Building Monitors with UNIX and C

Neil Dunstan

University of New England, Armidale, NSW, 2351, Australia.

1. Abstract

An outline is given for structuring concurrent programs written in C under UNIX System V using the concept of monitors. It is shown how a monitor can be implemented in UNIX System V using the semaphore and shared memory facilities. Monitors are a common topic in the studies of concurrent programming and operating systems.

2. Introduction

UNIX System V offers a variety of system calls for inter-process communication and synchronization. These facilities provide the means for low level solutions to many synchronization problems [Dunstan, 1989]. The monitor [Hoare, 1972] is a high level concurrent programming construct that is a feature of several programming languages designed for concurrent programming, such as Pascal-Plus [Welsh, 1979], Concurrent Euclid [Holt, 1983] and Turing-Plus [Holt and Cordy, 1988]. Monitors have also been incorporated in a language designed for teaching purposes [Terry, 1985].

A monitor is an information hiding module with a procedural interface. Only one process at a time may be *active* within the monitor. Typically, a monitor exists to coordinate access to a resource or group of resources. Processes wishing to use the resource call an appropriate monitor interface procedure. In some circumstances, a process may become blocked inside the monitor until the monitor's internal state indicates that it may continue. A blocked process is queued on a *condition* variable.

3. A Semaphore Implementation of Monitors

A semaphore implementation of monitors is given in [Hoare, 1972] and [Peterson and Silberschatz, 1985]. In this implementation, each interface procedure is surrounded by entry and exit protocols to ensure the integrity of the monitor. A semaphore is used to guard access to the monitor. Each *condition* variable has an associated semaphore and a count of the number of processes suspended on that *condition* as a result of invoking the *wait* operation. If a process *signal*s a *condition* and in doing so re-activates a suspended process, it must suspend itself on yet another semaphore used for this purpose. A count is kept of the number of such processes. When a process exits from an interface procedure, the exit protocol will release a process from this semaphore, if there is one, otherwise exclusion on the monitor is released.

4. Components of a Monitor

Three components of a monitor can be identified. The first contains the semaphores and counters needed for ensuring monitor security, that is, mutually exclusive access to the monitor. This component is identical for all monitors and may be represented as

```
struct security
{
        int mutex, signaller; /* semaphores */
        int sig_count; /* suspded signllers */
        }
```

where *mutex* represents the mutual exclusion semaphore, *signaller* represents the semaphore upon which signalling processes are suspended and *sig_count* is the number of suspended signalling processes.

The second component represents a *condition* variable.

```
struct condition
{
        int q; /* semaphore id */
        int count; /* # blocked on q */
        }
```

where *q* represents the semaphore associated with the condition and *count* is the number of processes waiting on the condition.

The third component encapsulates the internal state variables of the monitor. This will be different for each monitor application. Here, a single resource monitor (as in [Hoare, 1972]) is used as an example. In this example, only one *boolean* variable is required. It indicates the availability of the resource.

```
struct internal_state
{
        int busy;
        }
```

More complicated monitors will require more variables to record their internal state.

The following code sections for the single resource monitor example assume the variable declarations

```
struct security *msecurity;
struct condition *available;
struct internal_state *mstate;
```

5. Building a Monitor in UNIX

It is necessary for the three components identified in the previous section to be shared by all processes using the monitor. Although it is possible to use only one shared memory segment, it seems simpler for each component to be a distinct shared memory segment. Each process must attach to the shared memory segments. Since this procedure is common to all processes, it is best kept as a function in an include file. Each segment requires a unique key number and a pointer. Attaching to just one of the segments looks like

```
securityid = shmget( (key_t)SECURITY,
                        sizeof(struct security),
                                0666 | IPC_CREAT );
*msecurity = (struct security *)
                        shmat( securityid, 0, 0 );
```

The entry and exit protocols for each interface procedure of the monitor are

```
mentry()
{
        P( (*msecurity).mutex );
        }
 mexit()
{
        if( (*msecurity).sig_count > 0 )
                V( (*msecurity).signaller );
        else
                V( (*msecurity).mutex );
        }
```

The semaphore *P* and *V* operations used here are those defined in [Rochkind, 1986] as a simplified (and more traditional) interface to UNIX semaphores.

6. Operations on Conditions

When a process must wait on a condition within a monitor, exclusive access is released before the process suspends itself on the semaphore associated with the condition. The *wait* operation is

```
cwait( cond )
struct condition *cond;
{
        (*cond).count++;
        if( (*msecurity).sig_count > 0 )
                V( (*msecurity).signaller );
        else
                V( (*msecurity).mutex );
        P( (*cond).q );
        (*cond).count--;
        }
```

The corresponding *signal* operation will suspend the calling process if a process was waiting on the condition and is consequently released. This ensures that only one process remains active within the monitor. The suspended signaller may be released by the exit protocol of an interface procedure.

```
csignal( cond )
struct condition *cond;
{
        if( (*cond).count > 0 )
        {
                (*msecurity).sig_count++;
                V( (*cond).q );
                P( (*msecurity).signaller );
                (*msecurity).sig_count--;
                }
        }
```

7. Interface Procedures

For the single resource monitor, only two interface procedures are necessary. To ensure the integrity of the monitor's internal state, each interface procedure must contain the entry and exit protocols. The interface procedures are

```
acquire()
{
        mentry();
        if( (*mstate).busy )
                cwait( available );
        (*mstate).busy = TRUE;
        mexit();
        }

release()
{
        mentry();
        (*mstate).busy = FALSE;
        csignal( available );
        mexit();
        }
```

8. Monitor Initialization

The monitor must be initialized before any user process can access it. All semaphores must have a unique identification key. The *semtran* function used below is from [Rochkind, 1986].

```
/* initialize condition */
(*available).count = 0;

/* initialize internal state */
(*mstate).busy = FALSE;

/* establish semaphores */
(*msecurity).mutex = semtran( MUTEX );
(*msecurity).signaller =
                                semtran( SIGNALLER );
(*available).q = semtran( AVAILABLE );
```

```
/* initialize security */
(*msecurity).sig_count = 0;
V( (*msecurity).mutex ); /* initially 1 */
```

Bear in mind that the shared memory segments and semaphores should be removed from the system when the monitor is no longer in use.

9. Priority Conditions

Some synchronization problems (such as the diskhead scheduling problem [Hoare, 1972]) are elegantly solved by employing prioritized scheduling of processes waiting on conditions. The next process to be released is the one with the highest priority (lowest priority number). The *priority wait* operation must include a parameter to indicate the priority number. The code for this operation utilizes an unconventional feature of UNIX semaphores. A process can be delayed until the semaphore value can be decremented by an arbitrary number, not just 1 (as with conventional semaphores). The *semcall* function is defined in [Rochkind, 1986].

```
prioritycwait( cond, priority )
struct condition *cond;
int priority;
{
        (*cond).count++;
        if( (*msecurity).sig_count > 0 )
                V( (*msecurity).signaller );
        else
                V( (*msecurity).mutex );
        semcall( (*cond).q, -priority );
        (*cond).count--;
        }
```

The corresponding *priority signal* operation for priority condition variables continually adds one to the value of the semaphore until a single process is released (this will be the one with the lowest priority number)[1].

```
prioritycsignal( cond )
struct condition *cond;
{
        int newval, oldval;

        if( (*cond).count > 0 )
        {
                (*msecurity).sig_count++;
                oldval = semctl(
                        (*cond).q, 0, GETVAL, 0 );
                V( (*cond).q );
                newval = semctl(
                        (*cond).q, 0, GETVAL, 0 );
                while( newval == oldval+1 )
                {
                        oldval := newval;
                        V( (*cond).q );
                        newval = semctl(
                        (*cond).q, 0, GETVAL, 0 );
                        }
                P( (*msecurity).signaller );
                (*msecurity).sig_count--;
                }
}
```

10. Conclusion

Monitors, conditions and their operations can be easily built in C with UNIX System V system calls. In addition, the special features of UNIX semaphores enable the implementation of priority conditions.

All internal variables of a monitor are necessarily provided in memory segments which are shared by the processes using it. However, the code sections comprising the interface procedures and operations on conditions must be housed in include files.

---

[1] It is preferable to use GETNCNT (rather than GETVAL) for this purpose but it was found to be unreliable!

REFERENCES

W. E. Dijkstra (1968). Co-operating Sequential Processes, *Programming Languages*, (ed. F Genuys), 43-112, Academic Press.

N. Dunstan (1989). Synchronization Problems and UNIX System V, *ACM SIGCSE Bulletin*, Vol. 21, No. 4, Dec. 1989.

C. A. R. Hoare (1974), Monitors: An Operating System Structuring Concept, *Communications of the ACM*, Vol. 18, No. 2, 1974.

R. C. Holt (1983), *Concurrent Euclid, the UNIX System and TUNIS*, Addison-Wesley, 1983.

R. C. Holt and J. R. Cordy (1988), The Turing Programming Language, *Communications of the ACM*, Vol. 31, No. 12, Dec. 1988.

J. L. Peterson and A. Silberschatz (1985), *Operating System Concepts*, Addison-Wesley, Second Edition, 1985.

M. J. Rochkind (1986), *Advanced UNIX Programming*, Prentice-Hall, 1986.

P. D. Terry (1985), *CLANG - A Simple Teaching Language*, ACM SIGPLAN Notices, Vol. 20, No. 12, Dec. 1985.

J. Welsh (1979), Pascal Plus - Another Language for Modular Multiprogramming, *Software Practice and Experience*, Vol. 9, 947-957 (1979).