9

KERNEL DESIGN ISSUES

In this chapter we present some basic issues that should be considered during the design and the development of a hard real-time kernel for critical control applications. For didactical purposes, we illustrate the structure and the main components of a small real-time kernel, called DICK (*DI*dactic *C* *K*ernel), mostly written in C language, which is able to handle periodic and aperiodic tasks with explicit time constraints. The problem of time predictable intertask communication is also discussed, and a particular communication mechanism for exchanging state messages among periodic tasks is illustrated. Finally, we show how the runtime overhead of the kernel can be evaluated and taken into account in the schedulability analysis.

9.1 STRUCTURE OF A REAL-TIME KERNEL

A kernel represents the innermost part of any operating system that is in direct connection with the hardware of the physical machine. A kernel usually provides the following basic activities:

- Process management,
- Interrupt handling, and
- Process synchronization.

Process management is the primary service that an operating system has to provide. It includes various supporting functions, such as process creation and termination, job scheduling, dispatching, context switching, and other related activities.

The objective of the interrupt handling mechanism is to provide service to the interrupt requests that may be generated by any peripheral device, such as the keyboard, serial ports, analog-to-digital converters, or any specific sensor interface. The service provided by the kernel to an interrupt request consists in the execution of a dedicated routine (driver) that will transfer data from the device to the main memory (or viceversa). In classical operating systems, application tasks can always be preempted by drivers, at any time. In real-time systems, however, this approach may introduce unpredictable delays in the execution of critical tasks, causing some hard deadline to be missed. For this reason, in a real-time system, the interrupt handling mechanism has to be integrated with the scheduling mechanism, so that a driver can be scheduled as any other task in the system and a guarantee of feasibility can be achieved even in the presence of interrupt requests.

Another important role of the kernel is to provide a basic mechanism for supporting process synchronization and communication. In classical operating systems this is done by semaphores, which represent an efficient solution to the problem of synchronization, as well as to the one of mutual exclusion. As discussed in Chapter 7, however, semaphores are prone to priority inversion, which introduces unbounded blocking on tasks' execution and prevents a guarantee for hard real-time tasks. As a consequence, in order to achieve predictability, a real-time kernel has to provide special types of semaphores that support a resource access protocol (such as Priority Inheritance, Priority Ceiling, or Stack Resource Policy) for avoiding unbounded priority inversion. Other kernel activities involve the initialization of internal data structures (such as queues, tables, task control blocks, global variables, semaphores, and so on) and specific services to higher levels of the operating system.

In the rest of this chapter, we describe the structure of a small real-time kernel, called DICK (*DI*dactic *C* *K*ernel). Rather than showing all implementation details, we focus on the main features and mechanisms that are necessary to handle tasks with explicit time constraints.

DICK is designed under the assumption that all tasks are resident in main memory when it receives control of the processor. This is not a restrictive assumption, as this is the typical solution adopted in kernels for real-time embedded applications.

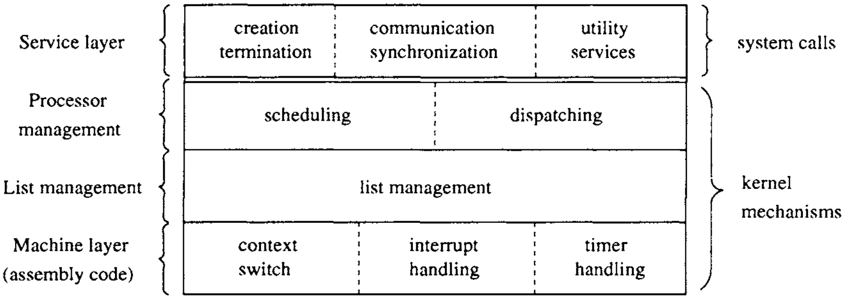

Figure 9.1 Hierarchical structure of DICK.

The various functions developed in DICK are organized according to the hierarchical structure illustrated in Figure 9.1. Those low-level activities that directly interact with the physical machine are realized in assembly language. Nevertheless, for the sake of clarity, all kernel activities are described in pseudo C.

The structure of DICK can be logically divided into four layers:

- **Machine layer**. This layer directly interacts with the hardware of the physical machine; hence, it is written in assembly language. The primitives realized at this level mainly deal with activities such as context switch, interrupt handling, and timer handling. These primitives are not visible at the user level.

- **List management layer**. To keep track of the status of the various tasks, the kernel has to manage a number of lists, where tasks having the same state are enqueued. This layer provides the basic primitives for inserting and removing a task to and from a list.

- **Processor management layer**. The mechanisms developed in this layer only concerns scheduling and dispatching operations.

- **Service layer**. This layer provides all services visible at the user level as a set of system calls. Typical services concern task creation, task abortion, suspension of periodic instances, activation and suspension of aperiodic instances, and system inquiry operations.

9.2 PROCESS STATES

In this section, we describe the possible states in which a task can be during its execution and how a transition from a state to another can be performed.

In any kernel that supports the execution of concurrent activities on a single processor, where semaphores are used for synchronization and mutual exclusion, there are at least three states in which a task can enter:

- **Running**. A task enters this state as it starts executing on the processor.
- **Ready**. This is the state of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task. All tasks that are in this condition are maintained in a queue, called the *ready queue*.
- **Waiting**. A task enters this state when it executes a synchronization primitive to wait for an event. When using semaphores, this operation is a *wait* primitive on a locked semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head of this queue is resumed when the semaphore is unlocked by another task that executed a *signal* on that semaphore. When a task is resumed, it is inserted in the ready queue.

In a real-time kernel that supports the execution of periodic tasks, another state must be considered, the IDLE state. A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period. In order to be awakened by the timer, a periodic job must notify the end of its cycle by executing a specific system call, *end_cycle*, which puts the job in the IDLE state and assigns the processor to another ready job. At the right time, each periodic job in the IDLE state will be awakened by the kernel and inserted in the ready queue. This operation is carried out by a routine activated by a timer, which verifies, at each tick, whether some job has to be awakened. The state transition diagram relative to the four states described above is shown in Figure 9.2.

Additional states can be introduced by other kernel services. For example, a *delay* primitive, which suspends a job for a given interval of time, puts the job in a sleeping state (DELAY), until it will be awakened by the timer after the elapsed interval.

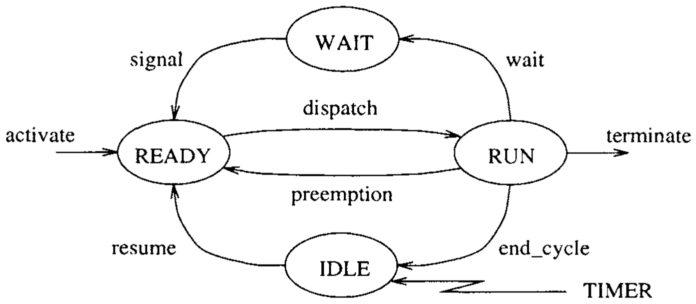

Figure 9.2 Minimum state transition diagram of a real-time kernel.

Another state, found in many operating systems, is the RECEIVE state, introduced by the classical message passing mechanism. A job enters this state when it executes a *receive* primitive on an empty channel. The job exits this state when a *send* primitive is executed by another job on the same channel.

In real-time systems that support dynamic creation and termination of hard periodic tasks, a new state needs to be introduced for preserving the bandwidth assigned to the guaranteed tasks. This problem arises because, when a periodic task τ_k is aborted (for example, with a *kill* operation), its utilization factor U_k cannot be immediately subtracted from the total processor load, since the task could already have delayed the execution of other tasks. In order to keep the guarantee test consistent, the utilization factor U_k can be subtracted only at the end of the current period of τ_k.

For example, consider the set of three periodic tasks illustrated in Figure 9.3, which are scheduled by the Rate-Monotonic algorithm. Computation times are 1, 4, and 4, and periods are 4, 8, and 16, respectively. Since periods are harmonic and the total utilization factor is $U = 1$, the task set is schedulable by RM (remember that $U_{lub} = 1$ when periods are harmonic).

Now suppose that task τ_2 (with utilization factor $U_2 = 0.5$) is aborted at time $t = 4$ and that, at the same time, a new task τ_{new}, having the same characteristics of τ_2, is created. If the total load of the processor is decremented by 0.5 at time $t = 4$, task τ_{new} would be guaranteed, having the same utilization factor as τ_2. However, as shown in Figure 9.4, τ_3 would miss its deadline. This happens because the effects of τ_2 execution on the schedule protract until the end of each period.

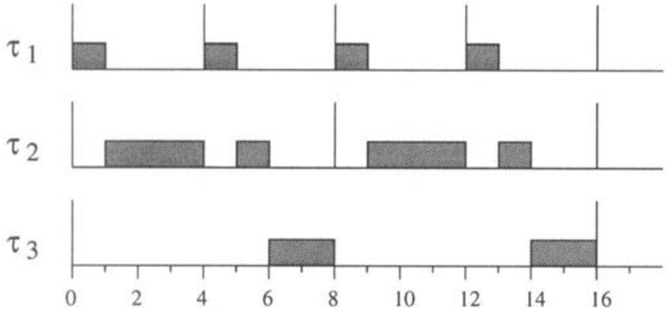

Figure 9.3 Feasible schedule of three periodic tasks under RM.

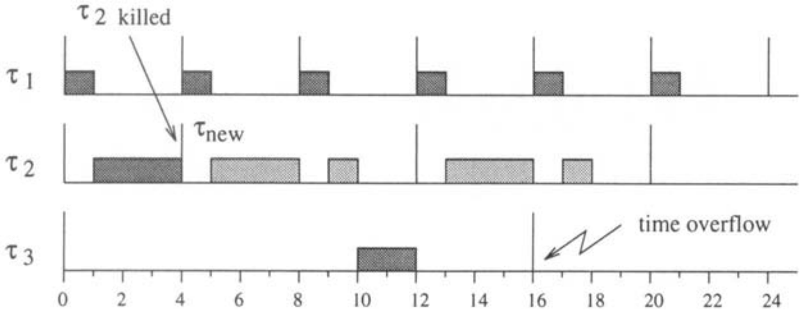

Figure 9.4 The effects of τ_2 do not cancel at the time it is aborted, but protract till the end of its period.

As a consequence, to keep the guarantee test consistent, the utilization factor of an aborted task can be subtracted from the total load only at the end of the current period. In the interval of time between the abort operation and the end of its period, τ_2 is said to be in a ZOMBIE state, since it does not exist in the system, but it continues to occupy processor bandwidth. Figure 9.5 shows that the task set is schedulable when the activation of τ_{new} is delayed until the end of the current period of τ_2.

A more complete state transition diagram including the states described above (DELAY, RECEIVE, and ZOMBIE) is illustrated in Figure 9.6. Notice that, at the end of its last period, a periodic task (aborted or terminated) leaves the system completely and all its data structures are deallocated.

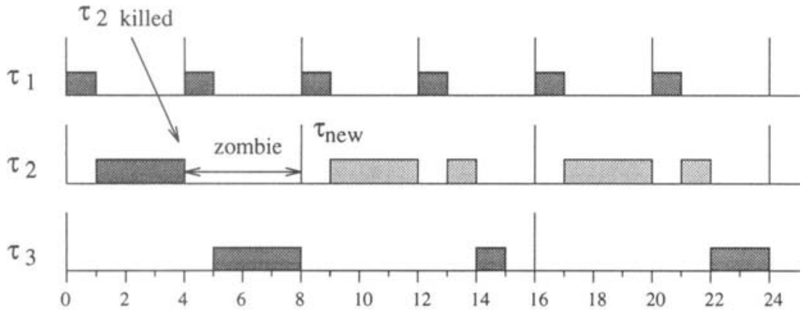

Figure 9.5 The new task set is schedulable when τ_{new} is activated at the end of the period of τ_2.

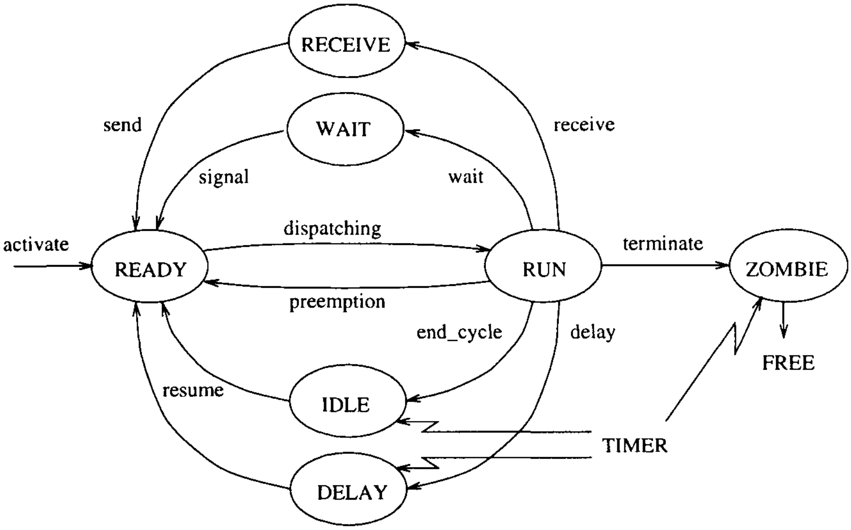

Figure 9.6 State transition diagram including RECEIVE, DELAY, and ZOMBIE states.

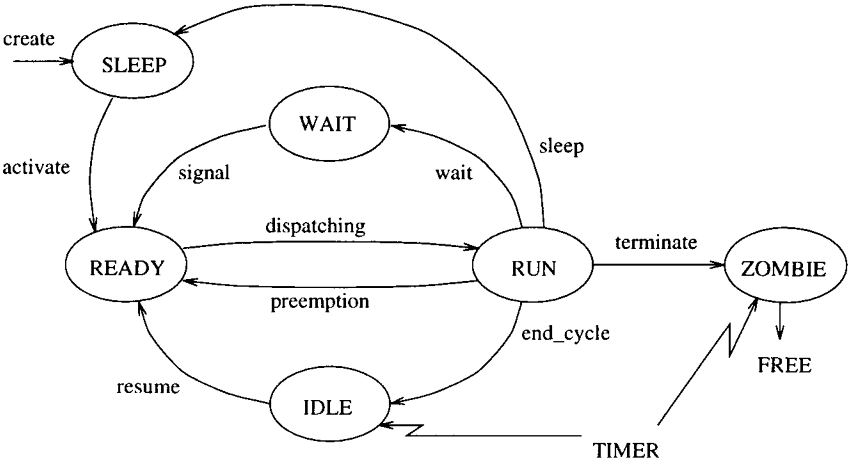

Figure 9.7 State transition diagram in DICK.

In order to simplify the description of DICK, in the rest of this chapter we describe only the essential functions of the kernel. In particular, the message passing mechanism and the delay primitive are not considered here; as a consequence, the states RECEIVE and DELAY are not present. However, these services can easily be developed on top of the kernel, as an additional layer of the operating system.

In DICK, activation and suspension of aperiodic tasks are handled by two primitives, *activate* and *sleep*, which introduce another state, called SLEEP. An aperiodic task enters the SLEEP state by executing the *sleep* primitive. A task exits the SLEEP state and goes to the READY state only when an explicit activation is performed by another task.

Task creation and activation are separated in DICK. The creation primitive (*create*) allocates and initializes all data structures needed by the kernel to handle the task; however, the task is not inserted in the ready queue, but it is left in the SLEEP state, until an explicit activation is performed. This is mainly done for reducing the runtime overhead of the activation primitive. The state transition diagram used in DICK is illustrated in Figure 9.7.

9.3 DATA STRUCTURES

In any operating system, the information about a task are stored in a data structure, the *Task Control Block* (TCB). In particular, a TCB contains all the parameters specified by the programmer at creation time, plus other temporary information necessary to the kernel for managing the task. In a real-time system, the typical fields of a TCB are shown in Figure 9.8 and contain the following information:

- An identifier; that is, a character string used by the system to refer the task in messages to the user;
- The memory address corresponding to the first instruction of the task;
- The task type (periodic, aperiodic, or sporadic);
- The task criticalness (hard, soft, or non-real-time);
- The priority (or value), which represents the importance of the task with respect to the other tasks of the application;
- The current state (ready, running, idle, waiting, and so on);
- The worst-case execution time;
- The task period;
- The relative deadline, specified by the user;
- The absolute deadline, computed by the kernel at the arrival time;
- The task utilization factor (only for periodic tasks);
- A pointer to the process stack, where the context is stored;
- A pointer to a directed acyclic graph, if there are precedence constraints;
- A pointer to a list of shared resources, if a resource access protocol is provided by the kernel.

In addition, other fields can be necessary for specific features of the kernel. For example, if aperiodic tasks are handled by one or more server mechanisms, a field can be used to store the identifier of the server associated with the task; or, if the scheduling mechanism supports tolerant deadlines, a field can store the tolerance value for that task.

Task Control Block
task identifier
task address
task type
criticalness
priority
state
computation time
period
relative deadline
absolute deadline
utilization factor
context pointer
precedence pointer
resource pointer
pointer to the next TCB

Figure 9.8 Structure of the Task Control Block.

Finally, since a TCB has to be inserted in the lists handled by the kernel, an additional field has to be reserved for the pointer to the next element of the list.

In DICK, a TCB is an element of the `vdes[MAXPROC]` array, whose size is equal to the maximum number of tasks handled by the kernel. Using this approach, each TCB can be identified by a unique index, corresponding to its position in the `vdes` array. Hence, any queue of tasks can be accessed by an integer variable containing the index of the TCB at the head of the queue. Figure 9.9 shows a possible configuration of the ready queue within the `vdes` array.

Similarly, the information concerning a semaphore is stored in a Semaphore Control Block (SCB), which contains at least the following three fields (see also Figure 9.10):

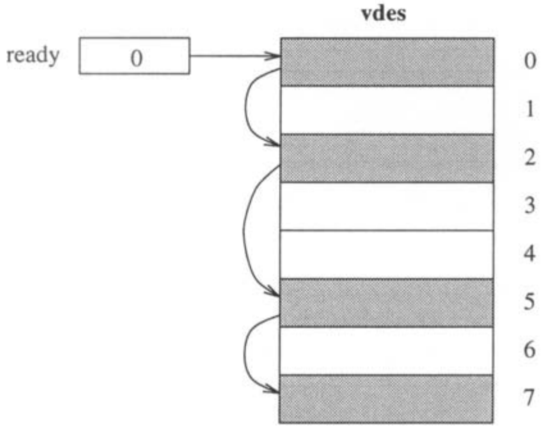

Figure 9.9 Implementation of the ready queue as a list of Task Control Blocks.

Semaphore Control Block

counter
semaphore queue
pointer to the next SCB

Figure 9.10 Semaphore Control Block.

- A counter, which represents the value of the semaphore;
- A queue, for enqueueing the tasks blocked on the semaphore;
- A pointer to the next SCB, to form a list of free semaphores.

Each SCB is an element of the `vsem[MAXSEM]` array, whose size is equal to the maximum number of semaphores handled by the kernel. According to this approach, tasks, semaphores, and queues can be accessed by an integer number, which represents the index of the corresponding control block. For the sake of clarity, however, tasks, semaphores and queues are defined as three different types.

```
typedef int     queue;                    /* head index        */
typedef int     sem;                      /* semaphore index   */
typedef int     proc;                     /* process index     */
typedef int     cab;                      /* cab buffer index  */
typedef char*   pointer;                  /* memory pointer    */
```

```
struct tcb {
    char     name[MAXLEN+1];    /* task name                    */
    proc     (*addr)();         /* first instruction address    */
    int      type;              /* task type                    */
    int      state;             /* task state                   */
    long     dline;             /* absolute deadline            */
    int      period;            /* task period                  */
    int      prt;               /* task priority                */
    int      wcet;              /* worst-case execution time    */
    float    util;              /* task utilization factor      */
    int      *context;          /* pointer to the context       */
    proc     next;              /* pointer to the next tcb      */
    proc     prev;              /* pointer to previous tcb      */
};
```

```
struct scb {
    int     count;              /* semaphore counter            */
    queue   qsem;               /* semaphore queue              */
    sem     next;               /* pointer to the next          */
};
```

```
struct tcb     vdes[MAXPROC];            /*  tcb array  */
struct scb     vsem[MAXSEM];             /*  scb array  */
```

```
proc      pexe;                  /* task in execution        */
queue     ready;                 /* ready queue              */
queue     idle;                  /* idle queue               */
queue     zombie;                /* zombie queue             */
queue     freetcb;               /* queue of free tcb's      */
queue     freesem;               /* queue of free semaphores */
float     util_fact;            /* utilization factor       */
```

9.4 MISCELLANEOUS

9.4.1 Time management

To generate a time reference, a timer circuit is programmed to interrupt the processor at a fixed rate, and the internal system time is represented by an integer variable, which is reset at system initialization and is incremented at each timer interrupt. The interval of time with which the timer is programmed to interrupt defines the unit of time in the system; that is, the minimum interval of time handled by the kernel (time resolution). The unit of time in the system is also called a system *tick*.

In DICK, the system time is represented by a long integer variable, called `sys_clock`, whereas the value of the tick is stored in a float variable called `time_unit`. At any time, `sys_clock` contains the number of interrupts generated by the timer since system initialization.

```
unsigned long    sys_clock;        /* system time         */
float            time_unit;        /* unit of time (ms)   */
```

If Q denotes the system tick and n is the value stored in `sys_clock`, the actual time elapsed since system initialization is $t = nQ$. The maximum time that can be represented in the kernel (the system *lifetime*) depends on the value of the system tick. Considering that `sys_clock` is an unsigned long represented on 32 bits, Table 9.1 shows the values of the system lifetime for some tick values.

tick	*lifetime*
1 ms	50 days
5 ms	8 months
10 ms	16 months
50 ms	7 years

Table 9.1 System lifetime for some typical tick values.

The value to be assigned to the tick depends on the specific application. In general, small values of the tick improve system responsiveness and allow to handle periodic activities with high activation rates. On the other hand, a very small tick causes a large runtime overhead due to the timer handling routine and reduces the system lifetime. Typical values used for the time resolution can vary from 1 to 50 milliseconds. To have a strict control on task deadlines and periodic activations, all time parameters specified on the tasks should be multiple of the system tick. If the tick can be selected by the user, the best possible tick value is equal to the greatest common divisor of all the task periods.

The timer interrupt handling routine has a crucial role in a real-time system. Other than updating the value of the internal time, it has to check for possible deadline misses on hard tasks, due to some incorrect prediction on the worst-case execution times. Other activities that can be carried out by the timer interrupt handling routine concern lifetime monitoring, activation of periodic tasks that are in idle state, awakening tasks suspended by a delay primitive, checking for deadlock conditions, and terminating tasks in zombie state.

In DICK, the timer interrupt handling routine increments the value of the `sys_clock` variable, checks the system lifetime, checks for possible deadline misses on hard tasks, awakes idle periodic tasks at the beginning of their next period and, at their deadlines, deallocates all data structures of the tasks in zombie state. In particular, at each timer interrupt, the corresponding handling routine

- Saves the context of the task in execution;
- Increments the system time;

- If the current time is greater than the system lifetime, generates a timing error;
- If the current time is greater than some hard deadline, generates a time-overflow error;
- Awakens those idle tasks, if any, that have to begin a new period;
- If at least a task has been awakened, calls the scheduler;
- Removes all zombie tasks for which their deadline is expired;
- Loads the context of the current task;
- Returns from interrupt.

The runtime overhead introduced by the execution the timer routine is proportional to its interrupt rate. In Section 9.7 we see how this overhead can be evaluated and taken into account in the schedulability analysis.

9.4.2 Task classes and scheduling algorithm

Real-world control applications usually consist of computational activities having different characteristics. For example, tasks may be periodic, aperiodic, time-driven, and event-driven and may have different levels of criticalness. To simplify the description of the kernel, only two classes of tasks are considered in DICK:

- HARD tasks, having a critical deadline, and
- Non-real-time (NRT) tasks, having a fixed priority.

HARD tasks can be activated periodically or aperiodically depending on how an instance is terminated. If the instance is terminated with the primitive *end_cycle*, the task is put in the idle state and automatically activated by the timer at the beginning of its next period; if the instance is terminated with the primitive *end_aperiodic*, the task is put in the sleep state, from where it can be resumed only by explicit activation. HARD tasks are scheduled using the Earliest Deadline First (EDF) algorithm, whereas NRT tasks are executed in background based on their priority.

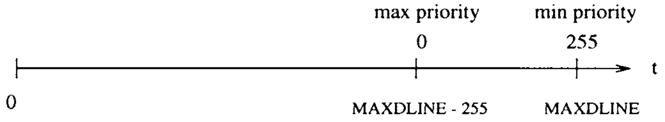

Figure 9.11 Mapping NRT priorities into deadlines.

In order to integrate the scheduling of these classes of tasks and avoid the use of two scheduling queues, priorities of NRT tasks are transformed into deadlines so that they are always greater than HARD deadlines. The rule for mapping NRT priorities into deadlines is shown in Figure 9.11 and is such that

$$d_i^{NRT} = MAXDLINE - PRT\_LEV + P_i,$$

where MAXDLINE is the maximum value of the variable `sys_clock` ($2^{31} - 1$), PRT_LEV is the number of priority levels handled by the kernel, and P_i is the priority of the task, in the range [0, PRT_LEV-1] (0 being the highest priority). Such a priority mapping slightly reduces system lifetime but greatly simplifies task management and queue operations.

9.4.3 Global constants

In order to clarify the description of the source code, a number of global constants are defined here. Typically, they define the maximum size of the main kernel data structures, such as the maximum number of processes and semaphores, the maximum length of a process name, the number of priority levels, the maximum deadline, and so on. Other global constants encode process classes, states, and error messages. They are listed below:

```
#define MAXLEN      12              /* max string length     */
#define MAXPROC     32              /* max number of tasks   */
#define MAXSEM      32              /* max No of semaphores  */
#define MAXDLINE    0x7FFFFFFF      /* max deadline          */
#define PRT_LEV     255             /* priority levels       */
#define NIL         -1              /* null pointer          */
#define TRUE        1
#define FALSE       0
#define LIFETIME    MAXDLINE - PRT_LEV
```

```
/*-----------------------------------------------------------*/
/*                    Task types                             */
/*-----------------------------------------------------------*/
#define HARD       1              /* critical task           */
#define NRT        2              /* non real-time task      */
/*-----------------------------------------------------------*/
/*                    Task states                            */
/*-----------------------------------------------------------*/
#define FREE       0               /* TCB not allocated      */
#define READY      1               /* ready state            */
#define EXE        2               /* running state          */
#define SLEEP      3               /* sleep state            */
#define IDLE       4               /* idle state             */
#define WAIT       5               /* wait state             */
#define ZOMBIE     6               /* zombie state           */
```

```
/*-----------------------------------------------------------*/
/*                  Error messages                           */
/*-----------------------------------------------------------*/
#define OK             0          /* no error                */
#define TIME_OVERFLOW  1          /* missed deadline         */
#define TIME_EXPIRED   2          /* lifetime reached        */
#define NO_GUARANTEE   3          /* task not schedulable    */
#define NO_TCB         4          /* too many tasks          */
#define NO_SEM         5          /* too many semaphores     */
```

9.4.4 Initialization

The real-time environment supported by DICK starts when the *ini_system* primitive is executed within a sequential C program. After this function is executed, the main program becomes a NRT task in which new concurrent tasks can be created.

The most important activities performed by *ini_system* concern

- Initializing all queues in the kernel;
- Setting all interrupt vectors;
- Preparing the TCB associated with the main process;
- Setting the timer period to the system tick.

```
void    ini_system(float tick)
{
proc   i;
    time_unit = tick;
    <enable the timer to interrupt every time_unit>
    <initialize the interrupt vector table>
    /* initialize the list of free TCBs and semaphores */
    for (i=0; i<MAXPROC-1; i++) vdes[i].next = i+1;
    vdes[MAXPROC-1].next = NIL;
    for (i=0; i<MAXSEM-1; i++) vsem[i].next = i+1;
    vsem[MAXSEM-1].next = NIL;
    ready = NIL;
    idle = NIL;
    zombie = NIL;
    freetcb = 0;
    freesem = 0;
    util_fact = 0;
    <initialize the TCB of the main process>
    pexe = <main index>;
}
```

9.5 KERNEL PRIMITIVES

The structure of DICK is logically divided in a number of hierarchical layers, as illustrated in Figure 9.1. The lowest layer includes all interrupt handling drivers and the routines for saving and loading a task context. The next layer contains the functions for list manipulation (insertion, extraction, and so on) and the basic mechanisms for task management (dispatching and scheduling). All kernel services visible from the user are implemented at a higher level. They concern task creation, activation, suspension, termination, synchronization, and status inquiry.

9.5.1 Low-level primitives

Basically, the low-level primitives implement the mechanism for saving and loading the context of a task; that is, the values of the processor registers.

```
/*-----------------------------------------------------------*/
/* save_context -- of the task in execution                  */
/*-----------------------------------------------------------*/
void    save_context(void)
{
int    *pc;                    /* pointer to context of pexe */
    <disable interrupts>
    pc = vdes[pexe].context;
    pc[0] = <register_0>        /* save register 0 */
    pc[1] = <register_1>        /* save register 1 */
    pc[2] = <register_2>        /* save register 2 */
         . . .
    pc[n] = <register_n>        /* save register n */
}
```

```
/*-----------------------------------------------------------*/
/* load_context -- of the task to be executed                */
/*-----------------------------------------------------------*/
void    load_context(void)
{
int     *pc;                    /* pointer to context of pexe */
    pc = vdes[pexe].context;
    <register_0> = pc[0];        /* load register 0 */
    <register_1> = pc[1];        /* load register 1 */
        ...
    <register_n> = pc[n];        /* load register n */
    <return from interrupt>
}
```

9.5.2 List management

Since tasks are scheduled based on EDF, all queues in the kernel are ordered by decreasing deadlines. In this way, the task with the earliest deadline can be simply extracted from the head of a queue, whereas an insertion operation requires to scan at most all elements of the list. All lists are implemented with bidirectional pointers (next and prev). The *insert* function is called with two parameters: the index of the task to be inserted and the pointer of the queue. It uses two auxiliary pointers, p and q, whose meaning is illustrated in Figure 9.12.

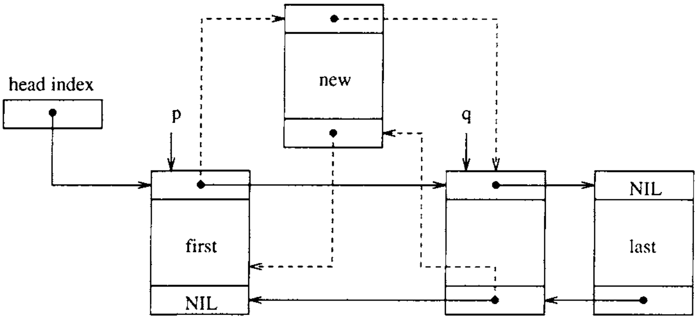

Figure 9.12 Inserting a TCB in a queue.

```
/*-----------------------------------------------------------*/
/* insert -- a task in a queue based on its deadline         */
/*-----------------------------------------------------------*/
void     insert(proc i, queue *que)
{
long    dl;          /* deadline of the task to be inserted */
int     p;           /* pointer to the previous TCB         */
int     q;           /* pointer to the next TCB             */
    p = NIL;
    q = *que;
    dl = vdes[i].dline;
    /* find the element before the insertion point */
    while ((q != NIL) && (dl >= vdes[q].dline)) {
        p = q;
        q = vdes[q].next;
    }
    if (p != NIL) vdes[p].next = i;
    else *que = i;
    if (q != NIL) vdes[q].prev = i;
    vdes[i].next = q;
    vdes[i].prev = p;
}
```

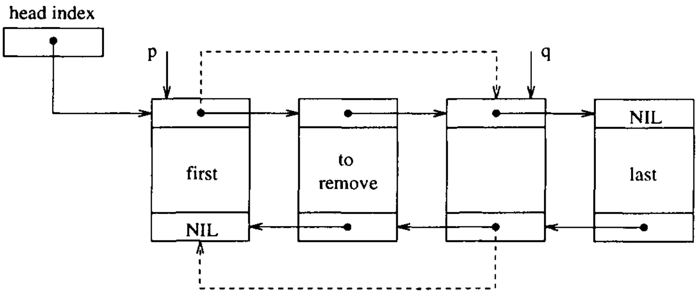

Figure 9.13 Extracting a TCB from a queue.

The major advantage of using bidirectional pointers is in the implementation of the extraction operation, which can be realized in one step without scanning the whole queue. Figure 9.13 illustrates the extraction of a generic element, whereas Figure 9.14 shows the extraction of the element at the head of the queue.

```
/*-----------------------------------------------------------*/
/* extract -- a task from a queue                           */
/*-----------------------------------------------------------*/
proc    extract(proc i, queue *que)
{
int    p, q;                      /* auxiliary pointers    */
    p = vdes[i].prev;
    q = vdes[i].next;
    if (p == NIL) *que = q;            /* first element     */
    else vdes[p].next = vdes[i].next;
    if (q != NIL) vdes[q].prev = vdes[i].prev;
    return(i);
}
```

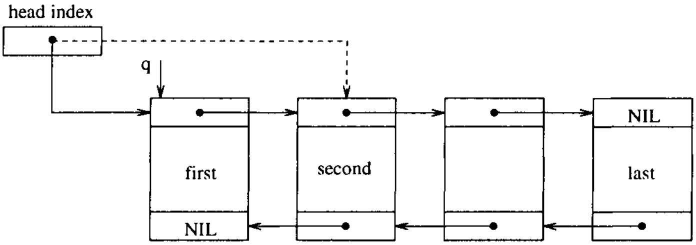

Figure 9.14 Extracting the TCB at the head of a queue.

```
/*-----------------------------------------------------------*/
/* getfirst -- extracts the task at the head of a queue      */
/*-----------------------------------------------------------*/
proc     getfirst(queue *que)
{
int    q;                /* pointer to the first element    */
    q = *que;
    if (q == NIL) return(NIL);
    *que = vdes[q].next;
    vdes[*que].prev = NIL;
    return(q);
}
```

Finally, to simplify the code reading of the next levels, two more functions are defined: *firstdline* and *empty*. The former returns the deadline of the task at the head of the queue, while the latter returns TRUE if a queue is empty, FALSE otherwise.

```
/*-----------------------------------------------------------*/
/* firstdline -- returns the deadline of the first task      */
/*-----------------------------------------------------------*/
long     firstdline(queue *que)
{
    return(vdes[que].dline);
}
```

```
/*-----------------------------------------------------------*/
/* empty -- returns TRUE if a queue is empty                 */
/*-----------------------------------------------------------*/
int     empty(queue *que)
{
    if (que == NIL)
        return(TRUE);
    else
        return(FALSE);
}
```

9.5.3 Scheduling mechanism

The scheduling mechanism in DICK is realized through the functions *schedule* and *dispatch*. The *schedule* primitive verifies whether the running task is the one with the earliest deadline. If so, no action is done, otherwise the running task is inserted in the ready queue and the first ready task is dispatched. The *dispatch* primitive just assigns the processor to the first ready task.

```
/*-----------------------------------------------------------*/
/* schedule -- selects the task with the earliest deadline   */
/*-----------------------------------------------------------*/
void    schedule(void)
{
    if (firstdline(ready) < vdes[pexe].dline) {
        vdes[pexe].state = READY;
        insert(pexe, &ready);
        dispatch();
    }
}
```

```
/*-----------------------------------------------------------*/
/* dispatch -- assigns the cpu to the first ready task       */
/*-----------------------------------------------------------*/
void    dispatch(void)
{
    pexe = getfirst(&ready);
    vdes[pexe].state = RUN;
}
```

The timer interrupt handling routine is called *wake_up* and performs the activities described in Section 9.4.1. In summary, it increments the *sys_clock* variable, checks for the system lifetime and possible deadline misses, removes those tasks in zombie state whose deadlines are expired, and, finally, resumes those periodic tasks in idle state at the beginning of their next period. Note, that if at least a task has been resumed, the scheduler is invoked and a preemption takes place.

```
/*------------------------------------------------------------*/
/* wake_up -- timer interrupt handling routine                */
/*------------------------------------------------------------*/
void     wake_up(void)
{
proc    p;
int     count = 0;
    save_context();
    sys_clock++;
    if (sys_clock >= LIFETIME) abort(TIME_EXPIRED);
    if (vdes[pexe].type == HARD)
        if (sys_clock > vdes[pexe].dline)
            abort(TIME_OVERFLOW);
    while ( !empty(zombie) &&
            (firstdline(zombie) <= sys_clock)) {
        p = getfirst(&zombie);
        util_fact = util_fact - vdes[p].util;
        vdes[p].state = FREE;
        insert(p, &freetcb);
    }
    while (!empty(idle) && (firstdline(idle) <= sys_clock)) {
        p = getfirst(&idle);
        vdes[p].dline += (long)vdes[p].period;
        vdes[p].state = READY;
        insert(p, &ready);
        count++;
    }
    if (count > 0) schedule();
    load_context();
}
```

9.5.4 Task management

It concerns creation, activation, suspension, and termination of tasks. The *create* primitive allocates and initializes all data structures needed by a task and puts the task in SLEEP. A guarantee is performed for HARD tasks.

```
/*-----------------------------------------------------------*/
/* create -- creates a task and puts it in sleep state       */
/*-----------------------------------------------------------*/
proc     create(
        char    name[MAXLEN+1],         /* task name           */
        proc    (*addr)(),              /* task address        */
        int     type,                   /* type (HARD, NRT)    */
        float   period,                 /* period or priority */
        float   wcet)                   /* execution time      */
{
proc    p;
    <disable cpu interrupts>
    p = getfirst(&freetcb);
    if (p == NIL) abort(NO_TCB);
    if (vdes[p].type == HARD)
        if (!guarantee(p)) return(NO_GUARANTEE);
    vdes[p].name = name;
    vdes[p].addr = addr;
    vdes[p].type = type;
    vdes[p].state = SLEEP;
    vdes[p].period = (int)(period / time_unit);
    vdes[p].wcet = (int)(wcet / time_unit);
    vdes[p].util = wcet / period;
    vdes[p].prt = (int)period;
    vdes[p].dline = MAX_LONG + (long)(period - PRT_LEV);
    <initialize process stack>
    <enable cpu interrupts>
    return(p);
}
```

```
/*-----------------------------------------------------------*/
/* guarantee -- guarantees the feasibility of a hard task    */
/*-----------------------------------------------------------*/
int     guarantee(proc p)
{
    util_fact = util_fact + vdes[p].util;
    if (util_fact > 1.0) {
        util_fact = util_fact - vdes[p].util;
        return(FALSE);
    }
    else return(TRUE);
}
```

The system call *activate* inserts a task in the ready queue, performing the transition SLEEP–READY. If the task is HARD, its absolute deadline is set equal to the current time plus its period. Then the scheduler is invoked to select the task with the earliest deadline.

```
/*-----------------------------------------------------------*/
/* activate -- inserts a task in the ready queue             */
/*-----------------------------------------------------------*/
int     activate(proc p)
{
    save_context();
    if (vdes[p].type == HARD)
        vdes[p].dline = sys_clock + (long)vdes[p].period;
    vdes[p].state = READY;
    insert(p, &ready);
    schedule();
    load_context();
}
```

The transition RUN–SLEEP is performed by the *sleep* system call. The running task is suspended in the sleep state, and the first ready task is dispatched for execution. Notice that this primitive acts on the calling task, which can be periodic or aperiodic. For example, the *sleep* primitive can be used at the end of a cycle to terminate an aperiodic instance.

```
/*-----------------------------------------------------------*/
/* sleep -- suspends itself in a sleep state                 */
/*-----------------------------------------------------------*/
void    sleep(void)
{
    save_context();
    vdes[p].state = SLEEP;
    dispatch();
    load_context();
}
```

The primitive for terminating a periodic instance is a bit more complex than its aperiodic counterpart, since the kernel has to be informed on the time at which the timer has to resume the job. This operation is performed by the primitive *end_cycle*, which puts the running task into the idle queue. Since it is assumed that deadlines are at the end of the periods, the next activation time of any idle periodic instance coincides with its current absolute deadline.

In the particular case in which a periodic job finishes exactly at the end of its period, the job is inserted not in the idle queue but directly in the ready queue, and its deadline is set to the end of the next period.

```
/*-----------------------------------------------------------*/
/* end_cycle -- inserts a task in the idle queue             */
/*-----------------------------------------------------------*/
void     end_cycle(void)
{
long    dl;
    save_context();
    dl = vdes[pexe].dline;
    if (sys_clock < dl) {
        vdes[pexe].state = IDLE;
        insert(pexe, &idle);
    }
    else {
        dl = dl + (long)vdes[pexe].period;
        vdes[p].dline = dl;
        vdes[p].state = READY;
        insert(pexe, &ready);
    }
    dispatch();
    load_context();
}
```

A typical example of periodic task is shown in the following code:

```
proc    cycle()
{
    while (TRUE) {
        <periodic code>
        end_cycle();
    }
}
```

There are two primitives for terminating a process: the first, called *end_process*, directly operates on the calling task; the other one, called *kill*, terminates the task passed as a formal parameter. Notice that, if the task is HARD, it is not immediately removed from the system but put in ZOMBIE state. In this case, the complete removal will be done by the timer routine at the end of the current period:

```
/*-----------------------------------------------------------*/
/* end_process -- terminates the running task                */
/*-----------------------------------------------------------*/
void    end_process(void)
{
    <disable cpu interrupts>
    if (vdes[pexe].type == HARD)
        insert(pexe, &zombie);
    else {
        vdes[pexe].state = FREE;
        insert(pexe, &freetcb);
    }
    dispatch();
    load_context();
}
```

```
/*-----------------------------------------------------------*/
/* kill -- terminates a task                                  */
/*-----------------------------------------------------------*/
void     kill(proc p)
{
    <disable cpu interrupts>
    if (pexe == p) {
        end_process();
        return;
    }
    if (vdes[p].state == READY) extract(p, &ready);
    if (vdes[p].state == IDLE)  extract(p, &idle);
    if (vdes[p].type == HARD)
        insert(p, &zombie);
    else {
        vdes[p].state = FREE;
        insert(p, &freetcb);
    }
    <enable cpu interrupts>
}
```

9.5.5 Semaphores

In DICK, synchronization and mutual exclusion are handled by semaphores. Four primitives are provided to the user to allocate a new semaphore (*newsem*), deallocate a semaphore (*delsem*), wait for an event (*wait*), and signal an event (*signal*).

The *newsem* primitive allocates a free semaphore control block and initializes the counter field to the value passed as a parameter. For example, `s1 = newsem(0)` defines a semaphore for synchronization, whereas `s2 = newsem(1)` defines a semaphore for mutual exclusion. The *delsem* primitive just deallocates the semaphore control block, inserting it in the list of free semaphores.

```
/*-----------------------------------------------------------*/
/* newsem -- allocates and initializes a semaphore          */
/*-----------------------------------------------------------*/
sem    newsem(int n)
{
sem   s;
    <disable cpu interrupts>
    s = freesem;              /* first free semaphore index */
    if (s == NIL) abort(NO_SEM);
    freesem = vsem[s].next;     /* update the freesem list */
    vsem[s].count = n;          /* initialize counter      */
    vsem[s].qsem = NIL;         /* initialize sem. queue   */
    <enable cpu interrupts>
    return(s);
}
```

```
/*-----------------------------------------------------------*/
/* delsem -- deallocates a semaphore                        */
/*-----------------------------------------------------------*/
void    delsem(sem s)
{
    <disable cpu interrupts>
    vsem[s].next = freesem;     /* inserts s at the head    */
    freesem = s;                /* of the freesem list      */
    <enable cpu interrupts>
}
```

The *wait* primitive is used by a task to wait for an event associated to a semaphore. If the semaphore counter is positive, it is decremented, and the task continues its execution; if the counter is less than or equal to zero, the task is blocked, and it is inserted in the semaphore queue. In this case, the first ready task is assigned to the processor by the *dispatch* primitive.

To ensure the consistency of the kernel data structures, all semaphore system calls are executed with cpu interrupts disabled. Notice that semaphore queues are ordered by decreasing absolute deadlines, so that, when more tasks are blocked, the first task awakened will be the one with the earliest deadline.

```
/*-----------------------------------------------------------*/
/* wait -- waits for an event                                */
/*-----------------------------------------------------------*/
void    wait(sem s)
{
    <disable cpu interrupts>
    if (vsem[s].count > 0) vsem[s].count --;
    else {
        save_context();
        vdes[pexe].state = WAIT;
        insert(pexe, &vsem[s].qsem);
        dispatch();
        load_context();
    }
    <enable cpu interrupts>
}
```

The *signal* primitive is used by a task to signal an event associated with a semaphore. If no tasks are blocked on that semaphore (that is, if the semaphore queue is empty), the counter is incremented, and the task continues its execution. If there are blocked tasks, the task with the earliest deadline is extracted from the semaphore queue and is inserted in the ready queue. Since a task has been awakened, a context switch may occur; hence, the context of the running task is saved, a task is selected by the scheduler, and a new context is loaded.

```
/*-----------------------------------------------------------*/
/* signal -- signals an event                                */
/*-----------------------------------------------------------*/
void    signal(sem s)
{
proc    p;
    <disable cpu interrupts>
    if (!empty(vsem[s].qsem)) {
        p = getfirst(&vsem[s].qsem);
        vdes[p].state = READY;
        insert(p, &ready);
        save_context();
        schedule();
        load_context();
    }
    else    vsem[s].count++;
    <enable cpu interrupts>
}
```

It is worth observing that classical semaphores are prone to the priority inversion phenomenon, which introduces unbounded delays during tasks' execution and prevents any form of guarantee on hard tasks (this problem is discussed in Chapter 7). As a consequence, this type of semaphores should be used only by non-real-time tasks, for which no guarantee is performed. Real-time tasks, instead, should rely on more predictable mechanisms, based on time-bounded resource access protocols (such as Stack Resource Policy) or on asynchronous communication buffers. In DICK, the communication among hard tasks occurs through an asynchronous buffering mechanism, which is described in Section 9.6.

9.5.6 Status inquiry

DICK also provides some primitives for inquiring the kernel about internal variables and task parameters. For example, the following primitives allow to get the system time, the state, the deadline, and the period of a desired task.

```
/*------------------------------------------------------------*/
/* get_time -- returns the system time in milliseconds        */
/*------------------------------------------------------------*/
float     get_time(void)
{
    return(time_unit * sys_clock);
}
```

```
/*------------------------------------------------------------*/
/* get_state -- returns the state of a task                   */
/*------------------------------------------------------------*/
int       get_state(proc p)
{
    return(vdes[p].state);
}
```

```
/*------------------------------------------------------------*/
/* get_dline -- returns the deadline of a task                */
/*------------------------------------------------------------*/
long      get_dline(proc p)
{
    return(vdes[p].dline);
}
```

```
/*------------------------------------------------------------*/
/* get_period -- returns the period of a task                 */
/*------------------------------------------------------------*/
float     get_period(proc p)
{
    return(vdes[p].period);
}
```

9.6 INTERTASK COMMUNICATION MECHANISMS

Intertask communication is a critical issue in real-time systems, even in a uniprocessor environment. In fact, the use of shared resources for implementing message passing schemes may cause priority inversion and unbounded blocking on tasks' execution. This would prevent any guarantee on the task set and would lead to a highly unpredictable timing behavior.

In this section, we discuss problems and solutions related to the most typical communication semantics used in operating systems: the synchronous and the asynchronous model.

In the pure synchronous communication model, whenever two tasks want to communicate they must be synchronized for a message transfer to take place. This synchronization is called a *rendez-vous.* Thus, if the sender starts first, it must wait until the recipient receives the message; on the other hand, if the recipient starts first, it must wait until the sender produces its message.

In a dynamic real-time system, synchronous communication schemes easily lead to unpredictable behavior, due to the difficulty of estimating the maximum blocking time for a process *rendez-vous.* In a static real-time environment, the problem can be solved off-line by transforming all synchronous interactions into precedence constraints. According to this approach, each task is decomposed into a number of subtasks that contain communication primitives not inside their code but only at their boundary. In particular, each subtask can receive messages only at the beginning of its execution and can send messages only at the end. Then a precedence relation is imposed between all adjacent subtasks deriving from the same father task and between all subtasks communicating through a send-receive pair. An example of such a task decomposition is illustrated in Figure 9.15.

In a pure asynchronous scheme, communicating tasks do not have to wait for each other. The sender just deposits its message into a channel and continues its execution, independently of the recipient condition. Similarly, assuming that at least a message has been deposited into the channel, the receiver can directly access the message without synchronizing with the sender.

Asynchronous communication schemes are more suitable for dynamic real-time systems. In fact, if no unbounded delays are introduced during tasks' communication, timing constraints can easily be guaranteed without increasing the

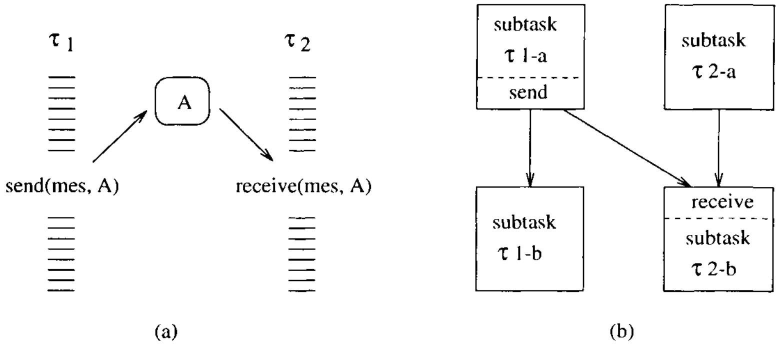

Figure 9.15 Decomposition of communicating tasks (a) into subtasks with precedence constraints (b).

complexity of the system (for example, overconstraining the task set with additional precedence relations). Remember that having simple on-line guarantee tests (that is, with polynomial time complexity) is crucial for dynamic systems.

In most commercial real-time operating systems, the asynchronous communication scheme is implemented through a *mailbox* mechanism, illustrated in Figure 9.16. A mailbox is a shared memory buffer capable of containing a fixed number of messages that are typically kept in a FIFO queue. The maximum number of messages that at any instant can be held in a mailbox represents its *capacity*.

Two basic operations are provided on a mailbox – namely, *send* and *receive*. A *send(MX, mes)* operation causes the message *mes* to be inserted in the queue of mailbox MX. If at least a message is contained on mailbox MX, a *receive(MX, mes)* operation extracts the first message from its queue. Notice that, if the kernel provides the necessary support, more than two tasks can share a mailbox, and channels with multiple senders and/or multiple receivers can be realized. As long as it is guaranteed that a mailbox is never empty and never full, sender(s) and receiver(s) are never blocked.

Unfortunately, a mailbox provides only a partial solution to the problem of asynchronous communication, since it has a bounded capacity. Unless sender and receiver have particular arrival patterns, it is not possible to guarantee that the mailbox queue is never empty or never full. If the queue is full, the sender must be delayed until some message is received. If the queue is empty, the receiver must wait until some message is inserted.

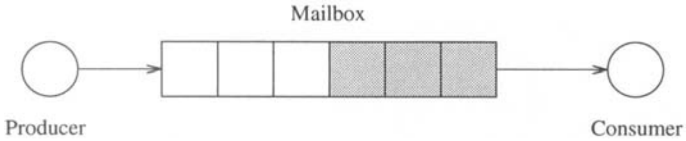

Figure 9.16 The mailbox scheme.

For example, consider two periodic tasks, τ_1 and τ_2, with periods T_1 and T_2, that exchange messages through a mailbox having a capacity of n. Let τ_1 be the sender and τ_2 the receiver. If $T_1 < T_2$, the sender inserts in the mailbox more messages than the receiver can extract; thus, after a while the queue becomes full and the sender must be delayed. From this time on, the sender has to wait the receiver, so it synchronizes with its period (T_2). Viceversa, if $T_1 > T_2$, the receiver reads faster than the sender can write; thus, after a while the queue becomes empty and the receiver must wait. From this time on, the receiver synchronizes with the period of the sender (T_1). In conclusion, if $T_1 \neq T_2$, sooner or later both tasks will run at the lowest rate, and the task with the shortest period will miss its deadline.

An alternative approach to asynchronous communication is provided by acyclical asynchronous buffers, which are described in the next section.

9.6.1 Cyclical asynchronous buffers

Cyclical Asynchronous Buffers, or CABs, represent a particular mechanism purposely designed for the cooperation among periodic activities, such as control loops and sensory acquisition tasks. This approach was first proposed by Clark [Cla89] for implementing a robotic application based on hierarchical servo-loops, and it is used in the HARTIK system [But93, BD93] as a basic communication support among periodic hard tasks.

A CAB provides a one-to-many communication channel, which at any instant contains the latest message or data inserted in it. A message is not consumed (that is, extracted) by a receiving process but is maintained into the CAB structure until a new message is overwritten. As a consequence, once the first message has been put in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.

Notice that, using such a semantics, a message can be read more than once if the receiver is faster than the sender, while messages can be lost if the sender is faster than the receiver. However, this is not a problem in many control applications, where tasks are interested only in fresh sensory data rather than in the complete message history produced by a sensory acquisition task.

CABs can be created and initialized by the *open_cab* primitive, which requires specifying the CAB name, the dimension of the message, and the number of messages that the CAB may contain simultaneously. The *delete_cab* primitive removes a CAB from the system and releases the memory space used by the buffers.

To insert a message in a CAB, a task must first reserve a buffer from the CAB memory space, then copy the message into the buffer, and finally put the buffer into the CAB structure, where it becomes the most recent message. This is done according to the following scheme:

```
buf_pointer = reserve(cab_id);
<copy message in *buf_pointer>
 putmes(buf_pointer, cab_id);
```

Similarly, to get a message from a CAB, a task has to get the pointer to the most recent message, use the data, and release the pointer. This is done according to the following scheme:

```
mes_pointer = getmes(cab_id);
<use message>
 unget(mes_pointer, cab_id);
```

Notice that more tasks can simultaneously access the same buffer in a CAB for reading. On the other hand, if a task P reserves a CAB for writing while another task Q is using that CAB, a new buffer is created, so that P can write its message without interfering with Q. As P finishes writing, its message becomes the most recent one in that CAB. The maximum number of buffers that can be created in a CAB is specified as a parameter in the *open_cab* primitive. To avoid blocking, this number must be equal to the number of tasks that use the CAB plus one.

9.6.2 CAB implementation

The data structure used to implement a CAB is shown in Figure 9.17. A CAB control block must store the maximum number of buffers (*max_buf*), their dimension (*dim_buf*), a pointer to a list of free buffers (*free*), and a pointer to the most recent buffer (*mrb*). Each buffer in the CAB can be implemented as a data structure with three fields: a pointer (*next*) to maintain a list of free buffers, a counter (*use*) that stores the current number of tasks accessing that buffer, and a memory area (*data*) for storing the message.

The code of the four CAB primitives is shown below. Notice that the main purpose of the *putmes* primitive is to update the pointer to the most recent buffer (MRB). Before doing that, however, it deallocates the old MRB if no tasks are accessing that buffer. Similarly, the *unget* primitive decrements the number of tasks accessing that buffer and deallocates the buffer only if no task is accessing it and it is not the MRB.

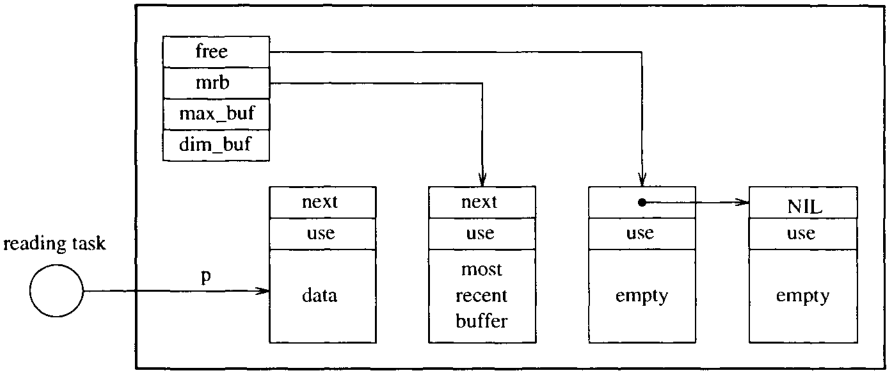

Figure 9.17 CAB data structure.

```
/*-----------------------------------------------------------*/
/* reserve -- reserves a buffer in a CAB                     */
/*-----------------------------------------------------------*/
pointer     reserve(cab c)
{
pointer    p;
    <disable cpu interrupts>
    p = c.free;                    /* get a free buffer      */
    c.free = p.next;               /* update the free list   */
    return(p);
    <enable cpu interrupts>
}
```

```
/*-----------------------------------------------------------*/
/* putmes -- puts a message in a CAB                         */
/*-----------------------------------------------------------*/
void     putmes(cab c, pointer p)
{
    <disable cpu interrupts>
    if (c.mrb.use == 0) {            /* if not accessed,     */
        c.mrb.next = c.free;         /* deallocate the mrb   */
        c.free = c.mrb;
    }
    c.mrb = p;                       /* update the mrb       */
    <enable cpu interrupts>
}
```

```
/*-----------------------------------------------------------*/
/* getmes -- gets a pointer to the most recent buffer        */
/*-----------------------------------------------------------*/
pointer     getmes(cab c)
{
pointer    p;
    <disable cpu interrupts>
    p = c.mrb;                     /* get the pointer to mrb */
    p.use = p.use + 1;             /* increment the counter  */
    return(p);
    <enable cpu interrupts>
}
```

```
/*-----------------------------------------------------------*/
/* unget -- deallocates a buffer only if it is not accessed  */
/*          and it is not the most recent buffer             */
/*-----------------------------------------------------------*/
void     unget(cab c, pointer p)
{
    <disable cpu interrupts>
    p.use = p.use - 1;
    if ((p.use == 0) && (p != c.mrb)) {
        p.next = c.free;
        c.free = p;
    }
    <enable cpu interrupts>
}
```

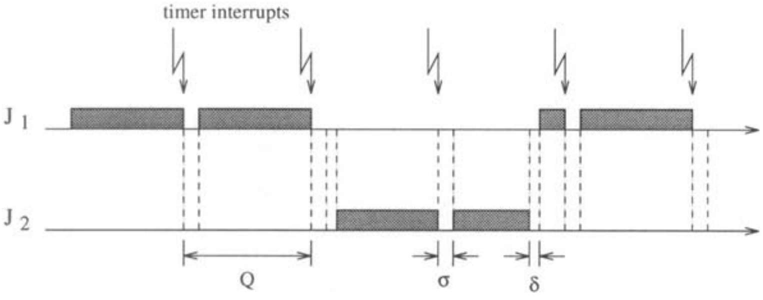

Figure 9.18 Effects of the overhead on tasks' execution.

9.7 SYSTEM OVERHEAD

The overhead of an operating system represents the time used by the processor for handling all kernel mechanisms, such as enqueueing tasks, performing context switches, updating the internal data structures, sending messages to communication channels, servicing the interrupt requests, and so on. The time required to perform these operations is usually much smaller than the execution times of the application tasks; hence, it can be neglected in the schedulability analysis and in the resulting guarantee test. In some cases, however, when application tasks have small execution times and tight timing constraints, the activities performed by the kernel may not be so negligible and may create a significant interference on tasks' execution. In these situations, predictability can be achieved only by considering the effects of the runtime overhead in the schedulability analysis.

The context switch time is one of the most significant overhead factors in any operating system. It is an intrinsic limit of the kernel that does not depend on the specific scheduling algorithm, nor on the structure of the application tasks. For a real-time system, another important overhead factor is the time needed by the processor to execute the timer interrupt handling routine. If Q is the system tick (that is, the period of the interrupt requests from the timer) and σ is the worst-case execution time of the corresponding driver, the timer overhead can be computed as the utilization factor U_t of an equivalent periodic task:

$$U_t = \frac{\sigma}{Q}.$$

Figure 9.18 illustrates the execution intervals (σ) due to the timer routine and the execution intervals (δ) necessary for a context switch. The effects of the timer routine on the schedulability of a periodic task set can be taken into

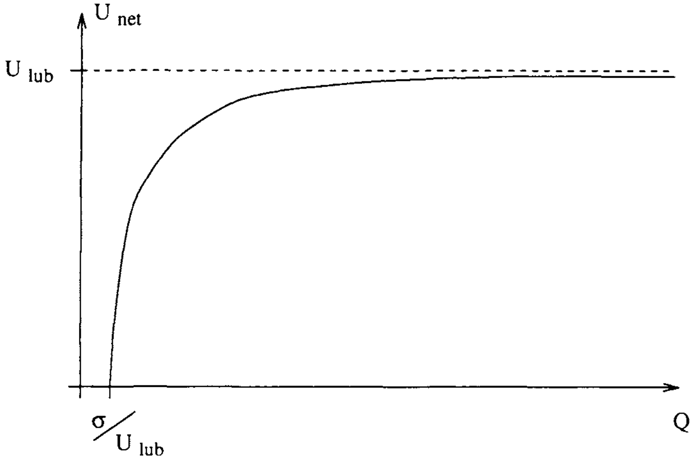

Figure 9.19 Net utilization bound as a function of the tick value.

account by adding the factor U_t to the total utilization of the task set. This is the same as reducing the least upper bound of the utilization factor U_{lub} by U_t, so that the net bound becomes

$$U_{net} = U_{lub} - U_t = U_{lub} - \frac{\sigma}{Q} = U_{lub}\left(\frac{Q - \sigma/U_{lub}}{Q}\right).$$

From this result we can notice that, to have $U_{net} > 0$, the system tick Q must always be greater than (σ/U_{lub}). The plot of U_{net} as a function of Q is illustrated in Figure 9.19. To have an idea of the degradation caused by the timer overhead, consider a system based on the EDF algorithm ($U_{lub} = 1$) and suppose that the timer interrupt handling routine has an execution time of $\sigma = 100\mu s$. In this system, a 10 ms tick would cause a net utilization bound $U_{net} = 0.99$; a 1 ms tick would decrease the net utilization bound to $U_{net} = 0.9$; whereas a $200\mu s$ tick would degrade the net bound to $U_{net} = 0.5$. This means that, if the greatest common divisor among the task periods is $200\mu s$, a task set with utilization factor $U = 0.6$ cannot be guaranteed under this system.

The overhead due to other kernel mechanisms can be taken into account as an additional term on tasks' execution times. In particular, the time needed for explicit context switches (that is, the ones triggered by system calls) can be considered in the execution time of the kernel primitives; thus, it will be charged to the worst-case execution time of the calling task. Similarly, the overhead associated with implicit context switches (that is, the ones triggered by the kernel) can be charged to the preempted tasks.

In this case, the schedulability analysis requires a correct estimation of the total number of preemptions that each task may experience. In general, for a given scheduling algorithm, this number can be estimated off-line as a function of tasks' timing constraints. If N_i is the maximum number of preemptions that a periodic task τ_i may experience in each period, and δ is the time needed to perform a context switch, the total utilization factor (overhead included) of a periodic task set can be computed as

$$U_{tot} = \sum_{i=1}^{n} \frac{C_i + \delta N_i}{T_i} + U_t = \sum_{i=1}^{n} \frac{C_i}{T_i} + \left(\delta \sum_{i=1}^{n} \frac{N_i}{T_i} + U_t \right).$$

Hence, we can write

$$U_{tot} = U_p + U_{ov},$$

where U_p is the utilization factor of the periodic task set and U_{ov} is a correction factor that considers the effects of the timer handling routine and the preemption overhead due to intrinsic context switches (explicit context switches are already considered in the C_i's terms):

$$U_{ov} = U_t + \delta \sum_{i=1}^{n} \frac{N_i}{T_i}.$$

Finally, notice that an upper bound for the number of preemptions N_i on a task τ_i can be computed as

$$N_i = \sum_{k=1}^{i-1} \left\lfloor \frac{T_i}{T_k} \right\rfloor.$$

However, this bound is too pessimistic, and better bounds can be found for particular scheduling algorithms.

9.7.1 Accounting for interrupt

Two basic approaches can be used to handle interrupts coming from external devices. One method consists of associating an aperiodic or sporadic task to each source of interrupt. This task is responsible for handling the device and is subject to the scheduling algorithm as any other task in the system. With this method, the cost for handling the interrupt is automatically taken in to account by the guarantee mechanism, but the task may not start immediately, due to the presence of higher-priority hard tasks. This method cannot be used for those devices that require immediate service for avoiding data loss.

Another approach allows interrupt handling routines to preempt the current task and execute immediately at the highest priority. This method minimizes the interrupt latency, but the interrupt handling cost has to be explicitly considered in the guarantee of the hard tasks.

Jeffay and Stone [JS93] found a schedulability condition for a set of n hard tasks and m interrupt handlers. In their work, the analysis is carried out by assuming a discrete time, with a resolution equal to a tick. As a consequence, every event in the system occurs at a time that is multiple of the tick. In their model, there is a set $\mathcal{I}$ of m handlers, characterized by a worst-case execution time C_i^H and a minimum separation time T_i^H, just as sporadic tasks. The difference is that interrupt handlers always have a priority higher than the application tasks.

The upper bound, $f(l)$, for the interrupt handling cost in any time interval of length l can be computed by the following recurrent relation [JS93]:

$$\begin{aligned} f(0) &= 0 \\ f(l) &= \begin{cases} f(l-1)+1 & \text{if } \sum_{i=1}^{m} \left\lceil \frac{l}{T_i^H} \right\rceil C_i^H > f(l-1) \\ f(l-1) & \text{otherwise.} \end{cases} \end{aligned} \tag{9.1}$$

In the particular case in which all the interrupt handlers start at time $t = 0$, function $f(l)$ is exactly equal to the amount of time spent by processor in executing interrupt handlers in the interval $[0, l]$.

Theorem 9.1 (Jeffay-Stone) *A set $\mathcal{T}$ of n periodic or sporadic tasks and a set $\mathcal{I}$ of m interrupt handlers is schedulable by EDF if and only if for all L, $L \geq 0$,*

$$\sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i \;\leq\; L - f(L). \tag{9.2}$$

The proof of Theorem 9.1 is very similar to the one presented for Theorem 4.2. The only difference is that, in any interval of length L, the amount of time that the processor can dedicate to the execution of application tasks is equal to $L - f(L)$.

It is worth to notice that equation (9.2) can be checked only for a set of points equal to release times less than the hyperperiod, and the complexity of the computation is pseudo-polynomial.