STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

DETECTING DEADLOCKS IN MULTITHREADED APPLICATIONS

Simona Daniela IURIAN[*]

Dedicated to Professor Sever Groze on his 65th anniversary

Received: February 8, 1995
AMS subject classification: 68Q05, 68Q10, 68Q60, 68Q90

REZUMAT. - Detectarea impasului în aplicaţii cu mai multe fire de execuţie. Este prezentată o modalitate de descriere a aplicaţiilor cu fire de execuţie multiple, bazată pe reţele Petri. Folosind acest model şi rezultate din teoria reţelelor Petri se dă un algoritm de detectare a impasului într-o astfel de aplicaţie.

A threads package

We introduce a specification of a threads package, which will be referred further in this paper for describing and analysing a multithreaded application. This package is an extension of the 'C' language and is suggested by the threads library from Windows NT (see [3]). So, for developing multithreaded application we will use 'C' enriched with few data types and functions, which are listed bellow.

The new data types and constants are:

typedef BYTE *PThreadID;	A handle for the thread
typedef FARPROC *PThreadFunction;	The function which contains the code of a thread
typedef int TCriticalRegion	Data type for a condition variable
typedef void far * TEvent	Data type for an event

The functions which are interesting for our purpose are presented bellow:

· PThreadId <u>ThreadCreate</u>(TThreadFunction ThreadFn)

[*] *"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, 3400 Cluj-Napoca, Romania*

The call of this function try to create a new thread. The returned value is a handle for the new thread and could be used in the future to refer it. The parameter ThreadFn indicates the code which will be executed by the new thread.

· void EnterCriticalRegion(TCriticalRegion RC)

A thread has to call this function if it needs to enter the critical region RC (see [2]). If the critical region RC is not available, the thread will wait for the other thread which executes RC to release it.

· void ReleaseCriticalRegion(TCriticalRegion RC)

A thread call this function if it finished the execution of the critical region RC. The critical section RC become available to another thread.

Also, it can be defined other function which permits to stop the execution of a thread, to resume the execution of a stopped thread and for synchronize multiple threads by waiting for the ocurrence of an event.

Writing a multithreaded application as a Petri Net

In this section we will discuss the synchronisation mechanisms and the creation and termination primitives which were introduced until now in terms of Petri nets. The analogies will be used to write a multithreaded application as a Petri net. The resulting Petri nets can be used to analyse the properties of a multithreaded application.

An application with a single thread can be written as a Petri net by associating to any instruction a transition and introducing a place between any two consecutive transitions, an initial place before the first transition and, (if any) a final place after the last(s) transition(s). A place define the state of the application (in fact, of the thread) and a transition represent an action to be taken. At the beginning is marked just the initial place of the Petri net, with a single token. At any moment, the marked place indicate the point where the execution of the thread is arrived.

The Petri nets can be used as an abstracting mechanism. Most of the instructions are irrelevant for our purpose. The only interesting operations are the threads synchronisation primitives and the control structures, if the different branches contains such synchronisation primitive or are creating new threads. So, we will ignore the uninteresting operations: the assignments and the calls of 'C' library functions. Varshavsky, in [5] presents a possibility to describe a sequential program as a Petri Net.

We will associate to a multithreaded application a Petri net in which the places may contain zero or a single token (this is a condition/event system, see [2] for details). Reisig, in [2], defines the notion of invariant in a Petri net. This is a vector I for which M*I=0, where M is the incidence matrix associate to the Petri net. It is shown that, for each case C of the Petri net and for each invariant I, the scalar product C*I is constant.

In order to write a multithreaded application as a Petri Net we need to define the configurations of transitions/places which corresponds to thread control primitives and have to be added to those described in [5].

ThreadCreate

In terms of Petri Nets, a thread can be viewed as a linear sequence of places and transitions. Each thread has a start place. The start place of the main thread will be marked. The start place of the others threads will be initially not marked.

If a thread creates a new thread, it will be created a new execution sequence corresponding to the newly created thread. This situation is depicted in figure 1a. If a thread executes the function ThreadCreate, the first place of the new thread and the next place of the old thread will become marked and further the two threads will be executed in parallel. It follows then, in a multithreaded application there could be more than one place marked at a moment.

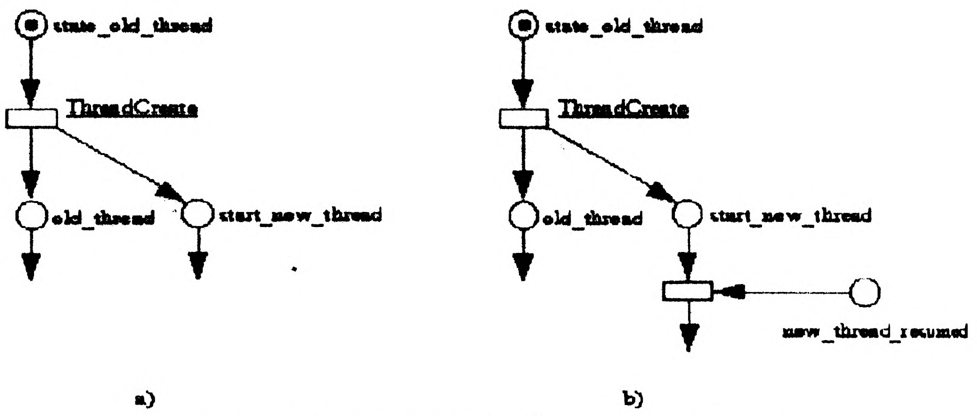

Figure 1. ThreadCreate and critical regions as Petri Nets

The critical region

If more than one thread needs to execute codes wh· h are mutually exclusive, they have to be synchronised. Since, the entering of such a code will begin with a call of EnterCriticalRegion, and the leaving of such a code will be marked by a call of ReleaseCriticalregion. The corresponding Petri net is depicted in figure 1b.

Detecting deadlocks in a multithreaded application

The deadlock is a very acute problem which can appear in a multithreaded application (see [4]). A deadlock is a state in which two or more threads wait each other to release a shared resource which may not be accesed simultaneously by the two threads. Because all threads participating to a deadlock are suspended and cannot release the shared resource, the whole application is blocked.

The next example describe a possible deadlock and will be used from now on.

```
TCriticalRegion rc1, rc2;

main()
{
        int Thread2ID=ThreadCreate(Thread2);
        while (1)
        {
                EnterCriticalRegion(rc1);
                EnterCriticalRegion(rc2);
                ReleaseCriticalRegion(rc2);
                ReleaseCriticalRegion(rc1);
                // do something;
        }
}
int Thread2()
{
        while (1)
        {
                EnterCriticalRegion(rc2);
                EnterCriticalRegion(rc1);
                ReleaseCriticalRegion(rc1);
                ReleaseCriticalRegion(rc2);
                // do something;
        }
}
```

It is obvious that there is a situation which represents a deadlock: when the main thread enter the critical region rc1, and the second thread enter the critical region rc2 at the same time. Then, the main thread wants to enter the critical region rc2, which is owned by the second thread. Also, the second thread waits for the releasing of the critical region rc1 which is blocked because of the main thread, and so on.

Generally, the deadlock is not as easy to detect. For detecting deadlocks there are two strategies: the posthumous way and the use of the invariants. The posthumous way consist of describing the application and executing it until it seems to be appeared a deadlock. After this, the problem is solved and the method is reitered.

In order to present the invariants method we will rewrite the previous example.

```
int Thread2()
{
        while (1)
        {
                EnterCriticalRegion(rc1);
                EnterCriticalRegion(rc2);
                ReleaseCriticalRegion(rc2);
                ReleaseCriticalRegion(rc1);
                // do something;
        }
}
```

Now, it is "obvious" that there could not appear a deadlock. For showing that we will enunciate a few evident propositions related to the flow control in the new application:

1. The critical region rc1 is either free or is accesed by Thread1, or is accesed by Thread2 (it follows from the definition of a critical region).
2. If the critical region rc1 is free, then Thread1 and Thread2 are executing the code from the beginning of the corresponding loop.
3. If the critical region rc1 is accesed by Thread1 (respectively Thread2), then Thread2 (respectively Thread1) is executing the code from the beginning of his while instruction, or is waiting for releasing the critical region rc1.
4. The critical region rc2 can be requested only after the critical region rc1 is accesed.
5. If the critical region rc2 is accesed by a thread, then this thread execute something between EnterCriticalRegion(rc2) and ReleaseCriticalRegion(rc2), and the other thread execute the code from the beginning of his while instruction or is waiting for the releasing of the critical region rc1.

These propositions are named invariants, because they are true at any moment of the application's execution. In a deadlock situation (from the definition), a thread A is suspended waiting the releasing of a critical region RC which is accessed by the thread B ($A \neq B$). The thread B must be waiting for the releasing of the critical region RC' (RC'≠RC) by the thread A. But the two threads cannot be simultaneously in a critical region (it follows from the above propositions), so a deadlock cannot appear.

To show formally that a multithreaded application is deadlock-free we may use the next algorithm:

1. Writing the multithreaded application as a Petri net.
2. Computing the Petri net invariants.
3. Generating all possible cases for the Petri net.
4. For each generated case:
    - Verifying if the case can be obtained from the initial case
    - If so, verifying if it represents a deadlock situation
    - If so, trying if that case verify the invariants. If all invariants are verified, then it is possible to appear a deadlock situation. If there is at least an invariant which is not verified by the case, then the multithreaded application cannot lead to the localised deadlock.
5. If there is at least a case, representing a deadlock, which can be obtained from the initial case and is verifying the application invariants, then there is possible to appear a deadlock in the execution of the application. If there is not such a case, then the application cannot lead to a deadlock situation.

For our example, the above steps are detailed further.

The Petri net for our initial application is depicted in figure 2.

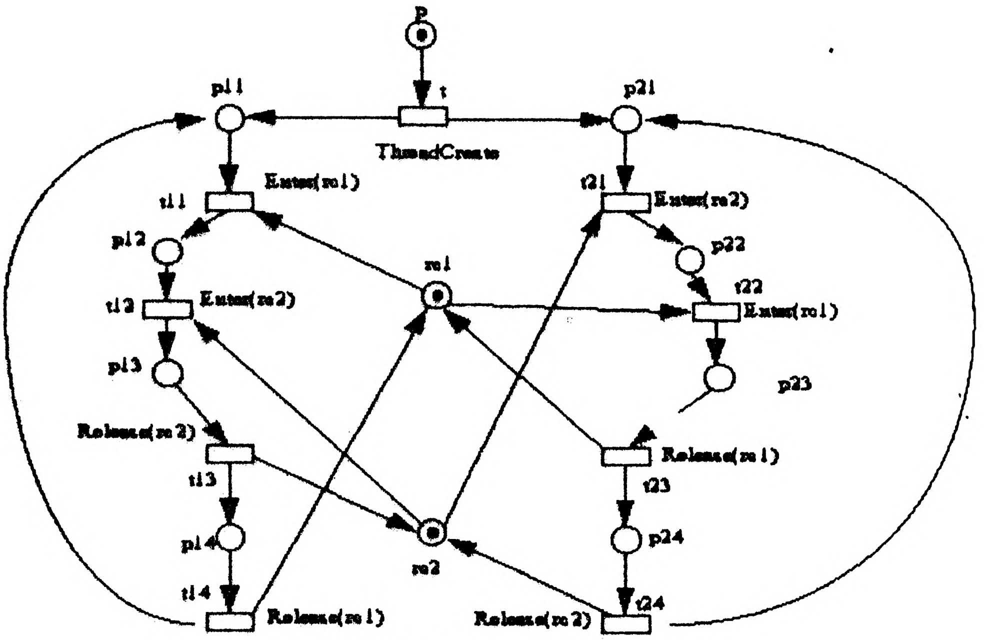

Figure 2. A Petri Net for our application

The incidence matrix (see [2]) M for this Petri net is the following:

	p	p11	p12	p13	p14	rc1	rc2	p21	p22	p23	p24
t	-1	1	0	0	0	0	0	1	0	0	0
t11	0	-1	1	0	0	0	-1	0	0	0	0
t12	0	0	-1	1	0	-1	0	0	0	0	0
t13	0	0	0	-1	1	1	0	0	0	0	0
t14	0	1	0	0	-1	0	1	0	0	0	0
t21	0	0	0	0	0	-1	0	-1	1	0	0
t22	0	0	0	0	0	0	-1	0	-1	1	0
t23	0	0	0	0	0	0	1	0	0	-1	1
t24	0	0	0	0	0	1	0	1	0	0	-1

The solutions I of the equation M*I=0 are the basis of the vector space that contains all invariants of the application (see [2]). In our case, the solutions are:

No.	p	p11	p12	p13	p14	rc1	rc2	p21	p22	p23	p23
1	1	1	1	1	1	0	0	0	0	0	0
2	1	0	0	-1	0	-1	0	1	0	0	0
3	-1	-1	0	0	0	0	1	0	0	1	0
4	1	1	0	1	0	1	-1	0	1	0	1

The 11 columns represents the 11 places. The meaning of an invariant is that the sum of the tokens from the corresponding places, multiplied by the number, is constant. In other words:

p+p11+p12+p13+p14 = constant

p+p21-p13-rc1 = constant

-p+rc2+p23-p11 = constant

p+p11+p13+rc1-rc2+p22+p24 = constant

By taking into account the initial case for the Petri net (that shown in figure 3) we can compute the real value of the constants from the above relations. So, because in the initial case, p=1, rc1=1, rc2=1 and the others places have no tokens (for those, the value is 0) we obtain the following relations:

p+p11+p12+p13+p14 = 1

p+p21-p13-rc1 = 0

-p+rc2+p23-p11 = 0

p+p11+p13+rc1-rc2+p22+p24 = 1

A case which represent a deadlock is <p12, p22>. For showing this we can construct the resource allocation graph (see [4]) and finding a cycle in this. For that case, it is very easy to see that the invariants are satisfied. Unfortunately, the verification of the invariants is a condition necessary but not sufficient for a case to be attained by starting from the initial case.

We can conclude that in our application it may occurs this deadlock.

REFERENCES

1. Krishnamurty, E.V., "Parallel Processing. Principles and Practice", Addison-Wesley Publishing Company, 1989
2. Reisig, W., "Petri Nets. An Introduction", Springer-Verlag, Berlin, 1985.
3. Richter, J., "Advanced Windows NT", Microsoft Press, 1994.
4. Tannenbaum, A., "Modern Operating Systems", Prentice-Hall, 1992
5. Varshavsky, V., "Self-Timed Control of Concurrent Processes", Kluwer Academic Publishers, 1990.