Operating Systems II

SoSe 2016

— Lab Assignment 1 —

Robust Programming 1: Process Pairs

— 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, Apr 25, 2016
Submission:	Son, May 15, 2016 before 23:59
Testing:	Fri, May 20, 2016 (room A126)

Preface

The purpose of this lab is to deepen the understanding of how process pairs can be used as a robust programming mechanism to provide continual service in the face of individual process failures. This lab builds upon the concepts and techniques discussed in the corresponding lectures. You will implement some of the presented techniques on a Linux system using the *C programming language (C99)* and gain practical experience in robust programming.

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 should have installed stable versions of the following basic development tools: GCC (4.9 or 5.0), Make (3.8 or 4.0), development files for libc, and a sensible editor (e.g. Vim or Emacs).

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) Process Management & Build Setup

Process management includes the creation of new processes, program execution, and process termination. A process can create a new child process by calling the `fork` function, see `man 2 fork`. The `fork` function is called once but returns twice. The main difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID (PID) of the new child. Both the child and the parent continue their execution with the instruction that follows the `fork` call. The child is a copy of the parent, i.e., the child gets a copy of the parent's data (heap and stack) and they share the same text segment.

A parent process can wait for the termination of its child processes by calling the `waitpid` function, see `man 2 waitpid`. The `waitpid` function blocks the caller until a child process terminates. If a child has already terminated, `waitpid` returns immediately with that child's status.

Usually, a build utility is used to manage build dependencies, compilation, testing, and related activities. In UNIX environments, the Make tool is a common choice. A Makefile describes a set of rules and dependencies that Make needs to execute in order to build a program. Have a look at `man 1 make` for details on the tool itself. Online tutorials such as the following two can help you to learn how to write simple Makefiles to automate your build process.

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

http://mrbook.org/blog/tutorials/make/

Task 1.1

Note that this task is best done in parallel with Task 1.2.

Write a Makefile to build, execute, and test the program you write in Task 1.2. Your Makefile should provide two build targets: `prog` and `exec` such that invoking `make prog` compiles your program and `make exec` starts your program. Note that the `exec` target should depend on the `prog` target, i.e., the program has to be built implicitly if this was not done explicitly before. Use GCC with the following flags for compiling your program:

```
--std=c99 -Wall –Werror -O1
```

Also add the target `test` to your Makefile which executes your program 5 times, counts how often your program terminated with and without error, and logs all relevant output including the test results (error/success counts) to a log file. Note that you may want to write a shell script that is invoked from your Makefile in order to have all the power of your shell available. Do not forget that the `test` target also depends on the `prog` target as you can only run tests on a program that has already been built.

Note that a well written Makefile always includes a `clean` target to remove all generated build artifacts such as the built program executable!

Task 1.2

Implement a program that spawns a child process and waits for its child to finish execution. Make use of the `fork` and `waitpid` functions. After spawning the child, each process has to print a text message that states its own process id (PID) and whether it is the parent or child process. Moreover, the parent has to print messages indicating the creation and termination of the child including the child's PID and exit status. See `man 3 stdio` for an overview of I/O functions that you should use.

Use the `usleep` function to artificially slow down your program. Have a look at `man 3 usleep` to learn how to use the function. Add a 250 ms (ms = millisecond) delay right after the `fork` call to slow down both the parent and the child process. Add another 500 ms delay in the child process after printing the child message to prolong its execution time.

Do not forget to implement proper error handling as this is very important for real world programs. You have to at least print meaningful error messages and terminate the program with a proper exit code! See the man pages (section 3) of the following functions: `exit, error, perror, errno`. The parent process should terminate with an error in case the child could not be created or terminated with an error.

Task 1.3

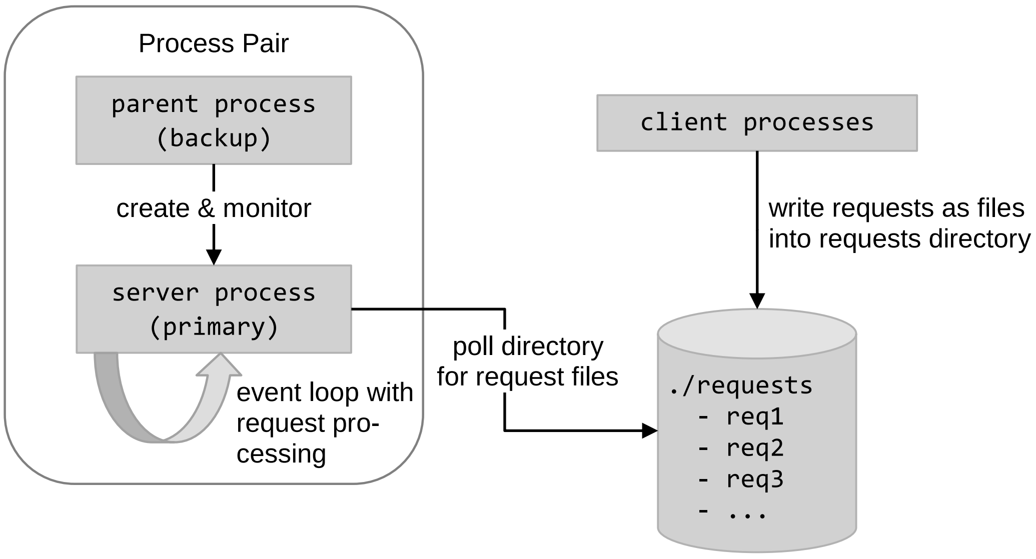
Test if your Makefile works as expected by invoking all its targets.

In order to test the `test` target, modify your program to include a 50 % change that the child process terminates with an error. See `man 3 rand` to learn about a useful function for this.

Include the log files from your test in the submission.

2) Simple Process Pairs

We want to develop a simple (almost) stateless server that serves requests received as files from the filesystem. To provide continual service in the face of possible process crashes, we make use of the Process Pairs concept in the variation where one process responds to requests whereas the other serves as backup and there is no state transfer between the two (Primary/Backup without state transfer). The following figure illustrates the overall architecture.



Arbitrary client processes can send requests by creating files in the `requests` directory; each file represents a request and must have a unique name in order not to overwrite already existing requests. Clients operate independently from the server (or backup) process, i.e., they add new requests at any point in time in parallel to the server's execution. Clients may also cancel requests that the server has not started processing by removing them from the `requests` directory.

The server process regularly polls the `requests` folder and iterates over all the requests. Each request has to be processed exactly once. For this purpose, the server opens a request file, does the processing, and deletes and closes the request file once the processing is complete. This request handling is illustrated by the following pseudo code snippet:

```
// server request loop
loop_forever {
  dir = open_directory("requests")
  for_each regular_file file_name in dir {
    fil = open_file(filename)
    req = read_request(fil)
    do_processing(req)
    delete_file(fil)
    close_file(fil)
  }
}
```

The server process is spawned by the backup process, which sits in the background and waits. In case the server process crashes, the backup process tries to create a new one. Only if this attempt fails too often, the backup process serves requests as a last resort to provide continual service. This process pair logic is illustrated in the following pseudo code snippet:

```
// process pair logic
loop {
  num_forks += 1 // initialized to 0
  p = fork_process()
  if in_child()
    return // and start request processing
  if not in_child() && p == FAILURE && num_forks > MAX_FORKS
    return // graceful degrade and start request processing w/o backup
  wait_for_crash(p)
}
```

Task 2.1

As first step to implementing the above described architecture, you have to develop the main program and the Makefile for building and testing it. Develop your Makefile according to the description in Task 1.1. The Makefile and the tests have to evolve along with your implementation. Note that you need to adapt and extend your `test` target for this and the following tasks, e.g., creating and canceling request.

Develop the above described process pair logic (see pseudo code) and encapsulate it in a function called `backup`. The `backup` function should only include the actual process management and not the request processing. The maximum number of server spawning attempts (see `num_forks` and `MAX_FORKS` in the pseudo code) has to be configurable via a function parameter. Note that the above pseudo code for `backup` is incomplete; it especially lacks proper error handling and other details that needs to be filled in by you in your implementation.

Design and implement your main program that uses the `backup` function to implement process pair functionality. The first thing your program should do after startup is printing a nice welcome message. After that, it should process command line arguments and then call `backup` with proper arguments to create the process pair. The request processing logic is part of the following task. So, instead of processing requests, the spawned server process should just print the message `"server %d\n"`, where `%d` is the server's PID, and terminate. The maximum number of server spawning attempts has to be configurable via a command line option `"-n N"`, where the valid range for `N` is 1 – 50 and its default value is 5.

Note that proper error handling, including printing error messages, is paramount in this lab, see man page references in section 1. For command line parameter handling, you may want to have a look at `man 3 getopt`.

Task 2.2

Extend your program from Task 2.1 and add the request logic as drafted in the pseudo code above. Encapsulate the server request loop in a function called `server`. Call this function from your main program instead of the server message printing. Have a look at the following man pages (section in brackets) to learn how to access and manipulate directories and files: `readdir(3), fopen(3), fclose(3), unlink(2)`. Note that you probably want to use `-D_BSD_SOURCE` as additional GCC flag in order to conveniently use the `d_type` field in `struct dirent` (see `man 3 readdir`) to distinguish regular files from other entry types.

For simplicity, all the request processing only involves file name processing, so you do not perform actual file I/O. The request processing should work as follows (`do_processing` in the pseudo code): The server prints `"server[%d] req: %s\n"`, where `%d` is the server process' PID and `%s` is the file name of the current request, and waits for 500 ms (see `usleep` in section 1).

Task 2.3

In this task, you have to test your program. To this end, extend your main program with an additional command line option `"-f N"` that configures the chance (in percent) that the server terminates with an error during request processing, before the request message is printed but after the request file was opened. The valid range for this `N` is 1 – 100 and its default value is 0. A value of 50 means that about every second request processing attempt causes a server failure. See `man 3 rand` to learn about a useful function for this. The failure chance should only be applied for requests that have the string `"fail"` in their file names, see `man 3 strstr`.

Extend the `test` target of your Makefile to include the generation and deletion of request files while your main program is executing. Start your main program with `-n 10 -f 50` and generate at least 100 requests from which about 40 have `"fail"` in their file name. Also do some manual tests and arbitrarily kill your server process (the `htop` tool may be handy). Continue by killing your backup (aka parent) process. Note that `touch` (`man 1 touch`) may be of use for creating empty request files.

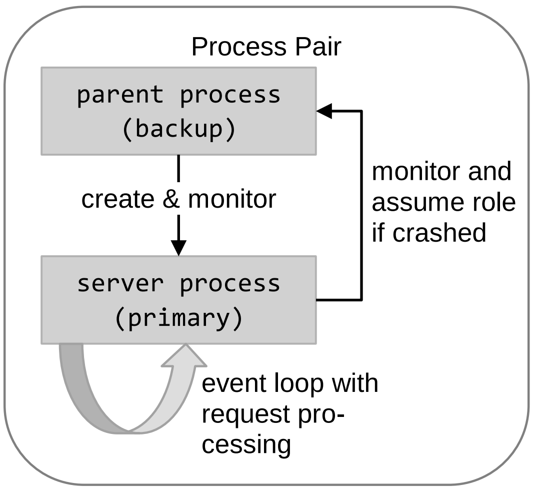
Observe your program's behavior during the tests and include the log files in your submission.

Question 2.1

Where do you see problems with the presented architecture and your implementation? Which kinds of errors/failures does your "service" survive and which not?

3) Process Pairs with Links

The architecture that you implemented in the previous section has some shortcomings. In this section, we want to improve the architecture according to the following illustration. The other parts are left untouched.



In this enhanced architecture, the `server` process monitors the `backup` process. In case the server detects a backup process crash, it clears its process state, assumes the role of the backup and spawns a new server process.

Task 3.1

Extend your program from Task 2.3 and implement the above described improvement. A simple option for implementing the monitoring approach inside the server process is to regularly poll the backup process to see if it still exists. For this purpose, write a new function called `backup_terminated` that is called as part of the request processing loop, e.g., before a new request file is opened. This new function should use a pipe between the parent and the child process to check if the parent has the pipe's read end open. As long as the read end is open, the child can write to it, but once the read end is closed, the child gets an error. Note that this usually only works if the signal `SIGPIPE` is ignored, see `man 2 signal`. Moreover, have a look at the following man pages (section 2) for details on very useful functions in this context: `pipe, write`.

Once `backup_terminated` detects that the backup process has terminated, it has to change the server process into a backup process. A simple and clean way to achieve this is to replace the current server process image by a fresh instance of your program. This has the same effects as if you kill and restart your whole program. See `man 3 exec` for details on how to achieve this. Note that the original command line options have to survive this process image replacement.

Task 3.2

Test your new program version from the previous task similar to the tests in Task 2.3. Extend the `test` target in the Makefile if need be.

Observe your program's behavior during the tests and include the log files in your submission.

Question 3.1

Where do you see problems with your new program version? Which kinds of errors/failures does your “service” survive now? Is there some kind of maximum time frame that your “service” can run before it must be restarted?

How would you improve the architecture and/or your implementation for productive use?

Task 3* (Bonus For More Fun)

Extend your program to use a configurable number of server processes that process requests in parallel. A command line option `-s N` should be used to set this number upon program start. Your new program version should be able to survive crashes of any server process (and respawn them) as well as crashes of the parent (backup?) process (recreate it). Since this is a bonus task for more fun, you can freely design your program improvement.

Do not forget to also extend your tests to check if your new functionality is working as expected.