An Implementation of an Operating System Kernel using Concurrent Object Oriented Language ABCL/c+

Norihisa Doi Yasushi Kodama*
Institute of Information Science, Keio University
4-1-1, Hiyoshi, Kohoku-ku, Yokohama, 223 Japan.

Ken Hirose
Department of Mathematics,
School of Science and Engineering,
Waseda University

Abstract

The ABCL/c+ is a C-based concurrent object-oriented language, designed as an extension of ABCL/1, a language developed by A.Yonezawa and others. In order to create the world of processes, a routine object is introduced which unifies procedures, functions, and objects. ABCL/c+ is then used to write an operating system kernel. The XINU operating system, developed by D. Comer and others of Bell Laboratories, is rewritten entirely in ABCL/c+. The result shows that concurrent object-oriented languages can produce a highly readable and understandable operating system kernel.

Key words and phrases: Concurrnet object-oriented language, Operating system kernel,ABCL.

1 Introduction

Object-orientation is a paradigm which enables us to produce highly reliable and reusable programs. A number of object-oriented languages have been developed and are currently being used, including Smalltalk-80 [7], Loops [1] and Esp [2]. Several **concurrent object-oriented languages**, with functions for concurrent programming, have also been developed, including ABCL/1 [10] [12], ConcurrentSmalltalk [11], Orient84/K [8], and BETA [9]. In these languages a one-to-one correspondence exists between an object definition and process, which is the unit for execution of an instance of the object. For example, in Smalltalk-80, everything is an object, and a process can be created with a block as its entity. Therefore, one of the reasons why it is difficult to write a concurrent program in Smalltalk-80 is that a process does not correspond to a class, which is the

*Current address: Nihon Sun Microsystems K.K., Ichibancho FS Bldg. 5F, 8 Ichiban-cho, Chiyoda-ku, Tokyo 102, Japan

unit component of a program [6]. In what are termed concurrent object-oriented languages, this correspondence is maintained, and concurrent processes can be described in a very natural way.

Since concurrent object-oriented languages can be used to describe concurrent processes easily, system programs and operating system kernels are attractive fields of application for them. However, aside from system programs, the "world of processes", i.e. an environment in which a process exists as the unit of activity, must be created to write an operating system kernel. A system kernel can be implemented as a process. This is not a good approach, however, even considering the ease of understanding and execution efficiency.

We extended the ABCL/1 language developed by A.Yonezawa et al., and developed a C-based concurrent object-oriented language, ABCL/c+ [5], which can be used to realize the world of processes, and tried to write an operating system kernel. We selected the XINU operating system [3] [4] developed by D. Comer and others of Bell Laboratories as the base, and rewrote it entirely in the ABCL/c+. As a result, it is shown that concurrent object-oriented language can produce very readable and understandable operating system kernels.

In the following, first, ABCL/c+ is described in section 2, then the relation between ABCL/c+ and C is sketched in section 3. Section 4 describes the development of the operating system kernel by using ABCL/c+, and finally the results are discussed in section 5.

2 Concurrent Object Oriented Language ABCL/c+

The ABCL/c+ is based on the language C, while the ABCL/1 language is based on Common Lisp. Therefore, ABCL/c+ is a language with types.

2.1 Object and message

Each object is in one of three **modes**: dormant, active, or waiting. Fig. 1 shows the transitions among the three modes [10]. Except for the routine object, which is discussed later, each object has two queues for retaining messages in the order of arrival. One is the queue for ordinary messages. The other is the queue for emergency messages, which can be accepted during the actions for an ordinary message, i.e. while the object is in the active mode. The actions specified for the ordinary message are temporarily suspended. The ordinary message is called an **ordinary mode message**, and the emergency message is called an **express mode message**. Their queues are called the **ordinary mode message queue** and **express mode message queue**, respectively. The message mode is explicitly identified by both the sender and receiver of the message. The pattern of acceptable messages, constraints, and the mode of the message are defined in an object.

Ordinary mode messages are processed by an object as follows (see Fig. 1). An object, when it is created, is in the dormant mode. If an acceptable message arrives, the object goes into the active mode, and starts performing the actions specified for the message. Any ordinary mode message arriving during the action is placed in the ordinary mode message queue. When the sequence of

actions is completed the object goes into the dormant mode. If messages are in the ordinary mode message queue, the object goes into the active mode again, and starts performing the actions. The select form (see 2.4) can be used during the actions to make the object go into the waiting mode and wait for the arrival of a specific message.

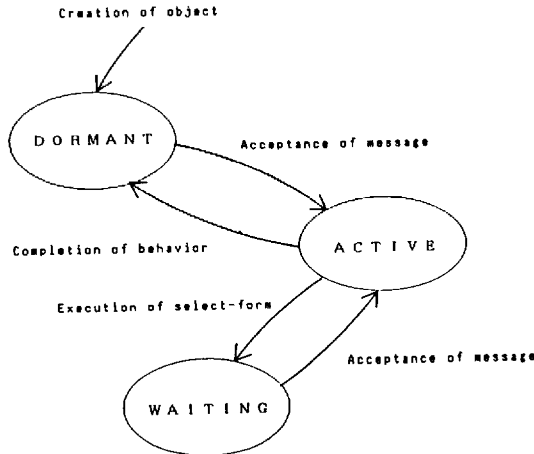

Fig. 1 Mode transition of an object

If a message arrives (or has arrived) while in the dormant mode, the object checks the ordinary mode message queue, starting from the first item, selects the first acceptable message, and deletes all the messages prior to it. If a message arrives (or has arrived) while in the waiting mode, the object checks the ordinary mode message queue, starting from the first item, selects the first acceptable message, and deletes it from the queue.

When an express mode message arrives, it is placed in the express mode message queue. It is not accepted if the object is accessing local memory, if the object is performing an action sequence specified by the atomic form (see 2.5), or if the object is performing the actions specified for an express mode message. Otherwise, if an express mode message is acceptable, the actions specified for the message are performed at once. Whether or not the actions for an ordinary mode message, which is interrupted by an express mode message, are to be resumed after the completion of the actions for express mode message can be specified by using the nonResume form (written as nonResume()). Unless the nonResume form is executed during the actions for the interrupting express mode message, the interrupted actions are resumed.

In addition to these objects, the ABCL/c+ has objects without any message queue, called **routine objects**. The routine objects are introduced to unify procedures, functions, and objects. They do not exist as a process. Routine object accepts only the call type message described later. A routine object can be defined alone, or its definition can be nested in an other routine or ordinary object. However, an ordinary object cannot be defined in a routine object. The routine object is introduced in order to write the operating system kernel for realizing concurrent objects, or the

world of concurrent processes. (However, the machine-dependent part, which cannot be written in C, must be written in another language, just as in the case where C is used. In the rewriting of XINU, described in section 4, assembly language was used for the machine-dependent part. Also, since an ordinary object exists as a process, objects which receive only the now type message cannot replace the routine objects.)

If a routine object is defined in an ordinary object, the routine object acts as an internal procedure or function, and corresponds to a routine in ABCL/1 [12].

The scope of the name in the nesting follows the scope rule of the block structure. The name which is in effect in the environment directly outside is in effect unless the same name is declared as a local name in an inner routine object.

2.2 Type of message passing

The types of message passing between objects include: past type, now type, future type, and call type for routine objects. The first three types are the same as those in ABCL/1 [10]. The semantics and syntax for message passing in ordinary and express modes are as follows.

(1) Past type

The sender object sends a message to the receiver object. If it has something to be processed, it continues the processing. If a reply should be sent to some other object, the destination object can be specified as the **reply destination**. The syntax for past type message passing is as follows({...} stands for the optionals):

Ordinary mode: [target object <= message {@ reply-destination}]

Express mode: [target object <<= message {@ reply-destination}]

(2) Now type

The sender object sends a message to the receiver object, and waits until a reply is returned. The syntax for now type message passing is as follows:

Ordinary mode: [target object <== message]

Express mode: [target object <<== message]

(3) Future type

The sender object, when it sends a message to the receiver object, specifies the "reply destination" in a variable called **future variable**, and continues its actions without waiting for the arrival of the reply. The reply sent to the specified destination can be accessed only by the sender of the message. When, at a later time, the reply is accessed, the commands "ready?(future-variable)" and "dequeueFuture(future-variable)" can be used to check to see if the reply has been sent or to take out the reply. The syntax for a future type message passing is as follows:

Ordinary mode: [target object <= message $ future-variable]

Express mode: [target object <<= message $ future-variable]

(4) Call type

The sender object sends a message to the receiving routine object, and waits until a reply is returned from the receiver. The syntax for call type message passing is as follows:

Ordinary mode: [target object <- message]

Express mode: [target object <<- message]

2.3 Object definition

In the ABCL/c+ ordinary objects are defined as follows:

```
Type/composite-type-declaration;
[object object-name
definition-of-routine-object;
                              . . .
state {
    type-of-state-variable state-variable:=initial-value;
                              . . .
}
script {
(=> message-pattern @ destination-variable where constraint # type
    type-declaration-of-pattern-variable; ... ;
    {
        declaration-of-temporary-variable; ... ;
        description-of-action; ... ;
    })
                              . . .
(=>> message-pattern @ destination-variable where constraint # type
    type-declaration-of-pattern-variable; ... ;
    {
        declaration-of-temporary-variable; ... ;
        description-of-action; ... ;
    })
                              . . .
}]
```

Routine objects are defined in the same form as ordinary objects except that -> and ->> are used instead of => and =>> to specify message patterns.

The following is a detailed explanation:

(1) 'State-variables' are variables which indicate the internal state of the object.

(2) An object accepts only those messages which match with 'message-pattern' and satisfy 'constraint.' Message patterns and constraints are checked from top to bottom. 'Reply-destination' and 'future-variable' are bound to 'destination-variable.' Reply is returned if the reply form is written in 'description-of-action.'

(3) When a message is accepted, actions defined in 'description-of-action' are performed sequentially. 'Temporary-variables' are those used by the object during the action. A temporary variable can be declared anywhere before the variable is referred to.

(4) The message pattern following '=>' is for an ordinary mode message, and the message pattern following '=>>' is for express mode message. Message patterns for call type messages are specified using '->' and '->>', respectively.

(5) 'Type' is the type of the value in the reply. Reply can be in any type allowed by the language C.

(6) 'Pattern-variable' is a variable used as an element in a message pattern. When a value is accepted as a message, the value is assigned to a pattern variable only if the type of the value and the type of the pattern variable coincide. 'Reply-destination' is also a pattern variable.

(7) The state part, destination-variable, constraint, and declaration-of-pattern-variable may be omitted.

2.4 Select form

The select form is specified as follows:

```
[select
    (=> message-pattern @ reply-destination where constraint # type
    type-declaration-of-pattern-variable; ... ;
    {
        description-of-action; ... ;
    }
                    . . .
    (=>> message-pattern @ reply-destination where constraint # type
    type-declaration-of-pattern-variable; ... ;
    {
        description-of-action; ... ;
    })]
```

If a select form is encountered in the description of actions, the object goes into the waiting mode, and waits for the message which matches with the message pattern specified in the select form. The first message which matches with the pattern and satisfies the constraint is accepted, and corresponding actions are performed. As noted earlier, the message accepted by the select form may have already arrived before the object goes into the waiting mode.

2.5 Atomic form

If a sequence of actions should not be interrupted by an express mode message, enclose the action sequence as follows:

```
atomic { description-of-action, ... , description-of-action }
```

Express mode message will not be accepted while the action sequence is performed.

2.6 Reply form

A reply can be sent back by using the following notation:

```
! value-form
```

Here, 'value-form' means variables other than a future variable, object definition form, now type message passing form, and primitive functions supplied by ABCL/c+.

2.7 Programming example

A simple example of programming in ABCL/c+ is shown in this section.

Example. A complex number is represented as an object. Fig. 2 shows an ABCL/c+ program which computes addition and subtraction between two complex numbers, the result being given as another object.

```
[object ComplexNumberCreator
 script {
  (=> [:New RealPart ImaginaryPart] # int
   float RealPart, ImaginaryPart;
    {
     ![object
      script {
       (=> [:Add ComplexNumber] # int
            int ComplexNumber;
            {
             ![ComplexNumberCreator <== [:New
              (RealPart +
               [ComplexNumber <== :GetRealPart])
              (ImaginaryPart +
               [ComplexNumber <== :GetImaginaryPart])]];
            })

       (=> [:Sub ComplexNumber] # int
            int ComplexNumber;
            {
             ![ComplexNumberCreator <== [:New
              (RealPart -
               [ComplexNumber <== :GetRealPart])
              (ImaginaryPart -
               [ComplexNumber <== :GetImaginaryPart])]];
            })

       (=> :GetRealPart # float
            {
             !RealPart;
            })

       (=> :GetImaginaryPart # float
            {
             !ImaginaryPart;
            })
      }];
    })
 }]
```

Fig. 2 ComplexNumberCreator

ComplexNumberCreator creates an object which represents a complex number. When a message :New is sent to the *ComplexNumberCreator* with *RealPart* (real part) and *ImaginaryPart* (imaginary part) as arguments, an object is created using the *RealPart* and *ImaginaryPart* as entities. The new object accepts messages :Add, :Sub, :GetRealPart, and :GetImaginaryPart. For example, 3+4i (object name is *compl1*) and 5+7i (object name is *compl2*) are created as follows:

```
compl1 := [ComplexNumberCreator <== [:New 3 4]];
compl2 := [ComplexNumberCreator <== [:New 5 7]];
```

The object representing the result of *compl1* + *compl2* can be created as follows:

```
[compl1 <== [:Add compl2]];
```

2.8 Necessity of routine objects

As indicated earlier, routine objects are introduced to unify procedures, functions, and objects. Routine object does not exist as a process. Therefore, its control mechanism and semantics are quite different from those of other objects. However, what is important is conceptual consistency, such as the existence of individual objects as the single substance, and execution of the task by message passing among the objects.

Conceptual consistency is most important when systems are developed. What is most important in the software implementation of the system is the architecture on which the concept is to be based and unified notation. Given the architecture and a unified notation we can avoid changing what we are thinking about, and therefore, we can develop systems easier to understand and with less errors.

Routine objects are introduced based upon these viewpoints, and they are not a deviation from the object-oriented concept.

2.9 Necessity of the express mode

Since the express mode causes one object to interrupt its actions and another to be executed, it is not very desirable when we consider the internal state of an object. However, in order to write an operating system kernel, we adopt the express mode in ABCL/c+ because it is very effective in the realization of input/output processing.

For instance, when we rewrote the XINU, input/output interruption is processed as an express mode message to the device handler corresponding to the interruption. For example, console manager (see Fig. 5) has two objects, one manages input from console, and the other manages output. They manage the input and output buffer, respectively. For instance, the input manager object accepts the message :GetC for taking out one character at a time in the order of their arrival from the input buffer, and the message [:Tty IIn Chr] for storing input characters in the input buffer (the input manager object can accept other messages). The message [:Tty IIn Chr] is an express mode message, while the message :GetC is an ordinary mode message. This is because it is necessary to store input data in the buffer as soon as possible, lest it should be lost.

The debugging of the objects which accept the express mode message is not different from ordinary interrupt processing. However, the simplicity of the notation should be noted.

3 Relation between ABCL/c+ and C

The C syntax can be used in the type/composite-type-declaration, state, and script parts, but not the function definition, which cannot be declared in C.

3.1 Type check

In ABCL/c+, types are checked either statically or dynamically depending on where the types are defined. Types defined in the type/composite-type-declaration part, which are global, and state and script parts, which are local, are checked statically by the C compiler. The types string, future, and list, which are introduced into ABCL/c+ especially for string manipulation, future variables handling and list manipulation respectively, are checked by the ABCL/c+ preprocessor.

But, the types of the components of a message are checked dynamically, when the message is tested against the messsage pattern as to whether it is acceptable or not. As message passing is done by byte-by-byte and as we would like to accept, for example, the value having type is short into the variable which type is long, only the types corresponding to the conversion characters which are used by stream I/O in the C are checked to the messages.

The type int is used for the type of a pointer to an object, because the value of a pointer to an object is the identification number of the object.

3.2 ABCL/c+ forms

Any kinds of forms of ABCL/c+, which are called ABCL/c+ forms, and statements of the language C can be used in the state and script parts. Among other things, two ABCL/c+ forms, object defining forms and value returning (now type) message passing forms, can be written on the right hand side of the assignment statement and assign the values to variables.

3.3 Alternation of the C syntax

As ABCL/c+ forms and C statements can be used together, the syntax of C is altered as follows [5]:

(1) Use assignment operator := instead of =.

(2) Use (* and *) instead of [and] as array subscript notations.

4 Description of Operating System Kernel

In order to verify the effectiveness of ABCL/c+, we tried to write an operating system kernel. The material chosen was the operating system kernel XINU [3] developed by D. Comer and others of Bell Laboratories. XINU is written in C, and consists of more than a hundred functions. The source file of XINU contains about 5850 lines of C code and 650 lines of assembly code. Deleting comments, it has approximately 4300 lines of C code and 550 lines of assembly code.

As shown in Fi. 3, the system has a 10 layer structure, including hardware and user programs.

Fig. 4 shows the relations among modules (functions) belonging to the process management layer and their data reference.

User programs

File system

Intermachine network communication

Device manager and device drivers

Real-time clock manager

Interprocess communication

Process coordination

Process manager

Memory manager

Hardware

Fig. 3 The layering of components in XINU

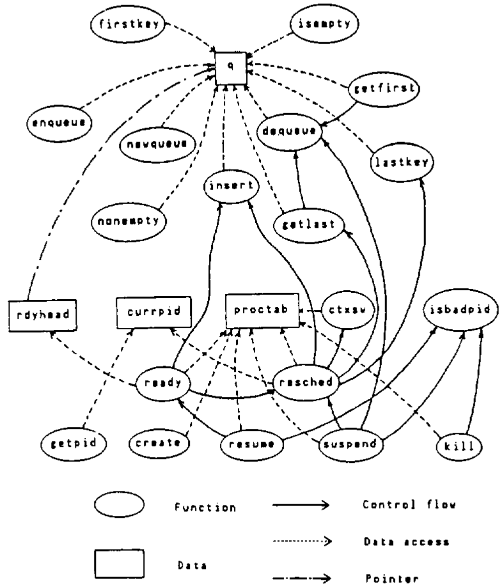

Fig. 4 The process manager layer of XINU

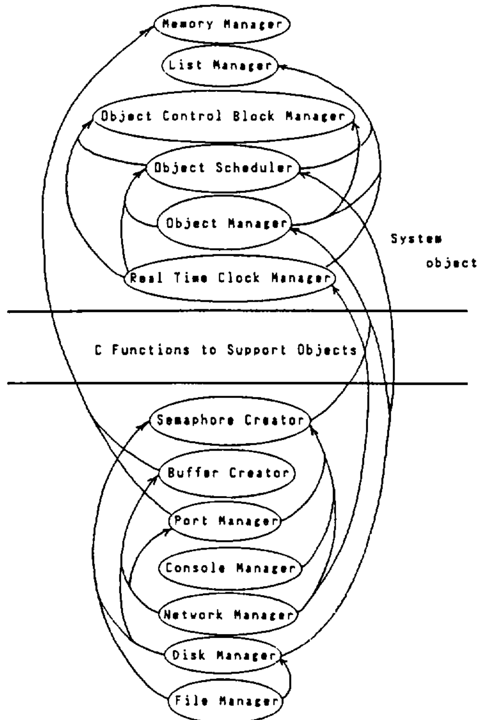

Fig. 5 Relationships between objects and message passing

Although they are structured in layers, it is not easy to understand the modules in such a complex relationship. We rewrote this system in ABCL/c+. Using basically the same data structure, the function of the new system is exactly the same as that of the old system. The objects and message passing relationships are shown in Fig. 5.

In Fig. 5 'memory manager' through 'real time clock manager' are the system objects which constitute the world of processes. Each of them is implemented as a routine object. 'Semaphore creator' and above are in the world of processes. Each of them corresponds to one or more objects (processes). 'C functions to support objects' in the middle are functions for object creation, and those for message passing. These C functions are automatically generated by ABCL/c+.

In the overall system, the command processor SHELL and ABCL/c+ compiler, which at present is the only language processor of our system, are placed on the kernel. When an object code of ABCL/c+ is executed, every time an object is created, 'C functions to support objects' are called, and they pass messages to and from system objects 'object manager', 'object scheduler', and 'object control block manager'. In so doing objects begin to exist as processes, and each process is executed in parallel by time-slicing. When an object (process) communicates with the console, it passes messages to and from 'console manager.' When it operates a file, it passes messages to and from 'file manager.'

In the following subsections, the two layers: 'process manager' and 'intermachine network communication' shown in Fig. 3 are discussed.

4.1 Process management

Generally speaking, this layer has four functions:

(1) list processing

(2) maintenance of process control block

(3) process schedule

(4) process management

List processing functions manage the ready list, semaphore list, and delta list. They use an array *q* which consists of three fields: priority, the predecessor, and the successor. The delta list is a list of sleeping processes in the order of the time at which they are to be awaken. These lists are bi-directional, with fields for the predecessor and successor. The second function is implemented as a table *proctab* for process management. The third function consists of the functions for scheduling and context switch using the ready list, and the variable *currpid* which maintains the identifier of the process currently being executed. The fourth function treats the creation/deletion and suspend/resume of processes.

The process control block *proctab* and the array *q* for lists are global data. For instance, *proctab* can be accessed from the upper layer 'interprocess communication' or from a still higher layer such as the 'real time clock manager' layer, as well as directly from this layer.

Using global data directly accessible or modifiable by other layers causes difficulties in the ease of understanding, in maintenance, and in revision.

Therefore, based upon functions and data, we divided this layer into the following routine objects:

(1) list manager

(2) object control block manager

(3) object scheduler

(4) object manager

Here, (1) (3) and (4) roughly correspond to (1) (3) and (4) above. However, the object control block manager, which consists of the object control block (process control block) and basic operations on it, did not exist explicitly in the original version.

```
[object Scheduler
 [object Schedule
  script {
   (-> :ReSched # int
        {
         int OldObjectID;

         if( ( [OCB <- [:ReferState :At CurrentObjectID]]
                 == CURRENT ) &&
           ( [List <- [:LastKey ReadyTailID]]
                 < [OCB <- [:ReferPriority :At CurrentObjectID]] ) )
          !OK;

         if( [OCB <- [:ReferState :At CurrentObjectID]]
             == CURRENT ) {
          [OCB <- [:PutState READY :At CurrentObjectID]];
          [List <- [:Insert CurrentObjectID ReadyHeadID
            [OCB <- [:ReferPriority :At CurrentObjectID]]]];
         }

         OldObjectID := CurrentObjectID;
         CurrentObjectID := [List <- [:GetLast ReadyTailID]];
         [OCB <- [:PutState CURRENT :At CurrentObjectID]];
         [Timer <- :Quantum];
         ctxsw( [OCB <- [:ReferObjectRegisters :At OldObjectID]],
           [OCB <- [:ReferObjectRegisters :At CurrentObjectID]] );
         !OK;
        })
  }]

  state {
   int CurrentObjectID;
   int ReadyHeadID;
   int ReadyTailID;
  }

 script {
  (-> :Initialize
        {
         ReadyHeadID := [List <- :NewQueue];
         ReadyTailID := ReadyHeadID + 1;
        })

  (-> [:Ready ObjectID ReScheduleBool] # int
        int ObjectID;
        Bool ReScheduleBool;
        {
         if( isbadpid( ObjectID ) ) !SYS_ERR;
         [OCB <- [:PutState READY :At ObjectID]];
         [List <- [:Insert ObjectID ReadyHeadID
          [OCB <- [:ReferPriority :At CurrentObjectID]]]];

         if( ReScheduleBool ) [Schedule <- :ReSched];
         !OK;
        })

  (-> :ReSchedule
        {
         [Schedule <- :ReSched];
         !OK;
        })

  (-> :ReferCurrentObjectID # int
        {
         !CurrentObjectID;
        })
 }]
```

Fig. 6 Object scheduler

For instance, object scheduler is as shown in Fig. 6. This takes the form of nested routine objects. When the message :Initialize is accepted, a message is sent to the list manager, *List*, to create the ready list. The first and last entries of the list are assigned to *ReadyHeadID* and *ReadyTailID*, respectively. From the message [:Ready ...], object identifier *ObjectID* and the boolean value *ReScheduleBool*, which indicates whether or not the schedule is to be rescheduled, are obtained as arguments. After the validity of the object identifier is checked, a message is sent

to object block manager, *OCB*, to put the object in the waiting state. Next a message is sent to the list manager, *List*, to insert the object into the ready list. At this time a message is sent to *OCB* to obtain the priority of this object, which is sent to *List*. Items in the ready list are in the order of their priority, from low to high. Lastly, the necessity of rescheduling is confirmed. If rescheduling is necessary, the internal routine object *Schedule* is requested to reschedule. If the message :ReSchedule is accepted, rescheduling is made by using an internal routine object. If the message :ReferCurrentObjectID is accepted, "current object" identifier *CurrentObjectID* is returned.

When the internal routine object, *Schedule*, accepts the message :ReSched, it does not perform rescheduling if the state of *CurrentObjectID* is 'CURRENT' and the priority of this object is higher than that of the object in the tail of the ready list (the ready list is in the ascending order of priority from low to high). Otherwise, the context is switched. First, if the state of *CurrentObjectID* is 'CURRENT', the state is changed to 'READY' (for example, when the real time clock manager accepts the :Sleep message, the state of the *CurrentObjectID* is set to 'DORMANT' before the :ReSchedule message is sent), and the identifier of the object and its priority are cataloged in the ready list. Then the object with the highest priority is taken out of the ready list, and its state is set to 'CURRENT'. Finally, *Timer* is set to the quantum time, and the context is switched by using ctxsw, which is implemented in the assembly language.

Thus, using nested routine objects, the scheduler can be written compactly, based upon the concepts consistent with other objects.

4.2 Network communication

Roughly speaking, this layer has the following three functions:

(1) message communication

(2) creation and management of buffer pools

(3) port management

The first function performs interprocess message communication via network, and consists of a number of system functions and several processes. A message is sent and received as a block. An outline of the transformation process is shown in Fig. 7.

The second function performs the creation of buffer pools to be used in the data link layer of (1) above and in disk management, as well as the allocation and release of buffers in the pools.

The third function performs the creation and management of "ports" which are the basic logical structure elements of frame and block in (1) above. A port is a list the basic element of which is a word.

We implemented the functions (2) and (3) each as an object, and function (1) by several objects. At present, transport, internet, and data link in Fig. 7 are implemented as one object each, and frame is implemented by 3 objects.

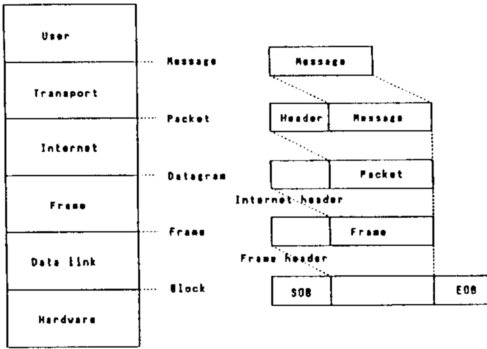

Fig. 7 Network software layers and data formats

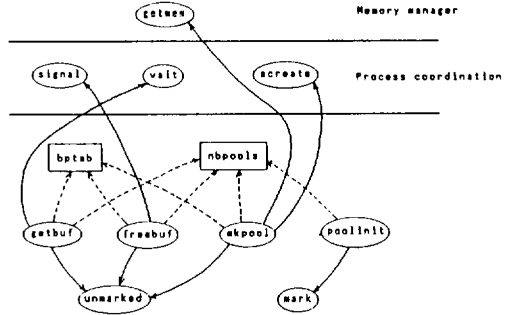

Fig. 8 Creation and management of buffer pool (a part of network communication)

For instance, modules (functions), data, and the reference relations of the buffer pool creation and management function in the original version are as shown in Fig. 8. The main function here is the dynamic management of the set of buffers with the size specified at buffer pool creation. The table *pbtab* is used to record, for each buffer pool, the size of buffer pool, pointer to the free buffer, and the semaphore for allocation management. The *nbpools* is used to count the number of buffer pools.

In the ABCL/c+ version, a buffer pool is created as an object, which is allocated. Each buffer has an identifier of the pool to which it belongs. Thus, no buffer is returned to a wrong buffer pool. Fig. 9 shows buffer pool creation and management object *BufferCreator*.

The object *BufferCreator* can accept :Initialize message and :New message. When the message :Initialize is accepted, state variable *NumberOfBufferPools*, which indicates the number of buffer pools currently supplied, is initialized to 0. When the message [:New ...] is accepted, the size of the buffer pool is computed, using *BufferSize* (size of a buffer) and *NumberOfBuffers* (the number of buffers), and a memory area with that size is allocated. Actually, the allocation is achieved by sending a call type message to the system object for memory management, *MemoryManager* (see Fig. 5). At this time, the pointer and pool identifier parts are added to each buffer (2*SizeOf(int)). *MemoryManager* returns the start address of the allocated area. If a memory area with the necessary size cannot be allocated, SYS_ERR will be returned.

If an area is allocated successfully, the state variable *NumberOfBufferPools* is incremented by 1. Then buffer pool is initialized by chaining all buffers using the pointer part of the buffers.

Lastly, the object *BufferPool*, which allocates and releases the buffers in the buffer pool, is created and returned to the sender of the message. The object *BufferPool* can accept the message :GetBuffer and the message :FreeBuffer. When the :GetBuffer message is accepted, the object checks to see if any buffer can be allocated in the pool. If not, it waits for the :FreeBuffer message to come in the select form and release a buffer. If a buffer is released, or one is available, the

first buffer in the free buffer list, whose root is *Next*, is taken out. *Me*, that is the ID of the object *BufferPool*, is set in the pool identifier part. Then the buffer is returned to the sender of the message. When :FreeBuffer message is accepted, it checks to see if the buffer in the message belongs to it. If not, it returns SYS_ERR. If it is the buffer which belongs to the object, the returned buffer is chained to the head of the free buffer list, and OK is returned to the sender of the message.

We can see that, by making an object for each buffer pool, which manages the buffers in the pool, we can at the same time clarify and simplify the functions.

```
[object BufferCreator
 state {
  Bool InitializeFlag := TRUE;
  int NumberOfBufferPools;
 }
 script {
  (=> [:Initialize] where ( InitializeFlag ) # int
   {
        InitializeFlag := FALSE;
        NumberOfBufferPools := 0;
        !OK;
   }
  (=> [:New BufferSize NumberOfBuffers] # int
   int BufferSize, NumberOfBuffers;
   {
        short Psw;
        char *Where, *Next;

        Disable( Psw );
        if( BufferSize < BUFFER_MIN_BYTE
          || BufferSize > BUFFER_MAX_BYTE
          || NumberOfBuffers < 1
          || NumberOfBuffers > BUFFER_MAX_NUMBER
          || NumberOfBufferPools >= MAX_BUFFER_POOLS
          || ( where := [MemoryManager <- [:GetMemory
           ( ( BufferSize + 2*SizeOf(int) )*NumberOfBuffers )]] )
           == SYS_ERR ) {
          Restore( Psw );
          !SYS_ERR;
        } else {
          Restore( Psw );
          NumberOfBufferPools++;
          Next := Where;
          BufferSize += 2*SizeOf(int);

          for( NumberOfBuffers-- ; NumberOfBuffers > 0 ;
            NumberOfBuffers--, Where += BufferSize )
            *( (int *)Where ) := (int)( Where + BufferSize );
          *( (int *)Where ) := (int)NULL;

          ![object BufferPool
           script {
            (=> :GetBuffer # int
                 {
                  int *Buffer;

                  if( Next == NULL )
                   [select
                    (=> [:FreeBuffer Buffer1] # int
                          int *Buffer1;
                          {
                           *Buffer1 := (int)Next;
                           Next := Buffer1;
                          })];
                  Buffer := Next;
                  (int)Next := *Buffer;
                  *( Buffer + 1 ) := Me;
                  !(int)Buffer;
                 })

            (=> [:FreeBuffer Buffer] # int
                 int *Buffer;
                 {
                  if( *( Buffer + 1 ) != Me ) !SYS_ERR;
                  else {
                   *Buffer := (int)Next;
                   Next := Buffer;
                   !OK;
                  }
                 })
          }];
        }
   })
 }]
```

Fig. 9 Buffer pool creator

5 Discussion

The routine object is introduced to unify procedures, functions, and objects. By introducing the routine object, procedures and functions for creation of the world of processes, as well as those in objects, can be constructed by the same concept as that of the objects. The example given in 4.1 clearly illustrated the advantage of this approach.

We have also shown that, in addition to its effectiveness in grouping global data along with operations on it, as seen in the case of the process control block, the object-oriented concept is

very effective in the management of "objects" which exist independently as seen in the example of the buffers in 4.2.

The buffers discussed in 4.2 are used to manage data transfer, triggered by external interruption, in the data link layer and disk management. By implementing each buffer pool as a "concurrent object", data processing mechanism can be made simple and clear.

Thus, we have shown that concurrent object-oriented languages, such as our ABCL/c+, are suited to describe an operating system kernel. Although it is desirable to describe even the machine dependent part in the same paradigm, we have not yet implemented it in the ABCL/c+ language.

6 Conclusion

In this paper, we presented the outline of concurrent object-oriented language ABCL/c+ and the result of writing an operating system kernel in ABCL/c+. As far as we know, this is the first attempt to write an operating system kernel in a concurrent object-oriented language. We have demonstrated that, using a language with the routine object, which is our extension of the ABCL/1, and the ability to describe the simple and powerful concurrent objects, such as in the ABCL, a very highly understandable operating system kernel can be written.

However, we also assume that function layers within an object can lead to an even higher degree of ease in writing and understanding programs. Therefore, in order to achieve the layer of functions, we tried to make it possible to nest the definitions of ordinary objects other than routine objects in ABCL/c+. However, we have not yet been successful in finding satisfactory approaches for the method inheritance and search strategies (depth first or breadth first). This point is still under investigation.

The development of the processor of ABCL/c+ and the rewriting of XINU using ABCL/c+ are carried out on the Sun workstation and Macintosh II. The final result, the ABCL/c+ version of the XINU operating system, will run on Macintosh II.

Acknowledgements

The authors wish to thank Akinori Yonezawa, who gave them a number of useful comments in preparing this paper. Also they wish to thank the referees for their useful suggestions and comments on the first version of the manuscript.

References

[1] Borow, D.G. and Stefik, M. *The LOOPS Manual, KB-VLSI-81-13.* Xerox PARC, 1983.

[2] Chikayama, T. *ESP Reference Manual, TR-044.* Technical Report, ICOT, 1984.

[3] Comer, D. *Operating System Design : The XINU Approach.* Prentice-Hall, 1984.

[4] Comer, D. *Operating System Design - Volume II: Internetworking with XINU*. Prentice-Hall, 1987.

[5] Doi, N. and Kodama, Y. *ABCL/c+ User Manual*. Institute of Information Science, Keio University, 1987. (In Japanese.)

[6] Doi, N. and Segawa, K. "Concurrent programming in Smalltalk-80." *Computer Software*, 3(1), 1986. (In Japanese.)

[7] Goldberg, A. and Robinson, D. *Smalltalk-80 : The Language and its Implementation*. Addison-Wesley, 1983.

[8] Ishikawa, H. and Tokoro, M. "Orient 84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation." in *Object-Oriented Concurrent Programming* (ed. Yonezawa, A. and Tokoro, M.). MIT Press, 1987.

[9] Kristensen, B.B., Madsen, O.L., Moller Pedersen, B., and Nygaard, K. "Multisequential execution in the BETA programming language." *Sigplan Notice*, 20(4), 1985.

[10] Shibayama, E. and Yonezawa, A. *ABCL/1 User's Guide*. Information Science Department, Tokyo Institute of Technology, 1986.

[11] Yokote, Y. and Tokoro, M. "Concurrent Programming in ConcurrentSmalltalk." in *Object-Oriented Concurrent Programming* (ed. Yonezawa, A. and Tokoro, M.). MIT Press, 1987.

[12] Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1." in *Object-Oriented Concurrent Programming* (ed. Yonezawa, A. and Tokoro, M.). MIT Press, 1987.