Porting 32-bit Software to 64-bit Platforms

Key Challenges & Solutions for C/C++

Author: Pavan D, Sagar Gaonkar

Abstract

Porting 32-bit applications to 64-bit platforms (processors and operating systems) is far from straightforward. Programmers struggle with the implementation of code involving pointers, data alignment, bit fields and memory addressing. The issues faced typically arise from some commonly made mistakes while programming the 32-bit software. If left uncorrected while porting to 64-bit processors, these mistakes prove costly and usually result in run-time issues and instability of software. This paper provides a step-by-step guide to programmers for porting a 32-bit application to a 64-bit processor. It is targeted primarily for programmers who use C/C++ though also relevant to other programming languages.

Figure 1 – Challenges in porting software from 32-bit to 64-bit processors

Contents

Figures

Introduction

With the increase in number of 64-bit processors ad OSes available in the market, it is becoming very important to make software/applications compatible to run on these processors/OSes *(referred together as platforms in this paper)*. One of the fastest means to achieve this is by migrating existing 32-bit applications to the 64-bit platform of interest. However, the challenges of migrating an existing 32-bit application to a 64-bit platform are many. In this whitepaper we discuss issues and challenges faced during such a porting activity. Debugging tools to be used during this process are also recommended.

Key differences between 64-bit and 32-bit platforms:

- Higher register sizes, memory address widths in 64-bit processors
- More integer and floating point registers in 64-bit processors
- 64-bit Operating Systems (OS) support addressing higher amount of memory(RAM)

The above mentioned differences make the 32-bit to 64-bit migration challenging. The issues faced during these migration can be broadly classified into **Processor/OS Agnostic Issues** and **Processor/OS Dependent Issues**.

Processor/OS Agnostic Issues

Pointer Arithmetic

Arithmetic operations using offsets from pointers are often done for quick access to the next data location in memory. This should ideally be done by querying the size of the data type and not by using fixed offsets in software because pointer sizes change when moving from a 32-bit processor to a 64-bit one. Fixed offsets result in inaccurate computations and therefore unintentional memory accesses, thereby causing instability of operation.

Example: Proper access in a 32-bit program but out of bound access for a 64-bit program

```
int             i               = -5;
unsigned int    j               = 2;
int             array[10]       = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int*            ptr             = array + 5;

ptr = ptr + (i + j); //Invalid pointer value on 64-bit platform
printf("%i\n", *ptr); //Access violation on 64-bit platform
```

Let us understand the calculation flow of "ptr + (i + j)" in the Example 1:

- According to C++ rules, variable i of int type is converted to unsigned int type
- Addition of i and j occurs. The result we get is value 0xFFFFFFFD of unsigned type
- Next, the calculation of "ptr + 0xFFFFFFFDu" takes place but the result depends on the pointer size on the processor's memory architecture. If the addition is performed in a 32-bit program, the given expression will be equivalent to "ptr - 3" and the resulting print would be 3. However, In a 64-bit program, 0xFFFFFFFDu value will be added to the pointer and the result

will be outbound pointer to the array. And we'll face problems while getting access to the item of this pointer.

Data Alignment

Depending on the platform and internal module processing requirements, there can be a situation where the pointers should be aligned to a particular integral multiple before processing. Care needs to be taken to avoid pointer arithmetic issues while deriving the aligned address. Typecasting to a void pointer or an integer to perform arithmetic operations can cause application to crash and not perform as expected when ported to 64-bit platforms.

Example: Incorrect typecasting memory pointers

```
if(0 != (((int)memory_ptr) & 0x03)) //Data loss
{
        used_memory       += (4 - (((int)memory_ptr) & 0x03)); //Data loss
        memory_ptr        = (void *)((int)memory_ptr + 4); //Data loss
        memory_ptr        = (void *)((int)memory_ptr & 0xFFFFFFFC); //Data loss
}
```

The above code will work fine on a 32-bit program but not on a 64-bit program. The mistake here is that the memory pointer is typecast to an invalid type, resulting in data loss.

Bitwise Operations

Bitwise operations when performed on a 32-bit data type and a 64-bit data type can be different. Since the number of bits are different in 64-bit data types when compared to 32-bit data types, it is important to write the code such that the data type is queried before performing the operations. This way the code can remain generic and can be executed on 32-bit as well as 64-bit platforms. Fixing the length of bits to be processed or moved using bitwise operators can result in issues while porting.

Processor/OS Dependent Issues

Data Model differences

Data models define the lengths of pointers and data types such as integers. While these lengths will be generally same in 32-bit platforms, the same is not true for 64-bit platforms. 64-bit platform adopt one of three basic models - LP64, ILP64 or LLP64. For instance, Windows follows LLP64 while iOS, OSX and Linux follow LP64, etc. These data models differ in the lengths of different data types.

During migration to 64-bit platforms it is important to understand the data model supported by the target platform. The model supported by the platform will govern the data types to be used and the pointer arithmetic associated with it.

Example: Data types and their sizes

```
int       i  = 2^31; //Valid for LP64, ILP64 and LLP64 models
long      l  = 2^63; //Valid for LP64 model. 'long' is 32bit in LLP64 and ILP64 models
long long ll = 2^63; //Valid for LP64, ILP64 and LLP64 models
```

Position Independent Shared objects (x86-64)

A compiler creates an executable by compiling source files and linking with other object files (system library). This executable will have virtual addresses of all its symbols (functions, variables, etc.) as

absolute address values (and not offsets from a base address). When executed, this executable locates each symbol using this absolute address value. This works well because an executable is always loaded and executed from the same virtual address. However, this approach is not workable for an executable that is dependent on a shared object, as the virtual address where the shared object is loaded is not known to the executable.

To address this challenge, 32bit (x86) platforms have a mechanism in-built which creates a link at run time in the form of a reference table containing the exact memory addresses of the symbols defined in shared objects. This makes the shared objects on 32bit (x86) platforms Position Independent. **Position Independent Code (or PIC)** is a body of machine code that executes properly regardless of its location in the program memory (absolute address).

The in-built mechanism of making shared libraries position independent was discontinued on 64bit platforms because of below reasons,

- It did not support addresses value needs to more than 32 bits
- It came at a cost of additional memory overhead every time shared objects were used

In order to make shared libraries position independent on 64bit platforms one needs to use '***-fPIC***' compiler option during compilation of the shared object. '*-fPIC*' is absolutely necessary for 64bit (x86-64) platforms for reasons stated above.

Disadvantages: PIC objects are usually slightly larger and slower at runtime than the equivalent non-PIC object because,

- There is a time overhead in replacing direct address references with relative addresses at compile time
- There is space overhead in maintaining information to help the runtime loader fill in the unresolved addresses at runtime

Page size differences

A page (also called memory page or virtual page) is a fixed-length contiguous block of virtual memory, which is the smallest unit of data for memory management in an operating system. In a system the thread stack sizes should be multiple of the page size. 64-bit platform tend to have higher page sizes than their 32-bit counterparts. This will necessitate that the thread stack sizes to be updated to be multiple of the page size of the 64-bit platform.

Example: 32-bit iOS has a page size of 4KB whereas in 64-bit iOS it is 16KB. So, if one uses a thread stack size of 20KB, it will work for 32-bit iOS but will return an error for 64-bit iOS. It is recommended to use APIs like 'getpagesize()' to get determine thread stack sizes instead of hardcoding them.

Recommended Tools

Valgrind

Valgrind is an open source software framework that provides tools/mechanisms to debugging applications while they are running on the platform. Some of the tools that Valgrind provides are memory error detector, two thread error detector, cache and branch-prediction profiler, call-graph generating cache, branch-prediction profiler and heap profiler. These are supported on multiple platforms like x86/Linux, ARM/Linux, ARM64/Linux, ARM/Android etc. Valgrind can help debug issues

faced while porting 32-bit applications to 64-bit platforms. Being a dynamic analysis tool, it helps in localizing the issue as well as specifying the reason. Issues like segmentation faults, software crashes due to memory misalignments, etc. can be easily located using Valgrind.

Static Code Analyzers

Static code analysis helps in identifying issues with source code by using techniques such as data flow analysis and taint analysis. There are multiple tools that statically analyze code to identify "potential" programming pitfalls. These tools help in assessing all parts of the code which may or may not get executed during a specific run. One of the popular open source static code analyzer is Clang. There are other sophisticated commercial licensed tools from multiple vendors that perform static code analysis.

Code Review

Code Review is a systematic examination of a programmer's source code. The intent is to locate and fix issues that have crept into the code during programming. There are different kinds of code review that can be conducted: walkthroughs, inspections or pair programming. Several commercial and open source tools are available in the market. Code reviews for the key considerations shared in this paper can help locate the porting bugs even before the code is run on the 64-bit platform.

Availability at Ittiam

Ittiam's adroitSDKs are available on multiple platforms including 32-bit and 64-bit variants. The guidelines described in this paper are based on Ittiam's extensive experience in delivering solutions on various processor/operating systems. The coding practices and system architecture of adroitSDK makes porting a simple and smooth experience.

Conclusion

As 64-bit platforms are being adopted in the industry, quick migration of existing 32-bit application is very important. In this paper, we discussed the common issues faced during such migration and ways to tackle then. This will also serve as a guideline for programmers while developing 32-bit systems to ensure that their software can be migrated easily to a 64-bit platform when required.

References

1. http://www.unix.org/version2/whatsnew/lp64_wp.html
2. https://www.technovelty.org/c/position-independent-code-and-x86-64-libraries.html
3. http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/
4. http://www.valgrind.org/

Disclaimer

This white paper is for informational purposes only. Ittiam makes no warranties, express, implied or statutory, as to the information in this document. The information contained in this document represents the current view of Ittiam Systems on the issues discussed as of the date of publication. It should not be interpreted to be a commitment on the part of Ittiam, and Ittiam cannot guarantee the accuracy of any information presented after the date of publication.