

Combining Oberon with Active Objects

Andreas R. Disteli
Patrik Reali

Institut für Computersysteme, ETH Zürich

Abstract. Multitasking is a topic on which many discussions have been held. There are different opinions about its need. Our own operating system, Oberon, was originally designed as a single user and single process system. While developing server systems and simulation kits we came to the conclusion that we need some notion of slim and easy to use process. We then decided to start a new project comprising the design of a new kernel and a new compiler for the support of concurrent execution of several processes on different priority levels with an appropriate protection mechanism preventing processes from inadmissible access to each other. The idea of active objects, simultaneously representing processes and containing both their action and their data, was the base for the 'Active-Oberon' project. Active objects are independent processes scheduled by the system. The management of the memory including stack allocation devolves on the new kernel. This takes much responsibility away from the programmer and makes the system safer.

The goal of this paper is to present the implementation of the 'Active-Oberon' project whose concepts are described in detail in [Gut96b].

Keywords: Object Oriented Programming, Active Objects, Concurrency, Multiprogramming, Operating Systems, Oberon

1. Introduction

The Oberon system is a single user and a single process operating system [Rei91] [WG92]. Although very small, it is a very powerful system which is sufficient for almost all applications. As long as only one person is working with an Oberon based computer, the single process system with its central loop is powerful enough. So, there is no explicit need for multitasking. In spite of this fact, the Oberon system allows multitasking, but in a very coarse way. As described in [WG92], it is possible to install tasks into the main processing loop. Alternating with the system tasks, i. e. keyboard and mouse handlers which are also part of the central loop, they are executed as a whole, without any scheduling opportunity. A task has to run from the beginning to the end of the current action and it cannot be interrupted by any other usertask or systemtask. It is obvious that this can result in remarkable delays for interactive system tasks. E. g. a task calculating some complex expression can block the whole system during its execution unless it is programmed as a state machine, executing a small piece of code operating on some globally saved state. For systems like a server, which offer a variety of different services, sometimes time consuming, this tasking system is unsatisfactory. Besides that, inverted programming of a state machine is much less natural than programming a process in a linear way.

Another point is that a task is represented as a procedure in a module. Data which are used in the next execution period of the task have to be global in order not to get lost. This means that there is no persistent local environment for a task. Also two tasks using the same environment, i. e. the same global data, would cause race conditions. It would be much more elegant to have an object which (a) can keep its data local for its whole lifetime,

(b) knows all about its behaviour and (c) could even appear in more than one instance at the same time.

For these reasons we launched the 'Active–Oberon' project with the aim to include multitasking into the existing Oberon system, allowing the programmer to write lightweight processes in a straightforward and secure way without taking care of scheduling. Scheduling should not burden the programmer but be part of the operating system. For example, even when lots of time consuming background processes are running, the user should get immediate control over the interactive part of the system without even noticing the background actions.

In our system we only use lightweight processes (threads) that have their own stack but share their global address space with the other processes in the system.

All these considerations lead us to the following requirements:

- the old Oberon tasks are eliminated
- there are active objects simultaneously containing data and action (process)
- the system controls the scheduling
- the user has only a restricted control over scheduling
- there are different priority levels for the processes
- the objects can be access–protected against one another
- as much management as possible should be done by the system instead of the user

The concepts of active objects and the Active Oberon language description can be found in detail in [Gut96b].

Respecting the Oberon philosophy we tried to keep the changes to a minimum. Case studies show that this approach is sufficient for a variety of applications (see chapter 4).

The embedding of these new concepts into the existing system required major changes at two places. The first one is the kernel which has to manage process scheduling and the second one is the compiler which has to support the new language features. Chapter 2 will first discuss the runtime environment and the kernel while chapter 3 will then focus on the compiler.

2. Runtime Environment

The whole runtime environment is concentrated in one module of the inner core, the kernel. It is based on the native Oberon port for PCs [Mul96]. Having complete control over the machine means that we can fully concentrate on the actual problems without having to take care of restrictions set by a foreign operating system.

The original kernel is responsible for memory management (allocating and collecting memory), interrupt handling and initialization of the system. The new kernel in addition contains the protection mechanism for processes, their management and their scheduling. Due to these new responsibilities we also had to change the memory organisation. The use of a different addressing technique was a crucial decision for the new memory management.

2.1 Organisation of the Memory

Every process needs its own stack for the local data and procedure activation frames. Normally the maximum stack size cannot be determined at compile time. In simple cases only the compiler can compute the needed stack size, if there are neither open array parameters nor recursion nor procedure variables. In all other cases the size is known at runtime only. In our old coroutine model [Gut] every coroutine was assigned a fixed-size stack at start time. The programmer had to make an estimate about the size the coroutine would need during its runtime. As there was no check against the stack boundary this insecurity often lead to an unpredictable behaviour of the system if the stack underflow. Another important restriction is that the stack must be contiguous if we do not want any additional tests, register setup or block relocation.

So we had to find a technique that allows us to create stack blocks at runtime as soon as the stack runs out. The new block should further be contiguous to the old stack and each lightweight process should independently be able to reduce or increase its stacksize during runtime.

Just focussing on stacks until now, we must not forget the heap. We did not intend to divide the main memory into predefined heap and stack blocks. The main memory should be treated like a general resource for both types of memory blocks.

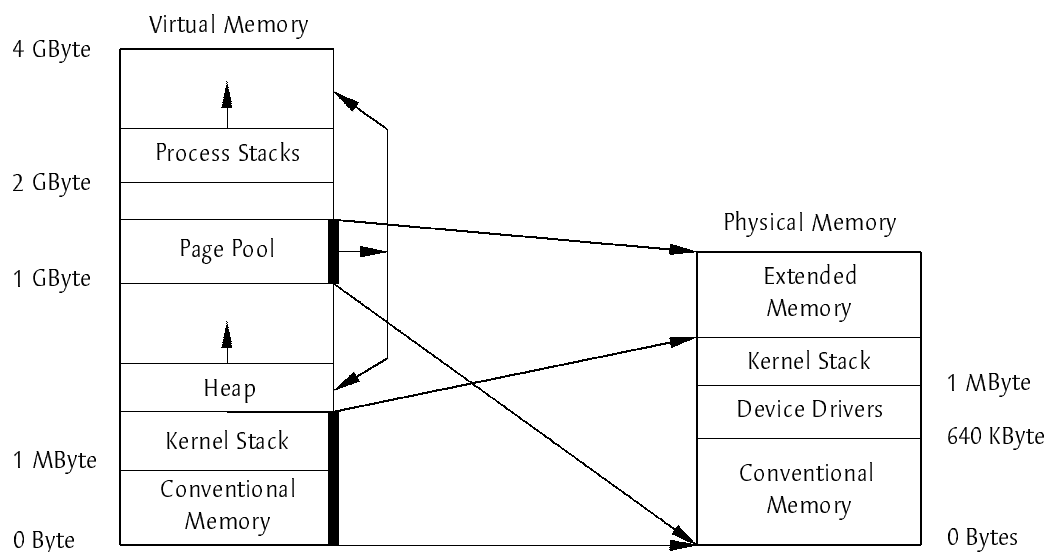


Fig. 1: Memory Layout

We found the solution in virtual addressing supported by the underlying hardware [Int91]. Each virtual address is translated into a physical address at runtime. This allows a memory block to look contiguous from the view of virtual addresses but is in fact split up into several blocks in different locations in physical memory. Or the other way round: we can allocate an arbitrary block in physical memory and simply adjust the mapping so that the new block is adjacent to the old stack address resp. the old heap address. This fits exactly our requirements mentioned above.

The only restrictions we have to deal with are the range of the virtual address space of 4GByte (given by the 32 bit architecture of the processor) and of course the amount of

physical memory. It would be possible to extend the amount of memory by using disk swapping, but in the current implementation we did without it.

The whole memory is divided into pages, each 4 KByte in size. The granularity of the pages is given by the hardware and cannot be changed. Figure 1 shows the memory layout for our system. The whole physical memory is mapped to virtual addresses starting at 1 GByte. All free pages are linked together in the so-called page pool. Whenever a new page is needed, it is taken out of this page pool and its virtual address is adjusted so that the stack as well as the heap remains a contiguous memory block in the virtual address space.

Virtual addresses of the process stacks start at 2 GByte. This allows us to use the upper 2 GByte address space for the stack management. Assuming a maximum size of 128 KByte per stack this results in 16'384 possible processes as each process has exactly one stack. If every process needs only the minimum of stack space the system needs 64 MByte only for the stacks. We could think of a smaller stack limit in favor of more processes, but the coarse granularity of the memory blocks and the resulting memory requirements justifies the chosen stack size. A remark about waste of memory is still justified here. A page granularity of 1 KByte would be much more adequate for our purpose. Many processes do not need 4 KByte of stack memory. On the other hand, there would be more stack faults for stack consuming processes. Fortunately the hardware design made the decision for us.

The conventional memory (the first MByte) is mapped directly, i. e. the virtual addresses are the same as the physical addresses, this allowing easier access to hardware data and memory mapped I/O.

2.2 Organisation of the stacks

For each instance of a process, the new kernel allocates a memory block of a fixed size, i. e. the minimum size is given by the page size. Using the capabilities of the underlying hardware we can detect a page fault (stack underflow) in this memory block and can react on it. On every underflow we allocate a new memory block for the stack and append it to the existing one(s). This mechanism of a dynamically growing stack is new to the Oberon system which always dealt with fixed sized stacks.

Another approach for determining the needed stack size would be to insert a stack checking mechanism in the compiler [Die94]. But this results in a much more complex compiler and slower code due to the stack check overhead. Concerning these facts, we decided to use the hardware support of virtual addressing and keep the generated code and the compiler small and fast to achieve best performance regarding compile time and code size.

As mentioned above, a lot of processes do not even need 4 KByte of stack. As this is the minimum size we can get we could think of using this space for other purposes, too. Since each process has at least one stack block allocated at start time we use this block for storing process relevant information. At the beginning of each stack there is a reserved area shown in figure 2. On every process switch (see below) we store the current process state in this area. This includes both the state of the CPU and, if needed, the FPU. Furthermore, we defined a small area reserved for the operating system which is used in connection with process switch and a stack setup for a termination routine (see below).

Due to the fact that every process has its own stack, we also had to make changes in the garbage collector. Besides traversing the heap it has to traverse now all the process stacks

and check them for pointer references. In the original system the garbage collector was only called at the end of a command and made no stack traversal at all. Now, the garbage collector is a process of its own and garbage collection on the stack has become an absolute need.

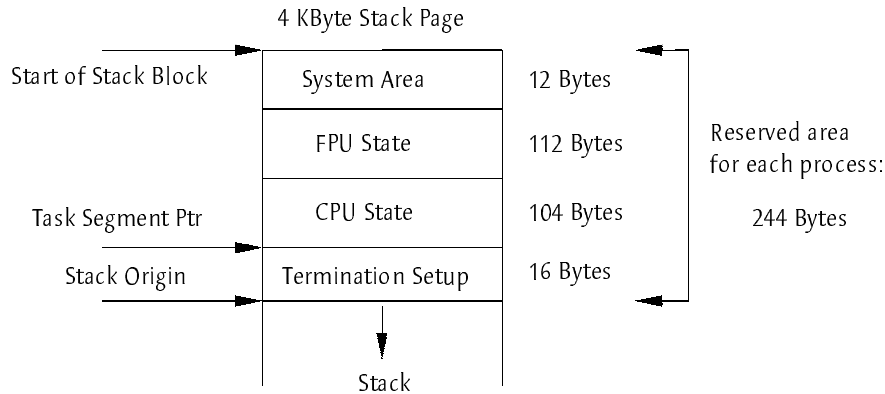


Fig. 2: Stack Block

2.3 Protection Mechanism

There are no explicit commands in the language for protecting and unprotecting an object from uncontrolled access. Both of these operations are done implicitly by the operating system. In the following we also use the terms *locking* and *unlocking* for protecting and unprotecting.

Whenever processes intend to write to some shared data, they need protection against each other. As long as a process is in a critical section (a section in which the working process must have exclusive access to the data) a second process must wait until the first one leaves this section.

Two processes can also be in the same critical section as long as they only read the data and do not modify it. A third process that wants to write data (*writer process*) would then have to wait until the two *reader processes* have finished, while another reader process could enter. It is obvious that reading and writing processes must exclude each other.

We use monitors [Hoa75] as a base for our implementation and added the readers/writers concept [Bac93].

In our system every object consists of a set of exclusive, shared and/or unprotected entrypoints. Entrypoints can either be module procedures in case the object to be entered is a module, or typebound procedures in case of an instance of an object. For detailed information about the dualism between modules and types see [Gut96b]. Each time a process enters an object or a module through an *exclusive* entrypoint, it tries to lock the referred object. If unlocked, the process enters the object and simultaneously locks it (Fig. 3.1). The same process may obviously enter an object recursively. This must be like that, otherwise a protected entry could not be called from within another protected entry in the same object. Note that it is under responsibility of the object to keep track of (a) which process entered and (b) the number of times the process entered (Fig. 3.2). An object locked by another process cannot let pass any new one trying to enter. In this case the new one is passivated by the operating system (see below: process management) and has to wait until the object will be unlocked (Fig. 3.3).

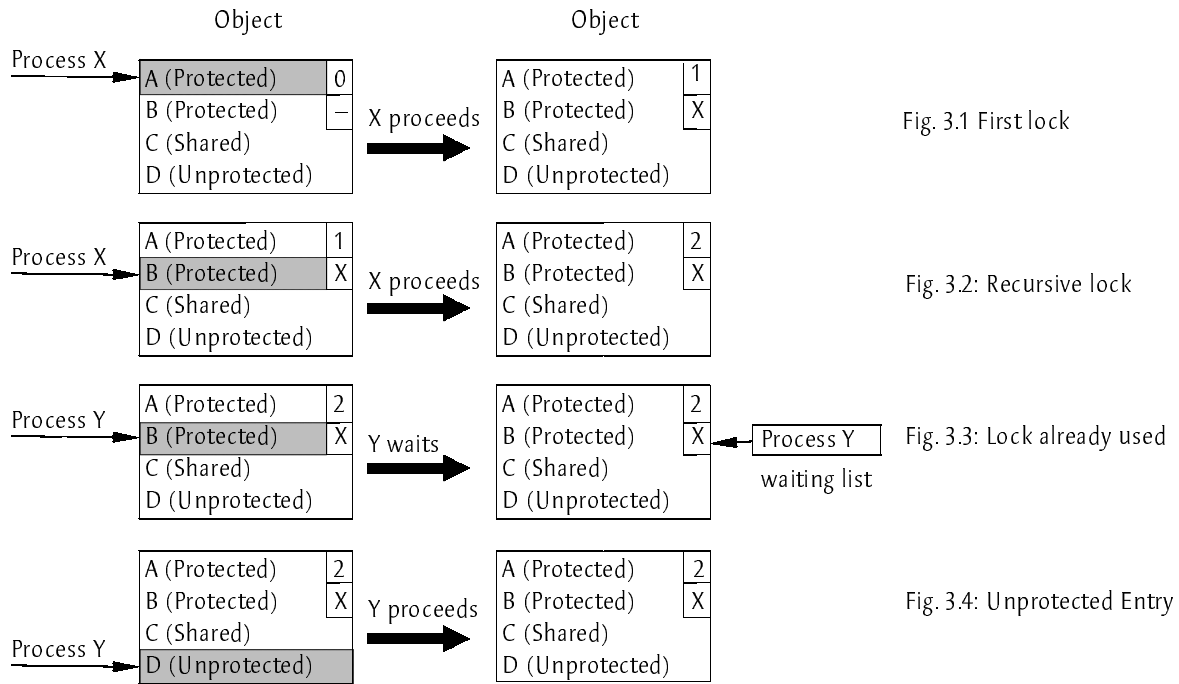


Fig. 3: Locks

Shared entrypoints work in a similar way. The difference is that as long as there are no writer processes trying to enter via some protected entry, reader processes can enter the object simultaneously via shared entries. An arriving writer process forces queuing of further reader processes until the writer has finished. The writer gets control as soon as all the readers already in the object have left. This guarantees that writer processes will get control as soon as possible and the data will be up to date for further reads.

On leaving a protected entry the operating system checks whether another process is waiting for the released object. If there is one, it will be activated, acquires the lock and gets control depending on its priority.

Unprotected entries do not check either for locks on entry or for waiting processes on exit (Fig. 3.4).

While a process is inside a critical section, it can occur that the process has to wait for a certain condition to become true. As mentioned above, the runtime environment then takes the process out of the *active queue*. If the lock would be kept while waiting for the condition, no other process could enter this object, although the condition can only become true through a call to another guarded entry of this object. This would lead to a deadlock: a process, waiting for a condition, keeps locking the entered object and no other process can enter this object and change the condition. For this obvious reason we have to unlock the object while waiting. It is the responsibility of the process to remember the object it has to relock when it gets reactivated.

2.4 Process Management

Every process is in one of four different states. A newly created process has its state set to *active* and is inserted into a queue which holds all active processes. Similarly to this 'active queue' there are also queues for the other states, namely queues for *passive*, *waiting* and

terminated processes. These queues are global to the system and contain only processes whose change of state depends on a global action, e. g. evaluation of a global condition or reactivation after an unconditional passivate (see below).

In addition there are queues local to the object. Conditions containing data local to the object can only be changed by another procedure call also local to the same object. Local conditions are therefore kept in a queue belonging to each object. Processes waiting for entry to an object are kept in another local queue. These local queues reduce search expenditure during runtime and promote locality and object orientation.

An active object can be passivated in two ways: unconditionally (using the *PASSIVATE* command), or waiting for a condition to become true (*PASSIVATE(condition)*). In the first case it has to be reactivated by a explicit call of *ACTIVATE*. We use this form whenever we do not know the exact condition of reactivation so that a different process needs to react. In the second case the operating system checks the condition and, if the condition is false, changes the state to 'waiting', i. e. it moves the object from the 'active queue' to the 'waiting queue'. Depending on the locality of the used data in the condition, the object is either inserted into the local queue and the reevaluation of the condition is done on the exit of a protected entry of this object, or it is in the global queue and the evaluation is done by a system task running on each timer interrupt. The reevaluation can be forced in both cases with an explicit call of *ACTIVATE*, but reactivation will only take place when the condition is true.

When reactivating an unconditionally passivated object (by calling *ACTIVATE(you)*), two cases can occur, leading to different transitions. If the reactivated object is allowed to enter the previously released object it is inserted back in the 'active queue'. If not, the reactivated process changes to the local waiting queue. As soon as the previously released object becomes free again, the waiting process relocks it and moves to the 'active queue'.

In a system with several processes there will normally be more than one process waiting for a locked object and also more than one process which can change back to active because the condition is true. In these cases the process with the highest priority gets control. If there are equal priorities and there is a process in the 'waiting queue', this process gets control first. The reason is that this process is already in the object (*PASSIVATE(cond)* is always called from within an object) while a process waiting for a lock is only about to enter the object. To guarantee fairness between processes, all queues are organized in first-in first-out order.

Terminating an object means removing it from the 'active queue' and inserting it into the 'terminate queue'. Its action is stopped but the object can still remain in the system as a passive object. The locks seized by the terminated object must be released, in order not to prevent other, still active processes from acquiring this lock. A terminated object cannot be reactivated, because the stack pages have been returned to the system. The terminate queue is checked periodically by the timer interrupt procedure. At this point all used stack and heap memory can be returned to the system. This means that a process can only terminate itself implicitly by reaching the end of its body. In particular there is no way to terminate a process remotely. Figure 4 shows all the possible transitions between the different queues:

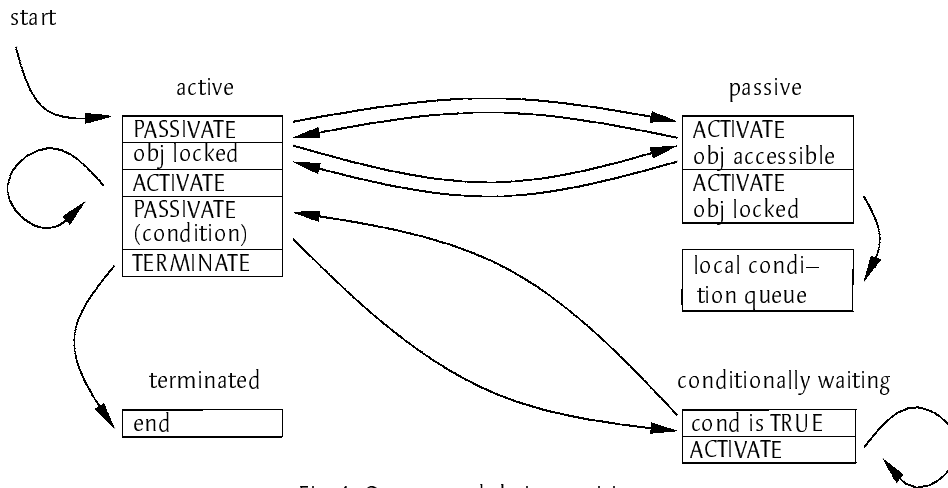


Fig. 4: Queues and their transitions

2.5 Process switch

The basic scheme of switching processes is simple. The state of the current process is stored on its stack. Then, the system selects the next process, restores its state and (re)starts its execution.

The underlying hardware provides different protection levels [Int91]. They must not be confused with our protection mechanism discussed above. Hardware protection simply means here controlled access to memory areas between system and user. All our processes run on the least privileged user level, i. e. they cannot use the privileged instructions. During a normal use of the system this does not matter. But when changing from one process to another, we exactly need some of these instructions. So, either we have to do expensive system calls to change the level of operation or we have to use a more privileged level for the user code. Both solutions are bad. Again we found a solution in the hardware support. The processor supports a special task switching mechanism. This allows us to store the whole CPU state and restore a new one by one instruction available at user level. Normally, such complex instructions are very slow compared with simple instructions as e. g. 'MOV' and 'ADD' and could be done much faster by reprogramming them using simple instructions. But measurements showed that, although too much is done by this instruction for our purposes, it is still faster than reprogramming it. Also it simplifies a lot as we shall see later.

When the processor performs a task switch it uses a so-called task segment [Int91] for storing the state of the old process and restoring the state of a new one. Some fields of this task segment contain information that we do not need. But as the layout of this segment is given by the hardware we cannot change it.

The processor offers an easy way for switching from one task to another. As the task segment contains all relevant data of a task, it would be easiest to store the state of the old task and restore the state of the new one. It is possible to do this by just using a special form of the 'JMP' instruction and indicate to which task segment, resp. to which process, we want to jump. The processor then performs the whole task switch by storing the state of the old task and restoring the state of the new one. This sounds very easy. In fact, it is so, but of course some setups must be done behind the scenes. Each process needs its own task segment which is located in the reserved stack area (Referred as CPU state field in Fig. 2).

The descriptor for this segment additionally needs 8 bytes in a system table. Assuming that we do not have thousands of processes this is reasonable. But this covers only the CPU. The FPU state is not saved by this procedure. Considering the amount of time needed for storing the FPU state and the fact that not all processes use the FPU lead to the following idea: The state of the FPU is only stored when another process is going to use it. For this we have to keep track of the last process which used the FPU. As soon as another process accesses the FPU we store the state into the FPU state field (Fig. 2) of the last process and reload the new state if there is one or init the FPU if there is none.

As can be seen here, switching from one process to another is quite simple and can be done very efficiently. With this technique we also do not have to care about privilege level restrictions of certain instructions, e. g. changing of segment limits, flushing page tables, etc.

2.6 Scheduler

Active objects can run in two modes. The first mode handles them in a timesliced way which means that there is no need of taking care of synchronous process switches by the programmer. The system takes over this task and schedules the active objects each time it gets control. Still, timeslicing does not exclude the use of synchronous control passing statements. The timesliced mode is indicated at the beginning of the active body with the keyword *{TIMESLICED}*. The second mode exclusively uses the synchronous passing statements *ACTIVATE*, *PASSIVATE* and *PASSIVATE(condition)*. Between these statements control is not passed to any other active object unless there is one with a higher priority. Of course, an active object running on a higher priority level always preempts the current process and gets immediate control. The only exception is a low priority process locking an object that the high priority process wants to enter. In this case even the high priority process has to wait until the lock is given back.

The timer interrupt handler routine is sketched in the following pseudo code:

```

PROCEDURE TimerHandler;
BEGIN
  IF current PC is outside the kernel THEN (* Kernel is a monitor module *)
    WHILE there are processes waiting in the Condition Queue DO
      IF condition is true THEN activate this process END
    END;
    Get next process from the 'active queue' depending on the priority;
    IF current process # next process THEN switch to next process (* higher priority *)
      (* If there are only processes with the same priority, next equals current *)
    ELSIF current process runs timesliced THEN switch to next ready process (* time slice is over *)
      (* there is always a ready process, as the system owns a 'idle' process *)
    END
  END
END TimerHandler;

```

This handler also makes clear that it can be very time consuming to evaluate all conditions at every timer interrupt.

For each condition the address of the code is known at compile time. Every time *PASSIVATE(cond)* is used this address and the corresponding object (SELF) are passed to the kernel which inserts these data into the (local or global) waiting queue. When evaluating a condition the system makes a call to the address of the condition and passes SELF as a

parameter. The returned boolean value indicates whether the condition is true or not. No context switch is needed!

Another frequent situation is waiting until a certain time has elapsed. Statements of the form

```
time := system time + dt;  PASSIVATE(time < system time);
```

are legal but can be handled more elegantly and more efficiently. Obviously, again if there are many of these conditions, evaluation gets very inefficient. As an optimisation, we introduced a timer object which offers a procedure 'Hold(dt)'. This object handles all waiting processes in a way that only a few tests have to be made every timer tick. This increases the efficiency of the system remarkably. Figure 5 shows the basic idea behind the implementation of this approach: a list sorted by reactivation time. All processes with the same reaction time are linked together and can be activated as a whole when time is ready.

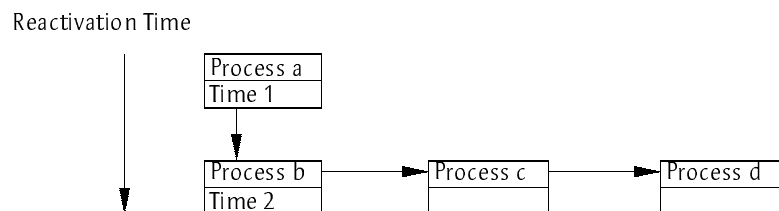


Fig. 5: Timer Object

3. The Active–Oberon Compiler

The Active–Oberon Compiler implementation is based on the OP2–Compiler [Cre91], a two pass compiler for the Oberon–2 language and generates code for Intel 80386 processors [Int91]. Because of differences in syntax and semantics Oberon–2 extensions have been discarded. The language implemented in the Active–Compiler is the Active–Oberon language [Gut96a] [Gut96b], a direct extension of the Oberon language [Wir88].

The relevant changes in Active–Oberon can be subdivided into three categories: (a) Syntax extensions to implement type bound procedures, (b) embedded protection and (c) embedded process management. The most important change of the compiler is the syntax extension. Perhaps surprisingly, it simplifies the parsing of the language a lot. Other changes have been made in the compiler to enhance code efficiency, in particular a facility for inline procedures has been introduced. Almost all the changes affect the parser, the back–end of the compiler remains nearly unchanged. Only five new system calls have been added.

3.1 Active Oberon Language Parsing

Instead of having three distinct syntax productions for modules, procedures and records, the Active–Oberon language unifies these productions in a single syntactical structure, here represented by the *DeclSeq* and the *Body* productions in EBNF notation:

```
$ Module      =  MODULE ["(" PROTECTED ")"] ident ";" [ImportList] DeclSeq Body ".".
$ ProcDecl    =  PROCEDURE {ProcTags} IdentDef [FormalPars] ";" DeclSeq Body.
$ RecType     =  RECORD ["(" Qualident ")"] DeclSeq Body.
```

```

$ DeclSeq      =  {CONST {ConstDecl ";" } | TYPE {TypeDecl ";" } | [VAR] {VarDecl ";" } |
                  ProcDecl ";" | ForwardDecl ";"}.
$ Body         =  [ BEGIN [ObjMode] StatementSeq | CODE {character} ] END [Ident].
$ ObjMode     =  "{" ObjModeSpec { "," ObjModeSpec } }".
$ ObjModeSpec =  PARALLEL "(" ConstExpr ")" | EXCLUSIVE | SHARED | TIMESLICED.

```

The semantics of the different *ObjModeSpec* are described above and in [Gut96b]. The new syntax allows the implementation to have only one parsing procedure for modules, procedures and records. Still there are some checks to be done, because the language semantics are not completely orthogonal (e.g. active procedures make no sense). The following table resumes the actual implementation restrictions:

	<i>Module</i>	<i>Procedure</i>	<i>Record</i>
local vars/fields	yes	yes	yes
constants	yes	yes	yes
local types	yes	yes	no *
local procedures	yes	yes	yes
body	yes	yes	yes
protectable	yes	no	yes
active	yes	no	yes

(*) Local record types are a temporary implementation restriction. Conceptually there is no reason for omitting them.

Procedures are not protectable but inherit the protection attributes from their context (i.e. the module or the object).

3.2 Code Patterns

In the Oberon System [Rei91] [WG92] the compiler and the system are bound very tightly and thus enable tight cooperation. This simplifies the compiler task by allowing assumptions about the runtime environment and therefore making possible the integration of run-time features (e.g. memory allocation) directly in the language semantics.

The interface between the compiler and the kernel is the object file. It contains the generated code, fixup information and the code itself.

The runtime environment is responsible for memory allocation and deallocation (via a garbage collector), data protection and process handling. These services are accessible by the compiler through system calls. In line with the Oberon philosophy, these services are integrated in the language in a transparent way and run-time system calls are automatically generated by the compiler. The number of system calls is kept to an absolute minimum. The Active Oberon kernel contains five new kinds of calls for providing the needed functionality to implement the added language semantics:

```

Kernel.Lock (SELF: PROTECTED);
Kernel.Unlock (SELF: PROTECTED);
Kernel.Start (body: PROCEDURE; priority: LONGINT; SELF: Kernel.ActiveObject);
Kernel.Passivate (condition: PROCEDURE(): BOOLEAN; contextPtr: LONGINT; SELF: Kernel.ActiveObject; global:
BOOLEAN);
Kernel.Activate (you: Kernel.ActiveObject);

```

The generation of code for processes and/or protection heavily relies on these new system calls, because these functions need substantial run-time support.

Object protection

When the code for a protected entry into an object is generated, the compiler automatically inserts the *Kernel.Lock* and *Kernel.Unlock* system calls. We optimized the average case by embedding part of the system call directly into the caller code and thus avoiding the expensive system call whenever possible. For example, the generated code for a call to the protected entry R is:

```
PROCEDURE R      enter
                  IF SELF.lock = 0 THEN    (* not locked yet, normal case *)
                      SELF.count := -1; SELF.lock := Kernel.CurrentProcess
                  ELSE
                      call Kernel.Lock (SELF)
                  END;

                  { code for x := 2 }

                  IF SELF.count < -1 THEN
                      call Kernel.Unlock (SELF)
                  ELSE
                      SELF.count := 0; SELF.lock := 0;
                      IF SELF.conditions # NIL THEN call Kernel.CheckConditions (SELF.conditions) END
                  END;
                  exit
```

The *Unlock* code pattern must again be inserted before every return statement. The variable *SELF* is always defined in every scope. Every type bound procedure has *SELF* as a hidden parameter. For example the call to R is:

```
active.R          push active
                  call R
```

NEW

The code generated for object generation is more complex because the semantics of the *NEW* instruction have been extended. The *NEW* instruction implies three steps now: (1) allocate an object in the heap (2) initialise it (3) call the object body. *NEW* and the initializer build one atomic operation because the initialisation of a process must be done before the process is started (Configuring a running process may lead to race conditions). The compiler can now generate different code patterns for the *NEW* instruction: one for active objects and one for passive objects. The code generation routine for the *NEW* instruction is like this:

```
PROCEDURE GenNew (x: Node);    (* node x represents the NEW instruction *)
VAR obj: Node;
BEGIN
    obj := GetObject (x);
    GenNewRecSystemCall (obj, GetTypeDescriptor(obj));
    IF IsActive (obj) THEN AdjustActivePointer (obj) END;
    IF HasInitialiser (obj) THEN GenCall (FindInitialiser (obj), FetchInitParameters (x)) END;
    IF IsActive (obj) THEN GenStartSystemCall (FindBody (obj), FindPrio (obj))
    ELSE GenCall (FindBody (obj), NoParams) END
END GenNew;
```

The generated code can be best understood with the help of an example:

```

NEW (active, 1)   Call Kernel.New (active)
                   adjust pointer
                   Call active.Init (1)
                   Call Kernel.Start (active.body, 2, active)

```

The need for pointer adjusting is a consequence of the special memory layout of the active objects and our garbage collector implementation. Some compiler assumptions about the system organisation are hard-coded in the generated binary code. The memory layout used by the runtime environment to represent memory blocks and type descriptor structures are the most important assumptions in this context. The type descriptors contain information about the type of the record and the addresses of the type bound procedures (methods). This knowledge is used by the compiler to implement efficient type tests and entry calls to type-bound procedures without having to ask the runtime environment for this information.

The most relevant change in the memory organisation are process descriptors. Process-oriented information has to be stored somewhere in memory, and for efficiency reasons this data is stored at some fixed offsets relative to the object's base, making the retrieval of the needed data possible without any additional indirection or computation for memory access. Because positive offsets are already used for local instance data, negative offsets are used to store the process descriptor. The record tag (pointer to the type descriptor) now includes special information telling the garbage collector that this is an active object and that the original record tag is located below the process data. Duplication of the record tag is necessary because of our Mark-and-Sweep garbage collector.

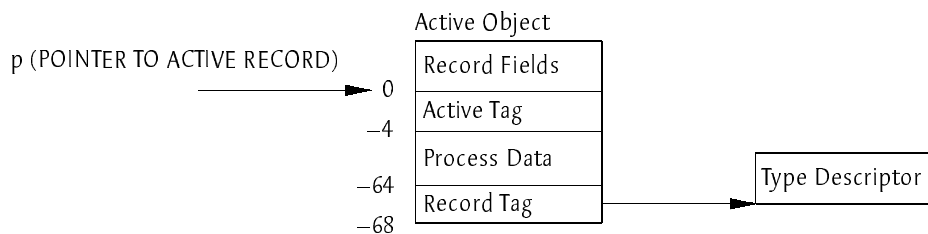


Fig. 6: Active Object Memory Layout

PASSIVATE

Another semantically complex instruction is passivate. A very simple implementation of passivate could be:

```

WHILE ~condition DO Kernel.PassControl END;

```

This implementation is very inefficient, because every single evaluation of the condition then requires a process switch. A much more efficient solution delegates the evaluation to the runtime environment thus avoiding a process switch. The condition in the PASSIVATE instruction is isolated in a procedure of type *PROCEDURE () : BOOLEAN* which is local to the procedure where PASSIVATE is called. The variables belonging to an outer scope are retrieved via static link, that is the linked list of all the procedure scopes. Access to local data without requiring a process switch is possible: the run-time environment pushes the pointer to the activation frame of the passivated procedure and then calls the condition

procedure. Therefore the condition procedure executes in the same context as the passivated procedure. For example, the type bound entry P is implemented as

```
hidden procedure @Guard1 (sl: StaticLink ): BOOLEAN
    enter
    return (sl.SELF.x # 1)

PROCEDURE P    enter
               call Kernel.Passivate (@Guard1, FP, SELF, FALSE)
               exit
```

The *FP* parameter is the current process frame pointer that will be used as a static link by the runtime environment to call the condition procedure. If a conditionless *PASSIVATE* is issued, no hidden local procedure needs to be generated, and a NIL pointer is passed to *Kernel.Passivate*. The fourth parameter (*FALSE*) is a hint to the runtime environment: the condition contains only local variables or fields and it needs to be reevaluated only when a protected entry of the same instance has been left. Note that this implementation is only possible if a 2-pass compiler is used, because the code for the condition has to be relocated.

ACTIVATE

The *ACTIVATE* instruction is mapped directly to the *Kernel.Activate* system call.

4. Conclusions

Considering the size of the compiled kernel we have succeeded in combining active objects and Oberon in a very efficient and elegant way. In the 30 Kbytes of kernel code we find memory management, interrupt handling, the whole process management and the scheduler described above. Having now a small but very powerful toolkit, we had to write test applications for proving its correctness and robustness. In additions to small simulation programs which take full advantage of all the new facilities, we are currently working on a server version of the Oberon system. All the services, the operating system itself and the network are based on the model of active objects. A beta version is currently running. It offers different services like online teletext, a dictionary, a print server, a file server and simple FTP and WWW servers. It turned out that the minimum number of primitives we added to the original operation system is powerful enough to implement a whole server system and that the Oberon idea of 'making things as simple as possible, but not simpler' successfully applies once more.

5. Acknowledgements

We would like to thank here Pieter Muller who has implemented the one-process native Oberon Intel System that was the base for our project. He brought a lot of good ideas into our discussions and helped us understanding the details of a native Kernel. His knowledge about the dark sides of PC-Hardware was indispensable.

Our thanks also go to Erich Oswald who wrote the 'Figures' graphic editor that was used for the design of the several figures in this text.

We also would like to thank J. Gutknecht for his helpful comments on earlier versions of this paper.

6. References

- [Bac93] Jean Bacon;
Concurrent Systems; An Integrated Approach to Operating Systems,
Database, and Distributed Systems; Addison Wesley; 1993
- [Cre91] Régis Crelier;
OP2: A Portable Oberon-2 Compiler;
Proceedings of the 2nd International Modula-2 Conference,
Loughborough, England, 58-67; 1991
- [Die94] Reinhard A. Dietrich;
Dynamic Stacks for Lightweight Processes; Diplomarbeit ETHZ; 1994
- [Gut96a] Jürg Gutknecht;
Oberon, Gadgets and Some Archetypal Aspects of Persistent Objects,
Information Sciences: An International Journal, 1996
- [Gut96b] Jürg Gutknecht;
Do the fish really need remote control?
Proceedings of the JMLC, Linz, Austria; 1997
- [Gut] Jürg Gutknecht;
Vorlesung: Simulation diskreter Systeme; ETH Zürich
- [Hoa75] C. A. R. Hoare;
Monitors: An Operating System Structuring Concept;
Communication of the ACM 18(2), 1975
- [Int91] Intel Corporation; 386 DX Microprocessor Programmer's Reference Manual; 1991
- [Mul96] Refer to <ftp://anonymous@ftp.inf.ethz.ch/pub/Oberon/System3/Native>
- [Rei91] M. Reiser;
The Oberon System: User Guide and Programmer's Manual;
Addison-Wesley, 1991
- [WG92] Niklaus Wirth, Jürg Gutknecht;
The Design of an Operating System and Compiler; ACM Press; 1992
- [Wir88] Niklaus Wirth;
The Programming Language Oberon;
Software – Practice and Experience 18(7): 671-690, July 1988