EX5 - Containers

Due: 31/07/2023

The purpose of this assignment is to provide you with hands-on experience working with containers and virtual machines (VMs) to gain an understanding of the different levels of isolation they offer. The assignment includes theoretical questions to explore the differences between containers and VMs, as well as a coding assignment to implement communication between a server and client using the file system, shared memory and sockets.

Part 1: Theoretical Questions (20 pts)

The following questions are here to help you understand the material, not to trick you. Please provide straightforward answers.
All questions are worth an equal amount of points.

1. What is a use case of Linux namespaces? What is their relationship with containers?
2. Present two differences between containers and VMs.
3. Can containers communicate via shared memory? If not, what changes would be necessary to enable shared memory communication?
4. Can containers connect via sockets? explain in two or three sentences.

Part 2: Virtualization and IPC (80 pts)

Introduction

The purpose of this assignment is for you to develop a better understanding of the different levels of isolation that VMs and containers offer in comparison to each other, and compared to running directly on the machine. As such, you will construct a communications system between *servers* and a *client*. The main objective of the system is for the client to detect and identify the environment that each server is running on.

The Server

You are required to implement the server as a library with the following functions:

void **start_communication**(const server_setup_information& *setup_info*, live_server_info& *server*);

- The **start_communication** function should create a socket to listen on and allocate a shared memory segment, using the information provided with the struct given as an argument server_setup_information& *setup_info*.
- The socket should listen to the IP of the machine running this function, on the *port* given with *setup_info.*
- The shared memory segment should be identified by the provided *shm_pathname* and *shm_proj_id* given with *setup_info*.
- The function should also fill the struct given as an argument live_server_info& *server* with the socket file descriptor (*socket_fd*) and the shared memory segment id (*shmid*).

void **create_info_file**(const server_setup_information& *setup_info*, live_server_info& *server*);

- The **create_info_file** function should create a file with the name *info_file_name* provided in the given in server_setup_information& *setup_info*.
- The file should be saved under the directory *info_file_directory* also provided in *setup_info*.
- This file should contain the relevant information about the shared memory segment and the socket from s*etup_inf*. The content of this file should be in the following format:

<IP>
<port>
<shm_pathname>
<shm_proj_id>

for example, for a socket listening on the IP address 1.1.1.1 on port 1234, and a shared memory segment identified with the path name “abcd” and proj_id 10, the info file’s content should be:

1.1.1.1
1234
abcd
10

- This function should save the created file path inside the struct live_server_info& *server* given as an argument under *info_file_path*.

int **get_connection**(const live_server_info& *server*);

- The **get_connection** function should try to establish a connection with a client.
- You should set a timeout of 120 seconds for the socket to get a connection from the client. Use the select() syscall with the server socket added into the *readfds* set argument, and the required timeout set in the *timeout* argument. You should call the select() syscall right before calling the accept() syscall.
- Return -1 if the connection failed, 0 on success.

void **write_to_socket**(const live_server_info& *server*, const std::string& *msg*);

- The **write_to_socket** function should send the specified *msg* via the server socket whose information is provided inside the struct *server* given as an argument.
- You may assume the socket whose fd is specified in *server* is initialized and connected to a client.
- You may assume the given *msg* does not contain any space characters.

void **write_to_shm**(const live_server_info& *server*, const std::string& *msg*);

- The **write_to_shm** function should write the specified message to the shared memory segment whose information is provided inside the given parameter *server*.

- You may assume the shared memory segment whose id is specified in *server* is already allocated.
- You may assume the given *msg* does not contain any space characters.

void **shutdown**(const live_server_info& *server*);

- The **shutdown** function should handle the cleanup and shutdown of the server, including the closing of the server socket, and deallocating the shared memory segment, both provided in the given parameter *server*.
- It should also delete the info file provided by *server* as well.

void **run**(const server_setup_information& *setup_info*, const std::string& *shm_msg*, const std::string& *socket_msg*);

The **run** function should do the following steps:

1. Set up a server by creating a socket and allocating a shared memory segment.
2. Write all the information regarding the socked and the shared memory segment into the info file.
3. Write the given *shm_msg* into the shared memory segment.
4. Wait for a client to connect. If no connection was established until the timeout is reached, skip on step 5.
5. Send the given *socket_msg* to the client.
6. Shutdown the server by closing the socket and deallocating the shared memory segment.

This API, including the struct server_setup_information, is contained in the file server.h. The struct live_server_info is contained in the file globals.h.

The Client

You are required to implement the client as a library with the following functions:

int **count_servers**(const std::string& *client_files_directory*, std::vector<live_server_info>& *servers*);

- The **count_servers** function should read the files at the specified directory, which was written by the servers. It should then attempt to connect to the servers using both shared memory and socket methods.

- You may assume that all the files located under *client_files_directory* are in the correct format, as specified above in **create_info_file**.
- Do not disconnect the sockets, if successfully connected!
- Do not deallocate the shared memory segment, if successfully allocated!
- **Most Important: it is possible that connecting into the server socket, or attaching into the shared memory segment won't succeed.** Think about what it means regarding the environment on which the server is running.
- For every file inside *client_files_directory*, you should add a *live_server_info* object to the *servers* vector.
- You should set a timeout of 5 seconds for the socket to establish a connection to each server. Use the select() syscall with the client socket added into the *writefds* set argument, and the required timeout set in the *timeout* argument. You should call the select() syscall right before calling the connect() syscall.
- If no shared memory segment is found with the given pathname and proj_id, you should store the value -1 inside *shmid*. Make sure that you don't add the flag `IPC_CREAT` when calling *shmget()* to check if the shared memory segment is allocated.
- The order of the servers inside *servers* should match the order of the files under the directory *client_files_directory* when sorted alphabetically.
- The function should return the number of servers found.

void **print_server_infos**(const std::vector<live_server_info>& *servers*);

- The **print_server_infos** function should read any messages from the servers via shared memory segments and sockets and print the number of servers running on each environment (host, VM and container) and in total, along with any received messages.
- Results should be printed in the following format:

  Total Servers: <num_servers>
  Host: <total_running_on_host>
  Container: <total_running_on_container>
  VM: <total_running_on_vm>
  Messages: <msg_1> <msg_2> … <msg_n>

- You may assume that each server found will send exactly one message via each communication method (if socket connected successfully or shared memory segment allocated successfully).

- If messages are received via both a shared memory segment and a socket, the message from the shared memory segment should be printed first.
- The order of the messages should match the order of the servers provided in the given vector *servers*.

void **get_message_from_socket**(live_server_info &*server*, std::string& *msg*);

- The **get_message_from_socket** function reads a message from the socket whose information is stored in *server*. The message should be saved in *msg*.
- You may assume the socket whose fd is specified in *server* is initialized and connected to a server.
- You may assume the given *msg* does not contain any space characters.

void **get_message_from_shm**(live_server_info &*server*, std::string& *msg*);

- The **get_message_from_shm** function reads a message from the shared memory segment whose information is stored in *server*. The message should be saved in *msg*.
- You may assume the shared memory segment whose *shmid* is specified in *server* is allocated.
- You may assume the given *msg* does not contain any space characters.

void **disconnect**(const std::vector<live_server_info>& *servers*);

- The **disconnect** function should disconnect from all the servers in the vector by disconnecting from their sockets and shared memory segments
- This function should **not** deallocate the shared memory segments. It is the responsibility of the servers to deallocate the shared memory segments.

void **run**(const std::string& *client_files_directory*);

The **run** function should do the following steps:
1. Try to connect to all the servers, whose information can be found inside the directory specified by the argument *client_files_directory*.
2. Print the summary on all servers, as well as any msg sent by the servers, using the format defined by **print_server_infos**.
3. Disconnect from all the servers.

Note that the *live_server_info* struct contains information necessary for connecting to a server.

This API is contained in the file client.h

Workflow:

Below we present a sketch of the workflow:

1. Start the servers in their respective environments. Ensure that the directory intended for writing is properly bound if running in a container or VM.
2. Create an info file and write the necessary information to it. This file will contain details required for communication.
3. Start the client.
4. Attempt to connect the client to the servers based on the information provided in the servers' info files.
5. Print the results, including any messages received from successfully connected shared memory and sockets.

Note that this is a high-level overview of the workflow.

Notes

- Make sure to appropriately handle errors and exceptions in your code.
- Use the provided function signatures and ensure your implementation aligns with the specified requirements.
- The following functions are likely to be of assistance with shared memory
  ftok, shmget, shmat, shmdt, shmctl.
- **Important note:** Unlike the memory allocated on the heap by the process, which will be freed in any case by the operating system at the end of the process running, a shared memory segment is **not** freed by the operating system, and therefore **it is your responsibility** of deallocating the shared memory segment yourself in any case!
- When working from the CS HUJI machines use the following command from the terminal to run a VM:

  rundeb11 -bind /cs/usr/<cs_user>/<path>/<to>/<dir>

  <path>/<to>/<dir> should be replaced with a relative path to the directory where you will save the server info files relative to your user home directory. You may also add the -serial flag to run only a terminal of the VM, which should suffice for this exercise, and to reduce the memory requirements

of the VM.
You should not add any other flag when using rundeb11

- When working from the CS HUJI machines use the following commands from the terminal to run a Docker (container):

  module load singularity
  singularity build dgcc.simg docker://gcc
  singularity run dgcc.simg --bind /cs/usr/<cs_user>/<path>/<to>/<dir>

  <path>/<to>/<dir> should be replaced with a relative path to the directory where you will save the server info files relative to your user home directory. **You should not add any other flag when using singularity.**

Simplifying Assumptions

You are allowed to assume the following:

1. The client only runs after all servers have finished starting up.
2. The servers are responsible for freeing the shared memory, and will do so after the client has accessed it.
3. You can assume that all files in the *client_files_directory* were created by servers that are currently running.
4. The client is running directly on the host machine.
5. **write_to_socket()** will only be called after the client has connected
6. **write_to_shm()** will only be called after the shared memory segment is allocated

Error Messages

The following error message should be emitted to *stderr*.

When a system call fails (such as memory allocation) you should print a **single line** in the following format:
"system error: *text*\n"
Where *text* is a description of the error. You should also free any resources allocated before exiting, such as the socket connection, the shared memory segment, and removing the info file. After all the resources are freed, exit with exit(1).

Background reading and Resources

1. https://wiki.cs.huji.ac.il/wiki/Private_Virtual_Machines
2. https://wiki.cs.huji.ac.il/wiki/Containers

Submission

Submit a tar file named ex5.tar that contains:

1. README file built according to the course guidelines.
2. All relevant source code files
3. Answers to the questions from section 1 in a file named ANSWERS.txt
4. Makefile - your makefile should generate two **static** libraries named: *libclient.a libserver.a* when running 'make' with no arguments.

Make sure that the tar file can be extracted and that the extracted files compile.

Guidelines

1. **Read the course guidelines**.
2. Design your program carefully before you start writing it. Pay special attention to choosing suitable data structures.
3. Always check the return value of the system calls you use.
4. Test your code thoroughly - write test programs and cross test programs with other groups.
5. During development, use asserts and debug printouts, but make sure to remove all asserts and any debug output from the library before submission.

Good luck!