MODULE SERIES
TRANSDISCIPLINARY ENGINEERING & SCIENCE

REAL TIME OPERATING SYSTEMS

Cengiz Erbas

ISSN: 1933-5423 | TAM-Vol.1-No.3, 2005

The Academy of Transdisciplinary Learning & Advanced Studies
TheATLAS Publications

THEATLAS BOOK SERIES ON TRANSDISCIPLINARY ENGINEERING & SCIENCE

SERIES EDITOR-IN-CHIEF
A. ERTAS

TRANSDISCIPLINE: Integrating science and engineering principles

"...Today, complexity is a word that is much in fashion. We have learned very well that many of the systems that we are trying to deal with in our contemporary science and engineering are very complex indeed. They are so complex that it is not obvious that the powerful tricks and procedures that served us for four centuries or more in the development of modern science and engineering will enable us to understand and deal with them. We are learning that we need a science of complex systems, and we are beginning to construct it..."

Nobel Laureate Herbert A. Simon
Keynote Speech, 2000 IDPT Conference

TheATLAS Publishing

ISSN: 1933-5423

Published in the United States of America by

www.theatlas.org

Abstract

Real-time operating systems refer to a category of operating systems used to build embedded and real-time applications. The first commercial examples of real-time operating systems appeared in the early 1980s. They were running mostly on *Motorola 68K* microprocessor family, and used mainly in *VME-based* control and data acquisition systems. Later in 1990s, real-time operating systems found applications in a wide-variety of industry segments, such as telecommunications, consumer electronics, networking, automotive, digital imaging, military and aerospace. They were ported to a number of CPU families: first to *Intel i960* and *Sun SPARC*, then *PowerPC, Intel x86, MIPS, ARM, Hitachi SH* and many others. Today, tens of different real-time operating systems are employed in the industry for building countless types of devices, including routers, switches, gateways, printers, copiers, scanners, cell phones, wireless basestations, set-top boxes, digital cameras, cable modems, DVD players, digital televisions, GPS receivers, satellite control systems, missiles, weapons, and security systems.

Even though there is an ever-increasing demand in the industry, building embedded and real-time systems is seen by many software engineers as an unknown area that requires special skills and expertise. Understanding the fundamentals of real-time operating systems is key to developing embedded and real-time software. In this module, we will provide an overview of the subject by giving examples from the most popular real-time operating systems, including *VxWorks, Windows CE, RT-Linux, QNX, Nucleus, OSE, OS-9000* and *uITRON*.

Keywords

Real Time Operating Systems, VME-based Control,Embedded and Real Time Software

1 Introduction

The field of Computer Science and Engineering has evolved from diverse disciplines (mathematics, electrical engineering, formal logic, and control systems, among others), and has grown explosively, probably more rapidly than any other discipline in history. In a few decades, the products and the notions of this new field have penetrated almost every aspect of our lives. The industry, as well as academia, has played a fundamental role in the rapid growth of Computer Science. Research advances in academia and the technological base in industry followed a similar pattern of evolution.

In the early days of Computer Science, there was the *era of sequential programs*. Academia developed the fundamentals of Computer Science on the sequential paradigm. During this period, the term *program* was used almost synonymously

with *sequential program* (with no concurrency and communication primitives). A great majority of algorithms of the period had this characteristic. The first axiomatic verification technique was developed for sequential programs [Hoare1969]. The relational algebra of database theory, complexity and automata theory, and the initial concepts in Software Engineering were all built on the elements of this paradigm. The early products of software industry also reflect all the main characteristics of this era. Most of the applications were developed primarily using sequential programs, often with COBOL (for business applications), and FORTRAN (for scientific applications).

The research topics in academia, as well as the technological base in industry, then, evolved into the *era of concurrency*. The concept of *program* was extended with communication and synchronization primitives. The academic research, in this era, covered topics such as, concurrency and synchronization, distributed processing, SIMD and MIMD machines, and parallel algorithms. Axiomatic verification technique was extended for parallel and concurrent programs [Owicki1976a]. New sets of formal models were introduced, such as communicating sequential processes [Hoare1985], and calculus of communicating systems [Milne1980]. Database theory and relational algebra was extended with parallel queries, and concurrency control techniques [Bernstein1987]. The concept of concurrency and communication made a tremendous impact also on software industry. UNIX operating system popularized the fundamental elements of this era. Programming languages, such as C and ADA, which provide communication and synchronization primitives, were introduced. Distributed databases, local area networks, client-server applications, and parallel computers are all samples of industrial products of this period.

Today, there is overwhelming evidence which points to another transformation. The new period will probably be named as *embedded real-time systems era*. The concept of *program*, in this coming period, will be extended with *timing constraints*. The development of the theoretical base of the real-time systems era has already been started within academia. There have been several attempts to extend axiomatic verification technique for real-time systems [Hooman1991, Schneider1991]. Scheduling has arisen as a separate field in Operating Systems research. Several formal models have been introduced to specify and analyze the timing behavior of programs. Temporal logic [Ostroff1989], timed CSP [Davies1993], and timed transition systems [Henzinger1993] are examples of these attempts. There are research projects in the area of real-time database management systems and query processing [Ozsoyoglu1995, Ulusoy1995] to extend the existing techniques with the ability to specify real-time constraints.

Software and hardware industry is also demonstrating a similar pattern of evolution. Predictability of the timing behavior of systems, and scalability and reliability needs are gaining fundamental importance, especially for embedded and

time-critical applications. Telecommunications, networking, digital imaging, consumer electronics, automotive, control systems and aerospace are samples of industry segments which drive this increasing demand for embedded applications. A new generation of operating systems and embedded software development tools has been introduced to address the needs of the new era. *VxWorks*, *Windows CE*, *RT-Linux*, *QNX*, *Nucleus*, *OSE*, *OS-9000* are well known examples of these products. The network industry, today, is in the process of developing new communication protocols (both for LANs and WANs) that can provide real-time services to applications. Real-time operating systems, together with high performance communication networks, will provide the necessary platform for development of future applications of the new period.

The main difference that separates real-time operating systems from general-purpose operating systems is that real-time operating systems are used to build embedded and real-time software, as opposed to traditional desktop applications. So let's first look at the underlying differences between these two types of software, and analyze what kind of operating system support each one requires, as opposed to the other.

1.1 Sequential Programs

We can classify programs into three categories: *sequential* programs, *concurrent* programs and *real-time* programs. A sequential program corresponds to a single thread of execution, that is, only one instruction at a time becomes eligible for execution. The behavior of a sequential program is described in terms of its *inputs* and *outputs*. We use the triplet,

$$5\{p\}\ S\ \{q\}$$

to denote the functional behavior of sequential programs. Here, p and q are assertions about the values of program variables upon entry and exit of S. The assertion p is called the *precondition* of S, and describes the program variables before the execution of S. The assertion q is the *postcondition* of S, and indicates the program state after the execution of S terminates. If S starts execution in a state where p is true, then q becomes true when it terminates.

1.1.1 Sequential Composition

A sequential program may consist of several program segments. We use the notation,

$$S = S_1; S_2; ...; S_n$$

to denote the sequential composition of the program segments S_1, S_2,..., S_n. The behavior of each program segment S_1, S_2, ..., S_n can be described in terms of its own preconditions and postconditions, as follows:

$$\{p_1\}\ S_1\ \{q_1\}$$
$$\{p_2\}\ S_2\ \{q_2\}$$
$$\ldots$$
$$\{p_n\}\ S_n\ \{q_n\}$$

Since the execution of S_2 immediately follows the execution of S_1, the precondition of S_2 should be identical to the postcondition of S_1, that is, $p_2 = q_1$. Similarly, $p_3 = q_2$, and so on. We use the following notation to demonstrate this characteristic of sequential composition[1]:

$$\frac{\{p\}\ S_1\ \{r\}, \{r\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}}$$

This rule allows us to treat the program code as a composition of smaller program segments. Depending on the objective, S_1 and S_2 may correspond to modules, procedures, language constructs, or arbitrary segment of instructions. At the most primitive level, a program can be expressed as the sequential composition of its atomic instructions.

1.1.2 Operating Systems Support for Sequential Composition

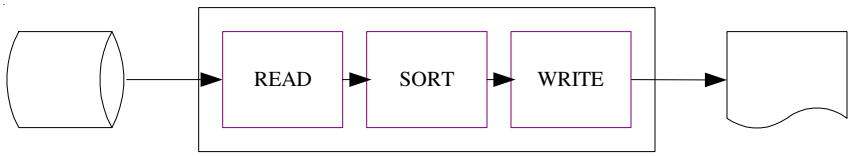
A sequential program *S* does not require an operating system to run on. It can operate on a *bare machine*. However, when running on an operating system, the kernel may interleave the execution of *S* for various reasons: to process timer interrupts and to perform periodic kernel services, to respond to keyboard interrupts, to process incoming network packets, or to schedule another program to run in multiprogramming systems. On such an environment, for correct operation of *S*, the operating system has to ensure that the sequential composition rule holds at all times throughout the execution of *S*. That means the operating system should make sure that the state of the program variables at the completion of each instruction should be maintained until the execution of the next instruction starts, even if the execution of *S* is interleaved. This is achieved simply by the implementation of the context switch mechanism within the operating system. When the execution of *S* is interleaved, the operating system first saves the context of *S* into the stack. When the execution of *S* is resumed, the operating system restores the context from the stack.

---

[1]In general we use the notation H_1, H_2, ..., H_n/C to express that the observable program behavior described by *C* can be derived from the logical combination of the behaviors described by H_1, H_2, …, H_n.

Example 1.1. Consider a program *S* that reads data buffers from an input file, sorts the data elements in the buffer, and writes to a printer. Let's assume that the input file is infinite in length, and each data buffer consists of 64 integers.

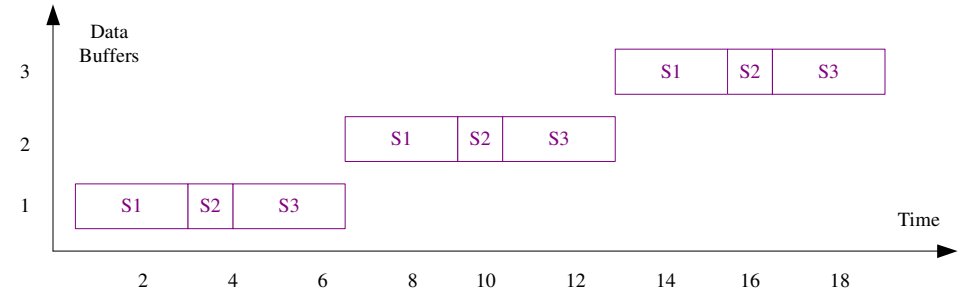
S can be implemented as a sequential program consisting of three program segments, $S = S_1; S_2; S_3$, where S_1 reads the data buffer from the file, S_2 sorts the data elements, and S_3 writes the data to the printer**.**



We will assume that I/O operations (reads from the file and writes to the printer) are performed synchronously. That means after requesting I/O, the program waits until the request has been fulfilled. A pseudo-code implementation of *S* is given below:

```
Read_Sort_Write() {
        int buffer[64];

        while (1) {
                /* {p1} */
                Read a buffer from the file;                    /* S1 */
                /* {q1} */

                /* {p2} */
                Sort the data elements;                         /* S2 */
                /* {q2} */

                /* {p3} */
                Write the buffer to printer;                    /* S3 */
                /* {q3} */
        }
}
```

Let's assume that the execution of *S1* and *S3* takes 2.5 msec and *S2* takes 1 msec per buffer. Since *S* is a sequential program, the processing of data buffers happen one at a time. The timing behavior of *S* can be illustrated as follows:



1.2 Concurrent Programs

A concurrent program consists of multiple threads of execution. The threads of a concurrent program cooperate with each other implicitly by sharing variables, or explicitly by exchanging messages. We use the notation

$$S = S_1 \parallel S_2 \parallel ... \parallel S_n$$

to denote the concurrent execution of the program segments S_1, S_2,... S_n. Depending on the context and the characteristics of the implementation, these program segments are sometimes referred to as *threads*, sometimes *tasks*, and sometimes *processes*. We will compare and contrast threads with tasks and processes in Chapter 2. Until then, we use *thread* as a general term to refer to concurrently running program segments.

Similar to sequential program segments, the behavior of each thread (or task, or process) S_1, S_2,... S_n can be described in terms of the state of the program variables before and after the execution of that thread:

$$\{p_1\}\ S_1\ \{q_1\}$$
$$\{p_2\}\ S_2\ \{q_2\}$$
$$...$$
$$\{p_n\}\ S_n\ \{q_n\}$$

However, we cannot treat each thread in isolation from the others as if it is a sequential program segment. We cannot guarantee that the state of program variables after the execution of an instruction will be maintained until the execution of the next instruction starts, because the execution can be interleaved with the execution

of another thread. Therefore, the *interference* between threads should also be taken into account to describe the semantics of concurrency. That is:

$$\frac{\{p_1\}\ S_1\ \{q_1\}, \{p_2\}\ S_2\ \{q_2\}, \textit{interference considerations}}{\{p_1 \wedge p_2\}\ S_1 \parallel S_2\ \{q_1 \wedge q_2\}}$$

1.2.1 Interference and Mutual Exclusion

Interference may arise when concurrent threads have access to shared variables. Let $S_1 = s_{11}; s_{12}; ...; s_{1e}$ and $S_2 = s_{21}; s_{22}; ...; s_{2f}$. The interference considerations are used to ensure that the execution of s_{2l} does not invalidate the assertions for S_1. We should demonstrate that

$$\{p_{1j} \wedge p_{2l}\}\ s_{2l}\ \{p_{2j}\}$$

$$\{q_1 \wedge p_{2l}\}\ s_{2l}\ \{q_1\}$$

where $1 \le j \le e$ and $1 \le l \le f$. If the above requirements do not hold, we say that interferes with S_l.

In order to ensure that the execution of s_{1j} and $s_{1\,j+1}$ are not interleaved by the execution of another statement s_{2l}, we can enforce $s_{2\,j}$ and $s_{2\,j+1}$ to be executed *atomically.* We use the following notation to restrict the interleaved executions of program segments:

$$\text{await } B \rightarrow S$$

If the Boolean expression *B* is true, then S is executed atomically. Otherwise, the execution of the statement is disabled until a later time when *B* is true. The evaluation of *B* and the execution of *S* are not interleaved with the execution of statements in concurrent threads. The behavior of *await* can be described as follows:

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \text{await } B \rightarrow S\ \{q\}}$$

The *await* statement is used to reduce the number of possible interleaved execution scenarios and to eliminate situations in which interference can occur.

1.2.2. Operating Systems Support for Mutual Exclusion

The operating systems provide two commonly used mechanisms to help eliminate interference and to protect concurrent access to shared variables. These are:

1. *Support for enforcing atomicity*: The ability to *lock the interrupts* or to *disable context switch* so that program segments can be executed atomically.
2. *Support for general mutual exclusion*: The capability to use *semaphores* to protect access only to the critical sections of program segments.

Enforcing atomicity through locking interrupts or a disabling context switch is sometimes undesirable because it excludes the interleaved execution of *any* two program segments regardless of whether they access to shared variables and can therefore interfere with each other.

A more efficient approach is to identify *critical sections*, and ensure that the execution of statements in a critical section of S_1 is not interleaved with the execution of statements in a critical section of S_2. However, we do not enforce atomicity for the execution within a critical section. Therefore, while S_1 is executing in a critical section, S_2 can be executing a noncritical section. In operating systems, mutual exclusion is generally implemented using *semaphores*.

Semaphores are implemented based on the semantics of the await statements, as follows:

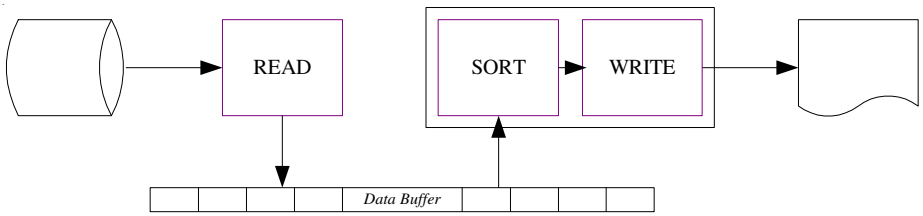
$$\texttt{P}: \quad \text{await } sem > 0 \rightarrow sem := sem - 1;$$
$$\texttt{V}: \quad sem := sem + 1;$$

Operating systems provide support for two different types of semaphores: *binary* semaphores and *counting* semaphores.

Example 1.2. An example for a sequential program is given in Example 1.1, where we developed a sequential program responsible for reading data elements from a file, sorting them and writing to printer. For demonstration purposes let's assume that the read and write operations do not require any CPU processing and can be performed in parallel. Then we can restructure the program to exploit this parallelism, as follows:

$$S = S_1 \parallel (S_2 ; S_3)$$

where S_1 corresponds to the thread that reads the data, and $(S_2; S_3)$ correspond to the one that sorts the data elements and writes to a printer.



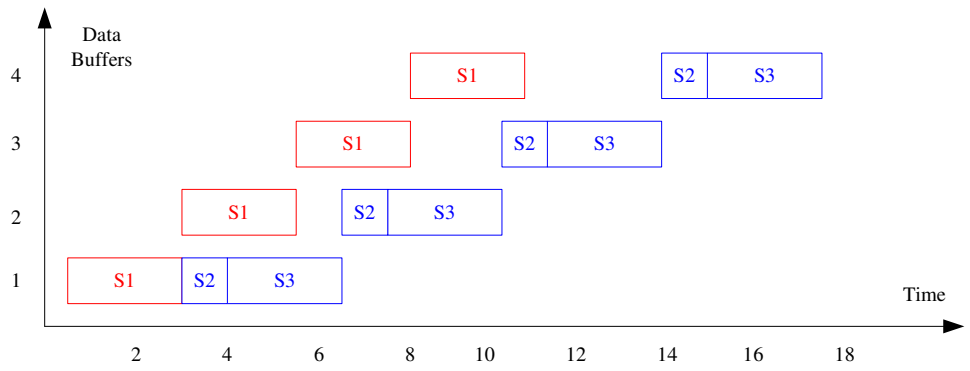
In this implementation, the two threads communicate with each other through the use of a shared memory data buffer. S1 allocates empty buffers and fills with data; S2 receives and processes the filled buffers; and S3 releases the buffers. In order to eliminate interference, we need to protect data buffers from getting accessed by both threads concurrently. That means accesses to the buffer pointers should be treated as critical regions and be protected by semaphores.

A pseudo-code implementation of *S* is given below:

```
int buffer[512];
int pBuffer;

/* Read thread */

Read() {
        while (1) {
                Lock semaphore;
                Get the buffer pointer;
                Unlock semaphore;

                /*  {p1}  */

                Read a buffer from a file;                    /* S1 */
                /*  {q1}  */

                Lock semaphore;
                Release the buffer pointer;
                Unlock semaphore;
        }
}

/* Sort and Write thread */

Sort_Write() {
        while (1) {
                Lock semaphore;
                Get the buffer pointer;
                Unlock semaphore;
```

```
                /* {p2} */
                Sort the data elements;                          /* S2 */
                /*  {q2}  */

                /*  {p3}  */
                Write the buffer to printer;                     /* S3 */
                /*  {q3}  */

                Lock semaphore;
                Release the buffer pointer;
                Unlock semaphore;
        }
}
```

Let's assume that the execution of *S1* and *S3* takes 2.5 msec and *S2* takes 1 msec for each buffer. Since $S = S_1 \parallel (S_2; S_3)$ contains two concurrent threads of execution, reading the data buffers (by S_1) happen concurrently with the processing of data buffers (by S_2; S_3). The timing behavior of *S* can be illustrated as follows:



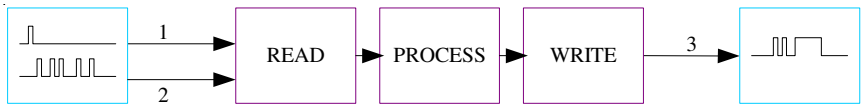
1.3 Real-Time Programs

A real-time program is a concurrent program that interacts with the physical environment within a predictable delay. The concurrent threads of a real-time program are called *controller* processes. The interactions between a controller process and the physical environment resemble the interactions between concurrent processes. However, this time one of the concurrent threads corresponds to a *plant* process representing the physical environment. The interactions with the plant process put certain demands on the timing behavior of real-time programs.

A real-time program interacts with the plant process to:

1. Monitor the events in the physical environment,
2. Acquire data from the environment when an event is detected, and
3. React to the event by processing the data elements and controlling/modifying the environmental parameters.

The following figure illustrates these three types of interactions between a controller process and a real-time program.



A real-time program detects an event in the environment either by receiving an interrupt or by polling an input port for status. It acquires data from the environment by reading input ports, and modifies the environmental parameters by writing to output ports.

The nature of the physical environment and the characteristics of the application impose certain timing constraints for the real-time program. The physical environment defines the minimum time distance between two consecutive events; and the application requirements specify the acceptable time delay to react to an event. In response, the real-time program may be required to detect all the events occurring in the environment, without missing any. Or it may be required to react to all the events within the acceptable time delay. A real-time program, therefore, is required to demonstrate certain timing properties in addition to meeting functional requirements.

The functional behavior, $\{p\}\ S\ \{q\}$, of a sequential or a concurrent program S can be deduced directly from its source code. On the contrary, the program code alone is not sufficient to specify real-time programs. Running a program, for example, on a faster processor generates a different timing behavior than running the same program on a slower processor. Similarly, running a program on a processor as the only task exhibits a different timing behavior than running the same program on the same processor with other tasks. The scheduling policy of a system affects the timing behavior of programs running on that platform. Therefore, the timing characteristics of the *run-time environment* should also be included in the specification of real-time programs.

1.3.1 Timing Constraints

In order to add *time* into the specification of programs, we first extend the program code $S = S_1; S_2; \ldots S_n$ with instructions of the form $< t = t + T_{S_i} ; >$

where t is a variable corresponding to *time*, and T_{S_i} is the execution time of the program segment S_i on a given processor. Here t represents the clock ticks of the processor. We refer to the extended program code as S^t.

$$S^t = S; < t = t + T_S >$$

Then we extend the preconditions and postconditions with timing constraints, as follows:

$$\{p \wedge p^t\}\ S^t\ \{q \wedge q^t\}$$

Here p^t and q^t are assertions about the value of the *time* variable t upon entry and exit of S. If the execution of S starts at a time t where the assertion is true, then the execution of S terminates at a time where becomes true.

Timing constraints can be added into the rule of sequential composition as follows:

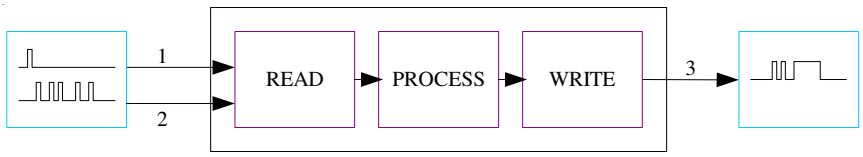
$$\frac{\{p \wedge p^t\}\ S_1^t\ \{r \wedge r^t\}, \{r \wedge r^t\}\ S_2^t\ \{q \wedge q^t\}}{\{p \wedge p^t\}\ S_1^t; S_2^t\ \{q \wedge q^t\}}$$

This rule allows us to treat the program code as a composition of smaller program segments, as follows:

$$S^t : \begin{Bmatrix} S_1; & t = t + T_{S_1}; \\ S_2; & t = t + T_{S_2}; \\ & \ldots \\ S_n; & t = t + T_{S_n}; \end{Bmatrix}$$

$$S^t = S_1; S_2; \ldots S_n; t = t + \sum_{i=1}^{n} T_{S_i}$$

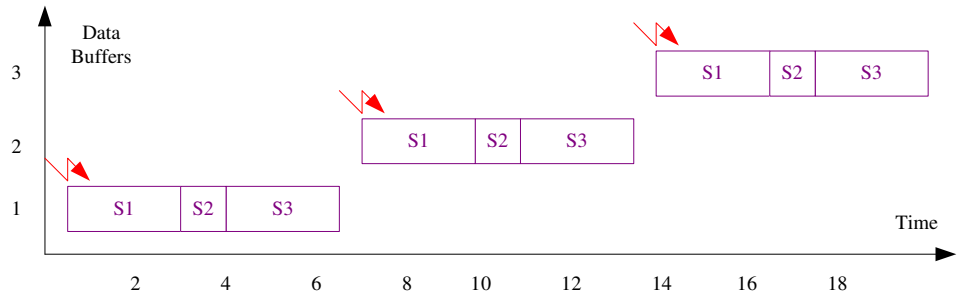
Example 1.3. Consider a program that reads a set of data from an input port, processes the data, and writes into an output port. Let's assume that the input port is connected to a physical process that generates events periodically once every t microseconds. Such a program can consist of three program segments, $S = S_1; S_2; S_3$, where S_1 corresponds to the program segment that polls for an event and reads the data from the input port. S_2 processes the data, and S_3 writes to the output port.



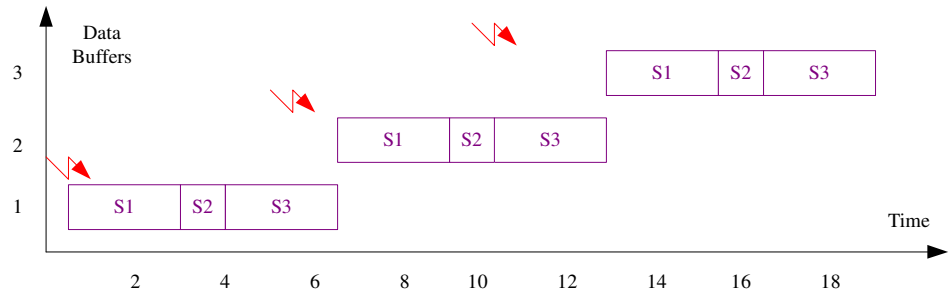
A pseudo-code implementation of S is given below:

```
Read_Process_Write() {
        while (1) {
                Poll input ports for an event;

                /* {p1} */
                Read the input ports;                          /* S1 */
                /* {q1} */

                /* {p2} */
                Process the data elements;                     /* S2 */
                /* {q2} */

                /* {p3} */
                Write to output ports;                         /* S3 */
                /* {q3} */
        }
}
```

Let's assume that the execution of *S1* and *S3* takes 2.5 msec and *S2* takes 1 msec per buffer. Let's also assume that events are taking place in the physical environment periodically once every 5 msec. Since *S* is a sequential program, the processing of data buffers happen one at a time. The timing behavior of *S* can be illustrated as follows:



If we assume that the events are taking place once every 5 msec, then the timing behavior of S can be shown as follows:



1.3.2 Multiprocessing and Multiprogramming Systems

Because the timing characteristics of the run-time environment is of critical importance, we differentiate multiprogramming (interleaved execution) from multiprocessing (maximal parallelism) when analyzing timing behavior of systems.

1. *Multiprocessing systems*: n processes are running on n processor, each on a separate processor. We use the notation $S = S_1 \parallel S_2 \parallel ... \parallel S_n$ to denote the concurrent execution of threads $S_1, S_2, ... S_n$ on a multiprocessing platform.

2. *Multiprogramming systems*: n processes are running on a single processor. We use the notation $S = S_1 \;|||\; S_2 \;|||\; ... \;|||\; S_n$ to represent concurrent execution of threads on a multiprogramming platform.

In multiprocessing systems, each processor and thus each thread operate on a separate time variable t_i. The timing behavior of two threads running on a multiprocessing system can be described as follows:

$$\frac{\{p_1 \wedge p_1^{t_1}\}\, S_1^{t_1}\, \{q_1 \wedge q_1^{t_1}\}, \{p_2 \wedge p_2^{t_2}\}\, S_2^{t_2}\, \{q_2 \wedge q_2^{t_2}\}}{\{p_1 \wedge p_1^{t_1} \wedge p_2 \wedge p_2^{t_2}\}\, S_1^{t_1} \parallel S_2^{t_2}\, \{q_1 \wedge q_1^{t_1} \wedge q_2 \wedge q_2^{t_2}\}}$$

Since every processor operates with its own system clock, under maximal parallelism, we keep a different time variable t_i for each processor. This variable is only modified by S^{t_i}, and therefore it keeps track of the execution time of S_i.

Multiprogramming systems assume interleaved execution of concurrent threads based on a *scheduling* policy. The timing behavior of multiprogramming systems

can thus only be analyzed with respect to a scheduling policy. The timing behavior of two threads running on a multiprogramming platform can be described as follows:

$$\frac{\{p_1 \wedge p_1^t\}\ S_1^t\ \{q_1 \wedge q_1^t\}, \{p_2 \wedge p_2^t\}\ S_2^t\ \{q_2 \wedge q_2^t\},\ \textit{scheduling considerations}}{\{p_1 \wedge p_1^t \wedge p_2 \wedge p_2^t\}\ S_1^t \ |||\ S_2^t\ \{q_1 \wedge q_1^t \wedge q_2 \wedge q_2^t\}}$$

Notice that all the threads running on a multiprogramming system share the same time variable *t*. In other words, time variable *t* serves as a shared variable for all the threads. Therefore, the interference considerations outlined in Section 1.2.1 apply to the time variable *t* as well. Since the time slices are allocated to threads by the scheduler, the affect of the scheduling policy should also be included as a part of the interference considerations. The most commonly used scheduling policies and their impact on the timing behavior of multiprogramming systems will be overviewed in Section 2.

Interrupt handling can be considered as the simplest form of interleaved execution. An occurrence of an interrupt I preempts the execution of a thread S_1, that is $S = S_1 \ ||| \ I$. This resembles the interleaved execution of two threads; however, this time one of the threads corresponds to the interrupt service routine.

1.3.3 Operating Systems Support for Timing Constraints

The functional behavior of sequential and concurrent programs can be deduced directly from the source code. However, as shown in the previous section, the program code alone is not sufficient to specify real-time programs. We made the following three extensions to specify and analyze timing constraints:

1. *Time instructions*: First we introduced a time variable *t*, and extend the program code $S = S_1; S_2; ... S_n$ with instructions of the form $< t = t + T_{S_i} ;>$ where T_{S_i} is the execution time of the program segment S_i.
2. *Time assertions*: Then we extended the precondition *p* and the postcondition *q* with assertions p^t and q^t about the value of *t* upon entry and exit of *S*.
3. *Scheduling considerations*: Finally we separated multiprogramming platforms from the multiprocessing platforms, and extended the rule of concurrency for multiprogramming systems with scheduling considerations.

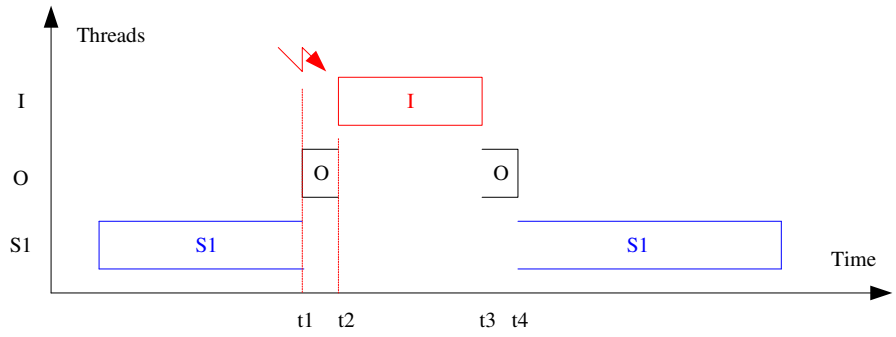
These extensions make certain assumptions about the underlying run-time environment. We can summarize the expectations that each extension put on the run-time environment in the following table:

Extensions	Run-time Environment Requirements
Time instructions	Deterministic execution time for instructions. Deterministic response time for Operating Systems services.
Time assertions	Minimum time distance between two consecutive events.Maximum acceptable time delay to react to an event.
Scheduling considerations	Deterministic interrupt latency.Deterministic context switch time.Priority-based scheduling as opposed to fairness-based.

To be able to extend the program code with timing instructions, we should have the knowledge of the execution time of the instructions. The processors typically provide deterministic execution time for instructions. One exception to this is the time spent to access cache versus memory versus virtual memory. The time instructions also require deterministic response time for the services provided by the operating system. For example, locking and unlocking a semaphore, or sending and receiving a message should be time bounded. The execution time of these primitives should not depend on the system state, such as the number of the tasks waiting for these objects in the system. Real-time operating systems, as opposed to general-purpose operating systems, provide deterministic timing interface for the programs.

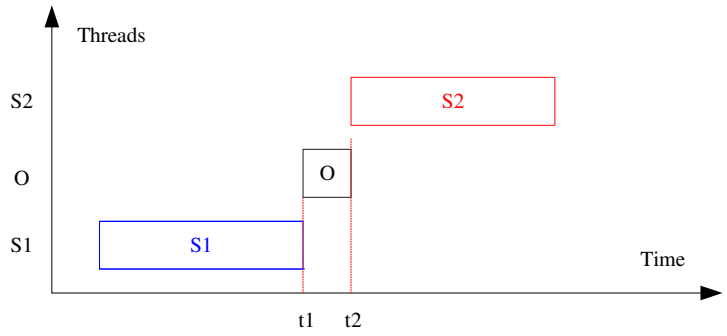
To be able to extend the preconditions and postconditions with time assertions, we should have the knowledge of the timing behavior of the physical environment. This includes the minimum time distance between two consecutive events and the maximum acceptable time delay to react to an event. The first parameter defines how quickly the system should read the environmental parameters after the occurrence of an event. The second one defines how fast the system should process the data and respond to that event.

To be able to include scheduling into the timing behavior analysis of multiprogramming systems, we should have the knowledge of the timing behavior of the scheduler. This includes the timing behavior when handling interrupts and when performing context switch between threads. The operating system may perform certain operations before calling the interrupt service routines to serve the interrupt, and after the completion of the interrupt service routine, as illustrated below:



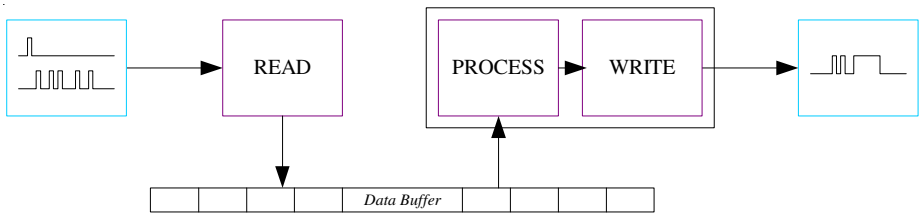
Let's assume that an interrupt is received at time t_1. Then the execution of the current task is interrupted and the interrupt service routine is invoked with a delay at time t_2. Here, the time delay $t_2 - t_1$ is called the *interrupt latency* of the system. The execution of the interrupt service routine completes at time t_3 and the execution returns to S_1 at time t_4. The operating system *overhead* is given by $(t_2 - t_1) + (t_4 - t_3)$. Real-time operating systems provide deterministic and low interrupt latency, which makes it easy to analyze the timing behavior of such systems.

The multiprogramming systems in general consist of interleaved execution of at least two threads. The operating system performs certain operations when switching the context from one thread to another, as illustrated below:



Here, the time delay $t_2 - t_1$ is called the *context-switch time*. This is the time it takes to change context from an executing thread to another thread. To be able to guarantee timing behavior, real-time operating systems provide deterministic context switch time.

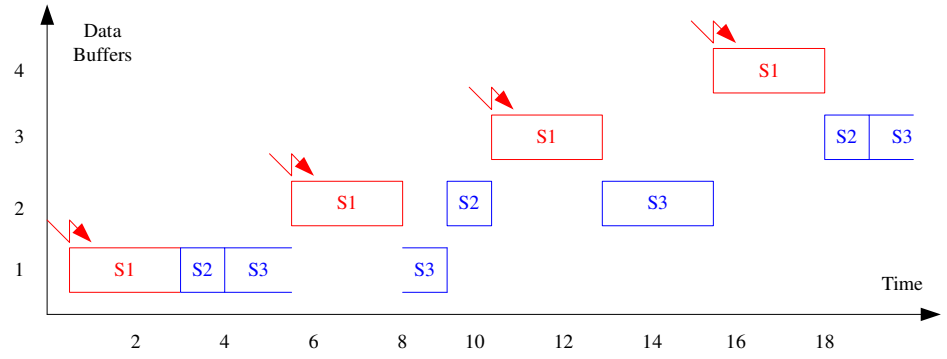
Example 1.4. Consider a program that reads a set of data from an input port, processes the data, and writes into an output port. Let's assume that the input port is connected to a physical process that generates events periodically once every t microseconds. Such a program can consists of two program segments, $S = S_1 \,||\, S_2$, where S_1 correspond to the interrupt handler that reads the data from the input port and puts into a buffer, and S_2 processes the data, and writes to the output port.



A pseudo-code implementation of S is given below:

```
int buffer[64];
int pBuffer;

Read_ISR() {
        /* {p1} */
        Read the input ports into a data buffer;          /* S1 */
        /* {q1} */
}

Process_Write() {
        while (1) {
                /* {p2} */
                Process the data elements;                /* S2 */
                /* {q2} */

                /* {p3} */
                Write the buffer to an output port;       /* S3 */
                /* {q3} */
        }
}
```

Let's assume that the execution of *S1* and *S3* takes 2.5 msec and *S2* takes 1 msec per buffer. Let's also assume that events are taking place in the physical environment periodically once every 5 msec. Since *S* is a sequential program, the processing of data buffers happens one at a time. The timing behavior of *S* can be illustrated as follows:



1.4 Bibliography

P.A. Bernstein, V. Hadzilacos, N. Goodman, 1987, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley.

J. Davies and S. Schneider, 1993, An introduction to timed CSP. Technical Monograph PRG-75, Oxford University Computing Lab.

T.A. Henzinger, Z. Manna, and A. Pnueli, 1994, Temporal Proof Methodologies for Timed Transition Systems, *Information and Computation*, (112):273-337.

C.A. Hoare, 1969, An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576-580.

C.A. Hoare, 1985, *Communicating Sequential Processes*. Prentice-Hall.

J. Hooman, 1991, Compositional Verification of Real-Time Systems using Extended Hoare Triples. *Real-Time: Theory in Practice*, REX Workshop Proceedings, LNCS 600, Springer-Verlag.

J.S. Ostroff, 1989, *Temporal Logic for Real-Time Systems*. Research Studies Press, England.

S. Owicki, D. Gries, 1976, *An axiomatic proof technique for parallel programs*, Acta Informatica, (6):319-340.

G. Ozsoyoglu, R.T. Temporal and Real-Time Databases: A Survey, *IEEE Transactions on Knowledge and Data Engineering*, 7(4).

R. Milne, 1980, *A Calculus of Communicating Systems*. Springer-Verlag.

F.B. Schneider, B. Bloom, K. Marzello, 1991, *Putting Time into Proof Outlines*, in Proceedings of the REX Workshop, "Real-Time Theory in Practice", LNCS 600, Springer-Verlag.

O. Ulusoy, 1995, Research Issues in Real-Time Database Systems, *Information Sciences*, 87(1-3).