Adding Support for Dynamically-Linked Programs to Condor Checkpointing

Jim Basney
jbasney@cs.wisc.edu

February 28, 1996

1 Introduction

Condor is a distributed batch processing system for UNIX[1] developed at the University of Wisconsin. This system schedules jobs on idle workstations in a network, resulting in more efficient resource utilization. It is of primary importance in Condor to ensure that the owner of a workstation does not pay a penalty for adding his or her workstation to the Condor *pool* of workstations. So, a job must have the ability to immediately *vacate* a workstation when the owner begins to use it, and either *migrate* to another idle workstation or queue until one becomes idle.

To allow migrating jobs to make *progress*, Condor must be able to start the vacated job from where it left off. Condor does this by writing a *checkpoint* of the process's state before vacating. A checkpoint file contains the process's data and stack segments, as well as information about open files, pending signals, and CPU state. Condor gives a program the ability to checkpoint itself by providing a checkpointing library. Programs submitted to be run by the Condor system are re-linked (but not re-compiled) to include this library. See [1] for a detailed description of the Condor checkpointing library.

Until now, Condor has required that jobs with checkpointing support be linked statically. However when Sun Microsystems released Solaris,[2] Condor was forced to contend with dynamic linking because a number of Solaris libraries are incompatible with static linking (an example is the network services library, necessary for the use of sockets, which are used by Condor to transfer checkpoint files from one workstation to another).

This paper describes an implementation of process checkpointing for dynamically-linked programs on Solaris.

2 Solaris Processes

Solaris processes may contain *mapped* segments in their address space in addition to traditional text, data, and stack segments, as illustrated in Figure 1. This mapped segment facility is used to support dynamically-linked, shared-text libraries. When a dynamically-linked process is `fork()`ed, the necessary libraries are mapped into the process's address space. The first time a specific library call is made, a fault occurs and a link is created between the text symbol and the function definition in the library. Processes may also load in new libraries at any point during their execution. There is no guarantee that libraries will be mapped into the same addresses for multiple executions of the same program.

[1] UNIX is a registered trademark of UNIX System Laboratories, Inc.
[2] Sun Microsystems and Solaris are registered trademarks of Sun Microsystems, Inc.

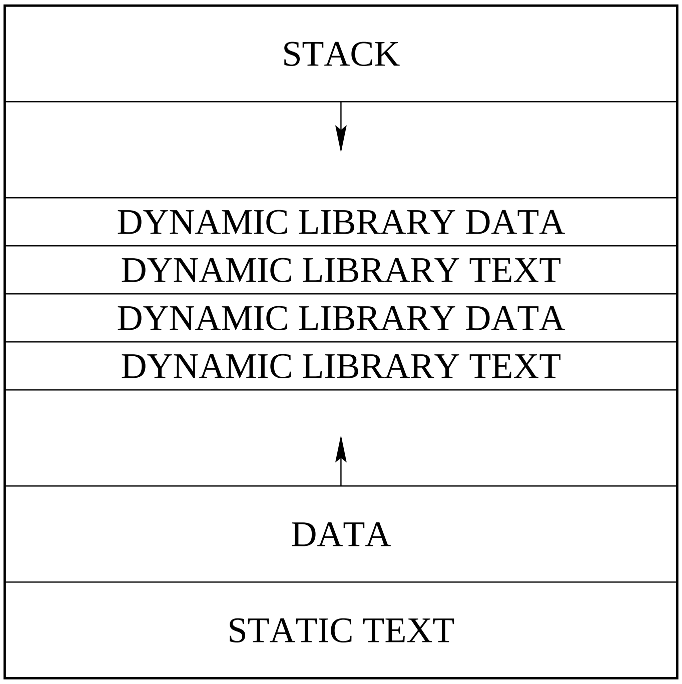

Figure 1: Solaris Process Address Space Layout

3 Required System Support

Condor must checkpoint the dynamic library data of the running process, in addition to the stack and data segments. The system must also ensure on restart that each dynamic library is mapped into the same area of virtual memory where it lived before the checkpoint, so that the dynamic links that have been set up are still valid. To do this, Condor must have support from the operating system to find all active segments, read the data in these segments, create new segments at arbitrary addresses, and write (restore) data into those new segments.

On Solaris, these facilities are provided by the `/proc` process file system, the `mmap()` function, and the `read()` and `write()` system calls. The following two sections describe hown Condor uses these facilities to add dynamic library support to the checkpoint and restart phases of the checkpoint library.

4 Checkpointing

To find all active segments, the checkpoint library uses the `ioctl()` interface to the `/proc` file system. In the `/proc` directory, there is a file for each running process, named by process-ID, which gives access to the process's memory contents. There is an `ioctl()` call to find the number of mapped memory segments in use by a given process and another to find information about each segment (virtual start address, size of segment in bytes, protection and attribute flags). Note that this call returns all process segments, including stack, data, and text segments. The `/proc` file system also appears on other Unix systems, so we hope that this approach is portable.

Condor must determine which segments are which, because some of the segments must be treated specially. The data and text segments are identified by comparing the addresses to the address of a static function (in the Condor checkpoint library) and the address of a global variable, respectively. The stack

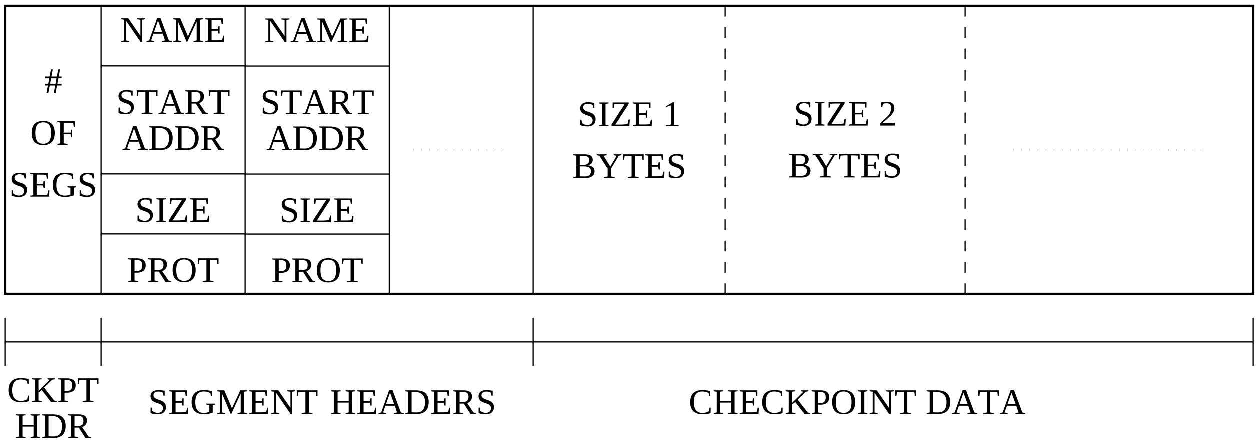

Figure 2: Checkpoint File Layout

segment(s) are identified by comparing the addresses to the stack pointer value and a system defined constant for the stack ending address. All other segments are assumed to contain dynamic library text or data.

Once all segments are identified, Condor saves all segments except the static text segment (because the text can be retrieved from the `a.out` file at restart). The `write()` system call is used to write bytes from addresses in memory to the file. Note that Condor saves the dynamic library text in addition to dynamic library data. There are a number of reasons for doing this. First, this is a simple way for Condor to ensure that the library text matches the library data on restart when migrating to a new machine which may have different versions of system libraries. If Condor did not ensure this, it is possible that library text would look for variables in incorrect locations. A second reason is that Condor must ensure that the library text is mapped back into the same location on restart, so that dynamic links are still valid. This behavior negates one of the benefits of dynamic linking (smaller executables) by increasing the size of the checkpoint file. However, we felt that this cost was acceptable compared to the complexity of comparing library versions on the checkpoint and restart machines and only moving libraries when necessary. It should at least be no worse than static checkpointing.

Figure 2 shows the Condor checkpoint file layout. Only one addition to this layout was made to support dynamic library checkpointing: a segment protection and attribute flag field in the segment header. The file layout was already general enough to handle an arbitrary number of saved segments.

5 Restart

On restart, Condor must restore all segments in the checkpoint file. The data segment is restored as in static checkpointing. Condor restores the stack segment last, and returns to the top of the stack as if returning from a signal handler (see [1] for a more detailed discussion of stack handling).

To restore the dynamic libraries, Condor first uses the `mmap()` function to allocate the necessary mapped segment in the virtual address space. This call maps the file `/dev/zero`[3] from the appropriate start address to end address, to provide an initialized segment. The protection and attribute flags for this new segment are set to be those saved in the checkpoint file, except that the segment always has write access enabled and the memory is always marked as private. Write access is necessary so that the saved bytes in the checkpoint

[3] `/dev/zero` is a special UNIX file which appears to contain an infinite number of zeros on reading and discards any data written to it

file can be written to the segment. The memory is not shared to ensure that our (possibly different) version of a system library does not interfere with other processes on the machine. Once the mapped segment is allocated, the `read()` system call is used to overwrite the memory with the saved bytes in the checkpoint file.

Note that the definitions for `mmap()` and `write()` must be forced to be static. Otherwise, we run into the possibility of overwriting the shared library text segment in which they are defined while we are in a call to either `mmap()` or `write()`, which can have disastrous consequences.

6 Conclusions

We have added dynamic library support to Condor checkpointing out of necessity, to allow us to support checkpointing on Solaris. Our checkpointing negates the benefits of dynamic, shared libraries (smaller executables and shared text) by instead storing library text in the checkpoint file and marking dynamic library segments as private on restart. However, we believe we have a general scheme of dynamic library checkpointing which will allow us to port Condor to any Unix systems which choose to follow the lead of Solaris and move to dynamic only distribution of system libraries.

References

[1] Todd Tannenbaum and Michael Litzkow. Checkpointing and migration of unix processes in the condor distributed processing system. *Dr. Dobbs Journal*, pages 40–48, Feb 1995.