Chapter 1

What is Real-Time Processing?

Consider a software system in which the inputs represent digital data from hardware such as imaging devices or other software system's and the outputs are digital data that control external hardware such as displays. The time between the presentation of a set of inputs and the appearance of all the associated outputs is called the **response time.** A **real-time system** is one that must satisfy explicit bounded response time constraints to avoid failure. Equivalently, a real-time system is one whose logical correctness is based both on the correctness of the outputs and their timeliness. Notice that response times of, for example, microseconds are not needed to characterize a real-time system – it simply must have response times that are constrained and thus predictable. In fact, the misconception that real-time systems must be "fast" is because in most instances, the deadlines are on the order of microseconds. But the timeliness constraints or deadlines are generally a reflection of the underlying physical process being controlled. For example, in image processing involving screen update for viewing continuous motion, the deadlines are on the order of 30 microseconds. In practical situations, the main difference between real-time and non-real-time systems is an emphasis on response time prediction and its reduction.

Upon reflection, one realizes that every system can be made to conform to the real-time definition simply be setting deadlines (arbitrary or otherwise). For example, a one-time image filtration algorithm for medical imaging, which might not be regarded as real-time, really is real-time if the procedure is related to an illness in which diagnosis and treatment have some realistic deadline. Because all systems can be made to look as if they were real-time, we refine the definition somewhat in terms of the system's tolerance to missed deadlines. For example, **hard real-time systems** are those where failure to meet even one deadline results in total system failure. In **firm real-time systems** some fixed small number of deadlines can be missed without total system failure. Finally, in **soft real-time systems** missing deadlines leads to performance degradation but not failure. Unless otherwise noted, when we say "real-time" throughout this tutorial, we mean hard real-time.

Another common misconception is that the study of real-time processing is really a non-issue because hardware is always getting faster. By throwing faster hardware at the problem deadlines can always be met. However, as we just stated, unless one can predict performance and hence bound response times, one can never be satisfied that deadlines are always being achieved. Moreover, faster hardware is not always available or suitable for certain applications.

Some feel that real-time performance is easy to achieve. As we hope to show in this tutorial, that is not always so, largely because most hardware and programming languages are not suitable for real-time demands.

1.1 Characteristics of Real-Time Systems

Real-time systems are often reactive and/or embedded systems. **Reactive systems** are those in which functionality is driven by ongoing, sporadic interaction with their environment, such as in virtual reality. **Embedded systems** generally do not have a generalized operating system interface and are used explicitly to control specialized hardware devices. For example, many imaging systems that reside in special hardware platforms, such as virtual reality, multimedia, and medical imaging, are embedded.

An important concept in real-time systems is the notion of an **event**, that is, any occurrence that results in a change in the sequential flow of program execution. Events can be divided into two categories: synchronous and asynchronous. **Synchronous events** are those that occur at predictable times such as execution of a conditional branch instruction or hardware trap. **Asynchronous events** occur at unpredictable points in the flow-of-control and are usually caused by external sources such as a clock signal. Both types of events can be signaled to the CPU by hardware signals.

There is an inherent delay between when an interrupt occurs and when the CPU begins reacting to it called the **interrupt latency**. Interrupt latency is due to both hardware and software factors. Interrupts may occur periodically (at fixed rates), aperiodically, or both. Tasks driven by interrupts that occur aperiodically are called sporadic tasks. Systems where interrupts occur only at fixed frequencies are called **fixed rate systems** and those with interrupts occurring sporadically are called **sporadic systems**. For example, many imaging systems involve updating a display at from 20 to 40 times per second. A fixed rate task of, say, 30 hertz might be assigned to perform the image update. On the other hand, a target acquisition algorithm may run only when a candidate target image is on hand.

Another characteristic of a robust real-time system is that it is deterministic. A system is said to be **deterministic** if for each possible state, and each set of inputs, a unique set of outputs and the next state of the system can be determined. In particular, a certain kind of determinism called **event determinism** means that the next state and outputs of the system are known for each set of inputs that trigger events. Thus, a system that is deterministic is event deterministic. While it would be difficult for a system to be deterministic only for those inputs that trigger events, this is plausible and so event determinism may not imply determinism. We are, however, only interested in pure deterministic systems. Finally, if in a deterministic system the response time for each set of outputs is known, then the system also exhibits **temporal determinism**. Each of these previous definitions of determinism implies that the system must have a finite number of states. This is a reasonable assumption to make in a digital computer system where all inputs are digitized to within a finite range. For any physical system there are certain states under which the system is considered to be "out of control" and the software controlling such a system must avoid these states. For example, in certain guidance systems for robots or aircraft, rapid rotation through a 180^o pitch angle can cause a physical loss of gyro control. The software must be able to foresee and prepare for this situation or risk losing control. One side benefit of designing deterministic systems is that one can guarantee that the system can respond at any time, and in the case of temporally deterministic systems, when they will respond. This reinforces the association of control with real-time systems.

1.2 Scheduling Issues

Although we are not concerned with scheduling issues in this tutorial, a few terms should be mentioned for future reference. Real-time operating systems need to provide for either multitasking or multiprocessing (or both). In **multitasking**, the operating system must provide sufficient functionality to allow multiple programs to run on a single processor so that the illusion of simultaneity is created. This functionality includes scheduling, intertask communication and synchronization, and memory management. In **multiprocessing** operating systems, more than one processor is available to provide for simultaneity. Although multitasking may take place within any given processor, the main challenges are in process assignment, interprocessor synchronization and communication, and memory management. We will discuss some of these issues shortly, and in subsequent chapters.

There are several kinds of single processor multitasking approaches. In **round-robin systems**, each task is assigned a fixed time quantum in which to execute. A clock is used to initiate an interrupt at a rate corresponding to the time quantum. Each task executes until it completes or its time quantum expires as indicated by the clock interrupt. When a task's time quantum expires, a snapshot of the machine must be saved so that the task can be resumed later. Such schemes are used when all tasks must be equitably scheduled.

A higher priority task is said to **preempt** a lower priority task if it interrupts the lower priority task, that is, a lower priority task is running when the higher priority task signals that it is about to begin. Such schemes are used when certain processes are more critical than others. For example, in avionics systems, an imaging process may be preempted to allow a weapons control process to run. As with the round-robin system, a snapshot of the machine must be saved so that the lower priority task can be resumed when the higher priority task has finished.

Systems that use preemption schemes instead of round-robin or first-come-first-serve scheduling are called **preemptive priority systems**. The priorities assigned to each interrupt are based on the urgency of the task associated with that interrupt. Preemptive priority schemes have the associated problem of hogging of resources by higher priority tasks. In this case, the lower priority tasks are said to be facing **starvation**. There are other, non-preemptive, priority scheduling schemes, but these are of less interest to us.

Prioritized interrupts can be either fixed priority or dynamic priority. **Fixed priority systems** are less flexible in that the task priorities cannot be changed once the system is implemented. **Dynamic priority systems** can allow the priorities of tasks to change during program execution – a feature that is particularly important in threat management systems. In a special class of fixed-rate preemptive priority interrupt driven systems called **rate-monotonic systems,** priorities are assigned so that the higher the execution frequency, the higher the priority. This scheme is common in embedded applications, particularly avionics systems.

Hybrid systems include those with interrupts occurring at both fixed rates and sporadically. The sporadic interrupts may represent a critical error that requires immediate attention and thus have highest priority. This type of system is also common in embedded applications. Another type of mixed system found in commercial operating systems is a combination of round-robin and preemptive systems. Here tasks of higher priority can always preempt those of lower priority; however, if two or more tasks of the same priority are ready to run, then they run in round-robin fashion.

A concept often used as a measurement of real-time system performance is **time-loading** or **CPU utilization,** which is a measure of the percentage of non-idle processing. A system is said to be **time-overloaded** if it is 100% or more time-loaded. Time-overloading occurs in interrupt driven systems when higher priority interrupt-driven tasks execute too frequently to allow lower priority tasks to finish on time. Systems that are time-overloaded are unstable and exhibit missed deadlines and unpredictable response times.

1.3 Real-Time Design Issues

Why study real-time systems? The design and implementation of real-time systems requires the careful consideration of a variety of issues, many of which we will address in subsequent pages. Among the tasks facing the real-time system designer are:

1. Selection of hardware and software and the appropriate mix needed for a cost-effective solution.

2. The decision to take advantage of a commercial real-time operating system or to design a special operating system.

3. Prediction and measurement of CPU utilization and achieving a safe but efficient level of utilization.

4. Selection of an appropriate software language for system development.

5. Maximizing system fault tolerance and reliability through careful design and rigorous testing.

Utilization%	Zone Type	Application Types
0 – 25	overkill	various
26 – 50	very safe	various
51 – 68	safe	various
69	theoretical limit	various
70 – 99	dangerous	embedded systems
100+	overload	stressed systems

Table 1.1: CPU Utilization Zones.

6. Design and administration of tests and selection of test and development equipment.

Addressing these issues for large or even modest projects can present a staggering task.

For example, consider the evaluation of CPU utilization. Table 1.1 shows some CPU utilization ranges and subjective assessments of them. Thus, while it might be desirable to underutilize a processor for the sake of future expansion, in the near term, the additional cost of the high-powered processor may not be justified. Utilization factors in the 26%–50% range are generally considered safe – the likelihood of missing deadlines for most systems is low (yes, even at very low utilization rates, deadlines can be missed). Arbitrarily, we designate the 51%–68% range as "safe," while approximately 70% is the theoretical limit for all preemptive priority systems. Beyond 70% there is a high risk of missing deadlines, and of course CPU utilization above 100% is potentially disastrous.

Table 1.2 lists some other problem issues in real-time system design along with possible solutions and their potential drawback. We list them here simply to show the scope of the real-time design problem – in this tutorial we are concerned primarily with the prediction and reduction of CPU utilization, and optimal performance. The references list sources that discuss other issues.

1.4 What Is Real-Time Image Processing?

Real-time image processing differs from "ordinary" image processing in that the logical correctness of the system requires not only correct but also timely

Problem	Solution(s)	Possible Drawback
System modeling and design	dataflow diagrams	cannot depict control flow
Suitable programming languages	Ada	poor and unpredictable performance
Kernel selection	commercial products	poor and unpredictable performance
Intertask communication	mailboxes, queues	degrade performance
Intertask synchronization	semaphores	deadlock
Memory management	dynamic allocation	fragmentation, degraded performance
Testing	test everything	not feasible

Table 1.2: Some real-time problems, possible solutions, and potential drawbacks.

outputs; that is, semantic validity entails not only functional correctness, but also deadline satisfaction. Because of its nature, there are both supports for and obstacles to real-time image processing. On the positive side, many imaging applications are well-suited for parallelization and hence faster, parallel architectures. Furthermore, many imaging applications can be constructed without using language constructs that destroy determinism. Moreover, special real-time imaging architectures are available or can theoretically be constructed.

On the down side, many imaging applications are time critical and are computationally intensive or data intensive. And as will be discussed, there are no standard programming languages available for real-time image processing. Finally, real-time processing science itself is still struggling to produce usable results, especially for parallel processing machines. To illustrate some of these issues, we now characterize two real-time image processing systems.

Multimedia

Multimedia computing generally involves microcomputer systems equipped with high-resolution graphics, CD-ROM drives, mice, high-performance sound cards, and multitasking operating systems that support these devices. Commercial applications for multimedia computing are largely found in education, sales, and marketing.

Multimedia applications involve concurrent programs and processors, shared peripherals, and the important notion that synchronization is at least as important as timeliness. For example in multimedia applications, it is clear that audio speech output must be synchronized with the image of a person speaking (this problem, we argue, can in fact be addressed in the hardware, operating system, and language implementation). Other, possibly more difficult real-time synchronization problems exist, however. Two notable instances are video dithering and compression.

Video dithering is used to extend the available color subset by careful arrangement/rearrangement of available colors at high speed. For example, point dithering operations, which act on a pixel without regard to its neighbors, add appropriate noise to the pixel color value before it is quantized. Cluster dithering adds patterns of noise via masks to neighborhoods of pixels before quantization. In either case, if high-speed motion video is being supported, the dithering rates need to be properly synchronized with respect to lighting, video hardware performance, and visual perception cues. Work on synchronized dithering algorithms is still nonexistent.

A well-known example of dithering is the construction of composite colors from the basic red, green, blue colors (RGB) in television sets and earlier graphics displays. Here, judicious activation of the appropriate pixel colors during a preset time interval creates the perception of many more colors than just the basic three. Similarly, in black-and-white television sets, dithering has allowed for the perception of numerous shades of gray in addition to the basic white and black.

A second problem in multimedia systems involves real-time compression. For most multimedia systems, large amounts of data need to be stored and retrieved at fast rates. Often hardware boards are used, but frequently software algorithms are most cost-effective. In either case, most of the algorithms are proprietary. Of the well-known, nonproprietary algorithms, such as Huffman encoding or block truncation coding, the latter is deterministic while the former is not. An important consideration in constructing deterministic, predictable and synchronized multimedia systems is whether the proprietary compression algorithms are deterministic. In multimedia systems, we need to implement these types of compression algorithms on-the-fly and synchronously.

Virtual Reality

Virtual reality systems are complex computer simulations involving visual, audio, tactile, and other feedback to entice a person's perceptual mechanisms into believing they are actually in an artificial world. While virtual reality has obvious applications in combat simulation and training, its most promising applications are civilian, including exercise and recreation, physical rehabilitation and therapy, occupational training, and psychological diagnosis training.

For example, instead of pedaling an exercise bicycle inside an ordinary gym, a user could don a head-mounted display with head-tracking, plug in earphones, and have the impression of riding down a country road, replete with chirping birds, bumpy roads, and sun in his or her eyes. A physical therapist, wearing the same helmet-type display, could see computer-generated muscles, bones, and tendons superimposed on the body of a patient, changing with bodily movement. Construction workers learning to build skyscrapers can safely practice techniques in a virtual reality simulator. Psychologists can diagnose and/or cure patients of such phobias as vertigo, claustrophobia, and so forth, without exposing the patients to actual physical danger.

There are many other applications for virtual reality in medicine, entertainment, and so forth, and recent books, movies, and television programs have popularized this technology. These versions of the technology, however, ignore the substrate of this technology: underlying these applications are complex distributed computer control systems involving state-of-the-art electronics, software, and algorithms, and requiring sophisticated design techniques to ensure efficacy and reliability. In virtual reality, just as in multimedia, synchronization-type problems exist. For example, in virtual reality-type flight simulators, even a slight skew in the synchronization of a pilot's commands and the resultant display update (e.g., a turn is made) can cause nausea. Similarly, huge amounts of data need to be stored and retrieved quickly to simulate artificial environments. Hence real-time compression problems are of great interest to virtual reality specialists.