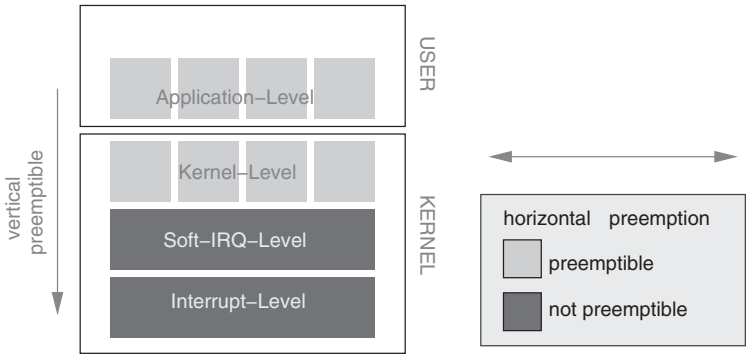


Abbildung 4-5
Unterbrechungsmodell

Da Funktionen, die im Interrupt-Kontext ablaufen, nicht schlafen gelegt werden können, kann ein kritischer Abschnitt auf dieser Ebene niemals per Semaphor geschützt werden. Hier wird auf dem lokalen Kern eine Interruptsperre eingesetzt und für die reale Parallelität ein Spinlock verwendet.

Aktuelle Linux-Kernel bieten die Möglichkeit von sogenannten Threaded Interrupts. Hierbei laufen Interrupts als Kernel-Threads, also im Kernel-Kontext ab. Vorteil: Threads können durch den Systemarchitekten untereinander und sogar bezüglich sonstiger User-Tasks priorisiert werden (siehe Abschnitt 5.3).

4.4 Umgang mit Zeiten

Betriebssysteme stellen Realzeitapplikationen Zeitgeber mit unterschiedlichen Eigenschaften zur Verfügung, über die das Zeitverhalten kontrolliert wird. Diese sind gekennzeichnet durch ihre

- Genauigkeit,
- die Zuverlässigkeit,
- den Bezugspunkt,
- die Darstellung und
- den maximalen messbaren Zeitbereich.

Das Betriebssystem repräsentiert Zeiten unter anderem mit den folgenden Datentypen (Darstellung):

- `time_t`: Zeit in Sekundenauflösung.
- `clock_t`: Timerticks.
- `struct timeval`: Zeit in Mikrosekunden-Auflösung.
- `struct timespec`: Zeit in Nanosekunden-Auflösung.
- `struct tms`: Accounting, Rechen- und Reaktionszeiten.
- `struct tm`: absolute Zeitangabe.

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};

struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};

struct tms {
    clock_t tms_utime;  /* user time */
    clock_t tms_stime;  /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};

struct tm {
    int tm_sec;         /* seconds */
    int tm_min;         /* minutes */
    int tm_hour;        /* hours */
    int tm_mday;        /* day of the month */
    int tm_mon;         /* month */
    int tm_year;        /* year */
    int tm_wday;        /* day of the week */
    int tm_yday;        /* day in the year */
    int tm_isdst;       /* daylight saving time */
};
```

Der Datentyp `time_t` repräsentiert typischerweise einen vorzeichenbehafteten `long`-Datentyp. Der negative Zeitbereich wird genutzt, um Zeiten vor dem Bezugspunkt (z. B. vor dem 1.1.1970) auszudrücken.

Die Strukturen `struct timeval` und `struct timespec` bestehen aus jeweils zwei Variablen, die einmal den Sekundenanteil und einmal den Mikro- beziehungsweise den Nanosekundenanteil repräsentieren. Die Darstellung erfolgt jeweils normiert. Das bedeutet, dass der Mikro- oder Nanosekundenanteil immer kleiner als eine volle Sekunde bleibt. Ein Zeitstempel von beispielsweise *1 Sekunde, 123456 Mikrosekunden* ist gültig, *1 Sekunde, 1234567 Mikrosekunden* ist ungültig. In normierter Darstellung ergäben sich *2 Sekunden, 234567 Mikrosekunden*.

Die Darstellungs- beziehungsweise Repräsentationsform reflektiert auch den darstellbaren Wertebereich. Da bei den dargestellten Datenstrukturen für den Typ `time_t` ein `long` eingesetzt wird, lassen sich auf einer 32-Bit-Maschine rund 4 Milliarden Sekunden zählen, auf einer 64-Bit-Maschine 2^{64} (mehr als 500 Milliarden Jahre).

Folgende Bezugspunkte für Zeiten haben sich eingebürgert:

- Start des Systems,

- Start eines Jobs und
- Start einer Epoche, beispielsweise »Christi Geburt« oder der 1.1.1970 (Unix-Epoche). Dieser Bezugspunkt weist zudem noch eine örtliche Komponente auf: Der Zeitpunkt 19:00 Uhr in Europa entspricht einem anderen Zeitpunkt in den USA (minus sechs Stunden zur Ostküste).

Die Genauigkeit wird beeinflusst durch die Taktung des Zeitgebers, deren Schwankungen und durch Zeitsprünge.

Das Attribut *Zuverlässigkeit* eines Zeitgebers beschreibt dessen Verhalten bei (bewussten) Schwankungen der Taktung und bei Zeitsprüngen: Ein Zeitgeber kann beispielsweise der Systemuhr folgen (CLOCK_REALTIME) oder unabhängig von jeglicher Modifikation an der Systemzeit einfach weiter zählen (CLOCK_MONOTONIC). Die POSIX-Realzeiterweiterung beziehungsweise Linux-Systeme definieren hierzu folgende Clocks [Manpage: clock_gettime]:

CLOCK_REALTIME
: Dieser Zeitgeber repräsentiert die systemweite, aktuelle Zeit. Er reagiert auf Zeitsprünge (sowohl vorwärts als auch rückwärts), die beispielsweise beim Aufwachen (Resume) nach einem Suspend (Schlafzustand des gesamten Systems) ausgelöst werden. Er reagiert ebenfalls auf unterschiedliche Taktungen, die beispielsweise durch NTP erfolgen. Dieser Zeitgeber liefert die Sekunden und Nanosekunden seit dem 1.1. 1970 UTC (Unix-Zeit) zurück.

CLOCK_MONOTONIC
: Dieser Zeitgeber läuft entsprechend seiner Auflösung stets vorwärts, ohne dabei Zeitsprünge zu vollziehen. Er ist also unabhängig von der mit Superuser-Privilegien zu verändernden Systemuhr. Allerdings reagiert dieser Zeitgeber auf Modifikationen der Taktung, die beispielsweise durch NTP erfolgen.

CLOCK_MONOTONIC_RAW
: Dieser Zeitgeber ist Linux-spezifisch. Er reagiert weder auf Zeitsprünge noch auf im Betrieb geänderte Taktungen (NTP).

CLOCK_PROCESS_CPUTIME_ID
: Dieser Zeitgeber erfasst die Verarbeitungszeit (Execution Time) des zugehörigen Prozesses. Das funktioniert aber nur zuverlässig auf Singlecore-Systemen beziehungsweise wenn sichergestellt werden kann, dass keine Prozessmigration stattfindet.

CLOCK_THREAD_CPUTIME_ID
: Dieser Zeitgeber erfasst die Verarbeitungszeit (Execution Time) des zugehörigen Threads. Das funktioniert aber nur zuverlässig

auf Singlecore-Systemen beziehungsweise wenn sichergestellt werden kann, dass keine Prozessmigration stattfindet.

Der maximal messbare Zeitbereich schließlich ergibt sich durch die Auflösung des Zeitgebers und die Bitbreite der Variablen:

```
zeitbereich = auflösung * 2
```

Im Folgenden werden zunächst Funktionen vorgestellt, mit denen die aktuelle Zeit bestimmt werden kann. Danach zeigen wir, wie zwei Zeitpunkte miteinander verglichen werden können. Das Rechnen mit Zeiten wird in Abschnitt 4.4.3 diskutiert. Schließlich folgt ein Abschnitt über das gewollte Schlafenlegen beziehungsweise die Implementierung von Timerfunktionen.

4.4.1 Aktuelle Zeit bestimmen

Es gibt unterschiedliche Systemfunktionen, mit denen die aktuelle Zeit gelesen werden kann. Favorisiert ist die Funktion `int clock_gettime(clockid_t clk_id, struct timespec *tp)`, die die Zeit seit dem 1.1.1970 (Unix-Zeit) als *Universal Time* (Zeitzone UTC) zurückliefert (`struct timespec`). Konnte die aktuelle Zeit gelesen werden, gibt die Funktion null, ansonsten einen Fehlercode zurück. Allerdings ist das Auslesen auf 32-Bit-Systemen problematisch, da der 32-Bit-Zähler am 19. Januar 2038 überläuft. Vor allem eingebettete Systeme, bei denen von einer jahrzehntelangen Standzeit ausgegangen wird, könnten von diesem Zählerüberlauf betroffen sein. Wichtig ist, dass sie für diesen Fall korrekt programmiert sind (siehe Abschnitt 4.4.2).

Tabelle 4-2
Funktionen zum Lesen von Zeiten

Funktion	Beschreibung
`time`	Gibt die Anzahl Sekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind.
`gettimeofday`	Gibt die Anzahl Sekunden und Mikrosekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind.
`clock_gettime`	Sekunden und Nanosekunden, die seit dem 1.1.1970 (UTC) vergangen sind. Die Genauigkeit kann per `clock_getres()` ausgelesen werden.
`times`	Liefert die aktuelle Zeit als Timerticks (Bezugszeitpunkt ist nicht genau definiert); zusätzlich auch die Verarbeitungszeit, die im Kernel und die im Userland angefallen ist.
TSC	Der TSC ist ein mit der Taktfrequenz der CPU getakteter, sehr genauer Zähler, auf den per Maschinenbefehle zugegriffen werden kann. Vorsicht: Die Taktfrequenz der CPU kann schwanken.

```
struct timepspec timestamp;
...
if (clock_gettime(CLOCK_MONOTONIC,&timestamp)) {
```

```
    perror("timestamp");
    return -1;
}
printf("seconds: %ld, nanoseconds: %ld\n",
    timestamp.tv_sec, timestamp.tv_nsec);
```

Durch die Wahl der Clock `CLOCK_PROCESS_CPUTIME_ID` beziehungsweise `CLOCK_THREAD_CPUTIME_ID` kann auch die Verarbeitungszeit ausgemessen werden (Profiling).

Die Genauigkeit der zurückgelieferten Zeit kann mithilfe der Funktion `clock_getres(clockid_t clk_id, struct timespec *res)` ausgelesen werden.

Beispiel 4-9
Auslesen der Zeitgebergenauigkeit

```
#include <stdio.h>
#include <time.h>

int main( int argc, char **argv, char **envp )
{
   struct timespec ts;

   clock_getres( CLOCK_MONOTONIC, &ts);
   printf("resolution CLOCK_MONOTONIC: %ld sec, %ld nanoseconds\n",
      ts.tv_sec, ts.tv_nsec );
   clock_getres( CLOCK_REALTIME, &ts);
   printf("resolution CLOCK_REALTIME: %ld sec, %ld nanoseconds\n",
      ts.tv_sec, ts.tv_nsec );
   return 0;
}
```

Die Funktion `clock_gettime()` ist nicht in der Standard-C-Bibliothek zu finden, sondern in der Realzeit-Bibliothek `librt`. Daher ist bei der Programmgenerierung diese Bibliothek hinzuzulinken (Parameter `-lrt`). Steht nur die Standard-C-Bibliothek zur Verfügung, kann `time_t time(time_t *t)` oder auch `int gettimeofday(struct timeval *tv, struct timezone *tz)` eingesetzt werden.

`time()` gibt die Sekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind.

```
time_t now;
...
now = time(NULL);
```

`gettimeofday()` schreibt an die per `tv` übergebene Speicheradresse die Sekunden und Mikrosekunden seit dem 1.1.1970. Das Argument `tz` wird typischerweise mit `NULL` angegeben.

Liegen die Sekunden seit dem 1.1.1970 vor (`timestamp.tv_sec`), können diese mithilfe der Funktionen `struct tm *localtime_r(const time_t`

*timep, struct tm *result); oder struct tm *gmtime_r(const time_t *timep, struct tm *result); in die Struktur struct tm konvertiert werden.

```
struct tm absolute_time;

if (localtime_r( timestamp.tv_sec, &absolute_time )==NULL) {
   perror( "localtime_r" );
   return -1;
}
printf("year: %d\n", absolute_time.tm_year);
```

Die Funktion time_t mktime(struct tm *tm) konvertiert eine über die Struktur struct tm gegebene Zeit in Sekunden seit dem 1.1.1970 (time_t).

Mithilfe der Funktion clock_t times(struct tms *buf) lässt sich sowohl die aktuelle Zeit zu einem letztlich nicht genau definierten Bezugspunkt als auch die Verarbeitungszeit (Execution Time) des aufrufenden Prozesses bestimmen. Die Zeiten werden als Timerticks (clock_t) zurückgeliefert. Die zurückgelieferte Verarbeitungszeit ist aufgeschlüsselt in die Anteile, die im Userland, und die Anteile, die im Kernel verbraucht wurden. Außerdem werden diese Anteile auch für Kindprozesse gelistet.

Mithilfe des Systemcalls sysconf(_SC_CLK_TCK) kann die Anzahl der Timerticks pro Sekunde ausgelesen werden. Der Kehrwert gibt dann die Zeitdauer eines Timerticks an (siehe Beispiel 4-10).

Beispiel 4-10 *Lesen von Verarbeitungszeiten*

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>
#include <time.h>

int main( int argc, char **argv, char **envp )
{
   struct tms exec_time;
   clock_t act_time;
   long ticks_per_second;
   long tickduration_in_ms;

   ticks_per_second = sysconf(_SC_CLK_TCK);
   tickduration_in_ms = 1000/ticks_per_second;

   act_time = times( &exec_time );
   printf("actual time (in ms): %ld\n", act_time*tickduration_in_ms);
   printf("execution time (in ms): %ld\n",
      (exec_time.tms_utime+exec_time.tms_stime)*tickduration_in_ms);
```

```
   return 0;
}
```

Sehr genaue Zeiten lassen sich erfassen, falls der eingesetzte Prozessor einen Zähler besitzt, der mit der Taktfrequenz des Systems getaktet wird. Bei einer x86-Architektur (PC) heißt dieser Zähler *Time-Stamp-Counter* (TSC). Der TSC kann auch von einer normalen Applikation ausgelesen werden, allerdings muss sichergestellt sein, dass sich die Taktfrequenz zwischen zwei Messungen nicht ändert. Alternativ kann man sich vom Betriebssystem über die Taktänderung informieren lassen.

4.4.2 Der Zeitvergleich

Sollen zwei Zeitstempel verglichen werden, die über eine `struct timeval` repräsentiert sind, kann das Makro `int timercmp(struct timeval *a, struct timeval *b, CMP)` eingesetzt werden. Für `CMP` ist der Vergleichsoperator »<«, »>«, »==«, »<=« oder »>=« einzusetzen. Die Linux-Manpage weist darauf hin, dass auf anderen Plattformen die Vergleiche »<=«, »==« und »>=« nicht immer korrekt implementiert sind, und empfiehlt daher, für eine portierbare Realzeitapplikation die negierten Varianten zu verwenden:

```
if (!timercmp( a, b, < )) {
    // a ist identisch b oder später
    ...
}
if (!timercmp( a, b, > )) {
    // a ist früher als b oder gleich
    ...
}
if (!timercmp( a, b, != )) {
    // a ist identisch mit b
    ...
}
```

Liegen die Zeitstempel in Form einer `struct timespec`, also mit Nanosekundenanteil vor, gibt es keine vorgefertigten Makros. Hier könnte alternativ die in Beispiel 4-11 dargestellte Funktion verwendet werden, die »-1« zurückgibt, falls der erste Zeitstempel (`first`) zeitlich vor dem zweiten Zeitstempel (`second`) liegt, »0«, falls beide Zeitstempel identisch sind, und »1«, falls der erste Zeitstempel zeitlich nach dem ersten Zeitstempel liegt.

Beispiel 4-11
Vergleich zweier Zeitstempel vom Typ struct timespec

```
/*********************************************************/
/* return  1: falls "first" zeitlich nach "second" liegt */
/* return  0: falls "first" und "second" identisch sind  */
/* return -1: falls "first" zeitlich vor "second" liegt  */
/*********************************************************/
int timespec_cmp( struct timespec *first, struct timespec *second )
{
    if (first->tv_sec > second->tv_sec)
        return 1; // first later than second (first > second)
    if (first->tv_sec < second->tv_sec)
        return -1; // first earlier than second (first < second)

    if (first->tv_nsec > second->tv_nsec)
        return 1; // first later than second (first > second)

    if (first->tv_nsec < second->tv_nsec)
        return -1; // first earlier than second (first < second)

    return 0; // first == second
}
```

Zwei Absolutzeiten (`struct tm`) werden am einfachsten über deren Repräsentation in Sekunden verglichen. Die Umwandlung erfolgt über die Funktion (`time_t mktime(struct tm *tm)`). Allerdings ist dabei zu beachten, dass es auf einem 32-Bit-System am 19. Januar 2038 zu einem Überlauf kommt. Wird einer der beiden Zeitstempel vor dem 19. Januar 2038 genommen, der andere danach, kommt es zu einem falschen Ergebnis, wenn nur die beiden Werte per »<« beziehungsweise »>« verglichen werden.

Das ist ein generelles Problem und kann dann gelöst werden, wenn sichergestellt ist, dass die zu vergleichenden Zeiten nicht weiter als die Hälfte des gesamten Zeitbereiches auseinanderliegen. In diesem Fall lassen sich die Makros einsetzen, die im Linux-Kernel für den Vergleich zweier Zeiten eingesetzt werden. Das Makro `time_after(a,b)` liefert *true* zurück, falls es sich bei `a` um eine spätere Zeit als `b` handelt. Das Makro `time_after_eq(a,b)` liefert *true* zurück, falls es sich bei `a` um eine spätere Zeit oder um die gleiche Zeit wie `b` handelt. Die Zeitstempel `a` und `b` müssen beide vom Typ `unsigned long` sein. Das wird durch den Compiler auch mithilfe des Schlüsselwortes »typecheck« sichergestellt. Natürlich können die Makros auch auf andere Datentypen angepasst werden (Kernel-Headerdatei [linux/jiffies.h]).

```
#define time_after(a,b)         \
        (typecheck(unsigned long, a) && \
         typecheck(unsigned long, b) && \
         ((long)(b) - (long)(a) < 0))
#define time_before(a,b)        time_after(b,a)

#define time_after_eq(a,b)      \
        (typecheck(unsigned long, a) && \
         typecheck(unsigned long, b) && \
         ((long)(a) - (long)(b) >= 0))
#define time_before_eq(a,b)     time_after_eq(b,a)
```

Beispiel 4-12 *Programmtechnische Realisierung von Zeitvergleichen*

```
#define time_after(a,b)         \
         ((long)(b) - (long)(a) < 0)
#define time_after_eq(a,b)      \
         ((long)(a) - (long)(b) >= 0)

...
struct timespec start, end;

clock_gettime( CLOCK_REALTIME, &start );
clock_gettime( CLOCK_REALTIME, &end );

if (time_after(start.tv_nsec, end.tv_nsec) ) {
   printf("%ld (start) is later than %ld (end)\n",
      start.tv_nsec, end.tv_nsec );
} else if (time_after_eq(start.tv_nsec, end.tv_nsec) ) {
   printf("%ld (start) is equal to %ld (end)\n",
      start.tv_nsec, end.tv_nsec );
} else {
   printf("%ld (start) is earlier than %ld (end)\n",
      start.tv_nsec, end.tv_nsec );
}
```

4.4.3 Differenzzeitmessung

In Realzeitapplikationen ist es häufig notwendig, eine Zeitdauer zu messen, Zeitpunkte zu erfassen oder eine definierte Zeit verstreichen zu lassen. Dabei sind folgende Aspekte zu beachten:

- Die Genauigkeit der eingesetzten Zeitgeber,
- die maximalen Zeitdifferenzen,
- Schwankungen der Zeitbasis, beispielsweise durch Schlafzustände,
- Modifikationen an der Zeitbasis des eingesetzten Rechnersystems (Zeitsprünge) und
- die Ortsabhängigkeit absoluter Zeitpunkte.

Zur Bestimmung einer Zeitdauer verwendet man häufig eine Differenzzeitmessung. Dabei wird vor und nach der auszumessenden Aktion jeweils ein Zeitstempel genommen. Die Zeitdauer ergibt sich aus der Differenz dieser beiden Zeitstempel.

Dies ist allerdings nicht immer ganz einfach. Liegt der Zeitstempel beispielsweise in Form der Datenstruktur `struct timespec` (als Ergebnis der Funktion `clock_gettime()`) vor, werden die Sekunden zunächst getrennt von den Nanosekunden subtrahiert. Ist der Nanosekundenanteil negativ, muss der Sekundenanteil um eins erniedrigt und der Nanosekundenanteil korrigiert werden. Dazu wird zu der Anzahl der Nanosekunden pro Sekunde (also eine Milliarde) der negative Nanosekundenanteil addiert (siehe Beispiel 4-13).

Beispiel 4-13 *Quellcodebeispiel zur Differenzzeitmessung*

```
#define NANOSECONDS_PER_SECOND 1000000000

struct timespec * diff_time( struct timespec before, struct timespec after,
    struct timespec *result )
{
    if (result==NULL)
        return NULL;

    if ((after.tv_sec<before.tv_sec) ||
        ((after.tv_sec==before.tv_sec) &&
        (after.tv_nsec<=before.tv_nsec))) { /* after before before */
        result.tv_sec = result.tv_nsec = 0;
    }
    result->tv_sec = after.tv_sec - before.tv_sec;
    result->tv_nsec= after.tv_nsec- before.tv_nsec;

    if (result->tv_nsec<0) {
        result->tv_sec--;
        /* result->tv_nsec is negative, therefore we use "+" */
        result->tv_nsec = NANOSECONDS_PER_SECOND+result->tv_nsec;
    }
    return result;
}
```

Für Zeitstempel vom Typ `struct timeval` kann der Code leicht umgeschrieben werden. Anstelle der `NANOSECONDS_PER_SECOND` sind `MICROSECONDS_PER_SECOND` einzusetzen. Sind die Zeitstempel vorzeichenlos, sieht die Rechnung für den Sekundenanteil etwas komplizierter aus, soll hier aber nicht weiter erläutert werden.

Für Zeitstempel vom Typ `struct timeval` steht zudem noch die Funktion `void timersub(struct timeval *a, struct timeval *b, struct timeval *res)` zur Verfügung. Diese legt die Differenz in normierter Form in `res` ab.

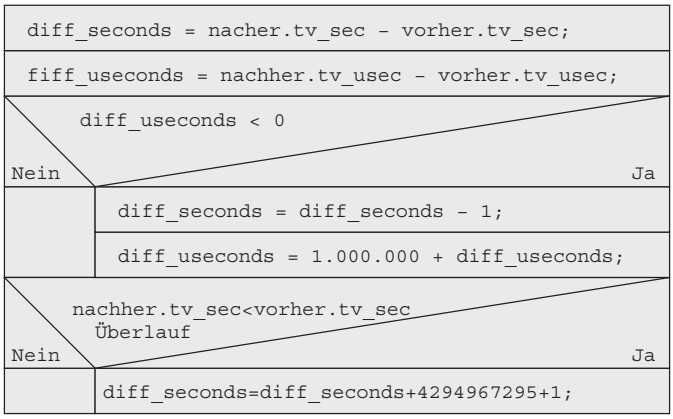


Abbildung 4-6 *Struktogramm Differenzzeitmessung (32-Bit-System)*

Einen Überlauf fängt Beispiel 4-13 übrigens durch eine Abfrage, ob der spätere Zeitstempel kleiner ist als der frühere Zeitstempel, direkt zu Beginn des Beispiels ab. Die Funktion returniert in diesem Fall »0«. Alternativ kann aber auch der Überlauf berücksichtigt werden. Dann ergibt sich der in Abbildung 4-6 in Form eines Struktogramms (siehe Abschnitt 7.2) dargestellte Ablauf, der für ein 32-Bit-System gültig ist.

Etwas einfacher ist die Differenzbildung, wenn aus der Datenstruktur eine einzelne Variable mit der gewünschten Auflösung, beispielsweise Mikrosekunden, generiert wird. Im Fall von `struct timeval` wird dazu der Sekundenanteil mit einer Million multipliziert und der Mikrosekundenanteil aufaddiert. Bei der Multiplikation können natürlich Informationen verloren gehen, allerdings geht der gleiche Informationsgehalt auch beim zweiten Zeitstempel verloren. Für die Differenzbildung ist dieser Umstand nicht relevant, solange der zu messende zeitliche Abstand kleiner als 1000 Sekunden ist und es während der Messung keinen Überlauf beim Sekundenanteil gibt.

```
/*Zeitstempel liegen als struct timeval vor */
time_in_usec=((nachher.tv_sec*1000000)+nachher.tv_usec)-
    ((vorher.tv_sec*1000000)+vorher.tv_usec);

/* Zeitstempel liegen als struct timespec vor */
time_in_usec=((nachher.tv_sec*1000000)+(nachher.tv_nsec/1000))-
    ((vorher.tv_sec*1000000)+(vorher.tv_usec/1000));
```

```
static int difference_micro(struct timeval *before,
    struct timeval *after)
{
    return (signed long long) after->time.tv_sec * 1000000ll +
        (signed long long) after->time.tv_usec -
        (signed long long) before->time.tv_sec * 1000000ll -
   (signed long long) before->time.tv_usec;
}
```

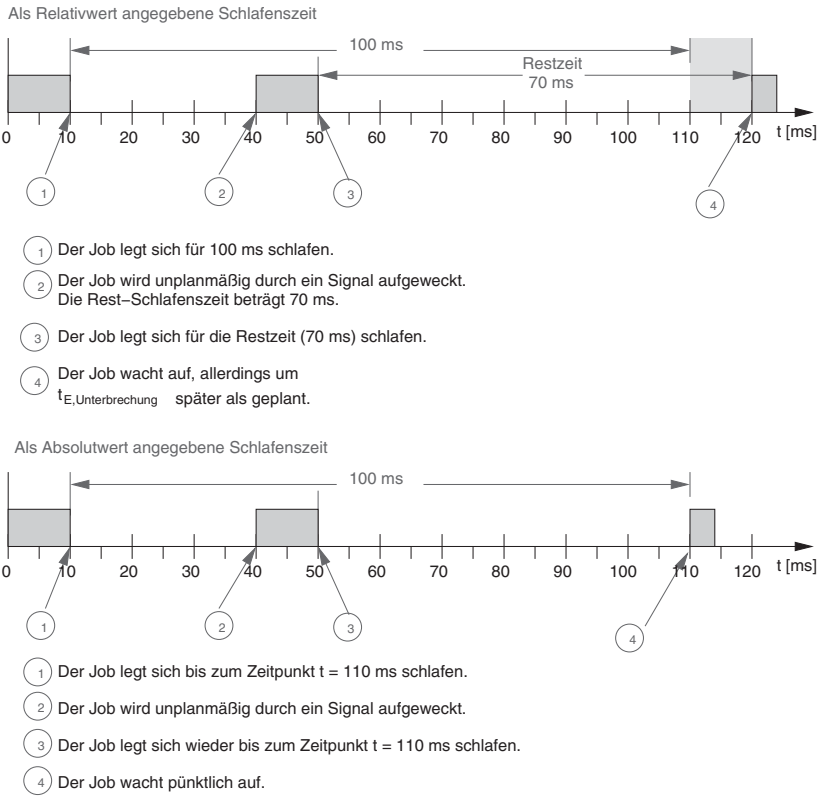
4.4.4 Schlafen

Threads legen sich für eine Zeitspanne oder aber bis zu einem Punkt, an dem sie aufgeweckt werden, schlafen (absolutes oder relatives Schlafen). Innerhalb von Realzeitapplikationen kann zusätzlich die zu verwendende Zeitquelle (`CLOCK_MONOTONIC` oder `CLOCK_REALTIME`) definiert werden. Beachten Sie, dass negative Zeitangaben beim Schlafenlegen typischerweise nicht definiert sind.

Tabelle 4-3 *Funktionen zum Schlafenlegen von Jobs*

Funktion	Beschreibung
`clock_nanosleep`	Auflösung Nanosekunden, Auswahl des Zeitgebers, Auswahl absolut/relativ.
`nanosleep`	Nanosekunden
`sleep`	Auflösung in Sekunden
`usleep`	Mikrosekunden

Abbildung 4-7 *Vorteile beim Schlafen mit absoluter Zeitangabe*



Ob relativ oder absolut geschlafen wird, hat durchaus Relevanz: Wird das Schlafen unterbrochen und danach mit der Restzeit neu aufgesetzt, kommt es durch den zusätzlichen Aufruf zu einer Verzögerung. Bei der

Verwendung einer absoluten Weckzeit fallen diese zusätzlichen Aufrufe zeitlich nicht ins Gewicht (Abbildung 4-7).

Ein Thread kann sich durch Aufruf der Funktion `int nanosleep(const struct timespec *req, struct timespec *rem);` für die über `req` definierte Zeitspanne schlafen legen. Hierbei wird zwar offiziell die Zeitquelle `CLOCK_REALTIME` verwendet, eine Änderung der Schlafenszeit wird aber nachgeführt. De facto basiert `nanosleep()` daher auf `CLOCK_MONOTONIC`. Der schlafende Thread wird aufgeweckt, wenn entweder die angegebene Relativzeit abgelaufen ist oder der Thread ein Signal gesendet bekommen hat. Ist Letzteres der Fall gewesen und hat der Aufrufer den Parameter `rem` mit einer gültigen Hauptspeicheradresse versehen, legt das Betriebssystem in diesem Speicher die noch übrig gebliebene Schlafenszeit ab.

Beispiel 4-14
Schlafenlegen per nanosleep()

```
...
struct timespec req, rem;
int error;
...
req.tv_sec  = 60;
req.tv_nsec = 1000;
while ((error=nanosleep(&req,&rem))==-1) {
    if (errno==EINTR) {
        req = rem;
    } else {
        perror("nanosleep");
        break;
    }
}
```

Genauer kann das Verhalten beim Schlafenlegen über die Funktion `int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *request, struct timespec *remain);` eingestellt werden. Über den Parameter `clock_id` wird die Zeitquelle (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`) eingestellt. Der Parameter `flag` erlaubt die Angabe, ob die Zeitangabe `request` relativ (`flag==0`) oder absolut (`flag==TIMER_ABSTIME`) zu interpretieren ist. Um die Restzeit bei einem durch ein Signal provozierten vorzeitigen Abbruch aufzunehmen, kann per `remain` eine Speicheradresse dafür übergeben werden. `request` schließlich enthält die Zeitangabe, bis zu der (`TIMER_ABSTIME`) oder die der Job schlafen soll. Bei Angabe der absoluten Zeitangabe ist auf die Normierung zu achten: Wird der Nanosekundenanteil größer oder gleich eine Milliarde, repräsentiert dieser Anteil also eine Sekunde oder mehr, ist mithilfe der Division und der Modulo-Operation eine Anpassung notwendig:

```
if (sleeptime.tv_nsec>999999999) {
    sleeptime.tv_sec += sleeptime.tv_nsec/1000000000;
    sleeptime.tv_nsec = sleeptime.tv_nsec%1000000000;
}
```

Die professionelle Programmierung mit POSIX-Funktionen zum Schlafenlegen eines Job – Beispiel 4-15, Beispiel 4-16 – zeigt im Übrigen die legere Variante des Schlafens mit relativer Zeitangabe.

Beispiel 4-15
Schlafenlegen einer Task mit absoluter Zeitangabe

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void sigint_handler(int signum)
{
   printf("SIGINT (%d)\n", signum);
}

int main( int argc, char **argv, char **envp )
{
   struct timespec sleeptime;
   struct sigaction sa;

   sigemptyset(&sa.sa_mask);
   sa.sa_flags = 0;
   sa.sa_handler = sigint_handler;
   if (sigaction(SIGINT, &sa, NULL) == -1) {
      perror( "sigaction" );
      return -1;
   }
   printf("%d sleeps for 10 seconds and 1 millisecond...\n", getpid());
   clock_gettime( CLOCK_MONOTONIC, &sleeptime );
   sleeptime.tv_sec  += 10;
   sleeptime.tv_nsec += 1000000;
   if (sleeptime.tv_nsec>999999999) {
      sleeptime.tv_sec += sleeptime.tv_nsec/1000000000;
      sleeptime.tv_nsec = sleeptime.tv_nsec%1000000000;
   }
   while (clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME,
      &sleeptime,NULL)==EINTR) {
      printf("interrupted...\n");
   }
   printf("woke up...\n");
   return 0;
}
```

Beispiel 4-16
Schlafenlegen einer Task mit relativer Zeitangabe

```
struct timespec sleeptime;
...
sleeptime.tv_sec  = 0;
sleeptime.tv_nsec = 250000000; /* 250 Millisekunden */
if ((error=clock_nanosleep(CLOCK_MONOTONIC,0,&sleeptime,NULL))!=0 ) {
    printf("clock_nanosleep reporting error %d\n", error);
}
```

Die Funktion `clock_nanosleep()` steht nur über die Realzeitbibliothek `librt` zur Verfügung. Beim Linken ist daher die Option `-lrt` mit anzugeben.

Neben `nanosleep()` finden sich in der Standard-C-Bibliothek noch weitere Funktionen, mit denen Threads schlafen gelegt werden können: `int usleep(useconds_t usec)`, `unsigned int sleep(unsigned int seconds)` und `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`.

Die Funktion `sleep()` bekommt die Schlafenszeit als eine Anzahl von Sekunden übergeben, bei `usleep()` sind es Mikrosekunden. `select()` ist eigentlich nicht primär dazu gedacht, Jobs schlafen zu legen. Diese Eigenschaft wurde insbesondere früher genutzt, um eine leicht portierbare Funktion zum Schlafenlegen bei Auflösung im Mikrosekundenbereich zu haben.

4.4.5 Weckrufe per Timer

Das Betriebssystem kann periodisch Funktionen, sogenannte Timer, aufrufen. Diese Funktionen werden typischerweise als *Signal-Handler* oder im Rahmen eines Threads aktiviert.

Dazu müssen zwei Datenstrukturen vorbereitet werden. `struct sigevent` speichert die Daten, die mit dem Timer selbst und mit der aufzurufenden Funktion zusammenhängen:

```
struct sigevent {
    int  sigev_notify;
    int  sigev_signo;
    union sigval sigev_value;
    void (*sigev_notify_function) (union sigval);
    void *sigev_notify_attributes;
    pid_t sigev_notify_thread_id;
};
```

Das Element `sigev_notify` legt fest, ob die Timerfunktion als Signal-Handler oder im Rahmen eines Threads aktiviert wird. Interessant sind die Werte `SIGEV_SIGNAL` und `SIGEV_THREAD`. Im Fall von `SIGEV_SIGNAL` legt `sigev_signo` die Nummer des Signals fest, welches nach Ablauf des Auslöseintervalls dem Job gesendet wird. Das Feld `sigev_value` nimmt einen

Parameter auf, der entweder dem Signal-Handler oder der im Thread abgearbeiteten Funktion übergeben wird. Der Signal-Handler selbst wird mit der Funktion `sigaction()` (siehe Abschnitt 4.7) etabliert. Unter Linux kann auch noch der Thread spezifiziert werden, dem das Signal zuzustellen ist. Dazu ist der Parameter `sigev_notify_thread_id` mit der Thread-ID zu besetzen und anstelle von `SIGEV_SIGNAL` ist `SIGEV_THREAD_ID` für `sigev_notify` auszuwählen.

Ist für `sigev_notifySIGEV_THREAD` ausgewählt, nimmt `sigev_notify_function` die Adresse der Funktion auf, die im Kontext eines Threads abgearbeitet wird. Per `sigev_notify_attributes` lässt sich die Thread-Erzeugung konfigurieren. Meistens reicht es aus, hier `NULL` zu übergeben.

Die Datenstruktur `struct sigevent` wird per `int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid)` dem Kernel übergeben. `clockid` spezifiziert den Zeitgeber (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`), `evp` die initialisierte `struct sigevent` und in `timerid` findet sich nach dem Aufruf die Kennung des erzeugten, aber deaktivierten Timers.

Der beziehungsweise die Zeitpunkte, zu denen die Timerfunktion aufgerufen werden soll, wird über `struct itimerspec` konfiguriert:

```
struct itimerspec {
    struct timespec it_interval;  /* Timer interval */
    struct timespec it_value;     /* Initial expiration */
};
```

`it_value` gibt in Sekunden und Nanosekunden die Zeit an, zu der erstmalig die Funktion oder der Signal-Handler aufgerufen werden soll, `it_interval` spezifiziert in Sekunden und Nanosekunden die Periode. Ob die Zeiten absolut oder relativ gesehen werden, wird beim Aufruf der Funktion `int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value, struct itimerspec * old_value)` über den Parameter `flags` (0 oder `TIMER_ABSTIME` festgelegt. `new_value` übernimmt die zeitliche Parametrierung. Falls `old_value` ungleich `NULL` ist, findet sich in dieser Datenstruktur nach dem Aufruf das vorhergehende Intervall.

Die Funktionen sind nur nutzbar, wenn die Realzeitbibliothek `librt` zur Applikation gebunden wird. Dazu ist beim Aufruf des Compilers die Option `-lrt` mit anzugeben.

Beispiel 4-17 *Periodische Taskaktivierung*

```
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <unistd.h>

void timer_function( union sigval parameter )
{
   printf("timer with id %d active\n", getpid());
   return;
```

```
}

int main( int argc, char **argv, char **envp )
{
   timer_t itimer;
   struct sigevent sev;
   struct itimerspec interval;

   printf("start process.\n");
   sev.sigev_notify            = SIGEV_THREAD;
   sev.sigev_notify_function   = timer_function;
   sev.sigev_value.sival_int   = 99;
   sev.sigev_notify_attributes = NULL;
   if (timer_create(CLOCK_MONOTONIC, &sev, &itimer ) == -1) {
      perror("timer_create");
      return -1;
   }
   interval.it_interval.tv_sec = 1;
   interval.it_interval.tv_nsec= 0;
   interval.it_value.tv_sec    = 1;
   interval.it_value.tv_nsec   = 0;
   timer_settime( itimer, 0, &interval, NULL ); /* activate timer */
   sleep( 5 );
   printf("end process.\n");
   return 0;
}
```

4.5 Inter-Prozess-Kommunikation

Die klassische Inter-Prozess-Kommunikation (Datenaustausch) bietet unter anderem die folgenden Methoden an:

- Pipes/Mailboxes/Messages
- Shared-Memory
- Sockets

Diese werden im Folgenden vorgestellt.

4.5.1 Pipes, Mailbox und Messages

Zum Datenaustausch bieten Betriebssysteme einen Mailbox-Mechanismus (`send`/`receive`-Interface) an. Dabei werden – im Regelfall unidirektional – Daten von Task 1 zu Task 2 transportiert und gequeued (im Gegensatz zu gepuffert; Daten gehen also nicht verloren, da sie nicht im Puffer überschrieben werden).