thread, the competing thread necessarily has less cache space available and thus suffers performance degradation. The objective of contention-aware scheduling is to allocate threads to cores in such a way as to maximize the effectiveness of the shared cache memory, and therefore to minimize the need for off-chip memory accesses. The design of algorithms for this purpose is an area of ongoing research and a subject of some complexity. Accordingly, this area is beyond our scope; see [ZHUR12] for a recent survey.

10.2 REAL-TIME SCHEDULING

Background

Real-time computing is becoming an increasingly important discipline. The operating system, and in particular the scheduler, is perhaps the most important component of a real-time system. Examples of current applications of real-time systems include control of laboratory experiments, process control in industrial plants, robotics, air traffic control, telecommunications, and military command and control systems. Next-generation systems will include the autonomous land rover, controllers of robots with elastic joints, systems found in intelligent manufacturing, the space station, and undersea exploration.

Real-time computing may be defined as that type of computing in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced. We can define a real-time system by defining what is meant by a real-time process, or task.[3] In general, in a real-time system, some of the tasks are real-time tasks, and these have a certain degree of urgency to them. Such tasks are attempting to control or react to events that take place in the outside world. Because these events occur in "real time," a real-time task must be able to keep up with the events with which it is concerned. Thus, it is usually possible to associate a deadline with a particular task, where the deadline specifies either a start time or a completion time. Such a task may be classified as hard or soft. A **hard real-time task** is one that must meet its deadline; otherwise it will cause unacceptable damage or a fatal error to the system. A **soft real-time task** has an associated deadline that is desirable but not mandatory; it still makes sense to schedule and complete the task even if it has passed its deadline.

Another characteristic of real-time tasks is whether they are periodic or aperiodic. An **aperiodic task** has a deadline by which it must finish or start, or it may have a constraint on both start and finish time. In the case of a **periodic task**, the requirement may be stated as "once per period T" or "exactly T units apart."

[3]As usual, terminology poses a problem, because various words are used in the literature with varying meanings. It is common for a particular process to operate under real-time constraints of a repetitive nature. That is, the process lasts for a long time and, during that time, performs some repetitive function in response to real-time events. Let us, for this section, refer to an individual function as a task. Thus, the process can be viewed as progressing through a sequence of tasks. At any given time, the process is engaged in a single task, and it is the process/task that must be scheduled.

Characteristics of Real-Time Operating Systems

Real-time operating systems can be characterized as having unique requirements in five general areas [MORG92]:

1. Determinism
2. Responsiveness
3. User control
4. Reliability
5. Fail-soft operation

An operating system is **deterministic** to the extent that it performs operations at fixed, predetermined times or within predetermined time intervals. When multiple processes are competing for resources and processor time, no system will be fully deterministic. In a real-time operating system, process requests for service are dictated by external events and timings. The extent to which an operating system can deterministically satisfy requests depends first on the speed with which it can respond to interrupts and, second, on whether the system has sufficient capacity to handle all requests within the required time.

One useful measure of the ability of an operating system to function deterministically is the maximum delay from the arrival of a high-priority device interrupt to when servicing begins. In non-real-time operating systems, this delay may be in the range of tens to hundreds of milliseconds, while in real-time operating systems that delay may have an upper bound of anywhere from a few microseconds to a millisecond.

A related but distinct characteristic is **responsiveness**. Determinism is concerned with how long an operating system delays before acknowledging an interrupt. Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt. Aspects of responsiveness include the following:

1. The amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR). If execution of the ISR requires a process switch, then the delay will be longer than if the ISR can be executed within the context of the current process.
2. The amount of time required to perform the ISR. This generally is dependent on the hardware platform.
3. The effect of interrupt nesting. If an ISR can be interrupted by the arrival of another interrupt, then the service will be delayed.

Determinism and responsiveness together make up the response time to external events. Response time requirements are critical for real-time systems, because such systems must meet timing requirements imposed by individuals, devices, and data flows external to the system.

User control is generally much broader in a real-time operating system than in ordinary operating systems. In a typical non-real-time operating system, the user either has no control over the scheduling function of the operating system, or can only provide broad guidance, such as grouping users into more than one priority

class. In a real-time system, however, it is essential to allow the user fine-grained control over task priority. The user should be able to distinguish between hard and soft tasks and to specify relative priorities within each class. A real-time system may also allow the user to specify such characteristics as the use of paging or process swapping, what processes must always be resident in main memory, what disk transfer algorithms are to be used, what rights the processes in various priority bands have, and so on.

Reliability is typically far more important for real-time systems than non-real-time systems. A transient failure in a non-real-time system may be solved by simply rebooting the system. A processor failure in a multiprocessor non-real-time system may result in a reduced level of service until the failed processor is repaired or replaced. But a real-time system is responding to and controlling events in real time. Loss or degradation of performance may have catastrophic consequences, ranging from financial loss to major equipment damage and even loss of life.

As in other areas, the difference between a real-time and a non-real-time operating system is one of degree. Even a real-time system must be designed to respond to various failure modes. **Fail-soft operation** is a characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible. For example, a typical traditional UNIX system, when it detects a corruption of data within the kernel, issues a failure message on the system console, dumps the memory contents to disk for later failure analysis, and terminates execution of the system. In contrast, a real-time system will attempt either to correct the problem or minimize its effects while continuing to run. Typically, the system notifies a user or user process that it should attempt corrective action then continues operation perhaps at a reduced level of service. In the event a shutdown is necessary, an attempt is made to maintain file and data consistency.

An important aspect of fail-soft operation is referred to as stability. A real-time system is stable if, in cases where it is impossible to meet all task deadlines, the system will meet the deadlines of its most critical, highest-priority tasks, even if some less critical task deadlines are not always met.

Although there is a wide variety of real-time OS designs to meet the wide variety of real-time applications, the following features are common to most real-time OSs:

- A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
- Interrupt latency (the amount of time between when a device generates an interrupt and when that device is serviced) is bounded and relatively short
- More precise and predictable timing characteristics than general purpose OSs

The heart of a real-time system is the short-term task scheduler. In designing such a scheduler, fairness and minimizing average response time are not paramount. What is important is that all hard real-time tasks complete (or start) by their deadline and that as many as possible soft real-time tasks also complete (or start) by their deadline.

Most contemporary real-time operating systems are unable to deal directly with deadlines. Instead, they are designed to be as responsive as possible to real-time

tasks so when a deadline approaches, a task can be quickly scheduled. From this point of view, real-time applications typically require deterministic response times in the several-millisecond to submillisecond span under a broad set of conditions; leading-edge applications (in simulators for military aircraft, for example) often have constraints in the range of 10–100 μs [ATLA89].

Figure 10.4 illustrates a spectrum of possibilities. In a preemptive scheduler that uses simple round-robin scheduling, a real-time task would be added to the ready queue to await its next timeslice, as illustrated in Figure 10.4a. In this case, the scheduling time will generally be unacceptable for real-time applications. Alternatively, in a nonpreemptive scheduler, we could use a priority scheduling mechanism, giving real-time tasks higher priority. In this case, a real-time task that is ready would be scheduled as soon as the current process blocks or runs to completion (see Figure 10.4b). This could lead to a delay of several seconds if a slow, low-priority task were executing at a critical time. Again, this approach is not acceptable. A more promising approach is to combine priorities with clock-based interrupts. Preemption points occur at regular intervals. When a preemption point occurs, the currently running task is preempted if a higher-priority task is waiting. This would include the preemption of tasks that are part of the operating system kernel. Such a delay may be on the order of several milliseconds (see Figure 10.4c). While this last approach may be adequate for some real-time applications, it will not suffice for more demanding applications. In those cases, the approach that has been taken is sometimes referred to as immediate preemption. In this case, the operating system responds to an interrupt almost immediately, unless the system is in a critical-code lockout section. Scheduling delays for a real-time task can then be reduced to 100 μs or less.

Real-Time Scheduling

Real-time scheduling is one of the most active areas of research in computer science. In this subsection, we provide an overview of the various approaches to real-time scheduling and look at two popular classes of scheduling algorithms.

In a survey of real-time scheduling algorithms, [RAMA94] observes that the various scheduling approaches depend on (1) whether a system performs schedulability analysis, (2) if it does, whether it is done statically or dynamically, and (3) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run time. Based on these considerations, the authors identify the following classes of algorithms:

- **Static table-driven approaches:** These perform a static analysis of feasible schedules of dispatching. The result of the analysis is a schedule that determines, at run time, when a task must begin execution.
- **Static priority-driven preemptive approaches:** Again, a static analysis is performed, but no schedule is drawn up. Rather, the analysis is used to assign priorities to tasks, so a traditional priority-driven preemptive scheduler can be used.
- **Dynamic planning-based approaches:** Feasibility is determined at run time (dynamically) rather than offline prior to the start of execution (statically).

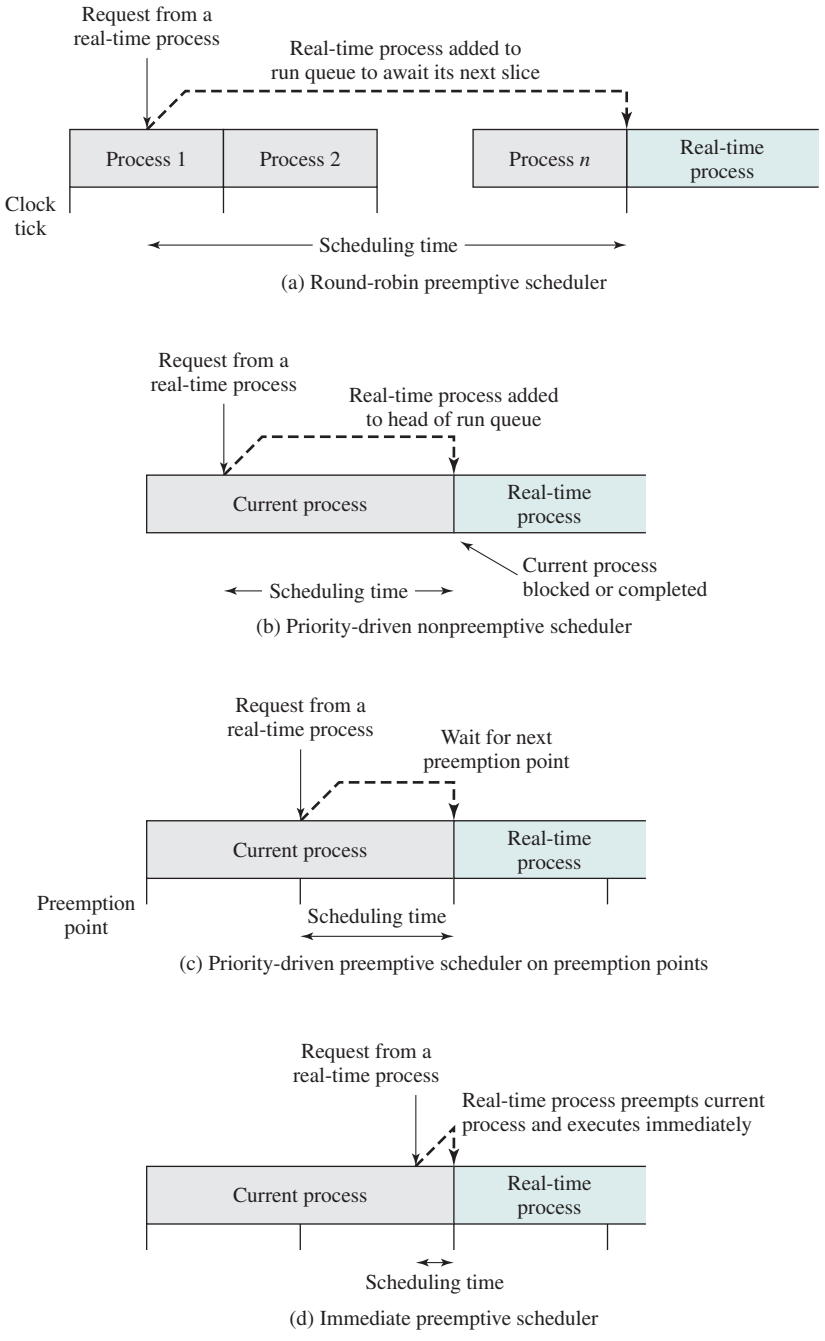


Figure 10.4 Scheduling of Real-Time Process

An arriving task is accepted for execution only if it is feasible to meet its time constraints. One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this task.

- **Dynamic best effort approaches:** No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.

Static table-driven scheduling is applicable to tasks that are periodic. Input to the analysis consists of the periodic arrival time, execution time, periodic ending deadline, and relative priority of each task. The scheduler attempts to develop a schedule that enables it to meet the requirements of all periodic tasks. This is a predictable approach but one that is inflexible, because any change to any task requirements requires that the schedule be redone. Earliest-deadline-first or other periodic deadline techniques (discussed subsequently) are typical of this category of scheduling algorithms.

Static priority-driven preemptive scheduling makes use of the priority-driven preemptive scheduling mechanism common to most non-real-time multiprogramming systems. In a non-real-time system, a variety of factors might be used to determine priority. For example, in a time-sharing system, the priority of a process changes depending on whether it is processor bound or I/O bound. In a real-time system, priority assignment is related to the time constraints associated with each task. One example of this approach is the rate monotonic algorithm (discussed subsequently), which assigns static priorities to tasks based on the length of their periods.

With **dynamic planning-based scheduling**, after a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival. If the new arrival can be scheduled in such a way that its deadlines are satisfied and that no currently scheduled task misses a deadline, then the schedule is revised to accommodate the new task.

Dynamic best effort scheduling is the approach used by many real-time systems that are currently commercially available. When a task arrives, the system assigns a priority based on the characteristics of the task. Some form of deadline scheduling, such as earliest-deadline scheduling, is typically used. Typically, the tasks are aperiodic, so no static scheduling analysis is possible. With this type of scheduling, until a deadline arrives or until the task completes, we do not know whether a timing constraint will be met. This is the major disadvantage of this form of scheduling. Its advantage is that it is easy to implement.

Deadline Scheduling

Most contemporary real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible, and hence emphasize rapid interrupt handling and task dispatching. In fact, this is not a particularly useful metric in evaluating real-time operating systems. Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times, neither too early nor too late, despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults. It follows that priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time.

There have been a number of proposals for more powerful and appropriate approaches to real-time task scheduling. All of these are based on having additional information about each task. In its most general form, the following information about each task might be used:

- **Ready time:** Time at which task becomes ready for execution. In the case of a repetitive or periodic task, this is actually a sequence of times that is known in advance. In the case of an aperiodic task, this time may be known in advance, or the operating system may only be aware when the task is actually ready.
- **Starting deadline:** Time by which a task must begin
- **Completion deadline:** Time by which a task must be completed. The typical real-time application will either have starting deadlines or completion deadlines, but not both.
- **Processing time:** Time required to execute the task to completion. In some cases, this is supplied. In others, the operating system measures an exponential average (as defined in Chapter 9). For still other scheduling systems, this information is not used.
- **Resource requirements:** Set of resources (other than the processor) required by the task while it is executing
- **Priority:** Measures relative importance of the task. Hard real-time tasks may have an "absolute" priority, with the system failing if a deadline is missed. If the system is to continue to run no matter what, then both hard and soft real-time tasks may be assigned relative priorities as a guide to the scheduler.
- **Subtask structure:** A task may be decomposed into a mandatory subtask and an optional subtask. Only the mandatory subtask possesses a hard deadline.

There are several dimensions to the real-time scheduling function when deadlines are taken into account: which task to schedule next and what sort of preemption is allowed. It can be shown, for a given preemption strategy and using either starting or completion deadlines, that a policy of scheduling the task with the earliest deadline minimizes the fraction of tasks that miss their deadlines [BUTT99, HONG89, PANW88]. This conclusion holds for both single-processor and multiprocessor configurations.

The other critical design issue is that of preemption. When starting deadlines are specified, then a nonpreemptive scheduler makes sense. In this case, it would be the responsibility of the real-time task to block itself after completing the mandatory or critical portion of its execution, allowing other real-time starting deadlines to be satisfied. This fits the pattern of Figure 10.4b. For a system with completion deadlines, a preemptive strategy (see Figure 10.4c or 10.4d) is most appropriate. For example, if task X is running and task Y is ready, there may be circumstances in which the only way to allow both X and Y to meet their completion deadlines is to preempt X, execute Y to completion, then resume X to completion.

As an example of scheduling periodic tasks with completion deadlines, consider a system that collects and processes data from two sensors, A and B. The deadline for collecting data from sensor A must be met every 20 ms, and that for B every 50 ms. It takes 10 ms, including operating system overhead, to process each sample of data from A and 25 ms to process each sample of data from B. Table 10.3 summarizes

Table 10.3 Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•

the execution profile of the two tasks. Figure 10.5 compares three scheduling techniques using the execution profile of Table 10.3. The first row of Figure 10.6 repeats the information in Table 10.3; the remaining three rows illustrate three scheduling techniques.

The computer is capable of making a scheduling decision every 10 ms.[4] Suppose under these circumstances, we attempted to use a priority scheduling scheme. The first two timing diagrams in Figure 10.5 show the result. If A has higher priority, the first instance of task B is given only 20 ms of processing time, in two 10-ms chunks, by the time its deadline is reached, and thus fails. If B is given higher priority, then A will miss its first deadline. The final timing diagram shows the use of earliest-deadline scheduling. At time $t = 0$, both A1 and B1 arrive. Because A1 has the earliest deadline, it is scheduled first. When A1 completes, B1 is given the processor. At $t = 20$, A2 arrives. Because A2 has an earlier deadline than B1, B1 is interrupted so A2 can execute to completion. Then B1 is resumed at $t = 30$. At $t = 40$, A3 arrives. However, B1 has an earlier ending deadline and is allowed to execute to completion at $t = 45$. A3 is then given the processor and finishes at $t = 55$.

In this example, by scheduling to give priority at any preemption point to the task with the nearest deadline, all system requirements can be met. Because the tasks are periodic and predictable, a static table-driven scheduling approach is used.

Now consider a scheme for dealing with aperiodic tasks with starting deadlines. The top part of Figure 10.6 shows the arrival times and starting deadlines for an example consisting of five tasks, each of which has an execution time of 20 ms. Table 10.4 summarizes the execution profile of the five tasks.

[4]This need not be on a 10-ms boundary if more than 10 ms has elapsed since the last scheduling decision.

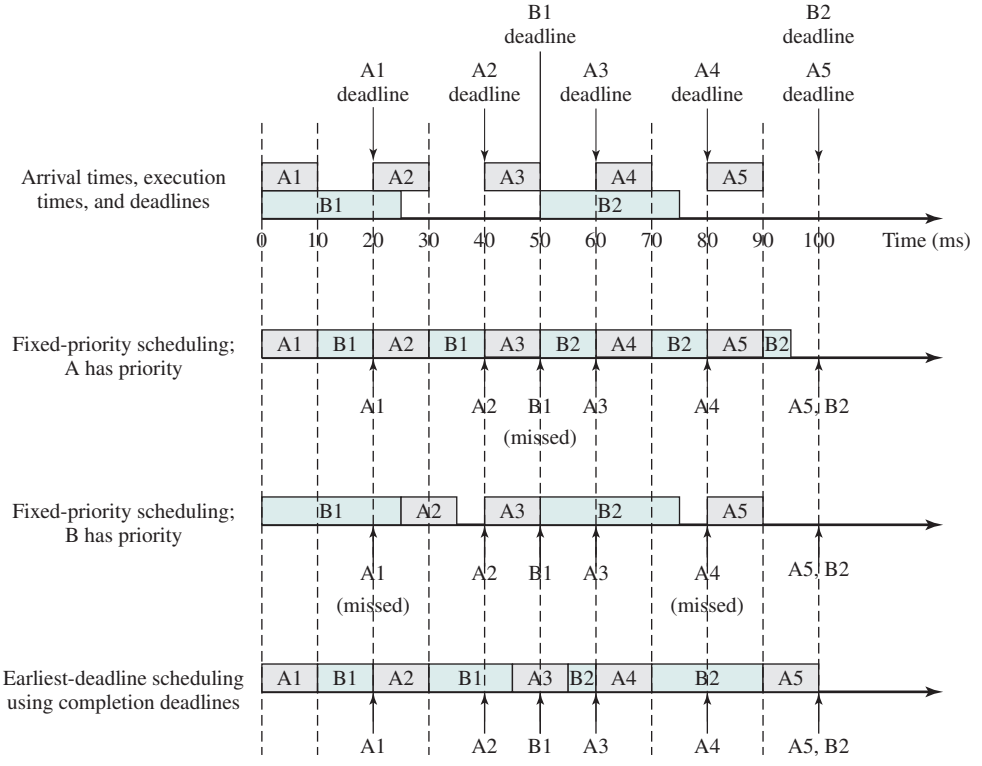


Figure 10.5 **Scheduling of Periodic Real-Time Tasks with Completion Deadlines (Based on Table 10.3)**

A straightforward scheme is to always schedule the ready task with the earliest deadline and let that task run to completion. When this approach is used in the example of Figure 10.6, note although task B requires immediate service, the service is denied. This is the risk in dealing with aperiodic tasks, especially with starting deadlines. A refinement of the policy will improve performance if deadlines can be known in advance of the time that a task is ready. This policy, referred to as earliest deadline with unforced idle times, operates as follows: Always schedule the eligible task with the earliest deadline and let that task run to completion. An eligible task may not be ready, and this may result in the processor remaining idle even though there are ready tasks. Note in our example the system refrains from scheduling task A even though that is the only ready task. The result is, even though the processor is not used to maximum efficiency, all scheduling requirements are met. Finally, for comparison, the FCFS policy is shown. In this case, tasks B and E do not meet their deadlines.

Rate Monotonic Scheduling

One of the more promising methods of resolving multitask scheduling conflicts for periodic tasks is rate monotonic scheduling (RMS) [LIU73, BRIA99, SHA94]. RMS assigns priorities to tasks on the basis of their periods.

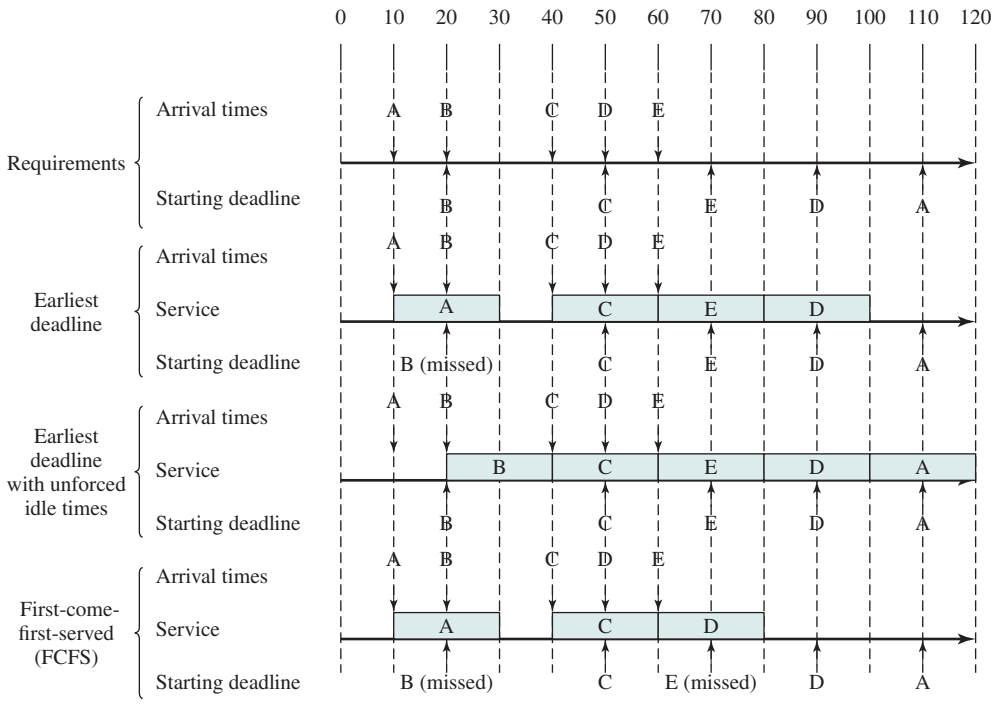


Figure 10.6 **Scheduling of Aperiodic Real-Time Tasks with Starting Deadlines**

For RMS, the highest-priority task is the one with the shortest period, the second highest-priority task is the one with the second shortest period, and so on. When more than one task is available for execution, the one with the shortest period is serviced first. If we plot the priority of tasks as a function of their rate, the result is a monotonically increasing function, hence the name "rate monotonic scheduling."

Figure 10.7 illustrates the relevant parameters for periodic tasks. The task's period, T, is the amount of time between the arrival of one instance of the task and the arrival of the next instance of the task. A task's rate (in hertz) is simply the inverse of its period (in seconds). For example, a task with a period of 50 ms occurs at a rate of 20 Hz. Typically, the end of a task's period is also the task's hard deadline, although some tasks may have earlier deadlines. The execution (or computation) time, C, is the amount of processing time required for each occurrence of the task. It

Table 10.4 Execution Profile of Five Aperiodic Tasks

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

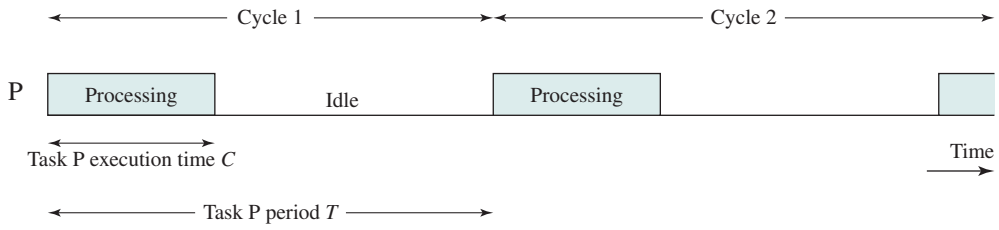
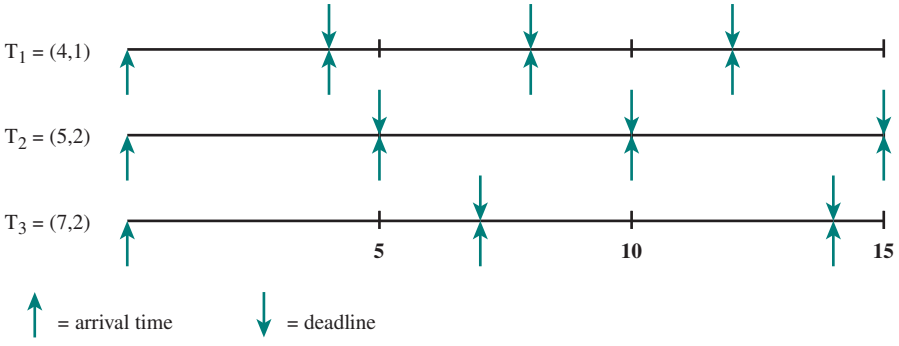
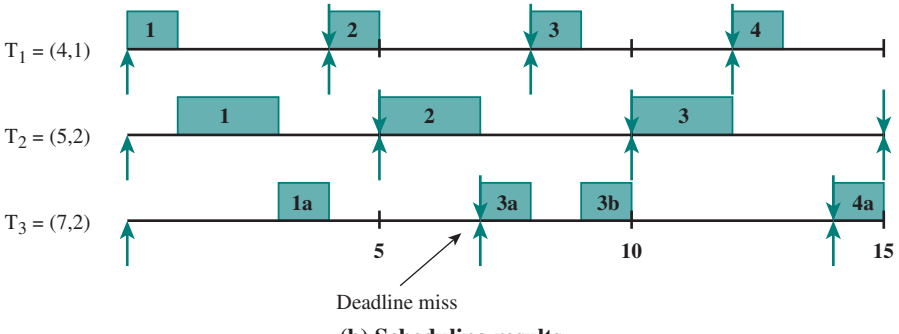


Figure 10.7 Periodic Task Timing Diagram

should be clear that in a uniprocessor system, the execution time must be no greater than the period (must have $C \leq T$). If a periodic task is always run to completion, that is, if no instance of the task is ever denied service because of insufficient resources, then the utilization of the processor by this task is $U = C/T$. For example, if a task has a period of 80 ms and an execution time of 55 ms, its processor utilization is 55/80 = 0.6875. Figure 10.8 is a simple example of RMS. Task instances are numbered sequentially within tasks. As can be seen, for task 3, the second instance is not executed because the deadline is missed. The third instance experiences a preemption but is still able to complete before the deadline.



(a) Arrival times and deadines for task $T_i = (P_i, C_i)$; P_i = period, C_i = processing time



(b) Scheduling results

Figure 10.8 Rate Monotonic Scheduling Example

One measure of the effectiveness of a periodic scheduling algorithm is whether or not it guarantees that all hard deadlines are met. Suppose we have n tasks, each with a fixed period and execution time. Then for it to be possible to meet all deadlines, the following inequality must hold:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ \ldots \ + \frac{C_n}{T_n} \leq 1 \tag{10.1}$$

The sum of the processor utilizations of the individual tasks cannot exceed a value of 1, which corresponds to total utilization of the processor. Equation (10.1) provides a bound on the number of tasks that a perfect scheduling algorithm can successfully schedule. For any particular algorithm, the bound may be lower. For RMS, it can be shown that the following inequality holds:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ \ldots \ + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \tag{10.2}$$

Table 10.5 gives some values for this upper bound. As the number of tasks increases, the scheduling bound converges to $\ln 2 \approx 0.693$.

As an example, consider the case of three periodic tasks, where $U_i = C_i/T_i$:

- **Task P_1:** $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
- **Task P_2:** $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$
- **Task P_3:** $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

The total utilization of these three tasks is $0.2 + 0.267 + 0.286 = 0.753$. The upper bound for the schedulability of these three tasks using RMS is

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \leq n(2^{1/3} - 1) = 0.779$$

Because the total utilization required for the three tasks is less than the upper bound for RMS ($0.753 < 0.779$), we know if RMS is used, all tasks will be successfully scheduled.

Table 10.5 Value of the RMS Upper Bound

n	$n(2^{1/n} - 1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
•	•
•	•
•	•
∞	$\ln 2 \approx 0.693$

It can also be shown that the upper bound of Equation (10.1) holds for earliest-deadline scheduling. Thus, it is possible to achieve greater overall processor utilization and therefore accommodate more periodic tasks with earliest-deadline scheduling. Nevertheless, RMS has been widely adopted for use in industrial applications. [SHA91] offers the following explanation:

1. The performance difference is small in practice. The upper bound of Equation (10.2) is a conservative one and, in practice, utilization as high as 90% is often achieved.
2. Most hard real-time systems also have soft real-time components, such as certain noncritical displays and built-in self tests that can execute at lower-priority levels to absorb the processor time that is not used with RMS scheduling of hard real-time tasks.
3. Stability is easier to achieve with RMS. When a system cannot meet all deadlines because of overload or transient errors, the deadlines of essential tasks need to be guaranteed provided that this subset of tasks is schedulable. In a static priority assignment approach, one only needs to ensure that essential tasks have relatively high priorities. This can be done in RMS by structuring essential tasks to have short periods or by modifying the RMS priorities to account for essential tasks. With earliest-deadline scheduling, a periodic task's priority changes from one period to another. This makes it more difficult to ensure that essential tasks meet their deadlines.

Priority Inversion

Priority inversion is a phenomenon that can occur in any priority-based preemptive scheduling scheme, but is particularly relevant in the context of real-time scheduling. The best-known instance of priority inversion involved the Mars Pathfinder mission. This rover robot landed on Mars on July 4, 1997, and began gathering and transmitting voluminous data back to Earth. But a few days into the mission, the lander software began experiencing total system resets, each resulting in losses of data. After much effort by the Jet Propulsion Laboratory (JPL) team that built the Pathfinder, the problem was traced to priority inversion [JONE97].

In any priority scheduling scheme, the system should always be executing the task with the highest priority. **Priority inversion** occurs when circumstances within the system force a higher-priority task to wait for a lower-priority task. A simple example of priority inversion occurs if a lower-priority task has locked a resource (such as a device or a binary semaphore) and a higher-priority task attempts to lock that same resource. The higher-priority task will be put in a blocked state until the resource is available. If the lower-priority task soon finishes with the resource and releases it, the higher-priority task may quickly resume and it is possible that no real-time constraints are violated.

A more serious condition is referred to as an **unbounded priority inversion**, in which the duration of a priority inversion depends not only on the time required to handle a shared resource but also on the unpredictable actions of other unrelated tasks. The priority inversion experienced in the Pathfinder software was unbounded and serves as a good example of the phenomenon. Our discussion follows that of

[TIME02]. The Pathfinder software included the following three tasks, in decreasing order of priority:

T_1: Periodically checks the health of the spacecraft systems and software

T_2: Processes image data

T_3: Performs an occasional test on equipment status

After T_1 executes, it reinitializes a timer to its maximum value. If this timer ever expires, it is assumed the integrity of the lander software has somehow been compromised. The processor is halted, all devices are reset, the software is completely reloaded, the spacecraft systems are tested, and the system starts over. This recovery sequence does not complete until the next day. T_1 and T_3 share a common data structure, protected by a binary semaphore `s`. Figure 10.9a shows the sequence that caused the priority inversion:

t_1: T_3 begins executing.

t_2: T_3 locks semaphore `s` and enters its critical section.

t_3: T_1, which has a higher priority than T_3, preempts T_3 and begins executing.

t_4: T_1 attempts to enter its critical section but is blocked because the semaphore is locked by T_3; T_3 resumes execution in its critical section.

t_5: T_2, which has a higher priority than T_3, preempts T_3 and begins executing.

t_6: T_2 is suspended for some reason unrelated to T_1 and T_3; T_3 resumes.

t_7: T_3 leaves its critical section and unlocks the semaphore. T_1 preempts T_3, locks the semaphore, and enters its critical section.

In this set of circumstances, T_1 must wait for both T_3 and T_2 to complete and fails to reset the timer before it expires.

In practical systems, two alternative approaches are used to avoid unbounded priority inversion: priority inheritance protocol and priority ceiling protocol.

The basic idea of **priority inheritance** is that a lower-priority task inherits the priority of any higher-priority task pending on a resource they share. This priority change takes place as soon as the higher-priority task blocks on the resource; it should end when the resource is released by the lower-priority task. Figure 10.9b shows that priority inheritance resolves the problem of unbounded priority inversion illustrated in Figure 10.9a. The relevant sequence of events is as follows:

t_1: T_3 begins executing.

t_2: T_3 locks semaphore `s` and enters its critical section.

t_3: T_1, which has a higher priority than T_3, preempts T_3 and begins executing.

t_4: T_1 attempts to enter its critical section but is blocked because the semaphore is locked by T_3. T_3 is immediately and temporarily assigned the same priority as T_1. T_3 resumes execution in its critical section.

t_5: T_2 is ready to execute, but because T_3 now has a higher priority, T_2 is unable to preempt T_3.

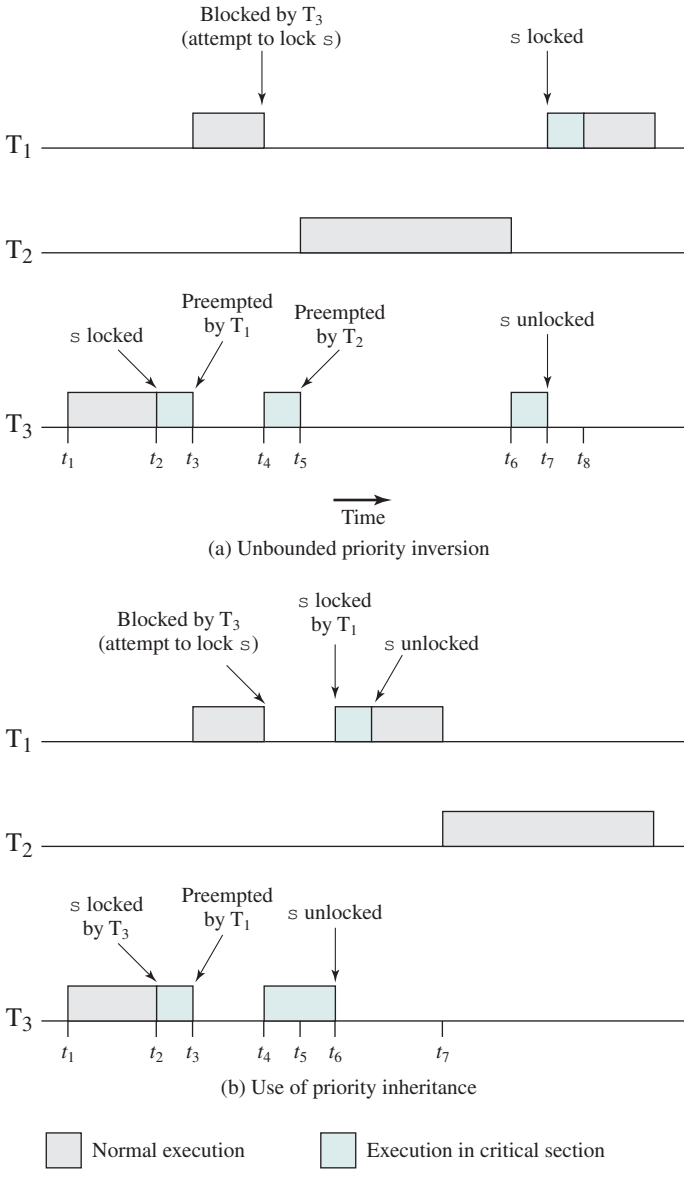


Figure 10.9 Priority Inversion

t_6: T_3 leaves its critical section and unlocks the semaphore: Its priority level is downgraded to its previous default level. T_1 preempts T_3, locks the semaphore, and enters its critical section.

t_7: T_1 is suspended for some reason unrelated to T_2, and T_2 begins executing.

This was the approach taken to solving the Pathfinder problem.

In the **priority ceiling** approach, a priority is associated with each resource. The priority assigned to a resource is one level higher than the priority of its highest-priority

PART 6 Embedded Systems

CHAPTER 13

EMBEDDED OPERATING SYSTEMS

Learning Objectives

After studying this chapter, you should be able to:

- Explain the concept of embedded system.
- Understand the characteristics of embedded operating systems.
- Explain the distinction between Linux and embedded Linux.
- Describe the architecture and key features of TinyOS.

In this chapter, we examine one of the most important and widely used categories of operating systems: embedded operating systems. The embedded system environment places unique and demanding requirements on the OS and calls for design strategies quite different than those found in ordinary operating systems.

We begin with an overview of the concept of embedded systems then turn to an examination of the principles of embedded operating systems. Finally, this chapter surveys two very different approaches to embedded OS design: embedded Linux and TinyOS. Appendix Q discusses eCos, another important embedded OS.

13.1 EMBEDDED SYSTEMS

This section introduces the concept of an embedded system. In doing so, we need to also explain the difference between a microprocessor and a microcontroller.

Embedded System Concepts

The term *embedded system* refers to the use of electronics and software within a product that has a specific function or set of functions, as opposed to a general-purpose computer, such as a laptop or desktop system. We can also define an embedded system as any device that includes a computer chip, but that is not a general-purpose workstation, or desktop or laptop computer. Hundreds of millions of computers are sold every year, including laptops, personal computers, workstations, servers, mainframes, and supercomputers. In contrast, tens of billions of microcontrollers are produced each year that are embedded within larger devices. Today, many, perhaps most devices that use electric power have an embedded computing system. It is likely in the near future, virtually all such devices will have embedded computing systems.

Types of devices with embedded systems are almost too numerous to list. Examples include cell phones, digital cameras, video cameras, calculators, microwave ovens, home security systems, washing machines, lighting systems, thermostats, printers, various automotive systems (e.g., transmission control, cruise control, fuel injection, anti-lock brakes, and suspension systems), tennis rackets, toothbrushes, and numerous types of sensors and actuators in automated systems.

Often, embedded systems are tightly coupled to their environment. This can give rise to real-time constraints imposed by the need to interact with the environment. Constraints, such as required speeds of motion, required precision of measurement,

and required time durations, dictate the timing of software operations. If multiple activities must be managed simultaneously, this imposes more complex real-time constraints.

Figure 13.1 shows in general terms an embedded system organization. In addition to the processor and memory, there are a number of elements that differ from the typical desktop or laptop computer:

- There may be a variety of interfaces that enable the system to measure, manipulate, and otherwise interact with the external environment. Embedded systems often interact (sense, manipulate, and communicate) with the external world through sensors and actuators, and hence are typically reactive systems; a reactive system is in continual interaction with the environment and executes at a pace determined by that environment.
- The human interface may be as simple as a flashing light or as complicated as real-time robotic vision. In many cases, there is no human interface.
- The diagnostic port may be used for diagnosing the system that is being controlled—not just for diagnosing the computer.
- Special-purpose field programmable (FPGA), application-specific (ASIC), or even nondigital hardware may be used to increase performance or reliability.
- Software often has a fixed function and is specific to the application.
- Efficiency is of paramount importance for embedded systems. These systems are optimized for energy, code size, execution time, weight and dimensions, and cost.

There are several noteworthy areas of similarity to general-purpose computer systems as well:

- Even with nominally fixed function software, the ability to field upgrade to fix bugs, to improve security, and to add functionality have become very important for embedded systems, and not just in consumer devices.

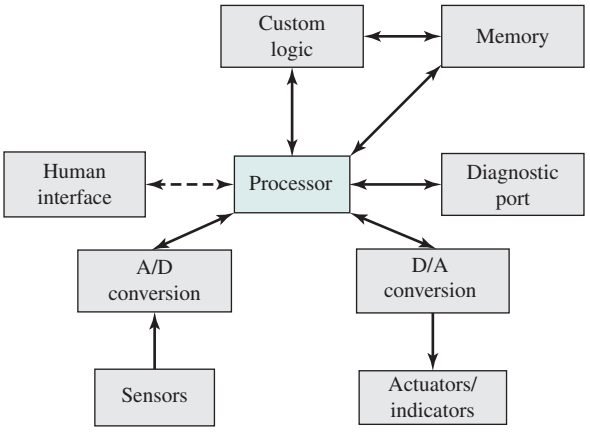


Figure 13.1 **Possible Organization of an Embedded System**

- One comparatively recent development has been of embedded system platforms that support a wide variety of apps. Good examples of this are smartphones and audio/visual devices, such as smart TVs.

Application Processors versus Dedicated Processors

Application processors are defined by the processor's ability to execute complex operating systems, such as Linux, Android, and Chrome. Thus, the application processor is general purpose in nature. A good example of the use of an embedded application processor is the smartphone. The embedded system is designed to support numerous apps and perform a wide variety of functions.

Most embedded systems employ a **dedicated processor**, which, as the name implies, is dedicated to one or a small number of specific tasks required by the host device. Because such an embedded system is dedicated to a specific task or tasks, the processor and associated components can be engineered to reduce size and cost.

Microprocessors

A microprocessor is a processor whose elements have been miniaturized into one or a few integrated circuits. Early microprocessor chips included registers, an arithmetic logic unit (ALU), and some sort of control unit or instruction processing logic. As transistor density increased, it became possible to increase the complexity of the instruction set architecture, and ultimately to add memory and more than one processor. Contemporary microprocessor chips include multiple processors, called cores, and a substantial amount of cache memory. However, as shown in Figure 13.2, a microprocessor chip includes only some of the elements that make up a computer system.

Most computers, including embedded computers in smartphones and tablets, as well as personal computers, laptops, and workstations, are housed on a motherboard. Before describing this arrangement, we need to define some terms. A **printed circuit board** (PCB) is a rigid, flat board that holds and interconnects chips and other electronic components. The board is made of layers, typically two to ten, that interconnect components via copper pathways that are etched into the board. The main PCB in a computer is called a system board or **motherboard**, while smaller ones that plug into the slots in the main board are called expansion boards.

The most prominent elements on the motherboard are the chips. A **chip** is a single piece of semiconducting material, typically silicon, upon which electronic circuits and logic gates are fabricated. The resulting product is referred to as an **integrated circuit**.

The motherboard contains a slot or socket for the processor chip, which typically contains multiple individual cores, in what is known as a *multicore processor*. There are also slots for memory chips, I/O controller chips, and other key computer components. For desktop computers, expansion slots enable the inclusion of more components on expansion boards. Thus, a modern motherboard connects only a few individual chip components, with each chip containing from a few thousand up to hundreds of millions of transistors.

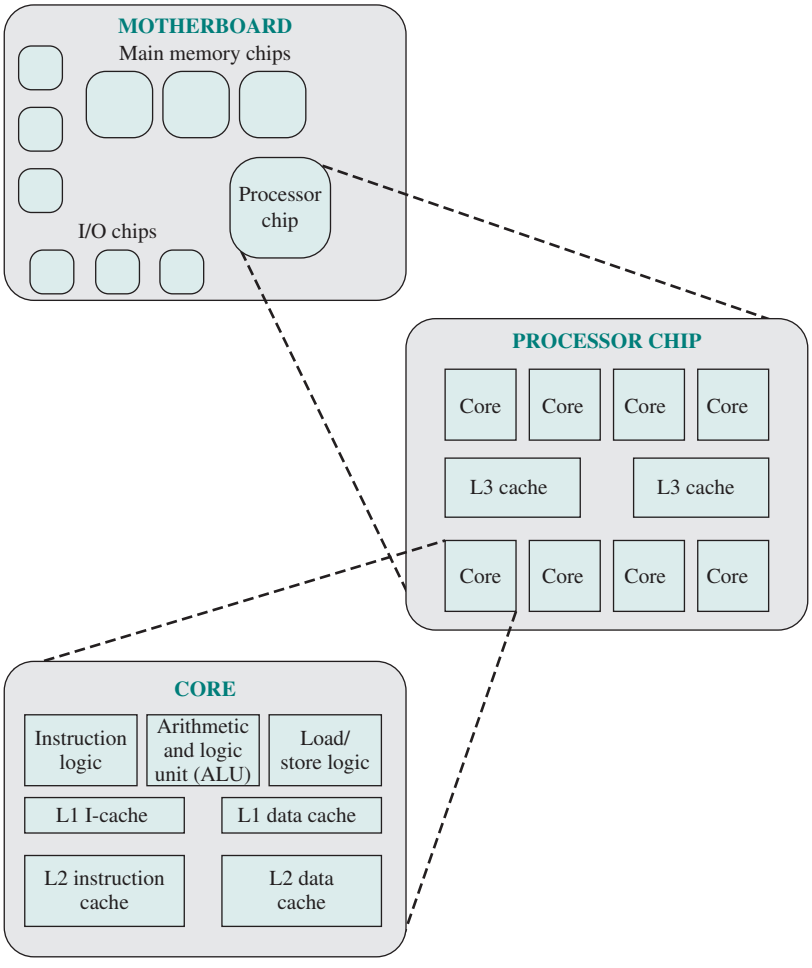


Figure 13.2 **Simplified View of Major Elements of a Multicore computer**

Microcontrollers

A **microcontroller** is a single chip that contains the processor, nonvolatile memory for the program (ROM or flash), volatile memory for input and output (RAM), a clock, and an I/O control unit. It is also called a "computer on a chip." A microcontroller chip makes a substantially different use of the logic space available. Figure 13.3 shows in general terms the elements typically found on a microcontroller chip. The processor portion of the microcontroller has a much lower silicon area than other microprocessors and much higher energy efficiency.

Billions of microcontroller units are embedded each year in myriad products from toys to appliances to automobiles. For example, a single vehicle can use 70 or more microcontrollers. Typically, especially for the smaller, less-expensive microcontrollers, they are used as dedicated processors for specific tasks. For example,

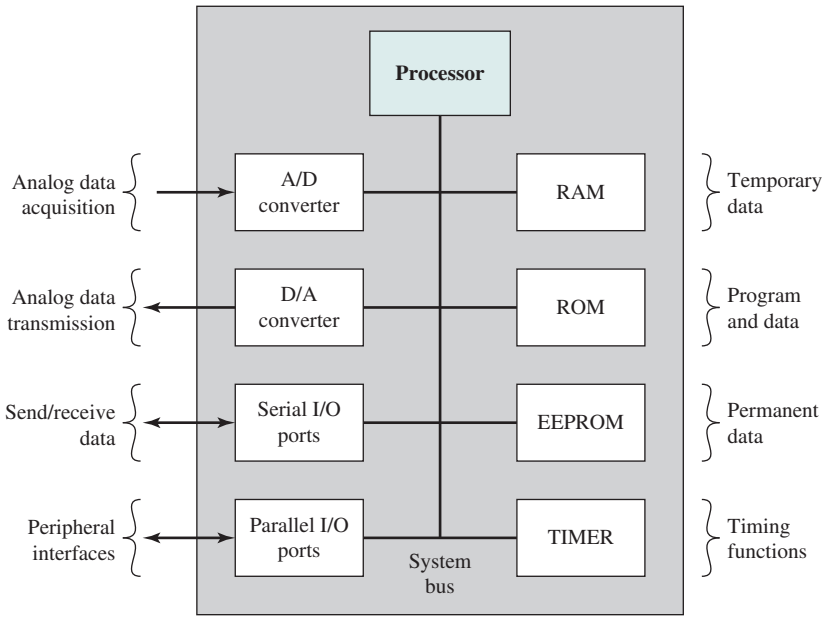


Figure 13.3 Typical Microcontroller Chip Elements

microcontrollers are heavily utilized in automation processes. By providing simple reactions to input, they can control machinery, turn fans on and off, open and close valves, and so forth. They are integral parts of modern industrial technology and are among the most inexpensive ways to produce machinery that can handle extremely complex functionalities.

Microcontrollers come in a range of physical sizes and processing power. Processors range from 4-bit to 32-bit architectures. Microcontrollers tend to be much slower than microprocessors, typically operating in the MHz range rather than the GHz speeds of microprocessors. Another typical feature of a microcontroller is that it does not provide for human interaction. The microcontroller is programmed for a specific task, embedded in its device, and executes as and when required.

Deeply Embedded Systems

A large percentage of the total number of embedded systems are referred to as **deeply embedded systems**. Although this term is widely used in the technical and commercial literature, you will search the Internet in vain (at least the writer did) for a straightforward definition. Generally, we can say a deeply embedded system has a processor whose behavior is difficult to observe both by the programmer and the user. A deeply embedded system uses a microcontroller rather than a microprocessor, is not programmable once the program logic for the device has been burned into ROM (read-only memory), and has no interaction with a user.

Deeply embedded systems are dedicated, single-purpose devices that detect something in the environment, perform a basic level of processing, then do something with the results. Deeply embedded systems often have wireless capability

and appear in networked configurations, such as networks of sensors deployed over a large area (e.g., factory, agricultural field). The Internet of Things depends heavily on deeply embedded systems. Typically, deeply embedded systems have extreme resource constraints in terms of memory, processor size, time, and power consumption.

13.2 CHARACTERISTICS OF EMBEDDED OPERATING SYSTEMS

A simple embedded system, with simple functionality, may be controlled by a special-purpose program or set of programs with no other software. Typically, more complex embedded systems include an OS. Although it is possible in principle to use a general-purpose OS (such as Linux) for an embedded system, constraints of memory space, power consumption, and real-time requirements typically dictate the use of a special-purpose OS designed for the embedded system environment.

The following are some of the unique characteristics and design requirements for embedded operating systems:

- **Real-time operation:** In many embedded systems, the correctness of a computation depends, in part, on the time at which it is delivered. Often, real-time constraints are dictated by external I/O and control stability requirements.
- **Reactive operation:** Embedded software may execute in response to external events. If these events do not occur periodically or at predictable intervals, the embedded software may need to take into account worst-case conditions and set priorities for execution of routines.
- **Configurability:** Because of the large variety of embedded systems, there is a large variation in the requirements, both qualitative and quantitative, for embedded OS functionality. Thus, an embedded OS intended for use on a variety of embedded systems must lend itself to flexible configuration so only the functionality needed for a specific application and hardware suite is provided. [MARW06] gives the following examples: The linking and loading functions can be used to select only the necessary OS modules to load. Conditional compilation can be used. If an object-oriented structure is used, proper subclasses can be defined. However, verification is a potential problem for designs with a large number of derived tailored operating systems. Takada cites this as a potential problem for eCos [TAKA01].
- **I/O device flexibility:** There is virtually no device that needs to be supported by all versions of the OS, and the range of I/O devices is large. [MARW06] suggests that it makes sense to handle relatively slow devices (such as disks and network interfaces) by using special tasks instead of integrating their drives into the OS kernel.
- **Streamlined protection mechanisms:** Embedded systems are typically designed for a limited, well-defined functionality. Untested programs are rarely added to the software. After the software has been configured and tested, it can be assumed to be reliable. Thus, apart from security measures, embedded systems have limited protection mechanisms. For example, I/O instructions need

not be privileged instructions that trap to the OS; tasks can directly perform their own I/O. Similarly, memory protection mechanisms can be minimized. [MARW06] provides the following example: Let `switch` correspond to the memory-mapped I/O address of a value that needs to be checked as part of an I/O operation. We can allow the I/O program to use an instruction such as `load register, switch` to determine the current value. This approach is preferable to the use of an OS service call, which would generate overhead for saving and restoring the task context.

- **Direct use of interrupts:** General-purpose operating systems typically do not permit any user process to use interrupts directly. [MARW06] lists three reasons why it is possible to let interrupts directly start or stop tasks (e.g., by storing the task's start address in the interrupt vector address table) rather than going through OS interrupt service routines: (1) Embedded systems can be considered to be thoroughly tested, with infrequent modifications to the OS or application code; (2) protection is not necessary, as discussed in the preceding bullet item; and (3) efficient control over a variety of devices is required.

Host and Target Environments

A key differentiator between desktop/server and embedded Linux distributions is that desktop and server software is typically compiled or configured on the platform where it will execute, while embedded Linux distributions are usually compiled or configured on one platform, called the host platform, but are intended to be executed on another, called the target platform (see Figure 13.4). The key elements that are developed on the host system and then transferred to the target system are the boot loader, the kernel, and the root file system.

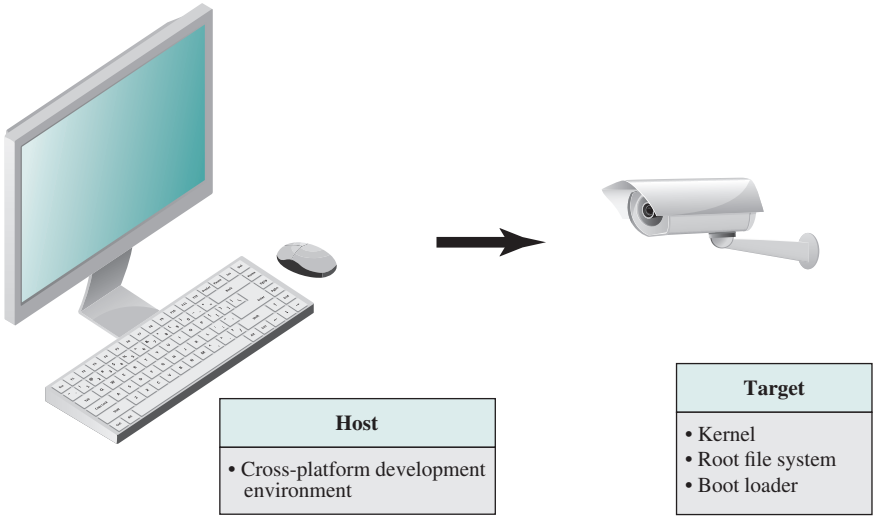


Figure 13.4 Host-Target Environment

BOOT LOADER The boot loader is a small program that calls the OS into memory (RAM) after the power is turned on. It is responsible for the initial boot process of the system, and for loading the kernel into main memory. A typical sequence in an embedded system is the following:

1. The processor in the embedded system executes code in ROM to load a first-stage boot loader from internal flash memory, a Secure Digital (SD) card, or a serial I/O port.
2. The first-stage boot loader initializes the memory controller and a few peripherals and loads a second-stage boot loader into RAM. No interaction is possible with this boot loader, and it is typically provided by the processor vendor on ROM.
3. The second-stage boot loader loads the kernel and root file system from flash memory to main memory (RAM). The kernel and the root file system are generally stored in flash memory in compressed files, so part of the book loading process is to decompress the files into binary images of the kernel and root file system. The boot loader then passes control to the kernel. Typically, an open-source boot loader is used for the second stage.

KERNEL The full kernel includes a number of separate modules, including:

- Memory management.
- Process/thread management.
- Inter process communication, timers.
- Device drivers for I/O, network, sound, storage, graphics, etc.
- File systems.
- Networking.
- Power management.

From the full kernel software for a given OS, a number of optional components will be left out for an embedded system. For example, if the embedded system hardware does not support paging, then the memory management subsystem can be eliminated. The full kernel will include multiple file systems, device drivers, and so on, and only a few of these may be needed.

A key differentiator between desktop/server and embedded Linux distributions is that desktop and server software is typically compiled on the platform where it will execute, while embedded Linux distributions are usually compiled on one platform but are intended to be executed on another. The software used for this purpose is referred to as a *cross-compiler*. Figure 13.5 illustrates its use.

ROOT FILE SYSTEM In an embedded OS, or any OS, a global single hierarchy of directories and files is used to represent all the files in the system. At the top, or root of this hierarchy is the root file system, which contains all the files needed for the system to work properly. The root file system of an embedded OS is similar to that found on a workstation or server, except that it contains only the minimal set of applications, libraries, and related files needed to run the system.

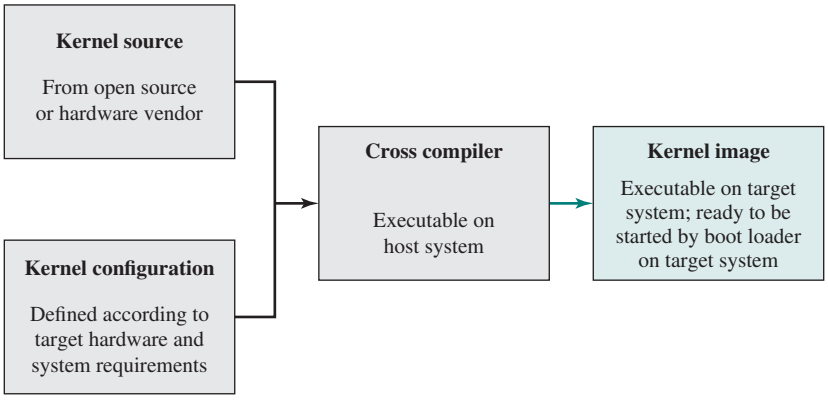


Figure 13.5 **Kernel Compilation**

Development Approaches

There are two general approaches to developing an embedded OS. The first approach is to take an existing OS and adapt it for the embedded application. The other approach is to design and implement an OS intended solely for embedded use.

Adapting an Existing Commercial Operating System

An existing commercial OS can be used for an embedded system by adding real-time capability, streamlining operation, and adding necessary functionality. This approach typically makes use of Linux, but FreeBSD, Windows, and other general-purpose operating systems have also been used. Such operating systems are typically slower and less predictable than a special-purpose embedded OS. An advantage of this approach is that the embedded OS derived from a commercial general-purpose OS is based on a set of familiar interfaces, which facilitates portability.

The disadvantage of using a general-purpose OS is that it is not optimized for real-time and embedded applications. Thus, considerable modification may be required to achieve adequate performance. In particular, a typical OS optimizes for the average case rather than the worst case for scheduling, usually assigns resources on demand, and ignores most if not all semantic information about an application.

Purpose-Built Embedded Operating System

A significant number of operating systems have been designed from the ground up for embedded applications. Two prominent examples of this latter approach are eCos and TinyOS, both of which will be discussed later in this chapter.

Typical characteristics of a specialized embedded OS include the following:

- Has a fast and lightweight process or thread switch
- Scheduling policy is real time and dispatcher module is part of scheduler instead of separate component.
- Has a small size

- Responds to external interrupts quickly; typical requirement is response time of less than 10 μs.
- Minimizes intervals during which interrupts are disabled
- Provides fixed or variable-sized partitions for memory management as well as the ability to lock code and data in memory
- Provides special sequential files that can accumulate data at a fast rate

To deal with timing constraints, the kernel:

- Provides bounded execution time for most primitives.
- Maintains a real-time clock.
- Provides for special alarms and time-outs.
- Supports real-time queuing disciplines such as earliest deadline first and primitives for jamming a message into the front of a queue.
- Provides primitives to delay processing by a fixed amount of time and to suspend/resume execution.

The characteristics just listed are common in embedded operating systems with real-time requirements. However, for complex embedded systems, the requirement may emphasize predictable operation over fast operation, necessitating different design decisions, particularly in the area of task scheduling.

13.3 EMBEDDED LINUX

The term *embedded Linux* simply means a version of Linux running in an embedded system. Typically, an embedded Linux system uses one of the official kernel releases, although some systems use a modified kernel tailored to a specific hardware configuration or to support a certain class of applications. Primarily, an embedded Linux kernel differs from a Linux kernel used on a workstation or server by the build configuration and development framework.

In this section, we highlight some of the key differences between embedded Linux and a version of Linux running on a desktop or server, then examine a popular software offering, μClinux.

Characteristics of an Embedded Linux System

KERNEL SIZE Desktop and server Linux systems need to support a large number of devices because of the wide variety of configurations that use Linux. Similarly, such systems also need to support a range of communication and data exchange protocols so they can be used for a large number of different purposes. Embedded devices typically require support for a specific set of devices, peripherals, and protocols, depending on the hardware that is present in a given device and the intended purpose of that device. Fortunately, the Linux kernel is highly configurable in terms of the architecture for which it is compiled and the processors and devices that it supports.

An embedded Linux distribution is a version of Linux to be customized for the size and hardware constraints of embedded devices, and includes software packages that support a variety of services and applications on those devices. Thus, an embedded Linux kernel will be far smaller than an ordinary Linux kernel.

Memory Size [ETUT16] classifies the size of an embedded Linux system by the amount of available ROM and RAM, using the three broad categories of small, medium, and large. Small systems are characterized by a low-powered processor with a minimum of 2 MB of ROM and 4 MB of RAM. Medium-sized systems are characterized by a medium-powered processor with around 32 MB of ROM and 64 MB of RAM. Large systems are characterized by a powerful processor or collection of processors combined with large amounts of RAM and permanent storage.

On a system without permanent storage, the entire Linux kernel must fit in the RAM and ROM. A full-featured modern Linux system would not do so. As an indication of this, Figure 13.6 shows the compressed size of the full Linux kernel as it has grown over time. Of course, any Linux system will be configured with only some of the components of the full release. Even so, this chart gives an indication of the fact that substantial amounts of the kernel must be left out, especially for small and medium-sized embedded systems.

Other Characteristics Other characteristics of embedded Linux systems include:

- **Time constraints:** Stringent time constraints require the system to respond in a specified time period. Mild time constraints are appropriate for systems in which timely response is not critical.

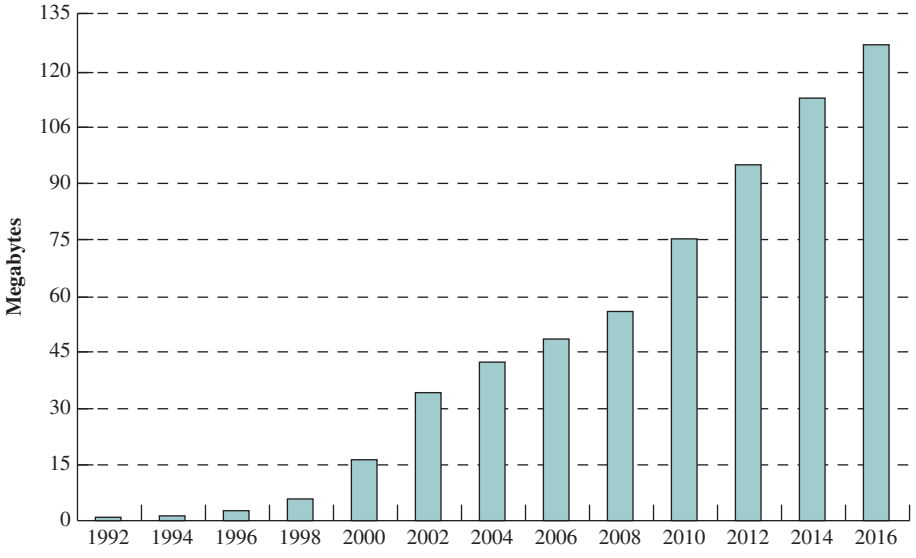


Figure 13.6 Size of Linux Kernel (shown in GZIP-compressed file size)

- **Networkability:** Networkability refers to whether a system can be connected to a network. Virtually all embedded devices today have this capability, typically wireless.
- **Degree of user interaction:** Some devices are centered on user interaction, such as smartphones. Other devices, such as industrial process control, might provide a very simple interface, such as LEDs and buttons for interaction. And other devices have no end user interaction, such as IoT sensors that gather information and transmit them to a cloud.

Table 13.1, from [ETUT16], gives characteristics of some commercially available embedded systems using a Linux kernel.

Embedded Linux File Systems

Some applications may create relatively small file systems to be used only for the duration of the application and which can be stored in main memory. But in general, a file system must be stored in persistent memory, such as flash memory or traditional disk-based storage devices. For most embedded systems, an internal or external disk is not an option, and persistent storage is generally provided by flash memory.

As with other aspects of an embedded Linux system, the file system must be as small as possible. A number of such compact file systems have been designed for use in embedded systems. The following are commonly used examples:

- **cramfs:** The Compressed RAM file system is a simple read-only file system designed to minimize size by maximizing the efficient use of underlying storage. Files on cramfs file systems are compressed in units that match the Linux page size (typically 4096 bytes or 4 MB, based on kernel version and configuration) to provide efficient random access to file contents.

Table 13.1 Characteristics of Example Embedded Linux Systems

Description	Type	Size	Time Constraints	Networkability	Degree of User Interaction
Accelerator control devices	Industrial process control	Medium	Stringent	Yes	Low
Computer-aided training system	Aerospace	Large	Stringent	No	High
Bluetooth device for accessing local information	Networking	Small	Mild	Yes	Very low
System control and data acquisition protocol converter	Industrial process control	Medium	Stringent	No	Very low
Personal digital assistant	Consumer electronics	Medium	Mild	Yes	Very high
Motor control device involved with space vehicle control	Aerospace	Large	Stringent	Yes	High

- **squashfs:** Like cramfs, squashfs is a compressed, read-only file system that was designed for use on low memory or limited storage size environments such as embedded Linux systems.
- **jffs2:** The Journaling Flash File System, version 2, is a log-based file system that, as the name suggests, is designed for use on NOR and NAND flash devices with special attention to flash-oriented issues such as wear leveling.
- **ubifs:** The Unsorted Block Image File System generally provides better performance than jffs2 on larger flash devices, and also supports write caching to provide additional performance improvements.
- **yaffs2:** Yet another Flash File System, version 2, provides a fast and robust file system for large flash devices. yaffs2 requires less RAM to hold file system state information than files systems such as jffs2, and also generally provides better performance if the file system is being written too frequently.

Advantages of Embedded Linux

Embedded versions of Linux began to appear as early as 1999. A number of companies have developed their own versions tailored to specific platforms. Advantages of using Linux as the basis for an embedded OS include:

- **Vendor independence:** The platform provider is not dependent on a particular vendor to provide needed features and meet deadlines for deployment.
- **Varied hardware support:** Linux support for a wide range of processor architectures and peripheral devices makes it suitable for virtually any embedded system.
- **Low cost:** The use of Linux minimizes cost for development and training.
- **Open source:** The use of Linux provides all of the advantages of open-source software.

μClinux

μClinux (microcontroller Linux) is a popular open-source Linux kernel variation targeted at microcontrollers and other very small embedded systems. Because of the modular nature of Linux, it is easy to slim down the operating environment by removing utility programs, tools, and other system services that are not needed in an embedded environment. This is the design philosophy for μClinux.

To get some feel for the size of a μClinux bootable image (kernel plus root file system), we look at the experience of EmCraft Systems, which builds board-level systems using Cortex-M microcontrollers and Cortex-A microprocessors [EMCR15]. These are by no means the smallest embedded systems that use μClinux. A minimal configuration could be as little as 0.5 MB, but the vendor found the size of a practical bootable image, with Ethernet, TCP/IP and a reasonable set of user space tools and applications configured, would be in the range of 1.5 to 2 MB. The size of RAM required for run-time μClinux operation would be in the range of 8 to 32 MB. These numbers are dramatically smaller than those of a typical Linux system.

Comparison with Full Linux Key differences between μClinux and Linux for larger systems include the following (see [MCCU04] for a further discussion):

- Linux is a multiuser OS based on UNIX. μClinix is a version of Linux intended for embedded systems typically with no interactive user.
- Unlike Linux, μClinix does not support memory management. Thus, with μClinix there are no virtual address spaces; applications must be linked to absolute addresses.
- The Linux kernel maintains a separate virtual address space for each process. μClinix has a single shared address space for all processes.
- In Linux, address space is recovered on context switching; this is not done in μClinix.
- Unlike Linux, μClinix does not provide the fork system call; the only option is to use vfork. The fork call essentially makes a duplicate of the calling process, identical in almost every way (not everything is copied over, for example, resource limits in some implementations, but the idea is to create as close a copy as possible). The new process (child) gets a different process ID (PID) and has the PID of the old process (parent) as its parent PID (PPID). The basic difference between vfork and fork is that when a new process is created with vfork, the parent process is temporarily suspended, and the child process might borrow the parent's address space. This continues until the child process either exits, or calls execve, at which point the parent process continues.
- μClinix modifies device drivers to use the local system bus rather than ISA or PCI.

The most significant difference between full Linux and μClinux is in the area of memory management. The lack of memory management support in μClinux has a number of implications, including:

- The main memory allocated to a process must generally be contiguous. If a number of processes swap in and out of memory, this can lead to fragmentation (see Figure 7.4). However, embedded systems typically have a fixed set of processes that are loaded at boot up time and continue until the next reset, so this feature is generally not needed.
- μClinux cannot expand memory for running process, because there may be other processes contiguous to it. Thus, the brk and sbrk calls (dynamically change the amount of space allocated for the data segment of the calling process) are not available. But μClinux does provide an implementation of malloc, which is used to allocate a block of memory from a global memory pool.
- μClinux lacks a dynamic application stack. This can result in a stack overflow, which will corrupt memory. Care must be taken in application development and configuration to avoid this.
- μClinux does not provide memory protection, which presents the risk of an application corrupting part of another application or even the kernel. Some implementations do provide a fix for this. For example, the Cortex-M3/M4 architecture provides a memory protection mechanism called MPU (Memory

Table 13.2 Size of Some Functions in GNU C Library and μClibc

glibc name	glibc size	μClibc name	μClibc size
libc-2.3.2.so	1.2M	libuClibc-0.9.2.7so	284K
ld-2.3.2.so	92K	libcrypt-0.9.2.7so	20K
libcrypt-2.3.2.so	20K	libdl-0.9.2.7so	12K
libdl-2.3.2.so	12K	libm-0.9.2.7so	8K
libm-2.3.2.so	136K	libnsl-0.9.2.7so	56K
libnsl-2.3.2.so	76K	libpthread-0.9.2.7so	4K
libpthread-2.3.2.so	84K	libresolv-0.9.2.7so	84K
libresolv-2.3.2.so	68K	libutil-0.9.2.7so	4K
libutil-2.3.2.so	8K	libcrypt-0.9.2.7so	8K

Protection Unit). Using the MPU, Emcraft Systems has added to the kernel an optional feature that implements process-to-process and process-to-kernel protection on par with the memory protection mechanisms implemented in Linux using MMU [KHUS12].

μCLIBC μClibc is a C system library originally developed to support μClinux and it generally used in conjunction with μClinux. However, μClibc can also be used with other Linux kernels. The main objective for μClibc is to provide a system library that is to provide a C library suitable for developing embedded Linux system. It is much smaller than the GNU C Library, which is widely used on Linux systems, but nearly all applications supported by glibc also work perfectly with μClibc. Porting applications from glibc to μClibc typically involves just recompiling the source code. μClibc even supports shared libraries and threading.

Table 13.2, based on [ANDE05], compares the sizes of functions in the two libraries. As can be seen, the space savings are considerable. These savings are achieved by disabling some features by default and aggressively rewriting the code to eliminate redundancy.

Figure 13.7 shows the top-level software architecture of an embedded system using μClinux and μClibc.

Android

As we have discussed throughout this book, Android is an embedded OS based on a Linux kernel. Thus, it is reasonable to consider Android an example of embedded Linux. However, many embedded Linux developers do not consider Android to be an instance of embedded Linux [CLAR13]. From the point of view of these developers, a classic embedded device has a fixed function, frozen at the factory. Android is more of a platform OS that can support a variety of applications that vary from one platform to the next. Further, Android is a vertically integrated system, including some Android-specific modifications to the Linux kernel. The focus of Android lies in

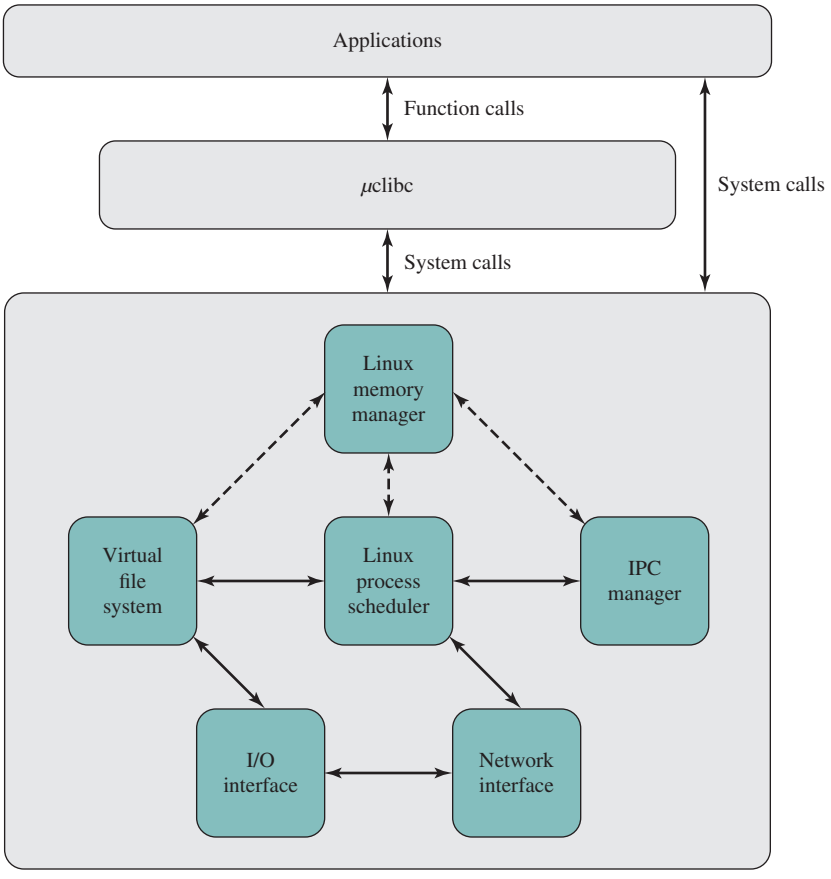


Figure 13.7 **μClinux/μClibc Software Architecture**

the vertical integration of the Linux kernel and the Android user space components. Ultimately, it is a matter of semantics, with no "official" definition of embedded Linux on which to rely.

13.4 TINYOS

TinyOS provides a more streamlined approach for an embedded OS than one based on a commercial general-purpose OS, such as an embedded version of Linux. Thus, TinyOS and similar systems are better suited for small embedded systems with tight requirements on memory, processing time, real-time response, power consumption, and so on. TinyOS takes the process of streamlining quite far, resulting in a very minimal OS for embedded systems. The core OS requires 400 bytes of code and data memory, combined.

TinyOS represents a significant departure from other embedded operating systems. One striking difference is that TinyOS is not a real-time OS. The reason for

this is the expected workload, which is in the context of a wireless sensor network, as described in the next subsection. Because of power consumption, these devices are off most of the time. Applications tend to be simple, with processor contention not much of an issue.

Additionally, in TinyOS there is no kernel, as there is no memory protection and it is a component-based OS; there are no processes; the OS itself does not have a memory allocation system (although some rarely used components do introduce one); interrupt and exception handling is dependent on the peripheral; and it is completely nonblocking, so there are few explicit synchronization primitives.

TinyOS has become a popular approach to implementing wireless sensor network software. Currently, over 500 organizations are developing and contributing to an open-source standard for Tiny OS.

Wireless Sensor Networks

TinyOS was developed primarily for use with networks of small wireless sensors. A number of trends have enabled the development of extremely compact, low-power sensors. The well-known Moore's law continues to drive down the size of memory and processing logic elements. Smaller size in turn reduces power consumption. Low power and small-size trends are also evident in wireless communications hardware, micro-electromechanical sensors (MEMS), and transducers. As a result, it is possible to develop an entire sensor complete with logic in a cubic millimeter. The application and system software must be compact enough that sensing, communication, and computation capabilities can be incorporated into a complete, but tiny, architecture.

Low–cost, small–size, low-power-consuming wireless sensors can be used in a host of applications [ROME04]. Figure 13.8 shows a typical configuration. A base station connects the sensor network to a host PC and passes on sensor data from the network to the host PC, which can do data analysis and/or transmit the data

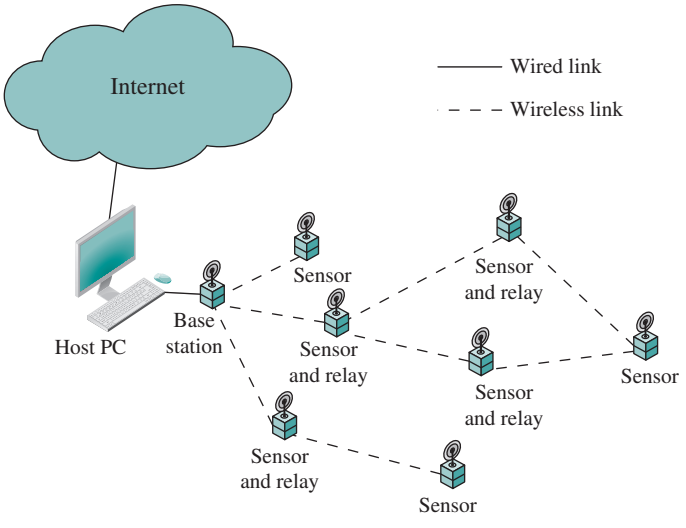


Figure 13.8 Typical Wireless Sensor Network Topology

over a corporate network or Internet to an analysis server. Individual sensors collect data and transmit these to the base station, either directly or through sensors that act as data relays. Routing functionality is needed to determine how to relay the data through the sensor network to the base station. [BUON01] points out that, in many applications, the user will want to be able to quickly deploy a large number of low-cost devices without having to configure or manage them. This means that they must be capable of assembling themselves into an ad hoc network. The mobility of individual sensors and the presence of RF interference means the network will have to be capable of reconfiguring itself in a matter of seconds.

TinyOS Goals

With the tiny, distributed sensor application in mind, a group of researchers from UC Berkeley [HILL00] set the following goals for TinyOS:

- **Allow high concurrency:** In a typical wireless sensor network application, the devices are concurrency intensive. Several different flows of data must be kept moving simultaneously. While sensor data are input in a steady stream, processed results must be transmitted in a steady stream. In addition, external controls from remote sensors or base stations must be managed.
- **Operate with limited resources:** The target platform for TinyOS will have limited memory and computational resources and run on batteries or solar power. A single platform may offer only kilobytes of program memory and hundreds of bytes of RAM. The software must make efficient use of the available processor and memory resources while enabling low-power communication.
- **Adapt to hardware evolution:** Most hardware is in constant evolution; applications and most system services must be portable across hardware generations. Thus, it should be possible to upgrade the hardware with little or no software change, if the functionality is the same.
- **Support a wide range of applications:** Applications exhibit a wide range of requirements in terms of lifetime, communication, sensing, and so on. A modular, general-purpose embedded OS is desired so a standardized approach leads to economies of scale in developing applications and support software.
- **Support a diverse set of platforms:** As with the preceding point, a general-purpose embedded OS is desirable.
- **Be robust:** Once deployed, a sensor network must run unattended for months or years. Ideally, there should be redundancy both within a single system and across the network of sensors. However, both types of redundancy require additional resources. One software characteristic that can improve robustness is to use highly modular, standardized software components.

It is worth elaborating on the concurrency requirement. In a typical application, there will be dozens, hundreds, or even thousands of sensors networked together. Usually, little buffering is done, because of latency issues. For example, if you are sampling every 5 minutes and want to buffer four samples before sending, the average latency is 10 minutes. Thus, information is typically captured, processed, and streamed onto the network in a continuous flow. Further, if the sensor sampling produces a

significant amount of data, the limited memory space available limits the number of samples that could be buffered. Even so, in some applications, each of the flows may involve a large number of low-level events interleaved with higher-level processing. Some of the high-level processing will extend over multiple real-time events. Further, sensors in a network, because of the low power of transmission available, typically operate over a short physical range. Thus data from outlying sensors must be relayed to one or more base stations by intermediate nodes.

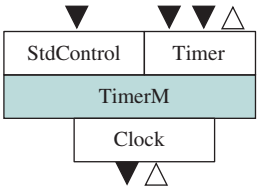
TinyOS Components

An embedded software system built using TinyOS consists of a set of small modules, called components, each of which performs a simple task or set of tasks and which interface with each other and with hardware in limited and well-defined ways. The only other software module is the scheduler, discussed subsequently. In fact, because there is no kernel, there is no actual OS. But we can take the following view. The application area of interest is the wireless sensor network (WSN). To meet the demanding software requirements of this application, a rigid, simplified software architecture is dictated, consisting of components. The TinyOS development community has implemented a number of open-source components that provide the basic functions needed for the WSN application. Examples of such standardized components include single-hop networking, ad hoc routing, power management, timers, and nonvolatile storage control. For specific configurations and applications, users build additional special-purpose components and link and load all of the components needed for the user's application. TinyOS, then, consists of a suite of standardized components. Some, but not all, of these components are used, together with application-specific user-written components, for any given implementation. The OS for that implementation is simply the set of standardized components from the TinyOS suite.

All components in a TinyOS configuration have the same structure, an example of which is shown in Figure 13.9a. The shaded box in the diagram indicates the component, which is treated as an object that can only be accessed by defined interfaces, indicated by white boxes. A component may be hardware or software. Software components are implemented in nesC, which is an extension of C with two distinguishing features: 1) a programming model where components interact via interfaces, and 2) an event-based concurrency model with run-to-completion task and interrupt handlers, explained subsequently.

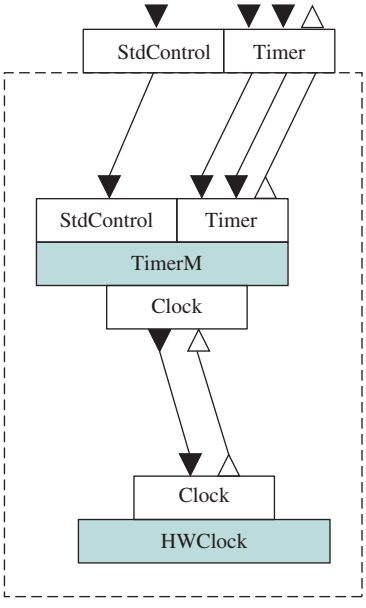
The architecture consists of a layered arrangement of components. Each component can link to only two other components, one below it in the hierarchy and one above it. A component issues commands to its lower-level component and receives event signals from it. Similarly, the component accepts commands from its upper-level component and issues event signals to it. At the bottom of the hierarchy are hardware components, and at the top of the hierarchy are application components, which may not be part of the standardized TinyOS suite but which must conform to the TinyOS component structure.

A software component implements one or more tasks. Each **task** in a component is similar to a thread in an ordinary OS, with certain limitations. Within a component, tasks are atomic: Once a task has started, it runs to completion. It cannot



(a) TimerM component

```
module TimerM {
   provides {
      interface StdControl;
      interface Timer;
   }
   uses interface Clock as Clk;
} ...
```



(b) TimerC configuration

```
configuration TimerC {
   provides {
      interface StdControl;
      interface Timer;
   }
}

implementation {
   components TimerM, HWClock;
   StdControl = TimerM.StdControl;
   Timer = TimerM.Timer;
   TimerM.Clk -> HWClock.Clock;
}
```

Figure 13.9 Example of Component and Configuration

be preempted by another task in the same component, and there is no time slicing. However, a task can be preempted by an event. A task cannot block or spin wait. These limitations greatly simplify the scheduling and management of tasks within a component. There is only a single stack, assigned to the currently running task. Tasks can perform computations, call lower-level components (commands) and signal higher-level events, and schedule other tasks.

Commands are nonblocking requests. That is, a task that issues a command does not block or spin wait for a reply from the lower-level component. A command is typically a request for the lower-level component to perform some service, such as initiating a sensor reading. The effect on the component that receives the command is specific to the command given and the task required to satisfy the command. Generally, when a command is received, a task is scheduled for later execution, because a command cannot preempt the currently running task. The command returns immediately to the calling component; at a later time, an event will signal completion to the calling component. Thus, a command does not cause a preemption in the called component, and does not cause blocking in the calling component.

Events in TinyOS may be tied either directly or indirectly to hardware events. The lowest-level software components interface directly to hardware interrupts, which may be external interrupts, timer events, or counter events. An event handler in a lowest-level component may handle the interrupt itself, or may propagate event messages up through the component hierarchy. A command can post a task that will signal an event in the future. In this case, there is no tie of any kind to a hardware event.

A task can be viewed as having three phases. A caller posts a command to a module. The module then runs the requested task. The module then notifies the caller, via an event, that the task is complete.

The component depicted in Figure 13.9a, TimerM, is part of the TinyOS timer service. This component **provides** the StdControl and Timer interface and **uses** a Clock interface. Providers implement commands (i.e., the logic in this component). Users implement events (i.e., external to the component). Many TinyOS components use the StdControl interface to be initialized, started, or stopped. TimerM provides the logic that maps from a hardware clock into TinyOS's timer abstraction. The timer abstraction can be used for counting down a given time interval. Figure 13.9a also shows the formal specification of the TimerM interfaces.

The interfaces associated with TimerM are specified as follows:

```
interface StdControl {
     command result_t init();
     command result_t start();
     command result_t stop();
}
interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}
interface Clock {
    command result_t setRate(char interval, char scale);
    event result_t fire();
}
```

Components are organized into configurations by "wiring" them together at their interfaces and equating the interfaces of the configuration with some of the interfaces of the components. A simple example is shown in Figure 13.9b. The uppercase C stands for Component. It is used to distinguish between an interface (e.g., Timer) and a component that provides the interface (e.g., TimerC). The uppercase M stands for Module. This naming convention is used when a single logical component has both a configuration and a module. The TimerC component, providing the Timer interface, is a configuration that links its implementation (TimerM) to Clock and LED providers. Otherwise, any user of TimerC would have to explicitly wire its subcomponents.

TinyOS Scheduler

The TinyOS scheduler operates across all components. Virtually all embedded systems using TinyOS will be uniprocessor systems, so only one task among all the tasks in all the components may execute at a time. The scheduler is a separate component. It is the one portion of TinyOS that must be present in any system.

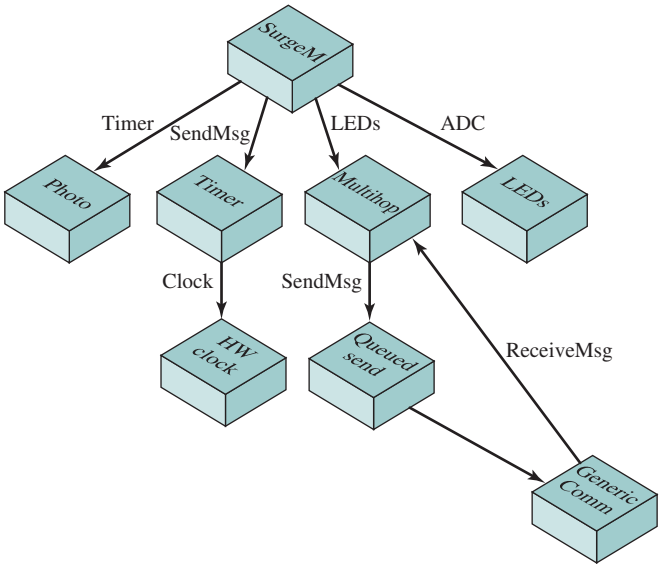
The default scheduler in TinyOS is a simple FIFO (first-in-first-out) queue. A task is posted to the scheduler (place in the queue) either as a result of an event, which triggers the posting, or as a result of a specific request by a running task to schedule another task. The scheduler is power aware. This means the scheduler puts the processor to sleep when there are no tasks in the queue. The peripherals remain operating, so one of them can wake up the system by means of a hardware event signaled to a lowest-level component. Once the queue is empty, another task can be scheduled only as a result of a direct hardware event. This behavior enables efficient battery usage.

The scheduler has gone through two generations. In TinyOS 1.x, there is a shared task queue for all tasks, and a component can post a task to the scheduler multiple times. If the task queue is full, the post operation fails. Experience with networking stacks showed this to be problematic, as the task might signal completion of a split-phase operation: If the post fails, the component above might block forever, waiting for the completion event. In TinyOS 2.x, every task has its own reserved slot in the task queue, and a task can only be posted once. A post fails if and only if the task has already been posted. If a component needs to post a task multiple times, it can set an internal state variable so that when the task executes, it reposts itself. This slight change in semantics greatly simplifies a lot of component code. Rather than test to see if a task is posted already before posting it, a component can just post the task. Components do not have to try to recover from failed posts and retry. The cost is one byte of state per task.

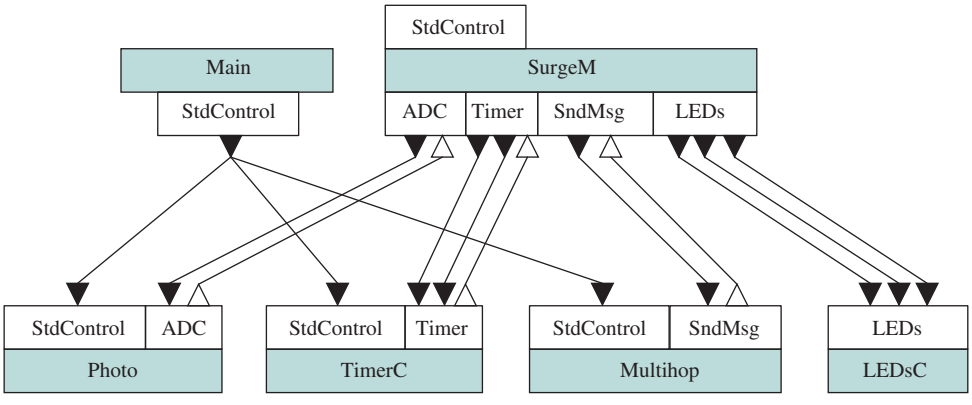
A user can replace the default scheduler with one that uses a different dispatching scheme, such as a priority-based scheme or a deadline scheme. However, preemption and time slicing should not be used because of the overhead such systems generate. More importantly, they violate the TinyOS concurrency model, which assumes tasks do not preempt each other.

Example of Configuration

Figure 13.10 shows a configuration assembled from software and hardware components. This simplified example, called Surge and described in [GAY03], performs periodic sensor sampling and uses ad hoc multihop routing over the wireless network to deliver samples to the base station. The upper part of the figure shows the components of Surge (represented by boxes) and the interfaces by which they are wired (represented by arrowed lines). The SurgeM component is the application-level component that orchestrates the operation of the configuration.



(a) Simplified view of the Surge application



(b) Top-level Surge configuration

LED = light-emitting diode
ADC = analog-to-digital converter

Figure 13.10 Examples of TinyOS Application

Figure 13.10b shows a portion of the configuration for the Surge application. The following is a simplified excerpt from the SurgeM specification.

```
module SurgeM {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface SendMsg;
    uses interface LEDs;
}
```

```
implementation {
     uint16_t sensorReading;
     command result_t StdControl.init() {
     return call Timer.start(TIMER_REPEAT, 1000);
     }
     event result_t Timer.fired() {
     call ADC.getData();
     return SUCCESS;
     }
     event result_t ADC.dataReady(uint16_t data) {
     sensorReading = data;
     ...send message with data in it...
     return SUCCESS;
     }
     ...
}
```

This example illustrates the strength of the TinyOS approach. The software is organized as an interconnected set of simple modules, each of which defines one or a few tasks. Components have simple, standardized interfaces to other components, be they hardware or software. Thus, components can easily be replaced. Components can be hardware or software, with a boundary change not visible to the application programmer.

TinyOS Resource Interface

TinyOS provides a simple but powerful set of conventions for dealing with resources. Three abstractions for resources are used in TinyOS:

1. **Dedicated:** A resource that a subsystem needs exclusive access to at all times. In this class of resources, no sharing policy is needed since only a single component ever requires use of the resource. Examples of dedicated abstractions include interrupts and counters.
2. **Virtualized:** Every client of a virtualized resource interacts with it as if it were a dedicated resource, with all virtualized instances being multiplexed on top of a single underlying resource. The virtualized abstraction may be used when the underlying resource need not be protected by mutual exclusion. An example is a clock or timer.
3. **Shared:** The shared resource abstraction provides access to a dedicated resource through an arbiter component. The arbiter enforces mutual exclusion, allowing only one user (called a client) at a time to have access to a resource and enabling the client to lock the resource.

In the remainder of this subsection, we briefly define the shared resource facility of TinyOS. The arbiter determines which client has access to the resource at which time. While a client holds a resource, it has complete and unfettered control. Arbiters assume clients are cooperative, only acquiring the resource when needed and holding on to it no longer than necessary. Clients explicitly release resources: There is no way for an arbiter to forcibly reclaim it.

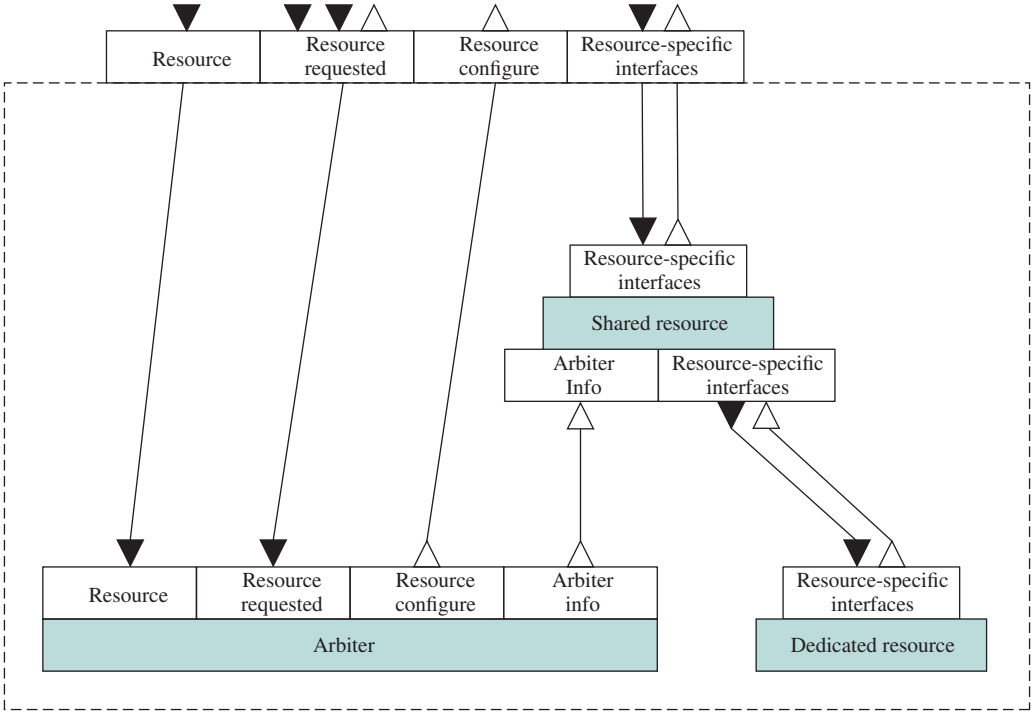


Figure 13.11 Shared Resource Configuration

Figure 13.11 shows a simplified view of the shared resource configuration used to provide access to an underlying resource. Associated with each resource to be shared is an arbiter component. The Arbiter enforces a policy that enables a client to lock the resource, use it, then release the resource. The shared resource configuration provides the following interfaces to a client:

- **Resource:** The client issues a request at this interface, requesting access to the resource. If the resource is currently locked, the arbiter places the request in a queue. When a client is finished with the resource, it issues a release command at this interface.
- **Resource requested:** This is similar to the Resource interface. In this case, the client is able to hold on to a resource until the client is notified that someone else needs the resource.
- **Resource Configure:** This interface allows a resource to be automatically configured just before a client is granted access to it. Components providing the Resource Configure interface use the interfaces provided by an underlying dedicated resource to configure it into one of its desired modes of operation.
- **Resource-specific interfaces:** Once a client has access to a resource, it uses resource-specific interfaces to exchange data and control information with the resource.

In addition to the dedicated resource, the shared resource configuration consists of two components. The Arbiter accepts requests for access and configuration from a

client and enforces the lock on the underlying resource. The shared resource component mediates data exchange between the client and the underlying resource. Arbiter information passed from the arbiter to the shared resource component controls the access of the client to the underlying resource.

13.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

application processors, chip	embedded operating system	microcontroller
commands	embedded system	motherboard
dedicated processor	events	printed circuit board
eCos	deeply embedded system	task
	integrated circuit	TinyOS

Review Questions

13.1. What is an embedded system?
13.2. What are some typical requirements or constraints on embedded systems?
13.3. What is an embedded OS?
13.4. What are some of the key characteristics of an embedded OS?
13.5. Explain the relative advantages and disadvantages of an embedded OS based on an existing commercial OS compared to a purpose-built embedded OS.
13.6. What is the target application for TinyOS?
13.7. What are the design goals for TinyOS?
13.8. What is a TinyOS component?
13.9. What software comprises the TinyOS operating system?
13.10. What is the default scheduling discipline for TinyOS?

Problems

13.1. In a particular sensor network that runs on TinyOS, the default FIFO algorithm has been replaced with a priority-based one that continually checks for high priority tasks before assigning it to schedulers. What constraints will this technique face?

13.2. **a.** The TinyOS Resource interface does not allow a component that already has a request in the queue for a resource to make a second request. Suggest a reason.

b. However, the TinyOS Resource interface allows a component holding the resource lock to re-request the lock. This request is queued for a later grant. Suggest a reason for this policy. *Hint:* What might cause there to be latency between one component releasing a lock and the next requester being granted it?

Note: The remaining problems concern eCos, discussed in Appendix Q.

13.3. With reference to the device driver interface to the eCos kernel (see Table Q.1), it is recommended that device drivers should use the `_intsave()` variants to claim and release spinlocks rather than the non-`_intsave()` variants. Explain why

13.4. Also in Table Q.1, it is recommended that `cyg_drv_spinlock_spin` should be used sparingly, and in situations where deadlocks/livelocks cannot occur. Explain why.

13.5. In Table Q.1, what should be the limitations on the use of `cyg_drv_spinlock_destroy`? Explain.

13.6. In Table Q.1, what limitations should be placed in the use of `cyg_drv_mutex_destroy`?

13.7. Why does the eCos bitmap scheduler not support time slicing?

13.8. The implementation of mutexes within the eCos kernel does not support recursive locks. If a thread has locked a mutex then attempts to lock the mutex again, typically as a result of some recursive call in a complicated call graph, then either an assertion failure will be reported or the thread will deadlock. Suggest a reason for this policy.

13.9. Figure 13.12 is a listing of code intended for use on the eCos kernel.

a. Explain the operation of the code. Assume thread B begins execution first, and thread A begins to execute after some event occurs.

b. What would happen if the mutex unlock and wait code execution in the call to `cyg_cond_wait`, on line 30, were not atomic?

c. Why is the while loop on line 26 needed?

13.10. The discussion of eCos spinlocks included an example showing why spinlocks should not be used on a uniprocessor system if two threads of different priorities can compete for the same spinlock. Explain why the problem still exists even if only threads of the same priority can claim the same spinlock.

```
1   unsigned char buffer_empty = true;
2   cyg_mutex_t mut_cond_var;
3   cyg_cond-t cond_var;
4
5   void thread_a( cyg_addrword_t index )
6   {
7      while ( 1 ) // run this thread forever
8      {
9         // acquire data into the buffer...
10
11        // there is data in the buffer now
12        buffer_empty = false;
13
14        cyg_mutex_lock( &mut_cond_var );
15
16        cyg_cond_signal( &cond_var );
17
18        cyg_mutex_unlock( &mut_cond_var );
19     }
20  }
21
22  void thread_b( cyg_addrword_t index )
23  {
24     while ( 1 ) // run this thread forever
25     {
26        cyg_mutex_lock( &mut_cond_var );
27
28        while ( buffer_empty == true )
29        {
30        cyg_cond_wait( &cond_var );
31        }
32
33
34        // get the buffer data...
35
36        // set flag to indicate the data in the buffer has been processed
37        buffer_empty = true;
38
39        cyg_mutex_unlock( &mut_cond_var );
40
41        // process the data in the buffer
42     }
43  {
```

VideoNote **Figure 13.12 Condition Variable Example Code**

CHAPTER 18

Distributed Processing, Client/Server, and Clusters

Learning Objectives

After studying this chapter, you should be able to:

- Present a summary of the key aspects of client/server computing.
- Understand the principle design issues for distributed message passing.
- Understand the principle design issues for remote procedure calls.
- Understand the principle design issues for clusters.
- Describe the cluster mechanisms in Windows 7 and Beowulf.

In this chapter, we begin with an examination of some of the key concepts in distributed software, including client/server architecture, message passing, and remote procedure calls. Then we examine the increasingly important cluster architecture.

Chapters 17 and 18 complete our discussion of distributed systems.

18.1 CLIENT/SERVER COMPUTING

The concept of client/server computing, and related concepts, has become increasingly important in information technology systems. This section begins with a description of the general nature of client/server computing. This is followed by a discussion of alternative ways of organizing the client/server functions. The issue of **file cache consistency**, raised by the use of file servers, is then examined. Finally, this section introduces the concept of middleware.

What Is Client/Server Computing?

As with other new waves in the computer field, client/server computing comes with its own set of jargon words. Table 18.1 lists some of the terms that are commonly found in descriptions of client/server products and applications.

Table 18.1 Client/Server Terminology

Applications Programming Interface (API) A set of function and call programs that allow clients and servers to intercommunicate.
Client A networked information requester, usually a PC or workstation, that can query a database and/or other information from a server.
Middleware A set of drivers, APIs, or other software that improves connectivity between a client application and a server.
Relational Database A database in which information access is limited to the selection of rows that satisfy all search criteria.
Server A computer, usually a high-powered workstation, a minicomputer, or a mainframe, that houses information for manipulation by networked clients.
Structured Query Language (SQL) A language developed by IBM and standardized by ANSI for addressing, creating, updating, or querying relational databases.

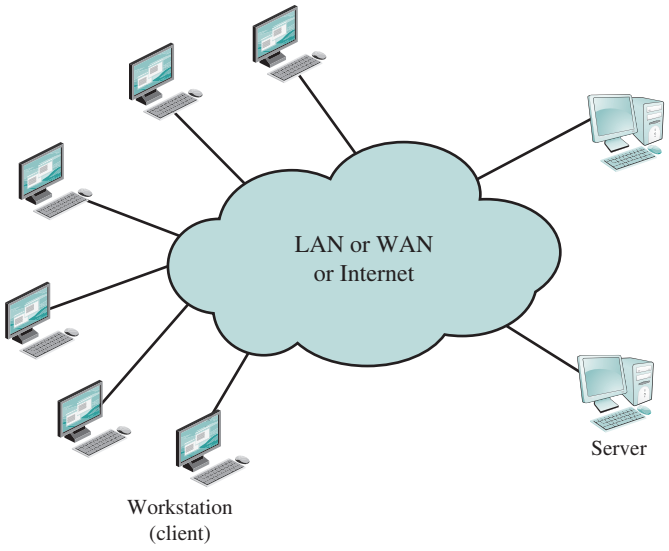


Figure 18.1 Generic Client/Server Environment

Figure 18.1 attempts to capture the essence of the client/server concept. As the term suggests, a *client/server environment* is populated by clients and servers. The **client** machines are generally single-user PCs or workstations that provide a user-friendly interface to the end user. The client-based station generally presents the type of graphical interface that is most comfortable to users, including the use of windows and a mouse. Microsoft Windows and Macintosh OS provide examples of such interfaces. Client-based applications are tailored for ease of use and include such familiar tools as the spreadsheet.

Each **server** in the client/server environment provides a set of shared services to the clients. The most common type of server currently is the database server, usually controlling a relational database. The server enables many clients to share access to the same database and enables the use of a high-performance computer system to manage the database.

In addition to clients and servers, the third essential ingredient of the client/server environment is the **network**. Client/server computing is typically distributed computing. Users, applications, and resources are distributed in response to business requirements and linked by a single LAN or WAN or by an internet of networks.

How does a client/server configuration differ from any other distributed processing solution? There are a number of characteristics that stand out and together, make client/server distinct from other types of distributed processing:

- There is a heavy reliance on bringing user-friendly applications to the user on his or her system. This gives the user a great deal of control over the timing and style of computer usage, and gives department-level managers the ability to be responsive to their local needs.
- Although applications are dispersed, there is an emphasis on centralizing corporate databases and many network management and utility functions. This

enables corporate management to maintain overall control of the total capital investment in computing and information systems and to provide interoperability so systems are tied together. At the same time, it relieves individual departments and divisions of much of the overhead of maintaining sophisticated computer-based facilities, but enables them to choose just about any type of machine and interface they need to access data and information.

- There is a commitment, both by user organizations and vendors, to open and modular systems. This means that the user has more choice in selecting products and in mixing equipment from a number of vendors.
- Networking is fundamental to the operation. Thus, network management and network security have a high priority in organizing and operating information systems.

Client/Server Applications

The key feature of a client/server architecture is the allocation of application-level tasks between clients and servers. Figure 18.2 illustrates the general case. In both client and server, of course, the basic software is an operating system running on the hardware platform. The platforms and the operating systems of client and server may differ. Indeed, there may be a number of different types of client platforms and operating systems and a number of different types of server platforms in a single environment. As long as a particular client and server share the same communications protocols and support the same applications, these lower-level differences are irrelevant.

It is the communications software that enables client and server to interoperate. The principal example of such software is TCP/IP. Of course, the point of all of this support software (communications and operating system) is to provide a base for distributed applications. Ideally, the actual functions performed by the application can be split up between client and server in a way that optimizes the use of resources. In some cases, depending on the application needs, the bulk of the applications

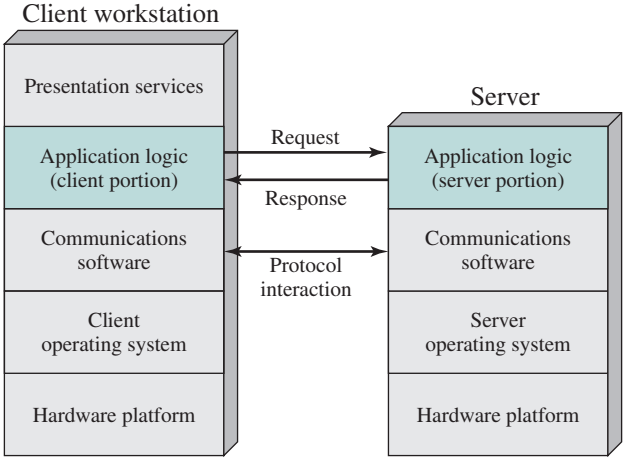


Figure 18.2 Generic Client/Server Architecture

software executes at the server, while in other cases, most of the application logic is located at the client.

An essential factor in the success of a client/server environment is the way in which the user interacts with the system as a whole. Thus, the design of the user interface on the client machine is critical. In most client/server systems, there is heavy emphasis on providing a **graphical user interface (GUI)** that is easy to use, easy to learn, yet powerful and flexible. Thus, we can think of a presentation services module in the client workstation that is responsible for providing a user-friendly interface to the distributed applications available in the environment.

DATABASE APPLICATIONS As an example that illustrates the concept of splitting application logic between client and server, let us consider one of the most common families of client/server applications: those that use relational databases. In this environment, the server is essentially a database server. Interaction between client and server is in the form of transactions in which the client makes a database request and receives a database response.

Figure 18.3 illustrates, in general terms, the architecture of such a system. The server is responsible for maintaining the database, for which purpose a complex database management system software module is required. A variety of different applications that make use of the database can be housed on client machines. The "glue" that ties client and server together is software that enables the client to make requests for access to the server's database. A popular example of such logic is the structured query language (SQL).

Figure 18.3 suggests that all of the application logic—the software for "number crunching" or other types of data analysis—is on the client side, while the server is only concerned with managing the database. Whether such a configuration is appropriate depends on the style and intent of the application. For example, suppose the primary

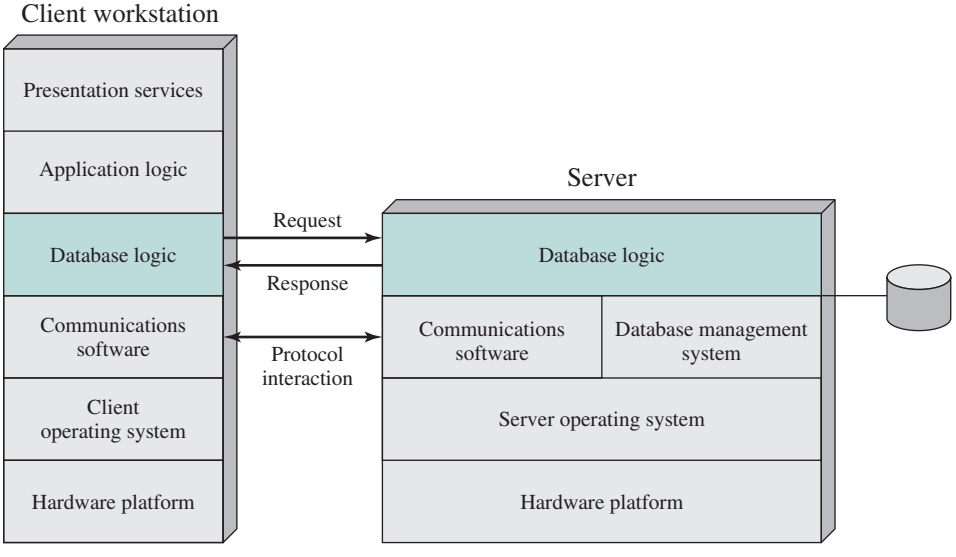
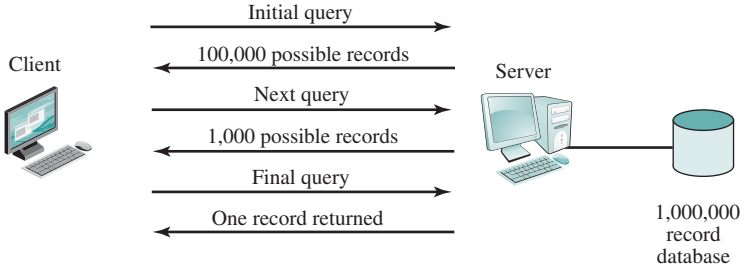


Figure 18.3 Client/Server Architecture for Database Applications

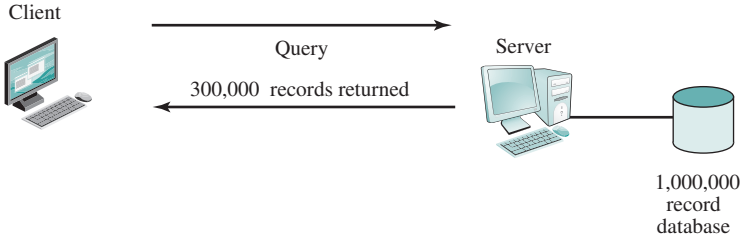
purpose is to provide online access for record lookup. Figure 18.4a suggests how this might work. Suppose the server is maintaining a database of 1 million records (called rows in relational database terminology), and the user wants to perform a lookup that should result in zero, one, or at most a few records. The user could search for these records using a number of search criteria (e.g., records older than 1992, records referring to individuals in Ohio, records referring to a specific event or characteristic, etc.). An initial client query may yield a server response that there are 100,000 records that satisfy the search criteria. The user then adds additional qualifiers and issues a new query. This time, a response indicating that there are 1,000 possible records is returned. Finally, the client issues a third request with additional qualifiers. The resulting search criteria yield a single match, and the record is returned to the client.

The preceding application is well-suited to a client/server architecture for two reasons:

1. There is a massive job of sorting and searching the database. This requires a large disk or bank of disks, a high-speed CPU, and a high-speed I/O architecture. Such capacity and power is not needed and is too expensive for a single-user workstation or PC.
2. It would place too great a traffic burden on the network to move the entire 1-million-record file to the client for searching. Therefore, it is not enough for the server just to be able to retrieve records on behalf of a client; the server needs to have database logic that enables it to perform searches on behalf of a client.



(a) Desirable client/server use



(b) Misused client/server

Figure 18.4 Client/Server Database Usage

Now consider the scenario of Figure 18.4b, which has the same 1-million-record database. In this case, a single query results in the transmission of 300,000 records over the network. This might happen if, for example, the user wishes to find the grand total or mean value of some field across many records or even the entire database.

Clearly, this latter scenario is unacceptable. One solution to this problem, which maintains the client/server architecture with all of its benefits, is to move part of the application logic over to the server. That is, the server can be equipped with application logic for performing data analysis as well as data retrieval and data searching.

CLASSES OF CLIENT/SERVER APPLICATIONS Within the general framework of client/server, there is a spectrum of implementations that divide the work between client and server differently. Figure 18.5 illustrates in general terms some of the major options for database applications. Other splits are possible, and the options may have a different characterization for other types of applications. In any case, it is useful to examine this figure to get a feel for the kind of trade-offs possible.

Figure 18.5 depicts four classes:

- **Host-based processing:** *Host-based processing* is not true client/server computing as the term is generally used. Rather, host-based processing refers to the

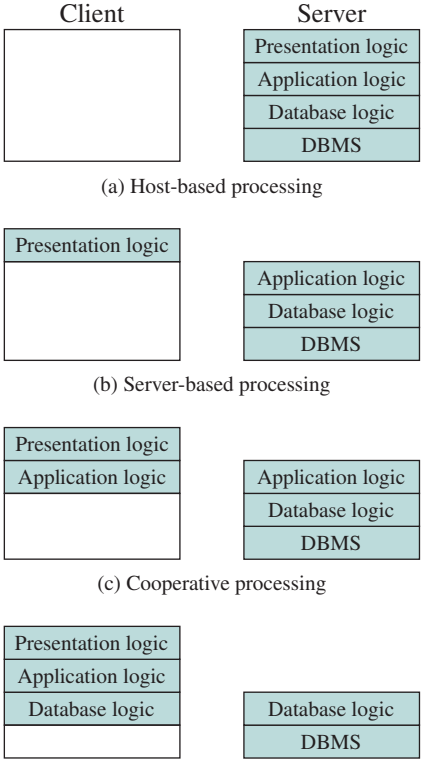


Figure 18.5 Classes of Client/Server Applications

traditional mainframe environment in which all or virtually all of the processing is done on a central host. Often the user interface is via a dumb terminal. Even if the user is employing a microcomputer, the user's station is generally limited to the role of a terminal emulator.

- **Server-based processing:** The most basic class of client/server configuration is one in which the client is principally responsible for providing a graphical user interface, while virtually all of the processing is done on the server. This configuration is typical of early client/server efforts, especially departmental-level systems. The rationale behind such configurations is that the user workstation is best suited to providing a user-friendly interface, and that databases and applications can easily be maintained on central systems. Although the user gains the advantage of a better interface, this type of configuration does not generally lend itself to any significant gains in productivity, or to any fundamental changes in the actual business functions that the system supports.
- **Client-based processing:** At the other extreme, virtually all application processing may be done at the client, with the exception of data validation routines and other database logic functions that are best performed at the server. Generally, some of the more sophisticated database logic functions are housed on the client side. This architecture is perhaps the most common client/server approach in current use. It enables the user to employ applications tailored to local needs.
- **Cooperative processing:** In a cooperative processing configuration, the application processing is performed in an optimized fashion, taking advantage of the strengths of both client and server machines and of the distribution of data. Such a configuration is more complex to set up and maintain but, in the long run, this type of configuration may offer greater user productivity gains and greater network efficiency than other client/server approaches.

Figures 18.5c and 18.5d correspond to configurations in which a considerable fraction of the load is on the client. This so-called **fat client** model has been popularized by application development tools such as Sybase Inc.'s PowerBuilder and Gupta Corp.'s SQL Windows. Applications developed with these tools are typically departmental in scope. The main benefit of the fat client model is that it takes advantage of desktop power, offloading application processing from servers and making them more efficient and less likely to be bottlenecks.

There are, however, several disadvantages to the fat client strategy. The addition of more functions rapidly overloads the capacity of desktop machines, forcing companies to upgrade. If the model extends beyond the department to incorporate many users, the company must install high-capacity LANs to support the large volumes of transmission between the thin servers and the fat clients. Finally, it is difficult to maintain, upgrade, or replace applications distributed across tens or hundreds of desktops.

Figure 18.5b is representative of a **thin client** approach. This approach more nearly mimics the traditional host-centered approach and is often the migration path for evolving corporate-wide applications from the mainframe to a distributed environment.

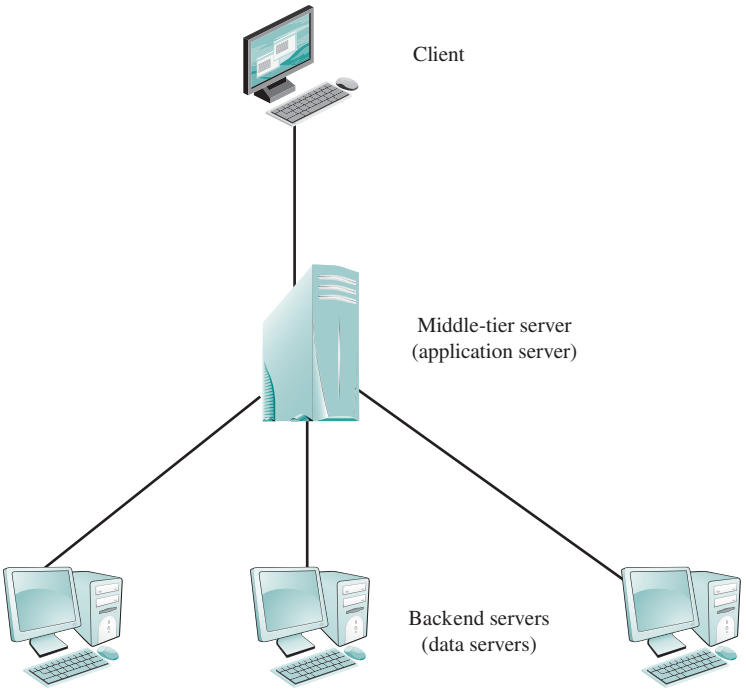


Figure 18.6 Three-Tier Client/Server Architecture

THREE-TIER CLIENT/SERVER ARCHITECTURE The traditional client/server architecture involves two levels, or tiers: a client tier and a server tier. A three-tier architecture is also common (see Figure 18.6). In this architecture, the application software is distributed among three types of machines: a user machine, a middle-tier server, and a backend server. The user machine is the client machine we have been discussing and, in the three-tier model, is typically a thin client. The middle-tier machines are essentially gateways between the thin user clients and a variety of backend database servers. The middle-tier machines can convert protocols and map from one type of database query to another. In addition, the middle-tier machine can merge/integrate results from different data sources. Finally, the middle-tier machine can serve as a gateway between the desktop applications and the backend legacy applications by mediating between the two worlds.

The interaction between the middle-tier server and the backend server also follows the client/server model. Thus, the middle-tier system acts as both a client and a server.

FILE CACHE CONSISTENCY When a file server is used, performance of file I/O can be noticeably degraded relative to local file access because of the delays imposed by the network. To reduce this performance penalty, individual systems can use file caches to hold recently accessed file records. Because of the principle of locality, use of a local file cache should reduce the number of remote server accesses that must be made.

Figure 18.7 illustrates a typical distributed mechanism for caching files among a networked collection of workstations. When a process makes a file access, the request is presented first to the cache of the process's workstation ("file traffic"). If not satisfied there, the request is passed either to the local disk, if the file is stored there ("disk traffic"), or to a file server, where the file is stored ("server traffic"). At the server, the server's cache is first interrogated and, if there is a miss, then the server's disk is accessed. The dual caching approach is used to reduce communications traffic (client cache) and disk I/O (server cache).

When caches always contain exact copies of remote data, we say the caches are **consistent**. It is possible for caches to become inconsistent when the remote data are changed and the corresponding obsolete local cache copies are not discarded. This can happen if one client modifies a file that is also cached by other clients. The difficulty is actually at two levels. If a client adopts a policy of immediately writing any changes to a file back to the server, then any other client that has a cache copy of the relevant portion of the file will have obsolete data. The problem is made even worse if the client delays writing back changes to the server. In that case, the server itself has an obsolete version of the file, and new file read requests to the server might obtain obsolete data. The problem of keeping local cache copies up to date to changes in remote data is known as the **cache consistency** problem.

The simplest approach to cache consistency is to use file-locking techniques to prevent simultaneous access to a file by more than one client. This guarantees consistency at the expense of performance and flexibility. A more powerful approach is provided with the facility in Sprite [NELS88, OUST88]. Any number of remote processes may open a file for read and create their own client cache. But when an open file request to a server requests write access and other processes have the file open for read access, the server takes two actions. First, it notifies the writing process that, although it may maintain a cache, it must write back all altered blocks immediately

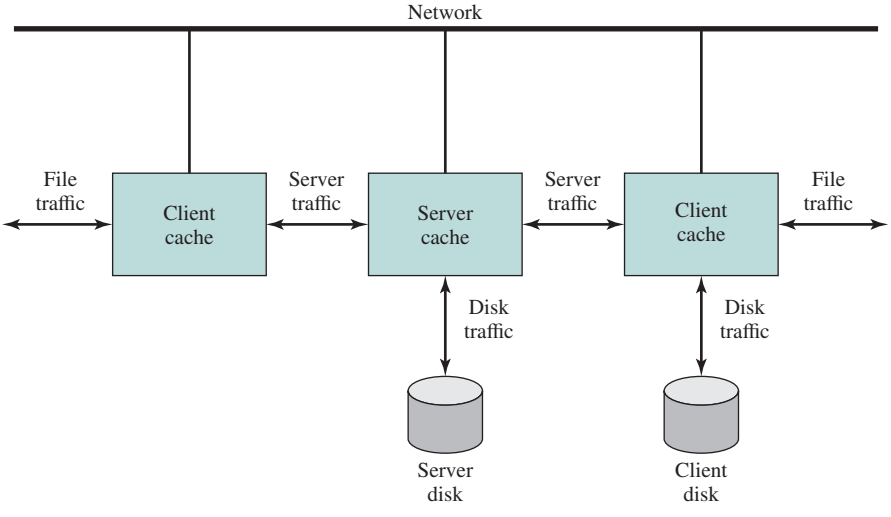


Figure 18.7 Distributed File Caching in Sprite

upon update. There can be at most one such client. Second, the server notifies all reading processes that have the file open that the file is no longer cacheable.

Middleware

The development and deployment of client/server products has far outstripped efforts to standardize all aspects of distributed computing, from the physical layer up to the application layer. This lack of standards makes it difficult to implement an integrated, multivendor, enterprise-wide client/server configuration. Because much of the benefit of the client/server approach is tied up with its modularity and the ability to mix and match platforms and applications to provide a business solution, this interoperability problem must be solved.

To achieve the true benefits of the client/server approach, developers must have a set of tools that provide a uniform means and style of access to system resources across all platforms. This will enable programmers to build applications that not only look and feel the same on various PCs and workstations, but that use the same method to access data regardless of the location of that data.

The most common way to meet this requirement is by the use of standard programming interfaces and protocols that sit between the application above and communications software and operating system below. Such standardized interfaces and protocols have come to be referred to as middleware. With standard programming interfaces, it is easy to implement the same application on a variety of server types and workstation types. This obviously benefits the customer, but vendors are also motivated to provide such interfaces. The reason is that customers buy applications, not servers; customers will only choose among those server products that run the applications they want. The standardized protocols are needed to link these various server interfaces back to the clients that need access to them.

There is a variety of middleware packages ranging from the very simple to the very complex. What they all have in common is the capability to hide the complexities and disparities of different network protocols and operating systems. Client and server vendors generally provide a number of the more popular middleware packages as options. Thus, a user can settle on a particular middleware strategy then assemble equipment from various vendors that support that strategy.

Middleware Architecture Figure 18.8 suggests the role of middleware in a client/server architecture. The exact role of the middleware component will depend on the style of client/server computing being used. Referring back to Figure 18.5, recall that there are a number of different client/server approaches, depending on the way in which application functions are split up. In any case, Figure 18.8 gives a good general idea of the architecture involved.

Note there is both a client and server component of middleware. The basic purpose of middleware is to enable an application or a user at a client to access a variety of services on servers without being concerned about differences among servers. To look at one specific application area, the structured query language (SQL) is supposed to provide a standardized means for access to a relational database by either a local or remote user or application. However, many relational database vendors, although they support SQL, have added their own proprietary extensions to

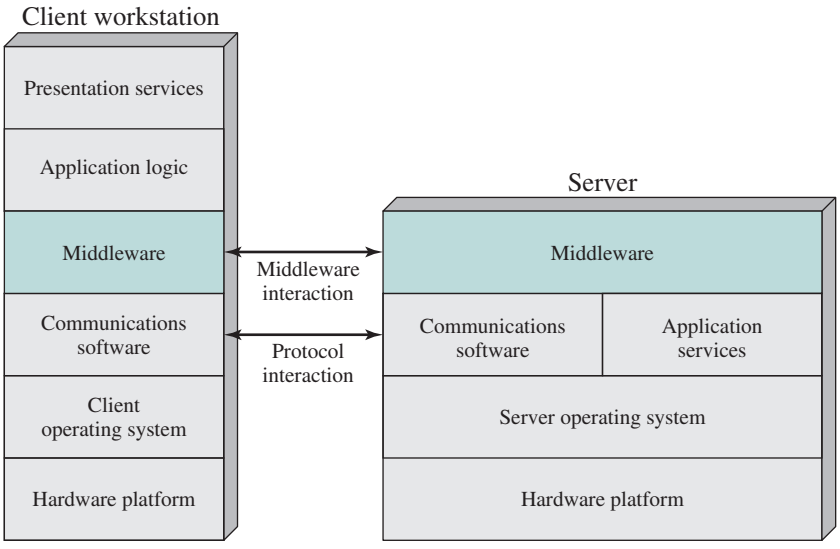


Figure 18.8 The Role of Middleware in Client/Server Architecture

SQL. This enables vendors to differentiate their products but also creates potential incompatibilities.

As an example, consider a distributed system used to support, among other things, the personnel department. The basic employee data, such as employee name and address, might be stored on a Gupta database, whereas salary information might be contained on an Oracle database. When a user in the personnel department requires access to particular records, that user does not want to be concerned with which vendor's database contains the records needed. Middleware provides a layer of software that enables uniform access to these differing systems.

It is instructive to look at the role of middleware from a logical, rather than an implementation, point of view. This viewpoint is illustrated in Figure 18.9. Middleware enables the realization of the promise of distributed client/server computing. The entire distributed system can be viewed as a set of applications and resources available to users. Users need not be concerned with the location of data or indeed the location of applications. All applications operate over a uniform applications programming interface (API). The middleware, which cuts across all client and server platforms, is responsible for routing client requests to the appropriate server.

Although there is a wide variety of middleware products, these products are typically based on one of two underlying mechanisms: message passing or remote procedure calls. These two methods are examined in the next two sections.

18.2 DISTRIBUTED MESSAGE PASSING

It is usually the case in a distributed processing systems that the computers do not share main memory; each is an isolated computer system. Thus, interprocessor communication techniques that rely on shared memory, such as semaphores, cannot be

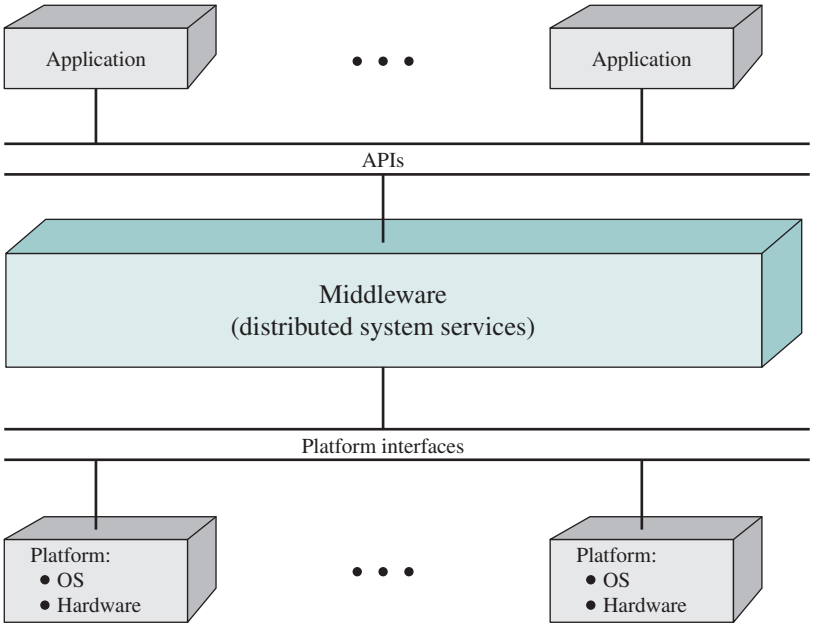


Figure 18.9 **Logical View of Middleware**

used. Instead, techniques that rely on message passing are used. In this section and the next, we look at the two most common approaches. The first is the straightforward application of messages as they are used in a single system. The second is a separate technique that relies on message passing as a basic function: the remote procedure call.

Figure 18.10a shows the use of message passing to implement client/server functionality. A client process requires some service (e.g., read a file, print) and sends a message containing a request for service to a server process. The server process honors the request and sends a message containing a reply. In its simplest form, only two functions are needed: Send and Receive. The Send function specifies a destination and includes the message content. The Receive function tells from whom a message is desired (including "all") and provides a buffer where the incoming message is to be stored.

Figure 18.11 suggests an implementation for message passing. Processes make use of the services of a message-passing module. Service requests can be expressed in terms of primitives and parameters. A primitive specifies the function to be performed, and the parameters are used to pass data and control information. The actual form of a primitive depends on the message-passing software. It may be a procedure call, or it may itself be a message to a process that is part of the operating system.

The Send primitive is used by the process that desires to send the message. Its parameters are the identifier of the destination process and the contents of the message. The message-passing module constructs a data unit that includes these two elements. This data unit is sent to the machine that hosts the destination process, using some sort of communications facility, such as TCP/IP. When the data unit is received in the target system, it is routed by the communications facility to the message-passing module. This module examines the process ID field and stores the message in the buffer for that process.

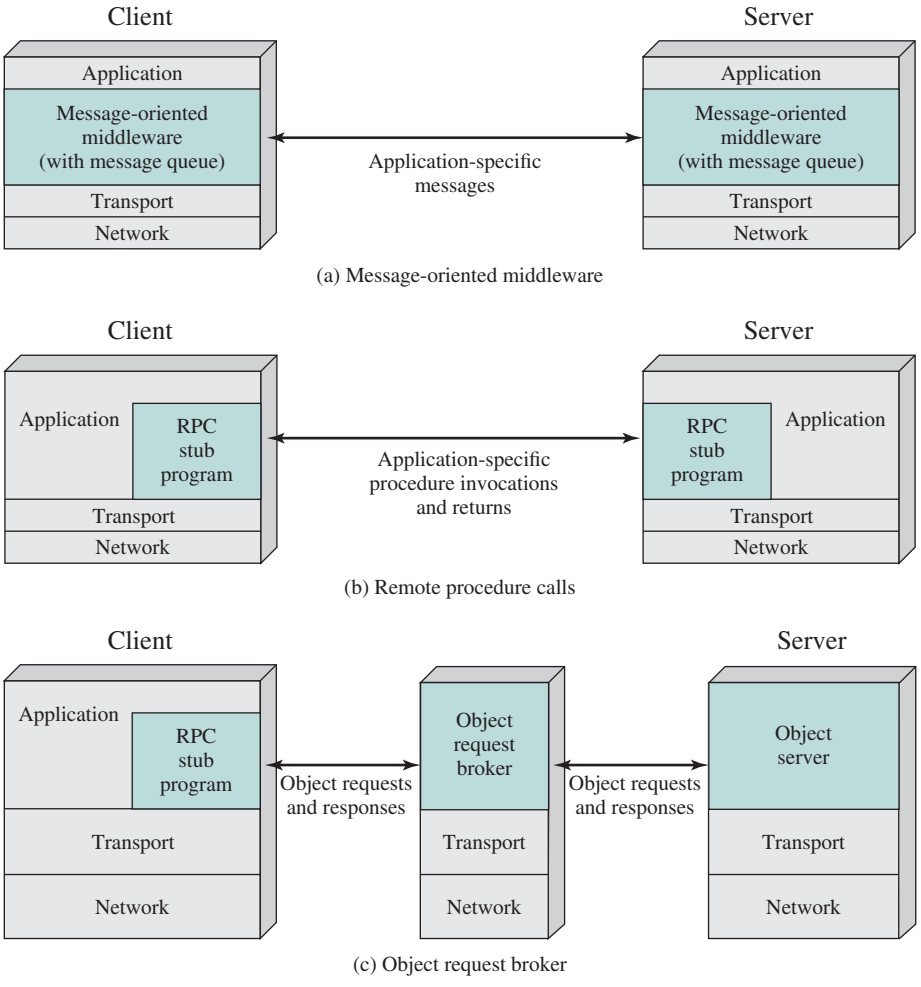


Figure 18.10 **Middleware Mechanisms**

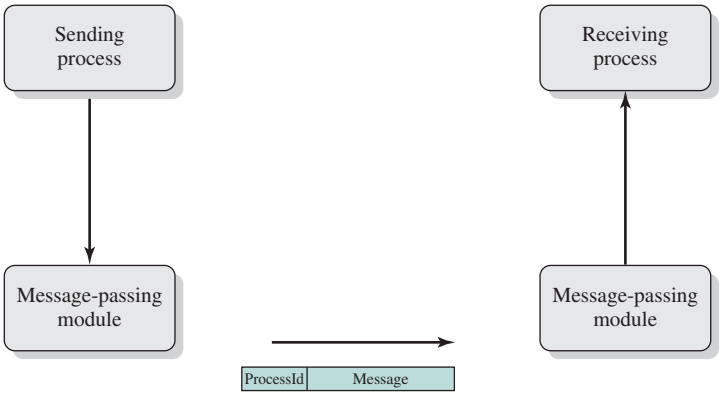


Figure 18.11 **Basic Message-Passing Primitives**

In this scenario, the receiving process must announce its willingness to receive messages by designating a buffer area and informing the message-passing module by a Receive primitive. An alternative approach does not require such an announcement. Instead, when the message-passing module receives a message, it signals the destination process with some sort of Receive signal then makes the received message available in a shared buffer.

Several design issues are associated with distributed message passing, and these are addressed in the remainder of this section.

Reliability versus Unreliability

A reliable message-passing facility is one that guarantees delivery if possible. Such a facility makes use of a reliable transport protocol or similar logic and performs error checking, acknowledgment, retransmission, and reordering of misordered messages. Because delivery is guaranteed, it is not necessary to let the sending process know the message was delivered. However, it might be useful to provide an acknowledgment back to the sending process so it knows that delivery has already taken place. In either case, if the facility fails to achieve delivery (e.g., persistent network failure, crash of destination system), the sending process is notified of the failure.

At the other extreme, the message-passing facility may simply send the message out into the communications network but will report neither success nor failure. This alternative greatly reduces the complexity and processing and communications overhead of the message-passing facility. For those applications that require confirmation that a message has been delivered, the applications themselves may use request and reply messages to satisfy the requirement.

Blocking versus Nonblocking

With nonblocking, or asynchronous, primitives, a process is not suspended as a result of issuing a Send or Receive. Thus, when a process issues a Send primitive, the operating system returns control to the process as soon as the message has been queued for transmission or a copy has been made. If no copy is made, any changes made to the message by the sending process before or even while it is being transmitted are made at the risk of the process. When the message has been transmitted or copied to a safe place for subsequent transmission, the sending process is interrupted to be informed that the message buffer may be reused. Similarly, a nonblocking Receive is issued by a process that then proceeds to run. When a message arrives, the process is informed by interrupt, or it can poll for status periodically.

Nonblocking primitives provide for efficient, flexible use of the message-passing facility by processes. The disadvantage of this approach is that it is difficult to test and debug programs that use these primitives. Irreproducible, timing-dependent sequences can create subtle and difficult problems.

The alternative is to use blocking, or synchronous, primitives. A blocking Send does not return control to the sending process until the message has been transmitted (unreliable service) or until the message has been sent and an acknowledgment received (reliable service). A blocking Receive does not return control until a message has been placed in the allocated buffer.

18.3 REMOTE PROCEDURE CALLS

A variation on the basic message-passing model is the remote procedure call. This is now a widely accepted and common method for encapsulating communication in a distributed system. The essence of the technique is to allow programs on different machines to interact using simple procedure call/return semantics, just as if the two programs were on the same machine. That is, the procedure call is used for access to remote services. The popularity of this approach is due to the following advantages.

1. The procedure call is a widely accepted, used, and understood abstraction.
2. The use of remote procedure calls enables remote interfaces to be specified as a set of named operations with designated types. Thus, the interface can be clearly documented, and distributed programs can be statically checked for type errors.
3. Because a standardized and precisely defined interface is specified, the communication code for an application can be generated automatically.
4. Because a standardized and precisely defined interface is specified, developers can write client and server modules that can be moved among computers and operating systems with little modification and recoding.

The remote procedure call mechanism can be viewed as a refinement of reliable, blocking message passing. Figure 18.10b illustrates the general architecture, and Figure 18.12 provides a more detailed look. The calling program makes a normal procedure call with parameters on its machine. For example,

CALL P(X, Y)

where

P = procedure name
X = passed arguments
Y = returned values

It may or may not be transparent to the user that the intention is to invoke a remote procedure on some other machine. A dummy or stub procedure P must be included in the caller's address space or be dynamically linked to it at call time. This procedure creates a message that identifies the procedure being called and includes the parameters. It then sends this message to a remote system and waits for a reply. When a reply is received, the stub procedure returns to the calling program, providing the returned values.

At the remote machine, another stub program is associated with the called procedure. When a message comes in, it is examined and a local CALL P(X, Y) is generated. This remote procedure is thus called locally, so its normal assumptions about where to find parameters, the state of the stack, and so on are identical to the case of a purely local procedure call.

Several design issues are associated with remote procedure calls, and these are addressed in the remainder of this section.

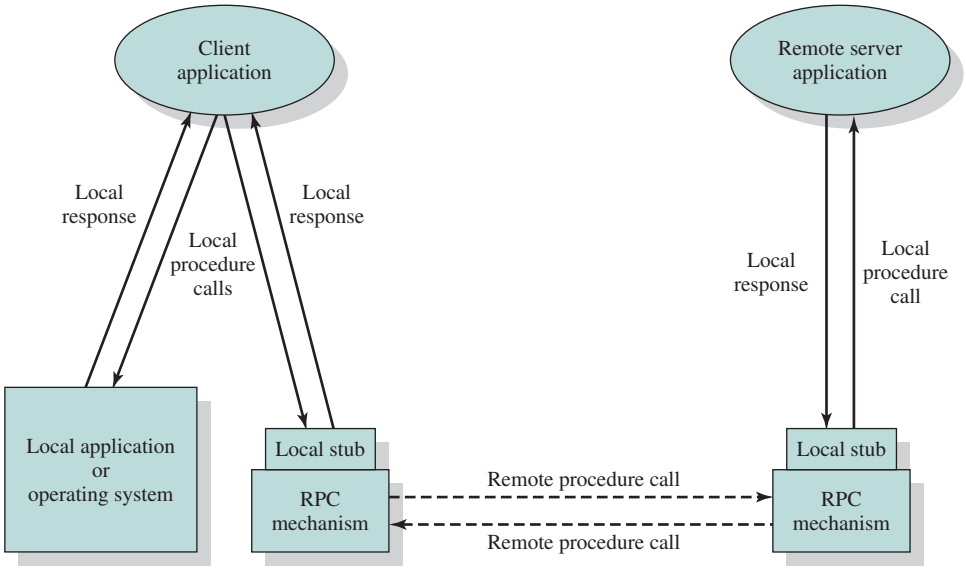


Figure 18.12 **Remote Procedure Call Mechanism**

Parameter Passing

Most programming languages allow parameters to be passed as values (call by value) or as pointers to a location that contains the value (call by reference). Call by value is simple for a remote procedure call: The parameters are simply copied into the message and sent to the remote system. It is more difficult to implement call by reference. A unique, system-wide pointer is needed for each object. The overhead for this capability may not be worth the effort.

Parameter Representation

Another issue is how to represent parameters and results in messages. If the called and calling programs are in identical programming languages on the same type of machines with the same operating system, then the representation requirement may present no problems. If there are differences in these areas, then there will probably be differences in the ways in which numbers and even text are represented. If a full-blown communications architecture is used, then this issue is handled by the presentation layer. However, the overhead of such an architecture has led to the design of remote procedure call facilities that bypass most of the communications architecture and provide their own basic communications facility. In that case, the conversion responsibility falls on the remote procedure call facility (e.g., see [GIBB87]).

The best approach to this problem is to provide a standardized format for common objects, such as integers, floating-point numbers, characters, and character strings. Then the native parameters on any machine can be converted to and from the standardized representation.

Client/Server Binding

Binding specifies how the relationship between a remote procedure and the calling program will be established. A binding is formed when two applications have made a logical connection and are prepared to exchange commands and data.

Nonpersistent binding means that a logical connection is established between the two processes at the time of the remote procedure call, and that as soon as the values are returned, the connection is dismantled. Because a connection requires the maintenance of state information on both ends, it consumes resources. The nonpersistent style is used to conserve those resources. On the other hand, the overhead involved in establishing connections makes nonpersistent binding inappropriate for remote procedures that are called frequently by the same caller.

With **persistent binding**, a connection that is set up for a remote procedure call is sustained after the procedure return. The connection can then be used for future remote procedure calls. If a specified period of time passes with no activity on the connection, then the connection is terminated. For applications that make many repeated calls to remote procedures, persistent binding maintains the logical connection and allows a sequence of calls and returns to use the same connection.

Synchronous versus Asynchronous

The concepts of synchronous and asynchronous remote procedure calls are analogous to the concepts of blocking and nonblocking messages. The traditional remote procedure call is synchronous, which requires that the calling process wait until the called process returns a value. Thus, the **synchronous RPC** behaves much like a subroutine call.

The synchronous RPC is easy to understand and program because its behavior is predictable. However, it fails to exploit fully the parallelism inherent in distributed applications. This limits the kind of interaction the distributed application can have, resulting in lower performance.

To provide greater flexibility, various **asynchronous RPC** facilities have been implemented to achieve a greater degree of parallelism while retaining the familiarity and simplicity of the RPC [ANAN92]. Asynchronous RPCs do not block the caller; the replies can be received as and when they are needed, thus allowing client execution to proceed locally in parallel with the server invocation.

A typical asynchronous RPC use is to enable a client to invoke a server repeatedly so the client has a number of requests in the pipeline at one time, each with its own set of data. Synchronization of client and server can be achieved in one of two ways:

1. A higher-layer application in the client and server can initiate the exchange then check at the end that all requested actions have been performed.
2. A client can issue a string of asynchronous RPCs followed by a final synchronous RPC. The server will respond to the synchronous RPC only after completing all of the work requested in the preceding asynchronous RPCs.

In some schemes, asynchronous RPCs require no reply from the server and the server cannot send a reply message. Other schemes either require or allow a reply, but the caller does not wait for the reply.

Object-Oriented Mechanisms

As object-oriented technology becomes more prevalent in operating system design, client/server designers have begun to embrace this approach. In this approach, clients and servers ship messages back and forth between objects. Object communications may rely on an underlying message or RPC structure or be developed directly on top of object-oriented capabilities in the operating system.

A client that needs a service sends a request to an object request broker, which acts as a directory of all the remote service available on the network (see Figure 18.10c). The broker calls the appropriate object and passes along any relevant data. Then the remote object services the request and replies to the broker, which returns the response to the client.

The success of the object-oriented approach depends on standardization of the object mechanism. Unfortunately, there are several competing designs in this area. One is Microsoft's Component Object Model (COM), the basis for Object Linking and Embedding (OLE). A competing approach, developed by the Object Management Group, is the Common Object Request Broker Architecture (CORBA), which has wide industry support. IBM, Apple, Sun, and many other vendors support the CORBA approach.

18.4 CLUSTERS

Clustering is an alternative to symmetric multiprocessing (SMP) as an approach to providing high performance and high availability and is particularly attractive for server applications. We can define a cluster as a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster; in the literature, each computer in a cluster is typically referred to as a *node*.

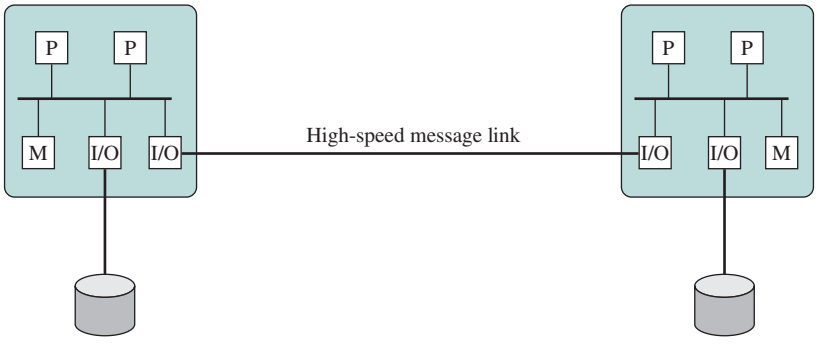
[BREW97] lists four benefits that can be achieved with clustering. These can also be thought of as objectives or design requirements:

- **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest stand-alone machines. A cluster can have dozens or even hundreds of machines, each of which is a multiprocessor.
- **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system.
- **High availability:** Because each node in a cluster is a stand-alone computer, the failure of one node does not mean loss of service. In many products, fault tolerance is handled automatically in software.
- **Superior price/performance:** By using commodity building blocks, it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

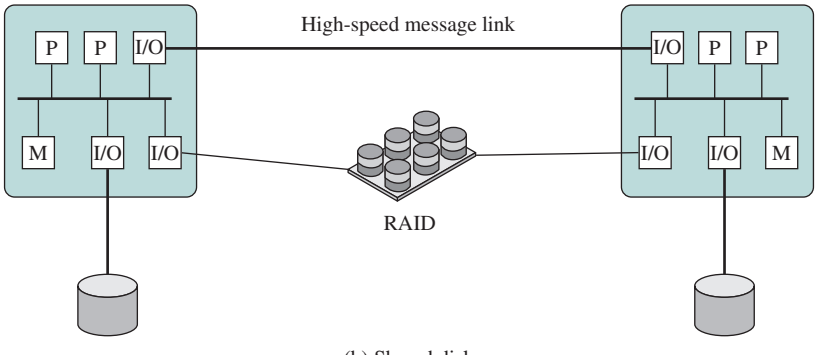
Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Figure 18.13a shows a two-node cluster in which the only interconnection is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster, or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so there is a connection between the server cluster and remote client systems. Note in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Figure 18.13, the other alternative is a shared disk cluster. In this case, there generally is still a message link between nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In Figure 18.13b, the common disk subsystem is a RAID system. The use of RAID or some similar redundant disk technology is common in clusters so the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.



(a) Standby server with no shared disk



(b) Shared disk

Figure 18.13 Cluster Configurations

Table 18.2 Clustering Methods: Benefits and Limitations

Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data are continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

A clearer picture of the range of clustering approaches can be gained by looking at functional alternatives. A white paper from Hewlett Packard [HP96] provides a useful classification along functional lines (see Table 18.2), which we now discuss.

A common, older method, known as **passive standby**, is simply to have one computer handle all of the processing load while the other computer remains inactive, standing by to take over in the event of a failure of the primary. To coordinate the machines, the active, or primary, system periodically sends a "heartbeat" message to the standby machine. Should these messages stop arriving, the standby assumes that the primary server has failed and puts itself into operation. This approach increases availability but does not improve performance. Further, if the only information that is exchanged between the two systems is a heartbeat message, and if the two systems do not share common disks, then the standby provides a functional backup but has no access to the databases managed by the primary.

The passive standby is generally not referred to as a cluster. The term cluster is reserved for multiple interconnected computers that are all actively doing processing while maintaining the image of a single system to the outside world. The term **active secondary** is often used in referring to this configuration. Three classifications of clustering can be identified: separate servers, shared nothing, and shared memory.

In one approach to clustering, each computer is a **separate server** with its own disks and there are no disks shared between systems (see Figure 18.13a). This arrangement provides high performance as well as high availability. In this case, some type of management or scheduling software is needed to assign incoming client requests to servers so the load is balanced and high utilization is achieved. It is desirable to

have a failover capability, which means that if a computer fails while executing an application, another computer in the cluster can pick up and complete the application. For this to happen, data must constantly be copied among systems so each system has access to the current data of the other systems. The overhead of this data exchange ensures high availability at the cost of a performance penalty.

To reduce the communications overhead, most clusters now consist of servers connected to common disks (see Figure 18.13b). In one variation of this approach, called **shared nothing**, the common disks are partitioned into volumes, and each volume is owned by a single computer. If that computer fails, the cluster must be reconfigured so some other computer has ownership of the volumes of the failed computer.

It is also possible to have multiple computers share the same disks at the same time (called the **shared disk** approach), so each computer has access to all of the volumes on all of the disks. This approach requires the use of some type of locking facility to ensure data can only be accessed by one computer at a time.

Operating System Design Issues

Full exploitation of a cluster hardware configuration requires some enhancements to a single-system operating system.

Failure Management How failures are managed by a cluster depends on the clustering method used (see Table 18.2). In general, two approaches can be taken to dealing with failures: highly available clusters and fault-tolerant clusters. A highly available cluster offers a high probability that all resources will be in service. If a failure occurs, such as a node goes down or a disk volume is lost, then the queries in progress are lost. Any lost query, if retried, will be serviced by a different computer in the cluster. However, the cluster operating system makes no guarantee about the state of partially executed transactions. This would need to be handled at the application level.

A fault-tolerant cluster ensures all resources are always available. This is achieved by the use of redundant shared disks and mechanisms for backing out uncommitted transactions and committing completed transactions.

The function of switching an application and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**. A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**. Failback can be automated, but this is desirable only if the problem is truly fixed and unlikely to recur. If not, automatic failback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems.

Load Balancing A cluster requires an effective capability for balancing the load among available computers. This includes the requirement that the cluster be incrementally scalable. When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications. Middleware mechanisms need to recognize that services can appear on different members of the cluster and may migrate from one member to another.

Parallelizing Computation In some cases, effective use of a cluster requires executing software from a single application in parallel. [KAPP00] lists three general approaches to the problem:

- **Parallelizing compiler:** A parallelizing compiler determines, at compile time, which parts of an application can be executed in parallel. These are then split off to be assigned to different computers in the cluster. Performance depends on the nature of the problem and how well the compiler is designed.
- **Parallelized application:** In this approach, the programmer writes the application from the outset to run on a cluster and uses message passing to move data, as required, between cluster nodes. This places a high burden on the programmer but may be the best approach for exploiting clusters for some applications.
- **Parametric computing:** This approach can be used if the essence of the application is an algorithm or program that must be executed a large number of times, each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios, then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an orderly manner.

Cluster Computer Architecture

Figure 18.14 shows a typical cluster architecture. The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently. In addition, a middleware layer of software is installed in each computer to enable cluster operation. The cluster middleware provides a unified system image to the user, known as a **single-system image**. The middleware may also be responsible for providing high availability, by means of load balancing and responding to failures in individual components. [HWAN99] lists the following as desirable cluster middleware services and functions:

- **Single entry point:** A user logs on to the cluster rather than to an individual computer
- **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- **Single control point:** There is a default node used for cluster management and control.
- **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.
- **Single memory space:** Distributed shared memory enables programs to share variables.
- **Single job-management system:** Under a cluster job scheduler, a user can submit a job without specifying the host computer to execute the job.

- **Single-user interface:** A common graphic interface supports all users, regardless of the workstation from which they enter the cluster.
- **Single I/O space:** Any node can remotely access any I/O peripheral or disk device without knowledge of its physical location.
- **Single process space:** A uniform process-identification scheme is used. A process on any node can create or communicate with any other process on a remote node.
- **Checkpointing:** This function periodically saves the process state and intermediate computing results, to allow rollback recovery after a failure.
- **Process migration:** This function enables load balancing.

The last four items on the preceding list enhance the availability of the cluster. The remaining items are concerned with providing a single-system image.

Returning to Figure 18.14, a cluster will also include software tools for enabling the efficient execution of programs that are capable of parallel execution.

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP has been around far longer.

The main strength of the SMP approach is that an SMP is easier to manage and configure than a cluster. The SMP is much closer to the original single-processor model for which nearly all applications are written. The principal change required in going from a uniprocessor to an SMP is to the scheduler function. Another benefit of the SMP is that it usually takes up less physical space and draws less power than

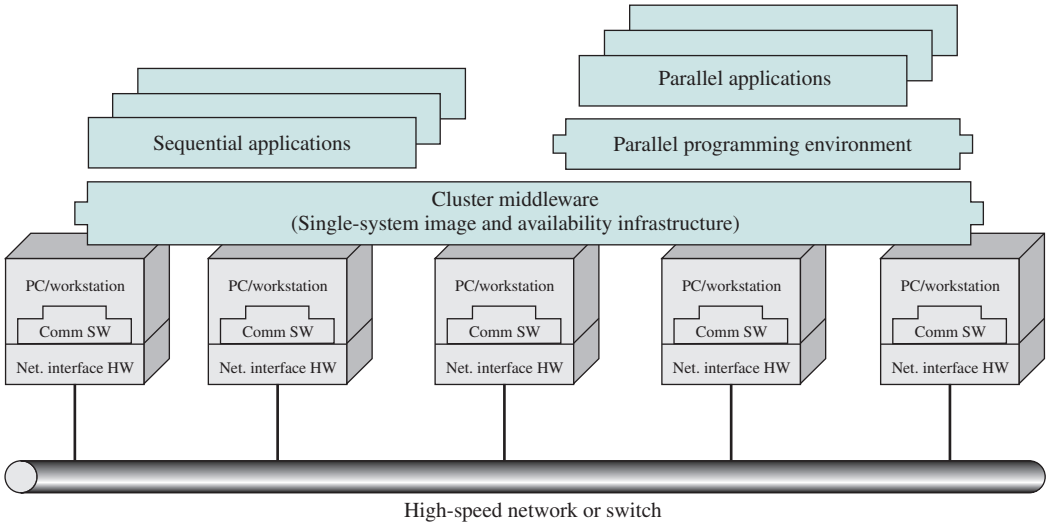


Figure 18.14 Cluster Computer Architecture

a comparable cluster. A final important benefit is that the SMP products are well established and stable.

Over the long run, however, the advantages of the cluster approach are likely to result in clusters dominating the high-performance server market. Clusters are far superior to SMPs in terms of incremental and absolute scalability. Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

18.5 WINDOWS CLUSTER SERVER

Windows Failover Clustering is a shared-nothing cluster, in which each disk volume and other resources are owned by a single system at a time.

The Windows cluster design makes use of the following concepts:

- **Cluster Service:** The collection of software on each node that manages all cluster-specific activity.
- **Resource:** An item managed by the cluster service. All resources are objects representing actual resources in the system, including hardware devices such as disk drives and network cards and logical items such as logical disk volumes, TCP/IP addresses, entire applications, and databases.
- **Online:** A resource is said to be online at a node when it is providing service on that specific node.
- **Group:** A collection of resources managed as a single unit. Usually, a group contains all of the elements needed to run a specific application, and for client systems to connect to the service provided by that application.

The concept of *group* is of particular importance. A group combines resources into larger units that are easily managed, both for failover and load balancing. Operations performed on a group, such as transferring the group to another node, automatically affect all of the resources in that group. Resources are implemented as dynamically linked libraries (DLLs) and managed by a resource monitor. The resource monitor interacts with the cluster service via remote procedure calls and responds to cluster service commands to configure and move resource groups.

Figure 18.15 depicts the Windows clustering components and their relationships in a single system of a cluster. The **node manager** is responsible for maintaining this node's membership in the cluster. Periodically, it sends heartbeat messages to the node managers on other nodes in the cluster. In the event that one node manager detects a loss of heartbeat messages from another cluster node, it broadcasts a message to the entire cluster, causing all members to exchange messages to verify their view of current cluster membership. If a node manager does not respond, it is removed from the cluster and its active groups are transferred to one or more other active nodes in the cluster.

The **configuration database manager** maintains the cluster configuration database. The database contains information about resources and groups and node

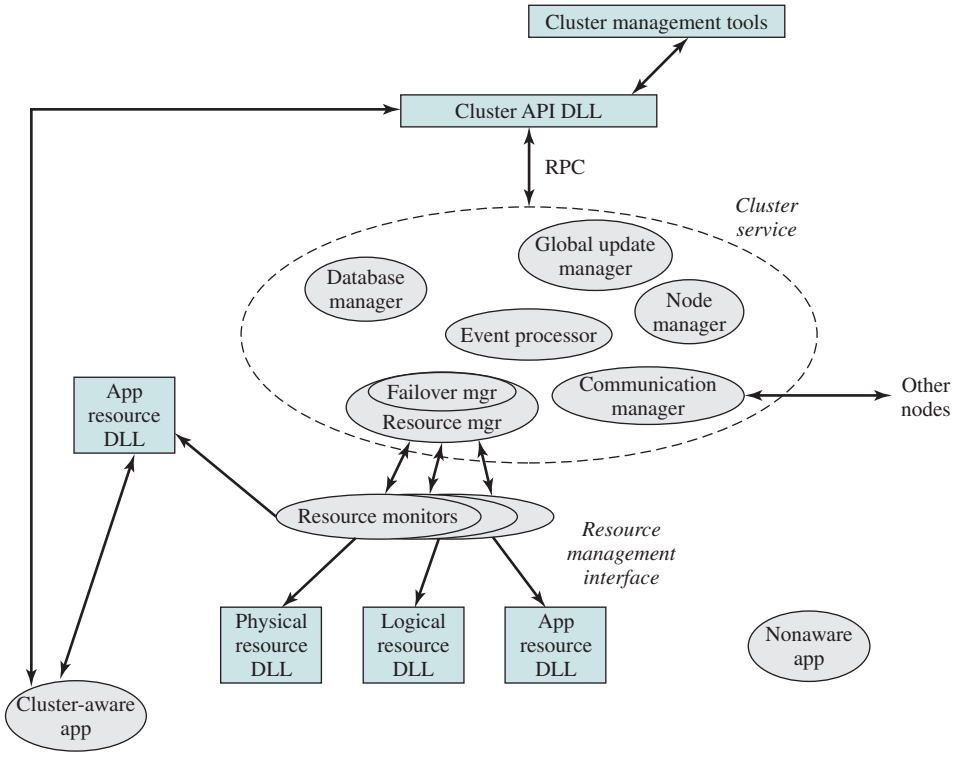


Figure 18.15 Windows Cluster Server Block Diagram

ownership of groups. The database managers on each of the cluster nodes cooperate to maintain a consistent picture of configuration information. Fault-tolerant transaction software is used to assure that changes in the overall cluster configuration are performed consistently and correctly.

The **resource manager/failover manager** makes all decisions regarding resource groups and initiates appropriate actions such as startup, reset, and failover. When failover is required, the failover managers on the active node cooperate to negotiate a distribution of resource groups from the failed system to the remaining active systems. When a system restarts after a failure, the failover manager can decide to move some groups back to this system. In particular, any group may be configured with a preferred owner. If that owner fails and then restarts, the group is moved back to the node in a rollback operation.

The **event processor** connects all of the components of the cluster service, handles common operations, and controls cluster service initialization. The communications manager manages message exchange with all other nodes of the cluster. The global update manager provides a service used by other components within the cluster service.

Microsoft is continuing to ship their cluster product, but they have also developed virtualization solutions based on efficient live migration of virtual

machines between hypervisors running on different computer systems as part of Windows Server 2008 R2. For new applications, live migration offers many benefits over the cluster approach, such as simpler management, and improved flexibility.

18.6 BEOWULF AND LINUX CLUSTERS

In 1994, the Beowulf project was initiated under the sponsorship of the NASA High Performance Computing and Communications (HPCC) project. Its goal was to investigate the potential of clustered PCs for performing important computation tasks beyond the capabilities of contemporary workstations at minimum cost. Today, the Beowulf approach is widely implemented and is perhaps the most important cluster technology available.

Beowulf Features

Key features of Beowulf include the following [RIDG97]:

- Mass market commodity components
- Dedicated processors (rather than scavenging cycles from idle workstations)
- A dedicated, private network (LAN or WAN or internetted combination)
- No custom components
- Easy replication from multiple vendors
- Scalable I/O
- A freely available software base
- Use of freely available distribution computing tools with minimal changes
- Return of the design and improvements to the community

Although elements of Beowulf software have been implemented on a number of different platforms, the most obvious choice for a base is Linux, and most Beowulf implementations use a cluster of Linux workstations and/or PCs. Figure 18.16 depicts a representative configuration. The cluster consists of a number of workstations, perhaps of differing hardware platforms, all running the Linux operating system. Secondary storage at each workstation may be made available for distributed access (for distributed file sharing, distributed virtual memory, or other uses). The cluster nodes (the Linux systems) are interconnected with a commodity networking approach, typically Ethernet. The Ethernet support may be in the form of a single Ethernet switch or an interconnected set of switches. Commodity Ethernet products at the standard data rates (10 Mbps, 100 Mbps, 1 Gbps) are used.

Beowulf Software

The Beowulf software environment is implemented as an add-on to commercially available, royalty-free base Linux distributions. The principal source of open-source Beowulf software is the Beowulf site at www.beowulf.org, but numerous other organizations also offer free Beowulf tools and utilities.

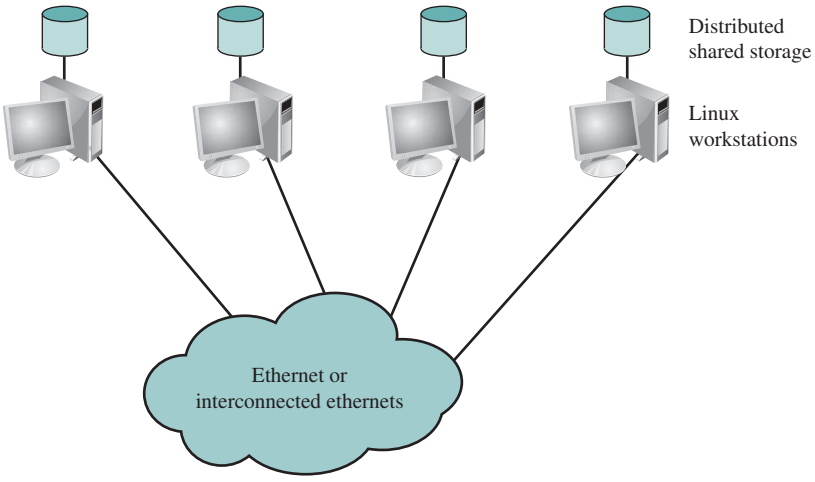


Figure 18.16 Generic Beowulf Configuration

Each node in the Beowulf cluster runs its own copy of the Linux kernel and can function as an autonomous Linux system. To support the Beowulf cluster concept, extensions are made to the Linux kernel to allow the individual nodes to participate in a number of global namespaces. The following are examples of Beowulf system software:

- **Beowulf distributed process space (BPROC):** This package allows a process ID space to span multiple nodes in a cluster environment and also provides mechanisms for starting processes on other nodes. The goal of this package is to provide key elements needed for a single-system image on Beowulf cluster. BPROC provides a mechanism to start processes on remote nodes without ever logging into another node, and by making all the remote processes visible in the process table of the cluster's front-end node.
- **Beowulf Ethernet channel bonding:** This is a mechanism that joins multiple low-cost networks into a single logical network with higher bandwidth. The only additional work over using single network interface is the computationally simple task of distributing the packets over the available device transmit queues. This approach allows load balancing over multiple Ethernets connected to Linux workstations.
- **Pvmsync:** This is a programming environment that provides synchronization mechanisms and shared data objects for processes in a Beowulf cluster.
- **EnFuzion:** EnFuzion consists of a set of tools for doing parametric computing. Parametric computing involves the execution of a program as a large number of jobs, each with different parameters or starting conditions. EnFusion emulates a set of robot users on a single root node machine, each of which will log into one of the many clients that form a cluster. Each job is set up to run with a unique, programmed scenario, with an appropriate set of starting conditions [KAPP00].

18.7 SUMMARY

Client/server computing is the key to realizing the potential of information systems and networks to improve productivity significantly in organizations. With client/server computing, applications are distributed to users on single-user workstations and personal computers. At the same time, resources that can and should be shared are maintained on server systems that are available to all clients. Thus, the client/server architecture is a blend of decentralized and centralized computing.

Typically, the client system provides a graphical user interface (GUI) that enables a user to exploit a variety of applications with minimal training and relative ease. Servers support shared utilities, such as database management systems. The actual application is divided between client and server in a way intended to optimize ease of use and performance.

The key mechanism required in any distributed system is interprocess communication. Two techniques are in common use. A message-passing facility generalizes the use of messages within a single system. The same sorts of conventions and synchronization rules apply. Another approach is the use of the remote procedure call. This is a technique by which two programs on different machines interact using procedure call/return syntax and semantics. Both the called and calling program behave as if the partner program were running on the same machine.

A cluster is a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster.

18.8 REFERENCES

ANAN92 Ananda, A.; Tay, B.; and Koh, E. "A Survey of Asynchronous Remote Procedure Calls." *Operating Systems Review*, April 1992.

BREW97 Brewer, E. "Clustering: Multiply and Conquer." *Data Communications*, July 1997.

GIBB87 Gibbons, P. "A Stub Generator for Multilanguage RPC in Heterogeneous Environments." *IEEE Transactions on Software Engineering*, January 1987.

HP96 Hewlett Packard. *White Paper on Clustering*. June 1996.

HWAN99 Hwang, K., et al. "Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space." *IEEE Concurrency*, January–March 1999.

KAPP00 Kapp, C. "Managing Cluster Computers." *Dr. Dobb's Journal*, July 2000.

NELS88 Nelson, M.; Welch, B.; and Ousterhout, J. "Caching in the Sprite Network File System." *ACM Transactions on Computer Systems*, February 1988.

OUST88 Ousterhout, J., et al. "The Sprite Network Operating System." *Computer*, February 1988.

RIDG97 Ridge, D., et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs." *Proceedings, IEEE Aerospace Conference*, 1997.

18.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

applications programming interface	failback	middleware
Beowulf	failover	remote procedure call (RPC)
client	fat client	server
cluster	file cache consistency	thin client
distributed message passing	graphical user interface (GUI)	
	message	

Review Questions

18.1. What is client/server computing?

18.2. What distinguishes client/server computing from any other form of distributed data processing?

18.3. What is the role of a communications architecture such as TCP/IP in a client/server environment?

18.4. Discuss the rationale for locating applications on the client, the server, or split between client and server.

18.5. What are fat clients and thin clients, and what are the differences in philosophy of the two approaches?

18.6. Suggest pros and cons for fat client and thin client strategies.

18.7. Explain the rationale behind the three-tier client/server architecture.

18.8. What is middleware?

18.9. Because we have standards such as TCP/IP, why is middleware needed?

18.10. List some benefits and disadvantages of blocking and nonblocking primitives for message passing.

18.11. List some benefits and disadvantages of nonpersistent and persistent binding for RPCs.

18.12. List some benefits and disadvantages of synchronous and asynchronous RPCs.

18.13. List and briefly define four different clustering methods.

Problems

18.1. Let α be the percentage of program code that can be executed simultaneously by n computers in a cluster, each computer using a different set of parameters or initial conditions. Assume the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of x MIPS.

a. Derive an expression for the effective MIPS rate when using the system for exclusive execution of this program, in terms of n, α, and x.

b. If $n = 16$ and $x = 4$ MIPS, determine the value of α that will yield a system performance of 40 MIPS.

18.2. An application program is executed on a nine-computer cluster. A benchmark program takes time T on this cluster. Further, 25% of T is time in which the application is running simultaneously on all nine computers. The remaining time, the application has to run on a single computer.

a. Calculate the effective speedup under the aforementioned condition as compared to executing the program on a single computer. Also calculate, the percentage of code that has been parallelized (programmed or compiled so as to use the cluster mode) in the preceding program.

b. Suppose we are able to effectively use 18 computers rather than 9 computers on the parallelized portion of the code. Calculate the effective speedup that is achieved.

18.3. The following FORTRAN program is to be executed on a computer, and a parallel version is to be executed on a 32-computer cluster:

```
L1:      DO 10 I = 1,1024
L2:      SUM(I) = 0
L3:      DO 20 J = 1, I
L4: 20 SUM(I) = SUM(I) + I
L5: 10 CONTINUE
```

Suppose lines 2 and 4 each take two machine cycle times, including all processor and memory-access activities. Ignore the overhead caused by the software loop control statements (lines 1, 3, 5) and all other system overhead and resource conflicts.

a. What is the total execution time (in machine cycle times) of the program on a single computer?

b. Divide the I-loop iterations among the 32 computers as follows: Computer 1 executes the first 32 iterations (I = 1 to 32), processor 2 executes the next 32 iterations, and so on. What are the execution time and speedup factor compared with part (a)? (Note the computational workload, dictated by the J-loop, is unbalanced among the computers.)

c. Explain how to modify the parallelizing to facilitate a balanced parallel execution of all the computational workload over 32 computers. A balanced load means an equal number of additions assigned to each computer with respect to both loops.

d. What is the minimum execution time resulting from the parallel execution on 32 computers? What is the resulting speedup over a single computer?