

Check-pointing Process Heaps and Cleaner API Interception for Process Migration

Fall 2000 – MCS Project

Mujtaba Khambatti

Project Advisor _____

Dr.Partha Dasgupta

Project Reader _____

Dr.Donald Miller

1 Introduction

Over the past decade, there has been a lot of research culminating in different implementations for Process migration. Today, the term ‘process migration’ has a different meaning to most people. For instance, earlier attempts at process migration focused on load balancing of processors in a distributed system [1, 4, 5]. However this conventional interpretation is not entirely true today. Extensive kernel support for such preemptive load balancing efforts had high overhead and also was found to be unnecessary [1].

We have shown that by targeting shrink wrapped applications we can use process migration to increase mobility, collaborative work, distributed systems management, automatic reconfiguration and fault tolerance [1, 8]. Moreover all this can be achieved without modifications to the kernel or the applications. Therefore the term ‘process migration’ takes on a different meaning in contemporary computing.

This approach of process migration relies on a concept called API interception that enables virtualization of resources. API interception essentially is a method of capturing the interactions between the application and the Operating System at the API level and possibly recording those interactions as information about the application. Virtualizations centers on de-coupling the application from the resources provided by the operating system, like files, network connections, etc. [1, 6, 9, 10]

My goals for the MCS project were the following:

1. Document existing code written by Ravikanth Nasika towards his Master’s thesis [2].
2. Implement process migration using the mediating connectors library.
3. Checkpoint the heaps of a process so that they can be brought up again at a new location.

2 Background

2.1 API Interception

For the purpose of process migration it is necessary to capture the state of the process as it was on the original machine in order to re-create the same state on the destination machine. A process can be thought of as comprising of binary information within its own process space and external connections. The former includes process text and data, while the latter encompasses connections with the operating system or anything that is outside the process space that the process itself does not have direct access to. Obtaining the external state of the process is less trivial as the records of these connections are maintained outside the process space. It is our experience through work in API interception [2, 3] that the external state of the process needs to be captured from the time of creation of the process, i.e. when the process is loaded. For this purpose we use API interception techniques [2, 3] that will acquire crucial information about the process like: what files it opens, what graphics devices it uses or what network connections it establishes [1]. In brief, API interception makes possible the following seemingly difficult tasks:

- It provides the application with new facilities without changing the code of the application.
- It incorporates more features into the operating system, without changing the API calls.

2.2 Virtualization

Virtualization is the fundamental idea behind the power of a process migration mechanism [9, 10]. As a concept it is already very much prevalent in contemporary computing. Any process can be considered to be a virtualization of a CPU that runs within a logical address space and accesses virtual memory [1]. For the purpose of process migration, we extend the scope of the virtualization model to include all the CPUs, memory and operating systems available to us. This will give us a high level view of a distributed system whose elements have been virtualized at the global level.

An application operates from within a shell. The shell is responsible for de-coupling the application from the operating system without the application noticing any difference in its environment. This makes it possible to map all the identification information, like resource handles, that the operating system actually uses to virtual handles that can be given to the application by the shell. The mapping between the physical handles and the virtual handles can

further be altered to enable the process to operate on a completely different machine with new physical handles for its files and network connections, etc. without noticing any change in the environment. We have successfully implemented such virtual handle translation tables [2, 3].

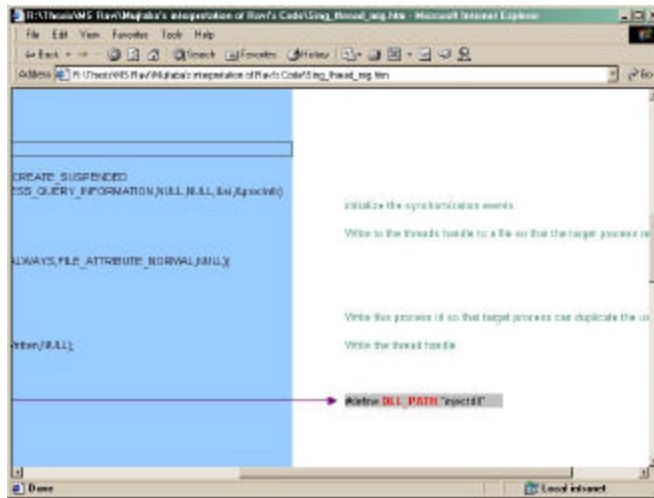


Fig.3: Apart from the cross-references and ease of readability that was demonstrated in fig 2, the above displays the textual comments that were included for sections of the code.

4 Implementing Process Migration with Mediating Connectors

4.1 Mediating Connectors

This toolkit provided by the University of Southern California, Information Sciences Institute, makes use of *mediators* and *wrappers*. The mediation of one or more APIs from a shared library provides, in effect, a new library that relies on the original. The new library exports the same interface as the original, and actually shares with the original, the binary code that is common to both. The wrapper author is relieved of the need to re-implement any portion of the library he does not wish to mediate. A wrapper comprises of a set of mediators. Each mediator in the set mediates a distinct function, which must be a function exported from some shared library. The functions mediated by a wrapper need not all come from the same library. A mediator has access to the same parameters as the function; and its return value, if any, will be seen as a value returned by the function. Within its implementation, a mediator may choose to call the actual function one or more times, using the result(s) of the call(s) to compute its own result. [11]

4.2 Wrapper Definition

The wrapper definition is done in a text file that can be used by the provided API for the purpose of API interception at runtime. A wrapper definition looks like this:

```
wrapper name implementation "C:\Code\WrapperDef\name.dll"  
  wrap CreateProcessW in kernel32 with My_CreateProcess size 40  
  ...  
  wrap CreateThread in kernel32 with My_CreateThread size 24
```

4.3 Wrapper Implementation

A wrapper is implemented by a Windows dynamic linked library. The wrapper's mediators must be functions exported from this library. A mediator must return the same type as the function it mediates. Every mediator must have its first parameter as type `void*` [11].

A mediator for the API `CreateThread` looks like this:

```
__declspec(dllexport) HANDLE WINAPI My_CreateThread(void *icall,
                                                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                                                    DWORD dwStackSize,
                                                    LPTHREAD_START_ROUTINE lpStartAddress,
                                                    LPVOID lpParameter,
                                                    DWORD dwCreationFlags,
                                                    LPDWORD lpThreadId)
{
    /* perform anything here */
    HANDLE rslt = (HANDLE)InnerCall(icall);
    return rslt;
}
```

4.4 Wrapper Programming

The dynamic linked library that is shipped along with the Mediating Connectors tool exports the functions that are needed to load and unload the wrapper in the process specified. Additionally it provides debugging functionality. The two important functions of this DLL are:

```
BOOL WrapProcess (const char *WrapperDefinitionFilePath,
                  const DWORD pidOfProcess, void *WrapperParam,
                  const BOOL secure)
```

```
enum UnwrapResult {UNWRAPPED, PROCESS_NOT_ACCESSIBLE,
                  NO_SUCH_WRAPPER};
```

```
UnwrapResult UnwrapProcess (const char *WrapperName,
                             const DWORD pidOfProcess)
```


4.5 Architectural Changes

The original process migration [2] is carried out using 3 important modules mentioned below:

1. The process loader
2. The DLL injector
3. The injected DLL and check-pointing code.

4.5.1 The Process Loader

This module is responsible for loading the process that potentially might be migrated. This module is called the middle layer, migrator, or vEX [12]. It sits on individual machines and allows the creation of a process that potentially can be migrated in the middle of its execution. The loader is closely coupled with the DLL injector. Since the Mediating Connectors toolkit now provided functionality for wrapping of the API calls made by a process, we could do away with the DLL injector. Thus the coupling between the first two modules was removed and a Wrapper library call was made for a simple and clean injection of a specified DLL into the loaded process.

4.5.2 The DLL injector

This module allocated memory in another process and then created a remote thread in that process. The procedure for DLL injection is described in detail in [2, 7]. Since the toolkit performed this operation, we no longer needed the injector.

4.5.3 The injected DLL and check-pointing code

According to specifications of the toolkit, the injected DLL had to contain the mediators for the API calls that we were interested in modifying. So the initial part of this assignment was to write the mediators for the API calls so that when the application made these calls, important information could be saved in data structures or files. This information would help define the external state of the process in terms of the open resources that it currently was using and also help identify internal state changes that took place at runtime, like heap creation, etc. Thus the mediators were written to maintain records of whatever might seem interesting.

Following is a list of some of the mediators that were written:

Original API	Name of Mediator
CreateProcessW	My_CreateProcess
CreateProcessAsUserA	My_CreateProcessAsUser
CreateThread	My_CreateThread
VirtualAlloc	myVirtualAlloc
HeapCreate	myHeapCreate
HeapAlloc	myHeapAlloc
GetStdHandle	myGetStdHandle
Sleep	mySleep
SetStdHandle	mySetStdHandle

Apart from implementation of these mediators in the wrapper, the toolkit required that on process termination (which would happen during a migration event), the injected DLL be unloaded. Therefore, code was added to ensure that the wrapper was successfully unloaded from the process memory.

Further, there was a fair amount of re-use from the original work done by Ravikanth in the implementation of the check-pointing code. However during his thesis work he was unable to successfully migrate processes with heaps. As part of the third goal, I implemented code that made heap check pointing successful.

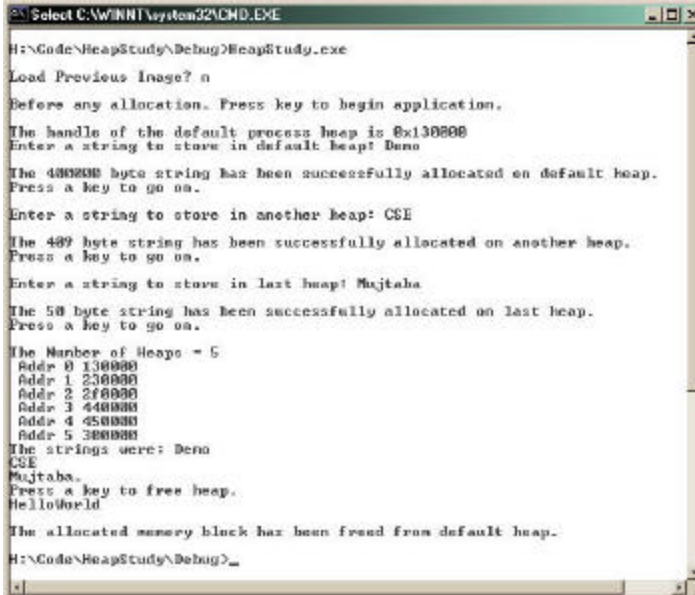
5 Check pointing the heaps of a process

A simple process was created that allocated memory using `new` / `malloc()`, and `HeapAlloc()` function calls. The heaps were check-pointed into a file at the end of the process and then successfully brought up again on a new machine.

Following are some screen snapshots of the process that check-pointed heaps:

5.1 First Execution

The process asks if there exists a previous image of the heaps to load. Then it asks for some data to store in each heap and allocates an arbitrary amount of space in each heap. Before process termination it displays the number of heaps of that process and the data that was stored in them. The last string on the console was hard-coded data allocated to the CRT heap. All heaps are now check-pointed into a binary file.

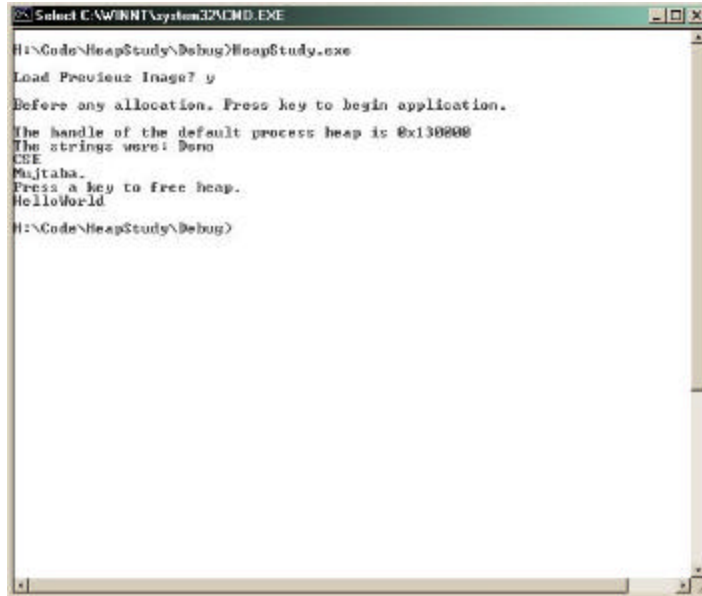


```
H:\Code\HeapStudy\Debug>HeapStudy.exe
Load Previous Image? n
Before any allocation. Press key to begin application.
The handle of the default process heap is 0x130000
Enter a string to store in default heap! Demo
The 400000 byte string has been successfully allocated on default heap.
Press a key to go on.
Enter a string to store in another heap: CSE
The 489 byte string has been successfully allocated on another heap.
Press a key to go on.
Enter a string to store in last heap! Mujtaba
The 50 byte string has been successfully allocated on last heap.
Press a key to go on.
The Number of Heaps = 5
Addr 0 130000
Addr 1 230000
Addr 2 2f0000
Addr 3 440000
Addr 4 650000
Addr 5 300000
The strings were: Demo
CSE
Mujtaba.
Press a key to free heap.
HelloWorld
The allocated memory block has been freed from default heap.
H:\Code\HeapStudy\Debug>
```

Fig.4: First Execution of the process

5.2 Second Execution on a different machine

The second time the process can be executed on a different machine. This time we bring up the check-pointed heap binaries and load them into the process space. This method has been successfully tested on different machine configurations of the Intel Pentium series and also on machines with Windows NT and Windows 2000.



```
Select C:\WINNT\system32\CMD.EXE
H:\Code\HeapStudy\Debug>HeapStudy.exe
Load Previous Image? y
Before any allocation. Press key to begin application.
The handle of the default process heap is 0x130000
The strings were: Demo
EIE
Muftaha.
Press a key to free heap.
HelloWorld
H:\Code\HeapStudy\Debug>
```

Fig. 5: Second Execution of the process

The Mediating Connectors toolkit is used here again to provide mediators for the following calls:

Original API	Name of Mediator
VirtualAlloc	myVirtualAlloc
HeapCreate	myHeapCreate

The Process Execution Block (PEB) is accessed as described in [2]. This provides vital information about some of the process heaps. The memory region of the process is queried to find out information about certain ranges of pages within the process memory that coincide with the heaps. The binary information from these heaps is then check-pointed into a file that can be restored later.

An interesting observation is that the CRT heap is implemented as a doubly linked list of blocks. The current implementation algorithm is not guaranteed to be compatible with future releases of the library. So especially for the CRT heap it is not advisable to checkpoint only the blocks that have data within them. However to provide a more universal approach, an entire section of memory between the CRT heap base address and the beginning of the data section is checkpointed. Although this might seem like a lot of memory to dump into a file and reload again, one must be reminded that the focus is not on the execution efficiency of a heap checkpoint. The measurement of the overhead incurred in such a checkpoint might comprise part of future work or further study.

6 Conclusion

I have successfully implemented all three parts of the goals that I had determined at the beginning of my MCS project. I am grateful to Dr.Partha Dasgupta for providing me both the resources and the advisement during the course of this project. I also wish to acknowledge the effort of Dr.Donald Miller who has taken his time to review and certify this project document by his valuable experience in related fields of work.

7 References:

- [1] Dasgupta P. General-purpose Process Migration, <http://cactus.eas.asu.edu/partha/Papers-PDF/proc-migration.pdf>
- [2] Nasika R. "Migration of Communicating Processes Via API Interception", Master's thesis, Arizona State University, 1999.
- [3] Hebbalalu R. "File Input/Output and Graphical Input/Output Handling for Nomadic Processes on Windows NT", Master's thesis, Arizona State University, 1999.
- [4] Litzkow M. "Remote Unix Turning Idle Workstations Into Cycle Servers", Proceedings of Usenix Summer Conference, 1987.
- [5] Petri S., Langendörfer H., "Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes", Operating Systems Review, Vol. 29, No. 4, October 1995, pp. 25-36.
- [6] Nasika R., Dasgupta P., "Transparent Migration of Distributed Communicating Processes", 13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS-2000). August 2000.
- [7] Richter J. "Advanced Windows", Third Edition, Microsoft Press.
- [8] McLaughlin D., Sardesai S., and Dasgupta P., "Preemptive Scheduling for Distributed Systems", 11th International Conference on Parallel and Distributed Computing Systems, 1998.
- [9] Baratloo A., Dasgupta P., Karamcheti V., and Kedem Z.M., "Metacomputing with MILAN", Heterogeneous Computing Workshop, International Parallel Processing Symposium, April 1999.
- [10] Dasgupta P., Karamcheti V., and Kedem Z.M., "Transparent Distribution Middleware for General Purpose Computations", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 1999.
- [11] Balzer, R.M.; Goldman, N.M., "Mediating Connectors", Information Sciences Institute, 19th IEEE International Conference on Distributed Computing Systems Workshops, 1999.
- [12] Boyd T., Dasgupta P., "Virtualizing Operating Systems for Seamless Distributed Environments", 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2000), November 2000.