MONITOR

Peter J. Denning, Naval Postgraduate School, Monterey, California
Walter F. Tichy, University of Karlsruhe, Karlsruhe, Germany

January 2008
Rev 6/5/08

Abstract: The term monitor denotes a control program that oversees the allocation of resources among a set of user programs.

Keywords: operating systems, synchronization, mutual exclusion, atomic operation

The term *monitor* denotes a control program that oversees the allocation of resources among a set of user programs. Along with *supervisor* and *executive*, it was an early synonym for operating system. An old example was the Fortran Monitor System (FMS), which appeared on the IBM 709 series beginning in the late 1950s to provide run-time support for Fortran programs. A later example was the Conversational Monitor System (CMS), a single-user interactive system that ran on a virtual machine within the IBM VM/370. Beginning in the early 1970s, the term *monitor* was slowly dissociated from operating systems and associated with object-class managers. The first such use was in secure operating systems: The *reference monitor* was an object-class manager that enforced an access policy among the users of those objects (1).

In 1974, Hoare (2) proposed programming language structures that simplified operating systems by providing a separate scheduler for each class of resources. The monitor maintains records of queues and resource states, and it gives external processes access through a high-level interface consisting of monitor procedures (today called methods). As with abstract data types, the monitor's methods restrict the caller to a few well-defined operations on the monitor's resources, hiding most details. Unlike abstract data types, monitors have internal locks that permit only one process to execute monitor instructions at a time, thereby permitting concurrent operations on the objects. Other processes must wait in a queue to enter the monitor. If a process in the monitor stops to wait for a resource to become available, the monitor must be unlocked so that another process (notably one that will release the desired resource) can gain access.

An example of a resource monitor is given below in Java, a modern language that supports the concept. With this monitor, a process can request that it be allocated a unit from a pool of resources; it can use that unit exclusively for a period of time, and then release it back to the pool. To set this up, a programmer includes these statements to declare a monitor for printers and to initialize it with a vector containing the names of all available printers:

```
ResourceMonitor printers;
printers = new ResourceMonitor(...);
```

Thereafter, the programmer can cause a thread (process) to acquire, use, and release a printer with this pattern:

```
myPrinter = printers.acquire();
 ...
 (use of printer)
 ...
 printers.release(myPrinter);
```

The Java code for the resource monitor is as follows:

```
class ResourceMonitor {
    Vector pool;
    /* initialization */
     ResourceMonitor(Vector initialResources) {
         pool = initialResources;
     }
    /* acquire an available resource; wait if none available */
     public synchronized Object acquire() {
         Object out;
         while (pool.isEmpty()) wait();
         out = pool.firstElement();
         pool.removeElement(out);
         return out;
     }
    /* return a resource to the pool */
     public synchronized void release(Object in) {
         pool.addElement(in);
         notify();
     }
 }
```

The methods `acquire()` and `release()` are declared as *public*, meaning that any process can call them, and *synchronized*, meaning that they are executed atomically. This implements the central feature of a monitor, which is that the operations are mutually exclusive and that their component steps cannot be interleaved. This feature is implemented by locks inserted by the compiler.

The Java operation `wait()` suspends the calling thread (process), releases the mutual exclusion lock on the methods (acquire, release), and records the process identifier in a waiting queue; `notify()` signals one of those. The signaled thread

is removed from the queue and resumes its operation after relocking the monitor. Note that in this example, the call to `wait()` appears in a loop; it is possible that the thread may find the pool empty by the time it reenters the monitor; in which case, it will call `wait()` again. This can occur if another thread overtakes the signaled one.

Monitors are also useful in structuring distributed systems. The monitor operations can be called via remote procedure calls (RPCs). Most RPC mechanisms have built-in error controls to guard against lost messages; a lost message could cause a deadlock when a thread called the release method but the monitor did not receive the release message.

A monitor callable via RPC must be organized to avoid deadlock when network connections fail. When a remote call message arrives at the monitor, the method to be executed is given to a "proxy thread" running on the server housing the monitor. Meanwhile, the client application thread that issued the RPC is suspended on its host computer until the proxy thread completes and returns the result. Even if the connection or the caller's host crashes, the proxy thread will complete and unlock the monitor, allowing other threads on other computers to call the monitor.

The programmer of the application that called the remote monitor sees some complications. They originate from the dynamic bindings between the programmer's computer and the server housing the monitor. The complications show up as new exceptions from the RPC system -- e.g., unknown host, failed or hung connection, parameter marshaling error, no such remote procedure, no such object, and no response within the time-out period. The responses to these exceptions will depend on the application.

BIBLIOGRAPHY

1. P. Denning and G. S. Graham, Protection: principles and practice, *Proc. AFIPS Spring Joint Comput. Conf.*, **40**, 417-429, 1972.
2. C. A. R. Hoare, Monitors: An operating system structuring concept, *ACM Commun.*, **17**(10), 1974.

FURTHER READING

P. Brinch Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

K. Arnold and J. Gosling, *The Java Programming Language*. Reading, MA: Addison-Wesley, 1996.