Didactic C Kernel: Chapter 10 of *Hard Real-Time Computing Systems By Giorgio C. Buttazzo*

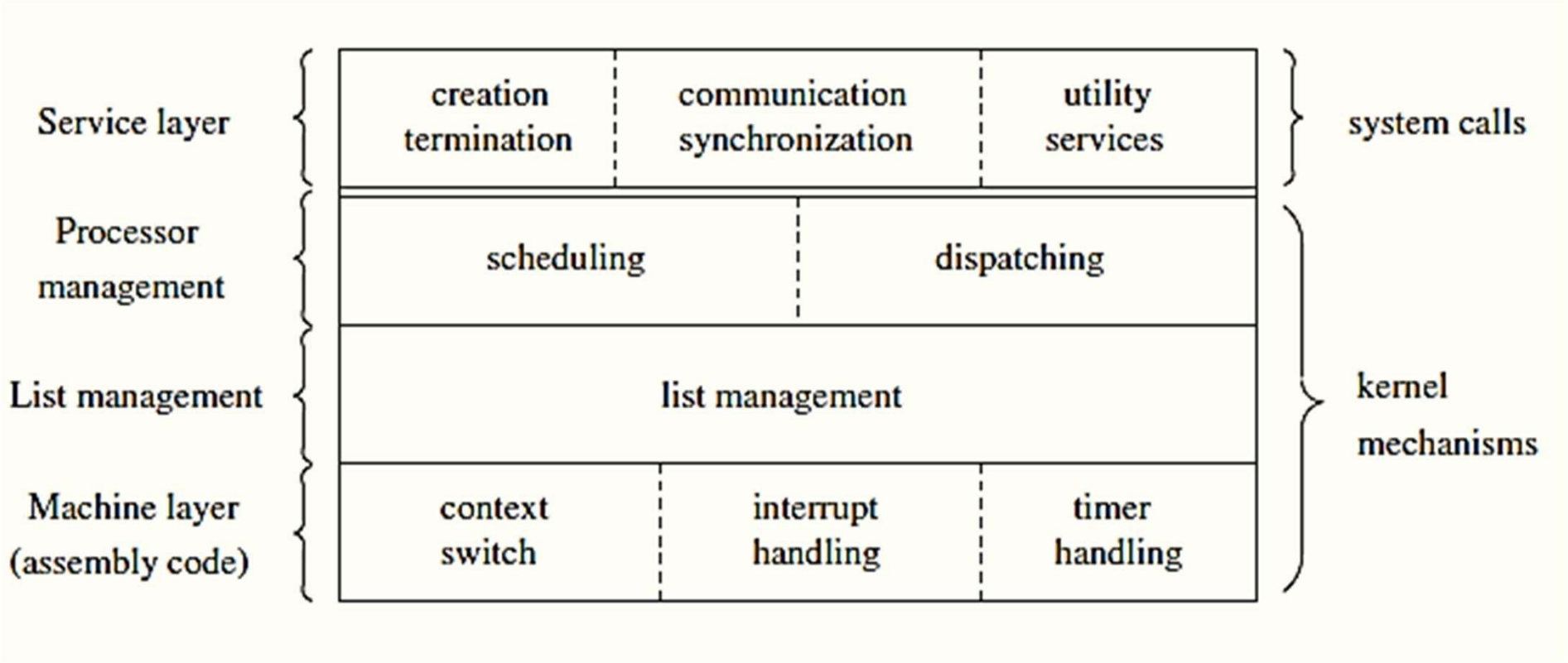
This chapter presents a hard-real-time kernel for critical control applications and covers:

- Basic issues to be considered during the design and the development of such kernel
- The structure and the main components for this small real-time kernel, called DICK (DIdactic C Kernel)
- The problem of time predictable inter-task communication
- A particular communication mechanism for exchanging state messages among periodic tasks is illustrated
- Evaluating the runtime overhead of the kernel for schedulability analysis

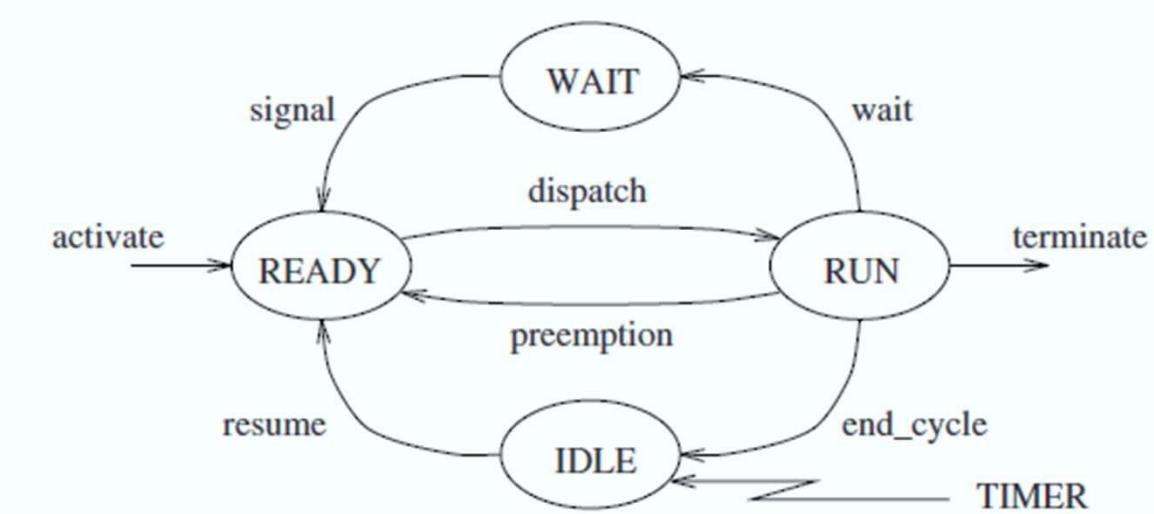
Structure of a Real-Time Kernel

- A kernel usually provides the following basic activities:
  - Process management – includes various supporting functions
    - Process creation and termination, job scheduling, dispatching, context switching etc.
  - Interrupt handling service – provides service to interrupt requests
    - The service consists in the execution of a dedicated routine (driver) that will transfer data from the device to the main memory (or vice versa).
    - In a real-time system, the interrupt handling mechanism must be integrated with the scheduling mechanism
  - Process synchronization and communication – basic mechanism
    - In order to achieve predictability, a real-time kernel must 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

The Structure of DIdactic C Kernel



Process States, Analysis



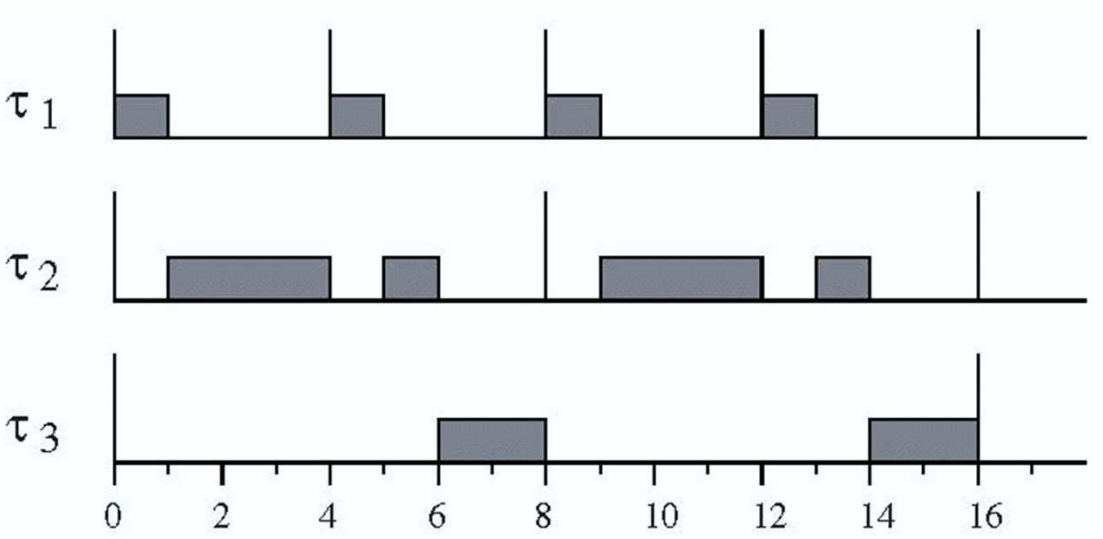
A real-time OS where semaphores are used for mutual exclusion and synchronization, we must have at least 3 main states:
RUN: …
READY: …
WAIT: …

- In addition, a real-time kernel that supports the execution of periodic tasks, another state must be considered, the IDLE state
- IDLE: …
- Additional states can be introduced by other kernel services. Example are:
  - DELAY …
  - RECEIVE …
  - ZOMBIE …

Argument for a New State for System that Support Dynamic Creation and Termination of Hard Periodic Tasks

Task scenario problem: set of 3 periodic tasks is scheduled using RM algorithm

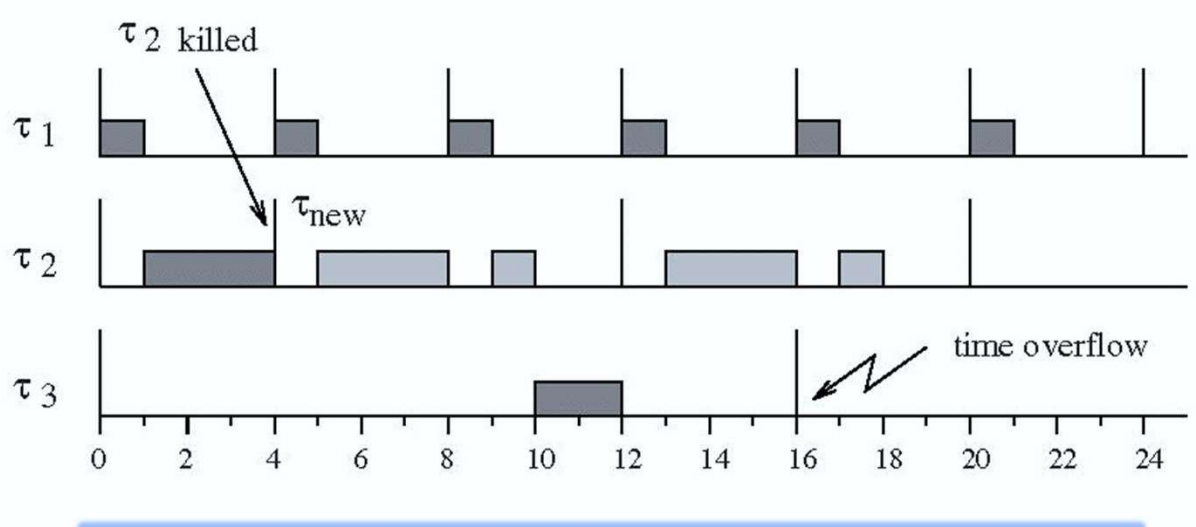
- Execution times are:
  - τ1 = 1; τ2 = 4; τ3 = 4;
- Task periods are: 4, 8, 16
  - Based on RM the priorities relation is: τ1 > τ2 > τ3



The Need of an Extra State

Condition problem to solve: suppose that τ2 is aborted at t=4 and a new task τ_{new} is created

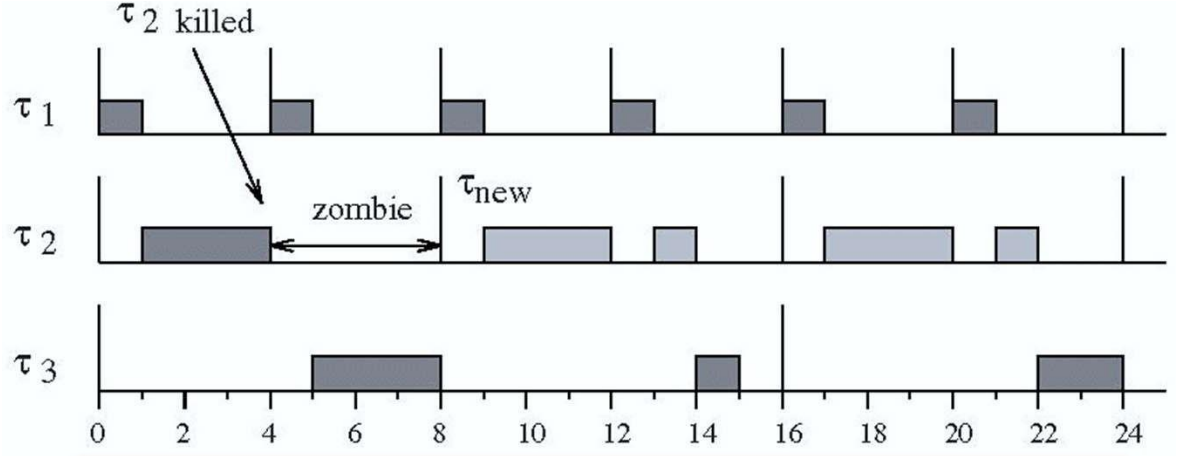
- τ3 would miss its deadline if a new task τ_{new} replaces the old task τ2 at time 4;
- This is due to the effects of τ2 execution on the schedule



The effects of τ2 in the utilization factor do not cancel at the time it is aborted, but prolong until the end of its period.

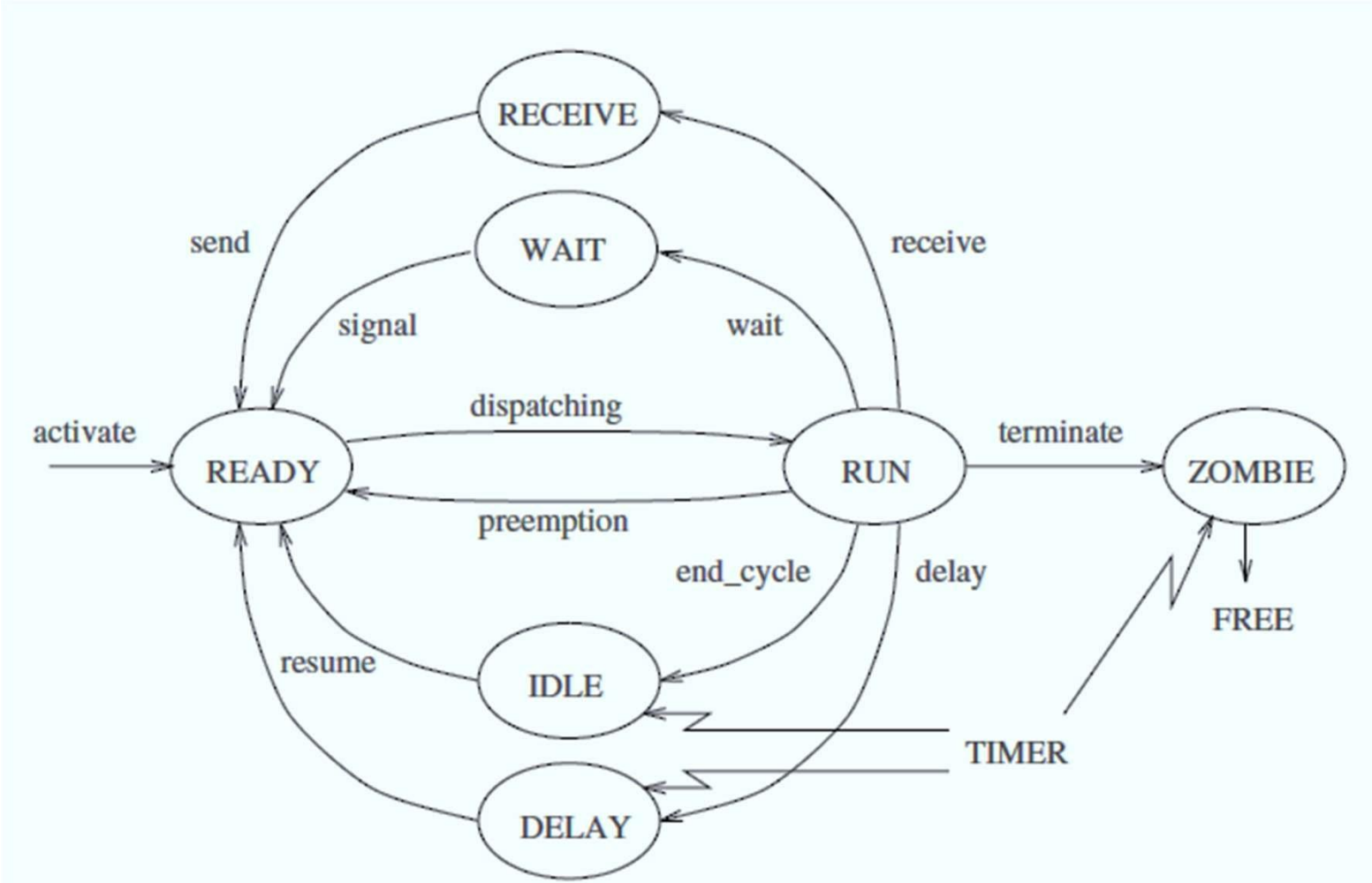
Solution to the State Problem

- The new task set is schedulable when τ_{new} is delayed until the end of the current period of $\tau 2$
- In the interval of time between the abort operation and the end of its period, $\tau 2$ will be placed in a new state to be called ZOMBIE state



In the ZOMBIE stated the task doesn't exist in the system but it continues to occupy processor bandwidth as related to the processor utilization factor.

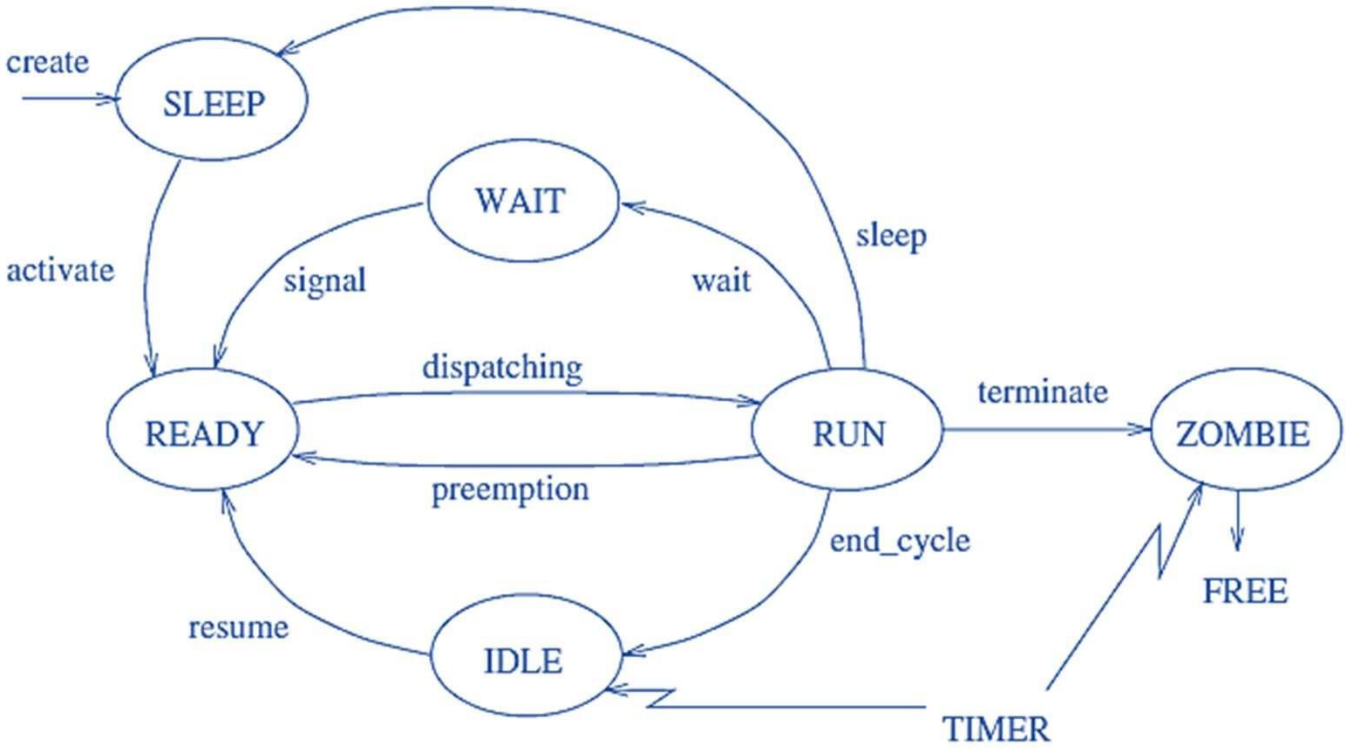
Complete State Transition Diagram for Kernel



Simplified State Transition Diagram in DIdactic C Kernel

The kernel presentation focuses only on the essential functions:

- message passing mechanism and delay primitives are not introduced



The kernel supports activation and suspension of aperiodic tasks by one new state and two primitives:

- SLEEP state; activate and sleep primitives.

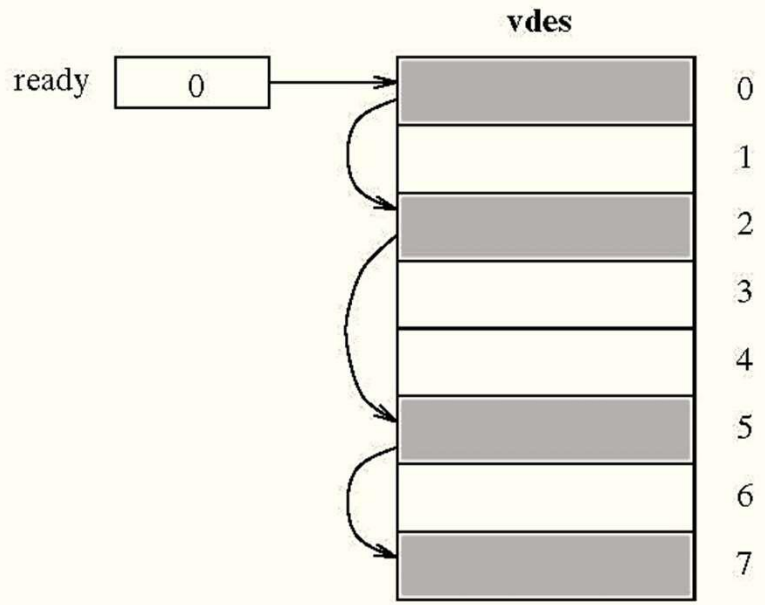
Data Structures

- An OS will store data information about a task in a data structure called the Task Control Block (TCB)
- A TCB contains all the parameters specified by the programmer at creation time + other temporary information used for management
- Typical fields for a real-time kernel are given in the table
  - Note the existence of new fields such as period, computation time, relative deadline, absolute deadline, and utilization factor.
- A TCB must be inserted in the lists handled by the kernel, so we have the *pointer to the next TCB*

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

Queue of Tasks

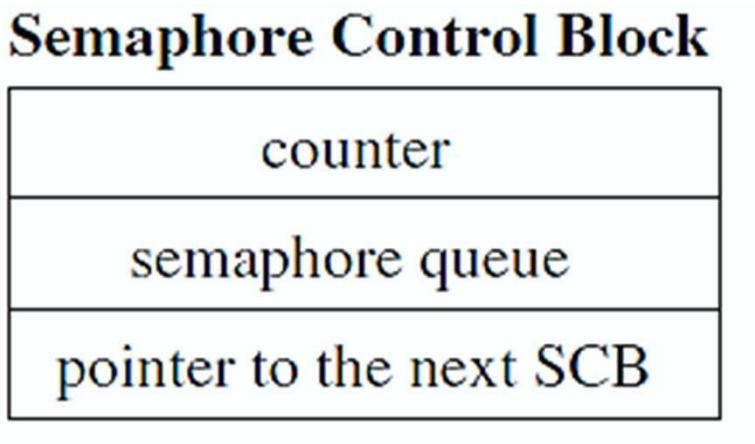
- In the kernel, a TCB is an element of the *vdes[MAXPROC]* array
- Each TCB can be then identified by a *unique index* corresponding to its position in the vdes array.
- As a result, any queue of tasks can be accessed by an integer variable containing the array index of the TCBs in the queue



Example of a ready queue configuration within the vdes array

Semaphore Control Block Data Structure

- The semaphore information is stored in a Semaphore Control Block (SCB) which will contain at least 3 fields as shown
- Each SCB is an element of the *vsem[MAXSEM]* array



Type Definitions

- Based on the array approach, tasks, semaphores, and queues can be accessed by an integer number that represents the index of the corresponding control block.
- There are different types however, that are defined for the integer access variables as shown in the table

```
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    */
```

Data Structures Description, TCB, SCB

```
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   */
};
```

Data Structures Description, TCB, SCB

```
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         */
```

Time Management

The kernel needs a time reference to manage its activity:

- A real-time timer is programmed to interrupt the processor at a fixed rate called a system tick – this is the kernel's time resolution
- The kernel's system time is represented by a long integer variable called sys_clock and the value of the tick by a float variable

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

- So, if *Q* denotes the system tick and *n* is the value sys_clock, the actual time elapsed since system initialization is: $t = n*Q$
- Typical time resolution values can vary from 1 to 50 ms

Tick Values Considerations

- The tick value is application specific
  - small tick values improve system responsiveness and allow handling periodic activities with high activation rates.
  - a very small tick value will cause a large runtime overhead due to the timer handling routine and reduces the system lifetime
- The maximum time (system lifetime) depends on tick value
- To better control task deadlines and periodic activations, all task time parameters specified should be multiple of the system tick
- If the tick value is user defined,
  - Make *tick value = the greatest common divisor of all the task periods*

Timer Interrupt Handling Routine

- At each tick interrupt, the interrupt handling routine must do:
  - Save the context of the task in execution
  - Increment the system time
  - Generate a timing error, if the current time is greater than the system lifetime
  - Generate a time-overflow error, if the current time is greater than some hard deadline
  - Awaken those idle tasks, if any, that must begin a new period
  - Call the scheduler, if at least a task needs be awakened
  - Remove all zombie tasks for which their deadline is expired
  - Load the context of the current task; and
  - Return from interrupt
- The runtime overhead introduced by the execution of the timer routine is proportional to its interrupt rate.

Task Classes

- Real-world control applications usually consist of computational activities having various characteristics:
  - Tasks may be periodic, aperiodic, time-driven, and event-driven, and may have different levels of criticality
- Only two classes of tasks are considered in DIdactic C Kernel
  - 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 of a task is terminated:
  - Terminated with *end_cycle* primitive, the task goes in IDLE state
  - Terminated with *end_aperiodic* primitive the task goes in SLEEP state

Scheduling Algorithm

- HARD tasks are scheduled using the Earliest Deadline First (EDF) algorithm
- NRT tasks are executed in background based on their priority.

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 10.11 and is such that

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

Global Constants

Defined so the description of the source code is clear.
Typically they define the maximum sizes for the main kernel data structures;
Or encode process classes -->

```
#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    */
/*-------------------------------------------------------------*/
```

Global Constants

Or states -->

Or error messages -->

```
/*-------------------------------------------------------------*/
/*                        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 */
```

Kernel Initialization

- The real-time environment is initialized when *ini_system* primitive is executed within a C main program
- This main program becomes a NRT task in which new concurrent tasks can be created

```
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>;
}
```

The most important activities performed by ini_system concern initializing the tick timer, all queues …

Kernel Primitives

- The structure of the kernel is logically divided into several hierarchical layers
  - The lowest layer includes:
    - all interrupt handling drivers, and
    - routines for saving and loading task context
  - Next layers up contain:
    - The functions for list manipulation (insertion, extraction, etc.)
    - Basic mechanisms for task management (dispatching and scheduling)
  - Top layer (user level) contains:
    - All kernel services visible by the user: task creation, activation, suspension, termination, synchronization, and status inquiry

Low-Level Primitives; save_context

Implement:

- The mechanism for saving and loading the context of a task;

Task Context: the content of the processor registers

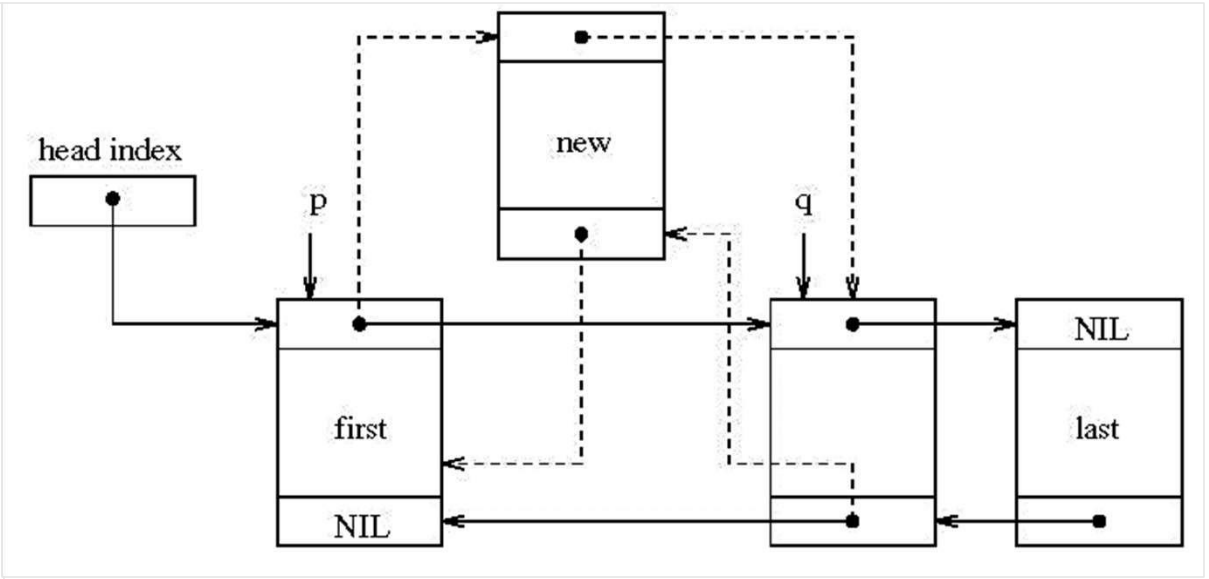
```
/*-------------------------------------------------------*/
/* save_context -- of the task in execution              */
/*-------------------------------------------------------*/
void     save_context(void)
{
int    *pc;                        /* pointer to context of pexe */
    <disable interrupts>
    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 */
}
```

Low-Level Primitives; load_context

```
/*-------------------------------------------------------------*/
/* 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 */
    <enable interrupts>
    <return from interrupt>
}
```

List Management, Insert

- Tasks are scheduled based on EDF and as a result all queues in the kernel are ordered by decreasing deadlines
- The task with the earliest deadline is at the head
- An insertion requires a scan through the list



List Management, Insert Function

The *insert* function is called with two parameters:

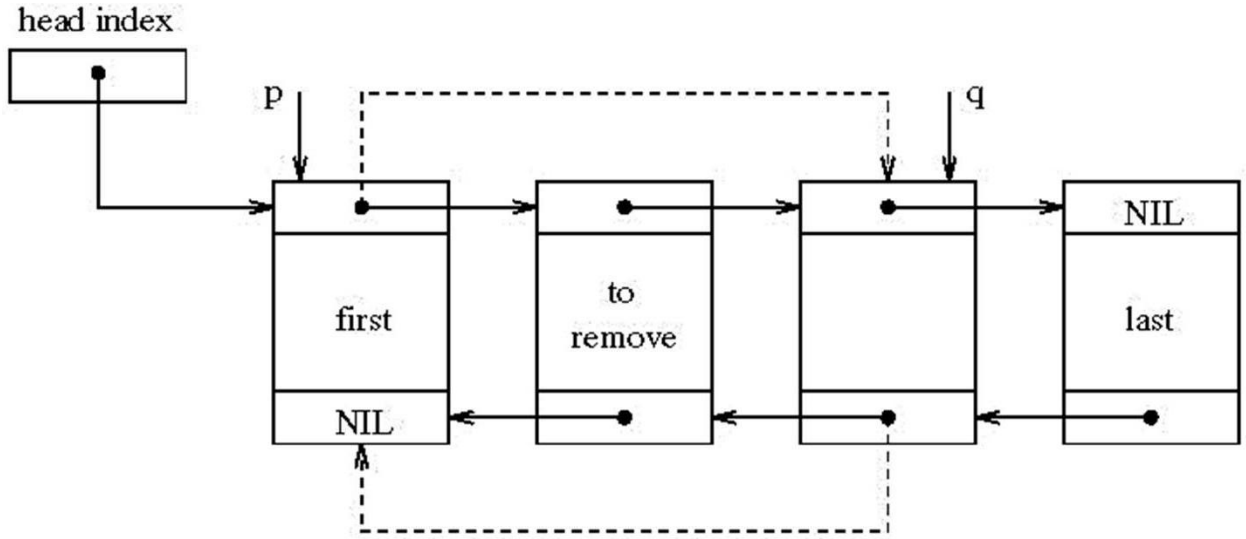
- the index of the task to be inserted and
- the pointer of the queue.

It uses two auxiliary pointers, p and q,

```
/*-------------------------------------------------------------*/
/* 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;
}
```

List Management, Extracting

- By using bidirectional pointers allows the optimization of the extraction operation
- Extraction can be realized in one step without scanning the entire queue



List Management, Extract Function

```
/*-------------------------------------------------------------*/
/* 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);
}
```

List Management, Extracting at the Head of Queue

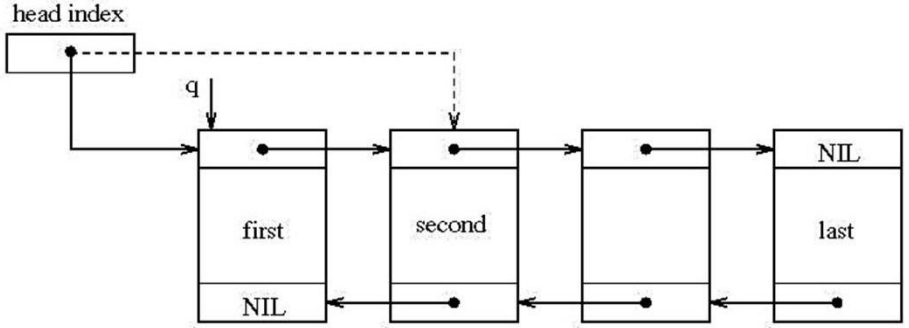


Figure 10.14 Extracting the TCB at the head of a queue.

```
/*-------------------------------------------------------------*/
/* 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);
}
```

List Management, Other Functions

In order to simplify the code reading of the next levels, two more functions are defined:

- *firstdline:* returns the deadline of the task at the head of the Queue;
- *empty:* 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);
}
```

Scheduling Mechanism

- The scheduling is implemented through the functions *schedule* and *dispatch*
- *schedule* primitive verifies whether the running task is the one with the earliest deadline
  - If so, there is no action,
  - 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

wake_up

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.

```
/* 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();
}
```

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* function call 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);
}
```

Task Management, Guarantee

```
/*-------------------------------------------------------*/
/* 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);
}
```

Task Management, Activate

- 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();
}
```

Task Management, Sleep

- The transition RUN–SLEEP is performed by the *sleep* system call.
- This primitive acts on the calling task, which can be periodic or aperiodic
  - The running task -> SLEEP state, and
  - The first ready task is dispatched for execution.
- The *sleep* primitive is 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[pexe].state = SLEEP;
    dispatch();
    load_context();
}
```

Task Management, Terminating a Periodic Task Instance – *end_cycle*

end_cycle

- informs the kernel about the time at which the timer has to resume the job.
- puts the running task into the *idle* queue.

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

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

```
/*------------------------------------------------------------*/
/* 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[pexe].dline = dl;
        vdes[pexe].state = READY;
        insert(pexe, &ready);
    }
     schedule ();
    load_context();
}
```

Task Management, Terminating a Process

There are two primitives for terminating a process:
1) end_process → operates on the calling task;
2) kill → operates on a task passed as a formal parameter.

```
/*-------------------------------------------------------------*/
/* 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 Process Primitive

For HARD tasks the terminating process is:
RUN → ZOMBIE state

- The ZOMBIE → FREE state will be done by *wake_up* timer routine at the end of the current period.

```
/*-------------------------------------------------------------*/
/* 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>
}
```

Semaphores

Used for:

- synchronization and mutual exclusion

Four primitives are implemented for semaphores:

- *newsem*; *delsem*; *wait*; and *signal*

```
/*-------------------------------------------------------*/
/* 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);
}
```

For example, *s1 = newsem(0)* defines a semaphore for synchronization, whereas *s2 = newsem(1)* defines a semaphore for mutual exclusion.

Semaphores, delsem ()

delsem()

- deallocates the semaphore control block, inserting it in the list of free semaphores

```
/*-------------------------------------------------------------*/
/* 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>
}
```

Semaphores, wait ()

wait () primitive

- used by a task to wait for an event associated with a semaphore.
- The state of the task, WAIT or continue is determined by the *vsem[s].count* (the semaphore count value)
- When task is blocked the first ready task *dispatched*

```
/*-----------------------------------------
/* 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>
}
```

Semaphores, signal ()

signal () primitive –

- used by a task to signal an event associated with a semaphore.
- if the semaphore's queue is empty, the counter is incremented, and the task continues its execution.
- If there are blocked tasks
  - the task with the earliest deadline (head of queue) is switched into the READY state
  - Possibly a context switch will take place … so schedule () is called.

```
/* 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>
}
```

Priority Inversion Issue

- 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 – see uC/OS-III presentation
  - This type of semaphores should be used only by non-real-time tasks, for which no guarantee is performed
- Real-time tasks, must support more predictable mechanisms:
  - Stack Resource Policy protocols
  - Asynchronous communication buffers – implemented by DIdactic C Kernel

Status Inquiry, Primitives Support

```
/*-------------------------------------------------
/* 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);
}
```

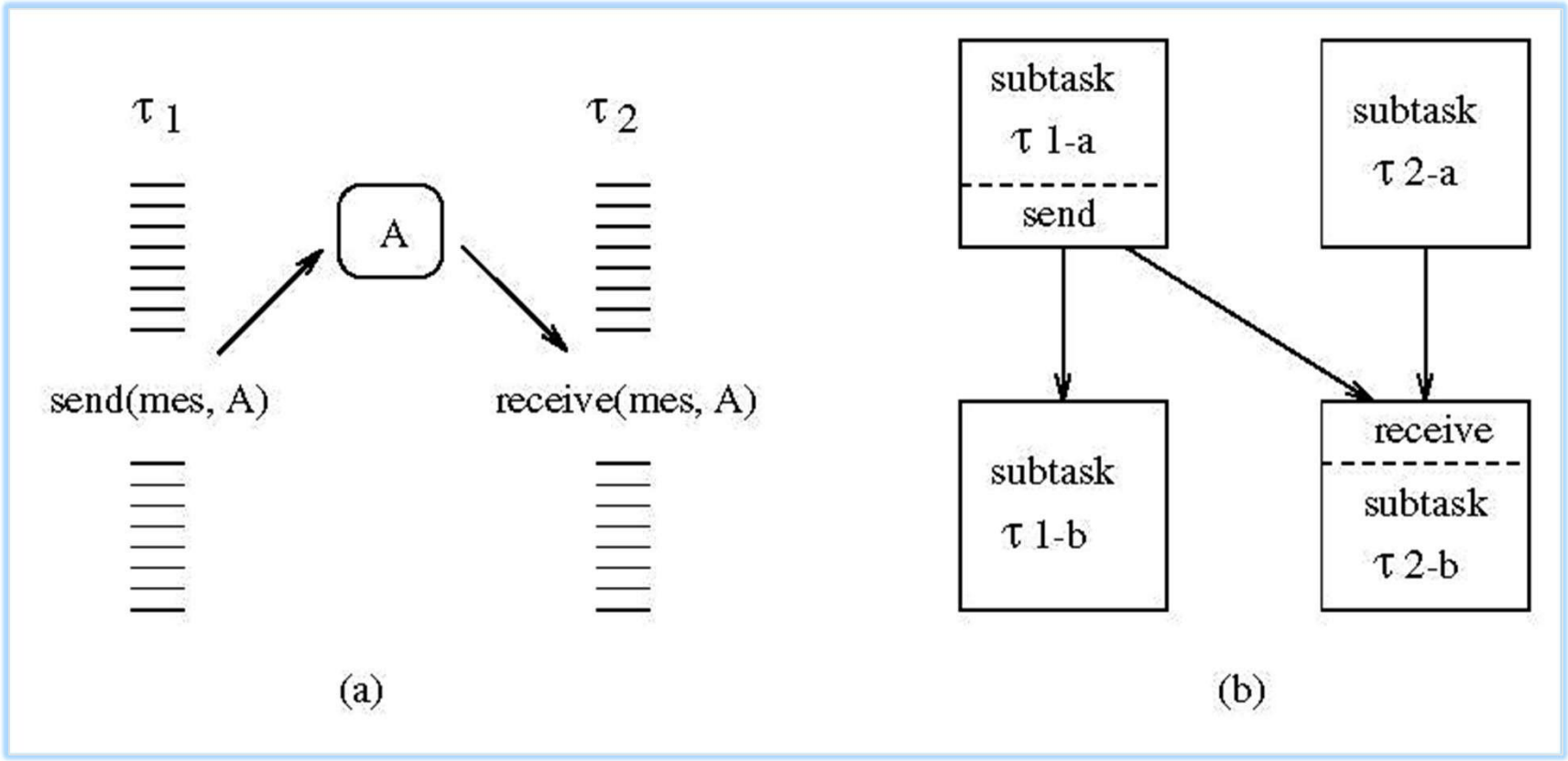
Inter-task Communication Mechanisms

- The use of shared resources for implementing message passing scheme may cause priority inversion and unbounded blocking on task's execution → this can prevent any guarantee on the schedule of the task set
- We will review some problems and solutions related to typical communication semantics in operating systems:
  - The synchronous model
  - The asynchronous model

Synchronous Communication Model

- In a 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*
- In a dynamic real-time system it is difficult to estimate 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
  - Each task is decomposed into a number of sub-tasks that contain communication primitives not inside their code but only at boundary
  - A precedence relation is imposed between adjacent subtasks

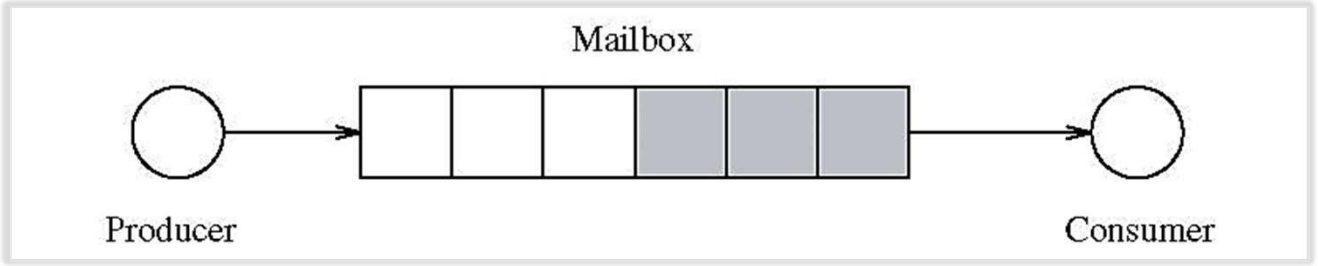
Example of Decomposition of Communication Tasks



Asynchronous Communication Model

- In a pure asynchronous communication scheme, communicating tasks do not have to wait for each other
  - The sender deposits its message into a channel and continue its execution independently of the receiver status
  - Also, the receiver can directly access the message (if exists) without synchronizing with the sender
- Asynchronous communication schemes are better suited for dynamic real-time systems
  - No unbounded delays are introduced during tasks communication
  - Timing constraints can be guaranteed without increasing the complexity of the system

Mailbox Mechanism



- In most commercial real-time operating systems we can use mailbox to implement the asynchronous communication scheme
  - A mailbox is a shared memory buffer that has a capacity; FIFO queue
- Two basic operations are implemented: *send* and *receive*
- As long as it is guaranteed that a mailbox is never empty and never full, sender(s) and receiver(s) are never blocked.
- Due to its fixed capacity, a mailbox provides only a partial solution to the problem of asynchronous communication

Example of Poor Asynchronous Communication using Mailbox Scheme

System: 2 periodic tasks τ1 and τ2 with periods T1 and T2 that each exchange messages through a mailbox having capacity n

Scenario: τ1 → sender; τ2 → receiver

- Case: T1 < T2 → the sender inserts in the mailbox more messages than the receiver can extract, and eventually the sender must be delayed
- Case: T1 > T2 → the receiver reads faster than the sender can send and eventually the receiver must wait.

- Conclusion: If T1 ≠T2, sooner or later both tasks will run at the lowest rate, and the task with the shortest period will miss its deadline

Cyclic Asynchronous Buffers (CABs)

CAB is an alternative solution to the asynchronous communication

Features of CAB:

- provides one-to-many communication channel which at any instant contains the latest message or data inserted in it
- A message is not consumed by a receiving process but is maintained into the CAB structure until a new message is overwritten
  - Once the first message has been put in a CAB, a task can never be blocked during receive operation
  - Since a new message overwrites the old one, a sender can never be blocked
- A message can be read more than once if receiver is faster than sender
- Messages can be lost if sender is faster than receiver
  - However, this is not a problem in many control applications where tasks are interested only in fresh sensory data rather than a complete message history

CAB Operation

- *open_cab():* creates and initializes a CAB
  - Parameters: CAB name; the dimension of the message; and the number of simultaneous messages allowed
- *delete_cab* (): deletes a CAB object

Task operations to insert a message in a CAB:

1. reserve a buffer from the CAB memory space
2. copy the message into the buffer,
3. put the buffer into the CAB structure where it becomes the most recent message:

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

Task operations to get a message from a CAB:

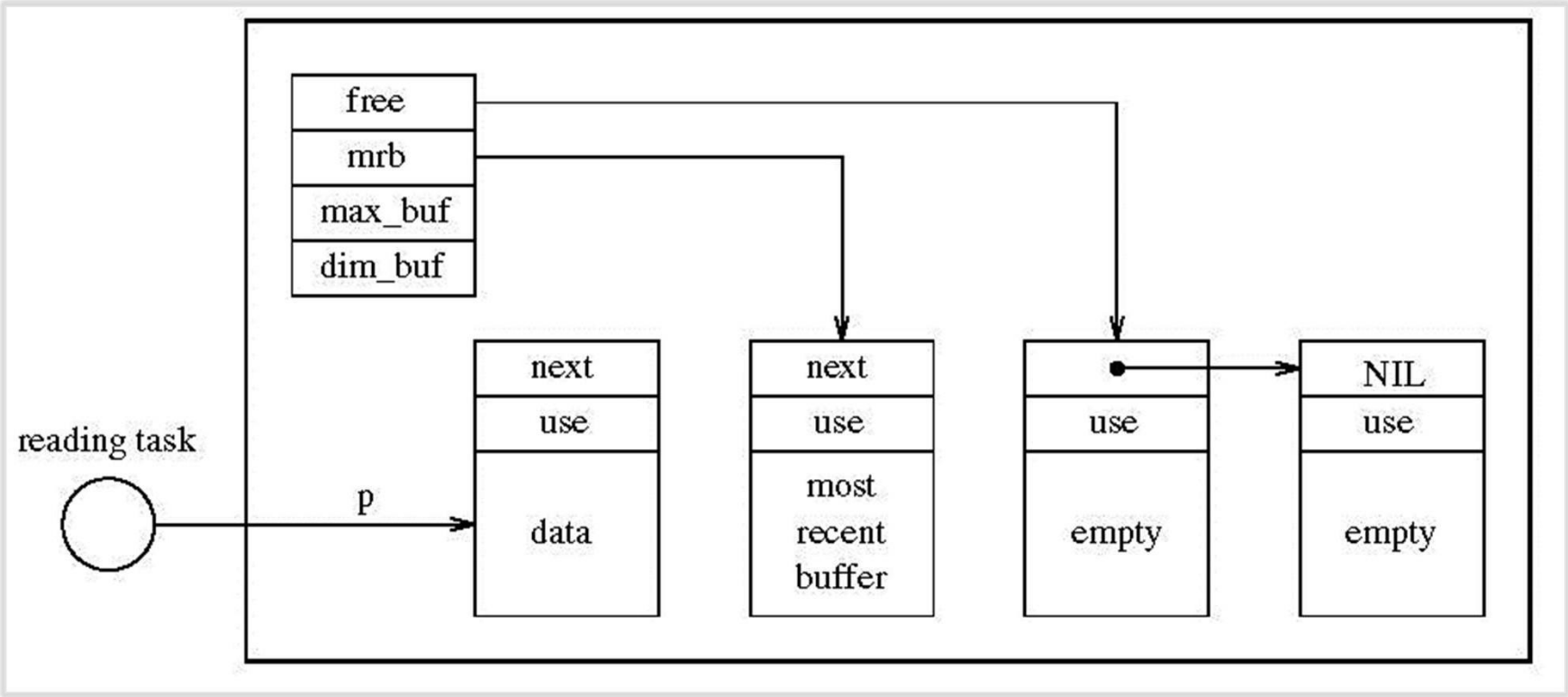
1. Get the pointer to the most recent message
2. Utilize the data,
3. Release the pointer

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

CAB Operation

- More tasks can simultaneously access the same buffer in a CAB for reading.
- 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.

CAB Control Block Data Structure



CAB Primitive *reserve*

```
/*-------------------------------------------------------------*/
/* 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>
}
```

CAB Primitive *putmes*

putmes() primitive will update the pointer to the most recent buffer (MRB).

But, before doing that, however, it deallocates the old MRB if no tasks are accessing that buffer.

```
/* 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>
}
```

CAB Primitives, getmes, unget

getmes() …

unget () …

- decrements the number of tasks accessing that buffer
- deallocates the buffer if no task is accessing it and it is not the MRB

```
/*-------------------------------------------------------------*/
/* 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>
```

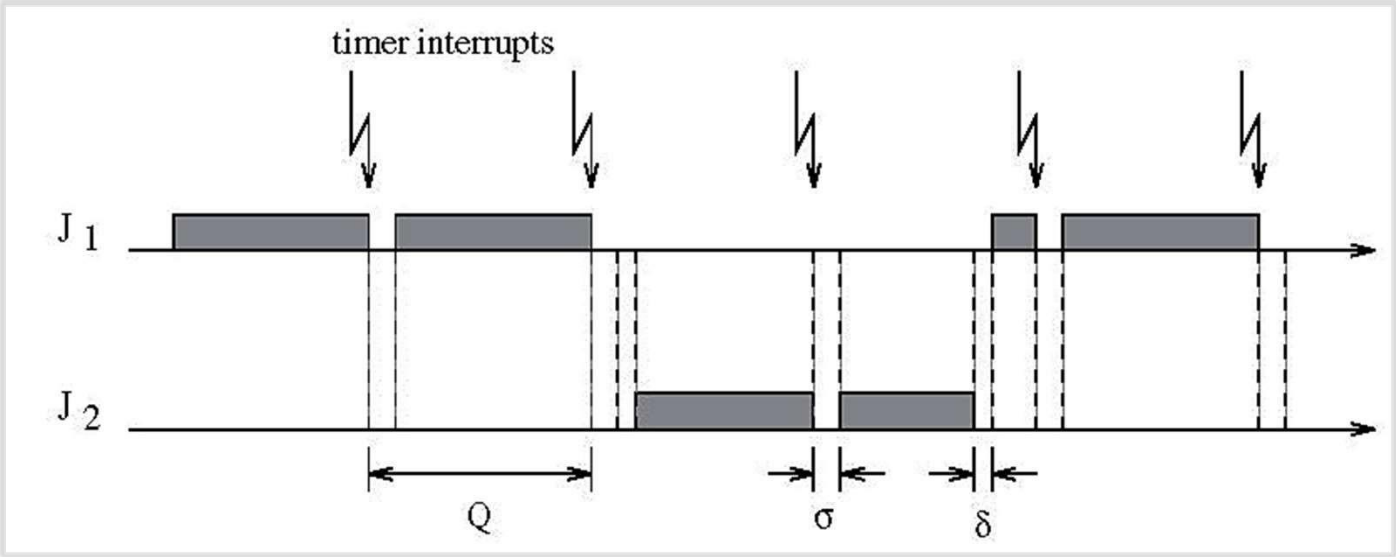
System Overhead

- The overhead of an OS represents the time used by the processor for handling all kernel mechanisms:
  - Operations such as enqueueing tasks, performing context switches, updating the internal data structures, sending messages to communication channels, servicing the interrupt requests, etc.
- In general this overhead 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

Key Overhead Issues

- If the execution time of tasks is small and we deal with tight timing constraints, then overhead must be analyzed
- The most important overhead times introduced by OS are:
  - context switch time
  - time needed by the processor to execute the timer interrupt handling routine

The Effects of the Timer Routine and Context Switch



The execution intervals (σ) due to the timer routine and the execution intervals (δ) necessary for a context switch play a role in the schedulability analysis of the system

Accounting for Timer Interrupt Overhead

Assume:

- Q system tick period
- σ is the worst-case execution time of the timer handler

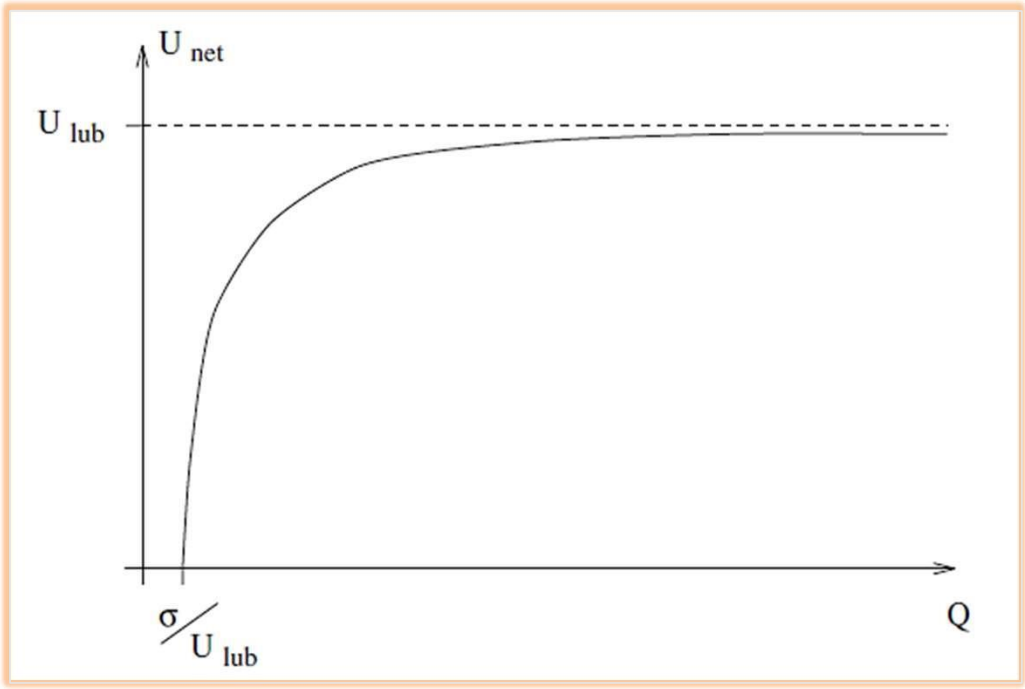
The timer overhead can be computed as the utilization factor:

$$U_t = \sigma / Q$$

- The effects of U_t into the schedulability analysis of the system can be taken into account by:
  - Adding the factor U_t to the total utilization of the task set U
  - Or, reducing the least upper bound of the utilization factor U_{lub} by U_t

U_{net} as a Function of Q

In order to obtain $U_{net} > 0$, the system tick Q must always be greater that (σ/U_{lub}) as it can be derived from the U_{net} relationship:



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

Accounting for Context Switch Overhead

- The context switch time is one of the most significant overhead factors in any operating system
  - 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
  - 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

Accounting for Interrupt

- Two basic approaches can be used to handle overhead related to interrupts coming from external devices

1. Associate 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
2. Allowing the interrupt handling routines to preempt the current task and execute immediately at the highest priority
   - This method minimizes the interrupt latency