Software Engineering Workshop (SEW-30) manuscript No.
(will be inserted by the editor)

Benoit Sonntag

Hardware Memory Segmentation for New Software Model

24-28 April 2006

Abstract This article is the conclusions of a study on the implementation of a new oriented object operating system. Indeed, starting from the analysis of our needs in communication for the development for our object mechanism, various problems appeared. In this study, we reveal that the poor *flexibility* in memory management is the *reflection* of the lack of segmentation usage. Nowadays, some processors offer advanced mecanisms of memory management through segmentation. Unfortunately, they are unusable some context as ANSI C programming. At first sight, the implementation of an operating system using such a mecanism would need prior rewriting of the stack in order to take account for memory allocation indexes. This article brings a simple and effective solution to allow processor segmentation. Moreover, this solution does not require any massive modification of the stack.

Keywords Segmentation · Software Model Architecture · virtual memory · stack · Object Operating System · inter-process communication · shared memory

1 Introduction

Some recent processors offer elaborate mechanisms of segmentation comparable with those of *Multics* [1]. The essential reason of the quasi non-existence of these mechanisms in many operating systems is caused by the stacks. However this mechanism can bring up a better management of memory and its protection. Moreover, we have known for long (cf. Multics, 1972) the possibilities offered inter-process communication, and the help it brings for the implementation of shared memory. Without losing any global consideration, our study is done on the implementation of the segmentation of a highly widespread processor : the family of processors INTEL 80386 and its higher versions. We benefit from qualities of the segmentation for the implementation of a new operating system, *Isaac*, our subject of research. The purpose of the project *Isaac* is to study and integrate concepts objects at the very heart of the operating system. However, the solution that is brought here for the use of segmentation in a C program remains valid for another operating system. Through its simplicity, the solution seems quite compelling to us and very much unique.

We begin in section 1.1 by reviewing certain elements of Unix's memory management. Then, section 1.2 presents our project of operating system that integrates our use of the processor segmentation. Section 1.3 details the problem of indexing memory in high-level languages. Section 2 presents the method we followed in section 3 to measure effectiveness of our solution. Related work is presented in section 4, thus concluding in section 5.

1.1 Review of Linux's memory management on a segmented architecture

In a UNIX system, a process occupies most of the virtually address-space. This does not facilitate good communication and the management of memory sharing between several processes [5] or distributed systems [4] (see fig. 1). Indeed, the present implementation of a C program in the memory of a segmented architecture does not use the segmentation of the processor. Precisely, it uses the same segments for the stack as for the data. It is clear that this use of the processor goes against the expectations and objectives of the manufacturer. This layout does not make it possible to detect efficiently the overflow of the stack in the stack. In *Linux*, the current solution consists in using an unusually large segment to prevent overlapping within the size limits imposed on a process by the system. Each process is therefore forced to use the same virtual slots of memory, thus requesting the abusive use of the MMU table. Indeed, each process has its own MMU table, and in this way its own virtual

B. Sonntag
INRIA-Lorraine / LORIA, 615 Rue du Jardin Botanique,
BP 101, 54602 VILLERS-LES-NANCY Cedex, FRANCE
E-mail: bsonntag@loria.fr

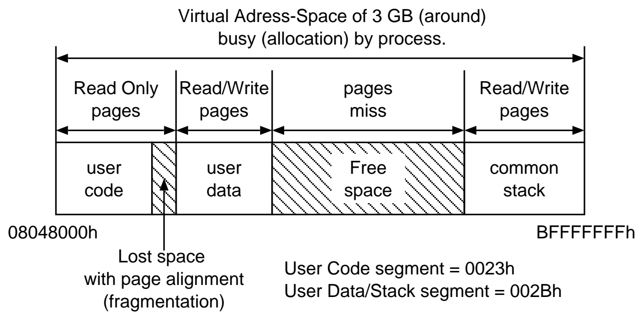

Fig. 1 Vision of the addressable space of a process in *Linux/Intel*.

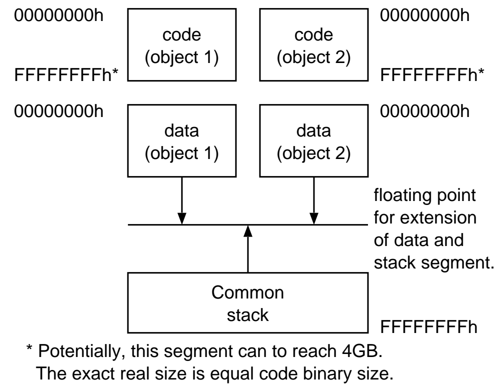

Fig. 2 Segmented vision of 2 objects *Isaac*: the two objects share the same stack of execution.

linear space of 4Go. The size of a MMU table can reach up to 4Mo of read-write memory.

In addition, we have noticed that code writing protection in *Linux* on a segmented architecture of the *INTEL 386* type is not made at the level of the code segment as *INTEL* [3] suggests : the code segment is an alias on the data segment/stack. To this extent nothing prevents at the level of segmentation from using the data segment to write in the code (see figure 1). Nevertheless, a protection in writing exists at the level of the memorized pages containing the code. This last point implies the code alignment on the size of a page. On average, half of the last allocated page is free and consequently is lost as a result of internal fragmentation. If there is n segments in memory and if the size of the pages is of p bytes, fragmentation makes us loose $np/2$ bytes. If we take in consideration that the code segment is not extensible, we could imagine a system of allocation which takes it into account and concatenates to the nearest byte the whole of the non-extensible segments. Inter-process protection would be maintained by segmentation and internal fragmentation would then disappear.

1.2 Usage of the segmentation of the processor in the Isaac operating system

Our problem of segmentation was revealed to us within the context of the implementation of a new operating system, *Isaac* [6], based on the concept of object sharing. In this section, we are going to introduce the central problematic of this project. As in the Mach [10] system, we stress the need for division of memory zones between various light processes. For example, in the traditional problem of producer/consumer, one can consider that the producer and the consumer are different processes which share the same data buffer. Our strategy is to extend this division in a finer cooperation between various processes of small size. We materialize this cooperation within our system by the use of dynamic, homogeneous and altogether fast objects, which function directly on the material. No other system layer is required for the implementation of these objects and no other entity than the object itself must be present in our system. This makes all the singularity and the power of our project.

Our study raises many problems primarily related to two following contradictions:
flexibility versus performance and communication versus system protection. Moreover, compatibility with the current source code is not to neglect and must be accounted for.

Among the diverse existing objects, we chose to implement those present in the prototype-based languages. Our system is built from small executables each representing a prototype of object. Each object will be able to communicate with others by inheritance or by message delivery in the manner of object-orientated languages like Self [11]. Here, we use our prototype language, Lisaac [7], notably with dynamic inheritance for Isaac operating system design [8]. The Lisaac compiler product ANSI C code. However, we should emphasize that the granularity of our objects is not as fine as in languages which possess *integer* or *boolean* types of objects. Indeed, in our operating system, we represent a physical entity of the material by a single object (for example: a hard disk, a screen, a keyboard). In the context of this article, we consider that our objects all belong to the same process. It appears obvious to us, that the choices at the level of the memory manager are of primary importance to ensure a fast communication between our objects. The communication is essentially done by sending messages to an object either client or parent. To simplify this mechanism, we need a memory space common to all our objects. This will work as a stack. We know that segmentation allows the separation of the code, the data and the stack in logically independent spaces of addressing. But also, it facilitates the sharing and the protection of these spaces [9]. Our idea is to make better use of this mechanism to allow the implementation of a common stack of execution between several objects which have their own data (see fig. 2). As in *Multics* [1] we chose to benefit from the advantages of the MMU and the modularity of the segmentation.

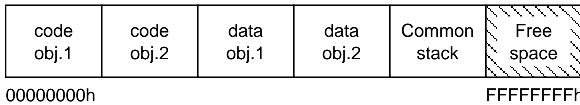

Fig. 3 Vision of the virtual linear space of *Isaac*.

Moreover, if one wishes that our system does not remain a theory, it is necessary that it has all the assets to allow a compatibility with the current systems. Recreating entirely a compiler adapted to our needs is not desirable for two obvious reasons : the time of development on a given architecture becomes too imposing, and the re-use of the existing C programs becomes too difficult. Hence for matters of portability, compatibility and reusability, we generate our objects with a C compiler. This gives us a solid base for a compatibility with GNU products. Our objects are compiled and can be seen like an individual C programs. For example, we regard the *emacs* program as an object communicating with the *libc* object ensuring a *Unix-type* of compatibility with the whole of our system. We produce a C program for each *Isaac* object and we use a common stack to all the objects belonging to the same process. But the C compilers do not make it possible to clearly separate stack spaces. Still, we may have a solution which allows it without modifying the compiler. On a whole, this allows better memory management and finer protections than today's. Moreover, another consequence arises from the reduction of the space virtually occupied by a program. This profit makes it possible to have a single space of addressing for several objets/programs (see fig. 3).

1.3 The problem with segmentation using a C program

In this part, we will explain why the C language is poorly adapted to the use of segmentation. Our demonstration stays valid for the majority of the current languages and compilers. That it is on a 32 or 64 bits processor, the address of an element (variable or function) in a *C* program is materialized only by one index. To address an element using segmentation, we need a couple of indexes : the first indicates the number of the segment to be used (segment register), the second represents the offset pointing on the element in question. In general, the main problem lies in the separation in two distinct segments of the data and the stack of a program in the memory. Indeed, in the following example, we can notice that the system is unable to infer which of segment is containing the data pointed by the variable `Ptr`. It can either point on the stack segment, on the data, or on the code.

```
int global ;
void main()
{ int local ;
  int *Ptr ;

  switch (getch()) {
    case 'G': Ptr = &global; break;
    case 'L': Ptr = &local; break;
    case 'C': Ptr = main; break;
  } ;

} ;
```

For most of the pointers, We can easily get to know by flow analysis which is the segment used. Nevertheless, this solution is not complete and forces us to make a massive modification of the stack. For example in a Intel architecture, it would be necessary to modify a compiler to consider a pointer, not on 32 bits but rather on 48 bits thus taking into account the number of the segment on 16 bits. Moreover, without a flow analysis to let us know in the most of the cases which of the segment to consider, it becomes compulsory to load a segment registers each time we access a pointer. It appears obvious that the overall execution performances would be profoundly diminished.

2 Our approach

Here, we show that our solution is applicable without massive modification of the compiler and that it admits an acceptable loss of performance. Indeed, we do not modify the instructions of use of the pointers generated by the compiler. Our technique uses logical considerations and some astuteness during the execution of the program. During a pointing instruction, the processor needs to know which segment to consider. In fact, this information is lost by the compiler as the pointer has only one offset. Altogether, the pre-indexation of registers of segment is by default correct. These are the considerations which will help us to solve most cases.

- the code always uses the code segment per default i.e. that any indirect instruction of branching or call uses implicitly the code segment register. Therefore, as the pointers of functions or labels are always used within the framework of execution of a block of instructions, there's no real problem of segmentation. Another use of this type of pointer does not hold sense in a correctly written program.
- The access to the local variables is always by default on the stack segment. Thus, there is no access problem to the local variables, even if the stack segment is different from the data segment. For example on an *INTEL* architecture, access to local variables uses the `ebp` index/offset register which implicitly takes the `ss` stack register as segment.
- the variables whether aggregated or allocated are by default on the data segment. For example on an *INTEL* architecture, all the instructions that are not using the `ebp` (base of the stack) and `esp` (top of the stack) index

registers take by default the `ds` register (data segment) as default segment.

There remains only the data pointers which can be either on the data segment or sometimes on the stack. By default the processor always takes the register of data to point. It is in this singular ambiguity that our system comes into play. To guarantee a good effectiveness and to clear any ambiguity, we decided to act not on the generated code, but during the execution of the program. By default, the processor will take one of the two segments, it is only in the event of failure that we take action. The difficulty resides in detecting an error of segment addressing at the time of use of a pointer and to redirect addressing towards the good segment. This lies upon three essential questions :

1. How can we detect the addressing errors due to the segmentation?
2. How and on which segment is it necessary to redirect this instruction of addressing?
3. How can we discern this error from a real addressing error?

2.1 How can we detect the addressing errors due to the segmentation?

To answer our first question, we have to consider the two following points : first of all, we know that a segment of the stack type starts at the top of stack up to the highest addresses (4 GigaBytes on 32 bits processors). On the other hand, a data segment starts at offset zero and extends up to the top of the stack. We can thus establish the following intervals :

$$I_{data} = [0..x] \text{ and } I_{stack} = [y..z] \text{ with } x < y \text{ and } z : \textit{Addressable offset limit.}$$

These two intervals clearly reflect the segments declared and maintained by the operating system. They guarantee to us that addressing with an error of segmentation causes a violation at the level of the operating system. This violation results in an overflow system exception which lets us solve the problem at the time of execution of the faulty instruction. Consequently, we will act at the segment overflow exception to redirect the segment register towards the good one, and thus allow to continue the execution of the program at the level of the faulty instruction.

2.2 How and on which segment is it necessary to redirect this instruction of addressing?

That brings us an easy answer to this second question, the choice of the segment is restricted to two possibilities,: the stack or the data segment. A gross analysis of the faulty instruction makes it possible to determine the segment register in question and to know its value. If this register points on the data segment then we redirect it on the stack segment or conversely. Our *ping-pong* (flip-flop) technique makes it possible to solve the problem of segmentation only if that is necessary.

2.3 How can we discern this error from a real addressing error?

The distinction of a segmentation error compared to an error of addressing is something much more delicate. This checking must be fast, because it does not bring anything other than some protection. Indeed, with the current implementation of our flip-flop in the overflow exception, an error of addressing would cause a everlasting loop without any exception triggering. To solve this problem, we thought of two possible implementations. The first simply consists in checking if the pointed element belongs to the one of the two intervals. This was not retained in our implementation as deemed ineffective. Indeed, if it is fast to detect the segment register in question by a gross analysis of the faulty instruction, it is not for the value of the pointer on certain instruction sets (*INTEL* for example). The body of the overflow exception : The second technique rests on the use of another exception system : the `trace` exception. Indeed, in the Trace mode, the processor starts the trace exception at the end of each instruction. At the time of the release of an overflow exception, two cases are to be considered : the pointed data is on the other segment (stack or data) or it program has indeed failed! To allow to clear this last ambiguity, we switch the processor to *Trace* mode, we redirect the register in question on the presumedly good segment and we re-execute the instruction. Thus, in the event of admission of this instruction by the processor, the trace exception is started. In the event of failure, the processor releases again an overflow exception. At this level, we know if the pointer is correct or if we have an irremediable error. In the case of a correct pointer, the processor is switched back to normal mode in the trace exception so that we can continue the program execution.

```
If (FlagTrace) Then
    Fatal Error!
Else
    If (Reg.Segment == Segment.data) Then
        Reg.Segment <- Segment.Stack.
    Else
        Reg.Segment <- Segment.data
    Endif
        FlagTrace <- True.
Endif.
```

The body of the trace execution:

```
FlagTrace <- False.
```

This solution although correct is not very efficient. Indeed, triggering overflow exception is very costly. For each segmentation problem we have two exceptions are triggered with this algorithm. For this reason, we had to modify slightly this algorithm to reduce the cost to a single exception trigger in most cases. If the processor is not switched to `trace` mode, one can ascertain that the overflow exception will be produced repeatedly on the same instruction. The idea is to lay by the position of the faulty instruction in the overflow exception. Hence, If the current position is the same as the previous one, the `trace` mode is triggered.

The optimized body of the overflow exception :

```
If (FlagTrace) Then
    Fatal Error!
Else
    If (Reg.Segment == Segment.data) Then
        Reg.Segment <- Segment.Stack.
    Else
        Reg.Segment <- Segment.data
    Endif
    If (PC_old == PC_fault) Then
        FlagTrace <- True.
    Endif
    PC_old <- PC_fault.
Endif.
```

Notice : the body of the `trace` exception does not undergo any modification. With this modification and for a correctly written program, the release of two exceptions to solve a problem of segmentation is almost non-existent. The solution we bring can be used to solve different problems than ours related to the communication or the management of the virtual memory in existing or future operating system.

2.4 implementation of the method

We established this method on our operating system which uses GNU compiler GCC on a Intel 386 architecture [3]. The modifications made to the compiler are as following :

- Detection and exit of the code segment towards the data segment of the index tables of label generated at the time of some important *switch* (to separate the segment of code from the data).
- Detection and exit of the code segment towards the data segment of the constant variables. (to separate the segment from code of the data).
- Rewriting of the file describing the memory mapping of the code and the data for the *linker*, thus forcing the starting of the offsets at zero for the data.

We can see here that no modification is internal to the compiler itself, and that as long as GCC does not change its assembler syntax code and the mapping file for the *linker*, our modifications are portable and follow the evolution of the versions of the compiler.

3 Measuring Performances

To allow to quantify as well as possible the loss of performance of our method, we took two measurements with two variants each. The first measurement gives an idea of the performance of our method during the execution of a program. The second measurement is used to quantify the loss of the performance in the worst case. For each measurement, the first variant consists in withdrawing the use of the `trace` mode to compare the speed at execution in the case of a comparison of the intervals with the faulty pointer. The second alternative represents implementation with checking by release of the `trace` mode. Measurements were taken on a 400MHz *INTEL pentium II*.

3.1 Overview of the execution of a program

The purpose of this measurement is to give an idea of the number of releases of the mechanism at the time of program execution. As our system is currently only a prototype, the only code of acceptable size that we have for this type of measurement is the system in itself. It has approximately 100 000 lines of C. Our measurement is made entirely on the loading of the system, from the boot to the loading of the graphic interface. We also carried out the loading of the system with the old allocation of the segments (thus without any release of exception), but the difference in performance in term of execution time is negligeable.

	Number of exceptions triggered	Execution Time
Without Segmentation	0	82.27 seconds
Without *trace*	9015	82.92 seconds
Without *trace*	9015	82.92 seconds

We can note that the number of exceptions triggered of our mecanism is relatively low. The overall performances are down by less than one percent of execution time. This seems satisfactory considering the possibilities that segmentation offers in the prospect of project.

3.2 In the worst cases

Our second test puts our method under the worst conditions by the program below. Indeed, with each iteration the pointer changes segment. That causes to start our mechanism of exception. Moreover, this release is always carried out on the same position in the code, thus causing the implementation of the `trace` mode to validate the instruction.

The second measurement was taken on the following program :

```
int global=0;
void main()
{ int local=0, Cpt;
  int *Ptr ;

  for (Cpt=0;Cpt<1000000000LU;Cpt++) {
    Ptr=((Cpt&1)?(&local):(&global));
    (*Ptr)++;
  };
};
```

	Number of exceptions triggered	Execution Time
Without Segmentation	0	34.16 seconds
Without *trace*	1 000 000 000	43.04 seconds
Without *trace*	2 000 000 000	46.37 seconds

Obviously, the performances in terms of time execution are considerably affected. We can notice an extra cost passing from 26% to 37%. Fortunately this type of program is not current. Besides, altogether in our Isaac system, no release of the `trace` mode is necessary to solve the problems related to segmentation. We thus avoid double exceptions.

4 Related Work

There is no recent related work. Today, the segmentation is used for an other usage: Extension of 32-bit pointer for memory manager.

To my knowledge, no compiler goes so far as to take into account the segmentation in a transparent manner, i.e. the possibility of mapping in memory any programs having the data and the stack in two distinct segments. Indeed, taking it into account without a modification of the source code is not easy. Nevertheless, amongst the compilers which we have tested on this subject, the implementation by the *Watcom C*[1] version 11 remains interesting. There is the */zu* option which makes it possible to declare during the compilation the separation of the stack segment with that of the data. The use of this option on a *C ansi* source causes the generation a *warning* of the type,: *'pointer truncated'* with each assignment of a pointer.

Example:

```
/* Our C ansi Pointer */
char *Ptr;

/* A variable in the data segment */
char Global;

void main()
{ char Local; /* In the stack segment */

  /* Output a Warning: Pointer truncated */
  Ptr = &Local;

  (*Ptr) ++; /* A dangerous Usage ! */
};
```

In our example, the message indicates that we are affecting a 48-bit pointer (segment + offset) delivered by `&Local` in a variable pointer `Ptr` of 32 bits (offset only). Thereafter, the use of this pointer causes the program to stop by exception release. To solve and correct this error, the compiler admits a new variable of pointer type on 48 bits, i.e. that it is necessary to modify the source code by adding `_far` to each declaration of pointer.

Here our example accepting segmentation :

```
/* Our 48-bit pointer (Segment + Offset).*/
char _far *Ptr;

/* Variable in the data segment.*/
char Global;

void main()
{ char Local;    /* In the stack segment.*/

  /* All pointers are on 48 bits*/
  Ptr = &Local;

  (*Ptr) ++;     /* No problem !*/
};
```

Thus the *Watcom C* compiler allows segmentation with a modification of the source code using a new type of declaration of a pointer. Other compilers talk about the management of the segmentation. We can quote work of the *mentor*[2] C compiler or the *INTEL*[3] one. But we could not test their implementation by the lack of compiler. It should be observed that in the Eighties, INTEL developed a proprietor language 'PLM' which carried out segmented code. The paper [2] shows another approach of the use of segmentation and of the memory management on the INTEL processor.

5 Conclusion

In this article, we denounce and criticize the lack of use of memory segmentation in operating systems. The advantages of this mechanism are numerous and we could underline certain small problems in the management of the memory which is not segmented under Unix (section 1.1). Within the framework of our research *Isaac* project, its use appears natural and essential for the implementation of communication techniques (section 1.2). In practice, the current compilers do not take into account in

[1] Copyright by Sybase.

[2] `http://www.mentor.com/embedded/papers/whitepapers/Using80x86/`

[3] `http://www.intel.com`

a transparent manner the processor segmentation and worst still, make its use impossible. Moreover, we saw that a massive modification of the compiler would be likely to affect the performances as regards to the execution time of a program (section 1.3). By taking into account these considerations that we propose a transparent method allowing a solution of this problem with a negligible cost of execution. This one applies without modification of the code generated by the compiler. It is based on the use of exception system to solve the problems of segmentation during the execution of a program (section 2). Some rare compilers take into account the segmentation of the processor, but that is not transparent for the programmer (section 4). The implementation of our method in this project enabled us to validate these performances and to achieve our goals by the use of segmentation (section 3).

References

1. A Bensoussan CCRD (1972) The Multics Virtual Memory: Concepts and Design. In: Communications of ACM, vol 15, number 5, pp 308–318
2. Tzi-cker Chiueh PP Ganesh Venkitachalam (1999) Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In: 17th ACM Symposium on Operating Systems Principles, ACM Press, pp 140–153
3. Hummel RL (1992) Structure and management Memory. In: Programmer's Technical Reference: The Processor and Coprocessor, Ziff-Davis Press, pp 83–104, ISBN : 1-56276-016-5
4. Mullender S (1993) Address-Space and Memory Management (15.2). In: Distributed Systems, Addison-Wesley and ACM press, pp 387–191, ISBN : 0-201-62427-3
5. Rifflet JM (1994) Espace d'adressage virtuel. In: La programmations sous unix, p 410
6. Sonntag B (2000) `http://isaacos.loria.fr/` or `http://www.isaacos.com`. Web site of `Isaac/Lisaac` project.
7. Sonntag B, Colnet D (2002) Lisaac: the power of simplicity at work for operating system. In: 40th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'2002), Sydney, Australia, Australian Computer Society, pp 45–52
8. Sonntag B, Colnet D, Zendra O (2002) Dynamic Inheritance : A powerful Mechanism for Operating System Design. In: Intercontinental Workshop on Object-Orientation and Operating Systems (OOOSWS'2002) - ECOOP'02 Workshop Reader, pp 25–30
9. Tanenbaum A (1992) Memory manager: Segmentation (3.7). In: Modern Operating Systems, Prentice Hall, pp 144–159
10. Tanenbaum A (1992) `Mach` case: shared memory (15.3.2). In: Modern Operating Systems, Prentice Hall, pp 731–739
11. Ungar D, Smith R (1987) Self: The Power of Simplicity. In: 2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), ACM Press, pp 227–241