Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Improved Stackmanagement in the Active Object Kernel

Roger Keller

Master Thesis

March 2006

Supervising Professor: Prof. Dr. J. Gutknecht
Supervising Assistant: Daniel Keller

Programming Languages and Runtime Systems Group
Institute of Computer Systems
Swiss Federal Institute of Technology
Zürich, Switzerland

Abstract

The Active Object System (Aos) kernel builds the base of the Bluebottle operating system, which has been developed at the Swiss Federal Institute of Technology (ETH) Zürich as the successor of the Native Oberon operating system.

The Aos Kernel primarily acts as a runtime environment for the Active Oberon Language. It provides the ability to create Active Objects[1]. Whenever an Active Object is created, a new stack must be allocated for it. Until now these stacks were statically sized and 128 KB of memory were reserved for each. For many processes, this amount was never really needed and therefore quite a lot of virtual memory was wasted. For processes which basically use only a minimum of stack – e.g. a few bytes – 128 KB would have been lost for the lifetime of that process.

The solution presented deals with this by only allocating one physical page of memory when a process is created. When the system discovers that a process is running out of stack, a new kernel routine (the Overflow Handler) is called which assigns that process' stack one or more new pages. When it discovers that a process does not need a previously allocated page anymore, it collects that page and can then reuse it for other processes. This kind of scheme is referred to as *Stack Sharing.*

This report describes the concepts and realization for

- the new *Object File Format* for Bluebottle.
- the *Reflection API* which is used to inspect the meta data of a module.
- the changes to the *Parallel Compiler*, based on the new *Object File Format* and *Reflection API.*
- the *Overflow and Underflow handling* at runtime to guarantee that a process can still fulfill its task.

On top of this, the report deals with the impact of the thesis on the Parallel Compiler (PaCo) and some issues detected in the current Kernel. This work was done in scope of a Master Thesis at the Swiss Federal Institure of Technology (ETH) Zürich.

[1]also known as lightweight processes or threads

Contents

Chapter 1.

Introduction

At the Computer Systems Institute of the Swiss Federal Institute of Technology (ETH) Zürich the Bluebottle[1] operating system is being developed. It is the successor of the Native Oberon[2] operating system.

The task of this master thesis contained three parts. Part one was to design and implement a new object file format which contains detailed meta information about the source code. Part two was to implement a new Reflection Application Programming Interface based on the meta data from part one. Part three was to evaluate mechanisms of stack sharing and to design and implement a new stack management for the Aos Kernel.

1.1. Problem Description

1. **A new Object File Format**
   The first task was to design and implement a new object file format for Bluebottle. The main motivation to have a new object file format was to have more sophisticated meta data about the modules' source code. Therefore the main focus was on the completeness and compactness of the meta data in a way that it still allows the Reflection API to work efficiently. The new object file format is described in appendix A.
2. **Reflection API**
   The second task was the new Reflection API, based on the meta data in the new object file format. The Reflection API should give the programmer detailed meta information about the source code of a module. It is described in chapter 3.
3. **Improved Stack Management**
   The last task was the new Stack Management. The main goal here was to break the limitations with respect to the maximum number of processes in the current Aos Kernel. The idea is that it should be possible in principle to have millions of active objects in the system. Chapter 4 points out how this was be achieved.

[1]see http://www.bluebottle.ethz.ch
[2]see http://www.oberon.ethz.ch

1.2. Motivation

In the Active Object System, an improved stack management is desirable for the following reasons:

- To support hundreds of thousands or even millions of threads
- A simulation e.g. of a population might need one thread for each individual in the simulated system
- Asynchronous Neuronal Networks can be modeled in an asynchronous manner with a thread per node
- Object-Oriented Databases might want to have threads representing database objects
- Component based systems need one thread per component

1.3. Terminology

Within this report several terms are used interchangeably. The term *Active Object* in Aos is an object which corresponds to a *lightweight process* or *thread*. The terms *process* and *lightweight process* are herein used synonymously. The terms *Aos* and *Aos Kernel* are used to describe the kernel of the Bluebottle operating system. The term *Bluebottle Operating System* is equivalent to *Aos Operating System*, since *Aos* is used for both of them.

The terms *Stack Area*, *Stack Overflow Area*, *Stack Region* and *Stack Overflow Region* are used to describe a single part of the stack of a process. The whole stack is always referred to as *Stack*.

The term *Stack Overflow* is used to describe the situation when a process discovers that the currently allocated stack space is not sufficient to execute a new procedure. The term *Stack Underflow* is used to describe the opposite situation when a stack does no longer need a specific *Stack Overflow Area*.

1.4. Overview

Chapter 2 gives some background about the Bluebottle operating system. Chapter 3 introduces the new Reflection Application Programming Interface (API). Chapter 4 describes the new stack management which was developed during this master thesis, and chapter 5 is a brief discussion about the implications of the thesis on the Parallel Compiler (PaCo).

Chapter 2.

Background

The Aos operating system is based on the Aos Kernel which was developed by P. J. Muller in his Phd thesis (see [1]). Bluebottle originates from the Programming Languages and Runtime Systems Group[1] in the Computer Systems Institute[2] at the Swiss Federal Institute of Technology (ETH) Zürich. It is a successor of the Native Oberon operating system. This chapter gives an overview of the current Aos focusing on the task(s). More precisely the Aos Kernel – the heart of Bluebottle – and the Active Oberon Language – an extension of the Oberon programming language – is presented. In order to understand Aos, a description of concepts must be given first. If the descriptions are not architecture independent, Intel's IA-32 architecture (see [2]) is referenced.

2.1. General Concepts

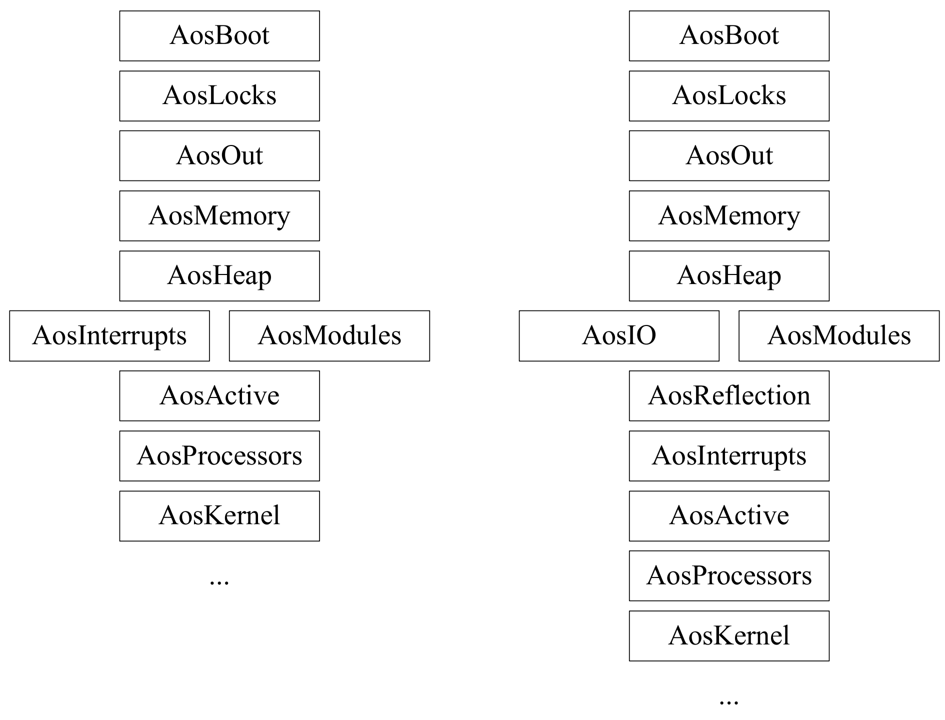
Modular Structure

The modular structure establishes the first level of abstraction according to the modular programming style. A module encapsulates data structures, types and procedures associated with each other. The import mechanism allows to use symbols – data structures, types and procedures – that are exported by other modules. The import graph *must* be acyclic. Thus the module hierarchies are simple and easy to understand. Figure 2.1 shows a diagram of the old and the new kernel module hierarchy. Table 2.1 gives a brief description of the first few modules in the Aos kernel.

2.2. Aos Kernel

The Aos kernel was originally designed and built by P. J. Muller ([1]). The first part of its modular layout is shown in figure 2.1(a). A short description of the modules is given in table 2.1.

[1]http://www.jg.inf.ethz.ch
[2]http://www.cs.inf.ethz.ch



(a) Module hierarchy in the old kernel (b) Module hierarchy in the new kernel

Figure 2.1.: The kernel module hierarchy

Module	**Description**
AosBoot	The first module in the hierarchy, gets control from the boot loader.
AosLocks	Low-level kernel locks (spin-locks).
AosOut	Low-level output module for tracing.
AosMemory	Memory management.
AosHeap	Heap management and garbage collector.
AosInterrupts	Low-level interrupt handling.
AosActive	Process management and scheduler.
AosProcessors	Processor management for Symmetric Multi-Processors (SMP).
AosIO	Input / Output interface.
AosModules	Module and type management.
AosReflection	Reflection API.

Table 2.1.: Aos kernel modules

The `AosActive` module is most certainly the heart of the Aos kernel. All runtime support for active objects can be found in it. Every active object runs its activities within the context of a lightweight process. The `AosActive` module is also responsible for the process management and scheduling. When an active object is instantiated, a new process is created. That process will run the code in the active object's body. Moreover every process has a priority assigned corresponding to its importance. Aos has five priority levels:

- MinPriority (=0): Lowest priority available, only idle processes should have this priority.
- Low (=1), Normal (=2) and High (=3): Priorities for user programs.
- MaxPriority (=4): Used by interrupt handlers (e.g. device drivers).

Aos provides a global ready queue for all processors. It is structured by priorities. If a thread finishes execution, the termination is prepared and performed by the `AosActive` module.

Critical Sections & Synchronization

Aos allows *every* object to act as a monitor (see [3]). During execution every object can be accessed exclusively. The monitor does not only control mutual exclusion but also provides a way to synchronize active objects – through the `AWAIT` statement in Active Oberon. `AWAIT` is a smart way to communicate with other threads using boolean conditions. As long as the boolean condition is false the process will be waiting. The necessary system calls are implemented in the `AosActive` module.

Memory Management

Aos uses one contiguous 32-bit virtual address space which is basically separated into the heap and the stack area. The heap virtual pages are mapped directly to the physical pages (as long as possible). This is used especially by device drivers, which need direct memory access. In the stack area the virtual memory pages are not mapped directly to physical memory pages. There the paging mechanism is used to enlarge a process stack if necessary. The page size according to [2] and [4] is set to 4 KB. When a process is started, its stack will be initialized to one physical page, although 128 KB are reserved in the virtual address space.

If the process crosses its stack border, a page fault is raised and that in turn adds a new page to the stack. The maximal stack size is limited to 128 KB. If a stack tries to allocate more memory space, a stack overflow exception will be raised. The first page of memory remains unmapped to detect NIL pointer accesses.

Dynamic Loading

In Aos all the code resides in modules. Aos loads the needed modules dynamically. Then the compiler generates an object file for every module. It contains the important information about the module. When a module is used the module loader will load it into memory and link it with other modules if necessary. The loaded module object contains its execution code and additional information. Building a new object file format was a part of this thesis. The result is shown in appendix A.

Garbage Collection

The Aos kernel comes with a Garbage Collector (implemented in `AosHeap`) which manages unused memory blocks in the heap. It belongs to the class of mark and sweep garbage collectors.

The most complete in-depth Aos kernel description can be found in [1].

2.3. Active Oberon Language

The Active Oberon Language is an object oriented programming language. It evolved from the Oberon language originally designed by Niklaus Wirth and Jürg Gutknecht. The Aos kernel provides the runtime environment to the Active Oberon Language. A language report of the Active Oberon language can be found in [5].

Chapter 3.

Reflection API

The new object file format contains detailed meta data about the source of the module. This chapter describes the *Reflection API* that can be used to access the meta data at runtime. Some typical usages of the Reflection API can be found in the new import plug-in for PaCo or the new stack trace plug-in (`AosStackTrace`). Section A.5 in the appendix describes the meta data section in the new object file format. To use the Reflection API add `AosReflection` to the module's import list.

3.1. Load Reflection

Parsing and loading the meta data section of a module is usually done using one of the load commands in the Reflection API. These commands are described in the following sections.

3.1.1. Load Reflection from Object File

```
AosReflection.Load*(moduleName: ARRAY OF CHAR; forceLoad,
   cache: BOOLEAN): Module;
```

Listing 3.1: Definition of `AosReflection.Load`

Parameters

- `moduleName`
  The name of the module of which the reflection is to be loaded, e.g. „AosReflection“ to load the meta data from *AosReflection.Bbx.*
- `forceLoad`
  Iff `TRUE` forces loading the meta data from disk in any case. This can be used to make sure the internal cache of the Reflection API is invalidated.

- `cache`
  `TRUE` if the meta data should be inserted to the internal cache after it has been loaded.

Return Value

Returns an instance of `AosReflection.Module` or `NIL` if the meta data could not be loaded.

3.1.2. Load Reflection from loaded Module

```
AosReflection.LoadFromModule*(module: AosModules.Module):
   Module;
```

Listing 3.2: Definition of `AosReflection.LoadFromModule`

Parameters

- `module`
  Reference to the module of which the meta data is to be retrieved.

Return Value

Returns an instance of `AosReflection.Module` or `NIL` if the meta data could not be loaded.

3.1.3. Load Reflection from Program Counter

```
AosReflection.LoadFromPC*(VAR pc: LONGINT): Module;
```

Listing 3.3: Definition of `AosReflection.LoadFromPC`

Parameters

- `pc`
  Absolute program counter (`EIP`) which is used to locate the module. If the module owning the code can be successfully located, `pc` will contain the offset of the instruction relative to the module's code base. It can therefore afterwards be used for procedure lookups using the Reflection API.

Return Value

Returns an instance of `AosReflection.Module` or `NIL` if the meta data could not be loaded.

3.1.4. Load Reflection from Stream

```
AosReflection.LoadFromStream*(inputStream: AosIO.Reader):
   Module;
```

Listing 3.4: Definition of `AosReflection.LoadFromStream`

Parameters

- `inputStream`
  An instance of `AosIO.Reader` which is used to load the meta data. This can be a reader on a file or on a stream which operates on a buffer in memory. If it is a file stream, it must already point to the begin of the meta data section.

Return Value

Returns an instance of `AosReflection.Module` or `NIL` if the meta data could not be loaded.

3.2. Reflection Objects

There are quite a few objects that can be used to inspect the meta data of a module. The following sections describe these objects and their interfaces. The type `String` is an alias for `Utilities.String`. Figure 3.1 shows a simple UML diagram of the objects described in the following sections.

3.2.1. `Symbol`

```
TYPE Symbol* = OBJECT
  VAR
    name-: String;  (* the name of the symbol *)
    owner-: Symbol; (* the owner of the symbol *)

  (* write the fully qualified name of the symbol to buffer *)
  PROCEDURE GetFullName*(VAR buffer: ARRAY OF CHAR);
END Symbol;
```

Listing 3.5: Interface of Object `Symbol`

`Symbol` is the base class for all Reflection Objects. The method `GetFullName` can be used to get the fully qualified name for the symbol. For this method, the fully qualified name would be `AosReflection.Symbol.GetFullName`.

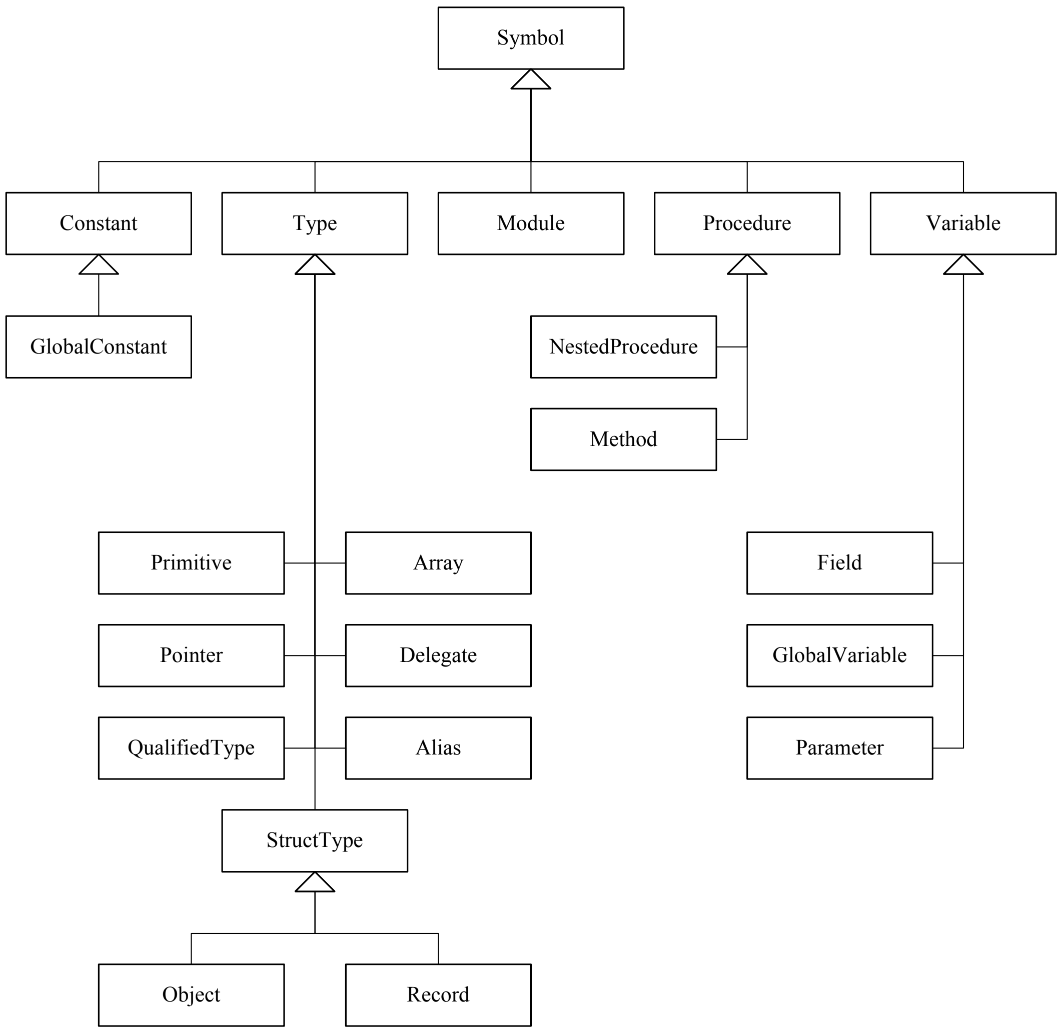


Figure 3.1.: Simple UML diagram of Reflection Objects

3.2.2. Module

```
TYPE Module* = OBJECT(Symbol)
  VAR
    fingerPrint-: LONGINT;
    version-: CHAR;
    targetArchitecture-: LONGINT;
    codeLength-, nofImports-, nofTypes-, nofDefinitions-,
   nofProcedures-, nofGlobalVariables-, nofGlobalConstants-:
   LONGINT;
    body-: Procedure;
    imports-: POINTER TO ARRAY OF String;
    types-: POINTER TO ARRAY OF Type;
    definitions-: POINTER TO ARRAY OF Definition;
    procedures-: POINTER TO ARRAY OF Procedure;
    variables-: POINTER TO ARRAY OF GlobalVariable;
    constants-: POINTER TO ARRAY OF GlobalConstant;
    maxStackSizeBody-: LONGINT;
    hasOperators-: BOOLEAN;

  (* find the type specified by typeName *)
  PROCEDURE FindType*(typeName: ARRAY OF CHAR): Type;

  (* find the definition specified by defName *)
  PROCEDURE FindDefinition*(defName: ARRAY OF CHAR):
   Definition;

  (* find the procedure specified by procName *)
  PROCEDURE FindProcedure*(procName: ARRAY OF CHAR): Procedure;

  (* find the global variable specified by varName *)
  PROCEDURE FindVariable*(varName: ARRAY OF CHAR):
   GlobalVariable;

  (* find the global constant specified by constName *)
  PROCEDURE FindConstant*(constName: ARRAY OF CHAR):
   GlobalConstant;

  (* find the symbol specified by name *)
  PROCEDURE FindSymbol*(name: ARRAY OF CHAR): Symbol;

  (* find the procedure which contains the relative pc *)
  PROCEDURE GetProcedureFromPC*(pc: LONGINT): Procedure;

  (* get an instance of the procedure iterator *)
  PROCEDURE GetProcedureIterator*(): ProcedureIt;

  (* set internal info (used for I/O plugins) *)
  PROCEDURE SetInfo*(fromLoadedMod: BOOLEAN; version: CHAR;
    name: ARRAY OF CHAR; targetArch, constLen, consts, codeLen:
    LONGINT);
END Module;
```

Listing 3.6: Interface of Object `Module`

`Module` is the main container for reflection objects. It contains all the types, definitions, procedures, global variables and global constants defined within the module's scope. It also provides the possibility to find the procedure that corresponds to a certain offset in the module's code section and therefore, together with `AosModules.Module`, allows one to easily discover what procedure is at a certain absolute address in memory.

3.2.3. Type

```
TYPE Type* = OBJECT(Symbol)
  VAR
    fingerPrint-: LONGINT;
    modifiers-: SET;

  (* returns TRUE iff the type is anonymous *)
  PROCEDURE IsAnonymous*(): BOOLEAN;

  (* returns TRUE iff the type is accessible from outside *)
  PROCEDURE IsPublic*(): BOOLEAN;
END Type;
```

Listing 3.7: Interface of Object `Type`

`Type` is the base class for all types (universal, primitive, user-defined as well as definitions). The method `IsAnonymous` specifies whether the type was inlined, like in the example given in listing 3.8.

```
VAR myHugeInt: RECORD high, low: LONGINT END;
```

Listing 3.8: Example of anonymous (inlined) types

3.2.4. StructType

```
TYPE StructType* = OBJECT(Type) (* abstract *)
  VAR
    size-: LONGINT;
    typeDescriptorOffset-: LONGINT;
    superType-: Type;
    nofFields-: LONGINT;
    fields-: POINTER TO ARRAY OF Field;
END StructType;
```

Listing 3.9: Interface of Object `StructType`

`StructType` is the *abstract* base class for composite types. It is used as base class for `Object` (see 3.2.5) and `Record` (see 3.2.6).

3.2.5. Object

```
TYPE Object* = OBJECT(StructType)
  VAR
    nofMethods-, nofImplements-: LONGINT;
    priority-: LONGINT;
    methods-: POINTER TO ARRAY OF Method;
    implements-: POINTER TO ARRAY OF Definition;

  (* returns TRUE iff the object's body is EXCLUSIVE *)
  PROCEDURE IsExclusive*(): BOOLEAN;

  (* returns TRUE iff the object is ACTIVE *)
  PROCEDURE IsActive*(): BOOLEAN;

  (* returns TRUE iff the object is a simple active object, see
    also section 4.3 *)
  PROCEDURE IsSimple*(): BOOLEAN;

  (* returns TRUE iff the object's body is SAFE *)
  PROCEDURE IsSafe*(): BOOLEAN;

  (* find the method given by methodName *)
  PROCEDURE FindMethod*(methodName: ARRAY OF CHAR): Method;

  (* find the symbol (field or method) given by name *)
  PROCEDURE FindSymbol*(name: ARRAY OF CHAR): Symbol;
```

```
END Object;
```

Listing 3.10: Interface of Object `Object`

`Object` describes an `OBJECT` composite type. It is also used to describe pointers to anonymous records (see listing 3.11 for an example).

```
TYPE MyHeapHugeInt = POINTER TO RECORD
    high, low: LONGINT
  END;
```

Listing 3.11: Example of a Pointer to an anonymous Record

3.2.6. Record

```
TYPE Record* = OBJECT(StructType)
END Record;
```

Listing 3.12: Interface of Object `Record`

`Record` describes a `RECORD` composite type. Its main purpose is to allow a faster distinction from `Object` than `StructType` does. Therefore the public interface is exactly the same as for `StructType`.

3.2.7. Pointer

```
TYPE Pointer* = OBJECT(Type)
  VAR
    toType-: Type;
END Pointer;
```

Listing 3.13: Interface of Object `Pointer`

`Pointer` is the container for `POINTER` types.

3.2.8. Array

```
TYPE Array* = OBJECT(Type)
  VAR
    length-: LONGINT; (* 0 for dynamic or open arrays *)
    elementType-: Type;
END Array;
```

Listing 3.14: Interface of Object `Array`

`Array` describes an array type. Please note that multi-dimensional arrays are modeled as a hierarchy of `Arrays` linked through the `elementType` field. Dynamic and open arrays do always have a (static) length of zero.

3.2.9. Delegate

```
TYPE Delegate* = OBJECT(Type)
  VAR
    returnType-: Type;
    nofParameters-: LONGINT;
    parameters-: POINTER TO ARRAY OF Parameter;

    (* returns TRUE iff the DELEGATE modifier was specified *)
    PROCEDURE NonStaticAllowed*(): BOOLEAN;
END Delegate;
```

Listing 3.15: Interface of Object `Delegate`

`Delegate` is used to describe a procedure handle. Please note that the parameters of a `Delegate` do not have a valid `offset` specified.

3.2.10. Alias

```
TYPE Alias* = OBJECT(Type)
  VAR
    aliasedType-: Type;
END Alias;
```

Listing 3.16: Interface of Object `Alias`

`Alias` describes an alias to an already existing type. An example is given in listing 3.17.

```
TYPE String = Utilities.String; (* shortcut *)
```

Listing 3.17: Example of an aliased type

3.2.11. QualifiedType

```
TYPE QualifiedType* = OBJECT(Type)
  VAR
    resolvedType-: Type;
    resolvedModule-: Module;

  (* makes sure that the type is being resolved if possible *)
  PROCEDURE Resolve*();
END QualifiedType;
```

Listing 3.18: Interface of Object `QualifiedType`

`QualifiedType` is used as a reference to another type either in the same module or in an imported module. Please note that (for performance reasons) only local references are automatically resolved. To resolve references to imported types, call the `Resolve` method.

3.2.12. `Primitive`

```
TYPE Primitive* = OBJECT(Type)
  VAR
    size-: LONGINT;
END Primitive;
```

Listing 3.19: Interface of Object `Primitive`

`Primitive` is used only to describe primitive and built-in types, such as `CHAR` and `INTEGER` but also `PTR` and `SYSTEM.BYTE`.

3.2.13. `Variable`

```
TYPE Variable* = OBJECT(Symbol)
  VAR
    type-: Type;
    offset-: LONGINT;
END Variable;
```

Listing 3.20: Interface of Object `Variable`

`Variable` is the base class for all variables. However, in its raw form it is only used for procedures' local variables.

3.2.14. `Field`

```
TYPE Field* = OBJECT(Variable)
  VAR
    modifiers-: SET;

  (* returns TRUE iff the field is public (r/w) *)
  PROCEDURE IsPublic*(): BOOLEAN;

  (* returns TRUE iff the field is public (read-only) *)
  PROCEDURE IsReadOnly*(): BOOLEAN;

  (* returns TRUE iff the field is an untraced pointer *)
  PROCEDURE IsUntraced*(): BOOLEAN;
END Field;
```

Listing 3.21: Interface of Object `Field`

`Field` describes a variable (a field) of a `RECORD`, `POINTER TO RECORD` or `OBJECT`.

3.2.15. `Parameter`

```
TYPE Parameter* = OBJECT(Variable)
  VAR
    modifiers-: SET;

  (* returns TRUE iff the parameter is passed by reference *)
  PROCEDURE IsVar*(): BOOLEAN;

  (* returns TRUE iff this is a SELF parameter for a method *)
  PROCEDURE IsSelf*(): BOOLEAN;
END Parameter;
```

Listing 3.22: Interface of Object `Parameter`

`Parameter` describes a formal parameter to a procedure, method or delegate.

3.2.16. `GlobalVariable`

```
TYPE GlobalVariable* = OBJECT(Variable)
  VAR
    fingerPrint-: LONGINT;
    modifiers-: SET;

  (* returns TRUE iff the variable is public (r/w) *)
  PROCEDURE IsPublic*(): BOOLEAN;

  (* returns TRUE iff the variable is public (read-only) *)
  PROCEDURE IsReadOnly*(): BOOLEAN;

  (* returns TRUE iff the variable is an untraced pointer *)
  PROCEDURE IsUntraced*(): BOOLEAN;
END GlobalVariable;
```

Listing 3.23: Interface of Object `GlobalVariable`

`GlobalVariable` describes a module-global variable.

3.2.17. Constant

```
TYPE Constant* = OBJECT(Symbol)
  VAR
    type-: Type;
    offset-: LONGINT;
    next-: Constant;
END Constant;
```

Listing 3.24: Interface of Object `Constant`

`Constant` is used to describe a constant value in a procedure's scope.

3.2.18. GlobalConstant

```
TYPE GlobalConstant* = OBJECT(Constant)
  VAR
    fingerPrint-: LONGINT;
    modifiers-: SET;

  (* returns TRUE iff the constant is public *)
  PROCEDURE IsPublic*(): BOOLEAN;
END GlobalConstant;
```

Listing 3.25: Interface of Object `GlobalConstant`

`GlobalConstant` is a `Constant` which is used to describe a module-global constant.

3.2.19. Procedure

```
TYPE Procedure* = OBJECT(Symbol)
  VAR
    fingerPrint-: LONGINT;
    modifiers-: SET;
    returnType-: Type;
    offsetStart-: LONGINT;
    offsetEnd-: LONGINT;
    nofParameters-: LONGINT;
    nofVariables-, nofConstants-, nofTypes-,
   nofNestedProcedures-: LONGINT;
    parameters-: POINTER TO ARRAY OF Parameter;
    variables-: POINTER TO ARRAY OF Variable;
    constants-: POINTER TO ARRAY OF Constant;
    types-: POINTER TO ARRAY OF Type;
    nestedProcedures-: POINTER TO ARRAY OF Procedure;
    inlinedCode-: InlinedCode;
    maxStackSize-, persistentStackSize-: LONGINT;

  (* returns TRUE iff the procedure is public *)
  PROCEDURE IsPublic*(): BOOLEAN;

  (* returns TRUE iff the procedure is to be inlined *)
  PROCEDURE IsInline*(): BOOLEAN;

  (* returns TRUE iff this is an overloaded operator *)
  PROCEDURE IsOperator*(): BOOLEAN;

  (* returns TRUE iff the procedure is an Oberon Command *)
  PROCEDURE IsCommand*(): BOOLEAN;

  (* returns TRUE iff the procedure is an Bluebottle Command *)
  PROCEDURE IsParCommand*(): BOOLEAN;

  (* returns TRUE iff this is a leaf procedure *)
  PROCEDURE IsLeaf*(): BOOLEAN;

  (* find the type given by typeName *)
  PROCEDURE FindType*(typeName: ARRAY OF CHAR): Type;

  (* find the nested procedure given by procName *)
  PROCEDURE FindProcedure*(procName: ARRAY OF CHAR):
   NestedProcedure;

  (* find the symbol given by name *)
  PROCEDURE FindSymbol*(name: ARRAY OF CHAR): Symbol;
END Procedure;
```

Listing 3.26: Interface of Object `Procedure`

`Procedure` describes a procedure. It is used as the base class for *nested procedures* and *methods*. It none of these sub classes are present, `Procedure` is a procedure in the module's scope.

3.2.20. NestedProcedure

```
TYPE NestedProcedure* = OBJECT(Procedure)
  VAR
    level-: LONGINT;

  (* always returns FALSE *)
  PROCEDURE IsPublic*(): BOOLEAN;
END NestedProcedure;
```

Listing 3.27: Interface of Object `NestedProcedure`

`NestedProcedure` describes a nested procedure. Because nested procedures can never be accessed from outside the owner procedure's scope, the `IsPublic` method always returns `FALSE`.

3.2.21. Method

```
TYPE Method* = OBJECT(Procedure)
  (* returns TRUE iff this is the constructor for an OBJECT *)
  PROCEDURE IsConstructor*(): BOOLEAN;

  (* returns TRUE iff this is the body method of an OBJECT *)
  PROCEDURE IsBody*(): BOOLEAN;

  (* returns TRUE iff this is an indexer *)
  PROCEDURE IsIndexer*(): BOOLEAN;
END Method;
```

Listing 3.28: Interface of Object `Method`

`Method` describes a method that belongs to an `Object`. The `Object` owning the method is stored in the `owner` field of the `Method`.

3.3. Applications

This section describes some possible applications of the new Reflection API. Another application is in the new stack management described in chapter 4. There the Reflection API is used to compute the spaced needed for procedure activation frames.

3.3.1. Garbage Collector: Pointer Recognition

The garbage collector in the old kernel used heurisitics to find pointers on the stacks. This could lead to objects and complete object structures being kept in memory unnecessarily because some – for instance `LONGINT` or `SET` – value on a stack looked like a pointer. To avoid this one can use the Reflection API to determine the exact locations of pointers on the stacks. The new module `AosObjectMarker` gives an example of how this can be done.

This way of marking objects and blocks in the heap is however not added to the system because there are several problems with the stacks. First, the `AWAIT` statements also create new frames on the stacks. But these frames do not belong to a real procedure – and they do also not conform to the calling convention – and therefore no meta data about them is availabe. Second, the interrupt handlers store the state of the interrupted process on the stack and they modify the stack frame in a way that they become invalid too, even though only temporarily. It would be necessary to redesign the evaluation of wait conditions and the interrupt handling to remove this flaw.

3.3.2. Object Serialization

In today's applications it is often required to serialize objects at runtime. Serialization is used for instance to make objects persistent by storing them on a harddisk drive or to distribute them over a network. The Reflection API can be used by a serialization library to inspect a class of objects and automatically serialize them to a byte stream. Of course also the deserialization form a byte stram can be achieved with the Reflection API.

3.3.3. Stack Traces & Module States

Another application for the Reflection API is the stack traces that are shown when an exception has occurred. The new stack trace plug-in `AosStackTrace` shows how this is done. `AosStackTrace` also shows how the module state can be inspected. The same mechanisms can be used to inspect – of course dynamically at runtime – all kinds of objects in the heap.

3.3.4. Other Applications

One could also think of a library to automatically generate a basic documentation for modules. This documentation would then only have to be extended with the description of the module's functionality. Another application is given in the new import plug-in (`PCImport`) for the parallel compiler.

Chapter 4.

Stack Sharing

4.1. The former stack layout

Until now each process was assigned a static stack of 128 KB of virtual memory when the process was created. At the beginning only the topmost of these 32 pages was physically allocated and mapped. The lowest page was never mapped to a physical page to easily detect stack overflows, indicated by a page fault. Page fault on any of the other 30 pages resulted in the corresponding page to be physically allocated and mapped. Therefore, once a stack has used all of its 31 pages 124 KB of physical memory were allocated to that process until it was terminated and finalized. This could result in a worst-case waste of physical memory of almost 124 KB per process. Figure 4.1 shows a simple scheme of the former stack layout.

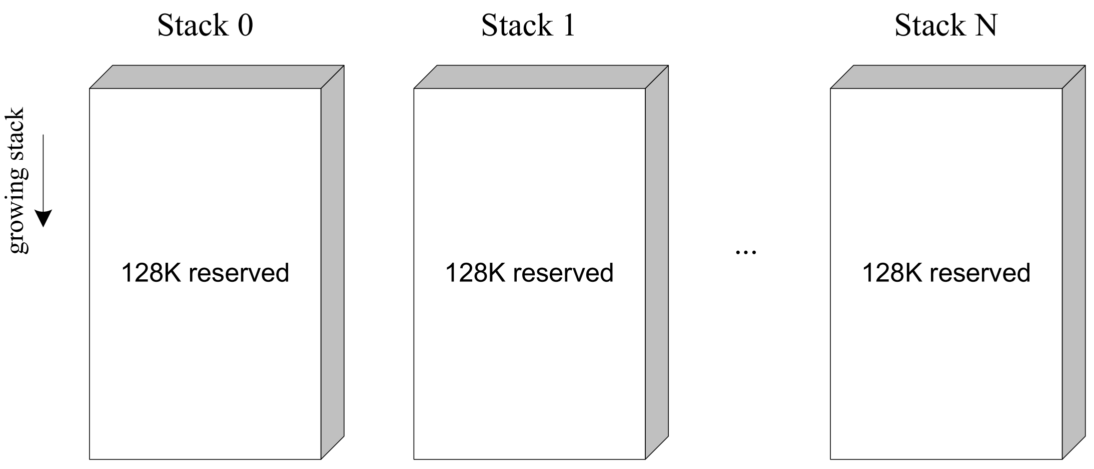


Figure 4.1.: Former layout of the Stacks

Because the stacks could only be allocated in the range between 2GB and 4GB, the maximum number of processes was given by $2GB \div 128KB \approx 16'000$. In fact it was less than 16'000 because the kernel stacks were also allocated in this region and the topmost 20MB are reserved for I/O devices (section 5.5 in [1] contains more detailed figures).

4.2. The new stack layout

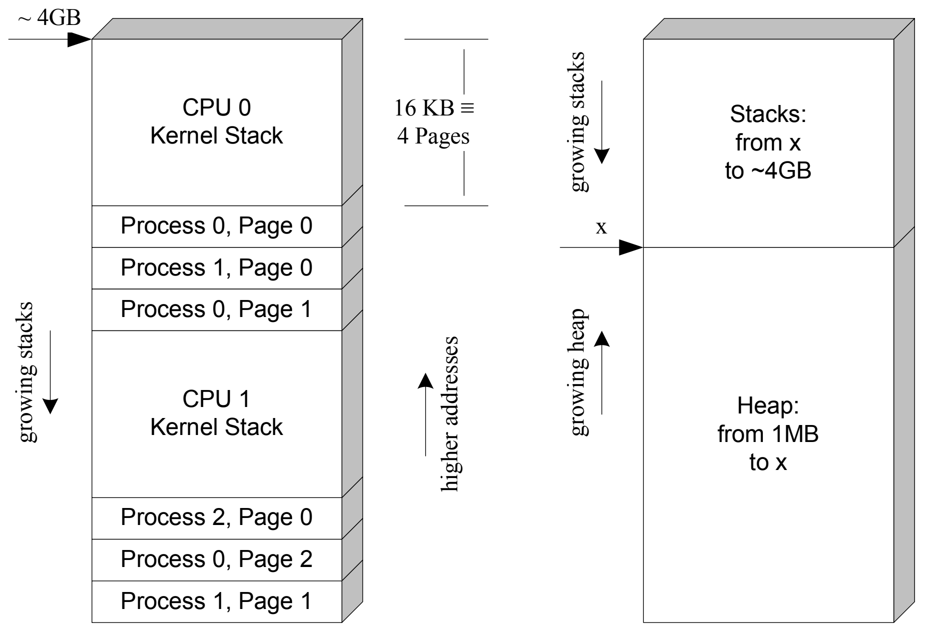
One of the most important goals of the thesis is to reduce the space wasted for stacks in virtual and physical memory. This was achieved with a combination of the stack sharing mechanisms from [6] and [7]. Figure 4.2 shows a scheme of the new stack layout. The main idea is that each created process gets exactly one page of physical and virtual memory. This is usually the topmost available virtual page and the physical page on top of the system's page stack. As long as the process is alive, this page will be assigned to it and the system will not reclaim it until the owning process is finalized. This step increases the maximum number of processes the system can have by a factor of 32. However, it's most likely that the processes will need more than the 4 KB available on a page. The situation when there's not enough space left on a page to execute a procedure is automatically determined by the process itself. Basically, the compiler injects runtime checks before the procedures' prologue and after its epilogue whose only task it is to detect these situations. Once such a situation is detected, the corresponding handler (as described in section 4.4) is called. The handler then allocates or deallocates (depending on the situation) stack for the process and resumes the process. This permits each process to have as much stack as it needs and reduces wasted space.

Together with a flexible boundary between heap and stacks, the maximum number of processes supported by the operating system no longer depends on the system's design, but much more on the effective amount of physical memory available as well as the design of the running processes.

4.3. Optimized Processes

The changes in the stack management described so far, does not allow to have several millions of processes in the system. To resolve, a category of optimized processes would need to be defined. With these processes in place it becomes possible to compute the maximum space needed on the stack at compile time. This however requires several restrictions that the body of the corresponding active object *must* meet. In particular it must fullfill the following conditions:

1. it *must not* call methods of the object itself or of any other object
2. it *must not* – directly or indirectly – have an `AWAIT` statement
3. it *must not* call assembler procedures or delegates
4. it *must not* call procedures from outside the module defining the object



(a) New Layout of the Stacks (b) Overall Memory Layout

Figure 4.2.: New Memory Layout

5. it *must not* call non-leaf procedures from inside the module

The first condition is required because methods can be overridden. In that case, the compiler can no longer decide how big the stack will be.

The second condition is an artificial condition. It makes the whole implementation a lot easier and less complex, if the kernel calls to `AosActive.Await` do not have to be considered. This is because the `AWAIT` statements can in principle contain calls to procedures or methods (see also the other four conditions). Also, the conditions can be evaluated on other processes' stacks which again makes the computation of the required space for those processes almost impossible.

The third condition is again artificial: Basically an assembler routine can issue any instruction (including `CALL`s) which by now cannot be checked in the compiler due to its design.

Condition number four is required because procedures from external modules can change in stack size without changing their fingerprints. Because the figures for stack size can so far only be computed at compile time, it is required that the called procedures do not change (or one would have to recompile the modules). This condition is somewhat related to condition number one.

The last condition is again an artificial condition to make the task and its implementation a bit easier. Basically this condition could also be changed to the less restrictive condition

> the body *must not* call – directly or indirectly – recursive procedures.

Section 4.7 discusses the handling of this optimized processes. The idea is that because the maximum stack size is known in advance, several optimized processes can be put on the same page, i.e. they do share the same page of stack.

Together with these optimized processes, the number of live processes that the system can manage is increased drastically, provided that most of the processes are of the optimized type.

4.4. Runtime Checks

4.4.1. Procedure Entry

The pseudo code in listing 4.1 and the assembler code in listing 4.2 show what the runtime checks at procedure entry look like. The checks are optimized to generate as less overhead as possible. Basically, only two subractions, two logical shifts and a few moves (mostly between registers) are necessary to find out if there's enough space left on the current page. If there is, no additional overhead is caused. The code in listings 4.1 and 4.2 are best viewed together with figure 4.3(b).

```
1 IF PrivilegeLevel # 0 THEN
2     IF ((ESP - spaceNeeded) DIV 4096) # (ESP DIV 4096) THEN
3         CallOverflowHandler
4     END
5 END
```

Listing 4.1: Runtime Check at Procedure Entry – Pseudo code

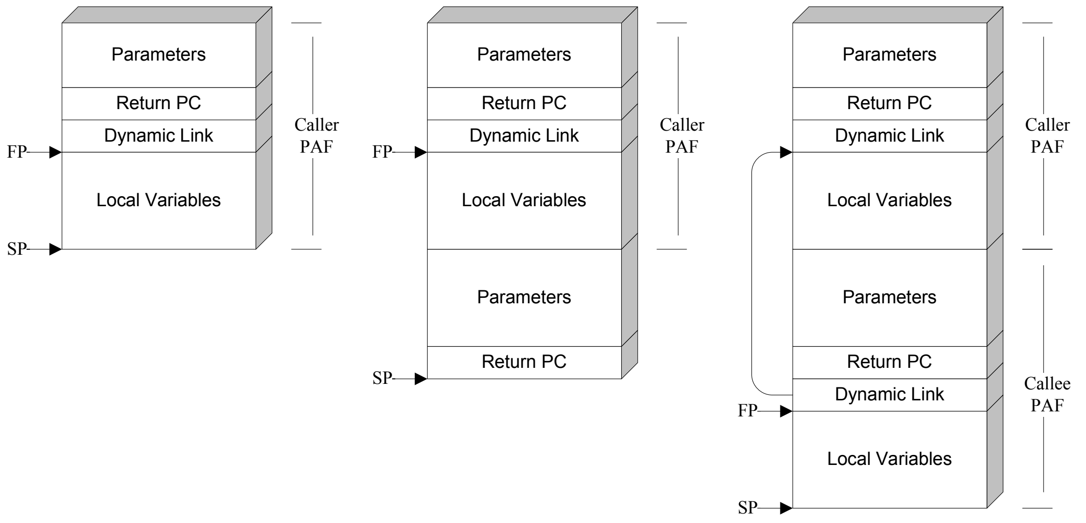
The check for a privilege level not equal to zero on line 1 of listing 4.1 is needed because Bluebottle has a separate kernel stack for each processor. Because these kernel stacks are statically sized – the system reserves 16 KB for each of them – there is no need to perform overflow and underflow handling if working on one of them. For more information see section 4.11.

```
        MOV     CX, CS
        AND     CL, 03H
        JZ      skip
        MOV     ECX, spaceNeeded
        MOV     EAX, ESP
        SUB     EAX, ECX
        SHR     EAX, 12
        MOV     EBX, ESP
        SHR     EBX, 12
        SUB     EBX, EAX
        JZ      skip
        INT     52      ; call Overflow Handler
skip:
        ; Procedure Prologue:
        PUSH    EBP
        MOV     EBP, ESP
```

Listing 4.2: Runtime Check at Procedure Entry – Assembler Code

Please note that the check for the privilege level does not use the register `AX`. This is a precaution such that the same runtime checks can be used for all modules. Especially `AosBoot` is relying upon the value in `EAX` which is put there by the bootloader and contains the offset of the boot table. So basically if `EAX` was used in the runtime checks, the system would not boot anymore.

The runtime checks at procedure entry can contain additional instructions to compute the dynamic size of the stack, if and only if one or more open arrays are passed by value. The calling convention (see [9], section 6.2.4) requires that for these parameters a local copy is allocated. Therefore – if somehow possible – it is usually a good idea to pass arrays by reference to increase performance.



(a) Stack before the call (b) Stack when the callee's runtime checks are executed (c) Stack when callee is running

Figure 4.3.: A new Procedure Activation Frame (PAF) is created

Figure 4.3 shows the different situations that occur when a procedure (the caller) calls another procedure (the callee): 4.3(a) shows the stack right before the caller pushes the arguments of for the callee. 4.3(b) shows the stack after the arguments have been pushed an the `CALL` instruction has be executed, and 4.3(c) shows the stack after the runtime checks have been executed and while the callee is running. In this figure, no overflow has happened.

4.4.2. Procedure Exit

The pseudo code in listing 4.3 and the assembler code in listing 4.4 show what the runtime checks at procedure exit look like. These checks too are optimized to generate as less overhead as possible. Here, only one addition, one logical AND and a few moves (mostly between registers) are necessary to find out if the procedure is a candidate for an underflow. If it isn't, no additional overhead is generated.

```
IF PrivilegeLevel # 0 THEN
    IF ((ESP + spaceForParams) MOD 4096) = 0 THEN
        CallUnderflowHandler
    END
END
```

Listing 4.3: Runtime Check at Procedure Exit – Pseudo code

As you can see from listing 4.4, these runtime checks are inserted after the caller's base pointer has been restored (`POP EBP`). This situation is exactly analogous to the situation shown in figure 4.3(b).

```
        ; Procedure Epilogue (1):
        MOV     ESP, EBP
        POP     EBP
        ; Runtime Checks
        MOV     CX, CS
        AND     CL, 03H
        JZ      skip
        MOV     ECX, spaceForParams
        MOV     EBX, ESP
        ADD     EBX, ECX
        AND     BX, 01000H
        JNZ     skip
        INT     53       ; call Underflow Handler
skip:
        ; Procedure Epilogue (2):
        RET     [paramsSize]
```

Listing 4.4: Runtime Check at Procedure Exit – Assembler Code

Please note that the registers EAX and EDX are not used within this runtime check. This is necessary because at the point the checks are executed, any return value has already been assigned to EAX or EDX:EAX respectively.

The different situations when a Procedure Activation Frame is destroyed are exactly the same as in figure 4.3, except that the order has of course changed.

4.5. Overflow Handling

As depicted in listing 4.2 the overflow handler is a kernel routine which is called through interrupt 52. The procedure registered to call on INT 52 is HandleStackOverflow in AosInterrupts. This procedure performes the following tasks.

1. Clear the Interrupt flag on the current processor (CLI).
2. Find the pointer to the process causing the overflow.
3. Load the Reflection of the Module containing the procedure causing the overflow.
4. Determine the procedure causing the overflow.
5. Calculate the total space needed at most for this procedure.
6. Allocate the appropriate number of pages and update statistics.
7. Update the stack structure of the process to contain the newly allocated area.
8. Initialize the topmost page of the new area with the header (see 4.5).
9. Copy the parameters for the procedure to the new stack area.
10. Resume the process on the new stack area.

Please note that setting the Interrupt flag when leaving the procedure is unnecessary because AosInterrupts.FieldInterrupt will clear the flag again when the overflow handler returned. However, it is not guaranteed that the Interrupt flag is cleared when the handler is called.

```
TYPE OverflowPageHeader = RECORD
        (* first 4088 bytes of the page are used as stack *)
        OldStackAdr:     LONGINT;
        OldStackPointer: LONGINT;
    END;
```

Listing 4.5: Definition of Stack Overflow Page Header

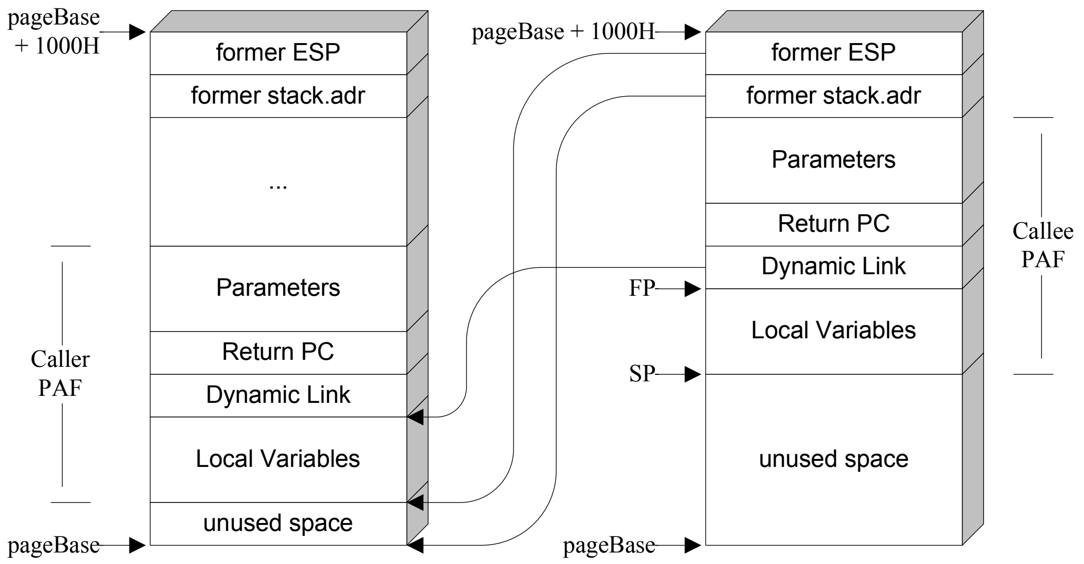


Figure 4.4.: Example of a Stack Overflow

4.6. Underflow Handling

The underflow handler is also a kernel routine. It is called through interrupt 53. The procedure registered to call on `INT 53` is `HandleStackUnderflow` in `AosInterrupts`. This procedure performes the following tasks.

1. Clear the Interrupt flag on the current processor (`CLI`).
2. Find out if this is a procedure which has caused a stack overflow earlier. Check if none of the following conditions evaluate to `TRUE`.
    a) The process' `EBP` register is zero (0).
    b) The page where `ESP` is on is not marked *Overflow*
    c) `OldStackAdr MOD 4096` (see listing 4.5) of the page where `ESP` is on is not equal to zero (0).
3. Find the pointer to the process causing the underflow.
4. Load the Reflection of the Module containing the procedure causing the underflow.
5. Calculate the total space needed at most for this procedure, including calls to other procedures.
6. Find out if there are orphaned pages at the end of the process' stack.

   a) If yes, remove the pages from the process' stack and insert them to the stack area pool.

7. Retrieve the values of `ESP` and `stack.adr` that were saved in the header when the overflow happened.
8. Free the corresponding number of pages and update statistics.
9. Resume the process on the former stack area.

4.7. Handling Optimized Processes

Because the stacks for optimized processes are always grouped on pages, they also fit with the new stack layout mentioned in 4.2. Figure 4.5 shows a diagram of how these pages are structured. Listing 4.6 shows a pseudo record definition for these pages.

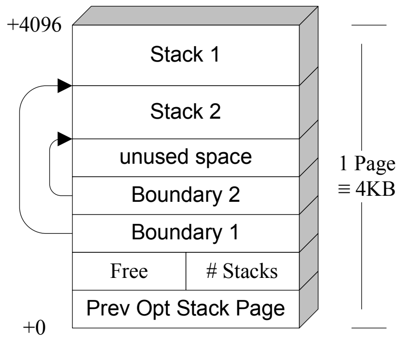


Figure 4.5.: Layout of an Optimized Stacks Page

The scheme from figure 4.5 and listing 4.6 does not allow to have static stacks which require more than 4084 bytes. However, according to the restrictions from section 4.3, it is impossible to have call stacks of a depth >2. Like this it is a very pathological case to have a stack of the size mentioned.

```
TYPE OptimizedStacksPage = RECORD
    PrevOptStackPage: LONGINT;
    NumberOfStacks: INTEGER;
    FreeSpace: INTEGER;
    BoundaryTable: ARRAY OF LONGINT;
    (* free space *)
    (* stacks *)
  END;
```

Listing 4.6: Pseudo Definition for Optimized Stacks Pages

Please note that the boundary table does not have a fixed number of elements. It does therefore grow towards the end of the page. On the other hand, the stacks grow from the end of the page towards its beginning.

Given the information in the header and in the boundary table of a page dedicated to optimized stacks, the system can easily find out which parts of the page belong to which processes. Listing 4.7 gives the algorithm to allocate a new optimized stack in pseudo code.

```
IF currentOptStacksPage.FreeSpace < maxSpaceNeeded + 4 THEN
  newOptStacksPage := NewPage();
  newOptStacksPage.FreeSpace := PageSize - HeaderSize;
  newOptStacksPage.NumberOfStacks := 0;
  newOptStacksPage.PrevOptStackPage := currentOptStacksPage;
  currentOptStacksPage := newOptStacksPage
END;

IF currentOptStacksPage.FreeSpace = PageSize - HeaderSize THEN
  currentOptStacksPage.BoundaryTable[0] := PageSize -
    maxSpaceNeeded;
ELSE
  i := IndexOfLastBoundaryTableEntry(currentOptStacksPage);
  currentOptStacksPage.BoundaryTable[i + 1] :=
    currentOptStacksPage.BoundaryTable[i] - maxSpaceNeeded;
END;

INC(currentOptStacksPage.NumberOfStacks);
DEC(currentOptStacksPage.FreeSpace, maxSpaceNeeded + 4);
```

Listing 4.7: Algorithm to allocate a new Optimized Stack

Please note that the algorithm in listing 4.7 does not try to fill holes between optimized stacks. This could lead to a huge fragmentation in some rare cases, when only one optimized stack is left on each optimized stack page. But this again is a very pathological case. It should however be possible to compact the stacks to minimize fragmentation. The meta data should provide all information necessary to relocate pointers on the stack.

After an optimized process has been finalized, its stack gets collected by the system. If it was the last stack on an optimized page, the page gets disposed too.

4.8. Integration in Memory Management

The complete implementation of the new stack layout does not have any impact on the rest of the memory management tasks. Instead, existing structures are used. Especially the distinction between pages for normal and optimized stacks was integrated in the page table entries for the MMU (see sections 3.6 and 3.7 of [4]). Bits 9 to 11 are available for system programmer's use. Table 4.1 gives the meanings of bits 9 to 11 of page table entries as described in section 3.6.7 of [4].

Bit	**Meaning**
9	1 indicates that the page is used as a stack page
10	1 indicates that this is the beginning of a stack overflow area
11	1 indicates that this is a page for optimized stacks

Table 4.1.: IA-32 Page Table Entry Bits 9 to 11

4.9. Symmetric Multi-Processors

There was a problem with the new stack management on Symmetric Multi-Processor (SMP) machines. Because the kernel now can contain implicit upcalls – these are the calls generated by the compiler to the overflow and underflow handlers in `AosInterrupts` – the locking order in the system can no longer be guaranteed. Having only one lock for the whole kernel would not help because of the broadcasts – for instance to flush the TLB – between processors. It could even lead to more deadlocks in that case. Figure 4.6 illustrates the problem. When the second processor arrives at (1) it tries to acquire the lock but it can't because the first processor (or another processor) already holds it while already handling the interrupt caused by the broadcast. P0 on the other hand (and all other processors except P1 too) are waiting for P1 in the front barrier in the broadcast handler and the system freezes.

If on the other hand we did use fine-granular locks (like in the old kernel) that would not work either because of lock order violations as mentioned previously. To avoid locking order violations, the kernel module locks for modules up to `AosInterrupts` were unified to one single lock. See section 4.11 for more information.

While investigating these deadlocks, also the freezes in the existing kernel were inspected. In fact also the old kernel did have potential deadlocks – these showed up reliably e.g. when compiling the whole system on a >2 processor system. However these deadlocks did not occur for kernel module locks but for object locks. The situation however was exactly the same like in figure 4.6. To resolve this issue, the spin lock acquisition in `AosLocks.AcquireObject` was adapted to temporarily allow interrupts again. With this change, the old kernel now seems to run stable with the maximum supported number processors.

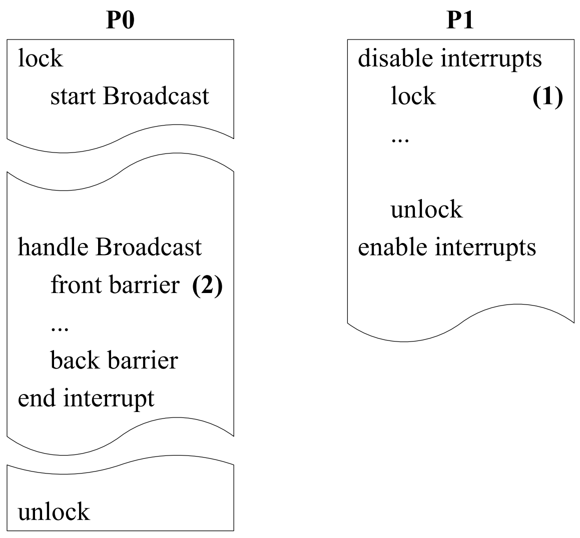


Figure 4.6.: Deadlocks on SMP machines

4.10. Performance

Several tests were ran with the new stack management enabled. This section discusses the results of these tests. Table 4.2 shows some figures about memory and time consumption. Please note that creating hundreds of thousands of processes is getting more and more slow. This is because the garbage collector is automatically ran after a certain amount of newly allocated blocks. In addition, creating one process basically involves creating three objects: the active object, the corresponding process and the corresponding finalizer node. Figure 4.7 shows the difference graphically (note that the scale on the x-axis is *not* linear).

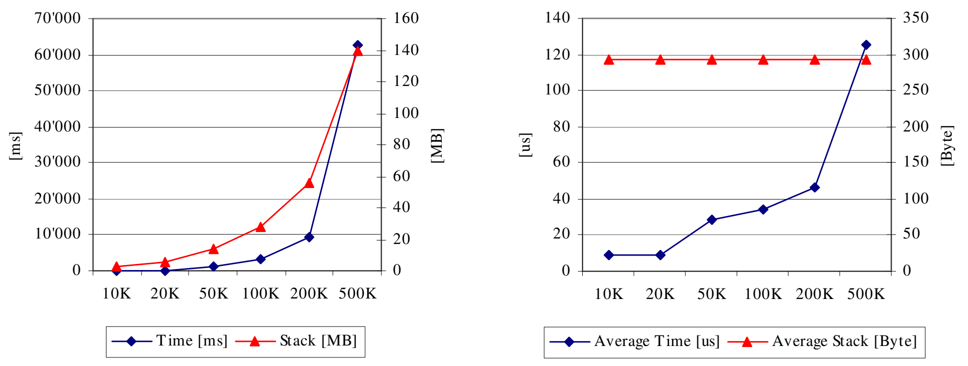
# Processes	Total Time	Total Stack	Total Heap	∅ Time	∅ Stack
10’000	89.8 ms	2.8 MB	10.1 MB	9.0 μs	293 Byte
20’000	178 ms	5.6 MB	20.1 MB	9.0 μs	293 Byte
50’000	1’410 ms	14 MB	50.4 MB	28 μs	293 Byte
100’000	3’400 ms	28 MB	101 MB	34 μs	293 Byte
200’000	9’300 ms	56 MB	201 MB	47 μs	293 Byte
500’000	62’700 ms	140 MB	504 MB	125 μs	293 Byte

Table 4.2.: Performance Tests for creating Optimized Active Objects

Now that the stacks are optimised one can see that all the objects of processes need more space on the heap than for their stacks. Especially for the kind of optimised active objects used for the measurements in table 4.2, the stacks in average are 293 bytes large whereas the processes and related objects (finalizer node and active object) in average need 1056 bytes (i.e. more than 1 KB). Figure 4.8 illustrates the different sizes of heap and stacks for the number of processes given in table 4.2.

Having such a huge amount of processes in the system does not really affect the performance of the scheduler, because the system keeps the processes which are ready to run in queues sorted by priority (as mentioned in section 2.2). Scheduling a new process can be done in $O(1)$, the same applies for preempting a process.

Regarding these facts – on a machine with 1 GB of physical memory – the maximum number of optimised processes of the above mentioned type is around 740'000, where the stacks take up around 200 MB while the heap is about 750 MB in size. The remaining ∼75 MB are used for the loaded modules, system processes and memory that's reserved for I/O.



(a) Total Time and Stack usage

(b) Average Time and Stack usage

Figure 4.7.: Time and Memory Chart

Please remember that the old kernel with the simple stack management did only support ∼15'000 processes at the same time. As such one can not compare the old system with the new one.

4.11. Known issues

Privilege Level awareness

As depicted in listings 4.1 and 4.3, the runtime checks always first determine whether the processor is in kernel mode (privilege level = 0) or in user mode (privilege level = 3). This is necessary because in privilege level 0, the processor is working on the corresponding kernel stack. When the system is started, the kernel stacks are initialized to 12 KB each (a further 4 KB are reserved to detect unhandled stack overflows on the kernel stacks). On top of this, with the current design of the kernel, it is not possible to handle overflows on the kernel stack for several reasons. Basically overflows on the

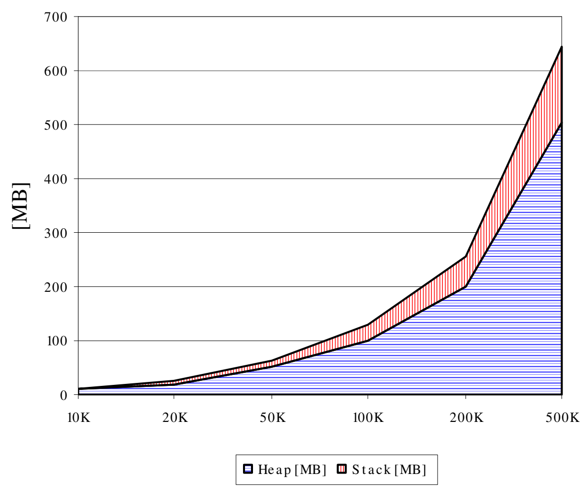


Figure 4.8.: Usage of heap and stack

kernel stack could happen in any of the modules before `AosInterrupts` which means that the overflow and underflow handler may not yet have been installed or ready at that time.

A solution could be to redesign the kernel such that all kernel calls are inter-privilege level calls, i.e. all code that belongs to the kernel is always executed in privilege level zero. In that case the kernel modules could be compiled without the runtime checks for overflow and underflow detection and one would probably not have problems with kernel locks. This solution was already applied to the problem specified in the next paragraph and works fine.

Saving the Registers

There is another problem that only occurs in `AosInterrupts`. The two procedures `FieldInterrupt` and `FieldIRQ` are used as targets for the vectors in the Interrupt Descriptor Table (IDT). It is essential that these two procedures first of all save the processor flags and registers. Adding runtime checks to these procedures would immediately overwrite some bits of the flags register and therefore disallow to properly resume the interrupted process. As mentioned in the previous paragraph, *not* adding runtime checks to `AosInterrupts` already solves this problem.

Register Spilling

In the current implementation of the Intel x86 code generation plug-in, register spilling is not supported. If it was supported, this would mean that the runtime checks would have to know how many registers can be spilled at most at any point in the procedure. This space would then also have to be taken into account when computing the space required for the procedure activation frame of a procedure. Generally, it would be the corresponding code generation plug-in's responsibility to provide this information. But with respect to the upcoming 64bit architectures – which usually have many more registers – register spilling becomes less important anyway.

Language Restrictions

For this stack sharing scheme to work it was also necessary to restrict the language. Until now it was possible to return open arrays from procedures. These procedures basically were overwriting their own stack frame with the resulting array. For the exact convention see [9], section 6.3.4 under *Returning Open Arrays.* This is no longer possible now because it would require that the stack frames of the caller and callee *must* be contiguous in any case. With the new stack management provided in this thesis, this cannot be guaranteed.

There's yet another problem with open or dynamic arrays. The compiler so far was able to correctly handle local open arrays like `arr` in the example from listing 4.8.

```
PROCEDURE SomeStringArithmetic*(len: LONGINT);
VAR arr: ARRAY len OF CHAR;
BEGIN
    (* do something with arr *)
END SomeStringArithmetic;
```

Listing 4.8: Example of a local open array

As one can see, arr is a local array whose length is computed at runtime using the parameters passed to the procedure. Evaluating the parameters to find the length of the local array (and therefore the space required for local variables) is not impossible, but would result in a rather time-consuming and complex runtime check. So for the sake of simplicity, this kind of local arrays has been forbidden too. It is questionable anyway if this kind of local arrays is allowed with respect to the language report [5] where it says that in `ARRAY <n> OF <type>`, *n* must be a constant.

Deadlock avoidance

According to section 3.12 of [4] the system does have to flush the TLBs of the processors whenever a stack page is unmapped. If now stack pages get unmapped by the underflow handler, this can lead to a deadlock as described in figure 4.6. To circumvent this problem, a pool for unused stack areas was introduced. The underflow handler puts unused pages into the pool whereas the overflow handler can take areas out of the pool. Also a new system-owned user process was introduced that periodically releases unused pages from the pool and then flushes the TLBs. Since this happens in user mode, there's no possibility to have deadlocks while flushing the TLBs anymore.

Another implementation issue comes with the locking order in the old kernel. The rule says that locks do have to be acquired top-down, i.e. higher module locks have to be acquired before lower modules locks. If there are no upcalls in the kernel this is usually no problem because the locking order is the same as the module order in the kernel. However with the new stack sharing scheme upcalls can happen implicitely through stack overflows and underflows. To resolve this problem, the locks below `AosInterrupts` were unified to one single lock.

Synchronization for optimized processes

To allow the usage of `AWAIT` statements in optimized processes, one would have to reserve a lot more space for optimized processes. Also it would be essential that the wait conditions would be of a very simple nature. Especially calls to procedures or methods should be forbidden. In that case the space needed to evaluate the wait condition would be known in advance and could be reserved by the overflow handler.

4.12. Other Stack Layouts considered

During the design phase of the stack sharing scheme introduced in section 4.2 also other stack sharing schemes were evaluated. Some of them are presented in this section. Please note that most of these approaches were used in Real-Time and Embedded Systems, where physical memory is usually very limited and an MMU not present at all.

4.12.1. Multitask Stack Sharing

Figure 4.9 shows a diagram of the Stack Sharing scheme introduced in [6]. The main idea here is that each process is still assigned a certain amount of stack space. Since the stacks grow top down there is usually a certain amount of free space at the lowest addresses of a process' stack. As long as this space is not used, it can be assigned – page wise – to other processes as some kind of overflow area. These overflow areas are spread amongst all the existing processes' stacks which in average automatically leads to balanced situations. The numbers in the pages at the bottom of each stack in figure 4.9 show how this could work: 1 is where the first overflow area is placed, 2 is where the second overflow area is placed and so on.

To realize this scheme it is required that the runtime information about each process is extended by an overflow pointer indicating where the last overflow area in the reserved space is located as well as of course information about the process' own overflow areas. The basic idea about the runtime checks used in this thesis' implementation (as presented in section 4.4) is taken from the scheme in [6]. This approach by itself however does not at all solve the problem of a fixed maximum number of processes in the system. The absolute maximum here – ignoring the fact that the heap also requires space – would be approximately 1 million processes.

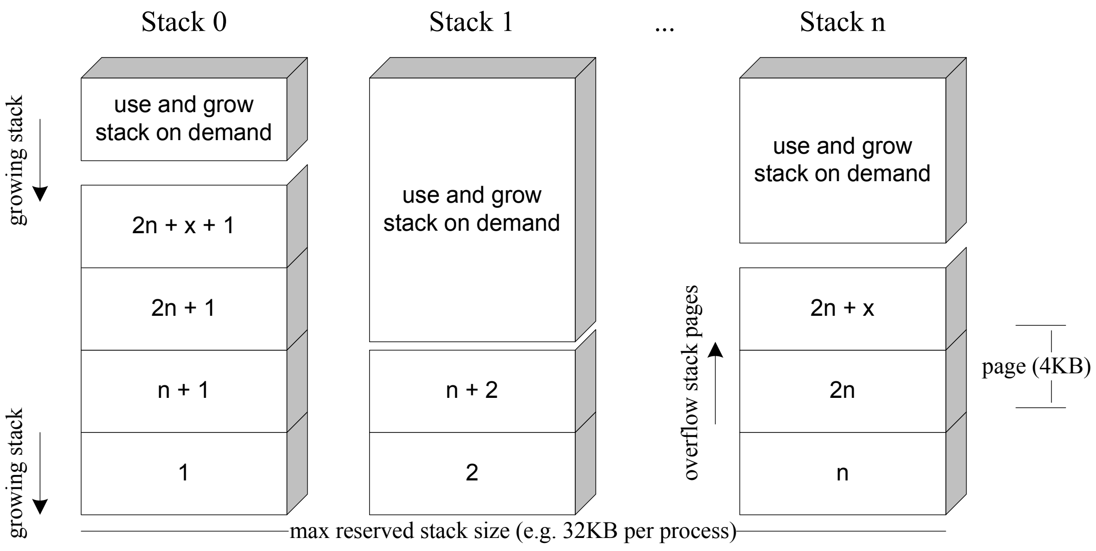

Stack 0
Stack 1
...
Stack n
use and grow stack on demand
use and grow stack on demand
use and grow stack on demand
2n + x + 1
2n + 1
n + 1
1
n + 2
2
2n + x
2n
n
growing stack
growing stack
overflow stack pages
page (4KB)
max reserved stack size (e.g. 32KB per process)

Figure 4.9.: Multitask Stack Sharing

4.12.2. A Hybrid Stack Sharing Scheme

Another mechanism evaluated is described in [7]. Here the idea is that an operating system would basically only need $O(n)$ stacks in memory, where n is the number of processors. Essentially several processes could share the same stack in memory because only one process can be running at a time. This however only applies to single processor systems. Whenever a task is switched out, the system can copy the whole stack from memory e.g. to a file or any other appropriate storage. If the task is to run again (when it is switched in) the system can copy the stack from the alternative storage to the correct location in memory again. Therefore it is possible to still have a fixed number of stacks (in memory) but a more or less unlimited number of processes using those stacks. However, the main disadvantage is that context switches become very expensive because on average there is two rather time-consuming copy operations – one to and one from the alternative storage. It should be noted that in the approach presented in [7] the approximative maximal stack size of the processes is known in advance.

Chapter 5.

The Parallel Compiler (PaCo)

To be able to generate the object files in the new format (as described in appendix A) the existing Parallel Compiler (see also chapter 5 of [9]) had to be changed. Since the compiler was built on a strictly modular basis, basically only the modules which deal with the object files had to be renewed. However, to be able to use the stack sharing scheme described in chapter 4 also other modules (e.g. the Intel x86 Code Generator Plug-in) had to be changed. For a complete list of changes see appendix C.

5.1. Object File Plug-in

The object file plug-in is responsible for generating the object file. It basically gets the symbol table and the code generated by the installed code generator plug-in and puts all the information in the correct format. It does also have to collect and prepare other important information for the runtime system, such as fix-ups in code and the exception table.

The object file plug-in uses the information loaded from the import plug-in (see section 5.2) for the verification section of the object file (see appendix A). All the sections of the object file are written in a more or less straight forward manner. This also means that parsing the object file can be done in one single pass.

Please note that there is still a capability to further optimize the output of the new object file plug-in. One could for example say that exported symbols are always listed first in the meta data section, such that the import plug-in for the Compiler could skip parsing symbols that can never be accessed from outside.

The object file plug-in also uses the new FingerPrint Module (PCFP) to finger print the symbols in the module. The main difference between the old finger printing mechanisms is described in section 5.3.

5.2. Import Plug-in

The import plug-in is responsible for loading the exported symbols from modules that are in the import list in the source code of the module to compile. It is based on the Reflection API described in chapter 3. After loading the meta data through the Reflection API the module generates the – public – symbols for the compiler's symbol table. There's also potential to optimize here: basically the import plug-in could skip the Reflection API and parse the meta data section (see section A.5) by itself. Together with the optimization mentioned in section 5.1 this could even lead to a compiler almost as fast as the old version of PaCo.

5.3. Finger Prints

With the old object file format it was essential that each and every symbol that was somehow exported was finger printed. This included fields of exported records and objects. With the new object file format, references are used rather than inlined structures (see [10]) and therefore it is completely sufficient to finger print the corresponding record only. As a consequence to this, the finger print *must* necessarily change whenever any part of the structure has changed. The base algorithms for the finger prints are still the same, however it is now only applied to global variables and constants, procedures and types but not to their respective local symbols. On top of this, the finger print is also applied to the module itself. This should make loading and linking modules a lot faster, because like this not all symbols have to be compared and verified. The loader and linker can now check if the finger print in the import section of one module is the same as the finger print in the export section of the other module. If they are the same, the verification was successful and the next module can be verified. With the old object files, every imported symbol had to be verified on its own, slowing down the load process depending on the number of symbols used.

5.4. Compiler and Language Flaws

During the implementation of the different parts of this thesis, some problems in the compiler or the Active Oberon language were detected. They are hereby discussed.

Nested Procedures

The language (according to [5]) and the compiler do not explicitely forbid that nested procedures are assigned to procedure variables. Especially code like in listing 5.1 is allowed.

```
 1 MODULE NestedProcTest;
 2
 3 VAR SomeProc: PROCEDURE(val: LONGINT);
 4
 5   PROCEDURE ProcA();
 6   VAR someLocalVar: LONGINT;
 7
 8     PROCEDURE NestedProc(val: LONGINT);
 9     BEGIN someLocalVar := val
10     END NestedProc;
11
12   BEGIN
13     someLocalVar := 1234H;
14     SomeProc := NestedProc;
15     NestedProc(5678H);
16     SomeProc(0DEADH)
17   END ProcA;
18
19 END NestedProcTest.
```

Listing 5.1: The nested procedures problem

Basically the assignment on line 14 should be prohibited as the signature of `NestedProc` and `PROCEDURE` are not the same: `NestedProc` requires the static link to the procedure activation frame of `ProcA` to be pushed as last (hidden) parameter. As such, the call on line 15 will succeed but the call on line 16 will fail. Therefore, assignments of the kind described above should be forbidden by the compiler.

Objects and Pointers to Records

The Active Oberon language does make an explicit difference between `OBJECT` and `POINTER TO RECORD`. The former is allowed to have methods and – at least according to the language report – types and constants whereas the latter is only allowed to have fields. Both however are represented exactly the same way in the compiler – and the object files – as `RECORD` which has an associated `POINTER TO`. On the one hand this is allright because both are a reference to a `RECORD` in the heap. On the other hand it is not, because they do not describe the same concept: whilst `OBJECT` is a container for data and code – remember, objects can even be active – `POINTER TO RECORD` can only contain data as suggested by its name.

Variable constants

As mentioned in section 4.11, the language report clearly says that the length of an array – if specified – must be a constant expression. In the example in listing 4.8 the parameter `len` is only constant with respect to the moment when the corresponding procedure activation frame is created. But basically `len` can be changed like a local *variable* and therefore the expression is not constant.

Object headers

Another problem that's also related to the runtime system comes with protected records. Objects are usually represented as protected records in the heap. The peculiarity with protected records is that they are prepended a header which is just a bit more than 32 bytes in size. This header is used for locking objects (therefore the name „protected records“). However the header is only needed if the object does contain any method having the `EXCLUSIVE` modifier.

So to save space, the compiler should mark objects which do not have any `EXCLUSIVE` modifier such that the runtime system can leave away the header for instances of these types.

Chapter 6.

Conclusion and Future Work

6.1. Conclusion

The thesis can be split into three tasks. The first task was to design a new object file format. This task has been accomplished. The second task was to update the compiler and the runtime system such that they are able to deal with the new object file format. Also part of the second subtask was the Reflection API introduced in chapter 3. This goal has been accomplished too. The third and most complex subtask was the design and implementation of the improved stack management. Regarding the fact that one can still use the system with the new stack management, this task can be considered accomplished too.

6.1.1. Object File Format

The new object file format stores detailed meta information about the modules. This information can automatically be parsed by the Reflection API. The new object file format was designed for fast verification, loading and relocation of modules at runtime. Together with the new object file format comes a new loader plug-in (`BbLoader`) and a new linker (`BbLinker` and `BbLinker0`) as well as new plug-ins for the Parallel Compiler: `PCBbOF` (Bbx Object File Generation) and `PCImport` (Bbx Object File Imports).

6.1.2. Reflection API

The Reflection API provides easy access to the detailed meta information in the new object files. It is used in the new import plug-in for PaCo as well as the new stack trace plug-in (`AosStackTrace`) and the stack overflow and underflow handlers in `AosInterrupts`. The Reflection API would for instance also allow to generate a complete object graph from a heap snapshot.

6.1.3. Improved Stack Management

The new stack management scheme introduced to Bluebottle as part of this thesis provides the following advantages. First, the space wasted for unused stacks is reduced a lot. Second, the maximum number of processes the system can manage is revoked and on top of this, it is no longer determined by the system but much more by its applications and the amount of physical memory available. This however implies that the system is now a bit slower (again depending on the applications) due to the additional stack management that has to be done at runtime.

6.2. Future Work

One of the goals of this thesis was to evaluate a new way of managing the processes' stacks. The results basically show that it is possible in principle to achive to have hundreds of thousands or even millions of processes in a system. It is also shown what an actual simple implementation might look like. A next step would be to (re-)design the kernel to support millions of threads from the beginning and still keep a clean design. This means that one would also have to find a proper way to still have fine-granular locks within the kernel. Basically it should also be possible that the stacks and the heap are no longer separated from each other but that they are – in a way – interleaved.

Acknowledgements

I want to thank several people who have contributed to the success of this thesis. First of all Daniel Keller for supervising me. He gave me the hints I needed at the right time. Luc Bläser for assisting me with several severe problems I encountered. Prof. Dr. J. Gutknecht for being my mentor and having assigned this interesting thesis to me. Ulrike Glavitsch Eggler for her patience while building the new crazy fresh releases. Michael Szediwy for his assistance with indexers and some other issues. Patrick Reali for his help with some compiler issues. Thomas Frey for his quick responses while still being in China. Many thanks also to Christina and Kevin Blanchard, Lutz Hartenfels and Christian Reichlin for proof-reading this report.

Appendix A.

New Object File Format

The *ObjectFile* consists of four main sections:

1. the *VerSection* (Verification Section) for checking the consistency and compatibility of imported symbols
2. the *RelocationSection* dealing with fix-ups
3. the *RuntimeSection* containing the data for the runtime system (constants, code, type descriptors)
4. the *MetaDataSection* providing detailed information about every symbol in the module.

The loader and linker do only have to analyse the sections in the *LoaderData* (i.e. the verification section, the relocation section and the runtime section). The reflection uses only the *MetaDataSection*. The compiler can check imported symbols using the reflection only. Figure A.1 shows a scheme of the new object file format. The *Header* is described in the following section.

```
ObjectFile        = Header LoaderData MetaDataSection.
LoaderData        = VerSection RelocationSection
                    RuntimeSection.
```

Listing A.1: Object File – EBNF

A.1. Header

```
Header            = OFTag OFVersion OFTarget relocationOffset
                    runtimeOffset metaDataOffset moduleName.
```

Listing A.2: Header – EBNF

The *relocationOffset*, *runtimeOffset* and *metaDataOffset* can be used to directly read and parse the specific section. *OFTarget* specifies the target architecture of the object file.

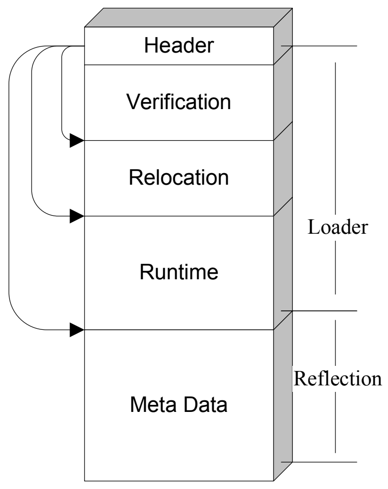


Figure A.1.: Scheme of the new Object Files

A.2. Verification Section

```
VerSection          = nofImports nofExports exportsOffset
                      Imports Exports.
```

Listing A.3: Verification Section – EBNF

The *Imports* and *Exports* sub-sections provide all information needed to check consistency and compatibility between two modules (the importing and the exporting module).

A.2.1. Imports subsection

```
Imports             = {ModuleImport}.
ModuleImport        = moduleName fingerPrint nofSymbols
                      symbolsLength {SymbolImport}.
SymbolImport        = symbolFlag symbolName fingerPrint.
```

Listing A.4: Imports subsection – EBNF

The *fingerPrint* of the module can be used to skip further fingerprint checking, iff they are the same for both modules (the importing and the exporting module), so it is an optimization to speed up the loading of rarely changed modules. *symbolsLength* can be used to skip the symbols for that module and proceed with the next imported module.

The *SymbolImport* rule provides, for each imported symbol (types, definitions, procedures, variables, constants), fingerprint information that can be used to check consistency and compatibility of the two modules, if necessary.

symbolFlag	**Meaning**
01X	Type
02X	Definition
04X	Procedure
08X	Variable
10X	Constant

Table A.1.: Values for *symbolFlag*

A.2.2. Exports subsection

```
Exports             = fingerPrint {Export}
Export              = symbolFlag symbolName fingerPrint
                      symbolOffset.
```

Listing A.5: Exports subsection – EBNF

Exports is the counterpiece to *Imports*. Therefore, it must provide the *fingerPrint* of the module as well as fingerprint information for each exported symbol.

For type descriptors, *symbolOffset* is the offset – in the constant section – of the pointer to the type descriptor. It is therefore added to the static base address of the module. For interface descriptors, *symbolOffset* is the offset of the pointer to the interface descriptor, again relative to the module's static base address. For procedures, *symbolOffset* is the offset of the procedure in the code section and is therefore to be added to the code base address of the module. For global variables, *symbolOffset* is the offset of the variable in the module's data section. It must therefore be added to its static base address. The data section is stored in the area below the static base address which basically means that *symbolOffset* values for global variables are always negative. For global constants *symbolOffset* is the offset of the constant in the constant section starting at the module's static base address.

A.3. Relocation Section

```
RelocationSection = ImportFixups SysCallFixups LocalFixups.
```

Listing A.6: Relocation Section – EBNF

The *RelocationSection* handles all fix-ups within the module's code. It provides the loader with the fix-up information for local and imported symbols as well as system calls and case tables.

A.3.1. Fix-ups for imported Symbols

```
ImportFixups      = nofImportFixups {ImportFixup}.
ImportFixup       = module nofSBFixups nofCBFixups nofTDFixups
                    {SBFixup} {CBFixup} {TDFixup}.
SBFixup           = symbolName nofOffsets {offset}.
CBFixup           = procName nofOffsets {offset}.
TDFixup           = typeDefName offset.
```

Listing A.7: Fix-ups for imported Symbols – EBNF

symbolName is the name of an imported constant or (global) variable. *procName* is the name of an imported procedure. For both – *StaticBaseFixup*s and *CodeBaseFixup*s – the offset is relative to the module's code base. They indicate where in the code the symbol or procedure is used and therefore which locations need to be updated (relocated) with the current address of the symbol.

TDFixup is used for placing the hidden pointer to the imported type's or definition's descriptor into the importing module's constant section, i.e. the *offset* is relative to the module's static base address.

A.3.2. Fix-ups for System Calls

```
SysCallFixups     = nofSysCallFixups {SysCallFixup}.
SysCallFixup      = syscallType nofOffsets {offset}.
```

Listing A.8: Fix-ups for System Calls – EBNF

The *syscallType*'s values are shown in table A.2. Kernel Calls are used to allocate new structures on the heap and to start, synchronize and signal processes. They are also used to register and lookup interfaces and interface method tables.

The last two Kernel Calls require that `AosInterfaces` be loaded and initialized *before* the module containing this kernel call is loaded. The compiler ensures this by adding `AosInterfaces` to the module's import list and calls to initialize and register interfaces at the beginning of the module's body.

Id		
dec	**hex**	**Kernel Call to**
253	FD	AosHeap.NewRec – create a new record or object
252	FC	AosHeap.NewSys – create a new system block
251	FB	AosHeap.NewArr – create a new array
250	FA	AosActive.CreateProcess – create a new process
249	F9	AosActive.Await – await a condition for synchronization
247	F7	AosActive.Lock – acqure a lock on an object
246	F6	AosActive.Unlock – release a lock on an object
245	F5	AosInterfaces.Register – register an interface implementation
244	F4	AosInterfaces.Lookup – lookup an interface

Table A.2.: Kernel Calls

A.3.3. Fix-ups for local Symbols

```
LocalFixups        = nofLocalSBFixups nofLocalCBFixups
                     nofCTFixups {SBLocalFixup} {CBLocalFixup}
                     {CTFixup}.
SBLocalFixup       = offset.
CBLocalFixup       = offset.
CTFixup            = offset.
```

Listing A.9: Fix-ups for local Symbols – EBNF

In a *SBLocalFixup*, the module's static base address has to be added to the value at the location found at *offset* in the code. In a *CBLocalFixup*, the module's code base address has to be added to the value at the location found at *offset* in the code. That is, for these two cases the *offset* is relative to the module's code base. In a *CTFixup*, the module's code base address has to be added to the value at the location found at *offset* in the constants section, that is *offset* is relative to the module's static base in this case.

A.4. Runtime Section

```
RuntimeSection     = constsLength dataLength codeLength
                     exTableLength nofTypeDescriptors
                     nofDefinitionImpls constants code
                     ExceptionTable {Descriptor}
                     {DefinitionImpl}.
```

Listing A.10: Runtime Section – EBNF

The runtime section contains the information needed to load the module into the runtime system. This includes the code, the constants, the exception table as well as the type descriptors. *constants* and *code* are byte streams. Their lengths are indicated by *constsLength* and *codeLength* respectively.

A.4.1. Type Descriptors

```
Descriptor         = ObjectDescriptor | RecordDescriptor |
                     DefDescriptor.
ObjectDescriptor   = 01X StructDescriptor MethodTable.
RecordDescriptor   = 02X StructDescriptor.
DefDescriptor      = 04X name ddOffset nofMethods nofNewMethods
                     SuperTypeTable.
StructDescriptor   = name tdOffset typeSize nofPtrOffsets
                     {pointerOffset} SuperTypeTable.
SuperTypeTable     = nofSuperTypes {offset}.
MethodTable        = nofMethodsTotal nofNewMethods {methodIndex
                     methodOffset}.
```

Listing A.11: Fix-ups for local Symbols – EBNF

The *tdOffset* in the *StructDescriptor* is the offset of the pointer to the type descriptor in the module's constant section. The *offset*s in the SuperTypeTable are the offsets of the pointer to the super type's type descriptor and is also relative to the module's constant section. Please note that also external (i.e. imported) types will have a pointer to their type descriptor in the local – i.e. importing – module's constant section. The *DefDescriptor* is analogous to the object and record descriptors, except that less information is required.

A.4.2. Exception Table

```
ExceptionTable     = {pcFrom pcTo pcExHandler}.
```

Listing A.12: Exception Table – EBNF

The *ExceptionTable* is a series of offset triplets which all point to the module's code section. Exceptions that occur anywhere between *pcFrom* and *pcTo* are handled by the code starting at *pcExHandler*. These entries need not be sorted as the loader and linker will do this. The Exception Table layout is taken from [8].

A.5. Metadata Section

```
MetaDataSection   = fingerPrint nofTypes nofDefs nofProcs
                    nofVars nofConsts {TypeDesc} {Definition}
                    {Proc} {Var} {Const}.
```

Listing A.13: Metadata Section – EBNF

The *MetaDataSection* provides detailed information about all symbols in the module. The loader and linker do not have to examine the *MetaDataSection* but they will store the raw byte stream in the loaded module's runtime structure. Basically only the Reflection API will be reading the meta data section. Please note that the *MetaDataSection* of a module contains the same fingerprints as the *Exports* subsection in the verification section of the same object file. This redundancy was added to keep the different sections in the object file independent of each other. It also allows that the compiler can retrieve all information about imported symbols through the Reflection API.

A.5.1. Types

```
TypeDesc          = ObjectType | PointerType | ArrayType |
                    RecordType | DelegateType | AliasType.
ObjectType        = 01X objName typeModifiers fingerPrint
                    tdOffset typeSize BaseType nofFields
                    nofMethods nofImplements {Field} {Method}
                    {ImplementedDef}.
PointerType       = 02X pointerName typeModifiers fingerPrint
                    BaseType.
ArrayType         = 04X arrayName typeModifiers arrayLength
                    fingerPrint Type.
RecordType        = 08X recName typeModifiers fingerPrint
                    tdOffset typeSize BaseType nofFields
                    {Field}.
DelegateType      = 10X delegateName typeModifiers fingerPrint
                    Type nofParams {Param}.
AliasType         = 80X aliasName typeModifiers fingerPrint
                    Type.
Field             = fieldName fieldModifiers Type
                    instanceOffset.
Method            = methodModifiers MethProcSignature
                    MethProcInfo Locals [InlinedCode].
ImplementedDef    = moduleName "." definitionName.
MethProcSignature = methodProcName Type nofParams {Param}.
MethProcInfo      = offsetStart offsetEnd.
```

```
Locals             = nofLocalVars nofLocalConsts nofTypes
                     nofNestedProcs {LocalVar} {LocalConst}
                     {TypeDesc} {NestedProc}.
BaseType           = NoType | (01X TypeQualifier) | (02X
                     TypeDesc).
NoType             = 00X.
Type               = BaseType | (04X BasicType).
TypeQualifier      = moduleName "." typeName {"." typeName}.
LocalVar           = varName Type sbOffset.
LocalConst         = constName Type sbOffset.
Param              = paramName paramModifiers Type offset.
```

Listing A.14: Metadata: Type descriptions – EBNF

The *AliasType* lets one define and export aliases for types. The compiler for instance uses this information when an imported alias on a type is used and therefore has to be resolved.

In the *BaseType* rule, the *TypeDesc* can be used to describe anonymous records. The *Type* in the *ArrayType* rule is used to describe the type of the elements of the array. It can also be an anonymous type descriptor.

For *DelegateType* entries, the *Type* field specifies the return type of the delegate.

A.5.2. Basic Types

```
BasicType          = Char | IntNum | FloatNum | Boolean | Set |
                     Ptr | SysByte.
Char               = 01X 01X.
IntNum             = ShortInt | Integer | LongInt | HugeInt.
ShortInt           = 02X 01X.
Integer            = 02X 02X.
LongInt            = 02X 04X.
HugeInt            = 02X 08X.
FloatNum           = Real | LongReal.
Real               = 04X 04X.
LongReal           = 04X 08X.
Boolean            = 08X 01X.
Set                = 10X 04X.
Ptr                = 20X 04X.
SysByte            = 80X 01X.
```

Listing A.15: Metadata: Basic types – EBNF

The *BasicType* describes all basic language types. The encoding scheme used is shown in listing A.16.

```
ABasicType          = basicTypeClass basicTypeLength.
```

Listing A.16: Metadata: Basic types – EBNF

basicTypeClass can be any value from table A.3. *basicTypeLength* is the length of the type's data and should usually be one of 01X, 02X, 04X or 08X.

Value	Basic Type
01X	character data
02X	integral numbers
04X	floating point numbers
08X	boolean data
10X	sets
20X	`PTR` and `ANY`
40X	types in `SYSTEM` (e.g. `SYSTEM.BYTE`)

Table A.3.: Values for *basicTypeClass*

Please note that `PTR` and `ANY` both get the value `20X 04X`. This is because the compiler does not draw a distinction between these two types.

A.5.3. Definitions

```
Definition          = defName fingerPrint ddOffset RefinedDef
                      nofMethods {MethProcSignature}.
RefinedDef          = moduleName defName.
```

Listing A.17: Metadata: Definitions – EBNF

Definition describes an interface (keyword `DEFINITION` in the language). It uses the same rule for the signature of its methods as the object types' methods and global procedures.

A.5.4. Procedures, Nested Procedures

```
Proc                = fingerPrint procedureModifiers
                      MethProcSignature MethProcInfo Locals
                      [InlinedCode].
NestedProc          = MethProcSignature MethProcInfo level
                      Locals.
```

Listing A.18: Metadata: Procedures & Nested Procedures – EBNF

A.5.5. Inlined Procedures

Because inlined procedures can also be exported, it is essential that the code of the procedure can be stored in the object file. In the old object file format, the inlined code was stored in the symbol file. Since inlined code is most likely only used by the compiler (more precisely, the loader amd linker do not need this information), the code is stored in the only section that needs to be accessed by the compiler, i.e. the meta data section. Here's what the inlined code rule looks like.

```
InlinedCode        = codeLength {codeByte}^codeLength.
```

Listing A.19: Metadata: Inlined Procedures – EBNF

The reflection detects if inlined code is present by evaluating the modifiers of the procedure. If the procedure is marked both public and inline, the *InlinedCode* rule is used and the inlined code is parsed. Storing inlined code in the object file also justifies the use of the architecture field (*OFTarget*) in the header. Methods can never be inlined because they can in principle be overridden in a subtype.

A.5.6. Variables

```
Var                = varName fingerPrint varModifiers Type
                     offset.
```

Listing A.20: Metadata: Variables – EBNF

The *Type* is a reference to the static type of the variable. *offset* is relative to the module's static base and points to the data section. It is always negative.

A.5.7. Constants

```
Const              = constName fingerPrint constModifiers Type
                     offset.
```

Listing A.21: Metadata: Constants – EBNF

Type is a reference to the type of the constant. This is usually one of `CHAR`, `INTEGER`, `LONGINT`, `HUGEINT`, `REAL`, `LONGREAL`, `BOOLEAN` or `ARRAY n OF CHAR`. *offset* is relative to the module's static base and points to the location in the constants section where the constant's value is stored. The value -1 (0FFFFFFFFH) can be used to indicate that the constant's value is not stored in the constant section.

Appendix B.

Language Extensions

In Active Oberon it is possible to have Assembler Procedures that can be inlined. For these procedures, the space needed on the stack cannot be exactly computed by the compiler for several reasons. Instead the programmer now has to give a hint to the compiler indicating how much stack will be needed at most. The example in listing B.1 should illustrate how to use this extension.

```
PROCEDURE -GetFlags*(): SET
CODE {STACK(4, 0)} {SYSTEM.i386}
  PUSHFD
  POP EAX
END GetFlags;
```

Listing B.1: Specifying stack size for assembler procedures

The first argument to STACK indicates how much the stack grows at most when the code of this procedure is being executed (with any input and execution scenario). The second argument is optional and indicates how big the stack will be after the procedure's code was completely executed, relative to the size when entering the procedure. This value should only be used for inlined assembler procedures, because all other procedures *must* clean up their stack before they terminate. An example of how to use this second argument is given in listing B.2. Please note that the second argument can be positive or negative.

```
PROCEDURE -PUSHFD();
CODE {STACK(4, 4)} {SYSTEM.i386}
  PUSHFD
END PUSHFD;

PROCEDURE -POPFD();
CODE {STACK(0, -4)} {SYSTEM.i386}
  POPFD
END POPFD;

PROCEDURE FlagSafeProc();
```

```
BEGIN
  PUSHFD;
  (* carry out some flags affecting operations *)
  POPFD;
END FlagSafeProc;
```

Listing B.2: Stacksize for inline Assembler Procedures

It is the developers responsibility to provide accurate figures for the stack space. The information about the stack does not go together with the information about the instruction set (e.g. `SYSTEM.i386`) because that part is not parsed by the Active Oberon parser but by the corresponding Assembler plug-in's own parser. This means that all the existing assembler plug-ins would have had to be changed.

Appendix C.

List of Changes

This chapter contains a list of all the changes that were applied to existing modules. The changes are grouped by the modules and sorted alphabetically.

AosActive Active Oberon Runtime Support
Support for new stack management, new active object to complete the startup process, modified pre-emption conditions.

AosConsole Boot Console
Installed new loader plug-in for extension „.Bbx“.

AosDecoder Object file decoder
Support for new object files, use Reflection API instead of own symbol file parser, decode instructions from binary file.

AosHeap Heap management & Garbage Collector
New procedure to determine if NilGC is still installed.

AosInterrupts Low-level interrupt handling
Stack overflow and underflow handling, stack area pool, new stack management.

AosKernel Kernel interface
New periodic stack area pool cleaner process.

AosLocks Fine-grained kernel locks
Support for re-entrant locks, deadlock avoidance for SMP machines.

AosMemory Virtual address space management
Support for new stack management, new flags for page table entries.

AosModules Module and Type management
Support for new object file format.

AosObjectMarker Mark objects referenced on the stacks
New module to find heap references from the stack exactly and mark them for the garbage collector.

AosProf Aos statistical profiler
Use of Reflection API for symbol lookup.

AosReflection Aos Reflection API New Reflection API module.

AosStackTrace Aos stacktrace plug-in.
New stack trace plug-in based on Reflection API.

AosTrap Trap handling and symbolic debugging
Support for new object files, new stack trace plug-in mechanism, removal of stack extension via page fault.

BbLoader Bluebottle module loader plug-in
New loader plug-in for new object files.

BbLinker Bluebottle boot linker
New boot linker module based on new object files.

BbLinker0 Bluebottle boot linker helper module
New helper module (heap and type management) for the new boot linker.

PC0 PaCo main module
Support for new object file format, new compiler flags */Z* (force new object file format) and */R* (do not insert runtime checks).

PCB PaCo semantic checker
Propagate maximal stack sizes.

PCBbOF PaCo Bbx object file generator
New object file generator plug-in.

PCBT PaCo backend common structures
Support for maximal stack sizes.

PCC PaCo intermediate code generation
Support for new object file format and maximal stack sizes.

PCFP PaCo finger printing
New fingerprint computation module.

PCG386 PaCo Intel x86 code generator
Add runtime checks at procedure entry and exit.

PCImport PaCo import module
New import plug-in for new object files.

PCLIR PaCo intermediate code representation
Support for maximal stack sizes.

PCM PaCo input and output module
Support for new object file format.

PCP PaCo active oberon parser
Support for language extensions and language restrictions (see 4.11).

PCO PaCo Intel x86 code patterns
Support for >64 KB code sections, dynamic size for code buffer.

PCS PaCo active oberon scanner
Support for language extensions.

PCT PaCo symbol table
Support for maximal stack sizes.

PCV PaCo symbol allocation
Support for maximal stack size computation.

ReflectionIO Reflection API I/O utility module
New module for loading the reflection from object files from disk.

Appendix D.

Task Description

Improved Stackmanagement in the Active Object Kernel

This project is about the optimisation of the stack management in a Multi-Threading Operating System, more precisely in Aos (the Active Object System). Presently each Thread is assigned an area of 128 KB in the virtual memory, where at least one physical page is mapped. This leads to an excessive limitation of the maximum number of Threads that can be active at the same time as well as an enormous fragmentation of the physical stack memory. The main goal of the project is to find a more sophisticated runtime architecture in the Aos Kernel which removes the mentioned deficiencies and which make the operating system more resource efficient with respect to memory, without affecting the efficiency. In particular, a new category of simple Threads shall be defined and optimised, such that these Threads do not waste any memory on the stack. Also to evaluate are possibilities of Stack-Sharing. As a side effect today's Objectfiles of the Aos system shall be enhanced with a complete meta data structure. Based on this meta data shall be designed and implemented a Reflection API.

The entire thesis shall be based on unobscured, completely transparent and reliable concepts. The implementation shall be done careful and as error-free as possible. At least the meta data enhancement and the Reflection API shall finally be entered in the Aos Release.

Appendix E.

Listings

Appendix F.

List of Figures

Appendix G.

List of Tables

Appendix H.

Bibliography

[1] Muller, P.J. *The Active Object System – Design and Multiprocessor Implementation*, Diss. ETH No. 14755, ETH Zürich, 2002.

[2] Intel®. *IA-32 Intel®Architecture Software Developer's Manual, Volume 1: Basic Architecture*, September 2005 http://www.intel.com

[3] Hoare, C.A.R. *Monitors: an operating system structuring concept*, Commun. ACM 17(10): 549-557 (1974)

[4] Intel®. *IA-32 Intel®Architecture Software Developer's Manual, Volume 3: System Programming Guide*, September 2005 http://www.intel.com

[5] Reali, P.R.C. *Active Oberon Language Report*, 2004, http://www.oberon.ethz.ch

[6] Middha, B., Simpson M., Barua R. *MTSS: Multi Task Stack Sharing for Embedded Systems*, Compilers, Architectures and Synthesis of Embedded Systems (CASES) 2005, September 2005.

[7] Wong, K.-F., Dagevill, B. *Supporting Thousands of Threads Using a Hybrid Stack Sharing Scheme*, ACM 1994.

[8] Szediwy, M. *Aos Thread Finalization & Termination*, Diploma Thesis, Department of Computer Science, ETH Zürich, February 2005.

[9] Reali, P.R.C. *Using Oberon's Active Objects for Language Interoperability and Compilation*, Diss. ETH No. 15022, ETH Zürich, 2003.

[10] Reali, P.R.C. *Symbol and Object File Format*, 2001, http://www.oberon.ethz.ch