Adding Concurrency to the Oberon System

Spiros Lalis* and Beverly A. Sanders

Institut für Computersysteme
Swiss Federal Institute of Technology (ETH Zürich)
ETH Zentrum
CH-8092 Zürich
Switzerland

Abstract. The Oberon system, developed by Niklaus Wirth and Jürg Gutknecht is unusual in that, although it has a "single process multitasking" window user interface, it does not support concurrent execution of programs. This approach yields a simple design and works surprisingly well, but it does have limitations of practical importance. In this report we describe a project, Concurrent Oberon, which introduces concurrency into the Oberon system while maintaining the simplicity and spirit of the original system.

1 Introduction

Oberon is the name of both a programming language designed by Niklaus Wirth [RW92], and an operating system [WG92] for the Ceres workstation [Ebe87] designed by Wirth and Jürg Gutknecht.[2] Both the language, which supports object oriented programming, and the system are notable for the elegance of their designs and significant functionality with a very small use of resources.

In contrast to window systems that associate a process and a separate address space with each window, the Oberon system supports "single process multitasking" with a process called the Oberon loop. Associated with each window, or viewer object is a dynamically installed procedure referred to as a handler. Keyboard and mouse drivers are polled by the Oberon loop, and when an event is detected, a message is sent via an upcall to the handler of the affected viewer which processes it and returns control back to the Oberon loop.

The conventional method for executing a "program" is via special procedures called commands that are called from within the viewer handlers during event processing. Specifically, clicking the middle mouse button on the command name appearing in a viewer causes the handler for the viewer to call the corresponding procedure. Once the command execution is finished, the Oberon loop again polls for input events. As a consequence, all commands in the system are executed sequentially and the system does not respond to input events during command execution. This approach, with a single thread of control, simplifies the system

* Supported in part by Swiss National Science Foundation grant 21-25280-88.

[2] Oberon has been ported to other hardware platforms where it runs on top of a standard operating system.

significantly, and makes command execution efficient since there is no overhead due to context switching. Moreover, it works very well for interactive applications, since most event handling such as displaying a character is essentially instantaneous and the most commonly used commands require little time.

Nevertheless, there are cases where this model of control is less appropriate. For example, it is inconvenient to implement a long computation as a command because its execution will block the system until it is finished, perhaps for several minutes or even hours. Also, there are applications which react to external events (e.g. arrival of a packet over a network) and where polling via explicitly invoked commands is not an acceptable solution.

To accommodate such applications the Oberon system offers a mechanism called tasks. Tasks are special procedures that are invoked periodically from within the Oberon loop. In other words, tasks can be viewed as commands that are executed repeatedly in the background without an explicit user request.

Although this mechanism can be used to implement some simple applications, it does not adequately solve the above mentioned problems. Its most important deficiency is that task execution can be delayed arbitrarily long by input handling and long running commands. This makes it impossible to guarantee that a certain task will be executed within a reasonable amount of time, which is vital for applications with real time constraints such as connection-oriented network protocols.

Another problem is that tasks, like commands, run until they terminate, thus blocking the system during their execution. In order to use the task mechanism to implement background processing that still allows interactive use of the system, the computation must be broken down into a sequence of task invocations. Since tasks and the Oberon loop are executed on the same stack, this requires explicitly saving the intermediate state of the computation each time before returning control to the Oberon loop[Rei91]. This can be extremely cumbersome for programs that cannot be modeled as simple state machines, for example programs with recursion. In addition, since control transfer between a task and the Oberon loop is explicit, tasks must be implemented to execute "long enough" but not "too long". If this is not the case, either the task will block the system, or processor time will be wasted mostly in useless control transfers. However, thinking about how much time is needed to execute a particular piece of code has little to do with the problem the programmer is actually trying to solve. Such calculations are annoying, machine dependent, and even impossible to make if computation time is a function of parameters that are not known at implementation time.

Our experience with tasks showed that these limitations are indeed of practical importance, and therefore we decided to develop a special version of the Oberon system that would be free of these problems. An additional motivation was the desire to employ Oberon, which is currently used in all lower division computer science courses at ETH, for programming exercises in concurrent programming. Our objective was to offer basic facilities for straightforward background processing, timely response to events, and low level synchronization

primitives that would allow various higher level techniques used in teaching and research in concurrency to be easily implemented. An important constraint was that the simplicity and spirit of the original system should be maintained as much as possible.

2 Introducing Concurrency

To achieve the goals stated above, we implemented a new version of the Oberon system called Concurrent Oberon. Concurrent Oberon provides threads (or light-weight processes) along with a simple priority scheduler. Threads encapsulate the state of a computation and allow control of the processor to be passed to different threads without requiring the programmer to explicitly save the state. The scheduler recognizes three priority levels. Round robin scheduling is used to allocate processor time among threads at a given level, provided there are no waiting threads with higher priority.

Since the "single process multi-tasking" approach has been demonstrated to work very well for user-driven applications, we maintained this aspect of the original Oberon system and also use a single thread to handle all keyboard and mouse events. By convention, the Oberon thread, which handles input from the mouse and keyboard, runs with normal priority, background threads have low priority, and threads which execute rarely, but must react quickly to events are assigned high priority. This convention provides fast response time where this is critical and makes the presence of background threads essentially invisible to the user. For reasons of compatibility with the standard system, the task mechanism is still available.

2.1 System Structure

Integrating concurrency in an elegant way into a system that was designed to be sequential posed an interesting engineering problem. Complicated globally shared data structures are pervasive in the Oberon system and careless use of concurrency could cause serious inconsistencies. There are several shared data structures which are not entirely encapsulated in an abstract data type and hence are accessed directly by clients, rather than via procedures. In addition, many Oberon applications are implemented using the implicit assumption that a sequence of operations is executed atomically, i.e, it is often assumed that the state of an abstract data type will not change between two successive operations. Thus the obvious approach of adding synchronization to the operations of an abstract data type would not be sufficient.

In Concurrent Oberon, the Oberon thread is a special thread which, by convention, is responsible for handling all keyboard and mouse events, controlling the screen, and accessing high level data structures related to input and display handling. The decision to delegate this work to a single thread means that most explicit synchronization can be avoided. New threads may be created to perform background activities and access such data structures only by communicating

in a controlled way with the Oberon thread. This design provides the desired functionality without adding significant complexity, and as an important practical matter, remains compatible with the standard Oberon system. Existing applications can be run on Concurrent Oberon without changes.

An important consequence of this approach is that concurrency is not completely transparent. Programs which are to be run as threads must be programmed to do so, in particular, they must synchronize with the Oberon thread when accessing shared I/O related data structures. We do not feel that this is a significant restriction, because typical background threads do not interact with the display often and programming the required synchronization is not difficult.[3]

All threads in Concurrent Oberon occupy a single global address space. For each module, only a single image is loaded in memory. This means that global variables of the module are shared by all threads running procedures in the module or modules who import the variables. This fact affects the structure of a module whose procedures will be executed as threads, or whose data will be accessed concurrently.

The lack of memory protection between unrelated threads would be fraught with difficulties if we were using an unsafe language because small programming errors, especially with pointers and arrays, could easily cause system crashes by unintentionally modifying memory belonging to a different thread. In our case, the strong type checking provided by the Oberon programming language essentially guarantees that such errors will be detected by the compiler, or cause a trap at run time. Specifically, the compiler and run time system guarantee that all pointers always point to an object of a given type and which is in the scope of the code being executed, thus eliminating the sort of errors that cause system crashes by accidentally modifying memory belonging to unrelated threads. Our system also detects stack overflow and terminates the offending thread.

The main source of vulnerability to programming errors which can cause serious problems in Concurrent Oberon is failure to properly synchronize with the Oberon thread. According to our experience so far, this is not a serious problem because the critical shared data structures are well known and, as will be described in section 3.6, synchronization can be implemented in a straightforward way.

3 Threads

In this section, we describe the module Threads, which provides the new capabilities in Concurrent Oberon. This module provides procedures for creating and destroying threads, a set of operations that can be used to implement arbitrary synchronization tools (e.g. semaphores, signals, monitors, conditions), methods for synchronizing and communicating with the Oberon thread, and a few procedures used in system programs.

[3] An analogous lack of transparency exists in the standard system as well. Procedures that are to be executed as commands must follow a particular protocol to acquire input parameters, and are therefore implemented differently.

3.1 Thread Definition, Create and Destroy

A thread descriptor is a record (object) containing state information about the thread needed by the scheduler. The definition is given below.

```
TYPE
  Thread = POINTER TO ThreadDesc;
  ThreadProc = PROCEDURE();
  ThreadDesc = RECORD
    state, priority: SHORTINT; (*read-only*)
    incNo: LONGINT; (*read-only*)
  END;

PROCEDURE Create (this: Thread; proc, trapproc: ThreadProc;
  wsp: LONGINT);
PROCEDURE Destroy (this: Thread);
```

To create a thread, the programmer must first allocate a thread descriptor using `NEW` and call the procedure `Create` with parameters `body, trapbody` and `wsp`. `wsp` indicates the desired size of the workspace (stack) while `proc` and `trapproc` are procedures to be executed when the thread is started or experiences a runtime error, respectively. The latter procedure can be used to perform cleanup operations or to restart the thread. If trapbody is set to `NIL`, no actions will be taken. `Create` allocates a workspace and initializes the state and priority of the thread to the default values: `suspended` and `low`, respectively. If a newly allocated thread descriptor is used as a parameter, the field `incNo` is set to 1, else, if an old descriptor is used to host the execution, then the incarnation number is incremented by 1. A descriptor may not be reused unless the corresponding execution has terminated. Procedure `Destroy` destroys the specified thread and releases its resources.

The Threads module exports two read-only variables:

```
VAR cur, oberon: Thread;
```

The variable `oberon` indicates the Oberon thread, and variable `cur` indicates the thread that is currently executing, and is updated by the scheduler each time control is transferred.

Parameters can be passed to a thread by using the type extension facility of the Oberon language [RW92]. New descriptor types can be defined by augmenting the base thread descriptor with new fields, and instances thereof may be used to invoke the `Create` operation. Thread procedures can access the additional fields of the descriptor of the currently executing thread using type guards on the variable `cur`. An example is given in section 3.5.

3.2 Priorities

The scheduler recognizes three priority levels and considers a thread for execution only when there are no ready threads of higher priority. Undesired starvation is avoided by giving threads appropriate priorities and by changing their priority when this is required. The priority of a thread is given in the `priority` field of the thread descriptor and can be changed by the procedure `SetPriority`.

```
CONST
  low = 0; norm = 1; high = 2;

PROCEDURE SetPriority (this: Thread; prio: SHORTINT);
```

As an example of priority adjustment, the Oberon thread performs input handling at a normal priority and decreases its priority when no more input events are detected to allow execution of low priority threads. Conversely, as soon as keyboard and mouse activity is sensed the priority of the Oberon thread is reset to normal. High priority should be reserved for threads that are suspended most of the time waiting for events that must be handled very quickly.

3.3 Control Operations

The following exported constants and procedures control the control state of threads as indicated by the `state` field of the thread descriptor.

```
CONST
  ready = 0; asleep = 1; suspended = 2; destroyed = 3; trapped = 4;

PROCEDURE Suspend;
PROCEDURE Resume (this: Thread);
PROCEDURE Pass;
PROCEDURE Sleep (msecs: LONGINT);
```

A thread whose state is `ready` is either running or waiting for the scheduler to allocate the processor. The state of a thread is changed to `suspended` by executing `Suspend`. A thread which is suspended will become ready when another process executes `Resume`. Immediately after the call of `Create`, a thread has a state of `suspended` and `Resume` must be called in order to change its state to `ready` and allow its execution. `Resume` does not specify the next thread to be called by the scheduler, but rather changes the state of the specified thread so that it may be chosen by the scheduler. `Sleep` changes the state of the thread to `asleep`. A thread with state `asleep` will become `ready` when resumed by another thread or after the specified amount of time has expired. States `destroyed` and `trapped` indicate that the thread has been destroyed or has experienced a run-time error, respectively. `Pass` calls the scheduler in order to pass control to another thread without changing the state of the originating thread.

In order to implement higher level synchronization primitives, we offer primitives that can be used to define atomic segments, i.e. sequences of instructions which should be executed atomically.

```
PROCEDURE BeginAtomic;
PROCEDURE EndAtomic;
```

`BeginAtomic` marks the current thread so that it will not be preempted by the scheduler. `EndAtomic` unmarks the thread. Thus a sequence of statements between a `BeginAtomic, EndAtomic` pair will be executed atomically.[4] Properly nested `BeginAtomic-EndAtomic` pairs are also allowed. Within an atomic section, control can be explicitly passed to other threads using `Suspend, Sleep` or `Pass`. When the thread resumes execution, the remaining segment will be atomic. Atomic segments are intended to be used primarily for the implementation of higher level synchronization primitives, not as an all purpose synchronization device. We have introduced atomicity as a high level concept. Thus the implementation of synchronization primitives using the Threads module does not depend on how preemption is implemented. This is in contrast to the usual approach of requiring the programmer to turn off interrupts to achieve atomicity. This approach requires the programmer to turn off all possible sources of interrupts that might cause preemption, which may introduce unnecessary delays and requires modification of synchronization primitives if new interrupt sources are added later. It might also be the case that "preemption" is not implementeded via interrupts. For example one could modify the compiler to insert instructions (which are not under the programmers control) for releasing the processor.

3.4 Semantics of Operations

In this section, we give formally defined abstract implementations of the above operations. In particular the definitions of the control operations serve to clarify the semantics of the primitives as well as allow formal reasoning about programs using them. In order to specify the abstract implementations, we will use the programming notation given in [And91]. This notation is an extension of Dijkstra's notation including an await statement for synchronization, and using angle brackets "<>" to enclose atomic commands, or sequences of commands. The proof rules described in [And91], can be used for formal correctness proofs.

```
PROCEDURE Create (this: Thread; proc, trapproc: ThreadProc;
  wsp: LONGINT);
    <stack of size wsp allocated ∧ this.state := suspended
    ∧ this.priority := low>


PROCEDURE Destroy (this: Thread);
  <if
```

[4] Hardware interrupts are still serviced within an atomic segment.

```
    this.state < destroyed → this.state := destroyed
  □
    this.state >= destroyed → skip
  fi>


PROCEDURE SetPriority (this: Thread; prio: SHORTINT);
  <this.priority := prio>


PROCEDURE Suspend;
  <th.state := suspended>;<await(th.state = ready)>

PROCEDURE Resume (this: Thread);
  <if
    this.state < destroyed → this.state := ready
  □
    this.state >= destroyed → skip
  fi>

PROCEDURE Pass;
  skip

PROCEDURE Sleep (msecs: LONGINT);
  <t := Time; th.state := asleep>;
  <await(Time ≥ t+msecs ∨ th.state = ready) → th.state := ready>
```

In the above, `t` is a fresh variable local to the thread and `th` is the (fixed) name of the current thread. The variable `Time` represents the value of the system clock. `Time` is modified only by a clock process which repeatedly increments its value. As a result, the only assertions containing `Time` which are allowed are those which will not be falsified by increasing its value , for example `Time` $\geq$ `n`. Typically, one does not formally reason with real time but uses `Sleep` to detect failures or indicate "wait only a finite amount of time", with the parameter serving as a hint to the scheduler about how long to wait. The value of the parameter then influences performance rather than correctness.

The procedures `BeginAtomic` and `EndAtomic` are equivalent to angle brackets. Calls to `Pass` and `Suspend` and `Sleep` appearing between `BeginAtomic` and `EndAtomic` explicitly release atomicity, which is continued after the call until the closing `EndAtomic` statement. In other words, in the context of a `BeginAtomic`, `EndAtomic` pair, the definition of `Suspend` and `Sleep` are modified by removing the initial "<" and the final ">". The definition of `Pass` becomes `"> skip <"`. For example: `BeginAtomic S1; S2; Suspend; S3; S4 EndAtomic` is equivalent to `< S2;S2 ; th.state := suspended>;<await(th.state = ready) ; S3;S4>`. The statement sequence `BeginAtomic S1; S2; Pass; S3; S4 EndAtomic` is equivalent to `< S2;S2 >;< S3;S4>`.

3.5 Example

To demonstrate how threads can be programmed using these primitives we give an example of a thread that increments a variable approximately every T *msecs*. The value of the increment and T are passed to the thread as parameters. In addition, to guarantee that the variable is incremented during input handling, the priority of the thread is set to normal:

```
TYPE
  MyThread = POINTER TO MyThreadDesc;
      (*MyThreadDesc is extension of base type Threads.ThreadDesc*)
  MyThreadDesc = RECORD (Threads.ThreadDesc)
    i, inc, T: LONGINT
  END;

PROCEDURE Body;
BEGIN
  (*Type guard to ensure that Threads.cur has type MyThread*)
  WITH Threads.cur: MyThread DO
     LOOP Threads.Sleep(Threads.cur.T);
       INC(Threads.cur.i, Threads.cur.inc)
     END;
  END;
END Body;

PROCEDURE Start;
  VAR t: Thread;
BEGIN
     (* Allocate thread descriptor and initialize added fields*)
   NEW(t); t.inc := 100; t.i := 0; t.T := 50;
     (* Allocate workspace and initialize base fields *)
   Threads.Create(t,Body,NIL,128);
     (* Increase priority, default was low*)
   Threads.SetPriority(t,Threads.norm);
     (* Change thread state to ready to allow execution *)
   Threads.Resume(t)
 END Start;
```

3.6 Synchronization with the Oberon thread

In order for a thread to interact with Oberon, for example to display output, it must modify data structures which are also modified by the Oberon thread. In this section we describe two sets of primitives which have been provided in order to allow threads to synchronize with the Oberon thread without causing inconsistencies in shared data structures, and also without requiring explicit synchronization in programs which will be executed solely by the Oberon thread. The

first approach ensures mutually exclusive access to shared data structures, while the second is similar to asynchronous message passing to the Oberon thread, which performs the requested command. Explicit synchronization is not necessary for file operations; in this case serializability is automatically guaranteed by the system.

Locking Shared Data Structures The easiest way to synchronize with the Oberon thread is using a special lock maintained by the Threads module. The lock is acquired and released with the following two procedures:

```
PROCEDURE LockOberon;
PROCEDURE UnlockOberon;
```

The lock is held initially by the Oberon thread and is released and reacquired between command and task execution. If a critical data structure is to be modified from within a thread, `LockOberon` must be invoked prior execution of the corresponding operations, and `UnlockOberon` must be called afterwards to release the lock. `LockOberon` suspends the current thread if another thread already holds the lock. Blocking is very likely to happen, since the lock is held by the Oberon thread during command and task execution. Nesting of these commands is allowed so that a command containing them may be executed from within the Oberon loop as well as by a background thread.

For example, if we wanted to modify the thread implemented in section 3.4 to print a message on the system log-viewer each time the variable is incremented, we could do this as follows:

```
PROCEDURE Body1;
  VAR W: Texts.Writer;
BEGIN
  WITH Threads.cur: MyThread DO
    Texts.OpenWriter(W); (*initialize writer*)
    LOOP
       Threads.Sleep(Threads.cur.T);
       INC(Threads.cur.i, Threads.cur.inc);
         (*write value to buffer*)
       Texts.WriteString(W, "i = ");
       Texts.WriteInt(W, Threads.cur.i, 0);
         (*display buffer contents in log viewer*)
       Threads.LockOberon;
       Texts.Append(Oberon.Log, W.buf);
       Threads.UnlockOberon
    END
  END
END Body1;
```

In the above example, `W` of type `Texts.Writer` is local to the thread and contains a buffer into which the desired output is written. The command `Texts.Append(Oberon.Log,W.buf)` appends the contents of the buffer to the Oberon Log, which is a shared text usually displayed in a viewer on the screen.

Asynchronous Communication An alternative approach to displaying output or otherwise accessing data structures shared with the Oberon thread is via "asynchronous message passing". This technique requires more effort to implement than using `LockOberon` and `UnlockOberon`, but is more appropriate when it is important that a thread produce output without blocking (e.g. monitoring a network or within a trap handler).

```
TYPE
  OberonAction = POINTER TO OberonActionDesc;
  OberonActionProc = PROCEDURE (this: OberonAction);
  OberonActionDesc = RECORD
    body: OberonActionProc
  END;

PROCEDURE QueueOberonAction (this: OberonAction);
PROCEDURE DoOberonActions;
```

The main idea of this approach is to put operations which access data structures shared with the Oberon thread into a queue which is processed from within the Oberon thread itself. `DoOberonActions` is not intended to be used by applications programmers but is called from within the Oberon thread. When it is called, it processes all `OberonActions` in the queue by executing the procedure which has been installed in the body field of the `OberonActionDesc` with the corresponding `OberonAction` pointer as a parameter. As with thread descriptors, arbitrary parameters can be passed to the body procedure by extending `OberonActionDesc` with additional fields.

We show how the previous example can be modified to use actions instead of locking.

```
TYPE
  MyAction = POINTER TO MyActionDesc;
  MyActionDesc = RECORD (Threads.OberonActionDesc)
    i: LONGINT
  END;

VAR W: Texts.Writer;

PROCEDURE ActionBody (this: Threads.OberonAction);
BEGIN
  WITH this: MyAction DO
    Texts.WriteString(W, "i = "); Texts.WriteInt(W, this.i, 0);
```

```
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END
END ActionBody;

PROCEDURE Body2;
  VAR a: Action;
BEGIN
  WITH Threads.cur: MyThread DO
    LOOP
      Threads.Sleep(Threads.cur.T); INC(Threads.cur.i);
        (*Create and initialize Action*)
      NEW(a); a.i := Threads.cur.i; a.body := ActionBody;
        (*queue action*)
      Threads.QueueOberonAction(a)
    END
  END
END Body2;
```

3.7 Systems Programming

Module Threads exports several procedures which are used by other system modules (Input and System) to implement preemption and trap handling. Although not intended to be used by application programmers, they appear in the interface and are briefly described here for completeness.

```
PROCEDURE Tick (ticks: LONGINT);

PROCEDURE Enumerate (proc: ThreadProc);
PROCEDURE StackState (this: Thread; VAR fp, pc: LONGINT);
```

`Tick` is used by the timer interrupt handler to advance the internal time of the scheduler. `Enumerate` applies an operation to all threads. `StackState` returns the last saved state of a thread, i.e. the frame pointer of the last procedure activation record and the address of the instruction last executed. This data is invalid for the currently executing thread. These procedures are used by the commands `System.ShowThreads, System.DumpThread, System.DestroyThread` and `System.DestroyAllThreads` which provide thread status information and control to the user.

The following procedures of module Threads may be called from withing interrupt handlers: `Resume, SetPriority, Pass,` and `Destroy`.[5]

[5] If destroy is called while the specified thread is within an atomic segment, the thread is marked, and its destruction actually carried out when the thread leaves the atomic segment. This allows, for example, the Oberon thread to be interrupted from the keyboard (with cntl-shift-del) safely.

4 Thread Status and Control

Module System has been extended with three new commands which allow the user to monitor and destroy threads via interactive commands.

```
PROCEDURE ShowThreads;
PROCEDURE DumpThread;
PROCEDURE DestroyThread;
PROCEDURE DestroyAllThreads;
```

Command `ShowThreads` opens a viewer that contains all running threads. For each thread, its identification number, body procedure and state are displayed. `DumpThread` opens a viewer with the stack dump of the selected thread and `DestroyThread` destroys the selected thread. In both cases, the target thread is identified by selecting its number on the screen. `DestroyAllThreads` destroys all threads except the Oberon thread. This should be done before turning the machine off to avoid potential inconsistencies in the file system.

5 Implementation

5.1 Module Structure

The module structure of the new system is given in figure 1. Modules belonging to the standard system which are not shown have been used without changes in Concurrent Oberon.

5.2 Scheduling

The scheduling policy has been designed to provide processor sharing between threads of the same priority, to make the presence of background threads invisible to the user, and to guarantee fast response time for specific events. The mechanism for preemption is a call to the scheduler from within an interrupt handler using the procedure `Threads.Pass`. The timer interrupt handler in module Input calls this procedure periodically in order to allow round robin scheduling of ready threads.

The priority of the Oberon thread is set to `normal` and the scheduler called whenever keyboard and mouse events are present. Mouse events are detected by polling the mouse device from within the timer interrupt handler. Keyboard events automatically result in an interrupt, which is also handled by module Input. After all input events have been processed the Oberon thread sets its priority to `low`, giving again low priority threads a chance to execute. With this scheme, background threads can utilize cycles not needed for interactive execution without affecting the response time of the interactive system. The structure of the Oberon thread is as follows:

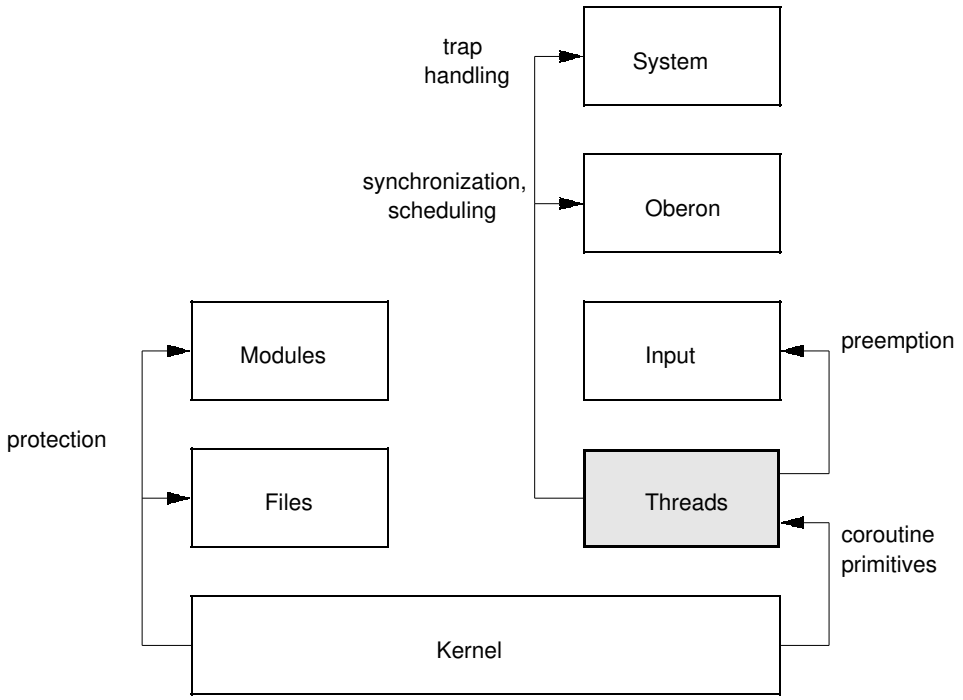

Fig. 1. Module Structure of Concurrent Oberon

```
Oberon-lock is already acquired
LOOP
  WHILE keyboard or mouse events present
  DO
    handle event;
    execute next task
    Threads.DoOberonActions;
  END;
  Threads.SetPriority(Threads.oberon, low);
  Threads.UnlockOberon ;
  Threads.Pass;
  Threads.LockOberon;
END;
```

5.3 Trap Handling

In the standard Oberon system, whenever a run time error occurs, a viewer containing the stack dump of the failed execution is opened from within the

trap handler. In Concurrent Oberon, the trap handler is implemented in module Threads and uses the facilities for asynchronous communication with the Oberon thread to display trap information. An action which opens a viewer and prints the stack state is defined as a special `OberonAction`. On the occurrence of a runtime error, a new action is created, its fields are initialized with the state information provided by the system's kernel, the action is queued using `QueueOberonAction`, and control is switched to an available thread. A viewer displaying the corresponding trap information is opened at a later point in time, when the Oberon thread finally processes the queued `OberonAction`.

5.4 Stack Management and Garbage Collection

Module Kernel was extended to provide a set of primitives supporting simple coroutines [Wir83], thus providing the basic mechanism for implementing concurrency in the system. Module Threads uses these primitives to implement the desired scheduling policy.

```
TYPE
  Stack = POINTER TO StackDesc;
  StackDesc = RECORD
     status: SHORTINT;
     inSVC: BOOLEAN;
   END;

PROCEDURE NewStack (VAR stack: Stack; wsp: LONGINT);
PROCEDURE InitStack (stk: Stack; proc: PROCEDURE);
PROCEDURE Transfer (stk: Stack);
PROCEDURE Current (VAR cur: Stack);

PROCEDURE StackState (this: Stack; VAR fp, pc: LONGINT);
```

`NewStack` allocates stack space of size `wsp` and returns a pointer to the corresponding stack descriptor. Stacks are collected by the garbage collector as soon as they are no longer referenced and the corresponding memory is returned to the system. To initialize an execution on a stack, `InitStack` must be called with the `stack` and the procedure `proc` as parameters. The execution is started when control is transferred to the initialized stack for the first time. Control transfer is implemented by procedure `Transfer` that implements the appropriate actions depending on the status of the current execution. This information is recorded in the fields `status` and `inSVC`. The `status` field indicates whether the execution is in normal mode, trap mode, or interrupt mode. In the latter case, `inSVC` indicates if a supervisor call has been interrupted. The stack of the current execution is retrieved with calls to procedure `Current`. Finally, procedure `StackState` can be used to retrieve the state of a stack's execution. Data returned by `StackState` is invalid for the current stack. In the current version of the system, these procedures have been implemented as supervisor calls which allows the interface of

module Kernel to be extended without requiring the symbol file to be changed. This is desirable since changing a symbol file of a module results in the need to recompile all clients of the module.[6]

To implement synchronization within module Files, two procedure variables are defined:

```
VAR lock, unlock: PROCEDURE;
```

These variables initially contain empty procedures, and at a later point in time, are overwritten by module Threads with real synchronization primitives (i.e. `BeginAtomic` and `EndAtomic`). This indirection allowed module Threads to be placed at a higher level in the module hierarchy of the Oberon system, namely outside the "inner core".

Additional changes to the Kernel module were modifications of the garbage collector to mark pointers on all stacks. In the standard system, there is only one stack belonging to the Oberon loop and the garbage collector is called only between commands when the stack is guaranteed to be empty. Obviously, in Concurrent Oberon, there is never a time when all stacks can be guaranteed to be empty, thus the more elaborate collector is required.

In older versions of the Concurrent Oberon system, stack memory was mapped on a separate address region and the memory management hardware of the processor was used to detect instances of stack overflow. In the current implementation stacks are allocated on the heap and like any other object, are automatically collected by the garbage collector. A modified Oberon compiler is used to make checks for stack overflow on each procedure call. This allows a safe implementation of Concurrent Oberon on machines which lack memory management hardware, such as the Ceres-3 workstation at the expense of slightly more expensive procedure calls.

6 Conclusion

Concurrency was not supported in the original Oberon system. We have argued that, although the Oberon experience has demonstrated that concurrency is not needed to support interactive applications in a system with a multiple window user interface, concurrency is nonetheless desirable in a single user system. We have demonstrated that it can be introduced without a significant increase in size or complexity by developing the Concurrent Oberon system. Since all interfaces remained the same or were extended, the semantics of the Oberon loop were preserved, Concurrent Oberon is completely compatible with the standard system and all applications run without change. So far, Concurrent Oberon has been successfully used as a basis for a research project on distributed programming [Lal94], several senior thesis projects (including the implementation of standard communication protocols TCP/IP [Ste92] which significantly increases

[6] If we had been willing to change the symbol file of the Kernel, the interface could have been simplified by replacing some of the procedures with exported variables.

the ability of the Oberon system to communicate with the outside world), and for student exercises in distributed systems and concurrent programming.

Acknowledgments: We would like to thank Professor Niklaus Wirth for providing the sources of the Oberon system, thereby making this project possible, and also for modifying the Oberon compiler to detect stack overflow. Professor Hanspeter Mössenböck, Martin Gitsels, and many others in the Institut für Computersysteme provided helpful comments on an earlier version of this paper. Finally, we would like to thank Philipp Heuberger and Martin Gitsels, who used Concurrent Oberon and provided valuable feedback during the development of the system.

References

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, 1991.

[Ebe87] Hans Eberle. *Development and Analysis of a Workstation Computer.* PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1987. Number 8431.

[Lal94] Spiros Lalis. *Distributed Object-Oriented Programming in a Network of Personal Workstations.* PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1994. in preparation.

[Rei91] Martin Reiser. *The Oberon System: User Guide and Programmer's Manual.* ACM Press, Addison-Wesley, 1991.

[RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula.* ACM Press, Addison-Wesley, 1992.

[Ste92] Michael Steiner. TCP/IP für Ceres. ETH Informatik Diplomarbeit (Senior Thesis), 1992.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* ACM Press, Addison-Wesley, 1992.

[Wir83] Niklaus Wirth. *Programming in Modula 2.* Springer Verlag, 1983.

This article was processed using the LaTeX macro package with LLNCS style