Journal of Computer Science 2 (10): 765-769, 2006
ISSN 1549-3636
© 2006 Science Publications

pmonpp: Pthread Monitor Preprocessor

Jinhua Guo
Department of Computer and Information Science, University of Michigan, Dearborn, MI 48128, USA

Abstract: Monitors have become an exceedingly important synchronization mechanism because they are a natural generalization of the object-oriented programming. A monitor construct encapsulates private data with public methods to operate on that data. Although the Pthread library contains dozens of functions for threading and synchronization, it does not provide direct support for the monitor. Students must explicitly provide mutual exclusion around "monitor procedures" using mutex locks. However, monitor procedures by definition execute with implicit mutual exclusion. This makes it hard to teach the monitor concept in class and explain the semantic differences between monitors and semaphores. To solve this problem, we have designed and implemented a monitor preprocessor for Pthreads that provides explicit support for monitors in Pthreads.

Key words: Concurrent programming, mutual exclusion, condition synchronization, process synchronization

INTRODUCTION

Process (thread) synchronization is fundamental to concurrent programs and is one of the most difficult topics in an operating system course. Semaphores and monitors are two general mechanisms that are taught in the operating system course for solving synchronization problems.

Semaphores were the first and remain one of the most important synchronization tools. They make it easy to protect critical sections and can be used in a disciplined way to implement process synchronization. However, semaphores are also a low-level mechanism because it is unstructured. Shared variables and the semaphores that protect them are global variables. Operations on shared variables and semaphores are distributed throughout program. It is very difficult to determine how a semaphore is being used (mutual exclusion or condition synchronization) without examining all of the code. Furthermore, their incorrect use can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur[1].

Monitors were suggested by Dijkstra, then by Brinch Hansen[2] and then named and popularized by Hoare in a seminal 1974 paper[3] and somewhat lost favor in the 1980s and early 1990s. However, monitors have regained importance with the object-oriented programming languages, such as Java[4] and Microsoft C#. In fact, the Java and C# programming languages make extensive use of monitors to provide mutual exclusion and synchronization in multithreaded applications.

Monitors have become an exceedingly important synchronization mechanism because they are a natural generalization of the object-oriented programming, which encapsulate data and operation declaration with a class. A monitor construct is an abstract data type, which encapsulates *private data* with *public methods* to operate on that data. Mutual exclusion is provided implicitly by ensuring that procedures in the same monitor are not executed concurrently. Condition synchronization in monitors is provided explicitly by means of condition variables. This makes a concurrent program easier to develop and easier to understand. Because of their utility and efficiency, monitors have been employed in several concurrent programming languages, most recently and notably in Java[5] and C#.

The Pthread library[6] is a standard set of C library routines for the UNIX cross-platform multithreaded programming. Since Pthread contains dozens of functions for threading and synchronization, we recommend students to implement their projects of thread synchronization problems using the Pthread library. Unfortunately, Pthread does not provide direct support for the monitor although it provides "condition variables" and "mutex locks", which are part of a monitor. Students must explicitly provide mutual exclusion around "monitor procedures" using mutex locks. However, the semantics of monitors defined by Hoare provide for implicit mutual exclusion during the execution of any monitor procedure. This makes it hard to teach the monitor concept in class and explain the semantic differences between monitors and semaphores. We have found that students are reluctant to use monitors because of this confusion – and therefore often fail to master the monitor concept.

To solve this problem, we have designed and implemented a Pthread monitor preprocessor (pmonpp) for Pthreads that provides explicit support for monitors in Pthreads. The preprocessor allows users to use true

Corresponding Author: Jinhua Guo, University of Michigan, Department of Computer and Information Science, Dearborn, MI 48128, USA, Tel: (313) 583-6439, Fax: (313) 593-4256

monitors in Pthread programming. The user writes a monitor specification file, which contains a single monitor. This file is translated by the preprocessor into proper Pthread codes using only mutex locks to guarantee mutual exclusion access and condition variables to allow general condition synchronization.

Monitor: Semaphores are like goto's and pointers: mistake prone, work okay but lack structure and "discipline".
For example, a disastrous typo:
V(S); criticalSection(); V(S)
This leads to deadlock:
P(S); criticalSection(); P(S)
Inappropriate use of nested critical sections can lead to deadlock:
P1: P(Q); P(S); ... V(S); V(Q);
P2: P(S); P(Q); ... V(Q); V(S);

A monitor is an object with some built-in mutual exclusion and thread synchronization capabilities. They are an integral part of the programming language so the compiler can generate the correct code to implement the monitor. Only one thread can be active at a time in the monitor, where "active" means executing a method of the monitor. Monitors also have *condition variables*, on which a thread can *wait* if conditions are not right for it to continue executing in the monitor. Some other thread can then get in the monitor and perhaps change the state of the monitor. If conditions are now right, that thread can *signal* a waiting thread, moving the latter to the ready queue to get back into the monitor when it becomes free.

Under the general topic of multitasking management, the monitor concept offers a solution to the low-level nature semaphore usage. A monitor construct is a high-level concurrency synchronization abstract offering safe data consistency. A monitor guarantees only one active thread/ process with exclusive rights to access the defined monitor variables and monitor procedures. Unlike semaphores, the monitor abstract has implicit mutual exclusion guaranteed for its protected members and functions. The syntax of a monitor is shown in Fig. 1.

Monitors also include the concept of condition variables. A condition variable is used to delay a process that cannot safely continue executing until the monitor's state satisfies some Boolean condition. It is also used to awaken a delayed process when the condition becomes true. Condition variables used within a monitor have three basic, distinct methods: wait, signal and broadcast. A thread calling wait on a particular condition variable is placed into the queue associated with that condition variable; a thread calling signal causes a thread wait on that conditional variable to be removed from the queue. Broadcast removes all threads from the queue.

A monitor's implementation of handling this signaling and thread queuing has two possibilities.

```
monitor monitor_name {
    shared variable declaration;

    procedure body P1 (...) { ... }
    procedure body P2 (...) { ... }
    ...
    procedure body Pn (...) { ... }

    {
        initialization code
    }
}
```
Fig. 1: Syntax of a monitor

A signal-and-exit monitor requires a thread to immediately exit the monitor upon signaling. Alternatively, a signal-and-continue monitor allows a thread inside the monitor to signal that the monitor will soon become available, but still maintain a lock on the monitor until the thread exits the monitor[1].

In the multithreaded Java applications, monitors are the primary mechanism providing mutual exclusion and synchronization. The key word *synchronized* imposes mutual exclusion on an object in Java. Java monitors are signal-and-continue monitors[4].

Mutex locks and condition variables in Pthreads: Although Pthreads does not provide direct support for the monitor, it provides "condition variables" and "locks", which are part of a monitor.

Locks in Pthreads are called mutex locks – or simply mutexes – because they are used to implement mutual exclusion. A critical section of code uses mutex as follows:
pthread_mutex_lock(&mutex);
critical sections;
pthread_mutex_unlock(&mutex);

Condition variables in Pthreads are very similar to the condition variables described in the previous subsection. The main operation on condition variables in Pthreads are *wait, signal and broadcast.* These must be executed while holding a mutex lock.

The parameters to pthread_cond_wait are a condition variable and a mutex lock. A thread that wants to wait must first hold the lock. For example, suppose a thread has already executed
pthread_mutex_lock(&mutex);
and then later executes
pthread_cond_wait(&cond, &mutex);

This causes the thread to release mutex and wait on cond. When the process resumes execution after a signal or broadcast, the thread will again own mutex and it will be locked. When another thread executes
pthread_cond_signal(&cond);
it awakens one thread (if one is blocked), but the signaler continues execution and continues to hold onto mutex.

The above description for using conditional variables and mutex locks is very similar to a monitor procedure except that the users need to explicitly provide mutual exclusion.

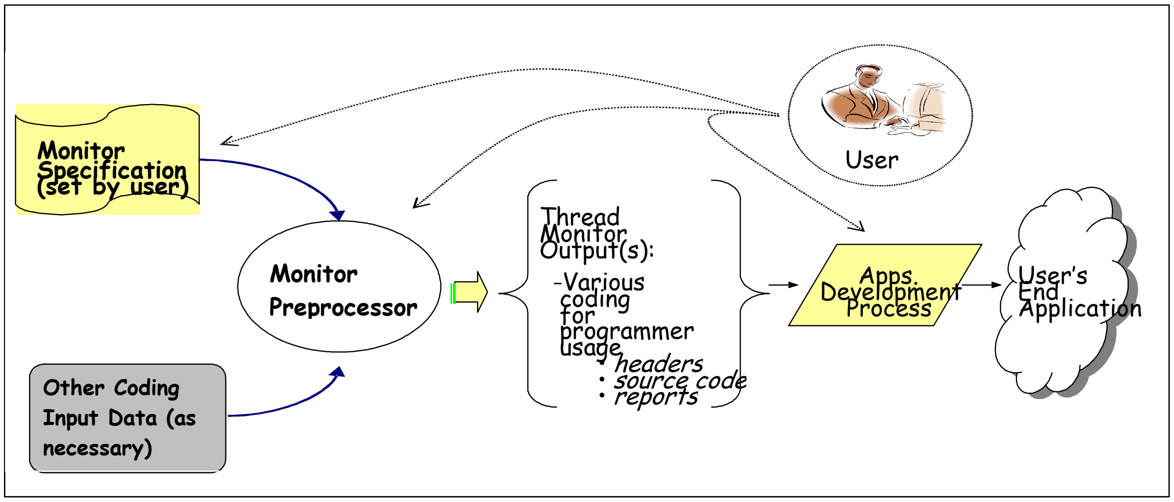
Monitor Specification (set by user)
Monitor Preprocessor
Other Coding Input Data (as necessary)
Thread Monitor Output(s):
-Various coding for programmer usage
• headers
• source code
• reports
User
Apps. Development Process
User's End Application

Fig. 2: An overview of the intended process

Monitors for Pthreads: As seen in the above subsection, a monitor procedure can be simulated using Pthreads by locking a mutex lock at the start of the procedure and unlocking the mutex at the end. Therefore, one straightforward solution to a monitor for Pthreads is a preprocessor application. A monitor preprocessor will take a programmer's monitor specification written in the syntax described in Fig. 1 and generate Pthread compliant stub files for an application's development in the programming language of ANSI C or C++. The desired monitor(s) will be implemented via mutex locks and condition variables. Figure 2 shows a graphical overview of the intended process.

Using a monitor preprocessor for producing these monitor stub files offers clear advantages of a straightforward creation process and a consist control over the monitor coding format. A similar preprocessor using a non-standard operating system has been created for a teaching aid in upper-level computer science course[7].

The alternative to the preprocessor would be to develop a common class or C++ template for monitors. Unfortunately, handling of improper object-orientated issue(s) such as inheritance and public method/ data make create a monitor class/ C++ template difficult to overcome.

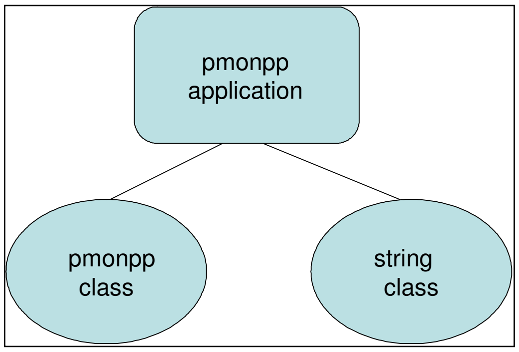

Fig. 3: Classes of the monitor preprocessor

Table 1: The errors detected by the monitor preprocessor

No.	Error Condition(s)
1	Missing command line argument
2	Missing keyword 'monitor' declaration
3	Multiple monitor keyword detected
4	Missing initialize() keyword
5	Missing destroy() keyword
6	Keyword detected in wrong section
7	Missing left brace '{' after keyword found
8	Missing right brace '}' or '};'
9	Inconsistent declaration of condition variables (total count mis-match)
10	Missing right parentheses ')'
11	File open error
12	File creation error
13	File read error
14	File write error

Monitor specification: Monitors are simply an abstract construct that encapsulates shared variables and methods and have a formatting syntax that is similar to a class definition (Fig. 1). Using a similar style of formatting, an example of monitor's specification is shown in Fig. 4.

The monitor specification developed for this application allows only one monitor to be defined per input file. The Pthread Monitor Preprocessor application uses this file to create the source output files. The output file names (*.h, *.cc) will have the same base filename as the monitor specification file. For instance, if the preprocessor is invoked with bounded_buffer.mon, the output files created would be bounded_buffer.h and bounded_buffer.cc. The output files form monitor class with POSIX compliant commands.

Each monitor specification file must declare only one monitor with the noted keyword. Subsequently, all data members defined (after the keyword monitor) will be made private members of the monitor class.

```
monitor bounded_buffer {
    typeT buf[n];      // an array of some type
    int front = 0;
    int rear = 0;
    int count = 0;
    initialize() {
        Condition_var not_full;
        Condition_var not_empty;
    }
    destroy() {
        Condition_var not_full;
        Condition_var not_empty;
   }

    sync_functions() {
        public void produce(typeT data) {
            while (count == n)
                wait(not_full);
            buf[rear] = data;
            rear = (rear + 1) % n;
            count++;
            signal(not_empty);
        }
        public void consume(typeT &result) {
            while (count == 0)
                wait(not_empty);
            result = buf[front];
            front = (front + 1) % n;
            count--;
            signal(not_full);
        }
     } //end of sync_functions
} // end monitor
```

Fig. 4: Monitor specification file: bounded_buffer.mon

```
void boundedBuffer_class::produce(typeT data)
{
    pthread_mutex_lock(&bmutex);
    while (count == n)
        pthread_cond_wait(&not_full, &bmutex );
    buf[rear] = data;
    rear = (rear + 1) % n;
    count++;
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&bmutex);
}

void boundedBuffer_class::consume(typeT
&result){
    pthread_mutex_lock(&bmutex);
    while (count == 0)
        pthread_cond_wait(&not_empty, &bmutex
    );
    result = buf[front];
    front = (front + 1) % n;
    count--;
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&bmutex);
}
```

Fig. 5: The generated produce and consume functions in pthreads

As shown in Fig. 4, four key sections are defined in the specification:
initialize()
destroy()
sync_functions()
unsync_functions()

The initialize and destroy sections place the declarations of the monitor specification file into the constructor and destructor of the monitor class. Any condition variables declared in the monitor must be defined in both of these sections.

Then the synchronized functions, those with automatic mutual exclusion, are defined. As a final option, the monitor specification allows for unsynchronized functions to be declared, which means the procedure is executed as a regular procedure call.

Preprocessor implementation: The Pthread Monitor PreProcessor (pmonpp) is constructed using object oriented program design and is implemented with C++. The main() function simply looks for a proper command line input and then initiates the *pmonpp* object. Two main classes of the preprocessor are the *pmonpp* class and the *string* class. Figure 3 shows the high-level object overview.

The *pmonpp* class is the primary handler for the Pthread Monitor Preprocessor application. The bulk of the work begins with the "kick-off" of the constructor method. This constructor sets up the output files and in-turn initiates the parsing operation. The parsing operation begins by looking for keywords and upon detecting a keyword, handlers are called for continuing the processing the designated section. The primary parser routine is implemented as a five level if-then-else structure searching for each section.

A *string* class was created since a strings class or template is not standard with all ANSI C++ compilers. The custom string class ensures consistent treatment of string operations. Essentially, this class provided basic token and "find" methods to enable the parsing operations mentioned as part of the *pmonpp* class.

A. Expected input: Only the monitor input file is required for proper operation; no other input is utilized for this application. When the user invokes the *pmonpp* application, a filename containing the monitor specification must be provided as well. Proper formatting of the monitor specification is assumed otherwise, as previously discussed, the note error condition will be displayed back to the user. The default line length (a supplied setting or constant of the *pmonpp* program is 255 characters per line); lines longer then this length will be clipped unless the preprocessor application is re-compiled with an adjusted value.

B. Expected output: As briefly mentioned, the successfully output of the *pmonpp* applications are two source code files. The output header file contains the monitor class declaration and the private data members. The monitor’s name is provided from the specification file. This header file also lists the class prototypes for the corresponding implementation file (*.cc).

Similarly, the implementation file contains the associated methods for the monitor class with all functions in the “sync” or “unsync” section made into class methods. The class constructor by default will create the appropriate Pthread mutex with its name derived from the monitor's; the destructor does the same as well.

Error conditions: The *pmonpp* application implements multiple error handlers. The default action upon detecting an error is to shutdown the application with a message supplied to default error pipe (i.e. the user's display screen).

The Table 1 lists many of the errors detected by the *pmonpp* operations and processing; again the end-action to halt operation for all detected errors. The *pmonpp* application only processes the defined keywords. No guarantee is made for the monitor's correctness, which is totally dependant upon proper coding placed into the specification file.

Algorithm analysis overview: The creation of the *pmonpp* object begins with the primary function of parsing the input file. This input file is parsed in a single line, linear fashion. By linear, meaning the tokens or keywords are determined in a forward fashion with no back up operations.

Many different mechanisms could have been implemented for the parsing operations, but after reviewing the Pthread Monitor Preprocessor requirements, the parse operations were deemed to fairly unique to the monitor specification. The coding efficiency could be improved by creating a centralized token recognition function.

Example: Here, we will demonstrate how to use the preprocessor to solve synchronization problems through a well-known bounded buffer problem.

The bounded buffer problem is commonly used to illustrate the power of synchronization primitives. A producer and a consumer process communicate by sharing a buffer having n slots. The buffer contains a queue of messages. The producer sends a message to the consumer by depositing the message at the end of the queue. The consumer receives a message by fetching the one at the front of the queue. Synchronization is required so that a message is not deposited if the queue is full and a message is not fetched if the queue is empty.

To solve this synchronization problem, the programmers just need to write a monitor specification in the syntax described in Fig. 1. The specification file bounded_buffer.mon is shown in Fig. 4. The monitor preprocessor will then take the programmer's monitor specification and generate Pthread compliant stub files in the language of C++. In Fig. 5, we show the produce and consume Pthread functions generated by the Pthread monitor preprocessor.

CONCLUSION

Concurrent programs are harder to write, debug, modify and prove correct than non-concurrent programs. Monitors are a high-level synchronization construct that provides more structure than semaphores. A monitor construct is an abstract data type, which encapsulates private data with public methods to operate on that data. Mutual exclusion is provided implicitly by ensuring that procedures in the same monitor are not executed concurrently. Condition synchronization in monitors is provided explicitly by means of condition variables. This makes a concurrent program easier to develop and easier to understand.

We have designed and implemented a monitor preprocessor for Pthreads that provides explicit support for monitors in Pthreads. The preprocessor allows users to use true monitors in Pthread programming. With the monitor preprocessor, the users can write their monitors in the syntax similar to the original Hoare's specification and without worrying the mutual exclusion. These monitor files can then be translated by the preprocessor into proper Pthread codes using only mutex locks to guarantee mutual exclusion access and condition variables to allow general condition synchronization.

The Pthread Monitor Preprocessor (pmonpp) has been used and tested by students in the operating system class since the fall 2004. The preprocessor make it easier to write concurrent programs using Pthreads for solving synchronization problems, it also can help students to have a deep understanding about how a monitor is implemented by analyzing the generated codes. The interested readers may find more about our work, including *pmonpp* software and documents at the following site:
http://www.engin.umd.umich.edu/~jinhua/pmonpp

REFERENCES

1. Sihberscatz, A., P.B. Galvin and G. Gagne, 2003. Operating System Concepts. John Wiley and Sons.
2. Hansen, B., 1993. Monitors and concurrent Pascal: A personal history. ACM SIGPLAN Notices. The second ACM SIGPLAN Conf. History of Programming Languages, 28: 3.
3. Hoare, C.A.R., 1974. Monitors: An operating system structuring concept. Comm. ACM, 17: 549–557.
4. Hyde, P., 1999. Java Thread Programming. Sams Publishing.
5. Gregory, R.A., 2000, Foundations of Multithreaded, Parallel and Distributed Programming. Addison Wesley.
6. Lewis, B. and D. Berg, 1998. Multithreaded Programming with Pthreads. Mountain View, CA: Sun Microsystems Press.
7. Hatcher, J. and D. Lowenthal, 1999. monpp: Monitors in Nachos. Technical Report TR-99-7. Department of Computer Science, University of Georgia.