Lab4 – Checkpointing Memory Data of a Process

Due: 11:59pm Wednesday, October 31, 2018

Instruction: In this lab, you need to implement partial functions of process checkpointing at the system level. As you may know, process checkpointing at the system level needs to save all the relevant runtime states of a process, including architectural, kernel-level, and user-level process information. So we can restart the target program from the saved checkpoint even if the original process terminates. Among all the states saved in a checkpoint, memory state is a very important component. This lab will focus on checkpointing memory data for a process. More specifically, the lab requires you to do the following three steps: (1) implement the basic function of checkpointing a memory range specified by a user program; (2) design an incremental checkpointing scheme; and (3) implement the incremental checkpointing scheme.

(1) **Basic checkpointing of memory (60 points):** As a first step, you need to implement a system call which takes a range of virtual addresses as the input parameters and saves the memory pages within the specified address range to a disk file. For example, the interface of the system call can be *int cp_range(void *start_addr, void *end_addr)*. Your code needs to walk through the VM areas and/or the page tables, and save the content of the page whose addresses are completely within the range of (*start_addr*, *end_addr*) to disk. Note that for each checkpoint (all memory data within the specified range), you can use a different file.

    To test your basic checkpointing scheme, you can write a test program similar to the following pseducode.

```
{
    int *array = malloc (size);
    // initialize the array;
    // take a checkpoint cp_range(array, array+size-1)   (e.g., filename: cp_P1)
    // modify data in array;
    // take a second checkpoint cp_range(array, array+size-1) (e.g., filename: cp_P2)
}
```

    After running the testing program, you can check the content of the two checkpoint files, i.e., cp_P1 and cp_P2, to see whether their content matches the array content, respectively. If yes, you successfully checkpoint the memory state of a process.

Otherwise, you need to spend more time on debugging it. After the initial test, you need to more rigorous test to see whether checkpointing a larger range works or not, e.g. , *cp_range (0x0, 0xBFFFFFFF)*. In the report, you need to document the design of the basic checkpointing scheme.

(2) Design of an incremental checkpointing scheme (30 points): As you may notice that the basic checkpointing scheme is not efficient if a process does not modify data in many memory pages between two consecutive checkpoints. An intuitive improvement is to take the checkpoint on the delta of the memory pages between two consecutive checkpoints -- the basis of incremental checkpointing. In particular, an incremental checkpointing scheme takes a full checkpoint for the first time and then only saves the memory pages that have been modified between two consecutive checkpoints. As the second step, you need to devise an incremental scheme at the kernel level. In the report, you need articulate: (1) how does your scheme achieve the incremental checkpointing? (2) what are the special cases your scheme need to handle? (3) how does your scheme handle these special cases? Note that in the report you should describe the design from the high level (conceptual) to the low level (kernel functions). For example, at the low level, what particular kernel functions need to be modified? And how? Basically, you need to understand the relevant kernel functions and implement the scheme on the paper.

(3) Implement the incremental checkpointing scheme (30 points): If you want to receive an additional 30 points, you need to implement your proposed scheme in Linux kernel. You can implement a different system call for the incremental checkpointing scheme, e.g., *int inc_cp_range (void *start, void *end).* Here, we assume that the series of invocation of *inc_cp_range(…)* will be provided with the same address range for the same process. Then you need to write a user-level test program to demonstrate that your scheme works. The test program is sketched as follows.

```
{
int *array = malloc (size);
// initialize the array;
// take a checkpoint inc_cp_range(array, array+size-1)   (e.g., filename: inc_cp_P1)
// modify data in array;
// take a second checkpoint inc_cp_range(array, array+size-1) (filename: inc_cp_P2)
}
```

After running the testing program, you can check the content of the two checkpoint files, i.e., inc_cp_P1 and inc_cp_P2, to see whether their content matches the initial array content and the modification, respectively. If yes, you successfully implement the

scheme. For measuring the improvement of space overhead, you can check and compare the sizes of second checkpoints under this scheme and the basic checkpointing scheme, i.e., inc_cp_P2 v.s. cp_P2.

[Not required for this lab] Want to explore more about both checkpionting schemes? You can use a real-world workload to measure the space and time overhead under both checkpointing schemes you implement.

New Submission Instruction: You need to submit the following files: (1) your report (explaining your design and implementation of the basic checkpointing scheme and the incremental checkpionting scheme, and discussing the changes you made to the kernel and reasons for such changes), (2) a kernel patch, (3) source code of user-level testing program/scripts, and (4) other relevant files that are needed for reproducing your demonstration without your help (e.g., a readme fie for setting up the demonstration).

You can generate a compressed tar ball file named lab4.tar.gz, which includes all the aforementioned files (do **NOT** include the compiled object files or other automatically generated files) and submit it on Carmen.

You can use the following command to make a diff file.

```
$> diff -Naur linux-2.6.9 linux-2.6.9lab# > lab#.diff
```

Before making the diff file, you should clean up the source tree by issuing the command “make mrproper” (You may want to read the file “linux-2.6.9lab#/Makefile” to get more information about several clean-up options).

Cleanup: Since we do not have enough disk space in each virtual machine, you need clean up the virtual machine after my evaluation for each of your lab solutions. First, delete the lab working directory at /usr/src/kernels/linux-2.6.9lab#. Then, delete the directory of installed modules at /lib/modules/linux-2.6.9lab#.