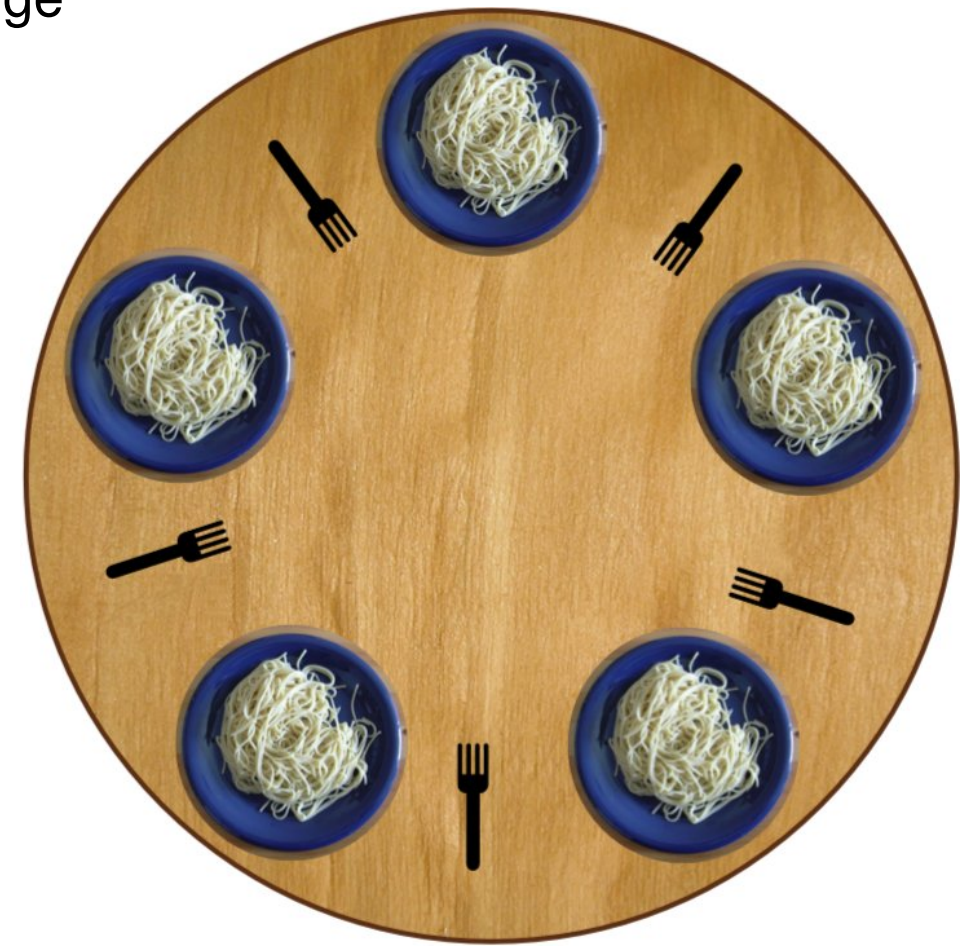
Verklemmungen: Gegenseitiges Blockieren von Ressourcen

- *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 $\longrightarrow$ *Verklemmung*

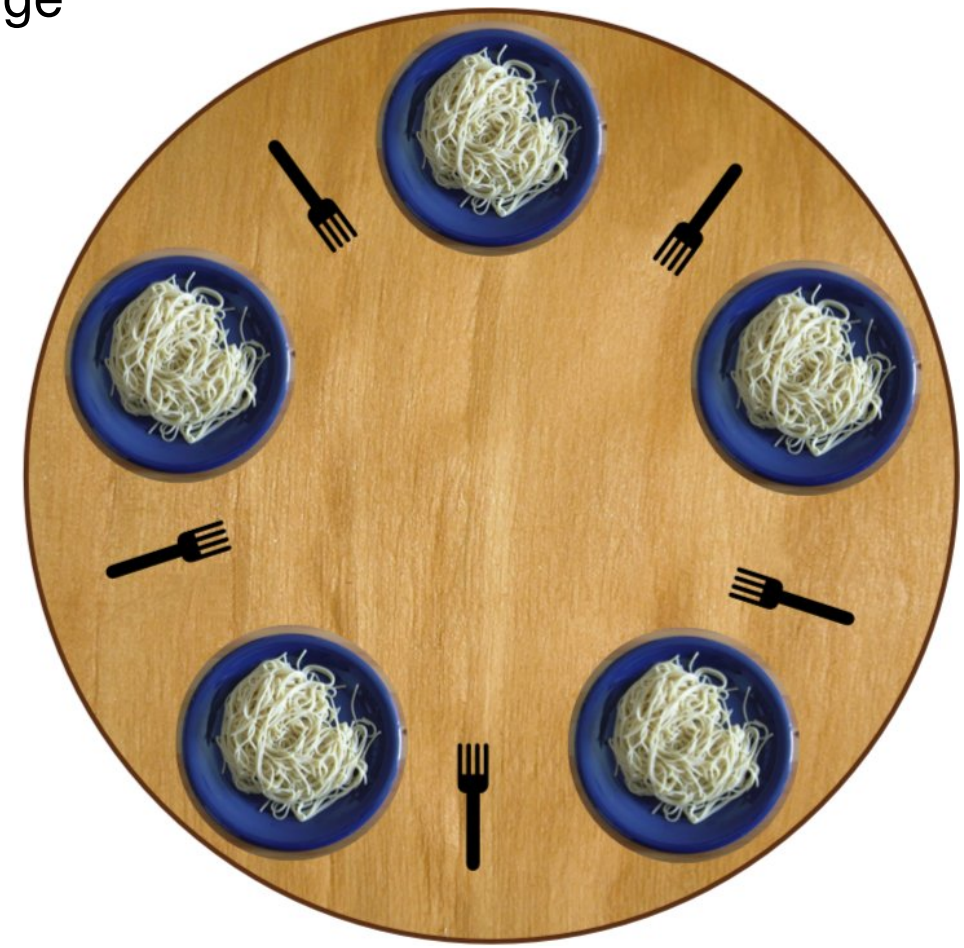


2 Theorie der Echtzeitsysteme

2.2 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 $\longrightarrow$ *Verklemmung*
schweigen $\longrightarrow$ *Deadlock*
philosophieren weiter $\longrightarrow$ *Livelock*

2 Theorie der Echtzeitsysteme

2.2 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

2 Theorie der Echtzeitsysteme

2.2 Ressourcen

Verklemmungen: Gegenseitiges Blockieren von Ressourcen

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

Beispiel: Philosophenproblem

Bedingungen für Verklemmungen:

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

2 Theorie der Echtzeitsysteme

2.3 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.

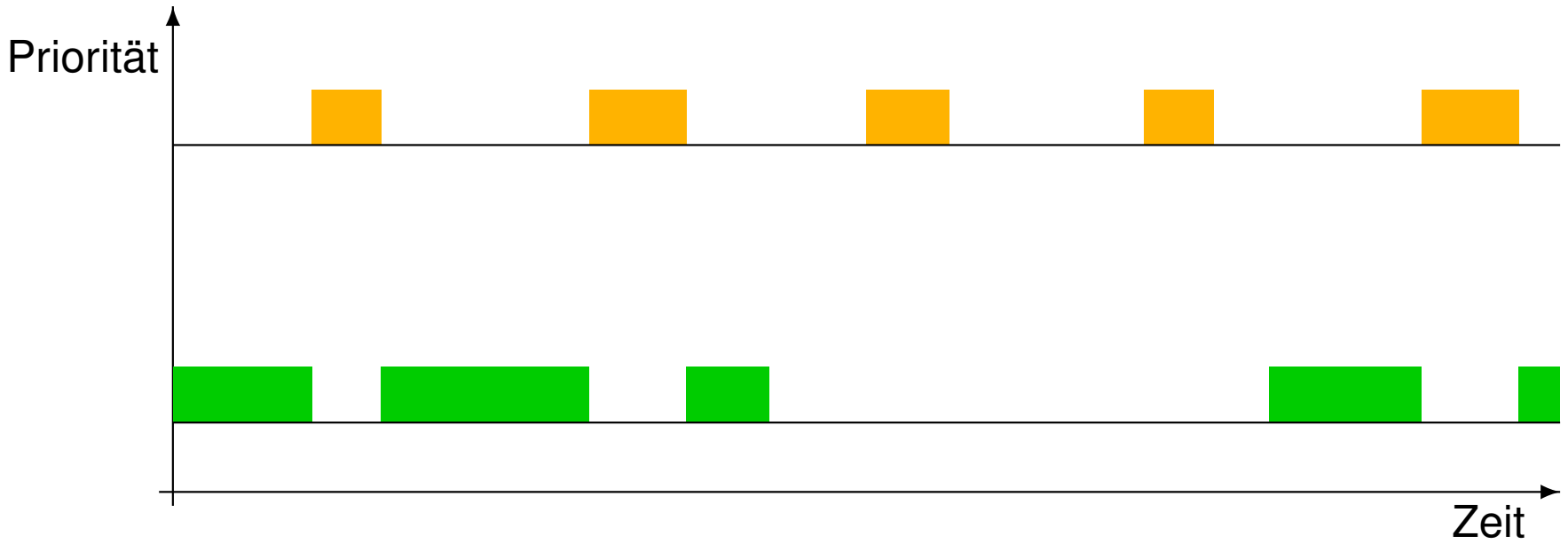
⟶ *statische Prioritätenvergabe*

2 Theorie der Echtzeitsysteme

2.3 Prioritäten

Echtzeitbetriebssysteme

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

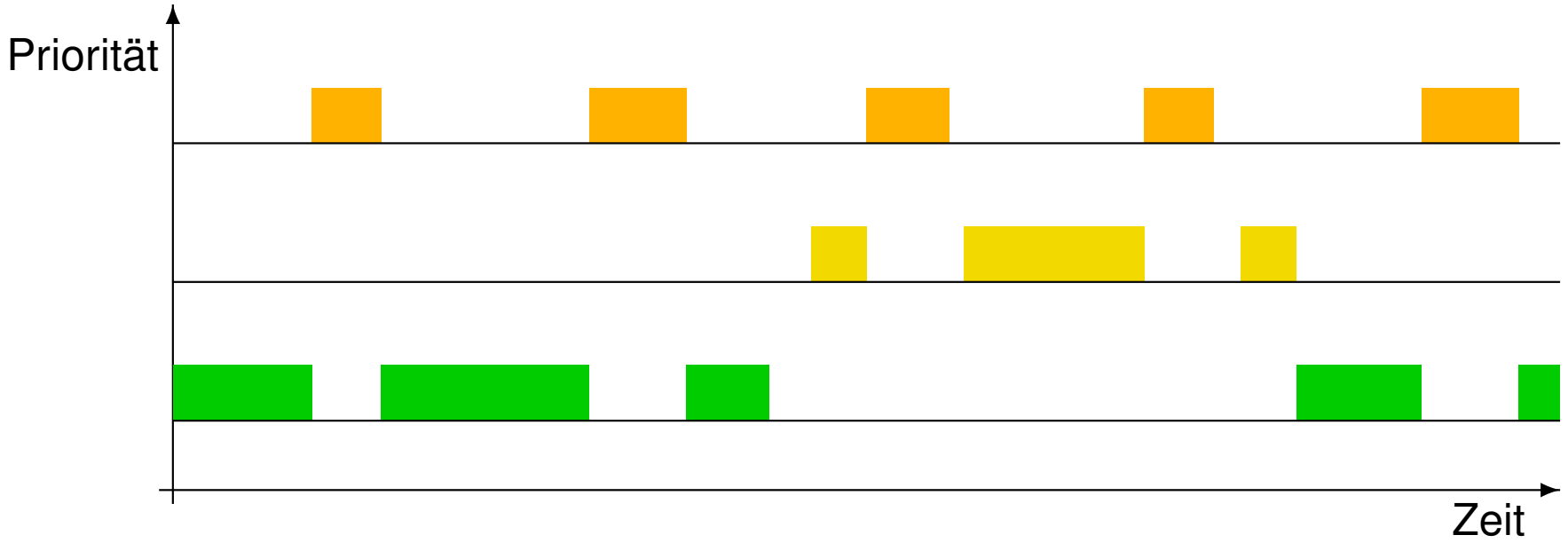


2 Theorie der Echtzeitsysteme

2.3 Prioritäten

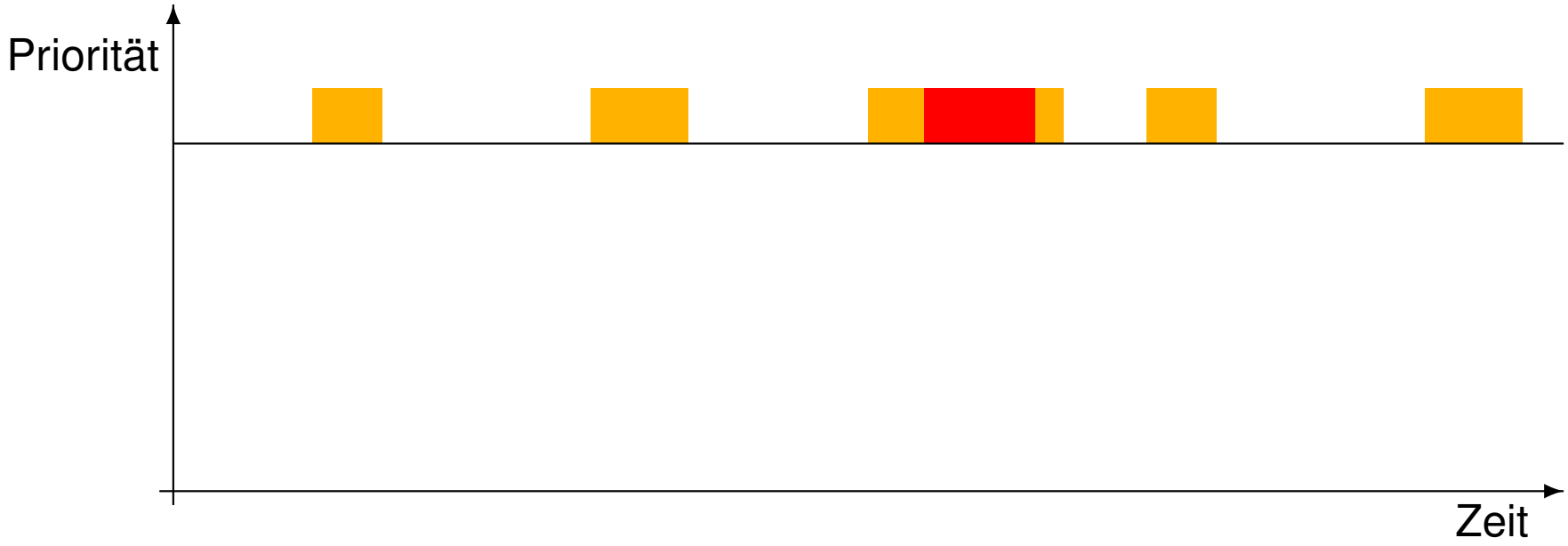
Echtzeitbetriebssysteme

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



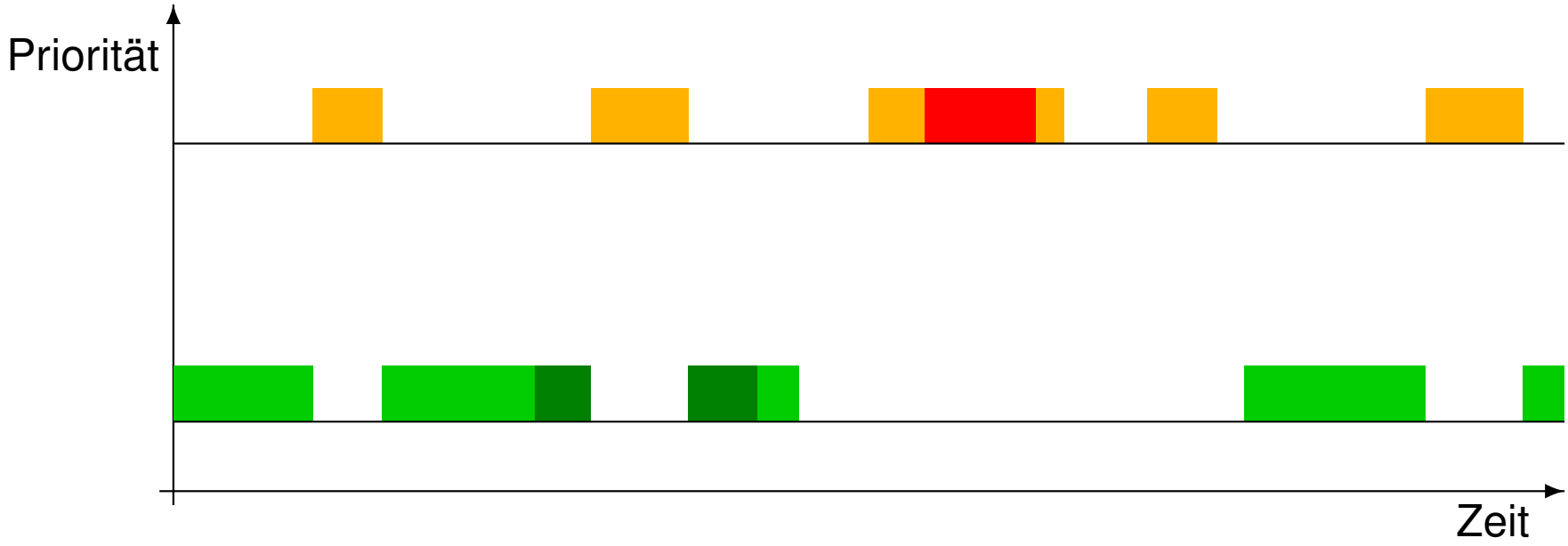
2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen



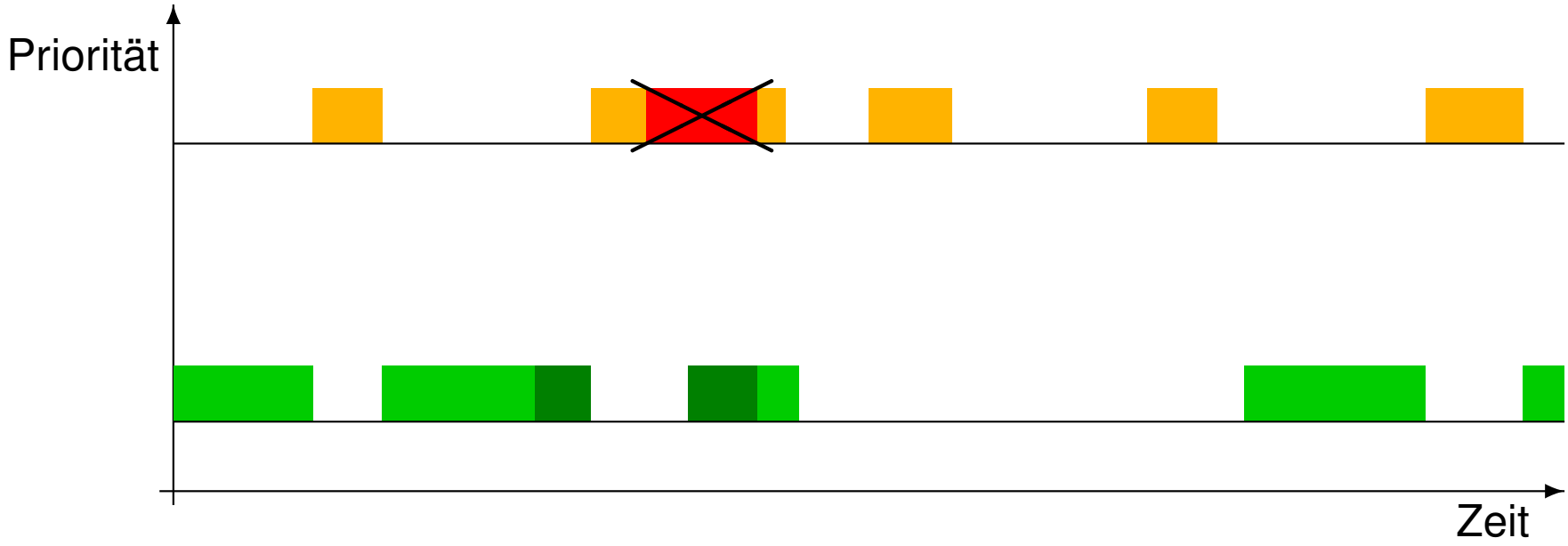
2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen



2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen



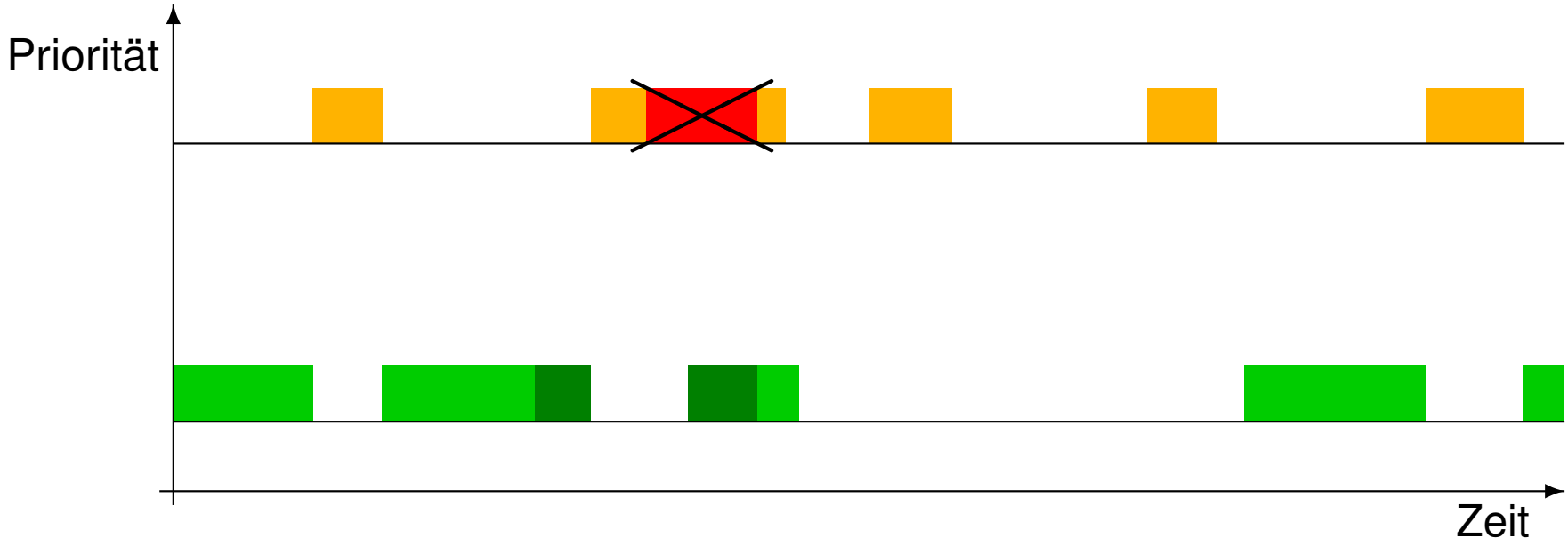
2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *begrenzte Prioritätsinversion*

maximale Verzögerung: Länge des kritischen Bereichs



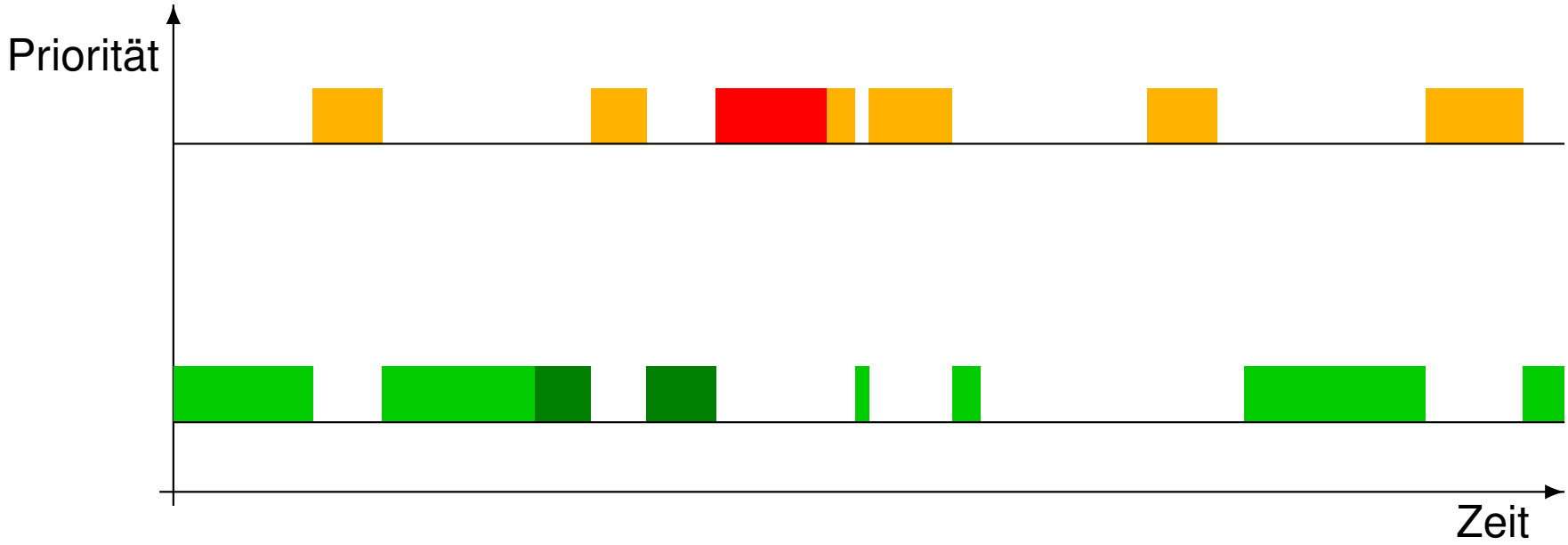
2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *begrenzte Prioritätsinversion*

maximale Verzögerung: Länge des kritischen Bereichs



2 Theorie der Echtzeitsysteme

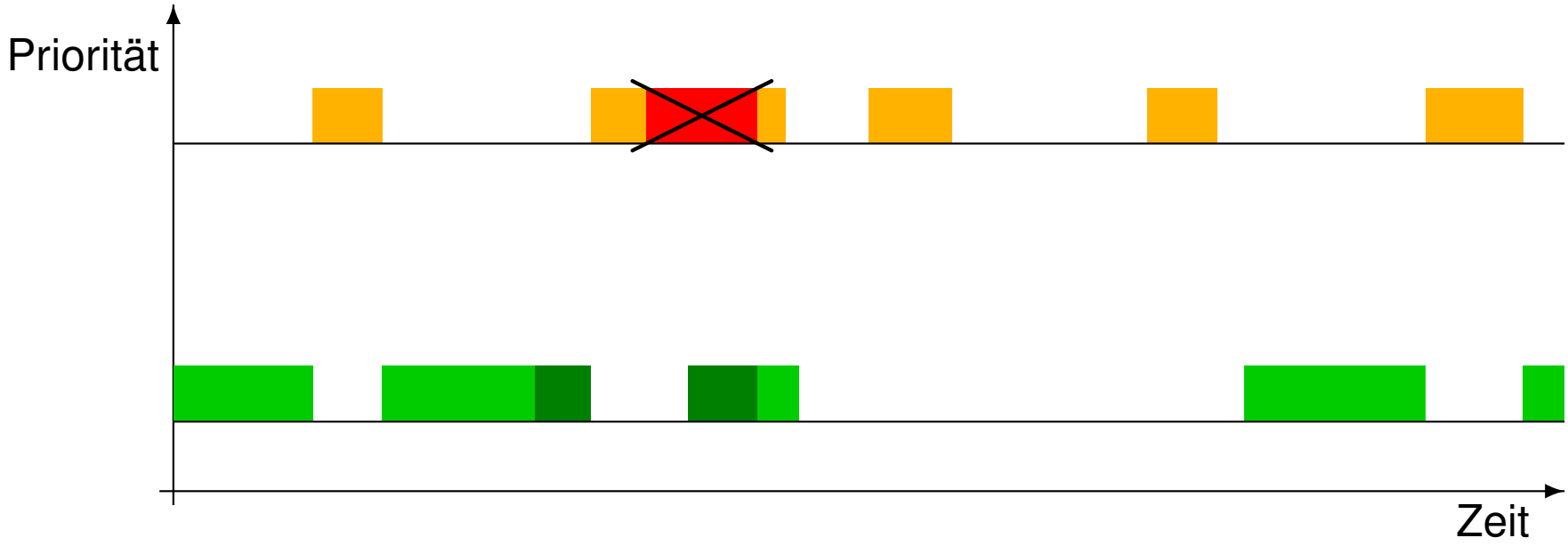
2.3 Prioritäten und Ressourcen

unbegrenzte Prioritätsinversion

2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

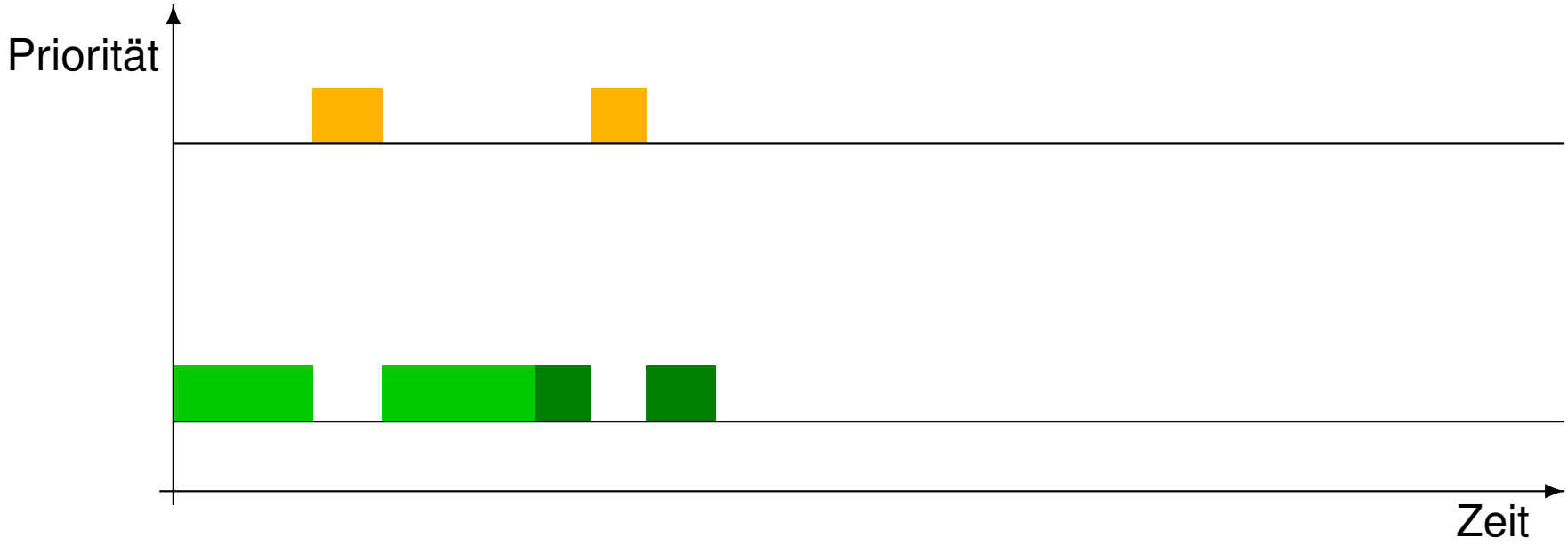
unbegrenzte Prioritätsinversion



2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

unbegrenzte Prioritätsinversion

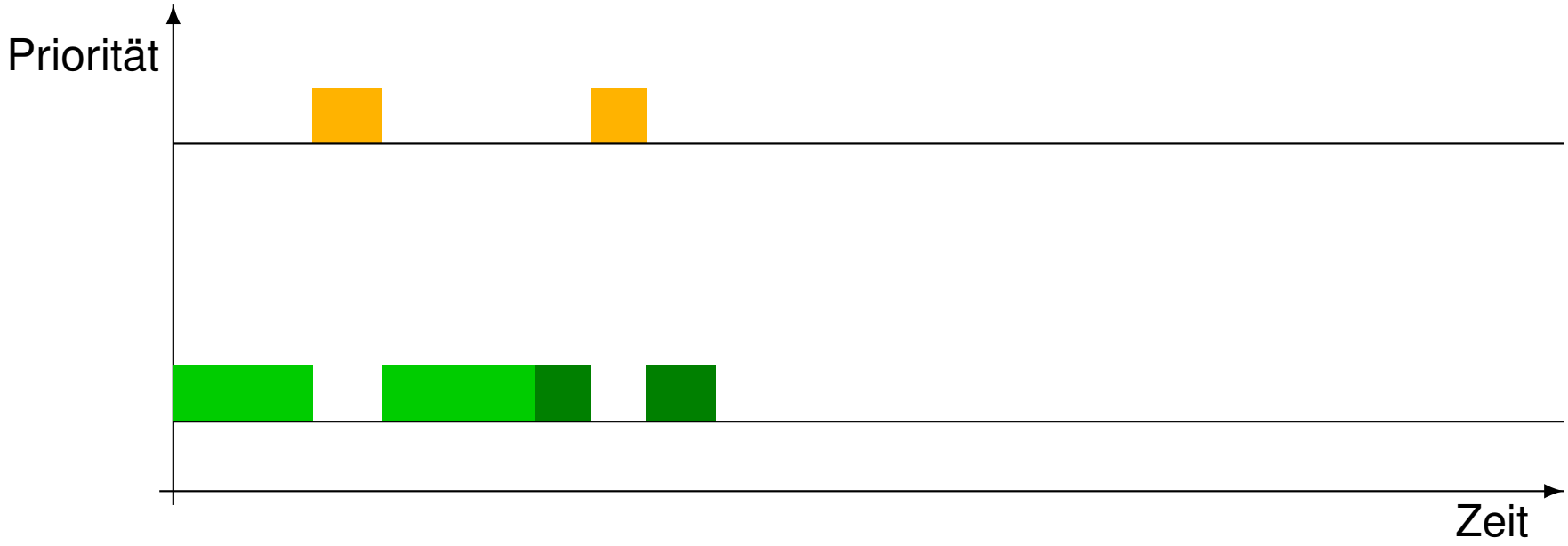


2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *unbegrenzte Prioritätsinversion*

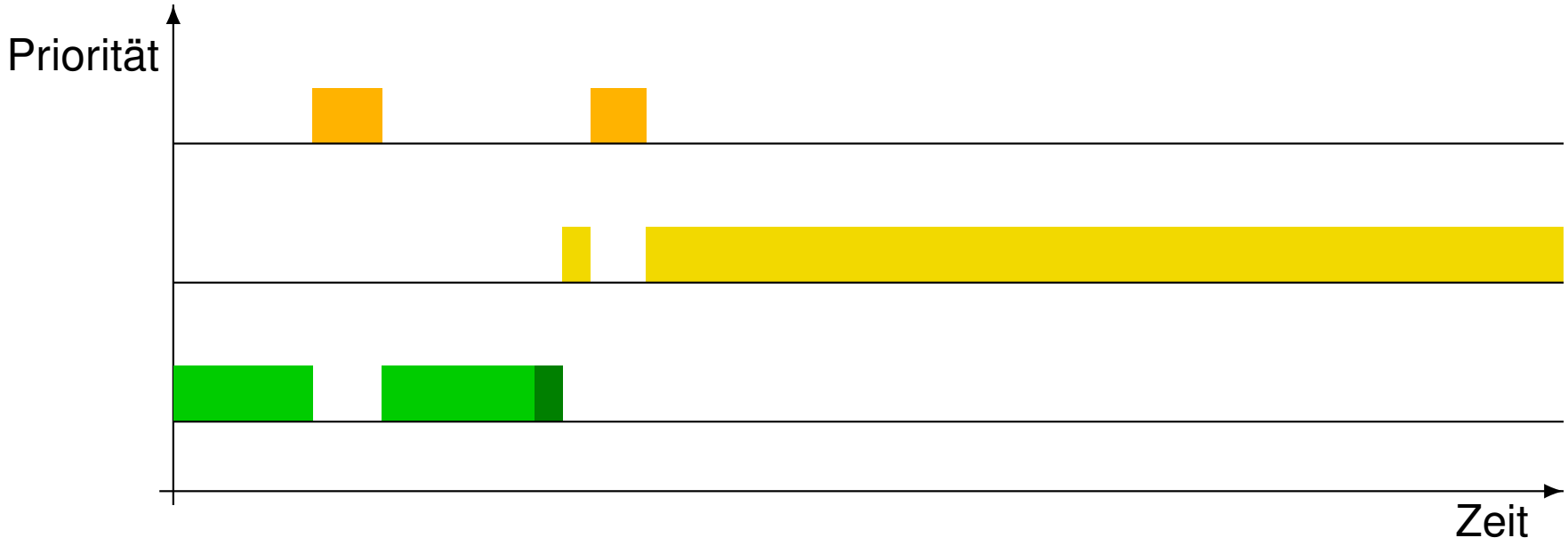


2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *unbegrenzte Prioritätsinversion*



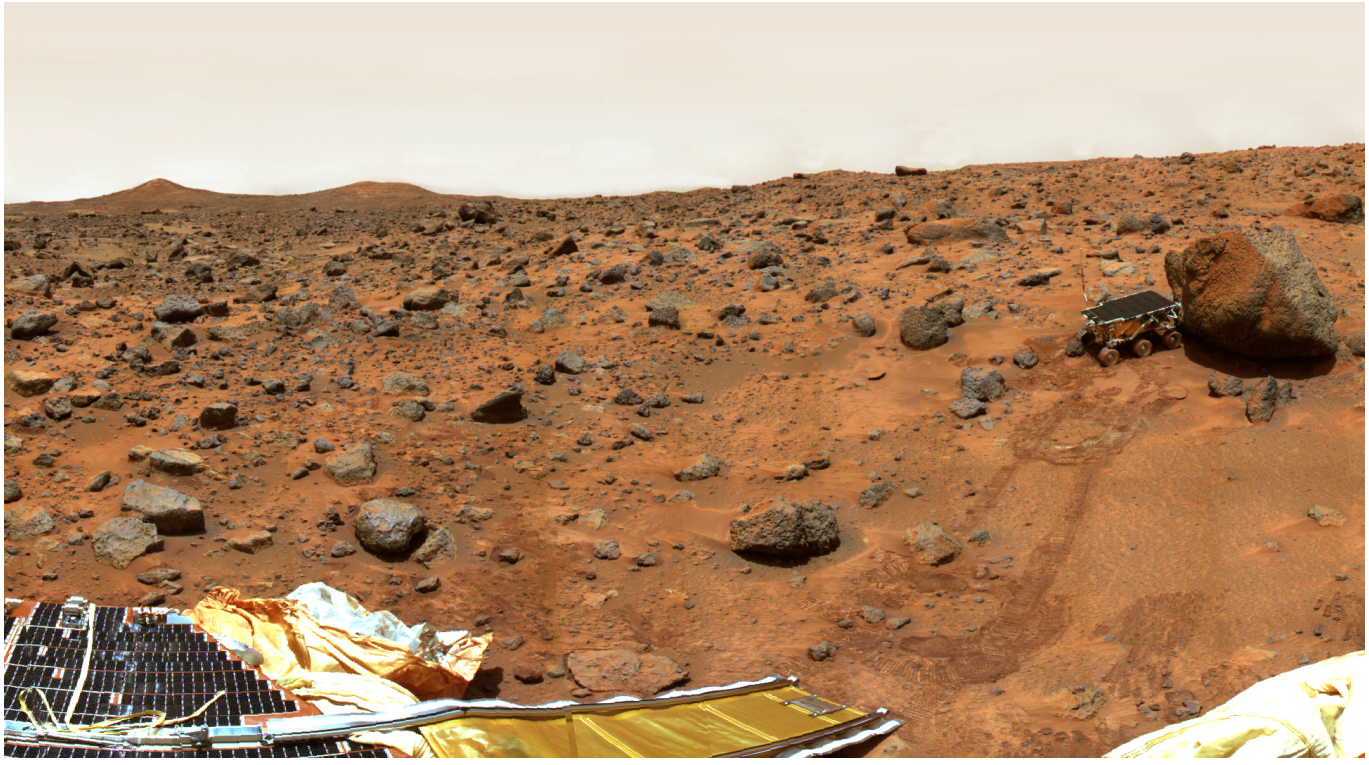
2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *unbegrenzte Prioritätsinversion*

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



2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *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.

2 Theorie der Echtzeitsysteme

2.3 Prioritäten und Ressourcen

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

$\longrightarrow$ *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

- *Priority Aging*
  Die Priorität wächst mit der Wartezeit.