32/64 Bit Portability Issues

Ray Lanza

Advanced OSF Software Group

11/19/91

64-bit architectures provide extended capabilities such as support for larger address spaces and scalar arithmetic ranges. These enhanced capabilities introduce a number of compatibility problems that must be addressed while porting software. Many of the compatibility problems can be avoided through careful design and attention to datatypes.

Overview

There is a set of issues involved in porting applications from ULTRIX to OSF1. Another set of issues exist in porting applications from a 32-bit OSF system to a 64-bit OSF system. These are two disjoint sets of issues. This paper deals with the later. Porting ULTRIX applications to OSF is documented in the HERCULES Porting Guide.

64-bit architectures provide wider registers to hold larger scalar datatypes and memory addresses. They also provide arithmetic and logical instructions to operate on these registers. It's these capabilities that differentiate 64 bit systems from 32-bit systems. Unfortunately these capabilities also introduce portability issues.

Most applications are written in one or more high level languages. A discussion of portability is only practical in the context of one of these languages, as applications written in assembly or macro assembly need complete rewrites or translations. For the purpose of this paper the language of interest is C but variations of the problems occur with other languages.

Much of the information in this paper is a direct result of work done on a port of ULTRIX-32 V4.0 to a 64-bit architecture. The port was done as a research/advanced development effort. In addition to the operating system, approximately 280 user level commands and utilities were ported. While this represents a considerable body of code, it's not clear if this is truly representative of user applications.

During the port we learned a number of things:

- Most well written programs compile and run without change
- Most 32/64 bit portability problems can be avoided through the use of good programming practices
- Most 32/64 bit portability problems are a direct result of changing one or more of the datatypes
- Programs living in a pure 64-bit environment can ignore most data size issues
- Programs producing/consuming data from 32-bit programs may need design and development effort to solve problems

Datatype Definitions

A 64-bit system provides support for greater addressibility and larger ranges for scalar arithmetic operations. Providing this functionality for user applications involves changing one or more of the scalar datatypes. Unfortunately these changes result in interoperability problems between 32/64 bit systems and are the main cause of portability problems.

The primary consideration for the choices outlined in the following table is in fact portability.

With these definitions it is possible to define data structures that do not change size from system to system.

Data Type	32 Bit System	64 Bit System
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
pointer	32	64

As you can see from the table ‘long’ and ‘pointer’ change from 32 to 64 bits. These changes provide the applications developer with support for all of the supported scalar types and expands addressing beyond the limits imposed by today’s 32-bit systems.

The ‘long long’ datatype will be supported in future versions of the 32 and 64-bit C compilers. It provides the ability to read and write 64 bit scalar data and may provide a degree of interoperability for new programs being developed.

Types of problems

There are five basic problems or issues that need to be addressed:

- Objects change size
- Sizeof(*) != Sizeof(int)
- 32 <=> 64 bit interoperability
- Atomicity
- Read/Write ordering

Objects Change Size

Data objects that include pointers or longs change size. The following is an example of a simplistic linked list data structure.

```
struct foo {
        struct foo *next, *prev;
        int mode;
        char *name;
};
```

On a 32-bit system this structure occupies 16 bytes of memory. On a 64-bit system this structure will be 32 bytes long. 12 bytes of the growth are a result of the three pointers doubling in length. The other 4 bytes are a result of alignment padding. [1]

The fact that this data structure changed size may not be a problem. If the program runs on a 64-bit system and the data it produces and consumes remains on 64-bit systems, no harm occurs except for a potential size problem. The problem exists in environments where the 64-bit program must consume data produced by a 32-bit program or it produces data that is consumed by a 32-bit program.

The key to defining compatible data structures is to avoid the use of long and pointer declarations. This may seem like a difficult task but in fact it's relatively easy. Declarations of long can be replaced by int to preserve sizes. In serious database-oriented applications pointers rarely appear in declarations that are written to mass storage devices. These applications are normally concerned about storage efficiency and already avoid pointers.

In the case where a 64-bit program must deal with 32-bit data structures containing pointers more work is required. One approach is to define a new data structure that encapsulates the old structure while preserving the alignments, and then being careful to perform I/O to and from the encapsulated structure. This technique assumes that data written out in pointer fields is irrelevant and will be filled in when the structure is memory resident. These problems are not limited to files on disk, they can happen with all forms of mass storage including tape.

*Sizeof(*) != Sizeof(int)*

Of all of the problems anticipated as a result of the datatypes chosen this was the one that concerned most people. The example below results in truncation of the value stored in 'buffer' inspite of the casts used. There were very few of these problems in the port of ULTRIX-32 to the 64-bit architecture. Most instances were in virtual memory related kernel code or the bourne shell, '/bin/sh'.

[1] OSF and ULTRIX compilers align data on 'natural boundaries'

```
int foo;
char *buffer;
        buffer = (char *)malloc(128);
        foo = ( int )buffer;
        buffer = (char *)foo;
```

The 64-bit version of the product will contain an enhanced version of lint that will issue a warning when an assignment of this type is attempted.

Another form of this problem is encountered with aliased data structures. We have seen code that defines multiple structures for the same object instead of using unions.

```
struct foo {                            struct bar {
        int src_addr, dst_addr;                 struct bar *next, *prev;
        char *name;                             char *name;
};                                      };
```

This is similar to a problem found in the 'ip' portion of the network code. The source and destination internet address are stored as 32 bit integers. As the data is passed up through the network layers the fields containing the network addresses are reused as linked list pointers. This works well on a system where a pointer is the same size as an int. It doesn't work with the datatype choices made for 64-bit systems.

Problems of this type are difficult to find and correct. In the 'ip' case lower layers of the system used the 'foo' declaration and upper layers used the 'bar' declaration. Running lint on all the modules in a program or using ANSI C function prototypes will detect a miss-match in arguments. Unfortunately an appropriate cast may hide this condition.

Another form of this problem occurs with function arguments. If you pass a 32-bit constant or an 'int' to a function which expects a 'long' or a pointer the object is likely to be truncated. Again using ANSI C function prototypes or 'lint' on older programs will solve this problem.

The final form of this problem involves default types. Constants defined without the 'L' suffix, variables declared as 'unsigned' without a type and function arguments declared without a type all default to 'int'. In many cases this may be correct however if you attempt to use these to hold a value longer than 32 bits you may observe truncation.

Interoperability

In previous sections the data portability problems were limited to mass storage devices. The mass storage related problems are really just a simple form of more general inter-process communications problems. Any time a 32-bit program communicates with a 64-bit program they must agree on the format and type of data. Therefore, the same issues can occur over a local or wide area network. If the 64-bit product supported both 32-bit and 64-bit execution environments the problem would be extended to include other IPC mechanisms such as System V shared memory and message queues.[1]

64-bit systems will support files larger than 2 gigabytes.[2] They will also function as fileservers for other systems including existing 32-bit systems. Files may be created that can't be fully addressed by 32-bit clients. It will also be possible to mount disks from 64-bit systems on 32-bit systems and have the same problem.

Atomicity

Some 64-bit architectures do not support byte or word load and store operations. The smallest unit of memory access is the longword, which is 32 bits. High level language byte and word accesses are accomplished with multiple instructions. In most cases this fact can be ignored by an applications developer. It becomes important whenever multiple threads of execution are sharing data. This can happen when two or more processes share data via shared memory or in a multi-threaded application where all threads share a common address space.

```
char byte[16]
```

As an example using the declaration above, if threadA attempts to update byte[0] while threadB updates byte[1] there is a chance that the access of one thread will affect the other even though they aren't trying to update the same variable. The following is an example of the code used to update a single byte.

```
8: byte[1] = 5;
 [tst.c: 8] 0x120000230: 43e0b40f   addq zero, 0x5, t7
 [tst.c: 8] 0x120000234: 203e0001   lda at, 1(sp)
 [tst.c: 8] 0x120000238: 2f410000   ldq_u k0, 0(at)        -----------
 [tst.c: 8] 0x12000023c: 49e1017b   insbl t7, at, k1
 [tst.c: 8] 0x120000240: 4b41005a   mskbl k0, at, k0
 [tst.c: 8] 0x120000244: 475b041a   bis k0, k1, k0
 [tst.c: 8] 0x120000248: 3f410000   stq_u k0, 0(at)        -----------
```

[1] Not part of the product plans at this time

[2] 32 bit systems are currently limited to 2 gigabytes

A 5 instruction sequence is used to load the quadword containing the byte, update the byte with the right value and store the result back to memory. If threadA starts this operation and is interrupted between the load and store and threadB gets control and updates byte[0] or byte[2] threadB's changes will be overwritten when threadA completes it sequence.

Programs that work on nonshared data on VAX and MIPS aren't affected because they have 'atomic' read and write access at the byte granularity level. The same program might fail on a 64-bit system and must be coded appropriately by using longword or quadword data aligned on natural boundaries.[1]

Read/Write Ordering

Some 64-bit multiprocessor systems pose an additional problem for applications developers. They do not guarantee write ordering between multiple processors[2]. If variable A and B are written to memory in order on one processor they will appear to be written in that order on that processor but might appear in a different order on another processor. Multiple programs and/or multithreaded programs running on a multiprocessor system sharing data cannot depend on write ordering. They must use memory barrier instructions to order writes.

Migration Aids

Lint is an effective tool used to find argument mismatches and dubious assignments. Programs written in ANSI C benefit from tighter control in these areas as well.

64-bit OSF products will include a 32-bit to 64-bit portability guide in addition to the ULTRIX to OSF migration guide.

Conclusion

Most 32-bit OSF programs should compile and run normally on 64-bit OSF systems. Programs that do not pay attention to the proper use of datatypes my fail but can be easily fixed with minor changes. The use of lint and/or ANSI C features will detect these abuses and the corrected program will be backwards source compatible.

Programs that do not read or write data containing pointers or long variables will have a great deal of interoperability and are also likely to run unchanged. Programs that do not have to produce or consume data for 32-bit programs should run unchanged.

[1] longword is equivelent to the int datatype and quadword is equivelent to the long datatype

[2] I/O devices are defined as processors

Programs that must deal with scalar data produced by 32-bit programs may require minor data definition changes. Programs that must deal with data containing 32-bit pointers may require significantly more work.