The Synchronization Mechanisms of The Parallel Object-Oriented Programming Language WARASA

2H−4

Yun JIANG, Akifumi MAKINOUCHI
Dept. Computer Sci. & Comm. Eng., Kyushu University

1 Introduction

We are developing an object oriented programming language called WARASA for writing parallel object oriented program on Mach based shared memory multiprocessors[JIA91].

WARASA is an extension to C++[STR86] by adding capability of parallelism. It attempts to combine the parallel mechanism based on shared memory multiprocessor with the object oriented paradigm, so as to allow programmers for implementing multithread programs and the interthread communication directly and easily.

In this paper, we describe one of the important features of WARASA, that is the synchronization abstraction.

2 Synchronization Mechanism in WARASA

The unit of parallel excution in WARASA is an object called autonomous object. When these autonomous objects run in parallel on different threads, they need synchronization and cooperation, for which some mechanism must be provided.

WARASA provides objects called synchronous objects for supporting synchronization and cooperation basically. The synchronization mechanism that supports the object synchronization must be simple on one hand, but efficient on the other hand.

In order to achieve this goal, three versions of synchronization mechanisms have developed in WARASA.

3 The Three Versions

In this section, the definitions and functions of three versions will be described using some examples.

3.1 Simple style

Simple style version uses the primitive synchronous mechanism of WARASA. The private data in a synchronous object only includes the condition variables whose type is system dependent. The methods definited for the object are simple synchronization operation without any codition judgment. If it is used for implementing synchronization in an application, users have to design another object called exclusive object in which shared variables are encapsulated, and both types of the objects must be used in pairs.

For example, in the consumer and producer problem, a synchronous class called push_pop_condition class is defined as follows:

```
sync class push_pop_condition{
  private:
    condition_t   push_available_sig;
    condition_t   pop_available_sig;
  public:
    push_pop_condition( );
   ~push_pop_condition( );
    void wait_pop_available( );
    void wait_push_available( );
    void make_pop_available( );
    void make_push_available( );
}
```

Further an exclusive object called stack has to be defined. When the stack is concurrently used by autonomous objects called consumer and producer, in order that these objects behave cooperatively, both push_pop_condition object and stack object have to be used in pairs. In order to pop data from the stack, the consumer program has to be written as follows:

```
pop_r = stack_OID -> stak_pop( );
while (pop_r == 0) {
    condition_OID -> wait_pop_avaiable( );
    pop_r = stack_OID -> pop( );
}
condition_OID -> make_push_avaiable( );
```

The synchronization process is depicted in Figure 1.

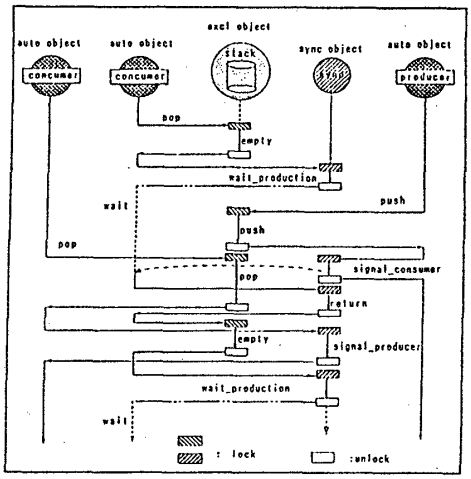

Figure 1. A synchronous Object in Simple Style Version

In the simple style version, the action on every object is very primitive, but a pair of a synchronous object and an exclusive object have to be always used. In addtion, a lot of locking/unlocking operations have to be executed.

3.2 Mixed style

We provide a mixed style version so as to improve the performance of the version mentioned above. In order to reduce the locking/unlocking operations, we combine the simple synchronous object with a simple exclusive object, to make other complex synchronous object in the mixed style version.

The private data of a synchronous object of this style is

The Synchronization Mechanisms of A Parallel Object-Oriented Programming Language WARASA
Yun JIANG, Akifumi MAKINOUCHI
Faculty of Engineering, Kyushu University

both condition variables and shared variables. Therefore, the operations on the condition variables and the ones on shared variables are also mixed in order to be the operations for one object. For example, a synchronous object called fork used in dining philosophers' problem is defined as follows;

```
sync class fork{
    private:
        int forks[5];
        int Yellow_card[5];
        condition_t fork_usable_Philo[5];
    public:
        fork( );
        ~fork( );
        int PickUp_fork(int Philo_num, int wait_times);
        void PutDown_fork(int Philo_num);
}
```

If we rewrite the style for the consumer and producer problem, in the mixed style,the consumer program that pop data from the stack above becomes very simple, that is:

```
pop_data= stack_oper_OID -> pop ( );
```

The mixed style is better than the simple style,because in the first, not only the autonomous class program becomes very simple but also the cost for locking/unlocking operations can be reduced. We can compare Figure 2. with Figure 1.

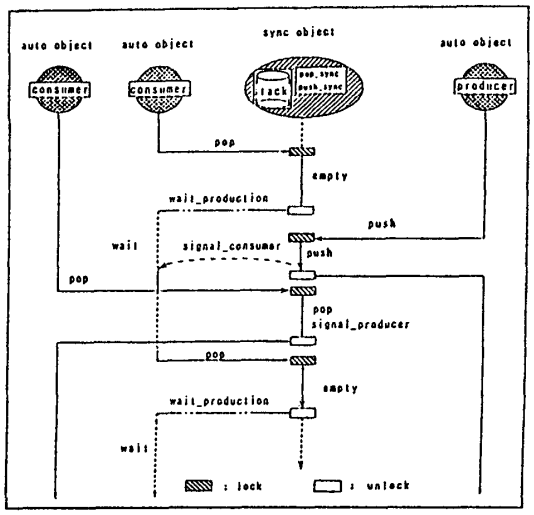

Figure 2. A synchronous Object in mixed Style Version

But the common problem in both styles is that users have to use the system-defined operations of the Mach operating system, when they design their synchronous classes. Users have to understand the syntax and semantics of these operations, before they use them, which is trublesome. So, we provide a inheritance style version.

3.3 Inheritance style

Inheritance style version is based on inheritance of C++. The Mach's system-defined operations are encapsulated in a superclass which becomes in turn a WARASA's system-defined class.

When the user needs to define a synchronous class in his own, he dosen't not to know the Mach's system-defined operations. The only thing that he has to know is how to inherit the synchronous superclass existing in WARASA.

For example, the synchronization mechanism used for a parallel hash-join algorithm uses the inheritance style. A parent_join_sync class is defined as a superclass, which is provided by WARASA, and a join_sync class is its derived class, designed by the user. The definition of the join_sync class is as follows:

```
sync class join_sync: public parent_join_sync{
    private:
        int bucket_sum_n;
        int bucket_i;
        int auto_object_sum_n;
        int object_i;
        pare_join_sync parent;
    public:
        join_sync( );
        ~join_sync( );
        void wait_join_divid_finish_sync( );
        int  bucket_assign( );
}
```

Figure 3. Depictes the function of the join_sync object.

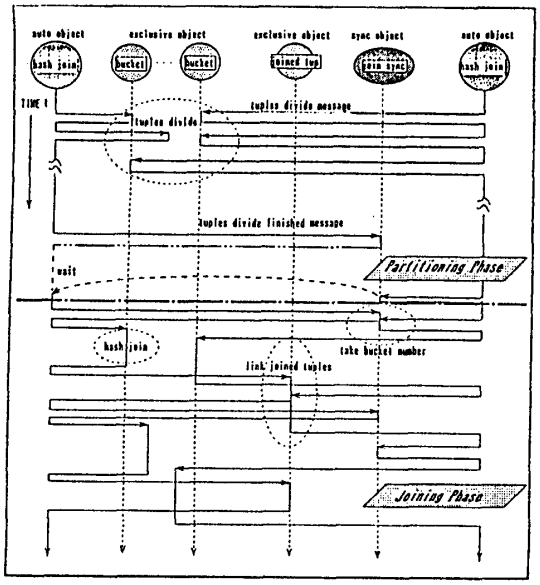

Figure 3. A Parallel Hash-join Algorithm

4 Conclusions

We have described the synchronization mechanism in WARASA. It plays an important role in supporting the autonomous objects that work cooperatively using high-level message passing. The determination of a more powerful synchronization mechanism version in designing WARASA is one of our current research subjests.

References

[JIA91] Y.Jiang and A.Makinouchi, "A Parallel Object-Oriented Persistent Programming Language WARASA," Information Processing Society of Japan, pp.4-181 - 4-182, Oct. 1991.

[STR86] B.Stroustrup, "The C++ Programming Language" Addison-Wesley, Reading, Mass., 1986.