An Approach of Windows Memory Management Simulation on Linux

Rui Li, Nanjun Yang, Shilong Ma
State Key Lab. of Software Development Environment
School of Computer Science & Engineering, Beihang University
Beijing 100191, China
{lirui, ynj.t.g, slma}@nlsde.buaa.edu.cn

***Abstract*—So far open source software has been developed for several decades. Linux has gradually become one of the major operating systems. The issue that Windows application migration can be migrate to Linux is raised. However, there is great difference in the implementation mechanism between Windows and Linux. In this research, we try to build an middle layer which between application and operating system to shield the differences between the underlying operating system for the upper layer application. The middle layer provide unified fixed interface by packaging different operating system calls, so as to make the same source code can be directly translated on different operating systems with no change. It can achieve the migration of the application in source code level. In this paper, we introduce the Simulation Windows memory management mechanism. We build a Windows-like virtual memory management mechanism which simulates Windows virtual memory management on Linux. And also implement Windows heap management mechanism on Linux. And finally, two experiments are given to test and verify our approach.**

Keywords- virtual memory management; heap management; simulation; operating system; algorithm; migration

I. INTRODUCTION

So far open source software has been developed for several decades. Excellent design and extraordinary performance, coupled with the strong support from IBM, INTEL, CA, ORACLE and other internationally renowned companies, Linux has gradually become one of the major operating systems. From the view of software providers, if the developed application software can adapt to both Windows and Linux, to some extent, it partly determines competitiveness and market share of the application software. From software user's opinion, it will make convenience for they work if the Windows application software they familiar with can run on Linux and provide the same operation model. But as we know, there is great difference in the implementation mechanism between Windows and Linux. Applications running on the operating system do not access the system resources directly. It follows the manner required by the operating system. This way is called system call. System call closely relate to operating system. And it is not compatible between two different operating systems. As Windows and Linux for example, system call with same external behavior but different function definition and internal implementation. Especially for Windows, System call is actually not the final interface between kernel and program, Windows completely based on DLL mechanism, its system calls packaged by the DLL mechanism, this is the so-called Windows API [1].

It had a goes in the computer field, any problem in computer science can be solved by another layer of indirection. So, we try to build a middle layer, it provide unified fixed interface by packaging different operating system calls, so as to make the same source code can be directly translated on different operating systems with no change. To achieve the migration of the program in source code level.

We know the memory management is very important in the operating system. For most Windows applications, it is almost inevitable to operate and manage the memory. But there are differences in memory management between Windows and Linux. For example, Windows memory management emphasizes the two-tier structure: virtual memory management and heap management. While in Linux memory management is not like that. And system calls of heap management in Linux cannot use to implements Windows functions which called "fixed address redistribution of the heap".[2] In our work, we build a Windows-like virtual memory management mechanism which simulates Windows virtual memory management on Linux. And also implement Windows heap management mechanism on Linux.

The remainder of this paper is structured as follows: in Section 2, we discuss related work; Section 3 contains a presentation of the simulation of the Windows memory management mechanism that we designed on Linux, the experimental results and conclusions are given in Section 4 and 5, in Section 6, some future directions we intend to follow are described.

II. RELATED WORK

The related works for VM Scheduling can be categorized into two categories: kernel differences compensate in kernel and kernel difference compensate out of kernel.

Some important effort research on kernel difference compensate out of kernel, such as [3], [4], [5], [3] is directed toward similar goals to ours'. In [3], the authors describe WINE which is not Windows emulator. It is an API conversion technology, use Linux system call function to implement the corresponding function of Windows API. The goal runs Windows programs on Linux can be achieved. In [4], the POSIX system call API library under Win32 system is proposed. The software on POSIX systems (such as Linux, BSD and other UNIX systems) can be migrated to Windows

978-0-7695-4863-0/12 $26.00 © 2012 IEEE
DOI 10.1109/WCSE.2012.34

by recompiling. In [5], a cross-platform, open source automated build system is introduced.

Longene [6] the Linux Compatible kernel project, is the study on kernel differences compensate in kernel. Longene was investment and presided in 2005 by Insigma Technology Co., Ltd. It was Designed to allow users to run Windows applications directly on Linux, without having to depend on the Windows operating system.

There are important differences between the approach we propose and the ones mentioned above. Notably:

• According to the API conversion technology, for API with obvious correspondence relationship, we also use API conversion method. However, for API without correspondence relationship, our approach reconstructs a set of Windows-like mechanisms to achieve these functions.

• Compare to Longene, our approach does not need to modify the Linux kernel. Developers does not require to familiar with the Linux kernel.

• It can be used as part of the application software.

III. WINDOWS MEMORY MANAGEMENT MECHANISM SIMULATION

The application basically cannot run without the memory management. But the memory management of Windows and Linux are different in design and implementation which we talk about in section 1. In this paper, we reconstruct Windows-like memory management mechanisms on Linux. For example, Windows virtual memory management, Windows heap management and so on.

A. Windows virtual memory management simulation

In Windows, virtual memory of process was divided by page size. Each page has three states, there are free, reserved and committed respectively. The free page can be redistribution. For example, it can be allocated as heap space. The reserved page cannot be redistribution and access. After the reserved page is mapped to physical memory, the state of page converts to "committed". Users can access through pre-defined way. In Linux it does not have the concept of reserved and committed. From above-mentioned, we know free page can be converted to reserved page. Reserved page or committed page can be converted to committed state. Other transformation methods cannot be allowed. Allocations, release, protection, locking and so on are the operation of the virtual memory.

Our approach is that open a special continuous space in virtual address of Linux as the scope of the "Windows-like virtual memory management". In another word, we use "Brk" in the program initialization to set aside a chunk of new space. All of the virtual memory management functions only acts on this space. Fig 1 shows the Windows-like virtual memory management solution.

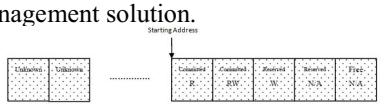

Fig.1. the Windows-like virtual memory management solution

In Linux, the page which is free or reserved will be set "PROT_NONE" to deny access. The committed page can translate the access rights which are user needs into expression forms of Linux.

The critical data structure of virtual memory management is listed as follows.

Data structure :

```
#define TOTAL_VIRTUAL_PAGES 1048576;// the total virtual page number of process, default value is 1048576, each page is 4KB
#define MAXIMUM_VIRTUAL_PAGES 131072;// Page number used for Windows-like virtual memory management
typedef struct _VM_INFO{
    DWORD PageSize;// The size of system page
    LPVOID Base;//The starting address of Windows -like virtual memory management
    char MemoryState[TOTAL_VIRTUAL_PAGES] ;// the state of page from the starting address
    SIZE_T AllocSize[TOTAL_VIRTUAL_PAGES] ;// the information of the committed page
    DWORD Protection[TOTAL_VIRTUAL_PAG ES];// the access rights of each page
    char Locked[TOTAL_VIRTUAL_PAGES];//the locked state of each page
  }VM_INFO
```

B. Windows heap management simulation

We know in Windows, heap is a section of reserved or committed page. So Windows heap management is built on the basis of the virtual memory management. The heap management is different from virtual memory management. The heap management needs special data structure which is not required in virtual memory management, such as control module, the memory block list and so on. Memory block which is allocated is linked by the list.

In Windows, the heap is divided into growable heap and non-growable heap. Growable heap and non-growable heap have initial size. Growable heap has no maximum size limit while non-growable has. For non-growable heap, any operation which try to allocate more space than maximum size will fail. The operation on heap includes create, allocate, release, destruction and redistribution. Redistribution can implements with the starting address of the memory block unchanged (default is can be changed). This function is one of heap management in Linux which cannot be realized.

Our approach is that linked the memory block by list, and operates on it. Next we would describe the simulation of growable heap and non-growable heap in details.

For growable heap, allocated and unallocated are all located in a continuous page after the initial address of the heap. Space of the initial size will be submitted and space of the maximum size will be reserved. With the allocation of heap space, committed page of the initial size will run out. Then the heap will continue to submit a page to meet the allocation requirements until the reserved page can not

satisfy the demand of distribution. Fig.2. shows the structure of growable heap.

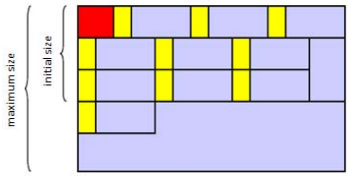

Fig.2. the structure of growable heap

For non-growable heap, just reserve and submit the page of initial size. When the page cannot meet the allocation needs. Virtual memory management will find another space to meet the allocation requirements by use “VirtualAlloc” till the system memory cannot meet the requirements of distribution. In this case, the heap space may be is not continuous. Fig.3. shows the structure of non-growable heap.

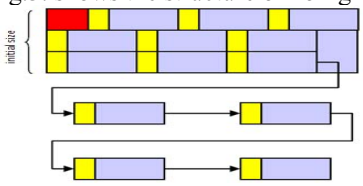

Fig.3. the structure of non-growable heap

The critical data structure of heap is listed as follows.

Data structure :

```
#define MAXIMUM_HEAPS XX//the maximum number
of heap, XX is the custom
int HeapNum;// The number of the heap in current
HANDLE HeapHandle[MAXIMUM_HEAPS];// the
handle of each heap
typedef struct _HEAP_BLOCK_HEAD{// The head of the
heap memory block
    struct _HEAP_BLOCK_HEAD *PreBlock;// The
previous memory block
    struct _HEAP_BLOCK_HEAD *NextBlock;// The
latter memory block
    DWORD Size;// Available size of block
    char Free;// Is a free block or not
} HEAP_BLOCK_HEAD
typedef struct _HEAP _HEAD{// The head of the heap
    HEAP_BLOCK_HEAD *FirstBlock;// The first
memory block of heap
    DWORD Size;// Memory size reserved for the heap
    DWORD MaximumSize;//The maximum size of heap
   char Growable;// Is a growable heap or not
   char Executable;// Whether you can run the code in the
heap or not
} HEAP_HEAD
```

As mentioned above, each heap has a head to record information, and so do each memory block in heap. Memory blocks are linked through the double-linked list. In the information of heap memory block, the most important information is the identity of free statement. Almost all operations of heap are performed in free block.

IV. EXPERIMENTS

In this section, we test and verify our approach with two experiments. One is for virtual memory management, and the other is for heap management.

A. Experiment Ⅰ

This experiment is designed for testing the Operating functions of virtual memory management on a special continuous space which is opened for "Windows-like virtual memory management".

First, we use “VirtualAlloc” to reserve a space whose starting address is “a” and size is 10,000 bytes.

Second, try to reserve the already reserved space.

Then, submit 100bytes from starting address “a”, and fill the committed space with random characters.

Finally, release the reserved space from starting address “a”. and then try to read the already released space.

The execution process can be presented graphically as below:

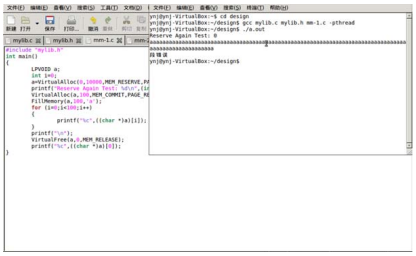

Fig.4. The execution process of Experiment Ⅰ

Fig 4 shows that the reserve operation of virtual memory was implement correctly. To reserve the already reserved space was lead to failure. The allocated virtual memory can read and write. Access operation was rejected when it was executed on released virtual memory. From the experiment result, all of the Windows virtual memory operations can be executed on the continuous space which is opened for "Windows-like virtual memory management".

B. Experiment Ⅱ

We simulate Windows heap management on the basis of virtual memory management. The test case we designed for the testing of growable heap and non-growable heap, especially for “fixed address redistribution of the heap”.

The experiment we designed is listed as follows.

Test Case :

1:Initialization：

- create heap h1, which is growable heap
- create non-growable heap h2, whose initial size is 10,000 bytes and maximum size is 100,000 bytes

2: (1)allocate a space whose starting address is “a1” and size is 100 bytes from h1, and fill the allocated space with random characters

(2) continue to allocate a space whose starting address is “a2” and size is 1,000,000 bytes from h1.

(3) output address “a1” and “a2”

3: (1) allocate a space whose starting address is “a1” and size is 100 bytes from h2

(2) allocate a space whose starting address is “a2” and size is 100 bytes from h2
(3) allocate a space whose starting address is “a3” and size is 100 bytes from h2
(4) allocate a space whose starting address is “a4” and size is 100 bytes from h2
(5) allocate a space whose starting address is “a5” and size is 30,000 bytes from h2
(6) output address “a1”, “a2”, “a3”, “a4” and “a5”

4: (1) output the size of the memory block “a1” in heap h2
(2) fill the allocated space “a4” in heap h2 with random characters
(3) release the allocated memory block which starting address is “a2” in heap h2.
(4) release the allocated memory block which starting address is “a3” in heap h2.

5: (1) try to modify the size of memory block which starting address is “a1” in heap h2 into 1,000 bytes with starting address unchanged
(2) try to modify the size of memory block which starting address is “a1” in heap h2 into 250 bytes with starting address unchanged
(3) output the size of the memory block “a1” in heap h2.
(4) try to modify the size of memory block which starting address is “a4” in heap h2 into 30,000 bytes, and output its starting address

5: destroy the heap h2, and and then try to read the memory block in already destroyed heap

The execution process can be presented graphically as below:

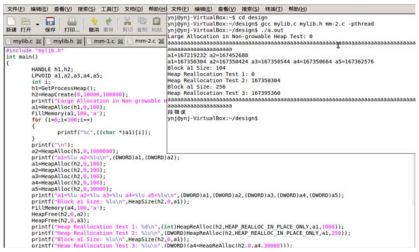

Fig.5. The execution process of Experiment Ⅱ

Fig 5 shows that the size of memory block allocated more than maximum size will not be allowed. It will call the virtual memory management to reserve and submit another new space when the size of memory block allocated more than initial size in both growable heap and non-growable heap. Not only variable address redistribution of the heap but also fixed address redistribution of the heap is simulated successfully. From this experiment result, our approach basically realizes the simulation of Windows heap manager on Linux.

V. CONCLUSIONS AND FUTURE WORK

For most Windows applications, it is almost related to memory management without exception. As an essential part of operating system, all types of operating systems provide this functionality. Due to different design concepts, different operating system has different concrete realization which leads to correspond in semantic incomplete. The absence of corresponding implementation in semantic between different operating systems could be an obstacle to Windows application that need to migrate to Linux with no or little changes. To address this problem, we proposed a simulation approach to implement Windows implementation mechanisms on Linux. Experiments show that the approach we introduced can help Windows program with no changes run on Linux directly and effective.

As mentioned earlier, the approach we proposed has some drawbacks. The most essential is time efficiency. The emulation layer which was added will cause the application to require a longer running time.

As future work, we will analysis the factor which affect the time efficiency and improve the algorithm we designed for smarter migration.

ACKNOWLEDGMENT

We would like to thank students of the same laboratory for their comments on the paper and also for proofreading the text. This work is partially supported by National Science and Technology Support Program (2011BAH14B04).

REFERENCES

[1] MSDN Library[EB/OL].:http://msdn.microsoft.com/en-us.
[2] Fan Wenqing, Zhou Binbin, An Jing. " Proficient in Windows API: Function, Interface, Programming examples"[M], Beijing, Posts and Telecom Press. Feb.2009.
[3] Wikipedia, WINE, http://en.wikipedia.org/wiki/Wine_(software)
[4] Geoffrey J.Noer, Cygwin: A free win32 porting layer for UNIX Application, August 1998, Cygwin distribution is available at http://www.cygwin.com/
[5] CMake, http://www.cmake.org/
[6] Longene. http://www.longene.org/whitepaper.php, 2009-02-01
[7] Kay A.Robbins, Steven Robbins. "UNIX System Programming: Communication, Concurrency and Threads"[M], Beijing, Mechanical Industry Press, 2005
[8] developerWorks[EB/OL]. : http://www.ibm.com/developerworks/cn/linux/kernel/syscall/part1/appendix.html#2
[9] Wikipedia[EB/OL].: http://en.wikipedia.org/wiki/C_standard _library.
[10] Common Language Runtime Overview, http://msdn.microsoft.com/en-us/library/ddk909ch(vs.71).aspx
[11] Migration station. http://www-128.ibm.com/developerworks/ondemand/migrate/Linux.htm