DUMP: Dump User Memory, Please

David W. Duffey, Daniel Andresen
Dept. of Department of Computing and Information Sciences
234 Nichols Hall
Kansas State University
Manhattan, KS 66506
{dduffey, dan}@cis.ksu.edu

June 25, 2002

1 Introduction

While working on several busy Linux research machines, we found the need to temporarily remove a running process's memory resources. Creating new swap areas and killing existing running processes (simulations) were not viable solutions to our problem.

Being involved with several Beowulf systems at school, we knew that these tasks could already be solved with complex clustering technology, but we desired a simpler solution. We needed a solution that was only a subset of what clustering technology could solve; we wanted a solution that was easy to use. Rather than migrating a process to another machine, we wanted to migrate a process to a future time.

Simply creating new swap areas is not always possible (or at least requires a sysadmin), and has limitations. Sysadmins have no control over which processes are swapped nor to where processes are swapped, and they have to find the space to create new swap areas. In addition, swap space cannot be archived, compressed, or moved to another location. It would be ideal if users could dump, compress, and move their own memory resources at will (think of it as a living core file).

DUMP is a loadable kernel module that allows a user to remove memory usage of a process and later inject its memory back into the process and continue execution. DUMP requires no complex cluster setup, special libraries, or new userspace utilities to work.

DUMP is not intended to be used for cluster process migration or checkpointing/restarting, rather it is the adaptation of some of this technology for use by the average user on a single machine. Process migration is the ability to move a currently executing process from one machine to another. Checkpointing/restarting is the ability to "remember" the state of a process at particular points of execution and then return to a previous state of execution if needed. The ability to checkpoint/restart a process often (but not necessarily) implies the ability to do process migration.

2 Related Work

Several userspace utilities provide checkpointing abilities but require modifying the process to be manipulated. Some implementations work at run-time by replacing loadable system libraries (like *libc*), or during development time by using special libraries, or by using a modified compiler. Unfortunately, this severely limits the number of applications that are able to be manipulated in they way we wanted. Standard binaries distributed with one's preferred Linux distribution will likely not work with these solutions.

Because a userspace solution would severely limit use by the average Linux user, we chose to focus on a kernel implementation.

MOSIX provides process migration through a Linux kernel patch [2, 1]. When a process executes, it has a unique home node (UHN) and this process can be migrated to another MOSIX-enabled machine (but a place-holder is left on its UHN). If the process tries to open a file, socket, or another resource provided by the kernel, the kernel will re-route the request to the UHN. MOSIX is not available as a kernel module.

Epckpt, is a Linux kernel patch that allows checkpointing and restarting of processes [3]. Before a process executes, a default signal handler is installed for checkpointing, and the kernel begins logging data on the process (open/close syscall, etc). As the process executes, Epckpt collects data that will allow it to be checkpointed. The process must be marked as checkpointable before execution begins, and it will introduce (a small) execution overhead. Once a process has been checkpointed, it can be killed and restarted on another Epckpt machine with a similar execution environment.

CRAK, part of the ZAP project, is an adaption of Epckpt which provides similar ability but as a kernel module (no patching necessary) [6, 5]. It also removes data collection overhead and does not require a special signal handler to be installed prior to execution. For all practical purposes, CRAK and Epckpt have the same ability to checkpoint/restart and/or migrate processes.

Unfortunately, these three solutions are too complex for the common single machine setup. MOSIX requires a complex cluster setup; MOSIX and Epckpt require kernel patching; Epckpt and CRAK use extra userspace utilities, /dev entries, and will break some applications.

3 Why DUMP?

By completely removing a process from a running kernel, Epckpt and CRAK are giving up important resources such as the Process ID, Device I/O, etc. "For example, top always seg-faults after the restart [with CRAK]" [6]. Epckpt and CRAK on the Linux platform are fundamentally flawed in that they cannot know which resources that have changed since last execution will affect the process. For example, a newly restarted application has a different Process ID, but it (or another process) could have stored the old PID in an integer variable. Another example, if a process modifies an open file, restarting the process from an earlier execution time may have unexpected results. Network communication is even more complex. It is my opinion that successful checkpointing/restarting requires a dramatic change at all levels of software engineering (from operating system, to userland software development).

DUMP leaves the process as a "place holder" that retains important process data (Process ID, locked memory, etc), while removing its biggest resource usage: memory. Of course, this is at the expense of not being able to migrate the process to another machine (or across a reboot). As long as a process can survive a SIGSTOP and SIGCONT, it can be DUMPed. Even this restriction can be removed in future releases by using 2.5 series kernel features (fake run queue). A robust implementation coupled with ease of installation and use are the primary goals of DUMP.

4 How It Works

The DUMP module works by reading and writing process' memory areas and their associated information through the /proc filesystem. The DUMP module will create a */proc/dump* directory that will contain a file for each process; the filenames are simply the pid of each respective process.

By reading from a process's associated pid-file, we will be DUMPing information from the process to the user and releasing the resourses from the kernel. This information can be saved to disk, compressed, and/or sent across a network device. The resources can be injected back into the process by simply writing the DUMPed data to the correct /proc entry.

4.1 Virtual Memory Areas

The proc filesystem includes a subdirectory for each process containing valuable process information, including virtual memory area maps (VMAs). If we look at /proc/740/maps we get 54 lines, each containing a VMA.

The first column shows the starting and ending addresses, followed by permissions (read, write, execute, shared), the offset into the source mapping (in this case a file), the device the map is taken from (in this case, the second partition on the first SCSI drive), and the inode of the source mapping.

The first line is sendmail's read-only code segment, the second is the initialized writable data segment, and the last entry is for uninitialized writable pages (code and data). The other 51 VMAs are similar in nature to these three.

4.2 What Is DUMPed

Since the first VMA is read-only and we have the file from which to restore, we only need to remember the filename and we can throw away its pages. The second VMA is writeable and has a mapped file, so we only need to save those pages that have been modified (currently DUMP saves all pages in a writable area). The third VMA has no mapped file, so we must dump all pages within this range.

4.3 File Format

The resulting file format of sendmail.dump is a binary file. First is an integer value containing the number of dumpable areas, in this case all 54. The first VMA structure immediately follows (start address, ending address, offset, protection bits (not discussed here), flags (permissions and more), filename length) and filename (if there is one). If the VMA is writable, then immediately following the optional filename are pages of data that make up the starting to ending address range. If the VMA is not writeable, we don't need to save the memory data and the next VMA follows.

The flags field also includes information about whether the VMA is a device mapping (such as XFree86 mapping a graphic card's memory area) or if the area is locked (for such use as DMA transfers). In either of these cases, the VMA would not be DUMPable.

5 Modularization

Much work was put into finding functionality that allows us to provide a module source that will work across many Linux distributions without modification to the distribution's kernel tree.

CRAK is able to use several existing kernel functions such as "do_mmap_pgoff" (adds a VMA to the *currently* running process) that DUMP cannot. This is because the CRAK uses a userspace utility to restart a process; the userspace utility actually becomes the restarted process (similar to a fork-exec) and a simple do_mmap_pgoff will be modifying the correct (currently executing) process. But one of the goals of DUMP was to avoid requiring special utilities, so we need to access (and manipulate) the memory mapping structures of the dumped process when it is *not* currently running. This means we had to rewrite certain procedures that assumed that the "current" (a pointer to the currently executing task)

START	END	PERM	OFFSET	DEVICE	INODE	MAPPED FILE
08048000	080b2000	r-xp	00000000	08:02	163350	/bin/sendmail
080b2000	080b5000	rw-p	0006a000	08:02	163350	/bin/sendmail
080b5000	080ff000	rwxp	00000000	00:00	0	
...						

Table 1: /proc/740/maps

FIELD	DESCRIPTION
vmacount	number of dumpable VMAs
...	
start	starting address of VMA
end	ending address of VMA
pgoff	offset into file
pgprot	protection bits
flags	permisions
fn_/len	filename length
filename	memory mapped filename
data	dumped (writable) pages
...	

Table 2: DUMP file format

process was the one to be manipulated.

We also needed access to non-exported symbols insert_vm_struct, vm_area_cachep and get_user_pages. CRAK had a similar need, but for different reasons, so we adopted CRAKs method of reading /boot/System.map to find the addresses of these non-static but non-exported symbols [6].

There is certainly more than one way to accomplish dumping and restoring another process's memory, but we found that functions differed from the stock 2.4 kernel tree and what RedHat distributed (fortunately we did development on both). Some kernel functions were exported in one tree, but static in another. Functions varied in parameters, static-ness, and whether they are exported or not. The combination of kernel functions we chose work on both kernel trees and allows DUMP to run on a stock kernel, or one provided by RedHat.

The code to provide the dynamic pid-files in /proc/dump was shamelessly pulled from linux/fs/proc. The 255 lines of code to modularize this function of DUMP would become about a 20 line patch to the kernel if we were to create a new "dump" entry in the currently existing /proc/[pid] architecture (which is not expandable without patching).

The code of DUMP would dramatically decrease in size and complexity if it were created as a kernel patch instead of a module, but this would limit the use and adoption of DUMP.

6 Resources

If you are interested in peeking at DUMP's code, we found several resources to be immensely helpful. Linux Device Drivers was helpful in explaining the overall picture of Linux memory management [4]. The kernel source is also a great source of information.

mm/mmap.c	Mapping files and VMA manipulation
fs/binfmt*	Loading executable data (uses mmap.c)
fs/proc/*	/proc fs examples
Documentiation/cachetlb.txt	Documents get_user_pages

Table 3: Useful kernel files

7 Conclusion

DUMP is easy to install, easy to use, and doesn't try to do too much.

DUMP is still considered alpha quality code and shouldn't be used on production systems. Several error conditions are not currently being handled (such as out-of-memory situations), and the existing code has not been thoroughly tested. Also, permissions are not enforced, which would allow any user to map any file (regardless of permissions) into one of their processes.

Future work on DUMP includes supporting copy-on-write pages (don't save COW pages if not modified) and creating a stable, robust, and secure module. Once DUMP has proven itself as a module, a simplified kernel patch will be made that would be more appropriate for inclusion into the standard kernel.

References

[1] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, March 1998.

[2] MOSIX, September 2002. http://www.mosix.org/.

[3] Eduardo Pinheiro. Truly-transparent checkpointing of parallel applications, September 2002. http://www.cs.rutgers.edu/∼edpin/epckpt.

[4] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers.* O'Reilly and Associates, 2nd edition, 2001.

[5] Gong Su Steven Osman, Dinesh Subhraveti and Jason Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, December 2002.

[6] Hua Zhong and Jason Nieh. CRAK: Linux checkpoint / restart as a kernel module. Technical Report UCS-014-01, Columbia University, Department of Computer Science, November 2001.

A Installation

You can download the latest tarball of DUMP from http://DavidDuffey.com/dump. Untar the archive and it will create a directory structure *dump*. The *module* directory has a Makefile that will build a loadable kernel module dump.o, The Makefile assumes /usr/src/linux and /boot/System.map are associated with your currently running kernel.

After loading the module, there will be a directory */proc/dump* that will contain a file for each process. The filenames are simply the pid of each respective process. If you read from /proc/dump/740 you will be dumping process 740's memory structures and content. If you write a suitable dumpfile to /proc/dump/740 you will be restoring process 740's memory.

B Sample Usage

After booting RedHat 7.3, top reports sendmail as having used 1824k of memory with 1304k of it being shared (1.4% of total memory). We dump the sendmail process by simply cat'ing the process's proc entry to a regular file, as shown in Figure 1.

After the DUMP, top reports 0k SIZE and 0k SHARE (0% of total memory). You might be wondering why the dumpfile is 900k, instead of 1824k or 520k (1824k - 1340k = 520k). This is because read-only memory mapped files don't need to be dumped (so less than 1824k), but copy-on-write shared memory is dumped even if it has not been modified (and so COW pages become non-shared after a dump/restore).

Restoring is just as simple, we write a valid DUMP file to the process's procfs entry (# cat sendmail.dump > /proc/dump/740). Immediately after the restore, top shows us that non-shared (and COW) memory has been restored (Figure 4). The 4k (one page) of shared memory is from the signal handler code of sendmail handling SIGCONT. After sendmail starts handling mail we can see it start to load more shared, read-only code.

The %MEM has dropped from 1.4% to 1.0% after restart. This reduction comes from shared initialization code that is no longer used after restart. It is possible that, as more of sendmail's functionality is exercised, the memory usage will actually grow slightly beyond what it would have if a dump/restore had not taken place.

```
\# cat /proc/dump/740 $>$ /tmp/sendmail.dump \\
\# ls -al /tmp/sendmail.dump \\
-rw-r--r--    1 root     root       899480 Jun 29 18:57 /tmp/sendmail.dump
```

Figure 1: Sample use of dump

WHEN	SIZE	SHARE	STAT	%MEM
before DUMP	1824	1304	S	1.4
after DUMP	0	0	TW	0.0
after restore	880	4	S	0.6
after execution	1348	472	S	1.0

Table 4: sendmail, as reported by top