Table 3.11 Decision table for inertial measurement system

Criterion	Description	Weight, w_1	A	B	C	D	E
m_1	Minimum interrupt latency	1	0.5	0.8	1	0.5	1
m_2	Number of tasks supported	0.1	0.5	0.5	0.5	1	1
m_3	Memory requirements	1	0.7	0.2	0.5	1	0.9
m_4	Scheduling mechanism	0.5	0.25	0.5	0.25	1	0.25
m_5	Intertask synchronization mechanism	1	0.5	1	0.5	1	1
m_6	Software support (warranty)	1	0.5	0.5	1	0.8	1
m_7	Software support (compiler)	1	1	0.75	1	1	0.5
m_8	Hardware compatibility	0.1	0.8	0.5	0.2	1	0.2
m_9	Royalty free		0	1	1	1	1
m_{10}	Source available	1	1	1	0	0.4	1
m_{11}	Context switch time	1	0.5	0.5	0.5	1	0.5
m_{12}	Cost	0.4	0.5	0.5	0.1	0.1	0.7
m_{13}	Supported network protocols	0.5	1	1	1	1	0.6
M			5.66	5.80	5.24	6.94	6.73

Here the metric M in equation 3.10 is computed for five candidate RTOS (A through E). From the last row it can be seen that RTOS D is the best fit in this case.

3.5 CASE STUDY: POSIX

POSIX is the IEEE's Portable Operating System Interface for Computer Environments. The standard provides compliance criteria for operating system services and is designed to allow applications programs to write applications that can easily port across operating systems. POSIX compliant systems are used widely in real-time applications. It is the intention here to describe the POSIX standard for the purposes of further explicating the concepts of RTOS with a robust and useful example.

3.5.1 Threads

Real-time POSIX provides a mechanism for creating concurrent activities by allowing for each process to contain several threads of execution. POSIX threads are very similar to Java's threads and Ada's tasks models.

POSIX threads (or `Pthreads`) are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library, though this library may be part of another library, such as `libc`. The following is the C interface for thread management in POSIX.[6]

```
Types are defined in #include <sys/types.h>
pthread_t       /* Used to identify a thread. */
phread_attr_t   /* Used to identify a thread attribute object. */
size_t          /* Used for sizes of objects. */

/* initialize and destroy threads attribute object */
   int   pthread_attr_init(pthread_attr_t *);
   int   pthread_attr_destroy(pthread_attr_t *);
```

[6] Refer to the UNIX man (online manual) pages to get further details on C interfaces to various POSIX functions.

```
/* cancel execution of a thread */
   int   pthread_cancel(pthread_t);
/* detach a thread */
   int   pthread_detach(pthread_t);
/* compare thread IDs */
   int   pthread_equal(pthread_t, pthread_t);
/* thread termination */
   void  pthread_exit(void *);
/* wait for thread termination */
   int   pthread_join(pthread_t, void **);
/* get calling thread's ID */
   pthread_t pthread_self(void);
/** Stack and scheduling related **/
/* set and get detachstate attribute */
   int   pthread_attr_setdetachstate(pthread_attr_t *, int);
   int   pthread_attr_getdetachstate(const pthread_attr_t *, int *);
/* set and get inheritsched attribute */
   int   pthread_attr_setinheritsched(pthread_attr_t *, int);
   int   pthread_attr_getinheritsched(const pthread_attr_t *, int *);
/* set and get schedparam attribute */
   int   pthread_attr_setschedparam(pthread_attr_t *, const struct sched_param
*);
   int   pthread_attr_getschedparam(const pthread_attr_t *, struct sched_param
*);
/* dynamic thread scheduling parameters access */
   int   pthread_getschedparam(pthread_t, int *, struct sched_param *);
   int   pthread_setschedparam(pthread_t, int , const struct sched_param *);
/* set and get schedpolicy attribute */
   int   pthread_attr_setschedpolicy(pthread_attr_t *, int);
   int   pthread_attr_getschedpolicy(const pthread_attr_t *, int *);
/* set and get stackaddr attribute */
   int   pthread_attr_setstackaddr(pthread_attr_t *, void *);
   int   pthread_attr_getstackaddr(const pthread_attr_t *, void **);
/* set and get stacksize attribute */
   int   pthread_attr_setstacksize(pthread_attr_t *, size_t);
   int   pthread_attr_getstacksize(const pthread_attr_t *, size_t *);

   int   pthread_getconcurrency(void);
   void *pthread_getspecific(pthread_key_t);
```

The naming conventions being followed in POSIX are shown in the Table 3.12. All identifiers in the threads library begin with `pthread_`.

The following example illustrates how to create multiple threads (five in this example) with the `pthread_create()` routine. Each thread does a simple print, and then terminates with a call to `pthread_exit()`. The example also demonstrates how to "wait" for thread completions by using the `Pthread` join routine.

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

void message_printer_function(void *ptr)
{
  char *message;
  message = (char*) ptr;
  printf("%s\n",message);
}

void main()
```

Table 3.12 POSIX naming scheme

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

```
{
   pthread_t thread[5];
   pthread_attr_t attribute;
   int errorcode,counter, status;
   char *message="TestPrint";
/* Initialize and set thread detached attribute */
   pthread_attr_init(&attribute);
   pthread_attr_setdetachstate(&attribute, PTHREAD_CREATE_JOINABLE);

   for(counter=0;counter<5;counter++)
   {
      printf("I am creating thread %d\n", counter);
      errorcode = pthread_create(&thread[counter],
&attribute,(void*)&message_printer_function,(void*)message);
      if (errorcode)
      {
         printf("ERROR happened in thread creation");
         exit(-1);
      }
   }
/* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attribute);

   for(counter=0;counter<5;counter++)
   {
      errorcode = pthread_join(thread[counter], (void **)&status);
      if (errorcode)
      {
         printf("ERROR happened in thread join");
         exit(-1);
      }
      printf("Completed join with thread %d\n",counter);
      /*printf("Completed join with thread %d status= %d\n",counter, status);*/
   }
   pthread_exit(NULL);
}
```

3.5.2 POSIX Mutexes and Condition Variables

Mutex variables are one of the primary means of implementing thread synchronization. The basic concept of a mutex as used in `Pthreads` is that only one thread is allowed to lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex, only one thread will be successful. No other thread can own/lock that mutex until the owning thread unlocks that mutex, and only the owner can unlock it. POSIX mutexes application program interfaces (APIs) are given below.

```
/** POSIX Mutexes **/
/* Creating/Destroying Mutexes */
   pthread_mutex_init(mutex, attr)
   pthread_mutex_destroy(mutex)
   pthread_mutexattr_init(attr)
   pthread_mutexattr_destroy(attr)

/* Locking/Unlocking Mutexes */
   pthread_mutex_lock(mutex)
   pthread_mutex_trylock(mutex)
   pthread_mutex_unlock(mutex)
```

As compared to mutexes, condition variables provide an alternative for threads to synchronize. The basic difference between mutexes and condition variables is that while mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

Without condition variables, threads need to continually poll (possibly in a critical section) to check if the condition is met. This could lead to unnecessary resource consumption, as the thread would be continuously busy in this activity. A condition variable facilitates to achieve the same goal without polling.

```
/** POSIX Condition Variables **/
/* Creating/Destroying Condition Variables */
   pthread_cond_init(condition, attr)
   pthread_cond_destroy(condition)
   pthread_condattr_init(attr)
   pthread_condattr_destroy(attr)

/* Waiting/Signalling On Condition Variables */
   pthread_cond_wait(condition, mutex)
   pthread_cond_signal(condition)
   pthread_cond_broadcast(condition)
```

The following example demonstrates the use of a mutex where a single reader and a single writer communicate via a shared memory.

```
#include <stdio.h>
#include <pthread.h>
#define SET 1
#define NOTSET 0
int info_in_buffer=NOTSET;
```

```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
void read_user(void)
{
  while(1)
  {
     /* check whether buffer is written and read data*/
      pthread_mutex_lock(&lock);
      if (info_in_buffer==SET)
      {
        printf("In read user \n");
        /* simulation the read operation by a wait (sleep(2)) */
        sleep(2);
        info_in_buffer=NOTSET;
      }
      pthread_mutex_unlock(&lock);
     /* giving the writer an opportunity to write to the buffer*/
      sleep(2);
  }
}

void write_user(void)
{
  while(1)
  {
     /* check whether buffer is free and write data*/
      pthread_mutex_lock(&lock);
      if (info_in_buffer==NOTSET)
      {
      printf("In write user \n");
      /* simulation the write operation by a wait (sleep(2)) */
      sleep(2);
        info_in_buffer=SET;
      }
      pthread_mutex_unlock(&lock);
     /* giving the reader an opportunity to read from the buffer*/
      sleep(2);
  }
}

void main()
{
  pthread_t Readthread;
  pthread_attr_t attribute;
  pthread_attr_init(&attribute);
  pthread_create(&Readthread,&attribute,(void*)&read_user,NULL);
  write_user();
}
```

3.5.3 POSIX Semaphores

POSIX provides counting semaphores and binary semaphores to enable processes running in different address spaces, or threads within the same address space, to synchronize and communicate using shared memory. The following prototypes are self-describing examples of their use.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
/* Initializes the semaphore object pointed by 'sem' */

int sem_destroy(sem_t *sem);
/* Destroys a semaphore object and frees up the resources it might hold */

/* The following three functions are used in conjunction with other
   processes. See man pages for more details.
*/
sem_t *sem_open(const char *name, int oflag, ...);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);

int sem_wait(sem_t *sem);
/* Suspends the calling thread until the semaphore pointed to by 'sem' has
   non-zero count. Decreases the semaphore count. */

int sem_trywait(sem_t *sem);
/* A non-blocking variant of sem_wait. */

int sem_post(sem_t *sem);
/* Increases the count of the semaphore pointed to by 'sem'. */

int sem_getvalue(sem_t *sem, int *sval);
/* Stores the current count of the semaphore 'sem' in 'sval'. */
```

3.5.4 Using Semaphores and Shared Memory

It is important that two processes not write to the same area of shared-memory at the same time, and this is where the semaphores are useful. Before writing to a shared memory region, a process can lock the semaphore to prevent another process from accessing the region until the write operation is completed. When the process is finished with the shared-memory region, the process unlocks the semaphore and frees the shared-memory region for use by another process.

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#include<sys/sem.h>

sem_t writer_lock;
sem_t reader_lock;

void read_user(void)
{
  while(1) {
    sem_wait(&reader_lock);
    /* simulate read operation by a delay*/
    printf("in reader task \n");
    sleep(2);
    sem_post(&writer_lock);
  }
}

void write_user(void)
{
```

```
  while(1) {
    sem_wait(&writer_lock);
    /* simulate read operation by a delay*/
    printf("in writer task \n");
    sleep(2);
    sem_post(&reader_lock);
  }
}

void main()
{
  pthread_t read_thread;
  pthread_attr_t attribute;
  sem_init(&writer_lock,0,1);
  sem_init(&reader_lock,0,1);
  sem_wait(&reader_lock);
  pthread_attr_init(&attribute);
  pthread_create(&read_thread,&attribute,(void*)&read_user,NULL);
  write_user();
}
```

3.5.5 POSIX Messages

Message queues work by exchanging data in buffers. Any number of processes can communicate through message queues. Message notification can be synchronous or asynchronous. The POSIX message passing through message-queue facilities provide a deterministic, efficient means for interprocess communication. Real-time message passing is designed to work with shared memory in order to accommodate the needs of real-time applications with an efficient, deterministic mechanism to pass arbitrary amounts of data between cooperating processes. The following prototypes describe the POSIX messaging capabilities.

```
mqd_t mq_open(const char *name, int oflag, ...);
/* Connects to, and optionally creates, a named message queue. */

int mq_send(mqd_t mqdes, const char *msg_ptr, oskit_size_t msg_len,
            unsigned int msg_prio);
/* Places a message in the queue. */

int mq_receive(mqd_t mqdes, char *msg_ptr, oskit_size_t msg_len,
               unsigned int *msg_prio);
/* Receives (removes) the oldest, highest priority message from the queue. */

int mq_close(mqd_t mqdes);
/* Ends the connection to an open message queue. */

int mq_unlink(const char *name);
/* Ends the connection to an open message queue and causes the
 queue to be removed when the last process closes it. */

int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr
*omqstat);
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
/* Set or get message queue attributes. */
```

```
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
/* Notifies a process or thread that a message is available in the queue. */
```

The following example illustrates sending and receiving messages between two processes using a message queue [Marshall96]. The following two programs should be compiled and run at the same time to illustrate the basic principle of message passing:

`message_send.c`	Creates a message queue and sends one message to the queue.
`message_rec.c`	Reads the message from the queue.

The full code listing for `message_send.c` is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ     128
/* Declare the message structure.  */
typedef struct msgbuf {
         long    mtype;
         char    mtext[MSGSZ];
         } message_buf;

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;

     /* Get the message queue id for the  "name" 1234, which was created by the
server.  */
    key = 1234;

(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\ %#o)\n", key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0)
    {
        perror("msgget");
        exit(1);
    }
    else
     (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

     /* We'll send message type 1 */

    sbuf.mtype = 1;

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
    (void) strcpy(sbuf.mtext, "Did you get this?");
    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
```

```
    buf_length = strlen(sbuf.mtext) + 1 ;

    /* Send a message.  */

    if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
        exit(1);
    }

   else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is created with a basic key and message flag `msgflg = IPC_CREAT | 0666 --` create queue and make it read and appendable by all.
- A message of type (`sbuf.mtype`) 1 is sent to the queue with the message "Did you get this?"

Receiving the preceding message as sent using `message_send` program is illustrated below. The full code listing for `message_send.c`'s companion process, `message_rec.c` is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define MSGSZ     128

/* Declare the message structure. */

typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;

main()
{
    int msqid;
    key_t key;
    message_buf  rbuf;

    /* Get the message queue id for the "name" 1234, which was created by the
server.  */

    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0) {
        perror("msgget");
        exit(1);
    }
```

```
    /* Receive an answer of message type 1. */
    if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }
    /* Print the answer.  */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is opened with msgget (message flag 0666) and the same key as `message_send.c}`.
- A message of the same type 1 is received from the queue with the message "Did you get this?" stored in `rbuf.mtext.`

3.5.6 Real-Time POSIX Signals

Signals are software representation of interrupts or exception occurrences. Signals asynchronously alter the control flow of a task. It is important to point out that no routine should be called from a signal handler that might cause the handler to block – it makes it impossible to predict which resources might be unavailable. Signals are used for many purposes:

- Exception handling
- Process notification of asynchronous event occurrence
- Process termination in abnormal situations
- Interprocess communication

However, there are several limitations of standard POSIX signals on their use in real-time applications. These include:

- Lack of signal queueing
- No signal delivery order
- Poor information content
- Asynchrony

POSIX real-time extensions (POSIX.4) improves the POSIX signals to applications. POSIX.4 defines a new set of application-defined real-time signals, and these signals are numbered from `SIGRTMIN` to `SIGRTMAX`. There must be `RTSIG_MAX >8` signals in between these two limits.

The `sigaction` defines all the details that a process need to know when a signal arrives. As real-time signals can be queued, the queueing option for a real-time signal is chosen by setting bit `SA_SIGINFO` in the `sa_flags` field of the `sigaction` structure of the signal.

```
struct sigaction{
  void (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;//SA_NOCLDSTOP or SA_SIGINFO
  void (*sa_sigaction)(int, siginfo_t*, void*);
  /*used for real-time signals!! Also, ''SA_SIGINFO''
    is set in ''sa_flags.''
   */
};

int sigaction(int sig, const struct sigaction *reaction,
              struct sigaction *oldreaction);
```

Real-time signals can carry extra data. `SA_SIGINFO` increases the amount of information delivered by each signal. If `SA_SIGINFO` is set, then the signal handlers have as an additional parameter a pointer to a data structure called a `siginfo_t` that contains the date value to be piggybacked.

The `sigqueue()` includes an application-specified value (of type `sigval`) that is sent as a part of the signal. It enables the queuing of multiple signals for any task. Real-time signals can be specified as offsets from `SIGRTMIN`. All signals delivered with `sigqueue()` are queued by numeric order, lowest numbered signals delivered first.

POSIX.4 provides a new and more responsive (or fast) synchronous signal-wait function called `sigwaitinfo`. Upon arrival of the signal, it does not call the signal handler (unlike `sigsuspend`), but unblocks the calling process.

3.5.7 Clocks and Timers

In developing real-time applications, clearly it is desirable to have facilities to set and get the time. For example, suppose a diagnostic program checks the health of the system periodically. Essentially, the program would execute one round of diagnostics and then wait for a notification to execute again, with the process repeating forever. This is accomplished by having a timer that is set to expire at a particular time interval. When the time interval expires, the program that set the timer is notified, usually through a signal delivery.

3.5.7.1 Time Management In order to generate a time reference, a timer circuit is programmed to interrupt the processor at a fixed rate. The internal system time is incremented at each timer interrupt. The interval of time with which the timer is programmed to interrupt defines the unit of time (also called "tick") in the system (time resolution). Typically, the system time is represented by a long integer (unsigned 32 bits) variable, whereas the value of the tick is stored in a float variable.

The values of the system lifetime (range) for some tick values (granularity) is shown in the Table 3.13. At any time, "`sys_clock`," a variable holding system time, contains the number of interrupts generated by the timer since the Epoch.

Table 3.13 System lifetime

Tick	Lifetime
1 microsecond	71.6 minutes
1 millisecond	50 days
10 millisecond	16 months
1 second	136 years

If k denotes system tick, and n is the value stored in `sys_clock`, then the actual time elapsed is kn.

3.5.7.2 POSIX Clock POSIX allows many clocks to be supported by an implementation. Each clock has its own identifier of type `clockid_t`. The commonly supported "time-of-day clock" is the `CLOCK_REALTIME` clock, defined in the `time.h` header file. The `CLOCK_REALTIME` clock is a systemwide clock, visible to all processes running on the system. The `CLOCK_REALTIME` clock measures the amount of time that has elapsed since 00:00:00 January 1, 1970.

As mentioned, `CLOCK_REALTIME` is commonly used as the `clock_id` argument in all clock functions. Some of the common clock functions and their descriptions are given in the Table 3.14. The value returned by a clock function is stored in a data structure called `timespec` that has two fields of the long-integer type, namely `tv_sec` representing the value in number of seconds since the Epoch, and `tv_nsec` representing the value in nanoseconds.

Table 3.14 POSIX clock functions

Function	Description
`clock_getres`	Returns the resolution of the specified clock `int clock_getres(clockid_t clock_id, struct timespec *res)`
`Clock_gettime`	Returns the current value for the specified value
	`int clock_gettime(clockid_t clock_id, struct timespec *tp)`
`Clock_settime`	Sets the specified clock to the specified value `int clock_settime(clockid_t clock_id, const struct timespec *tp)`

3.5.7.3 Determining Clock Resolution The following example calls the `clock_getres` function to determine clock resolution:

```
#include <unistd.h>
#include <time.h>
main(){
struct timespec clock_resolution;
int stat;
  stat = clock_getres(CLOCK_REALTIME, &clock_resolution);
  printf(''Clock resolution is %d seconds, %ld nanoseconds\n'',
           clock_resolution.tv_sec, clock_resolution.tv_nsec); }
```

3.5.7.4 Retrieving System Time The `clock_gettime` function returns the value of the systemwide clock as the number of elapsed seconds since the Epoch. The `timespec` data structure (used for the `clock_gettime` function) also contains a member to hold the value of the number of elapsed nanoseconds not comprising a full second.

```
#include <unistd.h>
#include <time.h>
main(){
 struct timespec ts;
 clock_gettime(CLOCK_REALTIME, &ts);
 printf(''clock_gettime returns:\n'');
 printf(''%d seconds and %ld nanoseconds\n'', ts.tv_sec, ts.tv_nsec); }
```

3.5.7.5 System Clock Resolution The system clock resolution on DEC's Alpha system is 1/1024 seconds or 976 microseconds),[7] that is, the system maintains time by adding 976 microseconds at every clock interrupt. The actual time period between clock ticks is exactly 976.5625 microseconds. The missing 576 microseconds (1024 * 0.5625) are added at the end of the 1024th tick, that is the 1024th tick advances the system time by 1552 microseconds.

Note that if an application program requests a timer value that is not an exact multiple of the system clock resolution (an exact multiple of 976.5625 microseconds), the actual time period counted down by the system will be slightly larger than the requested time period. A program that asks for a periodic timer of 50 milliseconds will actually get a time period of 50.78 milliseconds.

3.5.7.6 Timer So far, mechanisms that allow setting and getting the time in POSIX have been discussed. Beyond this, it is desirable to time a process's execution so that it gets to run on the processor at a specific time interval. As discussed earlier, POSIX timers provide a mechanism to control the frequency of a program execution. In order to use a timer to time a process it is necessary to:

- Create the time object within the kernel.
- Generate a signal to get notification.
- Choose either relative or absolute timer.

[7] *Guide to Realtime Programming*, Digital Equipment Corp., March 1996.

3.5.7.7 Creating a Timer The first step is to create a timer for the application by using the `timer_create()` function.

```
#include<signal.h>
#include<time.h>
timer_t timer_create(clockid_t clock_id, struct sigevent *event,
                  timer_t *timer_id);
```

As per POSIX standard, different platforms can have multiple time bases, but every platform must support at least the `CLOCK_REALTIME` time base. A timer based upon the system clock called `CLOCK_REALTIME` can be created. The `seconf` argument `event` points to a structure that contains all the information needed concerning the signal to be generated. This is essentially used to inform the kernel about what kind of event the timer should deliver whenever it "fires." By setting it `NULL`, the system is forced to use default delivery, which is defined to be `SIGALRM.`

The return value from `timer_create()` is effectively a small integer that just acts as an index into the kernel's timer tables.

3.5.7.8 Type of Timers Having created the timer, it is necessary to decide what kind of timer functionality it will have – a one-shot timer or a repeating timer. A one-shot timer is armed with an initial expiration time, expires only once, and then is disarmed. A timer becomes a periodic or repeating timer with the addition of a repetition value. The timer expires, then loads the repetition interval, rearming the timer to expire after the repetition interval has elapsed. The function `timer_settime()` actually sets and starts the timer. The struct `itimerspec` simply incorporates two `timespecs` to form a high-resolution interval timer structure:

```
struct itimerspec{
      struct timespec it_value,
                      it_interval;
};
int timer_settime (timer_t timerid, int flag,
                   struct itimerspec *value,
                   struct itimerspec *oldvalue);
```

This function sets the next expiration time for the timer specified. If flag is set to `Timer_ABSTIME`, then the timer will expire when the clock reaches the absolute value specified by `*value.it_value`. If flag is not set to `TIMER_ABSTIME`, the timer will expire when the interval specified by `value->it_value` passes. If `*value.it_interval` is nonzero, then a periodic timer will go off every `value->it_interval` after `value->it_value` has expired. Any previous timer setting is returned in `*oldvalue`. For example, to specify a timer that executes only once, 10.5 seconds from now, specify the following values for the members of the `itimerspec` structure:

```
newtimer_setting.it_value.tv_sec = 10;
newtimer_setting.it_value.tv_nsec = 500000000;
```

```
newtimer_setting.it_interval.tv_sec = 0;
newtimer_setting.it_interval.tv_nsec = 0;
```

To arm a timer to execute 15 seconds from now and then at 0.25-second intervals, specify the following values:

```
newtimer_setting.it_value.tv_sec = 15;
newtimer_setting.it_value.tv_nsec = 0;
newtimer_setting.it_interval.tv_sec = 0;
newtimer_setting.it_interval.tv_nsec = 250000000;
```

3.5.8 Asynchronous Input and Output

I/O operation is a key component in any real-time application. The real-time program is usually responsible for tracking or controlling the external environment in some desired way. Some of the common I/O operations typically found in real-time systems include:

- Data gathering/output from/to devices.
- Data logging (e.g., for monitoring purposes).
- Multimedia applications (playback or recording).
- Operations on (real-time) databases, keyboards, mice, etc.
- I/O devices: joysticks, keyboards.

It is important to note that UNIX I/O is synchronous, that is, the execution of a program has to wait while the I/O takes place. For example, UNIX `read` calls blocks the calling process until the user buffer is filled up with data being requested, or an error occurs during the I/O operation. However, many real-time applications, and those applications requiring high-speed or high-volume data collection and/or low-priority journaling functions, need to perform I/O asynchronously, that is, the system performs the I/O in parallel with the application, which is free to perform other tasks while the data are read in or written. When the I/O completes, the application receives some kind of notification, usually by the delivery of a `signal`.

3.5.8.1 Associated Data Structures Asynchronous I/O (AIO) operations are submitted using a structure called the AIO control block, or `aiocb`. This control block contains asynchronous operation information, such as the initial point for the read operation, the number of bytes to be read, and the file descriptor on which the AIO operation will be performed. The `aiocb` structure contains the following members:

```
struct aiocb{
  int             aio_fildes;    // File descriptor
  off_t           aio_offset;    // File offset
  volatile void   *aio_buf;      // Pointer to buffer
  size_t          aio_nbytes;    // Number of bytes to transfer
```

```
  int             aio_reqprio;   // Request priority offset
  struct sigevent aio_sigevent;  // Signal structure
  int             aio_lio_opcode;// Specifies type of I/O operation
};
```

It is important to understand what actually happens when the `aio_read/aio_write(...)` functions are called. In fact, the following code is performed:

```
lseek(a.aio_fildes, ...); // Seek to position
read(a.aio_fildes,...);   // Read data
sigqueue(...);            // Queue a signal to a process
```

The AIO operation is depicted in Figure 3.23. Error handling for synchronous I/O is simplified by looking at `errno`. For AIO, the system maintains a return value, and an `errno` value, for each asynchronous operation separately. These two values from the system are obtained via functions `aio_return` and `aio_error`. Each function gives back the return value, or `errno` value, associated with the asynchronous operation at that moment.

Multiple I/O operations are permitted in the POSIX AIO specification. AIO allows a combination of several `read` and `write` operations into a single system call, `lio_listio`. The function `lio_listio` permits transfer of large amounts of I/O simultaneously. The function `aio_suspend` waits for particular AIO operations to complete. It does not say which I/O completed, it just returns 0 when it has determined that one of the asynchronous operations has finished. AIO can be canceled using `aio_cancel`, though it is not recommended.

3.5.9 POSIX Memory Locking

Virtual memory, which allows for mapping of virtual addresses to different physical locations, is useful in real-time systems. In addition to paging (and associated thrashing problems), the key disadvantage of page swapping in real-time systems is the lack of predictable execution time. It is not uncommon that an application demands responsiveness that may be measured in microseconds, and the program is waiting milliseconds or more while the operating system is involved in

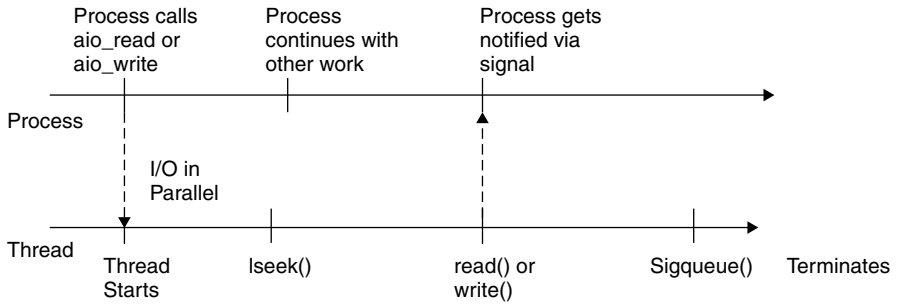

Figure 3.23 Asynchronous I/O operation.

disk access and in fetching the desired instructions in the memory. In a real-time system, it is often desirable to lock the important process's memory down so that the operating system does not page it, thereby making the execution times more predictable. In the case of many large processes, it is desirable to lock just the time-critical portions of these processes.

POSIX allows for a simple procedure to lock the entire process down.

```
#include <unistd.h>
#ifdef _POSIX_MEMLOCK
#include<sys/mman.h>
int mlockall(int flags);
int munlockall(void);
```

The function `mlockall` tries to lock all the memory down; this includes the program text, data, heap, and stack (Figure 3.24). Locking a process includes shared libraries that can be mapped in and other shared memory areas that the process may be using. Depending on the flags being specified, it will either lock all process's current mappings (`MCL_CURRENT`), or the process's current mapping and any future mappings that it may make (`MCL_CURRENT|MCL_FUTURE`). The function `munlockall` unlocks all locked memory and notifies the system that it is okay to page this process's memory if the system must do so. Assuming that `mlockall` is called with the `MCL_FUTURE` flag being set, the rightmost column in Figure 3.24 illustrates the effect of memory locking upon execution of `malloc`. Instead of locking down the entire process, POSIX permits the user to lock down part of the process:

```
#include <unistd.h>
#ifdef _POSIX_MEMLOCK_RANGE
#include<sys/mman.h>
int mlock(void *address, size_t length);
int munlock(void *address, size_t length);
#endif
```

The function `mlock` locks down the address range being specified, and `munlock` unlocks a range (Figure 3.25). If `mlock` is called for a range of memory, then

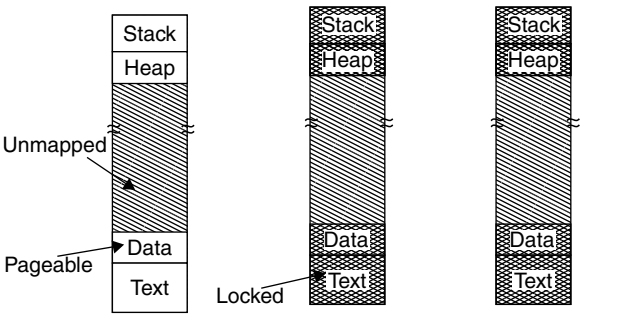

Figure 3.24 `mlockall` operation.

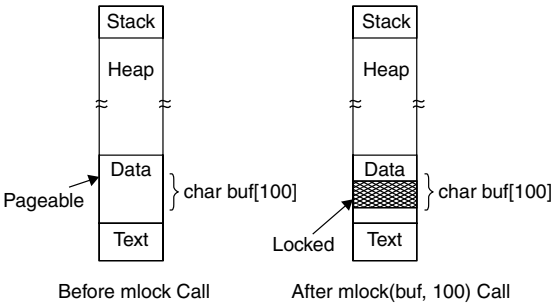

Figure 3.25 `mlock` operation.

calling `munlockall` unlocks the memory that has been locked with `mlock`. It is not possible to lock memory for all of a small section of code, then unlock it. Memory locking is a global technique that should not be performed by small, transient sections of code. In general, once a memory is locked, it should be locked down until the application is out of "real-time mode." The function `mlock` can cause a segmentation fault if an address is passed to it where there is not any memory for executable code.

As can be seen from this example, memory locking decreases execution times for the locked modules and, more importantly, can be used to guarantee execution time. At the same time, it makes fewer pages available for the application, encouraging contention.

3.6 EXERCISES

3.1 For the sample real-time systems described in Chapter 1, discuss which real-time architecture is most appropriate.

(a) Inertial measurement system

(b) Nuclear monitoring system

(c) Airline reservations system

(d) Pasta sauce bottling system

(e) Traffic light control

Make whatever assumptions you like, but document them.

3.2 Should a task be allowed to interrupt itself? If it does, what does this mean?

3.3 What criteria are needed to determine the size of the run-time stack in a multiple-interrupt system? What safety precautions are necessary?

3.4 Identify some of the limitations of existing commercial real-time kernels for the development of different mission- and safety-critical applications.

3.5 What are the desirable features that a system should have to provide for predictability in time-critical applications?

3.6 Discuss the difference between static and dynamic, on-line and off-line, optimal, and heuristic scheduling algorithms.