Operating Systems II

SoSe 2016

— Lab Assignment 2 —

Checkpointing

— Submission Email —

os-lab@deeds.informatik.tu-darmstadt.de

— Course Website —

https://www.deeds.informatik.tu-darmstadt.de/teaching/courses/ss2016/operating-systems-ii

Publication:	Mon, May 16, 2016
Submission:	Son, Jun 05, 2016 before 23:59
Testing:	Fri, Jun 10, 2016 (room A126)

Preface

The purpose of this lab is to further your understanding of how *checkpointing* mechanisms can be used to implement robust software. This lab builds on the concepts and techniques discussed in the corresponding lecture. For this lab, you will implement some of these techniques on a Linux system using the CRIU project.

Lab Environment

You should do the lab on your own machine. Make sure that you use a modern Linux distribution that includes recent (but not necessarily the latest) versions of the Linux kernel, the common command line and build tools. Most common distributions allow for easy installation and upgrade/downgrade of the required tools and software. Common choices for desktop Linux systems include Debian, Ubuntu, and Fedora. You need at least a 3.11 or later kernel.

As for the previous lab, you should have recent stable versions of GCC and Make as well as an editor of your choice available. You will need to do some additional preparatory setup work to install all the required libraries for this lab. A detailed description of these requirements is given as part of Task 1. Due to the nature of some of the tasks in this lab, they may be capable of crashing or destabilizing the system you are working on. For this reason, **you may want to consider working with a VM for this lab.**

Documentations and Finding Help

Most tools that have to be used in the lab come with a documentation. Make use of it! There are many forms of documentation, from simple plain text files to websites. In Linux systems, a common way to access documentations, especially for widely used standard tools and libraries, are the online reference manuals, the so-called man pages. Make extensive use of this feature. It is always a good idea to check the man pages if you are not sure how to use a certain tool or library. You can access the man pages with the following command:

```
man [section] <command-name>
```

Section 3 refers to C library functions and section 2 to system calls. The usage of section numbers is optional, but may be necessary when identical command names exist in different sections. For more details, try `man man` to access the man page for the man pages system ☺.

For general problems related to how tools work, how to accomplish a certain task using shell scripts, how to write Makefiles, etc., Stack Overflow and related online resources are a good place to start investigating: https://stackoverflow.com

Also, do not forget that online search engines and books from the library are your best friends!

Hints & Best Practice

We expect that you not only provide a *working and robust solution* for the tasks, but also adhere to *a consistent, structured, clean, and documented working and coding style*. Read the tasks thoroughly in order not to miss important details. Make sure that your solution matches the stated tasks. Keep your solution/code simple and do not lose sight of the actual tasks.

For tasks that require programming or scripting, you have to *implement proper error handling* mechanisms in your solution. For instance, function calls and program invocations may fail and their failure must be handled properly. Consult appropriate documentations to learn how error handling for the tools and libraries you use works. Often, problems with function calls and tools are indicated by magic return values (e.g. `-1` or `NULL`) or exit codes (e.g. `126`).

For programming tasks, make use of available build tools, compiler features, and linters. Make sure that your code compiles without any warnings by using the `-Wall` compiler switch during compilation (e.g., `gcc -Wall -o executable source.c`). If your code produces warnings during the testing session, we expect that you are able to explain each and every warning and justify why you did not fix it.

For building your solution, you should use the Make tool and write appropriate Makefiles instead of directly invoking the compiler. Automate your setup by writing shell scripts, e.g., for invoking a sequence of tools or gathering log files etc.

We expect each and every group member to know all the details of the tasks and your submitted group solution! During the testing session, every group member has to be able to explain, build, and execute the group solution. Moreover, each group member has to be able to answer questions related to the tasks and your solution.

Plagiarism & Fraud

We only accept *original solutions* of each individual group. Members of a group should work together very closely to solve the lab tasks. However, we do not support inter-group collaboration. If we notice that groups share the same solution, all involved groups will not get bonus points. Moreover, do not copy and paste code or other information from the Internet or other resources.

Be aware that there are a lot of tools that are able to automatically identify text and code that has been copied and pasted. We will make use of such tools to check whether your submitted solution is your original work.

Submission

Make sure to send your lab solution with all relevant files as tar archive via email to the address stated on the cover page before the deadline. You have to work in groups of 3 to 4 students. Do not submit individual solutions but send one email per group! ***In your submission email, include the names, matriculation numbers and email addresses of ALL group members.*** The testing date (see cover page) is mandatory for all group members. Therefore, ***indicate collisions with other TU courses on the testing date*** so that we can assign you a suitable time slot. Assigned time slots are fixed and cannot be changed. Students that do not appear for the assigned testing time will get no bonus points for that lab. During the test, we discuss details of your solutions with you to verify that you are the original authors and have a good understanding of your solution. The amount of bonus points you achieve depends on your performance in the testing session.

Miscellaneous

Please regularly check the course website as we publish up-to-date information and further updates there.

Questions regarding the lab can be sent to: os-lab@deeds.informatik.tu-darmstadt.de

Have Fun & Good Luck!

1) Setup Work

Over the course of this lab, you will be working with CRIU[1], a project to implement checkpointing functionality for Linux in userspace. For this first task, you will set up CRIU and familiarize yourself with the basic functionality of the `criu` tool. Note that some of the tasks you will be performing during this lab may require root access and not all the software you will be working with is entirely stable. For this reason, we recommend that you work in a virtual machine.

Prior to proceeding, you will need to install CRIU version 2.0 or later on your system. Most widely used Linux distributions provide packages for CRIU. If yours does not, you will have to build from source. Instructions on how to do this can be found on the CRIU website.

Once you have completed the installation, verify that it works correctly. You should have at least version 2.0 installed:

```
# criu --version
Version: 2.1
```

`criu` contains functionality to check if your kernel is configured correctly:

```
# sudo criu check
Looks good.
```

Finally, you need to check if the C headers for criu have been properly installed:

```
# echo '#include <criu/criu.h>' | cpp -H -o /dev/null 2>&1 | head -n1
. /usr/include/criu/criu.h
```

At this point, you should be able to checkpoint and restore processes using the `criu` command line utility, and you are encouraged to give that a try (for example, by following the instructions from the CRIU tutorial). You should also be able to compile C programs using the `criu.h` header file. Try out this minimal example to confirm that it works:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <criu/criu.h>

int main() {
    mkdir("dump", 0700);
    int fd = open(“dump”, O_DIRECTORY);
    criu_init_opts();
    criu_set_images_dir_fd(fd);
    criu_set_service_address("criu_service.socket");
    criu_set_service_comm(CRIU_COMM_SK);
    criu_check();
    return 0;
}
```

[1] https://criu.org

You should be able to compile and run the example like this:

```
# sudo criu service -d
# gcc -Wall -Wextra -o test test.c -lcriu
# sudo ./test
```

The first command starts the criu service as a background daemon. The example application first creates a directory for checkpointing information, then initializes and configures CRIU. Make sure you understand what each part of the example application does. You can consult the man pages, `criu.h` and the CRIU website as necessary. Do note that some of the commands may require root privileges.

Note that the given example program largely omits checking return values and error codes for the sake of brevity. The code you write is expected to contain these checks.

Task 1.1

> Set up and test CRIU as described above.
>
> Write a Makefile containing the customary default and clean targets that you can adapt or extend for the following tasks.
>
> Add the required error checking to the given example program.

2) Basic Checkpointing

Now that you have taken care of the basic setup work, you will use CRIU to implement a small program that utilizes checkpointing to recover from crashes. Your program should consist of a monitor and a worker component. The monitor should spawn the worker process. The worker component should read inputs from stdin and process them. The following inputs should be understood:

- Any positive integer N: Generate N random numbers in [0, N] and write them to stdout, one per line. Use `rand` (`man 3 rand`) to generate the random numbers.
- Any negative integer N: Exit with return value N.
- Any other input: Exit cleanly and write all generated numbers to a file, one per line. Note that the output should be written to file all at once on program exit.

Usage of your program could look like this:

```
# ./task2 out.txt
$ 5
2
3
1
1
4
$ x
Writing output to out.txt... Done.
#
```

Task 2.1

Implement the program described above. The worker process should behave as described, the monitor should restart the worker from scratch whenever it exits with a return value other than 0.

As in the example above, the name of the output file should be passed as a command line argument.

Task 2.2

In this task, you will extend your program from Task 2.1 with periodic checkpointing functionality. Your program should accept a second command line argument, specifying an interval *i* (in seconds). After every input, your worker should check if at least *i* seconds have passed since the last checkpoint was taken, and if so, checkpoint itself. The monitor should be adapted to restart the worker process from the last checkpoint instead of from scratch.

For this task, you may need the following functions from `criu.h`:

- `criu_dump`
- `criu_restore_child`
- `criu_set_shell_job`
- `criu_leave_running`

Task 2.3

At this point, the checkpointing functionality is controlled entirely by your worker and checkpointing is only possible after an input has been read. This is not particularly problematic in this small example but would be a severe limitation in case the worker process had to do any complex computations. In such a scenario, having to design the worker process in such a way that it would be capable of independently pausing its computations to take a checkpoint every *i* seconds would be highly undesirable. For this reason, you should move the responsibility for periodically triggering checkpointing to the monitor process.

In this task, you should extend the monitor to periodically send `SIGUSR1` to the worker process. The worker should respond to this signal by taking a new checkpoint. You may want to take a look at `man 7 signal` and `man 2 sigaction` for this task. `man kill` may also be useful as a way to manually send `SIGUSR1` to the worker for testing purposes.

Task 2.4

Write a test script that exercises the checkpointing functionality. If you have implemented the worker process correctly, you can induce worker terminations that look like crashes to the monitor process by passing negative integers to stdin. Your test script should check if the worker process is restored correctly following these inputs.

Your test script should also contain a test in which the worker is killed by way of an external signal (such as `SIGKILL`) to see if the crash detection in the monitor and the restoration from the last checkpoint work correctly in that case as well.

3) Custom Checkpointing

In the previous task, you utilized an existing tool for whole-process checkpointing. Since the small example program you implemented does not have a particularly extensive or complicated internal state, this level of complexity is not strictly necessary. For this task, you will implement a custom checkpointing mechanism for your example program without relying on CRIU.

Question 3.1

> Your program should produce *exactly the same output* after checkpoint restoration as during the initial run. What information does a checkpoint of your worker need to contain for this purpose? Are any changes to the worker required? Justify your answers!

Task 3.1

> Extend your program with 2 additional functions, `checkpoint` and `restore`.
>
> `checkpoint` should write the internal state of your worker to a `checkpoint.dat` file in the current working directory. `restore` should check if a `checkpoint.dat` file exists in the current working directory, and if so, read it and restore the worker state.
>
> Adapt your program to use the new functions instead of the CRIU checkpointing functions. Use your test script from Task 2.4 to test your implementation.
>
> You may want to take a look at `man 3 fread`.

Task 3.2

> The simple checkpointing mechanism you implemented in Task 3.1 has the drawback that it only keeps the last checkpoint around. Should the checkpoint.dat file be corrupted or should the state at the time the checkpoint was taken already contain an error, restoration would be impossible. In this task, you will address this shortcoming with a simple extension to your checkpointing mechanism. Instead of always using the `checkpoint.dat` file, checkpoints should be stored in timestamped files `checkpoint.timestamp.dat`, where `timestamp` is the Unix time when the checkpoint was taken.
>
> You should also extend your restoration mechanism to automatically use the most recent checkpoint. Extend the monitor to clean up old checkpoints when it is started.

Question 3.2

> How does your implementation react to broken checkpoint files? Try overwriting a `checkpoint.timestamp.dat` with random data and see how your program behaves. Can you come up with ways to check if the data restored from a checkpoint file is internally consistent? Can you think of a way to check if a checkpoint has been corrupted, even if the resulting state may still be internally consistent?

Bonus Task 3.3

> Implement a simple checksum of your choice for your checkpointing mechanism. Extend your restore functionality to detect corrupted checkpoint data. The resulting restore function should restore the latest non-corrupted checkpoint.